20 Socks

Ten red socks and ten blue socks are all mixed up in a dresser drawer. The 20 socks are exactly alike except for their color. The room is in pitch darkness and you want two matching socks.

What is the smallest number of socks you must take out of the drawer in order to be certain that you have a pair that match?
Operating system interfaces

- Share hardware across multiple processes
  - Illusion of private CPU, private memory
- Abstract hardware
  - Hide details of specific hardware devices
- Provide services
  - Serve as a library for applications
- Security
  - Isolation of processes, users, namespaces
  - Controlled ways to communicate (in a secure manner)
Typical UNIX OS

Kernel

System Call Interface

Scheduler

File System

Network Stack

Virtual Memory

User

Application (Shell)

System Libraries

Application (Apache Web Server)

System Libraries
System calls

- Provide user to kernel communication
  - Effectively an invocation of a kernel function

- System calls are the interface of the OS
System call

Process (e.g., Apache, shell)

User stack of a process (can grow up to 2GBs)

Code, data, heap

Interrupt Vector #80

int 0x80

Kernel code

vector80

EBP →

Argument 1
Argument 2
Calling EIP ++
Old EBP
Local variables
Saved local values, e.g. push EAX, etc.

Last stack frame

IDT

... CS : HANDLER_ADDR ...

...
User address space

Process (e.g., Apache, shell)

User stack of a process (can grow up to 2GBs)

Code, data, heap

int 0x80

Interrupt Vector #80

Kernel code

vector80

Kernel Address Space

Process Address Space

Last stack frame

Argument 1
Argument 2
Calling EIP ++
Old EBP
Local variables
Saved local values, e.g. push EAX, etc
Kernel address space

Process (e.g., Apache, shell)

User stack of a process (can grow up to 2GBs)

Code, data, heap

int 0x80

Interrupt Vector #80

Kernel Address Space

Argument 1
Argument 2
Calling EIP ++
Old EBP
Local variables
Saved local values, e.g. push EAX, etc

Last stack frame

EBP

IDT

Kernel code

vector80

Process Address Space

CS : HANDLER ADDR

...
Kernel and user address spaces

![Diagram showing kernel and user address spaces](image)

- **Process Address Space**
  - User stack of a process (can grow up to 2GBs)
  - Code, data, heap
  - **Interrupt Vector #80**

- **Kernel Address Space**
  - Kernel code
  - **vector80**

- **EBP**

- **IDT**
  - CS : Handler Addr
  - ...
  - ...
System calls, interface for...

- Processes
  - Creating, exiting, waiting, terminating
- Memory
  - Allocation
- Files and folders
  - Opening, reading, writing, closing
- Inter-process communication
  - Pipe
UNIX (xv6) system calls are designed around the shell
Ken Thompson (sitting) and Dennis Ritchie working together at a PDP-11
DEC LA36 DECwriter II Terminal
DEC VT100 terminal, 1980
Shell

- Normal process
- Interacts with the kernel through system calls
  - Creates new processes
fork() -- create new process

int pid;

pid = fork();
if(pid > 0){
    printf("parent: child=%d\n", pid);
    pid = wait();
    printf("child %d is done\n", pid);
} else if(pid == 0){
    printf("child: exiting\n");
    exit();
} else {
    printf("fork error\n");
}
More process management

- `exit()` -- terminate current process
- `wait()` -- wait for the child to exit
- `exec()` -- replace memory of a current process with a memory image (of a program) loaded from a file

```c
char *argv[3];
argv[0] = "echo";
argv[1] = "hello";
argv[2] = 0;
exec("/bin/echo", argv);
printf("exec error\n");
```
File descriptors

```
read(3, buf, size);
```
File descriptors: two processes

Process (e.g., Apache, shell)

read(3, buf, size);

Process (e.g., Apache, shell)

read(5, buf, size);
Two file descriptors pointing to a pipe

Process (e.g., Apache, shell)

```
read(3, buf, size);
```

Process (e.g., Apache, shell)

```
read(5, buf, size);
```
File descriptors

- An index into a table, i.e., just an integer
- The table maintains pointers to “file” objects
  - Abstracts files, devices, pipes
  - In UNIX everything is a pipe – all objects provide file interface
- Process may obtain file descriptors through
  - Opening a file, directory, device
  - By creating a pipe
  - Duplicating an existing descriptor
Standard file descriptors

• Just a convention
  • 0 – standard input
  • 1 – standard output
  • 2 – standard error

• This convention is used by the shell to implement I/O redirection and pipes
File I/O

- `read(fd, buf, n)` – read n bytes from `fd` into `buf`
- `write(fd, buf, n)` – write n bytes from `buf` into `fd`
char buf[512]; int n;
for(;;) {
    n = read(0, buf, sizeof buf);
    if(n == 0)
        break;
    if(n < 0) {
        fprintf(2, "read error\n");
        exit(); }
    if(write(1, buf, n) != n) {
        fprintf(2, "write error\n");
        exit();
    }
}
File I/O redirection

- `close(fd)` – closes file descriptor
  - The next opened file descriptor will have the lowest number
- `fork` replaces process memory, but
  - leaves its file table (table of the file descriptors untouched)
Example: cat < input.txt

```c
char *argv[2];
argv[0] = "cat";
argv[1] = 0;
if(fork() == 0) {
    close(0);
    open("input.txt", O_RDONLY);
    exec("cat", argv);
}
```
pipe - interprocess communication

- Pipe is a kernel buffer exposed as a pair of file descriptors
  - One for reading, one for writing
- Pipes allow processes to communicate
  - Send messages to each other
int p[2];
char *argv[2]; argv[0] = "wc"; argv[1] = 0;
pipe(p);
if(fork() == 0) {
    close(0);
    dup(p[0]);
    close(p[0]);
    close(p[1]);
    exec("/bin/wc", argv);
} else {
    write(p[1], "hello world\n", 12);
    close(p[0]);
    close(p[1]);
}

wc on the read end of the pipe
Pipes

- Shell composes simple utilities into more complex actions with pipes, e.g.
  
  ```bash
  grep FORK sh.c | wc -l
  ```

- Create a pipe and connect ends
Xv6 demo
Files

- Files
  - Uninterpreted arrays of bytes
- Directories
  - Named references to other files and directories
Creating files

- `mkdir()` – creates a directory
- `open(O_CREATE)` – creates a file
- `mknod()` – creates an empty files marked as device
  - Major and minor numbers uniquely identify the device in the kernel
- `fstat()` – retrieve information about a file
  - Named references to other files and directories
Fstat

- `fstat()` – retrieve information about a file

```c
#define T_DIR 1 // Directory
#define T_FILE 2 // File
#define T_DEV 3 // Device

struct stat {
    short type; // Type of file
    int dev; // File system’s disk device
    uint ino; // Inode number
    short nlink; // Number of links to file
    uint size; // Size of file in bytes
};
```
Links, inodes

- Same file can have multiple names – links
  - But unique inode number
- `link()` – create a link
- `unlink()` – delete file
- Example, create a temporary file

```c
  fd = open("/tmp/xyz", O_CREATE|O_RDWR);
  unlink("/tmp/xyz");
```
Xv6 system calls

fork() Create a process
exit() Terminate the current process
wait() Wait for a child process to exit
kill(pid) Terminate process pid
getpid() Return the current process’s pid
sleep(n) Sleep for n clock ticks
exec(filename, *argv) Load a file and execute it
sbrk(n) Grow process’s memory by n bytes
open(filename, flags) Open a file; the flags indicate read/write
read(fd, buf, n) Read n bytes from an open file into buf
write(fd, buf, n) Write n bytes to an open file
close(fd) Release open file fd
dup(fd) Duplicate fd
pipe(p) Create a pipe and return fd’s in p
chdir(dirname) Change the current directory
mkdir(dirname) Create a new directory
mknod(name, major, minor) Create a device file
fstat(fd) Return info about an open file
link(f1, f2) Create another name (f2) for the file f1
unlink(filename) Remove a file
Xv6 demo
In many ways xv6 is an OS you run today