Unit – IV – Operating System Support

5.1. Introduction: In distributed systems, the middleware applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages. Some middleware, such as java Remote Method Invocation (RMI) supports only a single programming language. But all middleware deals with the operating system and hardware.

The middleware layer uses protocols based on messages b/w processes to provide its higher-level abstractions. The main important aspect of middleware is the provision of location transparency and independence from the details of communication protocols, operating systems and the computer hardware.

The task of any operating system is to provide problem-oriented abstractions of the underlying physical resources such as the processors, memory, communications and storage media. An operating system such as UNIX or Windows-XP provides the programmer with files rather than disk blocks, and with sockets rather than raw network access. It takes over the physical resources on a single node and manages them to present these resources abstractions through the system-call interface.

Both UNIX and Windows-XP are examnples of *network operating systems*. They have a networking capability built into them and so can be used to access remote resources. With the network operating system, a user can remotely log in to another computer, using *rlogin* or *telnet*, and run processes there.

An operating system that produces a single system image like this for all the resouces in a distributed system is called a *distributed operating system*.

Middleware and network operating system: In fact, there is no distributed operating systems in general use, only network operating system such as UNIX, Mac OS, and Windows-XP. Because of these following two reasons.

The first reason is that the users have much invested in their application software. So they refused to adopt new operating system, whatever efficiency advantages it offers.

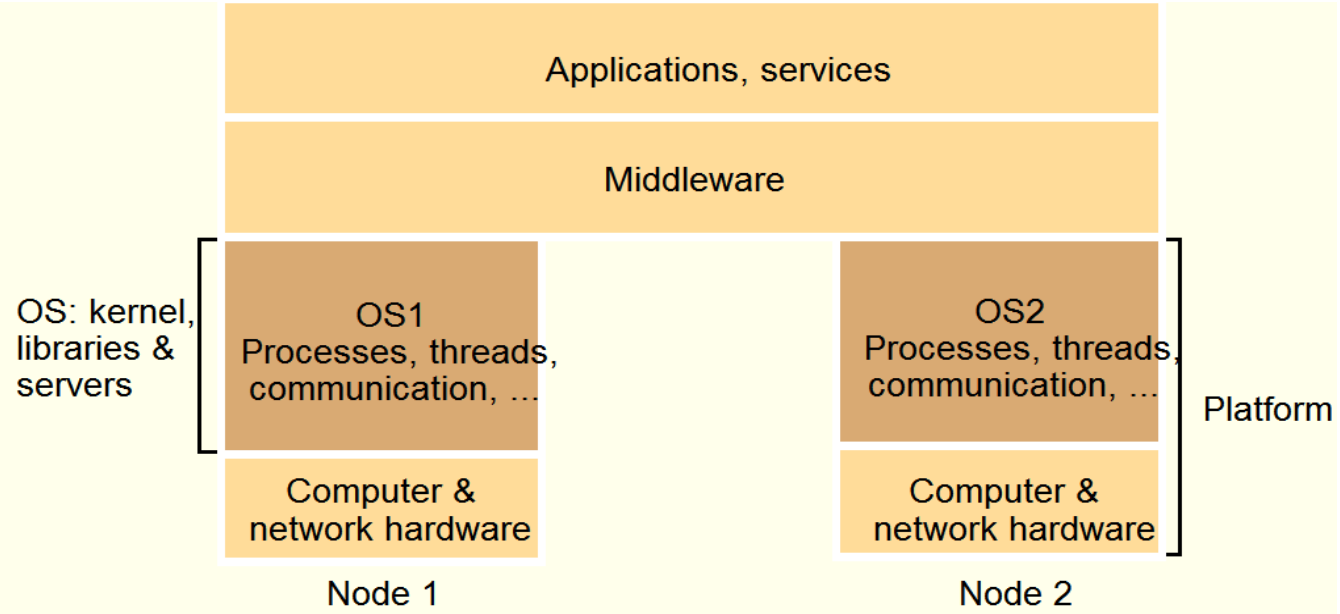
The second reason is that users tend to prefer to have a degree of autonomy for their machines because of its performance.

For example, vasanthi needs good interactive responsiveness while she writes her documents and would dislike it if srilatha's programs were slowing her down.

The combination of middleware and network operating systems provides an acceptable balance b/w the requirement for autonomy on the one hand, and network-transparent resource access on the other.

The network operating system enables users to run their favourite word processor and other standalone applications. Middleware enables them to take advantage of services that become available in their deistributed system.

5.2. The Operating system layer: The good performance of middleware and OS combination, the users are satisfied to distribute the data in distributed networking technology. In this case, middleware runs on a variety of OS platforms at the nodes of a distributed system. The OS running at a node with a librariy service which provides local hardware resources for processing, storage and communication.



The above figure 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.

→Middleware and the Operating System

- Middleware implements abstractions that support network-wide programming. Examples:
    - RPC and RMI (Sun RPC, Corba, Java RMI)
    - event distribution and filtering (Corba Event Notification, Elvin)
    - resource discovery for mobile and ubiquitous computing
    - support for multimedia streaming
- Traditional OS's (e.g. early Unix, Windows 3.0)
    - simplify, protect and optimize the use of local resources
- Network OS's (e.g. Mach, modern UNIX, Windows NT)
    - do the same but they also support a wide range of communication standards and enable remote processes to access (some) local resources (e.g. files).

→ Combination of middleware and network OS

- No distributed OS in general use
    - Users have much invested in their application software
    - Users tend to prefer to have a degree of autonomy for their machines
- Network OS provides autonomy
- Middleware provides network-transparent access resource

→ The relationship between OS and Middleware

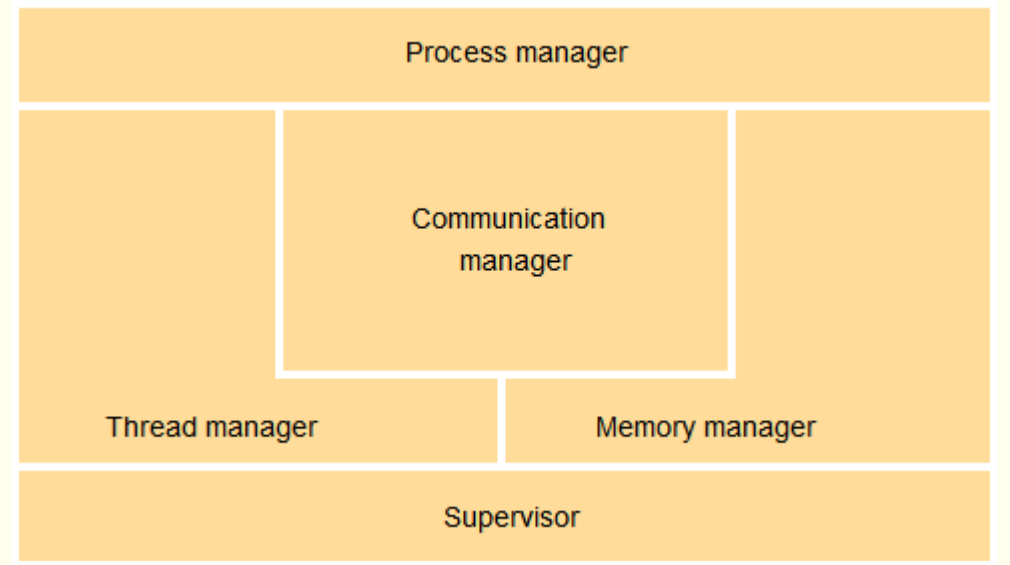
- The tasks of Operating System are processing, storage and communication
- The Components of Operating System are kernel, library, user-level services

- Middleware
    - It runs on a variety of OS-hardware combinations
    - remote invocations

→ **Functions that OS should provide for middleware**

- Encapsulation
  - provide a set of operations that meet their clients' needs
- Protection
  - Protect resource from illegitimate access. For example, files are protected from being read by users without read permissions, and devices registers are protected from application processes.
- Concurrent processing
  - support clients access resource concurrently
- Invocation mechanism: a means of accessing an encapsulated resource
  - Communication
    - Pass operation parameters and results between resource managers
  - Scheduling
    - Schedule the processing of the invoked operation

5.2.1. Core OS functionality:



- **Process manager**
  - Handles the creation of and operations upon processes. A process is 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 and are fully described.
- **Communication manager**
  - Communication between threads attached to different processes on the same computer. Why because, some kernels also support communication b/w threads in remote processes. Other kernels require additional service for external communication.

- **Memory manager**
  - Management of physical and virtual memory.
- **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.

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

In the common *symmetric processing architectre*, each processor executes the same kernel and the kernels paly 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 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 serval threads that handle several requests from clients simultaneously. For example providing access to a shared database.

5.3. Protection: The resources require protection from illegitimate (illegal) access. For example, the file can opened in two operations, such as *read* and *write*.

Therefore files are protected from being read by users without read permissions, and devices registers are protected from application processes.

The other type of illegitimate access is address of file, called file pointer. **For example,** smith managed to access file pointer variable directly. So she constructs a (*setFilePointerRandomly*) operation, which sets the file pointer to a random number.

Thus, Access the file pointer variable directly (*setFilePointerRandomly*) i.e Non-type-safe language vs. type-safe language.

- Type–safe language, e.g. Java or Modula-3.
- Non-type-safe language, e.g. C or C++

5.3.1. Kernels and protection: The kernel is a program that is distinguished by the facts that it always runs 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 resource except in acceptable ways.

A kernel process executes with the processor in *supervisor* (privileged) mode. The kernel arranges that other processes execute in *user* (unprivileged) mode.

- **Different execution mode**
  - *An address space:* a collection of ranges of virtual memory locations, in each of which a specified combination of memory access rights applies, e.g.: read only or read-write
  - *supervisor* mode (*kernel process*) / *use*r mode (*user process*)

- Interface between kernel and user processes: system call trap
- The kernel also sets up address spaces to protect itself and other processes from the accesses of an aberrant process
- To provide processes with their required virtual memory layout.
- 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.
- 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.

- **The price for protection**
  - switching between different processes take many processor cycles
  - a *system call trap* is a more expensive operation than a simple method call
  - 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.

5.4. Processes and Threads:

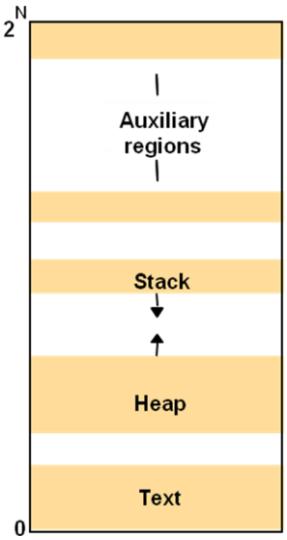
- **Process:** A process is nothing but an execution of program.
  - Problem: sharing between related activities is awkward and expensive.
  - Solution: To enhance the notion of process so that it could be associated with multiple activities.
  - Nowadays, a process consists of an *execution environment* together with one or more *threads*
- **Execution environment**
  - An execution environment is the unit of resource management. It means, a collection of local kernel managed resources to which its threads have access.
  - An execution environment represents the protection domain in which its threads execute.
  - Execution environment consist of
    - An address space
    - Thread synchronization and communication resources such as semaphores and communication interfaces (e.g. sockets)
    - Higher-level resources such as open files and windows
  - Shared by threads within a process
- **Thread:** A *process* is divided into number of smaller tasks; each task is called a "*Thread*". A *thread* is the operating system abstraction of an activity. Threads can be created and destroyed dynamically as needed. The central aim of multiple threads execution is to maximize the degree of concurrent execution b/w operations such as enabling the overlap of

computation with input and output, and enabling concurrent processing on multiprocessors. This can be helpful within servers.

For example one thread can process a client's request while a second thread servicing another request waits for a disk access to complete.

– Benefits

• Responsiveness, Resource sharing, Economy, Utilization of MP architectures



5.4.1. Address spaces

- An address space is a unit of management of *a process*'s virtual memory
- It is large up to 2^{32} bytes and sometimes up to 2^{64} bytes
- It consists of one or more regions
- **Region**
- an area of continuous virtual memory that is accessible by the threads of the owning process
- Each region is specified by the following Properties:
  - Its extent i.e. lowest virtual address and size.
  - Read/write/execute permissions for the process's threads.
  - Whether it can be grown upwards or downwards.
- **The number of regions is indefinite**
  - 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 and exceed the stack limits and control each stack's growth.
  - Another one is to map the address space. Access the map as array of bytes in memory is called *mapped file.*
  - Share memory between processes or b/w processes and the kernel is another factor leading to extra regions in the address space.
- **Region can be shared with the following uses:**
  - **Libraries**: Library code can be shared by being mapped as a region in the address space of processes that required it.
  - **Kernel**: Kernel code and data are mapped into every address space at the same location.
  - **Shared data and communication**: It is more efficient for the data to be shared by being mapped as regions in both address spaces than by being passed in messages b/w them.

5.4.2. Creation of new process in distributed system: The creation of a new process has traditionally been an indivisible operation provided by the operating system. For example, the UNIX *fork* system call creates a process with an execution environment copied from the caller. The UNIX *exec* system call transforms the calling process into one executing the code of a named program.

For a distributed system, the design of the process creation mechanism has to take account of the utilization of multiple computers.

The creation of 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 computer server.
- The creation of an execution environment.

5.4.2.1. ***Choice of*** **Process host:** Process allocation policies range from always -

- running new processes at their originator's computer
- sharing processing load between a set of computers

- Policy categories for load sharing:

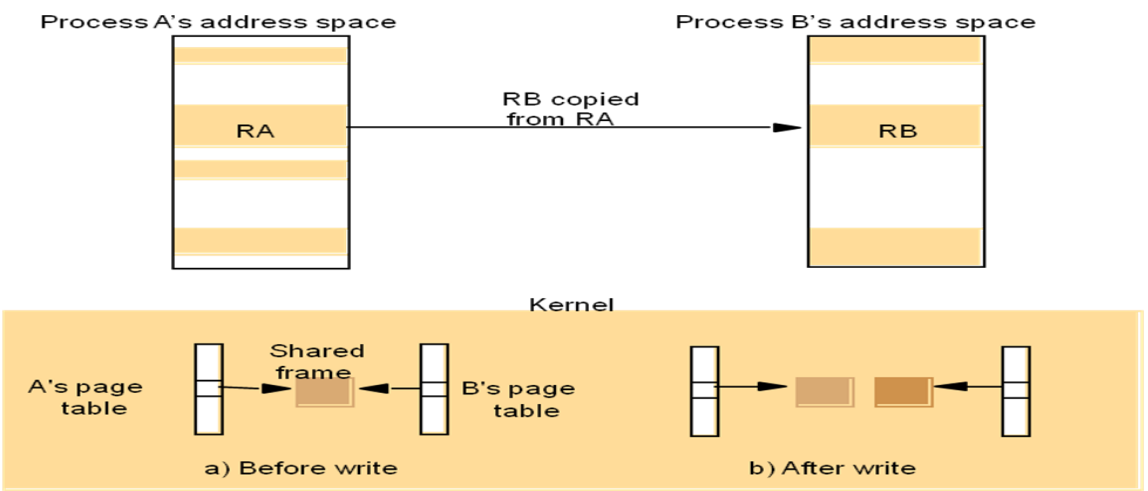
- The ***transfer policy*** determines whether to situate a new process locally or remotely. This may depend on whether the local node is lightly or heavily loaded.
- The ***location policy*** determines which node should host a new process selected for transfer. This decision may depend on the relative loads on their machine architectures and on any specialized resources they may possess.
- The ***migratory*** load-sharing systems can shift load at any time, not just when a new process is created. Use of this mechanism is called *process migration.*

5.4.2.2. **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.

There are two approaches to defining and initializing the address space of a newly created process.

- The first approach is used where the address space is of *statically defined format*. For example, it could contain just a program text region, heap region and stack region.
- The second approach is the address space can be with respect to an existing execution environment, e.g. *fork*

5.4.2.3. **Copy-on-write:** It is a general technique. It is also used in copying large messages etc. For example, regions RA and RB's memory is shared copy-on-write b/w two processes, A and B. This is shown in fig. Let us assume that process A set region RA to be copy-inherited by its child, process B, and that the region RB was created in process B.



5.4.3. Threads: A process is divided into number of smaller tasks, each task is called a "*Thread*". Number of threads with in a process execute at a time is called "multithreading". Supporting of multithreading is the additional capability of an operating system. Based on the functionality, threads are divided into 4 categories:

1. Responsiveness: It increases responsiveness to the user when a multithreaded web browser are using.

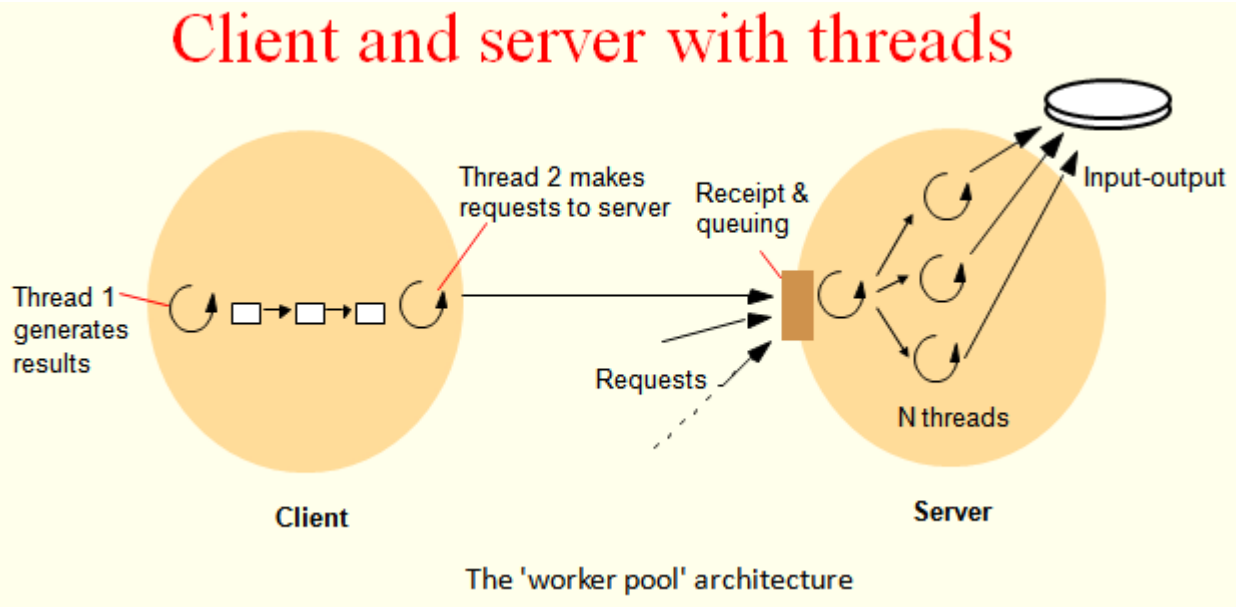
2. Resource sharing: Threads share the memory and the resources of the process to execute the application. In this case, the address space of shared memory allows several different threads of activity within the same address space.

3. Economy: Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong. For example, creating a process is about thirty times slower than is creating a thread and context switching is about five time slower.

4. Utilization of multiprocessor architectures: The benefits of multithreading can be greatly increased in a multiprocessor architecture. A single threaded process can only run on one CPU, but not more. Multithreading on a multi-CPU machine increases concurrency.

5.4.3.1. Architectures for multi-threaded servers:

Consider the server shown in fig that 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 assume that each thread applies the same procedure to process the requests. Let us assume that each request takes 2 milliseconds of processing and 8 milliseconds of I/O (input/output) delay when the server reads from a disk. Let us further assume for the moment that the server executes at a single-processor computer.

Consider the maximum server throughput is measured in client requests handled per second for different numbers of threads. Therefore 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.

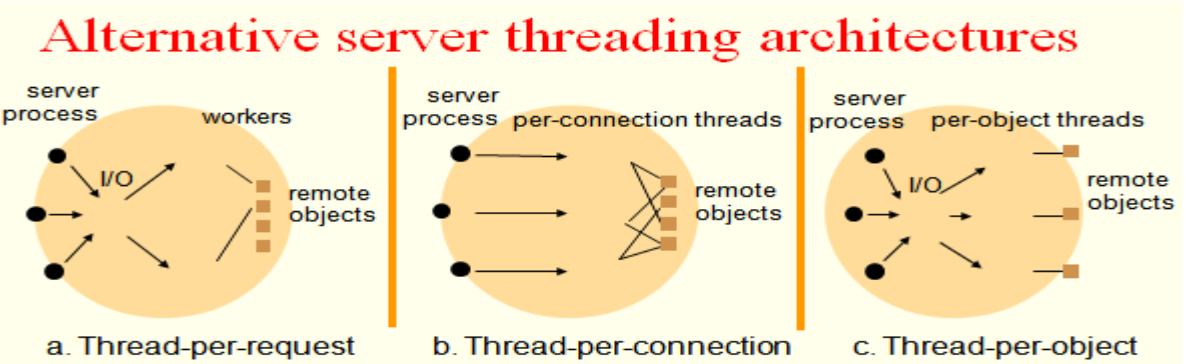
Now consider if the server pool contains two threads. Let us assume that one thread can be scheduled when another becomes blocked for I/O. This increases the server throughput. For

example, if all disk requests are serialized and take 8 milliseconds each, then the maximum throughput is 1000/8 = 125 requests per second.

The figure shows one of the possible threading architectures and that described how multi-threading enables servers to maximize their throughput and measured as the number of requests processed per second.

The figure shows one of the possible threading architecture is the *worker pool architecture.* In it's, 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 is 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.

5.4.3.2. Alternative server threading architectures:



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

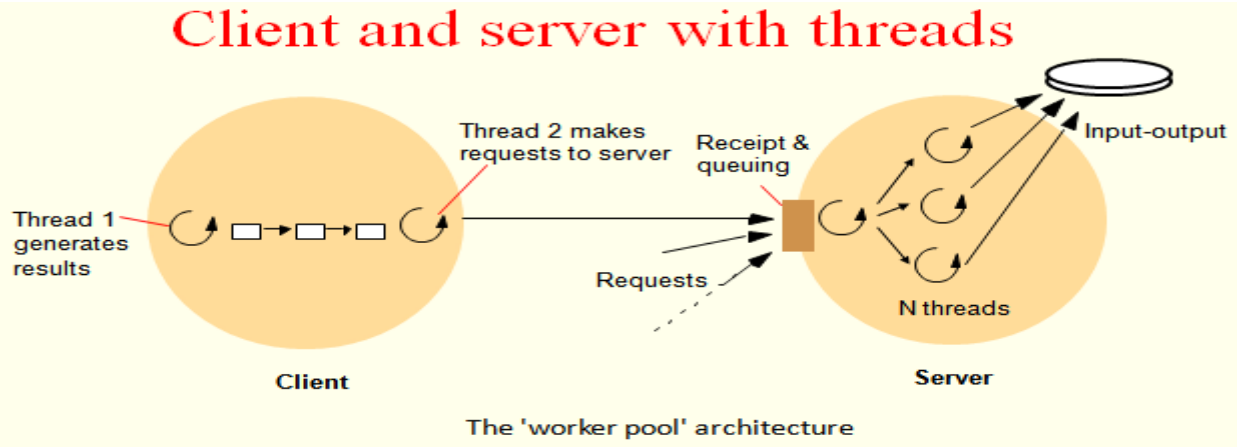
The ***thread-per-connection architecture*** associates a thread with each connection. The server creates a new worker thread when a client makes a connection and destroys the thread when the client closes the connection. In b/w, the client may make many requests over the connection, targeted at one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

The ***thread-per-object architecture*** 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 lowered thread-management overheads compared with the thread-per-request architecture. The disadvantage is that clients may be delayed while a worker thread has several outstanding requests but another thread has no work to perform.

5.4.3.3. Threads within clients: Threads can be useful for clients as well as servers. The figure 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 and able to continue computing further results.
- The second thread incorporates the client process and performs the remote method invocation and blocks.
- 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.



5.4.3.4. Threads versus multiple processes: Multi-threaded clients are evident in the examples of web browsers.

- Creating a thread is (much) cheaper than a process (~10-20 times)
- Switching to a different thread in same process is (much) cheaper (5-50 times)
- Threads within same process can share data and other resources more conveniently and efficiently (without copying or messages)
- Threads within a process are not protected from each other

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	

5.4.3.5. Threads Programming: Threads programming is concurrent programming. Much threads programming is done in a conventional language such as C with thread for library. The C threads package developed for the Mach operating system, the POSIX thread of IEEE 1003, pthreads, etc.

Thread(ThreadGroup group, runnable target, string name)

run()

start()

5.4.3.6. Thread lifetimes: A new thread is created on the same Java virtual machine (JVM) as its creator, in the SUSPENDED state. After it is made RUNNABLE with the ***start()*** method, it executes the ***run()*** method of an object designated in its constructor. The JVM and the threads on top of it all execute in a process on top of the underlying operating system. A thread ends its life when it returns from he **run()** method, or when its ***destroy()*** method is called.

Programs can manage threads in groups. Every thread belongs to one group, which it assigned at the time of its creation. Thread groups are useful when several applications coexist on the same JVM.

Thread groups also facilitate control of the relative priorities of threads. This is useful for browsers running applets, and for web servers running programs called *servlets*, which create dynamic web pages.

Java Thread Constructors 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 identifier as name the thread will execute the run() method of target.

setPriority (int newPriorty), getPriority()- *Sets and returns the thread priority.*

run() *- A thread executes the run() method of its target object if it has one and otherwise its own* ***run()-*** *method i.e; thread implements runnable.*

start()- *Changes the state of the thread from SUSPENDED to RUNNABLE.*

sleep(int milisecs)*-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.*

5.4.3.7. Thread synchronization: Programming a multi-threaded process requires great care. The main difficult issues are sharing of objects and the techniques used for thread coordination and cooperation.

Java provides the *synchronized* keyword for programmers to designate the well-known monitor construct for thread coordination. Programmers designate either entire methods or arbitrary blocks of code as belonging to a monitor associated with an individual object. The monitor's guarantee is that at most one thread can execute within it at any time.

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.

5.4.3.8. Thread scheduling: The thread scheduling are

1. *Preemptive scheduling:* In this, a thread may be suspended at any point to make way for another thread, even when the preempted thread would otherwise continue running.

2. *non-preemptive scheduling*: In this a thread runs until it makes a call to the threading system, when the system may de-schedule it and schedule another thread to run.

The advantage of non-preemptive scheduling is that any section of code that does not contain a call to the threading system is automatically a critical section. But it cannot allow in multiprocessor.

5.4.3.9. Threads Implementation: Threads can be implemented in many kernels to support multi-threaded processes. For example, windows, Linux, Solaris, Mach and Chorus. These kernels provide thread creation and management system calls, and they schedule individual threads. The multi-threaded processes must be implemented in a library of procedures linked to application programs. In this case, user-level threads cannot schedule them independently.

Scheduler activations:

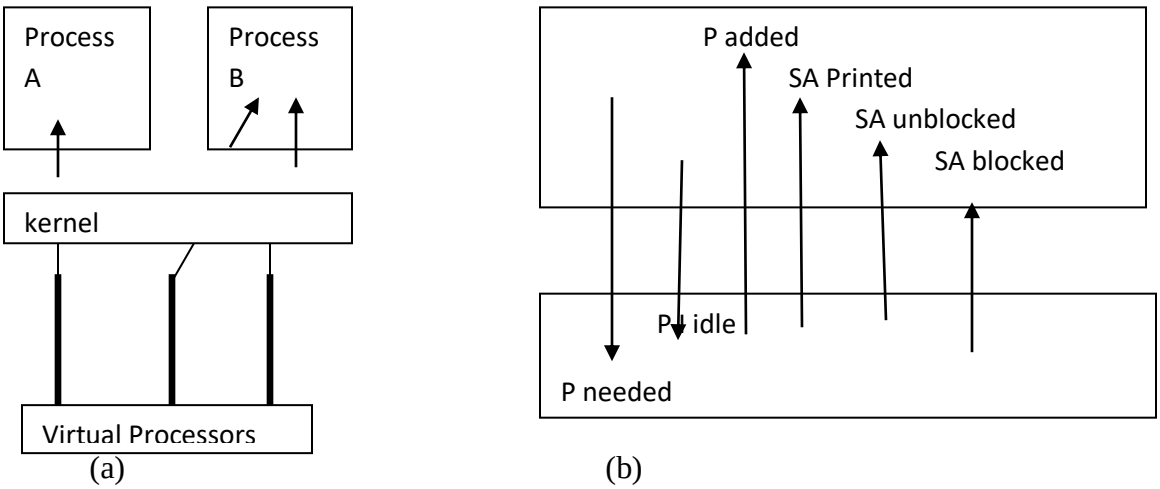


Fig (a): Assignment of Virtual processors to processes

Fig (b): Events between user level scheduler and kernel key: P=Processor; SA= Scheduler Activation

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

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

The four types of event that kernel notifies user level scheduler are as follows:

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 SA with the context of READY thread, which can thus recommend execution.

SA blocked : An SA has blocked in the kernel, and the kernel is using a fresh SA to notify the scheduler, scheduler sets the state if the corresponding thread to BLOCKED and can allocate a READY thread to 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 , the scheduler place the preempted thread in the READY list and revaluates the thread allocations.

5.5 Communication and invocation: Communication is the part of the implementation of an invocation. It means, a construct such as a remote method invocation, remote procedure call or event notification, whose purpose is to bring an operation on a resource in a different address space.

5.5.1. Communication:

Communication primitives: Some kernels designed for distributed systems have provided communication primitives to the all types of invocation. For example,

- TCP(UD
- P) Socket in Unix and Windows
- *DoOperation, getRequest, sendReply* in Amoeba
- Group communication primitives in V system

For example, if middleware provides RMI over UNIX's connected (TCP) sockets, then a client must make two communication system calls for each remote invocation. The principal reasons for using sockets are portability and interoperability.

Protocols and openness: One of the main requirements of the operating system is to

- Provide standard protocols that enable internetworking between middleware implementations on different platforms.
- integrate novel low-level protocols without upgrading their application
- Protocols are normally arranged in a stack of layers.
    - Static Stack: Many operating systems allow new layers to be integrated statically and permanently installed protocol as a "driver"
    - Dynamic stack
        - protocol stack be composed on the fly
        - E.g. web browser utilize wide-area wireless link on the road and faster Ethernet connection in the office

5.5.2. Invocation performance: Invocation performance is a critical factor in distributed system design. The more designers separate functionality b/w address spaces, the more remote invocations are required. Clients and servers may make many millions of invocation-related operations in their lifetimes, so that small fractions of milliseconds count in invocation costs.

- **Invocation costs:** Calling a conventional procedure or method, making a system call, sending a message, remote procedure calling and remote method invocations are all examples of invocation mechanisms. Each mechanism code to be executed out of scope of the calling procedure or object. This code can be helped to communicate the arguments and return the data values to the caller. This mechanism can be either synchronous or can be asynchronous.
- **Invocation over the network:** A *null* RPC 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.
    - Delay: the total RPC call time experienced by a client
    - Latency: the fixed overhead of an RPC is measured by null RPC
    - Throughput: the rate of data transfer between computers in a single RPC
    - An example
        - Threshold: one extra packet to be sent, might be an extra acknowledge packet is needed

The following are the main components accounting for remote invocation delay.

- **Marshalling:** It involves on copying and converting data.

- **Data copying:** Potentially message data is copied several times in the course of an RPC

The performance of RPC and RMI mechanisms is critical for effective distributed systems.

- Typical times for 'null procedure call':
  - Local procedure call< 1 microseconds
  - Remote procedure call ~ 10 milliseconds
- 'Network time' (involving about 100 bytes transferred, at 100 megabits/sec.) accounts for only .01 millisecond; the remaining delays must be in OS and middleware - latency, not communication time.

- **Factors affecting RPC/RMI performance**
  - marshalling/unmarshalling + operation despatch at the server
  - data copying:- application -> kernel space -> communication buffers
  - thread scheduling and context switching:- including kernel entry
  - protocol processing:- for each protocol layer
  - network access delays:- connection setup, network latency

Improve the performance of RPC

-Memory sharing: Shared regions may be used for rapid communication between processes and the kernel or b/w user processes. Data is communicated by *writing to* and *reading from* the shared region.

-Choice of protocol: If delay occurs during the request-reply interaction, then increase the performance of RPC for TCP.

- TCP/UDP E.g. Persistent connections: several invocations during one
- OS's buffer collect several small messages and send them together

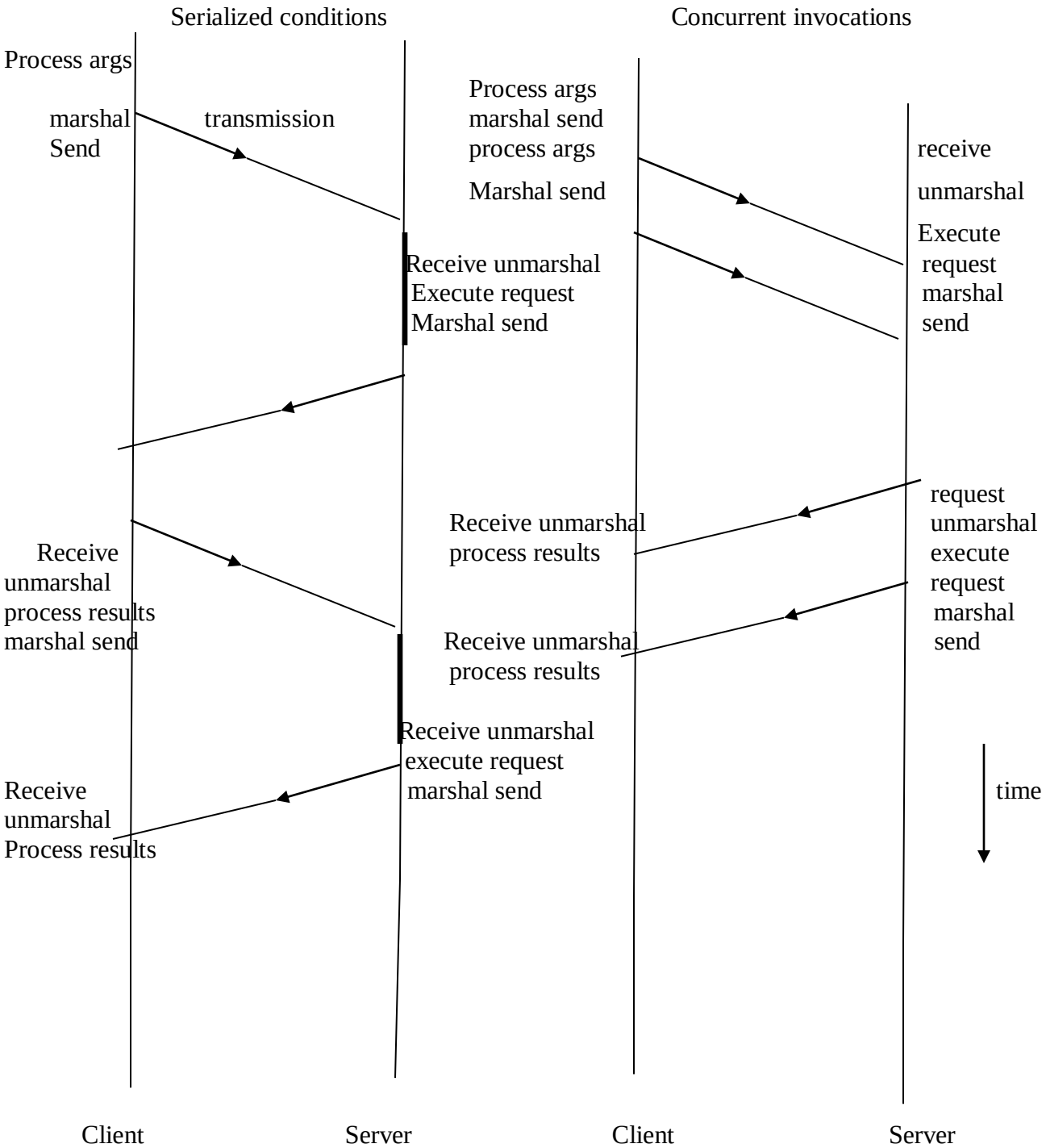
-**Invocation within a computer:** Most cross-address-space invocation takes place within a computer at time of installation of OS. The cost of RPC within a computer is growing in importance as a system performance parameter. A cross-address-space invocation is implemented within a computer exactly as it b/w computers, except that the underlying message passing happens to be local.

5.5.3. Asynchronous operation: A common technique to high latencies is asynchronous operations, which arises in two programming models. 1. Concurrent invocations and 2. Asynchronous invocations. 3.Persistent asynchronous invocations.

- ***Concurrent invocations:*** In this model the middleware provides only blocking invocations, but the application spawns multiple threads to perform blocking invocations concurrently.

E.g., A good example this is a web browser. A web page contains several images. The browser has to fetch each of these images in a separate HTTP GET.

Times for serialized and concurrent invocation



- ***Asynchronous invocation***: An *asynchronous invocation* is performed asynchronously with respect to the caller. That is, it is made with a non-blocking call, which returns as soon as the invocation request message has been created and is ready for dispatch.

- E.g., CORBA *one-way* invocations have maybe semantics, otherwise, the client uses a separate call to collect the results of the invocation. For example, the Mercury communication system and supports asynchronous invocations. In this case, an asynchronous operation returns an object called a *promise*. The caller uses the claim operation to obtain the result from the promise.

- ***Persistent asynchronous invocations:***

A system for persistent asynchronous invocation tries indefinitely to perform the invocation, until it is known to have succeeded or failed. For example, QRPC (Queued RPC) is used keep the requests of all clients in queue which applied to access the server. In this case, persistent asynchronous invocation tries to establish the connection on server if TCP stream is in difficult.