by Peter Collinson, **Hillside Systems**

Disk Partitions

Last month I looked at how the UNIX file system works, why there is the notion of the file system, and how it supports the processes that run on the kernel. As part of the explanation in this tale, I went into the nature of disks and how the operating system perceives them. We saw that disks hold a large number of 512 byte sectors, each with a separate address that the machine can access separately. This month, I am going to explore the way disks are used by the operating system itself.

Disks on UNIX are usually split into distinct chunks usually called partitions. The UNIX System V documentation refers to them as slices. You will find that Sun has partially picked up that terminology for compatibility purposes. One of the original reasons for implementing the notion of partitions on a UNIX disk was to permit the system to support multiprocessing. An area of a disk is needed to act as a storage area for the kernel. The storage area retains the memory images of running processes when there is insufficient memory to hold all the processes required to be run on the machine.

One of the jobs of the kernel is to manage processes, deciding which one to run and when. The kernel itself will also run from time to time. It's generally driven by hardware interrupts that tell it there's something to do, or started by system calls executed by the processes. When the kernel has nothing else to do, it starts a process running, which means the process is given the CPU, and it will run happily executing its own code.

Assuming that no hardware interrupt occurs, the process runs until it makes a system call. The process has perhaps decided to read or write some information, and the business of doing I/O means that the process can do nothing until the I/O operation is finished. The kernel then starts the I/O operation on behalf of the process and is told when it's done. In the meantime, some other process can be given the CPU to get on with some work.

Of course, we cannot rely on a process giving up the CPU to do some I/O, so the UNIX kernel has a scheduler that decides which is the most worthy process to get the CPU next. Some processes will be sleeping while waiting for an event, such as input to arrive or some output to be cleared. Some processes will be ready to run and the kernel picks the next one based on a priority value.

In early UNIX systems, the entire memory space of the process needed to be memory-resident to allow the process to run. As more and more processes were started, the system would run out of memory and action would be needed to start new processes. Processes that were not doing anything were moved out to disk in their entirety. We call this "swapping out" the process.

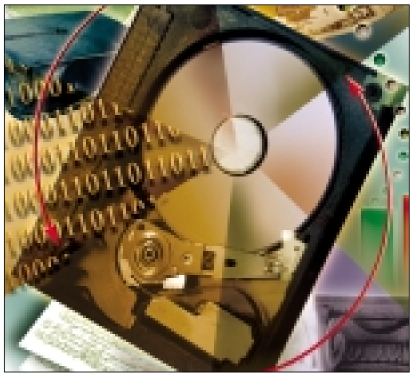
To make swapping an efficient operation, the kernel used the capabilities of the disk controller, telling the disk hardware to move some portion of the memory out to disk in a single operation. The data is moved automatically by the controller using Direct Memory Access (DMA) techniques. When the process has been swapped out to disk, it sticks there until it has something to do.

When the kernel saw that the process had some work to do, perhaps the input for which the process was waiting had become available, the kernel moved the memory image of the process back into memory again, and started it running. Of course, moving one process back in from disk ("swapping it in") may displace a less worthy process, in turn forcing it to be swapped out. A lot of disk bandwidth on busy systems was consumed by simply swapping processes in and out of memory. As things become busier and busier, process swapping could be all that can be happening on the machine, a situation known as "thrashing."

Incidentally, swapping was also part of the mechanism used to create a new process. The `fork()` system call diddled a little with some kernel data structures and swapped the process out, without losing the original memory image of the parent process. The child was swapped back in again as a new process.

In many ways, the early systems were moving too much data. With special hardware support, we don't actually need the entire code and data set of the process to be resident in memory. We can make progress when only some portion of the code and its data is directly available to the CPU. It's a little like waking up in the night and reading some of a book. At the moment you wake up, you only need to see the same page you were reading when you put the light out to go to sleep. You don't care about the part of the book you have read, or the part of the book you haven't read. At any instant in the execution of a process, we only need to have the portion of the program that is actually running in memory, plus any data areas the process is accessing at that instant.

To realize these ideas, computers were developed to implement "paging," usually with some form of hardware assistance. In a paging system, the memory image of the process is split into fixed-sized chunks or pages. To run the process, only those portions of the process actually in use need to be resident in memory. The term for this is the "working set" of the process. The hardware tells the kernel when a process tries to access some memory that's really on disk (called a "page fault") and the kernel arranges to bring the relevant page into memory. The process is put to sleep until the necessary information is resident. Incidentally, this mechanism is powerful because it means we can get



large processes running on the system. Processes can be larger than the physical memory of the machine because some of their address space can be on disk. Systems that use this technique have "virtual memory" because of this feature.

As we've seen, in both paging and swapping systems, the kernel needs to have some portion of the disk used as storage for processes not running. For UNIX, we tend to call this the "swap area," irrespective of whether the system is swapping or paging, due to the historical legacy of UNIX as a swapping system.

The swap area on the disk usually occupies a partition that isn't used by a normal file system. This was done originally because the kernel wanted swapping to be efficient. To swap something out, a contiguous section of disk is allocated to take a complete copy of the data that is to be stored. DMA allows us to make the copy from memory with a single command to the disk controller.

We would not get this efficiency if we planned to use the regular file system to store swapping information. As we saw last month, the file system is managed as a data structure using pointers to blocks. This data structure stores information in disk blocks (or groups of disk blocks) that may be randomly scattered over the disk. So the regular file system is not a good place to store swapping information because it's hard to get contiguous chunks of disk of the right size.

The original solution was to dedicate a chunk of disk to store swapping information. The kernel managed this space using a simple mechanism. When paging came along, this space was used to store pages from memory that had been altered. There is no absolute need in a paging system to have a separate partition, because the kernel is usually moving a fixed sized page between the disk and memory. However, it's still more efficient for the kernel to run its own paging file system. It doesn't have to worry about mapping the page size of the memory onto the block size used by the disk file system. The kernel also attempts to balance the load caused by paging across several disk spindles if the system has more than one disk containing a paging partition.

Partitions

We have seen that UNIX splits a disk into at least two partitions: one for the swap area and one for user files. It was normal in the early systems to split a single disk system into three: the paging space, one partition that contained the root of the file system, and one partition that started at `/usr` containing files that were frequently altered. This split mapped nicely onto the 2.5 MB RK05 disks I used for the UNIX Release 6 system I originally ran. The first RK05 held the root file system and the swap area, the second the user files.

If you used a single disk for the system, you still tended to split the disk into three parts: swap, root file system and `/usr`. The reasons for the split were various. A big one was self-defense–the root file system is needed to boot UNIX and if it became corrupt then you could be sunk. In the early systems, just closing the system down when things were running was fraught with danger. One of the earliest things I did on UNIX was to write code that killed all the running processes before halting the system, and I found that my need to repair the file system on disks reduced considerably.

By keeping the root file system small

(and full of files that rarely changed) you minimized the possibility of a bad close-down with corrupted files, or worse, corrupted directories appearing when the system was rebooted. When I could, I always moved the `/tmp` directory into a partition away from the root on my systems. All the other files on the root partition were updated rarely, so just moving `/tmp` onto another spindle enhanced robustness. Also, if the root partition was small, you could replicate it onto another disk spindle, leaving you with a fallback disk to use as a bootstrap in case of need.

Many of these reasons have disappeared with time. Files on `/tmp` have disappeared into `tmpfs`, a file system maintained in swap space. Also, some of the work done by Kirk McKusick with the fast file system for the BSD UNIX releases concentrated on making the file system more robust in the face of sudden failures. When that work was combined with the `fsck` program that automatically repaired a broken file system, UNIX gained a robust file system that rarely had catastrophic errors. The need for a duplicate root file system has disappeared because you can now boot the system from CD, creating a running UNIX system that can be used as a vehicle to repair a broken system.

Even though the software improved and file systems became more robust, systems were still configured with a small root partition, a swap partition and user files on extra partitions. The reason for staying with this configuration was the implementation of the Network File System (NFS). NFS made a reality of the diskless workstation. Each workstation needed its own configuration information stored (mostly in `/etc`) on the root partition. Attempts were made to minimize the replication of information, especially compiled programs, making the per workstation information as small as possible. Even if your workstation had a disk, it was usually small. The files that needed tailoring were held locally, while the system reached out over the network for the large files held in the `/usr` partition, so a small root was a win for that situation, too.

The requirement of supporting diskless workstations as a standard seemed to disappear with Solaris 8. Disks are now large and relatively cheap, so perhaps the demand for diskless workstations has evaporated. By default, Solaris 8 will install into two partitions on the disk: one for paging and one for user files. I split my two disks on my workstation into two identical patterns: around 5 GB of swap space, a 1.67 GB root partition (that's used for `/var` on the second disk), and a 6.31 GB remainder.

Mounting File Systems

Given a scenario where you can have several disks, each with several partitions that contain file systems, how do we allow users to access all the files? UNIX tackles this problem by allowing you to "mount" a file system onto an existing directory. The mount operation creates a single tree of files, where the jump from one partition to another is seamless. As you traverse the tree downwards, the system notices when you cross a mount point, and the name lookup will automatically jump to the root inode of the new partition. The mechanism also permits the "go up to the parent directory" operation to work as expected. As you go up by typing:

```
$ cd ..
```

the system spots that you've reached the top of one partition and will use the mount information to move into the partition where the mount point lives.

The fact that the mounting operation is seamless is a huge win. It allows you to organize your system quite flexibly, mov-

Postscript to my *ksh* article

Last year, I wrote an article on ksh ("The Korn Shell," *S/W Expert*, November 2000, *http://swexpert.com/C2/SE.C2.NOV.00.pdf*) and mentioned that I didn't know how to bind the keyboard arrow keys to permit inline editing. I was mailed the following information by several people–too many to enumerate here. Thanks to you all.

This text was pulled from mail from Andy Feldt from the Department of Physics and Astronomy, The University of Oklahoma.

```
#---------------------------------#
# The ksh has an undocumented way of binding the arrow keys to the  #
# line editing commands: (The ^A means the character Ctrl-a)       #
#---------------------------------#
# here is the mapping if you use Emacs-style editing...
alias -x __A='^P'       # <Up>    Ctrl-P: Line up (Previous)
alias -x __B='^N'       # <Down>  Ctrl-N: Line down (Next)
alias -x __C='^F'       # <Right> Ctrl-F: Character right (Forward)
alias -x __D='^B'       # <Left>  Ctrl-B: Character left (Backward)
alias -x __H='^H'       # <Home>  Ctrl-A: Go to line beginning
alias -x __P='^D'       # Ctrl-D: Delete character forward
alias -x __q='^E'       # <End>   Ctrl-E: Go to line end
```

Andy also tells me that this insert to the ksh start-up file works and has been tested on Solaris 2.5 through 2.7, AIX 4.3.0 through 4.3.3, HP-UX 10.10 and 10.20, DEC OSF 4.0, and in pdksh version 5.2.12.

ing files around as you need to. It can sometimes pose a problem, because it ties the geography of the file system to the partition layout. Some of these problems have been ameliorated by the invention of the symbolic link; it's now possible to create file layouts that include files from several partitions.

Mounting has proved to be a problem for removable devices such as CDs or floppies. The enviroment on PCs allows users to stick a CD into the drive, use it and press "eject" with no messing around. UNIX wants you to mount it, and also tends to want you to have superuser privilege to perform the mount operation. Once mounted, UNIX will retain information about the file system on the device, so to get it out of the drive, UNIX wants you to unmount it, and then press the eject button.

If possible, UNIX tends to disable the eject button on the device to prevent unintentional removal of the CD from the drive before UNIX has purged all the information about the file system on the CD. Actually, Sun has worked hard with Solaris to make the mounting and unmounting of CDs and floppies into an automatic user-friendly operation so the business of mounting or unmounting isn't needed.

Finding Out About Partitions

The mount command, (or more precisely, the mount system call) needs a way of accessing the partition to be used for the mounting operation, and in the UNIX style, each partition is accessible from files in `/dev`. The files here are mostly "special" files, accessing devices. Each partition has two access points: the "raw" device that allows a process to have access to the entire partition; and the "block" device that's used for mounting the device.

Programs that want to have access to the disk to tell you something about it will generally open the raw device. The raw device interface code moves data from the disk into the address space of the process. The raw device also allows us to give a user process complete access to a disk partition, perhaps to support its own file system in that partition. Some database applications use this feature to run their own special non-UNIX file systems.

In `/dev` on your Solaris system, you'll find raw devices in `/dev/rsdk` and block devices in `/dev/dsk`. The files in these directories are called things like `c0t1d0s7`. The name is coded to tell you where the disk is connected into the system. The coding here being *controller* 0, *target* 1, *disk* 0, *slice* 7. On Linux or systems derived from BSD, the devices tend to have names that are a bit more accessible, like `/dev/sd0h` for the block device and `/dev/rsd0h` for the raw version of the same disk. In this coding, the last letter conventionally gives the partition number, so "h" will be partition no. 7.

Systems tend to support a partition set that overlaps, so care is needed to use the correct partition name when using an application that creates file systems. It's usual to have one partition that maps onto the entire disk. Traditionally this was the "c" partition on BSD systems, and Solaris uses slice no. 2 for this. On Solaris, the format program allows you to examine and change partitions, assuming that you have superuser rights. Be careful, you can kill your system by repartitioning it. Although on UNIX we don't care too much which partition files reside, it can begin to matter when disks become full. The `df` command is used to find out the status of the various partitions on the disk. I personally find the output from standard System V `df` command to be less than useful, and use the version that lives in `/usr/ucb`. I create a symbolic link from my `/bin/` directory to access it easily.

Beware

I have simplified some portions of the information that appears in this article, largely because there are some nuances of how swapping and paging works that I didn't want to get into. Sometimes leaving things out makes other portions of the story easier to tell.

***Peter Collinson** runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on an UltraSPARC/10. Email:* `pc@cpg.com`.