CMPSCI 377 Operating Systems **Fall 2013**

Lecture 16: October 28

Lecturer: Prashant Shenoy *Scribe: Armand Halbert*

16.1 Kernel Memory Allocators

We have looked at memory allocation for processes but the kernel also needs to allocate memory for itself. The kernel cannot simply use malloc like a user process so it needs its own kmalloc. There are multiple reasons why the kernel cannot have a fixed amount of memory but one example is the list of PCBs will need to grow and shrink as processes come and go. One type of kernel memory allocator is the *Buddy Allocator* which can only allocate in sizes that are powers of 2. This type of allocator can lead to internal fragmentation by allocating more than will be used. For example, if the kernel needs 58 bytes of memory it will have to allocate 64 bytes anyway. Another type of allocator is the *Slab Allocator* that is used by Solaris and Linux. This allocator will group objects of the same size into a "slab". Each slab is just a region of memory dedicated to object of the same size. If more than a slab of objects is needed than another is allocated and the object cache will point to both (or more) slabs.

16.2 File System & Data Requirements

A File System is an abstraction layer on top of a disk provided by the OS. The file system translates the raw disk blocks to higher level objects like files and directories, and also provides additional functionality such as security. A file system may be run on top of a standard hard disk inside a desktop or laptop, or it can be on top of any other storage device such as a DVD or a removable flash drive. In all cases, there are certain requirements that users expect to be held for their data:

- **Persistence:** that the data written to the disk will still be available if the machine is powered down or crashes
- **Speed:** that data can be accessed and manipulated reasonably quickly
- **Size:** users may expect to be able to store large amounts of data (e.g. typically much more than can be held in RAM)
- **Sharing & Protection:** users may want to share some data and keep other data private
- **Ease of Use:** users want to be able to easily find, examine, and modify their data without needing to know about the exact hardware or physical layout of where data is stored

Some of these features are provided by the hardware itself, while others are abstractions provided by the operating system. Hard disks provide non-volatile memory—e.g. permanent storage that is not lost when the system is turned off. In some cases, the OS provides an additional level of persistence by writing data to disk with some redundancy, allowing files to be recovered if they are partially corrupted. The hardware is also responsible for speed and size requirements. The OS also provides protection and sharing mechanisms by keeping meta data about files that indicate the read, write, and execute privileges for different users.

The primary feature provided by the OS is related to ease of use. File systems give users the abstraction of organized files and directories. This is much easier to work with than if users needed to know about individual disk sectors where the raw data is stored. The OS also often provides higher level functionality such as search facilities.

16.3 Files

A **file** is the most basic logical unit within a storage device. Formally, a file is a named collection of data that is written to a storage device. A file may simply be filled with text data, or it could be the binary for an executable program. The actual contents of the file may be stored in different ways depending on the operating system and file system in use. Unix based systems store a file as a series of bytes, known as an unstructured system. In contrast, IBM mainframe systems store files as a collection of records or objects—a structured approach that can provide greater reliability and optimization for databases. Along with the data contents of a file, various file attributes are maintained such as its name, type, location, size, permissions, and creation time.

In order to keep track of which files are in use, the OS maintains a set of tables with information about which files are in use by which processes. The **Open File Table** is shared by all processes on the system and contains a list of files that are currently opened. Since a single file may be used by multiple applications at once, the Open File Table keeps track of how many applications are using the file (open count), meta data about its permissions and ownership, as well as pointers to any memory regions being used to buffer the file. This can help the OS optimize accesses from multiple processes to the same file. There is also a file table maintained for each running process, called the **Per-process File Table**, that contains additional information about the specific files used by each application. This table maintains pointers back to the OS-wide Open File Table, and also tracks information such as the current offset in the file for reading or writing, as well as whether the file was opened in read or write mode.

16.3.1 File System Interface

The file system presents an interface to users so that they can read and modify their data. Examples of common file operations include: Create(), Delete(), Open(), Close(), Read(), Write(), and Seek(). In addition, the file system exposes functionality for adjusting the attributes as files with functions like SetAttribute() and GetAttribute().

A call to **Create(name)** is used to define new files with a given name. To create a file, the file system will first reserve space on the physical disk (first verifying that there is sufficient space available and that the user has the proper permissions to create the file). Then, a file descriptor is created for the file containing meta data such as the name, location on disk, and its various attributes. This file descriptor is then added to the directory that will contain the file.

Depending on the operating system, files may or may not contain a type (e.g. MS Word files, JPG image file, or executable). The benefits of tracking file types is that it can lead to better error detection since the OS can determine if a file is of the correct type, ease of use for users since the OS can automatically perform the proper action when a file is accessed, and the storage system can potentially be optimized to organize files by type, leading to improved performance. On the other hand, tracking file types makes the OS more complicated since the OS needs a way to learn about new file types.

A call to **Delete(name)** will attempt to remove a file from the storage system. This will cause the file system to determine all of the disk blocks used by the file and free them to be used by other files. The file descriptor maintained in the directory must also be removed. In general, file systems will only allow a file

to be deleted if no other processes are accessing it, e.g. its open counter is zero.

The **Open(name,mode)** call is used to open a file for either reading or writing. When the open call is made, the OS must find the file within the file system and make a copy of its file descriptor in the system-wide Open File Table. The OS will verify that the user process has the proper permissions to access the file, and if so, it will increase the open count and return an ID to the process. The process's file table will be updated to include an entry for the new file, and the offset will be set to the start of the file.

Likewise, the **Close(fileId)** call will close the file referred to by fileID. This will remove the file's entry from the process's file table and reduce the open count in the OS's Open File Table. If the open count reaches 0, then the entry is removed from the system-wide table since no other processes are using it.

The **Read(fileID, from, size, bufAddress)** and **Read(fileID, size, bufAddress)** will both read data from the given file. The first function is used to read from an arbitrary position in the file—referred to as a "random access" read. The latter function is used for "sequential access" reads—the next *size* bytes are read from the current offset within the file and copied to the memory region referred to by *bufAddress*. Similar functions are used for writing to the file, or for seeking within a file. A seek call updates the current position in the file, and would typically be followed by a sequential read or write. Most operating systems also support the concept of **memory mapping** a file. This is used to map a portion of the process's virtual address space to a file. This means that reading or writing to that "memory" region will actually cause read or write methods to be called on the corresponding file. This can simplify the programming model for working with files since they can simply be treated like arrays in memory.

16.3.2 File Access Methods

¿From the programmer's perspective, accesses to files are typically either sequential or keyed. Sequential accesses are used by most programs, and simply keep a pointer into the file and either read or write to that location. After the call completes, the pointer advances based on the size of the read or write. In a key based access system, the address of the data being read or written is determined by the value of some key. Examples of key based accesses are searching within a database or using a hash table or dictionary. These access types must be translated by the OS to either sequential or random access patterns. Sequential accesses are the simplest perform, and can allow for various optimizations since the OS has a good idea what data will be read or written to next. Providing random accesses typically has lower performance since requests may come in any order, potentially leading to poor access locality within the disk. Note that random access can be implemented using seek and sequential access.

16.3.3 Naming within the File System

The OS needs a way of giving a unique name to every file in the system. While users typically refer to files by textual names, the OS general assigns numeric IDs to each file instead. The actual name of each file can then be determined by looking at the meta data stored in each directory. In early computer systems, the file system presented a **single-level directory** to the user. This meant that all files on the system resided within a single directory. This was very limiting since no two files could share the same name, and there was very little structure to how files were laid out. Later systems moved to **two-level directory** systems where each user was given a directory for her files, but all the files within a user's directory sill needed to be uniquely named. Modern operating systems support **Multi-level directories**, which provide a tree structured namespace. In this type of system, a directory is actually just a special type of file that can only be modified via system calls. The directory "file" simply stores a list of ¡name,fileDesc¿ pairs that refer to each of the files contained within the directory.

Sometimes it is useful to refer to a single file from multiple locations. Operating systems such as Unix support this through the use of **hard links**. A hard link creates a second connection to a file, allowing it to be reached from two locations. When a hardlink is created, the file system increases the reference count for the file—a file can only be deleted if both its open count and its reference count can be brought down to zero. Operating systems also support the idea of **soft links**. A soft link is another way to create a reference to a file, but the reference acts more like a short cut—it is not fully tied to the original file. Thus if the file referred to by a soft link is deleted, the soft link will simply become a broken link that does not point anywhere.

16.3.4 Protection

We have seen before that a key job of the Operating System is to provide protection. Protecting files within a storage system is also the OS's job. In systems such as Windows NT and modern Linux, **access lists and groups** are used to provide protection. An access list, or **ACL**, contains the names of all users which have access to a file, as well as what type of access they should have. However, such lists can easily become large and difficult to maintain. In Unix systems, **access control bits** are used instead. This provides slightly coarser grain permissions—access can be limited but not on a per-user basis. Instead, permissions are broken up into three different categories of users: the file's owner, other users in the file's group, and all other users. For each category, permissions can be set about whether that group can read, write, or execute the file. This system is very efficient since only a few bits of extra data need to be stored for each file.