Operating System support

Question

What is the role of the operating system(s) in a distributed system?

Agenda

The operating system layer

Protection of resources

Processes and threads

Communication and invocation

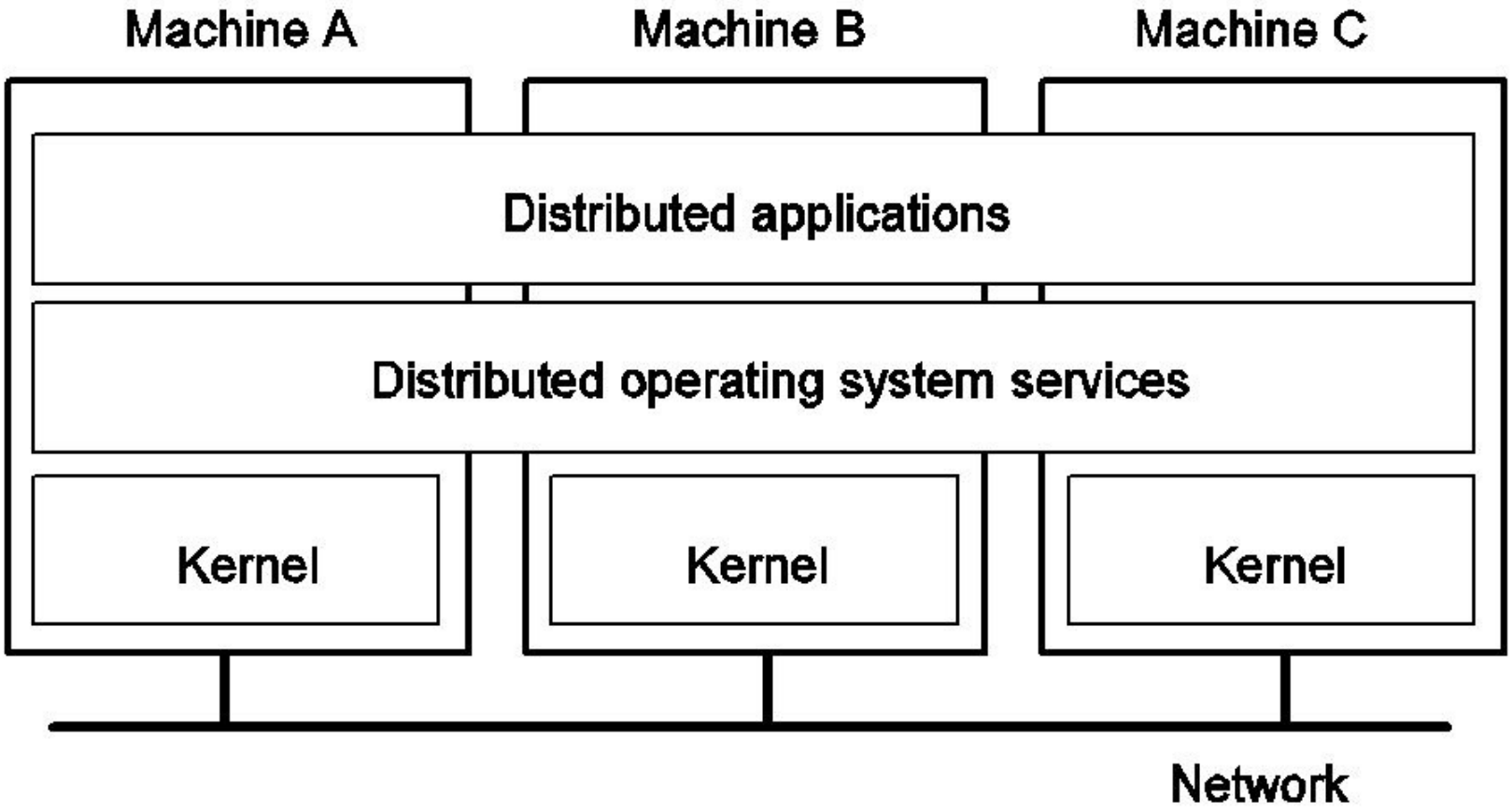
Operating system architecture

Virtualization at the operating system level

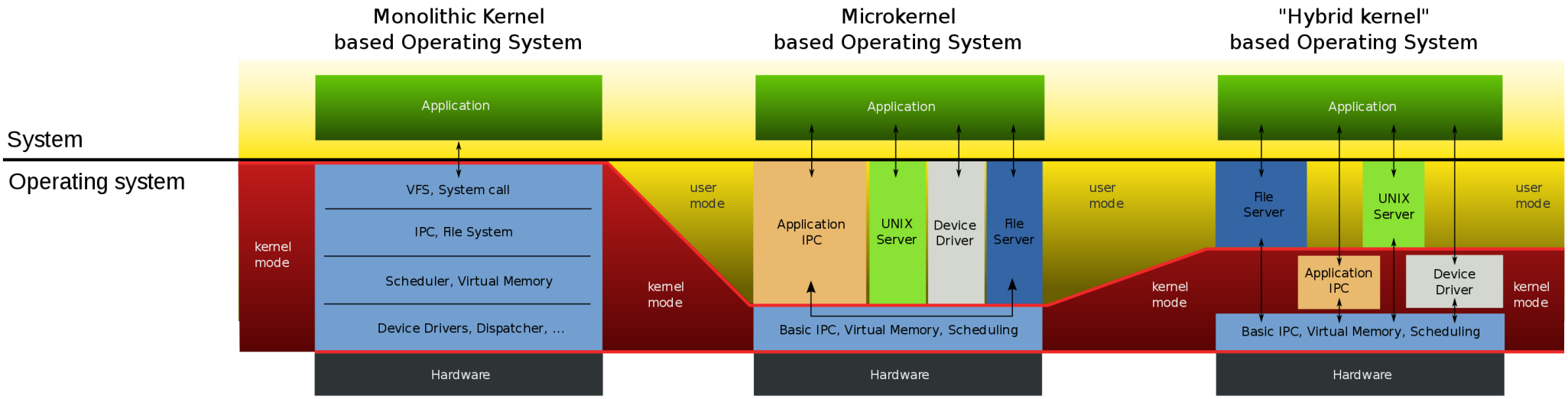
Networked OS vs distributed OS

- Networked OS: Linux, Windows, MacOS. They have networking capabilities built inside. They all manage their own resources
- Distributed OS: offers a single system image of a set of computing agents. Examples are Plan9 and Inferno (both from Bell Labs)

Distributed Operating Systems (DOS)



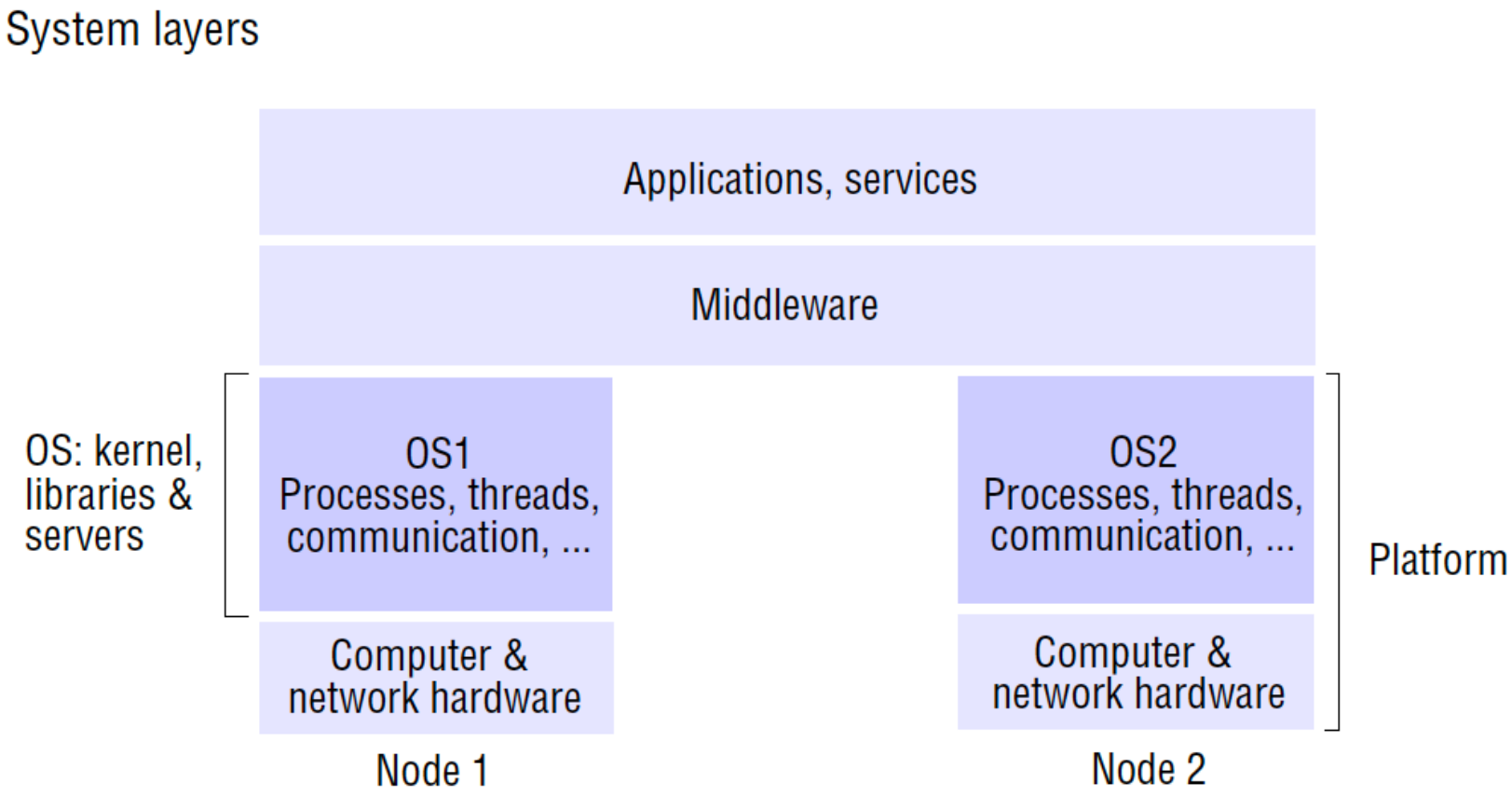
OS architectures



OS support for distributed objects

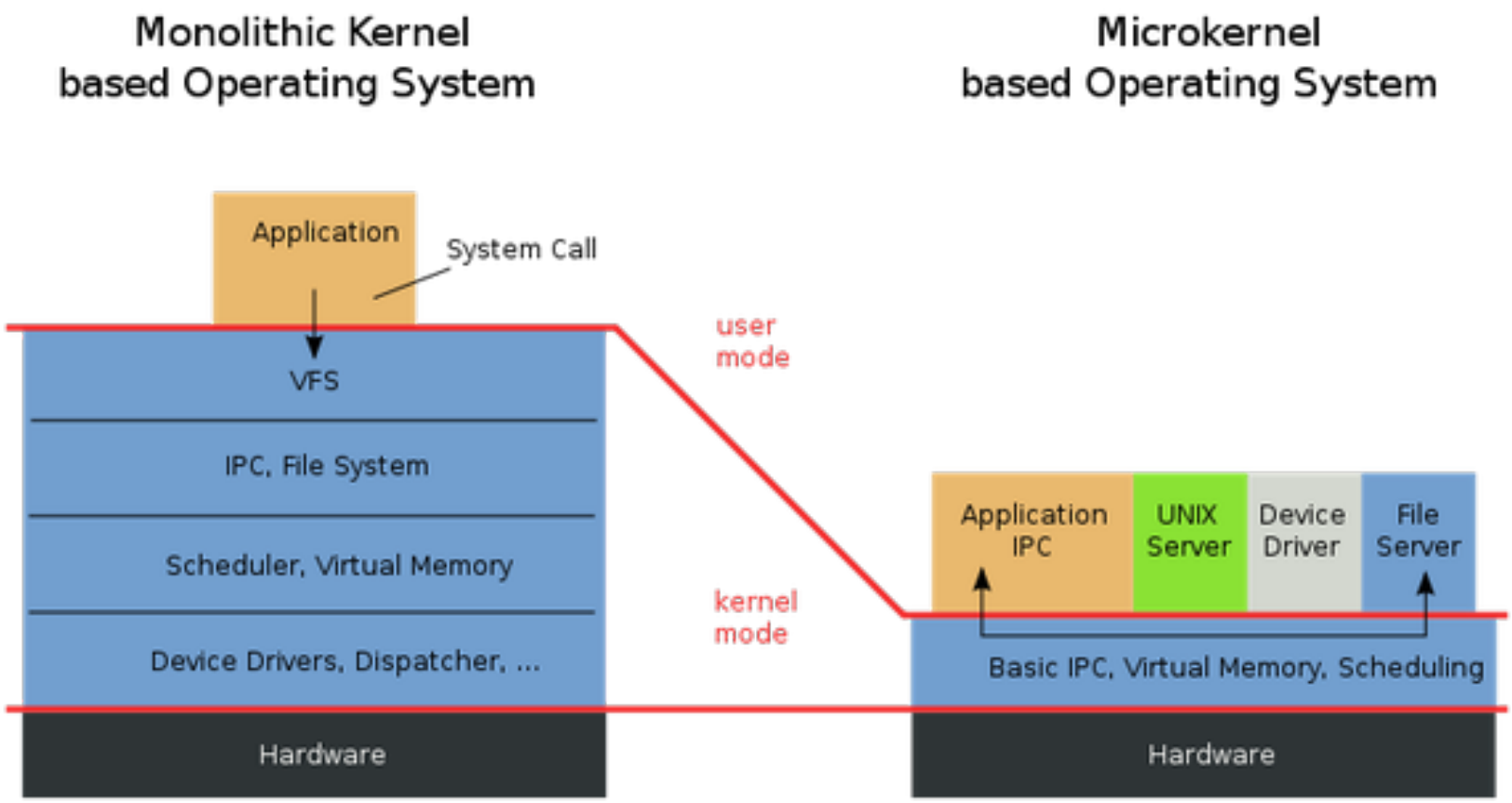
- In a single computer the operating system encapsulates and protects resources inside servers
- It offers the mechanisms required to access these resources, including communication and scheduling.
- We know the advantages and disadvantages of splitting functionality between protection domains – in particular, of splitting functionality between kernel- and user-level code.
- The trade-offs between kernel-level facilities and user-level facilities include a tension between efficiency and robustness

System layers



The operating system layer at each of two nodes supports a common middleware layer in providing a distributed infrastructure for applications and services.

Kernel patterns



https://www.quora.com/What-is-the-difference-between-a-monolithic-kernel-and-microkernel

Monolithic Kernel Vs Microkernel

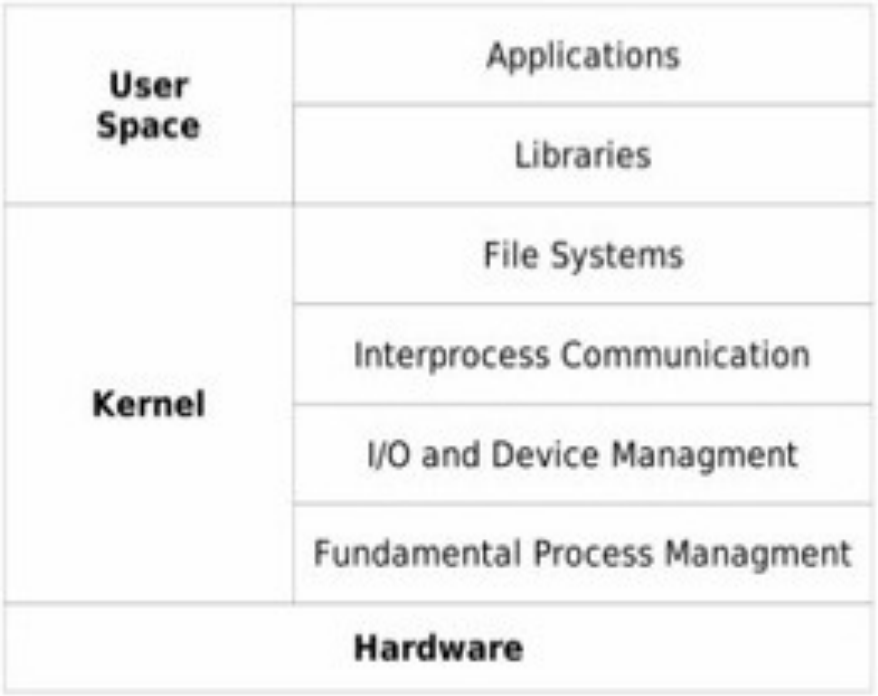


Figure 1: Monolithic kernel based operating system

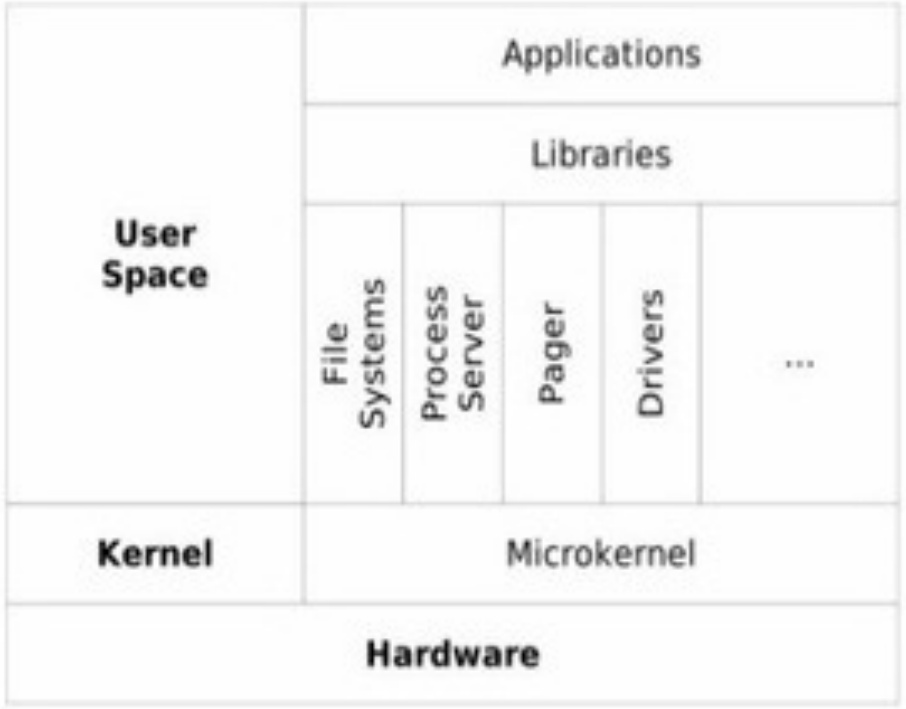


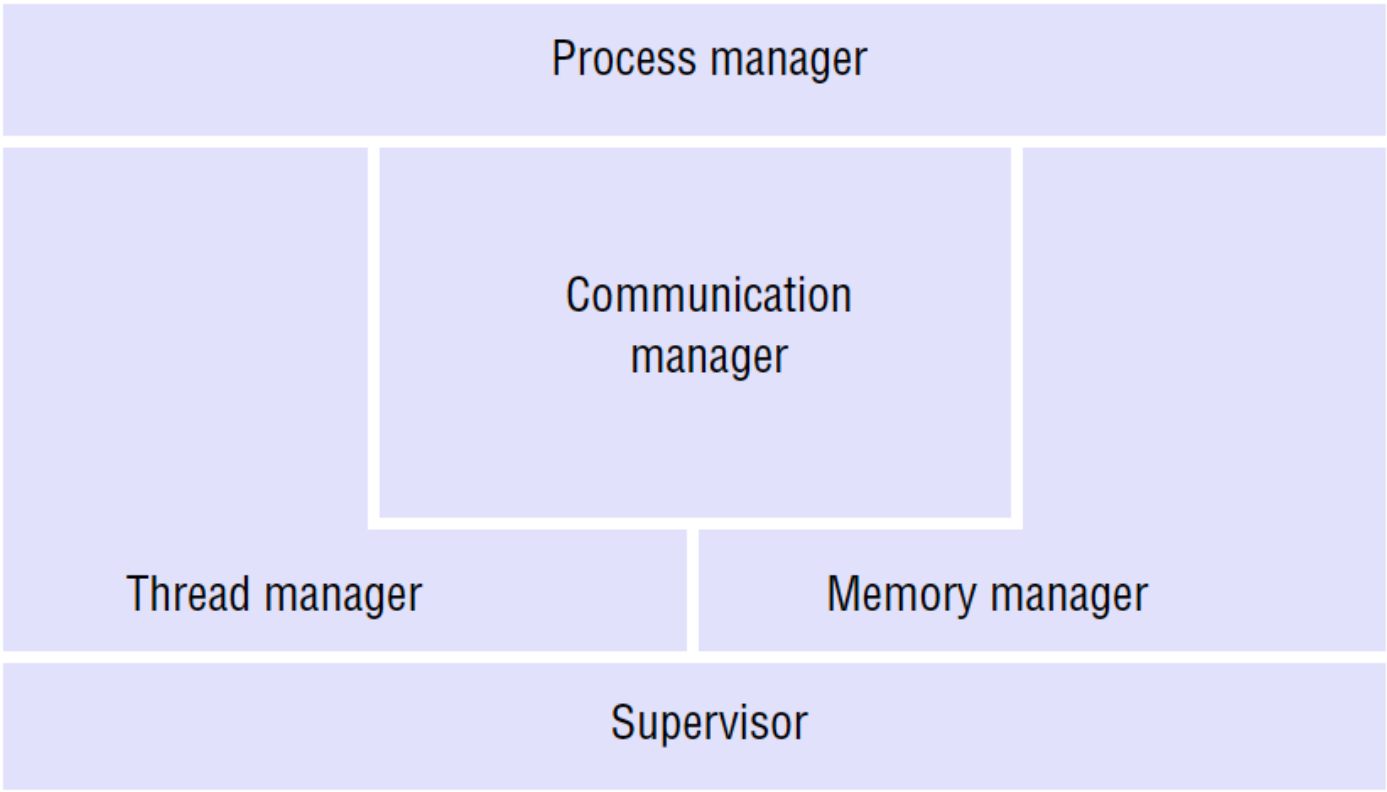
Figure 2: Microkernel based operating system

Architectural issues

Kernels and server processes are the components that manage resources and present clients with an interface to the resources. Their requirements:

- Encapsulation : They should provide a useful service interface to their resources- a set of operations that meet their clients' needs. Management of memory and devices used to implement resources should be hidden from clients.
- Protection : Resources require protection from illegitimate accesses – for example, files are protected from being read by users without read permissions, and device registers are protected from application processes.
- Concurrent processing : Clients may share resources and access them concurrently. Resource managers are responsible for achieving concurrency transparency

Core OS functionality

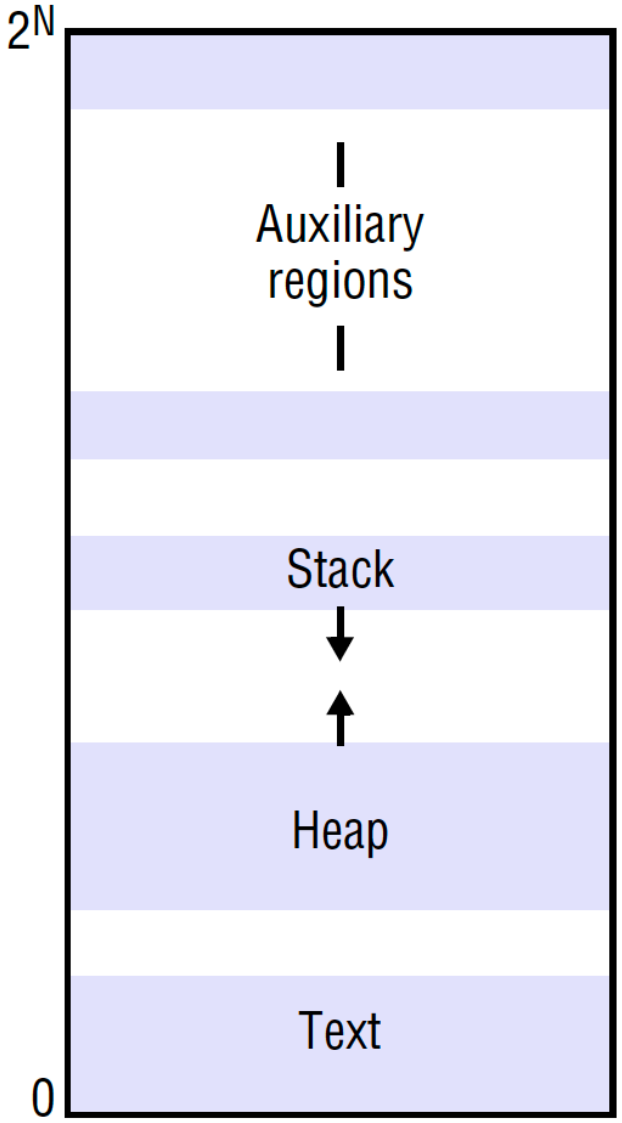


the core OS functions: process and thread management, memory management and communication between processes on the same computer (horizontal divisions in the figure denote dependencies).
The kernel supplies much of this functionality – all of it in the case of some operating systems

Address space

This representation of an address space as a sparse set of disjoint regions is a generalization of the UNIX address space, which has three regions:

- a fixed, unmodifiable text region containing program code;
- a heap, part of which is initialized by values stored in the program's binary file, and which is extensible towards higher virtual addresses;
- a stack, which is extensible towards lower virtual addresses



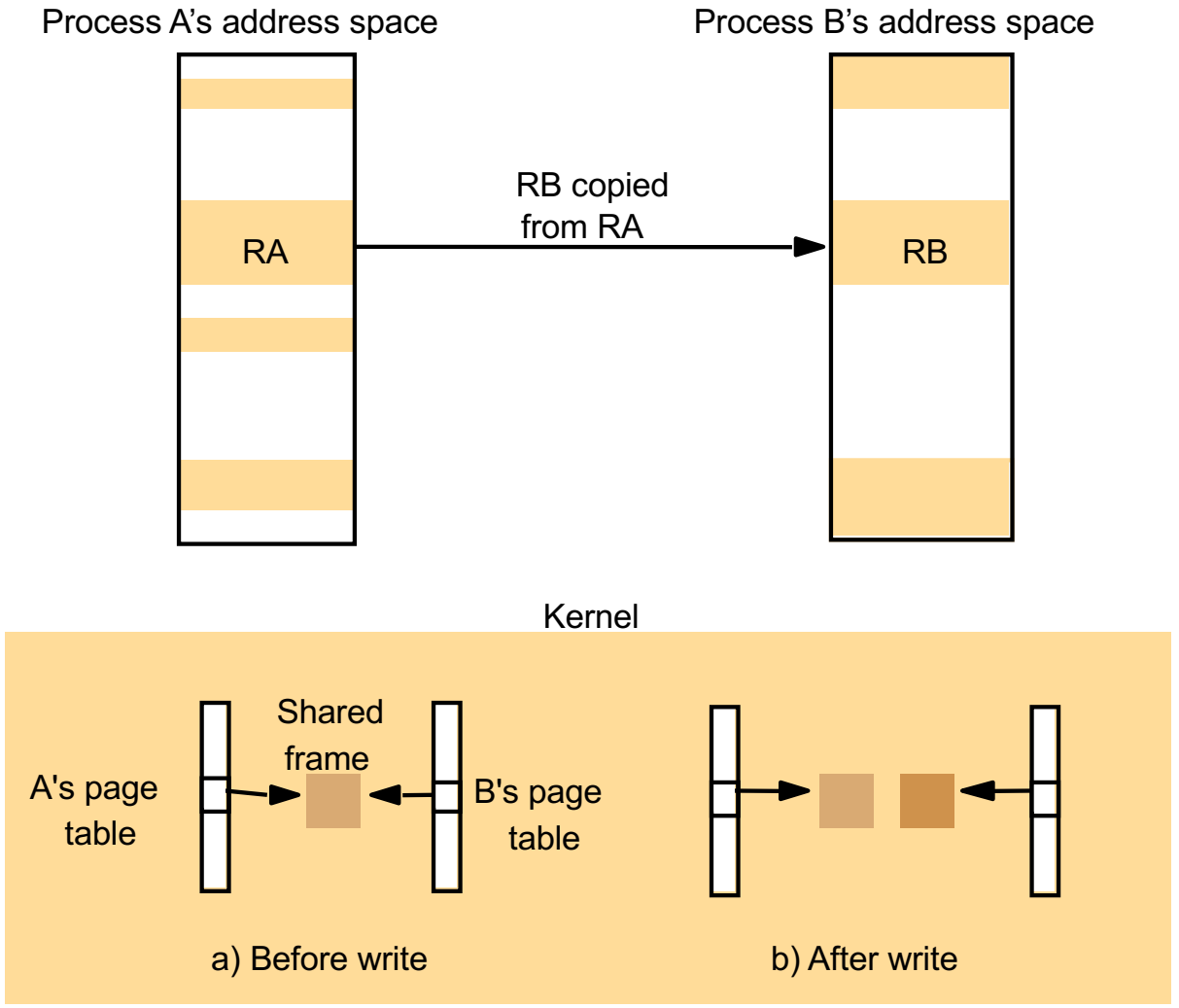
Fork

- In the case of UNIX *fork* semantics a newly created child process physically shares the parent's text region and has heap and stack regions that are copies of the parent's (as well as in initial contents).
- This scheme has been generalized so that each region of the parent process may be inherited by (or omitted from) the child process.
- An inherited region may either be shared with or logically copied from the parent's region.
- When parent and child share a region, the page frames (units of physical memory corresponding to virtual memory pages) belonging to the parent's region are mapped simultaneously into the corresponding child region.

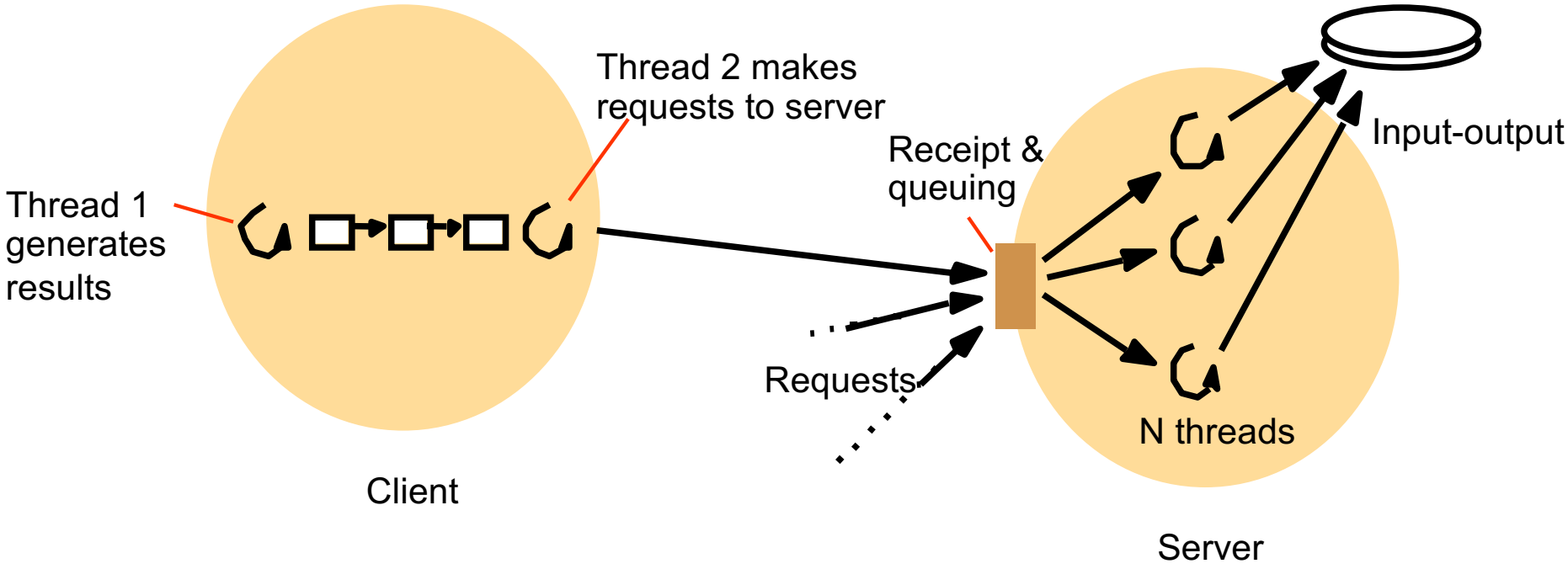
Copy-on-write

- Mach and Chorus apply an optimization called copy-on-write when an inherited region is copied from the parent.
- The region is copied, but no physical copying takes place by default.
- The page frames that make up the inherited region are shared between the two address spaces.
- A page in the region is only physically copied when one or another process attempts to modify it.

Copy-on-write



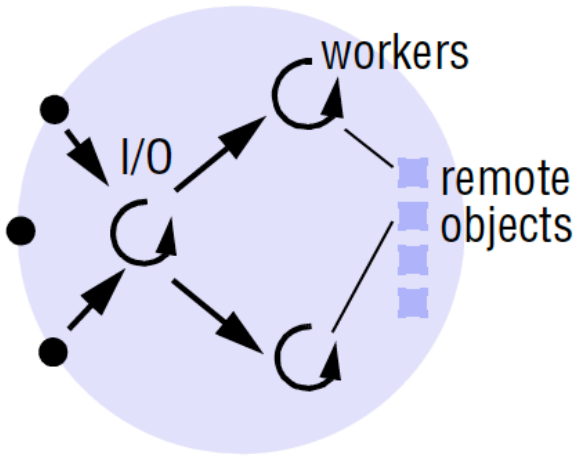
Client and server with threads



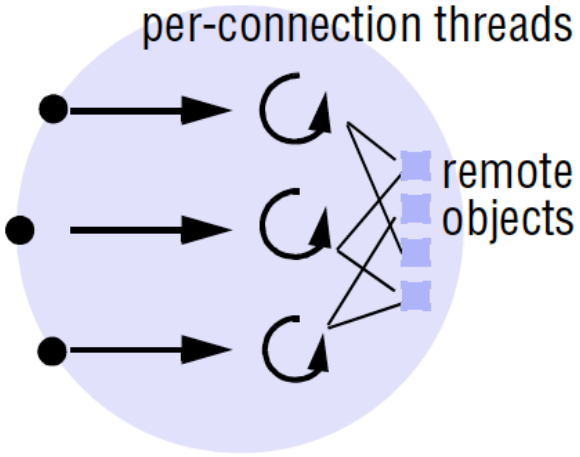
The server has a pool of one or more threads, each of which repeatedly removes a request from a queue of received requests and processes it

We consider the maximum server throughput, measured in client requests handled per second, for different numbers of threads

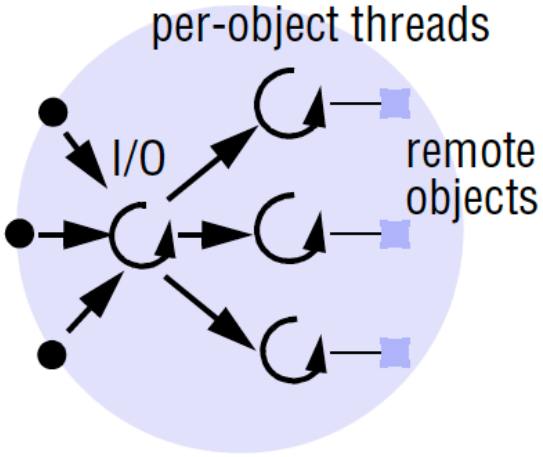
Alternative server threading architectures



a. Thread-per-request



b. Thread-per-connection



c. Thread-per-object

State associated with execution environments and threads

Execution environment	*Thread*
Address space tables	Saved processor registers
Communication interfaces, open files	Priority and execution state (such as *BLOCKED*)
Semaphores, other synchronization objects	Software interrupt handling information
List of thread identifiers	Execution environment identifier
Pages of address space resident in memory; hardware cache entries	

Process vs thread

We can summarize a comparison of processes and threads as follows:

- Creating a new thread within an existing process is cheaper than creating a process.
- More importantly, switching to a different thread within the same process is cheaper than switching between threads belonging to different processes.
- Threads within a process may share data and other resources conveniently and efficiently compared with separate processes.
- But, by the same token, threads within a process are not protected from one another.

Java thread constructor and management methods

Thread(ThreadGroup group, Runnable target, String name)

Creates a new thread in the *SUSPENDED* state, which will belong to *group* and be identified as *name*; the thread will execute the *run()* method of *target*.

setPriority(int newPriority), getPriority()

Set and return the thread's priority.

run()

A thread executes the *run()* method of its target object, if it has one, and otherwise its own *run()* method (*Thread* implements *Runnable*).

start()

Change the state of the thread from *SUSPENDED* to *RUNNABLE*.

sleep(int millisecs)

Cause the thread to enter the *SUSPENDED* state for the specified time.

yield()

Causes the thread to enter the *READY* state and invoke the scheduler.

destroy()

Destroy the thread.

Java thread synchronization calls

thread.join(int millisecs)

Blocks the calling thread for up to the specified time until *thread* has terminated.

thread.interrupt()

Interrupts *thread*: causes it to return from a blocking method call such as *sleep()*.

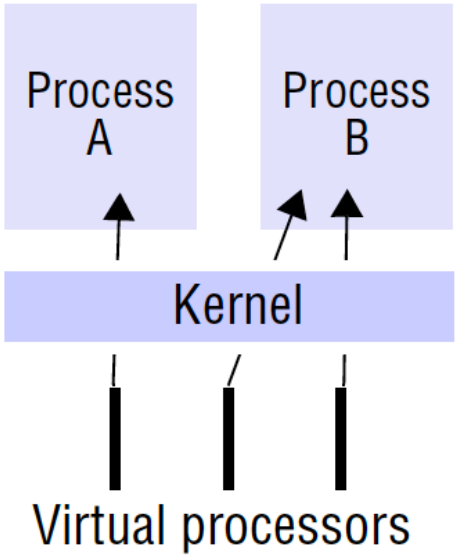
object.wait(long millisecs, int nanosecs)

Blocks the calling thread until a call made to *notify()* or *notifyAll()* on *object* wakes the thread, or the thread is interrupted, or the specified time has elapsed.

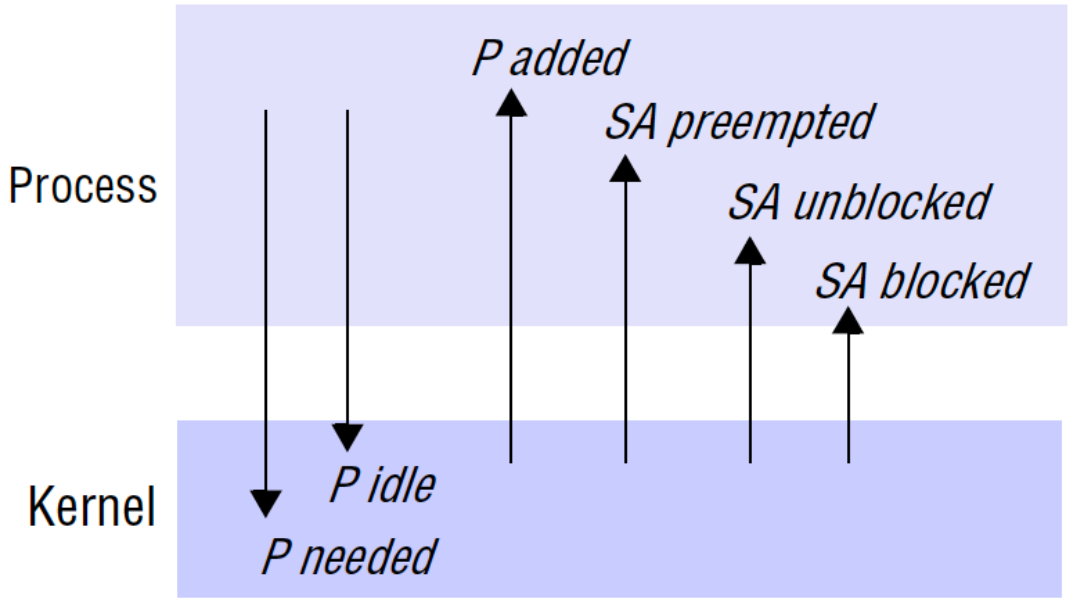
object.notify(), object.notifyAll()

Wakes, respectively, one or all of any threads that have called *wait()* on *object*.

Scheduler activations

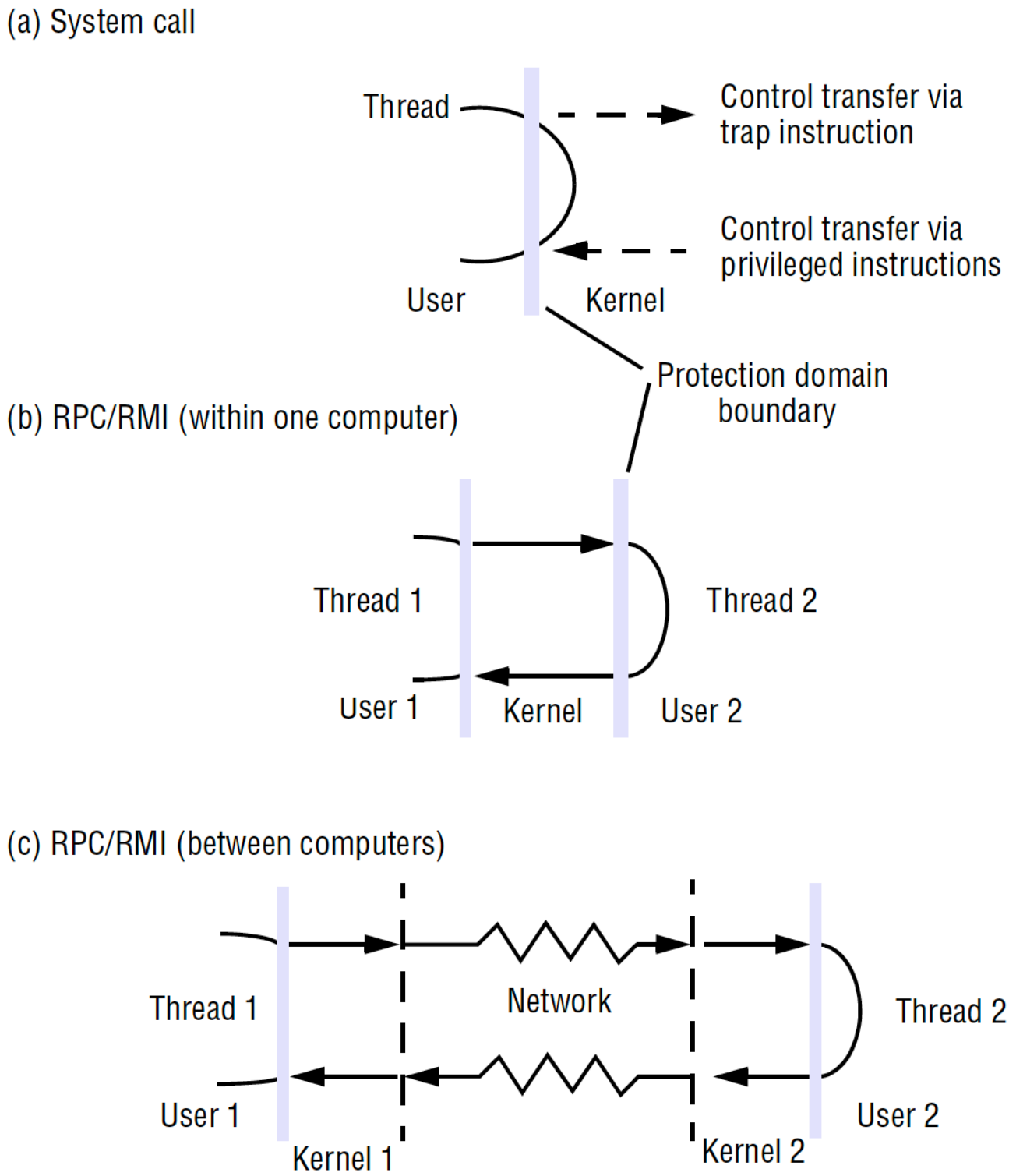


A. Assignment of virtual processors to processes

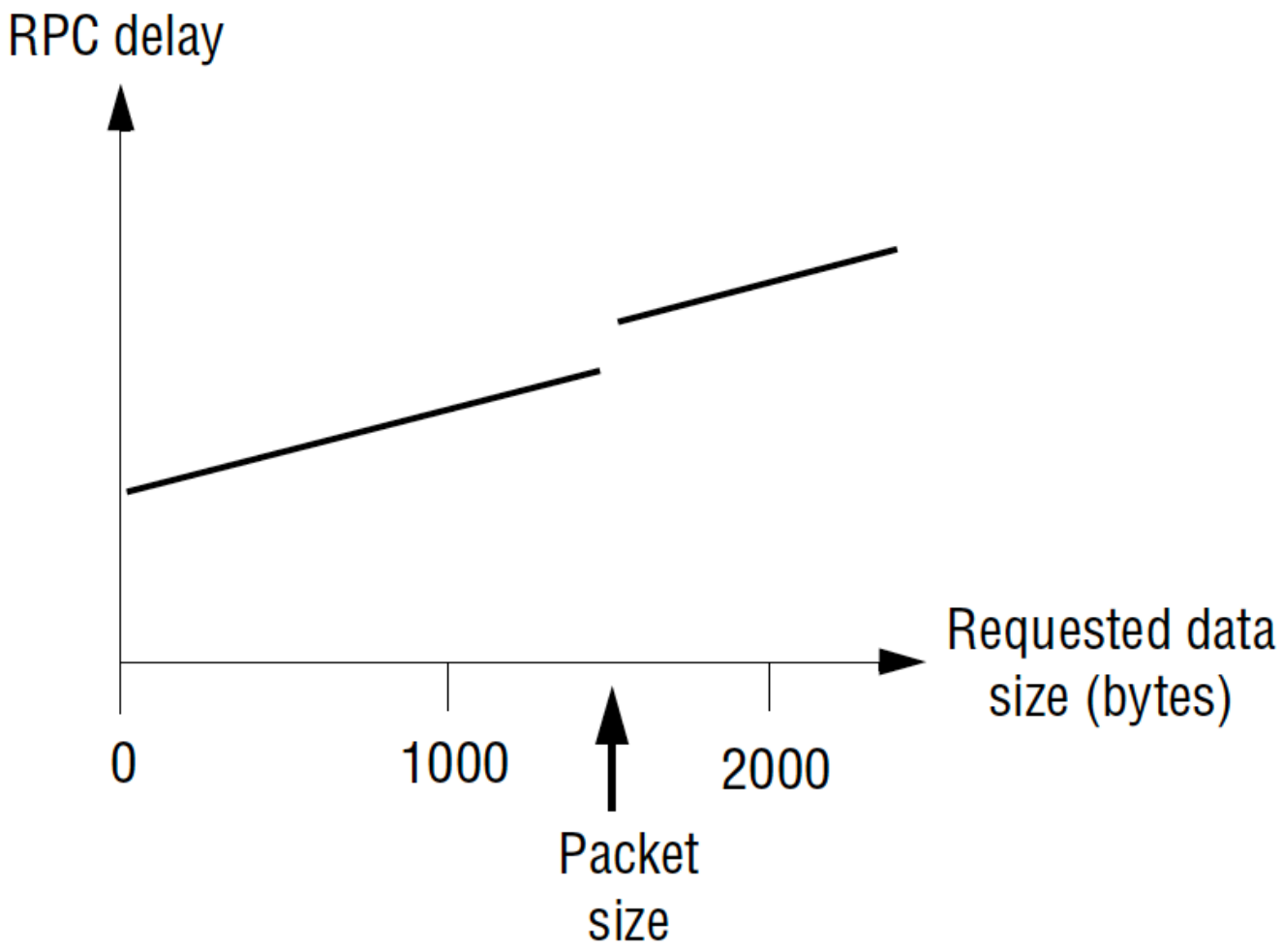


B. Events between user-level scheduler & kernel
Key: P = processor; SA = scheduler activation

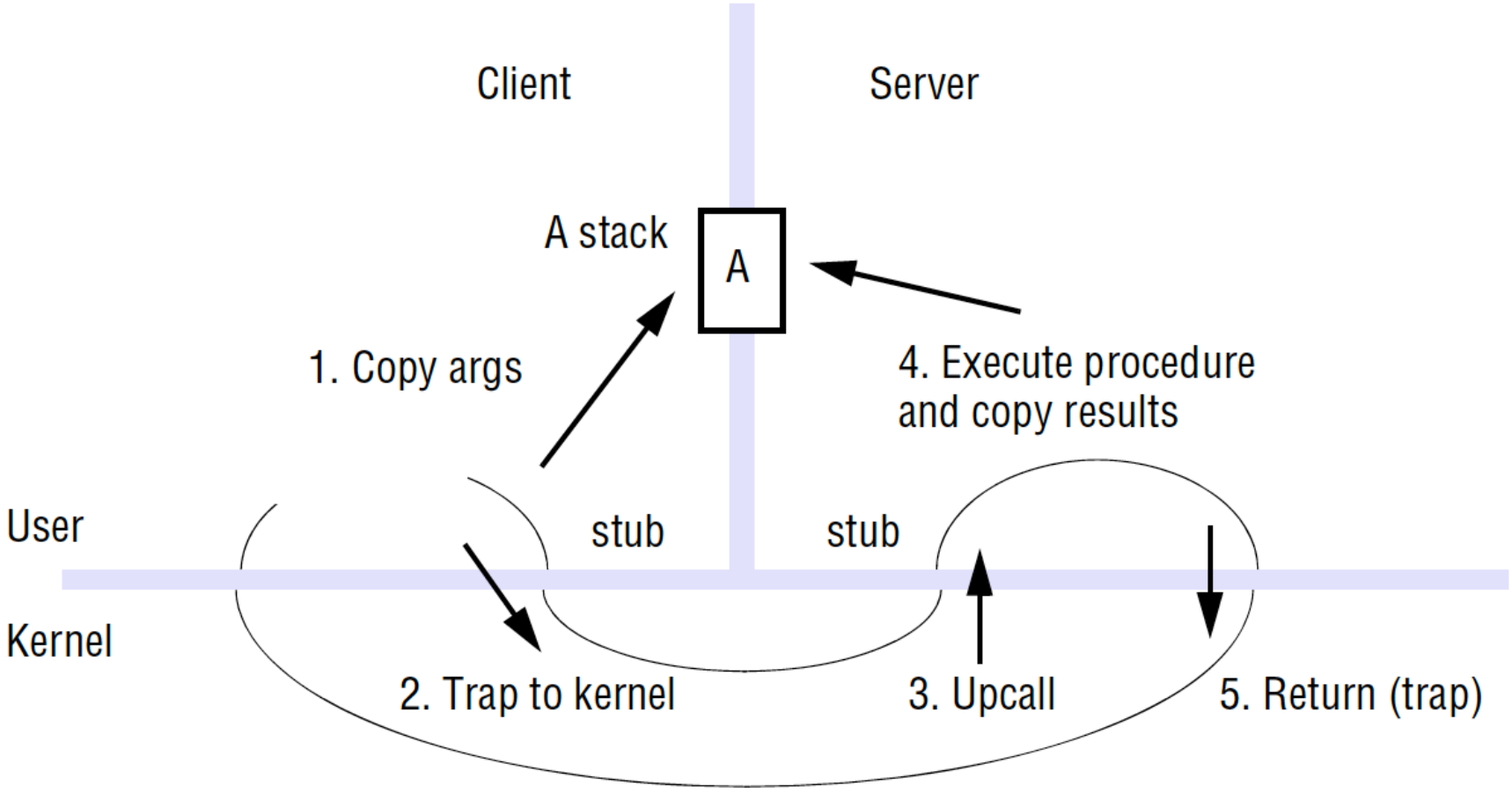
Invocations between address spaces



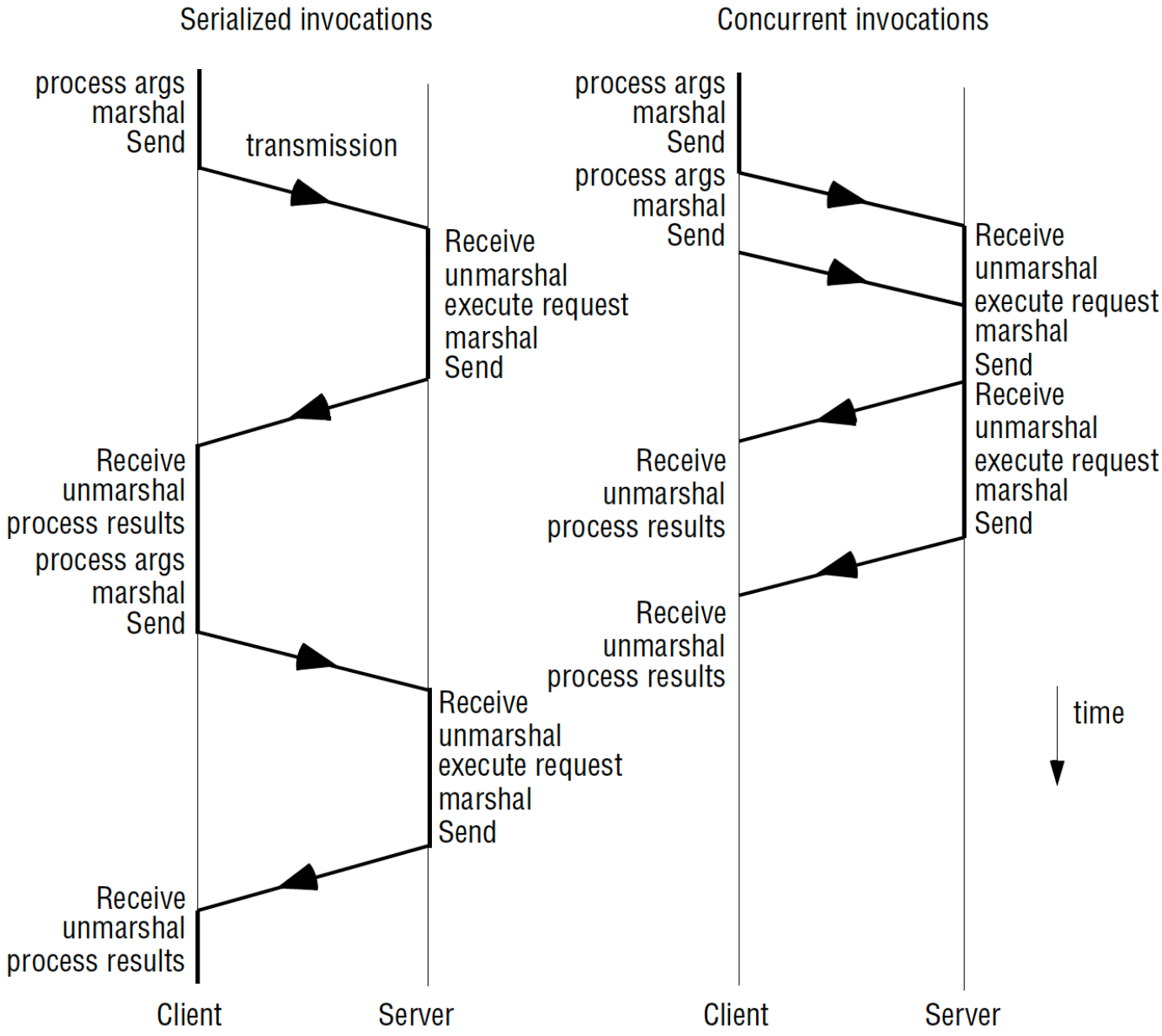
RPC delay against parameter size



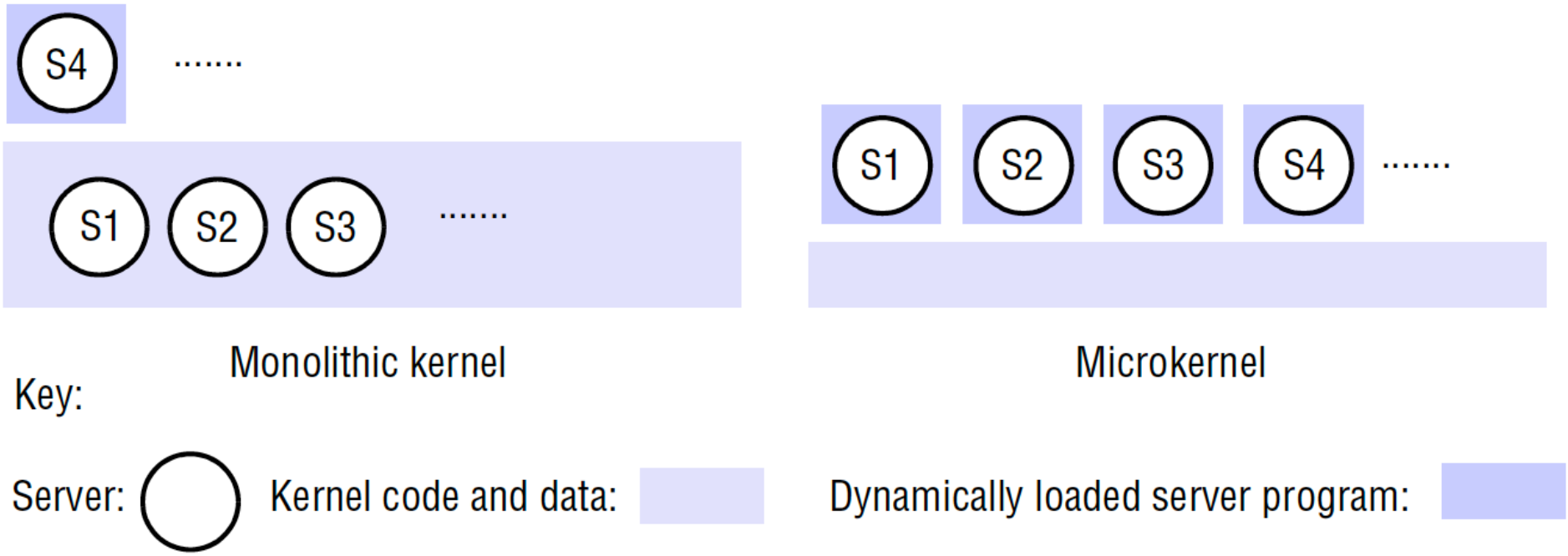
A lightweight remote procedure call



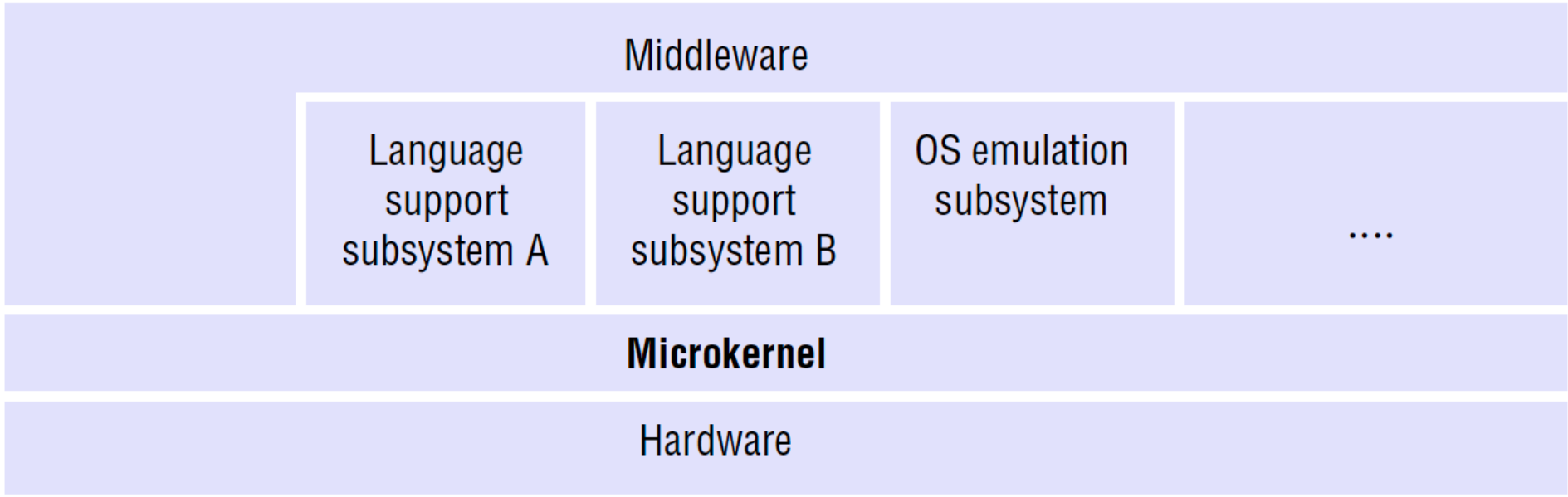
Times for serialized and concurrent invocations



Monolithic kernel and microkernel



The role of the microkernel



The microkernel supports middleware via subsystems

Server virtualization

Server virtualization can provide benefits such as:

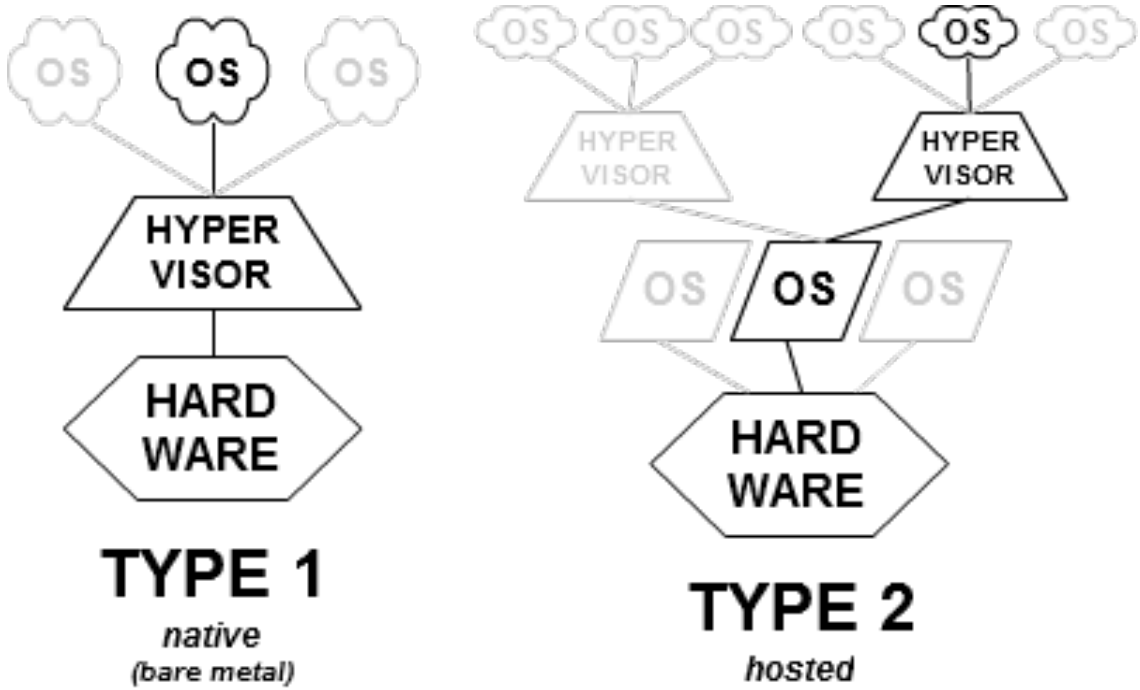
- consolidation leading to increased utilization
- rapid provisioning of additional resources
- dynamic fault tolerance against software failures (through rapid bootstrapping or rebooting)
- hardware fault tolerance (through migration of a virtual machine to different hardware)
- the ability to securely separate virtual operating systems
- the ability to support legacy software as well as new OS instances on the same computer

XEN

- Xen is a leading example of system virtualization, initially developed as part of the Xenoserver project at the Computer Laboratory, Cambridge University and now maintained by an open source community [www.xen.org].
- An associated company, XenSource, was acquired by Citrix Systems in 2007
- Since 2013 Xen is being developed by the Linux Foundation with support from Intel.

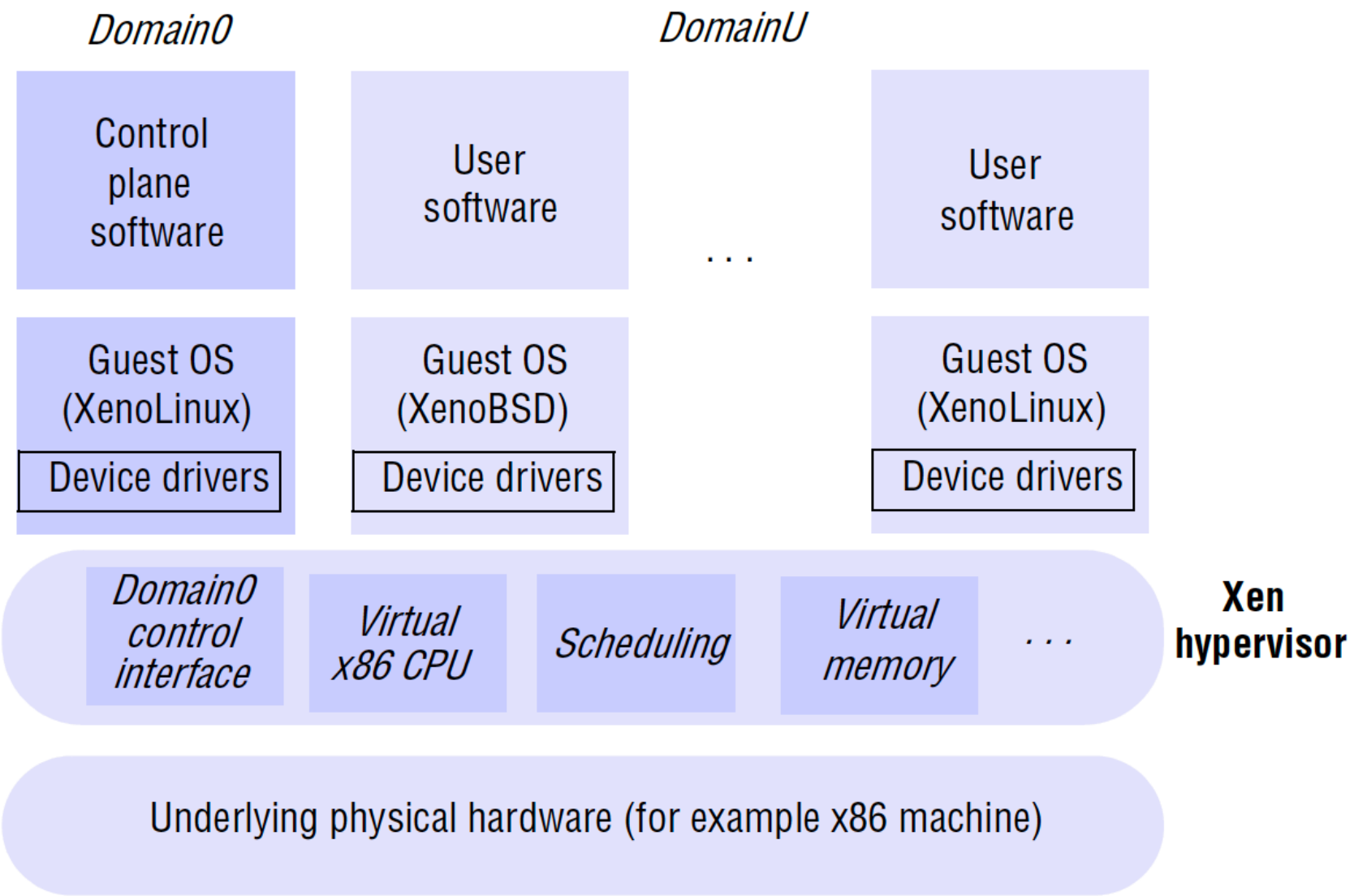
- https://xenproject.org

Hypervisors

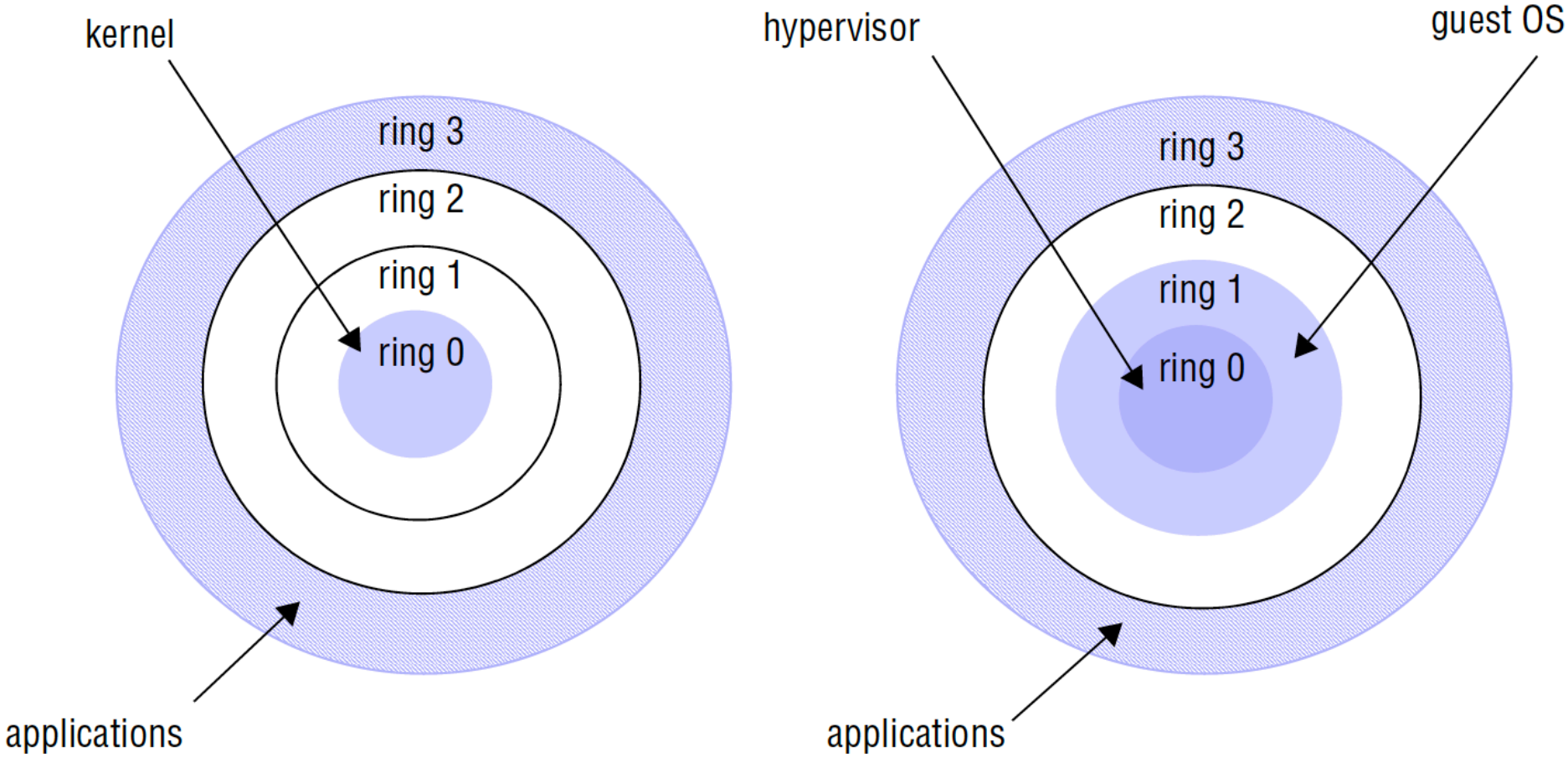


Type 1, native, hypervisors run directly on the host's hardware to control the hardware and to manage guest operating systems.
Example: Xen
Type 2 hypervisors run on a conventional operating system (OS) just as other computer programs do. A guest operating system runs as a process on the host. Type-2 hypervisors abstract guest operating systems from the host operating system. Examples: VMware, Parallels Desktop for Mac

The architecture of Xen



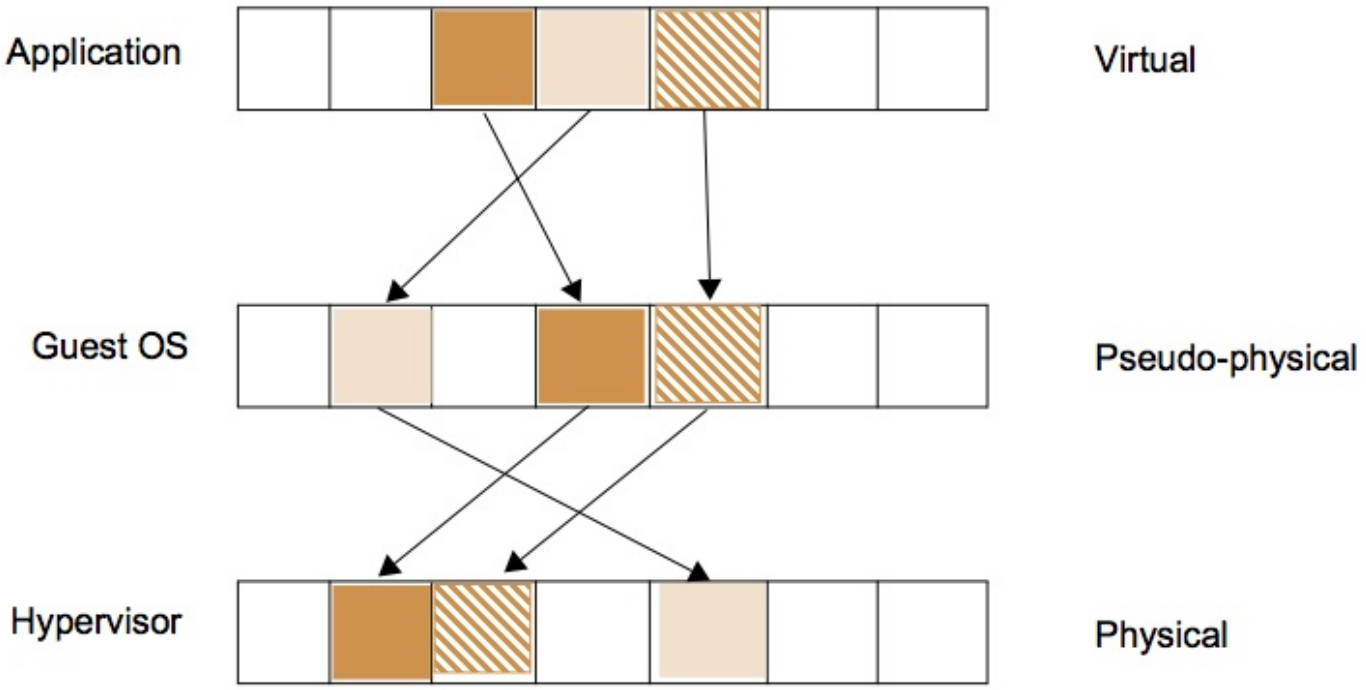
Use of rings of privilege



a) kernel-based operating systems

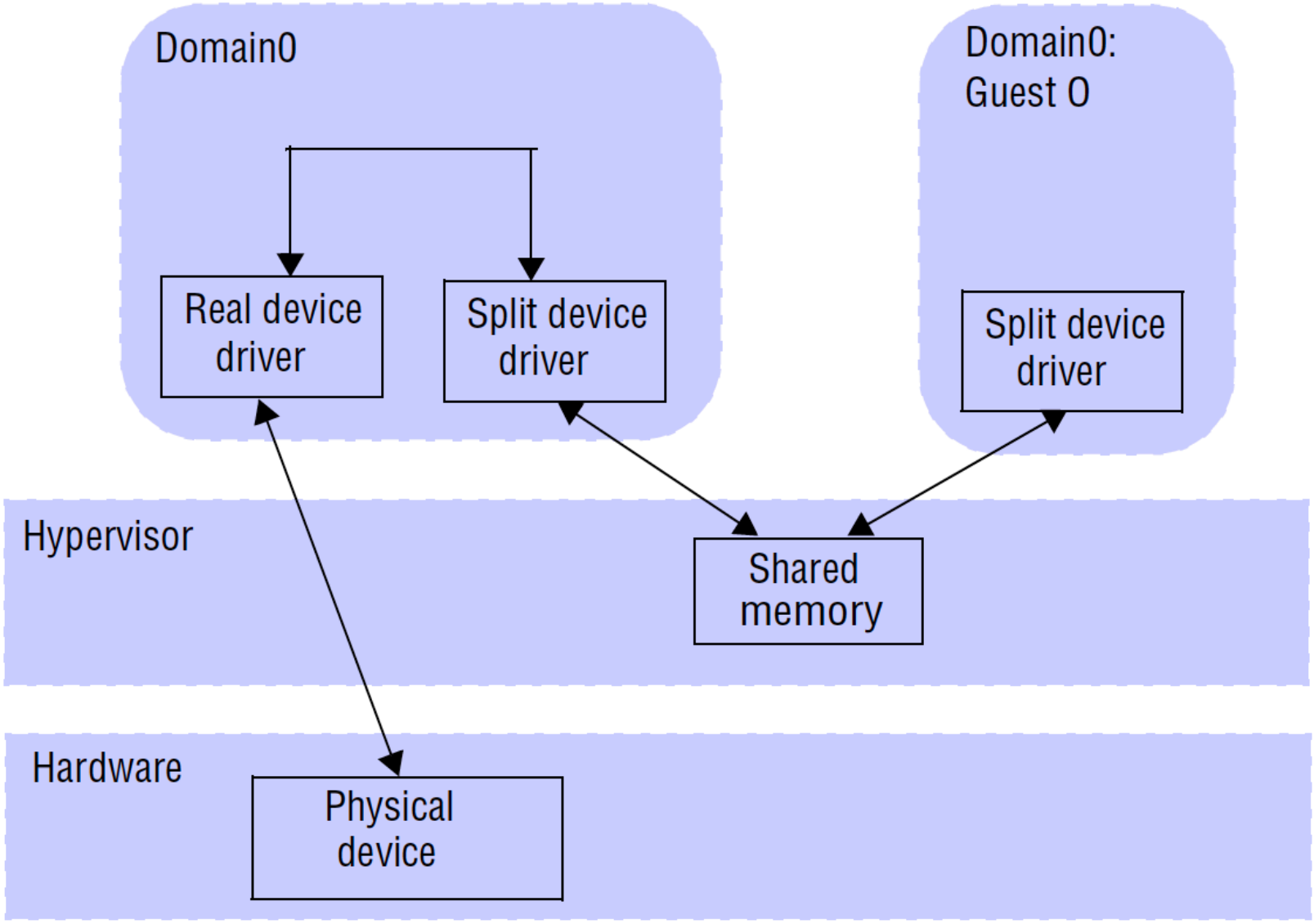
b) paravirtualization in Xen

Virtualization of memory management

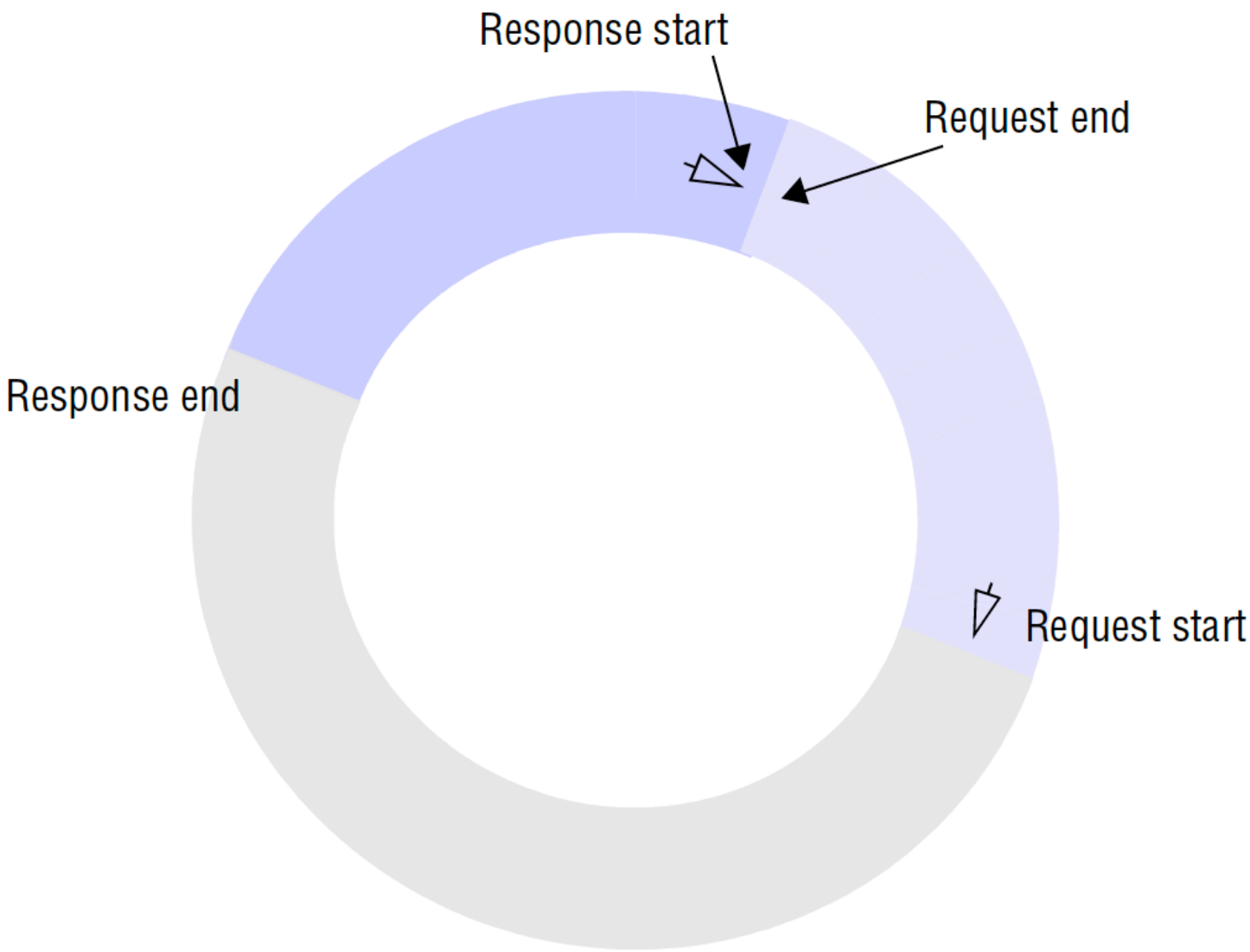


The role of the pseudo-physical memory is to provide this abstraction by offering a contiguous pseudo-physical address space and then maintaining a mapping from this address space to the real physical addresses.

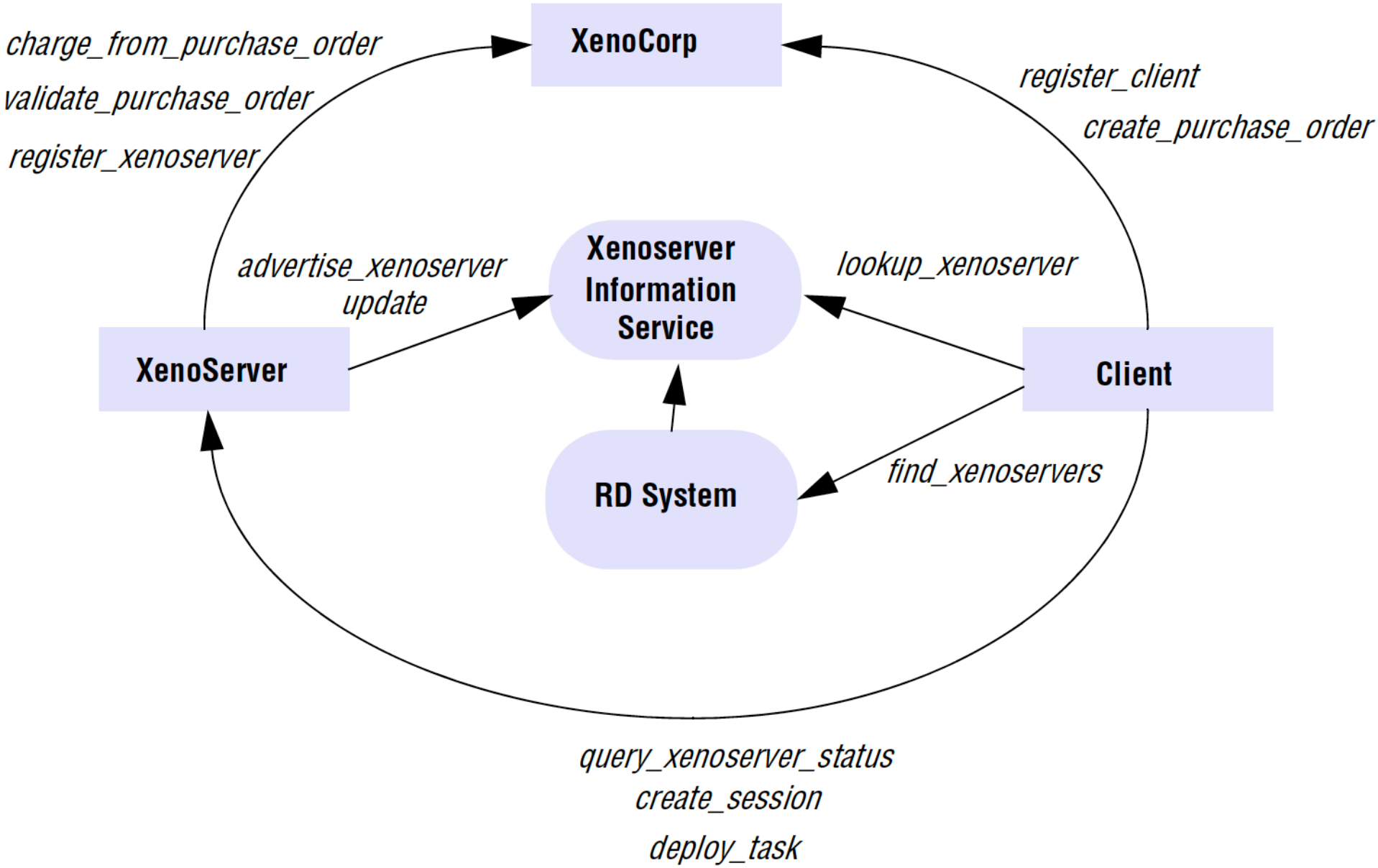
Split device drivers



I/O rings



The XenoServer Open Platform Architecture



Conclusions

We described how an operating system supports middleware

A process consists of an execution environment and threads

- an execution environment consists of an address space, communication interfaces and other local resources such as semaphores;
- a thread is an abstraction that executes within an execution environment. Address spaces need to be large and sparse in order to support sharing and mapped access to objects such as files.
- New address spaces may be created with their regions inherited from parent processes: a technique for copying regions is copy-on-write

Conclusions

- There are two main approaches to kernel architecture: monolithic kernels and microkernels.
- The main difference between them lies in where the line is drawn between resource management by the kernel and resource management performed by dynamically loaded (and usually user-level) servers.
- A microkernel must support at least a notion of process and interprocess communication.
- Virtualization offers an attractive alternative by providing emulation of the hardware and then allowing multiple virtual machines (and hence multiple operating systems) to coexist on the same machine

Exercise

Q: Why are some system interfaces implemented by dedicated system calls (to the kernel), and others on top of message-based system calls?

Solution

Why are some system interfaces implemented by dedicated system calls (to the kernel), and others on top of message-based system calls?

- Dedicated system calls are more efficient for simple calls than message-based calls (in which a system action is initiated by sending a message to the kernel, involving message construction, dispatch etc.).
- However, the advantage of implementing a system call as an RPC is that then a process can perform operations transparently on either remote or local resources.

Exercise

Paolo decides that every thread in his processes ought to have its own *protected* stack – all other regions in a process would be fully shared. Does this make sense?

Solution

Paolo decides that every thread in his processes ought to have its own *protected* stack – all other regions in a process would be fully shared. Does this make sense?

- If every thread has its own *protected* stack, then each must have its own address space.
- The idea is better described as a set of single-threaded processes, most of whose regions are shared: the advantage of sharing an address space has thus been lost.

Exercise

What thread operations are the most significant in cost?

Solution

What thread operations are the most significant in cost?

Thread switching tends to occur many times in the lifetime of threads and is therefore the most significant cost.

Next come thread creation/destruction operations, which occur often in dynamic threading architectures (such as the thread-per-request architecture).

Exercise

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

Solution

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

Most remote invocation costs also feature in message passing. However, if a sender uses asynchronous message passing then it is not delayed by scheduling, data copying at the receiver or waiting for acknowledgements