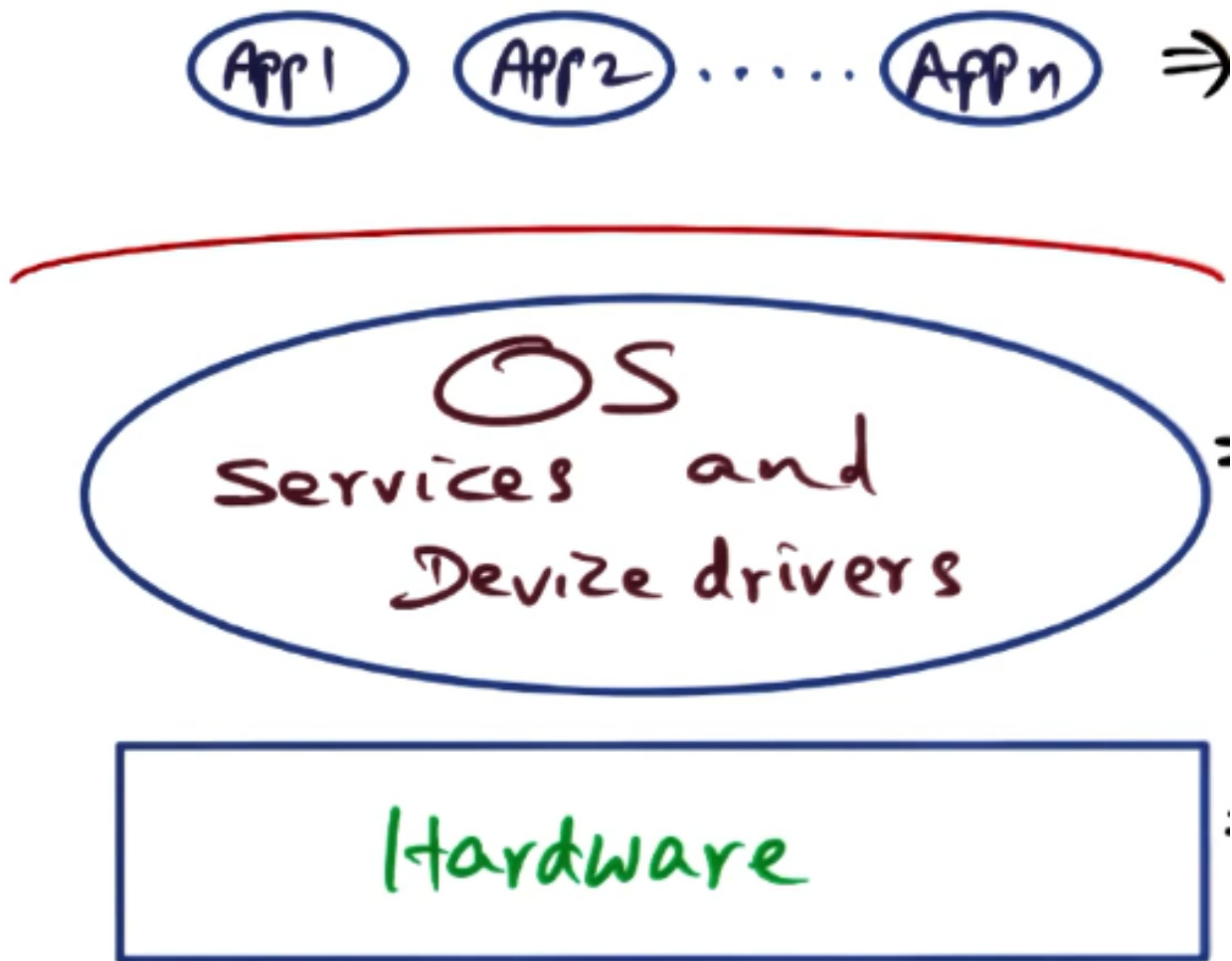


# OS Structure

## OS Structure Overview

- \* Operating System Services
  1. Process/thread management and scheduling
  2. Memory management (protection, sharing, demand paging)
  3. Interprocess communication (IPC)
  4. File system
  5. Access to I/O devices such as microphones and speakers
  6. Access to the network
- \* OS Structure: The way the operating system software is organized with respect to the applications that it serves and the underlying hardware that it manages
- \* Goals of OS Structure
  1. Protection: Within and across users and to the OS itself
  2. Performance: Time taken to perform the services
  3. Flexibility: Extensibility (not one size fits all)
  4. Scalability: Performance improves with more hardware resources
  5. Agility: Adapting to changes in application needs and/or resource availability
  6. Responsiveness: Reacting to external events
- \* Commercial OS: Don't meet all goals laid out above
  - Linux
  - MacOS
  - Windows
- \* Monolithic Structure
  - Each app has its own hardware address space
  - OS also has its own hardware address space (protected from apps)
  - Hardware is managed by the OS
  - OS services and device drivers are all contained in one blob
  - Reduces performance loss by consolidation
    - + All of the components are aware of each other and interactions can be optimized
  - Con: Less extensible (no customization, useful for different apps)
    - + Video games: Responsiveness is desired
    - + Computing prime numbers: Sustained CPU time is desired



Monolithic OS Structure

---

\* DOS-like Structure

- Disk operating system
- No protection layer between the application and the OS
- Pro: Improved performance (access to system services is like a procedure call)
- Con: Decreased protection (an errant application can corrupt the OS)
- In the early days of computing, only one process could run at a time
  - + Simplicity and performance took precedence
  - + No overhead for system calls vs developer procedures
- Loss of protection is unacceptable for a general-purpose OS

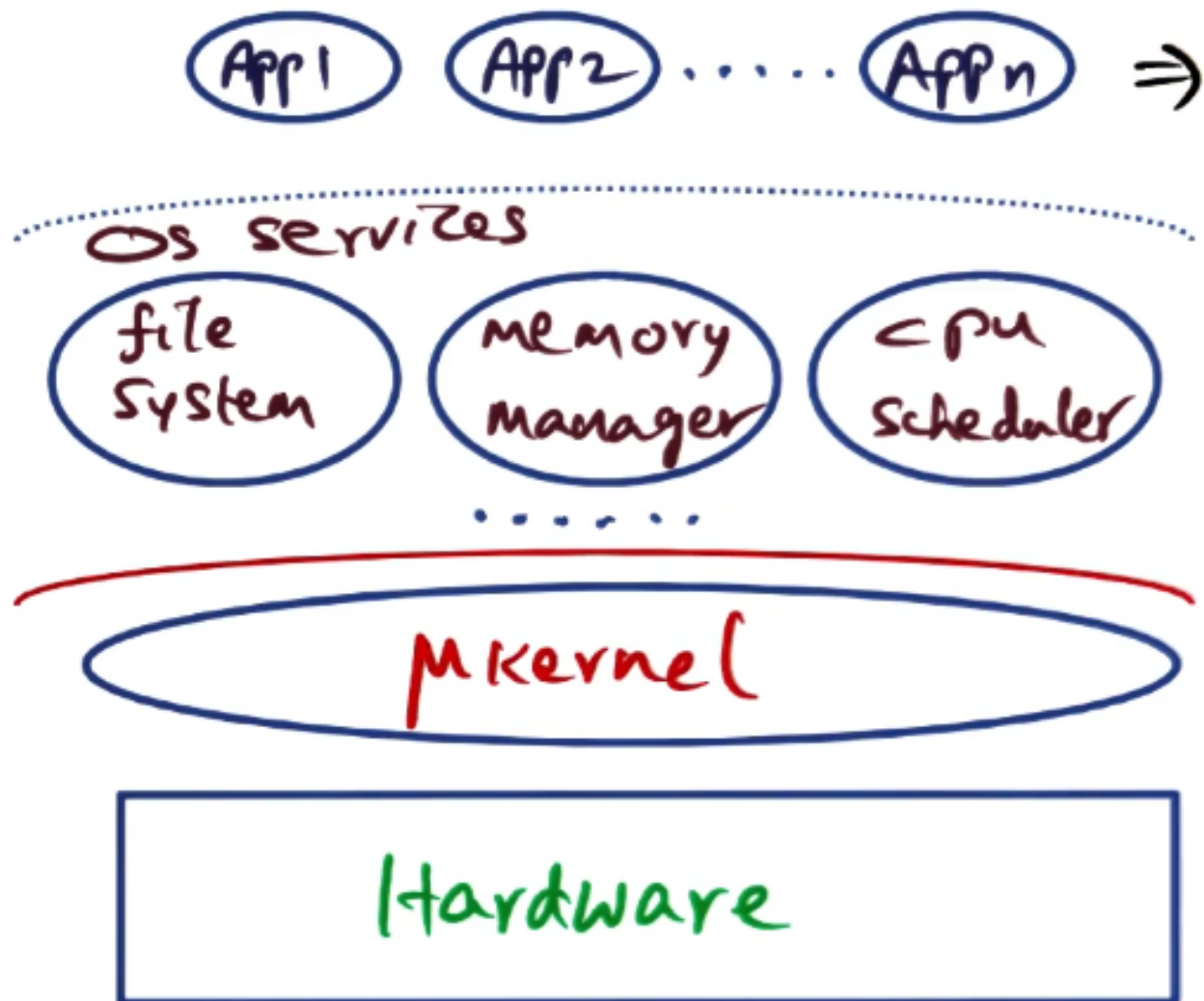
\* Opportunities for Customization

- Memory management (page fault)
  1. OS finds a free page frame (page replacement algorithm)
  2. OS updates page table for faulty process
  3. Resume process
- The page replacement algorithm can't be ideal for every application
- Can customize how page replacement is handled

\* Microkernel Structure

- Each app in its own hardware address space

- Microkernel runs in privileged mode and provides simple mechanisms
  - + Threads
  - + Address space
  - + IPC
- OS services are implemented on top of microkernel (same privilege as applications)
- No distinction between OS services and applications in terms of how they run
- Pro: Extensibility; services can be easily replaced
- Con: Performance; in a monolithic structure, filesystem and memory management are all within the kernel. In a microkernel structure, an app might go through the kernel, to the filesystem service, back to the microkernel, and back to the app. All of this is IPC overhead.
- Border crossings
  - + User space  $\leftrightarrow$  system space copying
  - + Explicit costs in changing context
  - + Implicit costs due to cache misses (change in locality)



Microkernel OS Structure

---

Feature	Monolithic	DOS-like	Microkernel
Extensibility		X	X
Protection	X		X
Performance	X	X	

## The SPIN Approach

### \* Two Premises:

1. Microkernel design compromises on performance due to frequent border crossings
2. Monolithic design does not lend itself to extensibility

### \* Goals

1. Thin (like microkernel) -> only mechanisms, not policies
2. Access to resources without border crossing (like DOS)
3. Flexibility for resource management (like microkernel) without sacrificing protection and performance (like monolithic)

### \* Approaches to Extensibility

- Hydra OS
  - + Kernel mechanisms for resource allocation
  - + Capability-based resource access
  - + Resource management as coarse-grained objects to reduce border crossing overhead
- Mach OS
  - + Microkernel-based
  - + Focused on extensibility and portability
  - + Performance wasn't prioritized

### \* SPIN's Approach to Extensibility

- Co-location of kernel and extensions to avoid border crossings
- Compiler enforced modularity (strongly typed language, Modula)
- Logical protection domains (not hardware address spaces)
- Dynamic call binding -> flexibility
- Make extensions as cheap as a procedure call

### \* Logical Protection Domains

- Modula-3 safety and encapsulation mechanisms
  - + Type safety, automatic storage management
  - + Objects, threads, exceptions, generic interfaces
- Fine-grained protection via capabilities
  - + Hardware resources (page frame)
  - + Interfaces (page allocation module)
  - + Collection of interfaces (entire virtual memory subsystem)
- Capabilities implemented as language supported pointers
  - + Modula-3 pointers are type-specific (no casting)

### \* SPIN Mechanisms for Protection Domains

- Create: initialize with object file contents and export names that are contained as entry point methods into the object to be externally visible
- Resolve: Names between source and target domains
  - + Once resolved, resource sharing at memory speeds
- Combine: To create an aggregate domain from smaller domains

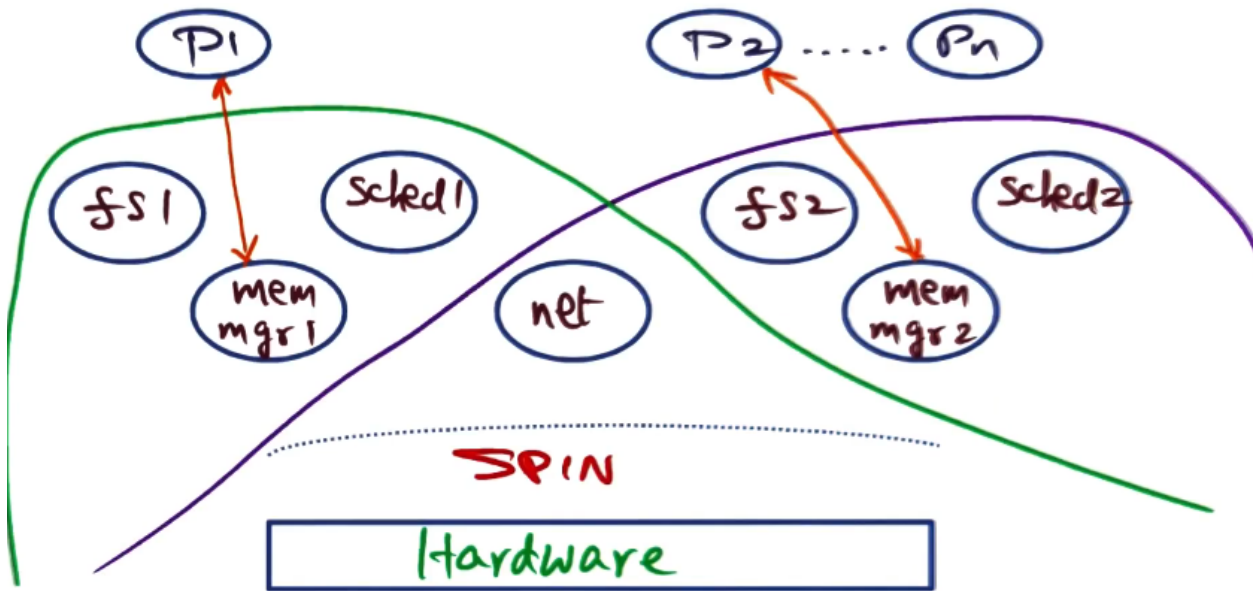
### \* Customized OS with SPIN

- No border crossing between services and mechanisms provided by SPIN

---

## Customized OS with SPIN

Create, Resolve, Combine



SPIN OS Structure

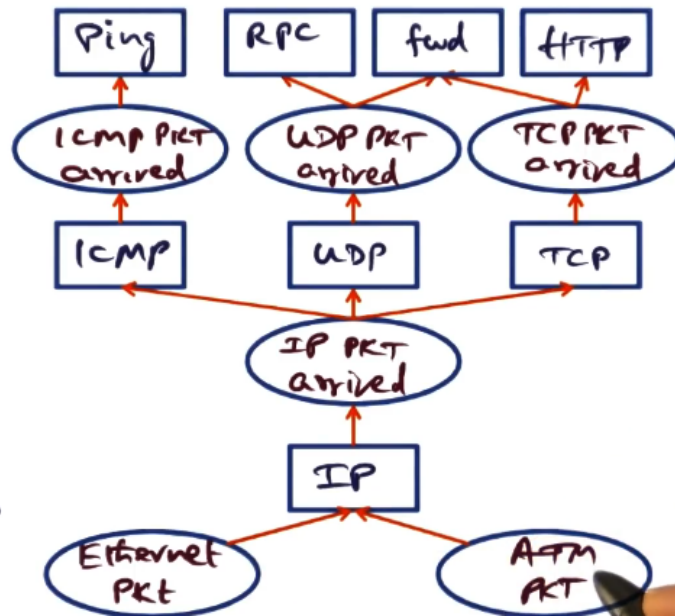
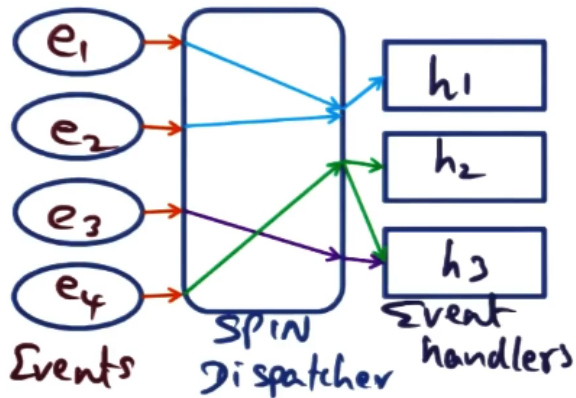
---

### \* SPIN Mechanisms for Events

- OS needs to be able to field events (interrupts, system calls)
- Event-based communication model
  - + Services register event handlers (one-to-one, one-to-many, many-to-one)

## SPIN Mechanisms for Events

### Event-based Communication model



SPIN Event Handlers

#### \* Default Core Services in SPIN

- SPIN provides interface procedures for implementing these services
- Automatically invoked when the hardware event occurs
- Memory management
  - + Physical address (allocate, deallocate, reclaim)
  - + Virtual address (allocate, deallocate)
  - + Translation (create/destroy address space, add/remove mapping)
  - + Event handlers (page fault, access fault, bad address)
- CPU Scheduling
  - + SPIN abstraction: strand (semantics defined by extension)
  - + Event handlers (block, unblock, checkpoint, resume)
  - + SPIN global scheduler (interacts with application threads package)
- Core services are trusted because they provide access to hardware mechanisms
  - + Services may need to step outside the language-enforced protection model to control the resources
  - + Applications that run on top of extensions must trust extension
  - + However, these are isolated (don't impact things that don't rely on the extension)

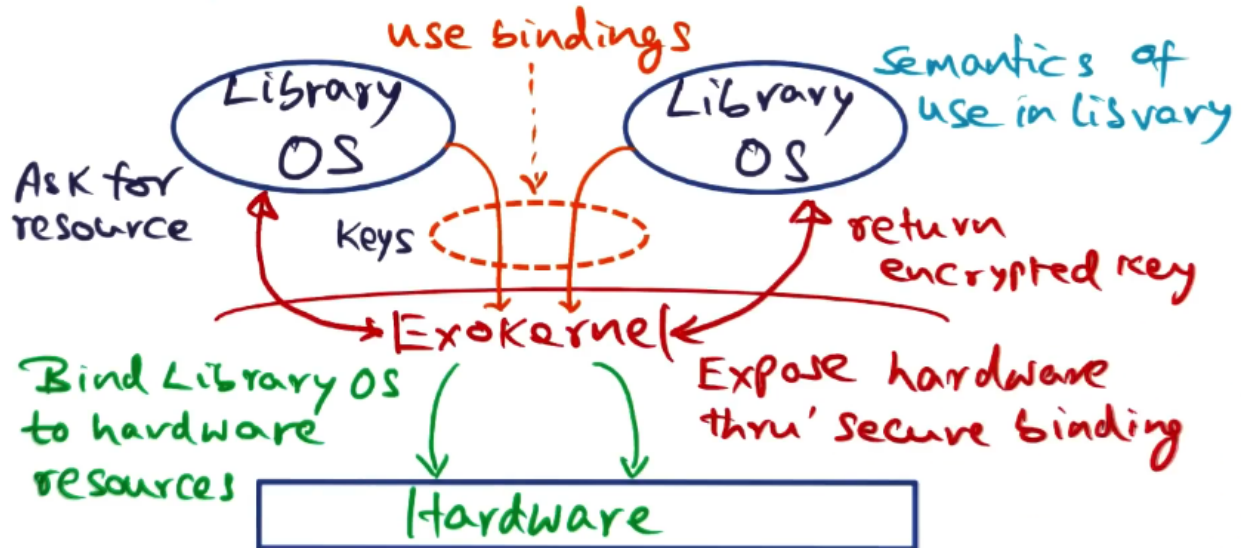
### The Exokernel Approach

#### \* Exokernel: Kernel exposes hardware explicitly to OS extensions

- Decouple authorization of hardware from actual use
- Expose hardware through secure binding (binds library OS to hardware resources)
- Gives encrypted key to library OS; semantics of use are up to library
  - + Can't be forged or transferred
- Library gives key to exokernel to validate it can use the resource

# Exokernel Approach to Extensibility

## Decouple Authorization from use



Exokernel OS Structure

### \* Examples of Candidate Resources

#### - TLB Entry

- + Virtual to physical mapping done by library
- + Binding presented to exokernel
- + Exokernel puts it into hardware TLB
- + Process in library OS uses multiple time without exokernel intervention

#### - Packet filter

- + Predicates loaded into kernel by library OS
- + Check on packet arrival by Exokernel

### \* Implementing Secure Bindings

1. Hardware mechanisms (TLB entry, physical page frame, frame buffer)
2. Software caching
  - + "Shadow" TLB in software for each library OS
  - + Exokernel dumps hardware TLB into software TLB data structure on context switch
3. Downloading code into kernel
  - + Functionally equivalent to SPIN extensions
  - + Compromises protection more than SPIN; as long as SPIN's logical protection domains are following Modula-3's compile and runtime verification, no compromise. Exokernel doesn't make this guarantee

### \* Memory Management in Exokernel

- When a thread incurs a page fault, it's caught by Exokernel
- Then, it's passed to the library OS through the registered handler
- Library services fault (might require requesting a page frame from Exokernel)

### \* Memory Management Using Software TLB

- On context switch, much of performance hit comes from loss of

cache locality

- To mitigate this overhead, Exokernel implements a software TLB
- Software TLB: Data structure representing a snapshot of the hardware TLB; captured on context switch for the library OS
- On context switch, Exokernel preloads TLB with software TLB

---

## Default Core Services in Exokernel

### Memory management



---

### Exokernel Memory Management

---

#### \* CPU Scheduling in Exokernel

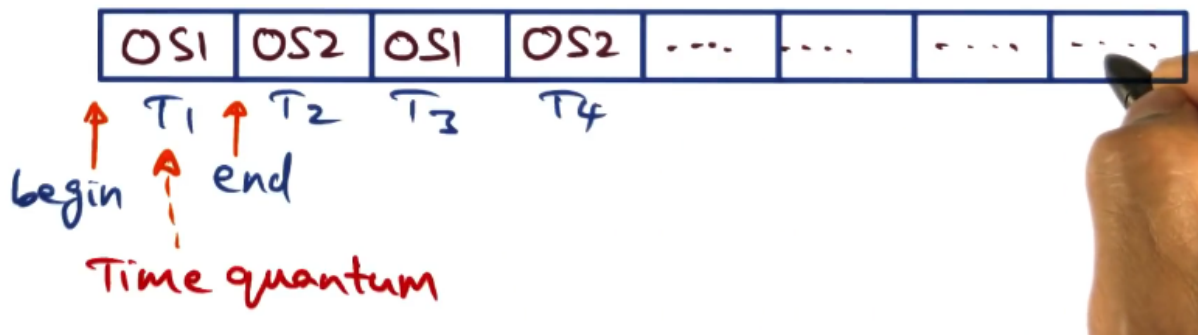
- Linear vector of "time slots"
- Each library OS will get one time quantum at a time
- When timing interrupt occurs, Exokernel will switch to next library OS
- Some time between switching when library OS gets to save state
- Exokernel only interferes for page faults



# Default Core Services in Exokernel

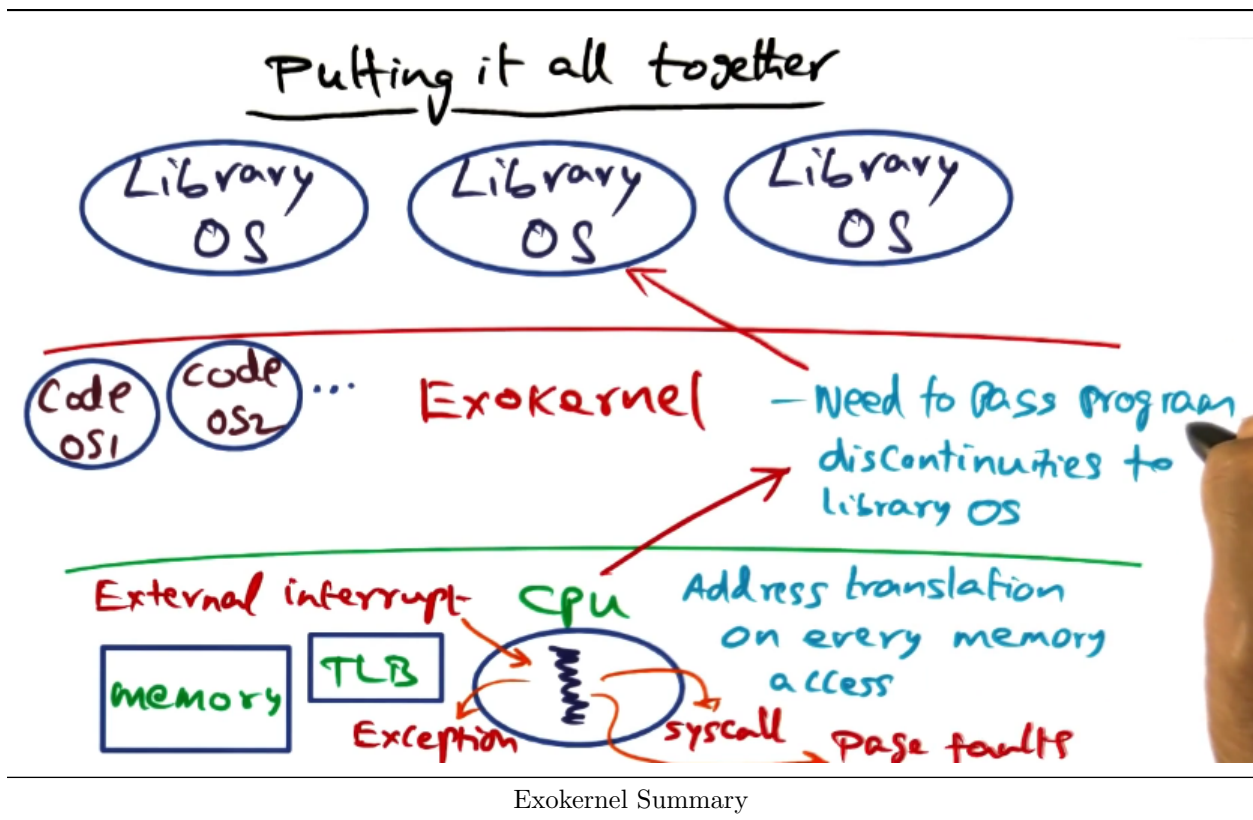
## CPU Scheduling

— Linear vector of "time slots"



### Exokernel CPU Scheduling

- \* Secure Binding - How can a library OS securely insert code into the kernel?
  - Performance optimization to avoid border crossings
  - In both SPIN and Exokernel, the ability to add extensions must be restricted to a trusted set of users
- \* Revocation of Resources
  - Exokernel needs a way of revoking resources (space/time) that have been allocated to a library OS
  - Revoke call is an upcall into library OS (repossession vector)
  - Library OS must clean up when it receives a repossession vector
    - + Stash data in a page to disk when Exokernel reclaims
  - Library can "seed" Exokernel for "autosave"
- \* Code Usage by Exokernel - Library OS gives code to run on specific events
  1. Packet filter for de-multiplexing of network packets
  2. Run code in Exokernel on behalf of library OS not currently scheduled (e.g., garbage collection for an application)
- \* Summary; Achieving extensibility, protection, and performance
  1. Performance: Exokernel allows for performance critical code in library OS to be downloaded securely into Exokernel
  2. Extensibility: Exokernel exposes hardware resources to library OS to allow them to implement functionality however is needed
  3. Protection: Exokernel provides encrypted keys to ensure that only the intended library OS can access hardware
  4. Exokernel must be involved when interrupts occur



#### \* Exokernel Data Structures

- PE data structure: Handler entry points for different event types
  - + Exceptions, interrupts, system calls, memory mappings
- Software TLB to preserve page mappings
  - + Must specify a set of guaranteed mappings for Exokernel to maintain
  - + Not all hardware TLB entries, just the specified ones
- Similar to event handler structure in SPIN OS

#### \* Performance Results of SPIN and Exokernel

- Can qualitatively argue that extensibility is provided, but must prove quantitatively that performance isn't being sacrificed
- UNIX: Monolithic
- Mach: Microkernel (CMU)
- Both SPIN and Exokernel outperform Mach regarding changing protection domains
- Both SPIN and Exokernel do as well as Unix for dealing with system calls

### The L3 Microkernel Approach

\* SPIN and Exokernel were comparing against CMU's Mach, a microkernel approach with a focus on portability

- However, can we design a microkernel with a focus on performance?

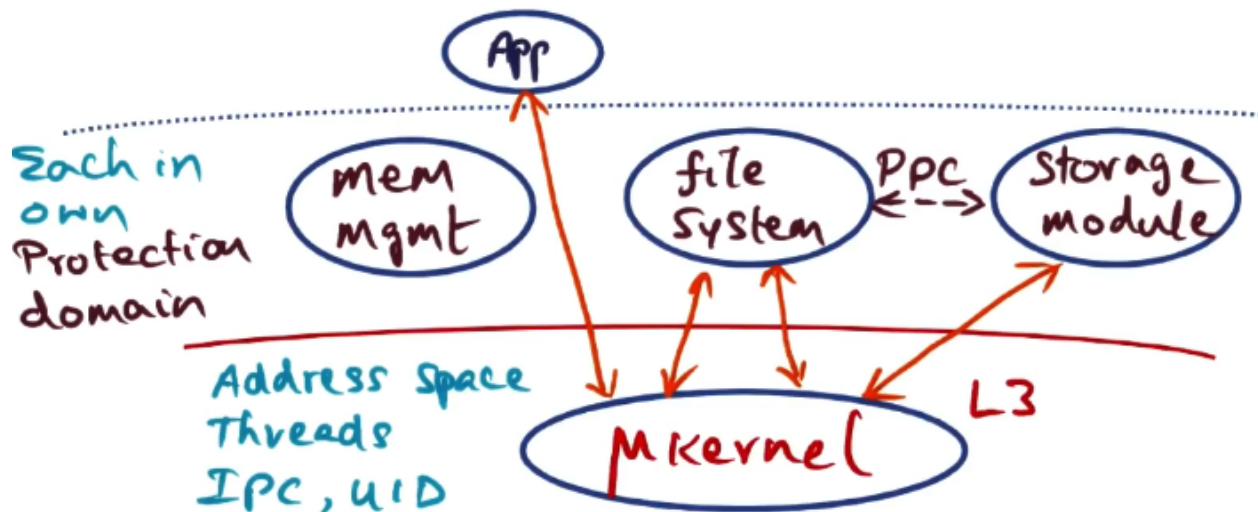
\* Microkernel: Each service in its own address space

- Applications sit on top of OS services
- OS services: Memory management, file system, storage module
- Microkernel provides simple abstractions (address space, IPC)

\* Potentials for Performance Loss

- Border crossings: Implicit and explicit costs

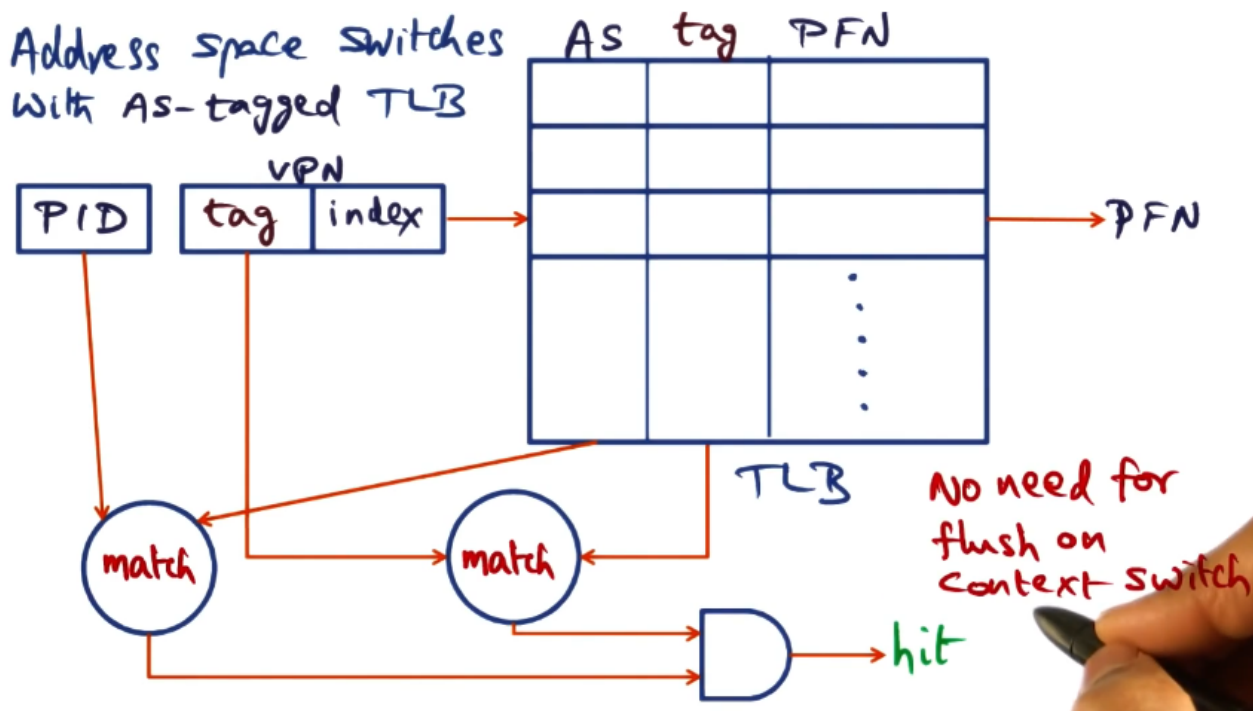
- + Applications and microkernel in different privilege levels
- + System services have to consult other services (cross address space)
- Protected procedure calls: 100x normal procedure calls
- \* L3 Microkernel
  - Aimed to debunk myths about microkernel-based OS structure
  - L3 microkernel provides address space, threads, IPC, and unique IDs
  - L3 puts each service in its own protection domain, not necessarily different address spaces
  - All about efficient implementation!



L3 OS Structure

- \* Strikes Against Microkernel
  - Kernel-user switches (border crossing cost)
  - Address space switches
    - + Basis for protected procedure calls for cross protection domain calls
    - + Crossing address spaces requires flushing the TLB
  - Thread switches and IPC
    - + Kernel mediation for protected procedure calls (PPC)
  - Memory effects
    - + Locality loss
- \* L3 Solution to User Border Crossing
  - L3 accomplishes border crossing in 123 processor cycles
    - + Includes TLB and cache misses
  - L3 calculated that the absolute minimum number of cycles on their architecture was 107; 123 is pretty good
  - Border crossing isn't inherently slower in their microkernel design
  - For reference, CMU's Mach took 900 cycles on the same architecture
- \* L3 Solution to Address Space Switches
  - Address space tagged TLBs: in addition to the tag and index, the TLB contains the process for which a particular TLB is present (MIPS)
  - This means the TLB doesn't have to be flushed on context switch
  - Liedtke's recommends exploiting architectural features if TLB is not address space-tagged
    - + Use segment registers to designate which addresses are valid for one running process (segment bounds are hardware enforced); this

- works well if no process needs the entire address space
- + Shared hardware address space for protection domains
- Large protection domain: If a process needs the entire address space and the hardware doesn't support address-tagged TLBs, a TLB flush is required
- However, this only addresses the explicit cost; loss in cache locality is significantly more expensive than the explicit cost
  - + Explicit cost  $\ll$  implicit cost
  - + TLB flush takes 864 cycles in Pentium processor
- Small protection domains
  - + Switches can be made efficient by careful construction
- Large protection domains
  - + Switching cost not important
  - + Cache effects and TLB effects dominate



Address Space Tagged TLB

#### \* Thread Switches and IPC

- Shown to be competitive to SPIN and Exokernel by construction
- Switch involves saving all volatile state of processor

#### \* Memory Effects

- Assumption is that loss of locality in a microkernel-based design  $\gg$  monolithic design
- Liedtke says that splitting the hardware address space into small protection domains using segment registers will result in warmer caches on context switches
  - + Each process only occupies a small memory footprint, which results in a smaller footprint in the caches as well
  - + Unavoidable if protection domains are large, regardless of how the operating system is designed

#### \* Mach's Expensive Border Crossing

- Focus on portability (can run on different processor architectures)
    - + Code bloat -> large memory footprint -> Less locality -> more cache misses -> longer latency for border crossing
  - Kernel memory footprint is the culprit, not microkernel itself
- \* Thesis of L3 for OS Structuring
1. Minimal abstractions in microkernel
    - + Address spaces, threads, IPC, unique IDs
    - + These four abstractions are needed by any subsystem that provides functionality to end users
  2. Microkernels are processor-specific in implementation (not portable)
  3. Right set of microkernel abstractions and processor-specific implementation -> efficient processor-independent abstractions at higher layers
    - + Processor-dependent kernel with processor-independent abstractions