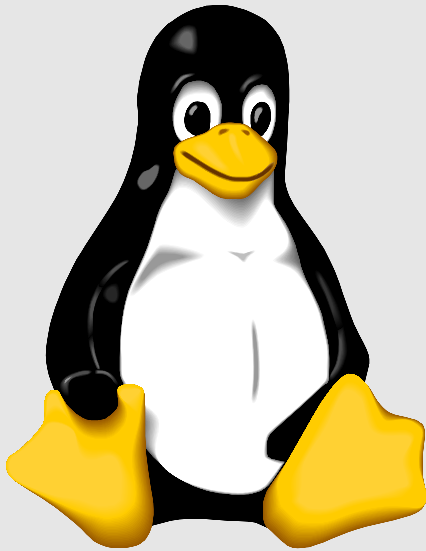
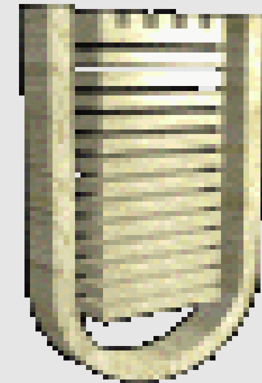
Linux Kernel Hacking Free Course

3rd edition

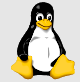
G.Grilli, University of Rome “Tor Vergata”

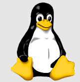
Compiling and installing the kernel

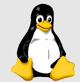




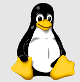
Contents:

 Why recompiling the kernel

 Obtaining a new kernel

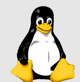
 Configuring and compiling a new kernel

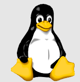
Kconfig sintax

 Installation process

LILO boot loader

GRUB boot loader

 How to apply a kernel patch

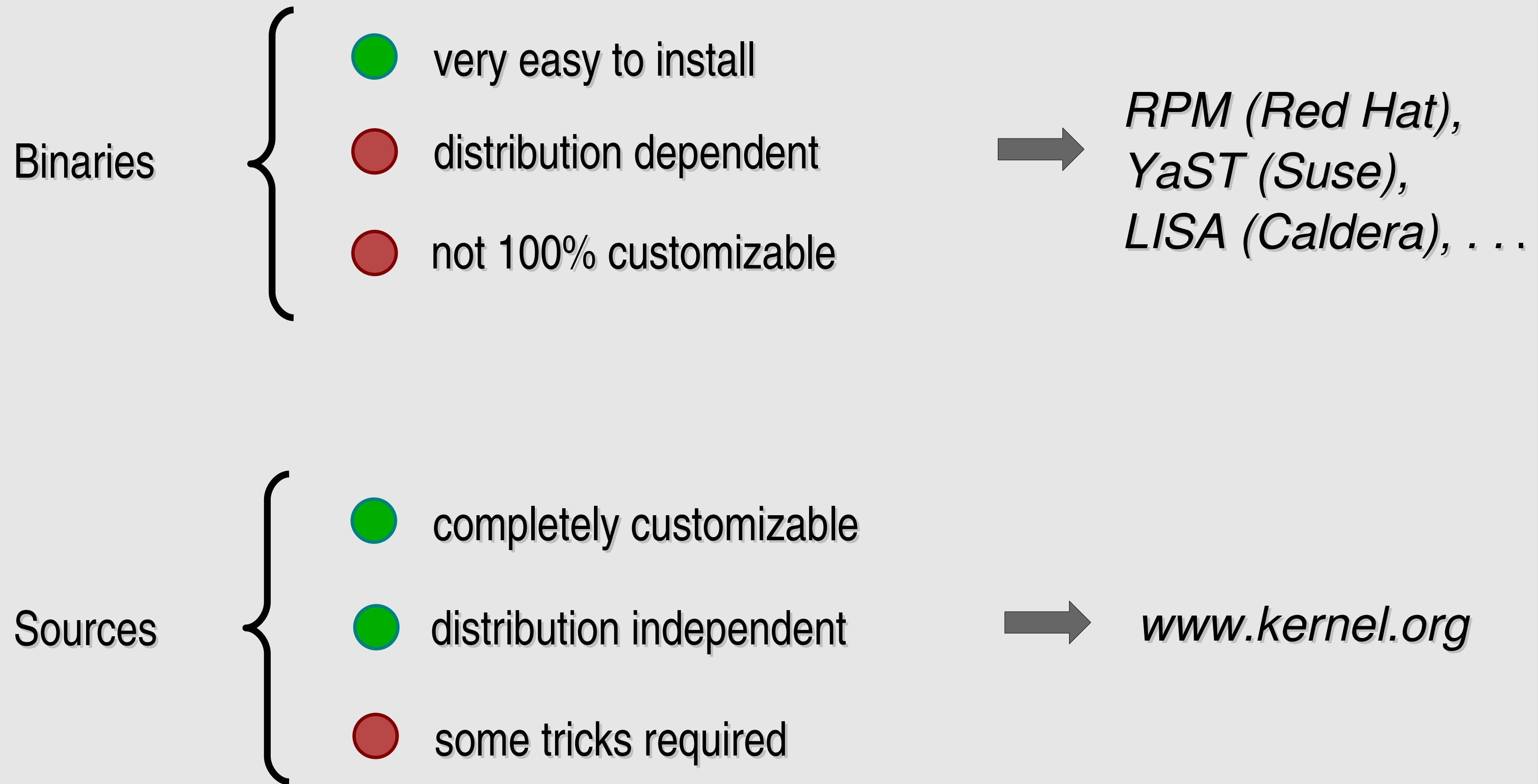
 Tips & Tricks to speed up the compilation process

Speed up the compilation time with DISTCC

Why to recompile the kernel?

- to take advantage of new features avaliable in the last kernel release
- 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 ever use
- to tailor the kernel specifically of your computer hardware, resulting in a performance boost

Obtaining a new kernel:



How to make a new kernel version

- move to the kernel sources directory:
  `cd /usr/src`

- copy the whole kernel directory:
  `cp -r linux-2.6.15 linux-2.6.15kh3`

- remove previous symbolic link: (optional)
  `rm -f linux`

- make a new symbolic link: (optional)
  `ln -s linux-2.6.15kh3 linux`

- move to the new kernel's root: (optional)
  `cd  linux`

- edit the Makefile:
  `"EXTRAVERSION = kh3"`

Understanding the kernel version numbering system (stable kernel version)

2.6.15.2

Major number (version)
represent very significant changes

Patch Level (extraversion)
bug fixes and updates

Minor number (sublevel)
new features

Understanding the kernel version numbering system (developing kernel version)

2.6.15-irqd

Major number (version)
represent very significant changes

Patch Level (extraversion)
bug fixes and updates

Minor number (sublevel)
bug fixes, updates, new features

Configure the kernel to be compiled (1)

Configuring a kernel means selecting the kernel functions and the kernel device drivers according to your hardware configuration and according to your needs. It can be accomplished in four ways:

`make config`
(terminal)

`make menuconfig`
(pseudo-graphical)

`make oldconfig`
(terminal, useful when you apply patches)

each configuration option can be answered in three possible ways by typing the character 'Y', 'M' or 'N':

'Y': feature will be compiled into the kernel image
'M': feature will be compiled but as a module
'N': feature will not be compiled

`make xconfig`
(graphical)

Configure the kernel to be compiled (2)

In order to manage all the dependencies related to the kernel configuration, a special configuration file is used.

kernel 2.2 - 2.4 ► three different configuration files, one for each utility

kernel 2.6 ► the utilities access a centralized configurations file called **`Kconfig`** using the same library, **`libkconfig`**.
The sintax used in **`Kconfig`** can be found in the kernel sources (**`Documentation/kbuild/`**).

N.B.:
The main **`Kconfig`** file is architecture dependent. For example, if you are compiling a new kernel for Intel IA32, it is **`arch/i386/Kconfig`**.

Kernel modules

- Modules are pieces of code 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
- Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image
- Without modules we are supposed to rebuild and reboot the kernel every time we want new functionalities

Configure the kernel to be compiled (3)

Example of Kconfig sintax:

```
mainmenu "Linux Kernel Configuration"

config X86_32
     bool
     default y
     help
       This is Linux's home port.  Linux was originally native to the Intel
       386, and runs on all the later x86 processors including the Intel
       486, 586, Pentiums, and various instruction-set-compatible chips by
       AMD, Cyrix, and others.

[. . .]

menu "Processor type and features"

config NR_CPUS
     int "Maximum number of CPUs (2-255)"
     range 2 255
     depends on SMP
     default "32" if X86_NUMAQ || X86_SUMMIT || X86_BIGSMP || X86_ES7000
     default "8"
     help
       This allows you to specify the maximum number of CPUs which this
       kernel will support.  The maximum supported value is 255 and the
       minimum value which makes sense is 2.
```

Compiling the kernel

Kernel compiling can be performed by 4 main steps but it depends also on the kernel version we are working with:

		Kernel 2.2 - 2.4	Kernel 2.6
step 1	building dependencies	`make dep`	`make` (steps 1–3)
step 2	building the "big compressed image" of the kernel	`make bzImage`	
step 3	building modules	`make modules`	
step 4	installing modules	`make modules_install`	`make modules_install`

Output of compilation process

```
root% make
         CHK     include/linux/version.h
         UPD     include/linux/version.h
         SYMLINK include/asm -> include/asm-i386
         HOSTCC  scripts/split-include
         HOSTCC  scripts/conmakehash
         [...]
         CC      arch/i386/kernel/process.o
         CC      arch/i386/kernel/semaphore.o
         CC      arch/i386/kernel/signal.o
         AS      arch/i386/kernel/entry.o
         CC      arch/i386/kernel/traps.o
         [...]
         CC      fs/partitions/check.o
         CC      fs/partitions/msdos.o
         LD      fs/partitions/built-in.o
         CC      fs/proc/task_mmu.o
         CC      fs/proc/inode.o
         [...]
         LD      arch/i386/boot/compressed/vmlinux
         OBJCOPY arch/i386/boot/vmlinux.bin
         HOSTCC  arch/i386/boot/tools/build
         BUILD   arch/i386/boot/bzImage
         Root device is (3, 1)
         Boot sector 512 bytes.
         Setup is 4736 bytes.
         System is 1297 kB
         Kernel: arch/i386/boot/bzImage is ready
```

Initial Ramdisk image (initrd) (1)

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

2) the kernel converts initrd into a "normal" RAM disk and frees the memory used by initrd

3) initrd is mounted read-write as root

4) `/linuxrc` is executed (this can be any valid executable, including shell scripts; it is run with uid 0 and can do basically everything init can do)

5) linuxrc mounts the "real" root file system

6) linuxrc places the root file system at the root directory using the `pivot_root()` system call

7) the usual boot sequence (e.g. invocation of `/sbin/init`) is performed on the root file system

8) the initrd file system is removed

Initial Ramdisk image (initrd) (2)

How to create an initrd in Suse Linux:

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

How to create an initrd in Red Hat Linux:

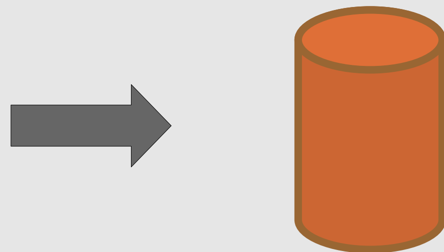
```
cd /boot
mkinitrd -v initrd-<kernel>.img <kernel>
```

`mkinitrd` creates an initial image used by the kernel for preloading the block device modules (such as IDE, SCSI or RAID) which are needed to access the root filesystem. Mkinitrd automatically loads filesystem modules (such as ext3 and jbd), IDE modules, all scsi_hostadapter entries in /etc/modprobe.conf, and raid modules if the system's root partition is on raid, which makes it simple to build and use kernels using modular device drivers.

Installing the kernel image on the system (1)

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

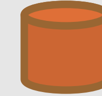
Now we must install the compressed kernel image on our system:



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

hard disk's mbr

Installing the kernel image on the system (2)



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.

Installing the kernel image on the system (3)

LILO

LInux LOader, or LILO, comes 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.

Installing the kernel image on the system (4)

Example `lilo.conf` file:

```
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
prompt
timeout=100
compact
default=Linux
image=/boot/vmlinuz-2.4.18-14
    label=Linux
    root=/dev/hdb3
    read-only
    password=linux
other=/dev/hda
    label=WindowsXP
```

`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 (4 - cont.)

Example `lilo.conf` file:

```
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
prompt
timeout=100
compact
default=Linux
image=/boot/vmlinuz-2.4.18-14
    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`: prompt for a password when you pass additional parameters at boot time

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

Installing the kernel image on the system (5)

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

Installing the kernel image on the system (6)

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.

Installing the kernel image on the system (7)

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

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

title Gentoo Linux (2.4.7-10)
    root (hd0,1)
    kernel /vmlinuz-2.4.7-10 ro
    root=/dev/hda3
    initrd /initrd-2.4.7-10.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!

Applying a kernel patch

If you wish to upgrade to a newer kernel, you can patch your current kernel instead of downloading an entire new kernel. By patching your existing kernel, you retain your settings from previous kernel compilations. Patching the kernel is a good choice if you wish to upgrade from your current patch level to the next consecutive patch level. For example, patching kernel 2.6.3 to 2.2.4 involves applying one patch. However, if you wish to upgrade the 2.6.0 kernel to 2.6.15, then a patch for each patch level must be applied sequentially. In this case, it may be better to download the entire 2.2.14 kernel.

- move the downloaded kernel patch to the /usr/src/linux directory:
  `cd /usr/src/linux`
- If you downloaded a patch with a .gz extension, execute the following command:
  `gunzip patch-2.6.x.gz`
- If you downloaded a patch with a .bz2 extension, execute the following command:
  `bunzip2 patch-2.6.x.bz2`
- apply the patch to the kernel source tree with the following command:
  `patch -p1 < patch-2.6.x`

Tips and tricks to speedup the compilation process (1)

- Avoid running **`make clean`** as much as possible since `make clean` destroys all the object files, thus forcing a recompilation of the whole kernel.
- If you are modifying only few source files in the kernel and you want to make a patch, try to use hard links instead of copying the whole kernel tree in order to speed up the process:

```
cp -al linux-2.6.15 linux-2.6.15kh3
```

But remember to break the hard link before modifying the file in this way:

```
cd /usr/src/linux-2.6.15kh3
cp kernel/fork.c kernel/1
mv kernel/1 kernel/fork.c
cd ..
diff -ruN linux-2.6.15 linux-2.6.15kh3 > mypatch
```

Tips and tricks to speedup the compilation process (2)

- If you have a multiprocessor box, you can compile the kernel using the **-j** (or **--jobs**) make option as follows:

`make -j N`	phase 1
`modules_install`	phase 2

where N is the number of jobs running concurrently; usually N is set so that:

N = #processors + 1

a) If the `-j` option is given without an argument, make will not limit the number of jobs that can run simultaneously. Usually, performances can decrease!

b) Benefits from using -j option in phase 2 are almost irrelevant.

Speed up the compilation time with DISTCC (1)

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
```

Speed up the compilation time with DISTCC (2)

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