Class 2: more systemd, udev and device trees

ESE3005

a word on “listening sockets”

(from stackoverflow, courtesy of David M. Syzdek):

“A client socket does not listen for incoming connections, it initiates an outgoing connection to the server. The server socket listens for incoming connections.

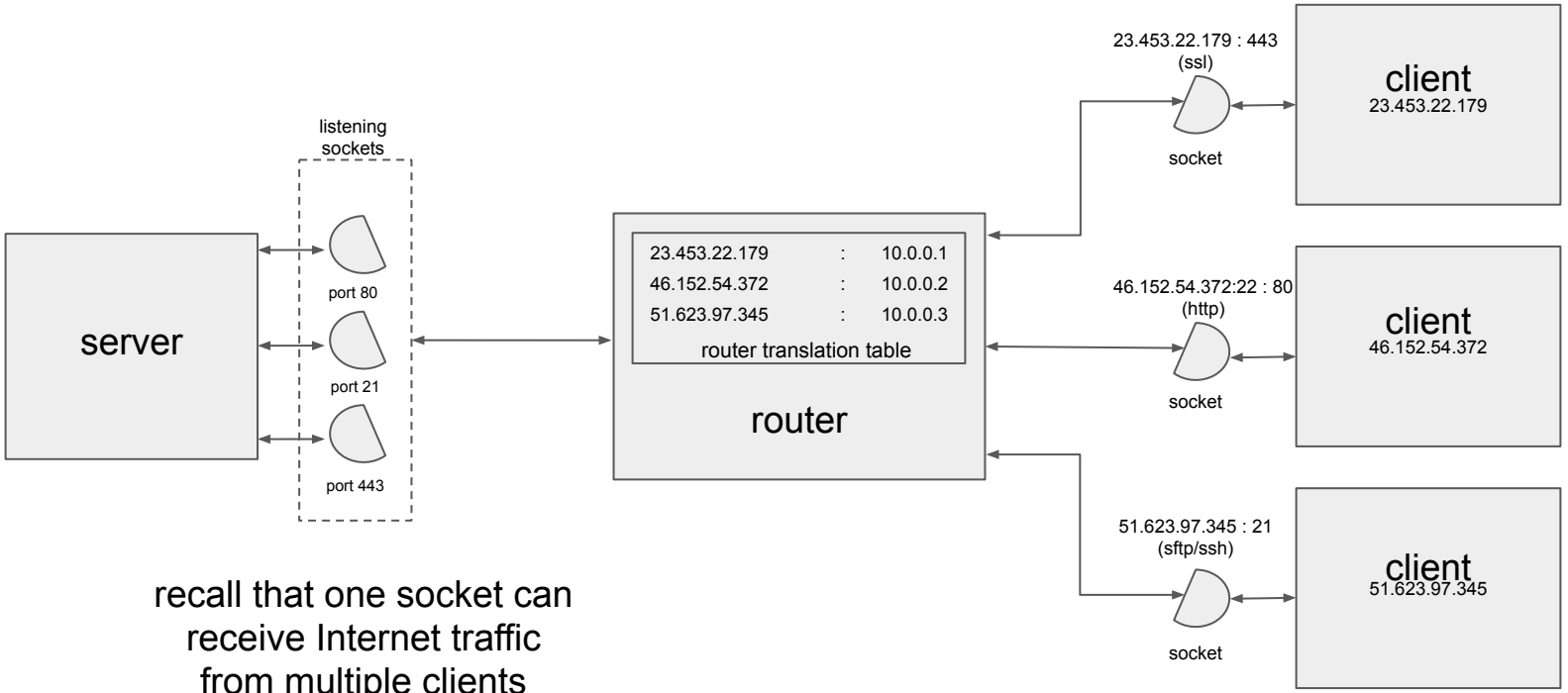
A server creates a socket, binds the socket to an IP address and port number (for TCP and UDP), and then listens for incoming connections. When a client connects to the server, a new socket is created for communication with the client (TCP only). A polling mechanism is used to determine if any activity has occurred on any of the open sockets.

A client creates a socket and connects to a remote IP address and port number (for TCP and UDP). A polling mechanism can be used (`select()`, `poll()`, `epoll()`, etc) to monitor the socket for information from the server without blocking the thread.

In the case that the client is behind a router which provides NAT (network address translation), the router re-writes the address of the client to match the router's public IP address. When the server responds, the router changes its public IP address back into the client's IP address. The router keeps a table of the active connections that it is translating so that it can map the server's responses to the correct client.”

there are a few things going on here

- first is the system-level operations of the network, amongst server, router, client (and the associated sockets)
- second are the Unix-like system calls made to kernel (on either the server, router or client machines), in order to accomplish various tasks
- various aspects of the TCP/IP or UDP protocols
- one of the chief initial operations of *systemd* is to create a number of *listening socket*s for all the services that require it; note that systemd was developed initially under RedHat Linux, which is used predominantly on server platforms; your Beaglebone, is unlikely to require many listening sockets, although it is important to keep in mind that the Beaglebone Black is configured to be a web server by default under Debian Linux from beagleboard.org



recall that one socket can receive Internet traffic from multiple clients

systemd system and start-up features

the systemd system and service manager provides the following main features:

- *Socket-based activation* — At boot time, systemd creates listening sockets for all system services that support this type of activation, and passes the sockets to these services as soon as they are started. This not only allows systemd to start services in parallel, but also makes it possible to restart a service without losing any message sent to it while it is unavailable: the corresponding socket remains accessible and all messages are queued. Systemd uses *socket units* for socket-based activation.
- *Bus-based activation* — System services that use D-Bus for inter-process communication can be started on-demand the first time a client application attempts to communicate with them. Systemd uses *D-Bus service files* for bus-based activation.
- *Device-based activation* — System services that support device-based activation can be started on-demand when a particular type of hardware is plugged in or becomes available. Systemd uses *device units* for device-based activation.
- *Path-based activation* — System services that support path-based activation can be started on-demand when a particular file or directory changes its state. Systemd uses *path units* for path-based activation.

systemd system and start-up features (cont'd)

- *Mount and automount point management* — Systemd monitors and manages mount and automount points. Systemd uses *mount units* for mount points and *automount units* for automount points.
- *Aggressive parallelization* — Because of the use of socket-based activation, systemd can start system services in parallel as soon as all listening sockets are in place. In combination with system services that support on-demand activation, parallel activation significantly reduces the time required to boot the system.
- *Transactional unit activation logic* — Before activating or deactivating a unit, systemd calculates its dependencies, creates a temporary transaction, and verifies that this transaction is consistent. If a transaction is inconsistent, systemd automatically attempts to correct it and remove non-essential jobs from it before reporting an error.
- *Backwards compatibility with SysV init* — Systemd supports SysV init scripts as described in the *Linux Standard Base Core Specification*, which eases the upgrade path to systemd service units.

. source: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/system_administrators_guide/chap-managing_services_with_systemd

systemd units

Systemd introduces the concept of *systemd units*. These units are represented by unit configuration files located in one of the directories listed in Table 10.2, “Systemd Unit Files Locations”, and encapsulate information about system services, listening sockets, and other objects that are relevant to the init system. For a complete list of available systemd unit types, see Table 10.1, “Available systemd Unit Types”.

Table 10.1. Available systemd Unit Types

Unit Type	File Extension	Description
Service unit	`.service`	A system service.
Target unit	`.target`	A group of systemd units.
Automount unit	`.automount`	A file system automount point.
Device unit	`.device`	A device file recognized by the kernel.
Mount unit	`.mount`	A file system mount point.
Path unit	`.path`	A file or directory in a file system.
Scope unit	`.scope`	An externally created process.
Slice unit	`.slice`	A group of hierarchically organized units that manage system processes.
Snapshot unit	`.snapshot`	A saved state of the systemd manager.
Socket unit	`.socket`	An inter-process communication socket.
Swap unit	`.swap`	A swap device or a swap file.
Timer unit	`.timer`	A systemd timer.

Table 10.2. Systemd Unit Files Locations

Directory	Description
`/usr/lib/systemd/system/`	Systemd unit files distributed with installed RPM packages.
`/run/systemd/system/`	Systemd unit files created at run time. This directory takes precedence over the directory with installed service unit files.
`/etc/systemd/system/`	Systemd unit files created by `systemctl enable` as well as unit files added for extending a service. This directory takes precedence over the directory with runtime unit files.

relationship with *SysV*

- under SysV, the system operates under various *runlevels*.
  - for example a runlevel of 0 corresponds to “halt the system (shutdown)”, a runlevel of 1 corresponds to *single-user mode*, while runlevels 2-5 correspond to *multi-user mode*

- with systemd, these runlevels have been replaced by *target units*; you can determine your boards current default target as follows:

```
$ systemctl get-default
```

TARGET NAMES	SysV runlevel	description
poweroff.target	0	halt the system
rescue.target	1, S	single-user mode; admin functions like checking filesystem
multi-user.target	2-4	regular multi-user mode with no windowing display
graphical.target	5	regular multi-user mode with windowing display
reboot.target	6	reboot the system
emergency.target		emergency shell on the main console

try this:

$ systemctl list-units --type=target

Creating a *systemd* service (VERY USEFUL!)

source: https://www.linode.com/docs/quick-answers/linux/start-service-at-boot/

1. create an executable C program (or bash script) and copy it to /usr/bin (or create a link to it there)
2. create a systemd unit file (like the one provided in the corresponding Week 2 github folder)
3. copy the unit file to /etc/systemd/system/ and make it have “644” permissions using chmod

Creating a *systemd* service

to start the service:

```
$ sudo systemctl start myservice
```

to check on the status of the service:

```
$ sudo systemctl status myservice
```

to stop service:

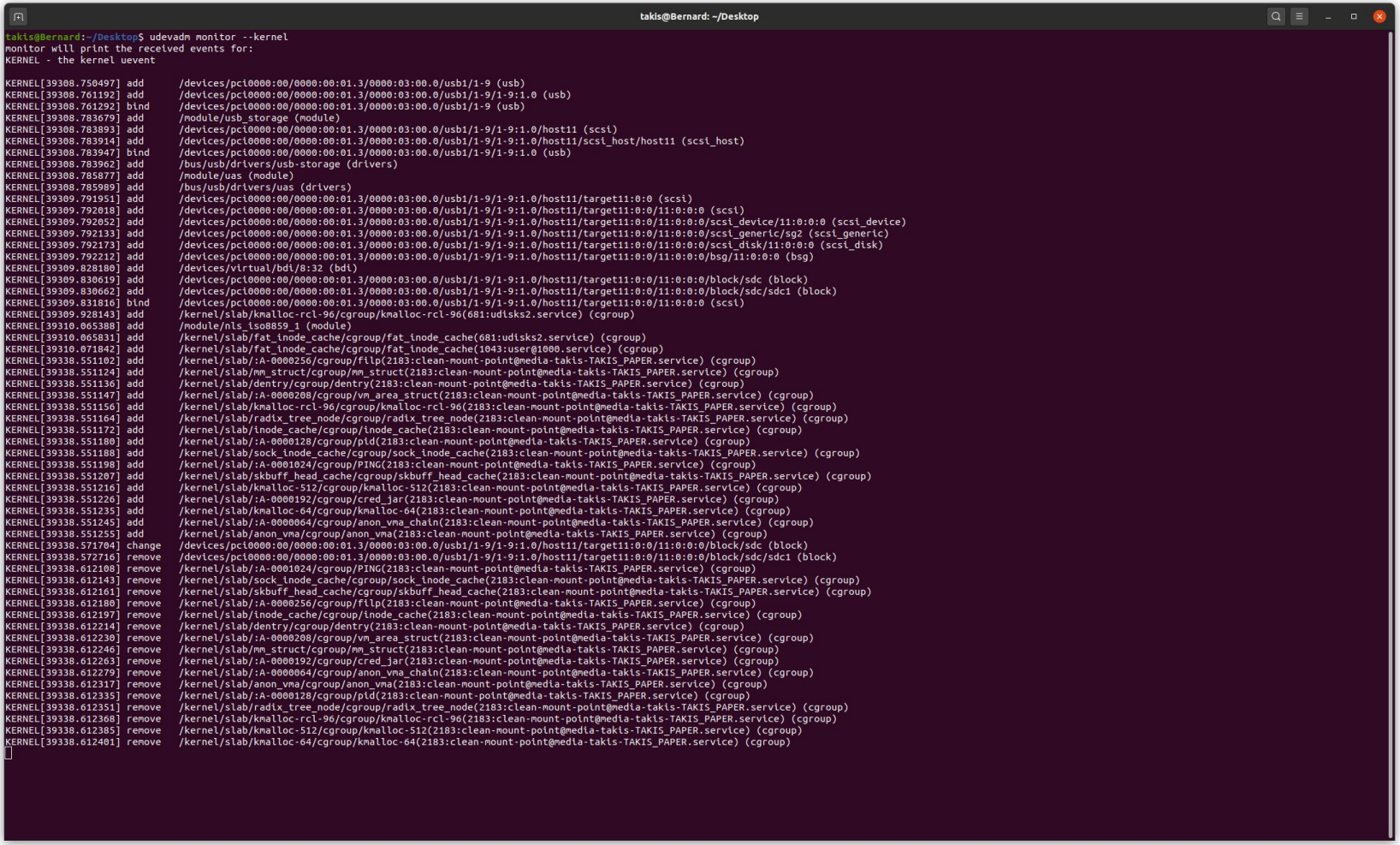
```
$ sudo systemctl stop myservice
```

to have it start automatically at boot:

```
$ sudo systemctl enable myservice
```

udev

- *udev rules* allow some control over devices on your system from within user space, i.e., without requiring root access
  - using root access in Linux is discouraged because of the potential for harm to the filesystem if mistakes are made; it is, therefore, better to do your work on the system as a regular user as much as possible
- udev was designed to make possible “hot pluggable” devices such as USB devices for video, sound or storage
- access to udev is furnished through the udev admin tool; to get a feel for this, try this command on your host machine:
  - $ udevadm monitor --kernel
  - now, plug in a flash drive to a USB port; what do you see?
  - now, eject the flash drive from the USB port, what do you see?
- **further reading:** https://wiki.debian.org/udev


takis@Bernard: ~/Desktop
takis@Bernard:~/Desktop$ udevadm monitor --kernel
monitor will print the received events for:
KERNEL - the kernel uevent
KERNEL[39308.750497] add /devices/pci0000:00/0000:00:01.3/0000:03:00.0/usb1/1-9 (usb)
KERNEL[39308.761192] add /devices/pci0000:00/0000:00:01.3/0000:03:00.0/usb1/1-9/1-9:1.0 (usb)
KERNEL[39308.761292] bind /devices/pci0000:00/0000:00:01.3/0000:03:00.0/usb1/1-9 (usb)
KERNEL[39308.783679] add /module/usb_storage (module)
KERNEL[39308.783893] add /devices/pci0000:00/0000:00:01.3/0000:03:00.0/usb1/1-9/1-9:1.0/host11 (scsi)
KERNEL[39308.783914] add /devices/pci0000:00/0000:00:01.3/0000:03:00.0/usb1/1-9/1-9:1.0/host11/scsi_host/host11 (scsi_host)
KERNEL[39308.783947] bind /devices/pci0000:00/0000:00:01.3/0000:03:00.0/usb1/1-9/1-9:1.0 (usb)
KERNEL[39308.783962] add /bus/usb/drivers/usb-storage (drivers)
KERNEL[39308.785877] add /module/uas (module)
KERNEL[39308.785989] add /bus/usb/drivers/uas (drivers)
KERNEL[39309.791951] add /devices/pci0000:00/0000:00:01.3/0000:03:00.0/usb1/1-9/1-9:1.0/host11/target11:0:0 (scsi)
KERNEL[39309.792018] add /devices/pci0000:00/0000:00:01.3/0000:03:00.0/usb1/1-9/1-9:1.0/host11/target11:0:0/11:0:0:0 (scsi)
KERNEL[39309.792052] add /devices/pci0000:00/0000:00:01.3/0000:03:00.0/usb1/1-9/1-9:1.0/host11/target11:0:0/11:0:0:0/scsi_device/11:0:0:0 (scsi_device)
KERNEL[39309.792133] add /devices/pci0000:00/0000:00:01.3/0000:03:00.0/usb1/1-9/1-9:1.0/host11/target11:0:0/11:0:0:0/scsi_generic/sg2 (scsi_generic)
KERNEL[39309.792173] add /devices/pci0000:00/0000:00:01.3/0000:03:00.0/usb1/1-9/1-9:1.0/host11/target11:0:0/11:0:0:0/scsi_disk/11:0:0:0 (scsi_disk)
KERNEL[39309.792212] add /devices/pci0000:00/0000:00:01.3/0000:03:00.0/usb1/1-9/1-9:1.0/host11/target11:0:0/11:0:0:0/bsg/11:0:0:0 (bsg)
KERNEL[39309.828180] add /devices/virtual/bdi/8:32 (bdi)
KERNEL[39309.830619] add /devices/pci0000:00/0000:00:01.3/0000:03:00.0/usb1/1-9/1-9:1.0/host11/target11:0:0/11:0:0:0/block/sdc (block)
KERNEL[39309.830662] add /devices/pci0000:00/0000:00:01.3/0000:03:00.0/usb1/1-9/1-9:1.0/host11/target11:0:0/11:0:0:0/block/sdc/sdc1 (block)
KERNEL[39309.831816] bind /devices/pci0000:00/0000:00:01.3/0000:03:00.0/usb1/1-9/1-9:1.0/host11/target11:0:0/11:0:0:0 (scsi)
KERNEL[39309.928143] add /kernel/slab/kmalloc-rcl-96/cgroup/kmalloc-rcl-96(681:udisks2.service) (cgroup)
KERNEL[39310.065388] add /module/nls_iso8859_1 (module)
KERNEL[39310.065831] add /kernel/slab/fat_inode_cache/cgroup/fat_inode_cache(681:udisks2.service) (cgroup)
KERNEL[39310.071842] add /kernel/slab/fat_inode_cache/cgroup/fat_inode_cache(1043:user@1000.service) (cgroup)
KERNEL[39338.551102] add /kernel/slab/:A-0000256/cgroup/filp(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.551124] add /kernel/slab/mm_struct/cgroup/mm_struct(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.551136] add /kernel/slab/dentry/cgroup/dentry(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.551147] add /kernel/slab/:A-0000208/cgroup/vm_area_struct(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.551156] add /kernel/slab/kmalloc-rcl-96/cgroup/kmalloc-rcl-96(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.551164] add /kernel/slab/radix_tree_node/cgroup/radix_tree_node(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.551172] add /kernel/slab/inode_cache/cgroup/inode_cache(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.551180] add /kernel/slab/:A-0000128/cgroup/pid(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.551188] add /kernel/slab/sock_inode_cache/cgroup/sock_inode_cache(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.551198] add /kernel/slab/:A-0001024/cgroup/PING(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.551207] add /kernel/slab/skbuff_head_cache/cgroup/skbuff_head_cache(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.551216] add /kernel/slab/kmalloc-512/cgroup/kmalloc-512(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.551226] add /kernel/slab/:A-0000192/cgroup/cred_jar(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.551235] add /kernel/slab/kmalloc-64/cgroup/kmalloc-64(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.551245] add /kernel/slab/:A-0000064/cgroup/anon_vma_chain(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.551255] add /kernel/slab/anon_vma/cgroup/anon_vma(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.571704] change /devices/pci0000:00/0000:00:01.3/0000:03:00.0/usb1/1-9/1-9:1.0/host11/target11:0:0/11:0:0:0/block/sdc (block)
KERNEL[39338.572716] remove /devices/pci0000:00/0000:00:01.3/0000:03:00.0/usb1/1-9/1-9:1.0/host11/target11:0:0/11:0:0:0/block/sdc/sdc1 (block)
KERNEL[39338.612108] remove /kernel/slab/:A-0001024/cgroup/PING(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.612143] remove /kernel/slab/sock_inode_cache/cgroup/sock_inode_cache(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.612161] remove /kernel/slab/skbuff_head_cache/cgroup/skbuff_head_cache(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.612180] remove /kernel/slab/:A-0000256/cgroup/filp(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.612197] remove /kernel/slab/inode_cache/cgroup/inode_cache(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.612214] remove /kernel/slab/dentry/cgroup/dentry(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.612230] remove /kernel/slab/:A-0000208/cgroup/vm_area_struct(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.612246] remove /kernel/slab/mm_struct/cgroup/mm_struct(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.612263] remove /kernel/slab/:A-0000192/cgroup/cred_jar(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.612279] remove /kernel/slab/:A-0000064/cgroup/anon_vma_chain(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.612317] remove /kernel/slab/anon_vma/cgroup/anon_vma(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.612335] remove /kernel/slab/:A-0000128/cgroup/pid(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.612351] remove /kernel/slab/radix_tree_node/cgroup/radix_tree_node(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.612368] remove /kernel/slab/kmalloc-rcl-96/cgroup/kmalloc-rcl-96(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.612385] remove /kernel/slab/kmalloc-512/cgroup/kmalloc-512(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)
KERNEL[39338.612401] remove /kernel/slab/kmalloc-64/cgroup/kmalloc-64(2183:clean-mount-point@media-takis-TAKIS_PAPER.service) (cgroup)

udev cont’d

from the debian wiki site:

- udev allows for rules that specify what name is given to a device, regardless of which port it is plugged into. For example, a rule to always mount a hard drive with manufacturer "iRiver" and device code "ABC" as /dev/iriver is possible. This consistent naming of devices guarantees that scripts dependent on a specific device's existence work.
- the udev system is composed of some kernel services and the **udevd** daemon. The kernel informs the udevd daemon when certain events happen. The udevd daemon is configured to respond to events with corresponding actions. The event information comes from the kernel - the actions happen in userspace. The responses to the events are configurable in "rules".
- the userspace udev functionality is implemented by the systemd-udevd.service Its config file is in /etc/udev/udev.conf. The rules files (which amount to more configuration for udevd) are taken from **/run/udev/rules.d**, **/etc/udev/rules.d** or **/lib/udev/rules.d**. Packages install rules in /lib/udev/rules.d), **while the /etc and /run locations provide a facility for the administrator to override the behavior of a package-provided rule**. If a file with the same name is present in more than one of these directories then the latter(s) file will be ignored. Files in there are parsed in alpha order, as long as the name ends with ".rules". When the config file or rules files are changed, the udevadm program should be used to instruct systemd-udevd to reload the rules (see below). [the numbered prefixes of the file names therefore give an implicit priority to the rules]

udev cont’d

The times when udevd (the udev daemon or background service) is active are:

1. at startup, it parses all the config files and rule files and builds a rules database in memory.
2. When an event happens, it checks its rule database and performs the appropriate actions.

Rules for rules:

1. rules are all on one line (lines can be broken with \ just before newline)
2. rules consist of "matches" and "actions"
3. matches and actions are "key" "operator" "value" triplets
4. matches have == or != for operator
5. actions have = (assignment) for operator
6. matches check one or more attributes of the event to see if the action will be applied
7. actions specify what will happen
8. example match: BUS=="usb"
9. example action: NAME="mydev"
10. example rule: KERNEL=="sd*[0-9]|dasd*[0-9]", ENV{ID_SERIAL}=="?*", \
    SYMLINK+="disk/by-id/$env{ID_BUS}-$env{ID_SERIAL}-part%n"
11. all matching rules will fire
12. earlier rules have precedence over later rules - so put your customizations early in the rules.d file list
13. actions like key="value" override
14. actions like key+="value" add to the actions that are executed, eg SYMLINK+="foo" means "in addition to any other symlinks you were going to make for this event, also make one called foo"

udev cont’d

Rules for rule sets:

1. All the rules are in one big rule space, although they are divided into several files.
2. The only organization in the rule space is the ability to set labels, and then to skip a bunch of rules during "match this event to rules" time by jumping forward with a GOTO action.
3. there is one other rule type called a label: eg LABEL="persistent_storage_end" These are used by regular rules that have "GOTO" actions, eg:
   `ACTION!="add", GOTO="persistent_storage_end"`
   Note that in this rule, the term ACTION is an attribute of an event and is being used as a condition for deciding if the GOTO action will be triggered.
4. It is polite to keep GOTOs to jump within a file (or you will have to worry about reordering the files)
5. Don't jump backwards to a label (didn't try it, but imagine it might end in an infinite loop? Maybe the udev code checks for that - but if it's going to be ignored (at best) why bother?)
6. You can set variables in ENV space in earlier rules and refer to them with later rules
7. The facility for dynamic rule creation exists (example: see z45_persistent-net-generator.rules)

Flattened Device Tree on Embedded Linux Systems

- according to Derek Molloy, “The flattened device tree (FDT) is a human-readable data structure that describes the hardware on a particular [platform]. The FDT is described using device-tree source (DTS) files, where a .dts file contains a board-level definition and a .dtsi file typically includes SoC-level definitions. The .dtsi files are typically included into a .dts file.” (page 272, *Exploring Beaglebone*, 2nd Ed.)

flattened device tree (DTS) format

```
/dts-v1/;

/ {
    node1 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        // hex is implied in byte arrays. no '0x' prefix is required
        a-byte-data-property = [01 23 34 56];
        child-node1 {
            first-child-property;
            second-child-property = <1>;
            a-string-property = "Hello, world";
        };
        child-node2 {
        };
    };
    node2 {
        an-empty-property;
        a-cell-property = <1 2 3 4>; /* each number (cell) is a uint32 */
        child-node1 {
        };
    };
};
```

FDT key-value pair data representations

- Text strings (null terminated) are represented with double quotes:
  - `string-property = "a string";`
- 'Cells' are 32 bit unsigned integers delimited by angle brackets:
  - `cell-property = <0xbeef 123 0xabcd1234>;`
- Binary data is delimited with square brackets:
  - `binary-property = [0x01 0x23 0x45 0x67];`
- Data of differing representations can be concatenated together using a comma:
  - `mixed-property = "a string", [0x01 0x23 0x45 0x67], <0x12345678>;`
- Commas are also used to create lists of strings:
  - `string-list = "red fish", "blue fish";`

FDT example (from elinux.org)

Consider the following imaginary machine (loosely based on ARM Versatile), manufactured by "Acme" and named "Coyote's Revenge":

- One 32-bit ARM CPU
- processor local bus attached to memory mapped serial port, spi bus controller, i2c controller, interrupt controller, and external bus bridge
- 256MB of SDRAM based at 0
- 2 Serial ports based at 0x101F1000 and 0x101F2000
- GPIO controller based at 0x101F3000
- SPI controller based at 0x10170000 with following devices
    - MMC slot with SS pin attached to GPIO #1
- External bus bridge with following devices
    - SMC SMC91111 Ethernet device attached to external bus based at 0x10100000
    - i2c controller based at 0x10160000 with following devices
        - Maxim DS1338 real time clock. Responds to slave address 1101000 (0x58)
    - 64MB of NOR flash based at 0x30000000

Initial structure

- The first step is to lay down a skeleton structure for the machine. This is the bare minimum structure required for a valid device tree. At this stage you want to uniquely identify the machine:

```
/dts-v1/;



/ {
    compatible = "acme,coyotes-revenge";
};
```

Describe each of the CPUs

- A container node named "cpus" is added with a child node for each CPU. In this case the system is a dual-core Cortex A9 system from ARM.

```
/dts-v1/;
/ {
    compatible = "acme,coyotes-revenge";
    cpus {
        cpu@0 {
            compatible = "arm,cortex-a9";
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
        };
    };
};
```

Node Names

It is worth taking a moment to talk about naming conventions. Every node must have a name in the form `<name>[@<unit-address>]`.

`<name>` is a simple ascii string and can be up to 31 characters in length. In general, nodes are named according to what kind of device it represents. ie. A node for a 3com Ethernet adapter would be use the name `ethernet`, not `3com509`.

The unit-address is included if the node describes a device with an address. In general, the unit address is the primary address used to access the device, and is listed in the node's `reg` property

Devices

```
/dts-v1/;
/ {
    compatible = "acme,coyotes-revenge";
    cpus {
        cpu@0 {
            compatible = "arm,cortex-a9";
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
        };
    };
    serial@101F0000 {
        compatible = "arm,pl011";
    };
    serial@101F2000 {
        compatible = "arm,pl011";
    };
    gpio@101F3000 {
        compatible = "arm,pl061";
    };
    interrupt-controller@10140000 {
        compatible = "arm,pl190";
    };

    spi@10115000 {
        compatible = "arm,pl022";
    };

    external-bus {
        ethernet@0,0 {
            compatible = "smc,smc91c111";
        };

        i2c@1,0 {
            compatible = "acme,a1234-i2c-bus";
            rtc@58 {
                compatible = "maxim,ds1338";
            };
        };

        flash@2,0 {
            compatible = "samsung,k8f1315ebm", "cfi-flash";
        };
    };
};
```

Devices Cont’d

Some things to notice in this tree:

- Every device node has a `compatible` property.
- The flash node has 2 strings in the compatible property.
- As mentioned earlier, node names reflect the type of device, not the particular model.

Understanding the `compatible` Property

Every node in the tree that represents a device is required to have the `compatible` property. `compatible` is the key an operating system uses to decide which device driver to bind to a device.

`compatible` is a list of strings. The first string in the list specifies the exact device that the node represents in the form `"<manufacturer>,<model>"`. The following strings represent other devices that the device is *compatible* with.

For example, the Freescale MPC8349 System on Chip (SoC) has a serial device which implements the National Semiconductor ns16550 register interface. The compatible property for the MPC8349 serial device should therefore be: `compatible = "fsl,mpc8349-uart", "ns16550"`. In this case, `fsl,mpc8349-uart` specifies the exact device, and `ns16550` states that it is register-level compatible with a National Semiconductor 16550 UART.

How Addressing Works

Devices that are addressable use the following properties to encode address information into the device tree:

- `reg`
- `#address-cells`
- `#size-cells`

Each addressable device gets a `reg` which is a list of tuples in the form `reg = <address1 length1 [address2 length2] [address3 length3] ... >`. Each tuple represents an address range used by the device. Each address value is a list of one or more 32 bit integers called *cells*. Similarly, the length value can either be a list of cells, or empty.

Since both the address and length fields are variable of variable size, the `#address-cells` and `#size-cells` properties in the parent node are used to state how many cells are in each field. Or in other words, interpreting a reg property correctly requires the parent node's #address-cells and #size-cells values. To see how this all works, lets add the addressing properties to the sample device tree, starting with the CPUs.

CPU addressing

The CPU nodes represent the simplest case when talking about addressing. Each CPU is assigned a single unique ID, and there is no size associated with CPU ids:

```
   cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        cpu@0 {
            compatible = "arm,cortex-a9";
            reg = <0>;
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
            reg = <1>;
        };
    };
```

In the `cpus` node, `#address-cells` is set to 1, and `#size-cells` is set to 0. This means that child `reg` values are a single uint32 that represent the address with no size field. In this case, the two cpus are assigned addresses 0 and 1. `#size-cells` is 0 for cpu nodes because each cpu is only assigned a single address.

You'll also notice that the `reg` value matches the value in the node name. By convention, if a node has a `reg` property, then the node name must include the unit-address, which is the first address value in the `reg` property.

Memory Mapped Devices

- Instead of single address values like found in the cpu nodes, a memory mapped device is assigned a range of addresses that it will respond to. `#size-cells` is used to state how large the length field is in each child `reg` tuple. In the following example, each address value is 1 cell (32 bits), and each length value is also 1 cell, which is typical on 32 bit systems.
- Each device is assigned a base address, and the size of the region it is assigned. The GPIO device address in this example is assigned two address ranges; 0x101f3000...0x101f3fff and 0x101f4000..0x101f400f.

```
/dts-v1/;

/ {
    #address-cells = <1>;
    #size-cells = <1>;
    ...
    serial@101f0000 {
        compatible = "arm,pl011";
        reg = <0x101f0000 0x1000 >;
    };
    serial@101f2000 {
        compatible = "arm,pl011";
        reg = <0x101f2000 0x1000 >;
    };
```

```
    gpio@101f3000 {
        compatible = "arm,pl061";
        reg = <0x101f3000 0x1000
               0x101f4000 0x0010>;
    };
    interrupt-controller@10140000 {
        compatible = "arm,pl190";
        reg = <0x10140000 0x1000 >;
    };
    spi@10115000 {
        compatible = "arm,pl022";
        reg = <0x10115000 0x1000 >;
    };
    ...

};
```