I/O Ports

Each device connected to the I/O bus has its own set of I/O addresses, which are usually called *I/O ports*. In the IBM PC architecture, the I/O address space provides up to 65,536 8-bit
I/O ports. Two consecutive 8-bit ports may be regarded as a single 16-bit port, which must start on an even address. Similarly, two consecutive 16-bit ports may be regarded as a single 32-bit port, which must start on an address that is a multiple of 4. Four special assembly language instructions called in, ins, out, and outs allow the CPU to read from and write into an I/O port. While executing one of these instructions, the CPU makes use of the address bus to select the required I/O port and of the data bus to transfer data between a CPU register and the port. I/O ports may also be mapped into addresses of the physical address space: the processor is then able to communicate with an I/O device by issuing assembly language instructions that operate directly on memory (for instance, mov, and, or, and so on). Modern hardware devices tend to prefer mapped I/O, since it is faster and can be combined with DMA.

An important objective for system designers is to offer a unified approach to I/O programming without sacrificing performance. Toward that end, the I/O ports of each device are structured into a set of specialized registers. The CPU writes into the *control register* the commands to be sent to the device and reads from the *status register* a value that represents the internal state of the device. The CPU also fetches data from the device by reading bytes from the *input register* and pushes data to the device by writing bytes into the *output register*.

Associating Files with I/O Devices

UNIX-like operating systems are based on the notion of a *file*, which is just an information container structured as a sequence of bytes. According to this approach, I/O devices are treated as files; thus, the same system calls used to interact with regular files on disk can be used to directly interact with I/O devices. As an example, the same write( ) system call may be used to write data into a regular file, or to send it to a printer by writing to the */dev/lp0* device file. Let's now examine in more detail how this schema is carried out.

Device Files

Device files are used to represent most of the I/O devices supported by Linux. Besides its name, each device file has three main attributes:

Type

Either *block* or *character*.

Major number

A number ranging from 1 to 255 that identifies the device type. Usually, all device files having the same major number and the same type share the same set of file operations, since they are handled by the same device driver.

Minor number

A number that identifies a specific device among a group of devices that share the same major number. The mknod( ) system call is used to create device files. It receives the name of the device file, its type, and the major and minor numbers as parameters. The last two parameters are merged in a 16-bit dev_t number: the eight most significant bits identify the major number, while the remaining ones identify the minor number. The MAJOR and MINOR macros extract the two values from the 16-bit number, while the MKDEV macro merges a major and minor number into a 16-bit number. Actually, dev_t is the data type specifically used by application programs; the kernel uses the kdev_t data type. In Linux 2.2 both types reduce to an unsigned short integer, but kdev_t will become a complete device file descriptor in some future Linux version.
Device files are usually included in the */dev* directory. The following illustrates the attributes of some device files. Notice how the same major number may be used to identify both a character and a block device.

Name Type Major Minor Description

/dev/fd0 block 2 0 Floppy disk
/dev/hda block 3 0 First IDE disk
/dev/hda2 block 3 2 Second primary partition of first IDE disk
/dev/hdb block 3 64 Second IDE disk
/dev/hdb3 block 3 67 Third primary partition of second IDE disk
/dev/ttyp0 char 3 0 Terminal
/dev/console char 5 1 Console

/dev/lp1 char 6 1 Parallel printer
/dev/ttyS0 char 4 64 First serial port
/dev/rtc char 10 135 Real time clock
/dev/null char 1 3 Null device (black hole)

Usually, a device file is associated with a hardware device, like a hard disk (for instance,
/dev/hda), or with some physical or logical portion of a hardware device, like a disk partition
(for instance, */dev/hda2*). In some cases, however, a device file is not associated to any real hardware device, but represents a fictitious logical device. For instance, */dev/null* is a device
file corresponding to a "black hole": all data written into it are simply discarded, and the file appears always empty. As far as the kernel is concerned, the name of the device file is irrelevant. If you created a device file named */tmp/disk* of type "block" with major number 3 and minor number 0, it would be equivalent to the */dev/hda* device file shown in the table. On the other hand, device filenames may be significant for some application programs. As an example, a communication program might assume that the first serial port is associated with the */dev/ttyS0* device file. But usually most application programs can be configured to interact with arbitrarily named device files.

File System Management

The Second Extended File system (Ext2) is native to Linux and is used on virtually every Linux system, Furthermore, Ext2 illustrates a lot of good practices in its support for modern file system features with fast performance.
General Characteristics Each Unix-like operating system makes use of its own file system. Although all such file systems comply with the POSIX interface, each of them is implemented in a different way.

The first versions of Linux were based on the Minix filesystem. As Linux matured, the *Extended Filesystem (Ext FS)* was introduced; it included several significant extensions but offered unsatisfactory performance. The *Second Extended Filesystem (Ext2)* was introduced in 1994: besides including several new features, it is quite efficient and robust and has become the most widely used Linux filesystem.

The following features contribute to the efficiency of Ext2:

• When creating an Ext2 filesystem, the system administrator may choose the optimal block size (from 1024 to 4096 bytes), depending on the expected average file length.

For instance, a 1024 block size is preferable when the average file length is smaller than a few thousand bytes because this leads to less internal fragmentation—that is, less of a mismatch between the file length and the portion of the disk that stores it. On the other hand, larger block sizes are usually preferable for files greater than a few thousand bytes because this leads to fewer disk transfers, thus reducing system overhead.

• When creating an Ext2 filesystem, the system administrator may choose how many inodes to allow for a partition of a given size, depending on the expected number of files to be stored on it. This maximizes the effectively usable disk space.

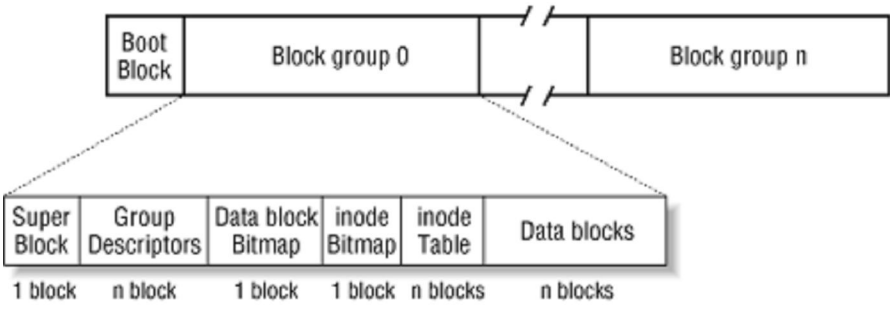
• The file system partitions disk blocks into groups. Each group includes data blocks and inodes stored in adjacent tracks. Thanks to this structure, files stored in a single block group can be accessed with a lower average disk seek time.

• The filesystem *preallocates* disk data blocks to regular files before they are actually used. Thus, when the file increases in size, several blocks are already reserved at physically adjacent positions, reducing file fragmentation.

• Fast symbolic links are supported. If the pathname of the symbolic link has 60 bytes or less, it is stored in the inode and can thus be translated without reading a data block.

Disk Data Structures

Figure 2 Layouts of an Ext2 partition and of an Ext2 block group



The first block in any Ext2 partition is never managed by the Ext2 filesystem, since it is reserved for the partition boot sector. The rest of the Ext2 partition is split into *block groups* , each of which has the layout shown in Figure 2. As you will notice from the figure, some data structures must fit in exactly one block while others may require more than one block. All the block groups in the filesystem have the same size and are stored sequentially, so the kernel can derive the location of a block group in a disk simply from its integer index. Block groups reduce file fragmentation, since the kernel tries to keep the data blocks belonging to a file in the same block group if possible. Each block in a block group contains one of the following pieces of information:

• A copy of the filesystem's superblock
• A copy of the group of block group descriptors
• A data block bitmap
• A group of inodes
• An inode bitmap
• A chunk of data belonging to a file; that is, a data block

If a block does not contain any meaningful information, it is said to be free.

Superblock

An Ext2 disk superblock is stored in an ext2_super_block structure. The __u8, __u16, and __u32 data types denote unsigned numbers of length 8, 16, and 32 bits respectively, while the __s8, __s16, __s32 data types denote signed numbers of length 8, 16, and 32 bits. The s_inodes_count field stores the number of inodes, while the s_blocks_count field stores the number of blocks in the Ext2 filesystem. The s_log_block_size field expresses the block size as a power of 2, using 1024 bytes as the unit. Thus, denotes 1024-byte blocks, 1 denotes 2048-byte blocks, and so on. These_log_frag_size field is currently equal to s_log_block_size, since block fragmentation is not yet implemented. The s_blocks_per_group, s_frags_per_group, and s_inodes_per_group fields store the number of blocks, fragments, and inodes in each block group, respectively. Some disk blocks are reserved to the superuser (or to some other user or group of users selected by the s_def_resuid and s_def_resgid fields). These blocks allow the system administrator to continue to use the filesystem even when no more free blocks are available for normal users. The s_mnt_count, s_max_mnt_count, s_lastcheck, and s_checkinterval fields set up the Ext2 filesystem to be checked automatically at boot time. These fields cause */sbin/e2fsck* to run after a predefined

number of mount operations has been performed, or when a predefined amount of time has elapsed since the last consistency check. (Both kinds of checks can be used together.) The consistency check is also enforced at boot time if the filesystem has not been cleanly unmounted (for instance, after a system crash) or when the kernel discovers some errors in it. The s_state field stores the value if the filesystem is mounted or was not cleanly unmounted, 1 if it was cleanly unmounted, and 2 if it contains errors.

Group Descriptor And Bitmap

Each block group has its own group descriptor, an ext2_group_desc structure
The bg_free_blocks_count, bg_free_inodes_count, and bg_used_dirs_count fields are used when allocating new inodes and data blocks. These fields determine the most suitable block in which to allocate each data structure. The bitmaps are sequences of bits, where the value specifies that the corresponding inode or data block is free and the value 1 specifies that it is used. Since each bitmap must be stored inside a single block and since the block size can be 1024, 2048, or 4096 bytes, a single bitmap describes the state of 8192, 16,384, or 32,768 blocks.

CHAPTER THREE – SOLARIS OPERATING SYSTEM

HISTORY

The history of Solaris, a Unix-based operating system developed by Sun Microsystems, displays that company's ability to be innovative and flexible. Solaris was introduced in the year 1987 out of an alliance between AT&T and Sun Microsystems to combine the leading Unix versions (BSD, XENIX, and System V) into one operating system.In 1991, Sun replaced it's existing Unix operating system (SunOS 4) with one based on SVR4. This new OS, Solaris 2, contained many new advances, including use of the Open Windows graphical user interface, NIS+, Open Network Computing (ONC) functionality, and was specially tuned for symmetric multiprocessing.

This kicked off Solaris' history of constant innovation, with new versions of Solaris being released almost annually over the next fifteen years. Sun was constantly striving to stay ahead of the curve, while at the same time adapting Solaris to the existing, constantly evolving wider computing world. The catalogue of innovations in the Solaris OS are too numerous to list here, but a few milestones are worth mentioning.

- Solar 2.5.1 in 1996 added CDE, the NFSv3 file system and NFS/TCP, expanded user and group IDs to 32 bits, and included support for the Macintosh PowerPC platform.
- Solaris 2.6 in 1997 introduced WebNFS file system, Kerberos 5 security encryption, and large file support to increase Solaris' internet performance.
- Solaris 2.7 in 1998 (renamed just Solaris 7) included many new advances, such as native support for file system meta-data logging (UFS logging). It was also the first 64-bit release, which dramatically increased its performance, capacity, and scalability.
- Solaris 8 in 2000 took it a step further was the first OS to combine datacenter and dot-com requirements, offering support for IPv6 and IPSEC, Multipath I/O, and IPMP.
- Solaris 9 in 2002 saw the writing on the wall of the server market, dropped OpenWindows in favour of Linux compatibility, and added a Resource Manager, the Solaris Volume Manager, extended file attributes, and the iPlanet Directory Server.
- Solaris 10, the current version, was released to the public in 2005 free of charge and with a host of new developments. The latest advances in the computing world are constantly being incorporated in new versions of Solaris 10 released every few months.

To mention just a few, Solaris features more and more compatibility with Linux and IBM systems, has introduced the Java Desktop System based on GNOME, added Dynamic Tracing (Dtrace), NFSv4, and later the ZFS file system in 2006.

Also in 2006, Sun set up the OpenSolaris Project. Within the first year, the OpenSolaris community had grown to 14,000 members with 29 user groups globally, working on 31 active projects. Although displaying a deep commitment to open-source ideals, it also provides Sun with thousands of developers essentially working for free.

SOLARIS PROCESSES

The process is one of the fundamental abstractions of Unix. Every object in Unix is represented as either a **file** or a **process**(with the introduction of the /proc structure, there has been an effort to represent even processes as files)**.** Processes are usually created with **fork**or a less resource alternative such as **fork1 or vfork.fork**duplicates the entire process context, while fork1 only duplicates the context of the calling thread. This can be useful for example, when **exec**will be called shortly.

Solaris like other UNIX systems, provide two modes of operation: **user mode** and **kernel (or system mode).** Kernel mode is a more privileged mode of operation.

Processes can be executed in either mode, but user processes usually operate in user mode.

SOLARIS PROCESS SCHEDULING

In Solaris, highest priorities are scheduled first. Kernel thread scheduling information can be revealed with ps –elcL**.** A process can exist in one of the following states:

- Running
- Sleeping
- Ready

✓ KERNEL THREADS MODEL

The kernel threads model consist of the following objects:

- **Kernel threads –** this is what is scheduled/executed on a processor
- **User threads –** the user-level thread state within a process
- **Process -** the object that tracks the execution environment of a program
- **Lightweight process (lwp) –** Execution context for a user tread. It associates a user thread with a kernel thread.

In Solaris 10 kernel, kernel services and tasks are executed as kernel threads. When a user thread is created, the associated lwp and kernel threads are also created and linked to the user thread.

KERNEL THREADS MODEL

An application's parallelism is the degree of parallel execution achieved.This is limited by the number of processors available in the hardware configuration. Concurrency is the maximum achievable parallelism in a theoretical machine that has an unlimited number of processors.

Threads are frequently used to increase an application's concurrency. A thread represents a relatively independent set of instructions within a program. A thread is a control point within a process. It shares global resources within the context of the process (address space, open files, user credentials, quotas, etc). Threads also have private resources (program counter, stack, register context, etc).

The main benefit of threads (as compared to multiple processes) is that the context switches are much cheaper than those required to change current processes. Even within a single-processor environment, multiple threads are advantageous because one thread may be able to progress even though another thread is blocked while

waiting for a resource. Inter-process communication also takes considerably less time for threads than for processes, since global data can be shared instantly.

The kernel threads model consist of the following objects:

- **Kernel threads –** this is what is scheduled/executed on a processor
- **User threads –** the user-level thread state within a process
- **Process -** the object that tracks the execution environment of a program
- **Lightweight process (lwp) –** Execution context for a user tread. It associates a user thread with a kernel thread.

In Solaris 10 kernel, kernel services and tasks are executed as kernel threads. When a user thread is created, the associated lwp and kernel threads are also created and linked to the user thread.

Kernel Threads

A kernel thread is the entity that is scheduled by the kernel. If no lightweight process is attached, it is also known as a system thread. It uses kernel text and global data, but has its own kernel stack, as well as a data structure to hold scheduling and synchronization information.

Kernel threads can be independently scheduled on CPUs. Context switching between kernel threads is very fast because memory mappings do not have to be flushed.

Lightweight Processes

A lightweight process can be considered as the swappable portion of a kernel thread.Another way to look at a lightweight process is to think of them as "virtual CPUs" which perform the processing for applications. Application threads are attached to available lightweight processes, which are attached to a kernel thread, which is scheduled on the system's CPU dispatch queue. LWPs can make system calls and can block while waiting for resources. All LWPs in a process share a common address space. IPC (inter-process communication) facilities exist for coordinating access to shared resources.

By default, one LWP is assigned to each process; additional LWPs are created if all the process's LWPs are sleeping and there are additional user threads that libthread can schedule. The programmer can specify that threads are bound to LWPs.

User Threads

User threads are scheduled on their LWPs via a scheduler in libthread. This scheduler does implement priorities, but does not implement time slicing. If time slicing is desired, it must be programmed in. Locking issues must also be carefully considered by the programmer in order to prevent several threads from blocking on a single resource.

Each thread has the following characteristics:

- Has its own stack.
- Shares the process address space.
- Executes independently (and perhaps concurrently with other threads).
- Completely invisible from outside the process.
- Cannot be controlled from the command line.
- No system protection between threads in a process; the programmer is responsible for interactions.
- Can share information between threads without IPC overhead.

- ✓ **PRIORITY MODEL**

The Solaris kernel is fully **preemptible.** This means that all threads, including the threads that support the kernel's own activities can be deferred to allow a higher-priority thread to run.

Solaris recognizes 170 different priorities, 0-169. Within these priorities fall a number of different scheduling classes:

- **TS (Timeshare):** This is the default class for processes and their associated kernel threads. Priorities falling within this class range 0-59 and are dynamically adjusted in an attempt to allocate processor resources evenly.
- **IA (Interactive):** This is an enhanced version of the TS class that applies to the in-focus window in the GUI. Its intent is to give extra resources to processes associated with that specific window. Like TS, IA's range is 0-59.
- **FSS (Fair-share scheduler):** This class is share-based rather than priority-based. Threads managed by FSS are scheduled based on their associated shares and the processor's utilization. FSS also has a range 0-59.
- **FX (Fixed-priority):** The priorities for threads associated with this class are fixed (in other words, they do not vary dynamically over the lifetime of the thread). FX also has a range 0-59.

- **SYS (system):** The SYS class is used to schedule kernel threads. Threads in this class are “bound” threads, which mean that they run until they block or complete. Priorities for SYS threads are in the 60-99 range.
- **RT (Real-time):** Threads in the RT class are fixed-priority, with a fixed time quantum. Their priorities range 100-159, so an RT thread will preempt a system thread. Of these, FSS and FX were implemented in Solaris 9.

Fair Share Scheduler

The default Timesharing (TS) scheduling class in Solaris attempts to allow each process on the system to have relatively equal CPU access. The nice command allows some management of process priority, but the new Fair Share Scheduler (FSS) allows more flexible process priority management that integrates with the project framework. Each project is allocated a certain number of CPU shares via the project. CPU-shares resource control and each project is allocated CPU time based on its CPU-shares value divided by the sum of the CPU-shares values for all active projects. Anything with a zero CPU-shares value will not be granted CPU time until all projects with non-zero CPU-shares are done with the CPU. The maximum number of shares that can be assigned to any one project is 65535.

FSS can be assigned to processor sets, resulting in more sensitive control of priorities on a server than raw processor sets.

The Fair Share Scheduler should not be combined with the TS, FX (fixed-priority) or IA (interactive) scheduling classes on the same CPU or processor set. All of these scheduling classes use priorities in the same range, so unexpected behavior can result from combining FSS with any of these. (There is no problem, however, with running TS and IA on the same processor set.)

Time Slicing for FSS

In FSS, the time quantum is the length of time that a thread is allowed to run before it has to release the processor. The QUANTUM is reported in ms. (The output of the above command displays the resolution in the RES parameter. The default is 1000 slices per second.

Fixed Priority Scheduling

FX scheduler sets policy scheduling for processes used by applications and users. These processes are fixed. For example, priocnt1 and dispadminare two utilities that control the Fixed-Priority Scheduling. The FX class is the same priority as the FSS, IA, and TS classes.

THE SOLARIS BOOTUP AND SHUTDOWN

The Solaris Boot process is made up of four phases and is illustrated in the figure below:

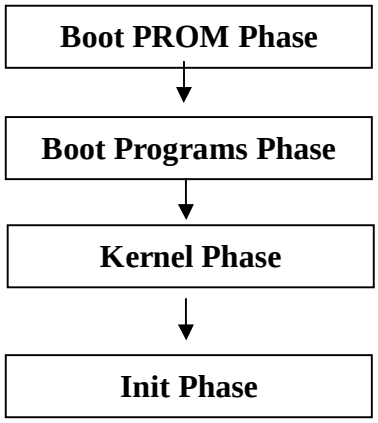


FIG. 1 SOLARIS BOOTUP PHASES

Boot PROM Phase: The hardware tests and initializes itself

Boot Programs Phase: The initial boot programs are loaded into the memory.

Kernel Phase: The kernel loads itself and its modules into memory and then unloads the boot programs from memory.

Init Phase: The init process is started by the kernel. The **init**process then executes the run control scripts.

Phase 1: The Boot PROM Phase

During this phase of the boot up, the system first powers up and checks itself. On the PROM chip is a program known as the monitor program. This program is used for initial system tests and diagnostics. It tests the system's memory, CPU and

mother board. It does not test all devices attached to the server, only the server's main components.

If a third-party device is attached to anSBus controller, the device driver is then loaded from a firmware chip on the device (some manufacturers don't include device drivers on the hardware itself). If the open boot variable **diag-level** is set to **max** and the variable **diag-switch**is set to **true**the system will perform extensive diagnostics during the power on self test. The banner information looks like the figure below:

Sun blade 100 (UltraSPARC-IIe) Keyboard present
OpenBoot 4.0, 128MB memory installed, Serial #50632835.
Ethernet Address 0:3:ba:2:c2:3d, Host ID: 8323c12b.

FIG. 2 Output from the banner command.

After the power on self test is complete, the boot process stops at the **O.K** prompt or continues to boot the Solaris operating system. This depends on the value of the OpenBoot**auto-boot**? variable:

- If the auto-boot variable is set to true, the system boots the device specified in the **boot-device** variable. The default boot device OpenBoot value on most system is the **disk or disk:a.** A second boot device **(net)** can be also be specified. If for some reason the first boot device does not work, the second boot device is tried.

- If the **auto-boot?**variable is set to **false** the system stops at the **OK** prompt.

Phase 2: Boot Program Phase

This phase starts when the system has checked itself and starts to load the **bootblk** program from the boot device. The **bootblk**program is a smallsection of code on the first sector of the first track of the first drive of the hard drive or tape device. When bootblk runs, it shows a message like

Fcode UFS Reader 1.12 00/07/17 15:48:16

Bootblkhas only one function. It loads the**ufsboot** program into the memory and then dies. When the **Fcode UFS Reader …bootblk**has done its work. The following message should now appear:

Loading: /platform/SUNW,Sun-Blade-100/ufsboot

Loading :/platform/sun4u/ufsboot

The ufsboot program loads the kernel into memory. After the program is loaded into memory, the ufsboot program dies.

It is important that a system administrator understand what is happening with the **ufsboot**program and the **bootblk** program. If the system messages shown above do not appear, the server may be dead or something may be wrong with these two programs, which will then need to be reloaded or repaired.

Phase 3: Kernel Phase

This phase starts when the initial boot programs **bootblk** and **ufsboot**have been loaded and the kernel is now starting to load. The kernel can be thought as the core program that defines the Solaris operating. The kernel uses the **ufsboot**program to read kernel modules into memory. A kernel module can be thought of as a dynamic piece of software code. Only the modules that are needed are loaded into the kernel. This makes the kernel faster and more efficient than if it always had to load all its modules into memory. After enough modules are loaded into memory, the **ufsboot** program dies.

When the front slash symbol (/) starts to swirl, the kernel is starting to load. The SunOS Release is now also shown. This indicates that the boot device is booting and working. If there are any further problems with the boot process, they will most likely be caused by an error in a run control script.

Phase 4: The Init Phase

The init phase starts after the kernel has loaded itself and its modules into memory. The **sched**process is the first process to be loaded. It has a PID (Process Identification Number) of zero (0), as shown with the **ps–ef**command. The**sched** process is responsible for the scheduling policy and priority of processes. After **sched** starts up, the process called **init** is started, with a PID of one (1). The innit process reads a text file /etc/innittab. Among other things, this file defines the default run level and controls how the init process calls up and executes run control scripts.

MEMORY MANAGEMENT

- **The process Memory Usage**

  The /usr/proc/bin/pmap command is available in Solaris 2.6 and above. It can help pin down which process is memory hog. /usr/proc/bin/pmap –x PID prints out details of memory use by a process. Summary statistics regarding process size can be found in the RSS column of ps – ly or top. dbx, the debugging utility in the SunPro package, has extensive memory leak detection built in. The source code will need to be compiled with the –g flag by the appropriate SunPro compiler. Ipcs –mb shows memory statistics for shared memory. This may be useful when attempting to size memory to fit expected traffic.

- **Swap Space**

  The Solaris virtual memory system combines physical memory with available swap space via **swapfs**. If insufficient total virtual memory space is provided, new processes will be unable to open.

- **Paging**

  Solaris uses both common types of paging in its virtual memory system. These types are:

- **Swapping**(swaps out all memory associated with a user process) and
- **Download paging** (swaps out the not recently used pages)

Which method is used is determined by comparing the amount of available memory with several key parameters

- **Solaris 8 Paging**

  Solaris 8 uses a different algorithm for removing from memory. This new architecture is known as the cyclical page cache. The cyclical page cache uses a file system free list to cache file system data only. Other memory objects are managed on a separate free list

SECURITY

File Integrity and Secure Execution

System administrators can detect possible attacks on their systems by monitoring for changes to file information. In the Solaris 10 OS, binaries are digitally signed, so administrators can track changes easily, and all patches or enhancements are embedded with digital signatures, eliminating the false positives associated with upgrading or patching file integrity-checking software.

User and Process Rights Management

In traditional UNIX platform-based operating systems, applications and users often need administrative access to perform their jobs. However, most implementations offer just one level of higher privilege: root or superuser. This means that any user or application given root access has the ability to make major changes to the operating system—and is typically the target of hacking attempts. The Solaris 10 OS offers unique User Rights Management (also known as role-based access control, or RBAC) and Process Rights Management (also known as privileges)

Network Service Protection

The Solaris 10 OS ships with Solaris IP Filter firewall software preinstalled. This integrated firewall can reduce the number of network services that are exposed to attack and provides protection against maliciously crafted networking packets. Starting in Solaris 10 8/07, the IP Filter firewall can also filter traffic flowing between Solaris Containers when it is configured in the Global Zone. In addition, TCP Wrappers are integrated into the Solaris 10 OS, limiting access to service-based allowed domains.

Cryptographic Services and Encrypted Communication

For high-performance, system-wide cryptographic routines, the Solaris Cryptographic Framework adds a standards-based, common API that provides a

single point of administration and uniform access to both software and hardware-accelerated, cryptographic functions. The pluggable Solaris Cryptographic Framework can balance loads across accelerators, increasing encrypted network traffic throughput, and it is available to applications written to use Public Key Cryptography Standards (PKCS) #11, Sun Java Enterprise System, NSS, OpenSSL, and Java Cryptographic Extension software.

Flexible Enterprise Authentication

The Solaris 10 OS delivers a number of flexible authentication features. At the foundation of Solaris is support for Pluggable Authentication Mechanism (PAM), which make it possible to add authentication services to Solaris dynamically. Sun and third-party vendors provide many PAM modules and customers can create their own to meet specific security needs.

Repeatable Security Hardening and Monitoring

New features in the Solaris 10 OS make it easier than ever to minimize and harden a system. The Reduced Networking Metacluster install option creates a minimized Solaris OS image, ready for administrators to add functionality and services in direct support of their system's purpose.

Mandatory Access Control and Labeling

If your system requirements include privacy, increased accountability, and reduced risk of security violations, then Solaris Trusted Extensions is for you. A standard part of Solaris, true multi-level security is available for the first time in a commercial-grade operating system that runs all your existing applications and is supported on over 1,200 x64/x86 and SPARC platforms.

- **WEAKNESS AND STRENGHT**

A security weakness in Solaris Trusted Extensions Policy configuration may allow a remote unprivileged user who has authorized or unauthorized access to the X server, to leverage an additional vulnerability which could lead to arbitrary code execution as a local privileged or unprivileged user.

Sun has acknowledged a weakness in Pidgin on Solaris, which can be exploited by malicious people to cause a DoS (Denial of Service).

- **CONCLUSION**

The development of the Solaris OS demonstrates Sun Microsystems' ability to be on the cutting edge of the computing world without losing touch with the current computing environment. Sun regularly releases new versions of Solaris

incorporating the latest development in computer technology, yet also included more cross-platform compatibility and incorporating the advances of other systems. The OpenSolaris project is the ultimate display of these twin strengths- Sun has tapped into the creative energy of developers across the world and receives instant feedback about what their audience wants and needs. If all software companies took a lesson from Sun, imagine how exciting and responsive the industry could be.

CHAPTER 4 : MS-DOS

1.0 INTRODUCTION

MS DOS is an acronym that stands for **M**icro**S**oft **D**isk **O**perating **S**ystem. It is often referred to as **DOS**. It is an old operating system for x86-based personal computers, purchased by Microsoft that manages everything on your computer: hardware, memory, files. It is an operating system that existed prior to Windows.
MS-DOS was the most commonly used member of the DOS family of operating systems, and was the main operating system for personal computers during the 1980s up to mid 1990s. It was preceded by M-DOS (also called MIDAS), designed and copyrighted by Microsoft in 1979. MSDOS was written for the Intel 8086 family of microprocessors, particularly the IBM PC and compatibles. It was gradually replaced on consumer desktop computers by operating systems offering a graphical user interface (GUI), in particular by various generations of the Microsoft Windows operating system. MS-DOS developed out of **QDOS** (**Q**uick and **D**irty **O**perating **S**ystem), also known as 86-DOS. DOS, as with any operating system, controls computer activity. It manages operations such as data flow, display, data entry amongst other various elements that make up a system.
The role of DOS is to interpret commands that the user enters via the keyboard.
These commands allow the following tasks to be executed:
- file and folder management
- disk upgrades
- hardware configuration
- memory optimization
- program execution

These commands are typed after the prompt, in the case of MS-DOS (Microsoft DOS, the most well known): the drive letter followed by a backslash, for example: A:\ or C:\. And after them, the enter key. The files that make up DOS involves:
IO.SYS : This is a program to handle input/output to your peripheral devices. It stays in memory when you run applications programs

MSDOS.SYS: This is a program for application programs to use. It contains special subprograms to make many commonly needed operations easy for programmers. COMMAND.COM : This program accepts the commands you enter and runs the right program. CONFIG.SYS: Configures the hardware environment Mouse , Printer, Keyboard , Country codes (time, date, currency),other devices and system commands AUTOEXEC.BAT: Programs/commands to be run at system

start Batch file (automatically executing set of programs/commands) IO.SYS and MSDOS are loaded into the PC memory by a special program called a boot record each time you start up DOS . The command used to initialize new disks with DOS,FORMAT/S puts this on the disk along with IO.SYS and MSDOS.SYS

2.0 HISTORY

MS-DOS (Microsoft Disk Operating System) is a single-user, single-tasking computer operating system that uses a command line interface. In spite of its very small size and relative simplicity, it is one of the most successful operating systems that have been developed to date.

A Quick and Dirty History

When IBM launched its revolutionary personal computer, the IBM PC, in August 1981, it came complete with a 16-bit operating system from Microsoft, MS-DOS 1.0. This was Microsoft's first operating system, and it also became the first widely used operating system for the IBM PC and its clones. MS-DOS 1.0 was actually a renamed version of QDOS (Quick and Dirty Operating System), which Microsoft bought from a Seattle company, appropriately named Seattle Computer Products, in July 1981. QDOS had been developed as a clone of the CP/M eight-bit operating system in order to provide compatibility with the popular business applications of the day such as WordStar and dBase. CP/M (Control Program for Microcomputers) was written by Gary Kildall of Digital Research several years earlier and had become the first operating system for microcomputers in general use.

QDOS was written by Tim Paterson, a Seattle Computer Products employee, for the new Intel 16-bit 8086 CPU (central processing unit), and the first version was shipped in August, 1980. Although it was completed in a mere six weeks, QDOS was sufficiently different from CP/M to be considered legal. Paterson was later hired by Microsoft. Microsoft initially kept the IBM deal a secret from Seattle Computer Products. And in what was to become another extremely fortuitous move, Bill Gates, the not uncontroversial cofounder of Microsoft, persuaded IBM to let his company retain marketing rights for the operating system separately from the IBM PC project. Microsoft renamed it PC-DOS (the IBM version) and MS-DOS (the Microsoft version). The two versions were initially nearly identical, but they eventually diverged.

The acronym DOS was not new even then. It had originally been used by IBM in the 1960sin the name of an operating system (i.e., DOS/360) for its System/360 computer. At that time the use of disks for storing the operating system and data was considered cutting edge technology. Until its acquisition of QDOS, Microsoft had been mainly a vendor of computer programming languages. Gates and co-founder Paul Allen had written Microsoft BASIC and were selling it on disks and tape mostly to PC hobbyists.

MS-DOS soared in popularity with the surge in the PC market. Revenue from its sales fuelled Microsoft's phenomenal growth, and MS-DOS was the key to company's rapid emergence as the dominant firm in the software industry. This product continued to be the largest single contributor to Microsoft's income well after it had become more famous for Windows. Subsequent versions of MS-DOS featured improved performance and additional functions, not a few of which were copied from other operating systems. For example, version 1.25, released in 1982, added support for double-sided disks, thereby eliminating the need to manually turn the disks over to access the reverse side.

Version 2.0, released the next year, added support for directories, for IBM's then huge 10MB hard disk drive (HDD) and for 360KB, 5.25-inch floppy disks. This was followed by version 2.11 later in the same year, which added support for foreign and extended characters. Version 3.0 launched in 1984, added support for 1.2MB floppy disks and 32MB HDDs. This was soon followed by version 3.1, which added support for networks. Additions and improvements in subsequent versions included support for multiple HDD partitions, for disk compression and for larger partitions as well as an improved diskchecking utility, enhanced memory management, a disk defragmenter and an improved text editor.

The final major version was 7.0, which was released in 1995 as part of Microsoft Windows 95. It featured close integration with that operating system, including support for long filenames and the removal of numerous utilities, some of which were on the Windows 95 CDROM. It was revised in 1997 with version 7.1, which added support for the FAT32 file system on HDDs.

3.0 OPERATING SYSTEM FUNCTIONS

3.1 SCHEDULING

Scheduling is a key concept in computer multitasking, multiprocessing operating system and real-time operating system designs. **Scheduling** refers to the way processes are assigned to run on the available CPUs, since there are typically many

more processes running than there are available CPUs. This assignment is carried out by software's known as a **scheduler** and **dispatcher**.
Objectives of a scheduler

 CPU utilization - to keep the CPU as busy as possible.
 Throughput - number of processes that complete their execution per time unit.
 Turnaround - total time between submission of a process and its completion.
 Waiting time - amount of time a process has been waiting in the ready queue.
 Response time - amount of time it takes from when a request was submitted ntil the first response is produced.
 Fairness - Equal CPU time to each thread.

But MS-DOS is non-multitasking, and as such did not feature a scheduler. MS-DOS was not designed to be a multi-user or multitasking operating system, but many attempts were made to retrofit these capabilities. Since it does not perform scheduling functions, when you run a sub process synchronously on MS-DOS, make sure the program terminates and does not try to read keyboard input. If the program does not terminate on its own, you will be unable to terminate it, because MS-DOS provides no general way to terminate a process. Pressing "ctrl C" or `C-<BREAK>' might
sometimes help in these cases.

3.2 MEMORY MANAGEMENT

MS-DOS Memory Management Functions

 Provide students with a brief overview of memory management in the MS-DOS operating system. Mention that to run a second job, the user must close or pause the first file before opening the second.

 Point out that the Memory Manager uses a first-fit memory allocation scheme in early DOS versions because it is the most efficient strategy in a single-user environment.

 Discuss briefly the two forms of main memory, ROM and RAM. MS-DOS provides three memory management functions- allocate, deallocate, and resize (modify). For most programs, these three memory allocation calls are not used.When DOS executes a program, it gives all of the available memory, from the start of that program to the end of RAM, to the executing process. Any attempt to

allocate memory without first giving unused memory back to the system will produce an “insufficient memory” error.

ALLOCATE MEMORY

Function (ah): 48h
Entry parameters: bx- Requested block size (in paragraphs)
Exit parameters: If no error (carry clear):
ax:0 points at allocated memory block
If an error (carry set):
bx- maximum possible allocation size
ax- error code (7 or 8)
This call is used to allocate a block of memory. On entry into DOS, bx contains the size of the requested block in paragraphs (groups of 16 bytes). On exit, assuming no error, the ax register contains the segment address of the start of the allocated block. If an error occurs, the block is not allocated and the ax register is returned containing the error code.
If the allocation request failed due to insufficient memory, the bx register is returned containing the maximum number of paragraphs actually available.

DEALLOCATE MEMORY

Function (ah): 49h
Entry parameters: es:0- Segment address of block to be deallocated
Exit parameters: If the carry is set, ax contains the error code (7,9)
This call is used to deallocate memory allocated via function 48h above. The es register cannot contain an arbitrary memory address. It must contain a value returned by the allocate memory function. You cannot use this call to deallocate a portion of an allocated block. The modify allocation function is used for that operation.

MODIFY MEMORY ALLOCATION

Function (ah): 4Ah
Entry parameters: es:0- address of block to modify allocation size
bx- size of new block
Exit parameters: If the carry is set, then
ax contains the error code 7, 8, or 9
bx contains the maximum size possible (if error 8)

This call is used to change the size of an allocated block. On entry, es must contain the segment address of the allocated block returned by the memory allocation function. Bx must contain the new size of this block in paragraphs. While you can almost always reduce the size of a block, you cannot normally increase the size of a block if other blocks have been allocated after the block being modified. Keep this in mind when using this function.
Group C Page 8

3.3 FILE SYSTEM

Before we go any further, it would be a good idea to look at the DOS file system. The **file system** lets us store information in named **files**. You can call a file anything you like which might help you remember what it contains as long as you follow certain basic rules:

1. File names can be up to 8 characters long. You can use letters and digits but only a few punctuation marks (! $ % # ~ @ - ( ) _ { }). You can't exceed 8 characters or use spaces or characters like * or ? or +. Names are case-insensitive, i.e. it doesn't matter whether you use capitals or lowercase letters; "A" and "a" are treated as the same thing.
2. File names can also have an **extension** of up to three characters which describes the type of file. There are some standard extensions, but you don't have to use them.
Examples include COM and EXE for executable programs, TXT for text files, BAK
for backup copies of files, or CPP for C++ program files. The extension is separated by a dot from the rest of the filename.
For example, a file called FILENAME.EXT has an 8-character name (FILENAME) followed by a three-character extension (.EXT). You could also refer to it as filename.txt since case doesn't matter, but I'm going to use names in capitals for emphasis throughout this document. Files are stored in **directories**; a directory is actually just a special type of file which holds a list of the files within it. Since a directory is a file, you can have directories within directories. Directory names also follow the same naming rules as other files, but although they can have an extension they aren't normally given one (just an 8-character name). The system keeps track of your **current directory**, and if you just refer to a file using a name like FILENAME.EXT it's assumed you mean a file of that name in the current directory. You can specify a **pathname** to identify a file which includes the directory name as well; the directory is separated from the rest of the name by a backslash ("\"). For example, a file called LETTER1.TXT in a directory called

LETTERS can be referred to as LETTERS\LETTER1.TXT (assuming that the current directory contains the LETTERS directory as a subdirectory). If LETTERS contains a subdirectory called PERSONAL, which in turn contains a file called DEARJOHN.TXT, you would refer to this file as Group C Page 9

LETTERS\PERSONAL\DEARJOHN.TXT (i.e. look in the LETTERS directory for PERSONAL\DEARJOHN.TXT, which in turn involves looking in the PERSONAL subdirectory for the file DEARJOHN.TXT).
Every disk has a **root directory** which is the main directory that everything else is part of. The root directory is called "\", so you can use **absolute pathnames** which don't depend on what your current directory is. A name like \LETTERS\LETTER1.TXT always refers to the same file regardless of which directory you happen to be working in at the time; the "\" at the beginning means "start looking in the root directory", so \LETTERS\LETTER1.TXT means "look in the root directory of the disk for a subdirectory called LETTERS, then look in this subdirectory for a file called LETTER1.TXT". Leaving out the "\" at the beginning makes this a **relative pathname** whose meaning is relative to the current directory at the time. If you want to refer to a file on another disk, you can put a letter identifying the disk at the beginning of the name separated from the rest of the name by a colon (":"). For example,

A:\LETTER1.TXT refers to a file called LETTER1.TXT in the root directory of drive A. DOS keeps track of the current directory on each disk separately, so a relative pathname like A:LETTER1.TXT refers to a file called LETTER1.TXT in the currently-selected directory on drive A. For convenience, all directories (except root directories) contain two special names: "." refers to the directory itself, and ".." refers to the parent directory (i.e. the directory that contains this one). For example, if the current directory is \LETTERS\PERSONAL, the name ".." refers to the directory \LETTERS, "..\BUSINESS" refers to \LETTERS\BUSINESS, and "..\.." refers to the root directory "\".
Group C Page 10

3.4 PROCESS MANAGEMENT

MS-DOS boot process