Pipes and Redirection

It just never stops

Benjamin Brewster

Adapted from slides by Jon Herlocker, OSU

Sharing open files

- When you exec, you replace your current process with a new one
- But your files are still open
  - This may not be what you want
- Are there other ways to share files?

Sharing open files

- Preventing open files from being shared across an exec
  - close-on-exec

- I/O redirection
  - Redirecting input to and from files on disk
  - Pipes: redirecting input and output between different processes
    - akin to command line piping, but not the same

Close on exec

- Tells the kernel to close open files on exec
  - why would we want to do this?

- Open files are "inherited" by child processes
  - Thus the file pointer is shared (!)
  - Security/sensitive data

Close on exec

- You set the "close-on-exec" flag for every file descriptor you do not want to share

- The "close-on-exec" flag is inherited through fork
  - So you can set the close-on-exec flag in the parent and if the child does an exec() the file will be closed, as well

Normally

Process A → fork() → Process A_1

File descriptor 5
open

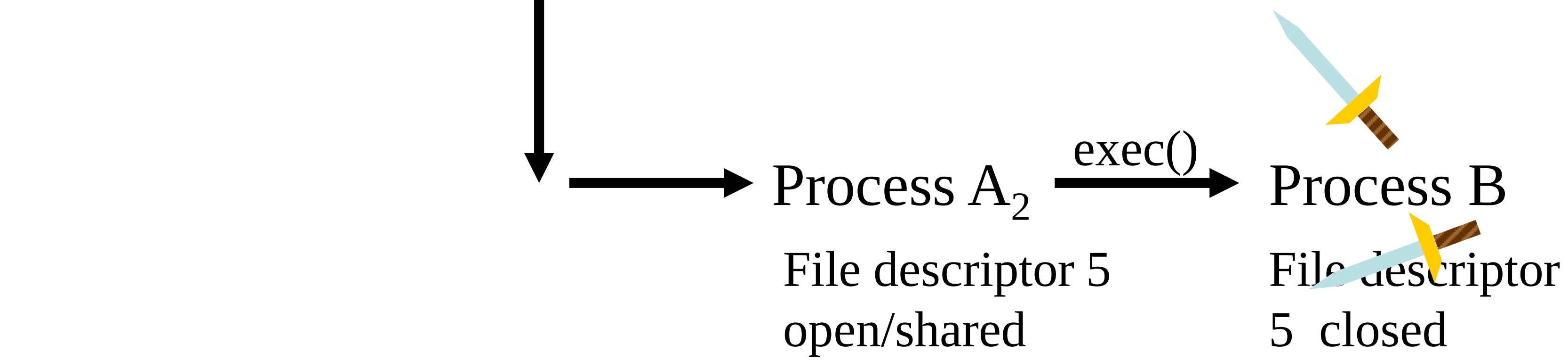
Process A_1
File descriptor 5
open/shared

Process A_2 $\xrightarrow{\text{exec()}}$ Process B

Process A_2
File descriptor 5
open/shared

Process B
File descriptor 5
open/shared

With close on exec



Close on exec example

```
#include <fcntl.h>
...
int fd;
fd = open("file", O_RDONLY);
…
fcntl(fd, FD_SETFD, 1);
…
exec...
```

It doesn’t look like much, but that’s close on exec

I/O redirection

- We saw I/O redirection in the shell
  - ls > file
  - stats < file1
  - cat longfile | more
  - find . -name paper -print 2> /dev/null
  - echo "an error occurred" 1>&2
- I/O redirection is possible *because* open files are shared across fork() and exec()

Important background

- The kernel opens stdin, stdout, and stderr automatically for every process created
- File descriptor 0 is stdin
- File descriptor 1 is stdout
- File descriptor 2 is stderr
- They default to reading and writing to the terminal

I/O redirection

- The trick: you can change where the standard I/O streams are coming from and/or going *after* the fork *but before* the exec

Redirecting stdout

Process starts

↓

opens new output file - fd = 3

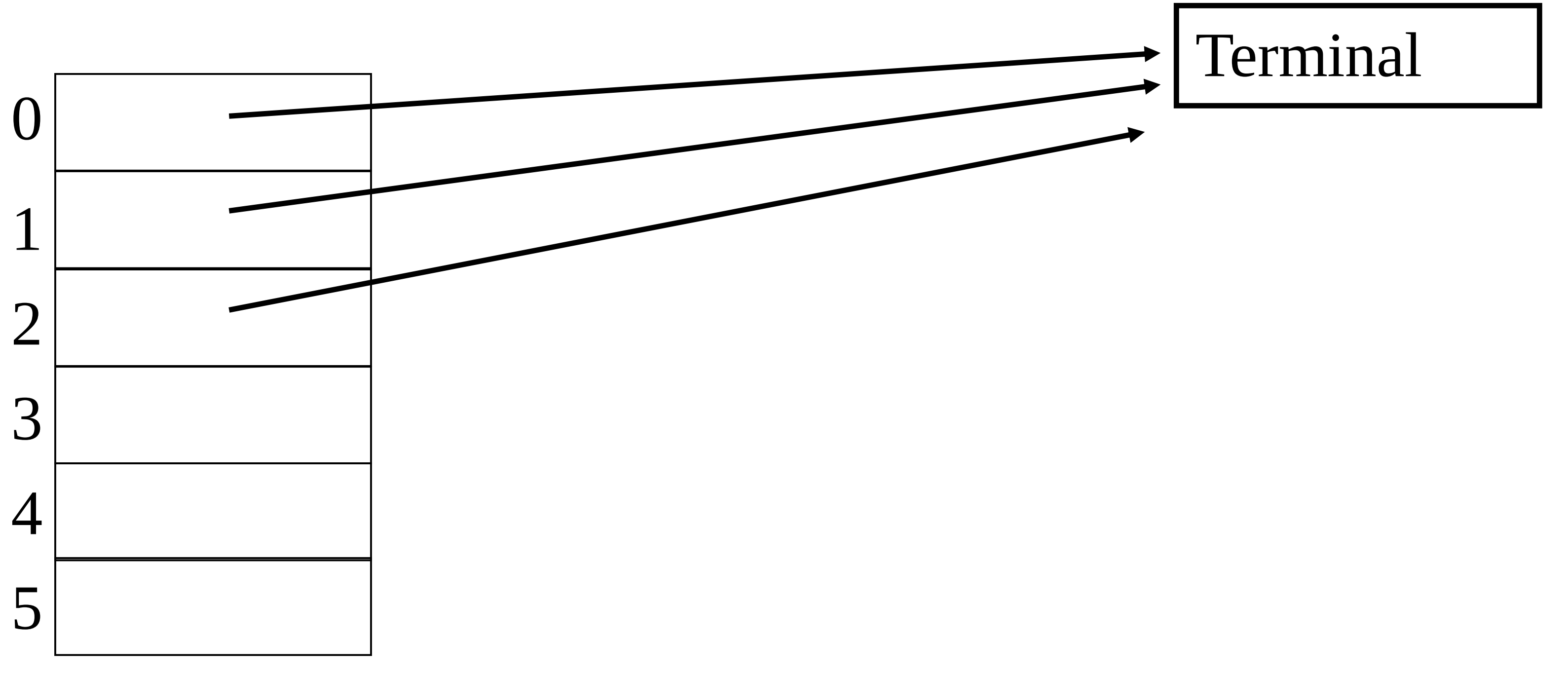
↓

Duplicates fd 3 into fd 1
(overwrites fd 1)

↓

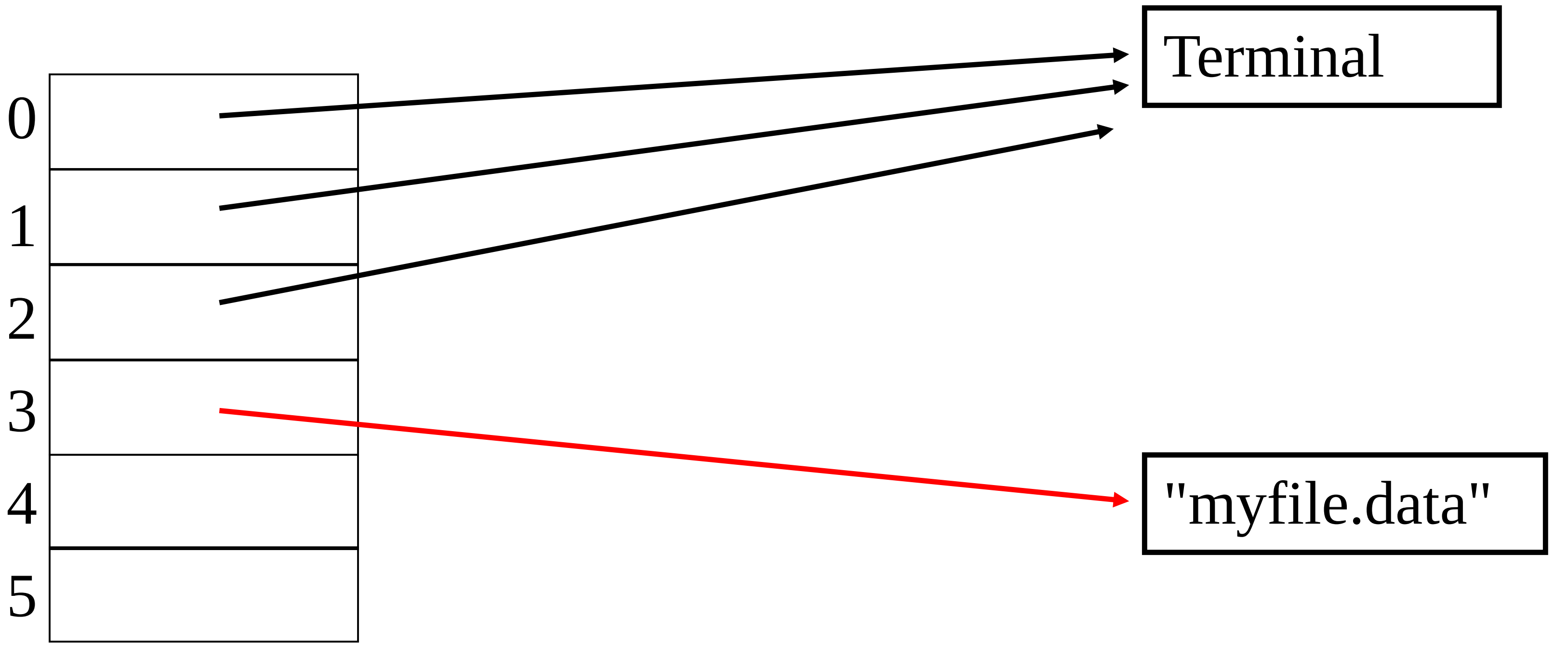
Calls exec

Redirecting I/O



Redirecting stdout

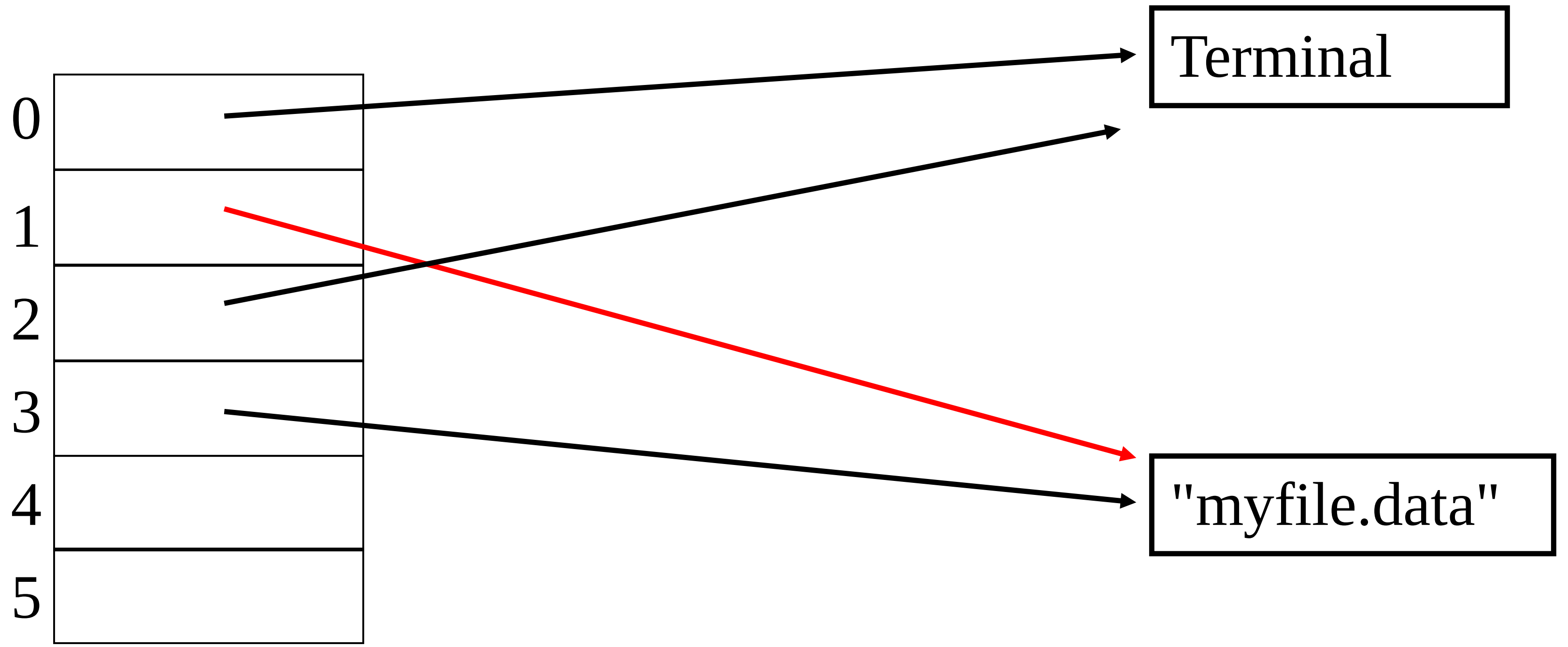
1. First open the new file



Redirecting stdout

2. Call `dup2()` to change fd 1 to point where fd 3 points:

```
dup2(3, 1);
```



```c
fd = open("myout.data", O_WRONLY|O_CREAT|O_TRUNC, 0644);
if (fd == -1)
{
   perror("open");
   exit(1);
}

fd2 = dup2(fd, 1);
if (fd2 == -1)
{
   perror("dup2");
   exit(2);
}

execlp(prog1, prog1, NULL);
perror("exec");
exit(3);
```

Change where stdout is pointing:

•Make it point to where our newly opened output file points

```
fd = open("myin.data", O_RDONLY);
if (fd == -1)
{
    perror("open");
    exit(1);
}

fd2 = dup2(fd, 0);
if (fd2 == -1)
{
    perror("dup2");
    exit(2);
}

execlp(prog1, prog1, NULL);
perror("exec");
exit(3);
```

Change where stdin is pointing:

- Make it point to where our newly opened input file points

Real Inter-Process Communication (IPC)

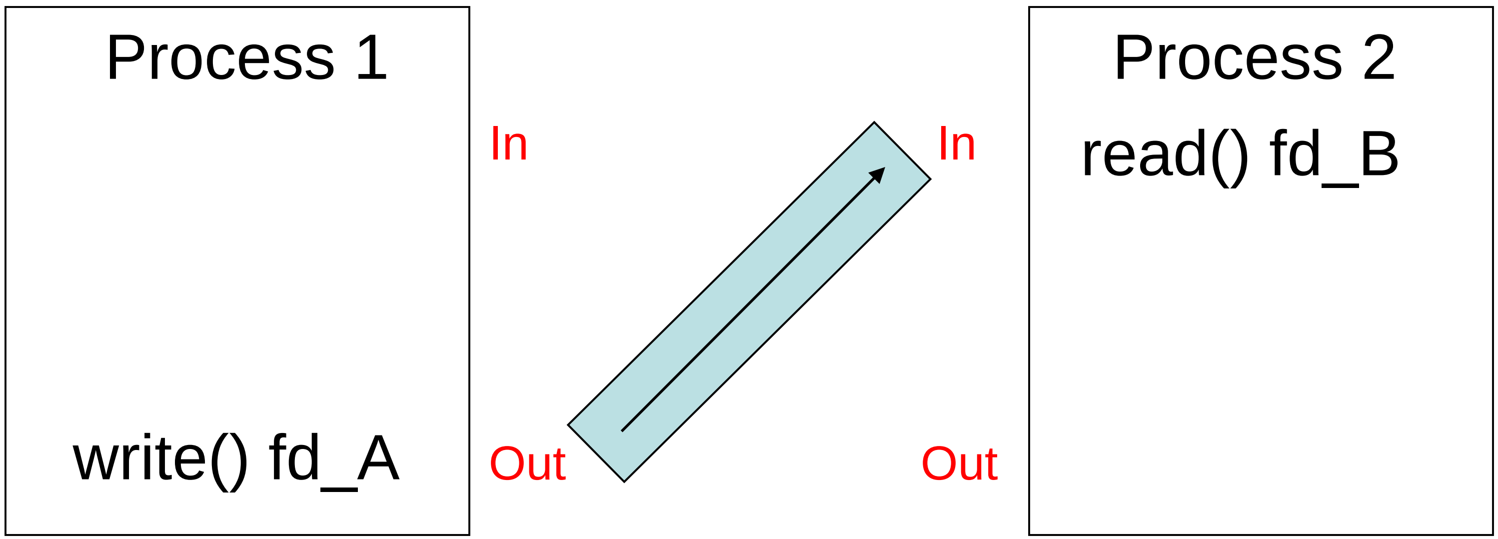
- IPC methods in UNIX
  - Intermediate/temporary files (often together with I/O redirection)
  - Pipes - IPC between two processes forked by a common ancestor process
  - FIFOs (named pipes) - communication between any two processes on the same machine
  - SysV message IPC - communication between any two processes on the same machine
    - Not a simple byte stream
    - Supports message categories - often used for priorities
  - Sockets - communication between any two processes potentially separated by a network

Between process IPC

- I/O redirection with `dup2()` allows you to redirect input and output between processes and files
- How do we redirect input between processes and other processes on the same machine?
  - Use temporary/intermediate files
    - Writes to disk are slow
    - No good way to track when the other process is ready to receive or send new data
  - Better answer: use pipes!

Pipes

- Pipes provide a way to connect an output-only file descriptor in one process to an input-only file descriptor in another process



Creating a Pipe

- Pipes are possible because file descriptors are shared across fork() and exec()
- A parent process creates a pipe
  - Results in two new open file descriptors, one for input and one for output
- The parent process forks (and possibly execs)
  - Parent and child have the fds created with the pipe
- The child process now reads from the input file descriptor, and the parent process writes to the output file descriptor
  - or vice-versa

The `pipe()` function

- You pass `pipe()` an array of two integers, where it stores the two new open file descriptors
- The first is the input file descriptor, and the second is the output file descriptor
- One of the descriptors is used by the parent process and the other is used by the child process

```
int r, pipeFDs[2];
char message[512];
pid_t spawnpid;

…

if (pipe(pipeFDs[2] == -1)
{
   perror("Hull Breach!");
   exit(1);
}

spawnpid = fork();
switch (spawnpid)
{
   case 0:  // Child
      close(pipeFDs[0]);  // close the input file descriptor
      write(pipeFDs[1], "hi parent, this is the child!!", 41);
      exit(0);

   default:  // parent
      close(pipeFDs[1]);  // close output file descriptor
      r = read(pipeFDs[0], message, sizeof(message));
      if (r > 0)
         printf("Message received from child: %s\n", message);
      exit(0);
}
```

flow control with `read()`

- `read()` succeeds if data is available
  - Recieves the data and returns immediately
  - The return value of `read()` tells you how many bytes were read - *it may be less than you requested*

- if data is not available, read() will block waiting for data (your process execution is suspended until data arrives)
  - `read()` is a system call

flow control with `write()`

- Write will not return until all the data has been written
  - `write()` is a system call

- Pipes have a certain size
  - Only so much data will fit in a pipe
  - If the pipe fills up, and there is no more room, write() will block until space becomes available (ie somebody `read`s the data from the pipe)

Pipe Recap

- `write()` puts bytes in the pipe, `read()` takes them out
- It is possible to determine the size of the pipe - see `fpathconf()`

Programming note

- Checking the return value of `read()` becomes very important
  - Not just if return value is -1 (an error)
  - The return value will tell you if the desired number of bytes was not read
- Same goes for `write()`

Closing Pipes

- What happens if a process closes their end of the pipe when the other process is still trying to read or write to the pipe?

Closing Pipes

- Process A closes output pipe.
  - If process B is currently blocked on a `read()`, then process B's `read()` will return 0
- Process B closes input pipe
  - If process A tries to write to the pipe, `write()` will return -1, and errno (in process A) will be set to EPIPE
  - Process A will be sent the SIGPIPE signal