0. Course Overview

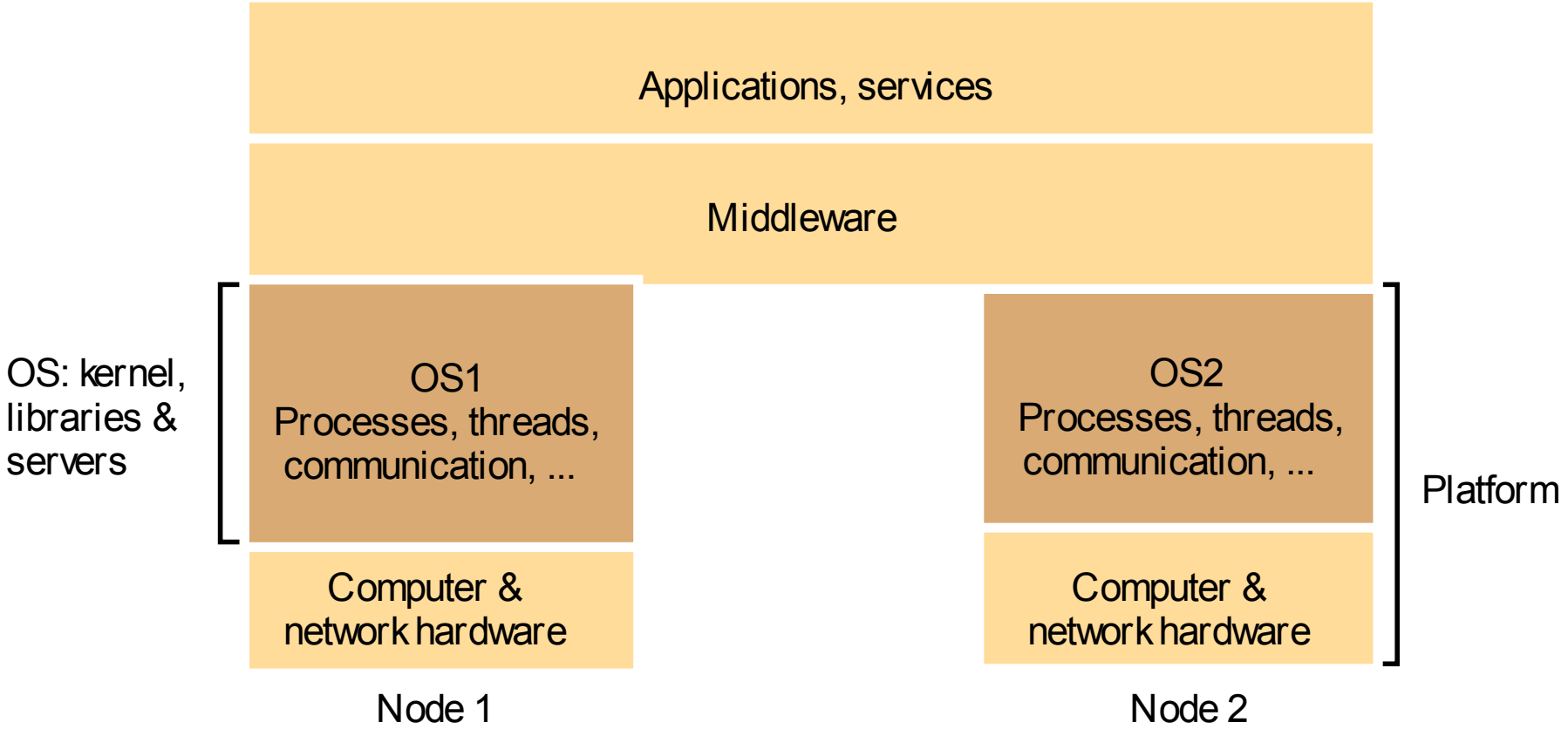
- I. Introduction
- II. Fundamental Concepts of Distributed Systems
  - Architecture models; network architectures: OSI, Internet and LANs; interprocess communication
- III. Time and Global States
  - Clocks and concepts of time; Event ordering; Synchronization; Global states
- IV. Coordination
  - Distributed mutual exclusion; Multicast; Group communication, Byzantine problems (consensus)
- **V. Distribution and Operating Systems**
  - Protection mechanisms; Processes and threads; Networked OS; Distributed and Network File Systems (NFSs)
- VI. Peer to peer systems
  - Routing in P2P, OceanStore, Bittorrent, OneSwarm, Ants P2P, Tor, Freenet, I2P
- VII. Security
  - Security concepts; Cryptographic algorithms; Digital signatures; Authentication; Secure Sockets

Operating Systems for Distributed Systems

- **Network Operating System**
  - one instance running per computer in a network
  - operating system manages local resources as well as access to network infrastucture
    - network file systems
    - rlogin
    - telnet
  - examples
    - Windows (98/NT/2000...)
    - Unix (Solaris, ...)
    - Linux
- **Distributed Operating System**
  - single image system
    - complete transparency for the user where programs run
    - OS has control over all nodes in system
  - not practically in use
    - compatability with existing applications
    - emulations offer very bad performance
  - example
    - Amoeba (Tannenbaum et al.)

Operating Systems for Distributed Systems

- **General Architecture in Practical Use**
  - network operating system + middleware layer



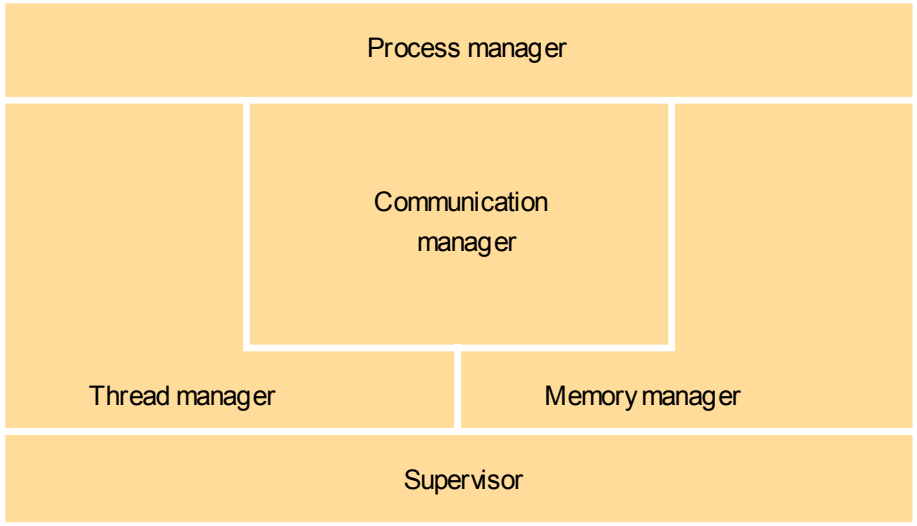
Operating Systems for Distributed Systems

- **Desiderata for Network Operating Systems**
  - provide support for middleware layer to work effectively
    - encapsulation
      - provide transparent service interface to resources of the computer
    - protection
      - protect resources from illegitimate access
    - concurrent processing
      - users/clients may share resources and access concurrently

Operating Systems for Distributed Systems

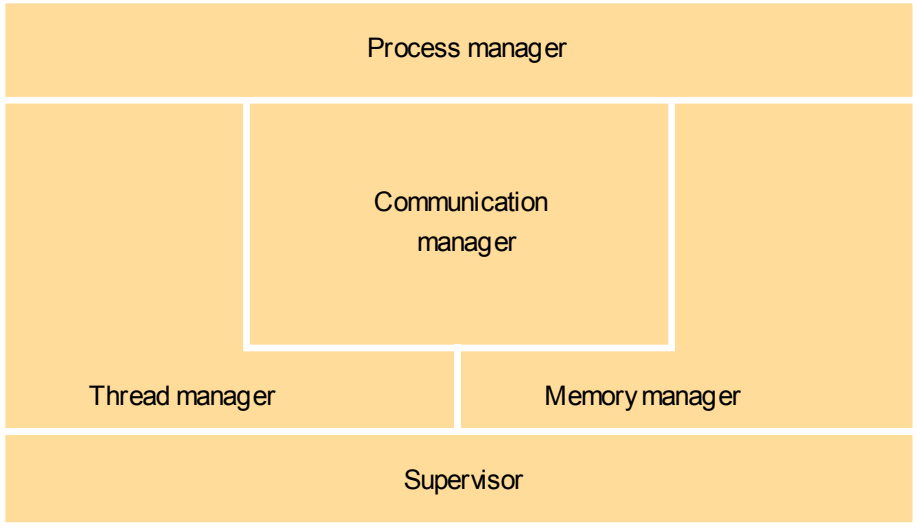
- **Process**
  - software in execution
  - unit of resource management for operating system
    - execution environment
      - address space
      - thread synchronization and communication resources (e.g., semaphores, sockets)
      - computing resources (file systems, windows, etc.)
    - threads
      - schedulable activities attached to processes
      - arise from the need for concurrent activities sharing resources within one process
        - concurrent input/output with problem computation
        - servers: concurrent processing of client requests, each request handled by one thread
  - processes vs. threads
    - threads are “lightweight” processes
      - processes expensive to create, threads easier to create and destroy
  - process instantiation
    - one thread will be instantiated as well, may instantiate offsprings

Operating Systems for Distributed Systems



- **Core Operating System Functionality**
  - process manager
    - maintains processes (creation, termination)
  - thread manager
    - creation, synchronization and scheduling
  - communication manager
    - communication between threads
      - in different processes
      - on different computers

Operating Systems for Distributed Systems



- **Core Operating System Functionality**
  - memory manager
    - management of physical and virtual memory
  - supervisor
    - dispatching of interrupts, system call traps and exceptions
    - control of memory management unit and hardware caches
    - processor and floating point unit register manipulations

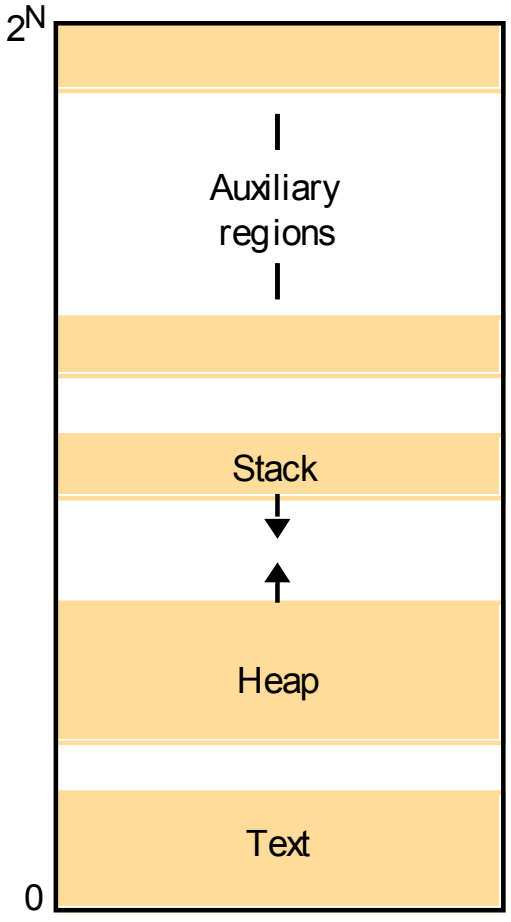
Protection Mechanisms

- **Protection against illegitimate access**
  - clients performing operations need to have right to do so
  - only specified operations may be performed on an object
- **Kernel**
  - core part of operating system that has complete access rights to any resource
  - processor modes
    - user
    - supervisor
      - kernel always executes in supervisor mode

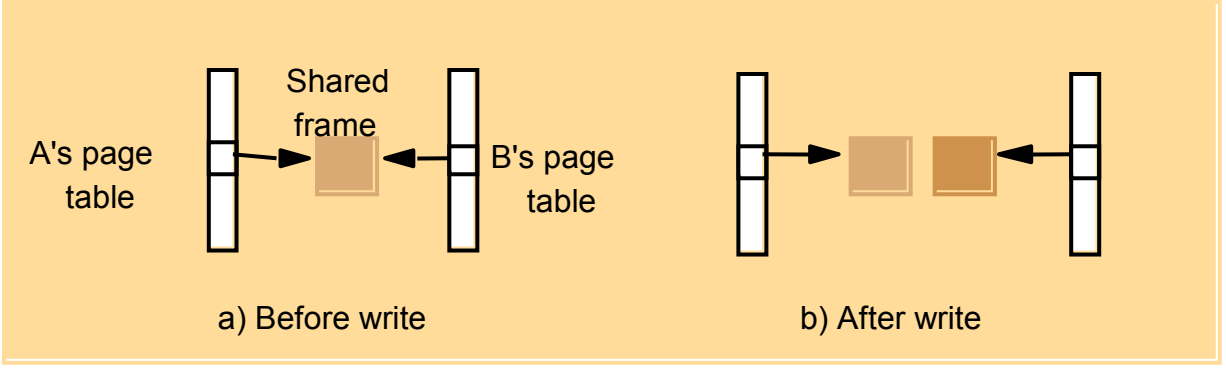
    some operations are only allowed in supervisor mode
  - kernel sets up *address spaces* to protect against illegitimate memory accesses
    - collection of ranges of virtual addresses (memory locations)
    - process cannot access memory locations outside it's address space
    - switching between processes entails switching of address spaces
      - may involve non-negligible amount of work, performance implications

Processes and Threads

- **Address Spaces**
  - regions of memory accessible to threads of that process
  - subdivided into regions
    - lowest address and length
    - read/write/execute permissions for threads in process
    - direction of growth
  - stack
    - for subroutines
    - sometimes one stack region per thread
  - text
    - region to map files into memory
  - shared region
    - regions of virtual memory mapped to identical physical memory for different processes
    - enables inter-process communication



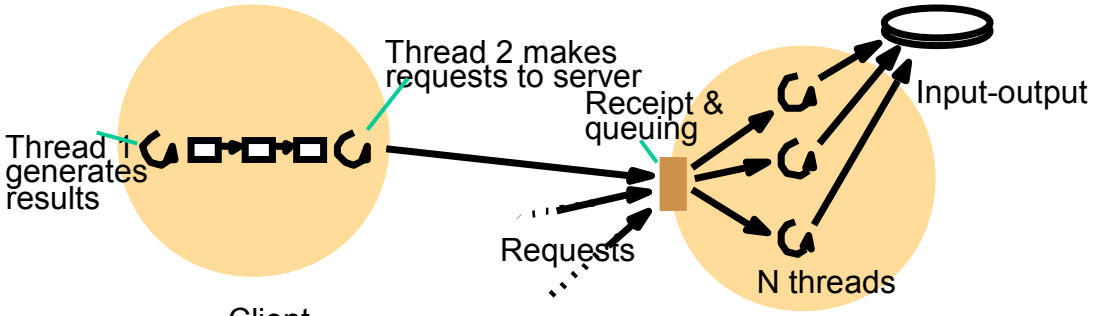
- **Process creation**
  - choice of target host (only for distributed operating systems)
  - creation of an execution environment
    - contents of newly allocated address space
      - initialized “empty”
      - initialized as a (partial) copy of parent’s address space
        - example: Unix `fork` command: child process shares text region with parent, has own copies of stack and heap (an extension allows choices which regions are shared, and which ones are inherited)
      - copy-on write (in Mach operating system)
        - inherited region initially shared
        - only when one process attempts to write, an interrupt handler will start copying the shared region to a new instance

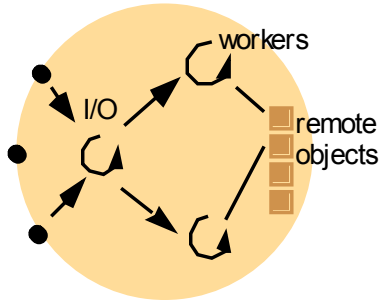


Processes and Threads

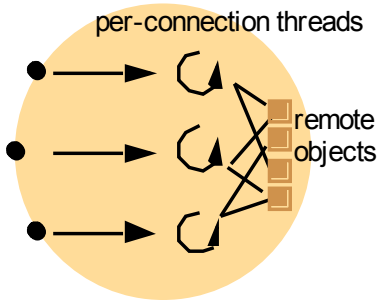
- **Performance considerations: handling server requests**
  - processing: 2 ms
  - IO delay (no caching): 8 ms
  - single thread
    - 10 ms per requests, 100 requests per second
  - two threads (no caching)
    - 8 ms per request, 125 requests per second
  - two threads and caching
    - 75% hit rate
    - mean IO time per request: 0.25 * 8ms = 2ms
      - 500 requests per second
    - increased processing time per request: 2.5 ms
      - 400 requests per second

Processes and Threads

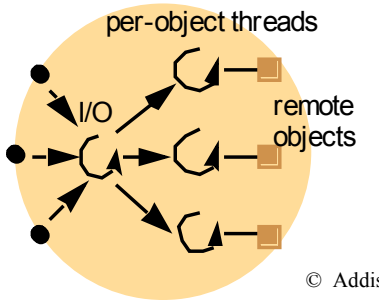




a. Thread-per-request



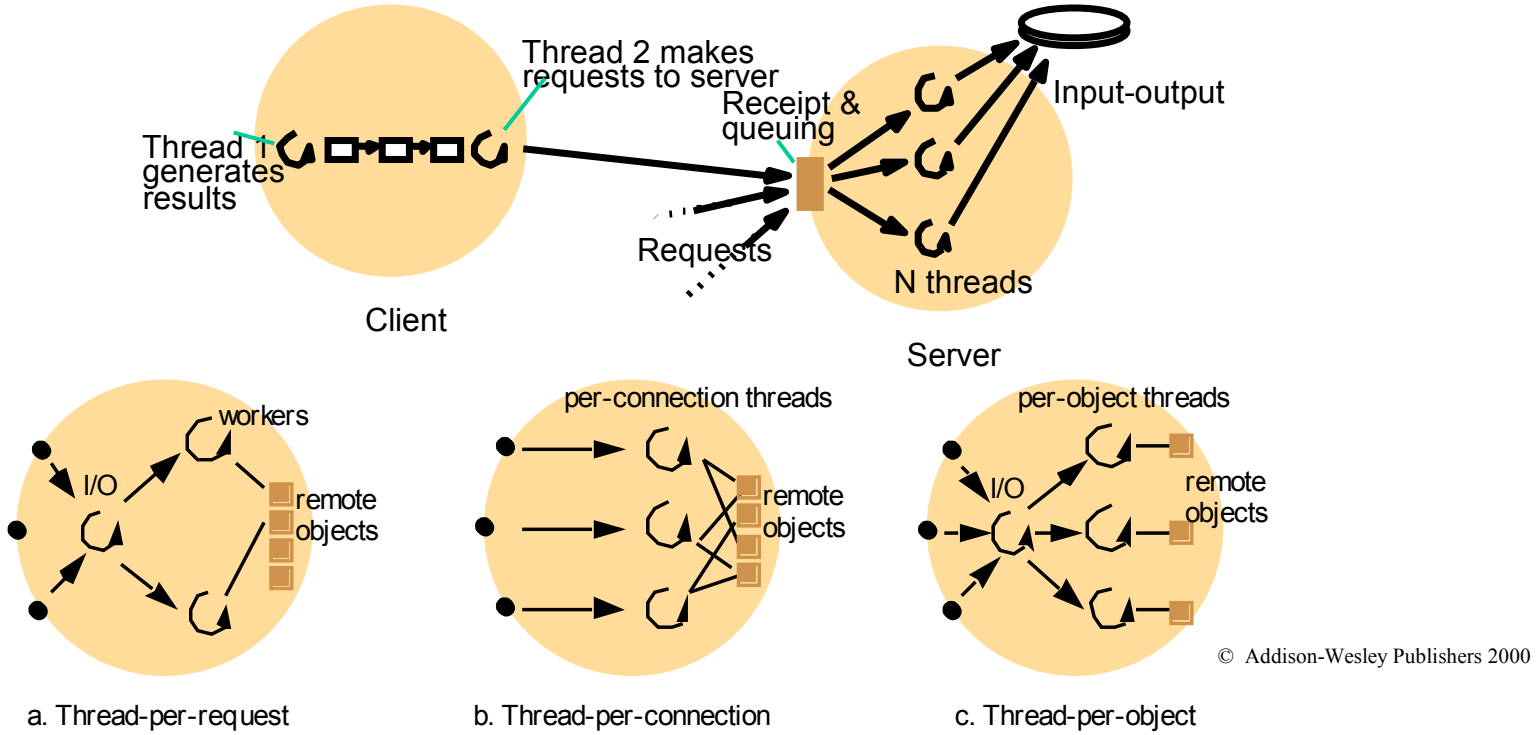
b. Thread-per-connection



c. Thread-per-object

- **Threads and Servers**
  - worker pool
    - pool of server threads serves requests in queue
    - possible to maintain priorities per queue
  - thread per request
    - thread lives only for the duration of request handling
    - maximizes throughput (no queuing)
    - expensive overhead for thread creation and destruction

Processes and Threads



- **Threads and Servers**
  - thread per connection/per object
    - compromise solution
    - no overhead for creation/deletion of threads
    - requests may still block, hence throughput is not maximal

Processes and Threads

Execution environment	*Thread*
Address space tables	Saved processor registers
Communication interfaces, open files	Priority and execution state (such as *BLOCKED*)
Semaphores, other synchronization objects	Software interrupt handling information
List of thread identifiers	Execution environment identifier
Pages of address space resident in memory; hardware cache entries	

- **Threads vs. multiple processes/execution environments**
  - creating a new thread is much less expensive than creating new execution environment
    - creating new thread:
      - allocate region of thread's stack and
      - set registers and processor status
    - creating new execution environment
      - create address space table, communication interfaces
      - new process starts with "empty" cache, therefore more cache misses than for new thread
    - experiment: new process under Unix 11ms, new thread under Topaz kernel: 1 ms

Processes and Threads

Execution environment	*Thread*
Address space tables	Saved processor registers
Communication interfaces, open files	Priority and execution state (such as *BLOCKED*)
Semaphores, other synchronization objects	Software interrupt handling information
List of thread identifiers	Execution environment identifier
Pages of address space resident in memory; hardware cache entries	

- **Threads vs. multiple processes/execution environments**
  - switching between threads more efficient than switching between processes
    - threads
      - scheduling (deciding which thread to run next)
      - context switching (saving processor's register state, loading new register contents)
      - domain transitions
        - if new thread is member of a different execution environment
        - cache misses more severe than in-domain switching
      - experimental results
        - process switch in Unix: 1.8ms, thread switch in Topaz: 0.4 ms

Processes and Threads

Execution environment	*Thread*
Address space tables	Saved processor registers
Communication interfaces, open files	Priority and execution state (such as *BLOCKED*)
Semaphores, other synchronization objects	Software interrupt handling information
List of thread identifiers	Execution environment identifier
Pages of address space resident in memory; hardware cache entries	

- **Threads vs. multiple processes/execution environments**
  - Easy sharing of data amongst processes in one execution environment
    - no need for message passing
    - communication via shared memory
  - No protection against malevolent threads
    - one thread can access other thread's data, unless a type-safe programming language is being used

Java Thread class

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.

Processes and Threads

- **Thread Groups**
  - every thread belongs to one group, assigned at thread creation time
  - thread groups useful to shield various applications running in parallel on one Java Virtual machine
    - thread in one group may not interrupt thread in another group
      - e.g., an application may not interrupt the windowing (AWT) thread
- **Java Thread Synchronization**
  - each thread's local variables and methods are private to it
    - thread has own stack
  - thread does not have private copies of static (class) variables or object instance variables
  - mutual exclusion via monitor concept
    - abstract data type first implemented in Ada
    - in Java: `synchronized` keyword
      - any object can only be accessed through one invokation 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

- **Java Thread Synchronization**
  - threads can be blocked and woken up
    - thread awaiting a certain condition calls an object's `wait()` method
    - other thread calls `notify()` or `notifyAll()` to awake one or all blocked threads
  - example
    - worker thread discovers no requests to be processed
      - calls `wait()` on instance of Queue
    - when IO thread adds request to queue
      - calls `notify()` method of queue to wake up worker

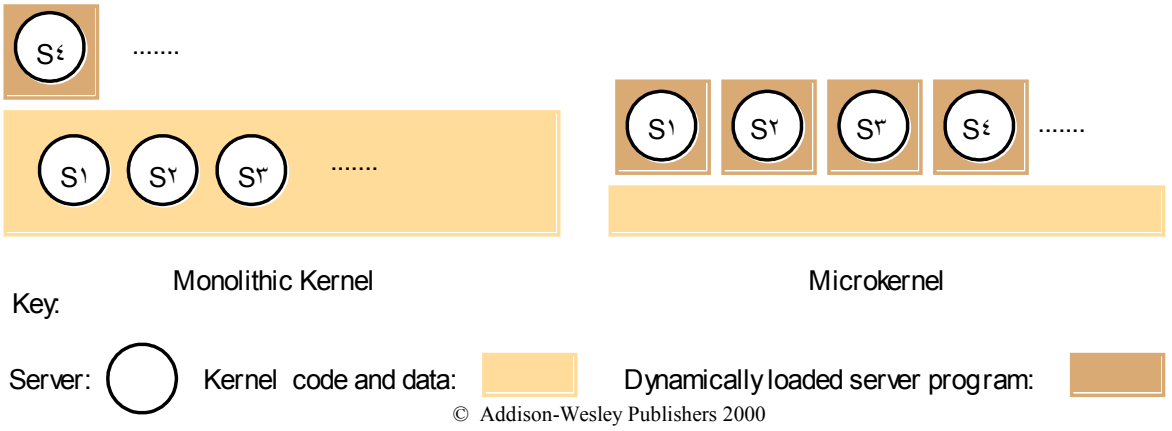
thread.join(int millisecs)
Blocks the calling thread for up to the specified time until *thread* has terminated.

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.

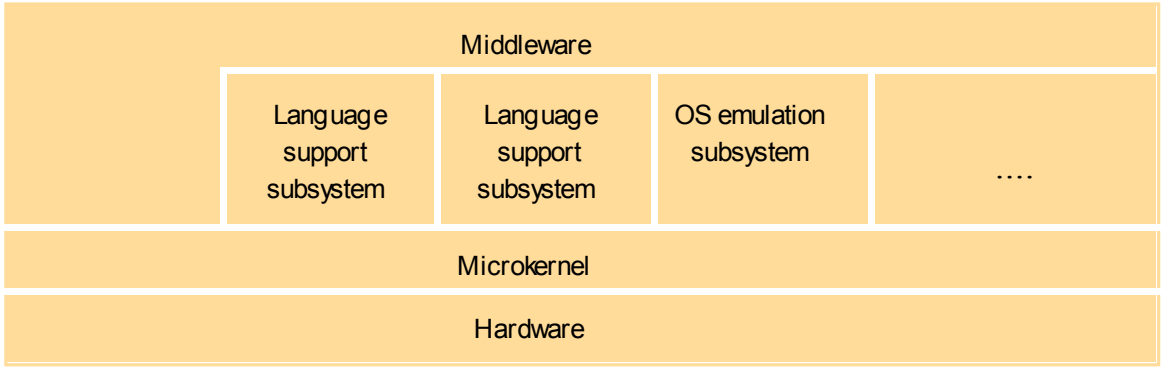
object.notify(), object.notifyAll()
Wakes, respectively, one or all of any threads that have called *wait()* on *object*.

Architecture of Networked Operating Systems



- **Monolithic Kernel vs. Microkernel**
  - goal: separation of concerns
    - e.g., separate resource management mechanisms from policies
    - example: separate context switching mechanism from policy deciding which process to schedule next
  - possible architecture: kernel performs only basic mechanisms, policies loaded dynamically by invoking services outside kernel
  - monolithic kernel
    - all essential functions implemented inside kernel
    - example: Unix
  - microkernel
    - only basic functionality in kernel, services dynamically loaded
    - servers run in user (unprivileged) mode

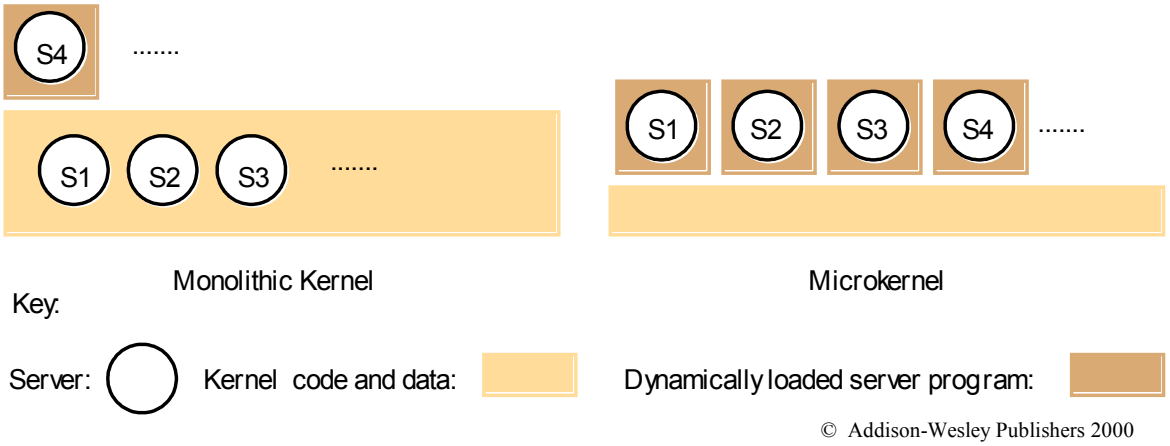
Architecture of Networked Operating Systems



The microkernel supports middleware via subsystems

- **Microkernel and Middleware**
  - microkernel a layer between hardware, services and middleware
  - for performance reasons, middleware may directly access microkernel routines
  - otherwise, access through
    - programming language APIs
    - OS emulation calls
      - e.g., Unix calls emulated on Mach distributed operating system

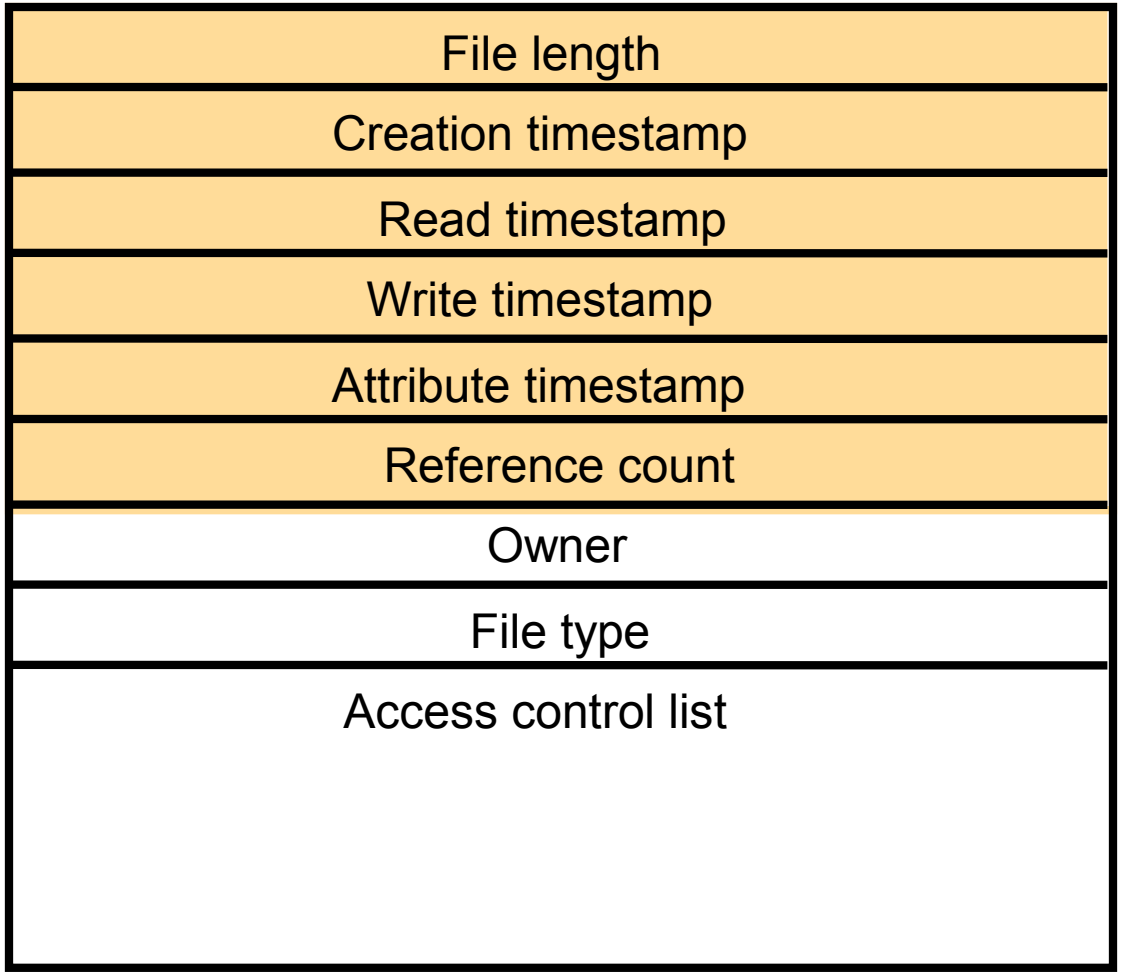
Architecture of Networked Operating Systems



- **Monolithic Kernel vs. Microkernel**
  - microkernel-based
    - advantage
      - extensibility
      - maintainability (modularity)
      - small kernel likely to be bug-free
    - disadvantage
      - invoking services involves context switches
      - essential system services executing in user mode
  - monolithic kernel
    - advantage
      - efficiency
    - disadvantage
      - all services execute in supervisor mode

File Systems

- **File System**
  - operating system interface to disk storage
- **File System Attributes (Metadata)**



Operations on Unix File System

filedes = ***open***(*name, mode*) *filedes* = ***creat***(*name, mode*)	Opens an existing file with the given *name*. Creates a new file with the given *name*. Both operations deliver a file descriptor referencing the open file. The *mode* is *read*, *write* or both.
status = ***close***(*filedes*)	Closes the open file *filedes*.
count = ***read***(*filedes, buffer, n*) *count* = ***write***(*filedes, buffer, n*)	Transfers *n* bytes from the file referenced by *filedes* to *buffer*. Transfers *n* bytes to the file referenced by *filedes* from buffer. Both operations deliver the number of bytes actually transferred and advance the read-write pointer.
pos = ***lseek***(*filedes, offset, whence*)	Moves the read-write pointer to offset (relative or absolute, depending on *whence*).
status = ***unlink***(*name*)	Removes the file *name* from the directory structure. If the file has no other names, it is deleted.
status = ***link***(*name1, name2*)	Adds a new name (*name2*) for a file (*name1*).
status = ***stat***(*name, buffer*)	Gets the file attributes for file *name* into *buffer*.

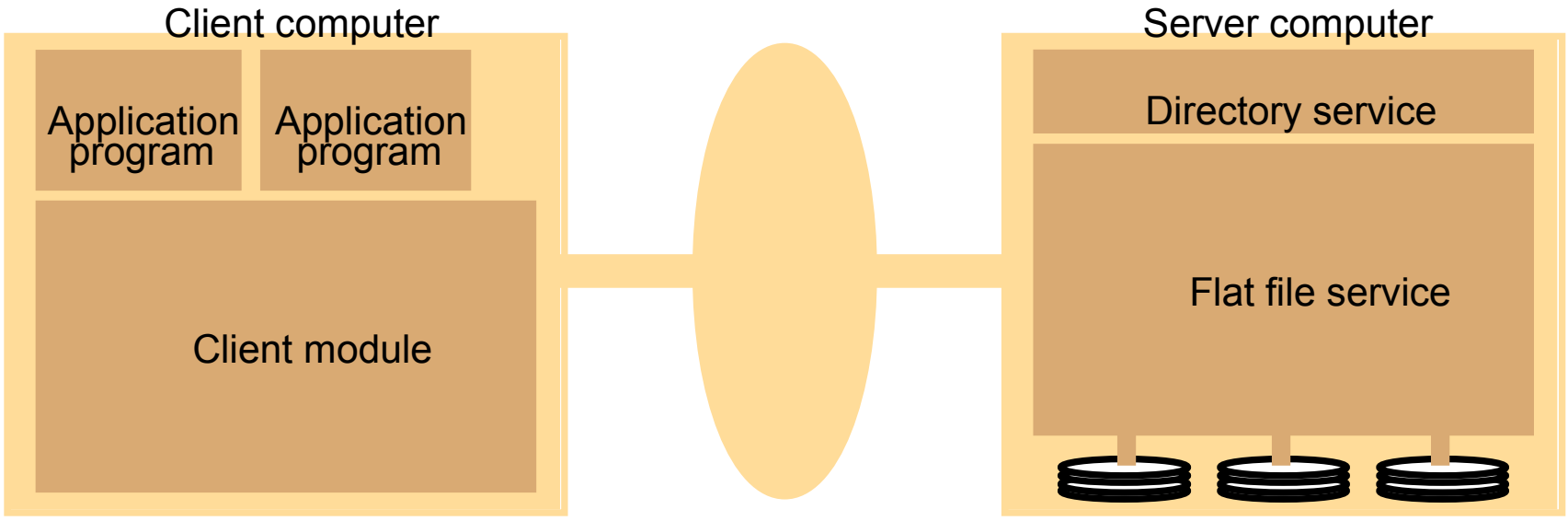
Distributed File System

- **Distributed File System**
  - file system emulating non-distributed file system behaviour on a physically distributed set of files, usually within an intranet
  - requirements
    - transparency
      - access transparency: hide distributed nature of file system by providing a single service interface for local and distributed files
        - programs working with a non-distributed file system should work without major adjustments on a distributed file system
      - location transparency: uniform, location independent name space
      - mobility transparency: file specifications will remain invariant if a file is physically moved to a different location within the dfs
      - performance transparency: load increase within normal bounds should allow a client to continue to receive satisfactory performance
      - scaling transparency: expansion by incremental growth
    - allow concurrent access
    - allow file replication
    - tolerate hardware and operating system heterogeneity

Distributed File System

- **Distributed File System**
  - requirements
    - fault tolerance: continue to provide correct service in the presence of communication or server faults
      - at-most-once semantics for file operations
      - at-least-once semantics for indempotent file operations
      - replication (stateless, so that servers can be restarted after failure)
    - consistency
      - one-copy update semantics
        - all clients see contents of file identically as if only one copy of file existed
        - if caching is used: after an update operation, no program can observe a discrepancy between data in cache and stored data
    - security
      - access control
      - client authentication
    - efficiency
      - latency of file accesses
      - scalability (e.g., with increasing number of concurrent users)

Architecture



- **Flat File Service**
  - performs file operations
  - uses “unique file identifiers” (UFIDs) to refer to files
  - flat file service interface
    - RPC-based interface for performing file operations
    - not normally used by application level programs
- **Directory Service**
  - mapping of UFIDs to “text” file names, and vice versa
- **Client Module**
  - provides API for file operations available to application program

Architecture

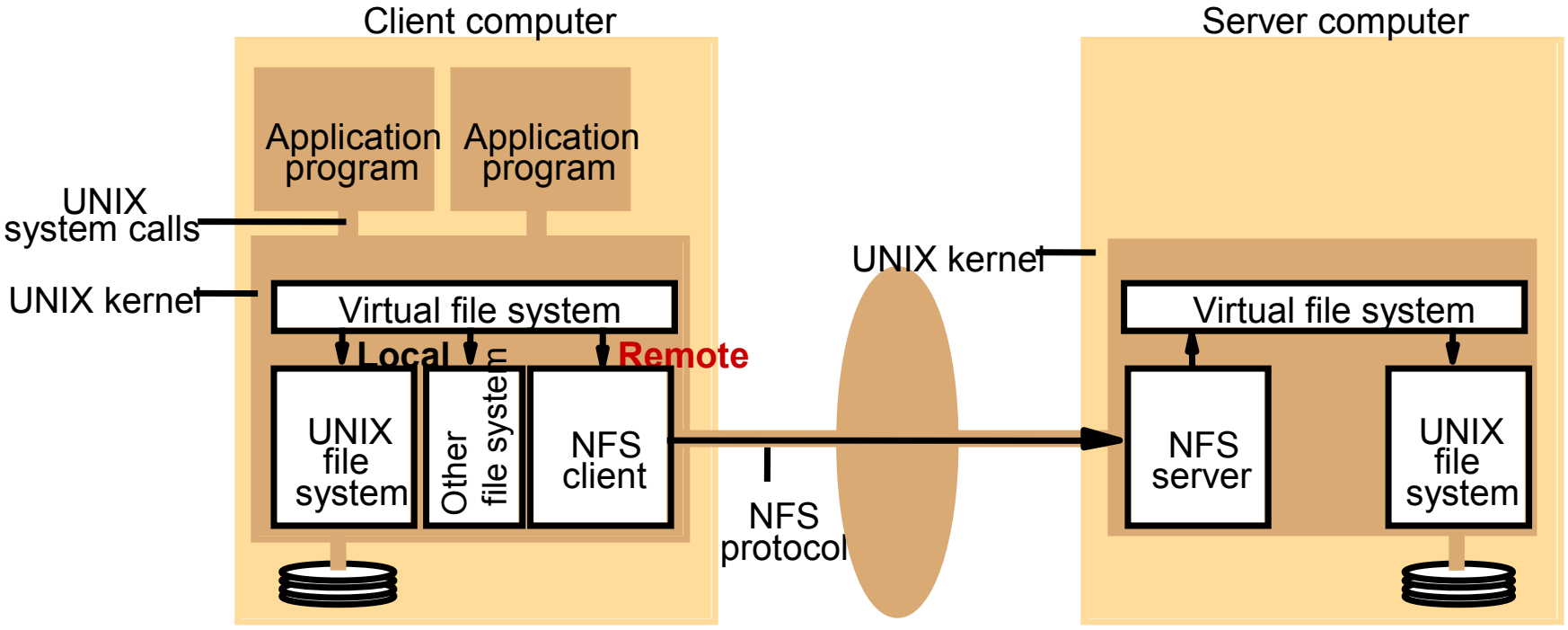
Read(FileId, i, n) -> Data — throws *BadPosition*	If $1 \leq i \leq Length(File)$: Reads a sequence of up to *n* items from a file starting at item *i* and returns it in *Data*.
Write(FileId, i, Data) — throws *BadPosition*	If $1 \leq i \leq Length(File)+1$: Writes a sequence of *Data* to a file, starting at item *i*, extending the file if necessary.
Create() -> FileId	Creates a new file of length 0 and delivers a UFID for it.
Delete(FileId)	Removes the file from the file store.
GetAttributes(FileId) -> Attr	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes (only those attributes that are not shaded in).

- **Flat File Service Interface**
  - comparison with Unix
    - every operation can be performed immediately
    - Unix maintains file pointer, reads and writes start at the file pointer location
    - advantages: fault tolerance
      - with the exception of create, all operations are indempotent
      - can be implemented as a stateless, replicated server

Architecture

- **Access Control**
  - at the server in dfs, since requests are usually transmitted via unprotected RPC calls
  - mechanisms
    - access check when mapping file name to UFID, returning cryptographic “capability” to requester who uses this for subsequent requests
    - access check at server with every file system operation
- **Hierachical File System**
  - files organized in trees
  - reference by pathname + filename
- **File Groups**
  - groups of files that can be moved between servers
  - file cannot change group membership
  - in Unix: filesystem
  - identification: must be unique in network
    - IP address of creating host
    - date of creation

SUN Network File System



- **Architecture of NFS Version 3**
  - access transparency
    - no distinction between local and remote files
    - virtual file system keeps track of locally and remotely available filesystems
    - file identifiers: file handles
      - filesystem identifier (unique number allocated at creation time)
      - i-node number
      - i-node generation number (because i-node-numbers are reused)

Selected NFS Server Operations - I -

lookup(dirfh, name) -> fh, attr	Returns file handle and attributes for the file *name* in the directory *dirfh*.
create(dirfh, name, attr) -> newfh, attr	Creates a new file name in directory *dirfh* with attributes *attr* and returns the new file handle and attributes.
remove(dirfh, name) status	Removes file name from directory *dirfh*.
getattr(fh) -> attr	Returns file attributes of file *fh*. (Similar to the UNIX *stat* system call.)
setattr(fh, attr) -> attr	Sets the attributes (mode, user id, group id, size, access time and modify time of a file). Setting the size to 0 truncates the file.
read(fh, offset, count) -> attr, data	Returns up to *count* bytes of data from a file starting at *offset*. Also returns the latest attributes of the file.
write(fh, offset, count, data) -> attr	Writes *count* bytes of data to a file starting at *offset*. Returns the attributes of the file after the write has taken place.
rename(dirfh, name, todirfh, toname) -> status	Changes the name of file *name* in directory *dirfh* to *toname* in directory to *todirfh*
link(newdirfh, newname, dirfh, name) -> status	Creates an entry *newname* in the directory *newdirfh* which refers to file *name* in the directory *dirfh*.

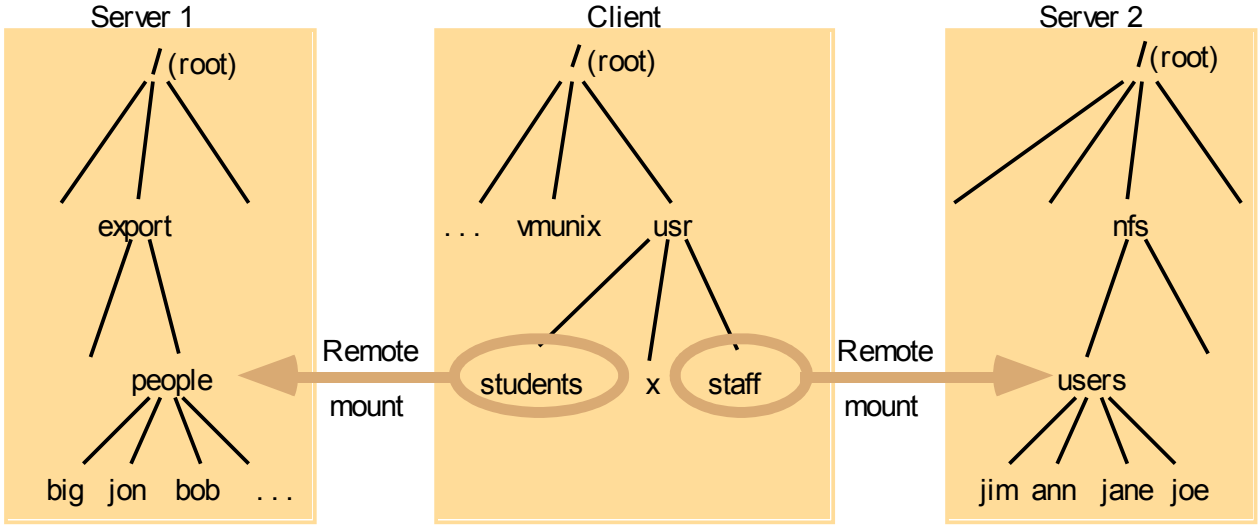
Selected NFS Server Operations - II -

symlink(newdirfh, newname, string) -> status	Creates an entry *newname* in the directory *newdirfh* of type symbolic link with the value *string*. The server does not interpret the *string* but makes a symbolic link file to hold it.
readlink(fh) -> string	Returns the string that is associated with the symbolic link file identified by *fh*.
mkdir(dirfh, name, attr) -> newfh, attr	Creates a new directory *name* with attributes *attr* and returns the new file handle and attributes.
rmdir(dirfh, name) -> status	Removes the empty directory *name* from the parent directory *dirfh*. Fails if the directory is not empty.
readdir(dirfh, cookie, count) -> entries	Returns up to *count* bytes of directory entries from the directory *dirfh*. Each entry contains a file name, a file handle, and an opaque pointer to the next directory entry, called a *cookie*. The *cookie* is used in subsequent *readdir* calls to start reading from the following entry. If the value of *cookie* is 0, reads from the first entry in the directory.
statfs(fh) -> fsstats	Returns file system information (such as block size, number of free blocks and so on) for the file system containing a file *fh*.

SUN Network File System

- **Access Control/Authentication**
  - NFS requests transmitted via Remote Procedure Calls (RPCs)
    - clients send authentication information (user / group IDs)
    - checked against access permissions in file attributes
  - potential security loophole
    - any client may address RPC requests to server providing another client's identification information
    - introduction of security mechanisms in NFS
      - DES encryption of user identification information
      - Kerberos authentication

SUN Network File System



- **Mounting of Filesystems**
  - making remote file systems available to a local client, specifying remote host name and pathname
  - mount protocol (RPC-based)
    - returns file handle for directory name given in request
    - location (IP address and port number) and file handle are passed to Virtual File system and NFS client
  - hard-mounted (mostly used in practice)
    - client suspended until operation completed
    - application may not terminate gracefully in failure situations
  - soft-mounted
    - error message returned after small number of retries

SUN Network File System

- **Caching in server and client indispensable to achieve necessary perfomance**
  - Server caching
    - disk caching as in non-networked file systems
    - read operations: unproblematic
    - write operations: consistency problems
      - write-through caching
        - store updated data in cache and on disk before sending reply to client
        - relatively inefficient if frequent write operations occur
      - commit operation
        - caching only in cache memory
        - write back to disk only when commit operation for file received

SUN Network File System

- **Caching indispensable to achieve necessary perfomance**
  - Client caching
    - caching of `read`, `write`, `getattr`, `lookup` and `readdir` operations
    - potential inconsistency: the data cached in client may not be identical to the same data stored on the server
    - time-stamp based scheme used in polling server about feshness of a data object (presumption of synchronized global time, e.g., through NTP)
      - Tc: time cache was last validated
      - $Tm_{client/server}$: time when block was last modified at the server as recorded by client/server
      - t: freshness interval
    - freshness condition
      - $(T\text{-}Tc < t) \vee (Tm_{client} = TM_{server})$
        - if $(T\text{-}Tc < t)$ (can be determined without server access), then entry presumed to be valid
        - if not $(T\text{-}Tc < t)$, then TM_{server} needs to be obtained by a `getattr` call
        - if $Tm_{client} = TM_{server}$, then data presumed valid, else obtain data from server and update Tm_{client}
    - note: scheme does not guarantee consistency, since recent updates may be invisible, one copy update semantics only approximated

SUN Network File System

- **Caching indispensable to achieve necessary perfomance**
  - Client caching
    - perfomance factors: how to reduce server traffic, in particular for `getattr`
      - receipt of TM_{server}, then update all Tm_{client} values related to data object derived from the same file
      - piggyback current attribute values on results of every file operation
      - adaptive algorithm for t
        - t too short: many server requests
        - t too large: increased chance of inconsistencies
        - typical values: 3 to 30 secs for files, 30 to 60 secs for directories
        - in Solaris, t is adjusted according to frequency of file updates

SUN Network File System

- **Caching indispensable to achieve necessary perfomance**
  - Client caching - `write` operations
    - mark modified cache page as “dirty” and schedule page to be flushed to server (asynchronously)
      - flush happens with closing of file, when `sync` is issued,
      - or when asynchronous block input-output (bio) daemon is used and active
        - when `read`, then read-ahead: when read occurs, bio daemon sends next file block
        - when `write`, then bio daemon will send block asynchronously to server
      - bio daemons: performance measure reducing probability that client blocks waiting for
        * `read` operations to return, or
        * `write` operations to be committed at the server

Andrew File System

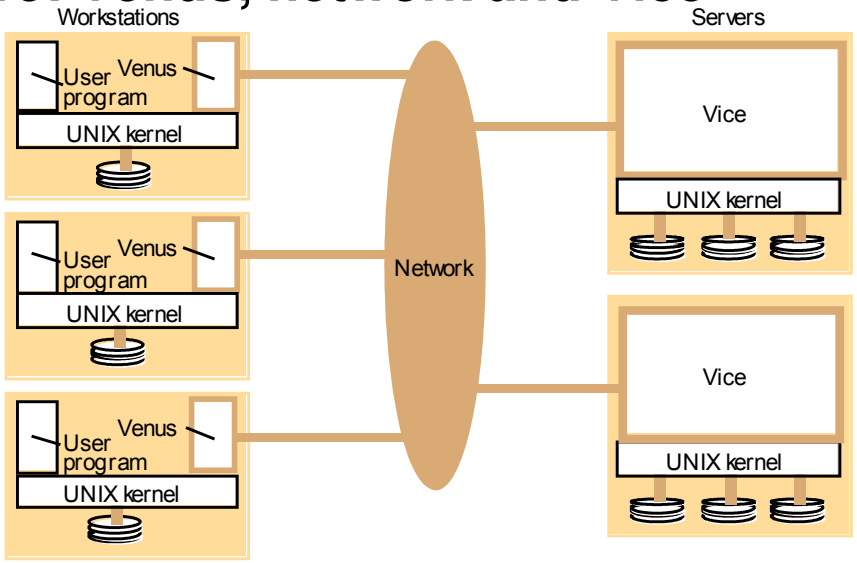
- **Andrew File System (AFS)**
  - started as a joint effort of Carnegie Mellon University and IBM
  - today basis for DCE/DFS: the distributed file system included in the Open Software Foundations's Distributed Computing Environment
  - some UNIX file system usage observations, as pertaining to caching
    - infrequently updated shared files and local user files will remain valid for long periods of time (the latter because they are being updated on owners workstations)
    - allocate large local disk cache, e.g., 100 MByte, that can provide a large enough working set for all files of one user such that the file is still in this cache when used next time
    - assumptions about typical file accesses (based on empirical evidence)
      - usually small files, less than 10 Kbytes
      - reads much more common than writes (appr. 6:1)
      - usually sequential access, random access not frequently found
      - user-locality: most files are used by only one user
      - burstiness of file references: once file has been used, it will be used in the nearer future with high probability

Andrew File System

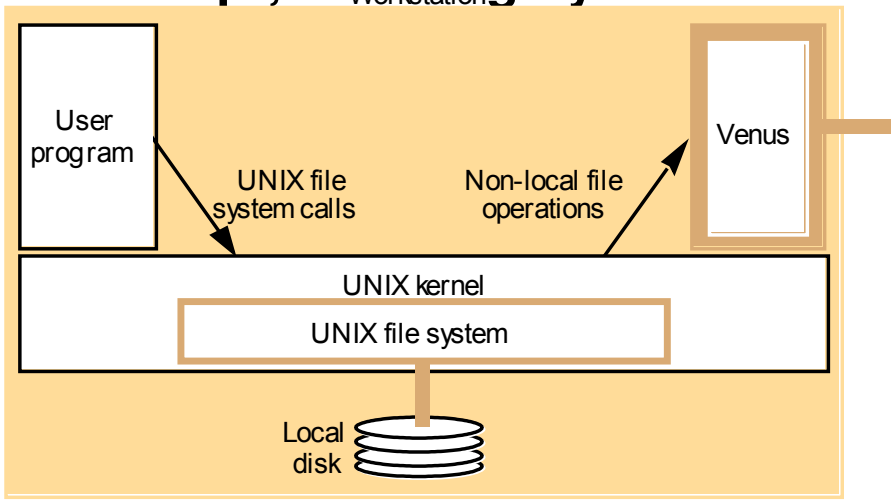
- **Andrew File System (AFS)**
  - design decisions for AFS
    - whole-file serving: entire contents of directories and files transfered from server to client (AFS-3: in chunks of 64 Kbytes)
    - whole file caching: when file transfered to client it will be stored on that client's local disk

Andrew File System

- **AFS architecture: Venus, network and Vice**



- **AFS system call intercept, handling by Venus**



Andrew File System

- **Implementation of file system calls - callbacks and callback promises**

User process	*UNIX kernel*	*Venus*	*Net*	*Vice*
open(FileName, mode)	If *FileName* refers to a file in shared file space pass the request to Venus.	Check list of files in local cache. If not present or there is no valid ***callback promise*** send a request for the file to the Vice server that is custodian of the volume containing the file.	→	Transfer a copy of the file and a ***callback promise*** to the workstation. Log the callback promise.
	Open the local file and return the file descriptor to the application.	Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.	←	
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy.			
close(FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice server that is the custodian of the file.	→	Replace the file contents and send a ***callback*** to all other clients holding *callback promises* on the file.

Andrew File System

- **Callback mechanism**
  - ensures that cached copies of files are updated when another client performs a close operation on that file
  - callback promise
    - token stored with cached file
    - status: *valid* or *cancelled*
  - when server performs request to update file (e.g., following a `close`), then it sends callback to all Venus processes to which it has sent callback promise
    - RPC from server to Venus process
    - Venus process sets callback promise for local copy to *cancelled*
  - Venus handling an `open`
    - check whether local copy of file has *valid* callback promise
    - if *canceled*, fresh copy must be fetched from Vice server

Andrew File System

- **Callback mechanism**
  - Restart of workstation after failure
    - retain as many locally cached files as possible, but callbacks may have been missed
    - Venus sends cache validation request to the Vice server
      - contains file modification timestamp
      - if timestamp is current, server sends *valid* and callback promise is reinstantiated with *valid*
      - if timestamp not current, server sends *cancelled*
  - Problem: communication link failures
    - callback must be renewed with above protocol before new `open` if a time T has lapsed since file was cached or callback promise was last validated
  - Scalability
    - AFS callback mechanism scales well with increasing number of users
      - communication only when file has been updated
      - in NFS timestamp approach: for each open
    - since majority of files not accessed concurrently, and reads more frequent than writes, callback mechanism performs better

Andrew File System

- **File Update Semantics**
  - to ensure strict one-copy update semantics: modification of cached file must be propagated to any other client caching this file before any client can access this file
    - rather inefficient
  - callback mechanism is an approximation of one-copy semantics
  - guarantees of currency for files in AFS (version 1)
    - after successful open: latest(F, S)
      - current value of file F at client C is the same as the value at server S
    - after a failed open/close: failure(S)
      - open close not performed at server
    - after successful close: updated(F, S)
      - client's value of F has been successfully propagated to S

Andrew File System

- **File Update Semantics in AFS version 2**
  - Vice keeps callback state information about Venus clients: which clients have received callback promises for which files
  - lists retained over server failures
  - when callback message is lost due to communication link failure, an old version of a file may be opened after it has been updated by another client
  - limited by time T after which client validates callback promise (typically, T=10 minutes)
  - currency guarantees
    - after successful open:
      - latest(F, S, 0)
        - copy of F as seen by client is no more than 0 seconds out of date
      - or (lostCallback(S, T) and inCache(F) and latest(F, S, T))
        - callback message has been lost in the last T time units,
        - the file F was in the cache before open was attempted,
        - and copy is no more than T time units out of date

Andrew File System

- **Cache Consistency and Concurrency Control**
  - AFS does not control concurrent updates of files, this is left up to the application
    - deliberate decision, not to support distributed database system techniques, due to overhead this causes
  - cache consistency only on open and close operations
    - once file is opened, modifications of file are possible without knowledge of other processes' operations on the file
    - any close replaces current version on server
      - all but the update resulting from last close operation processed at server will be lost, without warning
    - application programs on same server share same cached copy of file, hence using standard UNIX block-by-block update semantics
  - although update semantics not identical to local UNIX file system, sufficiently close so that it works well in practice

Enhancements

- **Spritely NFS**
  - goal: achieve precise one-copy update semantics
  - abolishes stateless nature of NFS -> vulnerability in case of server crashes
  - introduces open and close operations
    - open must be invoked when application wishes to access file on server, parameters:
      - modes: read, write, read/write
      - number of local processes that currently have the file open for read and write
    - close
      - updated counts of processes
  - server records counts in open files table, together with IP address and port number of client
  - when server receives open: checks file table for other clients that have the same file open
    - if open specifies write,
      - request fails if any other client has file open for writing,
      - otherwise other read clients are instructed to invalidate local cache copy
    - if open specifies read,
      - sends callback to other write clients forcing them to modify their caching strategy to *write-through*
      - causes all other read clients to read from server and stop caching

Enhancements

- **WebNFS**
  - access to files in WANs by direct interaction with remote NFS servers
  - permits partial file accesses
    - http or ftp would require entire files to be transmitted, or special software at the server end to provide only the data needed
  - access to “published” files through public file handle
  - for access via path name on server, usage of lookup requests
  - reading a (portion of) a file requires
    - TCP connection to server
    - lookup RPC
    - read RPC
- **NFS version 4**
  - similarly for WANs
  - usage of callbacks and leases
  - recovery from server faults through transparent moving of file systems from one server to another
  - usage of proxy servers to increase scalability