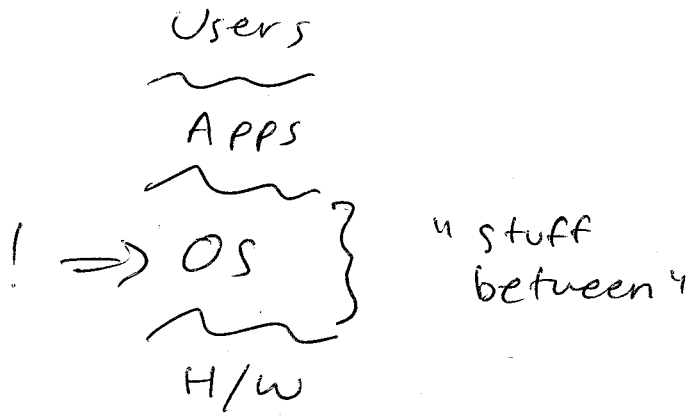
What is an OS?

Some say:



Two common: Standard Lib, Resource mgr

Common #1: Standard Library (OS as "virtual machine")

Converts H/W ⇒ something usable by apps

Why good?

⇒ Makes different devices look same (uniformity)
e.g. SCSI, IDE disk

⇒ Application can reuse common facilities (reusability)

⇒ Higher level abstractions (ease of use)

Why difficult?

⇒ what are the right abstractions?
e.g.
- Too low level:
- Too high level:

Common #2: Resource Mgr (Coordinator)

Bottom-up approach:

Resources make up system, OS must "manage" them

What is a resource?

CPU, memory, disk ("anything valuable")

Why good?

⇒ Allow >1 user to use resource (virtualization / multiplexing)

⇒ Protect apps from each other (protection)

⇒ Protect OS from app

⇒ Provide efficient/fair access to resource (performance)

Why difficult?

⇒ what mechanisms? } How to share effectively?
what policies? }

What functionality in OS?

⇒ No easy answer
(lots of outside factors)

Users/expectations ⇒ OS ⇐ Technology changes

⇒ OS must adapt (over time)

→ Abstractions to users
→ Algs to implement abstractions
→ Low-level implementation (H/W changes)

⇒ Current OS's
⇒ Evolution of these things

⇒ Major trend in history
→ H/W gets Cheaper + cheaper
⇒ (Computers / Person increases)

H/W:

Mainframe	Mini-computer	PC
many people (institution)	group	person

First systems

B/G
⇒ Huge, $, slow

⇒ I/O: punch cards, line printers

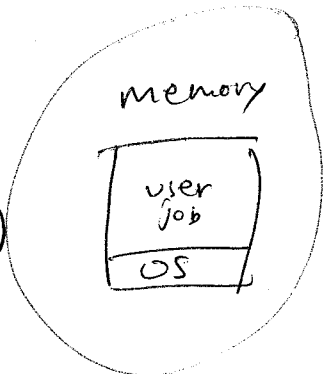
Goal:

Get system to work

Single operator runs/debugs
(reserve time, go in, etc.)

OS

simple library (no resource coordination)



Problem

Machine is $$ ⇒ need to use efficiently
(most of time machine is idle)

Need better performance: utilization
(throughput)

Batch Processing

Batch: Group of jobs submitted to machine together

operator collects, orders, runs 1/time
(resource mgr)

OS: same as before

Why good?

- Lower setup costs
- Operator quite skilled at using machine
- Machine busy more (programmers off thinking)

Why bad?

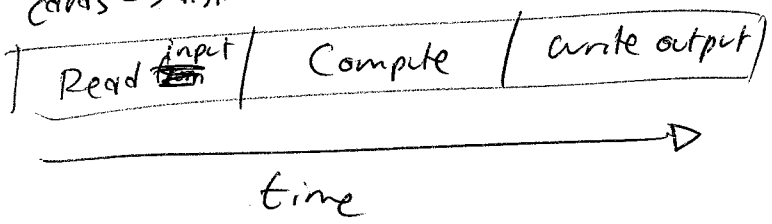
- must wait for results for long time (oops, forgot ; semicolon!)

⇒ Utilization ↑ interactivity ↓

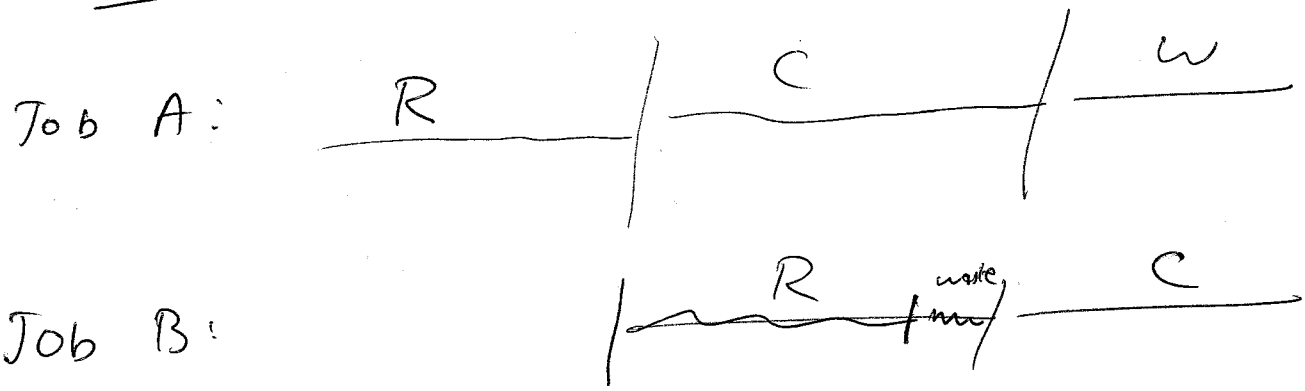
Spooling

Problem: I/O takes time too!

OLD: cards ⇒ disk disk ⇒ printer



Idea: Overlap execution

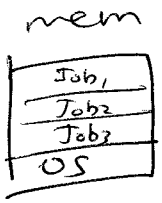


OS: Buffering, DMA, interrupts

Good: Better throughput / utilization

Bad: Still wasting CPU during Compute (disk I/O)

Multiprogramming

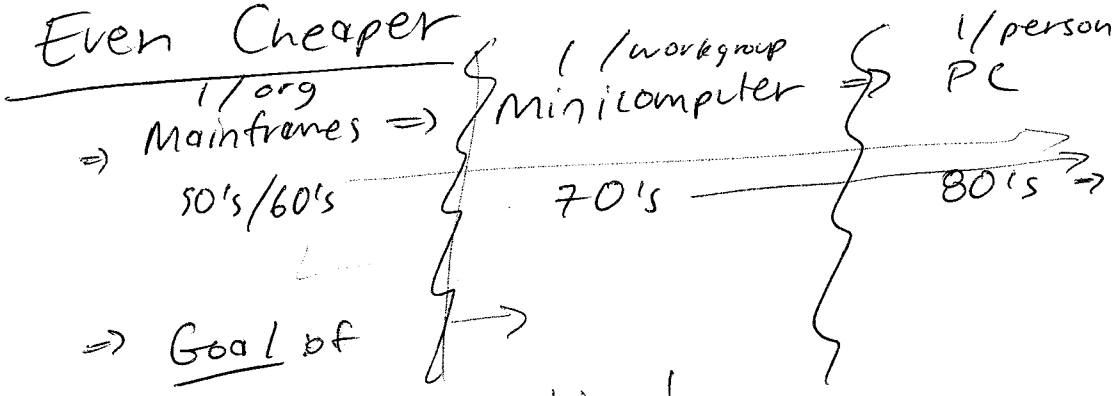


⇒ keep multiple "jobs" in memory
OS chooses which to run
when job waits for I/O, switch

⇒ New OS functions
  ⇒ Job scheduling mech/policies
  ⇒ mem mgmt / protection

⇒ Good: Better throughput
Bad: still not interactive

Even Cheaper



⇒ Goal of

Response time!

Concept: Time-sharing

Switch between jobs to give "appearance" of dedicated machine

Good: users submit jobs, immediate feedback

OS functionality

{ more complex scheduling, mem mgmt
Concurrency control, synchronization }

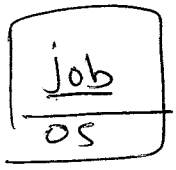
PC

whole computer cheap (1/user!)

Don't need so much functionality!

→ time-sharing, protection, VM,

⇒ OS is subroutine again (DOS)



⇒ H/W + users ⇒ OS functionality

Current Trends

- => Huge systems
- => Tiny systems

Complexity
=> millions of lines of code

Robustness
=> crash-proof(?)

} at odds

Secure
=> not easy to compromise

Why study OS?

Tangible
=> You may work for Solaris group at Sun
=> Administer /use/ program system well

Intangible
=> Curiosity ⇔ How stuff works
=> Broad understanding of CS
H/W, PL (synch), Algorithms, Performance
=> Challenging.
How to design/implement large/complex systems?