Tuning Resources with sysctl

/proc/sys/ - has directories for all interfaces the kernel offers: abi, crypto, debug, dev, fs, kernel, net, sunrpc, vm
/proc/sys/kernel/ for core kernel functionality, **/proc/sys/net/** for networking, **/proc/sys/vm/** for virtual memory
kernel files for different parameters like "osrelease" or "hostname"
cat /proc/sys/net/ipv4 spits out 1. Change this to off with **echo 0 > ip_forward** (not persistent)
vm contains "swappiness" holding a value 0-100 (eagerness of kernel to swapping out unneeded memory)
Default is 30. Greater value (60) might help kernel swap out faster. Change it the same way but it wouldn't stick after reboot (not persistent).

/etc/sysctl.conf contains the default system configurations which hold these sort of values. It should be edited if you want to change these values and have them stick after a reboot.

sysctl -a -- to display **/proc/sys/** values
sysctl -a | grep forward ---to show forwarding/ routing files/values

Just like you see in Java, net.ipv6.conf.lo.mc would refer to the file **/proc/net/ipv6/conf/lo/mc**

```
[root@rhelserver ~]# sysctl -a | grep forward
net.ipv4.conf.all.forwarding = 0
net.ipv4.conf.all.mc_forwarding = 0
net.ipv4.conf.default.forwarding = 0
net.ipv4.conf.default.mc_forwarding = 0
net.ipv4.conf.ens33.forwarding = 0
net.ipv4.conf.ens33.mc_forwarding = 0
net.ipv4.conf.lo.forwarding = 0
net.ipv4.conf.lo.mc_forwarding = 0
net.ipv4.ip_forward = 0
net.ipv6.conf.all.forwarding = 0
net.ipv6.conf.all.mc_forwarding = 0
net.ipv6.conf.default.forwarding = 0
net.ipv6.conf.default.mc_forwarding = 0
net.ipv6.conf.ens33.forwarding = 0
net.ipv6.conf.ens33.mc_forwarding = 0
net.ipv6.conf.lo.forwarding = 0
net.ipv6.conf.lo.mc_forwarding = 0
```

On boot the **sysctl** process is started. In the past, it just checked **/etc/sysctl.conf**, which is now empty (as of RHEL7). The file instructs that "default settings live in **/usr/lib/sysctl.d/00-system.conf** To override those settings enter new settings here, or in an **/etc/sysctl.d/<name>.conf** file"
So you can still put your custom stuff in **/etc/sysctl.conf**, but putting them in an **/etc/sysctl.d/<name>.conf** might be considered more organized if many custom changes are applied
Before you add custom scripts to **/etc/sysctl.d/** it will likely only contain **/etc/sysctl.d/00-sysctl.conf** which it turns out is simply a symlink to **/etc/sysctl.conf**

/usr/lib/sysctl.d/ contains: 00-system.conf 50-default.conf libvirtd.conf
The numbers in the filename represent the read-order

So, in **/etc/sysctl.d/** run **vim 50-ipforward.conf** and put in it **net.ipv4.ip_forward = 1**, save and on reboot you should be able to route IP packets.

The **sysctl** command promises to set the value and write to the **sysctl** files. Sander Vugt (whose examples I just used in this section on /proc and kernel modules) recommends against trusting it, instead suggesting to make sure by going directly to the source and echoing the value in **/proc** as demonstrated.

<u>sysctl command options</u>
sysctl to modify kernel parameters at runtime. The parameters available are those listed under /proc/sys/
-w {variable}={value} Write a parameter value/ change the sysctl setting.
{variable}={value} Set a key parameter value.
-n Disable the printing of the key name while displaying the kernel parameters.
-e Ignore errors about unknown keys.
-a Display all the parameter values that are currently available.
-A Display those in a tabular format.

More on /proc and /sys

cat /proc/cpuinfo - to display cpu info
/proc/net/ has files containing protocol information and settings, such as routing tables (used by netstat, ss)
/proc/meminfo - the used and unused memory and the shared memory and buffers used by the kernel.
/proc/version - produces same mostly as uname
/proc/cmdline - command line boot options passed to the kernel by the boot loader at boot time.
/proc/devices -list of device drivers (hardware) configured into the currently running kernel.
/proc/filesystems - list of filesystems that are configured into the kernel for mounting
/proc/partitions - partition info: the major and minor number of each partition, name, and number of blocks.
/proc/dma direct memory access. DMA gives hardware devices direct access to memory independent of the CPU.
/proc/interrupt lists the interrupt request (IRQ) channels in use.
/proc/iomem - mapping of the memory allocated to each device and the input/output port assignments for memory.
/proc/modules - lists the kernel modules that the computer is currently using.
/proc/bus - contains a file or directory for each USB device attached

Original UNIX systems needed to access structured data so user-space applications could find out about process attributes. Early programs like ps found out about the running processes by directly accessing **/dev/mem** or **/dev/kmem**, and interpreting the raw data). That requires root access, so the applications that use them have to be setuid or SGID. Also it's not good to expose system data directly to user-space. One solution was to make these types of info available through system calls, but creating new system calls over and over to export small process data wasn't so good either. The /proc filesystem was the answer- where interfaces and structures (directories and files) could be kept the same, even as the underlying data structures in the kernel changed.

On kernel startup virtual filesystems are made to reference hardware and resources:
- **udev** creates the virtual filesystem **/dev** to put hardware definitions (like addresses) into files representing them as interfaces so other things can talk to the them
- **sysctl** then helps set up **/proc** and **/sys** virtual filesystem representations of how the kernel modules and drivers to interact with those /dev device files

The **procfs** /proc virtual filesystem was only meant to hold legacy ***process*** information, system attributes from a few main systems. Since it is easily accessible from both kernel and user-space it eventually got the reputation as the convenient place to put other read/write system files to adjust settings, kernel and subsystem operation (cpuinfo, memory statistics, device information). Things then arbitrarily got thrown in created clutter with device data stuck in different spots all over the place.

The **sysfs** /sys virtual filesystem was implemented in kernel 2.5-2.6 (2003 or so) to organize things better, separate device and driver ***system*** information from **/proc** to add structure, provide a uniform way to expose system information and control points (settable system and driver attributes) to user-space from the kernel. For each object found to put in **/dev** on the system, the kernel automatically creates directories in **/sys** when drivers are registered based on the driver type and their values (representing the device hierarchy too)

Many of the legacy system information and control points are still accessible in /proc - entries already added to /proc were allowed to remain for backward compatibility (especially traditional items like CPU and memory). All new device busses and drivers are expected to expose their info and control points via sysfs.
Note: Many traditional UNIX and Unix-like operating systems use /sys as a symlink to the kernel source tree

<u>The /sys Directory Structure</u>
/sys/block - has an entry for each block device (mostly drives use data blocks)
/sys/bus/devices/ - symlinks for each device - point to device's directory under root/
/sys/bus/drivers/ has a directory for each device driver that is loaded
/sys/class/- has files for each class of devices
/sys/devices lists devices discovered, in a directory tree reflecting/representing the device hierarchy
/sys/firmware/
/sys/net/
/sys/fs/ - where each filesystem exporting attributes creates its hierarchy- see ./fuse.txt
/sys/dev/char/ and **/sys/dev/block/** hold symlinks named <major>:<minor> pointing to the appropriate

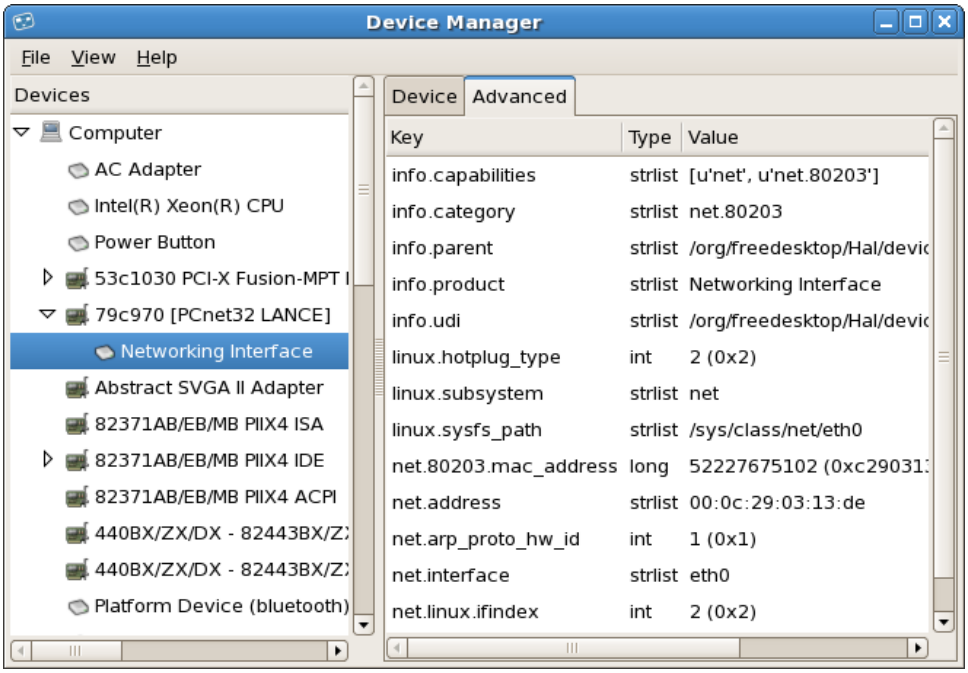
Quick examples:

Setting laptop brightness (not persistent)	**echo N > /sys/class/backlight/acpi_video0/brightness**
Get a network card's MAC address	cat /sys/class/net/enp1s0/address

/dev - Device Nodes

- Device drivers mapping service requests to device access represent HW resources in a device tree
- contains vital information such as the device type, with a minor # to identify device, major # to identify driver
- acts as an interface between the OS and hardware; part of OS or installed on-demand

Device Tree
- "a structure that lists all HW installed on a system and assigns device nodes to them."
- auto generated by the computer's RAM on startup, when a new device is installed, or when a device or system configuration is modified.



Special Devices

/dev/zero Provides unlimited null characters (0 bytes) for writing into any program or file. It is used for generating an empty file of certain size.

/dev/null Does not provide any data to a program or file. It discards all data written to it. It is used as an output file when the output is not required by the user and should be trashed.

/dev/random Functions as a random number generator; gathers random input from device drivers and other sources, saves it as bits in an entropy pool, output as bytes to applications.
When the pool is exhausted, will block the reading application until more random input is collected.

/dev/urandom Like /dev/random, except doesn't block the reading application if the entropy pool is exhausted; also uses pseudorandom input (less secure)

mknod [OPTION] {NAME} {TYPE} [MAJOR MINOR].
Create device files that are not present. Makes use of the major and minor node numbers of a device

Topics to cover:
The boot process
Troubleshooting logs of startup
  -boot events in the log files - **/var/log/messages** and **dmesg**

dmesg

dmesg shows info about all the hardware controlled by the kernel. CPU, memory, disk drives, network cards, etc. Shows the contents of **/var/log/dmesg**, which holds the kernel ring buffer (ring because it dumps old messages as new ones come in, maintaining the same file size)

It includes actions taken at startup, like configuring hardware devices.
See error messages as they occur with **watch "dmesg | tail -20"**
Setting the level to number 1 (-n 1) only allows panic messages to be seen.

dmesg > kernel_msgs.txt - dump current ring buffer contents into a file, handy for emailing for help troubleshooting
Pipe output to more, less, head, tail, or grep such as **dmesg | grep -i memory**
cat /var/log/dmesg can generate similar output, but running dmesg provides more control

Some options for dmesg:

-C, **--clear**	Clear the ring buffer. you can still view logs stored in '**/var/log/dmesg**' files
-c, **--read-clear**	Clear the ring buffer contents after printing.
-f, **--facility** *list*	Restrict output to defined (comma separated) list of facilities
-k, **--kernel**	Print kernel messages.
-l, **--level** *list*	Restrict output to defined (comma separated) list of levels.
-r, **--raw**	Print the raw message buffer, i.e., don't strip the log level prefixes.
-s, **--buffer-size** *size*	Output **size** to query the buffer; 16392 by default. If you set larger, can view entire buffer.
-u, **--userspace**	Print userspace messages.
-x, **--decode**	Decode facility and level (priority) number to human readable prefixes.
-n, **--console-level** *level*	Set priority level that messages are logged using syslog level number or name (*-n 1* or *-n alert* prevents all messages except emergency) All levels are still written to **/proc/kmsg**, so syslogd can still be used to control exactly where kernel messages appear. When the *-n* option is used, dmesg will not print or clear the kernel ring buffer.

- "dmesg Explained" article from Linux Gazette website explains **dmesg** lines
- *http://www.tldp.org/LDP/LG/issue59/nazario.html*

hald - Hardware Abstraction Layer (HAL) daemon provides all applications with data about current hardware.
dbus - Desktop Bus - Inter-Process Communication (IPC) system
- Allows processes to communicate with each other provides notification system for HAL events
- Can start services for an application's needs- applications register with dbus daemon to participate.
- Can send system wide alerts such as "new hardware detected" and "print queue modified."