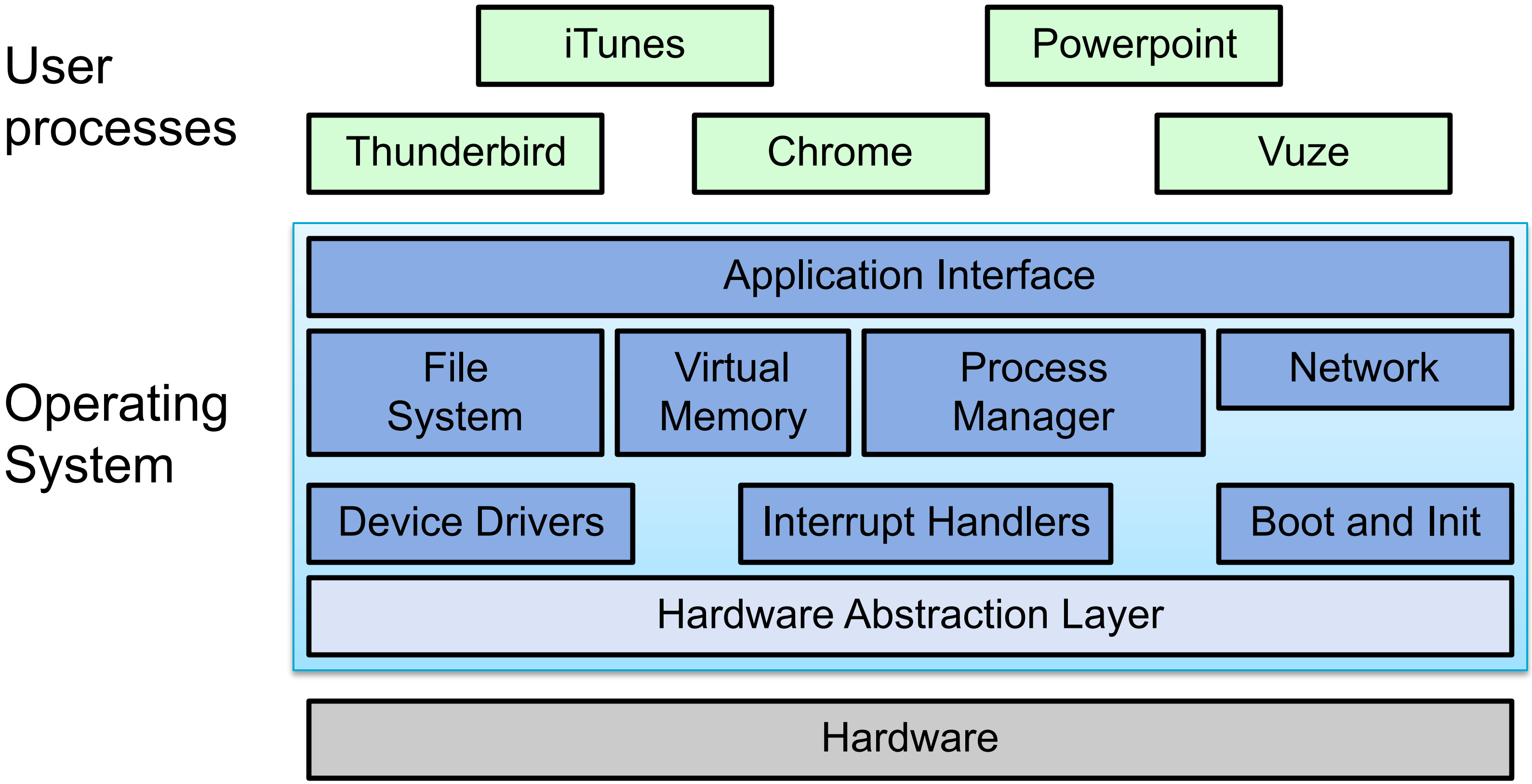
OS concepts and structure

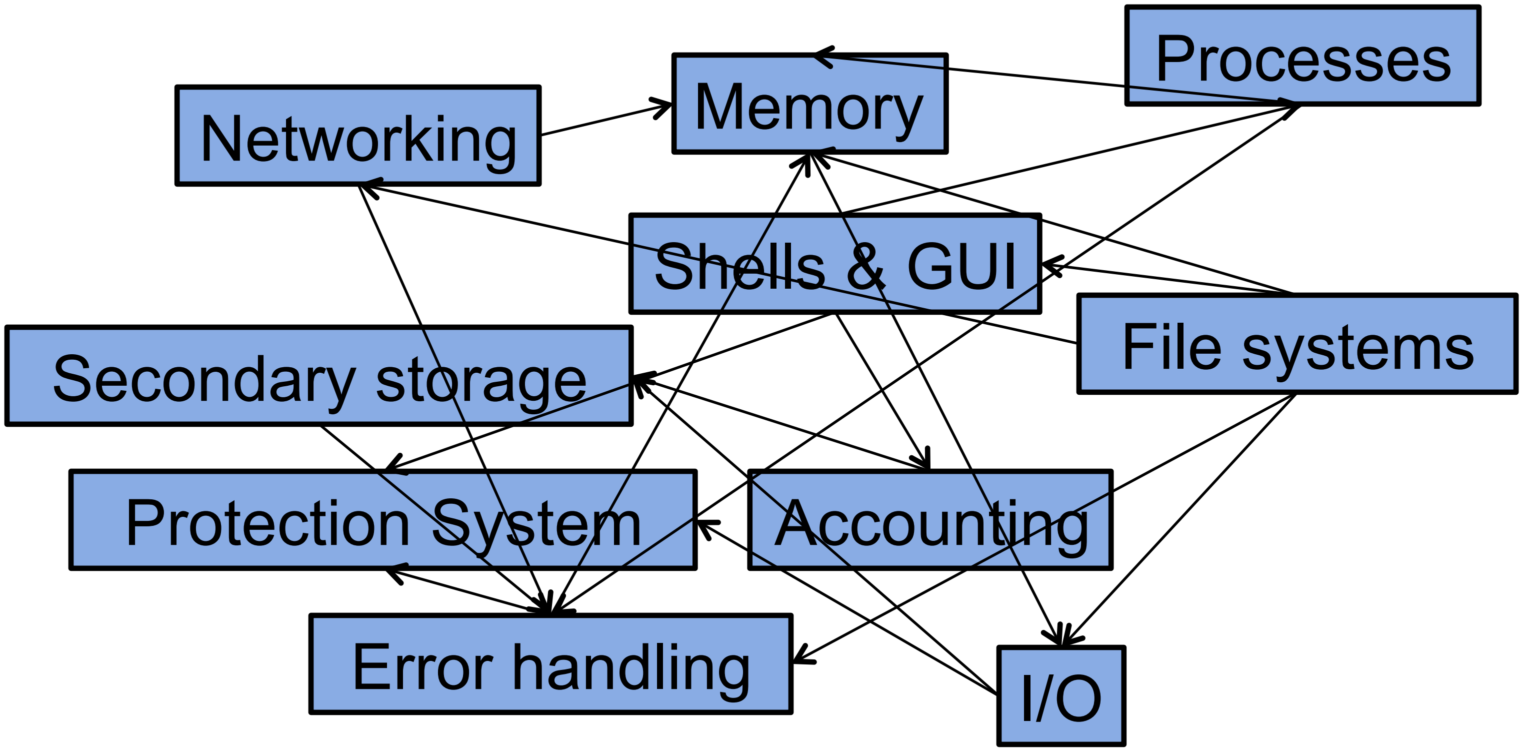
To do ...

- OS components & interconnects
- Structuring OSs
- Next time: Processes

Between hardware and your apps



Major OS components & abstractions



OS Views

- Perspectives, “OS as …
  - … the services it provides
  - … its components and interactions

- Services to …
  - Users via a GUI or a command interpreter/shell
  - Programmers via system calls

  - Some services are for convenience: FS management
  - Some to ensure efficient operation: Resource allocation

GUI or command interpreter (shell)

- GUI
  - Friendlier (desktop), if sometimes limiting
  - Xerox PARK Alto >> Apple >> Windows >> Linux

- Command interpreter
  - Handle (interpret and execute) user commands
  - Could be part of the OS: MS DOS, Apple II
  - Or just a special program: UNIX, Win XP
  - The command interpreter could
    - Implement all commands
    - Simply understand what program to invoke and how (UNIX)

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`

Major OS components & abstractions

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

Processes

- An OS executes many kind of activities
  - Each encapsulated in a process
- A program in execution
  - Address space, set of registers, OS resources
  - Threads and processes – for now consider each process to have a single thread (we'll revisit this later)
- 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 interface offered by the OS?

Memory management

- Main memory – directly accessed for CPU
  – Programs must be in memory to execute
  – Memory access is fast (e.g., 60 ns to load/store), but not persistent (won't survive power failures)

- OS must
  – Allocate memory space to processes
  – Decide how to allocate it to each process
  – Deallocate it when needed
  – Maintain mappings from physical to virtual memory
  – Decide when to remove a process from memory

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)
- Code interacting with disks are 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
  - FS knowledge of device details can help with performance
    - e.g., place related files close together on disk

File systems

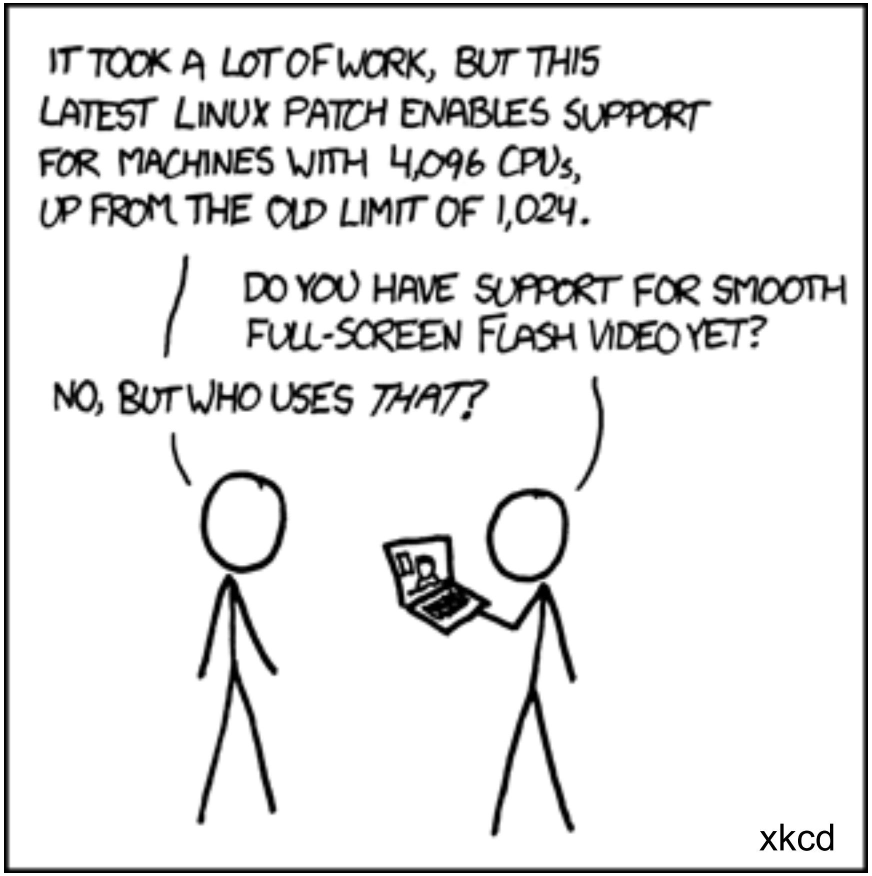
- Storage devices are hard to work with
  - File system offers a convenient abstraction
  - Defines logical abstractions/objects like files & directories
  - As well as operations on these objects
- A file is the basic unit of long-term storage
- A directory is just a special kind of file
  - … containing 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 must be protected
    - memory
    - processes
    - files
    - devices
    - …
- Mechanisms help detect and contain errors, and preventing malicious destruction

And now a short break ...

Supported Features



OS design & implementation

- A challenge …
  - Conflicting user and system goals
    - User – Convenient, easy to learn, reliable, safe, fast
    - System – Easy to design, implement, and maintain, flexible, reliable, error-free and efficient
  - Dealing with concurrency – users and devices
  - Some hostile users, others who want to collaborate
  - Long expected lives (Unix is ~40y, Windows ~ 30y) & no clear ideas on future needs
  - Portability and support for 1,000s of devices
  - 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

- 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 – Size, speed?, but who cares?!

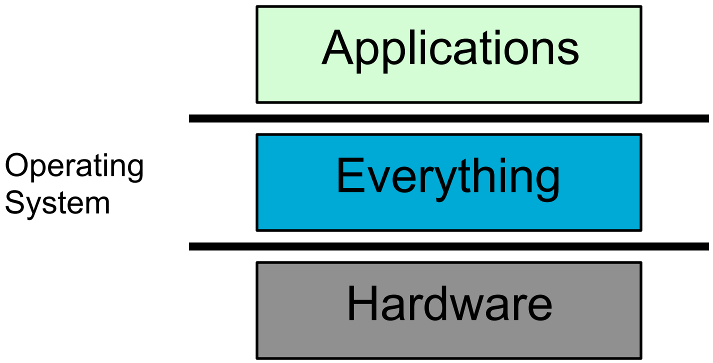
OS structure

- OS made of number of components
  - Process, memory managemetn, …
  - 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?

    http://www.makelinux.net/kernel_map/

Monolithic design

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



Layering – Traditional approach

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

Layer	Function
5	The operator
4	User programs
3	I/O management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Dijkstra’s THE system

Problems with layering

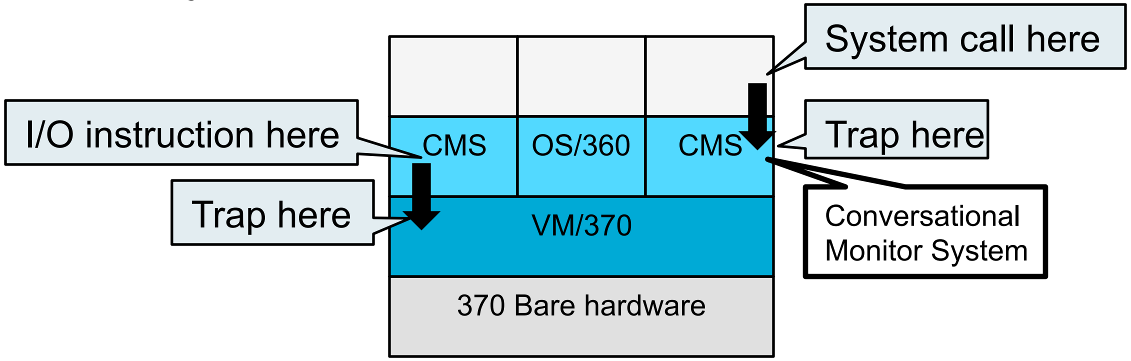
- Imposes hierarchical structure
  – but real systems have complex interactions
  – Strict layering isn't flexible enough
- Poor performance
  – Each layer crossing has an associated overhead
- Disjunction between model and reality
  – Systems modelled as layers, but not built that way

HAL – Hardware Abstraction Layer

- An example of layering in modern OSs
- Goal – to hide differences in hardware from most of the OS kernel
  - On a PC, you can consider it as the driver of the motherboard
  - BSD, Mac OS X, Windows NT, Linux, NetBSD all use a HAL either explicitly identified or not

Virtual machines

- Initial release of OS/360 were strictly batch but users wanted timesharing
  - IBM CP/CMS, later renamed VM/370 (‘79)
- Timesharing systems provides
  - Multiprogramming & Extended (virtual) machine
- Essence of VM/370 – separate the two
  - Heart of the system (VMM) does multiprogramming & provides multiple exact copies of bare HW to next layer up
  - Each VM can run any OS



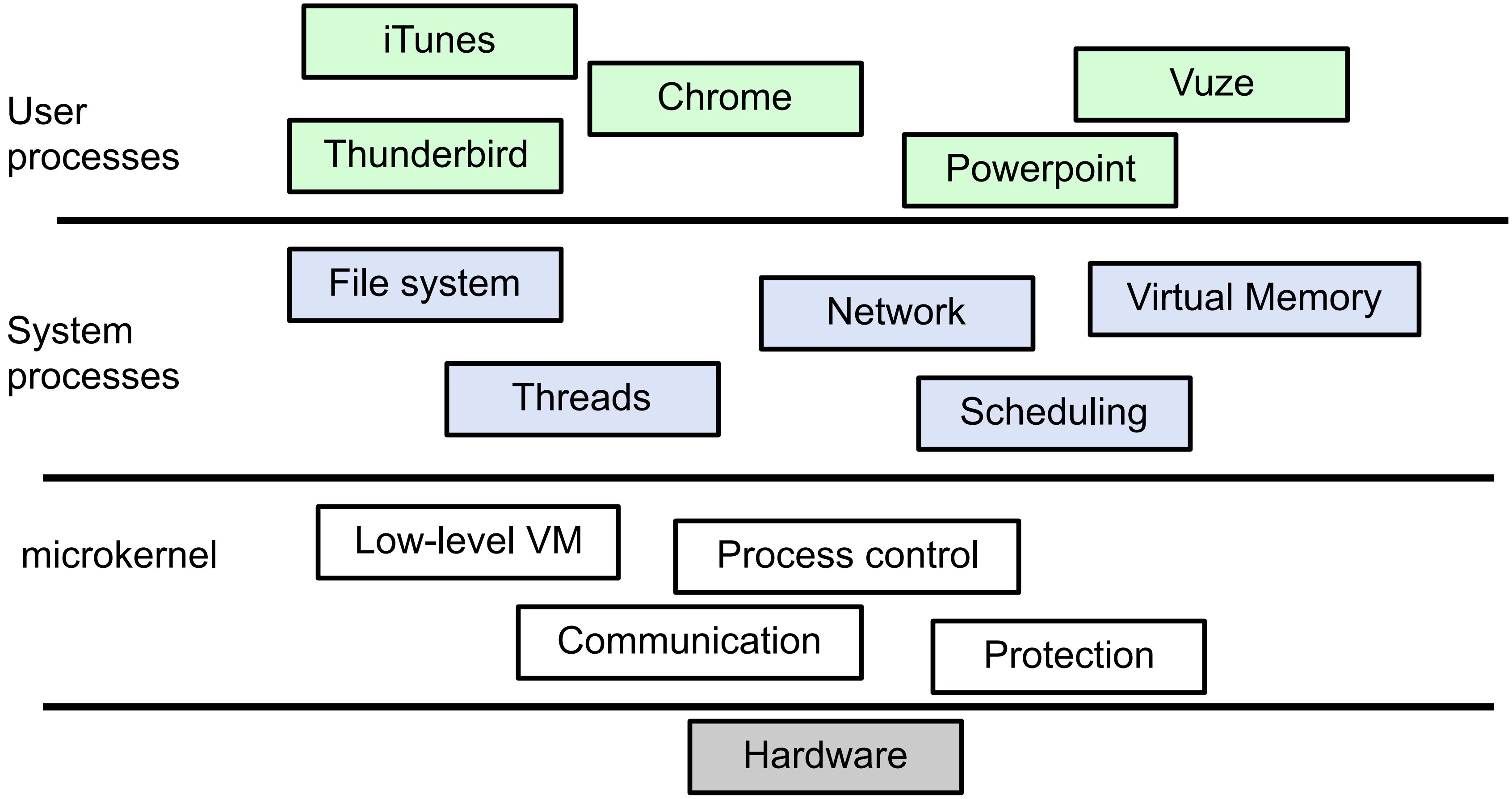
Virtual machines

- A resurgence in mid-90s, started with Rosenblum’s work on Disco and VMWare
  - Nowadays … Java VM, Xen, VritulaBox, Virtual Iron, VMLite, Simics, Parallels, Palacios, QEMU, …
- What for?
  - Server consolidation – from different services in different lightly used machine (administration cost)
  - Different applications for other OS in your desktop
  - Testing and debugging

Microkernels

- Popular in the late 80’s, early 90’s
  - Recent resurgence
- Goal: Minimal kernel, the rest in user-level
- What for?
  - Better reliability (isolation between components)
  - Ease of extension and customization
  - Poor performance (user/kernel boundary crossings)
- First microkernel Hydra (CMU, 1970)
  - … Mach (CMU), Chorus (UNIX-like), OS X (Apple), in some ways NT (Microsoft), L4 (Karlsruhe), MINIX 3, …

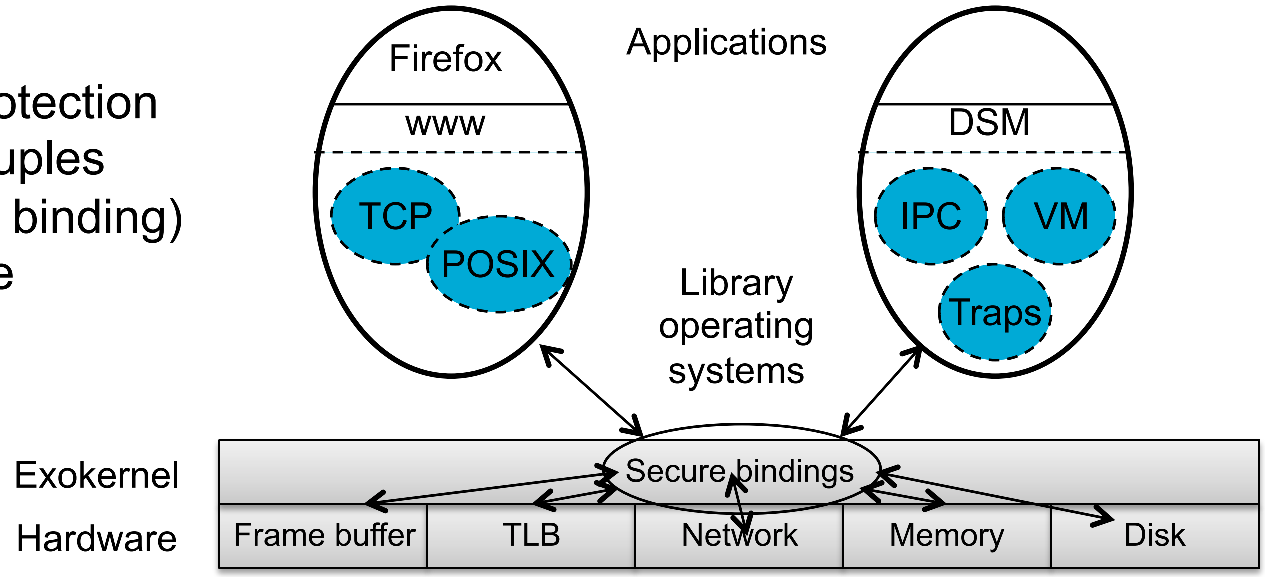
Microkernel



Exokernels

- OS, typically securely multiplexes & *abstract* physical resources
- But no OS abstractions fits all!
- Exokernel
  - A minimal OS securely multiplexes resources
  - Library OSes implement higher-level abstractions

Secure binding – a protection mechanism that decouples authorization (done at binding) from use of a resource



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