COT 4600 Operating Systems Fall 2009

Dan C. Marinescu

Office: HEC 439 B

Office hours: Tu-Th 3:00-4:00 PM

Lecture 9

- Last time:
  - Case study: the Unix file system
- Today:
  - Modular sharing
  - Metadata and Name Overloading
  - Addresses
- Next Time:
  - User-friendly names
  - Lifetime of names
  - Case study: URL

Unix File System

- Unix file system – hierarchical data organization:

  blocks → files → directories → file systems
  - the objects:
    - files – linear arrays of blocks of data; each file has a <u>cursor</u> giving the current position
    - directories – collections of files; tree structure
    - metadata - useful information about the file, not contained in the file (e.g., owner, access modes, last modified date, length, etc.)
  - supports:
    - creation, deletion, renaming of files and directories
    - reading data from and writing data to a file
    - reading and writing metadata describing a file

API for the Unix File System

OPEN(name, flags, model) → connect to a file

Open an existing file called name, or

Create a new file with permissions set to mode if flags is set.

Set the file pointer (cursor) to 0.

Return the file descriptor (fd).

CLOSE(fd) → disconnect from a file

Delete file descriptor fd.

READ(fd, buf,n) → read from file

Read n bytes from file fd into buf; start at the current cursor position and update the file cursor (cursor = cursor + n).

WRITE(fd, buf,n) → write to file

Write n bytes to the file fd from buf; start at the current cursor position and update the file cursor (cursor = cursor + n).

SEEK(fd, offset,whence) → move cursor of file

Set the cursor position of file fd to offset from the position specified by whence (beginning, end, current position)

API for the Unix File System (cont'd)

FSYNC(fd) → make all changes to file <u>fd</u> durable.
STAT(name) →read metadata
CHMOD, CHOWN → change access mode/ownership
RENAME(from_name,to_name) → change file name
LINK(name, link_name) →create a hard link
UNLINK(name) →remove <u>name</u> from directory
SYMLINK(name, link_name) →create a symbolic link
MKDIR(name) →create directory <u>name</u>
RMDIR(name) →delete directory <u>name</u>
CHDIR(name) → change current directory to name
CHROOT → Change the default root directory name
MOUNT(name,device) → mount the file system <u>name</u> onto <u>device</u>
UNMOUNT(name) → unmount file system <u>name</u>

Layers

- Unix file system uses a number of layers to hide the implementation of the storage abstraction from the users.
  - User-oriented names
    - 1. Symbolic link layer →integrates multiple file systems with symbolic names
    - 2. Absolute path name layer →provides a root for the naming hierarchies
    - 3. Path name layer → organizes files into naming hierarchies
  - Machine-user interface
    - 4. File name layer → Supplies human-oriented names for files
  - Machine-oriented names
    - 5. Inode number layer → provides machine-oriented names for files
    - 6. File layer → organizes blocks into files
    - 7. Block layer → Provides the physical address of data blocks

7. Block layer

- The storage device (disk) → a linear array of cells/blocks
- Block → fixed-size allocation unit (e.g., 512 bytes, 2,048 bytes); occupies several disk sectors.
- Block name → integer from a compact set, the offset from the beginning of the device
- Boot block → usually contains a boot program; Has a well-known name, 0.
- Super block → provides a description of the layout of the file system on the disk. Has a well-known name,1.
- Bitmap to keep track of free blocks and of defective blocks.

Disk layout

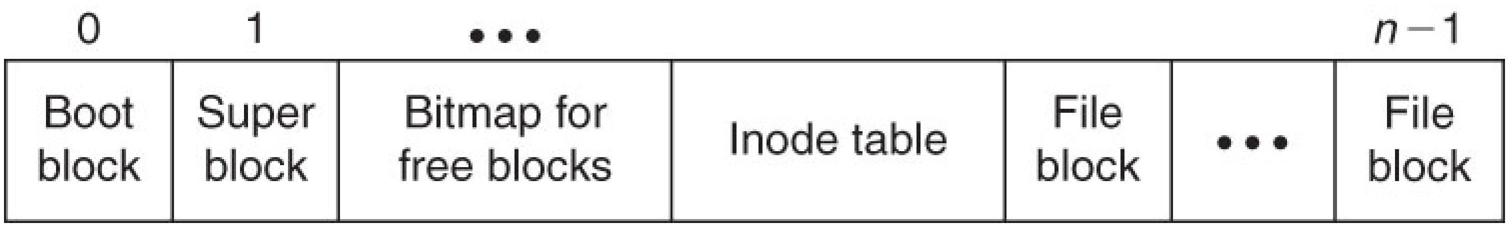


Figure 2.20

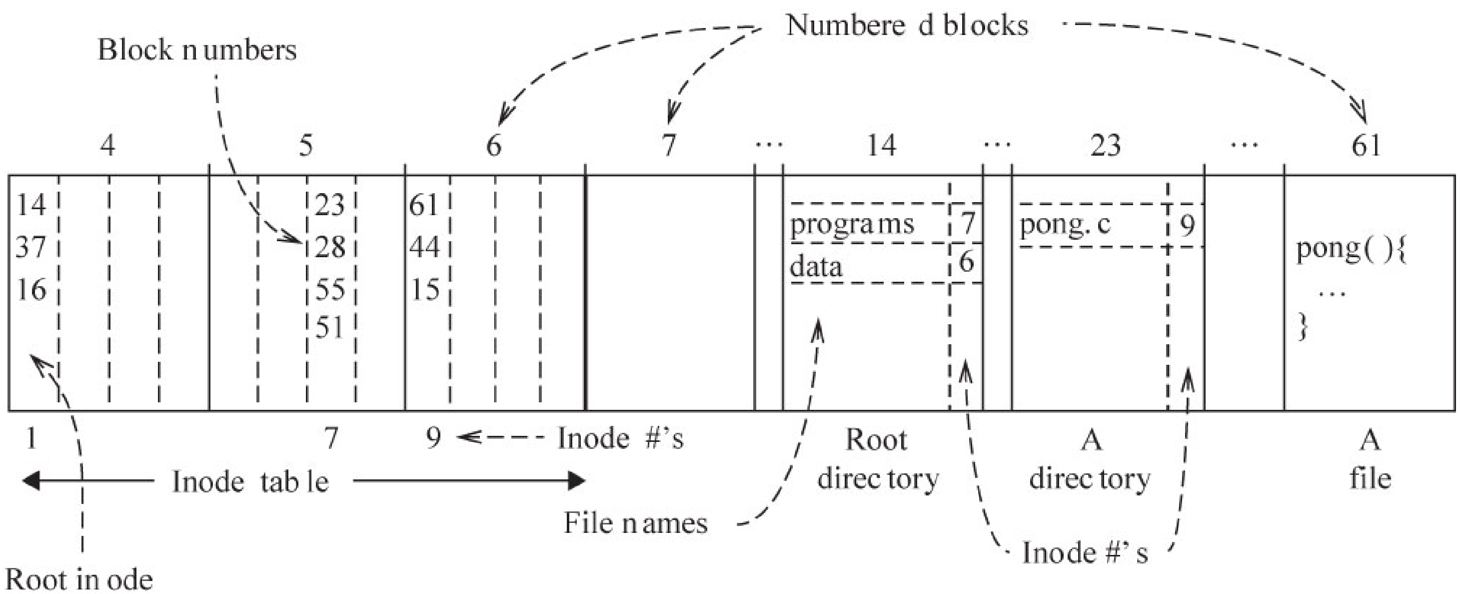

Numbered blocks
Block numbers
4
5
6
7
14
23
61
14
37
16
23
28
55
51
61
44
15
programs
7
data
6
pong. c
9
pong( ){
…
}
1
7
9
Inode #'s
Inode table
Root inode
File names
Root directory
A directory
Inode #' s
A file

Figure 2.22

6. File layer

- A file consists of multiple blocks.
- inode (index node)→ container for the metadata about the file; context for naming the blocks of the file

structure inode

    integer *block_number[N]* // the number of the block in the file

    integer *size* // file size in bytes

- Name mapping algorithm:

procedure *INDEX_TO_BLOCK_NUMBER (inode* **instance** i, **integer** index) returns **integer**

    return *i_block_numbers[index]*

- Indirect blocks → blocks that contain block numbers rather than data → to accommodate large file.
- Doubly indirect blocks → blocks that contain block numbers of indirect blocks
- Example: UNIX V6
  - the first N entries in *i_block_numbers* are indirect blocks and the N+1 is a doubly indirect block
  - the block size is 512 bytes;
  - an index consists of 2 bytes → an indirect block may contain 256 block numbers
  - the maximum file size is: (N-1) x 256 + 256 x 256 blocks

5. Inode number layer

- inodes are named and passed around by name (inode number)
- inode numbers are a compact set
- inode_table →
  - maps inode names to the starting block of the inode
  - stored at a known-location (e.g., at the beginning of the storage device)
- Inode manipulation functions: allocate, dealocate, add to list of free inodes, remove from list of free inodes
- Name mapping algorithm → procedure that returns the block that contains the byte at *offset* in a file identified by *inode_number*:

procedure *INODE_NUMBER_TO_BLOCK(***integer** *offset*, **integer** *inode _number*) **returns** *block*
  inode **instance** *i* ← *INODE_NUMBER_TO_INODE [inode,number]*
    offset_in_block ← *offset/blocksize*
    b_number ← *INDEX_TO_BLOCK_NUMBER(i, offset_in_block)*
    return *BLOCK_NUMBER_TO_BLOCK [b_number]*

4. File name layer

- Maps human-oriented names to machine-oriented names.
- Hides the metadata required by file management from the user.
- Directory →
  - durable object providing the context for binding between a file name and an inode number
  - it is a file
  - the inode data structure is extended with a type field to indicate if the inode is for a file or for a directory.
- To create a file:
  - allocate an inode
  - initialize the metadata for the file
  - bind the file name to the inode number

File name lookup

- The *lookup* procedure reads the blocks containing the data for directory *dir* searching for a file called *filename* and return the inode number for the file is it finds *the filename*

procedure *LOOKUP* (**character string** *filename*, **integer** *dir*) **returns integer**

block **instance** *b*

inode **instance** *i* ← *INODE_NUMBER_TO_INODE [dir]*

if i.type ne then **return** FAILURE

for offset **from** 0 to i.size -1 **do**

b ← *INODE_NUMBER_TO_BLOCK (offset,dir)*

return *BLOCK_NUMBER_TO_BLOCK [b_number]*

if *STRING_MATCH (filename, b)* **then**

return *INODE_NUMBER (filename,b)*

offset ← *offset +BLOCKSIZE*

return *FAILURE*

3. Path name layer

- Directories are hierarchical collections of files.
- The path name layer → structure for naming files in directories
- The name resolution algorithm:

procedure *PATH_TO_INODE_NUMBER* (**character string** *path*, **integer** *dir*) **returns integer**
  if *(PLAIN_NAME(path)* **return** *NAME_TO_INODE_NUMBER(path,dir)*
  else
    dir ← *LOOKUP (FIRST(path),dir)*
    path ← *REST(path)*
    return *PATH_TO_INODE_NUMBER (path,dir)*

Links

- link → synonym allowing the user to use in the context of the current directory a symbolic name instead of a long path name.
- This requires binding in a different context; it does not require any extension of the naming scheme.
- Example:

link(“/usr/applications/project/bin/programA”,”progA”

if the current directory is alpha (with inode # 113) then the link directive will add to the directory “/usr/applications/project/bin” a new entry → (progA, 113)

- The unlink removes the link. Removal of the last link to a file also removes the file.

2. Absolute path name layer

- UNIX shell starts execution with the working directory set to the inode number of user's home directory.
- To allow users to share files with one another the UNIX file system creates a context available to every user. This root directory binds a name to to each user's top-level directory

1. Symbolic link layer

- Allows a user to operate on multiple file systems
  mount(“dev/fd1”, “/flash”).

Practical design of naming schemes

- Transition from abstract models to practical ones.
- Name conflict → multiple modules have the same name.
- How to avoid name conflicts when modules are developed independently often by different individuals?
- The theoretical model tells us that we must specify a context for name resolution; but this is not so straightforward!!
  - Example: there are two versions of module A; one is used by module B and the other by module C. Conflict when module B uses C.

Single context → ambiguity

WORD_PROCESSOR → (INITIALIZE,SPELL_CHECK)
SPELL_CHECH →(INITIALIZE) (but a different version of it)

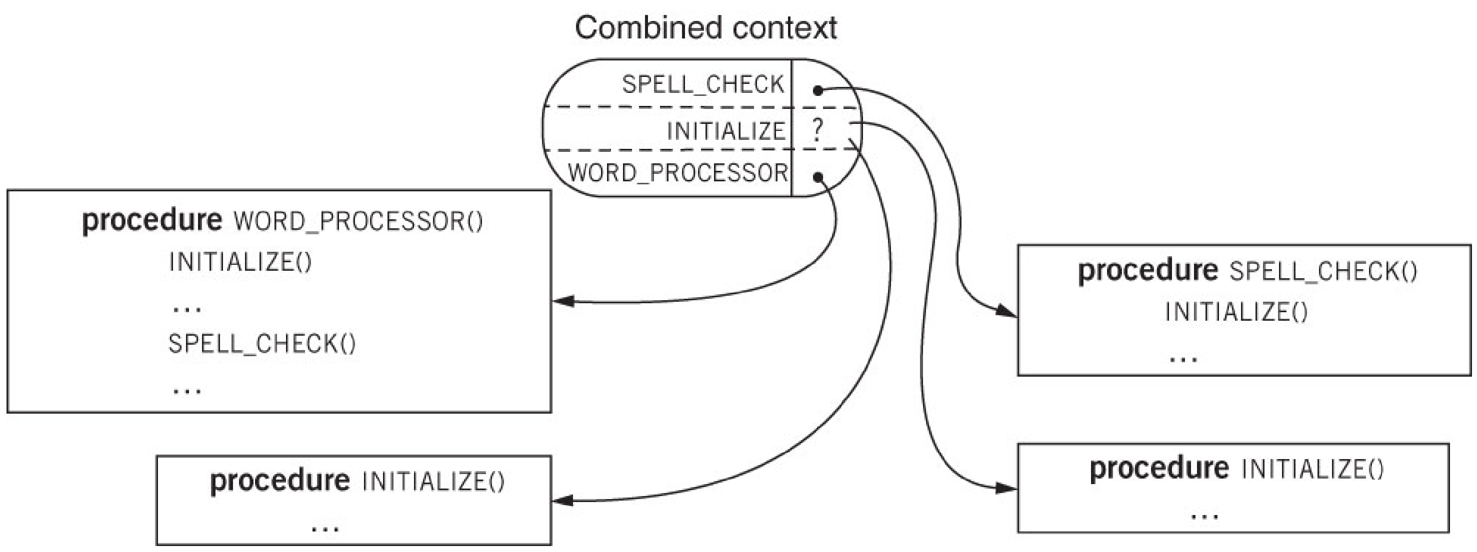


Figure 3.1

Two distinct contexts → how does the interpreter choose the context? It needs a basis for the contexts.

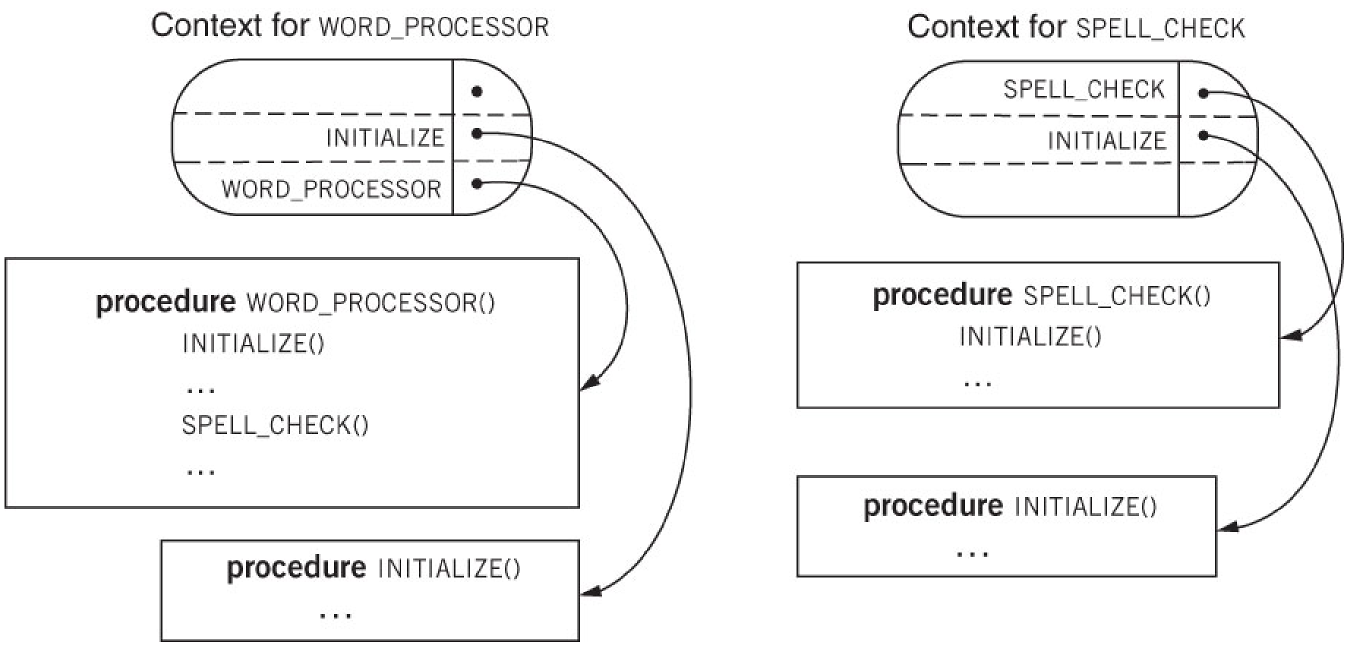


Figure 3.3

Add a context reference to each module telling the interpreter which context to use for that module? Not feasible to tinker with someone else’s modules.

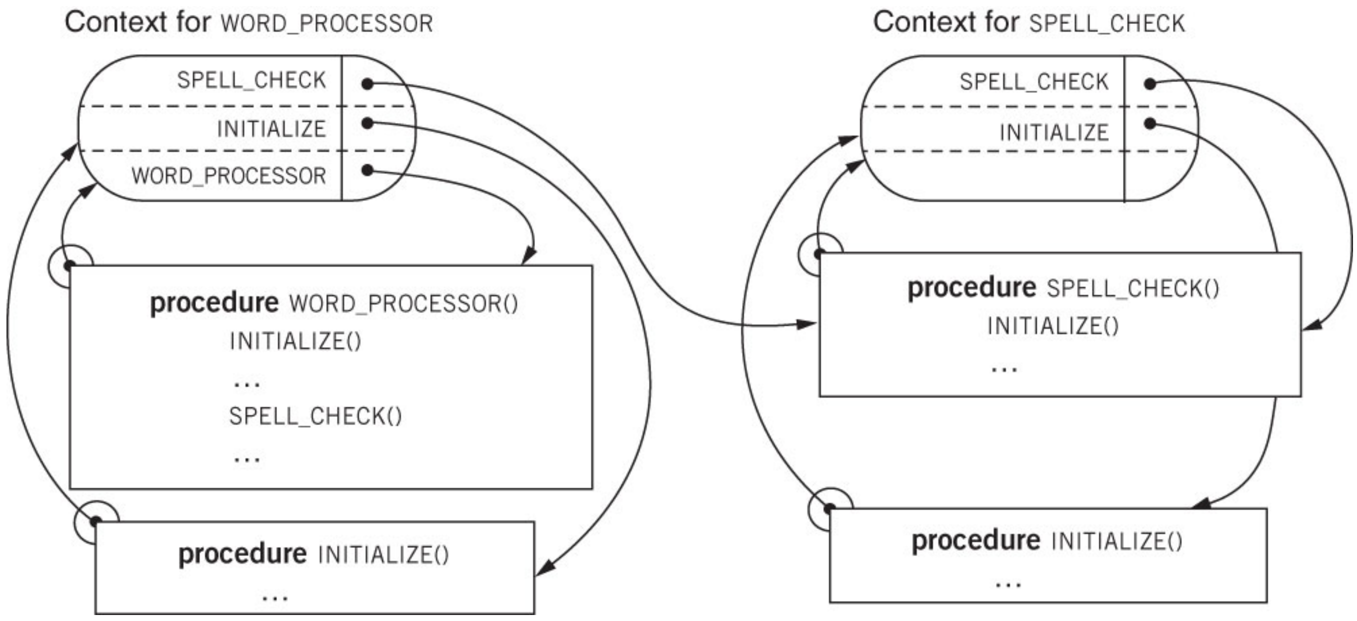


Figure 3.4

Have separate contexts but establish a link between them; the link points to the new context for the shared object

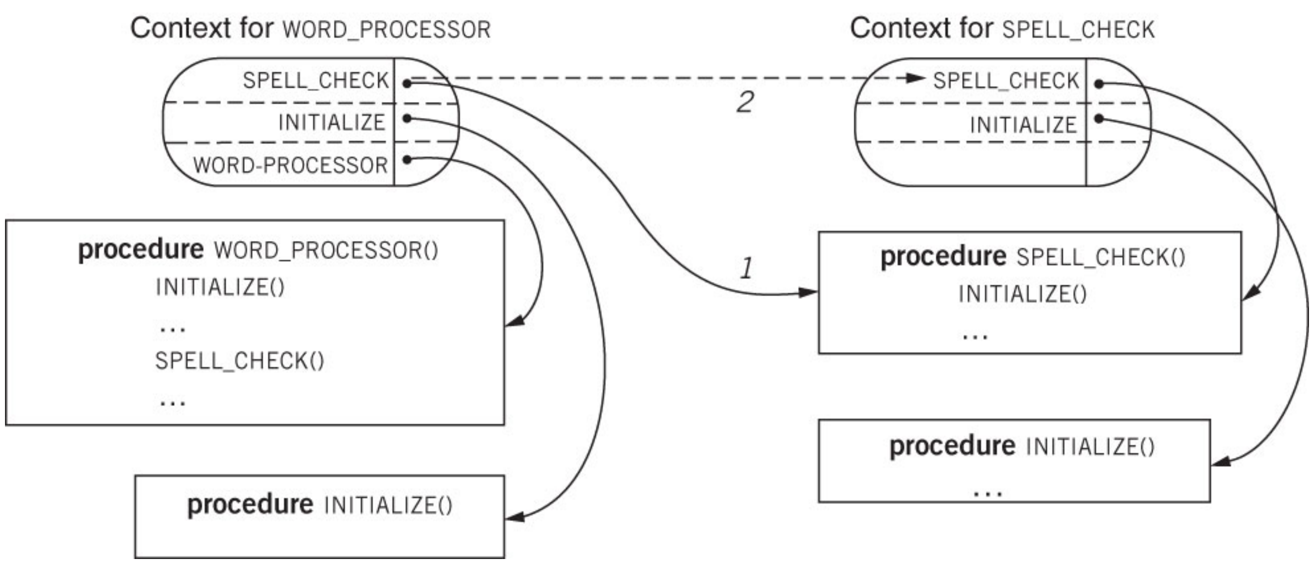


Figure 3.5

An elegant solution

- We need a systematic rather than ad-hoc ways to deal with the problem because programs contain many references to objects.
- Solution → associate the name of the object not with the object itself but with a structure consisting of pairs (original object, context).
- Some programming languages implement such a structure called closure and use static references.
- File systems rarely apply this solutions.

Metadata and name overloading

- Metadata → information about an object that
  - is useful to know about the object but cannot be found inside the object
  - may be changed without changing the object, e.g., the last date a file was referenced.
- Examples:
  - A user-friendly name; e.g., quadratic_solver → /user/local/bin/ linpack/quad.exe
  - The type of an object: e.g., Lecture9.ppt
- Where to place metadata?
  - In the same place with the data; e.g., the Unix file system stores metadata in inodes
  - In a separate place → process control block stored in the kernel space.
  - Overloading the name → e.g., Lecture9.ppt
- Overloading → adding metadata to a name
  - Contradicts the principle that names should only be used to reference objects
  - Creates a tension between the need to keep the name unchanged and the need to modify overloading information.

Names and overloading

- Pure names → names with no overloading.
- Fragile names → overloaded names which violate the idea of modular design. E.g.., adding location information on the file name.
- Opaque name to a module → the name has no overloading the module knows how to interpret.
  - A name may pass through several modules before reaching a module which knows how to interpret it.

Addresses

- Address →
  - A name used to locate an object
  - Not a pure name it is overloaded with metadata
  - Parsing an address provides a guide to the location
- Often chosen from a compact set of integers
  - Does address adjacency correspond to physical adjacency?
    - True in some cases; e.g., sectors on a disk
    - False in other cases; e.g., the area code of a phone number
  - Can we apply arithmetic operations to addresses?
    - Yes in some cases; e.g., memory references
    - No in other cases; e.g; telephone numbers (actually the phone numbers do not forms a dense set!!)
- Remember that overloading the causes name fragility

Changing addresses

- Changing addresses not hidden by a level of indirection is tricky.
- Solutions
  1. Search for all addresses and change them
  2. Make each user do a search for the object and if the search returns "object not found" detect that the address has been changed and supply the new address.
  3. If possible bind the object to both the old and the new name
  4. If the name is bond to an active agent place a forwarding scheme to the old address.
- The optimal solution → hide an address under a level of indirection.

  We'll discuss DNS, domain name services that map host names to IP addresses.

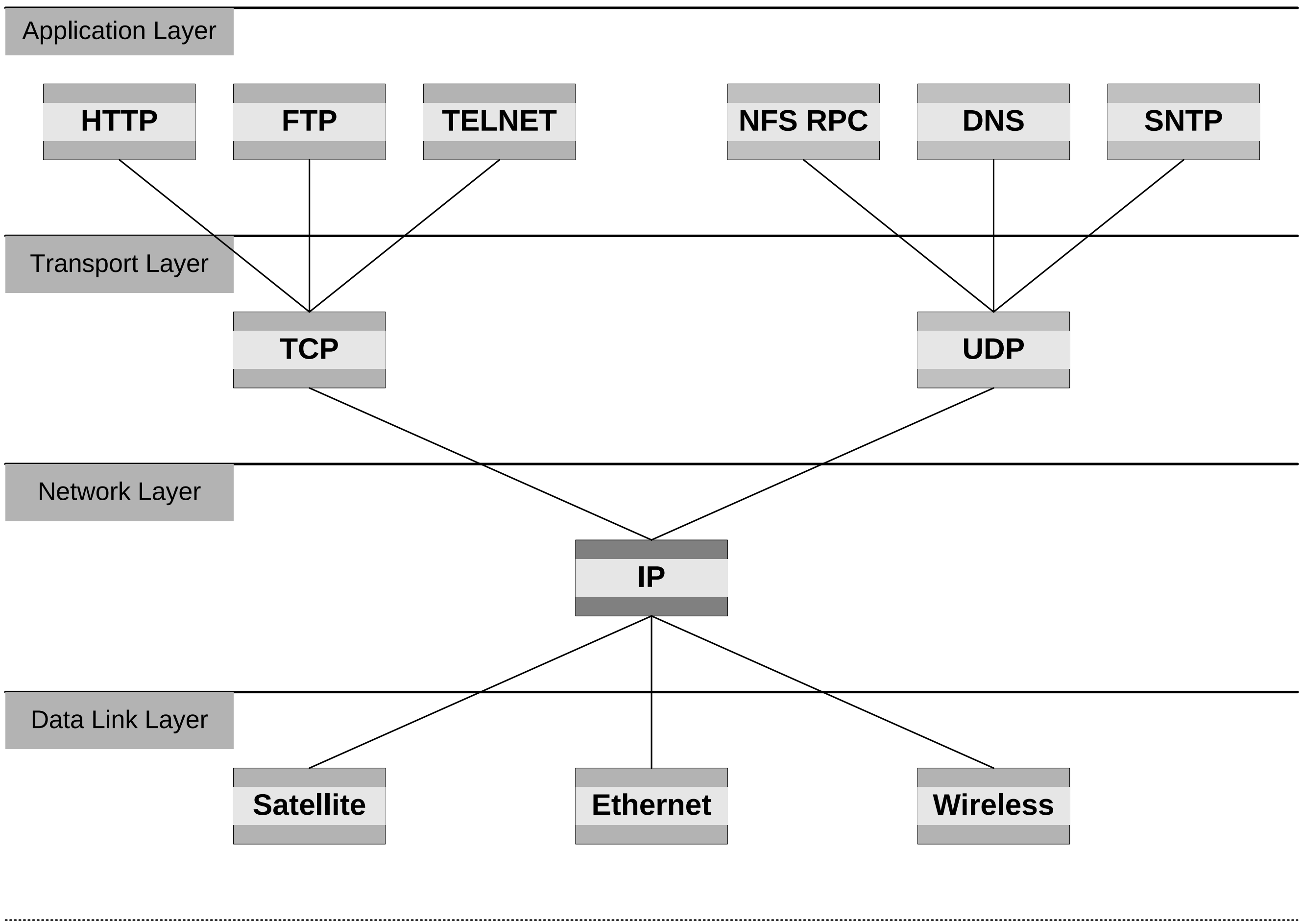
Unique names

- Problems arise when names must be generated at a high rate., e.g., for online banking transactions (billions a week).
- Danger of name collision.
- Solution
  - Using a fine timestamp→ read a fine digital clock (say with a resolution of microseconds) and convert the binary representation of the timestamp to a string of characters.
  - Use a random number generator with a very large name space.
  - For objects with a binary representation (e.g., files, images) use the object itself.
  - Hashing algorithms such as SHA (Secure Hash Algorithm) avoid the problem of long names. HSA produces names of fixed length.

Hierarchical naming schemes

- Think about naming in the Internet with hundred millions of hosts.
  - Unfeasible with a central authority.
  - Domain names
  - E.g., boticelli.cs.ucf.edu
- How to relate a hierarchical naming scheme used by Internet with the flat naming schemes used for MAC addresses?
  - MAC addresses do not have any overloading
  - ARP
  - RARP
  - DHCP

Application, Transport, Network, and Data Link Layer Protocols



Dynamic IP address assignment -DHCP