We basically do two simple things with devices, read and write. To implement a character device, we need to open the device and then close it when we re done. Linux treats devices as file systems. We do create a file, read/write in it then close the file in C/C++. We have to do the same here for devices.

***NOTE**: The texts below are copied from an unknown source**,** if anyone know it's sources, let me know - I'll acknowledge them.*

***This blog is for personal study purpose only and do not have any affiliation with any group or company.***

**Opening a char device**

• A character device driver is first accessed by executing its open() function.
– This function is executed in the sys_open() system call.
– The open() function will just not be called if it is not implemented by a specific device driver. This behavior may be observed in dentry_open().

•In the case that a character needs to implement its open() function, it will have to provide the following functionality:
– Increment the usage count, so that the driver may not be removed from the Kernel (in the case of a module).
– Check for hardware-specific problems associated with this particular device.
– Initialize the hardware, if it is needed.
– Identify the minor number of the device that was open and update the f_op pointer if necessary. This is needed for device sharing the same major number but having different device drivers (i.e., miscellaneous devices).
– Allocate the memory needed for the various data structures used in the device driver and initialize these structures.

**Reading characters from device**

• read() is called to read data from the device. The form of the read function is as follows:
static ssize_t device_read(struct file * file,
char * buffer, size_t count, loff_t *ppos)
• Note that the arguments of the read() method have changed in the 2.2 and subsequent Kernels.
– The new method passes only a file structure, from which we can find the disk inode associated with the device file.
• The read() method implemented by a device driver should copy the specified number of bytes into the buffer and return the actual number of bytes read (or an error code).
– The read() function may therefore read less data than was requested. In this case the returned value will be less than size passed in parameter.

– A negative return value means that there was an error.
– Note that the buffer field passed to the device drivers refers to the memory space of the User space process that invoked the read() system call.
– The copy_to_user() function must thus be used in order to return the data in the proper memory segment.
• The following is an example of the structure used in a simple read() method. It was taken from the PC Watchdog driver.

```
#define TEMP_MINOR 131
static ssize_t pcwd_read(struct file *file, char *buf, size_t count,
loff_t *ppos)
{
unsigned short c;
unsigned char cp;
/* Can't seek on this device, so return an error */
/* if the offset is not the same as current reading position */
if (ppos != &file->f_pos)
return -ESPIPE;
/* Find the minor number associated with the device file*/
switch(MINOR(file->f_dentry->d_inode->i_rdev))
{
/* Go ahead if this is the device we want to handle */
case TEMP_MINOR:
/*
* Convert metric to Fahrenheit, since this was
* the decided 'standard' for the return value.
*/
c = inb(current_readport);
cp = (c * 9 / 5) + 32;
/* Copy the result in User space */
if(copy_to_user(buf, &cp, 1))
return -EFAULT;
/* Only one byte can be read, thus return 1 */
return 1;
/* Return error if wrong minor number */
default:
return -EINVAL;
}
}
```

**Writing to a character device**

• The write() method implemented by a driver is invoked to write data to the device. It is defined as follows:
static ssize_t device_write(struct file *file,
const char *buf, size_t count, loff_t *ppos)
• write() should copy the specified number of bytes from the User space buffer into the device.
– The number of bytes specified by the value of count should be written to the device.
– Similarly to the read() method, the number returned by the write() function should match the value of count. If it's not the case, the data was partially written or, in the case of a negative value, an error occurred.
• The following is an example of the implementation of a write() function. It does not address hardware issues, which we will reserve for the next chapters.

```
static ssize_t pcwd_write(struct file *file, const char *buf, size_t
len, loff_t *ppos)
{
/* We can't seek on this device */
if (ppos != &file->f_pos)
return -ESPIPE;
if (len)
{
pcwd_send_heartbeat();
return 1;
}
return 0;
}
```

**Closing a char device**

• A character device driver is closed when a User space application no longer needs it.
– This function is executed in the sys_close() system call.
– The release() function will not be called if it is not implemented by a specific device driver. This behavior may be observed in fput().
• The release() function is in charge of the following steps:
– It decrements the usage count. This is necessary in order for the Kernel to be able to remove the module.
– Removes any unnecessary data from memory. This is particularly true for data placed in the private_data field of the file structure associated with the device.
– Shut down the physical device if needed. This includes any operation that must be executed in order to leave the hardware in a sane state, and disabling interrupts.
• The release() function associated with a particular driver will not be invoked if the open() function was not

called.
– Again, this may be observed in the fput() function.

Posted by gizmo at 12:01 AM 3 comments Links to this post

Monday, July 30, 2007

## Linux Device Driver Development Kit (DDK)

I was looking for device driver samples, but I couldn't find any suitable sample on which I can experiment on. Moreover I always knew Linux doesn't have any Device Driver Development Kit (DDK). But the wait is over.

There is a DDK, in fact it is available since May 2006. Here is the announcement of the author.
And here is the DOWNLOAD link.

According to the author,
It is a cd image that contains everything that a Linux device driver author would need in order to create Linux drivers, including a full copy of the O'Reilly book, "Linux Device Drivers, third edition" and pre-built copies of all of the in-kernel docbook documentation for easy browsing. It even has a copy of the Linux source code that you can directly build external kernel modules against.

Posted by gizmo at 11:50 PM 0 comments Links to this post

## L A N A N A and LINUX ALLOCATED DEVICES

The Major and Minor numbers are fixed and standard numbers. It is standardized and maintained by The **Linux Assigned Names And Numbers Authority** (LANANA) and it is a part of Free Standards Group. Regardless which distro it is, these major and minor numbers remain same. New devices can be registered through LANANA.

It is important to know the numbers, it is available in linux documentation. `(Documentation/devices.txt)`

You may not have this doc installed, so here goes an online link of the latest doc.

Current Linux 2.6+ Device List (Updated 24 July 2007)

Posted by gizmo at 11:30 PM 0 comments Links to this post

Saturday, July 28, 2007

## Planning to write a device driver

Planning for writing a device driver on this week. My primary target is to write a driver which would be able to talk to any USB device. The conversations can be very simple. The problem is all devices are recognizable and automatically mounted by linux (like plug n play). I guess I have to remove the mounted device from kernel but not physically, then would be able to use my driver.

These are simply assumptions. Now I am going through few online materials. Here are some useful links.

http://kernelnewbies.org/
http://kernelnewbies.org/Drivers <- this page is like heaven for me!
http://www.linuxplanet.com/linuxplanet/tutorials/1019/
http://www.linuxdevices.com/articles/AT5340618290.html
http://developer.osdl.org/dev/opendrivers/
http://www.tldp.org/HOWTO/IO-Port-Programming.html
http://www2.linuxjournal.com/article/2920

Well there's a totally dedicated site for USB devices
http://www.linux-usb.org/

Dedicated serial driver how-to's
http://www.easysw.com/~mike/serial/serial.html

I am still choosing which device to use. Cell phones can be a good choice. Cell phone manufacturers usually provide windows drivers and applications, but linux users are totally ignored. I may try to write a driver for my cell phone. Which is Motorola branded.
Fortunately Motorola has a developer network, that might be useful
http://developer.motorola.com/

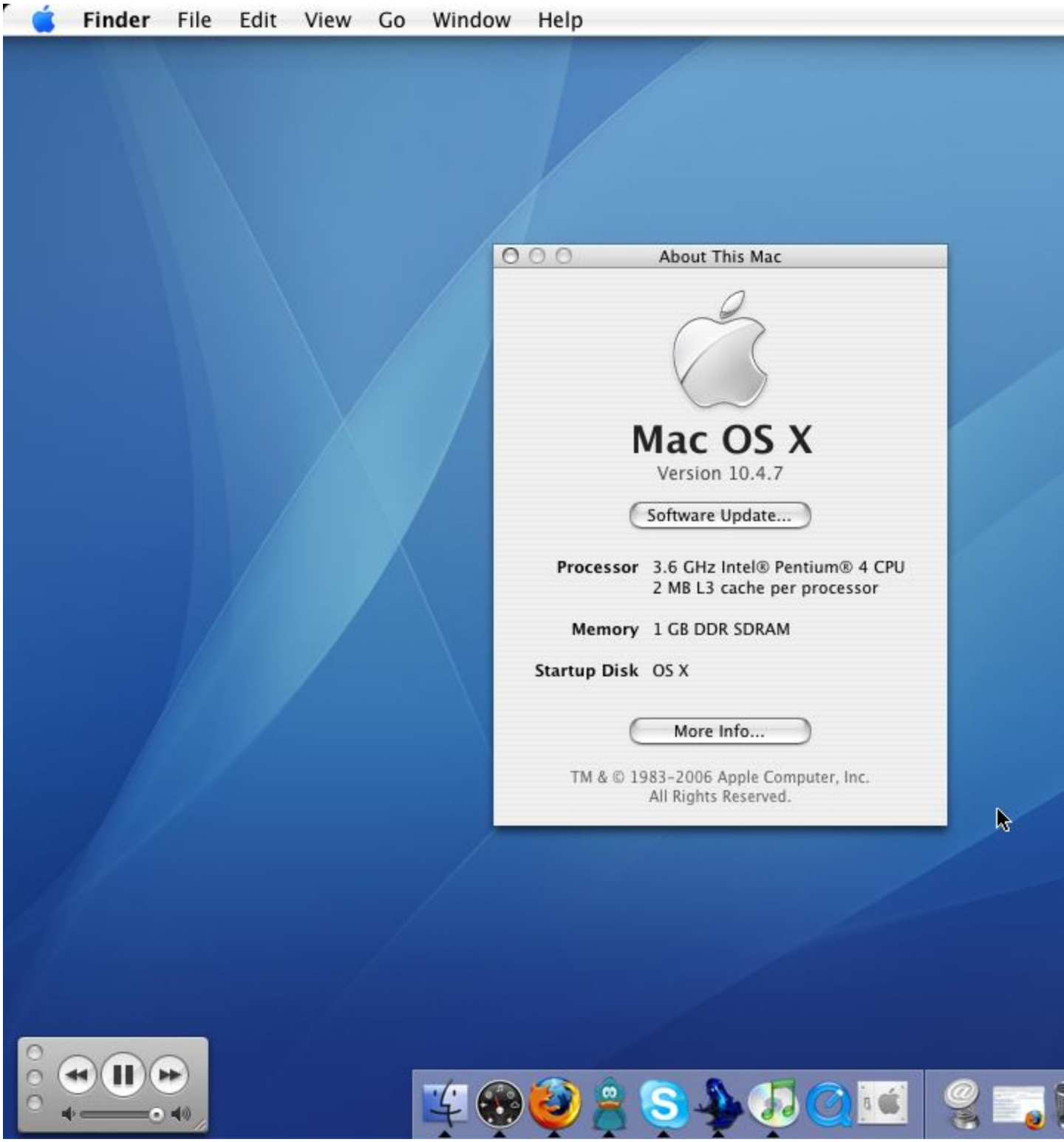
Posted by gizmo at 5:53 AM 0 comments Links to this post 

## Mac OSX on x86 PC!

Well this is not a new trend at all. Some computer geeks are doing this since the mac osx was released. But the release of Mac Intel introduced a new era of PC Mac'ing. I myself always wanted to use Mac OS since I was a kid. My first computer text book was Mac based. Though I never had money for buying a mac. Well, I do really a lot of experiments on loads of softwares and different OS's if I had a mac, that wasn't possible.

The first MacIntel's configuration used Intel D915GAV mainboard with Intel GMA900 built on PCI-E graphics and Realtek HD sound chipsets, 2.8GHz P4 HT processor, 80GB SATA HDD, 512MB DDR RAM. Coincidently my PC configuration matched exactly same except HDD, I have 160GB SATA HDD.

I always had the plan to use mac os on my pc, finally the dream came true.
Check out these screenshots (click the thumbnails to enlarge)


Finder File Edit View Go Window Help
About This Mac
Mac OS X
Version 10.4.7
Software Update...
Processor 3.6 GHz Intel® Pentium® 4 CPU
2 MB L3 cache per processor
Memory 1 GB DDR SDRAM
Startup Disk OS X
More Info...
TM & © 1983–2006 Apple Computer, Inc.
All Rights Reserved.

Courtesy: www.**osx86**project.org/

**This screenshot was not from any Mac or Virtual PC's**

**Disclaimer: Using Mac OSX or any other Mac product on non-Mac hardware is totally illegal.**
I do not encourage anyone to do this. Buy a mac.

MacOSX (or any other brand computer's OS) checks for Trusted Platform Module. If somehow we can remove the checking information from kernel, then it would pass the checking session.

**Here's a link where I learned (and already forgot) the routines of Mac OS X startup and driver loading**http://www.kernelthread.com/mac/osx/arch_startup.html

**My question is, how mac recognizes all the devices and loads the driver? If anyone knows, please post on comments.**

I managed a MacBook Pro's MacOSX DVD. Searched internet for information. In the end an osx86project guy (nick: Ramm) confirmed me that, I have to use a prepatched osx. If I want to modify the kernel, my life will be ruined (well I didn't mention that I study CSE).

**Summary: Mission failed :(**
**I need prepatched version of osx. The current prepatched version is 10.4.8 + upgrades to 10.4.9**

Well, in the end, I finally used Mac OS 7.0.1 on my pc last night, on Windows Vista, emulated by vMac. It is only 6MB. Here's a tutorial of how to do it. If you want a pre-compiled Mac OS 7.0.1, e-mail me.

Posted by gizmo at 4:52 AM 0 comments Links to this post

## Windows 2003 Server as Workstation

I do a lot of experiments whenever I get time. Sometimes I skip important jobs just to satisfy my mind with my experiments. Well, that can't be good but at least I learn some new things and can sleep at night (actually morning).

This might seem totally offtopic, well it is. But it is related to device drivers and Operating System so interested people, go on.

On last weekend, I took a project of using Windows 2003 Sever as workstation. Windows 2003 is not made for workstations, so it seemed an interested topic for me.

There are several reasons I was interested to use win 2003 on my PC, because

1. The kernel is new, SP1 kernel was of 2005 and SP2's kernel was of November 2006, comparing to XP, this is supposed be a huge difference.
2. More stable than XP or any other windows (except vista) as Microsoft says. (Google can't trace this blog, yay!)
3. Simpler than every way of XP!
4. I was bored with XP, I am using this since 2002.
5. I liked the boot screen of 2003 server :D

I have downloaded CD version of Windows 2003 Server R2 SP2 (legit 60 day evaluation copy) from microsoft's web. Google it, its available. Then Downloaded SP2 from same site.

Installation was exactly like XP. After installation, I had to do some tweaking, there are few tutorials to do the tweaks, but my target of this self project was to tweak it by myself. Which turned out to be the same as the online tutorials. In the end, my 2003 server became a totally usable as workstation. Which supports all multimedia extensions with DirectX 9.0c.

All the drivers worked nicely. Even though they warned me there are issues.

Windows Vista (RTM 6.0.6000) does not support my
USB Bluetooth device
Avermedia GO 007 FM (TV tuner)

Windows 2003 Server supports all the devices I have. Without Any issue.
And supports all the softwares and almost all games that runs on XP as well (I don't even care for giving a list).

On this weekend, my OS totally got ruined, as I forgot to install any antivirus, so I had to install other OS over this nice installation of win 2003. :(

THE END

Posted by gizmo at 4:11 AM 2 comments Links to this post

Monday, July 16, 2007

## Major and Minor numbers

One of the basic features of the Linux kernel is that it abstracts the handling of devices. All hardware devices look like regular files. They can be opened, closed, read and written using the same, standart system calls that are used to manipulate files.

Every device in the system is represented by a file. For block (disk) and character devices, these device files are created by the `mknod` command and they describe the device using major and minor device numbers.

The kernel needs to be told how to access the device. Not only does the kernel need to be told what kind of device is being accessed but also any special information, such as the partition number if it's a hard disk or density if it's a floppy, for example. This is accomplished by the major number and minor number of that device.

All devices controlled by the same device driver have a common major device number. The minor device numbers are used to distinguish between different devices and their controllers. Linux maps the device special file passed in system calls (say to mount a file system on a block device) to the device's device driver using the major device number and a number of system tables, for example the character device table, chrdevs. The major number is actually the offset into the kernel's device driver table, which tells the kernel what kind of device it is (whether it is a hard disk or a serial terminal). The minor number tells the kernel special characteristics of the device to be accessed. For example, the second hard disk has a different minor number than the first. The COM1 port has a different minor number than the COM2 port, each partition on the primary IDE disk has a different minor device number, and so forth. So, for example, /dev/hda2, the second partition of the primary IDE disk has a major number of 3 and a minor number of 2.

Reference: The Linux tutorial

Posted by gizmo at 2:30 AM 2 comments Links to this post

## Character Drivers and Block Drivers

I am stuck here for over a week. Actually I cannot decide what to do. The materials I have found so far, everybody seems to be concerned to dive into programming stuff rather than describing what is it. Well, I am trying to define some definition stuff without getting into much programming.

**Character Devices and character drivers:**

Character devices can be accessed as a stream of bytes, and this job is done by character drivers. Which means, the driver can access the device and transfer data as characters. The benefit of this method is, if the device is a sequential access device - character by character data transfer method is the most effective one. And we can use this method almost everything. Even if the device is a random-access device, we can emulate/pretend that the device is a sequential access device.

The serial port is the most familiar and widely used character device (/dev/ttys0). Character devices are represented by file nodes, and all the nodes are mounted on /dev directory. Though this device is treated as a file system, we cannot manipulate this file as conventional file (naturally).

**Block devices and block drivers**

Block devices are accessed by filesystem nodes just like a char device does (in /dev directory). It is treated like char devices almost in all ways, except one. That is, it trasnfers data by blocks (1KB per block, usually). And that gave the block devices the power of hosting the filesystems itself (disks, storage). Block devices are usually the disks and other storage devices, and virtual filesystems. Block devices are managed by kernel itself.

Here is a little note on linux file systems. click here

**A little comparison**

So in Linux, devices are treated as files. The benefit is, a common structure can be used for talking to all the devices. The idea nice. Then why do we have block drivers and character drivers. We could have done everything by character drivers. But the problem occurs when the random-access devices comes in. For example, if we use large storage devices, using character drivers won't be efficient, we need block drivers in this case. Again, for a keyboard, using a block driver is useless. So, we need these two distinct drivers.

Posted by gizmo at 12:23 AM 0 comments Links to this post 

Tuesday, July 10, 2007

## Checklist July 10, 2007

Topics I need to cover next in details are:

1. Character Devices and Block Devices
2. Major Number and Minor Numbers
3. Routines

Posted by gizmo at 4:04 AM 0 comments Links to this post 

# Linux Device Driver Architecture Components

I have discussed about the module structures before. Now lets have a look on overall linux device driver architecture components. So that we can sort out, what should be the next step of my research work.

**Device Drivers as modules**

Linux device drivers are managed by kernel itself. Modules works here as device drivers. Modules can interact with kernels directly. When we write device drivers, we use the module structure to specify custom routines. *(I have posted the basic module structure before, you can take a look on that)*.

To load the driver we use module_init and to unload we use module_exit. When the module_init is executed, the first routine is executed as well. *We have seen this example when we wrote the elementary module on previous post*. The kernel registers the driver by using a registration routine register_chrdev while loading the module and when unloading the module it uses unregister_chrdev. By the way, these are for character devices. I still don't know what actually these does, I'll find it out later.

**Naming Devices**

In linux, devices are labeled by numbers (0-255). These 256 numbers are known as major numbers. A major number is assigned to a particular device. So, Linux can handle 256 devices all together. Only 256 devices? No, they are major devices. Each major device can handle additional 256 devices of that type. Lets recalculate the number 256x256 = 65535. Well, now nobody can call this a small number. kernels documentation/devices.txt file is supposed to have the list of all major devices. I'll upload that file once I boot into ubuntu (Now I'm on Vista). By the way, the 0-th device is known as null device.

Applications can not access the device by the major number directly. Rather, apps use something called file system entries. I guess I didn't mention it earlier. The beauty of linux is, it can treat all physical devices, virtual devices and real file systems as file systems. The benefit of this is we can use the generic structure for all of them, without knowing what type of device we are using, still it is not that easy as it sounds right now. All driver entry points are located on a directory "/dev". For example, /dev/tty1 is the serial port 1 [COM1] . Applications which want to use any driver uses a system call to get the handle of the device. Once it gets the handle, it can gain access of the devices and can use the device by other system calls.

**Driver Installation**

Driver Installation is varied by linux distribution. As I have mentioned earlier. Linux uses modules as device drivers. All drivers are installed as modules and are located in directory "/lib/modules/kernel_x.x.xx". A configuration file located on /etc/modules.conf is used by the kernel [and user if you want to override it] to load and unload the modules. Module installing and uninstalling can be done by kernel module utilities like`insmod`, `rmmod` [we have used it on previous examples] and `modprobe`. Installing/inserting a module isn't

everything, the module has to be registered as a device entry in the directory "/dev". The utility `mknod` does this job.

To see what drivers have been installed, we can see the contents of "/proc/module". Which will show the list of all loaded modules/devices.

Posted by gizmo at 2:31 AM 0 comments Links to this post