Allocation of File Descriptors

- For *each process*, the kernel maintains a *file descriptor table*, which is an array of pointers to *"file objects"*
  - a file object represents an *opened* file
  - a *file descriptor* is simply an *index* to this array

- Whenever a process requests a new file descriptor, the *lowest* numbered file descriptor not already associated with an open file is selected; thus

```
#include <fcntl.h>
#include <unistd.h>
...
close(0);
fd = open("file", O_RDONLY);
```

  - the above will always associate "file" with file descriptor 0 (assuming that `open()` succeeds)

- You will need to implement the above rule in the kernel 2 assignment

Running It

```
if (fork() == 0) {
  /* set up file descriptor 1 in the child process */
  close(1);
  if (open("/home/bc/Output", O_WRONLY) == -1) {
    perror("/home/bc/Output");
    exit(1);
  }
  execl("/home/bc/bin/primes", "primes", "300", 0);
  exit(1);
}
/* parent continues here */
while(pid != wait(0)) /* ignore the return code */
  ;
```

- `close(1)` removes file descriptor 1 from *extended address space*
- file descriptors are allocated *lowest first* on `open()`
- *extended address space* survives *execs*
- new code is same as running

```
% primes 300 > /home/bc/Output
```

I/O Redirection

```
% primes 300 > /home/bc/Output
```

- The ">" parameter in a shell command that instructs the command shell to *redirect* the output to the given file
  - If ">" weren't there, the output would go to the display
- Can also redirect input

```
% cat < /home/bc/Output
```

  - when the "`cat`" program reads from file descriptor 0, it would get the data bytes from the file "/home/bc/Output"

File Descriptor Table

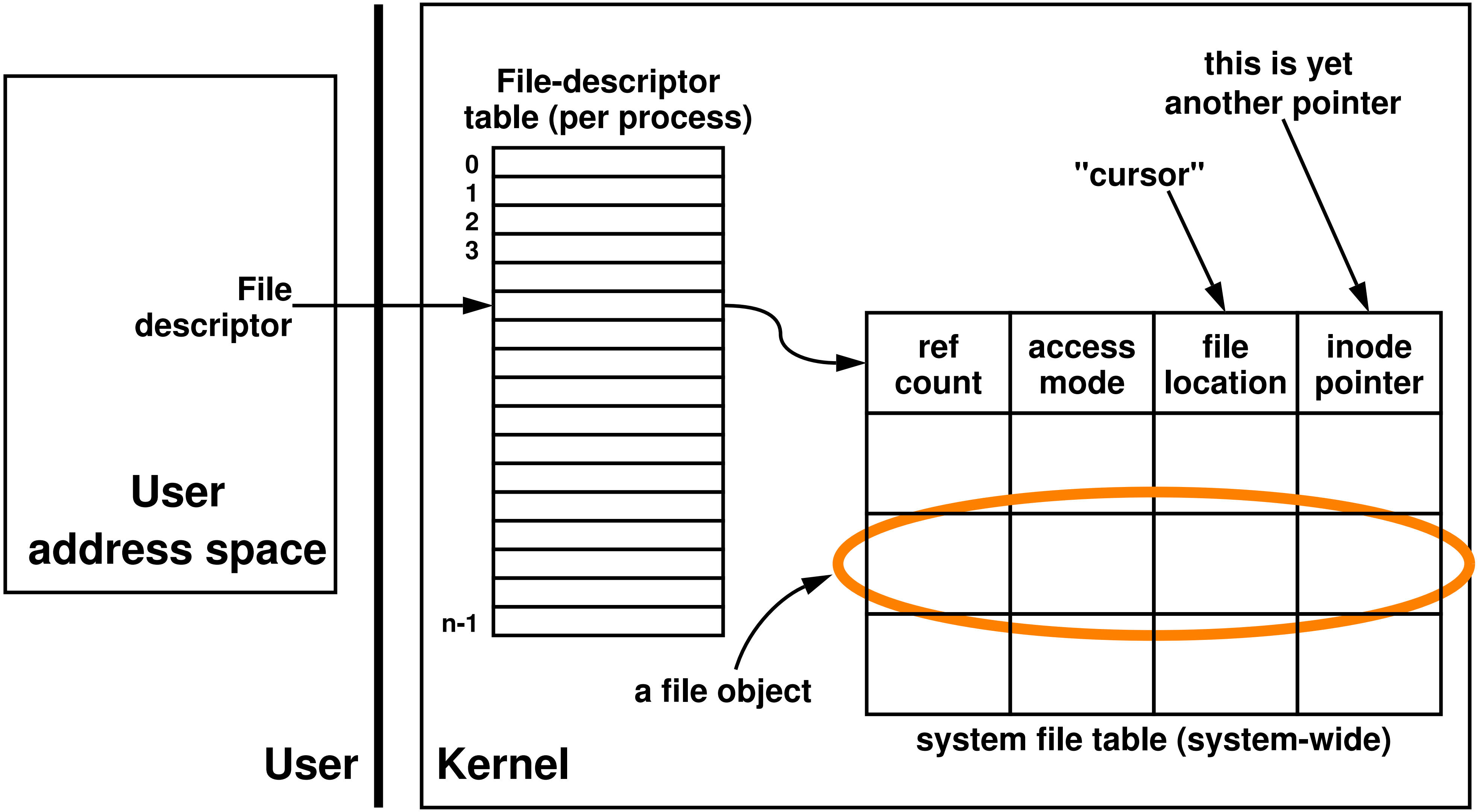
- A file descriptor refers not just to a file
  - it also refers to the *process's* current *context* for that file
    - includes how the file is to be accesses (how `open()` was invoked)
    - *cursor* position / file position
      - next location (zero-based array index) to read/write
      - initialized to 0 when a file is opened

File Object

- *Context* (or "execution context") information must be maintained by the OS and not directly by the user program
  - in this class, we will say that a *file object* is used to maintain the context information about an *opened file*
  - in addition to cursor position, a file object must also remember how a file was opened

- Let's say a user program opened a file with O_RDONLY
  - later on it calls `write()` using the opened file descriptor
  - how does the OS knows that it doesn't have write access?
    - stores O_RDONLY in context
  - if the user program can manipulate the context, it can change O_RDONLY to O_RDWR
  - therefore, user program must not have access to context!
    - all it can see is the handle
    - the file handle is an *index* into an array maintained for the process in kernel's address space

File-Descriptor Table



- *context* is not stored directly into the file-descriptor table
  - one-level of *indirection*

Ch 2: Multithreaded Programming

Bill Cheng

http://merlot.usc.edu/william/usc/

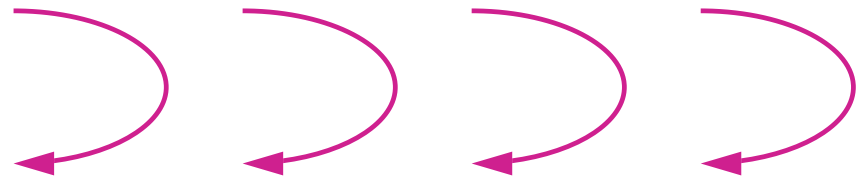
Overview

- Why threads?
- How to program with threads?
  - what is the API?
- Synchronization
  - mutual exclusion
  - semaphores
  - condition variables
- Pitfall of thread programmings

Concurrency

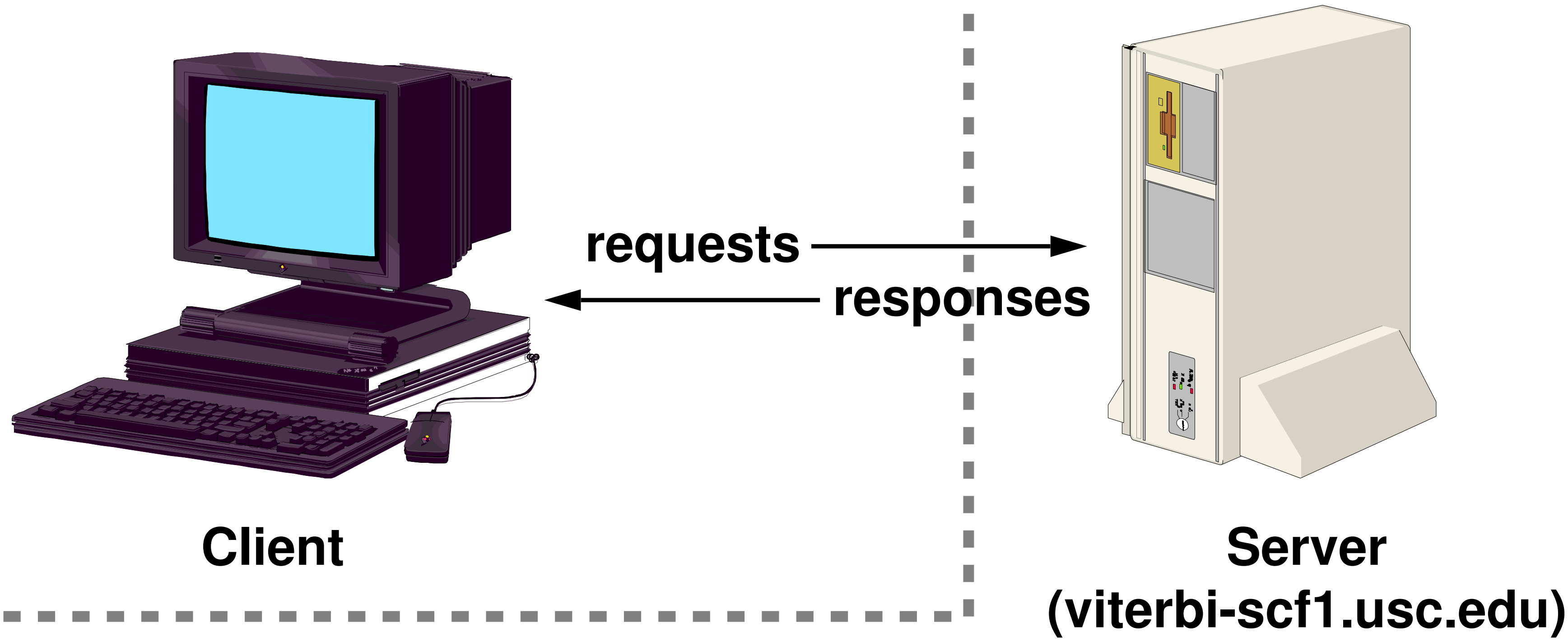
- Many things occur simultaneously in the OS
  - e.g., data coming from a disk, data coming from the network, data coming from the keyboard, mouse got clicked, jobs need to get executed
- If you have multiple processors, you may be able to handle things in parallel
  - that's real concurrency/parallelism
- If you only have one processor, you may want to make it look like things are running in parallel
  - do multiplexing to create the illusion
  - as it turns out, it's a *good idea* to do this even if you have only have one processor
- The down side is that if you want concurrency, you have to have *concurrency control* or bad things can happen

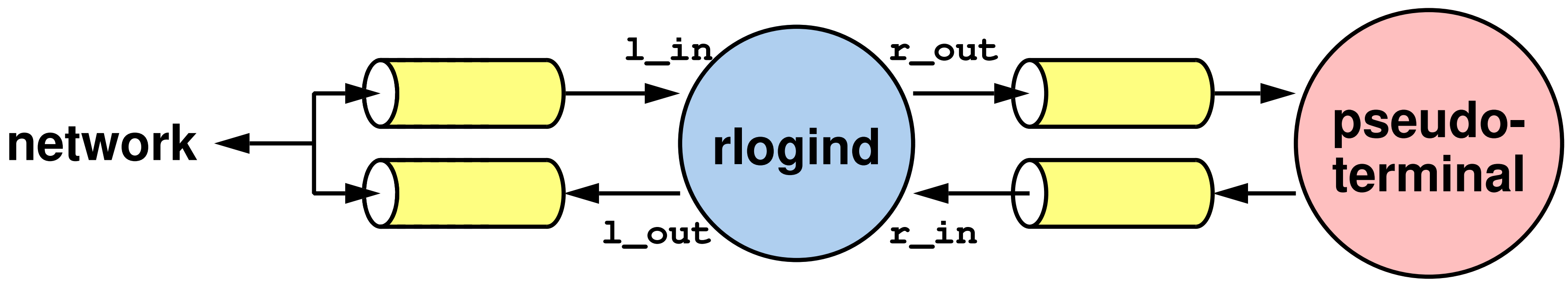
Why Threads?



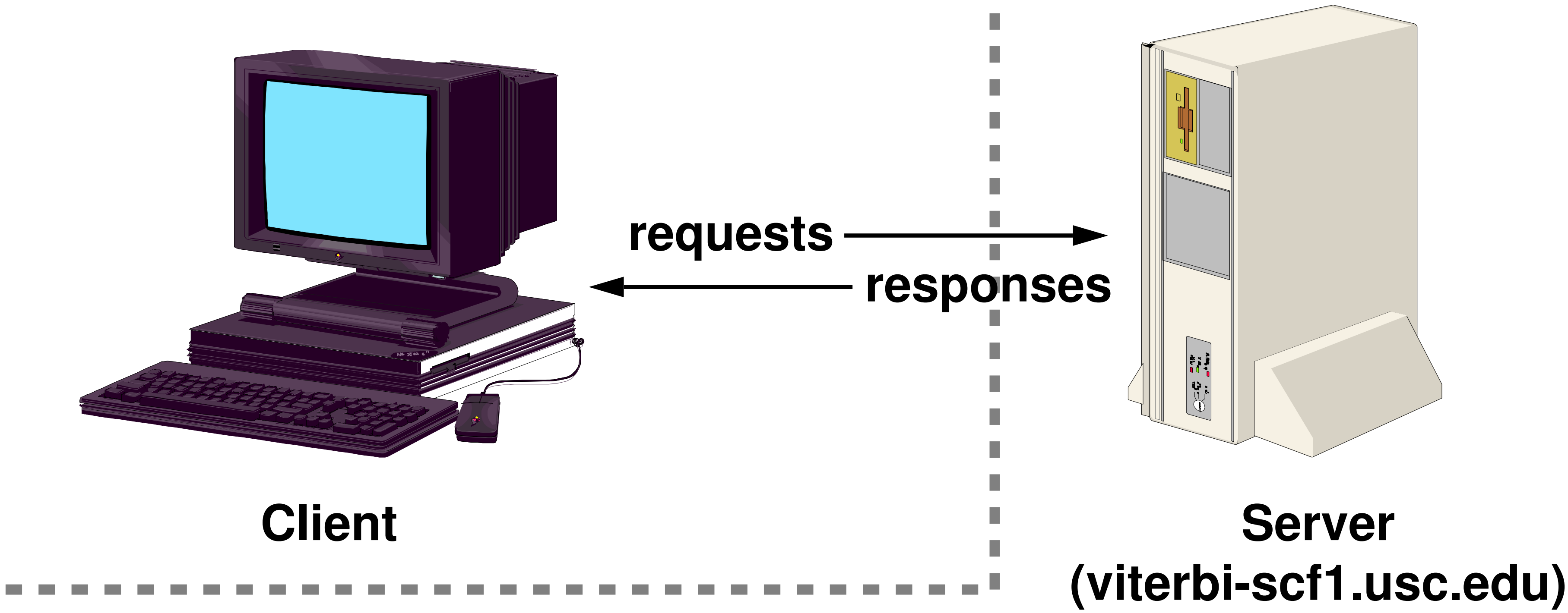
- Many things are easier to do with threads
  - *multithreading* is a *powerful paradigm*
  - makes your design *cleaner*, and therefore, less buggy
- Many things run faster with threads
  - if you are just waiting, don't waste CPU cycles, give the CPU to someone else, *without explicitly* giving up the CPU
- *Kernel threads* vs. *user threads*
  - basic concepts are the same
  - can easily do programming assignments for user-level threads
    - that's why we start here (to get your warmed up)!
    - for kernel programming assignments, you need to fill out missing parts of various kernel threads

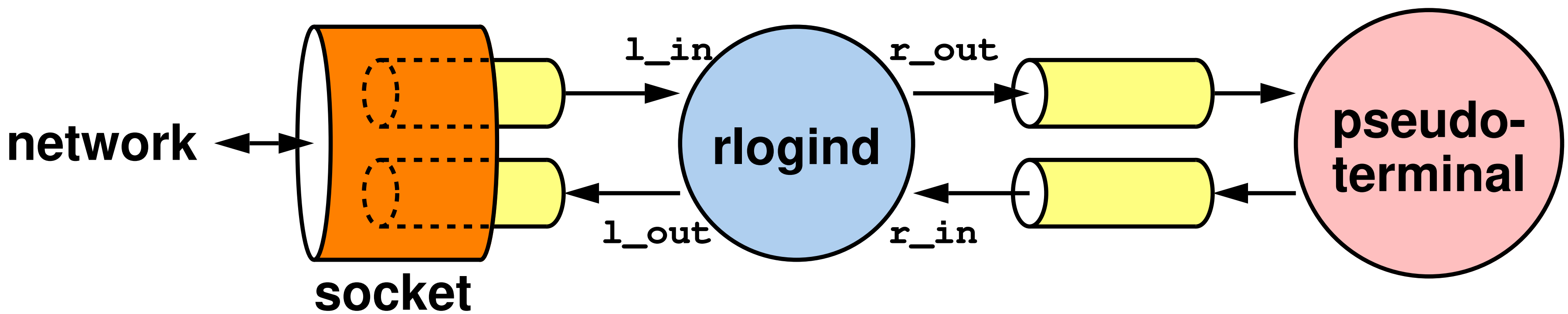
A Simple Example: `rlogind`





A Simple Example: `rlogind`

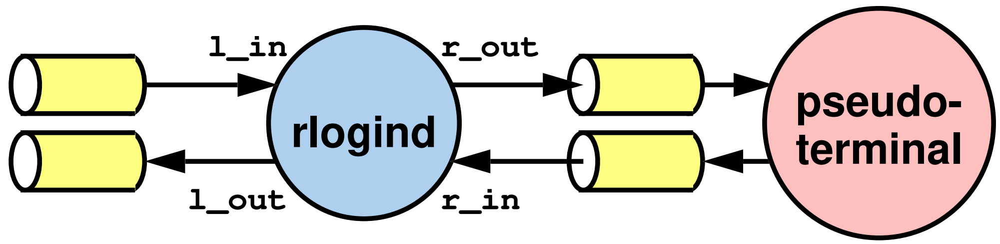




- for a socket, `l_in = l_out`, i.e., you read and write using the same file descriptor

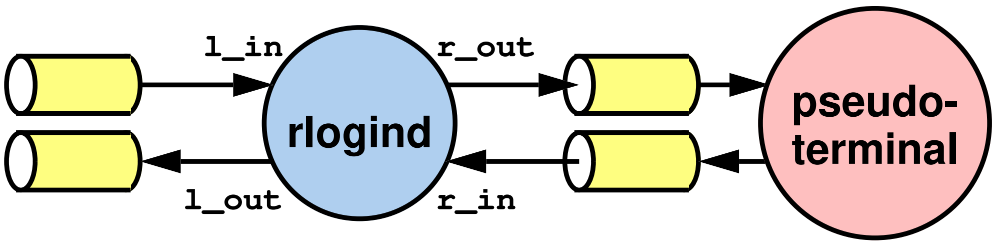
Life Without Threads

```
logind(int r_in, int r_out, int l_in, int l_out) {
  fd_set in = 0, out;
  int  want_l_write = 0, want_r_write = 0;
  int want_l_read = 1, want_r_read = 1;
  int eof = 0, tsize, fsize, wret;
  char fbuf[BSIZE], tbuf[BSIZE];

  fcntl(r_in, F_SETFL, O_NONBLOCK);
  fcntl(r_out, F_SETFL, O_NONBLOCK);
  fcntl(l_in, F_SETFL, O_NONBLOCK);
  fcntl(l_out, F_SETFL, O_NONBLOCK);

  while(!eof) {
    FD_ZERO(&in);
    FD_ZERO(&out);
    if (want_l_read) FD_SET(l_in, &in);
    if (want_r_read) FD_SET(r_in, &in);
    if (want_l_write) FD_SET(l_out, &out);
    if (want_r_write) FD_SET(r_out, &out);
    select(MAXFD, &in, &out, 0, 0);
    if (FD_ISSET(l_in, &in)) {
      if ((tsize = read(l_in, tbuf, BSIZE)) > 0) {
        want_l_read = 0;
        want_r_write = 1;
      } else { eof = 1; }
    }
```

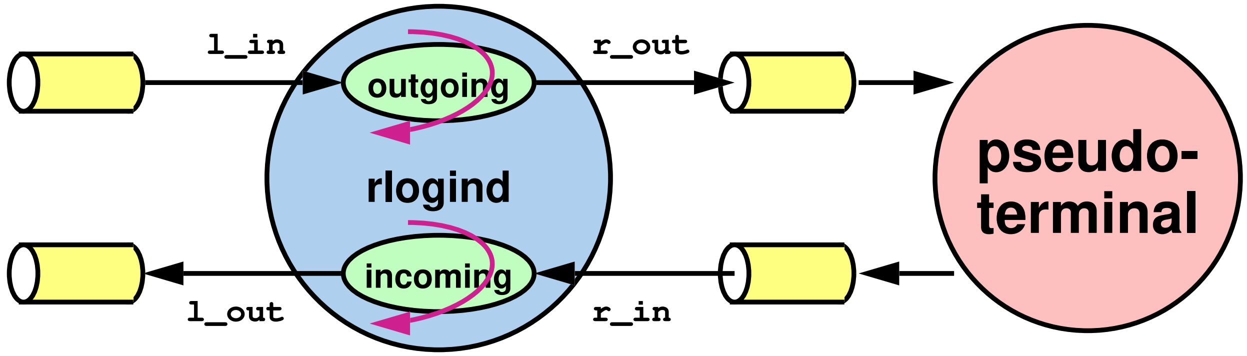


Life Without Threads

```
      if (FD_ISSET(r_in, &in)) {
        if ((fsize = read(r_in, fbuf, BSIZE)) > 0) {
          want_r_read = 0;
          want_l_write = 1;
        } else { eof = 1; }
      }
      if (FD_ISSET(l_out, &out)) {
        if ((wret = write(l_out, fbuf, fsize)) == fsize) {
          want_r_read = 1;
          want_l_write = 0;
        } else if (wret >= 0) {
          tsize -= wret;
        } else { eof = 1; }
      }
      if (FD_ISSET(r_out, &out)) {
        if ((wret = write(r_out, tbuf, tsize)) == tsize) {
          want_l_read = 1;
          want_r_write = 0;
        } else if (wret >= 0) {
          tsize -= wret;
        } else { eof = 1; }
      }
    }
  }
```



Life With Threads

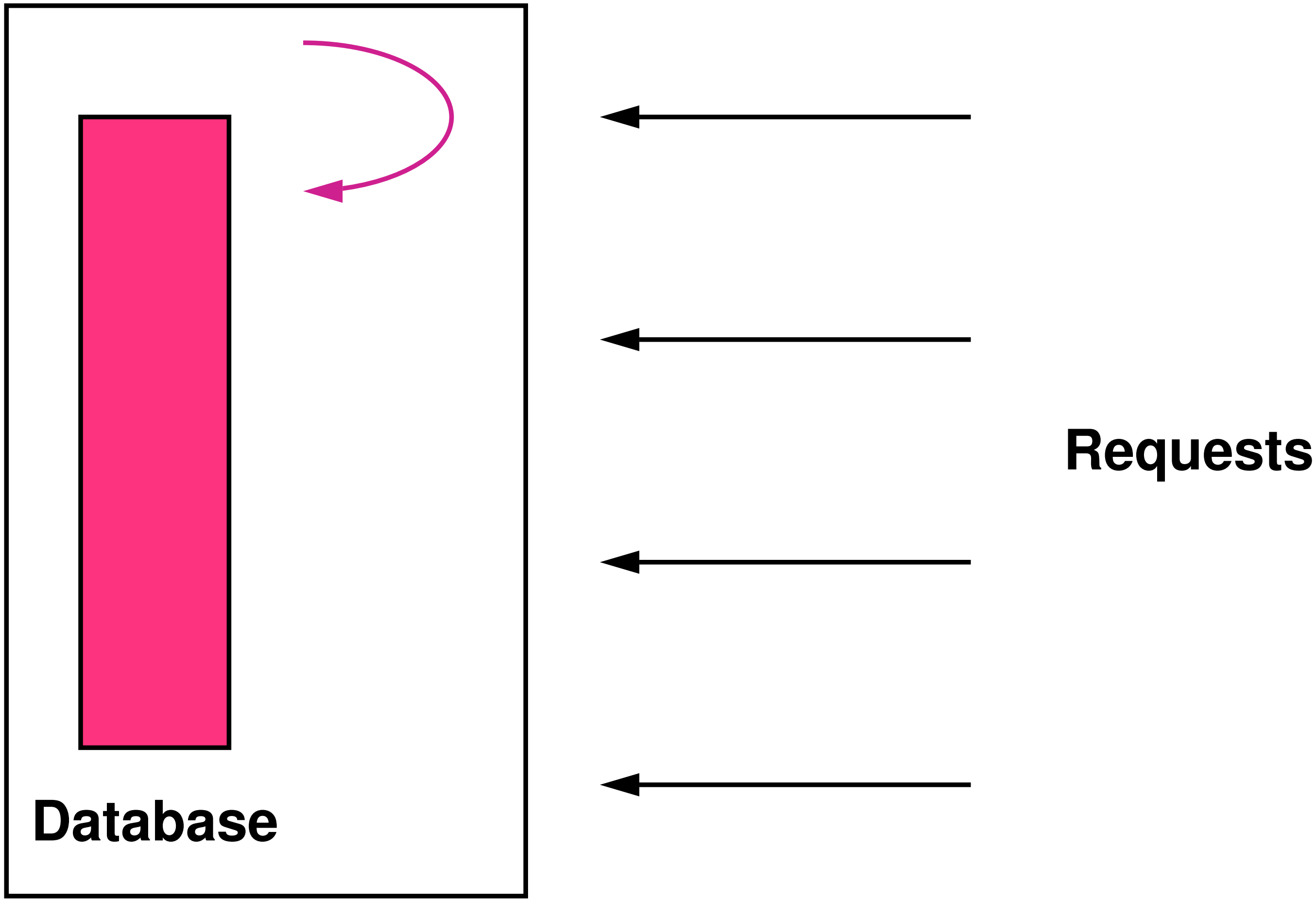



```
incoming(int r_in, int l_out) {
  int eof = 0;
  char buf[BSIZE];
  int size;

  while (!eof) {
    size = read(r_in, buf, BSIZE);
    if (size <= 0)
      eof = 1;
    if (write(l_out, buf, size) <= 0)
      eof = 1;
  }
}
```

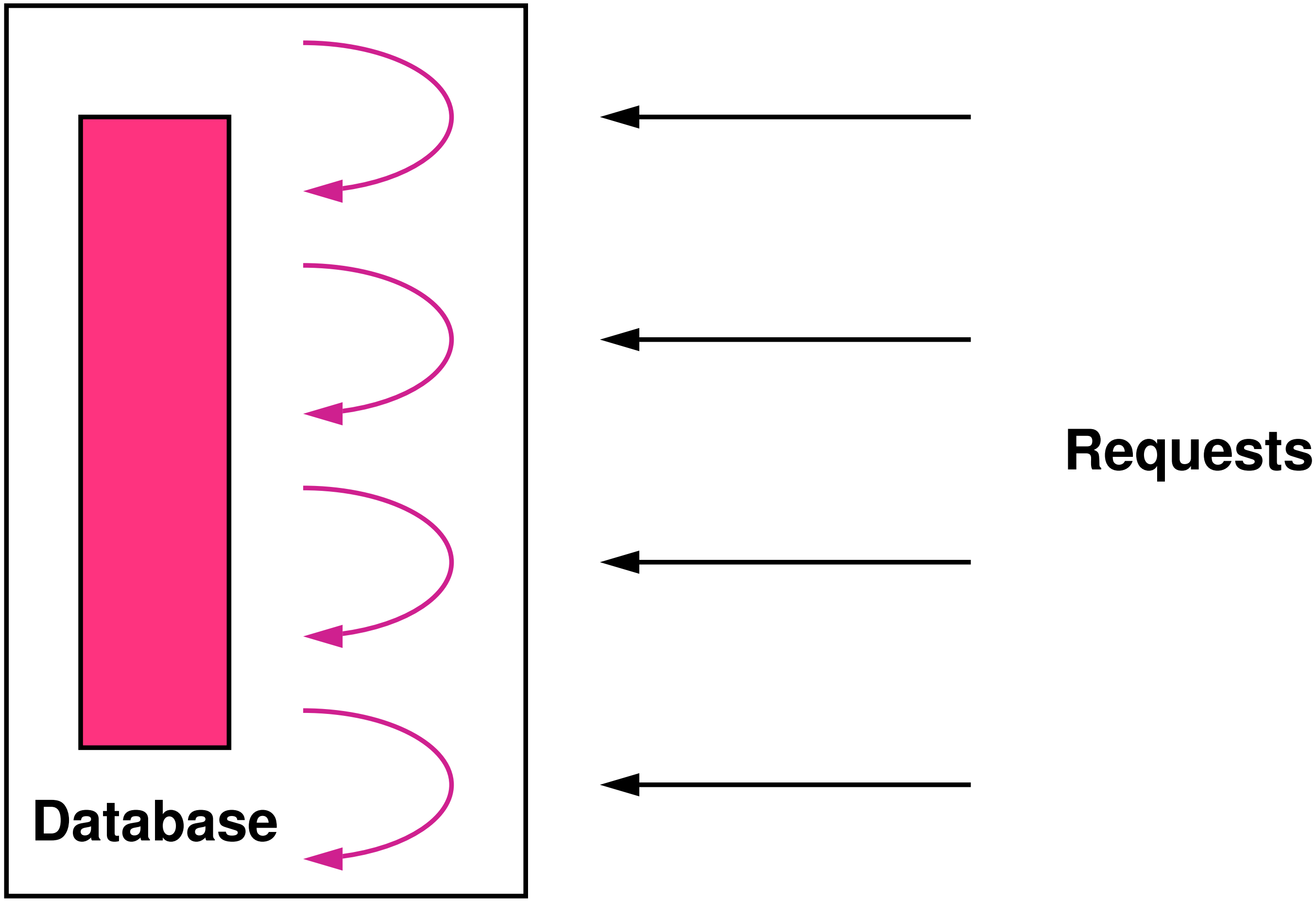
```
outgoing(int l_in, int r_out) {
  int eof = 0;
  char buf[BSIZE];
  int size;

  while (!eof) {
    size = read(l_in, buf, BSIZE);
    if (size <= 0)
      eof = 1;
    if (write(r_out, buf, size) <= 0)
      eof = 1;
  }
}
```

- don’t have to call `select()`

Single-Threaded Database Server



Multithreaded Database Server



- will be very difficult to implement this without using threads if you want to handle a large number of requests simultaneously

2.2 Programming With Threads

- *Threads Creation & Termination*
- Threads & C++
- Synchronization
- Thread Safety
- Deviations

Creating a POSIX Thread

- `man pthread_create`

```
SYNOPSIS
    #include <pthread.h>

    int pthread_create(
          pthread_t *thread,
          const pthread_attr_t *attr,
          void *(*start_routine)(void *),
          void *arg);

    Compile and link with -pthread.
```

- the `start_routine` is also known as the "*first procedure*" or "*thread function*" of the child thread
  - it's like `main()` for the child thread
- the "thread ID" of the newly created thread will be *returned* in the first argument of `pthread_create()`
  - may *not* be a *Thread Control Block*

Creating a POSIX Thread

```
start_servers( ) {
  pthread_t thread;
  int i;
  for (i = 0; i < 100; i++)
    pthread_create(&thread,     // thread ID
                   0,           // default attributes
                   server,      // first procedure
                   argument);   // argument of first
}                               //     procedure

void *server(void *arg) {
  // perform service
  return(0);
}
```

child thread *starts* executing here

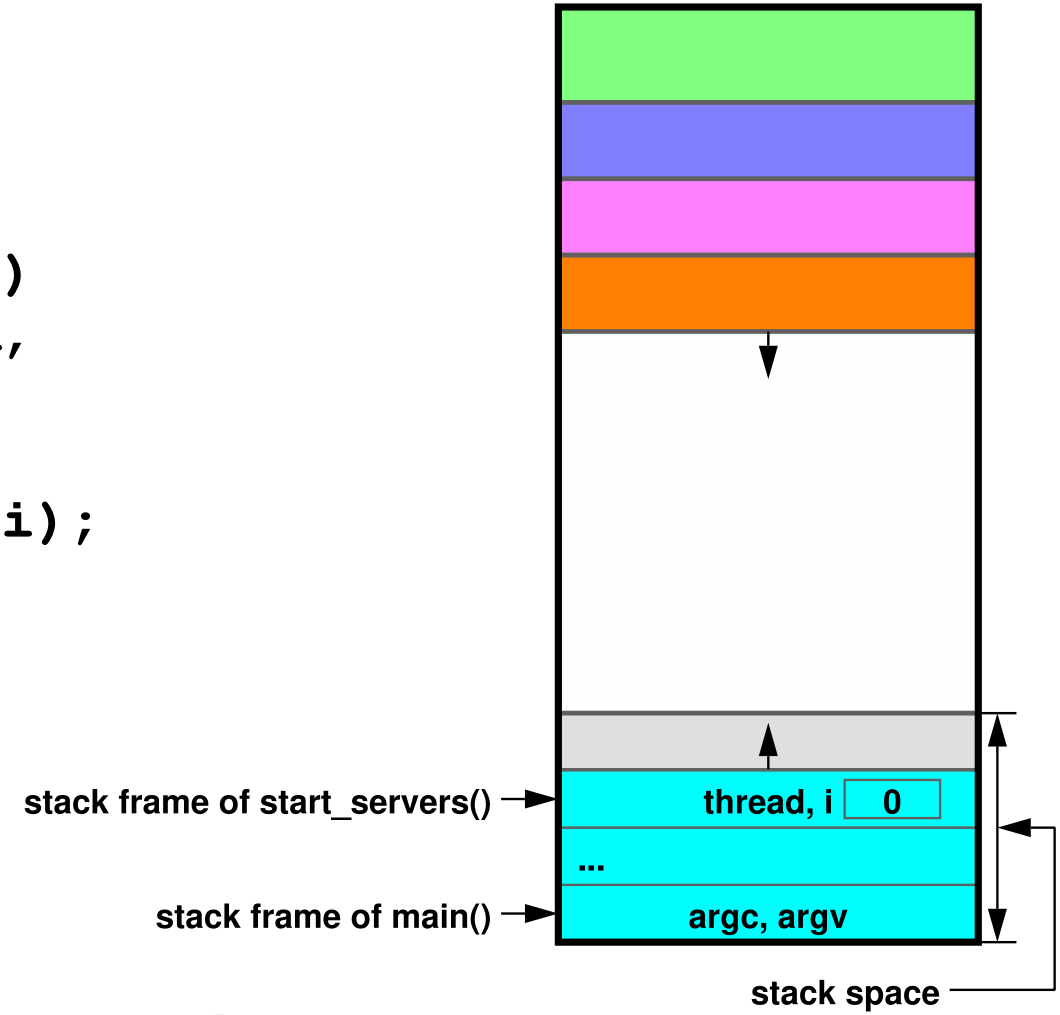
arg = argument (from caller)

child thread *ends* when *return* from its *start routine / first procedure*

- `pthread_create()` returns 0 if successful
- POSIX 1003.1c standard
  - *pthread* is a *user-space* library package
- *threads* in a process *shares the address space*

Creating a POSIX Thread

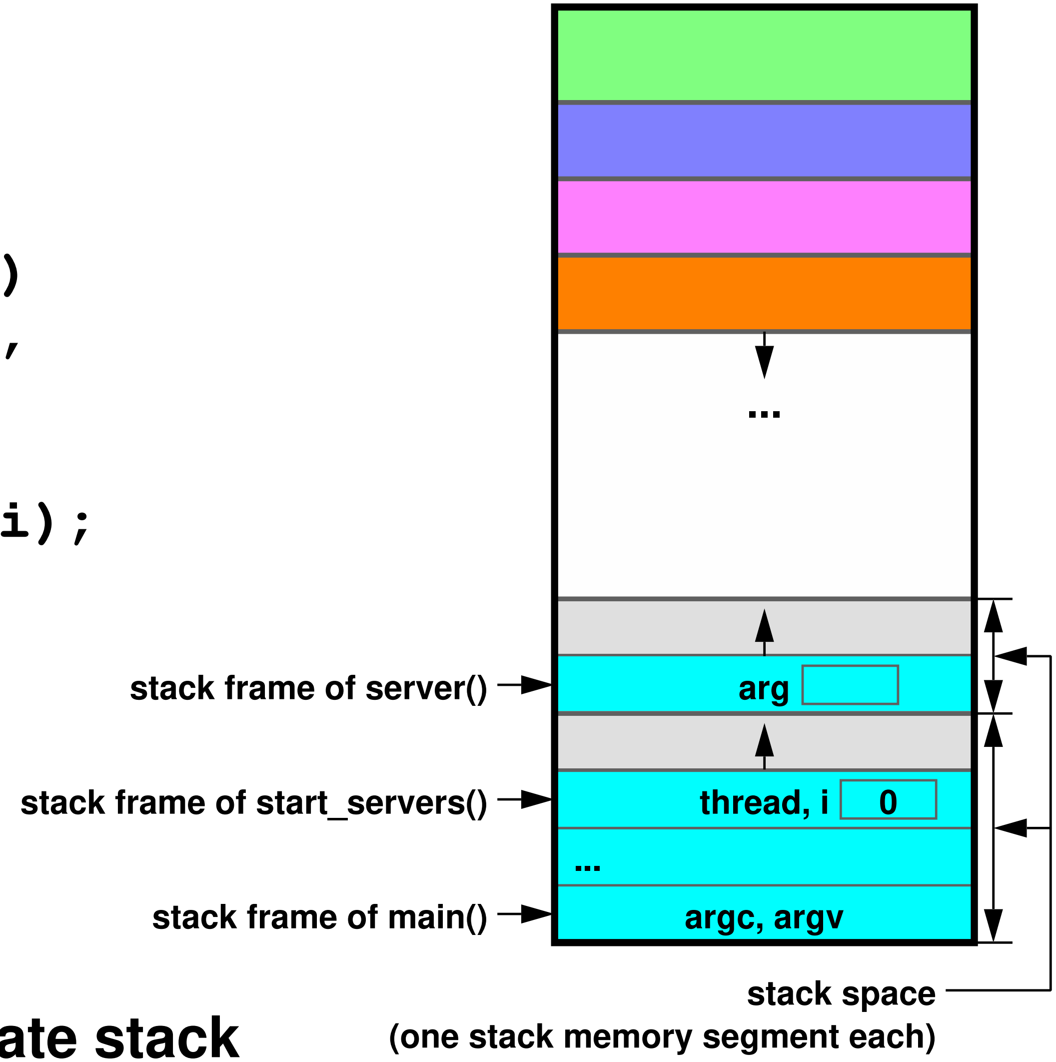
```
start_servers( ) {
  pthread_t thread;
  int i;
  for (i = 0; i < 100; i++)
    pthread_create(&thread,
                   0,
                   server,
                   (void*)i);
}

void *server(void *arg) {
  int k=(int)arg;
  // perform service
  return(0);
}
```



- every thread needs a separate stack
  - first stack frame in every child thread corresponds to `server()`
    - one arg in each of these stack frames

Creating a POSIX Thread

```
start_servers( ) {
  pthread_t thread;
  int i;
  for (i = 0; i < 100; i++)
    pthread_create(&thread,
                   0,
                   server,
                   (void*)i);
}

void *server(void *arg) {
  int k=(int)arg;
  // perform service
  return(0);
}
```



- every thread needs a separate stack
  - first stack frame in every child thread corresponds to `server()`
    - one arg in each of these stack frames
    - a stack space is in its own stack memory segment