OS structure

• The OS sits between application programs and the hardware
  – it mediates access and abstracts away ugliness
  – programs request services via exceptions (traps or faults)
  – devices request attention via interrupts
Major OS components

- Processes
- Memory
- I/O
- Secondary storage
- File systems
- Protection
- Security
- Networking
- Accounting
- Shells (command interpreter, or OS UI)
- GUI
Process management

• An OS executes many kinds of activities:
  – users’ programs
  – batch jobs or scripts
  – system programs
    • print spoolers, name servers, file servers, network daemons, …

• Each of these activities is encapsulated in a process
  – a process includes the execution context
    • PC, registers, VM, OS resources (e.g., open files), etc…
    • plus the program itself (code and data)
  – the OS’s process module manages these processes
    • creation, destruction, scheduling, …
Program/processor/process

- Note that a **program** is totally passive
  - just bytes on a disk that encode instructions to be run
- A **process** is an instance of a program being executed by a (real or virtual) processor
  - at any instant, there may be many processes running copies of the same program (e.g., an editor); each process is separate and (usually) independent
  - Use TASKMGR to list all processes

<table>
<thead>
<tr>
<th>Process A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code:</td>
</tr>
<tr>
<td>Stack:</td>
</tr>
<tr>
<td>PC:</td>
</tr>
<tr>
<td>Register:</td>
</tr>
<tr>
<td>List:</td>
</tr>
<tr>
<td>Error:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code:</td>
</tr>
<tr>
<td>Stack:</td>
</tr>
<tr>
<td>PC:</td>
</tr>
<tr>
<td>Register:</td>
</tr>
<tr>
<td>List:</td>
</tr>
<tr>
<td>Error:</td>
</tr>
</tbody>
</table>
States of a user process

(but wait until we talk about threads...)

- Running
- Ready
- Blocked
- Dispatch
- Interrupt
- Exception
Process operations

- The OS provides the following kinds operations on processes (i.e., the process abstraction interface):
  - create a process
  - delete a process
  - suspend a process
  - resume a process
  - clone a process
  - inter-process communication
  - inter-process synchronization
  - create/delete a child process (subprocess)
Memory management

• The primary memory (or RAM) is the directly accessed storage for the CPU
  – programs must be stored in memory to execute
  – memory access is fast (e.g., 60 ns to load/store)
    • but most 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
    • through page tables (hardware support feature)
  – decide how much memory to allocate to each process
    • a policy decision
  – decide when to remove a process from memory
    • also policy
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 (user or system) and 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, how to request I/O, how to handle interrupts or errors
    • examples: SCSI device drivers, Ethernet card drivers, video card drivers, sound card drivers, …
• **Note:** Windows has ~35,000 device drivers!
Secondary storage

• Secondary storage (disk, tape) is persistent memory
  – often magnetic media, survives power failures (hopefully)
  – This can be both good and bad

• 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
  – disk controllers are continually getting smarter

• 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 crude and awkward
  – e.g., “write 4096 byte block to sector 12”
• File system: a convenient abstraction
  – defines logical objects like files and directories
    • hides details about where on disk files live
  – as well as operations on objects like read and write
    • read/write byte ranges instead of blocks
• A file is the basic unit of long-term storage
  – file = named collection of persistent information
• A directory is just a special kind of file
  – directory = named file that contains names of other files
    and metadata about those files (e.g., file size)
• Note: Sequential byte stream is only one possibility!
File system operations

- The file system interface defines standard operations:
  - file (or directory) creation and deletion
  - manipulation of files and directories (read, write, extend, rename, protect)
  - copy
  - lock

- File systems also provide higher level services
  - accounting and quotas
  - (sometimes) backup
  - (sometimes) indexing or search
  - (sometimes) file versioning
Protection

• Protection is a general mechanism used throughout the OS
  – all resources needed to be protected
    • memory
    • processes
    • files
    • devices
    • CPU time
    • ...
  – protection mechanisms help to detect and contain unintentional errors, as well as preventing malicious destruction
Command interpreter (shell)

• A particular program that handles the interpretation of users’ commands and helps to manage processes
  – user input may be from keyboard (command-line interface), from script files, or from the mouse (GUIs)
  – allows users to launch and control new programs
• On some systems, command interpreter may be a standard part of the OS (e.g., MS DOS, Apple II)
• On others, it’s just non-privileged code that provides an interface to the user
  – e.g., bash/csh/tcsh/zsh on UNIX
• On others, there may be no command language
  – e.g., MacOS
Accounting

• Keeps track of resource usage
  – both to enforce quotas
    • “you’re over your disk space limit”
  – or to produce bills
    • timeshared computers like mainframes
    • hosted services
OS structure

- It’s not always clear how to stitch OS modules together:
OS structure

• An OS consists of all of these components, plus:
  – many other components
  – system programs (privileged and non-privileged)
    • e.g., bootstrap code, the init program, …

• Major issue:
  – how do we organize all this?
  – what are all of the code modules, and where do they exist?
  – how do they cooperate?

• Massive software engineering and design problem
  – design a large, complex program that:
    • performs well, is reliable, is extensible, is backwards compatible, …
Early structure: Monolithic

• Traditionally, OS’s (like UNIX) were built as a monolithic entity:

```
user programs

OS  everything

hardware
```
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

• What is the alternative?
  – find a way to organize the OS in order to simplify its design and implementation
Layering

• The traditional approach is layering
  – implement OS as a set of layers
  – each layer presents an enhanced ‘virtual machine’ to the layer above

• The first description of this approach was Dijkstra’s THE system
  – Layer 5: Job Managers (Execute users’ programs)
  – Layer 4: Device Managers (Handle devices and provide buffering)
  – Layer 3: Console Manager (Implements virtual consoles)
  – Layer 2: Page Manager (Implements virtual memories for each process)
  – Layer 1: Kernel (Implements a virtual processor for each process)
  – Layer 0: Hardware
Problems with layering

• Imposes hierarchical structure
  – but real systems are more complex:
    • file system requires VM services (buffers)
    • VM would like to use files for its backing store
  – strict layering isn’t flexible enough

• Poor performance
  – each layer crossing has overhead associated with it

• Disjunction between model and reality
  – systems modeled as layers, but not really built that way
Hardware Abstraction Layer

• An example of layering in modern operating systems
• Goal: separates hardware-specific routines from the “core” OS
  – Provides portability
  – Improves readability
Microkernels

• Popular in the late 80’s, early 90’s
  – recent resurgence of popularity
• 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)
  – Follow-ons: Mach (CMU), Chorus (French UNIX-like OS), OS X (Apple), in some opinions Windows NT (Microsoft)
• Windows NT (aka XP/Vista/Win7) designed as microkernel but executed as single kernel-mode image
Microkernel System Structure

user processes

system processes

user mode

user mode

kernel mode

microkernel

low-level VM

communication

hardware

high-level scheduling

network support

external paging

thread system

file system

low-level VM

protection

processor control

kernel mode
Summary and Next Time

• Summary
  – OS design has been an evolutionary process of trial and error. Probably more error than success
  – Successful OS’s designs have run the spectrum from monolithic, to layered, to micro kernels, to virtual machines
  – The role and design of an OS is still evolving
  – It is impossible to pick one “correct” way to structure an OS

• Next Time
  – Processes, one of the most fundamental pieces in an OS
  – What is a process, what does it do, and how does it do it