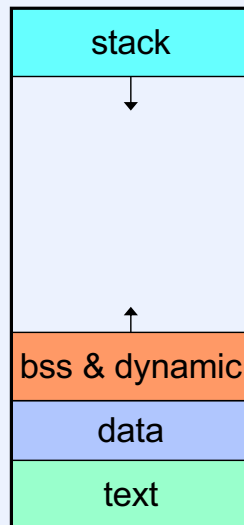


# Unix Structure (2)

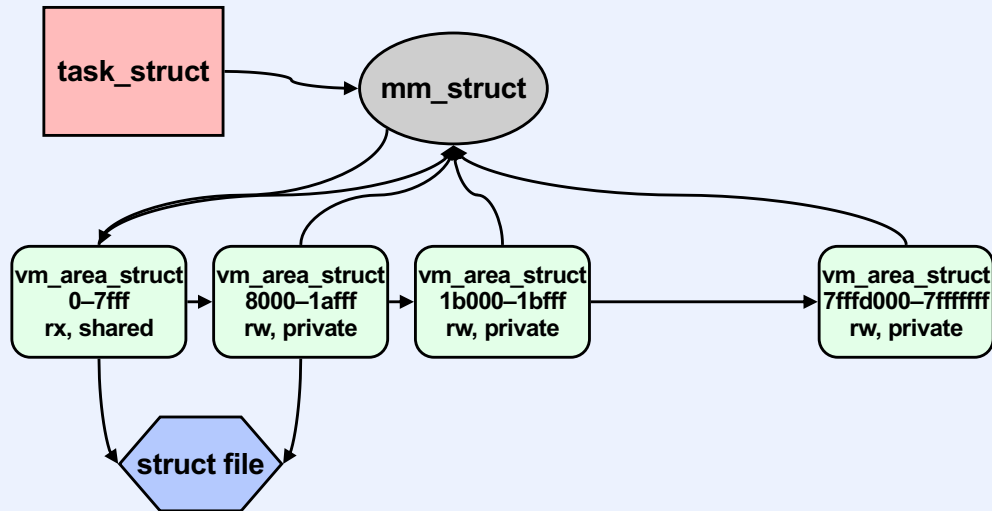
# Representing the Address Space

- Important component of a process is its address space
  - how is it represented?
- Can page tables represent a process's address space?

# Simple User Address Space

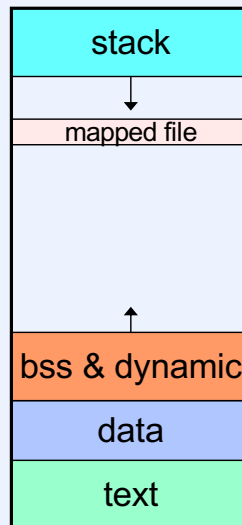


## Address-Space Representation Somewhat Simplified

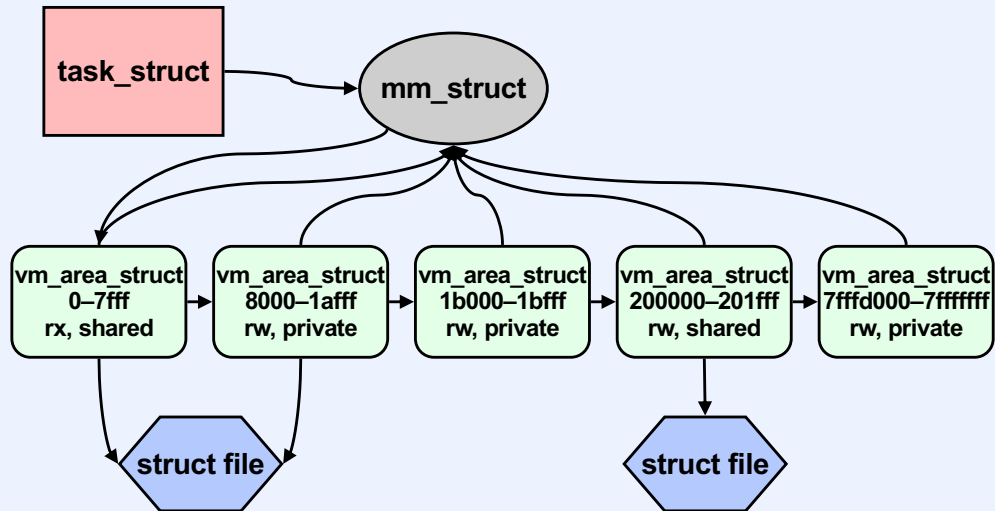


Here we have a somewhat simplified rendition of the Linux representation of a process's address space (on a 32-bit architecture). Representing the address space as a whole is the **mm\_struct**. It in turn refers to a linked list of **vm\_area\_structs**, one for each separately managed object that's mapped in. What's shown here, from left to right, are a text segment, a data segment, BSS, and the process's stack. The text and data segments get their pages from the same file. The BSS and stack segments are anonymous: when pages are first accessed, they are filled with zeroes; they have no permanent storage assigned to them. The **task\_struct** is the Linux data structure representing a process.

# Adding a Mapped File

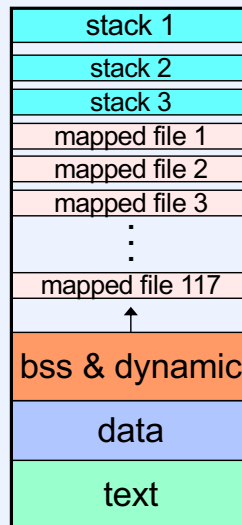


## Address-Space Representation: More Areas

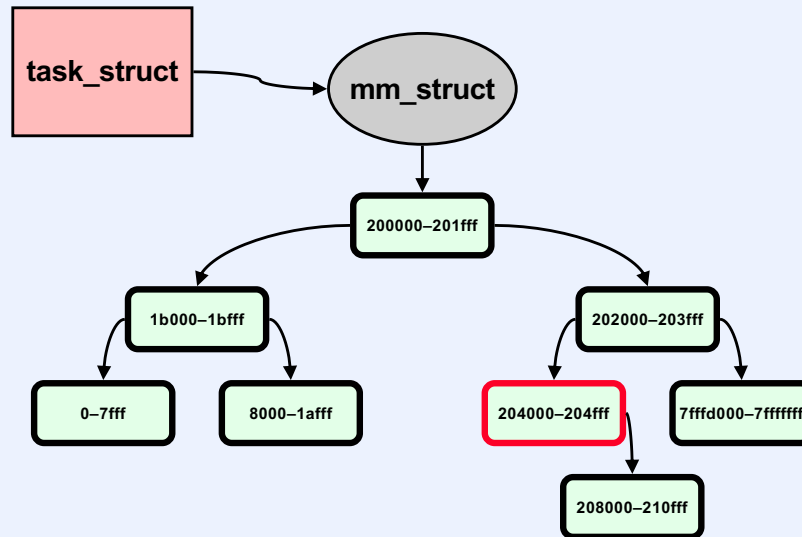


We add another **vm\_area\_struct** for the mapped file, whose pages, of course, come from the file.

# Adding More Stuff



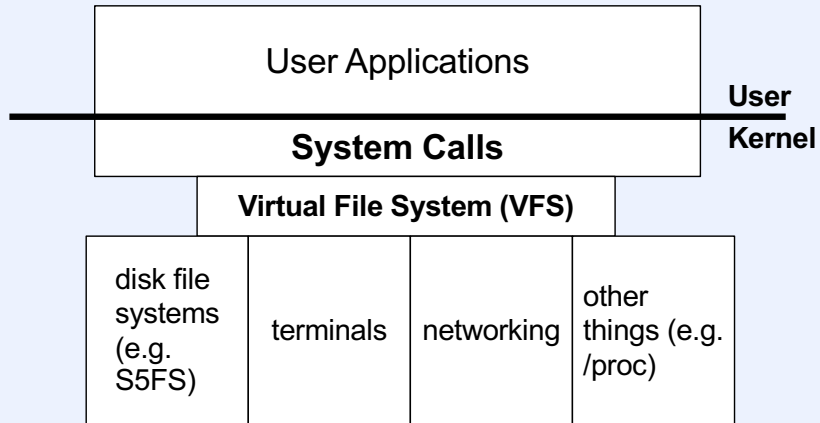
## Address-Space Representation: Reality



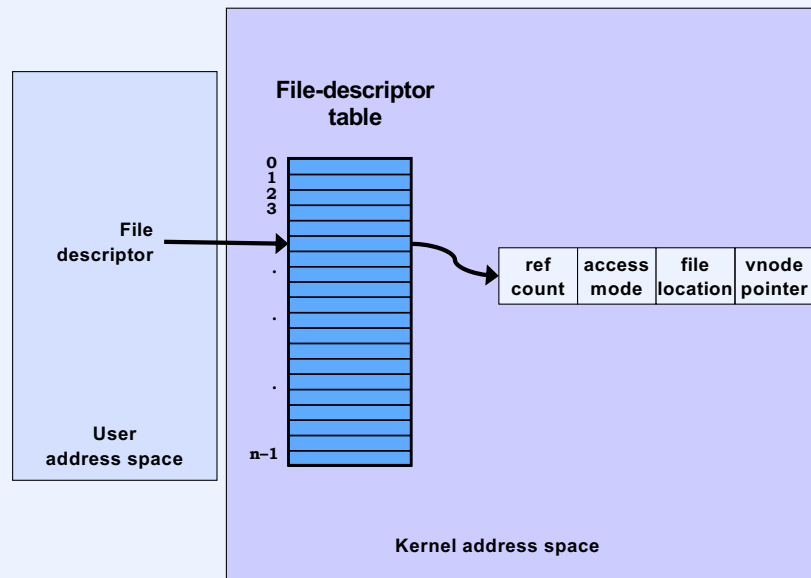
Since there could be a rather large number of regions in an address space, the actual data structure used to link the **vm\_area\_structs** is a form of balanced tree known as a **red-black tree**, in which each node is assigned a color. Its defining properties are that the root and leaves are black, that all paths from any particular node to its descendant leaves have the same number of black nodes, and that all children of red nodes are black.



# Layering



# File-Descriptor Table



A vnode is the representation of a file in the kernel. We discuss it soon.

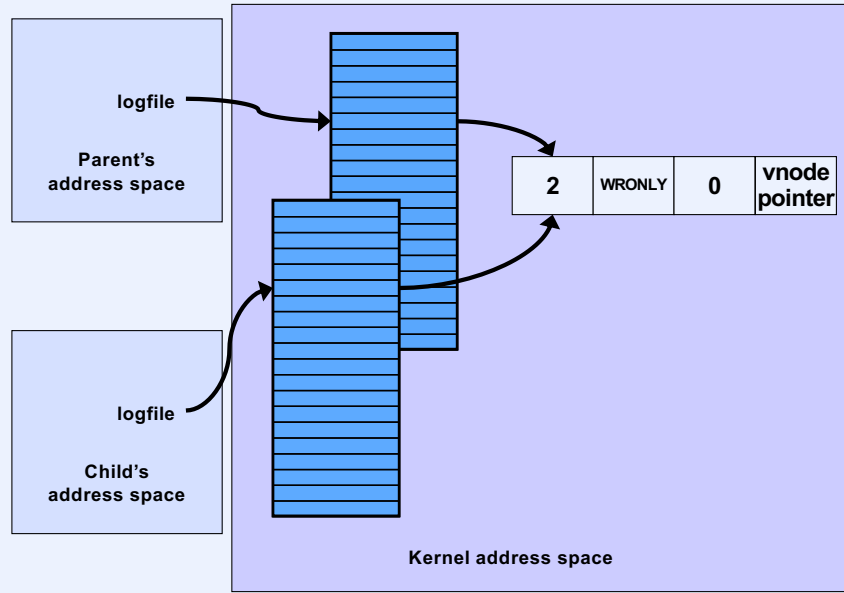
# Fork and File Descriptors

```
int logfile = open("log", O_WRONLY);
if (fork() == 0) {
    /* child process computes something, then does: */
    write(logfile, LogEntry, strlen(LogEntry));
    ...
    exit(0);
}

/* parent process computes something, then does: */

write(logfile, LogEntry, strlen(LogEntry));
...
```

# File Descriptors After Fork



## Quiz 1

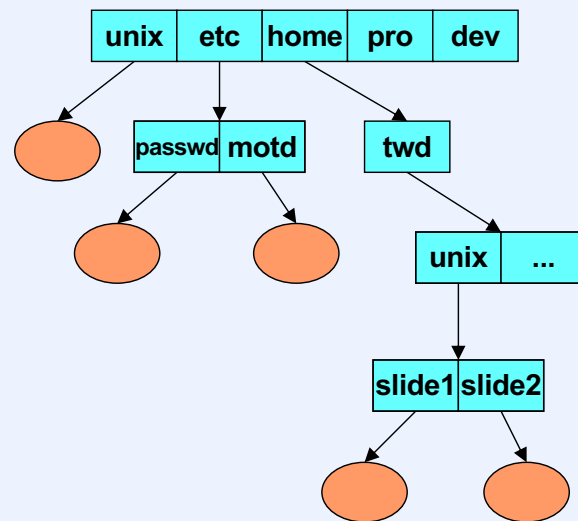
```
int fd1 = open("file", O_CREAT|O_RDWR, 0666);
unlink("file");
write(fd1, "123", 3);
int fd2 = open("file", O_CREAT|O_RDWR, 0666);
write(fd2, "4", 1);
if (fork() == 0) {
    write(fd1, "5", 1);
}
exit(0);
```

The final contents of file are:

- a) 4
- b) 45
- c) 453
- d) 12345

Assume "file" does not exist prior to execution of this code.

# Directories



Here is a portion of a Unix directory tree. The ovals represent files, the rectangles represent directories (which are really just special cases of files).

# Directory Representation

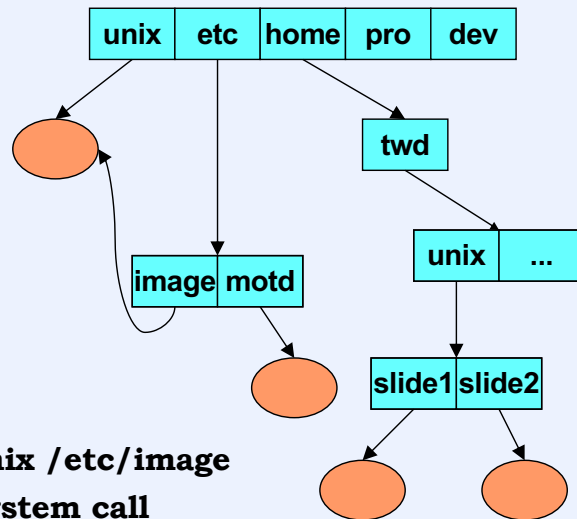
Component Name	Inode Number
----------------	--------------

directory entry

.	1
..	1
unix	117
etc	4
home	18
pro	36
dev	93

A directory consists of an array of pairs of **component name** and **inode number**, where the latter identifies the target file's **inode** to the operating system (an inode is an on-disk data structure maintained by the operating system that represents a file; the vnode we mentioned earlier is in kernel memory and contains a copy of the inode). Note that every directory contains two special entries, "." and "..". The former refers to the directory itself, the latter to the directory's parent (in the case of the slide, the directory is the root directory and has no parent, thus its ".." entry is a special case that refers to the directory itself).

# Hard Links



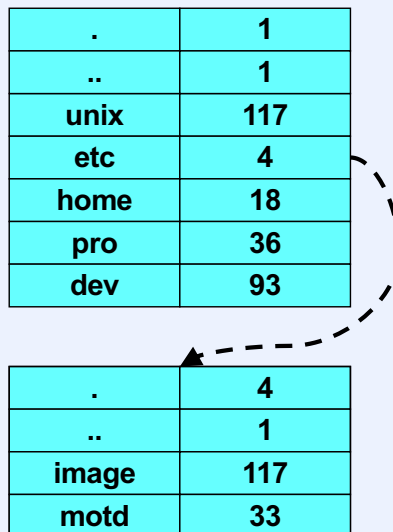
**% ln /unix /etc/image**  
**# link system call**

Here are two directory entries referring to the same file. This is done, via the shell, through the *ln* command which creates a (hard) link to its first argument, giving it the name specified by its second argument.

The shell's "ln" command is implemented using the link system call.



# Directory Representation



.	1
..	1
unix	117
etc	4
home	18
pro	36
dev	93

.	4
..	1
image	117
motd	33

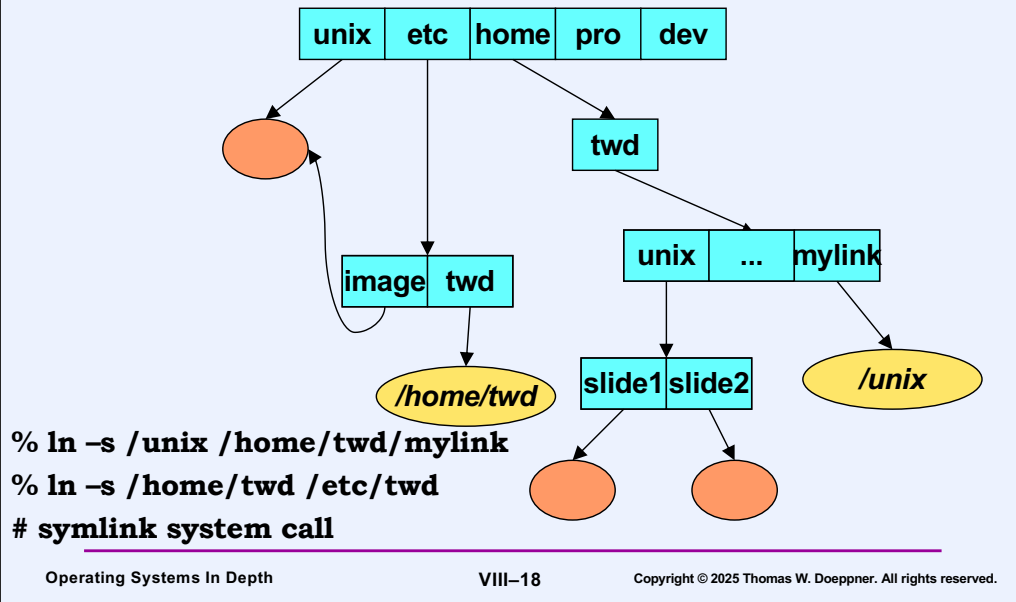
Here are the (abbreviated) contents of both the *root* (/) and */etc* directories, showing how **/unix** and **/etc/image** are the same file. Note that if the directory entry **/unix** is deleted (via the shell's "rm" command), the file (represented by inode 117) continues to exist, since there is still a directory entry referring to it. However, if **/etc/image** is also deleted, then the file has no more links and is removed. To implement this, the file's inode contains a link count, indicating the total number of directory entries that refer to it. A file is actually deleted only when its inode's link count reaches zero.

Note: suppose a file is open, i.e. is being used by some process, when its link count becomes zero. Rather than delete the file while the process is using it, the file will continue to exist until no process has it open. Thus the inode also contains a reference count indicating how many times it is open: in particular, how many system file table entries point to it. A file is deleted when and only when both the link count and this reference count become zero.

The shell's "rm" command is implemented using the **unlink** system call.

Note that **/etc/..** refers to the root directory.

# Soft Links



Differing from a hard link, a soft link (or symbolic link) is a special kind of file containing the name of another file. When the kernel processes such a file, rather than simply retrieving its contents, it makes use of the contents by replacing the portion of the directory path that it has already followed with the contents of the soft-link file and then following the resulting path. Thus referencing **/home/twd/mylink** results in the same file as referencing **/unix**. Referencing **/etc/twd/unix/slide1** results in the same file as referencing **/home/twd/unix/slide1**.

The shell's "ln" command with the "-s" flag is implemented using the **symlink** system call.

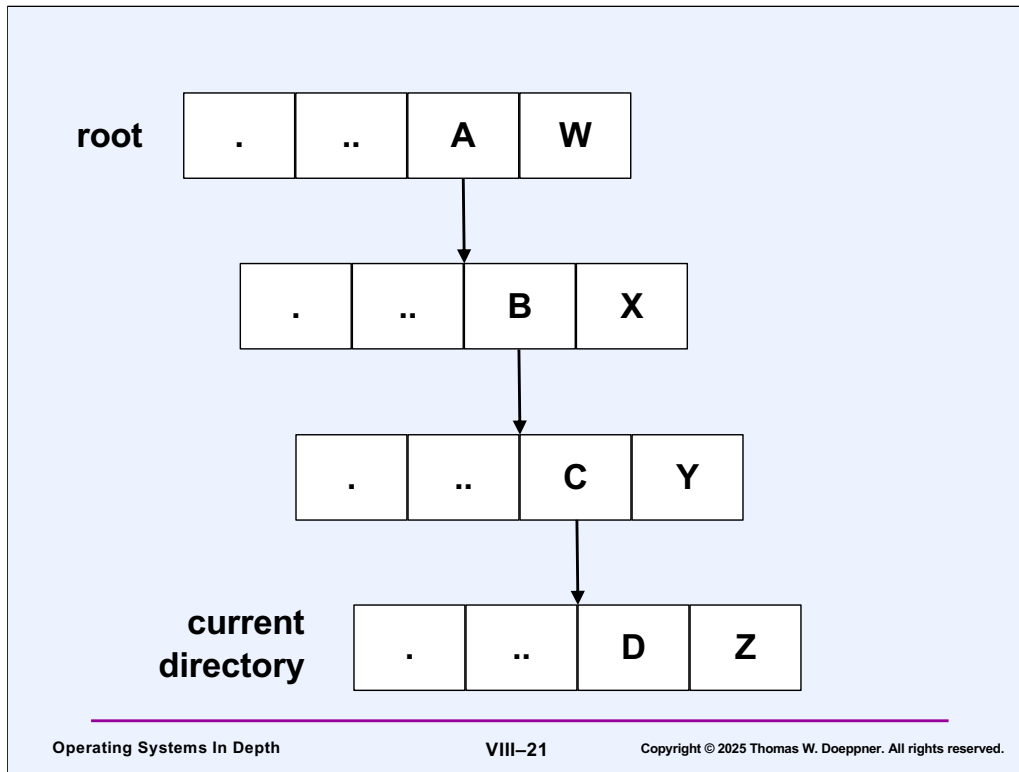
# Working Directory

- **Maintained in kernel for each process**
  - paths not starting with “/” start with the working directory
  - changed by use of the *chdir* system call
  - displayed (via shell) using “pwd”
    - how is this done?

The **working directory** is maintained (as the inode number (explained subsequently) of the directory) in the kernel for each process. Whenever a process attempts to follow a path that doesn't start with “/”, it starts at its working directory (rather than at “/”).

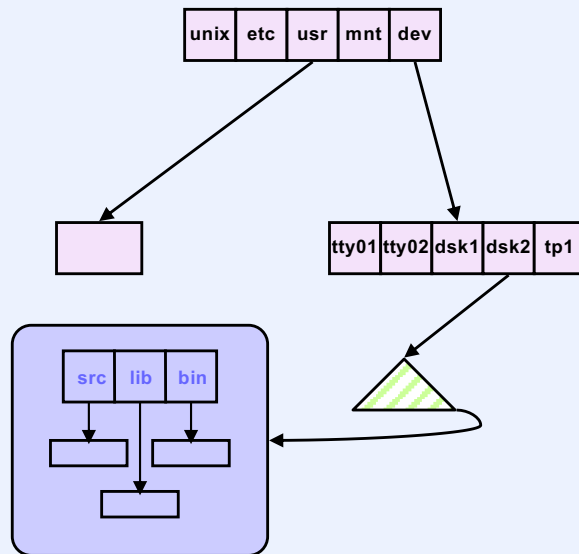
# **pwd**

- **Print Working Directory**
  - suppose the current working directory is **/A/B/C**
  - how can a program determine it?



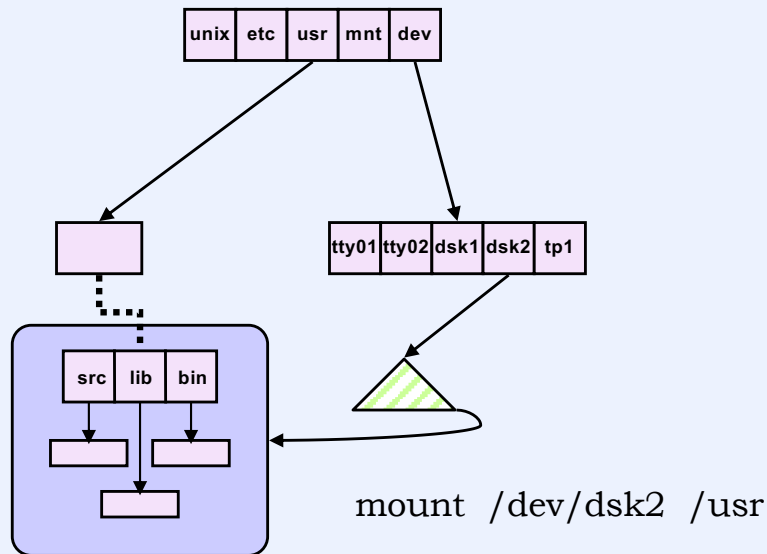
All we know about the current directory is that if we read the directory's "." entry, we'll get the directory's inode number (see slide VIII-15). If we open the file "..", we'll be opening the directory's parent directory. We can then search that directory for the entry containing the parent's inode number. The other half of that entry will contain the component name of the directory within its parent (which, in our example, is C). We can continue this all the way back to the root (where both the "." and ".." entries contain the same inode number).

# Mount Points (1)



A special file representing a disk does not provide access to the file system on the disk, but provides access to the disk as a device (the disk appears as if it were a single fixed-length file).

## Mount Points (2)



To access a disk as a container for a file system, the directories within the disk must be somehow connected to our system-wide directory hierarchy. I.e., we must be able to follow a path, starting from the root, that leads into the directory hierarchy of the disk's file system. We need more machinery than what we've seen up to now, since directory entries refer only to inodes within the file system containing the directory itself. The new mechanism is the notion of **mounting**, i.e., superimposing the root of a file-system's directory hierarchy on top of some other directory that is already accessible from the root of the system-wide hierarchy. Thus paths that lead into the mounted-on directory reach the root of the file-system hierarchy instead. In the example of the slide, we first execute the **mount** command, which causes the root of the file system in `/dev/dsk2` to be superimposed on the directory `/usr`. Thus we can continue the path to `/usr/lib`. This is accomplished via some magic in the kernel, as we'll soon see. The directory entry for "usr" (in the root directory), still contains the inode number of the original `/usr` directory (the one which has been mounted upon). This directory still exists; however, while it is mounted upon, its contents (and the files they refer to) are inaccessible (unless linked to via some other, accessible, directory).

Note that in S5FS file systems, each file system has its own inode space. Thus inode `x` on one file system refers to a particular file in that file system, while inode `x` on another file system refers to a particular file on that file system.

# Representing File Systems

```
class fs {  
    char dev[STR_MAX];           // device containing the f.s.  
    char mountpt[STR_MAX];       // where the f.s. is mounted  
    vnode *vnodecovered;         // file on which f.s. is mounted  
    vnode *root;                 // root of the f.s.  
    virtual void read_vnode(vnode *);  
    virtual void delete_vnode(vnode *);  
};
```

Here we use C++, since what we're showing is essentially an object with inheritance. Note that a virtual function (referred to as overrides in some languages) is something that is made concrete in a subclass. Thus, in our case, a particular kind of file system (a subclass) would provide definitions for the virtual functions.

To represent a file system, regardless of which sort, we use an instance of the **fs** class. The virtual functions must be instantiated: they are functions that initialize and delete **vnodes**: objects representing individual files in the kernel.

Up to now we've used **inodes** to refer to individual files. An **inode** is the actual data structure used for S5FS files. A **vnode** is an abstraction of an **inode**, and can be used to represent a file in any sort of file system.



# Representing Files

```
class vnode {  
    unsigned short refcount;  
    fs *vfsmounted;  
    fs *vfs;  
    unsigned long vno;  
    int mode;  
    int len;  
    link_list_t link;  
    kmutex_t mutex;  
    virtual int create(const char *, int, vnode **);  
    virtual int read(int, void *, int);  
    virtual int write(int, const void *, int);  
    ...  
    class mobj mobj;  
};
```

The **vnode** class is instantiated to represent individual files. The member **vfsmounted** is used if this vnode represents a directory on which the root of another file system is mounted; it refers to the mounted file system. The member **vfs** refers to the file system containing the file represented by this vnode. The virtual functions **create**, **read**, and **write** are just a few of the functions that must be implemented for files of any particular file system. See the VFS assignment handout for a complete list. The **class mobj** is used to access the contents of the file. We'll start to discuss it later in this lecture, but will cover it in detail later in the course.

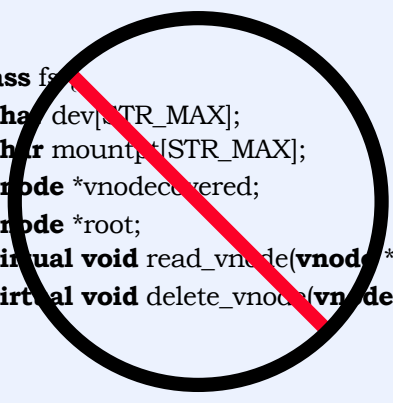
## **But Wait ...**

- **What's this about C++?**
  - real operating systems are written in C ...

We've been describing all this in terms of C++ classes, but we aren't going to be using C++ in our operating systems—we are using C, which has no classes.

## fs

```
class fs {  
    char dev[STR_MAX];  
    char mountpt[STR_MAX];  
    vnode *vnodecovered;  
    vnode *root;  
    virtual void read_vnode(vnode *);  
    virtual void delete_vnode(vnode *);  
};
```



```
typedef struct fs {  
    char fs_dev[STR_MAX];  
    char fs_mountpt[STR_MAX];  
    struct vnode *fs_vnodecovered;  
    struct vnode *fs_root;  
    fs_ops_t *fs_op;  
    /* function pointers */  
    void *fs_i;  
    /* extra stuff in subclasses */  
} fs_t;
```

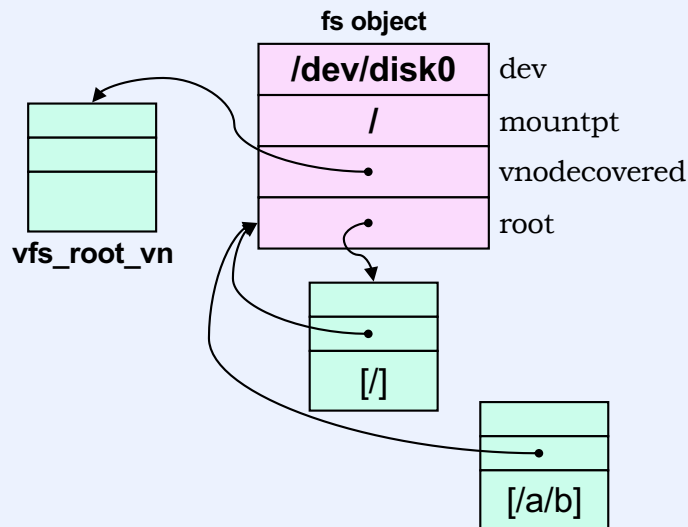
We've got to implement classes using C typedefs and structs.

# vnode

```
class vnode {  
    unsigned short refcount;  
    fs *vfsmounted;  
    fs *vfs;  
    unsigned long vno;  
    int mode;  
    int len;  
    link_list_t link;  
    kmutex_t mutex;  
    virtual int create(const char *, int,  
        vnode **);  
    virtual int read(int, void *, int);  
    virtual int write(int, const void *,  
        int);  
    ...  
    class mobj mobj;  
};
```

```
typedef struct vnode {  
    unsigned short vn_refcount;  
    struct fs *vn_vfsmounted;  
    struct fs *vn_vfs;  
    unsigned long vn_vno;  
    int vn_mode;  
    int vn_len;  
    link_list_t vn_link;  
    kmutex_t vn_mutex;  
    struct vnode_ops *vn_op;  
    /* function pointers */  
    struct mobj mobj;  
    void *vn_i;  
    /* extra stuff in subclasses */  
} vnode_t;
```

## Mounting a File System (1)

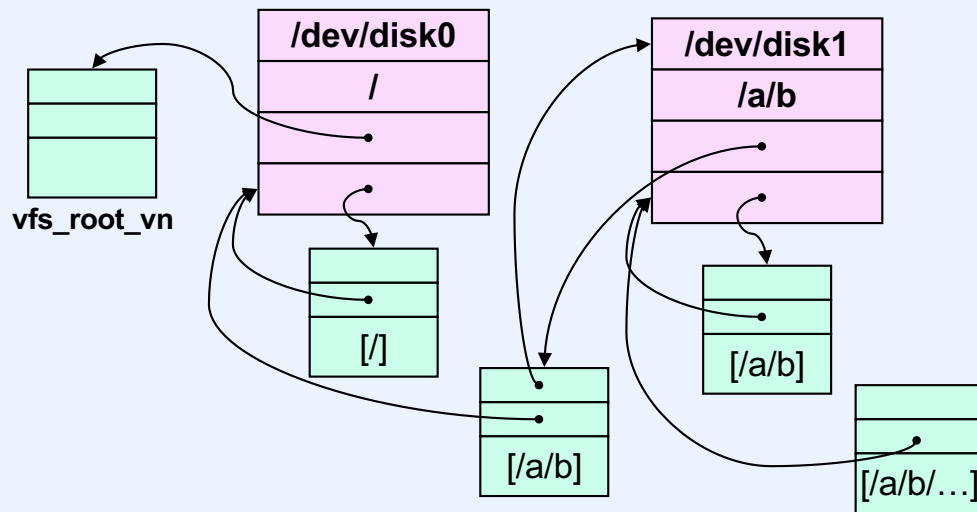


Here we have the fs object for the root file system (i.e., the one containing the root directory of the complete naming hierarchy), as well as the vnodes for two of its files: the root directory itself, as well as another directory, **/a/b**. Though the root directory is not mounted on another directory of some other file system, its **vnodecovered** member points to a special vnode, called **vfs\_root\_vn**.

Note that the path names in square brackets aren't really stored in the vnode objects – they're in the diagram just to help us see what's going on.

Also, note that these data structures are in the kernel – user programs have no direct access to them.

## Mounting a File System (2)



Here we've mounted another file system on the directory `/a/b`.

## Quiz 2

- Suppose the current working directory is `/A/B/M`. `/A/B` is a path in file system 1. `/M` is a directory in file system 2. File system 2 is mounted on file system 1 at `/A/B`. How does the `pwd` command handle this case?
  - a) it's easy: the `“..”` entry in `/A/B/M` refers to the directory `/A/B` and one simply continues backwards towards the root as if there were no mount point
  - b) a special system call is required to determine the path
  - c) one must start at the root and continue forwards towards the working directory

Hint: keep in mind that **fs** and **vnode** data structures are in the kernel, not accessible to user programs. Also, each file system has its own space of **inode** numbers.

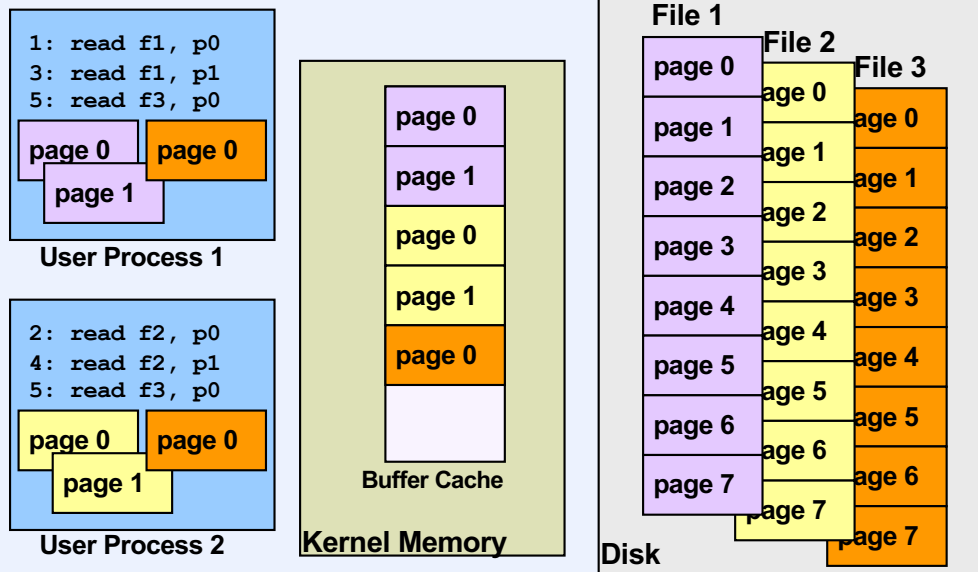
## Digression (Sort of)

- Weenix has two file systems
  - ramfs
    - trivial in-memory file system for testing purposes
    - caching of blocks isn't necessary
  - s5fs
    - non-trivial file system that you implement
    - caching of blocks is important
    - caching uses virtual-memory subsystem so that *mmap* works right

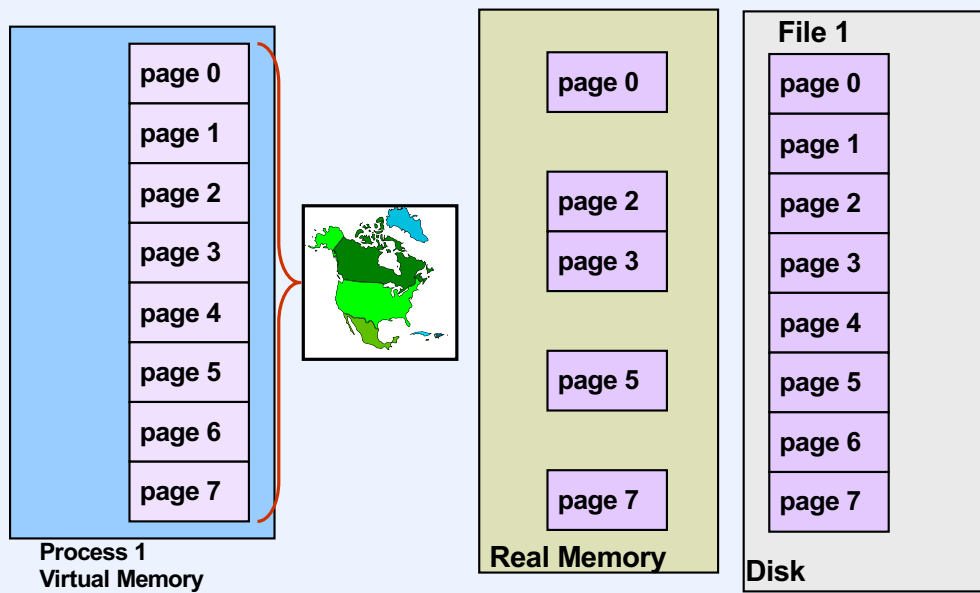
We provide the ramfs file system – you don't have to write it.



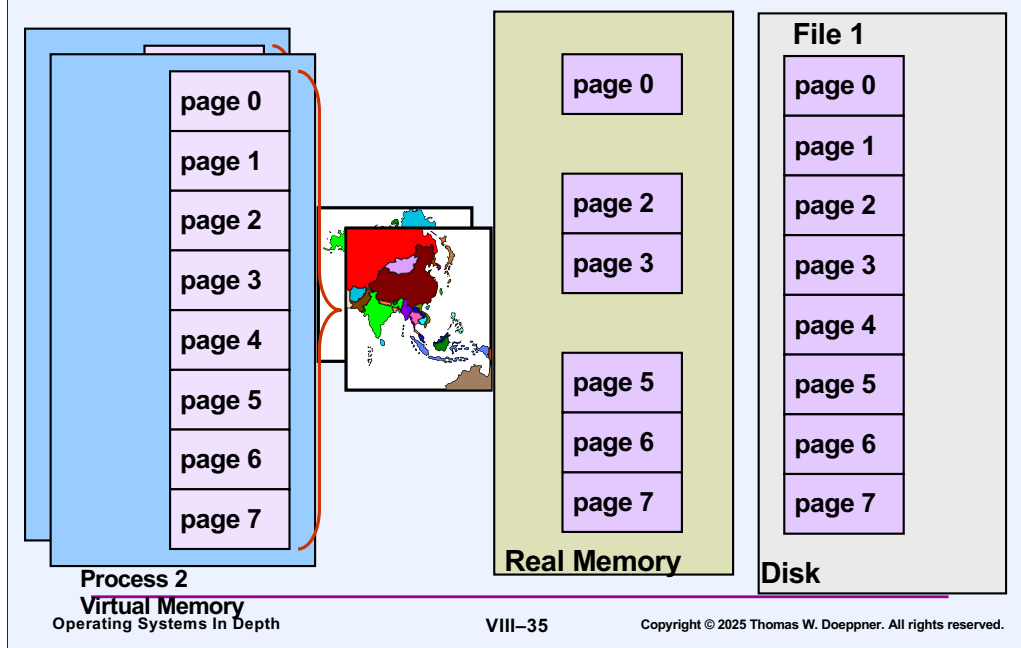
# Traditional I/O



# Mapped File I/O

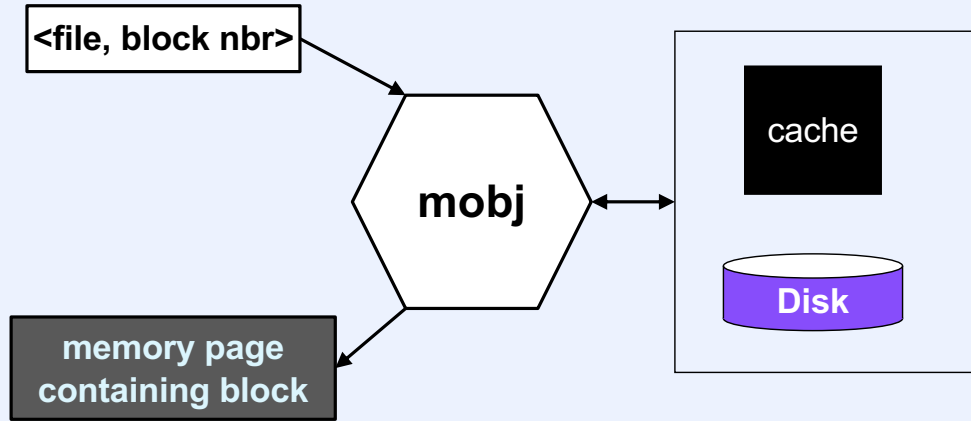


## Multi-Process Mapped File I/O



Note that it's important that mapped file I/O coexist with traditional I/O, so that, say, if a mapped file is modified, a thread reading it via the traditional interface will correctly read the modified blocks.

# Memory Objects



A memory object (mobj) is an abstraction implemented in the OS kernel that, given a block number within a file, finds the block (in either the cache or the on-disk file system) and returns a copy of it. How it works is, of course, very dependent on the virtual-memory subsystem and the type of file system.