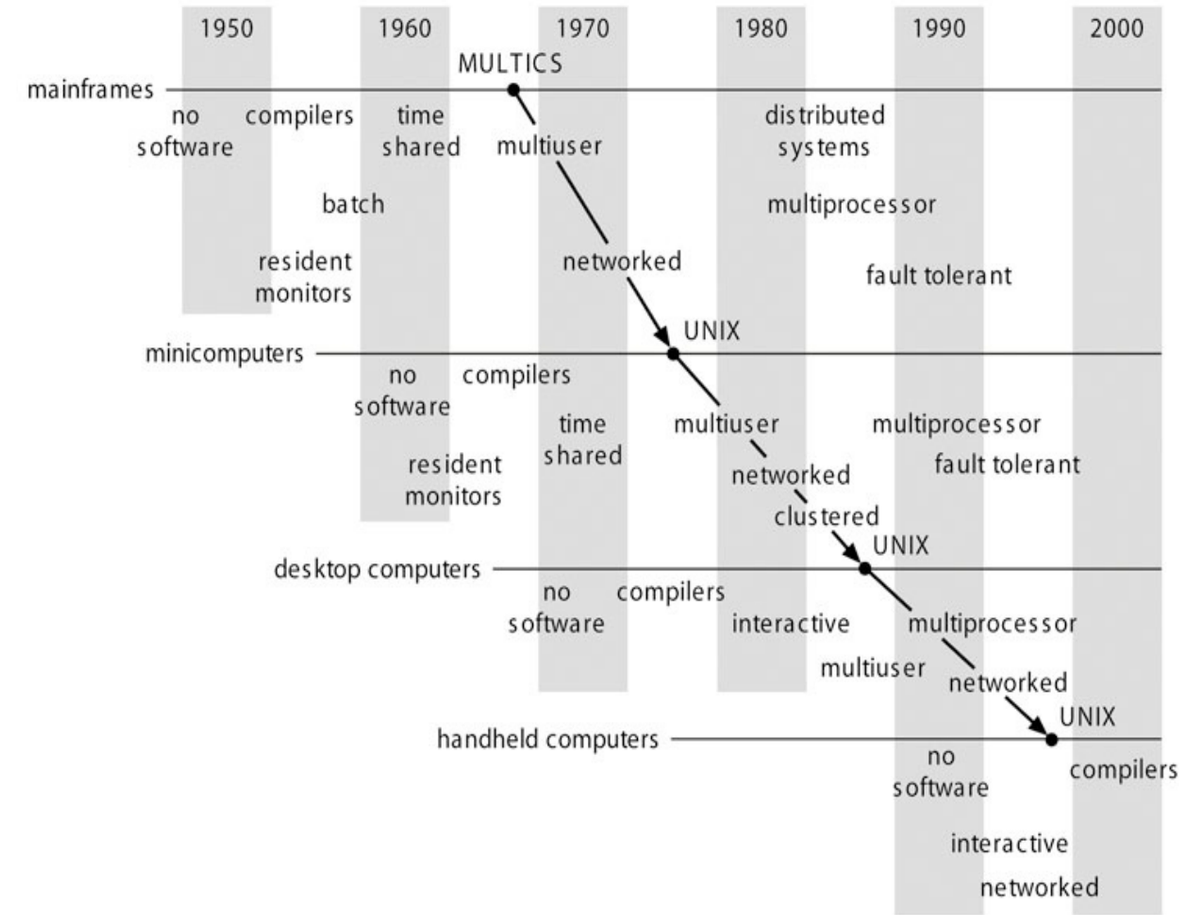
History of Operating Systems

Why review the history of OSs?

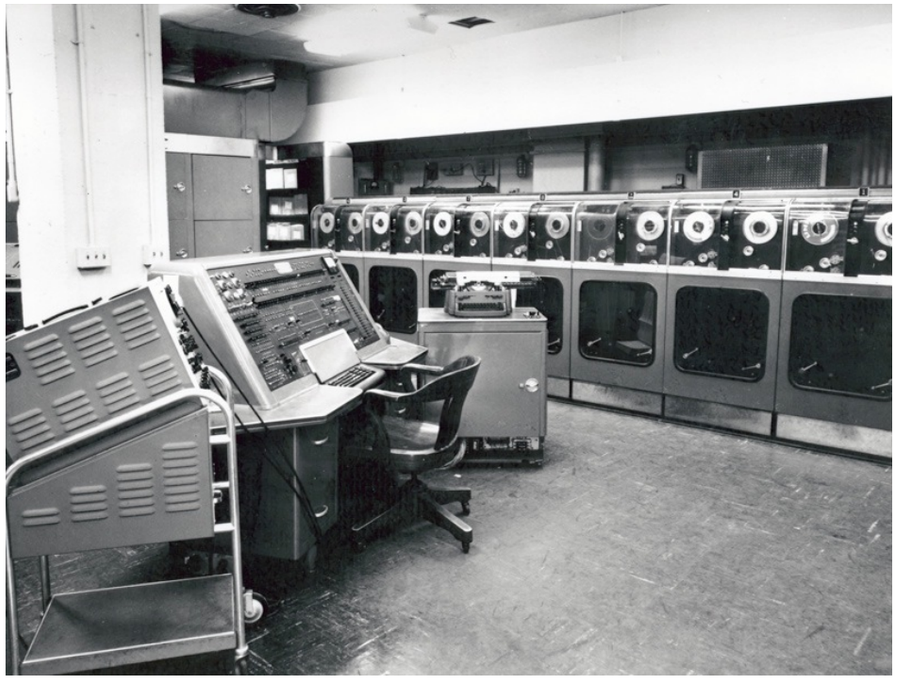
- Seeing how OSs developed shows the link between the capabilities of hardware and the design of OSs.
- It explains why OSs are as they are now.

- History seems to repeat.



This figure and some others in the slides are from an earlier version of the prescribed text *Operating Systems Concepts (8th edition)* by Silberschatz, Galvin and Gagne.

1950s



UNIVAC 1 14:40s

https://www.youtube.com/watch?v=ZU-IVshCAss

There was no UNIVAC provided operating system.

Scientists from the RAND Corporation have created this model to illustrate how a "home computer" could look like in the year 2004. However the needed technology will not be economically feasible for the average home. Also the scientists readily admit that the computer will require not yet invented technology to actually work, but 50 years from now scientific progress is expected to solve these problems. With teletype interface and the Fortran language, the computer will be easy to use and only

Total Control

- Circa 1950s
- Computers were very expensive.
- Users (they were all programmers) booked the whole machine. They had to:
    - prepare their program and data cards
    - do all the setup and loading required
    - control the computer through console switches
    - debug using console lights and switches
- This required a large amount of knowledge about the computer and peripherals.
- **Inefficient use of the machine.**
- Computers could execute 10s of thousands of instructions per second.
  But they were idle almost all the time.

Programming 1950's style

- *clear computer storage*
  - ready the compiler **paper** tape
  - ready the tape for the compiled output
  - put the source code cards in the card reader
  - set the switches to load the compiler
  - start the compiler
  - if errors - work out what they are and try again
- *clear computer storage*
  - ready the compiled object tape (still needs linking)
  - ready the i/o subroutines tape and linker
  - ready the tape for the output runnable program
  - load and run the linker
  - ready the printer or output tape
- *clear computer storage*
  - ready the runnable program tape
  - put the data cards in the card reader
  - load and run the program
  - if errors – work out what they are and try again

What did the OS look like?

- Most of the OS at this stage was comprised of the decisions and actions of the user.
- There were rudimentary components such as standard IO routines which were the forerunner of device drivers and system calls.
- No memory management – every address was reachable.
- No file system – the user loads the correct tapes.
- Security – the door could be locked.
- IO was polled for - no need for anything faster
- Some standard IO routines – useful code to read and write to tape and printers.
- Only one program at a time.
- No problems with synchronization.
- Programs communicate through paper tapes.
- User interface was almost the bare machine.
- Accounting maintained independently of the system.

Computer Operators & Off-lining

- Several speed-ups
  - experience
  - multiple operators (early multiprogramming?)
  - batching similar jobs together (sometimes called phasing)
  - keeping the programmer away from the computer

Changes

- No real changes from the hardware or OS perspective.
- But procedures were more formal.
- The first command UIs were instruction sheets to the operators.
- The next step was to automate some of these procedures.

Off-lining

- The arrival of magnetic tape substantially improved IO.
- Small cheap computers did the slow IO from paper tape to mag tape.
- And from mag tape to the printer.
- The Big Expensive Computer used the mag tapes for IO.
- Several programs submitted to the BEC on one tape.
- The first parallelism in computer systems.

Resident monitors

- The computer operators had formal procedures.
- Get the computer to help.

What it needs

- A program always in memory (hence the “resident”).
- A control language - commands had to be given to the resident monitor.
- The starting point of OSs.

Resident monitor could

- clear memory used by the last program (but not itself ☺️)
- load the next program
- find the data for the program
- jump to the start address of the new program, returning to the resident monitor when finished
- it also maintained the standard IO routines in memory

Control programs

- https://www.youtube.com/watch?v=PwftXqJu8hs IBM 1401 transistorised "In all, by the mid-1960s nearly half of all computer systems in the world were 1401-type systems."

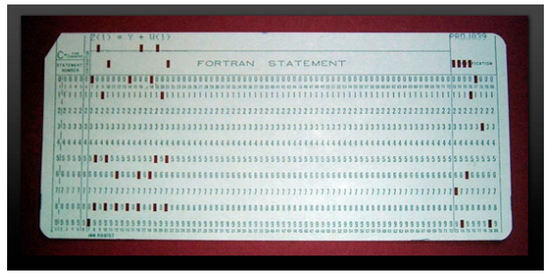
The resident monitor needed instructions.

Special cards that tell the resident monitor which programs to run

- $JOB
- $FTN
- $RUN
- $DATA
- $END

Special characters distinguish control cards from data or program cards:

- $ in column 1
- // in column 1 and 2
- 709 in column 1



The first Job Control Languages (JCLs) eventually became a command interpreter

What had changed?

- No memory management – every address was still reachable.
- Still no real file system, but there is a distinction between data and programs.
- Security – maintained by the operators.
- IO still polled for.
- Programmers now basically forced to use the standard IO routines.
- Only one program at a time. But two *programs* in memory.
- Still no problems with synchronisation.
- Problems with bad programs – system needed resetting when something bad happened.
- Depending on the types of devices the output of one program could automatically become the input of another.
- User interface was the JCL.
- Accounting still maintained independently of the system.

Changes in the hardware

Disk drives

- Disks provided substantially faster access to large amounts of storage.

Interruptible processors

- Devices raising interrupts and processors responding to them substantially changed the way IO was performed.
- Development from single location return addresses to the use of a stack.

I/O devices and the CPU can execute concurrently.

I/O is from the device to local buffer of controller.

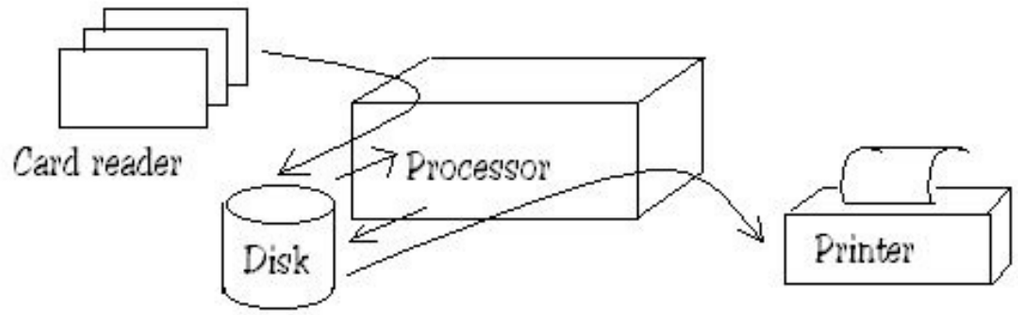
CPU moves data to main memory from local buffer of controller.

Device controller informs CPU that it has finished its operation by causing an *interrupt*.

SPOOLing

Simultaneous **P**eripheral **O**peration **O**n-**L**ine

- Time waiting for IO can be used.
- No longer need the small cheap computers for IO.



- Memory now holds
  - a running program
  - interrupt driven card reader control program
  - interrupt driven printer control program
  - disk control software
  - buffers for data being transferred between the computer and devices
  - a program loader
  - a JCL interpreter
  - a rudimentary file system – some data stays “permanently” on the disk

Multiprogramming

We are doing things simultaneously.

- processing a program
- reading cards for another program
- printing data for another program

The next step is obvious.

- Have several programs in memory at once.

What do we need?

- a lot more memory
- a scheduler
- a way of keeping track of which program is where in memory
- and where its data is, on card, disk etc
- better ways of handling errors
- a way to preserve the memory of each program

Memory protection

Can be provided by software.

- What is an example?
- A system that keeps programmers completely away from direct access to memory addresses.
- Alternatively, a check of every address by an instruction filter.

But far more efficiently and safely done by hardware.

Two requirements

- Operating modes and privileged instructions
- Limited address range

Provide hardware support to differentiate between at least two modes of operation.

1. *User mode* – execution done on behalf of a user.
2. *Kernel mode* (also *monitor mode*, *supervisor mode, privileged mode* or *system mode*) – execution done on behalf of the operating system.

Why do we need both?

If we had modes and privileged instructions but full memory access

- Obviously no memory protection
- but also no protection of the privileged instructions
    - put any code you want in the system areas of memory

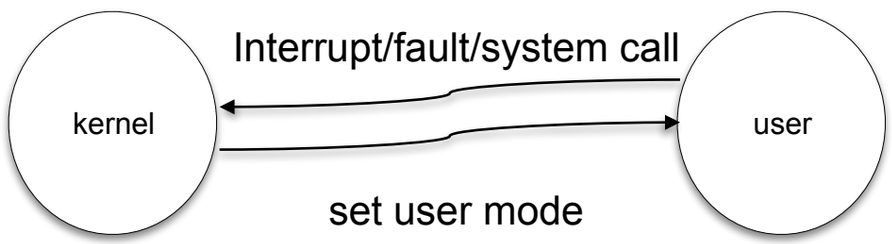
If we could limit memory accesses but there were no modes and privileged instructions

- Instructions are used to set up the memory management registers.
- If these are not privileged a user can change the area of memory available.

Processor modes

Mode bit

- Added to the hardware processor status register (or similar) to indicate which mode the processor is operating in.
- Interrupts, faults, system calls cause the processor to change mode and …
- jump to a particular location.
- Privileged instructions cannot be executed in user mode.



Memory protection

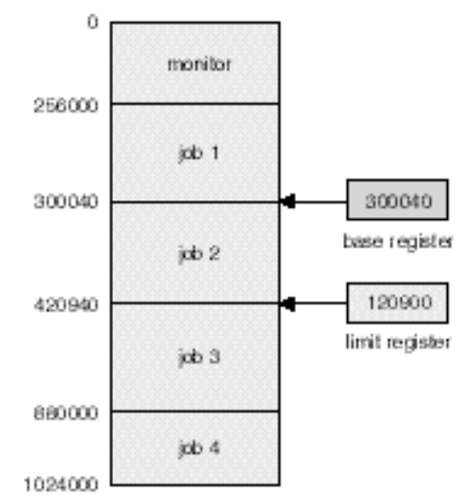
Each process gets allocated an area of memory which it can access.
All accesses outside that memory cause an exception (or fault).

- fixed size partitions

processes were designed to load in a particular partition

- base-limit registers
- memory pages

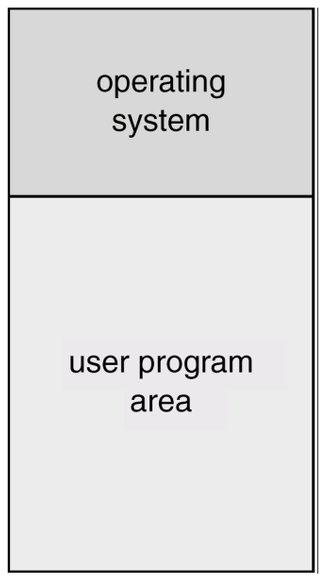
Devices can be protected using memory protection
or privileged instructions



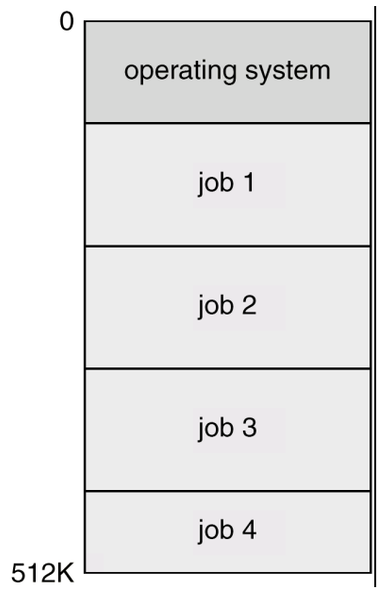
Batch systems

- With memory protection and processor modes we can safely have multiple programs in memory.

from



to



Batch system innovations

Each job had its own protected memory.

Disks with file systems.

- Files were associated with owners.

Scheduling was automated.

- The aim was to effectively utilise all of the expensive hardware.
- Computer operators now too slow.
- Individual jobs could be suspended or killed, allowing other jobs to progress.

Computer operators had consoles.

- Maybe even VDUs.

Accounting could now be done automatically.

But from the programmer's point of view nothing much had changed.

Before the next lecture

Read textbook sections

- 1.3 Computer-System Architecture
- 1.10 Computing Environments
- 2.8.5 Hybrid Systems

- If you want to know more about the current use of mainframes see:
  - https://www.ibm.com/it-infrastructure/z/education/what-is-a-mainframe