Background: Operating Systems

Brad Karp
UCL Computer Science

CS GZ03 / M030
5th October, 2009

Outline

- Goals of an operating system
- Sketch of UNIX
  - User processes, kernel
  - Process-kernel communication
  - Waiting for I/O
- Simple web server design

Why Discuss OS Now?

- Real distributed systems run on an OS
- OS details affect design, robustness, performance
  - Sometimes because of OS idiosyncrasies
  - More often because OS already solves some hard problems
- Ask questions if something isn't clear!
- Further reading:
  - General overview:

    Tanenbaum, *Modern Operating Systems,* 2nd Edition
  - Details of a modern UNIX:

    McKusick et al., *The Design and Implementation of the 4.4 BSD Operating System*

Goals of OS Designers

- Share hardware resources
  - e.g., one CPU, many applications running
- Protection (app-to-app, app-to-OS)
  - Bug in one app shouldn’t crash whole box or bring down other app
- Communication (app-to-app, app-to-OS)
- Hardware independence
  - Don’t want to rewrite apps for each new CPU, each new I/O device
- How? Using abstractions and well-defined interfaces

UNIX Abstractions

- Process
  - Address space
  - Thread of control
  - User ID
- Filesystem
- File Descriptor
  - File on disk
  - Pipe between processes
  - Network connection
  - Hardware device

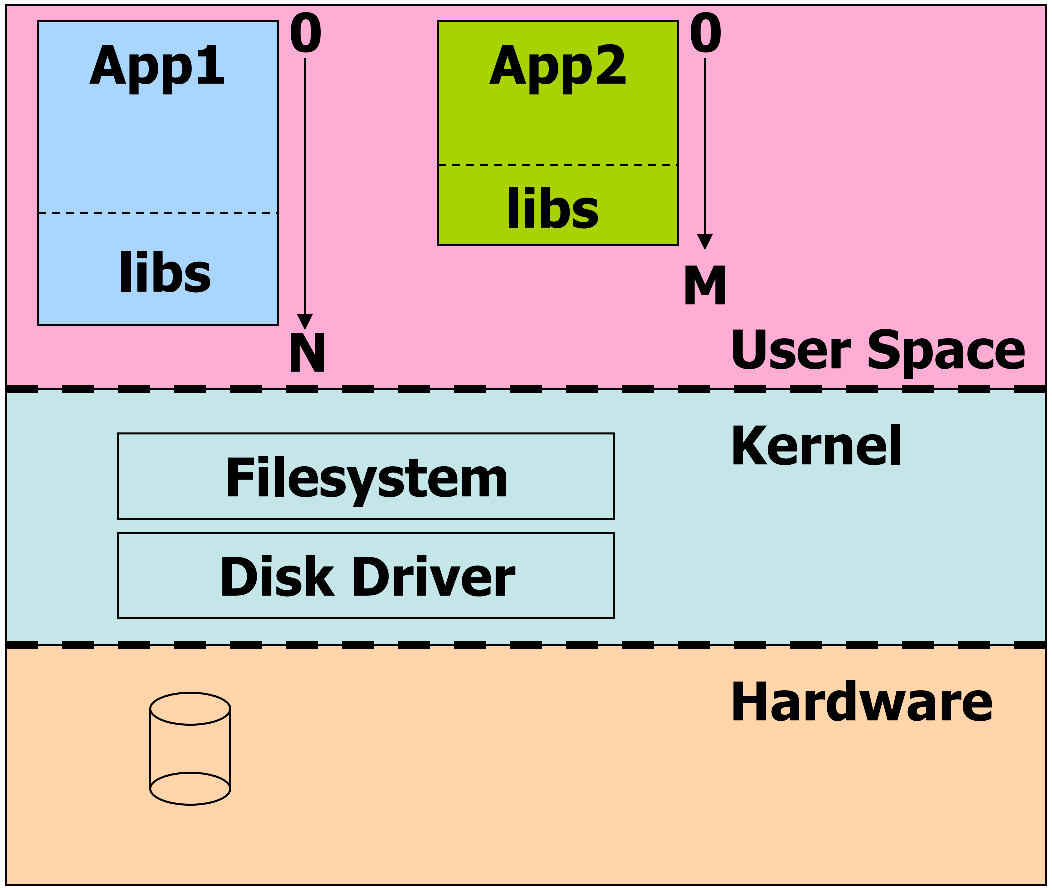
OS Virtualizes Hardware

- Kernel implements abstractions, executes with privilege to directly touch hardware
- OS multiplexes CPU, memory, disk, network among multiple processes (apps)
- Apps can share resources
- Apps can control resources
- Apps see simple interface

OS Abstraction Design

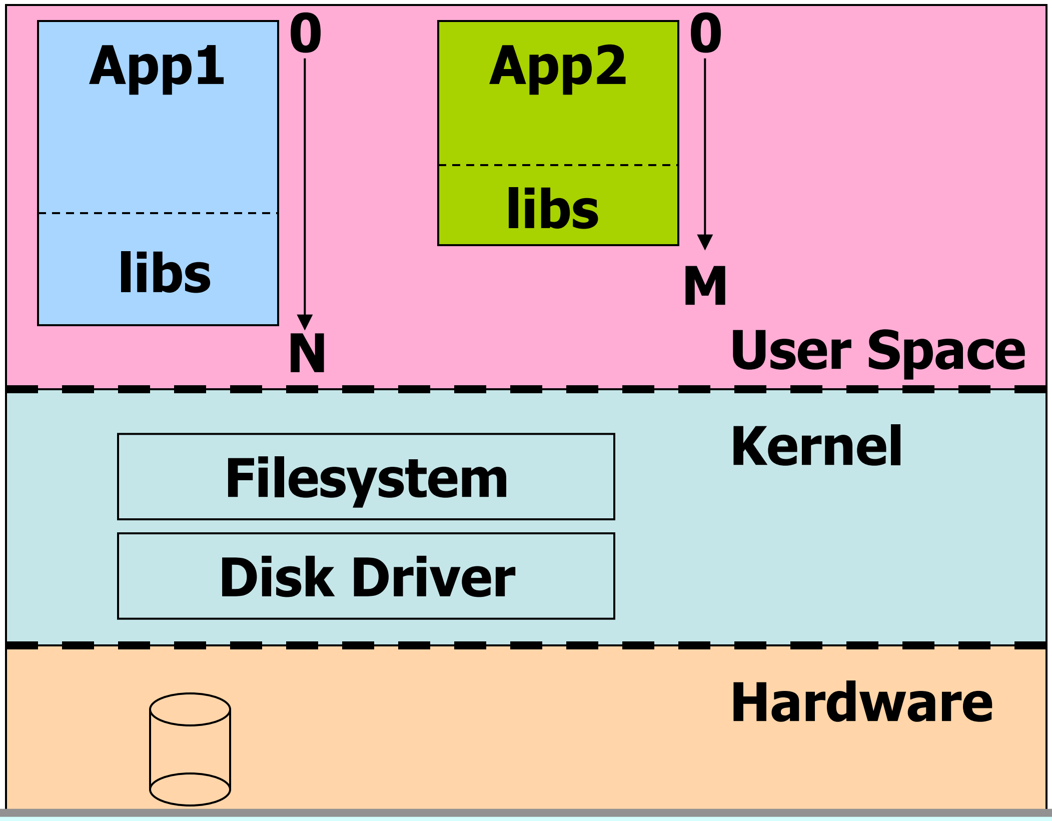
- OS abstractions interact
  - If can start program, must be able to read executable file
- Processes see system call interface to kernel abstractions
  - Looks like function call, but special
  - e.g., fork(), exec()
  - e.g., open(), read(), creat()

Typical UNIX System



- App1 and App2 in separate address spaces; protected from one another
- Hardware runs kernel with elevated privilege

Typical UNIX System



- App1 and App2 in separate address spaces; protected from one another
- Hardware runs kernel with elevated privilege

How do processes and kernel communicate?

How do processes and kernel wait for events (e.g., disk and network I/O)?

System Calls: Process-Kernel Communication

- Application closes a file:

```
close(3);
```

- C library:

```
close(x) {
    R0 <- 73
    R1 <- x
    TRAP
    RET
}
```

System Calls: Traps

- TRAP instruction:
  XP <- PC
  switch to kernel address space
  set privileged flag
  PC <- address of kernel trap handler

- Kernel trap handler:
  save regs to this process' "process control block" (PCB)
  set SP to kernel stack
  call sys_close(), ordinary C function
  …now executing in "kernel half" of process…
  restore registers from PCB
  TRAPRET

System Calls: TRAPRET

- TRAPRET instruction:

  PC <- XP

  clear privileged flag

  switch to process address space

  continue execution

System Call Properties

- Protected transfer
  - Process granted kernel privilege level by hardware
  - But jump must be to known kernel entry point
- Process suspended until system call finishes
- What if system call must wait (e.g., read() from disk)?

Blocking I/O

- On a busy server, system calls often must wait for I/O; e.g.,
- sys_open(path)

```
for each pathname component
    start read of directory from disk
    sleep waiting for disk read
    process directory contents
```

- sleep()

```
save kernel regs to PCB1 (including SP)
find runnable PCB2
restore PCB2 kernel registers (SP, &c.)
return
```

Blocking I/O

- On a busy server, system calls often must wait for I/O; e.g.,
- sys_open(path)

```
for each pathname component
```

Each user process has kernel stack
contains state of pending system call
System call "blocks" while awaiting I/O

- sleep()

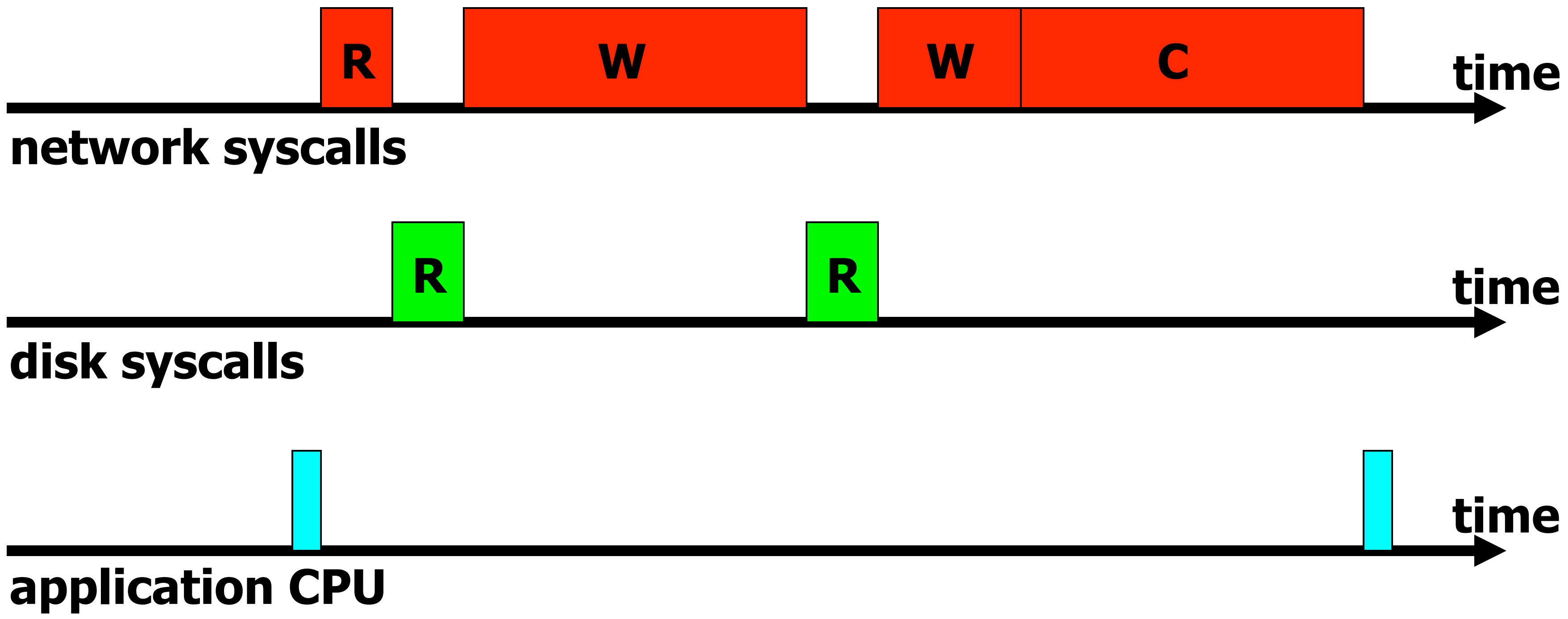
```
save kernel regs to PCB1 (including SP)
find runnable PCB2
restore PCB2 kernel registers (SP, &c.)
return
```

Disk I/O Completion

- How does process continue after disk I/O completes?
- Disk controller generates interrupt
- Device interrupt routine in kernel finds process blocked on that I/O
- Marks process as runnable
- Returns from interrupt
- Process scheduler will reschedule waiting process

How Do Servers Use Syscalls?

- Consider server_1() web server (in handout)

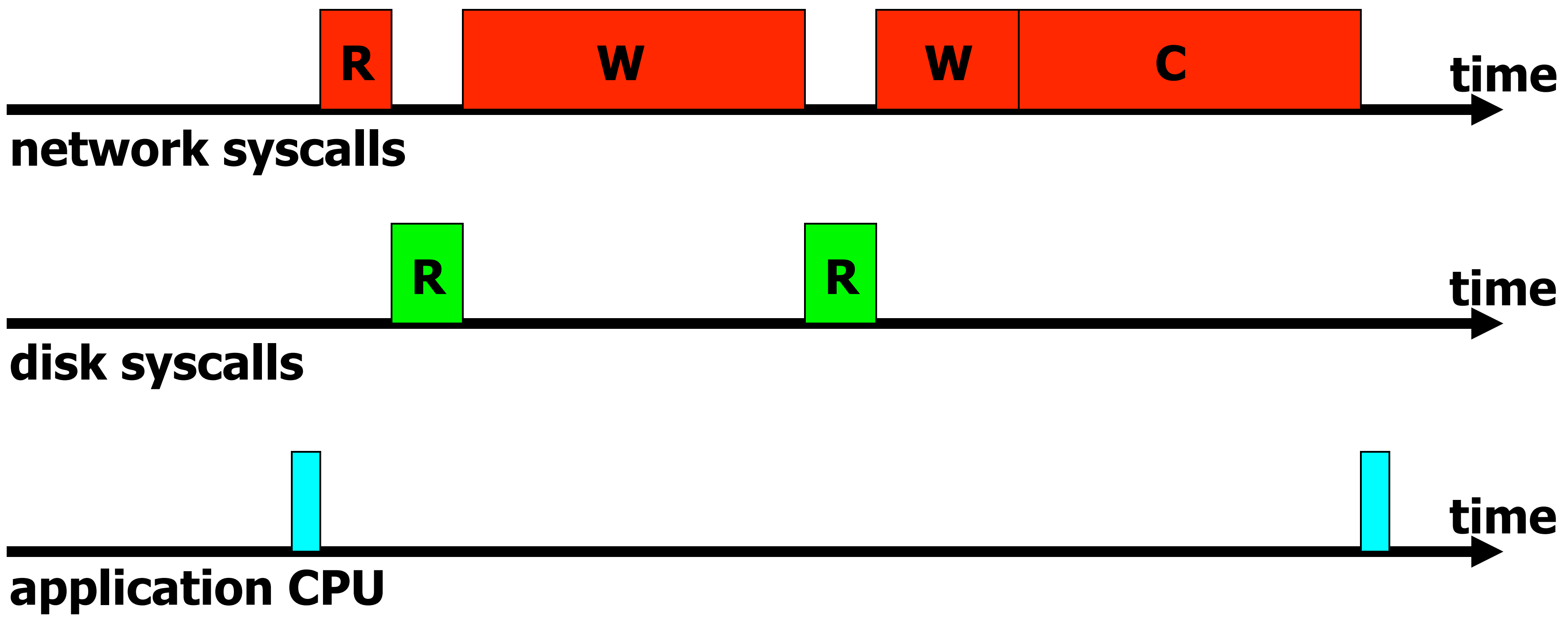


How Do Servers Use Syscalls?

Server waits for each resource in turn

Each resource largely idle

What if there are many clients?



Performance and Concurrency

- Under heavy load, server_1():
  - Leaves resources idle
  - …and has a lot of work to do!
- Why?
  - Software poorly structured!
  - What would a better structure look like?

Solution: I/O Concurrency

- Can we overlap I/O with other useful work? Yes:
  - Web server: if files in disk cache, I/O wait spent mostly blocked on write to network
  - Networked file system client: could compile first part of file while fetching second part
- Performance benefits potentially huge
  - Say one client causes disk I/O, 10 ms
  - If other clients' requests in cache, could serve 100 other clients during that time!

Solution: I/O Concurrency

- Can we overlap I/O with other useful work? Yes:
  - Web server: if files in disk cache, I/O wait spent mostly blocked on write to network

Next: how to achieve I/O concurrency!

- Performance benefits potentially huge
  - Say one client causes disk I/O, 10 ms
  - If other clients' requests in cache, could serve 100 other clients during that time!