INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE: CENTRATOM TRIESTE

The United Nations University

SMR/774 - 11

THIRD COLLEGE ON MICROPROCESSOR-BASED REAL-TIME CONTROL - PRINCIPLES AND APPLICATIONS IN PHYSICS
26 September - 21 October 1994

LINUX-DEVICE DRIVERS

Ulrich RAICH
C.E.R.N.-European Organization for Nuclear Research
E.P. Division
CH-1211 Geneva
SWITZERLAND

```
************************************************************
*                                                          *
*    Guide To Linux Driver Writing -- Character Devices    *
*                                                          *
*                              or,                         *
*                                                          *
*         The Wacky World of Driver Development (I)        *
*                                                          *
*                  Last Revision: Apr 11, 1993             *
*                                                          *
************************************************************
```

 Suggestions, criticisms, and comments to baruch@nynexst.com.

Right, now that that's over with, let's get into the fun stuff!

Introduction

There is a companion guide to this Guide, the Linux Character Device Tutorial. This tutorial contains working examples of driver code. It introduces the reader gently into each aspect of character device driver writing through experiments which are carried out by the programmer.

This Guide should serve as a reference to both beginning and advanced driver writers.

-=-=-=-=-=-=-

Some words of thanks:

Many thanks to:

Donald J. Becker (becker@metropolis.super.org)
Don Holzworth (donh@gcx1.ssd.csd.harris.com)
Michael Johnson (johnsonm@stolaf.edu)
Karl Heinz Kremer (khk@raster.kodak.com)
All the driver writers!

...and of course, Linus "Linux" Torvalds and all the guys who helped develop Linux into a BLOODY KICKIN' O/S!

-=-=-=-=-=-=-

...and now a word of warning:

Messing about with drivers is messing with the kernel. Drivers are run at the kernel level, and as such are not subject to scheduling. Further, drivers have access to various kernel structures. Before you actually write a driver, be *damned* sure of what you are doing, lest you end up having to re-format your harddrive and re-install Linux!

The information in this Guide is as up-to-date as I could make it. It also has no stamp of approval whatsoever by any of the designers of the kernel. I am not responsible for damage caused to anything as a result of using this Guide.

End of Introduction

Kernal-callable functions:

Note: There is no close for a character device. There is only release. See the file data structure below to find out how to determine the number of processes which have the device open.

-=-=-=-=-=-=-

init : Initializes the driver on bootup.

```
  unsigned long driver_init(unsigned long kmem_start, unsigned long kmem_end)

Arguments: kmem_start -- the start of kernel memory
           kmem_end   -- the end of kernel memory

Returns: The new start of kernel memory.  This will be different from the
  kmem_start argument if you want to allocate memory for the driver.

The arguments you use depends on what you want to do.  Remember that since
you are going to add your init function to kernel/chr_dev/mem.c, you can
make your call anything you like, but you have access to the kernel memory
start and end.

Generally, the init function initializes the driver and hardware, and
displays some message telling of the availability of the driver and
hardware.  In addition, the register_chrdev function is usually called here.


**************
open : Open a device

  static int driver_open(struct inode * inode, struct file * file)

Arguments: inode    -- pointer to the inode structure for this device
           file     -- pointer to the file structure for this device

Returns: 0 on success,
         -errno on error.

This function is called whenever a process performs open (or fopen) on
the device special file.  If there is no open function for the driver,
nothing spectacular happens.  As long as the /dev file exists, the
open will succeed.


**************
read : Read from a device

  static int driver_read(struct inode * inode, struct file * file,
                         char * buffer, int count)

Arguments: inode    -- pointer to the inode structure for this device
           file     -- pointer to the file structure for this device
           buffer   -- pointer to the buffer in user space to read into
           count    -- the number of bytes to read

Returns: -errno on error
         >=0 : the number of bytes actually read

If there is no read function for the driver, read calls will return EINVAL.


**************
write : Write to a device

  static int driver_write(struct inode * inode, struct file * file,
                          char * buffer, int count)

Arguments: inode    -- pointer to the inode structure for this device
           file     -- pointer to the file structure for this device
           buffer   -- pointer to the buffer in user space to write from
           count    -- the number of bytes to write

Returns: -errno on error
         >=0 : the number of bytes actually written

If there is no write function for the driver, write calls will return
EINVAL.


**************
lseek : Change the position offset of the device

  static int driver_lseek(struct inode * inode, struct file * file,
                          off_t offset, int origin)

Arguments: inode    -- pointer to the inode structure for this device
           file     -- pointer to the file structure for this device
           offset   -- offset from origin to move to (bytes)
```

```
          origin    -- origin to move from :
                       0 = from origin 0 (beginning)
                       1 = from current position
                       2 = from end

Returns: -errno on error
         >=0 : the position after the move

See Also: Data Structure 'file'

If there is no lseek function for the driver, the kernel will take the default
seek action, which is to alter the file->f_pos element.  For origins of 2,
the default action results in -EINVAL if file->f_inode is NULL, or it
sets file->f_pos to file->f_inode->i_size + offset otherwise.


***************
ioctl : Various device-dependent services

  static int driver_ioctl(struct inode *inode, struct file *file,
                          unsigned int cmd, unsigned long arg)

Arguments: inode     -- pointer to the inode structure for this device
           file      -- pointer to the file structure for this device
           cmd       -- the user-defined command to perform
           arg       -- the user-defined argument.  You may use this
                        as a pointer to user space, since sizeof(long)==
                        sizeof(void *).

Returns: -errno on error
         >=0 : whatever you like! (user-defined)

For cmd, FIOCLEX, FIONCLEX, FIONBIO, and FIOASYNC are already defined.
See the file linux/fs/ioctl.c, sys_ioctl to find out what they do.
If there is no ioctl call for the driver, and the ioctl command performed
is not one of the four types listed here, ioctl will return -EINVAL.


***************
select : Performs the select call on the device:

  static int driver_select(struct inode *inode, struct file *file,
                           int sel_type, select_table * wait)

Arguments: inode     -- pointer to the inode structure for this device
           file      -- pointer to the file structure for this device
           sel_type  -- the select type to perform :
                        SEL_IN (read)
                        SEL_OUT (write)
                        SEL_EX (exception)
           wait      -- see the section "Some Notes" for select.

Returns: 0 if the device is not ready to perform the sel_type operation
         !=0 if it is.

See the "Some Notes" section 'way below on information on how to use
the select call in drivers.  If there is no select call for the driver,
select will act as if the driver is ready for the operation.


***************
release : Release device (no process holds it open)

  static void driver_release(struct inode * inode, struct file * file)

Arguments: inode     -- pointer to the inode structure for this device
           file      -- pointer to the file structure for this device

The release call is activated only when the process closes the device as
many times as it has opened it.  That is, if the process has opened the
device five times, then only when close is called for the fifth time
will release be called (that is, provided there were no more calls to open!).
If there is no release call for the driver, nothing spectacular happens.


***************
readdir : Get the next directory entry

  static int driver_readdir(struct inode *inode, struct file *file,
                            struct dirent *dirent, int count)
```

```
Arguments: inode     -- pointer to the inode structure for this device
           file      -- pointer to the file structure for this device
           dirent    -- pointer to a dirent ("directory entry") structure
           count     -- number of entries to read (currently always 1)

Returns: 0 on success
         -errno on failure.

If there is no readdir function for the driver, readdir will return
-ENOTDIR.  This is really for file systems, but you can probably use
it for whatever you like in a non-fs device, as long as you return
a dirent structure.

See Also: dirent (data structure)


**************
mmap : Forget this.  According to the source (src/linux/mm/mmap.c),
       for character devices only /dev/[k]mem may be mapped.
       Besides, I'm not too clear on what it will do.


------------------------------------------------------------------------

Data structures:
----------------

dirent : Information about files in a directory.

#include <linux/dirent.h>

struct dirent {
        long            d_ino;                  /* Inode of file */
        off_t           d_off;
        unsigned short  d_reclen;
        char            d_name[NAME_MAX+1];     /* Name of file */
};


**************
file : Information about open files

According to the Hacker's Guide to Linux, this structure is mainly used
for writing filesystems, not drivers.  However, there is no reason it
cannot be used by drivers.

#include <linux/fs.h>

struct file {
        mode_t f_mode;
        dev_t f_rdev;                   /* needed for /dev/tty */
        off_t f_pos;                    /* Curr. posn in file */
        unsigned short f_flags;         /* The flags arg passed to open */
        unsigned short f_count;         /* Number of opens on this file */
        unsigned short f_reada;
        struct inode * f_inode;         /* pointer to the inode struct */
        struct file_operations * f_op;  /* pointer to the fops struct */
};


**************
file_operations : Tells the kernel which function to call for
                  which kernel function.

#include <linux/fs.h>

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 int);
        int (*mmap) (void);
        int (*open) (struct inode *, struct file *);
        void (*release) (struct inode *, struct file *);
        int (*fsync) (struct inode *, struct file *);
};
```

```
**************
inode : Information about the /dev/xxx file (or inode)

#include <linux/fs.h>

struct inode {
        dev_t           i_dev;
        unsigned long   i_ino;  /* Inode number */
        umode_t         i_mode; /* Mode of the file */
        nlink_t         i_nlink;
        uid_t           i_uid;
        gid_t           i_gid;
        dev_t           i_rdev;  /* Device major and minor numbers */
        off_t           i_size;
        time_t          i_atime;
        time_t          i_mtime;
        time_t          i_ctime;
        unsigned long   i_blksize;
        unsigned long   i_blocks;
        struct inode_operations * i_op;
        struct super_block * i_sb;
        struct wait_queue * i_wait;
        struct file_lock * i_flock;
        struct vm_area_struct * i_mmap;
        struct inode * i_next, * i_prev;
        struct inode * i_hash_next, * i_hash_prev;
        struct inode * i_bound_to, * i_bound_by;
        unsigned short i_count;
        unsigned short i_flags;  /* Mount flags (see fs.h) */
        unsigned char i_lock;
        unsigned char i_dirt;
        unsigned char i_pipe;
        unsigned char i_mount;
        unsigned char i_seek;
        unsigned char i_update;
        union {
                struct pipe_inode_info pipe_i;
                struct minix_inode_info minix_i;
                struct ext_inode_info ext_i;
                struct msdos_inode_info msdos_i;
                struct iso_inode_info isofs_i;
                struct nfs_inode_info nfs_i;
        } u;
};

See Also: Driver Calls: MAJOR, MINOR, IS_RDONLY, IS_NOSUID, IS_NODEV,
                        IS_NOEXEC, IS_SYNC

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

Driver calls:


**************
add_timer : Cause a function to be executed when a given amount of time
            has passed.

#include <linux/sched.h>

void add_timer(long jiffies, void (*fn)(void))

Arguments: jiffies -- The number of jiffies to time out after.
           fn      -- The function in kernel space to run after timeout.

Note!  This is NOT process-specific!  If you are looking for a way
to have a process go to sleep and timeout, look for ?
Excessive use of this function will cause the kernel to panic if there are
too many timeouts active at once.


**************
cli : Macro, Prevent interrupts from occuring

#include <asm/system.h>

#define cli() __asm__ __volatile__ ("cli"::)

See Also: sti
```

```
**************
free_irq : Free a registered interrupt

#include <linux/sched.h>

void free_irq(unsigned int irq)

Arguments: irq -- the interrupt level to free up

See Also: request_irq


**************
get_fs_byte, get_fs_word, get_fs_long : Get data from user space

Purpose: Allows a driver to access data in user space (which is in
         a different segment than the kernel!)

#include <asm/segment.h>

inline unsigned char get_fs_byte(const char * addr)
inline unsigned short get_fs_word(const unsigned short *addr)
inline unsigned long get_fs_long(const unsigned long *addr)

Arguments: addr -- the address in user space to get data from

Returns: the value in user space.

See Also: memcpy_fromfs, memcpy_tofs, put_fs_byte, put_fs_word, put_fs_long


**************
inb, inb_p : Inputs a byte from a port

#include <asm/io.h>

inline unsigned int inb(unsigned short port)
inline unsigned int inb_p(unsigned short port)

Arguments: port -- the port to input a byte from

Returns:  Byte received in the low byte.  High byte unused.

See Also: outb, outb_p


**************
IS_RDONLY, IS_NOSUID, IS_NODEV, IS_NOEXEC, IS_SYNC: Macros, check the status
    of the device on the filesystem

#include <linux/fs.h>

#define IS_RDONLY(inode) (((inode)->i_sb) && ((inode)->i_sb->s_flags &
                          MS_RDONLY))
#define IS_NOSUID(inode) ((inode)->i_flags & MS_NOSUID)
#define IS_NODEV(inode) ((inode)->i_flags & MS_NODEV)
#define IS_NOEXEC(inode) ((inode)->i_flags & MS_NOEXEC)
#define IS_SYNC(inode) ((inode)->i_flags & MS_SYNC)


**************
kfree, kfree_s : Free memory which has been kmalloced.

#include <linux/kernel.h>

#define kfree(x) kfree_s((x), 0)
void kfree_s(void * obj, int size)

Arguments : obj  -- pointer to kernel memory you want to free
            size -- size of block you want to free (0 if you don't know
                    or are lazy -- slows things down)


**************
kmalloc : Allocate memory in kernel space

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

```
void * kmalloc(unsigned int len, int priority)

Arguments: len      -- the length of the memory to allocate.  Must not be bigger
                       than 4096.
           priority -- GFP_KERNEL or GFP_ATOMIC.  GFP_ATOMIC causes kmalloc
                       to return NULL if the memory could not be found
                       immediately.  GFP_KERNEL is the usual priority.

Returns: NULL on failure, a pointer to kernel space on success.


**************
memcpy_fromfs, memcpy_tofs : Copies memory from user(fromfs)/kernel(tofs)
                             space to kernel/user space

#include <asm/segment.h>

inline void memcpy_fromfs(void * to, const void * from, unsigned long n)
inline void memcpy_tofs(void * to, const void * from, unsigned long n)

Arguments: to   -- Address to copy data to
           from -- Address to copy data from
           n    -- number of bytes to copy

See Also: get_fs_byte, get_fs_word, get_fs_long,
          put_fs_byte, put_fs_word, put_fs_long

Warning!  Get the order of arguments right!


**************
MAJOR, MINOR : Macros, get major/minor device number from inode i_dev entry.

#include <linux/fs.h>

#define MAJOR(a) (((unsigned)(a))>>8)
#define MINOR(a) ((a)&0xff)


**************
outb, outb_p : Outputs a byte to a port

#include <asm/io.h>

inline void outb(char value, unsigned short port)
inline void outb_p(char value, unsigned short port)

Arguments: value -- the byte to write out
           port  -- the port to write it out on

See Also: inb, inb_p


**************
printk : Kernel printf

#include <linux/kernel.h>

int printk(const char *fmt, ...)

Arguments: fmt -- printf-style format
           ... -- var-arguments, printf-style

Returns: Number of characters printed.


**************
put_fs_byte, put_fs_word, put_fs_long : Put data into user space

Purpose: Allows a driver to put a byte, word, or long into user space,
         which is at a different segment than the kernel.

#include <asm/segment.h>

inline void put_fs_byte(char val,char *addr)
inline void put_fs_word(short val,short * addr)
inline void put_fs_long(unsigned long val,unsigned long * addr)

Arguments: addr -- the address in user space to get data from
```

```
Returns: the value in user space.

See Also: memcpy_fromfs, memcpy_tofs, get_fs_byte, get_fs_word, get_fs_long

Warning!  Get the order of arguments right!


***************
register_chrdev : Register a character device with the kernel

#include <linux/fs.h>
#include <linux/errno.h>

int register_chrdev(unsigned int major, const char *name,
                            struct file_operations *fops)

Arguments: major -- the major device number to register as
           name  -- the name of the device (currently unused)
           fops  -- a file_operations structure for the device.

Returns: -EINVAL if major is >= MAX_CHRDEV (defined in fs.h as 32)
         -EBUSY if major device has already been allocated
         0 on success.


***************
request_irq : Request to perform a function on an interrupt

#include <linux/sched.h>
#include <linux/errno.h>

int request_irq(unsigned int irq, void (*handler)(int))

Arguments: irq     -- the interrupt to request.
           handler -- the function to handle the interrupt.  The interrupt
                      handler should be of the form void handler(int).
                      Unless you really know what you are doing, don't
                      use the int argument.

Returns: -EINVAL if irq>15 or handler==NULL
         -EBUSY if irq is already allocated.
         0 on success.

See Also: free_irq


***************
select_wait : Add a process to the select-wait queue

#include <linux/sched.h>

inline void select_wait(struct wait_queue ** wait_address, select_table * p)

Arguments: wait_address -- Address of a wait_queue pointer
           p            -- Address of a select_table

Devices which use select should define a struct wait_queue pointer and
initialize it to NULL.  select_wait adds the current process to a circular
list of waits.  The pointer to the circular list is wait_address.  If
p is NULL, select_wait does nothing, otherwise the current process is
put to sleep.

See Also: sleep_on, interruptible_sleep_on, wake_up, wake_up_interruptible


***************
sleep_on, interruptible_sleep_on : Put the current process to sleep.

#include <linux/sched.h>

void sleep_on(struct wait_queue ** p)
void interruptible_sleep_on(struct wait_queue ** p)

Arguments: q -- Pointer to the driver's wait_queue (see select_wait)

sleep_on puts the current process to sleep in an uninterruptible state.
That is, signals will not wake the process up.  The only thing which
will wake a process up in this state is a hardware interrupt (which
would call the interrupt handler of the driver) -- and even then the
interrupt routine needs to call wake_up to put the process in a running
```

state.

interruptible_sleep_on puts the current process to sleep in an interruptible state, which means that not only will hardware interrupts get through, but also signals and process timeouts ("alarms") will cause the process to wake up (and execute interrupt or signal handlers). A call to wake_up_interruptible is necessary to wake up the process and allow it to continue running where it left off.

See Also: select_wait, wake_up, wake_up_interruptible

****
sti : Macro, Allow interrupts to occur

```
#include <asm/system.h>

#define sti() __asm__ __volatile__ ("sti"::)
```

See Also: cli

****
sys_getegid, sys_getgid, sys_getpid, sys_getppid, sys_getuid, sys_geteuid :
  Funky functions which get various information about the current process,

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

int sys_getegid(void)
int sys_getgid(void)
int sys_getpid(void)
int sys_getppid(void)
int sys_getuid(void)
int sys_geteuid(void)
```

sys_getegid gets the effective gid of the process.
sys_getgid gets the group ID of the process.
sys_getpid gets the process ID of the process.
sys_getppid gets the process ID of the process' parent.
sys_geteuid gets the effective uid of the process.
sys_getuid gets the user ID of the process.

****
wake_up, wake_up_interruptible : Wake up _all_ processes waiting on the wait queue.

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

void wake_up(struct wait_queue **q)
void wake_up_interruptible(struct wait_queue **q)
```

Arguments: q -- Pointer to the driver's wait_queue (see select_wait)

See Also: select_wait, sleep_on, interruptible_sleep_on

Some notes

Interrupts, Drivers, and You!

First, a brief exposition on the Meaning of Interrupts. There are three ways by which a program running in the CPU may be interrupted. The first is the external interrupt. This is caused by an external device (that is, external to the CPU) signalling for attention. These are referred to as "interrupt requests" or "IRQs".

The second method is the exception, which is caused by something internal to the CPU, usually in response to a condition generated by execution of an instruction.

The third method is the software interrupt, which is a deliberately executed interrupt -- the INT instruction in assembly. System calls are implemented using software interrupts; when a system call is desired, Linux places the system call number in EAX, and performs an INT 0x80 instruction.

Since drivers usually deal with hardware devices, it is logical that driver interrupts should refer to external interrupts. There are 16 available IRQs -- IRQ0 through IRQ15. The following table lists the official uses of the various IRQs:

```
IRQ0    -- timer 0
IRQ1    -- keyboard
IRQ2    -- AT slave 8259 ("cascade")
IRQ3    -- COM2
IRQ4    -- COM1
IRQ5    -- LPT2
IRQ6    -- floppy
IRQ7    -- LPT1
IRQ8-12   ??????
IRQ13   -- coprocessor error
IRQ14,15  ??????
```

Writing drivers which can be interrupted requires care. Be aware that every line you write can be interrupted, and thus cause variable changes to occur. If you really want to protect critical sections from being interrupted, use the cli() and sti() driver calls.

Suppose you wanted to test some kind of funky condition, where success of the condition leads to going to sleep, and being woken up by an interrupt. Consider this code:

```
void driver_interrupt(int unused)
{
  if (!driver_stuff.int_flag) return;  /* Spurious interrupts
                                          are not unheard of */
  driver_stuff.int_flag=0;
  weird_wacky(); /* Do some weird and wacky stuff
                    here to handle the interrupt */
  disable_ints(); /* Disable the device from issuing interrupts */
  wake_up(&driver_stuff.wait_queue); /* Sets process to TASK_RUNNING */
}

if (conditions_are_ripe())
{
  driver_stuff.int_flag = 1;
  enable_ints(); /* Enable device to interrupt us */
  sleep_on(&driver_stuff.wait_queue); /* Sets process to TASK_UNINTERRUPTIBLE */
}
```

Assume we just leave the conditions_are_ripe code, determining that the conditions are ripe! We have just enabled the device to interrupt the machine. So we are now about to enter the sleep_on code, and what should happen but the pesky device issues an interrupt. Ka-chunk! and we enter the driver_interrupt routine, which does some weird and wacky stuff to handle the interrupt, and then we disable the device's interrupts. Ka-ching! we enter the wake_up function which sets the process up to run again. Boink! we exit the interrupt handler and commence where we left off (just about to enter the sleep_on code). Vooosh! we're now sleeping the process, awaiting an interrupt which will never occur, since the interrupt handler disabled the device from interrupts! What to do?

Use cli() and sti() to protect the critical sections of code:

```
cli();
if (conditions_are_ripe())
{
  driver_stuff.int_flag = 1;
  enable_ints(); /* Enable device to interrupt us */
  sleep_on(&driver_stuff.wait_queue); /* Sets process to TASK_UNINTERRUPTIBLE */
}
else sti();
```

First we clear interrupts. This is not the same as disabling device interrupts! This actually prevents a hardware interrupt from causing the CPU to execute interrupt code. In effect, the interrupt is deferred.

Now we can do our check and perform sleep_on, secure in the knowledge that the interrupt handler cannot be called. The sleep_on (and interruptible_sleep_on) call has a sti() in it in the right place, so you don't have to worry about calling sti() before sleep_on, and running into a race condition again.

Of course, with any interruptible device driver, you must be careful never to spend too much time in the interrupt routine if you are expecting more

than one interrupt, because you may miss your second interrupt.

-=-=-=-=-=-=-

Drivers and signals:

When a process is sleeping in an interruptible state, any signal can wake it up. This is the sequence of events which occurs when a sleeping process receives a signal:

(1) Set current->signal.
(2) Set the process to a runnable state.
(3) Execute the rest of the driver call.
(4) Run the signal handler.
(5) If the driver call in step 3 returned -ERESTARTNOHAND or -ERESTARTNOINTR, then return from the driver call with EINTR. If the driver call in step 3 returned -ERESTARTSYS, then restart the driver call. Otherwise, just return with whatever was returned from the driver call.

In the driver, you can tell if a sleep has been interrupted by a signal with the following code:

```
if (current->signal & ~current->blocked)
{
  /* Do things based on sleep interrupted by signal */
}
```

-=-=-=-=-=-=-

Drivers and timeouts:

Suppose you wanted to sleep on an interrupt, but also time out after a period of time. You could always use the add_timer, but that's frowned upon because there are only a limited number of timers available -- currently there are 64.

The usual solution is to manually alter the current process's timeout:

```
current->timeout = jiffies + X;
interruptible_sleep_on(&driver_stuff.wait_queue);
```

(Interruptible sleep_on must be used here to allow a timeout to interrupt the sleep). This will cause the scheduler to set the task running again when X jiffies has gone by. Even if the timeout goes off and the process is allowed to continue running, it is probably a good idea to call wake_up_interruptible in case the process needs to be rescheduled.

To find out if it was a timeout which caused the process to wake up, check current->timeout. If it is 0, a timeout occurred. Otherwise it should remain what you set it at. If a timeout did not occur, and something else woke the process up, you should set current->timeout to 0 to prevent the timeout from continuing.

The disadvantage of this method is that the process can only have one timeout at a time. Over *all* drivers.

-=-=-=-=-=-=-

The driver_select call:

When a process issues a select call, it is checking to see if the given devices are ready to perform the given operations. For example, suppose you want a driver to have a command written to it, and to disallow further commands until the current command is complete. Well, in the write call you would block commands if there is already a command operating (for example, waiting for a board to do something). But that would require the process to write over and over again until it succeeds. That just burns cycles.

The select call allows a process to determine the availability of read and write. In the above example, one merely has to select for write on that device's file descriptor (as returned by open), and the process would be put to sleep until the device is ready to be written to.

The kernel will call the driver's driver_select call when the process issues a select call. The arguments to the driver_select call are detailed above. If the wait argument is non-NULL, and there is no error condition caused

by the select, driver_select should put the process to sleep, and arrange to be woken up when the device becomes ready (usually through an interrupt).

If, however, the wait argument is NULL, then the driver should quickly see if the device is ready, and return even if it is not. The select_wait function does this already for you (see further).

Putting the process to sleep does not require calling a sleep_on function. It is the select_wait function which is called, with the p argument being equal to the wait argument passed to driver_select.

select_wait is pretty much equivalent to interruptible_sleep_on in that it adds the current process to the wait queue and sleeps the process in an interruptible state. The internals of the differences between select_wait and interruptible_sleep_on are relatively irrelevant here. Suffice it to say that to wake the process up from the select, one needs to perform the wake_up_interruptible call. When that happens, the process is free to run.

However, in the case of interruptible_sleep_on, the process will continue running after the call to interruptible_sleep_on. In the case of select_wait, the process does not. driver_select is called as a "side effect" of the select call, and so completes even when it calls select_wait. It is not select_wait which sleeps the process, but the select call. Nevertheless, it is required to call select_wait to add the process to the wait-queue, since select will not do that.

All one needs to remember for driver_select is:

(1) Call select_wait if the device is not ready, and return 0.
(2) Return 1 if the device is ready.

Calling select with a timeout is really no different to the driver than calling it without select. But there is one crucial difference. Remember timing out on interrupts above? Well, interrupt timeouts and select timeouts cannot co-exist. They both use current->timeout to wake the process up after a period of time. Remember that!

Installation notes:

Before you sit down and write your first driver, first make sure you understand how to recompile the kernel. Then go ahead and recompile it! Recompilation of the kernel is described in the FAQ. If you can't recompile the kernel, you can't install your driver into the kernel. [Although I hear tell of a package on sunsite which can load and unload drivers while the kernel is running. Until I test out this package, I won't include instructions for it here.]

For character devices, you need to go into the mem.c file in the (source)/linux/kernel/chr_dev directory, to the chr_dev_init function, and add your init function to it. Recompile the kernel, and away you go!

(BTW, would you manually have to do a mknod to make the /dev/xxx entry for your driver? Can you do it in the init function?)

In general, one installs a device special file in /dev manually, by using mknod:

```
mknod /dev/xxx c major minor
```

If you registered your character driver as major device X, then all accesses to /dev/xxx where major==X will call your driver functions.

```
************************************************************
*                                                          *
*  Tutorial To Linux Driver Writing -- Character Devices   *
*                                                          *
*                              or,                         *
*                                                          *
*       Now That I'm Wacky, Let Me Do Something (I)        *
*                                                          *
*                  Last Revision: Apr 11, 1993             *
*                                                          *
************************************************************
```

 Suggestions, criticisms, and comments to baruch@nynexst.com. This document, nor the work performed by Robert Baruch using Linux, nor the results of said work are connected in any way to any of the Nynex companies. This product 0% organic as defined by California Statute 4Z//7&A. No artificial coloring or flavoring.

Introduction

There is a companion guide to this Tutorial, the Guide to Linux Driver Writing -- Character Devices This Guide should serve as a reference to both beginning and advanced driver writers, and should be used in conjunction with this Tutorial.

Some words of thanks:

Many thanks to:

Donald J. Becker (becker@metropolis.super.org)
Don "May the Source be With You!" Holzworth (donh@gcx1.ssd.csd.harris.com)
Michael Johnson (johnsonm@stolaf.edu)
Karl Heinz Kremer (khk@raster.kodak.com)
Pat Mackinlay (mackinla@cs.curtin.edu.au)
...others too numerous to mention...
All the driver writers!

...and of course, Linus "That's LIN-uhks" Torvalds and all the guys who helped develop Linux into a BLOODY KICKIN' O/S!

...and now a word of warning:

Messing about with drivers is messing with the kernel. Drivers are run at the kernel level, and as such are not subject to scheduling. Further, drivers have access to various kernel structures. Before you actually write a driver, be *damned* sure of what you are doing, lest you end up having to re-format your harddrive and re-install Linux!

The information in this Tutorial is as up-to-date as I could make it. It also has no stamp of approval whatsoever by any of the designers of the kernel. I am not responsible for damage caused to anything as a result of using this Guide.

End of Introduction

CHAPTRE THE FIRSTE : How did *they* get the device driver in the kernel?

You have to realize that device drivers really are part of the kernel. The kernel can hook in to the functions in your device driver if you tell it the addresses of some standard functions. These standard functions are detailed in the Guide.

As a part of the kernel, the code of the device driver must be compiled in *with* the kernel. That is, you must alter some Makefiles to compile your driver and to get it archived into the chr_drv.a library, or you can archive it yourself and link it in to the kernel at a later compile stage.

The first step, before you even write a single line of driver code, is to make sure you know how to recompile the kernel. Then go ahead and actually do it, to be sure you (and your system) are sane. Of course, you need the sources to the kernel. If you have the SLS distribution of Linux, you already have the sources in /linux. If you don't have the sources, you can get it at one of these fine ftp sites near you:

```
tsx-11.mit.edu:/pub/linux
sunsite.unc.edu:/pub/Linux
```

Briefly, here's how to compile the kernel (at least this is how it's done in the SLS release):

Go to /linux (or wherever the source for Linux is)
You will see a directory which looks a lot like this:

```
-rw-r--r--  1 baruch      17982 Nov 10 07:54 COPYING
-rw-r--r--  1 baruch       1444 Jan 13 15:24 Configure
-rw-r--r--  1 baruch       6934 Feb 22 13:31 Makefile
-rw-r--r--  1 baruch       4078 Dec 12 06:45 README
drwxrwxr-x  2 baruch        512 Feb 22 13:34 boot
-rw-r--r--  1 baruch       1724 Feb  9 15:07 config.in
drwxrwxr-x  8 baruch        512 Feb 22 13:34 fs
drwxrwxr-x  4 baruch        512 Dec  1 19:40 include
drwxrwxr-x  2 baruch        512 Feb 22 13:34 init
drwxrwxr-x  5 baruch        512 Feb  9 15:11 kernel
drwxrwxr-x  2 baruch        512 Feb  9 15:11 lib
-rwxr-xr-x  1 baruch        166 Nov 10 07:54 makever.sh
drwxrwxr-x  2 baruch        512 Feb 22 13:34 mm
drwxrwxr-x  3 baruch        512 Feb  9 15:11 net
drwxrwxr-x  2 baruch        512 Feb 22 13:34 tools
drwxrwxr-x  2 baruch        512 Feb 22 13:34 zBoot
```

The README file should contain instructions, but here's how anyway:

Log in as root.

```
make clean    (Do this only once.  Otherwise you'll have to sit around
               for 45 minutes or so while the whole thing recompiles)
make config   (Answer the questions -- usually needed only the first time)
make dep      (Makes dependencies)
make          (makes the kernel)
```

You should end up with an Image file. This is the kernel. Put it where you like (LILO users should take it from there). To make a bootable disk, just pop a DOS formatted disk in drive A, and do:

```
make disk
```

CHAPTER TWO: The simplest driver you've ever seen.

Now, the directory you're interested in is <src>/kernel/chr_drv. This is where all the character device drivers are kept. Go to that directory. Open up a new file, and call it testdata.c. Here is what you should put in it:

```
=======================================
      File Listing 1: testdata.c
=======================================

#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/tty.h>
#include <linux/signal.h>
#include <linux/errno.h>

#include <asm/io.h>
#include <asm/segment.h>
#include <asm/system.h>
#include <asm/irq.h>

unsigned long test_init(unsigned long kmem_start)
{
  printk("Test Data Generator installed.\n");
  return kmem_start;
}
```

The include files are all there for convenience. You may need them later.
All this driver does is upon initialization, display a message.

Now, to get this driver into the kernel, you need to do several things.
The first two things do in the chr_drv directory:

I. Get the kernel to call your init function on bootup. To do this, edit the mem.c file, and go to the very end to the function chr_drv_init. It looks something like this:

```
long chr_dev_init(long mem_start, long mem_end)
{
      if (register_chrdev(1,"mem",&memory_fops))
            printk("unable to get major 1 for memory devs\n");
      mem_start = tty_init(mem_start);
      mem_start = lp_init(mem_start);
      mem_start = mouse_init(mem_start);
      mem_start = soundcard_init(mem_start);
      return mem_start;
}
```

You need to add your test_init function to the code. Put it right before the return:

```
mem_start = test_init(mem_start);
```

Save the file.

II. Edit the Makefile to compile testdata.c. Edit the Makefile, and add testdata.o to the OBJS list. This will cause the make utility to compile testdata.c into an object file, and then add it to the chr_drv.a library archive.

Save the file.

The next step is to re-compile the kernel. Go to the <src> directory, and do a make from the top as described in the first chapter. There is no point in doing a "make clean" or "make config". If all goes well, the make should proceed down to chr_drv, and compile your testdata.c file. If there are warnings or errors, do a ctrl-C to break out of the make, and fix the problem.

Once you are left with an Image file, put the Image file where LILO wants it, or use "make disk" to make a bootable disk. It's a good idea to save your old Image file (or save the disk it was on).

Now reboot. When Linux comes up again, you should see your message printed on bootup after all the character devices' messages, before any of the block device messages. If the message came up, have a soda. Jump up and down a little. (Well, first jump, _then_ have the soda).

If it didn't work, go back and find out what you did wrong. Are you sure you recompiled the kernel? Did it recompile with testdata.c? Did you reboot using the new kernel? Are you sure? Are you root? Maybe your kernel is bad or old. I have used 0.99pl6, with the new libc.so.4.3.2 shared library successfully, and I am currently using 0.99pl8 with libc.so.4.3.3.

CHAPTER THREE: A device driver that actually does something useful.

This example is taken from the _Writing UNIX Device Drivers_ book by George Pajari, published by Addison Wesley. It can usually be found in a Barnes and Noble bookstore, or any large bookstore which has a nice section on UNIX. The ISBN is 0-201-52374-4, and it was published in 1992. This book is highly recommended for the device driver writer.

This device driver will actually be read from. You can open and close it (which really won't do much), but the biggest thing it will do is allow you to read from it. This driver won't access any external hardware, and so it is called a "pseudo device driver". That is, it really doesn't drive any device.

Have your Guide handy? OK, now alter your testdata.c file so that it looks like this:

```
=====================================
      File Listing 2: testdata.c
=====================================

#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/tty.h>
#include <linux/signal.h>
#include <linux/errno.h>

#include <asm/io.h>
#include <asm/segment.h>
#include <asm/system.h>
#include <asm/irq.h>

static char test_data[]="Linux is really funky!\n";

static int test_read(struct inode * inode, struct file * file,
                     char * buffer, int count)
{
  int offset;

  printk("Test Data Generator, reading %d bytes\n",count);
  if (count<=0) return -EINVAL;
  for (offset=0; offset<count; offset++)
    put_fs_byte(test_data[offset % (sizeof(test_data)-1)], buffer+offset);
  return offset;
}

static int test_open(struct inode *inode, struct file *file)
{
  printk("Test Data Generator opened.\n");
  return 0;
}

static void test_release(struct inode *inode, struct file *file)
{
  printk("Test Data Generator released.\n");
}

struct file_operations test_fops = {
        NULL,           /* test_seek */
        test_read,      /* test_read */
        NULL,           /* test_write */
        NULL,           /* test_readdir */
        NULL,           /* test_select */
        NULL,           /* test_ioctl */
        NULL,           /* test_mmap */
        test_open,      /* test_open */
        test_release    /* test_release */
};

unsigned long test_init(unsigned long kmem_start)
{
  printk("Test Data Generator installed.\n");
  if (register_chrdev(21,"test",&test_fops));
    printk("Test Data Generator error: Cannot register to major device 21!\n");
  return kmem_start;
}

=====================================
```

OK, let's go over this. Look first at the test_init function. Notice the new function -- register_chrdev. This registers the character device with the kernel as using major device number 21. All devices (except for the really simple one in the last chapter) use major device numbers to be accessed. The kernel has an internal table of devices and their associated device functions which is indexed by major device number.

The device numbers go from 0 to MAX_CHRDEV-1. MAX_CHRDEV is defined in linux/fs.h, and is currently set at 32. In general, you want to stay away from devices 0-15 because those are reserved for the "usual" devices. Currently, these usual devices (according to the FAQ) are as follows:

---Excerpt from FAQ begins---

QUESTION: What are the device minor/major numbers?

The Linux Device List

```
    maintained by rick@ee.uwm.edu (Rick Miller, Linux Device Registrar)
                          February 17, 1993

Many thanks to richard@stat.tamu.edu, Jim Winstead Jr., and many others.

Majors:
 0.  Unnamed .  (unknown) ....  for proc-fs, NFS clients, etc.
 1.  Memory ..  (character) ..  ram, mem, kmem, null, port, zero
 2.  Floppy ..  (block) ......  fd[0-1]<[dhDH]{360,720,1200,1440}>
 3.  Hard Disk  (block) ......  hd[a-b]<[0-8]>
 4.  Tty .....  (character) ..  {p,t}ty<{S,[p-s][0-f]}><#>
 5.  tty .....  (character) ..  tty, cua[0-63]
 6.  Lp ......  (character) ..  lp[0-2] or par[0-2]
 7.  Tape ....  (block) ......  t[0-?] (reserved for Non-SCSI tape drives)
 8.  Scsi Disk  (block) ......  sd[a-h]<[0-8]>
 9.  Scsi Tape  (block) ......  <n>rmt[0-1]
10.  Mouse ...  (character) ..  bm, psaux (mouse)
11.  CD-ROM ..  (block) ......  scd[0-1]
12.  QIC-tape?  (character) ..  rmt{8,16}, tape<{-d,-reset}>
13.  XT-disk .  (block) ......  xd[a-b]<[0-8]>
14.  Audio ...  (character) ..  audio, dsp, midi, mixer, sequencer

---Excerpt from FAQ ends---
```

The FAQ goes on to break down the major devices by minor numbers. Each major device can be broken down into at most 256 minor devices (0-255). The device driver can determine which minor it is supposed to operate on. More on that later.

In any case, I've chosen major 21 for experimentation purposes. By the way, the name of the driver (here it's "test") is not important. The kernel does not do anything with it. [It would be nice if it would. Then you could interrogate the kernel and find out what drivers are installed!]

register_chrdrv also takes in a pointer to a file_operations structure. This structure tells the kernel which function to call for which kernel operation. The details of this structure is given in the Guide. For now, what is important is that we are telling the kernel to call test_read for read operations, test_open for open operations, and test_close for release operations.

If a driver has already taken major 21, register_chrdrv will return -EBUSY. Here, all we do is print a message saying that 21 is already taken.

Now, the test_open and test_release functions just print out things to the console. They are really there for debugging purposes, so that you can see when things happen.

The meat of the driver is the test_read function. The first thing it does is print out how many bytes were requested. Then it puts that many bytes into user space. Remember that the driver is executing at the kernel level, and the user space will be differnet from kernel space. We have to do some kind of translation to put the data which is in kernel space into the buffer which is in user space. We use here the put_fs_byte function.

The loop puts the string into the buffer, going back to the beginning of the string if necessary. Once the loop is finished, we just return the actual number of bytes read. The actual number may be different from the requested number. For example, you may be reading from the driver some kind of message which has a fixed size. You may want to code the driver so that if you attempt to read more than the message size, you will get only the message size, and no more. Here, we just give the process however many bytes it wants.

Now, let's get this driver into the kernel. But first what we'll do is create a special file which can be opened, read, and closed. Operations on this special file will activate your driver code.

The special files are normally stored in the /dev directory. Do this:

```
mknod /dev/testdata c 21 0
chmod 0666 /dev/testdata
```

This makes a special character (c) file called testdata, and gives it major 21, minor 0. The chmod makes sure that everyone can read and write the device.

Now recompile the kernel, and reboot. Once again, make sure you fix any warnings or errors in your testdata.c compilation.

Now, go to the /tmp directory (or whereever you want), and write this program:

```
=======================================
      File Listing 3: data.c
=======================================

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

void main(void)
{
  int fd;
  char buff[128];

  fd = open("/dev/testdata",O_RDWR);
  printf("/dev/testdata opened, fd=%d\n",fd);
  if (fd<=0) exit(0);
  printf("sizeof(buff)=%d\n",sizeof(buff));
  printf("Read returns %d\n",read(fd,buff,sizeof(buff)));
  buff[127]=0;
  printf("buff=\n'%s'\n",buff);
  close(fd);
}

=======================================
```

Compile it using gcc. Run it. If it said "Linux is really funky!" lots of times, pat yourself on the back (or whereever you want) for a job well done. If it didn't, check the output, and see where you went wrong. It could just be that you have a bad or old kernel.

The last line may be partial, since you're only printing out 127 characters.

```
++++++++++++++++++++++
    EXPERIMENT 1
++++++++++++++++++++++
```

Use mknod to make another special file, this one with minor 1. Call it something like /dev/testdata2. Change the device driver so that in the read call, it finds out which minor is being read from. Use this:

```
  int minor = MINOR(inode->i_rdev);
```

Print out the minor number, and depending on which minor it is, read from a different message string. Test your driver with code similar to data.c.

```
++++++++++++++++++++++
```

CHAPTER FOUR: You've learned to read, now you're gonna learn to write.

Now that you're reading strings, you may want to write strings and read them back. We'll go through two versions of this -- one that uses static memory, and one that dynamically allocates the memory.

Keeping your current driver, all you need to do is add a write function to it, not forgetting to put that write function into the file_operations structure of the driver.

Add this section of code to your driver above the file_operations structure declaration:

```
=======================================
  File Listing 4 (partial): testdata.c
=======================================

static char test_data[128]="\0";
static int test_data_size=0;

static int test_write(struct inode * inode, struct file * file,
  char * buffer, int count)
{
  printk("Write %d bytes\n",count);
  if (count>127) return -ENOMEM;
```

```
  if ((!test_data_size) || (count<=0)) return -EINVAL;
  memcpy_fromfs((void *)test_data, (void *)buffer, (unsigned long)count);
  test_data[127]=0;  /* NUL-terminate the string if necessary */
  test_data_size = count;
  return count;
}
```

==========

Also, alter the test_read function so that instead of using sizeof(test_data) as the size of the test_data string, it uses test_data_size.

In the test_write function, I have decided to prevent the acceptance of strings which are too big to fit (with a NUL-terminator) into the test_data area, rather than just writing only what fits. In this case, if the offered string is too long, I return ENOMEM. The write function in the user's process will return <0, and errno will be set to ENOMEM.

Also note that I have used the memcpy_fromfs function, which is real convenient -- much more convenient than looping a put_fs_byte.

Compile this driver, and test it by modifying data.c to write some data, then read it back.

++++++++++++++++++++++
    EXPERIMENT 2
++++++++++++++++++++++

Re-write the driver so that it can have two different strings for the two minor devices as in experiment 1.

++++++++++++++++++++++

Now that we can write data to the driver, it would be nice if we could dynamically allocate memory to store a string in. We will use kmalloc to do this. (Why is discussed later)

One thing which must be realized with kmalloc -- it can only allocate a maximum of one Linux page (4096 bytes). If you want more, you will have to create a linked list.

Change your driver so that instead of listing 4, you have this:

==========
  File Listing 5 (partial): testdata.c
==========

```
static char *test_data=NULL;
static int test_data_size=0;

static int test_write(struct inode * inode, struct file * file,
  char * buffer, int count)
{
  printk("Write %d bytes\n",count);
  if (count>4095) return -ENOMEM;
  if (test_data!=NULL) kfree_s((void *)test_data, test_data_size);
  test_data_size = 0;
  test_data = (char *)kmalloc((unsigned int)count, GFP_KERNEL);
  if (test_data==NULL) return -ENOMEM;
  memcpy_fromfs((void *)test_data, (void *)buffer, (unsigned long)count);
  test_data[count]=0;  /* NUL-terminate the string if necessary */
  test_data_size = count;
  return count;
}
```

==========

Here, instead of statically allocating memory for the string, we dynamically allocate it using kmalloc. Note first, that if we had already allocated a string, we free it first by using kfree_s. This is faster than using kfree, because kfree would have to search for the size of the object allocated. Here we know what the size was, so we can use kfree_s. kmalloc vs. malloc is discussed below.

Next, note that we use the GFP_KERNEL priority in the kmalloc. This causes the process to go to sleep if there is no memory available, and the process will wake up again when there is memory to spare. In general, the process will sleep until a page of memory is swapped out to disk.

In the event of catastrophic memory non-availability, kmalloc will return

NULL, and we should handle that case. Unfortunately here, we have already freed the previous string -- although that could be changed easily by kmallocing, then kfreeing.

The rest of the code reads as in listing 4.

When we get into the section on interrupt handling, we will discuss the use of GFP_ATOMIC as a kmalloc priority.

A brief excursion into kmalloc vs. malloc:

The malloc() call allocates memory in user space, which is fine if that's what you want. Here, we want to have the driver store information so that *any* process can use it, and so we have to allocate memory in the kernel. That means, kmalloc(). Further, there is a maximum of 4096 bytes which can be allocated in any one call of kmalloc. This means that you cannot be guaranteed to get contiguous space of over 4096 bytes. You will have to use a linked list of kmalloced buffers.

Alternatively, you can fool with the init section of the driver, and reserve contiguous space for yourself on init (but then it may as well be statically allocated).

CHAPTER FIVE: For my next trick, I...fall....a...sleep (SNNXXXX!!)

The thing which really saves multitasking operating systems is that many process sleep when waiting for events to occur. If this were not true, processes would always be burning cycles, and there would really be no big difference between running your processes at the same time, or one after the other.

But when a process sleeps, other processes get to use the CPU. In general, processes sleep when an event they are waiting for has not yet happened. The exception to this is processes which are designed to do work when nothing is happening. For example, you might have a process sitting around using cycles to calculate pi out to a zillion digits. That kind of background process should have its priority set real low so that it isn't executed often when other (presumably more important) processes have work to do.

Since processes sleep when waiting for events, and said events are usually handled by drivers, drivers must cause the processes which called them to sleep if not ready. This is the idea behind the select() call, which will be dealt with in a later chapter.

To illustrate sleeping and waking processes, we will alter our driver from listing 2 by adding a new write function and changing the read function around as follows:

File Listing 6 (partial): testdata.c

```
static char test_data[]="Linux is really funky!\n";
static int wakeups = 0;
static struct wait_queue *wait_queue = NULL;

static int test_write(struct inode * inode, struct file * file,
  char * buffer, int count)
{
  int i;

  printk("Write %d bytes\n",count);
  wake_up_interruptible(&wait_queue);
  printk("Woke %d processes.\n",wakeups);
  wakeups = 0;
  return count;
}

static int test_read(struct inode * inode, struct file * file,
                     char * buffer, int count)
{
  int offset;

  printk("Test Data Generator, reading %d bytes\n",count);

  printk("Process going to sleep\n");
  wakeups++;
```

```
  interruptible_sleep_on(&wait_queue);
  printk("Process has woken up!\n");

  for (offset=0; offset<count; offset++)
    put_fs_byte(test_data[offset % (sizeof(test_data)-1)], buffer+offset);
  return offset;
}
```

==========

Don't forget to put the test_write function in the file_operations struct!
But don't compile this driver just yet! Read on...

The operation of this driver is as follows: On a read, put the process to sleep. On a write, wake up all those processes which have gone to sleep in this driver. This will allow the processes to complete the read.

There are two new variables here, wakeups and wait_queue. The wait_queue is a circular queue of processes which are sleeping. It is FIFO, so that the process woken up is the first process which went to sleep.

The kernel handles the queue for us; all we need to do is supply a pointer to the queue and initialize it to NULL (i.e., the queue is empty).

We'll use the wakeups variable to tell us how many processes are taken off the wait_queue (i.e., woken up) -- which is the number of processes which have already gone to sleep. So each time a process is slept on, we increment wakeups. When a write request comes in, we wake up wakeups processes and reset wakeups to zero.

Simple, yes? Now we get into the sticky part.

In the Guide, you see that you can choose two ways of sleeping -- interruptible or not. Interruptible sleeps can be interrupted (i.e., the process is woken up) by signals (such as SIGUSR) and hardware interrupts. Non-interruptible sleeps can only be interrupted by hardware interrupts. Not even a kill -9 will wake up a non-interruptible process which is sleeping! Suppose you have a signal handler in your process which will react to signal 30 (SIGUSR). That is, you can do kill -30 <pid>. What happens?

When the scheduler gets around to checking the signalled process for runnability, it sees that there is a signal pending. This allows the process to continue to run where it left off, with a twist: when the process leaves kernel mode (the driver call) and enters user mode, the signal handler is called (if there is one). Once the signal handler function exits, one of two things can happen:

(1) If the original system call exited with -ERESTARTNOINTR, then the process will continue as if it calls the system call again with the same arguments.
(2) If the original system call did not exit with -ERESTARTNOINTR, but with -ERESTARTNOHAND or -ERESTARTSYS, then the process will continue exitting from the system call with -1, errno -EINTR.
(3) If the original system call did not exit with -ERSTARTNOINTR, -ERESTARTNOHAND, or -ERESTARTSYS, then the process will continue, exitting from the system call with whatever was returned.

You can see most of this (if you can read mutilated 80386 assembly) in <src>/kernel/sys_call.S and <src>/kernel/signal.c. Although signal handling has been considerably revamped for 0.99pl8, the basic sequence of operations is intact across patch levels. -ERESTARTNOHAND is new in 0.99pl8.

This is important -- the driver call should not be completed except for cleanup, since the kernel will return an error for you or redo the system call.

When the process continues to run before calling the signal handler, it picks up where it left off -- in the interruptible_sleep_on function. This function takes the process off the wait_queue automatically (which is nice). But then wakeups is not updated (which is not so nice). In that case, when a subsequent write comes in, the number of sleeping processes reported will be wrong!

[pulpit-pounding mode on]

Although for this driver ignoring this is not such a big deal, it is sloppy programming for a driver. Driver code must be so perfect that it operates like a well-oiled machine, with no slip-ups. One error -- one bit of code that gets out of sync -- and you can at least annoy users and make them throw up their hands in frustration, and at worst panic the kernel and make users

throw your code away in frustration! Also, there is nothing worse than spending time debugging an application when the bug is in the driver, or trying to code around a known driver flaw.

[pulpit-pounding mode off]

So how do we solve this out-of-sync problem?

Fact: ignoring interrupts, all processes are atomic when they are in the kernel. That is, unless a process performs an operation which can sleep (like the call to kmalloc we visited above), or a hardware interrupt comes in, the flow of execution goes from entering the kernel to leaving the kernel, with no time taken out to run anything else. This does not mean that the code in user space gets to continue to run. If the process leaves the system call and is not eligible to run, other processes may run and then later on the system call appears to have returned to the process. More on that later.

That fact is good to know. It means that as long as we are sure upon entering the test_write call that wakeups contains the correct number of sleeping processes, test_write will work 100%. That is, unless a hardware interrupt comes in which causes the driver to execute an interrupt handler, we are safe, but here we have no such handler, and so we can ignore that for now. We will deal with interrupts in a later chapter.

So we know that write doesn't really have to be changed. It's really the read that we're concerned about. What we need to do is after we get out of interruptible_sleep_on() we see if we were genuinely woken up through a wakeup call, or if we were signalled. If we were signalled, then we know that the write call wasn't the cause of the wakeup, and so we should really decrement wakeups.

Now for some loose ends. Remember that upon signalling, the kernel only flags the signal for the process, and sets the process to a runnable state. That does not mean that it can run immediately. Another process may get to run first, and that process may very well run the driver's write code, waking up all processes. Of course, we can consider the signalled process to be still asleep when it gets the signal, because it has not yet run its signal handler. So when that other process gets to run the write code, the number of sleeping processes is indeed correct, and wakeups is set to 0.

But now, when the signalled process is run again, the read code will attempt to decrement wakeups, making it -1! The next write will display the wrong number of sleeping processes!

One thing saves us -- the fact that we can detect in the read code that the write code was executed, simply because wakeups is 0. Remember that wakeups is incremented before the sleep, so it is guaranteed to be greater than 0 if the write code was not executed before waking up because of a signal.

So if the write code was executed, it really does not make sense to decrement wakeups, so we just say that only if wakeups is non-zero do we decrement.

To implement all this, add this code after the sleep:

```
=========================================
  File Listing 7 (partial): testdata.c
=========================================

if (current->signal & ~current->blocked) /* signalled? */
{
  printk("Process signalled.\n");
  if (wakeups) wakeups--;
  return -ERESTARTNOINTR; /* Will restart call automagically */
}

=========================================
```

Now that you've got that straightened out, let's add some more confusion to the mix. Suppose you're in the driver call, doing nice things, and then all of a sudden a nasty timer interrupt (task switch possibility) comes in. What now? Will there be a task switch? No. A RUNNING task in the kernel cannot be switched out, otherwise all hell would break loose. Whew! I'm glad we don't have to pay attention to that!

Well, now that we've gone through all the possible ways signals can make your insides twist, you can code the driver. Remember to put listing 7 into listing 6!

Here's how we're going to test this driver. Several processes will call read (and sleep). When they wake up, they're going to say that they were woken up (as opposed to printing out what they just read -- we already know that works). One process will do a write to wake the other processes up. This is the trigger process. Here is the code for the two types of processes:

File Listing 8: data.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>

/* The reader process */

void signal_handler(int x)
{
  printf("Called signal handler\n");
  signal(SIGUSR1, signal_handler); /* Reset signal handler */
}

void main(void)
{
  int fd;
  char buff[128];
  int rtn;

  signal(SIGUSR1, signal_handler); /* Setup signal handler */

  fd = open("/dev/testdata",O_RDWR);
  printf("/dev/testdata opened, fd=%d\n",fd);
  if (fd<=0) exit(0);

  rtn = read(fd,buff,sizeof(buff));
  printf("Read returns %d\n",rtn);
  if (rtn<0)
  {
    perror("read");
    exit(1);
  }
  printf("Process woken up!\n");
  close(fd);
}
```

File Listing 9: trigger.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>

/* The writer process */

void main(int argc, char **argv)
{
  int fd;
  char buff[128];
  int rtn;

  fd = open("/dev/testdata",O_RDWR);
  printf("/dev/testdata opened, fd=%d\n",fd);
  if (fd<=0) exit(0);

  if (argc>1)
  {
    kill(atoi(argv[1]),SIGUSR1);
    exit(0);
  }
  rtn = write(fd,buff,sizeof(buff));
  if (rtn<0)
```

```
    {
      perror("write");
      exit(1);
    }
    close(fd);
}
```

==========

Compile these programs using gcc. Now run two or three of the data processes:

```
  data &
```

The last thing each of these processes should print is

```
  Process going to sleep.
```

because all of these processes are asleep. Now run the trigger program:

```
  trigger
```

This should wake up all the other processes, which should say,

```
  Process woken up!
```

Had the read function returned an error (like EINTR), they would have said

```
  read: <error text>
```

Now, let's test to see if the signal detection and restart mechanism works. Run a single data process in the background via "data &". Remember it's pid. Now, run the trigger process with that pid as an argument:

```
  trigger <pid>
```

This will signal <pid> instead of waking it up via write. The driver should say,

```
  Process signalled.
  Called signal handler
```

but the process should not wake up, since we restarted the call. Only a write will stop the call.

++++++++++++++++++++
EXPERIMENT 3
++++++++++++++++++++

Re-write the driver so that instead of always restarting the call, it returns with EINTR on signal when the read call's count is a special value or values (say anything less than 1000). Test to see if the read call returns EINTR when the trigger program signals the reading process.

++++++++++++++++++++

CHAPTER SIX: I want this, that, that...no, THIS, and that. Or, selects!

The select call is one of the most useful calls created for interfacing to drivers. Without it, or a function like it, if you wanted to check a driver for readiness, you would have to poll it regularly. Worse, you would not be able to check multiple drivers for readiness at the same time!

But enough of this. You have select, so rejoice and be happy.

As already implied by the first paragraph, the select system call allows a process to check multiple drivers for readiness. For example, suppose you wanted the process to sit around and wait for one of two file descriptors to be ready for reading. Usually, if a descriptor is not ready for reading and you read it, it will put your process to sleep (or "block"). But you can only read one file descriptor at a time, and here you want to essentially block on _two_ fd's.

In that case, you use the select call. The syntax of select was already explained in the Guide, so let's go about implementing a select function in our driver.

Add the following code to the driver, and put the test_select function in

the fops structure:

File Listing 10 (partial): testdata.c

```
static int test_select(struct inode *inode, struct file *file,
                       int sel_type, select_table *wait)
{
  printk("Driver entering select.\n");
  if (sel_type==SEL_IN) /* ready for read? */
  {
    if (wakeups) /* Any process is sleeping in here */
    {
      select_wait(&wait_queue, wait);
      printk("Driver not ready\n");
      return 0;  /* Not ready yet */
    }
    return 1;  /* Ready */
  }
  return 1;  /* Always ready for writes and exceptions */
}
```

Here's what this function does. When a process issues a select call with this driver as one of the fd's to select on, the kernel will call test_select with sel_type being SEL_IN. If wakeups is non-zero (that is, processes have read without a process writing) then we will say that the driver is not ready for reading. In this case, select_wait will add the process to the wait_queue and immediately return. The return of 0 indicates that the driver is not ready for the operation.

For any other type of operation (or if there are no processes sleeping in read) we say the driver is ready (return 1).

The only thing that must be remembered is that we are using the same wait_queue structure for processes sleeping in read and processes sleeping in select. This means that writing to the driver will wake up both types of processes. If desired, a different wait_queue could be used, and the appropriate wake up code would have to be written.

Compile this new code into the kernel. We will test this driver by writing a new type of process which will call the select system call. Here is the new process' code:

File Listing 11: sel.c

```
#include <stdio.h> /* Doesn't hurt, can only help! */
#include <fcntl.h>
#include <sys/time.h> /* For FD_* and select */

void main(void)
{
  int fd;
  int rtn;
  fd_set read_fds;

  fd = open("/dev/testdata", O_RDWR);
  printf("/dev/testdata opened, fd=%d\n",fd);
  if (fd<=0) exit(0);
  printf("Entering select...\n");
  FD_ZERO(&read_fds);
  FD_SET(fd,&read_fds);
  rtn = select(&read_fds, NULL, NULL, NULL);
  if (rtn<0)
  {
    perror("select");
    exit(0);
  }
  printf("Select returns %d\n",rtn);
}
```

When the kernel is re-loaded, the first test we will perform is to see whether the select call returns immediately given that no processes are sleeping in read. Just run sel -- no need to run it in the background.

You should see something like:

```
Entering select...
Driver entering select.
Select returns 1
```

This is as it should be -- select has determined that one file descriptor is ready for reading.

Our next test is to see whether select sleeps properly. Run this:

```
data &
sel &
trip
```

When sel is run, you should see:

```
Entering select...
Driver entering select.
Read not ready
Driver entering select.
Read not ready
```

The select call in the kernel calls the test_select function again once if the first time the driver is not ready. However, the process is only added to the wait queue once -- the first time.

Once the trip program is run, you should see:

```
Process has woken up!
Read returns 1024
Driver entering select.
Select returns 1
```

That is, the data process woke up due to the write, as did the sel process. Note that the test_select function is called once again when the sel process is woken up. This is also a consequence of the kernel design, and is nothing to worry about. Those who are interested in the inner workings of the select call should look in the file <src>/fs/select.c.

A word about signals and select. Since the select call in the driver does not return any error code -- just 0 or non-0 -- there is no way to decide whether the select call should be restarted or not. Select will return -1, errno EINTR if interrupted by a signal.

CHAPTER SEVEN: This next chap -- oh, hello! -- this next chapter is about interrupts.

This chapter will be one of the most difficult chapters to go through as a tutorial, since some means of generating interrupts must be used to test things with. Furthermore, the interrupt must be one which is currently unused by the system, and one must be willing to mess around with a hardware device which is connected to the IRQ.

I will start out with something more controlled than external interrupts -- internal, or software, interrupts.

Why internal interrupts? There really is not such a big difference between internal and external interrupts. Certainly an IRQ is generated by a hardware device, but the hardware IRQ results in a software interrupt. I will discuss the required changes for dealing with hardware rather than software interrupts later in this chapter.

Note: The following paragraphs deal with 80386/80486 specific stuff. Those who are not really interested in the "why" of Linux interrupts may skip ahead!

To be able to use interrupts, we must first understand how Linux handles interrupts. Interrupts most often require a transfer of execution control from one code segment to another, and this may be accomplished in two ways. The first is by specifying the descriptor of the other executable segment, and the second is by a "gate".

In Linux, three functions are used to initialize gates: set_intr_gate, set_trap_gate, and set_system_gate.

set_intr_gate sets up a 32-bit interrupt gate with descriptor privilege level (DPL) 0 (the most privileged level).

set_trap_gate sets up a 32-bit task gate with DPL 0.

set_system_gate sets up a 32-bit task gate with DPL 3.

Each of these setups enter the gate into the interrupt descriptor table (IDT) so that when an INT n instruction is performed, the gate in the IDT corresponding to n is executed.

THIS ENDS 80386/80486 DISCUSSION.

The three Linux calls allow us to install an interrupt handler for any interrupt from 0x00 to 0xFF. We will use set_intr_gate to install an interrupt handler into interrupt 0x90.