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

```
#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

```
#include <unistd.h>

int close(int fd);
```

Closes a file descriptor

Reading and Writing

```
#include <unistd.h>

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

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

```
#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

```
#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;
}
```

Copy

Prints
`ELF`

Creating a Pipe

```
#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

```
#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;
}
```

Copy

Prints

```
Hello
```

Example: Pipe Read Waits on Write

```
#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;
}
```

Copy

Prints

`Hello`

after 1 second

Example: EOF Result

```
#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;
}
```

Copy

Prints

```
Hel
loW
orl
d
```

Example: Fork and Closing Pipes

```
#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;
}
```

Copy

Gets stuck, unless the `Close` call is uncommented

File Descriptors and Open Files

file descriptor table

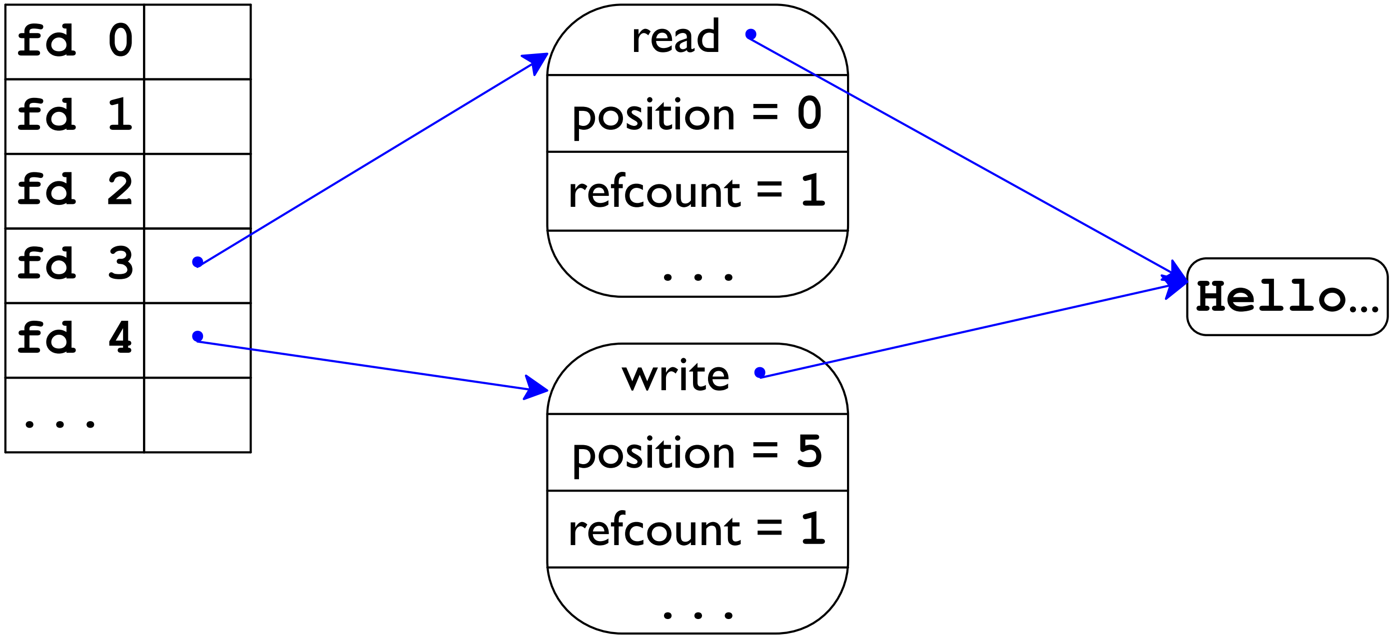
per-process

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 = 5

refcount = 1

. . .

`Hello...`

`pipe(fds)` **`. . .`**

File Descriptors and Open Files

file descriptor table

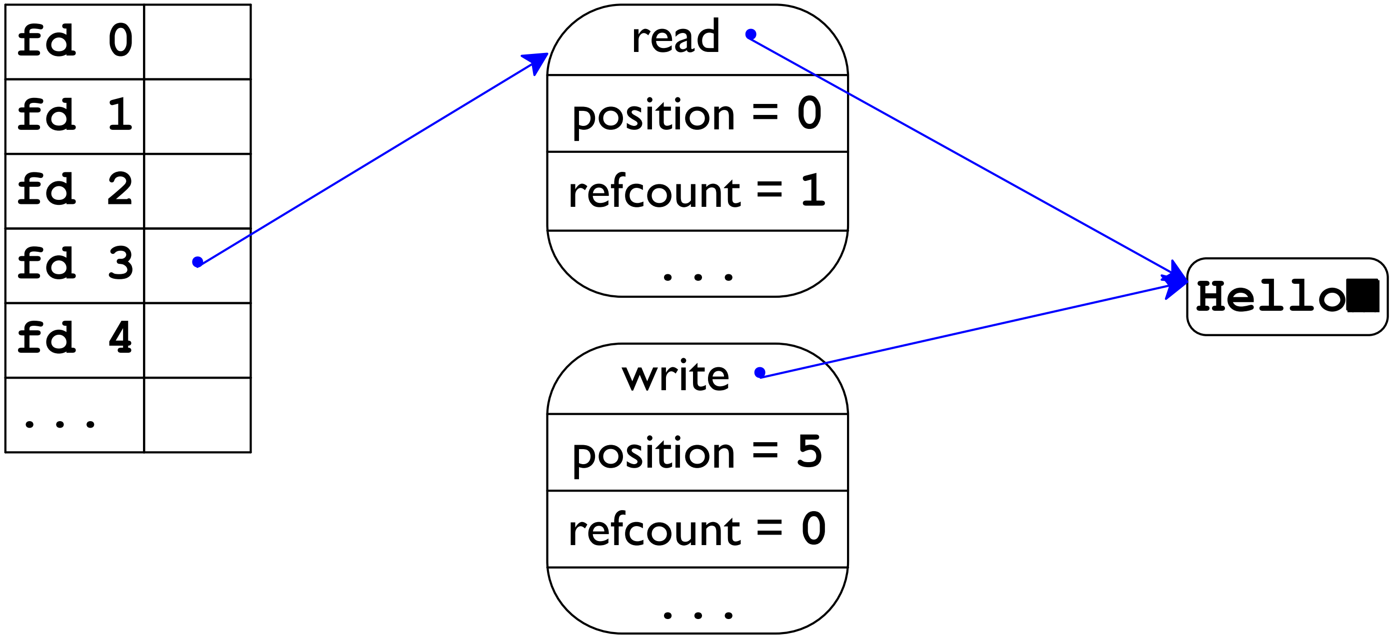
per-process

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 = 5

refcount = 0

...

Hello■

`pipe(fds)` **`close(fds[1])`** **`...`**

File Descriptors and Open Files

file descriptor table

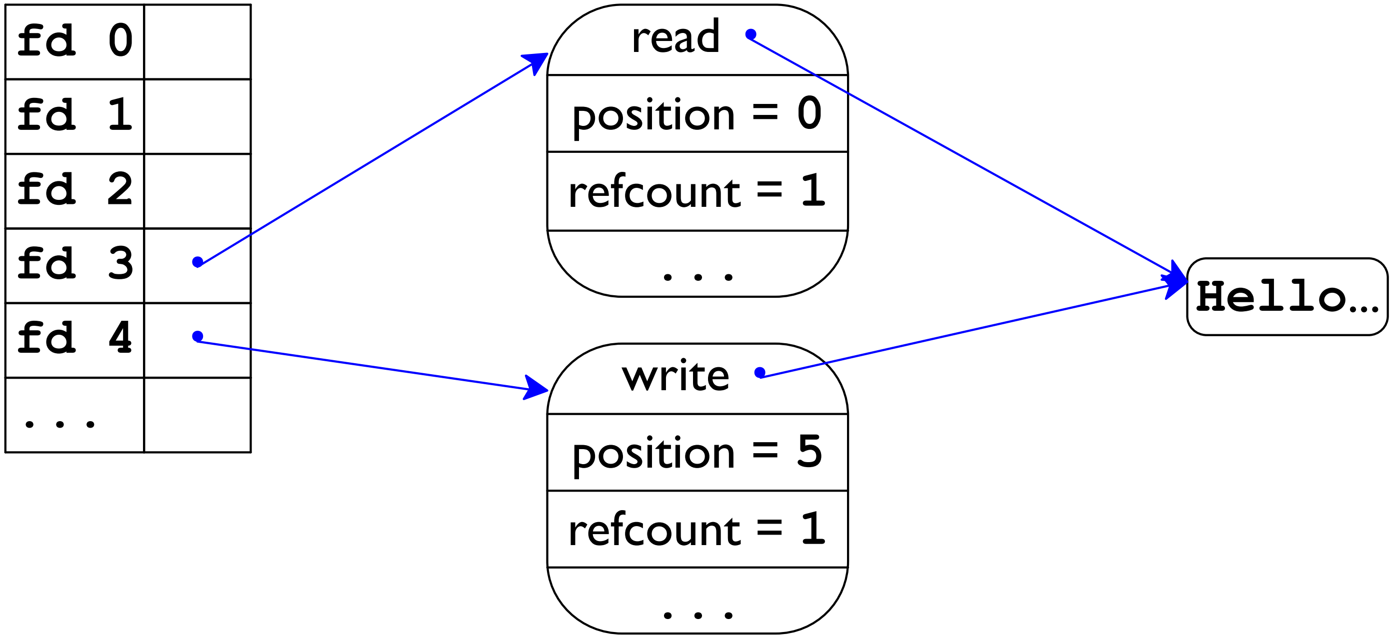
per-process

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

...

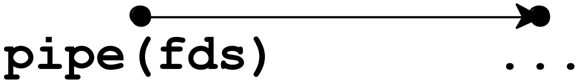
write

position = 5

refcount = 1

...

`Hello...`



`pipe(fds)`

`...`

File Descriptors and Open Files

file descriptor table

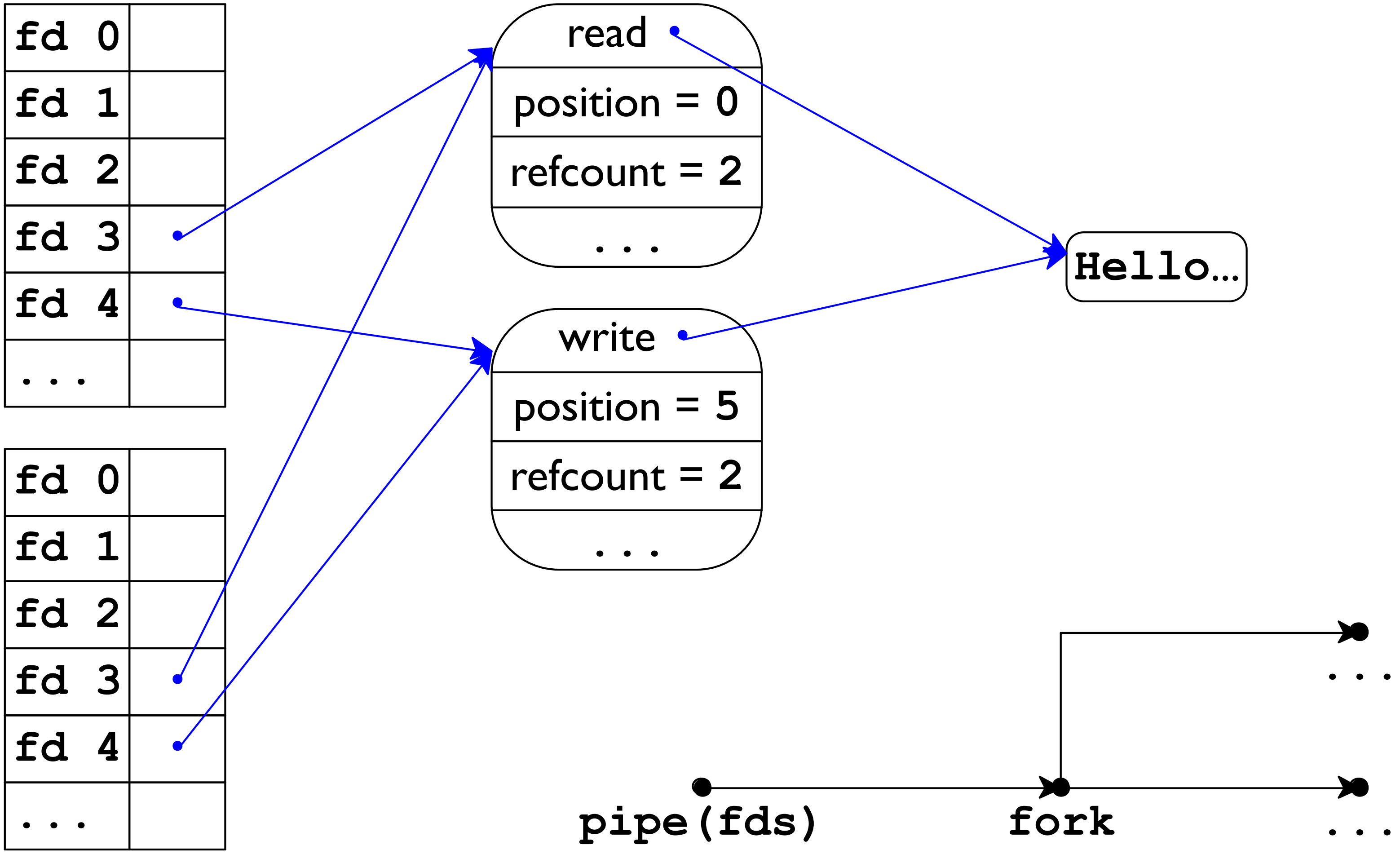
per-process

open file table

shared by all processes

underlying device

shared by all processes



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

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

read
position = 0
refcount = 2
...

write
position = 5
refcount = 2
...

`Hello...`

`pipe(fds)`
`fork`
...
...

Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

File Descriptors and Open Files

file descriptor table

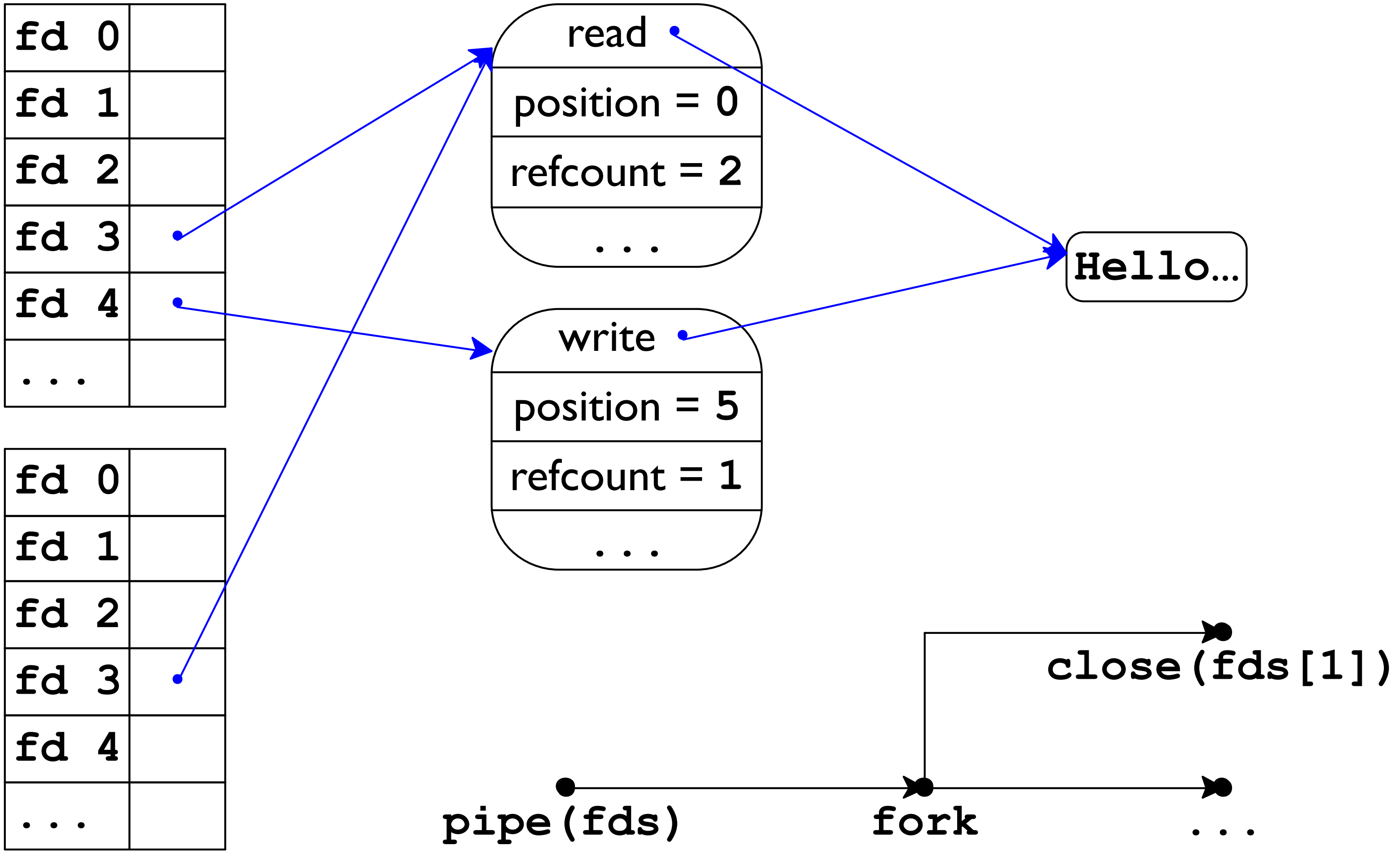
per-process

open file table

shared by all processes

underlying device

shared by all processes



`fd 0`

`fd 1`

`fd 2`

`fd 3`

`fd 4`

`...`

`fd 0`

`fd 1`

`fd 2`

`fd 3`

`fd 4`

`...`

read

position = 0

refcount = 2

...

write

position = 5

refcount = 1

...

`Hello...`

`close(fds[1])`

`pipe(fds)`

`fork`

`...`

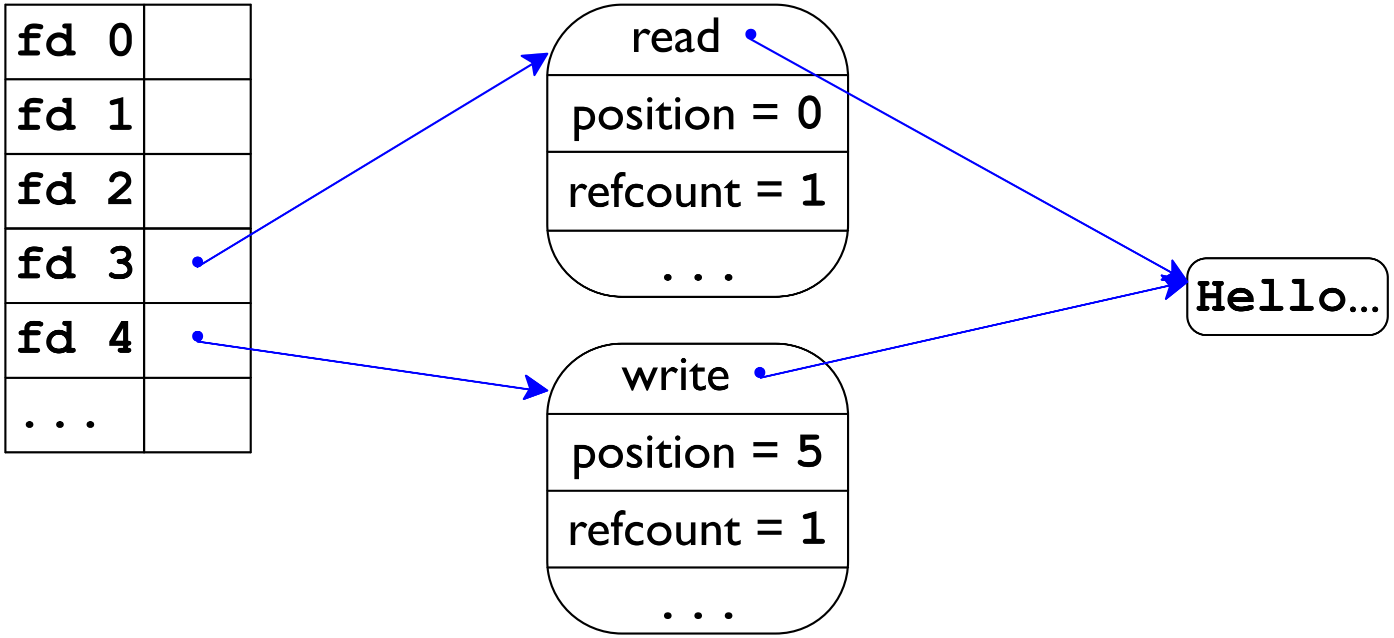
Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

File Descriptors and Open Files

file descriptor table
per-process

open file table
shared by all processes

underlying device
shared by all processes

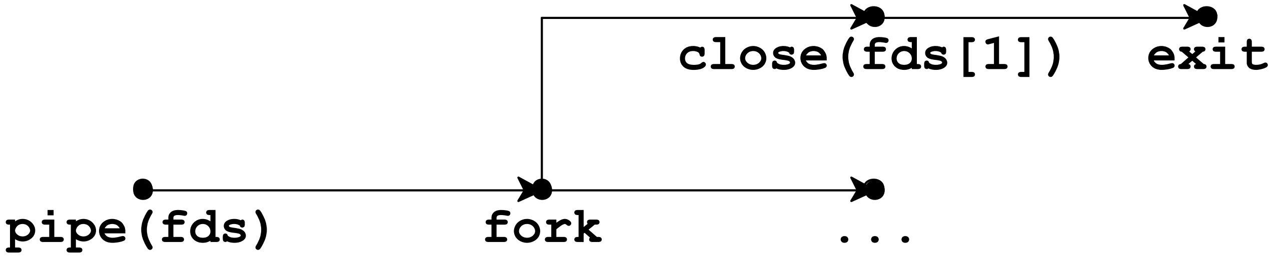


`fd 0`	
`fd 1`	
`fd 2`	
`fd 3`	
`fd 4`	
`. . .`	

read
position = 0
refcount = 1
. . .

write
position = 5
refcount = 1
. . .

`Hello...`



`pipe(fds)` **`fork`** **`. . .`**

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

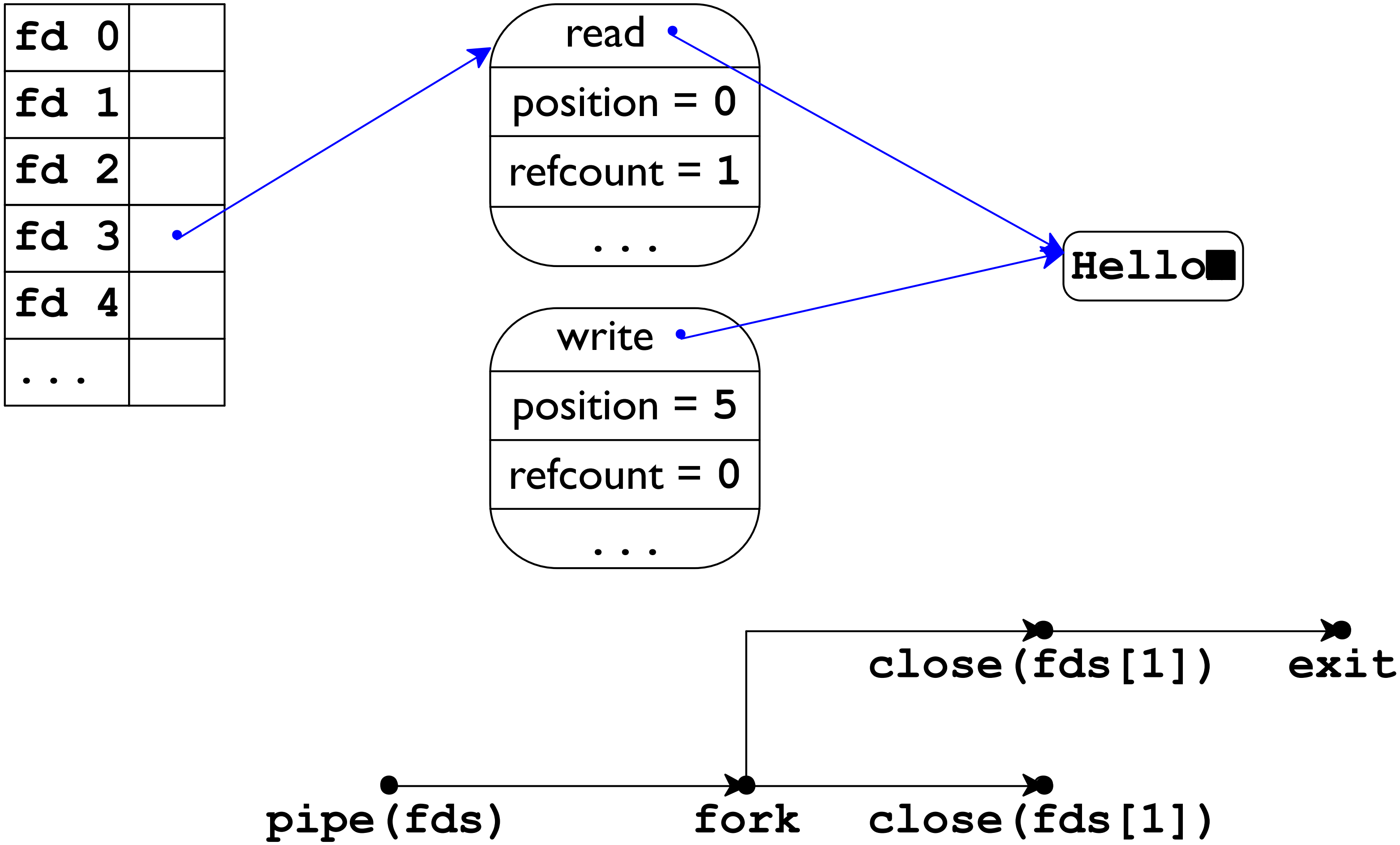
Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

File Descriptors and Open Files

file descriptor table
per-process

open file table
shared by all processes

underlying device
shared by all processes



Example: Pipe Write Can Wait on Read

```
#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) > 0) { }
  } else {
    int i;
    for (i = 0; i < 20000; i++)
      Write(fds[1], "Hello", 5);
    printf("done\n");
  }

  return 0;
}
```

Copy

Prints

`done`

after ~2 seconds

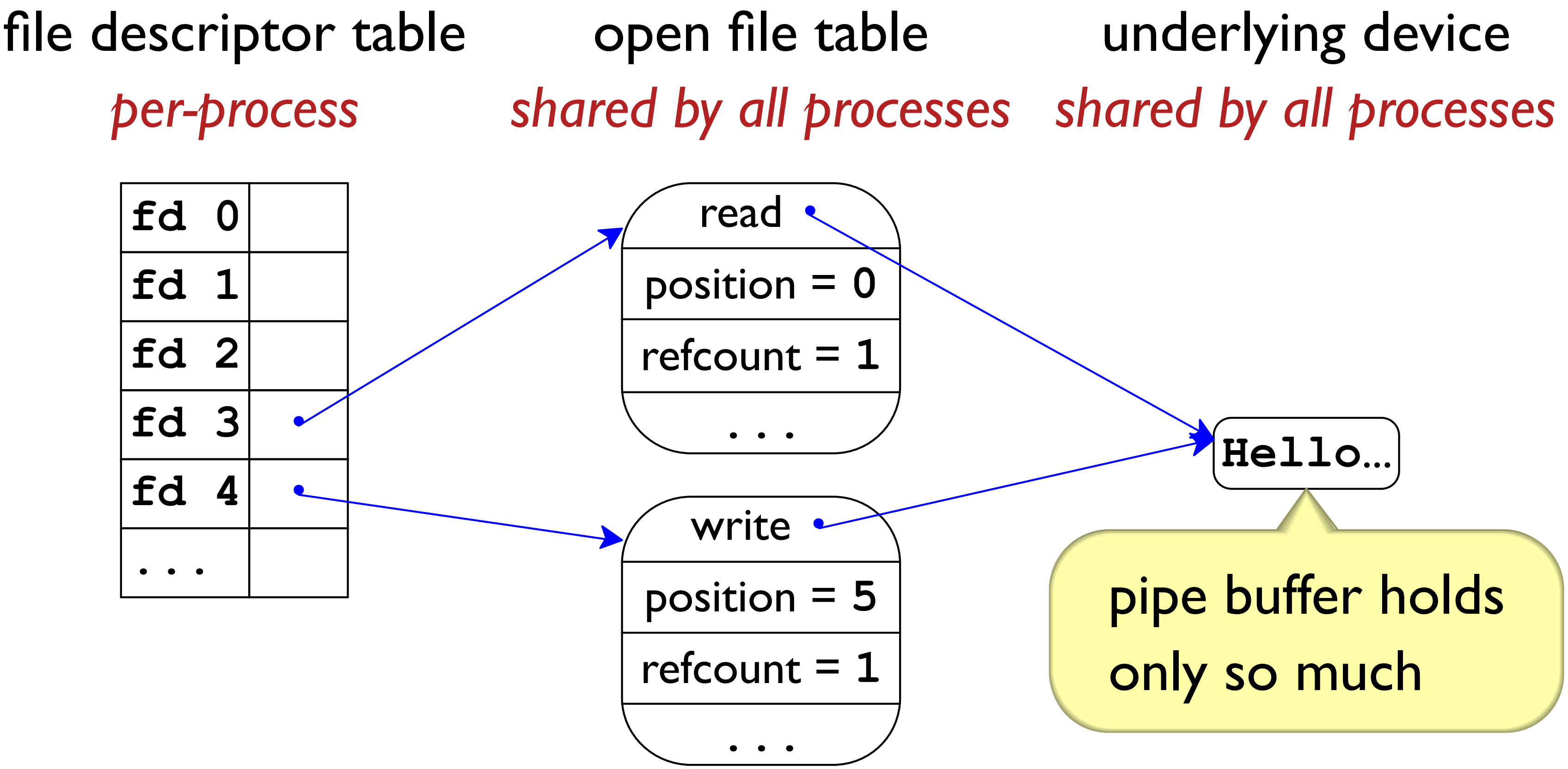
`Sleep(1)`

⇒ ~1 second

fewer iterations

⇒ ~0 seconds

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 = 5

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

```
#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;
}
```

Copy

Writes to output, reads from input, writes to error

Setting Standard File Descriptors

`fork` creates a process with the same file descriptors as the parent

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

```
#include <unistd.h>

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

Makes **`newfd`** refer to the same open file as **`oldfd`**

if **`newfd`** is already used, closes it first

Capturing Child Output

```
#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;
}
```

Copy

Prints

`Got: Hello!`

because **`printf`** writes to **1**

Redirecting File Descriptors

file descriptor table

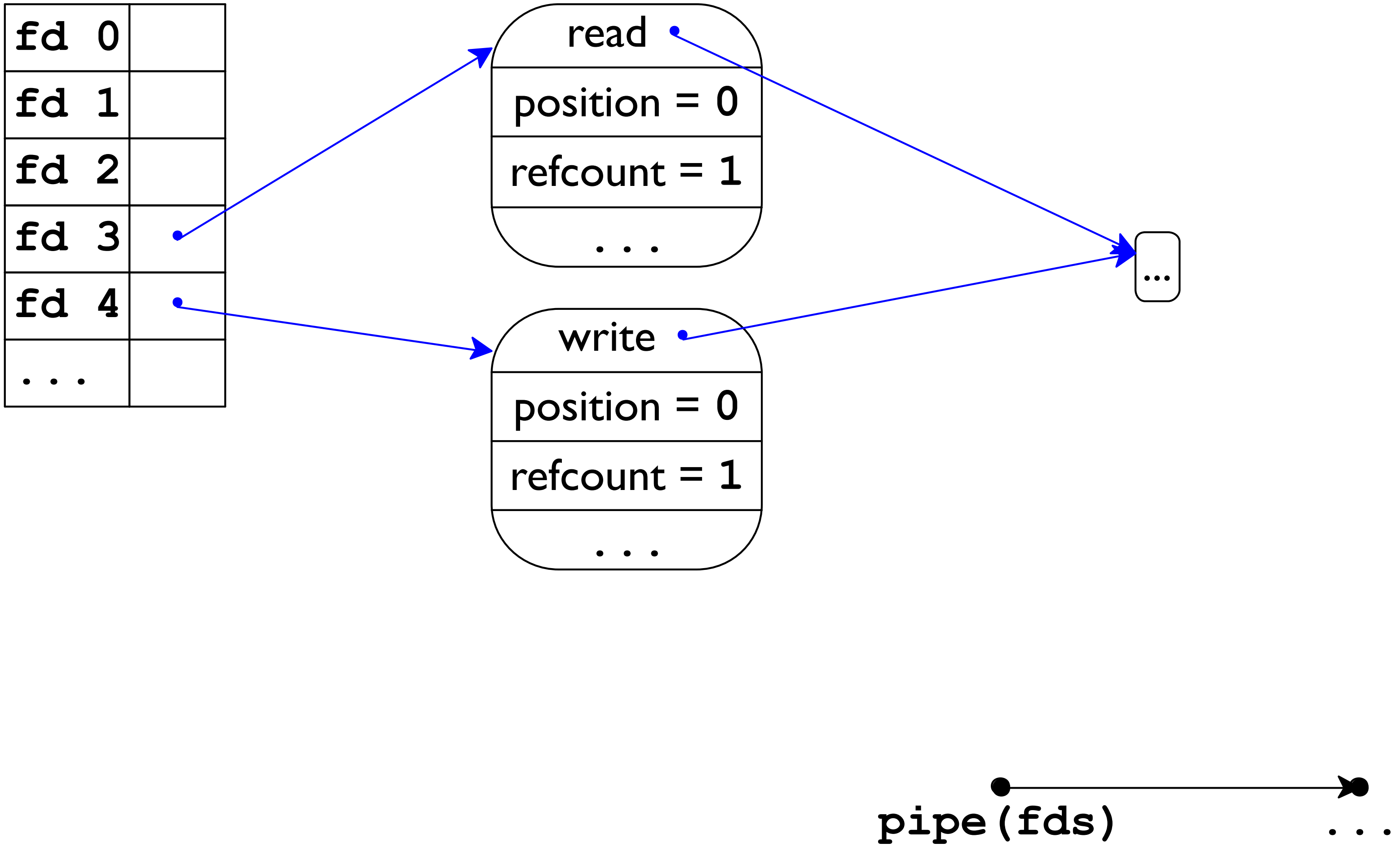
per-process

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 = 1

. . .

...

`pipe(fds)`

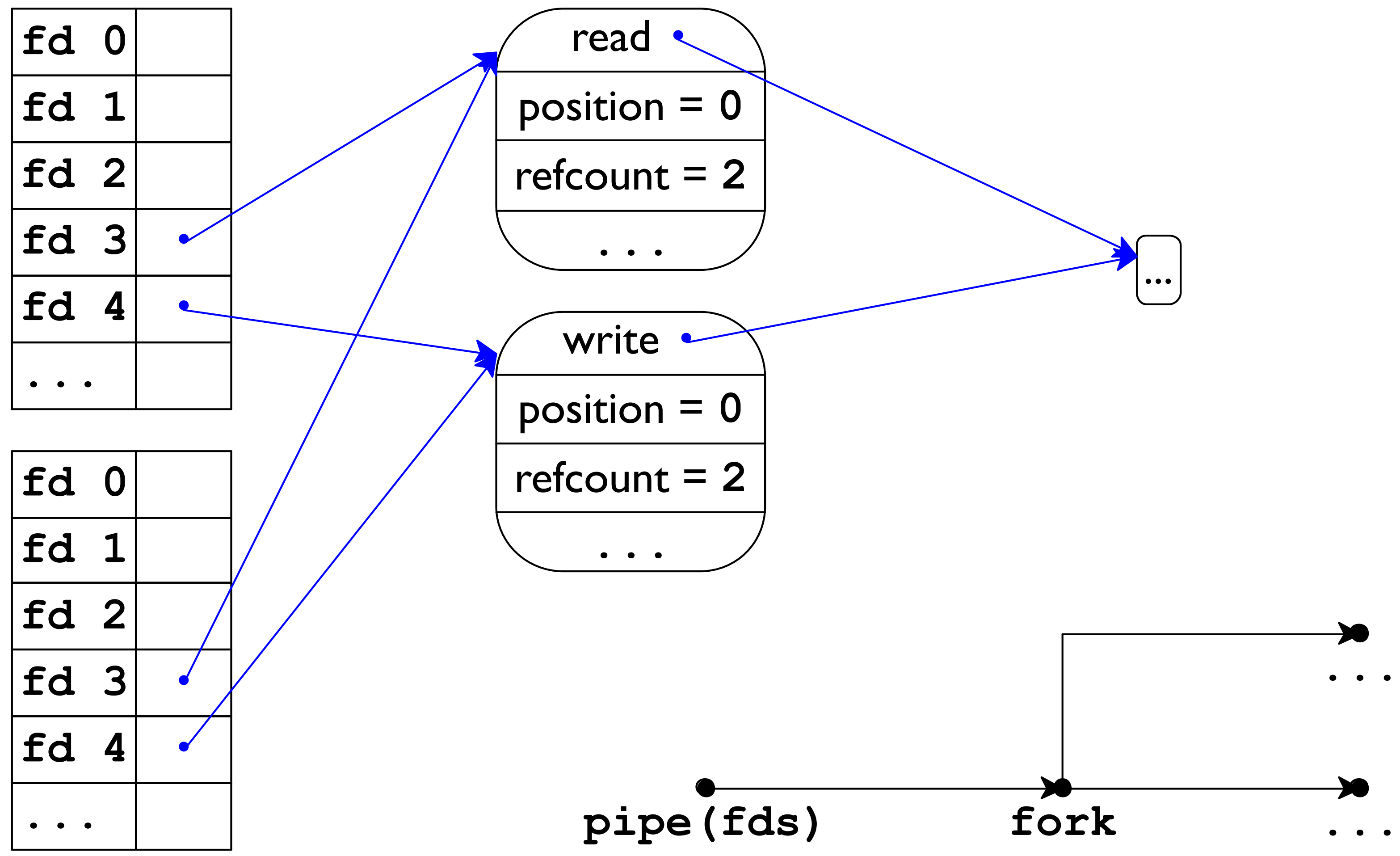
`. . .`

Redirecting File Descriptors

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`
`...`

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

read
position = 0
refcount = 2
. . .

write
position = 0
refcount = 2
. . .

...

`pipe(fds)`
`fork`
. . .
. . .

Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

Redirecting File Descriptors

file descriptor table

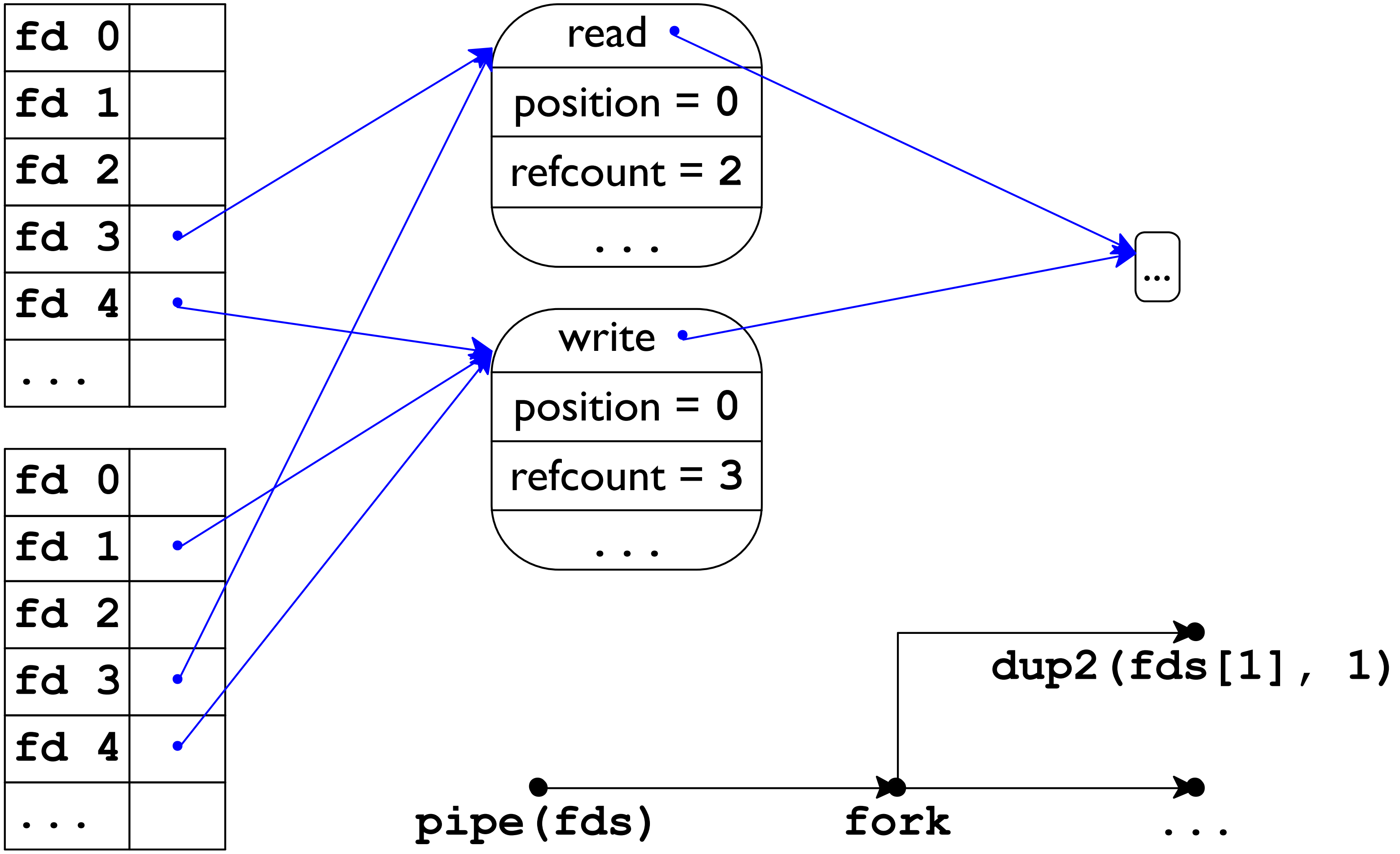
per-process

open file table

shared by all processes

underlying device

shared by all processes



`fd 0`

`fd 1`

`fd 2`

`fd 3`

`fd 4`

`...`

`fd 0`

`fd 1`

`fd 2`

`fd 3`

`fd 4`

`...`

read

position = 0

refcount = 2

...

write

position = 0

refcount = 3

...

...

`dup2(fds[1], 1)`

`pipe(fds)`

`fork`

`...`

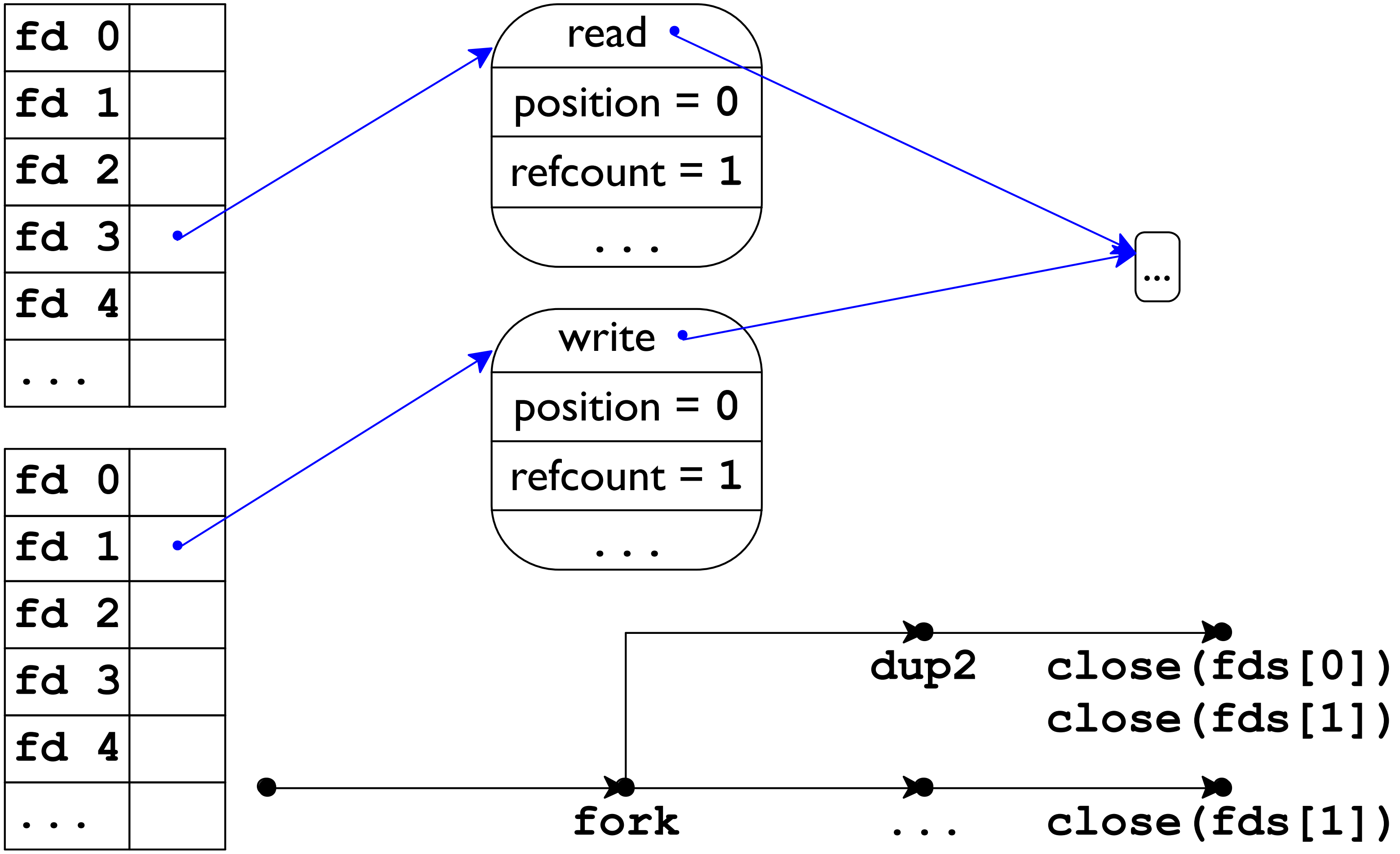
Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

Redirecting File Descriptors

file descriptor table
per-process

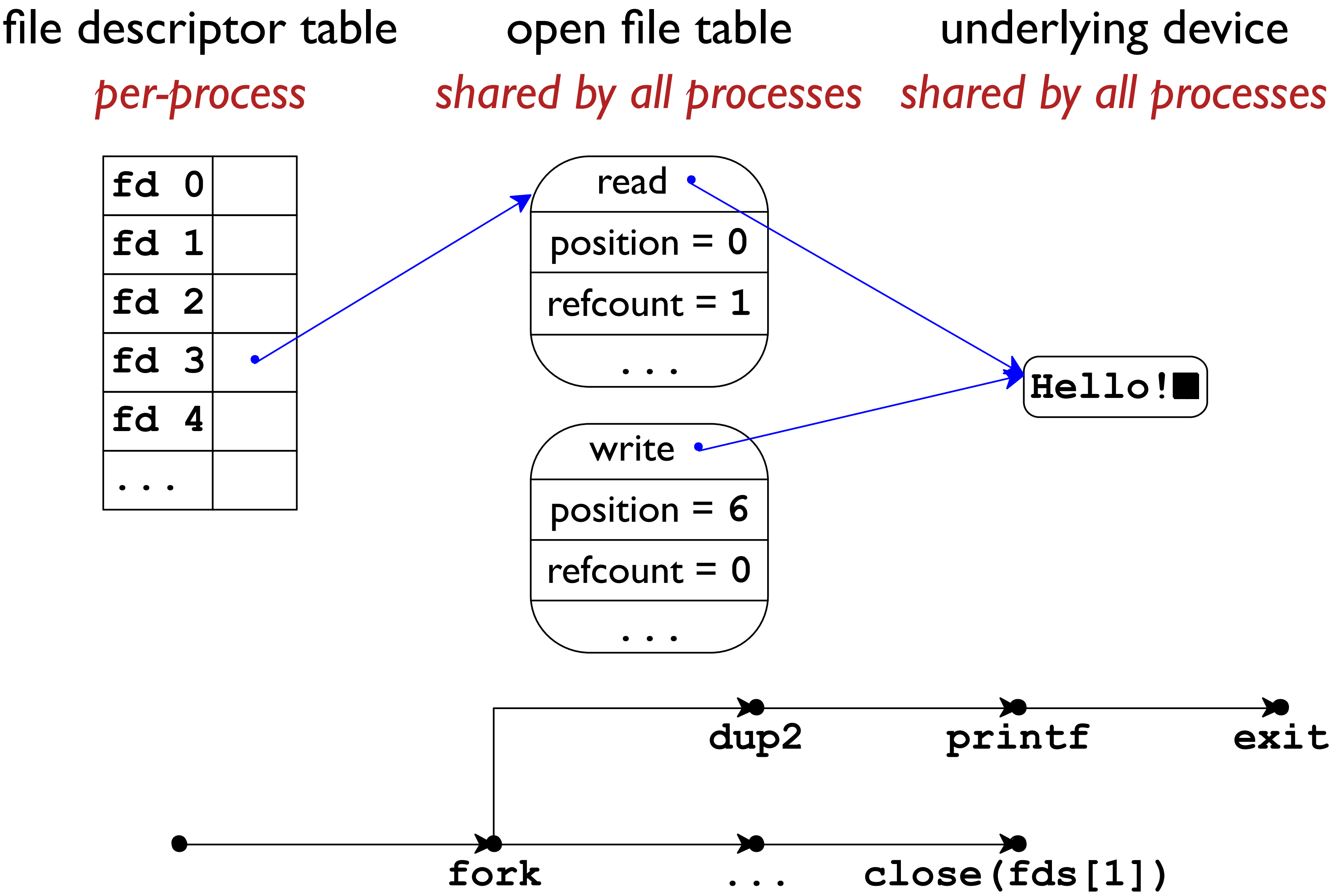
open file table
shared by all processes

underlying device
shared by all processes



Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

Redirecting File Descriptors



Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

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", ...)
      ```

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", ...)
    ```

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

```
#include "csapp.h"

#define ITERS 1000000

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

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

Copy

```
#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;
}
```

Copy

Unix I/O vs. C Library I/O

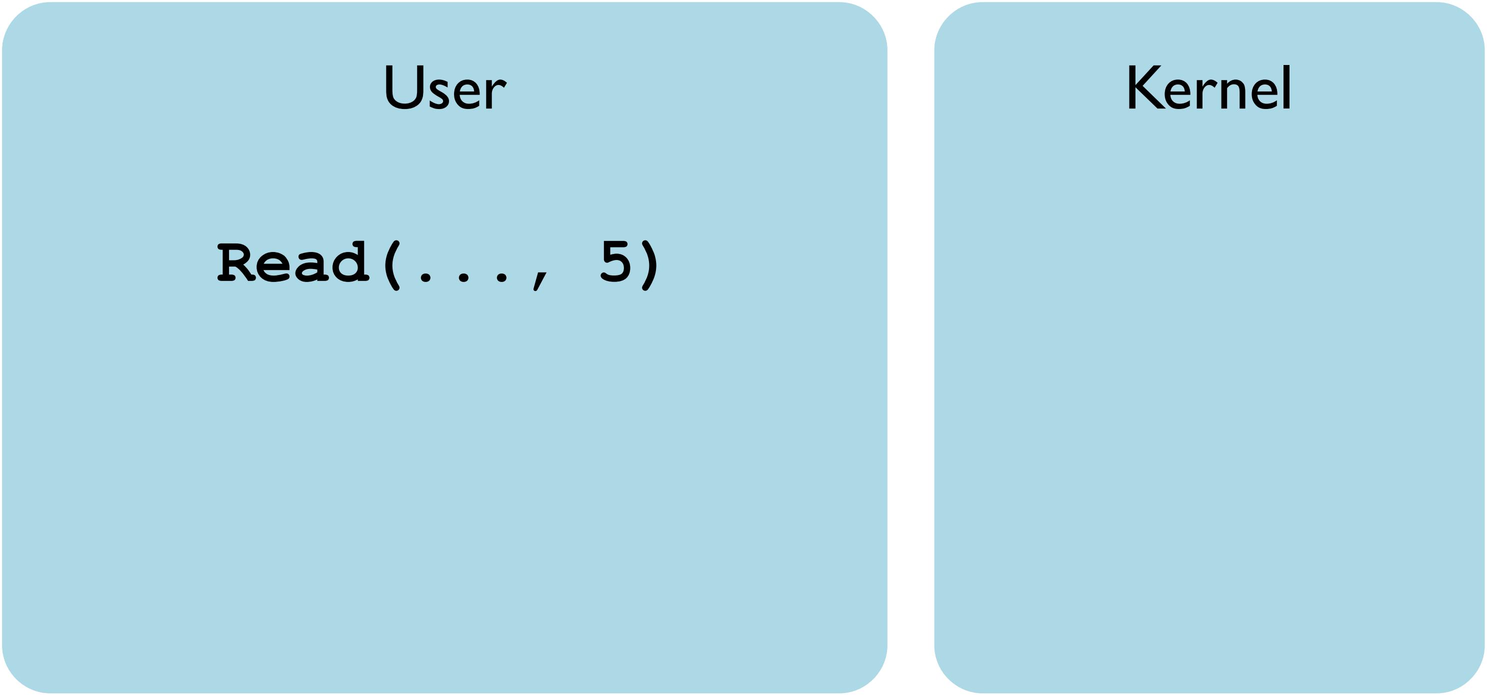
```
#include "csapp.h"

#define ITERS 1000000

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

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

10x faster!

```
#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;
}
```

Copy

Unix I/O vs. C Library I/O

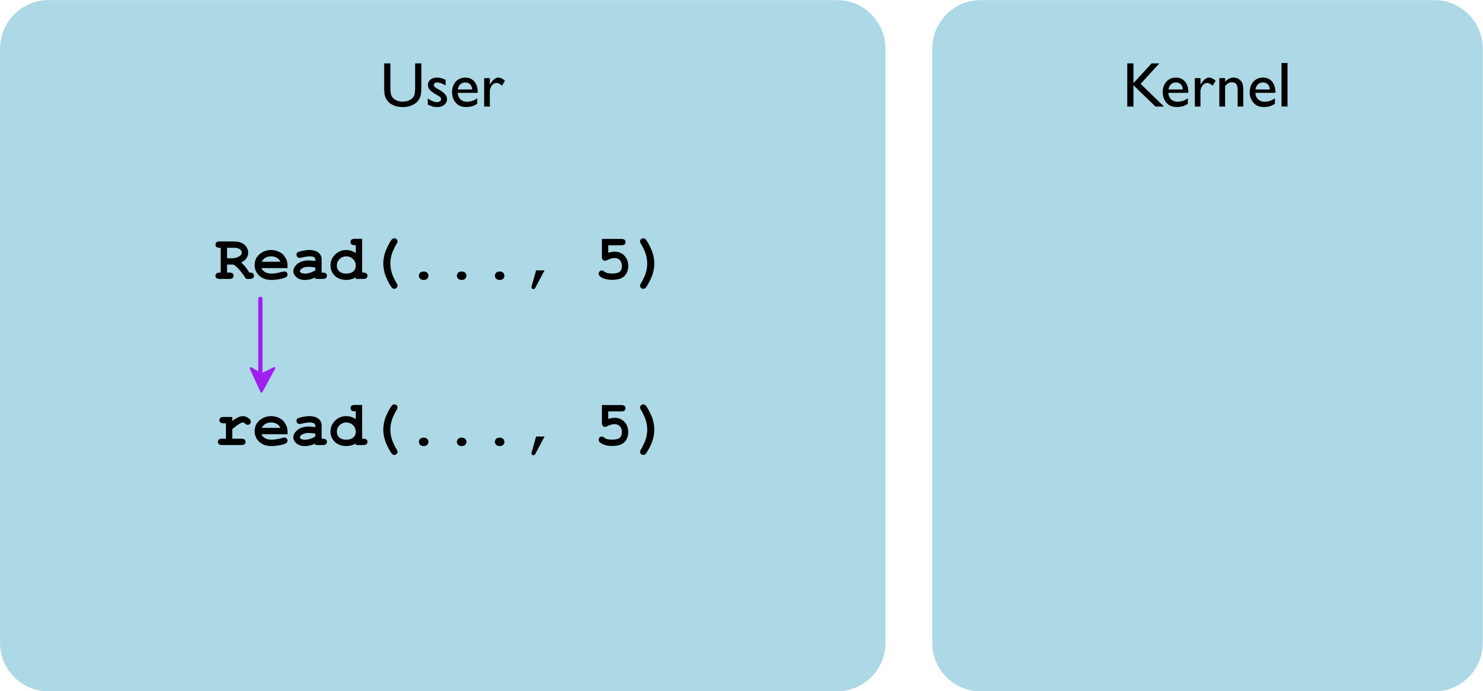


User

`Read(..., 5)`

Kernel

Unix I/O vs. C Library I/O



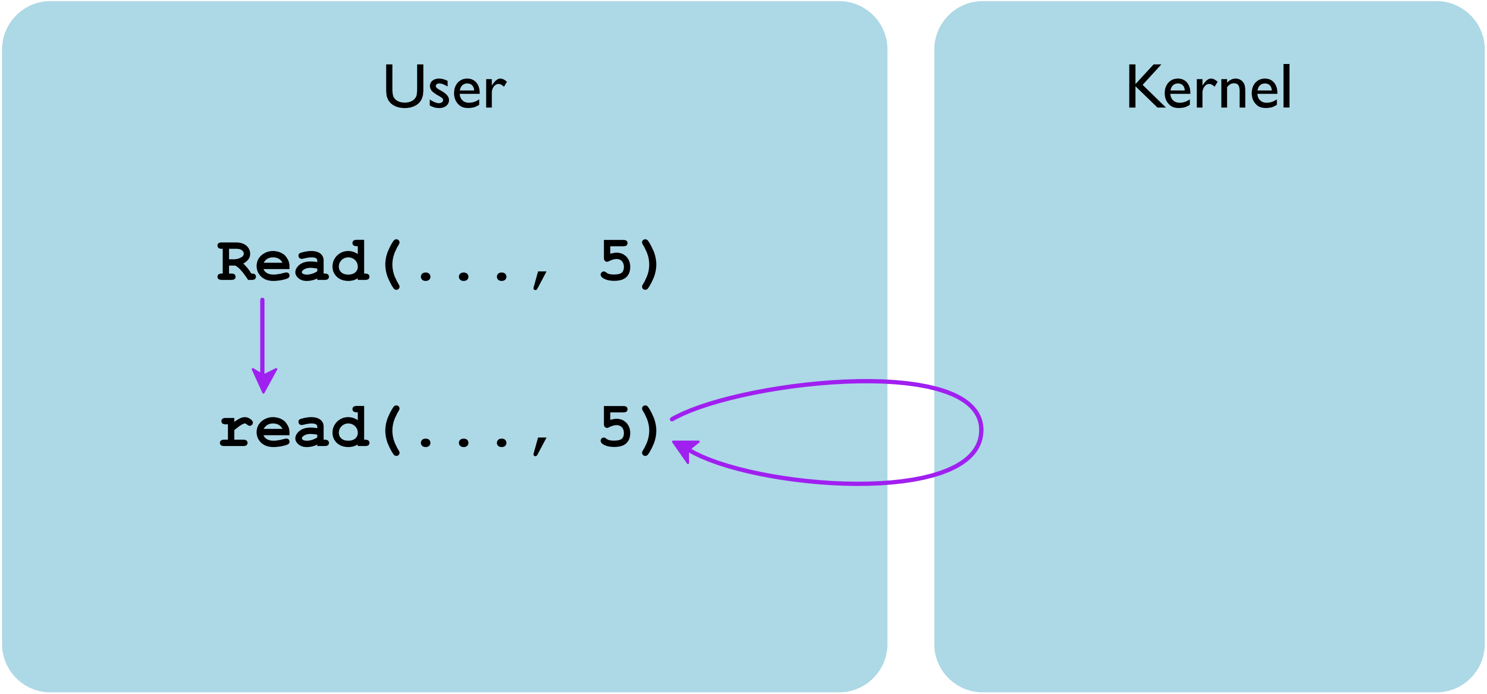
User

`Read(..., 5)`

`read(..., 5)`

Kernel

Unix I/O vs. C Library I/O



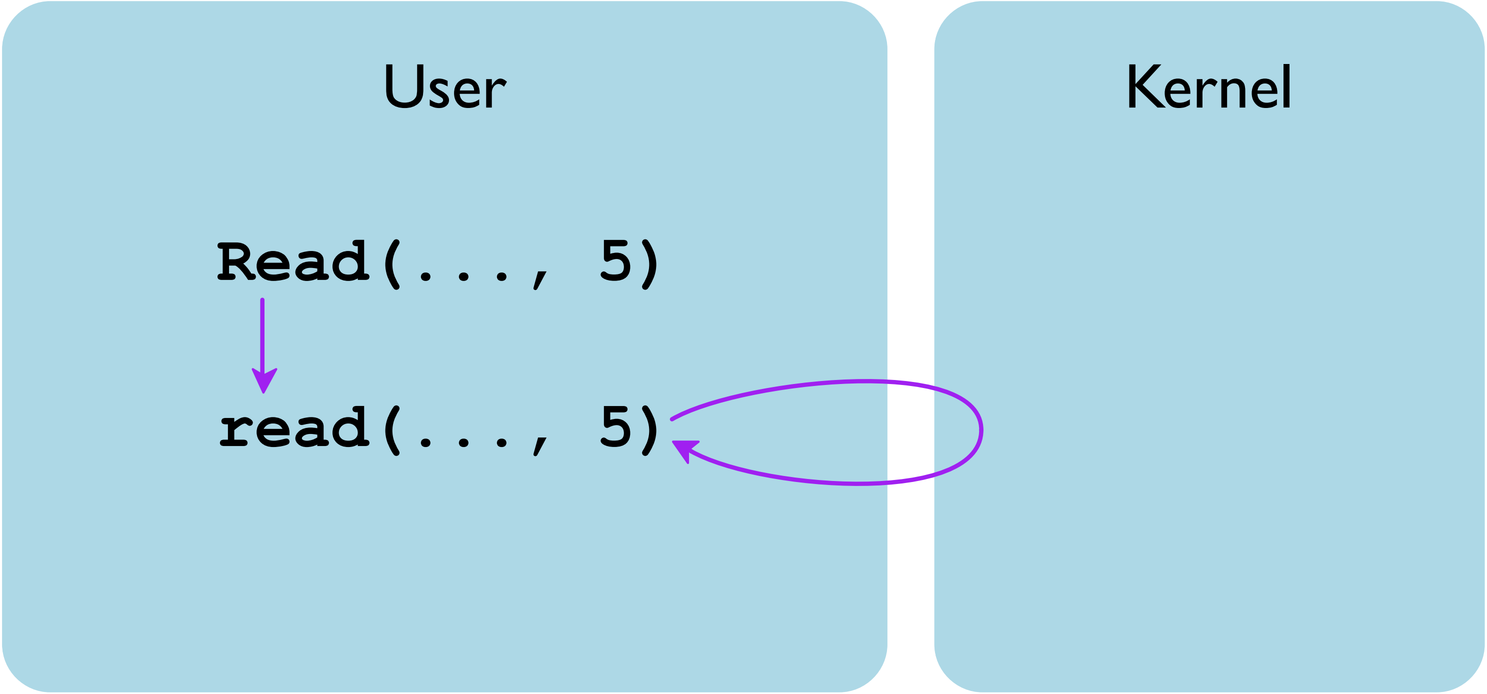
User

Kernel

`Read(..., 5)`

`read(..., 5)`

Unix I/O vs. C Library I/O



User

Kernel

`Read(..., 5)`

`read(..., 5)`

System call through kernel every time

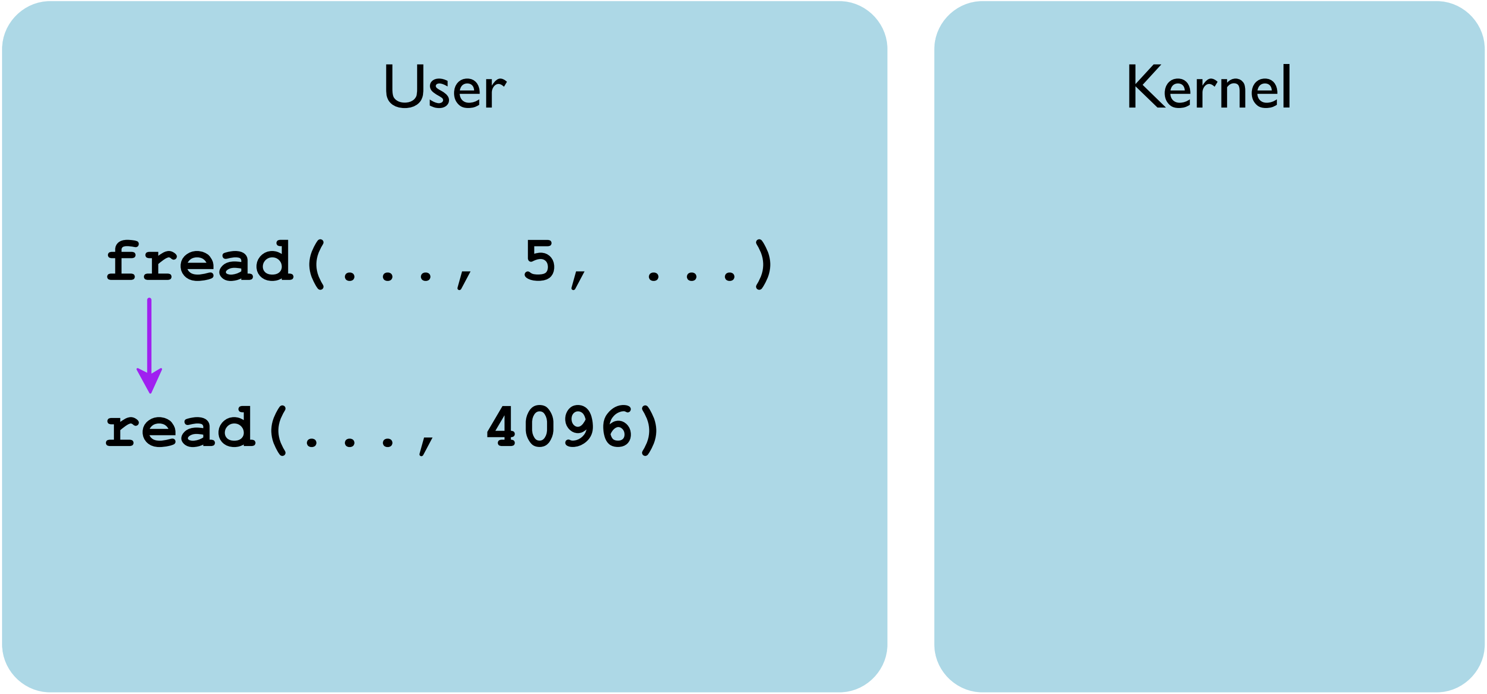
Unix I/O vs. C Library I/O

User

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

Kernel

Unix I/O vs. C Library I/O

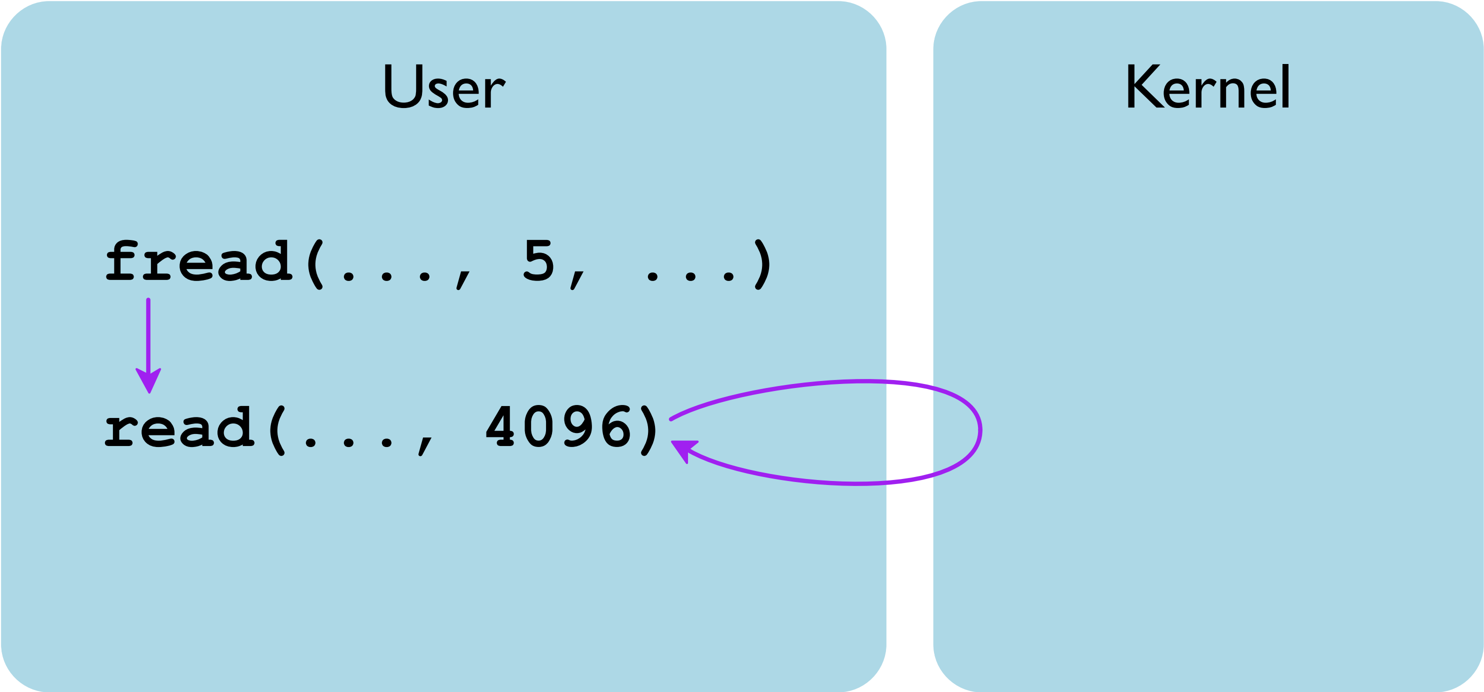


User

Kernel

```
fread(..., 5, ...)
read(..., 4096)
```

Unix I/O vs. C Library I/O



User

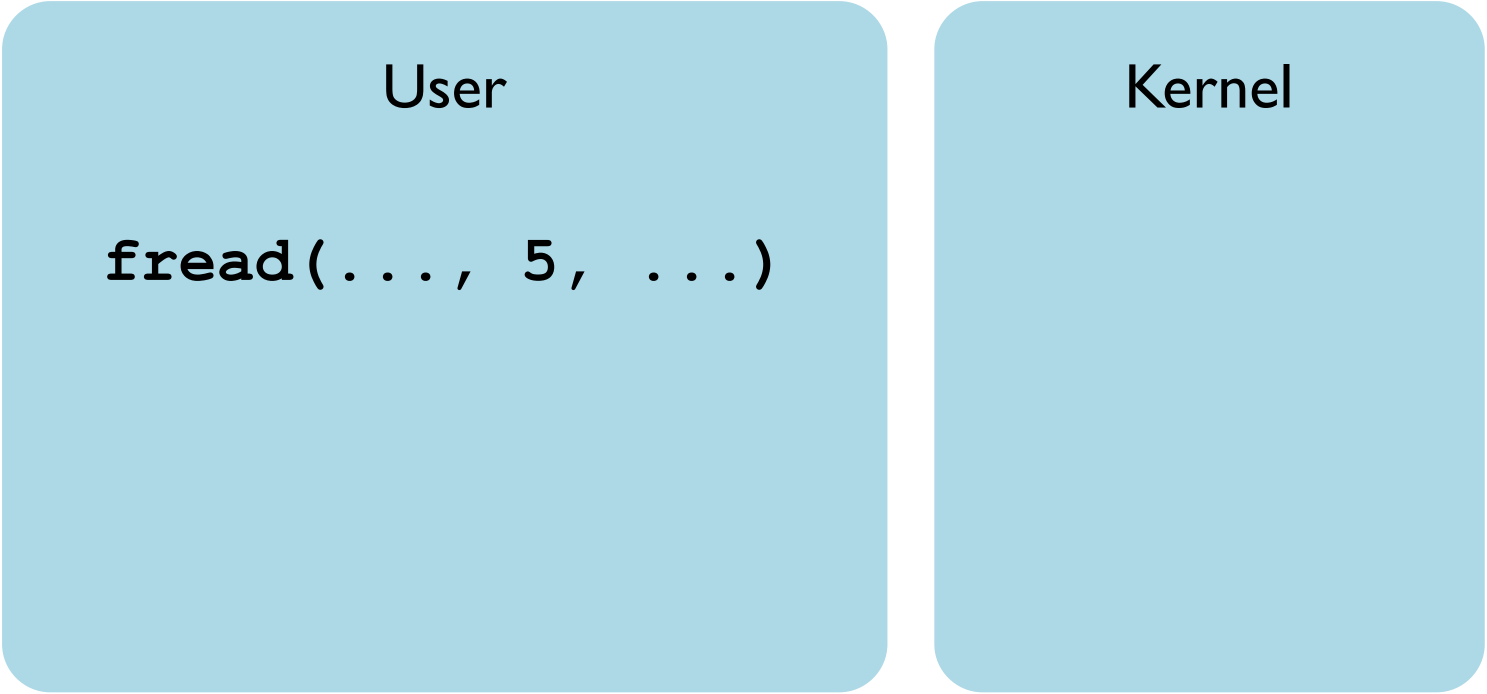
Kernel

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

`read(..., 4096)`

Extra bytes are stored in the **`FILE`** record

Unix I/O vs. C Library I/O



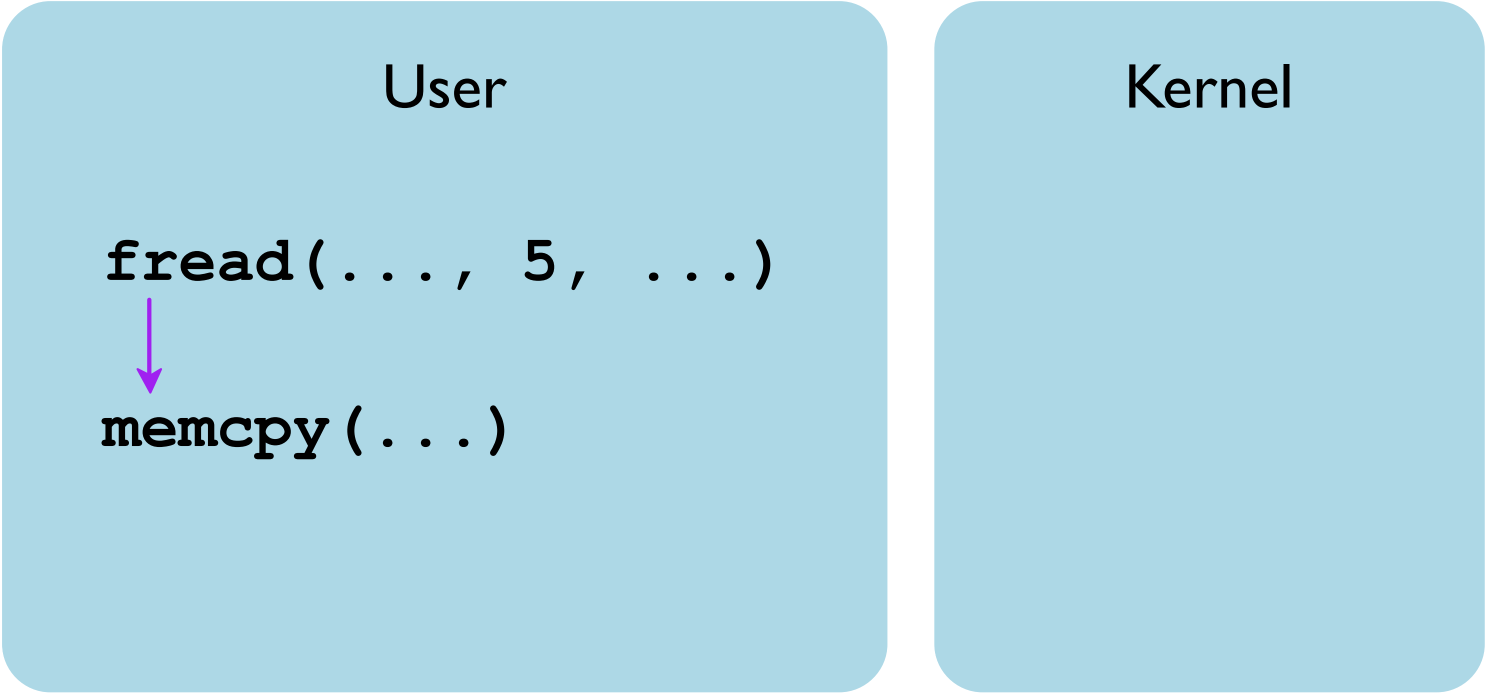
User

Kernel

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

Extra bytes are stored in the **`FILE`** record

Unix I/O vs. C Library I/O



User

Kernel

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

`memcpy(...)`

Extra bytes are stored in the **`FILE`** record

Fast when buffered bytes are available

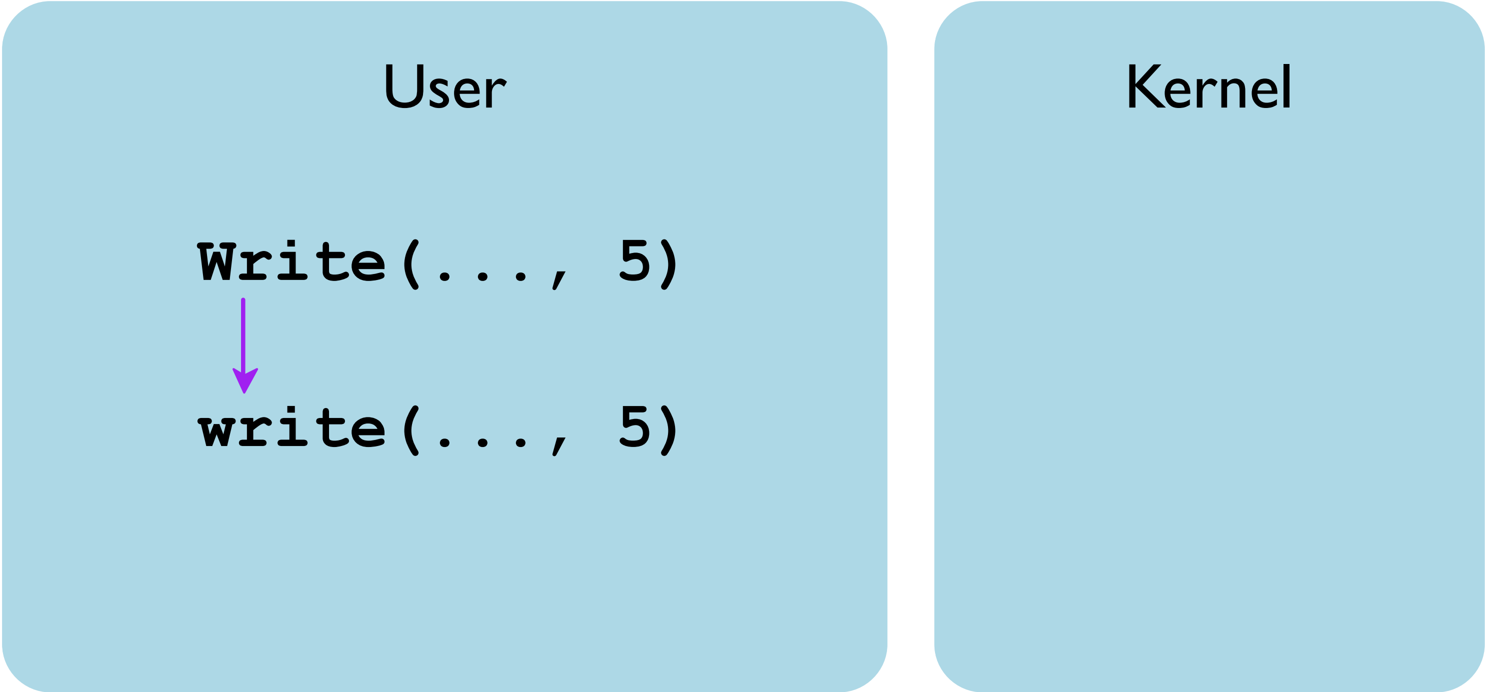
Unix I/O vs. C Library I/O

User

```
Write(..., 5)
```

Kernel

Unix I/O vs. C Library I/O

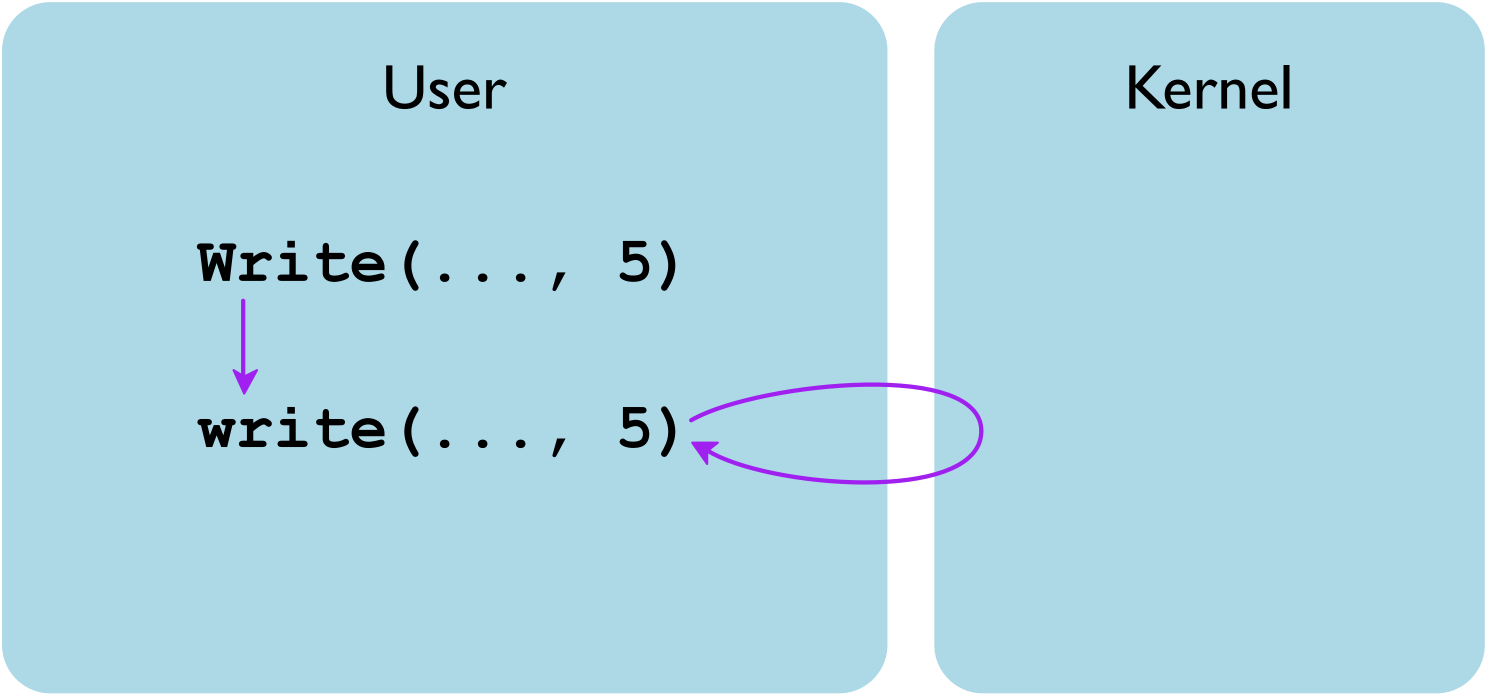


User

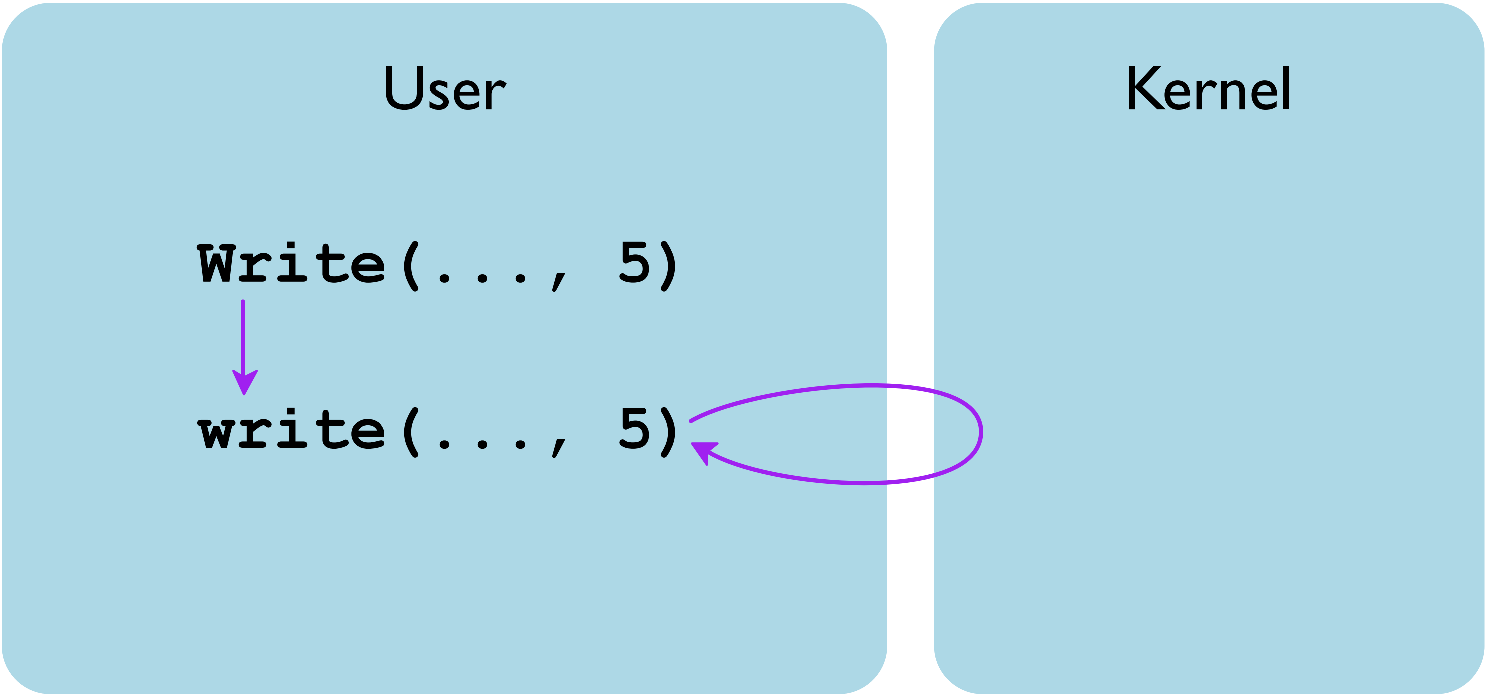
Kernel

```
Write(..., 5)
write(..., 5)
```

Unix I/O vs. C Library I/O



Unix I/O vs. C Library I/O



System call through kernel every time

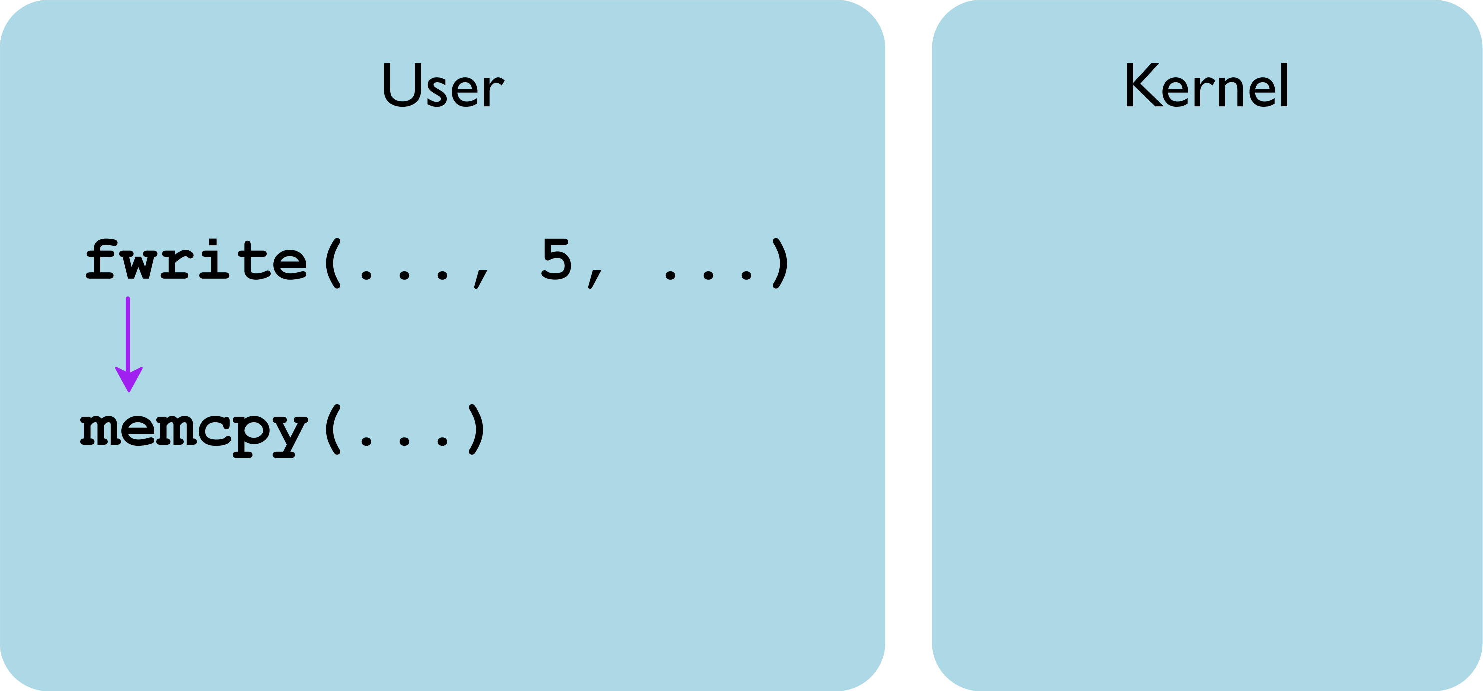
Unix I/O vs. C Library I/O

User

`fwrite(..., 5, ...)`

Kernel

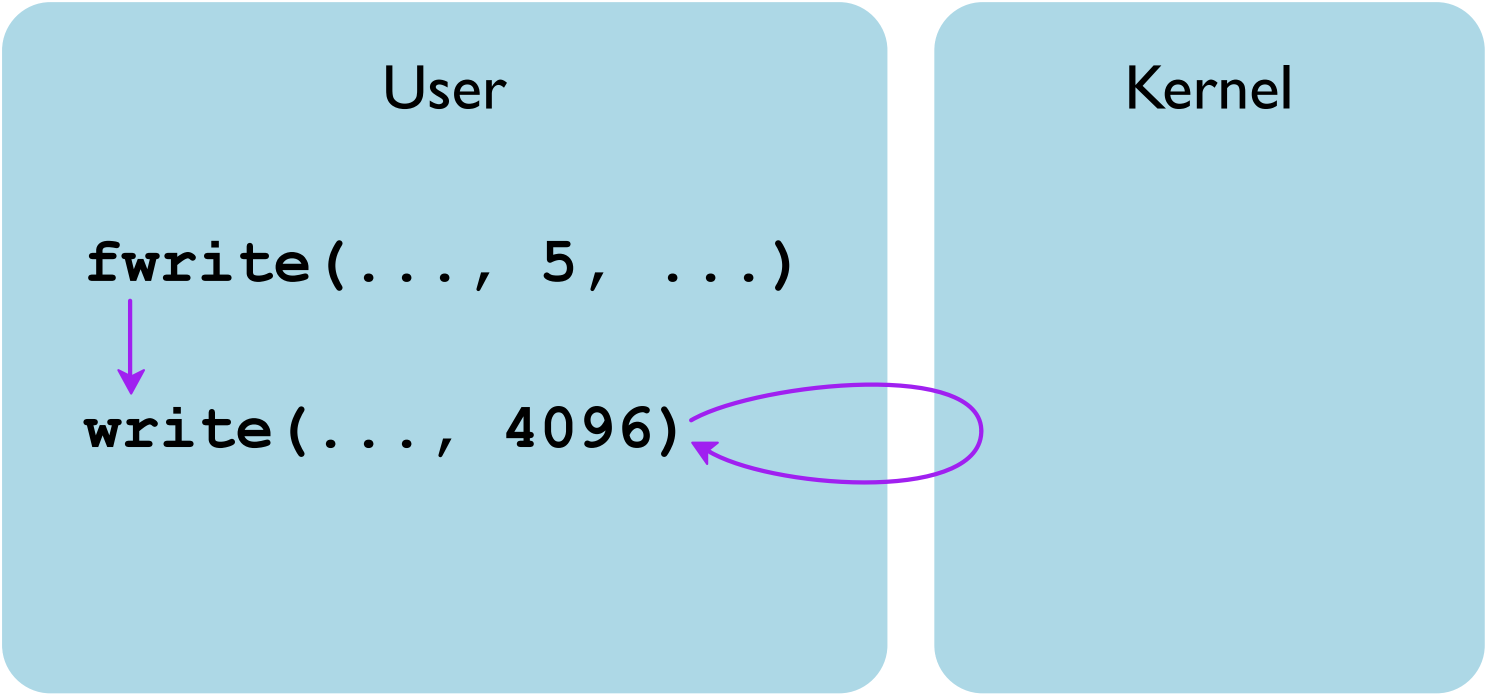
Unix I/O vs. C Library I/O



Written bytes are stored in the **`FILE`** record

Fast when buffer space is available

Unix I/O vs. C Library I/O



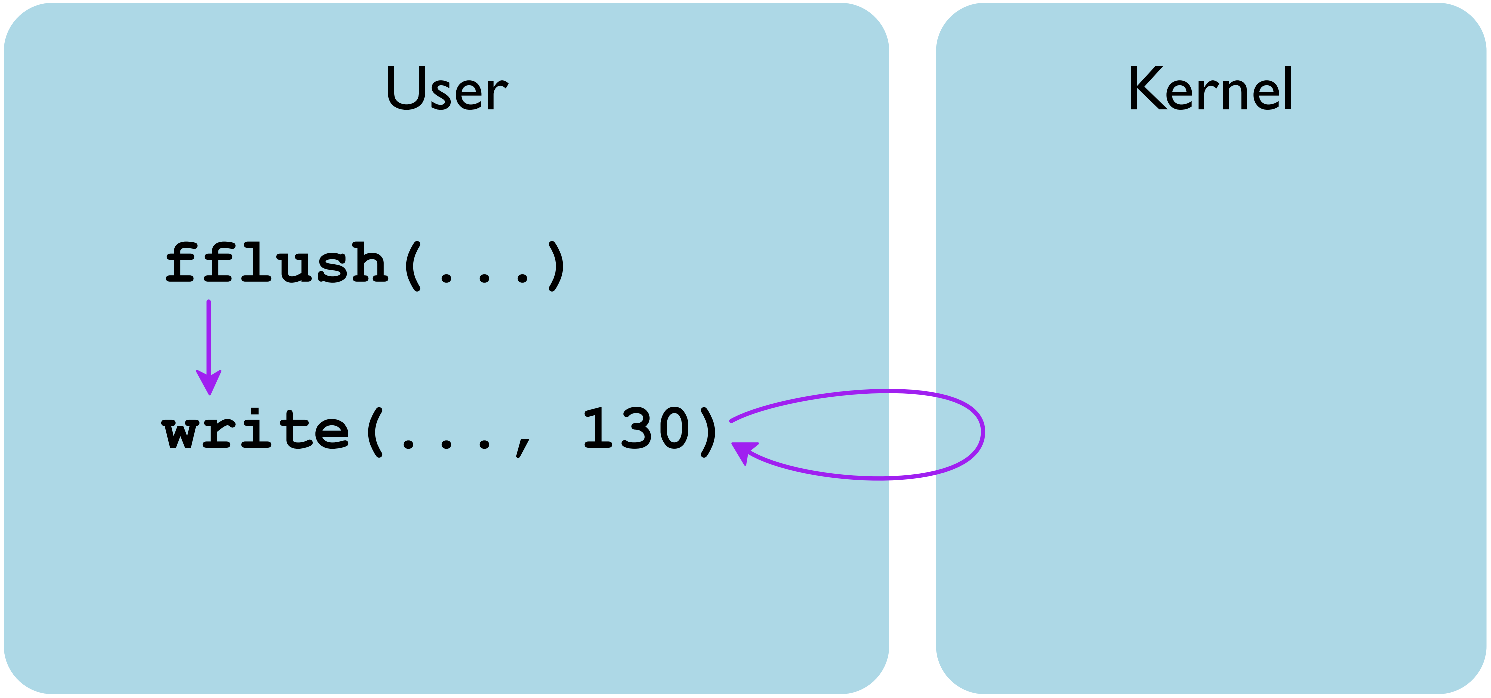
```
fwrite(..., 5, ...)
```

```
write(..., 4096)
```

Written bytes are stored in the **`FILE`** record

Bytes are flushed when the buffer is full

Unix I/O vs. C Library I/O



User

Kernel

`fflush(...)`

`write(..., 130)`

Written bytes are stored in the **`FILE`** record

Explicit flush also writes

Output Buffer Modes

Automatic flushes depend on the buffer mode

- ***Unbuffered*** — flush on every write
- ***Block buffered*** — flush when out of space
- ***Line buffered*** — flush when writing newline

```
printf("Hello\n");
```

Output Buffer Modes

Automatic flushes depend on the buffer mode

- ***Unbuffered*** — flush on every write
- ***Block buffered*** — flush when out of space
- ***Line buffered*** — flush when writing newline

Default buffer mode? **It depends**

- **`stderr`**: unbuffered
- terminal output: line buffered

determined by **`isatty()`**

- anything else: block buffered

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