TECHNO INTERNATIONAL
BATANAGAR

TECHNO INTERNATIONAL BATANAGAR

OPERATING SYSTEM TERM PAPER

TOPIC :: *Distributed Operating System*

Name : Sanket Ghosh

Roll : 33200118017

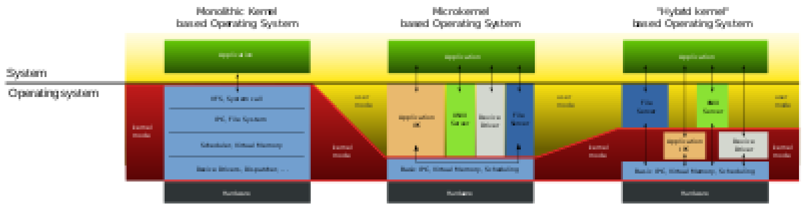
Reg. No : 183320110030

Year : 3rd Year

Semester: 5th

Dept. : Computer Science Engineering

Description:



Structure of monolithic kernel, microkernel and hybrid kernel-based operating systems

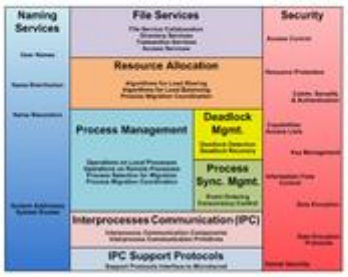
A distributed OS provides the essential services and functionality required of an OS but adds attributes and particular configurations to allow it to support additional requirements such as increased scale and availability. To a user, a distributed OS works in a manner similar to a single-node, monolithic operating system. That is, although it consists of multiple nodes, it appears to users and applications as a single-node.

Separating minimal system-level functionality from additional user-level modular services provides a "separation of mechanism and policy". Mechanism and policy can be simply interpreted as "what something is done" versus "how something is done," respectively. This separation increases flexibility and scalability.

Overview:

At each locale (typically a node), the kernel provides a minimally complete set of node-level utilities necessary for operating a node's underlying hardware and resources. These mechanisms include allocation, management, and disposition of a node's resources, processes, communication, and input/output management support functions. Within the kernel, the communications sub-system is of foremost importance for a distributed OS.

In a distributed OS, the kernel often supports a minimal set of functions, including low-level address space management, thread management, and inter-process communication (IPC). A kernel of this design is referred to as a *microkernel*. Its modular nature enhances reliability and security, essential features for a distributed OS. It is common for a kernel to be identically replicated over all nodes in a system and therefore that the nodes in a system use similar hardware. The combination of minimal design and ubiquitous node coverage enhances the global system's extensibility, and the ability to dynamically introduce new nodes or services.



System management components overview

System management

System management components are software processes that define the node's *policies*. These components are the part of the OS outside the kernel. These components provide higher-level communication, process and resource management, reliability, performance and security. The components match the functions of a single-entity system, adding the transparency required in a distributed environment.

The distributed nature of the OS requires additional services to support a node's responsibilities to the global system. In addition, the system management components accept the "defensive" responsibilities of reliability, availability, and persistence. These responsibilities can conflict with each other. A consistent approach, balanced perspective, and a deep understanding of the overall system can assist in identifying diminishing returns. Separation of policy and mechanism mitigates such conflicts.

Working together as an operating system

The architecture and design of a distributed operating system must realize both individual node and global system goals. Architecture and design must be approached in a manner consistent with separating policy and mechanism. In doing so, a distributed operating system attempts to provide an efficient and reliable distributed computing framework allowing for an absolute minimal user awareness of the underlying command and control efforts.

The multi-level collaboration between a kernel and the system management components, and in turn between the distinct nodes in a distributed operating system is the functional challenge of the distributed operating system. This is the point in the system that must maintain a perfect harmony of purpose, and simultaneously maintain a complete disconnect of intent from implementation. This challenge is the distributed operating system's opportunity to produce the foundation and framework for a reliable, efficient, available, robust, extensible, and scalable system. However, this opportunity comes at a very high cost in complexity.

The price of complexity

In a distributed operating system, the exceptional degree of inherent complexity could easily render the entire system an anathema to any user. As such, the logical price of realizing a distributed operation system must be calculated in terms of overcoming vast amounts of complexity in many areas, and on many levels. This calculation includes the depth, breadth, and range of design investment and architectural planning required in achieving even the most modest implementation.

These design and development considerations are critical and unforgiving. For instance, a deep understanding of a distributed operating system's overall architectural and design detail is required at an exceptionally early point. An exhausting array of design considerations are inherent in the development of a distributed operating system. Each of these design considerations can potentially affect many of the others to a significant degree. This leads to a massive effort in balanced approach, in terms of the individual design considerations, and many of their permutations. As an aid in this effort, most rely on documented experience and research in distributed computing power.

History:

Research and experimentation efforts began in earnest in the 1970s and continued through 1990s, with focused interest peaking in the late 1980s. A number of distributed operating systems were introduced during this period; however, very few of these implementations achieved even modest commercial success.

Fundamental and pioneering implementations of primitive distributed operating system component concepts date to the early 1950s. Some of these individual steps were not focused directly on distributed computing, and at the time, many may not have realized their important impact. These pioneering efforts laid important groundwork, and inspired continued research in areas related to distributed computing.

In the mid-1970s, research produced important advances in distributed computing. These breakthroughs provided a solid, stable foundation for efforts that continued through the 1990s.

The accelerating proliferation of multi-processor and multi-core processor systems research led to a resurgence of the distributed OS concept.

1950s

The DYSEAC

One of the first efforts was the DYSEAC, a general-purpose synchronous computer. In one of the earliest publications of the Association for Computing Machinery, in April 1954, a researcher at the National Bureau of Standards – now the National Institute of Standards and Technology (NIST) – presented a detailed specification of the DYSEAC. The introduction focused upon the requirements of the intended applications, including flexible communications, but also mentioned other computers:

Finally, the external devices could even include other full-scale computers employing the same digital language as the DYSEAC. For example, the SEAC or other computers similar to it could be harnessed to the DYSEAC and by use of coordinated programs could be made to work together in mutual cooperation on a common task… Consequently[,] the computer can be used to coordinate the diverse activities of all the external devices into an effective ensemble operation.

— *ALAN L. LEINER, System Specifications for the DYSEAC*

The specification discussed the architecture of multi-computer systems, preferring peer-to-peer rather than master-slave.

Each member of such an interconnected group of separate computers is free at any time to initiate and dispatch special control orders to any of its partners in the system. As a consequence, the supervisory control over the common task may initially be loosely distributed throughout the system and then temporarily concentrated in one computer, or even passed rapidly from one machine to the other as the need arises. …the various interruption facilities which have been described are based on mutual cooperation between the computer and the external devices subsidiary to it, and do not reflect merely a simple master-slave relationship.

— *ALAN L. LEINER, System Specifications for the DYSEAC*

This is one of the earliest examples of a computer with distributed control. The Dept. of the Army reports certified it reliable and that it passed all acceptance tests in April 1954. It was completed and delivered on time, in May 1954. This was a "portable computer", housed in a tractor-trailer, with 2 attendant vehicles and 6 tons of refrigeration capacity.

Lincoln TX-2

Described as an experimental input-output system, the Lincoln TX-2 emphasized flexible, simultaneously operational input-output devices, i.e., multiprogramming. The design of the TX-2 was modular, supporting a high degree of modification and expansion.

The system employed The Multiple-Sequence Program Technique. This technique allowed multiple program counters to each associate with one of 32 possible sequences of program code. These explicitly prioritized sequences could be interleaved and executed concurrently, affecting not only the computation in process, but also the control flow of sequences and switching of devices as well. Much discussion related to device sequencing.

Similar to DYSEAC the TX-2 separately programmed devices can operate simultaneously, increasing throughput. The full power of the central unit was available to any device. The TX-2 was another example of a system exhibiting distributed control, its central unit not having dedicated control.

Intercommunicating Cells

One early effort at abstracting memory access was Intercommunicating Cells, where a cell was composed of a collection of memory elements. A memory element was basically a binary electronic flip-flop or relay. Within a cell there were two types of elements, *symbol* and *cell*. Each cell structure stores data in a string of symbols, consisting of a name and a set of parameters. Information is linked through cell associations.

The theory contended that addressing is a wasteful and non-valuable level of indirection. Information was accessed in two ways, direct and cross-retrieval. Direct retrieval accepts a name and returns a parameter set. Cross-retrieval projects through parameter sets and returns a set of names containing the given subset of parameters. This was similar to a modified hash table data structure that allowed multiple values (parameters) for each key (name).

Cellular memory would have many advantages:

- A major portion of a system's logic is distributed within the associations of information stored in the cells,
- This flow of information association is somewhat guided by the act of storing and retrieving,

- The time required for storage and retrieval is mostly constant and completely unrelated to the size and fill-factor of the memory
- Cells are logically indistinguishable, making them both flexible to use and relatively simple to extend in size

This configuration was ideal for distributed systems. The constant-time projection through memory for storing and retrieval was inherently atomic and exclusive. The cellular memory's intrinsic distributed characteristics would be invaluable. The impact on the user, hardware/device, or Application programming interfaces was indirect. The authors were considering distributed systems, stating:

We wanted to present here the basic ideas of a distributed logic system with... the macroscopic concept of logical design, away from scanning, from searching, from addressing, and from counting, is equally important. We must, at all cost, free ourselves from the burdens of detailed local problems which only befit a machine low on the evolutionary scale of machines.

— Chung-Yeol (C. Y.) Lee, Intercommunicating Cells, Basis for a Distributed Logic Computer

Foundational work

Coherent memory abstraction

Algorithms for scalable synchronization on shared-memory multiprocessors

File System abstraction

Measurements of a distributed file system
Memory coherence in shared virtual memory systems

Transaction abstraction

Transactions
Sagas

Transactional Memory
Composable memory transactions
Transactional memory: architectural support for lock-free data structures
Software transactional memory for dynamic-sized data structures
Software transactional memory

Persistence abstraction

OceanStore: an architecture for global-scale persistent storage

Coordinator abstraction

Weighted voting for replicated data
Consensus in the presence of partial synchrony

Reliability abstraction

Sanity checks
The Byzantine Generals Problem
Fail-stop processors: an approach to designing fault-tolerant computing systems

Recoverability
Distributed snapshots: determining global states of distributed systems
Optimistic recovery in distributed systems

Three basic distributions

To better illustrate this point, examine three system architectures; centralized, decentralized, and distributed. In this examination, consider three structural aspects: organization, connection, and control. Organization describes a system's physical arrangement characteristics. Connection covers the communication pathways among nodes. Control manages the operation of the earlier two considerations.

Organization

A centralized system has one level of structure, where all constituent elements directly depend upon a single control element. A decentralized system is hierarchical. The bottom level unites subsets of a system's entities. These entity subsets in turn combine at higher levels, ultimately culminating at a central master element. A distributed system is a collection of autonomous elements with no concept of levels.

Connection

Centralized systems connect constituents directly to a central master entity in a hub and spoke fashion. A decentralized system (aka network system) incorporates direct and indirect paths between constituent elements and the central entity. Typically this is configured as a hierarchy with only one shortest path between any two elements. Finally, the distributed operating system requires no pattern; direct and indirect connections are possible between any two elements. Consider the 1970s phenomena of “string art” or a spirograph drawing as a fully connected system, and the spider's web or the Interstate Highway System between U.S. cities as examples of a *partially connected system*.

Control

Centralized and decentralized systems have directed flows of connection to and from the central entity, while distributed systems communicate along arbitrary paths. This is the pivotal notion of the third consideration. Control involves allocating tasks and data to system elements balancing efficiency, responsiveness, and complexity.

Centralized and decentralized systems offer more control, potentially easing administration by limiting options. Distributed systems are more difficult to explicitly control, but scale better horizontally and offer fewer points of system-wide failure. The associations conform to the needs imposed by its design but not by organizational chaos

Design considerations:

Transparency

Transparency or *single-system image* refers to the ability of an application to treat the system on which it operates without regard to whether it is distributed and without regard to hardware or other implementation details. Many areas of a system can benefit from transparency, including access, location, performance, naming, and migration. The consideration of transparency directly affects decision making in every aspect of design of a distributed operating system. Transparency can impose certain requirements and/or restrictions on other design considerations.

Systems can optionally violate transparency to varying degrees to meet specific application requirements. For example, a distributed operating system may present a hard drive on one computer as "C:" and a drive on another computer as "G:". The user does not require any knowledge of device drivers or the drive's location; both devices work the same way, from the application's perspective. A less transparent interface might require the application to know which computer hosts the drive. Transparency domains:

- *Location transparency* – Location transparency comprises two distinct aspects of transparency, naming transparency and user mobility. Naming transparency requires that nothing in the physical or logical references to any system entity should expose any indication of the entity's location, or its local or remote relationship to the user or application. User mobility requires the consistent referencing of system entities, regardless of the system location from which the reference originates.
- *Access transparency* – Local and remote system entities must remain indistinguishable when viewed through the user interface. The distributed operating system maintains this perception through the exposure of a single access mechanism for a system entity, regardless of that entity being local or remote to the user. Transparency dictates that any differences in methods of accessing any particular system entity—either local or remote—must be both invisible to, and undetectable by the user.
- *Migration transparency* – Resources and activities migrate from one element to another controlled solely by the system and without user/application knowledge or action.
- *Replication transparency* – The process or fact that a resource has been duplicated on another element occurs under system control and without user/application knowledge or intervention.

- *Concurrency transparency* – Users/applications are unaware of and unaffected by the presence/activities of other users.
- *Failure transparency* – The system is responsible for detection and remediation of system failures. No user knowledge/action is involved other than waiting for the system to resolve the problem.
- *Performance Transparency* – The system is responsible for the detection and remediation of local or global performance shortfalls. Note that system policies may prefer some users/user classes/tasks over others. No user knowledge or interaction. is involved.
- *Size/Scale transparency* – The system is responsible for managing its geographic reach, number of nodes, level of node capability without any required user knowledge or interaction.
- *Revision transparency* – The system is responsible for upgrades and revisions and changes to system infrastructure without user knowledge or action.
- *Control transparency* – The system is responsible for providing all system information, constants, properties, configuration settings, etc. in a consistent appearance, connotation, and denotation to all users and applications.
- *Data transparency* – The system is responsible for providing data to applications without user knowledge or action relating to where the system stores it.
- *Parallelism transparency* – The system is responsible for exploiting any ability to parallelize task execution without user knowledge or interaction. Arguably the most difficult aspect of transparency, and described by Tanenbaum as the "Holy grail" for distributed system designers.

Inter-process communication

Inter-Process Communication (IPC) is the implementation of general communication, process interaction, and dataflow between threads and/or processes both within a node, and between nodes in a distributed OS. The intra-node and inter-node communication requirements drive low-level IPC design, which is the typical approach to implementing communication functions that support transparency. In this sense, Interprocess communication is the greatest underlying concept in the low-level design considerations of a distributed operating system.

Process management

Process management provides policies and mechanisms for effective and efficient sharing of resources between distributed processes. These policies and mechanisms support operations involving the allocation and de-allocation of processes and ports to processors, as well as mechanisms to run, suspend, migrate, halt, or resume process execution. While these resources and operations can be either local or remote with respect to each other, the distributed OS maintains state and synchronization over all processes in the system.

As an example, load balancing is a common process management function. Load balancing monitors node performance and is responsible for shifting activity across nodes when the system is out of balance. One load balancing function is picking a process to move. The kernel may employ several selection mechanisms, including priority-based choice. This mechanism chooses a process based on a policy such as 'newest request'. The system implements the policy

Resource management

Systems resources such as memory, files, devices, etc. are distributed throughout a system, and at any given moment, any of these nodes may have light to idle workloads. *Load sharing* and load balancing require many policy-oriented decisions, ranging from finding idle CPUs, when to move, and which to move. Many algorithms exist to aid in these decisions; however, this calls for a second level of decision making policy in choosing the algorithm best suited for the scenario, and the conditions surrounding the scenario.

Reliability

Distributed OS can provide the necessary resources and services to achieve high levels of *reliability*, or the ability to prevent and/or recover from errors. Faults are physical or logical defects that can cause errors in the system. For a system to be reliable, it must somehow overcome the adverse effects of faults.

The primary methods for dealing with faults include *fault avoidance*, fault tolerance, and *fault detection and recovery*. Fault avoidance covers proactive measures taken to minimize the occurrence of faults. These proactive measures can be in the form of *transactions*, replication and backups. Fault tolerance is the ability of a system to continue operation in the presence of a fault. In the event, the system should detect and recover full functionality. In any event, any actions taken should make every effort to preserve the *single system image*.

Availability

Availability is the fraction of time during which the system can respond to requests.

Performance

Many benchmark metrics quantify performance; throughput, response time, job completions per unit time, system utilization, etc. With respect to a distributed OS, performance most often distills to a balance between process parallelism and IPC. Managing the task granularity of parallelism in a sensible relation to the messages required for support is extremely effective. Also, identifying when it is more beneficial to migrate a process to its data, rather than copy the data, is effective as well.

Synchronization

Cooperating concurrent processes have an inherent need for synchronization, which ensures that changes happen in a correct and predictable fashion. Three basic situations that define the scope of this need:

- one or more processes must synchronize at a given point for one or more other processes to continue,
- one or more processes must wait for an asynchronous condition in order to continue,
- or a process must establish exclusive access to a shared resource.

Improper synchronization can lead to multiple failure modes including loss of atomicity, consistency, isolation and durability, deadlock, livelock and loss of serializability.

Flexibility

Flexibility in a distributed operating system is enhanced through the modular and characteristics of the distributed OS, and by providing a richer set of higher-level services. The completeness and quality of the kernel/microkernel simplifies implementation of such services, and potentially enables service providers greater choice of providers for such services.

Research:

Replicated model extended to a component object model

Architectural Design of E1 Distributed Operating System
The Cronus distributed operating system
Design and development of MINIX distributed operating system

Complexity/Trust exposure through accepted responsibility

Scale and performance in the Denali isolation kernel

Multi/Many-core focused systems

The multikernel: a new OS architecture for scalable multicore systems.
Corey: an Operating System for Many Cores.
Almos: Advanced Locality Management Operating System for cc-NUMA Many-Cores.

Distributed processing ov

er extremes in heterogeneity

Helios: heterogeneous multiprocessing with satellite kernels.

Effective and stable in multiple levels of complexity

Tessellation: Space-Time Partitioning in a Manycore Client OS.