

CSC 539: Operating Systems Structure and Design

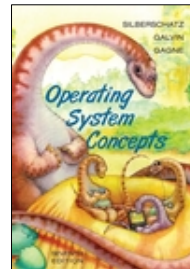
Spring 2006

See online syllabus at:

<http://www.dave-reed.com/csc539>

Course goals:

- to understand key operating system concepts and techniques, including process management, memory management, and file management.
- to appreciate the tradeoffs between performance and functionality in an OS (both theoretically and experimentally via simulation).
- to apply an understanding of OS concepts and tradeoffs to the evaluation of commercial operating systems.



1

What is an operating system?

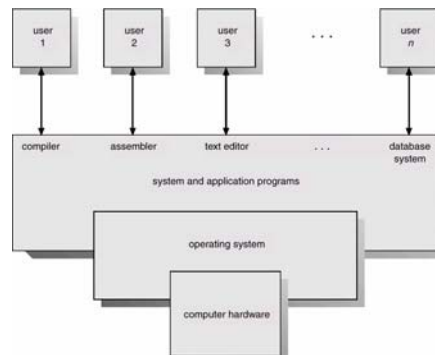
an OS is a collection of software that connects the user with the computer hardware

user perspective: OS is an interface

- makes interacting with the computer and running applications easy

system perspective: OS is a resource allocator

- controls and coordinates the hardware resources, services the application software



2

Why study operating systems?

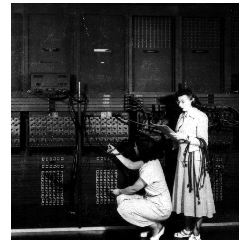
- as a user, you interact with an OS every time you use a computer
knowing how they work, tradeoffs, security risks, available options, etc. can make you a better user/consumer
- as an applications programmer, you write programs that rely on the OS
knowing how they work can lead to more efficient, more reliable programs
- as a systems programmer, you write code that integrates with the OS
must have a working knowledge of OS concepts and techniques
- as a computer scientist, you must know central, recurring themes
process management (scheduling algorithms, synchronization, deadlock)
memory management (caching, virtual memory, file systems)
human computer interaction (command-line interface, GUI)

3

History of operating systems

1940's: computers were special-purpose

e.g., ENIAC – wired to compute ballistics tables
to change the computation, had to rewire/reconfigure



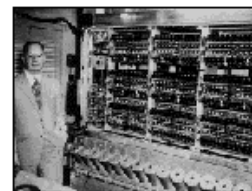
early 1950's: general-purpose computers were built

e.g., EDVAC, IAS – utilized the von Neumann architecture:
stored program in memory, CPU fetch-execute cycle

still, only a single user serviced at a time

typical setup:

- users would sign up for a time slot (e.g., 2am-3am)
- during that slot, had exclusive use of the computer
- must load program into memory space, overwriting old contents



no need for operating system: 1 program, 1 memory space, minimal peripherals

4

History of operating systems (cont.)

drawbacks of single-user, programmable computer

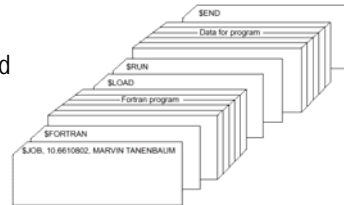
- inconvenient for user
- wasteful of computer time (computer was idle during setup, debugging, ...)

mid 1950's: batch processing was introduced

- idea: combine multiple user jobs into a single batch job

typical setup:

- users would submit jobs on cards/tape to computer operator
- operator would combine multiple jobs into a single batch job and load into card/tape reader
- each user job would be executed in turn
- users received output after *all* jobs finished



5

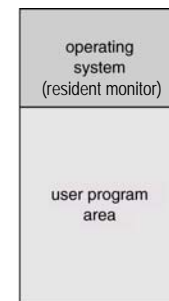
History of operating systems (cont.)

the primitive operating system in charge of executing the batch job was called a *resident monitor*

- it *resided* permanently in memory
- it *monitored* the execution of each job in succession

parts of a resident monitor

- control card interpreter (read & carry out card instructions)
- loader (load system programs & applications into memory)
- device drivers (control peripheral devices)



memory layout for a batch system

Note: when the monitor started a job, it handed over control of the computer to that job, regained control when the job terminated, then started the next job

6

History of operating systems (cont.)

advantages of batch systems

- moves much of the setup work from the user to the computer
- increased performance since could start a job as soon as previous job finished

drawbacks of batch systems

- turn-around time could be large from user standpoint
- more difficult to debug program
- due to the lack of a protection scheme, one batch job could affect pending jobs (e.g., read too many cards)
- a job could corrupt the monitor, thus affecting pending jobs
- a job could enter an infinite loop

eventually,

job corruption handled by modes (user vs. monitor), I/O limited to monitor mode

monitor corruption handled using memory protection

infinite loops handled by job execution timer

7

History of operating systems (cont.)

early 1960's: multiprogramming

as memory became cheaper, could load many user programs in memory simultaneously

- in contrast to batch, could partition the memory

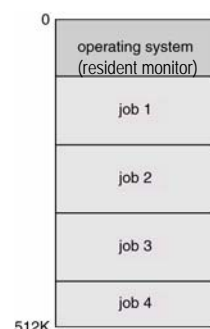
monitor (OS) could switch when a job became idle (e.g., I/O)

- since each job had its own memory partition, minimal overhead involved in switching

first multiprogramming OS: IBM OS/360

typical setup:

- users submitted jobs to computer operator
- jobs were loaded into separate memory partitions
- computer would begin executing first job
- during idle period, could switch to another job
- user could receive output as soon as their job terminated



**memory layout for a
multiprogramming
system**

8

History of operating systems (cont.)

mid 1960's: timesharing

timesharing (a.k.a. time-slicing) was the natural extension of multiprogramming

service multiple jobs "simultaneously" by automatically switching between jobs frequently (fractions of sec)

- to each user, appearance of sole use

```
bluejay> cd FILES
bluejay> ls
foo.txt
bluejay> more foo.txt
foo bar biz baz
bluejay> date
Sun Aug 25 23:47:26 CDT 2002
bluejay> █
```

first timesharing OS: CTSS, followed by MULTICS, UNIX

typical setup:

- multiple users connected via remote terminals
- each user could interact using a *command-line interface*
- each user would see results in real-time – job might require numerous time slices to complete, but fast enough to be unnoticeable

```
C:\>cd FILES
C:\FILES>dir
Volume in drive C has no label
Volume Serial Number is 1068-1506
Directory of C:\FILES

.           <DIR>          08-25-02  5:50p .
..          <DIR>          08-25-02  5:50p ..
foo        TXT           17  08-25-02  5:52p foo.txt
.           <DIR>          08-25-02  5:52p .
..          <DIR>          08-25-02  5:52p ..
4,996,16  free

C:\FILES>more foo.txt
foo bar biz baz

C:\FILES>time
CURRENT TIME IS 5:53:01.35p
ENTER NEW TIME:
C:\FILES>
```

9

History of operating systems (cont.)

late 1970's: desktop systems

personal computers bucked the trend of centralization

- provided a machine cheap enough to be used (and wasted) by a single user

typical setup:

- single user worked at dedicated machine
- at least early on, all peripherals were connected directly (networking not widespread until 90's)

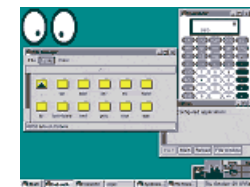
early PC's utilized a command-line interface

- CP/M (1977), MS-DOS (1981)

Graphical User Interface (GUI) introduced by Macintosh

- Mac-OS (1984), Windows (1985), Motif (1989)

WIMP (Windows/Icons/Menus/Pointer) interface was pioneered by Doug Engelbart in 1960's, first adopted at Xerox PARC



10

New computer architectures → new demands

multiprocessor systems (a.k.a. parallel systems, or tightly coupled systems)

more than one processor in close communication, share bus and clock, other resources can increase throughput, save on shared resources, provide greater reliability

- *symmetric multiprocessing (SMP)*: each CPU runs a copy of the OS, can communicate
most modern OS provide support for SMP
- *asymmetric multiprocessing*: master CPU assigns specific tasks to slave CPUs
more common in large systems with specialized hardware

distributed systems (a.k.a. loosely coupled systems)

distribute the computation among several processors, each with its own memory & resources
processors communicate via communications lines, e.g., high-speed buses or phone lines
similar benefits as multiprocessor, but generally simpler, cheaper, more scaleable (but slower?)

- may be structured as client/server or peer-peer, LAN or WAN

clustered systems

multiple systems share storage and are closely linked via LAN networking

- *symmetric clustering*: all systems run applications simultaneously, monitor each other
- *asymmetric clustering*: one machine runs, other monitors & waits in stand-by mode

11

New architectures (cont.)

real-time systems

some systems impose well-defined, fixed-time constraints

e.g., control for scientific experiments, medical imaging systems, industrial control systems, ...

- *hard real-time*: critical tasks must be completed in specified time
secondary storage limited or absent, data stored in short term memory or ROM
not compatible with timesharing
- *soft real-time*: critical tasks are given priority over other tasks, but no guarantees
limited utility in industrial control of robotics
useful in apps (multimedia, VR) requiring advanced OS features

handheld systems

OS functionality is migrating to PDAs, cellular phones

issues: limited memory, slow processors, wireless transfer, small display screens

12