CSE 451: Operating Systems
Winter 2003

Lecture 3
Operating System Structure:
Components and Interfaces

Hank Levy
levy@cs.washington.edu
412 Sieg Hall

OS Structure

- We've described how an OS is sandwiched between the hardware and user-level processes
  - and in the previous class, we talked about the relationship between OS and hardware
  - this class: look at the functionality that the OS provides to the user-level, and the components that provide it
  - later in the course: explore each component in detail
- A useful taxonomy:
  - modules: the major OS services and abstractions
  - interfaces: specific operations that modules provide
  - structure: the way the modules are stitched together

© 2003 Hank Levy
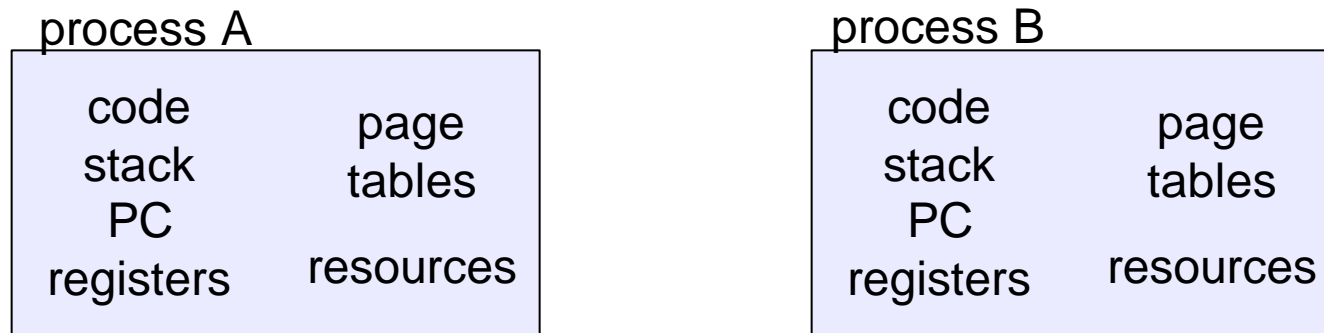
Major OS components

- processes
- memory
- I/O
- secondary storage
- file systems
- protection
- accounting
- shells (command interpreter, or OS UI)

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, …

© 2003 Hank Levy

Processes

- Note that a program is totally passive
  - just bytes on a disk that contain instructions to be run
- A process is an instance of a program being executed
  - at any instance, there may be many processes running copies of the same program (e.g. an editor); each process is separate and (usually) independent
  - Linux: `ps -auwwx` to list all processes

process A

code stack PC registers	page tables resources

process B

code stack PC registers	page tables resources

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 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
  - 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, …

© 2003 Hank Levy

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

© 2003 Hank Levy

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 long-term storage unit
  - 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)

© 2003 Hank Levy

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
  - backup
  - (sometimes) indexing or search
  - (sometimes) file versioning

© 2003 Hank Levy

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

© 2003 Hank Levy

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. MSDOS, Apple II)
- on others, it's just a non-privileged process 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
    - important for timeshared computers like mainframes

OS Structure

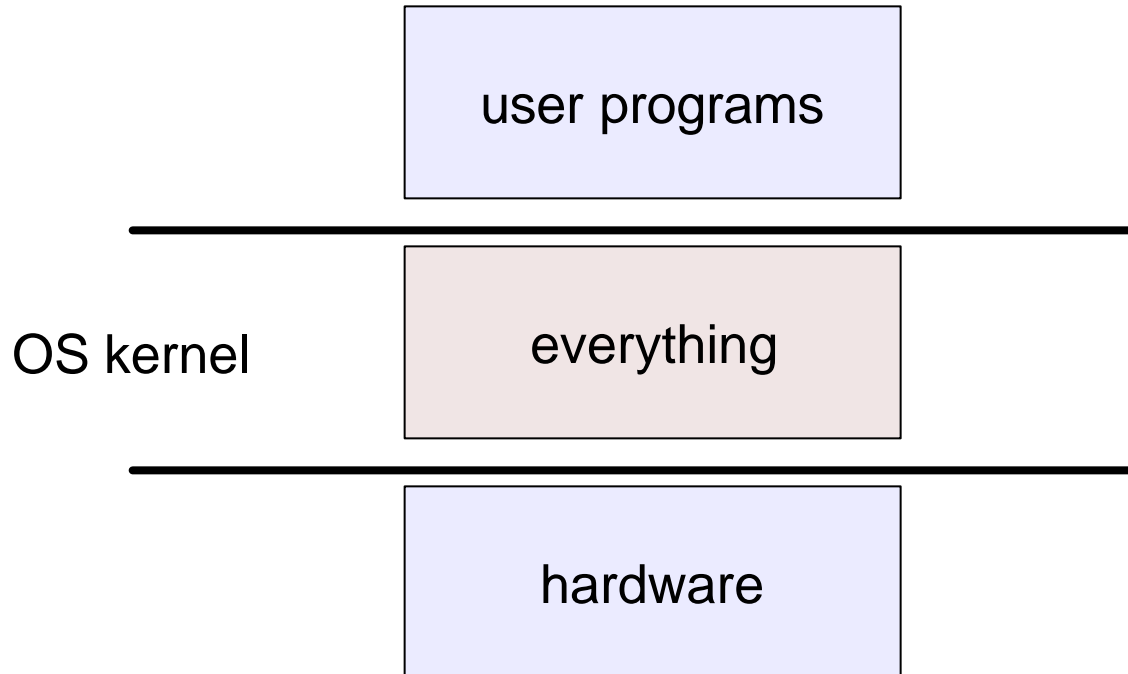- It's not always clear how to stitch OS modules together:
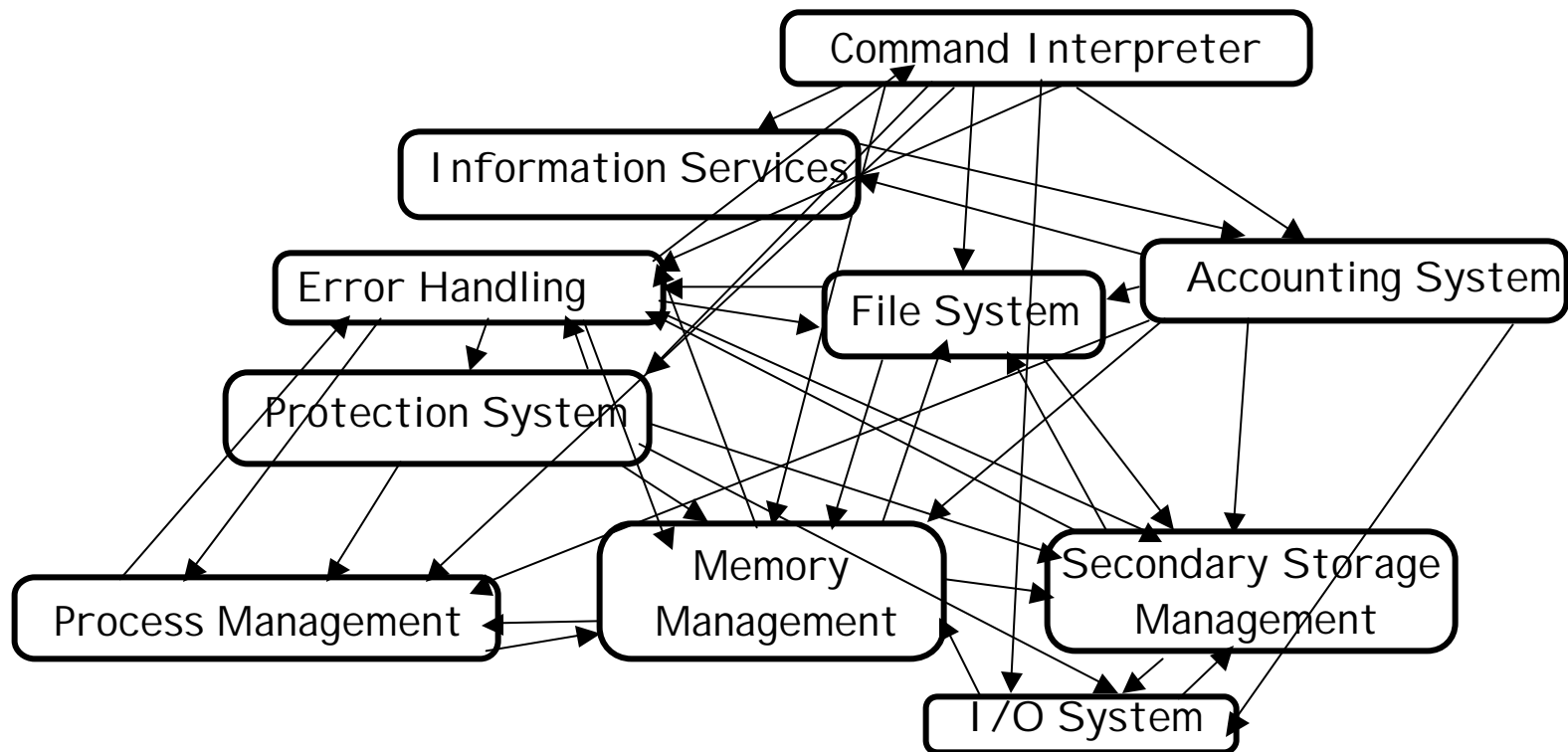
OS Structure

- 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, …

© 2003 Hank Levy

Early structure

- Traditionally, OS's (like UNIX) were built as a monolithic kernel:

```
        ┌──────────────────┐
        │  user programs   │
        └──────────────────┘
    ────────────────────────────
        ┌──────────────────┐
OS kernel │    everything   │
        └──────────────────┘
    ────────────────────────────
        ┌──────────────────┐
        │    hardware      │
        └──────────────────┘
```

Monolithic kernels

- 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 acts as a 'virtual machine' to the layer above
- The first description of this approach was Dijkstra's THE system
  - layer 0: hardware
  - layer 1: CPU scheduling
  - layer 2: memory management (sees virtual processors)
  - layer 3: console device (sees VM segments)
  - layer 4: I/O device buffering (sees 'virtual console')
  - layer 5: user programs (sees 'virtual I/O drivers')
- Each layer can be tested and verified independently

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

Microkernels

- Popular in the late 80's, early 90's
  - recent resurgence of popularity for small devices
- 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), and in some ways NT (Microsoft), OSX (Apple)

Microkernel structure illustrated

user processes

netscape   powerpoint

apache

system processes

file system   network

paging

threads   scheduling

user mode

microkernel

communication

low-level VM   processor control

protection

kernel

hardware