Building and Debugging SunOS Kernels

Hal L. Stern
Sun Microsystems
Northeast Area Consulting Group
Lincoln, MA
hal.stern@east.sun.com
May 6, 1991

Building kernels and looking at kernel core dumps should not be strictly reserved for people writing device drivers and porting operating systems. Every system administrator should be familiar with the procedures for building a kernel and analyzing a core dump from a system that has crashed or "hung." You will need to rebuild a kernel to perform configuration tuning, install new hardware, or integrate operation system patches. If you are able to analyze kernel problems, you will be able to identify troublesome hardware, drivers or required patches. This paper describes the procedures involved in configuring, building, debugging and analyzing a SunOS kernel on SPARC-based machines.

Introduction

This paper starts with the basics of configuring, building and booting a new kernel, and then looks at what happens when the new kernel (or an existing one) doesn't behave as well as expected.

Section 1, *Configuring and Building a Kernel*, looks at the kernel build area and the procedures needed to configure, build and boot a new kernel. It is a short summary of material that may also be found in the *SunOS System Administration Guide*.

Section 2, *Kernel Configuration*, explores the kernel configuration file, device identification and the configuration process in more detail.

Section 3, *Adding Devices*, contains a short summary of the steps needed to add a new device to the kernel. This is a detailed explanation of the installation process used by many layered products such as SunLink drivers and VME bus devices.

Section 4, *Debugging and Core Dump Analysis*, starts with a summary of kernel failures, and then covers crash analysis, tracing, and debugging techniques. It also takes a look at debugging a loadable device driver using `adb`.

The problems and techniques discussed in this paper are meant to be representative of actual issues facing a system administrator or kernel service developer.

1. Configuring and Building a Kernel

There are many reasons to build a new SunOS kernel:

- Performance tuning: increasing sizes of kernel parameters.
- Configuration changes: adding a new device, such as a disk, or adding a new service such as the lightweight process library. Alternatively, you may want to build a kernel that is as small as possible, freeing memory for use by user-level processes.
- Patch integration: installing a kernel object file patch, or building a new kernel object module from source code.

This paper assumes that you are building a kernel from an object code distribution, such as that in the `sys` kit on the SunOS installation tape. All of the examples will assume that you are running on a `sun4c` machine; if you are building kernels for a `sun4` machine, substitute the `sun4` kernel architecture for `sun4c`.

1.1. Kernel Build Area

Kernels are configured and built in the `/sys` directory, which is a symbolic link to the directory `/usr/kvm/sys`. Everything that is kernel architecture specific lives in one of the subdirectories of `/usr/kvm`, so this is the most logical place for the kernel files themselves to reside. It's important to note that you should only build a kernel for a machine on that machine or a machine of the same kernel architecture. This is particularly true for the members of the SPARCStation family, since the machines are binary compatible with other SPARC machines but have their own kernel architecture. To determine the kernel architecture of your machine, use `arch -k`:

```
% arch -k
sun4c
% arch
sun4
```

The output of `arch` (with no arguments) shows you the binary architecture of the machine; all SPARC machines will be `sun4`-compatible. However, machines with the `sun4c` kernel architecture - the SPARCStation 1, IPC, SLC and SPARCStation 2 - cannot run kernels built on `sun4` machines. The `sun4c` kernel architecture differs from the `sun4` in a number of ways, including a different page size, main bus architecture, and SCSI device driver structure.

In the `/sys/sun4c` subdirectory, you'll find the `OBJ` directory with all of the kernel object modules, the kernel configuration file directory, and one subdirectory for each kernel that has been built. The other subdirectories of `/sys` contain configuration, header and source files. The source files included in the object code-only kernel distribution are used to configure various options into or out of the kernel. Most of these files contain tables whose sizes are based on the number or types of devices configured into the kernel. The following summary describes the relevant subdirectories of `/sys` and the kernel features dependent on them:

conf.common Common configuration files and tables, such as the system parameter file `param.c`.

net — Low-level network interfaces. This directory has configuration files that define address families, the NIT interface, and the ARP protocol.

netinet — Internet protocols: TCP, UDP, IP and ICMP. The source files in this directory can be used to change buffering and other behavior of the TCP and UDP protocols.

nfs — Headers for the NFS protocol.

os — General operating system features; files in this directory build the generic kernel interfaces. The `init_sysent.c` file contains the table of system call entry points. The table of filesystem types is in `vfs_conf.c`.

rpc — Kernel's interface to RPC services. The kernel uses RPC to make NFS requests and to talk to the lock manager on local and remote hosts.

sbusdev — SBus devices. This directory does not exist on non-`sun4c` machines.

scsi — Sun's Common SCSI Architecture (SCSA) devices. This directory is not used on non-`sun4c` machines, since they use the older SCSI architecture.

sparc — Files specific to the SPARC architecture. The assembly linkage definitions, trap tables, processor priority levels and other CPU-specific things are defined here.

specfs — Special (raw) filesystem code. *specfs* is the filesystem type used for swap devices.

sun — Sun-specific kernel files that are kernel architecture independent, such as the STREAMS configuration file `str_conf.c`. All of the possible root and swap devices that may be found when using a generic kernel are defined in the **swapgeneric.c** file in this directory. This file also defines the order in which the devices are probed.

sun4c — Header files specific to the `sun4c` kernel architecture, including virtual memory system configuration, MMU hardware and system parameters. This directory also has the architecture-specific kernel configuration, build and object file subdirectories.

sundev — Sun-specific device drivers. All VME devices, such as disk controllers, ALM boards, and various framebuffers, as well as hardware on the CPU board such as the Zilog UART, are configured from this directory. In addition, the older, pre-SCSA SCSI devices and host adaptors use this directory rather than the `scsi` directory.

sunif — Sun network interfaces. The network interface subsumes the data link and physical layers of the network protocol stack. The drivers configured in this directory include the various Intel and Lance ethernet interfaces.

sys — Kernel header files. These should be identical to those in the `sys` subdirectory of the C compiler's default, `/usr/include`, although this directory may contain a few kernel building-specific files.

ufs — UNIX (local) file system configuration.

vm — Header files for the virtual memory system.

sun4c/OBJ — Object files built for the `sun4c` architecture.

sun4c/conf — Kernel configuration files. New kernel configuration files are created in this directory, which also contains the `files` table used by `config` to locate kernel components.

sun4c/XX — Kernel-specific build directory for the kernel named `XX`. This is where you'll build the kernel itself, and where object code for compiled source files ends up.

When adding new services to the kernel, you can create new subdirectories of `/sys`, or add files to the existing subdirectories. For example, a new network protocol called `bdlc` should go in a new subdirectory `/sys/bdlc`, while a new Sun-specific VME device driver would probably live in `/sys/sundev`. We'll come back to this issue later when we talk about adding files to the kernel. First, we'll take a look at building a new kernel from the stock configuration and object files.

1.2. Building a Kernel

Let's say you want to build a new kernel, adding in support for another SCSI disk and the asynchronous I/O facilities. The new SCSI disk will be device `sd1`, and the lightweight process stuff is defined as options `ASYNCHIO` and `LWP`. Don't worry about where these names come from; we'll get to that in the next section when we look at how the kernel configuration file is laid out. This short section is really meant to provide a quick walkthrough of building and installing a new kernel.

Edit Configuration File

The first step in building a new kernel is to create a configuration file for it. To be safe, copy the configuration file for the currently running kernel, and make your changes to the copy. Pick a name for your new kernel, and use that name for the configuration file. We'll assume that we have a system running the `ONEDISK` kernel, and we're going to make a new kernel called `TWODISK`:

```
# cd /sys/sun4c/conf
# cp ONEDISK TWODISK
```

The kernel name is used to identify the configuration file and kernel build directory only; it doesn't show up anywhere else. You will probably want to change the kernel identification string to match the kernel "name" so that you can trace a kernel image back to the configuration file that produced it. As a general rule, don't call a kernel "generic" unless it really does contain all possible devices and options.

Changing the kernel's identification string is done by editing the configuration file. To effect all of the kernel changes we want, we'll make a few changes to the configuration file that we just created, `/sys/sun4c/conf/TWODISK`:

[1] Locate the configuration line for `ident`, which looks something like:

```
# name this kernel ONEDISK
#
ident "ONEDISK"
```

Change the string to reflect the new kernel name. In this example, we'll make it `TWODISK`:

```
ident "TWODISK"
```

[2] Add support for the lightweight process facility. As you page through the configuration file, you'll see a section of `options` lines, of the form:

```
options RFS
options WINSVJ    # SunView 1 journaling support
```

Add the new options to this set of configuration lines:

```
options LWP       # lightweight process support
options ASYNCHIO  # asynchronous I/O
```

[3] Add in support for a second SCSI disk. Go down into the section of the configuration file that defines disk devices, and add (or uncomment) a line for the new disk we're adding to the kernel. You should see some lines that look like:

```
disk sd0 at scsibus0 target 3 lun 0   # first hard SCSI disk
#disk sd1 at scsibus0 target 1 lun 0  # second hard SCSI disk
```

Uncomment the line for device `sd1`, or add it if it's been removed from the configuration file. This line defines a new disk device on the SCSI bus. It's permissible to have devices in the configuration file that are not actually present on the system; when the kernel boots it will probe each device that is configured in and determine dynamically what is and is not attached. This process, known as *autoconfiguration*, produces the list of devices and addresses seen during the boot process.

Configure the Kernel

Now that you have modified the kernel configuration file, you need to create an area in which to build the kernel and a set of rules for assembling its components. The `config` utility performs both functions. If a kernel build area does not exist for the kernel, **config** creates one. Be warned that `config` will overwrite the contents of an existing kernel build directory - if you need to save a kernel configuration for later use, be sure to choose a new name for your new kernel rather than re-using an existing name.

`config` builds a Makefile for the kernel using information from the kernel configuration file and the parts lists of component files in the `/sys` directories. The Makefile produced by `config` contains kernel compilation options, the list of object files, and the rules and dependencies needed to rebuild each of the source files in the kernel. Every time you modify the kernel configuration, you must run `config` again, because you will be changing one of the options that builds the kernel Makefile.

To configure the kernel, run `config` from the `/sys/sun4c/conf` directory where the `TWODISK` configuration file was created:

```
# cd /sys/sun4c/conf
# config TWODISK
Doing a "make depend"
```

There are a variety of errors that can be reported by `config`: you might have defined too many disks in the system, or may have asked to configure in a SCSI disk without first defining a SCSI bus on which to attach it. `config` will identify any configuration errors and abort.

`config` first constructs header files that identify the devices configured into the kernel, then it puts together a template Makefile referencing all of the object and source files needed to build the new kernel. Finally, the "make depend" step builds the dependency list for each source file, adding to the list of rules in the Makefile. As you add source files to the kernel, the "make depend" step will require more time to complete.

Note that being able to configure the kernel doesn't mean that it will build correctly. `config` doesn't compile anything; it just builds rules for compiling the kernel. It's perfectly legal (and quite common) to configure a kernel with a source file that has syntax errors in it.

Build the Kernel

After a successful configuration, you're ready to build the kernel. To do so, simply move into the kernel build directory and use the Makefile created by `config`:

```
# cd ../TWODISK
# make
```

You will see a series of `cc` commands being executed. Each of the source (.c) files that is needed to build the kernel is compiled, and the resulting objects are linked with the appropriate object files in `/sys/sun4c/OBJ`. When the `make` procedure reports that it is *loading vmunix*, it's linking the kernel. The kernel build procedure finishes by telling you how large the new kernel is:

```
text    data   bss   dec     hex
1097728 163376 82280 1343384 147f98
```

These values are the size (in bytes) of the kernel code, initialized data and uninitialized data segments of the kernel. The total size reported is not the total amount of memory that will be used by the kernel once it is running, because the kernel dynamically allocates large tables as part of its initialization. Furthermore, as the kernel is running, it will dynamically allocate memory for various data buffers. Your best estimate of the actual size of the kernel is to subtract the amount of available memory (after the kernel has initialized) from the total memory in the system. Look for this information in the kernel's boot messages, recorded in `/var/adm/messages`:

```
mem = 16384K (0x1000000)
avail mem = 14671872
```

Also note that you *only* need to run `config` when you change the kernel configuration file. If you haven't changed the options used to build the kernel, or the list of component files that go into it, there's no need to build a new kernel Makefile and configuration header files. You can build and re-build a kernel by simply going into the build directory and issuing a `make`:

```
# cd /sys/sun4c/TWODISK
# make
```

If you're trying to iron bugs out of a device driver, you'll be building several versions of the kernel from the same build directory without reconfiguring it. The kernel contains its identification and version number as strings in the text image; each time you build a kernel from the same directory the version number is incremented. When you `config` a kernel, the version number is reset to one. We'll come back to the way in which version numbers and identification strings make it into the kernel when we cover kernel extensions in more detail.

1.3. Booting the New Kernel

Before you boot a new kernel, save the old one or one that you know will boot so that you can recover if your new kernel hangs the machine, crashes or fails to run for any other reason. The most common "safe" kernel names are `vmunix.o` or `vmunix.generic`; it's a good idea to keep a generic kernel around so you can recover just about any system configuration. The saved kernel image must be in the root directory (/) so that you can boot it in a pinch.

Once you're sure you have a fallback position with an old kernel, move the new kernel into place:

```
# cd /sys/sun4c/TWODISK
# mv vmunix /vmunix
```

At this point, the kernel image loaded into memory and the disk image in `/vmunix` no longer match, so commands that read the kernel memory to locate user or process information will not work until you reboot. Pay particular attention to `ps` and `uptime`, since they will not be able to find process information, system information or any other kernel data structures that they parse. If you need to run `ps` before you reboot, you can explicitly specify a kernel image name using the `-k` option:

```
# ps -aguxk /vmunix.generic /dev/mem
```

To avoid confusing innocent users (or anyone else), you should install the kernel as close as possible to the time at which you will reboot the system.

With the new kernel installed, reboot the system. Depending upon your level of confidence in the new kernel, either get coffee (exuding confidence) or spread the entrails of a goat on the console (exuding fear). If the new kernel won't boot or crashes, you can always boot your saved kernel image. From the boot prompt, specify the name of the fallback kernel that you created above:

```
> b vmunix.generic
```

You can get a list of bootable images by supplying a wildcard to the `b` command:

```
> b *
```

If you didn't save a kernel image and can't get the new kernel to boot, all is not lost if you're running on a diskless node. Halt the diskless machine, and log into its root filesystem server as *root*. On the server, copy a known, working version of the kernel into the client's root filesystem, and then reboot the client:

```
server# cd /export/root/deadclient
server# cp ../goodclient/vmunix vmunix
```

If you're running on a diskful system, it's time to get creative moving disks around (you can try attaching the disk to another, working system, and copying a kernel onto the root partition) or locate your SunOS installation tapes. The miniroot on the installation tape is a generic kernel, so you can boot it and avoid re-installing your system:

- Boot the miniroot from the installation tape
- Make sure the root filesystem on disk is clean, using `fsck`.
- Copy the `/vmunix` image from the miniroot to the disk's root partition.
- Reboot from the disk image.

Take the same precautions with kernels that you would with any sharp object. Make sure you have a recovery plan in place, and you should be able to experiment freely.

2. Kernel Configuration Details

Now that we've seen how to build, install and boot kernels, we'll look at the process of kernel configuration in more detail. The heart of the configuration process is the kernel configuration file, which is divided into four major sections: machine type and connections, devices, pseudo devices, and options. We won't go into the details of how to build configuration lines for external devices, since this material is well covered in the *SunOS System Administration Guide*. Instead, we'll focus on how the various lines in the configuration file turn into file names in the kernel Makefile.

2.1. Machine Type and Connections

The machine type and connections section describes the CPU and bus architectures of the host. Connection definitions determine what other kinds of devices `config` will allow you to add to this kernel. For example, you can't attempt to configure a VME-based disk controller into a desktop workstation that only has SBus slots.

The machine definition has two parts: an architecture specification and a CPU type. The architecture specification describes the kernel architecture of the machine.

```
machine "sun4c"
cpu "SUN4C_60" # Sun-4/60
cpu "SUN4C_70" # Sun-4/70
```

You can specify only one kernel architecture per configuration file, since all device and connection information is pretty hardware specific. However, you can have different CPU types defined in a single configuration file. This allows you to build a kernel that will run on different generations of machines with the same kernel architecture. If you look at the `GENERIC` configuration for *sun4c* machines, you'll see both the `SUN4C_60` (SPARCStation 1, IPC and SLC) and `SUN4C_70` (SPARCStation 2) CPU types defined. The same `GENERIC`-derived SunOS 4.1.1 kernel will run on both SPARCStation 1 class and SPARCStation 2 hosts.

Included with the machine type section are the kernel identification string and the `maxusers` parameter. `maxusers` is a misnomer: this value does not determine how many users may be logged in, and it has no relationship to any licensing information. It is used to size various kernel tables and buffer pools, and it should be made large enough to reflect the number of *virtual* users on a system. For example, an NFS server might have 40 NFS clients. The aggregate disks traffic generated by those clients looks (to the server) like the same load imposed by 40 users, so the server should be built with `maxusers` set to 40. When memory was a scarce resource, and shaving 10 kbytes out of the kernel would add 1% to the pool of memory available for user processes (on a 1 Mbyte machine), `maxusers` had to be kept as small as possible. However, today's SunOS kernel can easily grow to be over 1 Mbyte, and increasing `maxusers` to 128 or more does not consume a significant portion of the system's memory. Many performance problems can be traced to a system that is built with too small a value for `maxusers`.

On a `sun4c` machine, you'll find only one connection, defining the SCSI bus attached to the embedded SCSI processor (esp):

```
scsibus0 at esp # declare first scsi bus
```

Connections have much more importance on VME bus machines. On a SPARCServer 490, you'll find definitions for the various VME bus address spaces, as well as the I/O devices that are on-board and on-board memory address spaces:

```
controller obmem 4 at nexus ?
controller obio 4 at nexus ?
controller vme16d16 4 at nexus ?
controller vme24d16 4 at nexus ?
controller vme32d16 4 at nexus ?
controller vme16d32 4 at nexus ?
controller vme24d32 4 at nexus ?
controller vme32d32 4 at nexus ?
```

The *controller* keyword indicates that this device will have other devices connected to it, while the use of *nexus* is historical (read one of the books describing implementations of BSD UNIX on DEC hardware for details). The number in between the controller name and the "at nexus" keyword indicates the CPU type for which the connection definition is valid. CPU types are defined in `/usr/include/machine/cpu.h`; these values will be referenced by the `cpu` definitions just described. Even within a single kernel architecture, not all CPU types support all connection types. Some systems don't handle 32-bit VME addresses, for example, so they won't have a `vme32d32` connection defined. We'll touch on connection types again when we look at the way in which devices are defined to the kernel, but for `sun4c` machines, they aren't all that interesting.

2.2. Devices

The devices section includes one definition for each physical device attached to the system. "Attached" is used somewhat loosely, since some devices may reside on the CPU board rather than on some external bus. As mentioned above, the devices section may include more devices than are really present.

There are two ways in which devices are defined to the kernel: a full specification including bus address and interrupt vector, or an "autoloading" specification. The full specification is typically used for devices that plug in to some external bus, such as the VME bus, while the autoloading style is used for devices that are on the same "bus" as the CPU. The best (and only) example of the latter is the SBus.

A good example of a full device specification is the configuration line for the on-board Intel ethernet interface on a SPARCServer:

```
device ie0 at obio ? csr 0xf6000000 priority 3
```

This line defines the device name `ie0`, which is connected to the on-board I/O "bus" (`obio`). The `?` indicates that this devices can be connected to the `obio` bus of any CPU type; if this device could only be used with the SPARCServer 490, for example, the `?` would have been replaced with a `4` (since `4` is the CPU type for this machine, as defined in the `cpu.h` header file). The `csr` specification tells the kernel where the device will map its on-board registers, and the priority is the bus priority level on which the device will interrupt. It's also possible to specify the interrupt vector and interrupt routine, as in this configuration line for an additional Intel ethernet interface that is a VME bus device:

```
device ie2 at vme24d32 ? csr 0x31ff02 priority 3
      vector ieintr 0x76
```

The `sun4c` machines use the simpler "autoloading" device specification. SBus devices are addressed geographically (by slot) rather than within some global bus address space (as in the VME bus), so no `csr` or bus type specifications are needed. Instead, the SBus driver configuration simply lists the name of the device:

```
device-driver audioamd # AMD79C30A sound chip
device-driver le # LANCE ethernet
device-driver zs # UARTs
```

Aside from the fact that SBus drivers have physical hardware associated with them, their configurations are very similar to those of pseudo devices.

2.3. Pseudo Devices and Options

The pseudo devices section adds device drivers that have no corresponding piece of hardware associated with them. Pseudo-tty drivers and the network loopback are two good examples of pseudo devices. Consider adding a new network protocol (like HDLC) to the system: the protocol doesn't have any associated hardware with it, since the low-level hardware protocol is handled by something else. The new network protocol can be added as a pseudo device while the underlying hardware driver is added as a physical device.

Like the SBus device configuration, adding a pseudo device to the kernel merely involves listing the device name in the configuration file:

```
pseudo-device pty    # pseudo-tty's, also needed for SunView
pseudo-device ether  # basic Ethernet support
pseudo-device win128 # up to 128 windows
```

This example includes support for the pseudo tty drivers, used by `rlogin`, `telnet` and the various window systems. Pseudo tty devices provide a tty interface for applications like `rlogin` that need to talk to a tty device. There's no hardware under the `pty` driver - it's all done in software. Pseudo devices can have entries in the `/dev` directory along with physical devices; for example, the pseudo tty devices are `/dev/ttyp*`. If you open the device directly - like a terminal device - then it will have an entry in `/dev`. Network protocols that are sandwiched between low-level hardware and higher-level interfaces, for example, do not need `/dev` entries.

The last line of the example above configures 128 `win` devices; the device name is the prefix and the numeric suffix indicates how many "units" the kernel should configure. Unit numbers are usually well-defined for physical devices, since the `config` utility can count how many instances of each device are included in the kernel configuration. Without some external piece of hardware, though, there's nothing to count for pseudo devices, so the name-and-number syntax is used to explicitly define the number of units. If no number is supplied, the driver usually has some default number of units.

The configuration of kernel options is equally simple:

```
options NFSCLIENT  # NFS client side code
options NFSSERVER  # NFS server side code
```

Options are software bundles that provide specific services. In this example, we add the NFS client and server code to the kernel. Configuring an option also enables system calls that use the corresponding service.

In our previous example using the asynchronous I/O option, adding `options ASYNCHIO` to the kernel also enables the `aioread()` and `aiowrite()` system calls. Where is the correlation between the configuration option and the system call? The answer is in conditional compilation: when you configure an option, the name of the option is passed as a `#define` constant to the kernel source file compilation. The asynchronous I/

O system calls are conditionally compiled (look at `/sys/os/init_sysent.c`) into the system call entry point table if the constant `ASYNCHIO` is defined. The same holds true for the NFS client and server options: the system calls that support NFS operation are compiled into the kernel only if the appropriate options are given in the kernel configuration file.

You can also use configuration file options to increase the default sizes of System V IPC services. For example, the maximum size of a shared memory segment is set by the `SHMSIZE` constant in `/usr/include/sys/shm.h`. Override the default value of this constant by redefining the option in the kernel configuration file:

```
options      SHMSIZE=2048
```

This example increases the default shared memory segment size to 2 Mbytes (the size is given in 1024 byte increments).

2.4. Putting the Pieces Together: config

Now you should have a pretty good idea of what goes into the kernel configuration file, which is the primary input of the `config` utility. It's up to `config` to determine what files are needed to build the kernel, locate these files, and build a Makefile for the kernel that includes source file dependencies and rules for compiling and linking the kernel. In this section we'll look at the various files and file excerpts produced by `config` and see how they fit together when the kernel is compiled and linked.

Compiler Options

Some of the Makefile excerpts generated by `config` are hard-coded, for example, the include file search path and some basic compiler options. Other compiler options are defined by including `options` lines in the kernel configuration file. If you define the NFS options (as above), then you'll see these constants defined to the compiler when the kernel is built:

```
# cd /sys/sun4c/TWODISK
# make
...
cc -sparc -c -O -Dsun4c -DTWODISK -DSUN4C_70 -DSUN4C_60
-DWINSVJ -DVDDRV -DASYNCHIO -DLWP -DVFSSTATS -DRFS -DCRYPT
-DTCPDEBUG -DIPCSHMEM -DIPCSEMAPHORE -DIPCMESSAGE -DSYSAUDIT
-DSYSACCT -DHSFS -DTMPFS -DNFSSERVER -DNFSCLIENT -DUFS
-DQUOTA -DINET -DKERNEL -I. -I.. -I../..
../../netinet/in_proto.c
```

The laundry list of `#define` options on the `cc` command line comes from the `options` and CPU type definitions in the configuration file.

Unit Header Files

As mentioned above, `config` needs to determine how many units of each device type will be configured into the kernel. The `config` utility passes this information to the kernel build procedure in unit header files, which are placed into the kernel build directory, for example, `/sys/sun4c/TWODISK`. These header files each contain one line that as a

`#define` constant for the number of units. The file names are the device names with a `.h` header file suffix.

Consider the kernel configuration file with the following SCSI devices defined:

```
scsibus0 at esp                      # declare first scsi bus
disk sd0 at scsibus0 target 3 lun 0 # first hard SCSI disk
disk sd1 at scsibus0 target 1 lun 0 # second hard SCSI disk
tape st0 at scsibus0 target 4 lun 0 # first SCSI tape
```

`config` will generate three header files for these device configurations:

```
sd.h contains
#define NSD 2

st.h contains
#define NST 1

sr.h contains
#define NSR 0
```

The constants in each file start with the letter `N`, and use the capitalized device name as a suffix. Since there are two `sd` devices in the configuration file, `sd.h` defines `NSD` as 2 as well.

One unit header file will be generated for every standard device known by `config`, even if it does not appear in the configuration file. There are no CD-ROM devices in the above configuration, but `config` still generates an `sr.h` file, with `NSR` set to zero. A unit header file will also be generated for every pseudo-device defined, with the unit number defined as one (if no numeric suffix is given) or the number of units specified in the configuration line. For example, defining 128 window devices with:

```
pseudo-device  win128
```

creates the header file `win.h` containing:

```
#define NWIN 128
```

The number of units defined in these header files are used to size various per-device arrays, and also to conditionally compile drivers into the kernel. We'll see how this happens when we look at the device switch tables later on.

Kernel Object List

The more difficult task faced by `config` is locating all of the files needed to build the kernel:

[1] `config` starts with a list of all of the standard files that must go into every kernel. These files contain code for the basic kernel services such as the virtual memory and process management systems.

[2] For every device, pseudo device and option configured into the kernel, `config` adds

the files needed to support them to its component list.

[3] Once the file list has been built, `config` determines whether these files need to be compiled or if they are available in object form. It turns the list of source files into a list of object files. For every file in the list, `config` looks for the source file. If the source file cannot be found, then `config` references the object file in `../OBJ`. If the source file does exist, then `config` generates Makefile rules for compiling the source file into an object file, and adds a reference to the object file in the current (kernel build) directory.

The third step means that you can compile just a few of the standard kernel files and override the distributed object files in `/sys/sun4c/OBJ` by putting the source file in the "right place." `config` will find the source file, and compile it as part of the kernel build instead of using the default version in the object directory. This is how you build a bug patch - you deposit the few files that contain the patch on a test system, and run `config` to build a kernel Makefile that incorporates your modified source files. There's still one big piece missing: How does `config` know what files are needed, and what the paths are to these files? This information comes from two tables that list kernel component files and the conditions on which they are included in the kernel build.

The two file tables are:

```
/sys/conf.common/files.cmn
/sys/sun4c/conf/files
```

The `files.cmn` table lists components that are needed for every kernel independent of kernel architecture. The `files` table is architecture-specific, and lists files that may not exist for all systems. Most of the standard kernel files are in `files.cmn`, while things like SBus drivers are in the architecture-specific table. `config` opens the architecture-specific table, which includes the common table through a `#include` directive.

Both tables have the same format:

```
net/af.c          standard
net/if.c          standard
lwp/alloc.c       optional LWP
lwp/process.c     optional LWP
lwp/schedule.c    optional LWP
os/tty_pty.c      optional pty
os/tty_ptyconf.c  optional pty
```

The first entry on each line is the path to the source file. Path names are given relative to the `/sys` directory, and `config` converts them into paths that are relative to the kernel build directory when it generates compilation rules. The file name is followed by either of the keywords *standard* or *optional*, indicating whether the file is required in all kernels or whether it gets included as a result of a configuration choice. For files that are optional, the last items indicate the name of the device driver, pseudo device or kernel option that requires the file. These tokens are case-sensitive; kernel options are usually in upper case while driver names are in lower case. To be included, all of the tokens on the line must appear in the kernel configuration file.

3. Adding Devices

Kernel hacking includes several different kinds of work: patching a kernel bug, adding new files or services to the kernel, and adding a driver for a new physical device. To be able to do any of these jobs, you'll need to tie together knowledge of `config`, the files it generates and your list of source files. Building a bug patch is similar to adding a new device to the kernel, except that you don't have to edit any of the kernel configuration files: just drop the modified source file into the right directory (as defined by the configuration tables), configure the kernel and then build it. In this section, we'll look at all of the changes necessary to add a new service or device driver to the kernel, and then see how the configuration process assigns a version number to the new kernel.

3.1. Adding Kernel Files

Let's say we want to add a new service called `kom`, contained in a single C source file named `kom.c`. To educate `config` about our new driver, we need to decide where the file should go in the kernel area and also whether it will be an option or device driver. If this is a basic service that will be needed by every kernel, then it might make sense to make the file part of the standard component list. For this example, we'll put the new source file in the new directory `/sys/kom,` and treat it like a device driver.

[1] We create the new directory and copy the source file into it:

```
# mkdir /sys/kom
# cp kom.c /sys/kom
```

[2] We then need to add `kom.c` to the `files` table. Using our favorite text editor, we edit `/sys/sun4c/conf/files` and add the line:

```
kom/kom.c   optional kom device-driver
```

[3] Now `config` knows how to find the driver, but we still need to include it in our kernel. Assuming we're still working on the `TWODISK` kernel, we edit its configuration file `/sys/sun4c/conf/TWODISK` and add the device driver specification:

```
pseudo-device   kom       # kom driver
```

[4] Now we run `config` and build the kernel. If the `kom.c` file has syntax errors or won't build for some other reason, the kernel build will fail and we'll have to edit the source file and fix the problems.

As mentioned above, you don't need to reconfigure the kernel to just rebuild it. If you fix a bug in the `kom.c` source file, just type `make` in the kernel build directory to create a new kernel - there's no need to re-run `config` because the last Makefile you built is still valid. The Makefile rules are still applicable even if the source files are broken.

3.2. Device Switch Tables

Up to this point, we've looked at how the kernel gets devices configured into it, but we haven't touched on how a user gets to a particular device through the `/dev` directory. Major device numbers and the device switch tables are the two major components of this interface. In this section, we'll look at how the switch tables work and how the major numbers are defined.

Block and Character Devices

The UNIX kernel handles two kinds of device: block and character oriented. Block devices usually provide random access to their data, but only in block-sized chunks. Probably the only block devices you'll run into are floppy and hard disks, both of which handle their data in sector-sized blocks. Character devices present a byte-oriented interface, sometimes included random access as well. The tty drivers are classic character oriented devices, but random access isn't very well defined on them. A VME memory board, however, can also be viewed as a character-oriented device, although random access is both defined and required for this driver.

Block devices have a *strategy* routine that maps read and write requests into disk blocks and handles a queue of requests for the device. A smart strategy routine may sort incoming disk requests by cylinder number to minimize seek time. Obviously, this kind of re-ordering has hideous effects on character device interfaces, where users depend on having their requests executed in the order in which they were executed. It's also worth noting that some block-mode devices also have character-mode drivers associated with them, called ***raw*** devices, that are used when the device needs to be accessed in a purely sequential fashion. If you want to read an entire disk using `dd`, for example, you'd use the raw disk device instead of the block device.

Switch Tables

The kernel's interface to block and character devices is the `bdevsw` and `cdevsw` device switch tables. These tables define the kernel entry points for each services - read, write, open, close, and so on - provided by the device drivers. The `bdevsw` switch entry for the floppy driver, for example, looks like:

```
{ fdopen, fdclose, fdstrategy, fddump, /*16*/
  fdsize, 0 },
```

The number in comment brackets is the major device number for the floppy. If you take a look at the floppy's entry in `/dev,` you'll notice that the major device numbers match:

```
# ls -l /dev/fd0
brw-rw-rw- 2 root 16, 2 Oct 16 12:22 /dev/fd0
```

The leading `b` indicates that this is a block device. We'll see where the major device numbers come from later in this section.

A typical character device switch table entry is a little different:

```
{ ptcopen, ptcclose, ptcread, ptcwrite,    /*21*/
```

```
ptcioctl, nulldev, ptcselect, 0, 0, 0, },
```

This entry defines the interface to the pseudo-tty device driver. Note that the character device switch doesn't have *strategy* or *dump* routines in it, but it does have support for the familiar *ioctl()* and *select()* routines. Like the block-mode device, the character device entry has a major device number in comments, which corresponds to its major device number in its `/dev` directory entry:

```
# ls -l /dev/ptyp0
crw-rw-rw- 1 root 21, 0 Feb 26 15:03 /dev/ptyp0
```

Both the `cdevsw` and `bdevsw` tables are in the file `/sys/sun/conf.c`, which is included in a standard binary kernel distribution.

Major Device Numbers

Major device numbers are the link between the user and kernel device interfaces. When a user opens up a device, the kernel uses the major number of the device as an index into the appropriate character or block device table to find the device-specific *open()* routine. If you open `/dev/ptyp0`, which has major device number 21, the kernel finds the open routine in entry 21 in `cdevsw`, and invokes that routine to complete the *open()* system call.

Major device numbers are therefore implicitly defined by the order of entries in the `bdevsw` and `cdevsw` tables. If you follow the numbers in table comments, you'll see that they increase linearly as you go through the table. Because of this positional definition of major device numbers, you must maintain the number and order of devices in the table. Otherwise, you will break all device interfaces. If a device is not present in the kernel, it should be represented by a placeholder entry that uses the `nodev` entry points (here shown for the `bdevsw`):

```
{ nodev, nodev, nodev, nodev,   /*0*/
  0, 0 },
```

If you attempt to use a device that has *nodev()* placeholders in the switch table, the system call will return a "No such device" error.

This points out an interesting problem in building a kernel: device driver entry points are defined in the switch tables, which must contain entries for all devices, but if a device type isn't included in the configuration file, `config` won't include its associated object module in the kernel build. How do you prevent undefined routine references (to unconfigured devices) from the switch tables?

The keys are the `nodev` entry point and the unit header files. In the `conf.c` file, you'll see a long list of conditional compilations of the form:

```
#include "sd.h"
#if NSD > 0
extern int sdopen(), sdclose(), sdstrategy(),
       sdread(), sdwrite();
extern int sddump(), sdioctl(), sdsize();
#else
```

```
#define sdopen nodev
#define sdclose nodev
#define sdstrategy nodev
#define sdread nodev
#define sdwrite nodev
#define sddump nodev
#define sdioctl nodev
#define sdsize 0
#endif
```

If the number of SCSI disk devices defined in the unit header file is non-zero, then the `conf.c` file makes the external references to the driver entry points. If the device isn't used, however, all of the entry points are aliased to *nodev()*. The `bdevsw` table entry for the `sd` device, then, either points to the actual driver or is just a placeholder that refers to *nodev()*. If no `sd` devices are configured into the kernel, the `sd.o` module is not compiled into the kernel, and no external references are made to it from the switch tables.

3.3. Adding a New Device

To add a new device, you must edit `conf.c` and either replace or add a new entry in the appropriate switch table. Let's say we want to add a device for the `kom` character device we configured in a previous section. In `conf.c`, we first have to add a conditional compilation stanza:

```
#include "kom.h"
#if NKOM > 0
extern int komopen(), komclose(), komread(), komwrite();
extern int komioctl(), komselect();
#else
#define komopen nodev
#define komclose nodev
#define komread nodev
#define komwrite nodev
#define komioctl nodev
#define komselect nodev
#endif
```

The `kom.h` header file will be created by `config` when it sees the pseudo-device configuration line. Our driver, in file `kom.c`, should define all of these entry points and contain the code to implement them. If the user chooses to exclude the `kom` driver, then its entry points will become aliased to *nodev()*.

With the external references set up, we need to add an entry to the `cdevsw` for our new driver. We can either choose an unused entry, or add a new one to the end. **NOTE:** We can't just drop our new entry into the middle of the table, because this will skew all of the device numbers that follow our addition. The major device numbers are strictly position-dependent, so we must either replace an entry or add to the end of the table.

We'll add our driver to the end of the table:

```
{ komopen, komclose, komread, komwrite, /*104*/
  komioctl, nulldev, komselect, nulldev, 0, nulldev. },
```

The major device number - 104 - was derived by adding one to the major number of the device ahead of ours in the `cdevsw` table. Note that routines that are undefined for this driver are now listed as `nulldev` instead of `nodev`; `nodev` means the entire device is missing (and returns an error) while `nulldev` just stubs out a particular function (and does not return an error).

Now the kernel knows how to find our device, and the kernel build procedure knows where to find the object or source code for the driver. We just need to give the user an interface to the device, by creating entries in `/dev` for it. Let's assume that the `kom` driver defines four units by default, so we'll create four minor units of major unit 104:

```
# cd /dev
# mknod kom0 c 104 0
# mknod kom1 c 104 1
# mknod kom2 c 104 2
# mknod kom3 c 104 3
```

In a user application, we can access the kom device using *open*():

```
fd = open("/dev/kom0", O_RDONLY);
```

and have the kernel locate the open routine for character device 104; it then calls *komopen()* with the arguments passed in the system call. Our device-specific open routine does whatever initialization is required, and the user is ready to read from the device using the file descriptor passed back by *open()*.

3.4. Kernel Version Numbers

Each time you build the kernel, the version number is incremented and a timestamp is put into it. You'll see this information in the `/etc/motd` file when you log into the system. When building the kernel, a short source file called `vers.c` is created and compiled. This file contains timestamp, version and kernel name information taken from the following files:

`/sys/conf.common/RELEASE`	SunOS release number (eg, 4.1)
`/sys/sun4c/TWODISK/version`	Kernel version number
`/sys/conf.common/newvers.sh`	Script for building `vers.c`

You may want to change the default version and timestamp information, or add more information that identifies particular changes you have made to the kernel. Some OEMs use this feature to change the name of the operating system, or to add their own comments to the timestamp.

If you modify `newvers.sh`, don't remove the word `SunOS` from the timestamp unless you also change the boot script that creates `/etc/motd`. The boot procedure searches the kernel for the string `SunOS`, and copies the entire string into `/etc/motd`.

4. Debugging and Core Dump Analysis

Getting a kernel built and installed is the easy part of integrating new system services. Debugging problems within the kernel is one of the more difficult problems in the UNIX world. There is both a lack of nice tools (like `dbxtool` or Saber C) and a lack of a well-behaved, time-sequenced execution environment. In a user application, you have fine control over the order of statement execution and the global state. The kernel, on the other hand, is a collection of service routines that get called on demand by interrupt or user requests. You could be in middle of your new device driver when a higher priority interrupt comes in and changes some global state. For example, you may assume that you'll be able to use some buffer once you're in your driver. However, if another device interrupts and grabs that buffer before you do, any number of unpleasant things may happen if you fail to check for this condition (or prevent it) in your driver.

This section first looks at common debugging techniques, including the use of `printf`() inside the kernel and the `kadb` debugger. Then we'll look at core dumps in some detail, including an explanation of bad traps and panics. This short introduction should explain why kernel hackers are the way they are.

4.1. Debugging Techniques

There are several ways to debug a device driver or kernel service: you can include debugging code that prints messages about variables and state information, you can trace the calls made by a driver, or you use the kernel debugger `kadb` to stop in a particular routine and single-step through a problematic piece of code. Each approach has its merits and its best applications: time constraints may make debugging statements or single-stepping infeasible. This section covers each technique and offers some suggestions for choosing when to use each approach.

Using printf()

The most common, and sometimes most dangerous way to trace kernel calling sequences and variables is with the friendly `printf`() statement. Output from `print`() calls in the kernel is sent to the `/var/adm/messages` file, and appears in the console window (or console device, if you have a terminal attached as a console). The major problem with this approach is that calling the print routine many times can take several seconds. If you put `printf`() calls inside of a device driver that has tight timing constraints, you will probably violate those constraints and break the driver in new ways.

You can put `printf`() calls in code that is called at user level, and inside of interrupt routines that have no time dependencies and run at low interrupt priority (below the level of the UART chip). Adding debug statements to the top levels of a device driver's `read`() routine, for example, is unlikely to cause side effects, because the time required to print the debug messages will simply slow down the user process making the `read`() system call. If you put these debug statements in the interrupt handler, though, you may exceed some timeout period set by the device, or you may miss another interrupt during the time that the kernel is printing the debug output.

Serial devices and network interfaces are poor candidates for debugging with `printf`() calls. Parts of the serial line drivers will be run at high interrupt priority, so calling `printf`() from inside the driver may create race conditions when the processor priority is temporarily lowered to print the debug message. Both serial drivers and network interfaces rely on efficient execution of interrupt code, so adding the overhead of console output to these routines makes it more likely that they'll miss incoming data or packets. If you need to examine a calling sequence in a driver like this, use an in-memory call tracer.

Call Tracer

A call tracer records the sequence of routines called within a driver in a small, kernel-resident buffer. You can examine the calling sequence by dumping the trace buffer using `adb`. Here's an overly simple example of a call tracer for the `zz` device driver:

```
#define ZZ_MAXCALLS     1024
int   zz_calls[ZZ_MAXCALLS];
int   zz_debug = 0;
int   zz_depth = 0;
#define ZZ_TRACE(x) if (zz_debug && (zz_depth<ZZ_MAXCALLS)) \
            { zz_calls[zz_depth] = x; zz_depth++; }

in interrupt routine zz_intr():
ZZ_TRACE(1);

in read routine zz_read():
ZZ_TRACE(2);

in write routine zz_write():
ZZ_TRACE(3);
```

We define an identifier for each routine that we want to trace; these identifiers are poked into the `zz_calls` history array whenever a `ZZ_TRACE()` macro appears in the driver. To debug a section of code with this driver, you first set `zz_debug=1` in the live kernel, and then let the driver run (and hopefully crash and burn as well). Within `adb`, you can then examine the calling sequence by dumping `zz_calls`:

```
# adb -k -w /vmunix /dev/mem
physmem ff3
zz_debug/W 1
zz_debug: 0x0  = 0x1
...use driver....
zz_calls,8/D
zz_calls:   2     1     1     1
            3     1     1     2
```

In this case, you can tell that the driver's read routine was called, followed by three interrupts, and then a call to the write routine. You may want to add timestamps to the call history, so you can tell how much time elapsed between calls (if you're trying to find timing problems). It's critical that the tracing macro check the current stack usage, and stop tracing calls when the call history array is full. Overflowing an array in the kernel is just as dangerous as doing so in a user-level process: it causes data corruption, unexpected results, or a core dump.

If you want to run the driver again, you can reset the call history counter using `adb`:

```
# adb -k -w /vmunix /dev/mem
physmem ff3
```

```
zz_depth/W 0
zz_depth: 0x400  = 0x0
...use driver again...
```

Resetting the trace counter to zero allows you to fill the array up again with fresh trace data.

The kadb Debugger

Not every problem can be debugged using `printf()` calls or a call tracer, particularly if it involves device setup and initialization code. You can't use `adb` until your system is up and running. If you can't get a new kernel to boot properly, you need to use `kadb` to examine the kernel during its nascent configuration stages. `kadb` is similar to adb in its user interface (or lack of one). The major difference is that `kadb` is a bootable image: you load it first, and then it loads the kernel so that you can debug the kernel before the system becomes functional.

Boot `kadb` from the PROM monitor:

```
> b kadb
```

If you are debugging a diskless client, you'll probably have to specify the boot device and `kadb` image:

```
> b le()kadb
```

If you want to set breakpoints before the kernel is loaded, boot `kadb` interactively using the `-d` option:

```
> b kadb -d
```

Once `kadb` is up and running, you can escape from the kernel back into the debugger using the L1-A abort sequence (or the BREAK key on a terminal). Within `kadb`, aborting the system again takes you back to the PROM monitor. See the manual pages for `kadb` and adb for more information on setting breakpoints, single stepping and continuing execution.

4.2. Core Dumps

The debugging techniques described in the previous section are useful if you are adding code or new devices to the operating system. However, you may experience system crashes or hangs with purely "standard" software and hardware. These problems fall into three classes: panics, bad traps and system hangs. The best way to debug any of these problems is with a core dump.

Saving Core Dumps

A core dump is a set of two files - `vmunix.`*n* and `vmcore.`*n* - that contain the kernel image that was executing and a complete physical memory image from the time the system crashed. When the kernel produces a core dump, these two files are written to the end of the swap space. When the system reboots, the `savecore` utility is used to copy them from the swap device into the `vmunix` and `vmcore` disk files.

To successfully save a core dump, you should have a swap space that is at least one and one-half times as large as the physical memory on your machine. The core dump gets written to the end of the swap devices so that the system can reboot and start basic services using the swap space at the beginning of the swap device; if there's any overlap you'll damage the core dump.

To save a core dump upon reboot, make sure you uncomment the `savecore` invocation in `/etc/rc.local`:

```
# Default is to not do a savecore
#
mkdir -p /var/crash/`hostname`
#               echo -n 'checking for crash dump... '
intr savecore /var/crash/`hostname`
#               echo ''
```

The core dump may be as large as your physical memory size plus the size of the kernel. Ensure that you have sufficient free space in the crash dump directory, and change the default dump directory if you are short of space. The core dump is copied after NFS and other system services are started, so you can put the core dumps on an NFS-mounted partition (if you have *root* access permission to the filesystem).

Types of Failures

Now that you know how to retrieve a core dump, it's useful to know how and why they are produced in the first place. Core dumps are produced whenever the system panics, as a result of an unexpected hardware event or a software-detected error. Probably the more familiar event is a "panic" message, produced whenever the kernel encounters a "can't happen" state. A typical panic message is:

```
panic: iechkcca
```

The panic messages are often cryptic, but are unique and have well-defined conditions that cause them. Usually, a panic occurs when the kernel's "reality check" fails: the system's view of the world (in software) does not agree with what the hardware is telling it. For example, the kernel might think that a disk isn't busy, but when it goes to start a command on the disk controller, the controller reports that the drive is in use. Somewhere, the kernel marked the device inactive when it really wasn't.

Panics may also be caused by race conditions that are exposed by new and faster processors. The previous panic example, `iechkcca`, is produced when the CPU checks the condition codes on the Intel ethernet chip and finds that they don't agree with what they *should* have been. The problem really lies in a race condition between the Intel hardware and the CPU: the Intel chip takes a little while to change the condition codes, and if the CPU checks them again before the chip has had a chance to reset them, it appears that the codes were never reset. A faster CPU will "beat" the Intel chip.

Core dumps are also produced by "bad traps." A bad trap is an unexpected hardware event, and usually implies a kernel address translation fault. This is a fancy way of saying that bad traps are generally kernel-level segmentation violations or bus errors. A user process dumps core when it causes a segmentation violation - and the kernel does the same. The most common cause of a bad trap is a dereferenced null pointer in some kernel code. In addition to producing a core dump, the bad trap handler also prints a stack traceback so that you can tell where the system encountered the problem. No traceback is produced for panics.

You can use panics to take a snapshot of the system for analysis that might otherwise mask the problem you're trying to detect. For example, if you're trying to find a timing problem or a race condition, you may want to cause a system panic when you detect that the kernel's state has become inconsistent with reality. Detecting these sorts of problems using debugging output is hard, because the race conditions can be eliminated through the addition of a the printf() calls. Force a panic in your code, and analyze the core dump outside of these time constraints.

4.3. Analyzing Core Dumps

Use the `adb` debugger to analyze core dumps saved by `savecore`:

```
# adb -k vmunix.0 vmcore.0
physmem ff3
```

`adb` takes the kernel image name and the core dump file as arguments; the `-k` option tells adb to perform kernel memory mapping on the core file. Use the `$c` command to generate a traceback of the kernel's stack at the time the system crashed:

```
sunwr8# adb -k vmunix.2 vmcore.2
physmem bec
$c
_panic(0xf81372b3,0xf8252c34,0x10,0x80,0xf8253) + 6c
_trap(0x9,0xf8252c34,0x10,0x80,0x2,0x0)+ 184
fault_60(?)
_uiomove(0x0,0x0,0x1,0xf8252eac,0xf8252ea4,0xd)+ a4
_lbwrite(0x0,0xf8252eac,0x340,0xff052ff0,0xf8147a34,0xd) + c4
_spec_rdwr(0xff052e44,0xf8252eac,0x1,0x0,0x1040,0x6801)+ 1bc
_vno_rw(0xf8161bb4,0x1,0xf8252eac,0xd,0xff052e44,0x0) +a4
_rwuio(0xf8161bb4,0xf8252eac,0xf8252ea4,0xd,0xd,0xf8252eac) + 2b0
_write(0xf8252fe0,0x20,0xf8121f28,0xf8121f48,0xf8253000) + 34
_syscall(0xf8253000) + 3b4
```

There are two things worth noting about using `adb` on SPARC systems:

- It's difficult for `adb` to determine what arguments were passed to a routine, because arguments are passed in register windows and not in the stack. Therefore, `adb` shows five or six arguments for each routine, even if the routine only takes one or two arguments.
- The traceback for a panic will be accurate, but on a bad trap, the kernel's stack may have already been damaged. It's possible that the stack reflects a point before the trap actually occurred. The example above was produced by a "bad trap" panic, and it shows a damaged stack.

How do we know that stack was damaged? First of all, the `fault_60`() routine is shown without any arguments; this could indicate that adb isn't clear on how this routine got called (maybe because it wasn't called by the code that produced the trap). Also, the panic occurred in a well-known kernel routine (uiomove), which is strange if the routine was called with well-defined arguments (that is, we didn't ask uiomove to copy data to a null pointer).

The best way to sort out a bad trap panic is to resort to the traceback recorded in the messages file /var/adm/messages. If you don't have the messages file corresponding to the core dump, you can look at the kernel's message buffer using adb:

```
sunwr8# adb -k vmunix.2 vmcore.2
physmem bec
msgbuf,80/s
....lots of messages from system boot....
Apr 18 11:14:08 sunwr8 vmunix: BAD TRAP
Apr 18 11:14:08 sunwr8 vmunix: pid 257, 'iclient': Data fault
Apr 18 11:14:08 sunwr8 vmunix: kernel write fault at addr=0x10,
 pme=0x0
Apr 18 11:14:08 sunwr8 vmunix: Sync Error Reg 80<INVALID>
Apr 18 11:14:08 sunwr8 vmunix: pc=0xf80bde44, sp=0xf8252c80,
 psr=0x4000c0, context=0x3
Apr 18 11:14:08 sunwr8 vmunix: g1-g7: f8000000, f8141000, ff050d90,
 ff0552a0, f8253000, 0, f813ac00
Apr 18 11:14:08 sunwr8 vmunix: Begin traceback... sp = f8252c80
Apr 18 11:14:08 sunwr8 vmunix: Called from f808dbd8, fp=f8252ce0,
 args=0 f8252eac 340 ff052ff0 f8147a34 d
Apr 18 11:14:08 sunwr8 vmunix: Called from f8065094, fp=f8252d68,
 args=ff052e44 f8252eac 1 0 1040 6801
```

The traceback indicates the program counter (pc) address at which the trap occurred. We'll use pass this address to adb to locate the real kernel location causing the problem:

```
f80bde44,4?ia
_lbwrite+0xd0: st %i3, [%i0 + 0x10]
_lbwrite+0xd4: st %g0, [%i0 + 0x18]
_lbwrite+0xd8: st %i5, [%i0 + 0x14]
_lbwrite+0xdc: ld [%i4 + 0x1c], %l2
```

What this shows us is that the panic didn't occur in uiomove(), but instead happened inside of the lbwrite() routine. Taking the program counter address from the traceback, we used adb to disassemble the code at this address, and adb's symbolic output shows us the routine name in which the trap occurred. We can deduce two things from the traceback and instruction disassembly:

- The panic was caused by a null pointer dereference. Notice that the fault address is 0x10, and the instruction causing the fault was writing into an address offset by 0x10. The address itself is probably a pointer with a NULL value.
- The same store instruction shows up with the same register three times in a row. This part of the code must have been filling in a structure without first checking to see if the structure existed.

As confirmation of these ideas, let's look at the code for `lbwrite()` and the data structures that it is manipulating:

```
 struct lb_unit {
      int   state;      /* state of device */
      char  *name;      /* name of this endpoint */
      char  *peer_name; /* name of peer  */
      struct lb_unit *peer;   /* peer unit */
      char  *buf;
      int   buflen;
};

lbwrite()
{     struct lb_unit *lbdest;
      ....
      lbdest = lbp->peer;
      lbdest->peer = lbp;
      lbdest->buf = NULL;
      lbdest->buflen = 0;
}
```

Now the pieces fit together: the driver was attempting to use the `lbdest` pointer, but `lbdest` is NULL. Notice that the `peer` element of the `lb_unit` structure has an offset of 16 bytes (0x10) - the same offset that appeared in the traceback.

4.4. Debugging Hung Systems

Panics let you know rather emphatically that there is a problem. System hangs, on the other hang, are much harder to detect and debug. A system hang can be caused when the kernel runs out of memory, or when many processes become deadlocked on the same resource. A system that is hung may respond to keyboard input, or answer `ping` requests, but most or all of the processes on the system will end up waiting on some resource. The first step in determining what made the system hang is to force a core dump.

When you take a system into the PROM monitor, the g0 command will force a "`panic: zero`," followed by a core dump and reboot. On a SPARCStation, put the monitor in new command mode and perform a `sync`:

```
sun4 system:
> g0
panic: zero

sun4c system:
> n
ok sync
```

You can run `ps`, `netstat`, `ipcs` and other kernel memory reading utilities on the core dump from a hung system to see what process and resource states were when the system became unusable. For example, let's assume that we're debugging the `zz` device driver, and the system has become deadlocked when several processes try to use it. Using the `-k` flag with `ps`, we can look at the process information in the core dump files:

```
 % ps -agxlk vmunix.2 vmcore.2
       F  UID   PID  PPID CP PRI NI  SZ  RSS WCHAN    STAT TT  TIME COMMAND
   80003    0     0     0  0 -25  0   0    0 runout   D    ?   0:58 swapper
20088000    0     1     0  0   5  0  52    0 child    IW   ?   0:00 /sbin/init -
   80003    0     2     0  0 -24  0   0    0 child    D    ?   0:02 pagedaemon
   88000    0    52     1  0   1  0  68    0 select   IW   ?   0:00 portmap
   88000    3    55     1  0   1  0  36    0 select   IW   ?   0:00 ypbind
20008001 1461   155   149  5   1  01824 1804 Sysbase  S    co  2:02 zzdriver
20008000 1461   149   112  1   5  0  28    0 child    IW   co  0:00 zztest
20008001 1461   153   149  5   1  01824 1804 Sysbase  S    co  2:02 zzdriver
20008001 1461   154   149  5   1  01824 1804 Sysbase  S    co  2:02 zzdriver
```

There are three processes - 153, 154 and 155 - stuck in wait states. Using adb on the core dump, let's see where they got deadlocked:

```
# adb -k vmunix.2 vmcore.2
physmem ff3
0t155$<setproc
                pid 155
$c
sw_bad(?)
_swtch(0x800ae6,0xf813ef90,0x4000e6,0xff01dc48,0x1,0xf8245000) + 80
_sleep(0xf813ef90,0x11c,0xf8116c00,0x0,0xa00,0xf8165ee0) + 188
_zz_checkstate(0xff01cb80,0xa,0x2,0x440,0x1c,0xff01dc48) + d4
_zz_read(0xa,0xf8244ea4,0x0,0x0,0xff01cb80,0x4000e1) + 30
_spec_rdwr(0xff092c74,0xf8244ea4,0x0,0x0,0x640,0x0) + 128
_vno_rw(0xf81614d0,0x0,0xf8244ea4,0x2000,0xff092c74,0x0) + a4
_rwuio(0xf81614d0,0xf8244ea4,0xf8244eb8,0x2000,0x2000) + 2b0
_read(0xf8244fe0,0x18,0xf811ff30,0xf811ff48,0xf8245000) + 34
_syscall(0xf8245000) + 3b4
```

The setproc macro locates the specified process and sets up the kernel to point to its address space. The traceback shows that the processes called read() to get data from the device; read() passed the generic system call down to the zz_read() routine for the zz device. However, zz_read() must have decided that it needed to check the state of the device, and when doing the state verification it called sleep(). The processes that are deadlocked may all be sleeping on the same event, which probably never occurred.

You can use adb on a live system to look at the same process and state information as the system is running; this may help you diagnose why one a process goes to sleep and starts the chain of events that leads to a deadlock. There are a wide variety of macros in /usr/lib/adb (setproc is one of them); use these macros to examine various kernel data structures on a live system or in a core dump.

4.5. Debugging Loadable Drivers

Loadable drivers present an interesting problem for debugging: because they are linked into the kernel once it is already running, their symbols are not part of the symbol table contained in the /vmunix image. To debug a loadable driver on a live system, you must build a new kernel image that contains symbols from the loadable driver and the running kernel image. To do so, use the incremental linking feature of the SunOS linker.

Let's say you've built and installed a loadable driver called `hello`. The `modstat` utility shows you where it was linked into the kernel:

```
# modstat
Id  Type  Loadaddr      Size   B-major  C-major  Sysnum   Mod Name
 1   Sys  ff0ea000      1000                      190     hello
```

The *loadaddr* reported by `modstat` is the relocation address of the module. We can have `ld` build a new kernel image from the `/vmunix` file and the `hello.o` object file, using this load address to specify where the `hello.o` code should be placed:

```
# ld -A /vmunix -T ff0ea000 hello.o -o vmunix.hello
# adb -k -w vmunix.hello /dev/mem
physmem ffe
```

The -`A` flag tells the linker that we're adding a new object file to the existing executable; the `-T` flag tells `ld` where to put the new code. We deposit the new kernel image in `vmunix.hello`, and then use that image with `adb`. The version of the kernel that is loaded in-core will match that in `vmunix.hello`, so we can look at symbols and code from `hello.o` using `adb`.

Further Reading

Any short tutorial like this is necessarily incomplete. For information, refer to the following sources:

The Sun Technology Papers, edited by Mark Hall and John Barry (Spring-Verlag, 1990). This book has an excellent description of the SPARC architecture and how the first SunOS port to SPARC was accomplished.

UNIX System Administration Handbook, by Evi Nemeth, Garth Snyder and Scott Seebass (Prentice-Hall, 1989). A little bit of everything for the system administrator.

System Performance Tuning, by Mike Loukides (O'Reilly & Associates, 1990). Throughout the book and in an appendix there are descriptions of just about every knob that can be turned in SunOS and System V Release 4.

Writing a Unix Device Driver, by Janet Egan and Thomas Teixeira (WIley & Sons, 1988). The book focuses on System V Release 3 and MASSCOMP's RTU UNIX system, but there are some good examples in it, including a fairly elaborate call tracer.

Writing Device Drivers, in the SunOS 4.1.1 Documentation Set (Sun Microsystems, 1990). A good place to look for memory maps, kernel address space layout, kernel services used by device drivers and other Sun-specific details.

The Design and Implementation of the 4.3 BSD UNIX Operating System, by Marshall McKusick, Mike Karels, Sam Leffler and John Quarterman (Addison-Wesley, 1989). This book explains where all of the "historical" names and techniques come from.

Acknowledgments

Many people contributed to this paper. Michael Bender (Sun) provided the ideas and scripts for the loadable driver debugging discussion. Janice McLaughlin (Sun) suggested the call tracer, and helped analyze several of the problems we uncovered with it. Rohit Aggrawal and Saul Wold (Sun); Randy Rohrbach (Foxboro Company); and Paul Bradford (Bytex) participated in discussions of debugging methods and actually tried out some of my suggestions. Rohit Aggrawal, Chuck Kollars and Chris Drake (Sun) reviewed the early versions of this manuscript; the paper has benefited significantly from their comments.