Unit-5

Operating system support

→Introduction:

Now we will discuss how middleware is supported by the operating system facilities at the nodes of a distributed system. The operating system facilitates the encapsulation and protection of resources inside servers and it supports the mechanisms required to access these resources, including communication and scheduling.

An important aspect of distributed systems is resource sharing. Client applications invoke operations on resources that are often on another node or at least in another process. Applications (in the form of clients) and services (in the form of resource managers) use the middleware layer for their interactions. Middleware enables remote communication between objects or processes at the nodes of a distributed system.

Below the middleware layer is the operating system (OS) layer. The task of any operating system is to provide problem-oriented abstractions of the underlying physical resources – the processors, memory, networks, and storage media.
There are two operating system concepts: Network operating systems and Distributed operating systems.

Both UNIX and Windows are examples of ***network operating systems.*** They have a networking capability built into them and so can be used to access remote resources.Access is network transparent to some types of resources. There are multiple system images, one per node. With a network operating system, a user can remotely log into another computer, and run processes there.

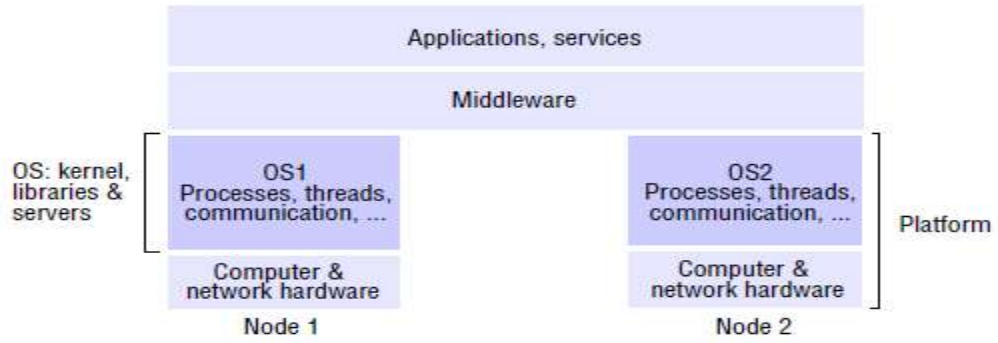
By contrast, the operating system has control over all the nodes in the system, and it transparently locates new processes at whatever node suits its scheduling policies. An operating system that produces a single system image like this for all the resources in a distributed system is called a ***distributed operating system.***

Middleware and network operating systems: In fact, there are no distributed operating systems in general use, only network operating systems such as UNIX, Mac OS and Windows. The combination of middleware and network operating systems provides an acceptable balance between the requirements for autonomy on the one hand and network transparent resource access on the other. The network operating system enables users to run their word processors and other standalone applications. Middleware enables them to take advantage of services that become available in their distributed system.

→The operating system layer:

Users will only be satisfied if their middleware–OS combination has good performance. Middleware runs on a variety of OS–hardware combinations (platforms) at the nodes of a

distributed system. The OS running at a node – a kernel and associated user-level services such as communication libraries – provides its own flavour of abstractions of local hardware resources for processing, storage and communication.



System layers

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. Kernels and the client and server processes that execute upon them are the chief architectural components of distributed system. Kernels and server processes are the components that manage resources and present clients with an interface to the resources.

We require at least the following of them:

- ***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.

Clients access resources by making remote invocarion methods to a server or system calls to a kerner. The means of accessing a resource can be called as an invocation mechanism. A combination of libraries, kernels and servers may be called upon to perform the following invocation related tasks:

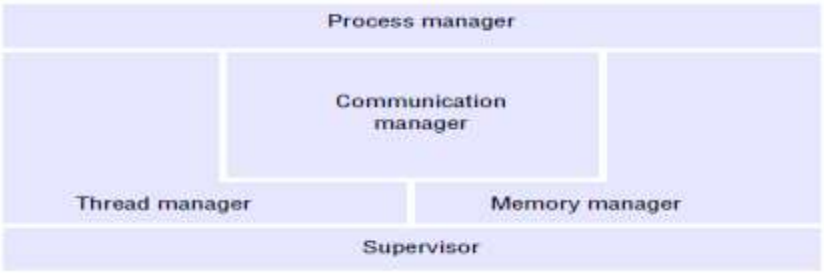
- ***Communication*:** Operation parameters and results have to be passed to and from resource managers, over a network or within a computer.

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

Below 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 core OS components and their responsibilities are:

- ***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.



Core OS functionality

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

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

- ***Memory manager*:** Management of physical and virtual memory. It discuss the utilization of memory management techniques for efficient data copying and sharing.

- ***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.

Some kernels can execute on shared memory multiprocessors which are described below:
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.

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.

→Protection:

The resources under the control of operating system require protection from illegitimate accesses. As an example for 'illegitimate access' to a resource, consider a file. Open files have only two operations, ***read* and *write*.**

Protecting the file consists of two sub-problems.

1) 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 requires **cryptographic techniques.**

2) 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. We can protect resources from illegitimate invocations**.** One way is to use a **type-safe programming language**, such as **Sing#,** an extension of **C# or Modula-3**. 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++.

Kernels and protection:

The kernel is a program that remains loaded from system initialization and its code is executed with complete access privileges for the physical resources on its host computer. Most processors have a hardware mode register whose setting determines whether privileged instructions can be executed. 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 abnormal process. An address space is a collection of ranges of virtual memory locations. 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

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.*** 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.

→Processes and threads:-

The traditional operating system notion of a process that executes a single activity was found in the 1980s to be unequal to the requirements of distributed systems. The traditional process makes sharing between related activities uncomfortable and expensive.

The solution is the notion of the process 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. 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** consists of:

- 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.

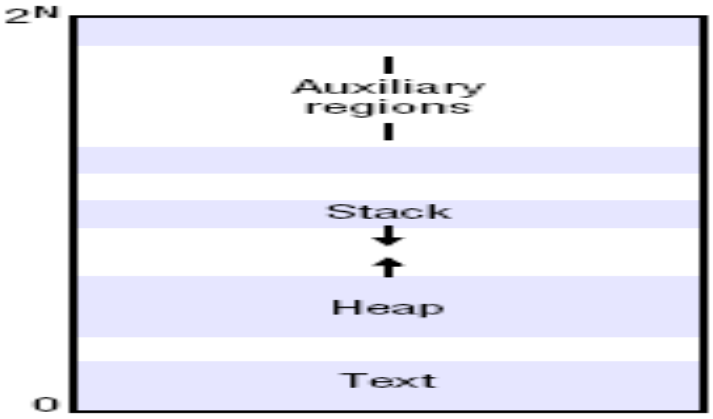
Threads can be created and destroyed dynamically, as needed. The central aim of having multiple threads of execution is to maximize the degree of concurrent execution between operations, thus enabling the overlap of computation with input and output, and enabling concurrent processing on multiprocessors. This can be particularly helpful within servers, where concurrent processing of clients' requests can reduce the tendency for servers to become bottlenecks. For example, one thread can process a client's request while a second thread servicing another request waits for a disk access to complete.

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.

As many older operating systems allow only one thread per process. Sometimes use the term ***multi-threaded*** *process* for emphasis. In operating system the process means heavy weight process and thread means light weight process.

→Address spaces:

An address space is a unit of management of a process's virtual memory. It is large (typically up to 2^{32} bytes and sometimes up to 2^{64} bytes) and consists of one or more *regions*. A region is an area of contiguous virtual memory that is accessible by the threads of the owning process. Regions do not overlap.



Address space

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.

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

- a fixed, un modifiable 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.

The need to support a separate stack for each thread is 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.

Another motivation is to enable files in general to be mapped into the address space. A ***mapped file*** is one that is accessed as an array of bytes in memory.

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*** 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.

→Creation of new process:

The creation of a new process is 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.

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

The creation of a new process can be separated into two independent aspects:

1) the choice of a target host,
2) The creation of an execution environment.

1) Choice of target host: In general, process allocation policies range from 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:

→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.

Process location policies may be ***static*** **or** ***adaptive*****.** The static policies operate without regard to the current state of the system. They are based on a mathematical analysis aimed at optimizing a parameter such as the overall process throughput. They may be deterministic or probabilistic.

Adaptive policies, on the other hand, apply heuristics to make their allocation decisions, based on unpredictable runtime factors such as a measure of the load on each node.

Load-sharing systems may be **centralized, hierarchical or decentralized**. In the first case there is one *load manager* component, and in the second there are several, organized in a tree structure.

Load managers collect information about the nodes and use it to allocate new processes to nodes. In hierarchical systems, managers make process allocation decisions as far down the tree as possible, but managers may transfer processes to one another, via a common ancestor, under certain load conditions.

In ***sender-initiated*** load-sharing algorithms, the node that requires a new process to be created is responsible for initiating the transfer decision. It typically initiates a transfer when its own load crosses a threshold.

By contrast, in ***receiver-initiated*** algorithms, a node whose load is below a given threshold advertises its existence to other nodes so that relatively loaded nodes can transfer work to it.

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***: the transfer of an executing process from one node to another.. While several process migration mechanisms have been constructed, they have not been widely deployed. This is largely because of their expense and the tremendous

difficulty of extracting the state of a process that lies within the kernel, in order to move it to another node.

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.

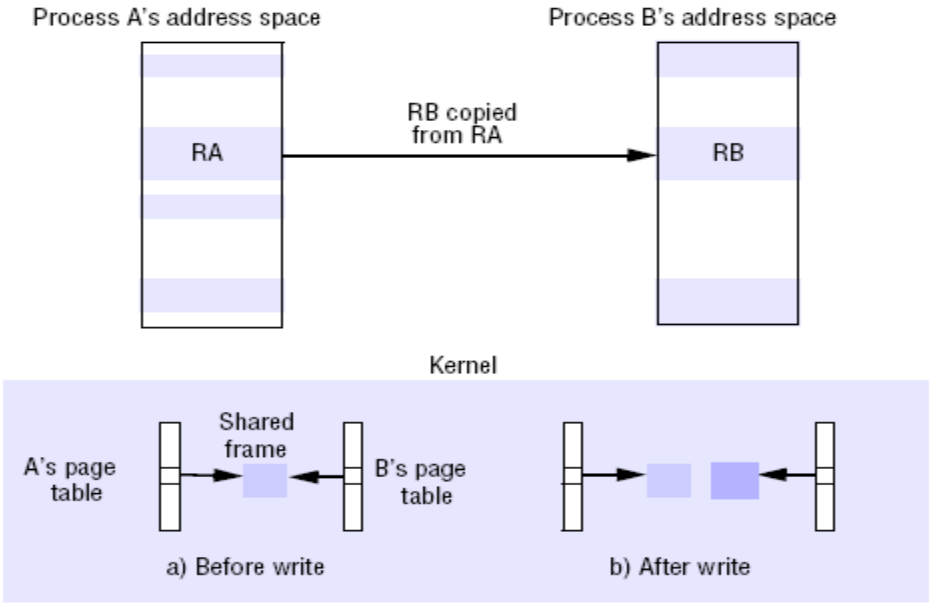
There are two approaches to defining and initializing the address space of a newly created process.

1) The address space is of a statically defined format. For example, it could contain just a program text region, heap region and stack region.
2) The address space can 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). This scheme has been generalized so that each region of the parent process may be inherited by (or omitted from) the child process. An inherited region may either be shared with or logically copied from the parent's region.

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

Copy-on-write is a general technique used in copying large messages. Let us follow through an example of regions *RA* and *RB*, whose memory is shared copy-on-write between two processes, *A* and *B* (Figure below). Let us assume that process *A* set region *RA* to be copy-inherited by its child, process *B*, and that the region *RB* was thus created in process *B*.



Copy-on-write

We assume that the pages belonging to region *A* are resident in memory. Initially, all page frames associated with the regions are shared between the two processes' page tables. If a thread in either process attempts to modify the data, a hardware exception called a ***page fault*** is taken. Let us say that process *B* attempted the write. 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 – it does not matter which – and the old frame number is left in the other page table. The two corresponding pages in processes *A* and *B* are then each made writable once more at the hardware level. After all of this has taken place, process *B*'s modifying instruction is allowed to proceed.

→Threads:-

Here we examine the advantages of enabling client and server processes to possess more than one thread. Consider a server that has a pool of one or more threads, each of which repeatedly removes a request from a queue of received requests and processes it. For the sake of simplicity, we assume that each thread applies the same procedure to process the requests. 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) in a single processor computer.

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. This increases the server throughput. If all disk requests are serialized and take 8 milliseconds each, then the maximum throughput is **1000/8 = 125 requests per second.**

Suppose that disk block caching is introduced. If a 75% hit rate is achieved, the mean I/O time per request reduces to (0.75*0 + 0.25*8) = 2 milliseconds, and the maximum 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**.

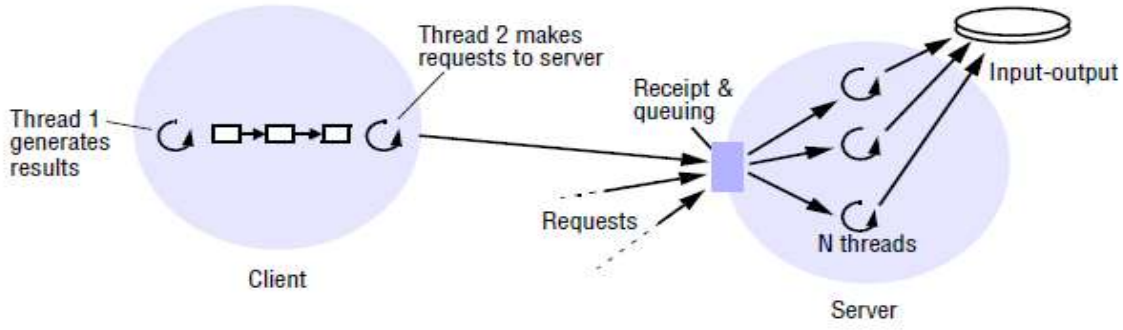
The throughput can be increased by using a shared-memory multiprocessor to ease the processor bottleneck. Multiple threads can be scheduled to run on the multiple processors.

Architectures for multi-threaded servers:-

The threading architectures are:

- Worker pool architecture
- Thread per request architecture
- Thread per connection architecture and
- Thread per object architecture

Following figure shows the ***worker pool architecture*.** The server creates a fixed pool of 'worker' threads to process the requests when it starts up. The module marked 'receipt and queuing' in the figure below is implemented by I/O thread and receives requests from a collection of sockets and places them on a shared queue. These requests are retrieved from the queue by worker threads.



Worker pool

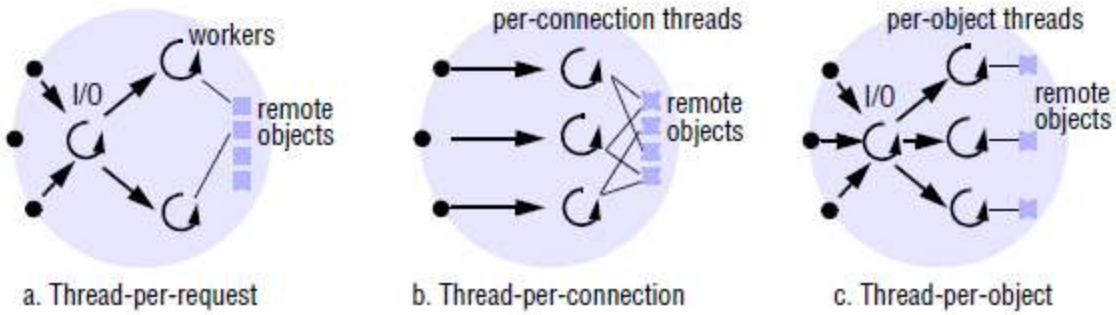
A disadvantage of this architecture is its inflexibility: the number of worker threads in the pool may be too few to deal adequately with the current rate of request arrival.

In the ***thread-per-request architecture*** *(fig a)* the I/O thread creates 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*** *(fig b)* 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*** *(fig c)* 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.



server threading architectures

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. The worker pool architecture figure shows a client process with two threads. The first thread generates results to be passed to a server by remote method invocation. Remote method invocations typically block the caller, even when there is 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.

Threads versus multiple processes: 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.

The following table shows some of 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 the threads 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.

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

• Creating a new thread within an existing process is cheaper than creating a process.

• 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, threads within a process are not protected from one another.

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	

State associated with execution environments and threads

The overhead associated with ***creating a process*** are greater than those of ***creating a new thread***. It is assumed that it takes 11milliseconds to create a process and about 1millisecond to create a thread.

Switching between threads sharing the same execution environment is considerably cheaper than switching between threads belonging to different processes.The overheads associated with thread switching are related to ***scheduling*** and ***context switching***.

Scheduling is the process of choosing the next thread to run.

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

A ***context switch*** is the transition between contexts that takes place when switching between threads. 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*.

It is assumed that it takes 1.8milliseconds to switch between processes and 0.4milliseconds to switch between threads.

The aliasing problem • 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.

In the example above of the client process with two threads, the first thread generates data and passes it to the second thread, which makes a remote method invocation or remote procedure call. Since the threads **share** an address space, there is no need to use message passing to pass the data. Both threads may access the data via a common variable.

Herein lies both the advantage and the disadvantage of using multi-threaded processes. The advantage is the convenience and efficiency of access to shared data. The disadvantage is a thread can alter data used by another thread.

If protection is required, then either a type safe language should be used or it may be preferable to use multiple processes instead of multiple threads.

Threads programming: Threads programming is concurrent programming studied in the field of operating systems. Threads programming is done with a threads library. The C Threads package is an example of this. More recently, the **POSIX** Threads standard, known as ***pthreads,*** has been widely adopted. Some languages provide direct support for threads, including Ada95, Modula-3 and Java.

The Java *Thread* class includes the constructor and management methods listed in below.

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.

Java thread constructor and management methods

Thread lifetimes :

A new thread is created on the same Java virtual machine (JVM) 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. Threads can be assigned a priority, so that a Java implementation that supports priorities will run a particular thread in preference to any thread with lower priority. A thread ends its life when it returns from the *run()* method or when its ***destroy()*** method is called.

Programs can manage threads in groups. Every thread belongs to one group, which it is assigned at the time of its creation applications. Thread groups are useful. One example of their use is security: by default, a thread in one group cannot perform management operations on a thread in another group.

Thread synchronization : Programming a multi-threaded process requires great care. The main difficult issues are the 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. *Addto*() and *removeFrom*() methods are designated as synchronized methods in *Queue* class in our example.

Java allows threads to be blocked and woken up via arbitrary objects that act as condition variables. A thread that needs to block awaiting a certain condition calls an object's ***wait()*** method. Another thread calls ***notify()*** to unblock at most one thread or ***notifyAll()*** to unblock all threads waiting on that object. These methods belong to the *Object* class.

As an example, when a worker thread discovers that there are no requests to process, it calls *wait()* on the instance of *Queue*. When the I/O thread adds a request to the queue, it calls the queue's *notify()* method to wake up a worker.

The java synchronization methods are given as follows:

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*.

Java thread synchronization methods

The ***join()*** method blocks the caller until the target thread's termination. The ***interrupt()*** method is useful for waking a waiting thread. All the standard synchronization primitives, such as semaphores, can be implemented in Java. But care is required, since Java's monitor guarantees apply only to an object's *synchronized* code; a class may have a mixture of ***synchronized*** **and non-*synchronized*** methods.

Thread scheduling :

Scheduling is the activity of deciding which thread has to be given access to resources. An important distinction is between preemptive and non preemptive scheduling of threads.

In *preemptive scheduling*, a thread may be suspended at any point to make way for another thread.

In ***non-preemptive scheduling***, a thread runs until it makes a call to the threading system when the system may reschedule 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. On the other hand, non-preemptively scheduled threads cannot take advantage of a multiprocessor. Nonpreemptively scheduled threads are also unsuited to real-time applications.

Threads implementation: Many kernels provide support for multi-threaded processes, such as Windows, Linux, Solaris, Mach and Mac OS X. These kernels provide thread-creation and -management system calls, and they schedule individual threads.

Some other kernels have only a single-threaded process abstraction. Multithreaded processes must then be implemented in a library of procedures linked to application programs.

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.

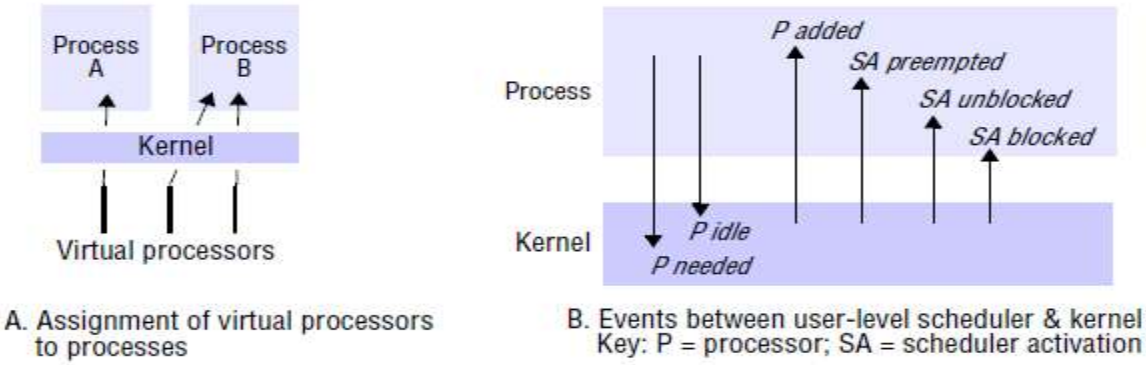
User-level threads implementations, on the other hand, have significant advantages over kernel-level implementations:

- Certain thread operations are significantly less costly. For example, switching between threads belonging to the same process does not necessarily involve a system call.
- 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.

It is possible to combine the advantages of user-level and kernel-level threads implementations. A user-level scheduler assigns each user-level thread to a kernel-level thread. This scheme can take advantage of multiprocessors, and also benefits because some thread-creation and thread-switching operations take place at user level. The scheme's disadvantage is that it lacks flexibility: if a thread blocks in the kernel, then all user-level threads assigned to it are also prevented from running, regardless of whether they are eligible to run.

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. The number of virtual processors assigned to a process depends on such factors as the applications' requirements, their relative priorities and the total demand on the processors.

The following figure 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.



A. Assignment of virtual processors to processes

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

Scheduler activations

Figure (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. An SA is a unit of allocation of a time slice 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 four types of event that the kernel notifies the user-level scheduler of are as follows:

Virtual processor allocated**:** The kernel has assigned a new virtual processor to the process, and this is the first time slice 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. In order to create the notifying SA, the kernel either allocates a new virtual processor to the process or preempts another SA in the same process. In the latter case, it also communicates the preemption event to the scheduler, which can reevaluate its allocation of threads to SAs.

SA preempted**:** The kernel has taken away the specified SA from the process; the scheduler places the preempted thread in the *READY* list and reevaluates the thread allocation.