Writing Linux Device Drivers

Michael K. Johnson
Developer, Red Hat Software
johnsonm@redhat.com

From a talk given at Spring DECUS '95 in Washington, DC.

Introduction

Naturally, there is far more about writing Linux device drivers than can be covered in 50 minutes. Fortunately, I am not enough of an expert to get bogged down in details, so you stand a chance of getting a helpful overview. There is some documentation available on writing device drivers for Linux; my own *Linux Kernel Hackers' Guide* (the KHG) is the main source for beginners. However, the details change from time to time as Linux matures, and many other details simply are not documented yet. This means that I can give you a skeleton for your driver, and give you some advice, but writing the driver may be a little bit of an adventure.

If there is something that you need to do that isn't covered in this introductory tutorial, and which has been overlooked in the KHG, the next option is to look through other device drivers to see how they handle the problem. Chances are good that you are not the first person to encounter that particular problem. It is also likely that if you put some time into looking around and can't figure out what to do, you can find help on the linux-kernel mailing list or on the comp.os.linux.development.system Usenet group.

Overview

Linux is a clone of Unix. As in all versions of Unix, hardware devices are presented to normal programs as "special" files. Therefore, devices implement file semantics within the kernel. Because of this, it is worth taking a short look at how files in general are treated in Linux before attempting to understand how device drivers are written.

Files

The generic filesystems header file, `<linux/fs.h>`, defines several structures for accessing files. `super_block` holds basic information about each filesystem, and `super_operations` is a structure of pointers to functions which are associated with a filesystem's superblock. Through that structure are reached `inode_operations` and `file_operations`, the last defining functions that can be used to access files. In normal filesystems, there is one set of file operations for all files in the filesystem, but they do not attempt to define any operations on device special files. Instead, those devices define their own file operations functions and register their own `file_operations` structure with the VFS.

The VFS

The VFS is the common abbreviation for the Virtual Filesystem Switch. Generic filesystem operations are handled by generic filesystem code, and only when filesystem-dependent or device-dependent operations need to be done is the code for that specific filesystem or device actually called. The function needed is looked up in the proper instance of one of the `*_operations` structures and called. The VFS code is kept in the fs/ subdirectory of the Linux kernel source, and the code to the individual filesystems is kept in subdirectories of the fs/ subdirectory.

Operations

What I mean by "operations" may not be very clear at this point. An operation is something that needs to be done as a result of a system call, or buffer cache activity, or because of hardware irregularities. Nearly all operations are caused directly or indirectly by system calls, and so you can think of the VFS as code that translates raw system calls into filesystem operations.

Special files and filesystems

All versions of Unix have device special files. This concept has even been picked up by such primitive operating systems as Microsoft's DOS. However, the VFS is so flexible that not only can special files be created, special filesystems can to. Linux has a filesystem called the proc filesystem, or "procfs", which is essentially a special filesystem. The files in this filesystem are not stored on disk; they are instead generated on-the-fly from kernel data structures. (No, I'm not way off topic.) These files are very similar to hardware devices, because they generate files from non-file data and present it to the user in the shape of a file. They are, you could say, virtual devices designed to report on the state of the kernel.

File Operations

Knowing this, it should not be suprising that devices "export" their functionality to the VFS by registering a `file_operations` structure with the VFS. We will see exactly how this is done later. Here's the `file_operations` structure:

```
struct file_operations {
        int (*lseek) (struct inode *, struct file *, off_t, int);
        int (*read) (struct inode *, struct file *, char *, int);
        int (*write) (struct inode *, struct file *, char *, int);
        int (*readdir) (struct inode *, struct file *, struct dirent *, int);
        int (*select) (struct inode *, struct file *, int, select_table *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        int (*mmap) (struct inode *, struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        void (*release) (struct inode *, struct file *);
        int (*fsync) (struct inode *, struct file *);
        int (*fasync) (struct inode *, struct file *, int);
        int (*check_media_change) (dev_t dev);
        int (*revalidate) (dev_t dev);
};
```

Some of the names of these function pointers should look suspiciously like system calls with which you are familiar. `lseek()`, `read()`, `write()`, `readdir()`, `select()`, `ioctl()`, `mmap()`, `open()`, and `fsync()` all are called directly or indirectly by the system calls of the same name. `release()` is called on `close()` and when a file is closed by a process exiting, or calling `exec()` when close-on-exec is set on the file. `check_media_change()` and `revalidate()` are not really file operations, they are device operations, as can be seen by their arguments. `fasync()` is a bit unusual; it is called when `fcntl(fd, F_SETFL, FASYNC)` (or `~FASYNC`) is called; devices implementing this need to be aware when this change is made. The functions are provided with sensible defaults; most of the time, more than half of the functions are set to `NULL` because the VFS does the right thing without having to call the driver. You will see this in the skeleton driver presented.

Data Structures

The Linux kernel is monolithic. Only one thread of control created by a system call is active at any time. This means that device drivers do not need to lock their data structures as a general rule. The exception is that interrupt handling routines can run at any time, and data structures that are shared with interrupt handlers do need to be protected.

The Kernel Interface

By far the easiest way to develop almost any Linux device driver is as a run-time loadable kernel module. Written correctly, these modules can easily be used in their normal form as loadable modules, or re-compiled and linked with the rest of the kernel. The symbol `MODULE` is defined whenever a module is being compiled. The symbol `__KERNEL__` is always defined when compiling kernel code, even when compiling modules; it is used in the kernel include files so that only kernel code includes certain kernel-specific definitions. This is necessary because the kernel include files are used as part of the standard include file hierarchy.

I will start by presenting a very simple character device driver which implements a simple form of /dev/zero. Note that it does not deal with the memory-management uses of /dev/zero with which some of you may be familiar; this is intended to be a simple example that sends me on as few tangents as possible. All it does is allow writing of any values and reading all zero values. The code to do reading and writing takes 13 lines total; the rest of this file is a skeleton that anyone writing any device driver will find useful.

```
/* Compile with "gcc -O -DMODULE -D__KERNEL__ -c zero.c" */
```

All Linux kernel code should be compiled with optimization becuase it requires certain gcc extensions that are only activated with optimization turned on.

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

All device drivers should include `<linux/config.h>` before including any other file.

```
#ifdef MODULE
#include <linux/module.h>
#include <linux/version.h>
#else
#define MOD_INC_USE_COUNT
#define MOD_DEC_USE_COUNT
#endif
```

Kernel symbols that are exported to modules have their names "mangled" in a way similar to C++, so that changes in kernel structures will be noticed. This causes the module not to be loaded, because if a kernel structure is changed, loading an old module that has been compiled with a different version of the structure can damage system integrity. MOD_INC_USE_COUNT and MOD_DEC_USE_COUNT are documented later.

```
#include <linux/types.h>
#include <linux/fs.h>
#include <linux/mm.h>    /* for verify_area */
#include <linux/errno.h> /* for -EBUSY */
#include <asm/segment.h> /* for put_user_byte */
```

All of these are included by nearly every device driver. Real device drivers will of course include other include files as well.

```
static int zero_major;
```

All modules that dynamically allocate their major number, as this one does (that comes later), need to store their major number somewhere. For this simple driver, I use a `static int`. In a more complex driver that saves a lot of state, this might be part of a static structure.

```
static int read_zero(struct inode * node, struct file * file, char * buf, int count)
  int left;

  if (verify_area(VERIFY_WRITE, buf, count) == -EFAULT) return -EFAULT;
  for (left = count; left > 0; left--) {
    put_user_byte(0, buf);
    buf++;
  }
  return count;
}
```

Whenever the `read()` system call is called on this device, `read_zero()` is called. The `put_user_byte()` function puts a byte into user memory. It is not as easy as saying `*buf++ = '\000';` because the execution stream is in kernel memory space, and the pointer `buf` points to user-space memory. `read_zero()` is expected to return the number of bytes actually written into the read buffer.

Before actually writing the zeros into the buffer provided, we verify that the entire buffer is legal to write in, using a `verify_area()` call. This prevents us from generating kernel-space faults from the reading process if the reading process passes in a pointer to non-existent memory or a count that makes part of the buffer lie in non-existent memory.

```
static int write_zero(struct inode * inode, struct file * file, char * buf, int coun
  return count;
}
```

This function is called whenever the `write()` system call is called on this device. This function is expected to return the number of bytes written. `write_zero()` ignores its input and returns a successfully completed write operation.

```
static int lseek_zero(struct inode * inode, struct file * file, off_t offset,
                      int orig) {
```

```
  return file->f_pos=0;
}
```

The semantics of /dev/zero are unusual. Normally, the lseek function would check to make sure that the seek was in-bounds for the device and set `file->f_pos = offset`, or return an appropriate error. The reason that we implement this function at all is because lseek would otherwise fail if someone tried to open the device with STDIO in append (`"a"`) mode.

```
static int open_zero(struct inode *inode, struct file * file) {
  MOD_INC_USE_COUNT;
  return 0;
}
static void release_zero(struct inode *inode, struct file * file) {
  MOD_DEC_USE_COUNT;
}
```

These are included here only because this is a loadable module; there is nothing that needs to be allocated or deallocated when this device is opened or closed. However, if the module is in use, we would like to forbid removing the module, and this allows us to keep track of whether the module is in use or not.

```
static struct file_operations zero_fops = {
  lseek_zero,
  read_zero,
  write_zero,
  NULL,         /* no special zero_readdir */
  NULL,         /* no special zero_select */
  NULL,         /* no special zero_ioctl */
  NULL,         /* no special zero_mmap */
  open_zero,
  release_zero,
  NULL,         /* no special fsync */
  NULL,         /* no special fasync */
  NULL,         /* no special check_media_change */
  NULL,         /* no special revalidate */
};
```

This is the `file_operations` structure being filled in to be passed back to the VFS.

```
#ifndef MODULE
long zero_init(long mem_start, long mem_end) {
  if (zero_major = register_chrdev(0, "zero", &zero_fops))
    printk("unable to get major for zero device\n");
  return mem_start;
}
```

If this device is being compiled directly into the kernel, this initialization function will need to be called from somwhere when the kernel is booting. Most character devices are initialized from `mem_init()` in mem.c. It is allowed to allocate a static block of memory for itself by keeping a pointer to `mem_start`, adding the amount of memory needed to `mem_start`, and returning a new pointer. It should not return a pointer greater than `mem_end`.

N.B.: Allocating memory this way is deprecated, since it makes writing a loadable device driver harder. This is vestigial functionality from when the Linux kernel malloc() was not able to allocate more than 4096 bytes at once.

```
#else
int init_module(void)
{
  if ((zero_major = register_chrdev(0, "zero", &zero_fops)) == -EBUSY) {
    printk("unable to get major for zero device\n");
    return -EIO;
  }
  return 0;
}
```

This is a function equivalent to zero_init() which is called when the module is loaded into a running system. Note that it takes zero arguments instead of two. This is because memory management setup has been completed and there is no nice large chunk of memory to grab from. This function should actually be very similar to zero_init(), since grabbing memory is deprecated. They can't be the same because they have to return different values. However, zero_init() could conceivably be written to call init_module() and init_module() could be included whether or not the driver was being compiled as a module. Choose what looks the cleanest for your own device.

Note that the register_chrdev() function is called with the first argument 0. The first argument can either be a requested major number, in which case the function returns failure (-EBUSY) if that major number is already allocated, or it can be 0, in which case the first available major number greater than 64 (see MAX_BLKDEV and MAX_CHRDEV in <linux/major.h>) is allocated and returned, or if all possible slots are taken up, -EBUSY is returned.

```
void cleanup_module(void)
{
  unregister_chrdev(zero_major, "zero");
}
#endif
```

This function is called when a request is made to remove the module from the kernel.

Character vs. Block Devices

The main difference between character-mode and block-mode devices in the Linux kernel is the way that requests to transfer data are made. As you can see in the device above, character devices have read and write functions that are called directly whenever I/O is required. Block devices generally don't have read or write functions. Instead, they implement a "strategy routine" or "request function" which is called not directly by system calls, but indirectly by the buffer cache.

If a program requests data from a file, the particular filesystem which holds that data determines what block the data is on and requests that block from the buffer cache. That block might be cached, in which case the request is satisfied by the buffer cache, or it might not, in which case the buffer cache creates a request for the device driver to fetch the block from disk and store the data in a buffer.

Of course, while finding that data block, the filesystem might have to read other blocks containing directory entries and inodes. When it needs to read those blocks, it requests them from the buffer cache in the same way it requests any other data block. The buffer cache does not need to know what the blocks are used for.

This indirect approach speeds up disk access considerably, and ends up simplifying some things, as crazy as that sounds. For one thing, the block device driver has very little interaction with user programs; calls to `ioctl()` are likely to be the most common direct interaction. This means that block device drivers don’t have to be as suspicious of their input as character device drivers. By the time a request for a data block has made it to the strategy routine, it has been pretty well checked to make sure it is valid. There is no question of writing to user-space memory and there is no chance that a buggy user-level program passed in bad arguments that need to be checked.

This is fortunate, because other facets of block device drivers are more complicated. There is more infrastructure to be set up than for a simple character device driver. Especially with interrupt-driven block devices, there are opportunities for race conditions that need to be watched out for.

Here is a simple example of what a non-interrupt-driven request function would look like.

```
static void do_foo_request(void) {

repeat:
  INIT_REQUEST;
  /* check to make sure that the request is for a valid physical device */
  if (!valid_foo_device(CURRENT->dev)) {
    end_request(0);
    goto repeat;
  }
  if (CURRENT->cmd == WRITE) {
    if (foo_write(CURRENT->sector, CURRENT->buffer, CURRENT->nr_sectors << 9)) {
      /* successful write */
      end_request(1);
      goto repeat;
    } else {
      end_request(0);
      goto repeat;
    }
  if (CURRENT->cmd == READ) {
    if (foo_read(CURRENT->sector, CURRENT->buffer, CURRENT->nr_sectors << 9)) {
      /* successful read */
      end_request(1);
      goto repeat;
    } else {
      end_request(0);
      goto repeat;
    }
  }
}
```

If this looks needlessly complex to you, realize that non-interrupt-driven device drivers do not take full advantage of the infrastructure. Interrupt-driven drivers, by contrast, only start things going, and then return without calling `end_request()` at all; the interrupt handler (or handlers) and timeout functions (if any) do that when a request has been satisfied or there has been an error. Here is an example of a vaguely-defined interrupt-driven device driver.

```
static int foo_busy; /* foo_init or init_module sets this to zero */

static void do_foo_request(void) {

  if (foo_busy)
    /* another request is being processed;
       this one will automatically follow */
    return;
  foo_busy = 1;
  foo_initialize_io();
}

static void foo_initialize_io(void) {

  if (CURRENT->cmd == READ) {
    SET_INTR(foo_read_intr);
  } else {
    SET_INTR(foo_write_intr);
  }
  /* send hardware command to start io
     based on request; just a request to
     read if read and preparing data for
     entire write; write takes more code */
}

static void foo_read_intr(void) {
  int error=0;

  CLEAR_INTR;
  /* read data from device and put in
     CURRENT->buffer; set error=1 if error
     This is actually most of the function... */
  /* successful if no error */
  end_request(error?0:1);
  if (!CURRENT)
    /* allow new requests to be processed */
    foo_busy = 0;
  /* INIT_REQUEST will return if no requests */
  INIT_REQUEST;
  /* Now prepare to do IO on next request */
  foo_initialize_io();
}

static void foo_write_intr(void) {
  int error=0;

  CLEAR_INTR;
  /* data has been written. error=1 if error */
  /* successful if no error */
  end_request(error?0:1);
  if (!CURRENT)
    /* allow new requests to be processed */
    foo_busy = 0;
  /* INIT_REQUEST will return if no requests */
  INIT_REQUEST;
  /* Now prepare to do IO on next request */
  foo_initialize_io();
}
```

I cannot fully cover block device drivers within a 50-minute talk, but this should give you an impression of what Linux block device drivers are like, and demonstrate that the request function interface is a flexible, working abstraction. The KHG contains more information, if you wish to explore further. You could even (gasp) read the source code to a few simple working drivers, such as the ramdisk driver.

The Hardware Interface

Linux provides convenience routines for accessing I/O ports, hardware interrupts, and DMA channels. The KHG covers most of the functions mentioned here in more detail.

I/O access

The Linux kernel provides a service to register I/O port usage. Before probing for a device in an initialization function, the driver should call `check_region()`. It takes two arguments; the beginning and length of the range of I/O ports you wish to access. It will return `-EBUSY` if any of the ports are in use, and 0 otherwise. This will keep you from confusing other devices by reading from or writing to their I/O ports in your attempt to probe for your device. Then your driver should register the ports it wishes to use with the `request_region()` function, which takes three arguments: the beginning of the region, the length of the region, and the name of the driver. When your driver is unloaded, it should call `release_region()` with two arguments, the beginning and length of the region, to release the region.

To access I/O ports, 12 inline functions are available by including <asm/io.h>. Six functions read data from ports, and each takes one argument: the name of the port. The other six write data to ports, and they take two arguments, the first being the value to write, and the second being the port to write it to. Each function has a size designation: `b` stands for byte, `w` for word (16 bits), and `l` for long (32 bits). Half of the functions are "pausing" functions that pause briefly when writing; a lot of hardware is a little slow on the uptake when it is being read or written, and is unable to keep up with the CPU. These functions have `_p` appended to their names. Here is the list: `inb()`, `inb_p()`, `outb()`, `outb_p()`, `inw()`, `inw_p()`, `outw()`, `outw_p()`, `inl()`, `inl_p()`, `outl()`, `outl_p()`, and `inb()`.

Hardware interrupts

`request_irq()` requests an IRQ from the kernel, and installs an interrupt handler on that IRQ if successful.
Takes four arguments:
`unsigned int irq` is the number of the IRQ being requested.
`void (*handler)(int, struct pt_regs *)` is a pointer to the interrupt handling function.
`unsigned long flags` set to `SA_INTERRUPT` to request a "fast" interrupt, or 0 to request a normal "slow" interrupt. The process scheduler is not run when returning from a fast interrupt, but it may be run when returning from a slow interrupt.
`const char *device` a string containing the name of the device. It is used to give the name of the driver in the /proc/interrupts listing.
Your handler will then be called whenever that particular interrupt occurs. The handler does not need to be re-entrant.

`free_irq()` frees up an IRQ. It takes one argument; the interrupt number to free.

`cli()`, which stands for CLear Interrupt enable, disables interrupts temporarily. `sti()`, SeT Interrupt enable, re-enables them. These are used to prevent race conditions where an interrupt-driven function and a system call (or function called from a system call) access the same data structures.

DMA channels

DMA channels in the PC are peculiar devices. `<asm/dma.h>` has declarations of functions that can be used to manage DMA operations, as well as some documentation. While initilizing the driver, call `request_dma(channel, "name")`, where `channel` is the DMA channel that you wish to allocate, and `"name"` is the name of the driver. The name will be used to show the owner of the channel in /proc/dma. When setting up DMA, use a sequence somewhat like the following.

```
cli();
diable_dma(channel);     /* Turn it off */
clear_dma_ff(channel);   /* Clear pointer flip/flop */
/* Set DMA mode. Some of these are defined in
 * dma.h. Others (such as auto-initialize mode)
 * aren't there but you can either (a) find them
 * in other drivers (the znet Ethernet card driver
 * has a few) or (b) figure out the hex value to
 * plug into the 8237's registers. Get the specs
 * on the 8237 DMA controller chip if you don't
 * have them already.
 */
set_dma_mode(channel, DMA_MODE_READ);
/* Set transfer address and page bits for your channel */
set_dma_addr(channel, buffer);
/* Set tranfer size */
set_dma_count(channel, count);
enable_dma(channel);
sti();
```

You will still have to make the device do the DMA, as well. Other functions are available for managing DMA depending on what you need to do; all of these functions except for `disable_dma()`, `enable_dma()`, `request_dma()`, and `free_dma()` should be called with interrupts **disabled**.

Make sure that you read all the comments in dma.h, as they will help you avoid many possible mistakes in programming DMA. It is probably also worth reading the source code for other drivers that use DMA. Also, read the actual `dma_*()` function source code which is in `<asm/dma.h>` and compare it to the documentation for the device for which you are writing a driver to make sure that you understand what you are doing; DMA is probably the easiest hardware programming interface to use incorrectly.

Kernel convenience routines

Linux provides many routines that device drivers commonly use. The KHG covers most of the functions mentioned here in more detail.

Access to user memory

Linux has 8 functions for moving data between user space and kernel space. Their names are mostly self-explanitory, and they are all declared in `<asm/segment.h>`. The `get_user_*()` functions, `get_user_byte()`, `get_user_word()`, and `get_user_long()`, each take one argument, the address from which to fetch data. The `put_user_*()` functions, `put_user_byte()`, `put_user_word()`, and `put_user_long()`, each take two arguments, the first being the value to put, and the second being the user-space address at which to put it. Two `memcpy()`-like functions are also available; `memcpy_fromfs(to, from, n)` copies `n` bytes to kernel address `to` from user address `from`, and `memcpy_tofs(to, from, n)` copies `n` bytes to user address `to` from kernel address `from`. The reason the name uses "fs" instead of "user" is that on Linux/i386, the "fs" register is used to point to user space while in kernel mode.

Before accessing memory, use `verify_area()` to avoid kernel-space segmentation faults in case of error. `verify_area()` takes three arguments: the first is the type (`VERIFY_WRITE` or `VERIFY_READ`), the second is the address at which to start validating, and the third is the number of bytes to validate.

Memory allocation

Several sets of functions are available for memory allocation. The first is `kmalloc()`/`kfree()`/`kfree_s()`. These works much like the `malloc()`/`free()` available in the C library, except that the limit on the request size is smaller. Also, kmalloc takes two arguments instead of one; the first argument is the usual size of the region to allocate, and the second is the "priority". This is one of the `GFP_*` defines in the file `<linux/mm.h>`. If the driver can be safely pre-empted, then `GFP_KERNEL` should be used. If not, or from within an interrupt handler, `GFP_ATOMIC` should be used. Only use `GFP_ATOMIC` if absolutely necessary, because it places a larger strain on the memory management system. In order to allocate DMA-able memory, `GFP_DMA` should be used. This may allow pre-emption to take place, so be careful where you use it.

Do be careful to free everything when you are done using it, because kernel memory is non-swapable which makes memory leaks more serious than in user-space programs. Also, be careful not to free memory before you are finished using it, because freeing memory and the continuing to use it will usually cause a kernel fault--and that’s if you are lucky. If you are unlucky, it will silently corrupt memory.

Sleeping

The first rule about sleeping is that only kernel code that is called from user code can sleep. Kernel code called from an interrupt handler cannot sleep.

If a device driver needs to sleep on an event, it can call one of several functions that are available for doing so, which work for most instances. However, some drivers need to sleep on multiple events, or do something else to avoid race conditions. In Linux, a task in kernel mode can set its state hint to a sleeping mode and keep executing for a while before calling the scheduler, which schedules another task to run. This is extremely flexible, and is partially covered in the KHG. Several devices use this to good effect, including simple devices like the lp parallel port driver and complex ones like the serial driver.

There are two functions for simple sleeping on an event: `sleep_on()` and `interruptible_sleep_on()`. There are two corresponding functions for waking up all processes sleeping on an event: `wake_up()` and `wake_up_interruptible()`.

Timers

To simply go to sleep for a short time, measured in "jiffies" (hundredths of a second), the following code can be used. *jiffies_to_wait* determines a *minimum* length of time to wait. Using this code requires you to include <linux/sched.h>.

```
current->state = TASK_INTERRUPTIBLE;
current->timeout = jiffies + jiffies_to_wait;
schedule();
```

While the process is paused, other processes will run, and may well run in kernel space, so do not depend on the state of static data structures remaining the same after the call to schedule.

Timers that act like hardware interrupts are also available. Include <linux/timer.h> and allocate a struct timer_list. First pass a pointer to your structure to init_timer(), then fill in the expires, data, and function members, then call add_timer() with a pointer to your structure as the argument. expires gives the number of jiffies after which to time out, data gives the argument to pass to the timer handler, and function is a pointer to the timer handler function. When the function is called, it will not be executed in the context of a running process, so it will not be able to access any user-space data. Just like with a hardware interrupt handler, only kernel-space data structures will be available.

It is possible to request multiple timers at once by making a list of these timer structures; read <linux/timer.h> for details. Most of the time, this will not be necessary.

Reporting information

There are several ways to report information. printk() is a kernel version of the libc printf() function which does not handle floating point numbers. It prints to the screen unless a kernel logging daemon such as klogd is running, in which case it is logged to system log files. printk() enables interrupts, and is not safe to call from withing cli()/sti()-protected code. Even if it did not explicitly enable interrupts, it causes implicit I/O and might cause pre-emption to occur. If you need to debug interrupt-disabled code by printing to the screen, use sprintf() to fill a string and use console_print() to display it on the screen. console_print() is not declared in any header files, you will need to declare it with

```
extern void console_print(const char *);
```

before using it. It is defined in drivers/char/console.c.

It is also possible to use gdb to read /proc/kcore to do inspection-only debugging of the kernel. This currently does not work with loadable modules, but a kernel patch is available to allow inspection of loaded modules as well.

Block requests

Block devices do I/O by iterating over a sorted list of requests for I/O. When a request has been fulfilled, an inline function called end_request() is called, which appropriately handles the request, including waking up any processes sleeping on the request and cleaning up the request list. The INIT_REQUEST; macro is then called; if there are no requests left, it causes the function to exit. Otherwise, it sets up the next request.

Trademarks

Acknowledgements

Thanks to Matt Welsh for his help understanding DMA under Linux.