

**cs6210**

# introduction

Advanced Operating Systems is a graduate-level course that addresses a broad range of topics in operating system design and implementation, including:

- Operating system structuring
- Synchronization, communication and scheduling in parallel systems
- Distributed systems, their communication mechanisms, distributed objects and middleware
- Failures and recovery management
- System support for Internet-scale computing

By tracing the key ideas of today's most popular systems to their origins in research, the class highlights key developments in operating system design over the last two decades and illustrates how insight has evolved to implementation.

lesson1

# **lesson1**

# abstractions

## Definitions

Term	Definition
abstraction	A well understood interface that hides all the details within a subsystem.

## Quizzes

### Which is an OS?

- Firefox
- MacOS
- Android

Firefox is a browser - not an operating system but an application. Android is a system software stack that sits on top of an operating system. **MacOS** is the only choice here that is an actual operating system.

### Which is not an abstraction?

- Door knob
- Instruction set architecture
- Number of pins coming out of a chip
- AND logic gate
- Transistor
- INT data type
- Exact location of base pads in a baseball field

A door knob is something we use to enter a room, however, we don't always know the exact inner workings of a door knob - this is an abstraction. Instruction set architecture of a processor is an abstraction - it tells us what the processor is capable of doing, not how it does it. AND logic gates are an abstraction - doesn't describe how it conducts a logic AND, just its functionality. Transistors allow us to turn it from solid state 0 to 1 - however we don't

know explicitly how it is conducting this operation. The INT data type doesn't describe to us how it is interpreted by the compiler, it just implements an integer.

The number of pins coming out of a chip is not an abstraction - very explicit. The exact locations of base pads in a baseball field explicitly describe their location, not an abstraction.

**Name as many abstractions between the two extremes.**

*electrons – > GoogleEarth*

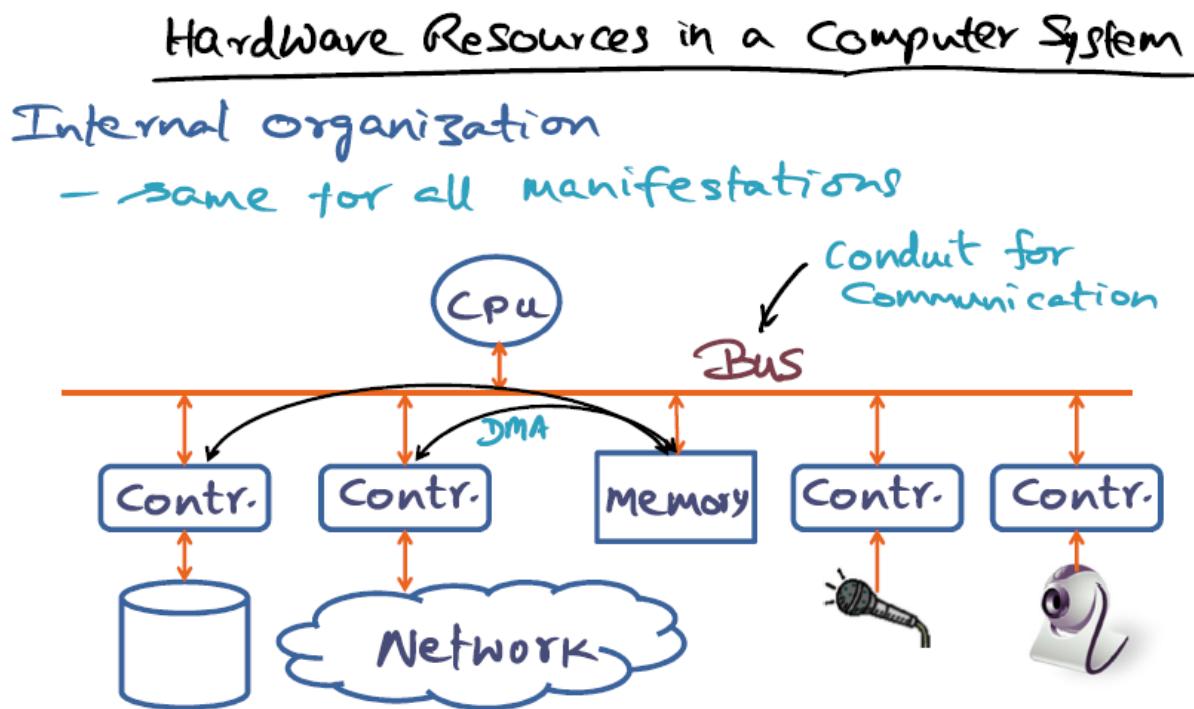
From the highest abstraction (Google Earth) to the lowest abstraction (electrons):

- Applications
- System Software (operating system, compilers, etc.)
- Instruction Set Architecture
- Machine Organization (datapath and control)
- Sequential and Combinational Logic Elements
- Logic Gates
- Transistors

# hardware resources

## Hardware resources in a computing system

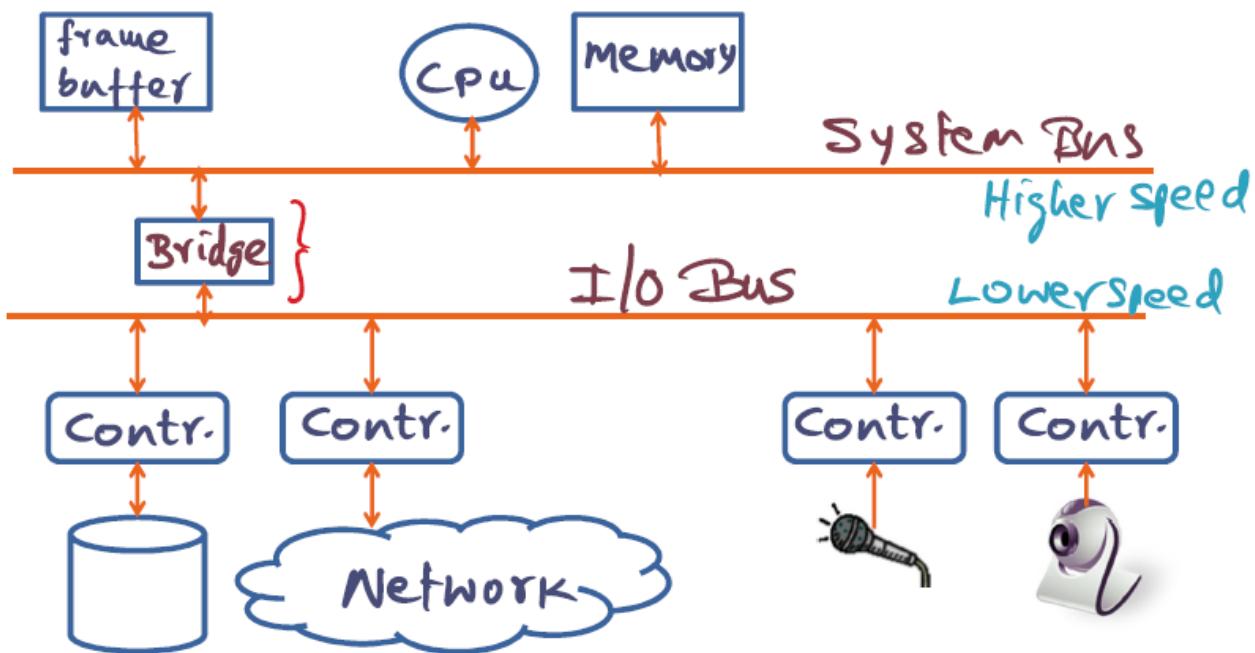
The internal organization of a computing system is the same for all manifestations. Below is a representation of what the hardware resources would look like in a traditional computing system.



## Bus organization

Computing systems utilize communication buses to relay information between devices, the CPU, and memory. Below is a representation of the buses that exist within a traditional computing system.

# Organization With I/O Bus



The CPU utilizes the system bus to directly interface with the memory of the computing system. The CPU utilizes a bridge to schedule interactions with devices residing on the I/O bus. Devices on the I/O bus can also directly interact with the memory utilizing the bridge if they are devices that leverage direct memory access (DMA).

## Definitions

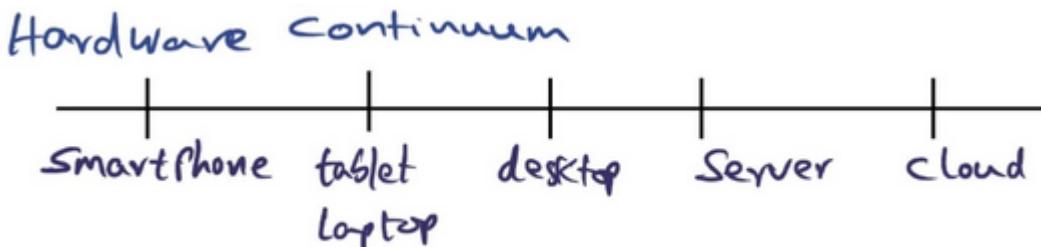
Term	Definition
bus	The conduit for communication between the central processing unit (CPU) and the devices connected that create the computing system.
system bus	A synchronous communication device between the CPU and memory. This bus has a high-speed communication bandwidth.
input / output (I/O) bus	A bus primarily intended for the devices to communicate with the CPU. This bus has a low-speed communication bandwidth.

---

bridge	Specialized input / output processor for scheduling the devices that need to communicate with the memory or the CPU.
direct memory access (DMA)	A capability provided by some computer bus architectures that allows data to be sent directly from an attached device (such as a disk drive) to the memory on the computer's motherboard.
frame buffer	Is a portion of random-access memory (RAM) containing a bitmap that drives a video display.

---

## Quizzes



**Are the internal organizations of the computing systems represented in this continuum vastly different?**

- Yes
- No

**No** is the right answer here. The hardware inside a computing system consists of the processor, memory, and input devices. Whether we are discussing smartphones, to cloud computing systems, the internal organization will not change tremendously.

# operating system functionality

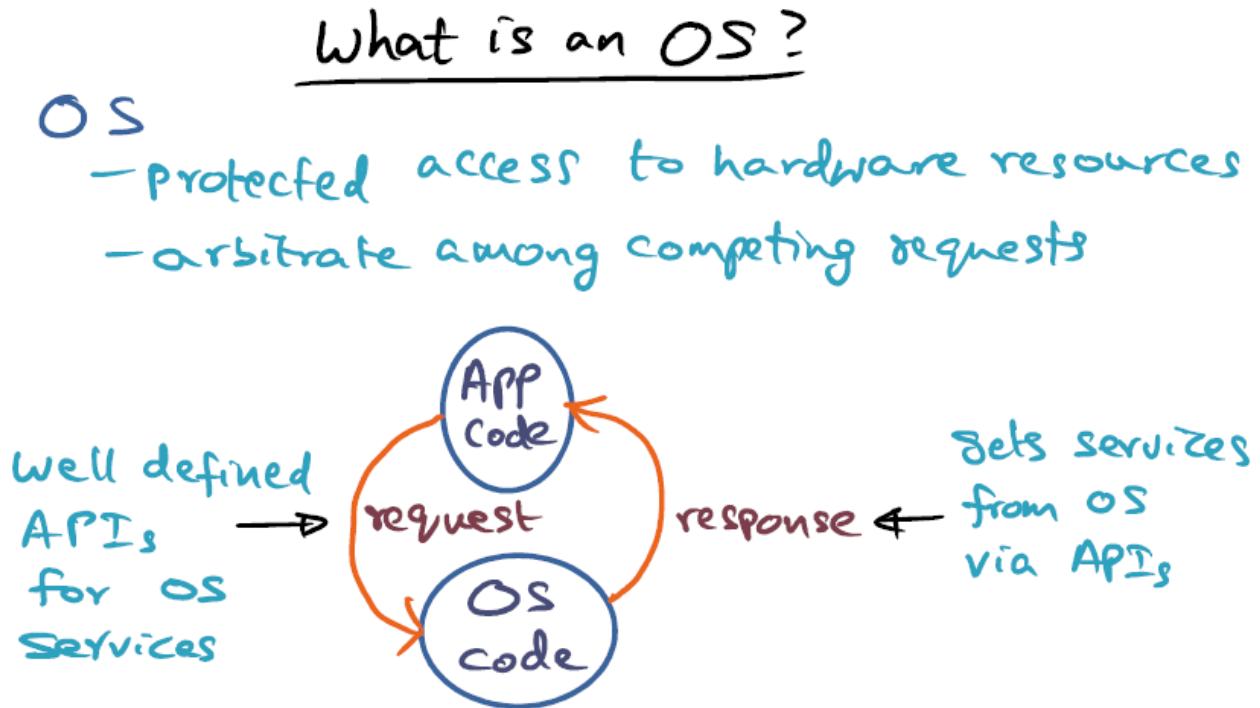
## What is an operating system?

An operating system contains code to access the physical resources of the computing system and arbitrates the competing requests for these hardware resources. An operating system is a complex program that abstracts these features for application programmers using well-defined application programming interfaces (API).

Below is a list of functionalities operating systems are designed to execute:

- protect access to hardware resources
- arbitrate among competing requests for hardware resources

Below is a high-level representation describing an operating system.



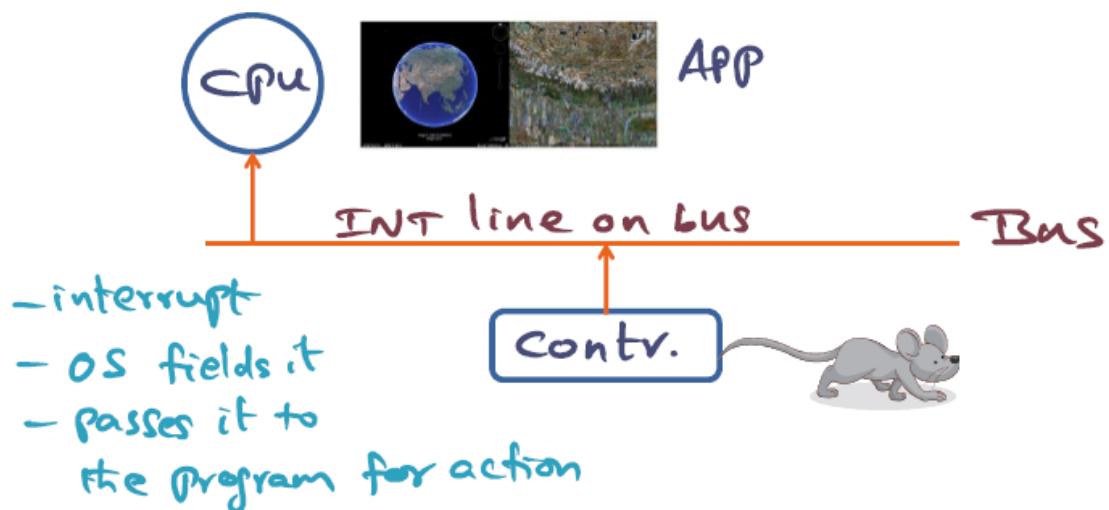
## What happens when you click your mouse?

In this scenario, we are running the Google Earth application. We select a point on the globe by clicking the mouse. So what happens? When you click your mouse, an interrupt is created on the dedicated interrupt line. This interrupt is served to the CPU. The operating

system fields this interrupt and determines the appropriate interrupt handler to execute for this mouse click action. The results of the execution of this interrupt handler are passed to the Google Earth program, and allows the user to interact with the program via the mouse.

Below is a high-level representation of how interrupts are served to applications via the interrupt bus, CPU, interrupt handler, and operating system:

What happens when you click your mouse?



## Definitions

Term	Definition
application programming interface (API)	A system of tools and resources in an operating system, enabling developers to create software applications.
interrupt	A signal created and sent to the CPU that is caused by some action taken by a hardware device.

---

interrupt handler	Are initiated by hardware interrupts, software interrupt instructions, or software exceptions, and are used for implementing device drivers or transitions between protected modes of operation, such as system calls.
----------------------	--

---

## Quizzes

**Select the choices that apply to the functionalities provided by an operating system.**

- An operating system is a resource manager.
- An operating system provides a consistent interface to the hardware resources.
- An operating system schedules applications on the CPU.
- An operating system stores personal information such as credit card numbers, social security numbers, email addresses, etc.

The last choice is not something you would want an operating system to do.

**What happens when you click the mouse?**

- Mouse clicks correspond to specific CPU instructions.
- Mouse click starts up a specific program to read spatial coordinates of the mouse.
- Mouse click results in a CPU interrupt.

Mouse clicks result in a CPU interrupt. This interrupt will eventually result in a system service reading the spatial coordinates of the mouse.

# managing the cpu and memory

## Introduction

The operating system serves to enact policies regarding the use of the resources of the computing system. These include:

- Requiring applications to share hardware resources
- Requiring applications to share the CPU
- Preventing applications from over-writing other applications' code or data

The operating system tries to accomplish all of these actions while also getting out of the way as quickly as possible in order to enable the applications.

## Catering to resource requirements

Applications usually have these resource needs:

- Time to run on the CPU
- Memory
- Access to peripheral devices

The operating system is not aware of an application's resource requirements, however, at launch the operating system can create a memory footprint related to that application. The memory footprint of an application includes:

- The code to be executed
- Global data
- The stack
- The heap

This memory footprint is created by the operating system loader when an application is launched. The application is able to ask the operating system for more resources at runtime. The operating system will perform this service on behalf of the application, and the application can continue executing.

Below is a high-level representation of how operating systems cater to application resource requirements.

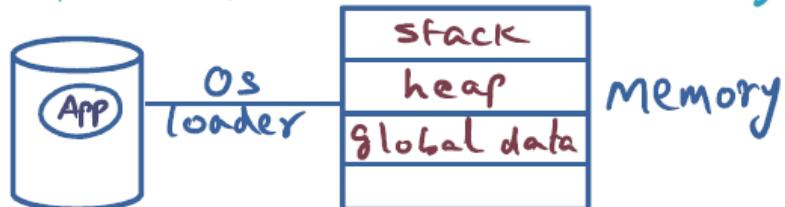
## Catering to Resource Requirements

Resource needs of applications

- CPU, memory, peripheral devices

App launch time

- Know how to create memory footprint



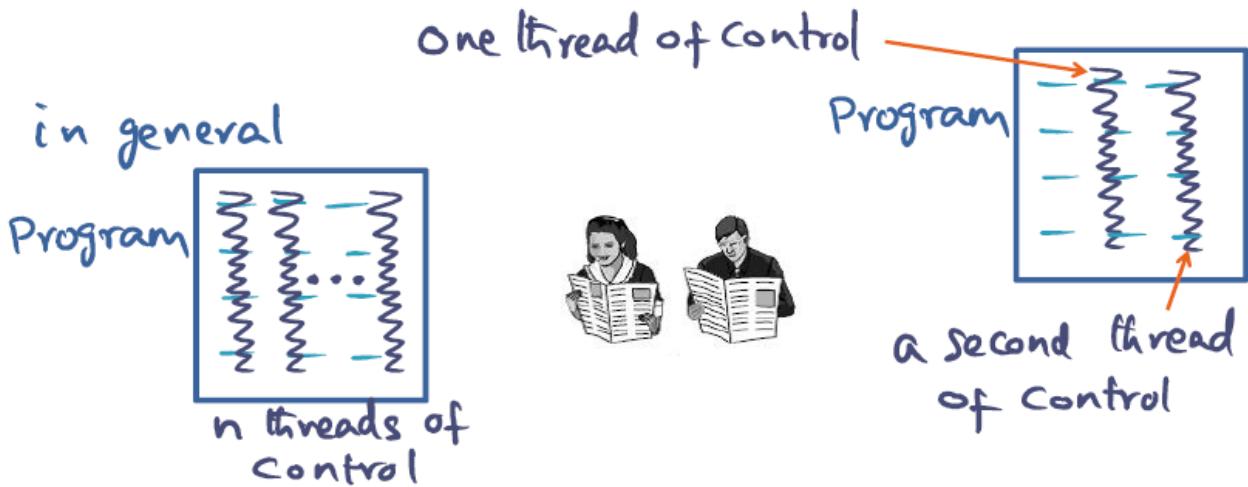
- App asks for additional resources at runtime

## Difference between a process and a thread

Threads represent one instance of execution within a process on the CPU. A process is the state of all the threads that are executing within an application.

Below is a high-level representation of the comparison between a process and a thread.

# Difference between Process + Thread?



Process = Program + state of all threads executing in the program

## Memory related operating system abstraction

Processes contain an address space that is distinct from another process. The operating system provides the abstraction for each process, and protects processes from other processes attempting to access their respective address space. The operating system implements this abstraction using the underlying hardware capabilities of the computing system.

## Definitions

Term	Definition
memory footprint	The amount of main memory that a program uses or references while running.
stack	An area of main storage (system memory) that is used for return addresses, procedure arguments, temporarily saved registers, and locally allocated variables.
heap	A region of memory that stores variables in a running program.

---

loader	Part of an operating system that is responsible for loading programs and libraries.
program	Static image that is to be loaded into memory.
process	A program in execution.
thread	A path of execution within a process.

---

## Quizzes

**Computers seemingly run several programs in parallel: e-mail, web browsing, music, etc.  
If there is only one CPU in the computing system, how is this possible?**

- Multiple cores exist on the CPU.
- Trick question, only one application runs at a time.
- The operating system schedules the time each application is allowed to run on the CPU.

The operating system multiplexes multiple applications to run for a specific time on the CPU, presenting the illusion that the applications are all running in parallel.

**If the operating system is a broker of resources, is it taking precious resources away from an application?**

- Yes
- No
- Maybe

The operating system is also a program that must execute on the CPU. A good operating system will utilize the minimal amount of time to execute its functions. Most of the time, the resources of the computing system will be primarily utilized by the applications.

lesson2

# **lesson2**

# operating system structure

## Goals of operating system structure

Goal	Description
Protection	Protecting the user from the system, and vice versa, as well as protecting users from one another. Also, implementing protection to protect the user from their own mistakes.
Performance	Time taken to execute operating system services on behalf of an application.
Flexibility	Also called extensibility, a service provided by the operating system is adaptable to the requirements of an application.
Scalability	The performance of an operating system increases as hardware resources are increased.
Agility	The ability to adapt to the changes in an application's resource needs and / or resource availability.
Responsiveness	How quickly the operating system reacts to external events.

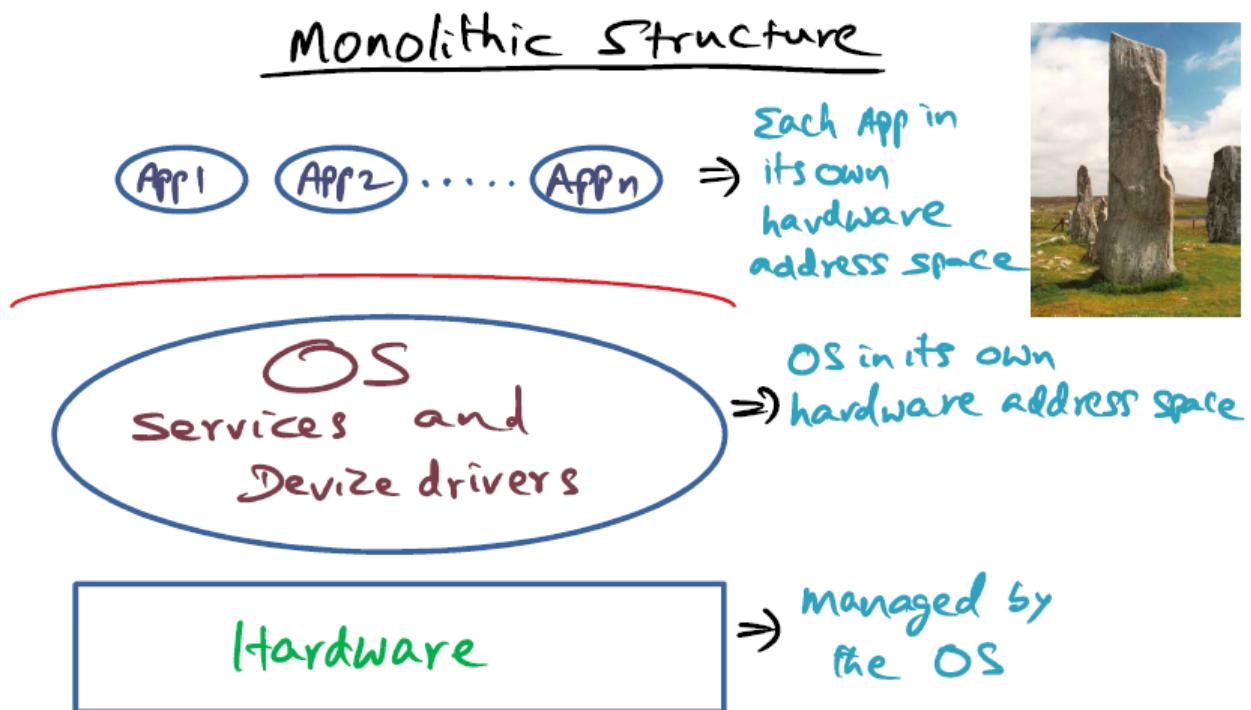
## Monolithic structure

In the monolithic operating system structure, all operating system services are contained within its own hardware address space, protected from all user-level applications. The hardware is also directly managed by the operating system. Each application running on the operating system is contained within its own hardware address space.

Due to the segregation implemented in the structure of this operating system, the operating system is protected from the malfunctions that can occur in the execution of user-level applications. When an application needs any system service, a context switch is conducted into the hardware address space of the operating system.

A monolithic structured operating system reduces performance loss by consolidating all system services. What is lost with the monolithic operating system structure is the inability to customize operating system services for the needs of different applications.

Below is a high-level representation of the monolithic operating system structure.



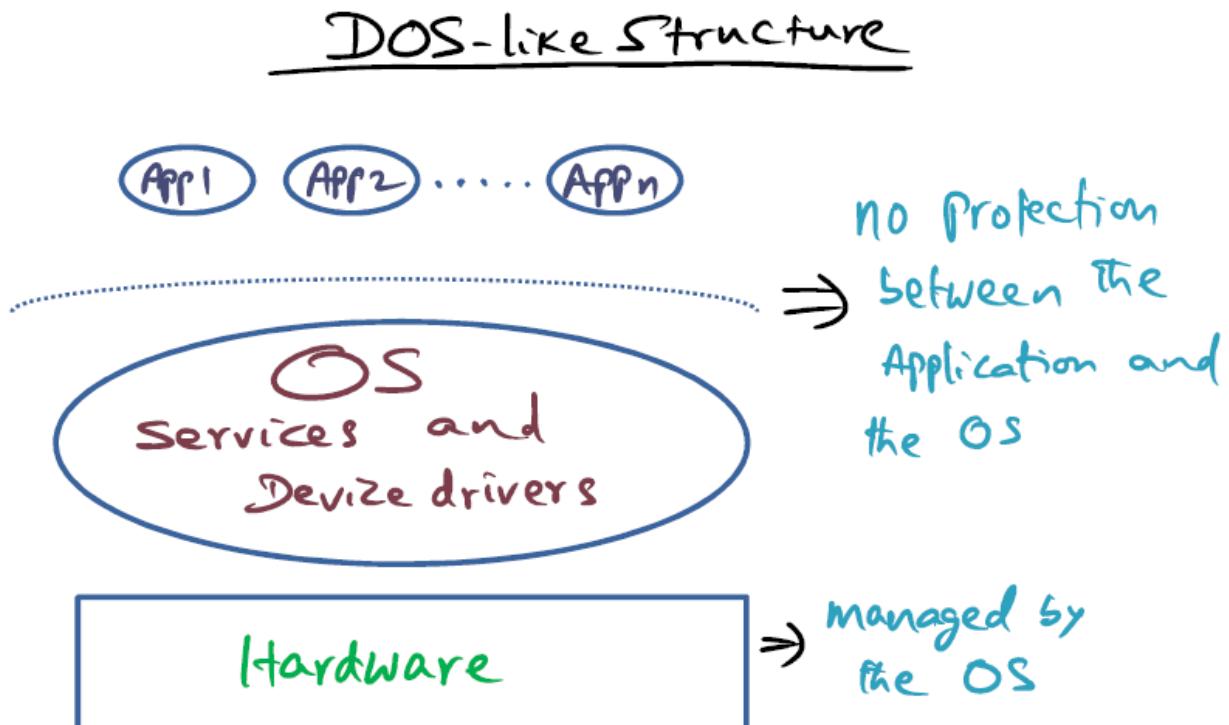
## DOS-like structure

The primary difference between the monolithic operating system structure and the DOS-like operating system structure is that there exists no hard separation between the applications' and the operating system's address spaces.

This structure provides applications the ability to make system calls into the operating system and acquire system services at the same speed of a procedure call. This feature also presents a security issue, as an errant application can compromise the integrity of the operating system and corrupt its data structures.

The loss of protection that is experienced with a DOS-like structured operating system is unacceptable for modern computing systems.

Below is a high-level representation of the DOS-like operating system structure.



## Microkernel structure

In the microkernel operating system structure, each application has its own hardware address space. The microkernel runs in a privileged mode of the architecture and implements a small number of mechanisms for accessing hardware resources.

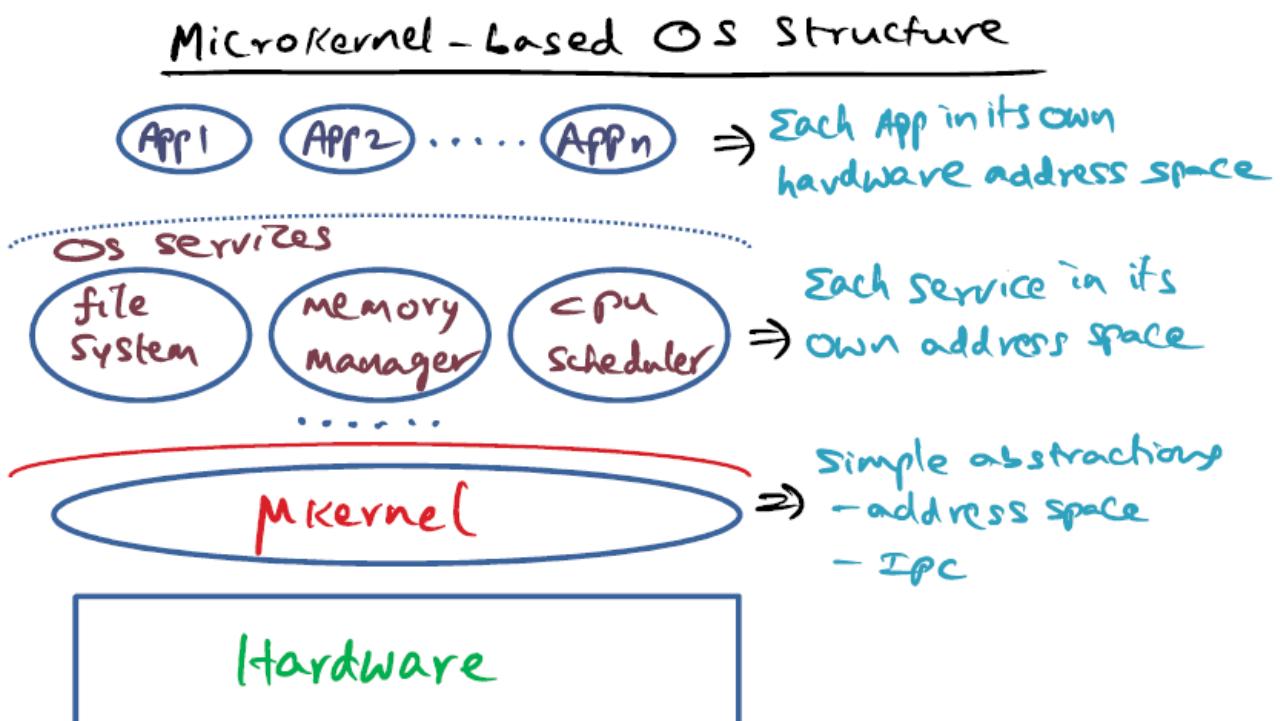
This is a key distinction, the microkernel implements no policies for use of the hardware resources - it only provides abstractions for the underlying system hardware.

The operating system services are implemented as servers that reside on top of the microkernel and execute with the same privilege as the user-level applications. Each system service is contained within its own hardware address space, as well.

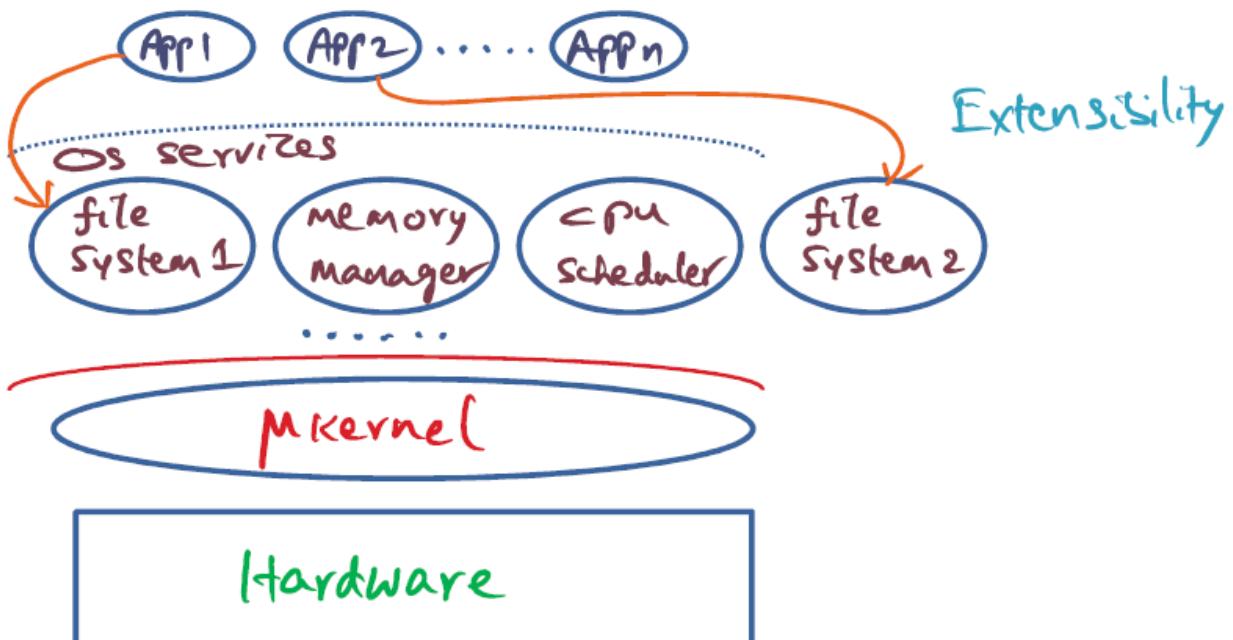
The microkernel implements interprocess communication, allowing the applications the ability to communicate between each other and with the system services. This also allows the system services to communicate among one another.

The microkernel provides a greater level of extensibility, allowing programmers to customize services and tailor system services to application needs.

Below is a high-level representation of the microkernel operating system structure.



Advantage of microkernel - based design



## Downsides to the microkernel structure

In comparison to the monolithic operating system structure, the microkernel operating system structure must conduct more communication between system services, applications, and the microkernel itself in order to allow applications to conduct privileged operations. There could potentially be several context switches required in order to contact all of the requisite servers for a particular system call.

Listed are the performance costs associated with the microkernel operating system structure:

Cost	Description
Context switching	Switching from the application level hardware address space to the microkernel address space, excessively.
Change in locality	Constant changes of the contents residing within the cache due to the consistent context switches occurring.
Content copying	There is a need to copy information from the system services' address space to the address space of the microkernel.

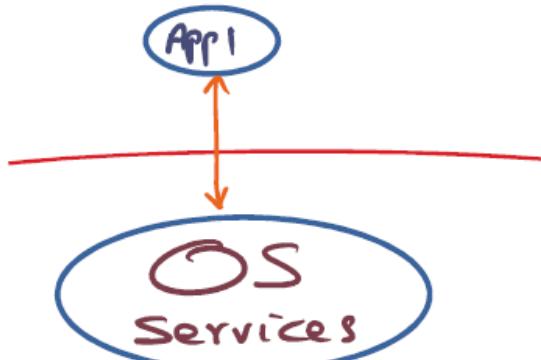
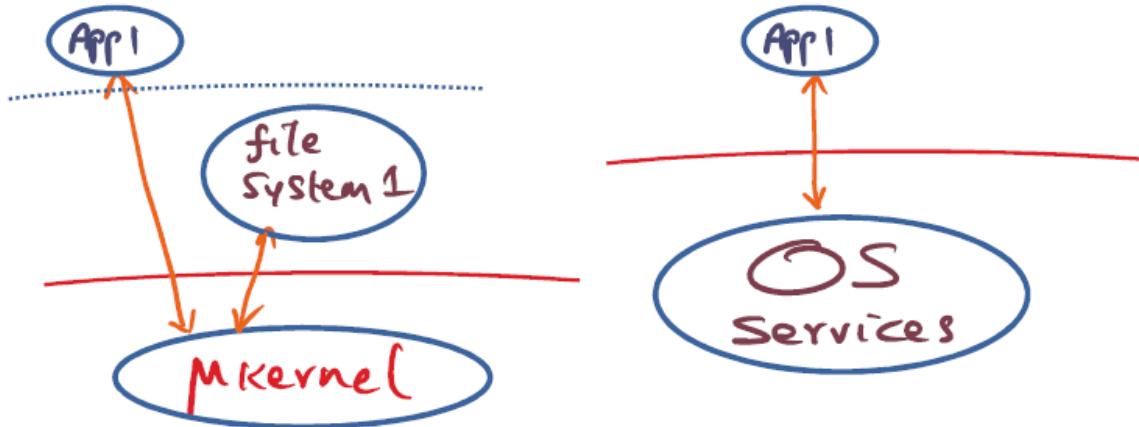
Below is a high-level representation of the downsides of a microkernel operating system structure.

## Why Performance Loss?

Border crossings

- change in locality

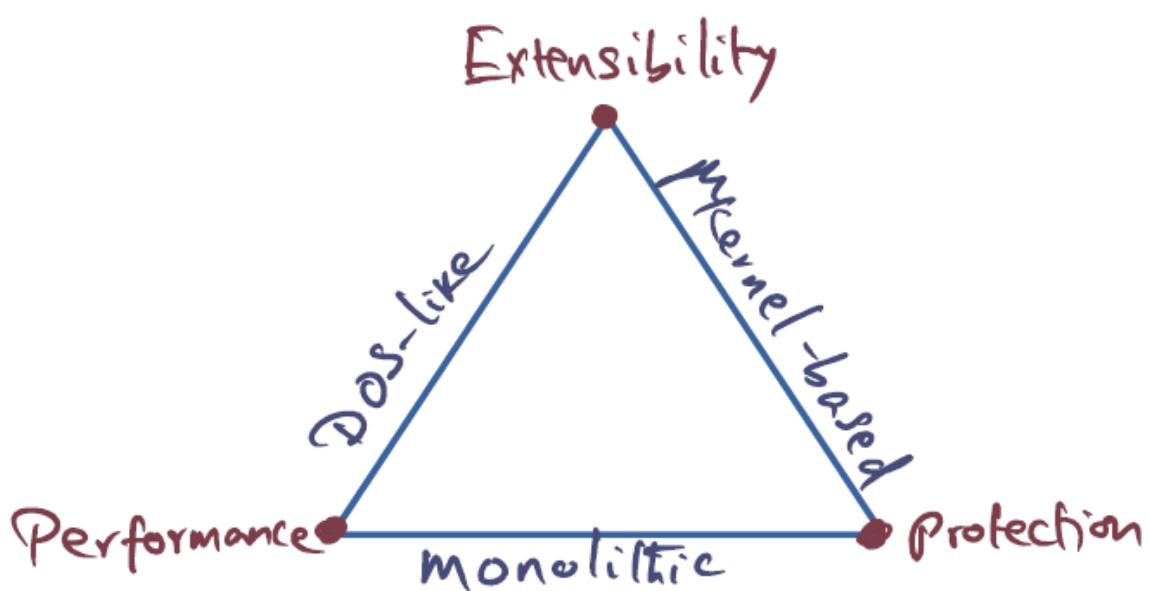
- user space  $\leftrightarrow$  system space copying



## Balance of an operating system structure

Below is a representation of the operating system structure triangle, comparing the attributes of performance, extensibility, and protection.

What do we want?



## Definitions

Term	Definition
operating system structure	The way the operating system is organized with respect to the applications that it serves and the underlying hardware that it manages.

## Quizzes

**Name system services you can expect from an operating system.**

- Process / thread management and scheduling.
- Memory management.
  - Protection, sharing, demand paging.
- Interprocess communication.
- File system management.
- Network access.
- Access to I/O devices such as microphones and speakers.

**Why is the structure of the operating system so important?**

- Protection
- Performance
- Flexibility
- Scalability
- Agility
- Responsiveness

**What is gained with a DOS-like operating system structure?**

Performance - access to system resources like procedure call.

**What is lost with a DOS-like operating system structure?**

Protection - an errant application can corrupt the operating system.

Based on the discussion above, rate the features of each of the described operating system structures.

### Question/Answer

Based on discussion thus far....

Feature	Monolithic OS	DOS-like OS	μKernel OS
Extensibility		✓	✓
Protection	✓		✓
Performance	✓	✓	

# the SPIN approach

## Operating system goals, revisited

What type of operating system structure are we looking to implement? In a perfect world, we want an operating system structure that is:

- Thin and implements only mechanisms, not policies, like the microkernel operating system structure.
- Provides access to resources without crossing between privilege levels like the DOS operating system structure.
- Is flexible in its resource management like a microkernel structured operating system without sacrificing protection and performance like a monolithic structured operating system.

In a nutshell, we want performance, protection, and extensibility.

## Approaches to extensibility

Historically, there was research conducted on extensibility as far back as 1981. [Hydra OS](#) attempted to tackle extensibility in a capability-based manner with these objectives:

- Kernel mechanisms for resource allocation.
- Capability-based resource access.
- Implementing resource management using capabilities as coarse-grained objects to reduce privilege level crossing overhead.

[Mach](#) was also a microkernel-based operating system designed during that time period that focused on extensibility and portability. Performance took a hit as Mach attempted to maximize portability and extensibility for various different architectures.

The results of attempting to implement these two operating systems and the discovery of their lack of performance in comparison to the monolithic kernel gave the microkernel operating system structure negative publicity.

## SPIN's approach to extensibility

The key features SPIN attempts to implement in pursuit of extensibility while also maintaining performance and protection are:

- Co-location of the kernel and its extensions in an attempt to avoid crossing between privilege levels too often.
- Compiler enforced modularity utilizing a strongly-typed programming language, preventing dangerous memory accesses from occurring between the kernel and the extensions.
- Utilizing logical protection domains, not hardware address spaces, to enforce protection.
- Application's ability to dynamically bind to different implementations of the same interface functions, providing flexibility.

Because the extensions are co-located with the kernel, extensions and their advertised system services are as cheap as a procedure call. No privilege level crossing occurs.

## Logical protection domains

SPIN implements its logical protection domains using the [Modula-3](#) programming language. Modula-3 provides these features and abstractions:

- Type safety
- Auto storage management
- Objects
- Threads
- Exceptions
- Generic interfaces

All of these features enables the implementation of system services as objects with well-defined entry points. This is leveraged to create logical protection domains within SPIN that can be co-located with the kernel. Through this, these logical protection domains provide protection and performance.

This object-oriented approach provides the designer the ability to define system resources as capabilities, with definitions as narrow or as broad as desired, thus providing protection. Possible ways to declare resources as objects include:

- Hardware resources (e.g. page frames)
- Interfaces (e.g. the page allocation module)
- A collection of interfaces (e.g. an entire virtual machine)

Capabilities to these objects were supported by the language in the form of pointers.

## **SPIN mechanisms for protection domains**

Mechanism	Description
Create	Initialize with object file contents and export names from the namespace.
Resolve	Resolve names between the source and target domains. This dynamically binds two protection domains. Once resolution is complete, resource sharing can occur at memory speeds.
Combine	Creates an aggregate protection domain, exporting entrypoints which is a union of the combined domains.

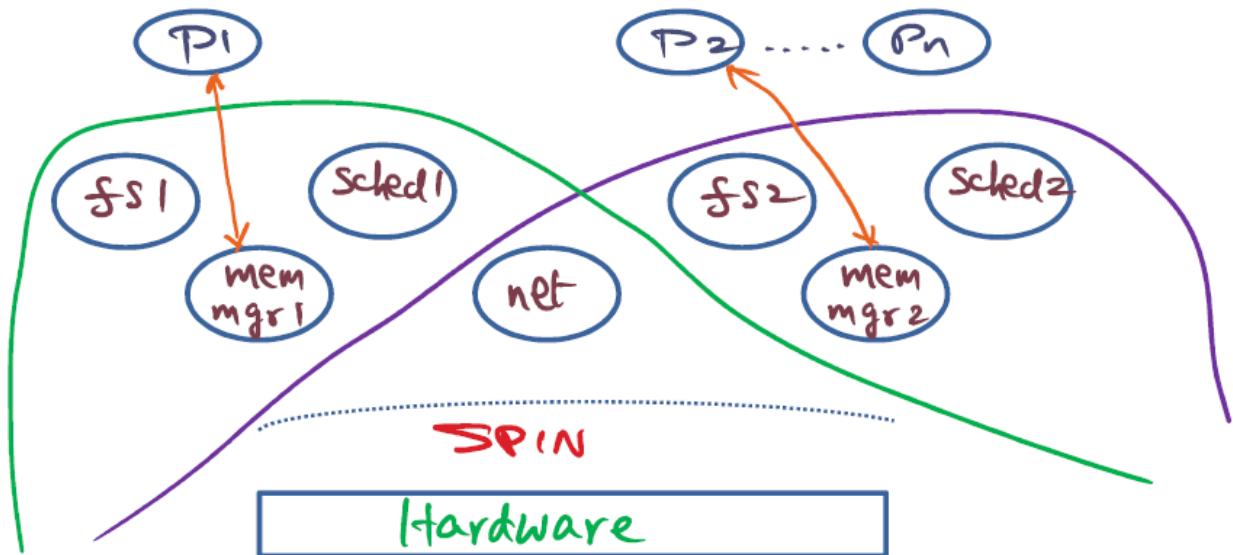
These mechanisms enabled by the Modula-3 operating system are how SPIN provides extensibility while maintaining performance and protection.

## **Customized operating system with SPIN**

Below is a high-level representation of how a SPIN operating system could be customized with custom extensions.

## Customized OS with SPIN

Create, Resolve, Combine

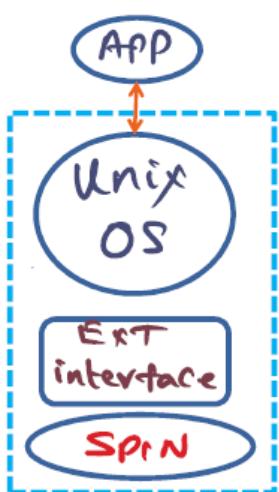


## Example extensions

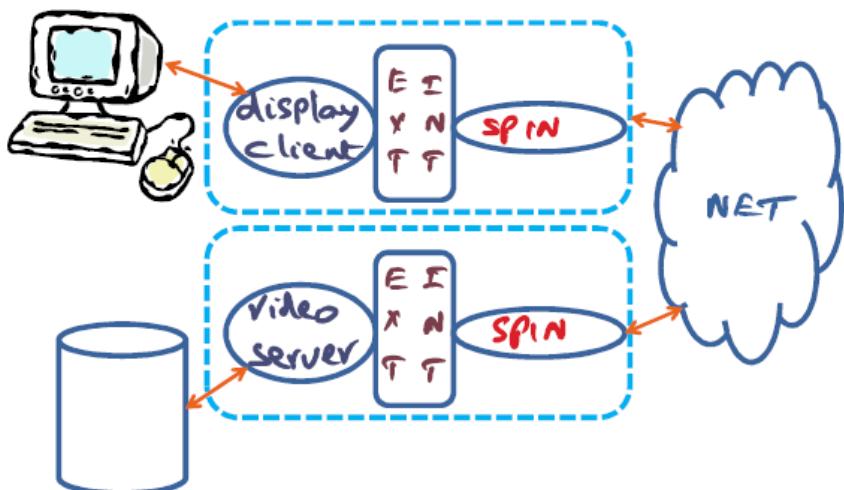
Below is a high-level representation of example extensions for SPIN.

### Example Extensions

#### Unix Server



#### Video Server + display client



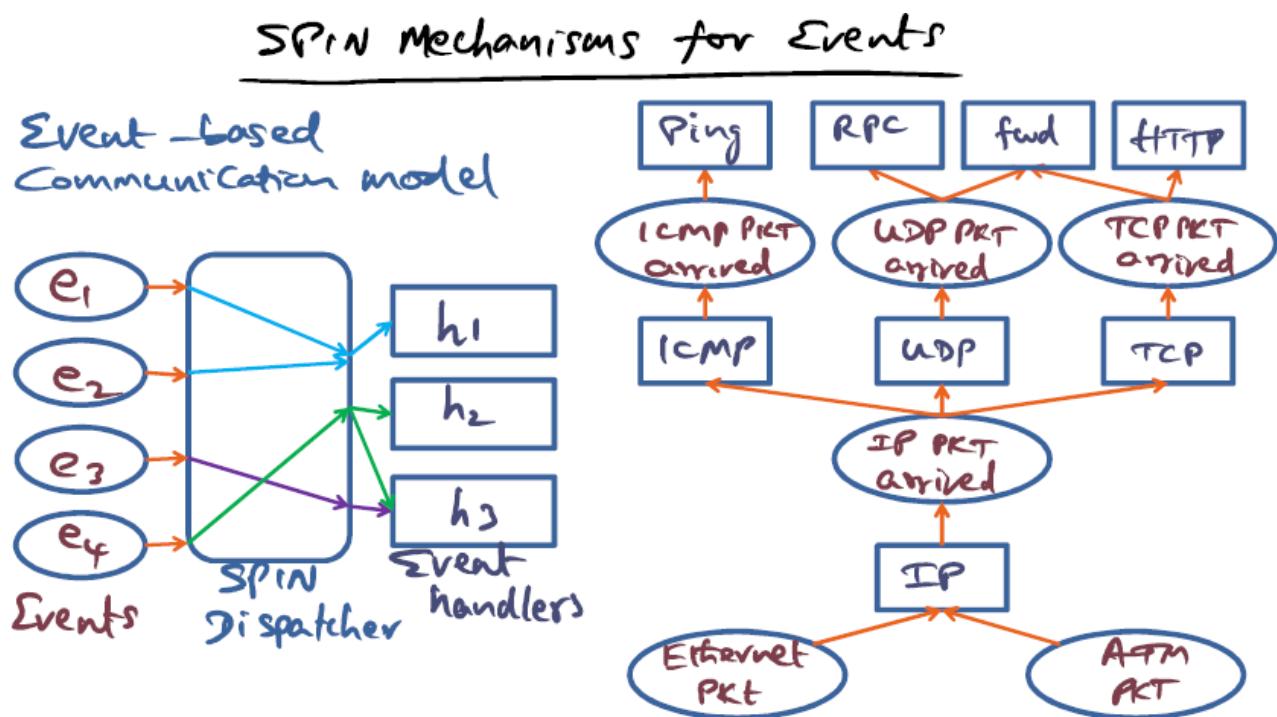
## SPIN mechanisms for events

SPIN needs to support fielding external events that occur, such as:

- External interrupts
- Exceptions
- Page faults
- System calls

SPIN supports external events using an event-based communication model. SPIN supports the mapping of events to event handlers in a one-to-one, one-to-many, and many-to-one manner.

Below is a high-level representation of how SPIN uses its event-based communication model to handle events.



## Default core services in SPIN

Below is a table listing the interface functions SPIN exposes to extensions for use in implementing memory management.

Resource	Description

---

Physical address	allocate, deallocate, reclaim
Virtual address	allocate, deallocate
Translation	create / destroy address spaces, add / remove mapping between virtual addresses to physical frames
Event handlers	page fault, access fault, bad address

---

These interface functions are just a header file provided by SPIN. All of the inner-workings of these functions will have to be implemented by the extension author.

Below is a table listing the interface functions SPIN exposes to extensions for use in implementing CPU scheduling.

Resource	Description
Strand	SPIN provided abstraction that represents a thread, used by the SPIN global scheduler.
Event handlers	block, unblock, checkpoint, resume
SPIN global scheduler	Decides the amount of time given to a particular extension to run on the CPU.

## Final assessment

Core, trusted services within SPIN provide access to hardware mechanisms. These services will have to exit the language-defined protection to control the hardware resources. The applications that run using an extension have to trust the extension and the information it returns. Extensions to core services only affect the applications that use that extension. Errant extensions will not affect applications not using the errant extension.

## Definitions

Term	Definition
capability	a communicable, unforgeable token of authority
pointer	a programming language object that stores a memory address

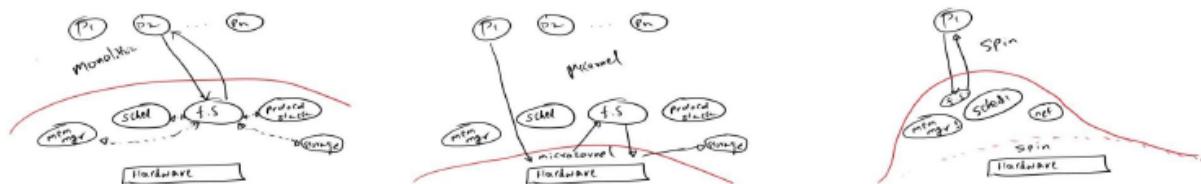
## Quizzes

**What is the difference between pointers in C and pointers in Modula-3?**

- No difference exists.
- C pointers are more restricted.
- Modula-3 pointers are type-specific.

Pointers in Modula-3 cannot be forged. There is no way to subvert the protection mechanism that is built into the Modula-3 language. There is no possible way to cast pointers to a different type in Modula-3.

**Which of the below structures will result in the least amount of privilege level crossings?**



*Which of the above structures will result in least border crossings?*

- Monolithic
- Microkernel
- SPIN
- Either SPIN or Monolithic

# the exokernel approach

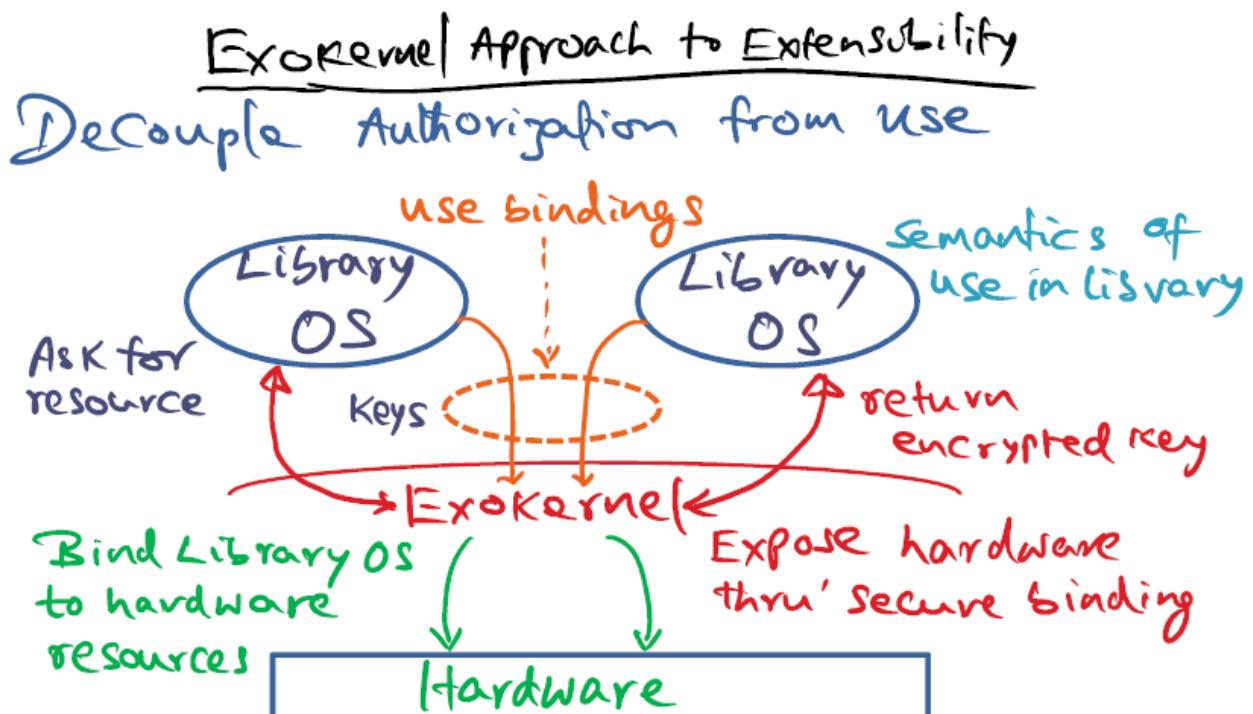
## Introduction

The main idea behind the exokernel is to explicitly expose the hardware resources to processes attempting to use said resources. The exokernel **decouples authorization from use** for hardware resources.

Library operating systems request a hardware resource from the exokernel and in turn the exokernel provides the library operating system with an encrypted key. The exokernel binds the library operating system to the requested hardware resource and the library operating system can now utilize the hardware resource without the exokernel's interference by presenting the encrypted key.

How the library operating system utilizes the hardware resource is entirely up to it. The exokernel is only designed to protect the resources and expose the resources when correct credentials are presented for authentication.

Below is a high-level representation explaining this policy.



## Implementing secure bindings

Three types of methods are available that allow the exokernel to securely bind resources to library operating systems:

Method	Description
Hardware mechanisms	e.g. TLB entry, page frame, or a portion of the frame buffer
Software caching	'shadow' TLB in software for each library operating system
Downloading code into the kernel	functionally equivalent to SPIN extensions

## Default core services in the exokernel

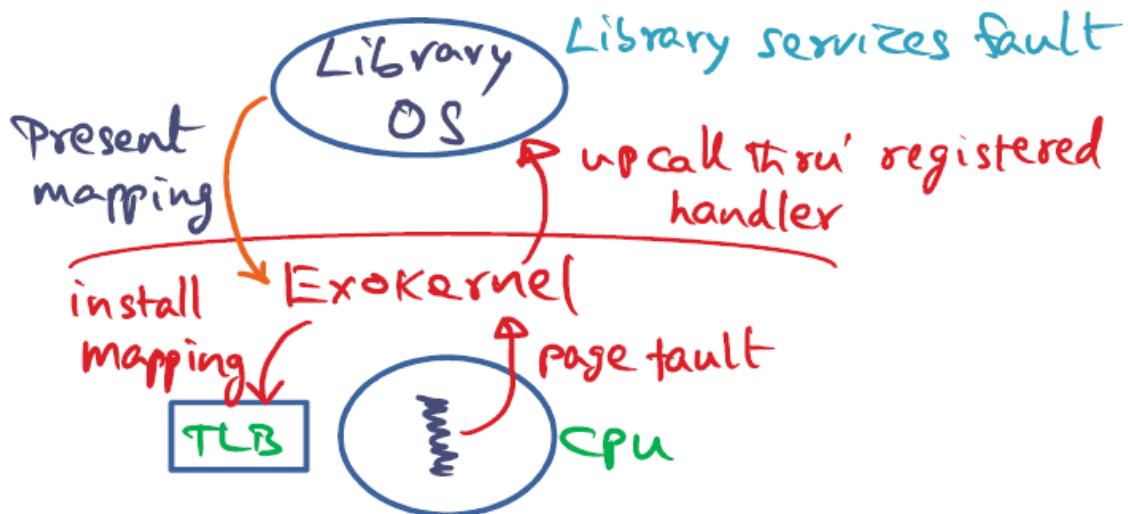
The exokernel doesn't resolve page faults or virtual to physical memory mappings for pages. Instead, the exokernel provides the page fault to the library operating system or application that owns the thread running on the CPU that generates the page fault. The library operating system services the page fault and presents the correct mapping of virtual to physical memory to the exokernel, along with the library operating system's key to access the hardware.

The exokernel will proceed to update the TLB, install the mapping, and then the library operating system's process / thread will be ran again on the CPU.

Below is a high-level representation of this mechanism.

## Default Core Services in Exokernel

### Memory Management

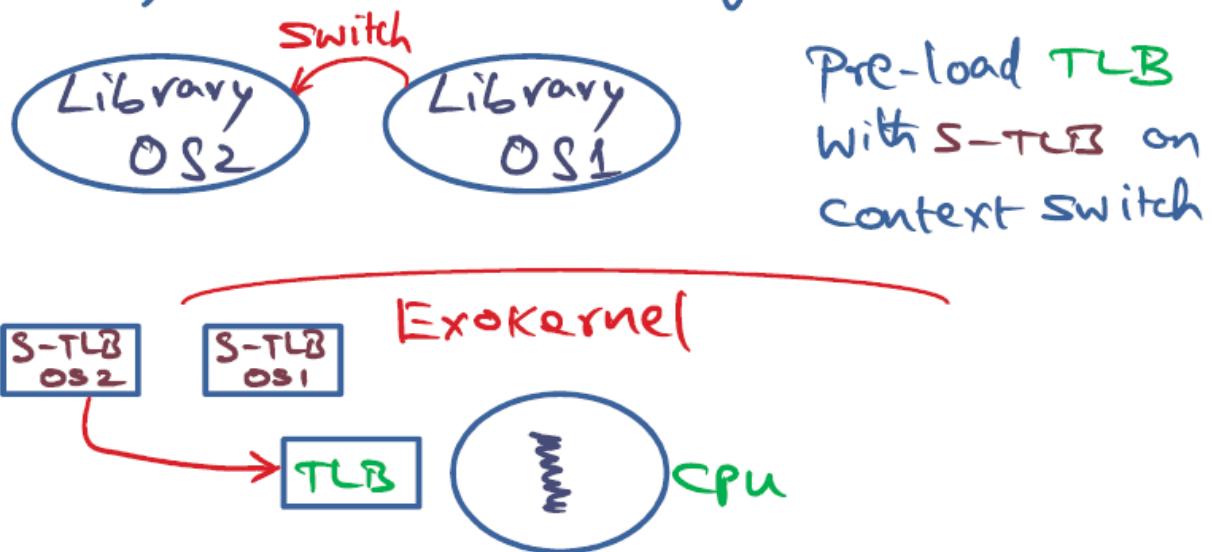


### Memory management using the software TLB

When a context switch occurs, one of the biggest performance losses is the loss of locality within the cache and the TLB. The exokernel utilizes a software TLB for each library operating system that represents each system's memory mapping. When a context switch occurs, the exokernel will place some subset of the library operating system's software TLB into the hardware TLB. This helps to mitigate the loss of locality from context switching.

Below is a high-level representation of the exokernel's mechanism for maintaining software TLBs.

# Memory Management : Using S-TLB



## CPU scheduling

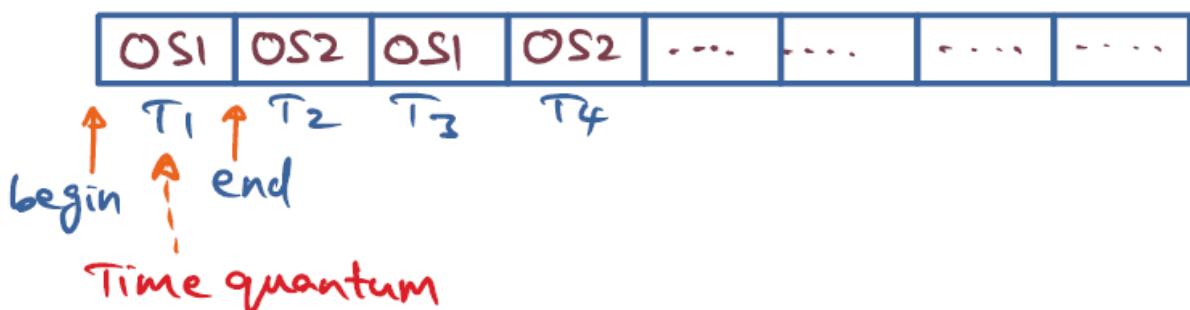
The exokernel maintains a linear vector of time slots for each library operating system to occupy. Each slot represents a time quantum, and the time quantum dictates how long each library operating system is allowed to execute on the CPU. On startup, the library operating system is allowed to dictate which locations within the linear vector it wants to occupy.

Below is a high-level representation of the exokernel's mechanism for scheduling library operating systems.

## Default Core Services in Exokernel

### CPU Scheduling

— Linear vector of "time slots"



### Revocation of resources

The exokernel needs a mechanism to revoke resources from different library operating systems. The exokernel doesn't have an understanding of what the library operating system is using the CPU for, however, unlike SPIN's abstraction of strands.

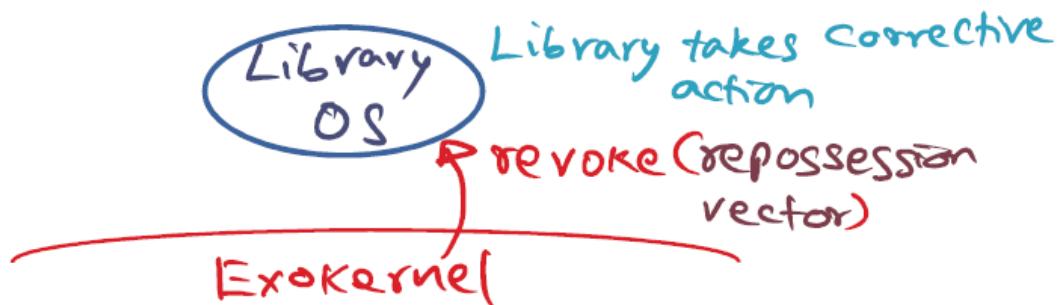
The exokernel has a revoke call that provides a library operating system a repossession vector, describing the resources the exokernel is going to revoke from the library operating system.

The library operating system cleans up whatever it was doing with the defined resources in the repossession vector, usually by saving the contents to disk. The library operating system is provided the mechanism to seed the exokernel, allowing the library operating system to request an autosave mechanism whenever the exokernel intends upon revoking resources.

Below is a high-level representation of the exokernel's revocation mechanism.

## Revocation of Resources

Space (memory) and time (CPU)



library can "seed" Exokernel for "autosave"

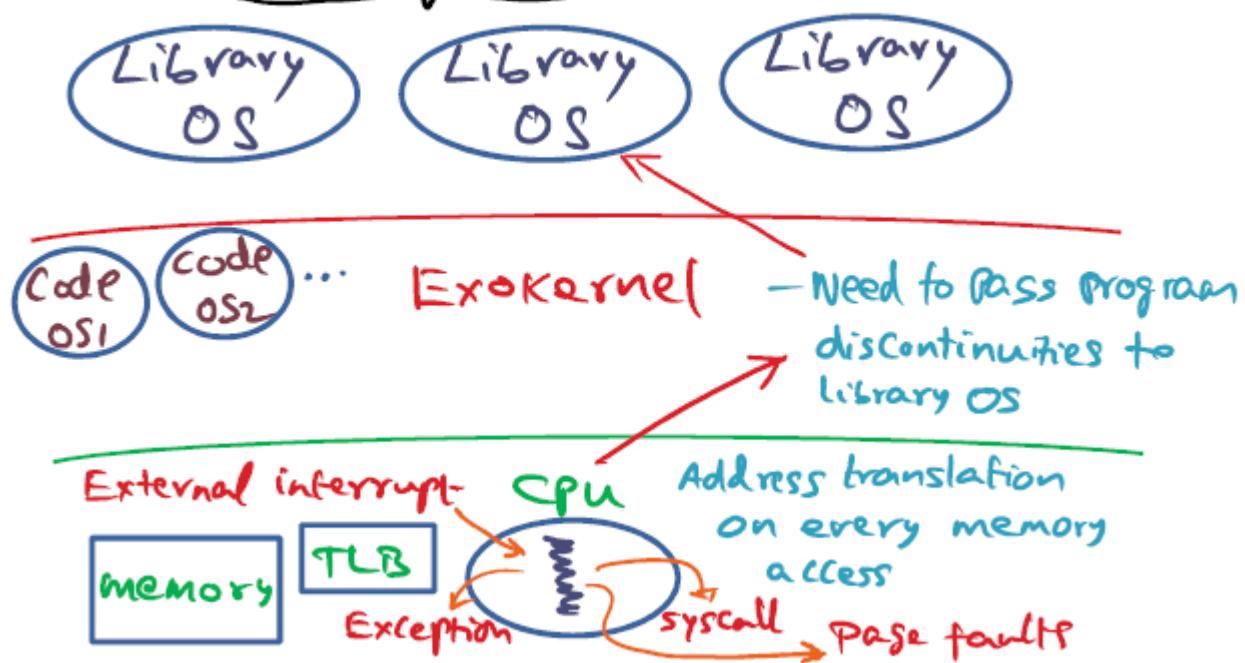
## **Putting it all together**

Code that is performance critical can be downloaded into the exokernel creating somewhat of an extension for a particular library operating system.

The exokernel passes all program discontinuities to the library operating system that owns the thread currently executing on the CPU. In order to do this with fine granularity, the exokernel will maintain a state for each library operating system. This allows the exokernel to maintain exceptions, page faults, syscall traps, external interrupts, etc. for the next time a library operating system is scheduled to handle the discontinuity.

Below is a high-level representation of these concepts.

## Putting it all together



## Exokernel data structures

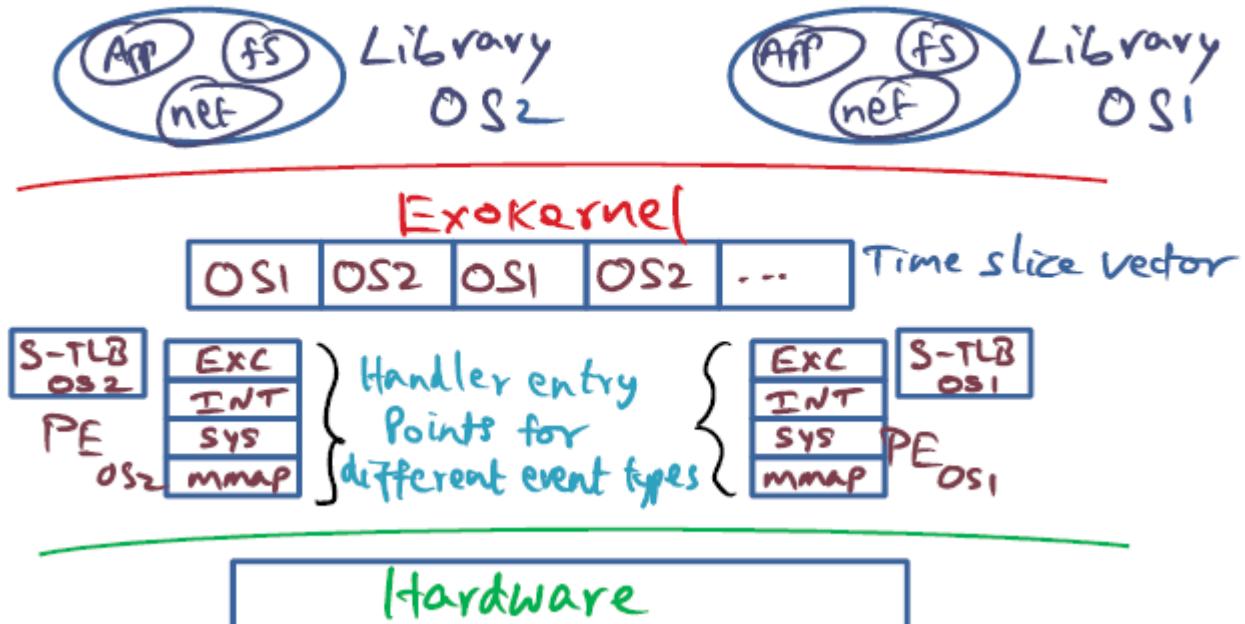
The exokernel maintains a **PE** data structure on behalf of each library operating system containing:

- entry points for library operating systems to deal with different discontinuities
  - exceptions
  - interrupts
  - syscalls
  - page faults (requests for an addressing context)
- Software TLB

Downloading code into the kernel allows for first level interrupt handling by the exokernel on behalf of a library operating system.

Below is a high-level representation of the exokernel data structures maintained for each library operating system.

## Exokernel Data Structures



## Performance results of SPIN and exokernel

Key takeaways:

- SPIN and exokernel are better at procedure calls between protection domains than a microkernel.
- SPIN and exokernel performance is on par with the monolithic kernel operating system structure for conducting syscalls.

## Definitions

Term	Definition
translation lookaside buffer (TLB)	a memory cache that is used to reduce the time taken to access a user memory location.

---

page fault	a type of exception raised by computer hardware when a running program accesses a memory page that is not currently mapped by the memory management unit (MMU) into the virtual address space of a process.
software TLB	a TLB maintained by the exokernel for each library operating system in order to mitigate the performance loss experienced by context switches
time quantum	the amount of time a library operating system is allowed to execute on the CPU
repossession vector	a vector describing the resources the exokernel intends upon revoking from a library operating system
library operating system	operating system which the services that a typical operating system provides, such as networking, are provided in the form of libraries and composed with the application and configuration code to construct a unikernel.

---

## Quizzes

**Exokernel's mechanism for library operating systems to "download code into the kernel" vs. SPIN "extensions and logical protection domains". Which one of these mechanisms compromises protection more?**

- SPIN
- Exokernel

SPIN enforces protection at compile-time and has runtime verification. For the exokernel, a library operating system can download arbitrary code into the kernel. Not very safe.

**Give a couple of examples of how "code downloaded" from a library operating system may be used by the exokernel.**

1. Installation of a packet filter for de-multiplexing of network packets.
2. Run code in the exokernel on behalf of the library operating system not currently scheduled (e.g. garbage collection for an application).

# the L3 microkernel approach

## Introduction

The creators of the SPIN and exokernel operating systems approached their research with the assumption that microkernel design had inherently poor performance. This is due to the fact that the most popular microkernel design of the period was Mach - a microkernel aimed at providing portability, not performance.

In this portion of the lesson we will look at L3, an operating system design that attempts to prove microkernel designs can provide performance.

## Potentials for performance loss

- **Border crossings (context switches)**
  - Implicit and explicit costs are imposed when conducting so many context switches, trapping into the microkernel.
    - Explicit cost - application trapping into microkernel space.
    - Implicit cost - trapping into a system service in order to complete the requested service of the application.
- **Protected procedure calls**
  - Expensive operations in which system services need to communicate.
  - System services must trap into the microkernel in order to build an IPC connection - the two system services are in different address spaces.
    - Implicit cost - locality loss every time we conduct a context switch for protected procedure calls.

Below is a high-level representation of the potential locations within a microkernel where we can lose performance.

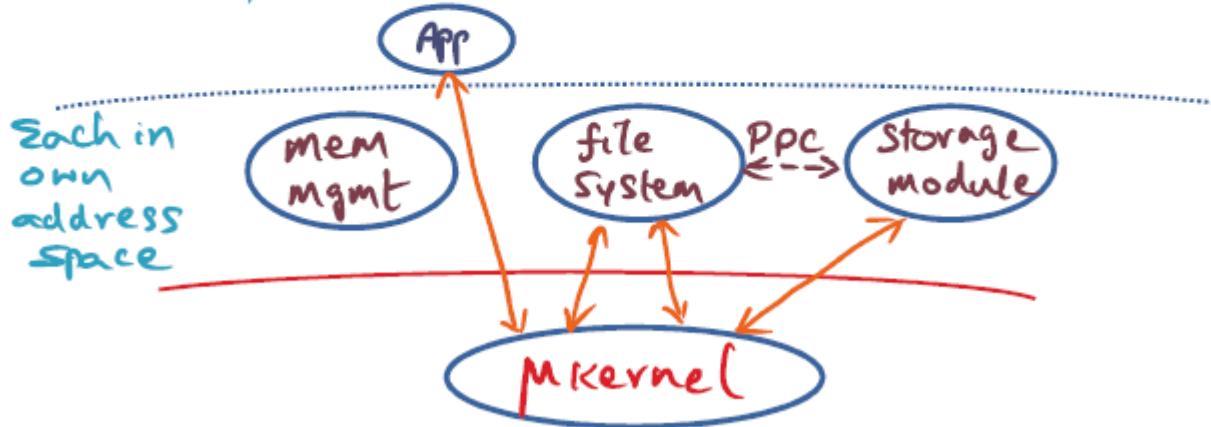
## Potentials for Performance Loss

### Border Crossings

- implicit + explicit costs

### Protected procedure calls

- 100x normal procedure calls



## L3 microkernel

By proof of construction, L3 attempts to *debunk* myths about microkernel-based OS structure.

The key distinction about the L3 microkernel is that each system service has to exist within its own protection domain, however, each system service doesn't have to exist within its own address space.

## Strikes against the microkernel

Below is a table describing the issues that exist within the design of a microkernel that impact performance.

Issue	Definition

---

Kernel-user switches	border cross cost (context switches)
Address space switches	<p>basis for protected procedure calls for cross protection domain calls</p> <ul style="list-style-type: none"><li>going across hardware address spaces, at a minimum, requires a context switch and the flushing of the TLB</li></ul>
Thread switches and IPC	kernel mediates all protected procedure calls
Memory effects	<p>locality loss when we context switch to different address spaces of different system services</p> <ul style="list-style-type: none"><li>each thread for each system service will most likely encounter a cold cache or a TLB with bad translations</li></ul>

---

Below is a high-level representation of the issues listed above.

## Strikes against μkernel

Kernel - user switches  
- border crossing cost



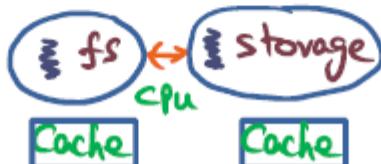
Address space switches  
- basis for PPC for cross protection domain calls



Thread switches + IPC  
- Kernel mediation for PPC



Memory effects  
- Locality loss



## Debunking user-kernel border crossing myth

L3 provides empirical proof in their paper that their context switch utilizes **123 processor cycles** - including re-mapping the TLB and resolving cache misses.

Mach on the same hardware utilizes **900 process cycles** to conduct a context switch, resolve cache misses, and re-map the TLB.

## Address space switches

When a context switch takes place, do we have to flush the TLB? Is the entire virtual to physical address mapping invalid?

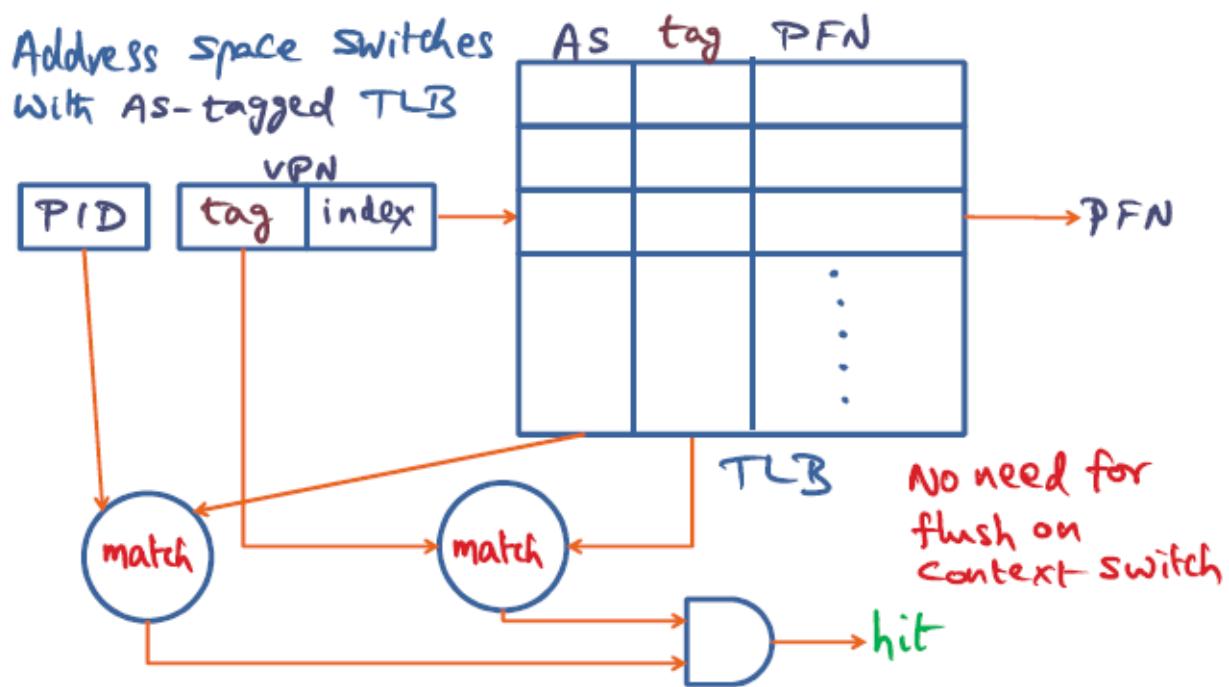
It depends on whether the TLB has a mechanism to recognize the virtual to physical address is tagged for a distinct process. This is called an **address space tag TLB**, where each TLB entry is tagged for a process's address space.

## Address space switches with address space tagged TLB

The TLB has two tags, the address space tag, and the virtual to physical mapping tag. If the process ID within the address space tag and the virtual to physical mapping tag matches the entry in the TLB, we have a TLB hit.

No need for us to conduct a context switch as the hardware allows us to store multiple different process' virtual to physical mappings within the TLB.

Below is a high-level representation of a TLB with address space tagging.



## Liedtke's suggestions for avoiding a TLB flush

- Exploit architectural features
  - e.g. segment registers in x86 + powerpc
- Share hardware address space for protection domains
  - Use segment registers to bound protection domains within the hardware address space
  - The hardware will check these registers for all attempts to access protection domains

We can use the hardware defined, segment register bounded protection domains to determine if the TLB is valid for a specific protection domain. If the protection domain is

valid, then we know we don't need to flush the TLB.

## Large protection domains

If the segment boundaries overlap because the protection domains are excessively large, we must conduct a TLB flush - if that TLB is not address space tagged. For large protection domains, the implicit cost is greater than the explicit cost - our cache locality will greatly suffer. **Example:** 864 cycles for TLB flush in a Pentium processor.

## Upshot for address space switching

- **Small protection domains**
  - Can be made efficient by careful construction and implementation.
- **Large protection domains**
  - Switching is not that significant.
  - The loss of locality, cache effects, and TLB effects will dominate in terms of cost.

## Thread switches and IPC

By construction, L3 has shown to be competitive with the SPIN and exokernel architectures. Context switching involves saving all volatile state of a processor.

## Memory effects

What do we mean by memory effects? Not all memory needed to allow a process to execute will be located within physical memory or the cache. The memory effect is essentially,:

- **How warm are caches when we context switch?**

Leidtke suggests packing small protection domains into the physical address space and protect them using segment registers. The cache will be warmer more often because we reduce the implicit cost of address space switching - our protection domains will occupy less space.

You can't avoid the implicit costs imposed by protection domains with large address spaces, even within monolithic, SPIN, or exokernel architectures.

## Reasons for Mach's expensive border crossings

- **Focus on Portability**
  - Code bloat - causes large memory footprint
  - Less locality - more cache misses

All of this causes longer latency for border crossing. Mach's kernel memory footprint is the culprit of Mach's lack of performance. Portability and performance objectives in Mach's design contradict each other.

## Thesis of L3 for operating system structuring

The thesis for the implementation of L3 is one related to how operating systems should be structured. Here are some suggestions from the thesis:

- Minimal abstractions should be implemented by the microkernel.
- Microkernels should be processor-specific in implementation.
- The correct set of abstractions and processor-specific implementation will enable processor-independent abstractions at higher layers of the operating system stack.

## Definitions

Term	Definition
address space tag	tag within a TLB that specifies the process ID generating the translation.
segment register	stores the starting addresses of a data.

## Quizzes

**Why did Mach take roughly 800 or more cycles than L3 for user-kernel switching?**

- L3 uses a faster processor.
- Liedtke is smarter.
- Due to Mach's design priorities.
- Microkernels are slow by definition.

The design priorities of Mach were different than L3. Mach aimed for both extensibility and portability rather than performance.

# paper review questions

## SPIN paper

These review questions are related to the paper, "Extensibility, Safety and Performance in the SPIN Operating System" written by Brian N. Bershad et. al.

### What features of Modula-3 are essential for implementing the extensibility features of SPIN? Why?

- The design of SPIN leverages the language's safety and encapsulation mechanisms, specifically:

Feature	Description
Interfaces	declares the visible parts of an implementation module - all other definitions within the module are hidden. This encapsulation mechanism is enforced at compile-time.
Type safety	prevents code from accessing memory arbitrarily. Pointers may only refer to objects of its referent's type. This safety mechanism is enforced through a combination of compile-time and run-time checks.
Automatic storage management	prevents memory used by a live pointer's referent from being returned to the heap and reused for an object of a different type.

**A strand is an abstraction provided by SPIN for processor scheduling. Explain what this means. In particular, how will a library operating system use the strand abstraction to achieve its desired functionality with respect to processor scheduling? [Hint: Think of the data structures to be associated by the library operating system with a strand.]**

- A **strand** is similar to a thread in traditional operating systems in that it reflects some processor context. Unlike a thread, a **strand** has no minimal or requisite kernel state other than a name.
- The library operating system's thread package defines an implementation of the **strand** interface for its own threads.

**How can SPIN ensure that a given library operating system does not become a CPU hog, shutting out other library operating systems? [Hint: Think of how SPIN can control "macro" resource allocation at start-up time for a library operating system.]**

- SPIN provides core services by default, one of these is the **global scheduler** for extensions. The **global scheduler** uses the different strands of each extension to schedule them for dispatch to the CPU.

**Repeat the above question for physical memory.**

- SPIN provides a **physical page service** that may, at any time, reclaim physical memory by raising the *PhysAddr.Reclaim* event. This interface allows the hanlder of the event to volunteer an alternative page, which may be of less importance than the candidate page. The **translation service** invalidates any mappings to the reclaimed page.

## **Exokernel paper**

These review questions are related to the paper, "Exokernel: An Operating System Architecture for Application-Level Resource Management" written by Dawson R. Engler, et. al.

**A library operating system implements a paged virtual memory on top of Exokernel. An application running in this operating system encounters a page fault. Walk through the steps from the time the page fault occurs to the resumption of this application. Make any reasonable assumptions to complete this exercise (stating the assumptions).**

- The memory management unit raises an exception for the page fault.
- The Exokernel determines which library operating system owns the thread in which the page fault emanated from.
- The Exokernel delivers the page fault to the library operating system for handling.
- The library operating system services the page fault and presents the correct TLB mapping to the Exokernel, along with its key to access memory.
- The Exokernel updates the TLB, installs the mapping of virtual to physical memory, and dispatches the thread of the library operating system onto the CPU.

**"The high-level goals of SPIN and Exokernel are the same." Explain why this statement is true.**

- Both operating system architectures attempt to create highly extensible structures while still maintaining performance and protection. This is evident in their design which establishes a very minimal kernel, allowing for most of the system services to be defined and executed within user-space.

**Map the mechanisms in Exokernel to corresponding mechanisms in SPIN.**

Goal	Exokernel mechanism	SPIN mechanism
Extensibility	ability for library operating systems to download code into the kernel	dynamic binding to different implementations of the same interface functions
Performance	time slot scheduling as a core service	global scheduler for strands as a core service
Protection	securely bind library operating systems to resources	logical protection domains implemented at compile-time

**Explain how the mechanisms in Exokernel help to meet the high-level goals.**

- Secure bindings help to meet the goal of protection.
- Code downloading and the ability to define library operating systems meet the goal of extensibility.
- The core services provided by the Exokernel, time slots for scheduling, revocation, and software TLBs, meet the goal of performance.

**Repeat the above question for SPIN.**

- Logical protection domains, extensions implemented in the strongly-typed Modula-3 programming language, meet the goal of protection.
- CPU scheduling through the strand abstraction and the ability for extensions to be co-located with the kernel meets the goal of performance.
- The ability for developers to create their own extensions using Modula-3 meets the goal of extensibility.

**L3 microkernel paper**

These review questions are related to the paper, "On micro-Kernel Construction" written by Jochen Liedtke.

**What is the difference between a "thread" as defined in Liedtke's paper and a Strand in SPIN?**

SPIN does not define a thread model for applications, but a structure on which an implementation of a thread model rests. A strand has no minimal or requisite kernel state other than a name.

In contrast, Liedtke's determines that a thread is an essential component of a microkernel. The L3 microkernel defines a thread's address space and all changes to a thread's address space must be mediated by the microkernel, enforcing protection.

**Liedtke argues that a microkernel should fully exploit whatever the architecture gives to get a performance-conscious implementation of the microkernel. With this argument as the backdrop, explain how context switching overhead of a microkernel may be mitigated in modern architectures. Specifically, discuss the difference in approaches to solving this problem for the PowerPC and Intel Pentium architectures. Clearly explain the architectural difference between the two that warrant the difference in the microkernel implementation approaches.**

Address space switches are often considered costly. These switches are often costly due to the requirement to flush the TLB. Using tagged TLBs removes the overhead of flushing, since an address space switch is transparent to the TLB.

The microkernel can be designed with an architecture-specific implementation to manage segment registers in the PowerPC and Intel Pentium architectures, effectively creating a tagged TLB. For the Pentium and i486 architectures, address space switches can be optimized by multiplexing many small spaces with a large space.

**Explain the notions of "independence" and "integrity" in the L3 microkernel.**

Notion	Description
Independence	It must be possible to engineer a subsystem in a manner such that it will not be disrupted by the existence of another subsystem.

---

Integrity	The guarantees given by a particular subsystem must be reliable, in that they should not be vulnerable to corruption by another subsystem.
-----------	--

---

**All the subsystems in L3 live in user space. Every such subsystem has its own address space. The address space mechanism "grant" allows an address space to give a physical memory page frame to a peer subsystem. Can this compromise the "integrity" of a subsystem? Explain why or why not.**

The *grant* mechanism implies that there is a sender and receiver conducting IPC to complete the transfer of memory. The onus is on the subsystem designer, the L3 microkernel will not conduct checking of the memory page being transferred via the *grant* mechanism. When the two subsystems initiate IPC, it's possible a malicious or errant subsystem could compromise the integrity of another subsystem by granting a page frame intended to do so.

lesson3

# **lesson3**

# virtualization basics

## Platform virtualization

The main concept described in this slide is using virtualization to provide services to multiple customers using one set of hardware. Every customer will have the same experience with less cost.

Providing virtual platforms to different customers without the customers having to understand the platform being provided. The customer is able to run applications without having to configure the hardware themselves.

## Utility computing

The concept described here is that multiple customers can share resources, but also be able to use a diverse set of operating systems on the hardware resources provided.

The customers are also able to split the cost of maintenance for the hardware, instead of each customer acquiring their own hardware.

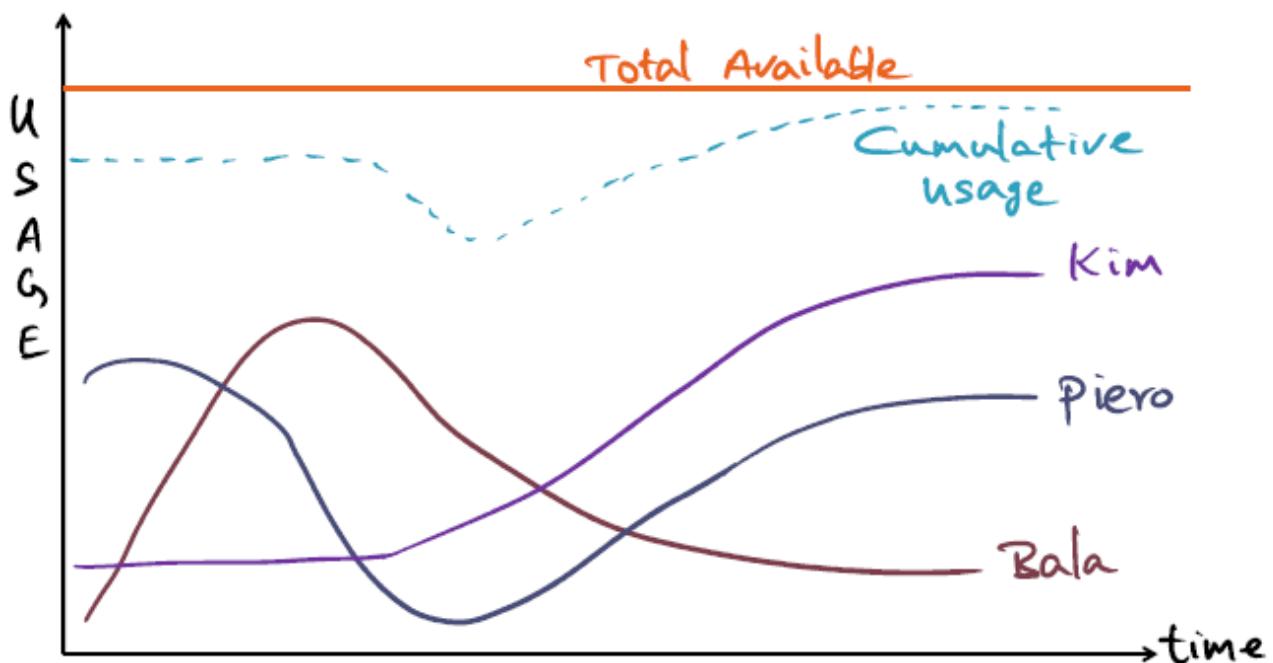
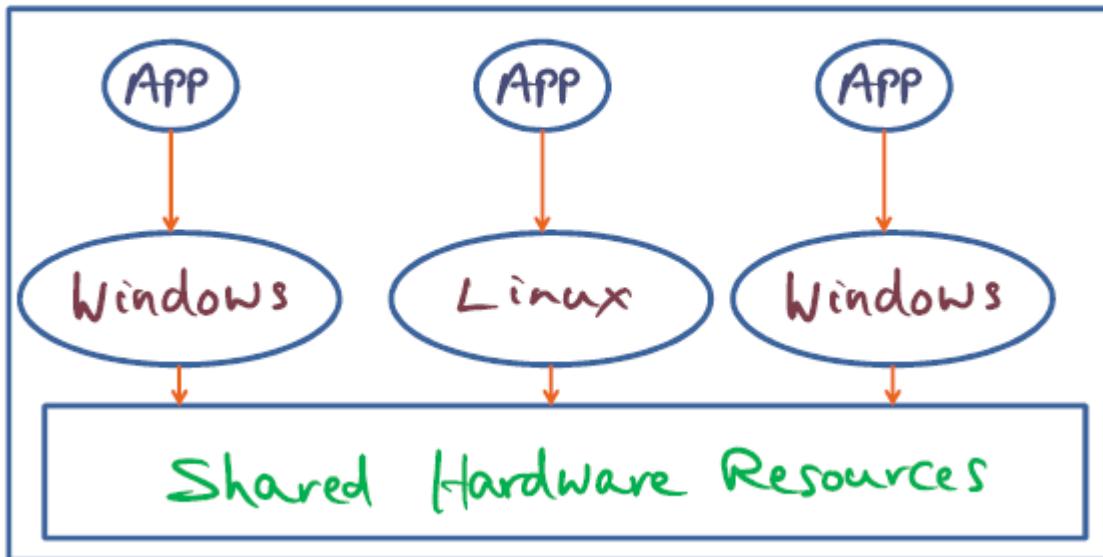
The hardware resources available will always be greater than the individual resource requirements of each customer. This allows the customer to have access to more resources than they would be able to leverage on their own, monetarily.

**Virtualization is extensibility applied at the granularity of an entire operating system, as opposed to the individual services running on a single operating system.**

Below are high-level representations of these concepts.

## Utility Computing

Bala Inc. Piero Inc. Kim Inc. ....



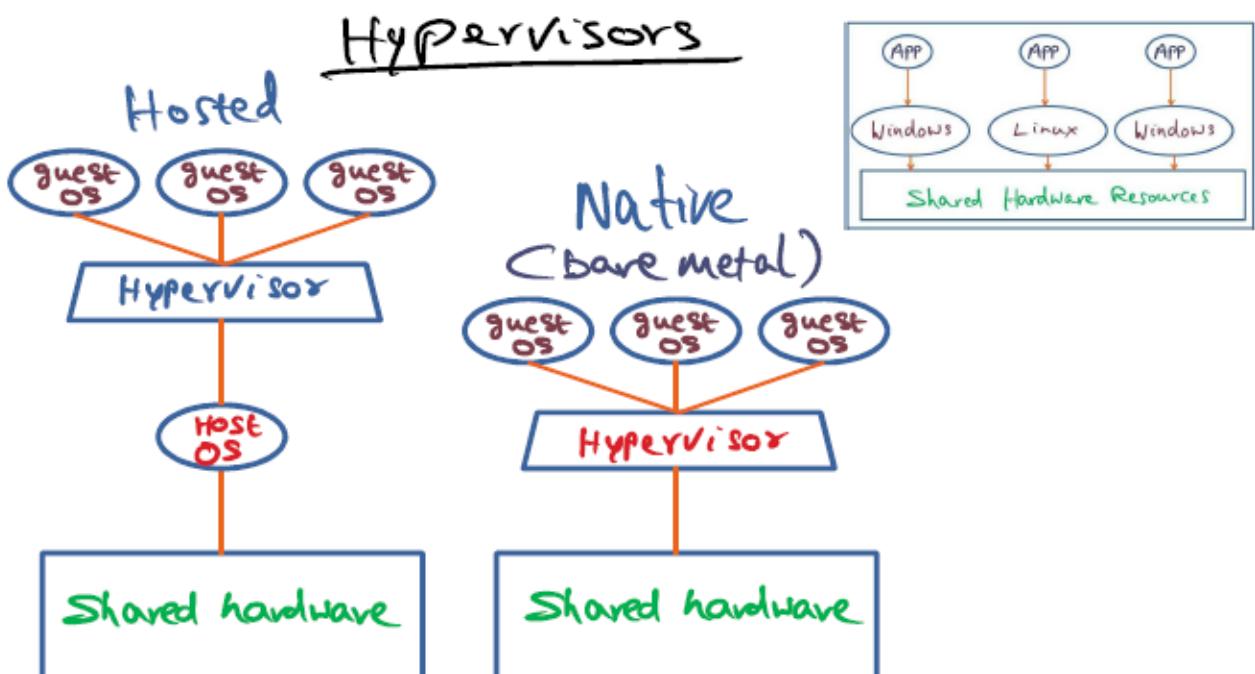
## Hypervisors

Virtual machine managers (or hypervisors) monitor and manage the guest operating systems running on the shared hardware resources.

There are two types of hypervisors:

Hypervisor type	Description
Native hypervisor	<p>running on bare metal, all guest operating systems interact directly with the native hypervisor. Provide the best performance for the guest operating systems.</p> <ul style="list-style-type: none"><li>VMWare ESXi</li></ul>
Hosted hypervisor	<p>running on top of the host operating system, running as an application. The guest operating systems are still clients of the hosted hypervisor.</p> <ul style="list-style-type: none"><li>Technologies that implement hosted hypervisors:<ul style="list-style-type: none"><li>VMWare</li><li>VirtualBox</li></ul></li></ul>

Below is a high-level representation of hypervisors.



## Full virtualization

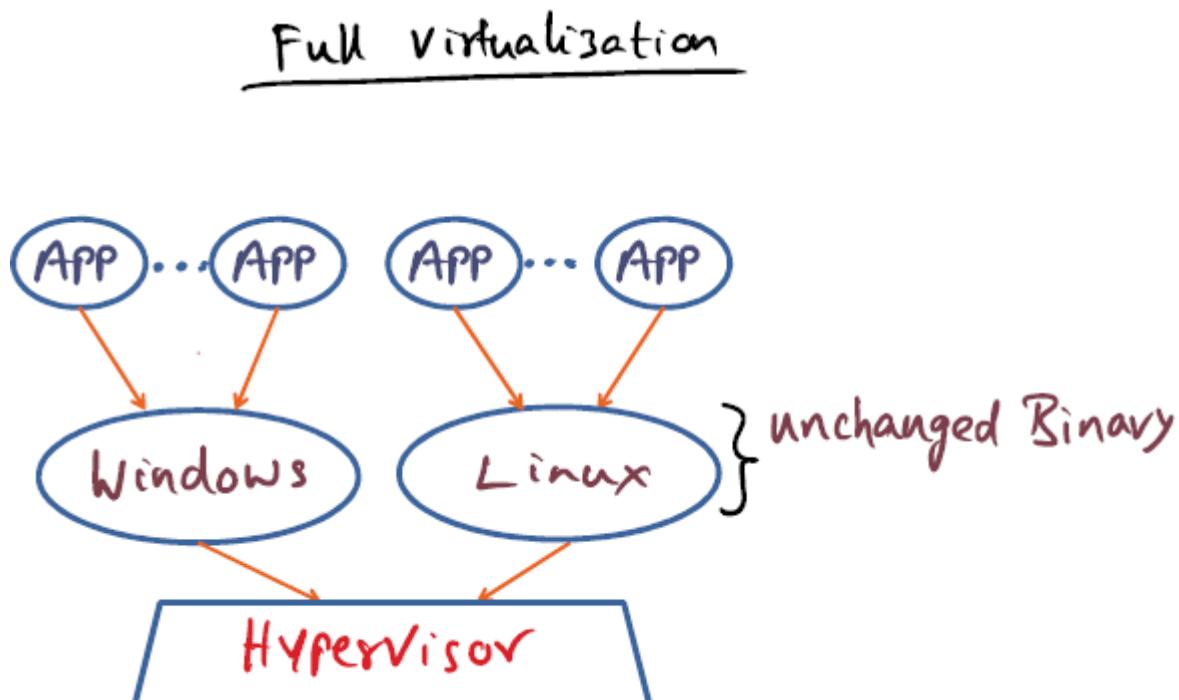
Leaving the guest operating system completely untouched, not a single line of code has been modified for it to interact with the hypervisor.

This requires some finesse, guest operating systems are running as user processes - thus they don't have access to privileged instructions. When these guest operating systems attempt to execute privileged instructions, the hypervisor will have to emulate the intended function and provide the result back to the guest.

Some issues exist with this **trap and emulate** strategy - some privileged instructions will fail silently within the hypervisor; the guest will never know if they succeeded or not.

To fix this, the hypervisor will implement a **binary transition strategy**, looking for instructions that will fail silently on the target hardware and deal with those instructions carefully to catch the issue and take appropriate actions.

Below is a high-level representation of a fully virtualized virtual machine.



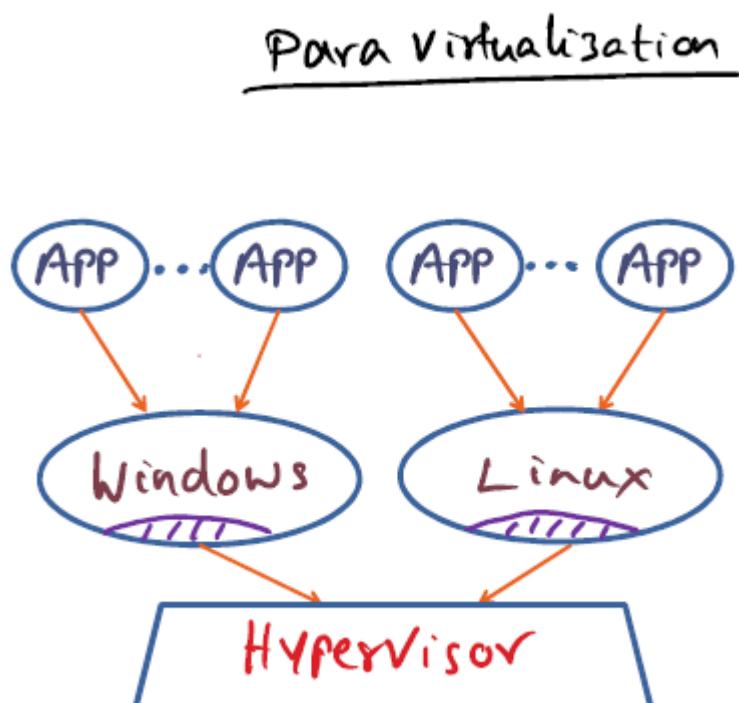
## Para virtualization

Another approach to virtualization is to *modify* the source code of the guest operating system. This allows us to avoid problematic instructions, but also allows us to implement optimizations.

Para-virtualized operating systems will be aware of the fact that they are guests within a hypervisor, and this will be knowledgeable about the hardware instructions directly available for them to use.

Applications will still only be able to use the API of guest operating system that it is running on, but the operating system itself will utilize para-virtualization to optimize its interaction with the hardware.

Below is a high-level representation of a para-virtualized virtual machine.



## Definitions

Term	Definition
hypervisor	computer software, firmware or hardware that creates and runs virtual machines

---

virtual machine	an emulation of a computer system
--------------------	-----------------------------------

---

## Quizzes

**What comes to your mind when you hear the word "virtualization"?**

- Memory systems
- Data centers
- JVM
- Virtual Box
- IBM VM/370
- Google glass
- Cloud computing
- Dalvik
- VMWare Workstation
- The move "Inception"

**What percentage of guest operating system code may need modification with para-virtualization?**

- ~10%
- ~30%
- ~50%
- less than 2%

Less than 2% of the guest operating system code needs to be modified in order to enable para-virtualization. This is proven by Xen and their experiments with para-virtualization.

# memory virtualization

## Introduction

What needs to be done in the system software stack to support virtualization? The question boils down to:

- How do we be flexible, performance conscious, and safe while implementing virtualization?

Memory hierarchy for virtualization is crucial to performance. Even efficient handling of device virtualization is heavily reliant on how the memory system is virtualized and made available to guest operating systems.

## Memory hierarchy

The biggest issue here is handling virtual memory, namely the virtual address to physical memory mapping. This is part of the key functionality of memory management for any operating system.

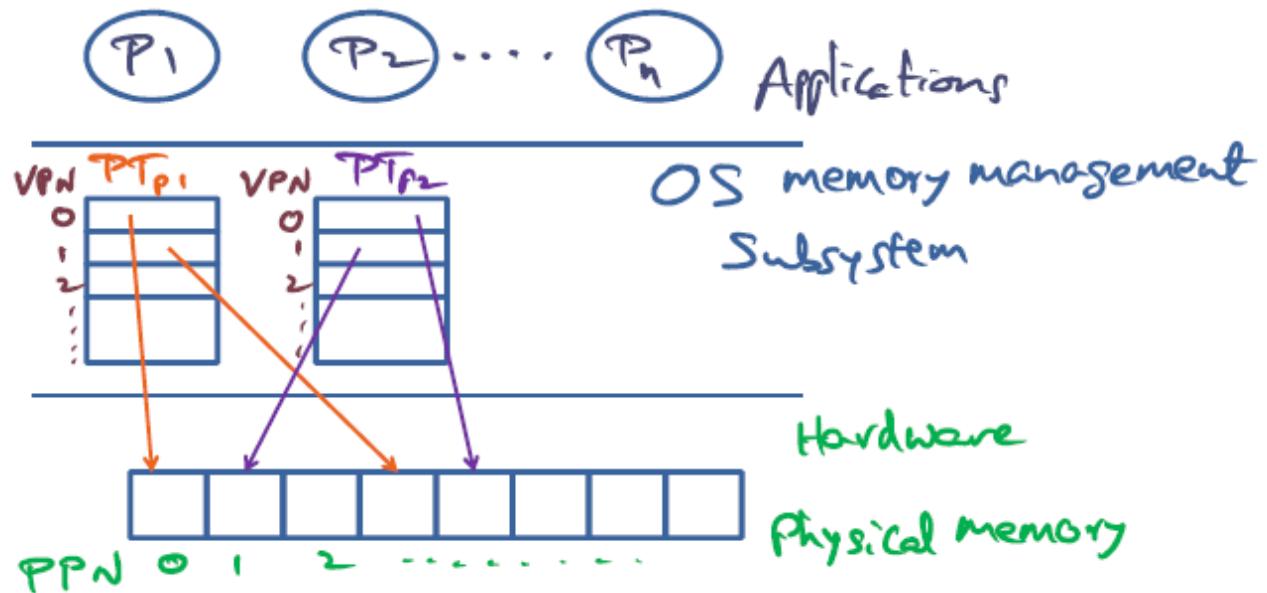
## Memory subsystem recall

Each process is contained within its own **protection domain**. The operating system manages a page table for each process, and each process contains its own listing of virtual page numbers that are contiguous in its view.

The virtual address space for a given process is not contiguous within the physical memory, but scattered across it. When a process utilizes a virtual address, a translation is conducted using the page table to reference a location within physical memory.

Below is a high-level representation of the concepts described above.

## Memory Subsystem Recall



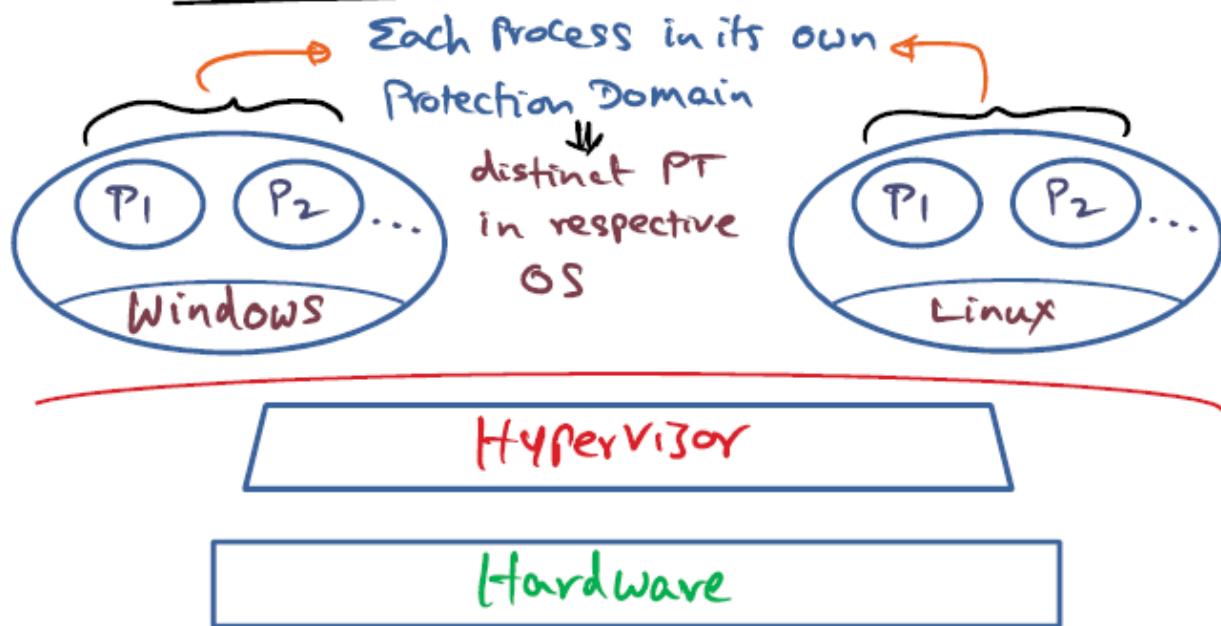
## Memory management and hypervisor

In this case, each guest operating system is considered its own process and **protection domain**. Each operating system within its own protection domain contains their own paging system for each process running inside of the guest operating system.

The hypervisor does not track the page tables for each of the applications running inside of each guest operating system. The hypervisor is only concerned with the page table of each **protection domain** (guest operating system).

Below is a high-level representation of the concepts described above.

## Memory Management and Hypervisor



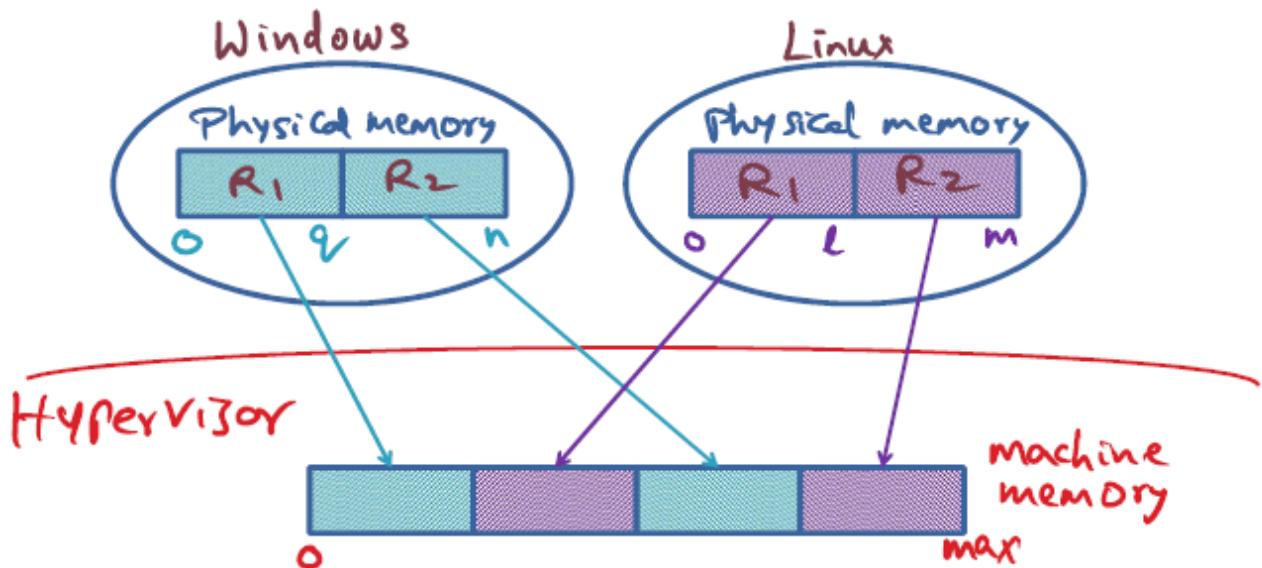
### Memory manager zoomed out

Guest operating systems will consider their physical memory contiguous, but in the *real* physical memory (**machine memory**) the guest operating systems are not stored in a contiguous manner. Their memory is also virtual and mapped to physical memory by a page table managed by the hypervisor.

The memory requirements of the guest operating systems are also **bursty**, randomly requiring large amounts of memory but also releasing it later. They hypervisor may not be able to provide another contiguous region of memory to a guest, making the continuity of memory within the guest operating system an illusion.

Below is a high-level representation of the concepts described above.

## Memory Manager - zoomed out



## Zooming back in

In a virtualized setting, there exists 3 levels of indirection. The hypervisor will maintain **machine page numbers** (MPN) that will be utilized to conduct memory translations between the guest operating systems and the host's memory resources.

This is what the memory hierarchy looks like for a virtualized system from most abstract to least abstract:

### Abstraction      Definition

Virtual page numbers      page numbers of processes running inside a virtual machine; translated to physical page numbers using the guest operating system's page table

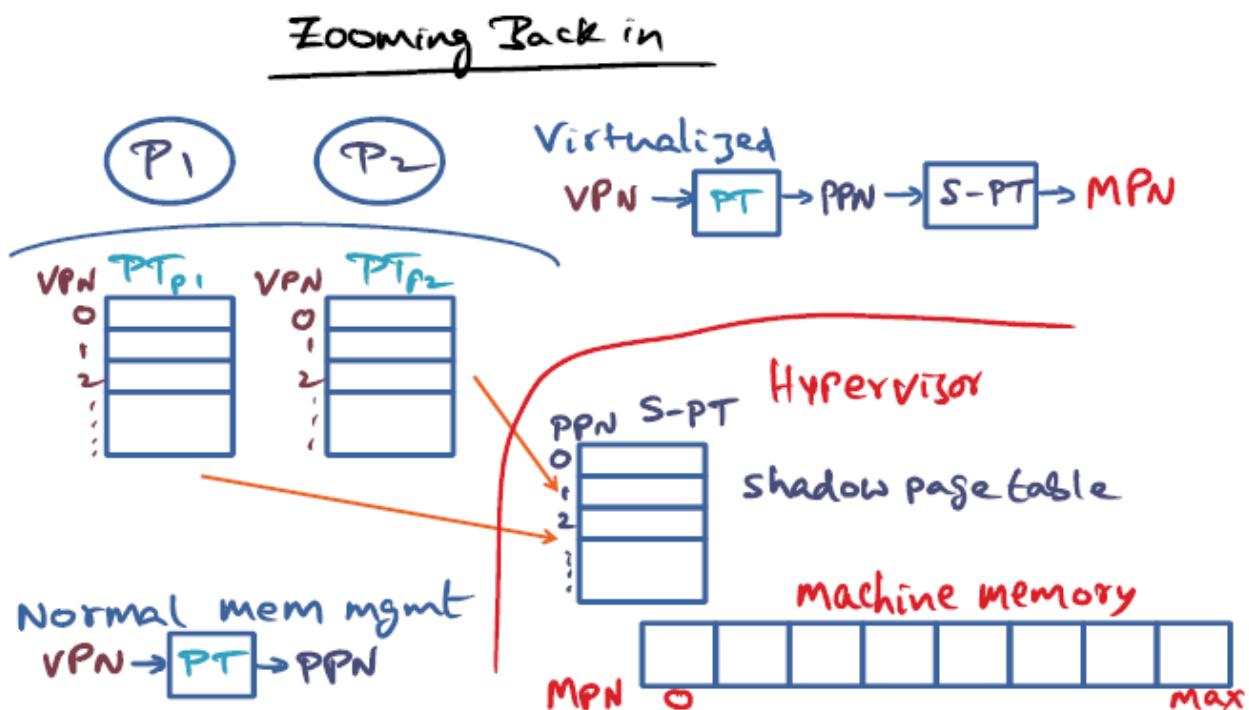
Physical page numbers      pages managed by guest operating systems; translated to machine page numbers using the hypervisor's shadow page table

---

Machine  
page numbers      pages managed by the hypervisor

---

Below is a high-level representation of the concepts described above.



## Shadow page table explained

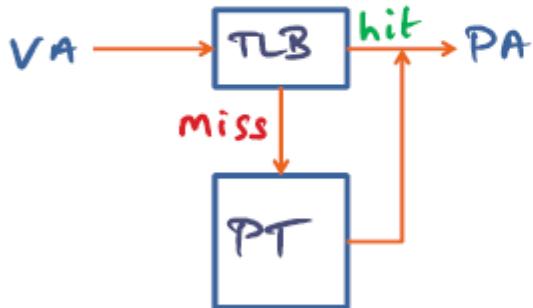
Usually, the CPU uses the page table for address translation. The CPU will receive a virtual address, access the **Translation Lookaside Buffer** (TLB) to see if translation is already cached. If it is a *hit*, the translation is essentially immediate. If it is a *miss*, the CPU will consult the page table for the virtual to physical address translation. Once this is complete, it will stash the virtual to physical translation in the TLB.

In virtualized settings, the hardware page table *is* the shadow page table. Below is a high-level representation of the concepts described above.

## Shadow Page table

In many architectures (e.g., x86)

— CPU uses Page Table for address translation



⇒ hardware PT is really the S-PT

## Efficient mapping (full virtualization)

How do we make the virtual to physical to machine address translation process efficient?

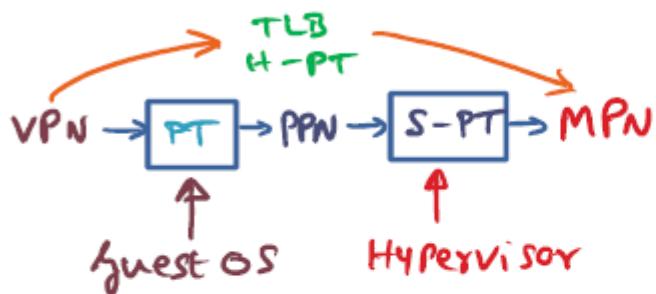
All changes to the page table by the guest operating system are privileged accesses. These instructions will be trapped by the hypervisor, and the hypervisor will proceed to update the same mapping in the **shadow page table**. The translations that occur from virtual to physical are stored in the hardware **TLB** and hardware page table.

This bypasses the guest operating system's page table - each time a process generates a virtual address the translation is conducted using the hardware page table and TLB.

This allows us to not have to use the guest operating system for every virtual address space translation, prior to the translation being conducted between the guest operating system and the hypervisor.

Essentially, the hypervisor tracks the memory locations of every unprivileged process running in every guest operating system. Below is a high-level representation of the concepts described above.

## Efficient Mapping (Full virtualization)



How to make this efficient?

- PT/TLB updates of guest OS trapped
- S-PT updated by hypervisor
- translations installed into TLB/hardware PT

## Efficient mapping (para-virtualization)

The burden of efficient mapping is placed upon each guest operating system. Each guest operating system maintains its contiguous "physical memory" and also knows where it exists with the machine memory.

The burden of VPN to PPN to MPN mapping is all handled by the guest operating system.

For example, in Xen, a set of **hypercalls** are provided to the para-virtualized guest operating systems to tell the hypervisor about changes to the hardware page table. This allows the guest operating system the ability to allocate and initialize a page table data structure within the memory originally allocated to it by the hypervisor.

A **switch hypercall** is also provided for the guest operating system to switch contexts to a different user mode application. The hypervisor is unaware of the switches, it just handles requests made by the guest operating system.

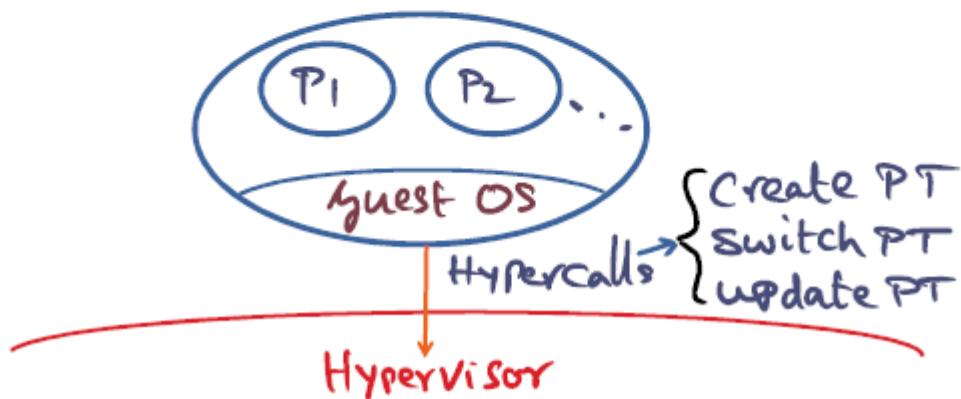
Updating page tables is also provided, the guest operating system notifies the hypervisor what modifications needs to be made to the page table data structure. Below is a high-level representation of the concepts described above.

## Efficient Mapping (Para virtualization)

Shift Burden to Guest OS

- maintain contiguous "physical memory"

- map to discontiguous hardware Pages

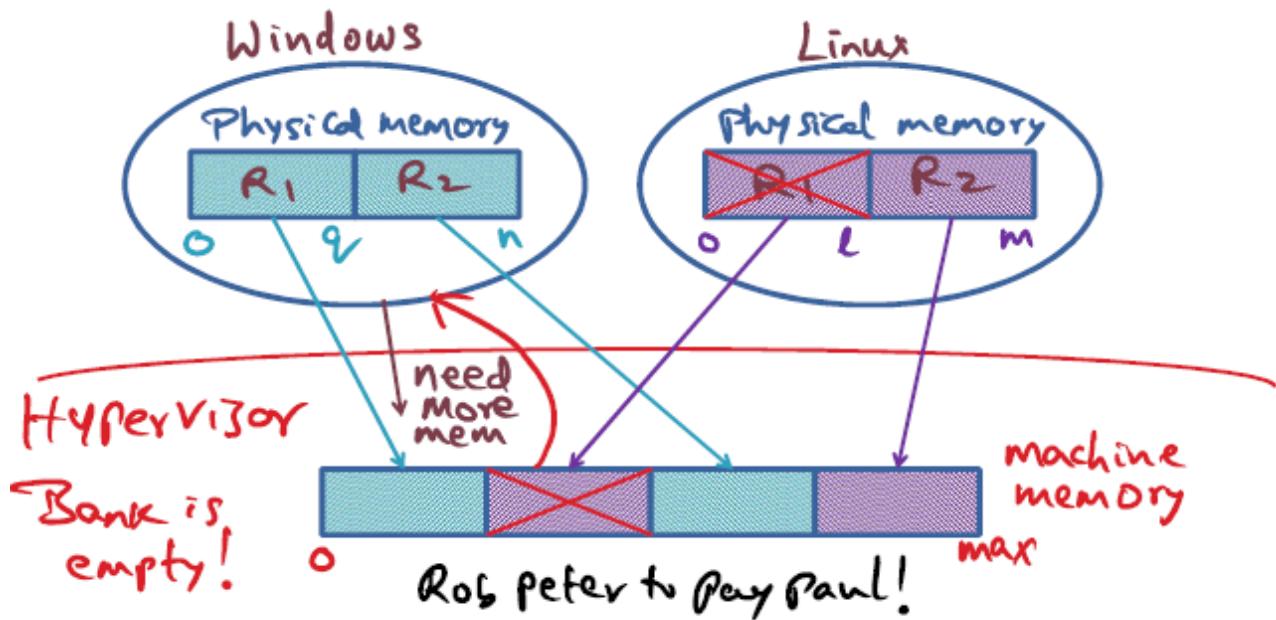


## Dynamically increasing memory

How do we dynamically increase the amount of physical memory allocated to each guest operating system that needs it?

Essentially we monitor the memory requirements of each operating system and reap memory from operating systems not utilizing their full memory space. Then we provide that memory to operating systems currently experiencing memory pressure. Below is a high-level representation of the concepts described above.

## Dynamically Increasing Memory



## Ballooning

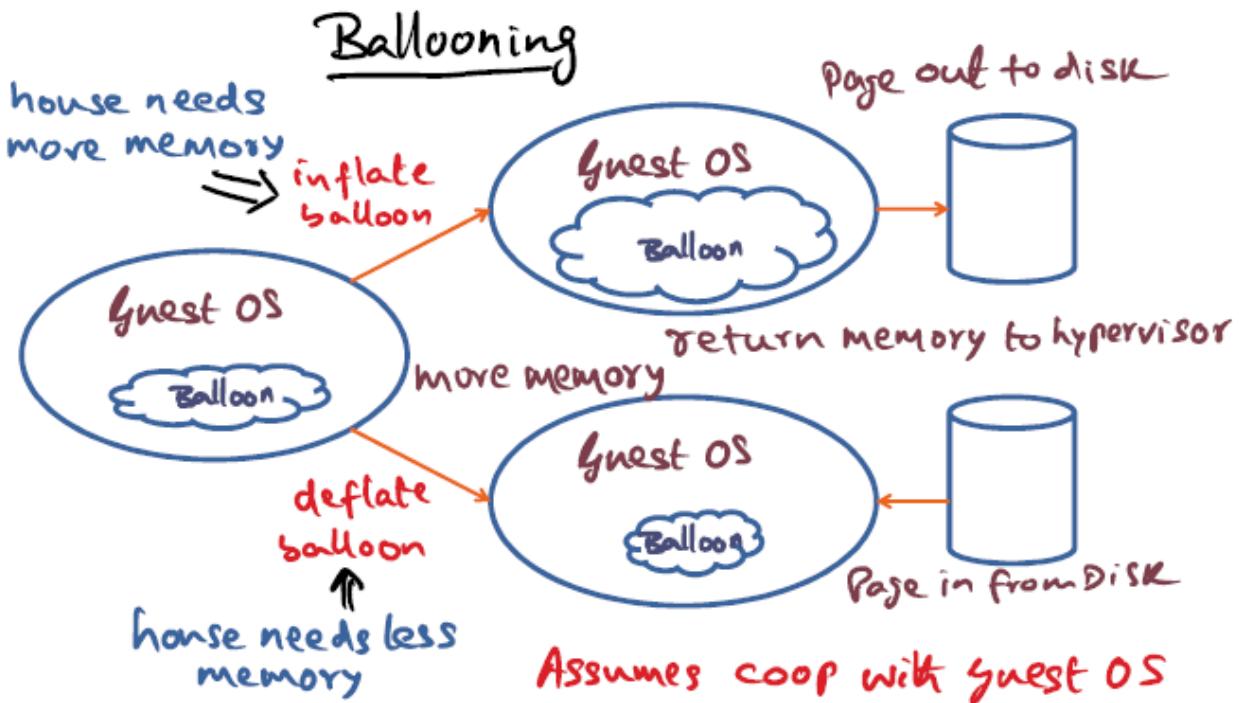
A special device driver is installed in every guest operating system (despite full or para virtualization). This device driver is called a **balloon**. This driver is the key for managing memory pressures that may be experienced by guest operating systems.

Let's say the hypervisor needs more memory, suddenly. The hypervisor will contact operating systems that are currently not utilizing all of their memory and interact with the **balloon** driver. The hypervisor instructs the balloon device driver to begin inflating - the balloon device driver will begin requesting memory from the guest operating system.

This forces the guest operating system to begin paging memory to disk in order to satisfy the requirements of the **balloon** driver. Once the **balloon** driver has acquired all of the requested memory, the **balloon** will return the memory back to the hypervisor.

So what if the guest needs more memory and the hypervisor has excess memory? The hypervisor will contact the **balloon** driver of the guest operating system and instruct it to deflate - contracting its memory footprint. Through this, it is actually releasing memory into the guest operating system. This allows the guest operating system to page in the processes that were previously paged out to the disk.

This technique assumes that the guest operating system and the **balloon** drivers coordinate with the hypervisor. Below is a high-level representation of the concepts described above.



## Sharing memory across virtual machines

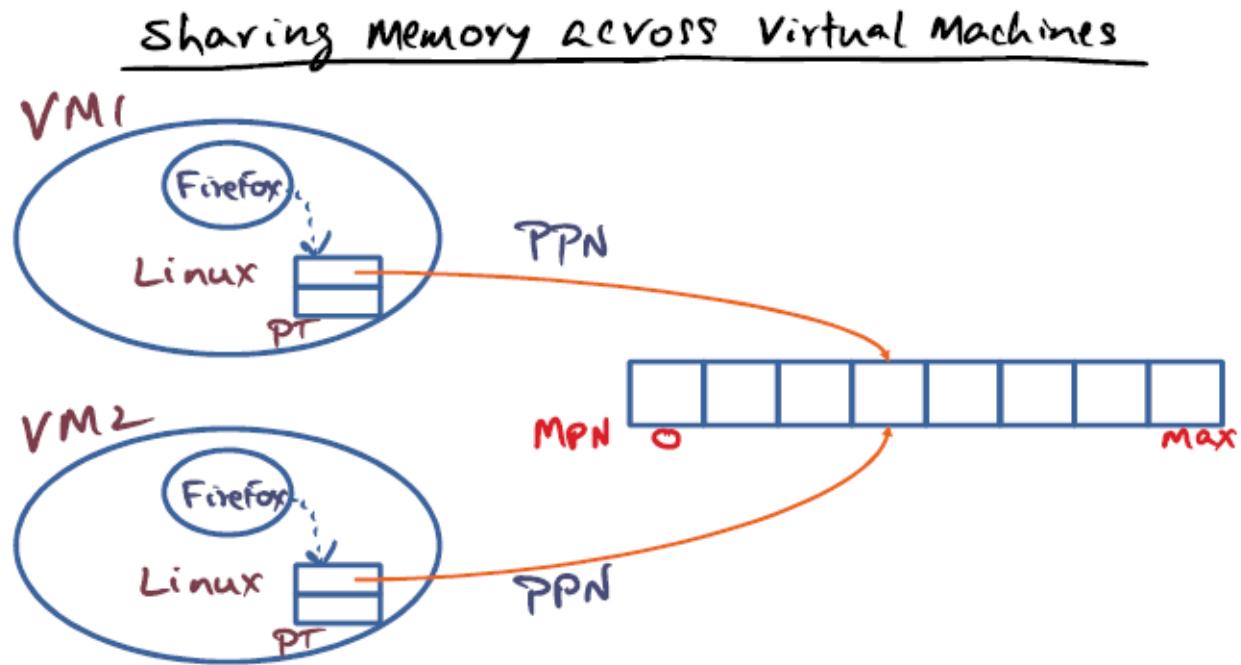
Can we share memory across virtual machines, without affecting the integrity of the protection domains that exist between each guest operating system?

Yes, we can!

Based upon the **similarity** of virtual machines existing within a hypervisor, we can avoid duplication by allowing guest operating systems to point to the same locations in memory. Of course core pages are going to be separate, but applications that are similar can share the same locations within the machine memory.

One method is by having the virtual machines and the hypervisor to cooperate in sharing. The guest operating system has hooks to allow the hypervisor to mark pages as **copy on write** - when a guest operating system writes to a page, the other guest will fault and utilize a different page. All things are fine when the guest operating systems are reading a page, but as soon as a write occurs, the guest operating systems must separate and begin

referencing to different pages. Below is a high-level representation of the concepts described above.



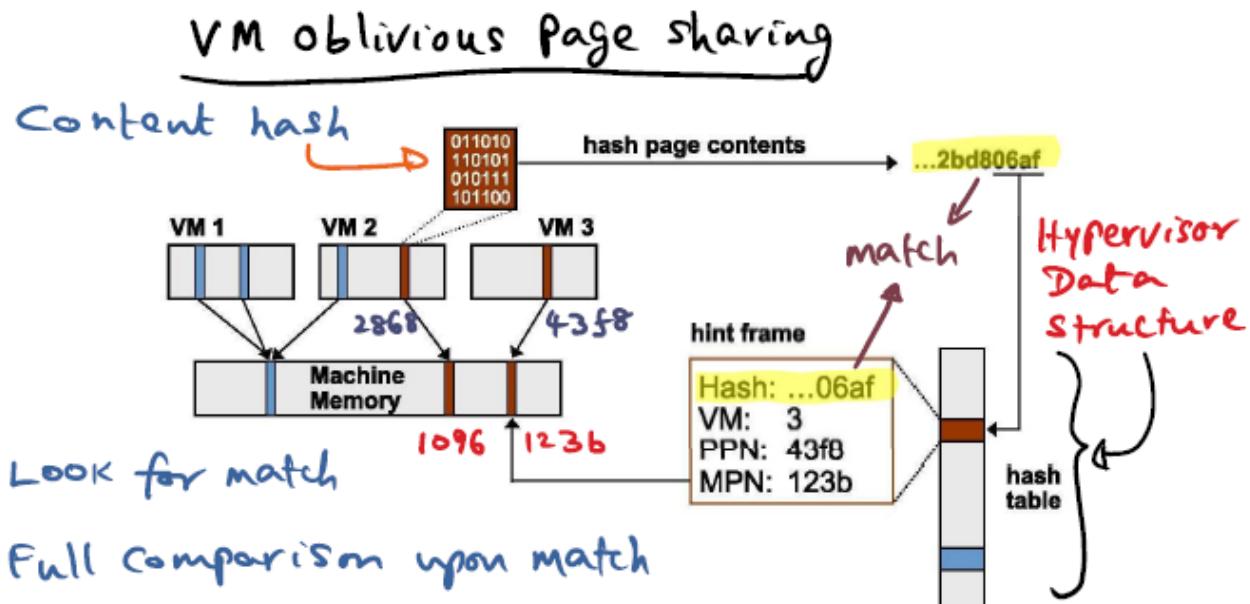
## VM oblivious page sharing

So now we're trying to achieve the same effect of memory sharing, as annotated above, but the guest operating systems will be completely oblivious to the fact they are sharing memory locations.

The technique that is about to be described is used by VMWare's ESX server implementation. The central idea is that the hypervisor conducts **content-based** sharing. VMware manages a hash table data structure within the hypervisor and each location within the hash table contains a content hash of the memory being used by a guest operating system.

If a page in a different guest operating system, content-wise, is the same the page of a different operating system, the guest operating systems will be oblivious to the fact that they are sharing the same page in machine memory. The hypervisor maintains hashes of the content of each page for every guest operating system, looks for matches, and utilizes this as a hint as to whether or not we can have these guest operating systems share locations in memory.

If we have a match, the hypervisor conducts a full comparison of the guest operating system's memory footprints. This is done because a hash match is not an absolute indicator that the memory footprints of two guest operating systems are exactly the same. The content in both memory locations could've changed since the last check. Below is a high-level representation of the concepts described above.



## Successful match

If the two different memory locations contain the same content, the hypervisor will modify the PPN → MPN mapping for one of the guest operating systems, mapping the PPN address to the same MPN address as the other operating system. Now, when a PPN → MPN translation is conducted for either guest operating system, they both will point to the same machine memory location.

The reference count within the hash table entry for the specific hash will be incremented. This mapping from PPN → MPN will not be marked as **copy on write** - thus if the guest operating systems effect the integrity of the memory location, the guest operating systems will copy their content into different memory locations and separate their usage.

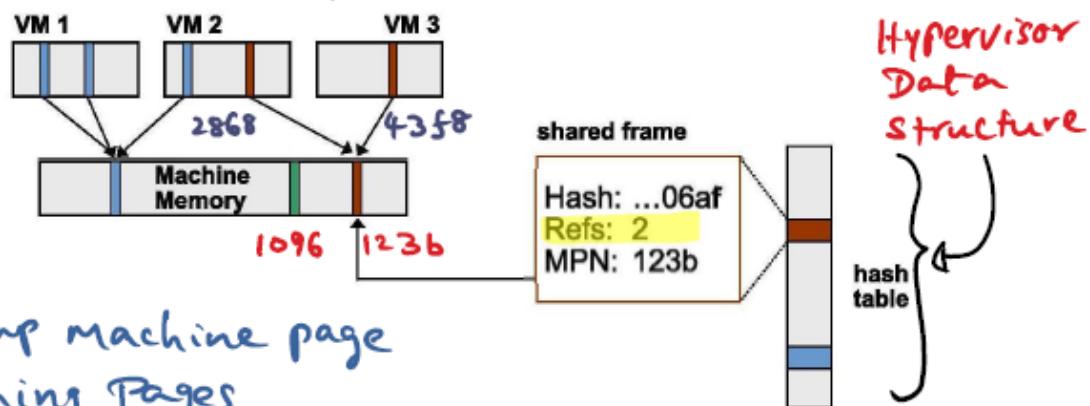
This should not be done when there is active usage of the system - this is computationally expensive. This is a function of the **background activity of the server** when a low load is experienced.

This technique is applicable to both fully and para-virtualized systems.

Below is a high-level representation of the concepts described above.

### Successful match

- modify PPN → MPN for VM2
- mark as copy on write



Free up Machine page

Scanning Pages

- background activity of server

## Memory allocation policies

Different policies for memory allocations:

Policy	Description
Pure share based approach	what you pay for is what you get. Problem with this is that this policy could lead to hoarding.
Working-set based approach	if a working set of a guest operating system increases, more memory is allocated. If a low load is experienced, memory is released.
Dynamic idle-adjusted shares approach	tax idle pages more than active pages. Hoarders will be taxed more memory if they're not using it. People using a majority of their memory will be taxed less.

---

Reclaim most idle memory	we reclaim idle memory from all virtual machines that are not actively using it. This allows for sudden working set increases that might be experienced by a particular domain.
--------------------------	---

---

## Definitions

Term	Definition
Machine page number	Page numbers maintained with the shadow table by the hypervisor; actual physical memory in a virtualized environment
Balloon driver	A special device installed in every guest operating system utilized to reclaim or allocate memory to a guest operating system.
Shadow page table	Page table maintained by the hypervisor in fully virtualized systems that contains virtual to machine page mappings
Hypercall	Utilized by guest operating systems in a virtualized environment to request a privileged action be taken on behalf of the guest operating system by the hypervisor.

## Quizzes

### Who keeps PPN -> MPN mapping in a fully virtualized system?

- Guest Operating System
- Hypervisor

In the case of a fully virtualized system, the guest operating systems have no knowledge that they are being virtualized. Therefore, the hypervisor must maintain the **shadow page table**.

### Who keeps PPN -> MPN mapping in a para-virtualized system?

- Hypervisor

Guest Operating System

Para-virtualized systems know the virtualization exists. Therefore, it knows its physical memory will be fragmented in the machine memory. Usually, the PPN  $\rightarrow$  MPN mapping is kept in the guest operating system.

# cpu and device virtualization

## Introduction

The challenge that exists when virtualizing CPUs and devices is giving the **illusion** to each guest operating system that they own the CPU. That is, each guest operating system doesn't know that other guests exist running on the same CPU.

This **illusion** is being provided by the hypervisor at the *granularity of the entire operating system*.

The hypervisor must field events arising due the execution of processes that belong to a guest operating system. There are going to be **discontinuities** or exceptions that will occur, and the hypervisor must ensure that the correct guest operating system handles these instances.

## CPU virtualization

Each guest operating system is already scheduling the processes that it currently hosts onto the CPU. This is great, however, the hypervisor sits in between each guest operating system and the CPU. The hypervisor must provide the **illusion** to each guest operating system that the guest owns the CPU. This will allow each guest OSs the ability to schedule its processes accurately on the CPU.

The hypervisor must have a precise way in keeping track of the time a particular guest OS uses on the CPU. This would be from the point of view of billing the different customers (guest operating systems).

### Methods of scheduling guest operating systems onto the CPU:

- Proportional share
- Fair share

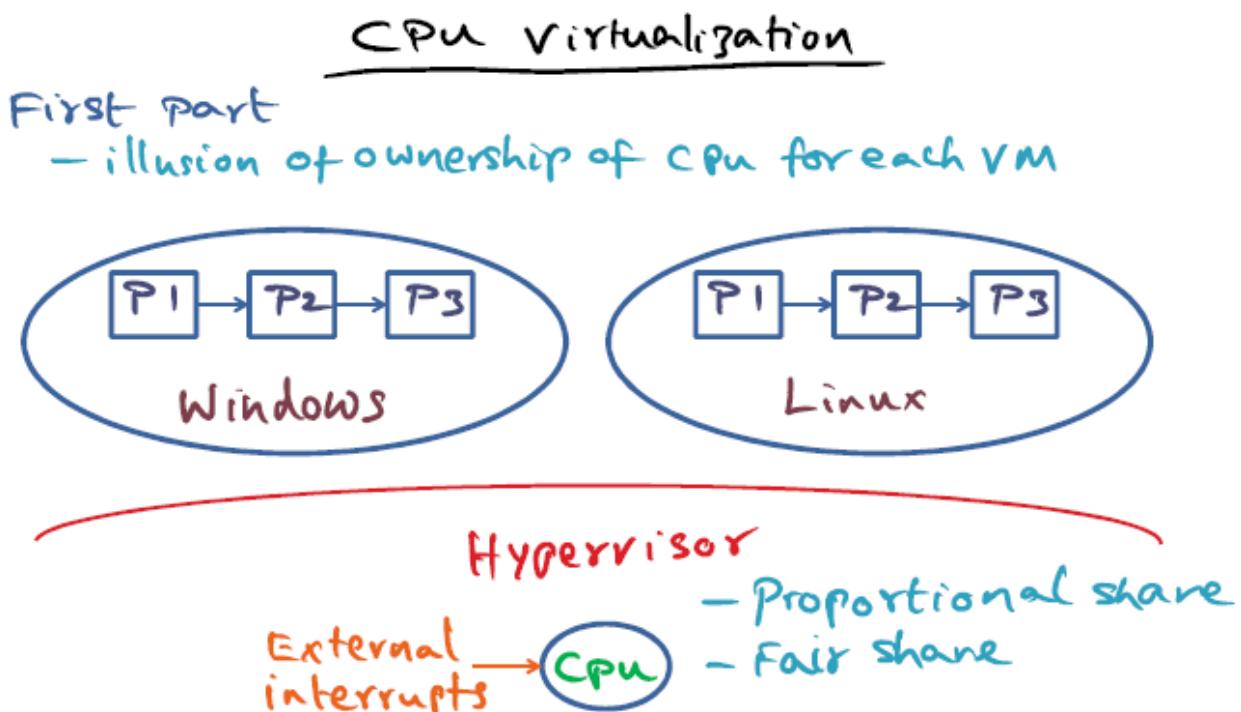
In either of these cases, the hypervisor has to account for the time used on the CPU on behalf of a different guest during the time allocated for a particular guest. This can happen

if an external interrupt occurs that is intended for a guest operating system while a process for a different VM is currently running on the CPU.

The hypervisor will essentially track the stolen time of a guest operating system used by other guest operating systems. The hypervisor will **reward** those VMs that weren't allowed to run due to some extenuating circumstance. The challenge that exists when virtualizing CPUs and devices is giving the **illusion** to each guest operating system that they own the CPU. That is, each guest operating system doesn't know that other guests exist running on the same CPU.

This **illusion** is being provided by the hypervisor at the *granularity of the entire operating system*.

The hypervisor must field events arising due the execution of processes that belong to a guest operating system. There are going to be **discontinuities** or exceptions that will occur, and the hypervisor must ensure that the correct guest operating system handles these instances. Below is a high-level representation of the concepts described above.



Second part of CPU virtualization (common to full and para)

The hypervisor must be able to deliver events to the parent guest operating system. These events are events generated by a **process** that belongs to the guest operating system currently executing on the CPU.

So what are the things this process can do that might require intervention and notification to the guest operating system?

- One example would be the opening of a file. This is a **privileged** instruction that will be handled by the guest operating system.
- Another example could be the process incurring a page fault. Maybe not all of its virtual address space is currently mapped to machine memory.
- Another example would be the creation of an *Exception*.
- Another example could be an external interrupt.

These are all called program **discontinuities** that can interrupt the normal execution of a process. All of these **discontinuities** have to be passed up, through the hypervisor, to the parent of the process (the guest operating system).

These events will be delivered as software interrupts to the guest operating system.

Some quirks exist based upon the architecture, and the hypervisor may have to deal with this. Some of the operations that the guest operating system may have to do to deal with these **discontinuities** may require the guest operating system to have **privileged** access to the hardware.

This presents a problem, especially in a fully virtualized environment - the guest operating system is unaware that it's virtualized. When the guest operating system conducts the **privileged** operation to remediate a **discontinuity**, the **privileged** instruction will **trap** into the hypervisor. Unfortunately, some **privileged** instructions *fail silently*, the guest operating system will have no knowledge that this instruction failed or never reached the hypervisor.

The hypervisor will have to determine, for fully virtualized systems, where these quirks may exist and edit the binary of the guest operating system in order to catch these instructions.

In a fully virtualized environment, all interactions between the guest operating system and the hypervisor are **implicitly** conducted via **traps** for system calls.

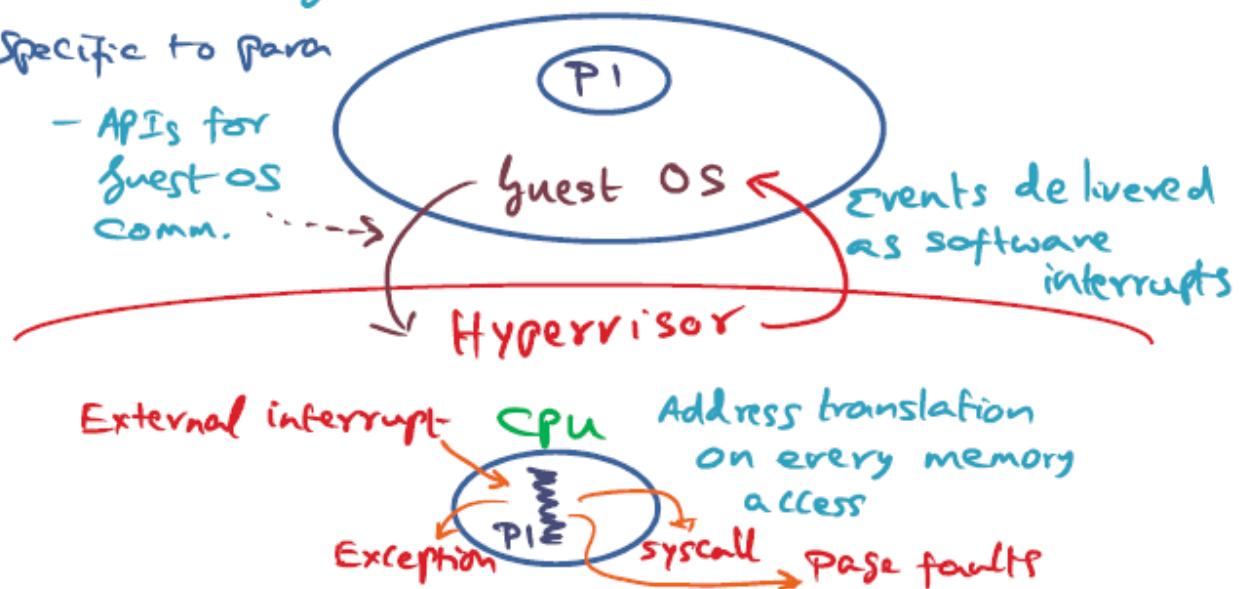
In a para-virtualized environment, the guest operating system is provided APIs in order to ask the hypervisor to conduct some **privileged** instruction. This is evident when we discussed how para virtualized operating systems conduct management of their page tables for different user level processes. Below is a high-level representation of the concepts described above.

Second part (common to full + para)

— Delivering events to Parent Guest OS

Specific to para

- APIs for guest os comm.



## Device virtualization intro

The next issue is virtualizing devices, giving the **illusion** to the guest operating systems that they own particular I/O devices.

## Full virtualization for devices

- **trap and emulate** all accesses to the those devices. The guest operating system does not know it is currently virtualized.

## Para-virtualization for devices

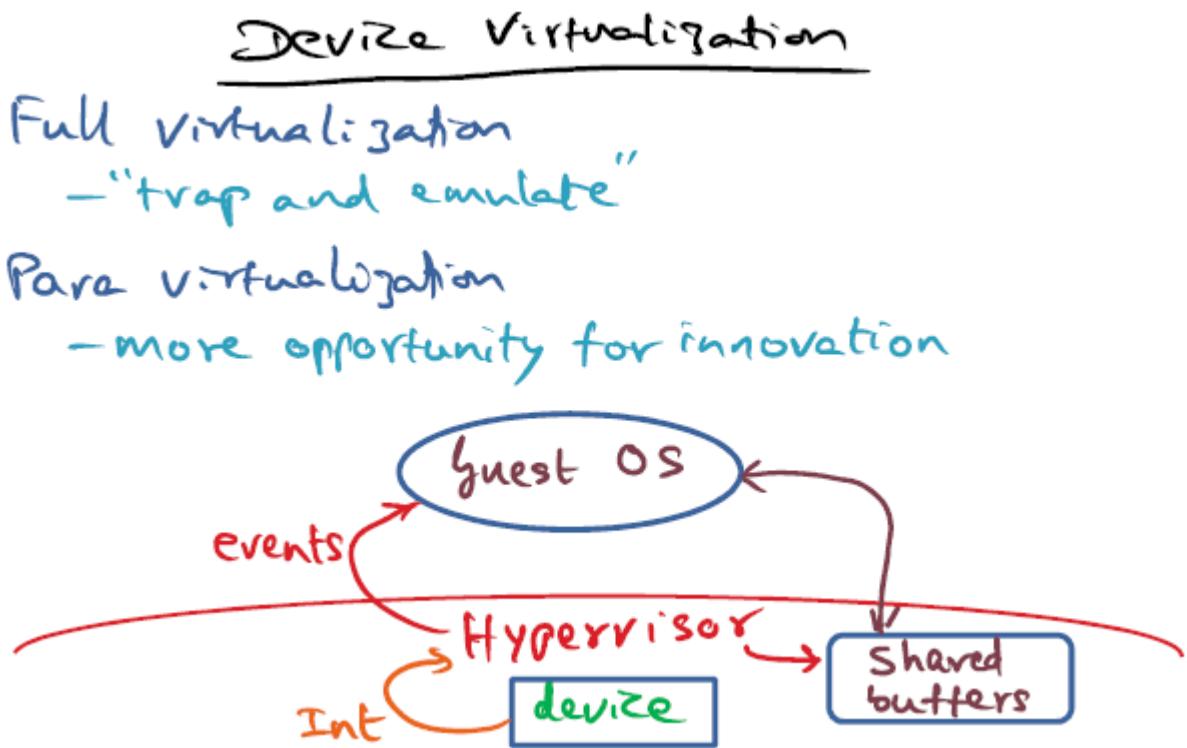
- **more opportunity for innovation** - the para-virtualized system is aware of the devices available for it to use. The set of hardware devices that are available to they hypervisor

are also available to the guest operating system. The hypervisor has the ability to provide an API for the guest operating system to use - providing some level of transparency and optimization.

Some things we have to worry about:

- How do we transfer control of the devices between the hypervisor and the guest?
- How do we transfer data from the devices to the correct recipient?

Hardware devices require manipulation by the hypervisor in a privileged state. The hypervisor is in a different protection domain compared to the guests. Below is a high-level representation of the concepts described above.



## Control transfer

- **Full virtualization:**
  - implicit traps by the guest -> handled by the hypervisor
  - software interrupts are served from the hypervisor -> guest
- **Para virtualization:**
  - explicit hypercalls guest -> hypervisor (results in control transfer)

- software interrupts are generated from the hypervisor -> guest
  - the additional facility that exists here is that the guest has control via hypercalls on when event notifications need to be delivered from the hypervisor.
  - similar to an operating system disabling interrupts.

## Data transfer

- **Full virtualization**
  - data transfer is, again, implicit
- **Para Virtualization**
  - explicit -> provides an opportunity to innovate

Two aspects exists for data transfer:

- CPU time, the hypervisor has to de-multiplex the data to the correct owners upon receiving an interrupt from a device. Hypervisor must account for the computational time it took to manage the buffers on behalf of the virtualized operating systems.
- The second aspect is how the memory buffers are managed. How are they allocated either by the guest or by the hypervisor?

In Xen, to conduct data transfers and requests, the guest operating systems define a ring buffer type structure in which they produce and place their I/O requests.

Xen will consume the request, process the request, acquire the data from the requested device, and provide the data back to the guest via the ring.

Below is a high-level representation of the concepts described above.

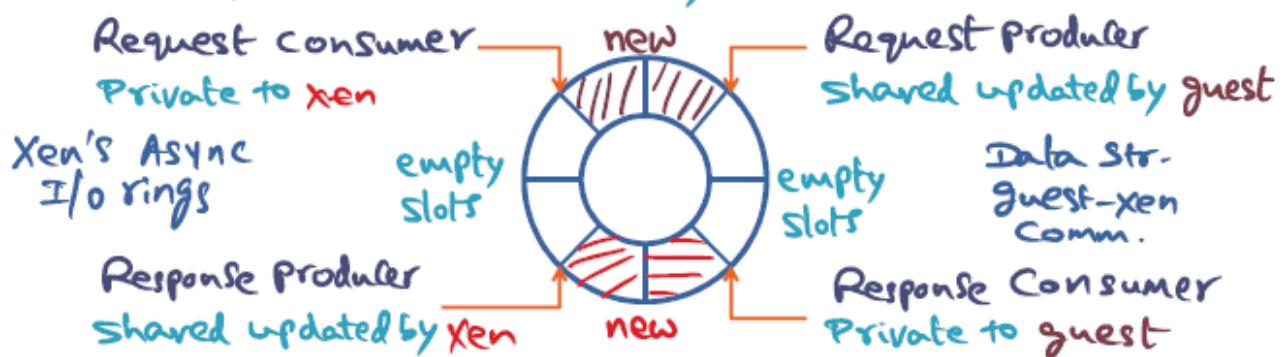
## Data Transfer

### Full Virtualization

- implicit

### Para Virtualization (e.g., Xen)

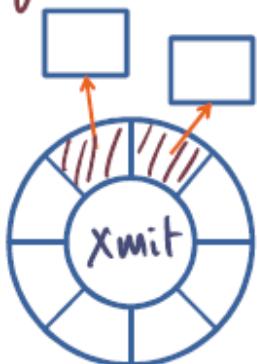
- explicit  $\Rightarrow$  opportunity to innovate



## Control and Data Transfer in Action

### Network Virtualization

guest os buffers



Transmit

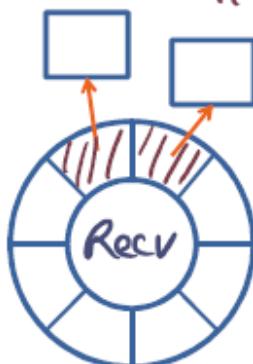
- descriptors enqueued
- buffers in guest OS
- no copying
- page pinned

RR PKT scheduler  
in xen

Receive

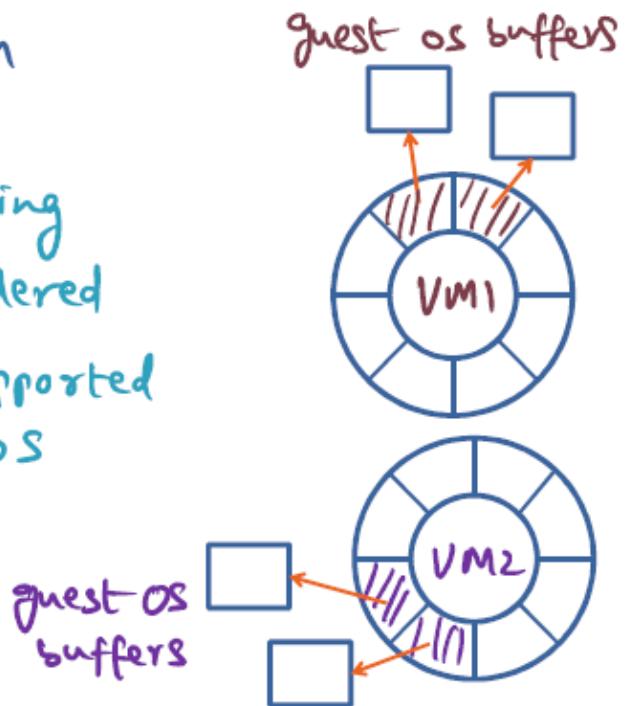
- exchange received packet for guest OS page
- no copying

guest os buffers



## Disk I/o Virtualization

- No copying into xen
- Request from competing domains may be reordered
- Reorder barrier supported by xen for guest OS semantics



## Measuring time

The whole concept is having good ways of measuring these resources for accurate billing:

- CPU usage
- Memory usage
- Storage usage
- Network usage

Virtualized environments need to have mechanisms in place to store records for each one of these metrics for customers.

## Definitions

Term	Definition
Ring buffer	a data structure that uses a single, fixed-size <b>buffer</b> as if it were connected end-to-end

## **Quizzes**

# paper review questions

## Xen Paper

These review questions are related to the paper, "Xen and the Art of Virtualization" written by Paul Burham, et. al.

**Imagine a Linux OS on top of Xen (XenLinux). A user launches an application, "foo", on top of XenoLinux. Walk through the steps that happen before `foo` acutally starts running.**

1. The process spawning `foo` calls `fork` to create a new process which includes its process control block. This will result in a `hypercall` to Xen, as the guest operating system needs to create a new page table for the newly created process. Xen will create a new page table utilizing the Memory Management Unit (MMU) and will provide the base pointer of the page table back to the guest OS.
2. `exec` will be called to load the `foo` application into memory, requiring another `hypercall` to Xen. Xen will utilize the MMU to update the page table for the process.
3. Finally, when `foo` is dispatched to run by the guest OS, another `hypercall` will be made to Xen to conduct a context switch for `foo`, and `foo`'s page table will be utilized as it executes on the CPU.

**`foo` starts executing and issues a blocking system call:**

- `fd = fopen("bar");`
- **Show all the interactions between XenoLinux and Xen for this call. Clearly indicate when `foo` resumes execution.**
  - This causes a software trap into the guest OS.
  - A context switch is conducted as the working set for the OS is loaded.
  - The guest OS acquires the file descriptor for `foo`.
  - Another context switch is conducted as `foo` is dispatched back onto the CPU to resume execution.
  - Xen provides para-virtualized operating systems the ability to install fast handlers for `system calls`.

**Upon resumption, `foo` executes another blocking system call:**

- `fwrite(fd, bufsize, buffer);`
- **Show all the interactions between XenoLinux and Xen for this call. Clearly indicate when `foo` resumes execution.**
  - This causes a software trap into the guest OS.
  - The guest OS will generate an I/O request and place the request into the asynchronous I/O ring buffer maintained by the Xen hypervisor.
  - The guest OS will advance the Request Producer pointer in the asynchronous I/O ring buffer.
  - The next time the Xen hypervisor is able to conduct round-robin servicing of I/O requests, it will consumer the I/O request made by the guest OS.
  - The Xen hypervisor will advance the Request Consumer pointer.
  - Once the request is complete, the Xen hypervisor will advance the Response Producer pointer, however, this was a write operation so there will only be an indication that the I/O request to the device was successful.
  - This all occurs asynchronously so, as soon as the Request Producer pointer is updated by the guest OS, a context switch will occur and `foo` will resume execution in order to either continue writing or continue execution.

**Construct and analyze a similar example for network transmission and reception by `foo`**

- - Software trap into the guest OS.
  - Guest OS handles the async I/O interaction with the ring buffer, stages a packet for transmission.
  - The packet is inspected by the Virtual Firewall Router (VFR) prior to being transmitted.
  - Transmission occurs.
  - Interrupt occurs as response is received by network interface card.
  - Xen handles response and uses VFR to determine packet destination.
  - Xen places packet into async I/O ring buffer and notifies guest OS that a packet is available.
  - Guest OS handles the async I/O interaction and provides response packet to `foo`.
  - `foo` is dispatched to execute on the CPU and its working set is loaded into memory.

**All processes in XenoLinux occupy the virtual address space 0 through VMMAX, where VMMAX is some system defined limit of virtual memory per process. XenoLinux itself is a**

**protection domain on top of Xen and contains all the processes that run on top of it. Given this, how does XenoLinux provide protection of processes from one another? [Hint: Work out the details of how the virtual address spaces of the process are mapped by XenoLinux via Xen.]**

1. Upon creation of the XenoLinux protection domain, Xen provisions some space within machine memory for the protection domain. XenoLinux is aware of its memory locations, whether they are contiguous or dis-contiguous.
2. With this in mind, when creating space for a new process and creating its page table, XenoLinux does not have write access but must explicitly conduct `hypercalls` in order to have Xen create the required space for the new page table. This page table is also marked as read only by Xen, and all the memory references between virtual addresses to physical addresses are created.
3. Any time the page table for a process needs to be updated, XenoLinux will `hypercall`, requesting Xen to conduct the updates. These updates can be delayed so that they can be conducted all at once.
4. XenoLinux will conduct protection of each process like an operating system normally would. XenoLinux knows the location of each process's page table and its virtual to physical memory mappings. With this, XenoLinux is able to determine valid and invalid memory references.

**Give three important motivations for OS virtualization.**

- Extensibility (greater usage of hardware resources at a lower cost)
- Flexibility (able to run multiple different types of OSs and applications on the same hardware without significant modification)
- Redundancy (continuity of services - if one operating system crashes, others are protected from this and can continue to provide services)

## **VMWare ESX**

**What is the difference between VMWare Workstation and VMWare ESX server? Discuss how OS calls from a user process running inside a VM are handled in two cases.**

- VMWare Workstation is *host-based*. VMWare ESX server is a hypervisor, does not run within a host OS.

- Both provide the illusion to the guest VM that it is running bare-metal. VMWare Workstation traps system calls, conducts the system call itself to the host OS, and then returns the result to the guest OS. VMWare ESX server manages system hardware directly - it validates privileged instructions, conducts the instructions on behalf of guest VMs, and provides the results back to the guests.

**Discuss the difference in memory virtualization in VMWare and Xen.**

- VMWare references the actual physical memory as *machine memory*. Each guest OS thinks its memory is actual *physical memory*, however, transparent to the guest OS its memory is not contiguous. In addition, the VMWare ESX hypervisor maintains a *shadow page table* to translate virtual memory addresses to machine memory addresses.

**What are the approaches to reclaiming memory in a virtualized over-subscribed server?**

**Discuss the pros and cons of each approach.**

- The standard method involves introducing another level of paging. This would involve paging some of the pages of a VM to swap space on a disk. This would require a *meta-level* page replacement policy to be implemented by the hypervisor. The hypervisor not only has to identify a candidate to reclaim memory from, but also what particular pages need to be swapped out from that VM.
- Two mechanisms were generated from this venture:
  - Ballooning - installing a pseudo device driver into each guest VM. The hypervisor can inflate the balloon to reclaim pages from a VM. The VM will use its own internal paging policy in order for it to determine what memory is actually good to be paged. This allows for less interference within the VM by the hypervisor - only works with VMs that will cooperate. The balloon may also be unable to reclaim memory quickly enough to meet system demands. Upper bounds of balloon sizes may also be imposed by guest OSs.
  - Demand paging - memory is reclaimed by paging out to an ESX server swap area on disk, without the guest's consent. This is faster and able to meet the system's demands for memory faster, but can effect guest OS performance significantly. Also requires the implementation of a swap daemon and its algorithm to determine candidate pages for swap.

**Describe how ballooning may be used with Xen.**

- Ballooning would not be transparent to guest OSs like it is with VMWare ESX server. Each domain would be aware of the balloon's existence and its size. Each domain would be able to make a `hypercall` to Xen to request more memory from the balloon. The balloon would then deflate similar to VMWare ESX. Xen would also be able to notify the guest OS that the balloon will begin to inflate as memory requirements increase across the machine. The machine would be able to anticipate the need for enacting its paging mechanism, and will be able to decide which pages need to be swapped out of memory.

### **Explain how page sharing works in VMWare.**

- Page sharing is transparent to all guest OSs. VMWare ESX periodically checks the content of each page being used by each guest OS. VMWare ESX records a hash for each page being used by different VMs and maintains a hash table. If a hash for the content of a page matches to a hash within the hash table, VMWare ESX will conduct a full scan of the memory to ensure the pages match completely. Afterwards, VMWare ESX, transparent to the guest OSs involved, will have each guest OS point to the same page for that particular memory reference. All future changes will be *copy-on-write* to maintain the privacy of each page between each guest OS - the OSs will then separate their memory usage.

### **Explain the allocation policy of VMWare.**

- VMWare uses a *shares-per-pages* ratio. Customers that have paid for more resources will receive those resources and victim clients that have paid less will be required to release resources. There is a minimum amount of resources you can have allocated based upon how much you pay. VMWare ESX also implements a *idle memory tax* to help balance the possible inequality effects that might be experienced if a customer pays a lot for resources, then hoards those resources and doesn't use them. VMWare ESX by default implements a 75% taxation rate on *idle memory*. The basic idea is clients are charged more for active pages than idle pages.

lesson4

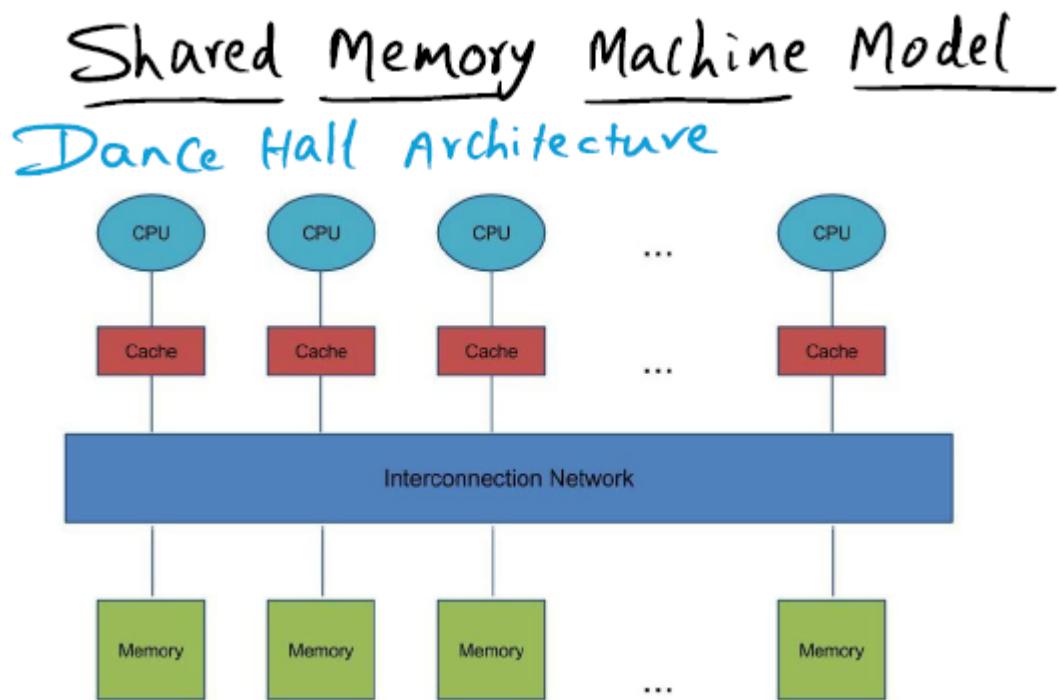
# **lesson4**

# shared memory machines

Different structures for shared memory machines:

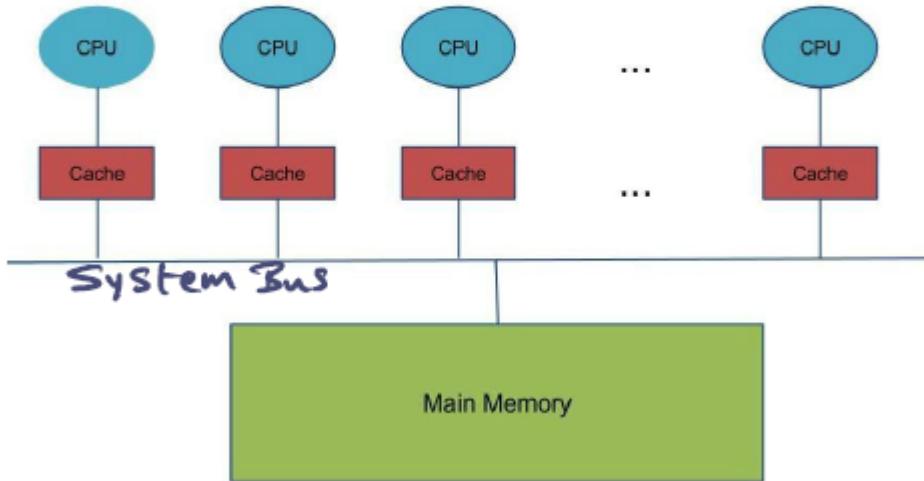
- Dance Hall Architecture
- Symmetric Multiprocessor (SMP)
- Distributed Shared Memory Architecture (DSM)

Below are high-level representations of the above structures.



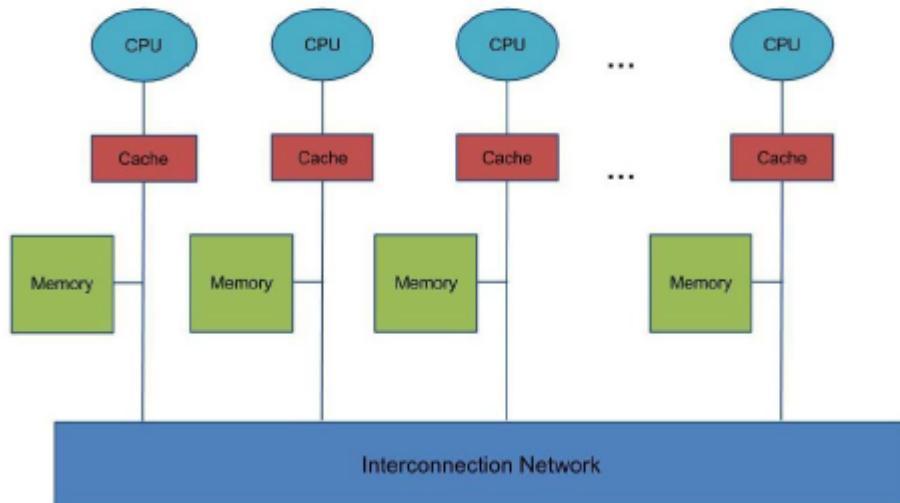
## Shared Memory Machine Model

### SMP (Symmetric MultiProcessor)



## Shared Memory Machine Model

### DSM (Distributed Shared Memory)

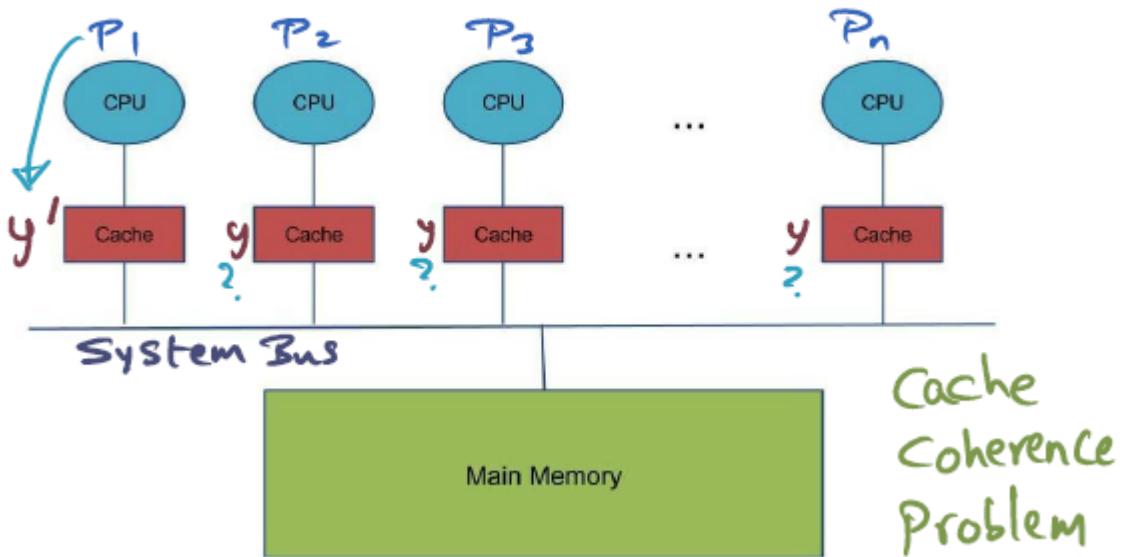


## Shared memory and caches

This section essentially covers the **cache coherence problem** for shared memory systems. This also covers the **memory consistency model** between hardware and software.

Below is a high-level representation of the cache coherence problem.

# Shared Memory and Caches



## Memory Consistency Model

Programmer needs to know what to expect - this is what is generated by a **memory consistency model**.

### Sequential Consistency (SC)

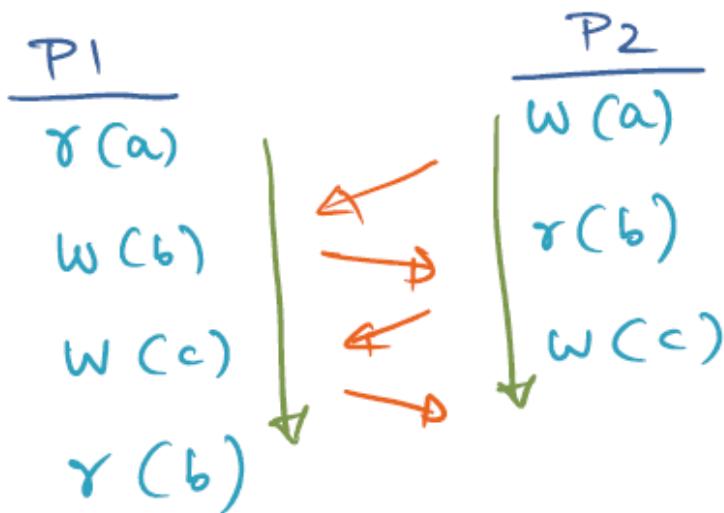
Memory accesses are going to execute sequentially - the program order will be maintained. Arbitrary interleaving mechanism is added to this model.

- Program order + arbitrary interleaving

This model can be thought of as card shuffling.

Below is a high-level representation of sequential consistency.

## Sequential consistency (SC)



Program  
order  
+  
arbitrary  
interleaving

## Memory consistency and cache coherence

Question: assume  $a = b = 0$  initially.

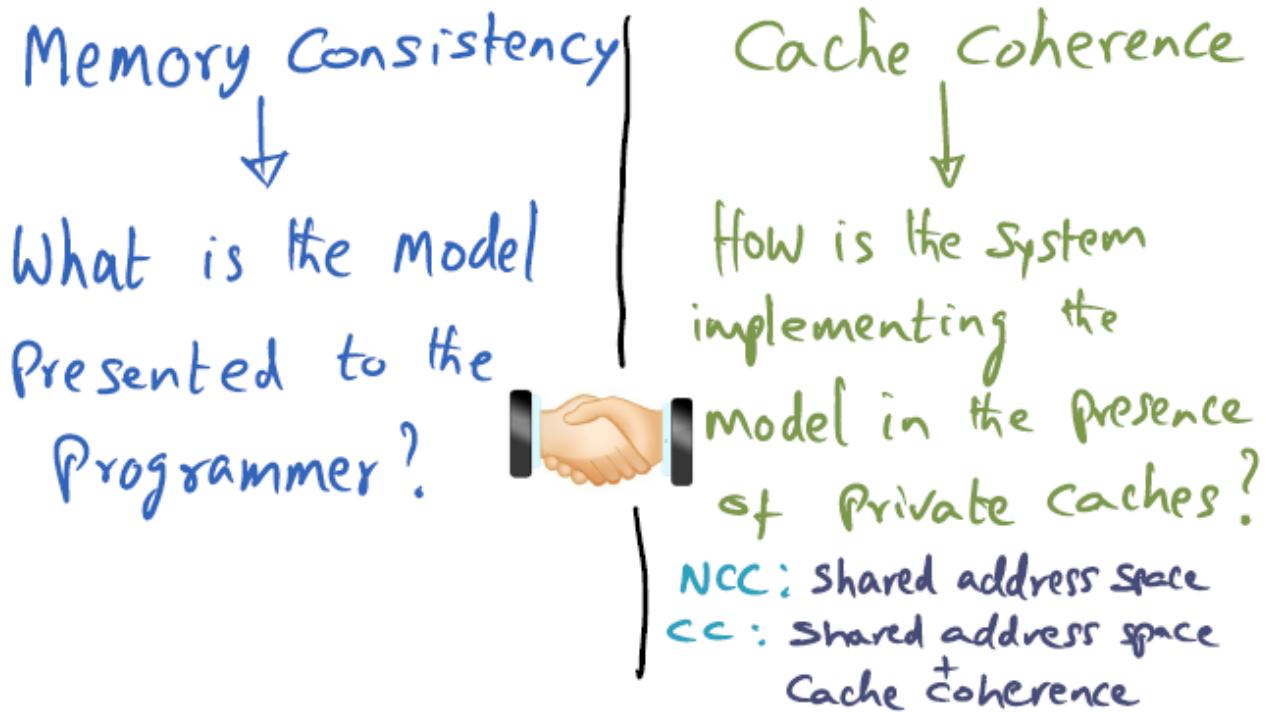
Process P1	Process P2
$a = a + 1$	$d = b$
$b = b + 1$	$c = a$

What possible values for  $d$  and  $c$ ?

- $c = 0, d = 1$   $\leftarrow$  still not possible with **sequential consistency**
- $c = d = 0$
- $c = d = 1$
- $c = 1, d = 0$

- Memory consistency - what is the model presented to the programmer?
- Cache coherence - how is the system implementing the model in the presence of private caches?

These are the two aspects about distributed memory systems that must work together to implement these policies. Below is a high-level representation of the concepts discussed above.



## Hardware cache coherence

Term	Definition
Non-cache coherent multiprocessors	hardware that does not support cache coherence. System software must implement policies and mechanisms.
Cache coherent multiprocessors	hardware that supports cache coherence. System software is not required to handle mechanisms.
write invalidate	if a memory location is present in other caches, and it has been <i>written</i> to, the hardware will invalidate the location in all other caches. This is conducted via a broadcast via the system bus.
write update	if a CPU writes to a particular memory location, the hardware will send an update and all caches will <i>update</i> their memory locations.

Both of these mechanisms cause *overhead*, tying up the system bus or whatever communication network exists to conduct broadcasts.

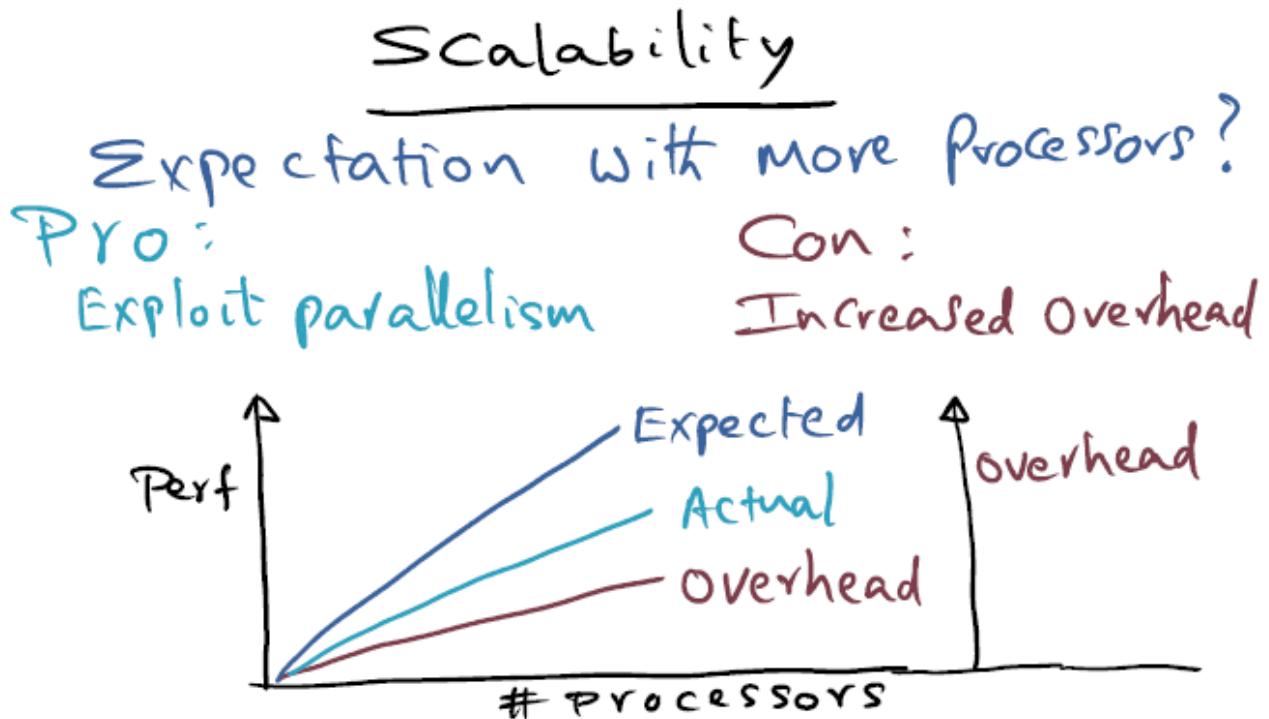
## Scalability

Naturally, we expect the performance of a system increases with the increase of processors. However, you must keep in mind that overhead is incurred when more processors are added - especially if they're sharing resources like memory.

Pro	Con
Ability to exploit parallelism	Increased overhead

Our actual delivered performance will eventually plateau as overhead increases as we leverage more parallelism.

Don't share memory as much as possible across threads. Shared memory works well the less memory we share. User shared data structures need to be kept to a minimum. Below is a high-level representation of the concepts listed above.



## Quizzes

**Question:** assume `a = b = 0` initially.

Process P1

`a = a + 1`

`b = b + 1`

Process P2

`d = b`

`c = a`

What possible values for `d` and `c`?

- `c = 0, d = 1`
- `c = d = 0`
- `c = d = 1`
- `c = 1, d = 0`

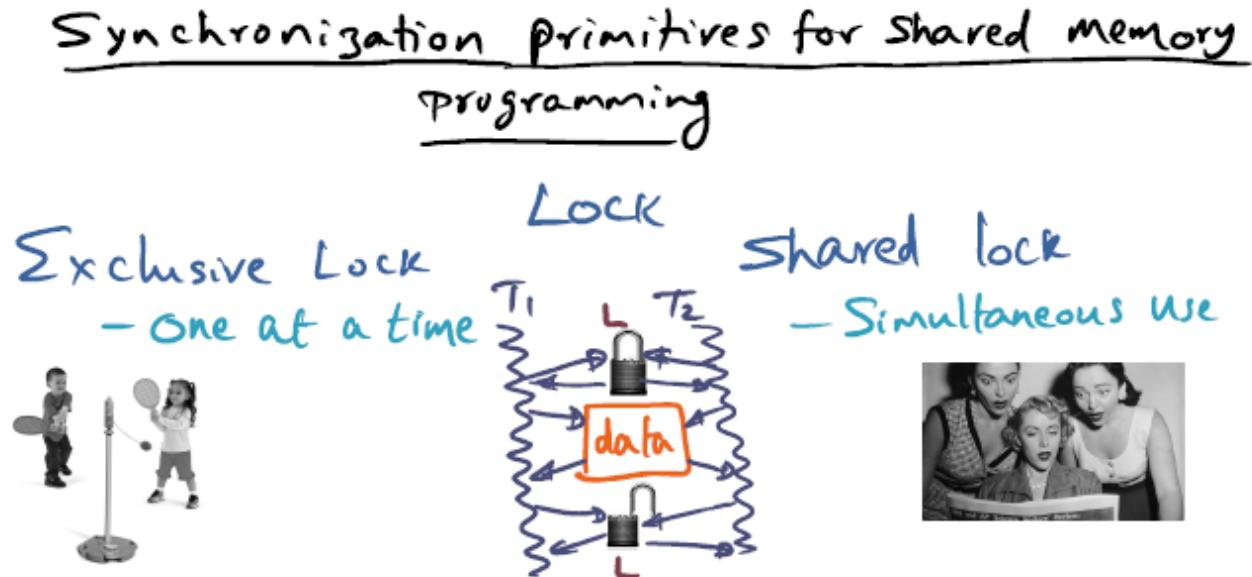
# synchronization

## Synchronization primitives for shared memory programming

What is a lock? A `lock` is something that allows a thread to make sure that, when accessing a particular piece of shared data, it is not being modified by another thread. Locks come in different flavors.

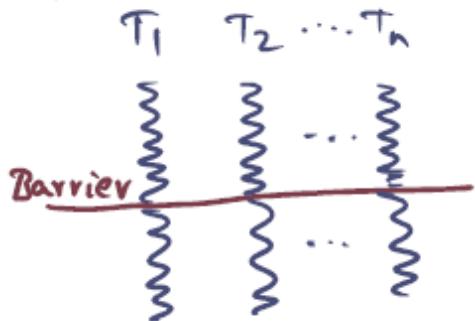
Lock type	Description
Mutually exclusive lock	<code>mutex</code>
Shared lock	simultaneous use of a memory location.
Barrier	a particular point in computation across multiple threads that provide a notification that a location in execution has been met.

Below is a high-level representation of the concepts described above.



# Synchronization primitives for shared memory programming

## Barrier



## Atomic operations

### Atomic Read-Modify-Write (RMW) Instructions

Instruction	Description
Test-and-set()	return the current value in the <code>mem_location</code> and setting the <code>mem_location</code> to <code>1</code> .
Fetch-and-inc()	return the current value in <code>mem_location</code> and increment <code>mem_location</code>

## Scalability issues with synchronization

Issues that exist:

- Latency - how long does it take for a thread to acquire a lock?
- Waiting time - how long does a thread have to wait to acquire a lock?
- Contention - how long does it take in the presence of contention to win an unlocked lock?

## Naive spinlock (spin on T + S)

Essentially spinning on the `test` and `set` instruction. Super naive, high contention, latency, and wasteful.

```
while (test_and_set(lock) == locked);
```

Below is a high-level representation of the naive spinlock.

## Naive Spinlock (Spin on T+S)

`LOCK(L):`

```
    while (T+S(L) == locked);
```

`unlock(L):`

```
    L = unlocked;
```



## Caching spinlock (spin on read)

This version will read the value of the lock from the cache. The cache consistency mechanism implemented by the hardware will update the value of the lock in the cache of each processor. This allows us to avoid contention across the communication bus.

```
while (L == locked);
```

Unfortunately, contention is introduced because multiple processors will attempt to acquire the lock simultaneously, because the caches across each processor will update

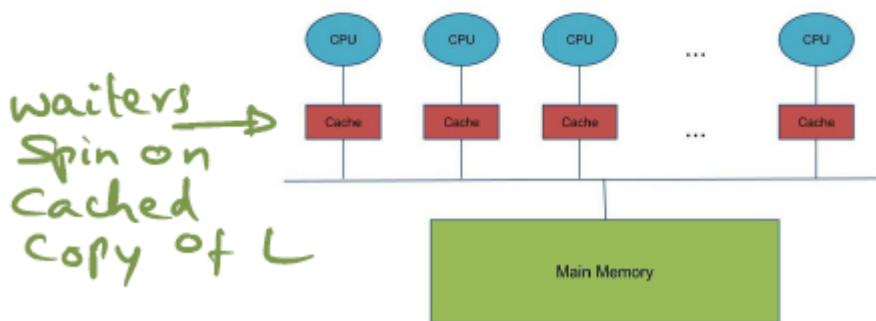
simultaneously.

Below is a high-level representation of the caching spinlock.

## Caching spin lock (Spin on read)

$\text{LOCK}(L)$ :

While ( $L == \text{locked}$ );  
if ( $T + S(L) == \text{locked}$ ) go back;



## Spinlocks with delay

Delay type	Description
Static delay	this algorithm uses a variable amount a delay before attempting to contend for a lock. This reduces contention across the communication bus. This uses a static delay based on the processor's ID. This makes it harder for higher delay processors to acquire a lock.
Delay with exponential backoff	uses an exponential backoff on every failure to contend and acquire the lock. Decreases contention better than static for a high number of processors, but also works better for a lower number of processors. Delay is increased if there's a lot of contention.

Below is a high-level representation of spinlocks with delay.

## Spinlocks with Delay

### Delay after lock release

```

while((L==locked) or
      (T+s(L)==locked))
{
    while (L==locked);
    delay (d[Pi]);
}

```

### Delay with exp. backoff

```

while
  (T+s(L)==locked)
{
    delay (d);
    d = d*2;
}

```

## Ticket lock

```

1 int my_ticket = fetch_and_inc(L->next_ticket);
2
3 while (1)
4     sleep (my_ticket - L->now_serving);
5     if (L->now_serving == my_ticket)
6     {
7         return;
8     }

```

This algorithm is good because it preserves fairness, but the once the lock is released, contention is created on the network as everyone acquires the updated value.

Below is a high-level representation of the ticket lock.

## Ticket Lock

```

struct lock{
    int next-ticket;
    { int now-serving;
};

acquire-lock(L):
    int my-ticket = fetch-and-inc(L->next-ticket);

loop
    pause (my-ticket - L->now-serving);
    if (L->now-serving == my-ticket) return;
    ↑
    got it!

```

now-serving      16      25

my-ticket

release-lock(L):  
L->now-serving ++;

## Spinlock summary

`read_and_test_and_set`, `test_and_set` with delay, and `ticket_lock` spinlock algorithms all have their issues. No fairness exists with the first two, and the final one is fair but still noisy on the communication bus.

Only **one** thread will be able to acquire the lock at the end of the day, so why are they all contending? The thread that will receive the lock next should already be aware it is next in line, and the rest of the threads will wait their turn.

This is implemented in the **array based queuing lock**.

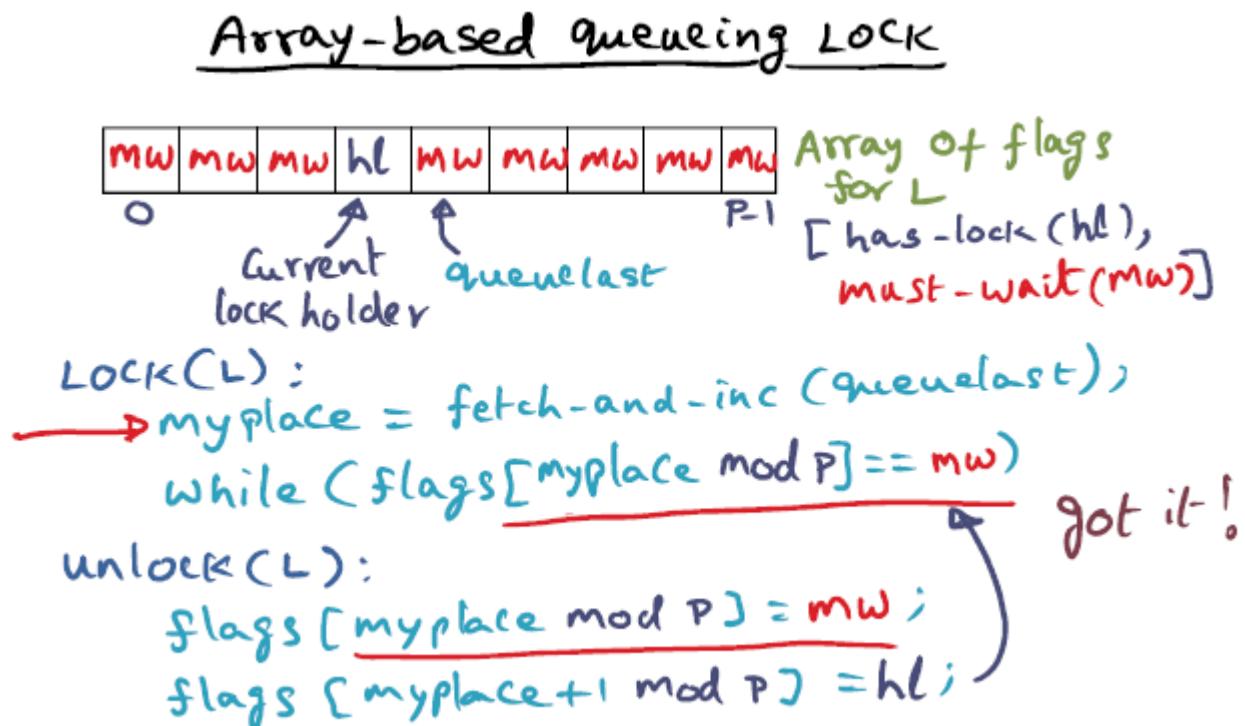
## Array based queuing lock

The array-based queue lock is initialized to the number of processors currently in the computing system. The `queuelast` variable is initialized at array location `0`. To join the queue, a processor fetches the value of `queuelast` and saves it locally. Then the processor increments the `queuelast` variable. This is an atomic operation, no race conditions will occur when attempting to join the queue. If the architecture does not support `fetch_and_inc`, we can use `test_and_inc` instructions.

The processor will continue to check its location in the array until it `has_lock`. Releasing the lock involves the processor setting its own location to `must_wait` and then sets the next location in the array to `has_lock`.

The benefits of this lock algorithm is that it is fair and not noisy (causing contention on the communication bus). The downside is the size of the data structure to host processor status (space complexity  $O(n)$ ).

Below is a high-level representation of the array-based queuing lock.



## Link based queuing lock

The link based queuing lock is initialized with a dummy node (the head), and is dynamically sized with the number of processors requesting the lock. Every new requestor of the lock creates a structure called a `q_node`.

```

1  typedef struct _q_node q_node;
2  struct _q_node
3  {
4      bool got-it;           //TRUE / FALSE

```

```

5     struct q_node next;      //Successor
6 };

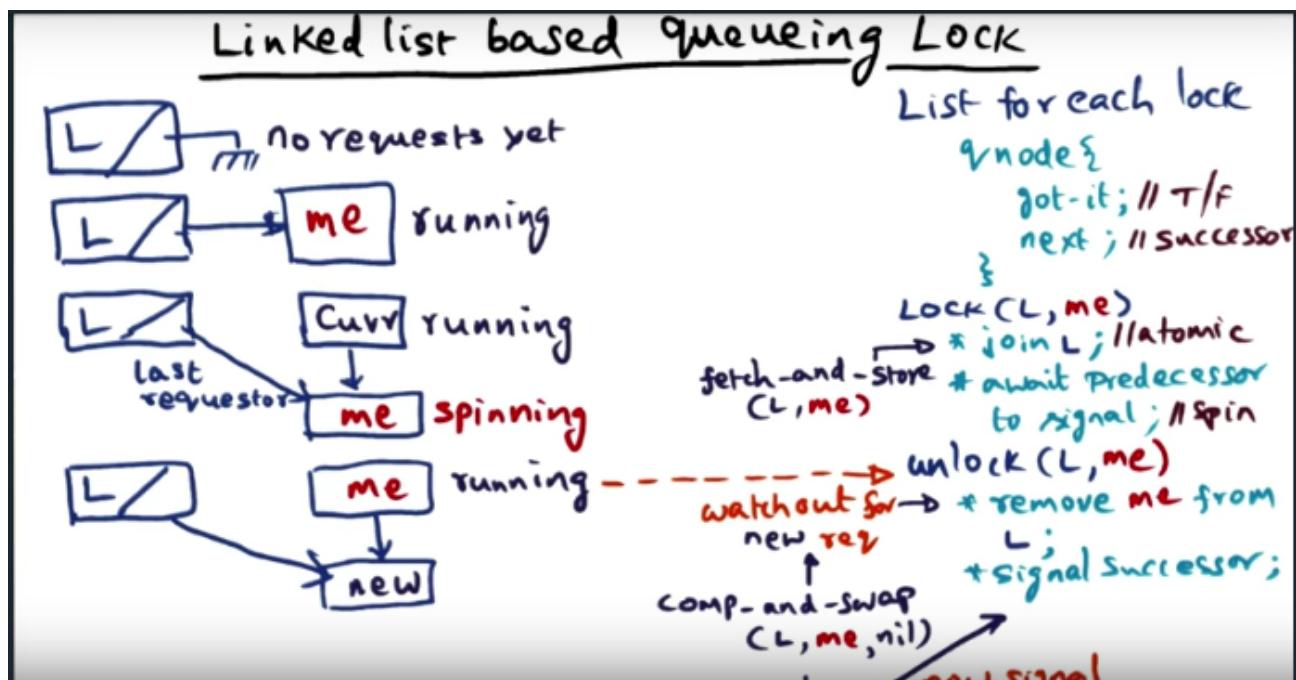
```

An atomic instruction is used to access the lock, and logic is implemented when a processor adds itself to a lock. What occurs is the processor determines what other processor is using the lock, and uses that specific processor's `next` field to point to itself. Then the processor will wait on the lock variable `got-it` until it is `true`.

The `unlock` function for a processor involves the processor removing itself from the list. If there is no `next`, the processor will set the successor to `NULL`. If a new request is forming, however, a race condition occurs. In this case, the processor will conduct a `comp_and_swap` to see if another processor is actively attempting to acquire the lock.

Lots of subtleties exist in the implementation of this synchronization algorithm - it requires more complex instructions ( `fetch_and_store` and `comp_and_swap` ). The best part about this algorithm is that its space complexity is dynamic and set to the size of the number of requestors. A downside is there exists linked-list maintenance that incurs overhead.

Below is a high-level representation of the concepts discussed above.



## Quizzes

Assuming only read / write *atomic instructions*, is it possible to achieve the programmer's intent?

P1

Modify `struct_t a`

P2

Wait for mod

Use `struct_t a`

- Yes
- No

What are the problems with naive spinlock?

- too much contention
- does not exploit caches
- disrupts useful work

Grade the algorithms by filling this table.

Question  
Grade the algorithms by filling in this table  
Solution

Algorithm	Latency (low/med/high)	Contention (low/med/high)	Fair (Y/N)	Spin (pvt/sh)	RMW ops per CS (low/med/high)	Space ovhd (low/med/high)	Signal only one on lock release (Y/N)
Spin on T&S	low	high	N	S	high*	low	N
Spin on read	low	med	N	S	med*	low	N
Spin w/delay ✓	low++	low+	N	S	low+	low	N
Ticket lock	low	low++	Y	S	low++	low+	N
Anderson ✓	low+	low	Y	P	1	high	Y
MCS ✓	low+	low	Y	P	1(max 2)	med	Y

\* proportional to contention for lock

# communication

## Barrier synchronization

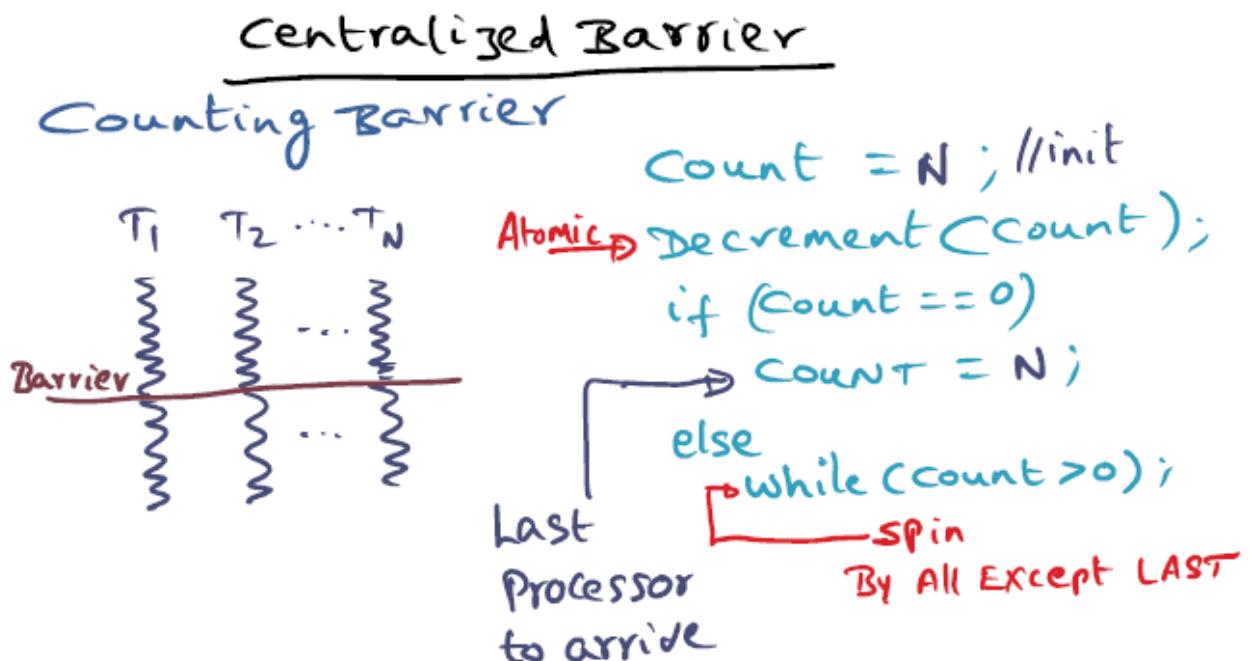
Processors executing different threads will reach a **barrier** and will attempt to wait for all other processors to reach the **barrier**. This portion of the lesson will discuss algorithms used for **barrier synchronization**, their benefits and pitfalls.

### Centralized barrier

Also known as a **counting** barrier, the centralized barrier uses a **count** that is equal to the number **N** of processors. As processors reach the barrier, the **count** will be atomically decremented. Processors will **busy-wait** on the count until it reaches **0**.

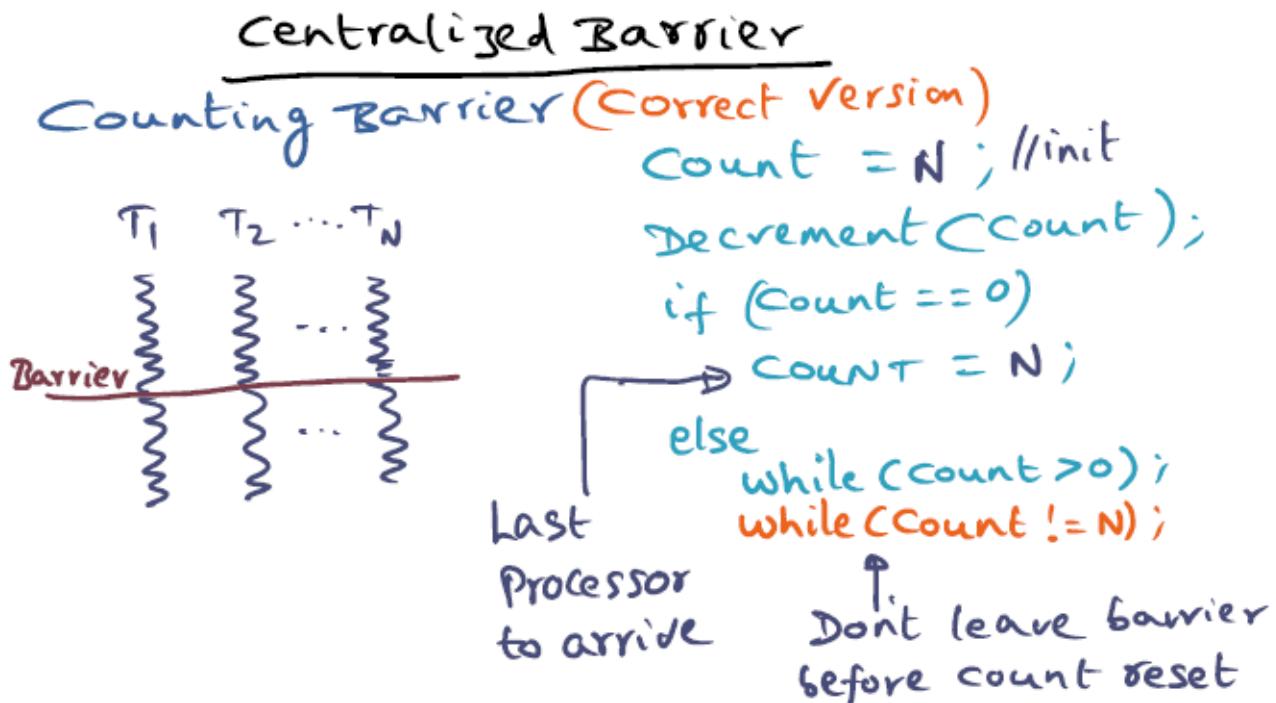
The last processor will arrive, see that the count is **0**, and will reset the count to **N**. All the other processors will see that the count was set to **0** by the last processor and will be released until they reach the next barrier.

Below is a high-level representation of the centralized barrier.



## Counting barrier

The above algorithm contains an issue, however, in that processors being released can race to the next barrier and fall through, detecting that the count is still 0. Below is a fixed high-level representation of the algorithm in which the processors spin until the count is N again, then they are released.



## Sense reversing barrier

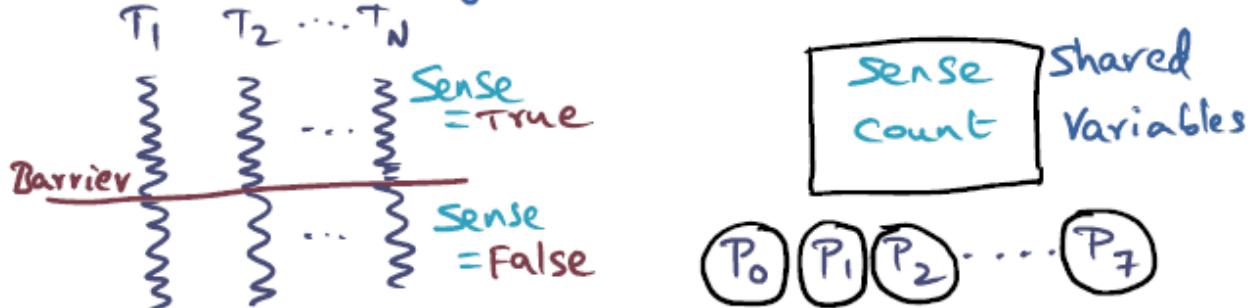
In the sense reversing barrier, we remove the requirement to sense the count becoming 0. A sense variable is initialized as a boolean true, and all processors spin on this sense variable until the sense variable becomes false.

As processors reach the barrier, they decrement the count and spin on the sense variable. The last processor reaches the count, resets the count to N, and reverses the sense variable. This removes one spin operation in comparison to the original centralized barrier, however, because of the single shared sense variable, there is a large amount of contention across the communication bus.

Below is a high-level representation of the sense reversal barrier algorithm.

## Centralized Barrier

### Sense reversing barrier



All except last:

- Decrement count
- Spin on Sense reversal

Last:

- Reset count to N; reverse sense

## Tree barrier

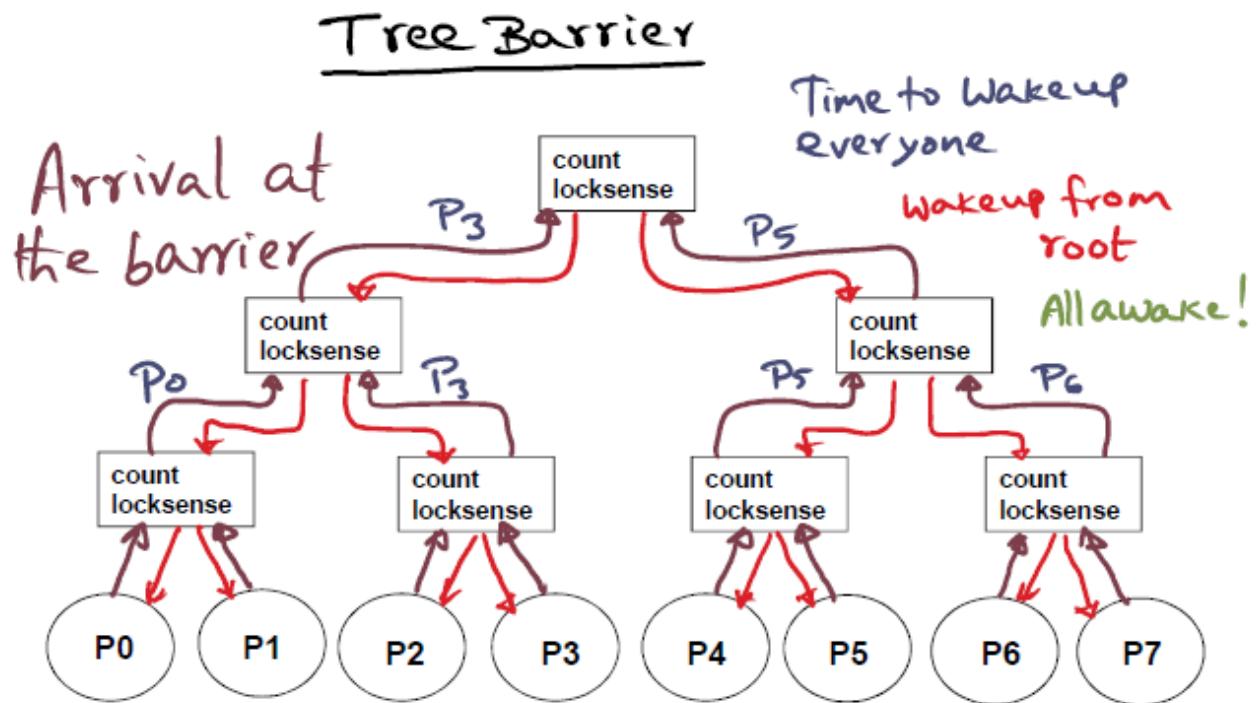
K processors share a memory location that contains a **count** variable and a **locksense** variable. The last processor to arrive at the previous memory location shares another memory location with L processors at the next level of the tree. The same series of actions takes place at this memory location, waiting for a partner to arrive at the barrier.

This continues to happen until the two final processors meet at the tree's root, signifying that all processors have arrived at the barrier. This algorithm essentially uses the sense reversal barrier algorithm, but it attempts to reduce contention by breaking up the shared memory locations into multiple barriers.

Problems with this algorithm include:

- The lock sense flag that a particular processor spins on is not statically defined. It's dynamic based upon the pattern that arises when processors reach a barrier.
- Contention is based upon the **ary-ness** of the tree. The number of processors allowed to spin on a lock sense flag.
- Contention is also dependent upon whether or not the system is cache-coherent. If cache-coherence is not forced, processors could be spinning on remote memory, causing contention.

Below is a high-level representation of the tree barrier algorithm.



### 4-Ary arrival tree (MCS tree barrier)

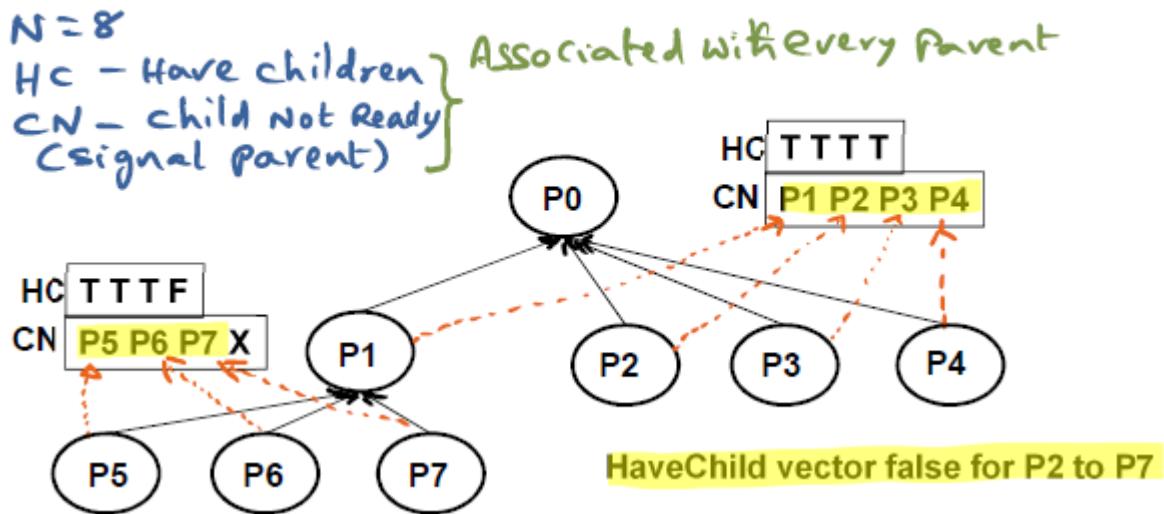
In this tree, parent processors are identified. Each processor has two data structures:

`have_children` and `child_not_ready`. The parents account for the ready status of their children, at most four (4), and will wait for the children to be ready before crossing the barrier. Each child will utilize the parent's static data structure within shared memory and notify the parent it has arrived at the barrier.

A benefit of this algorithm is that each processor is assigned a unique spot in each parent's data structure, reducing contention. In a cache-coherent system, all the arrival data can be stored in one word, locally in the parent's cache further reducing contention.

Below is a high-level representation of the MCS tree barrier.

## MCS Tree Barrier (4-ary arrival)



## Binary wakeup tree (MCS tree barrier)

In this tree, every processor is assigned a unique, static spot again. Each processor contains a data structure called the `child_pointer` - parents utilize this data structure to wake up its children.

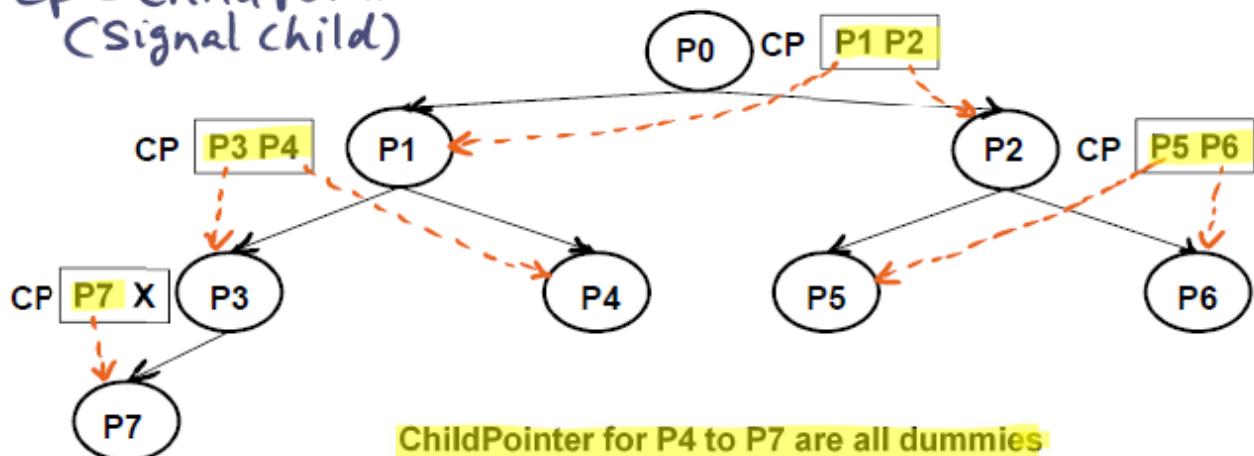
When the parent at the root of the tree realizes that all processors have arrived at the barrier, the root parent will signal the two children within its `child_pointer` data structure. This happens recursively until all children of each parent are woken up.

Again, the memory locations are set in specific locations, no dynamic memory locations are set. Through this static assignment, we reduce contention as much as possible.

Below is a high-level representation of the binary wakeup tree.

## MCS Tree Barrier (binary wakeup)

cp - child pointer  
(Signal child)



All Awake!

## Tournament barrier

This is essentially a barrier where **matches** are implemented between two processors.

Based upon **N** players, there will be **log2N rounds**. The matches are fixed, and the actual lineup is fixed as well. p0 will always battle p1 and win, p2 will always battle p3 and win, and so on. This allows the algorithm to statically define the location where the loser will notify the winner that it has won, helping to reduce contention on memory locations that are being spun on.

After the final match, the final winner will tell the final loser that it's time to wakeup, everyone has reached the barrier. This winner to loser notification will happen recursively until all processors are woken up.

The loser will also spin on a local, statically defined variable. This is convenient even for non-cache-coherent systems, but overall it reduces contention because each process is spinning on local variables.

The virtues of this algorithm:

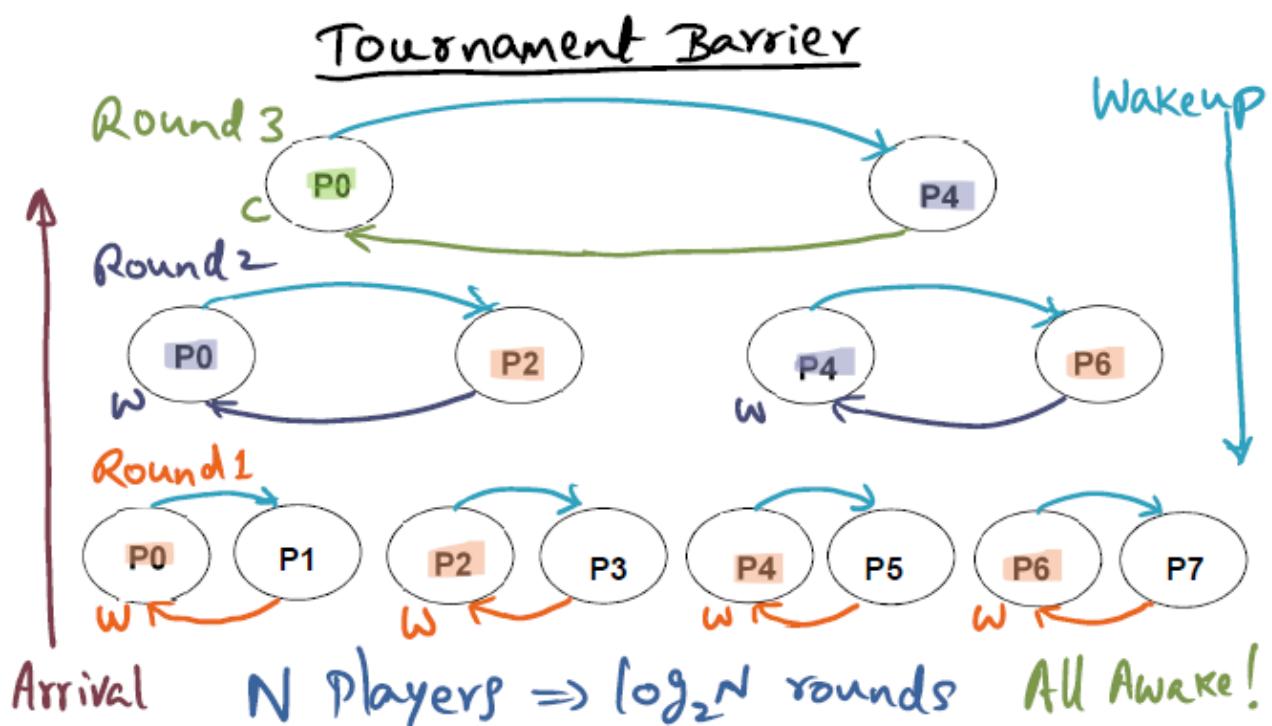
- Spin locations are statically determined, in comparison to the tree barrier.
- No need for a `fetch_and_atomic` operation, in comparison to the tree barrier. Allows for simpler implementation.

- The amount of communication, in comparison to the tree barrier, is exactly the same:  $\log_2 N$ .
- Communication between winners and losers can be done in parallel.
- Even if the system is not a shared memory machine, the tournament barrier still works.

The pitfalls of this algorithm:

- Cannot leverage spatial locality like the MCS barrier can.

Below is a high-level representation of the tournament barrier.



## Dissemination barrier

This barrier works by the diffusion of information among participating processors. What's nice about this barrier is that the number of processors ( $N$ ) does not need to be a power of 2. The idea is we have multiple rounds of information dissemination, and at the end each processor is going to gossip to or receive gossip from every other processor.

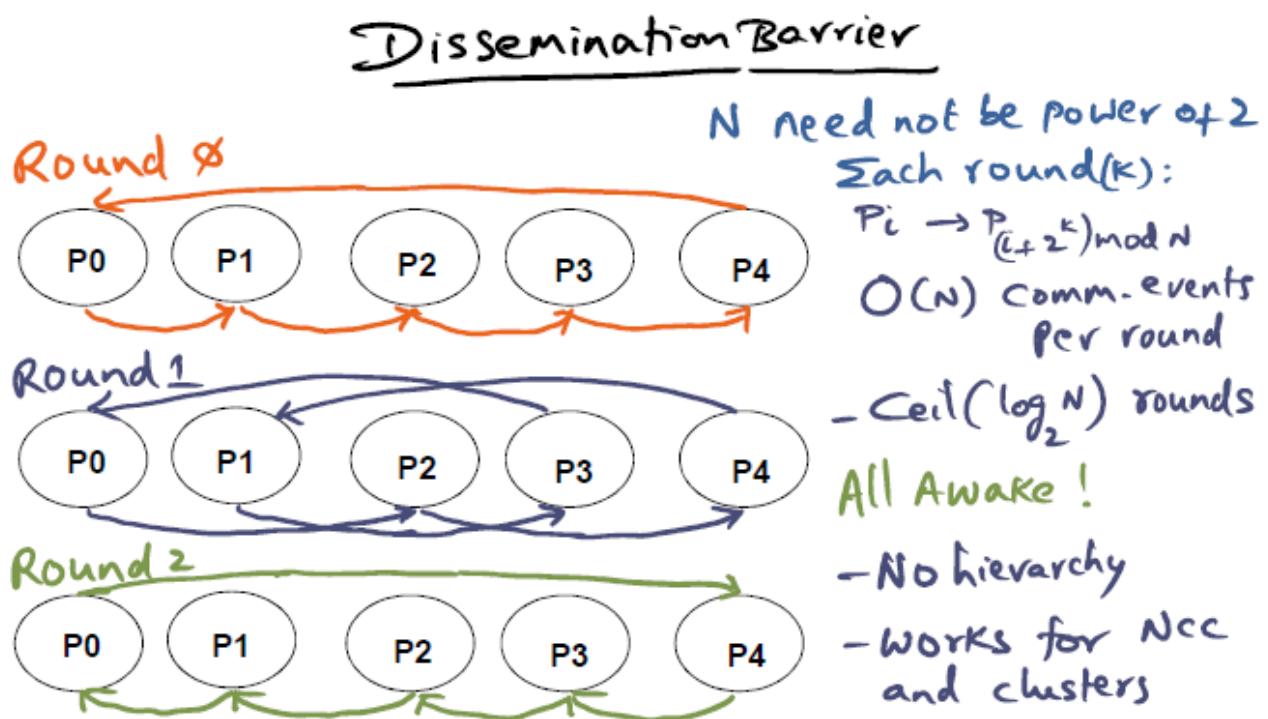
Each processor will know when a round is complete when it both sends a message and receives a message. An order of  $O(N)$  communication events are occurring every round.

In this algorithm, there is no hierarchy as communication is conducted via dissemination. On shared memory machines, messages can be disseminated to each processor's statically determined spin location. As always, sense reversal needs to be conducted at the end of this barrier.

The virtues of the dissemination barrier are as follows:

- No hierarchy
- Works for non-cache-coherent NUMA machines as well as clusters.
- Every processor independently makes a decision to send a message for the round.
- Every process independently makes a decision to move onto the next round.
- The communication complexity is  $O(n \log 2n)$

Below is a high-level representation of the dissemination barrier algorithm.



## Performance evaluation

The only spin algorithms worth using for synchronization are `ticket_lock`, `array_queue_lock`, and `list_queue_lock`. The only barrier synchronization algorithms worth using are the `tournament_barrier`, `mcs_tree`, and `dissemination`.

The algorithm that's best for a computing system depends on the flavor of architecture:

- Cache-coherent symmetric multiprocessor
- Cache-coherent NUMA
- Non-cache-coherent NUMA
- Multiprocessor clusters

## Definitions

Term	Definition
cluster	multiprocessors in which the processors don't share memory; only form of communication is through message passing
NUMA machine	non-uniform memory access; a <a href="#">computer memory</a> design used in <a href="#">multiprocessing</a> , where the memory access time depends on the memory location relative to the processor.

## Quizzes

### Do you see any problem with the centralized barrier algorithm?

Yes. Before the last processor sets the count to N, other processors may race to the next barrier and fall through.

### How many rounds for the dissemination barrier completion?

- $n \log 2n$
- $\log 2n$
- $\text{ceil}(\log 2n)$
- $n$

The ceiling of  $\log 2n$  is the number of rounds. We use ceiling because the number of processor does not need to be a power of 2.

# lightweight rpc

## RPC and client-server systems

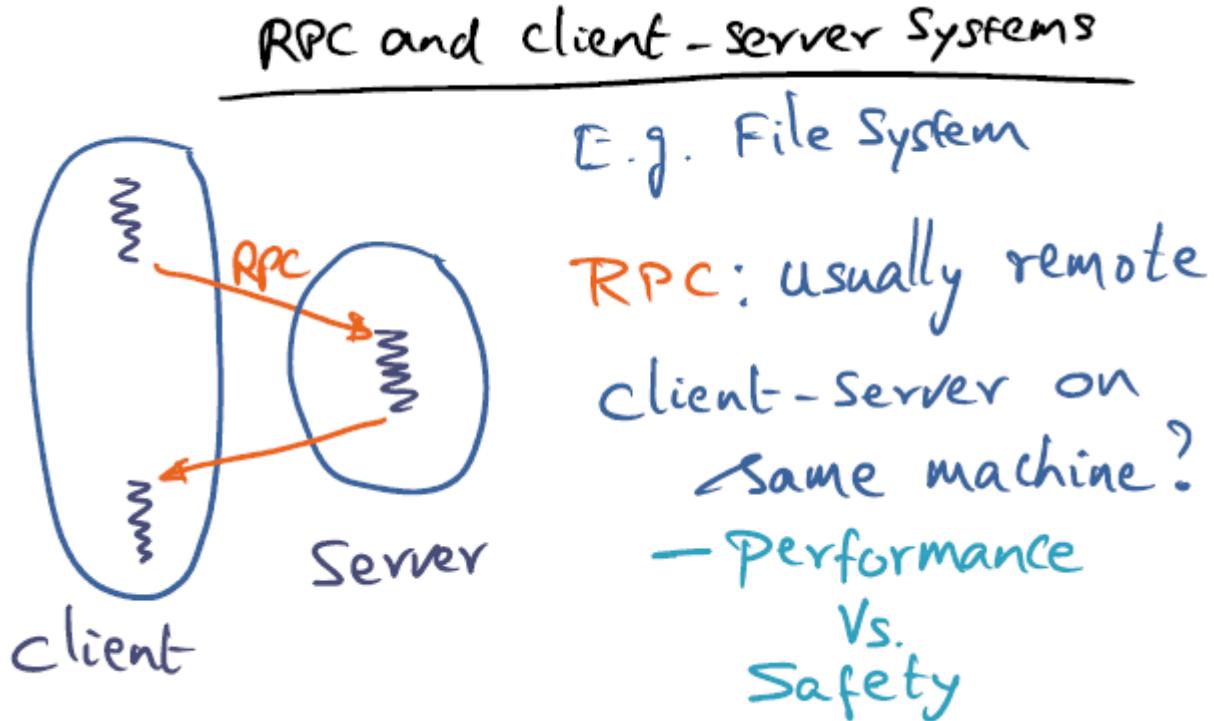
This portion of the lesson discusses efficient communication across address spaces within a distributed system.

Remote procedure call (RPC) is the mechanism used to build this client-server mechanism in a distributed system. Should we use RPC on the same machine? The main concern here is the relationship between performance and safety.

We want to ensure servers and clients reside within different protection domains. This also means we incur overhead because RPC will have to cross address spaces. We would like to be able to make RPC across protection domains as efficient as normal procedure calls.

Why is that a good idea? This allows us to create a protected procedure call, and will encourage system designers to separate services into different protection domains.

Below is a high-level representation of the topics discussed above.



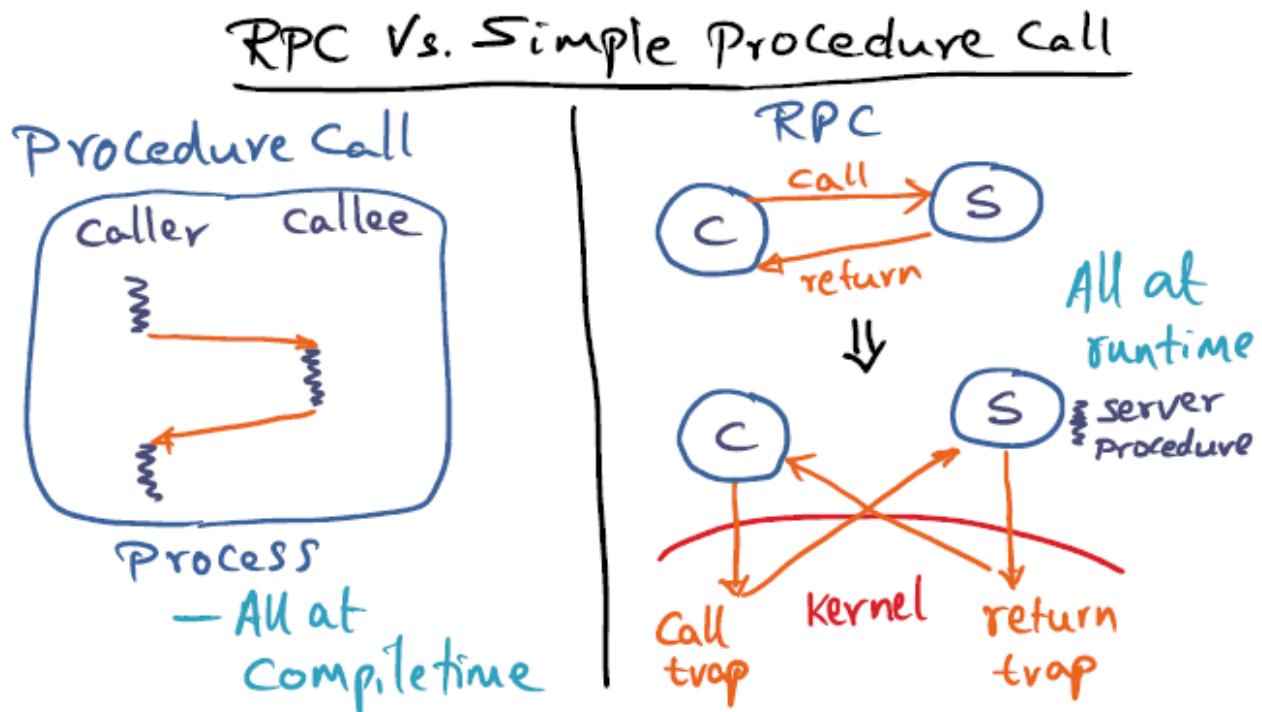
## RPC vs simple procedure call

Procedure calls are resolved at compile time, essentially calling a function within a process.

For remote procedure call, it's the same, however, the call traps into the kernel. The kernel validates the call, copies the arguments into kernel buffers, and copies the arguments into the address space of the callee. The callee is then scheduled to execute the procedure. When the callee has completed the execution of the procedure, the results are provided back to the kernel, copied into kernel buffers, and copied into the caller's address space. All of these actions are resolved at runtime.

These context switches, traps, and copy instructions incur overhead thus hurting performance. Throughout this exchange, the kernel is validating the procedure call / exchange between the client and the server.

Below is the high-level representation of the concepts discussed above.



## Copying overhead

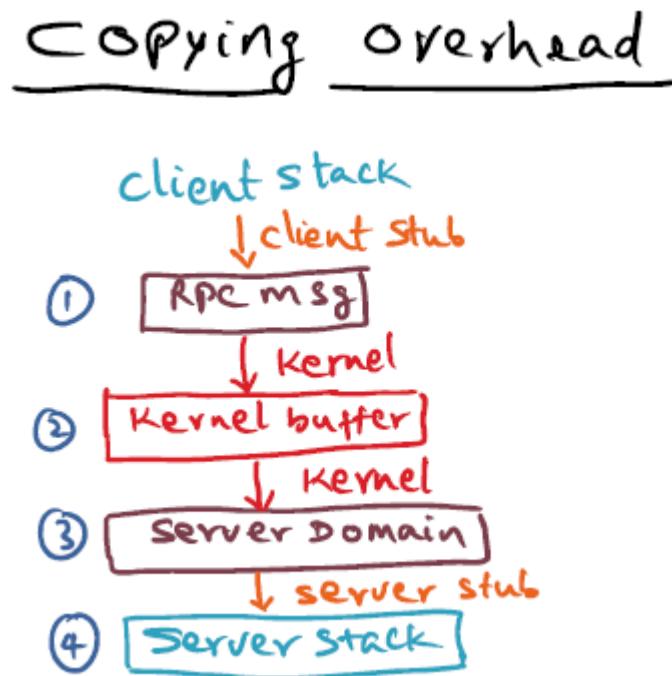
For a regular procedure call, all data movement occurs within user address space within the process. Copying between the kernel and user space occurs on every RPC. We would like to

accomplish not having the kernel involved in moving data between the client and server for a RPC.

Let's break down an RPC. When a client prepares to conduct a RPC, it marshals the data into an RPC message using a client stub. This is the only way the client can communicate this information to the kernel. Next, the client traps into the kernel and the kernel copies the marshaled data from user address space into a kernel buffer. The kernel then scheduled the server to execute on the CPU and copies the marshaled data from the kernel buffer into server address space. The server then unmarshals the data from the buffer and extracts the arguments using the server stub.

All of this is conducted in reverse for the server returning the data back to the client.

Below is a high-level representation of the mechanisms discussed above.



## Making RPC cheap

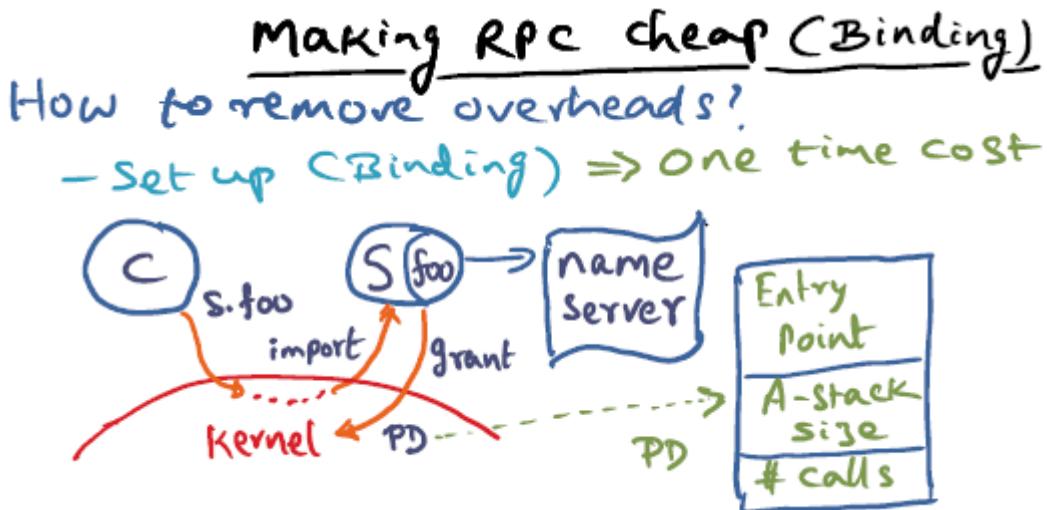
How do we remove the overheads presented in RPC? We optimize the common case, as is tradition. The common case are the actual calls being made by the client and the server. We want to leverage the locality of the information used for each RPC that resides within the cache.

First, we begin by binding the client to the server, which occurs once. How does this work? The server advertises an entrypoint procedure, publishing it on some name server managed by the kernel. Anyone on the system can use the name server to discover the entrypoint of the server.

The server waits for bind requests to arrive from the kernel. Clients interested in the procedure being advertised issue the RPC, the call traps into the kernel on the first execution, and the kernel checks with the server to see if the client is a legitimate caller. The server can determine whether to grant or deny permission.

Once the validation has been completed, the kernel creates a procedure descriptor. The procedure descriptor is a data structure that resides within the kernel, representing the procedure entrypoint. The server describes to the kernel the characteristics of the procedure entrypoint, including the address of the procedure to be called in server address space, the expected argument stack, and how many simultaneous calls will be accepted for a procedure.

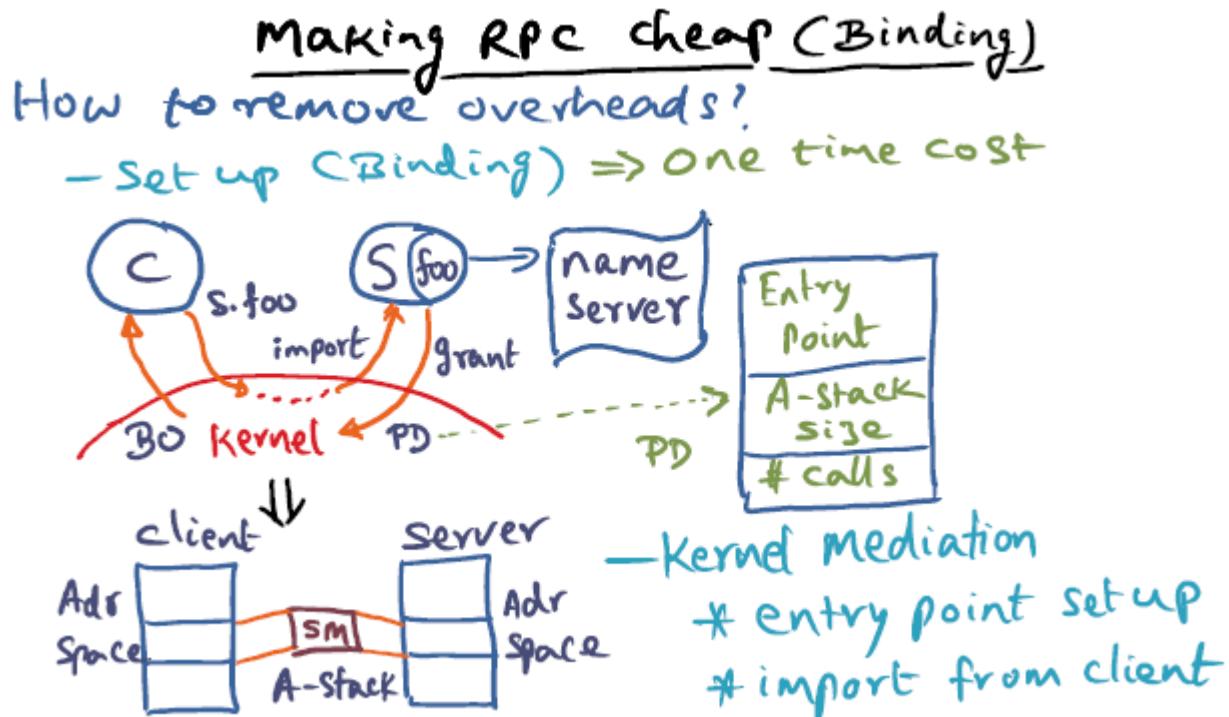
Below is a high-level representation of the concepts discussed above that are implemented in pursuit of making RPC cheaper.



Once all this information is stored in the procedure descriptor, the kernel establishes a buffer called the A-stack (argument stack). This argument stack is actually shared memory mapped between the client and server address spaces. Now the client and server can read / write from the A-stack and the kernel is no longer involved in the data transfer for an RPC.

The kernel authenticates the client, allowing the client to make future calls to the remote procedure. The kernel provides the client a binding object, similar to a capability, that the client must provide the kernel every time the client wants to conduct a RPC to the server.

Below is a high-level representation of the kernel mediation that takes place.



## Making RPC cheap (actual calls)

All kernel copying overheads are now eliminated after the first call is conducted. The client still utilizes the client stub to marshal data for the argument stack. All arguments must be passed by value, not reference, because the data needs to traverse between the two address spaces.

Client traps into the kernel, the client stub presents the binding object to the kernel, the kernel exposes the procedure descriptor thus exposing the entrypoint procedure address for the RPC.

As per the semantics of an RPC call, the client is blocked and the server is scheduled to run at its entrypoint for the RPC. The kernel borrows and doctors the client thread to execute within the address space of the server. The program counter of the client is set to the

address of the entrypoint procedure within the server address space. The server is provided an execution stack by the kernel to conduct its work in.

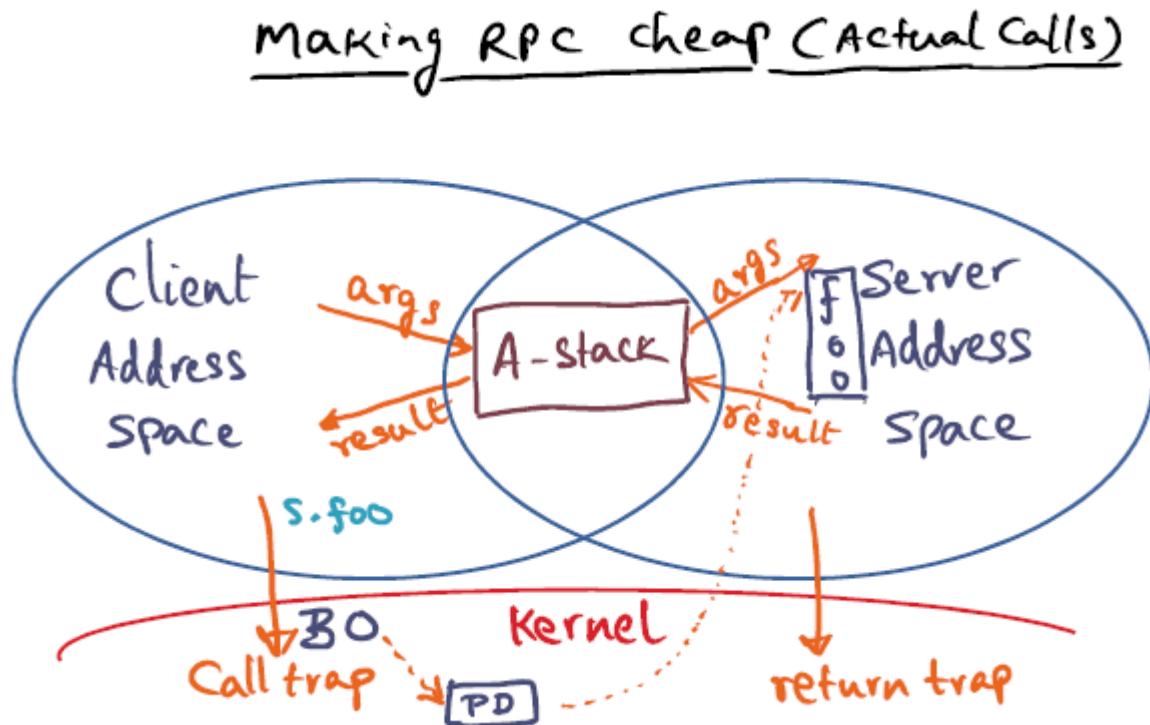
Once the client is doctored, control is transferred to the server. The server stub will copy the arguments from the A-stack into the execution stack. Once the procedure completes successful execution, the results will be copied into the A-stack. The server will then trap back into the kernel. The kernel has no need to validate the return trap.

Control is provided back to the client, the client stub copies the results from the A-stack, and the client continues execution with the results provided.

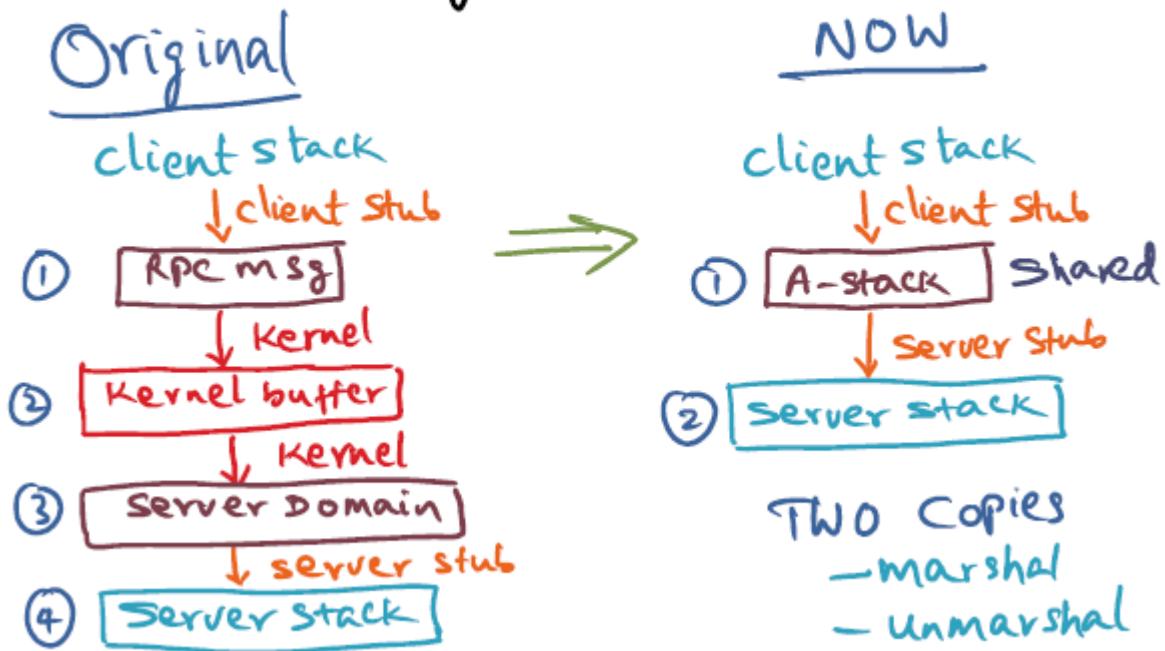
Summary:

- Copying through the kernel is completely eliminated.
- Traps still exist into the kernel in order to conduct validation and the return trap.
- Switching the domains incur overhead because of the context switch.

Below are high-level representations of the mechanisms discussed above.



## Making RPC cheap (Actual Calls)



## RPC on SMP

When implementing this on a shared memory multiprocessor, we can preload the server domains on a particular processor, only allowing the server to run on the processor. The caches will remain warm for a processor bound to a server domain, leveraging locality. The kernel can also inspect the popularity of a particular service and determine if it wants to dedicate other CPUs to be bound to the particular service and host duplicate server domains.

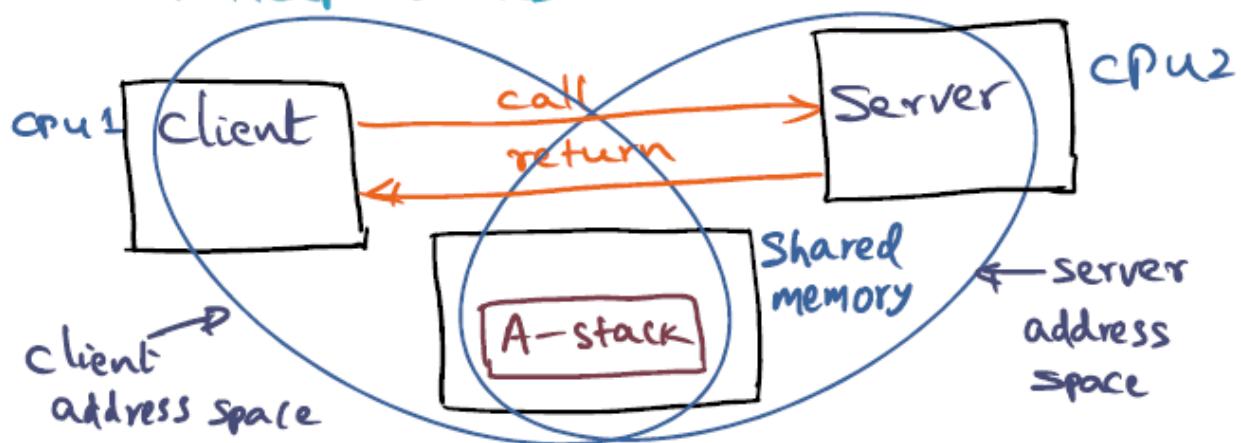
Below is a high-level representation of the concept described above.

## RPC on SMP

Exploit multiple processors

\* Pre-load server domain

\* Keep caches warm



## Quizzes

In an RPC, how many times does the kernel copy data from user address spaces into the kernel and vice versa?

- once
- twice
- thrice
- four times

Copy instructions from user address space into kernel space and vice versa is conducted four times. That's a lot of overhead.

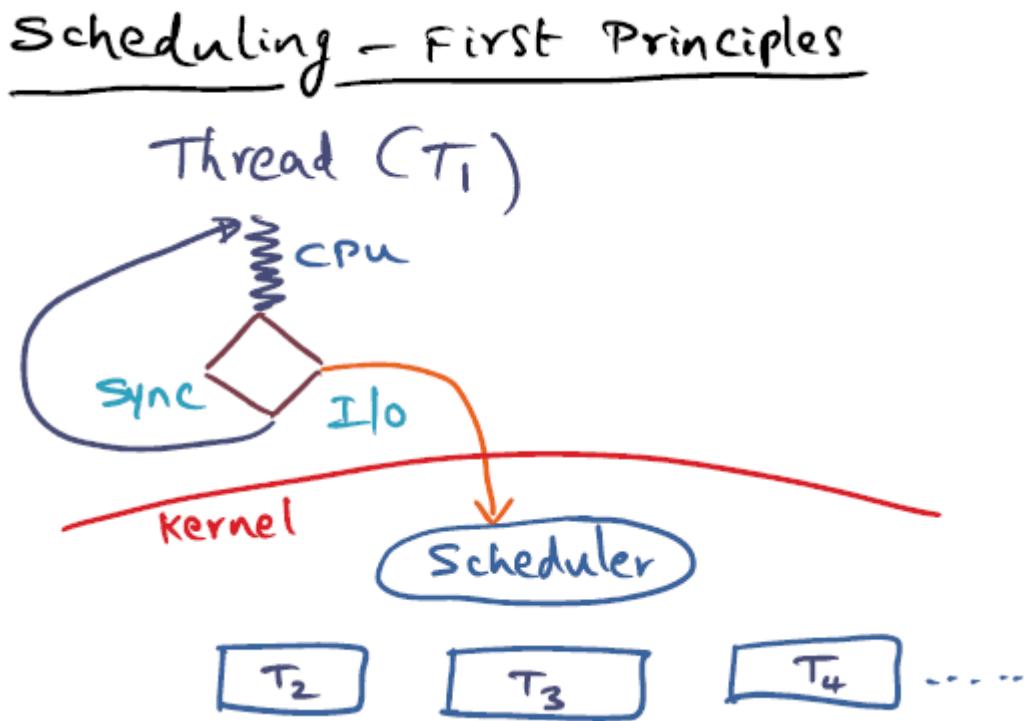
# scheduling

## Scheduling first principles

We are reminded that an effective way to increase performance, particularly for shared memory systems, is to keep the caches warm and leverage locality.

Typically, the normal execution of a thread is to compute for a while and then making a blocking system I/O call, attempt to synchronize with others threads, or, if its a compute-bound thread, its time quantum will expire. Fundamentally, this is a point in which the scheduler must make a decision to pick another thread to execute on the processor.

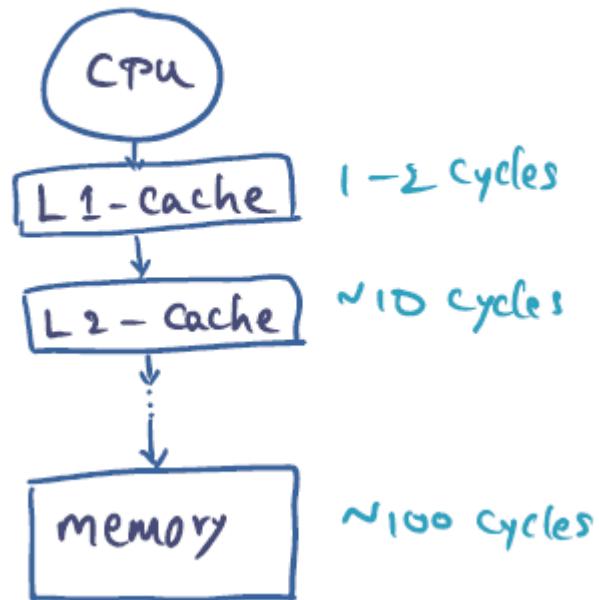
Below is a high-level representation of the dilemma discussed above.



## Memory hierarchy refresher

Below is a high-level representation of the memory hierarchy within a computing system.

## Memory hierarchy refresher



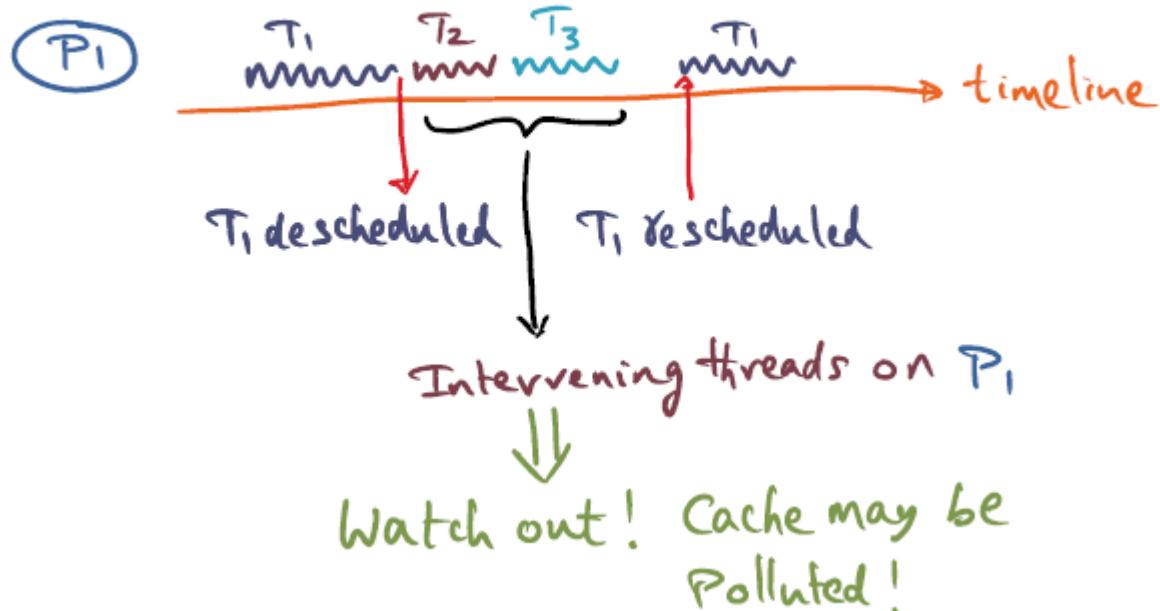
## Cache affinity scheduling

The concept covered here is that, if a thread is descheduled it makes a lot of sense to reschedule the thread back onto the same processor in order to leverage the cache of the processor. This policy attempts to utilize the cache affinity of a thread for a cache being utilized by a processor.

A problem is raised when other threads execute on the processor and pollute the cache, harming the locality that the original thread attempts to leverage.

Below is a high-level representation of cache affinity.

## Cache affinity scheduling



## Scheduling policies

Policy	Description
first come, first server	ignores affinity for fairness
fixed processor	thread always executes on a fixed processor
last processor	processor will execute the thread that most recently ran on it
minimum intervening	for every thread, we store its affinity information and execute the thread on the processor that it has the highest affinity for

## Minimum intervening policy

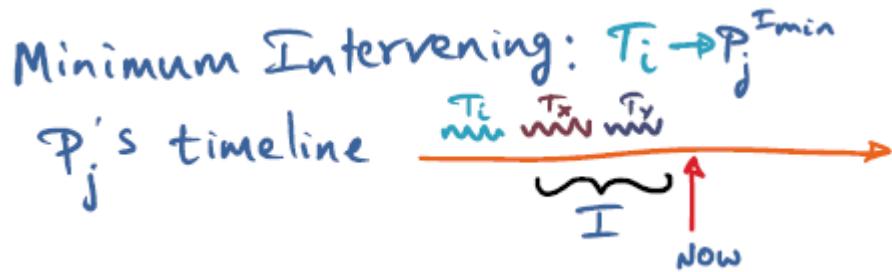
In the Minimum Intervening Policy, we determine the affinity of a thread for a processor by counting the number of threads that intervened on a processor after a thread executed on a

processor. So, if a thread was descheduled from a processor and then two other threads executed on a process, the intervention variable is now two (2).

The smaller the number, the higher the affinity for that particular processor. There is an affinity index associated for every processor in the computing system. The scheduler will pick a processor for the thread in which the intervention variable is the minimum.

If there are a large amount of processors, we enact a limited minimum intervening policy. This will only store the metadata for the top few processors, avoiding holding a ridiculous amount of information for a particular thread.

Below is a high-level representation of the minimum intervening policy and the calculation of the intervention variable for a thread's affinity index representing a particular processor.



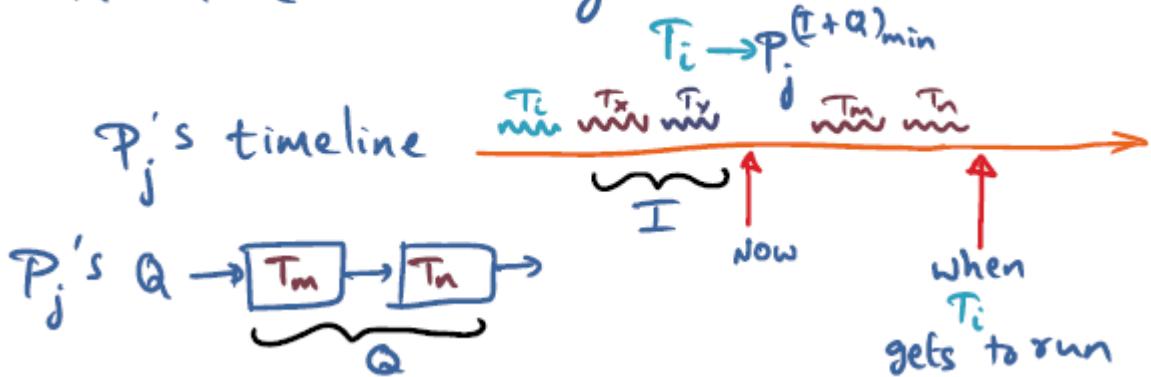
## Minimum intervening policy plus queue policy

This is still the same algorithm as the minimum intervening policy, however, when a scheduling decision is made, another thread could be running on the chosen processor.

In this policy, the scheduler maintains a queue of threads that will be executing on a particular processor. So, the intervention variable won't just be calculated from threads that have executed on the processor in the past, but also future threads that will be executing on the selected processor. This is done by viewing the queue being maintained for that particular processor.

Below is a high-level representation of the concept described above.

Minimum Intervening Plus queue:



## Summarizing scheduling policies

Below is a representation of all the scheduling policies discussed thus far.

### Scheduling Policies

FCFS : Ignores affinity for fairness

Fixed processor:  $T_i$  always on  $P_{\text{fixed}}$

Last processor:  $T_i$  on  $P_{\text{last}}$

Minimum Intervening:  $T_i \rightarrow p_j^{I_{\min}}$

Minimum Intervening Plus queue:

$T_i \rightarrow p_j^{(I+Q)_{\min}}$

Policy	Summary
first come, first served	focus on fairness
fixed processor	focus on cache affinity of thread

last processor	focus on cache affinity of thread
minimum intervening	focus on cache pollution when thread gets to run
minimum intervening plus queue	focus on cache pollution when thread gets to run

In the above table, the amount of information the scheduler has to keep increases as you traverse down the table. More information also allows the scheduler to make a better decision, as well.

## Implementation issues

Lets' discuss how an operating system would implement the previously discussed scheduling policies.

One possibility is the operating system maintains one global queue of the run-able threads. This makes sense for the implementation of the first come, first served scheduling policy. The global queue becomes infeasible when the number of processors becomes very large. The data structure becomes unmanageable, and contention increases as each processor attempts to access the data structure.

Another option is to keep local queues for each processor based on affinity. The organization of the local queue for each processor will depend on the scheduling policy being implemented.

There exists a specific equation to determine a thread's position in the queue or its priority.

$$t_i \text{priority} = BP_i + age_i + affinity_i$$

### Variable      Definition

affinity\_i      affinity if an affinity based scheduling policy is being implemented

BP\_i      base priority of the current thread when the thread was created

---

age <sub>i</sub>	how long the thread has existed in the computing system
------------------	---

---

Below is a high-level representation of the concepts discussed above.

## Implementation Issues

Queue Based

- Global Que



- Affinity-based local queues



$$T_i \text{ 's Priority} = BP_i + Age_i + Affinity_i$$

- Determines position in the queue

## Performance

Each scheduling policy will be rated on its performance based upon the metrics below:

---

Figure of merit	Definition
-----------------	------------

---

Throughput how many threads get executed or completed per unit of time?

Response time user centric; if a thread is started, how long does it take to start?

Variance user centric; does the time it takes to run a thread vary depending upon when the thread is run?

First come, first served scheduling is fair, however, it doesn't account for affinity or the size of a job. This results in a high variance.

Regarding the memory footprint, the larger the memory footprint of a thread, the longer it takes for the working set of a thread to be placed in the cache. This signifies to us that cache affinity is important for performance. The minimum intervening policies are great policies when the multiprocessor is handling a light to medium load.

If a very heavy load exists, it is likely that by the time a thread gets a chance to run on a processor, all of the cache has been polluted. For a heavy load, the fixed processor scheduling policy is best.

Essentially, we as operating system designers need to pay attention to the load on the multiprocessor as well as the kind of workload we intend to cater to. These will be the deciding factors for implementing a scheduling policy. Most operating systems dynamically change their scheduling policy based on these heuristics.

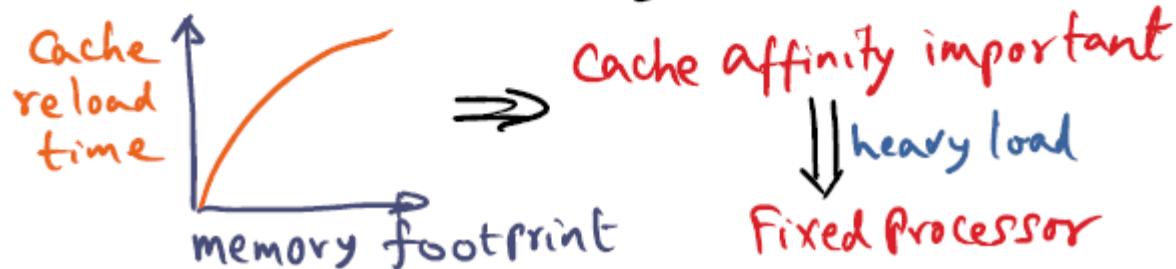
There also exists the idea of procrastination. The processor is ready to do some work, however, it inserts an idle loop. Why would this occur? A processor would read the runqueue and realize there exists no thread that has run on it before. If it schedules any one of those threads, none of those threads will have their working set loaded within the cache of the processor.

Below is a high-level representation of the concepts discussed above.

## Performance

### Figures of merit

- Throughput → System Centric
- Response time } → User centric
- Variance }



### Cache affinity and multicore

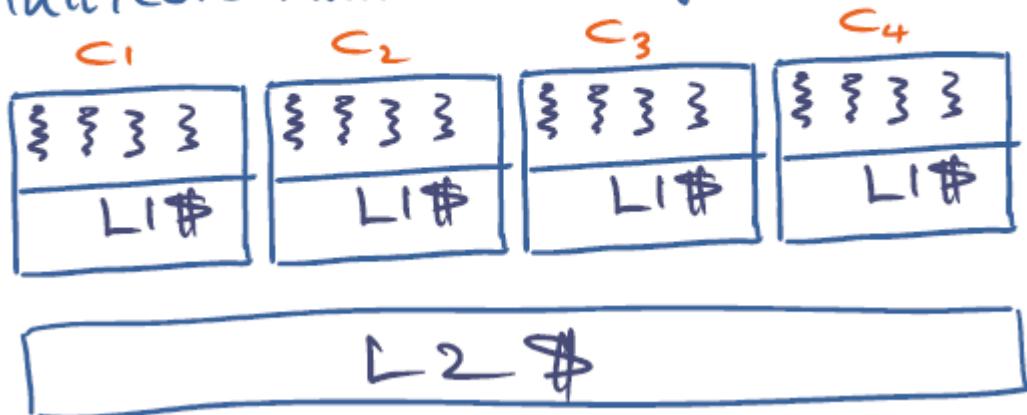
In modern multicore processors, each processor is hardware multithreaded. If a thread currently running on a modern multicore processor experiences a long latency operation, like a cache miss that requires the system to retrieve the value from memory, the hardware will switch to another thread and execute. This is done without operating system intervention.

As operating system designers, schedulers should leverage the structure of the multicore multithreaded processors and attempt to schedule threads that have cache affinity for a specific core. Schedulers want to try and ensure that the working set of a thread is contained in either the L1 or L2 cache.

Below is a high-level representation of the modern multicore multithreaded processor structure.

## Cache Affinity and Multicore

Multicore Multithreaded Processors



## Cache aware scheduling

When conducting cache aware scheduling, the operating system should schedule a number of cache hungry threads and cache frugal threads - the overall memory footprint of the threads being scheduled should be no larger than the size of the L2 cache. The operating system scheduler should be cache aware and attempt to leverage as much of the L2 cache as possible, but also avoid scheduling threads with large memory footprints that will cause other threads to have to conduct memory accesses.

How does a scheduler know which threads are cache frugal or cache hungry? We have to collect these heuristics over time as the threads execute. The scheduler won't initially know the memory footprint of a thread, however, the scheduler will maintain heuristics of a thread to make better scheduling decisions in the future. The overhead for this information gathering has to be kept at minimum in terms of time, of course.

Below is a high-level representation of the cache aware scheduling concept.

## Cache Aware Scheduling



- Cache frugal threads -  $C_{ft}$

- Cache hungry threads -  $C_{ht}$

$$\sum_1^n C_{ft} + \sum_1^m C_{ht} < \text{size}(L2*)$$

$$m+n = 16$$

## Definitions

## Quizzes

How should the scheduler choose the next thread to run on the CPU?

- first come, first served
- highest static priority
- highest dynamic priority
- thread whose memory contents are in the CPU cache

Why are all the choices valid?:

- First come, first served addresses fairness.
- Highest static priority addresses the fact that a customer is paying for their job to execute with highest priority.
- Dynamic priority changes due to the interactivity of the load - scheduler may want to boost the priority of a process.
- This last choice addresses the attempt to leverage locality - the thread with a cached working set is likely to do well.

Which processor will we enqueue Ty for based upon the scheduling policy?

### Question

$P_u$ 's Q  $\rightarrow T_x$

$P_v$ 's Q  $\rightarrow T_m \rightarrow T_y \rightarrow T_r \rightarrow T_n$

$T_y$ 's affinity:  $P_u^T = 2$ ;  $P_v^T = 1$  } Num Intervening tasks

Scheduling Policy: Min Inter. plus q

Which Q for  $T_y$ ?

- Pu's queue
- Pv's queue

We enqueue  $T_y$  onto  $P_u$ 's queue because, by the time  $T_y$  executes on  $P_v$ , the number of intervening tasks will be 5. The number of intervening tasks on  $P_u$ 's queue by the time  $T_y$  executes will be 3.

# shared memory multiprocessor operating system

## Operating system for parallel machines

Modern parallel machines face a lot of challenges when attempting to implement the algorithms and techniques discussed previously in a scalable manner. What are some of these challenges?

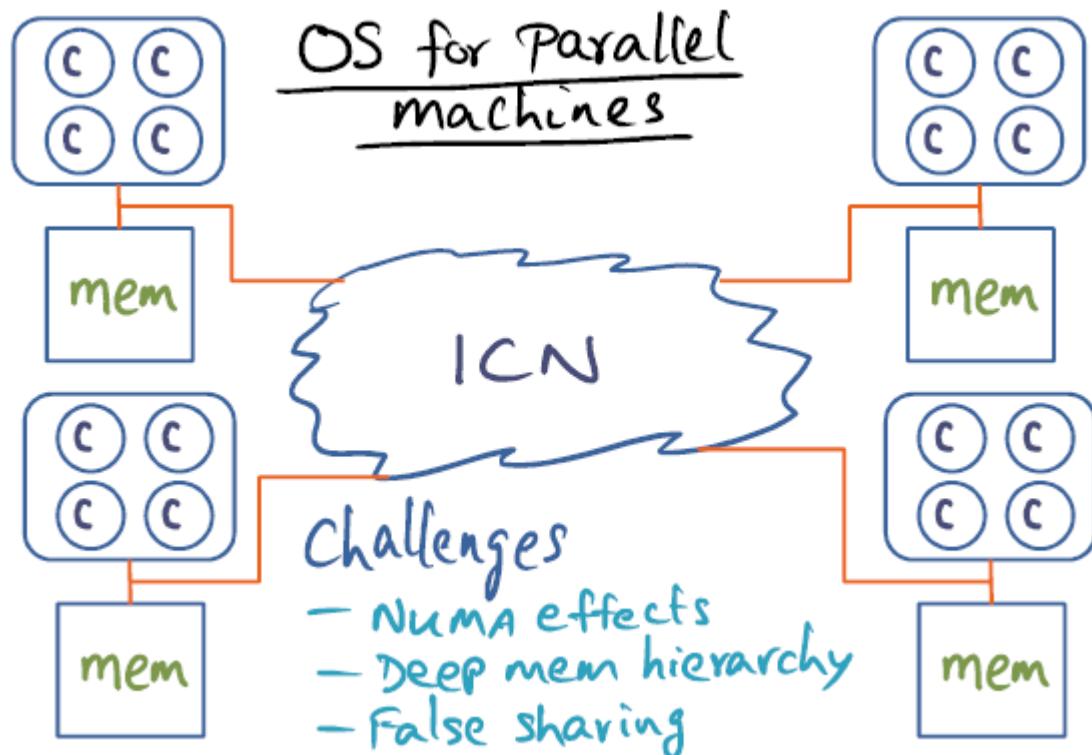
First, the size bloat of the operating system due to the additional features that need to be added to the operating system. This results in system software bottlenecks, especially for global datastructures.

In addition, the memory latency that is incurred when processors need to conduct a memory access versus utilizing the cache - 1:100 ratio of time required to conduct a memory access. Now let's take into account the fact that we're attempting to design a distributed system using some interconnection network with Non-Uniform Memory Access (NUMA). Memory accesses by a processor can be done locally, but they can also be done across the interconnection network, as well, further increasing the latency of memory accesses.

Another challenge we encounter is the deep memory hierarchy that exists for each node. Modern processor cores contain an L1, L2, and L3 cache.

Finally, there's the issue of false sharing. Even though two threads running on two different cores of a multiprocessor are programatically unrelated, the cache hierarchy may make the block that contains the individual memory location being utilized by these two different cores to be on the same cache block. The processors are executing completely unrelated operations and appear to be sharing locations in the cache, however, they are actually false sharing memory locations within the cache.

Below is a high-level illustration of a parallel machine, its nodes, and the challenges faced when designing operating systems for parallel machines.



## Principles

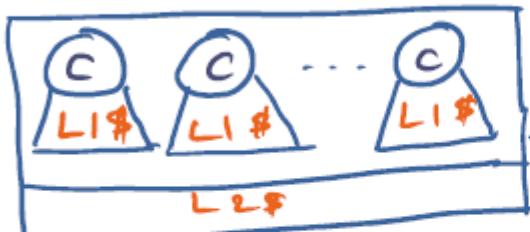
Let's discuss some general principles that should be followed by operating system designers when designing operating systems for parallel machines.

Principle	Description
Cache conscious decisions	leverage locality and exploit cache affinity for scheduling decisions
Reduce shared data structures	limit shared system data structures to reduce contention for memory locations
keep memory access local	reduce the distance between the processor and the memory its attempting to access

Below is a high-level representation of the principles described above.

# Principles

Cache Conscious decisions



Limit shared System Data structures  
Keep memory accesses local

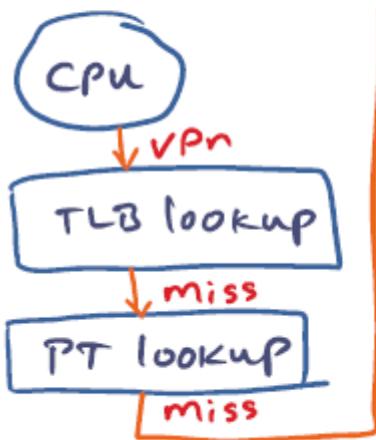


## Refresher on page fault servicing

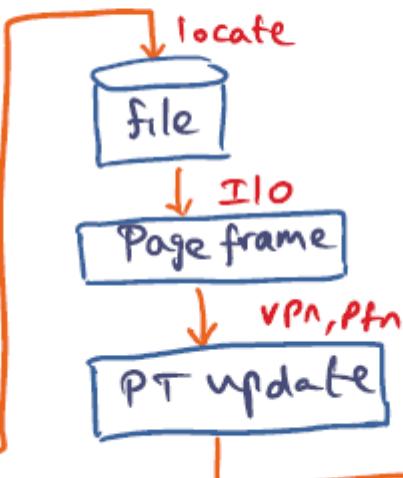
Below is a high-level representation of how a page fault is serviced.

### Refresher on page fault service

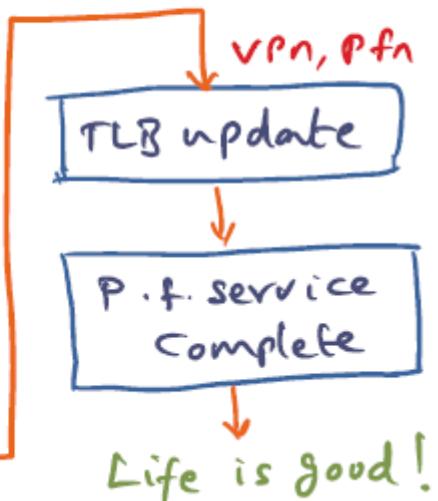
Thread Specific



Avoid  
Serialization



Processor Specific



The main takeaway here is we want to avoid serialization. TLB and page table lookups only occur for a specific thread and can be conducted in parallel. The same goes for TLB updates, as these actions are processor specific.

Reading a page from memory, conducting an I/O operation to do this, creating a page frame for the newly swapped memory, and conducting a page table update takes a lot of time and can't be done in parallel. We want to avoid this serialized set of actions as much as possible to avoid bottlenecks on the parallel system.

## **Parallel operating systems and page fault servicing**

Let's take into account an easy example scenario. Two threads exist with independent workloads. Both threads experience a page fault. Because there are no shared memory data structures between the two threads, no serialization will be experienced.

Now let's take into account a hard example scenario. We have multiple threads executing within a process, all sharing the same address space. The parallel operating system takes advantage of the concurrency being implemented for this process and executes threads from the process on two different nodes within the system. In this scenario, the address space and page table is shared across each thread and the TLBs are shared for each thread on a particular node.

What we would want to do for the scenario above is to limit the amount of sharing of operating system data structures for the threads running on different nodes. The operating system data structures that the segregated threads interact with should be distinct and separate. This will help to ensure scalability for the parallel operating system.

Below is a high-level representation of the scenarios and concepts discussed above.

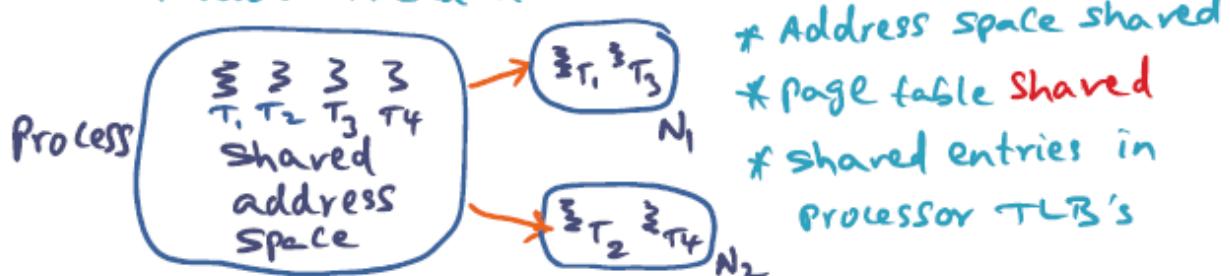
## Parallel OS + page fault service

### Easy scenario

- multiprocess workload \* Threads independent
- 
- $N_1$   $T_1, T_2, T_3, T_4$  ...  $N_n$   $T_n$
- \* page tables distinct
  - \* no serialization

### Hard scenario

- multi-threaded workload



## Recipe for scalable structure in a parallel operating system

The recipe for designing a scalable, parallel operating systems will be discussed in this section. First, for every subsystem we design in a parallel operating system, we should consider these concepts:

- Determine, functionally, what needs to be done for the service / subsystem we are implementing. In the systems being discussed, the hardware is already parallelized, completing most of the hard work for us.
- To ensure the concurrent execution of the service, however, we should minimize the shared data structures. Less sharings equals more scalability.
- Depending on the usage of a data structure, we want to replicate and partition data structures where possible. This provides more concurrency and less locking of shared data structures.

## Tornado's secret sauce

Tornado achieves the scalability discussed above using a concept called the clustered object. The key concept is that Tornado provides the illusion of a single object to each piece of the operating system, however, when a piece of the operating system references said

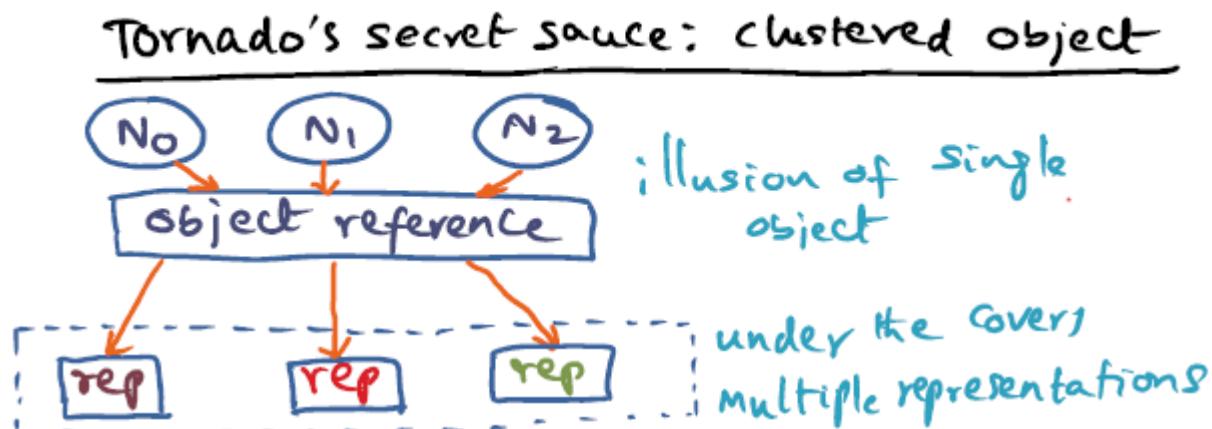
object it is actually using one of the many representations of the object. These references are separated by nodes.

What's the degree of clustering? How is that defined? What is the granularity? The designer of a system service for Tornado has the ability to make this decision. The designer can choose a:

- singleton representation
- one per core representation
- one per CPU representation
- one per group of CPUs

Now, if we have replicated representations of an object, we need to ensure the object maintains consistency across the different nodes. Tornado suggests using protected procedure calls to conduct updates of clustered objects to maintain consistency.

Below is a high-level representation of a clustered object.



Degree of clustering?

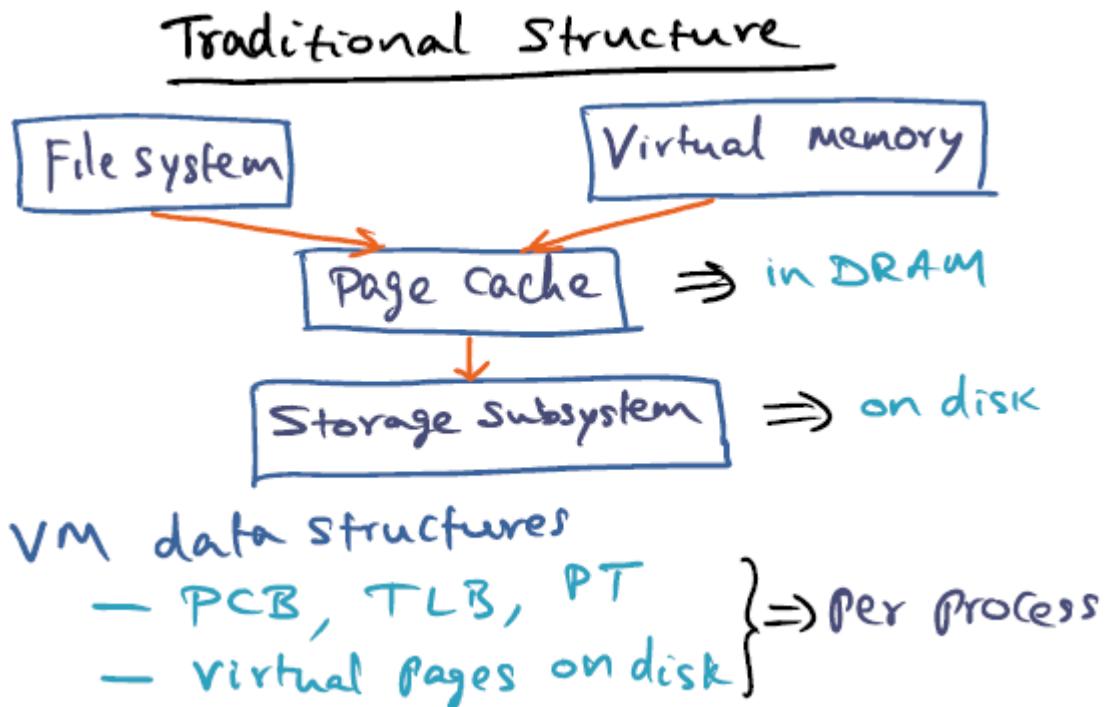
- choice of implementor of service
  - \* Singleton rep, onepercore, oneperCPU, one per group of CPUs, ...
- PTC for consistency

**Traditional structure**

This is just a quick review of the traditional memory hierarchy used for memory management. Covers file system descriptors hosted in the page cache (in RAM), the storage subsystem, and the representations of processes' virtual memory to include their process control block, software translation lookaside buffer, page table, and virtual pages on disk.

In an effort to achieve scalability, we want to eliminate as many of the centralized data structures as possible.

Below is a high-level representation of the concepts discussed above.



## Objectization of memory management

Now that we are abstracting away parts of the operating system into objects, we'll start with the process object. The process object is similar to the process control block and represents the address space of a process used by multiple threads of execution on the same CPU.

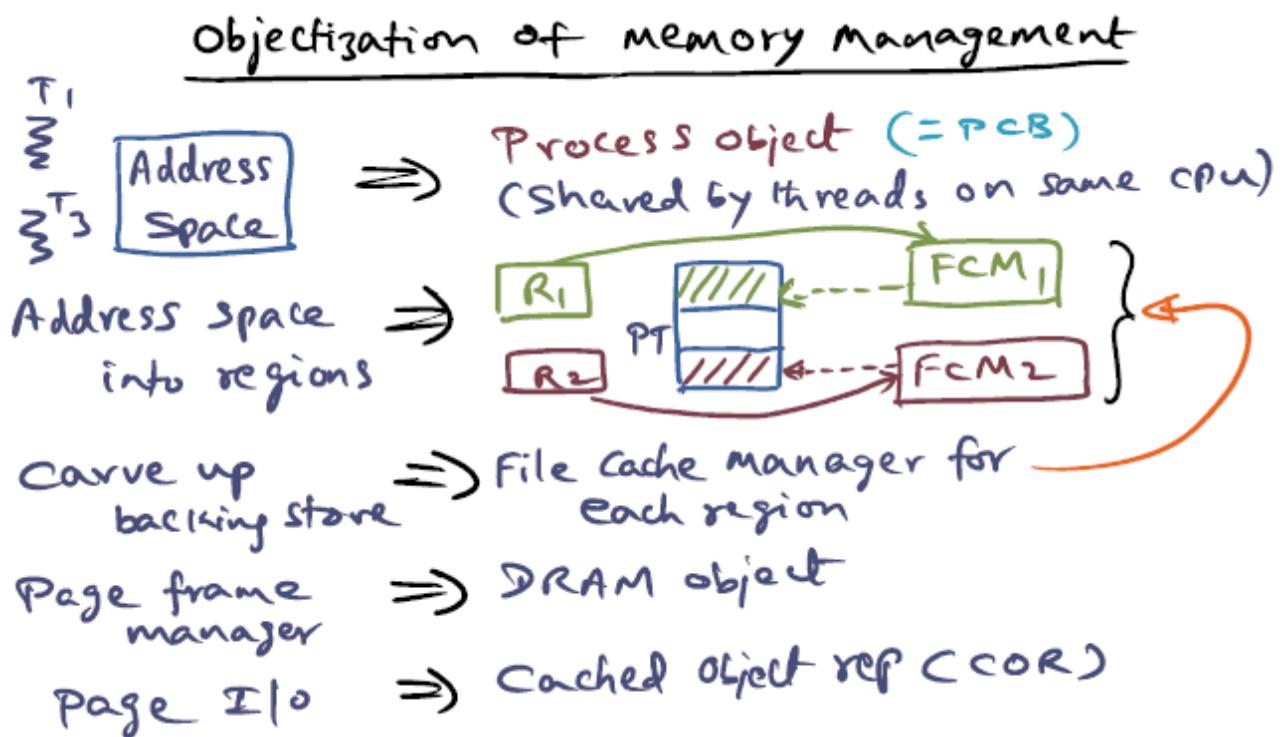
In order to practice the concepts discussed above, the address space (process object) will be broken into different regions. Threads don't execute over the entire address space of a process, so we don't need to maintain the entire address space for each thread on a different node.

With these regions being defined, another object is created called the File Cache Manager that backs the memory locations of each region.

Another object, the page frame manager (DRAM object), acts as a page frame service. When the page fault service needs a page frame, it contacts the page frame manager to acquire the contents of the File Cache Manager from the storage subsystem for a particular region.

Finally, there is the cached object representation (COR) that is responsible for knowing the location of the object the page frame manager is looking for on the backing storage subsystem. The COR conducts the actual page I/O.

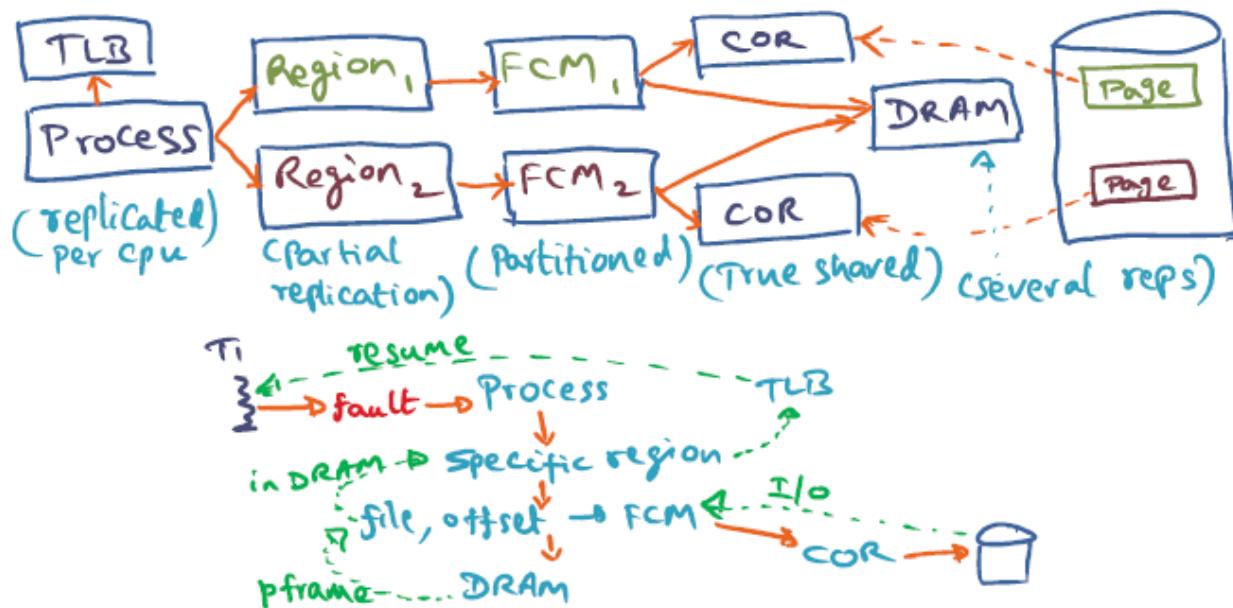
Below is a high-level representation of the concepts discussed above.



## Objectized structure of VM manager

Below is a high-level representation of the objectized memory management service described above. Included is the page fault flow within an objectized memory management service, as well as how each object within the service could be clustered.

## Objectified structure of VM manager



## Advantages of clustered objects

Clustered objects provide a lot of advantages. Here are some examples:

- Allows incremental optimization
- Usage pattern determines level of replication
- Usage pattern allows for different implementations
- Less locking of shared data structures

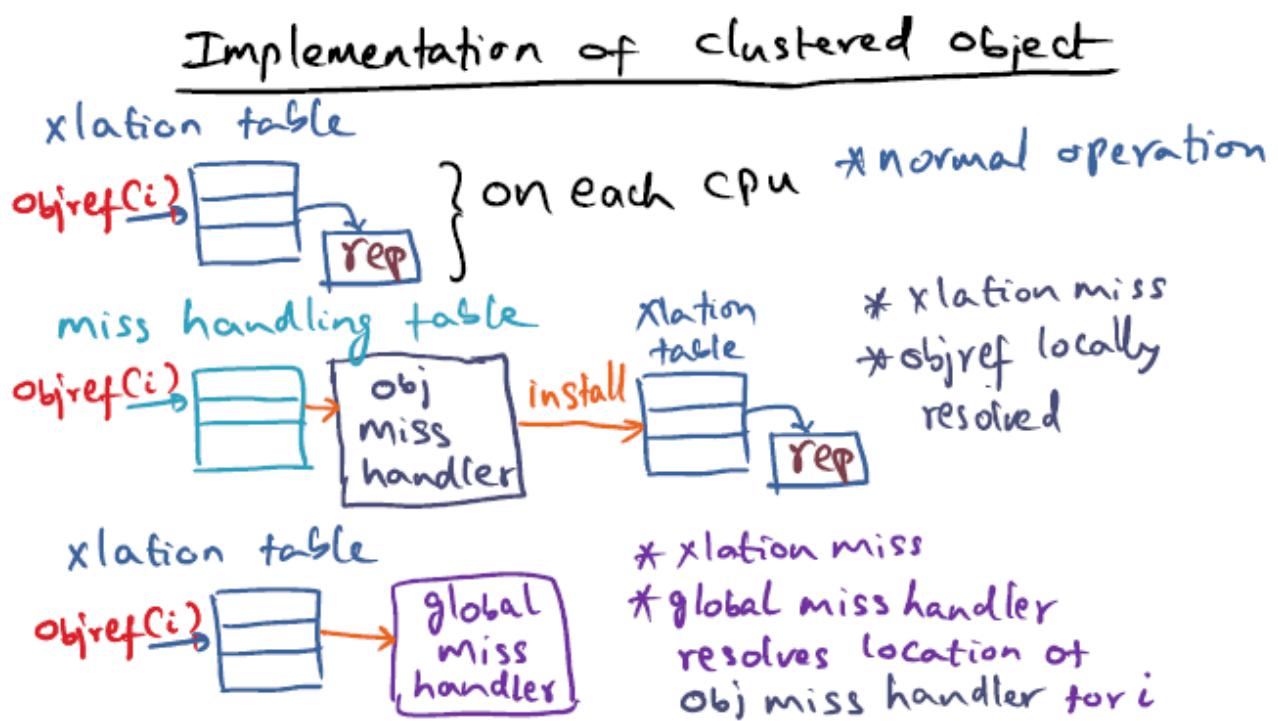
## Implementation of clustered objects

Each clustered object reference is accompanied by a translation table. The translation table maps a reference to a representation in memory. There also exists the possibility that an object reference incurs a miss. When this happens, the object reference is added to the miss handling table, and the object miss handler kicks in.

The object miss handler determines if the object reference requires a new representation be created in the translation table or if the object reference is referring to an already existing representation. Once a decision is made, the object miss handler will install the object reference to representation mapping in the translation table.

There is a possibility that the object miss handler may not be local. This is because the miss handling table is a partitioned data structure, it doesn't exist for every node. This displays the necessity for a global miss handler. If the miss handling table does not have an object miss handler for a specific object reference, we use the global miss handler. The global miss handler exists on every node, knows the location of the miss handling table for each object reference, and can obtain representations of object references across all nodes. The global miss handler will then populate the reference to representation mapping in the local translation table for a node.

Below is a high-level representation of the concepts described above.



## Non-hierarchical locking and existence guarantee

Hierarchical locking of objects kill concurrency. If we want integrity for objects of course we want to lock critical data structures, however, if the path taken by a thread for a page fault is different from another thread, there exists little reason why the page fault servicing can't be done in parallel - without locking the critical data structures.

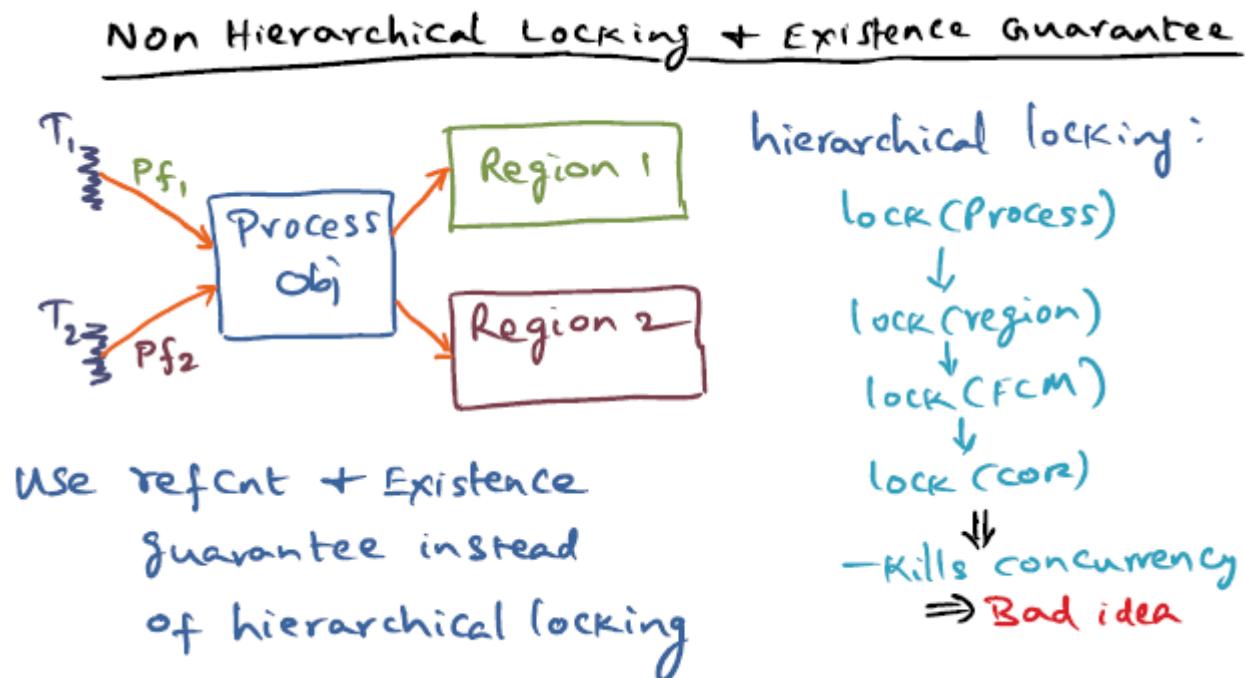
So we still want integrity of the process object and there's a possibility we need to lock it so it can't go away. How can a process object just go away? It's possible the operating system has decided to migrate a process object from one processor to another processor.

To resolve this issue, we instantiate a reference count for the process object. The operating system will utilize the reference count and existence guarantee instead of hierarchical locking. So when a thread utilizes the process object, it increments a reference count for the process object denoting that it is in use. Now other operating system entities will know not to modify or migrate the object because the reference count is not 0.

This non-hierarchical locking mechanism included with the existence guarantee promotes concurrency.

For hierarchical locking, locks should be encapsulated in the individual object representation, not across all representations for an object reference. This reduces the scope and limits contention for the lock. The service provider should also ensure the integrity of a replicated region through protected procedure call. Even though the hardware provides cache coherency, there's no way to guarantee replicas of a region are consistent.

Below is a high-level representation of the concepts discussed above.



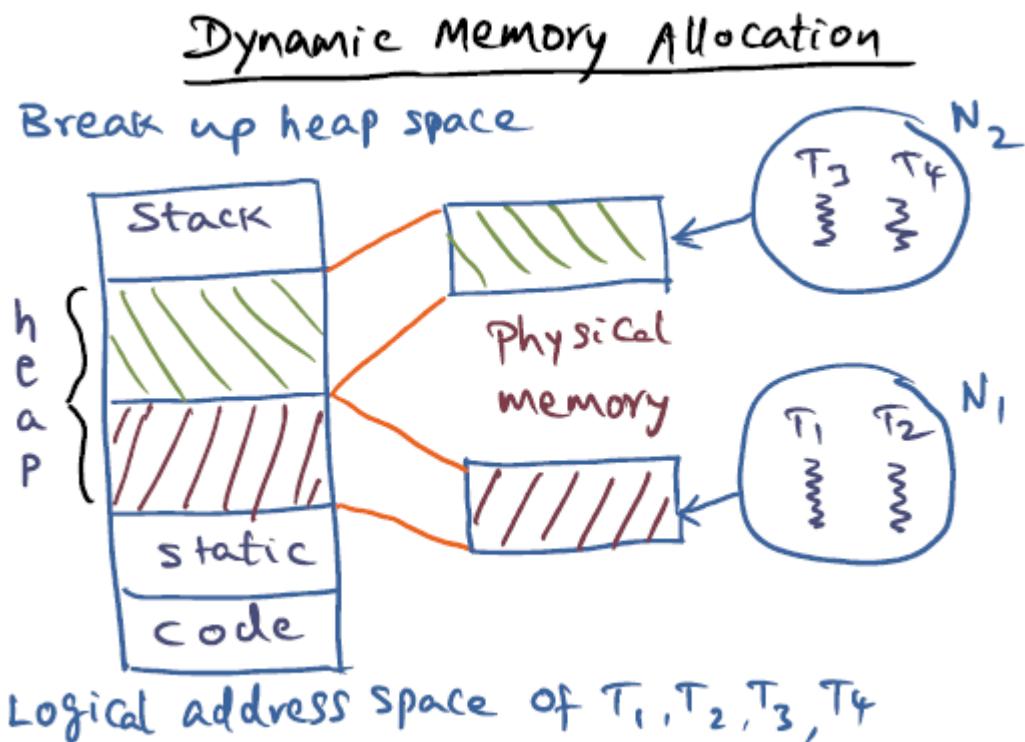
## Dynamic memory allocation

It's important to make sure that memory allocation is scalable and this includes the heap space of processes. One possibility is to take the heap space of the process and partition it for the multiple threads executing within the process on separate nodes. We'll associate

the portions of the heap that reside within physical memory and associate them with the nodes of the threads utilizing the partitioned heap space.

For NUMA machines, this removes the bottleneck that exists for threads when they attempt to acquire memory from the heap. Now, instead of allocating memory from some central location, they can acquire the memory from the physical memory residing on their local node. This mechanism also avoid false sharing of memory between the threads executing on separate nodes.

Below is a high-level representation of the concepts discussed above.



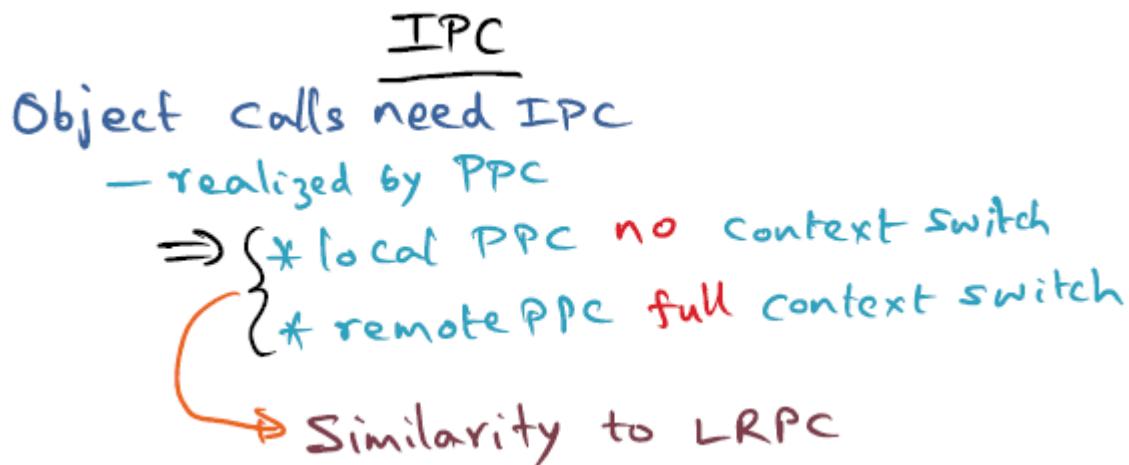
## IPC

The functionalities of the Tornado operating system are contained within the clustered objects we described earlier. These clustered objects must communicate in order to implement the services of the parallel operating system. We need some form of interprocess communication between these objects, with the thinking that some objects can act as a client, and other objects can act as a server.

So all of this interprocess communication is, again, realized by the protected procedure call mechanism. If the protected procedure call is conducted between a client and a server executing on the same processor, no context switch is required. If the protected procedure call is conducted between a client and a server executing on different processors, a remote protected procedure call, a full context switch is incurred.

Tornado uses this IPC mechanism to ensure clustered objects and their replicas remain consistent across all nodes. This has to all happen within software because, even with a cache coherent system, the replicas are not guaranteed to be consistent across all nodes because they are software based.

Below is a quick sketch of the concepts discussed above.



## Tornado summary

Tornado is an operating system designed for parallel systems. These are its features:

- Object oriented design for scalability. Clustered objects and protected procedure call attempts to preserve locality while also ensuring concurrency.
- Multiple implementations of operating system objects. Allows for incremental optimization and dynamic adaptation of object implementations.
- Use of reference counting to avoid hierarchical locking of objects. Locks held by an object are confined to the specific representation of the object.
- Tornado also works to optimize the common case:
  - page fault handling

- Limiting the sharing of operating system data structures by replicating critical data structures to promote scalability and concurrency.

## Summary of ideas in Corey operating system

The central principle in structuring an operating system for a shared memory multiprocessor is to limit the amount of sharing for kernel data structures. The sharing of kernel data structures limits concurrency and increases contention.

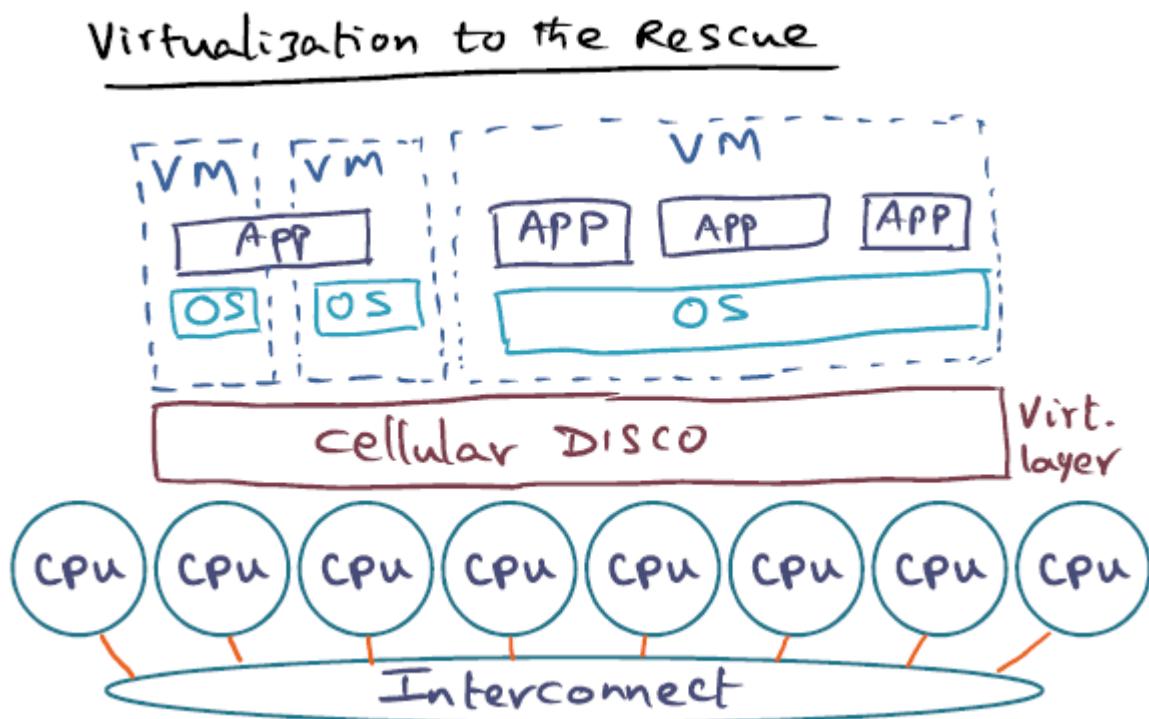
The Corey operating system attempts to prove this principle in its implementation. The Corey operating system provides mechanisms for the applications to give hints to the kernel. The Corey operating system is similar to Tornado in that it attempts to reduce the amount of sharing of kernel data structures. Below is a comparison of features between the Corey operating system and Tornado.

Corey	Tornado
Address ranges in an application; exposed to the application; threads hint to the kernel where it is going to operate	Regions within an application; transparent to the application
shares; threads hint to the kernel the system data structure they intend to use; threads can communicate their intent through the shares mechanism (whether it be private access or access by multiple threads)	
dedicated cores for kernel activity; allows us to confine the locality of kernel data structures to dedicated cores	

## Virtualization to the rescue

Cellular Disco was a study at Stanford in pursuit of finding an operating system that would leverage virtualization in tandem with the multiprocessing nature of the underlying hardware.

Cellular Disco was a thin virtualization layer that managed hardware resources such as the CPU, I/O devices, memory management, etc. One thing that is always difficult in operating system design is I/O - most of the device driver code that resides within an operating system is third party code written specifically for the I/O device. Cellular Disco attempts to tackle this issue by abstracting away I/O management for the virtualized systems by construction. Below is a high-level representation of Cellular Disco on a multiprocessor system.

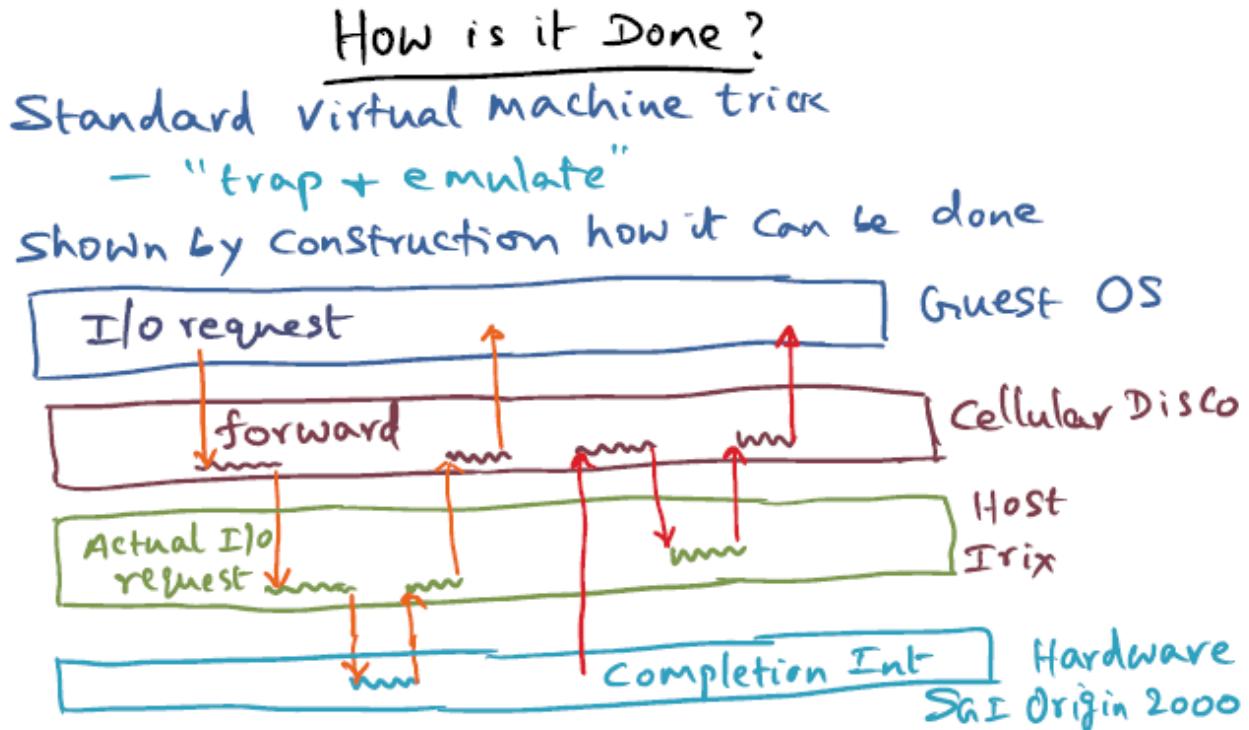


Cellular Disco abstracts I/O operations for the guest operating system by utilizing the usual trap and emulate mechanism of virtual machine monitors. When guests send I/O requests, Cellular Disco provides that to the host operating system for management, leveraging the already existing I/O subsystem that will resolve interaction with the device drivers. When an external I/O interrupt is received, Cellular Disco has identified itself as the interrupt handler, mimics the external I/O interrupt to the host operating system, and then the host operating system routes the external I/O interrupt back to Cellular Disco for handling - eventually the external I/O reaches the guest operating system.

The purpose of Cellular Disco was to prove by construction how to develop an operating system for new hardware without completely re-writing the operating system. They also prove that a virtual machine monitor can manage the resources of a multiprocessor as well as a native operating system. They manage to mitigate the amount of overhead required to

conduct the trap and emulate mechanism, maintaining efficiency for the applications executing within the guest operating system.

Below is a high-level representation of the concepts described above.



lesson5

# **lesson5**

# definitions

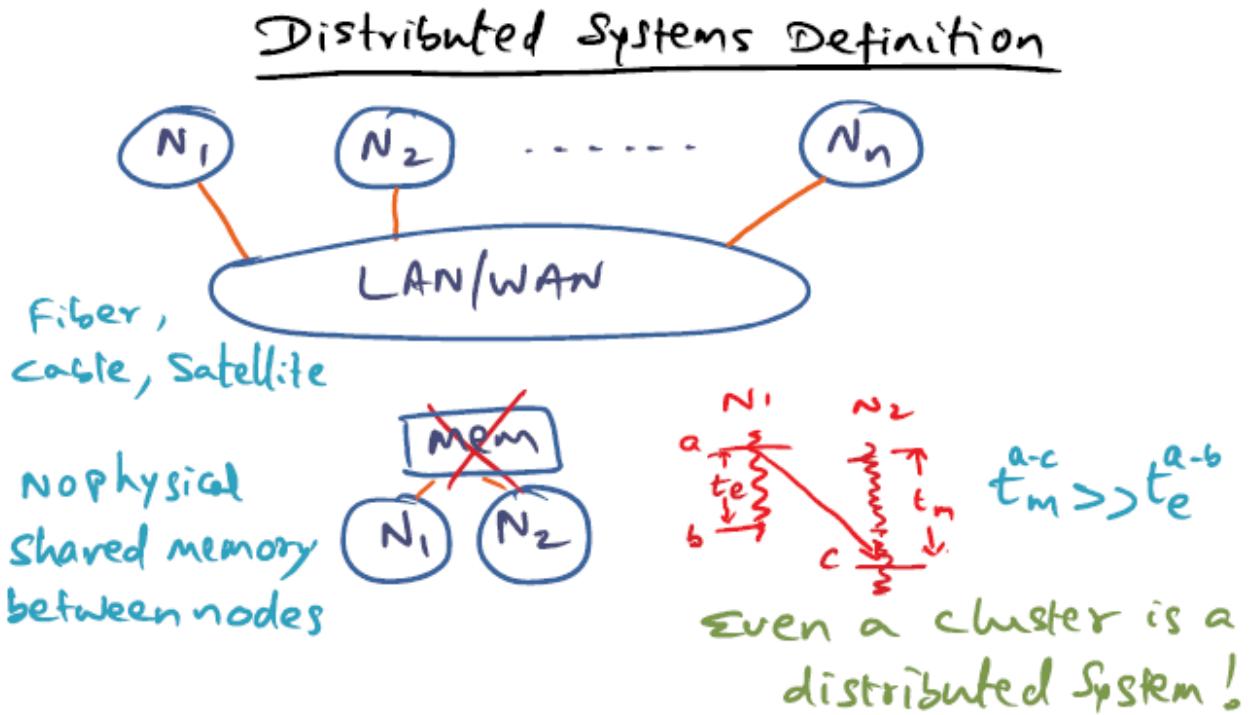
## Distributed system definition

A distributed system is a collection of nodes interconnected by a Local Area Network or Wide Area Network. There is no physical memory for communication shared between the nodes of the distributed system. The time for communication between nodes within the system is significantly larger than the event-computation time.

**Lamport's Definition of a Distributed System** - a system is distributed if the message transmission time is not negligible compared to the time between events in a single process.

The importance of the inequality between these two times is that this inequality dictates the design of the algorithms that are going to be utilized to implement the distributed system.

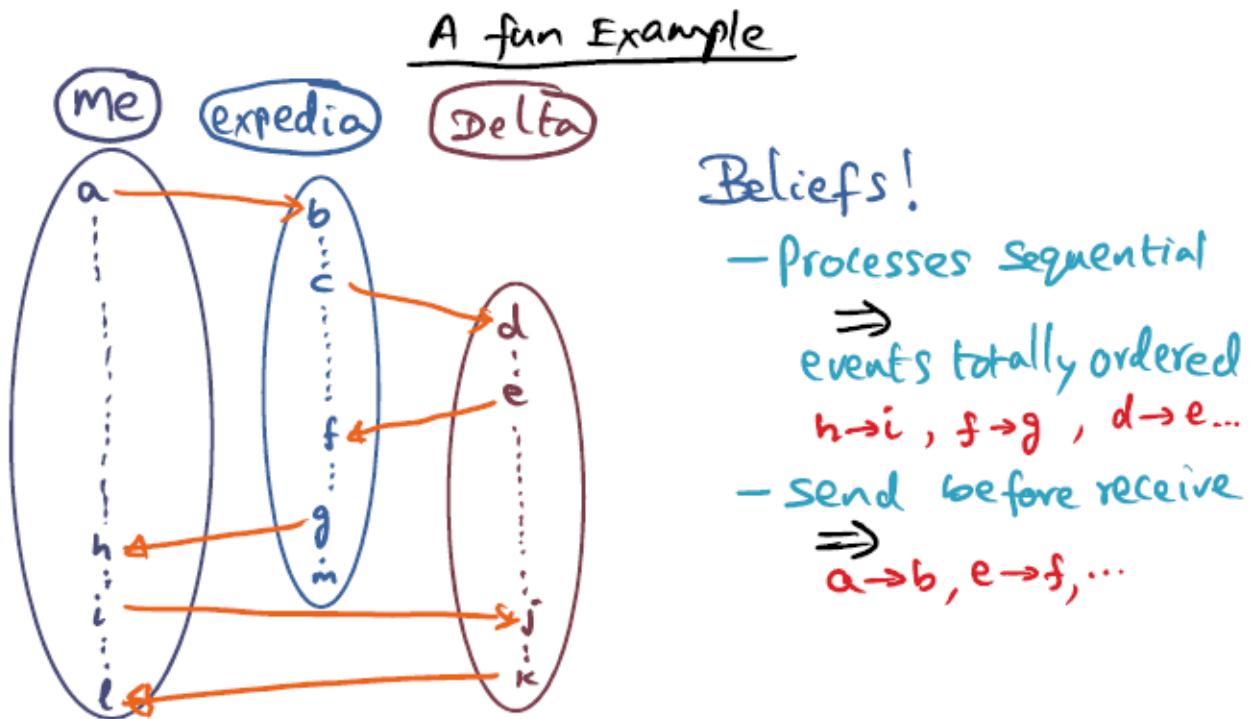
Below is a high-level representation of the concepts discussed above.



## An example

In distributed computing, there exist a set of beliefs or expectations that we have for the order in which operations are conducted. We expect processes within an exchange to be sequential, events to be totally ordered, and that sending occurs before receiving.

Below is a high-level representation of the concepts discussed above.



## Happened before relationship

The happened before relation and notation is just an assertion that a particular action took place before another particular action. This relation can be asserted for events that take place within a process, as well as across a distributed system when involved with communication between nodes.

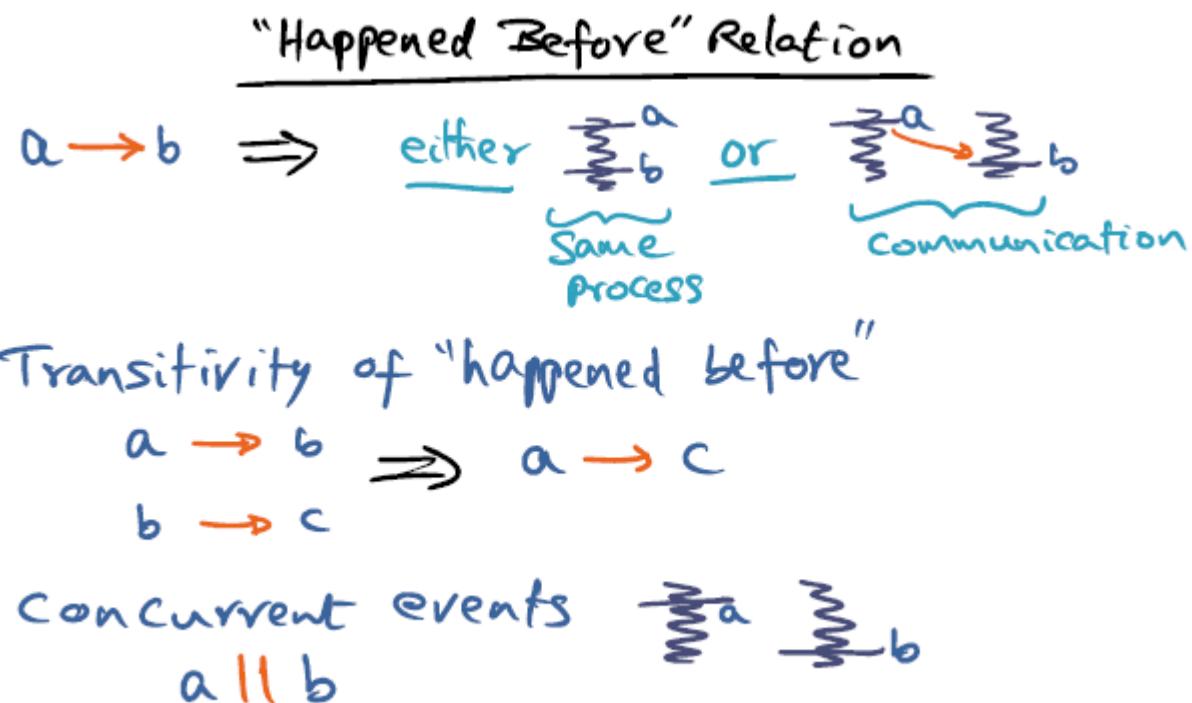
The "happened before" relationship is also transitive. A particular action, **c**, takes place after action **b**, and action **b** takes place after action **a**, then action **c** takes place after action **a**.

Lastly, there are concurrent events, events in which there is no apparent relation. This usually occurs when events take place on separate nodes and there is no communication

between the events. We cannot say anything about the ordering of these events because they are not related in any way.

In the construction of a distributed system, it's important to understand and keep in mind these two types of events. Not doing so could be the cause of synchronization and timing bugs.

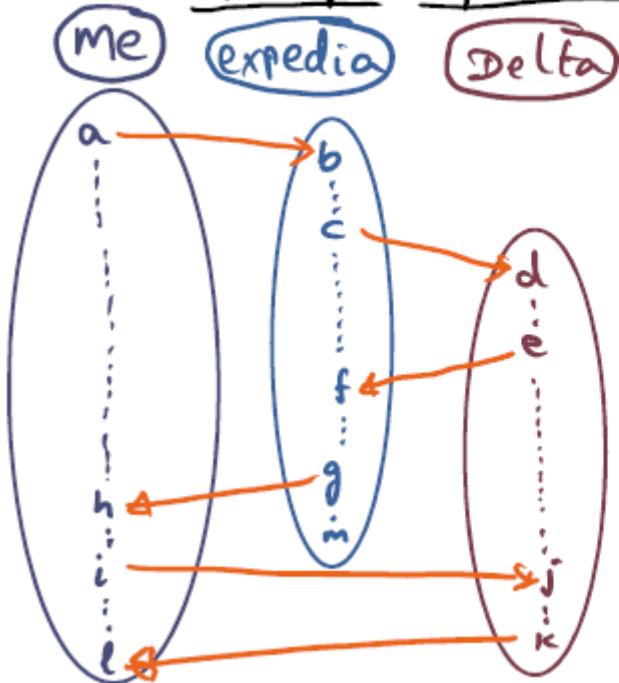
Below is a high-level representation of the concepts described above.



## Identifying events

Below is a the previous example of events taking place within a distributed system, with the "happened before" and concurrent events identified.

## Example of Event Ordering with →



Events connected by →

$$\begin{array}{ll}
 a \rightarrow h \rightarrow i \rightarrow l & a \rightarrow b \quad i \rightarrow j \\
 b \rightarrow c \rightarrow f \rightarrow g \rightarrow m & c \rightarrow d \quad k \rightarrow l \\
 d \rightarrow e \rightarrow j \rightarrow k & e \rightarrow f \quad g \rightarrow h
 \end{array}$$

Transitive →

$$a \rightarrow e, d \rightarrow m, \dots$$

concurrent events //

$$\begin{array}{ll}
 h \parallel m & m \parallel j \\
 i \parallel m & m \parallel k \\
 l \parallel m &
 \end{array}$$

## Definitions

Term	Definition
event-computation time	the time required for a node to conduct some significant processing
communication time	the time required for a node to communicate with another node within a distributed system
concurrent events	events in which there is no apparent relationship between the events

## Quizzes

### What is a distributed system?

- Nodes connected by LAN / WAN
- Communication only via messages
- A model where the communication time is longer than event-computation time

**Consider the following set of events:**

N1

f ->

b

N2

a

g

**What can you say about the relation between a and b ?**

- a -> b
- b -> a
- neither

You cannot say anything about the relation between a and b given the order of the operations above.

# Lamport clocks

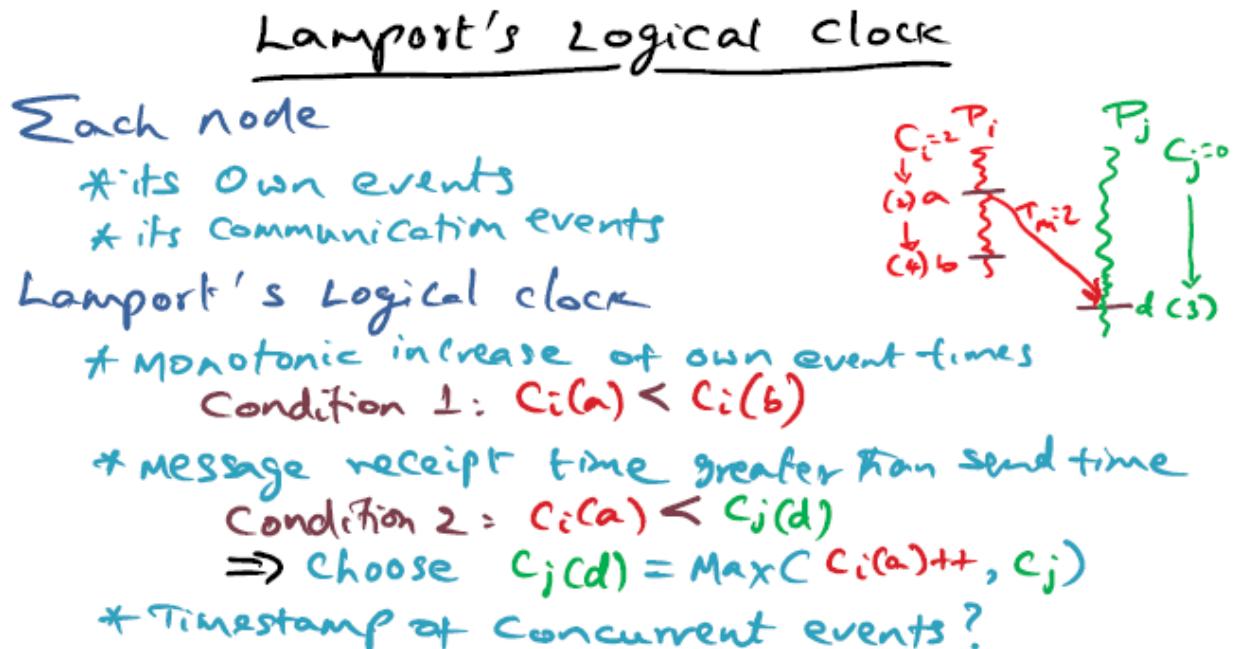
## Lamport's logical clock

Nodes within a distributed system are only aware of two kinds of events: its own events and its communication events.

Lamport's logical clock builds on this idea, attempting to associate a timestamp with every event that occurs on the node, or every communication event that occurs between peer nodes. A condition is that each timestamp of each event is unique and is increased monotonically for sequential events - no two events will have the same timestamp.

In Lamport's logical clock, timestamps are also generated for communication events, however this must be done in coordination with the target node. A condition exists in which the timestamp for the communication event occurring on the peer node is either greater than the timestamp for the source node, or the max timestamp number on the target node.

Below is a high-level representation of the concepts described above.

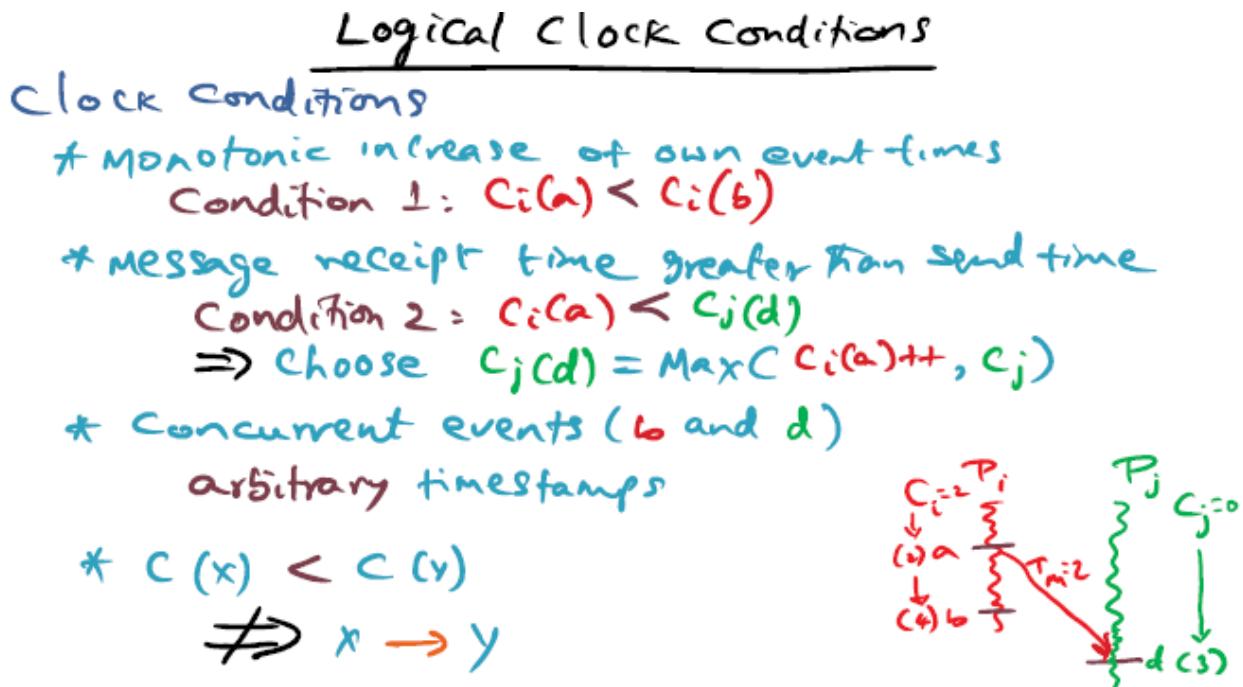


## Logical clock conditions

- The Lamport logical clock abides by the condition that there is a monotonic increase for all event times within a node.
- The Lamport logical clock abides by the condition that the message receipt time of a node will always be greater than the send time.
- The Lamport logical clock abides by the condition that it is possible for concurrent events to have arbitrary timestamps.
  - Just because there exists some event, x, that has a timestamp less than some event, y, this doesn't mean that they are related in the "happened before" relation.

Thus, Lamport's logical clock gives us a partial order for when events happened across the distributed system.

Below is a high-level representation of the conditions described above.



## Need for a total order

Will a partial order of events be enough to create deterministic algorithms for the implementation of a distributed system? It works for many situations, however, this section will discuss why a total order of events is needed for distributed system design.

In the example provided below, the professor describes the use of his car between his family, and how negotiation is conducted by timestamping messages requesting the use of

the car. If a tie occurs, all nodes agree upon a tie breaker - in this case being age.

An image of this example is provided below.



## Lamport's total order

In order to assert that some event,  $a$ , occurs before some other event,  $b$ , we must meet these conditions:

$$C_i(a) < C_j(b)$$

or

$$(C_i(a) = C_j(b)) \text{ and } (P_i << P_j)$$

The tie break variable must be a well known condition within the distributed system. So, there is no single total order. There order is only derived from the tie break variable that is established within the distributed system.

Below is a high-level representation of the concepts discussed above.

Lamport's Total Order

Condition for  $\Rightarrow$

$a \Rightarrow b$  iff

$$C_i(a) < C_j(b)$$

or

$$C_i(a) = C_j(b) \text{ and } P_i \ll P_j$$

$\ll$  arbitrary "Well known" condition  
to break a tie)

No single total order

## Distributed mutual exclusion lock algorithm

So how do we create a distributed mutex using Lamport's clock? We don't have shared memory to implement the lock, so we have to conduct this across an entire network using messages. So here's how the algorithm works using Lamport's Logical Clock:

- Every process maintains its own private queue data structure.
- The private queues are ordered by the "happened before" relationship discussed earlier.
- Requests for a mutex are going to be timestamped.
- To request a lock, a process will broadcast to all other processes the lock request and its respective timestamp.
- The requestor will enqueue this request into their local queue (ordered by timestamp).
- After saving the received broadcast into their private queues, all the processes will acknowledge to the initial requestor the request for the mutex.
- Ties are broken by Process ID.

It is possible for the queues between each process to not be consistent or the same. This is due to the fact that the requestor will save their request prior to broadcast, and the latency incurred when broadcasting the request means that the queues between two process will not be the same until the broadcast is received.

So how does a process know that it holds the mutex?

- A process will check to see if its request is at the top of the queue. If so, it can begin to assume it holds the mutex.
- A process has received acknowledgements from all the other nodes in the system.
- A process has received lock requests from other processes that occur later in the queue.

So what?

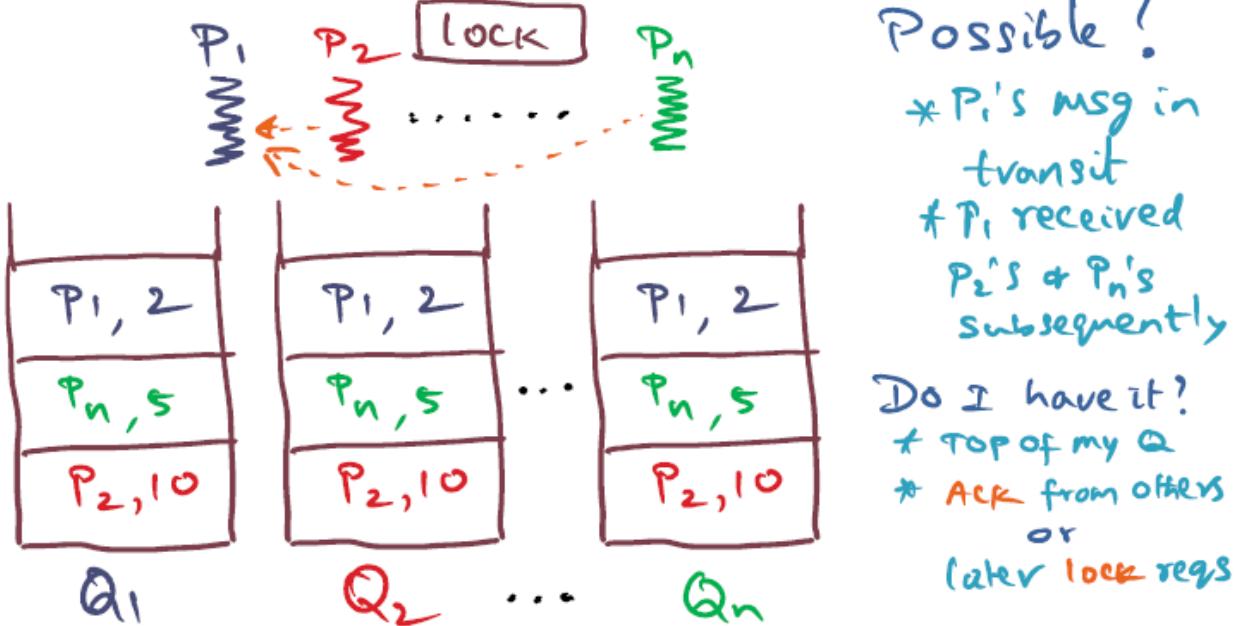
- Lamport's Logical Clock, providing us the ability to derive a total order from partial orders, allows processes to make decisions on local information (the local queue) about whether or not they hold a mutex within a distributed system.

How does a process release the mutex?

- A process will broadcast an **unlock** message to all the other processes.
- A process will remove the lock entry from its local queue.
- Peer processes receiving the **unlock** broadcast will remove the request from their local queues.

Below is a high-level representation of the concepts described above.

## Distributed M.E. lock algorithm



So how is correctness guaranteed with this algorithm? Correctness of the algorithm is based on both its construction as well as some assumptions we can make.

- Construction
  - The local queues are totally ordered and compliant with Lamport's Logical Clock algorithm. Process IDs are used to break timestamp ties between requests.
- Assumptions
  - Messages between any two processes arrive in order.
  - There is no loss of messages.

## Message complexity

Essentially, when a lock request and subsequent unlock message is sent, the message complexity is:

$$3(N - 1)$$

There is the initial lock request from a process (broadcasted), the lock request acknowledgement from the peers, and then an unlock message from the initial requesting

process (broadcasted).

Of note, there is no unlock acknowledgement being sent by the peers. This is because there is the assumption that there is no message loss within this distributed lock algorithm.

Can the message complexity of this algorithm be better?

- Absolutely. We can defer acknowledgement messages if my local request for a lock precedes the requestor's.
- This can improve the complexity to be:

$$2(N - 1)$$

Below is a high-level representation of the concepts discussed above.

Message complexity

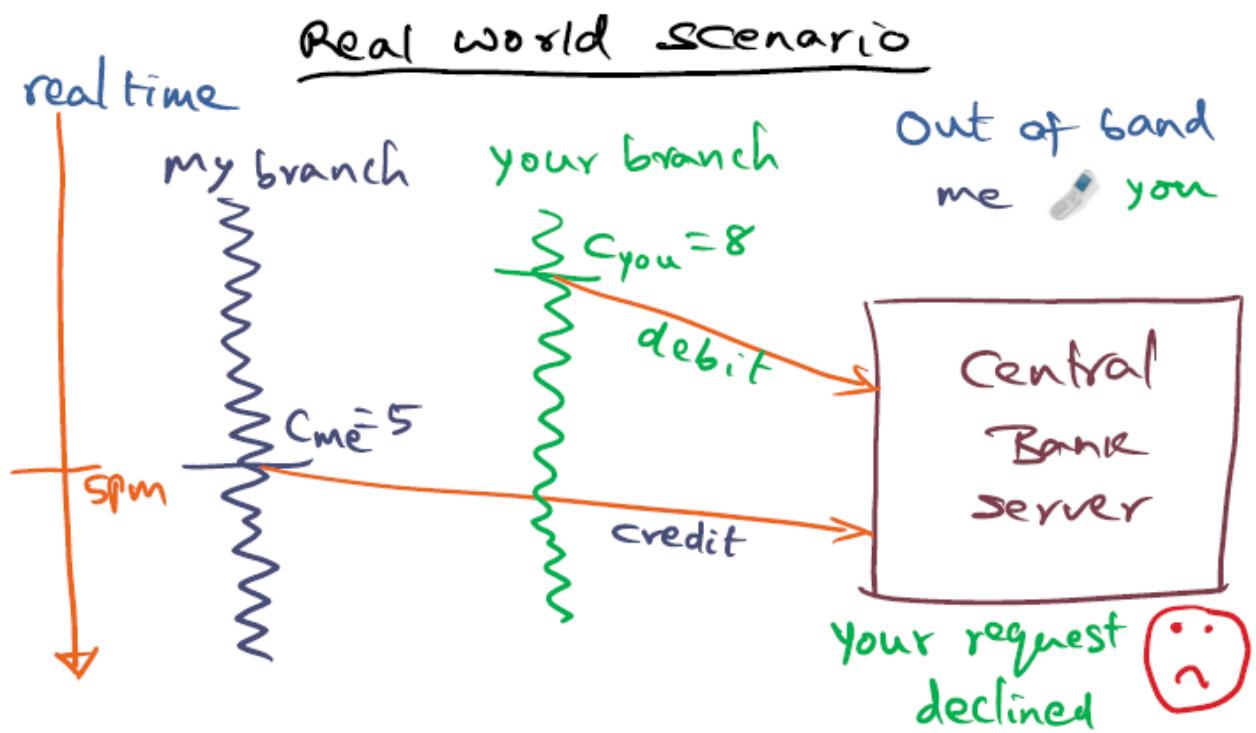
$\text{LOCK}(L) \Rightarrow$  N-1 req msgs  
⋮  
N-1 ACK msgs  
⋮  
 $\text{unlock}(L) \Rightarrow$  N-1 unlock msgs  
Total  $3(N-1)$

Can we do better?  
\* defer ACKS if my req precedes yours  
 $\Rightarrow$  combine with unlock  $\Rightarrow 2(N-1)$

## Real world scenario

In the recent discussions, we've been using Lamport's **Logical Clock** to make decisions and communicate across a distributed system. This setup has allowed each process to be tracking the same time for timestamps, however, in real-world scenarios this often isn't possible. Processes across a distributed system can have different notions of what the current time is due to **clock drift**.

Below is a high-level representation of a real-world scenario in which clock drift can cause problems.



## Lamport's physical clock

Lamport's **Physical Clock** is the same as the logical one, except with some minor differences. Again, events have to occur chronologically (event a being before event b), but now we conduct all calculations in real time. To do this, some conditions must be met:

- Physical clock conditions:
  - **PC1:** There needs to be some bound on an individual node's clock drift.
  - **PC2:** There needs to be some bound on the mutual clock drift between different nodes.

The values of the individual clock drift represented in PC1 (**k**) and the values of the mutual clock drift represented in PC2 (**epsilon**) have to be negligible compared to the inter-process communication time across the distributed system.

Below is a high-level representation of the concepts discussed above, as well as the equations for the physical clock conditions.

## Lamport's Physical clock

$a \rightarrow b$

$$\Rightarrow c_i(a) < c_j(b)$$

$$c_i(t) \xrightarrow{a} \underbrace{c_i(t)}_{\text{PC}} \xrightarrow{b} \underbrace{c_j(t)}_{\text{PC}} \leq c_j(t)$$

Physical clock conditions:

1) PC1 (bound on individual clock drift)

$$\left( \frac{dc_i(t)}{dt} - 1 \right) < k \quad \forall i; (k \ll 1)$$

2) PC2 (bound on mutual drift)

$$\forall i, j: |c_i(t) - c_j(t)| < \epsilon$$

## IPC time and clock drift

Conditions that must be met for IPC time and clock drift to avoid anomalies:

1. The difference between the time the message was received and the time the message was sent must be greater than 0.
2. The amount of clock drift should be negligible compared to the inter-process communication time.

Using these two conditions, we can derive that inter-process communication time must be greater than or equal to mutual clock drift divided by 1 minus individual clock drift. In summary, the mutual clock drift is very small in comparison to inter-process communication time.

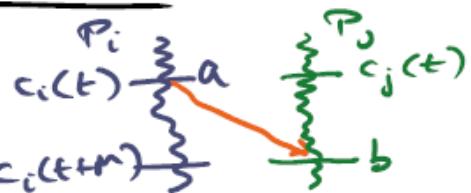
Below is a high-level representation of these conditions as well as their equations.

## IPC time and clock drift

Let  $\mu$  be lower bound on IPC

To avoid anomalies if

a on  $P_i \mapsto b$  on  $P_j$



$$1) C_i(t + \mu) - C_j(t) > 0$$

$$2) C_i(t + \mu) - C_i(t) > \mu(1 - \kappa)$$

(difference equation formulation of PC1)

using ① + ② and sound  $\Leftarrow$  on mutual drift

$$\mu \geq \epsilon / (C_1 - \kappa)$$

to avoid anomalies

## Definitions

Term	Definition
------	------------

---

clock drift	refers to several related phenomena where a clock does not run at exactly the same rate as a reference clock.
-------------	---

---

## Quizzes

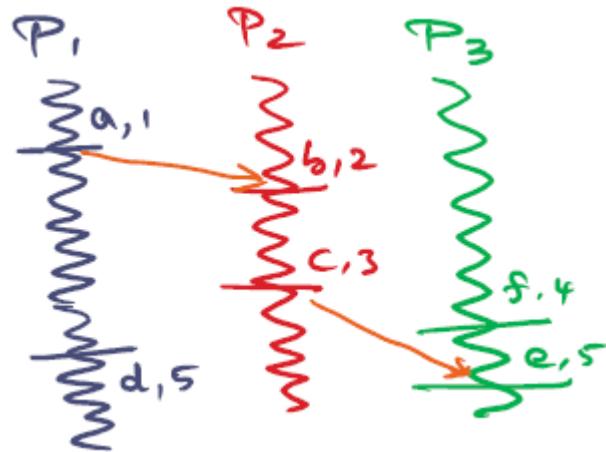
$$C(a) < C(b)$$

Choose the right answer:

- a  $\rightarrow$  b
- b  $\rightarrow$  a
- a  $\rightarrow$  b
  - if (a + b in event in the same process)

- o if (a is a send event and b is the corresponding receive)

What is the total order using Process ID to break the tie?



$(a, 1) \rightarrow (b, 2) \rightarrow (c, 3) \rightarrow (f, 4) \rightarrow (d, 5) \rightarrow (e, 5)$

Partial orders by  $\rightarrow$   
 $a \rightarrow b \rightarrow c \rightarrow e; f \rightarrow e$   
 - concurrent events ||  
 $a \parallel \{b, c, f, e\}; f \parallel \{a, d, b, c\}$

Total order respecting  
 timestamps + PID  $\Rightarrow$   
 $a \ b \ c \ f \ d \ e$

How many messages are exchanged among all the nodes for each lock acquisition followed by a lock release?

- N - 1
- 2 (N - 1)
- 3 (N - 1)

# latency limits

## Introduction

Lamport's clock laid the foundation and theoretical conditions required to achieve deterministic communications between distributed systems. In this section, we will discuss how these communication mechanisms can be implemented in operating systems efficiently enough to satisfy the conditions recently discussed.

This section discusses how efficient communication software is implemented with an application interface to the kernel, as well as what occurs inside the kernel in regard to the protocol stack.

## Latency vs. throughput

RPC performance is very crucial when it comes to building client / server systems. RPC is the main method of building distributed systems. There are two components involved in creating the latency observed during message communication within a distributed system:

- **Hardware overhead**

- Dependent upon how the network interfaces with the computer. Typically, most modern computers have a network controller that transfers messages from system memory to its network buffer using direct memory access (DMA). The network controller then places the message onto the physical link - this is where bandwidth imposed by the physical link occurs.
- In some hardware systems, the CPU may be involved in the memory transfer between physical memory and the network buffer of the network controller. The CPU will conduct I/O operations to move the data between the two devices.

- **Software overhead**

- This overhead is the overhead generated by the operating system and is additive to the hardware overhead.
- This includes the mechanisms the operating system uses to make sure the message is available in physical memory for transmission via the network controller.

Below is a high-level representation of the concepts discussed above.

## Latency vs. Throughput

Latency

elapsed time

Throughput

Events per unit time

Bandwidth: throughput measure

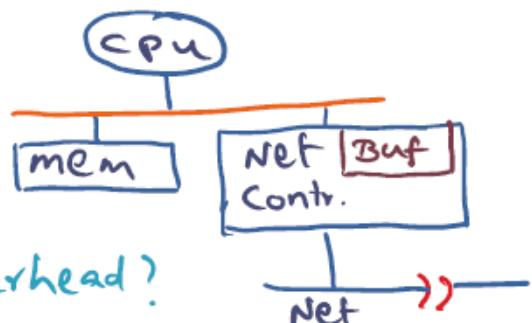
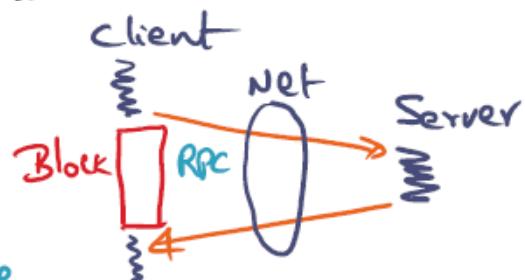
RPC performance

Hardware overhead

Software overhead

Focus of this lesson

How to reduce software overhead?



## Components of RPC latency

I won't get too in-depth of the components of an RPC call. At this point the reader should understand everything involved with this form of communication.

### 1. Client call

1. Client sets up arguments for the RPC call.
2. Client makes a call into the kernel.
3. Kernel validates the call.
4. Kernel marshals the arguments into a network packet.
5. Kernel sets up network controller to conduct network transmission.

### 2. Controller latency

1. Network controller conducts DMA to retrieve message and places data into its network buffer.
2. Network controller transmits message onto the physical link.

### 3. Time on the wire

1. Depends on the distance between the two nodes, as well as the bandwidth of the physical link.

### 4. Interrupt handling

1. Message arrives at destination node in the form of an interrupt to the operating system.

2. Network controller of the destination node moves message from its network buffer into memory using DMA.

#### 5. Server setup to execute procedure call

1. Locate the correct server procedure.
2. Dispatch the server procedure onto the CPU.
3. Unmarshal the network packet.
4. Execute the procedure call.

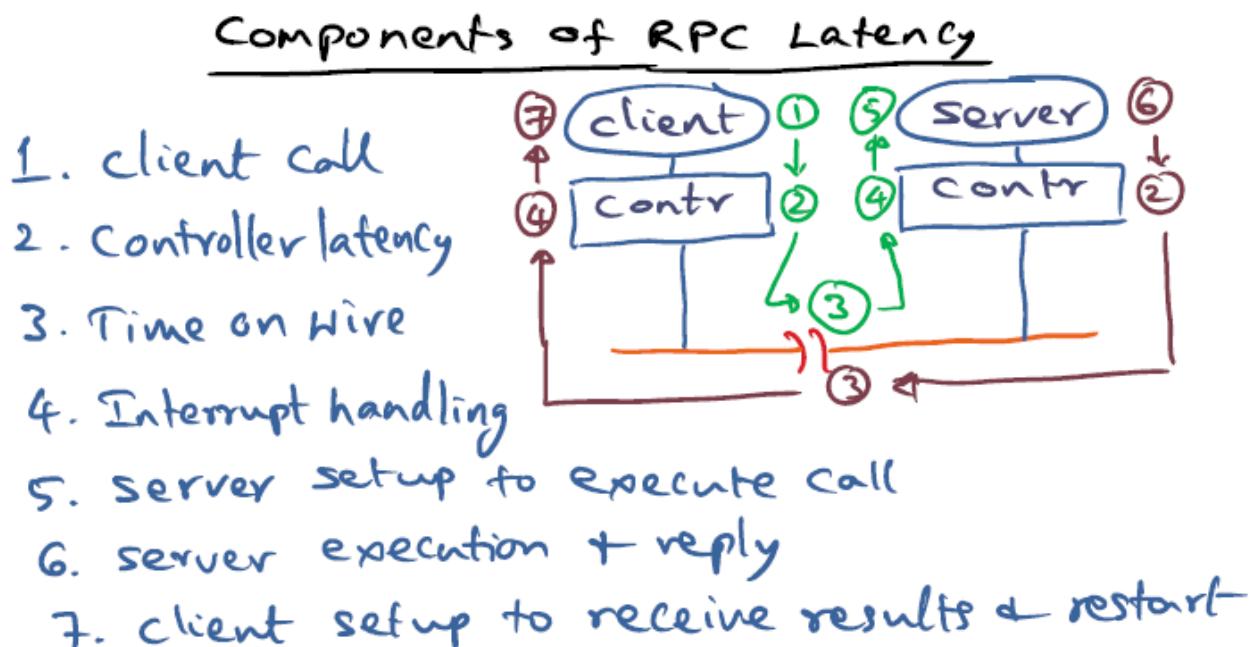
#### 6. Server execution and reply

1. Latency involved with server execution of the procedure is outside our control.
2. Reply is formed into a network packet and marshaled.
3. Item number 2 repeats.
4. Item number 3 repeats.
5. Item number 4 repeats.

#### 7. Client setup to receive results and restart execution

1. Client is dispatched onto the CPU to receive results.

Below is a high-level representation of the steps discussed above.



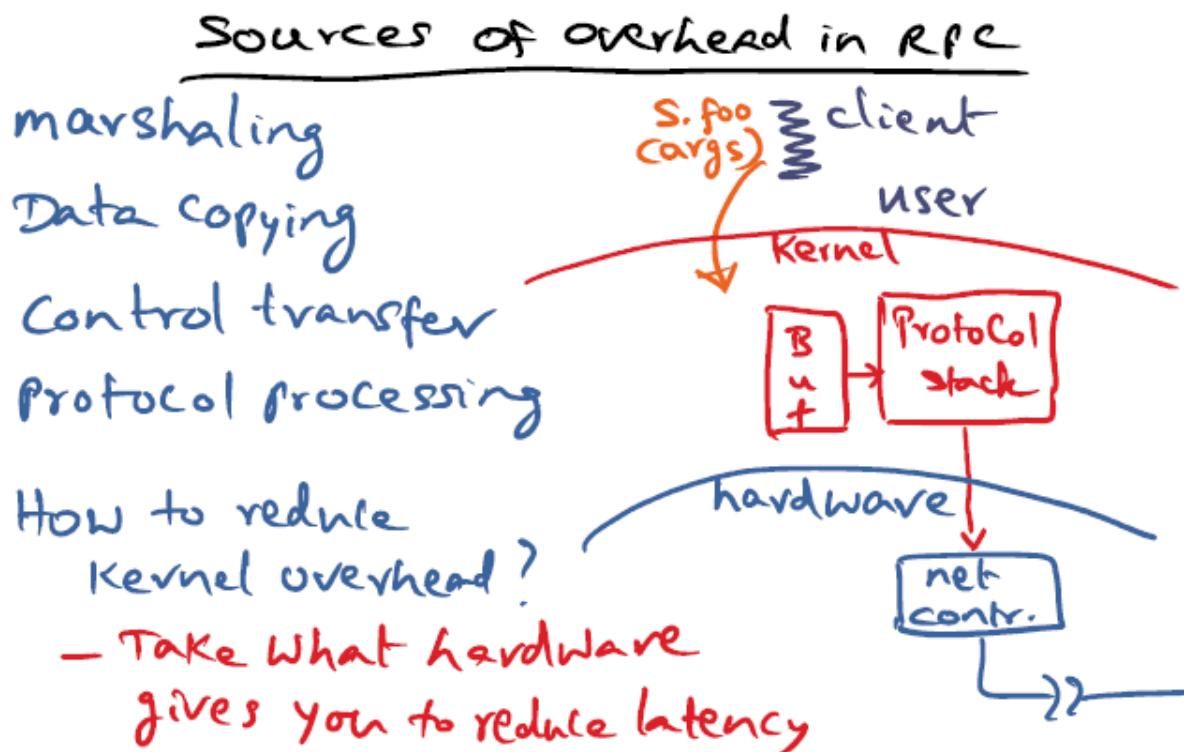
### Sources of overhead in RPC

The sources of overhead that we can control and mitigate when designing the RPC process are:

- Marshaling
- Data copying
- Control transfer
- Protocol processing

How do we reduce the kernel overhead involved in RPC? We need to take what the hardware provides us and leverage its capabilities in order to reduce overhead.

Below is a high-level representation of the overheads involved in RPC.



## Marshaling and data copying

The biggest source of overhead in marshaling is the data copying overhead. Up to three copies of the message data could be involved when marshaling data for an RPC. Where are these copy operations being generated?

- The client stub makes the first copy of the message data from the client process stack into an RPC message.
- Copying of the RPC message from client address space into kernel address space (kernel buffer).

- DMA from kernel buffer to network controller buffer.

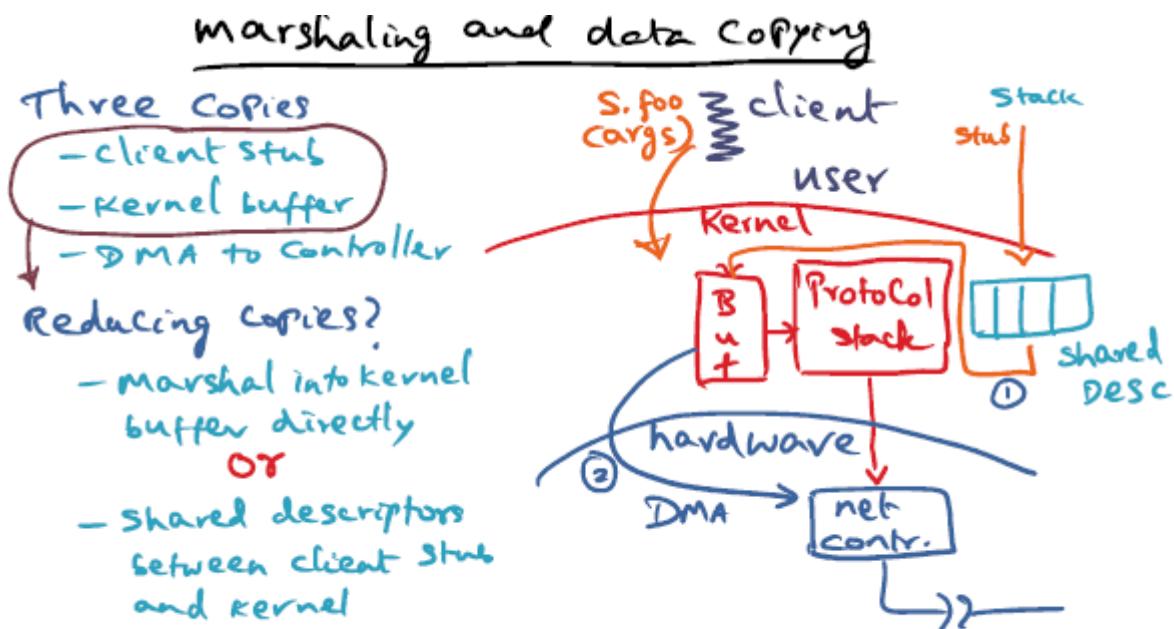
Copying overhead is the biggest source of overhead for RPC latency. So how do we reduce the number of copy operations?

- The DMA of the message data from the kernel buffer into the network controller buffer is unavoidable.
- What we **can** do is marshal the arguments of the client's RPC directly into the kernel buffer - avoiding copying arguments from the stack into a user-space buffer and then having to copy them, again, into kernel space. The procedure for the client stub that the client will call into will reside within kernel space.

The problem with placing stack data from user-space directly into the kernel is that this operation is dangerous. We cannot guarantee the data being placed into the kernel buffer by the client is correctly sanitized.

The second option to remove the number of copy operations to conduct an RPC is to utilize shared descriptors between the client stub and the kernel. The client stub will remain in user-space and will use the shared descriptors to describe to the kernel the data structures that will reside within the kernel buffer.

Below is a high-level representation of the shared descriptors mechanism.



## Control transfer

The second source of overhead in a RPC is the control transfer overhead. There are potentially four control transfers conducted for an RPC.

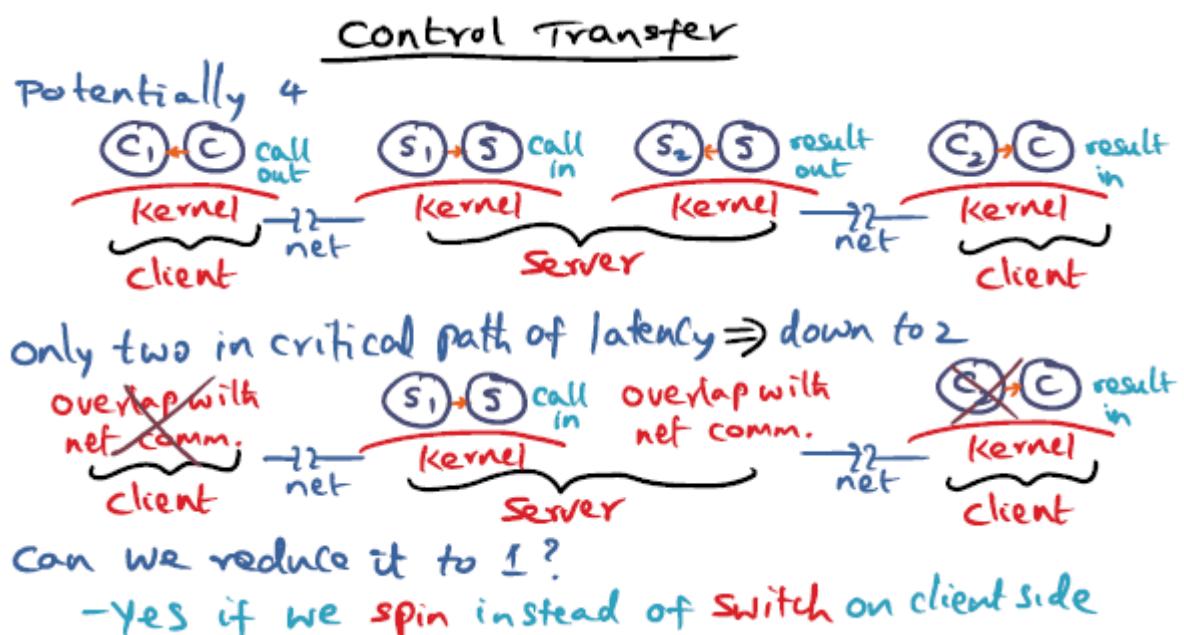
1. Client calls into the kernel to conduct an RPC.
2. Call is received by server machine as an interrupt - server procedure executes.
3. Server calls into the kernel to send the RPC reply.
4. Reply is received by client machine as an interrupt - client executes to receive reply.

Steps 2 and 4 are in the critical path of RPC latency. Having to dispatch the server or client procedures to receive RPC messages causes latency.

We can reduce the the number of context switches down to two by overlapping the context switches with the network communication being conducted for the RPC.

Can we reduce the number of context switches to one? Absolutely. If we can determine that the RPC and the server procedure are relatively fast and respond quickly, the client can spin instead of block on the CPU allowing the client procedure to send and receive the reply quickly. This only works if the server procedure is fast, however. If we block for too long, we'll waste CPU time and resources.

Below is a high-level representation of the concepts described above.



## Protocol processing

Another portion of RPC that creates latency is protocol processing. Here we must decide how we want to deliver the message to the other node. So what transport method are we going to use for a particular RPC?

- If we're in a Local Area Network (LAN), we know our connection is pretty reliable. This indicates that we should probably focus on reducing latency. One axiom is that the concepts of reliability and latency will always be at odds with one another. So in choosing our transport method, we must always look to make the most sensible compromise. So how do we reduce latency in a LAN?
  - Don't conduct low level acknowledgements of messages. Chances are our message reached its intended destination.
  - Utilize hardware checksums for packet integrity.
  - Since the client blocks, we don't need to buffer the message on the client side. If the message gets corrupted or lost, we'll just resend the call.
  - We do need buffering on the server side, however, for the reply. That way, if the client resends the request because the reply was lost, we can easily resend the reply - this avoids re-executing the server procedure.

Below is a high-level representation of the concepts discussed above.

### Protocol Processing

what transport for RPC?  
LAN reliable  $\Rightarrow$  reduce latency

choices for reducing latency in transport  
No low level ACKS  
Hardware checksum for packet integrity  
No client side buffering since client blocked  
overlap server side buffering with  
result transmission

## Definitions

Term	Definition
latency	elapsed time
throughput	events per unit time
bandwidth	measure of throughput
RPC message	contiguous buffer of data that represents arguments or a reply for an RPC

## Quizzes

**It takes 1 minutes to walk from the library to the classroom. The hallway is wide enough for 5 people.**

**1. What is the latency incurred by traveling from the library to the classroom?**

- 1 minute.

**2. What is the throughput achieved if 5 people walk from the library to the classroom?**

- 5 people per minute.

# active networks

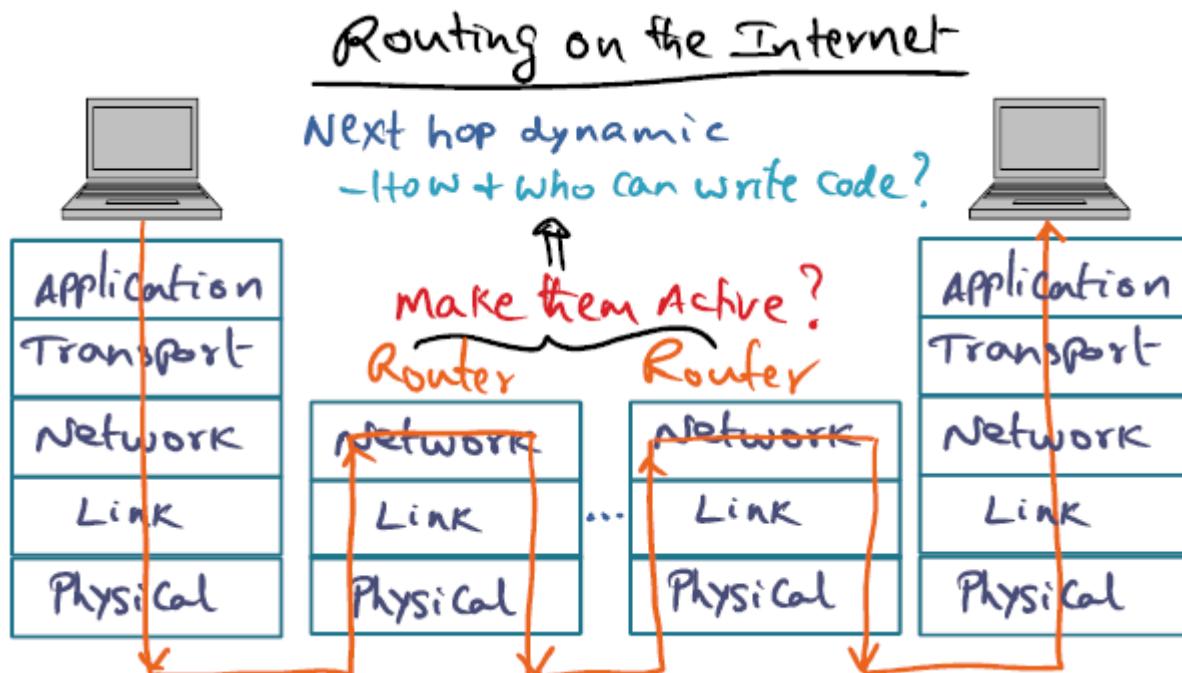
## Routing on the internet

At this point the reader should understand the OSI model and TCP/IP stack. I only took notes by exception, commenting on things that were novel presented within this slide.

What does it mean to make a router **active**? A router becomes **active** when, instead of utilizing its routing table to conduct the routing of a packet, it actually executes code in order to dynamically decide the next hop of the currently inspected packet.

This mechanism is implemented by passing packets to a router than contain code for the router to execute, that way the router conducts a custom set of instructions for that packet. This provides us the ability to create customized flows for network services, and every network can have its own way of choosing the route from source to destination.

Below is a high-level representation of the concepts described above.

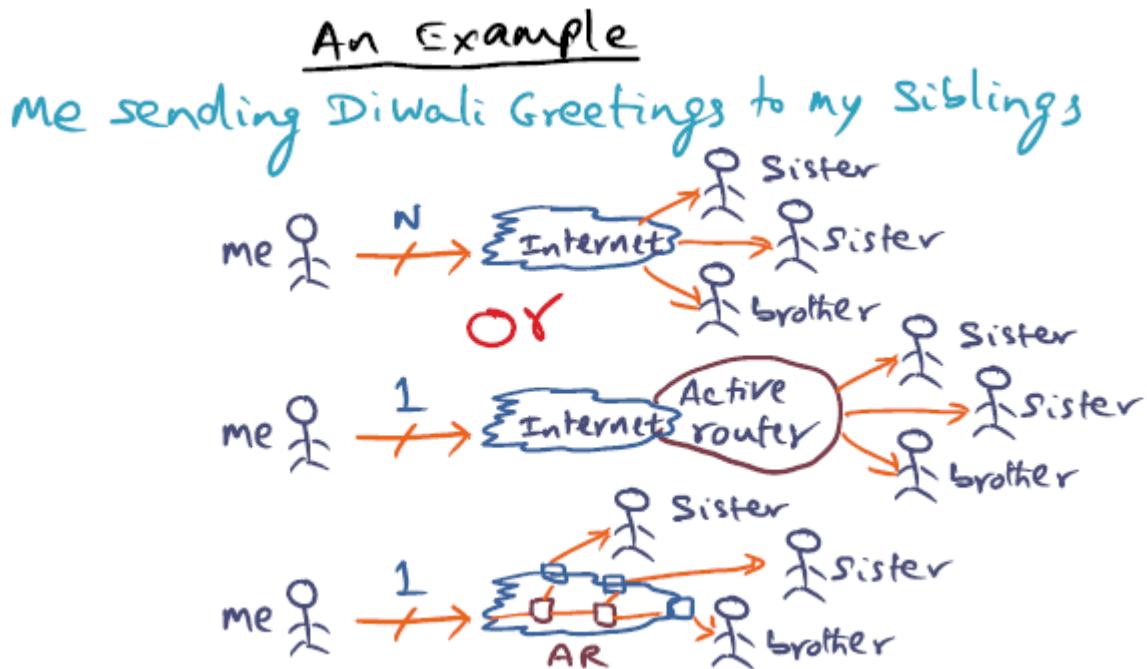


## The vision behind active networks

Essentially we utilize this extra code placed into a packet for the router to analyze so that we can leverage more powerful abstractions and requests for the routing of a packet. We

can conduct broadcasts to multiple destinations and recipients without having to specifically send a packet to each recipient.

An example of the concept discussed above is shown below.



## Implementation of active networks

The operating system provides quality of service (QoS) APIs to the application developer for use in hinting to the operating system how specific network interactions should be treated. This allows the application developer to specify network flows that might have real-time constraints, etc.

The operating synthesizes these QoS hints and generates executable code that it then places in the packet. In other words, the protocol stack of the operating system has to support these QoS improvements and expose an API to support customized network flows.

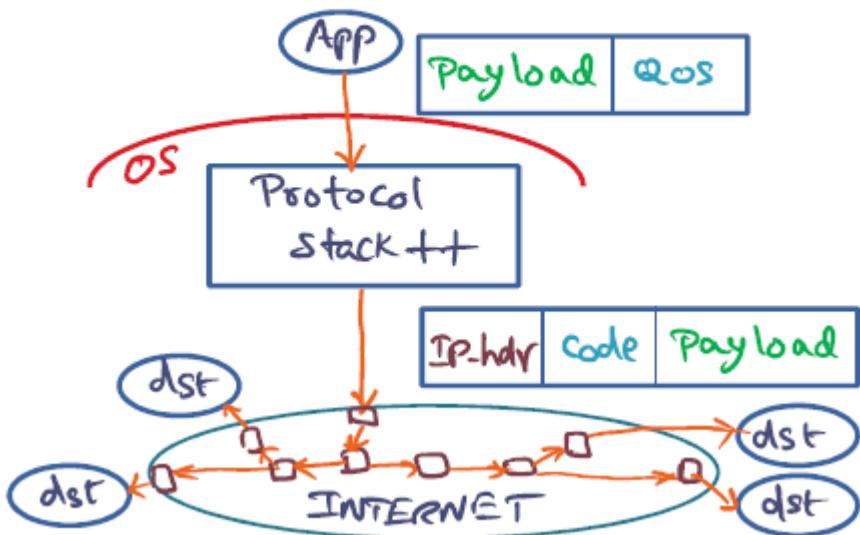
The synthesized code placed into the packet by the operating system allows routers in between the source and destination nodes to make more informed routing decisions, allowing developers to create software-defined network flows.

The primary challenge is that the network stack of an operating system has to be augmented to support the above idea, and the routers expected to conduct this software-

defined routing also have to support this new concept. We cannot expect all nodes and all routers to have these features available.

Below is a high-level representation of the concepts discussed above.

## How to implement the vision?



## ANTS toolkit

The ANTS toolkit provides a method of avoiding having to augment the operating system and its protocol stack in order to support software-defined routing and QoS code synthesizing.

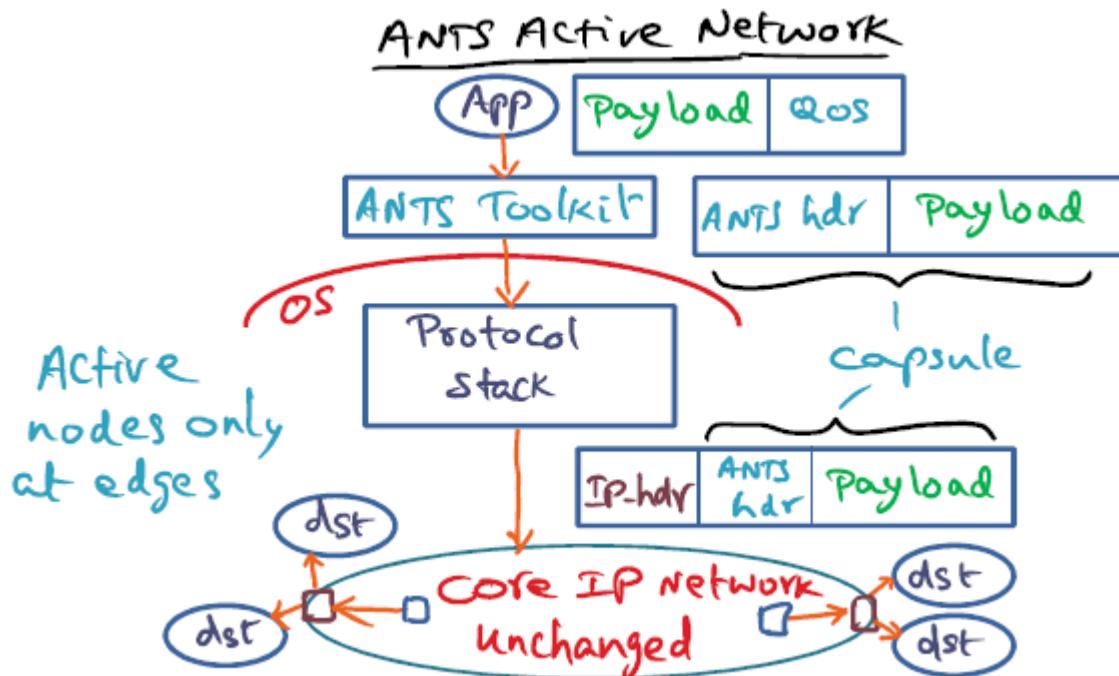
The ANTS toolkit provides an API to the programmer to hint QoS requirements, and the ANTS toolkit will encapsulate the original packet payload with an ANTS header. The operating system will place an IP header onto the now encapsulated payload and transmit the packet onto the network.

The IP packet with an ANTS encapsulated payload will route normally across a network, unless the router that receives the packet is an active node. An active node will be able to process the ANTS header and make intelligent routing decisions based upon the contents of the header.

This encapsulation mechanism removes the pain point of having to modify the operating system to support software-defined routing. However, what about the routers? Not all of them will be active.

To defeat this limitation, we ensure that active nodes reside at the edge of the network. This way, we can leave the core IP network unchanged and all of the intelligent routing decisions will be made at the edge.

Below is a high-level representation of the concepts described above.



## ANTS capsule and API

The header of an ANTS packet consists of three parts:

- The original IP header
  - Important for routing a packet to its destination if the node does not support active routing.
- The payload generated by the application
- The ANTS header with two fields:
  - The **type** field
    - The **type** field is a way by which you can identify the code that has to be executed to process this encapsulated ANTS packet.

- The **type** field is an MD5 hash of the code that needs to be executed on this capsule.
- The **prev** field
  - The identity of the upstream node that successfully processed the encapsulated ANTS packet of this type.
  - Useful information to identify the code that needs to be executed in order to process this ANTS encapsulated packet.

Of note, the ANTS capsule within the IP packet does not contain the code that needs to be executed to process the ANTS capsule. The ANTS capsule only contains a type field used as a vehicle to identify the code that needs to be executed to process the capsule.

As for the API, the API allows us to define the routing of the packet for intelligent forwarding through the network. Thus, we can virtualize the network flow regardless of the actual physical topology.

The second part of the API allows us to leverage soft storage. Soft storage is storage that's available in every router node for personalizing the network flow with respect to a particular type of capsule. Soft storage is where we store the code for a particular type of capsule so that we can conduct intelligent routing decisions and code execution.

The API provides the programmer the ability to put, get, and remove objects from soft storage, and soft storage's data structures use key, value pairs. Storing code associated with the particular type of a capsule is important for personalizing the network flow of said capsules - nodes that hold the personalized code within their soft store can re-use the code for the respective type of capsule.

The soft store can also be used to store things like computed hints about the state of the network. These hints can be used for future capsule processing for capsules of the same type.

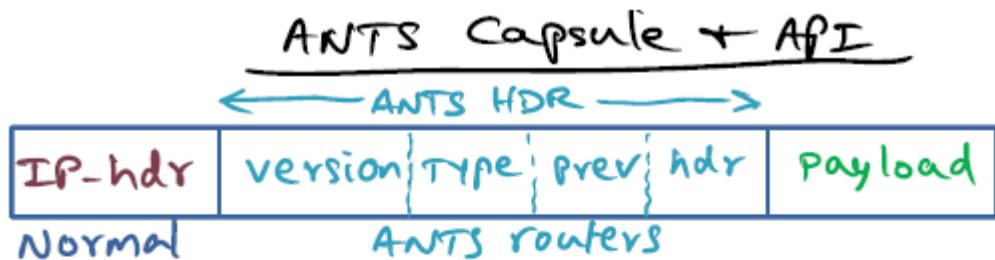
The final portion of the API allows the programmer to acquire metadata about the state of the network.

The final takeaway from this section is that the ANTS API is very small and for good reason. We are attempting to execute code on routers residing on the public internet. We

need to be considerate of other network interactions that are occurring across these nodes and maintain important characteristics about our interactions with them:

- Programs that we execute on router nodes must be easy to program.
- Programs that we execute on router nodes must be easy to debug, maintain, and understand.
- Programs that we execute on router nodes should execute very quickly.

Below is a high-level representation of the concepts discussed above.



### ANTS API

Method	Description
<code>int getAddress()</code>	Get local node address
<code>ChannelObject getChannel()</code>	Get receive channel
<code>Extension findExtension(String ext)</code>	Locate extended services
<code>long time()</code>	Get local time
<code>Object put(Object key, Object val, int age)</code>	Put object in soft-store
<code>Object get(Object key)</code>	Get object from soft-store
<code>Object remove(Object key)</code>	Remove object from soft-store
<code>void routeForNode(Capsule c, int n)</code>	Send capsule towards node
<code>void deliverToApp(Capsule c, int a)</code>	Deliver capsule to local application
<code>void log(String msg)</code>	Log a debugging message

## Capsule implementation

What are the actions taken on capsule arrival at an active node?

- Each capsule contains a **type** field that is a cryptographic fingerprint of the original capsule code.
- When a node receives a capsule, two things are possible:
  - The node has seen capsules of this type before, retrieves the code for this capsule from its soft store, and executes the code - forwarding the capsule towards its desired destination.
  - This is the first time this node has seen a capsule of this type. The node references the **prev** node and requests the code for this type of capsule from the **prev** node.

- After receiving the code from the **prev** node, the node executes the code and stores the code into its local soft store.

The first time a particular capsule enters the network, no node will have the correct code for it. However, as network flows are completed, nodes will all have the capsule's code stored in soft storage - allowing us to leverage locality in the processing of network flows for a capsule type.

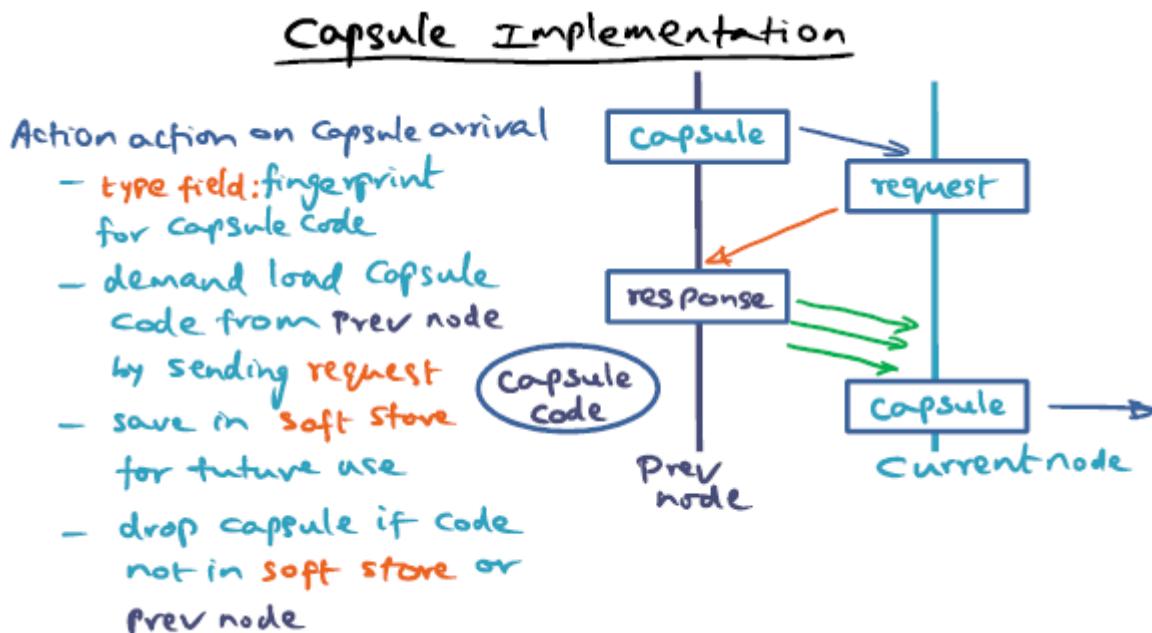
How does a node ensure it got the correct code from the **prev** node?

- By using the cryptographic hash contained in the **type** field of a capsule, a node can hash the received code from the **prev** node and compare the hashes.

What if I request the code from the **prev** node and the **prev** node does not have the code in its soft storage?

- The node will drop the capsule.
- Higher level acknowledgements will detect that packets / capsules have been dropped in the network flow.
- The source of the network flow will re-transmit the capsule until the issue is resolved.

Please find below a high-level representation of the concepts discussed above.



## Potential applications

Active networks can be used for a variety of applications. Here are some examples:

- Protocol independent multicasts
- Reliable multicasts
- Congestion notifications
- Private IP (PIP)
- Anycasting

Active networks are useful for building applications that are difficult to deploy in the internet. We overlay our own desires for network flow on top of the physical topology of the network. Some things to keep in mind for application development leveraging active networks:

- Applications should be expressible
- Applications should be compact
- Applications should be fast
- Applications should not rely on all the nodes being active

## Pros and cons of active networks

Active networks provide these pros:

- Flexibility from the perspective of an application developer
- The ability to ignore physical network layout and implement virtual network flows

Active networks provide these cons:

- Protection threat, code is executing on shared nodes used by multiple customers
  - ANTS implements some safeguards to combat this:
    - ANTS is written in Java so all code executing on a router node is sandboxed
- Protection threat, code spoofing
  - Utilization of strong cryptographic hashes allows router nodes to inspect checksums
- Integrity of the soft storage, a particular network flow could consume all of the soft storage of a node

- ANTS API is restrictive and designed to emplace limitations on this kind of abuse
- Resource management threat, flooding the network with packets
  - ANTS API is restrictive and designed to prevent this. The internet is already susceptible to flooding, however, so nothing really changes with the implementation of ANTS

## Feasible?

Router vendors hate opening up the network for application developers to extend or make changes. It is only feasible to implement virtual network flows using active routers at the edge of the network.

Software routing requires computation and is slower than hardware routing. This also restricts software routing to only occur at the edge of the network.

And finally, there are social reasons why this is hard to adopt. The user community is not comfortable having arbitrary code executing on public routing fabric where their traffic exists. While there may be protection in place to prevent abuse, there can never be a guarantee that malicious actors won't attempt to utilize software network flows for attacks.

## Quizzes

**In your opinion, what can be roadblocks to the Active Networks Vision?**

- Need buy-in from router vendors
- ANTS software routing cannot match throughput needed in Internet Core
- Makes the Internet more vulnerable
- Susceptible to code spoofing

Clearly, if we want to do anything on a router, we need buy-in from router manufacturers. The second point is, routers are dumb animals for a reason. There is so much internet traffic that ANTS software routing would not be able to match the speed of the Internet Core.

# systems from components

## The big picture

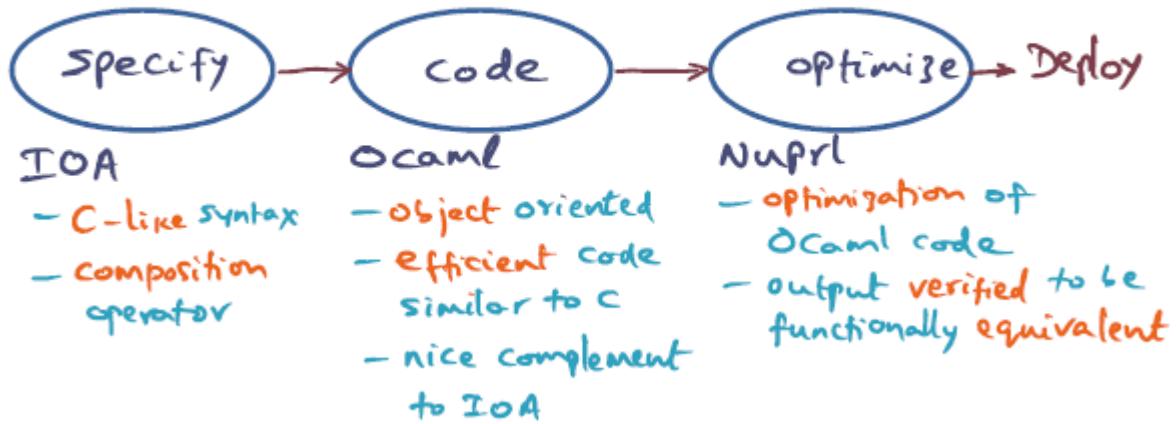
We put theory and practice together using the **design cycle**. Here, we'll explain every step in the design cycle as we work to create a system component.

- Specify
  - Using IO Automator, we develop the system requirements and specify them using the syntax of the language. (C-like syntax)
  - The **composition operator** in IO Automator allows for the expression of the specification of an entire subsystem that we wish to build.
- Code
  - Now we convert the specification expressed by IO Automator into code that can actually be executed.
  - We use **OCaml** (object oriented categorical abstract machine language).
  - OCaml is a great candidate for component based design due to its object-oriented nature and the fact that it's a functional programming language.
  - The code generated by OCaml is as efficient as C code, which is very important to operating system developers.
- Optimize
  - **NuPrl** is a theoretical framework for optimization of OCaml code. The input is OCaml code and the output is optimized but functionally equivalent OCaml code.
  - NuPrl uses a theorem-proving framework in order to conduct optimization and verifies through theorem-proving that the resulting code generated is equivalent to the input code.

Below is a high-level representation of the big-picture discussed above.

## Ensemble - Big Picture

### Design Cycle



### From specification to implementation

We start with an **Abstract Behavioral Specification** using IO Automator. We present the high-level logical specification of the components - the properties that we want the subsystem to adhere to.

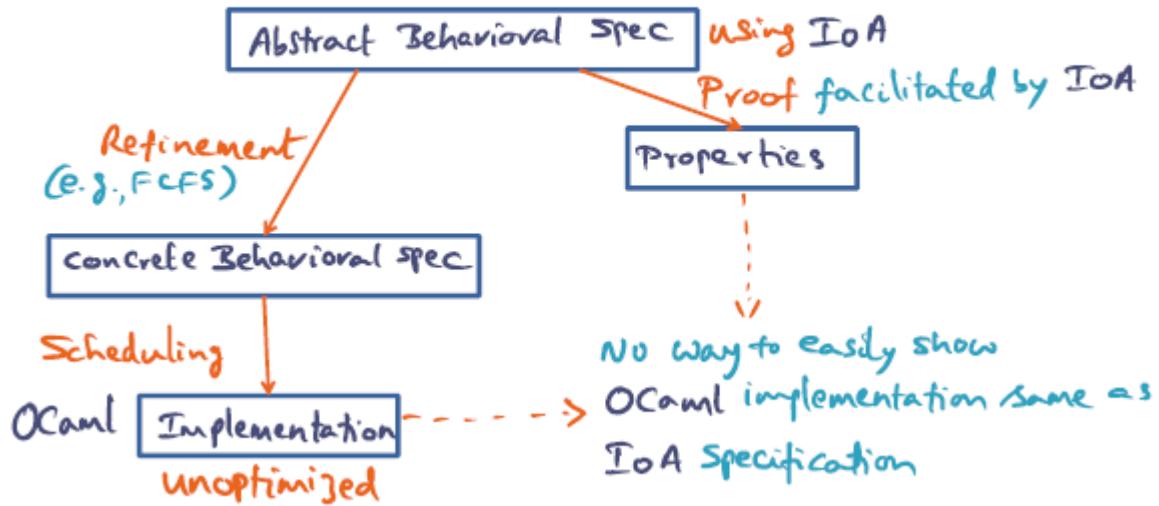
Next we create the **Concrete Behavioral Specification**, not implementation just yet, but refinement of the abstract specification created earlier.

Finally, we arrive at implementation using OCaml, generating executable code that achieves the behavior outlined in the original abstract behavioral specification. Keep in mind, this generated executable code is unrefined and not the most optimal implementation - yet.

One word of caution, however, there is no guarantee that the implementation is actually meeting the abstract behavioral specification. There's no easy way to show that the implementation is the same as the abstract behavioral specification.

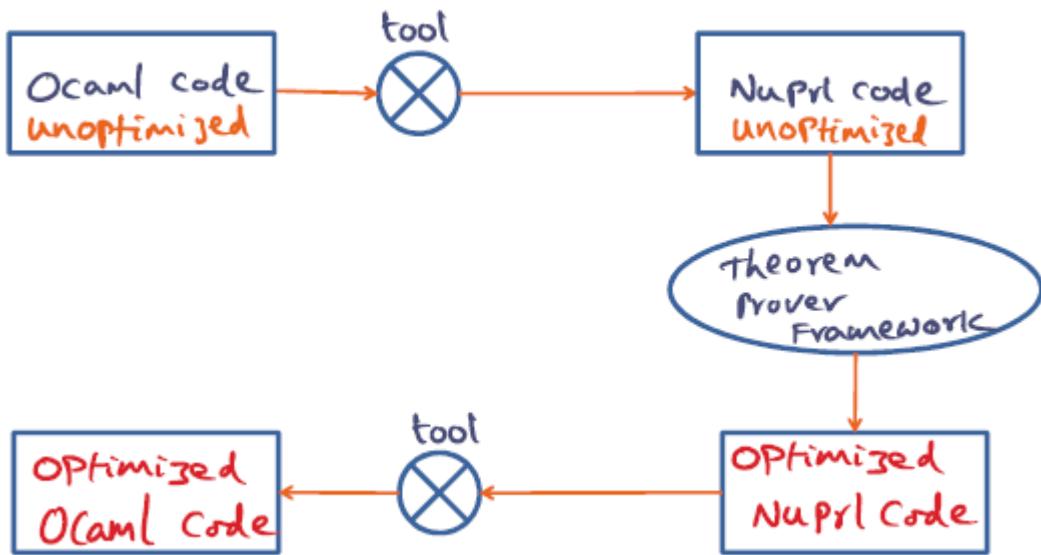
Below is a high-level representation of the process discussed above.

## Digging Deeper – From Spec to Implementation



Below is a high-level representation of how unrefined OCaml code is optimized by NuPrl.

## Digging Deeper – From Implementation to Optimization



## Putting the methodology to work

Below is a high-level representation of how we conduct software engineering to mimic VLSI design when attempting to design an operating system component.

## Putting this methodology to work

Start with ION Spec

— Abstract spec

— Concrete Spec

How to synthesize the stack from concrete spec?

Getting to an unoptimized OCaml implementation

— Ensemble Suite of microprotocols



\* flow control, Sliding window, encryption, scatter/gather, etc.

\* well defined interfaces allowing composition  
\* facilitates component based design

Recall original goal: mimic VLSI design

## How to optimize the protocol stack?

Layering of components could actually lead to inefficiencies. Unlike VLSI, software components utilize interfaces for layering - there will be well-defined boundaries between components. This requires the copying of arguments between components, leading to inefficiency.

Several sources of optimization are possible:

- Explicit memory management instead of implicit garbage collection
- Avoid marshaling / unmarshaling across layers of components
- Buffering in parallel with transmission
- Header compression
- Locality needs to be enhanced for common code sequences

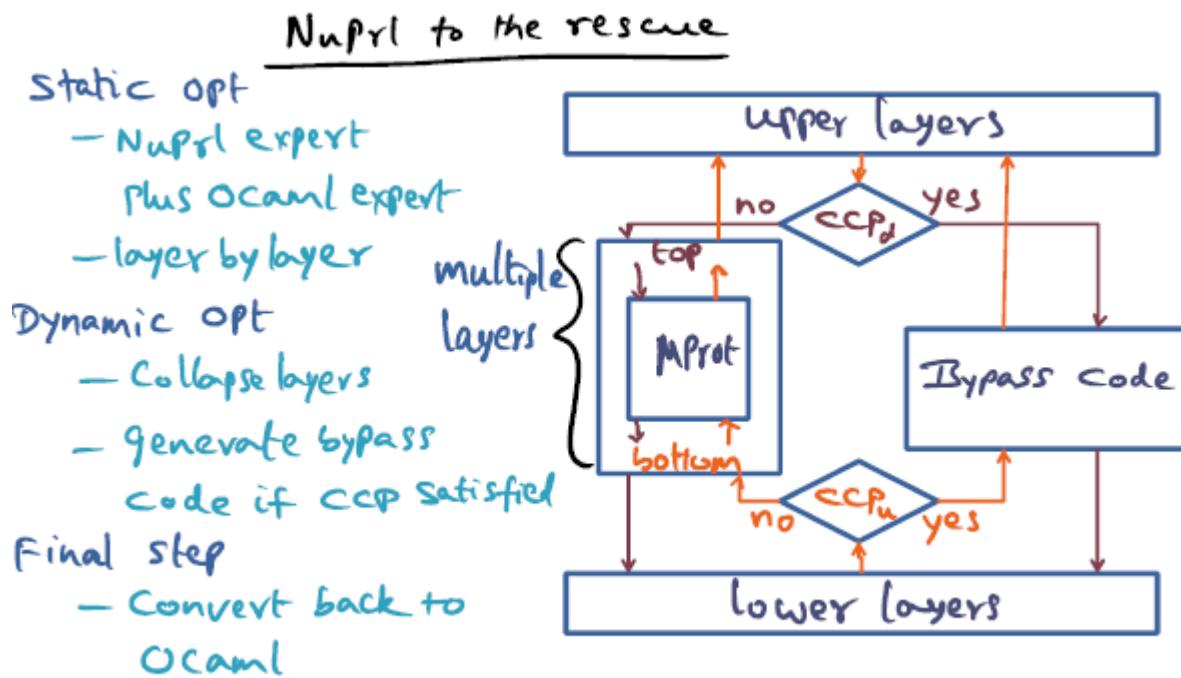
## NuPrl to the rescue

NuPrl can conduct static optimization for OCaml code, however, this is unhelpful as we have multiple layers that make up our components. We need to collapse common events that happen.

NuPrl can also conduct dynamic optimization, collapsing layers and generating bypasses if the code's common case predicate is satisfied. Common case predicates are synthesized from the decision statements within the micro-protocols - allowing us the ability to bypass and optimize the code.

Finally, we convert all of this back to optimized OCaml code, and the theorem-proving framework can prove that the unoptimized code and the optimized code are functionally the same.

Below is a high-level representation of the concepts discussed above.



lesson6

# lesson6

# spring operating system

## How to innovate operating systems

There is always a conundrum on how to innovate operating systems. Should we create a brand new operating system? Should we create a better implementation of a known operating system?

Research and industry is usually restrained by the marketplace it serves. Sun Microsystems in its heyday was making Unix Workstations, large complex server systems, etc. The marketplace demands were influenced Sun's research - most of the applications running on business machines were legacy and it would cost much more for those businesses to move to an entirely brand new operating system.

For Sun to innovate, they had to make changes where it made sense - without releasing an entirely brand new product. This is where the Spring operating system came to fruition. The attempted to preserve the external interface and APIs that application developers could program against, while at the same time making improvements on internal components and then integrating them correctly.

Object oriented approaches to this problem are a good choice in ensuring that we can innovate the internal components of a piece of software, while maintaining the external interface exposed to the customer.

## Object based vs. procedural design

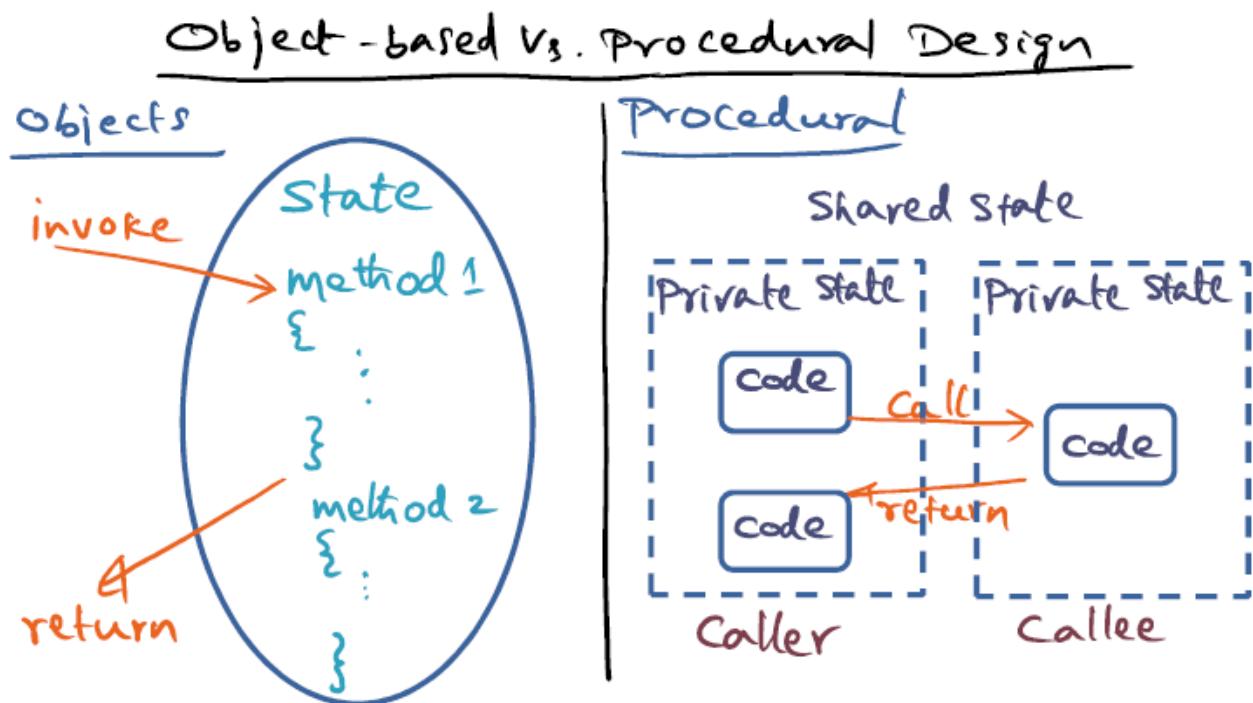
Monolithic kernels usually leverage a procedural design in their implementation. The code is one monolithic entity with shared state and global variables, and maybe some private state between procedure caller and callee. Subsystems make procedure calls to each other, and this is how monolithic kernels are constructed.

In contrast, objects contain the state and the state is not visible outside of the object. Methods are exposed to manipulate the state as well as retrieve information about the state of the object. With object based design, you achieve the implementation of strong interfaces and isolation of the state of an object from everything else.

So if we have strong interfaces and isolation, isn't this similar to what we discussed in the structuring of operating systems? Is this similar to the implementation of protection domains, and will we incur performance costs having to conduct border crossings?

There are ways to make the protection domain crossings performance conscious. The Spring operating system, for example, applies object orientation in building the operating system kernel; the key takeaway being if object oriented programming is good enough to implement a high-performance kernel, it's good enough to implement higher levels of software, as well.

Below is a high-level representation of the concepts discussed above.



## Spring approach

The Spring approach to operating system design is to adopt strong interfaces for each subsystem. The only thing exposed for a subsystem is what services are provided. The internal components of a subsystem can be changed entirely, however, the external interface will always remain unchanged.

The Spring approach also wanted to ensure that the system was open and flexible, allowing for the integration of third party software into the operating system.

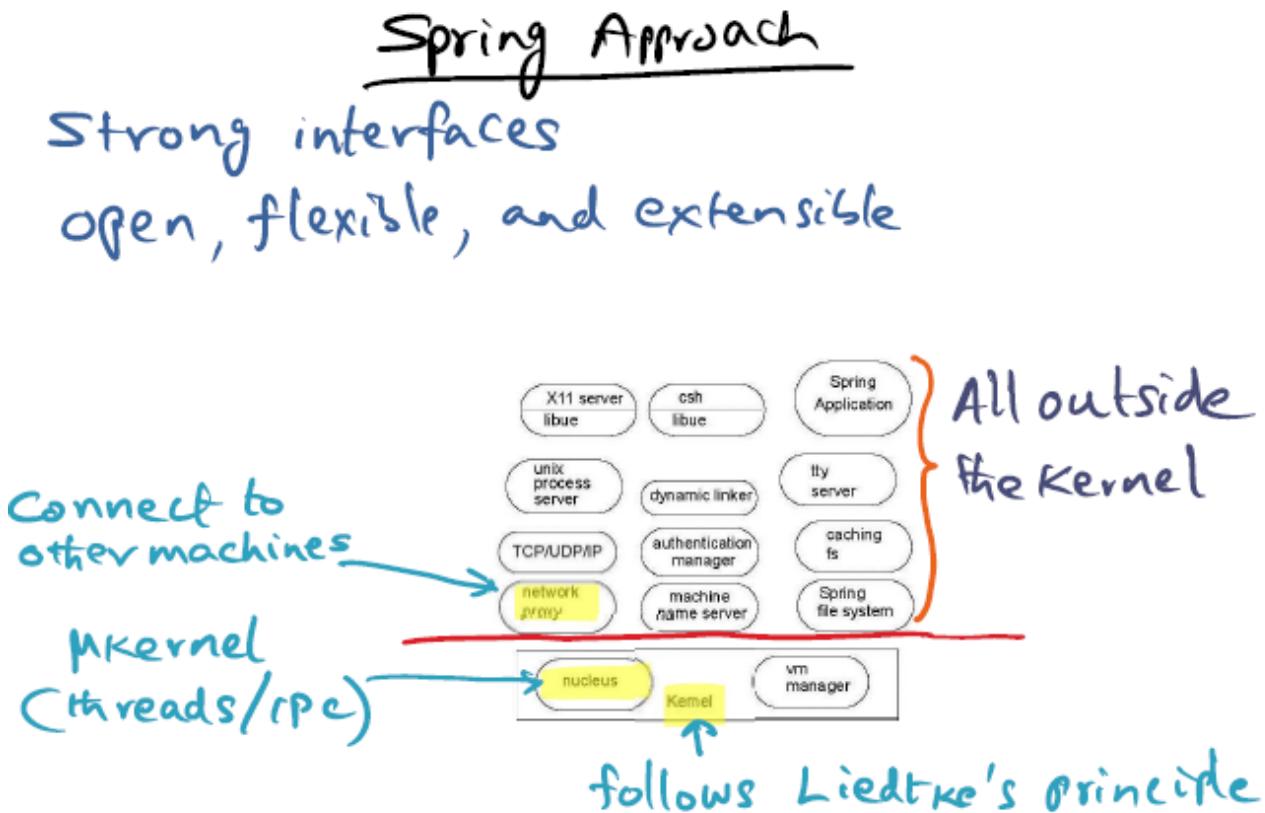
To be open and flexible, we must also accept that software can be implemented in different languages. This is why the Spring approach adopted the interface definition language (IDL). Third party software vendors can use the interface definitions in building their own subsystems that integrate with the Spring operating system.

The Spring system utilizes the micro-kernel approach to extensibility. The Spring micro-kernel includes:

- the nucleus which provides abstractions for threads and inter-process communication
- the virtual memory manager

All of the other system services are outside of the kernel. Spring is Sun's answer to building a network operating system using the same interface (the UNIX interface).

Below is a high-level representation of the Spring operating system structure.



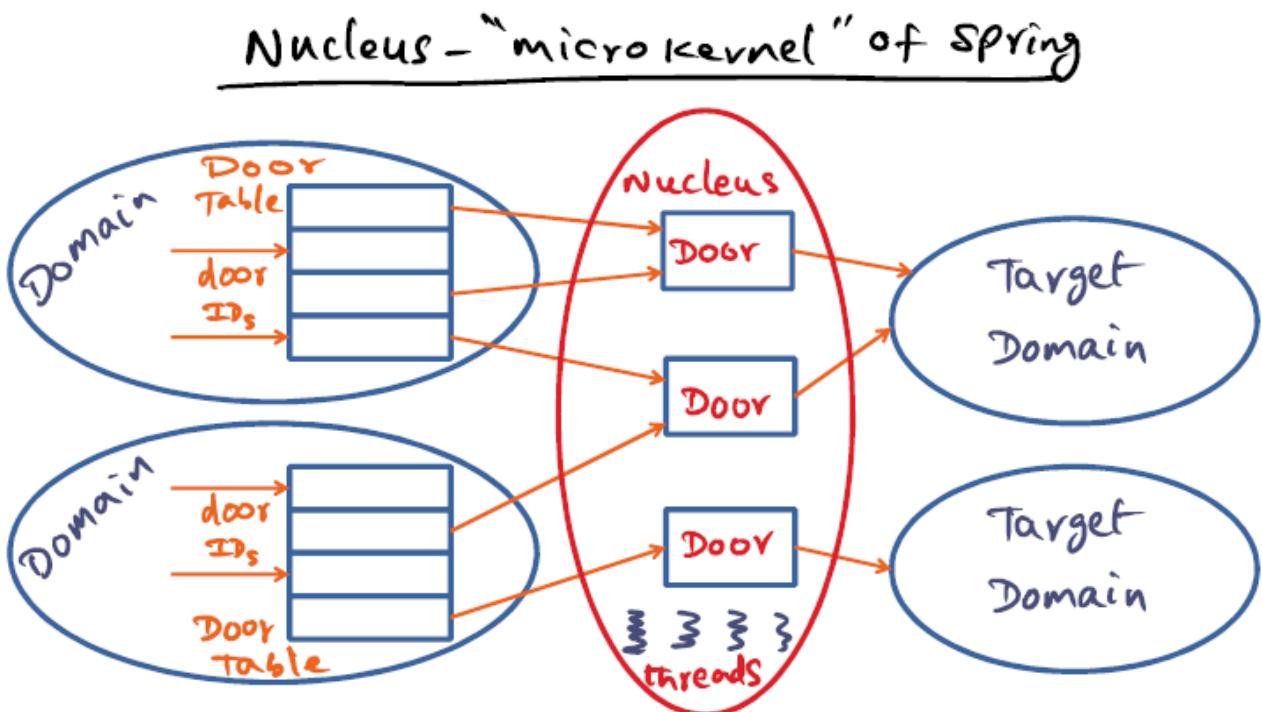
## Nucleus micro-kernel of Spring

The abstractions available in the nucleus of spring are as follows:

Abstraction	Description
domain	similar to a UNIX process; threads execute within domains
door	software capability to a domain; entrypoints for target domains; pointer to C++ object
door handle	essentially the same as a file descriptor, but for doors to domains
door table	listing of door handles

The nucleus is involved in every door call. When the invocation occurs, the nucleus inspects the door handle, allocates a server thread in the target domain and executes the invocation. Its a protected procedure call - the client thread is deactivated and activated within the client domain. The thread is deactivated again and reactivated within the client domain's address space. This is very similar to LRPC - conducting very fast cross-domain calls using this door mechanism. These mechanisms leverage the good attributes of object oriented programming while avoiding performance hits.

Below is a high-level representation of the concepts discussed above.



## Object invocation across the network

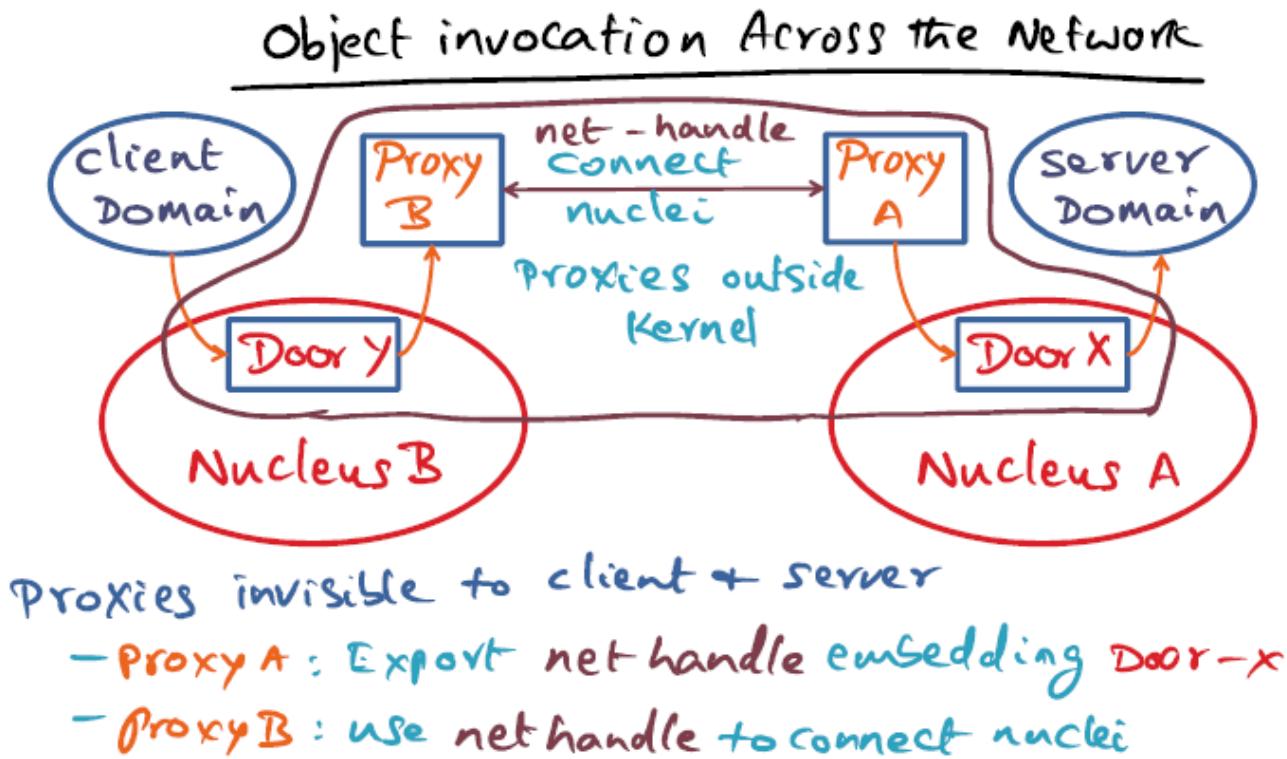
Spring is a network operating system, so how do we accomplish object invocation across the network? Well object invocation is between client and server domains is extended using network proxies.

Network proxies can potentially employ different network protocols when communicating with domains hosted on different nodes. This allows the network interaction between a client domain and a server domain to be completely transparent, the client and server only have to worry about utilizing the correct doors.

So what are the steps in implementing this mechanism?

- Proxies export a **net handle** embedding a particular door for a server domain.
- Proxies wishing to utilize the server domain will use the **net handle** to connect nuclei with the peer Spring operating system.

Below is a high-level representation of network object invocation.

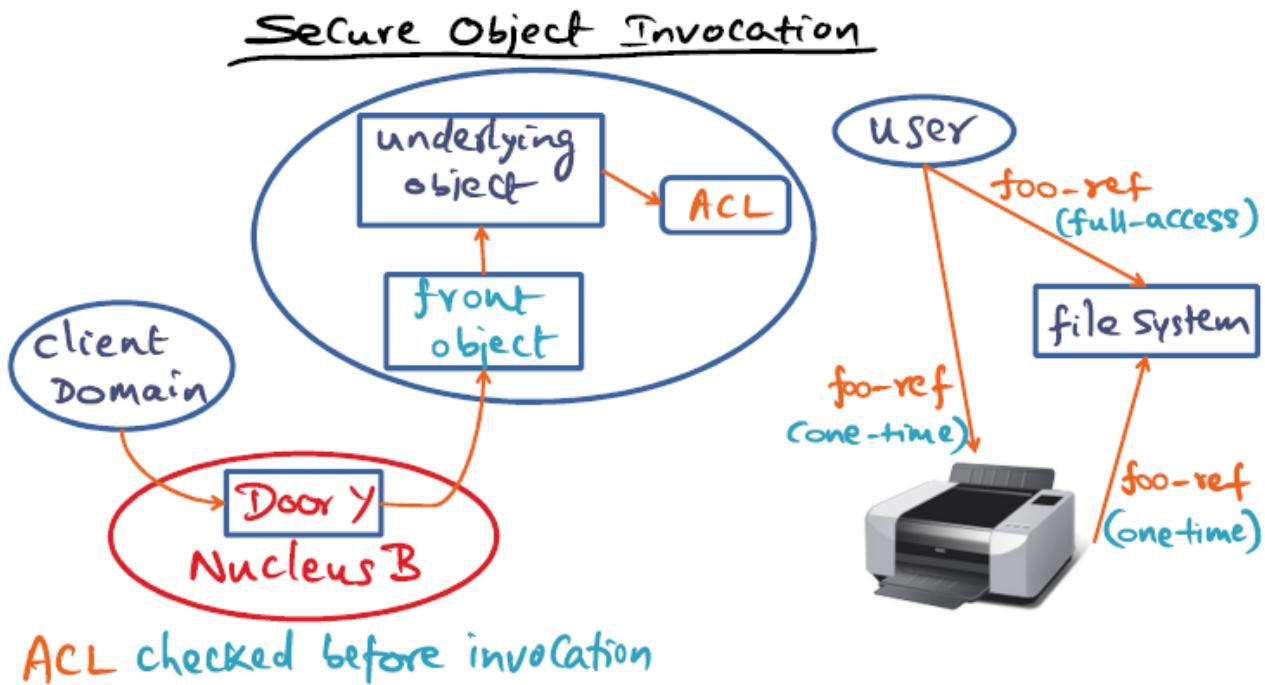


## Secure object invocation

Spring also allows for policies to be implemented for objects, allowing objects to have different access privileges for different user. Spring uses **front objects** outside the **underlying object** - this is what the client directly interfaces with when using the **door**. The **front object** checks the **access control list** (ACL) to see if a user has the permissions to use the object.

Client domains have the ability to transfer **doors** to other domains for use. The privileges of the **door** do not have to be transitive, however. The client domain can decide if it wants to provide the same privileges for other domains using the door, or it can provide lesser privileges.

Below is a high-level representation of the concepts discussed above.



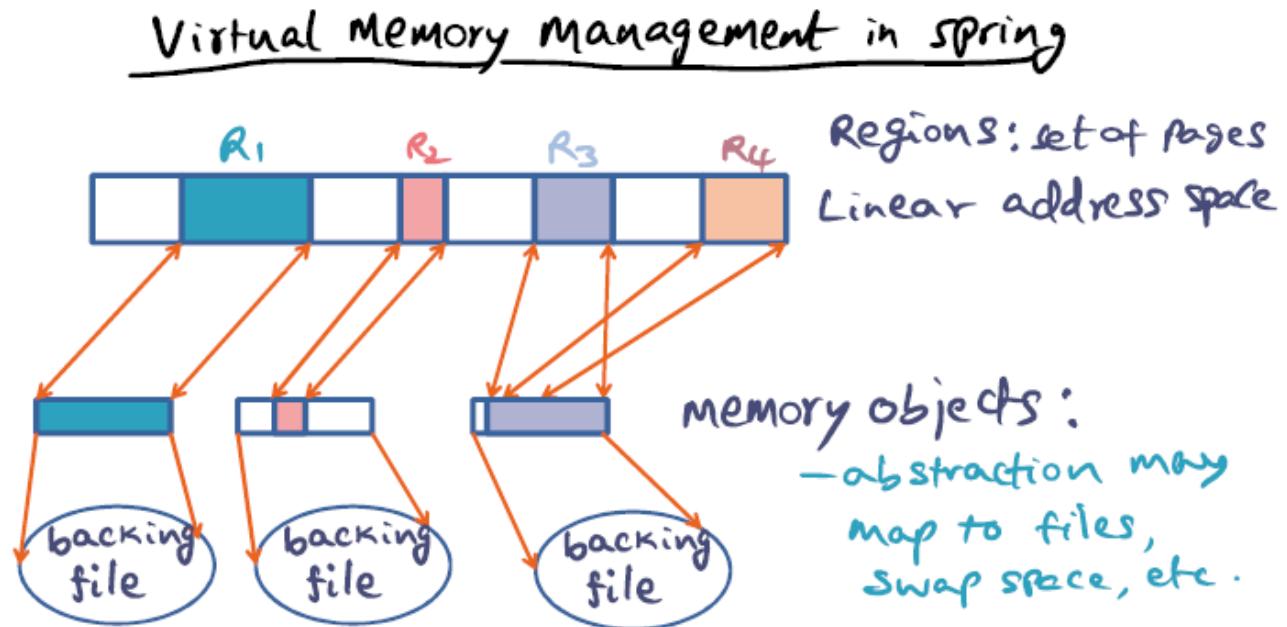
## Virtual memory management in Spring

The virtual memory manager in Spring is in charge of managing the linear address space of every process. The virtual memory manager breaks this linear address space into **regions**. Regions are a set of **pages**.

The virtual memory manager abstracts things further, creating what are called **memory objects**. **Regions** are mapped to different **memory objects**. The abstraction of **memory**

**objects** allows a region of virtual memory to be associated with backing files, swap space, etc. It is also possible that multiple memory objects can map to the same backing file, or whatever entity is being referenced by a memory object.

Below is a high-level representation of the concepts discussed above.

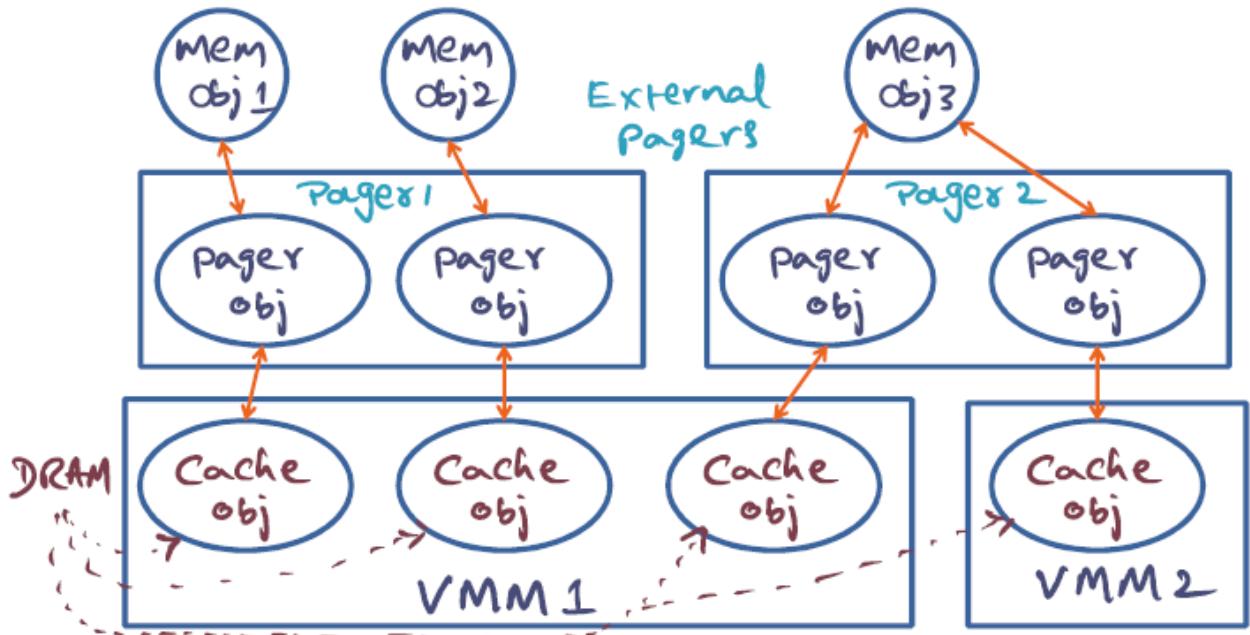


### Memory object specific paging

The virtual memory manager needs to place the memory object into RAM for a process to be able to access the memory it contains. The virtual memory manager leverages **pager objects** that will make and establish connections between virtual memory and physical memory. The **pager object** will create a **cached object** representation in RAM that will represent a portion of the memory object in RAM for processes to address.

Below is a high-level representation displaying two processes with their linear address space being managed by two different virtual memory managers. The virtual memory managers utilize **pager objects** to represent a portion of **memory objects** in the linear address space using **cache objects**.

## Memory object specific Paging



So what happens when two different address spaces are referencing the same memory object using two different paging objects? What about the coherence of the different representations?

- It is the responsibility of the two pager objects to coordinate cached object coherence if needed.

## Spring system summary

Below is a listing of all the primitives used by Spring to construct an object-oriented network operating system.

## Spring System Summary

object oriented kernel

- nucleus → Threads + IPC
- microkernel → nucleus + Address space
- Door + Doortable → basis for cross domain calls
- object invocation and cross machine calls
- virtual memory management
  - \* Address space object, memory object, External pages, Cached obj

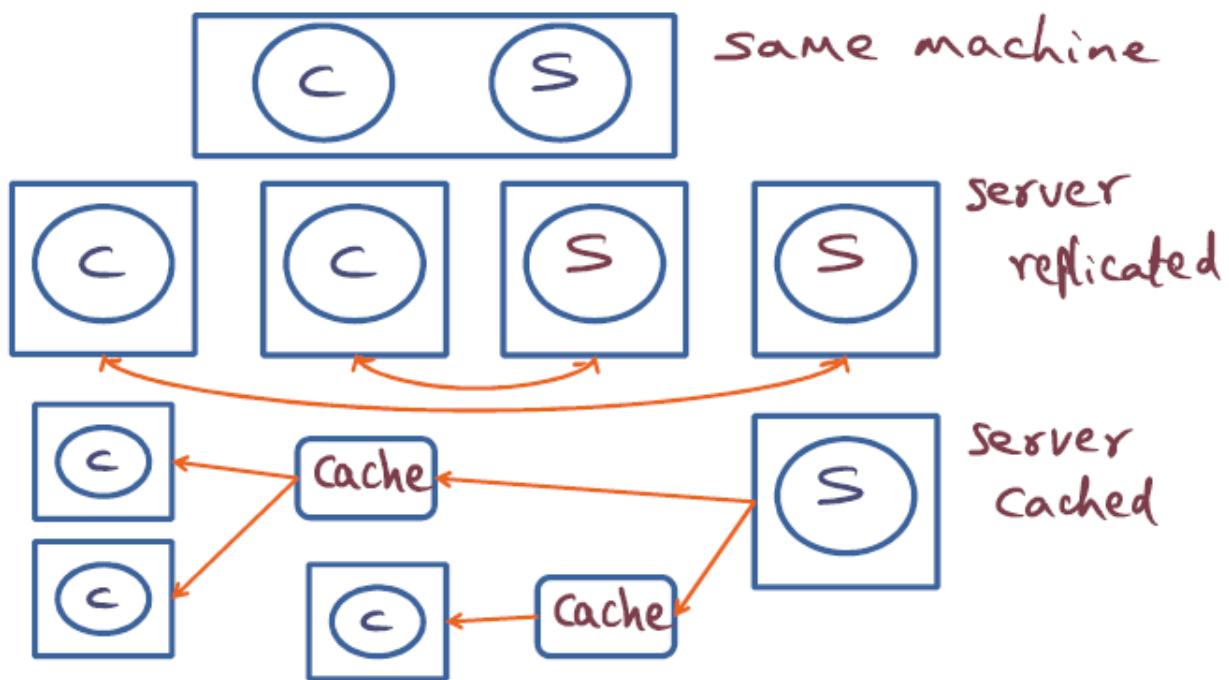
The contrast between Tornado and Spring is that Tornado uses clustered objects as a method to optimize implementing kernel services. In the Spring system, object technology permeates the entire design, being used as system structuring - not just for optimization.

## Dynamic client server relationship

Client / server interactions should be agnostic to where they are physically located - whether it be on the same machine or across a network. Additionally, the Spring system routers client / server interactions to duplicate servers if the workload is large enough. And there's also the ability to create a cached copy of the server that the client can interface with.

Below is a high-level representation of the concepts discussed above.

## Dynamic client server relationship



## Subcontracts

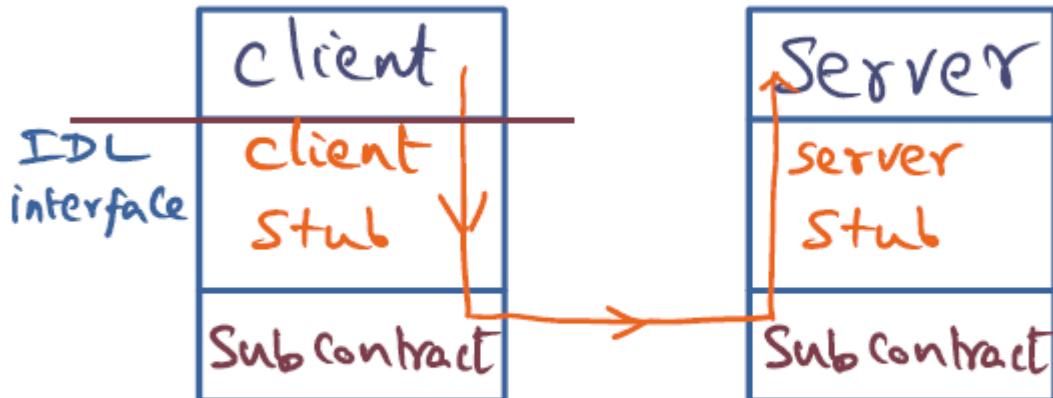
So how are these dynamic relationships between clients and servers created? They use a mechanism called the **subcontract**. You could use the analogy of off-loading work to some third party, where that third party conducts work on behalf of the original vendor.

The **subcontract** is the interface provided for realizing the IDL interface between a client and a server. **Subcontracts** hide the runtime implementation of a server that a client can call.

This simplifies client side stub generation, everything is detailed in the **subcontract**. The **subcontract** can be changed at any time - each one can be discovered and installed at run time. All they do is advertise the IDL they implement, no more details need to be provided to the client stub.

Below is a high-level representation of this concept.

## Subcontract

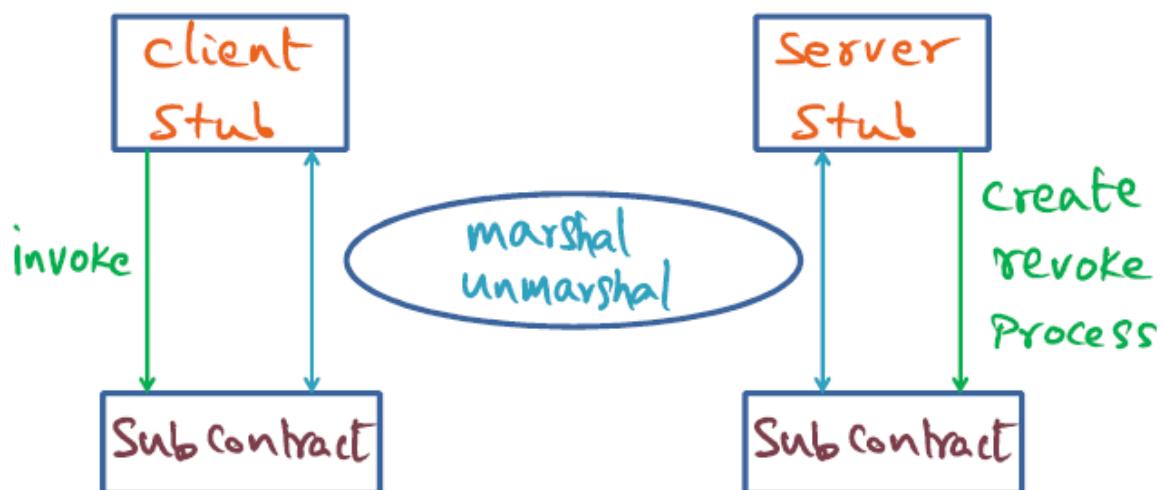


- client side stub generation simplified
- subcontract responsible for details

## Subcontract interface for stubs

Below is a high-level representation for how the client and server stub interface with the subcontract. This includes the marshaling and unmarshaling of data, invoking the subcontract, etc.

## Subcontract interface for stubs



## Quizzes

Identify the similarities and differences in abstractions provided by the Nucleus and Liedtke's micro-kernel.

### Question

This question concerns abstractions in Nucleus vis à vis Liedtke's microkernel

Feature	Nucleus	Liedtke
Threads	✓	✓
IPC	✓	✓
Address Space	✗	✓

# java rmi

## Java distributed object model

The great thing about Java's distributed object model is that all of the things the programmer would have to worry about, marshaling / unmarshaling, etc. is handled by the Java distributed object model. The object model provides these abstractions:

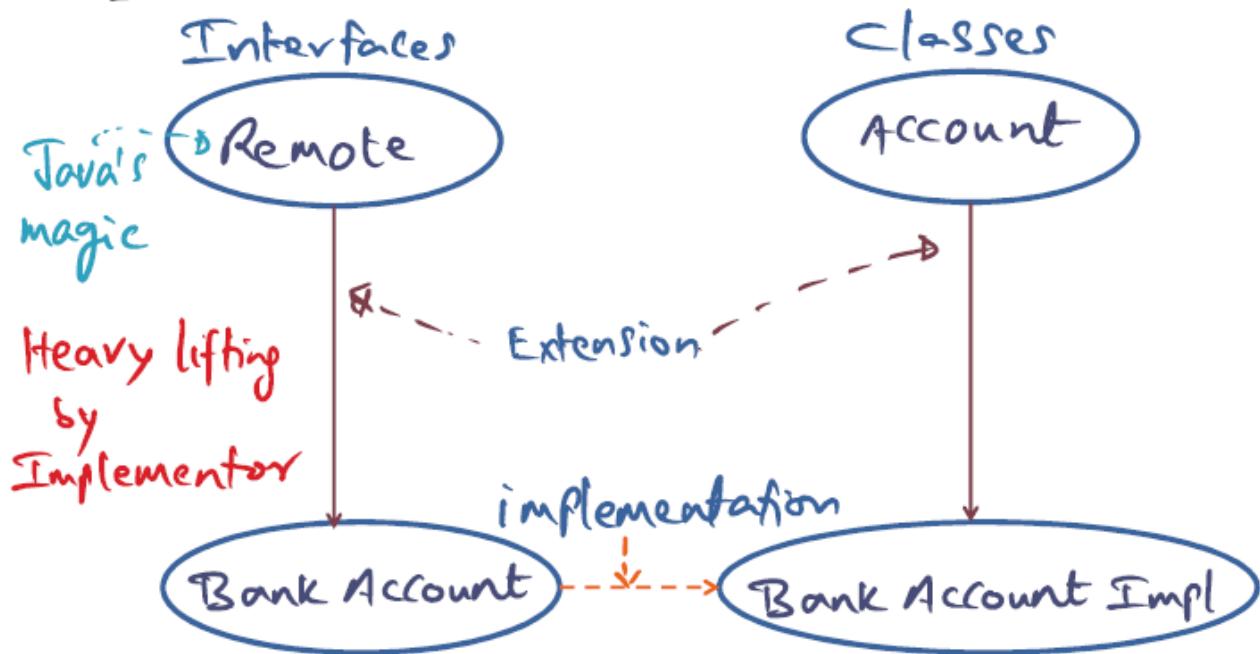
Abstraction	Definition
remote object	accessible from different address spaces
remote interface	declarations for methods in a remote object
failure semantics	clients deal with RMI exceptions
similarities / differences to local objects	<ul style="list-style-type: none"><li>• object references can be parameters</li><li>• parameters only as a value / result</li></ul>

## Reuse of local implementation

With Java it's possible to re-use local implementation of objects and expose them to the network by pairing them with remote interfaces. This is done by providing an extension of the local version of the class. Clients will interface with the public version of the class, however, all interactions will essentially be forwarded to the local implementation of the object.

Below is a high-level representation of the concepts discussed above.

## First choice: Reuse of local implementation



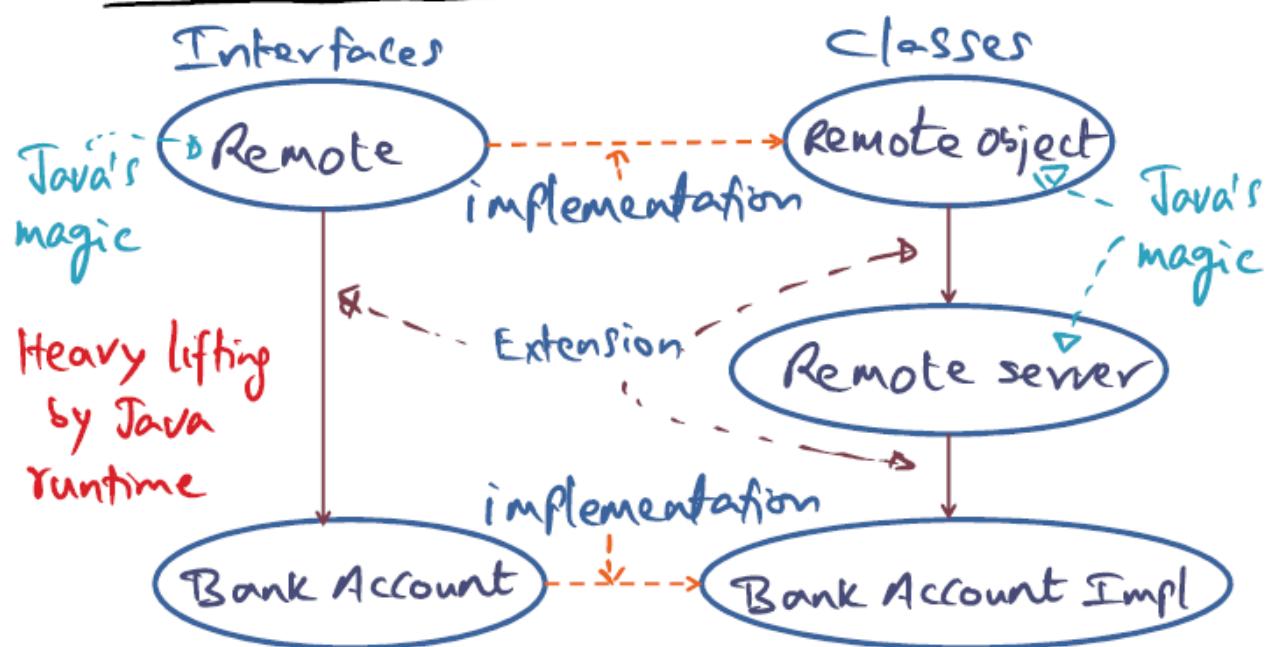
## Reuse of remote

The second choice is to re-use the remote object class. Again, the developer creates the bank account class and exposes the methods that are in the object using the remote interface. Now clients can all interface with the remote version of the bank account.

The bank account class this time, however, is derived from the remote object and remote server classes as an extension. Now, when bank account implementation objects are instantiated, they immediately become visible on the network for clients to access. This requires less setup and work required from the developer.

Below is a high-level representation of the concepts discussed above.

## Second choice: Rense of "Remote"



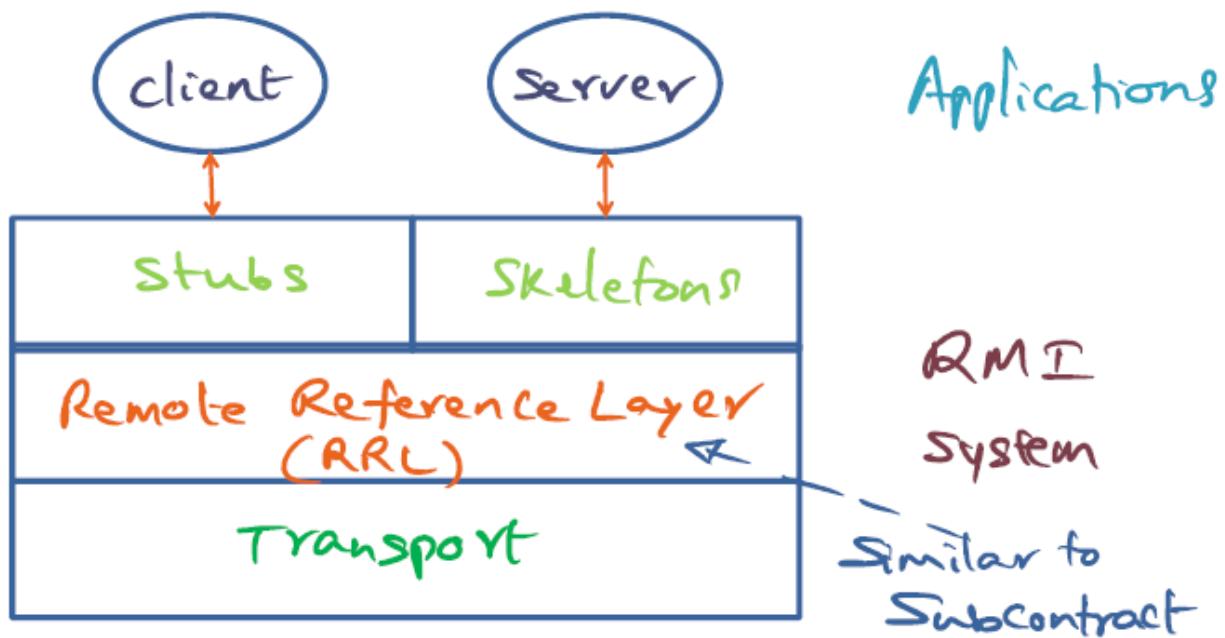
## RMI implementation

RMI is implemented using a remote reference layer (RRL). Client stubs initiate remote method invocation calls using RRL. Marshaling / unmarshaling, sending data over the network, etc. is handled by the RRL.

Similarly, the server uses the skeleton to interface with the RRL for marshaling / unmarshaling, etc. The skeleton calls into the server to execute the procedure, generate results, and the results are returned back to the RRL via the skeleton.

These events are called **serialization** and **deserialization** by the Java community.

## RMI Implementation - RRL



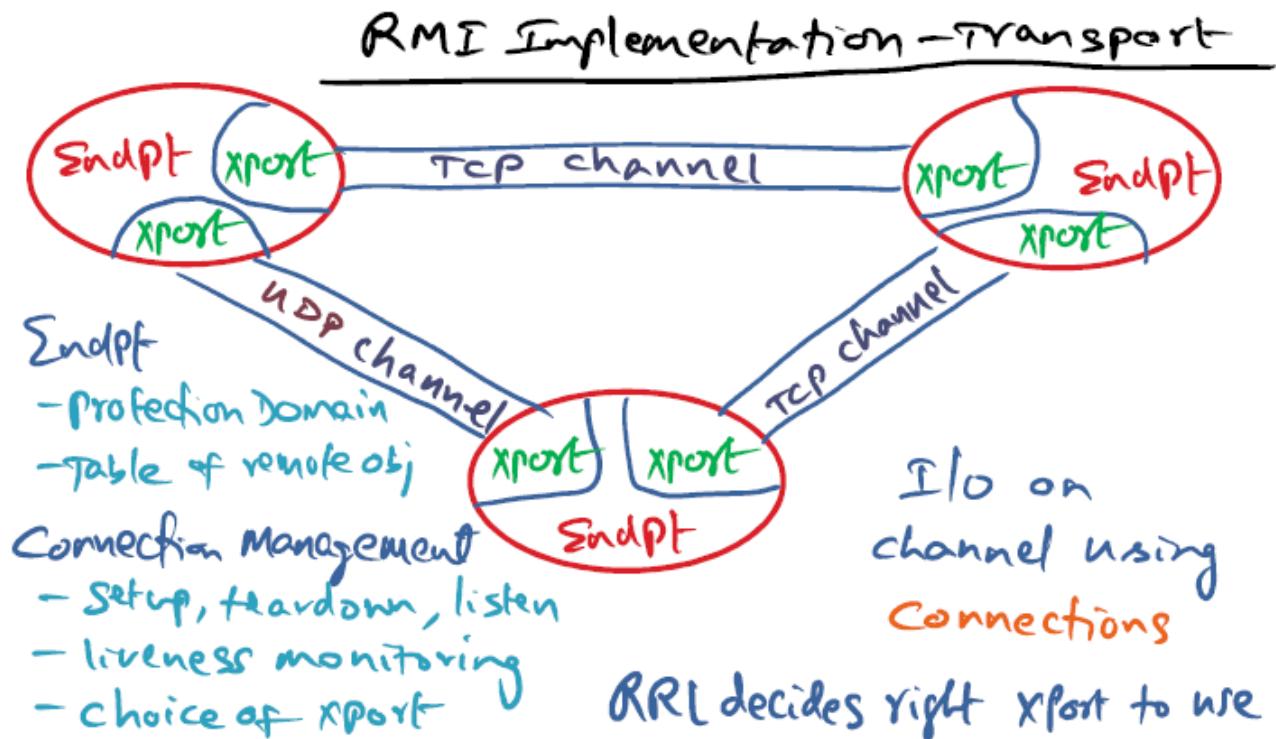
### Transport layer

Here are some abstractions provided by the transport layer in Java RMI:

Abstraction	Description
endpoint	protection domain; table of remote objects
connection management	<ul style="list-style-type: none"><li>setup, teardown, listening, establishing connections</li><li>liveness monitoring</li><li>choice of transport mechanism</li></ul>
transport	<ul style="list-style-type: none"><li>UDP, TCP</li></ul>
connections	used for I/O operations; established by the connection manager

The RRL layer makes the decision upon what type of **transport** is to be used to complete an invocation between a client and a server.

Below is a high-level representation of the concepts discussed above.



## Quizzes

Java was originally invented as a language for use in?

- the Android platform
- embedded devices
- B2B commerce on the WWW

Which implementation should be preferred and why?

### Question

Which implementation you prefer? And Why?

Local

Implementor makes instances of objects  
remotely accessible  $\Rightarrow$  not preferable

Remote

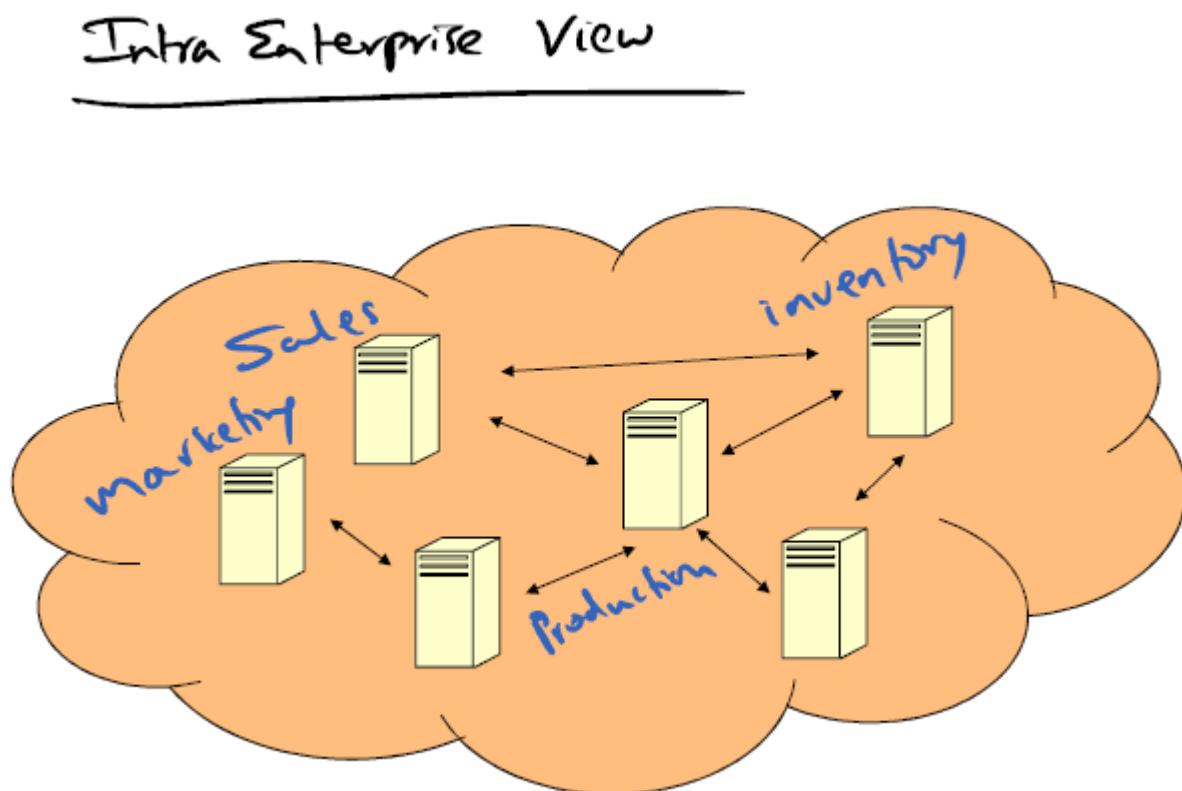
Java RMI does the heavy lifting to make  
server object visible to network clients  
 $\Rightarrow$  Preferable

# enterprise java beans

## Inter enterprise view

When using the internet or interfacing with an organization, everything looks super simple, however, an enterprise is made up of multiple networks.

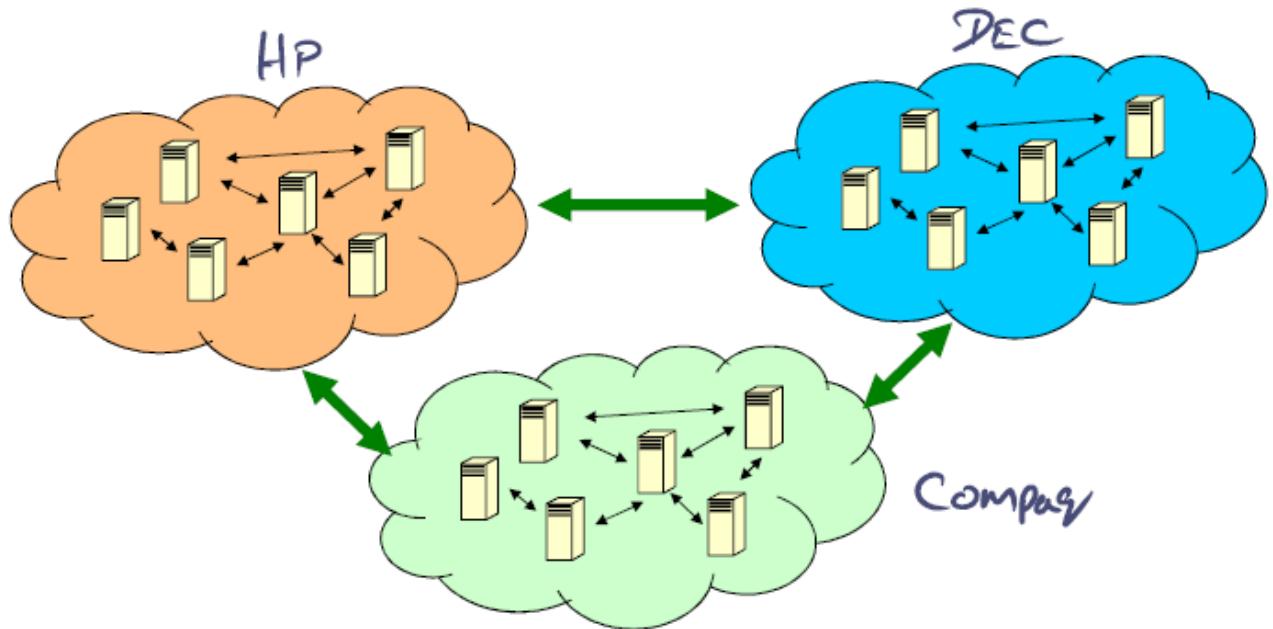
Below is a high-level representation of the intra enterprise view.



Now let's look at the complexity of the inter enterprise view. Using some service could include using multiple different enterprises, and as we can see, things can get pretty complex.

Below is a high-level representation of the inter enterprise view.

## Inter enterprise View



The challenges that we encounter in enterprise transformation are many, to include:

- interoperability
- compatibility
- scalability
- reliability

## N tier applications

N tier applications are some sort of service that we provide to customers that have multiple layers involved in completing transactions. When we provide these types of services, there are a few things that we focus on to ensure the service is worth using:

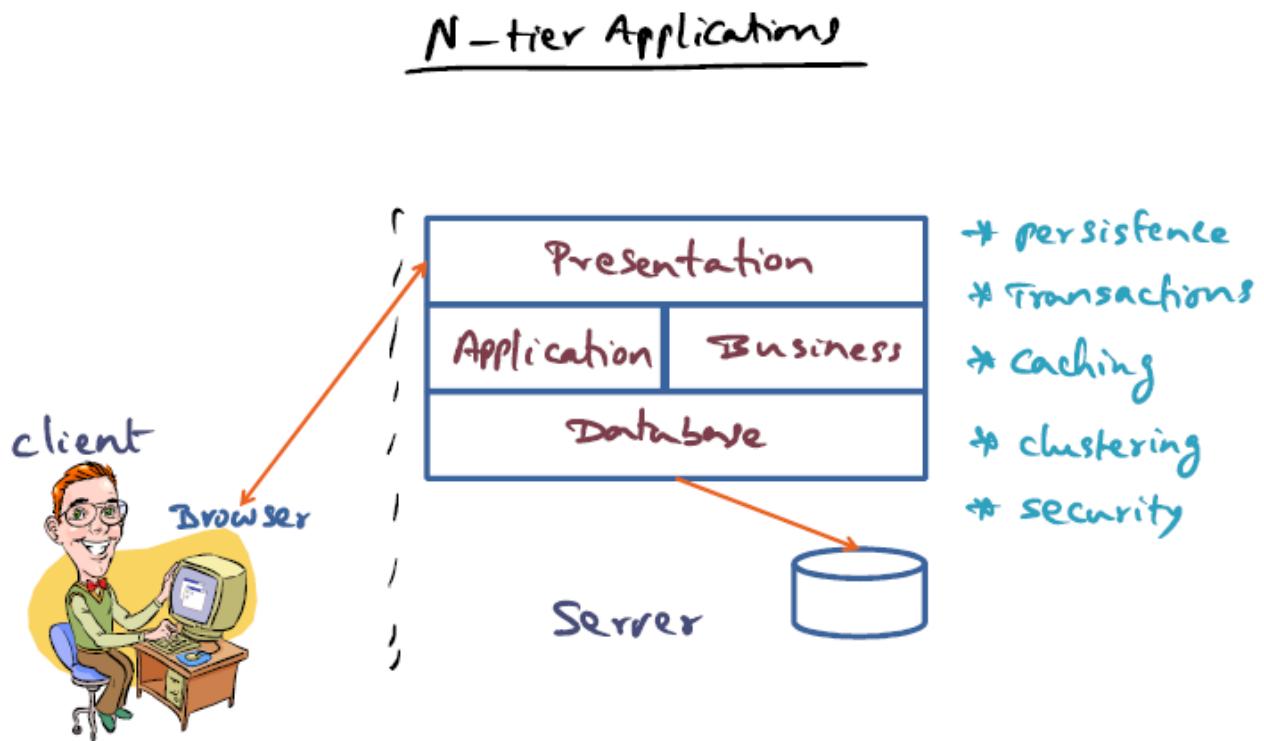
Concern	Description
persistence	retaining the data of a customer's session so that the customer can begin working from where they left off
transactions	some defined method of how transactions occur, like a shopping cart

---

caching	caching of the data closer to the customer so that the data doesn't have to be retrieved from the database each time the client wishes to view it
clustering	taking related services and grouping them together in order to improve their performance
security	ensuring that the customer's interaction with a service is completely private and secure and not abused by outside observers

---

Below is a high-level representation of the concepts discussed above.



So how do we structure an N tier application like this?

We want to reduce the amount of network communication that needs to be conducted, reduce security risks for the users, and we want to increase concurrency for handling an individual request.

## Structuring N tier applications

The Java Enterprise Edition framework utilizes four containers for constructing an enterprise application service.

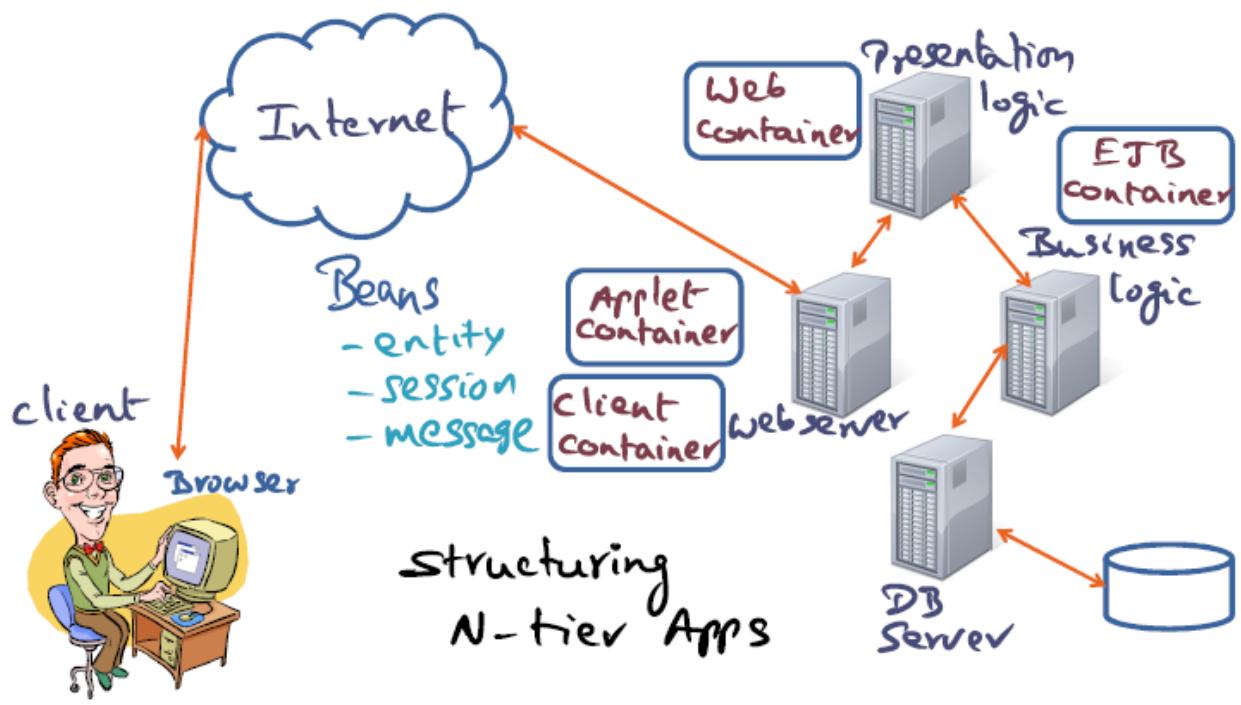
Container	Description
client container	typically resides on a webserver; interacts with the browser of the endpoint
applet container	typically resides on a webserver; augments the client container
web container	provides the presentation logic; dynamically creates pages
ejb container	manages business logic

The key hope is that we exploit as much of reuse as possible of components and this is done with **Java Beans**. The containers host the beans and allows you to package Java Beans and make them available within containers. Java Beans can represent:

Bean type	Description
entities	may be a row of a database; persistent objects with primary keys
session	associated with a particular client and session; temporal; could be stateful or stateless
message	useful for asynchronous behavior

The finer the grain of the implementation of these beans, the more we can leverage concurrency for requests. This also makes the business logic more complex, however.

Below is a high-level representation of the concepts discussed above.



## Coarse grained session beans

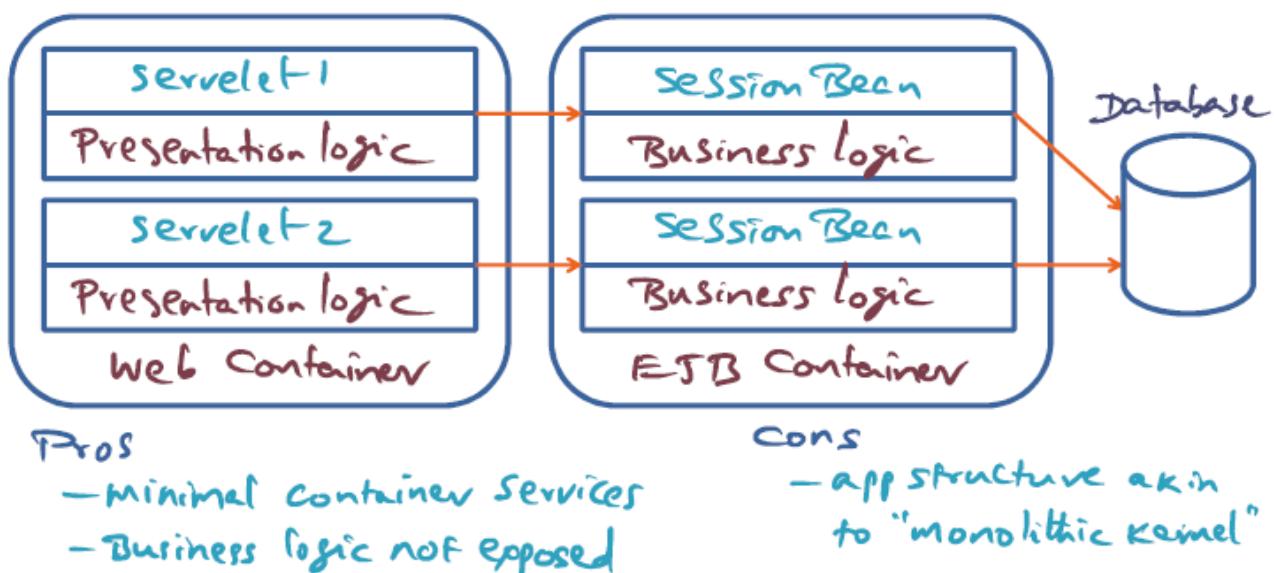
Servelet represents sessions with different clients. Coarse grained sessions beans are associated with each servelet. The session bean worries about the data accesses needed to the database in order for the business logic to execute correctly.

The EJB container usually helps the session beans in conducting their job, providing required services, however, these coarse grained beans are responsible for handling all interactions with the database required to complete the business logic. Thus, the EJB container isn't having to do much and only worries about mitigating conflicts that might arise in terms of external accesses to the database as sessions conduct concurrent operations.

Pros	Cons
<ul style="list-style-type: none"> <li>minimal container services required</li> <li>business logic not exposed to</li> </ul>	<ul style="list-style-type: none"> <li>app structure akin to "monolithic kernel"</li> </ul>

Below is a high-level representation of the concepts discussed above.

## Design Alternative 1 : coarse grain session Beans



## Data access object

Accessing the database is probably the slowest link in the entire process of providing our service to the customer. We need to leverage concurrency and parallelism and we do this by placing the business logic into the web container.

Instead of using sessions beans in the business logic container, we use entity beans to represent data access objects. This provides parallel access to the unit of granularity in terms of database access. Now servelets can create parallel database access requests, reducing the time for data accesses.

When servelets request the same database access, there is also an opportunity presented in which entity beans can cluster these requests and provide the data back to the clients requesting it.

Entity beans also represent some form of persistence, and there are two ways of doing this:

Persistence mechanism	Description
Container managed persistence (CMP)	the container facilitates persistence of the data access objects

---

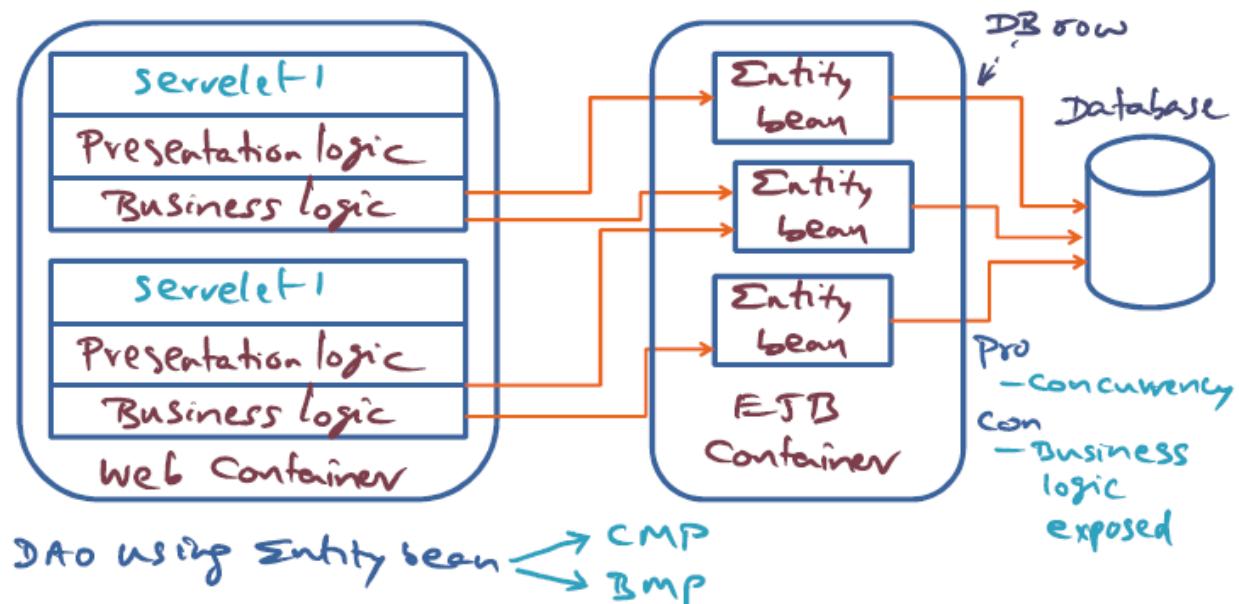
Bean managed persistence (BMP)	individual beans manage the persistence of data access objects
--------------------------------	--

---

Pros Cons

- concurrency
- business logic exposed to web container

## Design Alternative 2: Data Access Object



## Session bean with entity bean

This corrects the fact that the previous alternative exposed the business logic to the web container. We use **session facades** in tandem with the business logic. The entity bean data access objects remain. The **session facades** worry about the data access needs of the business logic it serves.

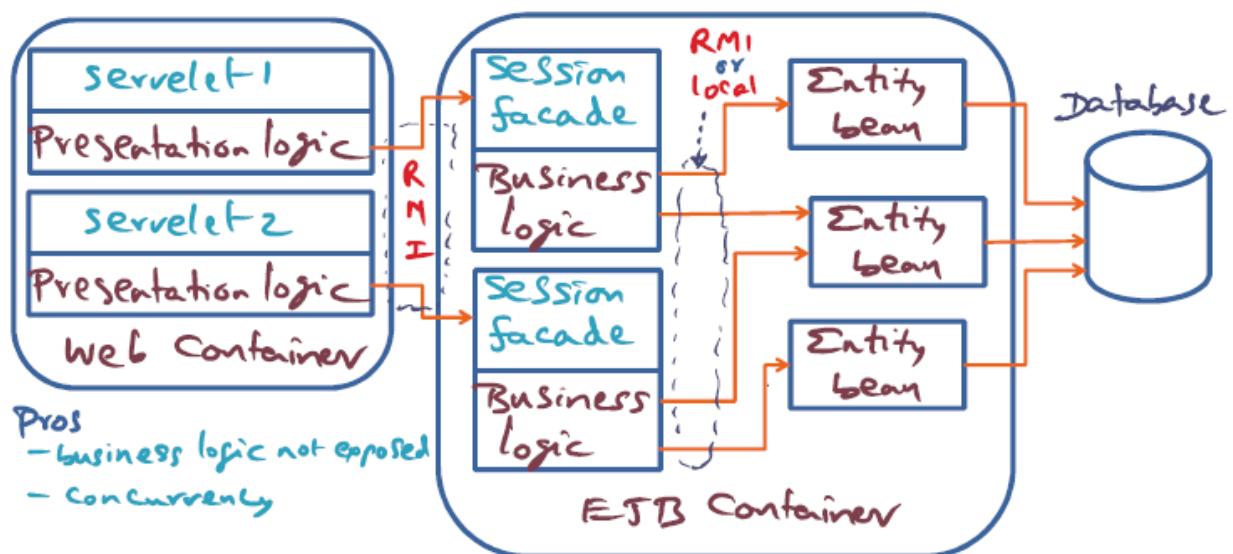
This structure allows us to still manage parallelism, because session facades can make concurrent calls to multiple entity beans that represent data access objects.

There are two methods for **session facades** to communicate with the entity beans: via RMI or local. RMI is useful because it doesn't matter where the entity beans are located, allowing

for flexibility. Local access provides us speed, but the entity beans have to be co-located. We can choose to construct using either / or.

Pros	Cons
<ul style="list-style-type: none"> <li>• business logic not exposed</li> <li>• able to leverage concurrency</li> </ul>	<ul style="list-style-type: none"> <li>• incurring additional network access in order to conduct data access (can be mitigated through co-location)</li> </ul>

### Design Alternative 3 : Session bean with entity bean



lesson7

# **lesson7**

# global memory systems

## Context

This portion of the slides provides some context for the need of global memory systems, or the problem that global memory systems attempts to solve.

For a distributed system, the **memory pressure** may be experienced differently across each node. Is it possible to use idle memory across the cluster to alleviate the **memory pressure** of a particular node? Is remote memory access faster than using paging from the node's local disk?

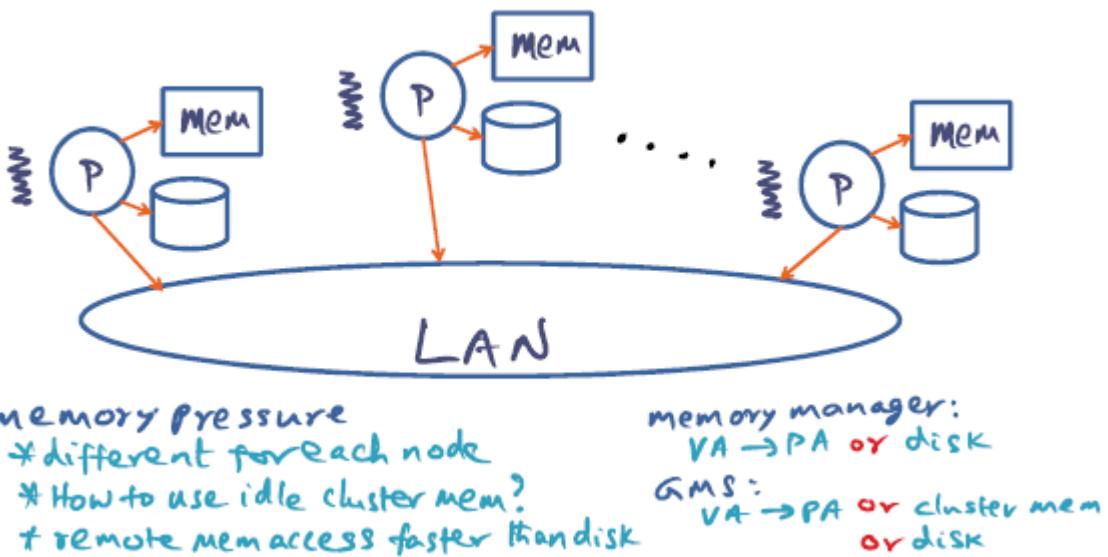
Because of the advances in networking technology, 10 Gb/s speeds from the LAN to a host are now possible, making it faster than reading approx. 2 MB/s from disk.

So the difference between this and regular memory management is this:

Memory management	Description
memory manager	<ul style="list-style-type: none"><li>• VA → PA</li><li>• disk</li></ul>
global memory system	<ul style="list-style-type: none"><li>• VA → PA</li><li>• cluster memory</li><li>• disk</li></ul>

GMS will only be used for reads. GMS will not interfere when it comes to writing to the disk. The only pages that can be in cluster memory are pages that have been paged out that are not dirty. This avoids worrying about node failures across the network.

## Context for global memory system



## GMS basics

Some basics about GMS:

- **Cache** refers to physical memory (i.e. **DRAM**) *not* processor cache.
- There exists a sense of **community** to handle page faults at a node.
- The physical memory of a node is comprised of a **working set** and **spare memory**. The **spare memory** is what is leveraged by the GMS to provide idle memory to other nodes experiencing memory pressure.
- Coherence of the shared pages across multiple nodes is up to the application.
- The **globally oldest page** in the system is the candidate page for page replacement.

There are two states for pages within the GMS:

State	Description
public	pages within the local cache of a node can be private or shared depending upon if that page is being actively shared by more than one node at a time
private	everything that resides within the global cache of a node are private copies of pages

## Handling page faults - case 1

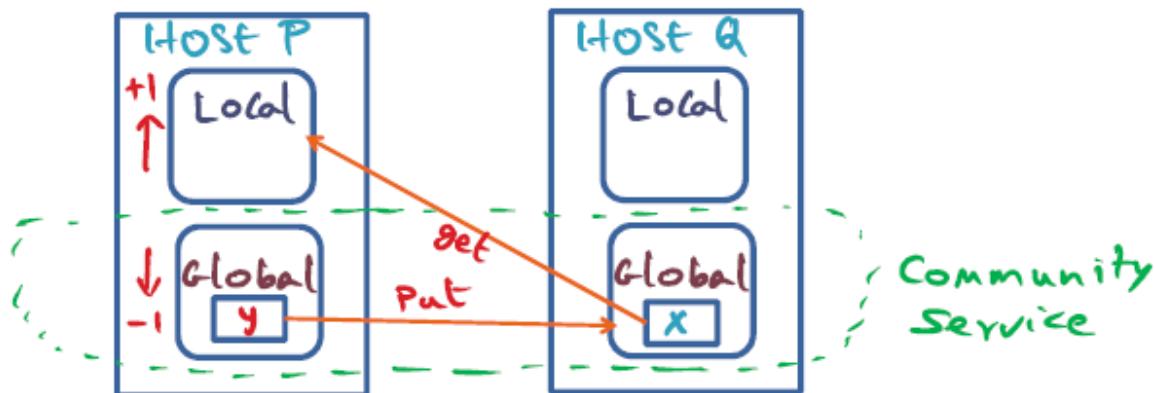
Below is a high-level representation for a case in which a process experiences some memory pressure and encounters a page fault. The process then has a hit for the page it's looking for in the global cache. The process brings the page from the global cache into its local memory, and pages out the oldest page into global memory.

The local memory space has increased by 1, and the global has decreased by 1.

### Handling Page faults - Case 1

#### Common Case

- Page fault for X on node P
- hit in global Cache of some node Q



### Handling page faults - case 2

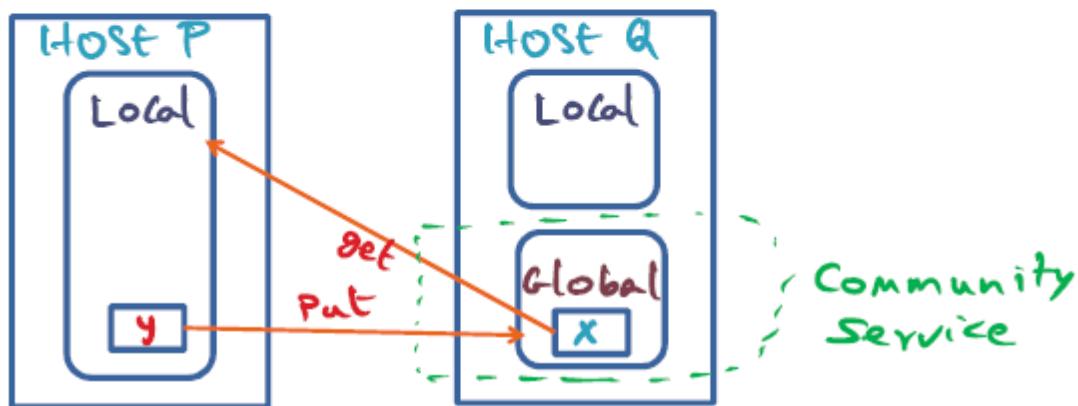
In the below high-level example, all of the memory is being consumed by the working set in Host P, there is no free memory available to provide global memory to the community service.

The only difference between this case and the previous one is that the oldest page, the candidate page, is being paged from local memory into global memory.

## Handling Page faults - Case 2

Common case with memory pressure at P

- Page fault for X on node P
- swap LRU page Y for X



## Handling page faults - case 3

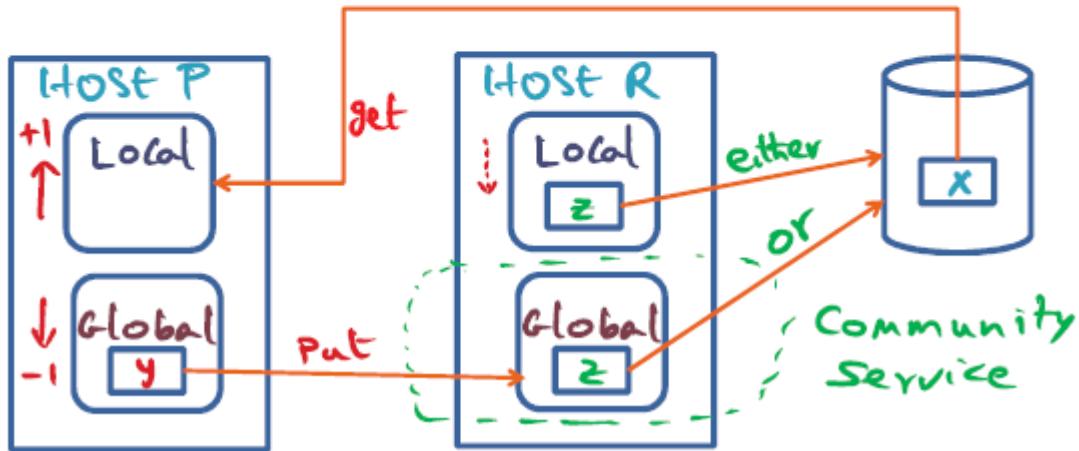
In the below high-level example, the faulting page is not contained within clustered memory. The only copy of the requested page resides on disk. GMS will identify the globally oldest page and swap that page to disk. As the original faulting node receives the page from disk, it will decrease its global memory space, replacing a page and placing it in the community space.

If the globally oldest page is clean, we can just drop the page entirely. This happens usually if the globally oldest page being replaced is in the global cache. It must be written to disk if it's dirty, this usually happens if the globally oldest page is in the local cache.

## Handling Page faults - Case 3

Faulting page on disk

- page fault for X on node P
- page not in cluster



## Handling page faults - case 4

In the below high-level example, a node page faults on a page that is actively being shared. The GMS will notice that the page currently exists within a different node's local cache, and instead of ripping that page away from that node, it will just copy the page to the local cache of the faulting node.

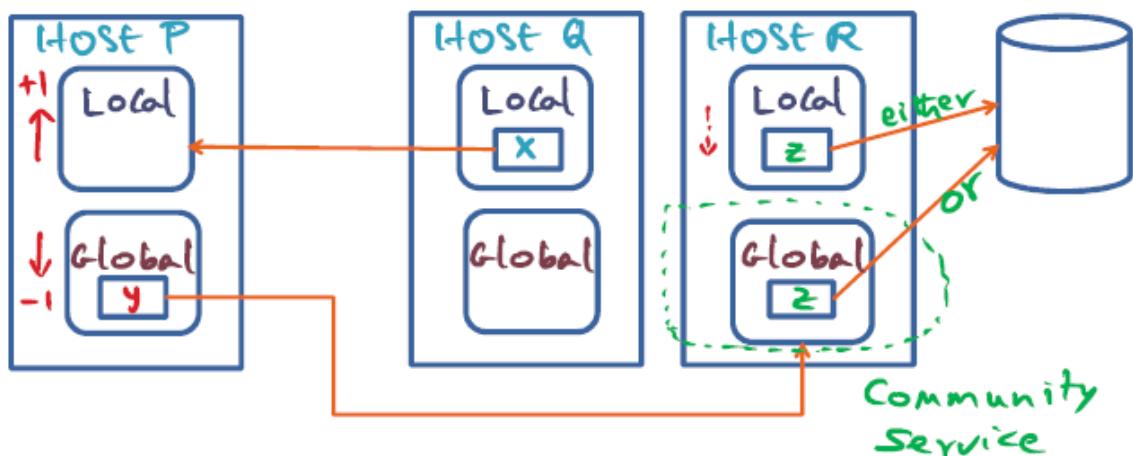
The faulting node will increase its local cache space, and the global cache space of the faulting node will decrease. The global cache space of the faulting node will be pushed to community space, and the globally oldest page will be swapped to disk.

The same swapping requirements occur if the globally oldest page is dirty or clean.

## Handling Page faults - Case 4

Faulting Page actively shared

- Page fault for  $X$  on node P
- page in some peer node Q's local cache

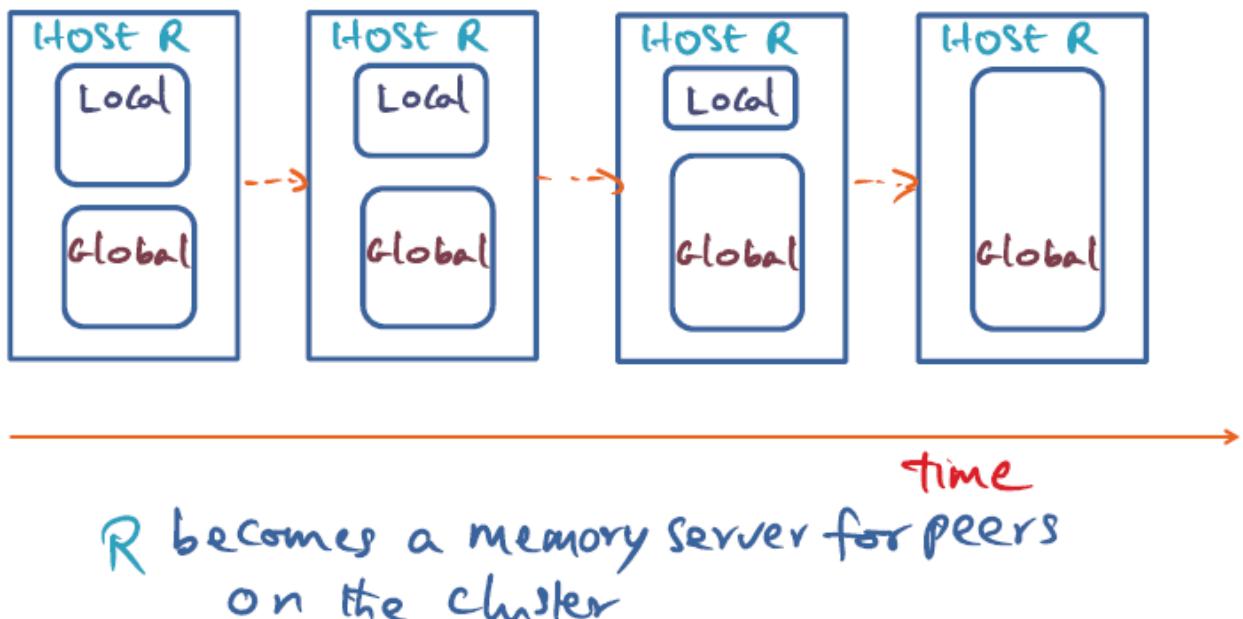


## Behavior of the algorithm

The behavior that emerges from this algorithm is described in the high-level representation below. Essentially, the most idle host in the cluster will gradually become a memory server for the rest of the clusters as its local memory shrinks to accommodate the global memory cache.

If the idle node begins doing some work, however, the split between local and global is not static. Thus, the node should begin to start reclaiming memory for private use as its working set increases in size.

## Behavior of Algorithm



## Geriatrics

We must manage the age of the oldest page within the GMS in order to swap it to disk when necessary. The age management cannot be assigned to one particular system, we must distributed the management workload evenly across all nodes within the cluster.

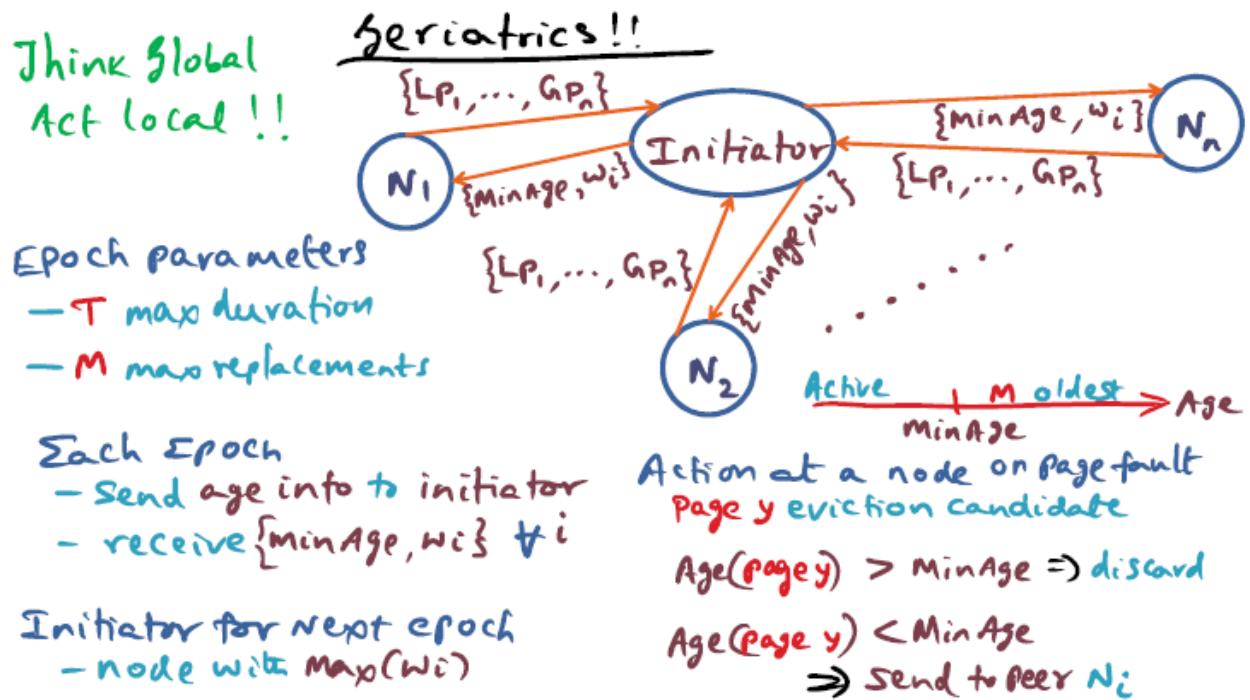
With this, the GMS establishes an **initiator** who declares that pages will be swapped in the upcoming epoch. In each epoch, each node sends the age of each page to the **initiator**. The **initiator** calculates a **minimum age** - this **minimum age** will be the youngest age a page can be before it is swapped. The **initiator** also calculates a weight for each node dependent upon the percentage of page replacement being conducted for that node in comparison to the rest of the cluster. The **minimum age, weight** tuple for all nodes gets sent back to each node, notifying them of the candidate pages for swapping.

The **initiator** of the management routine at the next epoch becomes the node with the maximum **weight**. We can intuitively determine that the node with the maximum **weight** is not very active. This makes it the prime candidate for becoming the next **initiator**.

For each node when it comes to page replacement, if a page happens to be older than the **minimum age**, if we encounter a page fault we can just discard that soon to be evicted

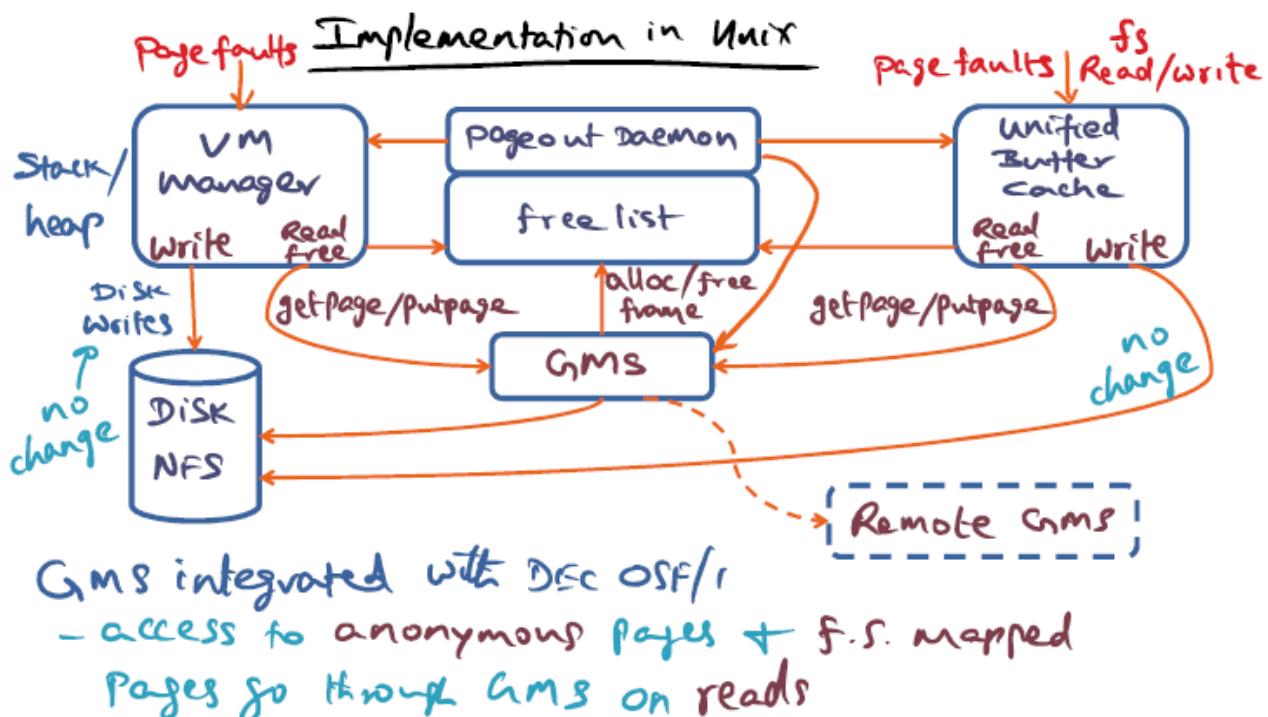
page. If a page incurs a page fault and it's less than **minimum age**, we'll send the page to a peer node.

Below is a high-level representation of the concepts discussed above.



## GMS implementation in UNIX

GMS requests a daemon to dump the TLB, and inspects this information to derive the age information for pages at every node.



## GMS distributed data structures

To track pages, GMS converts virtual addresses from nodes into **universal IDs**. These **universal IDs** uniquely identify a page. There are three key data structures utilized by GMS to manage **universal IDs**:

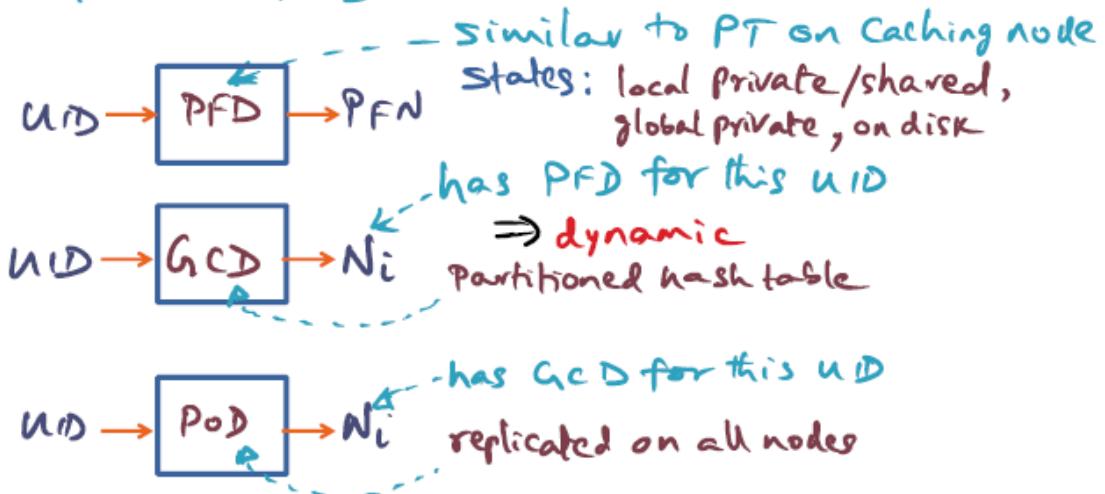
Data structure	Description
PFD (page frame directory)	similar to a page table, converts UIDs to page frames; pages can be in four different states: <ul style="list-style-type: none"> <li>• local private</li> <li>• local shared</li> <li>• global private</li> <li>• on disk</li> </ul>
GCD (global cache directory)	cluster-wide hash table; returns the node that holds the PFD for a UID
POD (page ownership directory)	replicated on all nodes; returns the node that holds the GCD for a UID

## Data structures

VA → UID      

IP-Addr	disk partition	i-node	offset
---------	----------------	--------	--------

  
- derived from VM + UBC



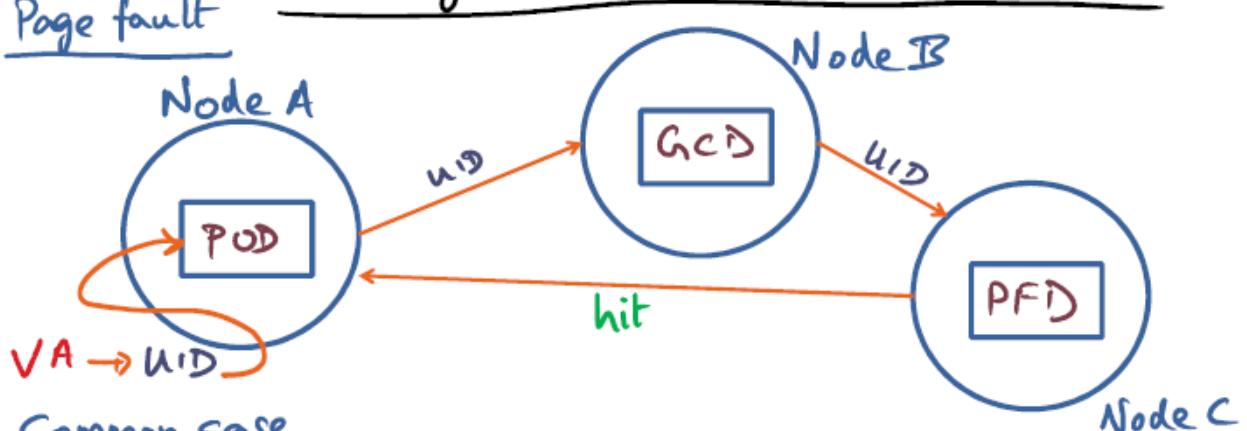
## Putting the data structures to work

Below is a high-level representation of how the data structures described above assist in a node resolving a page fault across a cluster. Up to three communication events can occur in order to find a page in a cluster.

The below representation shows an uncommon case, usually page faults occur for **non-shared** pages - Node A will usually contain both the PoD and the GCD.

## Putting the Data structures to work

Page fault



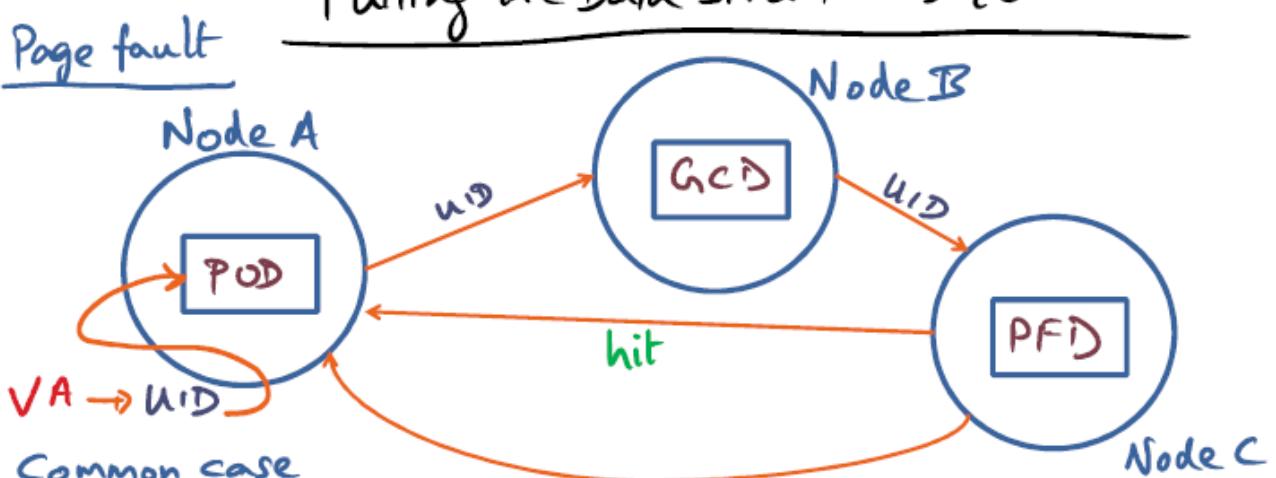
Common case

- Page non-shared
- $\Rightarrow$  A + B same
- Page fault service quick

What happens when the PFD has a miss? This is possible when the PFD evicts a page but has yet to notify the GCD holder. This is common scenario is common. An uncommon scenario, however, is when the POD of the node that has a page fault has a stale POD. Remember, PODs are stored by all nodes - stale PODs will only occur when a POD update has not propagated to the node yet. POD updates occur when there is the addition or deletion of a node from the cluster.

## Putting the Data structures to work

Page fault



Common case

- Page non-shared
- $\Rightarrow$  A + B same
- Page fault service quick

miss?

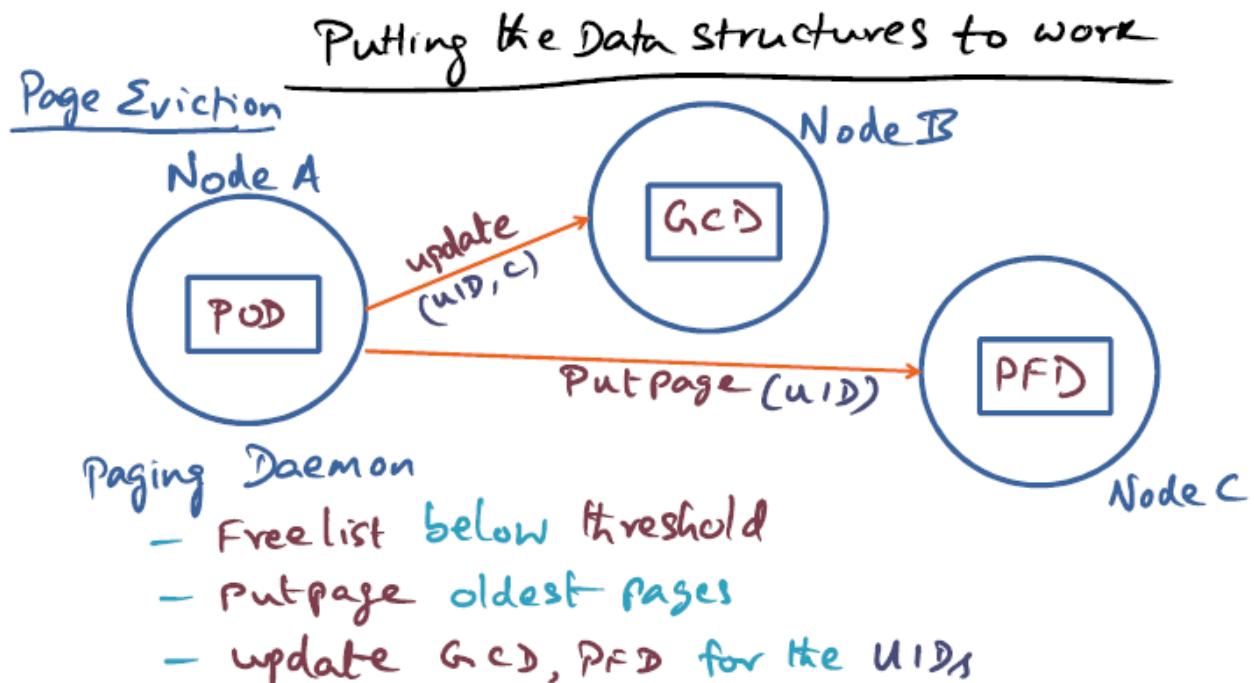
$\Rightarrow$  Uncommon Case

- POD changing due to node addition/deletion

Every node has a paging daemon as discussed earlier. The virtual memory manager utilizes the paging daemon to conduct the page replacement algorithm and execute page replacement mechanisms. Of note, the virtual memory manager always contains some free pages on the node for page faults, because we want to conduct our operations in an aggregate manner.

As noted earlier, the paging daemon is integrated with the cluster's GMS. When the free pages list of a node falls below a certain threshold, the paging daemon. Essentially we modify the virtual memory manager and the unified buffer cache to visit GMS prior to attempting to resolve page faults by using the disk. Writes to disk are unchanged. The pageout daemon will now defer to GMS when discarding clean pages - dirty pages will still be written to disk. It will place the oldest pages on the node onto a different node, coordinating with the cluster's GMS. The candidate node is chosen based on the weight information derived from the geriatric management that occurs at each epoch.

The page daemon then updates the GCD with a new PFD for the respective UID that was evicted to a different node. Again, all of these operations do not occur on every page eviction, but aggregately when meeting the free page threshold.



## Quizzes

Fill in the table with what happens to the boundary between local and global on each node.

Faulting Page X	Faulting node P	Node Q with page X	Node R with LRU page
	L G	L G	L G
in Q's global	+1 -1	no change	no change
in Q's global P's global empty	no change	no change	no change
on disk	+1 -1	Not applicable	EI +1
actively shared with Q	+1 -1	no change	EI +1

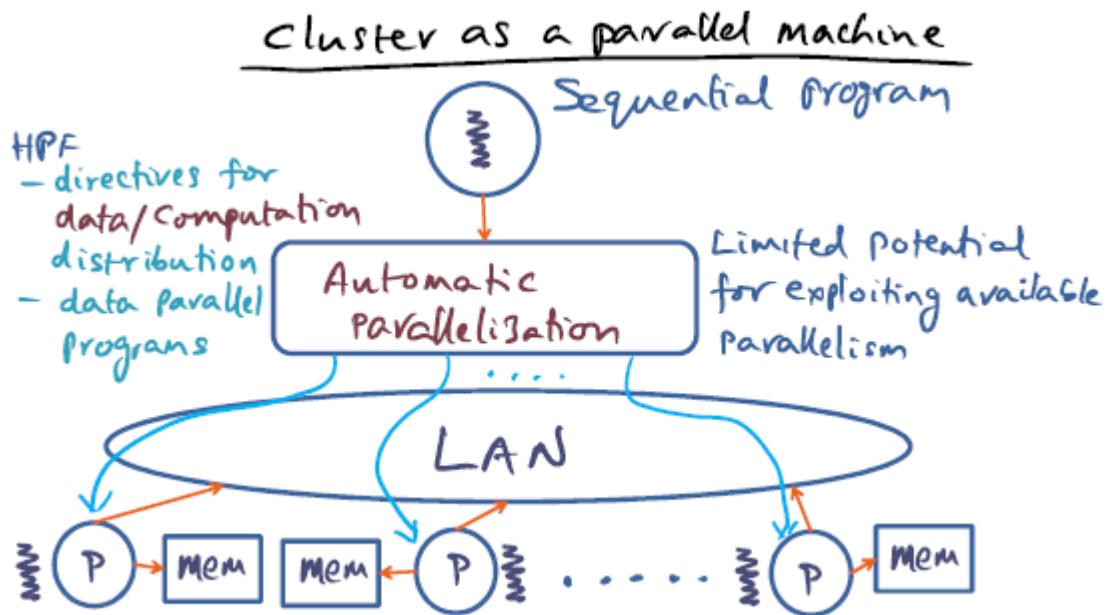
# distributed shared memory

## Cluster as a parallel machine

We can leverage what's known as **automatic parallelization** to leverage the nodes of a cluster for parallelizing a sequential program. This is what is called an **implicitly parallel program**.

**Automatic parallelization** is done at compile time - an example would be using **High Performance FORTAN** which leverages **automatic parallelization**. HPF provides the programmer to use directives for **data / computation distribution**.

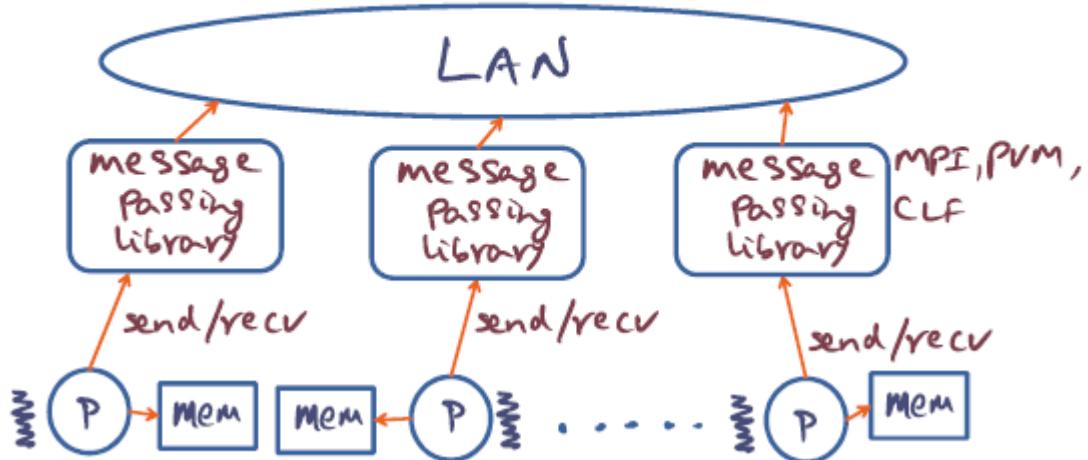
Below is a high-level representation.



There's also the ability to create explicitly parallel programs leveraging message passing between nodes. The programming will leverage a **message passing library** to leverage the true physical nature of the cluster. The only problem with this style of parallelization is that it's a bit tougher to program using **message passing libraries**. Programming with shared memory definitely easier and requires less of a change in thinking.

## Cluster as a parallel machine

Parallel program: message passing

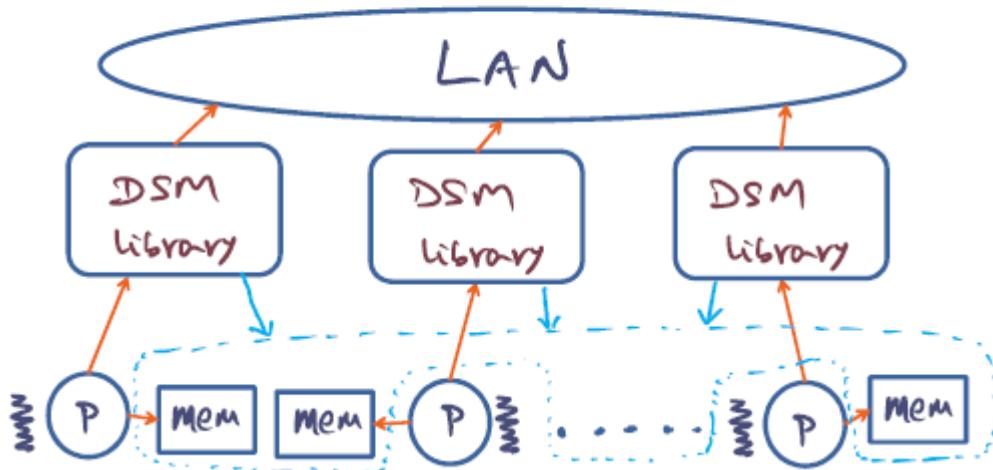


All of what we discussed above drives the motivation for creating the abstraction of distributed shared memory. We give the illusion to the programmer that all of the memory in the cluster is shared using a **distributed shared memory library**.

The programmer can use the same set of primitives to parallelize a sequential program without having to tailor their implementation to the clustered environment.

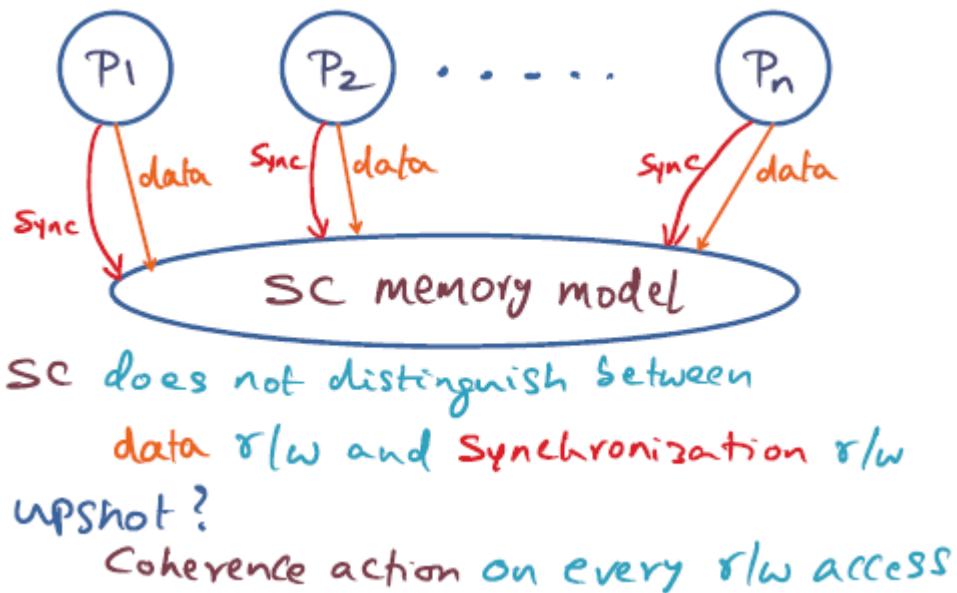
## Cluster as a parallel machine

Parallel program: distributed shared memory



Sequential consistency memory model

We discussed in past lectures memory consistency models and the sequential consistency model. Luckily for us, the sequential consistency model does not distinguish between data read / write operations and synchronization read / write operations. This provides us with the advantage of coherence actions on every read / write access to the distributed shared memory.



## Typical parallel program

Using the example of a typical parallel program, we show that, in the sequential consistency model, there is delineation between lock read / write operations and shared memory operations. With that in mind, once a process acquires a lock it can safely assume the shared memory it modifies is same from outside read / write operation.

Thus, there is no need for coherence actions to occur across the cluster for shared memory that is locked until the lock is released.

## Typical parallel program

P<sub>1</sub>

```
Lock(L); //acq  
read(a);  
write(b);  
Unlock(L); //rel
```

P<sub>2</sub>

```
Lock(L); //acq  
write(a);  
read(b);  
Unlock(L); //rel
```

P<sub>2</sub> does not access  
data P<sub>1</sub> releases L  
 $\Rightarrow$  no need for Coherence action for a and b  
until release

## Release consistency

The above discussion spurred the need to research **release consistency**. The gist of **release consistency** is that, if two processes utilize the same lock, all coherence actions need to take place prior to the release of the lock by a process and the acquisition of a lock by another process that's protecting some shared memory.

With this mechanism, all coherence actions only have to happen at the release point. This allows us exploitation of computation on a process with communication that may be happening through the coherence mechanism corresponding with the memory actions that may be taking place inside a critical section.

## Release Consistency

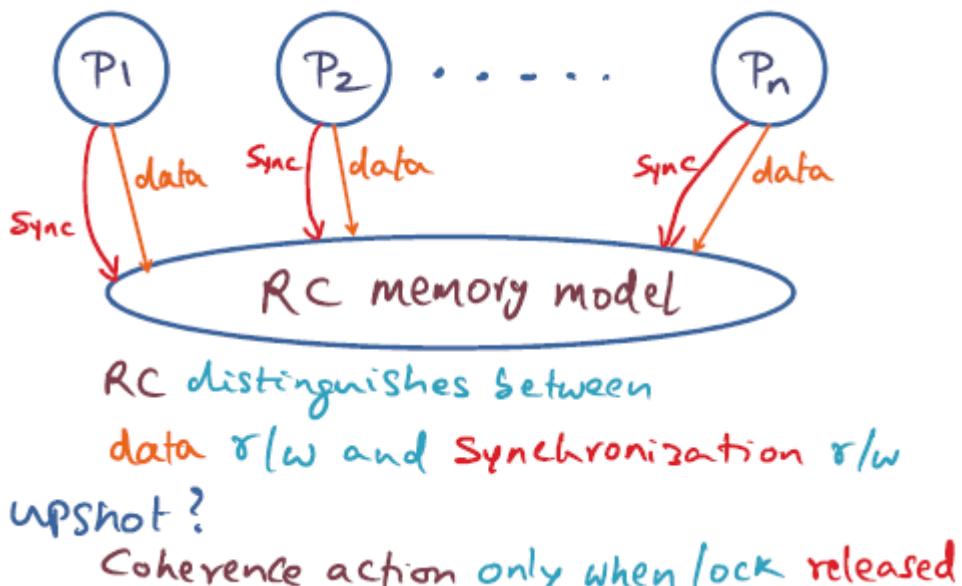
P1:  
 $a_1: \text{acq}(CL)$   
 data  
 accesses  
 $r_1: \text{rel}(L)$

P2:  
 $a_2: \text{acq}(CL)$   
 data  
 accesses  
 $r_2: \text{rel}(L)$

If  $P1: r_1 \xrightarrow{\text{hb}} P2: a_2$   
 — All coherence actions prior to  $P1: r_1$   
 should be complete before  $P2: a_2$

## Release consistency memory model

The release consistency model actually distinguishes between data and synchronization read / write operations. Coherence actions will only take place when locks are released.



## Advantage of release consistency over sequential consistency

## Advantage of RC over SC

- no waiting for coherence actions on every memory access

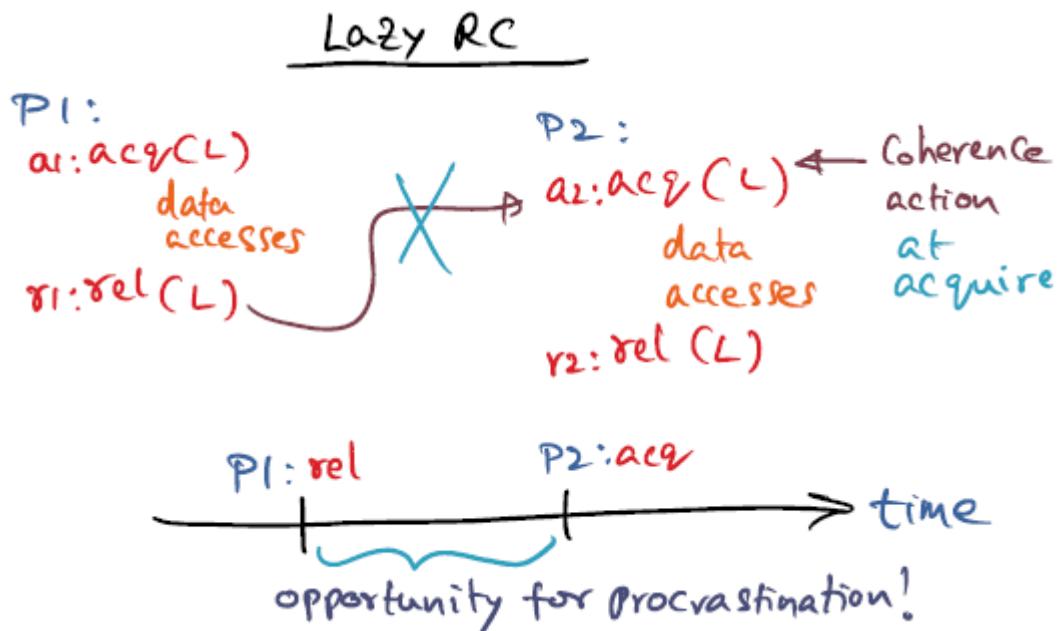
⇒ overlap computation with communication

⇒ better performance for RC over SC

## Lazy release consistency

Eager release consistency is what we described earlier. All of the coherence operations take place prior to the release of the lock.

For lazy release consistency (LRC) coherence actions only occur upon the acquisition of the lock by some other process.



## Eager vs Lazy RC

In Eager RC, all release operations require a broadcast push to all processes. In Lazy RC, all acquisition operations are just a pull, no broadcast required.

Pros of Lazy RC over Eager?

- Less messages across the network

What are the cons of Lazy RC over Eager?

- More latency experienced during lock acquisition.

## **Software distributed shared memory**

Distributed shared memory (DSM) cannot manage the global memory for a cluster like what would normally happen for multiple processors sharing memory, as the hardware usually conducts all of the coherence operations - this is infeasible across a network.

To combat this, DSM provides a global virtual memory abstraction for all processors within a cluster, and partitions address space of the global virtual memory to different processor. DSM exercises distributed ownership, each process is responsible for the consistency of its assigned partition of global memory.

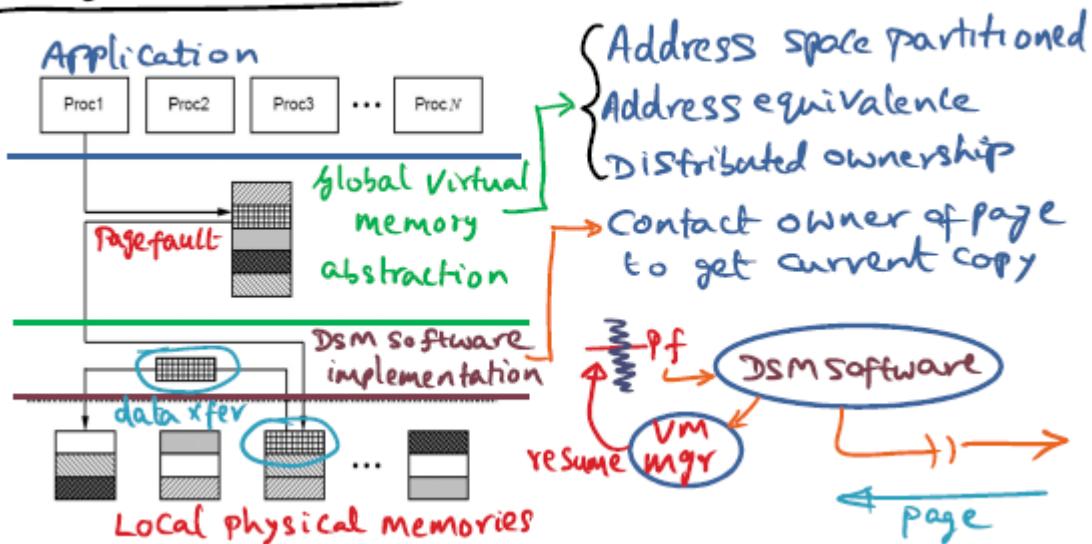
The DSM software layer tracks ownership of pages, providing it the ability to acquire the owner of a page in order to make a copy of the page. This way, the DSM software can handle page faults for processors across the network. The DSM software will then contact the virtual memory manager of a processor to update the page table of the processor that is experiencing a page fault.

DSM implements a single-writer protocol in which processors that wish to write to a page will notify the DSM, the DSM will notify the page owner, and then the page owner will invalidate all copies of the page across the system to maintain coherency. Any number of readers can read a page, of course.

There still exists the possibility of false sharing, just like in a shared memory multiprocessor. The granularity of consistency is being conducted at pages, so small parts of the page can cause invalidation if updated - this causes ping-ponging of the pages as invalidation occurs.

With the information described above, we can discern that the single-writer protocol and page based coherence granularity don't play well together.

## Software DSM



## Multiple-writer coherence protocol with LRC

To combat false sharing, we essentially create **diffs** of items that were modified on a page in shared memory. When another processor acquires the lock, it **invalidates** the pages. When the processor enters a critical section, the processor will then fetch the current version of the page by consulting the DSM for the owner of the page.

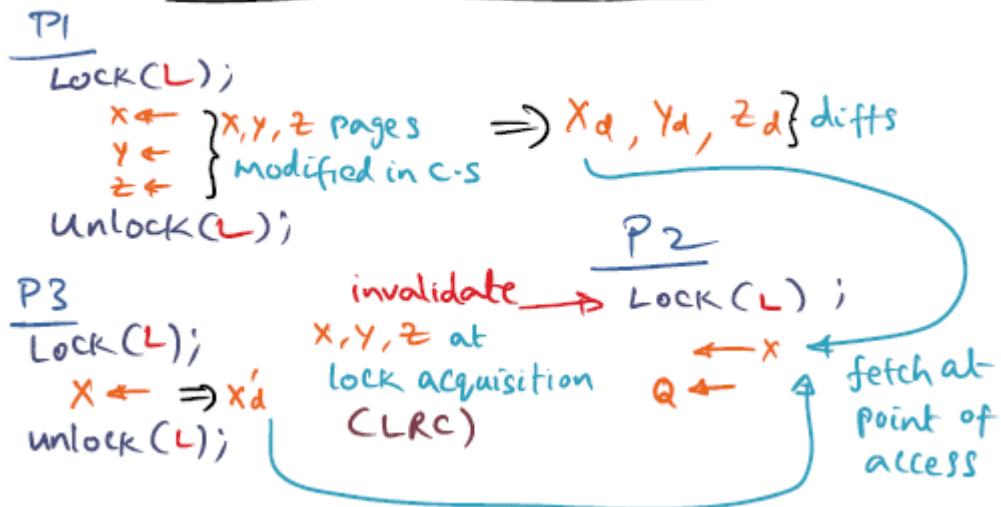
So what happens when there have been two changes to a shared page prior to a processor acquiring the lock? The DSM software knows that two **diffs** exist for a page, and will acquire the diffs from the different processors. Then it will provide the diffs and the original page to the processor attempting to access the page.

If a new page is modified for a lock, the new page that has been modified gets associated with that lock. Now, when other processors acquire the lock, they will also invalidate the pages and have to fetch them from the DSM when they read / write.

So where is the multiple writer portion of this?

DSM differentiates between locations within shared pages based upon the locks used to access specific portions of memory. So if a different lock is used than the ones used previously to read / write to a location on the page that is completely unrelated, the DSM knows that that particular memory location is unrelated - preventing false sharing.

## LRC with Multi-Writer Coherence Protocol



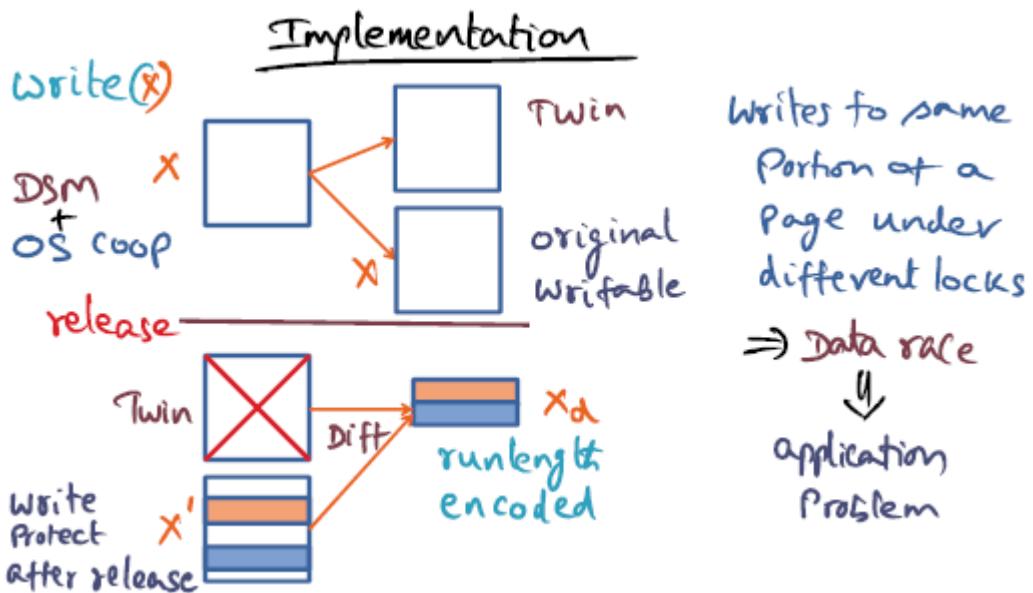
## Implementation

When a write occurs for a page, the DSM creates a twin and maintains the original page. All writes are made to the original page, the twin exists but does not reside within anyone's page table.

Upon lock release, the DSM computes the **diff** of the twin and the now modified page - into a **runlength encoded diff**. The original page is then write-protected, and cannot be written to until the lock is acquired again. The twin pages is destroyed. When a different processor acquires the lock, the DSM provides the **diff** of the edited page to the processor when it attempts to read / write to the modified page.

What happens when writes happen to the same location on a page under different locks? This is a **data race**, and this is the programmer's fault. Nothing DSM can do about it.

After a while, there could be diffs lying around lots of nodes. It would be a hassle to go gather all of the diffs for a processor attempting to use the page right? DSM uses garbage collection. If the page has begun to change enough, the owner of the page will acquire all of the diffs of the page and will apply the diffs to the original copy of the page. This occurs when a paging daemon checks up on the diffs created at its node.

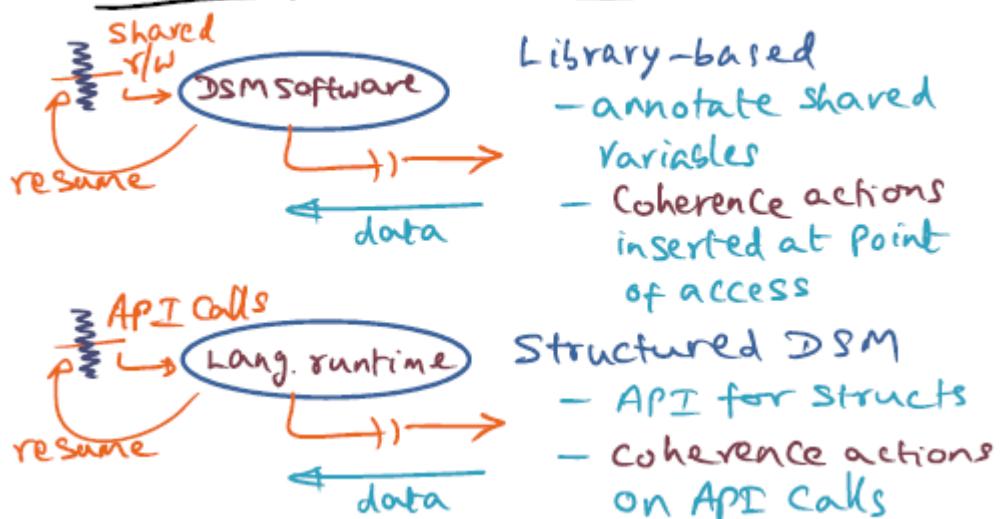


## Non-page based DSM

There are methods to conduct non-pages base DSM that won't be too low of granularity where a performance hit occurs. These solutions are implemented for specific binary applications, requiring no operating system support.

Granularity	Description
library-based	<ul style="list-style-type: none"> <li>• provides programming to annotate shared variables</li> <li>• coherence actions inserted at point of access</li> </ul>
structured dsm	<ul style="list-style-type: none"> <li>• API for shared structures</li> <li>• coherence actions take place during API calls.</li> </ul>

## Non-page-based DSM



## Scalability

As usual, as the number of processors increase, performance might decrease due to the amount of overhead. This is true in the case of shared memory multiprocessors, and it's also true in the case of DSM.

Scalability  
Expectation with more Processors?

<b>Pro:</b> Exploit parallelism	<b>Con:</b> Increased overhead
------------------------------------	-----------------------------------



## DSM and speedup

If the sharing is too fine-grained across a cluster, speed up expectations dwindle with DSM. The computation to communication ratio has to be very high if we want to achieve

performance increase - our critical sections need to be lengthy.

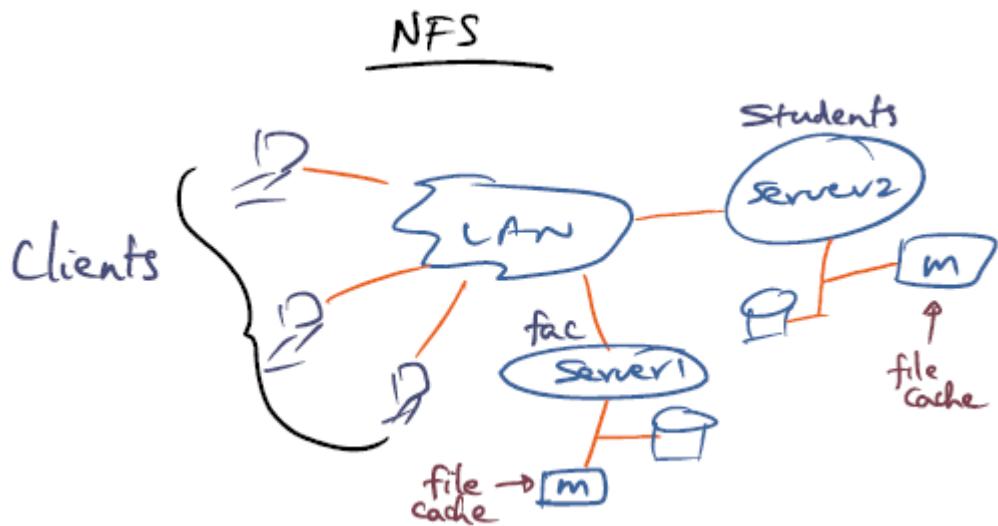
In addition, if the code has a lot of dynamic data structures manipulated with pointers, this can result in implicit communication across the network. Pointer codes may result in increasing overhead for distributed shared memory.

# distributed file systems

## NFS

Clients on the network, servers on the network, files distributed across servers, clients view the file system as one centralized view. Servers cache files that it retrieves from the disk into memory to avoid reading from disk all the time.

A single server fielding client requests becomes a bottleneck for performance. The file system cache is also limited because it's confined to the space within the server. Can we distribute file system access?



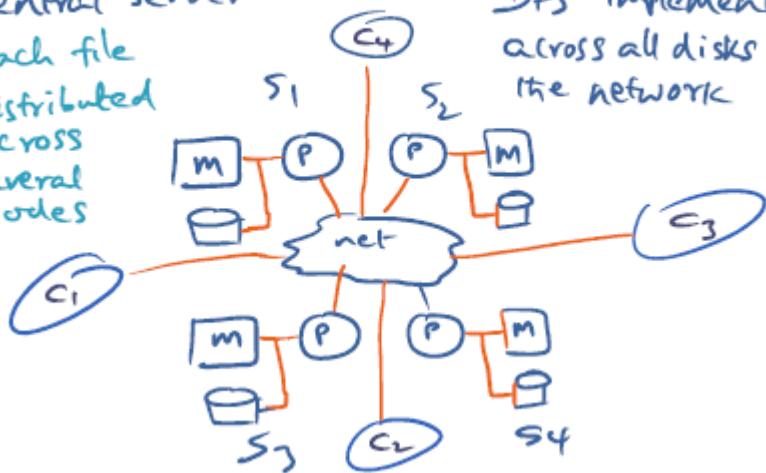
## DFS

With a DFS, there is no central server for a set of files. We distribute files across several different nodes across the network. The I/O bandwidth that's available can be cumulatively leveraged to serve clients. Our file cache size increases as we can leverage the caches of all the servers in the cluster. These mechanisms start to push us towards the implementation of *server-less* filesystems.

## DFS

No Central Server  
— Each file distributed across Several nodes

DFS implemented across all disks in the network



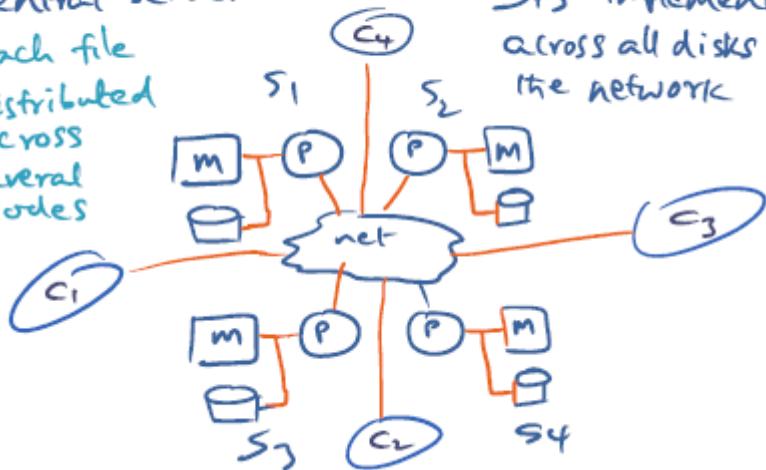
## XFS

Leveraging log-based striping, cooperative caching, dynamic management of data + metadata, subsetting of storage servers and distributed log cleaning to created a distributed file system.

## DFS

No Central Server  
— Each file distributed across Several nodes

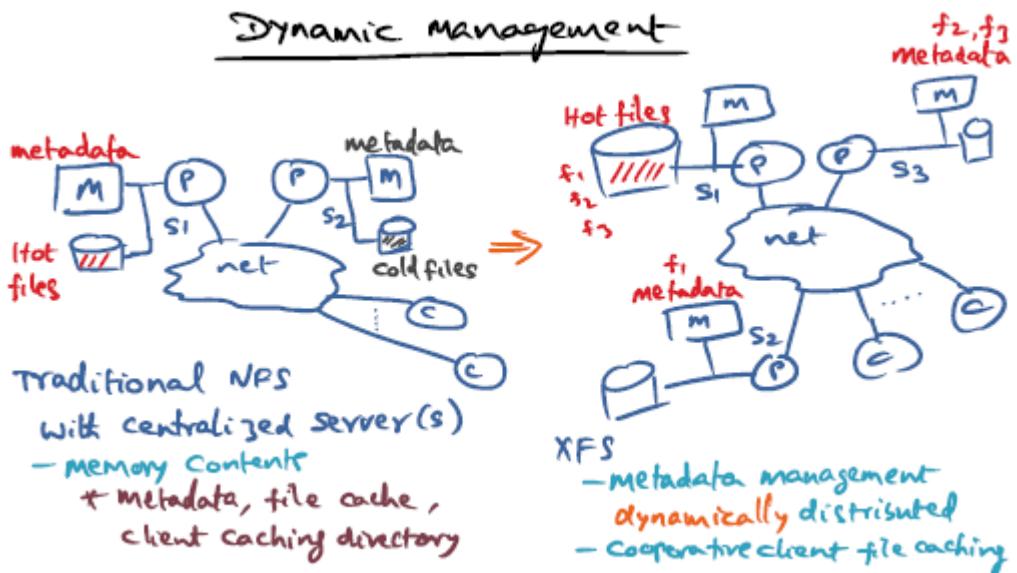
DFS implemented across all disks in the network



## Dynamic management

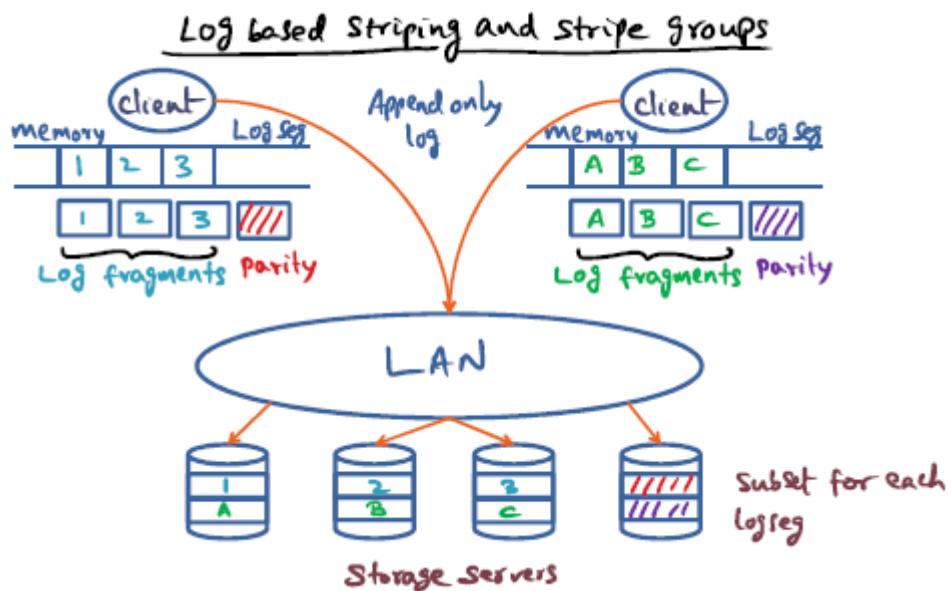
In comparison to a traditional network file system, metadata management for a file is dynamically distributed across all nodes within the cluster. This way, when a server

contains **hot files**, other idle servers can handle metadata requests. This also provides the ability for nodes to conduct cooperative file caching.



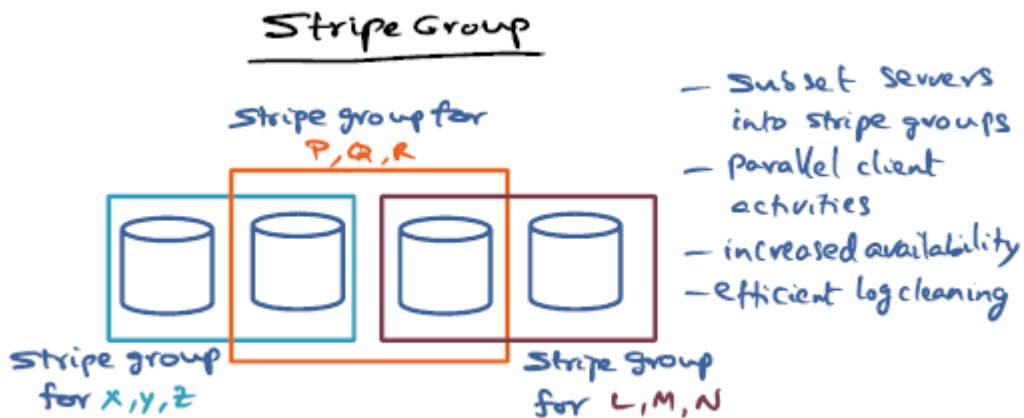
## Log based striping and stripe groups

For log segments, we implement what are called stripe groups. Stripe groups are a subset of the cluster of disks available to write log segments to, because we don't want to write log segments across the entire cluster - avoid small writes.



## Stripe group

Below is an example of stripe groups. Allows for parallel client activities, increased availability and efficient log cleaning. Different subset of servers can complete different client requests, increasing performance. Different cleaning services can be assigned to different stripe groups as well. If a server fails, other stripe groups will still be able to serve clients of that particular stripe group - our system won't degrade because we didn't distribute logs across the entire cluster.



## Cooperative caching

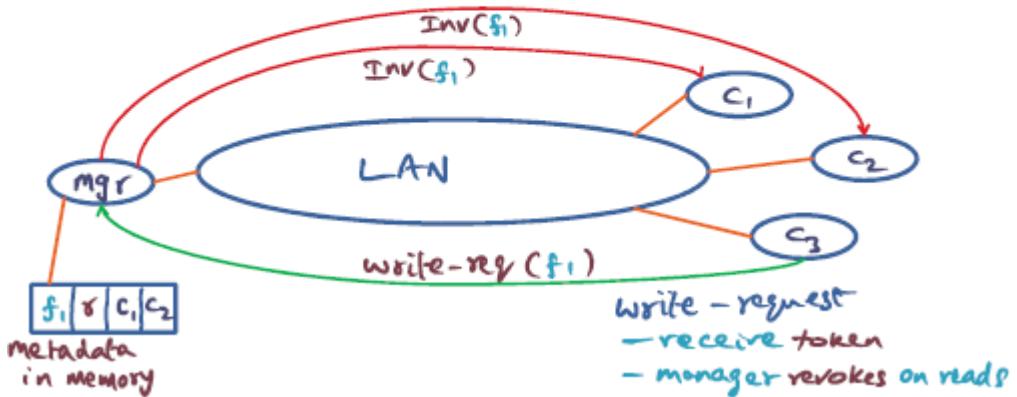
File blocks are used as units of coherence. A manager manages the coherence of a file block and tracks the clients that are concurrently accessing a file. Clients request to a manager the ability to write / read a file block.

The manager invalidates file blocks when they are written to, and the clients acknowledge that they have invalidated their copies of the files. Then the write privilege is given to the writing client for the block - it's a token. This token can be revoked by the manager at any time.

## cooperative caching

### Cache Coherence

- Single writer, multiple readers
- file block unit of coherence



If a client has a copy of the file, we can leverage **cooperative caching** - instead of going to disk to get the file a client with the file in its cache will serve a read request.

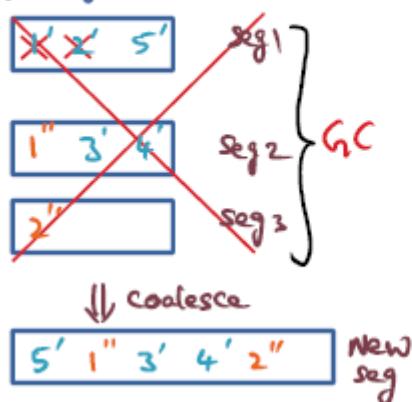
## Log cleaning

High-level representation shows how log maintenance is conducted, killing old log segments. It also shows how logs are aggregated and then the old log segments are garbage collected.

This isn't done by a single manager, it's actually done as a distributed activity. In XFS, the clients are also responsible for log cleaning - there is no differentiation between clients and servers in XFS. The clients are responsible for knowing the utilization of log segments for aggregation.

## Log cleaning

### Log Segment evolution



writes to file blocks 1, 2, 5  
—may belong to different files

block 1 overwritten  $\Rightarrow$  kill old block  
writes to blocks 3 + 4

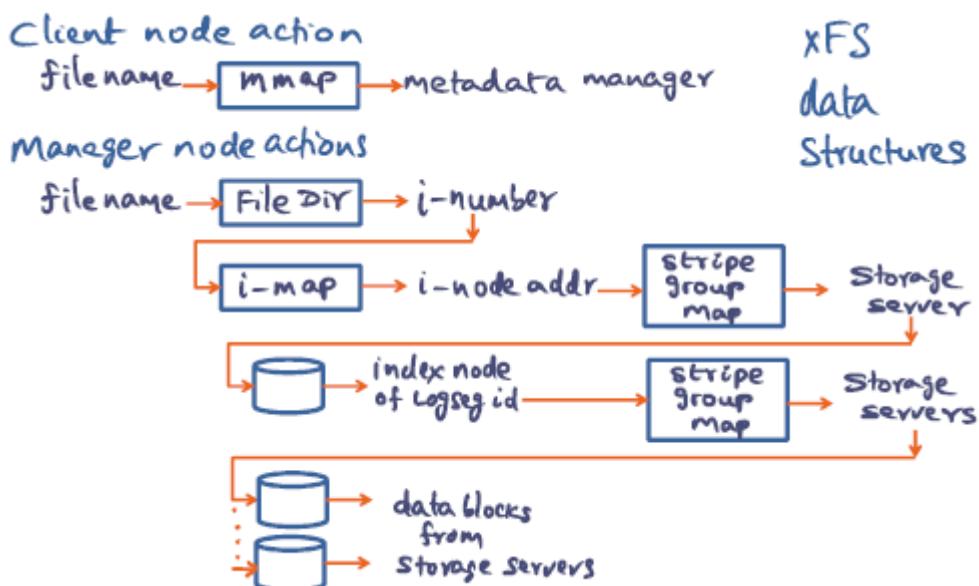
block 2 overwritten  $\Rightarrow$  kill old block

Aggregate all "live" blocks into  
new segment

## XFS data structures

A lot more data structures are involved in XFS to manage files across the distributed file system.

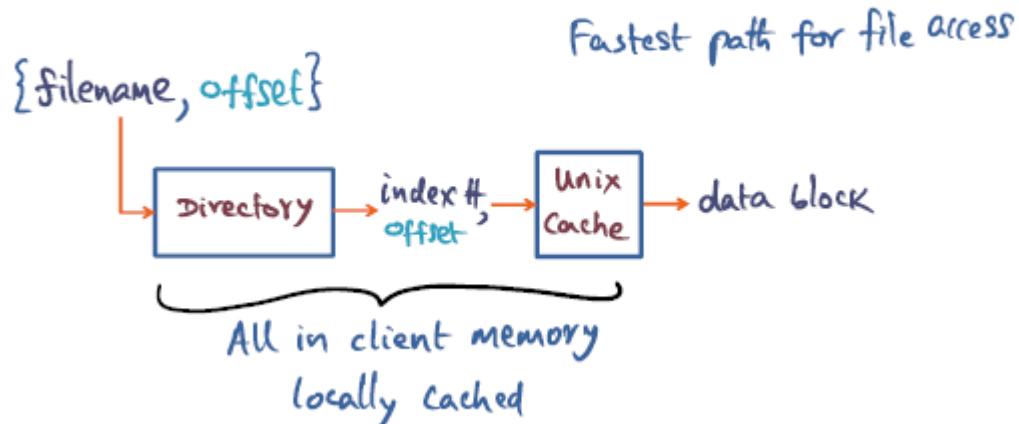
Below is a high-level representation of the XFS data structures used to conduct file access.



## Client reading a file

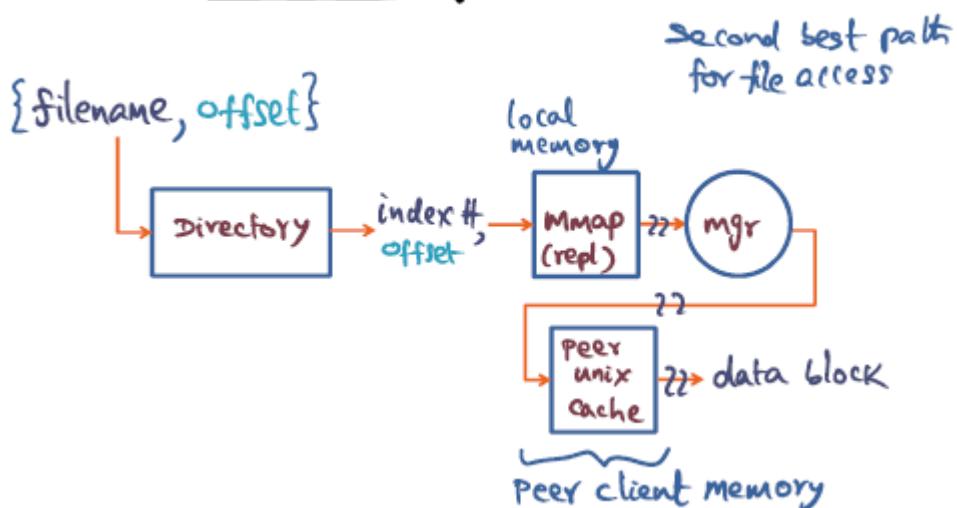
Fastest path for file access for a client. The client will read the file from its local cache.

## Client Reading a file – Own Cache



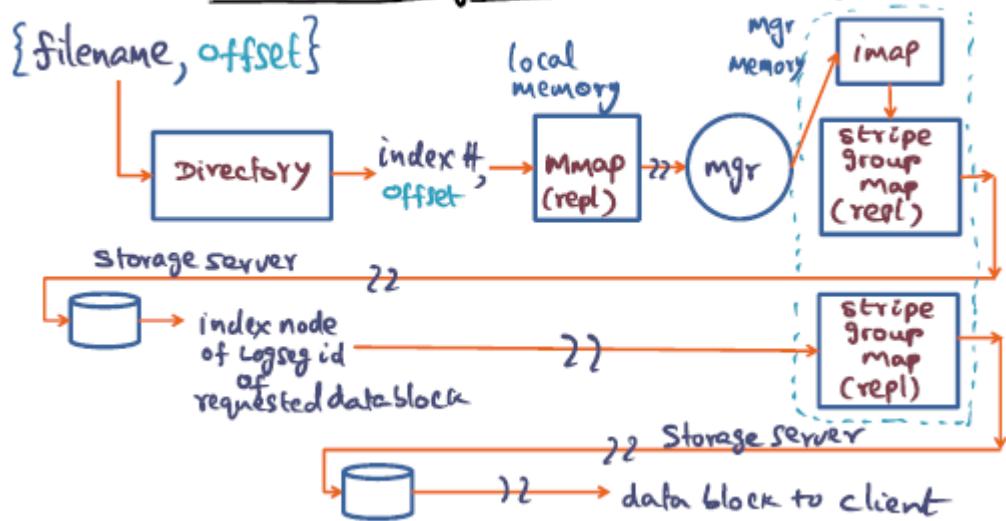
Second fastest path is to read the file from a peer who has the file cached. The local manager map will tell us who the manager node is for the particular file that I want to read. The manager node tells us who has the file cached, and then I read the file from the cache of the peer across the network.

## Client Reading a file – get from peer cache



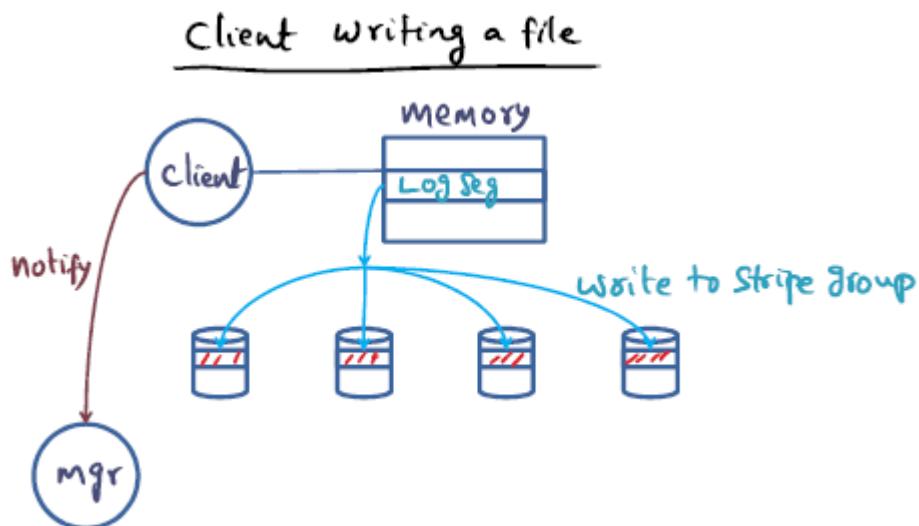
Longest method of acquiring a file. Manager map points me to manager node, manager node says it's not located in anyone's cache, the manager node looks up the file with the imap, finds the stripe group, gets the logging ID of the file block, gets the stripe group map again, and then we finally read the file block from the storage server. Once we finally acquire the file block, we then cache the file locally.

## Client Reading a file – the real long way!



## Client writing files

The client writes to the file block in its local memory, knows the log segment that the file block belongs to and then flushes the log segment to the stripe group. The client then notifies the manager about the log segments that have been flushed to the stripe group.



lesson9

# lesson9

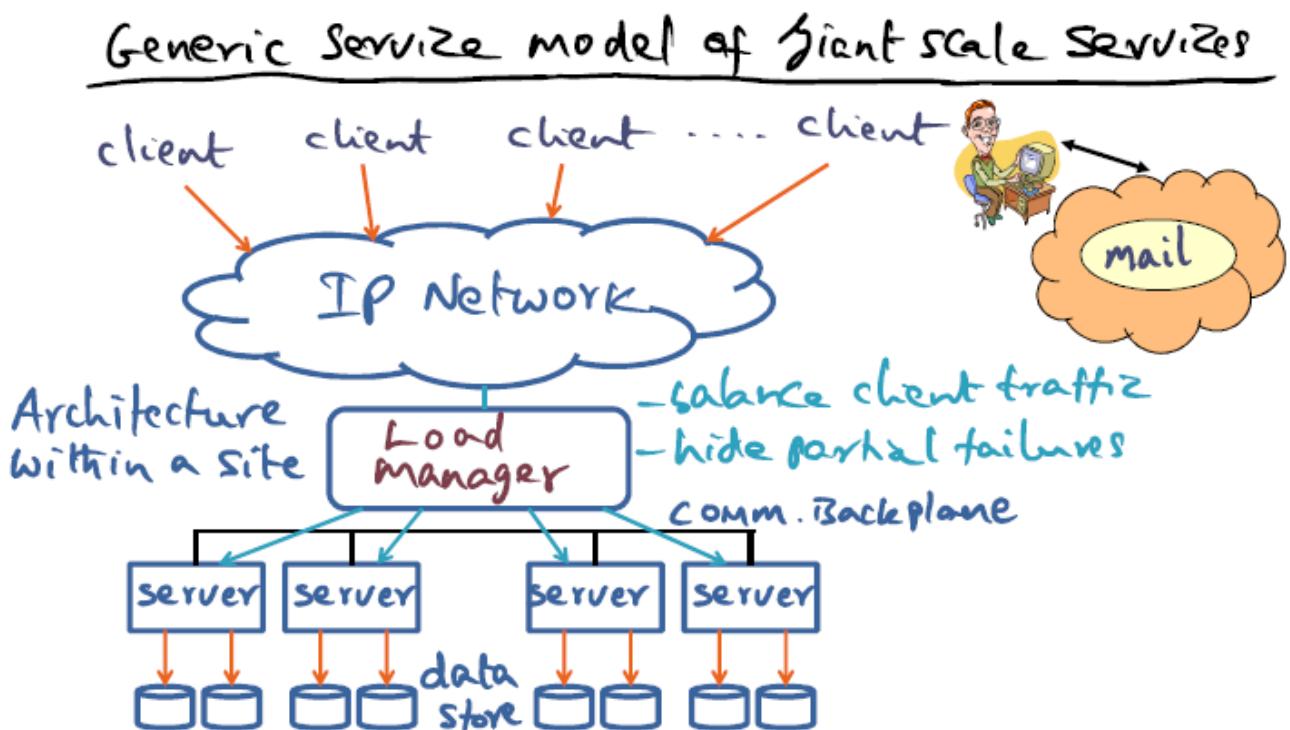
# Giant scale services

## Generic service model of giant scale services

Below is a high level representation of the architecture used to implement giant scale services. These types of services use multiple services and data stores to satisfy a massive number of client requests. Architectures within these sites use a very important mechanism to maintain quality of service, a **load manager**.

A **load manager** completes these two functions for giant scale services:

- Balances client traffic - make sure no particular server is overloaded.
- Hides partial failures - due to the law of large numbers, a node within the architecture **will** eventually fail.

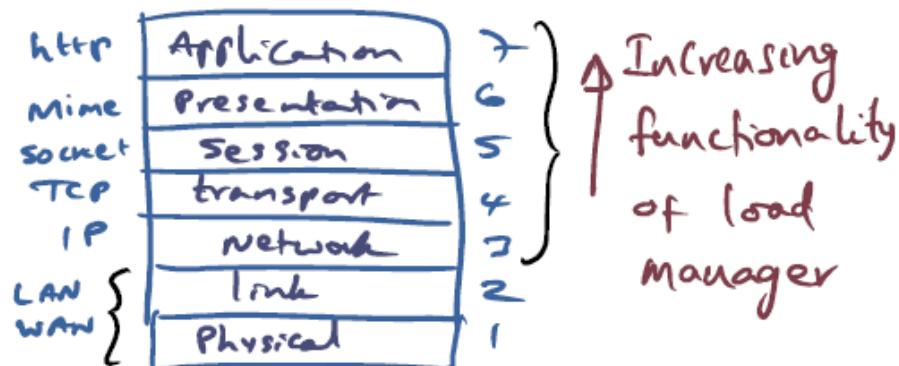


## Load management choices

The high-level representation of the OSI stack below is just describing the increasing functionality of the load manager as we travel up the stack. This is because applications and protocols become more complex, and we can make smarter load balancing decisions.

## Load management choices

### OSI reference model



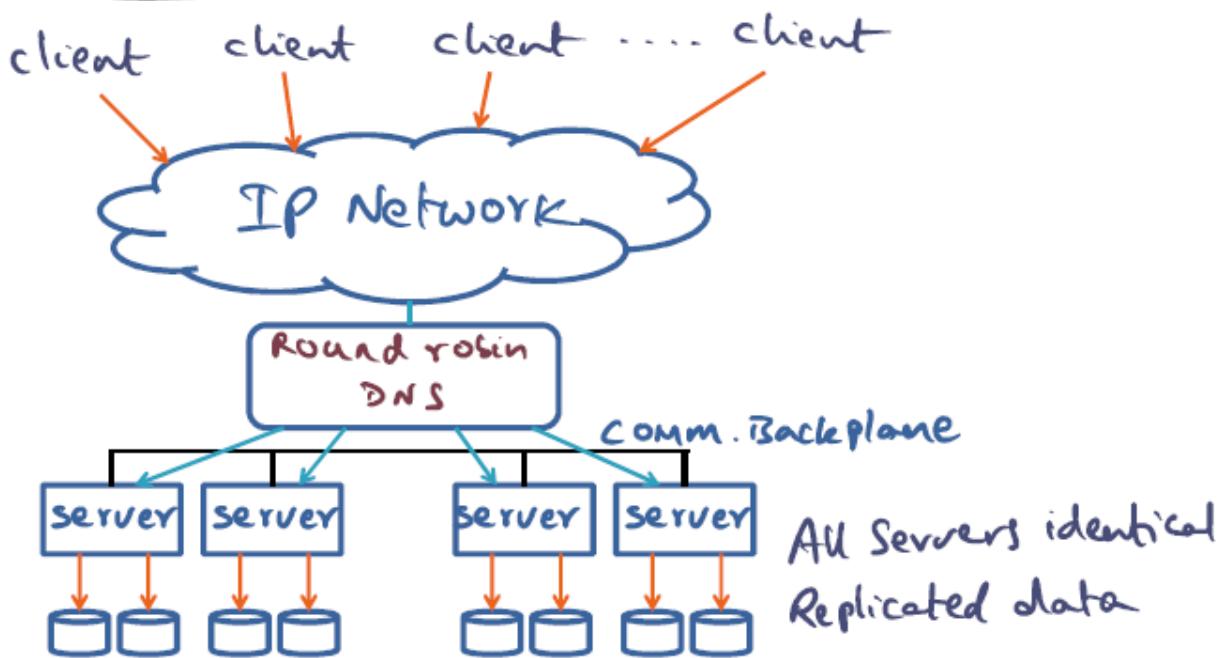
### Load management at the network level

If the load balancer is operating at the network level, the load balancer essentially acts as a **DNS** server. All requests for a particular host name are being balanced in a **round-robin** manner, each client being provided the host name for a domain name in a sequential order.

To achieve this type of load balancing, all servers must be identical and replicate the data that they serve. This way, clients notice no difference when provided a different hostname for a resolved domain.

The main problem with load balancing at the network level is the load balancer is not able to hide downed server nodes.

## Load management at Network level

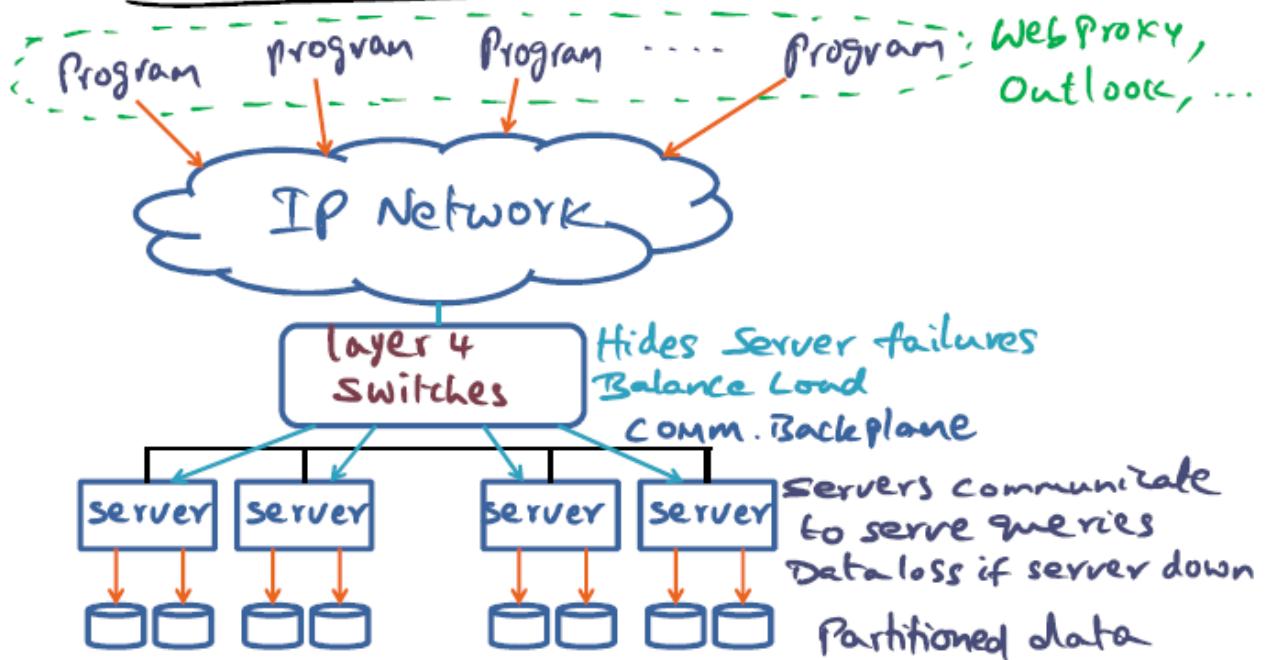


Now we look at having the load balancer at a higher level of the OSI model. In this architecture, we are able to hide server failures from clients as we are able to isolate downed server nodes. A load balancer higher than the network layer could be implemented with layer 4 switches.

This architecture also allows servers to implement server-specific front-end nodes. Instead of dealing with arbitrary client requests, we are able to deal with specific kinds of requests. Moving the load balancing to a higher level also provides us the ability to inspect the type of device making a request, allowing us to make a decision to point it to a server node servicing that particular device type.

In this architecture, we show a partitioned data scheme - if a server is unable to serve a client request it can use the network backbone to find a server that can. We face data loss, however, if a server is down; this architecture won't be able to provide the data the client needs from the downed server.

## Load management at transport level or higher

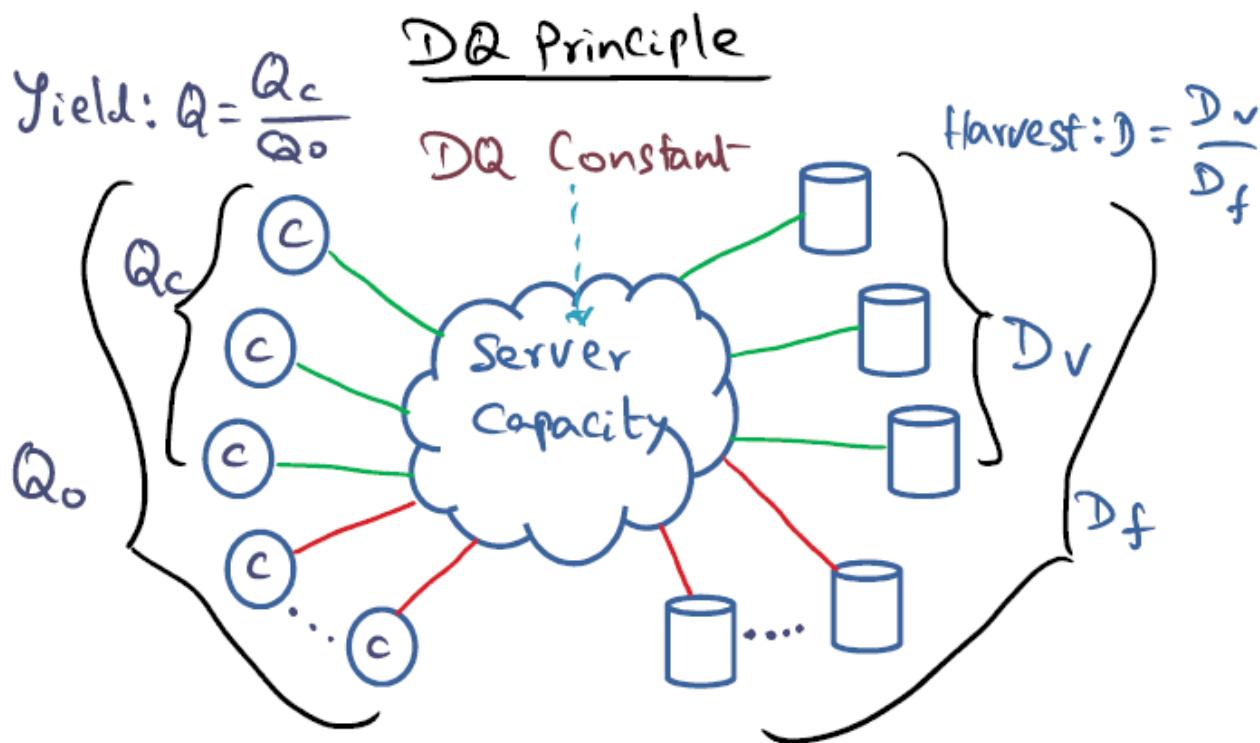


## DQ principle

Below is a high-level representation of the DQ principle. It shows the definitions for some key values:

Value	Definition
yield: Q	number of client requests completed / number of client requests
harvest: D	amount of data available / total amount of data
DQ principle	data per query * queries per second -> constant

These values directly compete with one another. If you increase the yield, you must decrease the harvest and vice versa.



## Replication vs partitioning

Replication maintains 100 percent harvest under a fault. In contrast, partitioned data stores will lose a percentage of their harvest under a fault. The opposite is true if we study the yield during a fault. Replicated data stores will lose a percentage of their yield while partitioned data stores will maintain 100 percent yield.

"" The traditional view of replication silently assumes that there is enough excess capacity to prevent faults from affecting yield. We refer to this as the load redirection problem because under faults, the remaining replicas have to handle the queries formerly handled by the failed nodes. Under high utilization, this is unrealistic. ""

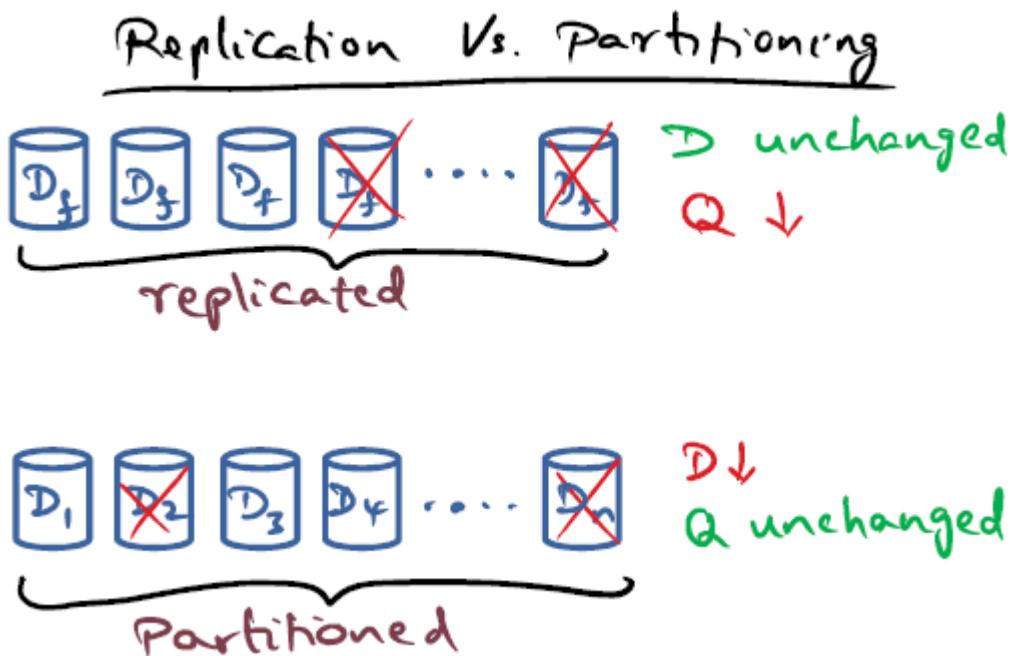
"" "[ ... ] you should always use replicas above some specified throughput. ""

"" You will enjoy more control over harvest and support for disaster recovery, and it is easier to grow systems via replication than by repartitioning onto more nodes. ""

"" Finally, we can exploit randomization to make our lost harvest a random subset of the data, (as well as to avoid hot spots in partitions). Many of the load-balancing switches can use a (pseudorandom) hash function to partition the data, for example. This makes the

average and worst-case losses the same because we lose a random subset of "average" value.""""

- Lessons from Giant-Scale Services



## Graceful degradation

""" [ ... ] graceful degradation: We can either limit Q (capacity) to maintain D, or we can reduce D and increase Q. """

""" We can focus on harvest through admission control (AC), which reduces Q, or on yield through dynamic database reduction, which reduces D, or we can use a combination of the two. """

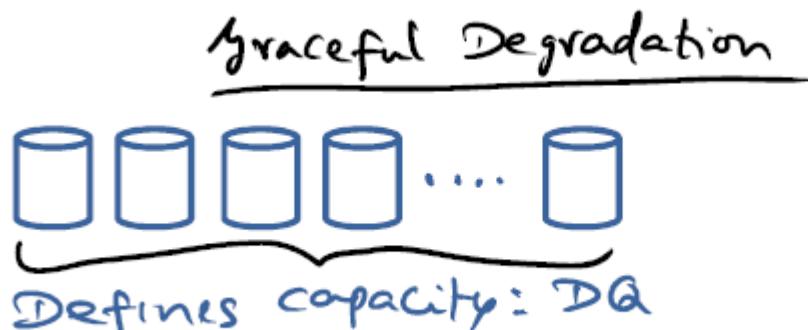
""" The larger insight is that graceful degradation is simply the explicit process for managing the effect of saturation on availability; that is, we explicitly decide how saturation should affect uptime, harvest, and quality of service. Here are some more sophisticated examples taken from real systems: """

Example	Description

cost-based admission control (AC)	perform AC based on the estimated query cost - reduces the average data required per query, increasing Q
priority or value-based AC	user pays commission to guarantee their query will be executed; reduce the required DQ by dropping low-value (unpaid for) queries independent of their DQ cost
reduced data freshness	under saturation, a site can make data expire less frequently - reduces freshness but also reduces work, thus increasing yield at expense of harvest

"" As you can see, the DQ principle can be used as a tool for designing how saturation affects our availability metrics. We first decide which metrics to preserve (or at least focus on), and then we use sophisticated AC to limit Q and possibly reduce the average D. We also use aggressive caching and database reduction to reduce D and thus increase capacity. """

- Lessons from Giant-Scale Services



How do we deal with server saturation

DQ constant



- D fixed; Q ↓  
OR

- D ↓; Q fixed

## Online evolution and growth

"""" [ ... ] we must plan for continuous growth and frequent functionality updates. """"

"""" By viewing online evolution as a temporary, controlled reduction in DQ value, we can act to minimize the impact on our availability metrics. """"

$$\Delta DQ = n * u * averageDQ/node = DQ * u$$

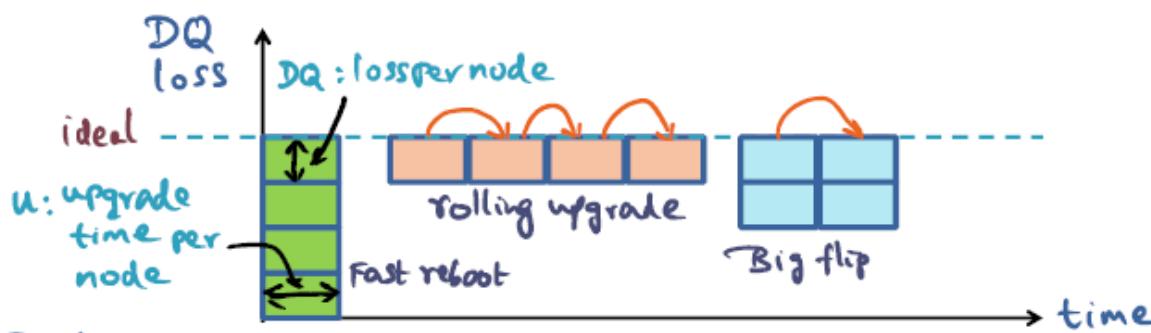
Variable	Description
u	fixed amount of time per node
n	nodes

Three main approaches are used for online evolution:

Approach	Description
fast reboot	quickly reboot all nodes into the new version; guarantees some downtime; by upgrading during off-peak hours, we can reduce the yield impact
rolling upgrade	we upgrade nodes one at a time in a “wave” that rolls through the cluster; old and new versions must be compatible because they will coexist
big flip	update the cluster one half at a time by taking down and upgrading half the nodes at once;

"""" The three regions have the same area, but the DQ loss is spread over time differently. All three approaches are used in practice, but rolling upgrades are easily the most popular. The heavyweight big flip is reserved for more complex changes and is used only rarely. The approaches differ in their handling of the DQ loss and in how they deal with staging areas and version compatibility, but all three benefit from DQ analysis and explicit management of the impact on availability metrics. """"

## Online Evolution and growth



**Fast**  
- off peak (e.g., use diurnal Server Property)

**Rolling**  
- wave upgrade

**Big Flip**  
- Half the nodes at once

DQ loss same for all three strategies  
 $= DQ * u * n$

- Lessons from Giant-Scale Services

## Definitions

Term	Definition
clients	web browsers, e-mail readers, etc
network	public internet, or a private intranet
load manager	a level of indirection between the service's external name and the servers' physical names
servers	system's workers, combining CPU, memory, and disks into replicable units
persistent data store	replicated or partitioned database that is spread across server disks
backplane	system-area-network handling inter-server traffic; redirection of queries to correct servers, coherence traffic for data stores

---

mtbf mean-time-between-failure

mttr mean-time-to-repair

uptime  $(\text{mtbf} - \text{mttr}) / \text{mtbf}$

yield (Q) queries completed / queries offered

harvest (D) data available / complete data

---

## Quizzes

Check off the ones you see below as "giant scale" services.

- Online airline reservation
- Purchasing a laptop online
- Web mail
- Google search
- Streaming a movie from Netflix

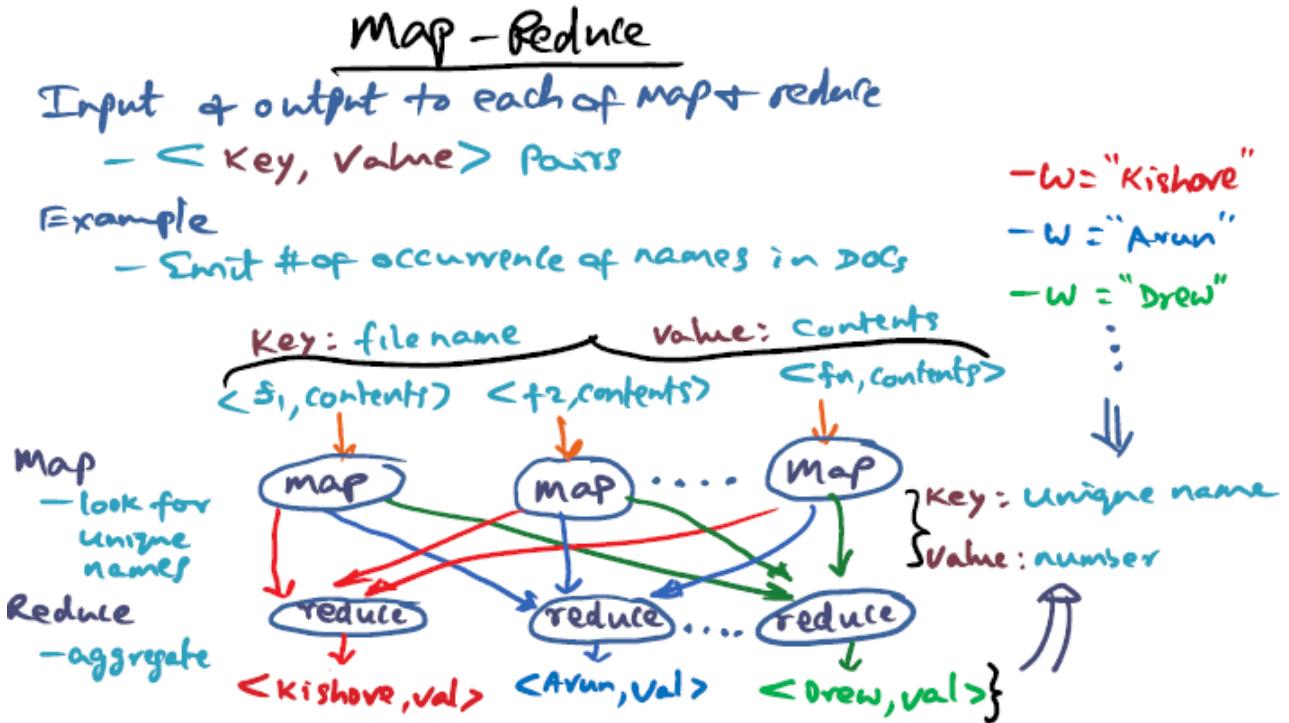
# MapReduce

## MapReduce

"" The computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: Map and Reduce. "" - Simplified Data Processing on Large Clusters

Method	Description
map	written by the user, takes an input pair and produces a set of intermediate key/value pairs.
reduce	accepts an intermediate key and a set of values for that key. It merges together these values to form a possibly smaller set of values.

Below is a high-level representation of the MapReduce computation.



## Why MapReduce?

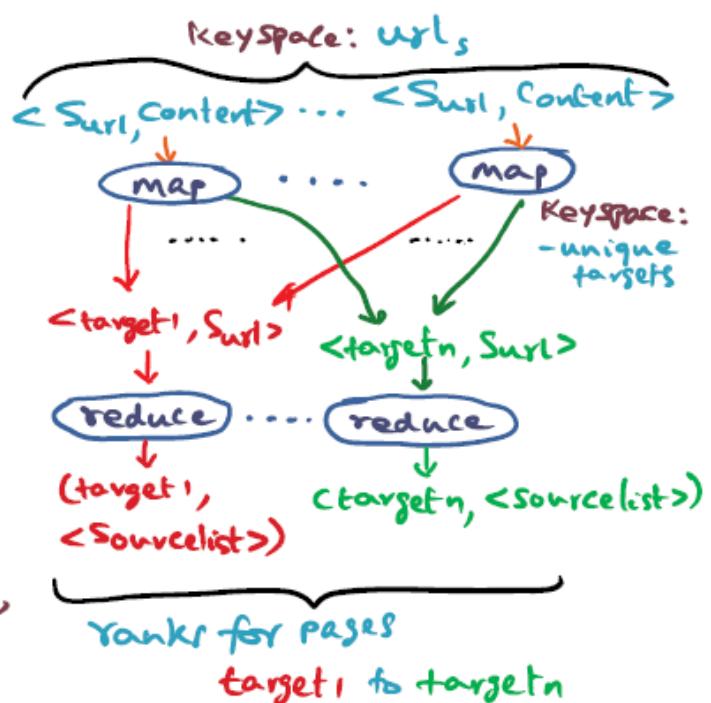
MapReduce is applicable to a lot of real-world problems. Some examples:

- Distributed grep
- Count of URL access frequency
- Reverse web-link graph
- Term-vector per host
- Inverted index
- Distributed sort

Below is a high-level representation that uses MapReduce to search for the occurrence of a **targetUrl** among a group of **sourceUrls**.

## Why Map-Reduce?

- Several processing steps in giant-scale services expressible
- Domain expert writes
  - \* Map
  - \* reduce
- runtime does the rest
  - \* instantiating number of mappers, reducers
  - \* data movement



## Heavy lifting done by the runtime

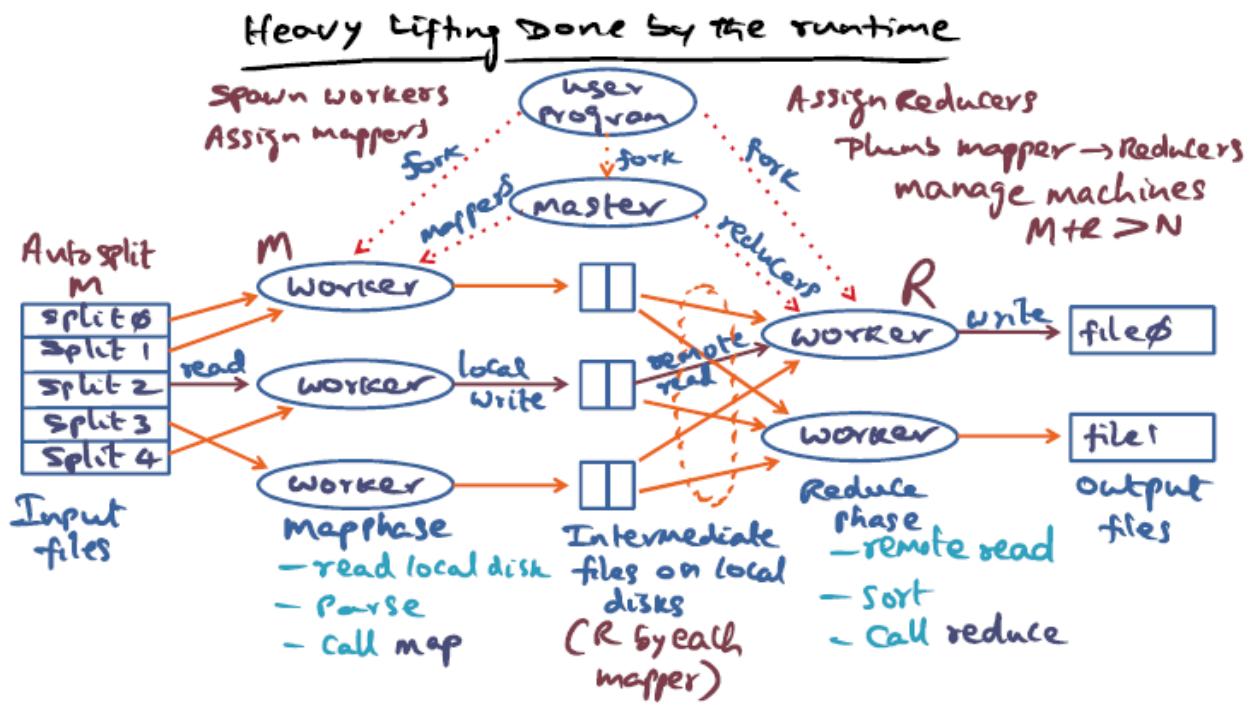
"" The Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of M splits. The input splits can be processed in parallel by different machines. "" - Simplified Data Processing on Large Clusters

"" Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g.,  $\text{hash}(\text{key}) \bmod R$ ). The number of partitions (R)

and the partitioning function are specified by the user. "" - Simplified Data Processing on Large Clusters

The following actions occur when the user program calls the **MapReduce** function:

1. **MapReduce** library splits the input files into M pieces. The **MapReduce** library starts up many copies of the program on a cluster of machines.
2. One of the copies of the program is special - the master. The rest of the copies are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign.
3. A worker assigned a Map task reads the contents of the input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.
4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.
5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.
6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.
7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.



## Issues to be handled by the runtime

"" The master keeps several data structures. For each map task and reduce task, it stores the state (idle, in-progress, or completed), and the identity of the worker machine (for non-idle tasks). "" - Simplified Data Processing on Large Clusters

### Fault tolerance

"" The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial idle state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to idle and becomes eligible for rescheduling. "" - Simplified Data Processing on Large Clusters

"" We rely on atomic commits of map and reduce task outputs to achieve this property. Each in-progress task writes its output to private temporary files. A reduce task produces one such file, and a map task produces  $R$  such files (one per reduce task). When a map task completes, the worker sends a message to the master and includes the names of the  $R$  temporary files in the message. If the master receives a completion message for an already completed map task, it ignores the message. Otherwise, it records the names of  $R$  files in a master data structure. "" - Simplified Data Processing on Large Clusters

## **Locality**

"" We conserve network bandwidth by taking advantage of the fact that the input data is stored on the local disks of the machines that make up our cluster. "" - Simplified Data Processing on Large Clusters

## **Task granularity**

"" We subdivide the map phase into M pieces and the reduce phase into R pieces, as described above. Ideally, M and R should be much larger than the number of worker machines. Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails: the many map tasks it has completed can be spread out across all the other worker machines. "" - Simplified Data Processing on Large Clusters

## **Backup tasks**

"" We have a general mechanism to alleviate the problem of stragglers. When a MapReduce operation is close to completion, the master schedules backup executions of the remaining in-progress tasks. The task is marked as completed whenever either the primary or the backup execution completes. "" - Simplified Data Processing on Large Clusters

# Content distribution networks

## DHT

Distributed hash tables (DHT)s are used to store the location of content on the internet. DHTs are key, value pairs. Here's are the objects that comprise a DHT:

Object	Definition
content hash (key)	the content hash is a numerical sum of the content to be hosted within the content distribution network; a hashing algorithm is used to ensure content hashes don't collide
value	the value of the DHT is the node ID where the content is stored within the content distribution network

So where are the key, value pairs stored for pairs listed in a DHT? Key, value pairs are stored on nodes that have a name closest to the content hash. Clients will query for the content, receive a response from the node that is hosting the key, value pair for the content, and provide the client with the **value** - this points the client to the node that hosts the content described in the **key**.

Below is a high-level representation of the concepts discussed above.

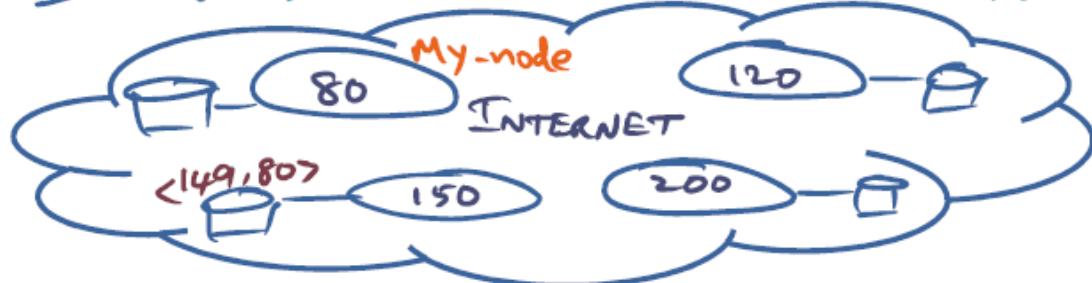
## DHT

### Content Distribution Networks

$\langle \text{Key}, \text{value} \rangle$

Content hash ↑ node-id where content stored  
content-hash = 149  $\Rightarrow \langle 149, 80 \rangle \Rightarrow$  Where to store?

DHT: Key ≈ node-id for storing  $\langle \text{key}, \text{value} \rangle$



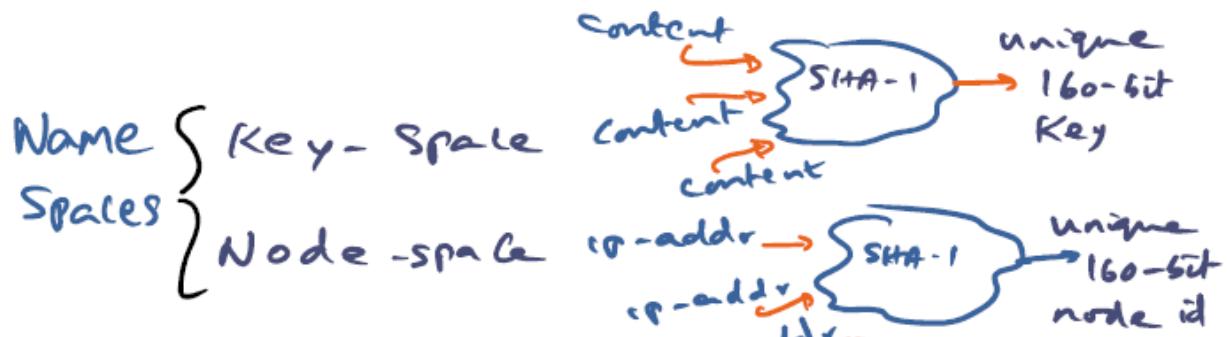
### DHT details

For a distributed hash table, two namespaces are used: **key\_space** and **node\_space**. They keys in each space are hashes of the real names of the content and the node, and each hashing algorithm provides a unique hash.

The objective is for the content key to be as close as possible to the node ID, with a best case scenario of the content key and the node ID to be an exact match. APIs are available to the programmer to directly interact with the DHT, using **putkey**, and **getkey** procedure calls.

Below is a high-level representation of the concepts discussed above.

## DHT Details



Objective :

$$\langle \text{Key} \rangle \rightarrow \text{nodeid} \langle N \rangle$$

such that  $\langle \text{Key} \rangle \approx \langle N \rangle$

APIs

— Putkey, getkey

## CDN (an overlay network)

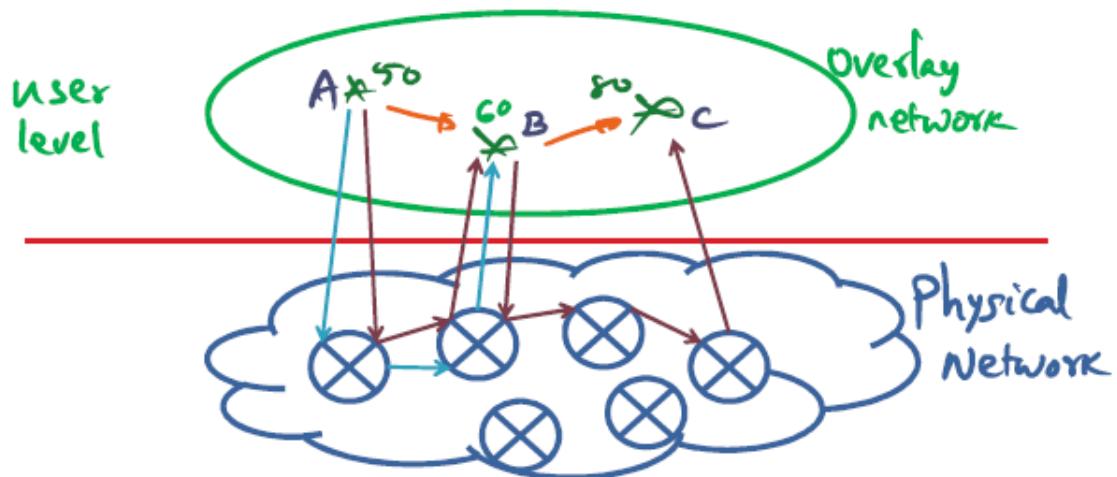
An **overlay network** is a routing table at the user level, implemented in a shared application running on multiple nodes - virtual network on top of the physical network. Each node within the content distribution network advertises their node ID, and maps this to their IP address. Each server will be able to address a particular node within the CDN using its node ID  $\rightarrow$  IP address mapping.

Below is a high-level representation of how nodes can route messages to one another using a CDN routing table.

## CDN - an overlay network

routing table at A  
(node id = 50)

name	node id	next hop
B	60	60
C	80	60



## Overlay networks in general

Below is a high-level representation of how addresses within Layer 2  $\rightarrow$  Layer 3  $\rightarrow$  Layer 7 are translated for the implementation of an overlay network.

### Overlay Networks in general

#### Examples of overlay

OS Level

- IP network is an overlay on LAN

IP Addr	Mac Addr

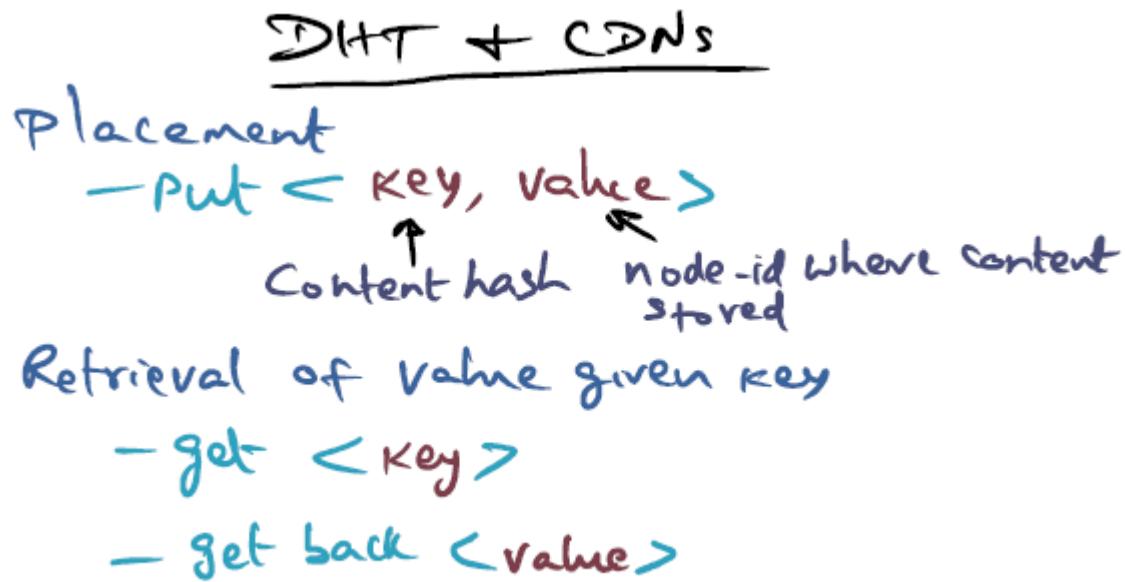
App Level

- CDN is an overlay on TCP/IP

Node ID	IP Addr

## DHT and CDNs

Below is a high-level representation of how content keys and their respective node values are stored in a DHT. This representation also shows how, with a given content key, you can receive the node ID for where the content is stored from the DHT.



## Traditional approach (greedy)

The reason we place key, value pairs on nodes **N** where **N** is close to **key** is because it makes routing to other nodes simpler. The routing tables per node in a CDN can only be so large so, if someone requests content for a **key**, because **keys** are stored on nodes close to the key's value, we can best-guess determine that the content is stored on node **N**.

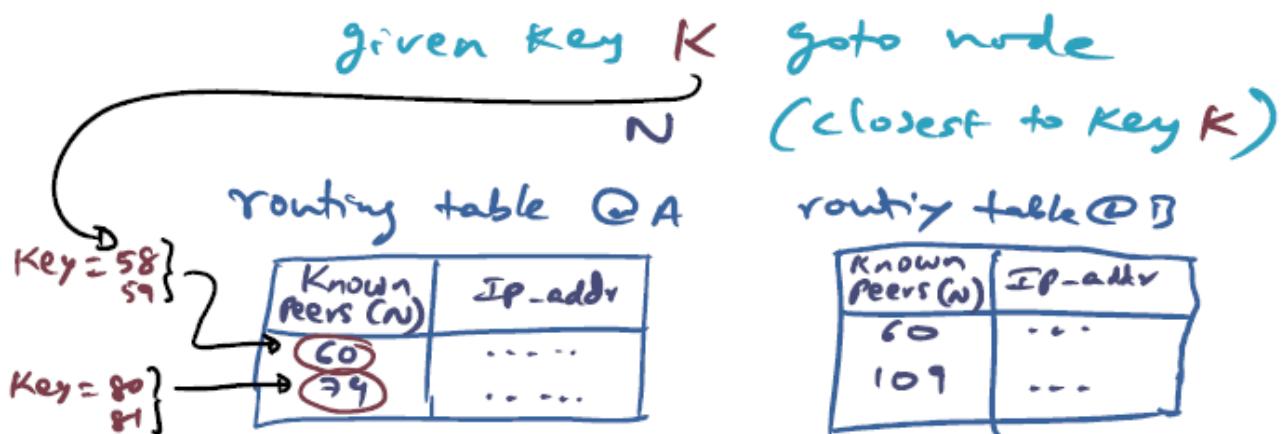
If a node doesn't know the IP address of the target key -> node, we'll eventually conduct enough routing where we'll finally find the node that contains the requested key, value and can provide us the IP address of the node.

Below is a high-level representation of the concepts discussed above, and probably clarifies the concepts I attempted to describe.

Traditional approach (greedy)

place  $\langle \text{key}, \text{value} \rangle$  in node  $N$ , where  $N \approx \text{key}$

retrieve :



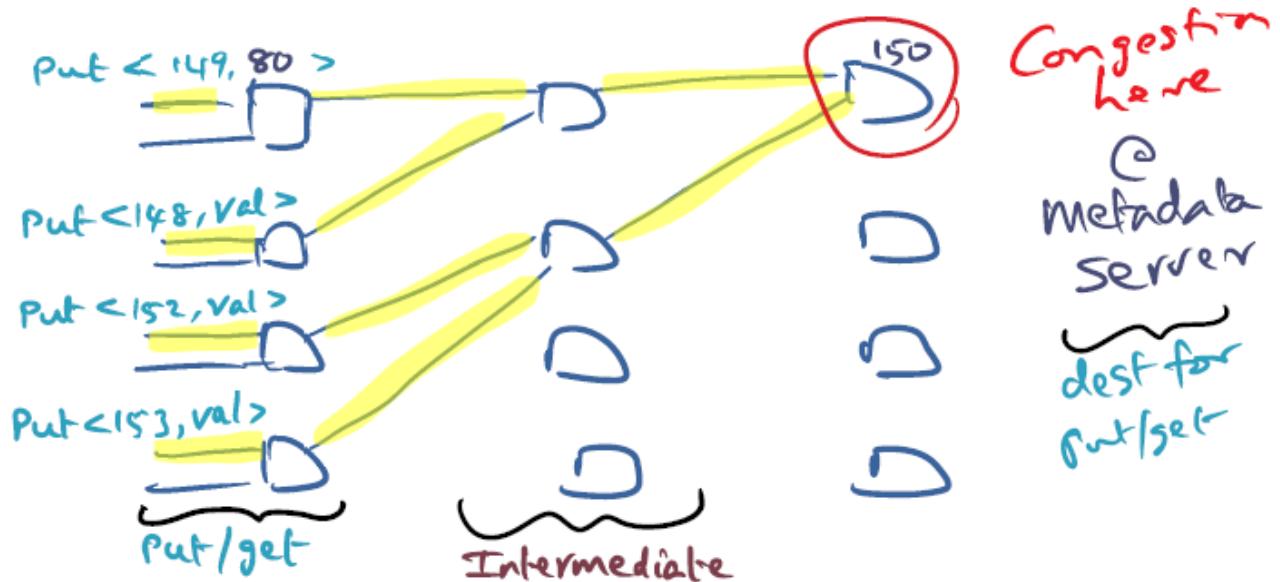
## Metadata overload

The traditional approach creates a scenario in which a particular server can become a hotspot for metadata as well as for content retrieval. Imagine some scenario where a large number of users are generating content and all of their keys are very similar. All of these keys will reside on one particular node because its node ID is closest to their keys. This server will become overloaded with metadata, and the storage of key, value pairs will be uneven across the content distribution network.

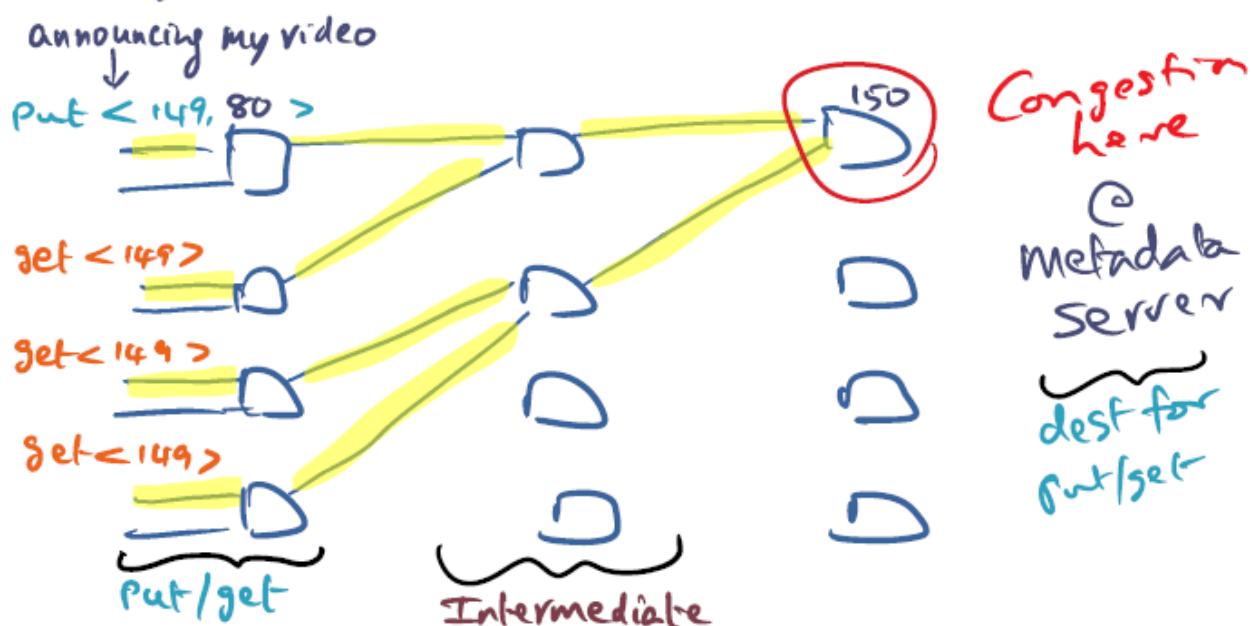
Imagine a scenario in which a particular piece of content becomes really popular and everyone is requesting that content. The node that contains the key, value pair for said content will become a hotspot, and will be busy routing clients requesting the content to the actual node storing the content. This server will become overloaded responding to content requests - responses will not be conducted in a distributed manner as only one server will be conducting work.

Below is a high-level representation of this problem.

Greedy Approach leads to Metadata server overload



Greedy Approach leads to Metadata server overload



## Origin server overload

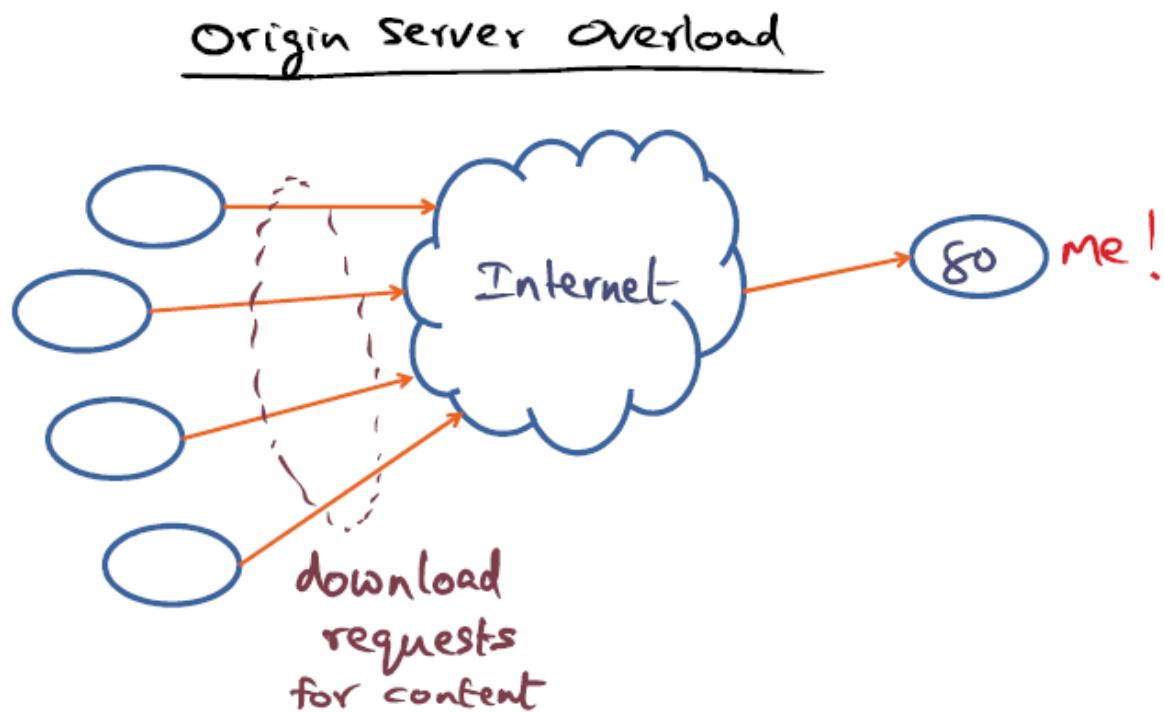
The origin server overload problem is the fact that one specific server hosting content will receive all the requests for a particular piece of content - essentially DOSsing the server.

This DOSsing by people attempting to view the content of a server is called the **slashdot** effect.

How do we combat the **slashdot** effect? There are two solutions:

- Utilize a webproxy to directly serve content after caching it for users close to the webproxy
  - Not good enough for the **slashdot** effect, especially if people want live content
- CDNs
  - content is mirrored and stored based upon geographic location
  - user requests are dynamically re-routed to geo-local mirrors
  - content producers pay CDN providers to avoid origin overload

Below is a high-level representation of origin server overload.



## Tree saturation

Tree saturation is what we described earlier, with keys matching particular nodes and nodes being overloaded with metadata. The tree is saturated as well because all requests for a particular key, value pair are happening only on specific parts of the tree the comprises the CDN.

Coral DHT, a framework / application that attempts to democratize content distribution, uses what's called a **sloppy DHT**. This **sloppy DHT** doesn't strictly adhere to the traditional rule where a key must match as closely as possible to its node ID.

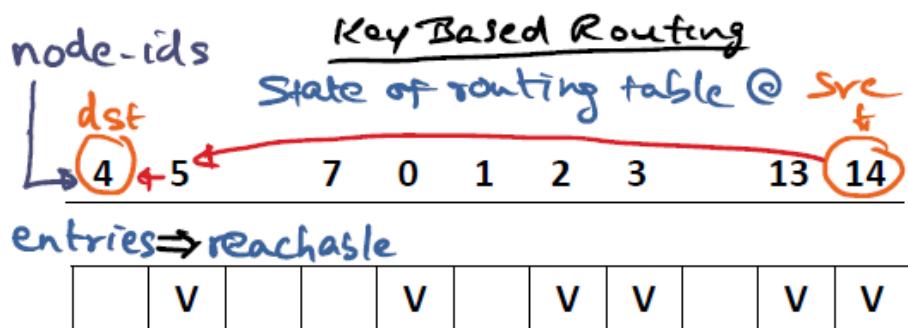
How does Coral implement a **sloppy DHT**?

- Coral DHT utilizes a novel key-based routing algorithm.
- The basis of the algorithm is based upon the distance between two nodes, calculated using xor:

$$(N_{src})XOR(N_{dst})$$

## Key-based routing

Below is a high-level representation of key-based routing use the **traditional (greedy) approach**.



Intuition in **greedy** approach

- get as close to the desired **dst** in node-id namespace
- he may know route to get me to desired **dst**

Objective?

- reach **dst** with fewest number of hops
- ⇒ "me first" approach leads to **congestion**

## Coral key-based routing

For Coral key-based routing, everything is based upon distance from the actual destination. A table is created in which each **node ID** is xor'd with the destination.

The difference between this approach and the greedy approach is that we don't immediately travel to the XOR distance that is closest to our destination. Travel is similar to a binary sort, in which, our 1st hop is half of our XOR distance, second is half of the previous calculation, and so on until we reach our destination.

Below is a high-level representation of how Coral key-based routing works.

### Coral Key-based routing

- Each hop go to some node that is

half the distance to dst in node-id namespace

dst	4	5	7	0	1	2	3	13	src	14
	1			4		6	7		9	10

node-id XOR 4

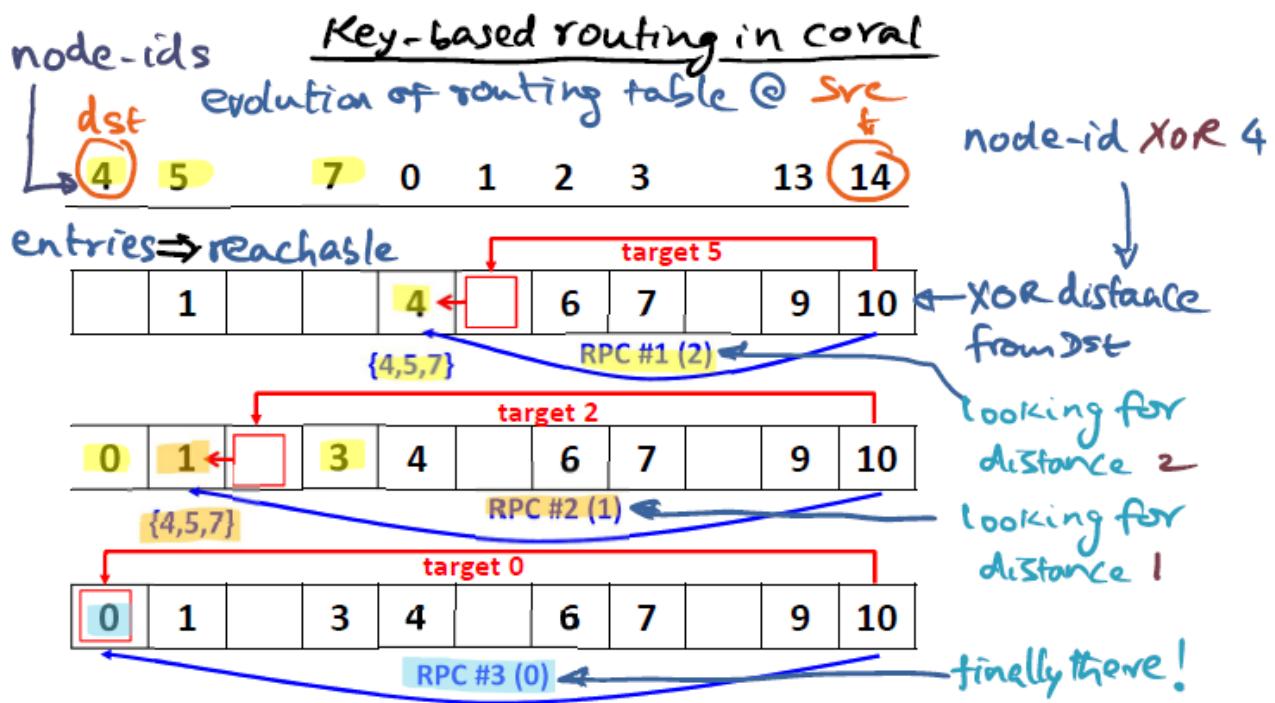
↓  
XOR distance  
from dst

XOR distance from src to dst = 10

1st hop go to  $\frac{10}{2} \Rightarrow 5$  distant

2nd hop go to  $\frac{5}{2} \Rightarrow 2$  distant

3rd hop go to  $\frac{2}{2} \Rightarrow 1$  distant  
... finally home !!



## Coral sloppy DHT

### Coral primitives

Primitive	Description
put (key, value)	<ul style="list-style-type: none"> <li><b>value</b> is the node ID of the proxy with content for the <b>key</b></li> <li><b>put</b> can be initiated by the origin server or by a proxy that has just download the content and is willing to serve as a proxy that will cache and serve the content</li> <li><b>put</b> places the key, value in an <b>appropriate node</b></li> </ul>
appropriate node	<ul style="list-style-type: none"> <li>an appropriate node has to be the node closest to the value of a <b>key</b> within a key, value pair</li> <li>If there is not an exact <b>key, node</b> match, we can determine the candidate node by looking at two parameters: <b>full</b> and <b>loaded</b></li> </ul>
full	the <b>full</b> primitive defines the nodes that are proxies that are able to store max I values for a <b>key</b> - metadata (spatial)

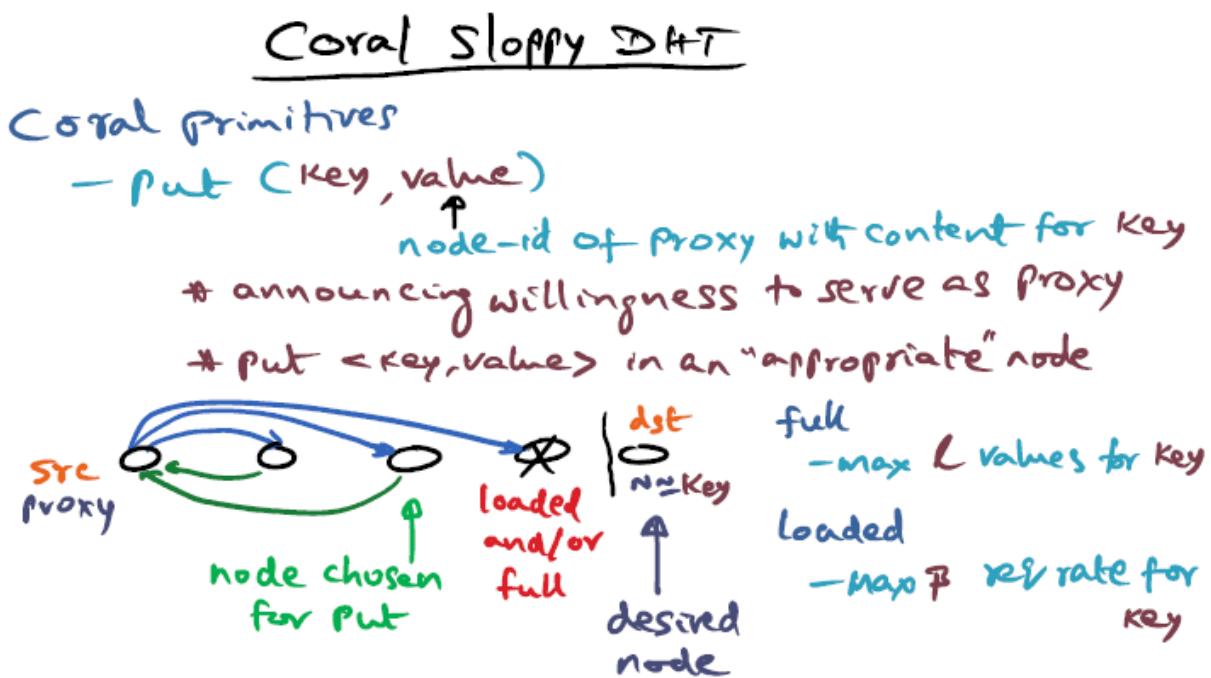
loaded

the **loaded** primitive defines the nodes that are proxies that are able to handle max **B** requests / time for a **key** - temporal limitation

The **put** operation is comprised of two phases, the forward phase and the backwards phase. In the forward phase, the source of the **put** operation is attempting to place the key, value pair in a particular node - some destination. The source will ask each node if it is **full** or **loaded**. If not, the source will increment half the distance to the next node, and continue this cycle until it finds a node that is **full** or **loaded**.

The source will then retract its steps to find a node that isn't **full** or **loaded** and will place its key, value pair there. This helps to distribute key, value pair metadata across the CDN.

Below is a high-level representation of this mechanism.



## Quizzes

Who started content distribution networks (CDN)s and why?

Who?

IBM

- Napster
- Facebook
- Netflix
- Akamai
- Apple
- Microsoft
- Google

Why?

- World news
- E-commerce
- Online education
- Music sharing
- Photo & video sharing

lesson10

# **lesson10**

# ts-linux

## Introduction

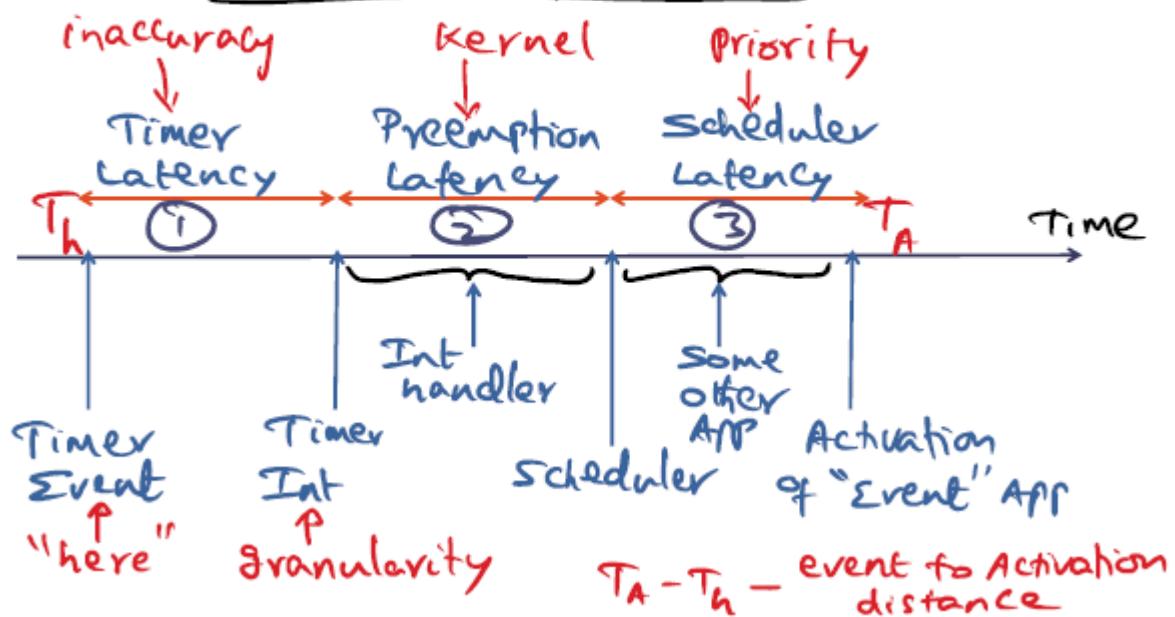
General purpose operating systems traditionally targeted throughput oriented applications (databases and scientific applications), however, applications like video games and audio / video players emerged needing soft-real-time guarantees - these applications are latency sensitive.

## Sources of latency

Latency source	Description
timer latency	generated from the inaccuracy of the timing mechanism; latency between the point an event happens and when the timer interrupts
preemption latency	interrupt occurs during an operation in which the kernel cannot be preempted
scheduler latency	latency incurred when the application that is destined to receive an interrupt cannot be dispatched until a higher priority application is suspended

Below is a high level representation of the sources of latency described above.

## Sources of Latency



## Timers available

timer	description
periodic timer	standard in most UNIX operating systems; operating system is interrupted at regular periods; incurs event recognition latency; max timer latency is equal to the period itself
one-shot	exact timers; can be programmed for an exact point of delivery; extra overhead for the operating system
soft	no timer interrupt; operating system polls at strategic times to check for events; extra polling overhead and latency
firm	combines the advantages of all the above three timers; proposed in ts-linux; attempts to avoid all cons from each timer type

Below is a high-level representation of the concepts described above.

choice of Timer	Pro	Con
Periodic	Periodicity	event recognition latency
One-shot	timely	overhead
Soft	reduced overhead	Polling overhead, latency
Firm	Combines all of the above	

## Firm timer design

The fundamental idea behind firm timer design is to provide accurate timing with low overhead. It attempts to combine the mechanisms of the **one-shot** and **soft** timers. Reminder, one-shot is a timer that occurs exact when we need it, however this incurs overhead. Similarly, the soft timer incurs overhead because the operating system must check for these events.

Firm timers implement an **overshoot** abstraction, a knob between **hard** and **soft** timers. The overshoot represents the time between a one-shot timer expiration and a programmed one-shot timer interrupt. During the **overshoot** time, if events occur that require kernel interaction, such as a system call, the kernel will also identify expired one-shot timers and, instead of waiting to dispatch them at a later time as programmed, will dispatch the timers while the kernel is handling privileged operations. This allows the kernel to preemptively handle one-shot timer's while conducting unrelated privileged operations.

The advantage of this is that one-shot timers handled early will not cause interrupts - saving us from the overhead incurred from the context switch that would've happened. The firm timer design utilizing overshoots provides us the timing accuracy desired while also avoiding the overhead of being interrupted by timers.

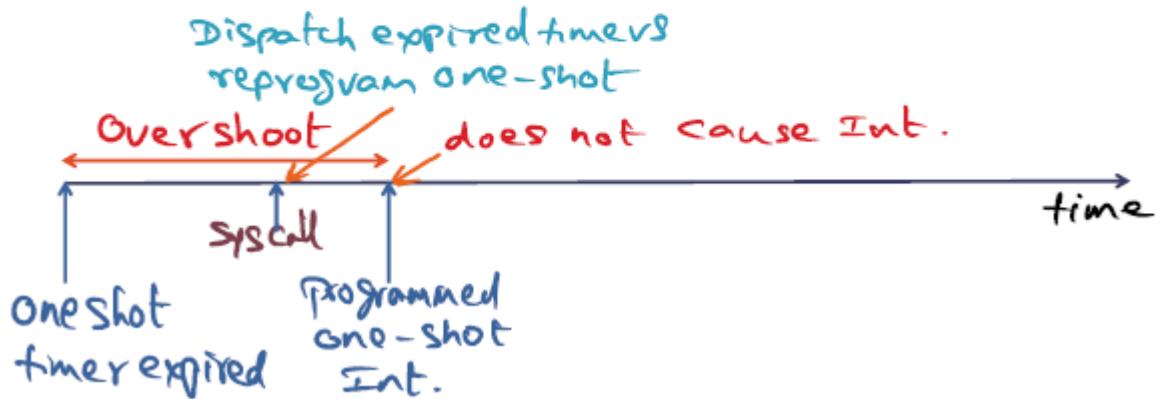
Below is a high level representation of the concepts discussed above.

## Firm Timer Design

Accurate timing with low overhead

— combine one-shot and soft timers

— **Overshoot**: knob between hard and soft timers



## Firm timer implementation

The firm timer design uses these components for implementation:

- Timer queue data structure - queue of timers sorted by expiry time
- APIC timer hardware - hardware utilized to reprogram timers in a short amount of cycles; decrements on each bus cycle
- Soft timers - eliminates needs for field one shot interrupts

## Firm Timer Implementation

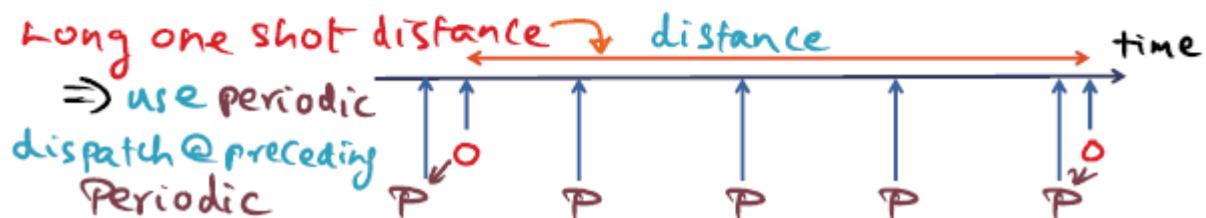
Timer-q data structure →  $T_1|10 \rightarrow T_2|12 \rightarrow T_3|15$   
 Sorted by expiry time

APIC timer hardware

- reprogramming: few cycles

Soft timers

- eliminates need for fielding oneshot int



## Reducing kernel preemption latency

There are multiple approaches in reducing the kernel preemption latency. Here are some of the approaches explained:

- Explicit insertion of preemption points in the kernel
- Allow preemption anytime the kernel is not manipulating shared data structures

The **lock-breaking pre-emptible kernel** combines the two ideas discussed above.

Acquisition of the lock, manipulation of shared data, and releasing of the lock, then preempt the kernel.

Below is a high level representation of the concepts discussed above.

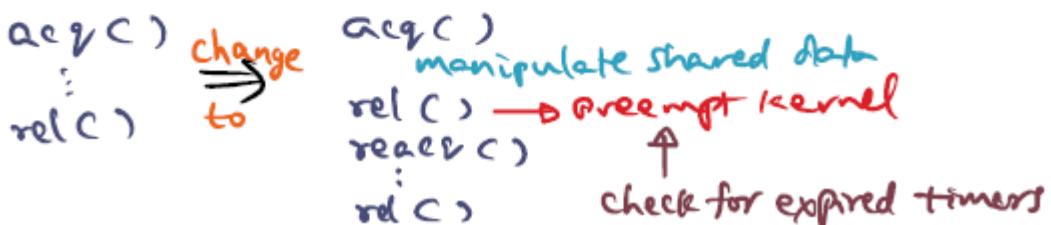
## Reducing Kernel Preemption Latency

### Approaches

- Explicit insertion of preemption points in kernel
- Allow preemption anytime kernel not manipulating shared data structures

### Lock-Breaking Preemptible Kernel

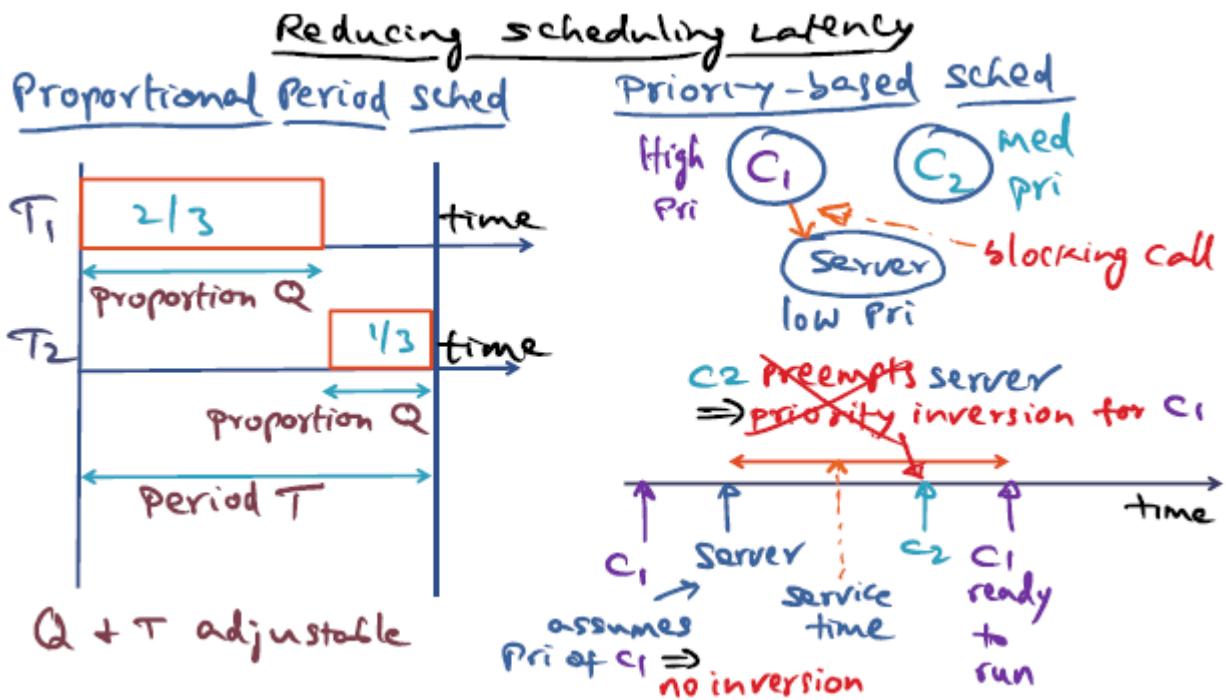
- Combines the above two ideas



Proportional based scheduling is described in TS-Linux, each task gets some portion of time to execute its task. There's also priority based scheduling that avoid priority inversion because, when a task calls a server process, the server will assume the priority of the callee in order to avoid being preempted.

TS-Linux avoids scheduling latency by using proportional period scheduling for admission control and avoid priority inversion through priority-based scheduling. It reserves time for time sensitive tasks and throughput sensitive tasks.

Below is a high level representation of the concepts discussed above.

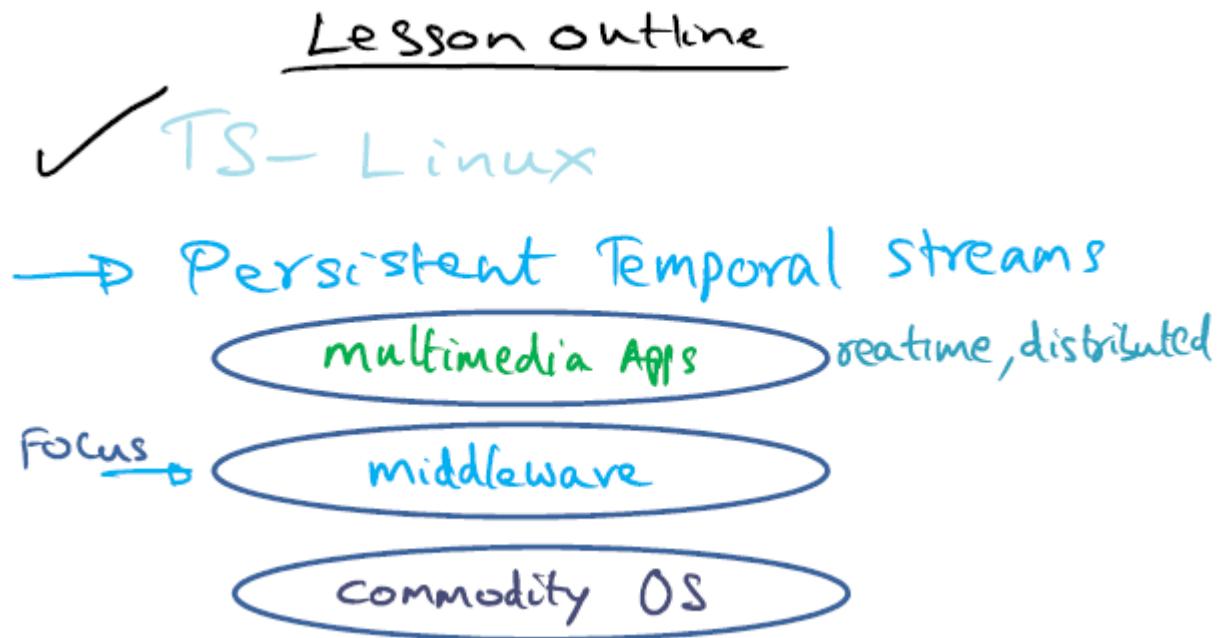


## Conclusion

TS-Linux is able to provide quality of service guarantees for real-time applications running on commodity operating systems such as Linux. Using admission control - proportional period scheduling - TS-Linux ensures that throughput-oriented tasks are not starved for CPU time.

# pts

This lesson covers the middleware that resides between commodity operating systems and novel multimedia applications that are realtime and distributed.



## Programming paradigms

This section covers the abstractions that we currently use for distributed applications:

- Parallel programs leverage the **pthreads** library / API for the implementation of parallel programs.
- Distributed programs leverage the **sockets** library / API for distributed program implementation.

Conventional distributed programs leverage the **sockets** library / API to communicate with distributed services, like a network file system, however this library doesn't have the level of abstraction needed for emerging novel multimedia distributed applications.

## Programming Paradigms

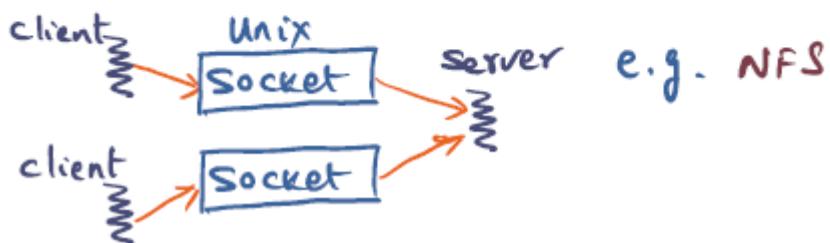
### Parallel program

- Pthreads: API for parallel programs

### Distributed Programs

- Sockets: API for distributed programs

### Conventional Distributed Program



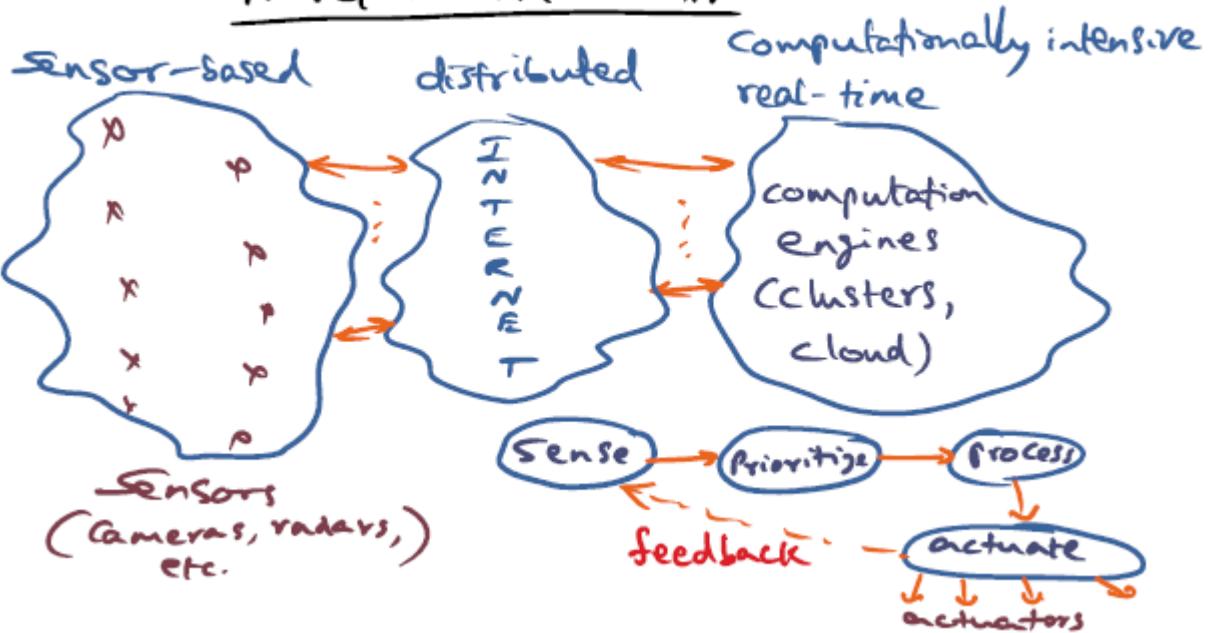
## Novel multimedia apps

An example of novel multimedia applications would be an application that includes distributes sensors. These sensors could be temperature sensors, cameras, barometric pressure sensors, etc. and they all produce data that is pushed to the internet.

An application can leverage this data and make decisions, prioritizing interesting data, processing it, and acting upon it - creating a feedback loop. This feedback loop, depending upon the data being sent from the sensors, can require computational intensive processing.

This generates the need for computational engines, or clusters.

## Novel Multimedia Apps



## Things to avoid

Given the example of a large-scale situational awareness multimedia application, there are some overheads we probably want to avoid in order to increase the efficiency and usability of the application:

Overhead	Description
infrastructure overhead	large amounts of data are being generated by sensors; filtering must be done to avoid overloading the infrastructure with unnecessary data
cognitive overhead	minimize the amount of human interaction necessary to interpret data being retrieved from the sensor framework
false positives and negatives	avoid generating flags or indicators of events that aren't actually happen, but may look like they have occurred due to the manner in which data is being processed, or due to the algorithm being used to generate flags for anomalous behavior

## Example programming model for situational awareness

This slide describes how a sequential program for video analysis would conduct detection, tracking, recognition, and alarming of specified images for a video stream. It raises the question: **how do we scale sequential video analysis and tracking for thousands of cameras?**

Persistent temporal streams (PTS) is a proposed solution to this problem, leveraging distributed programming solutions, and will be discussed in the rest of this lesson.

## Pts programming model

Two abstractions are provided by PTS, **threads** and **channels**. Threads create time-sequenced data objects and place these objects in the **channels**. For channels, there can be multiple producers and consumers of the data residing in **channels**.

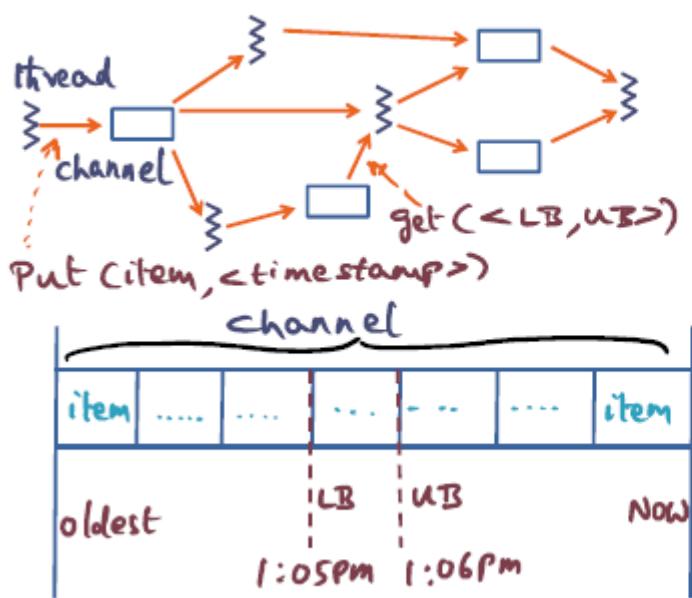
**Channels** basically represent the temporal evolution of a thread. Threads that wish to consume data from channels utilize an upper-bound and lower-bound timestamp - the channel returns the specified information stored.

Nutshell, PTS allows:

- The ability to associate timestamps with data items produced by a computation.
- Propagation of temporal causality.
- Allowing a computation to correlate incoming streams and derive temporal relations.

Below is a high-level representation of a how a program could leverage the **thread** and **channel** abstraction provided by PTS to compute and make decisions.

## PTS Programming Model



```

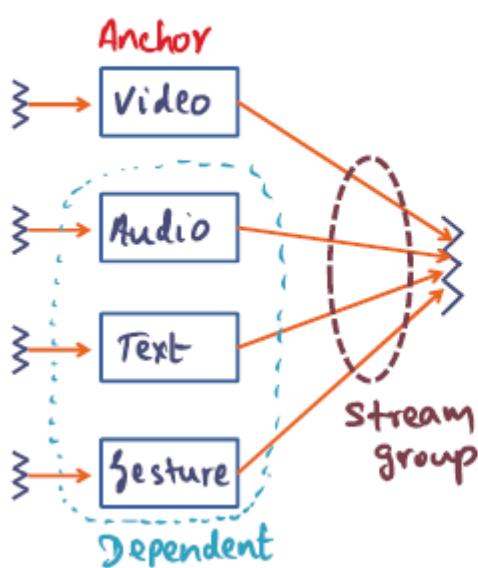
channel ch1 =
    lookup("Video channel");
while(1){
    //get data
    response r =
        ch1.get(<LB, UB>);
    //process data
    :
    //produce output
    ch2.put(item, <ts>);
}

```

## Bundling streams

PTS provides programmers the abstraction of the **groupget** mechanism, where we can bundle the acquisition of data from multiple streams at a required timestamp. This abstraction removes the requirement of having to call `get` on every single source of multimedia within a group. Programmers can identify specific streams as **anchor** streams, all the other streams are dependent streams to the **anchor**, and all timestamps will be in reference to the anchor.

## PTS: Bundling Streams



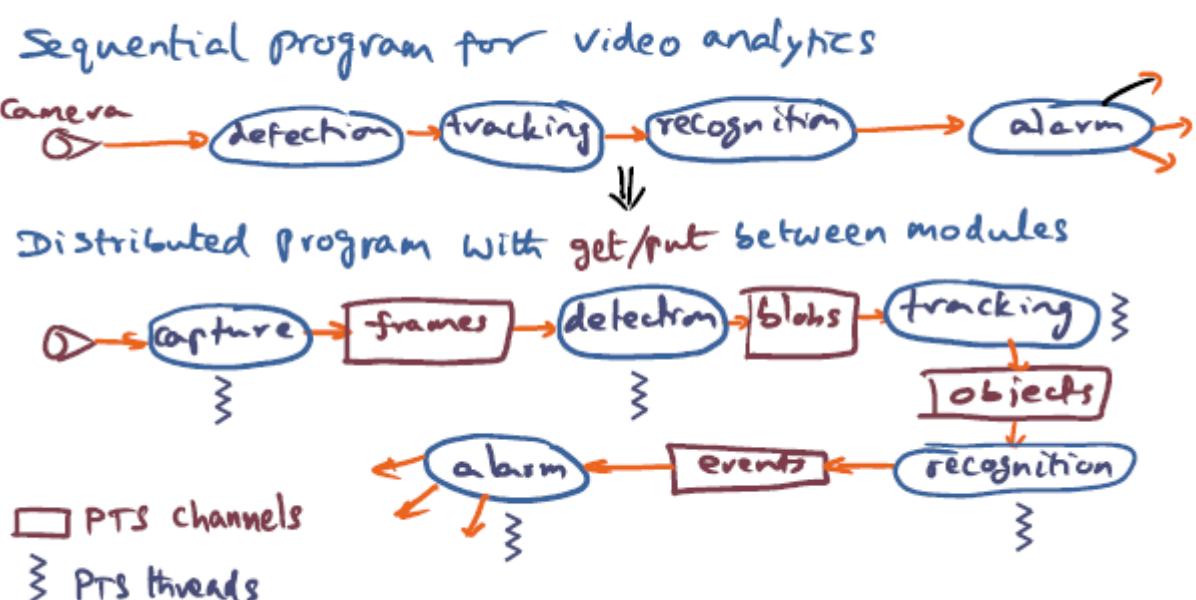
group get :

- get corresponding time-stamped items from all the streams in the group

## The power of simplicity

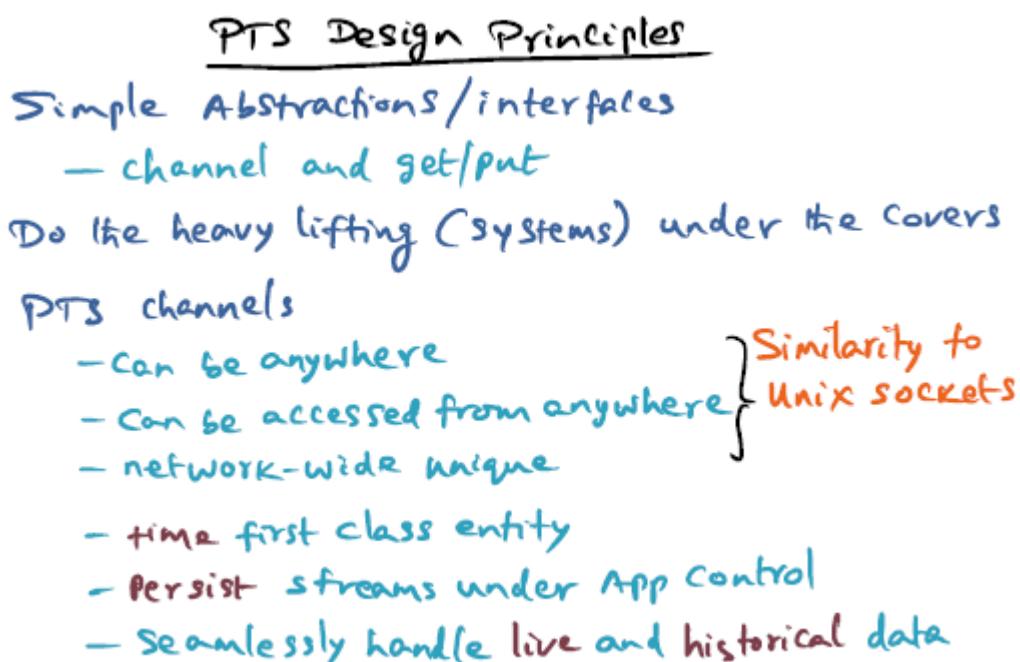
The below high level representation shows how PTS can simplify the sequential detection application that was described earlier using abstractions like **put** and **get**.

### Power of Simplicity



## Pts design principles

Below is a high level representation of the abstractions that PTS uses to make large scale multimedia applications simple to implement.



Some interested points to be noticed here:

- PTS channels use time as a first class entity - meaning all data access are time based, queries can manipulate the specific time in which they want to put / get some data
- PTS provides persistent streams under application control - meaning they are persisted to hard storage like disks
- Allows for seamless handling of historical and live data using the runtime system; all operations are abstracted away from the end user / thread / application
- PTS channels are similar to UNIX sockets

## Persistent channel architecture

Components of the channel architecture:

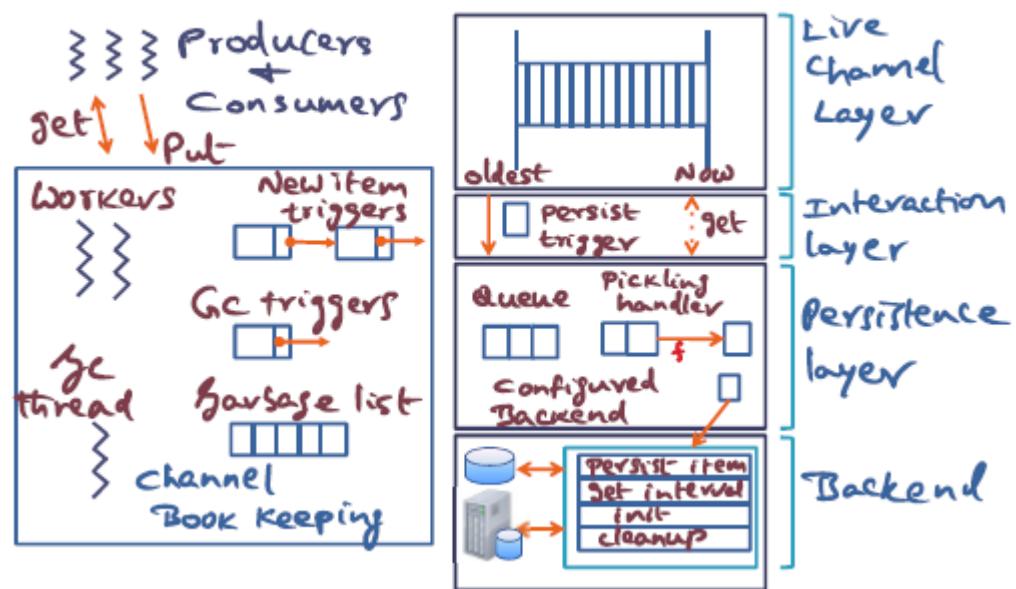
Component	Description

Producers and consumers	<b>get</b> and <b>put</b> data into the channel
Workers	respond to new item triggers and garbage collection triggers
Garbage list	expired content that is no longer relevant in the context of the channel
Live channel layer	Contains a list chronological list of data from oldest to now, keeps the most current events
Garbage collection thread	Does housekeeping for the garbage list
Interaction layer	Layer that allows interaction between the live channel layer and the persistence layer
Persistence layer	Applications can specify a function to determine how often items are persisted and where they're stored
Backend	Archived data

An application can keep data for as long as it wants; applications can decide how long items persist. All of this occurs transparently to the user - the channel handles all of these mechanisms internally.

Below is high level representation of the architecture of persistent channels.

## Persistent channel Architecture



lesson8

# **lesson8**

# Irvm

## Persistence

Why do we need lightweight recoverable virtual memory? Many operating system subsystems require persistence, like inodes for files on the disk.

How do we provide this persistence? We make virtual memory persistent. Subsystems won't have to worry about flushing memory into disk in order to persist specific data structures. This also provides us the ability to easily recover from power failure, software crashes, etc.

Who will use this abstraction? Subsystem designers, but only if it's performant.

If we make virtual memory persistent, we could possibly incur a lot of overhead and latency as the virtual memory is consistently being written to disk upon each update. A solution to this latency issue is to maintain a log of changes rather than continuously writing to disk, essentially buffering the changes of virtual memory. Eventually when all the data is dirty and needs to be written, we conduct one large I/O operation to write to disk.

This method described above avoids the latency incurred when writing every time, while also avoiding the issue of writing to possibly random locations on the disk.

Below is a high level representation of the concepts discussed above.

## Persistence

Why?

- need of OS subsystems

How?

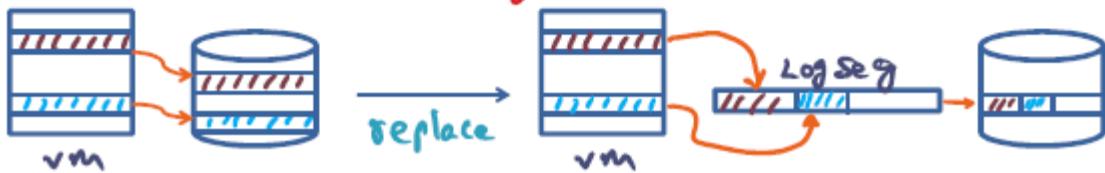
- make virtual memory persistent

Who will use it?

- Subsystem designers if it is **performant**

How to make it efficient?

- use **persistent logs** to record changes to VM



## Server design

The below high level representation shows how a server could provide subsystem designers the facility to maintain persistent data structures. The gist is that subsystem designers would use an API to identify locations in memory that need to be backed by some persistent data segment residing on the server. This avoids the possibility of persisting data from the subsystem that isn't important, or meant to be volatile.

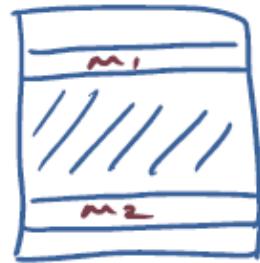
## Server Design

- persistent metadata  $m_1, m_2, \dots, m_n$
- normal data structures + code

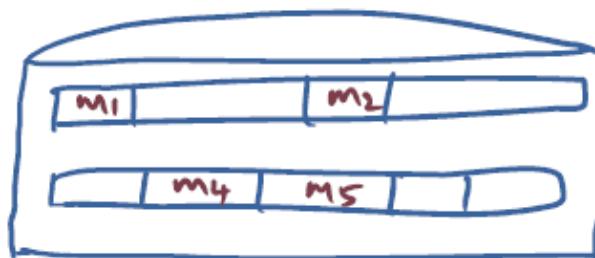


Create external data segments to back persistent data structures

- apps manage their persistence needs



Virtual address space of server



Designer's choice  
to use single or  
multiple data  
segments

## Reliable virtual memory (RVM) primitives

### initialization

- initialize(options)
  - provides the subsystem programmer the ability to initialize the log data structure to begin recording changes to the application's memory
- map(region, options)
  - application specifies the locations in memory in which it wants to persist to external data segments
- unmap(region)
  - undoes a map operation

### body of server code

- begin\_xact(tid, restore\_mode)
  - begins a transaction - starts logging the changes to some location in memory and buffers for writes to persistent memory
- set\_range(tid, addr, size)

- sets the range address range of memory to be logged for writing to persistent storage
- end\_xact(tid, commit\_mode)
  - completes a transaction and stores the changes for the memory addresses modified into persistent memory
- abort\_xact(tid)
  - aborts the transaction, does not write to persistent memory

### **gc to reduce log space**

- flush()
- truncate()
  - done my LRVM automatically for logs
  - also provided for application flexibility

### **miscellaneous**

- query\_options(region)
- set\_options(options)
- create\_log(options, len, mode)

The main take away is RVM is simple in its design and provides primitives for control of its operations. **Transactions** in the context of RVM are completely different than your normal database definition, and much simpler. **Transactions** are only related to the specification and persistence of memory regions.

Below is the slide related to the portion of this lecture.

## RVM Primitives

### Initialization

- initialize(options)
- map(region, options)
- unmap(region)

### Gc to reduce log space

- flush()
  - truncate()
- ↓  
Provided for  
app flexibility

### Body of server code

- begin-xact(tid, restore-mode)
- set-range(tid, addr, size)
- end-xact(tid, commit-mode)
- abort-xact(tid)

### Miscellaneous

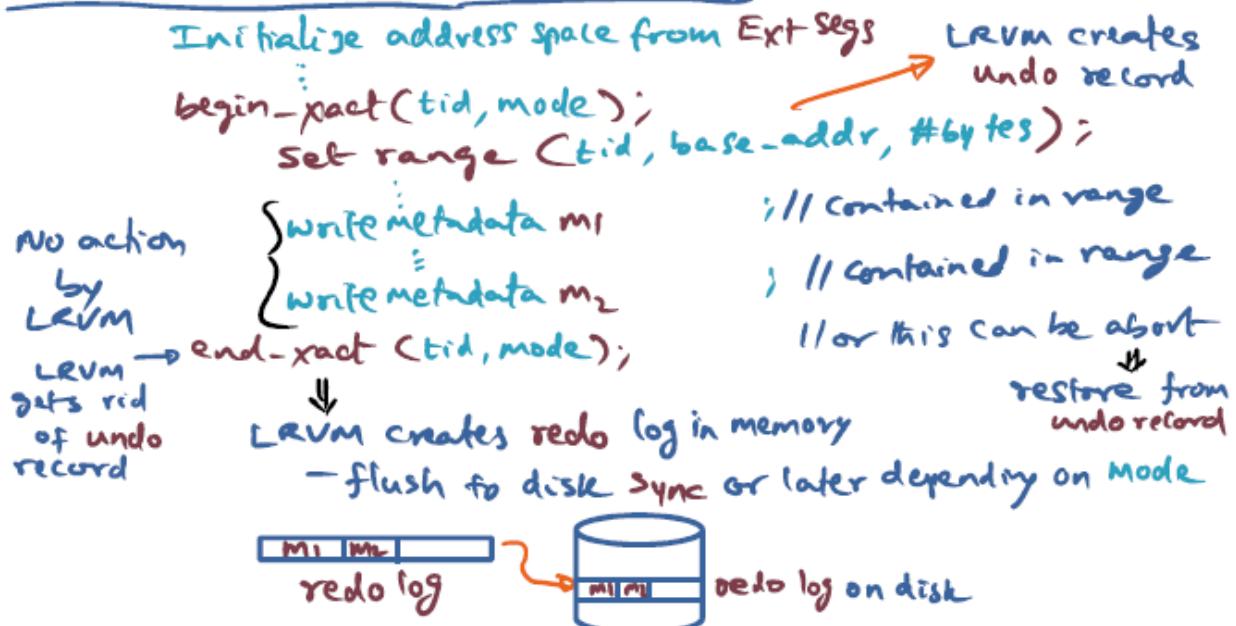
- query-options(region)
- set-options(options)
- create-log(options, len, mode)

⇒ Simplicity - small set of primitives

## How the server uses the primitives

Below is a high level representation of how RVM supports transactions, creates undo records, creates redo logs, and flushes changes to disk.

### How the Server uses the primitives



## Transaction optimizations

RVM provides developers to give hints to the library when it wants to utilize undo and redo logs. You can defer to use an undo log if you know the changes you want to make will be committed. You can also defer the flushing of a redo log if you know future changes are going to occur.

If a transaction's flush is deferred, this creates a window of vulnerability that the developer has to accept as risk to their data being lost on system failure. Transactions can be seen as insurance for a developer's memory for an application.

### Opportunities for server to optimize transactions

no-restore mode in begin-xact

- no need to create in-mem undo record

no-flush mode in end-xact

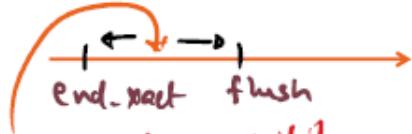
- no need to sync flush redo log to disk

⇒ lazy persistence

⇒ upshot?

⇒ window of vulnerability

use xactions as insurance



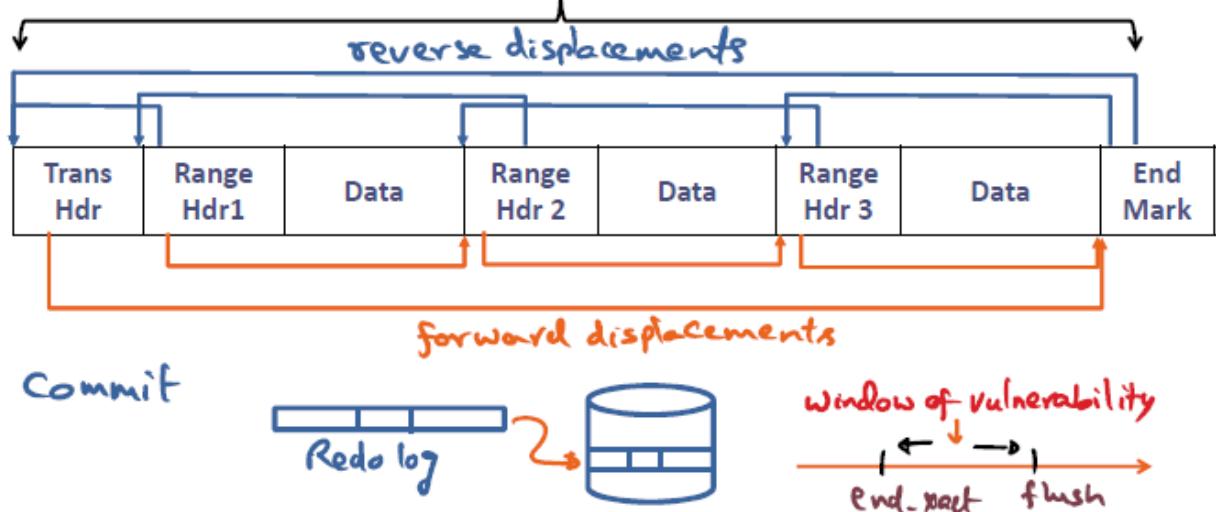
## Implementation

Below is a high level representation of the redo log structure.

## Implementation

### Redo log

- All changes to different regions between begin and end xact



## Crash recovery

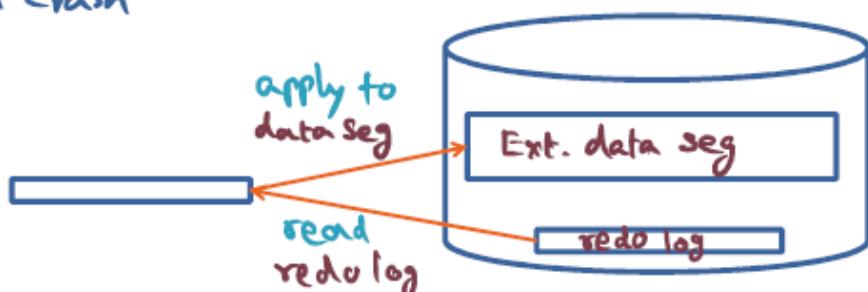
Below is a high level representation of how RVM recovers from crashes using the redo log record hosted on the disk.

## Crash Recovery

reverse displacements



Resume from Crash



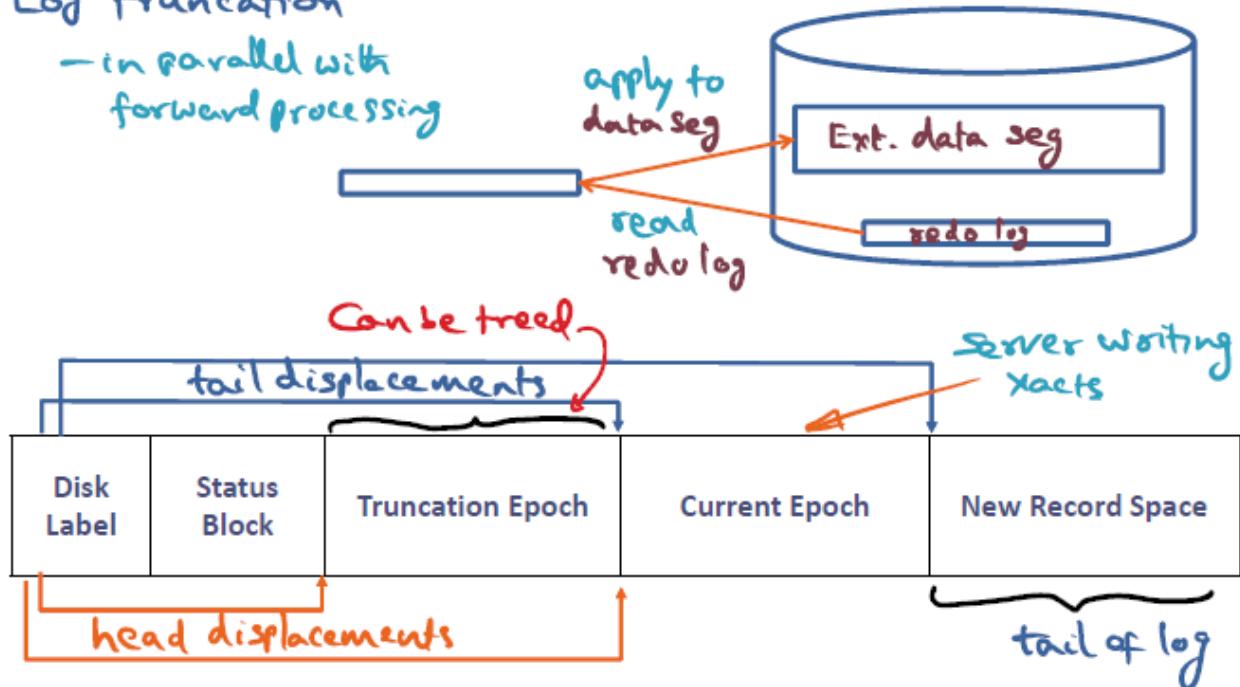
## Log truncation

Below is a high level representation of how RVM conducts log truncation. Essentially, LRVM determines that the redo log has reached a critical size and some of the redo log needs to be written to disk. The log is split into epochs, with a truncation epoch, a current epoch, and a new record space generated on the truncation of an epoch.

Servers wait for transaction to occur, and those are stored in the current epoch. Truncation epochs are moved to permanent storage, and a new record space is generated in tandem (parallel) with forward processing for transactions.

## Log truncation

- in parallel with forward processing



# riovista

## System crash

There are two problems that lead to a system crash: power failures and software crashes. RioVista poses a solution, using hardware to solve our problem of losing information at power failure (e.g a UPS). This will make the problem of power failures disappear, making the only source of failure software crashes. RioVista poses to store portion of main memory in a system that will survive crashes, both software and hardware.

### System Crash

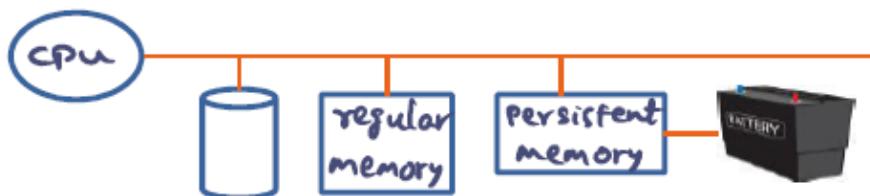
Two problems concerning failure

#### - Power failure

\* Can we throw some hardware at problem and make it disappear (e.g., UPS Power Supply)

#### - Software Crash

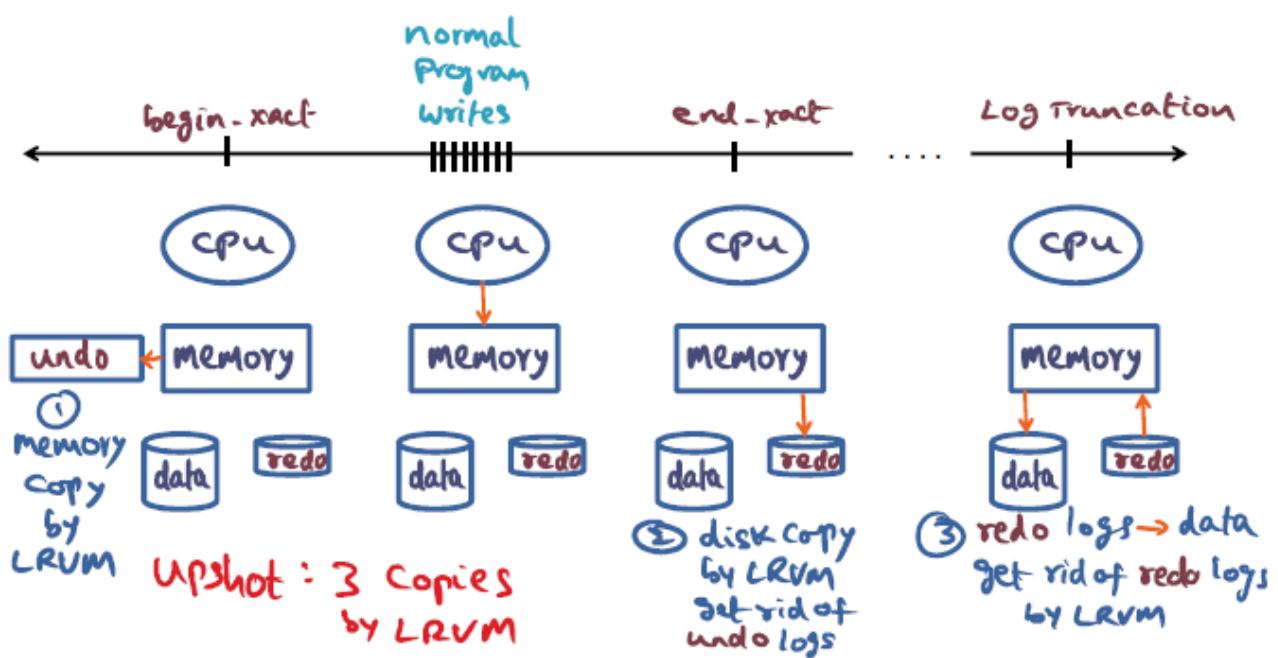
\* reserve a portion of main memory that survives crashes



## LRVM revisited

Below is a high level representation covering the steps of a transaction using LRVM. Biggest issue with LRVM is the undo log and redo log are lost during power failure.

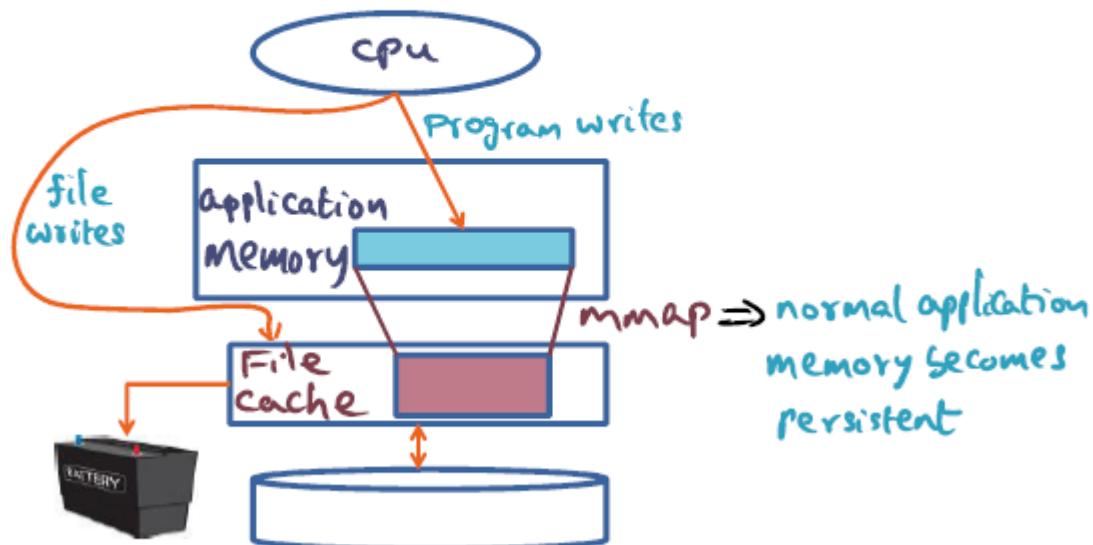
## LRVM Revisited



## Rio file cache

Below is a high level representation of how Rio creates a persistent file cache using hardware that has a backup UPS power supply. The file cache also has virtual memory protection to protect it from a stray operating system when the OS is crashing upon power failure.

## Rio File Cache



## Vista RVM on top of Rio

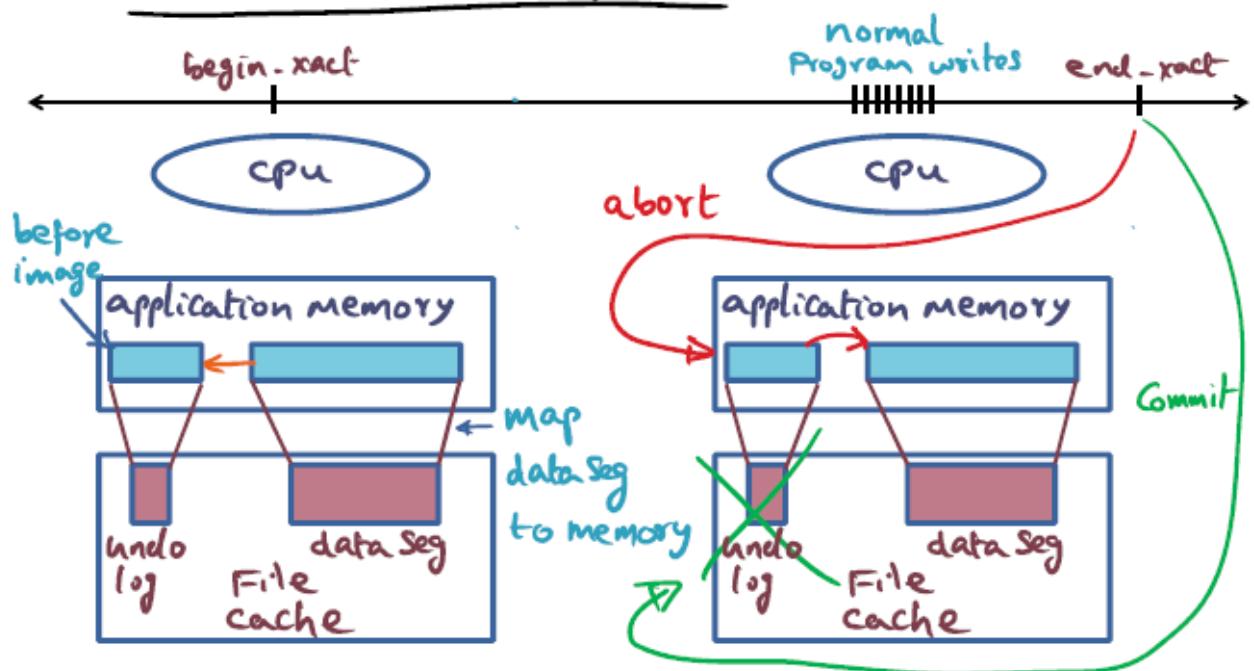
Vista RVM is similar to LRVM, however, it leverages the Rio file cache. The undo log that was previously created in RAM is now hosted in persistent memory of the Rio file cache. The data segment that is identified as persistent by the programmer are also mapped to data segments in the Rio file cache.

If an abortion of the transaction occurs, the undo log is merged into the memory, thus it's also merged into the mapped portion of the file cache. If the transaction is committed, the undo log is destroyed. There is no need to flush a redo log like LRVM, however, because all the changes have already occurred - memory between the application and the file cache was mapped the entire time.

The implication for this method of transactions is that there is no disk I/O at all in order to create persistent memory.

Below is a high level representation of how transactions are conducted with Vista RVM and Rio file cache.

## Vista - RVM on top of Rio



## Crash recovery

Crash recovery is treated just like an abort. When the operating system boots, the file cache will recover the old image of an application from the undo log and undo the changes to the application.

## Vista simplicity

Vista's performance is by virtue of its simplicity. Why is it so simple? There are no redo logs, no log truncation code. Checkpointing and recovery code are simplified because all transactions utilize the Rio file cache. No group commit optimizations have to be made because commits are automatic, the only data structure maintained is the undo log.

Vista is simple like LRV, but more performance efficient.

## Vista Simplicity

700 lines of code in Vista

- 10K lines in LRVN

Why?

- no redo logs or truncation code
- checkpointing and recovery code simplified
- no group commit optimizations

Upshot

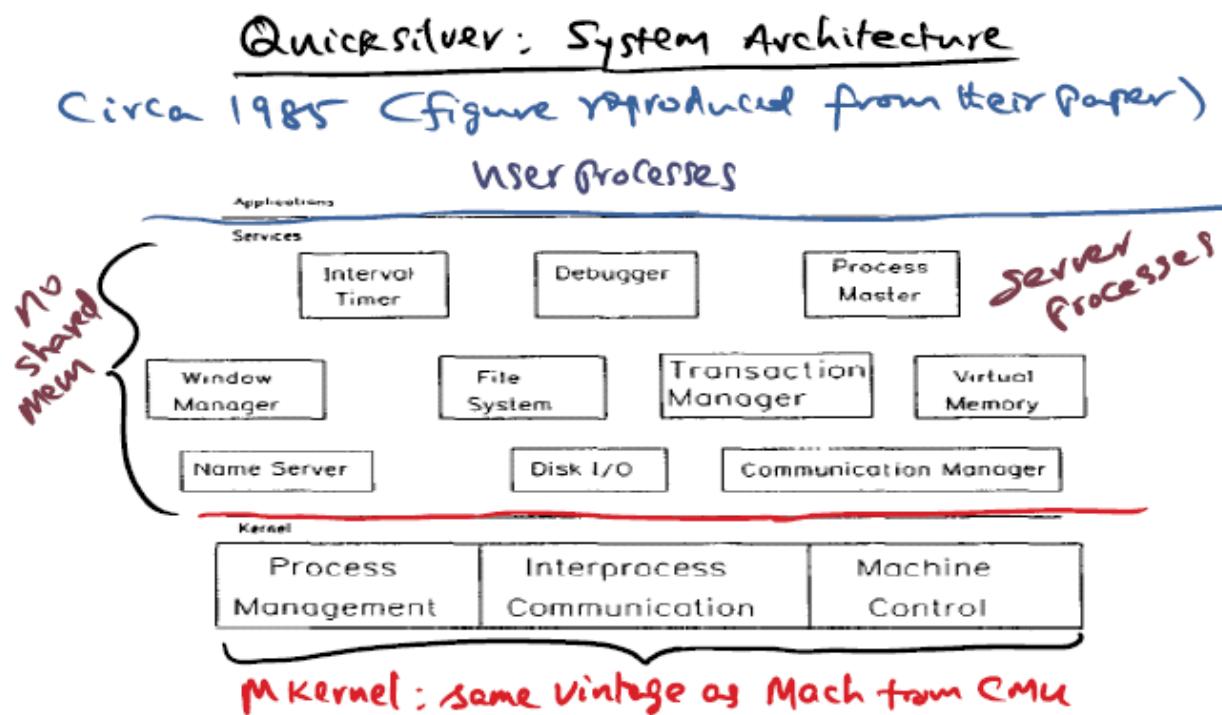
- simple like LRVN but performance efficient

# quicksilver

## Quicksilver system architecture

Quicksilver is a project started by IBM in 1984. The main argument Quicksilver raises is that system recovery should not be an after thought to operating system design, but a first class citizen - implemented into the core of an operating system. There are a lot of parallels between Quicksilver's operating system structure and the structure of a network operating system. Both have microkernels handling smaller, privileged operations, and dispersed server processes that implement subsystems of the operating system.

Below is a high level representation of Quicksilver's operating system architecture.

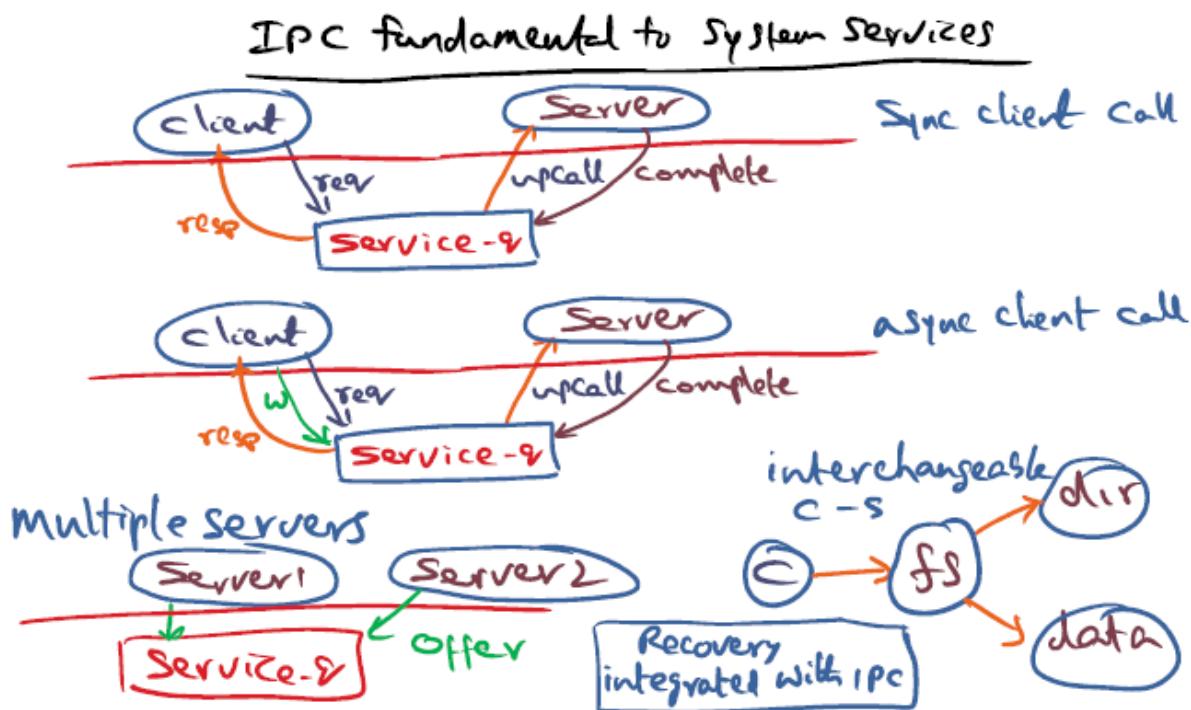


## IPC fundamental to system services

Clients and servers within the Quicksilver operating system conduct inter-process communication by using a service queue. The service queue is globally accessible, and different servers can handle client requests.

The IPC mechanisms provides the ability for synchronous and asynchronous calls. The most important point is that recovery mechanisms are built into the IPC mechanism - integrated as described before. Recovery is not taking a backseat in Quicksilver.

Below is a high level representation of the IPC mechanism used in Quicksilver's system services.

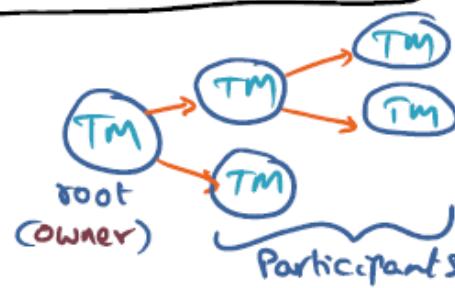
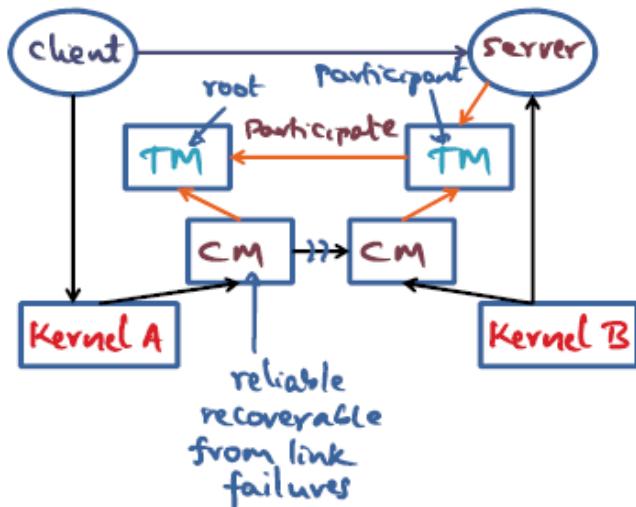


## Bundling distributed IPC and transactions

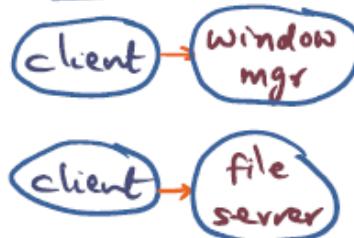
Below is a high level representation of how transactions are handled in a distributed manner within a Quicksilver system. Each node within the system participates in a transaction of data, and each node persists the changes of the transaction between client and server. Multiple nodes can elect to participate in a transaction, further increasing the ability to recover from system crashes.

## Bundling Distributed IPC & Actions

Transaction  $\Rightarrow$  Secret source for recovery management



### Examples

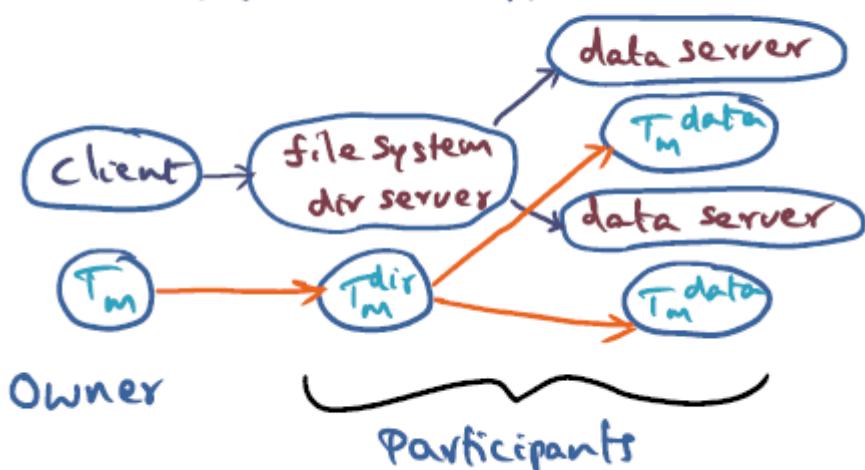


## Transaction management

Below is a high level representation of how a transaction tree expands as a client, server interaction occurs to acquire files from a data server. Keep in mind, the owner can select other nodes to be transaction managers if cleanup needs to be done if the owner fails.

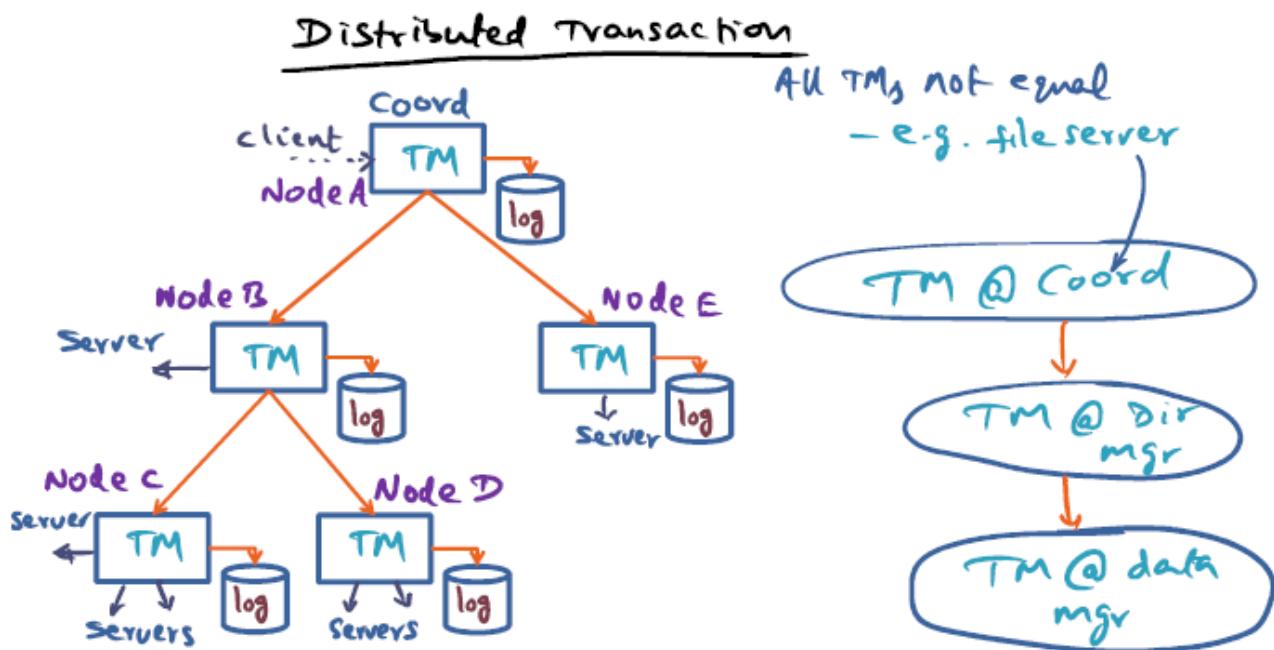
## Transaction management

Coordinator can be different from owner



## Distributed transaction

Below is a high level representation of how a distributed transaction occurs. At each node, they have a transaction manager that will occasionally log the occurrences, results, and data of transactions. At the top, the transaction coordinator is the client. Clients are usually the most brittle of transaction managers, thus, other transaction managers are aware when other managers crash.

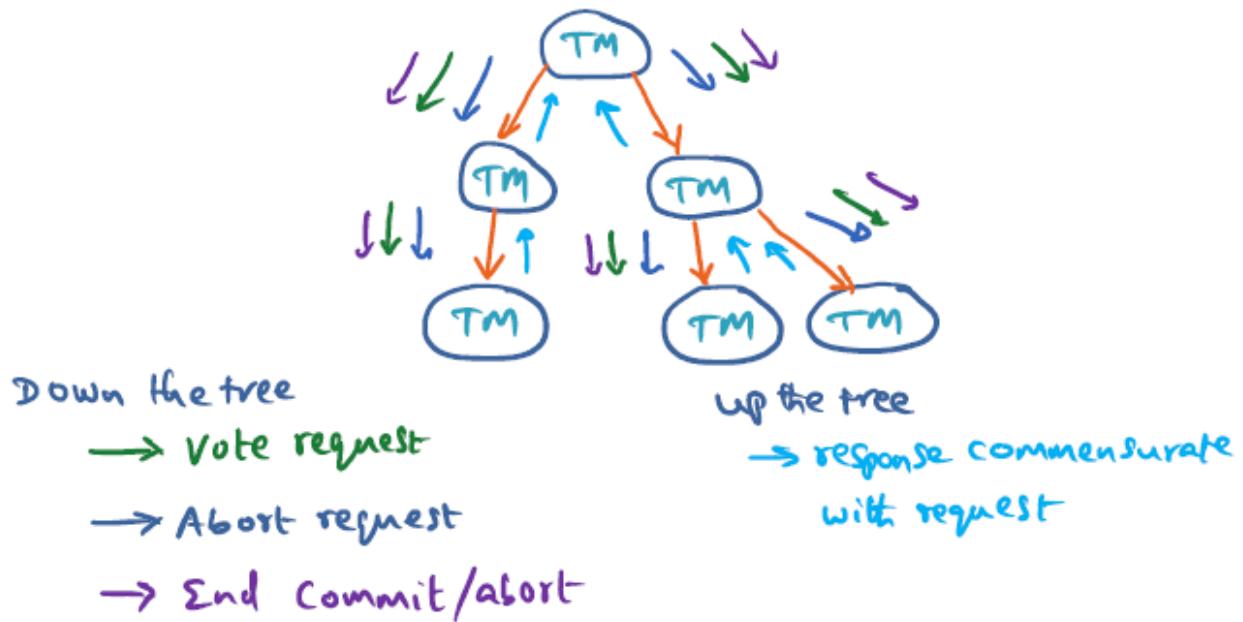


## Commit initiated by coordinator

The transaction coordinator can send **vote requests**, **abort requests**, and **end commit / abort** messages. And all the other transaction managers will contact the coordinator and acknowledge the completion of the request.

So say the client conducts a transaction to open a page on a window manager. If that client closes, the delegated coordinator after the client will notify the window manager that the transaction is aborted. The window manager can then acknowledge the abortion and cleanup any state that it had for the transaction.

## Commit initiated by coordinator



## Upshot of bundling IPC and recovery management

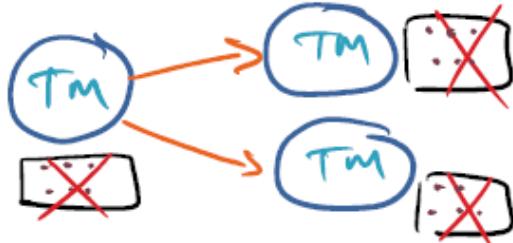
Advantages of this setup is Quicksilver provides better ability to cleanup breadcrumbs left behind by failed clients and servers (memory, filehandles, communication handles, orphaned windows / processes, etc.)

No extra communication is required for recovery because it is all bundled in the transaction / IPC mechanisms. The overhead that exists for recovery is similar to LRVM - log records must be written and flushed to disk to maintain persistent state.

## upshot of bundling IPC + Recovery

### Reclaim Resources

- breadcrumbs left behind by failed clients/servers



Examples:

memory, filehandles,  
communication handles,  
orphan windows, ...

No Extra Communication for recovery

only mechanism in OS, policy up to each service

- low overhead mechanisms for simple services
- weighty mechanisms for services such as FS

## Implementation

Key sectors of implementation for Quicksilver is log maintenance:

- Transaction managers write log records for recovering persistent state
- The frequency of these log forces or writing to disk will impact performance
  - This not only impacts the particular node, but all other nodes involved in the transaction

Services must choose mechanisms that match their recovery requirements. Some services need large amounts of persistent state, while others may need to only persist some data every once in a while.

lesson11

# **lesson11**

# principles of information security

## terminologies

When do computing systems release information? Jerome H. Saltzer and Michael D. Schroeder define two terms when discussing the protection of information in computing systems:

Term	Definition
privacy	denotes a socially defined ability of an individual to determine whether, when, and to whom personal information is to be released
security	describes the techniques that control who may use or modify the computer or the information contained in it

They also define potential security violations, and these can be placed into three categories:

Category	Description
unauthorized information release	an unauthorized person is able to read and take advantage of information stored in the computer
unauthorized information modification	an unauthorized person is able to make changes in stored information - a form of sabotage
unauthorized denial of use	an intruder can prevent an authorized user from referring to or modifying the information, even though the intruder may not be able to refer to or modify the information

The goal of protection a computing system is to prevent all of these potential security violations. This is a **negative statement**, however, because it would be hard to prove that all bugs have been found and remedied within every piece of software for a computing system.

## Levels of protection

Levels of protection are defined in the paper mentioned above:

Protection level	Description
unprotected	<p>no levels of protection</p> <ul style="list-style-type: none"><li>• MS-DOS (contained mistake prevention, but no protection)</li></ul>
all or nothing	<p>completely segregated user / system contexts</p> <ul style="list-style-type: none"><li>• VM-370 and other time sharing systems (each user given the illusion they had their own machine - could only communicate by explicitly conducting I/O)</li></ul>
controlled sharing	access lists associated for files
user programmed sharing controls	similar to UNIX file system access rights (user, group, world)
strings on information	organization that generates the information provides labels for the information, requiring that only individuals with the right access are able to access the information (example would be military TOP SECRET / SECRET / UNCLASSIFIED / etc.)

## Design principles

The paper mentioned above also identifies eight design principles that go hand-in-hand with the levels of protection identified above:

Principle	Description

economy of mechanisms	easy to verify
fail safe defaults	explicitly allow access to information
complete mediation	the security mechanism should not take any shortcuts
open design	publish the design, protect the keys - breaking of keys should be computationally infeasible
separation of privileges	two keys by two different entities required to access
least privilege	"need to know" based controls
least common mechanism	only place mechanisms in system context where they are necessary (if a library conducts no privileged function, it must remain in user space - no point in placing code within kernel space)
psychological acceptability	the mechanisms are easy-to-use by users, else they won't follow security mechanisms / enforcement

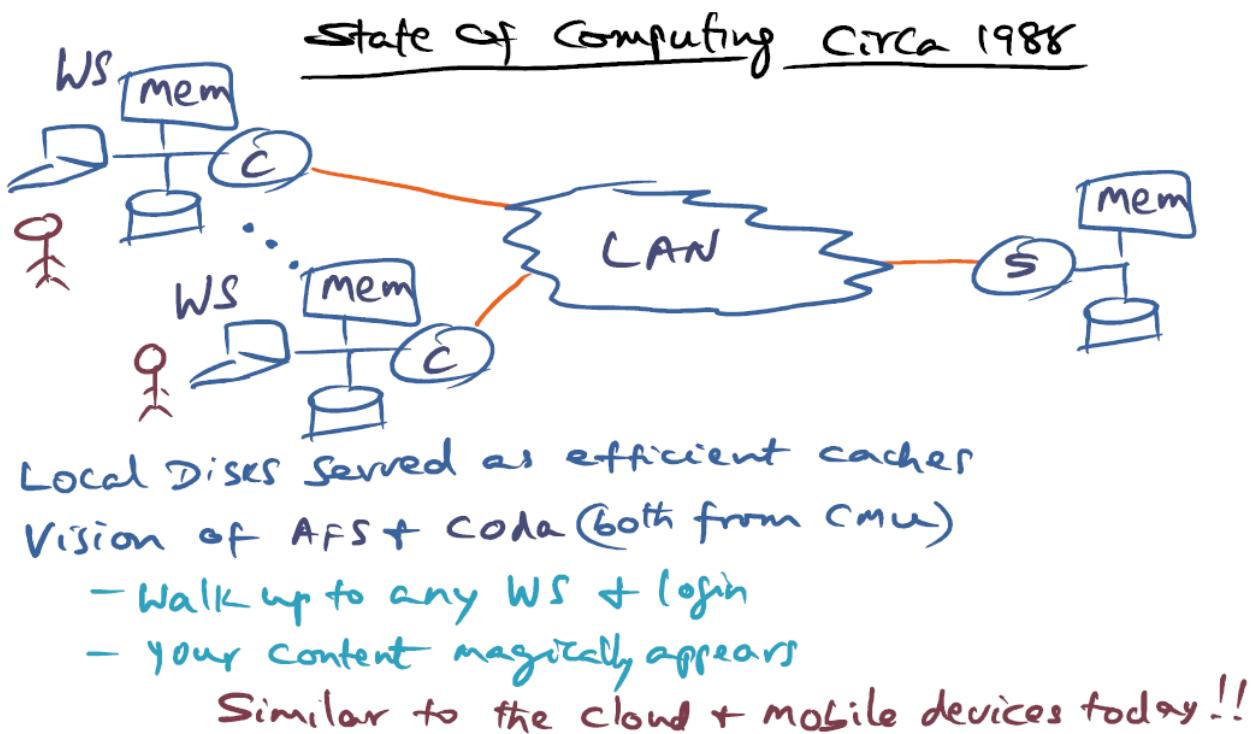
## Takeaways

- Craft the system so that it's difficult to crack the protection boundary - computationally infeasible.
- Detection of security violations rather than prevention.

# security in andrew

The purpose of the Andrew file system was to create a secure, distributed file system for students to use on campus at any terminal. Students would be able to access their personal files on any computer on campus, with security being enforced at every terminal while the files were shared on a central server.

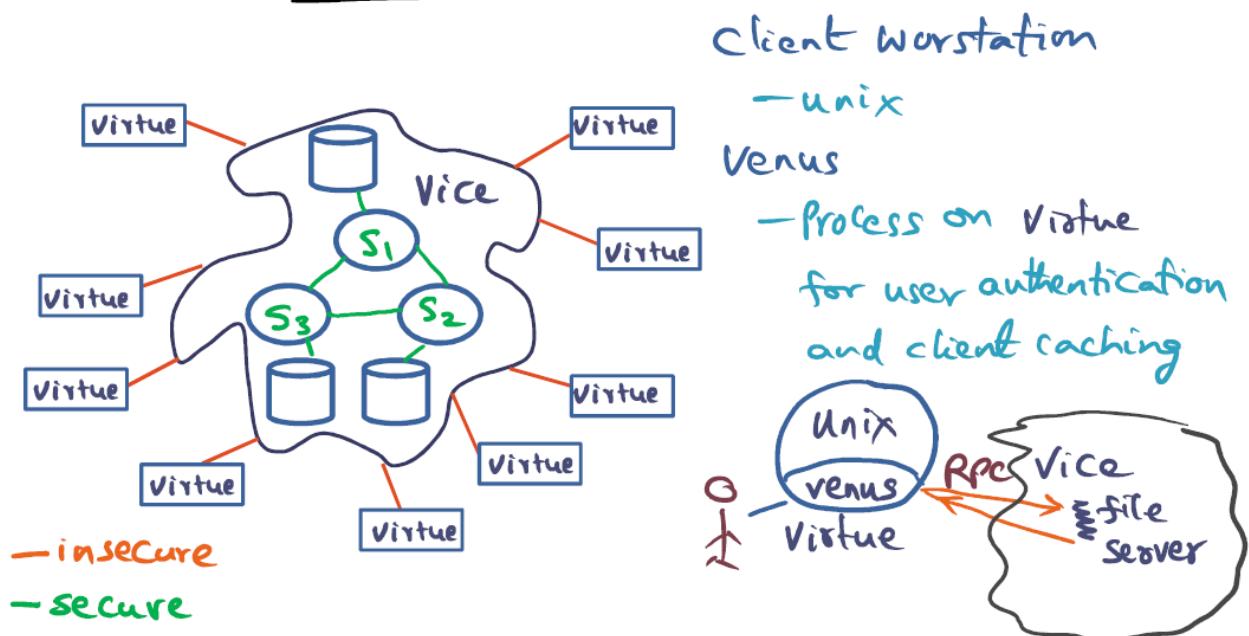
Below is a high level representation of the concept.



## Andrew architecture

Below is a high level representation of the Andrew secure, distributed file system architecture.

## Andrew Architecture



Below is a table with the description for each component of the architecture:

Component	Description
virtues	client workstations, connected by insecure network links to a local area network in order to access servers - traffic emanating from virtues has to be encrypted
servers	connected with secure network links
vice	the secure server environment where servers can conduct secure communication and file transfer - no need for information encryption
venus	application on the virtues that conducts authentication and client-side caching of retrieved files

## Encryption refresher

### Private key encryption

- Symmetric keys (e.g. passwords used to login)

- sender encrypts data with a key and sends the cipher-text to the recipient
- recipient receives the cipher-text and decrypts the data with the same key

## **Public key system**

- Asymmetric key (pair of keys)
  - public key is published and used to encrypt data
  - private key remains unpublished and is used to decrypt data
    - sender encrypts data with the public key and sends the cipher-text to the recipient
    - recipient receives the cipher-text and decrypts that data with the private key
  - encrypting the data with the public key is a one-way function and is only decryptable with the private key
  - decryption of the cipher-text with private key is also a one-way function and is only encryptable with the public key

## **Challenges for the Andrew system**

The challenges include:

Challenge	Description
authentication of users	when a user logs in, the system has to verify that the user logging in is who they say they are
authentication of the server	when a user logs into the workstation and receives a message from the server, the user has to be assured that the server they're communicating with is the actual Andrew server
preventing replay attacks	even though encryption is being used to protect information, we need to ensure that someone doesn't replay an encrypted message in order to fool us
isolation of users	the users are shielded from one another, either through unintentional or intentional actions

## **Choices made by Andrew designers**

The Andrew secure, distributed file system designers decided to use **private key cryptography** to implement secure communication within the system. The key distribution problem didn't seem too large of a problem for a community the size of the CMU campus.

In order to conduct this method of security for communication, the identity of the sender has to be communicated in clear text.

A dilemma for the designers that was presented is, at this time the normal method of authentication was regular UNIX usernames and passwords. However, from the discussion from the previous chapter, the overuse of a security mechanism can expose that security mechanism to attack. Utilizing usernames and passwords for everything, to include the secure RPC between virtues and the vice, presents a security hole and an opportunity for attackers to violate the security of the system.

So what should the designers of the Andrew secure, distributed file system use as the **identity** and **private key** in order to conduct secure RPC between the virtues and the vice?

## Andrew solution

The solution implemented by the designers of Andrew to the predicament identified above is as follows:

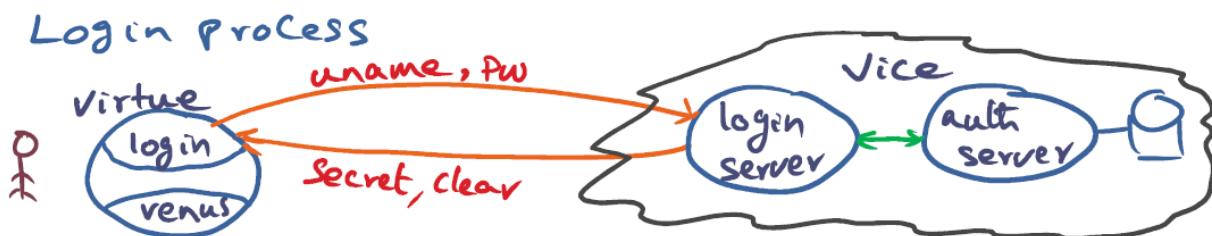
- **username** and **password** were only used for logging in to the actual work station
- all communication with between the **venus** and the **vice** use an **ephemeral id** and **keys**

This gives rise to three classes of client-server interaction:

Class	Description
login	username and password presented to the workstation
RPC session establishment	venus establishes a session with the vice to retrieve files for work
file system access during session	leveraging of the venus RPC session with the vice to retrieve files for work. RPC session is closed once files are retrieved and stored locally within the cache. Once you're done working on the files, the files are returned to the server with a new venus to vice session.

## Login process

Below is a high level representation of how a login is conducted within the Andrew secure, distributed file system.



Cleartoken: Data structure  
⇒ Extract handshake key client (HKC)

Secrettoken: Cleartoken encrypted with key

Known only to vice

⇒ Unique for this login session

↓  
use as ephemeral client-id  
for this login session

use HKC as private key  
for establishing a new  
RPC session

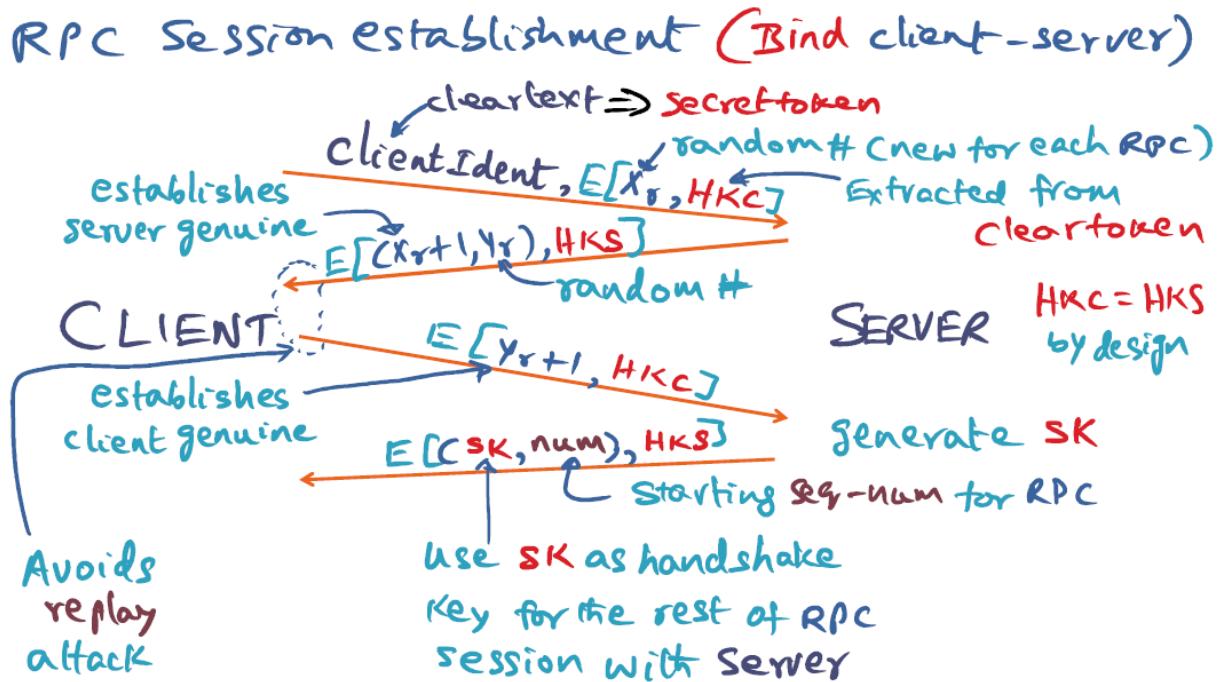
Users login with a **username** and **password** securely over the insecure links and are presented with a pair of keys, the **cleartoken** and the **secrettoken**.

Token	Description
cleartoken	data structure containing the handshake key client
secrettoken	cleartoken encrypted with a private key only known to vice - used as the ephemeral client id for this login session
handshake key client	used as a private key for establishing a new RPC session

This system prevents the username and password being communicated over the insecure link too often. The secret token will be used as the client id for all further communication, and the handshake key will be used for encryption and establishing RPC sessions.

## RPC session establishment

Below is a high level representation of how RPC sessions are established between venus and the vice in the Andrew secure, distributed file system.

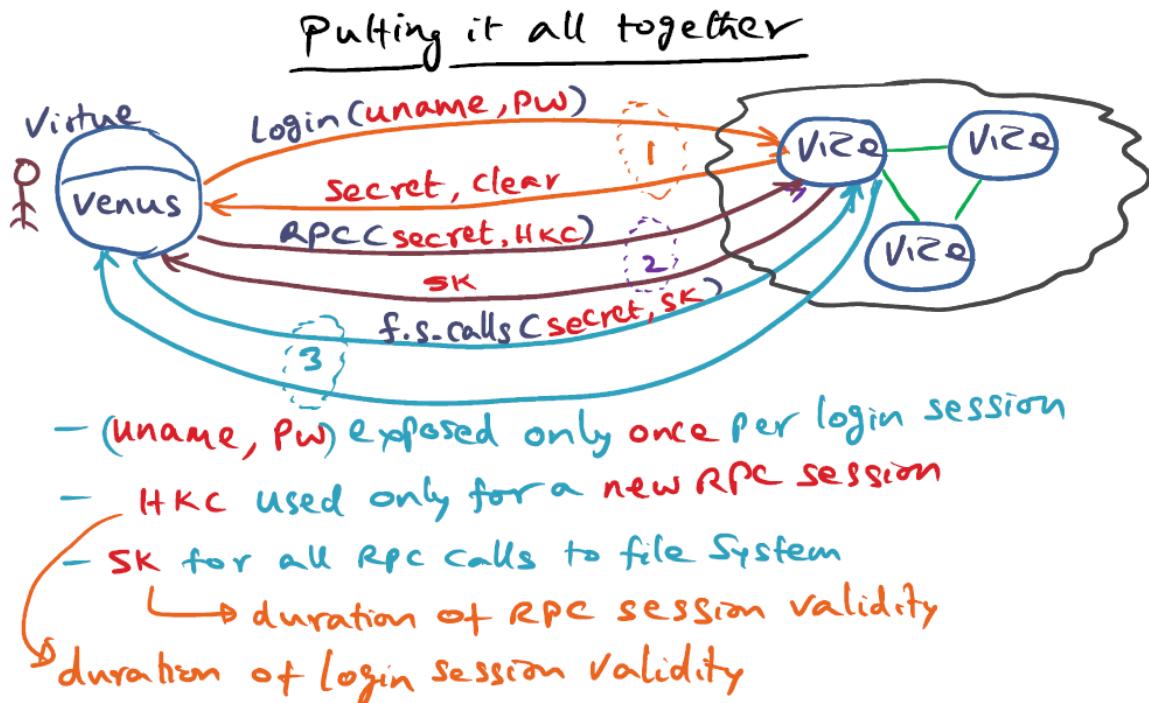


Some special notes:

- The random number ( $X_r$ ) is incremented each time a handshake interaction occurs in order to establish that the server is genuine.
- The server provides a random number ( $Y_r$ ) that is incremented by the client each time a handshake interaction occurs in order to establish that the client is genuine.
- Once the handshake is complete, the server creates a sequence number for the rest of the RPC session.
- Once the handshake is complete, the server creates a session key for the rest of the RPC session.
- The mechanism to increment these random number avoids replay attacks by a person sniffing traffic.

## Putting it all together

Below is a high level representation of the full sequence of actions used to conduct authentication, session establishment, communication, and transfer of files between a virtue and a vice in the Andrew secure, distributed file system.



Special notes:

- Username and password is exposed over the encrypted link once for a login session.
- The handshake key is only valid for the initiation for a new RPC session.
- The session key is only valid for the duration of the RPC session.
- The secrettoken provided by the vice that represents the login session is the only piece of data that exists for the duration of the user's session with the Andrew file system.

## Andrew file system security report card

Does the Andrew file system support mutual suspicion?

- Yes  
 No

Yes, the Andrew file system enforces suspicion of other users as well as suspicion of the server.

#### **Does the Andrew file system support protection from system for users?**

- Yes
- No

There's no way to protect the user from the system. The user must trust the system not to modify its files hosted on the system.

#### **Does the Andrew file system support confinement of resource usage?**

- Yes
- No

Users can consume a lot of network bandwidth and cause a denial of service.

#### **Does the Andrew file system support authentication?**

- Yes
- No

Through the use of username and passwords, session creation is supported by an authentication mechanism.

#### **Does the Andrew file system support server integrity?**

- Yes
- No

The servers in the vice are protected by a firewall, but all communication between the servers are not encrypted. If an attacker compromises the vice, the Andrew file system's integrity is compromised. The Andrew file system has to be physically secured in order to support integrity.