Linux Os Overview Essay, Research Paper

The Linux Operating System The Linux Operating SystemThe following report contains a brief overview of the Linux operating system – in particular, the system managers. First, the Linux interface is described, followed by the system managers:- memory manager- process manager- processor manager- file manager- device managerThis report is targeted at readers who have a general knowledge of computing and operating system fundamentals, and want to know a little more about Linux as compared to other operating systems in general. This report is a part of a series which outlines some of the more popular commercially available operating systems. The Linux InterfaceThe Linux operating system has a user interface very much like that of Unix. Linux has command line interface on the most basic level, but it also incorporates the use of a graphical user interface. The graphical user interface used by Linux is X-windows. X-windows, unlike traditional PC windowing systems, is completely decoupled from the underlying operating system, so much so that, in theory, another windowing system could be installed instead of X-windows without making any difference to the underlying operating system. This, however, is completely unnecessary as X-windows places no constraints upon any graphical programs with respect to window decorations or mouse focus policies. This merely provides the framework. X-windowsThe layout of the desktop, the look and feel, is a function of a special program known as the window manager. An X-windows window manager provides window decoration, window state and placement, icon management, mouse pointer focus policy, popup menus and user specified key and mouse button bindings. There are many different window managers available for Linux / X-windows. Some of these emulate the Windows ‘95 look and feel, others emulate the NextStep look and feel and there are completely different ones again. As one Linux user put it “the user, not the software company, controls the look and feel of the desktop”. There are several highly popular Linux shells commercially available to the public: KDE, E-Desktop by Corel, Debian, and Gnome. Linux is “open source,” meaning that the actual code is available publicly. This allows a user with sufficient software development knowledge to further tailor the system to his/her preferences.The big advantage of X-windows over the traditional PC windowing system is that it is network transparent. What that means is that XClients (X-windows programs) can display on any XServer in the network. The XServer provides the framework under which the XClients run. One practical application of this is that if some CPU intensive program needs to be run, then it can be run on the most appropriate host, but the program’s display can be on the local host. Some applications might only be able to run on particular nodes, those with special hardware devices attached or say a database installed. This poses no problem for Xclients; the XClient simply displays on whichever host it is told to. This allows for true multitasking across a network by allowing for terminals to simply run a shell that controls, via the network, another Linux machine transparently, without the user of the terminal noticing the difference. Of course, this totally depends on network bandwidth.The X-windows system is not limited to Linux or to Unix at all. There are other shells currently commercially available for Windows.Another notable attribute of the Linux operating system is its ability to take the basic functionality of the operating system, package it up using a shell script such as PERL (Programmers Extraction Report Language) and essentially add new functionality through scripting. This scripting is similar, but significantly more powerful, than the batch processing of MS-DOS. Memory ManagerThis section will talk about how the Linux operating system manages memory, both physical and virtual, and how programs are loaded into memory.Physical MemoryThe Linux kernel uses a page allocator called a “buddy heap” to manage physical memory for most cases. The spaces (groups of pages) in memory are organized in pairs as follows: each small space of free memory in the system is kept track of, and each space has a kind of a “sister” space of free memory. If both become free at the same time, they combine to create a larger free space, which, in turn can combine with it’s larger sister space and form an even larger space. By the same token, if a small job comes in to the system and doesn’t fit well into any of the larger spaces, those larger spaces can be divided into two sibling spaces and the job will fit into one of them. If the space is still too large, the pieces will keep dividing until an appropriate fit is found, or until the smallest size until is reached (one page). In order to allocate memory area which is smaller than a page, the “kmalloc” service is used.The kmalloc service is used when a component of the operating system needs smaller blocks of memory, whose size may not be known in advance. The kmalloc service allocates pages which can then be split into smaller areas. These regions allocated by the kmalloc service cannot be dis-allocated at a later time by the kmalloc system, which means that even in the case of memory shortage, the kmalloc will not let go of memory that it has claimed.The Linux memory manager uses two types of cache: the buffer cache and the page cache. The main cache is the buffer cache, through which the input/output is performed to disk drives and other such devices. Virtual MemoryThe Linux virtual-memory system keeps track of every address location that some process can see. It stores the view of these addresses logically as follows: the instructions the virtual memory manager receives concerning the state of the address state is mapped. The address space looks like a set of regions which correspond to a virtual address holding contiguous pages of free memory. Each of these regions is associated with a set of properties (read and write privileges, associated files), and each region is sorted using a binary search tree for fast access. The kernel also maintains a second, more physical view of the address spaces, which is stored in the process page table. This view is used to determine the exact location in memory (or on disk) for each virtual memory page. ProgramsLinux programs are loaded into memory using the “exec” system call. When this happens, the pages of the program are mapped into virtual memory. When the program goes to read a page, the page will load into physical memory. Once the program is running, the memory manager needs to worry about linking with system libraries. This can be done all at once upon loading of the program by including the library functions in the executable file (static linking), or it can be done dynamically. The advantage of dynamic linking is obviously that it is wasteful to generate copies of the same functions when we could get away with only loading one copy of the function (as in the case of dynamic linking). Linux implements Dynamic linking by reading the list of libraries that the program will need when the program starts (this information is embedded in the program). Once the list is read, all of the library functions are mapped into virtual memory in the same way as the program was, and as the pages are needed, they are brought into physical memory. Process & Processor ManagementLinux is a multiprogramming and multiprocessing operating system. It organizes many processes vying for the CPU’s attention. Each competing process is kept as a separate entity, with its own rights and attributes. This allows each process to run independently, unaffected by another process crashing. Each process has its own virtual address space and only interacts with other processes through kernel-managed mechanisms. The Linux scheduler chooses among processes to find the most appropriate one to run on the CPU. All of the attributes and privileges of each process are stored in a data structure named task\_struct. Linux then maintains an array, named task, that contains pointers to every task\_struct in the system. The task array can hold a maximum of 512 task\_struct entries, thus the system can have a maximum of 512 processes at any one time. As well, the current, running task\_struct is pointed to by the current pointer. The task\_struct data structure contains all the information about a process. Its fields can be divided into a number of functional areas:-State-Scheduling Information-Identifiers-Inter-Process Communication-Links-Times and Timers-File System-Virtual Memory-Processor-Specific ContentThese areas each contain fields that completely define a process and allow the process to be multiprogrammed. StateThe state of a process is specified as waiting, running, stopped, and “zombie”. Waiting indicates that the process is waiting for a resource or for an event to occur. Running means that the process is either currently being executed on a CPU, or is ready and waiting to be executed on a CPU. Stopped indicates that the process has been stopped, which often happens when a program is being debugged. Zombie indicates that the process is a dead process. Scheduling InformationThe scheduling information contained in the task\_struct allows the scheduler to make a fair decision as to which processes should be allocated CPU time. In Linux, there is no pre-emption of processes by other processes. Another process cannot “take” the CPU away from the current, running processes. It must wait until the running process relinquishes the CPU, such as when it has to wait for a system event like I/O. However, Linux does use pre-emptive scheduling to prevent a process without system calls from taking a disproportionate amount of CPU time. Each process is given a time slice, usually about 200ms on the CPU, at the end of which, a new process is selected, and the old process must again wait for its turn. The registers, program-counter, and other process-specific data of the outgoing process are saved into its task\_struct for the next time that it receives processor attention. The registers, program-counter, and other process-specific data of the new process are then loaded from its task\_struct into the CPU. The Linux scheduler determines which process should be allocated CPU time according to four fields in the task\_struct scheduling information:policy: The policy field indicates which scheduling policy should be applied to this task\_struct. Linux supports two scheduling policies: normal and real-time. Real time processes will always have a higher priority than normal processes. Real-time processes also have two different types of policy: FIFO and round-robin. This is indicated here.

priority: The priority field indicates the priority of the process. This is also the amount of time that the process will be allowed to run on the processor when it receives its turn. The time is measured in “jiffies”. rt\_priority: The field indicates the priority of a real-time process. counter: This is the total amount of time left that the process is allowed to run on the processor for this turn. Each clock tick that the process has CPU-attention, this number is decremented by one. The process scheduler looks through the processes waiting for CPU time and gives the real-time processes a higher priority weighting than the normal processes. The priority assigned to a normal process is counter. The priority assigned to a real-time process is counter plus 1000. Because the current process has consumed some of its time-slice, indicated by the counter, it is has a disadvantage to other processes that have equal priority in the system. If several processes have the same priority, the one nearest the front of the queue will be selected and the current process will be placed at the rear of the queue. Linux supports symmetric multiprocessing (SMP). In an SMP system, each task\_struct also contains the number of the processor that it is currently running on, stored in processor, as well as the last processor it ran on, stored in last\_processor. Processes that last ran on a certain processor gain a slight priority advantage over processes last run on other processor, because there is usually a slight performance overhead in switching processors. A process can be restricted to one processor as well, by specifying a certain processor number in the processor\_mask. IdentifiersEvery process in the system has a unique process identifier. It is a number. Each process also has user and group identifiers, which allow and disallow the process to have access to certain files and devices. According to the user and group identifiers, the process may be restricted to no access, read-only access, write access, etc. for certain files. LinksThe links information in the task\_struct allows the process to be related to the process that created it. In Linux, every process in the system has a parent process, except for the initial system process. New processes are not created, but are cloned from previous processes. The task\_struct of each process has pointers to its parent process, sibling processes, and children processes. (You can see a tree of the family relationships of the running processes in Linux by typing pstree at the command prompt.) Times and TimersInformation pertaining to time in the task\_struct are the process creation time and the consumed CPU time during the process’s full lifetime. Each clock tick that this process has CPU attention, one “jiffy” is incremented in this field. As well, information about interval timers may be stored in the task\_struct. Interval timers are used by processes to signal themselves when a certain time interval expires. Every clock tick, the jiffies in the interval timers are decremented, until they expire, signaling the process. File SystemThe task\_struct contains two pointers to data structures that maintain file system information that is specific to the process. One of these pointers is fs, and it points to the fs\_struct data structure. The other pointer is named files, and it points to the files\_struct data structure. The fs\_struct contains pointers to the process’s VFS inodes as well as the value of the umask, which is the default mode that new files will be created in. The files\_struct contains the pointers to a file data structure for each of the files that the process is using. files\_struct can contain a maximum of 256 different file pointers. The file data structures, pointed to by the files\_struct, contain all relevant information about the process’s use of the files, for example, the position in the file that the next read or write will occur. There is always at least three files open for each process, standard input, standard output, and standard error. Virtual MemoryLinux uses demand paging. To implement this, in each process’ task\_struct, there is a pointer to an mm\_struct data structure that holds all information about the contents of each process’ virtual memory space, as well as contains information about the process’ loaded executable image and has a pointer to the process’ page table. Processor-Specific ContentThe process-specific content that must be saved when the CPU switches to serve another process is saved here. The registers, stacks, and program counter must be stored for the next time the process receives time on the CPU. The management of processes in Linux is done primarily through the maintenance of the task\_struct data structure for each process. The task\_struct contains all information about the process that is necessary to allow it to execute in this multiprogrammed environment. File ManagerFiles are organized on the Linux platform in a way quite similar to the UNIX platform. On a Unix system a file does not have to necessarily be a physical data block on a hard disk. Files can be used to represent other file system objects such as device drivers or network sockets. Underlying the Linux file system is the Virtual File System (VFS)The Virtual File System As with some of its other system features, Linux takes advantage of the principles of object-oriented design in the Virtual File System. Each object in the file system has two parts to it, the attributes of the object, and the operations to manipulate those objects. Within the VFS, there are three main object types as follows:+ Inode-object+ File-object+ File-system objectEach of the above objects has a defined set of operations that are accessed from a function table that each object points to. With this capability the VFS can perform a generic operation on these objects without having to know what kind of object it is actually dealing with. Therefore the object could be a networked file, a disk file, etc, and this would be transparent to the file system and to the user. An example of the power of the VFS is in the Linux Proc File System. This system treats each directory as an active process, and each directory contains information about that process. As a result, instead of querying the process manager in the operating system, information about active processes is read directly from the file system.File System HierarchyThe file-system object represents a connected set of files that forms a self contained directory hierarchy. The operating-system kernel maintains a single file-system object for each disk device mounted as a file system and for each networked file system currently connected. The VFS identifies every inode by a unique file system-inode number pair, and it finds the inode corresponding to a particular inode number by asking the file-system object to return the inode with that number. An inode-object represents the file as a whole, and a file-object represents a point of access to the data in the file. Directory files themselves are inode-objects, and are dealt with slightly differently from other files. The Linux programming interface defines a number of operations on directories, such as creating, deleting, and renaming a file in a directory. Figure 1.1 Linux File System HierarchyPhysical Disk Allocation On most standard Linux distributions the file system used is called Extfs32. This file system is based on the Berkley Software Distribution (BSD) Fast File System (ffs). Files are stored in a set of data blocks that are located through pointers in index blocks with up to three levels of indirection, the first index block being the files inode-object. Directories are a list of entries that include the following:+ the length of the entry+ the name of the file+ the inode number of the inode-object for the entryDisk space is allocated to files in blocks of 1 kilobyte, although block sizes of 2 and 4 kilobytes are available. These blocks are purposefully made small to guard against internal block fragmentation. Also blocks that are logically adjacent (such as in one file) are placed into physically adjacent blocks on the physical disk. The Ext2fs file system is partitioned into multiple block groups. When allocating a file, ext2fs must first select the block group for that file. For data blocks, it attempts to choose the same block group as that the file’s inode-object has been allocated. For inode allocations, it selects the same block group as the file’s parent directory. The Linux Operating System uses innovative means for file management. Fragmentation is very low and the file system is very scalable to other system uses. Device Manager The device I/O system in Linux is also very similar to the UNIX system. Linux device drivers work through a special kernel code that directly accesses the hardware. The kernel uses the special files to make the services that the device offers to normal user programs. One end of the file can be opened normally and the other end is attached to the kernel. So: hardware kernel special file user program and the same path back from user program to hardware; this is the general idea of how the I/O system works in Linux. Linux splits all devices into three classes: – block devices- character devices- network devices.Block devicesBlock devices are devices that allow random access to fixed-sized blocks of data, such as hard disk and floppy disk. Block devices provide functionality to ensure disk access is as fast as possible. In order to achieve this functionality, two system components are introduced: the block buffer cache and the request manager. l Block buffer cache: a pool of buffers for active I/O, and a cache for completed I/O.l Request manager: software that reads and writes buffer contents to and from a block-device driver.Character deviceDevices do not need to support all the functionality of regular files, such as a mouse. The kernel performs almost no preprocessing, but rather passes on the request to the device and let the device deal with it.Network deviceNetwork devices are dealt with differently from block and character devices. Networking is a key area of functionality for Linux. Linux supports the standard Internet protocols used for most UNIX-to-UNIX communications, but it also implements a number of protocols native to non-UNIX operating systems. These protocols are implemented under Linux as drivers that, at one end appears to the terminal system, and at the other end appears to the networking system. Three layers of software implement the Linux kernel: the socket interface, protocol drivers, and network-device drivers. ReferencesOperating System Concepts 5th Edition. Silberschatz, Abraham. Galvin, Peter Baer.http://www.grad.math.uwaterloo http://www.linux.org/http://www.itknowledge.com/reference/archive/0672310120/ewtoc.htmlSlackware Linux Unleashed, Third Edition (Publisher: Macmillan Computer Publishing) Author(s): Timothy Parker et al ISBN: 0672310120 Publication Date: 03/10/97