# Housekeeping (Lecture 3 - 9/4/2013)

- Warmup #1 due at 11:45pm on Friday, 9/13/2013
  - if you have code from a previous semester, be very careful and *not copy any code from it*
    - it's best if you just get rid of it
  - get started soon
    - if you are stuck, make sure you come to see me during office hour next Monday

- *Grading guidelines* is the *ONLY* way we will grade and we will grade on `nunki.usc.edu` in our grading account (which you don't have access to)
  - due to our *fairness* policy
  - it's a good idea to run your code against the grading guidelines

- You need to keep up with the lecture materials
  - anything you don't understand fully, come to see me soon
    - or post a message to the class Google Group

# Housekeeping (Lecture 3 - 9/4/2013)

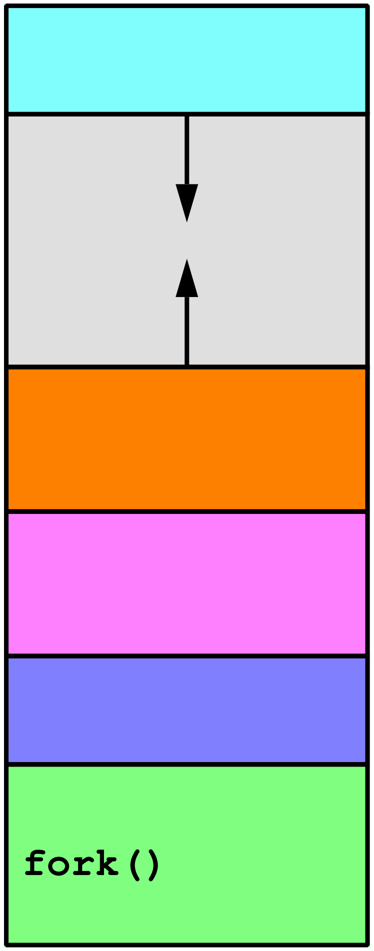
- **Please do not set your class Google Group e-mail delivery preference to "No email"**
  - **because all important announcements will be sent to the class Google Group**
  - **if you do that, I will change it to "All email"**
  - **if you just want to file them away, setup a filter to automatically put our Google Group messages into a folder**

- **New lectures schedule, starting next week**
  - ***(AM section)*** **MW 10:00am - 11:25am in WPH B27**
    - **my plan is to have the class ends at 11:20am**
  - ***(PM section)*** **MW 12:25pm - 1:50pm in SLH 102**
    - **my plan is to keep the two sections synchronized**
  - **I will change the class web page tonight**

# Creating a Process

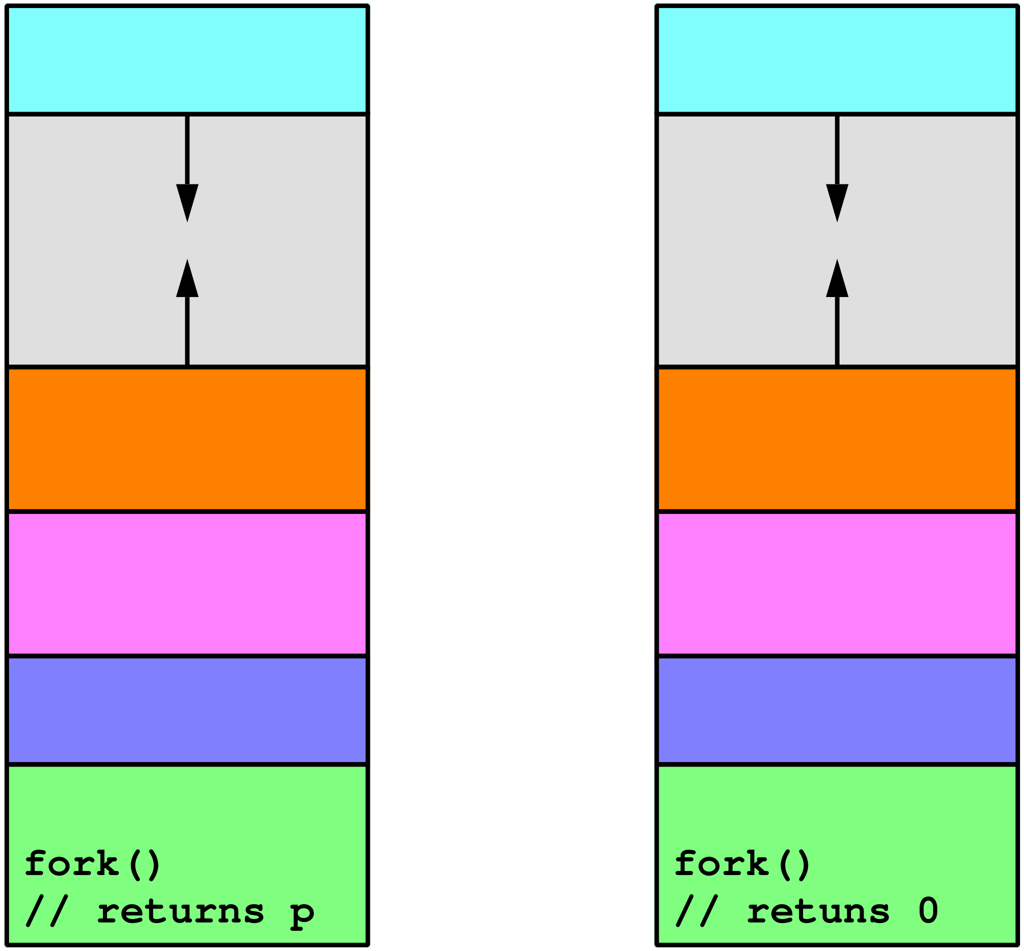
- Creating a process is deceptively simple
  - make a copy of a process (the parent process)
    - call `fork()`
    - the process where `fork()` is called is the *parent* process
    - the copy is the *child* process
    - in a way, `fork()` returns twice
      - once in the parent, the returned value is the *process ID (PID)* of the child process
      - once in the child, the returned value is 0
      - a PID is 16-bit long
  - this is the *only* way to create a process
- Making a copy of the entire address space can be expensive
  - Ch 7 shows speed up tricks
  - e.g., text segment is read-only so parent and child can share it
- Example: relationship between a shell (i.e., a command interpreter, such as `/bin/tcsh`) and `/bin/ls`

# Creating a Process: Before




parent proces

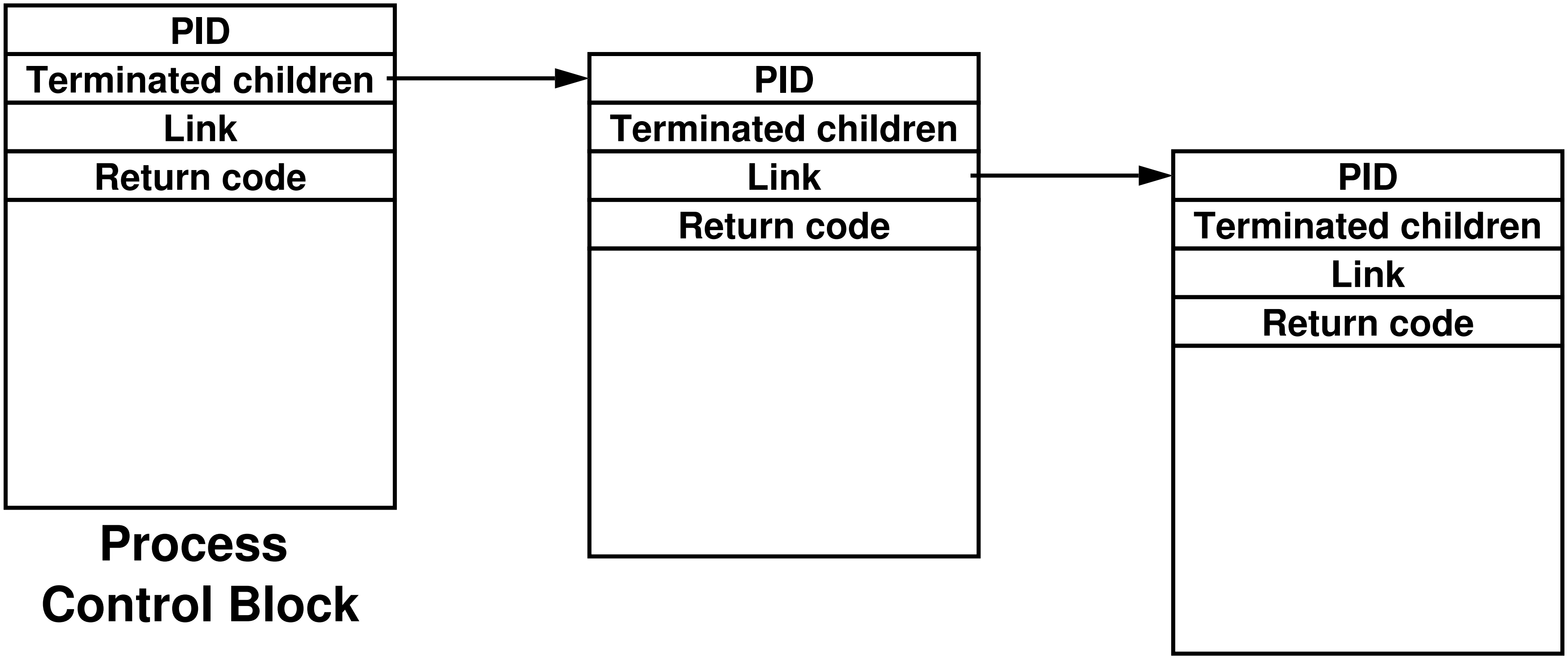
# Creating a Process: After




parent proces

child proces
(pid = p)

# Process Control Blocks




- PCB is a kernel data structure
  - pretty much every field is unsigned
  - return code (when a process dies) is 8-bit long
    - so that the parent process can know what happened to child
  - the "Link" field points to the next PCB
    - but, the next PCB in what list?

# Fork and Wait

```
short pid;
if ((pid = fork()) == 0) {
  /* some code is here for the child to execute */
  exit(n);
} else {
  int ReturnCode;
  while(pid != wait(&ReturnCode))
     ;
  /* the child has terminated with ReturnCode as
     its return code */
}
```

- e.g., `/bin/tcsh` forks `/bin/ls`
- what does `exit(n)` do other than copying `n` into PCB?
  - least significant 8-bits of `n`
- what happens when `main()` calls `return(n)`?
  - eventually, `exit(n)` will be invoked
- `wait()` is a blocking call
  - it reaps dead child processes one at a time

# Process Termination Issues

- PID is only 16-bits long
  - OS must not reuse PID too quickly or there may be ambiguity
- When `exit()` is called, the OS must not free up PCB too quickly
  - parent needs to get the return code
  - it's okay to free up everything else (such as address space)
- Solutions for both is for the terminated child process to go into a *zombie* state
  - only after `wait()` returned with the child's PID and the PID be reused and the PCB be freed up
  - but what if the parent calls `exit()` while the child is in the zombie state?
    - process 1 (the process with PID=1) inherits all the zombie children of this parent process
    - process 1 keeps calling `wait()` to reap the zombies

# 1.3 A Simple OS

- OS Structure
- Processes, Address Spaces, & Threads
- Managing Processes
- *Loading Program Into Processes*
- Files

# Loading Programs Into Processes

- How do you run a program?
  - make a copy of a process
    - any process
  - replace the child process with a new one
    - wipe out the child process
    - using a family of system calls known as *exec*
  - kind of a waste to make a copy in the first place
    - but it's the only way
    - also, the OS does not know if the reason the parent process calls `fork()` is to run a new program or not

# Exec

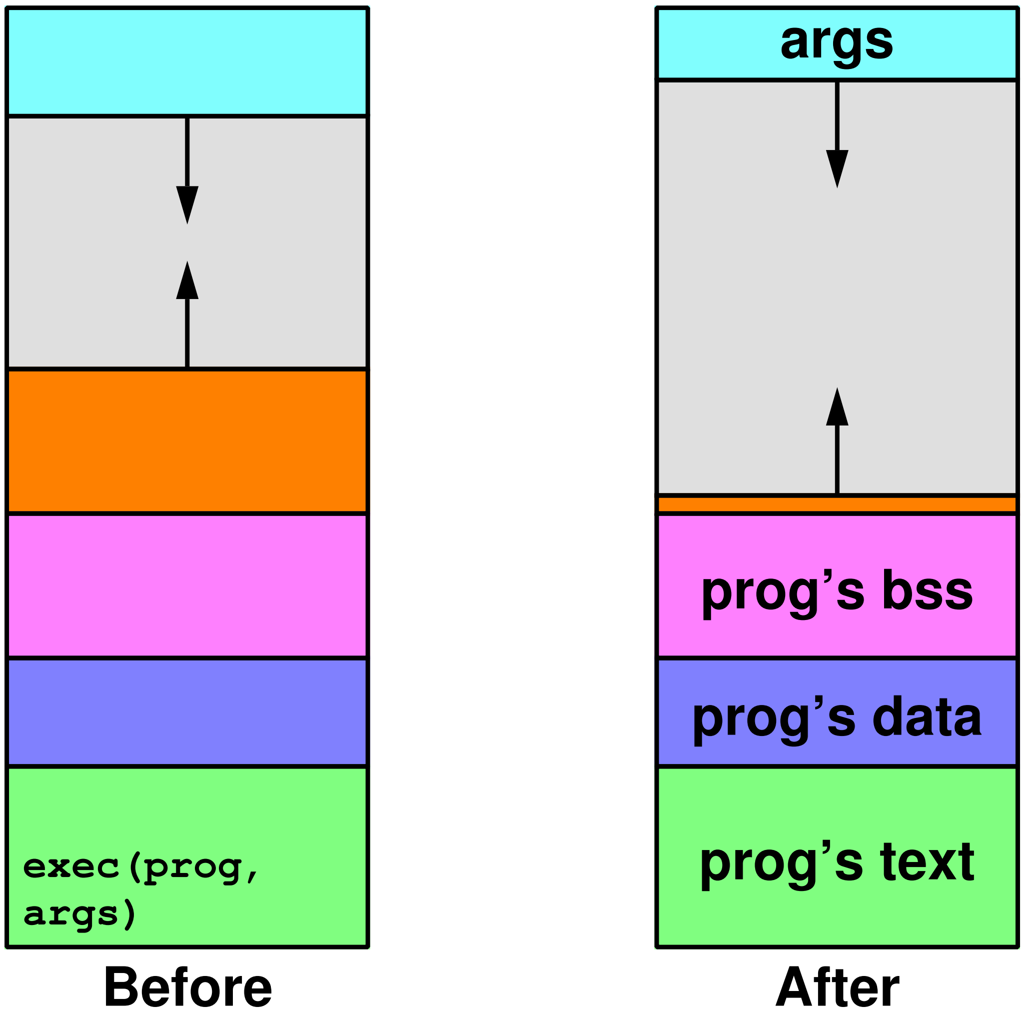
```
int pid;
if ((pid = fork()) == 0) {
  /* we'll soon discuss what might take place before
     exec is called */
  execl("/home/bc/bin/primes", "primes", "300", 0);
  exit(1);
}
/* parent continues here */
while(pid != wait(0)) /* ignore the return code */
  ;
```

- what does execl() do?
  - "man execl" says:

    ```
    int execl(const char *path,
              const char *arg, ...);
    ```

  - isn't "primes" in the 2nd argument kind of redundent?
  - what's up with "..."?
    - this is called *"varargs"*

# Loading a New Image



# Exec

```
% primes 300
```

- **Your login shell forks off a child process, load the primes program on top of it, wait for the child to terminate**
  - **the same code as before**
  - `exit(1)` **would get called if somehow** `execl()` **returned**
    - **if** `execl()` **is successful, it cannot return since the code is gone (i.e., the code segment has been replaced by the code segment of "primes")**

# Put It All Together

**Parent (shell)**

```
fork()
```

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

**Applications**

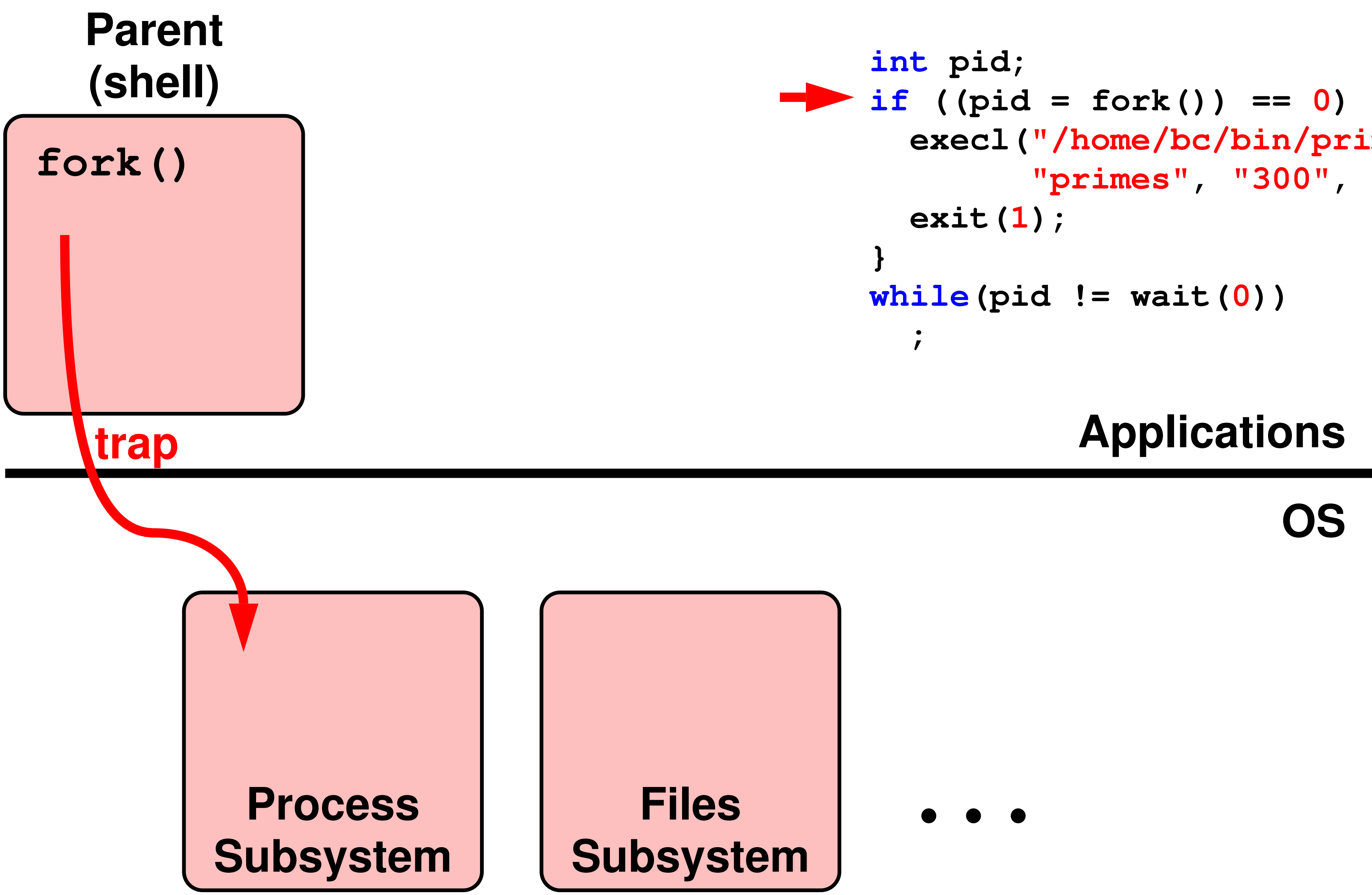
**OS**

**Process Subsystem**

**Files Subsystem**

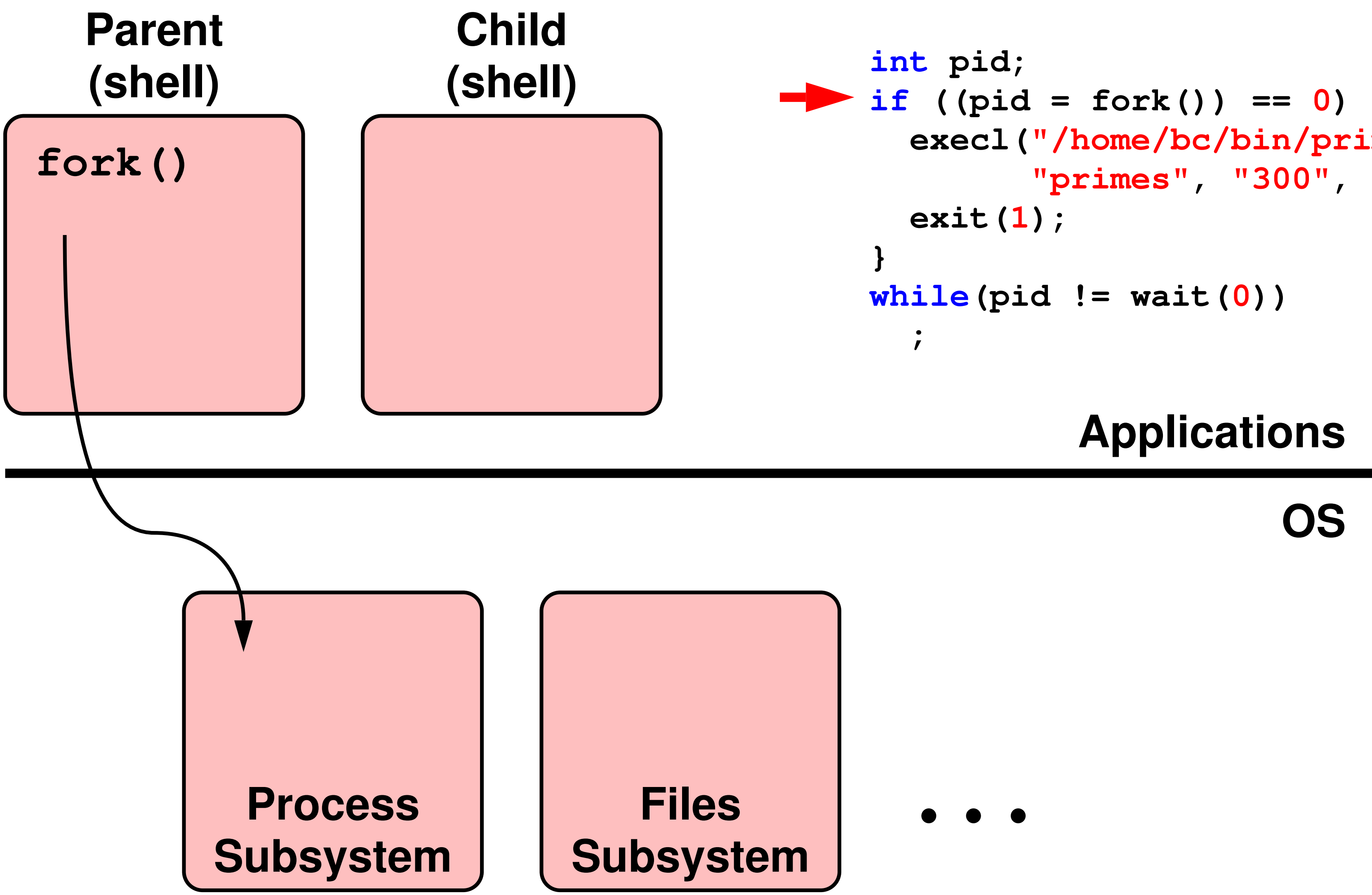
• • •

# Put It All Together



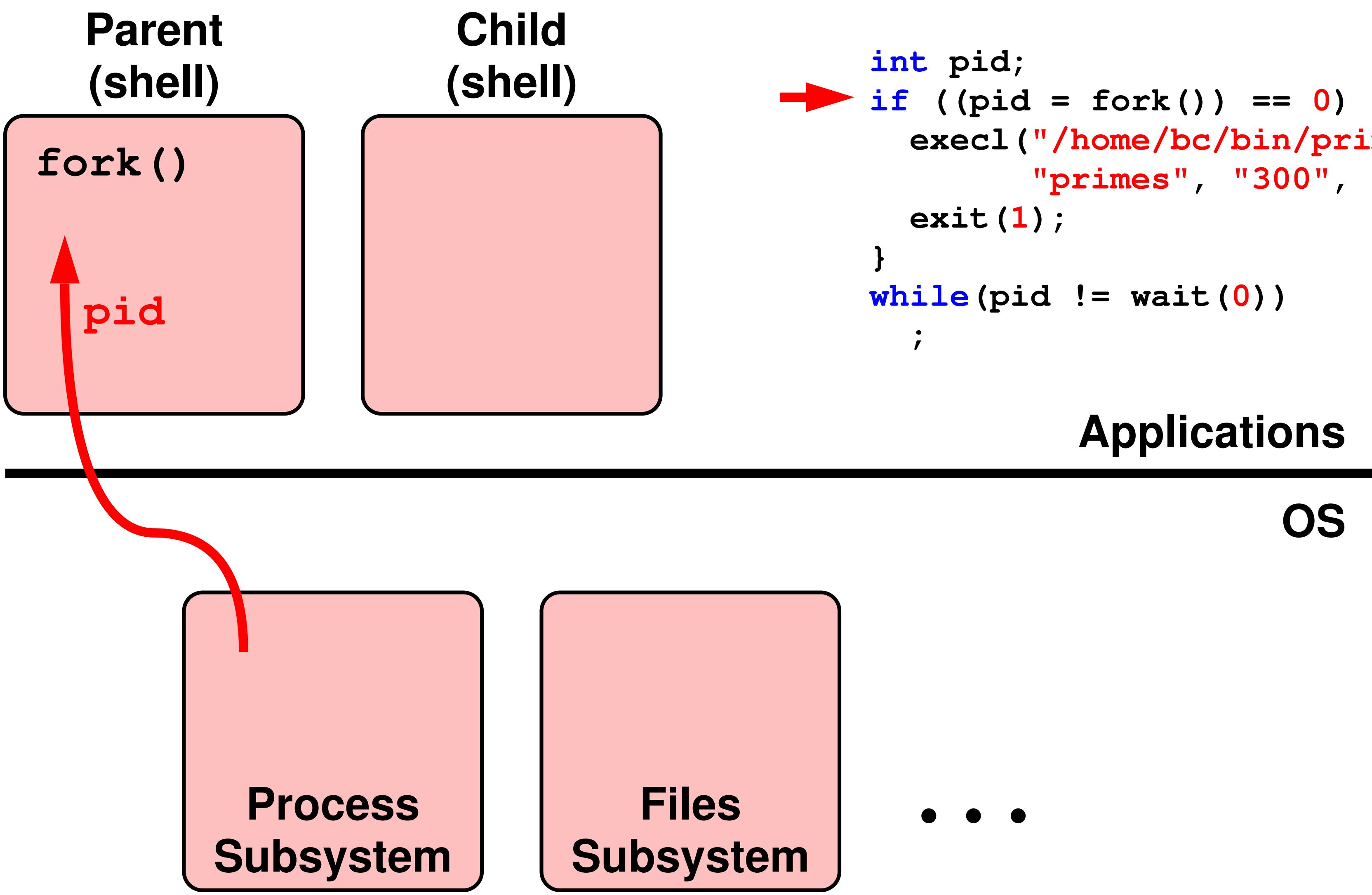

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

# Put It All Together



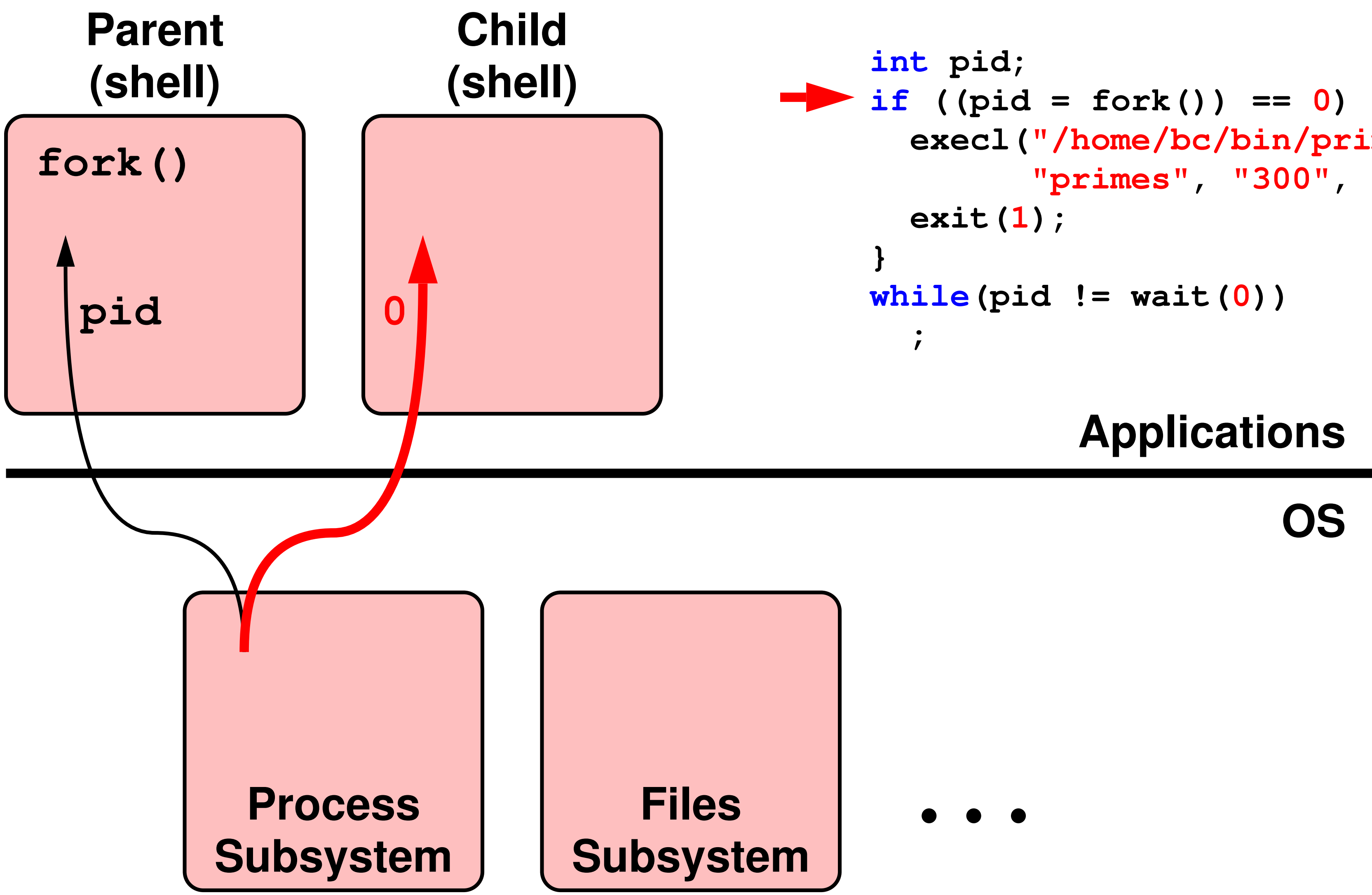

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

# Put It All Together



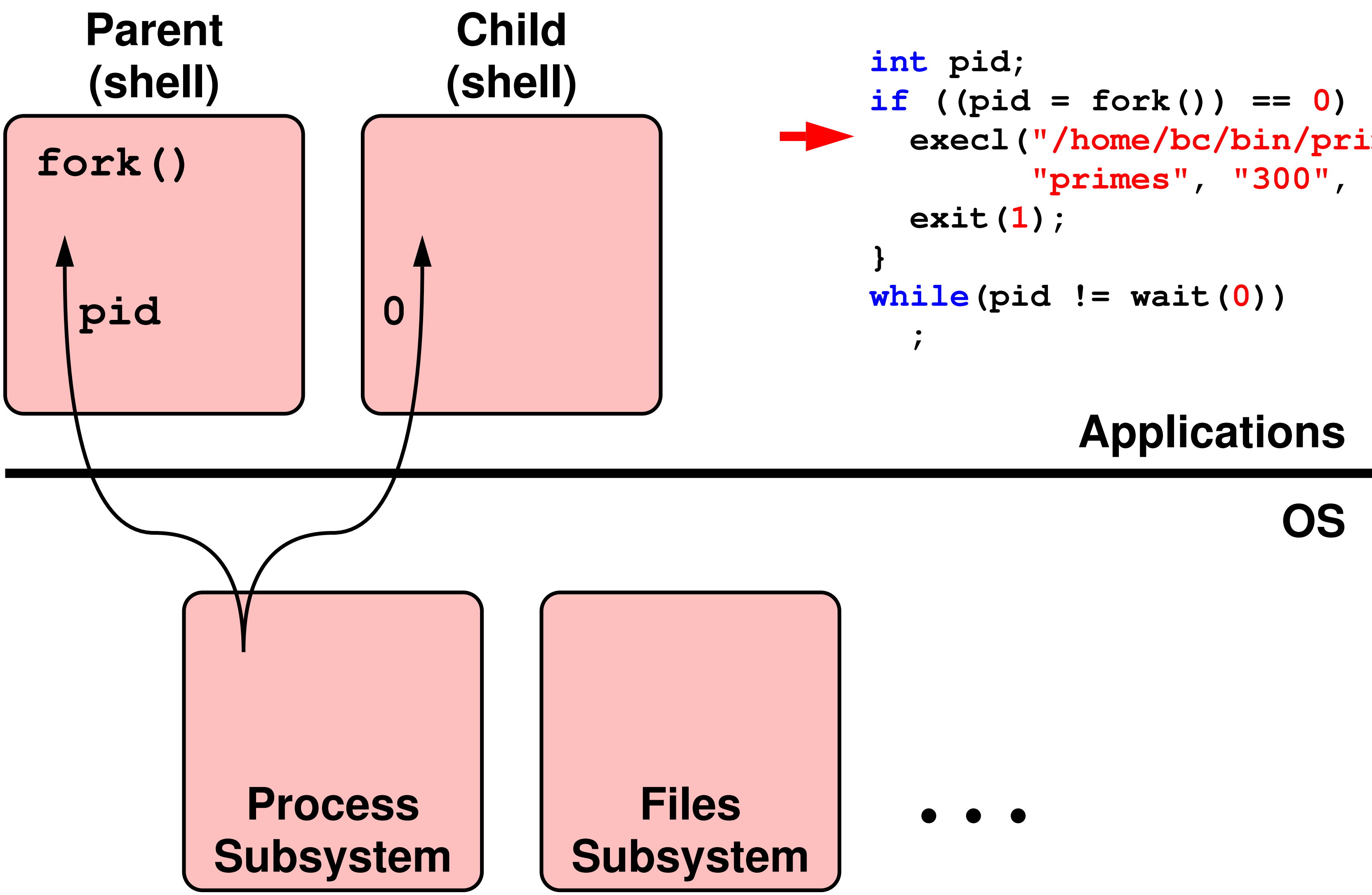

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

# Put It All Together




```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

# Put It All Together




```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

# Put It All Together

Parent (shell)

`fork()`

Child (shell)

`execl()`

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

Applications

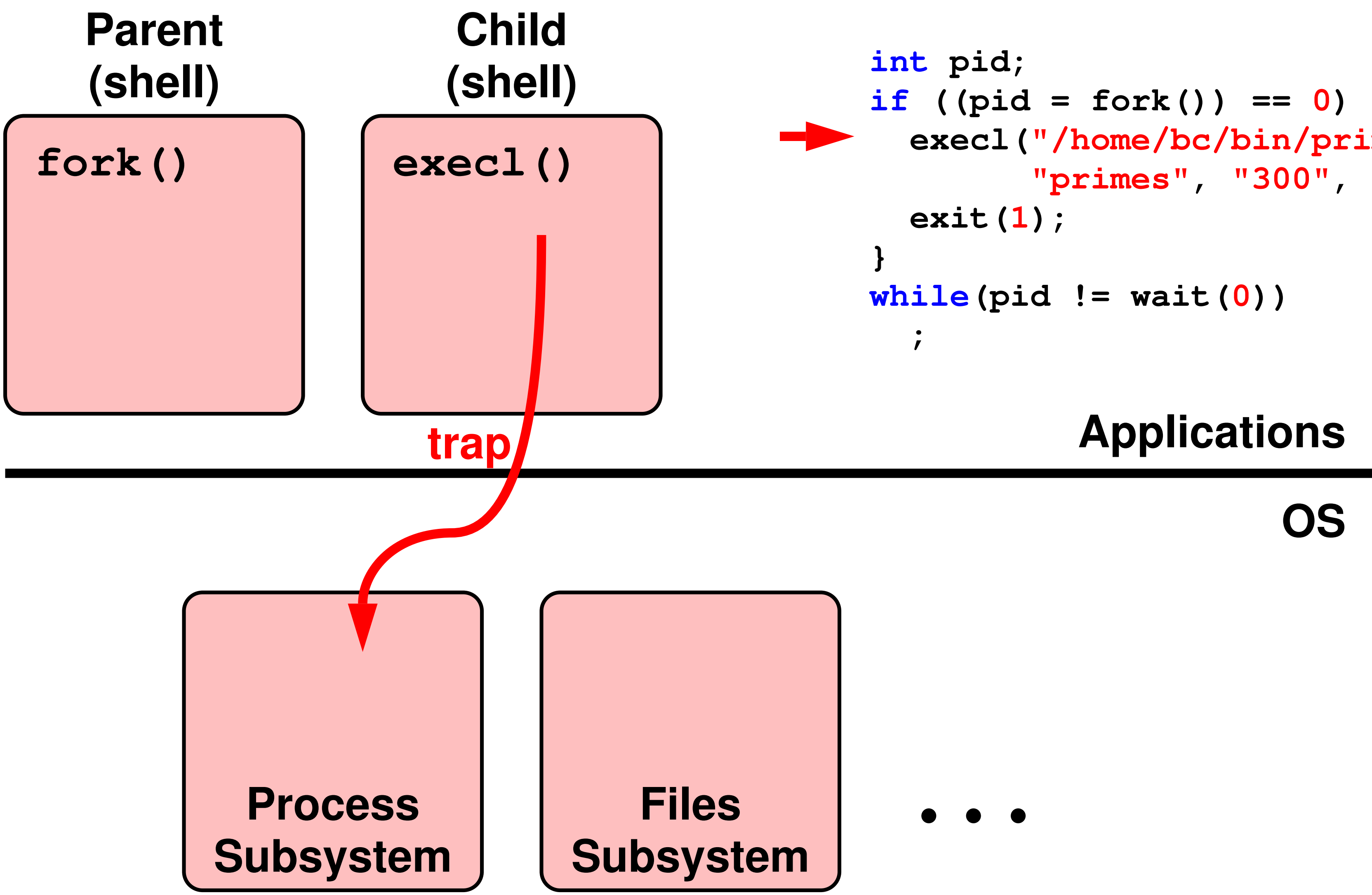
OS

Process Subsystem

Files Subsystem

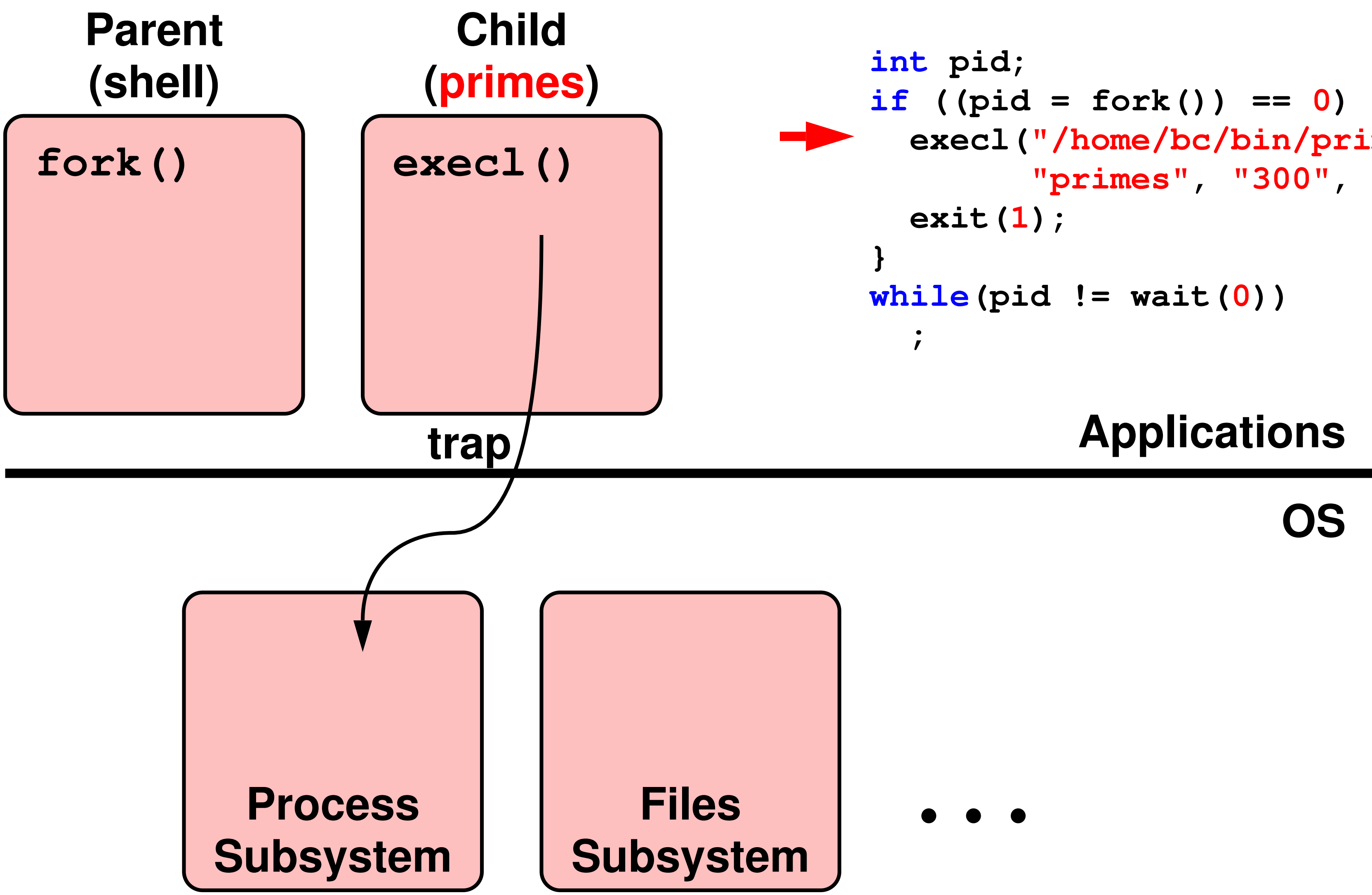
. . .

# Put It All Together




```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

# Put It All Together




```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

# Put It All Together

**Parent (shell)**

```
fork()
wait()
```

**Child (primes)**

```
execl()
```

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

**Applications**

**OS**

**Process Subsystem**

**Files Subsystem**

. . .

# Put It All Together

Parent
(shell)

fork()
wait()

Child
(primes)

execl()

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

trap

Applications

OS

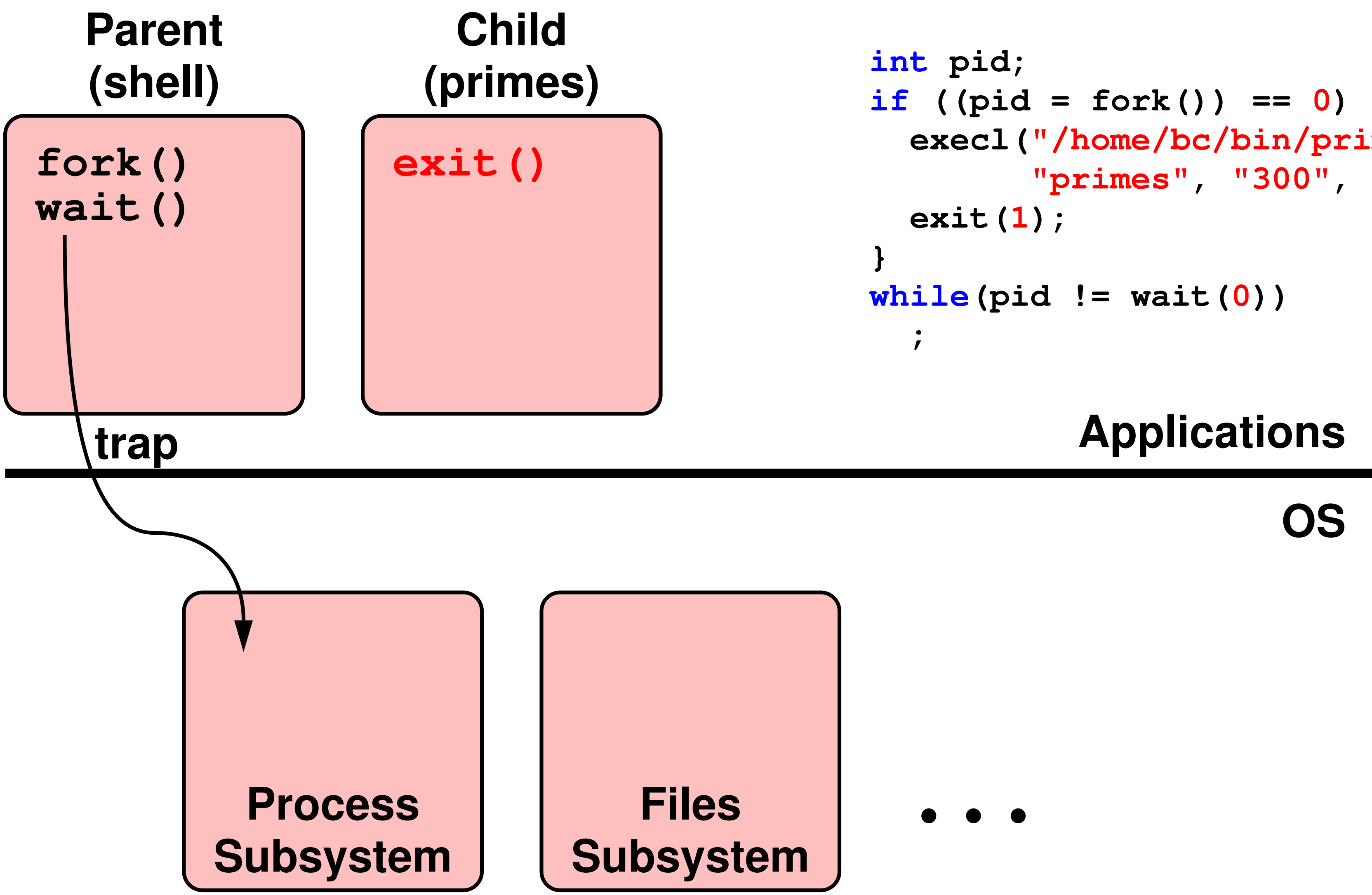
Process Subsystem

Files Subsystem

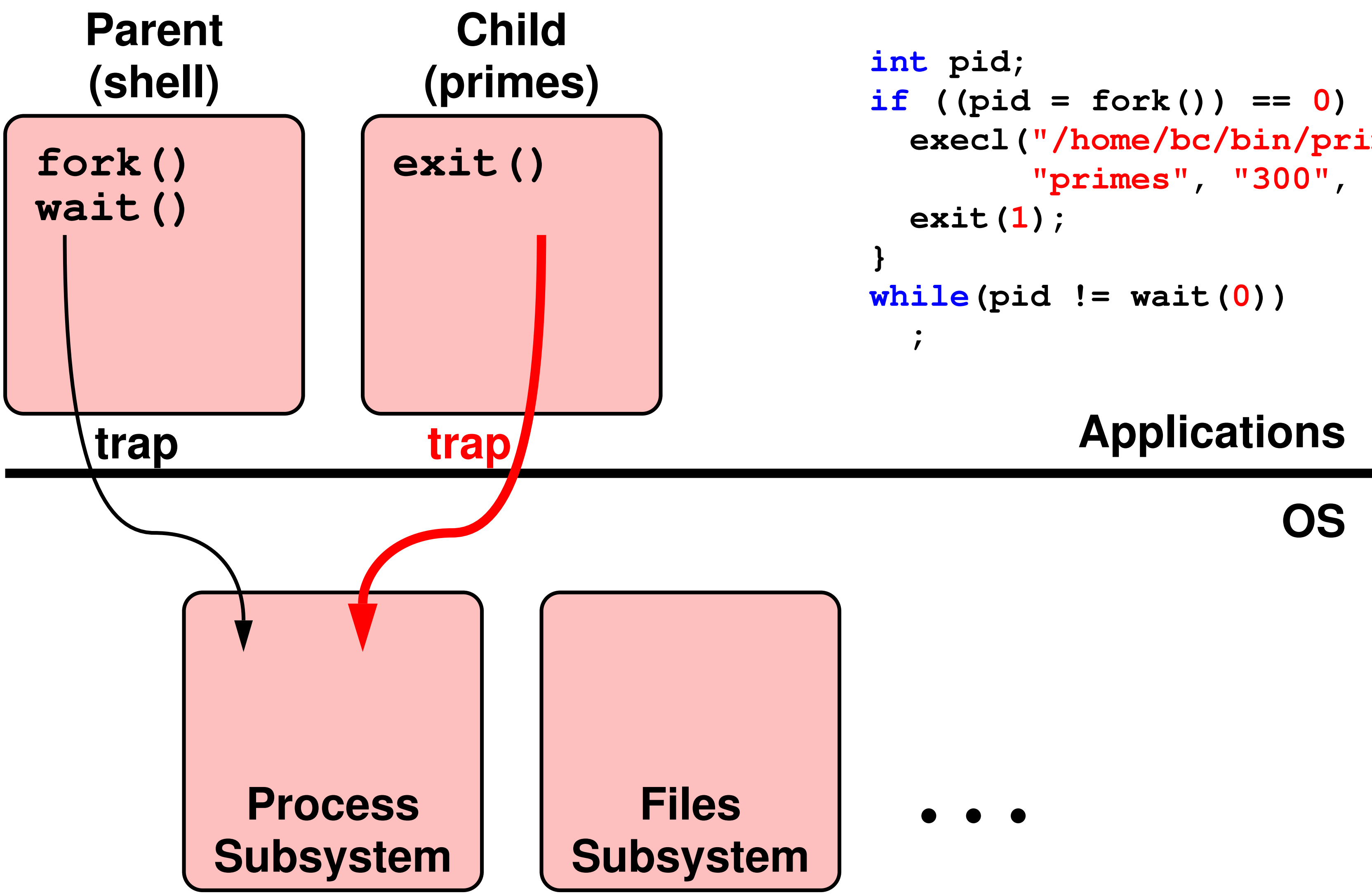
. . .

# Put It All Together

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

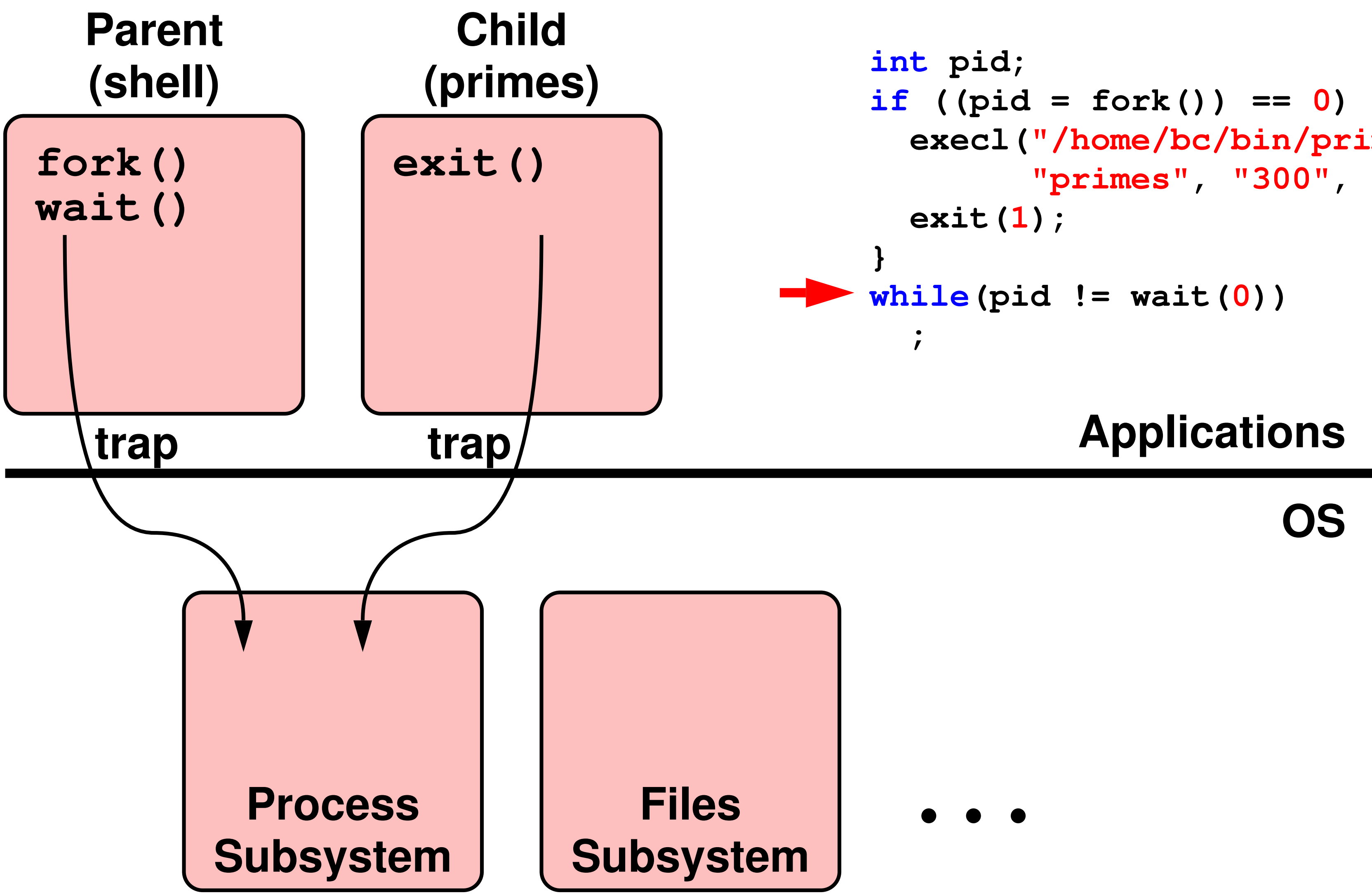


# Put It All Together



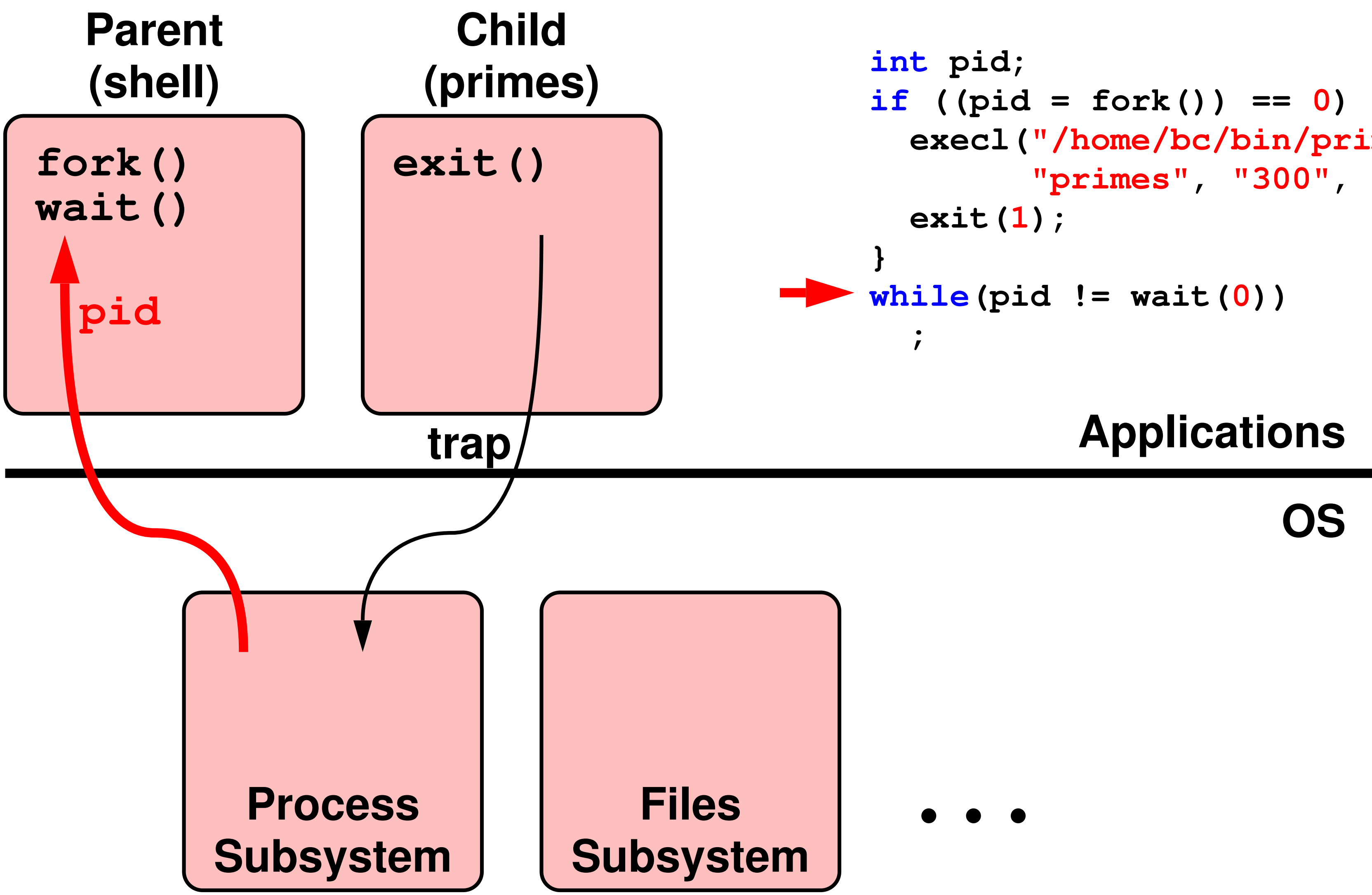

```c
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

# Put It All Together




```c
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

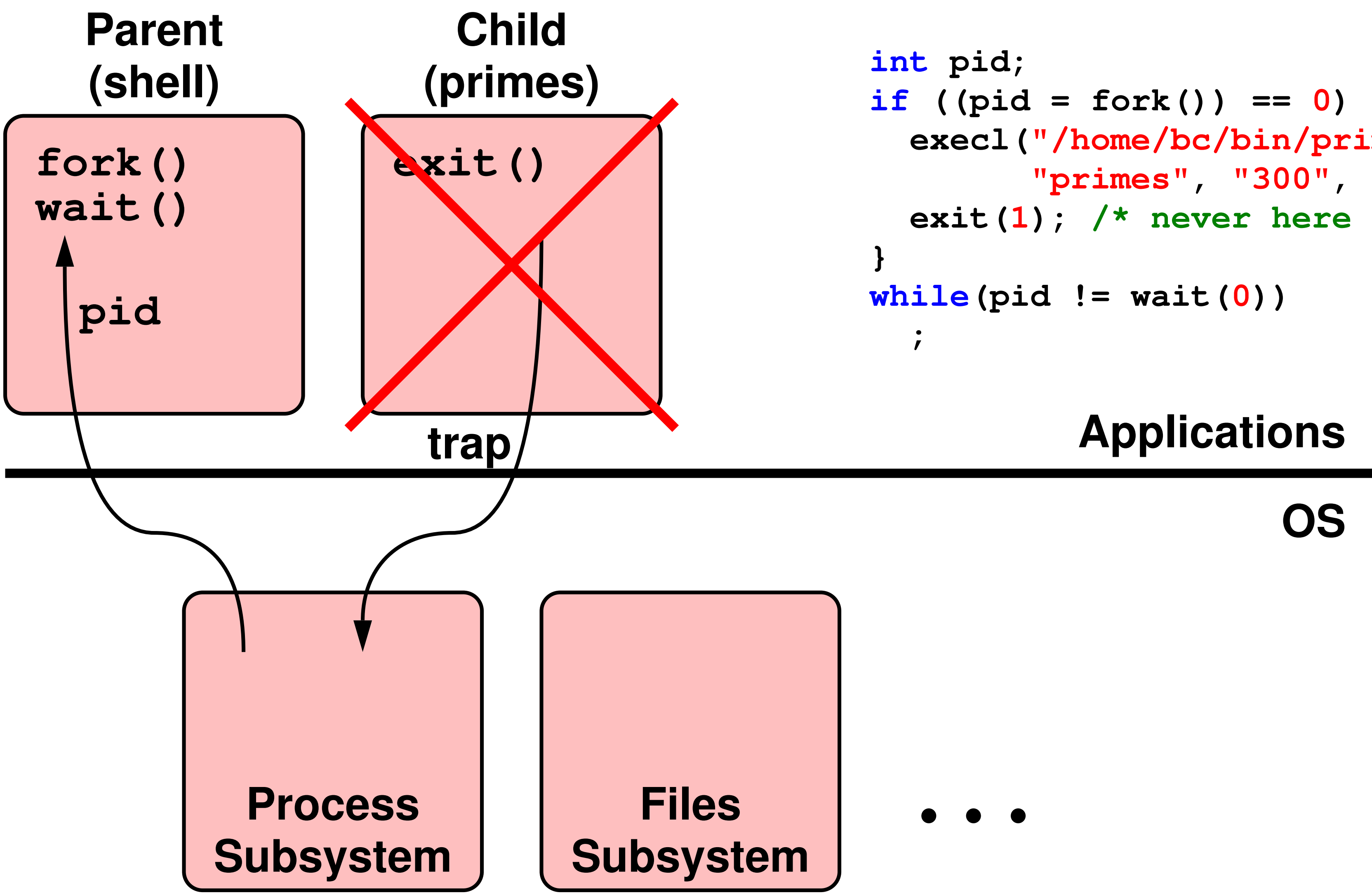
# Put It All Together




```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

# Put It All Together

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1); /* never here */
}
while(pid != wait(0))
  ;
```

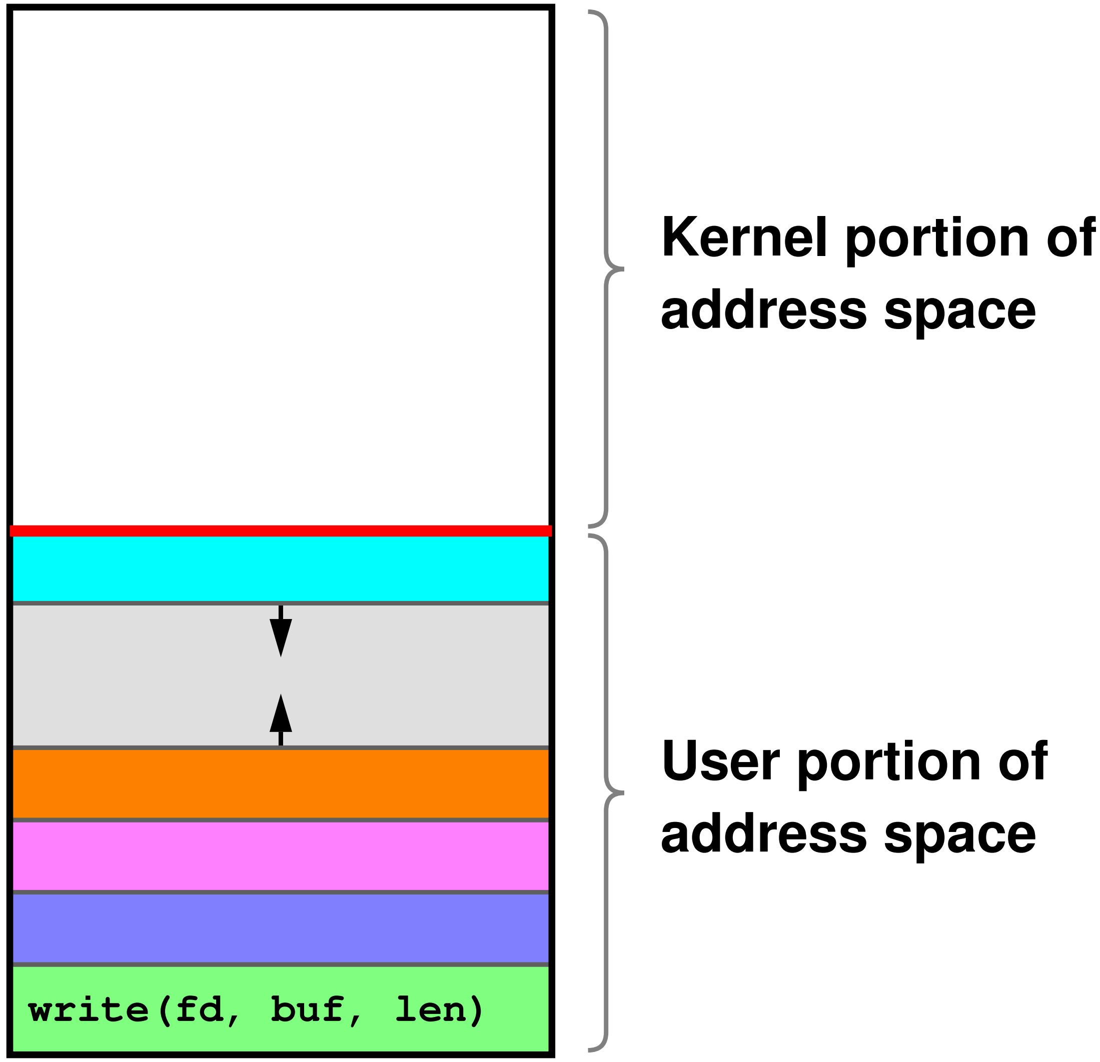


# More On System Calls

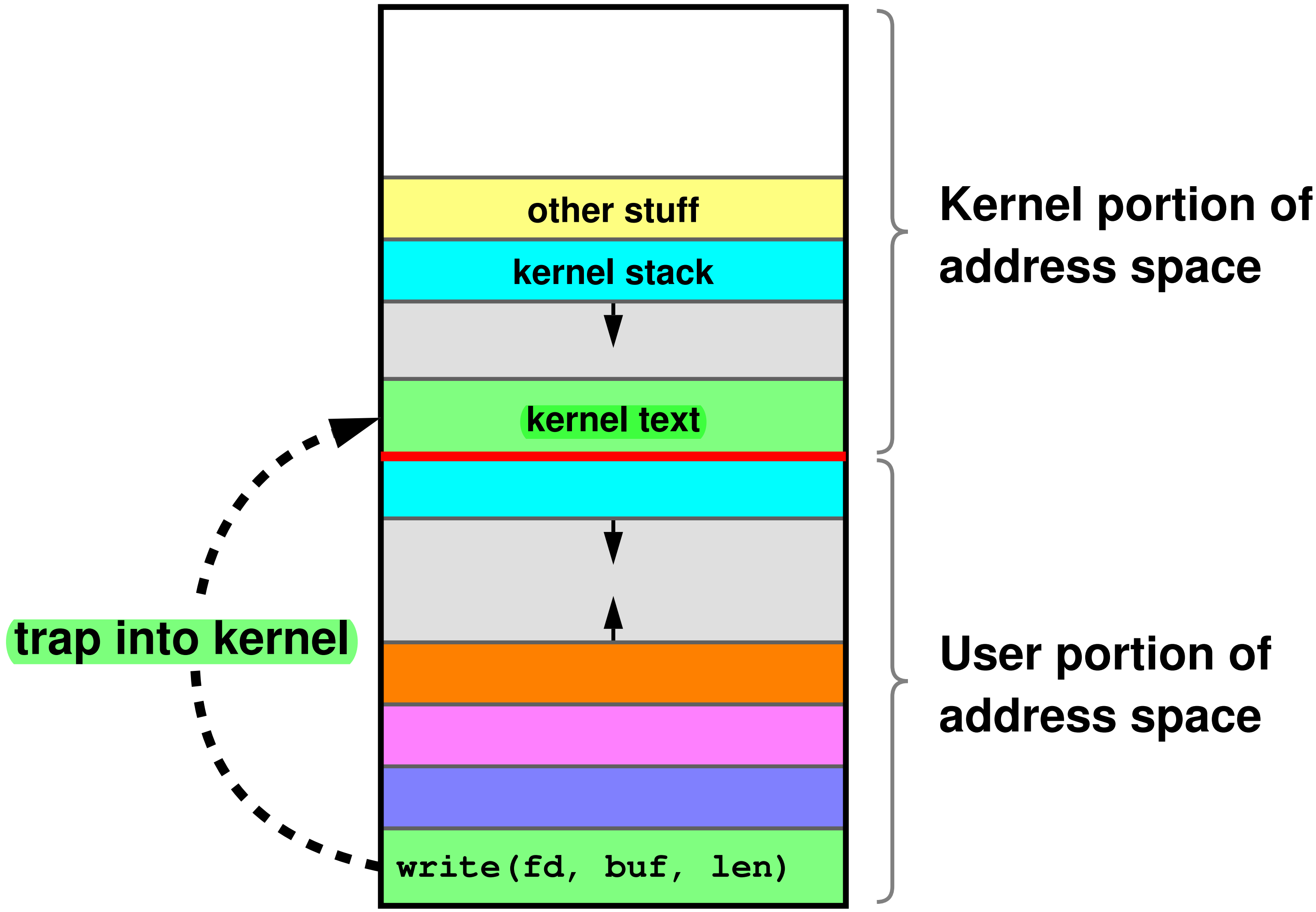
- Sole interface between user and kernel
- Implemented as library routines that execute *"trap" machine instructions* to enter kernel
- Errors indicated by returns of -1; error code is in *errno*

```
if (write(fd, buffer, bufsize) == -1) {
  // error!
  printf("error %d\n", errno);
  // see perror
}
```

# System Calls

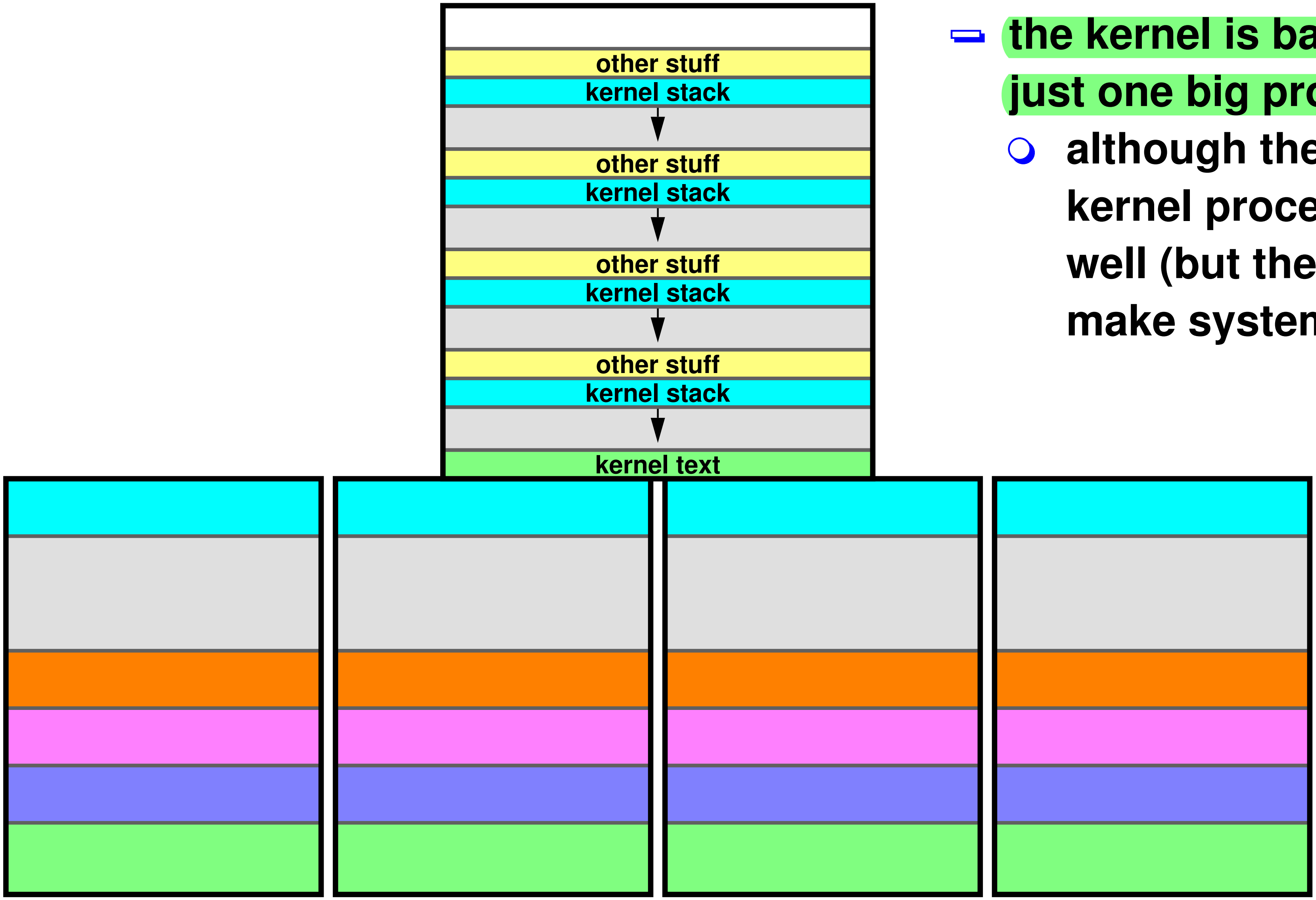


# System Calls




Is this the same *"thread of execution"*?

# Multiple Processes




- the kernel is basically just one big process!
  - although there are kernel processes as well (but they don't make system calls)

# 1.3 A Simple OS

- OS Structure
- Processes, Address Spaces, & Threads
- Managing Processes
- Loading Program Into Processes
- *Files*

# Files

- Our "primes" program wasn't too interesting
  - it has no output!
  - cannot even verify that it's doing the right thing
  - other program cannot use its result
  - how does a process write to someplace *outside the process*?
- The notion of a *file* is our Unix system's sole abstraction for this concept of "someplace outside the process"
  - modern Unix systems have additional abstractions
- Files
  - abstraction of persistent data storage
  - means for fetching and storing data outside a process
    - including disks, another process, keyboard, display, etc.
    - need to *name* these different places
      - hierarchical naming structure
    - part of a process's *extended address space*
      - file "cursor position" is part of "execution context"

# Naming Files

- Directory system
  - shared by all processes running on a computer
    - although each process can have a different view
    - Unix provides a means to restrict a process to a subtree
      - by redefining what "root" means for the process
  - name space is outside the processes
    - a user process provides the name of a file to the OS
    - the OS returns a *handle* to be used to access the file
      - after it has verified that the process is allowed *access* along the *entire path*, starting from root
    - user process uses the handle to read/write the file
      - avoid access checks
- Using a handle to refer to an object managed by the kernel is an important concept
  - handles are essentially an *extension* to the process's *address space*
    - can even survive execs!

# The File Abstraction

- A file is a simple array of bytes
- Files are made larger by writing beyond their current end
- Files are named by paths in a naming tree
- System calls on files are *synchronous*
- File API
  - `open()`, `read()`, `write()`, `close()`
  - e.g., `cat`

# File Handles (File Descriptors)

```
int fd;
char buffer[1024];
int count;
if ((fd = open("/home/bc/file", O_RDWR) == -1) {
  // the file couldn't be opened
  perror("/home/bc/file");
  exit(1);
}
if ((count = read(fd, buffer, 1024)) == -1) {
  // the read failed
  perror("read");
  exit(1);
}
// buffer now contains count bytes read from the file
```

- what is O_RDWR?
- what does `perror()` do?
- *cursor* position in an opened file depends on what functions/system calls you use
  - what about C++?

# Standard File Descriptors

- Standard File Descriptors
  - 0 is `stdin` (by default, the keyboard)
  - 1 is `stdout` (by default, the display)
  - 2 is `stderr` (by default, the display)

```
main() {
  char buf[BUFSIZE];
  int n;
  const char *note = "Write failed\n";

  while ((n = read(0, buf, sizeof(buf))) > 0)
    if (write(1, buf, n) != n) {
      (void)write(2, note, strlen(note));
      exit(EXIT_FAILURE);
    }
  return(EXIT_SUCCESS);
}
```

# Back to Primes

- Have our primes program write out the solution, i.e., the `primes[]` array

```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
   ...
   for (i=1; i<nprimes; i++) {
     ...
   }
   if (write(1, prime, nprimes*sizeof(int)) == -1) {
     perror("primes output");
     exit(1);
   }
   return(0);
}
```

  - the output is not readable by human

# Human-Readable Output

```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
  ...
  for (i=1; i<nprimes; i++) {
    ...
  }
  for (i=0; i<nprimes; i++) {
    printf("%d\n", prime[i]);
  }
  return(0);
}
```

# Allocation of File Descriptors

- 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);
```

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

# 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

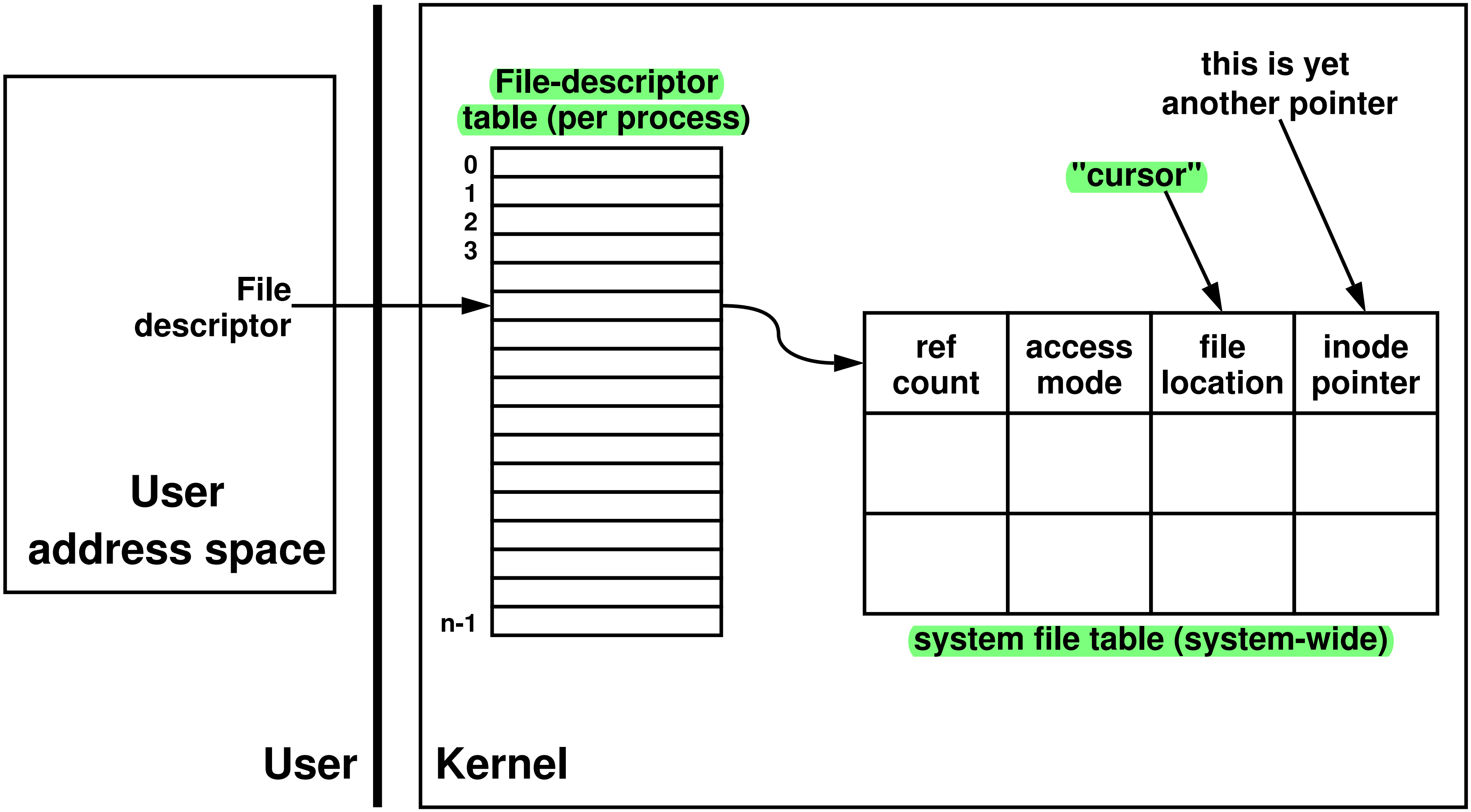
```
% cat < /home/bc/Output
```

  - when the "`cat`" program reads from file descriptor 0, it would get the data byes 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
- *Context* information must be maintained by the OS and not directly by the user program
  - 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/cs402-f13***

# Overview

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

# Concurrency

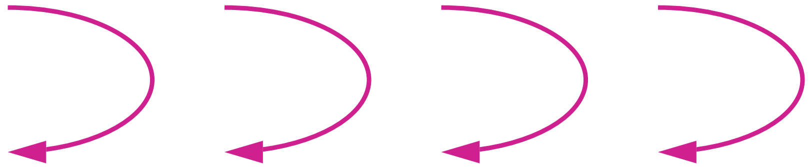
- **Many things occur simultaneously in the OS**
  - **e.g., data coming from a disk, data coming from the network, data coming from the keyboard, mouse got clicked, jobs need to get executed**

- **If you have multiple processors, you may be able to handle things in parallel**
  - **that's real concurrency**

- **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 multiple processors**

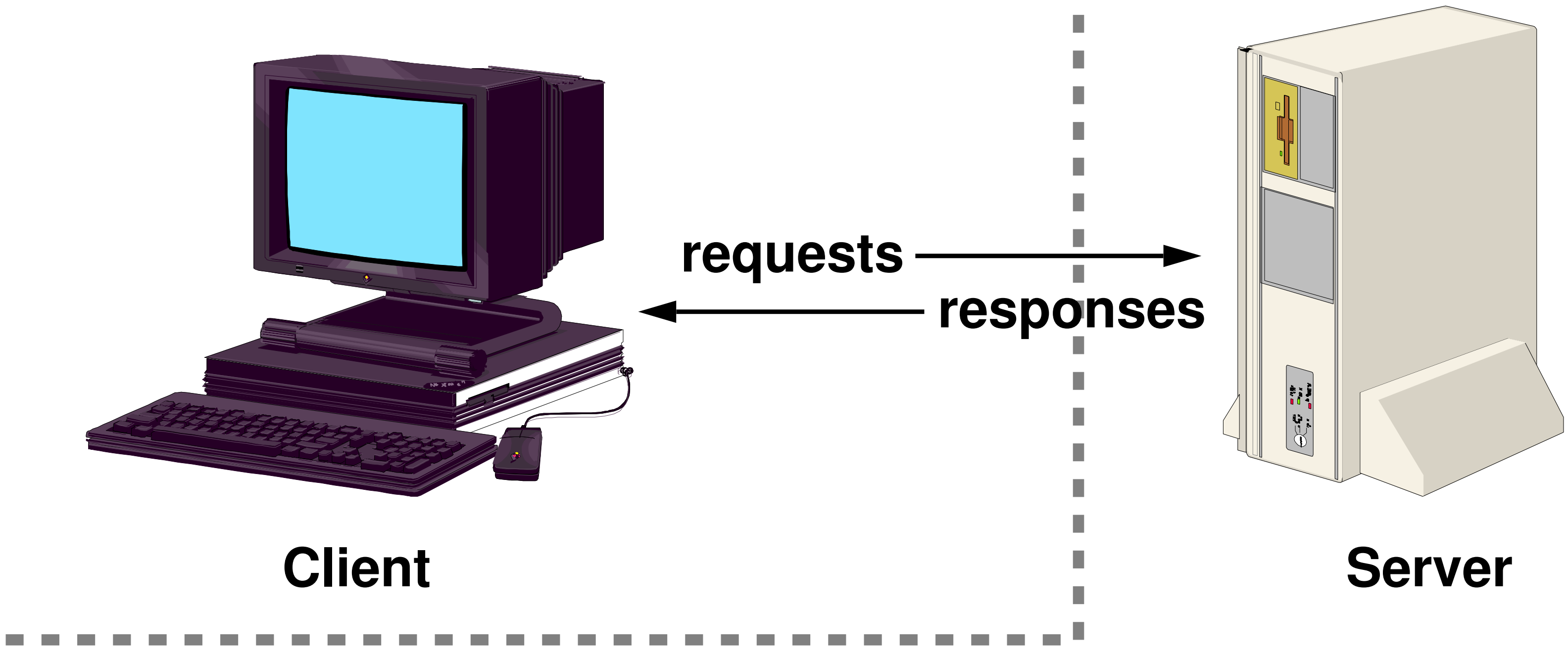
- **If you have concurrency, you have to have *concurrency control***

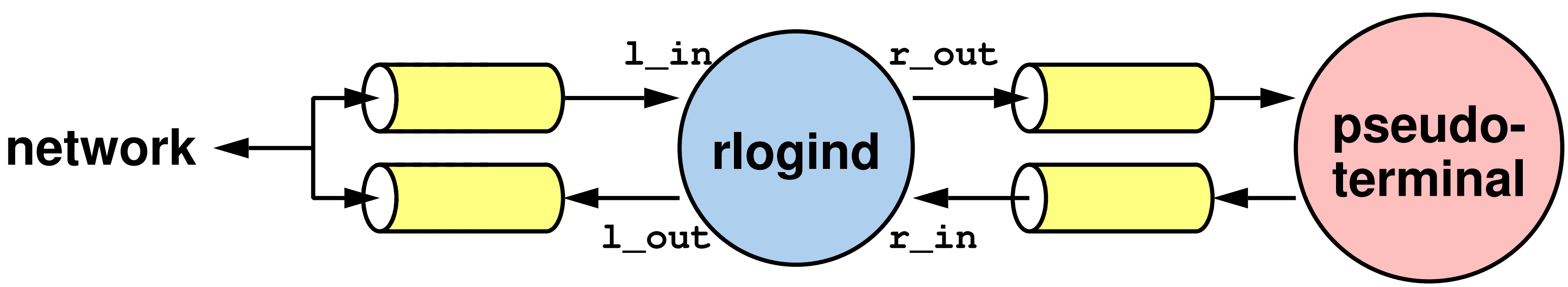
# Why Threads?



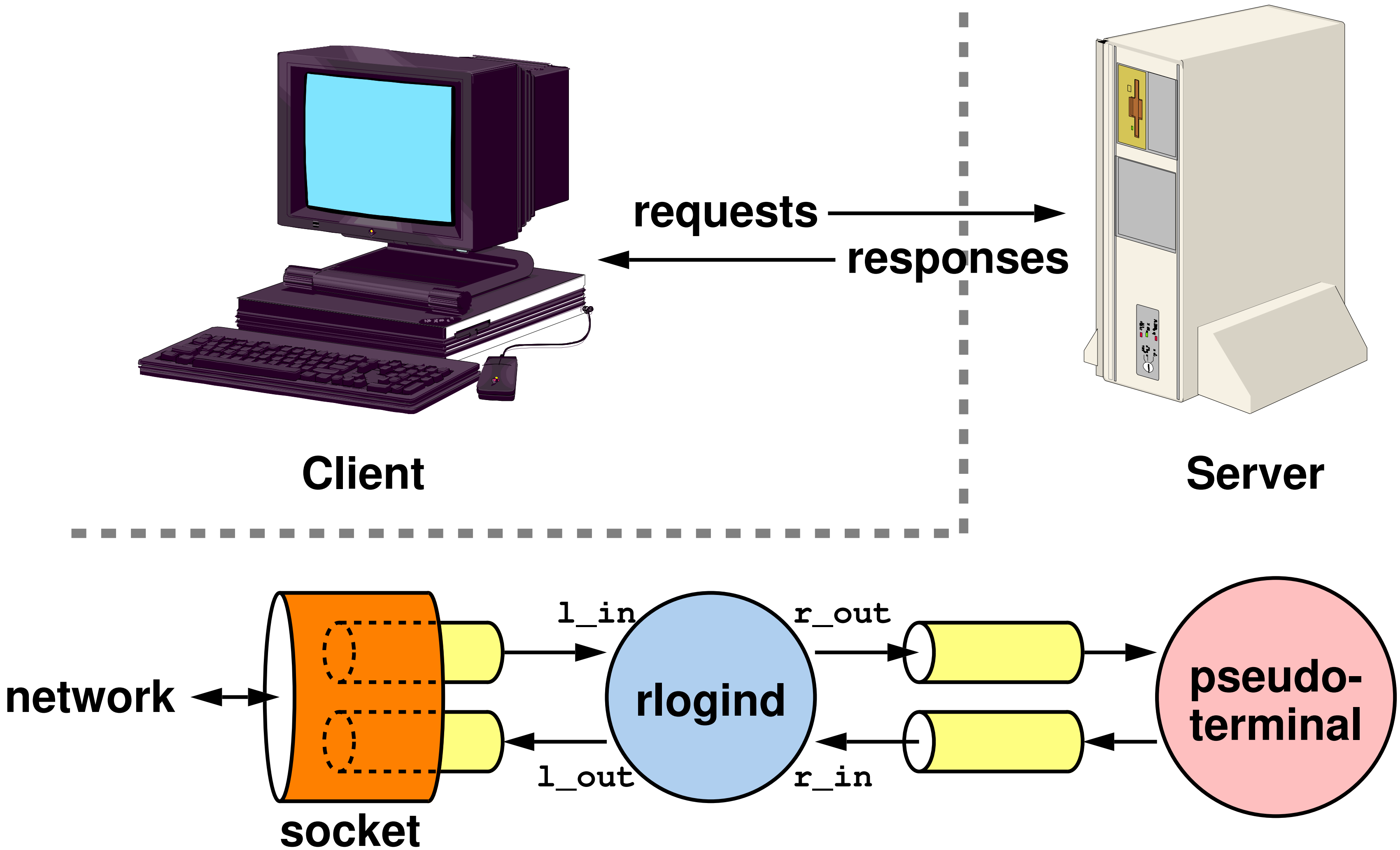
- **Many things are easier to do with threads**
  - ***multithreading*** **is a** ***powerful paradigm***
  - **makes your design** ***cleaner*****, and therefore, less buggy**
- **Many things run faster with threads**
  - **if you are just waiting, don't waste CPU cycles, give the CPU to someone else,** ***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`



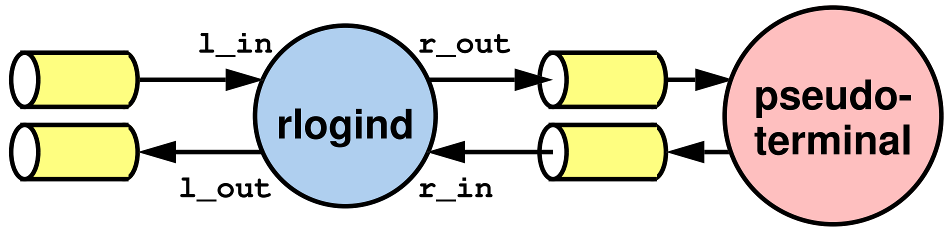
# A Simple Example: `rlogind`




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

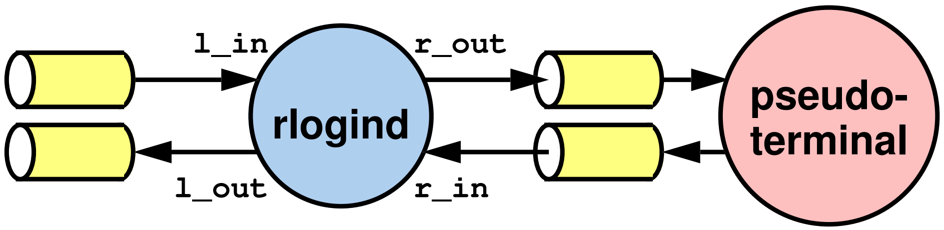
# Life Without Threads

```
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; }
    }
```

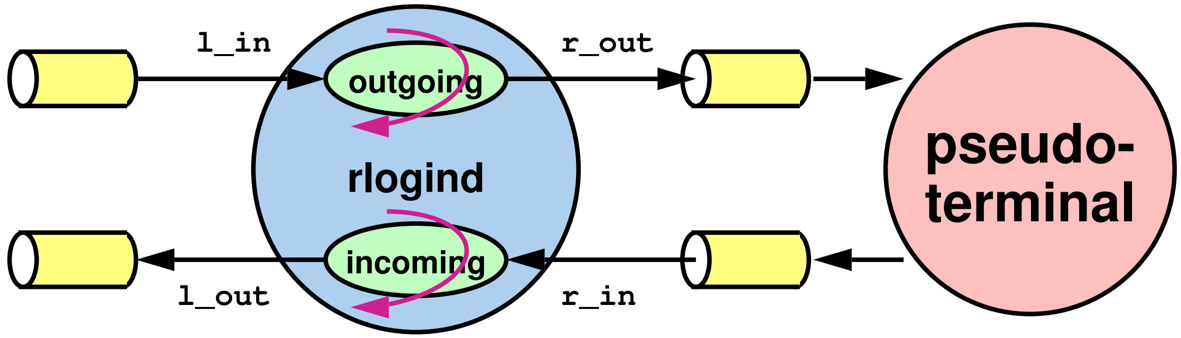


# Life Without Threads

```
    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

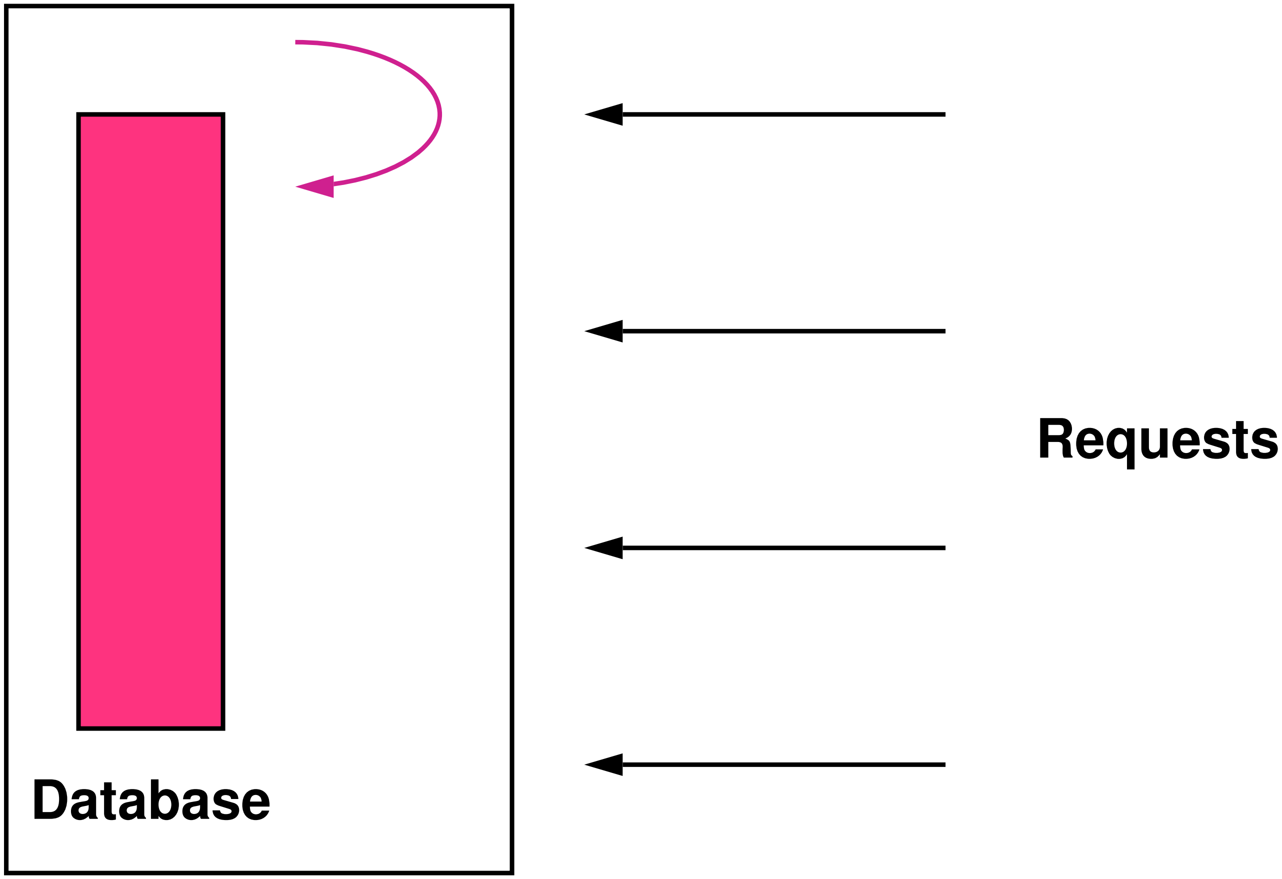



```
incoming(int r_in, int l_out) {
  int eof = 0;
  char buf[BSIZE];
  int size;

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

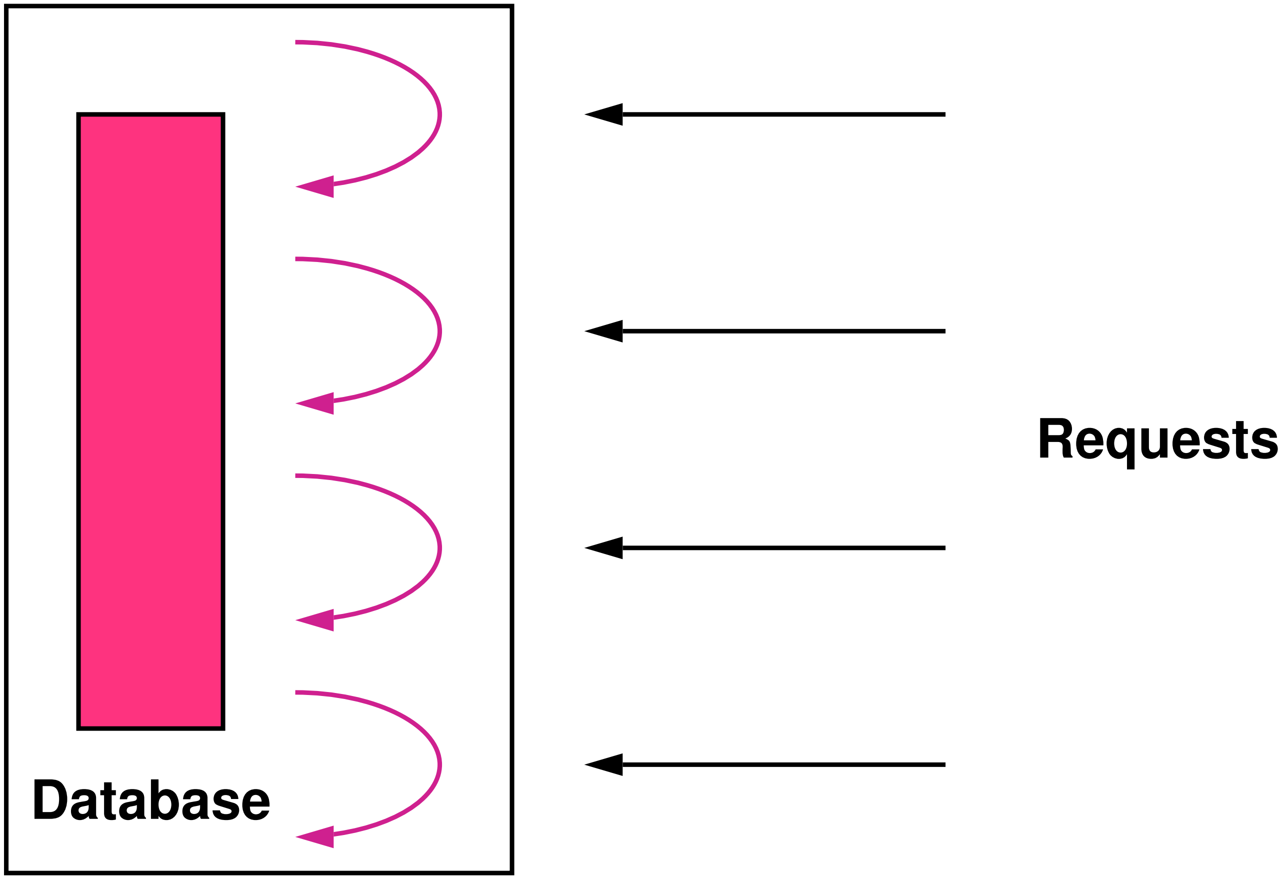
```
outgoing(int l_in, int r_out) {
  int eof = 0;
  char buf[BSIZE];
  int size;

  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.
```