

Change of lecture theatre

For the rest of the semester the Friday lecture at 2pm will be in LibB15.

Monday 2pm MLT1

Wednesday 2pm MLT1

Friday 2pm LibB15

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.

It reminds us of the things learnt in 210 and 252

History seems to repeat.

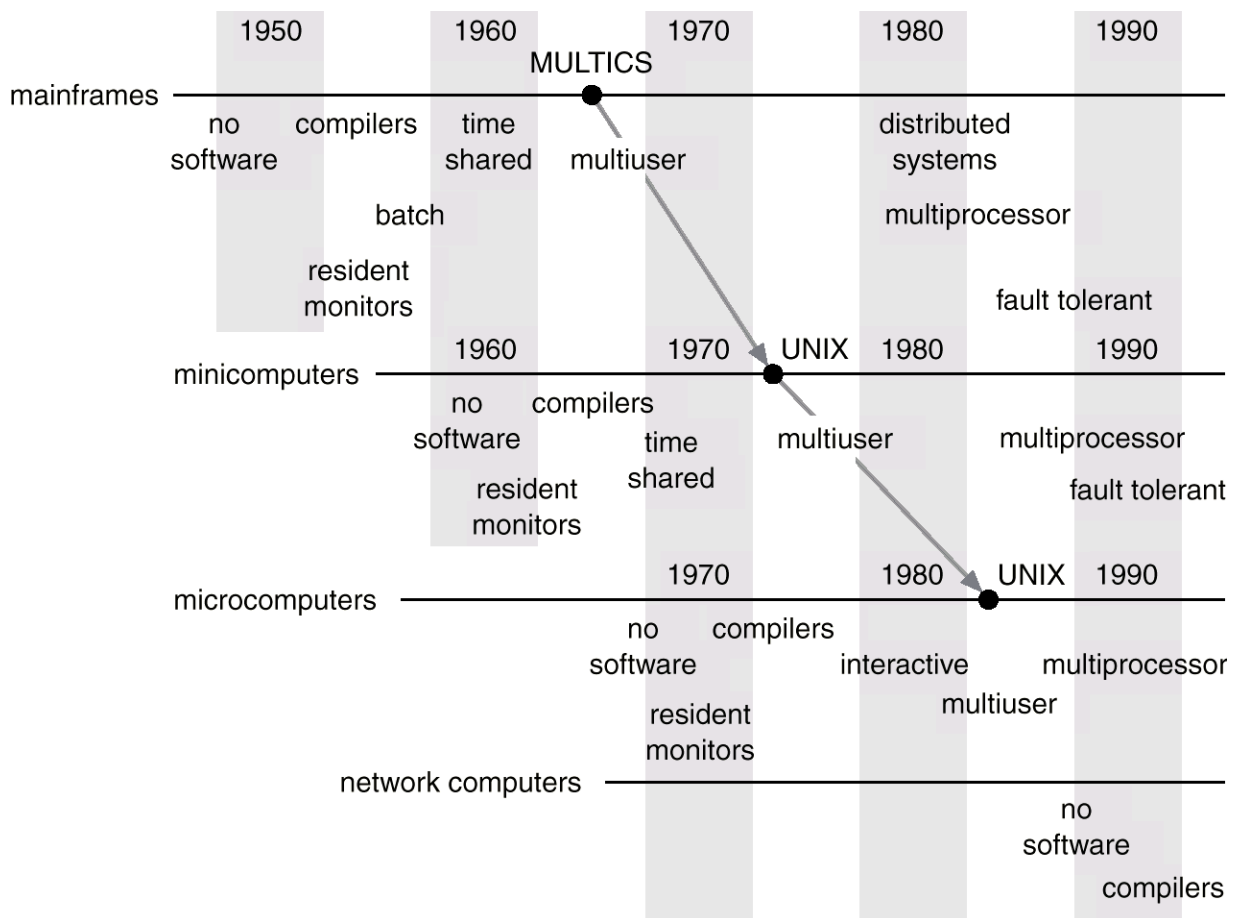


Figure from *Operating Systems Concepts with Java* by Silberschatz, Galvin and Gagne.

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 1000s of instructions per second.

But they were idle almost all the time.

COMPSCI 101 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 is 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 - how else?
- 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 UI was 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

- 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

What was missing?

Control programs

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).

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 synchronization.
- 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. Why?

Change 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.

CPU moves data from/to main memory to/from local buffers

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

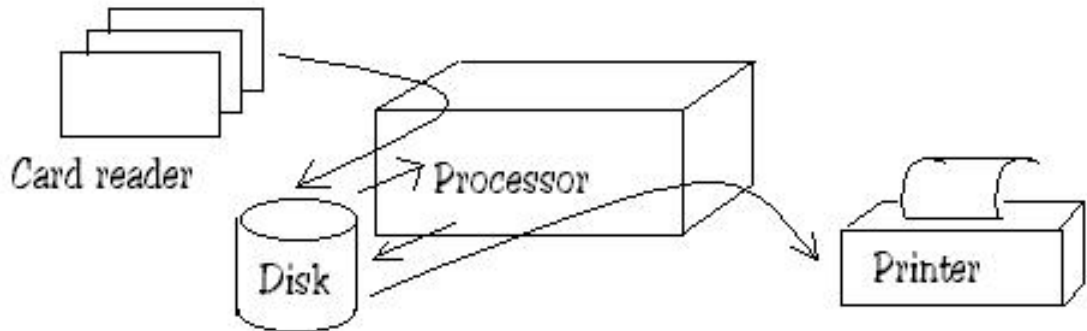
Device controller informs CPU that it has finished its operation by causing an *interrupt*.

SPOOLing

Simultaneous Peripheral Operation On-Line

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

Ruby threads

```
def simpleThreadProcedure(t)
  for count in (1..100)
    printf "from thread %2d: %3d\n", t,
                                                count
  end
end

threads = []

for i in (1..10)
  threads << Thread.new(i) do |thread|
    simpleThreadProcedure(thread)
  end
end

threads.each {|thread| thread.join}
```

Before the next lecture

Read textbook sections

- 1.2 Mainframe Systems
- 2.1 Computer-System Operation
- 2.2 I/O Structure
- 2.5 Hardware Protection