Outline

- Review: Computer-system structures
  - I/O structures
  - storage structures and hierarchy
    - memory, secondary storage, tape
  - hardware support for protection

- Operating-system structures
  - the functional view (what it does)
  - the components view (designer’s view)
  - the services view (user/programmers’ view)
  - the structure view (implementers’ view)

[Silberschatz/Galvin: Chapters 2-3]

The Hardware of a Modern Computer System

The diagram illustrates the hardware components of a modern computer system, including CPU, memory, disk controller, display controller, tape-drive controller, printer controller, I/O Bus Bridge, and Memory Bus.

Computer-System Structures (1): Input/Output

- Device controllers
  - special-purpose processors
  - local buffer storage
  - controllers contain registers
    - control (write-only)
    - data (read-write)
    - status (read-only)

- Instructions
  - read/write I/O addresses
    - e.g., video memory
  - registers in I/O controllers addressed as memory

- Interrupts
  - device controllers can interrupt the CPU
**Interrupt Handling**

- Interrupts are “asynchronous requests for service”

- When an interrupt occurs, the CPU
  - preserves the present CPU state
    - this includes its registers and program counter
  - forces execution of code at an interrupt address
    - this may be dependent on the source of the interrupt
    - typically, table-driven: a table stores addresses of interrupt handlers
  - interrupt handlers
    - perform the requested service
    - selective processing of other interrupts
      - e.g., only higher-priority interrupts may be handled
    - resumes the interrupted program

- Most modern OSes are interrupt-driven

**Solution: Direct Memory Access (DMA)**

- The main idea: add a special device to “intervene” between the device controller and the system’s memory

- Operation
  - the CPU tells this DMA controller
    - the “chunk” size to be transferred
      - e.g., 128 - 4096 bytes (sectors) for disks
    - the starting address in memory where this chunk ought to be stored
  - the DMA controller
    - accesses the secondary device via its controller
    - transfers the chunk from the device to system memory (and vice-versa)

- Benefit: Interrupts are now less frequent
  - at the level of chunks of data: only to indicate completion
  - hence, CPU can do a lot of work between interrupts

**I/O Operation**

- Two approaches: Synchronous and Asynchronous

- Problem with the above schemes: CPU handles all I/O
  - it can spend all its time doing interrupt processing
    - disk I/O, network I/O, video I/O

**Memory-mapped I/O**

- Traditionally, the CPU could directly access only main memory
  - all other devices are handled via controllers
  - accessed using special I/O instructions

- In most recent systems
  - controller’s registers are mapped into RAM space
    - results in uniform treatment of the I/O devices
      - can be handled via memory management procedures
      - all addressing is to RAM space
      - DMA access, interrupt handling, polling, …
  - controller’s buffers are mapped into RAM space
    - makes sense if the I/O is to a device that is particularly fast
      - e.g., a CRT screen where each pixel is an addressable location in RAM
Computer-System Structures (2): Storage

- Primary storage: Main memory (volatile)
  - accessed directly using load/store instructions
    - 1 cycle (registers), 2-5 cycles (cache), 20-50 cycles (RAM)
  - before: only one outstanding memory operation, CPU waits for completion
  - now: several outstanding operations
- Secondary storage: Disks (non-volatile)
  - accessed using a disk controller
  - supports random access but with non-uniform cost
- Tertiary storage: Tapes (non-volatile)
  - typically used only for backup
  - very inefficient support for random access
- Organized as a hierarchy
  - small amount of faster, more expensive storage closer to the CPU
  - larger amounts of slower, less expensive storage further away

Storage Hierarchy

- Rationale
  - keep CPU busy: lots of fast memory
  - keep system cost down
- How does it work
  - caching: upon access, move datum or instruction and its neighbors into higher levels of the hierarchy
  - replacement when a level fills up
- Why does it work
  - Real programs demonstrate locality
    - e.g.: rows and columns of a matrix
    - e.g.: sequential instructions
  - once a datum or instruction is used, things “near” them are likely to be used “soon”

Structure of Disks

- A stack of platters are mounted on a spindle
  - spindle rotates the platters past read/write heads
  - heads are separated from the platters by a few microns!
  - the separation permits very high rates of rotation
    - 60 - 150 rotations per second
  - magnetic fields induce electrical fluctuations in the head and vice-versa
  - result is a string of 0s and 1s being read/written

Disk Access Costs

- Each platter organized as several concentric tracks
- Data is written into sectors (portions of a track)
- Access time = Seek time + Rotational latency + Disk bandwidth
  - Seek time: time to move the disk arm (head) to the right track
  - Rotational latency: time for the correct sector to appear under the head
  - Disk bandwidth: rate of data transfer
  - currently, seek time and rotational latency are bottlenecks
- OS/Architecture interactions (more details in Lectures 15-17)
  - disk scheduling should minimize seek time and rotational latency
    - similar to elevator scheduling
  - file caches to minimize disk accesses
  - log-structured file system design to improve disk write performance
Magnetic Tapes

- Primarily used as a backup device
- Very similar to audio tape technology
- Sequential access
  - the read/write head moves past the tape sequentially
  - once positioned, transfer is quite rapid
- Can store terabytes ($10^{12}$) of data

Computer-System Structures (3): Protection

- Goal: Prevent user processes from accidently/maliciously damaging
  - the OS structures
  - parts of other process's memory space
  - other user's I/O devices

- Mechanisms address different ways in which protection breaks down
  - dual-mode operation
    - user process taking over part of the OS and using this to overwrite other processes or even modify the OS itself (as in MS-DOS)
  - privileged instructions
    - user process intervening in I/O of another process via control of the I/O handlers and indirectly causing damage
  - memory protection
    - user process directly accessing another user process' storage
  - CPU protection via timers
    - hanging the OS -- e.g., via an infinite loop

Dual-mode Operation and Privileged Instructions

- Dual-mode operation
  - supervisor and user modes
  - system starts off in supervisor mode and reenters it for interrupt processing
  - operating system gains control in supervisor mode

- Privileged instructions
  - restrict use of certain instructions to supervisor mode
    - I/O, including interrupt control
      - may be done by memory mapping
    - affect memory mapping
    - affect CPU mode (user/supervisor)
  - hardware support crucial for performance and for atomicity

Memory Protection

- Basic method: Memory is divided into segments

- Furthermore
  - logical addresses are mapped to physical addresses
    - provides sharing, etc.
  - hardware support for address mapping
  - on a memory protection violation
    - user process traps to (interrupts) the OS
### CPU Protection via Timers

- Timers maintain a count of elapsed (system) clock ticks
- Used for
  - interrupting hung processes
  - context switching in time-shared systems
- Access to timers is (usually) privileged

---

### Outline

- **Review: Computer-system structures**
  - I/O structures
  - storage structures and hierarchy
    - memory, secondary storage, tape
    - hardware support for protection
- **Operating-system structures**
  - the functional view (what it does)
  - the components view (designer’s view)
  - the services view (user/programmers’ view)
  - the structure view (implementers’ view)

[Silberschatz/Galvin: Chapters 2-3]

---

### Functional View: Directly Motivated

- Program execution and handling:
  - starting programs, managing their execution, and communicating results
- I/O operations:
  - mechanisms for initiating and managing I/O
- File-system management:
  - creating, maintaining and manipulating files
- Communication:
  - between processes of the same user
    - such as sending the results of an input request to a user program,
    - between different users
- Exception detection and handling:
  - protection related issues
  - safety in the case of power failures via backups
  - detecting undesirable states such as printers out of paper
Components View

- Components
  - processes
  - storage
  - I/O
    - devices
    - files
  - protection
    - users

- Expect a system component for each
  - intertwined in practice

Components View: Processes

- A process is
  - the dynamic entity defined by the execution of a program
  - typically, program + run-time control information
    - control information includes memory maps, program counter values etc.
    - may also have context information
  - OS attempts to treat all processes uniformly

- Processes may play different roles
  - user processes
  - OS (system) processes

- A single process can spawn other processes
  - a computation typically requires many processes
    - shell, one or more user processes, one or more system processes

Components View: Process Management

- Operations
  - creation and termination
  - suspension and resumption
    - due to interrupts, context switches ...
  - synchronizing processes
    - making sure that a process that is waiting on an I/O waits till it is completed and does not wait forever, i.e., wakes up soon after the I/O process terminates
    - managing concurrent access to shared resources
  - communication
    - between two processes enabling them to cooperate
    - deadlock detection and avoidance

- Detailed discussion in Lectures 3-11

Components View: Storage Management

- Managing main memory
  - allocating main memory to active processes
    - maintaining a map of allocated vs. free memory
  - deallocating currently used memory to make room for new processes

- Managing secondary storage
  - managing the free sectors/tracks on disks
  - allocating this storage to programs
  - scheduling access requests to the disk

- Detailed discussion in Lectures 12-14, 17
Components View: I/O Management

- Devices
  - device drivers
  - accepting an I/O request and invoking the appropriate device driver

- Files
  - non-volatile representations of user/system programs and data
  - file systems
    - support logical organization of data that the user might want to see
    - map data onto the physical storage devices and orchestrate their access/update
  - operations
    - creation, manipulation, and deletion of files and directories
    - moving files from primary to secondary storage while maintaining structure
    - interaction with the memory manager
    - backup and protection

- Detailed discussion in Lectures 15-18

Services View

- Command interpreter
- Two levels of specification
  - system calls
    - the interfaces through which processes invoke specific OS functions
  - system programs
    - may use capabilities not available via system calls
    - the interfaces at these levels can be standardized
      - e.g. POSIX, network protocols, ...

Services View: Command Interpreters

- Interface between the user and the operating system
  - can be a part of the kernel or a separate process (shell)
  - striking differences between OSes
    - range from GUIs to (cryptic) control card interpreters
  - typical commands
    - process creation and (implicitly) destruction
    - I/O handling and file system manipulation
    - communication: interact with remote devices
    - protection management: changing file/directory access control, etc.

- Three varieties of shells:
  - the interpreter contains the code for the requested command (e.g., delete)
  - the interpreter calls a system routine to handle the request
  - the interpreter spawns new process(es) to handle the request
    - process lookup through some general procedure
    - most extensible

Services View: System Calls

- Interface between a process and the operating system
  - arguments typically passed in registers, a memory block, or on the stack

  Process control
  - load, execute, end, abort, create/terminate
  - get and set process attributes (priorities, allowable execution times)
  - wait for time/event, signal event, allocate memory

  File manipulation
  - create/delete, open/close, read/write and reposition
  - get/set file attributes (protection parameters, locations in directories ...)

  Device manipulation
  - same set of calls as above (devices are treated as special files) + attach/detach

  Information maintenance
  - get/set system data (time, memory/CPU usage), process and device attributes

  Communications
  - create/delete links, send/receive messages
  - transfer status information
  - modes can be message passing or shared memory
Services View: System Programs

- Each system call is usually supported by a system program
- Additionally, we also have standard applications
  - command interpreter
  - programming language support:
    - compilers, assemblers, debuggers
  - loaders and linkers:
    - for enabling the execution of the programs that start new processes and connect different ones
  - application programs:
    - text processing software, graphics support, spreadsheets, games

Structure View

- OSes as client-server systems
  - single server
  - multiple servers
  - serve yourself (e.g., UNIX)
- How to structure OS functionality
  - layering
  - virtual machines
- Designing and implementing an OS

Structure View: OSes as Client-Server Systems

Single-server

<table>
<thead>
<tr>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>user process</td>
</tr>
<tr>
<td>user process</td>
</tr>
<tr>
<td>user process</td>
</tr>
<tr>
<td>user process</td>
</tr>
</tbody>
</table>

Many servers

<table>
<thead>
<tr>
<th>OS1</th>
<th>OS2</th>
<th>OS3</th>
</tr>
</thead>
<tbody>
<tr>
<td>user process</td>
<td>user process</td>
<td>user process</td>
</tr>
</tbody>
</table>

Serve yourself

<table>
<thead>
<tr>
<th>OS1</th>
<th>OS2</th>
<th>OS3</th>
<th>OS4</th>
</tr>
</thead>
<tbody>
<tr>
<td>user process</td>
<td>user process</td>
<td>user process</td>
<td>user process</td>
</tr>
</tbody>
</table>

Most OSes are hybrids of these structures

Structure View: UNIX Structure

- Execution of a UNIX process
  - fork creates a new process: copies memory, PCB, adds to ready queue
  - new process executes
    - initially in supervisor mode: OS executes last bit of fork code
    - fork then returns to user mode: user process resumes
    - process switches to supervisor mode when user code executes a system call
      - OS services system call
        - process switches to user mode upon completion and user code resumes

Shared System Tables

<table>
<thead>
<tr>
<th>OS</th>
<th>OS</th>
<th>OS</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>user process</td>
<td>user process</td>
<td>user process</td>
<td>user process</td>
</tr>
</tbody>
</table>
Structure View: Structure of OS Designs

- Traditionally, adhoc and unstructured approaches
- Structured designs
  - encapsulate the modules separately
  - have a hierarchy of layers where each layer strictly calls procedures from the layers below

The THE OS (Dijkstra)       The VENUS OS (Liskov)
Layer 0: hardware          Layer 0: hardware
Layer 1: CPU scheduler     Layer 1: instruction interpreter
Layer 2: memory manager    Layer 2: CPU scheduling
Layer 3: operator-console device driver Layer 3: I/O channel
Layer 4: I/O buffering     Layer 4: virtual memory
Layer 5: user program layer Layer 5: device drivers and schedulers
Layer 6: user programs

Structure View: Layered System Design

- A general philosophy that builds on the above approach
  - decompose functionality into layers such that
    - hardware is level 0, and layer i accesses functionality at layers (i-1) or less
    - access via appropriately defined system calls
- Advantages:
  - modular design: well-defined interfaces between layers
  - prototyping/development:
    - association between function and layer eases overall OS design
    - OS development and debugging is layer-by-layer
- Disadvantages
  - substantial design time since layers have to be designed carefully
  - crossing multiple layers (of system calls) leads to inefficiency
  - OSs are not naturally hierarchical
    - two-level approaches (micro-kernel + one more) are most successful

Structure View: Virtual Machines

- Logical conclusion of layered approach
  - the OS offers an abstract machine to execute on
    - hides the specifics and details of hardware
    - provides a complete copy of the underlying machine to the user
      - pioneered under the name Virtual Machine (VM) by IBM
      - became a household name with Java :)
- Mechanisms for creating this illusion
  - CPU scheduling
    - sharing the CPU in a transparent way
  - memory management
    - having a large virtual space to address
  - additional features such as file-systems and so on are offered through file management sub-systems
    - problem is with partitioning the disk
      - VM introduced minidisks: partitioning of tracks on the physical disk

Structure View: Virtual Machines (contd.)

- Operation
  - user-level code executes as is (hopefully)
  - supervisor-level code executes at user-level
    - privileged instructions are simulated
      - generate trap to VM emulator
    - hidden registers and I/O instructions are simulated
- Advantages
  - a software created abstraction that is a replica of the underlying machine
  - additional functionality can be provided
    - can run different Oses on different VMs
    - each VM is completely isolated from others and hence is secure
- Disadvantages:
  - performance degradation inevitable
    - can be alleviated with hardware support (e.g., PAL codes in Alpha chips)
Structure View: Mechanisms vs. Policies

- Mechanisms describe *how* things are done
  - the function of CPU scheduling is an example
  - does not include decisions as to which process gets priority

- Policies describe *what* will be done
  - deciding whether programs with more I/O get priority over those requiring processing is an example
  - this policy can be an input parameter and can change from one scheduler to the next

- Microkernel-based operating systems provide only mechanisms
  - policies are implemented at the user-level

Structure View: System Generation

- Regenerate the OS for new machine parameters
  - machine parameters include:
    - memory size, I/O device parameters, address maps, etc.

- Approaches
  - recompile
  - have pre-compiled parameterized libraries; link rather than recompile
  - use table-driven run-time selection

Lecture Summary

- Computer-system structures
  - I/O structures
  - storage structures
  - support for protection

- Operating-system structures
  - functional view
  - components view
  - services view
  - structure view

Next Lecture

- Processes and Threads
  - process concept
  - process scheduling
  - operation on processes
  - cooperating processes
  - threads

Reading

- Silberschatz/Galvin: Sections 4.1-4.5