Computer Science 425
Distributed Systems

CS 425 / CSE 424 / ECE 428

Fall 2012

Indranil Gupta (Indy)

December 6, 2012

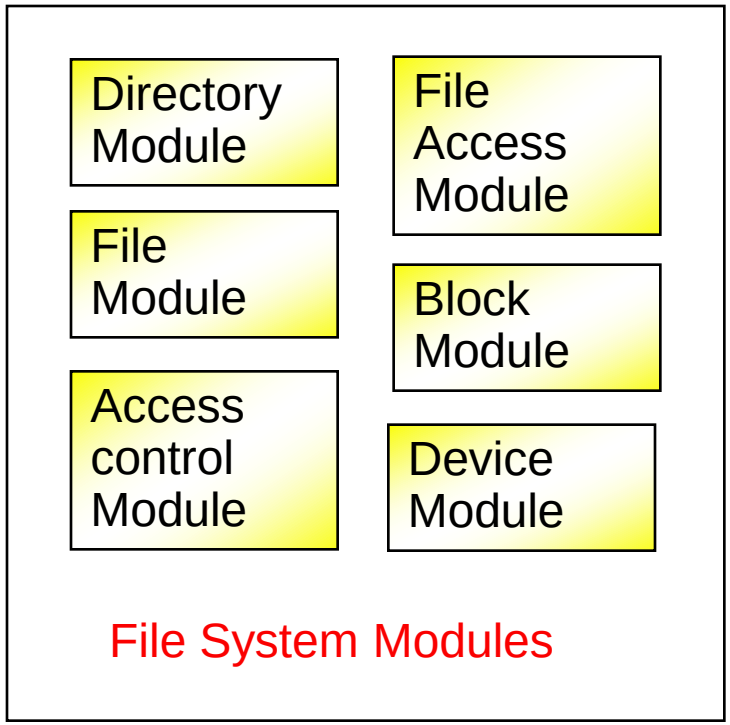
Lecture 28

Distributed File Systems and Distributed Shared Memory

Chapter 12 (relevant parts), Sections 6.5, Chapter 6 from Tanenbaum textbook

File Attributes & System Modules

- length
- creation timestamp
- read timestamp
- write timestamp
- attribute timestamp
- reference count
- file type
- ownership
- access control list



UNIX File System Operations

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 links to it, it is deleted from disk.
status = link(name1, name2)	Creates a new link (*name2*) for a file (*name1*).
status = stat(name, buffer)	Gets the file attributes for file *name* into *buffer*.

Distributed File System (DFS) Requirements

- **Transparency** - server-side changes should be invisible to the client-side.
  - *Access transparency*: A single set of operations is provided for access to local/remote files.
  - *Location Transparency:* All client processes see a uniform file name space.
  - *Migration Transparency*: When files are moved from one server to another, users should not see it.
  - *Performance Transparency*
  - *Scaling Transparency*
- **File Replication**
  - A file may be represented by several copies for read/write efficiency and fault tolerance.
- **Concurrent File Updates**
  - Changes to a file by one client should not interfere with the operation of other clients simultaneously accessing the same file.

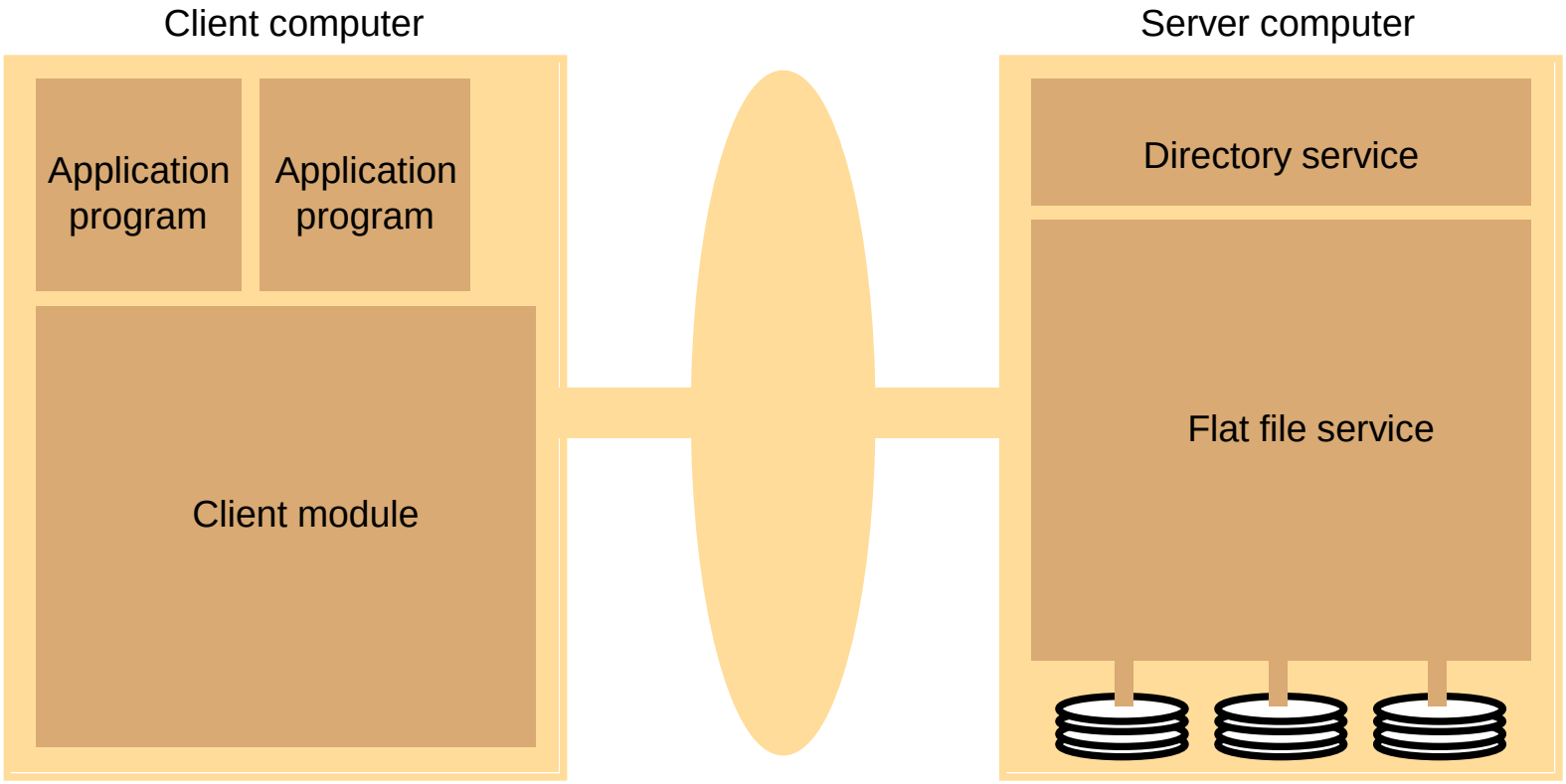
DFS Requirements (2)

- **Concurrent File Updates**
  - **One-copy update semantics: the file contents seen by all of the clients accessing or updating a given file are those they would see if only a single copy of the file existed.**
- **Fault Tolerance**
  - **At most once invocation semantics.**
  - **At least once semantics. OK for a server protocol designed for idempotent operations (i.e., duplicated requests do not result in invalid updates to files)**
- **Security**
  - **Access Control list = per object, list of allowed users and access allowed to each**
  - **Capability list = per user, list of objects allowed to access and type of access allowed (could be different for each (user,obj))**
  - **User Authentication: need to authenticate requesting clients so that access control at the server is based on correct user identifiers.**
- **Efficiency**
  - **Whole file vs. block transfer**

Basic File Service Model

- E.g., SUN NFS (Network File System) and AFS (Andrew File System)
- An abstract model :
  - Flat file service
    - implements create, delete, read, write, get attribute, set attribute and access control operations.
  - Directory service: is itself a client of (i.e., uses) flat file service.
    - Creates and updates directories (hierarchical file structures) and provides mappings between user names of files and the unique file ids in the flat file structure.
  - Client service/module: A client of directory and flat file services
    - Runs in each client's computer, integrating and expanding flat file and directory services to provide a unified API (e.g., the full set of UNIX file operations).
    - Holds information about the locations of the flat file server and directory server processes.

File Service Architecture



Flat File Service Operations

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).

(1) Repeatable operation: No read-write pointer. Except for Create and delete, the operations are idempotent, allowing the use of at least once RPC semantics.

(2) Stateless servers: No file descriptors. Stateless servers can be restarted after a failure and resume operation without the need to restore any state.

In contrast, the UNIX file operations are neither idempotent nor consistent.

Access Control

- In UNIX, the user's access rights are checked against the access mode requested in the open call and the file is opened only if the user has the necessary rights.
- In DFS, a user identity has to be passed with requests – server first authenticates the user.
  - An access check is made whenever a file name is converted to a UFID (unique file id), and the results are encoded in the form of a capability which is returned to the client for future access.
    - » Capability = per user, list of objects allowed to access and type of access allowed (could be broken up per (user,obj))

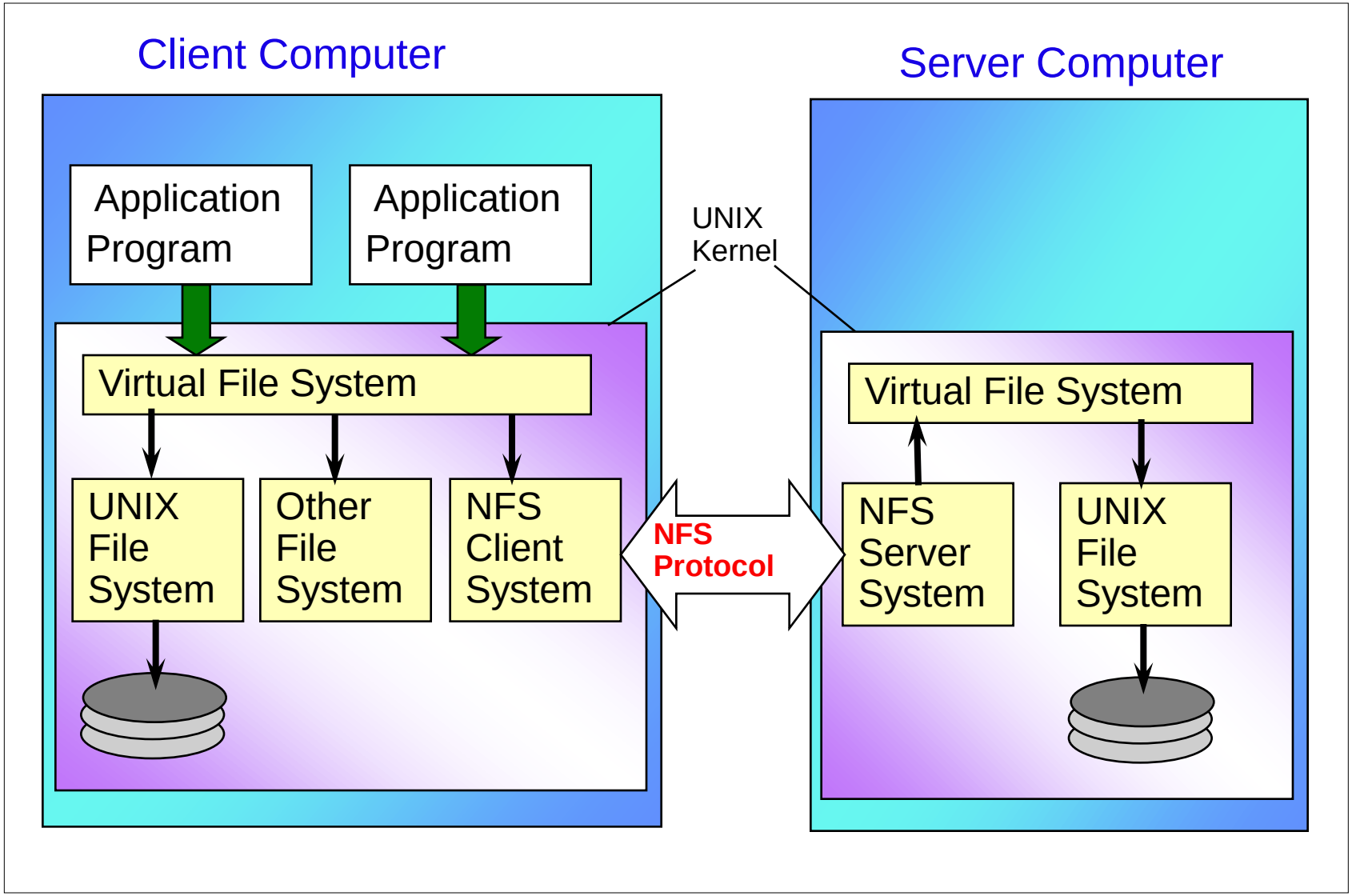
Directory Service Operations

Lookup(Dir, Name) -> FileId — throws *NotFound*	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, File) — throws *NameDuplicate*	If *Name* is not in the directory, adds (*Name*, *File*) to the directory and updates the file's attribute record. If *Name* is already in the directory: throws an exception.
UnName(Dir, Name) — throws *NotFound*	If *Name* is in the directory: the entry containing *Name* is removed from the directory. If *Name* is not in the directory: throws an exception.
GetNames(Dir, Pattern)->NameSeq	Returns all the text names in the directory that match the regular expression *Pattern*. Like *grep*.

(1) Hierarchical file system: The client module provides a function that gets the UFID of a file given its pathname. The function interprets the pathname starting from the root, using *Lookup* to obtain the UFID of each directory in the path.

(2) Each server may hold several *file groups,* each of which is a collection of files located on the server. A file group identifier consists of IP address + date, and allows (i) file groups to migrate across servers, and (ii) clients to access file groups.

Network File System (NFS)



(If you're interested, more NFS slides are in the Backup Slides section of this slide deck)

NFS Architecture -- VFS

- **Virtual file system module**
  - **Translates between NFS file identifiers and other file systems's (e.g., UNIX) identifiers.**
    - » **The NFS file identifiers are called *file handle*s.**
    - » **File handle = *Filesystem/file group* identifier + *i-node* number of file + i-node generation number.**
  - **Keeps track of filesystems (i.e., NFS file groups, different from a "file system") that are available locally and remotely.**
    - » **The client obtains the first file handle for a remote filesystem when it first *mounts* the filesystem. File handles are passed from server to client in the results of lookup, create, and mkdir operation.**
  - **Distinguishes between local and remote files.**
    - » **VFS keeps one VFS structure for each mounted filesystem and one v-node per open file.**
      - **A VFS structure relates a remote filesystem to the local directory on which it is mounted.**
      - **A v-node contains an indicator to show whether a file is local or remote. If the file is local, it contains a reference to the i-node; otherwise if the file is remote, it contains the file handle of the remote file.**

Server Caching

- File pages, directories and file attributes that have been read from the disk are retained in a *main memory buffer cache.*
- *Read-ahead* anticipates read accesses and fetches the pages following those that have most recently been read.
- In *delayed-write*, when a page has been altered, its new contents are written back to the disk only when the buffered page is required for another client.
  - In comparison, Unix *sync* operation writes pages to disk every 30 seconds
- In *write-through,* data in write operations is stored in the memory cache at the server immediately and written to disk before a reply is sent to the client.
  - *Better strategy to ensure data integrity even when server crashes occur. More expensive.*

Client Caching

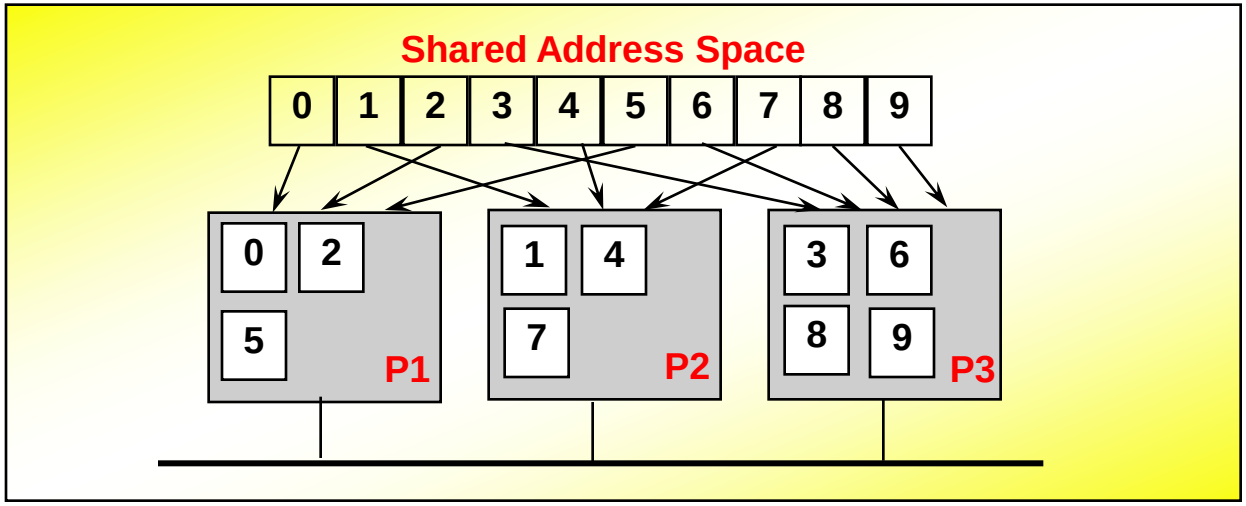
- **A timestamp-based method is used to validate cached blocks before they are used.**
- **Each data item in the cache is tagged with**
  - ***Tc*: the time when the cache entry was last validated.**
  - ***Tm*: the time when the block was last modified at the server.**
  - **A cache entry at time T is valid if**

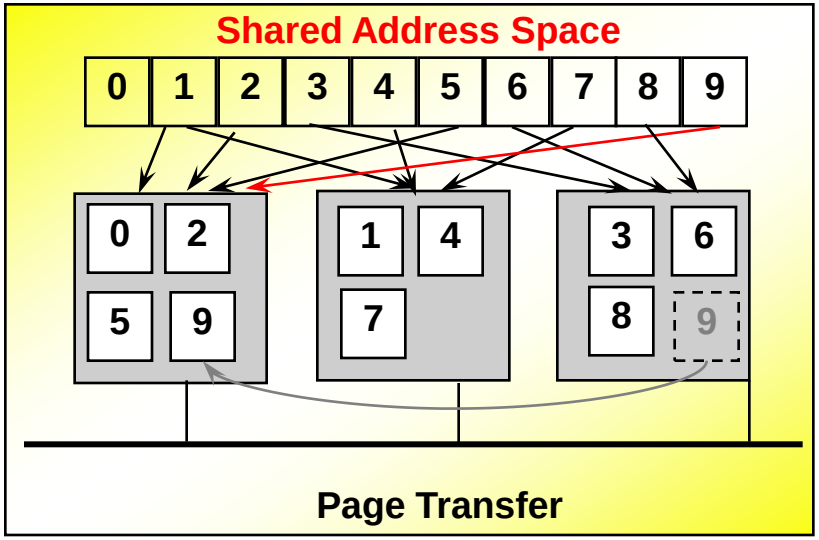
    $(T-Tc < t)$ or $(Tm_{client} = Tm_{server})$.
  - ***t=freshness interval***
    - » ***Compromise between consistency and efficiency***
    - » ***Sun Solaris: t is set adaptively between 3-30 seconds for files, 30-60 seconds for directories***

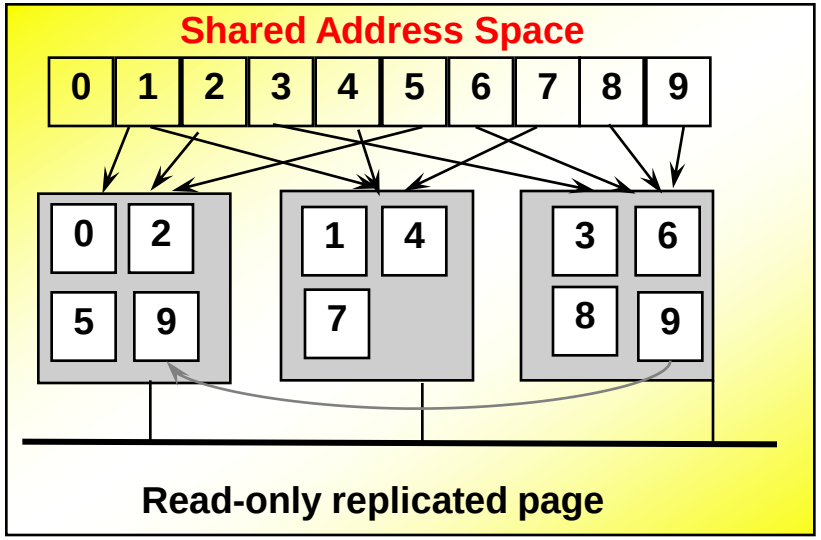
Client Caching (Cont'd)

- When a cache entry is read, a validity check is performed.
  - If the first half of validity condition (previous slide) is true, the the second half need not be evaluated.
  - If the first half is not true, Tm_{server} is obtained (via *getattr()* to server) and compared against Tm_{client}
- When a cached page (not the whole file) is modified, it is marked as dirty and scheduled to be flushed to the server.
  - Modified pages are flushed when the file is closed or a *sync* occurs at the client.
- Does not guarantee one-copy update semantics.
- More details in textbook

Distributed Shared Memory

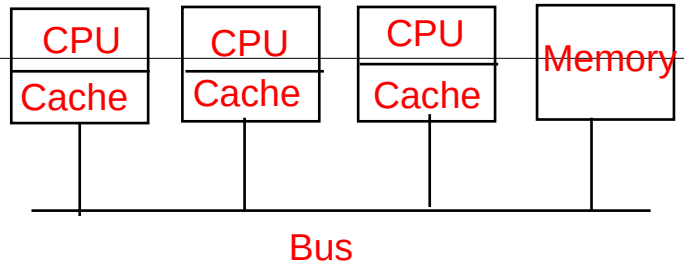






Shared Memory vs. Message Passing

- In a *multiprocessor*, two or more processors share a common main memory. Any process on a processor can read/write any word in the shared memory. All communication through a bus.
  - E.g., Cray supercomputer
  - Called Shared Memory
- In a *multicomputer*, each processor has its own private memory. All communication using a network.
  - E.g., CSIL PC cluster
  - Easier to build: One can take a large number of single-board computers, each containing a processor, memory, and a network interface, and connect them together. (called COTS="Components off the shelf")
  - Uses Message passing
- Message passing can be implemented over shared memory.
- Shared memory can be implemented over message passing.
- *Let's look at shared memory by itself.*



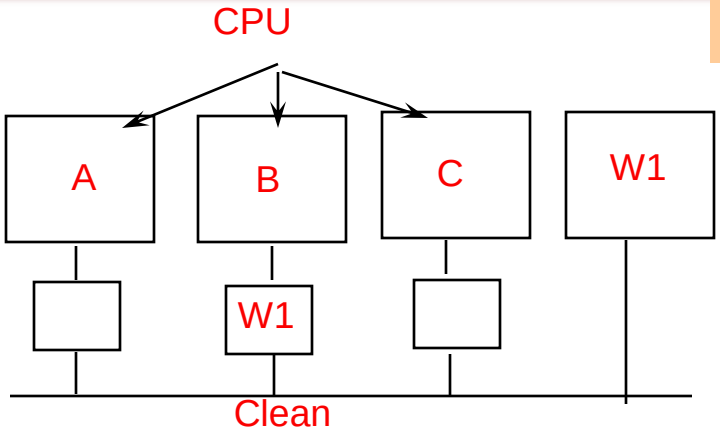
Cache Consistency – Write Through

Event	Action taken by a cache in response to its own operation	Action taken by other caches in response (to a remote operation)
Read hit	Fetch data from local cache	(no action)
Read miss	Fetch data from memory and store in cache	(no action)
Write miss	Update data in memory and store in cache	Invalidate cache entry
Write hit	Update memory and cache	Invalidate cache entry

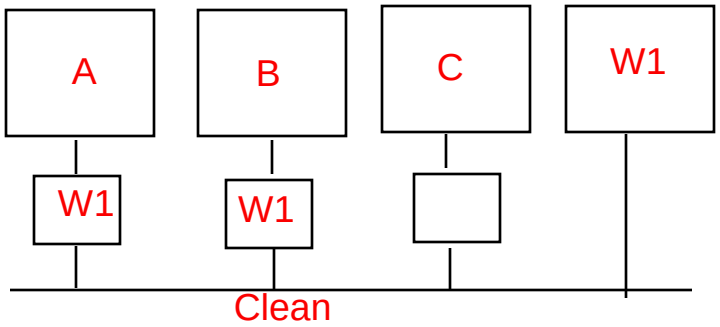
All the other caches see the write (because they are snooping on the bus) and check to see if they are also holding the word being modified. If so, they invalidate the cache entries.

Cache Consistency – Write Once

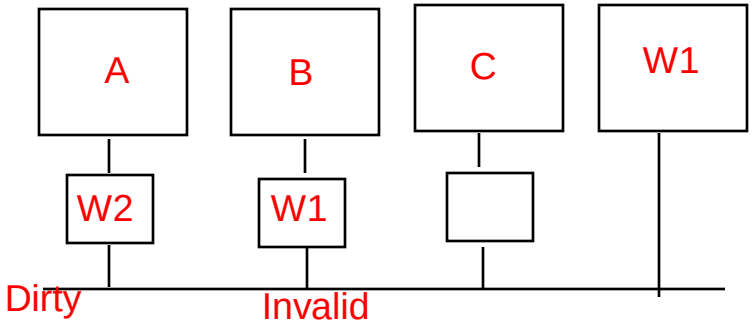
•For write, at most one CPU has valid access



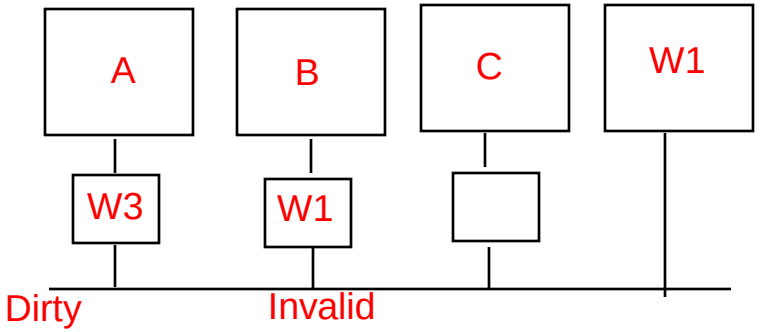
Initially both the memory and B have an updated entry of word W.



A reads word W and gets W1. B does not respond but the memory does

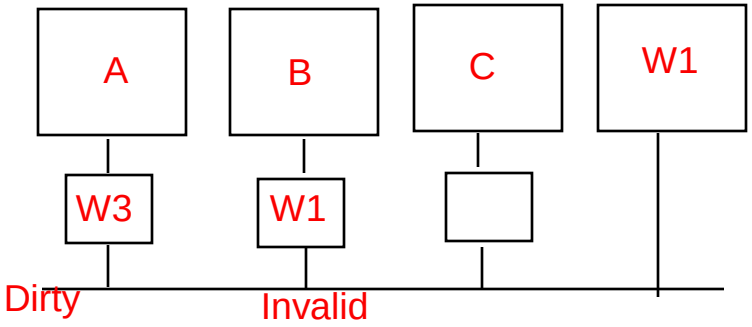


A writes a value W2. B snoops on the bus, and invalidates its entry. A′s copy is marked as dirty.

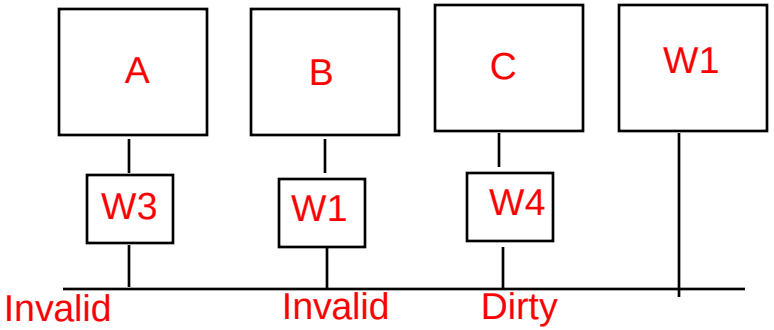


A writes W again. This and subsequent writes by A are done locally, without any bus traffic.

Cache Consistency – Write Once



A writes a value W3. No bus traffic is incurred



C writes W. A sees the request by snooping on the bus, asserts a signal that inhibits memory from responding, provides the values. A invalidates it own entry.
C now has the only valid copy.

The cache consistency protocol is built upon the notion of snooping and built into the memory management unit (MMU).
All above mechanisms are implemented in hardware for efficiency.

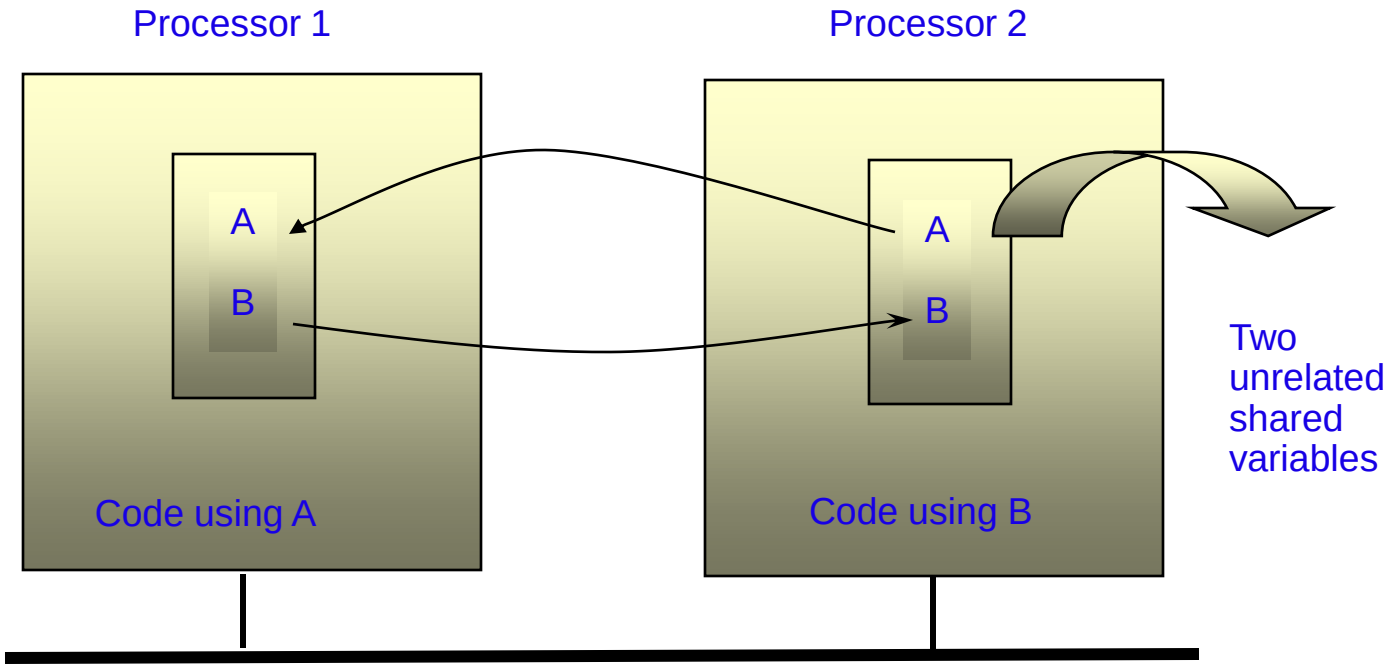
The above shared memory can be implemented using message passing instead of the bus.

Granularity of Chunks

- When a processor references a word that is absent, it causes a page fault.
- On a page fault,
  - the missing page is just brought in from a remote processor.
  - A *region* of 2, 4, or 8 pages including the missing page may also be brought in.
    - » Locality of reference: if a processor has referenced one word on a page, it is likely to reference other neighboring words in the near future.
- **Region size**
  - Small => too many page transfers
  - Large => *False sharing*
  - Above tradeoff also applies to page size

False Sharing

Page consists of two variables A and B



Occurs because: Page size > locality of reference
Unrelated variables in a region cause large number of pages transfers
Large page sizes => more pairs of unrelated variables

Achieving Sequential Consistency

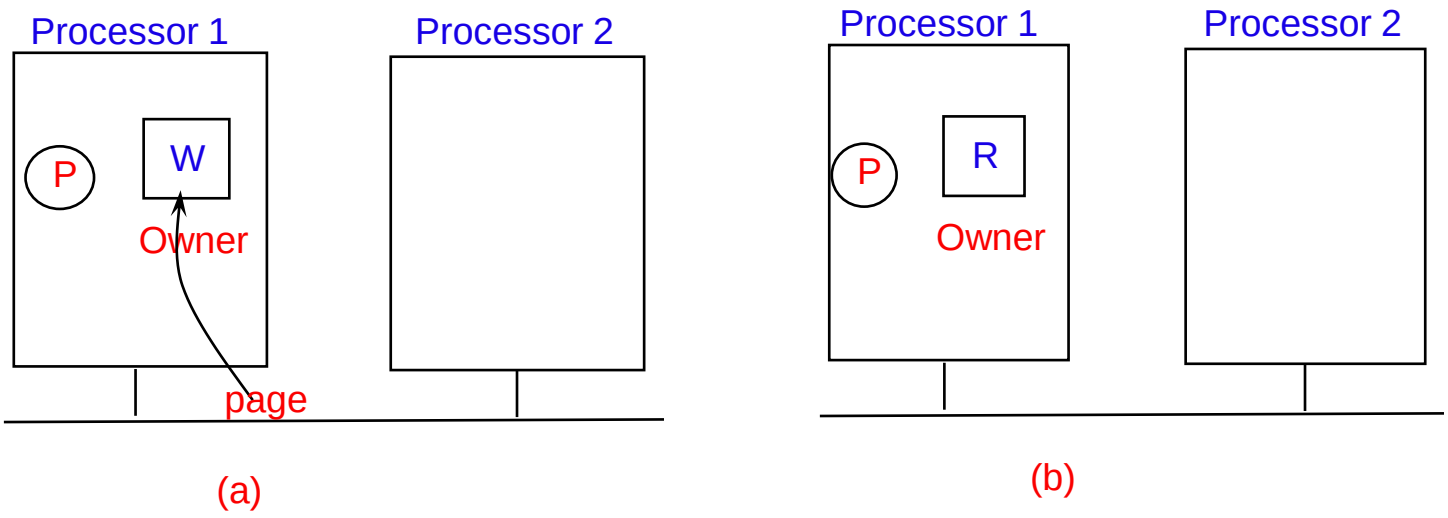
- **Achieving consistency is not an issue if**
  - **Pages are not replicated, or…**
  - **Only read-only pages are replicated.**
- **But don't want to compromise performance.**
- **Two approaches are taken in DSM**
  - ***Update*: the write is allowed to take place locally, but the address of the modified word and its new value are broadcast to all the other processors. Each processor holding the word copies the new value, i.e., updates its local value.**
  - ***Invalidate*: The address of the modified word is broadcast, but the new value is not. Other processors invalidate their copies. (Similar to example in first few slides for multiprocessor)**

  Page-based DSM systems typically use an invalidate protocol instead of an update protocol. ? [Why?]

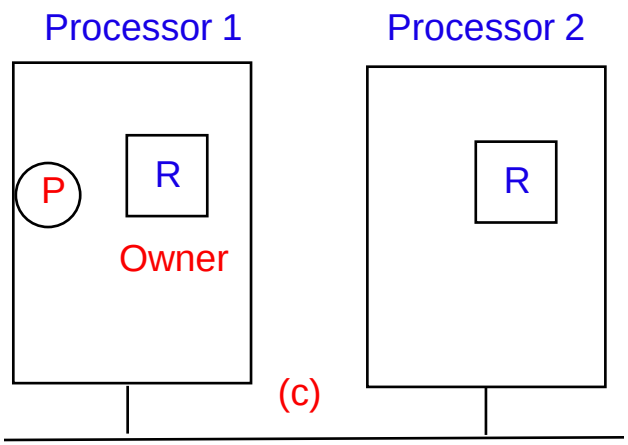
Invalidation Protocol to Achieve Consistency

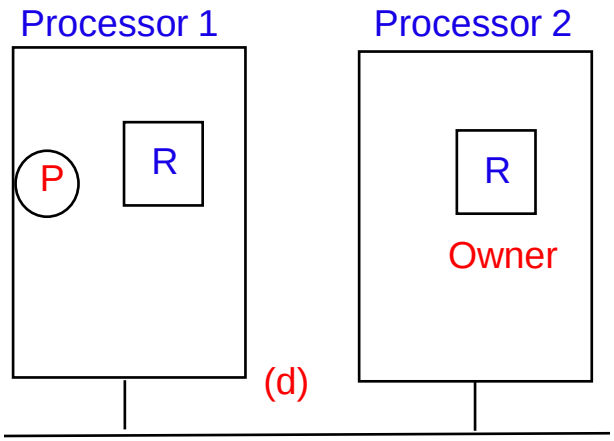
- **Each page is either in R or W state.**
  - **When a page is in W state, only one copy exists, located at one processor (called current "owner") in read-write mode.**
  - **When a page is in R state, the current/latest owner has a copy (mapped read-only), but other processors may have copies.**

Suppose Processor 1 is attempting a read: Different scenarios

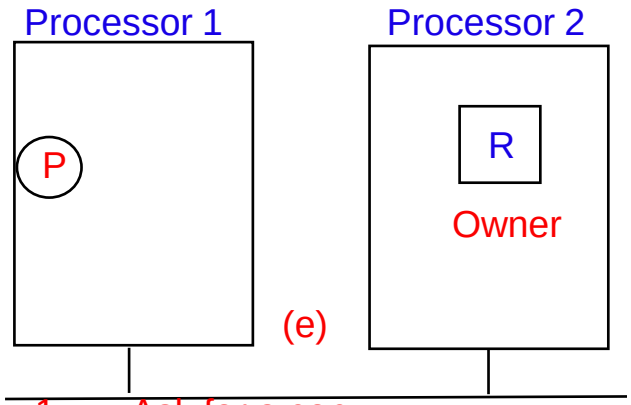


Invalidation Protocol (Read)

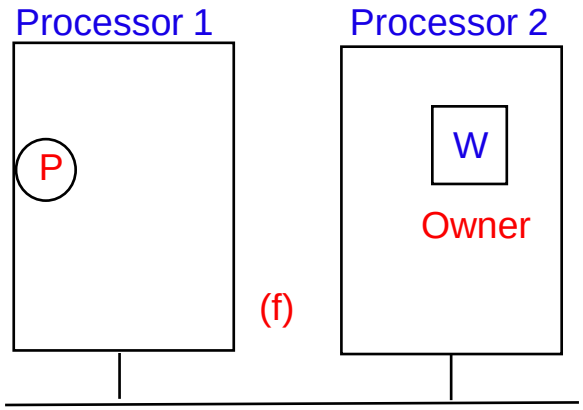




In the first 4 cases, the page is mapped into its address space, and no trap occurs.



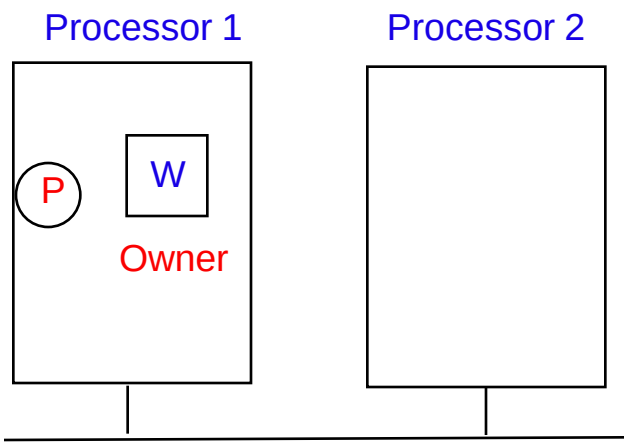
1. Ask for a copy
2. Mark page as R
3. Do read

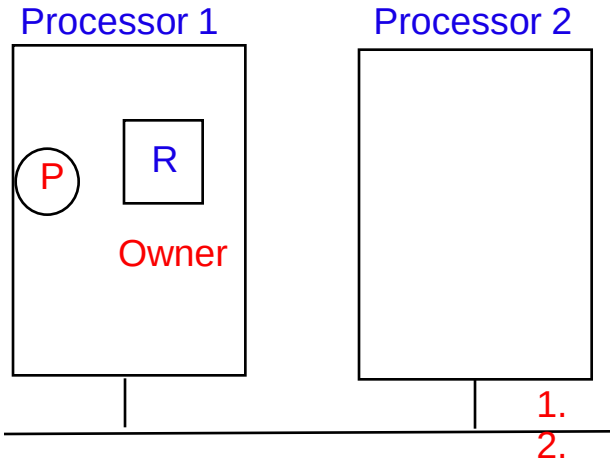


1. Ask P2 to degrade its copy to R
2. Ask for a copy
3. Mark page as R
4. Do read

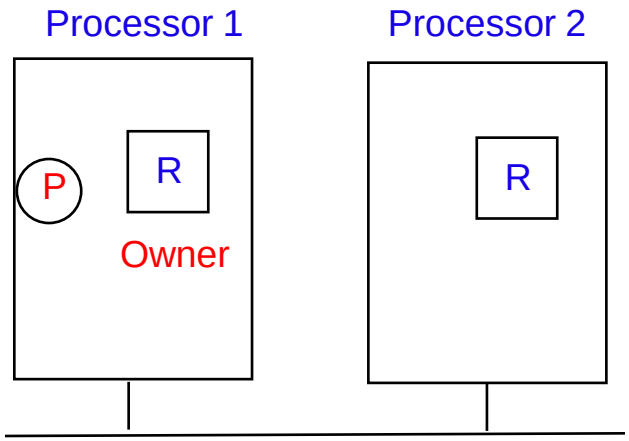
Invalidation Protocol (Write)

Suppose Processor 1 is attempting a write: Different scenarios

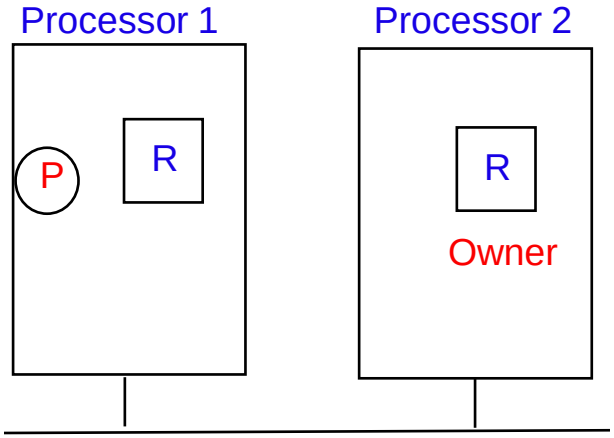




1. Mark page as W
2. Do write



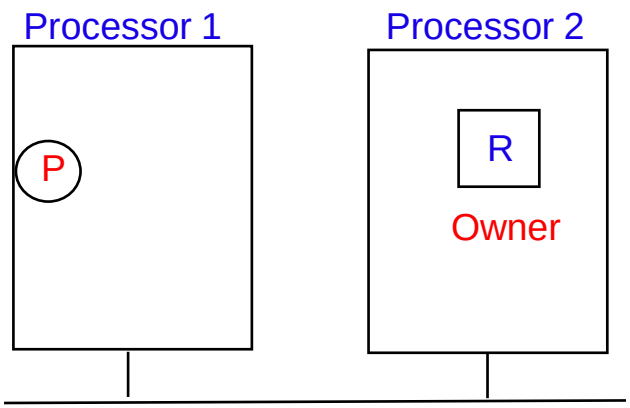
1. Invalidate other copies
2. Mark local page as W
3. Do write



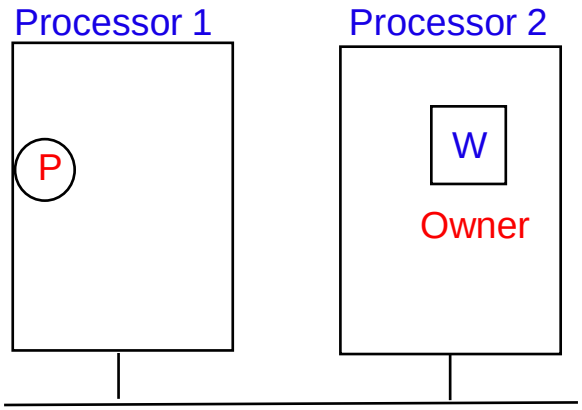
1. Invalidate other copies
2. Ask for ownership
3. Mark page as W
4. Do write

Invalidation Protocol (Write)

Suppose Processor 1 is attempting a write: Different scenarios



1. Invalidate other copies
2. Ask for ownership
3. Ask for a page
4. Mark page as W
5. Do write

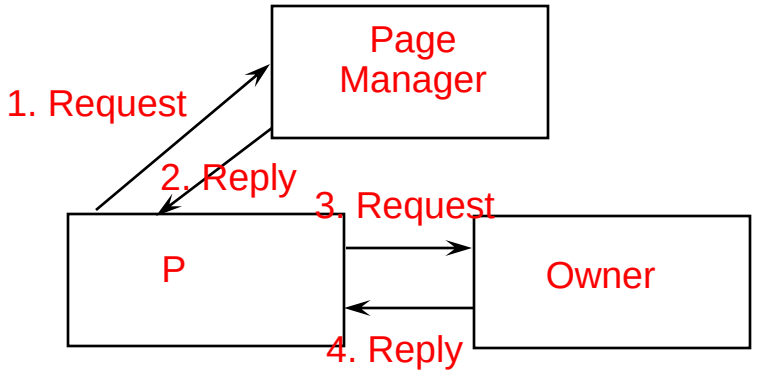


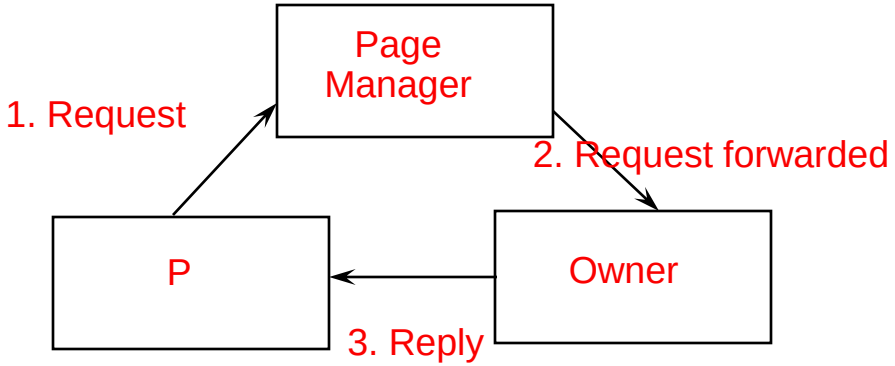
1. Invalidate other copies
2. Ask for ownership
3. Ask for a page
4. Mark page as W
5. Do write

Finding the Owner

- Owner is the processor with latest updated copy. How do you locate it?

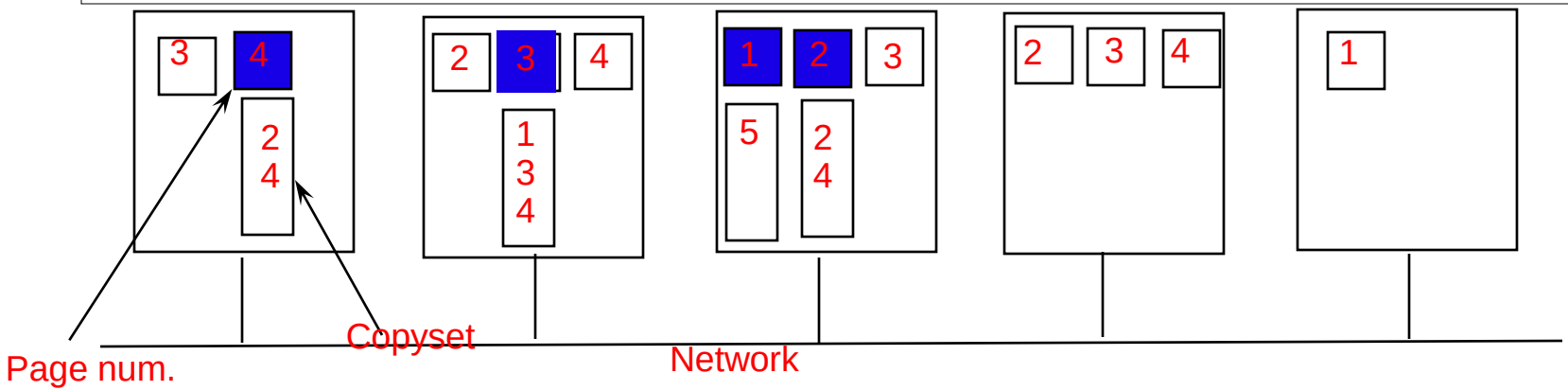
1. Do a broadcast, asking for the owner to respond.
   - Broadcast interrupts each processor, forcing it to inspect the request packet.
   - An optimization is to include in the message whether the sender wants to read/write and whether it needs a copy.
2. Designate a page manager to keep track of who owns which page.
   - A page manager uses incoming requests not only to provide replies but also to keep track of changes in ownership.
   - Potential performance bottleneck → multiple page managers
     - » Map pages to page managers using the lower-order bits of page number.





How does the Owner Find the Copies to Invalidate

- **Broadcast a msg giving the page num. and asking processors holding the page to invalidate it.**
  - **Works only if broadcast messages are reliable and can never be lost. Also expensive.**
- **The owner (or page manager) for a page maintains a *copyset* list giving processors currently holding the page.**
  - **When a page must be invalidated, the owner (or page manager) sends a message to each processor holding the page and waits for an acknowledgement.**



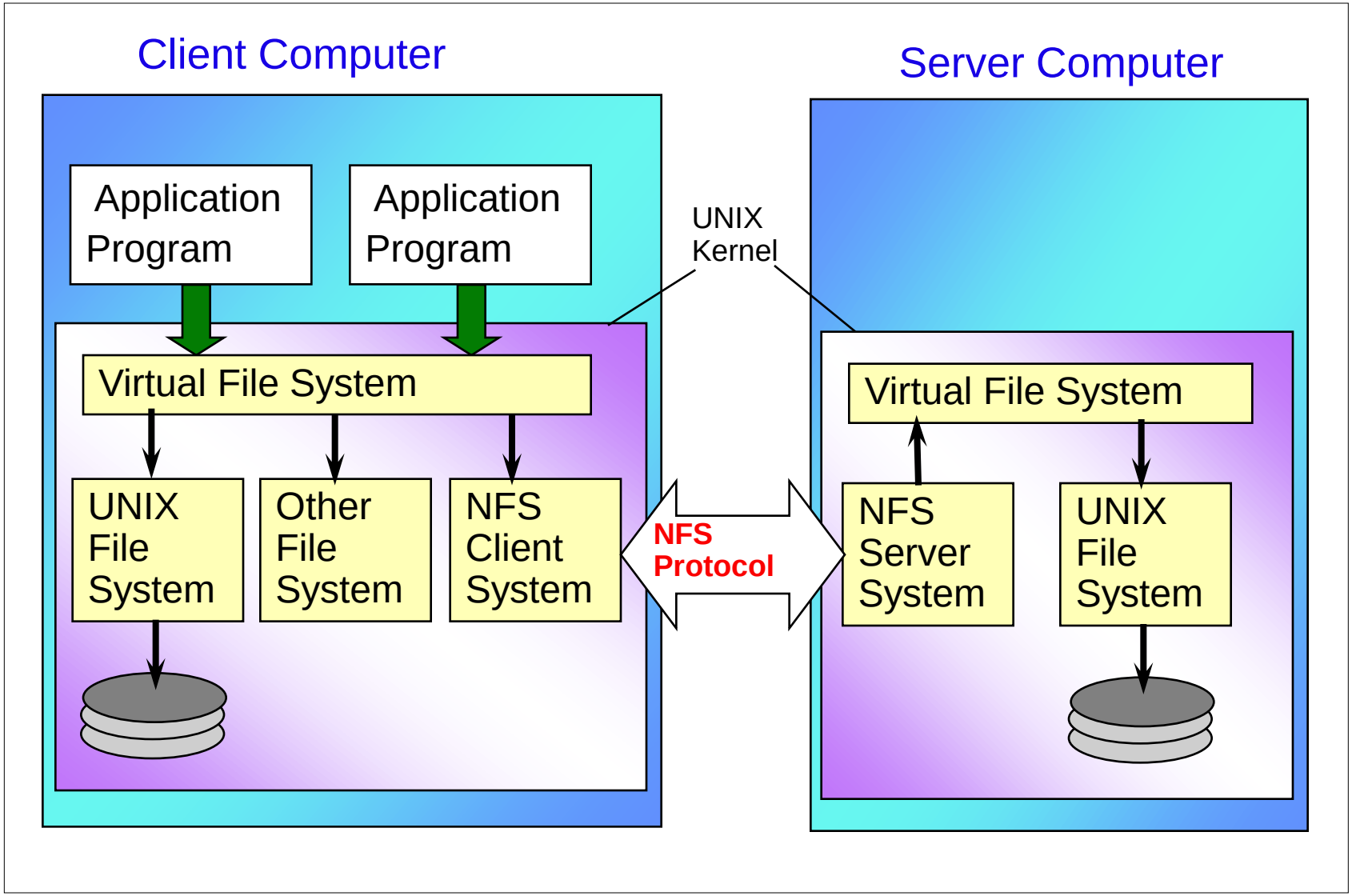
Announcements

- **MP4 due this Sunday. Demos next Monday (watch Piazza/wiki for signup sheet)**
- Mandatory to attend next Tuesday's lecture (last lecture)

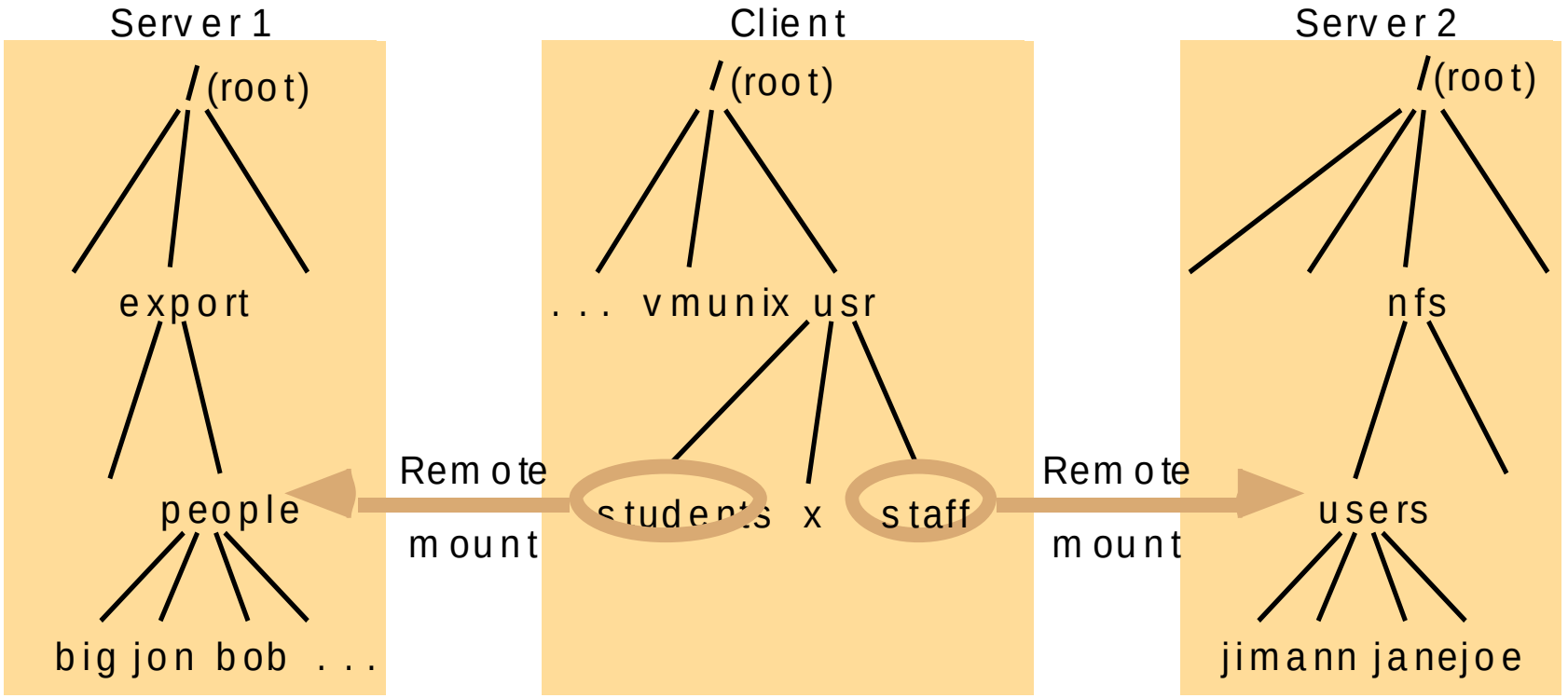
- **Final Exam, December 14 (Friday), 7.00 PM - 10.00 PM**
  - **Roger Adams Laboratory – 116 (1RAL – 116)**
  - **Do not come to our regular DCL classroom!**
  - **Also on website schedule**
  - **Allowed to bring a cheat sheet: two sides only, at least 1 pt font**
- **Conflict exam**
  - Please email Indy by this Friday (Dec 7) if you feel you might need to take a conflict exam
  - Conflict exam will likely be tougher than regular final exam

Backup Slides (Not Covered)

Network File System (NFS)



Local and Remote File Systems Accessible on an NFS client



Note: The filesystem mounted at */usr/students* in the client is actually the sub-tree located at */export/people* in Server 1; the file system mounted at */usr/staff* in the client is actually the sub-tree located at */nfs/users* in Server 2.

Hard mounting (retry f.s. request on failure) vs. Soft mounting (return error on f.s. access failure) – Unix is more compatible with hard mounting

NFS Client and Server

- **Client**
  - **Plays the role of the client module from the basic/vanilla model.**
  - **Integrated with the kernel, rather than being supplied as a library.**
  - **Transfers blocks of files to and from server via RPC. Caches the blocks in the local memory.**
- **Server**
  - **Provides a conventional RPC interface at a well-known port on each host.**
  - **Plays the role of file and directory service modules in the architectural model.**
  - **Mounting of sub-trees of remote filesystems by clients is supported by a separate mount service process on each NFS server.**

NFS Server Operations (simplified) – 1

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*.

NFS Server Operations (simplified) – 2

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*.