Opiframe Oy

Module Programming Basics

Kernel Basics

- Linux kernel is a monolithic kernel.
- This means that all sensitive hardware related control is handled by the kernel.
- Virtual memory is usually divided into two distinct parts, kernel space and user space.

Kernel Basics

- In monolithic kernels all controls reside inside kernel space.
- User space has basically only addressing and the stuff that user processes create there.
- This means that all operations that user processes want to do need permission from the kernel.

Kernel Basics

- The kernel itself is divided into two parts, the hardware dependent and hardware independent part.

- Hardware dependent part is also the only part that needs recompiling when you change hardware platforms.

- In kernel there are multiple distinct parts which work in unison to service the processes and control the access to and operation of hardware.

Kernel Basics

- For example we have memory manager, virtual file system, network stack, scheduler, process management, device drivers and system call interface.
- System call interface is a thin layer meant to provide services to user space.
- This layer can be architecture dependent even within the same processor family.

Kernel Basics

- The memory management subsystem is one of the most important parts of the operating system.

- Virtual memory makes the system appear to have more memory than it actually has by sharing it between competing processes as they need it.

Kernel Basics

- The memory management subsystem provides:
- Large Address Space: The operating system makes the system appear as if it has a larger amount of memory than it actually has.
- Protection: Each process has its own virtual address space.

Kernel Basics

- Memory mapping is used to map image and data files into a processes address space.
- In memory mapping, the contents of a file are linked directly into the virtual address space of a process.
- Although virtual memory allows processes to have separate (virtual) address spaces, there are times when you need processes to share memory.

Kernel Basics

- For example there could be several processes in the system running the ”bash” command shell.
- The physical memory location where the ”bash” is stored is then represented by each of the processes virtual address space.
- Thus there is no need for actual multiple copies of the ”bash” binary.

Kernel Basics

- In a virtual memory system all addresses are virtual addresses and not physical addresses.
- These virtual addresses are converted into physical addresses by the processor based on information held in a set of tables maintained by the operating system.
- To make this translation easier, virtual and physical memory are divided into handy sized chunks called pages.

Kernel Basics

- Each of these pages is given a unique number; the page frame number (PFN).
- In this paged model, a virtual address is composed of two parts; an offset and a virtual page frame number.
- Each time the processor encounters a virtual address it must extract the offset and the virtual page frame number.
- To do this the processor uses page tables.

Kernel Basics

- One way to save physical memory is to only load virtual pages that are currently being used by the executing program.
- For example, a database program may be run to query a database. In this case not all of the database needs to be loaded into memory, just those data records that are being examined.
- Linux uses a Least Recently Used (LRU) page aging technique to fairly choose pages which might be removed from the system.

Kernel Basics - VFS

- In Linux the separate file systems the system may use are not accessed by device identifiers.
- Instead they are combined into a single hierarchical tree structure that represents the file system as one whole single entity.
- Linux adds each new file system into this single file system tree as it is mounted.

Kernel Basics - VFS

- All file systems, of whatever type, are mounted onto a directory.
- This directory is known as the mount directory or mount point.
- The building blocks of VFS are superblocks, inodes and caches.
- The superblock is the container for high-level metadata about a file system.

Kernel Basics - VFS

- The superblock is a structure that exists on disk (actually, multiple places on disk for redundancy) and also in memory.
- The superblock provides the basis for dealing with the on-disk file system.
- It defines the file system's managing parameters.

Kernel Basics - VFS

- On disk, the superblock provides information to the kernel on the structure of the file system on disk.
- In memory, the superblock provides the necessary information and state to manage the active (mounted) file system.
- Because Linux supports multiple concurrent file systems mounted at the same time, each super_block structure is maintained in a list.

Kernel Basics - VFS

- Within the kernel, another management object called vfsmount provides information on mounted file systems.

- The list of these objects refers to the superblock and defines the mount point, name of the /dev device on which this file system resides, and other higher-level attachment information.

- Linux manages all objects in a file system through an object called an inode (short for index node).

Kernel Basics - VFS

- The inode consists of data and operations that describe the inode, its contents, and the variety of operations that are possible on it.
- The inode refers to the file operations that are possible on it:
- Most map directly to the system-call interfaces (for example, **open()**, **read()**, **write()**, and **flush()**)
- There is also a reference to inode-specific operations (**create**, **lookup**, **link**, **mkdir**, and so on)
- Finally, there's a structure to manage the actual data for the object that is represented by an address space object similar to processes address space.

Kernel Basics - VFS

- The hierarchical nature of a file system is managed by another object in VFS called a dentry object.
- A file system will have one root dentry (referenced in the superblock), this being the only dentry without a parent.
- All other dentries have parents, and some have children.

Kernel Basics - Processes

- The kernel is in charge of creating and destroying processes and handling their connection to the outside world (input and output).

- Communication among different processes (through signals, pipes, or interprocess communication primitives) is basic to the overall system functionality and is also handled by the kernel.

- In addition, the scheduler, which controls how processes share the CPU, is part of process management.

Kernel Basics - Processes

- More generally, the kernel's process management activity implements the abstraction of several processes on top of a single CPU or a few of them

- Each process in linux has:
- A process id (PID)
- A parent process id (PPID)
- An user id (UID)
- An effective user id (EUID)
- and similar ids for groups.

Kernel Basics - Networking

- Networking must be managed by the operating system, because most network operations are not specific to a process: incoming packets are asynchronous events.
- The packets must be collected, identified, and dispatched before a process takes care of them.
- The system is in charge of delivering data packets across program and network interfaces, and it must control the execution of programs according to their network activity.

Kernel Basics – Loadable modules

- Each piece of code that can be added to the kernel at runtime is called a *module*.
- The Linux kernel offers support for quite a few different types (or classes) of modules, including, but not limited to, device drivers.
- Each module is made up of object code (not linked into a complete executable).
- That can be dynamically linked to the running kernel by the insmod program and can be unlinked by the rmmod program.

Kernel Basics – Version numbering

- Mainline kernel is currently a three series (latest stable as of today 3.14.2, release candidate is 3.15-rc3)
- There was no major change from two to three series
- Linus Torvalds just decided that Linux would not have 2.40 kernel because that would be wrong.
- Number goes Kernel Version (changes rarely), Major Revision (changes every now and then), Minor Revision (changes often) and fourth is Corrections.

Kernel Basics – First module

- Because Linux runs on a variety of architectures and supports zillions of I/O devices, it's not feasible to permanently compile support for all possible devices into the base kernel.
- Distributions generally package a minimal kernel image and supply the rest of the functionalities in the form of kernel modules.
- During runtime, the necessary modules are dynamically loaded on demand. This feature is disabled in standard Android kernels.

Kernel Basics – First module

- Most small and medium-sized applications perform a single task from beginning to end.
- Every kernel module just registers itself in order to serve future requests, and its initialization function terminates immediately.
- In other words, the task of the module's initialization function is to prepare for later invocation of the module's functions.

Kernel Basics – First module

- It's as though the module were saying, "Here I am, and this is what I can do."
- The module's exit function (**hello_exit** in the example) gets invoked just before the module is unloaded.
- It should tell the kernel, "I'm not there anymore; don't ask me to do anything else."

Kernel Basics – First module

- This kind of approach to programming is similar to event-driven programming.
- Each and every kernel module is event-driven .
- Another major difference between event-driven applications and kernel code is in the exit function.

Kernel Basics – First module

- Whereas an application that terminates can be lazy in releasing resources or avoids clean up altogether, the exit function of a module must carefully undo everything the init function built up, or the pieces remain around until the system is rebooted.
- Another important difference between kernel programming and application programming is in how each environment handles faults.

Kernel Basics – First module

- A segmentation fault is relatively harmless during application development.
- A kernel fault kills the current process at least, if not the whole system.
- Three basic rules that aren’t really about any driver type specially should be kept in mind when creating code in kernel:

Kernel Basics – First module

- The kernel has a very small stack and your functions must share that stack with the entire kernel-space call chain.
- Often, as you look at the kernel API, you will encounter function names starting with a double underscore (_ _).
- Functions so marked are generally a low-level component of the interface and should be used with caution.

Kernel Basics – First module

- The double underscore says to the programmer: "If you call this function, be sure you know what you are doing.“
- Example of such is the **copy_to_user** function where the non-double underscore version does some necessary checks before attempting the copy. _ _**copy_to_user** does not.

Kernel Basics – First module

- Kernel code should not do floating point arithmetic.
- It is possible inside **kernel_fpu_begin()/kernel_fpu_end()**, but certainly not recommended.
- Linus Torvalds: In other words: the rule is that you really shouldn't use FP in the kernel.

Kernel Basics – First module

- Let’s tackle our first module, hello.c:

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
   printk(KERN_ALERT "Hello, world\n");
   return 0;
}

static void hello_exit(void)
{
   printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Kernel Basics – First module

- This module defines two functions.
- One to be invoked when the module is loaded into the kernel (**hello_init**)
- Other for when the module is removed (**hello_exit**).

Kernel Basics – First module

- The **module_init** and **module_exit** lines use special kernel macros to indicate the role of these two functions.
- Another special macro (MODULE_LICENSE) is used to tell the kernel that this module bears a free license.
- Other MODULE_X macros include MODULE_AUTHOR, MODULE_VERSION and many more.

Kernel Basics – First module

- The **printk** function is defined in the Linux kernel and made available to modules.
- The messages appear in kernel buffer which you can check by typing **dmesg** in your Android emulator shell.
- The use of **module_init** is mandatory.

Kernel Basics – First module

- This macro adds a special section to the module's object code stating where the module's initialization function is to be found.
- Without this definition, your initialization function is never called.
- As you might imagine compiling this code is not as simple as it is with applications.
- The kernel build tree is necessary in order to compile any module.

Kernel Basics – First module

- To build a module mymodule.ko from its source file mymodule.c, create a one-line Makefile and execute it as follows:

```
bash> cd /path/to/module-source/
bash> echo "obj-m += mymodule.o" > Makefile
bash> make -C /path/to/kernel-sources/ M=`pwd` modules
```

- Remember to set envinromental variables ARCH and CROSS_COMPILE either before with export or at command time like when compiling the whole kernel.

Kernel Basics – First module

- Make command starts by changing its directory to the one provided with the -C option (that is, your kernel source directory).
- There it finds the kernel's top-level makefile
- The M= option causes that makefile to move back into your module source directory before trying to build the modules target.

Kernel Basics – First module

- This target, in turn, refers to the list of modules found in the obj-m variable.

- If you have a module called *module.ko* that is generated from two source files (called, say, *file1.c* and *file2.c*), the correct incantation would be:
- obj-m := module.o
- module-objs := file1.o file2.o

Kernel Basics – First module

- Modules are strongly tied to the data structures and function prototypes defined in a particular kernel version.
- The interface seen by a module can change significantly from one kernel version to the next.
- The kernel does not just assume that a given module has been built against the proper kernel version.

Kernel Basics – First module

- One of the steps in the build process is to link your module against a file (called *vermagic.o*) from the current kernel tree.

- This object contains a fair amount of information about the kernel the module was built for, including:
- The target kernel version,
- compiler version,
- the settings of a number of important configuration variables.

- When an attempt is made to load a module, this information can be tested for compatibility with the running kernel.

Kernel Basics – First module

- If you need to compile a module for a specific kernel version, you will need to use the build system and source tree for that particular version.
- Kernel interfaces often change between releases.
- If you are writing a module that is intended to work with multiple versions of the kernel, you likely have to make use of macros and #ifdef constructs to make your code build properly.

Kernel Basics – First module

- Most dependencies based on the kernel version can be worked around with preprocessor conditionals by exploiting KERNEL_VERSION and LINUX_VERSION_CODE.

- The best way to deal with version incompatibilities is by confining them to a specific header file.

Kernel Basics – First module

- As a general rule, code which is explicitly version (or platform) dependent should be hidden behind a low-level macro or function.

- High-level code can then just call those functions without concern for the low-level details.

- Code written in this way tends to be easier to read and more robust.

Kernel Basics – First module

- Several parameters that a driver needs to know can change from system to system.

- These can vary from the device number to use to numerous aspects of how the driver should operate.

- If your driver controls older hardware, it may also need to be told explicitly where to find that hardware's I/O ports or I/O memory addresses.

Kernel Basics – First module

- The kernel supports these needs by making it possible for a driver to designate parameters that may be changed when the driver's module is loaded.
- These parameter values can be assigned at load time by **insmod** or **modprobe**.
- When adding parameters you need to give them a name and a type.

Kernel Basics – First module

- Parameters are added with macro **module_param(name, type, perm).**
- The permissions have to do with kernel symbols and whether someone can change their values.
- Usually we use S_IRUGO: user, group and other can read our parameter.

Kernel Basics – First module

- Kernel specific permissions from linux/stat.h:

S_IRWXUGO	(S_IRWXU|S_IRWXG|S_IRWXO)
S_IALLUGO	(S_ISUID|S_ISGID|S_ISVTX|S_IRWXUGO)
S_IRUGO	(S_IRUSR|S_IRGRP|S_IROTH)
S_IWUGO	(S_IWUSR|S_IWGRP|S_IWOTH)
S_IXUGO	(S_IXUSR|S_IXGRP|S_IXOTH)

Kernel Basics – First module

- Most common types supported are int, charp (char pointer), bool, invbool (inversed boolean), long, short, uint, ulong and ushort.
- So have an int parameter named myParam which cannot be changed later we’d have:

```
static int myParam = 1; //Always give an initialization value!
moduleparam(myParam, int, S_IRUGO);
```

Kernel Basics – Kernel Symbols

- When a module is loaded, any symbol exported via the **EXPORT_SYMBOL** (**EXPORT_SYMBOL_GPL** for “gpl only” code) macro by the module becomes part of the kernel symbol table.
- In the usual case, a module implements its own functionality without the need to export any symbols at all.
- You need to export symbols whenever other modules may benefit from using them.

Kernel Basics – Kernel Symbols

- Module stacking is useful in complex projects.
- If a new abstraction is implemented in the form of a device driver, it might offer a plug for hardware-specific implementations.
- For example, the video-for-linux set of drivers is split into a generic module that exports symbols used by lower-level device drivers for specific hardware.

Kernel Basics – Kernel Symbols

- According to your setup, you load the generic video module and the specific module for your installed hardware.
- Support for parallel ports and the wide variety of attachable devices is handled in the same way, as is the USB kernel subsystem.
- When using stacked modules, it is helpful to be aware of the **modprobe** utility.

Kernel Basics – Kernel Symbols

- According to your setup, you load the generic video module and the specific module for your installed hardware.
- Support for parallel ports and the wide variety of attachable devices is handled in the same way, as is the USB kernel subsystem.

Kernel Basics – Device Model intro

- The Linux way of looking at devices distinguishes between three fundamental device types.
- Each module usually implements one of these types.
- Thus any driver is classifiable as a char module, a block module, or a network module.

Kernel Basics – Device Model intro

- This division of modules into different types, or classes, is not a rigid one; the programmer can choose to build huge modules implementing different drivers in a single chunk of code.
- Good programmers, nonetheless, usually create a different module for each new functionality they implement.
- Decomposition is a key element of scalability and extendability.

Kernel Basics – Device Model intro

- A character (**char**) device is one that can be accessed as a stream of bytes (like a file).
- A char driver is in charge of implementing this behavior.
- Such a driver usually implements at least the **open**, **close**, **read**, and **write** system calls.

Kernel Basics – Device Model intro

- Char devices are accessed by means of filesystem nodes, such as **/dev/tty1** and **/dev/lp0**.
- The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially.

Kernel Basics – Device Model intro

- Like char devices, block devices are accessed by filesystem nodes in the ***/dev*** directory.
- A block device is a device (e.g., a disk) that can host a filesystem.
- Linux allows the application to **read** and **write** a block device like a char device it permits the transfer of any number of bytes at a time.

Kernel Basics – Device Model intro

- Like a char device, each block device is accessed through a filesystem node.
- The difference between them is transparent to the user.
- Block drivers have a completely different interface to the kernel than char drivers.

Kernel Basics – Device Model intro

- Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts.
- Usually, an *interface* is a hardware device, but it might also be a pure software device, like the loopback interface.
- A network interface is in charge of sending and receiving data packets.

Kernel Basics – Device Model intro

- It is driven by the network subsystem of the kernel and has no idea of the larger picture beyond individual packets.
- A network driver knows nothing about individual connections; it only handles packets.
- There are other ways of classifying driver modules that are orthogonal to the above device types

Kernel Basics – Device Model intro

- In general, some types of drivers work with additional layers of kernel support functions for a given type of device.
- One can talk of universal serial bus (USB) modules, serial modules, SCSI modules, and so on.
- Every USB device is driven by a USB module that works with the USB subsystem.

Kernel Basics – Device Model intro

- The device itself shows up in the system as a char device (a USB serial port,), a block device (a USB memory card reader), or a network device (a USB Ethernet interface).
- The 2.6 device model provides an abstraction of a unified device model for the kernel.

Kernel Basics – Device Model intro

- It is used within the kernel to support a wide variety of tasks, including:

- Power management and system shutdown
- Communications with user space
- Hotpluggable devices
- Device classes
- Object lifecycles

Kernel Basics – Device Model intro

- The **kobject** is the fundamental structure that holds the device model together.
- It was initially conceived as a simple reference counter, but its responsibilities have grown over time.

Kernel Basics – Device Model intro

- The tasks handled by **struct kobject** and its supporting code now include:

- Reference counting of objects
- Sysfs representation
- Data structure glue
- Hotplug event handling

Kernel Basics – Device Model intro

- Often, when a kernel object is created, there is no way to know just how long it will exist.
- One way of tracking the lifecycle of such objects is through reference counting.
- When no code in the kernel holds a reference to a given object, that object has finished its useful life and can be deleted.

Kernel Basics – Device Model intro

- Every object that shows up in sysfs has, underneath it, a **kobject** that interacts with the kernel to create its visible representation.

- The **kobject** subsystem handles the generation of events that notify user space about the comings and goings of hardware on the system.

Kernel Basics – Device Model intro

- The **kobject** relates to devices in following way.
- Every device type has its own struct.
- The kobject struct is a part of that struct.
- The gritty details of kobject initiliazation, reference counting and such have fortunately been implemented in most of the device model higher level code.

Kernel Basics – Device Model intro

- It is rare (even unknown) for kernel code to create a standalone **kobject.**
- A **device class** describes a type of device, like an audio or network device.
- A **struct kset** represents a set of kernel objects.
- More on kobjects at your kernel documentation. Also accessible at http://www.mjmwired.net/kernel/Documentation/kobject.txt

Kernel Basics – Device Model intro

- A **subsystem** is a representation for a high-level portion of the kernel as a whole.
- **Subsystems** usually (but not always) show up at the top of the sysfs hierarchy.
- Some example **subsystems** in the kernel include **block_subsys** (***/sys/block***, for block devices) and **devices_subsys (*/sys/devices***, the core device hierarchy)
- A driver author almost never needs to create a new subsystem

Kernel Basics – Memory basics

- The kmalloc allocation engine is a powerful tool and easily learned because of its similarity to malloc.
- The function is fast (unless it blocks) and doesn't clear the memory it obtains; the allocated region still holds its previous content.
- The allocated region is also contiguous in physical memory.

Kernel Basics – Memory basics

- The prototype for kmalloc is:

```
#include <linux/slab.h>
void *kmalloc(size_t size, int flags);
```

- Other versions like **kzalloc** (zeroed memory) exist.

Kernel Basics – Memory basics

- The first argument to kmalloc is the size of the block to be allocated.
- The second argument, the allocation flags, is much more interesting, because it controls the behavior of kmalloc in a number of ways.
- The most commonly used flag, **GFP_KERNEL**, means that the allocation is performed on behalf of a process running in kernel space.

Kernel Basics – Memory basics

- This is internally performed by calling, eventually, _ _get_free_pages, which is the source of the GFP_ prefix
- In other words, this means that the calling function is executing a system call on behalf of a process.
- Using **GFP_KERNEL** means that **kmalloc** can put the current process to sleep waiting for a page when called in low-memory situations.

Kernel Basics – Memory basics

- A function that allocates memory using **GFP_KERNEL** must, therefore, be reentrant and cannot be running in atomic context.

- While the current process sleeps, the kernel takes proper action to locate some free memory, either by flushing buffers to disk or by swapping out memory from a user process.

- **GFP_KERNEL** isn't always the right allocation flag to use; sometimes kmalloc is called from outside a process's context.

Kernel Basics – Memory basics

- This type of call can happen, for instance, in interrupt handlers, tasklets, and kernel timers. In this case, the current process should not be put to sleep, and the driver should use a flag of **GFP_ATOMIC** instead.

- The kernel normally tries to keep some free pages around in order to fulfill atomic allocation.

- When **GFP_ATOMIC** is used, kmalloc can use even the last free page. If that last page does not exist, however, the allocation fails.

Kernel Basics – Memory basics

- The flags also include:
- **GFP_USER**
- Used to allocate memory for user-space pages; it may sleep.

- **GFP_HIGHUSER**
- Like **GFP_USER**, but allocates from high memory, if any.

- **GFP_NOIO**
- **GFP_NOFS**

Kernel Basics – Memory basics

- These flags function like **GFP_KERNEL**, but they add restrictions on what the kernel can do to satisfy the request.
- A **GFP_NOFS** allocation is not allowed to perform any filesystem calls, while **GFP_NOIO** disallows the initiation of any I/O at all.
- They are used primarily in the filesystem and virtual memory code where an allocation may be allowed to sleep, but recursive filesystem calls would be a bad idea.

Kernel Basics – Memory basics

- The kernel manages the system's physical memory, which is available only in page-sized chunks. As a result, kmalloc looks rather different from a typical user-space malloc implementation.

- The kernel uses a special page-oriented allocation technique to get the best use from the system's RAM.

- Linux handles memory allocation by creating a set of pools of memory objects of fixed sizes.

Kernel Basics – Memory basics

- Allocation requests are handled by going to a pool that holds sufficiently large objects and handing an entire memory chunk back to the requester.
- The memory management scheme is quite complex
- The kernel can allocate only certain predefined, fixed-size byte arrays.
- If you ask for an arbitrary amount of memory, you're likely to get slightly more than you asked for, up to twice as much.

Kernel Basics – Basic Char Device

- Let's look at the internals of a character (or char) device driver, which is kernel code that sequentially accesses data from a device.
- Char drivers can capture raw data from several types of devices: printers, mice, watchdogs, tapes, memory, RTCs, and so on.
- They are however, not suitable for managing data residing on block devices capable of random access such as hard disks, floppies, or compact discs.

Kernel Basics – Basic Char Device

- To access a char device, a system user invokes a suitable application program.
- The application is responsible for talking to the device, but to do that, it needs to elicit the identity of a suitable driver.
- The contact details of the driver are exported to user space via the */dev* directory*.*

Kernel Basics – Basic Char Device

- Within the kernel, the **dev_t** type (defined in <linux/types.h>) is used to hold device numbers, both the major and minor parts
- As of Version 2.6.0 of the kernel, **dev_t** is a 32-bit quantity with 12 bits set aside for the major number and 20 for the minor number.
- Your code should, of course, never make any assumptions about the internal organization of device numbers

Kernel Basics – Basic Char Device

- It should, instead, make use of a set of macros found in <linux/kdev_t.h>.
- To obtain the major or minor parts of a **dev_t**, use:
- **MAJOR(dev_t dev);**
- **MINOR(dev_t dev);**

Kernel Basics – Basic Char Device

- If, instead, you have the major and minor numbers and need to turn them into a dev_t, use:
- **MKDEV(int major, int minor);**
- One of the first things your driver will need to do when setting up a char device is to obtain one or more device numbers to work with.
- The necessary function for this task is register_chrdev_region, which is declared in <linux/fs.h>:

Kernel Basics – Basic Char Device

- **int register_chrdev_region(dev_t first, unsigned int count, char *name);**
- Here, first is the beginning device number of the range you would like to allocate.
- The minor number portion of first is often 0, but there is no requirement to that effect. count is the total number of contiguous device numbers you are requesting.

Kernel Basics – Basic Char Device

- Note that, if count is large, the range you request could spill over to the next major number.
- Everything will still work properly as long as the number range you request is available.
- Finally, name is the name of the device that should be associated with this number range

Kernel Basics – Basic Char Device

- It will appear in */proc/devices* and sysfs.
- As with most kernel functions, the return value from **register_chrdev_region** will be 0 if the allocation was successfully performed.
- **register_chrdev_region** works well if you know ahead of time exactly which device numbers you want.

Kernel Basics – Basic Char Device

- Often, however, you will not know which major numbers your device will use; there is a constant effort within the Linux kernel development community to move over to the use of dynamicly-allocated device numbers.
- The kernel will happily allocate a major number for you on the fly, but you must request this allocation by using a different function:
- **int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);**

Kernel Basics – Basic Char Device

- With this function, **dev** is an output-only parameter that will, on successful completion, hold the first number in your allocated range.
- **firstminor** should be the requested first minor number to use; it is usually 0.
- The **count** and **name** parameters work like those given to **request_chrdev_region**.

Kernel Basics – Basic Char Device

- Regardless of how you allocate your device numbers, you should free them when they are no longer in use.

- Device numbers are freed with:

void unregister_chrdev_region(dev_t first, unsigned int count);

- The usual place to call unregister_chrdev_region would be in your module's cleanup function.

Kernel Basics – Basic Char Device

- The above functions allocate device numbers for your driver's use, but they do not tell the kernel anything about what you will actually do with those numbers
- Before a user-space program can access one of those device numbers, your driver needs to connect them to its internal functions that implement the device's operations.

Kernel Basics – Basic Char Device

- Let's look at what needs to be done when we create a char driver.
- From a code-flow perspective, char drivers have the following:
- An initialization (or init()) routine that is responsible for initializing the device and seamlessly tying the driver to the rest of the kernel via registration functions

Kernel Basics – Basic Char Device

- A set of entry points (or methods) such as open(), read(), ioctl(), llseek(), and write().
- These directly correspond to I/O system calls invoked by user applications over the associated */dev node.*
- Interrupt routines, bottom halves, timer handlers, helper kernel threads, and other support infrastructure. These are largely transparent to user applications

Kernel Basics – Basic Char Device – init

- The driver init() method is the bedrock of the registration mechanism. It's responsible for the following:
- Requesting allocation of device major numbers.
- Allocating memory for the per-device structure.
- Connecting the entry points (open(), read(), and so on) with the char driver's cdev abstraction.
- Associating the device major number with the driver's cdev.
- Creating nodes under */dev and /sys.*
- Initializing the hardware.

Kernel Basics – Basic Char Device – init

- The kernel uses structures of type struct cdev to represent char devices internally.

- Before the kernel invokes your device's operations, you must allocate and register one or more of these structures.

- To do so, your code should include <*linux/cdev.h*>, where the structure and its associated helper functions are defined.

Kernel Basics – Basic Char Device – init

- There are two ways of allocating and initializing one of these structures.

- If you wish to obtain a standalone cdev structure at runtime, you may do so with code such as:

- **struct cdev *my_cdev = cdev_alloc( );**
- **my_cdev->ops = &my_fops;**

Kernel Basics – Basic Char Device – init

- Chances are, however, that you will want to embed the cdev structure within a device-specific structure of your own.
- In that case, you should initialize the structure that you have already allocated with:
- **void cdev_init(struct cdev *cdev, struct file_operations *fops);**

Kernel Basics – Basic Char Device – init

- Either way, there is one other struct cdev field that you need to initialize.
- Like the file_operations structure, struct cdev has an owner field that should be set to THIS_MODULE.
- Once the cdev structure is set up, the final step is to tell the kernel about it with a call to:

Kernel Basics – Basic Char Device – init

- **int cdev_add(struct cdev *dev, dev_t num, unsigned int count);**
- dev is the cdev structure, num is the first device number to which this device responds
- Count is the number of device numbers that should be associated with the device.

Kernel Basics – Basic Char Device – init

- Often count is one, but there are situations where it makes sense to have more than one device number correspond to a specific device.
- There are a couple of important things to keep in mind when using cdev_add.
- The first is that this call can fail.

Kernel Basics – Basic Char Device – init

- If it returns a negative error code, your device has not been added to the system.
- It almost always succeeds, however, and that brings up the other point: as soon as **cdev_add** returns, your device is "live" and its operations can be called by the kernel.
- You should not call **cdev_add** until your driver is completely ready to handle operations on the device.

Kernel Basics – Basic Char Device – init

- Within init function you also need to create an inode for the device under /dev/.
- There are a few ways of doing this but we are going to create a device class and a device into that class.
- Device class represents a group of similar devices, usually by same manufacturer, which have common characteristics.
- Creating a device class and then populating the class with devices allows the system administrator to handle the rules for creating the nodes.

Kernel Basics – Basic Char Device – init

- When a device appears under a class in /sys/ device managers such as udev can have rules to handle the node creation
- Standard operating procedure for a class of character devices is to create an only-superuser accessable device under /dev/
- So lets call:
- **class_create(owner, name)**
- **struct device *device_create(struct class *cls, struct device *parent, dev_t devt, void *drvdata, const char *fmt, ...);**

Kernel Basics – Basic Char Device – init

- To remove a char device from the system, call:
- **void cdev_del(struct cdev *dev);**
- Let’s look at the example code.

Kernel Basics – Basic Char Device – init

```
•   static int __init my_init(void) {

•   if (alloc_chrdev_region(&pseudo_dev_nro, 0, 1, DEVICE_NAME) < 0) {
•           printk(KERN_DEBUG "pseudoram: Can't register device\n");
•           return -1;
•     }
•   devices = kmalloc(sizeof(struct pseudoram_dev),GFP_KERNEL);
•     if (devices == NULL) {
•                 printk(KERN_DEBUG "pseudoram: Can't reserve memory for devices\n");
•                 return -1;
•     }
•     pseudoram_class = class_create(THIS_MODULE,DEVICE_NAME);
•     cdev_init(&devices->cdev, &pseudo_fops);
•      devices->cdev.owner = THIS_MODULE;


•     if (cdev_add(&devices->cdev, pseudo_dev_nro, 1)) {
•                 printk(KERN_DEBUG "pseudoram: Bad cdev\n");
•                 return 1;
•           }
•     device_create(pseudoram_class, NULL, pseudo_dev_nro, NULL, "pseudo0");
•   }
```

Kernel Basics – Basic Char Device – open

- The open method is provided for a driver to do any initialization in preparation for later operations.

- In most drivers, open should perform the following tasks:

- Check for device-specific errors (such as device-not-ready or similar hardware problems)
- Initialize the device if it is being opened for the first time
- Update the file_operations pointer, if necessary
- Allocate and fill any data structure to be put in file_pointer->private_data

Kernel Basics – Basic Char Device – open

- The first order of business, however, is usually to identify which device is being opened.
- The prototype for the open method is:
- **int (*open)(struct inode *inode, struct file *filp);**

Kernel Basics – Basic Char Device – open

- In this case the kernel hackers have done the tricky stuff for us, in the form of the **container_of** macro, defined in <*linux/kernel.h*>:
- **container_of(pointer, container_type, container_field);**
- This macro takes a pointer to a field of type container_field, within a structure of type container_type, and returns a pointer to the containing structure.
- **struct pseudoram_dev *device;**
- **device = container_of(inode->i_cdev, struct pseudoram_dev, cdev);**

Kernel Basics – Basic Char Device – release

- The role of the release method is the reverse of open.
- Sometimes you'll find that the method implementation is called device_close instead of device_release.
- Either way, the device method should perform the following tasks:
- Deallocate anything that open allocated in file_pointer->private_data
- Shut down the device on last close

Kernel Basics – Basic Char Device – r/w

- The read and write methods both perform a similar task, that is, copying data from and to application code.
- Therefore, their prototypes are pretty similar, and it's worth introducing them at the same time:
- **ssize_t read(struct file *filp, char _ _user *buff, size_t count, loff_t *offp);**
- **ssize_t write(struct file *filp, const char _ _user *buff, size_t count, loff_t *offp);**

Kernel Basics – Basic Char Device – r/w

- For both methods, **filp** is the file pointer and **count** is the size of the requested data transfer.
- The **buff** argument points to the user buffer holding the data to be written or the empty buffer where the newly read data should be placed.
- Finally, **offp** is a pointer to a "long offset type" object that indicates the file position the user is accessing.

Kernel Basics – Basic Char Device – r/w

- Let’s look at some pointers on generating read and write.
- Let us repeat that the buff argument to the read and write methods is a user-space pointer.
- Therefore, it cannot be directly dereferenced by kernel code.

Kernel Basics – Basic Char Device – r/w

- There are a few reasons for this restriction:
- Depending on which architecture your driver is running on, and how the kernel was configured, the user-space pointer may not be valid while running in kernel mode at all. There may be no mapping for that address, or it could point to some other, random data.
- Depending on which architecture your driver is running on, and how the kernel was configured, the user-space pointer may not be valid while running in kernel mode at all.

Kernel Basics – Basic Char Device – r/w

- There may be no mapping for that address, or it could point to some other, random data.
- Attempting to reference the user-space memory directly could generate a page fault, which is something that kernel code is not allowed to do.
- The result would be an "oops," which would result in the death of the process that made the system call.

Kernel Basics – Basic Char Device – r/w

- The pointer in question has been supplied by a user program, which could be buggy or malicious.
- If your driver ever blindly dereferences a user-supplied pointer, it provides an open doorway allowing a user-space program to access or overwrite memory anywhere in the system.
- Obviously, your driver must be able to access the user-space buffer in order to get its job done.

Kernel Basics – Basic Char Device – r/w

- This access must always be performed by special, kernel-supplied functions, however, in order to be safe.
- If your device needs to copy a whole segment of data to or from the user address space, this capability is offered by the following kernel functions, which copy an arbitrary array of bytes and sit at the heart of most read and write implementations:

Kernel Basics – Basic Char Device – r/w

- **int long copy_to_user(void _ _user *to, const void *from,**
  unsigned long count);

- **int long copy_from_user(void *to, const void _ _user *from,**
  unsigned long count);

Kernel Basics – Basic Char Device – r/w

- Although these functions behave like normal **memcpy** functions, a little extra care must be used when accessing user space from kernel code.
- The user pages being addressed might not be currently present in memory, and the virtual memory subsystem can put the process to sleep while the page is being transferred into place
- This happens, for example, when the page must be retrieved from swap space.

Kernel Basics – Basic Char Device – r/w

- The net result for the driver writer is that any function that accesses user space must be reentrant, must be able to execute concurrently with other driver functions, and, in particular, must be in a position where it can legally sleep.
- The role of the two functions is not limited to copying data to and from user-space: they also check whether the user space pointer is valid.
- If the pointer is invalid, no copy is performed.

Kernel Basics – Basic Char Device – r/w

- If an invalid address is encountered during the copy, on the other hand, only part of the data is copied.
- In both cases, the return value is the amount of memory still to be copied.
- If you don't need to check the user-space pointer you can invoke _ _**copy_to_user** and _ _**copy_from_user** instead.

Kernel Basics – Basic Char Device – r/w

- This is useful, for example, if you know you already checked the argument.
- Be careful, however; if, in fact, you do not check a user-space pointer that you pass to these functions, then you can create kernel crashes and/or security holes.
- Whatever the amount of data the methods transfer, they should generally update the file position at ***offp** to represent the current file position after successful completion of the system call.

Kernel Basics – Basic Char Device – r/w

- The kernel then propagates the file position change back into the file structure when appropriate.
- Both the read and write methods return a negative value if an error occurs
- A return value greater than or equal to 0, instead, tells the calling program how many bytes have been successfully transferred.

Kernel Basics – Basic Char Device – r/w

- If some data is transferred correctly and then an error happens, the return value must be the count of bytes successfully transferred.
- The error does not get reported until the next time the function is called.
- Implementing this convention requires, of course, that your driver remember that the error has occurred so that it can return the error status in the future.

Kernel Basics – Basic Char Device – r/w

- The return value for read is interpreted by the calling application program:
- If the value equals the count argument passed to the read system call, the requested number of bytes has been transferred. This is the optimal case.
- If the value is positive, but smaller than count, only part of the data has been transferred

Kernel Basics – Basic Char Device – r/w

- This may happen for a number of reasons, depending on the device.
- Most often, the application program retries the read. For instance, if you read using the fread function, the library function reissues the system call until completion of the requested data transfer
- If the value is 0, end-of-file was reached (and no data was read).
- A negative value means there was an error.

Kernel Basics – Basic Char Device – r/w

- The value specifies what the error was, according to <*linux/errno.h*>.
- Typical values returned on error include -EINTR (interrupted system call) or -EFAULT (bad address).
- What is missing from the preceding list is the case of "there is no data, but it may arrive later." In this case, the read system call should block.

Kernel Basics – Basic Char Device – r/w

- If a char driver's **write()** method returns successfully, it implies that the driver has assumed responsibility for the data passed down to it by the application.

- However it does not guarantee that the data has been successfully written to the device.

- If an application needs this assurance, it can invoke the **fsync()** system call.