KERNEL LINUX

Uncle Linus Wants You!

Introduction

What is a kernel?

The kernel is the part of the system that:

- manages the hardware
- allocates resources eg. memory pages or CPU cycles
- is responsible for the file system and network communication
- provides an abstraction layer for the applications: the userland

The Linux kernel

Main features

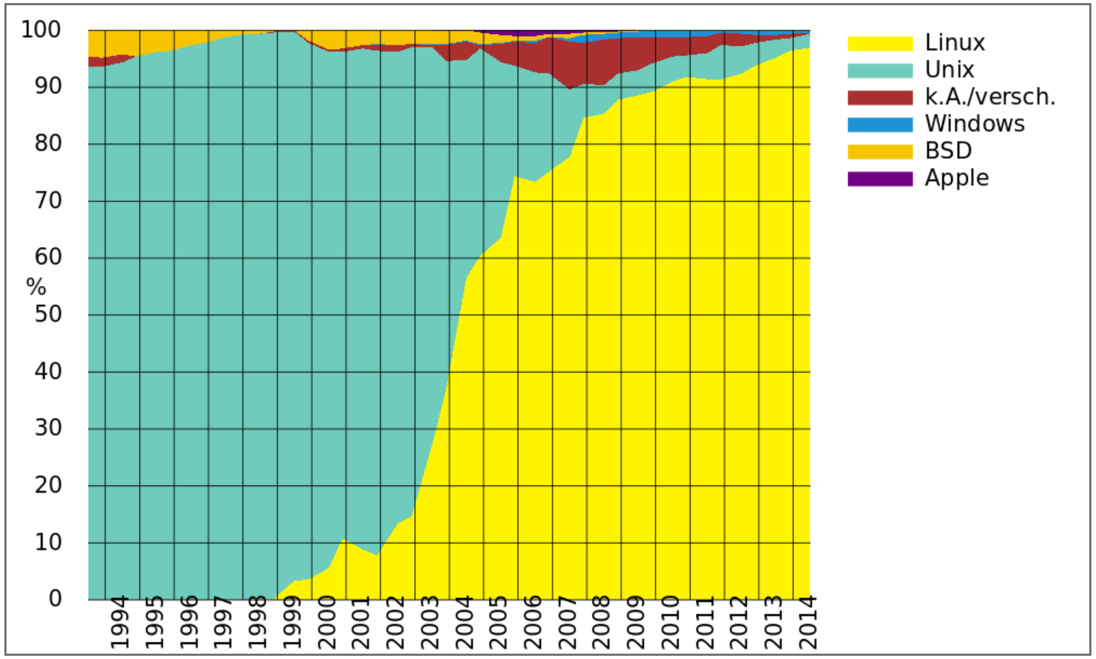
- Portable
- Versatile
- Stable
- Mature
- Secure
- Robust

Who is behind Linux?

- An open source community
- Around 1200 developers for one release
- Around 200 maintainers
- Around 80% of the changes are sponsored
- Linus Torvalds makes the official releases

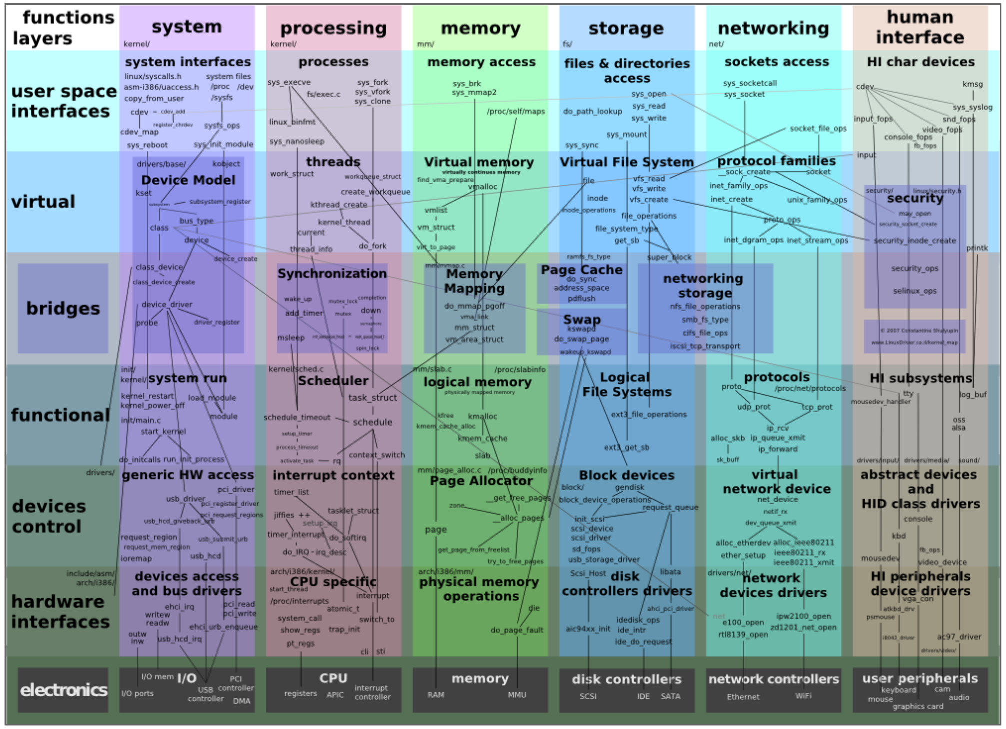
Usage and hardware support

Linux supports more computer architecture than any other OS.



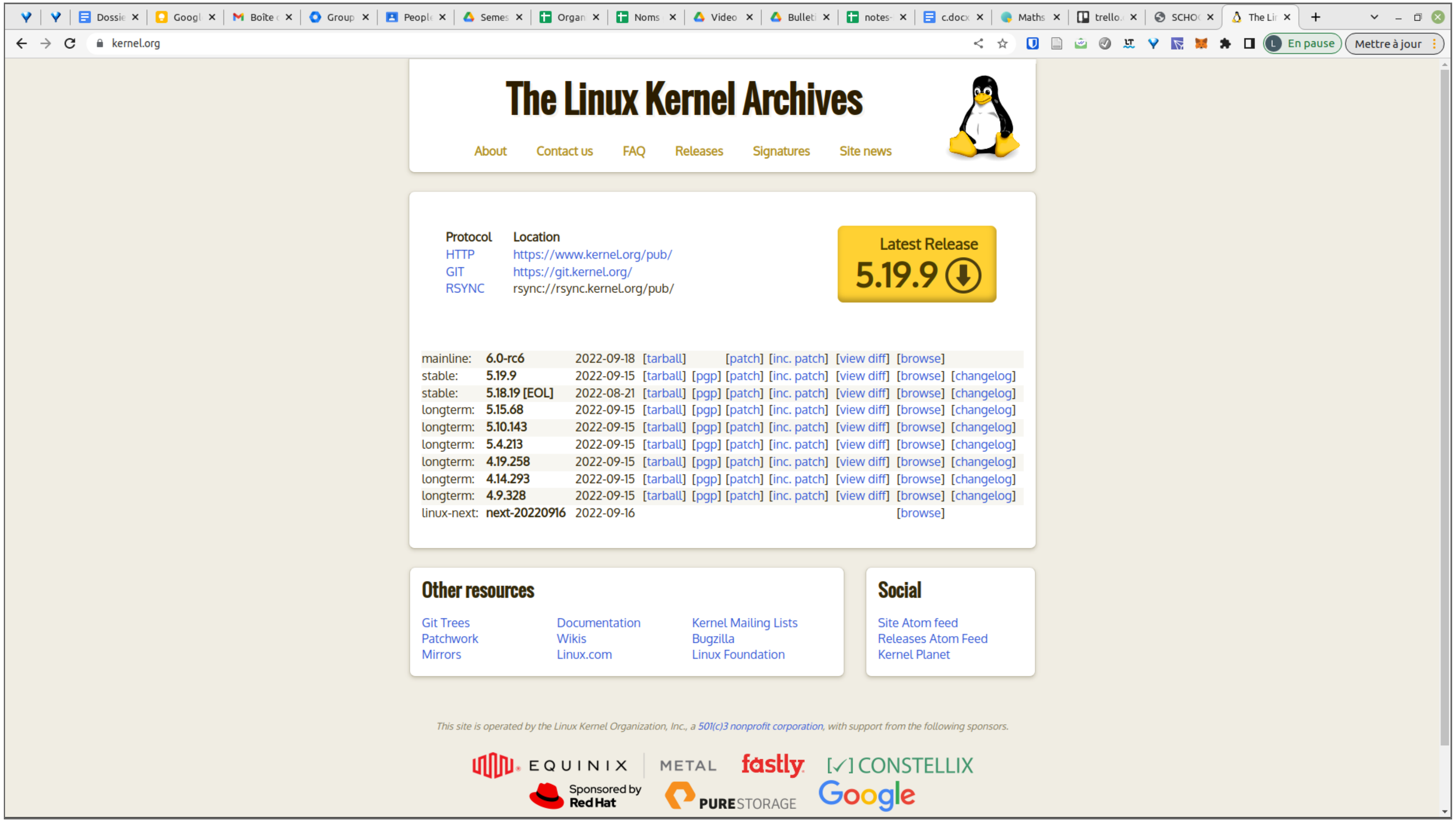
Meeting Linux

Overview



functions
layers
system
processing
memory
storage
networking
human interface
user space interfaces
virtual
bridges
functional
devices control
hardware interfaces
electronics
system interfaces
processes
memory access
files & directories access
sockets access
HI char devices
Device Model
threads
Virtual memory
Virtual File System
protocol families
security
Synchronization
Memory Mapping
Page Cache
Swap
networking storage
system run
Scheduler
logical memory
Logical File Systems
protocols
HI subsystems
generic HW access
interrupt context
Page Allocator
Block devices
virtual network device
abstract devices and HID class drivers
devices access and bus drivers
CPU specific
physical memory operations
disk controllers drivers
network devices drivers
HI peripherals device drivers
I/O
CPU
memory
disk controllers
network controllers
user peripherals

The source


kernel.org
The Linux Kernel Archives
About
Contact us
FAQ
Releases
Signatures
Site news
Protocol Location
HTTP https://www.kernel.org/pub/
GIT https://git.kernel.org/
RSYNC rsync://rsync.kernel.org/pub/
Latest Release
5.19.9
mainline: 6.0-rc6 2022-09-18 [tarball] [patch] [inc. patch] [view diff] [browse]
stable: 5.19.9 2022-09-15 [tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]
stable: 5.18.19 [EOL] 2022-08-21 [tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]
longterm: 5.15.68 2022-09-15 [tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]
longterm: 5.10.143 2022-09-15 [tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]
longterm: 5.4.213 2022-09-15 [tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]
longterm: 4.19.258 2022-09-15 [tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]
longterm: 4.14.293 2022-09-15 [tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]
longterm: 4.9.328 2022-09-15 [tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]
linux-next: next-20220916 2022-09-16 [browse]
Other resources
Git Trees
Patchwork
Mirrors
Documentation
Wikis
Linux.com
Kernel Mailing Lists
Bugzilla
Linux Foundation
Social
Site Atom feed
Releases Atom Feed
Kernel Planet
This site is operated by the Linux Kernel Organization, Inc., a 501(c)3 nonprofit corporation, with support from the following sponsors.

The kernel source tree (1/3)

- *arch/* Architecture-specific code
- *block/* Block I/O layer
- *crypto/* Cryptographic API
- *Documentation/* Kernel source documentation
- *drivers/* Device drivers (except sound ones)
- *firmware/* Device firmware needed for some devices
- *fs/* Filesystems infrastructure
- ***include/* Kernel headers**
  - *include/linux/* Core kernel headers
  - *include/uapi/* User space API headers
- *init/* Kernel initialization
- *ipc/* System V InterProcess Communication (sem, shm, msgqueue)

The kernel source tree (2/3)

- *kernel/* Linux kernel core
- *lib/* Routine library (lists, trees, string, etc.)
- *mm/* Memory management
- *net/* Network support code (not drivers!)
- *samples/* Sample code (trace, kobject, krpobes)
- *scripts/* Scripts for internal or external use
- *security/* Security frameworks
- *sound/* Sound code and drivers
- *tools/* User space tools
- *usr/* Generate the initramfs
- *virt/* Virtualization infrastructure (kvm)

The kernel source tree (3/3)

- *COPYING* The kernel licence (GPLv2)
- *CREDITS* The people who have contributed to the kernel
- *Kbuild* Kernel build system
- *Kconfig* Description of configuration parameters
- *MAINTAINERS* Maintainer of subsystems and drivers
- *Makefile* Base kernel Makefile
- *README* Overview of the kernel
- *REPORTING-BUGS* Instructions for reporting bugs

Development process

- Each file has a set of maintainers.
- They are responsible for triaging bugs, reviewing patches and directing changes.
- Patches are sent to mailing-lists for review.
- Once approved, they are sent to the maintainer of the subsystem.
- Subsystem maintainers review them and place them in a special branch.
- They send this branch to Linus Torvalds, that will merge it to his branch.

Release cycle

- The merge window is open (~2 weeks).
- Release candidates (*-rc*) are published.
- No new feature is added, only bug fixes (~6-10 weeks).
- The final release is tagged by Linus Torvalds.

Configuring the kernel

- The kernel is a single file, resulting of the compilation process.
- Compile-time options (*-D* flags) can be used to select which features are compiled-in and their settings.
- However, after the boot process, it can load module from the filesystem at runtime to extend its features. Each module is a single file.

  Configuring the kernel is:
    - Choosing what’s going into the main file, and what will be built as modules.
-
    - Setting various options.

Kernel options

Options have the form *CONFIG_FEATURE* and a type, eg.

- *CONFIG_MODULES* boolean (true/false)
- *CONFIG_INITRAMFS_ROOT_UID* integer
- *CONFIG_INITRAMFS_SOURCE* string
- *CONFIG_MAGIC_SYSRQ_DEFAULT_ENABLE* hex
- *CONFIG_BTRFS_FS* tristate (true/module/false)

- Options can depend on others.

Two types of dependencies:

- A *select* B, enabling A enables B
- A *depends on* B, A is not visible until B is enabled

Kconfig

```
config BTRFS_FS
    tristate "Btrfs filesystem support"
    select CRYPTO
    select CRYPTO_CRC32C
    select ZLIB_INFLATE
    select ZLIB_DEFLATE
    select LZO_COMPRESS
    select LZO_DECOMPRESS
    select RAID6_PQ
    select XOR_BLOCKS

    help
      Btrfs is a general purpose copy-on-write filesystem
      with extents, writable snapshotting, support for
      multiple devices and many more features focused on
      fault tolerance, repair and easy administration.
```

[....]

Config files

.config

Simple text files, *key*=*value* format.

Default *.config* files

- *make defconfig*: new config with default from $ARCH supplied defconfig
- *make i386_defconfig*: request defconfig from a platform

Editing *.config* files

- *make config*: text based
- *make menuconfig*, *make nconfig*: menu/ncurses interface
- *make xconfig*, *make gconfig*: graphical interface
- *make oldconfig*: upgrade *.config* with options from the new release

Vendor kernels

Linux distributions typically enable a lot of kernel features and drivers, most of them are compiled as modules.
Read the config file of your kernel
Require *CONFIG_IKCONFIG_PROC=y*:

```
$ zcat /proc/config.gz
CONFIG_64BIT=y
CONFIG_X86_64=y
CONFIG_X86=y
CONFIG_INSTRUCTION_DECODER=y
CONFIG_OUTPUT_FORMAT="elf64-x86-64"
CONFIG_ARCH_DEFCONFIG="arch/x86/configs/x86_64_defconfig"
...
```

Build

- Only interact with the top-directory Makefile

```
$ make menuconfig # Edit .config
$ make # Build the kernel and modules
```

Produces:

- …
- *vmlinux*: ELF object of the kernel, cannot be booted
- *arch/x86/boot/bzImage*: bootable compressed kernel image
- ***/*.ko*: Modules

More targets

- *make help*: list all available targets
- *make modules*: build/rebuild modules
- *make headers_install*: “install” headers in the local *usr/*
- ***make modules_install*: install to */lib/modules/KVER***
  - *INSTALL_MOD_PATH=dir/* to select the directory

Interacting with Linux

The command line: kernel parameters

- **It is a string for runtime configuration:**
  - no recompilation
  - can be builtin, using the *CONFIG_CMDLINE* option
  - can be used to pass arguments to the *init* program
- **Many kernel options, examples:**
  - *root=/dev/sda1* the root filesystem
  - *console=ttyS0* where to write kernel messages
  - *debug*, *loglevel=7* kernel verbosity
  - *usbcore.blinkenlights=true* also available for modules
- More documentation: *Documentation/kernel-parameters.txt*

Syscalls

- The system call is the fundamental interface between an application and the Linux kernel.
- Typically accessed via wrapper functions of the libc. The name of the wrapper function is usually the same as the name of the system call that it invokes.
- More than 320 on x86, some are architecture-specific, but most are common.
- One of the key component of Linux’ maturity.

More details in the next lesson!

The kernel log

- **The kernel stores msgs in a circular log buffer:**
  - */proc/ksmg* for raw output
  - */dev/kmsg* for structured message reading
- By default the kernel log is outputted on the console, see the *console*= kernel parameter.
- The *dmesg* tool (diagnostic message) can be used to read those messages.
- When using *systemd*, *journalctl -k* displays the kernel log.

Loadable kernel modules (LKM)

LOADING

- Require *CONFIG_MODULES=y*
- *insmod*: Plug a *.ko* file into the kernel.
- *modprobe*: Load a module (no *.ko*) and its dependencies.

  Both handle module parameters:
-
    - *$ insmod ./snd-intel8x0m.ko index=-2*

Unloading

- Require *CONFIG_MODULE_UNLOAD=y*
- *rmmod*: Unplug the module.
- *modprobe -r*: Also remove unused dependencies.

INFO

- *lsmod*: Show the status of modules in the Linux Kernel.
- *modinfo*: Show information about a Linux Kernel module.

Pseudo file systems

There are many pseudo file systems in Linux, here are some of them:

- *proc*: (*/proc*) processes info, raw stuff, etc.
- *sysfs*: (*/sys*) structured information about various kernel subsystems, tied to *kobjects*
- *devtmpfs*: (*/dev*) kernel populated devices nodes

More details in the next lesson!

Writing code for Linux

Essential developer tools

- The C language (ISO C99 and gnu extensions)
- GNU Make
- Git

Useful developer tools

- *cscope*: Browse source code
- LXR: Linux Cross Reference
- *scripts/**

Cscope

- **http://cscope.sourceforge.net/**
- Built in VIM and Emacs!

  Search:

    - Symbol definition
    - Symbol usage
    - Function callers/callee
    - Text
- - …

  You must generate the database.
- - Use *make cscope* to get the database of your architecture.

LXR (**http://lxr.free-electrons.com**)

LXR (**http://lxr.free-electrons.com**)

```
static int __ref kernel_init(void *unused)
{
        int ret;

        kernel_init_freeable();
        /* need to finish all async __init code before freeing the memory */
        async_synchronize_full();
        free_initmem();
        mark_rodata_ro();
        system_state = SYSTEM_RUNNING;
        numa_default_policy();

        flush_delayed_fput();

        if (ramdisk_execute_command) {
                ret = run_init_process(ramdisk_execute_command);
                if (!ret)
                        return 0;
                pr_err("Failed to execute %s (error %d)\n",
                       ramdisk_execute_command, ret);
        }

        /*
         * We try each of these until one succeeds.
         *
         * The Bourne shell can be used instead of init if we are
         * trying to recover a really broken machine.
         */
        if (execute_command) {
                ret = run_init_process(execute_command);
                if (!ret)
                        return 0;
                pr_err("Failed to execute %s (error %d).  Attempting defaults...\n",
                        execute_command, ret);
        }
```

Coding style

> 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.
>
> – Documentation/CodingStyle

- Indentation uses tabs.
- Tabs are 8 spaces.
- No more than 80 chars per line (more a guideline than a hard rule).

Coding style

```
int foo(void)
{
    return 0;
}
```

```
if (cond) {
    do_foo();
    do_bar();
}
```

```
if (cond) {
    do_foo();
    do_bar();
} else {
```

```
    do_baz();
}
```

Coding style

```
if (cond)
    do_foo();
else
    do_baz();
```

```
int get_bar(struct bar *p)
{
    int r;

    r = kmalloc(sizeof(*p), GFP_KERNEL);
    if (!r)
        return -ENOMEM;

    return 0;
}
```

Kernel space

- No access to the usual libc.
- No unbreakable memory protection.
- No floating-point operations.
- Fixed-size stack.
- Preemptive and Symetric Multi-Processors (SMP) capable. Synchronization and concurrency are major concerns.

Your first module

Where to put-it?

Two alternatives:

- **Inside the official kernel tree:**
  - Integrated in the kernel repository
  - Can be built statically
  - Supported by the community: debug and update
- **Out of tree:**
  - In a directory outside the kernel source
  - Needs to be built separately

Hello Module

```c
/* hello.c */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/printk.h>

static __init int hello_init(void)
{
    pr_info("Load the module!\n");
    return 0;
}

static __exit void hello_exit(void)
{
    pr_info("Unload the module.\n");
}
```

```
module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Hello module");
MODULE_AUTHOR("Lionel Auroux");
```

Explanations

Macros

- __*init*: The code is discarded after initialization (both for static and module compilation).
- __*exit*: The code is discarded if the module is built statically or if module unloading is disabled.

Module setup/cleanup

```
int mod_init(void);
void mod_exit(void);
```

Explanations

Module metadata

- Retrievable using *modinfo*.
  ***MODULE_LICENSE* is important:**
    - Non-free software modules have no access to GPL exported symbols.
-
    - Non-free modules set the proprietary taint flag on the kernel.
- Also useful: *MODULE_VERSION*, *MODULE_INFO*, *MODULE_SOFTDEP*
- More information: *include/linux/module.h*

printk()

- *printk()* is the kernel space version of *printf*.

  Multiple logging levels:

  - *pr_emerg(), pr_alert(), pr_crit(), pr_err(), pr_warn(),*
- *pr_notice(), pr_info(), pr_debug()*
- Modern code uses *pr_xxx*.
- *pr_devel()* while you develop your code.
- *pr_cont()* for continuing lines with no newline (*n*).
- Formats are described in *Documentation/printk-formats.txt*.

pr_fmt

You define the *pr_fmt* macro to set the default format to all your *pr_xxx* calls.

```
#define pr_fmt(fmt) KBUILD_MODNAME ": " fmt
#include <linux/printk.h>
...
    pr_devel("Test.\n");
```

Compiling your module

```
ifeq ($(KERNELRELEASE),)
# Assume the source tree is where the running kernel was built
# You should set KERNELDIR in the environment if it's elsewhere
KERNELDIR ?= /lib/modules/$(shell uname -r)/build

# The current directory is passed to sub-makes as argument
PWD := $(shell pwd)

modules:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
modules_install:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) modules_install
clean:
	rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versions

.PHONY: modules modules_install clean
```

```
else
# called from kernel build system: just declare our module
obj-m := hello.o
endif
```

Dynamic debug

- Enabled if your kernel have *CONFIG_DYNAMIC_DEBUG=y*.
- Registers a control file: */sys/kernel/debug/dynamic_debug/control*.
- Display the current configuration by reading this file.
- Enable debug calls using:

```
# debug for the module `mymodule`
$ cd /sys/kernel/debug/dynamic_debug
$ echo 'module mymodule +p' > control
```

- Documentation in *Documentation/dynamic-debug-howto.txt*

Submitting patches

1. **Commit:**
   - *git commit –signoff*
2. **Make a patch:**
   - *git format-patch origin..master*
3. **Check your patch:**
   - *./scripts/checkpatch.pl 0001-my-commit.patch*
4. **Send email:**
   - *git send-email 0001-my-commit.patch*

More documentation

More documentation

IRC

- *irc.oftc.net #kernelnewbies*

Web

- **http://kernelnewbies.org**

Books

Warning: Linux moves fast, those books contain outdated information.

- Linux device drivers: **LDD3**
- Linux Kernel Development
- Linux System Programming
- Linux in a Nutshell
- Understanding the Linux Kernel
- The Linux Programming Interface

What is a syscall?

User space can issue requests to the kernel in order to access its resources or perfrom restricted operations.
You can think of a syscall as regular function call, but where the code being called is in the kernel.
Syscalls usages:

- Manipulating files and VFS: *open*, *read*, *write*, …
- System setup: *gettimeofday*, *swapon*, *shutdown*…
- Processes management: *clone*, *mmap*, …
- Manipulating devices: *ioctl*, *mount*, …
- Cryptography and security: *seccomp*, *getrandom*, …
- …

The syscall userland interfaces

In assembly

- On x86

```
mov eax, 1 ; exit
int 0x80 ; or sysenter
```

- Syscall number: *eax*
- Arguments: *ebx*, *ecx*, *edx*, *esi*, *edi*, *ebp*, then use the stack
- On x86_64

```
mov rax, 60 ; exit
syscall
```

- Syscall number: *rax*
- Arguments: *rdi*, *rsi*, *rdx*, *rcx*, *r8* and *r9*, no args on memory

`syscall(2)`

```
#include <unistd.h>
#include <sys/syscall.h> /* for __NR_xxx */

long syscall(long number, ...);
```

- Copies the arguments and syscall number to the registers.
- Traps to kernel code.
- Sets *errno* if the syscall returns an error.

Don’t panic!

- You will learn all about that in kernel from scratch!
- You almost never use direct calls to *syscall(2)*.
- Your libc provides wrappers for most of the syscalls you need.
- Linux also abstracts all thoses details in kernel code.
- For a list of the Linux system calls, see *syscalls(2)*.

`vdso(7)`

- Virtual Dynamically linked Shared Objects
- Small shared library (8k) that the kernel automatically maps into the address space of all user-space applications.
- Contains non priviledged code and data: *gettimeofday*, *time*, *clock_gettime*, … (arch-depedent)
- The ELF must be dynamically linked.

Why?

- Making system calls can be slow.
- On x86 32bit, *int 0x80* is expensive: goes through the full interrupt-handling paths in the processor’s microcode as well as in the kernel.
- Even if there is a dedicated instr (*syscall*), context switching must be done.

Context switch

A context is:

- The CPU registers (including the instruction pointer)
- **The state of a process (including threads):**
    - Memory state: stack, page tables, etc.
    - CPU state: registers, caches, etc.
    - Process scheduler state
    - …

vdso in action

```
$ cat time.c
int main(int ac, char **av) {
    printf("%d\n", time(0));
}
$ gcc time.c -o time -static
$ strace -e time ./time
time(NULL)                              = 1411171041
1411171041
+++ exited with 11 +++
$ gcc time.c -o time
$ ldd ./time
        linux-vdso.so.1 (0x00007fffe1735000)
        libc.so.6 => /usr/lib/libc.so.6 (0x00007fee5e753000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fee5eb01000)
$ strace -e time ./time
```

```
1411171118
+++ exited with 11 +++
```

Implementation

- Defining a syscall

Use the *SYSCALL_DEFINEx(syscall, …)* macros anywhere in Linux code.

These macros expands to:

- *SYSCALL_METADATA(syscall, …)* generate metadata used in the *FTRACE* tracing framework.
- *__SYSCALL_DEFINEx(syscall, …)* more function definition expansion.
- Ultimatly expand to: *asmlinkage long SyS_syscall(..)*
- *asmlinkage* means that arguments are on the stack.

Example

In *kernel/signal.c*:

```
SYSCALL_DEFINE0(pause)
{
        while (!signal_pending(current)) {
                current->state = TASK_INTERRUPTIBLE;
                schedule();
        }
        return -ERESTARTNOHAND;
}
```

```
#include <asm/current.h>
...
pr_debug("The process is \"%s\" (pid %i)\n",
     current->comm, current->pid);
```

```c
static inline int signal_pending(struct task_struct *p)
{
    return unlikely(
        test_tsk_thread_flag(p,TIF_SIGPENDING));
}
```

`schedule()`

Ask the scheduling subsystem to pick the next process to run.

- The syscalls tables

See *arch/x86/entry/syscalls/syscall_{32,64}.tbl*.

- *syscall_32.tbl*

```
# <number> <abi> <name> <entry point> <compat entry point>
0       i386    restart_syscall         sys_restart_syscall
1       i386    exit                    sys_exit
2       i386    fork                    sys_fork
stub32_fork
3       i386    read                    sys_read
4       i386    write                   sys_write
5       i386    open                    sys_open
compat_sys_open
6       i386    close                   sys_close
```

syscall_64.tbl

```
0       common  read                    sys_read
1       common  write                   sys_write
2       common  open                    sys_open
3       common  close                   sys_close
4       common  stat                    sys_newstat
5       common  fstat                   sys_newfstat
...
16      64      ioctl                   sys_ioctl
...
514     x32     ioctl                   compat_sys_ioctl
```

Generation

- Kbuild calls the *syscalltbl.sh* to generate *arch/x86/include/generated/asm/syscalls_{64,32}.h*
- Same with *syscallhdr.sh*

A guided tour of some syscalls

sysinfo

kernel/sys.c

```
SYSCALL_DEFINE1(sysinfo,
                struct sysinfo __user *, info)
{
        struct sysinfo val;

        do_sysinfo(&val);

        if (copy_to_user(info, &val,
                         sizeof(struct sysinfo)))
                return -EFAULT;

        return 0;
}
```

__user

- Used by tools such as *sparse* to statically check the use of userspace pointers.
- *# define __user __attribute__((noderef, address_space(1)))*

`copy_to_user`

- Copy data from kernel land to user land.

  Checks that all bytes are writeable, using:

-
  - *access_ok(VERIFIY_WRITE, addr_to, length)*

ioctl

```
#include <sys/ioctl.h>

int ioctl(int d, unsigned long request, ...);
```

- Control devices.

  A big mess:

  - Request numbers encodes data.
  - Request data is untyped (*void **).
- See LDD3, Chapter 6: Advanced Char Driver Operations.

clone

clone

```
SYSCALL_DEFINE5(clone, unsigned long, clone_flags,
                unsigned long, newsp,
                int __user *, parent_tidptr,
                int __user *, child_tidptr,
                int, tls_val)
{
    return do_fork(clone_flags, newsp, 0, parent_tidptr,
child_tidptr);
}
```

fork

```
SYSCALL_DEFINE0(fork)
{
    return do_fork(SIGCHLD, 0, 0, NULL, NULL);
}
```

vfork

```
SYSCALL_DEFINE0(vfork)
{
    return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, 0,
                    0, NULL, NULL);
}
```

personality

```
#include <sys/personality.h>

int personality(unsigned long persona);
```

- Sets the process execution domain
- Used by *setarch*

  Tweak:

  - uname-2.6
  - exposed architecture (*i386*, *i486*, *i586*, etc.)
  - *STICKY_TIMEOUT*
- - …

reboot

```
#include <unistd.h>
#include <linux/reboot.h>

int reboot(int magic, int magic2, int cmd, void *arg);

This system call will fail (with EINVAL) unless magic equals
LINUX_REBOOT_MAGIC1 (that is, 0xfee1dead) and magic2 equals
LINUX_REBOOT_MAGIC2 (that is, 672274793).  However, since 2.1.17
also
LINUX_REBOOT_MAGIC2A (that is, 85072278) and since 2.1.97 also
LINUX_REBOOT_MAGIC2B (that is, 369367448) and since 2.5.71 also
LINUX_REBOOT_MAGIC2C (that is, 537993216) are permitted as value
for
magic2. (The hexadecimal values of these constants are
meaningful.)
```

rt_XXX syscalls

The addition or real-time signals required the widening of the signal set structure (*sigset_t*) from 32 to 64 bits. Consequently, various system calls were superseded by new system calls that supported the larger signal sets.

Linux < 2.0	**Linux >= 2.2**
sigaction(2)	*rt_sigaction(2)*
sigpending(2)	*rt_sigpending(2)*
sigprocmask(2)	*rt_sigprocmask(2)*
sigreturn(2)	*rt_sigreturn(2)*
sigsusprend(2)	*rt_sigsuspend(2)*
sigtimedwait(2)	*rt_sigtimedwait(2)*

Going further than syscalls

- There are places in the kernel where the complexity of the task goes bewond a call to a function.
- *ioctl* has grew dangerously.
- For example, *netlink(7)* aims to replace *ioctl* for network configuration.

References

- **http://lwn.net/Articles/604287/**
- **http://lwn.net/Articles/604515/**
- **https://www.kernel.org/doc/htmldocs/kernel-hacking**
- Searchable Linux Syscall Table: **https://filippo.io/linux-syscall-table/**