Objects and Capabilities

Peter J. Denning

Digital objects are pervasive

- We hear about digital objects all the time -- files, documents, videos, sounds, processes, and more
- In the OS, many of these map into files, pipes, devices, directories, buffers, and virtual machines
- Challenge: manage large numbers of digital objects belonging to many different users
  - Allocate storage
  - Map names to memory locations
  - Protect them
  - Share them when allowed

- Around 1965, OS designers defined interfaces for different kinds of OS objects – for example, files, processes, devices -- and introduced shareable, unforgeable “capabilities” to point to them.
- Around 1965, OS designers also devised kernel layers as object managers for OS objects
- This ideas of object classes first appeared in a language SIMULA 1967
- Type hierarchies (allowing subtypes) appeared by 1970.
- First object oriented language SMALLTALK 1972

Object Manager

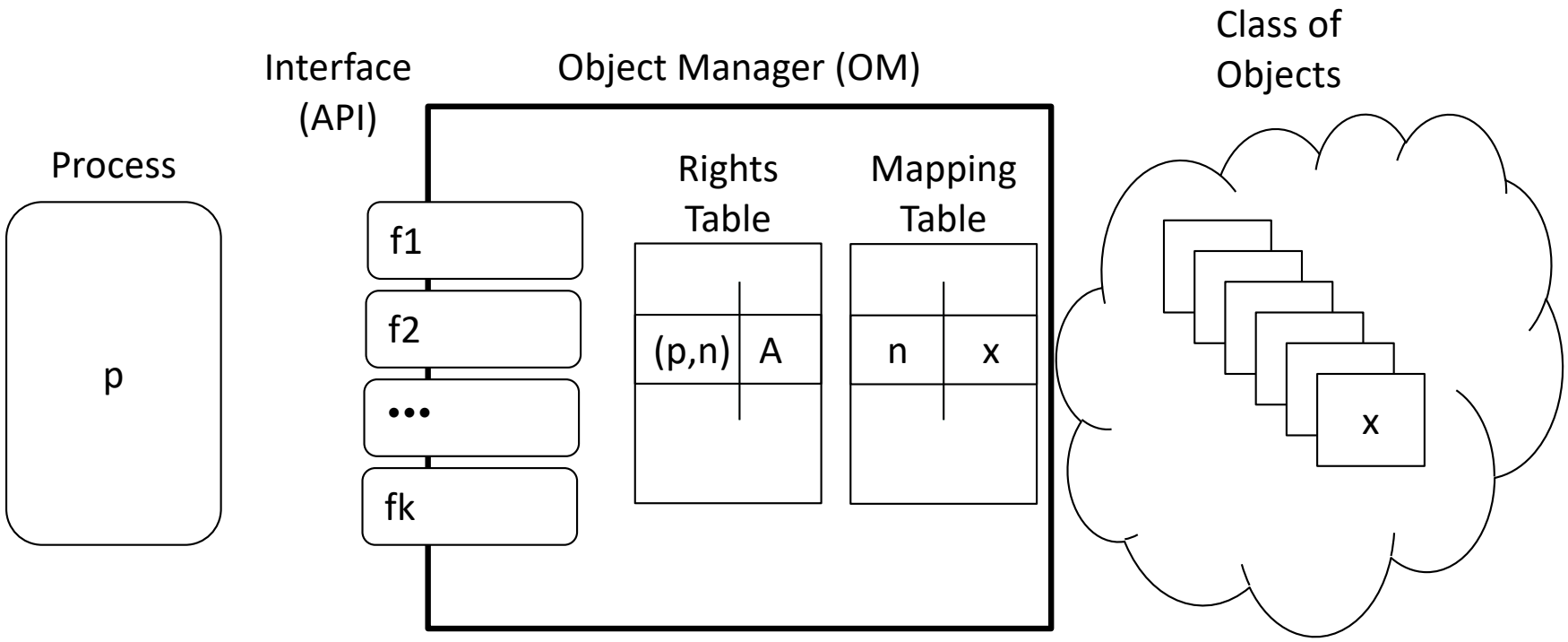
- All objects of same type managed by an object manager
  - Acts as reference monitor (access control)
  - Simple user interface with limited number of operations (its API)
  - Creates new objects and pointers to them
  - Deletes existing objects
  - Hides a lot of implementation detail
- Use file system as a working example

THE GENERAL SITUATION

p, one of many processes, seeks access to object x

f1…fk is the interface (a set of functions) to object manager (OM)

A is k-bit access code: which fi are allowed



Process p generates request (p,fi,n) by calling fi with argument n.

OM gets access code A for (p,n) from Rights Table.

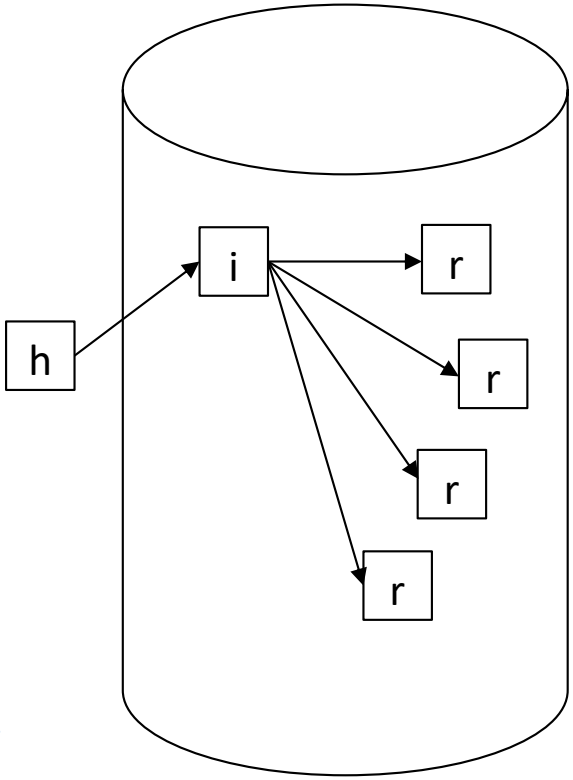
From A, OM verifies that p has right to invoke fi and, if so, OM maps n to the memory location x of object, applies fi to x, and returns a response.

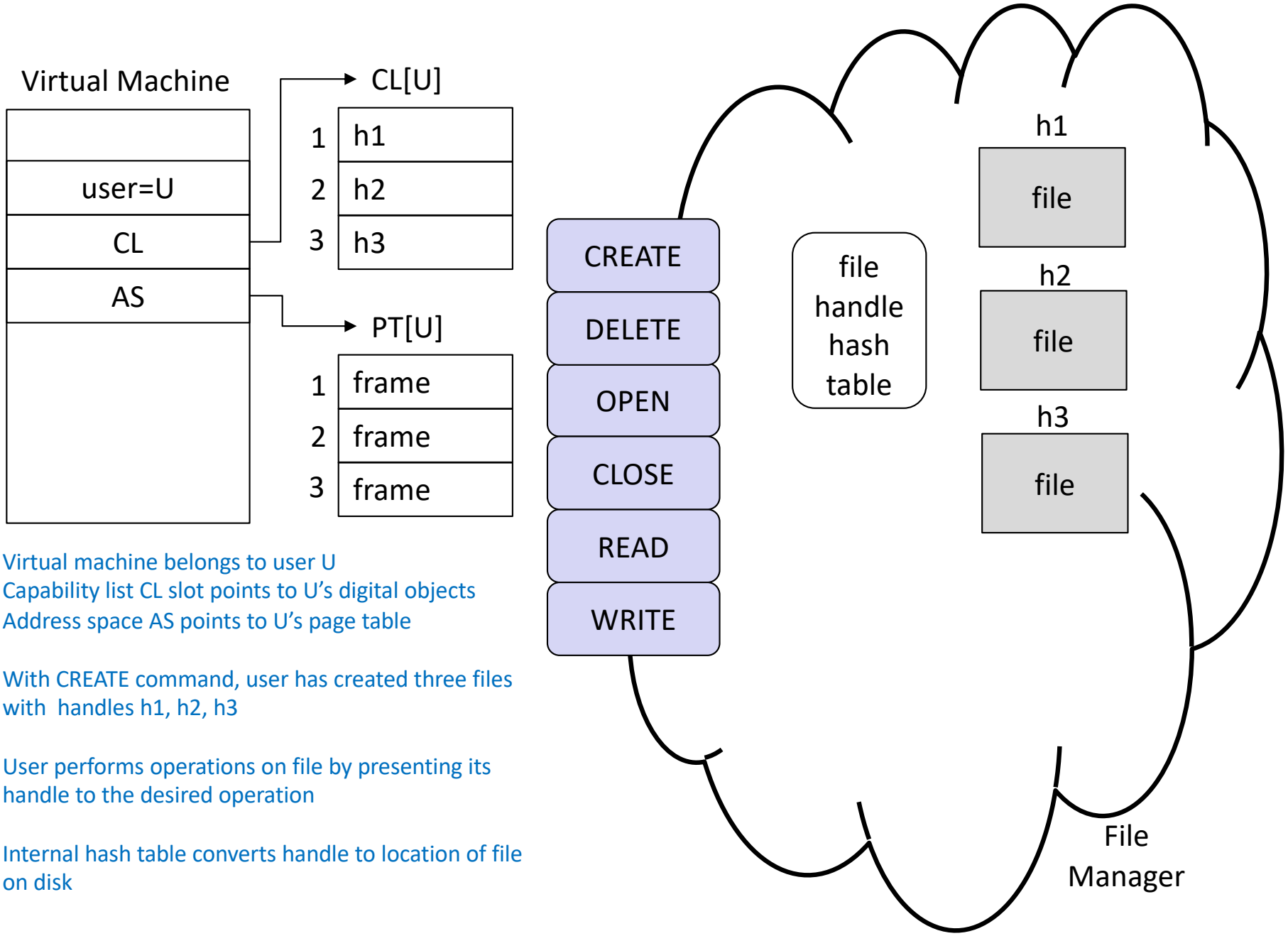
Implementation Issues

- Mapping names to objects
  - Given name n, where is object x it designates?
  - Performance of the translation n to x?
  - Validate access codes in parallel with translation (no time penalty)
  - General ideas already discussed in the name-mappings module
- Access code verification
  - Where do access codes come from and where stored?
  - Performance of access rights validation?
  - General ideas already discussed in the access module

File System

- File: container of a stream of bits with name and length
- Stored in persistent memory (usually disk)
  - Broken into fixed chunks called records (r)
  - Records scattered around the disk
  - Index record (i) points to all the others
- File handle (h) points to index record
  - Must be UFAT: Unique for All Time
  - h = (time stamp, MAC)
    where MAC = media access control ID
- Simple interface
  - CREATE, DELETE, OPEN, CLOSE, READ, WRITE





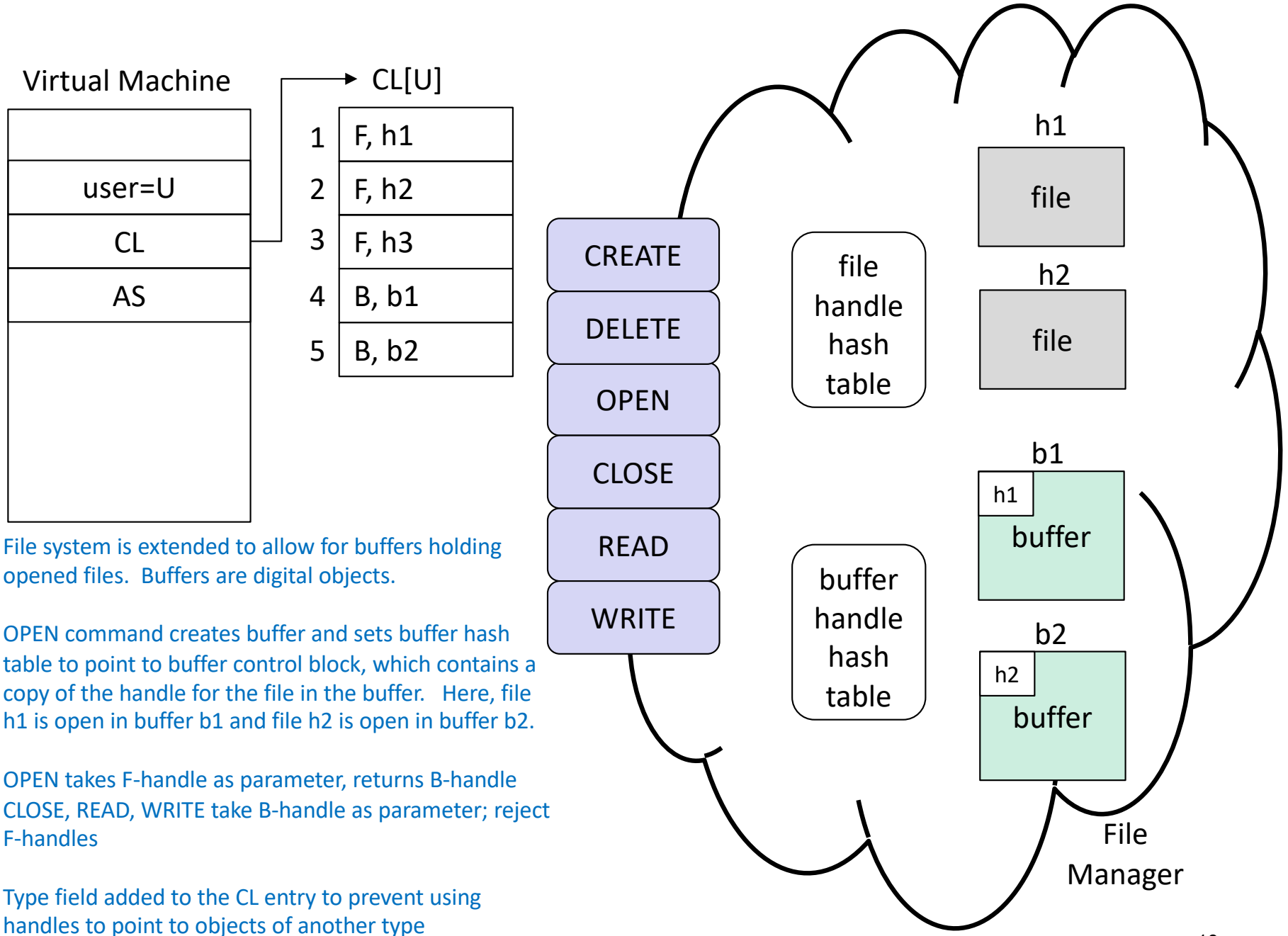
Virtual machine belongs to user U
Capability list CL slot points to U's digital objects
Address space AS points to U's page table

With CREATE command, user has created three files with handles h1, h2, h3

User performs operations on file by presenting its handle to the desired operation

Internal hash table converts handle to location of file on disk

- Problem: Severe performance problem if direct READ and WRITE to disk for every file access is allowed because disk much slower than RAM
- Solution: File buffer, a region of file system private memory containing an image of the file
  - Load entire file into a main memory buffer in proper linear order
  - Created by OPEN operation
  - All READ and WRITE performed on the buffer
  - CLOSE operation copies buffer back to disk, replacing earlier version
- Now there are two types of handles: file, buffer
  - Each operation validates parameter handles for proper type



File system is extended to allow for buffers holding opened files. Buffers are digital objects.

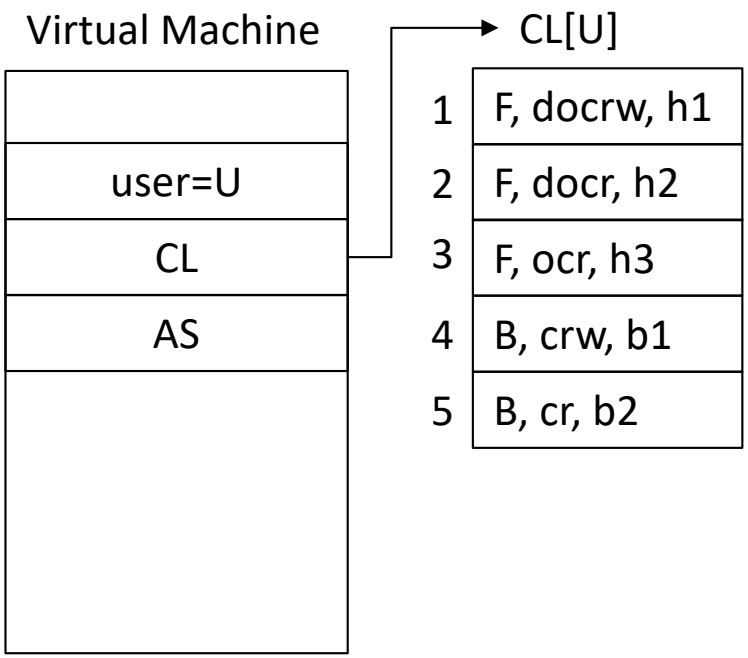
OPEN command creates buffer and sets buffer hash table to point to buffer control block, which contains a copy of the handle for the file in the buffer. Here, file h1 is open in buffer b1 and file h2 is open in buffer b2.

OPEN takes F-handle as parameter, returns B-handle
CLOSE, READ, WRITE take B-handle as parameter; reject F-handles

Type field added to the CL entry to prevent using handles to point to objects of another type

- Problem: access control … which operations are allowed?
  - CREATE always allowed
  - DELETE and OPEN work only on files
  - READ, WRITE, and CLOSE work only on buffers
- To each file handle, attach 5-bit code, one bit enabling each non-CREATE operation
- To each buffer handle, attach 3-bit code, one bit enabling each of CLOSE, READ, and WRITE, inherited from the file handle access code
- Each operation validates incoming parameters for
  - Proper type
  - Proper bit of access code set

- Initial handles
  - file handle (from CREATE) has all 5 bits on
  - buffer handle (from OPEN) inherits file handle's CLOSE, READ, and WRITE bits
- Owner can make copies with reduced privileges



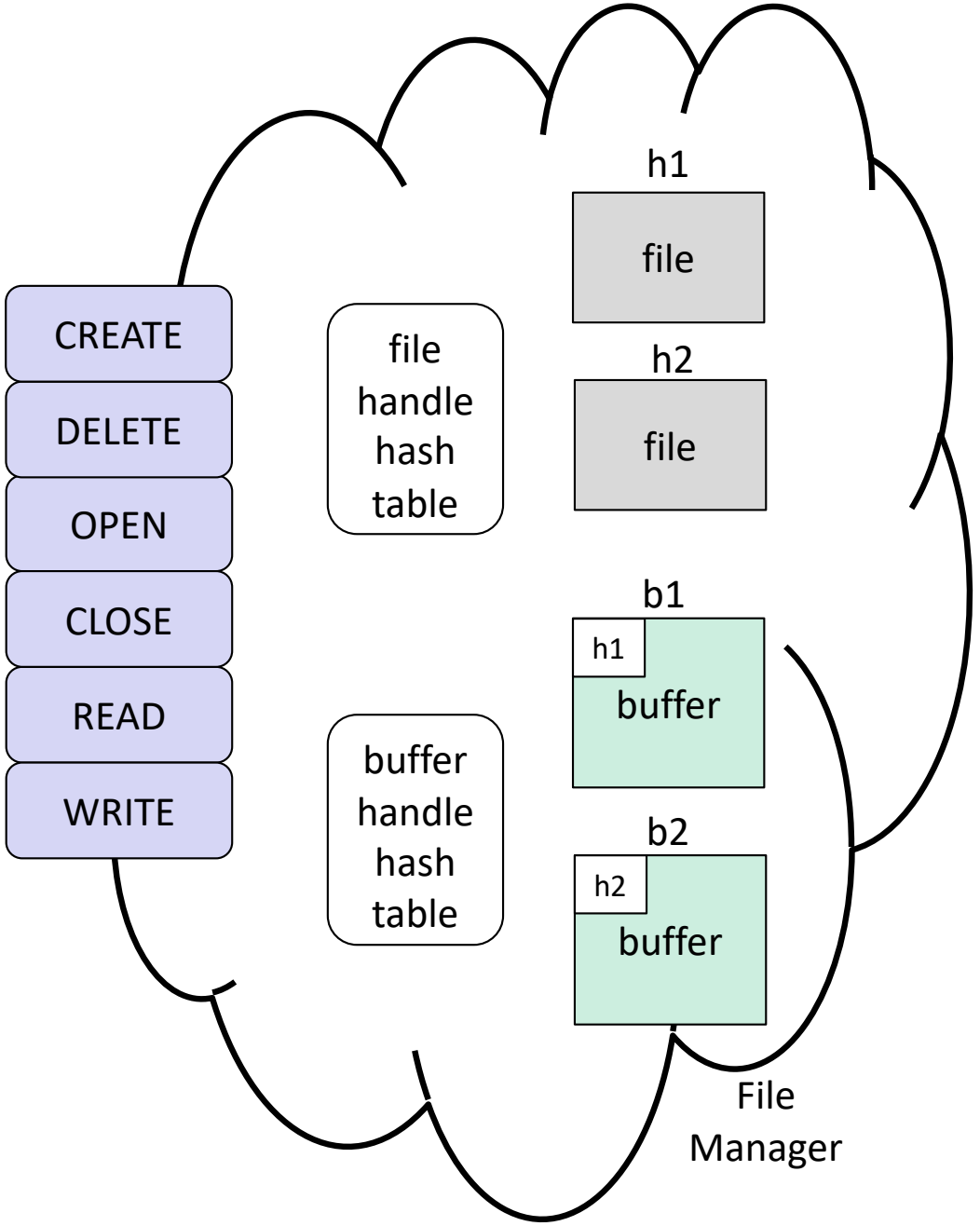
Handles are extended with access codes. Each operation checks parameter handle for proper type and, if so, checks that access bit allows this operation.

Initial file handle (from CREATE) has all 5 access bits on.

Initial buffer handle (from OPEN) inherits bit for CLOSE, READ, WRITE from file handle

From now on, the term **capability** refers to the entire CL entry (type, access, handle)

Capabilities are **unforgeable**: protected unmodifiable once issued because the CL is inaccessible to processes



Capabilities

- A capability is an unforgeable certificate granting particular access to a particular digital object
- Capability = (type, access, handle)
  - type indicates type of object pointed to
  - access indicates which operations are allowed
  - handle is UFAT pointer to the object
- Object manager has a hash table for each type of handle it supports, mapping handles to descriptors (control blocks) of its digital objects
- Having a capability is proof of permission to access
- The C-list (CL) implements unforgeability: no process can modify the contents of C-list, which is in kernel space

File Manager Interface

- i = CREATE
  - Create empty file and place a new all-access file capability at CL[i]
- DELETE(i)
  - Delete the file pointed to by CL[i] and clear CL[i]
- j = OPEN(i)
  - Create a buffer loaded with the records of file pointed to by CL[i]
  - Create a buffer capability a CL[j] with the crw access bits of CL[i]
- CLOSE(j)
  - Write buffer back to disk, replacing older version, delete buffer, and clear CL[j]
- L = READ(j, A)
  - Copy buffer CL[j] to caller's address space region (base, length) = (A,L)
- WRITE(j, (A,L))
  - Replace buffer CL[j] contents with caller's address space region (A,L)

Streams

- A string of bits appearing in
  - File: string of bits with length
  - Pipe: bits that have been sent by the producer and not yet received by the consumer
  - Device: source of bits (e.g., keyboard) or sink of bits (e.g., display)
- Sense of flow; file is snapshot of a flow
- Basis of pipeline model for commands

What does OPEN do?

- Why open file
  - set up RAM buffer to avoid expensive disk access for every read and write
- Why open pipe
  - set up buffer and semaphores for producer-consumer relationship between virtual machines
- Why open device
  - set up buffer and semaphores to allow consumer relationship with source, producer relationship with sink

Generic Stream Interface

- X in {Files, pipes, devices} are stream objects with separate interfaces
  - CREATE_X, DELETE_X, OPEN_X, CLOSE_X, READ_X, WRITE_X for X = FILE, PIPE, DEVICE
- Can simplify to generic
  - CREATE, DELETE, OPEN, CLOSE, READ, WRITE
- Generic operation reads the type field of incoming capability parameter and routes to the appropriate operation.
  - j = OPEN(i) reads CL[i].type=X and routes to OPEN_X(i)
  - The returned capability CL[j] is of proper type for X
  - The returned CL[j].access inherits from CL[i].access

Simple Sharing

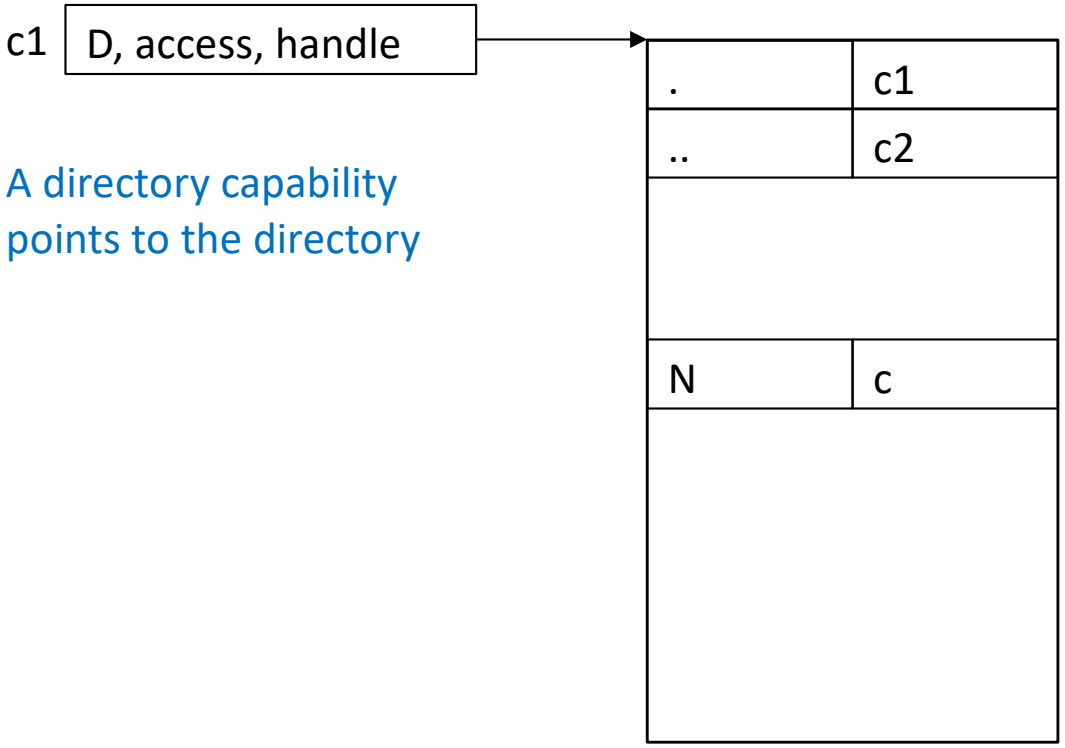
- Two operations are needed for sharing capabilities
- j = COPY_CAP(i,mask) – make a copy in the same C-list
  - CL[j] identical to CL[i] except that its access code attenuated: CL[j].access = CL[i].access AND mask
- c = MAKE_PUBLIC(i) – make a public capability that is unforgeable outside the kernel
  - c is a bit pattern = (t,a,h,IP,check), where t, a, h are from CL[i], IP is this machine's IP address, and check is a crypto hash of (t,a,h,IP)
  - Any recipient can verify c is not modified by computing the hash of its (t,a,h,IP) and comparing with check
  - Attenuated public form c = MAKE_PUBLIC(COPY_CAP(i,mask))

- Note internal and public forms of capability
  - Internal: c=(t,a,h) stored in C-list in kernel space
  - Public: c = (t,a,h,IP,check) can be sent anywhere in the network
- Both are unforgeable
- The c.IP allows recipient to process the capability on the host containing object

Directories

- Problem. How does user manage all the file caps?
  - Could be thousands of them
  - The user must keep track of all the index numbers, e.g., the i in CL[i], which are assigned by the OS by CREATE
- Solution: Directories.
- Directory is a table in which every entry (N,c) associates user-chosen symbolic name N with a public capability c
  - No two entries can have the same N
  - Capability c is public form because it is outside the kernel
  - What goes wrong if c=CL[i] and directory entry is reduced to (N,i)?

Directory is a digital object with a capability pointing to it



A directory capability points to the directory

By convention first entry is called "." and points to this directory.

Second entry is called ".." and points to the parent directory.

Typical entry (N,c) can point to a file or a directory.

Additional topics for directories discussed in the next module

- Can be arranged in trees by including entries for subdirectories
- Why first entry points to self
- Why second entry points to parent
- CD, the current directory pointer in a virtual machine
- PATH, list of directory pathnames to search for executable files
- Directories may be stored in files, but are not themselves files or stream objects

Basic Directories Interface

- d = CREATE_DIR(p)
  - create a new empty directory (initialized with (".", d) and ("..", p) and return public directory capability pointing to it.
- DELETE_DIR(d)
  - Delete the directory pointed to by directory capability d
- c = SEARCH(d,N)
  - d is directory capability, N is symbolic name, c the associated public capability
- c = SEARCH_PATH(PATH,N)
  - search all directories named in PATH for name N

- ATTACH(d, (N,c))
  - create a new entry (N,c) in directory d
  - N must be distinct from other names in d
- REMOVE(d,N)
  - remove the entry (N,c) from directory d
- RENAME(d,N,M)
  - change entry (N,c) to (M,c) if M distinct from other names in d

Capabilities Manager

- By now you should have a good idea how capabilities work and how they are used to point to objects in the upper levels of the kernel
- We collect all the capability operations into the Capability Manager interface at Level 5

Capabilities Interface

- i = CREATE_CAP(t,h)
  - Generate (t,a,h) in caller's C-list, return position i, a is all-access
  - Function CREATE_X uses this to create capability and map its handle to descriptor and location of new digital object of type t=X
- DELETE_CAP(i)
  - delete entry CL[i]

- c = CREATE_CL( )
  - create an empty CL
  - CREATE_VM stores c into the CL slot of a newly created VM
- DELETE_CL(c)
  - delete the capability list c

- j = COPY_CAP(i,mask)
  - defined earlier
- c = MAKE_PUBLIC(i)
  - defined earlier
- j = INSERT_CL(c)
  - where c is public cap, verify and delete checksum, store in CL[j] – note that CL must store IP for public capabilities.
- REMOVE_CL(j)
  - delete CL[j] from current CL

Confinement

- Problem: confine untrusted software to a low-privilege domain.
- Solution: encapsulate the untrusted software in its own virtual machine
  - The initial stateword of the VM's thread is empty
  - The initial instruction pointer is IP=0 (assumes by convention linker-loader puts the entry of the main program at address space location 0)
  - Put in the VM's CL capabilities for only the objects (usually only files) that the software specs say it needs
- Execution errors by encapsulated software are confined to the address space and objects in the CL; no other address space or digital object is affected
- The "perfect sandbox"

Summary

- Capability addressing is “natural” for digital objects in the user kernel
- Capability grants its holder allowed access to the object pointed to by its handle
- Access code enables interface functions
- Fast mapping as in virtual memory
- Capabilities can be shared publicly
- Capabilities enable least-privilege confined software