Unit-4 Operating System Support

Syllabus: Introduction, the Operating System Layer, Protection, Process and Threads; Address Space, Creation of a New Process, Threads

Topic 01: 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.
  - 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.
- The combination of middleware and network operating systems provides an acceptance balance between the requirement of autonomy and network transparency.
- The network operating systems allows users to run their favorite word processor and other standalone applications.
- Middleware enables users to take advantage of services that become available in their distributed system.

Topic 02: Operating System Layer

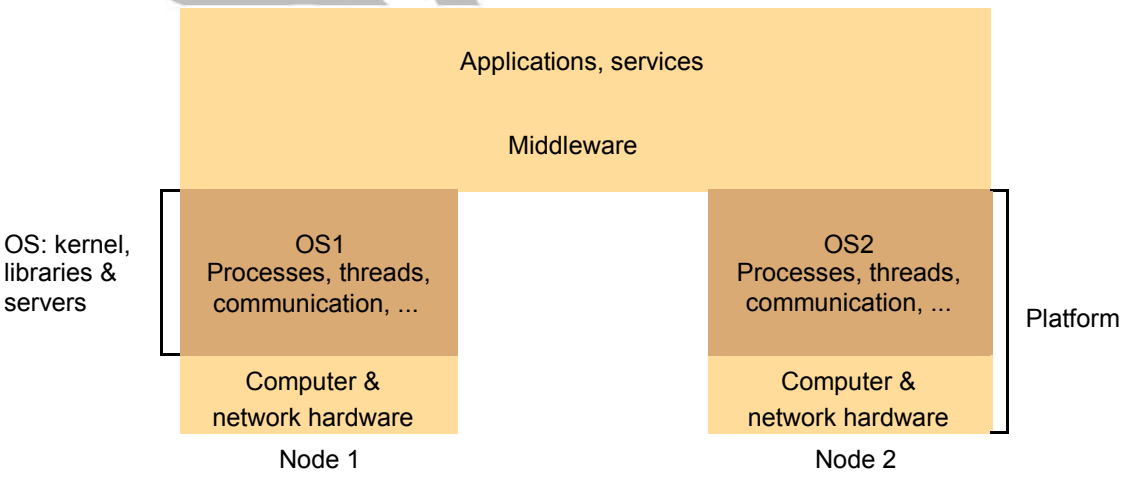


Figure 1. System layers

- 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.
- Kernels and server processes are the components that manage resources and present clients with an interface to the resources.
- The OS facilitates:
  - Encapsulation
  - Protection
  - Concurrent processing
- Invocation mechanism is a means of accessing an encapsulated resource.

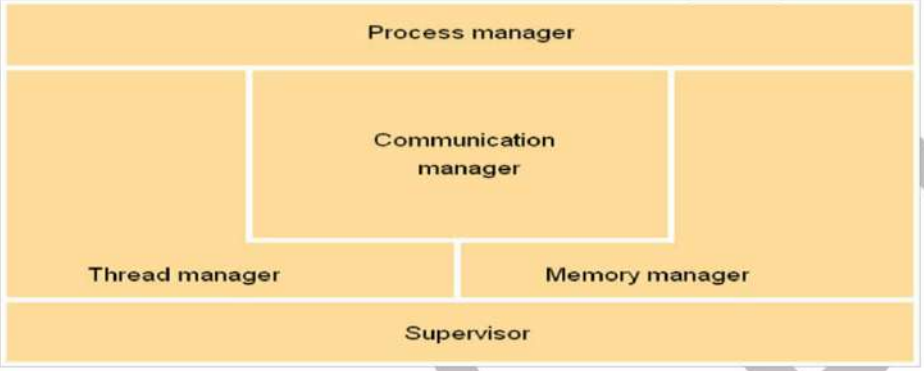


Figure 2. Core OS functionality

- Figure 2 shows the core OS components:
  - Process manager
  - Thread manager
  - Memory manager
  - Communication manager
  - Supervisor

Topic 03: Protection

- Protection means protecting resources (files) from illegitimate

accesses. Kernels and protection

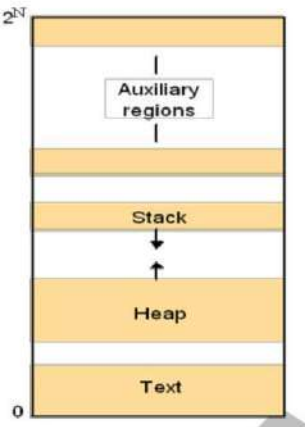
- The kernel is a program that is distinguished by the facts that it always runs and its code is executed with complete access privileges for the physical resources on its host computer.
- It can control the memory management unit and set the processor registers so that no other code may access the machine's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

- 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
- 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 read only 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.
- 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.

Topic 04: Process and Threads

- Process
  - 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 (e.g., semaphores, sockets)
    - Higher-level resources (e.g., file systems, windows)
- 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**, separated by inaccessible areas of virtual memory.
- A region is an area of contiguous virtual memory that is accessible by the threads of the owning process.
- Regions do not overlap.
- The aim of having multiple threads of execution is :
  - 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

Motivation for Indefinite Number of Regions:

- Need to support a separate stack for each thread.
- To enable files
  - *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
  - A *shared memory* region(shared region) is one that is backed by the same physical memory as one or more regions belonging to other address spaces.
- The uses of shared regions include the following:
    - *Libraries*
    - *Kernel*
    - *Data sharing and communication*

Topic 05: 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.

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.
- The *transfer policy* determines whether to situate a new process locally or remotely(load).
- The *location policy* determines which node should host a new process selected for transfer.
- *Process location policies* may be static or adaptive.
- The *static policies* operate without regard to the current state of the system, although they are designed according to the system's expected long-term characteristics.
- 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, 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
- 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 a ***decentralized load-sharing system***, nodes exchange information with one another directly to make allocation decisions.

- The **Spawn system for** example, considers nodes to be 'buyers' and 'sellers' of computational resources and arranges them in a (decentralized) 'market economy'.
- 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.

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 a statically defined format. For example, it could contain just a program text region, heap region and stack region.
- Address space regions are initialized from an executable file or filled with zeros as appropriate
- Alternatively, 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. 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

- **Copy-on-write** is a general technique – for example, it is also used in copying large messages – so we take some time to explain its operation here.
- Let us follow through an example of regions RA and RB, whose memory is shared copy-on-write between two processes, A and B .

- For the sake of definiteness, 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.
- We assume, for the sake of simplicity, 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.

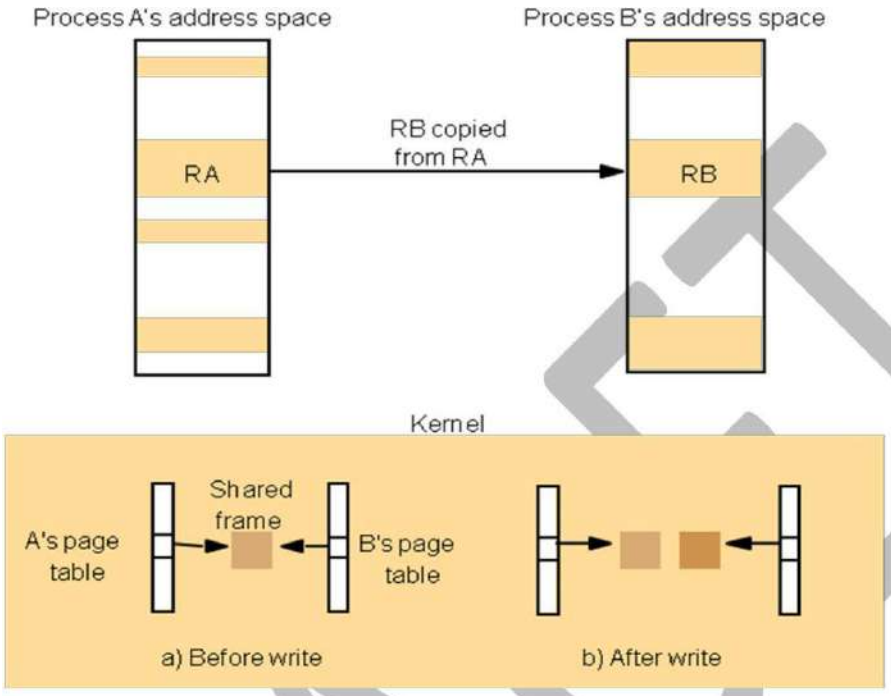


Fig: Copy on Write

- 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.
- 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:

- Threads are schedulable activities attached to processes.

- The aim of having multiple threads of execution is :
  - To maximize degree of concurrent execution between operations
  - To enable the overlap of computation with input and output
  - To enable concurrent processing on multiprocessors.
- Threads can be helpful within servers:
  - Concurrent processing of client's requests can reduce the tendency for servers to become bottleneck.
    - E.g. one thread can process a client's request while a second thread serving another request waits for a disk access to complete.

Processes vs. Threads

- Threads are "lightweight" processes,
- Processes are expensive to create but threads are easier to create and destroy.

Threads programming

- Thread programming is concurrent programming.
- Java provides methods for creating, destroying and synchronizing threads.
- The Java thread class includes the constructor and management methods listed in Figure 3.
- The thread and object synchronization methods are in Figure 4.

Thread(ThreadGroup group, Runnable target, String name)
Creates a new thread in the *SUSPENDED* state, which will belong to *group* and be identified as *name*; the thread will execute the *run()* method of *target*.
setPriority(int newPriority), getPriority()
Set and return the thread's priority.
run()
A thread executes the *run()* method of its target object, if it has one, and otherwise its own *run()* method (*Thread* implements *Runnable*).
start()
Change the state of the thread from *SUSPENDED* to *RUNNABLE*.
sleep(int millisecs)
Cause the thread to enter the *SUSPENDED* state for the specified time.
yield()
Enter the *READY* state and invoke the scheduler.
destroy()
Destroy the thread.

Figure 3. Java thread constructor and management methods

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*.

Figure 4. Java thread synchronization calls

- Programs can manage threads in groups.
- Every thread belongs to one group assigned at thread creation time.
- Thread groups are useful to shield various applications running in parallel on one Java Virtual Machine (JVM).
  - A thread in one group cannot perform management operations on a thread in another group.
    - E.g., an application thread may not interrupt a system windowing (AWT) thread.

Thread synchronization

- The main difficult issues in multi-threaded programming are the sharing of objects and the techniques used for thread coordination and cooperation.
- Each thread's local variables in methods are private to it.
  - Threads have private stack.
- Threads do not have private copies of static (class) variables or object instance variables.
- Java provides the synchronized keyword for thread coordination.
  - any object can only be accessed through one invocation of any of its synchronized methods.

- an object can have synchronized and non-synchronized methods.

- example
  - synchronized addTo() and removeFrom() methods to serialize requests in worker pool example.

- Threads can be blocked and woken up
  - The thread awaiting a certain condition calls an object's wait() method.
  - The other thread calls notify() or notifyAll() to awake one or all blocked threads.

- 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 the worker.

- The Java synchronization methods are given in Figure 4.

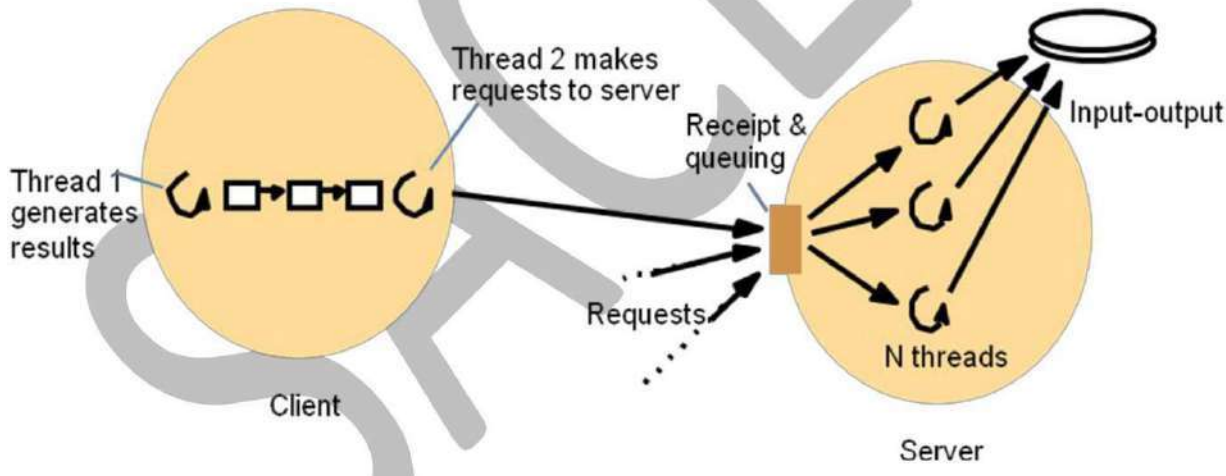


Fig: Client and server with threads