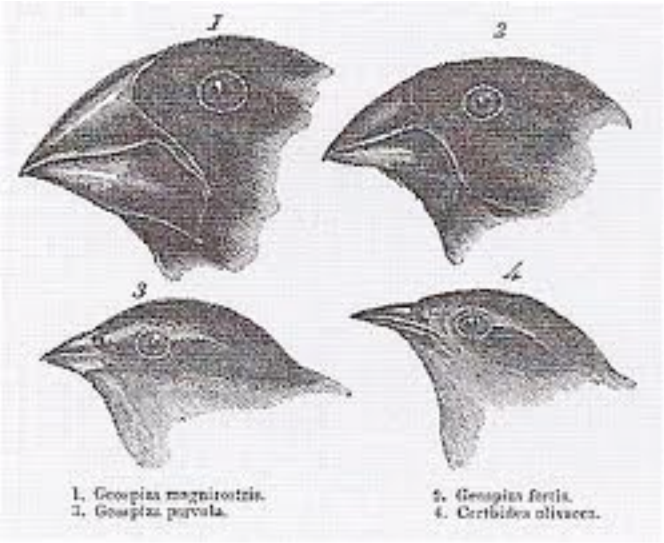
History of Operating Systems

Portions of this material courtesy Jennifer Wong and Gene Stark

Natural Selection

- Almost all OS design is about trade-offs
- What drives these trade-offs?
  - Hardware
  - User Applications
- Observation: These change
  over time



Meta-Example: Caching

- If reading something is slow, caches are a great idea
- If reading something is fast, maintaining caches can slow things down
- Historically, the use of caching is proportional to network latency (relative to other resources)
  - Pendulum swings back and forth over time

Identify fundamentals, predict future, profit!

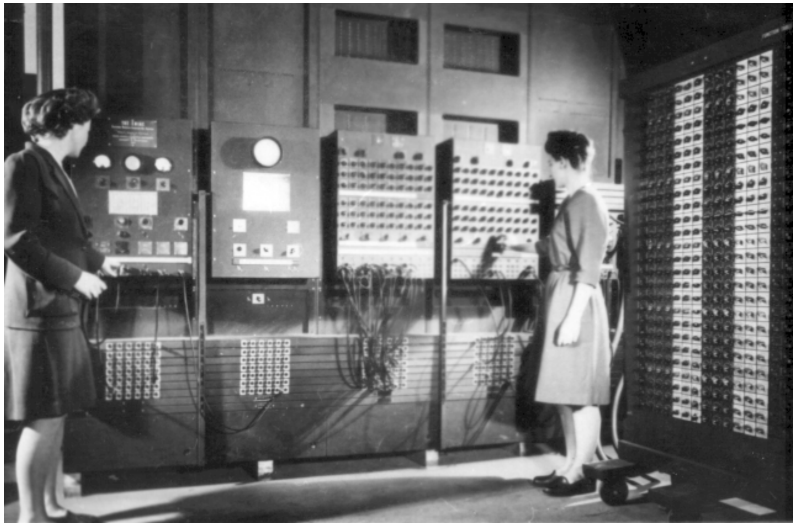
That said…

- Early history really is just figuring out how to make things work sensibly
- And some principles are not trade-offs

Let’s look at history of HW and apps

1940's – First Computers

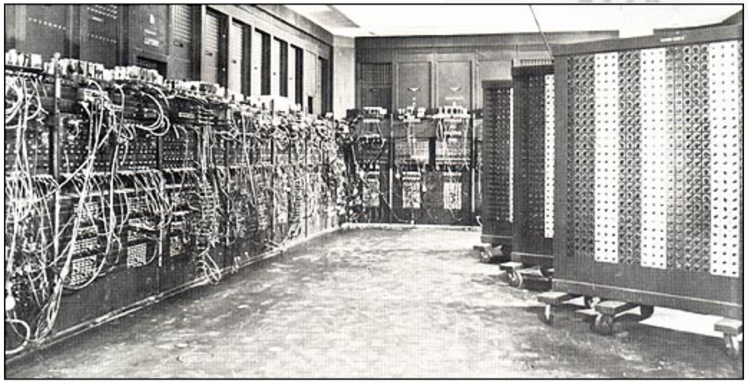
- One user/programmer at a time (serial
  - Program loaded manually using switches
  - Debug using the console lights
- ENIAC
  - 1st gen purpose machine
  - Calculations for Army
  - Each panel had specific function



ENIAC (Electronic Number Integrator and Computer)

1940's – First Computers

- Vacuum Tubes and Plugboards
- Single group of people designed, built, programmed, operated and maintained each machine
- No Programming language, only absolute machine language (101010)
- O/S? What is an O/S?
- All programs basically did numerical calculations



Among the first assignments given to Eniac, first all-electronics digital computer, was a knotty problem in nuclear physics. It produced the answer in two hours. One hundred engineers using conventional methods would have needed a year to solve the problem

Pros:

- Interactive – immediate response on lights
- Programmers were women

Cons:

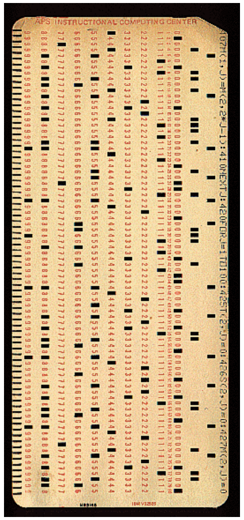
- Lots of Idle time
  - Expensive computation
- Error-prone/tedious
- Each program needs all driver code

What problem do you think was fixed first?

Idle time!

- Computers were ridonculously expensive
- Switching programs meant manually replugging stuff
  - Minutes of downtime if you are lucky
- If I spend $1m/yr for a computer, each minute of downtime costs ~$1.90!
- Any ideas?

1950's Hardware Innovation



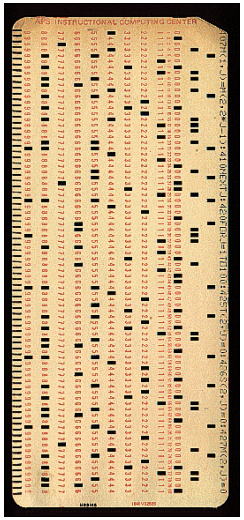
- The punch card
  - Represents plug choices
  - Selected (*programmed*) offline at a desk
    - Write-only memory
  - But can be quickly swapped in/out
- A sequence of punch cards can represent a more sophisticated program

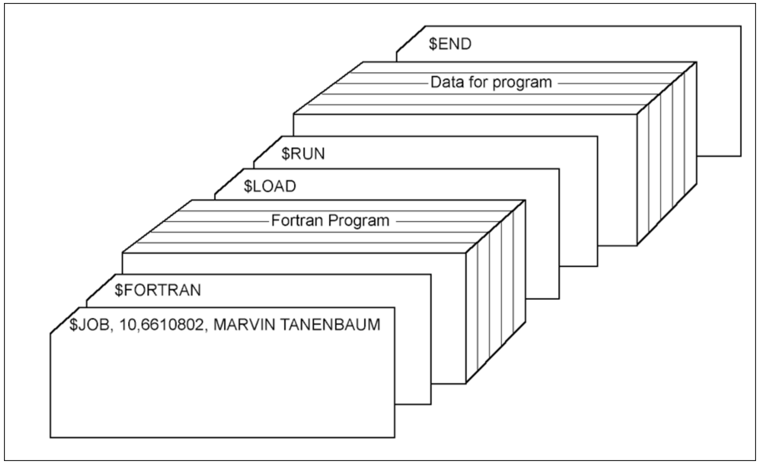
- Your tech-literate (grand?) parents will share punch card stories at Thanksgiving
  - Spoiler: They drop the deck

1950's OSes: Batch Processing

- Programs were decks of cards
- The OS was called a *resident monitor*
- Pseudo code for the OS:

```
while ( next job ) {
    pick job;
    run job to completion;
}
```





From Monitor to OS

- Resident monitor was a basic OS
  - Software
  - Always in memory
  - Controls the sequence of events
  - Reads in job and gives control of CPU to that job
  - Job completion returns to monitor

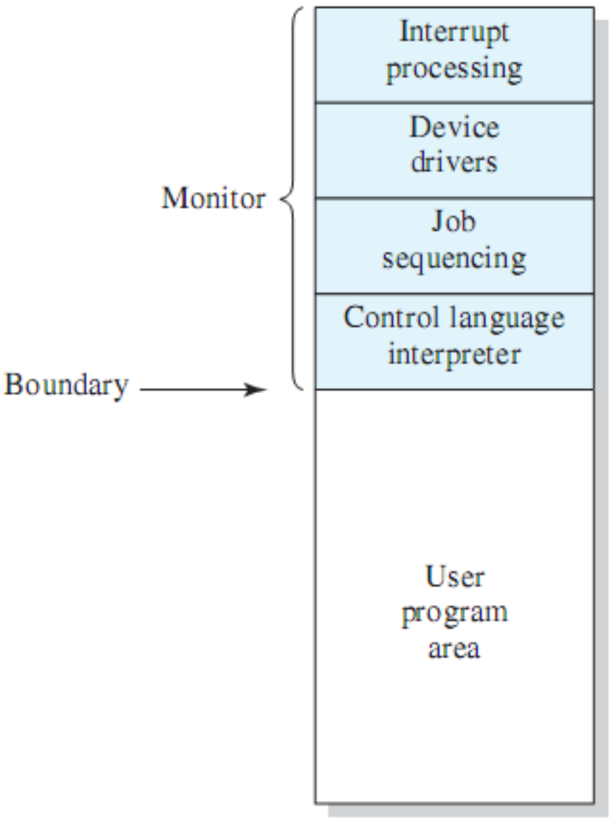


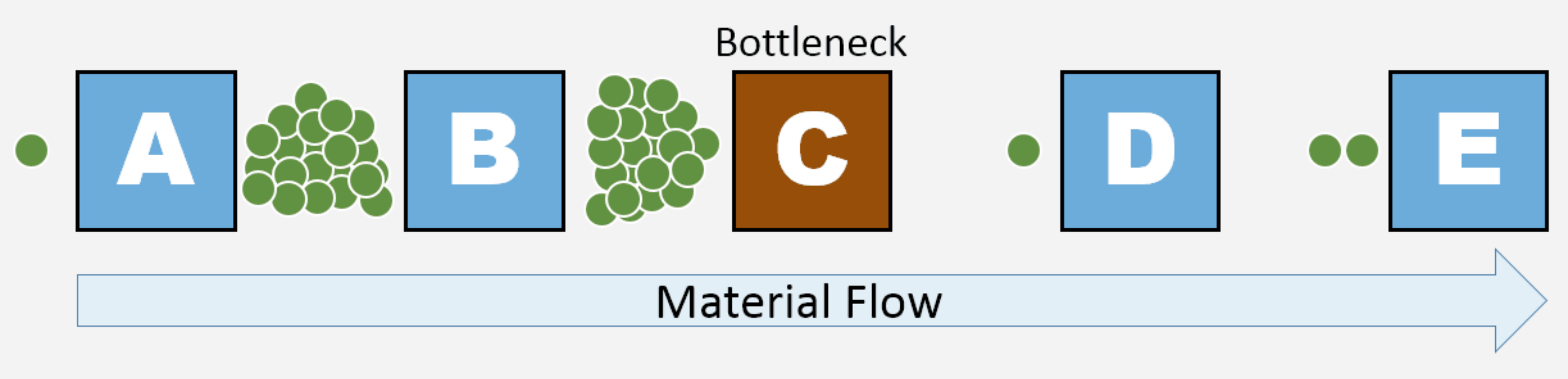
Figure 2.3 Memory Layout for a Resident Monitor

Back to idle time…

- Does batch processing reduce idle time?
  - Yes, by reducing time to switch jobs
- How?
  - Keep as many pending jobs as possible ready
- Key Principle: Keep the CPU busy!
  - Perhaps obvious, but still drives a ton of innovation
  - Albeit filling smaller idle periods (more to come…)

Nomenclature: Bottleneck

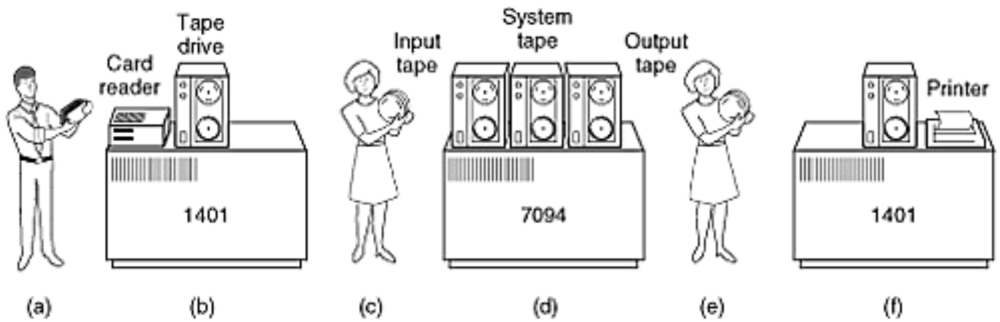
- In a well-conditioned system, everything produces and consumes at same rate



- A *bottleneck* is when one stage is slower
- Batch processing removes a bottleneck on loading a program into the system (online to offline programming)

Image from wikipedia

1950's – Batch Processing



IBM 7090

Pros:

- CPU kept busy, less idle time
- Monitor could provide I/O services

Cons:

- No longer interactive – longer turnaround time
- Debugging more difficult
- Buggy jobs could require operator intervention

So, are we done with idle time yet?

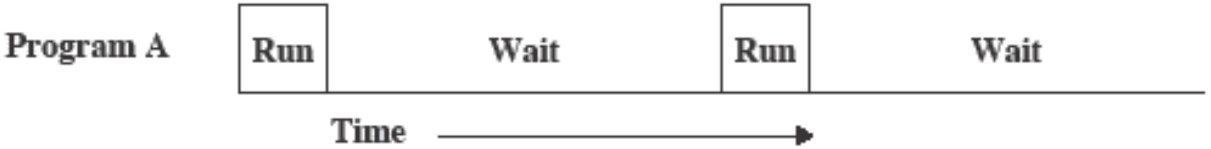
Tacit assumption: All work CPU-bound

- Modern context: obviously false
- Tape and other I/O devices introduced
- I/O is S L O O O O O O O O O O O O O O O O W W W
  - Compared to CPU

- Even on modern computers:
  - CPU: 3 billion cycles per second per core
  - The fastest, most bleeding-edge, flash: any guesses?
    - ~1.2 million I/O operations per second
  - Regular old hard disks:
    - About 100 I/O operations per second on a good day

Uniprogramming

- Processor must wait for I/O instruction to complete before preceding



(a) Uniprogramming

The I/O Problem

- Jobs start having I/O
- I/O takes a long time
  - CPU is idle during I/O

Monitor Pseudo-Code

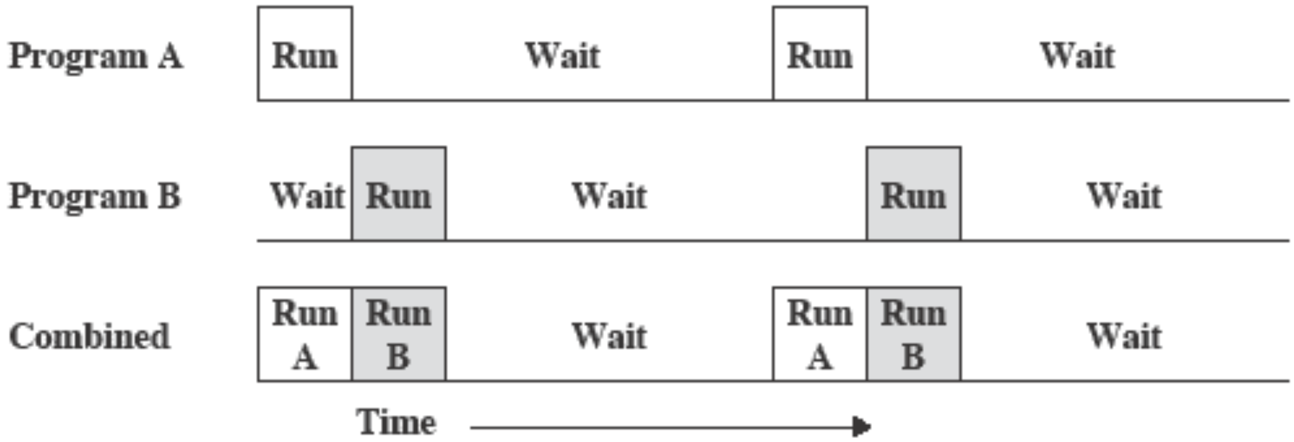
```
while ( next job ) {
      pick job;
      run job to completion;
}
```

Soft Target

Ideas? What is the bottleneck?

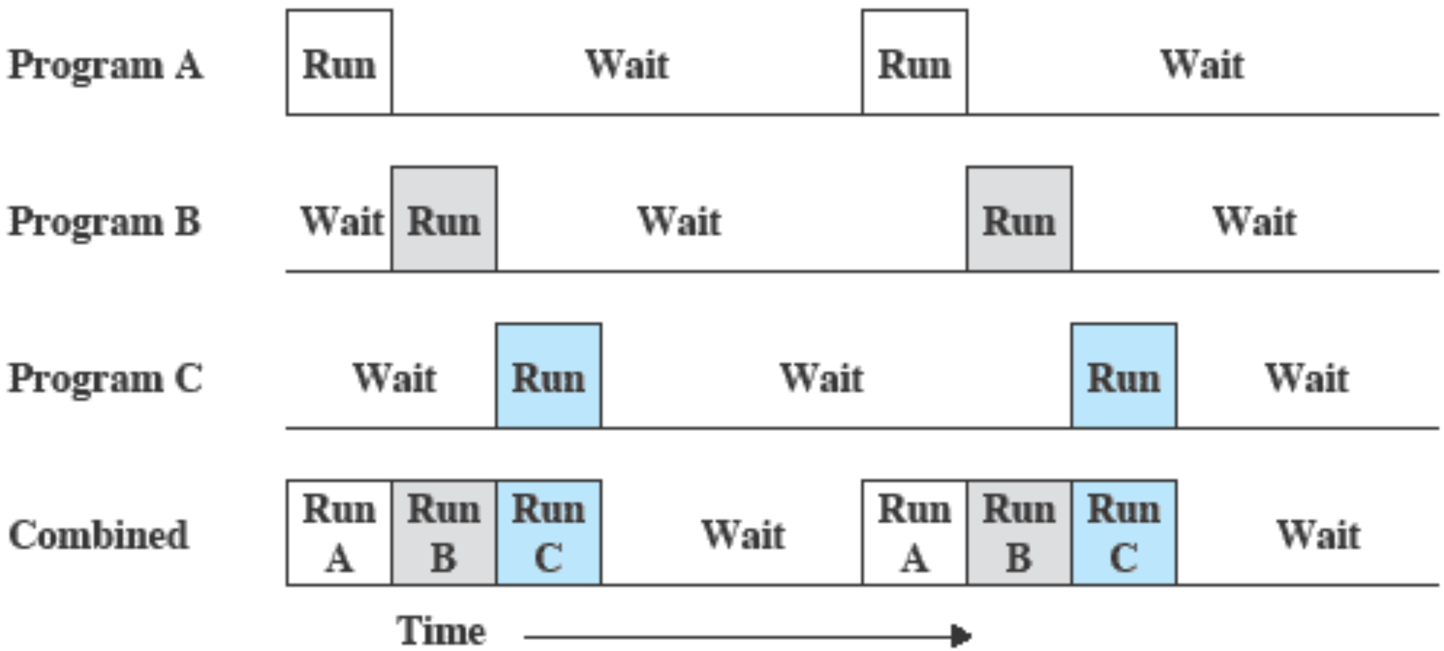
Multiprogramming

- When one job needs to wait for I/O, the processor can switch to another job



(b) Multiprogramming with two programs

Multiprogramming



(c) Multiprogramming with three programs

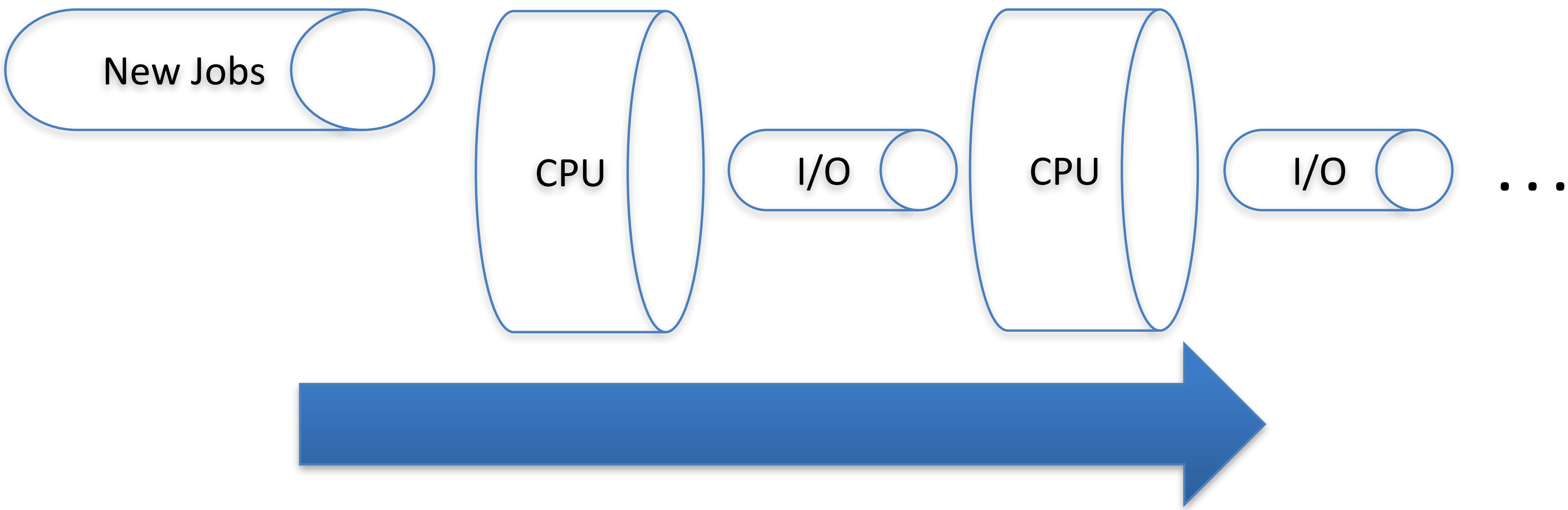
Multiprogramming Pseudo-Code

```
while ( next job ) {
    pick job;
    run job to completion or blocking event (e.g., I/O);
}
```

- Note, monitor and multiple jobs in memory
  - Monitor protects jobs' memory from each other

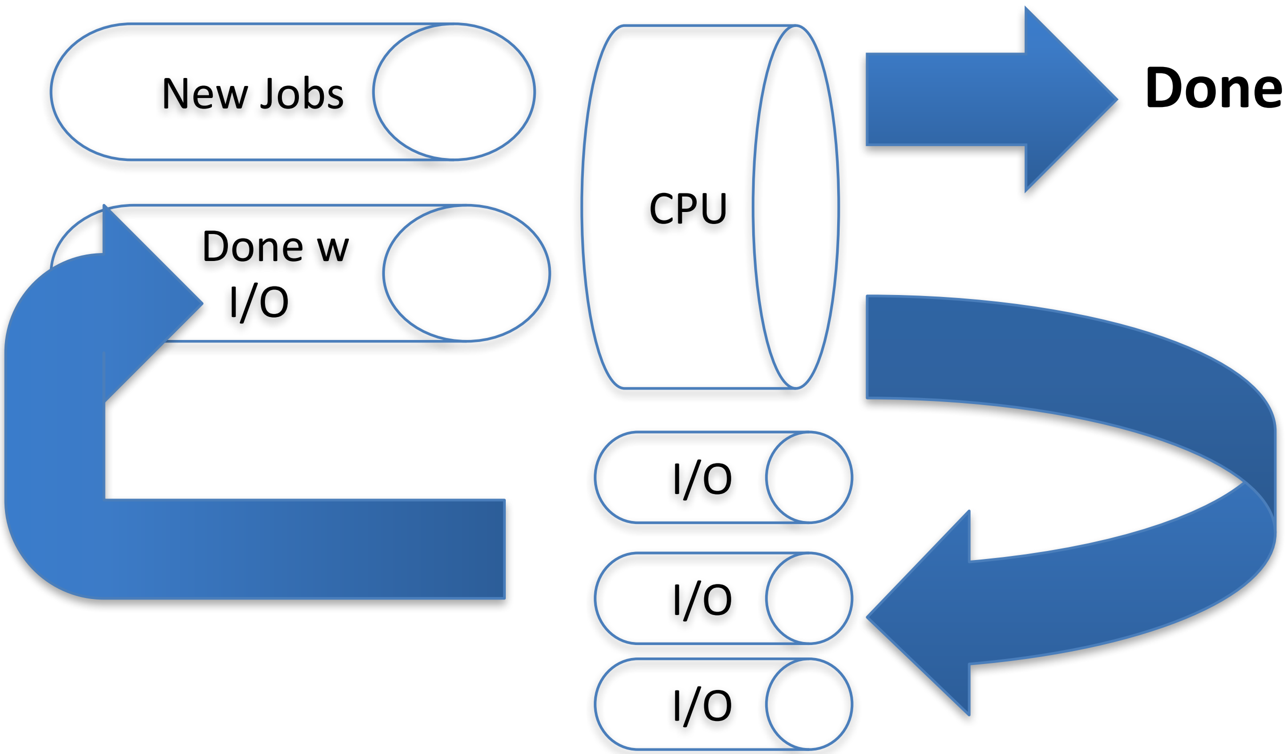
But did we remove the bottleneck?

- Not exactly, I/O is still slow and a bottleneck
- We really just tried to add more sources



But did we remove the bottleneck?

- Not exactly, I/O is still slow and a bottleneck
- We really just tried to add more sources



1960's – Multiprogramming (a.k.a. time-sharing)

IBM System 360

Pros:

- Paging and swapping (RAM)
- Interactive
- Output available at completion
- CPU kept busy, less idle time

Cons:

- HW more complex
- OS complexity

1970's - Minicomputers and Microprocessors

- Trend toward many small personal computers or workstations, rather than a single mainframe.
  - Advancement of Integrated circuits
- Timesharing in Unix
  - Multiple "dumb terminals" (graphics and keyboard)
  - Sharing one machine (CPU, storage, etc)

"User" I/O

- You can model terminal I/O just like any other high-latency device (e.g., disk, network)
- Example:
  - User presses a key
  - OS + program do a little work
  - App blocks for next keystroke
  - OS schedules something else
- Even in 70s, CPUs faster than human typing
  - Thus, one CPU could comfortably accept input from multiple users
  - Computation induced by those commands a different story…

For interactive apps, **you** are the bottleneck

1980's – Personal Computers & Networking

- Microcomputers = PC (size and $)
- MS-DOS, GUI, Apple, Windows

- Networking: Lower cost by sharing resources
  - Not cost-effective for every user to have printer, backed up hard drive, etc.
  - Rise of cheap, local area networks (Ethernet), and access to wide area networks (Arpanet).

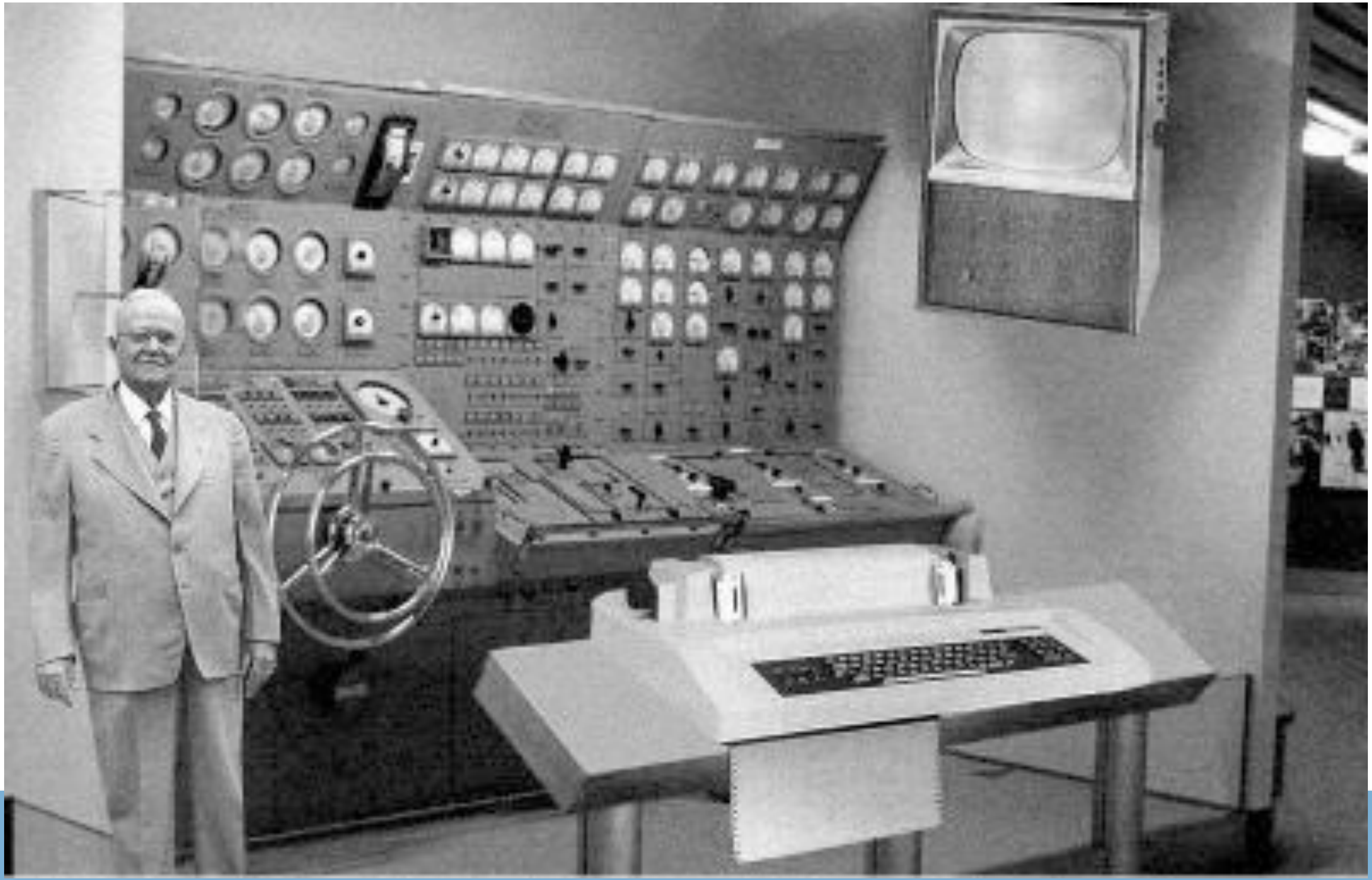
1980's – Personal Computers & Networking

- OS issues:
  - Communication protocols, client/server paradigm
  - Reliability, consistency, availability of distributed data
  - Heterogeneity
  - Reducing Complexity
- Ex: Byte Ordering

1990's – Global Computing

- Dawn of the Internet
  - Global computing system
- Powerful CPUs cheap! Multicore systems
- High speed links
- Standard protocols (HTTP, FTP, HTML, XML, etc)
- OS Issues:
  - Communication costs dominate
    - CPU/RAM/disk speed mismatch
    - Send data to program vs. sending program to data
  - QoS gurantees
  - Security

In the year 2000…

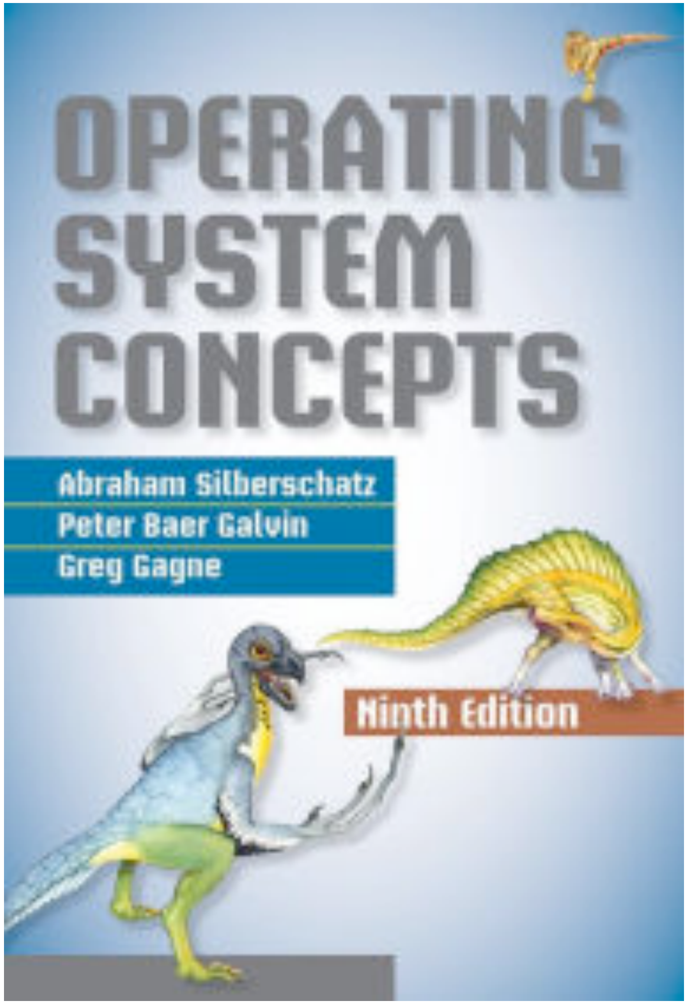


2000's – Embedded and Ubiquitous Computing

- Mobile and wearable computers
- Networked household devices
- Absorption of telephony, entertainment functions into computing systems

- OS issues:
  - Security, privacy
  - Mobility, ad-hoc networks,
  - Power management
  - Reliability, service guarantees

Are we done?


OPERATING SYSTEM CONCEPTS
Abraham Silberschatz
Peter Baer Galvin
Greg Gagne
Ninth Edition

What hardware changes?

- Multi-core
  - We can't make cores faster, but we can give you more of them
  - OS issues: Synchronization is hard (more later)
- Cloud computing
  - Lower costs, on-demand "elastic" resource allocation
  - OS issues: security, job placement,
    - Networking/caching redux
- Embedded Devices: IoT, wearables, etc
  - Dealing with heterogeneity
  - Need new abstractions for devices

Summary

- OSes began with big expensive computers used interactively by one user at a time.
- Batch systems kept computer busier.
- Time-sharing overlaps computation and I/O, keeping the CPU even busier
- Multiprogramming made systems interactive and supported multiple users
- Cheap CPU/memory/storage make communication the dominant cost.
- Multiprogramming still central for handling concurrent interaction with environment.

Meta-Summary

- We know how to build a working OS
- But OS research and development will continue!
  - New and evolving hardware (master #3)
    - Arguably wearables are master #1 too
  - New and evolving apps (master #2)
- A lot of this course will be understanding design trade-offs
  - If you can map new hardware/apps to these trade-offs, you can predict shifts in OS design