Dec 4, 2006
Lecture 15: Unix/Linux OS

November 12, 2007

1 ADMIN

ANNOUNCEMENT: the final exam is on Monday, December 18, from 2-3:30 in our usual classroom.

In these last 2 weeks, we will examine two OS: Linux (Chapter 20) and Windows XP (Chapter 21). Note that before we go into Linux, we need to understand Unix, and this is Appendix A of the book. This Appendix is online from publishers. Unfortunately, their pdf file is readable but does not easily printing. The lecture notes directory provides a printable postscript version of this pdf file for your convenience.

2 REVIEW

- Q: Describe the ways of using the reference bit and modify bit in paging hardware.

  A: (1) reference bit is used in second-chance algorithm

  (2) modify bit is used to save on writing out pages that are not modified

  (3) both bits together divide the pages into 4 classes, which can be used for page replacement rules.

- Q: Use the struct datastructure to write the specs for a page table entry, and the corresponding specs for a process table entry. Describe the algorithms for using these two tables.

- Q: What is "memory-mapped files"? What are some consequences for file manipulation?

  A: Think of this as demand paging for files.

  This can give faster access and manipulation of files, but writes to files are no longer immediate/synchronous events.

3 UNIX History

In 1969, Ken Thompson of Bell Labs started developing Unix on an idle PDP-7 computer. He was soon joined by Dennis Ritchie. The name UNIX is a pun on the then state-of-art MULTICS system. In 1983, Thompson and Ritchie wond the ACM Turing Award for their work on Unix. This award is the computer scientists' equivalent of Nobel Prize.

Unix was always written as a system for programmers, developers and researchers. As such, its environment is rich with developer's tools. In fact, it lead to the term "Unix Tool" as a philosophy for developing software. From the start, it was extremely popular in universities and research labs. The C Language was developed to support the development of Unix (the original Unix was written in Assembler language).

In 1976, Version 6 of Unix was released to Universities. In 1978, Bill Joy began to work on a Unix for the VAX machine at UC Berkeley. This project introduced virtual memory and demand paging. But the biggest impact was their funding from DARPA to develop various networking protocols (in particular, TCP/IP). These protocols are responsible for the rapid rise of the Internet (from 60 nodes in 1984 to over 8,000 in 1993). The last Berkeley release was 4.4BSD in 1993. There are MANY strains of UNIX, in part, thanks to its development environment and the fact that it has to be adapted to various hardware. The original UNIX line from Bell Labs called UNIX System V, and the Sun's version called Solaris 2, etc, are all commercial products.

Appendix A describes a particular strain, called FreeBSD, that is completely free, with no strings attached. This project started in 1993, and is derived from 4.4BSD-Lite (a variant of 4.4BSD). It is developed for the Intel platform. Chapter 20 describes yet another Unix development, the LINUX project.

4 Design Principles

- Unix is designed as an interactive TIME SHARE system (recall the MULTICS connection)

  –User interacts with the computer using the SHELL interface

- Its file systems is hierarchical (beginning at the root "
  ").

  – So each file is a leaf in this hierarchy.

  – Another interesting feature is that I/O Devices are manipulated like files! E.g., to write to the terminal, we simply write to a special file representing the terminal!

- Multiprocessing is another design principle

  – CPU Scheduling is based on priority

  – Demand paging is used

  – Process swapping is only used when there is excess paging

- The tools concept is critical:

  – Two important tools for developing software are the **make** Program and the **SCCS** (alternatively **CVS**) tool.

  – Our class is familiar with the make program – it is a way to automate various tasks in developing software.

  – Basically SCCS/CVS are tools to keep track of file versions, and to allow concurrent access to files by many users. This is critical in developing software.

- From the beginning, UNIX is notable for being small.

  There are modern pressures to add functionalities that make it grow. Two such functionalities are

  – networking

  – windowing support

- UNIX LAYERS:

4. User	application software		
3. System	shells, compilers/interpreters, system libraries, tools		
2. Kernel	signals I/O drivers	file system disk and tape	CPU scheduling virtual memory
1. Hardware	terminal controllers	device controllers	memory

  The Kernel is the most important and characteristic of Unix. The system programs access the kernel using System Calls (aka System Interface).

5 Unix File Systems

We will go into detail of one main aspect of Unix, its file system. Some of this information may be found in Chapter 11 (general file systems) and Chapter 12 (implementation of file systems) of Sibers.

- There are at least two layers in the description of files: one is the logical or user view, the other is the physical view.
- The physical memory is broken up into **blocks**. Typically, **block size** is 4K. Note that hardware disk sectors are 512 Bytes, and so block sizes are small multiples of this.

  Sometimes, there is an additional **fragment size**, which is between 512 Bytes and block size. A file would use the regular block sizes for all but the last one.

  Q: What are the trade-off issues in choosing block sizes?

  A: Larger block sizes allows faster access for larger files; but for smaller files, this causes more internal fragmentatin and is slower.
- A hardware disk can be partitioned into different file systems. Each file system is obtained by dividing the partition into blocks.
  1. Block 0 is the **boot block** (or boot control or boot partition block). If we want to boot an OS from this partition, this block is used. Otherwise it can be empty.
  2. Block 1 called the **superblock**. Contains number of blocks, size of blocks, free-block count, free-block points, i-node count, i-node pointers, etc. Note: i-node is also called File Control Block (FCB).
  3. Block 2 to some Max are the blocks for i-nodes (see below).
  4. The rest are data-blocks.
- Logically, a file is a sequence of blocks. There are two main kinds of files: **regular files** and **directory files**. There is a third, **non-disk files**. See below.

  REMARK: In Windows, directories are not treated as a special kind of file.
- There is an **i-node** (short for **index node** for each file. It contains the following information about a (regular) file:
  1. user ID and group ID of owner of file,
  2. time of last modification and access,
  3. number of hard links to file,
  4. mode (read/write/execute permissions),
  5. type of file (regular, directory, symbolic link, char/block/socket dev),
  6. 12 pointers to data blocks on disk. Thus, access to the first 48 ($= 12 \times 4$ K) of data is very fast.
  7. 3 additional indirect block pointers: single indirect, double indirect, and triple indirect.
     Since each block holds 4K bytes, and each pointer is 4 bytes, the first single indirect can access 4MB of data. The second can access 4GB. The triple indirect is not used, since current systems address space of 32-bit cannot go beyond 4GB.

  Normal users refer to files by their names and path. The System uses i-nodes to refer to files.

  Q: As user, how do you get the i-node number of a file?

  A: Type "ls -i file-name"
- A directory file is just a sequence of variable length triples: (length, inode-ptr, file-name).

  Each triple corresponds to a file in the directory. The inode-ptr refers to the i-node of the file, and file-name can be up to 255 chars long. The length refers to this file-name length.

  The first two entries in the file refers to "." and "..".

- Given a path, there is the obvious sequential algorithm to search directories (starting from / or from current directory).

  To avoid infinite loops, we cound the number of symbolic liks encountered and stop when a limit (8) is reached. REMARK: can't a hard link cause infinite loop too? If so, hard links should be counted as well.

- LINKS:

  Hard link are directory entries, like ordinary entries. So each hard link has a corresponding i-node.

  Symbolic links: such links do not creat a new i-node, only an entry in some directory.

  Hard links has no effect on the search algorithm, but symbolic links affect the search algorithm as follows. When our search reaches a symbolic link, we start the search all over, starting from the head of a path name that is associated with the symbolic link!

  Q: How to you verify that on your unix that hard links creates a new i-node but symbolic links does not?

  A: Create a symbolic and hard link to some file "foo" and look at their inodes:
  @ ln -s foo symfoo
  @ ln foo hardfoo
  @ ls -i foo symfoo hardfoo

- HOW TO HANDLE NON-DISK FILES.

  There are 3 main types: block device, char device, or sockets. We call appropriate drivers to handle them.

- Opening and Closing files.

  After we found the i-node of the file, we allocate a **open-file structure** for this i-node. An index into the **open-file (structure) table** is what we call a **file descriptor**. This open-file table is PER PROCESS.

- A directory name cache can be used to hold recent directory-to-inode translations, to speed up file access.

  There is a in-core list of i-nodes. All currently open files have a copy of their i-node here.

- It turns out, we want an intermediate structure between open-file structure and the i-node. The *current position* of an open file is one information that does not belong to either i-node or the per-process open-file structure. Hence we store this information in an entry of the **system-wide open-file (structure) table**. This entry, in turn, points to the i-node. It can also keep track of the number of processes that has opened this file. (See Fig A.7 of Appendix A)

  Let us illustrate a situation where we need this intermediate structure. Suppose a process P0 opens a file F1, and then fork two processes: P1 to write some head info into F1, followed by P2 to write additional info into F2. Note that P0's open-file structure for F1 will be inherited by P1 and P2 through the fork.

  Now, after P1 has done its work, the updated location should be accessible to P2. But this information, if stored in P1's open-file table, would not be accessible to P2. But if both of them points to the "system-wide open-file structure", then P2 can continue where P1 left off!

- **Clustering Schemes.** It is best to allocate an i-node and its data blocks from the smake locality in a disk. FreeBSD has a concept called **cylinder group** (cylinder refers to a locality on the disk). Each cylinder block has a superblock, array of inodes and data blocks, just as in a partition. All cylinder blocks have the same superblock. Block allocation can use such information to allocate free blocks to preserve locality (and other considerations). See p.908 of Appendix A.

- **Free Space Management.** There are three basic methods:

  (1) The simplest method is to keep a linked list of all the free blocks.

  (2) We could also use bitvectors to track the free blocks.

  (3) Finally, we could let each free block keep the addresses of the free contiguous blocks (i.e., an address for the beginning of the block and a count of how many following blocks are free).

  Let us discuss scheme (3): Which of these free blocks should recursively be used to store similar information? If ALL the free blocks are used in this way, then we cannot easily give away the current free block to requests for blocks. An intermediate solution is to just use the last two free blocks for recursively storing such free-block information.

  Advantage: we have a branching factor of two, but all but two of the free blocks in the current direct can be directly given away.

  If a free block is released (by file delete), this free block can often be inserted directly at the root. This is most efficient.

- **File Consistency Problem.** Suppose the computer crashes. The open-file table is generally lost. We need to check the consistency of the file system. The loss of an i-node, a directory block, or a free-list block is most serious. To deal with this, most OS has a utility program to check consistency of a file system. In Unix, it is called "fsck" and Windows it is called "scandisk". We normally run this utility on system reboot, especially after a crash. In Unix, we need to check consistency of two things: block structure and file structure.

  Let us first consider block consistency: we will consider the problem under the assumption that free blocks are kept in a free-list. Basically, every block must exactly once, either in the free list or referenced by some i-node. Suppose b is the block number and $free[b]$ and $inode[b]$ indicates how many times b appears in either list. Then we have the following states:

  - $free[b] + inode[b] = 1$: b is consistent
  - $free[b] + inode[b] = 0$: b is missing
  - $free[b] > 1$: b is duplicated in free list
  - $inode[b] > 1$: b is duplicated in file system list

  So our algorithm begins by initializing $free[b] = inode[b] = 0$ for all b. Then we go through the free list, and increment $free[b]$ for each b that we find in the free list. We also go through the file hierararchy, and for each i-node, we look for all the blocks pointed to via this i-node. Presumably, there is enough main memory for storing the $free[b]$ and $inode[b]$ arrays.

  In case of inconsistency, we need to take some action: if a block is missing, we add it to the free list. If it is duplicated in the inode list, we duplicate the data in the block (by getting blocks from the free list), and modify the inodes that point to these blocks. If it is duplicated in the free list (either because $free[b] > 1$ or ($free[b] = 1$ and $inode[b] > 0$), we can fix the free list.

  Next consider checking consistency of the file system: This is basically a check that the number of hard links to an inode is correct. We simply recompute this number for each inode.

  Finally, it is possible to combine both block consistency and file consistency check into one single algorithm.