CHAPTER 7

OPERATING SYSTEM SUPPORT

Topics

7.1 Introduction
7.2 The operating system layer
7.3 Protection
7.4 Processes and threads
7.5 Communication and invocation
7.6 Operating system architecture
7.7 Virtualization at the operating system level

Introduction

- Many distributed operating systems have been investigated, but there are none in general/wide use.

- But network operating system are in wide use for various reasons both technical and non-technical.
  - Users have much invested in their application software; they will not adopt a new operating system that will not run their applications.

Introduction

- The second reason against the adoption of distributed operating system is that users tend to prefer to have a degree of autonomy for their machines, even in a organization.

- Unix and Windows are two examples of network operating systems.

- Those have a networking capability built into them and so can be used to access remote resources using basic services such as rlogin and telnet.

Network OS

- Built-in networking capability
- Multiple system images, one per node
- Retain autonomy in managing resources on its own node
- Users have to be involved to schedule processes across the nodes

Distributed OS

- One single system image.
- OS has control over all the nodes in the system.
- May transparently locate new process at whatever node suits its scheduling policies such as load balancing
- Not available in general use.
- Users have much invested in application software.
- Users prefer to have a degree of autonomy for their machines

Middleware + OS

- Provide balance between autonomy and network transparent resource access on the other
- Enables users to take advantage of services that become available in their distributed system.

Introduction

- Middleware enables users to take advantage of services that become available in their distributed system.

The Operating System Layer

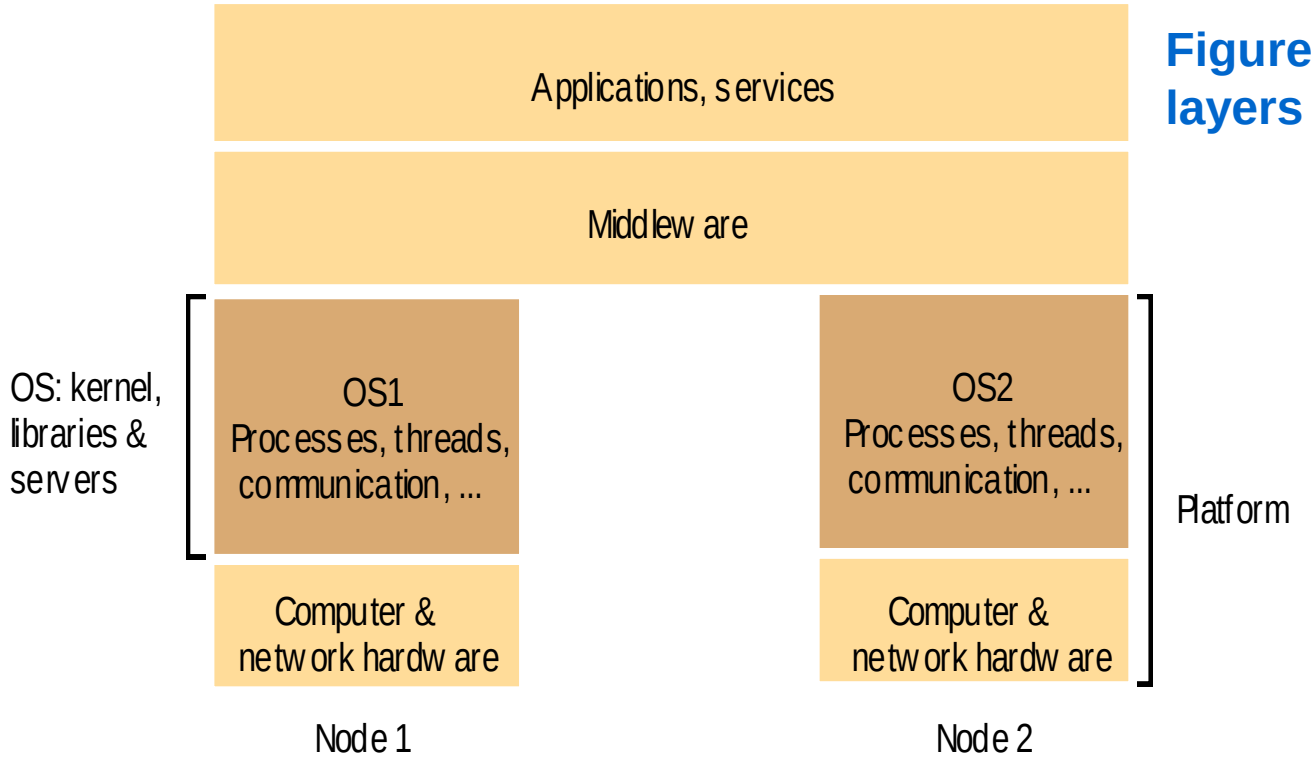


Figure 1. System layers

Figure 7.1 shows how the operating system layer at each of two nodes supports a common middleware layer in providing a distributed infrastructure for applications and services.

The Operating System Layer

- Figure 1 shows how the operating system layer at each of two nodes supports a common middleware layer in providing a distributed infrastructure for applications and services.

- Our goal in this chapter is to examine the impact of particular OS mechanisms on middleware's ability to deliver distributed resource sharing to users. Kernels and the client and server processes that execute upon them are the chief architectural components that concern us.

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

The Operating System Layer

- The OS facilitates:

Encapsulation - They should provide a useful service interface to their resources that is, a set of operations that meet their clients' needs. Details such as 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

Communication- Operation parameters and results have to be passed to and from resource managers, over a network or within a computer.

Scheduling- When an operation is invoked, its processing must be scheduled within the kernel or server.

- **Invocation mechanism is a means of accessing an encapsulated resource.**

7.2 The Functionalities of the components of OS

Process manager - Creation of and operations upon processes. A process is a unit of resource management, including an address space and one or more threads.

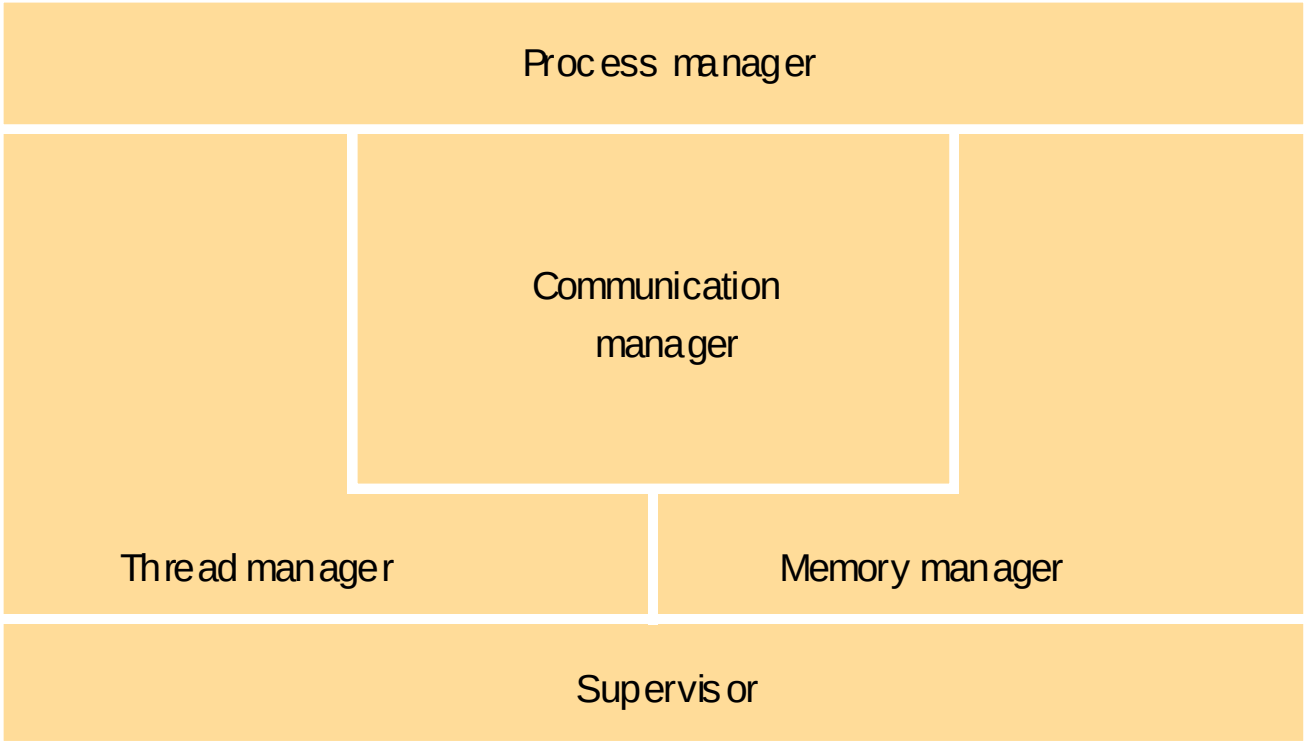
Thread manager - Thread creation, synchronization and scheduling. Threads are schedulable activities attached to processes.

Memory manager - Management of physical and virtual memory for efficient data copying and sharing.

Communication manager -Communication between threads attached to different processes on the same computer. Some kernels also support communication between threads in remote processes.

Supervisor - Dispatching of interrupts, system call traps and other exceptions; control of memory management unit and hardware caches; processor and floating-point unit register manipulations. This is known as the Hardware Abstraction Layer in Windows.

Figure 2. Core OS functionality



This figure shows the core OS functionality that we shall be concerned with: 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.

Shared-memory multiprocessors

Shared-memory multiprocessor computers are equipped with several processors that share one or more modules of memory (RAM).
The processors may also have their own private memory. Multiprocessor computers can be constructed in a variety of forms .

The simplest and least expensive multiprocessors are constructed by incorporating a circuit board holding a few (2–8) processors in a personal computer. In the common *symmetric processing architecture*, each processor executes the same kernel and the kernels play largely equivalent roles in managing the hardware resources. The kernels share key data structures, such as the queue of runnable threads, but some of their working data is private.

- Each processor can execute a thread simultaneously, accessing data in the shared memory, which may be private (hardware-protected) or shared with other threads. Multiprocessors can be used for many high-performance computing tasks.

- In distributed systems, they are particularly useful for the implementation of high performance servers because the server can run a single program with several threads that handle several requests from clients simultaneously – for example, providing access to a shared database

7.3 Protection

- Resources require protection from illegitimate accesses. However, threats to a system's integrity do not come only from maliciously contrived code.

- Benign code that contains a bug or that has unanticipated behaviour may cause part of the rest of the system to behave incorrectly.

- To understand what we mean by an 'illegitimate access' to a resource, consider a file. Let us suppose, for the sake of explanation, that open files have only two operations, *read* and *write*.

- Protecting the file consists of two sub-problems. The first is to ensure that each of the file's two operations can be performed only by clients with the right to perform it.

- For example, Smith, who owns the file, has *read* and *write* rights to it. Jones may only perform the *read* operation. An illegitimate access here would be if Jones somehow managed to perform a *write* operation on the file. A complete solution to this resource-protection sub-problem in a distributed system require cryptographic techniques.

- The other type of illegitimate access, which we address here, is where a misbehaving client sidesteps the operations that a resource exports. In our example, this would be if Smith or Jones somehow managed to execute an operation that was neither *read* nor *write*.

- Suppose, for example, that Smith managed to access the file pointer variable directly. She could then construct a *setFilePointerRandomly* operation, that sets the file pointer to a random number. Of course, this is a meaningless operation that would upset normal use of the file.

- We can protect resources from illegitimate invocations such as *setFilePointerRandomly*. One way is to use a type-safe programming language, such as Sing#, an extension of C# used in the Singularity project or Modula- 3.

Type-safe languages

- In, type-safe languages no module may access a target module unless it has a reference to it
- it cannot make up a pointer to it, as would be possible in C or C++

- it may only use its reference to the target module to perform the invocations (method calls or procedure calls) that the programmer of the target made available to it.

- It may not, in other words, arbitrarily change the target's variables. By contrast, in C++ the programmer may cast a pointer however she likes, and thus perform non-type-safe invocations.
- We can also employ hardware support to protect modules from one another at the level of individual invocations, regardless of the language in which they are written.
- To operate this scheme on a general-purpose computer, we require a kernel.

Kernel & Protection

- The kernel is a program that is distinguished by the facts that it remains loaded from system initialization and its code is executed with complete access privileges for the physical resources on its host computer. In particular, it can control the memory management unit and set the processor registers so that no other code may access the machine's physical resources except in acceptable ways.

- Most processors have a hardware mode register whose setting determines whether privileged instructions can be executed, such as those used to determine which protection tables are currently employed by the memory management unit. A kernel process executes with the processor in *supervisor* (privileged) mode; the kernel arranges that other processes execute in *user* (unprivileged) mode.

- The kernel also sets up *address spaces* to protect itself and other processes from the accesses of an aberrant process, and to provide processes with their required virtual memory layout. An address space is a collection of ranges of virtual memory locations, in each of which a specified combination of memory access rights applies, such as readonly or read-write. A process cannot access memory outside its address space.

- The terms *user process* or *user-level process* are normally used to describe one that executes in user mode and has a user-level address space (that is, one with restricted memory access rights compared with the kernel's address space).

When a process executes application code, it executes in a distinct user-level address space for that application; when the same process executes kernel code, it executes in the kernel's address space. The process can safely transfer from a user-level address space to the kernel's address space via an exception such as an interrupt or a ***system call trap*** – the invocation mechanism for resources managed by the kernel.

A system call trap is implemented by a machine-level ***TRAP*** instruction, which puts the processor into supervisor mode and switches to the kernel address space. When the *TRAP* instruction is executed, as with any type of exception, the hardware forces the processor to execute a kernel-supplied handler function, in order that no process may gain illicit control of the hardware.

Programs pay a price for protection. Switching between address spaces may take many processor cycles, and a system call trap is a more expensive operation than a simple procedure or method call.

7.4 Processes and threads

- Traditional process of traditional operating system make sharing between related activities awkward and expensive.

- The solution reached was to enhance the notion of a process so that it could be associated with multiple activities .Nowadays, a process consists of an execution environment together with one or more threads.

- A *thread* is the operating system abstraction of an activity (the term derives from the phrase ‘thread of execution’). An *execution environment* is the unit of resource management: a collection of local kernel managed resources to which its threads have access.
- An execution environment primarily
  - an address space;
  - thread synchronization and communication resources such as semaphores and communication interfaces (for example, sockets);
  - higher-level resources such as open files and windows. consists of:

An analogy for threads and processes

- The following memorable, if slightly unsavory, way to think of the concepts of threads and execution environments was published on the *comp.os.mach* USENET group and is by Chris Lloyd.

- An execution environment consists of a stoppered jar and the air and food within it. Initially, there is one fly – a thread – in the jar. This fly can produce other flies and kill them, as can its progeny.

- Any fly can consume any resource (air or food) in the jar. Flies can be programmed to queue up in an orderly manner to consume resources. If they lack this discipline, they might bump into one another within the jar – that is, collide and produce unpredictable results when attempting to consume the same resources in an unconstrained manner.

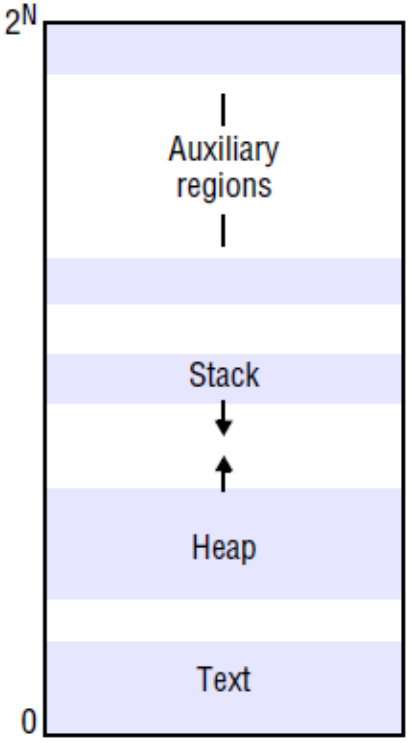
- Flies can communicate with (send messages to) flies in other jars, but none may escape from the jar, and no fly from outside may enter it. In this view, originally a UNIX process was a single jar with a single sterile fly within it.

- An execution environment provides protection from threads outside it, so that the data and other resources contained in it are by default inaccessible to threads residing in other execution environments.

- But certain kernels allow the controlled sharing of resources such as physical memory between execution environments residing at the same computer.

7.4.1 Address spaces

Figure 7.3 Address space



An address space, is a unit of management of a process's virtual memory. It is large (typically up to 232 bytes, and sometimes up to 264 bytes) and consists of one or more *regions*, separated by inaccessible areas of virtual memory.

A region (Figure 7.3) is an area of contiguous virtual memory that is accessible by the threads of the owning process. Regions do not overlap.

Each region is specified by the following properties:

- its extent (lowest virtual address and size);
- read/write/execute permissions for the process's threads;
- whether it can be grown upwards or downwards.

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; and a stack, which is extensible towards lower virtual addresses.

Factors motivating the provision of an indefinite number of regions are:

- One of these is the need to support a separate stack for each thread. Allocating a separate stack region to each thread makes it possible to detect attempts to exceed the stack limits and to control each stack's growth.

- Unallocated virtual memory lies beyond each stack region, and attempts to access this will cause an exception (a page fault). The alternative is to allocate stacks for threads on the heap, but then it is difficult to detect when a thread has exceeded its stack limit.

- Another motivation is to enable files in general – not just the text and data sections of binary files – to be mapped into the address space. A *mapped file* is one that is accessed as an array of bytes in memory. The virtual memory system ensures that accesses made in memory are reflected in the underlying file storage.

- The need to share memory between processes, or between processes and the kernel, is another factor leading to extra regions in the address space.
- A *shared memory region* (or *shared region* for short) is one that is backed by the same physical memory as one or more regions belonging to other address spaces. Processes therefore access identical memory contents in the regions that are shared, while their non-shared regions remain protected.

- The uses of shared regions include the following:
  - ***Libraries***: Library code can be very large and would waste considerable memory if it was loaded separately into every process that used it. Instead, a single copy of the library code can be shared by being mapped as a region in the address *spaces of processes that require it.*
  - ***Kernel****: Often the kernel code and data are mapped into every address space at the same location. When a process makes a system call or an exception occurs, there is no need to switch to a new set of address mappings.*
  - ***Data sharing and communication****: Two processes, or a process and the kernel, might need to share data in order to cooperate on some task. It can be considerably more efficient for the data to be shared by being mapped as regions in both address spaces than by being passed in messages between them*

7.4.2 Creation of a new process

- For a distributed system, the design of the process-creation mechanism has to take into account the utilization of multiple computers; consequently, the process-support infrastructure is divided into separate system services.
- The creation of a new process can be separated into two independent aspects:
  - The choice of a target host, for example, the host may be chosen from among the nodes in a cluster of computers acting as a compute server
  - The creation of an execution environment (and an initial thread within it)

- **Choice of process host :** The choice of the node at which the new process will reside – the process allocation decision – is a matter of policy. In general, process allocation policies range from always running new processes at their originator's workstation to sharing the processing load between a set of computers. Two policy categories for load sharing are - *transfer policy & location policy*

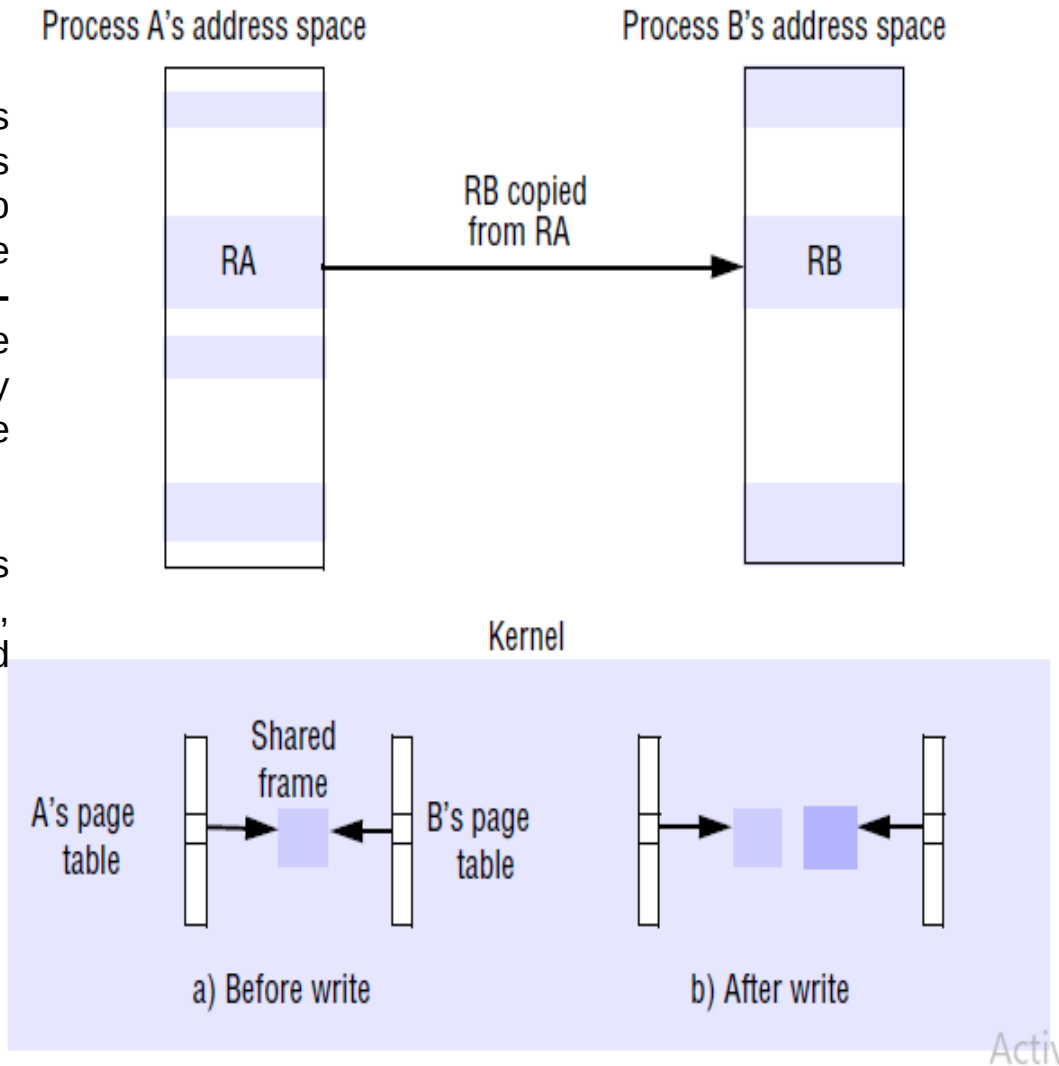
- The ***transfer policy*** determines whether to situate a new process locally or remotely. This may depend, for example, on whether the local node is lightly or heavily loaded.
- The ***location policy*** determines which node should host a new process selected for transfer. This decision may depend on the relative loads of nodes, on their machine architectures or on any specialized resources they may possess.
- The **V system** and **Sprite Systems** both provide commands for users to execute a program at a currently idle workstation (there are often many of these at any given time) chosen by the operating system.
- In the **Amoeba system** the *run server* chooses a host for each process from a shared pool of processors.
- In all cases, the choice of target host is transparent to the programmer and the user.
- Those programming for explicit parallelism or fault tolerance, however, may require a means of specifying process location.
- **Process location policies may be *static* or *adaptive***. They are based on a mathematical analysis aimed at optimizing a parameter such as the overall process throughput. They may be **deterministic** ('node A should always transfer processes to node B') or **probabilistic** ('node A should transfer processes to any of nodes B–E at random').
- **Load-sharing systems may be centralized, hierarchical or decentralized**.
- In ***sender-initiated*** load-sharing algorithms, the node that requires a new process to be created is responsible for initiating the transfer decision.
- ***Migratory*** **load-sharing systems** can shift load at any time, not just when a new process is created. They use a mechanism called *process migration*.

Creation of a new execution environment

- Once the host computer has been selected, a new process requires an execution environment consisting of an address space with initialized contents (and perhaps other resources, such as default open files).
- **Address space can be of a statically defined format**. For example, it could contain just a program text region, heap region and stack region. In this case, the address space regions are created from a list specifying their extent. Address space regions are initialized from an executable file or filled with zeros as appropriate.

- The address space can also be defined with respect to an existing execution environment. In the case of UNIX *fork* semantics, for example, the newly created child process physically shares the parent's text region and has heap and stack regions that are copies of the parent's in extent (as well as in initial contents).

Figure 7.4 Copy-on-write



Initially, all page frames associated with the regions are shared between the two processes' page tables. The pages are initially **write-protected** at the hardware level, even though they may belong to regions that are logically writable

If a thread in either process attempts to modify the data, a hardware exception called a *page fault* is taken

The **page fault handler** allocates a new frame for process *B* and copies the original frame's data into it byte for byte. The old frame number is replaced by the new frame number in one process's page table.

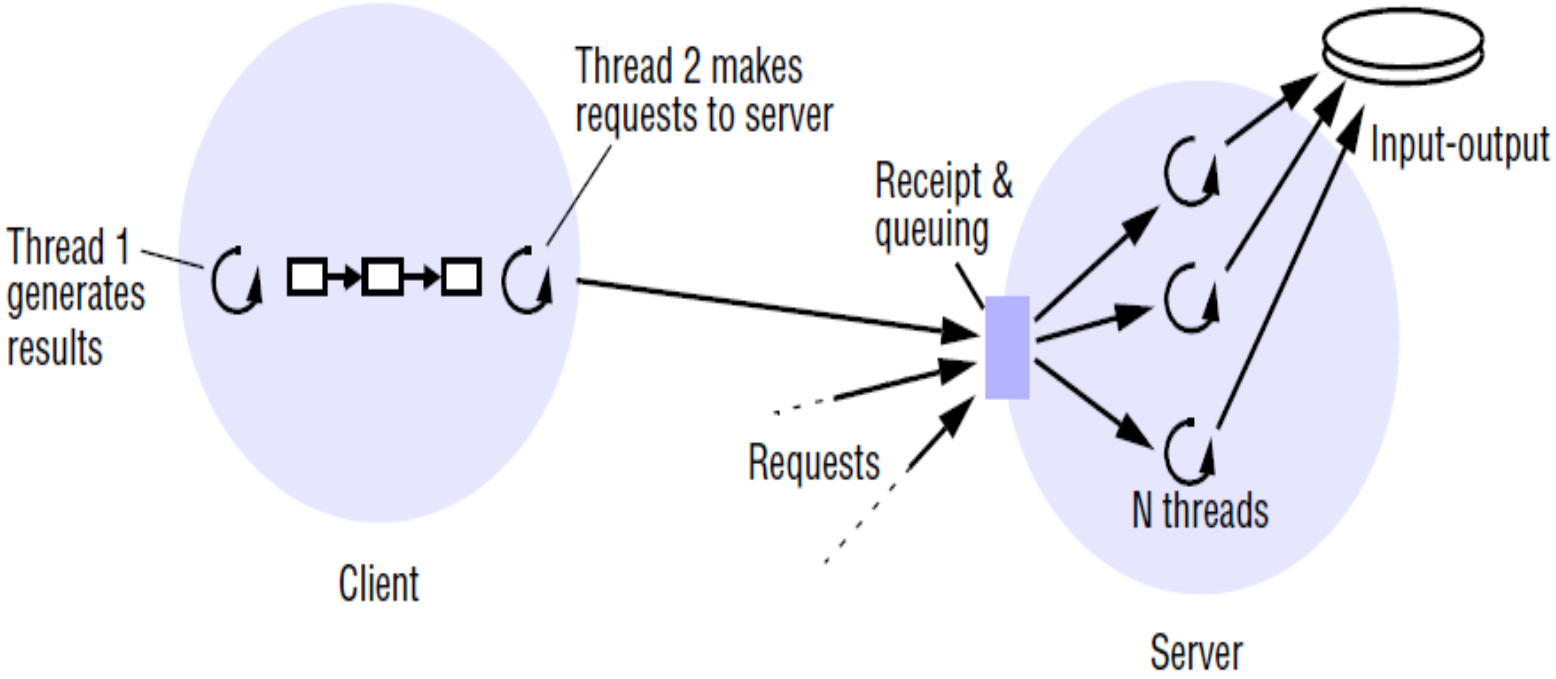
The two corresponding pages in processes *A* and *B* are then each made writable once more at the hardware level.

- 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 happens* 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.

7.4.3 Threads

Figure 7.5 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.

Let us assume that each request takes, on average, 2 milliseconds of processing plus 8 milliseconds of I/O (input/output) delay when the server reads from a disk (there is no caching). Let us further assume for the moment that the server executes at a single-processor computer.

- Let us further assume for the moment that the server executes at a single-processor computerived requests and processes it.

- Consider the *maximum* server throughput, measured in client requests handled per second, for different numbers of threads. If a single thread has to perform all processing, then the turnaround time for handling any request is on average 2 + 8 = 10 milliseconds, so this server can handle 100 client requests per second. Any new request messages that arrive while the server is handling a request are queued at the server port.

- Now consider what happens if the server pool contains two threads. We assume that threads are independently schedulable – that is, one thread can be scheduled when another becomes blocked for I/O. Then thread number two can process a second request while thread number one is blocked, and vice versa. This increases the server throughput. Unfortunately, in our example, the threads may become blocked behind the single disk drive. If all disk requests are serialized and take 8 milliseconds each, then the maximum throughput is 1000/8 = 125 requests per second.

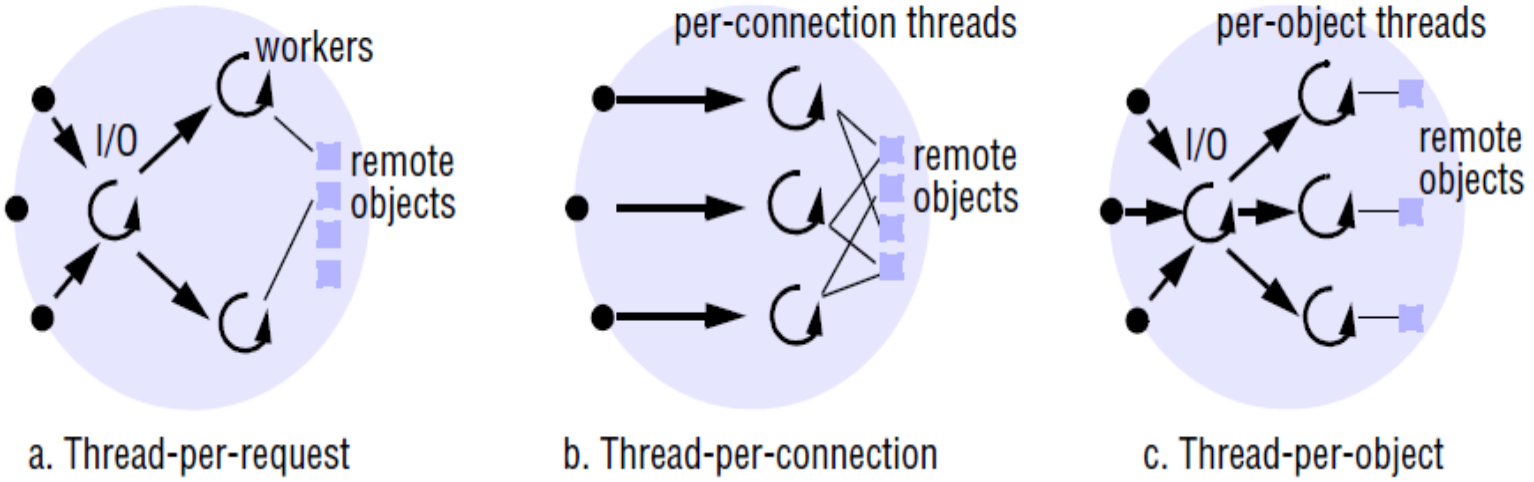
- Suppose, now, that disk block caching is introduced. The server keeps the data that it reads in buffers in its address space; a server thread that is asked to retrieve data first examines the shared cache and avoids accessing the disk if it finds the data there.
- If a 75% hit rate is achieved, the mean I/O time per request reduces to (0.75X0 + 0.25x8) = 2 milliseconds, and the maximum theoretical throughput increases to 500 requests per second.
- But if the average *processor* time for a request has been increased to 2.5 milliseconds per request as a result of caching (it takes time to search for cached data on every operation), then this figure cannot be reached. The server, limited by the processor, can now handle at most 1000/2.5 = 400 requests per second.
- The throughput can be increased by using a **shared-memory multiprocessor** to ease the processor bottleneck. A multi-threaded process maps naturally onto a shared memory multiprocessor. The shared execution environment can be implemented in shared memory, and the multiple threads can be scheduled to run on the multiple processors. Consider now the case in which our example server executes at a multiprocessor with two processors.
- Given that threads can be independently scheduled to the different processors, then up to two threads can process requests in parallel.

Architectures for multi-threaded servers

- Figure 7.5 shows one of the possible threading architectures, the ***worker pool architecture***. In its simplest form, the server creates a fixed pool of 'worker' threads to process the requests when it starts up. The module marked '**receipt and queuing**' in Figure 7.5 is typically implemented by an 'I/O' thread, which receives requests from a collection of sockets or ports and places them on a shared request queue for retrieval by the workers.

- A disadvantage of this architecture is its inflexibility: as we saw with our worked-out example, the number of worker threads in the pool may be too few to deal adequately with the current rate of request arrival. Another disadvantage is the high level of switching between the I/O and worker threads as they manipulate the shared queue.

Figure 7.6 Alternative server threading architectures (see also Figure 7.5)



In the ***thread-per-request architecture*** (Figure 7.6a) the I/O thread spawns a new worker thread for each request, and that worker destroys itself when it has processed the request against its designated remote object. This architecture has the advantage that the threads do not contend for a shared queue, and throughput is potentially maximized because the I/O thread can create as many workers as there are outstanding requests. Its disadvantage is the overhead of the thread creation and destruction operations.

- The ***thread-per-connection architecture*** (Figure 7.6b) associates a thread with each connection. The server creates a new worker thread when a client makes a connection and destroys the thread when the client closes the connection. In between, the client may make many requests over the connection, targeted at one or more remote objects.

- The ***thread-per-object architecture*** (Figure 7.6c) associates a thread with each remote object. An I/O thread receives requests and queues them for the workers, but this time there is a per-object queue. In each of these last two architectures the server benefits from lower thread management overheads compared with the thread-per-request architecture. Their disadvantage is that clients may be delayed while a worker thread has several outstanding requests but another thread has no work to perform.

Threads within clients

Threads can be useful for clients as well as servers. Figure 7.5 also shows a client process with two threads. The first thread generates results to be passed to a server by remote method invocation, but does not require a reply.

Remote method invocations typically block the caller, even when there is strictly no need to wait. This client process can incorporate a second thread, which performs the remote method invocations and blocks while the first thread is able to continue computing further results.

The first thread places its results in buffers, which are emptied by the second thread. It is only blocked when all the buffers are full. The case for multi-threaded clients is also evident in the example of web browsers. Users experience substantial delays while pages are fetched; it is essential therefore, for browsers to handle multiple concurrent requests for web pages.

Threads versus multiple processes

A comparison of processes and threads is as follows:

- Threads are cheaper to create and manage than processes, and resource sharing can be achieved more efficiently between threads than between processes because threads share an execution environment.
- 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.
- The overheads associated with creating a process are in general considerably greater than those of creating a new thread. A new execution environment must first be created, including address space tables.

The main state components that must be maintained for execution environments and threads, respectively. An execution environment has an address space, communication interfaces such as sockets, higher-level resources such as open files and thread synchronization objects such as semaphores; it also lists thethreads associated with it. A thread has a scheduling priority, an execution state (such as *BLOCKED* or *RUNNABLE*), saved processor register values when the thread is *BLOCKED*, and state concerning the thread's software interrupt handling. A *software interrupt* is an event that causes a thread to be interrupted

Figure 7.7 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	

A thread has a scheduling priority, an execution state (such as *BLOCKED* or *RUNNABLE*), saved processor register values when the thread is *BLOCKED*, and state concerning the thread's software interrupt handling.

A ***software interrupt*** is an event that causes a thread to be interrupted (similar to the case of a hardware interrupt). If the thread has assigned a handler procedure, control is transferred to it. UNIX signals are examples of software interrupts.

Translation lookaside buffer (TLB)

- Memory management units usually include a hardware cache to speed up the translation between virtual and physical addresses, called a *translation lookaside buffer* (TLB). TLBs, and also virtually addressed data and instruction caches, suffer in general from the so-called *aliasing problem*.

- The same virtual address can be valid in two different address spaces, but in general it is supposed to refer to different physical data in the two spaces. Unless their entries are tagged with a context identifier, TLBs and virtually addressed caches are unaware of this and so might contain incorrect data. Therefore the TLB and cache contents have to be flushed on a switch to a different address space

Context Switch

- A *context switch* is the transition between contexts that takes place when switching between threads, or when a single thread makes a system call or takes another type of exception. It involves the following:
  - the saving of the processor's original register state, and the loading of the new state;
  - in some cases, a transfer to a new protection domain – this is known as a domain transition.
- Switching between threads sharing the same execution environment entirely at user level involves no domain transition and is relatively cheap. Switching to the kernel, or to another thread belonging to the same execution environment via the kernel, involves a domain transition.
- The cost is therefore greater but it is still relatively low if the kernel is mapped into the process's address space. When switching between threads belonging to different execution environments, however, there are greater overheads.

Thread Programming & Thread Lifetimes

Figure 7.8 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()
Sets and returns 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()
Changes the state of the thread from *SUSPENDED* to *RUNNABLE*.

sleep(long millisecs)
Causes the thread to enter the *SUSPENDED* state for the specified time.

yield()
Causes the thread to enter the *READY* state and invokes the scheduler.

destroy()
Destroys the thread.

Figure 7.9 Java thread synchronization calls

thread.join(long millisecs)

Blocks the calling thread for up to the specified time or 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, 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*.

Threads implementation

- Threads can be kernel threads or user-level threads.
- When no kernel support for multi-threaded processes is provided, a user-level threads implementation suffers from the following problems:
  - The threads within a process cannot take advantage of a multiprocessor.
  - A thread that takes a page fault blocks the entire process and all threads within it.
  - Threads within different processes cannot be scheduled according to a single scheme of relative prioritization.
- **User-level threads implementations, on the other hand, have significant advantages over kernel-level implementations-**

- **c**ertain thread operations are significantly less costly.
- Given that the thread-scheduling module is implemented outside the kernel, it can be customized or changed to suit particular application requirements.
- Many more user-level threads can be supported than could reasonably be provided by default by a kernel.

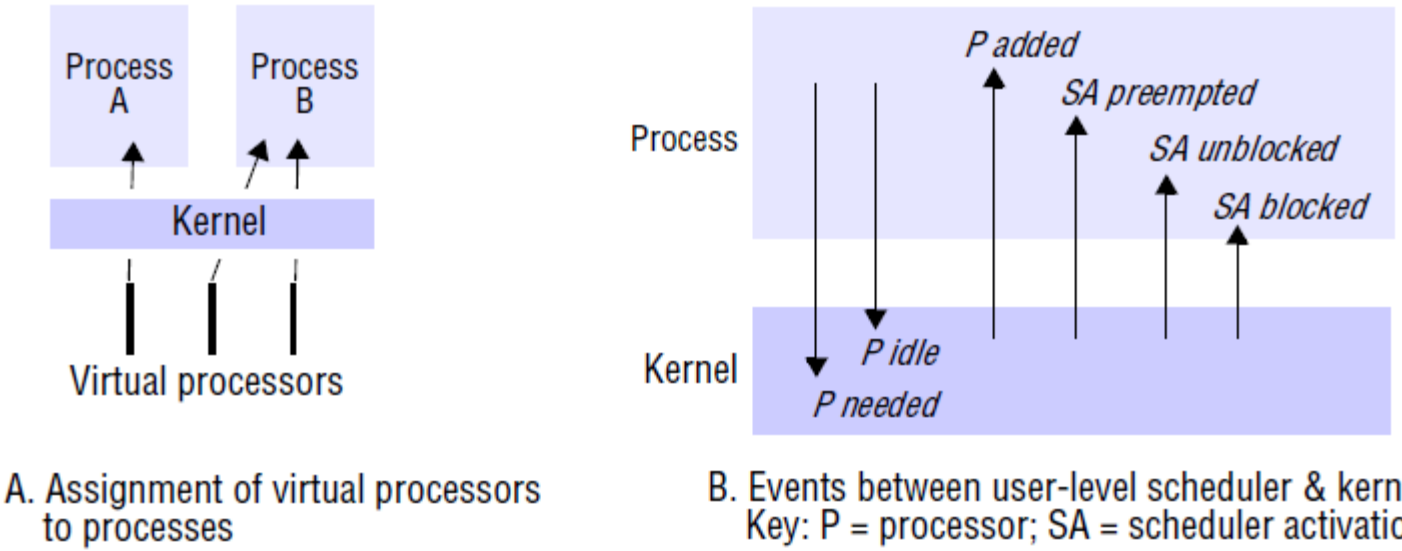
Scheduler activations design

- The insight driving these designs is that what a user-level scheduler requires from the kernel is not just a set of kernel-supported threads onto which it can map user-level threads.

- The user-level scheduler also requires the kernel to notify it of the *events* that are relevant to its scheduling decisions.

- In hierarchic, event-based scheduling system, the main system components are a kernel running on a computer with one or more processors, and a set of application programs running on it. Each application process contains a user-level scheduler, which manages the threads inside the process. The kernel is responsible for allocating *virtual processors* to processes.

Figure 7.10 Scheduler activations



A. Assignment of virtual processors to processes

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

Figure 7.10(a) shows an example of a three-processor machine, on which the kernel allocates one virtual processor to process *A*, running a relatively low-priority job, and two virtual processors to process *B*. They are *virtual* processors because the kernel can allocate different physical processors to each process as time goes by, while keeping its guarantee of how many processors it has allocated.

Figure 7.10(b) shows that a process notifies the kernel when either of two types of event occurs: when a virtual processor is 'idle' and no longer needed, or when an extra virtual processor is required.

- Figure 7.10(b) also shows that the kernel notifies the process when any of four types of event occurs. A *scheduler activation* (SA) is a call from the kernel to a process, which notifies the process's scheduler of an event. Entering a body of code from a lower layer (the kernel) in this way is sometimes called an *upcall*.

- The kernel creates an SA by oading a physical processor's registers with a context that causes it to commence execution of code in the process, at a procedure address designated by the user-level scheduler.

- An SA is thus also a unit of allocation of a timeslice on a virtual processor. The user-level scheduler has the task of assigning its *READY* threads to the set of Sas currently executing within it. The number of those SAs is at most the number of virtual processors that the kernel has assigned to the process

The four types of event that the kernel notifies the user-level scheduler are -

- ***Virtual processor allocated*:** The kernel has assigned a new virtual processor to the process, and this is the first timeslice upon it; the scheduler can load the SA with the context of a *READY* thread, which can thus recommence execution.
- ***SA blocked*:** An SA has blocked in the kernel, and the kernel is using a fresh SA to notify the scheduler; the scheduler sets the state of the corresponding thread to *BLOCKED* and can allocate a *READY* thread to the notifying SA.
- ***SA unblocked***: An SA that was blocked in the kernel has become unblocked and is ready to execute at user level again; the scheduler can now return the corresponding thread to the *READY* list.
- ***SA preempted***: The kernel has taken away the specified SA from the process (although it may do this to allocate a processor to a fresh SA in the same process); the scheduler places the preempted thread in the *READY* list and reevaluates the thread allocation.

7.5 Communication and invocation

We cover operating system design issues and concepts by asking the following questions about the OS:

- What communication primitives does it supply?
- Which protocols does it support and how open is the communication implementation?
- What steps are taken to make communication as efficient as possible?
- What support is provided for high-latency and disconnected operation?

Communication primitives

- In practice, middleware, and not the kernel, provides most high-level communication facilities found in systems today, including RPC/RMI, event notification and group communication.

- Developing such complex software as user-level code is much simpler than developing it for the kernel. Developers typically implement middleware over sockets giving access to Internet standard protocol sockets using TCP but sometimes unconnected UDP sockets.

- The principal reasons for using sockets are portability and interoperability: middleware is required to operate over as many widely used operating systems as possible, and all common operating systems, such as UNIX and the Windows family, provide similar socket APIs giving access to TCP and UDP protocols.

Protocols and openness

- One of the main requirements of the operating system is to provide standard protocols that enable interworking between middleware implementations on different platforms. Several research kernels developed in the 1980s incorporated their own network protocols tuned to RPC interactions – notably Amoeba RPC.

- Given the everyday requirement for access to the Internet, compatibility at the level of TCP and UDP is required of operating systems for all but the smallest of networked devices.

- Protocols are normally arranged in a *stack* of layers.

7.5.1 Invocation performance

- Software overheads often predominate over network overheads in invocation times – at least, for the case of a LAN or intranet. This is in contrast to a remote invocation over the Internet – for example, fetching a web resource.
- On the Internet, network latencies are highlyvariable and relatively high on average; throughput may be relatively low, and server load often predominates over per-request processing costs. For an example of latencies, Bridges *et al.* [2007] report minimal UDP message round-trips taking average times of about 400 milliseconds over the Internet between two computers connected across US geographical regions, as opposed to about 0.1 milliseconds when identical computers were connected over a single Ethernet.
- **Invocation costs** • Calling a conventional procedure or invoking a conventional method, making a system call, sending a message, remote procedure calling and remote method invocation are all examples of invocation mechanisms. Each mechanism causes code to be executed outside the scope of the calling procedure or object.

Figure 7.11 Invocations between address spaces

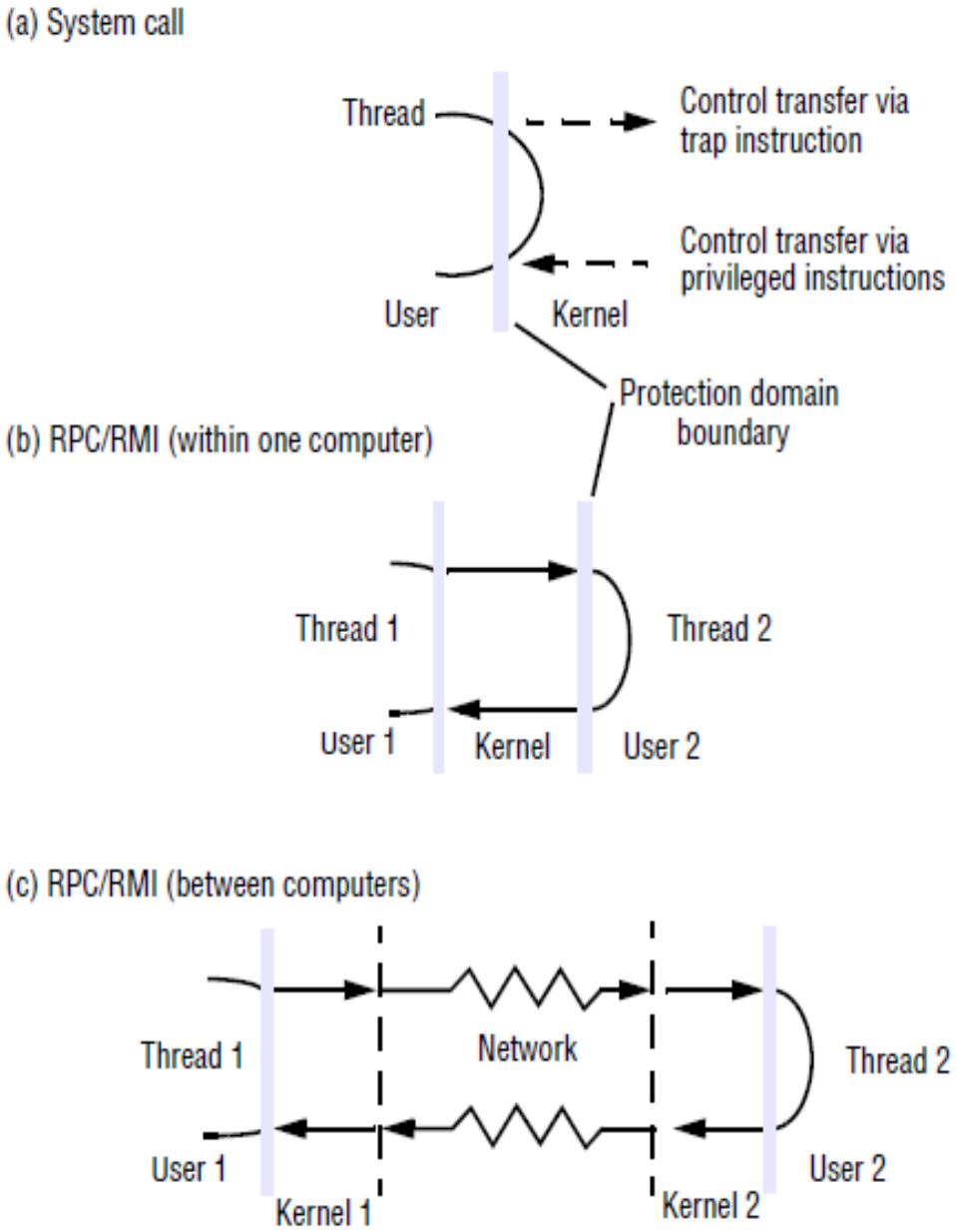


Figure 7.11 shows the particular cases of a system call, a remote invocation between processes hosted at the same computer, and a remote invocation between processes at different nodes in the distributed system

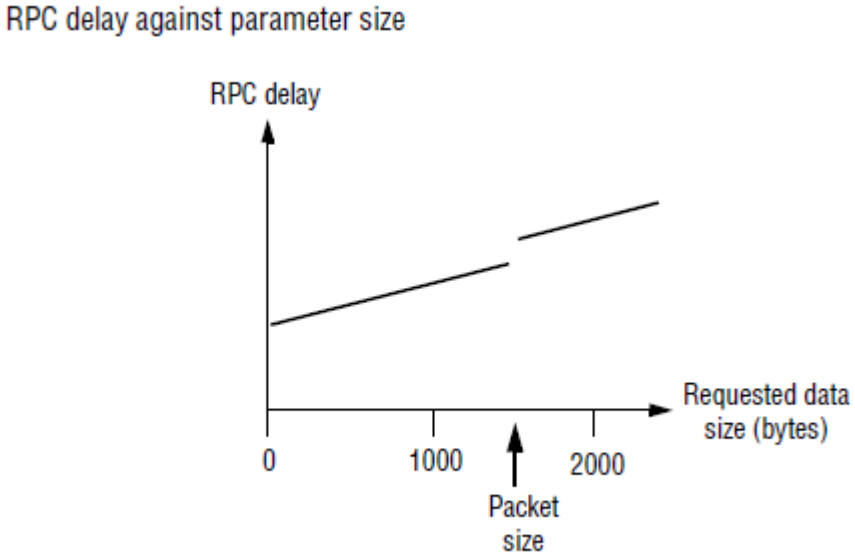
The important performance-related distinctions between invocation mechanisms, apart from whether or not they are synchronous, are whether they involve a domain transition (that is, whether they cross an address space), whether they involve communication across a network and whether they involve thread scheduling and switching.

Invocation over the network

- A *null RPC* (and similarly, a *null RMI*) is defined as an RPC without parameters that executes a null procedure and returns no values. Its execution involves an exchange of messages carrying some system data but no user data.

- The time taken by a null RPC between user processes connected by a LAN is on the order of a tenth of a millisecond.

- By comparison, a null conventional procedure call takes a small fraction of a microsecond. Approximately 100 bytes in total are passed across the network for a null RPC. With a raw bandwidth of 100 megabits/second, the total network transfer time for this amount of data is about 0.01 milliseconds



The delay is roughly proportional to the size until the size reaches a threshold at about network packet size.

As the amount of data is increased, the throughput rises as those overheads become less significant.

Recall that the steps in an RPC are as follows (RMI involves similar steps):

- A client stub marshals the call arguments into a message, sends the request message and receives and unmarshals the reply.
- At the server, a worker thread receives the incoming request, or an I/O thread receives the request and passes it to a worker thread; in either case, the worker calls the appropriate server stub.
- **T**he server stub unmarshals the request message, calls the designated procedure, and marshals and sends the reply.

The following are the main components accounting for remote invocation delay, besides network transmission times:

- ***Marshalling***: Marshalling and unmarshalling, which involve copying and converting data, create a significant overhead as the amount of data grows.

Data copying: Potentially, even after marshalling, message data is copied several times in the course of an RPC:

- across the user–kernel boundary, between the client or server address space and kernel buffers;
- across each protocol layer (for example, RPC/UDP/IP/Ethernet);
- between the network interface and kernel buffers

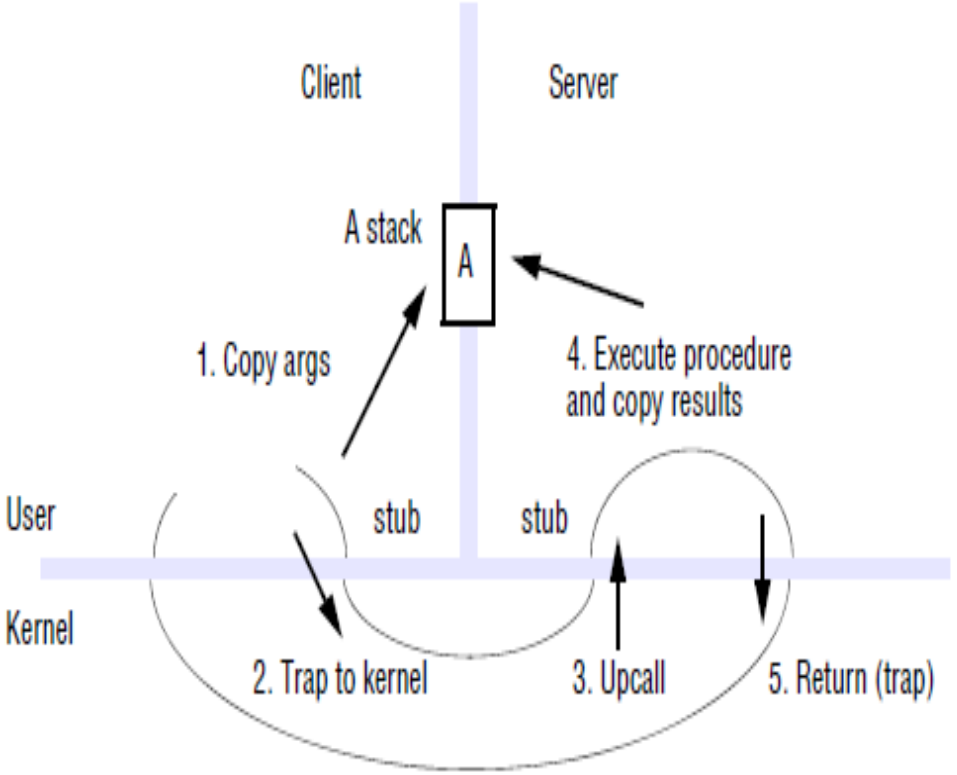
Packet initialization: This involves initializing protocol headers and trailers, including checksums. The cost is therefore proportional, in part, to the amount of data sent.

Thread scheduling and context switching: These may occur as follows:

- Several system calls (that is, context switches) are made during an RPC, as stubs invoke the kernel's communication operations.
- One or more server threads is scheduled.
- If the operating system employs a separate network manager process, then each *Send* involves a context switch to one of its threads.

Waiting for acknowledgements: The choice of RPC protocol may influence delay, particularly when large amounts of data are sent.

Figure 7.13 A lightweight remote procedure call



The client and server are able to pass arguments and return values directly via an A stack.

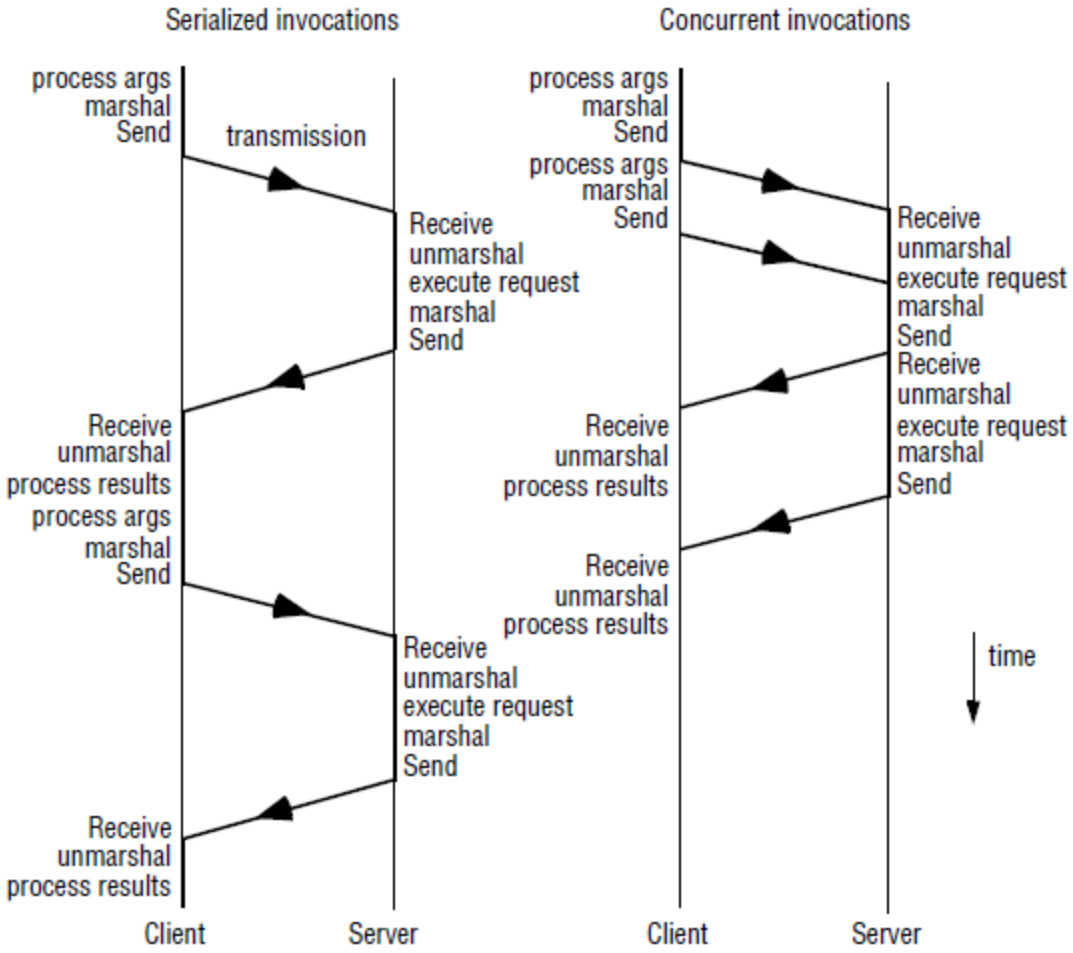
The same stack is used by the client and server stubs.
In *lightweight RPC (*LRPC), arguments are copied once: when they are marshalled onto the A stack. In an equivalent RPC, they are copied four times:
1. from the client stub's stack onto a message;
2. from the message to a kernel buffer,
3. from the kernel buffer to a server message, and
4. from the message to the server stub's stack.

There may be several A stacks in a shared region, because several threads in the same client may call the server at the same time.

7.5.2 Asynchronous operation

A common technique to defeat high latencies of Internet is asynchronous operation, which arises in two programming models: concurrent invocations and asynchronous invocations.

Figure 7.14 Times for serialized and concurrent invocations



- Figure 7.14 shows the potential benefits of interleaving invocations (such as HTTP requests) between a client and a single server on a single-processor machine. In the serialized case, the client marshals the arguments, calls the *Send* operation and then waits until the reply from the server arrives – whereupon it *Receives*, unmarshals and then processes the results. After this it can make the second invocation.

- Sometimes the client does not require any response (except perhaps an indication of failure if the target host could not be reached). For example, CORBA *oneway* invocations have *maybe* semantics. Otherwise, the client uses a separate call to collect the results of the invocation.

- A system for persistent asynchronous invocation tries indefinitely to perform the invocation, until it is known to have succeeded or failed, or until the application cancels the invocation. An example is Queued RPC (QRPC) in the Rover toolkit for mobile information access.

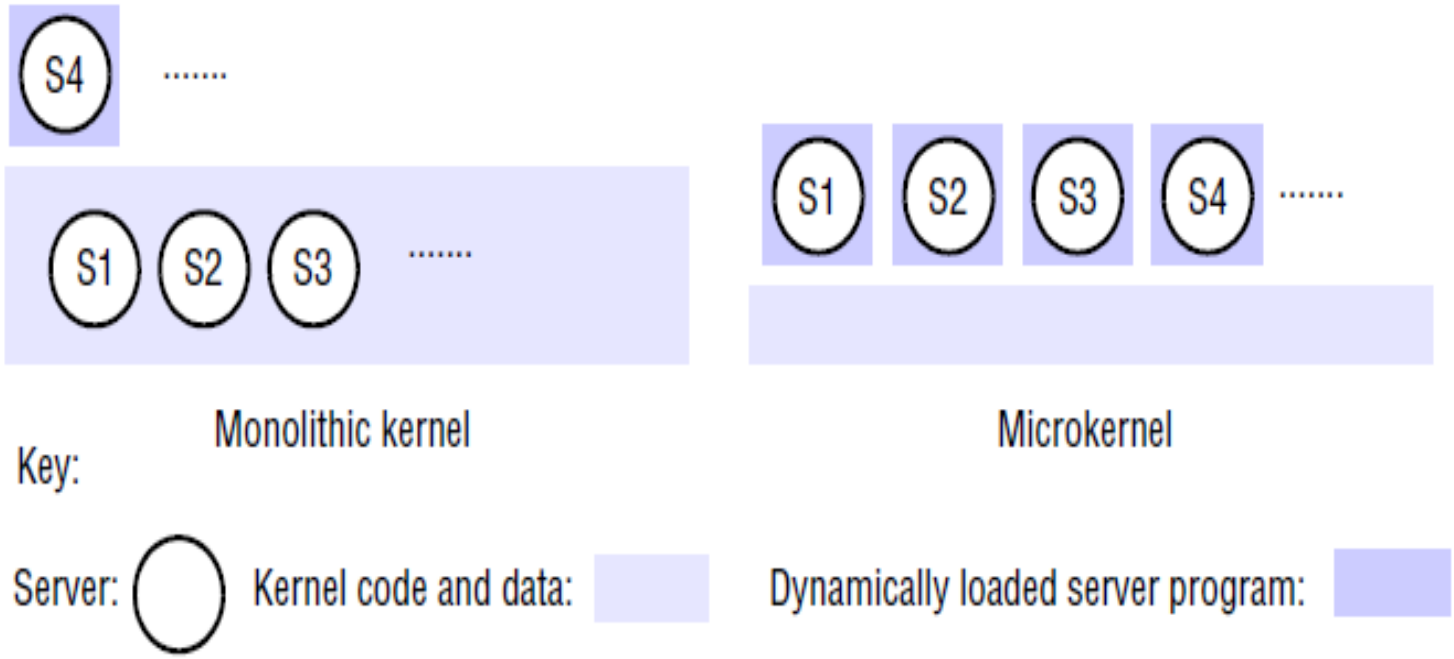
7.6 Operating system architecture

An open distributed system should make it possible to:

- run only that system software at each computer that is necessary for it to carry out its particular role in the system architecture – system software requirements can vary between, for example, mobile phones and server computers, and loading redundant modules wastes memory resources;
- allow the software (and the computer) 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.

The separation of fixed resource management *mechanisms* from resource management *policies*, which vary from application to application and service to service, has been a guiding principle in operating system design for a long time

Figure 7.15 Monolithic kernel and microkernel



There are two key examples of kernel design: the so-called *monolithic* and *microkernel* approaches. These designs differ primarily in the decision as to what functionality belongs in the kernel and what is to be left to server processes that can be dynamically loaded to run on top of it. Although microkernels have not been deployed widely, it is instructive to understand their advantages and disadvantages compared with the typical kernels found today.