Accessing Files

Chris Brown

In This Module …

Unbuffered I/O

Sequential access

Random access

Using the standard library

Buffered I/O

Formatted I/O

Advanced Techniques

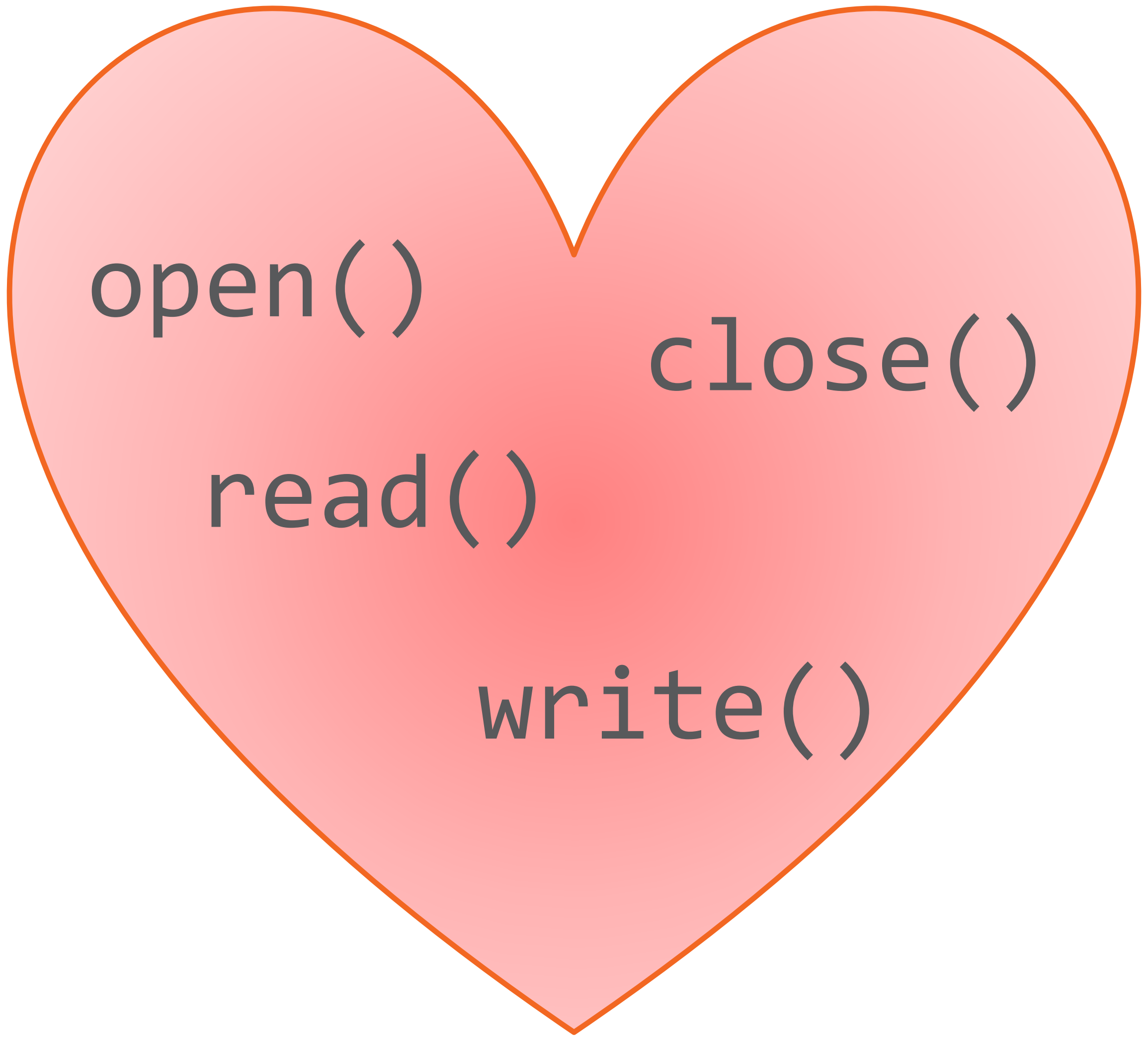
Scatter/gather I/O

Mapping files into memory

Demonstration:

Four ways to copy a file

The Heart of the Matter



Unbuffered I/O

Short of crawling out over the disc with a tiny magnet, these system calls are the lowest level of input/output in Linux

Opening a File

Pathname of the file to be opened

Access mode of the file (only used if the file is being created)

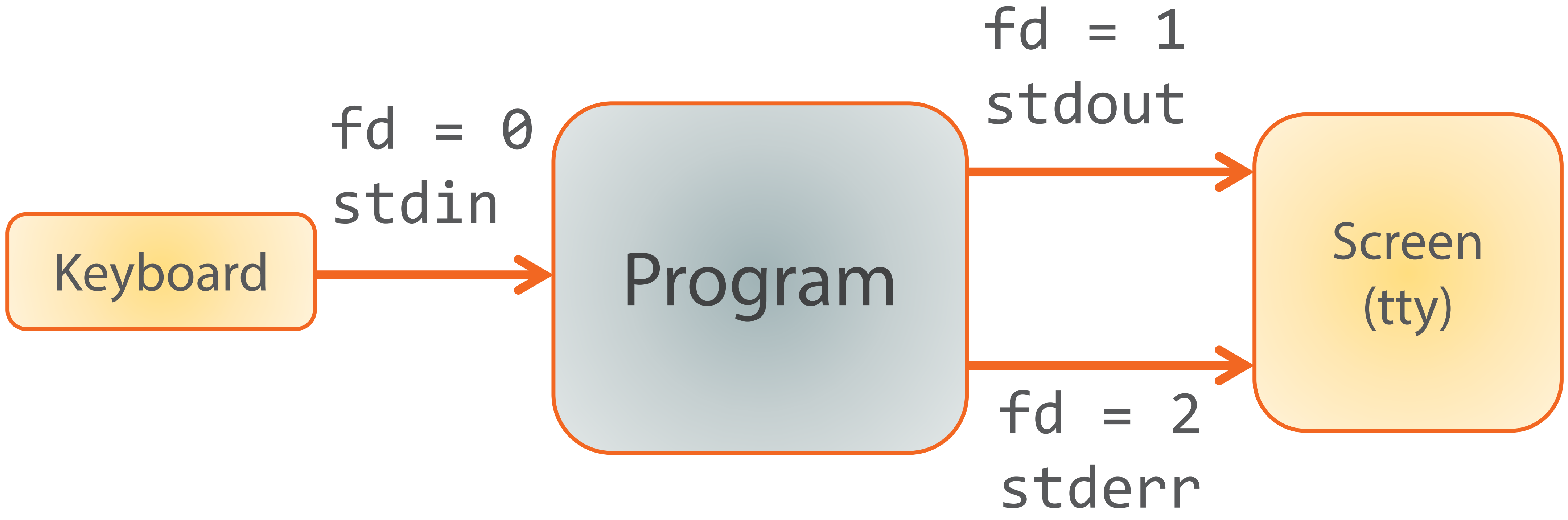
```
fd = open(name, flags, mode)
```

Returns lowest available descriptor

Must include one of:
`O_RDONLY, O_WRONLY, O_RDWR`

May include one or more of:
`O_APPEND, O_CREAT, O_TRUNC`

Standard Streams



Using and Combining Symbolic Constants

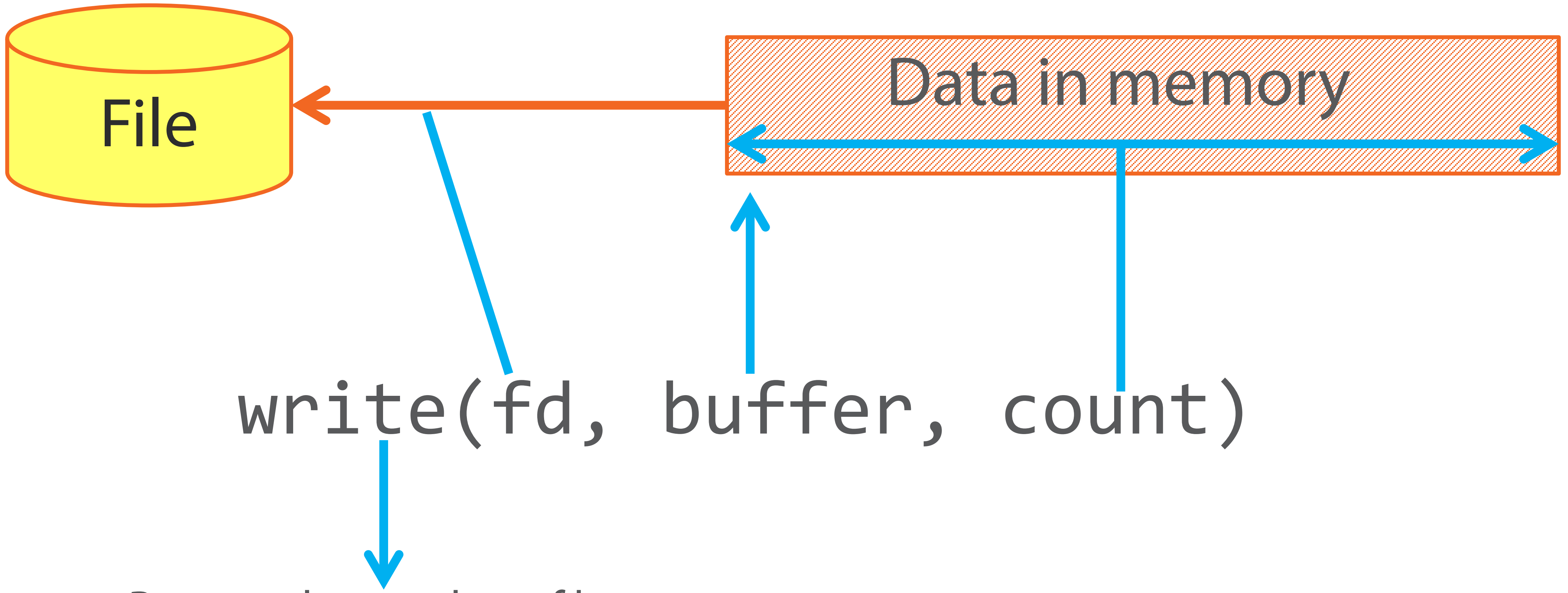
- Some system calls accept flag arguments, specified using symbolic constants
- Some are integer constants (1, 2, 3, 4, …)
  - These are mutually exclusive (you must specify exactly one)
- Some are single-bit values, e.g.:

```
#define O_CREAT    0100
#define O_TRUNC   01000
#define O_APPEND  02000
```

These flags may be combined using a bitwise 'OR'

```
fd = open("foo", O_RDWR | O_TRUNC | O_APPEND);
```

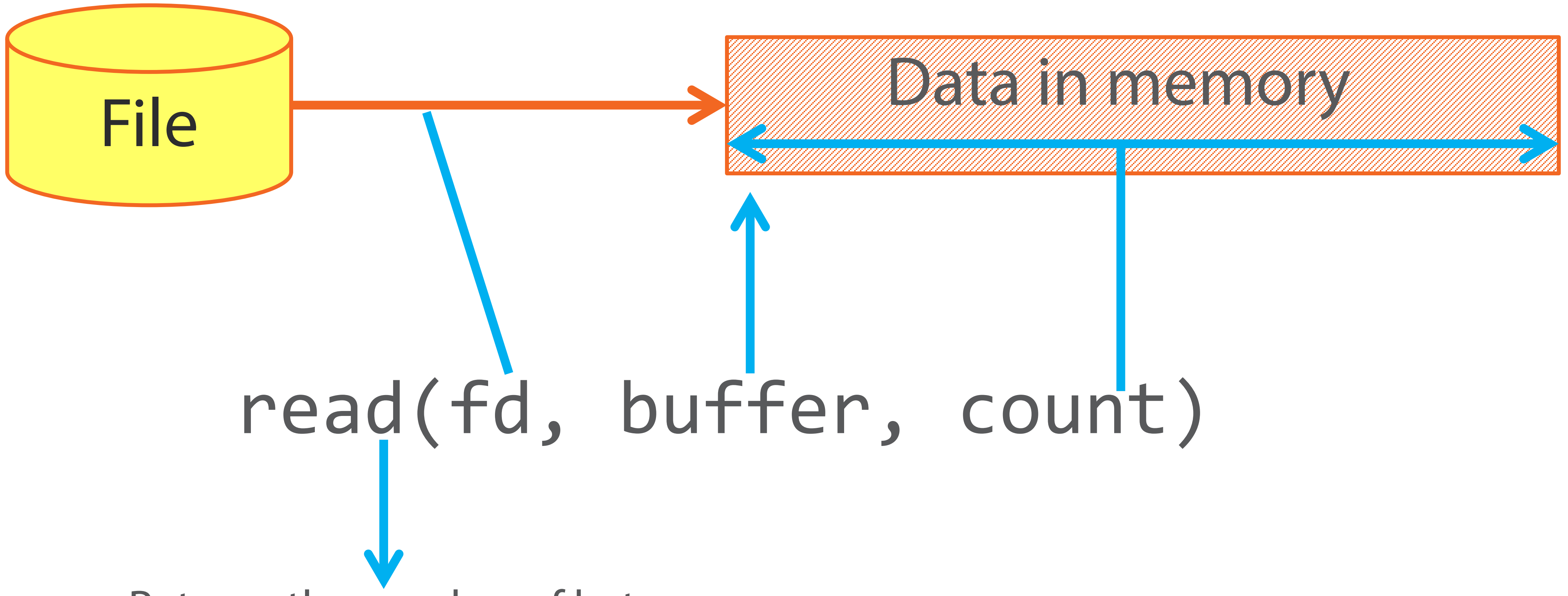
Unbuffered Output



```
write(fd, buffer, count)
```

Returns the number of bytes actually written (-1 on error)

Unbuffered Input



```
read(fd, buffer, count)
```

Returns the number of bytes actually read (0 on end-of-file)

Closing a File

An open file descriptor

```
close(fd)
```

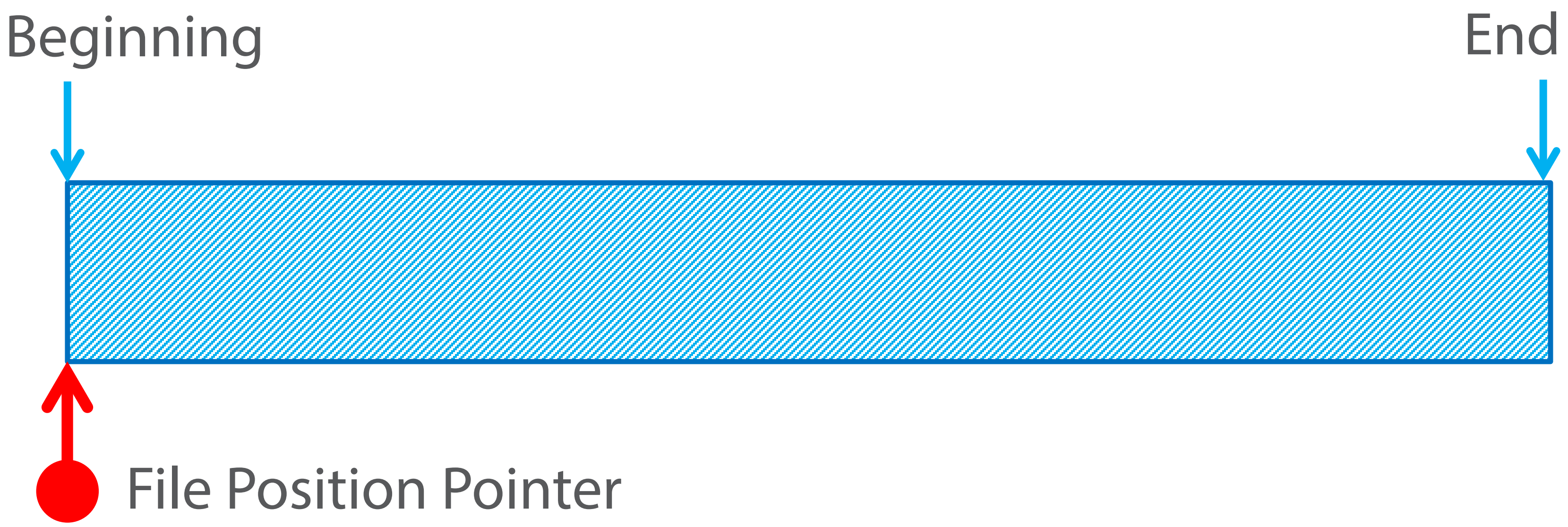
Closes the descriptor

Makes it available for re-use

Descriptors are implicitly closed when a process terminates

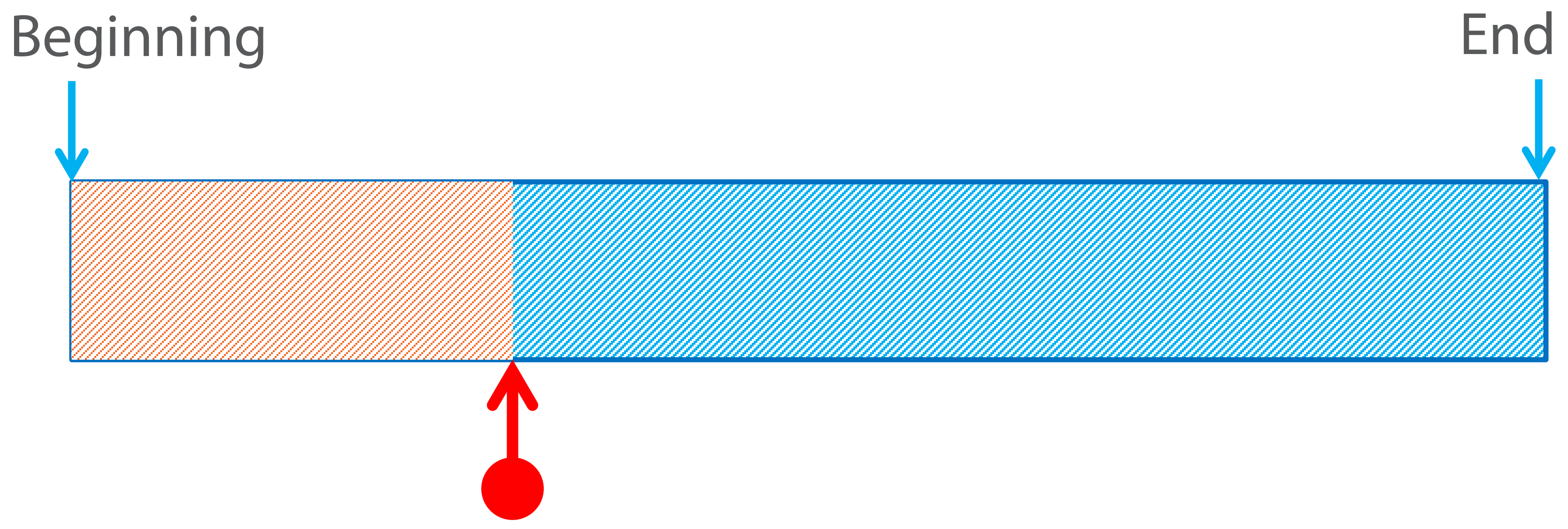
There is a finite limit on how many descriptors a process can have open

Sequential Access



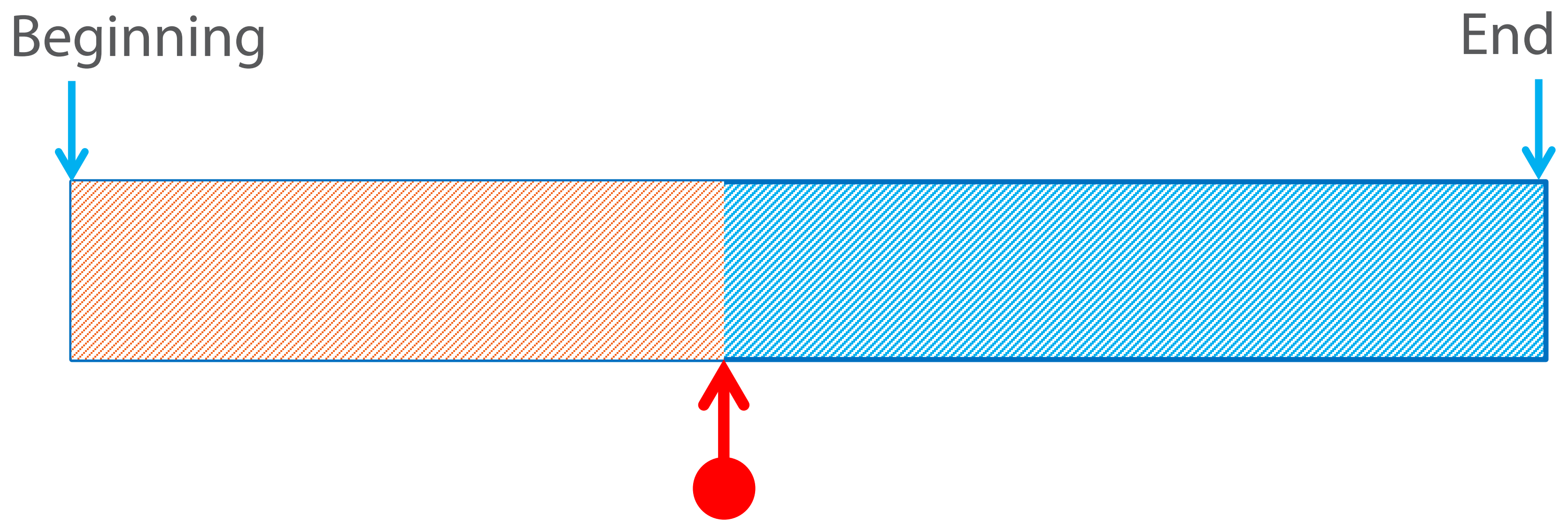
```
read(fd, buffer, 1200)
```

Sequential Access



```
read(fd, buffer, 1200);

read(fd, buffer, 600);
```

Sequential Access



```
read(fd, buffer, 1200);
read(fd, buffer, 600);
```

Random Access

The file position pointer may be explicitly repositioned:

```
lseek(fd, offset, whence)
```

File descriptor

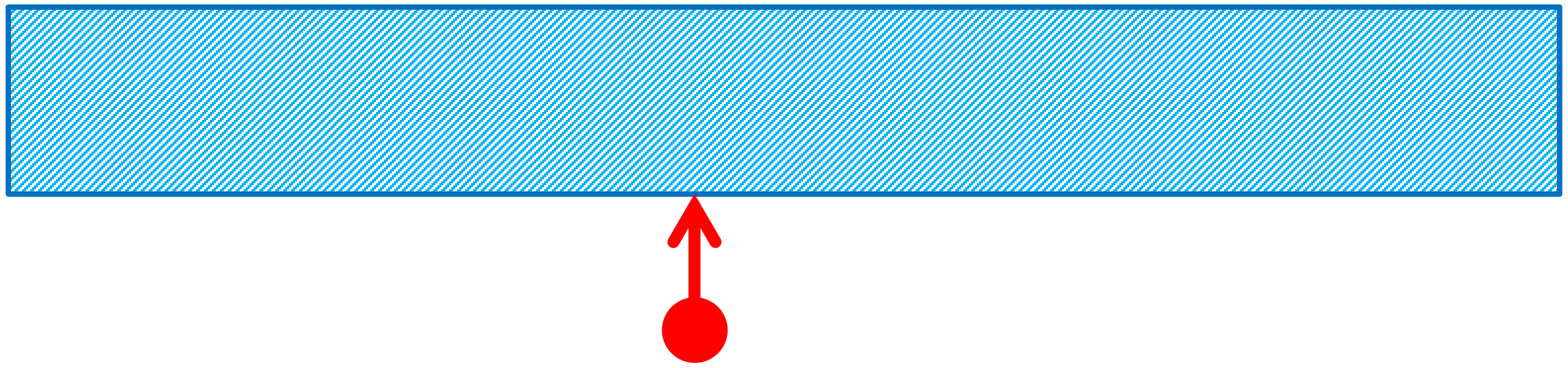
Byte offset. May be positive or negative.

Specifies where the offset is relative to:

SEEK_SET	Relative to start of file
SEEK_CUR	Relative to current position
SEEK_END	Relative to end of file

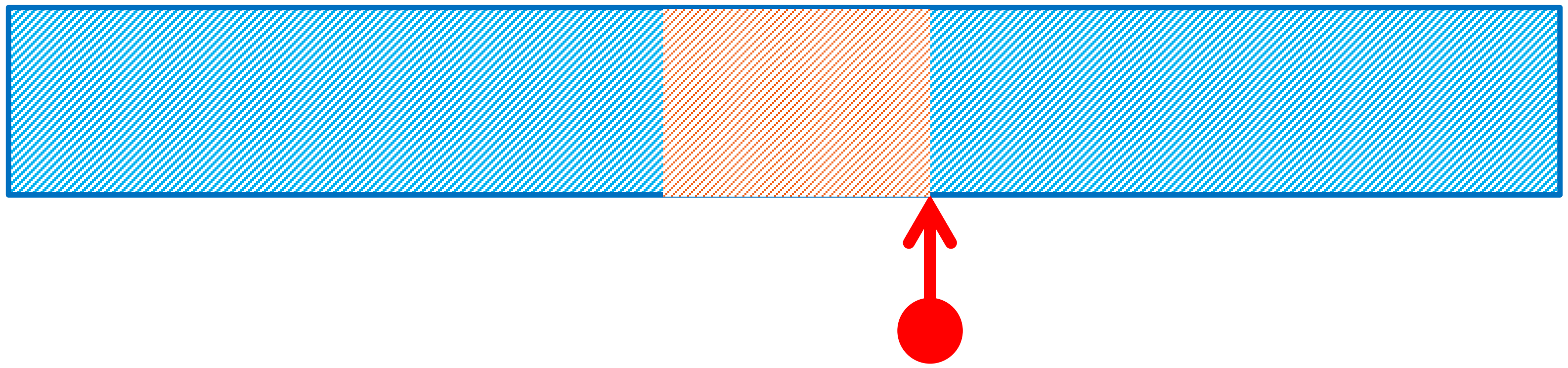
Random Access Examples

Before



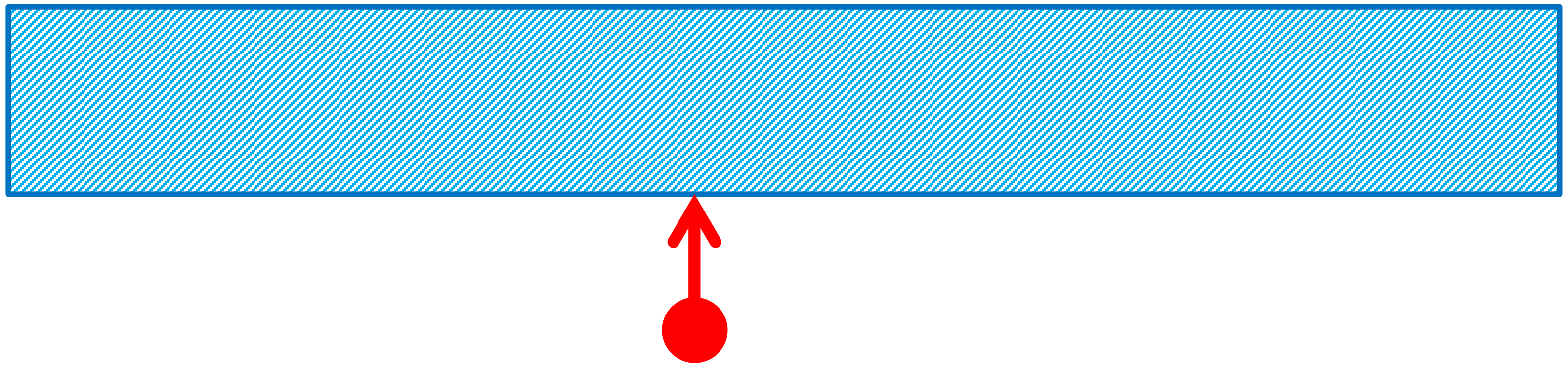
```
lseek(fd, 100, SEEK_CUR);
```

After



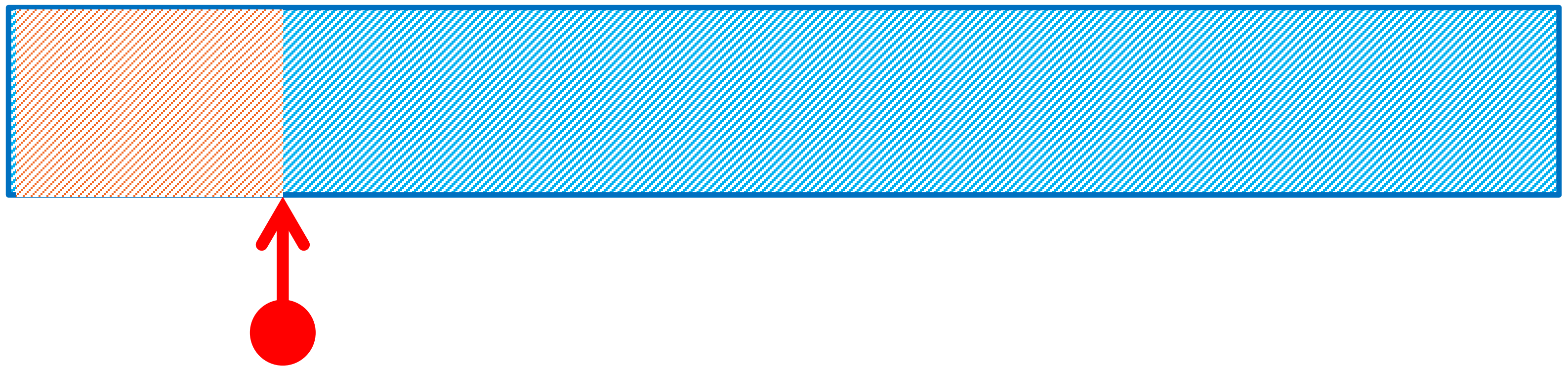
Random Access Examples

Before



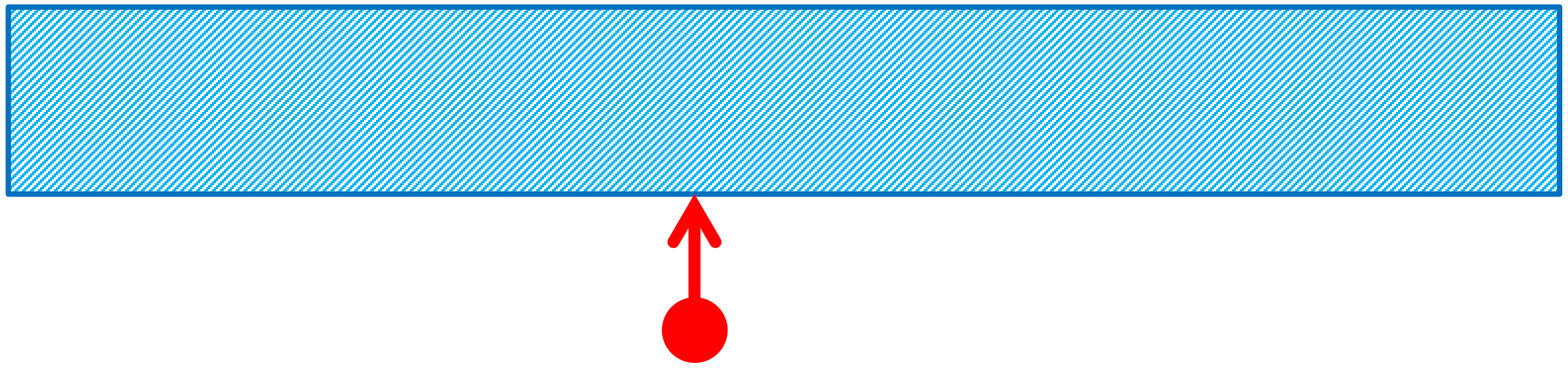
```
lseek(fd, 100, SEEK_SET);
```

After



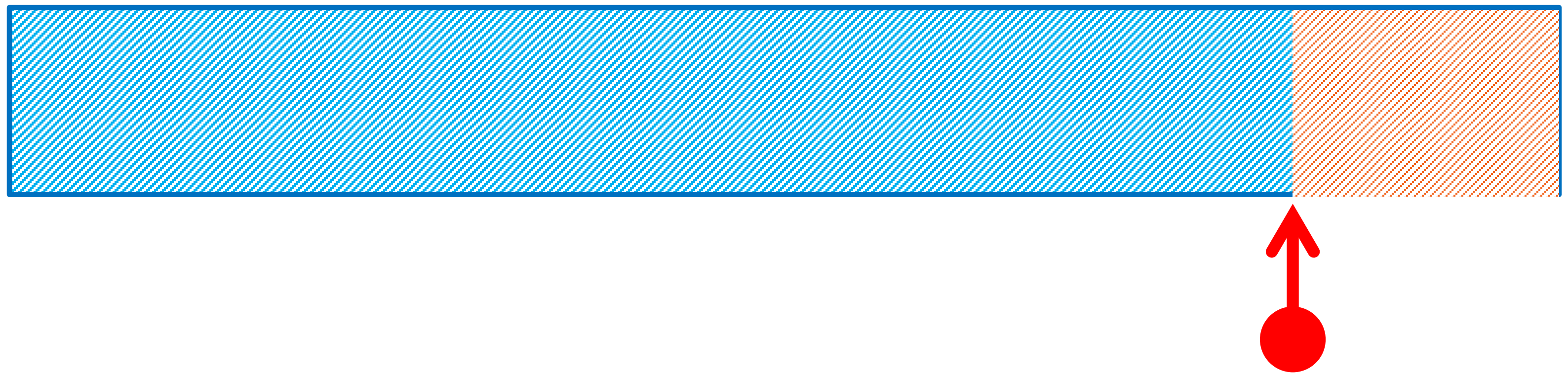
Random Access Examples

Before

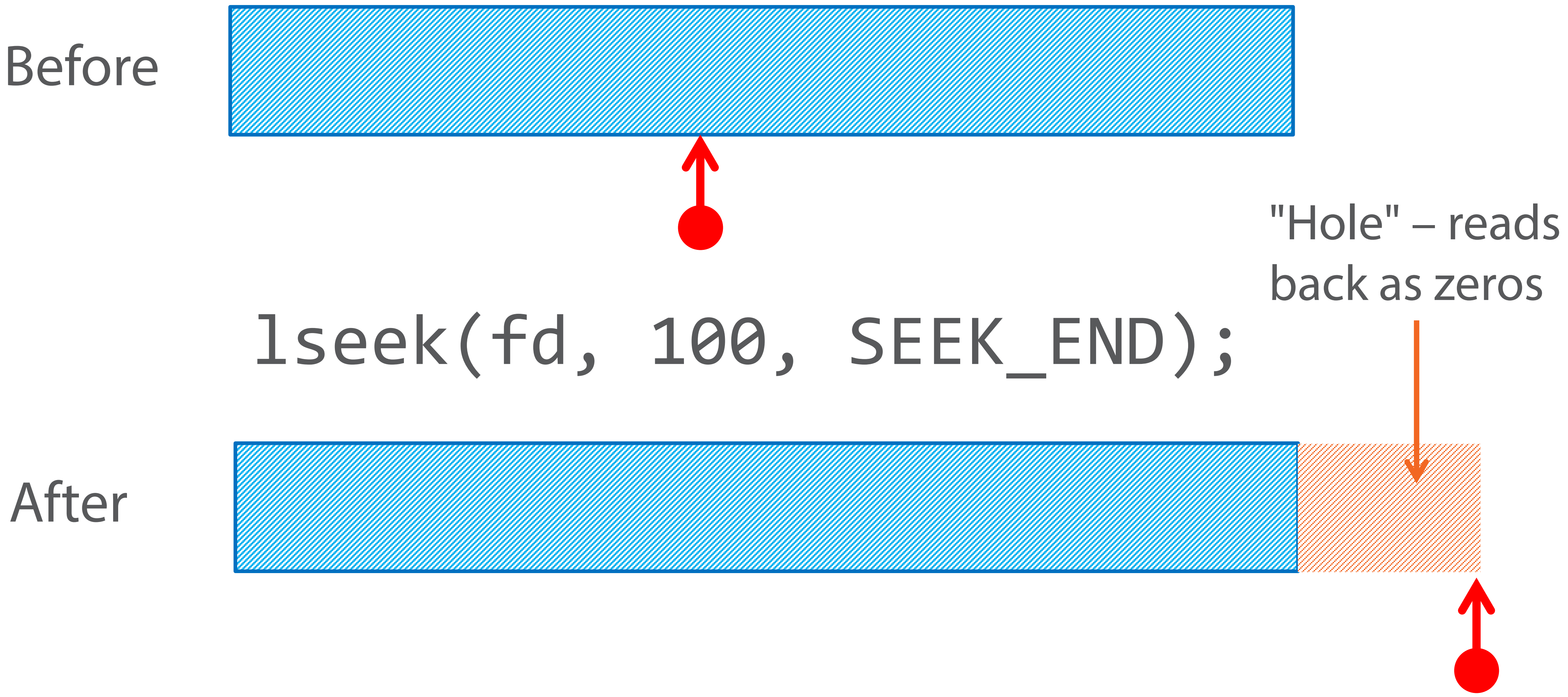


```
lseek(fd, -100, SEEK_END);
```

After



Random Access Examples



```
lseek(fd, 100, SEEK_END);
```

Random Access Example

```
#include <unistd.h>
#include <fcntl.h>
struct record {                                 /* Define a "record" */
    int id;
    char name[80];
};

void main()
{
    int fd, size = sizeof(struct record);
    struct record info;

    fd = open("datafile", O_RDWR);   /* Open for read/write */
```

Random Access Example

```
    lseek(fd, size, SEEK_SET);  /* Skip one record */
    read(fd, &info, size);      /* Read second record */

    info.id = 99;               /* Modify record */
    lseek(fd, -size, SEEK_CUR); /* Backspace */
    write(fd, &info, size);     /* Write modified record */

    close(fd);
}
```

File IO and the Standard C Library

The Standard C library also specifies file IO routines

- Buffered
- Available on any conformant "C" environment

Opening a File

Pathname of the
file to be opened

```
fd = fopen(name, mode)
```

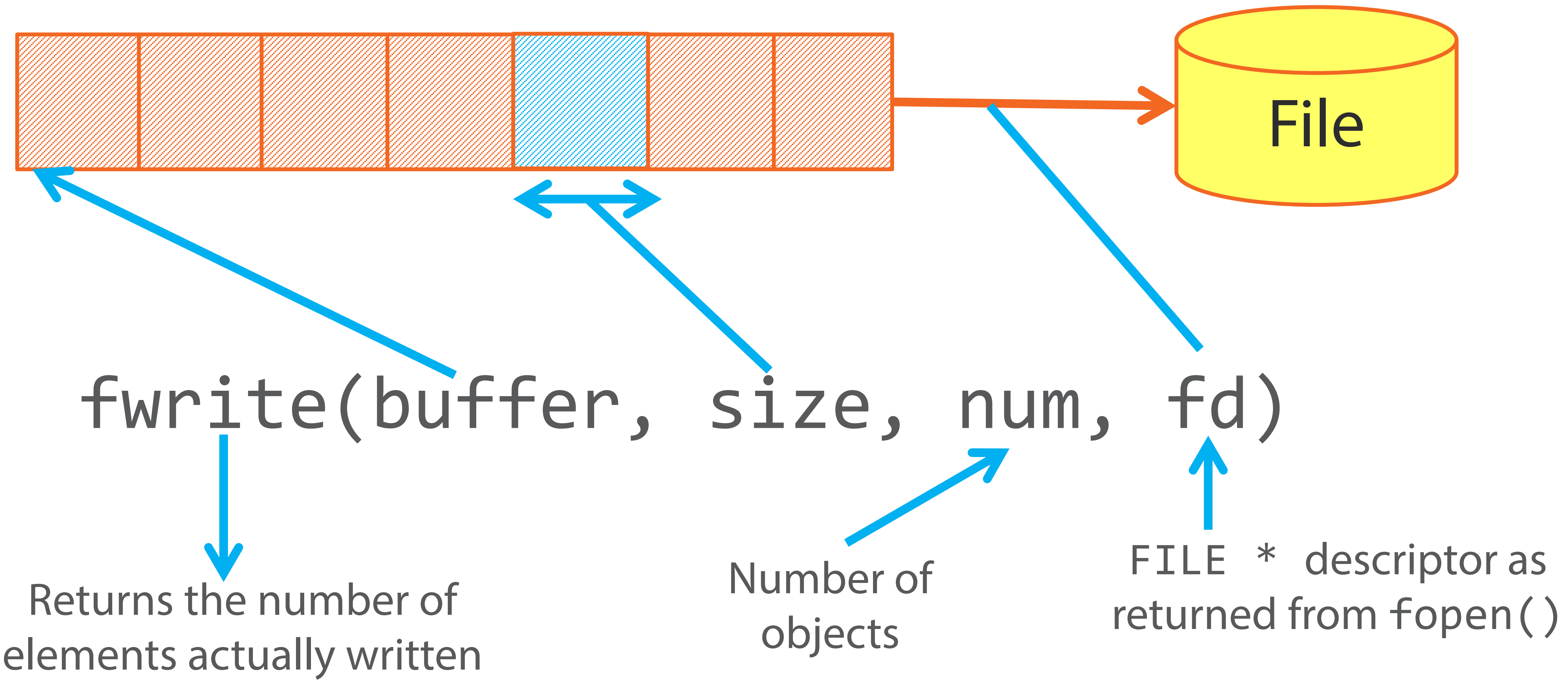
Returns a descriptor
of type `FILE *`
(or `NULL` on error)

Valid modes include:

`"r"`	open text file for reading
`"w"`	truncate and open for writing
`"r+"`	open text file for update

Append `"b"` to the mode for binary files

Output



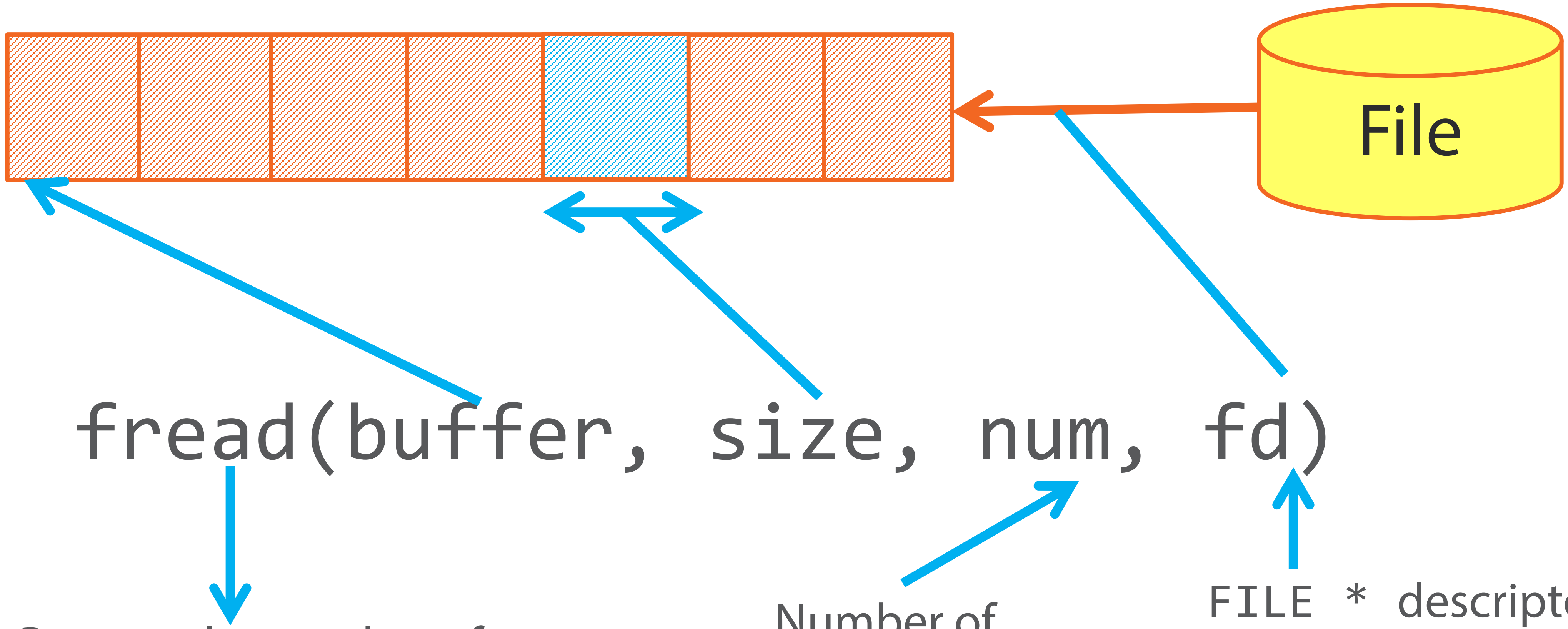
```
fwrite(buffer, size, num, fd)
```

Returns the number of elements actually written

Number of objects

`FILE *` descriptor as returned from `fopen()`

Input



```
fread(buffer, size, num, fd)
```

Returns the number of elements actually read

Number of objects

`FILE *` descriptor as returned from `fopen()`

Closing a File

An open file descriptor

↓

```
fclose(fd)
```

Closes the descriptor

Flushes any buffered data

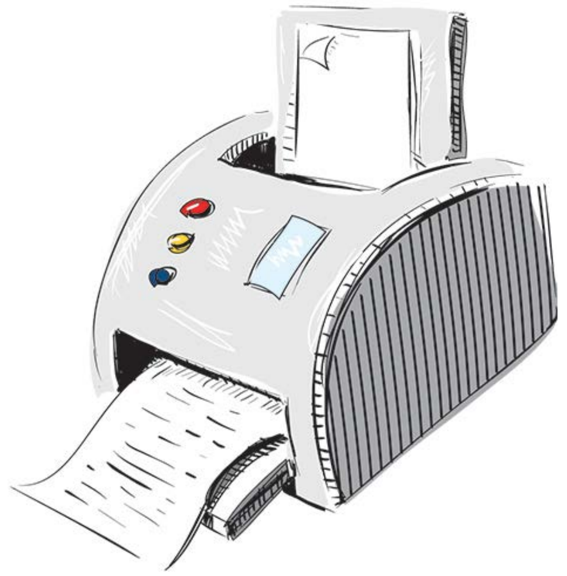
Descriptors are implicitly closed when a process terminates

There is a finite limit on how many descriptors a process can have open

So What's the Difference?

Feature	Low-level IO	Standard Library IO
Read/write access	`open(), close(), read(), write()`	`fopen(), fclose(), fread(), fwrite()`
Random access	`lseek()`	`fseek(), rewind()`
Type of descriptor	`int`	`FILE *`
User-space buffering?	No	Yes
Part of C standard?	No	Yes

Formatted IO



`printf()` and friends

printf()

- Generates a formatted string and writes it to standard output

```c
char *name = "Sharon";
int age = 45;
double wage = 34500.00;
printf("%12s is %d and earns %f", name, age, wage);
```

printf()

- Generates a formatted string and writes it to standard output

```
char *name = "Sharon";
int age = 45;
double wage = 34500.00;
printf("%12s is %d and earns %f", name, age, wage);
```

printf()

- Generates a formatted string and writes it to standard output

```
char *name = "Sharon";
int age = 45;
double wage = 34500.00;
printf("%12s is %d and earns %f", name, age, wage);
```

Returns the number of characters printed

Other text is treated literally

`printf()` Format Codes

`%d`	decimal integer
`%8d`	… right-justified in 8 character field
`%-8d`	… left justified
`%s`	string
`%12.3f`	double, in 12 character field with 3 digits after the decimal point

See "`man 3 printf`" for the details

printf's Friends and Relations

```
fd = fopen(…);
fprintf(fd, "hello");
```

(Use stderr to write an error message)

```
char[100] buf;
sprintf(buf, "hello")
```

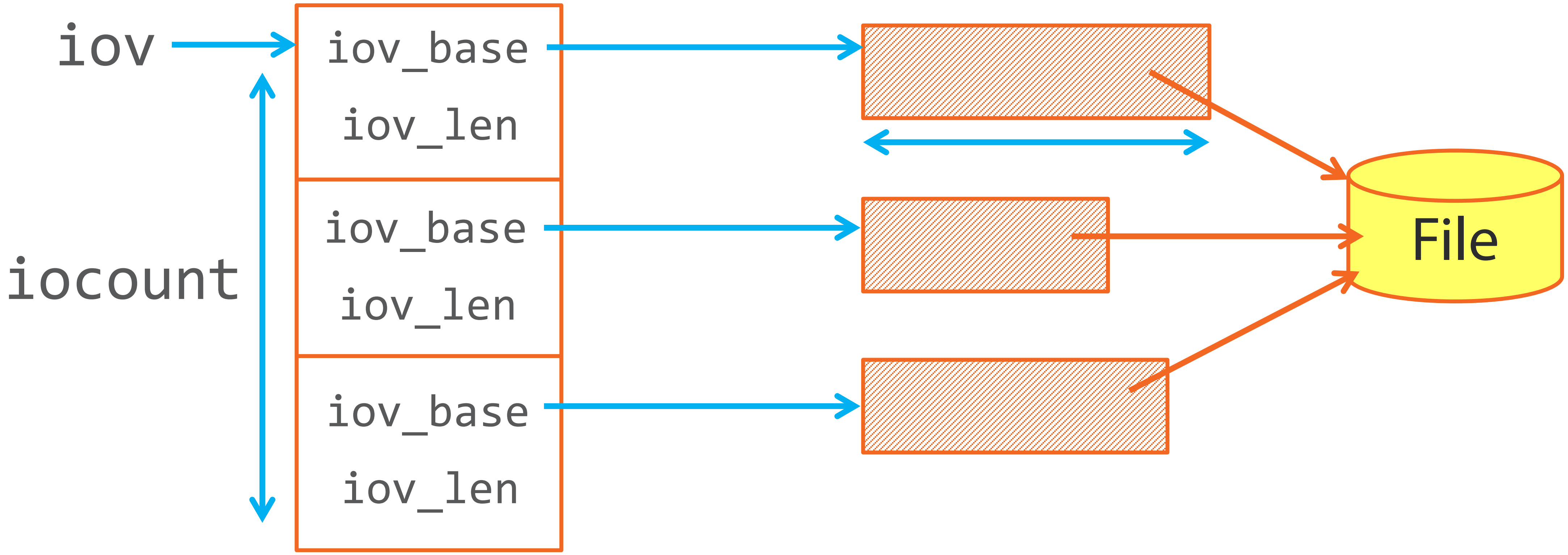
Formats a string into memory

Scatter/Gather IO

- Read or write multiple buffers of data in a single call
- Atomic
- `readv()` and `writev()`

Scatter/Gather IO

```
writev(fd, iov, iocount)
```



Mapping Files into Memory

`mmap()` maps a file into memory and allows you to access it as if it were an array

Mapping Files into Memory

Set this to NULL to allow the kernel to choose the address

PROT_READ
PROT_WRITE

File descriptor from open()

```
mmap(addr, length, prot, flags, fd, offset)
```

The length of the mapping

MAP_SHARED
MAP_PRIVATE

Offset within the file

Returns the address at which the file has been mapped

Random Access Using mmap()

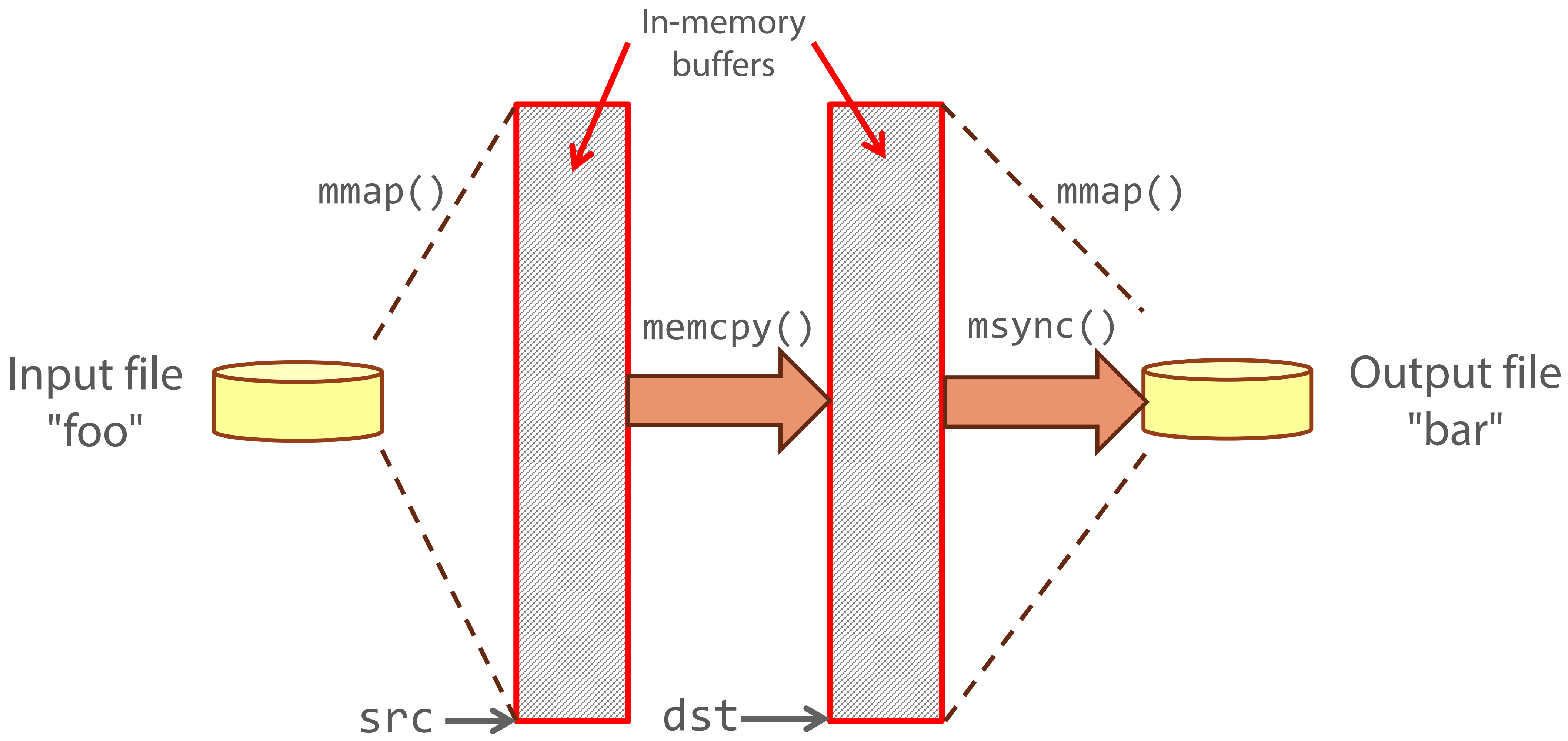
```
#include <sys/mman.h>
#include <fcntl.h>
#include <stdlib.h>

struct record {
  int id;                         /* Define a "record" */
  char name[80];
};

int main()
{
  int fd;
  size_t size;
  struct record *records;         /* Pointer to an array of records */
```

Random Access Using mmap( )

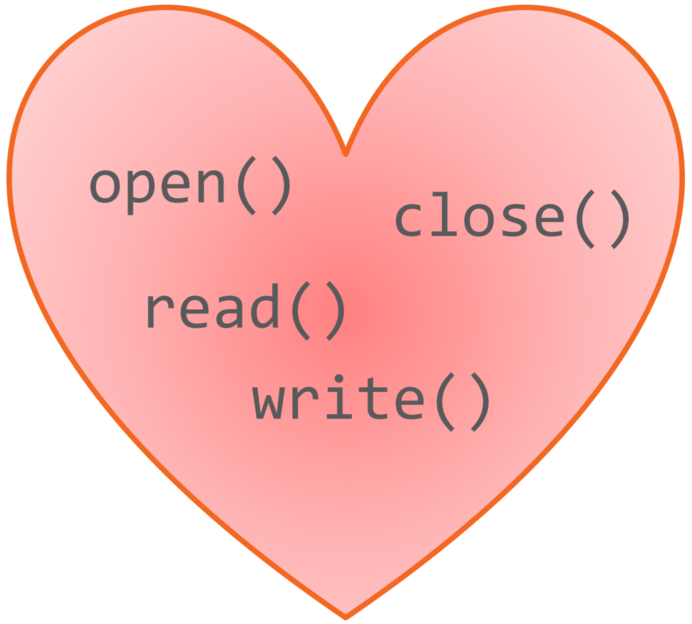
```
  fd = open("foo", O_RDWR);
  size = lseek(fd, 0, SEEK_END);     /* Get  size of file */

  records = (struct record *)mmap(NULL, size, PROT_READ | PROT_WRITE,
                                  MAP_PRIVATE, fd, 0);
  records[1].id = 99;      /* Update record 1 */

  msync(records, size, MS_SYNC);
}
```

Map in the whole file, viewing it as an array of records.

Copying a File Using mmap()



Module Summary



The heart of File IO

Seeking and random access

Buffered IO – `printf()` and friends

Advanced topics:
scatter/gather and memory-mapped IO

Moving Forward …

Coming up in the next module:

File-system management

files, inodes, links and directories