Outline

- Introduction
- Background
- Distributed DBMS Architecture
- Distributed Database Design
- Semantic Data Control
- Distributed Query Processing
- Distributed Transaction Management
- Distributed Database Operating Systems
  - Distributed DBMS Requirements
  - Problem Areas
  - Architectural Issues
- Open Systems and Interoperability
- Parallel Database Systems
- Distributed Object Management
- Concluding Remarks

Operating System Functions

- Hardware management
- Process management
- Resource allocation (scheduling)
- Storage management and access (I/O)
- Memory management
- File system service
- Protection
- Better reliability (survivability)
- Scalability (expendability)
- Improved performance
- Support for heterogeneity

Distributed OS functions

Distributed DBMS Requirements

- More complex access method
- More suitable buffer and memory management
- Finer granularity of concurrency control
- Support for transaction commit/recovery
- Higher level access support (views)

Bottlenecks

- Processor bottleneck
  - unsuitability of von Neumann architecture for non-numeric computation
  - technological limitations
  - database machines
  - new technologies
- Database bottleneck
  - problems of managing large volumes of data
  - distributed database management

Abstraction Levels

Abstraction level	Definition	Objects	
total transparency	global conceptual schema	relations	Global abstractions
fragmentation transparency	fragmentation schema	fragments of relations	
replication transparency	replication schema	multiple copies of fragments of relations	
network transparency	remote communication services	remotely located multiple copies of fragments	
logical data independence	local conceptual schema	local relations	Local abstractions
physical data independence	physical schema	records, access paths	
file system	file definitions and buffer management	physical records, pages	
storage and I/O system	disk storage definition (VTOC)	tracks, physical blocks	

Data stored on disks

Issues

- Naming, access control and protection
- Scheduling
  - Local process management
  - Distributed scheduling
- Remote communication
- Persistent data management
- Buffer and memory management
- Transaction support

Naming and Transparency

- Naming of system resources should permit transparent access.
- Transparency :
  - separation of the higher level semantics of the system from the lower level implementation issues
  - accessing distributed resources should be identical to accessing local resources
- Three considerations
  - Types of transparencies
  - Who should provide these services
  - Where do you perform the name translation

Who Should Provide Transparency?

- The Operating System should take the responsibility of providing network and distribution transparency and should provide *support* for the replication transparency.
- The Distributed DBMS should be responsible for providing full replication and fragmentation transparency.

Data independence	DOS & Distributed DBMS
Network transparency	DOS
Distribution transparency	DOS
Replication transparency	DOS
Fragmentation transparency	Distributed DBMS

Naming and Transparency

- Naming is the fundamental means of providing transparency.
- Hard to build transparency on top of an OS naming facility that does not support it.
  - path specification
  - aliasing
  - remote login to a DBMS
- Naming should also permit concurrent access

Name Translation

Place of translation?

- Depends on network topology, control algorithms, etc.
  - Centralized
  - Distributed
  - Keep track of the names and addresses of recently communicated objects.

Access Control & Naming

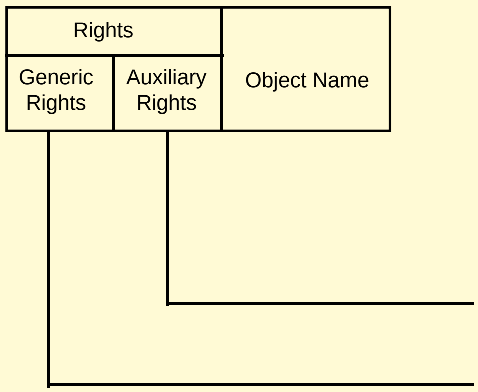
- Authorization Control
  - Only authorized users should have access to resources
  - Users should have access only to the resources permitted to them
  - Implemented by security levels and groups of users
- Authentication
  - Determining that a user is indeed who he/she claims to be
  - Password, fingerprint identification, etc

Capabilities

- A uniform and unified mechanism for handling naming and access control problems.
- A *capability* can be thought of as a pointer to an object which the owner of the capability can access.

Capability Structure

- A capability indicates two things :
  - an object which the owner of the capability can access
  - what types of operations the owner can perform on that object



Some Possible Generic Rights

CreateObjectRts	allows an object to be created
CopyRts	allows the generation of a new capability for the same object
ReadDataRts	permits the data section of the object to be read
WriteDataRts	permits updating the data of the object
ReadCapaRts	allows querying of capabilities owned by the named object
WriteCapaRts	permits changing of object capabilities
RestrictRts	prevents amplification (alteration of rights)

Two Fundamental Issues

- Rights amplification
  - composite objects use other object ⇨ users of composite objects may need rights they do not have
  - a server operation may amplify the rights of the client
  - should have a mechanism to stop amplification ⇨ **RestrictRts** generic right
- Protection Domain
  - small ⇨ change domain at every procedure call
    - very tight control
    - very expensive
  - large ⇨ change domain at object boundary

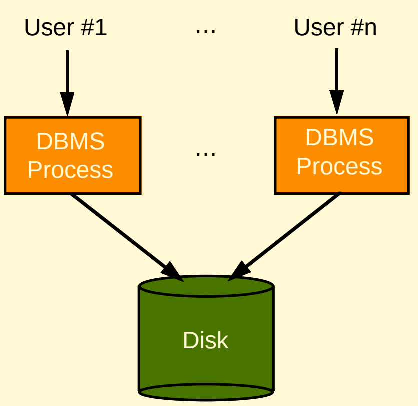
Process Management

- A process is a program in execution.
- A program can execute as more than a single process.
- Local process management
- Distributed scheduling

Process Management Functions

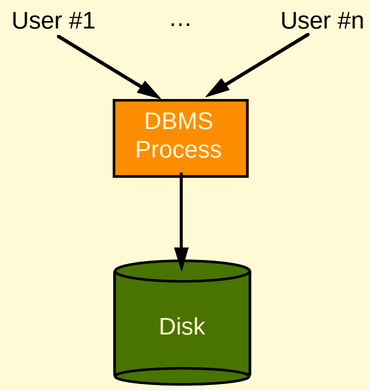
- Low level process control
  - Create
  - Destroy
  - Fork
  - Join
- Dispatching
  - running, ready, blocked states
- Synchronization ➩ semaphore object
  - Create
  - Destroy
  - Initialize
  - P
  - V
- Deadlock Management

Process-Per-User Approach



- Process manager has to permit sharing of data segments between processes
- Expensive to create a process for each user request
- Each DBMS request (e.g., I/O) causes a context switch
- Convoy phenomenon

Server Process Approach



- DBMS should be able to do its own multi-tasking
  - Multi-threading
- Efficient communication between the users and the server has to be established

Distributed Scheduler

- Two primitives are necessary for distributed scheduling of tasks:

  schedule

  unschedule

- The distributed scheduling problems dead with the policies involved in defining the semantics of these two primitive operations.

Scheduling on a Single Processor

- FCFS
- Round–robin
- Shortest time first
- Priority

Distributed Process Management

- How to manage system state tables
  - distributed → no immediate knowledge of system state at other sites
  - centralized → reliability and performance concerns
- How to extend either of process structuring paradigms to distributed case
  - process-per-user → process migration
  - server process → abstraction level for communication between the uses and the server
- Conflicting distributed scheduling objectives (due to Tanenbaum & van Renesse)
  - maximizing throughput
  - minimizing response time
  - load balancing

Distributed Scheduling

- Co-scheduling
  - Find a subset of processes that should run together and schedule them to execute on different processors, but during the same time slice.
- Localization
  - Find a subset of processes that should run together and schedule them to execute on the same processor.
- Load Balancing
  - Schedule a process on a processor which has the least load and constantly monitor processor loads.

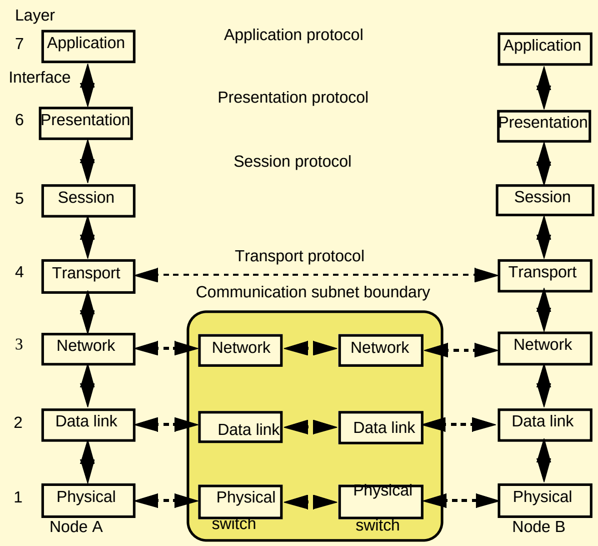
Communication Between System Objects

- Very important because
  - ties the components together
  - has significant impact on performance
  - has significant impact on ease of use
- Requirements:
  - User view: Access to remote resources should be the same as access to local resources
  - Efficient
  - Flexible, easy to use and semantically rich set of primitives
  - Reliable
    - do not lose messages
    - return all undeliverable messages to the sender
  - Remote communication should be compatible with existing network protocols

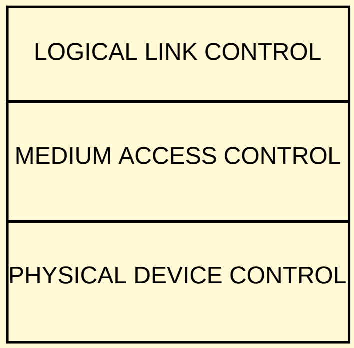
Alternative Communication Primitives

- Message passing
- Remote procedure calls

Physical Message Passing – OSI Architecture



Physical Message Passing – IEEE 802 Architecture



Logical Message Passing

- Two primitives
  - send
  - receive
- Four issues :
  - How is message passing achieved?
  - Are primitives blocking or nonblocking?
  - Are primitives reliable or unreliable?
  - Are the primitives buffered or unbuffered?

Message Passing Alternatives

- Remote communication
  - No choice; message has to be physically delivered
- Local communication
  - Physically copying from sender's address space to receiver's
    - uniform treatment of remote and local messages
  - Rearrange pointers
    - efficient

Blocking/Nonblocking Primitives

- Blocking send
  - The sender is blocked following **send** until a reply is received
  - Equivalent to a **send** followed by a **wait**
- Nonblocking send
  - The sender continues executing following **send**
- Blocking receive
  - The receiver is blocked following **receive** until a message is received from the identified object
- Nonblocking receive

Reliable/Unreliable Primitives

- Unreliable send
  - The sender puts the message on the net and wishes it good luck
  - It is up to the user to include the necessary code in his program to do the end-to-end acknowledgement, retransmission, etc.
- Reliable send
  - The primitive guarantees message transmission
- Unreliable receive
  - The receiver does not take any action
- Reliable receive
  - The receiver sends an ack as soon as message is in the buffer

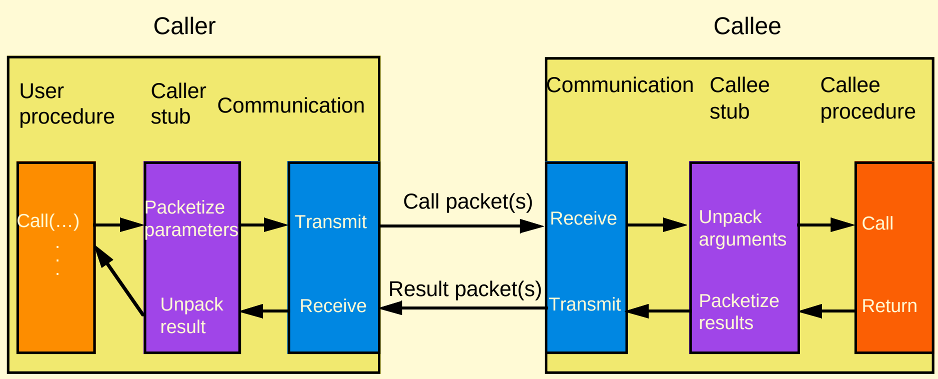
Buffering/Unbuffering Primitives

- Buffered send
  - The sender can send more than one message, without waiting for the receiver to be ready
- Unbuffered send
  - The sender has to wait for the receiver to remove the message from the buffer

Remote Procedure Calls

- Uses the well-known procedure call semantics.
- The caller makes a procedure call and then waits. If it is a local procedure call, then it is handled normally; if it is a remote procedure, then it is handled as a remote procedure call.
- Caller semantics is identical to a blocked send; callee semantics is identical to a blocked receive to get the parameters and a nonblocked send at the end to transmit results.

Remote Procedure Calls



Comparison of Mechanisms

Persistent Data Management

❶ What problems do the DBMSs have with OS file systems?

❷ Can the DBMS be used as the OS persistent data manager (file system)?

❸ What is the role of the programming languages in addressing persistence? How do they enter this picture?

❹ Can the distributed file system designs address the distributed DBMS requirements?

File System Issues

- Lack of record structure
  - Can that abstraction be built on top of character streams?
  - Can a record structured file system be built without losing the flexibility of character streams?
- Access mode incompatibility
  - Clustering vs physical scattering
  - DBMS should not decide, but advise

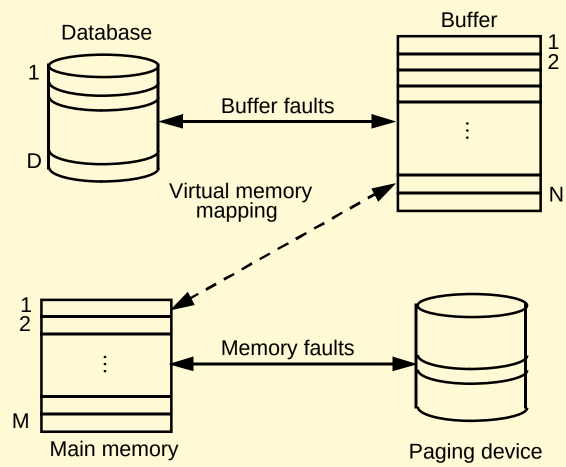
Access Methods

- DBMSs have more sophisticated access
  - simple reusal
  - loop reusal
  - unclustered index access
  - clustered index access
- no B-tree support
- no set-at-a-time access

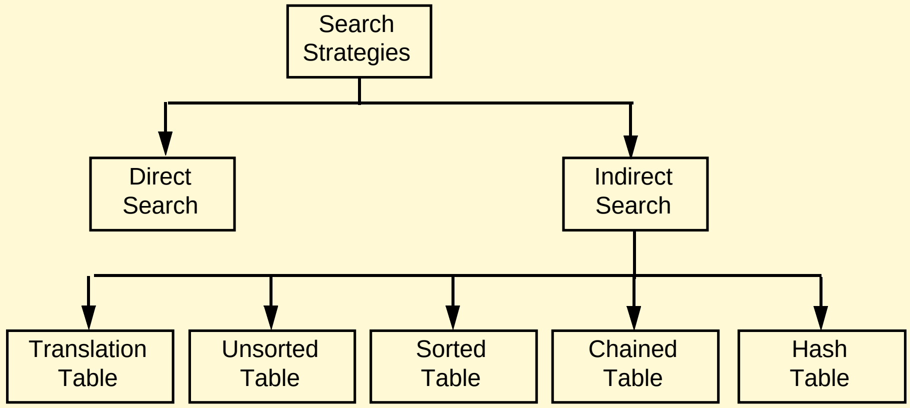
Buffer Management

- DBMS page reference pattern regular ➪ buffer requirement can be calculated precisely
- DBMS may benefit from page prefetching and knows its page reference pattern
- Least recently used (LRU) page replacement algorithm fails for important access patterns
- LRU delay-writes log pages ➪ reliability risk
- Double paging
- Functions
  - *Search* the buffer pool for a given page
  - *Allocate* a free buffer page and *load*
  - Choose a buffer page for *replacement*

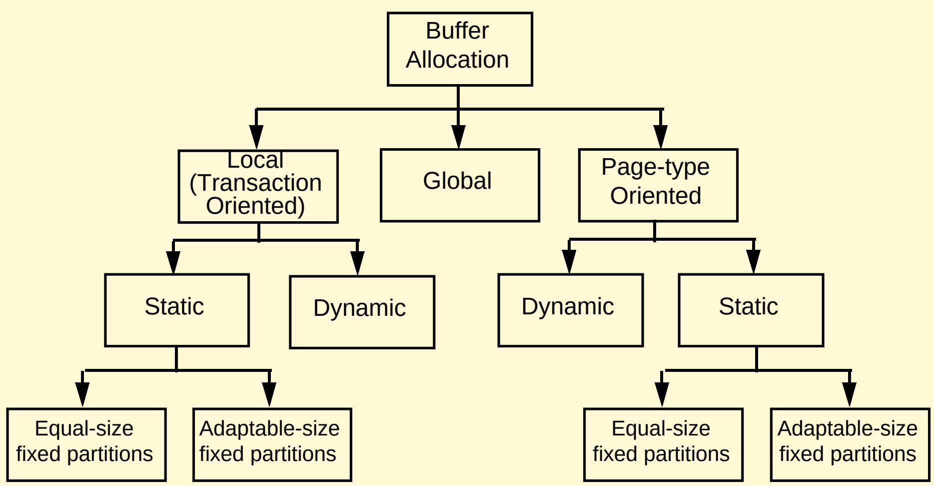
Virtual Memory Management



Buffer Management Strategies



Buffer Allocation Strategies



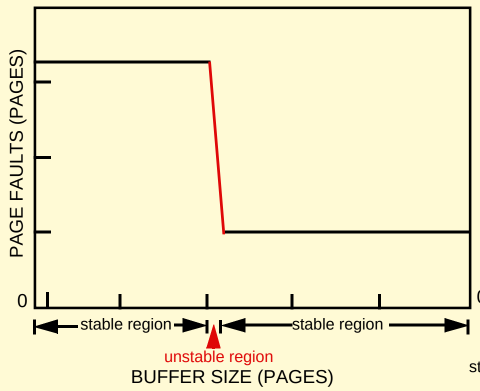
Page Replacement Algorithms

- Page reference by **fix/unfix** primitives.
- Alternative algorithms
  - FIFO
  - LRU
  - CLOCK
  - Working set (WS)
- More recent studies
  - Hot Set Model (HSM)
  - Query Locality Set Model (QLSM) and DBMIN Algorithm

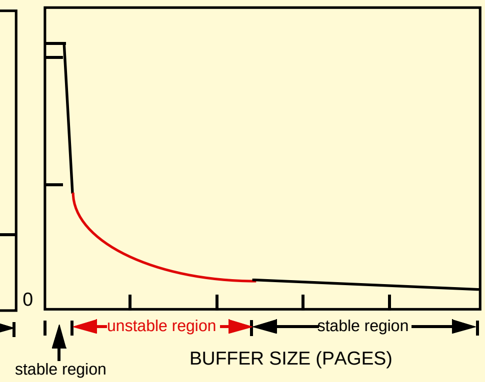
Hot Set Model

- During query optimization determine the necessary number of buffer pages and do not run a process with insufficient number of frames.
- Eliminates or minimizes
  - internal thrashing
  - external thrashing
- similar to working set
- static, a priori estimator
- allocates buffers according to queries' buffer demand
- LRU based

Page Fault Behavior

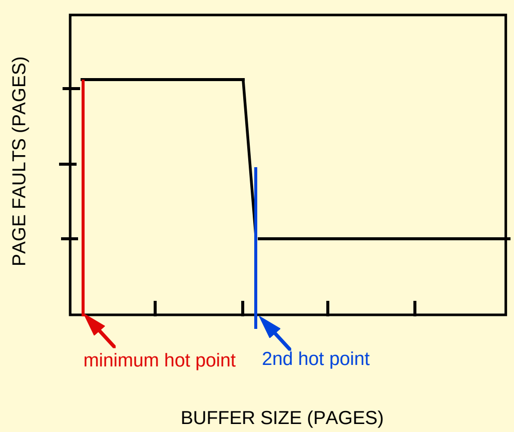


$R_1 \bowtie R_2$
using nested join with
sequential scans

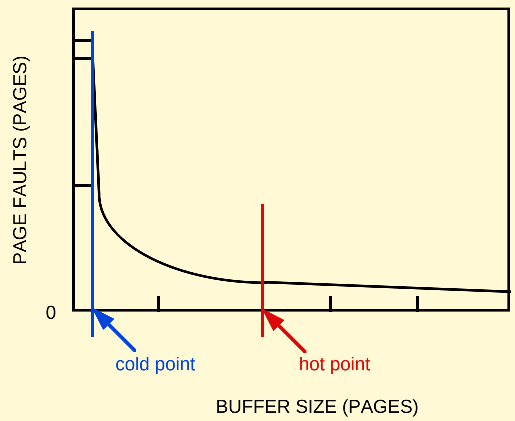


$R_1 \bowtie R_2$
using nested join with
sequential scan on outer
and clustered index on inner

Hot Points



Cold Points



Basic Principle of HSM

❝Buffer sizes inside a stable interval, and different from the hot point relative to that interval, do not produce any benefit in terms of fault reduction, while using more buffer resources.❞

Determining Hot Points

- Find the hot points for each primitive access path
  - simple reusal
  - loop reusal
  - unclustered index reusal
  - clustered index reusal
- Compute hot points for each access strategy that is used by the Distributed DBMS

Management Algorithm

- Query Processor
  - determines *query hot set*
    - the hot point, which is not larger than the system buffer space, at which *buffer consumption* is minimum
  - buffer consumption = hot set size ∗ expected response time in isolation
- Scheduler
  - schedule a query only if enough buffer space exists to accommodate its hot set
  - some over-committment is possible
- Buffer manager
  - local LRU chains
  - search entire buffer ⇨ sharing
  - allocate buffers from free list; if unavailable steal from deficient queries

QLSM/DBMIN

- Separates the modeling of reference behavior from buffer management algorithm
- Allocates buffers according to the requirements of the files that are accessed
- QLSM
  - Provides a classification of primitive access paths that can be used in the allocation algorithm
- DBMIN
  - Determines the *locality set* of files
    - set of buffered pages associated with a file instance
  - Allocates to each file a local buffer pool sufficient to hold its locality set
  - Manages the buffer pool locally

QLSM Reference Classification

- Sequential References
  - Straight sequential (SS)
    - each page referenced only once
    - locality set = 1 buffer page
  - Clustered sequential (CS)
    - sequential scan, local backups during scan
    - merge join, inner relation
    - locality set = keep the records that are in a cluster together in buffer
  - Looping sequential (LS)
    - repeated sequential references
    - nested join, inner relation
    - locality set = entire file
    - If too big, use MRU

QLSM Reference Classification

- Random References
  - Independent random (IR)
    - a series of independent accesses
    - data page access during a unclustered index scan
    - locality set = single buffer page
  - Clustered random (CR)
    - locality of reference in random accesses
    - similar behavior to the clustered sequential scan
    - locality set = each page containing a record in a cluster should be kept in buffer

QLSM Reference Classification

- Hierarchical References
  - Straight hierarchical (SH)
    - one traversal from the root of the index to the leaves
    - locality set = 1 buffer page
  - Hierarchical with straight sequential (H/SS)
    - tree traversal followed by a sequential scan on the leaves
    - locality set = same as SS
  - Hierarchical with clustered sequential (H/CS)
    - tree traversal followed by another type of scan on the leaf page
    - locality set = same as CS
  - Looping hierarchical (LH)
    - repeated accesses to the index structure
    - join where inner relation is indexed on the join field
    - locality set = 1 buffer page for the root index

DBMIN

- Global free list
- Entire buffer is searched ⇨ sharing
- Local buffer management for each file
- Replacement policy according to the reference pattern

Buffer Management Requirements

- Hot set model can be accommodated on top of a working-set/LRU based buffer and memory manager if the system can be modified to permit the DBMS to specify its own reference string.
- DBMIN requires complete change of the buffering techniques.
- Both of these can be implemented within the DBMS on top of the kernel that we discussed.
- The kernel design that we discussed also permits the uniform handling of the buffer management and the memory management issues.

Transaction Support

Issues:

❶ **Should the transaction management be supported as a standard operating system service?**

- Transaction concept would be available for *all* applications, not just the DBMS
- The OS guarantees the ACIDity of any process that runs as a transaction

❷ Should the OS provide some functional support for database transactions?

- If yes, how?
- Transactions continue to be DBMS primitives, but their management is shared between the OS and the DBMS

Benefits of OS Transaction Management

- Elimination of convoys
  - DBMS semaphores **are** OS semaphores
  - OS does not deactivate the DBMS process
- Reduces locking overhead
  - Microcode or special hardware
- Performance of recovery measures can be improved undo/redo logging
  - chained I/O; group commit
- Reduced deadlock management overhead
  - OS can do it for all
  - OS can do it during idle machine cycles

Issues in OS Transaction Management

Three fundamental questions :

❶ Who implements the locking primitives?

❷ Where should the transaction manager be placed?

❸ Granularity of entities on which transactions operate

Who Implements Locking Primitives?

- Transaction manager provides no locking/ Buffer manager provides no locking
  - Why have a transaction manager?
- Transaction manager provides logical locking/ Buffer manager provides no locking
  - Would not work; consistency of data would be compromised.
- Transaction manager provides no locking/ Buffer manager provides page locking
  - Inflexible ⇨ single level of locking granularity
  - Inefficient ⇨ treatment of meta data (e.g., index table)
- Transaction manager provides no locking/ Buffer manager provides page locking
  - Nested transactions and multi-mode locking

Where Should a Transaction Manager Reside?

- Transaction manager in OS kernel/Buffer manager in OS kernel
  - Can implement the OS using transactions for reliability
  - Is it desirable to place the buffer manager in the kernel?
- Transaction manager in OS kernel/Buffer manager within DBMS
  - Kernel process sends a message to a non-kernel process for each lock/unlock

Where Should a Transaction Manager Reside?

- Transaction manager at user level/Buffer manager in DBMS
  - Kernel reliability needs to be handled differently
  - If message-based communication, four messages for each lock/unlock
- Transaction manager at the user level/Buffer manager in the kernel
  - Similar to above
  - Each lock/unlock causes two kernel calls or messages

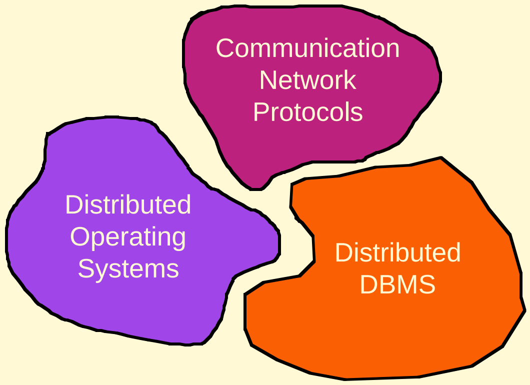
Granularity of Entities

- Page is typical
  - block-oriented I/O spanning multiple pages difficult
  - predicate locking is difficult
- This is closely related to how persistent data management is performed

Potential Solutions

- Multilevel transaction model
- Nested transactions
- Designing a uniform persistent data storage manager that also provides concurrency
- Some existing systems
  - Camelot
    - provides a transaction management layer on top of Mach kernel
  - QuickSilver
    - experimental distributed OS which provides transaction as a reliable computation primitive
  - Argus
    - supports transaction concept as a language primitive

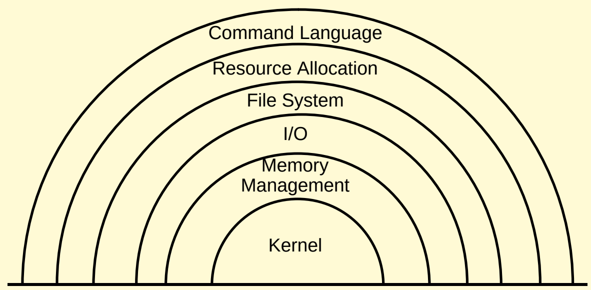
Architectural Cooperation Problem



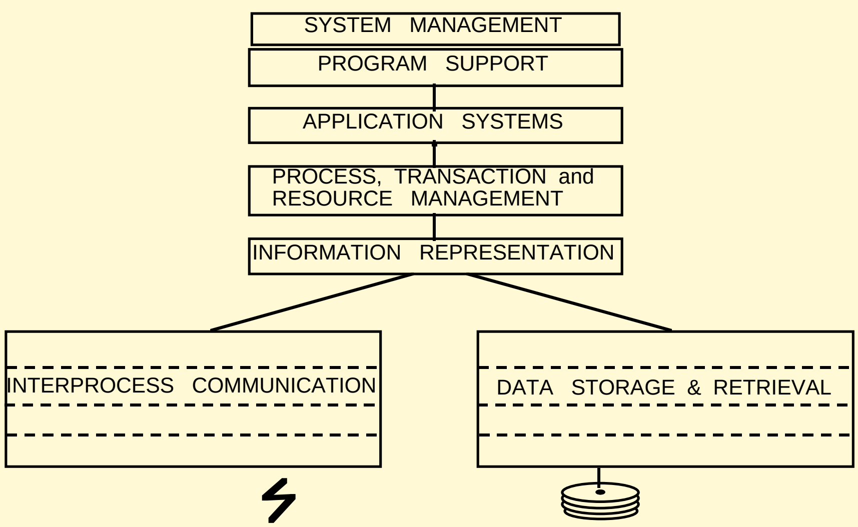
Architectural Paradigms

- Layered architecture
- Unstructured, all kernel (UNIX)
- Client-server (network of servers)
- Object-oriented operating systems

Layered Architecture



Layered Architecture Extensions



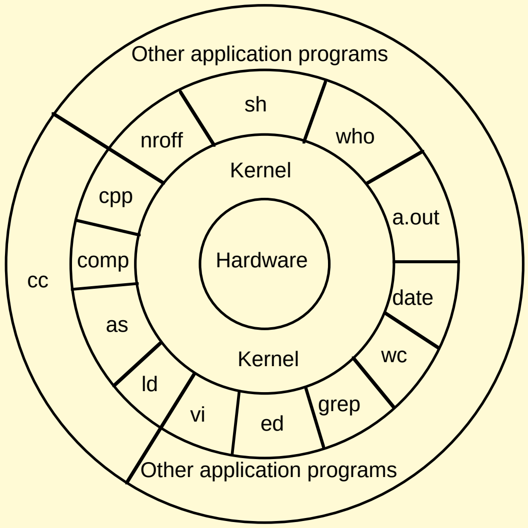
Layered Architecture Extensions

Level	Name	Object	Example operations
15	Shell	User prog. env.	shell language statements
14	Directories	Directories	create,destroy,search,list
13	User processes	User processes	fork quit, kill, suspend,resume
12	Stream I/O	Streams	open, close, read, write
11	Devices	external devices & peripherals	create, destroy, open, close read, write
10	File system	Files	create, destroy, open, close, read, write
9	Communications	Pipes	create, destroy, open, close, read, write
8	Capabilities	Capabilities	create, validate, attenuate
7	Virtual memory	Segments	read, write, fetch
6	Local 2ndary store	block of data	read, write, allocate,
5	Primitive process	process, semaphores	suspend, resume, wait, signal
4	Interrupts	Fault handler prog's	invoke, mask, unmask, retry
3	Procedures	procedure segments	call, return
2	Instruction set	Microprog. interp.	load, store,...
1	Electronic circuits		

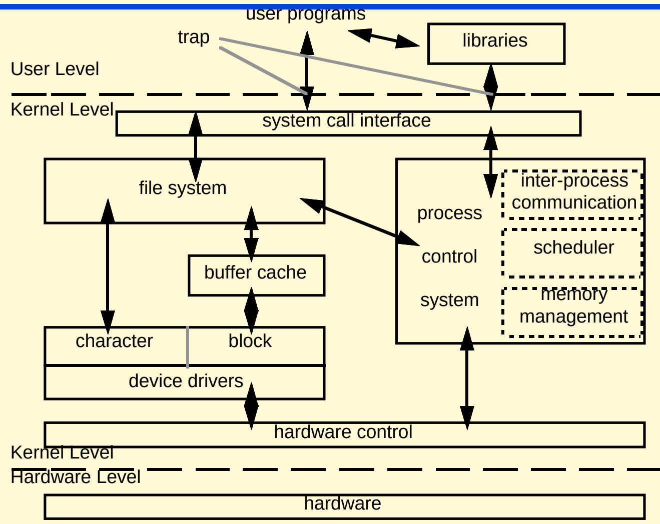
Layered Architecture Extensions

- Advantages
  - Well-understood
  - Clean interface
- Disadvantages
  - Very costly
  - DBMS is still hostage to DOS implementation

UNIX Architecture



UNIX Kernel Architecture



UNIX Architecture

- Advantages
  - Very popular
  - Some nice services
- Disadvantages
  - Horrible for DBMS support
  - Very large kernel; portability problems
  - Kernel imposes services on applications

Client-Server Model

- The operating system is organized as a collection of a set of server modules, each of which provides a simple service.
- Was developed for distributed computing to access remote services (e.g., file servers, printer servers, etc).
- Can be carried to its natural conclusion where the single machine OS services are provided by means of modules as well.

Client-Server Model

- Advantages
  - Abstraction of services
  - Modularization
  - Potential for redundancy (process pairs)
  - Dynamic vs. static layering
- Disadvantages
  - Potential cost of message passing

Object-Oriented OS Model

- The operating system is designed as a set of modules each of which implement one type of object.
- Advantages:
  - User sees the operating system services as a collection of objects.
  - Each OS service module can implement an OS object
  - System can be made easily extendable
    - the "software IC" approach
  - Easy to provide alternative services with similar functions
  - Uniform treatment of persistent and transient data
  - Easier to develop reliable systems

Conclusions

❶ Small kernel

❷ No static layering of operating system services

- Implement only those DBMS functions that can be efficiently provided as kernel services and the get out of the way.

❸ Separation of policies from mechanisms

❹ Object orientation for system structuring