Cloud computing and distributed systems

Chapter #7
Operating system support

Zeynep Yücel

Ca' Foscari University of Venice
zeynep.yucel@unive.it
yucelzeynep.github.io

Introduction

- Remote communication support by OS
- Main role: abstraction of physical resources
- Two perspectives
  - Network OS
  - Distributed OS

Introduction

- Network OS: UNIX, Windows
- Networking capability
  - Network-transparent access
    - NFS file access
    - File sharing simplification
- Defining characteristic: Independent resource management
  - One system image for each node
- Remote login and process execution

Introduction

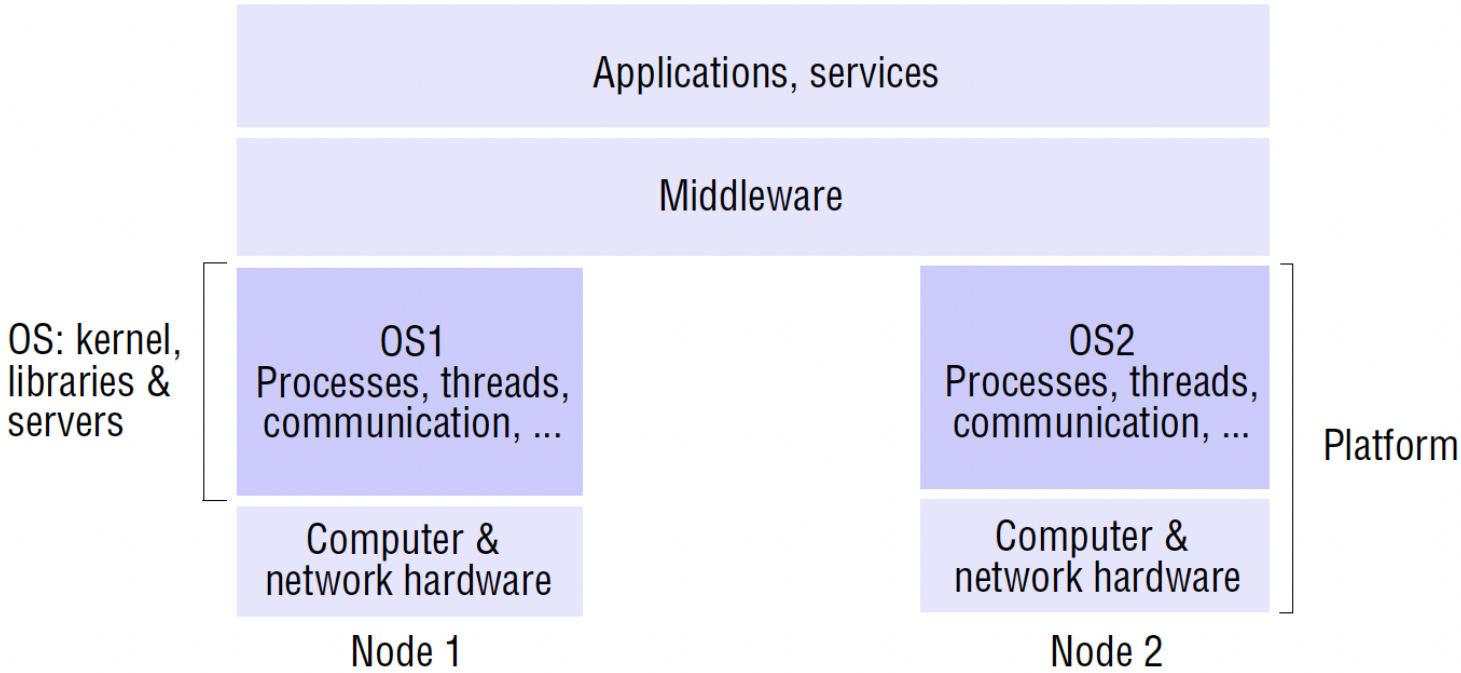
- Distributed OS concept
  - Control over all nodes
  - Resource management across nodes
  - Single system image
- Transparent process allocation
- Process creation at least-loaded node

Introduction

Middleware and network operating systems

- No widely used distributed OS
  - User reluctance to switch
  - Preference for autonomy
- Combination of middleware and network OS
  - Network OS enables standalone applications
  - Middleware provides distributed services

The operating system layer



- Middleware operates on multiple OS-hardware combinations
- OS provides abstractions for local resources and aids middleware
  - OS layer supports common middleware

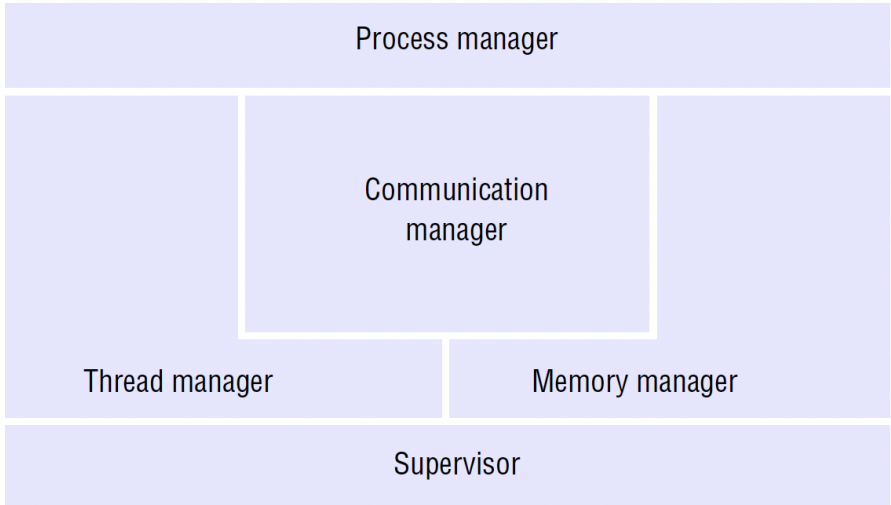
The operating system layer

- Kernel: Core OS component
- Main components: kernel, server processes
  - Server process executes upon kernel
- Kernels and server processes should offer:
  - Encapsulation
  - Protection
  - Concurrent processing

The operating system layer

- Clients access resources through RMIs or system calls.
- Accessing encapsulated resources: invocation mechanism.
- Key tasks include:
  - Communication
  - Scheduling

The operating system layer



- Core OS functionalities:
  - Process and thread management
  - Memory management
  - Process communication
- OS software designed for portability
  - High-level languages

The operating system layer

- Core OS components:
  - Process Manager
  - Thread Manager
  - Communication Manager
  - Memory Manager
  - Supervisor

Protection

- Resource protection from malicious and benign code with bugs
- Illegitimate access: Unauthorized actions
  - Operations allowed only for authorized users
- Bypass prevention of allowed interactions

Protection

Kernels and protection

- The kernel remains loaded after system initialization.
- Runs with full access to physical resources.
- Ensures authorized process access.
- Operates in supervisor mode.
  - Other processes run in user mode.

Protection

Kernels and protection

- Kernel sets up address spaces for protection.
  - Address space: virtual memory location (ranges)
  - Access rights apply
- User processes operate in limited access address spaces.
  - No access beyond address space
- Processes can switch from user to kernel space.

Processes and threads

- Traditional OS: Process executes single activity.
  - Insufficient for concurrent applications.
- Process enhancement: Association with multiple activities.
- A process now consists of:
  - Execution environment
  - 1+ thread(s)

Processes and threads

- Execution environments are costly to create.
  - Shared by multiple threads.
- Threads can be created and destroyed dynamically.
- Goal: increase task overlap and resource use.
- Term "thread" may confuse with "process".
  - Use multi-threaded process for clarity.

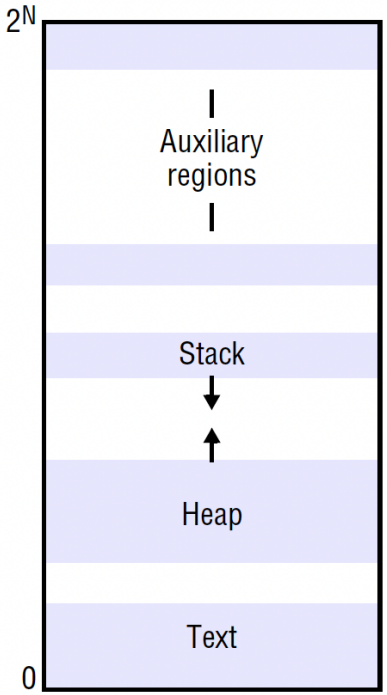
Processes and threads

Address space

- Address space: management unit of virtual memory.
- Contains accessible regions of memory.
- Each region has specified:
  - Extent (address, size)
  - Permissions
  - Growth potential
- Sparse set of disjoint regions.
  - Gaps for growth allowance.

Processes and threads

Address space



- UNIX address space: Three regions
  - Text region
  - Heap
  - Stack

Processes and threads

Address space

- "Stack" vs "Heap":
  - Stack:
    - Stores temporary variables.
    - Automatic memory allocation.
    - Faster access.
  - Heap:
    - Dynamic memory allocation.
    - Manual allocation/deallocation.
    - Slower access.

Processes and threads

Address space

- Indefinite regions motivated by:
  - Separate stack for each thread: manage limits, detect overflows.
  - Files mapped into address space.
  - Shared memory regions for processes.

Processes and threads

Creation of new process

- Process creation: traditionally a single indivisible operation.
- In distributed systems, involves multiple computers.
- Separate into two aspects:
  - Target host selection
  - Execution environment creation

Processes and threads

Choosing a target host

- Node choice: policy matter.
- Process allocation policies:
  - Running locally
  - Distributing across computers
- Two policy categories:
  - Transfer policy: Local or remote
  - Location policy: Node selection

Processes and threads

Choosing a target host

- Load-sharing systems can be:
  - Centralized: One load manager
  - Hierarchical: Several load managers
  - Decentralized: Nodes exchange info
- Simplicity in load-sharing schemes crucial.

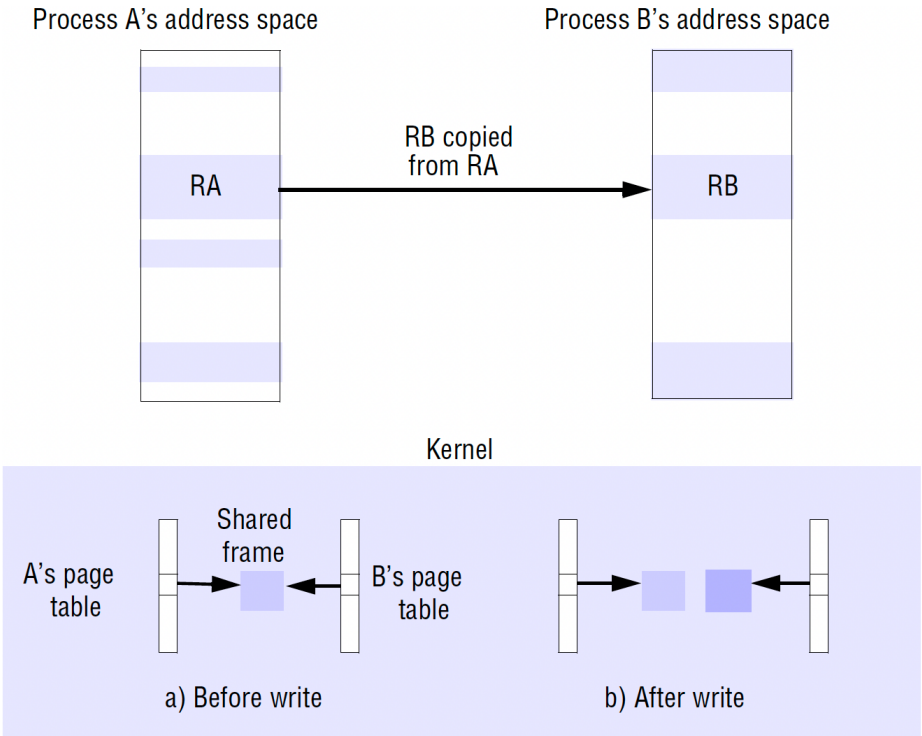
Processes and threads

Creation of new execution environment

- Execution environment requires address space.
- Two setup approaches:
  - Statically defined address spaces.
  - Based on existing execution environment.

Processes and threads

Creation of new execution environment - Copy-on-write

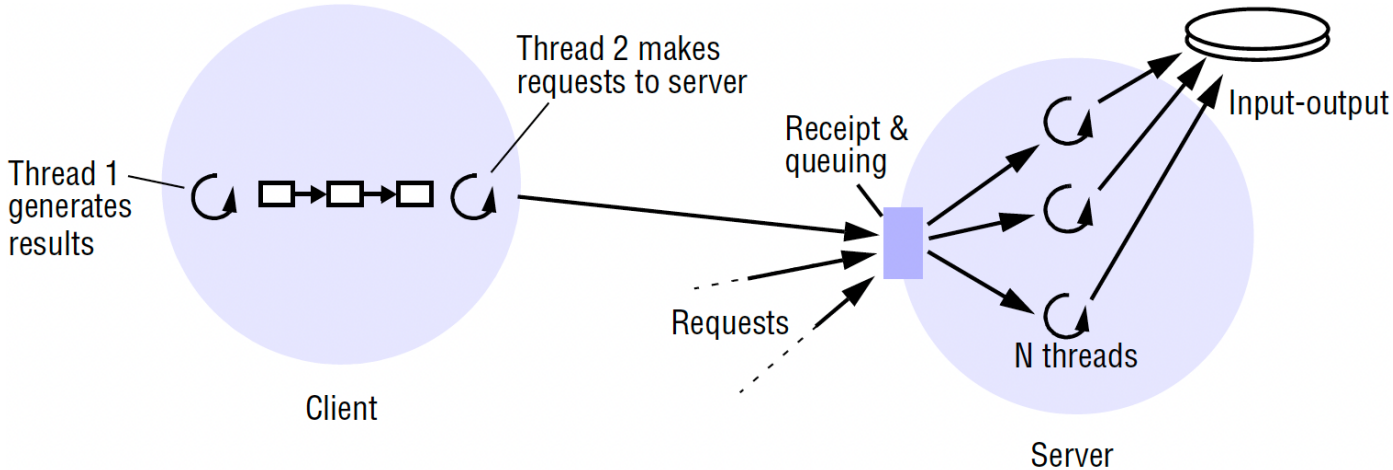


- Copy-on-write: Optimize memory use.
- Initial sharing, then physical copying occurs on modification.

- Shared region example: Parent (A) and child (B).
- Page frames mapped into B's region.
- Write to shared page triggers copy.

Processes and threads

Threads



- Process consists of execution environment and threads.
- Server handles requests using threads.
- It processes requests from a queue.

Processes and threads

Threads

- Case 1: One thread, processing 2 msec, I/O 8 msec.
- Max throughput: 100 requests/sec.
- Case 2: Two threads, one scheduled when another is blocked.
- Max throughput: 125 requests/sec.

Processes and threads

Threads

- Introducing disk block caching improves throughput.
- Assume 75% hit rate: I/O 2 msec.
- If processing $>$ I/O, throughput limited by processor.

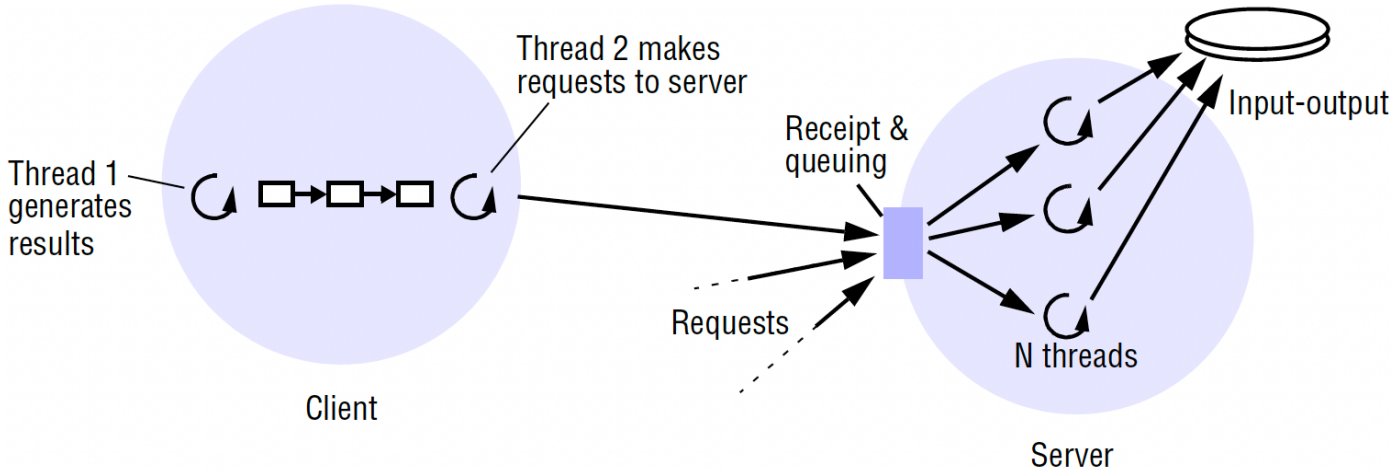
Processes and threads

Architectures for multi-threaded servers

- Multi-threading improves throughput.
- Different threading architectures:
  - Worker pool
  - Thread-per-request
  - Thread-per-connection
  - Thread-per-object

Processes and threads

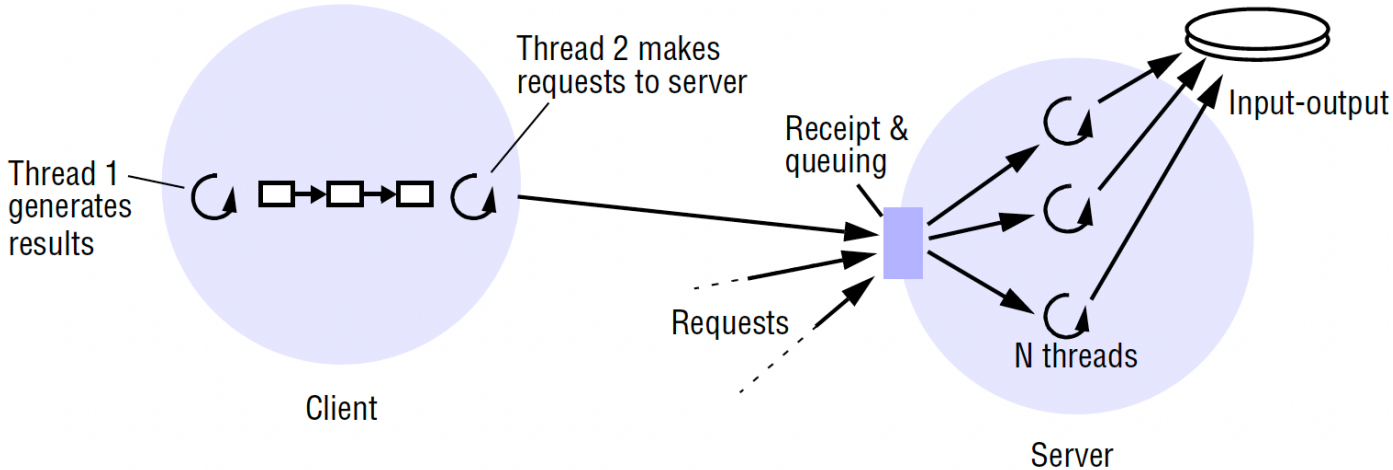
Worker pool architecture



- Worker pool architecture: Fixed number of worker threads.
- I/O thread implements receipt and queuing.
- Handles varying priority requests.

Processes and threads

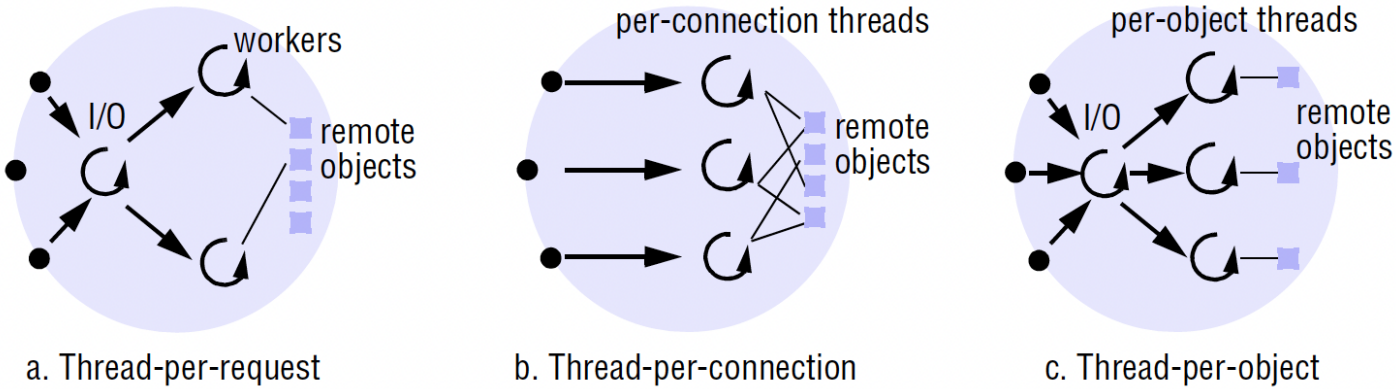
Worker pool architecture



- Disadvantages of worker pool architecture:
  - Insufficient worker threads for high rates.
  - Significant switching between I/O and worker threads.

Processes and threads

Thread-per-request architecture

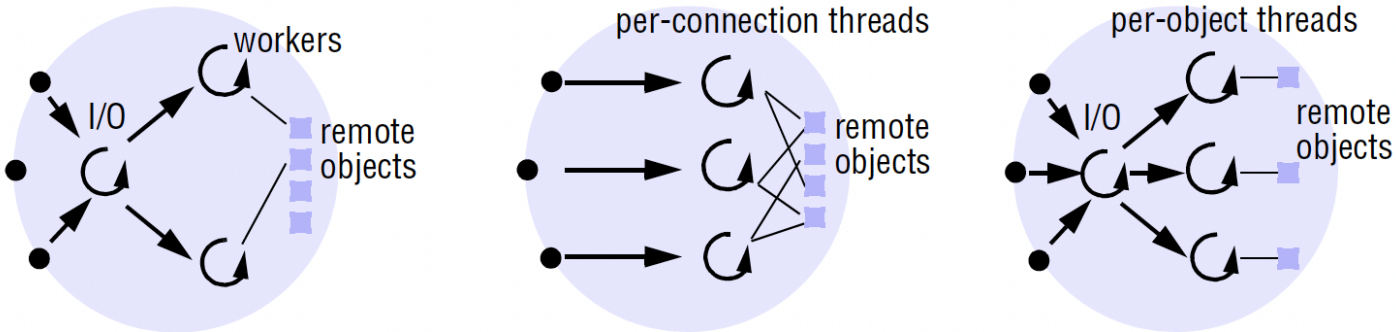


a. Thread-per-request b. Thread-per-connection c. Thread-per-object

- Thread-per-request architecture: New worker thread for each request.
- Thread destroys itself after processing.
- Advantage: Maximize throughput.
- Disadvantage: Thread creation/destruction overhead.

Processes and threads

Architectures for multi-threaded servers

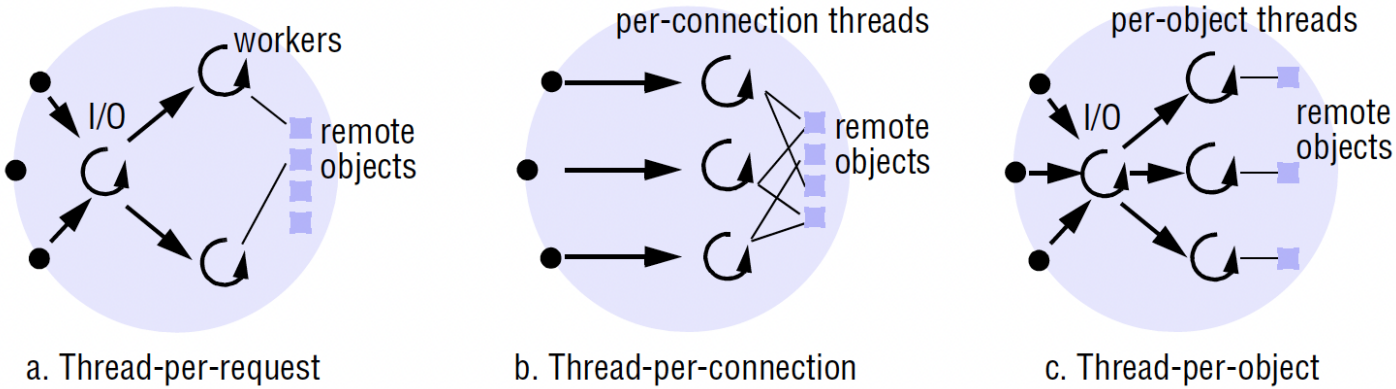


a. Thread-per-request b. Thread-per-connection c. Thread-per-object

- Thread-per-connection architecture: New thread for each client connection.
- Destroys thread upon client closure.
- Handles multiple requests over connection.

Processes and threads

Architectures for multi-threaded servers

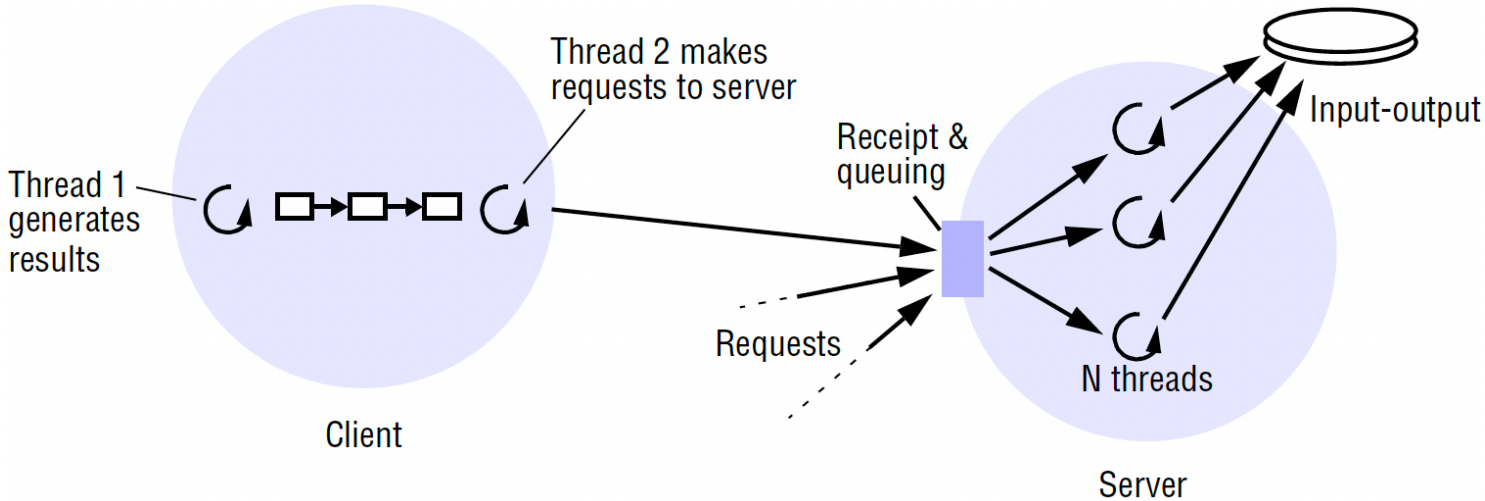


a. Thread-per-request b. Thread-per-connection c. Thread-per-object

- Object-per-connection architecture: New thread for each remote object.
- I/O thread queues requests per object.
- Advantage: Lower thread management overhead.
- Disadvantage: Uneven workload among threads.

Processes and threads

Threads within clients



- Threads useful for clients too.
- If thread-1 sends results to server, it needs to block.
- If thread-2 handles sending, thread-1 proceeds without blocking.

Processes and threads

Threads versus multiple processes

- Threads allow overlapping computation with I/O.
- Multi-threaded processes preferred over single-threaded for:
  - Cheaper creation/management.
  - Efficient resource sharing.

Processes and threads

Threads versus multiple processes

- Creating a new thread: stack allocation only.
- Creating a new process: new execution environment required.
- Key comparisons:
  - Creating thread cheaper than process.
  - Switching between threads cheaper than processes.
  - Threads share data/resources easily.
  - Threads lack protection from each other.

Processes and threads

Thread scheduling - Preemptive or nonpreemptive

- Thread scheduling manages execution across threads.
- Can be preemptive or nonpreemptive.
  - Preemptive: Threads can be interrupted anytime.
  - Non-preemptive: Threads run until voluntarily yielding.
- Non-preemptive scheduling cannot effectively use multiprocessors.
- Applications may implement their own scheduling policies.

Processes and threads

Threads implementation - Kernel support vs user-level

- Many kernels (Windows, Linux) support multi-threaded processes natively.
- Some systems only support single-threaded processes.
- User-level implementations have advantages and disadvantages.

Processes and threads

Threads implementation - Kernel support vs user-level

- User-level threads face issues without kernel support:
  - Cannot leverage multiprocessor.
  - Page fault in one thread blocks the entire process.
  - Threads from different processes lack prioritization.

Processes and threads

Threads implementation - Kernel support vs user-level

- User-level threads have benefits over kernel-level:
  - Thread operations can be cheaper.
  - Scheduling module can be customized.
  - Support for many user-level threads.

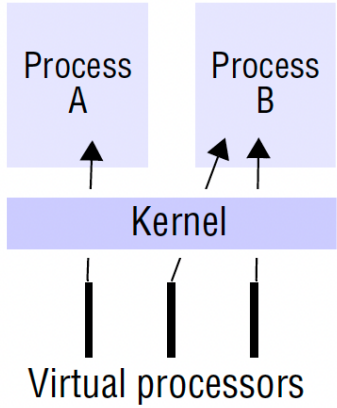
Processes and threads

Threads implementation - FastThreads

- Combine user-level and kernel-level advantages.
- Research on hierarchical scheduling for efficiency.
- Key implementation: FastThreads.

Processes and threads

FastThreads



A. Assignment of virtual processors to processes

- Includes a kernel and application programs.
- Each application has a user-level scheduler.
- Kernel allocates virtual processors based on needs and priorities.

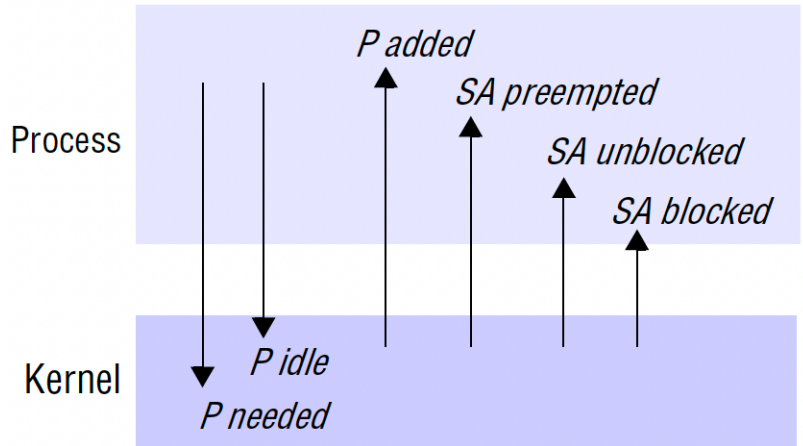
Processes and threads

Threads implementation - Activation scheduler

- Scheduler: Decides which threads run and when.
- Scheduler Activation (SA): OS interacts with the scheduler.
- Helps manage threads better, especially on multiple processors.

Processes and threads

Threads implementation - Activation scheduler

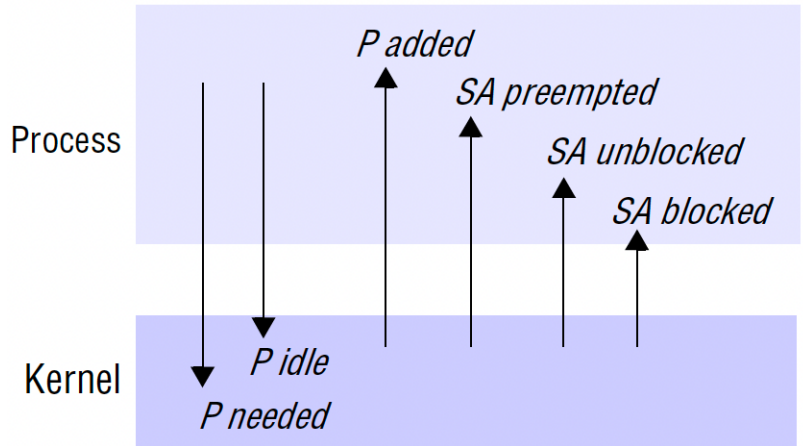


B. Events between user-level scheduler & kernel
Key: P = processor; SA = scheduler activation

- Processes can request or release virtual processors.
- Kernel informs processes about four event types.

Processes and threads

Threads implementation - Four types of events

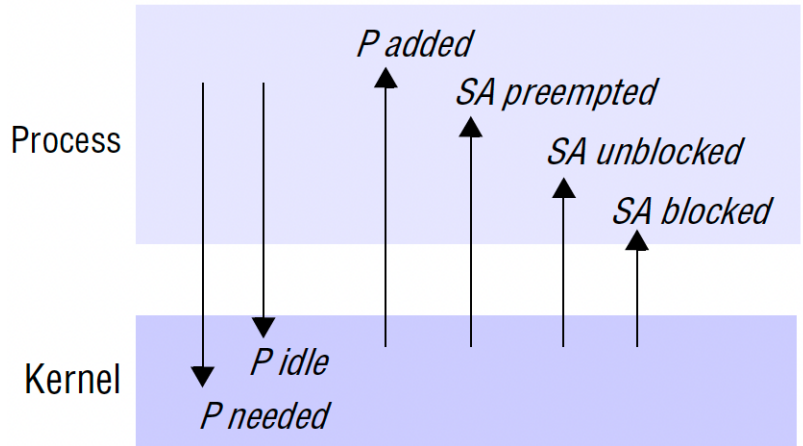


B. Events between user-level scheduler & kernel
Key: P = processor; SA = scheduler activation

- 1. Virtual processor allocated: New virtual processor assigned.
- 2. SA blocked: Thread cannot continue, notify scheduler.

Processes and threads

Threads implementation - Four types of events



B. Events between user-level scheduler & kernel
Key: P = processor; SA = scheduler activation

- 3. SA unblocked: Previously blocked thread can run again.
- 4. SA preempted: Running thread stopped to give time to another.

Communication and invocation and Communication primitives

- Communication in OSs as part of invocation.
- Invocation: Request action on a resource, usually not in the same place.
- Questions:
  - Communication primitives supplied
  - Supported protocols and openness
  - Steps for efficient communication
  - Support for high-latency operations

Communication and invocation and Communication primitives

- Certain kernels offer specific communication primitives.
- Higher-level functions mostly provided by middleware.
- Middleware typically uses sockets for interoperability.

Communication and invocation

Protocols and openness

- OS needs standard protocols for middleware compatibility.
- Early kernels had own protocols, not widely adopted.
- Recent kernels allow user choice of networking protocols.
- OS support for TCP and UDP essential for internet access.

Communication and invocation

Protocols and openness

- Protocols typically stacked in layers.
- Dynamic protocol composition adapts to application needs.
- Improves efficiency, especially in wireless contexts.

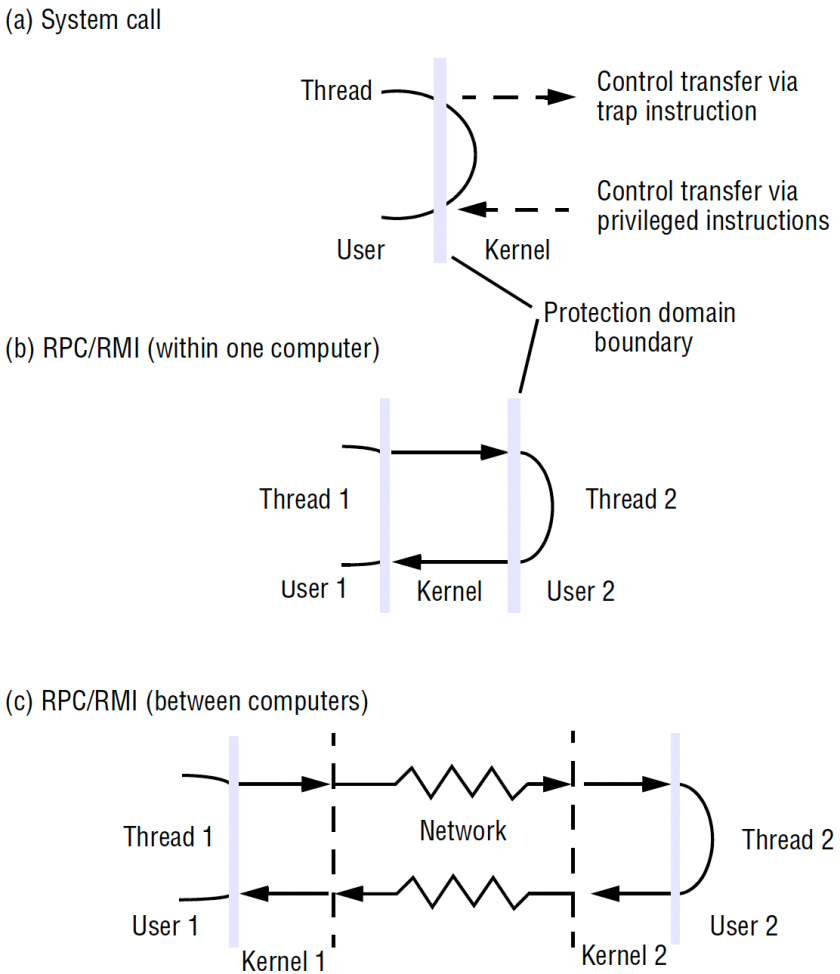
Communication and invocation

Invocation performance

- Invocation performance crucial in distributed system design.
  - More separation increases remote invocations.
  - Small delays matter due to high invocation frequency.
- Invocations over LANs focus on software overheads.
- Invocations over Internet have higher latencies.

Communication and invocation

Invocation costs



- Invocation examples:
  - Conventional procedures, system calls.
  - Remote invocations (RPC, RMI).
- Mechanisms may vary (e.g. synchronous or asynchronous etc).

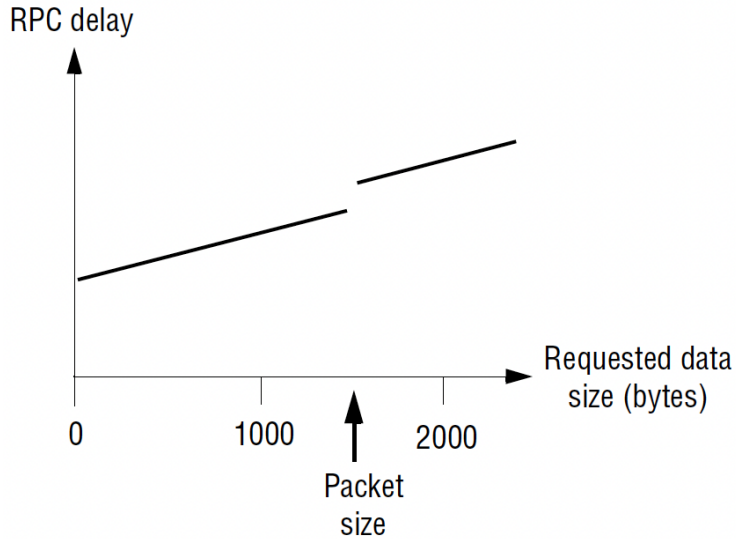
Communication and invocation

Invocation over the network

- A null RPC exchanges system data.
- Conventional procedure call: Fraction of a microsecond.
- Null RPC over LAN: About 0.1 milliseconds.
- Most RPC delays due to kernel actions and runtime code.

Communication and invocation

Invocation over the network



- Null invocation costs measure fixed overhead.
- Let an RPC fetch data from a server.
- Delay increases with size, beyond packet size threshold, more packets needed.

Communication and invocation

Invocation over the network

- RPC throughput reflects single data transfer rate.
- Throughput low for small data amounts.
- Throughput increases with larger data sizes.

Communication and invocation

Client and server operations in an RPC

- Client stub marshals arguments, sends request, receives and unmarshals reply.
- Server worker thread receives request, calls stub, processes, and sends reply.

Communication and invocation

Invocation delays

- Delays influenced by:
  - Un/marshalling overhead.
  - Data copying between spaces.
  - Packet initialization and header setup.
  - Thread scheduling and context switching.
  - Waiting for acknowledgements.
- Improving OS design and better memory management reduces overhead.

Communication and invocation

Memory sharing

- Memory sharing allows fast communication.
- Data shared via a shared region.
- Avoids copying data to kernel space.
- Synchronization needed for data access.
- Benefits of shared regions must outweigh setup costs.

Communication and invocation

Choice of protocol

- Delay during request-reply over TCP may not be worse than UDP.
- TCP can have less delay for large messages.
- TCP's buffering may negatively impact performance.
- TCP connection costs can make it less efficient than UDP for small requests.

Communication and invocation

Choice of protocol

- HTTP 1.0 creates separate TCP connection for each request.
- TCP's slow-start algorithm can delay data transfer.
- HTTP 1.1 uses persistent connections for multiple requests.
- Overriding OS buffering can reduce invocation delays.

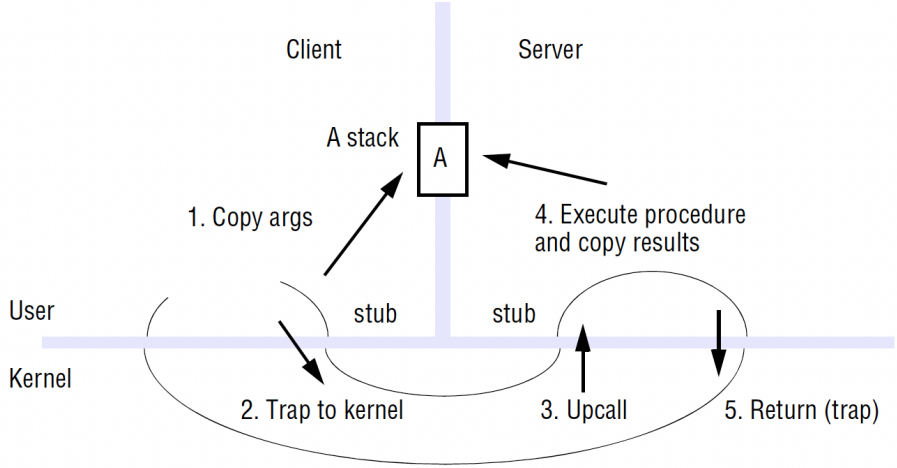
Communication and invocation

Invocation within a computer

- Most cross-address-space invocations occur within a computer.
- Cost of RPC within a computer growing in importance.
- Lightweight RPC (LRPC) introduced for efficiency.

Communication and invocation

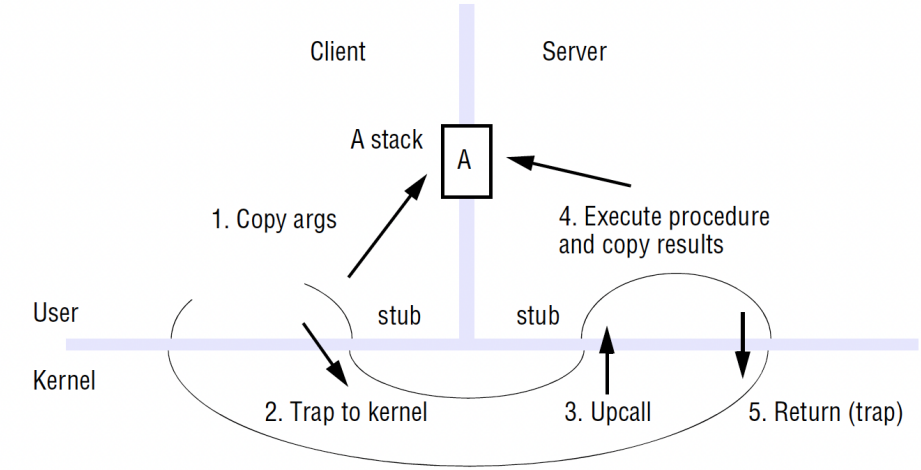
Invocation within a computer - LRPC - What does it optimize?



- LRPC optimizes data copies and scheduling.
- Shared memory allows direct argument passing.
- Arguments copied only once in LRPC.

Communication and invocation

Invocation within a computer - LRPC - Steps



- Client thread traps to kernel presenting it with a capability.
- Kernel switches thread context to server.
- Execution and result copying occur.
- Thread returns to kernel after execution.

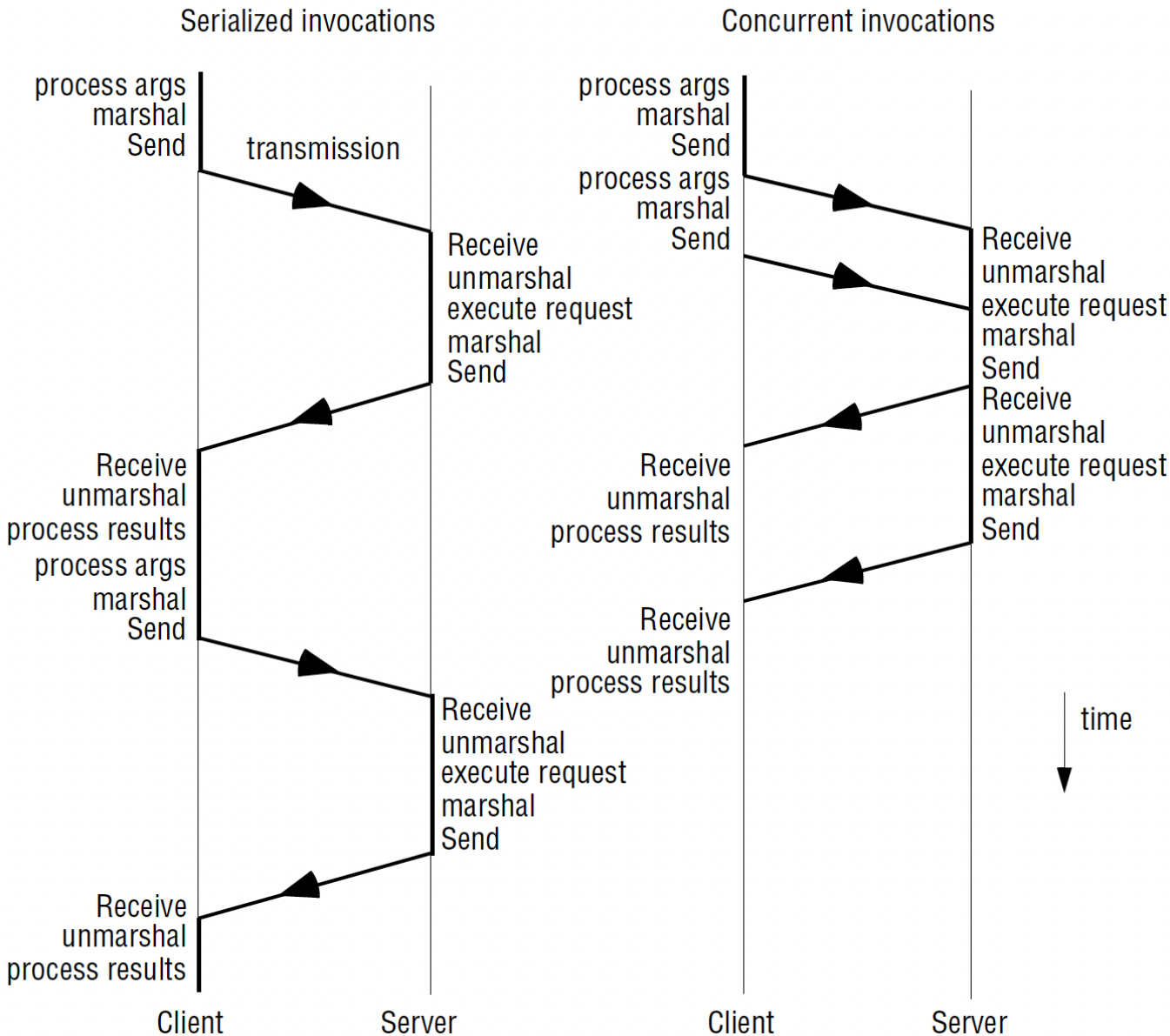
Asynchronous operation

Asynchronous operation

- OS aids middleware for efficient remote invocation.
- Asynchronous operation minimizes impact of delays.
- Two programming models:
  - Concurrent invocations
  - Asynchronous invocations

Asynchronous operation

Making invocations concurrently



Asynchronous operation

Making invocations concurrently

- Serialized: Client waits for response.
- Concurrent: Multiple threads send requests simultaneously.
- Reduces total communication delay compared to sequential requests.

Asynchronous operation

Asynchronous invocations

- Asynchronous invocation: Non-blocking call.
  - Synchronous waits for response.
  - Asynchronous continues with other tasks.
- Promise: Status of request without immediate response.
  - Acts as a placeholder for result.
  - Claim operation obtains result from promise.
  - Ready operation checks if promise holds a result.

Asynchronous operation

Persistent asynchronous invocations

- Traditional asynchronous invocation fails after timeouts.
- Persistent asynchronous invocation resists disconnections.
- Queues requests and responses until reconnect.
- Allows prioritization of requests.

Operating system architecture

- Openness: Ability to add or replace components.
- Key points of open distributed systems:
  - Run only necessary software.
  - Change services independently.
  - Alternatives for the same service.
  - Add new services without disruption.
- Separate fixed from variable resource management.
- Kernel provides basic mechanisms, with dynamic server modules.

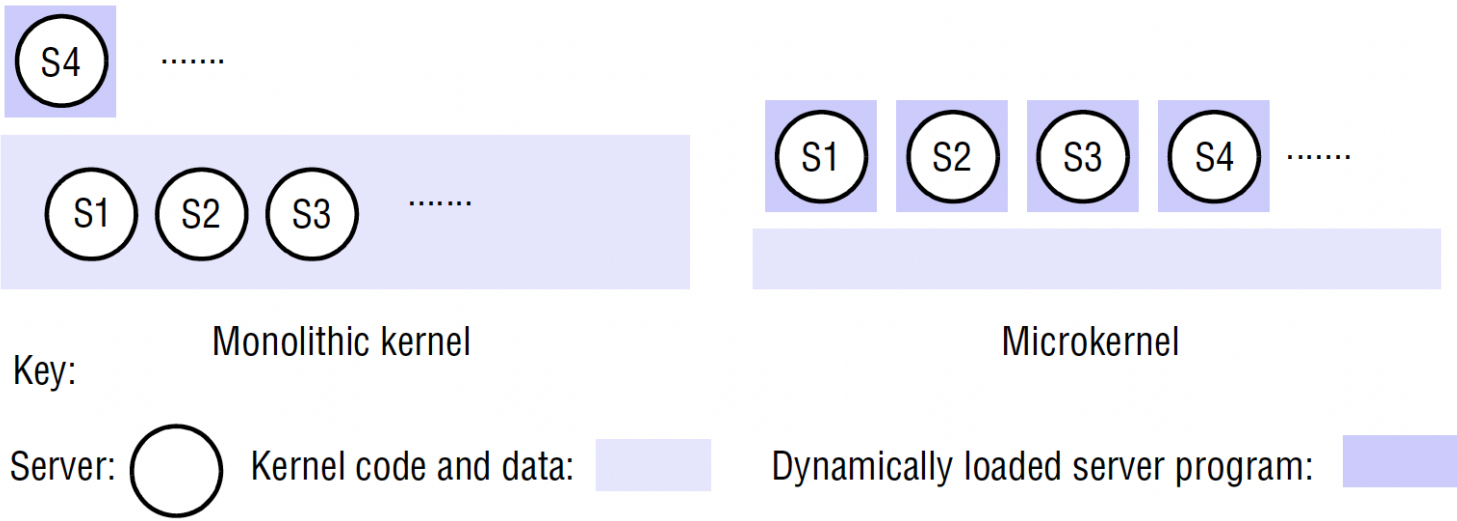
Operating system architecture

Monolithic kernels and microkernels

- Two main kernel designs: monolithic and microkernels.
- Monolithic kernels include all basic OS functions.
- Microkernels provide only essential functions.

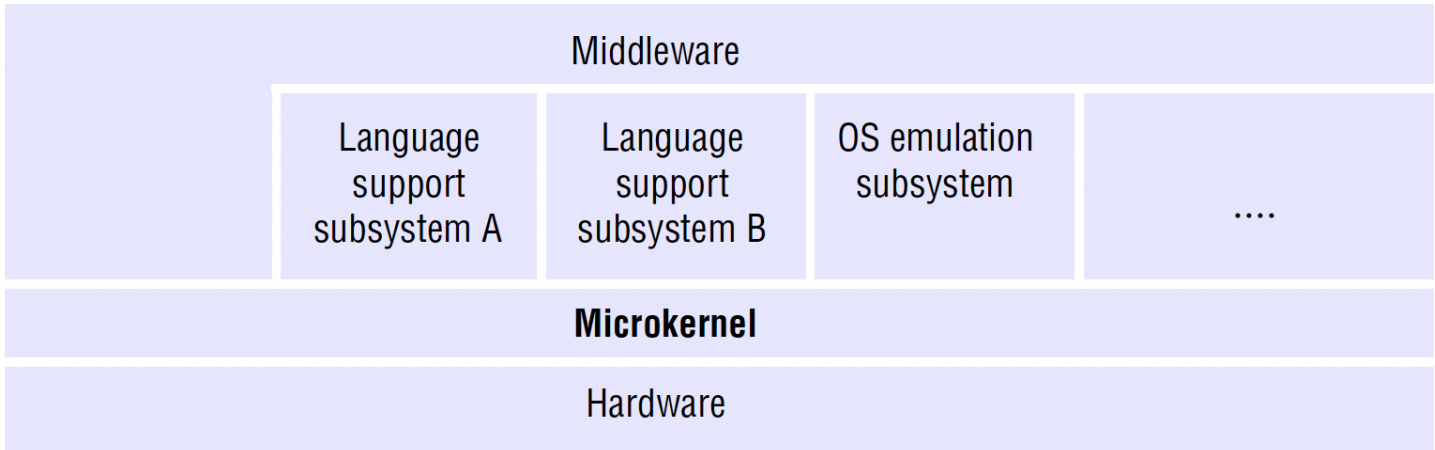
Operating system architecture

Monolithic kernels and microkernels



Operating system architecture

Monolithic kernels and microkernels



The microkernel supports middleware via subsystems

- Microkernels aim for extensibility and emulation.
- Act as a layer between hardware and subsystems.
- Middleware can interact with microkernel or use higher-level interfaces.

Operating system architecture

Comparison

- Microkernel-based systems are extensible and less prone to bugs.
- Monolithic designs are more efficient but costly in system calls.
- Large kernels are harder to maintain.
- Monolithic designs can use software engineering techniques.
- Windows uses a combination but remains less modular.

Operating system architecture

Some hybrid approaches

- Hybrid approaches trade efficiency for protection.
- Mach and Chorus microkernels run servers as user processes, but integrity can be in danger.
- SPIN, Nemesis are examples of hybrid approaches.

Virtualization at the operating system level

- Virtualization abstracts computing resources.
- Divides physical resources into multiple virtual resources.
- System virtualization creates virtual machines (VMs).
- Each VM runs its own OS and applications.

Virtualization at the operating system level

- Virtualization allows assignment of services to VMs.
- Reduces need for physical servers and energy consumption.
- Essential for cloud computing services.
- Enables easy creation and removal of VMs.

Virtualization at the operating system level

- VM monitor manages VMs, provides hardware interface.
- In full virtualization, VMs act like physical machines.
- Advantages: Run existing software without modification.
- Challenges: Achieving good performance.

Virtualization at the operating system level

- Paravirtualization offers a new interface to VMs.
- Requires OS changes for compatibility.
- Virtualization allows running multiple OSs on a single computer.

Virtualization at the operating system level

- Virtualization allows running multiple OSs on a single computer.
- Microkernels provide a small core and use external services.
- Microkernels can emulate various OS interfaces.

Virtualization at the operating system level

- Microkernel-based systems are extensible and modular.
- Monolithic systems are efficient but harder to maintain.
- Hybrid approaches try to balance efficiency and protection.
- Virtualization technology can enhance resource management.

Virtualization at the operating system level

- Virtualization vs. Microkernels:
  - Virtualization is a technology that allows you to run multiple OSs on a single computer.
  - Microkernels are a specific type of OS design. They allow multiple OSs to run together, but they do this by using a small "core" (the microkernel) that provides basic services.
- Functioning
  - In virtualization, the guest OS can run almost directly on the hardware of the computer, which means they often do not need many changes to work.
  - In microkernels, if you want to run an OS, it has to be "emulated" meaning the microkernel creates an environment that mimics what that OS needs to run. This can be complex and requires some extra effort.
- The biggest benefit of virtualization is that you can run applications without having to change or rewrite them.

Discussion topic

- What thread operations are the most significant in cost?

- Which factors identified in the cost of a remote invocation also feature in message passing?

Discussion topic

Network transmission time accounts for 20% of a null RPC and 80% of an RPC that transmits 1024 user bytes (less than the size of a network packet). By what percentage will the times for these two operations improve if the network is upgraded from 10 megabits/second to 100 megabits/second?

Discussion topic

A null RMI that takes no parameters, calls an empty procedure and returns no values delays the caller for 2.0 milliseconds. Explain what contributes to this time.

In the same RMI system, each 1K of user data adds an extra 1.5 milliseconds. A client wishes to fetch 32K of data from a file server. Should it use one 32K RMI or 32 1K RMIs