

# OS Concepts and structure

---



## Today

- OS services
- OS interface to programmers/users
- OS components & interconnects
- Structuring OSs

## Next time

- Processes

# OS Views

---

- Vantage points
  - OS as the services it provides
    - To users and applications
  - OS as its components and interactions
- OS provides a number of services
  - To users via a command interpreter/shell or GUI
  - To application programs via system calls
  - Some services are for convenience
    - Program execution, I/O operation, file system management, communication
  - Some to ensure efficient operation
    - Resource allocation, accounting, protection and security

# Command interpreter (shell) & GUI

---

- Command interpreter

- Handle (interpret and execute) user commands
- Could be part of the OS: MS DOS, Apple II
- Could be just a special program: UNIX, Windows XP
  - In this way, multiple options – shells – are possible
- The command interpreter could
  - Implement all commands
  - Simply understand what program to invoke and how (UNIX)

- GUI

- Friendlier, through a desktop metaphor, if sometimes limiting
- Xerox PARC Alto >> Apple >> Windows >> Linux

# System calls

- Low-level interface to services for applications
- Higher-level requests get translated into sequence of system calls
- Writing `cp` – copy source to destination

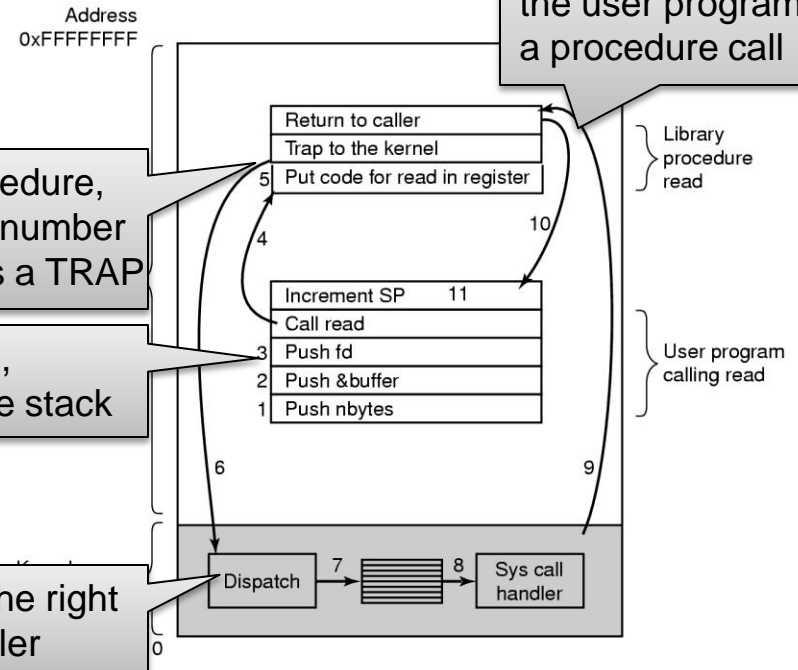
- Get file names
- Open source
- Create destination
- Loop
  - Read from source
  - Copy to destination
- Close destination
- Report completion
- Terminate

Then call the library procedure, which places the syscall number in a register, and executes a TRAP

Before calling the syscall, push parameters onto the stack

Kernel runs the right sys call handler

Before returning to the user program as a procedure call



**Making the system call:** `read(fd, buffer, nbytes)`

# Major OS components & abstractions

---

- Processes
- Memory
- I/O
- Secondary storage
- File systems
- Protection
- Accounting
- Shells & GUI
- Networking

# Processes

- A program in execution
  - Address space
  - Set of registers
- To get a better sense of it
  - What data do you need to (re-) start a suspended process?
  - Where do you keep this data?
  - What is the process abstraction I/F offered by the OS
    - Create, delete, suspend, resume & clone a process
    - Inter-process communication & synchronization
    - Create/delete a child process

Call	Description
<code>pid = fork()</code>	Create a child process identical to the parent
<code>pid = waitpid(pid, &amp;statloc, options)</code>	Wait for a child to terminate
<code>s = execve(name, argv, environp)</code>	Replace a process' core image
<code>exit(status)</code>	Terminate process execution & return status

# Memory management

- Main memory – the directly accessed storage for CPU
  - Programs must be stored in memory to execute
  - Memory access is fast (e.g., 60 ns to load/store)
    - but memory doesn't survive power failures
- OS must
  - Allocate memory space for programs (explicitly and implicitly)
  - Deallocate space when needed by rest of system
  - Maintain mappings from physical to virtual memory
    - e.g. through page tables
  - Decide how much memory to allocate to each process
  - Decide when to remove a process from memory

Call	Description
<code>void *sbrk(intptr_t increment)</code>	Increments program data space by 'increment' bytes

# I/O

---

- A big chunk of the OS kernel deals with I/O
  - Hundreds of thousands of lines in NT
- The OS provides a standard interface between programs & devices
  - file system (disk), sockets (network), frame buffer (video)
- Device drivers are the routines that interact with specific device types
  - Encapsulates device-specific knowledge
    - e.g., how to initialize a device, request I/O, handle errors
  - Examples: SCSI device drivers, Ethernet card drivers, video card drivers, sound card drivers, ...



# Secondary storage

---

- Secondary storage (disk, tape) is persistent memory
  - Often magnetic media, survives power failures (hopefully)
- Routines that interact with disks are typically at a very low level in the OS
  - Used by many components (file system, VM, ...)
  - Handle scheduling of disk operations, head movement, error handling, and often management of space on disks
- Usually independent of file system
  - Although there may be cooperation
  - File system knowledge of device details can help optimize performance
    - e.g., place related files close together on disk

# File systems

---

- Secondary storage devices are hard to work with
- File system offers a convenient abstraction
  - Defines logical abstractions/objects like files & directories
  - As well as operations on these objects
- A file is the basic unit of long-term storage
- A directory is just a special kind of file
  - ... containing names of other files & metadata
- Interface:
  - File/directory creation/deletion, manipulation, copy, lock
- Other higher level services: accounting & quotas, backup, indexing or search, versioning

# Protection

---

- Protection is a general mechanism used throughout the OS
  - All resources needed to be protected
    - memory
    - processes
    - files
    - devices
    - ...
- Protection mechanisms help to detect and contain errors, as well as preventing malicious destruction

# OS structure

---

- OS made of number of components
  - Process & memory management, file system, ...
  - and system programs
    - e.g., bootstrap code, the init program, ...
- Major design issue
  - How do we organize all this?
  - What are the modules, and where do they exist?
  - How do they interact?
- Massive software engineering
  - Design a large, complex program that:
    - performs well, is reliable, is extensible, is backwards compatible, ...

# OS design & implementation

---

- *User goals and System goals*
  - User – convenient to use, easy to learn, reliable, safe, fast
  - System – easy to design, implement, & maintain, also flexible, reliable, error-free & efficient
- Affected by choice of hardware, type of system
- Clearly conflicting goals, no unique solution
- Some other issues complicating this
  - Size: Windows XP ~40G SLOC, RH 7.1 17G SLOC
  - Concurrency – multiple users and multiple devices
  - Potentially hostile users, but some users want to collaborate
  - Long expected lives & no clear ideas on future needs
  - Portability and support to thousands of device drivers
  - Backward compatibility

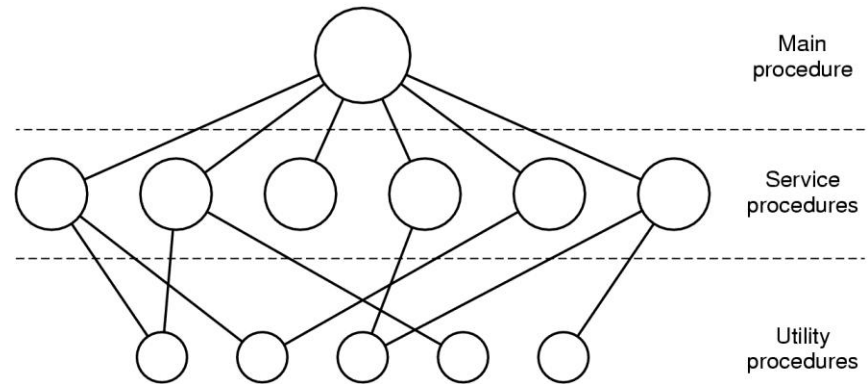
# OS design & implementation

---

- A software engineering principle – separate policy & mechanism
  - Policy: What will be done?
  - Mechanism: How to do it?
  - Why do you care? Max flexibility, easier to change policies
- Implementation on high-level language
  - Early on – assembly (e.g. MS-DOS – 8088), later Algol (MCP), PL/1 (MULTICS), C (Unix, ...)
  - Advantages – faster to write, more compact, easier to maintain & debug, easier to port
  - Cost – Slower, but who cares?!

# Monolithic design

- Major advantage:
  - Cost of module interactions is low (procedure call)
- Disadvantages:
  - Hard to understand
  - Hard to modify
  - Unreliable (no isolation between system modules)
  - Hard to maintain
- Alternative?
  - How to organize the OS in order to simplify its design and implementation?



# Layering

- The traditional approach
  - Implement OS as a set of layers
  - Each layer shows an enhanced ‘virtual mach’ to layer above
- Each layer can be tested and verified independently

Layer	Description
5: Job managers	Execute users' programs
4: Device managers	Handle device & provide buffering
3: Console manager	Implements virtual consoles
2: Page manager	Implements virtual memory for each process
1: Kernel	Implements a virtual processor for each process
0: Hardware	

Dijkstra's THE system



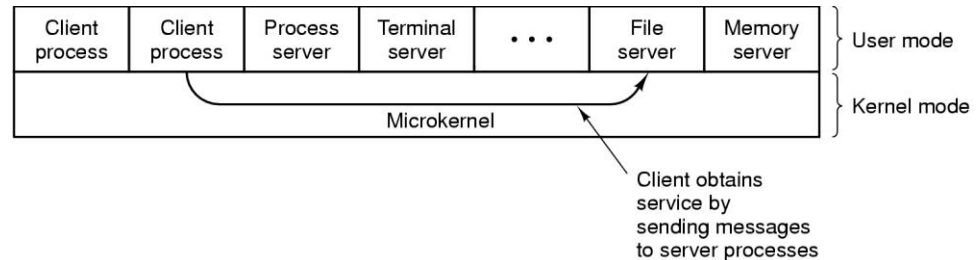
# Problems with layering

---

- Imposes hierarchical structure
  - but real systems have complex interactions
  - Strict layering isn't flexible enough
- Poor performance
  - Each layer crossing implies overhead
- Disjunction between model and reality
  - Systems modelled as layers, but not built that way

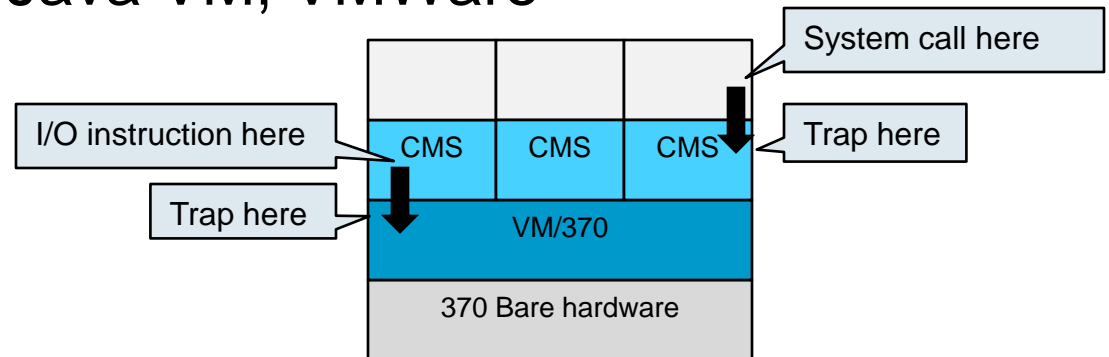
# Microkernels

- Popular in the late 80's, early 90's
  - Recent resurgence
- Goal
  - Minimize what goes in kernel
  - Organize rest of OS as user-level processes
- This results in
  - Better reliability (isolation between components)
  - Ease of extension and customization
  - Poor performance (user/kernel boundary crossings)
- First microkernel system was Hydra (CMU, 1970)
  - ... Mach (CMU), Chorus (French UNIX-like OS), OS X (Apple), in some ways NT (Microsoft), L4 (Karlsruhe), ...



# Virtual machines

- Initial release of OS/360 were strictly batch but users wanted timesharing
  - IBM CP/CMS, later renamed VM/370 ('79)
- Note that timesharing systems provides (1) multiprogramming & (2) extended (virtual) machine
- Essence of VM/370 – separate the two
  - Heart of the system (VMM) does multiprogramming & provides to next layer up multiple exact copies of bare HW
  - Each VM can run any OS
- More recently – Java VM, VMWare



# Summary & preview

---

- Today
  - The mess under the carpet
  - Basic concepts in OS
  - OS design has been an evolutionary process
  - Structuring OS - a few alternatives, not a clear winner
- Next ...
  - Process – the central concept in OS
    - Process model and implementation
    - What it is, what it does and how it does it