W4118 Operating Systems
OS Overview

Junfeng Yang

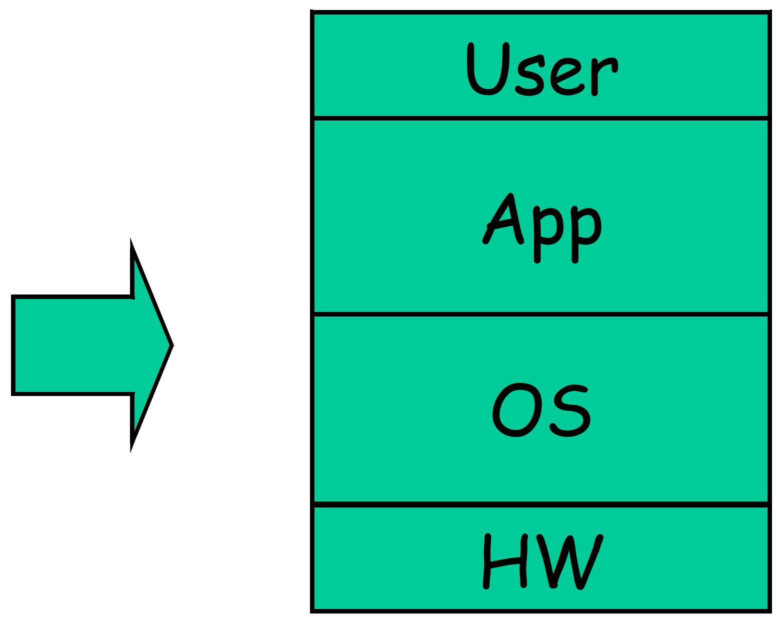
Outline

- OS definitions
- OS abstractions/concepts
- OS structure
- OS evolution

What is OS?

- "A program that acts as an intermediary between a user of a computer and the computer hardware."

"stuff between"



Two popular definitions

- Top-down perspective: hardware abstraction layer, turn hardware into something that applications can use

- Bottom-up perspective: resource manager/coordinator, manage your computer's resources

OS = hardware abstraction layer

- "standard library" "OS as virtual machine"
  - E.g. printf("hello world"), shows up on screen
  - App can make system calls to use OS services
- Why good?
  - Ease of use: higher level of abstraction, easier to program
  - Reusability: provide common functionality for reuse
    - E.g. each app doesn't have to write a graphics driver
  - Portability / Uniformity: stable, consistent interface, different OS/version/hardware look same
    - E.g. scsi/ide/flash disks

Why abstraction hard?

- What are the right abstractions ???
  - Too low level ?
    - Lose advantages of abstraction
  - Too high level?
    - All apps pay overhead, even those don't need
    - Worse, may work against some apps
      - E.g. Database
- Next: example OS abstractions

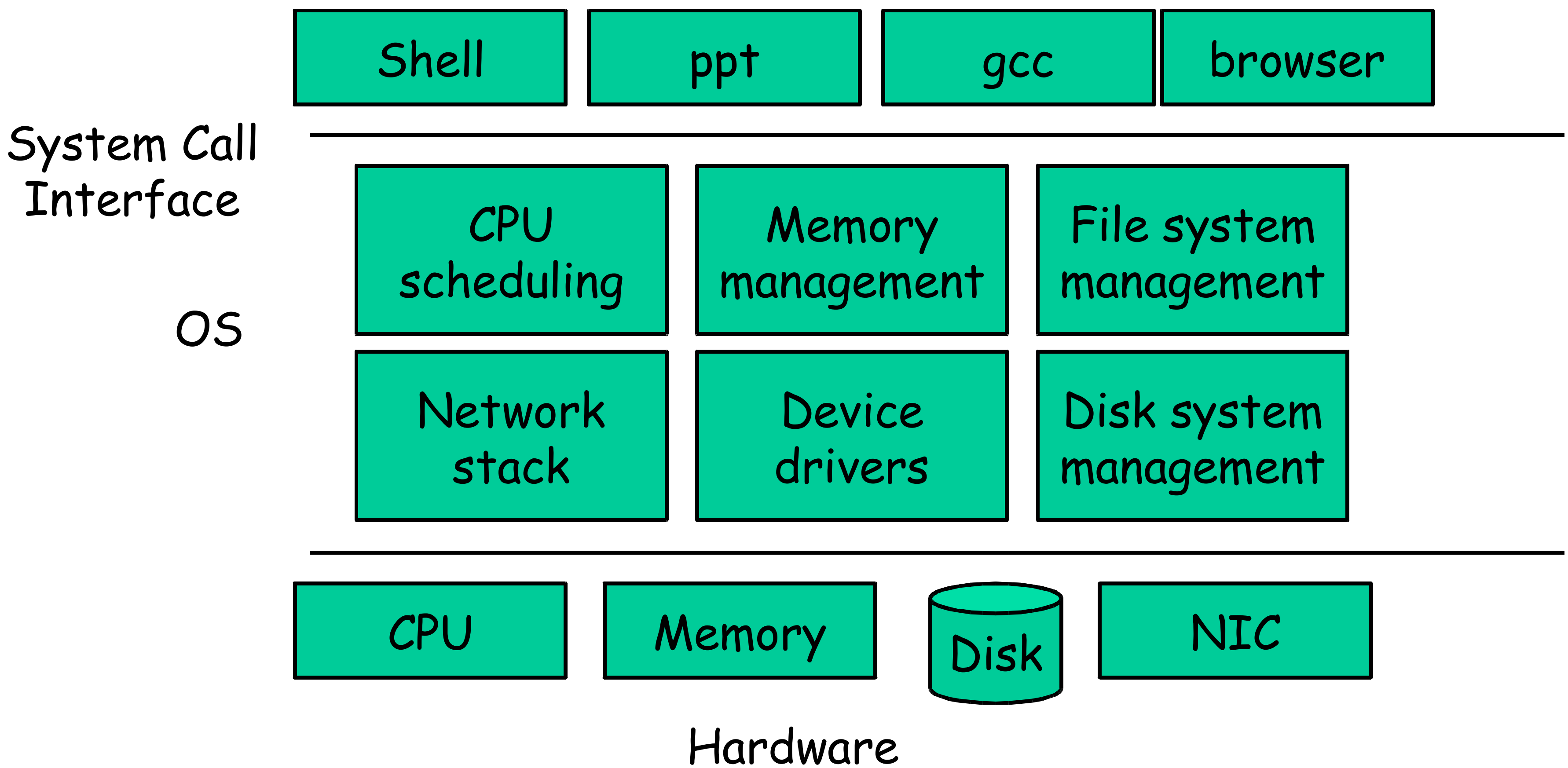
Two popular definitions

- Top-down perspective: hardware abstraction layer, turn hardware into something that applications can use

- Bottom-up perspective: resource manager/coordinator, manage your computer's resources

OS = resource manager/coordinator

- Computer has resources, OS must manage.
  - Resource = CPU, Memory, disk, device, bandwidth, …



OS = resource manager/coordinator (cont.)

- Why good?
  - Sharing/Multiplexing: more than 1 app/user to use resource
  - Protection: protect apps from each other, OS from app
    - Who gets what when
  - Performance: efficient/fair access to resources

- Why hard? Mechanisms v.s. policies
  - Mechanism: how to do things
  - Policy: what will be done
  - Ideal case: general mechanisms, flexible policies
    - Difficult to design right

Outline

- OS definitions
- OS abstractions/concepts
- OS structure
- OS evolution

OS abstraction: process

- Running program, stream of running instructions + process state
  - A key OS abstraction: the applications you use are built of processes
    - Shell, powerpoint, gcc, browser, …

- Easy to use
  - Processes are protected from each other
    - process = address space
  - Hide details of CPU, when&where to run

Process creation system calls

- int fork (void)
  - Create a copy of the invoking process
  - Return process ID of new process in "parent"
  - Return 0 in "child"

- int execv (const char* prog, const char* argv[])
  - Replace current process with a new one
  - prog: program to run
  - argv: arguments to pass to main()

- int wait (int *status)
  - wait for a child to exit

Simple Shell Example

```
// parse user-typed command line into command and args
...

// execute the command
switch(pid = fork ()) {
      case -1: perror ("fork"); break;
      case 0: // child
            execv (command, args, 0);  break;
      default: // parent
            wait (0); break; // wait for child to terminate
}
```

Process communication system calls

- int pipe(int fds[2])
  - Creates a one way communication channel
  - fds[2] is used to return two file descriptors
  - Bytes written to fds[1] will be read from fds[0]
- Often used together with fork() to create a channel between parent and child

OS abstraction: thread

- "miniprocesses," stream of instructions + thread state
  - Convenient abstraction to express concurrency in program execution and exploit parallel hardware

```
for(;;) {
    int fd = accept_client();
    create_thread(process_request, fd);
}
```

  - More efficient communication than processes

OS abstraction: file

- Array of bytes, often persistent across reboot
  - Nice, clean way to read and write data
  - Hide the details of disk devices (hard disk, CDROM, flash …)

Related abstraction: directory

- Collection of file entries

File system calls

- int open(const char *path, int flags, int mode)
  - Opens a file and returns an integer called a file descriptor to use in other file system calls
  - Default file descriptors
    - 0 = stdin, 1 = stdout, 2 = stderr

- int write(int fd, const char* buf, size_t sz)
  - Writes sz bytes of data in buf to fd at current file offset
  - Advance file offset by sz

- int close(int fd)

- int dup2 (int oldfd, int newfd)
  - makes newfd an exact copy of oldfd
  - closes newfd if it was valid
  - two file descriptors will share same offset

Outline

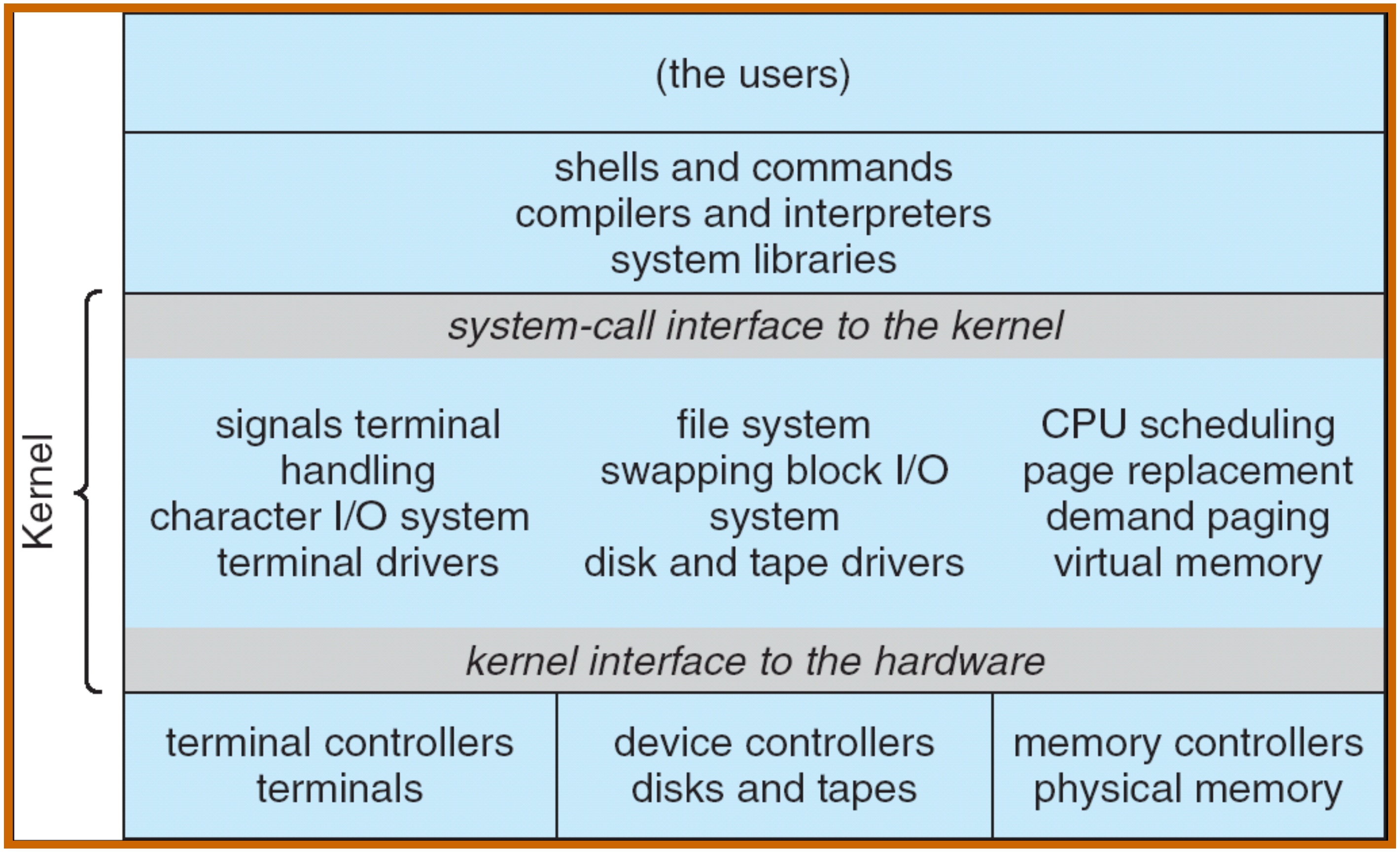
- OS definitions and functionalities
- OS abstractions/concepts
- OS structure
- OS evolution

OS structure

- Can define OS by structure: what goes into the kernel?
  - Kernel: most interesting part of OS
    - Can do everything
    - Manages other parts of OS
- Different structures lead to different
  - Performance, functionality, ease of use, security, reliability, portability, extensibility, cost, …
- Tradeoffs depend on technology and workload

Example OS structure: monolithic

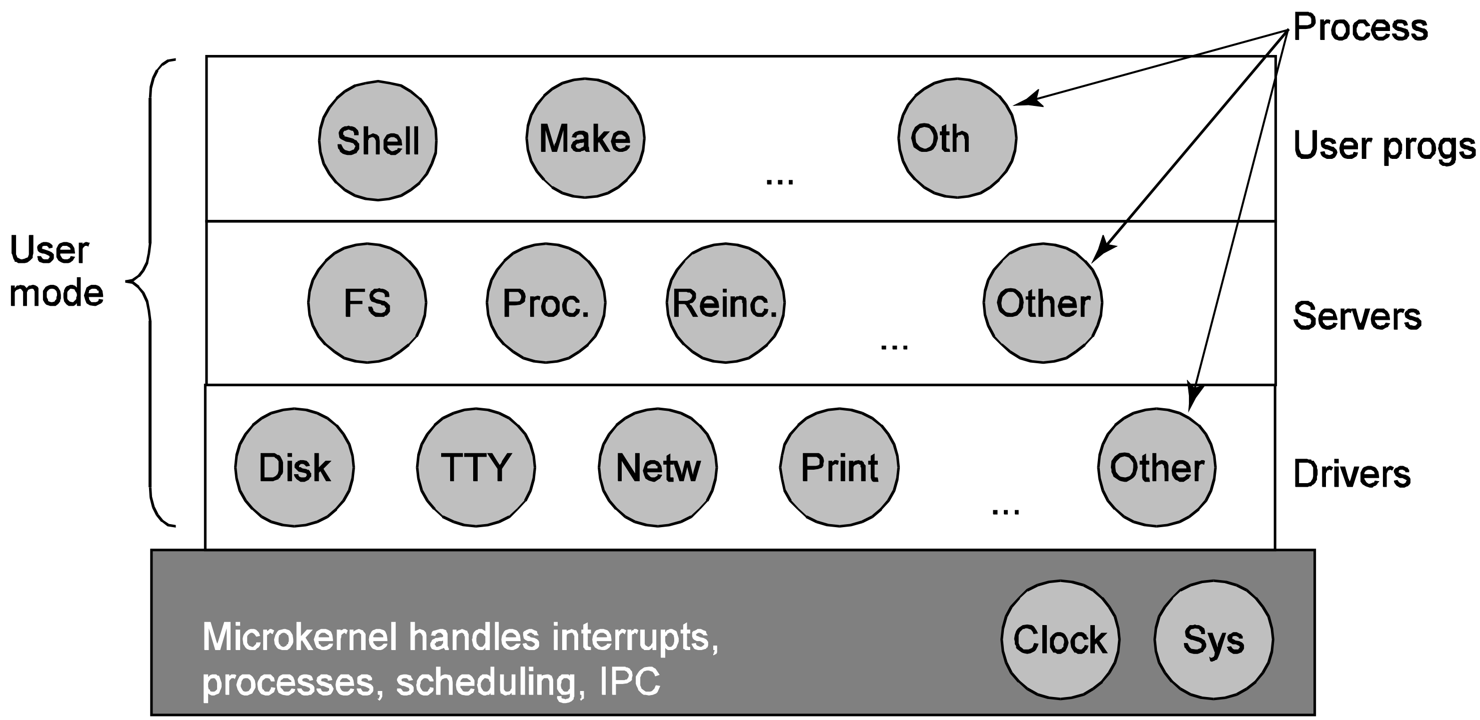
- Most traditional stuff in kernel



Unix System Architecture

Example OS structure: microkernel

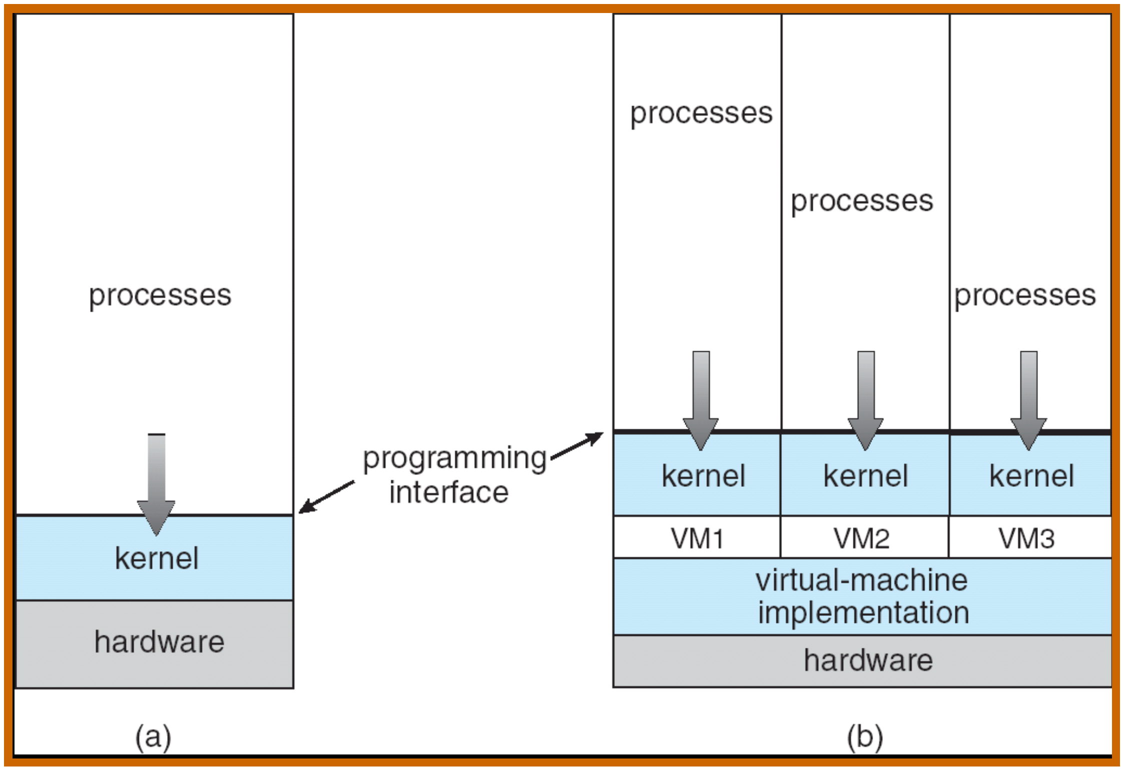
- Try to move stuff out of kernel



Minix 3 System Architecture

Example OS structure: virtual machines

- Exports a fake hardware interface so that multiple OSes can run on top



Non-virtual Machine

Virtual Machine

Outline

- OS definitions and functionalities
- OS abstractions/concepts
- OS structure
- OS evolution

OS evolution

- Many outside factors affect OS
- User needs + technology changes ➔ OS must evolve
  - New/better abstractions to users
  - New/better algorithms to implement abstractions
  - New/better low-level implementations (hw change)
- Current OS: evolution of these things

Major trend in History

- Hardware: cheaper and cheaper
- Computers/user: increases

- Timeline
  - 70s: mainframe, 1 / organization
  - 80s: minicomputer, 1 / group
  - 90s: PC, 1 / user

70s: mainframe

- Hardware:
  - Huge, $$$, slow
  - IO: punch card, line printer
- OS
  - simple library of device drivers (no resource coordination)
  - Human OS: single programmer/operator programs, runs, debugs
  - One job at a time
- Problem: poor performance (utilization / throughput)

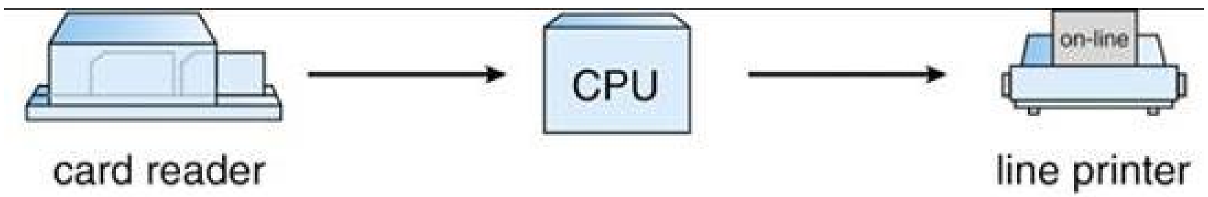
  Machine $$$, but idle most of the time because programmer slow

Batch Processing

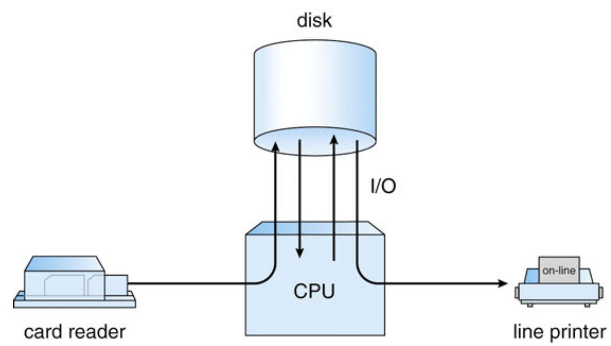
- Batch: submit group of jobs together to machine
  - Operator collects, orders, runs (resource coordinator)
- Why good? can better optimize given more jobs
  - Cover setup overhead
  - Operator quite skilled at using machine
  - Machine busy more (programmers debugging offline)
- Why bad?
  - Must wait for results for long time
- Result: utilization increases, interactivity drops

Spooling

- Problem: slow I/O ties up fast CPU
  - Input ➔ Compute ➔ Output
  - Slow punch card reader and line printer

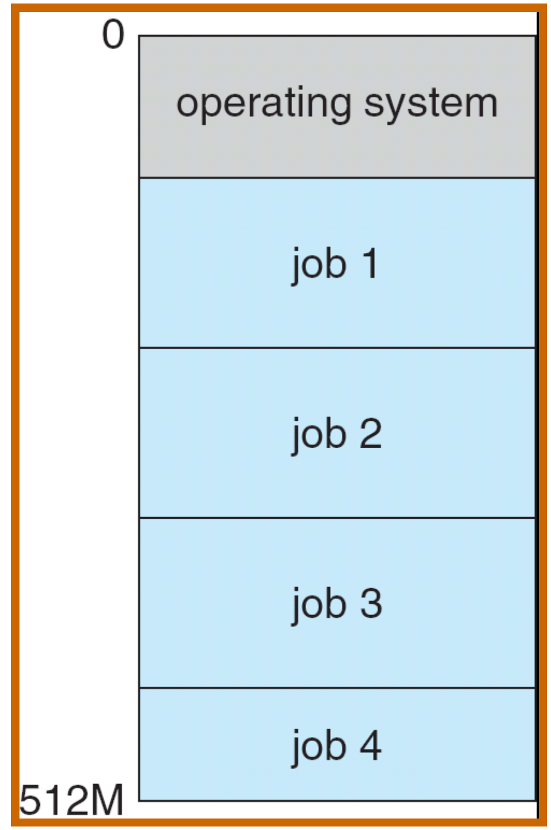


- Idea: overlap one job's IO with other jobs' compute
- OS functionality
  - buffering, DMA, interrupts
- Good: better utilization/throughput
- Bad: still not interactive



Multiprogramming

- Spooling allows multiple jobs
- Multiprogramming
  - keep multiple jobs in memory, OS chooses which to run
  - When job waits for I/O, switch
- OS functionality
  - job scheduling, mechanism/policies
  - Memory management/protection
- Good: better throughput
- Bad: still not interactive



80s: minicomputer

- Hardware gets cheaper. 1 / group
- Need better interactivity, short response time

- Concept: timesharing
  - Fast switch between jobs to give impression of dedicated machine

- OS functionality:
  - More complex scheduling, memory management
  - Concurrency control, synchronization

- Good: immediate feedback to users

90s: PC

- Even cheaper.   1 / user
- Goal: easy of use, more responsive
- Do not need a lot of stuff

- Example: DOS
  - No time-sharing, multiprogramming, protection, VM
  - One job at a time
  - OS is subroutine again

- Users + Hardware → OS functionality

Current trends?

- Large
  - Users want more features
  - More devices
  - Parallel hardware
  - Result: large system, millions of lines of code
- Reliability, Security
  - Few errors in code, can recover from failures
  - At odds with previous trend
- Small: e.g. handheld device
  - New user interface
  - Energy: battery life
  - One job at a time. OS is subroutine again

Next lecture

- PC hardware and x86 programming