Linux Kernel Programming

Linux Kernel programming

Agenda

- Storage Strategy
- Start-up init program
- Module parameters
- Running kernel module on Qemu, UML and RPI
- Procfs, Sysfs, Debugfs and seqfile
- timers

Storage Strategy

- Flash memory has become much less expensive over the past few years as storage capacities have increased from tens of megabytes to tens of gigabytes.

- The Flash memory uses a Linux memory technology device layer, MTD.

  drivers/mtd/

- The flash technology with decide choice of filesystem.

- The different storage choice available are
  - Unmanaged Flash

    NOR

    NAND
  - Managed Flash

    SSD/MMC/SD CARD etc

Storage Strategy - NOR

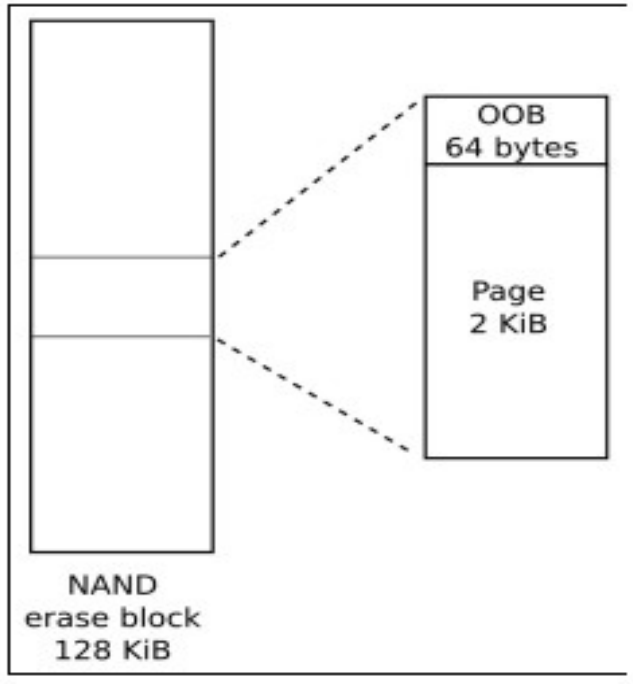
- Nor Flash is expensive but reliable and can be mapped into the CPU address space, which allows you to execute code directly from flash. NOR flash chips are low capacity ranging from few mega bytes to giga bytes or so.
- Erase is done in Blocks normally 128 KB. Erase cycle is Normally required before write. Erase cycle would turn all bits to 1.
- Erase cycle damages the memory cells slightly and after a number of cycles the erase block becomes unreliable and cannot be used any more.
- Maximum number of erase cycle would be mentioned in datasheet.
- The kernel and even root filesystem can also be located in flash memory avoiding the need to copying them into RAM, and thus creating devices with small memory foot prints. The techniquq is known as execute in place (XIP).
- The standard register-level interfaces for NOR flash chips called the common flash interface or CFI. This follows JEDEC standards.

Storage Strategy - NAND

- Nand Flash memory is much cheaper than NOR and is available in higher capacities in range of tens of megabytes to tens of gigabytes. How , it needs lot of hardware and software support to turn it into a useful storage medium.
- NAND flash is cheaper than NOR flash and has a higher capacity.
- The NAND flash stores one bit per memory cell known as SLC or single level cell (First generation) or two bits per cells in Multi level cell (MLC) or three bits per cell in tri level cell (TLC).
- As the number of bits per cells increased complexity of hardware has increased and reliability reduced.
- NAND flash erase block size in the range 16KB to 512KB. Erasing sets each bits to 1.
- NAND flash can be read and written in pages (2KB to 4 KB). Since they cannot be accessed byte-by-byte they cannot be mapped into address space and so code and data have to copied into RAM before they can be accessed.
- Data transfer to and from the chips are prone to bits flips, which can be detected and corrected using error correction codes (ECCs).
- SLC chips generally use a simple hamming code. This can correct single bit error per page.
- The MLC uses BCH code this can correct 8 bits error per page.

Storage Strategy - NAND

- The ECCs have to be stored somewhere and so there is an extra area of memory per page known as the out-of-band (OOB) area, or the spare area.



Storage Strategy - NAND

- During production, the manufacturer tests all the blocks and marks any that fail by setting a flag in the OOB area of the each page in the block.
- The NAND flash driver should detect this and mark it as bad.
- After space has been taken in the OOB area for a bad block flag and ECC bytes, there are still some bytes left. Some flash filesystem make use of these free bytes to store filesystem metadata.
- NAND flash chips requires a NAND flash controller which is part of SOC.
- The NAND flash standard register level interface is know as open nand flash interface or ONFi.

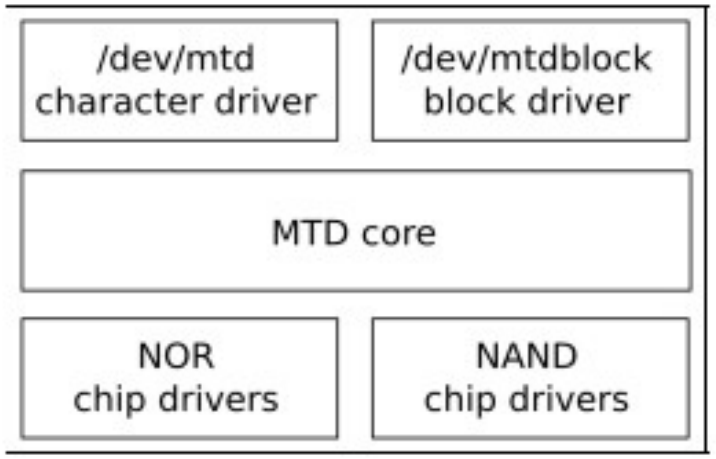
Storage Strategy – Managed Flash

- Managed Flash using NAND flash internally and handles all hardware complexities.
- The most common are SD card, MMC , eMMC, SSD.

Storage Strategy – Bootloader

- Bootloader like U-Boot has drivers to handle NOR, Nand and Managed Flash.

Storage Strategy – Kernel



- MTD partitioning

```
mtdparts=<mtddef>[;<mtddef]
<mtddef>  := <mtd-id>:<partdef>[,<partdef>]
<mtd-id>  := unique name for the chip
<partdef> := <size>[@<offset>][<name>][ro][lk]
<size>    := size of partition OR "-" to denote all remaining
             space
<offset>  := offset to the start of the partition; leave blank
             to follow the previous partition without any gap
<name>    := '(' NAME ')'
```

```
mtdparts=:512k(SPL)ro,780k(U-Boot)ro,128k(U-BootEnv),
4m(Kernel),-(Filesystem)
```

Storage Strategy – Kernel

- MTD partitioning info on device

```
# cat /proc/mtd
dev:    size   erasesize   name
mtd0: 00080000 00020000  "SPL"
mtd1: 000C3000 00020000  "U-Boot"
mtd2: 00020000 00020000  "U-BootEnv"
mtd3: 00400000 00020000  "Kernel"
mtd4: 07A9D000 00020000  "Filesystem"
```

```
# mtdinfo /dev/mtd0
mtd0
Name:                           SPL
Type:                           nand
Eraseblock size:                131072 bytes, 128.0 KiB
Amount of eraseblocks:          4 (524288 bytes, 512.0 KiB)
Minimum input/output unit size: 2048 bytes
Sub-page size:                  512 bytes
OOB size:                       64 bytes
Character device major/minor:   90:0
Bad blocks are allowed:         true
Device is writable:             false
```

Storage Strategy – Kernel

- MTD partitioning information in device tree

```
nand@0,0 {
  #address-cells = <1>;
  #size-cells = <1>;
  partition@0 {
    label = "SPL";
    reg = <0 0x80000>;
  };
  partition@80000 {
    label = "U-Boot";
    reg = <0x80000 0xc3000>;
  };
  partition@143000 {
    label = "U-BootEnv";
    reg = <0x143000 0x20000>;
  };
  partition@163000 {
    label = "Kernel";
    reg = <0x163000 0x400000>;
  };
  partition@563000 {
    label = "Filesystem";
    reg = <0x563000 0x7a9d000>;
  };
};
```

Storage Strategy – Kernel

- MTD partitioning information in device tree

```
static struct mtd_partition omap3beagle_nand_partitions[] = {
  {
    .name           = "X-Loader",
    .offset         = 0,
    .size           = 4 * NAND_BLOCK_SIZE,
    .mask_flags     = MTD_WRITEABLE,    /* force read-only */
  },
  {
    .name           = "U-Boot",
    .offset         = 0x80000;
    .size           = 15 * NAND_BLOCK_SIZE,
    .mask_flags     = MTD_WRITEABLE,    /* force read-only */
  },
  {
    .name           = "U-Boot Env",
    .offset         = 0x260000;
    .size           = 1 * NAND_BLOCK_SIZE,
  },
  {
    .name           = "Kernel",
    .offset         = 0x280000;
    .size           = 32 * NAND_BLOCK_SIZE,
  },
  {
    .name           = "File System",
    .offset         = 0x680000;
    .size           = MTDPART_SIZ_FULL,
  },
};
```

Storage Strategy – Kernel

- NAND flash can be simulated using nandsim

mtd/nand/raw/nandsim.c

Storage Strategy – Filesystem

- JFFS2 (Journaling Flash File system 2): This was the first flash filesystem for Linux, and still in use today. It works for NOR and NAND memory, but is slow during mount.
- YAFFS2 (Yet Another Flash File system 2): This is similar to JFFS2, but specifically for NAND Flash memory. It is adopted by Google as the preferred filesystem on Android devices.
- UBIFS (unsorted Block Image file system): This works in conjunction with UBI block driver and create a reliable flash filesystem. IT works well with both NOR and NAND.
- Squashfs : This is read only filesystem.

Storage Strategy – Filesystem choices

- **Permanent**, **read-write data**: Runtime configuration, network parameters, passwords, data logs, and user data
- **Permanent**, **read-only data**: Programs, libraries, and configurations files that are constant, for example, the root filesystem
- **Volatile data**: Temporary storage, for example, `/tmp`

The choices for read-write storage are as follows:

- `NOR`: `UBIFS` or `JFFS2`
- `NAND`: `UBIFS`, `JFFS2`, or `YAFFS2`
- `eMMC`: `ext4` or `F2FS`

For read-only storage, you can use any of these, mounted with the `ro` attribute. Additionally, if you want to save space, you could use `squashfs`. Finally, for volatile storage, there is only one choice, `tmpfs`.

Start-up init program

- Init program is the first program to run on system
- Init program is the ancestor for all other process and this would setup system and start-up all other processes.
- The job of the init program is to take control of the system and set it running.
- Task of init program
  - Start all daemon programs and configures system parameters and other things needed to get the system into working state.
  - Optionally it launches login daemon such as getty, on terminals that allow a login shell.
  - It adopts processes that become orphaned.
  - It catches SIGCHLD and handles them thus preventing Zombie process.
  - It restarts daemons that have terminated.
  - It handles system shutdown.

Start-up init program

Metric	BusyBox init	System V init	systemd
Complexity	Low	Medium	High
Boot-up speed	Fast	Slow	Medium
Required shell	ash	ash or bash	None
Number of executables	0	4	50(*)
libc	Any	Any	glibc
Size (MiB)	0	0.1	34(*)

Start-up init program - Busybox

BusyBox has a minimal `init` program that uses a configuration file, `/etc/inittab`, to define rules to start programs at boot up and to stop them at shutdown. Usually, the actual work is done by shell scripts, which, by convention, are placed in the `/etc/init.d` directory.

`init` begins by reading `/etc/inittab`. This contains a list of programs to run, one per line, with this format:

```
<id>::<action>:<program>
```

The role of these parameter is as follows:

- `id`: This is the controlling Terminal for the command
- `action`: This is the conditions to run this command, as shown in the following paragraph
- `program`: This is the program to run

Start-up init program - Busybox

The actions are as follows:

- `sysinit`: Runs the program when `init` starts before any of the other types of actions.
- `respawn`: Runs the program and restarts it if it terminates. It is typically used to run a program as a daemon.
- `askfirst`: This is the same as `respawn`, but it prints the message *Please press Enter to activate this console* to the console, and it runs the program after *Enter* has been pressed. It is used to start an interactive shell on a Terminal without prompting for a username or password.
- `once`: Runs the program once but does not attempt to restart it if it terminates.
- `wait`: Runs the program and waits for it to complete.
- `restart`: Runs the program when `init` receives the signal `SIGHUP`, indicating that it should reload the `inittab` file.
- `ctrlaltdel`: Runs the program when `init` receives the signal, `SIGINT`, usually as a result of pressing *Ctrl* + *Alt* + *Del* on the console.

Start-up init program - Busybox

- `shutdown`: Runs the program when `init` shuts down.

Here is a small example that mounts `proc` and `sysfs` and runs a shell on a serial interface:

```
null::sysinit:/bin/mount -t proc proc /proc
null::sysinit:/bin/mount -t sysfs sysfs /sys
console::askfirst:-/bin/sh
```

For simple projects in which you want to launch a small number of daemons, and perhaps start a login shell on a serial Terminal, it is easy to write the scripts manually. This would be appropriate if you are creating a **roll your own** (**RYO**) embedded Linux. However, you will find that hand-written `init` scripts rapidly become unmaintainable as the number of things to be configured increases. They tend not to be very modular and so need updating each time a new component is added.

Start-up init program - Busybox

- `shutdown`: Runs the program when `init` shuts down.

Here is a small example that mounts `proc` and `sysfs` and runs a shell on a serial interface:

```
null::sysinit:/bin/mount -t proc proc /proc
null::sysinit:/bin/mount -t sysfs sysfs /sys
console::askfirst:-/bin/sh
```

For simple projects in which you want to launch a small number of daemons, and perhaps start a login shell on a serial Terminal, it is easy to write the scripts manually. This would be appropriate if you are creating a **roll your own** (**RYO**) embedded Linux. However, you will find that hand-written `init` scripts rapidly become unmaintainable as the number of things to be configured increases. They tend not to be very modular and so need updating each time a new component is added.

Start-up init program – System V init

- Init program was inspired by Unix System V.
- The System V init has concepts of run levels.

There are 8 runlevels numbered from 0 to 6, plus S:

- `S`: Runs startup tasks
- `0`: Halts the system
- `1` to `5`: Available for general use
- `6`: Reboots the system

Levels 1 to 5 can be used as you please. On the desktop Linux distributions, they are conventionally assigned as follows:

- `1`: Single user
- `2`: Multi-user with no network configuration
- `3`: Multi-user with network configuration
- `4`: Not used
- `5`: Multi-user with graphical login

Start-up init program – System V init

The `init` program starts the default `runlevel` given by the `initdefault` line in `/etc/inittab`. You can change the runlevel at runtime using the command `telinit [runlevel]`, which sends a message to `init`. You can find the current runlevel and the previous one using the `runlevel` command. Here is an example:

```
# runlevel
N 5
# telinit 3
INIT: Switching to runlevel: 3
# runlevel
5 3
```

Start-up init program – System V init

The format of each line in `inittab` is as follows:

```
id:runlevels:action:process
```

The fields are shown here:

- `id`: A unique identifier of up to four characters.
- `runlevels`: The runlevels for which this entry should be executed. This was left blank in the BusyBox `inittab`
- `action`: One of the keywords given in the following paragraph.
- `process`: The command to run.

The actions are the same as for BusyBox `init`: `sysinit`, `respawn`, `once`, `wait`, `restart`, `ctrlaltdel`, and `shutdown`. However, System V `init` does not have `askfirst`, which is specific to BusyBox.

As an example, this is the complete `inittab` supplied by the Yocto Project target `core-image-minimal` for the `qemuarm` machine:

```
# /etc/inittab: init(8) configuration.
# $Id: inittab,v 1.91 2002/01/25 13:35:21 miquels Exp $

# The default runlevel.
id:5:initdefault:

# Boot-time system configuration/initialization script.
# This is run first except when booting in emergency (-b) mode.
si::sysinit:/etc/init.d/rcS

# What to do in single-user mode.
~~:S:wait:/sbin/sulogin
# /etc/init.d executes the S and K scripts upon change
# of runlevel.
#
# Runlevel 0 is halt.
# Runlevel 1 is single-user.
# Runlevels 2-5 are multi-user.
```

Start-up init program – System V init

The format of each line in `inittab` is as follows:

```
id:runlevels:action:process
```

The fields are shown here:

- `id`: A unique identifier of up to four characters.
- `runlevels`: The runlevels for which this entry should be executed. This was left blank in the BusyBox `inittab`
- `action`: One of the keywords given in the following paragraph.
- `process`: The command to run.

The actions are the same as for BusyBox `init`: `sysinit`, `respawn`, `once`, `wait`, `restart`, `ctrlaltdel`, and `shutdown`. However, System V `init` does not have `askfirst`, which is specific to BusyBox.

As an example, this is the complete `inittab` supplied by the Yocto Project target `core-image-minimal` for the `qemuarm` machine:

```
# /etc/inittab: init(8) configuration.
# $Id: inittab,v 1.91 2002/01/25 13:35:21 miquels Exp $

# The default runlevel.
id:5:initdefault:

# Boot-time system configuration/initialization script.
# This is run first except when booting in emergency (-b) mode.
si::sysinit:/etc/init.d/rcS

# What to do in single-user mode.
~~:S:wait:/sbin/sulogin
# /etc/init.d executes the S and K scripts upon change
# of runlevel.
#
# Runlevel 0 is halt.
# Runlevel 1 is single-user.
# Runlevels 2-5 are multi-user.
```

Start-up init program – System V init

```
# Runlevel 6 is reboot.

l0:0:wait:/etc/init.d/rc 0
l1:1:wait:/etc/init.d/rc 1
l2:2:wait:/etc/init.d/rc 2
l3:3:wait:/etc/init.d/rc 3
l4:4:wait:/etc/init.d/rc 4
l5:5:wait:/etc/init.d/rc 5
l6:6:wait:/etc/init.d/rc 6
# Normally not reached, but fallthrough in case of emergency.
z6:6:respawn:/sbin/sulogin
AMA0:12345:respawn:/sbin/getty 115200 ttyAMA0
# /sbin/getty invocations for the runlevels.
#
# The "id" field MUST be the same as the last
# characters of the device (after "tty").
#
# Format:
#  <id>:<runlevels>:<action>:<process>
#

1:2345:respawn:/sbin/getty 38400 tty1
```

Start-up init program – System V init – init.d

Each component that needs to respond to a runlevel change has a script in `/etc/init.d` to perform the change. The script should expect two parameters: `start` and `stop`. I will give an example of this later.

The runlevel handling script, `/etc/init.d/rc`, takes the runlevel it is switching to as a parameter. For each runlevel, there is a directory named `rc<runlevel>.d`:

```
# ls -d /etc/rc*
/etc/rc0.d  /etc/rc2.d  /etc/rc4.d  /etc/rc6.d
/etc/rc1.d  /etc/rc3.d  /etc/rc5.d  /etc/rcS.d
```

There you will find a set of scripts beginning with a capital `S` followed by two digits, and you may also find scripts beginning with a capital `K`. These are the `start` and `kill` scripts. Here is an example of the scripts for runlevel 5:

```
# ls /etc/rc5.d
S01networking   S20hwclock.sh   S99rmnologin.sh S99stop-bootlogd
S15mountnfs.sh  S20syslog
```

These are in fact symbolic links back to the appropriate script in `init.d`. The `rc` script runs all the scripts beginning with a `K` first, adding the `stop` parameter , and then runs those beginning with an `S` adding the `start` parameter . Once again, the two digit code is there to impart the order in which the scripts should run.

Starting and stopping services

You can interact with the scripts in `/etc/init.d` by calling them directly. Here is an example using the `syslog` script, which controls the `syslogd` and `klogd` daemons:

```
# /etc/init.d/syslog --help
Usage: syslog { start | stop | restart }

# /etc/init.d/syslog stop
Stopping syslogd/klogd: stopped syslogd (pid 198)
stopped klogd (pid 201)
done

# /etc/init.d/syslog start
Starting syslogd/klogd: done
```

All scripts implement `start` and `stop`, and they should also implement `help`. Some implement `status` as well, which will tell you whether the service is running or not. Mainstream distributions that still use System V `init` have a command named `service` to start and stop services, which hide the details of calling the scripts directly.

Start-up init program – Systemd

`systemd`, https://www.freedesktop.org/wiki/Software/systemd/, defines itself as a *system and service manager*. The project was initiated in 2010 by Lennart Poettering and Kay Sievers to create an integrated set of tools for managing a Linux system based around an `init` daemon. It also includes device management (`udev`) and logging, among other things. `systemd` is state of the art and is still evolving rapidly. It is common on desktop and server Linux distributions and is becoming popular on embedded Linux systems too, especially on more complex devices. So, how is it better than System V `init` for embedded systems?

- The configuration is simpler and more logical (once you understand it). Rather than the sometimes convoluted shell scripts of System V `init`, systemd has unit configuration files, which are written in a well-defined format.
- There are explicit dependencies between services rather than a two digit code that merely sets the sequence in which the scripts are run.
- It is easy to set the permissions and resource limits for each service, which is important for the security.
- It can monitor services and restart them if needed.
- There are watchdogs for each service and for `systemd` itself.
- Services are started in parallel, potentially reducing boot time.

Start-up init program – Systemd

```
# systemctl  get-default
# systemctl  list-units     --type    service
# systemctl  list-units     --type    target
# systemctl  status    simpleserver
# systemctl  enable    simpleserver
# systemctl  start     simpleserver
# systemctl  stop      simpleserver
# systemctl  restart   simpleserver
```

Kernel Module parameters

As a user program does, a kernel module can accept arguments from the command line. This allows dynamically changing the behavior of the module according to the given parameters, and can avoid the developer having to indefinitely change/compile the module during a test/debug session.

In order to set this up, you should first declare the variables that will hold the values of command line arguments, and use the module_param() macro on each of these. The macro is defined in include/linux/moduleparam.h (this should be included in the code too: #include <linux/moduleparam.h>), shown as follows:

module_param(name, type, perm);

This macro contains the following elements:

- **name**: The name of the variable used as the parameter
- **type**: The parameter's type (bool, charp, byte, short, ushort, int, uint, long, ulong), where charp stands for char pointer
- **perm**: This represents the /sys/module/<module>/parameters/<param> file permissions. Some of them are S_IWUSR, S_IRUSR, S_IXUSR, S_IRGRP, S_WGRP,and S_IRUGO

Kernel Module parameters

where:

S_I is just a prefix

R: read, W: write, X: execute

USR: user, GRP: group, UGO: user, group, others

One can eventually use a | (OR operation) to set multiple permissions. If perm is 0, the file parameter in sysfs will not be created. You should use only S_IRUGO read-only parameters, which I highly recommend; by making a | (OR) with other properties, you can obtain fine-grained properties.

When using module parameters, you should use MODULE_PARM_DESC in order to describe each of them. This macro will populate the module info section with each parameter's description.

Kernel Module parameters

```
#include <linux/moduleparam.h>
[...]

static char *mystr = "hello";
static int myint = 1;
static int myarr[3] = {0, 1, 2};

module_param(myint, int, S_IRUGO);
module_param(mystr, charp, S_IRUGO);
module_param_array(myarr, int,NULL, S_IWUSR|S_IRUSR); /*  */

MODULE_PARM_DESC(myint,"this is my int variable");
MODULE_PARM_DESC(mystr,"this is my char pointer variable");
MODULE_PARM_DESC(myarr,"this is my array of int");
```

Kernel Module parameters

```
static int foo()
{
    pr_info("mystring is a string: %s\n", mystr);
    pr_info("Array elements: %d\t%d\t%d", myarr[0], myarr[1], myarr[2]);
    return myint;
}
```

To load the module and feed our parameter, we do the following:

```
# insmod hellomodule-params.ko mystring=“test" myint=15 myArray=1,2,3
```

Kernel Module Handson1

```
# sudo insmod example2.ko myint=11 mylong=22 myshort=45 myintArray=55,66
```

Kernel Module Handson2

Kernel Module Run on UML,Qemu,RPI

UML

make -C /home/test/uml/linux-5.1.16/ M=${PWD} ARCH=um

mount -t sysfs sysfs /sys

#./linux-5.1.16/vmlinux rootfstype=hostfs rootflags=/home/test/uml/rootfs_busybox/ rw mem=256M init=/linuxrc

Qemu

make -C /home/test/qemu/linux-5.1.16/ M=${PWD} ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-

Copy files to rootfs and build it

find . | cpio -o --format=newc > ../rootfs.img

cd ..

gzip -c rootfs.img > rootfs.img.gz

Kernel Module Run on UML,Qemu,RPI

```
# qemu-system-arm -M vexpress-a9 -m 512M -dtb linux-5.1.16/arch/arm/boot/dts/vexpress-v2p-ca9.dtb -kernel linux-5.1.16/arch/arm/boot/zImage -initrd rootfs.img.gz -append "root=/dev/ram  rdinit=/linuxrc"
```

Debugging Kernel Module

```
# qemu-system-arm -S -s -M vexpress-a9 -m 512M -dtb linux-5.1.16/arch/arm/boot/dts/vexpress-v2p-ca9.dtb -kernel linux-5.1.16/arch/arm/boot/zImage -initrd rootfs.img.gz -append "root=/dev/ram rdinit=/linuxrc"
```

Procfs,Debugfs, Sysfs and Seqfile

- There are three commonly used pseudo file systems in the kernel: procfs, debugfs, and sysfs.
- Procfs: The proc filesystem is a pseudo-filesystem which provides an interface to kernel data structures.
- Sysfs: The filesystem for exporting kernel objects.
- debugfs : Debugfs exists as a simple way for kernel developers to make information available to user space.

Procfs,Debugfs, Sysfs and Seqfile

They are used for data exchange between the Linux kernel and user space, but the applicable scenarios are different:

- The earliest history of procfs was originally used to interact with the kernel to obtain various information such as processors, memory, device drivers, and processes.

- Sysfs is closely tied to the kobject framework, and kobject exists for the device driver model, so sysfs is for device drivers.

- Debugfs is born from the name of the name, so it is more flexible.

Procfs,Debugfs, Sysfs and Seqfile

Their mounting methods are similar, let's do an experiment:

```
$ sudo mkdir /tmp/{proc,sys,debug}
$ sudo mount -t proc nondev /tmp/proc/
$ sudo mount -t sysfs nondev /tmp/sys/
$ sudo mount -t debugfs nondev /tmp/debug/
```

Procfs,Debugfs, Sysfs and Seqfile

The following is a brief introduction to the usage of these three file systems. Before you introduce, please write down their official documentation:

procfs — Documentation/filesystems/proc.txt

sysfs — Documentation/filesystems/sysfs.txt

debugfs — Documentation/filesystems/debugfs.txt

Debugfs

struct dentry *debugfs_create_dir(const char *name, struct dentry *parent)

struct dentry *debugfs_create_file(const char *name, umode_t mode, struct dentry *parent, void *data,const struct file_operations *fops)

Procfs

Many or most Linux users have at least heard of proc. Some of you may wonder why this folder is so important.

On the root, there is a folder titled “proc”. This folder is not really on /dev/sda1 or where ever you think the folder resides. This folder is a mount point for the procfs (Process Filesystem) which is a filesystem in memory. Many processes store information about themselves on this virtual filesystem. ProcFS also stores other system information.

It can act as a bridge connecting the user space and the kernel space. Userspace programs can use proc files to read the information exported by the kernel. Every entry in the proc file system provides some information from the kernel.

Procfs

The entry “meminfo” gives the details of the memory being used in the system.

To read the data in this entry just run

```
cat /proc/meminfo
```

Similarly the “modules” entry gives details of all the modules that are currently a part of the kernel.

```
cat /proc/modules
```

Procfs

It gives similar information as lsmod. Like this more, proc entries are there.

/proc/devices — registered character and block major numbers

/proc/iomem — on-system physical RAM and bus device addresses

/proc/ioports — on-system I/O port addresses (especially for x86 systems)

/proc/interrupts — registered interrupt request numbers

/proc/softirqs — registered soft IRQs

/proc/swaps — currently active swaps

/proc/kallsyms — running kernel symbols, including from loaded modules

/proc/partitions — currently connected block devices and their partitions

/proc/filesystems — currently active filesystem drivers

/proc/cpuinfo — information about the CPU(s) on the system

Procfs

Most proc files are read-only and only expose kernel information to user space programs.

proc files can also be used to control and modify kernel behavior on the fly. The proc files need to be writable in this case.

For example, to enable IP forwarding of iptable, one can use the command below,

echo 1 > /proc/sys/net/ipv4/ip_forward

The proc file system is also very useful when we want to debug a kernel module. While debugging we might want to know the values of various variables in the module or maybe the data that the module is handling. In such situations, we can create a proc entry for ourselves and dump whatever data we want to look into in the entry.

Procfs

The proc entry can also be used to pass data to the kernel by writing into the kernel, so there can be two kinds of proc entries.

1. An entry that only reads data from the kernel space.
2. An entry that reads as well as writes data into and from kernel space.

Procfs

You can create the directory under /proc/* using the below API.

```
struct proc_dir_entry *proc_mkdir(const char *name, struct proc_dir_entry
                                  *parent)
```

where,

`name`: The name of the directory that will be created under `/proc`.

`parent`: In case the folder needs to be created in a subfolder under `/proc` a pointer to the same is passed else it can be left as `NULL`.

Creating Procfs Entry

The creation of proc entries has undergone a considerable change in kernel version 3.10 and above. In this post, we will see one of the methods we can use in Linux kernel version 3.10. Let us see how we can create proc entries in version 3.10.

```
struct proc_dir_entry *proc_create ( const char *name, umode_t mode,
struct proc_dir_entry *parent, const struct file_operations *proc_fops )
```

sysfs

- Sysfs is a virtual filesystem exported by the kernel, similar to /proc. The files in Sysfs contain information about devices and drivers. Some files in Sysfs are even writable, for configuration and control of devices attached to the system. Sysfs is always mounted on /sys.

- The directories in Sysfs contain the hierarchy of devices, as they are attached to the computer.

- Sysfs is the commonly used method to export system information from the kernel space to the user space for specific devices. The sysfs is tied to the device driver model of the kernel. The procfs is used to export the process-specific information and the debugfs is used to use for exporting the debug information by the developer.

sysfs

Kernel Objects

The heart of the sysfs model is the kobject. Kobject is the glue that binds the sysfs and the kernel, which is represented by struct kobject and defined in <linux/kobject.h>. A struct kobject represents a kernel object, maybe a device or so, such as the things that show up as directory in the sysfs filesystem.

Kobjects are usually embedded in other structures.

It is defined as,

```
#define KOBJ_NAME_LEN 20

struct kobject {
 char *k_name;
 char name[KOBJ_NAME_LEN];
 struct kref kref;
 struct list_head entry;
 struct kobject *parent;
 struct kset *kset;
 struct kobj_type *ktype;
 struct dentry *dentry;
};
```

sysfs

Some of the important fields are:

struct kobject

|– name (Name of the kobject. Current kobject is created with this name in sysfs.)

|– parent (This is kobject's parent. When we create a directory in sysfs for the current kobject, it will create under this parent directory)

|– ktype (the type associated with a kobject)

|– kset (a group of kobjects all of which are embedded in structures of the same type)

|– sd (points to a sysfs_dirent structure that represents this kobject in sysfs.)

|– kref (provides reference counting)

It is the glue that holds much of the device model and its sysfs interface together.

So kobject is used to create kobject directory in /sys. This is enough. We will not go deep into the kobjects.

sysfs

There are several steps to creating and using sysfs.

1. Create a directory in `/sys`
2. Create Sysfs file

sysfs

Create a directory in /sys

We can use this function (`kobject_create_and_add`) to create a directory.

```
struct kobject * kobject_create_and_add ( const char * name, struct kobject
                                          * parent);
```

Where,

<name> – the name for the kobject

<parent> – the parent kobject of this kobject, if any.

If you pass `kernel_kobj` to the second argument, it will create the directory under /sys/kernel/. If you pass `firmware_kobj` to the second argument, it will create the directory under /sys/firmware/. If you pass `fs_kobj` to the second argument, it will create the directory under **/sys/fs/**. If you pass `NULL` to the second argument, it will create the directory under /sys/.

sysfs

To create a single file attribute we are going to use '`sysfs_create_file`'.

```
int sysfs_create_file ( struct kobject *  kobj, const struct attribute
* attr);
```

Where,

`kobj` - *object we're creating for.*

`attr` - attribute descriptor.

Once you have done with the sysfs file, you should delete this file using

`sysfs_remove_file`

```
void sysfs_remove_file  ( struct  kobject  *  kobj, const  struct  attribute
* attr);
```

Where,

`kobj` - *object we're creating for.*

`attr` - attribute descriptor.

Sleep

- Non-busy-wait alternatives for millisecond or longer delays

```
#include <linux/delay.h>

void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds);
```

- **msleep** and **ssleep** are not interruptible
- **msleep_interruptible** returns the remaining milliseconds

Delay

```
#include <linux/delay.h>

void ndelay(unsigned long nsecs); /* nanoseconds */
void udelay(unsigned long usecs); /* microseconds */
void mdelay(unsigned long msecs); /* milliseconds */
```

- Perform busy waiting

Copy Data to/from User space

unsigned long copy_from_user(void *to, const void __user *from,
unsigned long n)
unsigned long copy_to_user(void __user *to, const void *from,
unsigned long n)

In both cases, pointers prefixed with __user point to user space (untrusted) memory. n represents the number of bytes to copy. from represents the source address, and to is the destination address. Each of these returns the number of bytes that could not be copied. On success, the return value should be 0. Please do note that with copy_to_user(), if some data could not be copied, the function will pad the copied data to the requested size using zero bytes.

Copy Data to/from User space

When it comes to copying single and simple variables, such as char and int, but not larger data types, such as structures or arrays, the kernel offers dedicated macros in order to quickly perform the desired operation. These macros are put_user(x, ptr) and get_user(x, ptr), which are explained as follows:

put_user(x, ptr): This macro copies a variable from kernel space to user space. x represents the value to copy to user space, and ptr is the destination address in user space. The macro returns 0 on success, or -EFAULT on error. x

must be assignable to the result of dereferencing ptr. In other words, they must have (or point to) the same type.

get_user(x, ptr): This macro copies a variable from user space to kernel space, and returns 0 on success or -EFAULT on error. Please do note that x is set to 0 on error. x represents the kernel variable to store the result, and ptr is the source address in user space. The result of dereferencing ptr must be assignable to x without a cast. Guess what it means.

Delayed Exeution

Linux kernel keeps track of number of ticks since the kernel has booted in a global variable called jiffies. The number of ticks per second depends upon HZ which is defined to 100 in most of the systems, therefore jiffies is incremented every 10 ms.

Thank you