Linux Device Drivers

Dr. Wolfgang Koch

Friedrich Schiller University Jena

Department of Mathematics and Computer Science

Jena, Germany

wolfgang.koch@uni-jena.de

Linux Device Drivers

1. Introduction
2. Kernel Modules
3. Char Drivers
4. Advanced Char Drivers
5. Interrupts

3. Char Drivers

- File Operations
- Device Files, Major & Minor Numbers
- file_operations Structure
- register_chardev, Choice of Major Number
- mknod
- register_chardev_region – the new way

3. Char Drivers – cont.

- read( ), put_user( )
- open( ), release( ), Usage Count
- file Structure, llseek( )
- write( ), get_user( )
- Race conditions, Atomic Variables
- Spinlocks, Semaphores

File Operations

in UNIX (Linux) input/output devices are treated very much like ordinary files (remember –
file descriptor 0: `standard input`, fd 1: `standard output`)

in this way easy redirection of input and output is possible

applications use the same system calls:
`open(), read(), write(), ioctl(), ... close()`

both (files and devices) can be accessed as a stream of bytes
both are represented as nodes in the file system

File Operations

classes of devices (drivers) –

- **Character (char) devices:**
  can be accessed as a stream of bytes (characters)
- **Block devices:**
  read/write from/to the hardware – whole blocks (e.g. 2 KBytes)
  fast access to the hardware (using DMA) –
  filesystems, storage devices
  but usually the user can use the same (unblocked) system calls:
  `open(), read(), write(), ...` (buffered in/output)

both types are accessed through a filesystem node

File Operations

char devices, block devices

3rd type:

Network interfaces
different from char or block drivers
(data packages instead of streams,
different system calls, no filesystem nodes)

in this tutorial we only deal with **char drivers**
(they are relatively simple, but usually sufficient,
most drivers are char drivers)

File Operations

applications use the same system calls to access devices as with ordinary files (`#include <unistd.h>, <fcntl.h>`):

```
open()
read()
write()
ioctl()
lseek()
...
close()
```

(see `'man 2 open'` etc.
closer descriptions are given
in the following chapters)

to every supported function there is a counterpart (a method)
in the driver (not all functions are always supported:
`lseek()` not in serial input)

Device Files

input/output devices are treated very much like ordinary files
both are represented as nodes in the filesystem

ordinary files, `ls -l` :

```
-rw-r--r--   1 nwk   users   130 2007-06-13 12:04 Makefile
-rw-r-----   1 nwk   users  1630 2007-05-27 13:59 rwko1.c
-r--r--r--   1 nwk   users  1928 2007-05-30 13:52 rwko1.o
-rw-r-----   1 nwk   users  2051 2007-05-27 14:51 rwko2.c
```

special files, device files, `ls -l /dev/` :

```
brw-------   1 nwk   disk   2, 0 2007-03-14 14:07 /dev/fd0
crw--w--w-   1 nwk   tty    4, 0 2007-08-21 14:14 /dev/tty0
crw-rw----   1 root  tty    4, 1 2007-08-25 09:25 /dev/tty1
crw-rw----   1 root  root  13,32 2007-03-14 14:07 /dev/mouse
```


Device Files

ordinary files, `ls -l` :

```
-rw-r--r--   1 nwk   users   130 2007-06-13 12:04 Makefile
-rw-r-----   1 nwk   users  1630 2007-05-27 13:59 rwko1.c
```

file size

special files, device files, `ls -l /dev/` :

```
brw-------   1 nwk   disk   2, 0 2007-03-14 14:07 /dev/fd0
crw--w--w-   1 nwk   tty    4, 0 2007-08-21 14:14 /dev/tty0
crw-rw----   1 root  tty    4, 1 2007-08-25 09:25 /dev/tty1
```

major, minor number

c – char device
b – block device   fd0, tty1, ...   “device files“

Major & Minor Number

```
crw--w--w-   1 nwk   tty    4, 0 2007-08-21 14:14 /dev/tty0
crw-rw----   1 root  tty    4, 1 2007-08-25 09:25 /dev/tty1
```

major, minor number

the major number identifies the driver associated with the device, different devices may have the same major number – they are managed by the same driver (may be in a different way according to the minor number) –
modern kernels allow multiple drivers to share a major number

an application identifies the device by its device file name, the kernel uses the major number at `open` time to dispatch execution to the appropriate driver

Major & Minor Number

```
crw--w--w-   1 nwk   tty    4, 0 2007-08-21 14:14 /dev/tty0
crw-rw----   1 root  tty    4, 1 2007-08-25 09:25 /dev/tty1
```

major, minor number

the major number identifies the driver associated with the device

the major number is a small integer (0 .. 255 in version 2.4), currently used numbers can be found in `/proc/devices`
(255 – a very limited number → 12 bit in kernel 2.6)

device files are created by the `mknod` command:

```
# mknod devfilename c major minor
 (mknod /dev/mydev c 253 0)
```

they can be removed by `# rm devfilename`

file_operations Structure

applications use system calls to access devices:

```
open(), read(), write(), ...
```

to every supported function there is a counterpart
(a method – like OOP, polymorphy) in the driver
(not all functions are supported in every driver)

the kernel uses the **file_operations structure** (defined in `<linux/fs.h>`) to access the driver's functions

for every possible function (system call) it contains a pointer to the function in the driver that implements this operation – or `NULL` for unsupported operations (maybe defaults)

file_operations Structure

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

ssize_t device_read(struct file *filp,
          char *buffer, size_t len, loff_t *offs);
int device_open   (struct inode *, struct file *);
int device_release(struct inode *, struct file *);

static struct file_operations fops = {
  read:     device_read,
  open:     device_open,
  release:  device_release,
  owner:    THIS_MODULE
};
```


file_operations Structure

```
static struct file_operations fops = {
  read:     device_read,
  open:     device_open,
  release:  device_release,
  owner:    THIS_MODULE
};
```

the tagged initialization of a structure (extension in gcc),
order doesn't matter, all the rest of the fields are set to `NULL`
→ portable (the definition of the structure often has changed)

`owner` field: used to maintain the usage count

register_chardev

the old way (still available in kernel 2.6 – emulated)

```
static int  major = 240;
static char dev_name[]="my_dev";
    // appears in /proc/devices

int rwko_init(void)
{
  int res;
  res = register_chrdev(major, dev_name, &fops);
  if (res<0) { print_x(...); return res; }

  return 0;
}
```

register_chardev

```
res = register_chrdev(major, dev_name, &fops);
if (res<0) { print_x(...); return res; }

#include <linux/fs.h>
int register_chrdev(
           unsigned int major,
           const char  *name,
           struct file_operations *fops
           );
```

return value: negative on failure

if `major=0` – dynamic allocation of a major number (-> res)

register_chardev

removing the driver, releasing the major number:

```
void rwko_exit(void)
{
  int k = unregister_chrdev(major, dev_name);
  if (k < 0) {
    sprintf(me,"exit error %d \r\n", k);
    print_x(me);
  }
}
```

`major` and `dev_name` must match, later release of the major number (after failing here) will be difficult, `exit()` has no return value – issue a warning !

Choice of Major Number

the major number identifies the driver associated with the device

the major number is a small integer (0 .. 255 in version 2.4), list of most common devices in `Documentation/devices.txt`
→ `240-254 local/experimental use`

currently used numbers in `/proc/devices`

if we call `register_chrdev(major, dev_name, &fops);`
with `major=0` –
we get a dynamically allocated major number

Choice of Major Number

if we call `register_chrdev(major, dev_name, &fops)`
with `major=0` – we get a dynamically allocated major number

```
int rwko_init(void)
{
  major = register_chrdev(0, dev_name, &fops);
  if (major<0) { print_x(); return major; }

  sprintf(me," Major: %d \r\n", major);
  print_x(me);   return 0;
}
```

drawback: we cannot run `mknod` in advance

mknod

dynamically allocated major number –
we cannot run `mknod` in advance

in `rwko1.c`:

```
major = register_chrdev(0, "rw1dev", &fops);
sprintf(me,“ Major: %d \r\n“, major);
print_x(me); ...
```

```
# insmod rwko1.ko          ==>  Major: 253

> less /proc/devices
Character devices:
...
253 rw1dev        <===
254 pcmcia
```


mknod

```
# insmod rwko1.o          ==>  Major: 253

> less /proc/devices
Character devices:
...
253 rw1dev
254 pcmcia

# mknod /dev/mydev c 253 0

>ls -l /dev/my*
crw-r--r--   1 root  root  253, 0 2007-08-21 15:10 /dev/mydev

# chmod 666 /dev/mydev          (if necessary)

# rm /dev/mydev
```


mknod

dynamically allocated major number –
we cannot run `mknod` in advance

1. run `insmod file.ko`
2. retrieve `major` (`/proc/devices`)
3. run `mknod /dev/name c major 0`
4. run `chmod` (if necessary)
5. check `ls -l /dev/`

this can be done with the help of a shell script using awk

mknod

this can be done with the help of a shell script (owner root) using awk:

```
#!/bin/sh
module="rwko1"
devnam="rw1dev"
device="mydev"

insmod ./$module.ko $* || exit 1

major=`awk "\\$2==\"$devnam\" {print \\$1}"  \
  /proc/devices`

# echo major = $major
mknod /dev/$device c $major 0
```

New way: register_chardev_region

old way (still available in kernel 2.6 – emulated)

```
res = register_chrdev(major, dev_name, &fops);
```

new way (in kernel 2.6)

```
int res = register_chrdev_region(
       dev_t first, uint count, char *dev_name);
```

or – for dynamicly allocated device numbers

```
int res = alloc_chrdev_region( dev_t *dev,
   uint firstminor, uint count, char *dev_name);
```


New way: register_chardev_region

```
int res = register_chrdev_region(
       dev_t first, uint count, char *dev_name);
```

internal representation of device numbers: type `dev_t`
type `dev_t` (kernel 2.6: 32 bit, 2.4: type `kdev_t` 16 bit ?)
holds both – major and minor number (12 / 20 bit)

don't rely on the bits, use macros (defined in `<linux/kdev_t.h>`

```
major = MAJOR(dev_t dev);
minor = MINOR(dev_t dev);
 dev  = MKDEV(int major, int minor);
```


New way: register_chardev_region

modern kernels allow multiple drivers to share a major number

```
int res = register_chrdev_region(
       dev_t first, uint count, char *dev_name);
```

`dev_t first` first device number (major / minor) of the region
`uint count` total number of contiguous device numbers (can cross major-number boundary)
`char *dev_name` device name (appears in `/proc/devices`)

There is no parameter `&fops` for the connection with our file-operations structure – we will need another way.

New way: register_chardev_region

```
int register_chrdev_region(
       dev_t first, uint count, char *dev_name);
```

for dynamically allocated device numbers

```
int alloc_chrdev_region( dev_t *dev,
   uint firstminor, uint count, char *dev_name);

dev_t dev;
res = alloc_chrdev_region(&dev, 0, 1, dev_name);
if (res<0) return res;
major = MAJOR(dev);
```

New way: register_chardev_region

```
int register_chrdev_region(
       dev_t first, uint count, char *dev_name);
```

or

```
int alloc_chrdev_region( dev_t *dev,
   uint firstminor, uint count, char *dev_name);
```

free the region (in the cleanup function)

```
void unregister_chrdev_region(
                   dev_t first, uint count);
```


New way: register_chardev_region

```
int register_chrdev_region(
       dev_t first, uint count, char *dev_name);
```

There is no parameter `&fops` for the connection with our file-operations structure – we will need another way.

Structure `cdev` (`<linux/cdev.h>`) to represent devices.

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

or (if defined in a structure) `struct cdev my_cdev;`

```
cdev_init(&my_cdev, &my_fops);
```


New way: register_chardev_region

Structure `cdev` (`<linux/cdev.h>`) to represent devices.

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

(or `cdev_init(&my_cdev, &my_fops);`

```
 my_cdev->owner = THIS_MODULE;  //  ???
```

Final step:

```
int cdev_add(struct cdev *,dev_t num, uint count);
```

Remove (in cleanup): `void cdev_del(struct cdev *);`

read( )

we can implement a `read()` method in our driver without having an `open()` method implemented:

an `open()` call in an application program is a system call, it calls the `open()` function in the kernel

this kernel function first initializes necessary data fields (e.g. the `file` structure) and then in turn calls the `open()` method of the driver

the `open()` method is supposed to initialize the device, if no initialization is required, no `open()`method is needed

read( )

the `read` field in the `file_operations` structure has the following prototype (different from the `read()` system call):

```
ssize_t (*read) (struct file *filp,
        char *buffer, size_t len, loff_t *offs);
```

the following parameters are provided by the caller (kernel) :

`struct file *filp`	– a pointer to the `file` structure
`char *buffer`	– a pointer to a **buffer in user space**
`size_t len`	– the number of bytes to be read
`loff_t *offs`	– a pointer to the `f_pos` field in the `file` structure (see below)

read( )

`ssize_t (*read) (struct file *filp, ... );`
is supposed to yield following **return values** (signed size type):

- a non-negative return value represents the number of bytes successfully read (may be less than `len` – this is no error)
- return value zero – end of file (it's no error)
  (if there will be data later, the driver should block)
- negative return value – error (v. `<asm/errno.h>`)

our method `device_read` must match this prototype:

```
static ssize_t device_read (struct file *filp,
   char *buffer, size_t len, loff_t *offs);
```


read( ), put_user( )

```
static ssize_t device_read (struct file *filp,
   char *buffer, size_t len, loff_t *offs);
```

in a first example driver we don't access a real hardware device – we just read from a buffer inside the driver

in order to demonstrate the properties of `read()` we only transfer 10 bytes per `read()` call

we have to transfer data from kernel space to user space:

```
#include <asm/uaccess.h>
put_user (char kernel_item, char *user_buff);
```


read( ), put_user( )

we have to transfer data from kernel space to user space:

```
put_user (char kernel_item, char *user_buff);
```

data transfer from kernel space to user space (and vice versa) cannot be carried out through pointers or `memcpy();`
one reason: memory in user space may be **swapped out**
(another reason: security holes)

we use macros and functions that can deal with page faults –

macro: `put_user(kernel_item, usr_ptr);`
fast, the size of the data transfer is recognized from `usr_ptr`
function:

```
ulong copy_to_user(void *to, void *from, ulong bytes);
```

read( ), put_user( )

memory in user space may be **swapped out** , we use macros and functions that can deal with page faults –

the page-fault handler can put the process to sleep
→ our method **must be re-entrant**
it must be capable of running in more than one context at the same time – don't keep status information in global variables

there are macros / functions that don't do the check and are faster:

```
__put_user(item, ptr);
ulong __copy_to_user(void *to, void *from, ulong bytes);
```

(test once using `access_ok()`)

read( ), example driver

```
static char mess[]="The goal of this tutorial ";
static char *mp;   // mp=mess; in init() or open()

static ssize_t device_read(struct file *filp,
       char *buffer, size_t len, loff_t *offs)
{
  unsigned int  i;

  for(i=0; i<10; i++){
    if(i==len) break;
    if(*mp==0) break;
    put_user(*mp++, buffer++);
  }

  if (debug>2) print_x(...);  return i;
}
```


read( ), example driver

we write an application (`app1`) containing:

```
int  fd, k=1 ;
char inbu[100];

fd = open("/dev/mydef", O_RDONLY);
...
while(k>0){
  k = read(fd,inbu,14);
  if (k<0){ perror(" read "); break;}
  inbu[k]=0;
  printf(" read %2d : %s \n", k, inbu);
}
```


read( ), example driver

call `app1` (with the example driver loaded) :

```
read 10 : The goal o
read 10 : f this tut
read  6 : orial
read  0 :
```

call `app1` again:

```
read  0 :
```

unless the driver was unloaded and reloaded again, since pointer `*mp` was set to `&mess` in `init()`
→ do it in `open()`

read( ), example driver

call `app1` (with the example driver loaded) :

```
read 10 : The goal o
read 10 : f this tut
read  6 : orial
read  0 :
```

but reading the device file with **`cat`** :

```
> cat /dev/mydev
The goal of this tutorial >
```

all bytes seem to be read in one go – `cat` reissues `read()` until it gets return value zero (`EOF`), in the same way work `fread()` and `fgets()` (in libc)

open( )

call `app1` again: `read 0`

unless the driver was unloaded and reloaded again, since pointer `*mp` was set to `&mess` in `init()`

→ do it in `open()`

```
static int device_open(struct inode *inode,
                       struct file  *filp)
{
  mp = mess;
  return 0;
}
```


open( )

`open()` does any initialization in preparation for later operations:

- increment the usage count (not necessary in Linux 2.4, 2.6)
- check for device-specific errors (device not ready)
- initialize the device, if it is being opened for the first time
- identify the minor number
- allocate and fill data structures (`filp->private_data`)

the reverse method is `release()`

```
static int device_release(struct inode *inode,
                          struct file  *filp);
```


Usage Count

in modern kernels the system automatically keeps a usage count (if the field `owner: THIS_MODULE` is included in the `file_operations` structure) in order to determine whether the module can be safely removed

we inspect `/proc/modules` several times while running and ending our application `app1` from several consoles:

```
rwko1      688  0
rwko1      688  1
rwko1      688  2
rwko1      688  0
```

only if the usage count is 0, the module can be unloaded

file Structure

our driver doesn’t behave like reading an ordinary file:

```
> app1                    > app2

read 10 : The goal o
                          read 10 : f this tut
read  6 : orial
                          read  0 :
read  0 :
```

the driver is opened two times, both use the same pointer **`*mp`**,
it is a **global variable** (initialized in `open()`, used in `read()`)

file Structure

the driver is opened two times, both use the same pointer `*mp`,
it is a global variable (initialized in `open()`, used in read`()`)

How can we change this without having additional parameters in `open()` and `read()` ? – We need a **Handle Context**.

we have one useful parameter: **`struct file *filp`**

the `file` structure (defined in `<linux/fs.h>`) describes
an open file or device in kernel space,
it is not to be confused with `FILE` in user space

file Structure

the `file` structure (defined in `<linux/fs.h>`) describes
an open file or device in kernel space,
it is created by the kernel on the `open` system call
and is passed to any function that operates on the file

there are only few fields in `file` that are important for us:

`struct file_operations *f_op;`

a pointer to our `fops` struct, may be changed,
for example to deal with different minor numbers

file Structure

there are only few fields that are important for us:

`loff_t f_pos;`
the current reading or writing position (64 bit integer)
do not change it directly – the last parameter of
`read()` and `write(): loff_t *offs` points to it

`void *private_data;`
for private use – either directly or as a pointer to allocated
memory (don’t forget to free the memory in `release()`)

– we can use both of them for our problem
the latter is more flexible, we will use `*offs` for `llseek()`

file Structure

```
void *private_data;
```

for private use – exactly what we need for a context:

```
static int device_open(struct inode *inode,
                       struct file  *filp)
{
  // mp = mess;
  filp->private_data = mess;
  return 0;
}
```


file Structure

```
static ssize_t device_read(struct file *filp,
      char *buffer, size_t len, loff_t *offs)
{
  unsigned int  i;  char *mp;
  mp = filp->private_data;
  for(i=0; i<10; i++){
    if(i==len) break;
    if(*mp==0) break;
    put_user(*mp++, buffer++);
  }
  filp->private_data = mp;
  return i;
}
```


file Structure

now the driver behaves like reading an ordinary file:

```
> app1                     > app2
read 10 : The goal o
                           read 10 : The goal o
                           read 10 : f this tut
read 10 : f this tut
read  6 : orial
                           read  6 : orial
                           read  0 :
read  0 :
```

both applications are independent of each other, `filp->private_data` is specific to every instance of an open file/driver

file Structure

there are few more fields in the `file` structure that can be important for us:

`struct dentry f_dentry;`

`*inode` isn't an argument to `read()` and `write()` anymore (it's hardly ever used in drivers), but we can access it by: `filp->f_dentry->d_inode`

`mode_t f_mode;`

`FMODE_READ, FMODE_WRITE`, the read/write file permissions you don't need to check them for `read()` or `write()` (the kernel checks it) but maybe for `ioctl()`

`unsigned int f_flags;`

file flags, such as `O_NONBLOCK, FASYNC` (v. blocking I/O)

llseek( )

the `file_operations` structure contains a field:

```
loff_t (*llseek) (struct file *filp,
       loff_t offs, int whence);
```

it is the counterpart of `lseek()`, it is used to change the current read/write position in a file and returns that new position

although this function makes no sense in most char hardware devices, there is a default behavior when the method is not provided by the driver, using the `f_pos` field of the `file` structure – we can seek from the beginning or from the current position, but not from the end

llseek( )

`llseek()` uses the `f_pos` field of the `file` structure, we are not to change it directly – the last parameter of `read()` and `write()`: `loff_t *offs` points to it

we have to do only few changes in our `read()` method:

```
// mp = filp->private_data;
if (*offs > strlen(mess)) return -EFAULT;  //Bad addr
 mp = mess + *offs;

 for(i=0; i<10; i++){ ... put_user(*mp++, buffer++); }

 //filp->private_data = mp;
*offs += i;
```


llseek( )

we write an application program using

```
k = lseek(fd, n, SEEK_SET);
k = lseek(fd, n, SEEK_CUR);
```

in order to test the new driver

since there is a default method, but `llseek()` makes no sense in most hardware devices (data flow rather than a data area), sometimes it is advisable to write an own dummy method, simply returning `-ESPIPE` (it translates to "Illegal seek")

write( )

similar to `read()` there is a `write()` function:

```
ssize_t device_write(struct file *filp,
    const char *buffer, size_t len, loff_t *offs);
```

with the same parameters and return values, except for the `const` attribute of `*buffer`

the user must have write access to the device file:

```
# chmod a+w /dev/mydev
```

write( )

similar to `read()` there is a `write()` function –

data transfer from user space to kernel space is performed with a macro:

```
get_user(local, ptr);
```

or a function:

```
ulong copy_from_user(void *to,void *from,ulong bytes);
```

again there are faster macros and functions that don't do a check:

```
__get_user(item, ptr);
ulong __copy_from_user(void *to, void *from,
                                    ulong bytes);
```


write( )

in order to demonstrate `read()` and `write()` together we write a char device driver that handles a **circular buffer**

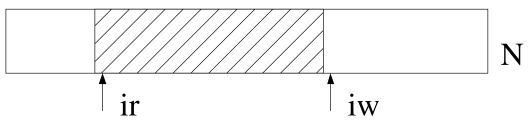
two different versions:

- buffer and pointers to **head** and **tail** as global variables, two applications may write to and read from the same buffer – they can communicate through this "pipe", we will need this version to demonstrate blocking `read()`
- buffer and pointers to head and tail in allocated memory (using `kmalloc()` and `kfree()`) pointed to by `filp->private_data`; two applications may independently write to and read from their own buffer

write( )

a char device driver that handles a **circular buffer**

- buffer and pointers to **head** and **tail** as global variables, two applications may write to and read from the same buffer – they can communicate through this "pipe",



ir = iw - buffer empty
iw = ir -1 (mod N) - buffer full

write( )

(ir = iw - buffer empty, iw = ir -1 (mod N) - buffer full)

```
static int use_count, ir, iw;

static int device_open(
        struct inode *inode, struct file *filp)
{
  if( use_count == 0 )  {  ir=iw=0; }
  use_count++;

  return 0;
}
```

Race Conditions

```
if( use_count == 0 )
{
  ir = iw = 0;              // Initialization
}
use_count++;
```

Can something go wrong, when the driver is preemptive?

It seems not (in this case of simple Initialization) –
but `use_count++` may be not atomic
– depending on the processor and the compiler

Race Conditions

`use_count++` may be not atomic:

```
reg := use_count

inc reg

use_count := reg
```

(depending on the processor and the compiler)

Race Conditions

`use_count++` may be not atomic:

```
                                        use_count reg
reg := use_count                            5    5
                  <-- use_count++
                  in another process        6    5
inc reg                                     6    6

use_count := reg                            6    6
```

now `use_count` is 6, should be 7 – **Race Condition**

Race Conditions

`use_count++` may be not atomic

write–access to **shared variables** should be **atomic**
(otherwise a read can get an inconsistent view and a
write can be overwritten or damage the consistent state)

- single variables → **atomic variables** or **atomic bit operations**
- complicated variables (structures, linked lists)
  → set up a critical section

short code, that doesn't block: **Spinlocks** (prevent preemption)
otherwise: **Semaphores** (waiting threads sleep)

Race Conditions

```
#define  BU   0x3F8      // serial Port, Buffer
#define  LCR  BU+3       // Line Control Register

if( use_count == 0 )     // Initial. serial I/O
 {
   outb(0x80, LCR);      // Baudrate follows
   outb(  12, BU );      // 12 – 9600 Baud
   outb(   0, BU+1);     //  0 – high part

   outb(0x13, LCR);      // 13 – no parity, 8 bit
   outb(   0, BU+1);     // no interrupts
 }
 use_count++;
```


Race Conditions

```
static spinlock_t lock = SPIN_LOCK_UNLOCKED;

...
spin_lock(&lock);
 if( use_count == 0 ) {
   outb(0x80, LCR);      // Baudrate follows
   outb(  12, BU );      // 12 – 9600 Baud
   outb(   0, BU+1);     //  0 – high part
   outb(0x13, LCR);      // 13 – no parity, 8 bit
   outb(   0, BU+1);     // no interrupts
 }
 use_count++;
spin_unlock(&lock);
```


Race Conditions

in `read()` we use `put_user()`, in `write()` we use `get_user()` – they may block
→ we cannot use spinlocks

we use **semaphores** (waiting threads sleep)

```
static struct semaphore sema;
sema_init(&sema,1);           // in init()

   if (down_interruptible(&sema))
             return -ERESTARTSYS;
     // critical section
   up(&sema);
```


Race Conditions

use of **semaphores:**

```
ssize_t device_read(*filp, *buffer, len, ...)
{
  int i=0;
  if (down_interruptible(&sema))
             return -ERESTARTSYS;
    while(i<len){
       if(ir==iw) break;
       put_user(rbuf[ir++], buffer++);
       i++; if(ir==N) ir=0;
     }
  up(&sema);
  return i;
}
```