File Descriptors

Unix philosophy: everything is a file

• main.c
• a.out
• /dev/sda1 — the whole disk
• /dev/tty2 — a terminal
• /proc/cpuinfo — CPU as deduced by the kernel
• unnamed channels of communication including input and output streams

A file descriptor is a handle to a file’s input and/or output represented as an int
Opening Files

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int flags);
```

Open a file, where `flags` is typically `O_RDONLY`, `O_WRONLY`, or `O_RDWR`

Adding `O_CREAT` implies an extra argument

```c
#include <unistd.h>

int close(int fd);
```

Closes a file descriptor
Reading and Writing

```c
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t n);
```

Reads from `fd`, putting up to `n` bytes into `buf`

```c
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t n);
```

Write to `fd`, using up to `n` bytes from `buf`

Result in either case is number of bytes read/written

or \(-1\) for an error
Example: Reading a File

```c
#include "csapp.h"

int main(int argc, char **argv) {
    int fd = Open(argv[0], O_RDONLY, 0);
    char buf[5];

    Read(fd, buf, 4);
    buf[4] = 0;

    printf("%s\n", buf+1);
    return 0;
}
```

Prints ELF
Creating a Pipe

```c
#include <unistd.h>

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

Create an unnamed “file”

just in memory — not on a disk

- `fds[0]` is the read end
- `fds[1]` is the write end
Example: Data through a Pipe

```c
#include "csapp.h"

int main(int argc, char **argv) {
    int fds[2];
    char buf[6];

    pipe(fds);

    Write(fds[1], "Hello", 5);

    Read(fds[0], buf, 5);
    buf[5] = 0;

    printf("%s\n", buf);
    return 0;
}
```

Prints
Hello
Example: Pipe Read Waits on Write

```c
#include "csapp.h"

int main(int argc, char **argv) {
    int fds[2];

    pipe(fds);

    if (Fork() == 0) {
        Sleep(1);
        Write(fds[1], "Hello", 5);
    } else {
        char buf[6];
        Read(fds[0], buf, 5);
        buf[5] = 0;
        printf("%s\n", buf);
    }

    return 0;
}
```

Prints
Hello
after 1 second
```c
#include "csapp.h"

int main(int argc, char **argv) {
    int fds[2];
    char buf[6];

    pipe(fds);

    Write(fds[1], "Hello", 5);
    Write(fds[1], "World", 5);
    Close(fds[1]);

    while (1) {
        ssize_t n = Read(fds[0], buf, 3);
        if (n == 0) break;
        buf[n] = 0;
        printf("%s\n", buf);
    }

    return 0;
}
```

Prints:

```
HelloWorld
```
Example: Fork and Closing Pipes

```c
#include "csapp.h"

int main(int argc, char **argv) {
    int fds[2];
    pipe(fds);

    if (Fork() == 0) {
        Write(fds[1], "Hello", 5);
        Close(fds[1]);
    } else {
        // Close(fds[1]);
        while (1) {
            char buf[6];
            ssize_t n = Read(fds[0], buf, 3);
            if (n == 0) break;
            buf[n] = 0;
            printf("%s\n", buf);
        }
    }
    return 0;
}
```

Gets stuck, unless the `Close` call is uncommented
File Descriptors and Open Files

file descriptor table  open file table  underlying device
per-process          shared by all processes  shared by all processes

<table>
<thead>
<tr>
<th>fd 0</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>fd 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fd 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fd 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fd 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|                | position = 0 |              |
| read           | refcount = 1  | ...          |
|                |              |              |

write

| position = 0 | refcount = 1 |
|              | ...          |

pipe(fds)

Hello...
File Descriptors and Open Files

- **file descriptor table**: per-process
- **open file table**: shared by all processes
- **underlying device**: shared by all processes

```
  fd 0
  fd 1
  fd 2
  fd 3
  fd 4
  ...
```

```
  read
  position = 0
  refcount = 1
  ...

  write
  position = 0
  refcount = 0
  ...
```

```
pipe(fds)  close(fds[1])  ...
```
File Descriptors and Open Files

- **File Descriptor Table**: 
  - *Per-process*
  - fd 0
  - fd 1
  - fd 2
  - fd 3
  - fd 4
  - ...

- **Open File Table**: 
  - *Shared by all processes*
  - read
    - position = 0
    - refcount = 1
    - ...
  - write
    - position = 0
    - refcount = 1
    - ...

- **Underlying Device**: 
  - *Shared by all processes*
  - Hello...

- **Pipe (fds)**
  - pipe(fds) ...

- **Miscellaneous**
File Descriptors and Open Files

- **File descriptor table**: per-process
- **Open file table**: shared by all processes
- **Underlying device**: shared by all processes

```plaintext
fd 0
fd 1
fd 2
fd 3
fd 4
...
```

```plaintext
read
position = 0
refcount = 2
...
```

```plaintext
write
position = 0
refcount = 2
...
```

```plaintext
pipe(fds)
fork
...
```
File Descriptors and Open Files

file descriptor table  open file table  underlying device
per-process  shared by all processes  shared by all processes

fd 0
fd 1
fd 2
fd 3
fd 4
...

fd 0
fd 1
fd 2
fd 3
fd 4
...

read
position = 0
refcount = 2
...

write
position = 0
refcount = 1
...

Hello...

pipe(fds)
fork

close(fds[1])
File Descriptors and Open Files

file descriptor table    open file table    underlying device
  per-process           shared by all processes  shared by all processes

<table>
<thead>
<tr>
<th>fd 0</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>fd 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fd 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fd 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fd 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

read
  position = 0
  refcount = 1
  ...

write
  position = 0
  refcount = 1
  ...

Hello...

close(fds[1])  exit

pipe(fds)  fork  ...

fork...
File Descriptors and Open Files

- **file descriptor table** (per-process)
- **open file table** (shared by all processes)
- **underlying device** (shared by all processes)

<table>
<thead>
<tr>
<th>fd 0</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>fd 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fd 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fd 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fd 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
pipe(fds)
fork
```

- `close(fds[1])`
- `exit`

```
read
position = 0
refcount = 1
...

write
position = 0
refcount = 0
...
```

Hello
#include "csapp.h"

int main(int argc, char **argv) {
    int fds[2];

    pipe(fds);

    if (Fork() == 0) {
        char buf[6];
        Sleep(2);
        while (Read(fds[0], buf, 6) > 1) {
        }
    } else {
        int i;
        for (i = 0; i < 20000; i++)
            Write(fds[1], "Hello", 5);
        printf("done\n");
    }

    return 0;
}
Pipe Buffer Size

file descriptor table  
*per-process*

| fd 0 |  |
| fd 1 |  |
| fd 2 |  |
| fd 3 |  |
| fd 4 |  |
| ...  |  |

open file table  
*shared by all processes*

- read
  - position = 0
  - refcount = 1
  - ...

- write
  - position = 0
  - refcount = 1
  - ...

underlying device  
*shared by all processes*

Hello...

pipe buffer holds only so much
Input, Output, and Error

Every process starts with at least 3 file descriptors:

• 0 = standard input (read)
• 1 = standard output (write)
• 2 = standard error (write)
Using Standard File Descriptors

```c
#include "csapp.h"

int main() {
    char buffer[32];
    int n;

    Write(1, "Your name? ", 11);

    n = Read(0, buffer, 32);

    Write(2, "Unknown: ", 9);
    Write(2, buffer, n);

    return 0;
}
```

Writes to output, reads from input, writes to error
Setting Standard File Descriptors

`fork` creates a process with the same ports as the parent

A shell needs a way to redirect input, output, and errors

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```

`dup2` makes `newfd` refer to the same open file as `oldfd`

  if `newfd` is already used, closes it first
#include "csapp.h"

int main() {
    pid_t pid;
    int fds[2], n;
    Pipe(fds);

    pid = Fork();
    if (pid == 0) {
        Dup2(fds[1], 1);
        printf("Hello!");
    } else {
        char buffer[32];
        Close(fds[1]);
        Waitpid(pid, NULL, 0);
        n = Read(fds[0], buffer, 31);
        buffer[n] = 0;
        printf("Got: %s\n", buffer);
    }
    return 0;
}
Redirecting File Descriptors

file descriptor table  open file table  underlying device

per-process          shared by all processes    shared by all processes

<table>
<thead>
<tr>
<th>fd 0</th>
<th>read</th>
<th>pipe(fds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd 1</td>
<td>position = 0</td>
<td></td>
</tr>
<tr>
<td>fd 2</td>
<td>refcount = 1</td>
<td></td>
</tr>
<tr>
<td>fd 3</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>fd 4</td>
<td>write</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>position = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>refcount = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Redirecting File Descriptors

file descriptor table | open file table | underlying device
---|---|---
*per-process* | *shared by all processes* | *shared by all processes*

| fd 0 | | | read |
|------|---|---|
| fd 1 | | | position = 0 |
| fd 2 | | | refcount = 2 |
| fd 3 | | | ... |
| fd 4 | | | write |
| ... | | | position = 0 |
| | | | refcount = 2 |
| | | | ... |

pipe(fds) |
---
fork |
---
...
Redirecting File Descriptors

file descriptor table

per-process

<table>
<thead>
<tr>
<th>fd 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd 1</td>
</tr>
<tr>
<td>fd 2</td>
</tr>
<tr>
<td>fd 3</td>
</tr>
<tr>
<td>fd 4</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

open file table

shared by all processes

<table>
<thead>
<tr>
<th>read</th>
</tr>
</thead>
<tbody>
<tr>
<td>position = 0</td>
</tr>
<tr>
<td>refcount = 2</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>position = 0</td>
</tr>
<tr>
<td>refcount = 3</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

underlying device

shared by all processes

dup2(fds[1], 1)
Redirecting File Descriptors

- **file descriptor table**: per-process
- **open file table**: shared by all processes
- **underlying device**: shared by all processes

```plaintext
<table>
<thead>
<tr>
<th>fd 0</th>
<th>fd 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd 2</td>
<td>fd 3</td>
</tr>
<tr>
<td>fd 4</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fd 0</th>
<th>fd 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd 2</td>
<td>fd 3</td>
</tr>
<tr>
<td>fd 4</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
dup2  close(fds[0])
close(fds[1])
```

```
fork  ...  close(fds[1])
```
Redirecting File Descriptors

file descriptor table | open file table | underlying device
---|---|---
per-process | shared by all processes | shared by all processes

| fd 0 | read | Hello! |
| fd 1 | position = 0 |
| fd 2 | refcount = 1 |
| fd 3 | ... |
| fd 4 | write |
| ... |
| close(fds[1]) |

fork | ... | dup2 | printf | exit
Shell Pipeline

$ cat *.c | grep fork | wc -l

- `pipe(fdsA)` for a `cat-to-grep` connection
- `pipe(fdsB)` for a `grep-to-wc` connection
- `fork` three times; in those children:
  - `dup2(fdsA[1], 1)
    exec("/bin/cat", ...)
  - `dup2(fdsA[0], 0)
    dup2(fdsB[1], 1)
    exec("/bin/grep", ...)
  - `dup2(fdsB[0], 0)
    exec("/bin/wc", ...)"
$ cat *.c | grep fork | wc -l

- pipe(fdsA) for a **cat-to-grep** connection
- pipe(fdsB) for a **grep-to-wc** connection
- fork three times; in those children:
  - dup2(fdsA[1], 1)
    exec("/bin/cat", ...)
  - dup2(fdsA[0], 0)
    dup2(fdsB[1], 1)
    exec("/bin/grep", ...)
  - dup2(fdsB[0], 0)
    exec("/bin/wc", ...)

Before exec, plus parent:
  close(fdsA[0])
  close(fdsA[1])
  close(fdsB[0])
  close(fdsB[1])
Shell Pipeline

```
$ cat *.c | grep fork | wc -l
```

Pipe buffer limit keeps `cat` from getting too far ahead of `grep`
Unix I/O vs. C Library I/O

- **Unix**
  - file descriptors as `int`
  - `open`, `read`, `write`, ...

- **Standard C**
  - file handles as `FILE*`
  - `fopen`, `fread`, `fwrite`, ...

Convert from file descriptor to `FILE*` using `fdopen`

Predefined:
- `stdin = fdopen(0, "r")`
- `stdout = fdopen(1, "w")`
- `stderr = fdopen(2, "w")`
Unix I/O vs. C Library I/O

```c
#include "csapp.h"

#define ITERS 1000000

int main() {
    int fds[2];
    int i;

    Pipe(fds);
    if (Fork() == 0) {
        FILE *out = fdopen(fds[1], "w");
        for (i = 0; i < ITERS; i++)
            Write(fds[1], "Hello", 5);
    } else {
        FILE *in = fdopen(fds[0], "r");
        char buffer[5];
        int n = 0;
        for (i = 0; i < ITERS; i++)
            n += Read(fds[0], buffer, 5);
        printf("%d\n", n);
    }
    return 0;
}
```

```c
#include "csapp.h"

#define ITERS 1000000

int main() {
    int fds[2];
    int i;

    Pipe(fds);
    if (Fork() == 0) {
        FILE *out = fdopen(fds[1], "w");
        for (i = 0; i < ITERS; i++)
            fwrite("Hello", 1, 5, out);
    } else {
        FILE *in = fdopen(fds[0], "r");
        char buffer[5];
        int n = 0;
        for (i = 0; i < ITERS; i++)
            n += fread(buffer, 1, 5, in);
        printf("%d\n", n);
    }
    return 0;
}
```
#include "csapp.h"

#define ITERS 1000000

int main() {
    int fds[2];
    int i;

    Pipe(fds);
    if (Fork() == 0) {
        FILE *out = fdopen(fds[1], "w");
        for (i = 0; i < ITERS; i++)
            Write(fds[1], "Hello", 5);
    } else {
        FILE *in = fdopen(fds[0], "r");
        char buffer[5];
        int n = 0;
        for (i = 0; i < ITERS; i++)
            n += Read(fds[0], buffer, 5);
        printf("%d\n", n);
    }
    return 0;
}
Unix I/O vs. C Library I/O

User

Read(..., 5)

Kernel
Unix I/O vs. C Library I/O

User

\text{Read}(\ldots, 5)

\downarrow

\text{read}(\ldots, 5)

Kernel
Unix I/O vs. C Library I/O

User

Read(..., 5)

Kernel

read(..., 5)
Unix I/O vs. C Library I/O

User

Read(..., 5)

read(..., 5)

Kernel

System call through kernel every time
Unix I/O vs. C Library I/O

User

fread(..., 5, ...)

Kernel
Unix I/O vs. C Library I/O

User

fread(..., 5, ...)

read(..., 4096)

Kernel
Unix I/O vs. C Library I/O

User

\texttt{fread(..., 5, ...)}

\texttt{read(..., 4096)}

Kernel

Extra bytes are stored in the \texttt{FILE} record
Unix I/O vs. C Library I/O

User

fread(..., 5, ...)

Kernel

Extra bytes are stored in the FILE record
Unix I/O vs. C Library I/O

User

`fread(...)`, 5, `...`  

`memcpy(...)`

Kernel

Extra bytes are stored in the `FILE` record

Fast when buffered bytes are available
I/O Options

Unix I/O
+ Precise control
- Slow for small transfers
- Partial reads/write possible due to limits or signals

Standard C
+ Fast via buffering
+ Many conveniences
- Less control

From csapp.c:

- **sio**....: convenience around Unix I/O
- **rio**....: partial-handling wrapper around Unix I/O