2. The kernel source

The kernel is the part of the system that handles the hardware, allocates resources like memory pages and CPU cycles, and usually is responsible for the file system and network communication.

2.1 Kernel versions

Linux kernels have a peculiar numbering system. After 0.01 and 0.02 that were very preliminary,

```
From:       torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject:    Free minix-like kernel sources for 386-AT
Date:       5 Oct 91 05:41:06 GMT

Do you pine for the nice days of minix-1.1, when men were men and wrote
their own device drivers? Are you without a nice project and just dying
to cut your teeth on a OS you can try to modify for your needs? Are you
finding it frustrating when everything works on minix? No more all-
nighters to get a nifty program working? Then this post might be just
for you :-)

As I mentioned a month(?) ago, I'm working on a free version of a
minix-lookalike for AT-386 computers.  It has finally reached the stage
where it's even usable (though may not be depending on what you want),
and I am willing to put out the sources for wider distribution.  It is
just version 0.02 (+1 (very small) patch already), but I've successfully
run bash/gcc/gnu-make/gnu-sed/compress etc under it.

Sources for this pet project of mine can be found at nic.funet.fi
(128.214.6.100) in the directory /pub/OS/Linux. ...
```

we got 0.10, 0.11, 0.12, already usable, next 0.95, almost there. But from 0.95 to 1.0 was a long way, and after 0.99 we got 0.99pl1, ..., 0.99pl15, where the later patch levels were again subdivided 0.99pl14a, ..., 0.99pl14z, 0.99pl15a, ..., 0.99pl15j. After the stable 1.0 the development series was called 1.1, and the next stable kernel was 1.2. At this point a rule was defined:

There are development kernels, that quite possibly may eat your disks. Never boot them without making sure that you have a good backup. These are the versions with an odd number in the middle, like 2.5.37. And there are "stable" versions, with an even number in the middle, like 2.0.39, or 2.2.20, or 2.4.19. No version is bugfree, so also the stable versions change slowly (e.g. from 2.4.0 to 2.4.19) but a more or less recent stable version should work fine on ordinary hardware. Vendors like SuSE and RedHat often have private kernel modifications, so system behaviour may change when you switch to an "official" kernel.

Since 2.6 the situation is a bit different: stable and development series have merged now. Patches are first tried out in the -mm series, release candidates are suffixed -rcN for some N, bug fixes after a stable release get a fourth digit (like 2.6.8.1 after 2.6.8).

The command

```
finger linux@kernel.org
```

will tell you the version numbers of various kernel versions.

2.2 Obtaining the kernel source

Make sure you have half a GB free somewhere. Get the kernel by anonymous ftp from `ftp://ftp.nl.kernel.org` (with `nl` replaced by something else if this mirror doesn't have the most recent stuff yet; `de` and `fi` do more frequent updates). For the most recent kernel versions, the appropriate directory is `/pub/linux/kernel/v2.6`. There are full distributions, like `linux-2.6.8.1.tar.gz` (44 MB), or patches relative to the previous version, like `patch-2.6.8.1.gz`. Release candidates live in the `testing` subdirectory, and have names like `patch-2.6.9-rc1.bz2`. Very recent snapshots (as patch relative to the current kernel version) live in the `snapshots` subdirectory, and have names like `patch-2.6.9-rc1-bk9.bz2`. Finally, the individual changesets still live in a `v2.5` directory, see kernel/v2.5/testing/cset/.

filename	unpack command
`xxx.gz`	`gunzip xxx.gz`
`xxx.bz2`	`bzip2 -d xxx.bz2`
`xxx.tar.gz`	`tar xfz xxx.tar.gz`
`xxx.tar.bz2`	`tar xfj xxx.tar.bz2`

For example, the commands

```
% ncftp ftp.nl.kernel.org
ncftp / > cd /pub/linux/kernel/v2.6
ncftp /pub/linux/kernel/v2.6 > get linux-2.6.0-test11.tar.bz2
ncftp /pub/linux/kernel/v2.6 > quit
% tar xvjf linux-2.6.0-test11.tar.bz2
```

will give you a kernel source tree (212 MB).

There are various other access methods, by http, rsync, git etc, see kernel.org.

Distribution vendors have their own kernel patches. On kernelnewbies one can browse recent vendor patches.

Hypertext index of the kernel source

See lxr.linux.no.

Source management

In the beginning, the Linux source management tools were email, diff and patch. Since Feb 2002, Linus used Larry McVoy's bitkeeper distributed source management system. The fact that this is a commercial system, and not open source, was a source of a lot of controversy. In April 2005 Linus announced that he dropped bitkeeper. Within a few weeks he developed a new source management system, git.

Marcelo's old 2.4 tree is still visible in bitkeeper format at linux.bkbits.net.

Linus' current tree is visible as linux-2.6.git. The actual files.

Some notes on the passive use of git: First repository download:

```
% git-clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git linux-2.6
```

This gets the kernel source tree, today 536 MB. Of this, the git metadata takes 218 MB, almost all of which belongs to `.git/objects/pack/pack-f91...99fc.idx` (15 MB) and `.git/objects/pack/pack-f91...99fc.pack` (200 MB).

What did we get? Compare with the result of getting linux-2.6.23.tar.bz2, upgrading to 2.6.24-rc3 using testing/patch-2.6.24-rc3.bz2, upgrading to 2.6.24-rc3-git6 using snapshots/patch-2.6.24-rc3-git6. Now

```
% diff -r tar/linux-2.6* git/linux-2.6
Only in git/linux-2.6: .git
diff -r tar/linux-2.6*/Makefile git/linux-2.6/Makefile
4c4
< EXTRAVERSION = -rc3-git6
---
```

```
> EXTRAVERSION = -rc3
%
```

So this git clone produced roughly the same result as getting the tar file, except that the additional subdirectory .git contains the entire metadata and history since git was started.

We can look at the history since 2005-04-16 (Linux-2.6.12-rc2) by

```
% git log | less
```

and see that there were 74426 commits in less than 1000 days, about 77 per day on average, with a maximum of 914 commits on Tue Oct 16 2007. A graphical version of the history is given by gitk.

More history:

```
% git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/old-2.6-bkcvs.git old-2.6-bk
```

yields a repository with the history from 2002-02-05 to 2005-04-04, all data from the bitkeeper time.

One can browse a lot of git repositories at git.kernel.org.

Upgrading to current is done by

```
% cd git/linux-2.6
% git pull git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
```

Looking at the changes for a file, say `fs/autofs/inode.c` is done by

```
% git-whatchanged fs/autofs/inode.c
```

(for the commit messages), or

```
% git-whatchanged -p fs/autofs/inode.c
```

(for the actual changes).

2.3 Compiling the kernel

Do a `cd` into the top source directory (with `arch`, `Documentation`, etc.), then do something like

```
make oldconfig
make depend
make bzImage
```

There is a number of configuration targets, like `xconfig`, `menuconfig`, `oldconfig`. Pick one and specify what kind of a kernel you want. (What hardware must be supported, what filesystem types, etc.) If you are greedy and make a big kernel, you cannot boot from floppy. If you are very greedy, you may not be able to boot at all, so asking for "everything" may be counterproductive. The target `oldconfig` means "the same as last time", and uses the file `.config` that you hopefully still have. When starting from scratch it may take a number of attempts before you have a kernel that boots and supports your hardware.

Recent kernels may require recent compiler or binutils to compile, and recent utilities to use. See `Documentation/Changes`.

2.4 Booting a new kernel

Probably you use some boot loader, like `lilo` or `grub`. Or perhaps you want to boot from a floppy.

After compilation you have a file `arch/i386/boot/bzImage` (assuming that was on a PC). For a bootfloppy, `dd` this file to an empty diskette. Otherwise, copy the file to `/boot`, add it to `/etc/lilo.conf` or `/boot/grub/grub.conf` or so, run `lilo` and `lilo -R` if you are a lilo user, and reboot into your new kernel.

Grub allows you a menu with possible kernels to boot. That is nice. Lilo has a much better feature: `lilo -R` allows you to set the kernel to boot into for the next time only. So, for kernel development lilo is easier than grub: make a new kernel, try a boot, probably something will fail, and the next reboot is into the good old solid kernel again.

On the other hand, lilo has a disadvantage: you **must** rerun `lilo` after installing a new kernel, and very obscure things will happen if you forget.

Never delete your old kernel. Maybe the new one doesn't work. (And if you use Appletalk only once a month, and Appletalk doesn't work in the new kernel, it'll take a month before you notice.)

Problems booting

What if the kernel doesn't boot? There are many possible explanations.

If the boot crashes very early, say, after

```
Uncompressing Linux... Ok, booting the kernel.
```

then check that it was compiled for the right hardware. If the processor type was chosen correctly, check whether things go better with fewer options selected. A too big kernel will crash.

If the crash comes later, then hopefully the boot messages give some indication of what point in the boot process was reached when things went wrong.

A common mistake is to want to have as many modules as possible. If the driver needed to access disk (or partition, or filesystem) for the root filesystem is a module, then it must be loaded before it can be loaded from disk, and that is impossible. The typical reaction is the panic

```
Kernel panic: VFS: Unable to mount root from 08:07
```

where the hex numbers indicate the device.

Examples of config files

Example of a `grub.conf` file:

```
# /boot/grub/grub.conf
#
default=0
timeout=10
splashimage=(hd0,1)/boot/grub/splash.xpm.gz
title 2.4.18-pre7-ac3a-unclip-scsi
        root (hd0,1)
        kernel /boot/bzImage-2.4.18-pre7-ac3a-unclip-scsi ro root=/dev/hda2
title 2.4.20pre4bs
        root (hd0,1)
        kernel /boot/bzImage-2.4.20pre4bs ro root=/dev/hda2
title Red Hat Linux (2.4.7-10)
        root (hd0,1)
        kernel /boot/vmlinuz-2.4.7-10 ro root=/dev/hda2
        initrd /boot/initrd-2.4.7-10.img
title WINNT
        rootnoverify (hd1,1)
        chainloader +1
title DOS
        rootnoverify (hd0,0)
        chainloader +1
```

Example of a `lilo.conf` file:

```
# /etc/lilo.conf
#
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
prompt
timeout=50
image=/boot/vmlinuz-2.0.34-0.6
        label=linux
        root=/dev/hda5
        read-only

# 2.2.1 plus disk output
image=/boot/bzImage-test
        label=test
        root=/dev/hda5
        read-only

other=/dev/hda1
        label=w95
        table=/dev/hda
```

and another one:

```
# /etc/lilo.conf
#
boot    = /dev/hda
#
disk    = /dev/hda
  bios  = 0x80
disk    = /dev/hdb
  bios  = 0x81
disk    = /dev/hde
  bios  = 0x82
disk    = /dev/hdf
  bios  = 0x83
#
change-rules
reset
read-only
menu-scheme = Wg:kw:Wg:Wg
lba32
prompt
timeout = 80
message = /boot/message

  image = /boot/bzImage-2.5.38a
  label = 2.5.38a
  root  = /dev/hdb6
  append = "rootfstype=ext3 hdc=ide-scsi"

  image = /boot/bzImage-2.4.19a
  label = 2.4.19a
  root  = /dev/hdb6
  append = "rootfstype=ext3 hdc=ide-scsi"

  # SuSE kernel; warning: eth0 and eth2 interchanged
  image  = /boot/vmlinuz
  label  = suse
  root   = /dev/hdb6
  vga    = 791
  initrd = /boot/initrd
  append = " ide=nodma apm=off acpi=off hdc=ide-scsi"

  image  = /boot/memtest.bin
  label  = memtest86
```

Exercise *Get, compile and boot 2.4.17. It is stable.*

2.5 Modules

It is possible to load kernel code at run time without rebooting. Most hardware and filesystem drivers exist as module, and the commands `lsmod`, `insmod`, `rmmod` allow you to manipulate them. Normally life is easier if you just compile all you need into the kernel, but when developing a new driver it is very useful to be able to `insmod` the latest version, try it, `rmmod` again, edit and compile the driver, and repeat.

There is a kernel module loader that, when enabled, will find and load modules automatically when needed. Since modules come from disk, the code required to find them (disk driver, filesystem driver) cannot itself be a module.

During configuration there are the choices Y=yes, N=no, M=module. The command `make modules` compiles the modules.

It is very easy to make modules oneself. An example:

```
/*
 * demo-module.c
 *
 * Compile with
 *   gcc -I/path-to-linux-tree/include -D__KERNEL__ -DMODULE -O2 \
 *     -Wall -Wstrict-prototypes -c -o demo-module.o demo-module.c
 */

#include <linux/init.h>
#include <linux/module.h>

static int __init demo_init(void) {
        printk("initializing..\n");
        return 0;
}

static void __exit demo_exit(void) {
        printk("goodbye!\n");
}

module_init(demo_init);
module_exit(demo_exit);
MODULE_LICENSE("GPL");
```

Now after compilation an `insmod demo-module.o` will produce the message `initializing..`, while `rmmod demo-module` will say `goodbye!`. (Where are these messages? Wherever you are sending kernel output. Maybe on some virtual console, or maybe in a system log, like `/var/log/messages`. Recent messages are also visible in the output of the `dmesg` command.)

Written like this, the module will work both when compiled into the kernel, and when inserted as a separate module.

Later we'll make less trivial modules.

Why this MODULE_LICENSE? After getting many kernel bug reports caused by the insertion of third party modules that were binary-only and hence cannot be fixed, developers made the kernel keep track of any non-open code encountered, so that bug reports involving non-open code could be simply disregarded. One says that the kernel was *tainted*.

Some docs on kernel modules live on www.faqs.org. Note - this is rather outdated material, written mostly for 2.0 and 2.2. Many details are a bit different now.

Modules under Linux 2.5 and 2.6

The above was good enough for normal purposes under Linux 2.4. Things have become more complicated under Linux 2.5, and it is now easiest not to invoke gcc by hand but to let the kernel make system do the job. Instead of invoking `gcc` we make a 1-line `Makefile`:

```
# cat Makefile
obj-m := demo-module.o
#
```

and then compile the module with

```
make -C /path-to-linux-tree SUBDIRS=$PWD modules
```

This time the module is called `demo-module.ko` with `.ko` instead of `.o`. And everything works:

```
# insmod demo-module.ko
initializing..
# rmmod demo-module
goodbye!
```

Modules and copyright

Above we mentioned that the kernel is distributed under GPL. What about modules?

Copyright law protects a work and derived works. Remains the question how far "derived" reaches in case of the kernel. Are user space programs "derived works"? The file `COPYING` says

```
   NOTE! This copyright does *not* cover user programs that use kernel
 services by normal system calls - this is merely considered normal use
 of the kernel, and does *not* fall under the heading of "derived work".
```

making clear that no copyright on user space programs is claimed. Good. Are modules derived works? Generally the answer is claimed to be Yes. Here some fragments of conversation.

Links

The module programming guide (2001).

2.6 Subsystems - layout of the tree

The kernel initialization code (after the architecture-specific part has finished) is found in `init`.

The process handling (fork, signal, exit) lives in `kernel`.

The Unix filesystem interface (open, close, read, write, chdir, link, unlink, stat, mount, umount) lives in the subdirectory `fs`.

SysV interprocess communication in `ipc`.

Next, the various filesystems. They live in subdirectories below `fs` (like `adfs`, `affs`, `autofs`, `bfs`, `coda`, `cramfs`, `devfs`, `devpts`, `driverfs`, `efs`, `ext2`, `ext3`, `fat`, `hfs`, `hpfs`, `intermezzo`, `isofs`, `jfs`, `minix`, `msdos`, `ncpfs`, `nfs`, `ntfs`, `proc`, `qnx4`, `ramfs`, `reiserfs`, `romfs`, `smbfs`, `sysv`, `udf`, `ufs`, `vfat`, `xfs`). And so does the partition table reading code (in `fs/partitions`).

Then the memory management, `kmalloc()`, swapping, etc. is found in `mm`.

Then the network code (not the device drivers, but the various protocols, such as TCP/IP), in `net`. The device drivers, say for ethernet or so, live in `drivers/net`.

Linux runs on lots of different architectures: Intel and non-Intel PC, DEC Alpha, Apple MacIntosh, PPC, Atari, etc. etc. All architecture-specific code is found under `arch` (maybe with subdirectories `alpha`, `arm`, `arm26`, `cris`, `h8300`, `i386`, `ia64`, `m68k`, `m68knommu`, `mips`, `mips64`, `parisc`, `ppc`, `ppc64`, `s390`, `s390x`, `sh`, `sparc`, `sparc64`, `um`, `v850`, `x86_64`).

The I/O subsystem and device drivers live mostly under `drivers`. It has subdirectories `scsi`, `ide`, `usb` etc.

The include files for the kernel source live under `include` (and not under `/usr/include`). It is a very bad idea to make a symlink `/usr/include/linux` pointing at some development source tree.

Under `lib` some useful library routines. Under `scripts` some scripts used for kernel compilation.

A random collection of docs lives under `Documentation`, some parts nicely formatted with DocBook markup. The commands `make htmldocs` and `make psdocs` will take the specially formatted doc-comments from the kernel source, producing `kernel-api.html` and `kernel-api.ps`, describing the public interfaces to the various kernel subsystems.

2.7 The C code

The Linux kernel is written in C, and compiled by gcc. Various gcc extensions are used, so it is not easy to use a different compiler. A few architecture-specific fragments are in assembler.

The code is self-contained, no library is linked in. In particular, standard routines like `printf()` are not available. (But there is `printk()` that does more or less the same.) However, the kernel has its own library, and things like `strcpy()` exist.

The desired style should be obvious from looking at the main parts of the kernel. Some device drivers are painful to behold.

Linus commented on the desired coding style in the file `Documentation/CodingStyle`. A quote:

```
First off, I'd suggest printing out a copy of the GNU coding standards,
and NOT read it.  Burn them, it's a great symbolic gesture.
```

Generally, Kernigham & Ritchie style layout is used, with 8-space tabs. Try `indent -kr -i8`.

Goto

Note that `goto`'s are not avoided. Instead of

```
        err = allocate_foo();
        if (!err) {
                err = allocate_bar();
                if (!err) {
                        err = allocate_baz();
                        if (!err) {
                                ...
                        } else {
                                deallocate_bar();
                                deallocate_foo();
                        }
                } else
                        deallocate_foo();
        }
        return err;
```

or

```
        err = allocate_foo();
        if (err)
                return err;
        err = allocate_bar();
        if (err) {
                deallocate_foo();
                return err;
        }
```

```
        err = allocate_baz();
        if (err) {
                deallocate_bar();
                deallocate_foo();
                return err;
        }
        ...
```

one writes the much clearer

```
        err = allocate_foo();
        if (err)
                goto out;
        err = allocate_bar();
        if (err)
                goto out1;
        err = allocate_baz();
        if (err)
                goto out2;
        ...

out2:
        deallocate_bar();
out1:
        deallocate_foo();
out:
        return err;
```

Not only is this cleaner, it is also faster: the main path of the code is not cluttered with error-handling code.

__user and sparse

Linus wrote a C parser meant to keep track of the distinction between pointers to kernel and to user space. The latter are annotated with `__user`. The parser is known as `sparse`, and has a bitkeeper home. There is also a [non-bitkeeper source](). After installing this parser, one may run a check by doing `make C=1` (check the source files that get recompiled) or `make C=2` (check all). This is very incomplete, work in progress.

Lists

A standard idiom found throughout the kernel are the list handling primitives `list_add`, `list_del`, `list_entry`, `list_for_each`, `list_for_each_entry`.

These primitives describe doubly linked circular lists, with nodes

```
struct list_head {
        struct list_head *next, *prev;
};
```

with the obvious

```
/* add new node after given node */
void list_add(struct list_head *new, struct list_head *before) {
        struct list_head *after = before->next;

        after->prev = new;
        new->next = after;
        new->prev = before;
        before->next = new;
}
```

and

```
void list_del(struct list_head *entry) {
        entry->prev->next = entry->next;
        entry->next->prev = entry->prev;
}
```

Now `list_for_each` runs over the list nodes different from the starting node:

```
#define list_for_each(pos, head) \
        for (pos = (head)->next; pos != (head); pos = pos->next)
```

It is called with a local temp variable `pos`.

Usually a list_head is member of a larger structure. For example, a `struct inode` contains `struct list_head i_hash, i_list, i_dentry, i_devices` four such structs, so that each inode is on four cyclic lists. If we walk such a cyclic list, then we find for example the address of the `i_dentry` field of the inode; in order to get the address of the inode itself we have to subtract the offset of the `i_dentry` field in a struct inode. This is done by the macro `container_of`:

```
#define container_of(ptr, type, member) ({                      \
        const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
        (type *)( (char *)__mptr - offsetof(type,member) );})
```

Thus, `container_of(x, struct inode, i_dentry)` returns the address of the inode that has an `i_dentry` field with address `x`.

In this list handling context we have the alias `list_entry` of `container_of`:

```
/**
 * list_entry - get the struct for this entry
 * @ptr:        the &struct list_head pointer.
 * @type:       the type of the struct this is embedded in.
 * @member:     the name of the list_struct within the struct.
 */
#define list_entry(ptr, type, member) \
        container_of(ptr, type, member)
```

Now we can iterate over all structs containing the nodes of a given cyclic list using `list_for_each_entry`:

```
#define list_for_each_entry(pos, head, member)                          \
        for (pos = list_entry((head)->next, typeof(*pos), member);      \
             &pos->member != (head);                                    \
             pos = list_entry(pos->member.next, typeof(*pos), member))
```

Again, this visits all such structs except for the starting one.

ERR_PTR

Another idiom found all over the place is defined in `err.h`: ERR_PTR casts a long to a pointer, PTR_ERR casts a pointer to a long, and IS_ERR applied to a pointer checks whether the numerical value lies between -999 and -1 (inclusive) and hence by convention is the negative of an error number.

Links

Rusty's unreliable kernel hacking guide.

2.8 Logging kernel messages

The kernel prints messages using the `printk()` function. We want to see them somewhere on a console, and also log them to a file.

Loglevel

The function `printk()` (see `kernel/printk.c`) is very similar to the well-known `printf()` in user space. One difference is that texts printed start with a priority level in the form of a string like `"<4>"`.

```
#define KERN_EMERG      "<0>"   /* system is unusable                   */
#define KERN_ALERT      "<1>"   /* action must be taken immediately     */
```

```
#define KERN_CRIT       "<2>"   /* critical conditions                  */
#define KERN_ERR        "<3>"   /* error conditions                     */
#define KERN_WARNING    "<4>"   /* warning conditions                   */
#define KERN_NOTICE     "<5>"   /* normal but significant condition     */
#define KERN_INFO       "<6>"   /* informational                        */
#define KERN_DEBUG      "<7>"   /* debug-level messages                 */
```

In `/proc/sys/kernel/printk` we see four numbers: `console_loglevel` (7), `default_message_loglevel` (4), `minimum_console_loglevel` (1), `default_console_loglevel` (7).

Messages with priority larger than `console_loglevel` are printed to the console. (Higher priority means smaller priority level.) Lines without explicit priority indication are printed with priority `default_message_loglevel`. The smallest allowed value of `console_loglevel` is `minimum_console_loglevel`. The default value of `console_loglevel` is `default_console_loglevel`. (Thus, by default, all is printed to the console, except for debug messages. Distributions often set the `console_loglevel` to 1, suppressing messages to the console that might disturb users.)

These values can be changed using the `syslog()` system call (see below), or using the `sysctl()` system call, or by echoing new values to `/proc/sys/kernel/printk`.

```
# cat /proc/sys/kernel/printk
1       4       1       7
# echo "2 4 3 7" > /proc/sys/kernel/printk
# cat /proc/sys/kernel/printk
2       4       3       7
#
```

The `sysctl()` version goes as follows:

```
% cat > printk_sysctl.c << EOF
#include <stdio.h>
#include <sys/sysctl.h>

#define SIZE(a) (sizeof(a)/sizeof((a)[0]))

int name[] = { CTL_KERN, KERN_PRINTK };
int printk_params[4];
int new_params[4];

int main(int argc, char **argv) {
        int paramlth = sizeof(printk_params);

        if (argc == 1) {
                /* report */
                if (sysctl(name, SIZE(name),
                           printk_params, &paramlth, 0, 0)) {
                        perror("sysctl");
                        exit(1);
                }
                printf("got %d bytes:\n", paramlth);
                printf("console_loglevel: %d\n", printk_params[0]);
                printf("default_message_loglevel: %d\n", printk_params[1]);
                printf("minimum_console_loglevel: %d\n", printk_params[2]);
                printf("default_console_loglevel: %d\n", printk_params[3]);
        } else if (argc == 5) {
                int i;

                for (i=0; i<4; i++)
                        new_params[i] = atoi(argv[i+1]);
                /* set */
                if (sysctl(name, SIZE(name),
                           0, 0, new_params, sizeof(new_params))) {
                        perror("sysctl");
                        exit(1);
                }
                printf("set new printk parameters\n");
        } else {
                fprintf(stderr, "Call: %s [N N N N]\n", argv[0]);
```

```
                exit(1);
        }
        return 0;
}
EOF
% cc -o printk_sysctl printk_sysctl.c
% ./printk_sysctl
got 16 bytes:
console_loglevel: 2
default_message_loglevel: 4
minimum_console_loglevel: 3
default_console_loglevel: 7
% ./printk_sysctl 1 4 1 7
sysctl: Operation not permitted
% su
# ./printk_sysctl 1 4 1 7
set new printk parameters
```

Ringbuffer

Messages are printed to a ring buffer, so that later messages overwrite earlier ones. The size of this buffer can be set at compile time. Long ago it was 4096. Today 16384 is the default, on some architectures up to 131072.

This ringbuffer is available for reading via `/proc/kmsg`. However, this is a read-once interface: data disappears as soon as it is read. There is also the `dmesg` command, that reads nondestructively.

Console

The console(s) printed to are determined at boot time (but see below). A kernel boot parameter `console=` gives a virtual or serial console to be printed to, and one may have several such lines. The format is `console=device,option`, e.g., `console=tty0`: print to the foreground VT (this is the default), or `console=tty12`: print to `/dev/tty12`, or `console=ttyS1,2400`: print to the serial line `/dev/ttyS1` at 2400 baud, or `console=lp0`: print to the printer. The option for a serial line is a number specifying baud rate, followed by a letter (one of `n`,`o`,`e`) specifying parity (none, odd, even), followed by a number of bits. Default is 9600n8.

In order to use a serial console, serial port support (CONFIG_SERIAL_8250) and console on serial port (CONFIG_SERIAL_8250_CONSOLE) must be enabled in the kernel config.

The file `/dev/console` refers to the last console mentioned, or to `/dev/tty0` by default.

A 2400 baud serial console.

Instead of an old terminal, one may also use another PC as serial console. Connect both machines with a null-modem cable. Run some communications program on the Serial Console machine, for example kermit.

```
(SC) # kermit
C-kermit> set port /dev/ttyS1
C-kermit> set speed 38400
C-kermit> set carrier-watch off
C-kermit> connect
...
```

These commands can be put into `.kermrc`. Kermit will become unhappy when programs on the other side fiddle with the serial line. The typical result is "Communications disconnect". Instead of convincing all programs that touch serial lines (such as the `setserial` boot script, and the X server) to leave the line used alone, one can simply rename the device:

```
# mv /dev/ttyS0 /dev/ttyS0-sc
```

Netconsole

When the kernel crashes during the boot, one can log the messages via the netconsole to a different machine. On the other machine, run

```
% netcat -u -l 5555
```

(this listens to incoming UDP packets on the specified port and prints the contents on stdout). On the booting machine make sure the kernel was compiled with `CONFIG_NETCONSOLE=y` and give kernel command line parameters

```
netconsole=4444@192.168.1.40/eth2,5555@192.168.1.50/01:23:45:67:89:AB
```

that is, the source port, source IP address, source ethernet device, destination port, destination IP address, destination MAC address. The port numbers are arbitrary.

Redirecting kernel VT output

If the console is a VT, then it is possible to redirect kernel messages to a different, already existing, VT by use of the `TIOCLINUX` ioctl:

```
/* Send kernel messages to a given console */
/* (Abbreviated version of setlogcons.c) */
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>

int main(int argc, char **argv){
        int cons;
        struct { char fn, subarg; } arg;

        if (argc == 2)
                cons = atoi(argv[1]);
        else
                cons = 0;       /* current console */

        arg.fn = 11;            /* redirect kernel messages */
        arg.subarg = cons;      /* to specified console */
        if (ioctl(0, TIOCLINUX, &arg)) {
                perror("TIOCLINUX");
                exit(1);
        }
        return 0;
}
```

Redirecting console output to a pseudotty

Using the `TIOCCONS` ioctl one can redirect console output to a pseudotty. This is what `xterm -C` and `xconsole` do. A small demo:

```
/*
 * compile with
 *  cc -o pseudocons pseudocons.c -lutil
 */
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
```

```
#include <unistd.h>
#include <string.h>
#include <termios.h>
#include <pty.h>
#include <sys/ioctl.h>

void
die(char *s) {
        perror(s);
        exit(1);
}

int main() {
        int masterfd, slavefd, fd;
        char c;

        if (openpty(&masterfd, &slavefd, NULL, NULL, NULL) < 0)
                die("openpty");
        printf("got master\n");

        if (ioctl(slavefd, TIOCCONS, 0)) {
                if (errno != EBUSY)
                        die("TIOCCONS");
                printf("trying to steal console\n");
                fd = open("/dev/tty0", O_WRONLY);
                if (fd < 0)
                        die("open /dev/tty0 fails");
                if (ioctl(fd, TIOCCONS, 0))
                        die("TIOCCONS tty0");
                if (ioctl(slavefd, TIOCCONS, 0))
                        die("TIOCCONS");
        }
        printf("got slave console\n");

        while (read(masterfd, &c, 1) == 1)
                printf("%c\n", c);

        return 0;
}
```

with application:

```
% cc -Wall -o pseudocons pseudocons.c -lutil
% ./pseudocons
got master
trying to steal console
TIOCCONS tty0: Operation not permitted
% su
# ./pseudocons
got master
trying to steal console
got slave console
H
o
e
r
a
.
.
```

where the output arose because of

```
# echo Hoera.. > /dev/console
```

done in another window.

syslog / klogctl

The `syslog()` system call (not to be confused with the library function of the same name) serves to set logging parameters and give access to the ringbuffer. The glibc interface is called `klogctl()`. The prototype is

```
int klogctl(int type, char *bufp, int len);
```

The first parameter specifies one of 10 possible subfunctions:

```
/*
 * Commands to do_syslog:
 *
 *      0 -- Close the log.  Currently a NOP.
 *      1 -- Open the log. Currently a NOP.
 *      2 -- Read from the log.
 *      3 -- Read all messages remaining in the ring buffer.
 *      4 -- Read and clear all messages remaining in the ring buffer
 *      5 -- Clear ring buffer.
 *      6 -- Disable printk's to console
 *      7 -- Enable printk's to console
 *      8 -- Set level of messages printed to console
 *      9 -- Return number of unread characters in the log buffer
 */
```

For details, see `syslog(2)`. A common setup is to have a daemon `klogd(8)` read the kernel ringbuffer, and feed the messages to `syslogd(8)`. What happens next depends on the `syslogd` configuration, see `syslog.conf(5)`. In the file `/etc/syslog.conf` one can specify what mailboxes and consoles and files and internet connections system log messages should be sent to. This is much more than just kernel messages - many daemons report their things via the syslog mechanism. For example, lines

```
kern.warn;*.err;authpriv.none    /dev/tty10
*.*;mail.none;news.none         -/var/log/messages
```

cause messages in certain categories to be sent to `/dev/tty10` and to the file `/var/log/messages`. The leading `-` says that it is not necessary to `sync()` after every write.

Under X one can start `xconsole`. It captures messages written to `/dev/console` (or some other specified file) and displays them in an X window. A similar effect may be achieved by `xterm -C`. You may have to be root to get permission to open `/dev/console`. (Or you can make some login script invoke `GiveConsole`.) The mechanism is provided by the TIOCCONS ioctl discussed above.

Deadlock

Inserting calls to `printk()` is safe on most places in the kernel. Of course one should not put such calls in the code that handles `printk()` output, or an infinite amount of output will be generated, or, in case such code acquires locks, a deadlock will result. The variable `oops_in_progress` can be set to 1 to break certain locks, and to make sure klogd is not woken up.

Very early printk

If the kernel hangs or crashes early in the boot sequence, before the console has been initialized, one can trace what is happening by writing messages "by hand" to video memory. See videochar.txt.

2.9 Keyboard interface to the kernel

It is possible to directly ask kernel info from the keyboard (when at a virtual console), also when no process is reading it. Sometimes it is possible in this way to collect some information when most of the kernel has crashed.

Show Registers

First of all Show_Registers (AltGr-ScrollLock), where AltGr is the right Alt key. It produces a register dump.

```
Pid: 0, comm:              swapper
EIP: 0060:[<c01088f4>] CPU: 0
EIP is at default_idle+0x24/0x30
 EFLAGS: 00000246    Not tainted
EAX: 00000000 EBX: c01088d0 ECX: 00000175 EDX: c12bef50
ESI: c0532000 EDI: c01088d0 EBP: 0008e000 DS: 007b ES: 007b
CR0: 8005003b CR2: 084b496c CR3: 0f913000 CR4: 00000290
Call Trace:
 [<c0108972>] cpu_idle+0x32/0x50
 [<c0105000>] _stext+0x0/0x20
```

This dump is written using `printk` hence also goes to wherever printk output goes (e.g. to `/var/log/messages`).

Show State

Next is Show_State (Ctrl-ScrollLock). It produces a very long output, listing for every process in the system some data and a stack trace.

```
emacs         S 00000082 4195332920  4988   4987                     (NOTLB)
Call Trace:
 [<c032a883>] normal_poll+0x113/0x12b
 [<c011db3f>] schedule_timeout+0x7f/0xa0
 [<c011dab0>] process_timeout+0x0/0x10
 [<c014e3cf>] do_select+0x1ef/0x230
 [<c014e060>] __pollwait+0x0/0xa0
 [<c014e759>] sys_select+0x319/0x470
 [<c0109c85>] restore_sigcontext+0x115/0x140
 [<c0119a13>] sys_gettimeofday+0x43/0xa0
 [<c010a72b>] syscall_call+0x7/0xb
```

Since the result probably scrolls off the screen, one may have to use the `dmesg` command, or inspect the syslog, in order to read the output.

Show Memory

Finally Show_Memory (Shift-ScrollLock). It shows the memory situation.

```
Mem-info:
DMA per-cpu:
cpu 0 hot: low 2, high 6, batch 1
cpu 0 cold: low 0, high 2, batch 1
Normal per-cpu:
cpu 0 hot: low 28, high 84, batch 14
cpu 0 cold: low 0, high 28, batch 14
HighMem per-cpu: empty

Free pages:       15440kB (0kB HighMem)
Active:31177 inactive:22639 dirty:6 writeback:0 free:3860
DMA free:6000kB min:128kB low:256kB high:384kB active:2316kB inactive:3008kB
Normal free:9440kB min:1020kB low:2040kB high:3060kB active:122392kB inactive:87548kB
HighMem free:0kB min:0kB low:0kB high:0kB active:0kB inactive:0kB
DMA: 134*4kB 71*8kB 40*16kB 27*32kB 17*64kB 2*128kB 0*256kB 0*512kB 0*1024kB 1*2048kB 0*4096kB = 600
Normal: 0*4kB 2*8kB 1*16kB 0*32kB 51*64kB 26*128kB 5*256kB 1*512kB 1*1024kB 0*2048kB 0*4096kB = 9440
HighMem: empty
Swap cache: add 0, delete 0, find 0/0, race 0+0
Free swap:      281044kB
65520 pages of RAM
0 pages of HIGHMEM
1891 reserved pages
31170 pages shared
0 pages swap cached
```

Other commands

Also a few virtual console housekeeping commands are implemented via this same mechanism. There are keys for scroll up (Shift-PgUp) and scroll down (Shift-PgDn). Keys to change virtual console: Decr_Console (Alt-LArrow), Incr_Console (Alt-RArrow), Last_Console (unbound).

Finally there are three commands: Boot (Ctrl-Alt-Del), KeyboardSignal (Alt-UpArrow), SAK (unbound).

Key bindings can be changed using `loadkeys`.

Magic SysRequest

When so configured, there is one more keyboard mechanism. Build the kernel with CONFIG_MAGIC_SYSRQ enabled. In some kernel versions you then have an active Magic SysRequest key, but in other kernels you first have to do

```
echo 1 > /proc/sys/kernel/sysrq
```

to activate this key. A command is given by pressing three keys simultaneously: Alt-SysRq-X, where X is the command letter. (On a serial console one gives a Break, followed by the command letter.)

Since Linux 2.4.21/2.5.66 it is also possible to do

```
echo X > /proc/sysrq-trigger
```

with the same effect as the keystroke.

Command letters

- r Set kbd mode to XLATE (not RAW)
- k SAK (Secure Attention Key)
- b Immediate reboot, no umount, no sync
- o Shut Off system
- s Sync
- u Remount all filesystems read-only
- p Print registers on console, like Show_Registers above.
- t Print tasks on console, like Show_State above.
- m Print memory info on console, like Show_Memory above.
- 0-9 Set console loglevel
- e Send every process SIGTERM
- i Send every process SIGKILL

One should make sure that this facility is always switched off in a production environment.

2.10 Profiling the kernel

There are several facilities to see where the kernel spends its resources. A simple one is the profiling function, that stores the current EIP (instruction pointer) at each clock tick.

Boot the kernel with command line option `profile=2` (or some other number instead of 2). This will cause a file `/proc/profile` to be created. The number given after `profile=` is the number of positions EIP is shifted right when profiling. So a large number gives a coarse profile. The counters are reset by writing to `/proc/profile`. The utility `readprofile` will output statistics for you. It does not sort - you have to invoke `sort` explicitly. But given a memory map it will translate addresses to kernel symbols.

See `kernel/profile.c` and `fs/proc/proc_misc.c` and `readprofile(1)`.

For example:

```
# echo > /proc/profile
...
# readprofile -m System.map-2.5.59 | sort -nr | head -2
510502 total                                      0.1534
508548 default_idle                           10594.7500
```

The first column gives the number of timer ticks. The last column gives the number of ticks divided by the size of the function.

The command `readprofile -r` is equivalent to `echo > /proc/profile`.

Oprofile

A more advanced mechanism is given by *oprofile*.

Prepare kernel

Build a kernel (2.5.43 or later) with CONFIG_PROFILING=y and CONFIG_OPROFILE=y. Now the kernel knows about the `oprofilefs` virtual filesystem. The utilities mentioned below will mount it on `/dev/oprofile`. It is a good idea to add `idle=poll` to the kernel command line; this will make sure time spent in the idle thread is properly accounted for.

Install oprofile

Get the oprofile utility from http://oprofile.sourceforge.net/. Configure with `./configure --with-kernel-support`, then `make` and (as root) `make install`. This yields binaries and a man page:

```
% ls /usr/local/bin
op_dump  op_merge    op_start  op_time       opcontrol    oprofiled
op_help  op_session  op_stop   op_to_source  oprof_start  oprofpp
% ls /usr/local/share/oprofile
stl.pat
% ls /usr/local/share/doc/oprofile
oprofile.html
% ls /usr/local/man/man1
op_help.1   op_time.1       opcontrol.1  oprofiled.1
op_merge.1  op_to_source.1  oprofile.1   oprofpp.1
%
```

All man pages are links to the `oprofile` man page. Thus, there are two sources of information:

```
% man oprofile
% mozilla /usr/local/share/doc/oprofile/oprofile.html
```

Setup oprofile

Make sure you are root and your PATH contains `/usr/local/bin`. The first action required is invoking `opcontrol --setup ...`. This will create a setup file `/root/.oprofile/daemonrc`. You need the big kernel toplevel `vmlinux` file (not `bzImage`, and not the smaller `arch/i386/boot/compressed/vmlinux`).

```
# opcontrol --setup --vmlinux=/foo/vmlinux
# cat /root/.oprofile/daemonrc
IGNORE_MYSELF=0
SEPARATE_LIB_SAMPLES=0
SEPARATE_KERNEL_SAMPLES=0
VMLINUX=/foo/vmlinux
BUF_SIZE=0
one_enabled=1
#
```

The precise setup call needed depends on your hardware. The above example is for an old Pentium without hardware performance counters. For a P3, use

```
opcontrol --setup --vmlinux=/foo/vmlinux --ctr0-event=CPU_CLK_UNHALTED --ctr0-count=100000
```

For a P4, use

```
opcontrol --setup --vmlinux=/foo/vmlinux --ctr0-event=GLOBAL_POWER_EVENTS --ctr0-unit-mask=1 --ctr0-
```

For an Athlon or x86-64, use

```
opcontrol --setup --vmlinux=/foo/vmlinux --ctr0-event=RETIRED_INSNS --ctr0-count=100000
```

There are many other possible setup options. The command `op_help` will list them once you have done the setup :-). (See also the sourceforge site for [Intel P6/PII/PIII](), [Intel P4]() and [AMD]() events.)

The general idea is that one specifies a certain type of event (such as CPU_CLK_UNHALTED, a CPU clock cycle), and a count (like the 100000 above). Now once every count occurrences of this event the value of EIP is recorded. Pick the value of count suitably: with a 400 MHz machine a count of 100000 for CPU_CLK_UNHALTED means 4000 interrupts/second. Too small a count and the machine dies an interrupt death. Too large a count and the profile will be very coarse.

Use of oprofile

First start the oprofile daemon.

```
# opcontrol --start-daemon
Using log file /var/lib/oprofile/oprofiled.log
Daemon started.
# ps ax | grep oprofile
... /usr/local/bin/oprofiled ...
```

Next clear out old profiling data.

```
# opcontrol --reset
```

Next start measuring, do whatever should be measured, and stop measuring.

```
# opcontrol --start
Profiler running.
# do_something
# opcontrol --stop
Stopping profiling.
#
```

One now has profiling data below `/var/lib/oprofile/`. See below for what to do with the data.

When no more profiling is needed, kill the daemon:

```
# opcontrol --shutdown
Stopping profiling.
Killing daemon.
#
```

To clean up all data generated by oprofile (after generating any desired output):

```
# rm -r /var/lib/oprofile
```

A partial cleanup is done by the `opcontrol --reset` mentioned above.

Output

To get a printout of the data for `/bin/foo`, use, e.g. `oprofpp -l -i /bin/foo`. The output will be boring, unless `/bin/foo` was not stripped after compilation, so that it contains symbol information.

The programs and libraries that were invoked (and hence are suitable arguments for `oprofpp -l -i ...`) and the numbers of ticks spent in each are given by the command `op_time`. With the `-l` option the output will be split according to symbol.

```
# op_time | tail -3
3134       9.0531  0.0000 /lib/i686/libc-2.2.4.so
4813      13.9032  0.0000 /usr/bin/find
26212     75.7178  0.0000 /foo/vmlinux
# op_time -l | tail -6
c012c7c0 703      2.04509     kmem_cache_alloc        /foo/vmlinux
c02146c0 786      2.28655     __copy_to_user_ll       /foo/vmlinux
c0169b70 809      2.35345     ext3_readdir            /foo/vmlinux
c01476f0 854      2.48436     link_path_walk          /foo/vmlinux
c016fcd0 1446     4.20655     ext3_find_entry         /foo/vmlinux
00000000 4591     13.3556     (no symbol)             /usr/bin/find
#
```

To get statistics for the kernel only, use the `vmlinux` name specified.

```
# oprofpp -l -i /foo/vmlinux | tail
c012ca30 488      1.86174     kmem_cache_free
c010e280 496      1.89226     mask_and_ack_8259A
c010a61a 506      1.93041     restore_all
c0119220 603      2.30047     do_softirq
c0110b30 663      2.52938     delay_tsc
c012c7c0 703      2.68198     kmem_cache_alloc
c02146c0 786      2.99863     __copy_to_user_ll
c0169b70 809      3.08637     ext3_readdir
c01476f0 854      3.25805     link_path_walk
c016fcd0 1446     5.51656     ext3_find_entry
#
```

One can get disassembly or annotated source with indication on where the counts occurred.

```
# op_to_source -a -i /foo/vmlinux
...
/* 424 1.618% */
c0169ac0 <ext3_check_dir_entry>:
 /* 6 0.02289% */
c0169ac0:       push   %edi
 /* 56 0.2136% */
c0169ac1:       push   %esi
c0169ac2:       push   %ebx
c0169ac3:       mov    0x18(%esp,1),%esi
 /* 43 0.164% */
c0169ac7:       xor    %ebx,%ebx
c0169ac9:       mov    0x14(%esp,1),%edi
c0169acd:       movzwl 0x4(%esi),%ecx
 /* 23 0.08775% */
c0169ad1:       cmp    $0xb,%ecx
c0169ad4:       jg     c0169ae0 <ext3_check_dir_entry+0x20>
c0169ad6:       mov    $0xc032b160,%ebx
...

# op_to_source --source-dir=. --output-dir=/tmp -i /path/binary
# diff -u ./source.c /tmp
...
 int get_line(register FILE *f, int *length)
+/* get_line 24 54.55% */
...
+       /* 5 11.36% */
        c = Getc (f);
...
```

This profile with annotated source is available only for binaries compiled with `-g2` (and not stripped) so that they still contain source line numbers.

2.11 Debugging the kernel

There does exist a kernel debugger `kdb`, maintained as a patch on the kernel source. It can be obtained by anonymous ftp from ftp://oss.sgi.com. For an introduction, see ibm/developerworks.

There are patches for Linux 2.2 and 2.4. I have not seen them for 2.5 yet.

Early stages

Specifying the boot parameter `initcall_debug` causes the kernel to print the addresses of all initcalls it executes. This may allow one to pinpoint the guilty part when the kernel crashes at boot time. (Since 2.5.67.)

2.12 Submitting patches

Read `Documentation/SubmittingPatches` and perhaps `Documentation/SubmittingDrivers`. See also The perfect patch and Linux kernel patch submission format.

2.13 Talking about the kernel

The main forum is the mailing list known as lk or lkml or linux-kernel with submission address `linux-kernel@vger.kernel.org`. Subscribe to the mailing list by sending the message

```
subscribe linux-kernel
```

to `Majordomo@vger.kernel.org`. Archives exist. There are many more specialized lists, such as `linux-net` or `linux-fsdevel`. There is a "kernelnewbies" IRC channel, and mailing list, and website, with a lot of useful information. Lots of other places exist.

Idiom

Brown paper bag bug - A bug one is deeply ashamed of.

Before the release of 2.2.0:

```
From: Linus Torvalds <torvalds@transmeta.com>
Subject: (fwd) 2.2.0-final
Date: 1999/01/22
...
In short, before you post a bug-report about 2.2.0-final, I'd like you to
have the following simple guidelines:

 "Is this something Linus would be embarrassed enough about that he would
  wear a brown paper bag over his head for a month?"

and

 "Is this something that normal people would ever really care deeply
  about?"

If the answer to either question is "probably not", then please consider
just politely discussing it as a curiosity on the kernel mailing lists
rather than even sending email about it to me
...
```

After the release of 2.2.0:

```
From: Linus Torvalds <torvalds@transmeta.com>
Subject: Linux-2.2.1 - the Brown Paper Bag release
Date: 1999/01/28
```

```
...
The subject says it all. We did have a few paper-bag-inducing bugs in
2.2.0, so there's a 2.2.1 out there now, just a few days after 2.2.0.

Oh, well. These things happen,

                Linus
```

Next Previous Contents