Building the Linux kernel using git

Author's note: I am writing this in as impartial a way as I can, because I think the user should decide what is best for them.

Where to build

There are two common places to build the kernel. It is important that wherever you build the kernel, it must be a location that does not change nor risk deletion or corruption. For example do **NOT** build it in `/tmp`.

1. You can build the kernel in `/usr/src` and then you will have to be root in order to build the kernel. This has the advantage that it works well for multi-user systems.

```
mkdir -p /usr/src
cd /usr/src
su
```

2. You can build the kernel in your home directory to a directory that you should not delete. For example you can use `~/.local/src` or something similar. You can build the kernel as a regular user. The disadvantage is that other users on a multi-user system will not have access to the source.

```
mkdir -p ~/.local/src
cd ~/.local/src
```

Where to get the kernel source

There are two ways to get the kernel source. We will assume the version is as follows.

```
version=3.10.24
base=3.10
```

1. You can use git to get the kernel source. This has the advantage that it will avoid issues if linux.org is cracked. A disadvantage is that the source directory may get large as new releases are pulled.

```
git clone --depth 1
git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git # this
only needs to be run once at the very beginning
cd linux-stable
git pull # run this every time to update the source
git checkout v$version # you MUST checkout a kernel version or you will get
the base version, i.e. 3.10
git log # just check to make sure the top of the log says the version you
want
git tag -v v$version # this is not necessary, but it validates the source
using gpg public key, which you need to import
```

To easily import a gpg key, you need the RSA key ID 0000000E. Run

```
gpg --search 0000000E
```

and choose the key you want to import, or if that doesn't work then run

```
gpg --keyserver wwwkeys.pgp.net --recv-keys 0000000E
```

The key here is just an example.

2. You can download the source tarball from http://kernel.org/. The best way to stay up-to-date is to download the base version tarball and then download the individual version patch that you want, and apply it before compiling.

```
kernelsite="ftp://www.kernel.org/pub/linux/kernel/v3.x"
lftp -c "open $kernelsite; get linux-$base.tar.xz linux-$base.tar.sign"
unxz -k "linux-$base.tar.xz"
gpg --verify "linux-$base.tar.sign"
# the following is done with every new kernel update
tar -xf "linux-$base.tar.xz"
mv "linux-$base" "linux-$version"
cd "linux-$version"
lftp -c "open $kernelsite; get patch-$version.xz patch-$version.sign"
unxz "patch-$version.xz"
gpg --verify "patch-$version.sign"
patch -p1 < "patch-$version"
```

Patching a kernel can be confusing, because patches are relative to the base kernel version. Lets say you currently have the kernel source for 3.10.24, and you want to upgrade to 3.10.25. You can either revert the 3.10.24 patch (getting you back to 3.10), or you can download kernel version 3.10 and apply the 3.10.25 patch. If you try to apply the 3.10.25 patch over 3.10.24, it will fail. Thus, it is recommended to keep the source tarball for the base version in the kernel build directory, especially if you plan on regularly upgrading your kernel for security fixes.

Here is a script that downloads the latest patch for the current running kernel. It assumes you already have an xz compressed tarball in the kernel build directory.

kernupd

```
#!/bin/sh

error () # error
{
    echo "$1"
```

```
    exit 1
}

kerneldir="$HOME/.local/src"
kernelsite="ftp://www.kernel.org/pub/linux/kernel/v3.x"
currentver="$(uname -r)"
basever="$(uname -r | rev | cut -d. -f1 --complement | rev)"
newver="$(lftp -c "open $kernelsite; ls" | awk '{ print $9 }' | grep
"patch-$basever.*\.xz" | sort -V | tail -n1 | sed 's/patch-//' | sed
's/\.xz//')"

if test "$currentver" = "$newver"
then
    echo 'Kernel is up to date.'
else
    echo "Updating kernel to $newver"
    cd "$kerneldir"
    tar -xf "linux-$basever.tar.xz" || exit 1
    mv "linux-$basever" "linux-$newver" || exit 1
    cd "linux-$newver"
    lftp -c "open $kernelsite; get patch-$newver.xz patch-$newver.sign"
|| exit 1
    unxz "patch-$newver.xz"
    gpg --verify "patch-$newver.sign" || exit 1
    patch -p1 < "patch-$newver" || exit 1
    make mrproper
    zcat /proc/config.gz > .config
    make oldconfig
    echo "cd $PWD"
fi

exit 0
```

How to build

First run the following to make sure everything is setup properly for a build.

```
make clean
make mrproper # note that this deletes any .config file in the current
directory
```

You have many choices when it comes to how to build the kernel. It is best to read the README file in the kernel source directory to get all the current options, as well as `Documentation/kbuild/kconfig.txt` for details on all the options.

README

```
 - Alternative configuration commands are:
```

```
     "make config"      Plain text interface.

     "make menuconfig"  Text based color menus, radiolists & dialogs.

     "make nconfig"     Enhanced text based color menus.

     "make xconfig"     X windows (Qt) based configuration tool.

     "make gconfig"     X windows (Gtk) based configuration tool.

     "make oldconfig"   Default all questions based on the contents of
                        your existing ./.config file and asking about
                        new config symbols.

     "make silentoldconfig"
                        Like above, but avoids cluttering the screen
                        with questions already answered.
                        Additionally updates the dependencies.

     "make olddefconfig"
                        Like above, but sets new symbols to their
default
                        values without prompting.

     "make defconfig"   Create a ./.config file by using the default
                        symbol values from either arch/$ARCH/defconfig
                        or arch/$ARCH/configs/${PLATFORM}_defconfig,
                        depending on the architecture.

     "make ${PLATFORM}_defconfig"
                        Create a ./.config file by using the default
                        symbol values from
                        arch/$ARCH/configs/${PLATFORM}_defconfig.
                        Use "make help" to get a list of all available
                        platforms of your architecture.

     "make allyesconfig"
                        Create a ./.config file by setting symbol
                        values to 'y' as much as possible.

     "make allmodconfig"
                        Create a ./.config file by setting symbol
                        values to 'm' as much as possible.

     "make allnoconfig" Create a ./.config file by setting symbol
                        values to 'n' as much as possible.

     "make randconfig"  Create a ./.config file by setting symbol
```

```
                        values to random values.

     "make localmodconfig" Create a config based on current config and
                           loaded modules (lsmod). Disables any module
                           option that is not needed for the loaded
modules.

                           To create a localmodconfig for another
machine,
                           store the lsmod of that machine into a file
                           and pass it in as a LSMOD parameter.

                   target$ lsmod > /tmp/mylsmod
                   target$ scp /tmp/mylsmod host:/tmp

                   host$ make LSMOD=/tmp/mylsmod localmodconfig

                           The above also works when cross compiling.

     "make localyesconfig" Similar to localmodconfig, except it will
convert
                           all module options to built in (=y) options.

   You can find more information on using the Linux kernel config tools
   in Documentation/kbuild/kconfig.txt.
```

Yet another option is to start with one of the slackware kernel configs found in the `source/k` directory of your slackware install media. Just copy the config file in, rename it to `.config` and run one of the configuration commands, typically `make oldconfig` if you are compiling a kernel version newer than the config file you copied in.

You can also use the kernel config of your currently running kernel located at `/proc/config.gz` by running

```
zcat /proc/config.gz > .config
```

Note, however, that `/proc/config.gz` may not exist if the kernel was not configured to expose it.

If you choose to build in `/usr/src` and use `xconfig` or `gconfig` please read howtos:slackware_admin:kernelbuilding#x_and_su

What to build

First, make sure you understand the difference between built-in and module. Built-in means that the driver or feature is included in the kernel, and is loaded into RAM by the bootloader on boot. Module means that the driver is built as a module and is

loaded by the kernel once it mounts the filesystem. This is very important, because in order to load a module from the filesystem, the kernel needs drivers to handle the IDE or SATA controller as the well as the filesystem on the HDD or SDD. You **MUST** either build-in the IDE/SATA controller modules (PATA/SATA/AHCI) needed to access your HDD/SDD as well as the filesystem driver needed to mount the filesystem on which the modules are located **OR** you can make an initrd. Failing to do so will result in the `Kernel Panic-not syncing: VFS: unable to mount root fs on unknown block(8,2)` error.

So, should you build anything as a module, why not just make everything bulit-in. Certainly this is an option, and this is what is done for the slackware huge kernel, because it doesn't have access to the modules. However, on an old machine you will notice a difference between booting the huge kernel (slow) and booting the generic kernel (fast). This is because loading a large kernel into RAM takes longer than a smaller one. There is another issue with building everything into the kernel in that some drivers may conflict with one another and you won't be able to unload the modules because they are built-in. You can usually prevent built-in drivers from working on boot using the kernel command line which you can set in you bootloader config, for example `/etc/lilo.conf`.

Highlights of the kernel config

What follows is a documentation of some of the more important or interesting features of the kernel. This will not be comprehensive, because there are tons and tons of options to configure. It will be just highlights of the options considered to be more interesting or important than the rest. Read the Help on each option for more info. They are presented in the same order as they are in the kernel config along with where to find them.

> **General setup**
>
> > Local version - append to kernel release

This option lets you add a string to the end of the kernel version, in case you want to install more that one kernel of the same version.

> > Kernel compression mode

This lets you select the kernel compression mode. LZMA is the good default choice. Higher compression (LZMA/XZ) requires more RAM and processor usage, but is faster to load from disk to RAM. Lower compression (Gzip/LZO) requires less RAM and processor usage, but is slower to load from disk to RAM.

> > Kernel .config support
> >
> > > Enable access to .config through /proc/config.gz

Make sure to enable this so you can have easy access to your current kernel config at /proc/config.gz

> Automatic process group scheduling

This option can greatly improve the performance of responsiveness of multi-threaded machines. So you can run `make -j4` without causing other programs to stutter.

> Initial RAM filesystem and RAM disk (initramfs/initrd) support

Make sure to enable this if you are going to make an initrd.

> Disable heap randomization

This option should be disabled on machines make after the year 2000 for security reasons.

> Optimize very unlikely/likely branches

Can increase kernel performance.

> **Enable the block layer**
>
> Partition Types —>
>
> EFI GUID Partition support

Enable this if you plan on using the new GPT partition scheme and want to boot from UEFI.

> IO Schedulers —>
>
> Deadline I/O scheduler

Causes performance improvement if you use the JFS filesystem.

> CFQ I/O scheduler
>
> CFQ Group Scheduling support

This option allows CFQ to recognize task groups and control disk bandwidth allocation, and can improve performance.

You can enable both and test each one out by running

```
cat /sys/block/sd?/queue/scheduler
echo deadline > /sys/block/sd?/queue/scheduler
echo cfq > /sys/block/sd?/queue/scheduler
```

> **Processor type and features**
>
> > Symmetric multi-processing support

This should be enabled for multi-core and multi-processor machines.

> > Intel Low Power Subsystem Support

Enable if you have a newer system with Intel Lynxpoint PCH. Check the output of `lspci` for `Lynx Point`.

> > Processor family

For maximum performance you should choose the right processor family.

To identify an unknown processor, first run

```
cat /proc/cpuinfo
```

If you still cannot tell what processor family to choose, then look up the `cpu family` and `model` online or on the gentoo wiki.

> > Maximum number of CPUs

Set this to the number of CPUs you have.

> > SMT (Hyperthreading) scheduler support

This is for Hyperthreading machines, i.e. virtual cores.

> > Multi-core scheduler support

This is for multi-core machines, i.e. real cores.

You can enable both to be safe, in case you don't know.

> > Preemption Model

Your selection depends on how you use your computer, and will make a difference performance-wise.

>

> > > No Forced Preemption (Server)

High latency, Maximum throughput. Good for data servers.

> > > Voluntary Kernel Preemption (Desktop)

Medium latency, Medium throughput. Apps will still run smoothly when the system is under load.

> > > Preemptible Kernel (Low-Latency Desktop)

Low latency, Low throughput. Good for media servers, or low-latency embedded systems.

> Reroute for broken boot IRQs

Enable if you have problems with spurious interrupts.

> Machine Check / overheating reporting

Make sure to enable this along with Intel or AMD. This allows the kernel to respond to system overheating.

> Low address space to protect from user allocation

65536 for most architectures, 32768 for ARM. This can help with NULL pointer deference bugs.

> Transparent Hugepage Support

This can increase the performance of apps that require lots of RAM at once, such as p7zip, LZMA, LZMA2 (xz).

> > Transparent Hugepage Support sysfs defaults

Selecting always will allow hugepages to work without needing to mount hugepages or configure the application to use them, unlike madvise.

> Enable cleancache driver

Can increase performance by reducing disk I/O. However, I don't know how stable it is.

> MTRR cleanup support

Can improve performance.

> > > MTRR cleanup enable value (0-1)

1 to enable it

> > > MTRR cleanup spare reg num

This is the number of disabled/unused/spare MTRR registers.

As an example, run dmesg and look for

```
MTRR variable ranges enabled:
  0 base 000000000 mask F80000000 write-back
  1 base 07EF00000 mask FFFF00000 uncachable
  2 base 07F000000 mask FFF000000 uncachable
  3 disabled
  4 disabled
  5 disabled
  6 disabled
```

Here, registers 3,4,5,6 are disabled/unused/spare, so MTRR cleanup spare reg num = 4 total unused registers.

> > > EFI runtime service support

Enable if you want to boot from UEFI.

> > > EFI stub support

Enable if you want to boot from UEFI.

> > > Enable -fstack-protector buffer overflow detection

Can prevent buffer overflows on systems with gcc version 4.2 and up.

> > > Timer frequency

Always pick 1000Hz for systems that need to run multimedia. This number is proportional to interactive responsiveness. You want lower frequencies on servers and higher frequencies on desktops. However, if you have Tickless System enabled timer interrupts will only trigger as-needed, so it may be best to round up.

Power management and ACPI options

ACPI

Always enable: Button, Fan, Processor, Thermal Zone. Without these, your computer (especially laptops) may overheat because ACPI cannot access thermal monitoring or fans.

CPU Frequency scaling

CPU Frequency scaling

Enable this if you have frequency scaling enabled in the UEFI/BIOS (Speedstep or similar).

'performance' governor

This sets CPU frequency to the maximum available.

'powersave' governor

This sets CPU frequency to the minimum available.

'userspace' governor for userspace frequency scaling

This allows userspace programs to set the CPU frequency.

'ondemand' cpufreq policy governor

This governor is recommended for desktops.

'conservative' cpufreq governor

This governor is recommended for laptops/netbooks. Although similar to the 'ondemand' governor, frequency is gracefully increased and decreased rather than jumping to 100% when speed is required.

x86 CPU frequency scaling drivers

Intel P state control

This driver is mutually exclusive with the ACPI Processor P-States driver. It is a newer driver for Sandy Bridge processors and may cause problems.

> Processor Clocking Control interface driver

This is only required for HP ProLiant servers, which use this interface. Otherwise, disable it.

> ACPI Processor P-States driver

This is the recommended driver for newer CPUs Intel (Enhanced) Speedstep enabled and AMD K10 and newer.

> AMD Opteron/Athlon64 PowerNow!

This is for K8/early Opteron/Athlon64 processors.

> Intel Enhanced SpeedStep (deprecated)

This is a deprecated option that has been superseded by the ACPI Processor P-States driver, so leave this disabled.

> Intel Pentium 4 clock modulation

This is a hack for Pentium 4s that may cause severe slowdowns and noticeable latencies, so disable it.

> **Bus options**
>
> Message Signaled Interrupts (MSI and MSI-X)

This can offload IRQ interrupts by using MSI instead. However, if your BIOS is buggy this may need to be disabled.

> **Executable file formats / Emulations**
>
> Kernel support for scripts starting with #!

You **MUST** say yes here or you will not be able to run script that start with #! and then the interpreter.

> IA32 Emulation

Allows 32-bit emulation via multi-lib on 64-bit systems.

> **Device Drivers**
>
> Block devices
>
> Normal floppy disk support

Should be module or the floppy drive may malfuction.

> Loopback device support

Required by some disk encryption methods.

> SCSI device support
>
> Asynchronous SCSI scanning

Will speed up booting, but all SCSI drivers **must** be built-in for it to work properly.

> Serial ATA and Parallel ATA drivers
>
> AHCI SATA support

If you enable AHCI in the BIOS this **must** be built-in or part of initrd. If you don't use AHCI, then do the same but for your SATA or PATA driver.

> Multiple devices driver support (RAID and LVM)
>
> Device mapper support
> Crypt target support

Required by cryptsetup.

> Input device support —>
>
> Keyboards —>
>
> AT keyboard

Enable this if you have a standard AT or PS/2 keyboard.

> Mice —>
>
> PS/2 mouse

Enable this if you have a PS/2 mouse.

> Multimedia support —>

Media USB Adapters —>
USB Video Class (UVC)
UVC input events device support

If you have a newer webcam, you should enable this or it won't work.

Graphics support
/dev/agpgart (AGP Support)
Intel 440LX/BX/GX, I8xx and E7x05 chipset support

Enable this if you have an integrated Intel card.

Direct Rendering Manager

This should almost always be enabled.

Nouveau (nVidia) cards

If you use nouveau, you only need this driver, because it has framebuffer support.

Support for frame buffer devices
VESA VGA graphics support

This should almost always be enabled.

EFI-based Framebuffer Support

You should enable this if you want to boot from UEFI.

nVidia Framebuffer Support

Again this is **NOT** needed by nouveau.

Sound card support
Advanced Linux Sound Architecture
PCI sound devices
Intel HD Audio

> > > > > Pre-allocated buffer size for HD-audio driver

A larger buffer (e.g. 2048) is preferred for systems using PulseAudio.

> USB support
> > xHCI HCD (USB 3.0) support

You should enable this for newer systems.

> > > EHCI HCD (USB 2.0) support
> > > Improved Transaction Translator scheduling

Can greatly improve transfer speeds over USB 2.0 in some cases.

> Real Time Clock
> > PC-style 'CMOS'

Enable this or you won't be able to set the clock from Linux. It is often accidentally omitted.

> > Staging drivers —>

Avoid drivers in this section if you value system stability.

> > Generic Dynamic Voltage and Frequency Scaling (DVFS) support

This provides frequency scaling support for devices. It also has governors similar to those for CPU frequency scaling. I'm not sure what devices require this.

> **Firmware Drivers**
> > EFI (Extensible Firmware Interface) Support —>
> > > EFI Variable Support via sysfs

This option is deprecated in favor of EFI Variable filesystem. It can cause data inconsistency issues

> **File systems**

You **MUST** either build-in the driver for the filesystem on which the kernel modules reside **OR** create an initrd, or the kernel will **NOT** boot. You should also build-in the driver for the filesystem on which the kernel itself resides in case it resides on a different filesystem type than the modules e.g. EFI VFAT partition.

> > DOS/FAT/NT Filesystems

> > > MSDOS fs support
> > > VFAT (Windows-95) fs support

You need these built-in if you use EFI. The kernel will boot if they are modules, but catch22 type situations can easily result.

> > > NTFS file system support
> > > NTFS write support

You need this if you want to write to NTFS filesystems.

> CD-ROM/DVD Filesystems
> > UDF file system support

You need this if you plan on reading or writing disks with the UDF filesystem.

> > > Miscellaneous filesystems —>
> > > > EFI Variable filesystem

You should enable this if you want to boot from UEFI.

> **Kernel hacking**
> > Magic SysRq key

Enable if you want to use SysRq REISUB (p.cogx on dvorak) to safely shutdown a hung system.

> > Allow gcc to uninline functions marked 'inline'

Enable for gcc 4.x but not for gcc 3.x

> **Security Options**
> > Restrict unprivileged access to the kernel syslog

Disable this if you want users to be able to use the 'dmesg' command.

> **Cryptographic API**

Make sure to build-in all algorithms you plan to use for cryptography using cryptsetup, especially if you plan on full disk encryption, otherwise you won't be able to decrypt your disk and thus will not be able to boot. Note that there are optimized and 64-bit versions to choose from.

> > Parallel crypto engine

Converts an arbitrary crypto algorithm into a parallel algorithm that executes in kernel threads. It allows multi-threading of any crypto algorithm.

> **Library routines**
>
> > JEDEC DDR data

You should probably enable this so that the JEDEC data from your RAM is available to drivers that need it.

If you are wondering what drivers you need, make sure to take a look at the output of these commands.

```
lsmod
/sbin/lspci -k
lsusb
dmesg
```

You may want to consider disabling Staging drivers and EXPERIMENTAL and OBSOLETE drivers if you want a stable, modern kernel. Some can be left on for good reason.

Building

To speed up building, you can use the `-j` option for make. The maximum it can be set to is the number of cores or processors plus one. However, if you want to do something while the kernel is building, like browse the web, you may want to use just the number of cores or processors.

```
cores=4
```

```
make -j$cores
```

Installing

Before installing the kernel, you may want to uninstall Slackware packages containing the default kernels, modules, and source. You can also choose to leave these packages installed if you plan on using these default kernels. Whatever you do, do **NOT** uninstall the old kernel headers. See Alien Bob's guide for more on these topics.

First, you should make sure to remove any previously installed kernel modules at `/lib/modules/$version`.

```
su
```

```
rm -r /lib/modules/$version # make sure you know what is being removed
```

Now, you can install the new modules.

```
make modules_install
```

Next you should install the kernel itself. This script will install the kernel and make sure it is installed properly.

installkernel.sh

```
#!/bin/sh
# installs kernel only, this should be run only from the kernel source
directory

error() # error
{
    echo "ERROR: $1"
    exit 1
}

# make sure we are root
if test ~ != '/root'
then
    error 'This script must be run as root'
fi

# remove the old
rm -f /boot/config.old
rm -f /boot/System.map.old
rm -f /boot/vmlinuz.old

# rename the present
mv /boot/config /boot/config.old
mv /boot/System.map /boot/System.map.old
mv /boot/vmlinuz /boot/vmlinuz.old

# copy in the new
cp arch/x86/boot/bzImage /boot/vmlinuz
cp System.map /boot
cp .config /boot/config

# for elilo
bootdir="/boot/efi/EFI/Slackware"
if test -d "$bootdir"
then
    cp arch/x86/boot/bzImage "$bootdir"/vmlinuz
fi
```

```
# change permissions of vmlinuz
chmod a-rwx,u+r /boot/vmlinuz

# check install
echo
if cmp arch/x86/boot/bzImage /boot/vmlinuz
then
    echo 'Kernel installed correctly'
else
    error 'Kernel install failed'
fi
if cmp System.map /boot/System.map
then
    echo 'System.map installed correctly'
else
    error 'System.map install failed'
fi
if cmp .config /boot/config
then
    echo 'Kernel config installed correctly'
else
    error 'Kernel config install failed'
fi
if test -d "$bootdir"
then
    if cmp arch/x86/boot/bzImage "$bootdir"/vmlinuz
    then
        echo 'kernel installed to EFI correctly'
    else
        error 'kernel install to EFI failed'
    fi
fi
echo
echo 'Kernel install completed successfully'
echo 'Remember to run lilo if you use it'
echo

exit 0
```

lilo

If you chose not to build-in the IDE/SATA controller modules (PATA/SATA/AHCI) needed to access your HDD/SDD as well as the filesystem driver needed to mount the filesystem on which the modules are located, you **MUST** make an inird.

If you use lilo, which is the default bootmanager on Slackware, you should edit `/etc/lilo.conf` and then run

```
lilo
```

Here is an example lilo.conf

```
# append options here if you need any kernel parameters on boot
append=" vt.default_utf8=0"
# this should point to the device you want to boot
boot = /dev/sda
# not necessary, but lilo complains if it is not here
lba32
# the compact option speeds up boot time significantly, but may
not work on all systems
compact
# This is needed if you want lilo to prompt you for what to boot
prompt
# This is given in tenths of a second, so 600 for every minute.
You can comment this out if you single-boot.
timeout = 1200
# Override dangerous defaults that rewrite the partition table:
change-rules
  reset
# Normal VGA console, the safest choice.
# You can choose others if you want a framebuffer console, but
you must have framebuffer support or the screen will go black.
vga = normal
# End LILO global section
# Linux bootable partition config begins
# the path to the Linux kernel boot image
image = /boot/vmlinuz
# the partition where the Linux kernel is located
  root = /dev/sda1
# how this entry is shown on the boot screen
  label = Linux
  read-only
# Linux bootable partition config ends
```

Sources

- The howto was written by H_TeXMeX_H
- The README is an excerpt from the README included with the kernel source.
- The installkernel.sh script was written by H_TeXMeX_H
- The lilo.conf excerpt is from the default lilo.conf that comes with Slackware plus a few options.
- Thanks to Alien Bob for his kernel building howto. It inspired me to write a restructured and

updated kernel building guide.
- Updated by metaschima

howtos, author htexmexh, kernel, software