Distributed Systems

COMP90015 2019 SM1

OS Support

Lectures by Aaron Harwood

Overview

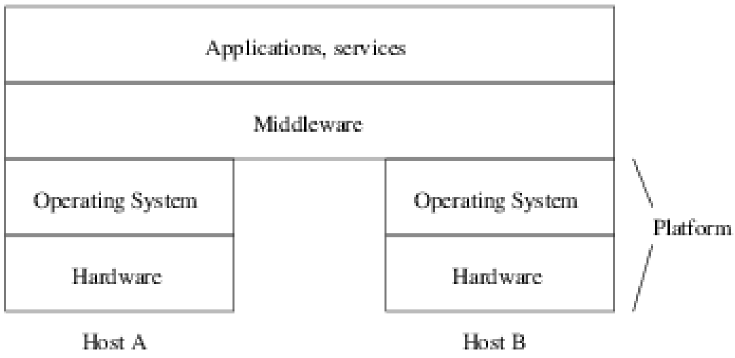
- operating system layer
- protection
- processes and threads
- communication and invocation
- operating system architecture

Networking versus Distributed OS

A *networked operating system* provides support for networking operations. The users are generally expected to make intelligent use of the network commands and operations that are provided. Each host remains autonomous in the sense that it can continue to operate when disconnected from the networking environment.

A *distributed operating system* tries to abstract the network from the user and thereby remove the need for the user to specify how the networking commands and operations should be undertaken. This is sometimes referred to as providing a *single system image*. Each host may not have everything that would be required to operate on its own, when disconnected from the network.

The figure depicts two different hosts, each with its own hardware and operating system, or platform, but conceptually supporting a consistent middleware that supports distributed applications and services.



If the operating system is divided into *kernel* and *server* processes then they:

- *Encapsulate* resources on the host by providing a useful service interface for clients. Encapsulation hides details about the platform's internal operations; like its memory management and device operation.

- *Protect* resources from illegitimate access, from other users and other clients that are using resources on that host. Protection ensures that users cannot interfere with each other and that resources are not exhausted to the point of system failure.

- *Concurrently process* client requests, so that all clients receive service. Concurrency can be achieved by sharing time -- a fundamental resource -- called time sharing.

E.g. a client may allocate memory using a kernel system call, or it may discover an network address by using a server object. The means of accessing the encapsulated object is called an *invocation method*.

The core OS components are:

- *Process manager* -- Handles the creation of processes, which is a unit of resource management, encapsulating the basic resources of memory (address space) and processor time (threads).
- *Thread manager* -- Handles the creation, synchronization and scheduling of one or more threads for each process. Threads can be scheduled to receive processor time.
- *Communication manager* -- Handles interprocess communication, i.e. between threads from different processes. In some cases this can be across different hosts.
- *Memory manager* -- Handles the allocation and access to physical and virtual memory. Provides translation from virtual to physical memory and handles paging of memory.
- *Supervisor* -- Handles privileged operations, i.e. those that directly affect shared resources on the host, e.g. to and from an I/O device. The supervisor is responsible for ensuring that host continues to provide proper service to each client.

Protection

Resources that encapsulate space, such as memory and files, typically are concerned with *read* and *write* operations. Protecting a resource requires ensuring that only legitimate read and write operations take place.

Legitimate operations are those carried out only by clients who have the right to perform them. A legitimate operation should also conform to resource policies of the host, e.g. a file should never exceed 1 GB in size or at most 100MB of memory can be allocated.

In some cases the resource may also be protected by giving it the property of *visible* versus *invisible*. A visible resource can be discovered by listing a directory contents or searching for it. An invisible resource should be known a priori to the client; it can be guessed though.

Resources that encapsulate time, i.e. processes, are concerned with *execute* operations. In this case a client may or may not have the right to create a process. Again, host based policies should be enforced.

The kernel is that part of the operating system which assumes full access to the host's resources. The kernel begins execution soon after the host is powered up and continues to execute while the host is operational. The kernel has access to all resources and shares access to all other processes that executing on the host. To do this securely requires hardware support at the machine instruction level, which is supplied by the processor using two fundamental operating modes:

- *supervisor* mode -- instructions that execute while the processor is in supervisor (or privileged) mode are capable of accessing and controlling every resource on the host,
- *user* mode -- instructions that execute while the processor is in user (or unprivileged) mode are restricted, by the processor, to only those accesses defined or granted by the kernel.

Most processors have a register that determines whether the processor is operating in user or supervisor mode.

Before the kernel assigns processor time to a user process, it puts the processor into user mode.

A user process accesses a kernel resource using a *system call*. The system call is an *exception* that puts the processor into supervisor mode and returns control to the kernel.

Other kinds of exceptions can occur, such as when an illegal instruction has been issued, a resource is trying to be accessed that is not allowed, or when I/O is pending.

Processes and threads

A process encapsulates the basic resources of memory and processor time. It also encapsulates other higher level resources.

Each process:

- has an address space and has some amount of allocated memory,
- consists of one or more threads that are given processor time, including thread synchronization and communication resources,
- higher-level resources like open files and windows.

Threads have equal access to the resources encapsulated within the process.

Resource sharing or interprocess communication is required for threads to access resources in other processes. E.g. shared memory or socket communication.

Address spaces

Most operating systems allocate a *virtual* address space for each process. The virtual address space is typically byte addressable and on a 32 bit architecture will typically have 2^32 byte addresses.

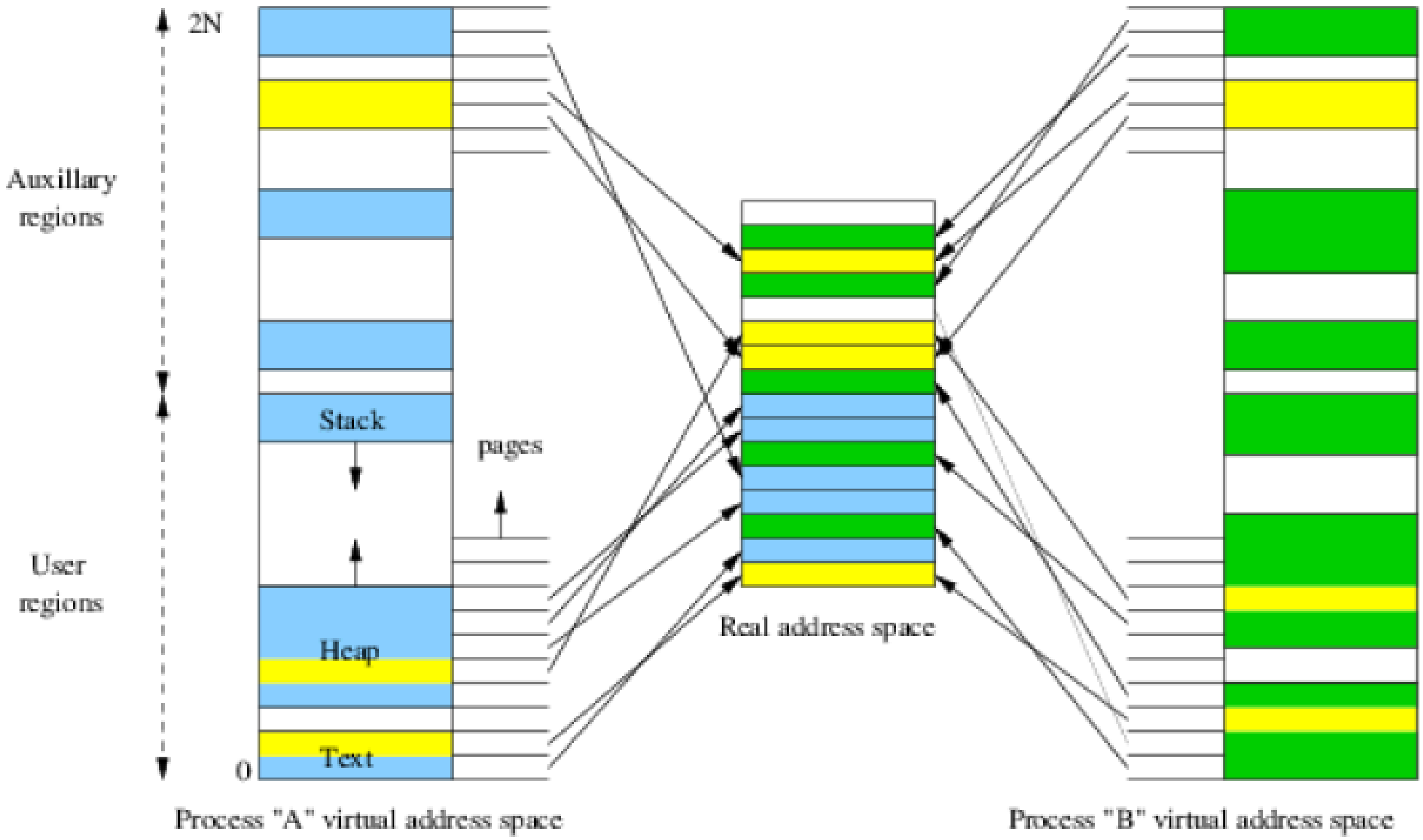
The virtual address space can be divided into *regions* that are contiguous and do not overlap.

A *paged* virtual memory scheme divides the address space into fixed sized blocks that are either located in physical memory (RAM) or located in swap space on the hard disk drive.

A *page table* is used by the processor and operating system to map virtual addresses to real addresses. The page table also contains access control bits for each page that determine, among other things, the access privileges of the process on a per page basis.

The operating system manages the pages, swapping them into and out of memory, in response to process memory address accesses.

Example virtual address space mappings


2N
Auxillary regions
User regions
0
Stack
pages
Heap
Text
Real address space
Process "A" virtual address space
Process "B" virtual address space

Shared memory

Two separate addresses spaces can share parts of real memory. This can be useful in a number of ways:

- *Libraries*: The binary code for a library can often be quite large and is the same for all processes that use it. A separate copy of the code in real memory for each process would waste real memory space. Since the code is the same and does not change, it is better to share the code.

- *Kernel*: The kernel maintains code and data that is often identical across all processes. It is also often located in the same virtual memory space. Again, sharing this code and data can be more efficient than having several copies.

- *Data sharing and communication*: When two processes want access to the same data or want to communicate then shared memory is a possible solution. The processes can arrange, by calling appropriate system functions, to share a region of memory for this purpose. The kernel and a process can also share data or communicate using this approach.

Examples of each of these shared memory uses are shown in the previous figure.

Creation of a new process

The operating system usually provides a way to create processes. In UNIX the `fork` system call is used to duplicate the caller's address space, creating a new address space for a new process. The new process is identical to the caller, apart from the return value of the fork system call is different in the caller. The caller is called the parent and the new process is called the child.

In UNIX a `exec` system call can be used to replace the caller's address space with a new address space for a new process that is named in the system call.

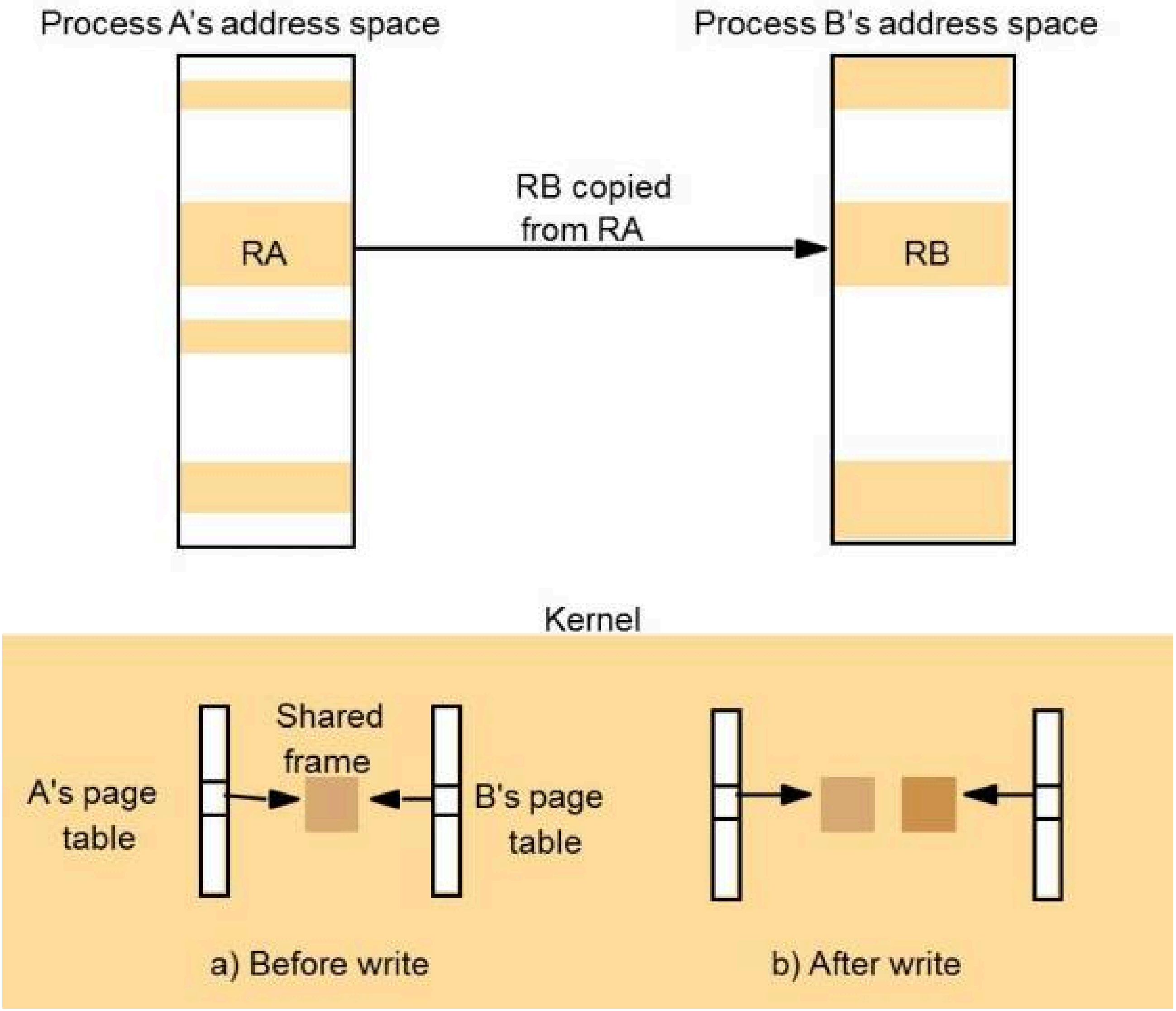
A combination of fork and exec allows new processes to be allocated.

Copy on write

When a new process is created using fork, the address space is copied. The new process' code is identical and is usually read-only so that it can be shared in real memory and no actual copying of memory bytes is required. This is faster and more efficient than making a copy.

However the data and other memory regions may or may not be read-only. If they are writable then the new process will need its own copy when it writes to them.

Copy on write is a technique that makes a copy of a memory region only when the new process actually writes to it. This saves time when allocating the new process and saves memory space since only what is required to be copied is actually copied.


Process A's address space
Process B's address space
RB copied
from RA
RA
RB
Kernel
A's page
table
Shared
frame
B's page
table
a) Before write
b) After write

New processes in a distributed system

In a distributed system there is a choice as to which host the new process will be created on. In a distributed operating system this choice would be made by the operating system.

The decision is largely a matter of policy and some categories are:

- *transfer policy* -- determines whether the new process is allocated locally or remotely.
- *location policy* -- determines which host, from a set of given hosts, the new process should be allocated on.

The policy is often transparent to the user and will attempt to take into account such things as relative load across hosts, interprocess communication, host architectures and specialized resources that processes may require.

When the user is programming for explicit parallelism or fault tolerance then they may require a means for specifying process location. However, it is desirable to make these choices automatic as well.

Process location policies may be *static* or *adaptive*. Static policies do not take into account the current state of the distributed system. Adaptive policies receive feedback about the current state of the distributed system.

A *load manager* gathers information about the current state of the distributed system. Load managers may be:

- *centralized* -- a single load manager receives feedback from all other hosts in the system.

- *hierarchical* -- load managers are arranged in a tree where the internal nodes are load managers and the leaf nodes are hosts.

- *decentralized* -- there is typically a load manager for every host and load managers communicate with all other load managers directly.

In a *sender-initiated* or push policy, the local host is responsible for determining the remote host to allocate the process. In the *receiver-initiated* or pull policy, remote hosts advertises to other hosts that new processes should be allocated on it.

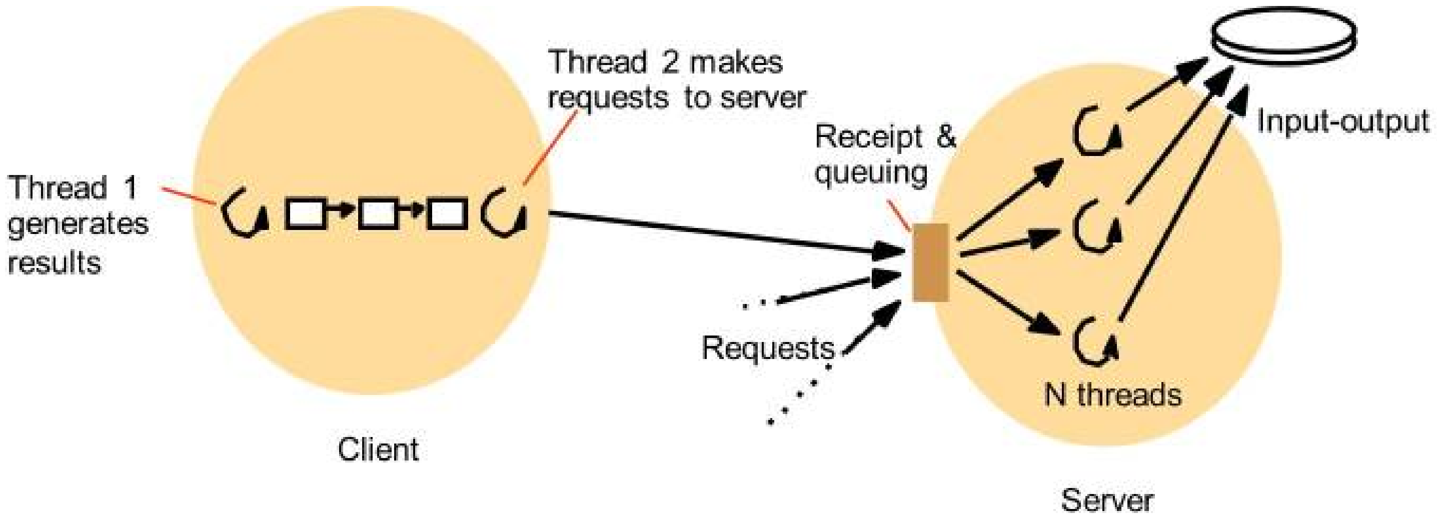
Process migration

Processes can be *migrated* from one host to another by copying their address space. Depending on the platform and on the resources that are in current use by the process, process migration can be more or less difficult.

Process code is often CPU dependent, e.g. x86 versus SPARC. This can effectively prohibit migration. Ensuring that hosts are homogeneous can eliminate this problem. Using virtual machines can also help, by avoiding CPU dependent instructions.

Even if the host platforms are such that a process can migrate, the process may be using host resources such as open files and sockets that further complicates the migration.

Threads



Typically there is significant use of threads in a distributed system.

Performance bottleneck

Consider a server than handles client requests. Handling the request requires a disk access of 8ms and 2ms of processor time. A process with a single thread takes 10ms of time and the server can complete 100 requests/second.

If we use two threads, where each thread independently handles a request then while the first thread is waiting for the disk access to complete the second thread is executing on the processor. In general if we have many threads then the server becomes bottlenecked at the disk drive and a request can be completed every 8ms which is 125 requests/second.

Consider when disk accesses are cached with a 75% hit rate and an increase in processor time to 2.5ms (the increase is due to the cache access). The disk access is now completed in 0.75x0 + 0.25x8 = 2ms. Hence the requests are bottlenecked at the processor and the maximum rate is 1000/2.5=400 requests/second.

Worker pool architecture

Creating a new thread incurs some overhead that can quickly become the bottleneck in a server system. In this case it is preferable to create threads in advance.

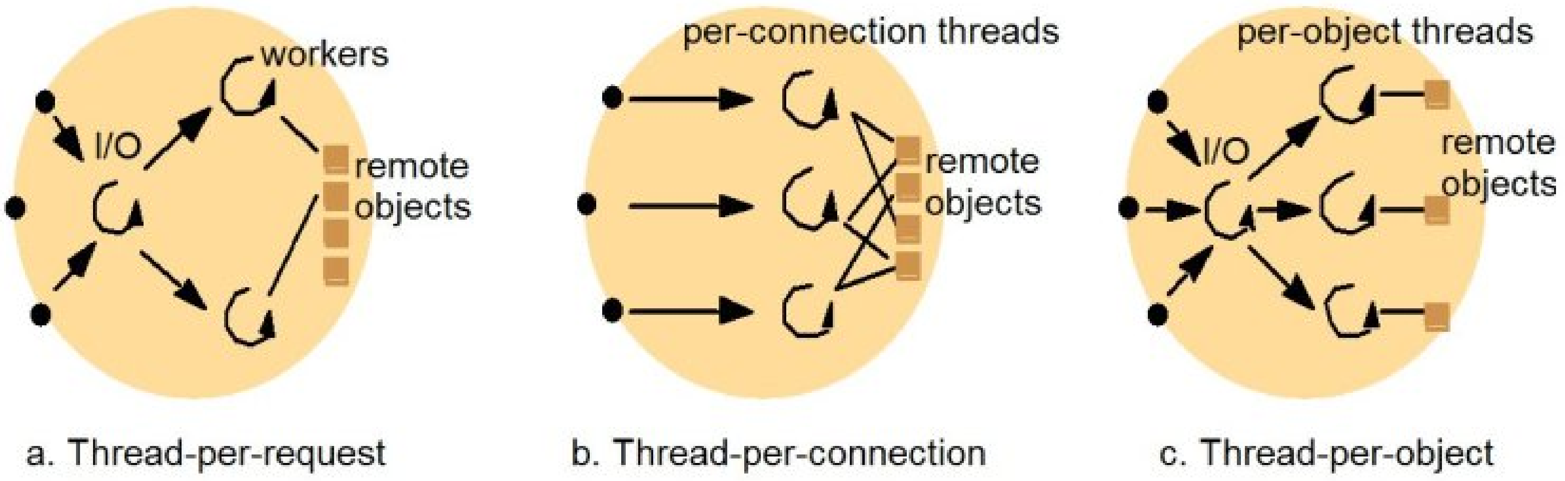
It is okay to have a single I/O thread to receive the requests since the I/O is the bottleneck.

In the *worker pool* architecture the server creates a fixed number of threads called a worker pool. As requests arrive at the server, they are put into a queue by the I/O thread and from there assigned to the next available worker thread.

Request priority can be handled by using a queue for each kind of priority. Worker threads can be assigned to high priority requests before low priority requests.

If the number of workers in a pool is too few too handle the rate of requests then a bottleneck forms at the queue. Also, the queue is shared and this leads to an overhead.

Alternative Threading



Thread-per-request architecture

In the *thread-per-request* architecture a separate thread is created by the I/O thread for each request and the thread is deallocated when the request is finished.

This allows as many threads as requests to exist in the system and avoids accessing a shared queue. Potential parallelism can be maximized.

However, thread allocation and deallocation incurs overheads as mentioned earlier. Also, as the number of threads increases then the advantages from parallelism decreases (for a fixed number of processors) and the overhead incurred in context switching between threads can outweigh the benefit. Managing large numbers of threads can be more expensive than managing a large queue.

Thread-per-connection architecture

A single client may make several requests. In the previous architectures, each request is treated independently of the client connection.

In the *thread-per-connection* architecture, a separate thread is allocated for each *connection* rather than for each *request*. This can reduce the overall number of threads as compared to the thread-per-request architecture.

The client can make several requests over the single connection.

Thread-per-object architecture

In the *thread-per-object* architecture, a worker thread is associated for each remote object or resource that is being accessed. An I/O thread receives requests and queues them for each worker thread.

Threads within clients

Threads are useful within clients in the case when the request to the server takes considerable time. Also, communication invocations often block while the communication is taking place.

E.g., in a web browser the user can continue to interact with the current page while the next page is being fetched from the server. Multiple threads can be used to fetch each of the images in the page, where each image may come from a different server.

Threads versus multiple processes

It is possible to use multiple processes instead of multiple threads. However the threaded approach has the advantages of being cheaper to allocate/deallocate and of being easy to share resources via shared memory.

One study showed that creating a new process took about 11ms while creating a new thread took 1ms. While these times may be decreasing as as technology improves, the relative difference is likely to remain.

This is because, e.g., processes require a new address space which leads to a new page table, while threads require in comparison only a new processor context.

Context switching between threads is also cheaper than between processes because of cache behavior concerned with address spaces. Some processors are optimized to switch between threads (hyper-threading) and so this can lead to greater advantages for the threading model; though it has caveats as well.

A *processor context* comprises the values of the processor registers such as the program counter, the address space identifier and the processor protection mode (supervisor or user).

Context switching involves saving the processor's original state and loading the new state. When changing from a user context to a kernel context it also involves changing the protection mode which is a called a *domain transition*.

One study showed that context switching between processes took 1.8ms while switching between threads belonging to the same process took 0.4ms. Again, it is the relative difference that is of interest here.

Thread programming

Thread programming is largely concerned with the study of concurrency, including:

- race condition,
- critical section,
- monitor,
- condition variable, and
- semaphore.

In Java, the `Thread` class is used to implement threads.

Java thread 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.
- `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()` -- Enter the `READY` state and invoke the scheduler.
- `destroy()` -- Destroy the thread.

Thread lifetime

A new thread is created on the Java virtual machine (JVM) as its creator.

It is created in the `SUSPENDED` state.

It is made `RUNNABLE` using the `start()` method and executes the `run()` method (either designated or its own).

Threads can be assigned priorities for scheduling purposes and can be managed in groups.

Groups provide protection of threads within a group from threads in other groups.

Java thread synchronization

- `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`.

Consider a shared function to generate unique identifiers:

```
int greatest=0;
Stack idStack=new Stack();
public int generateUniqueID(){
   if(!idStack.empty()){
      Integer top =
         (Integer) idStack.pop();
      while(top.intValue()>greatest)
         top = (Integer) idStack.pop();
      return top.intValue();
   }
   greatest++;
   return greatest-1;
}
```

This code doesn't work when several threads call it concurrently.

Consider when there is exactly one element on the stack. The condition to check whether the stack is empty may be tested concurrently by two or more threads. All threads will assume that the stack contains a value to be popped and will attempt to pop it. In fact there are a lot of problems with executing this code concurrently.

One way to avoid these problems is to allow only one thread to be executing the function at a time. This eliminates concurrency for this function which reduces the parallelism but avoids the concurrent access problem.

Thread scheduling

Some thread scheduling is *preemptive*. In this case a running thread is suspended at any time, usually periodically, to allow processor time for other threads.

Other the thread scheduling is *non-preemptive*. In this case a running thread will continue to receive processor time until the thread *yields* control back to the thread scheduler.

Non-preemptive scheduling has the advantage that concurrency issues, such as mutual exclusion to a shared variable, are greatly simplified.

However, non-preemptive scheduling does not guarantee that other threads will receive processor time since an errant thread can run for an arbitrarily long time before yielding. This can have negative effects on performance and usability.

Real time scheduling of threads is not support by vanilla Java. Real time imposes another set of constraints on thread scheduling.

Thread implementation

Depending on the kernel, the threads of a process may or may not be schedulable across different processors.

Some kernels provide system calls only to allocate processes. Some kernels provide system calls to allocate threads as well; they can be call *kernel threads*. This can depend on whether the kernel itself is threaded. Most modern OSes have threaded kernels.

If the kernel does not provide thread allocation (and even if it does), it is possible for the user to manage *user threads* within a process.

In the case of user threads, the kernel gives processor time to the process and the process is responsible for scheduling the threads.

User versus kernel threads

User threads within a process cannot take advantage of multiple processors.

A user thread that causes a page fault will block the entire process and hence all of the threads in that process.

User threads within different processes cannot be scheduled according to a single scheme of relative prioritization.

Context switching between user threads can be faster than between kernel threads.

A user thread scheduler can be customized by the user, to be specific for a given application.

Usually the kernel places limits on how many kernel threads can be allocated, a user thread scheduler can usually allocate more threads than this.

It is possible to combine kernel level threads with user level threads.

Communication and invocation

An invocation such as a remote method invocation, remote procedure call or event notification is intended to bring about an operation on a resource in a different address space.

- What communication primitives are available?
- Which protocols are supported and how open are they?
- What is done to make the communication efficient?
- What additional support is provided, e.g. for high-latency communication and issues such as disconnection?

The kernel can provide communication primitives such as TCP and UDP, and it can also provide higher level primitives. In most cases the higher level primitives are provided by middleware because it become too complex to develop them into the kernel and because standards are not widely accepted at the higher level.

Protocols and openness

Using open protocols, as opposed to closed or proprietary protocols, facilitates interoperation between middleware implementations on different systems.

Experience has shown that kernels which implement and require their own network protocols do not become popular.

A design choice of some new kernels is to leave the implementation of networking protocols to servers. Hence a choice of networking protocol can be more easily made.

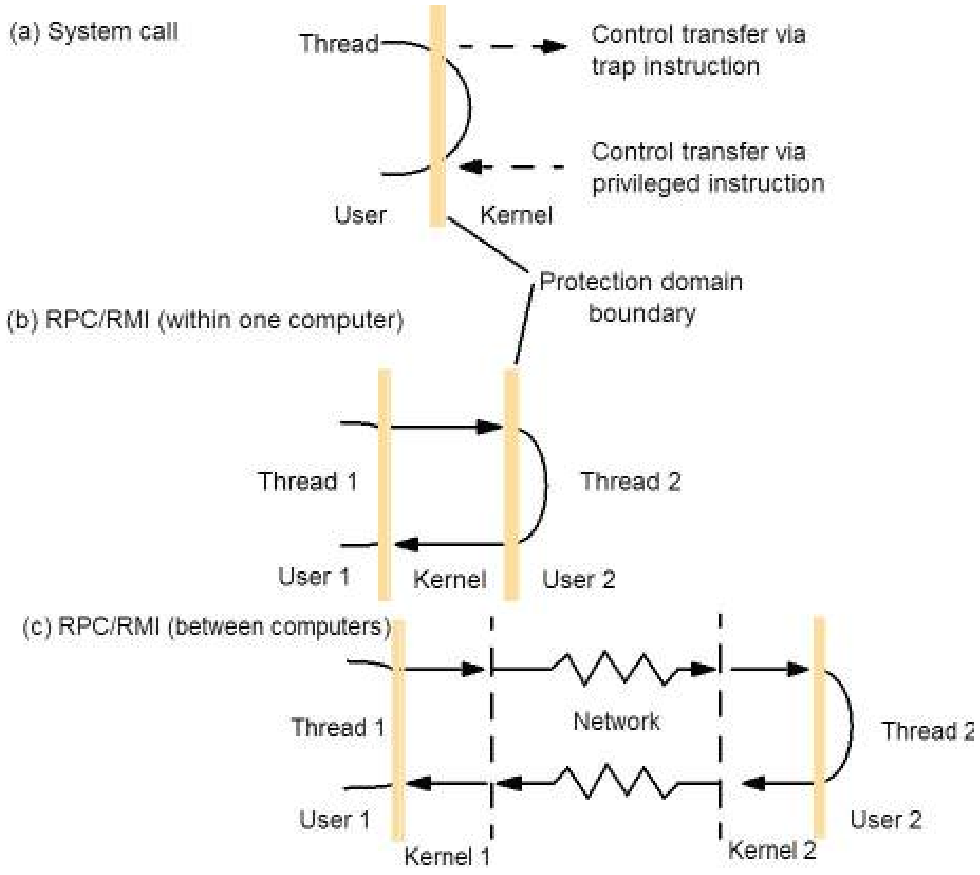
The kernel is still required to provide device drivers for new networking devices such as infrared and BlueTooth. Either the kernel can transparently select the appropriate device driver or middleware should be able to dynamically select the appropriate driver. In either case, standard protocols such as TCP and UDP allow the middleware and application to make use of the new devices without significantly changes.

Invocation performance

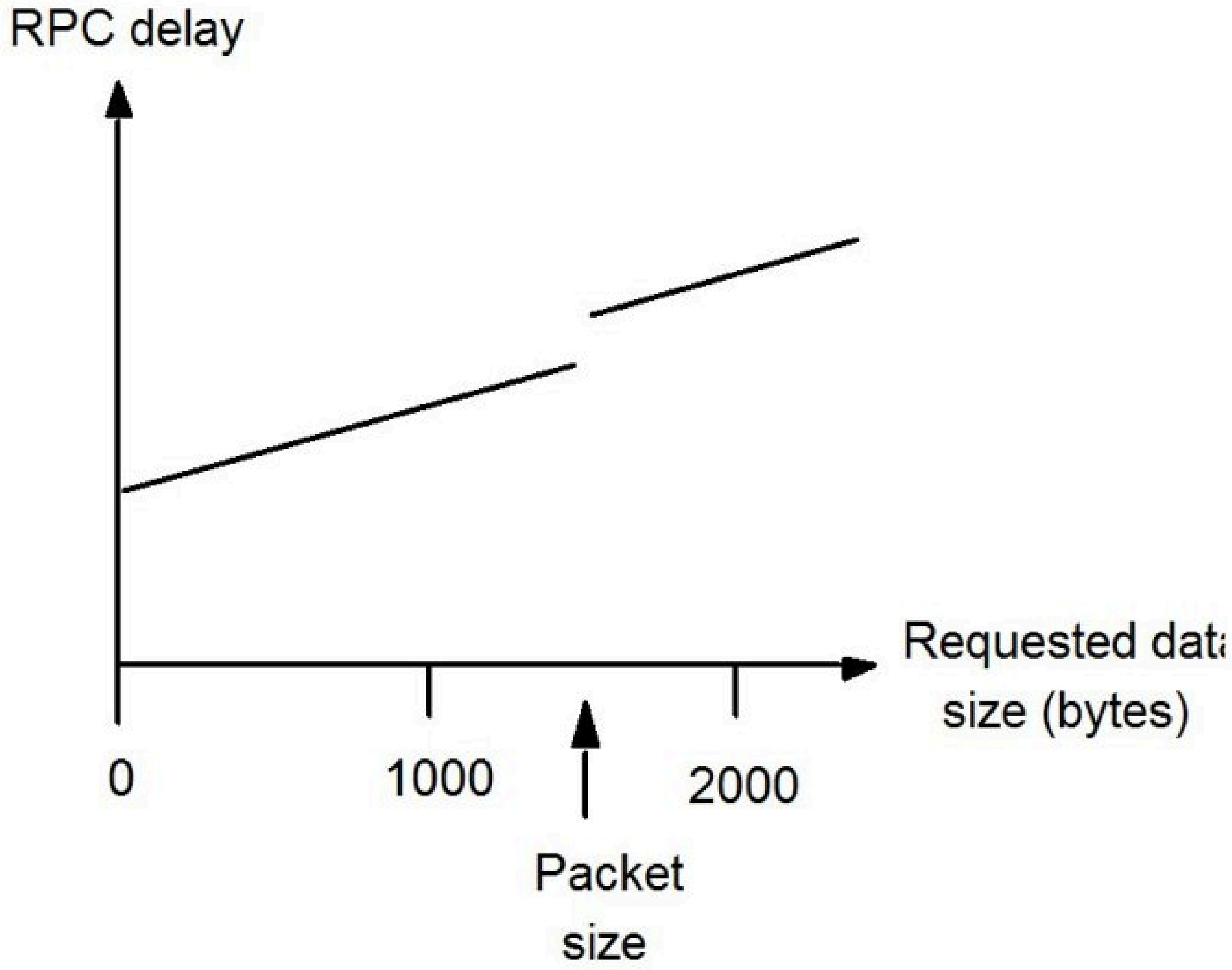
The performance of an invocation can be categorized into three kinds, depending on what is required to invoke the resource:

- User space procedure -- minimal overhead, allocating stack space.
- System call -- if a system call is required then the overhead is characterized by a domain transition two and from the kernel.
- Interprocess on the same host -- in this case there is a domain transition to the kernel, to the other process, back to the kernel and back to the calling process.
- Interprocess on a remote host -- this case includes the same overheads as the previous case, plus the overhead of network communication between the two kernels.

One study showed that the time for a null RPC between user processes on two 500MHz PCs across a 100Mbps LAN is in the order of tenths of a millisecond. By comparison an equivalent user space procedure call takes a small fraction of a microsecond.


(a) System call
Thread
Control transfer via trap instruction
Control transfer via privileged instruction
User
Kernel
Protection domain boundary
(b) RPC/RMI (within one computer)
Thread 1
Thread 2
User 1
Kernel
User 2
(c) RPC/RMI (between computers)
Thread 1
Network
Thread 2
User 1
User 2
Kernel 1
Kernel 2

RPC Delay versus size


RPC delay
Requested dat
size (bytes)
0
1000
2000
Packet
size

The following are the main factors contributing to delay for RMI, apart from actual network delay:

- marshalling -- copying and converting data becomes significant as the amount of data grows.
- data copying -- after marshalling, the data is typically copied several times, from user to kernel space, and to different layers of the communication subsystem.
- packet initialization -- protocols headers and checksums take time, the cost is proportional to the size of the data.
- thread scheduling and context switching -- system calls and server threads.
- waiting for acknowledgments -- above the network level acknowledgments.

Communication via shared memory

Shared memory can be used to communicate between user processes and between a user process and the kernel.

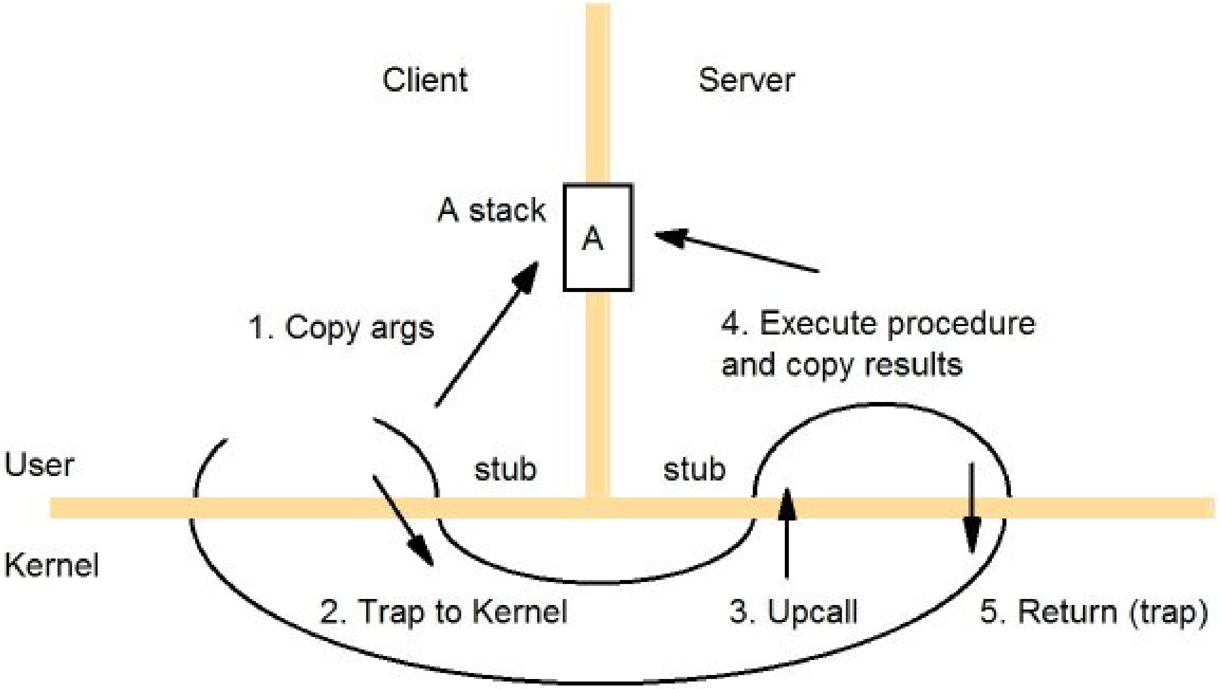
Data is communicated by writing to and reading from the shared memory, like a "whiteboard".

In this case, when communicating between user processes the data is not copied to and from kernel space.

However the processes will typically need to use some synchronization mechanism to ensure that communication is deterministic.

In UNIX, a shared memory area can be obtained using `shmget()`, the area can be attached to a process using `shmat()`, detached using `shmdt()` and controlled (e.g. specifying read/write permissions) using `shmctl()`. The `shmget()` is a system call that returns an identifier and the identifier must be communicated to other processes. Thus there is some overhead in setting up a shared memory area.

Lightweight RPC



Choice of protocol

Connection-oriented protocols like TCP are often used when the client and server are to exchange information in a single session for a reasonable length of time; e.g. telnet or ssh. These protocols can suffer if the end-points are changing their identity, e.g. if one of the end-points is mobile and roaming from one IP address to another, or e.g. if a DHCP based wireless base station experiences a lot of contention and the clients to the base station are continually having their IP address change.

Connection-less protocols like UDP are often used for request-reply applications, that do not require a session for any length of time. E.g., finding the time from a time server. Because these protocols often have less overhead, they are also used for applications that require low latency, e.g. in streaming media and online games.

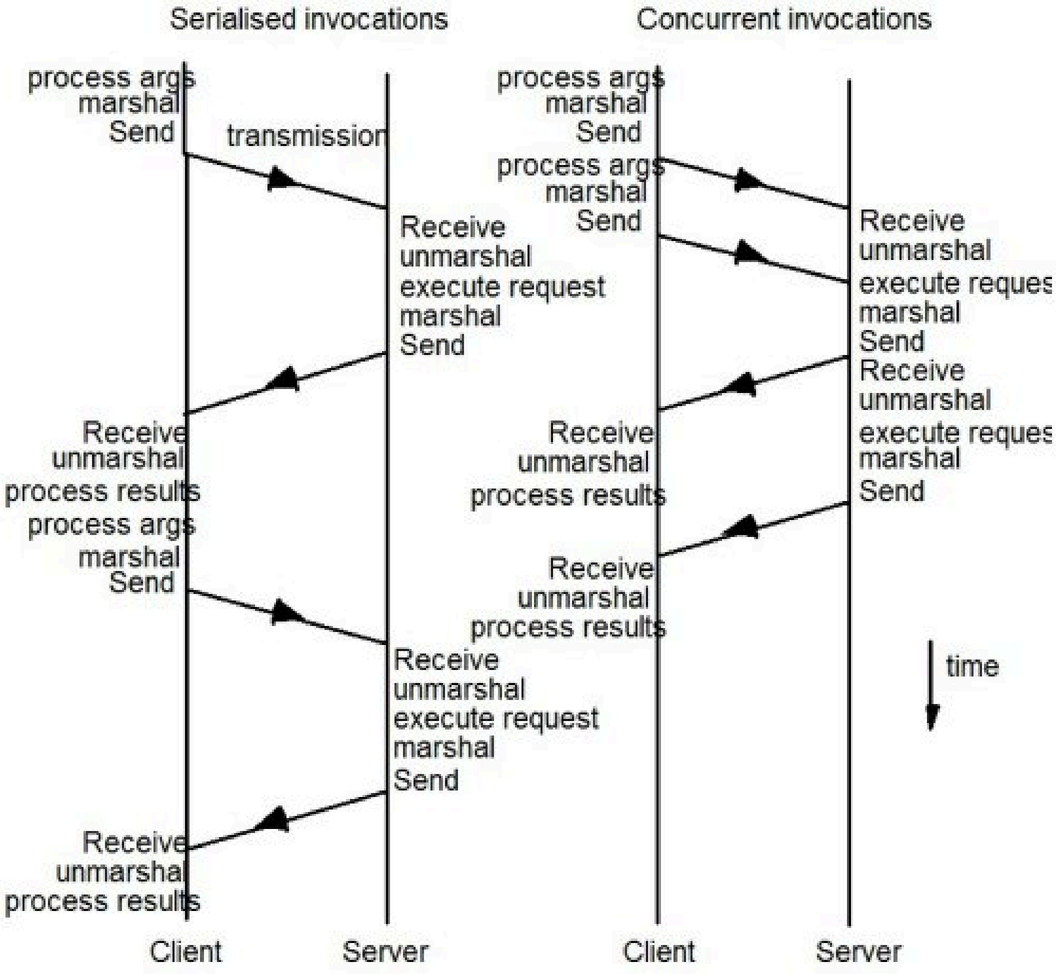
In many cases, cross address space communication happens between processes on the same host and so it is preferable to (transparently) choose a communication mechanism (such as using shared memory) that is more efficient. Lightweight RPC is an example of this.

Concurrent and asynchronous operation

In many applications it is possible to have a number of outstanding or *concurrent* requests. In other words it is not necessary to wait for the response to a request before moving on to another request. This was exemplified in the use of threads for the client, where e.g. several requests to different web servers could be undertaken concurrently.

The telnet client/server is an example of *asynchronous* operation. In this case, whenever a key is typed at the keyboard the client sends the key to the server. Whenever output is available from the server it is sent to the client and received data from the server is printed on the terminal by the client whenever it arrives. Sends and receives are not synchronized in any particular way.

An asynchronous invocation returns without waiting for the invocation request to be completed. Either the caller must periodically check whether the request has completed or the caller is notified when the request has completed. The caller is required to take appropriate action if the request fails to complete.


Serialised invocations
Concurrent invocations
process args
marshal
Send
transmission
Receive
unmarshal
execute request
marshal
Send
Receive
unmarshal
process results
process args
marshal
Send
Receive
unmarshal
execute request
marshal
Send
Receive
unmarshal
process results
Client
Server
process args
marshal
Send
process args
marshal
Send
Receive
unmarshal
execute reques
marshal
Send
Receive
unmarshal
execute reques
marshal
Send
Receive
unmarshal
process results
Receive
unmarshal
process results
time
Client
Server

Persistent asynchronous invocations

With usual networking circumstances it is possible to automatically retry a nonresponsive request after a timeout, retrying some number of times before the request is considered to have failed. Usually the timeout period is some number of seconds and is sufficient for example when making a request from a client on the Internet to a server on the Internet.

However, with the use of mobile devices that can experience excessively long disconnection times it is preferable to handle asynchronous invocations in a different way. A *persistent asynchronous invocation* is placed in a queue at the client and attempts are made to complete the request as the client roams from network to network. At the server side, the response to the request is put into a "client mailbox" and the client is required to retrieve the response when it can.

Persistent invocations allow the user to select which kind of network (e.g. GSM or Internet) will be used to service the request. Queued RPC is an example of this.

Operating system architecture

An open distributed system should make it possible to:

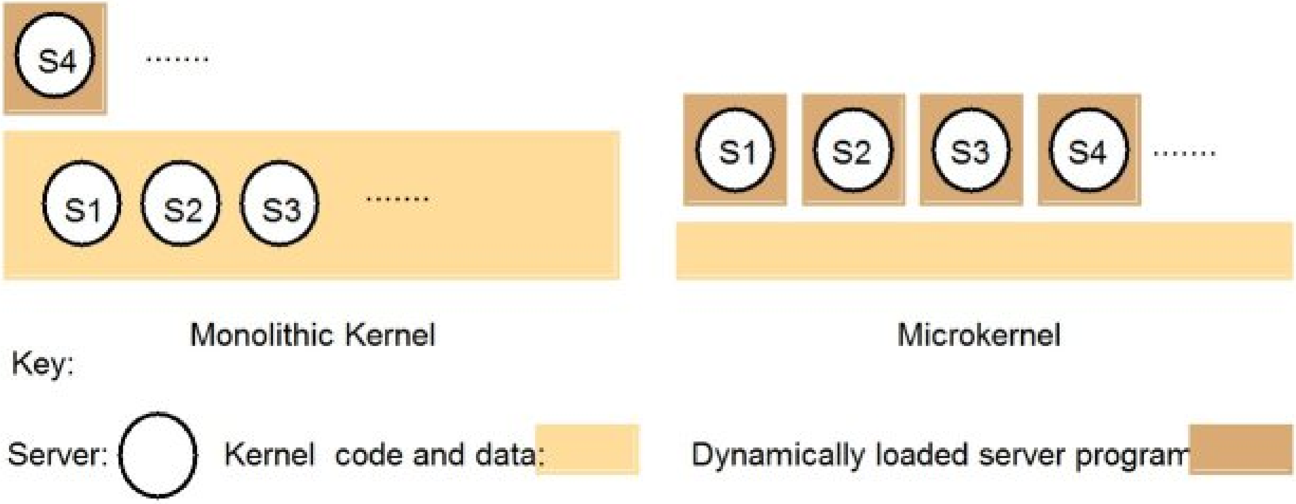
- Run only that system software that is specifically required by the hosts hardware, e.g. specific software for a mobile phone or personal digital assistant. This leaves as much resources as possible available for user applications.
- Allow the software (and the host) implementing any particular service to be changed independently of other facilities.
- Allow for alternatives of the same service to be provided, when this is required to suit different users or applications.
- Introduce new services without harming the integrity of existing ones.

Monolithic and Micro kernel design

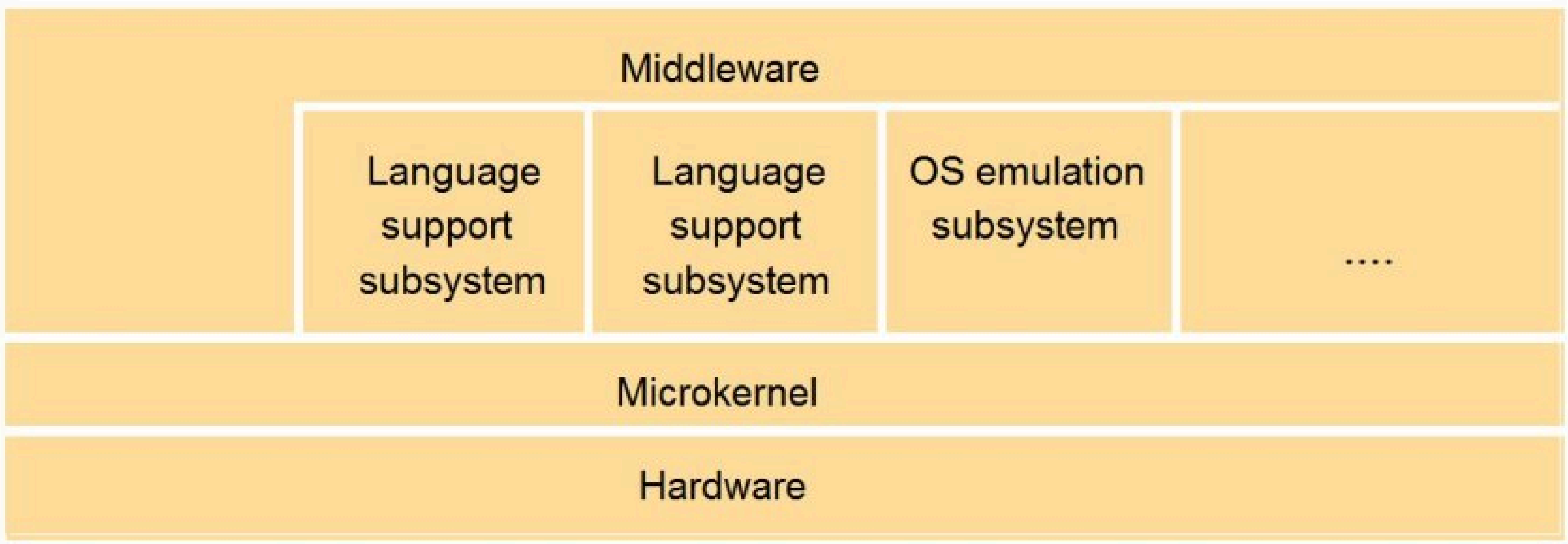
The UNIX operating system is described as having a *monolithic kernel*. In this case there is significantly detailed and diverse functionality supplied by the kernel. Changing kernel functionality is more difficult than changing a server functionality. Same implementations use *kernel modules* to load and unload functionality into the kernel.

In contrast, the Mach operating system is described as a *micro kernel*. In this case the kernel provides only the fundamental functionality such as address space management, threads and local interprocess communication. All other functionality is provided by servers.

Monolithic kernels can sometimes be more efficient because complex functionality can be provided through a smaller number of system calls and interprocess communication is minimized. However micro kernels allow a greater extensibility and have a better ability to enforce modularity behind memory protection boundaries. A small kernel is also more likely to be free of bugs.


S4
....
S1
S2
S3
....
Monolithic Kernel
S1
S2
S3
S4
....
Microkernel
Key:
Server:
Kernel code and data:
Dynamically loaded server program

Role of Microkernel



The microkernel supports middleware via subsystems

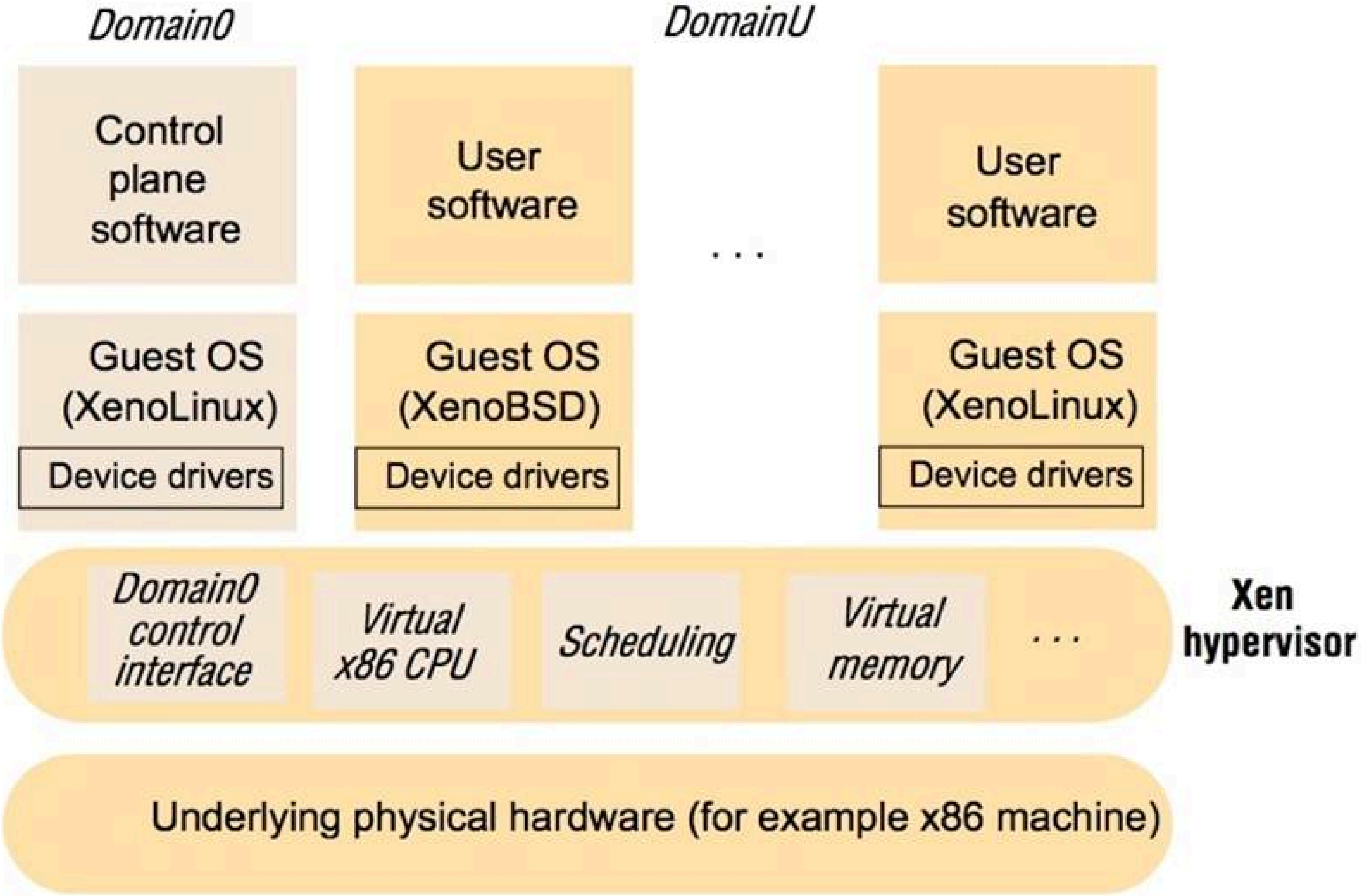
Emulation and virtualization

The adoption of microkernels is hindered in part because they do not run software that a vast majority of computer users want to use. Microkernels can use binary *emulation* techniques, e.g. emulating another operating system like UNIX, to overcome this. The Mach OS emulates both UNIX and OS/2.

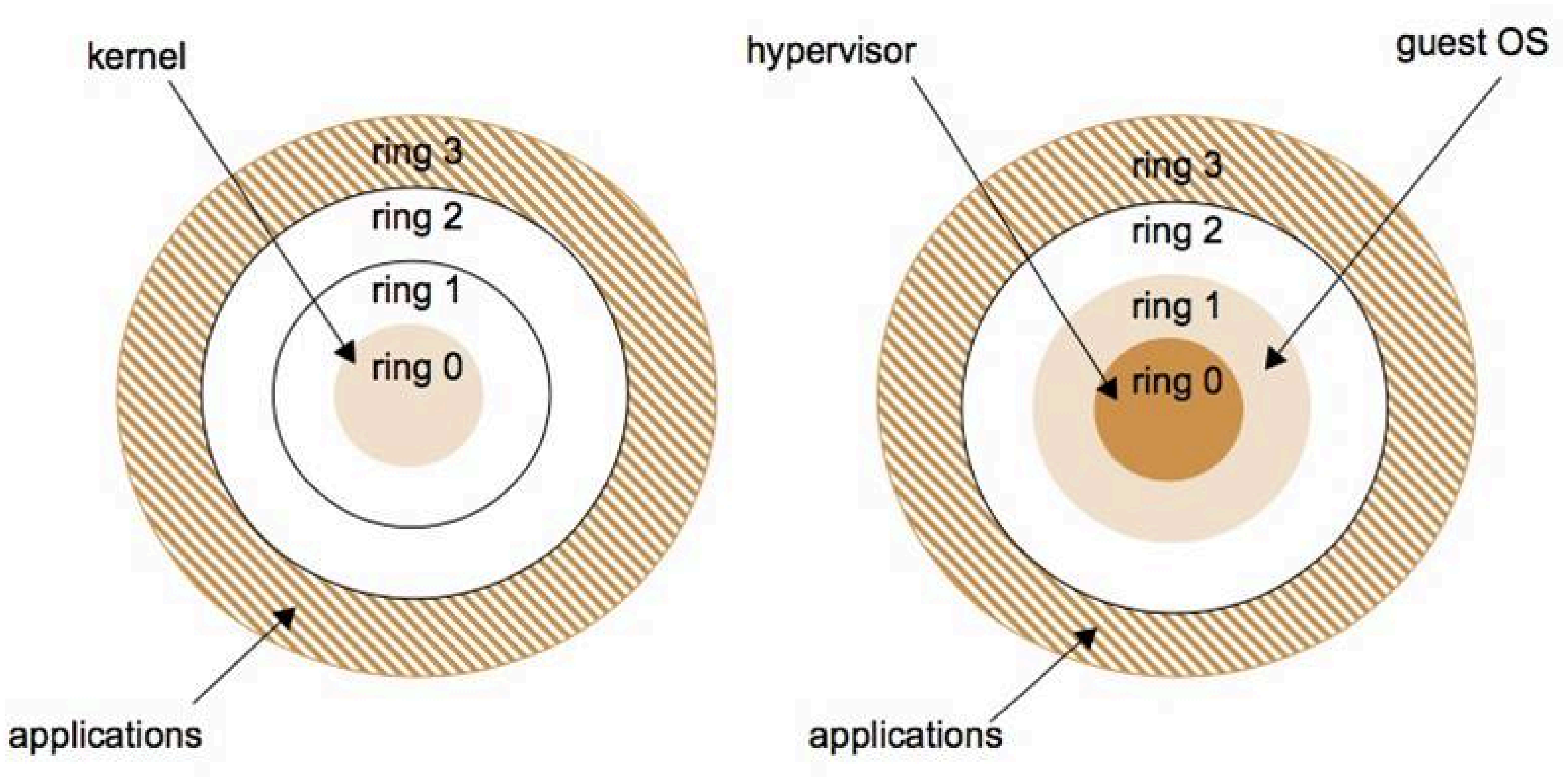
Virtualization can also be used and can be achieved in different ways. Virtualization can be used to run multiple instances of virtual machines on a single real machine. The virtual machines are then capable of running different kernels. Another approach is to implement an OS' application programming interface, which is what the UNIX Wine program does for Windows; i.e. it implements the Win32 API on UNIX.

Today, two prominent research systems that use virtualization are *Xen* and *PlanetLab*.

Xen Architecture


Domain0
DomainU
Control plane software
User software
. . .
User software
Guest OS (XenoLinux)
Device drivers
Guest OS (XenoBSD)
Device drivers
Guest OS (XenoLinux)
Device drivers
Domain0 control interface
Virtual x86 CPU
Scheduling
Virtual memory
. . .
Xen hypervisor
Underlying physical hardware (for example x86 machine)

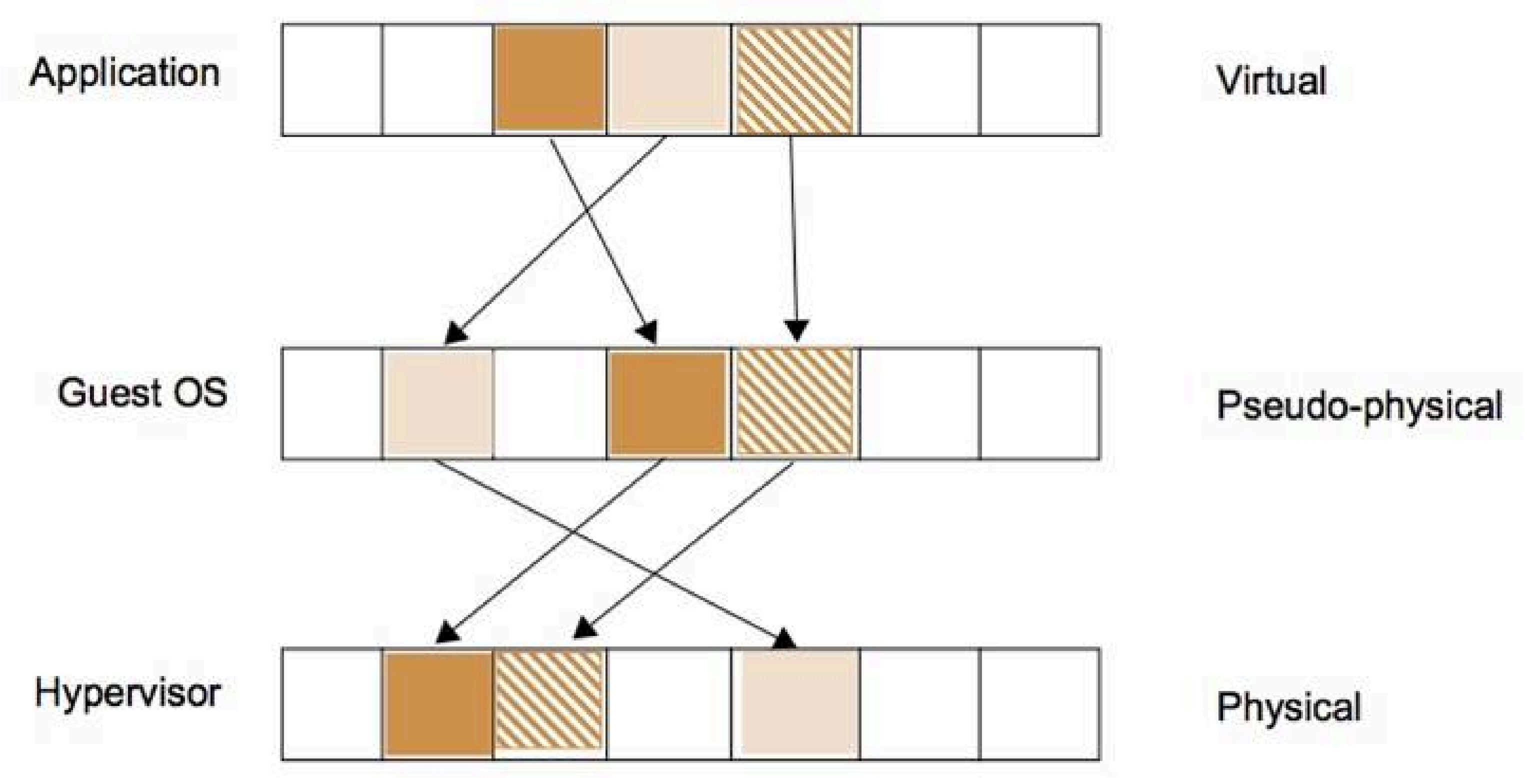
Privileges



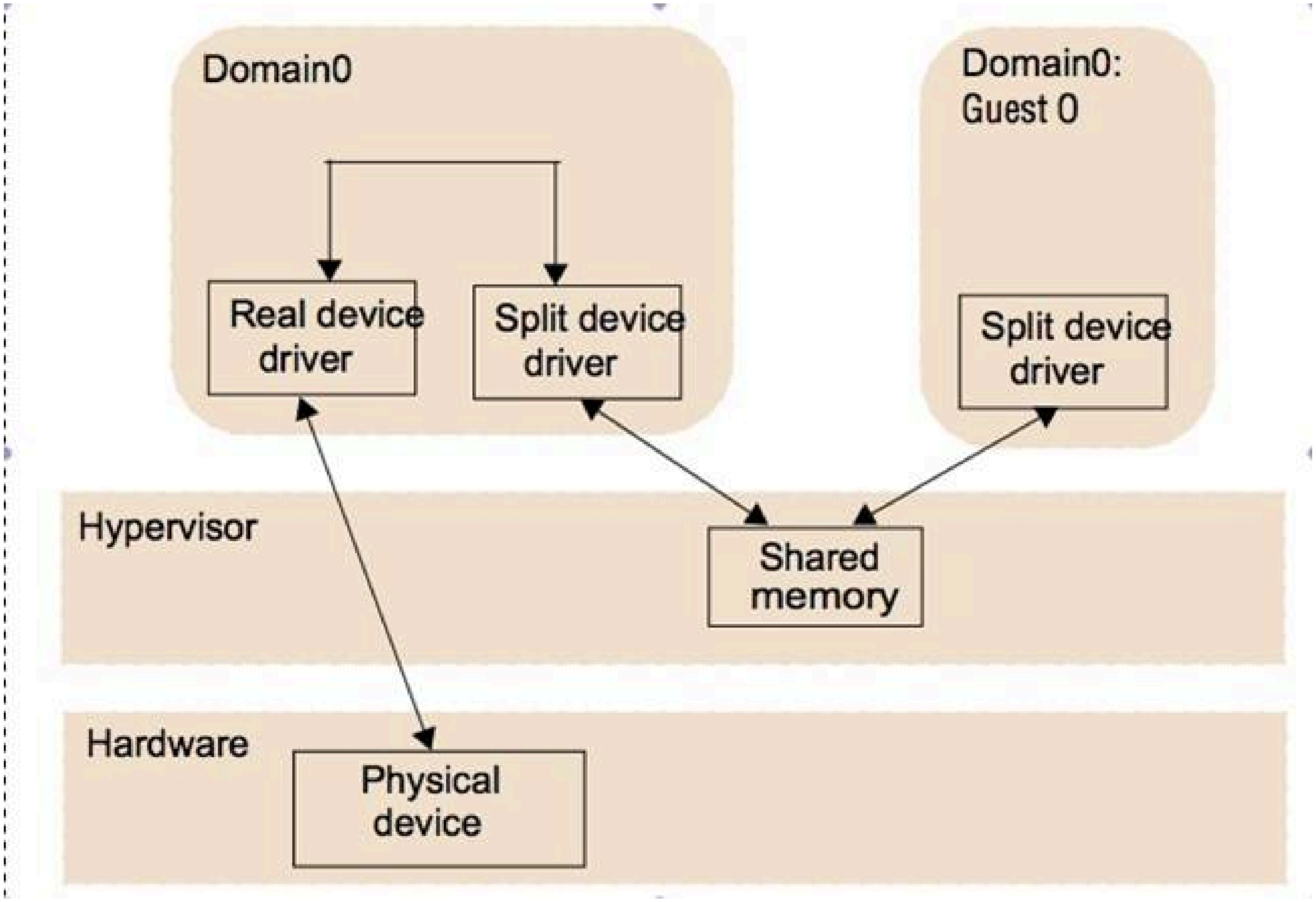
a) kernel-based operating systems

b) paravirtualization in Xen

Virtual Memory



Split Device Drivers


Domain0
Real device driver
Split device driver
Domain0: Guest 0
Split device driver
Hypervisor
Shared memory
Hardware
Physical device