Concurrent Servers and Clients

Chris Brown

In This Module …

The process-per-client concurrent server model

Avoiding zombies

Concurrent servers using `select()`

Maintaining state in single-process servers

The Need for Concurrency

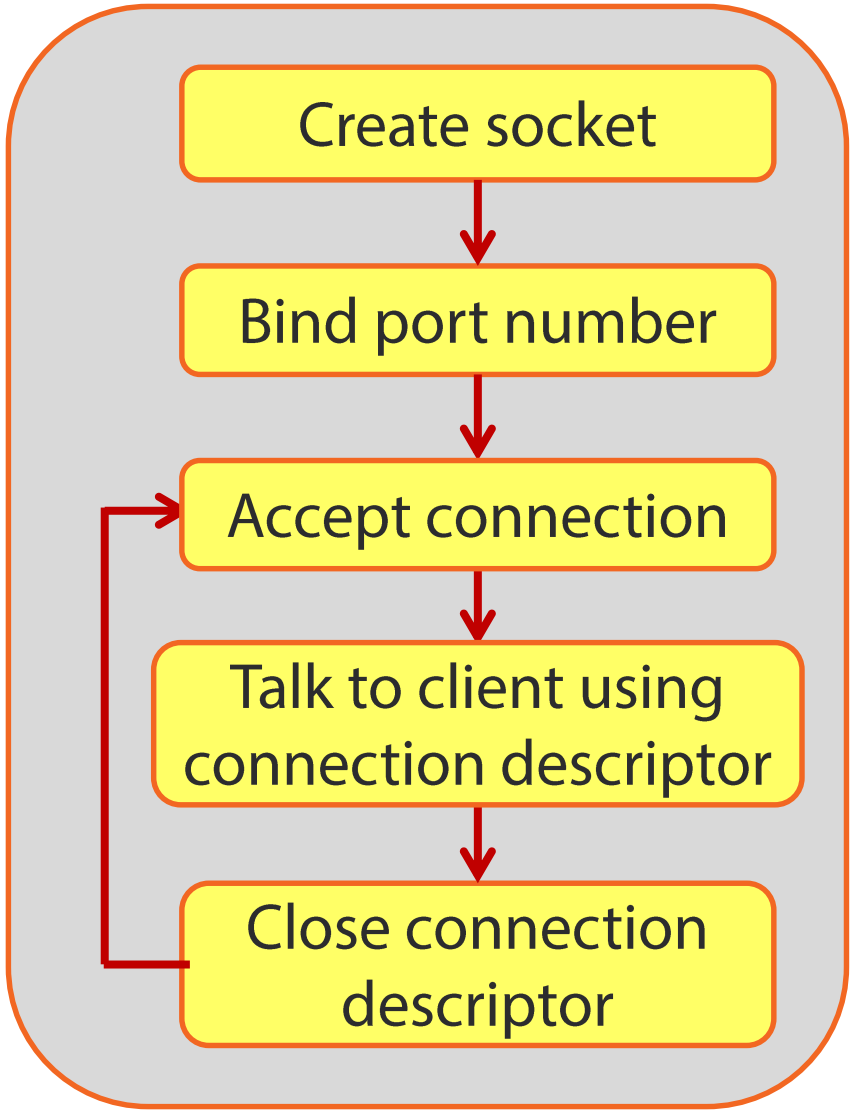
- The TCP-based server we wrote earlier was iterative
  - Completed dialogue with one client before accepting a connection from the next
  - Clients have to wait
  - If the connection queue fills up, connections will be refused
- The servers we will write in this chapter are concurrent
- - Able to conduct dialogues with multiple clients simultaneously

Iterative Server Schema (recap)

```
sock = socket( ... );
bind(sock, ...);
listen(sock, 5);

while(1) {
  fd = accept(sock, ...);
  /* Conduct dialogue with client,
     using fd for input and output
  */
  close(fd);
}
```

Iterative Servers Illustrated

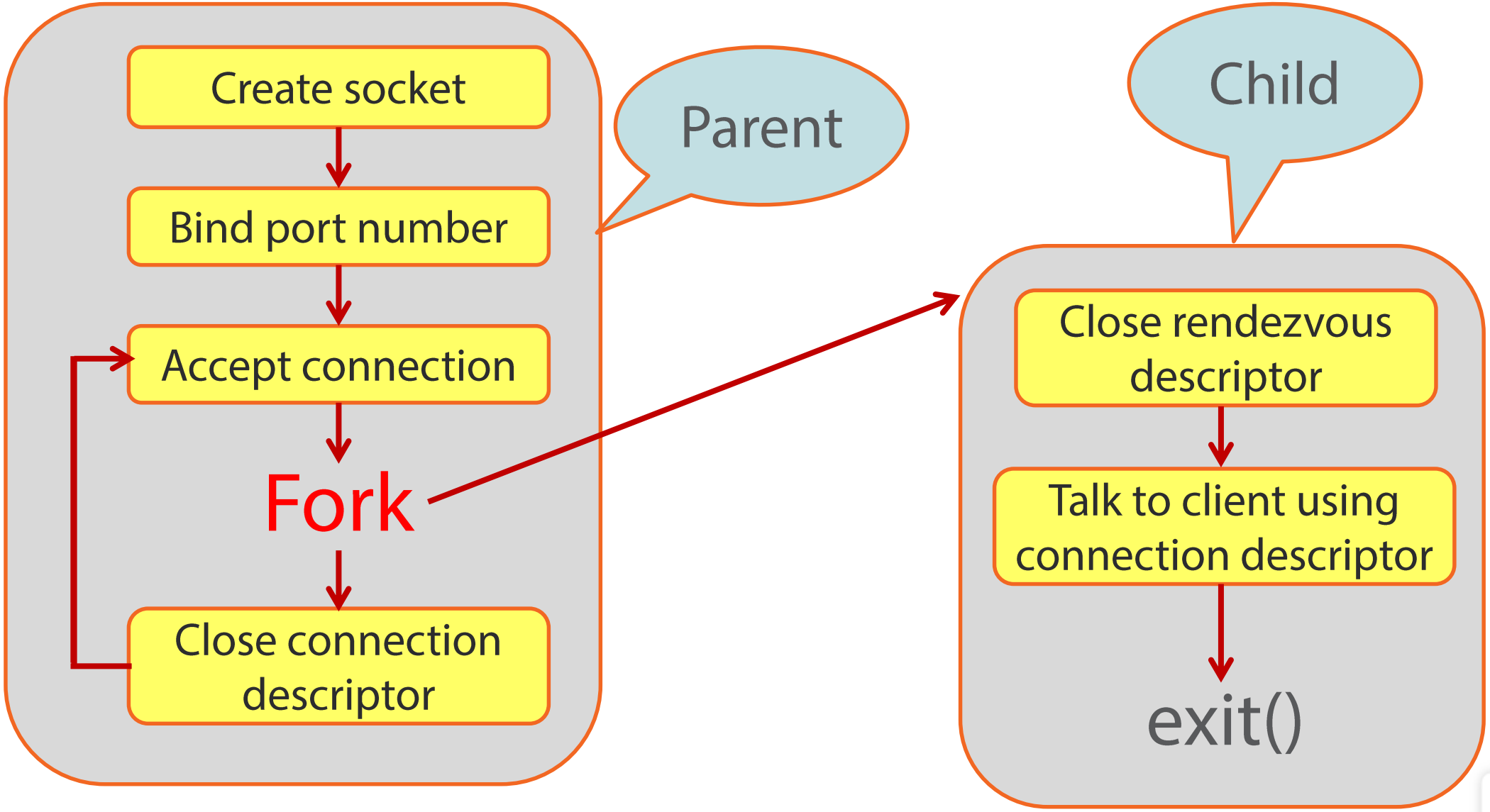


Concurrent Server Schema

```
sock = socket( ... );
bind(sock, ...);
listen(sock, 5);

while(1) {
  fd = accept(sock, ...);
  if (fork() == 0) {
    close(sock);  // Child – process request
    /* Conduct dialogue with client,
       using fd for input and output */
    exit(0);
  } else
      close(fd);  // Parent
}
```

Concurrent Servers Illustrated



Demonstration

Concurrent "hangman" server

The Game of "Hangman"

protection

e?

s?

t?

a?

i?

The Zombie Problem

```
#include <unistd.h>
#include <stdlib.h>

main()
{
  if (fork() == 0) exit(0);
  sleep(1000);
}
```

The Zombie Solution

```
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

main()
{
  signal(SIGCHLD, SIG_IGN);
  if (fork() == 0) exit(0);
  sleep(1000);
}
```

Concurrency Within a Single Process

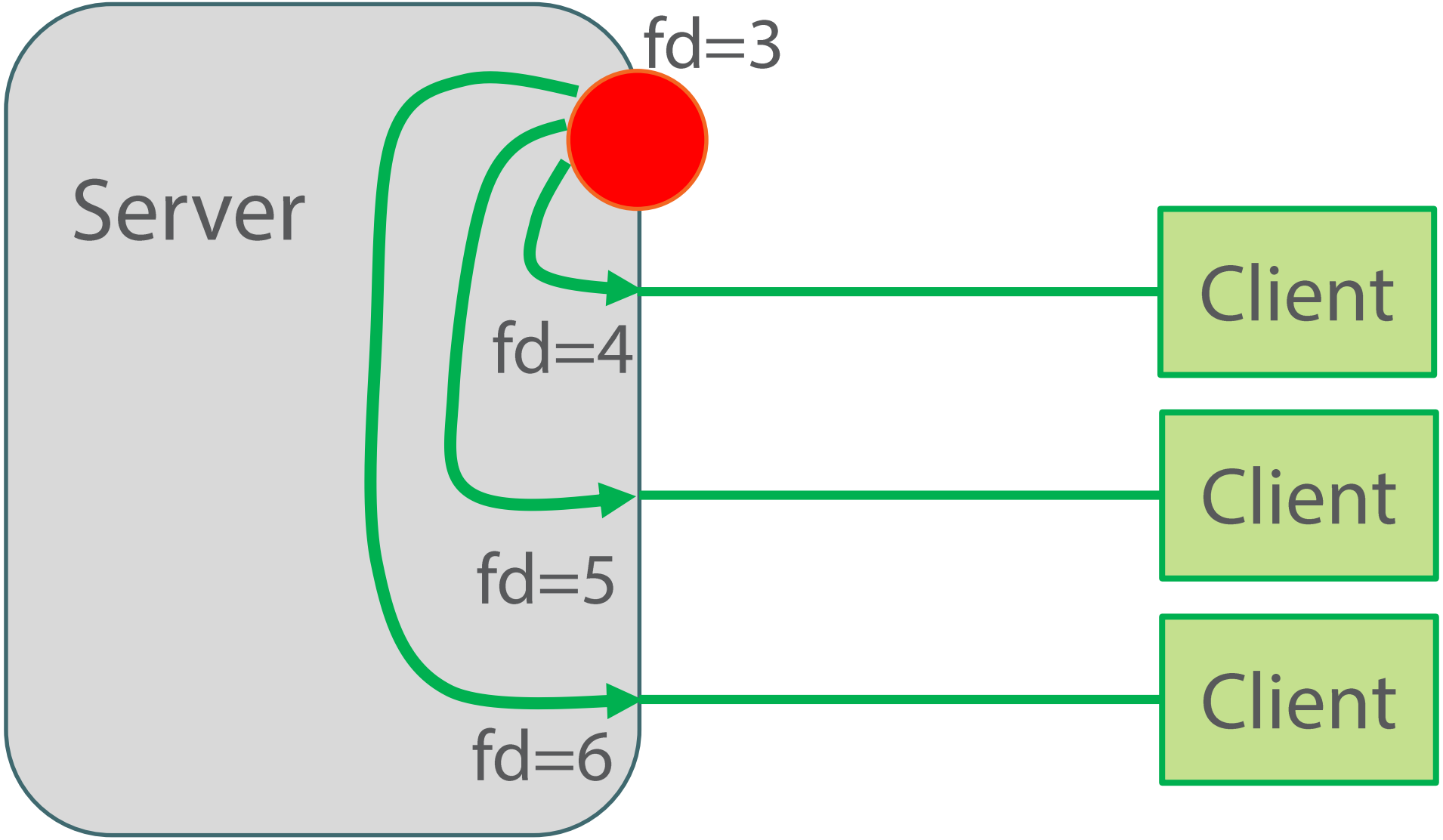
Single-process concurrent servers

The `select()` call

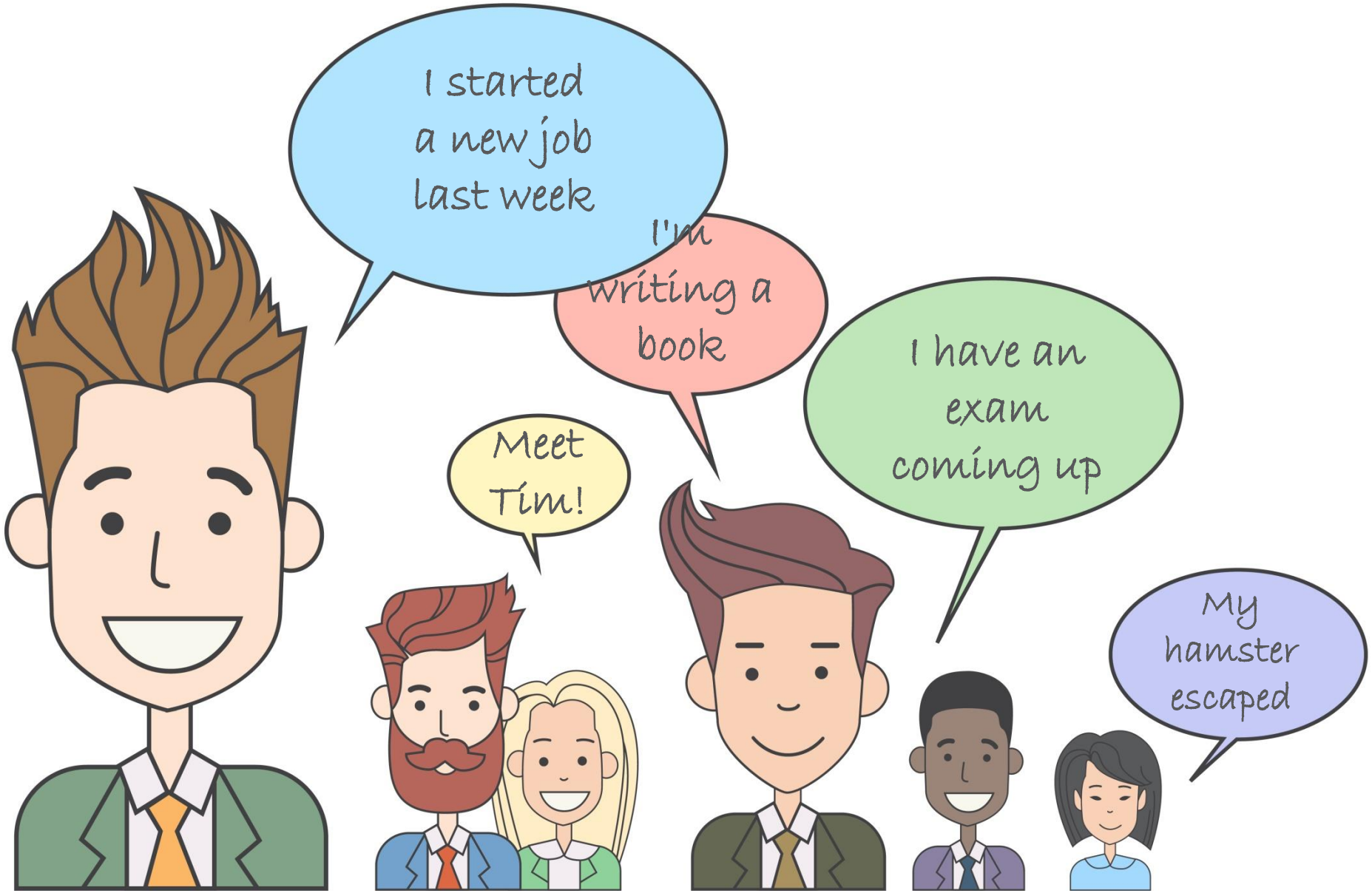
Maintaining State

Demonstration

Supporting Multiple Clients



Maintaining Per-Client State



State in the Concurrent Hangman Server

? What per-client state does the hangman server need to store?

—The "target" word
—The word as guessed so far
— How many lives remain

```
prohibit
-r--ibi-
7
```

? How does it store it?

—Each child process has its own instance of the program's variables

State in the rot13 Server

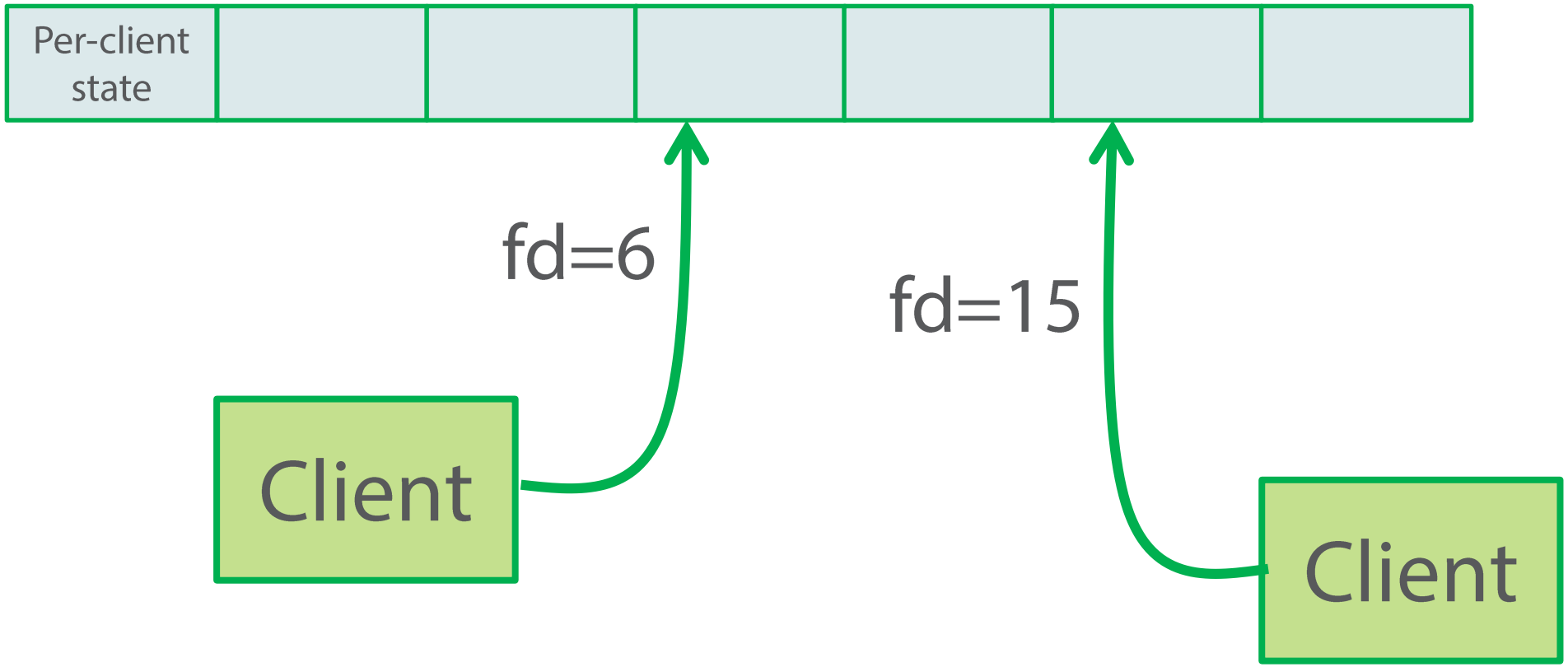
? What per-client state does the *rot13* server need to store?

—None!

—Each request is serviced entirely without reference to earlier requests

—Stateless

Maintaining State in Single-Process Servers



Sharing State

Sharing State

Single-process concurrent server

Easy!

— Just use global variables

— No risk of race conditions

Process-per-client concurrent server

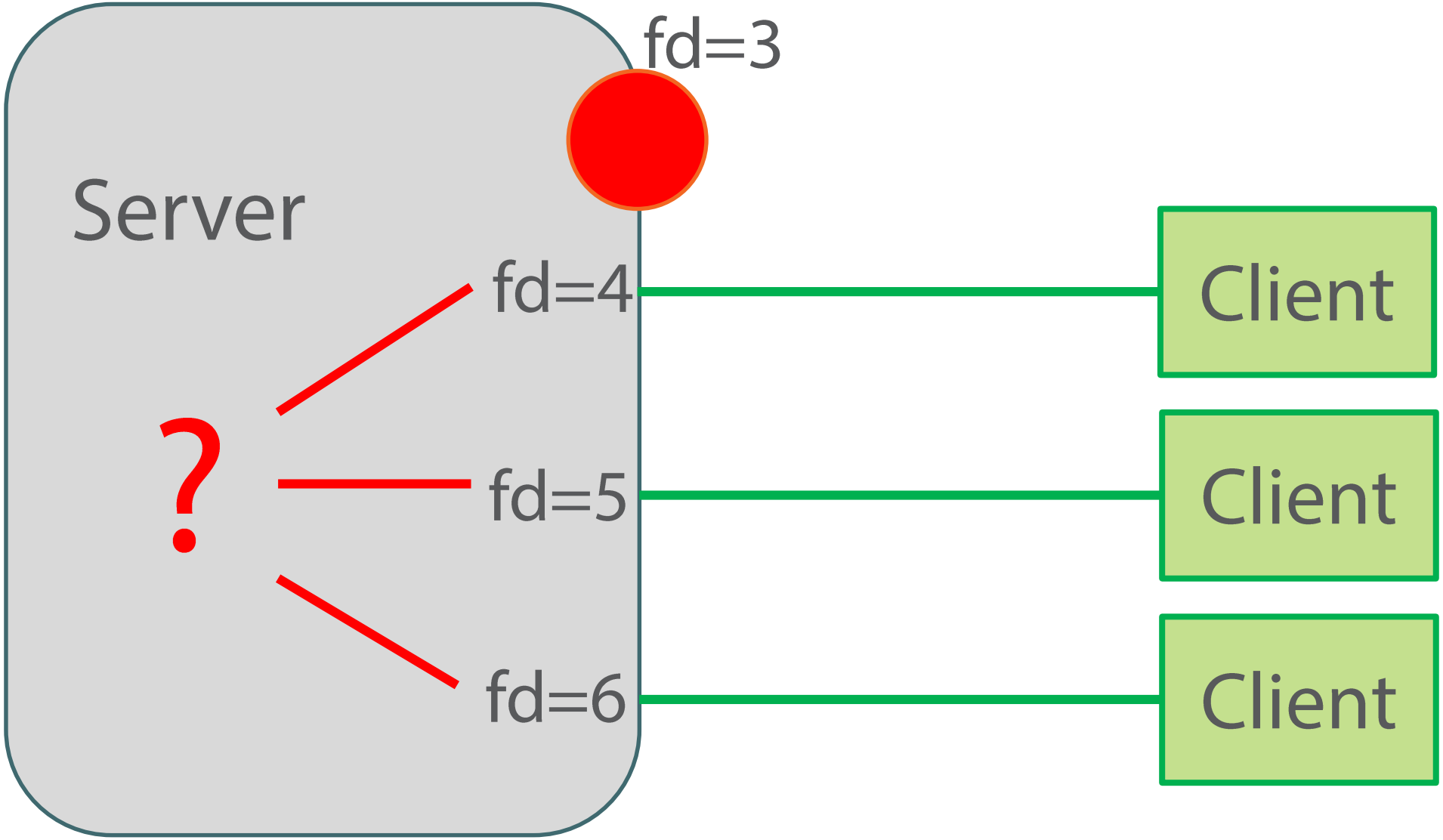
Harder

— Create a shared memory segment

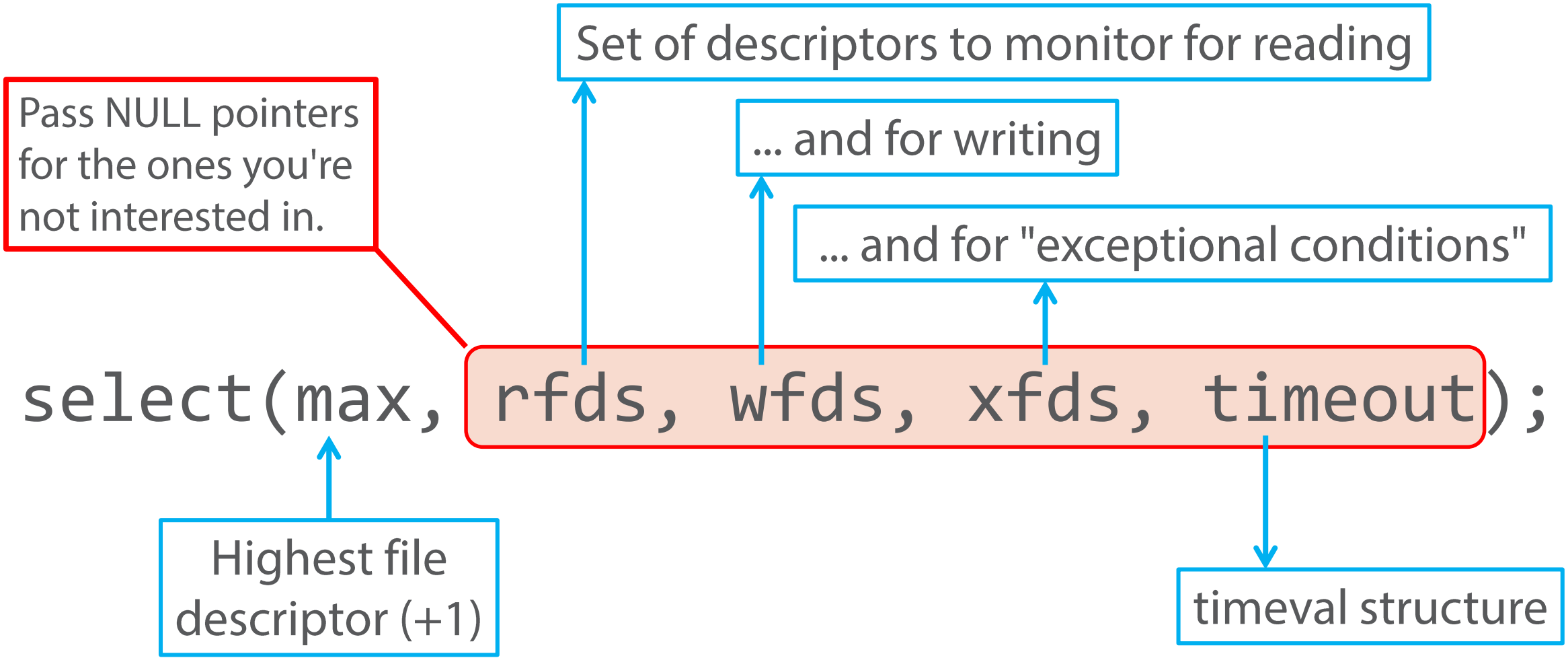
— Or use a file or a database

— Need to avoid race conditions

Managing Multiple File Descriptors

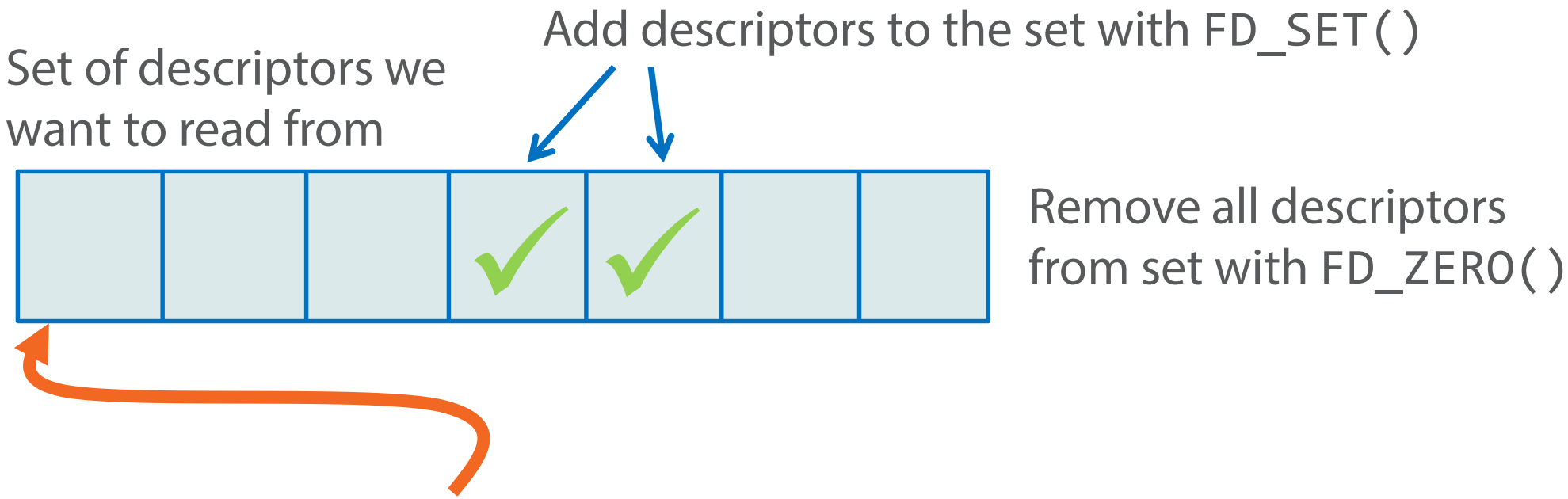


The `select()` System Call



Using Descriptor Sets with `select()`

Before the call ...

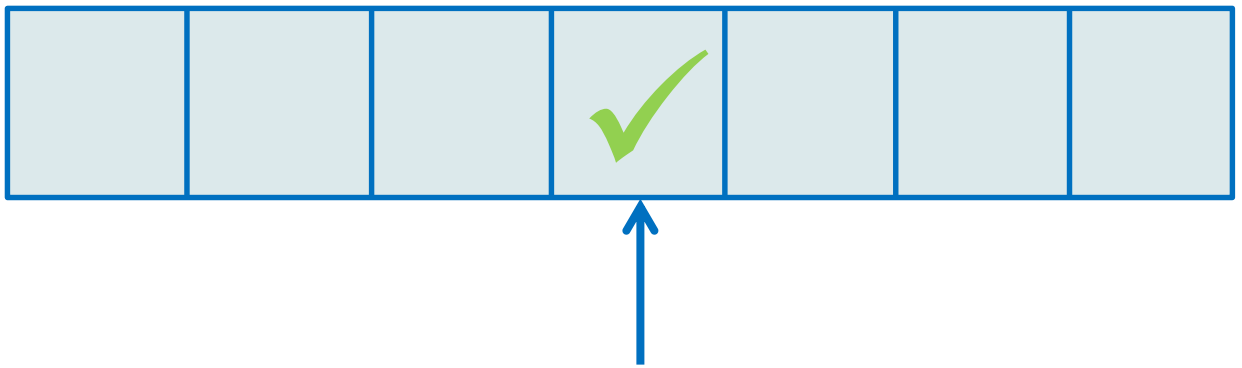



```
select(max, rfds, wfds, xfds, timeout);
```

Using Descriptor Sets with `select()`

After the call ...

Set of descriptors that
are ready for reading



Test if a descriptor is ready with `FD_ISSET()`

Simple `select()` Example

```
fd_set myset;
/* Put descriptors f1 and f2 into myset */
FD_ZERO(&myset);
FD_SET(f1, &myset);
FD_SET(f2; &myset);

/* Wait until f1 or f2 is ready for reading /*
select(16, &myset, NULL, NULL, NULL);
if (FD_ISSET(f1, &myset)) {
    // Read from descriptor f1
}
if (FD_ISSET(f2, &myset)) {
    // Read from descriptor f2
}
```

Demonstration

Concurrent
single-process server

Module Summary

The need for concurrency

Classic process-per-client concurrent servers

State and how to maintain it

Concurrent servers using `select()`

Moving Forward …

Coming up in the next module:

- Threads (compared to processes)
- The "pthreads" API
- Concurrent servers and clients using threads
- How to be "thread-safe"