



Solaris Bootcamp

Boot Process

**Author:
Tom Jackiewicz**

<i>Kernel Bootstrap and Initialization</i>	4
Overview	4
Bootstrap and Initialization	4
Device Trees	6
Device Path Names, Addresses, and Arguments	6
Device Alias	6
Displaying the Device Tree	7
<i>Booting and Testing Your System</i>	8
Booting Your System	8
Booting for the Expert User	9
Running Diagnostics	11
Testing the SCSI Bus	12
Testing Installed Devices	12
Testing the Diskette Drive	13
Testing Memory	13
Testing the Clock	14
Testing the Network Controller	14
Monitoring the Network	14
Displaying System Information	15
Resetting the System	15
Setting Configuration Variables	16
Displaying and Changing Variable Settings	16
Input and Output Control	16
<i>Sun Processor Architecture</i>	16
The SuperSPARC Family	17
HyperSPARC	17
MicroSPARC and MicroSPARC-II	18
UltraSPARC	18
The Ultra 5 and Ultra 10 (Darwin Series)	19
The Ultra 1, Ultra 2 and Ultra 30, Ultra 60	19
Ultra Enterprise	19
<i>Kernel main() and the /etc/system file</i>	19
<i>Kernel Threads</i>	20
<i>File Systems</i>	22

Boot Block	22
The Structure of UFS File System Cylinder Groups	22
The Boot Block	22
Create the boot blocks	22
The Superblock	23
Inodes	23
Data Blocks	24
Free Blocks	24
<i>Exercise</i>	25
<i>Run Levels</i>	26
How to Determine a System's Run Level	27
The /etc/inittab File	27
Example-Default inittab File	28
What Happens When the System Is Brought to Run Level 3	28
<i>Run Control (rc) Info</i>	29
Using a Run Control Script to Stop or Start Services	29
How to Use a Run Control Script to Stop or Start a Service	30
Example-Using a Run Control Script to Stop or Start a Service	30
Adding a Run Control Script	30
Disabling a Run Control Script	31
Run Control Script Summary	31
<i>Exercise</i>	33
<i>Conclusion</i>	33

Kernel Bootstrap and Initialization

Overview

QUESTION: How does the Solaris kernel differ from mach or bsd? How does the Solaris kernel differ from Linux? Do we want to discuss kernel architecture basics?

The binary object modules that make up the Solaris kernel exist in a well-defined directory name in the root and usr filesystems. The core modules that make up the minimum required for a usable system are split across multiple directories. All kernel modules will be in one of three possible categories that determine their placement in the directory hierarchy: platform and hardware independent, platform dependent, or hardware class dependent. The /kernel directory tree contains operating system modules that are platform independent, including the genunix executable object that is loaded initially at boot time.

The two remaining directory trees are categorized as hardware-class dependent and platform dependent. The hardware-class is derived from the process type, in conjunction with the processor-to-memory bus architecture. For example, when the SparcStation/Server 1, 1+, 2, and the variants such as the IPC were introduced, they were dependent on the sun4c architecture. Later on, with the introduction of the Mbus (or XDBus on larger system systems) bus architecture, the combination of process and bus architectures was referred to as sun4m. The sun4m class of machines would include the SparcStation 5, 10, 20 and their variants. With the advent of the UltraSparc-I and the Ultra Port Architecture (UPA), a new architecture, called sun4u, was introduced. For people coming from the land of Intel architectures, this is often a strange concept to comprehend – the architecture is based on a combination of “motherboard”, “bus”, and “processor”, and not just the processor itself. It is often very confusing to the system administrator who has to comprehend that the HyperSparc, SuperSparc, and SuperSparc-II processors are all part of the sun4m architecture.

The platform category is the specific machine type: the type of system, preceded by a SUNW for Sun-based platforms. Examples include, SUNW,Ultra-2, and SUNW,Ultra Enterprise-10000. Kernel modules that deal specifically with “box” dependencies, such as environmental controls and interconnect-specific support, reside in the directory tree beginning with /platform/<platform-name>. More specifics related to the processor and bus types are included in a following section.

Other loadable kernel modules are located under the /usr directory tree, which also contains a kernel (/usr/kernel) and platform (/usr/platform) set of subdirectories. The modules under these directories are kernel modules that are loaded as needed: either on demand, as a result of a user or application process requesting a service that requires the loading of one of the modules, or manually with the modload(1M) command. For example, the kernel exec support for each executing code written in the Java programming language is in /usr/kernel/exec and has the filename of javaexec. This module is loaded by the kernel the first time a Java program is started on the system. Note that you can instruct the system to load optional modules at boot time by using the forcload directive in the /etc/system kernel configuration file.

Bootstrap and Initialization

The boot process loads the binary kernel objects that make up the Solaris kernel from a bootable system disk into physical memory, initialize that kernel structures, sets system tuneable parameters, starts system

processes (daemons), and brings the system to a known, usable state. The basic steps involved in booting can be summarized as follows:

- The boot command reads and loads the bootblock into memory.
- The bootblock locates and loads the secondary boot program, ufsboot, into memory and passes control to it.
- Ufsboot locates and loads the core kernel images and the required kernel runtime linker into memory and passes control to the loaded kernel.
- The core kernel locates and loads mandatory kernel modules from the root disk directory tree and executes the main startup code.
- The kernel startup code executes, creating and initializing kernel structures, resources, and software components.
- The system executes shell scripts from the system directories, bringing that system up to the init state specified in /etc/inittab.

The preliminary steps shown above are based on a local disk boot. A network boot uses a different secondary boot program called inetboot.

The initial stage of the boot process requires support from system firmware, typically loaded into a PROM, such that a bootable device can be addressed and that some minimal I/O functionality exists for reading of a primary bootblock.

Sun SPARC-based systems implement such firmware, known as the OpenBoot PROM (OBP), in the Forth programming language. OpenBoot is a standard for Boot Firmware as defined by IEEE Standard 1275-1994. OpenBoot, immediately after your system is power on, performs the following basic system tasks:

- Provide a Power-On, Self Test (POST)
- Test and initialize the system hardware
- Determine the hardware configuration
- Boot the operating system
- Provide interactive debugging facilities for testing hardware and software

Although OpenBoot was first implemented by Sun Microsystems on SPARC systems, its design is processor-independent. One might want to perceive the functionality given by the OBP as the same type of functionality given by the BIOS within Intel architectures – yet with more power and control.

On a system that has completed its POST tests, the boot process is initiated either with an explicit boot(1M) command or automatically if the auto-boot parameter (stored within the NVRAM) is set to true. The boot(1M) command supports several options, including specifying a boot device and bootfile on the command line, flagging a boot into the kernel debugger (kadb), or running the boot in a more verbose mode for debugging problems.

On SPARC systems using the default UNIX file system (UFS) for the root system, the bootblock is located on physical disk sectors 1 thru 15. This can be done using the installboot(1M) command. The bootblock code is maintained in the /usr/platform/<arch>/lib/fs/ufs/bootblk file. The utility dd(1) is used to copy the bootblock to the specified disk partition. When alternate file systems, such as Veritas, are used to encapsulate the root disk, alternate methods are used.

Once ufsboot is loaded, the runtime linker, krtld, is also loaded. The linker is loaded based on information provided in the ELF headers of the kernel. Control is then passed onto krtld, which notes dependencies on additional binary object files.

With all the core binary objects (unix, krtld, genunix, platmod, and the CPU module) loaded into memory and linked, krtld passes control to unix, which is loaded near the top of physical memory space, to take over execution. The Solaris kernel is now running, the Memory Management Unix (MMU) is enabled and

the first 16 megabytes of memory are mapped. At this point in time, system devices are ready to be mapped.

Device Trees

A device tree is a data structure describing the devices attached to a system.

Devices are attached to a host computer through a hierarchy of interconnected buses. OpenBoot represents the interconnected buses and their devices as a tree of nodes. A node representing the host computer's main physical address bus forms the tree's root node.

Nodes with children usually represent buses and their associated controllers, if any. Each such node defines a physical address space that distinguishes the devices connected to the node from one another. Each child of that node is assigned a physical address in the parent's address space.

The physical address generally represents a physical characteristic unique to the device (such as the bus address or the slot number where the device is installed). The use of physical addresses to identify devices prevents device addresses from changing when other devices are installed or removed.

Nodes without children are called leaf nodes and generally represent devices. However, some such nodes represent system-supplied firmware services. Both the user and the system can operating system can determine the hardware configuration of the system by inspecting the device tree

Device Path Names, Addresses, and Arguments

OpenBoot deals directly with hardware devices in the system. Each device has a unique name representing the type of device and where that device is located in the system addressing structure. The following example shows a full device path name:

```
/sbus@1f,0/SUNW,fas@e,8800000/sd@3,0:a
```

A full device path name is a series of node names separated by slashes (/). The root of the tree is the machine node, which is not named explicitly but is indicated by a leading slash (/). Each node name has the form:

```
driver-name@unit-address:device-arguments
```

Device Alias

A device alias, or simply, **alias**, is a shorthand representation of a device path.

For example, the alias `disk` may represent the complete device path name:

```
/sbus@1f,0/esp@0,40000/sd@3,0:a
```

Systems usually have predefined device aliases for the most commonly used devices, so you rarely need to type a full device path name.

Table 1 - Examining and Creating Device Aliases

Command	Description
<code>devalias</code>	Display all current device aliases
<code>devalias alias</code>	Display the device path name corresponding to

	alias.
devalias <i>aliasdevice-path</i>	Define an alias representing device-path. If an alias with the same name already exists, the new value supersedes the old.

User-defined aliases are lost after a system reset or power cycle. If you want to create permanent aliases, you can either manually store the devalias command in a portion of non-volatile RAM (NVRAM) called nvramrc, or use the nvalias and nvunalias commands. (For more details, see the Sun documentation.)

Displaying the Device Tree

You can browse the device tree to examine and modify individual device tree nodes. The device tree browsing commands are similar to the Solaris™ (TM) commands for changing, displaying and listing the current directory in the Solaris directory tree. Selecting a device node makes it the current node.

Table 2 - Commands for Browsing the Device Tree

Command	Description
.properties	Display the names and values of the current node's properties.
dev <i>device-path</i>	Choose the indicated device node, making it the current node.
dev <i>node-name</i>	Search for a node with the given name in the subtree below the current node, and choose the first such node found.
dev ..	Choose the device node that is the parent of the current node.
dev /	Choose the root machine node.
device-end	Leave the device tree.
" <i>device-path</i> " find-device	Choose device node; similar to dev.
Ls	Display the names of the current node's children.
Pwd	Display the device path name that names the current node.
see <i>wordname</i>	Decompile the specified word.
Show-devs [<i>device-path</i>]	Display all the devices directly beneath the specified device in the device tree. show-devs used by itself shows the entire device tree.
Words	Display the names of the current node's methods.
" <i>device-path</i> " select-dev	Select the specified device and make it the active node.

Booting and Testing Your System

This chapter describes the most common tasks that you perform using OpenBoot. These tasks enable you to:

- Boot your system
- Run diagnostics
- Display system information
- Reset the system

Booting Your System

The most important function of OpenBoot firmware is to boot the system. Booting is the process of loading and executing a stand-alone program such as an operating system. Booting can either be initiated automatically or by typing a command at the User Interface. *It must be noted that certain commands actually hang the sun4u class of machines.*

The boot process is controlled by a number of configuration variables. (Configuration variables are discussed in detail in Chapter 3, Setting Configuration Variables) The configuration variables that affect the boot process are:

- auto-boot?

This variable controls whether or not the system automatically boots after a system reset or when the power is turned on. This variable is typically true.

- boot-command

This variable specifies the command to be executed when auto-boot? is true. The default value of boot-command is boot with no command line arguments.

- diag-switch?

If the value is true, run in the Diagnostic mode. This variable is false by default.

- boot-device

This variable contains the name of the default boot device that is used when OpenBoot is not in diagnostic mode.

- boot-file

This variable contains the default boot arguments that are used when OpenBoot is not in diagnostic mode.

- diag-device

This variable contains the name of the default diagnostic mode boot device.

- diag-file

This variable contains the default diagnostic mode boot arguments.

Booting for the Expert User

Booting is the process of loading and executing a client program. The client program is normally an operating system or an operating system's loader program, but boot can also be used to load and execute other kinds of programs, such as diagnostics. (For more details about loading programs other than the operating system, see Chapter 5, Loading and Executing Programs").

Booting usually happens automatically based on the values contained in the configuration variables described above. However, the user can also initiate booting from the User Interface.

OpenBoot performs the following steps during the boot process:

- The firmware may reset the machine if a client program has been executed since the last reset. (The execution of a reset is implementation dependent.)
- A device is selected by parsing the boot command line to determine the boot device and the boot arguments to use. Depending on the form of the boot command, the boot device and/or argument values may be taken from configuration variables.
- The bootpath and bootargs properties in the /chosen node of the device tree are set with the selected values.
- The selected program is loaded into memory using a protocol that depends on the type of the selected device. For example, a disk boot might read a fixed number of blocks from the beginning of the disk, while a tape boot might read a particular tape file.
- The loaded program is executed. The behavior of the program may be further controlled by the argument string (if any) that was either contained in the selected configuration variable or was passed to the boot command on the command line.

Often, the program loaded and executed by the boot process is a secondary boot program whose purpose is to load yet another program. This secondary boot program may use a protocol different from that used by OpenBoot to load the secondary boot program. For example, OpenBoot might use the Trivial File Transfer Protocol (TFTP) to load the secondary boot program, while the secondary boot program might then use the Network File System (NFS) protocol to load the operating system.

Typical secondary boot programs accept arguments of the form:

filename -flags

where **filename** is the name of the file containing the operating system and **-flags** is a list of options controlling the details of the start-up phase of either the secondary boot program, the operating system or both. Please note that, as shown in the boot command template immediately below, OpenBoot treats all

such text as a single, opaque argument string that has no special meaning to OpenBoot itself; the arguments string is passed unaltered to the specified program.

The boot command has the following format:

```
ok boot [device-specifier] [arguments]
```

Table 3 - Optional boot Command Parameters

Parameter	Description
[<i>device-specifier</i>]	<p>The name (full path name or devalias) of the boot device. Typical values include:</p> <p>cdrom (CD-ROM drive) disk (hard disk) floppy (3-1/2" diskette drive) net (Ethernet) tape (SCSI tape)</p> <p>If device-specifier is not specified and if diagnostic-mode? returns false, boot uses the device specified by the boot-device configuration variable.</p> <p>If device-specifier is not specified and if diagnostic-mode? returns true, boot uses the device specified by the diag-device configuration variable.</p>
[<i>arguments</i>]	<p>The name of the program to be booted (e.g. stand/diag) and any program arguments.</p> <p>If arguments are not specified and if diagnostic-mode? returns false, boot uses the file specified by the boot-file configuration variable.</p> <p>If arguments are not specified and if diagnostic-mode? returns true, boot uses the file specified by the diag-file configuration variable.</p>

Since a device alias cannot be syntactically distinguished from the arguments, OpenBoot resolves this ambiguity as follows:

- If the space-delimited word following **boot** on the command line begins with /, the word is a device-path and, thus, a device-specifier. Any text to the right of this device-specifier is included in arguments.
- Otherwise, if the space-delimited word matches an existing device alias, the word is a device-specifier. Any text to the right of this device-specifier is included in arguments.
- Otherwise, the appropriate default boot device is used, and any text to the right of **boot** is included in arguments.

Consequently, **boot** command lines have the following possible forms.

- ok boot

With this form, **boot** loads and executes the program specified by the default boot arguments from the default boot device.

- ok boot device-specifier

If **boot** has a single argument that either begins with the character / or is the name of a defined devalias, **boot** uses the argument as a device specifier. **boot** loads and executes the program specified by the default boot arguments from the specified device.

For example, to explicitly boot from the primary disk, type:

- ok boot disk

To explicitly boot from the primary network device, type:

- ok boot net

If **boot** has a single argument that neither begins with the character / nor is the name of a defined devalias, **boot** uses all of the remaining text as its arguments.

- ok boot arguments

boot loads and executes the program specified by the arguments from the default boot device.

- ok boot device-specifier arguments

If there are at least two space-delimited arguments, and if the first such argument begins with the character / or if it is the name of a defined devalias, **boot** uses the first argument as a device specifier and uses all of the remaining text as its arguments. **boot** loads and executes the program specified by the arguments from the specified device.

For all of the above cases, **boot** records the device that it uses in the bootpath property of the /chosen node. **boot** also records the arguments that it uses in the bootargs property of the /chosen node.

Device alias definitions vary from system to system. Use the **devalias** command, described in Chapter 1, Overview ", to obtain the definitions of your system's aliases.

Running Diagnostics

Several diagnostic routines are available from the User Interface. These on-board tests let you check devices such as the network controller, the floppy disk system, memory, installed SBus cards and SCSI devices, and the system clock.

The value returned by `diagnostic-mode?` controls:

- The selection of the device and file that are used by the boot and load commands (if the device and file are not explicitly specified as arguments to those commands).
- The extent of the diagnostics performed during power-on self-test, and the (implementation dependent) number of diagnostic messages produced.

OpenBoot will be in diagnostic mode and the `diagnostic-mode?` command will return true if at least one of the following conditions is met:

- The configuration variable `diag-switch?` is set to true.
- The machine's diagnostic switch (if any) is "on".
- Another system-dependent indicator requests extensive diagnostics.

When OpenBoot is in the Diagnostic mode, the value of `diag-device` is used as the default boot device and the value of `diag-file` is used as the default boot arguments for the boot command.

When OpenBoot is not in the Diagnostic mode, the value of `boot-device` is used as the default boot device and the value of `boot-file` is used as the default boot arguments for the boot command.

Testing the SCSI Bus

To check a SCSI bus for connected devices, type:

```
ok probe-scsi
```

```
Target 1
```

```
Unit 0 Disk SEAGATE ST1480 SUN04246266 Copyright (C) 1991 Seagate All rights reserved
```

```
Target 3
```

```
Unit 0 Disk SEAGATE ST1480 SUN04245826 Copyright (C) 1991 Seagate All rights reserved
```

```
ok
```

The actual response depends on the devices on the SCSI bus.

Testing Installed Devices

To test a single installed device, type:

```
ok test device-specifier
```

In general, if no message is displayed, the test succeeded.

Note -

Many devices require the system's `diag-switch?` parameter to be true in order to run this test.

Testing the Diskette Drive

The diskette drive test determines whether or not the diskette drive is functioning properly. For some implementations, a formatted, high-density (HD) disk must be in the diskette drive for this test to succeed.

To test the diskette drive, type:

```
ok test floppy
Testing floppy disk system. A formatted
disk should be in the drive.
Test succeeded.
ok
```

Note:

Not all OpenBoot systems include this test word.

To eject the diskette from the diskette drive of a system capable of software-controlled ejection, type:

```
ok eject-floppy
ok
```

Testing Memory

To test memory, type:

```
ok test /memory
Testing 16 megs of memory at addr 4000000 11
ok
```

Note:

Not all OpenBoot systems include this test word.

In the preceding example, the first number (4000000) is the base address of the testing, and the following number (11) is the number of megabytes to go.

Testing the Clock

(Note: Not all OpenBoot systems include this test word.)

To test the clock function, type:

ok **watch-clock**

Watching the 'seconds' register of the real time clock chip.

It should be ticking once a second.

Type any key to stop.

1
ok

The system responds by incrementing a number once a second. Press any key to stop the test.

Testing the Network Controller

To test the primary network controller, type:

ok **test net**

Internal Loopback test - (result)

External Loopback test - (result)

ok

The system responds with a message indicating the result of the test.

Note:

Depending on the particular network controller and the type of network to which your system is attached, various levels of testing are possible. Some such tests may require that the network interface be connected to the network.

Monitoring the Network

(Note: Not all OpenBoot systems include this test word.)

To monitor a network connection, type:

ok **watch-net**

Internal Loopback test - succeeded

External Loopback test - succeeded

Looking for Ethernet packets.

'.' is a good packet. 'X' is a bad packet.

Type any key to stop

.....X.....X.....

ok

The system monitors network traffic, displaying "." each time it receives an error-free packet and "X" each time it receives a packet with an error that can be detected by the network hardware interface.

Displaying System Information

The User Interface provides one or more commands to display system information. **banner** is provided by all OpenBoot implementations; the remaining commands represent extensions provided by some implementations. These commands, listed in Table 8, let you display the system banner, the Ethernet address for the Ethernet controller, the contents of the ID PROM, and the version number of OpenBoot. (The ID PROM contains information specific to each individual machine, including the serial number, date of manufacture, and Ethernet address assigned to the machine.)

Table 4 - System Information Commands

Command	Description
banner	Display power-on-banner
show-sbus	Display list of installed and probed Sbus devices
.enet-address	Display current Ethernet address
.idprom	Display ID PROM contents, formatted.
.traps	Display a list of processor-dependent trap types.
.version	Display version and date of the boot PROM
.speed	Display processor and bus speed

Resetting the System

Occasionally, you may need to reset your system. The **reset-all** command resets the entire system and is similar to performing a power cycle.

To reset the system, type:

ok **reset-all**

If your system is set up to run the power-on self-test (POST) and initialization procedures on reset, these procedures begin executing once you initiate this command. (On some systems, POST is only executed after power-on.) Once POST completes, the system either boots automatically or enters the User Interface, just as it would have done after a power cycle.

Setting Configuration Variables

This chapter describes how to access and modify non-volatile RAM (NVRAM) configuration variables.

System configuration variables are stored in the system NVRAM. These variables determine the start-up machine configuration and related communication characteristics. You can modify the values of the configuration variables, and any changes you make remain in effect even after a power cycle. Configuration variables should be adjusted cautiously.

The procedures described in this chapter assume that the user interface is active. See Chapter 1, Overview for information about entering the user interface.

Displaying and Changing Variable Settings

Command	Description
Printenv	Display current variables and current default values
Setenv <i>variable value</i>	Set <i>variable</i> to the given decimal or text <i>value</i>
set-default <i>variable</i>	Reset the value of <i>variable</i> to the factory default
set-defaults	Reset variable values to the factory defaults
STOP-N	Reset to factory defaults [unsupported]
Password	Set security-password

Input and Output Control

- **input-device**
- **output-device**
- screen-#columns
- screen-#rows

A list of all the commands that can be executed in the OpenBoot can be obtained with:

ok words

Sun Processor Architecture

We have discussed many things related to the various types of processors and busses that Sun Microsystems has been involved in over the past few generations of microcomputing. We will now discuss in more detail some of the differences that these various processors have.

Early microprocessor-based servers, even relatively large departmental systems, were simple in an architectural sense. The SPARCserver 390 (announced in 1989) used an architecture derived from then-current technical workstations. The concept of actually splitting up the types of servers between

workstation and server models was only a recent phenomenon. For example, the server version of the SparcStation 20 was merely a SparcStation 20 with the nameplate changed to be SparcServer 20 and without the additional video capabilities.

Many different SPARC processor designs are available now, each designed to meet different goals. The following discussion highlights the most significant features of each from a configuration perspective.

The SuperSPARC Family

Until the advent of Ultra computing, one of the more common SPARC systems was based on the SuperSPARC and its derivatives. There are three slightly different SuperSPARC implementations, all designed by Sun engineers and fabricated in Texas Instrument's BiCMOS process. The original chip was first delivered in 1992 in 33-, 36-, and 40 MHz versions, followed by the SuperSPARC+, which arrived in 1993 at 50-, and 60 MHz. The final member of the family is the SuperSPARC-II, offered in 75- and 85 MHz models.

A member of the first generation of superscalar RISC processors, the SuperSPARC was designed to attain relatively low clock rate, but also accomplish a great deal of work in each clock cycle. Accordingly, it can dispatch up to four separate instructions in each clock cycle: two integer instructions, one floating point instruction, and a branch. When executing out of cache, the processor is able to sustain an execution rate of about 1.3 instructions per clock, although memory delays mean that realistic execution is on the order of 0.85 instructions per clock. Although floating point performance is anemic, integer performance is good, and the chip is amount the most efficient on a SPECint92 per clock basis.

Designed to operate in both desktops and servers, the SuperSPARC design actually has two discrete external interfaces, one for Mbus and one for an external cache controller. In Mbus mode, the processor can be used without other logic to implement multiprocessor systems. The processor adapts to the Mbus clock; if a SPARCstation 20's Model 50 module were plugged into a SPARCstation 10, it would operate at the latter's 40 MHz clock speed. The processor can also be mated with a SuperCache external cache controller and an SRAM cache array; this arrangement is used in all other SuperSPARC-based systems. The SuperCACHE also has two different interfaces, one for Mbus found in desktop systems and one for Xbus for servers. The SuperCache senses the bus type and acts accordingly, permitting most modules to operate correctly in either type of system.

HyperSPARC

Similar in some respects to the SuperSPARC, the hyperSPARC processor was designed independently by engineers at Ross Technologies at approximately the same time that the SuperSPARC was being designed by Sun. This chip is also a first-generation superscalar design. Sun did not participate in the design; instead, the Ross team chose to build a fully compatible competing chip to the SuperSPARC using the open and published Mbus and SPARC V8 interfaces. The resulting processor makes very different design tradeoffs while pursuing the same overall goal. The same Mbus interface is supported, and the two processors implement the same SPARC V8 instruction set with the same semantics, but many different implementation details are different. Unlike SuperSPARC, the hyperSPARC design does not include an XDBus interface, configuring these processors in XDBus systems is impossible.

Most fundamentally, the hyperSPARC is a much more conservative architectural design than SuperSPARC, resulting in a processor that accomplishes much less each clock cycle, but which can attain far higher clock rates. Whereas the SuperSPARC can dispatch up to four instructions per clock, hyperSPARC can dispatch only one integer and one floating point instruction per clock. This design simplifies the internal organization and is one of the primary reasons hyperSPARC clock rates approach three times those of the SuperSPARC. Current hyperSPARC processors operate at clock rates greater than

200 MHz, which the fastest SuperSPARC runs at just 85 MHz. In order to more easily achieve the higher clock rates, Ross opted for a full custom CMOS process technology. CMOS is a much better-understood technology than BiCMOS for design houses without resources available to Intel. Because of this, hyperSPARC chips were being designed years after the end of life for SuperSPARC processors was reached – thus creating the capability of having multiple 200+ MHz processors available to give the UltraSPARC chips a run for their money in the late 1990's.

MicroSPARC and MicroSPARC-II

The lowest-cost parts in the SPARC family, the MicroSPARC architecture is designed almost exclusively to accommodate workstation requirements. Instead of a multiprocessor implementation consisting of several chips, the MicroSPARC and MicroSPARC-II systems consists essentially of one main chip. Desktop SuperSPARC and hyperSPARC systems have several large ASICs implementing Mbus, Sbus, and memory control. The MicroSPARC chip itself contains all of the CPU functions (CPU, cache, and memory management unit), plus Sbus and a memory controller.

The MicroSPARC design has, by quite a margin, the lowest (best) memory latency of its generation of SPARC processors. This is a necessity: the MicroSPARC-1 has only 6 KB of cache, and MicroSPARC-II has 24 KB. Because the caches are very small, latency to memory is at a premium, and the architecture makes allowances for these realities. In addition, the MicroSPARC is made vastly simpler by not having to make allowances for multiprocessing. Finally, the fact that MicroSPARC is a single integrated chip, with few high-performance off-chip interconnects means that it simply takes less time to get from one functional unit to another. This contributes both to excellent memory latency and to very tight control of the Sbus. Boards that are extremely intolerant of Sbus or Sbus interrupt latencies work very well in the MicroSPARC architecture.

Unfortunately, except for workgroup servers, the MicroSPARC-II does not have the horsepower to run most applications, and the original MicroSPARC is completely inadequate. Even with low-latency memory, the processor quickly saturates under the heavy memory traffic generated by these applications.

UltraSPARC

Using the experience gained from the first generation of superscalar processors, Sun design the UltraSPARC to deliver leading-edge performance in many dimensions. Although the SuperSPARC family succeeded in delivering adequate performance for most applications, it suffered from a complex design, relatively slow access to memory, contention for a TLB shared between user and system contexts, and moderate ability to copy memory. UltraSPARC uses a relatively conservative full-custom CMOS design (for a section generation RISC processor), to implement a 64-bit, four way superscalar design. The new design features a much cleaner system interface, a limited but highly optimized implementation-specific instruction set extension designed to accelerate graphics and server applications, a much faster memory interface – and clock rates ranging from 143 MHz to 250 MHz during its first generation. The UltraSPARC-1 implements a proper superset of the SPARC version 9 software architecture.

There are five UltraSPARC-I and UltraSPARC-II system implementations to discuss. The most recent and simplest is the highly integrated uniprocessor UltraSPARC Iii, a successor to the microSPARC. The original UltraSPARC workstations are based on a simple UPA (Ultra Port Architecture) configuration. An expanded UPA configuration is used in the workgroup server systems. The Enterprise Server range uses a Gigaplane bus to link large numbers of CPUs and I/O buses. The E10000 Starfire uses an active backplane crossbar to support the very biggest configurations.

The Ultra 5 and Ultra 10 (Darwin Series)

The UltraSPARC Iii provides a highly integrated design for building low-cost systems in the same way as its microSPARC predecessor. The CPU chip includes a complete memory controller, external cache controller, I/O bus controller (32-bit PCIbus this time) and a UPA slave interface for Creator graphics options. These servers are meant to be used as workstations.

The Ultra 1, Ultra 2 and Ultra 30, Ultra 60

These are the first generations of the Ultra SPARC chip and are primarily meant to be used as higher end workstations or cheap, redundant servers. The Ultra 2 is one of the most common desktop of server machines you'll see in the field today.

Ultra Enterprise

When talking about the Enterprise series of servers, you're no longer just discussing the various chip types. These servers are labeled as Enterprise-ready because of the redundance Bus architecture utilizing a UPA switch to control all aspects of the host's processing power, memory, and graphics capabilities. As the servers get larger, more busses are included so that no resources are ever really shared. Higher-end machines use a Gigaplane interface for latency optimization.

Kernel main() and the /etc/system file

Once all the main work has been done, the kernel enters its main() function. This is where the code begins a sequence of function calls into the higher-level areas of the kernel to continue the initialization process. The main() function is part of the kernel hardware-independent code. It immediately calls into hardware-specific routines for specific initialization steps, then returns to the common code to complete the boot process. The platform-specific startup() function does some preliminary memory initialization: determining available physical memory after the core kernel is loaded, setting up of the kernel address space symbols, and allocating memory page structure. The kernel memory (KVM) subsystem and the kernel kstat (kernel statistics) framework are initialized. Once these steps are completed, the operating system banner is displayed on the console.

SunOS Release 5.7 Version Generic 64-bit [UNIX® System V Release 4.0]
Copyright © 1983-1998, Sun Microsystems, Inc.

Now you know why it the cursor on your screen moves around for such a long time before the operating system is loaded!

The kernel then creates a linked list of sysparam data structures, such that each entry on the /etc/system file has a corresponding data structure in the list. The sysparam structure is defined in /usr/include/sys/sysconf.h. The files in the sysparam structure are generically self-explanatory. Space is defined for parameter-specific information and the necessary pointer to maintain the linked list. Each line within the /etc/system file is walked through and variables are set.

You will notice that whenever anyone does any kernel-level tuning of your operating system, it is done mainly through this file. Parameters such as maxusers, and rlimits would be set through this file. The maxusers parameter is calculated on the basis of the amount of physical memory installed on the system, where maxusers equals the number of megabytes of RAM, to a maximum of 1024. If maxusers is set in

/etc/system, that value is used. Any value higher than 2048 is automatically reduced to 2048. Once maxusers is established, the startup code sets other kernel limits, based on maxusers.

Maxpid – Maximum process ID value. If pidmax is set in /etc/system, the pidmax value will be tested against a minimum reserved_procs value and a maximum value (30000). If pidmax is less than reserved_procs or greater than 30000, it is set to 30000. Otherwise, the user-defined value is set.

Max_nprocs – Maximum number of processes, systemwide, (set as $10 + 16 * \text{maxusers}$).

Maxuprc – Maximum processes per non-root user, set as $(\text{max_nprocs} - \text{reserved_procs})$. Reserved_procs has a default value of 5.

Ufs_ninode and ncsiz – High limits for the UFS inode cache and directory name cache. Both of these caches are set to the same value, $(4 * (\text{max_nprocs} + \text{maxusers}) + 320)$.

Ndquot – The number of UFS disk quota structures, systemwide, set as $((\text{maxusers} * 40) / 4) + \text{max_nprocs}$.

As you can see, maxusers is not really the maximum number of users the system will support. It is simply a parameter that can be set to drive the number of several configurable kernel resources that indirectly affect the load of volume of users the system can handle. Many things factor in to the load capabilities of the system, not the least of which are what constitutes a user, what a user does, and how a user connects to the system.

Other kernel parameters that are related to the STREAMS subsystem and miscellaneous areas of the system are also set at this time. These parameters use hard-coded default values that are not driven by the maxusers parameter unless a value has been set in /etc/system. These variables are nstrpush, strmsgsz, strctlsz, ngroups_max, rstchown, hz, hires_hz, hires_tick, autoup, and rlimits. Complete lists of kernel tunables are available all over the Internet.

Some of the ones that are most important would be those related to resource limits. For example, when rlim_fd_max and rlim_fd_cur are set within /etc/system, these control per-process resource limits for things such as file descriptors. Many system performance tuning guides require you to increase these limits.

It should also be noted that Solaris is a very dynamic environment and that as you upgrade between versions of Solaris, you should clear out the /etc/system file of most resources. Only those required by applications such as Oracle should remain. Others, however, might greatly decrease your system performance. For example, some limits were drastically increased between Solaris version 2.5 and Solaris 7 so an administrator upgrading from these versions of Solaris would have actually ended up decreasing important kernel parameters and hindered the host.

Kernel Threads

During the boot process, the system creates several kernel threads that are not visible outside the operating system; they cannot be seen by examination of the /proc directory hierarchy or through the ps(1) command. They show up on the systemwide linked list of kernel threads and behave as daemon threads, called by different areas of the operating system when appropriate. These threads are only part of the multithreaded design of the Solaris kernel, and they perform very specific functions. They all execute in the SYS scheduling class and are unique in that they are the only kernel threads in the system that do not have support for context information and a stack. The following is a list of the kernel threads created at boot time:

Thread_reaper – Cleans up exited (zombie) kernel threads placed on death row.

`Mod_uninstall_daemon` – For checkpoint/resume (CPR) support; unloads kernel modules when the system checkpoints.

`Hotplug_daemon` – Device hotplug support. Adds the `dev_info` node for an added device to the system device tree.

`Kmem_async_thread` – Garbage collector thread for the kernel slab allocator.

`Seg_pasync_thread` – Reclaims pagelocked pages that have not been referenced for a while.

`Ksyms_update_thread` - Updates the dynamic kernel symbol table, `/dev/ksyms`, as kernel modules are loaded.

`Callout_thread` – Processes the kernel callout queue by means of the lock interrupt handler.

`Cpu_pause` – Per-processor; for CPU state changes.

`Modload_thread` – Supports dynamic loading of kernel modules.

`Background` – STREAMS-based; services STREAMS queues. One of these threads is created for every CPU on a system at boot time.

`Freebs` – STREAMS-based; manages list of free STREAMS blocks. One of these threads is created for every CPU on a system at boot time.

`Qwriter_outer_thread` – STREAMS-based; processes out STREAMS syncq messages. One of these threads is created for every CPU on a system at boot time.

All kernel threads, even those listed above are part of the operating system, need a process context to execute, in order to have the required software context state (for example, an address space). Kernel threads created by the operating system are linked to PID 0 for contextual information.

Much of the kernel architecture is layered architecture and many of the kernel's major subsystems call into a common set of lower-level services. In some respects, the kernel architecture is similar to that of a client/server application, except that the interfaces in this case are, for the most part, private to the kernel. That is, many of the callable routines are not public interfaces available for use for general application software development. The other obvious distinction is that the users of the services, or clients, in this context, are other kernel subsystems as opposed to application code. Exceptions to the private interface generalization are those interfaces defined for device drivers and kernel STREAMS modules.

The kernel module loading facility serves as a good example of one such service. Several phases involved in dynamic module loading are loading the module into memory, establishing a kernel address space mapping for module segments, linking the module's segments into the kernel, and performing the module install function.

The loadable module types are defined in `/usr/include/sys/modctl.h`; they are device drivers, system calls, file systems, miscellaneous modules, streams modules, scheduling classes, and exec modules (exec support for executable objects). For each of these module types, some specific kernel installation steps are required to complete the loading process. The steps involved in loading a module into the kernel are similarly conceptually to when a dynamically linked program is started under Solaris. That is, shared objects to which a binary is linked are dynamically loaded into kernel memory and linked into the process's address space during execution.

The system loads the modules required for a functional system at boot time. A path variable that defines the module search path guides the boot code in locating the appropriate kernel objects. The Solaris kernel is dynamic in nature and kernel objects can be loaded and unloaded during the life of a running system. This is because the kernel's symbol table exists as a dynamic entity. It is maintained through a pseudodevice, `/dev/ksyms`, and its corresponding device driver, `/usr/kernel/drv/ksyms`. The kernel symbol table is updated by a kernel thread created specifically for that purpose – the kernel runtime linker issues a wakeup to the `ksyms_update_thread()` when a module is loaded (or unloaded), and the kernel symbol table is updated to reflect the current state of the kernel objects.

As we've seen, the dynamic loading of a kernel module is facilitated through two major kernel subsystems: the module management code and the kernel runtime linker. These kernel components make use of other kernel services, such as the kernel memory allocator, kernel locking primitives, and the kernel's `ksyms`

driver, taking advantage of the module design of the system and providing a good example of a layered model.

File Systems

Solaris provides a flexible framework that allows multiple and different file system types within the environment. The most common file system is the Unix file system (UFS), which is used by default to hold all of the files in the root directory hierarchy of the Unix file system. UFS is known as an ondisk or regular file system, which means that it uses a storage device to hold real file data. There are other ondisk file systems (such as VxFS) but the Solaris kernel also has support for various virtual file systems such as NFS.

The kernel has a default support for using UFS to hold its initial boot information. If, for example, we were to use Veritas' file system (VxFS) as the root disk (by encapsulating it), the kernel would need to be able to support VxFS as an official file system in order to understand its data and not look at it as a corrupt UFS object.

Boot Block

The Structure of UFS File System Cylinder Groups

When you create a UFS file system, the disk slice is divided into cylinder groups, which is made up of one or more consecutive disk cylinders. The cylinder groups are then further divided into addressable blocks to control and organize the structure of the files within the cylinder group. Each type of block has a specific function in the file system.

A UFS file system has these four types of blocks

Block Type	Stores
Boot block	Information used when booting the system
Superblock	Detailed information about the file system
Inode	All information about a file
Storage or data block	Data for each file

This section provides additional information about the organization and function of these blocks.

The Boot Block

The boot block stores the procedures used in booting the system. If a file system is not to be used for booting, the boot block is left blank. The boot block appears only in the first cylinder group (cylinder group 0), and is the first 8 Kbytes in a slice.

Create the boot blocks

```
# installboot /usr/platform/^uname-i/lib/fs/ufs/bootblk /dev/rdisk/devicename
```

The Superblock

The superblock stores much of the information about the file system. A few of the more important things it contains are:

- Size and status of the file system
- Label (file system name and volume name)
- Size of the file system logical block
- Date and time of the last update
- Cylinder group size
- Number of data blocks in a cylinder group
- Summary data block
- File system state: clean, stable, or active
- Path name of the last mount point

The superblock is located at the beginning of the disk slice, and is replicated in each cylinder group. Because the superblock contains critical data, multiple superblocks are made when the file system is created. Each of the superblock replicas is offset by a different amount from the beginning of its cylinder group. For multiple-platter disk drives, the offsets are calculated so that a superblock appears on each platter of the drive. That way, if the first platter is lost, an alternate superblock can always be retrieved. Except for the leading blocks in the first cylinder group, the leading blocks created by the offsets are used for data storage.

A summary information block is kept with the superblock. It is not replicated, but is grouped with the first superblock, usually in cylinder group 0. The summary block records changes that take place as the file system is used, and lists the number of inodes, directories, fragments, and storage blocks within the file system.

Inodes

An inode contains all the information about a file except its name, which is kept in a directory. An inode is 128 bytes. The inode information is kept in the cylinder information block, and contains:

- The type of the file, which is one of the following:
 - Regular
 - Directory
 - Block special
 - Character special
 - Symbolic link
 - FIFO, also known as named pipe
 - Socket

- The mode of the file (the set of read-write-execute permissions)
- The number of hard links to the file
- The user ID of the owner of the file
- The group ID to which the file belongs
- The number of bytes in the file
- An array of 15 disk-block addresses
- The date and time the file was last accessed
- The date and time the file was last modified
- The date and time the file was created.

The array of 15 disk addresses (0 to 14) point to the data blocks that store the contents of the file. The first 12 are direct addresses; that is, they point directly to the first 12 logical storage blocks of the contents of the file. If the file is larger than 12 logical blocks, the 13th address points to an indirect block, which contains direct block addresses instead of file contents. The 14th address points to a double indirect block, which contains addresses of indirect blocks. The 15th address is for triple indirect addresses, if they are ever needed.

Data Blocks

The rest of the space allocated to the file system is occupied by data blocks, also called storage blocks. The size of these data blocks is determined at the time a file system is created. Data blocks are allocated, by default, in two sizes: an 8-Kbyte logical block size, and a 1-Kbyte fragmentation size.

For a regular file, the data blocks contain the contents of the file. For a directory, the data blocks contain entries that give the inode number and the file name of the files in the directory.

Free Blocks

Blocks not currently being used as inodes, as indirect address blocks, or as storage blocks are marked as free in the cylinder group map. This map also keeps track of fragments to prevent fragmentation from degrading disk performance.

UFS Logging

Integrated UFS logging is another new feature, where a UFS filesystem can be mounted with logging enabled. (It's a mount(1M) option, e.g., "mount -o logging /dev/dsk/c0t0d0s0 /mnt".) UFS logging captures writes to the filesystem and logs changes to a log area in the filesystem before applying the changes to the filesystem metadata structures (inodes, superblock, etc.).

UFS logging is designed to eliminate filesystem corruption due to a crash by synchronously writing to the log area, and subsequently applying the changes to the filesystem. This significantly reduces the possibility that the filesystem could be left in an inconsistent state, requiring fsck(1M) to repair or (even worse) restore from a backup tape.

When a write transaction is sent to the filesystem that has been mounted with logging enabled, the changed data is written sequentially to the log, and then to a commit record. If a crash occurs, changes that haven't been committed are discarded, and committed transactions are rolled in the filesystem (much like logging in an RDBMS). The log scanning process is much faster than on fsck, so reboot times following crashes are greatly reduced. UFS logging also has the advantage of making synchronous writes to the filesystem faster, since all the updates are grouped together and written sequentially to the log, allowing the synchronous write to return faster. This holds true for directory updates as well.

The log itself is not on a separate partition, and is instead allocated from free blocks on the filesystem. If a separate partition is desired for logging, installing and configuring Sun Solstice DiskSuite is required. The log space required is about 1 megabyte (MB) of log for 1 GB of filesystem, with a maximum of 64 MB. The log is flushed regularly as it fills up. A filesystem unmount will also cause the log to be flushed.

Exercise

You have connected 2 external disks to the system and you have an internal disk in this system.

1. How do you determine the SCSI ID of all the disks? If they are not unique, set the SCSI ID to be unique.
2. How do you boot from one of the external disks, and also set this disk as the default boot disk?

You have a dual-headed system with 2 graphics cards on sbus slot 0 and 1. The default output device is pointing to the graphics card in slot 0.

1. How do you change the default output device to the graphics card in slot 1?
2. How do you create an alias for the graphics card in slot 1?

How do you boot a system from the network?

How do you reset all the NVRAM values to factory defaults?

For more in-depth documentation, visit: <http://docs.sun.com>

Run Levels

A system's run level (also known as an init state) defines what services and resources are available to users. A system can be in only one run level at a time.

The Solaris environment has eight run levels, which are described in the following table. The default run level is specified in the `/etc/inittab` file as run level 3.

Table 5 - Solaris Run Levels

Run Level	Init State	Type	Use this Level
0	Power-down state	Power-down	To shut down the operating system so that it is safe to turn off power to the system
S or s	Single-user state	Single-user	To run as a single user with all file systems mounted and accessible.
1	Administrative state	Single-user	To access all available file systems with user logins allowed.
2	Multiuser state	Multiuser	For normal operations. Multiple users can access the system and the entire file system. All daemons are running except for the NFS server daemons.
3	Multiuser state with NFS resources shared	Multiuser	For normal operations with NFS resource-sharing available.
4	Alternative multiuser state		This level is currently unavailable
5	Power-down state	Power-down	To shut down the operating system so that it is safe to turn off power to the system. If possible, automatically turn off power on systems that support this feature.
6	Reboot state	Reboot	To shut down the system to run level 0, and then reboot to multiuser state (or

			whatever level is the default in the inittab file).
--	--	--	---

How to Determine a System's Run Level

Use the `who -r` command to determine a system's current run level for any level except run level 0.

\$ `who -r`

Example-Determining a System's Run Level

\$ `who -r`

```
.    run-level 3 Sep 1 14:45  3  0 S
$
```

run-level 3	Identifies the current run level
Sep 1 14:45	Identifies the date of last run level change.
3	Is the current run level.
0	Identifies the number of times at this run level since the last reboot
S	Identifies the previous run level.

The System run level can be altered using the `init` command

The /etc/inittab File

When you boot the system or change run levels with the `init` or `shutdown` command, the `init` daemon starts processes by reading information from the `/etc/inittab` file. This file defines three important items for the `init` process:

- The system's default run level
- What processes to start, monitor, and restart if they terminate
- What actions to be taken when the system enters a new run level

Each entry in the `/etc/inittab` file has the following fields:

`id:rstate:action:process`

Table 6 - Fields in the inittab File

Field	Description
id	A unique identifier for the entry.
rstate	A list of run levels to which this entry applies.
action	How the process specified in the process field is to be run. Possible values include: <code>initdefault</code> , <code>sysinit</code> , <code>boot</code> , <code>bootwait</code> , <code>wait</code> , and <code>respawn</code> .
process	The command to execute.

Example-Default inittab File

The following example shows an annotated default inittab file:

```

1 ap::sysinit:/sbin/autopush -f /etc/iu.ap
2 ap::sysinit:/sbin/soconfig -f /etc/sock2path
3 fs::sysinit:/sbin/rcS sysinit >/dev/msglog 2<>/dev/msglog </dev/console
4 is:3:initdefault:
5 p3:s1234:powerfail:/usr/sbin/shutdown -y -i5 -g0 >/dev/msglog 2<>/dev/...
6 sS:s:wait:/sbin/rcS >/dev/msglog 2<>/dev/msglog </dev/console
7 s0:0:wait:/sbin/rc0 >/dev/msglog 2<>/dev/msglog </dev/console
8 s1:1:respawn:/sbin/rc1 >/dev/msglog 2<>/dev/msglog </dev/console
9 s2:23:wait:/sbin/rc2 >/dev/msglog 2<>/dev/msglog </dev/console
10 s3:3:wait:/sbin/rc3 >/dev/msglog 2<>/dev/msglog </dev/console
11 s5:5:wait:/sbin/rc5 >/dev/msglog 2<>/dev/msglog </dev/console
12 s6:6:wait:/sbin/rc6 >/dev/msglog 2<>/dev/msglog </dev/console
13 fw:0:wait:/sbin/uadmin 2 0 >/dev/msglog 2<>/dev/msglog </dev/console
14 of:5:wait:/sbin/uadmin 2 6 >/dev/msglog 2<>/dev/msglog </dev/console
15 rb:6:wait:/sbin/uadmin 2 1 >/dev/msglog 2<>/dev/msglog </dev/console
16 sc:234:respawn:/usr/lib/saf/sac -t 300
17 co:234:respawn:/usr/lib/saf/ttymon -g -h -p ""uname -n` console login: "
-T terminal-type -d /dev/console -l console -m ldterm,ttcompat

```

What Happens When the System Is Brought to Run Level 3

- The init process is started and reads the /etc/default/init file to set any environment variables. By default, only the TIMEZONE variable is set.
- **Then init reads the inittab file to do the following:**
 - Identify the **initdefault** entry, which defines the default run level (3).
 - Execute any process entries that have **sysinit** in the action field, so that any special initializations can take place before users login.
- Execute any process entries that have 3 in the rstate field, which matches the default run level, 3.

Table 7 - Run Level 3 Action Key Word Descriptions

Key Word	Starts the Specified Process
Powerfail	Only when the system receives a power fail signal
Wait	And waits for its termination
Respawn	If it does not exist. If the process already exists, continue scanning the inittab file.

Table 8 - Run Level 3 Command Descriptions

Command or Script Name	Description
/usr/sbin/shutdown	Shuts down the system. The init process runs the shutdown command only if the system has received a powerfail signal.
/sbin/rcS	Mounts and checks root (/), /usr, /var, and /var/adm

	file systems.
/sbin/rc2	Starts the standard system processes, bringing the system up into run level 2 (multiuser mode).
/sbin/rc3	Starts NFS resource sharing for run level 3.
/usr/lib/saf/sac -t 30	Starts the port monitors and network access for UUCP. This process is restarted if it fails.
/usr/lib/saf/ttymon -g -h -p "`uname -n` console login: " -T terminal_type -d /dev/console -l console	Starts the ttymon process that monitors the console for login Requests. This process is restarted if it fails. The terminal_type on a SPARC based system is sun. The terminal_type on an IA based system is AT386

Run Control (rc) Info

The Solaris software environment provides a detailed series of run control (rc) scripts to control run level changes. Each run level has an associated rc script located in the /sbin directory:

rc0, rc1, rc2, rc3, rc5, rc6, rcS

For each rc script in the /sbin directory, there is a corresponding directory named /etc/rcn.d that contains scripts to perform various actions for that run level. For example, /etc/rc2.d contains files used to start and stop processes for run level 2.

The /etc/rcn.d scripts are always run in ASCII sort order. The scripts have names of the form:
[KS][0-9][0-9]*

Files beginning with K are run to terminate (kill) a system process. Files beginning with S are run to start a system process.

Run control scripts are also located in the /etc/init.d directory. These files are linked to corresponding run control scripts in the /etc/rcn.d directories.

Using a Run Control Script to Stop or Start Services

One advantage of having individual scripts for each run level is that you can run scripts in the /etc/init.d directory individually to turn off functionality without changing a system's run level.

How to Use a Run Control Script to Stop or Start a Service

- Become superuser.
- Turn off functionality.

```
# /etc/init.d/filename stop
```

- Restart functionality.

```
# /etc/init.d/filename start
```

- Use the `pgrep` command to verify whether the service has been stopped or started.
- `# pgrep -f service`

Example-Using a Run Control Script to Stop or Start a Service

Turn off NFS server functionality by typing:

```
# /etc/init.d/nfs.server stop
# pgrep -f nfs
#
```

Restart the NFS services by typing:

```
# /etc/init.d/nfs.server start
# pgrep -f nfs
141
143
245
247
# pgrep -f nfs -d | xargs ps -fp
daemon 141 1 40 Jul 31 ? 0:00 /usr/lib/nfs/statd
root 143 1 80 Jul 31 ? 0:01 /usr/lib/nfs/lockd
root 245 1 34 Jul 31 ? 0:00 /usr/lib/nfs/nfsd -a 16
root 247 1 80 Jul 31 ? 0:02 /usr/lib/nfs/mountd
```

Adding a Run Control Script

If you want to add a run control script to start and stop a service, copy the script into the `/etc/init.d` directory and create links in the `rcn.d` directory you want the service to start and stop.

See the README file in each `/etc/rcn.d` directory for more information on naming run control scripts. The procedure below describes how to add a run control script.

How to Add a Run Control Script

1. Become superuser.
2. Add the script to the /etc/init.d directory.

```
# cp filename /etc/init.d
# chmod 0744 /etc/init.d/filename
# chown root:sys /etc/init.d/filename
```

3. Create links to the appropriate rcn.d directory.

```
# cd /etc/init.d
# ln filename /etc/rc2.d/Snnfilename
# ln filename /etc/rcn.d/Knnfilename
```

4. Use the ls command to verify that the script has links in the specified directories.

```
# ls /etc/init.d/ /etc/rc2.d/ /etc/rcn.d/
```

It is in good practice to actually make sure that the new startup script exist in the S99 hierarchy and adhere to all standard startup parameters. That is, having a new script exist somewhere in the middle of all the standard scripts makes it confusing for any new system administrators. Having a script not respond to 'stop', 'start', or 'reload' is also difficult and should be avoided.

Disabling a Run Control Script

Disable a run control script by renaming it with a dot (.) at the beginning of the new file name. Files that begin with a dot are not executed. If you copy a file by adding a suffix to it, both files will be run.

How to Disable a Run Control Script

1. Become superuser.
2. Rename the script by changing the case of the S to lowercase.

```
# cd /etc/rcn.d
# mv Sfilename sfilename
```

3. Verify the script has been renamed.

```
# ls
# sfilename
```

It is important to not actually remove files out of these directories in case certain services that are not necessary at the moment be necessary in the future. Nothing is more difficult than a system administrator trying to start processes without the appropriate startup scripts – its simply a good practice.

Run Control Script Summary

<i>Script Name</i>	Description (Performs the following Tasks)
/sbin/rc0	<ul style="list-style-type: none"> • Stops system services and daemons • Terminates all running processes

	<ul style="list-style-type: none"> • Unmounts all file systems
/sbin/rc1	<p>Runs the /etc/rc1.d scripts to perform the following tasks:</p> <ul style="list-style-type: none"> • Stops system services and daemons • Terminates all running processes • Unmounts all file systems • Brings the system up in single-user mode
/sbin/rc2	<p>Runs the /etc/rc2.d scripts to perform the following tasks:</p> <ul style="list-style-type: none"> • Mounts all local file systems • Enables disk quotas if at least one file system was mounted with the quota option • Saves editor temporary files in /usr/preserve • Removes any files in the /tmp directory • Configures system accounting • Configures default router • Sets NIS domain and ifconfig netmask • Reboots the system from the installation media or a boot server if either /.PREINSTALL or /AUTOINSTALL exists • Starts inetd and rpcbind and named, if appropriate • Starts Kerberos client-side daemon, kerbd • Starts NIS daemons (ypbind) and NIS+ daemons (rpc.nisd), depending on whether the system is configured for NIS or NIS+, and whether the system is a client or a server • Starts keyserv, statd, lockd, xntpd, and utmpd • Mounts all NFS entries <ul style="list-style-type: none"> • Starts nsd (name service cache daemon) • Starts automount, cron, LP print service, sendmail, utmpd, and vold daemons
/sbin/rc3	<p>Runs the /etc/rc3.d scripts to perform the following tasks:</p> <ul style="list-style-type: none"> • Cleans up sharetab • Starts nfsd • Starts mountd • If the system is a boot server, starts rarpd, rpc.bootparamd, and rpld • Starts snmpdx (Solstice Enterprise Agents(TM) process).
/sbin/rc5 and /sbin/rc6	<p>Runs the /etc/rc0.d/K* scripts to perform the following tasks:</p> <ul style="list-style-type: none"> • Kills all active processes • Unmounts the file systems
/sbin/rcS	<p>Runs the /etc/rcS.d scripts to bring the system up to run level S. The following tasks are performed from these scripts:</p> <ul style="list-style-type: none"> • Establishes a minimal network • Mounts /usr, if necessary • Sets the system name • Checks the root (/) and /usr file systems • Mounts pseudo file systems (/proc and /dev/fd) • Rebuilds the device entries for reconfiguration boots • Checks and mounts other file systems to be mounted in single-user mode

Exercise

1. How do you change the run level of a system?
2. What happens when the system is changed from a lower run level to a higher run level, and vice versa?

For more in-depth documentation, visit: <http://docs.sun.com>

Conclusion

Once you have gotten to this point, you probably know more than is necessary in order to use your Solaris system. However, its always nice to know the internals of how everything works and the design process involved. Modules covered more specifics and daily operations of your Solaris environment will follow this overview of the Boot Process.