IC221: Systems Programming (SP16)

Lec 21: File System, Kernel Data Structures, and Open Files

1 Review: What is a File System?

Recall, that a **file system** is an organization of files into directories and folders. There are many different types of file systems and many different implementations, which is closely linked to the type of storage device which contains the data, e.g., a hard disk versus a thumb drive versus a CDrom, and the operating systems, e.g., Mac, Linux, or Windows.

The purpose of a file system is to maintain and organize **secondary storage**. As opposed to RAM which is volatile and does not persist across reboots of the computer, secondary storage is designed to permanently store data. The file system provides a convenient representation of the data layout which is maintained by the operating system. There are wide varieties of file system implementations that describe different ways to organize the meta-data necessary for maintaining a file system. For example, the standard file system type on Windows is called NTFS, which stands for Windows NT Files System, and the standard file systems for Linux/Unix systems is called ext3 or ext4 depending on the version.

An O.S. has a root file system where primary data and system files are stored. On Unix systems, this is the base file system, indicated by a single / at the root. On Windows, this is commonly called the C:\ drive. The O.S. can also **mount** other file systems, like plugging in a USB drive or a inserting a CD-ROM, and the user can gain access to these other file systems through file exploration, which can use different layouts and data organizations, for example FAT32 on a USB drive or ISO 9660 on a CD rom.

However, from a system's programming perspective, the programs we write are agnostics to the underlying implementation of the file system; we open a file with `open()` and the read from the file with `read()` and write with `write()`. The details of the implementation are completely transparent. How does the O.S. maintain this illusion? That is the topic of these next lessons — we will explore the file system implementation details support the programs we've been writing so far.

2 Kernel Data Structures

To understand the file system, you must first think about how the kernel organizes and maintains information. The kernel does a lot of bookkeeping; it needs to know which process are running, what their memory layout is, what open files processes have, and etc. To support this, the kernel maintains three important tables/structures for managing open files for processs: the process table, the file table, and the v-node/i-node info.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

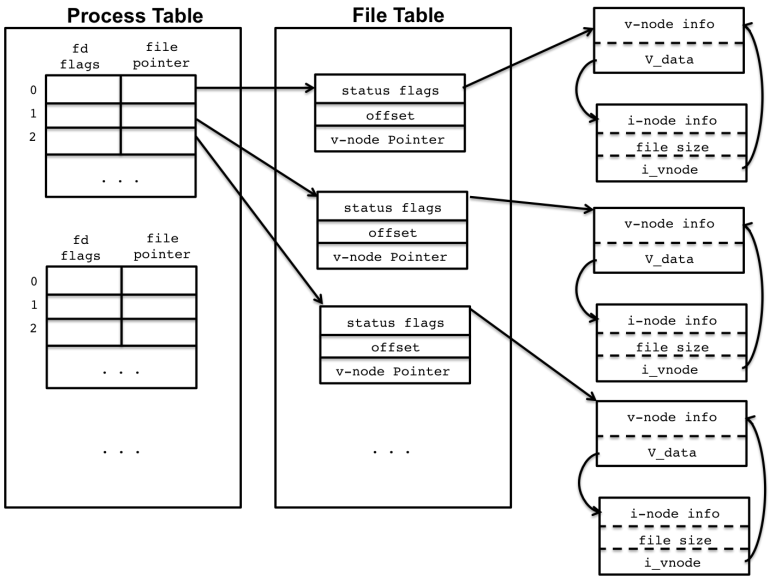


Figure 1: Kernel File System Data Structures

We'll go through each of these parts individually below.

2.1 Process Table

The first data structure is the process table, which stores information about all currently running processes. This includes information about the memory layout of the process, current execution point, and etc. Also included is the open file descriptors.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

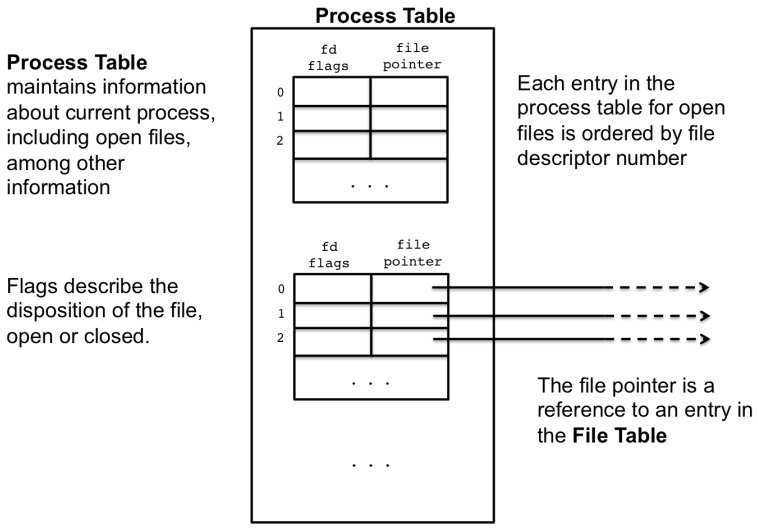


Figure 2: The Process Table

As we know, all process start with with three standard files descriptors, 0, 1, 2, and these numbers and other file descriptors are indexes and stored in the open file table for that process's entry in the process table. Every time a new file is open, an new row in the open file table is added, indexed at the value of the file descriptor, e.g., 3 or 4 or 5 or etc.

Each row in the open file table has 2 values. The first are the flags, which describe disposition of the file, such as being open or closed, or if some action should be taken with the file when it is closed. The second value is a reference to the file table entry for that open file, which is a global list, across all process, of currently opened files.

One interesting note about the process table entries is that whenever a process forks to create a child, the entire process table entry is duplicated, which includes the open file entries and the their file pointers. This is how two process, the parent and the child, can share an open file. We saw examples of this earlier, and we'll look at it again later in the lesson.

2.2 File Table

Whenever a new file is open, system wide, a new entry in the global file table is created. These entries are shared amongst all process, for example when a file is opened by two different process, they may have the same file descriptor number, e.g., 3, but each of the file descriptors will reference a different entry in the file descriptor table.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

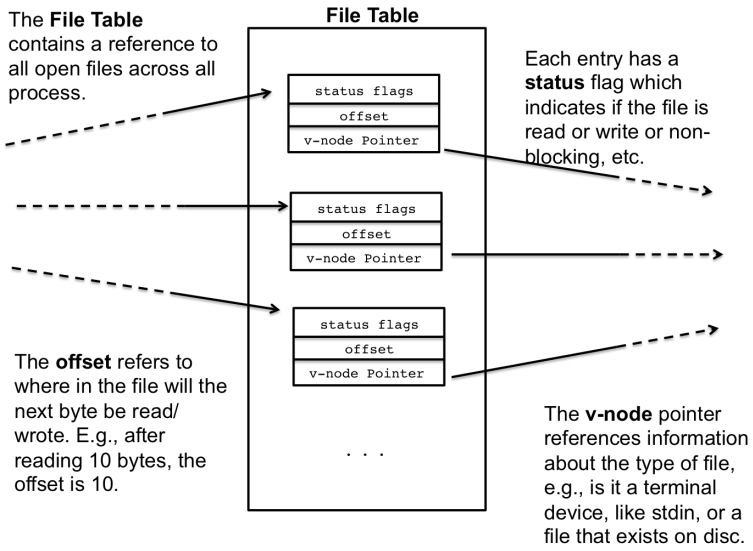


Figure 3: The File Table

Each file table entry contains information about the current file. Foremost, is the status of the file, such as the file read or write status and other status information. Additionally, the file table entry maintains an **offset** which describes how many bytes have been read from (or written to) the file indicating where to read/write from next. For example, when a file is initially opened for reading, the offset is 0 since nothing has been read. After reading 10 bytes, the offset has been moved forward to 10 since 10 bytes have been read from the file. This is the mechanisms that allows program to read a file in sequence. Later, we'll see how we can manipulate this offset and read from different parts of the file.

The last entry in the table is a **v-node pointer** which is a reference to a virtual node (v-node) and index node (i-node). These two nodes contain information about how to read the file.

2.3 V-node and I-node Tables

The v-nodes ad i-nodes are references to the file's file system disposition and underlying storage mechanisms; it connects the software with hardware. For example, at some point the file being opened will need to touch the disc, but we know that there are different ways to encode data on a disc using different file systems. The v-node is an abstraction mechanism so that there is a unified way to acces this information irrespective of the underlying file system implementation, while the i-node stores specific access information.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

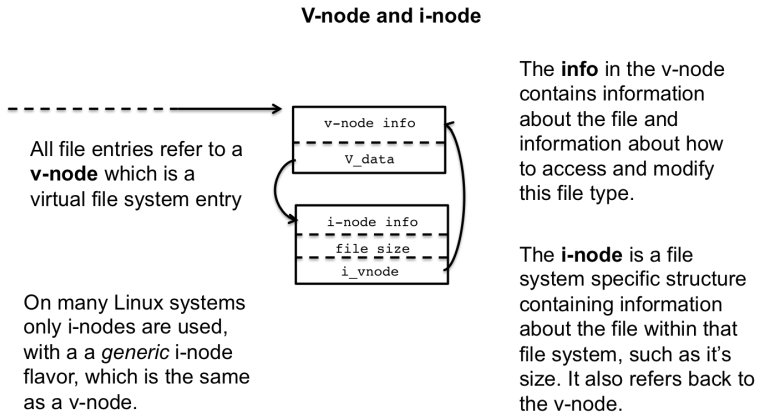


Figure 4: V-Nodes and I-Nodes

Another way to view the distinction between v-nodes and i-nodes is that a v-node is like a file as it exists within the file system. Abstractly, it could be anything and stored on any device — it could even not be a file, like `/dev/urandom` or `/dev/zero`. An i-node, conversely, describes *how* that file should be accessed, including which device is it stored on and the device specific read/write procedures.

On Linux and many Unix systems, v-nodes are not used explicitly, but rather there is **only** i-nodes; however, i-nodes can serve dual purpose. An i-node can be both a generic abstract representation of file, like a v-node, and also store device specific instructions. We'll stick to the v-node/i-node distinction for this discussion because it simplifies many of the concepts.

3 Reviewing Open Files

Now that we have a better appreciation for the kernel data structures let's review some of the common uses of file descriptors and how this matches our understanding of the kernel data structures.

3.1 Standard File descriptors

The standard file descriptors are created by `getty` and are associated with a terminal device driver, but we use the standard file descriptors like other open files, with `read()` and `write()`. They must have entries in the process table and the file table as well as v-nodes.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

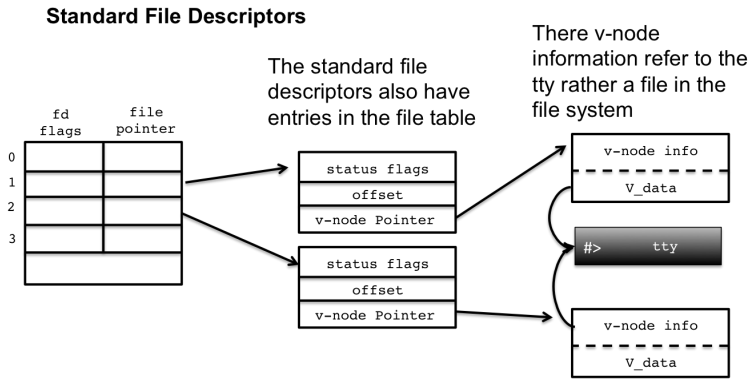
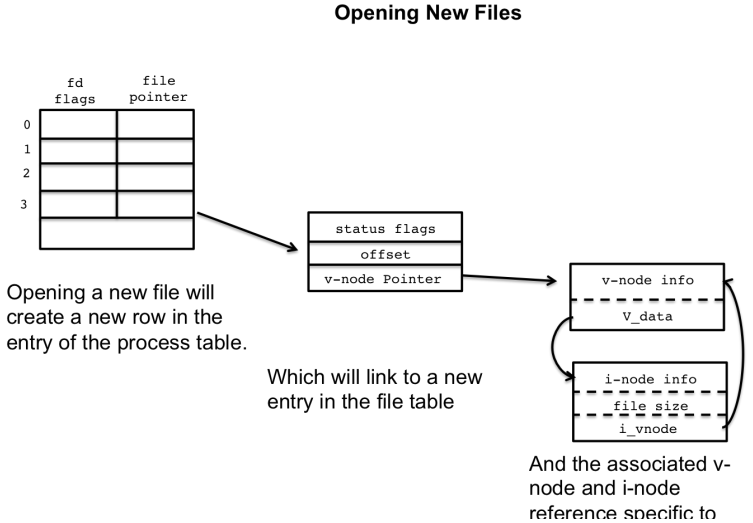


Figure 5: Standard File Descriptors

However, the standard file descriptors are not files on disc, but rather associated with a different device, the terminal device. Which means the v-node entry refers the terminal device i-node which stores the underlying access-functions that enable a user to read and write from the terminal.

3.2 Opening a New File

When we open a new file with open() a new file descriptor is generated, usually one more than the last file descriptor. A new entry in the file descriptor table is also provided, indexed at the file descriptor number, and a new entry is created in the open file table is generated.



Create PDF in your applications with the Pdfcrowd HTML to PDF API
PDFCROWD

where that file lives

Figure 6: Opening a New File

If this file exists on disc, the file table entry will reference a v-node that references i-node information that can read/write data from disc or the specific device that the file is stored on.

3.3 Sharing Files Between Parent and Child

When a process forks, the entire process table entry is duplicated, including all the open file descriptors.

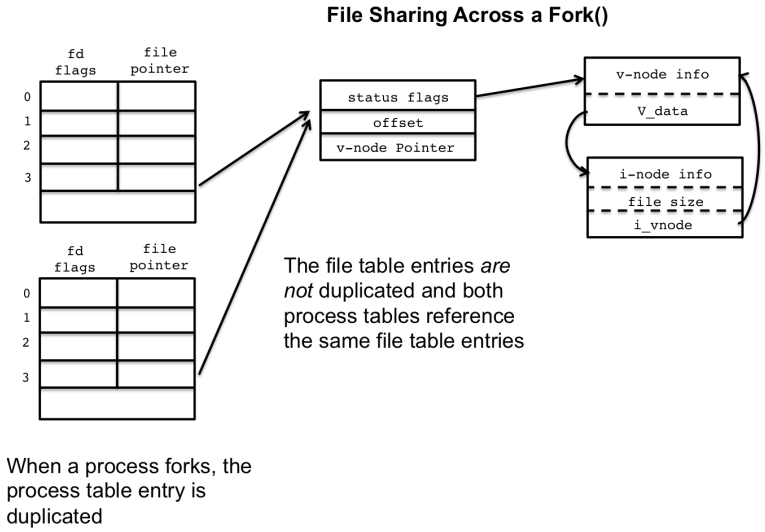


Figure 7: File Sharing

But, there is no duplication in the file table entries. The file descriptors in both parent and child reference the same file table entry. Note that the file offset is stored in the file table entry, so when one process reads from the file, it moves the offset, and when the other process reads from the file, it will start reading from where the first process left off. This explains the following program we looked at in earlier lessons:

```
int main(int argc, char * argv[]){

 int status;
 pid_t cpid, pid;

 int i=0;

 while(1){ //loop and fork children

   cpid = fork();
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
    if( cpid == 0 ){
      /* CHILD */

      pid = getpid();

      printf("Child: %d: i:%d\n", pid, i);

      //set i in child to something differnt
      i *= 3;
      printf("Child: %d: i:%d\n", pid, i);

      _exit(0); //NO FORK BOMB!!!
    }else if ( cpid > 0){
      /* PARENT */

      //wait for child
      wait(&status);

      //print i after waiting
      printf("Parent: i:%d\n", i);

      i++;
      if (i > 5){
        break; //break loop after 5 iterations
      }

    }else{
      /* ERROR */
      perror("fork");
      return 1;
    }

    //pretty print
    printf("--------------------\n");
  }

  return 0;
}
```

Both parent and child, while having different process table entries, share a file table entry. Each successive read in the child avances the offset in the entry, and the same occurs in the parent. The result is that the parent and its children alternate between reading each byte in the file.

3.4 Duplicating Files

We've also looked at file duplication with `dup2()`, and that also has a representation within the kernel data structures. Recall that a `dup2()` will duplicate one file descriptor onto another.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

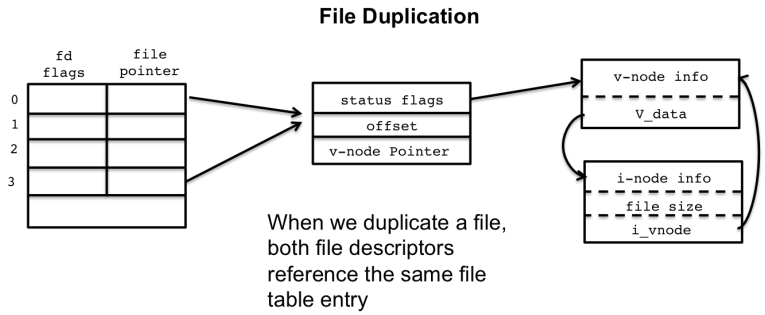


Figure 8: File Descriptor Duplication

Within the kernel data structures, this means that two entries in the file descriptor table reference the same file table entry. As a result, a read and write to either of the file descriptor is the same, as in, they reference the same file.

3.5 Pipes

Pipes are more like the standard file descriptors as they do not refer to files within the file system, but are rather kernel data structures themselves used to pipe data between the write and read end of the pipe.

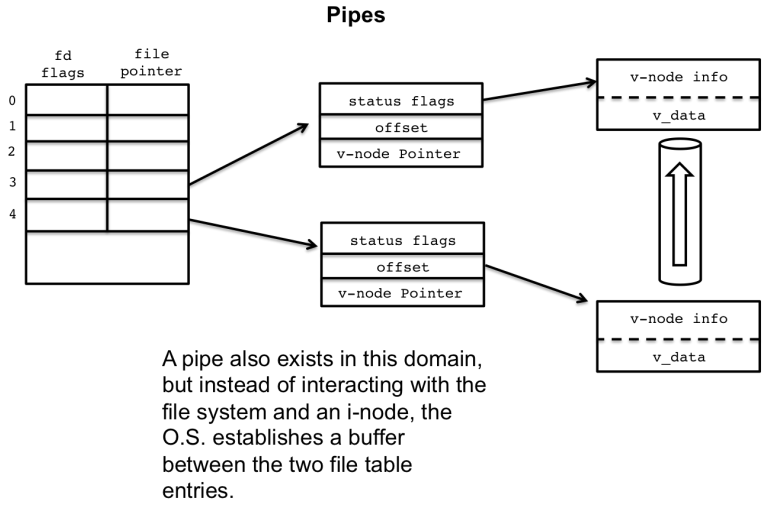


Figure 9: Pipes

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

A call to `pipe()` will create two file descriptors, a read and write file descriptor for either end of the pipe. Each of these file descriptors will have entries in the file table, but there v-node entries are linked via the kernel buffer.

4 Mounting File Systems under a Virtual File System

As you can see from the examples above, that the kernel data structures provide a lot of flexibility for working with a variety of files in different situations. The abstraction of the v-node and i-node is key: irrespective of the underlying implementation and data layout, from the v-node up, we can use a single interface while the i-node provides device specific information. This allows the file system to mix up different v-nodes under a single interface even when each of the files a v-node represents may exist on varied devices. This process is called mounting; merging multiple devices and their file systems into a unified file system under a unified abstraction.

5 Mounts

A file system in unix, while its convenient to think of it a singular thing under the root `/`, it is actually a conglomeration of multiple file systems. Each file system is mounted into the root file system at **mount points** and is treated differently under the covers by the kernel and i-nodes, but from an systems programmer perspective, each of the different file systems is treated the same because we have the abstraction layers of v-nodes.

The command that mounts a file system is called `mount`, and it also can provide information about currently mounted file systems. Let's look at a typical output for a lab machine:

```
/dev/sda1 on / type ext4 (rw,errors=remount-ro)
proc on /proc type proc (rw,noexec,nosuid,nodev)
sysfs on /sys type sysfs (rw,noexec,nosuid,nodev)
none on /sys/fs/fuse/connections type fusectl (rw)
none on /sys/kernel/debug type debugfs (rw)
none on /sys/kernel/security type securityfs (rw)
udev on /dev type devtmpfs (rw,mode=0755)
devpts on /dev/pts type devpts (rw,noexec,nosuid,gid=5,mode=0620)
tmpfs on /run type tmpfs (rw,noexec,nosuid,size=10%,mode=0755)
none on /run/lock type tmpfs (rw,noexec,nosuid,nodev,size=5242880)
none on /run/shm type tmpfs (rw,nosuid,nodev)
binfmt_misc on /proc/sys/fs/binfmt_misc type binfmt_misc (rw,noexec,nosuid,nodev)
rpc_pipefs on /run/rpc_pipefs type rpc_pipefs (rw)
gvfs-fuse-daemon on /var/lib/lightdm/.gvfs type fuse.gvfs-fuse-daemon (rw,nosuid,nodev,user=li
nuzee.academy.usna.edu:/home/mids on /home/mids type nfs (rw,nfsvers=3,bg,addr=10.1.83.28)
zee.cs.usna.edu:/courses on /courses type nfs (rw,bg,addr=10.1.83.18)
zee.cs.usna.edu:/home/scs on /home/scs type nfs (rw,nfsvers=3,bg,addr=10.1.83.18)
```

There is a lot going on here, let's focus on a few key lines:

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
/dev/sda1 on / type ext4 (rw,errors=remount-ro)
```

This line says that the device `/dev/sda1` is mounted on `/` the root of the file system. This is the root file system. Devices that start with `sd`, which stands for scuzzy drive, refer to discs. The type of the file system is ext4, the standard linux file system. The rest of the lines are mount options.

Many of the intermediary lines are not important to this discussion:

```
proc on /proc type proc (rw,noexec,nosuid,nodev)
sysfs on /sys type sysfs (rw,noexec,nosuid,nodev)
none on /sys/fs/fuse/connections type fusectl (rw)
none on /sys/kernel/debug type debugfs (rw)
none on /sys/kernel/security type securityfs (rw)
udev on /dev type devtmpfs (rw,mode=0755)
devpts on /dev/pts type devpts (rw,noexec,nosuid,gid=5,mode=0620)
tmpfs on /run type tmpfs (rw,noexec,nosuid,size=10%,mode=0755)
none on /run/lock type tmpfs (rw,noexec,nosuid,nodev,size=5242880)
none on /run/shm type tmpfs (rw,nosuid,nodev)
binfmt_misc on /proc/sys/fs/binfmt_misc type binfmt_misc (rw,noexec,nosuid,nodev)
rpc_pipefs on /run/rpc_pipefs type rpc_pipefs (rw)
gvfs-fuse-daemon on /var/lib/lightdm/.gvfs type fuse.gvfs-fuse-daemon (rw,nosuid,nodev,user=li
```

You'll notice that the `proc` file system is mounted on `/proc` and we've used that to access information about running processes. But the real action is at the bottom where the home directories are mounted:

```
nuzee.academy.usna.edu:/home/mids on /home/mids type nfs (rw,nfsvers=3,bg,addr=10.1.83.28)
zee.cs.usna.edu:/courses on /courses type nfs (rw,bg,addr=10.1.83.18)
zee.cs.usna.edu:/home/scs on /home/scs type nfs (rw,nfsvers=3,bg,addr=10.1.83.18)
```

The `/homes/mids` and `/courses` and `/home/scs` directories are actually network mounted file systems (NFS). They are not stored locally on a lab machine but rather remotely on a machine in Ward Hall. This is why when you log into any computer in the lab, you get the same home directory and files.

We can now view the larger file system on a lab computer

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

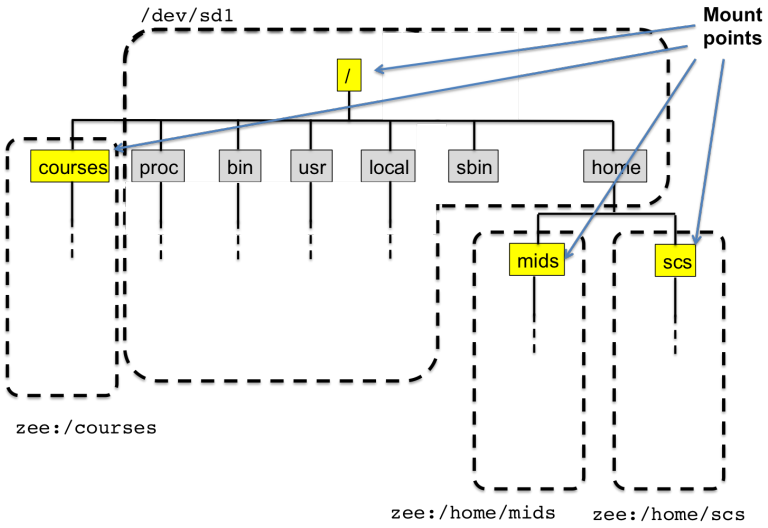


Figure 10: Mount Points

Each of these different mounted file systems are combined under the larger filesystem, and each file system has a different mechanisms for reading and writing. The file system on /dev/sd1 writes to and from a disc, while NFS uses the network to do read and writes. All of this is combined and unified and abstracted away from the user, which is where the real power comes from.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD