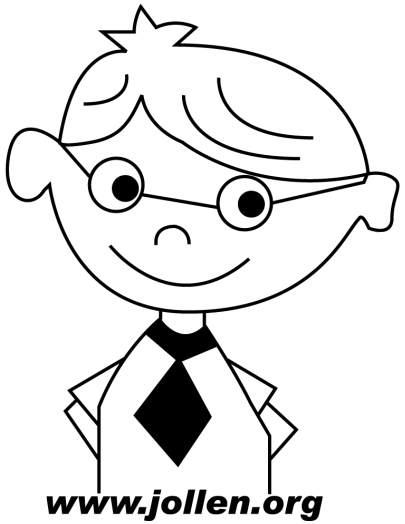


主講：Jollen Chen <jollen@jollen.org>
Blog：http://www.jollen.org/blog
筆記：http://www.jollen.org/wiki
課程：http://www.jollen.org/consulting

主辦單位：仕橙3G教室 (Moko365 Inc)
開課日期：2012/11/17 開課 (假日班)
課程時間：09:30~16:30、共計48小時

Linux 驅動程式完整訓練: Jollen 的 10 堂課《第28期》

本教材由仕橙3G教室製作與維護

LINUX DEVICE DRIVERS, 2nd Edition. O'REILLY.
重點整理 Chap. 3, 5-9, 12-13 (共8章)

LINUX DEVICE DRIVERS, 2nd Edition. O'REILLY.

-- Char Driver

Character Drivers

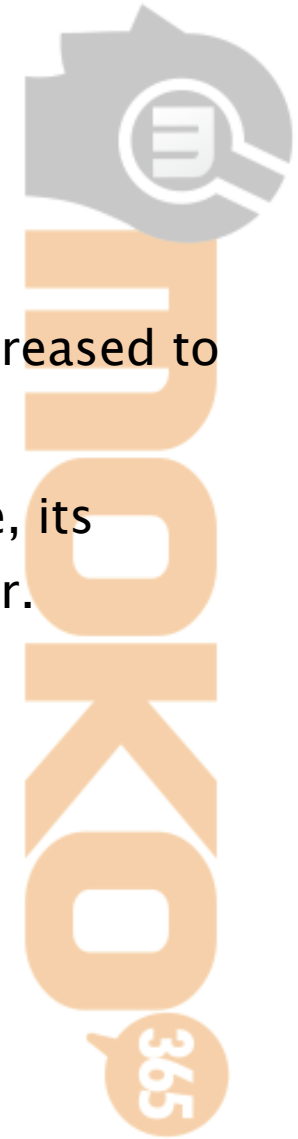
- The first step of driver writing is defining the capabilities (the mechanism) the driver will offer to user programs.
- Char devices are accessed through **device files** (special files) in the filesystem.
- Device files are located in the */dev* directory.

Character Drivers

- The major number identifies the driver associated with the device.
- The kernel uses the major number at *open* time to dispatch execution to the appropriate driver.
- The minor number is used only by the driver specified by the minor number.

Character Drivers

- **fops** is used to invoke driver's entry points.
- The 2.0 kernel supported 128 devices; 2.2 and 2.4 increased to 256.
- Once the driver has been registered in the kernel table, its operations are associated with the given major number.



Character Drivers

- Whenever an operation is performed on a character device file associated with that major number, the kernel finds and invokes the proper function from the *file_operations* structure.
- The pointer passed to *register_chrdeve* should point to a global structure within the driver.

Dynamic Allocation of Major Numbers

- If the argument *major* is set to 0 when you call *register_chrdev*, the function selects a free number and returns it.
- For private drivers, we strongly suggest that you use dynamic allocation to obtain your major device number, rather than choosing a number randomly from the ones that are currently free.

Dynamic Allocation of Major Numbers

- If your driver is meant to be useful to the community at large and be included into the official kernel tree, you'll need to apply to be assigned a major number for exclusive use.
- You can't create the device nodes in advance because the major number assigned to your modules won't always the same.

Dynamic Allocation of Major Numbers

- You won't be able to use loading-on-demand of your driver.
- Once the number has been assigned, you can read it from */proc/devices*.

Removing a Driver from the System

- When a module is unloaded from the system, the major number must be released.
- **oops** is the kind of fault message the kernel prints that when it tries to access invalid addresses.

dev_t and kdev_t

- The combined device number resides in the field *i_rdev* of the *inode* structure.
- Some driver functions receive a pointer to *struct inode* as the first argument.
- Calling the pointer *inode*, the function can extract the device number by looking at *inode->i_rdev*.

dev_t and kdev_t

- Historically, Unix declared *dev_t* (device type) to hold the device numbers. It used to be a 16-bit integer value defined in *<sys/types.h>*.
- Within the Linux kernel, *kdev_t* is used. This data type is designed to be a black box for every kernel function.

dev_t and kdev_t

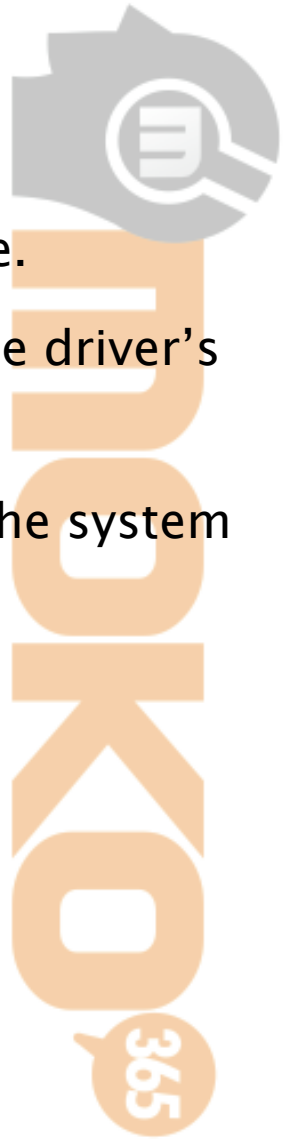
- User programs do not know about *kdev_t* at all, and kernel functions are unaware of what is inside a *kdev_t*.
- If *kdev_t* remains hidden, it can change from one kernel version to the next as needed, without requiring changes to everyone's device drivers.

dev_t and kdev_t

- The information about *kdev_t* is confined in *<linux/kdev_t.h>*.
- There's no need to include the header explicitly in the drivers, because *<linux/fs.h>* does it for you.
- As long as your code uses macros and functions to manipulate device number, it should continue to work even as the internal data structures change.

File Operations

- An open device is identified internally by a *file* structure.
- The kernel uses the *file_operations* structure to access the driver's functions.
- The operations are mostly in charge of implementing the system calls.



File Operations

- We can consider the file to be an object and the functions operating on it to be its method, using object-oriented programming terminology to denote actions declared by an object to act on itself.
- A *file_operations* structure or a pointer to one is called *fops*.

File Operations

- Each field in the structure must point to the function in the driver that implements a specific operation, or be left *NULL* for unsupported operations.
- The exact behavior of the kernel when a *NULL* pointer is specified is different for each function.

File Operations

- Data structure *file* includes a pointer to its own *file_operations*.
- Tagged structure initialization syntax works only on 2.4 kernels.
- A more portable approach is to use the *SET_MODULE_OWNER* macro, defined in *<linux/module.h>*.

The file Structure

- *struct file*, defined in *<linux/fs.h>*, is the second most important data structure used in device drivers.
- A *struct file*, on the other hand, is a kernel structure that never appears in user programs.

The file Structure

- The *file* structure represents an **open file.**
- Every open file in the system has an associated *struct file* in kernel space.
- It's created by the kernel on *open* and is passed to any function that operates on the file, until the last *close*.
- After all instances of the file are closed, the kernel releases the data structure.

The file Structure

- Disk files use *struct file*.
- Drivers never fill *file* structures; they only access structures created elsewhere.

The open Method

- *open* usually increments the usage count for the device so that the module won't be unloaded before the file is closed.
- The count is then decremented by the *release* method.
- The first thing to do is identify which device is involved. *(inode->i_rdev)*

The open Method

- Different minor numbers are used to access different devices or to open the same device in a different way.
- A driver never actually knows the name of the device being opened, just the device number.
- The preferred way to implement aliasing is creating a symbolic link.

The open Method

- Usually, a driver doesn't invoke its own *fops*, because they are used by the kernel to dispatch the right driver method.
- Given that the kernel can maintain the usage count of the module via the *owner* field in the *file_operations* structure, you may be wondering why we increment that count manually here.

The open Method

- To be portable to older kernels, *scull* increments its own usage count.
- This behavior will cause the usage count to be too high on 2.4 systems, but that is not a problem because it will still drop to zero when the module is not being used.

The release Method

- Sometimes you'll find that the method implementation is called device_close instead of device_release.
- It is important to decrement the usage count if you incremented it at open time.
- Not every close system call causes the release method to be invoked. Only the ones that actually release the device data structure invoke the method.

The release Method

- The *close* system call executes the *release* method only when the counter for the *file* structure drops to zero, which happens when the structure is destroyed.
- The *flush* method is called every time an application calls *close*.
- Usually there's nothing to perform at close time unless *release* is involved.

The release Method

- The kernel automatically closes any file at process exit time by internally using the *close* system call.

A Brief Introduction to Race Conditions

- A *semaphore* is a general mechanism for controlling access to resources.
- A semaphore may be used for *mutual exclusion*.
- Semaphore must be initialized prior to use by passing a numeric argument to *sema_init*.

A Brief Introduction to Race Conditions

- The semaphore should be initialized to a value of 1, which means that the semaphore is available.
- The function to obtain a semaphore is often call P, in Linux you'll need to call down or down_interruptible.
- The down_interruptible function can be interrupted by a signal, whereas down will not allow signals to be delivered to the process.

A Brief Introduction to Race Conditions

- A process that obtains a semaphore must always release it afterward.
- The release function V, Linux uses *up*.

read and write

- As far as data transfer is concerned, the main issue associated with the two device methods is the need to transfer data between the kernel address space and the user address space.
- User-space addresses cannot be used directly in kernel space.

read and write

- One big difference between kernel-space addresses and user-space addresses is that memory in user-space can be swapped out.
- The user pages being addressed might not be currently present in memory, and the page-fault handler can put the process to sleep while the page is being transferred into place.

read and write

- Any function that accesses user space must be reentrant and must be able to execute concurrently with other driver functions.
- They also check whether the user space pointer is valid.
- If you don't need to check the user-space pointer you can invoke *__copy_to_user* and *__copy_from_user* instead.

read and write

- They should in general update the file position at **offp* to represent the current file position after sucessful completion of the system call.

readv and writev

- These “vector” versions take an array of structures, each of which contains a pointer to a buffer and a length value.
- Each *iovec* describes one chunk of data to be transferred; it starts at *iov_base* (in user space) and is *iov_len* bytes long.
- The *count* parameter to the method tells how many *iovec* structures there are.

Main Advantages of devfs

- Device entry points in /dev arecreated at device initialization and removed at device removal.
- The device driver can specify device names, ownership, and permission bits, but user-space programs can still change ownershop and permission (but not the filename).
- There is no need to allocate a major number for the device driver and deal with minor numbers.

LINUX DEVICE DRIVERS, 2nd Edition. O'REILLY.

-- Enchanced Char Driver Operations

ioctl

- Control operations are usually supported via the ioctl method.
- The ioctl system call offers a device specific entry point for the driver to handle “commands.”
- These control operations are usually not available through the read/write file abstraction.

ioctl

- Most *ioctl* implementations consist of a *switch* statement that selects the correct behavior according to the *cmd* argument.

Choosing the ioctl Commands

- The command numbers should be unique across the system in order to prevent errors caused by issuing the right command to the wrong device.
- To choose *ioctl* numbers for your driver according to the new convention, you should first check *include/asm/ioctl.h* and *Documentation/ioctl-number.txt*.

Choosing the ioctl Commands

- The header defines the bitfields you will be using: type (magic number), ordinal number, direction of transfer, and size of argument.
- The header file *<asm/ioctl.h>*, which is included by *<linux/ioctl.h>*, defines macros that help set up the command numbers as follows: *_IO(type,nr)*, *_IOR(type,nr,dataitem)*, *_IOW(type,nr,dataitem)*, and *_IOWR(type,nr,dataitem)*.

The Return Value

- The POSIX standard, however, states that if an inappropriate *ioctl* command has been issued, then *–ENOTTY* should be returned.

The Predefined Commands

- Though the *ioctl* system call is most often used to act on devices, a few commands are recognized by the kernel.
- If you choose the same number for one of your *ioctl* commands, you won't ever see any request for that command.

The Predefined Commands

- The predefined commands are divided into three groups.
- First, those that can be issued on any file (regular, device, FIFO, or socket).
- Second, those that are issued only on regular files.
- Third, those specific to the filesystem type.

The Predefined Commands

- Commands in the last group are executed by the implementation of the hosting filesystem.
- Device driver writers are interested only in the first group of commands, whose magic number is “T.”

Using the ioctl Argument

- When a pointer is used to refer to user space, we must ensure that the user address is valid and that the corresponding page is currently mapped.
- It's the driver's responsibility to make proper checks on every user-space address it uses and to return an error if it is invalid.

Using the ioctl Argument

- Address verification for kernels 2.2.x and beyond is implemented by the function access_ok, which is declared in <asm/uaccess.h>.
- Unlike most functions, access_ok returns a boolean value: 1 for success and 0 for failure.
- If it returns false, the driver will usually return -EFAULT to the caller.

Using the ioctl Argument

- access_ok ensures that the address does not point to kernel-space memory.
- Most driver code need not actually call access_ok. The memory-access routines described later take care of that for you.
- In addition to the copy_from_user and copy_to_user functions, the programmer can exploit a set of functions that are optimized for the most-used data sizes.

Capabilities and Restricted Operations

- Privilege is an all-or-nothing thing.
- The Linux kernel as of version 2.2 provides a more flexible system called capabilities.
- A capability-based system leaves the all-or-nothing mode behind and breaks down privileged operations into separate subgroups.

Capabilities and Restricted Operations

- The full set of capabilities can be found in *<linux/capability.h>*.
- Before performing a privileged operation, a device driver should check that the calling process has the appropriate capability with the *capable* function (defined in <sys/sched.h>).

Device Control Without ioctl

- Sometimes controlling the device is better accomplished by writing control sequences to the device itself.
- This behavior has the advantage of permitting the remote control of devices.
- Controlling by write is definitely the way to go for those devices that don't transfer data but just respond to commands.
- The advantage of direct device control is that you can use cat to move the camera without writing and compiling special code to issue the ioctl calls.

Blocking I/O

- One problem that might arise with *read* is what to do when there's no data yet, but we're not at end-of-file.
- The default answer is "go to sleep waiting for data."

Going to Sleep and Awakening

- Whenever a process must wait for an event, it should go to sleep.
- All, however, work with the same basic data type, a wait queue (*wait_queue_head_t*).
- A wait queue is exactly that – a queue of processes that are waiting for an event.

Going to Sleep and Awakening

- Sleeping is accomplished by calling one of the variants of *sleep_on*.
- Driver writers should almost always use the interruptible instances of these functions/macros.
- Typically a driver will wake up sleepers in its interrupt handler once new data has arrived.

Going to Sleep and Awakening

- An important thing to remember with wait queues is that being woken up does not guarantee that the event you were waiting for has occurred.

A Deeper Look at Wait Queues

- *schedule* returns only when somebody else has woken up the process and set its state to *TASK_RUNNING*.
- As suggested, *wait_event* is now the preferred way to sleep on an event, because *interruptible_sleep_on* is subject to unpleasant race conditions.

A Deeper Look at Wait Queues

- One other reason for calling the scheduler explicitly, however, is to do exclusive waits.
- Only one process will be able to read that data; all the rest will simply wake up, see that no data is available, and go back to sleep.
- If processes sleep in an exclusive mode, they are telling the kernel to wake only one of them. The result is improved performance in some situations.

Writing Reentrant code

- Reentrant code is code that doesn't keep status information in global variables and thus is able to manage interwoven invocations without mixing anything up.
- If status information is needed, it can either be kept in local variables within the driver function (each process has a different stack page in kernel space where local variables are stored), or it can reside in private_data within the filp accessing the file.

Writing Reentrant code

- Using local variables is preferred because sometimes the same filp can be shared between two processes (usually parent and child).
- If you need to save large amounts of status data, you can keep the pointer in a local variable and us kmalloc to retrieve the actual storage space.
- In this case you must remember to kfree the data.

Writing Reentrant code

- You need to make reentrant any function that matches either of two conditions.
- First, if it calls schedule, possibly by calling sleep_on or wake_up.
- Second, if it copies data to or from user space, because access to user space might page-fault, and the process will be put to sleep while the kernel deals with the missing page.

Writing Reentrant code

- Every function that calls any such functions must be reentrant as well.

Blocking and Nonblocking Operations

- In the case of a blocking operation, which is the default, the following behavior should be implemented in order to adhere to the standard semantics.
- First, if a process calls read but no data is (yet) available, the process must block.
- Second, if a process calls write and there is no space in the buffer, the process must block , and it must be on a different wait queue from the one used for reading.

Blocking and Nonblocking Operations

- The output buffer is almost always useful for squeezing more performance out of the hardware.
- The performance gain of implementing an output buffer in the driver results from the reduced number of context switches and user-level/kernel-level transitions.

Blocking and Nonblocking Operations

- The choice of a suitable size for the output buffer is clearly device specific.
- The behavior of *read* and *write* is different if *O_NONBLOCK* is specified.
- Applications must be careful when using the *stdio* functions while dealing with nonblocking files, because they can easily mistake a nonblocking return for *EOF*.

Blocking and Nonblocking Operations

- Usually, opening a device either succeeds or fails, without the need to wait for external events.
- Only the *read*, *write* and *open* file operations are affected by the nonblocking flag.

A Sample Implementation: scullpipe

- It's worth repeating that a process can go to sleep both when it calls schedule, either directly or indirectly, and when it copies data to or from user space.
- The driver returns -ERESTARTSYS to the caller; this value is used internally by the virtual filesystem (VFS) layer, which either restarts the system call or returns -EINTR to user space.

poll and select

- Applications that use nonblocking I/O often use the poll and select system calls as well.
- select was introduced in BSD Unix, whereas poll was the System V solution.
- The driver's method will be called whenever the user-space program performs a poll or select system call involving a file descriptor associated with the driver.

Asynchronous Notification

- Not all the devices support asynchronous notification.
- Applications usually assume that the asynchronous capability is available only for sockets and ttys.
- Pipes and FIFOs don't support it, at least in the current kernels.
- If more than one file is enabled to asynchronously notify the process of pending input, the application must still resort to poll or select to find out what happened.

The Driver's Point of View

- It's clear that all the work is performed by *fasync_helper*.
- Invoking *fasync* is required only if *filp->f_flags* has *FASYNC* set, calling the function anyway doesn't hurt and is the usual implementation.

Seeking a Device

- The llseek method implements the lseek and llseek system calls.
- We have already stated that if the llseek method is missing from the device's operations, the default implementation in the kernel performs seeks from the beginning of the file and from the current position by modifying filp->f_ops, the current reading/writing position within the file.

Seeking a Device

- The only device-specific operation is retrieving the file length from the device.

Access Control on a Device File

- None of the code shown up to now implements any access control beyond the filesystem permission bits.

Another Digression into Race Conditions

- Semaphores can be expensive to use, because they can put the calling process to sleep.
- Spinlocks will never put a process to sleep.
- Spinlocks can be the ideal mechanism for small critical sections.

Restricting Access to a Single User at a Time

- The next step beyond a single system-wide lock is to let a single user open a device in multiple processes but allow only one user to have the device open at a time.

Blocking open as an Alternative to EBUSY

- Wait queues were created to maintain a list of processes that sleep while waiting for an event.
- The problem with a blocking-open implementation is that it is really unpleasant for the interactive user, who has to keep guessing what is going wrong.
- This kind of problem (different, incompatible policies for the same device) is best solved by implementing one device node for each access policy.

Cloning the Device on Open

- Another technique to manage access control is creating different private copies of the device depending on the process opening it.
- When copies of the device are created by the software driver, we call them virtual devices – just as virtual consoles use a single physical tty device.

LINUX DEVICE DRIVERS, 2nd Edition. O'REILLY.

-- Flow of Time

Time Intervals in the Kernel

- The first point we need to cover is the timer interrupt, which is the mechanism the kernel uses to keep track of time intervals.
- Interrupts are asynchronous events that are usually fired by external hardware; the CPU is interrupted in its current activity and executes special code (the Interrupt Service Routine, or ISR).

Time Intervals in the Kernel

- Timer interrupts are generated by the system's timing hardware at regular intervals; this interval is set by the kernel according to the value of HZ.
- No driver writer should count on any specific value of HZ.

Time Intervals in the Kernel

- Every time a timer interrupt occurs, the value of the variable *jiffies* is incremented.
- *jiffies* is initialized to 0 when the system boots, and will possibly overflow after a long time of continuous system operation.

Knowing the Current Time

- Kernel code can always retrieve the current time by looking at the value of *jiffies*.

Delaying Execution

- Longer delays can make use of the system clock; shorter delays typically must be implemented with software loops.

Long Delays

- Busy loop completely locks the processor for the duration of the delay; the scheduler never interrupts a process that is running in kernel space.
- The best way to implement a delay, however, is to ask the kernel to do it for you.
- The timeout value represents the number of *jiffies* to wait, not an absolute time value.

Short Delays

- Sometimes a real driver needs to calculate very short delays in order to synchronize with the hardware.
- The former uses a software loop to delay execution for the required number of microseconds.
- The *udelay* function is where the BogoMips value is used.

Short Delays

- The *udelay* call should be called only for short time lapses because the precision of *loops_per_second* is only eight bits.
- The suggested maximum value for *udelay* is 1000 microseconds (one millisecond).
- *udelay* is a busy-waiting function.

Task Queues

- One feature many drivers need is the ability to schedule execution of some tasks at a later time without resorting to interrupts.
- Linux offers three different interfaces for this purpose: task queues, tasklets (as of kernel 2.3.43), and kernel timers.

Task Queues

- They are most useful when writing interrupt handlers.
- Kernel timers are used to schedule a task to run at a specific time in the future.
- A typical situation in which you might use task queues or tasklets is to manage hardware that cannot generate interrupts but still allows blocking read.

Task Queues

- You can declare your own task queue and trigger it at will, or you can register your tasks in predefined queues, which are run (triggered) by the kernel itself.

The Nature of Task Queues

- The queue itself is a list of structures (the tasks) that are owned by the kernel module declaring and queuing them.

How Task Queues Are Run

- A task queue, as we have already seen, is in practice a linked list of functions to call.
- When run_task_queue is asked to run a given queue, each entry in the list is executed.
- Most important, they almost certainly are not run when the process that queued the task is executing. They are, instead, run asynchronously.

How Task Queues Are Run

- In fact, task queue are often run as the result of a “software interrupt.”
- When you are outside of process context (i.e., in interrupt mode), you must observe the following three rules.
- First, no access to user space is allowed. Because there is no process context, there is no path to the user space associated with any particular process.

How Task Queues Are Run

- Second, the *current* pointer is not valid in interrupt mode, and cannot be used.
- Third, no sleeping or scheduling may be performed. Interrupt-mode code may not call *schedule* or *sleep_on*; it also may not call any other function that may sleep. Semaphores also may not be used since they can sleep.

How Task Queues Are Run

- Kernel code can tell if it is running in interrupt mode by calling the function *in_interrupt()*.
- A task can requeue itself in the same queue from which it was run.

Predefined Task Queues

- The easiest way to perform deferred execution is to use the queues that are already maintained by the kernel.
- Your driver can use only three of them: the scheduler queue, tq_timer, tq_immediate.

The scheduler queue

- The scheduler queue is, in some ways, the easiest to use. Because tasks executed from this queue do not run in interrupt mode, they can do more things; in particular, they can sleep.
- Rather than use *queue_task* directly, code using this queue must call *schedule_task* to put a task on the queue.

The scheduler queue

- As of 2.4.0-test11, the kernel runs a special process, called *keventd*, whose sole job is running tasks from the scheduler queue.
- In normal situations, tasks placed in the scheduler queue will run very quickly.

The scheduler queue

- A task that resubmits itself to the scheduler queue can run hundreds or thousands of times within a single timer tick.
- Even on a very heavily loaded system, the latency in the scheduler queue is quite small.

The timer queue

- The timer queue is different from the scheduler queue in that the queue (tq_timer) is directly available.
- Also, of course, tasks run from the timer queue are run in interrupt mode.
- You're guaranteed that the queue will run at the next clock tick.
- For timer queue, it must use queue_task to get things going.

The immediate queue

- This queue is run via the bottom-half mechanism.
- Bottom halves are run only when the kernel has been told that a run is necessary; this is accomplished by “marking” the bottom half.
- The necessary call is *mark_bh(IMMEDIATE_BH)*.

The immediate queue

- Be sure to call *mark_bh* after the task has been queued; otherwise, the kernel may run the task queue before your task has been added.
- The immediate queue is the fastest queue in the system – it's executed soonest and is run in interrupt time.

The immediate queue

- The queue is consumed either by the scheduler or as soon as one process returns from its system call.
- It's clear that the queue can't be used to delay the execution of a task – it's an "immediate" queue.
- Its purpose is to execute a task as soon as possible.
- Please note that you should not reregister your task in this queue.

Running Your Own Task Queues

- A custom queue is not automatically run by the kernel. The programmer who maintains a queue must arrange for a way of running it.
- You need to register a function to trigger the queue in one of the predefined queues.
- A custom queue can be useful whenever you need to accumulate jobs and execute them all at the same time.

Tasklets

- This mechanism, called tasklets, is now the preferred way to accomplish bottom-half tasks; indeed, bottom halves themselves are now implemented with tasklets.
- They are always run in interrupt time. Like task queues, tasklets will be run only once, even if scheduled multiple times, but tasklets may be run in parallel with other (different) tasklets on SMP systems.

Tasklets

- On SMP systems, tasklets are also guaranteed to run on the CPU that first schedules them, which provides better cache behavior and thus better performance.
- Each tasklet has associated with it a function that is called when the tasklet is to be executed.

Tasklets

- When your driver wants to schedule a tasklet to run, it calls tasklet_schedule.
- Tasklets may reschedule themselves in much the same manner as task queues.
- If your driver implements multiple tasklets, however, it should be prepared for the possibility that more than one of them could run simultaneously.
- In that case, spinlocks must be used to protect critical sections of the code.

Kernel Timers

- The ultimate resources for time keeping in the kernel are the timers. Timers are used to schedule execution of a function (a timer handler) at a particular time in the future.
- On the other hand, kernel timers are similar to task queues in that a function registered in a kernel timer is executed only once.

Kernel Timers

- The kernel timer are organized in a doubly linked list. This means that you can create as many timers as you want.
- A timer is characterized by its timeout value (in jiffies) and the function to be called when the timer expires.
- The timeout of a timer is a value in jiffies. Thus, timer->function will run when jiffies is equal to or greater than timer->expires.

Kernel Timers

- Once a *timer_list* structure is initialized, *add_timer* inserts it into a sorted list, which is then polled more or less 100 times per second.
- The timer expires at just the right time, even if the processor is executing in a system call.
- Timers can be another source of race conditions, even on uniprocessor systems.

Kernel Timers

- Any data structures accessed by the timer function should be protected from concurrent access, either by being atomic types or by using spinlocks.
- One must also be very careful to avoid race conditions with timer deletion.

LINUX DEVICE DRIVERS, 2nd Edition. O'REILLY.

-- Getting Hole of Memory

Getting Hole of Memory

- Modules are not involved in issues of segmentation, paging, and so on, since the kernel offers a unified memory management interface to the drivers.

The Real Story of kmalloc

- *kmalloc()* is fast – unless it blocks – and it doesn't clear the memory it obtains.
- The allocated region still holds its previous content.
- The allocated region is also contiguous in physical memory.

The Flags Argument

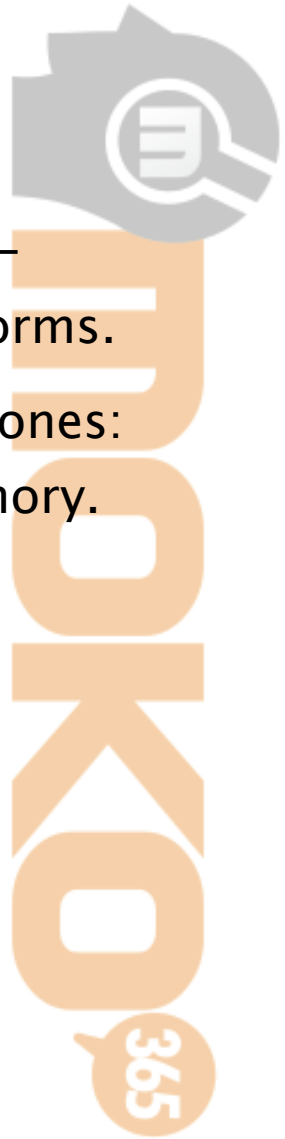
- The first argument to *kmalloc* is the size of the block to be allocated.
- The second argument, the allocation flags, is much more interesting, because it controls the behavior of *kmalloc* in a number of ways.

The Flags Argument

- Using *GFP_KERNEL* means that kmalloc can put the current process to sleep waiting for a page when called in low-memory situations.
- A Function that allocates memory using *GFP_KERNEL* must therefore be reentrant.
- While the current process sleeps, the kernel takes proper action to retrieve a memory page, either by flushing buffers to disk or by swapping out memory from a user process.

Memory Zones

- Both *__GFP_DMA* and *__GFP_HIGHMEN* have a platform-dependent role, although their use is valid for all platforms.
- Version 2.4 of the kernel knows about three memory zones: DMA-capable memory, normal memory, and high memory.



Memory Zones

- While allocation normally happens in the normal zone, setting either of the bits just mentioned requires memory to be allocated from a different zone.
- The idea is that every computer platform that must know about special memory ranges will fall into this abstraction.

Memory Zones

- DMA-capable memory is the only memory that can be involved in DMA data transfers with peripheral devices.
- This restriction arises when the address bus used to connect peripheral devices to the processor is limited with respect to the address bus used to access RAM.

Memory Zones

- High memory is memory that requires special handling to be accessed.
- High memory is a concept that only applies to the x86 and SPARC platforms, and the two implementations are different.

The Size Argument

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

- 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.

The Size Argument

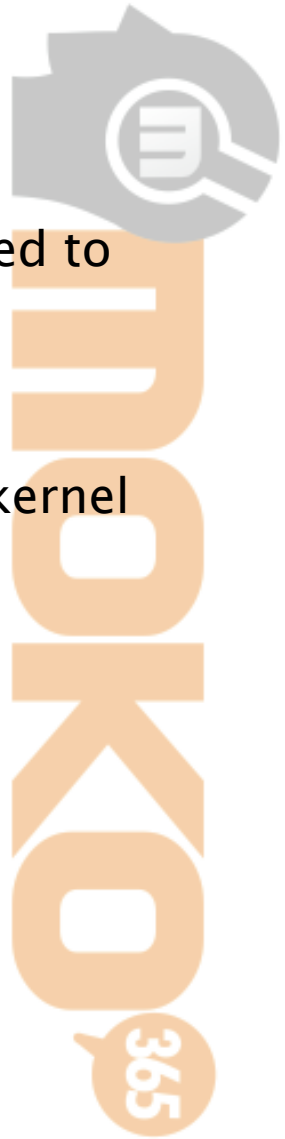
- The data sizes available are generally powers of two.
- In any case, the maximum size that can be allocated by *kmalloc* is 128KB.

Lookaside Caches

- The USB and ISDN drivers in Linux 2.4 use caches.
- A constructor is called when the memory for a set of objects is allocated; because that memory may hold several objects, the constructor may be called multiple times.

Lookaside Caches

- Constructors and destructors may or may not be allowed to sleep, according to whether they are passed the *SLAB_CTOR_ATOMIC* flag.
- One side benefit to using lookaside caches is that the kernel maintains statistics on cache usage.



get_free_page and Friends

- If a module needs to allocate big chunks of memory, it is usually better to use a page-oriented technique.
- The program calling these allocation functions must be prepared to handle an allocation failure.
- It's easy to degrade system responsiveness by allocating too much memory.

vmalloc and Friends

- *vmalloc*, which allocates a contiguous memory region in the virtual address space.
- Memory addresses returned by *kmalloc* and *get_free_pages* are also virtual addresses.

vmalloc and Friends

- The (virtual) addresses range used by *kmalloc* and *get_free_pages* features a one-to-one mapping to physical memory, possibly shifted by a constant *PAGE_OFFSET* value; the functions don't need to modify the page tables for that address range.

vmalloc and Friends

- The address range used by *vmalloc* and *ioremap*, on the other hand, is completely synthetic, and each allocation build the (virtual) memory area by suitably setting up the page tables.
- Address available for *vmalloc* are in the range from *VMALLOC_START* to *VMALLOC_END*.

vmalloc and Friends

- Addresses allocated by *vmalloc* can't be used outside of the microprocessor, because they make sense only on top of the processor's MMU.
- The right time to call *vmalloc* is when you are allocating memory for a large sequential buffer that exists only in software.

vmalloc and Friends

- Like *vmalloc*, *ioremap* builds new page tables; unlike *vmalloc*, however, it doesn't actually allocate any memory.
- The return value of *ioremap* is a special virtual address that can be used to access the specified physical address range.
- Functions like *readb* should be used.

vmalloc and Friends

- *ioremap* is most useful for mapping the (physical) address of a PCI buffer to (virtual) kernel space.
- It can be used to access the frame buffer of a PCI video device; such buffers are usually mapped at high physical addresses, outside of the address range for which the kernel builds page tables at boot time.

vmalloc and Friends

- There is almost no limit to how much memory *vmalloc* can allocate and *ioremap* can make accessible.
- Both *ioremap* and *vmalloc* are page oriented (they work by modifying the page tables);.
- The relocated or allocated size is rounded up to the nearest page boundary.

vmalloc and Friends

- One minor drawback of *vmalloc* is that it can't be used at interrupt time because internally it uses *kmalloc*(*GFP_KERNEL*) to acquire storage for the page tables, and thus could sleep.

Boot-Time Allocation

- If you really need a huge buffer of physically contiguous memory, you need to allocate it by requesting memory at boot time.
- A module can't allocate memory at boot time; only drivers directly linked to the kernel can do that.

Boot-Time Allocation

- Allocating memory at boot time is a “dirty” technique, because it bypasses all memory management policies by reserving a private memory pool.

Acquiring a Dedicated Buffer at Boot Time

- When the kernel is booted, it gains access to all the physical memory available in the system.
- Normal allocation returns memory addresses that are above *MAX_DMA_ADDRESS*; low memory is at addresses lower than that value.

Acquiring a Dedicated Buffer at Boot Time

- After a driver has taken some memory, it has no way of returning it to the pool of free pages; the pool is created after all the physical allocation has taken place.
- It makes available an area of consecutive physical memory that is suitable for DMA.
- If, however, you need many pages and they don't have to be physically contiguous, vmalloc is by far the best function to use.

The bigphysarea Patch

- Another approach that can be used to make large, contiguous memory regions available to drivers is to apply the *bigphysarea* patch.

Reserving High RAM Addresses

- The last option for allocating contiguous memory areas, and possibly the easiest, is reserving a memory area at the end of physical memory.

LINUX DEVICE DRIVERS, 2nd Edition. O'REILLY.

-- Hardware Management

I/O Ports and I/O Memory

- Every peripheral device is controlled by writing and reading its registers.
- Most of the time a device has several registers, and they are accessed at consecutive addresses, either in the memory address space or in the I/O address space.

I/O Ports and I/O Memory

- Linux implements the concept of I/O ports on all computer platforms.
- Peripheral bus has a separate address space for I/O ports, but not all devices map their registers to I/O ports.
- Most PCI devices map registers into a memory address region.

I/O Ports and I/O Memory

- The I/O memory approach is generally prefered because it doesn't require use of special-purpose processor instructions.

I/O Registers Conventional Memory

- The main difference between I/O registers and RAM is that I/O operations have side effects, while memory operations have none.
- Reordering can also happen both at compiler level and at hardware level.

I/O Registers Conventional Memory

- The memory barrier will enforce the completion of the writes in the necessary order.
- *mb()* will be slower than *wmb()*.

Using I/O Ports

- I/O ports are the means by which drivers communicate with many devices.
- I/O ports must be allocated before being used by your driver.
- Remapping port addresses to memory address.
- On 64-bit architectures, the port address space uses a 32-bit data patch.

Using I/O Memory

- I/O memory is simply a region of RAM-like locations that the device makes available to the processor over the bus.
- The way used to access I/O memory depends on the computer architecures, bus, and device being used, though the principles are the same everywhere.

Using I/O Memory

- According to the computer platform and bus being used, I/O memory may or may not be accessed through page tables.
- Whether or not ioremap() is required to access I/O memory, direct use of pointers to I/O memory is a discouraged practice.
- The wrapper functions used to access I/O memory are both safe on all platforms and optimized away whenever straight pointer dereferencing can perform the operation.

Directly Mapped Memory

- Several computer platforms reserve part of their memory address space for I/O locations, and automatically disable memory management for any (virtual) address in that memory range.
- Two address ranges, 512 MB each, of MIPS processors are directly mapped to physical addresses.

Directly Mapped Memory

- Memory access to either of those address ranges bypasses the MMU, and any access to one of those ranges bypasses the cache as well.
- A section of these 512 MB is reserved for peripheral devices, and drivers can access their I/O memory directly by using the noncached address range.

Directly Mapped Memory

- When access to directly mapped I/O memory area, you shouldn't dereference your I/O pointers.
- To write code that will work across systems and kernel versions, you must avoid direct accesses and instead use wrapper function.

Directly Mapped Memory

- In modern versions of the kernel, the wrapper functions are available across all architectures.
- As a device driver writer, you needn't worry about how the wrapper functions work.

Software-Mapped I/O Memory

- Physical address can be either hardwired (ISA) in the device or assigned by system firmware at boot time (PCI).
- Addresses of PCI devices are assigned by system software and written to device memory.

Software-Mapped I/O Memory

- For software to access I/O memory, they must be a way to assign a virtual address to the device.
  - The role of *ioremap()*.
- Once equipped with *ioremap()*, a device driver can access any I/O memory address, whether it's directly mapped to virtual address space or not.

Software-Mapped I/O Memory

- In situations in which all of I/O memory is already visible through noncacheable address.
- Under Linux 2.0, *ioremap()* refused to remap any non-page-aligned memory region.

ISA

- The memory range (384-KB) between 640-KB (0xA0000) and 1-MB (0x100000) on x86.
- This memory range belongs to the non-directly-mapped class of memory.
- Mapping the physical ISA addresses into kernel virtual addresses in order to access to ISA memory..

ISA

- It may be necessary simply to see if a device is present at a given address or not.
- It is necessary to actually probe memory to see what is there.
  - RAM is mapped to the address.
  - ROM is there.
  - The area is free.

ISA

- Detecting memory doesn't cause collisions with other devices, as long as you take care to restore any byte you modified while you were probing.
- It is worth noting that it's always possible that writing to another device's memory will cause that device to do something undesirable.

LINUX DEVICE DRIVERS, 2nd Edition. O'REILLY.

-- Interrupt Handling

Installing an Interrupt Handler

- If your device generates interrupts at truly random times, you should set *SA_SAMPLE_RANDOM* flag.
- The interrupt handler can be installed either at driver initialization or when the device is first opened.

Installing an Interrupt Handler

- Although installing the interrupt handler from within the module's initialization function might sound like a good idea, it actually isn't.
- Because the number of interrupt lines is limited, you don't wan to waste them.

Installing an Interrupt Handler

- Requesting the interrupt at device open, on the other hand, allows some sharing of resources.
- The correct place to call request_irq is when the device is first opened, before the hardware is instructed to generate interrupts.
- The place to call free_irq is the last time the device is closed, after the hardware is told not to interrupt the processor any more.

The /proc Interface

- */proc/interrupts* is not architecture dependent, whereas */proc/stat* is: the number of fields depends on the hardware underlying the kernel.

Autodetecting the IRQ Number

- Autodetection of the interrupt number is a basic requirement for driver usability.
- Sometimes autodetection depends on the knowledge that some devices feature a default behavior that rarely, if ever, changes.
- In this case, the driver might assume that the default values apply.

Autodetecting the IRQ Number

- Some devices are more advanced in design and simply "announce" which interrupt they're going to use.
- In this case, the driver retrieves the interrupt number by reading a status byte from one of the device's I/O ports or PCI configuration space.
- When the target device is one that has the ability to tell the driver which interrupt it is going to use, autodetecting the IRQ number just means probing the device, with no additional work required to probe the interrupt.

Kernel-assisted probing

- The Linux kernel offers a low-level facility for probing the interrupt number.
- It only works for nonshared interrupts.
- The programmer should be careful to enable interrupts on the device after the call to *probe_irq_on* and to disable them before calling *probe_irq_off*.

Kernel-assisted probing

- Depending on the speed of your processor, you may have to wait for a brief period to give the interrupt time to actually be delivered.

Do-it-yourself probing

- Enable all unused interrupts, then wait and see what happens.
- We can, however, exploit our knowledge of the device.
- To probe for all interrupts, you have to probe from IRQ 0 to IRQ *NR_IRQS*-1, where *NR_IRQS* is defined in *<asm/irq.h>* and is platform dependent.

Fast and Slow Handlers

- Older versions of the Linux kernel took great pains to distinguish between “fast” and “slow” interrupts.
- In modern kernels most of the differences between fast and slow interrupts have disappeared.
- There remains only one: fast interrupts (those that were requested with the SA_INTERRUPT flag) are executed with all other interrupts disabled on the current processor.

Fast and Slow Handlers

- On modern systems, *SA_INTERRUPT* is only intended for use in a few, specific situations (such as timer interrupts).
- Unless you have a strong reason to run your interrupt handler with other interrupts disabled, you should not use *SA_INTERRUPT*.

The internals of interrupt handling on the x86

- Probing for IRQs is done by setting the *IRQ_WAITING* status bit for each IRQ that currently lacks a handler.
- *probe_irq_off*, when called by a driver, need only search for the IRQ that no longer has *IRQ_WAITING* set.

Implementing a Handler

- The only peculiarity is that a handler runs at interrupt time and therefore suffers some restrictions on what it can do. (same with task queues)
- A handler can't transfer data to or from user space, because it doesn't execute in the context of a process.

Implementing a Handler

- Handlers also cannot do anything that would sleep, such as calling *sleep_on*, allocating memory with anything other than *GFP_ATOMIC*, or locking a semaphore.
- Handlers cannot call *schedule*.

Implementing a Handler

- The role of an interrupt handler is to give feedback to its device about interrupt reception and to read or write data according to the meaning of the interrupt being serviced.
- A typical task for an interrupt handler is awakening process sleeping on the device if the interrupt signals the event they're waiting for.

Implementing a Handler

- The programmer should be careful to write a routine that executes in a minimum of time, independent of its being a fast or slow handler.
- If a long computation needs to be performed, the best approach is to use a tasklet or task queue to schedule computation at a safer time.

Enabling and Disabling Interrupts

- Calling any of these functions may update the mask for the specified irq in the programmable interrupt controller (PIC), thus disabling or enabling IRQs across all processors.
- It is possible to call these functions from an interrupt handler, but enabling your own IRQ while handling it is not usually good practice.

Tasklets and Bottom-Half Processing

- One of the main problems with interrupt handling is how to perform longish tasks within a handler.
- Linux (along with many other systems) resolves this problem by splitting the interrupt handler into two halves.

Tasklets and Bottom-Half Processing

- The so-called top half is the routine that actually responds to the interrupt – the one you register with request_irq.
- The bottom half is a routine that is scheduled by the top half to be executed later, at a safer time.
- The big difference between the top-half handler and the bottom half is that all interrupts are enabled during execution of the bottom half.

Tasklets and Bottom-Half Processing

- The top half saves device data to a device-specific buffer, schedules its bottom half, and exits: this is very fast.
- This setup permits the top half to service a new interrupt while the bottom half is still working.
- Every serious interrupt handler is split this way.

Tasklets and Bottom-Half Processing

- All of the restrictions that apply to interrupt handlers also apply to bottom halves.
- Tasklets were introduced late in the 2.3 development series; they are now the preferred way to do bottom-half processing, but they are not portable to earlier kernel versions.

Tasklets and Bottom-Half Processing

- The older bottom-half (BH) implementation exists in even very old kernels, though it is implemented with tasklets in 2.4.

Tasklets

- Tasklets are also guaranteed to run on the same CPU as the function that first schedules them.
- An interrupt handler can thus be secure that a tasklet will not begin executing before the handler has completed.
- Another interrupt can certainly be delivered while the tasklet is running, so locking between the tasklet and the interrupt handler may still be required.

The BH Mechanism

- All BH bottom halves are predefined in the kernel, and there can be a maximum of 32 of them.
- Since they are predefined, bottom halves cannot be used directly by modules.
- Whenever some code wants to schedule a bottom half for running, it calls *mark_bh*.

Interrupt Sharing

- There is nothing in the design of the hardware itself that says that interrupt lines cannot be shared.
- The problems are on the software side.
- Suitably aware drivers for ISA devices can also share an IRQ line.
- With a level-triggered interrupt line, the peripheral device asserts the IRQ signal until software clears the pending interrupt.

Interrupt Sharing

- This behavior is safe with regard to sharing but may lead to lockup if a driver fails to clear its interrupt source.
- Whenever possible, it's better to support sharing because it presents fewer problems for the final user.

Installing a Shared Handler

- When a shared interrupt is requested, *request_irq* succeeds if either the interrupt line is free or any handlers already registered for that line have also specified that the IRQ is to be shared.
- A shared handler must be able to recognize its own interrupts, and should quickly exit when its own device has not interrupted.

Installing a Shared Handler

- No probing function is available for shared handlers.
- The only available technique for probing shared lines, then, is the do-it-yourself way.
- A driver using a shared handler needs to be careful about one more thing: it can't play with *enable_irq* or *disable_irq*.

Interrupt-Driven I/O

- Whenever a data transfer to or from the managed hardware might be delayed for any reason, the driver writer should implement buffering.
- A good buffering mechanism leads to *interrupt-driven I/O*.

Interrupt-Driven I/O

- For interrupt-driven data transfer to happen successfully, the hardware should be able to generate interrupts with the following 2 semantics.
- First, for input, the device interrupts the processor when new data has arrived and is ready to be retrieved by the system processor.

Interrupt-Driven I/O

- The actual actions to perform depend on whether the device uses I/O ports, memory mapping, or DMA.
- Second, for output, the device delivers an interrupt either when it is ready to accept new data or to acknowledge a successful data transfer.

Interrupt-Driven I/O

- Memory-mapped and DMA-capable devices usually generate interrupts to tell the system they are done with the buffer.
- Interrupt-driven I/O introduces the problem of synchronizing concurrent access to shared data items and all the issues related to race conditions.

Race Conditions

- The most common ways of protecting data from concurrent access are as following 3 situations.
- First, using a circular buffer and avoiding shared variables.
- Second, using spinlocks to enforce mutual exclusion.

Race Conditions

- Third, using lock variables that are atomically incremented and decremented.
- Semaphores may not be used in interrupt handlers.

Using Circular Buffers

- Using a circular buffer is an effective way of handling concurrent-access problems.
- If data is written at interrupt time, you must be careful when accessing head multiple times.
- You should either declare it as *volatile* or use some sort of locking.

Using Spinlocks

- A function that holds a spinlock for too long can waste much time because other CPUs are forced to wait.
- If you are using spinlocks in interrupt handlers, however, you must use the IRQ-disabling versions.

Using Lock Variables

- The kernel provides a set of functions that may be used to provide atomic (noninterruptible) access to variables.
- The Linux kernel exports two sets of functions to deal with locks: bit operations and access to the “atomic” data type.

Bit operations

- The whole operation happens in a single step, no interrupt (or other processor) can interfere.
- As mentioned, however, it is better to use spinlocks in new code, unless you need to perform useful work while waiting for the lock to be released.

Atomic integer operations

- The facility offered by *atomic.h* is much stronger than the bit operations just described.
- The operations are very fast because they compile to a single machine instruction whenever possible.

Going to Sleep Without Races

- The way to solve this problem is to go halfway to sleep before performing the test.
- The idea is that the process can add itself to the wait queue, declare itself to be sleeping, and then perform its tests.

LINUX DEVICE DRIVERS, 2nd Edition. O'REILLY.

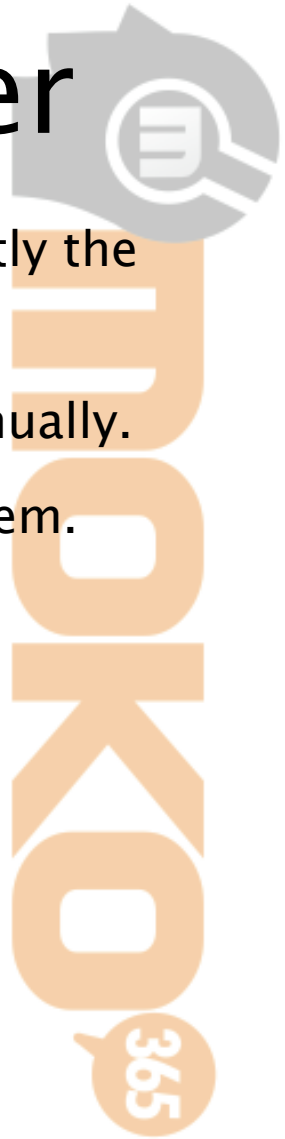
-- Loading Block Drivers

Registering the Driver

- *register_blkdev* uses a structure of type *block_device_operations* instead.
- The structure is still sometimes referred to by the name *fops* in block drivers; we'll call it *bdops* to be more faithful to what the structure is and to follow the suggested naming.

Registering the Driver

- The *open*, *release*, and *ioctl* methods listed here are exactly the same as their char device counterparts.
- Block drivers must still maintain their usage count manually.
- All I/O to block device is normally buffered by the system.



Registering the Driver

- User processes do not perform direct I/O to these devices.
- Kernel provides a single set of read and write functions for block devices, and drivers do not need to worry about them.
- A block driver must eventually provide some mechanism for actually doing block I/O to a device.

Registering the Driver

- The method used for these I/O operations is called *request*.
- The *request* method handles both read and write operations.
- It is associated with the queue of pending I/O operations for the device.
- By default, there is one such queue for each major number.

Registering the Driver

- The default value for the hardware sector size is 512 bytes.
- These arrays define the number of sectors to be read in advance by the kernel when a file is being read sequentially.
- Reading data before a process asks for it helps system performance and overall throughput.

Registering the Driver

- A slower device should specify a bigger read-ahead value, while fast devices will be happy even with a smaller value.
- The primary difference between the two arrays is these two.
- First, *read_ahead* is applied at the block I/O level and controls how many blocks may be read sequentially from the disk ahead of the current request.

Registering the Driver

- Second, *max_readahead* works at the filesystem level and refers to blocks in the file, which may not be sequential on disk.
- Kernel development is moving toward doing read ahead at the filesystem level, rather than at the block I/O level. In the 2.4 kernel, however, read ahead is still done at both levels, so both of these arrays are used.

Registering the Driver

- There is one read_ahead[] value for each major number, and it applies to all its minor numbers.
- max_readahead, instead, has a value for every device. The values can be changed via the driver's ioctl method.
- max_sections[][] should normally be set to the largest transfer that your hardware can handle.

Registering the Driver

- In the 2.4.0 kernel, *register_disk* does nothing when invoked in this manner.
- The real purpose of *register_disk* is to set up the partition table.
- All block drivers, however, make this call whether or not they support partitions, indicating that it may become necessary for all block devices in the future.

Registering the Driver

- Here, the call to *fsync_dev* is needed to free all references to the device that the kernel keeps in various caches.
- *fsync_dev* is the implementation of *block_fsync*, which is the *fsync* "method" for block devices.

The Header File blk.h

- All block drivers should include the header file *<linux/blk.h>*.
- It provides functions for dealing with the I/O request queue.
- *DEVICE_INTR* is used to declare a pointer variable that refers to the current bottom-half handler.

The Header File blk.h

- Modern drivers no longer use *DEVICE_ON* and *DEVICE_OFF* macros, and *DEVICE_ON* does not even get called anymore.

Handling Requests: A Simple Introduction

- The *request* function should perform the following four tasks for each request in the queue.
- First, check the validity of the request.
- Second, perform the actual data transfer.
- Third, clean up the request just processed.
- Fourth, loop back to the beginning to consume the next request.

Handling Requests: A Simple Introduction

- The *INIT_REQUEST* macro performs a *return* when the request queue is empty.
- The *CURRENT* macro always describes the request to be processed.
- The *request* function has one very important constraint: it must be atomic. It must not sleep while carrying out its tasks.

Performing the Actual Data Transfer

- By accessing the fields in the *request* structure, usually by way of *CURRENT*, the driver can retrieve all the information needed to transfer data between the buffer cache and the physical block device.
- A single *request* function deals with all the minor numbers: *rq_dev* can be used to extract the minor device being acted upon.

Performing the Actual Data Transfer

- The driver should refer to *current_nr_sectors* and *ignore nr_sectors*.

The I/O Request Queue

- The management of this queue is complicated; the performance of the system depends on how it is done.
- With disks, the amount of time required to transfer a block of data is typically quite small.
- The amount of time required to position the head (seek) to do that transfer, however, can be very large.

The I/O Request Queue

- The Linux kernel works to minimize the number and extent of the seeks performed by the device.
- The *request_queue* structure looks somewhat like *file_operations* and other such objects.

The request structure and the buffer cache

- Linux maintains a buffer cache, a region of memory that is used to hold copies of blocks stored on disk.
- Blocks that are adjacent on disk are almost certainly not adjacent in memory.
- Every block passed to a driver's request function either lives in the buffer cache, or, on rare occasion, lives elsewhere but has been made to look as if it lived in the buffer cache.

The request structure and the buffer cache

- All of the buffer heads attached to a single request will belong to a series of adjacent blocks on the disk.
- A request is, in a sense, a single operation referring to a (perhaps long) group of blocks on the disk.
- This grouping of blocks is called clustering.

The I/O request lock

- In Linux 2.2 and 2.4, all request queues are protected with a single global spinlock called *io_request_lock*.
- The kernel always calls the *request* function with the *io_request_lock* held.
- A driver that holds this lock for a long time may well slow down the system as a whole.

The I/O request lock

- Well-written block drivers often drop this lock as soon as possible.
- Block drivers that drop the *io_request_lock* must be written with a couple of two important things in mind, however.
- First, the *request* function must always reacquire this lock before returning, since the calling code expects it to still be held.

The I/O request lock

- Second, as soon as the *io_request_lock* is dropped, the possibility of reentrant calls to the *request* function is very real; the function must be written to handle that eventuality.
- A variant of this latter case can also occur if your *request* function returns while an I/O request is still active.

The I/O request lock

- Some drivers handle *request* function reentrancy by maintaining an internal request queue.

How the blk.h macros and functions work

- Getting reasonable performance requires a deeper understanding of how the queue works.
- The fields of the request structure that we looked at earlier – sector, current_nr_sectors, and buffer – are really just copies of the analogous information stored in the first buffer_head structure on the list.

How the blk.h macros and functions work

- A *request* function that uses this information from the *CURRENT* pointer is just processing the first of what might be many buffers within the request.

Clustered Requests

- Clustering, as mentioned earlier, is simply the practice of joining together requests that operate on adjacent blocks on the disk.
- There are two advantages to doing things this way.
- First, clustering speeds up the transfer.
- Second, clustering can also save some memory in the kernel by avoiding allocation of redundant request structures.

Clustered Requests

- To take advantage of clustering, a block driver must look directly at the list of *buffer_head* structures attached to the request.
- When the I/O on each buffer completes, your driver should notify the kernel by calling the buffer's I/O completion routine: *bh->b_end_io(bh, status)*.

Clustered Requests

- You also, of course, need to remove the *request* structure for the completed operations from the queue.
- In the latter case, the *b_end_io* function should not have been called on that last buffer, since *end_request* will make that call.

Multiqueue Block Drivers

- The 2.4 kernel makes life easier by allowing the driver to set up independent queues for each device.
- Most high-performance drivers take advantage of this multiqueue capability.
- Th operate in a multiqueue mode, a block driver must define its own request queues.

Multiqueue Block Drivers

- Devices using the default queue have no such function, but multiqueue devices must implement it.
- Like the *request* function, *sbull_find_queue* must be atomic (no sleeping allowed).
- Each queue has its own *request* function, though usually a driver will use the same function for all of its queues.

Doing Without the Request Queue

- Not all block devices benefit from the request queue.
- Other types of block devices also can be better off without a request queue.
- Block drivers that need more control over request queuing, however, can replace that function with their own “make request” function.

Doing Without the Request Queue

- A RAM-disk driver, instead, can execute the I/O operation directly.
- The “make request” function must arrange to transfer the given block, and see to it that the *b_end_io* function is called when the transfer is done.
- Often a driver-specific make request function will not actually transfer the data.

How Mounting and Unmounting Works

- Block devices differ from char devices and normal files in that they can be mounted on the computer's filesystem.
- When a filesystem is counted, there is no process holding that *file* structure.
- When the kernel mounts a device in the filesystem, it invokes the normal *open* method to access the driver.

The ioctl Method

- Block drivers share a number of common *ioctl* commands that most drivers are expected to support.
- The 2.4 kernel has provided a function, *blk_ioctl*, that may be called to implement the common commands.
- Often the only ones that must be implemented in the driver itself are *BLKGETSIZE* and *HDIO_GETGEO*.

Removable Devices

- The operations in question are *check_media_change* and *revalidate*.
- The former is used to find out if the device has changed since that last access, and the latter reinitializes the driver's status after a disk change.

check_media_change

- When the device is removable but there is no way to know if it changed, return 1 is a safe choice.
- This is the behavior of the IDE driver when dealing with removable disks.

Revalidation

- The validation function is called when a disk change is detected.
- It is also called by the various stat system calls implemented in version 2.1 of the kernel.
- The action performed by revalidate is device specific, but revalidate usually updates the internal status information to reflect the new device.

Extra Care

- The kernel automatically calls *check_disk_change* at *mount* time, but not at *open* time.
- If the driver keeps status information about removable devices in memory, it should call the kernel *check_disk_change* function when the device is first opened.

The Generic Hard Disk

- Every partitionable device needs to know how it is partitioned. The information is available in the partition table.
- Kernel offers “generic hard disk” support usable by all block drivers.
- This structure describes the layout of the disk(s) provided by the driver; the kernel maintains a global list of such structures.

Partition Detection

- When a module initializes itself, it must set things up properly for partition detection.
- The driver should also include its *gendisk* structure on the global list.
- There is no kernel-supplied function for adding *gendisk* structures; it must be done by hand.

Partition Detection

- The only thing the system does with this list is to implement */proc/partitions*.
- Fixed disks might read the partition table only at module initialization time and when *BLKRRPART* is invoked.
- Drivers for removable drives will also need to make this call in the *revalidate* method.

Partition Detection

- When a partitionable module is unloaded, the driver should arrange for all the partitions to be flushed, by calling fsync_dev for every supported major/minor pair.
- All of the relevant memory should be freed as well, of course.
- It is also necessary to remove the gendisk structure from the global list.

The Device Methods for spull

- We need to make use of the partition information stored in the gendisk->part array by register_disk.
- This array is made up of hd_struct structures, and is indexed by the minor number.

The Device Methods for spull

- The *hd_struct* has two fields of interest: *start_sect* tells where a given partition starts on the disk, and *nr_sects* gives the size of that partition.
- *ioctl* should access specific information for each partition.

Interrupt-Driven Block Drivers

- When the driver is interrupt driven, the *request* function spawns a data transfer and returns immediately without calling *end_request*.
- The top-half or the bottom-half interrupt handler calls *end_request* when the device signals that the data transfer is complete.

Interrupt-Driven Block Drivers

- The *request* function for an interrupt-driven device instructs the hardware to perform the transfer and then returns; it does not wait for the transfer to complete.
- A real driver would almost certainly defer much of this work and run it from a task queue or tasklet.

LINUX DEVICE DRIVERS, 2nd Edition. O'REILLY.

-- MMAP and DMA

The Memory Map and struct page

- The kernel has used logical addresses to refer to explicit pages of memory. Logical addresses are not available for high memory.
- This data structure is used to keep track of just about everything the kernel needs to know about physical memory; there is one *struct page* for each physical page on the system.

The Memory Map and struct page

- The kernel maintains one or more arrays of *struct page* entries, which track all of the physical memory on the system.
- Code that is meant to be portable should avoid direct access to the array whenever possible.

Page Tables

- Note that Linux uses a three-level system even on hardware that only supports two levels of page tables or hardware that uses a different way to map virtual addresses to physical ones.
- Each process has its own page directory (PGD), and there is one for kernel space as will.

Page Tables

- A *pte_t* contains the physical address of the data page.
- The kernel doesn't need to worry about doing page-table lookups during normal program execution, because they are done by the hardware.
- The kernel must arrange things so that the hardware can do its work.

Page Tables

- Device drivers, too, must be able to build page tables and handle faults when implementing *mmap*.
- Page-table lookup begins with a pointer to *struct mm_struct*. The pointer associated with the memory map of the current process is *current->mm*, while the pointer to kernel space is described by *&init_mm*.

Page Tables

- This macro returns a boolean value that indicates whether the data page is currently in memory.
- Pages may be absent, of course, if the kernel has swapped them to disk or if they have never been loaded.

Page Tables

- Each process in the system has a *struct mm_struct* structure, which contains its page tables.
- It also contains a spinlock called *page_table_lock*, which should be held while traversing or modifying the page tables.

Virtual Memory Areas

- The kernel needs a higher-level mechanism to handle the way a process sees its memory.
- The memory map of a process is made up of the following three areas.
- First, an area for the program's executable code (often called text).

Virtual Memory Areas

- Second, one area each for data, including initialized data (that which has an explicitly assigned value at the beginning of execution), uninitialized data (BSS), and the program stack.
- Third, one area for each active memory mapping.

Virtual Memory Areas

- A driver that implements the *mmap* method needs to fill a VMA structure in the address space of the process mapping the device.
- These fields in *struct vm_area_struct* may be used by device drivers in their *mmap* implementation.

The mmap Device Operation

- Mapping a device means associating a range of user-space addresses to device memory. Whenever the program reads or writes in the assigned address range, it is actually accessing the device.
- The mapped area must be a multiple of PAGE_SIZE and must live in physical memory starting at an address that is a multiple of PAGE_SIZE.

The mmap Device Operation

- There are sound advantages to using *mmap* when it's feasible to do so.
- Most PCI peripherals map their control registers to a memory address, and a demanding application might prefer to have direct access to the registers instead of repeatedly having to call *ioctl* to get its work done.

The mmap Device Operation

- With *mmap*, the kernel performs a good deal of work before the actual method is invoked.
- Much of the work has thus been done by the kernel; to implement *mmap*, the driver only has to build suitable page tables for the address range and, if necessary, replace *vma->vm_ops* with a new set of operations.

The mmap Device Operation

- There are two ways of building the page tables.
- First, doing it all at once with a function called *remap_page_range*.
- Second, doing it a page at a time via the *nopage* VMA method.

Using remap_page_range

- The job of building new page tables to map a range of physical addresses is handled by *remap_page_range*.
- References to device memory should not be cached by the processor.

Using remap_page_range

- Often the system BIOS will set things up properly, but it is also possible to disable caching of specific VMAs via the protection fields.
- Disabling caching at this level is highly processor dependent.

Adding VMA Operations

- In particular, the *open* method will be invoked anytime a process forks and creates a new reference to the VMA.
- The kernel will not call the driver's *release* method as long as a VMA remains open, so the usage count will not drop to zero until all references to the VMA are closed.

Mapping Memory with nopage

- When as user process attempts to access a page in a VMA that is not present in memory, the associated nopage function is called.
- The nopage function must locate and return the struct page pointer that refers to the page the user wanted.
- This function must also take care to increment the usage count for the page it returns by calling the get_page macro.

Mapping Memory with nopage

- One situation in which the *nopage* approach is useful can be brought about by the *mremap* system call, which is used by applications to change the bounding addresses of a mapped region.
- The Linux implementation of *mremap* doesn't notify the driver of changes in the mapped area.

Mapping Memory with nopage

- Actually, it does notify the driver if the size of the area is reduced via the *unmap* method, but no callback is issued if the area increases in size.
- The *nopage* method, therefore, must be implemented if you want to support the *mremap* system call.

Mapping Memory with nopage

- If the *nopage* method is left *NULL*, kernel code that handles page faults maps the zero page to the faulting virtual address.
- The zero page is a copy-on-write page that reads as zero and that is used, for example, to map the BSS segment.

Mapping Memory with nopage

- If a process extends a mapped region by calling mremap, and the driver hasn't implemented nopage, it will end up with zero pages instead of a segmentation fault.
- Note that this implementation will work for ISA memory regions but not for those on the PCI bus. PCI memory is mapped above the highest system memory, and there are no entries in the system memory map for those addresses.

Remapping Specific I/O Regions

- If your driver has no *nopage* method, it will never be notified of this extension, and the additional area will map to the zero page.

Remapping RAM

- nopage will not work with PCI memory areas, so extension of PCI mappings is not possible.
- In Linux, a page of physical addresses is marked as “reserved” in the memory map to indicate that it is not available for memory management.
- On the PC, for example, the range between 640 KB and 1 MB is marked as reserved, as are the pages that host the kernel code itself.

Remapping RAM

- An interesting limitation of *remap_page_range* is that it gives access only to reserved pages and physical addresses above the top of physical memory.
- Reserved pages are locked in memory and are the only ones that can be safely mapped to user space; this limitation is a basic requirement for system stability.

Remapping RAM with the nopage method

- The way to map real RAM to user space is to use *vm_ops->nopage* to deal with page faults one at a time.
- *scullp* doesn't release device memory as long as the device is mapped.

Remapping RAM with the nopage method

- To maximize allocation performance, the Linux kernel maintains a list of free pages for each allocation order, and only the page count of the first page in a cluster is incremented by get_free_pages and decremented by free_pages.
- The mmap method is disabled for a scullp device if the allocation order is greater than zero, because nopage deals with single pages rather than clusters of pages.

Remapping RAM with the nopage method

- Code that is intended to map RAM according to the rules just outlined needs to implement *open*, *close*, and *nopage*; it alos needs to access the memory map to adjust the page usage counts.

Remapping Virtual Addresses

- A true virtual address, remember, is an address returned by a function like *vmalloc* or *kmap* – that is, virtual address mapped in the kernel page tables.
- Most of the work of *vmalloc* is building page tables to access allocated pages as a continuous address range.

Remapping Virtual Addresses

- The *nopage* method, instead, must pull the page tables back apart in order to return a *struct page* pointer to the caller.
- *remap_page_range* is already usable for building new page tables that map I/O memory to user space.

The kiobuf Interface

- As of version 2.3.12, the Linux kernel supports an I/O abstraction called the kernel I/O buffer, or kiobuf.
- The kiobuf interface is intended to hide much of the complexity of the virtual memory system from device drivers.
- Many features are planned for kiobufs, but their primary use in the 2.4 kernel is to facilitate the mapping of user-space buffers into the kernel.

The kiobuf Structure

- The key to the kiobuf interface is the maplist array.
- Functions that operate on pages stored in a kiobuf deal directly with the page structures – all of the virtual memory system overhead has been moved out of the way.
- This implementation allows drivers to function independent of the complexities of memory management.

Mapping User-Space Buffers and Raw I/O

- Unix systems have long provided a “raw” interface to some devices which performs I/O directly from a user-space buffer and avoids copying data through the kernel.
- The Linux kernel has traditionally not provided a raw interface, for a number of reasons.

Mapping User-Space Buffers and Raw I/O

- It is worth noting that in Linux systems, there is no need for block drivers to provide this sort of interface.
- The raw device, in *drivers/char/raw.c*, provides this capability in an elegant, general way for all block devices.
- The block drivers need not even know they are doing raw I/O.

Mapping User-Space Buffers and Raw I/O

- Raw I/O to a block device must always be sector aligned, and its length must be a multiple of the sector size.
- Other kinds of devices, such as tape drives, may not have the same constraints.
- Raw device access is often not the best approach, but for some applications it can be a major improvement.

Direct Memory Access and Bus Mastering

- DMA is the hardware mechanism that allows peripheral components to transfer their I/O data directly to and from main memory without the need for the system processor to be involved in the transfer.

Overview of a DMA Data Transfer

- A variant of the asynchronous approach is often seen with network cards.
- The processing steps in all of these cases emphasize that efficient DMA handling relies on interrupt reporting.

Allocating the DMA Buffer

- The main problem with the DMA buffer is that when it is bigger than one page, it must occupy contiguous pages in physical memory because the device transfers data using the ISA or PCI system bus, both of which carry physical addresses.

Do-it-yourself allocation

- Perform aggressive allocation until you are able to get enough consecutive pages to make a buffer.
- We strongly discourage this allocation technique if there's any other way to achieve your goal.

Bus Addresses

- DMA-based hardware uses bus, rather than physical, addresses.
- Sometimes the interface bus is connected through bridge circuitry that maps I/O addresses to different physical addresses.

DMA on the PCI Bus

- It handles the details of buffer allocation and can deal with setting up the bus hardware for multipage transfers on hardware that supports them.

Dealing with difficult hardware

- The function *pci_dma_supported* should be called for any device that has addressing limitations.

DMA mappings

- A DMA mapping is a combination of allocating a DMA buffer and generating an address for that buffer that is accessible by the device.
- Setting up a useful address for the device may also, in some cases, require the establishment of a *bounce buffer*.

DMA mappings

- Bounce buffers are created when a driver attempts to perform DMA on an address that is not reachable by the peripheral device – a high-memory address, for example.
- The PCI code distinguishes between two types of DMA mappings, depending on how long the DMA buffer is expected to stay around.

Consistent DMA mappings

- These exist for the life of the driver.
- A consistently mapped buffer must be simultaneously available to both the CPU and the peripheral.
- The buffer should also, if possible, not have caching issues that could cause one not to see updates made by the other.

Streaming DMA mappings

- These are set up for a single operation.
- The kernel developers recommend the use of streaming mappings over consistent mappings whenever possible.
- Each streaming DMA mapping uses one or more mapping registers on the bus
- Streaming mappings can be optimized in ways that are not available to consistent mappings.

Setting up streaming DMA mappings

- There are three important rules that apply to streaming DMA mappings.
- First, the buffer must be used only for a transfer that matches the direction value given when is was mapped.
- Second, once a buffer has been mapped, it belongs to the device, not the processor.

Setting up streaming DMA mappings

- Only after *pci_unmap_single* has been called is it safe for the driver to access the contents of the buffer. This rule implies that a buffer being written to a device cannot be mapped until it contains all the data to write.
- Third, the buffer must not be unmapped while DMA is still active.

Setting up streaming DMA mappings

- When a buffer is mapped for DMA, the kernel must ensure that all of the data in that buffer has actually been written to memory.
- Data written to the buffer by the processor after the flush may not be visible to the device.
- The bounce buffer is just a separate region of memory that is accessible to the device.

Setting up streaming DMA mappings

- If a buffer is mapped with a direction of PCI_DMA_TODEVICE, and a bounce buffer is required, the contents of the original buffer will be copied as part of the mapping operation.
- PCI_DMA_BIDIRECTIONAL bounce buffers are copied before and after the operation, which is often an unnecessary waste of CPU cycles.

Setting up streaming DMA mappings

- A driver will need to access the contents of a streaming DMA buffer without unmapping it.
- pci_sync_single has been provided to make this possible.
- This function should be called before the processor accesses a PCI_DMA_FROMDEVICE buffer, and after an access to a PCI_DMA_TODEVICE buffer.

Scatter-gather mappings

- Scatter-gather mappings are a special case of streaming DMA mappings.
- One reason is that some smart devices can accept a *scatterlist* of array pointers and lengths and transfer them all in one DMA operation.
- “zero-copy” networking is easier if packets can be built in multiple pieces.

Scatter-gather mappings

- Another reason to map scatterlists as a whole is to take advantage of systems that have mapping registers in the bus hardware.
- Physically discontiguous pages can be assembled into a single, contiguous array from the device's point of view.

Scatter-gather mappings

- This technique works only when the entries in the scatterlist are equal to the page size in length (except the first and last), but when it does work it can turn multiple operations into a single DMA and speed things up accordingly.
- If a bounce buffer must be used, it makes sense to coalesce the entire list into a single buffer (since it is being copied anyway).

Scatter-gather mappings

- Your driver should transfer each buffer returned by pci_map_sg.
- The address and length of the buffers to transfer may be different from what was passed in to pci_map_sg.
- Note that nents must be the number of entries that you originally passed to pci_map_sg, and not the number of DMA buffers that function returned to you.

Scatter-gather mappings

- Scatter-gather mappings are streaming DMA mappings, and the same access rules apply to them as to the single variety.
- If you must access a mapped scatter-gather list, you must synchronize it first by calling *pci_dma_sync_sg*.

How ARM support PCI DMA

- ARM, MIPS, PowerPC and IA-32 (x86) supports the PCI DMA interface, but it is mostly a false front. There are no mapping registers in the bus interface, so scatterlists cannot be combined and virtual addresses cannot be used.
- There is no bounce buffer support, so mapping of high-memory addresses cannot be done. The mapping functions on the ARM architecture can sleep, which is not the case for the other platforms.

DMA for ISA Devices

- The ISA bus allows for two kinds of DMA transfers: native DMA and ISA bus master DMA.
- Native DMA uses standard DMA-controller circuitry on the motherboard to drive the signal lines on the ISA bus.
- ISA bus master DMA, on the other hand, is handled entirely by the peripheral device.

DMA for ISA Devices

- The driver has little to do: it provides the DMA controller with the direction, bus address, and size of the transfer.
- It also talks to its peripheral to prepare it for transferring the data and responds to the interrupt when the DMA is over.
- The maximum transfer size is therefore 64 KB for the slave controller and 128 KB for the master.

Registering DMA usage

- We recommend that you take the same care with DMA channels as with I/O ports and interrupt lines; requesting the channel at *open* time is much better than requesting it from the module initialization function.
- Delaying the request allows some sharing between drivers.

Registering DMA usage

- We also suggest that you request the DMA channel after you've requested the interrupt line and that you release it before the interrupt.

Talking to the DMA controller

- The driver needs to configure the DMA controller either when *read* or *write* is called, or when preparing for asynchronous transfers.
- This latter task is performed either at *open* time or in response to an *ioctl* command, depending on the driver and the policy it implements.

Talking to the DMA controller

- The controller is protected by a spinlock, called *dma_spin_lock*.
- The spinlock should be held when using the functions described next. It should not be held during the actual I/O, however.
- A driver should never sleep when holding a spinlock.

Talking to the DMA controller

- The information that must be loaded into the controller is made up of three items: the RAM address, the number of atomic items that must be transferred (in bytes or words), and the direction of the transfer.
- The channel should be disabled before the controller is configured.

Talking to the DMA controller

- The only thing that remains to be done is to configure the device board.
- For configuring the board, the hardware manual is your only friend.