This lecture is covered in Section 4.1 of the textbook.
A Unix process’s address space appears to be three regions of memory: a read-only text region (containing executable code); a read-write region consisting of initialized data (simply called data), uninitialized data (BSS—a directive from an ancient assembler (for the IBM 704 series of computers), standing for Block Started by Symbol and used to reserve space for uninitialized storage), and a dynamic area; and a second read-write region containing the process’s user stack (a standard Unix process contains only one thread of control).

The first area of read-write storage is often collectively called the data region. Its dynamic portion grows in response to sbrk system calls. Most programmers do not use this system call directly, but instead use the malloc and free library routines, which manage the dynamic area and allocate memory when needed by in turn executing sbrk system calls.

The stack region grows implicitly: whenever an attempt is made to reference beyond the current end of stack, the stack is implicitly grown to the new reference. (There are system-wide and per-process limits on the maximum data and stack sizes of processes.)
The only way to create a new process is to use the `fork` system call.
By executing `fork` the parent process creates an almost exact clone of itself which we call the child process. This new process executes the same text as its parent, but contains a copy of the data and a copy of the stack. This copying of the parent to create the child can be very time-consuming. We discuss later how it is optimized.

Fork is a very unusual system call: one thread of control flows into it but two threads of control flow out of it, each in a separate address space. From the parent’s point of view, fork does very little: nothing happens to the parent except that fork returns the process ID (PID — an integer) of the new process. The new process starts off life by returning from fork. It always views fork as returning a zero.
Fork and Wait

```
short pid;
if ((pid = fork()) == 0) {
    /* some code is here for the child to execute */
    exit(n);
} else {
    int ReturnCode;
    while (pid != wait(&ReturnCode))
        ;
    /* the child has terminated with ReturnCode as its return code */
}
```
Note that in Linux, the “process control block” is known as the “task_struct”.
int pid;
if ((pid = fork()) == 0) {
    /* we'll soon discuss what might take place before exec is called */
    execl("/home/twd/bin/primes", "primes", "300", 0);
    exit(1);
}
/* parent continues here */

while(pid != wait(0))    /* ignore the return code */
    ;
Most of the time the purpose of creating a new process is to run a new (i.e., different) program. Once a new process has been created, it can use the `exec` system call to load a new program image into itself, replacing the prior contents of the process’s address space. Exec is passed the name of a file containing a fully relocated program image (which might require further linking via a runtime linker). The previous text region of the process is replaced with the text of the program image. The data, BSS and dynamic areas of the process are “thrown away” and replaced with the data and BSS of the program image. The contents of the process’s stack are replaced with the arguments that are passed to the main procedure of the program.
Quiz 1

```c
int A, B, C, D;
A=1;
if (fork() > 0) {
    B=1;
    A=111;
} else {
    C=2;
    if (fork() > 0) {
        D=222;
    } else {
        D=A+B+C;
        // what value is now
        // in D for this process?
    }
}
exit(0);
```

Answer:

a) 0
b) 3
c) 113
d) indeterminate
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

- stack
- bss & dynamic
- data
- text
Here we have a somewhat simplified rendition Linux representation of a process’s address space. 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

- stack
- mapped file
- bss & dynamic
- data
- text
We add another `vm_area_struct` for the mapped file, whose pages, of course, come from the file.
Adding More Stuff

- stack 1
- stack 2
- stack 3
- mapped file 1
- mapped file 2
- mapped file 3
- mapped file 117
- bss & dynamic
- data
- text
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.
Quiz 2

```c
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) 12345
- b) 453
- c) 45
- d) 4
File-Descriptor Table

Kernel address space

User address space

File descriptor

File-descriptor table

ref
count
access
mode
file
location
inode
pointer

0
1
2

n-1
Allocation of File Descriptors

- Whenever a process requests a new file descriptor, the lowest numbered file descriptor not already associated with an open file is selected; thus

```
#include <fcntl.h>
#include <unistd.h>

close(0);
fd = open("file", O_RDONLY);
```

- will always associate file with file descriptor 0 (assuming that the open succeeds)

One can depend on always getting the lowest available file descriptor.
Redirecting Output ... Twice

if (fork() == 0) {
    /* set up file descriptors 1 and 2 in the child process */
    close(1);
    close(2);
    if (open("/home/twd/Output", O_WRONLY) == -1) {
        exit(1);
    }
    if (open("/home/twd/Output", O_WRONLY) == -1) {
        exit(1);
    }
    exec("/home/twd/bin/program", "program", 0);
    exit(1);
}

/* parent continues here */
Redirected Output

File-descriptor table

User address space

Kernel address space

File descriptor 1

File descriptor 2

1  WRONLY  0  inode pointer

1  WRONLY  0  inode pointer
Redirected Output After Write

File descriptor 1

File descriptor 2

User address space

Kernel address space

File-descriptor table

1 WONLY 100 inode pointer

1 WONLY 0 inode pointer
Sharing Context Information

```c
if (fork() == 0) {
    /* set up file descriptors 1 and 2 in the child process */
    close(1);
    close(2);
    if (open("/home/twd/Output", O_WRONLY) == -1) {
        exit(1);
    }
    dup(1); /* set up file descriptor 2 as a duplicate of 1 */
    execl="/home/twd/bin/program", "program", 0);
    exit(1);
}
/* parent continues here */
```
Redirected Output After Dup

File descriptor 1
File descriptor 2

User address space

File-descriptor table

Kernel address space

inode pointer

2 WONLY 100
Fork and File Descriptors

```c
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

- Parent's address space
  - logfile
- Child's address space
  - logfile
- Kernel address space
  - Inode pointer
  - 2
  - WRONLY
  - 0
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).
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 data structure maintained by the operating system that represents a file). 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).
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.
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.
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.
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 “/”).
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).
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, the path “/usr”, when followed after the mount command shown is executed, actually leads to the root of the file system contained within /dev/dsk2. 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).
To represent a file system, regardless of which sort, we use an instance of the `fs` class. The virtual functions must be instantiated and are routines that initialize and delete `vnodes`: objects representing individual files in the kernel.
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.
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 vnodedcovered member points to a special vnode, called vfs_root_vn.
Here we’ve mounted another file system on the directory /a/b.
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.
We've got to implement classes using C typedefs and structs.

```c
typedef struct fs {
    char fs_dev[STR_MAX];
    char fs_mountpt[STR_MAX];
    vnode *fs_vnodecovered;
    vnode *fs_root;
    virtual void read_vnode(vnode *);
    virtual void delete_vnode(vnode *);
} fs_t;
```
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);
    ...
};

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 */
    void *vn_i;
    /* extra stuff in subclasses */
}