Lecture 1: Operating Systems

Shubhani

COL331/COL633

Class details

- Mixed undergraduate and graduate
- Instructor: Dr. Sorav Bansal
- Web page

  https://iitd-plos.github.io/os/2020
- Piazza

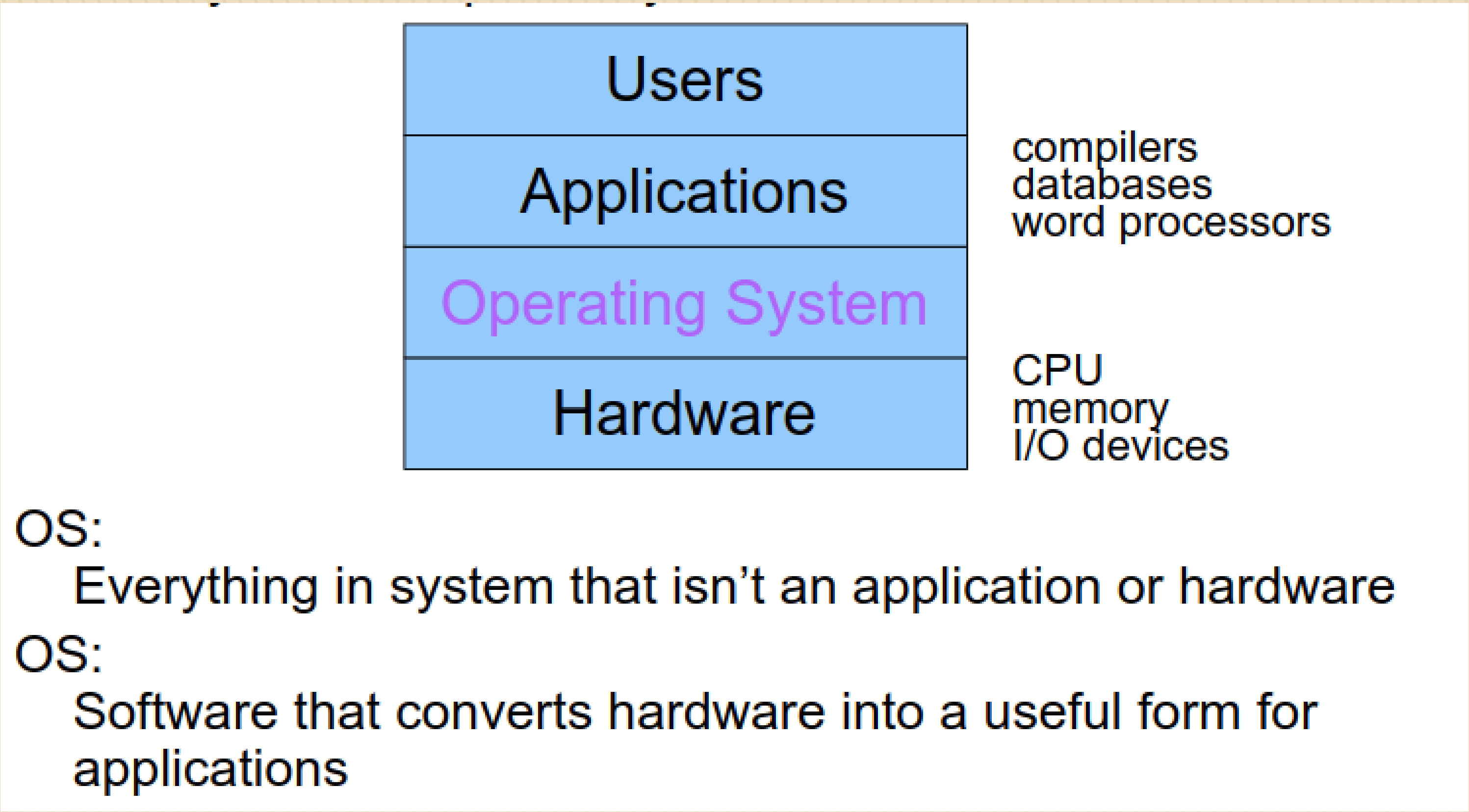
  https://piazza.com/iit_delhi/fall2019/col331col633

Course Outline

- Lectures
  - Understand operating system design and implementation
- Reading
  - Xv6 book + source code
- Labs
  - Hands-on experience extending a small O/S

What is an Operating System?

Operating System (OS)



Users

Applications

Operating System

Hardware

compilers
databases
word processors

CPU
memory
I/O devices

OS:
- Everything in system that isn’t an application or hardware

OS:
- Software that converts hardware into a useful form for applications

Design Approach

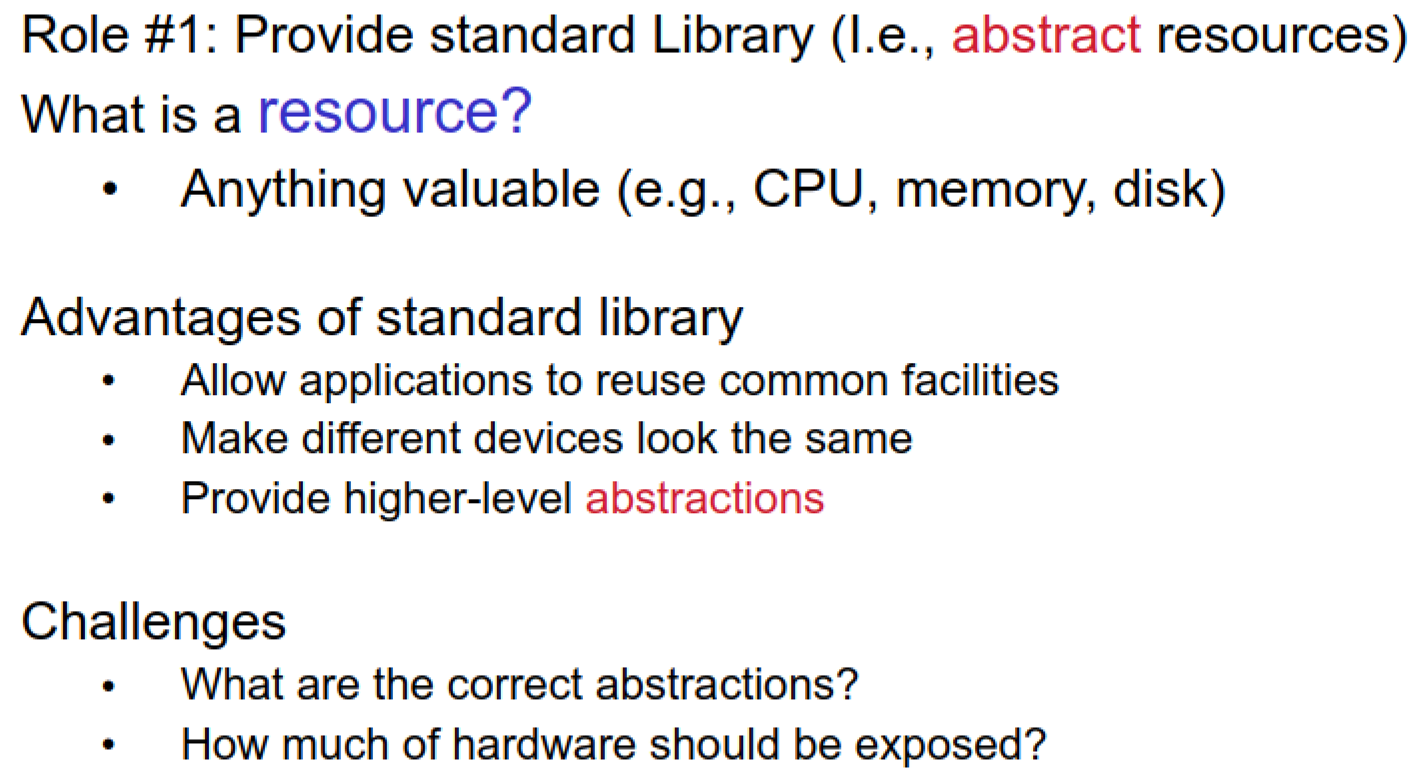
Monolithic Software

- All software components (applications) are contained in a single program and can directly communicate with each other using function calls.

Issues:

- Hard to manage and update
- Trust issues between different programs

Operating System (OS)



Role #1: Provide standard Library (I.e., abstract resources)

What is a resource?

- Anything valuable (e.g., CPU, memory, disk)

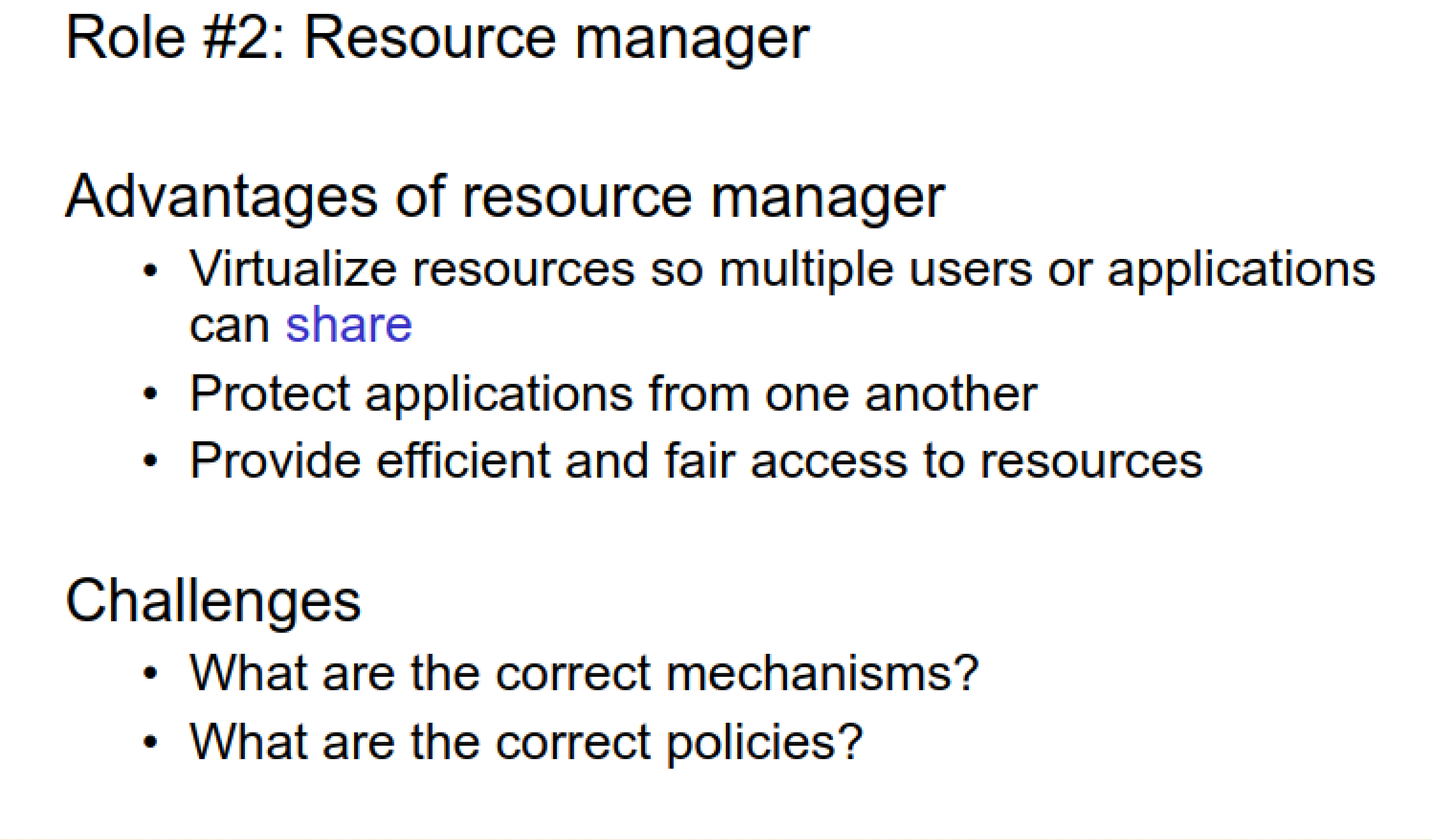
Advantages of standard library

- Allow applications to reuse common facilities
- Make different devices look the same
- Provide higher-level abstractions

Challenges

- What are the correct abstractions?
- How much of hardware should be exposed?

Operating System (OS)



Role #2: Resource manager

Advantages of resource manager

- Virtualize resources so multiple users or applications can share
- Protect applications from one another
- Provide efficient and fair access to resources

Challenges

- What are the correct mechanisms?
- What are the correct policies?

Operating System (OS)

- Abstract the hardware for convenience and portability
- Support a wide range of applications
- Multiplex the hardware among multiple applications
- Isolate applications in order to contain bugs and Security
- Provide high performance

OS research

- Variety of hardwares ranging from embedded devices to multi-core systems
- Reliability
- Performance

What is the right set of abstractions to be provided by an OS?

OS abstractions

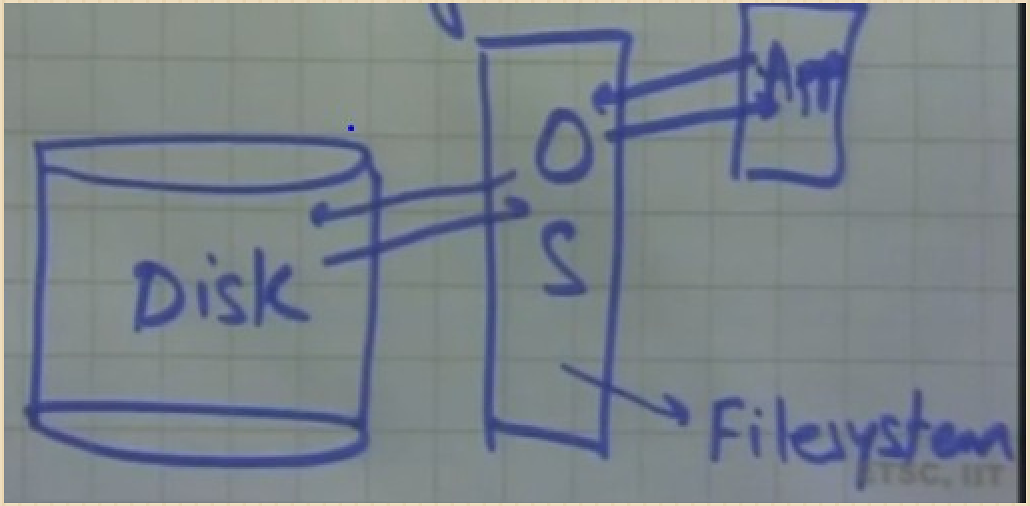
- Filesystem -- disk
- Process -- CPU
- Address space -- memory
- Interactive shell -- execute commands

OS abstractions

- **Filesystem -- disk**
- Process -- CPU
- Address space -- memory
- Interactive shell -- execute commands

File system abstraction

- How should the OS manage a persistent device?
- What are the APIs?

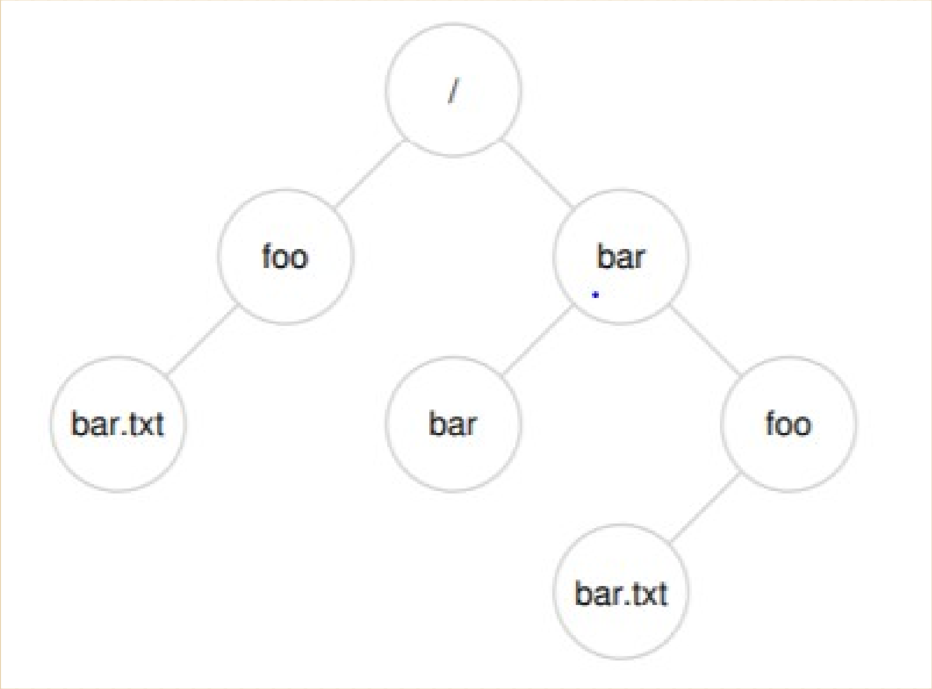


File system abstraction

- File– Identified with file name (human readable) and a OS-level identifier (“inode number”)
- Directory contains other subdirectories and files, along with their inode numbers.
- Stored like a file, whose contents are filename-to-inode mappings

File system abstraction

Files and directories arranged in a tree, starting with root ("/")



OS APIs

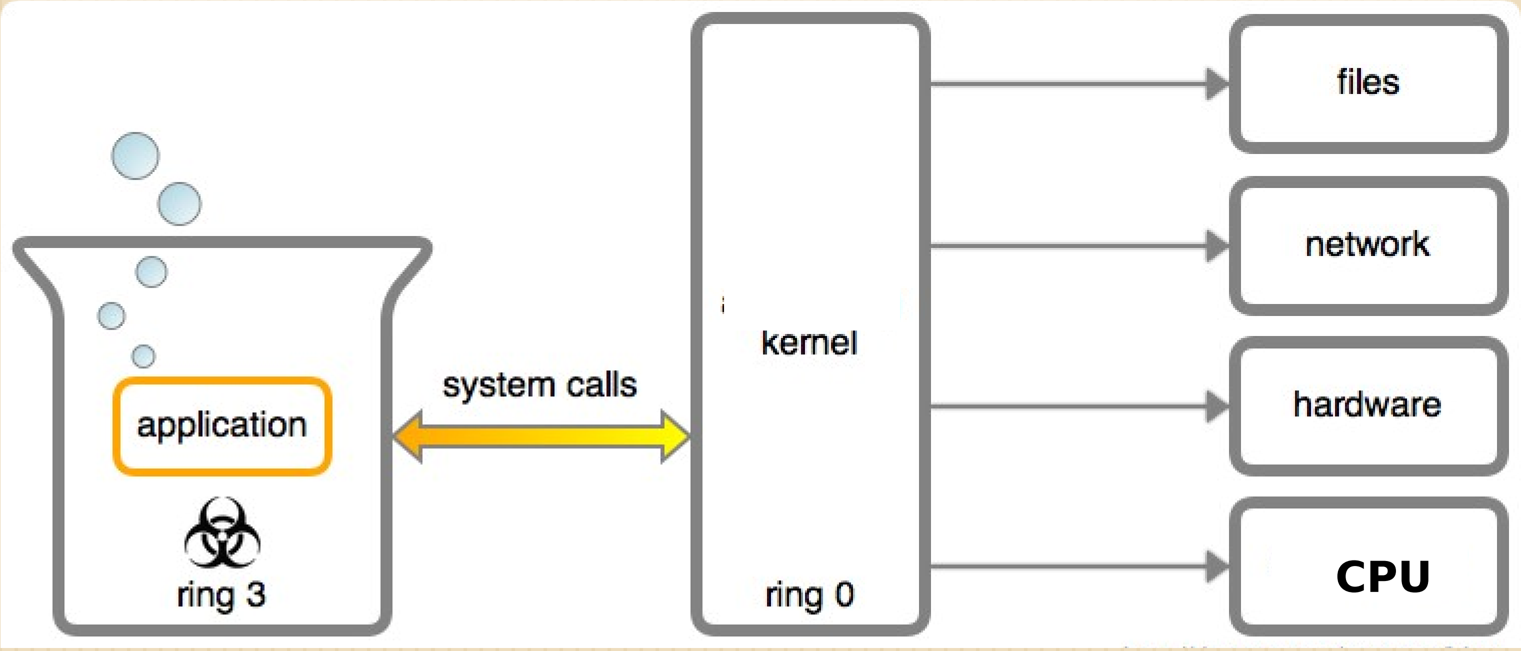
What API does the OS provide to user programs?

- API = Application Programming Interface
- = functions available to write user programs

API provided by OS is a set of “system calls” – System call is a function call into OS code that runs at

- a higher privilege level of the CPU
- Sensitive operations (e.g., access to hardware) are allowed only at a higher privilege level

OS APIs or System calls



Creating Files

int fd = open("filename")

- Returns a number called “file descriptor”
- A file descriptor (fd) is just an integer, private per process
- Existing files must be opened before they can be read/written, Also uses open system call, and returns fd
- All other operations on files use the file descriptor
- ***close()*** system call closes the file

Reading/Writing Files

Reading/writing files: read()/write() system calls

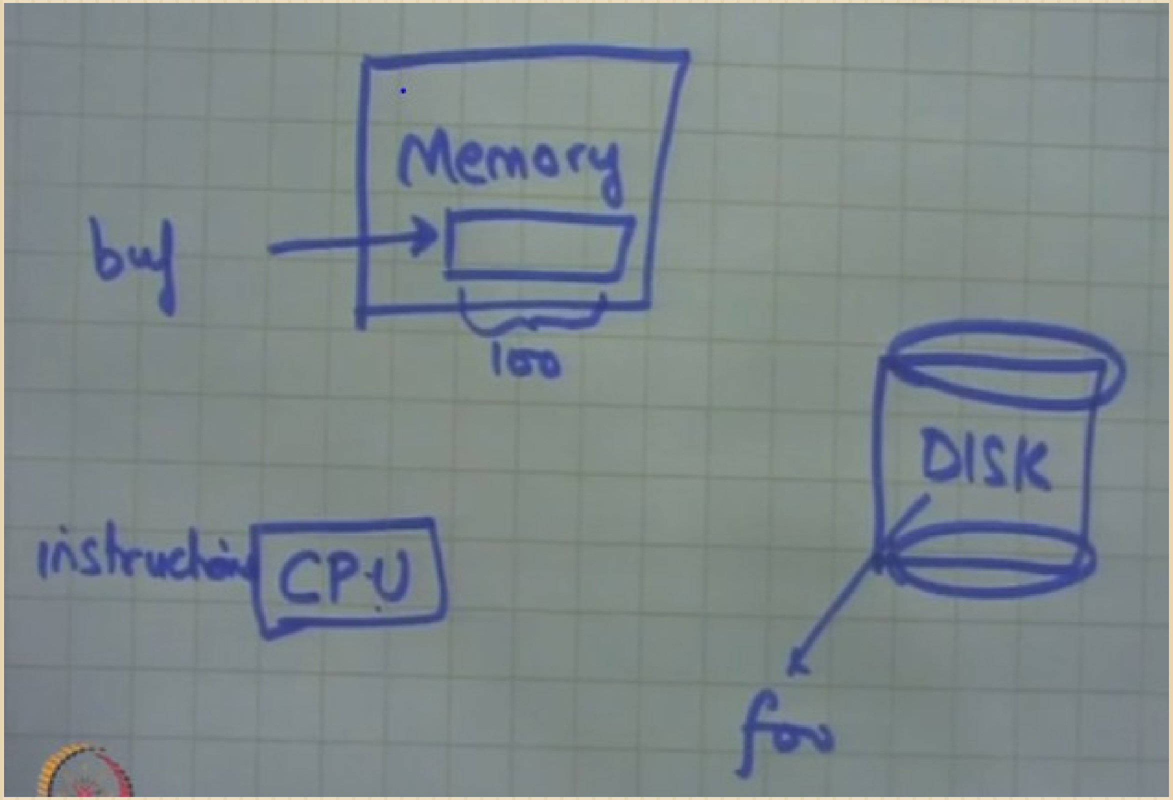
- Arguments: file descriptor, buffer with data, size

read(fd, buf, 100)

write(fd, buf, 100)

File system abstraction

```
int fd = open("foo")
read(fd, buf, 100)
write(fd, buf, 100)
close(fd)
```



OS abstractions

- Filesystem -- disk
- **Process -- CPU**
- Address space -- memory
- Interactive shell -- execute commands

Process Abstraction

- OS provides the process abstraction
    - Process: a running program
    - OS creates and manages processes and Loads program executable (code, data) from disk to memory
- Each process has the illusion of having the complete CPU
- OS timeshares CPU between processes
- OS enables coordination between processes

Process Abstraction

- A unique identifier (PID)
- Memory image
  - Code & data (static)
  - Stack and heap (dynamic)
- CPU context: registers
  - Program counter
  - Stack pointer
- File descriptor table
  - Pointers to opened files and devices

Process Abstraction

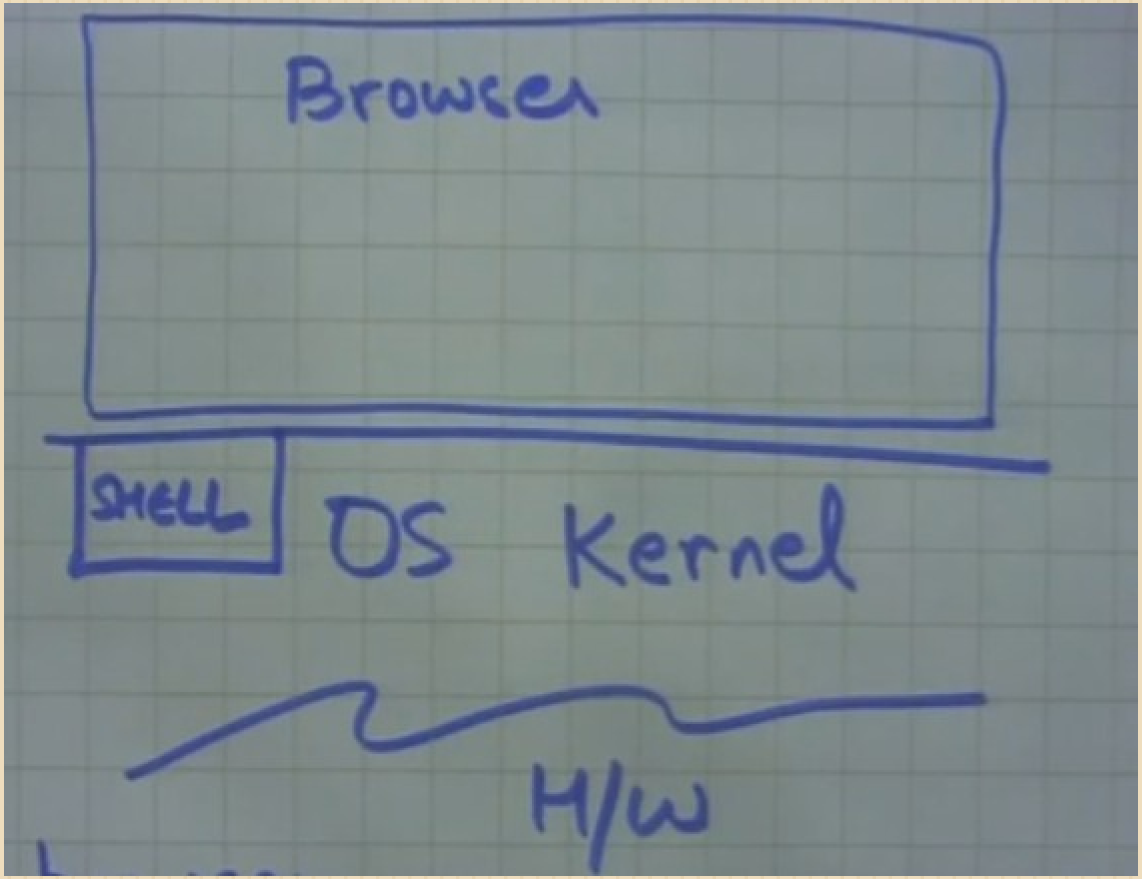
- Allocates memory and creates memory image
    - Loads code, data from disk exe
    - Creates runtime stack, heap
- Opens basic files – STD IN, OUT, ERR
- Initializes CPU registers
    - PC points to first instruction

Interactive Shell

- Special program inside operating system
- Will take commands from user
- Interpret the command as filename
- Loads the filename as a process in memory
- Transfers the control to newly created process

Interactive Shell

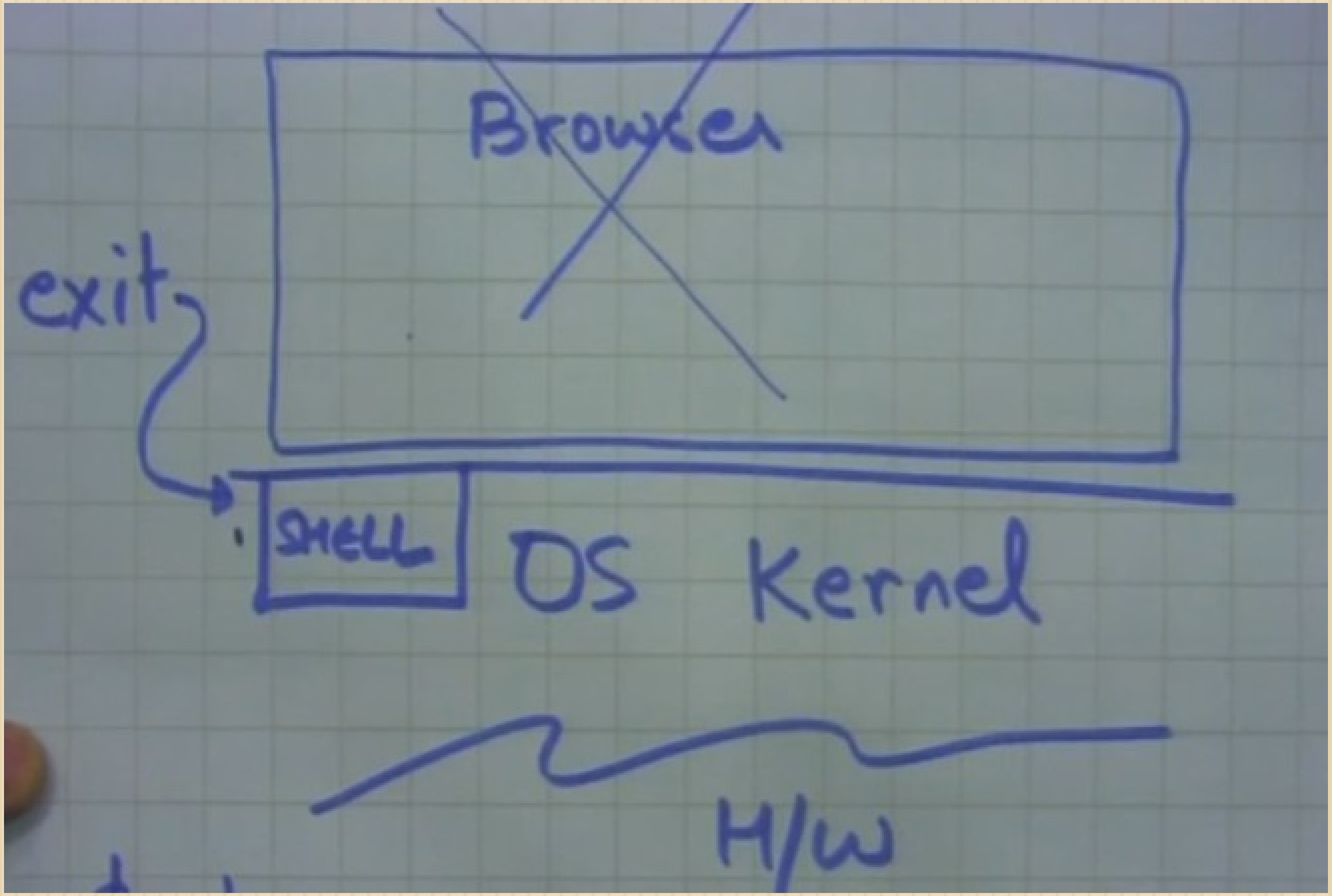
- $ browser



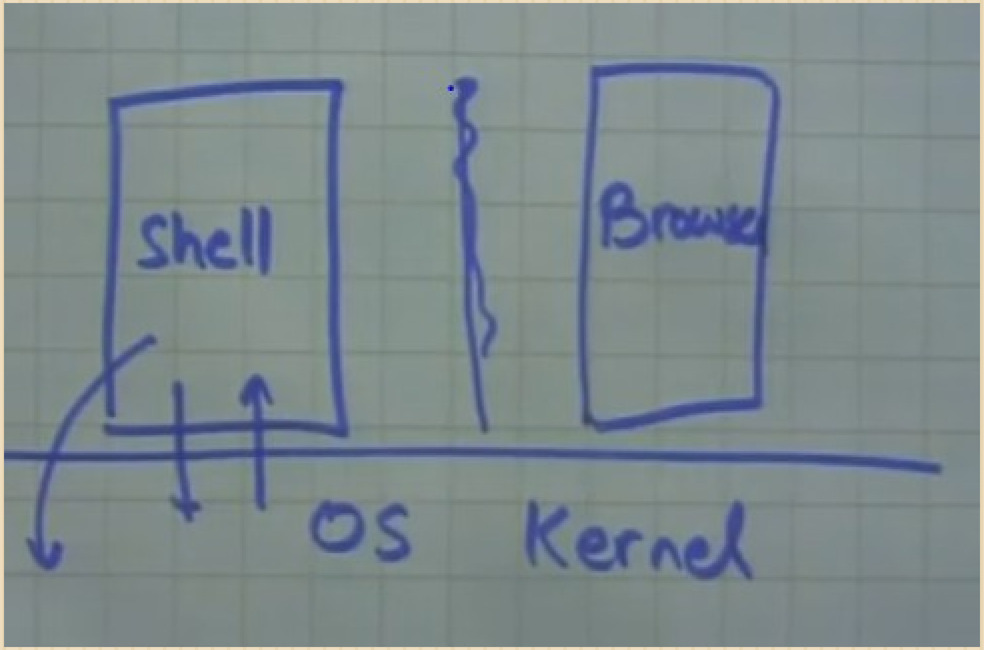
Interactive Shell

- $ browser

  $ |



Interactive Shell



Process related system calls

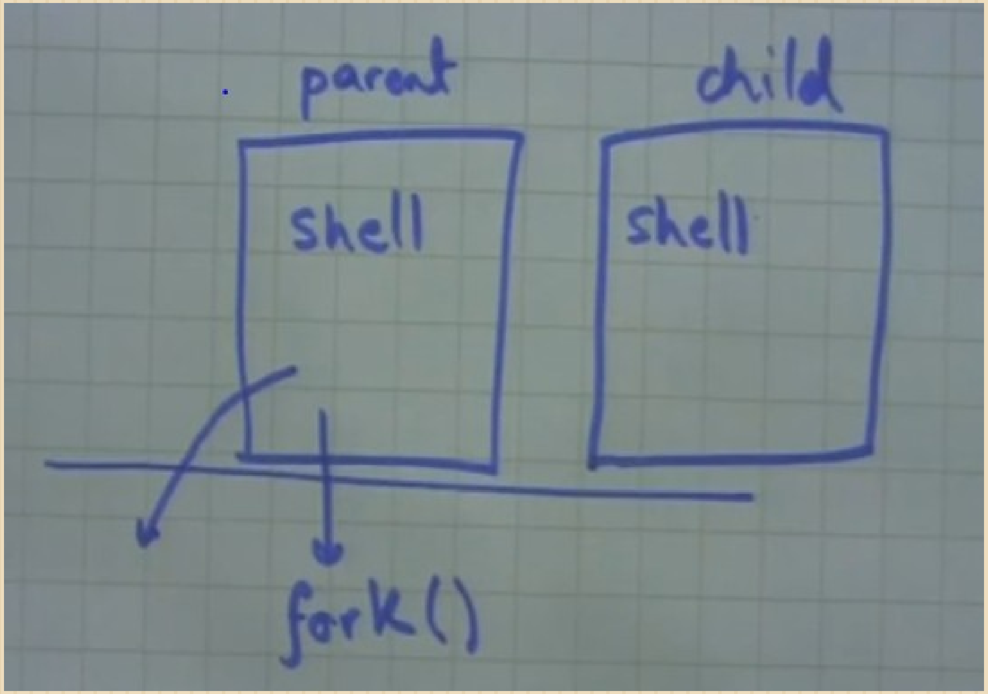
- fork()
- exec()
- exit()

Process related system calls

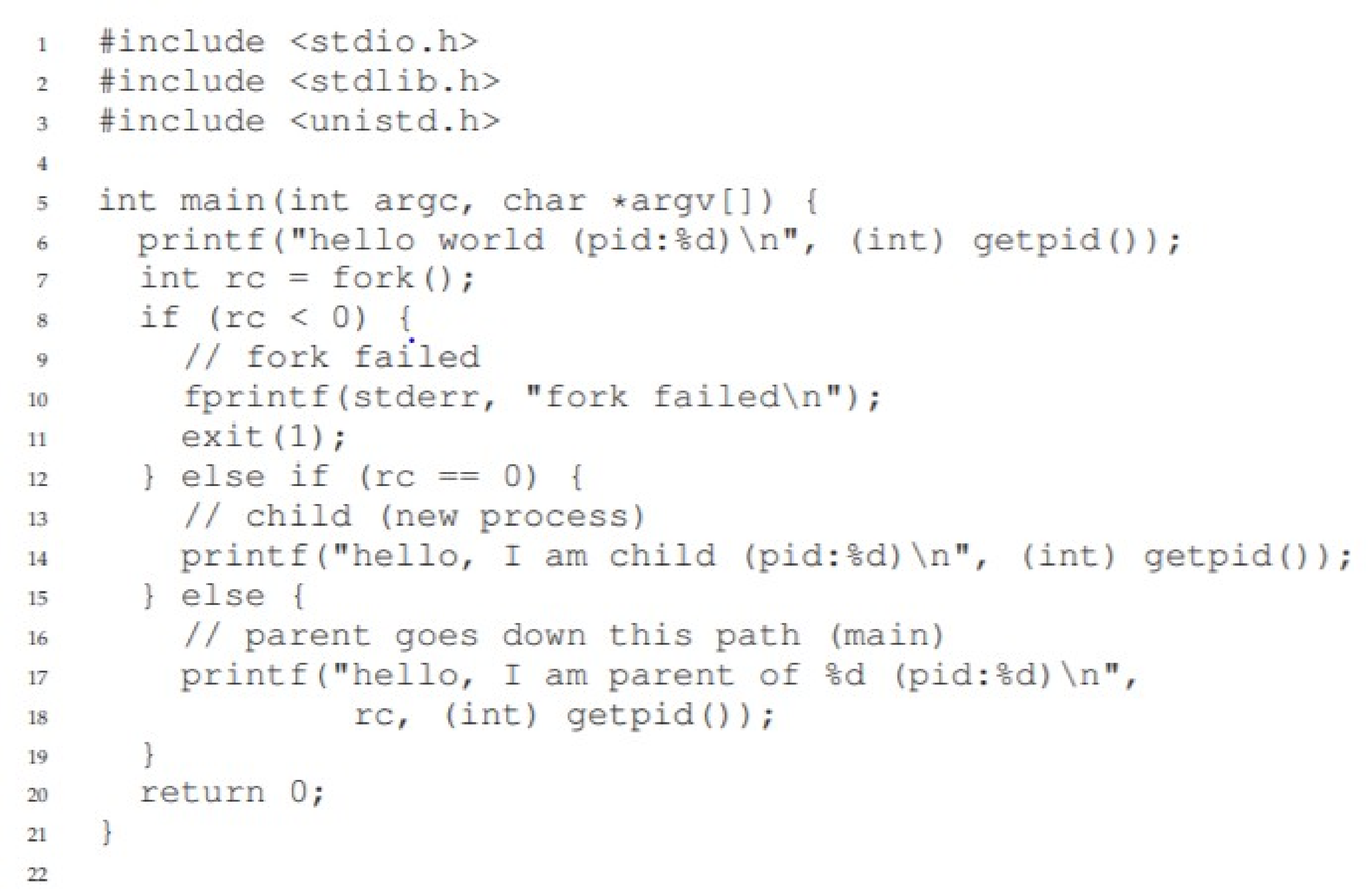
- ***exec("filename")***
  - Makes a process execute a given executable
  - e.g. SHELL process makes a system call
    - exec("browser")
  - "SHELL" program will be replaced by "browser" program in memory

Process related system calls

- fork() creates a new child process
  - e.g. SHELL process makes a fork() system call



Process related system calls

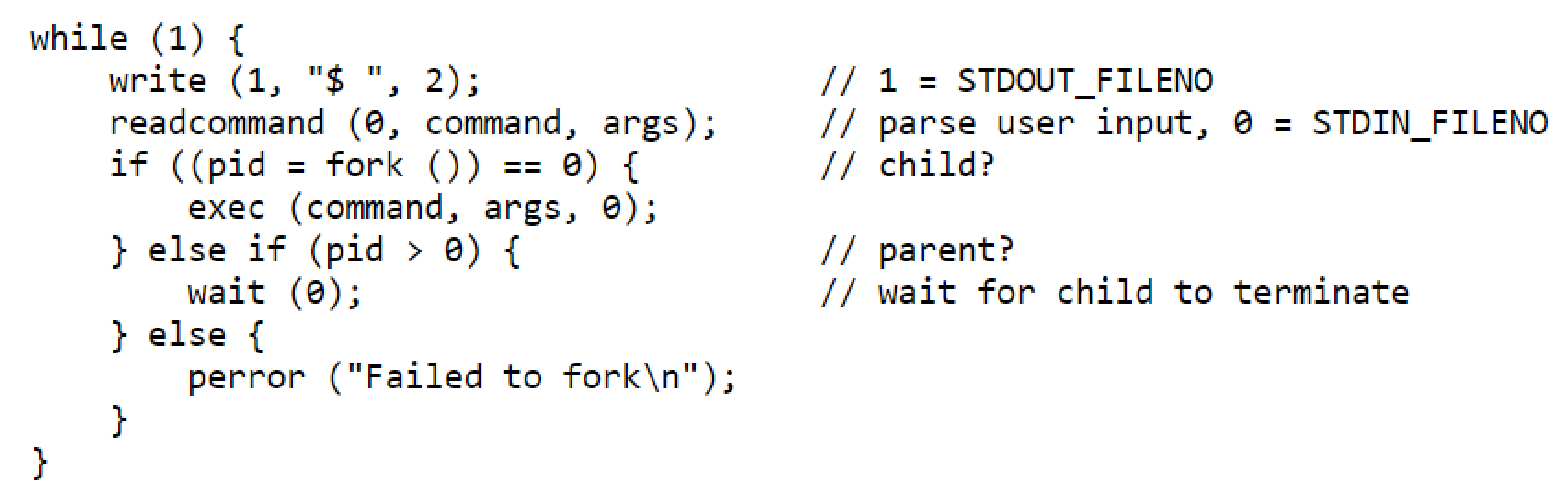


```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
  } else {
    // parent goes down this path (main)
    printf("hello, I am parent of %d (pid:%d)\n",
            rc, (int) getpid());
  }
  return 0;
}

```

Process related system calls



```
while (1) {
    write (1, "$ ", 2);                 // 1 = STDOUT_FILENO
    readcommand (0, command, args);     // parse user input, 0 = STDIN_FILENO
    if ((pid = fork ()) == 0) {         // child?
        exec (command, args, 0);
    } else if (pid > 0) {               // parent?
        wait (0);                       // wait for child to terminate
    } else {
        perror ("Failed to fork\n");
    }
}
```

OS APIs

So, should we rewrite programs for each OS?

- POSIX API: a standard set of system calls that an OS must implement
- Programs written to the POSIX API can run on any POSIX compliant OS
- Program language libraries hide the details of invoking system calls
- The printf function in the C library calls the write system call to write to screen
- User programs usually do not need to worry about invoking system calls