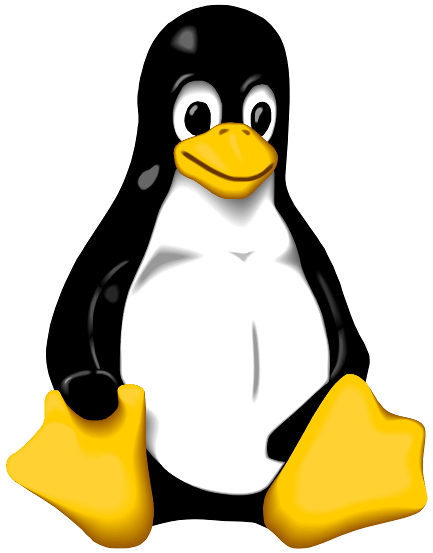
Linux Kernel Hacking Free Course

4th edition

Ing. Gianluca Grilli
System Programming Research Group
University of Rome “Tor Vergata”
gianluca.grilli@uniroma2.it

Compiling and installing the kernel



Contents:

Why recompiling the kernel?

- to take advantage of new features available in the last kernel release (scheduler, memory management, hw management)
- to take advantage of hardware not included in the stock kernel that came with the distribution you installed
- to close potential holes from modules or features that you do not even use
- to tailor the kernel specifically on your computer hardware, resulting in a performance boost

Linux kernel versioning system (1/2)

2.6.24.3

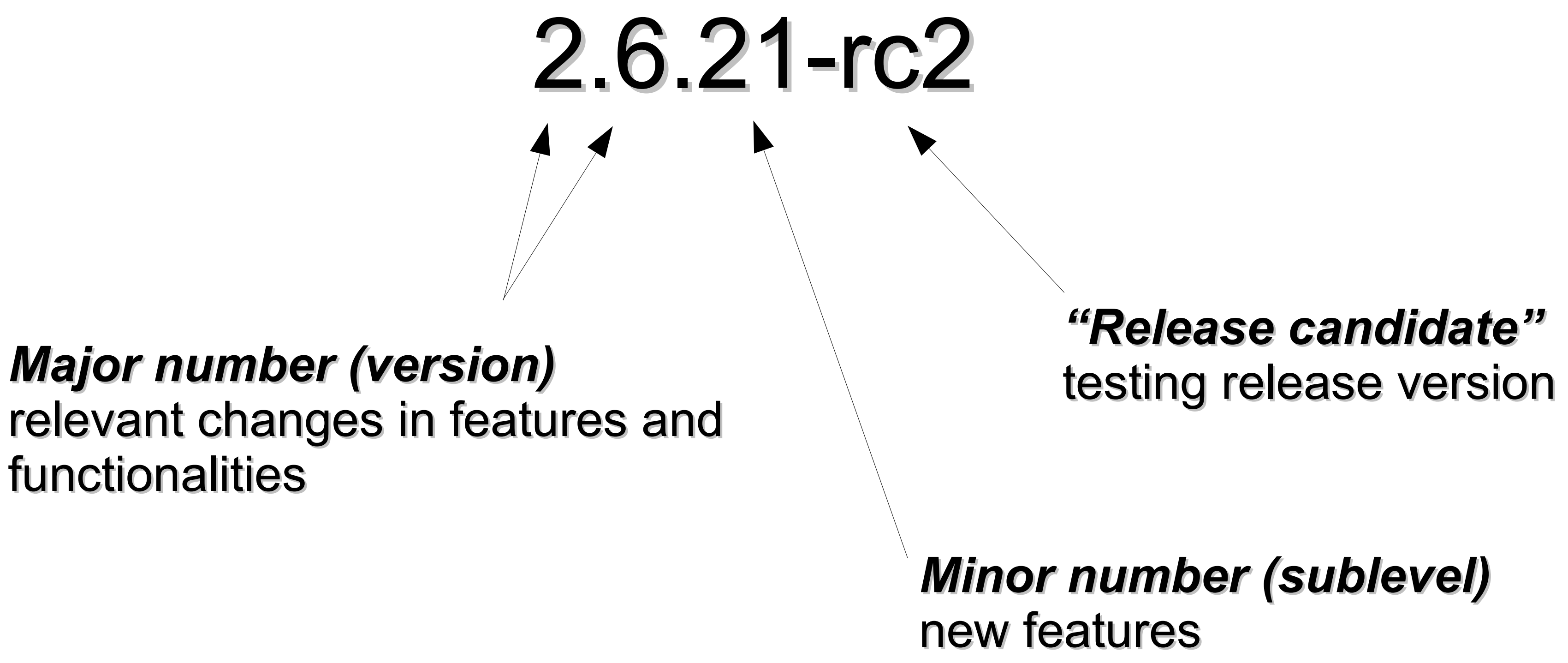
Major number (version)
relevant changes in features and functionalities

??? (bug-fix)
bug fixes that do not require a new minor number

Minor number (sublevel)
new features

Important: when you want to apply a new patch, you need necessarily a base version (i.e.: 2.6.24 and not 2.6.24.3).

Linux kernel versioning system (2/2)
(developing version, “request for comment”)



2.6.21-rc2

Major number (version)
relevant changes in features and functionalities

Minor number (sublevel)
new features

“Release candidate”
testing release version

Kernel modules

Modules are relocatable file objects (relocatable ELF) that can be loaded and unloaded into the kernel upon demand.

They extend the functionality of the kernel without the need to reboot the system. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system.

Note that kernel 2.6 introduces a new file naming convention: kernel modules now have a .ko extension (in place of the old .o extension) which easily distinguishes them from conventional object files. The reason for this is that they contain an additional .modinfo section that where additional information about the module is kept.

In order to see lot's of useful informations about a module such as bugreports, license information and even a short description of the parameters it accepts, use the following command:

```
modinfo mymodule.ko
```

Static or modular kernel?

Static

- 🟢 security: an attacker cannot install rootkits
- 🔴 the binary file can be huge
- 🔴 modifying and testing monolithic systems takes longer
- 🔴 when a bug surfaces within the core of the kernel, the effects can be far reaching

Modular

- 🟢 lower memory usage and faster boot time
- 🟢 on demand capability versus spending time recompiling a whole kernel
- 🟢 faster developing time for drivers because reboot is not required (provided the kernel is not destabilized)
- 🔴 It takes time everytime you access a module's functionality (it has to be loaded into memory)

Obtaining a new kernel

Binaries
- 🟢 easy to install
- 🔴 distribution dependent
- 🔴 not customizable

→ *RPM (Red Hat), YaST (Suse), LISA (Caldera), Synaptic (Ubuntu),...*

Sources
- 🟢 100% customizable
- 🟢 distribution independent
- 🔴 some skills required

→ *www.kernel.org*

How to prepare a new kernel version

- move to the kernel sources directory:

```
cd /usr/src
```

- explode the archive:

```
tar xvf linux-2.6.24.3.tar.bz2
```

- remove the symbolic link (if present):

```
rm -f linux
```

- create a new symbolic link (optional):

```
ln -s linux-2.6.24.3 linux
```

- enter the new kernel's root directory:

```
cd  linux
```

Now you are ready to configure your new kernel !

Configuring the Kernel (1/11)

In the configuration process we are interested in selecting the features we want to be compiled in the new kernel. Features can be compiled as modules or built in into the kernel.

Kernel configuration is not painless and some expertise and a good knowledge of the hardware is needed in order to select the correct features.

An incorrectly configured kernel can get stuck even on boot phase!

The configuration file is stored in a hidden file in the kernel's root directory: **`/usr/src/linux-2.6.24.3/.config`**

The first step in the configuration process is collecting the proper informations about the system by using the tools provided with the linux distribution and the user manuals of the connected devices.

Configuring the Kernel (2/11) – gathering informations

These simple utilities allow you to gather informations on your hardware:

`lspci -xv`	detailed informations about the PCI bus
`lsusb`	USB devices
`cat /proc/cpuinfo`	CPU
`dmesg`	“diagnostic message”, important messages printed by the kernel during boot
`cat /proc/filesystems`	filesystems currently in use

Note: lspci needs superuser privileges to access some informations!

Configuring the Kernel (3/11)

If you're running a linux kernel, it is fully functional and recognizes properly the hardware installed in the system, a good idea could be trying to use as much as possible the old config file.

In this case, use the command "**`uname -a`**" to see your current kernel version and enter the **`/boot`** directory looking for a file similar to this one (let's suppose that you are running the kernel 2.6.20 and you want to upgrade to 2.6.24 version):

```
/boot/config-2.6.20-smp
```

Otherwise, if the kernel you are running is compiled with a specific option, you can find a compressed config file in the /proc directory. In that case, you can copy it into the new kernel's root as it follows:

```
cp /proc/config.gz /usr/src/linux/config.gz && \\
mv /usr/src/linux/config.gz /usr/src/linux/.config.gz
gunzip /usr/src/linux/.config.gz
```

Configuring the Kernel (4/11)

What if you don't have a previous configuration file?

If you don't have a configuration file and you don't have a deep knowledge of your hardware, you can go through the so called *“dirty compilation” (DC).*

A DC compilation consists in selecting as built-in almost every features related to common hardware like, for example, motherboard chipsets and EIDE/SATA/ SCSI filesystems.

Once you have your new kernel up and running, you can “strip” your configuration file step by step by removing the unnecessary features and, thus, obtaining a clean, light kernel image.

Configuring the Kernel (5/11)

Hint: how to know exactly the name for a PCI device driver

Sometimes, devices from many different vendors use the same chipset. For example, many video cards are based on "*nvidia*" chipset.

So, even if two video cards can be different, they are almost the same and, probably, the driver they use is the same.
In this case, it would be really useful to know exactly the name of the driver in order to compile it within the new kernel.

The linux kernel do exactly the same thing everytime it loads automatically a driver when it recognizes a specified device installed into the system.

Let's try to understand how this mechanism works.

Configuring the Kernel (6/11)

Hint: how to know exactly the name for a PCI device driver

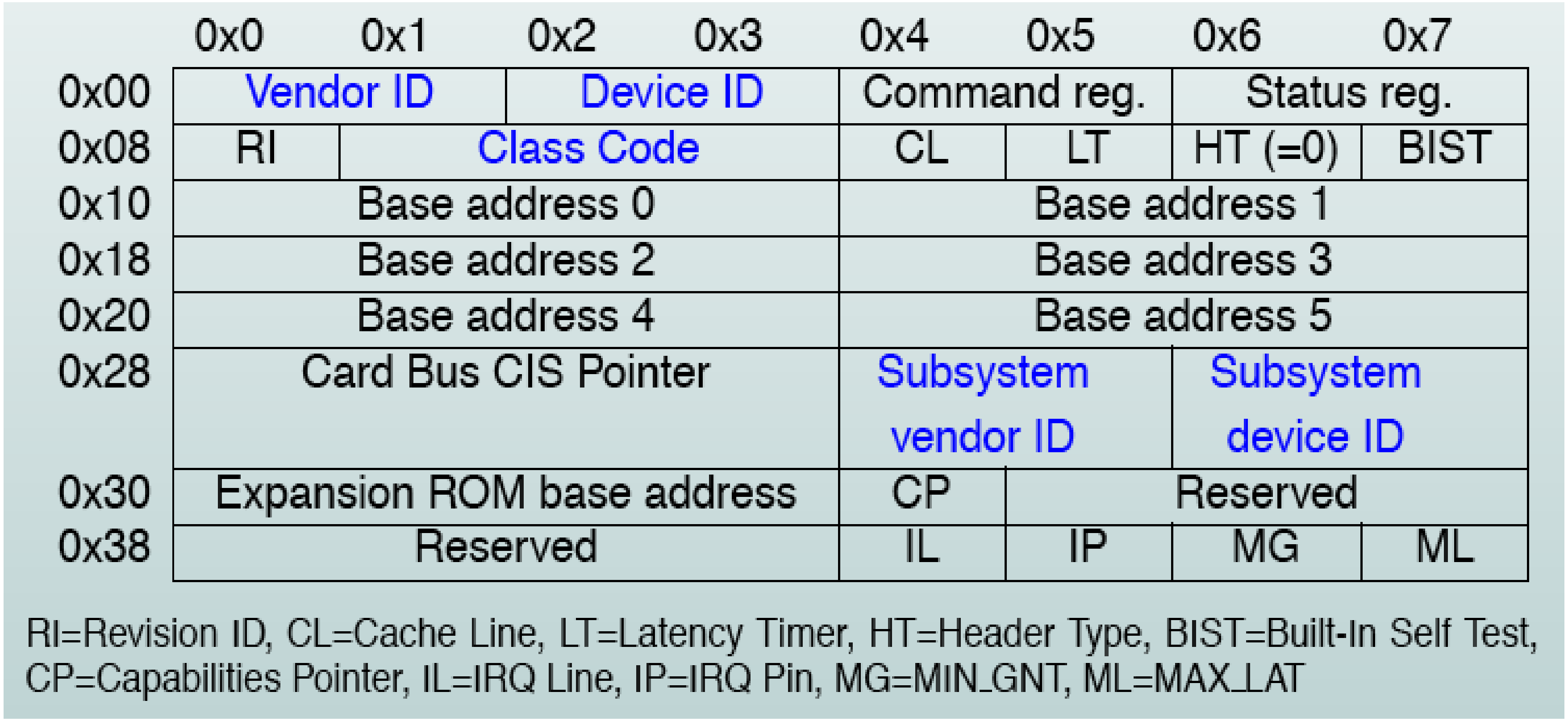
A PCI device driver has two main data structures:

`struct pci_device_id`	It is used to identify the PCI device through several parameters provided by the vendor
`struct pci_driver`	In this structure we can find functions used to handle certain events and a pointer to the pci_device_id table.

Configuring the Kernel (7/11)

Hint: how to know exactly the name for a PCI device driver

These are the informations stored in the device firmware:



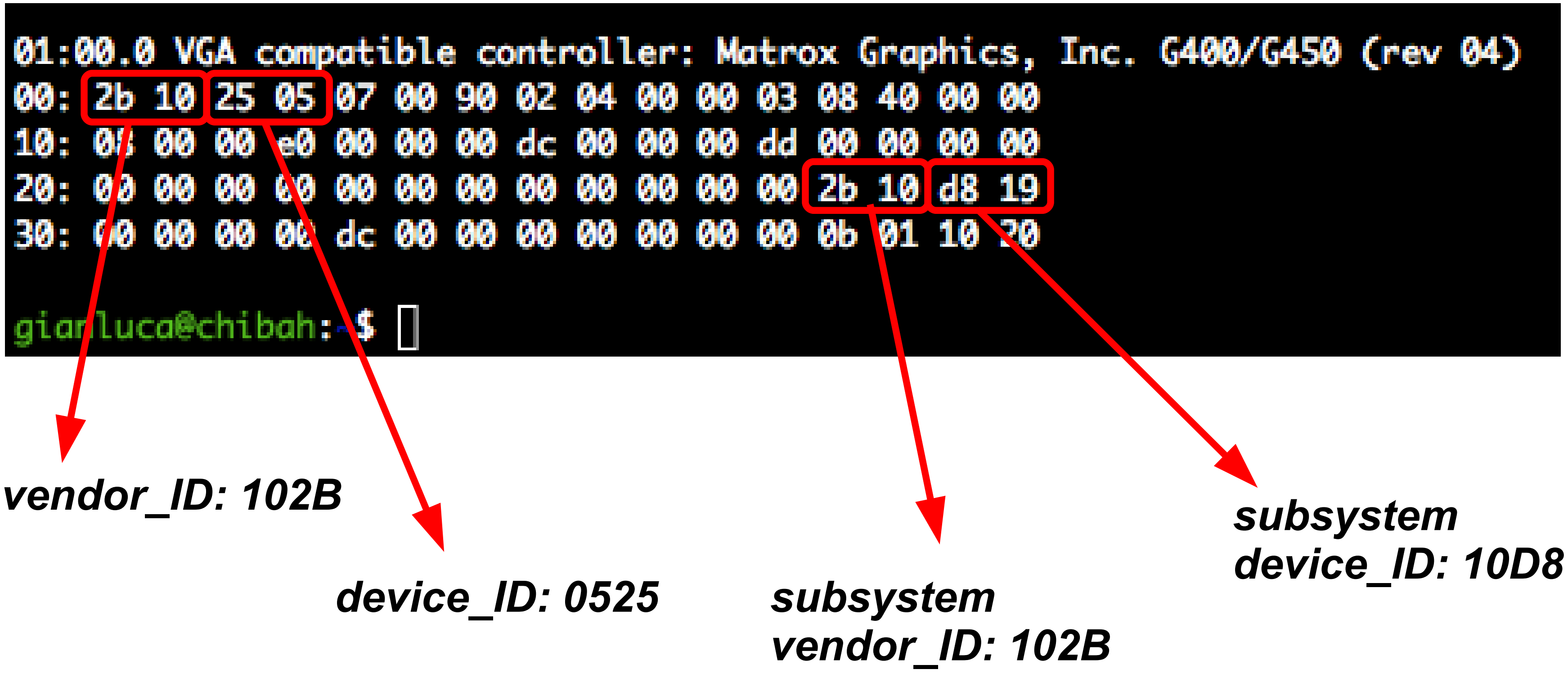
	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7
0x00	Vendor ID		Device ID		Command reg.		Status reg.	
0x08	RI	Class Code			CL	LT	HT (=0)	BIST
0x10	Base address 0				Base address 1			
0x18	Base address 2				Base address 3			
0x20	Base address 4				Base address 5			
0x28	Card Bus CIS Pointer				Subsystem vendor ID		Subsystem device ID	
0x30	Expansion ROM base address				CP	Reserved		
0x38	Reserved				IL	IP	MG	ML

RI=Revision ID, CL=Cache Line, LT=Latency Timer, HT=Header Type, BIST=Built-In Self Test, CP=Capabilities Pointer, IL=IRQ Line, IP=IRQ Pin, MG=MIN_GNT, ML=MAX_LAT

Configuring the Kernel (8/11)

Hint: how to know exactly the name for a PCI device driver

We can retrieve the same informations with the command **`lspci -xv`** :



```
01:00.0 VGA compatible controller: Matrox Graphics, Inc. G400/G450 (rev 04)
00: 2b 10 25 05 07 00 90 02 04 00 00 03 08 40 00 00
10: 08 00 00 e0 00 00 00 dc 00 00 00 dd 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 2b 10 d8 19
30: 00 00 00 00 dc 00 00 00 00 00 00 00 0b 01 10 20

gianluca@chibah:~$ 
```

Configuring the Kernel (9/11)

Hint: how to know exactly the name for a PCI device driver

The only thing you have to do is:

1) determine vendor_id and device_id tramite with the command **`lspci -xv`**

2) search recursively in the "driver" section **`/usr/src/linux/drivers`** in order to find a driver with the same parameters in the **`pci_device_id`** data structure

3) once you have found the driver, try to locate the inizialization of the **`char *name`** pointer inside the **`pci_driver`** structure

Configuring the Kernel (10/11)

Several tools can be used to configure the linux kernel:

`make config`
(terminal)

`make menuconfig`
(pseudo-graphical)

`make oldconfig`(*)
(terminal)

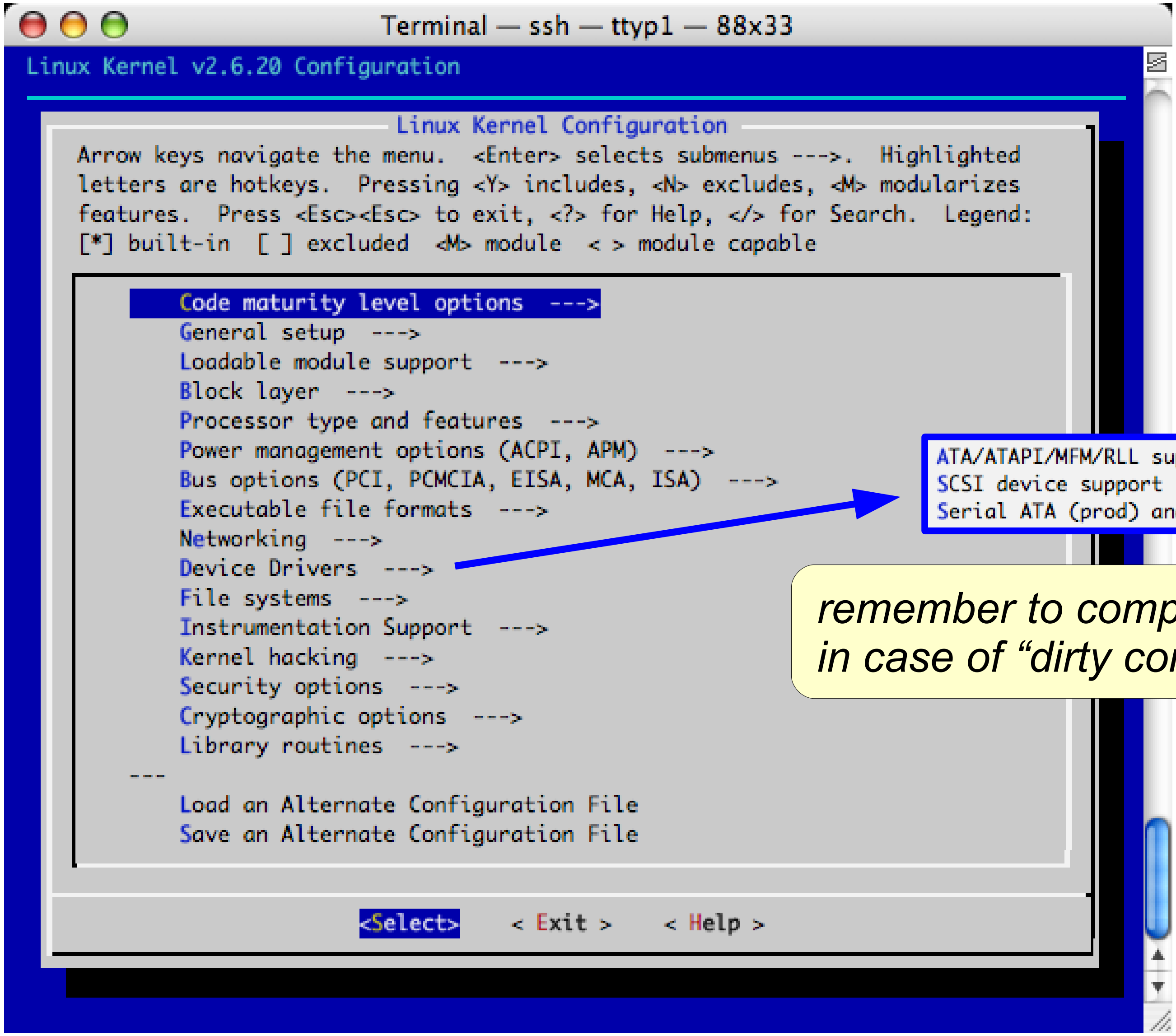
`make xconfig`
(graphical)

It is possible to choose three different options:

'Y': feature will be included statically in the kernel
'M': feature will be compiled as module
'N': feature will not be compiled at all

(*) It is really useful when you apply patches or use config files from previous kernel versions.

Configuring the Kernel (11/11) – make menuconfig example



```
Terminal — ssh — ttyp1 — 88x33
Linux Kernel v2.6.20 Configuration
                         Linux Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted
  letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes
  features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend:
  [*] built-in  [ ] excluded  <M> module  < > module capable

         Code maturity level options  --->
         General setup  --->
         Loadable module support  --->
         Block layer  --->
         Processor type and features  --->
         Power management options (ACPI, APM)  --->
         Bus options (PCI, PCMCIA, EISA, MCA, ISA)  --->
         Executable file formats  --->
         Networking  --->
         Device Drivers  --->
         File systems  --->
         Instrumentation Support  --->
         Kernel hacking  --->
         Security options  --->
         Cryptographic options  --->
         Library routines  --->
     ---
         Load an Alternate Configuration File
         Save an Alternate Configuration File

                  <Select>    < Exit >    < Help >
```

```
ATA/ATAPI/MFM/RLL support  --->
SCSI device support  --->
Serial ATA (prod) and Parallel ATA (experimental) drivers
```

remember to compile as "static" these features in case of "dirty compiling"!

Compiling the kernel

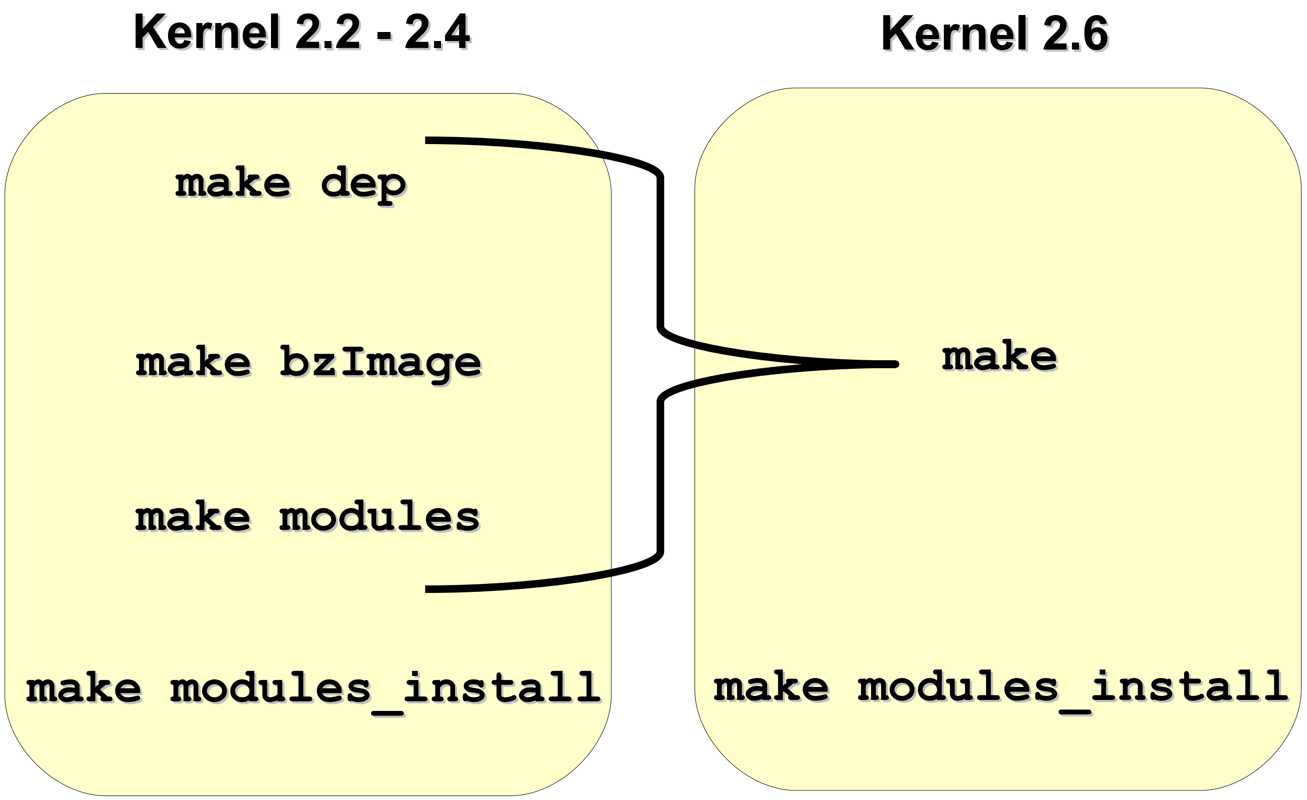
The compiling process is made up of several steps and it depends on the kernel version:

step 1 create dependencies

step 2 create a compressed kernel image

step 3 compile modules

step 4 install modules in **`/lib/modules`**



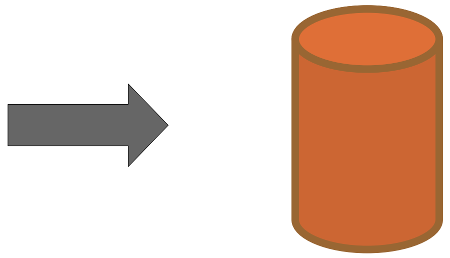
```
make dep
make bzImage
make modules
make modules_install
```

```
make
make modules_install
```

Installing the kernel image on the system (1/8)

Once the compilation process is over, we should have our modules in the **`/lib/modules/linux-2.6.X-xxx`** folder, the kernel image **`vmlinux`** in **`/usr/src/linux`** and the big compressed image bzImage in **`/usr/src/linux/arch/i386/boot/bzImage`**.

Now we must install the compressed kernel image on our system:



(It is not the safer choice, but very common)

Note: we refer to intel x86 architecture.

Installing the kernel image on the system (2/8)

Hard disk's MBR

The need of a boot loader

A boot loader loads the operating system. When your machine loads its operating system, the BIOS reads the first 512 bytes of your bootable media (which is known as the master boot record, or MBR). You can store the boot record of only one operating system in a single MBR, so a problem becomes apparent when you require multiple operating systems. Hence the need for more flexible boot loaders.

The master boot record itself holds two things -- either some of or all of the boot loader program and the partition table (which holds information regarding how the rest of the media is split up into partitions). When the BIOS loads, it looks for data stored in the first sector of the hard drive, the MBR; using the data stored in the MBR, the BIOS activates the boot loader.

Linux most popular boot loaders: LILO and GRUB.

Note: we refer to intel x86 architecture.

Installing the kernel image on the system (3/8)

LILO

LInux LOader, or LILO, used to come as standard on all distributions of Linux. As one of the older/oldest Linux boot loaders, its continued strong Linux community support has enabled it to evolve over time and stay viable as a usable modern-day boot loader. Some new functionality includes an enhanced user interface and exploitation of new BIOS functions that eliminate the old 1024-cylinder limit.

LILO configuration is all done through a configuration file located in /etc/lilo.conf. The next slide will show an example configuration for dual booting a Linux and Windows machine.

Note: we refer to intel x86 architecture.

Installing the kernel image on the system (4/8)

Example `lilo.conf` file:

```
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
prompt
timeout=100
compact
default=Linux
image=/boot/vmlinuz-2.6.24.3
    label=Linux
    root=/dev/hdb3
    read-only
    password=linux
other=/dev/hda
    label=WindowsXP
```

Note: we refer to intel x86 architecture.

`boot=` tells LILO where to install the boot loader

`map:` points to the map file used by LILO internally during bootup

`install`: is one of the files used internally by LILO during the boot process

`prompt`: tells LILO to use the user interface

`timeout`: is the number of tenths of a second that the boot prompt will wait before automatically loading the default OS, in this case Linux

`compact`: makes the boot process quicker by merging adjacent disk read requests into a single request

`default`: tells LILO which image to boot from by default

`label`: identifies the different OS you want to boot from at the user interface at runtime (avoid spaces!)

Installing the kernel image on the system (5/8)

Example `lilo.conf` file:

```
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
prompt
timeout=100
compact
default=Linux
image=/boot/vmlinuz-2.6.24.3
    label=Linux
    root=/dev/hdb3
    read-only
    password=linux
other=/dev/hda
    label=WindowsXP
```

`root=` tells LILO where the OS file system actually lives

`read-only`: tells LILO to perform the initial boot to the file system read only. Once the OS is fully booted, it is mounted read-write

`password`: user will be prompted for a password to start linux in single mode

`other`: acts like a combination of the image and root options, but for operating systems other than Linux

Note: we refer to intel x86 architecture.

Installing the kernel image on the system (6/8)

Since lilo.conf is not read at boot time, the MBR needs to be "refreshed" when this is changed. If you do not do this upon rebooting, none of your changes to lilo.conf will be reflected at startup. Like getting LILO into the MBR in the first place, you need to run:

```
/sbin/lilo -v -v
```

(very high verbosity level)

LILO boot error codes

L →	the first stage boot loader has been loaded and started, but it can't load the second stage boot loader
LI →	the first stage boot loader was able to load the second stage boot loader, but has failed to execute it
LIL →	the second stage boot loader has been started, but it can't load the descriptor table from the map file.
LIL? →	the second stage boot loader has been loaded at an incorrect address
LIL- →	the descriptor table is corrupt
LILO →	all parts of LILO have been successfully loaded

Note: we refer to intel x86 architecture.

Installing the kernel image on the system (7/8)

GRUB (GRand Unified Bootloader)

- GRUB provides a true command-based, pre-OS environment on x86 machines to allow maximum flexibility in loading operating systems with certain options or gathering information about the system.
- GRUB supports Logical Block Addressing (LBA) mode.
- GRUB's configuration file is read from the disk every time the system boots, preventing the user from having to write over the MBR every time a change the boot options is made.

Note: we refer to intel x86 architecture.

Installing the kernel image on the system (8/8)

Example `/boot/grub/grub.conf` file:

```
default=0
timeout=10
splashimage=(hd0,1)/grub/sc.xpm.gz

title Gentoo Linux (2.6.24.3)
    root (hd0,1)
    kernel /vmlinuz-2.6.24.3 ro
    initrd /initrd-2.6.24.3.img

title Windows 2000
    rootnoverify (hd0,0)
    chainloader +1
```

`default=` tells grub which image to boot from by default (grub starts counting from 0)

`timeout`: is the number of a second that the boot prompt will wait before automatically loading the default OS, in this case Linux

`splashimage`: graphical initial boot screen

`title`: identifies the different OS you want to boot from at the user interface at runtime

`root (hd0,1)`: the Linux partition is on /dev/hda2

`initrd`: tells grub where to find the initial ramdisk

`rootnoverify`: similar to root, but don't attempt to mount the partition. This is useful for when an OS is outside of the area of the disk that GRUB can read

`chainloader`: this line is necessary for Grub to go into win2k's loader. Be careful with the spacing!

Note: we refer to intel x86 architecture.

Tricks and hints to speed up the compilation process (1/4)

- Avoid as much as possible to issue the make clean command because it deletes all the object files, thus forcing the compiling process to start from the beginning.
- If you're modifying only few files, it is convenient using symbolic links instead of copying the whole kernel source tree :

```
cp -al linux-2.6.24 linux-2.6.24custom
```

Remember to remove the link before modifying each file:

```
cd /usr/src/linux-2.6.24custom
cp kernel/fork.c kernel/1
mv kernel/1 kernel/fork.c
cd ..
diff -ruN linux-2.6.24 linux-2.6.24custom > mypatch
```

Tricks and hints to speed up the compiling process (2/4)

➡ If you are compiling the kernel on a multiprocessor or multicore machine, use the parallel compiling option **`-j`** (or --**`jobs`**) as follows:

```
make -j N               (stage 1)
modules_install         (stage 2)
```

Where N represents the number of concurrent jobs. Usually, you can set the N parameter according to this simple rule:

N = #CPUs (or cores) + 1

Important:

a) If you use the **`-j`** option without arguments, the gcc compiler will not limit the number of jobs that can run simultaneously, thus leading to poor performances.

Tricks and hints to speed up the compiling process – DISTCC (3/4)

DISTCC is a program to distribute builds of C, C++, Objective C or Objective C++ code across several machines on a network. You can start your distributed compilation process in almost 30 seconds.

1) for each machine, download distcc, unpack, and do:

```
./configure && make && sudo make install
```

2) on each of the servers, run `distccd --daemon`, with `--allow` options to restrict access

3) put the names of the servers in your environment:

```
export DISTCC_HOSTS='localhost red green blue'
```

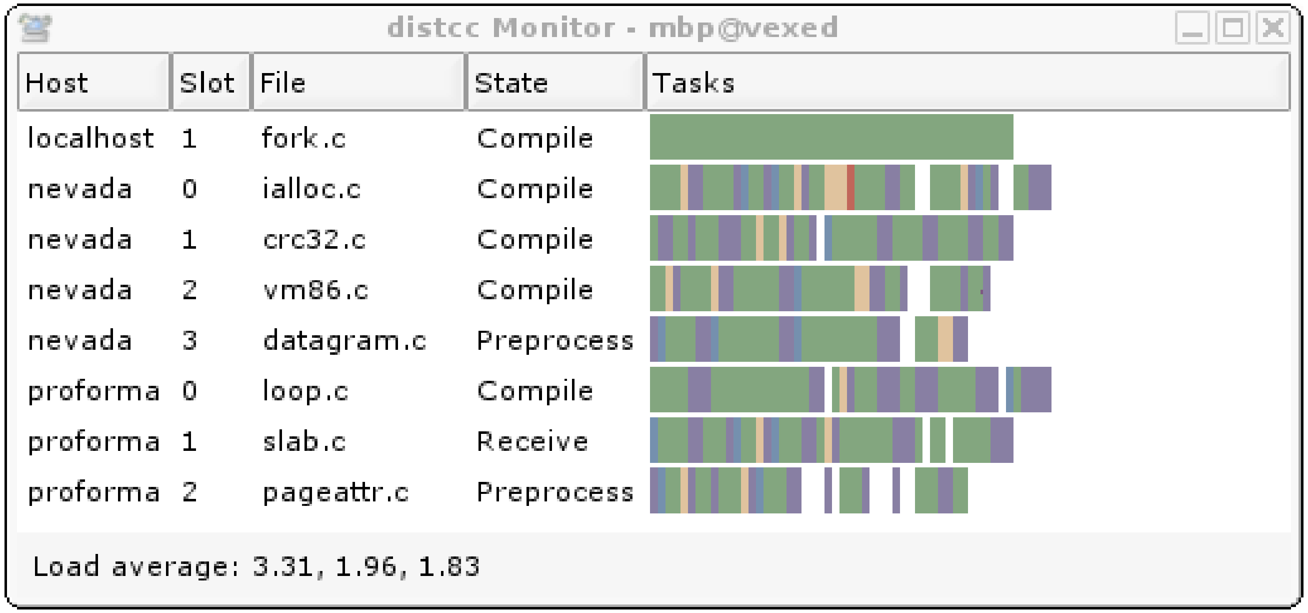
4) build your code:

```
cd ~/usr/src/linux-2.6.15; make -j8 CC=distcc
```

Tricks and hints to speed up the compiling process – DISTCC (4/4)

DISTCC is nearly linearly scalable for small numbers of machines: building Linux 2.4.19 on a single 1700MHz Pentium IV machine with distcc 0.15 takes 6 minutes, 45 seconds. Using distcc across three such machines on a 100Mbps switch takes only 2 minutes, 30 seconds: 2.6x faster. The (unreachable) theoretical maximum speedup is 3.0x, so in this case distcc scales with 89% efficiency.

DISTCC monitor screenshot



Host	Slot	File	State	Tasks
localhost	1	fork.c	Compile	
nevada	0	ialloc.c	Compile	
nevada	1	crc32.c	Compile	
nevada	2	vm86.c	Compile	
nevada	3	datagram.c	Preprocess	
proforma	0	loop.c	Compile	
proforma	1	slab.c	Receive	
proforma	2	pageattr.c	Preprocess	

Load average: 3.31, 1.96, 1.83

(source: http://distcc.samba.org/)

Initial Ramdisk image (initrd) (1/3)

Initrd is an initial root filesystem useful when we compile as modules features that kernel needs at boot time (i.e.: ext3, xfs, jfs, USB, RAID)

In fact, to access at boot time an ext3 root filesystem, the kernel needs the ext3 module that is stored in the root filesystem itself!

The solution would be loading the needed drivers in ram by using the initrd.

At boot, the kernel mounts the initrd as part of the two-stage boot process to load the modules to make the real file systems available and get at the real root file system.
Its lifetime is short, only serving as a bridge to the real root file system

Anyway, you can avoid using an initrd image if the features you need at boot time are statically compiled into the kernel.

Initial Ramdisk image (initrd) (2/3)

This is how initrd works:

1) the boot loader loads the kernel and the initial RAM disk

2) the kernel mount the initrd file as a simple ramdisk and free the previously allocated space

3) initrd is mounted as root fs in read-only mode

4) `/linuxrc` is executed (it can be any executable file launched with uid 0 and it can do, basically, whatever init does)

5) linuxrc mounts the "real" root filesystem

6) linuxrc moves the root filesystem to the root directory by using the **`pivot_root()`** system call

7) the boot sequence is completed executing the runlevel services (for example, by launching `/sbin/init`) in the root filesystem

8) initrd is unmounted.

Initial Ramdisk image (initrd) (3/3)

There are several ways to create an initrd image because the tools used to automate this process are distribution dependent.

Suse Linux:

```
cd /boot
mkinitrd -k vmlinuz-<kernel> -i initrd-<kernel>
```

Red Hat Linux:

```
cd /boot
mkinitrd -v initrd-<kernel>.img <kernel>
```

Slackware Linux:

```
cd /boot
mkinitrd -c -k 2.6.7 -m jbd:ext3 -f ext3 -r /dev/hdb3
mkinitrd -c -k 2.6.7 -m reiserfs
```

(for ext3 fs)
(for reiser fs)

Kernel upgrade from 2.6.17 to 2.6.20 version in Ubuntu linux generating a .deb package (1/2)

Ubuntu Linux (Debian based distribution) has an excellent tool used to compile the kernel: ***make-kpkg.***
This utiliy generates “.deb” packages that you can easily install with dpkg.

Packages allow you to compile the kernel on machine that is different from the target. Once you have done, you can install .deb packages (they include also kernel modules) and they will even modify your bootloader.

In the next example, we will perform an upgrade from version 2.6.22.14-generic to version 2.6.24.3 downloaded from *www.kernel.org*.

Kernel upgrade from 2.6.17 to 2.6.20 version in Ubuntu linux generating a .deb package (2/2)

After you have downloaded and exploded your new kernel in **`/usr/src`**, created the symbolic link **`/usr/src/linux -> /usr/src/linux-2.6.24.3`** and properly configured it, you can do the followings:

1) export $CONCURRENCY_LEVEL, equivalent to "-j N" option, with "N = #cpu/core +1" :

```
export CONCURRENCY_LEVEL=N
```

2) launch the kernel compiling in order to generate a ".deb" package:

```
make-kpkg -initrd --revision=686smp \\
   kernel_image kernel_headers modules_image
```

3) if make-kpkg you can now install the .deb package located in /usr/src :

```
dpkg -i linux-image-2.6.24.3_686smp.deb
```

Hint: faster reboot with Kexec (1/3)

Kexec is a fastboot mechanism that allows booting a Linux kernel from the context of an already running kernel without going through system boot firmware.

Benefits

- Reboot is faster
- The BIOS has no chance to throw errors on you
- You don't have to wait until the SCSI controller has enumerated all devices.
- You don't depend on a working GRUB or LILO configuration
- You can tell the kernel to kexec on panic (see “KDUMP” for details)

Caveats

- Some hardware might not work after kexec'ing the new kernel.
- Video problems like no display or corrupted display are possible
- Someone needs to hack /sbin/reboot to optionally kexec an already loaded kernel instead of the last reboot stage
- kexec does not sync, or unmount filesystems so if you need that to happen you need to do that yourself

Hint: faster reboot with Kexec (2/3)

The Kexec system call breaks up into three pieces:

A generic part which loads the new kernel from the current address space, and very carefully places the data in the allocated pages

A generic part that interacts with the kernel and tells all of the devices to shut down. Preventing on-going DMAs, and placing the devices in a consistent state so a later kernel can reinitialize them

A machine specific part that includes the syscall number and then copies the image to it's final destination, then jumps into the image at entry

Hint: faster reboot with Kexec (3/3)

In order to use Kexec, you have to:

1. ensure that the kernel has support for the kexec system calls (the kernel which you boot does not require kexec support)

2. install the kexec-tools (avaliable as .rpm, .tar.gz, .deb...)

3. load into memory the kernel you wish to be loaded after reboot. It may look something like this:

```
kexec -l /boot/vmlinuz-2.6.24.3 \\
--initrd=/boot/initrd.img-2.6.24.3 --append="root=/dev/sda"
```

4. reboot with kexec:

```
kexec -e
```