Linux Kernel Internals

An Introduction

Jörg Faschingbauer

www.faschingbauer.co.at

jf@faschingbauer.co.at

Table of Contents

1. How Linux Boots
    - Overview
    - Mounting Filesystems
    - `chroot`
    - Bind and Move Mounts
    - Early Userspace
2. Kernel Management
    - Kernel Source
    - Kernel Modules
3. Kernel Internals
    - Duplicating
    - Character Devices
    - Locking
    - Communication
    - Memory
    - Interrupts
4. Realtime
    - Realtime Preemption
    - Xenomai

Overview

1 How Linux Boots
  - Overview
  - Mounting Filesystems
  - `chroot`
  - Bind and Move Mounts
  - Early Userspace

2 Kernel Management
  - Kernel Source
  - Kernel Modules

3 Kernel Internals
  - Duplicating
  - Character Devices
  - Locking
  - Communication
  - Memory
  - Interrupts

4 Realtime
  - Realtime Preemption
  - Xenomai

Overview

1. How Linux Boots
   - Overview
   - Mounting Filesystems
   - `chroot`
   - Bind and Move Mounts
   - Early Userspace
2. Kernel Management
   - Kernel Source
   - Kernel Modules
3. Kernel Internals
   - Duplicating
   - Character Devices
   - Locking
   - Communication
   - Memory
   - Interrupts
4. Realtime
   - Realtime Preemption
   - Xenomai

Booting - Overview

Many terms → confusion:

- Root Filesystem
- Root Directory
- Kernel Commandline
- Userspace
- initrd
- initramfs
- OS Image
- `init`

Root Filesystem

- Definition: the *Root Filesystem* is the filesystem where the first program is
- “Userspace is born”
- Trditionally called `init` (but can be anything)
- Problem: how does the kernel know where the root filesystem is?
- Kernel commandline: for example, `root=/dev/sda1`, or `root=/dev/mtdblock3`
- `/sbin/init` if not otherwise specified. Explicit: `init=/my/init`
- Driver for root filesystem has to be built into kernel image
    - Modules are loaded from userspace

→ Kernel *mounts* root filesystem as specified on kernel commandline (visible in `/proc/cmdline`)

Root Filesystem, More Complex

Problem: a filesystem's parameters aren't always as simple as `/dev/sda1` ...

- Network Filesystem (NFS). Historically implemented in the kernel.
- Encrypted partition → many parameters (algorithm, pass phrase, ...)
- Logical Volume Manager (LVM)
- ...

→ Not easily governed via the kernel commandline
→ **Solution:** "Early Userspace"

Overview

FASCHINGBAUER

1 How Linux Boots
- Overview
- Mounting Filesystems
- chroot
- Bind and Move Mounts
- Early Userspace

2 Kernel Management
- Kernel Source
- Kernel Modules

3 Kernel Internals
- Duplicating
- Character Devices
- Locking
- Communication
- Memory
- Interrupts

4 Realtime
- Realtime Preemption
- Xenomai

The `mount` Command

Hierarchy of Unix systems is transparently extensible (*26 drive letters? What the ...?!?*)

- "Mounting" a filesystem on a *mount point*
- Hierarchy is *transparently* composed of multiple filesystems
- Filesystem is contained in a *block device*

Mounting, e.g.:

```
# mount /dev/sdb1 /mnt/usb-stick
# mount /dev/mmcblk0p3 /home
```

The `mkfs` Command

How are filesystems created? (Doze: how are partitions *formatted*?)

- Thousands of different filesystems: `ext2`, `ext3`, `xfs`, `btrfs`, ...
- Every filesystem has a different *format*
- → Filesystem specific `mkfs` programs; z.B. `mkfs.ext2`
- Flash filesystems are different
    - Operate directly in flash memory → no block device involved

`mkfs`

```
# mkfs.ext2 /dev/sdb1
```

Loop Mounts — Filesystem in a File (1)

FASCHINGBAUER

Question: if `/dev/sda1` looks like a file, why shouldn't a real file contain a file system?

Answer: who said it cannot?

- `mkfs` can operate on files (everything is a file, right?)
- *But*: a file is not a block device → "loop" mount

Step one: create empty file

```
# dd if=/dev/zero of=./my-image bs=4096 count=1024
```

Loop Mounts — Filesystem in a File (2)

FASCHINGBAUER

Step two: filesystem into file

```
# mkfs.ext2 ./my-image
mke2fs 1.41.14 (22-Dec-2010)
./my-image is not a block special device.
Proceed anyway? (y,n) y
```

`man mkfs.ext2` → `-F` to suppress annoying question

Check: file type?

```
# file ./my-image
./my-image: Linux rev 1.0 ext2 filesystem data, ...
```

Loop Mounts — Filesystem in a File (3)

Mounting the *image* on a *mount point* …

Loop-mounting `my-image`

```
# mkdir ./my-mountpoint
# mount -o loop ./my-image ./my-mountpoint
# ls ./my-mountpoint/
lost+found
```

Loop Mounts — Filesystem in a File (4)

Image is now mounted → one can modify it just like any other filesystem

```
# cp -r ~jfasch ./my-mountpoint
# umount ./my-mountpoint
```

Overview

1. How Linux Boots
   - Overview
   - Mounting Filesystems
   - `chroot`
   - Bind and Move Mounts
   - Early Userspace
2. Kernel Management
   - Kernel Source
   - Kernel Modules
3. Kernel Internals
   - Duplicating
   - Character Devices
   - Locking
   - Communication
   - Memory
   - Interrupts
4. Realtime
   - Realtime Preemption
   - Xenomai

The Root Directory

The root directory is special:

- Absolute paths (e.g. `/bin/bash`) do start there
- There are no entries above it
- → Cannot escape → "Jail"

Exact definition:

- "Root directory" is a process attribute → each process can have its own root directory
- Path lookup starts there
- A process's "root directory" attribute is inherited → child processes have the same root as its parent

→ not so special at all!

Changing Root Directory — `chroot` (System Call

System Call `chroot` (→ `man -s 2 chroot`)

- Changes path lookup for the calling process (*and does nothing else*)
- Current Working Directory (CWD) remains the same
- Open files remain open
- → relatively useless on its own

Changing Root Directory — `chroot` (Command)

Command `chroot` (→ `man chroot`)

- Shell Command
- Combines `chroot()` with execution of a program
- Program must exist in new root
- All prerequisites (shared libraries, ...) must exist in new root

→ “Chroot Jail”

`chroot`: Demo Time

… working environment with `/bin/bash` …

`chroot`: Use Cases

- Environment for services that are not trustworthy (better yet: containers, virtual machines)
- Build environment for other systems (building for Ubuntu on a Fedora system for example)
- “Boot-through”: booting into a temporary RAM filesystem (*initramfs*), load drivers from there (NFS, encryption, whatever), mount *real root*, and then boot into the now-mounted *real root*

Overview

FASCHINGBAUER

1 How Linux Boots
- Overview
- Mounting Filesystems
- `chroot`
- Bind and Move Mounts
- Early Userspace

2 Kernel Management
- Kernel Source
- Kernel Modules

3 Kernel Internals
- Duplicating
- Character Devices
- Locking
- Communication
- Memory
- Interrupts

4 Realtime
- Realtime Preemption
- Xenomai

Bind Mounts

- Chroot-Jail is *a jail*
- → Symbolic links to the outer world don't work

→ **Bind Mounts**

- Mount files and directories, rather than device nodes
- → Mount points can be files and directories

Bind Mounts: Demo Time

Bind Mount: example

```
# mkdir -p ./my-mountpoint/home/jfasch
# mount -o bind /home/jfasch ./my-mountpoint/home/jfasch
```

...

Move Mounts

To move mount points cries for trouble (`umount` is confused ...)

Clean method:

Move mounts

```
# mkdir old-mountpoint new-mountpoint
# mount /dev/sda1 old-mountpoint
# mount --move old-mountpoint new-mountpoint}
```

Use: Initramfs is a typical example

- Main task: prepare real/final root filesystem
- Temporarily mounted somewhere
- At the time of switching (→ `chroot`) into the real root filesystem, `procfs` und `sysfs` are moved there

Overview

1. How Linux Boots
   - Overview
   - Mounting Filesystems
   - `chroot`
   - Bind and Move Mounts
   - Early Userspace
2. Kernel Management
   - Kernel Source
   - Kernel Modules
3. Kernel Internals
   - Duplicating
   - Character Devices
   - Locking
   - Communication
   - Memory
   - Interrupts
4. Realtime
   - Realtime Preemption
   - Xenomai

Late vs. Early Userspace

“Late Userspace”

- Kernel has to do *a lot* to make root filesystem available
- Hardware initialization (SATA, MTD, ...)
- Mounting the filesystem, applying the right parameters
  - Parameters usually passed via *kernel commandline*
- → inflexible!

Doing complicated things does not belong in the kernel → “Early Userspace”

RAM Filesystem

`ramfs` - **RAM Filesystem**

- Simple filesystem in RAM
- Grows and shrinks with content

Elder brother, the fat and dumb *ramdisk* ...

- Fixed sized block device in RAM
- Contains a real file system

Initial RAM Filesystem — `initramfs`

- Kernel has always a `cpio` archive built-in
- Empty by default
- During boot: unpacked into a RAM filesystem → `initramfs`
- If the filesystem contains `/init` → done. `/init` (PID 1) takes control over booting.
- Else → as before, `root=/dev/sda1` etc.

Initial RAM Filesystem — Demo Time

- `CONFIG_INITRAMFS_SOURCE` (“General setup”)
- Don’t forget `console 5 1`

Overview

FASCHINGBAUER

1. How Linux Boots
   - Overview
   - Mounting Filesystems
   - chroot
   - Bind and Move Mounts
   - Early Userspace
2. Kernel Management
   - Kernel Source
   - Kernel Modules
3. Kernel Internals
   - Duplicating
   - Character Devices
   - Locking
   - Communication
   - Memory
   - Interrupts
4. Realtime
   - Realtime Preemption
   - Xenomai

Overview

1. How Linux Boots
   - Overview
   - Mounting Filesystems
   - chroot
   - Bind and Move Mounts
   - Early Userspace
2. Kernel Management
   - Kernel Source
   - Kernel Modules
3. Kernel Internals
   - Duplicating
   - Character Devices
   - Locking
   - Communication
   - Memory
   - Interrupts
4. Realtime
   - Realtime Preemption
   - Xenomai

Kernel Source

- Maintained with Git
- → Distributed
- Not centrally maintained
- Linux Torvalds plays the role of "integrator"
- → Pulls changes on a regular basis
- Releases on `www.kernel.org`
- Linus' development tree: `github.com/torvalds/linux`

```
$ git clone https://github.com/torvalds/linux.git
```

Kernel Source Overview

Top level directory

- `Documentation`: large hierarchy of `.txt` files
    - Varying quality (it's getting better though)
    - Must-read for developers
- `include/uapi`: header files for use by userspace
- `include` (except `uapi`): internal header files
- `kernel`: core kernel implementation (`sched/`, `irq/`, `time/`, ...)
- `block`, `crypto`, `ipc`, `security`, `sound` ...: various "subsystems"
- `drivers`: this is where most code is

Git, Configuration, Build, ...

Best learned from the Internet ...

- `www.faschingbauer.co.at/de/howtos/raspi-kernel-build/`
- `Documentation/kbuild/` in the kernel source

Overview

FASCHINGBAUER

1. How Linux Boots
   - Overview
   - Mounting Filesystems
   - chroot
   - Bind and Move Mounts
   - Early Userspace
2. Kernel Management
   - Kernel Source
   - Kernel Modules
3. Kernel Internals
   - Duplicating
   - Character Devices
   - Locking
   - Communication
   - Memory
   - Interrupts
4. Realtime
   - Realtime Preemption
   - Xenomai

Which Modules are Loaded?

/proc/modules

```
 $ cat /proc/modules
cfg80211 506427 0 - Live 0xbf119000
rfkill 21324 1 cfg80211, Live 0xbf10e000
i2c_bcm2708 5960 0 - Live 0xbf102000
bcm2835_gpiomem 3695 0 - Live 0xbf0fe000
...
```

More information: `lsmod`

```
$ lsmod
Module                  Size  Used by
cfg80211              506427  0
rfkill                 21324  1 cfg80211
i2c_bcm2708             5960  0
bcm2835_gpiomem         3695  0
...
```

Module Metadata

```
$ modinfo i2c_bcm2708
filename:       /lib/modules/4.1.10-rt-jfasch+/kernel/drivers/
alias:          platform:bcm2708_i2c
license:        GPL v2
author:         Chris Boot <bootc@bootc.net>
description:    BSC controller driver for Broadcom BCM2708
srcversion:     E126C7409891BBDF7859E58
alias:          of:N*T*Cbrcm,bcm2708-i2c*
depends:
intree:         Y
vermagic:       4.1.10-rt-jfasch+ preempt mod_unload modversi
parm:           baudrate:The I2C baudrate (uint)
parm:           combined:Use combined transactions (bool)
```

Loading Modules: `insmod`

Loading a single module: `insmod`

```
# insmod /lib/modules/4.1.10-rt-jfasch+/kernel/drivers/i2c/i2
```

Fails when dependencies are not satisfied ...

```
# insmod /lib/modules/4.1.10-rt-jfasch+/kernel/sound/soc/...
...bcm/snd-soc-hifiberry-dac.ko
insmod: ERROR: could not insert module /lib/modules/4.1.10-...
...rt-jfasch+/kernel/sound/soc/bcm/snd-soc-hifiberry-dac.ko: .
...Unknown symbol in module
```

Loading Modules: `modprobe`

Load a module, along with all its dependencies

- Unlike `insmod`, the module must be *installed*
- Uses generated `modules.dep` in `/lib/modules/$(uname -r)`
- → `depmod`

```
# modprobe snd-soc-hifiberry-dac
```

Unloading Modules: `rmmod` vs. `modprobe`

Multiple ways to unload code ...

- `rmmod` *modulename*: unloads module only
    - Leftover dependencies (modules that are not used anymore)
- `modprobe -r` *modulename*
    - Cleans up dependency graph
    - Unloads all modules which are not used anymore

Overview

1. How Linux Boots
   - Overview
   - Mounting Filesystems
   - chroot
   - Bind and Move Mounts
   - Early Userspace
2. Kernel Management
   - Kernel Source
   - Kernel Modules
3. Kernel Internals
   - Duplicating
   - Character Devices
   - Locking
   - Communication
   - Memory
   - Interrupts
4. Realtime
   - Realtime Preemption
   - Xenomai

Kernel Concepts

Kernel: is/supplies the world where *processes* live

- *Schedules* processes → *fair* and *realtime*
- Provides *entry points* for processes
  - System calls: `open()`, `read()`, `write()`, `close()`, and hundreds more
  - Character devices: dedicated communication with device drivers (accessible like files)
  - Sysfs: dedicated communication with device drivers (the modern way)
  - ...
- Handles device interrupts
- Extremely parallel
  - Processes switch to kernel mode via system calls
  - Kernel threads
  - Interrupts
  - → Many locking primitives for different purposes

Parallel Programming: *Process Context*

Process context: everything that can be identified by a *process ID*

- Processes (and threads) that execute in user mode → process address space
- Processes (and threads) that execute in kernel mode → kernel address space
- Kernel threads → kernel address space

Preemption ...

- Process context is subject to *scheduling*
- Fair scheduling: *preemption* at end of time slice
- Realtime: *preemption* when higher priority process/thread is runnable

Parallel Programming: *Race Conditions*

When do race conditions occur?

- Two processes/threads share the same address space
- Manipulate the same data structure

In kernel address space?

- Userspace processes executing a system call ("switch to kernel mode")
- Kernel threads

Protection through locking

- Mutexes: locker has to *wait* until unlocked
- Spinlocks: locker *loops* until unlocked
    - *Atomic context*

Parallel Programming: *Atomic Context*

Atomic context is where code must not sleep!

- Interrupt service routine
    - Interrupts disabled
    - No preemption, no scheduling, no nothing
    - → primary source of latency
- "Bottom half" — code that runs in interrupt context (not subject to scheduling), but interrupts are already enabled
    - Deferred work → "tasklet", "soft-IRQ"
    - Best avoided because not easily controllable, realtime-wise
- All code that holds a *spinlock*

Parallel Programming: Atomic vs. Process Contex

FASCHINGBAUER

Atomic context must not sleep

- Preemption disabled → prioritization impossible
- High latency if atomic code runs for too long
- Severe restrictions
  - Paging
  - Locking is difficult
  - ...

Process context ...

- Subject to scheduling → easily prioritized (be it realtime or not)
- Easy locking

Conclusion

- Atomic context best avoided
- ... at least when absolute control is desired

Overview

FASCHINGBAUER

1. How Linux Boots
   - Overview
   - Mounting Filesystems
   - `chroot`
   - Bind and Move Mounts
   - Early Userspace
2. Kernel Management
   - Kernel Source
   - Kernel Modules
3. Kernel Internals
   - Duplicating
   - Character Devices
   - Locking
   - Communication
   - Memory
   - Interrupts
4. Realtime
   - Realtime Preemption
   - Xenomai

File Descriptors, Open File, I-Node

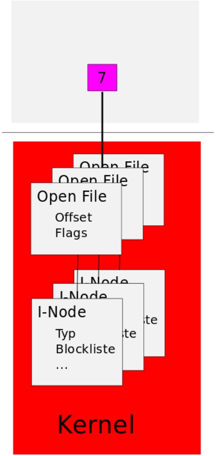
File descriptor is a "handle" to a more complex structure

File ("Open File")

- Offset
- Flags

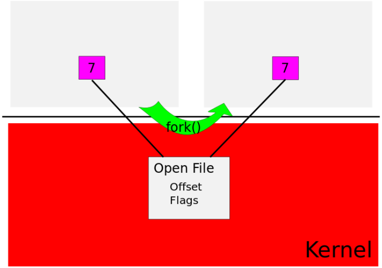
I-Node

- Type
- Block list
- ...



File Descriptors and Inheritance

- A call to `fork()` inherits file descriptors
- → reference counted copy of the same "Open File".
- → Processes share flags and offset!
- File closed (*open file* freed) only when last copy is closed



Duplicating File Descriptors

man 2 dup

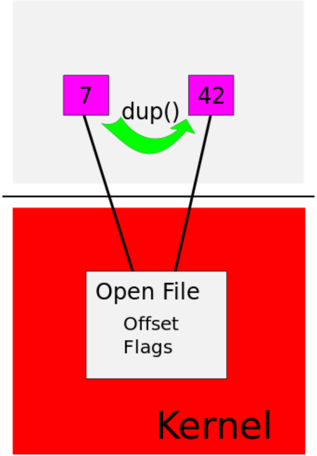
```
int dup(int oldfd);
```

- Return: new file descriptor

man 2 dup2

```
int dup2(int oldfd, int newfd);
```

- `newfd` already open/occupied → implicit `close()`



Example: Shell Stdout-Redirection (1)

Stdout-Redirection

```
$ /bin/echo Hello > /dev/null
```

- Redirection is a shell responsibility (`/bin/bash`)
- `echo` writes “Hello” to standard output.
- … and does not want/have to care where it actually goes

Example: Shell Stdout-Redirection (2)

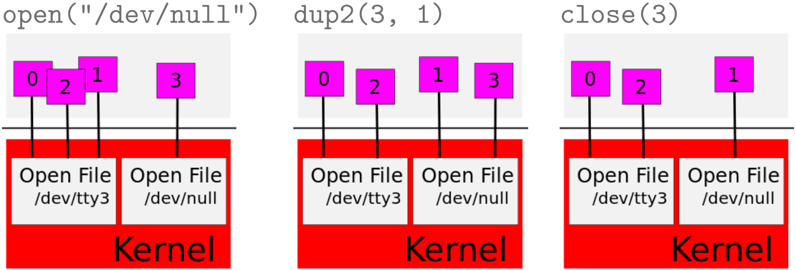
FASCHINGBAUER

Stdout-Redirection

```
$ strace -f bash -c '/bin/echo Hallo > /dev/null'
[3722] open("/dev/null", O_WRONLY|O_...) = 3
[3722] dup2(3, 1) = 1
[3722] close(3) = 0
[3722] execve("/bin/echo", ...) = 0
```

(`fork()`, `exec()`, `wait()` omitted for clarity.)

Example: Shell Stdout-Redirection (2)



Overview

1 How Linux Boots
- Overview
- Mounting Filesystems
- `chroot`
- Bind and Move Mounts
- Early Userspace

2 Kernel Management
- Kernel Source
- Kernel Modules

3 Kernel Internals
- Duplicating
- Character Devices
- Locking
- Communication
- Memory
- Interrupts

4 Realtime
- Realtime Preemption
- Xenomai

Character Devices

"Everything is a file" → so are *driver interfaces*

- Path name so userspace can find driver interface
- Commonly stored in `/dev` (but not necessarily so)
- *Major number*: driver identification
- *Minor number*: functionality inside driver

```
crw-r----- 1 root kmem 1,  1 Nov 13 14:23 /dev/mem
crw-rw-rw- 1 root root 1,  3 Nov 13 14:23 /dev/null
crw-r----- 1 root kmem 1,  4 Nov 13 14:23 /dev/port
crw-rw-rw- 1 root root 1,  8 Nov 13 14:23 /dev/random
crw-rw-rw- 1 root root 1,  9 Nov 13 14:23 /dev/urandom
crw-rw-rw- 1 root root 1,  5 Nov 13 14:23 /dev/zero
```

Character Devices: Creation

Good old Unix way ...

```
# mknod ~/random c 1 8
# cat ~/random
... entropy ...
```

Problems:

- Populating `/dev` by hand is cumbersome
- One node for every piece of hardware that might possibly exist
  - Distributions used to ship with a huge tarball of `/dev` entries
- Running out of major numbers
- Historically, every driver had its own unique major number
- Major number conflicts → central registry, like PCI vendor numbers?

Character Devices: Creation

Linux way: `devtmpfs`

- File system that contains device nodes
- Automatically populated by the kernel
- … with a little driver support

```
$ mount
...
devtmpfs on /dev type devtmpfs (rw,relatime,...)
...
```

Interface Definition

FASCHINGBAUER

Character devices are interfaces

- Driver writer supplies methods (read, write, ...)
- Semantics are up to the implementor
- Good Unix citizenship encouraged (but not enforced)

```
#include <linux/fs.h>

struct file_operations my_ops = {
    .owner = THIS_MODULE,
    .open = my_open,
    .read = my_read,
    .write = my_write,
    .unlocked_ioctl = my_ioctl
};
```

Available Methods

More methods “overloadable” …

- All methods receive `struct file` as “this” parameter
- `open`: implements `man -s 2 open` — inode already loaded, `struct file` allocated → “constructor”
- `read`: implements `man -s 2 read`
- `write`: implements `man -s 2 write`
- `unlocked_ioctl`: implements `man -s 2 ioctl`
- `flush`: reference count decremented
- `release`: reference count reached zero → `struct file` freed

`open()`: Userspace

FASCHINGBAUER

`man -s 2 open`

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

- Opens and/or creates a file
- Many flags/parameters
- Permissions
- Driver not concerned with all that
- → *Virtual File System* layer

`open()`: Kernelspace

- All complicated stuff done by VFS layer
- Hook for driver to associate driver data with `struct file`
- Looks weird
- Is simple
- → Later by example

`ioctl()`: Userspace

Swiss army knife ...

- Used to communicate with drivers
- All that doesn't fit in `read()`, `write()`

`man -s 2 ioctl`

```
#include <sys/ioctl.h>
int ioctl(int fd, unsigned long request, ...);
```

- `fd`: handle to open device node
- `request`: device specific request code
- `...`: (if any) a single parameter
  - Usually a pointer
  - Can be integer, but should be of pointer size
  - Type depends on value of `request`

`ioctl()`: Kernelspace

```
static long my_ioctl(
    struct file *file,
    unsigned int request,
    unsigned long arg) {...}
```

- `file`: (as always) in-kernel pendant to userspace file descriptor
- `request`: userspace request
- `arg`: the "..." parameter from userspace. descriptor. Cast arbitrarily, depending on `request`

Filling in Functionality: `struct cdev`

- `struct cdev`: *the* device object
- This is what is opened
- Created, initialized by driver
- Announced to userspace through device node
- Usually embedded in a larger structure
- → `container_of` macro

→ `20-cdev-manual-mknod.c`

Overview

1. How Linux Boots
   - Overview
   - Mounting Filesystems
   - `chroot`
   - Bind and Move Mounts
   - Early Userspace
2. Kernel Management
   - Kernel Source
   - Kernel Modules
3. Kernel Internals
   - Duplicating
   - Character Devices
   - Locking
   - Communication
   - Memory
   - Interrupts
4. Realtime
   - Realtime Preemption
   - Xenomai

Locking in the Kernel

Userspace parallelism is simple …

- *All* code is preemptible
- … no way of disabling preemption
- Critical sections are best protected by a mutex (`pthread_mutex_t`)

Kernel parallelism is different …

- Schedulable code
  - Processes in kernel mode
  - kernel threads
- Non-schedulable code
  - Interrupt service routines
  - Other atomic code (spinlock holders)

Mutual Exclusion: Mutex

Process context vs. process context

- Classic mutex semantics
- Binary semaphore
- If held, arriving processes have to wait — they are *scheduled*

```
#include <linux/mutex.h>
struct mutex mutex;
```

OO-like constructor and destructor

```
mutex_init(&mutex);
mutex_destroy(&mutex);
```

Mutex: Locking (1)

Locking is done in many different ways ...

- Preferred version: “interruptible”

```
int error = mutex_lock_interruptible(&mutex);
```

- Puts the caller to sleep if lock is held by someone else
    - Attention: no protection against self-deadlock!
- “Interruptible”: return `-EINTR` (“Interrupted system call”) if process receives a signal
    - Good old Unix
- Uninterruptible sleeps should be used with care

Mutex: Locking (2)

Recursive locking …

```
int error = mutex_lock_interruptible_nested(&mutex);
```

- Same process may lock multiple times (no deadlock)
- Must unlock as many times
- Use is questionable though

Polling …

```
int error = mutex_trylock(&mutex);
```

- Lock if not held
- Otherwise, return `-EAGAIN` immediately
- Use is even more questionable than recursive

Mutex: Releasing

At the end of the critical section …

```
mutex_unlock(&mutex);
```

- Releases the lock
- Wakes up waiter if any

Realtime Mutex

`struct mutex` **does not support priority inheritance**

Linus Torvalds does not like realtime

"Friends don't let friends use priority inheritance. Just don't do it. If you really need it, your system is broken anyway."

- `lwn.net/Articles/178253/`
- Features from the `PREEMPT_RT` tree keep trickling in
- → "Realtime" mutex with priority inheritance
- Used just like ordinary mutex

```
#include <linux/rtmutex.h>
struct rt_mutex mutex;
```

Mutual Exclusion: Spinlock

Atomic context must not sleep → *busy waiting*

- The only locking possibility in atomic context
- Can also be used in process context
  - Cheap — no context switch if lock is held
  - Interrupts off on local CPU → anti-realtime

How does it work?

- On a *Uniprocessor*
  - Disable interrupts
  - $\Longrightarrow$ preemption disabled
  - $\Longrightarrow$ lock in its simplest form
- On a *Multiprocessor*
  - Set “locked” flag (atomically)
  - Disable interrupts on local processor
  - $\Longrightarrow$ no preemption on local processor
  - Remote processors busy wait around the “locked” flag (atomically trying to *test-and-set* it)

Spinlock: Initialization

```
#include <linux/spinlock.h>
spinlock_t lock;
```

Initialization

```
spin_lock_init(&lock);
```

- No destructor available

Spinlock: Usage

Too many variations ...

- Multiple spinlocks can be acquired in a lock chain
- Most variations don't keep track of interrupt state
    - Too easy to re-enable interrupts *too early*
    - One cannot always control the call chain
- → Only one variation that is *really safe*

```
unsigned long irqflags;

spin_lock_irqsave(&lock, irqflags);
...
spin_unlock_irqrestore(&lock, irqflags);
```

- No nesting (no recursive locks) → deadlock

Mutual Exclusion: Conclusion

There is always a tradeoff ...

- Spinlocks are good
    - No expensive *context switch* during lock contention
    - Can be used in (between) interrupt context and process context
- Spinlocks are bad
    - No sleep! (→ no easy memory allocation, no easy this, no easy that)
    - Must be held *very short* → no scheduling/preemption on local processor
- Mutexes are good
    - Sleeping allowed
    - Everything's easy
- Mutexes are bad
    - Expensive *context switch* during lock contention
    - Cannot be used in interrupt context
    - → no easy data sharing between process and interrupt context

Overview

FASCHINGBAUER

1 How Linux Boots
  - Overview
  - Mounting Filesystems
  - `chroot`
  - Bind and Move Mounts
  - Early Userspace
2 Kernel Management
  - Kernel Source
  - Kernel Modules
3 Kernel Internals
  - Duplicating
  - Character Devices
  - Locking
  - Communication
  - Memory
  - Interrupts
4 Realtime
  - Realtime Preemption
  - Xenomai

Communication: Wait Queues

Wait conditions in the kernel

- Processes (user space and kernel) want to do nothing when there's nothing to do
- Suspend themselves on *wait conditions*
- Wakeup when condition becomes true
- Producer/consumer relationships

Most basic (and widely used) wait condition ...

```
#include <linux/wait.h>

wait_queue_head_t wait_queue;
init_waitqueue_head(&wait_queue);
```

Wait Queue: Waiting (1)

Typical usage pattern

```
do_lock(&lock);
while (!condition) {
    do_unlock(&lock);
    error = wait_event_interruptible(&wait_queue, condition);
    if (error == -EINTR) /*interrupted by signal*/
        return error;
    else {
        /* handle other errors */
    }
    do_lock(&lock);
}
handle_data(...);
do_unlock(&lock);
```

Wait Queue: Waiting (2)

Remarks

- `lock` can be any kind of lock (wait queue is not tied to a lock type)
- `condition` is checked with the lock held (clearly)
- Use *interruptible* sleeps wherever possible
  - Otherwise the waiting process cannot be killed (`Ctrl-C`, for example)
  - Same with mutex waits, same with *any* waits

Wait Queue: Waking

Multiple wait functions …

Preferred: wake up one interruptible waiter

```
wake_up_interruptible(&wait_queue);
```

Remarks

- Normally there should only be interruptible waiters
- `wake_up_interruptible_all()`: "thundering herd"

Wait Queue: Conclusion

Wait queues

- Not the only communication device
    - *Completion*: one-shot device (→ LDD3)
    - *Semaphore*: most basic (at the basis of all others)
- Wakeup possible in interrupt (wake does not sleep)
- Waiting only possible in process context

Overview

1 How Linux Boots
- Overview
- Mounting Filesystems
- `chroot`
- Bind and Move Mounts
- Early Userspace

2 Kernel Management
- Kernel Source
- Kernel Modules

3 Kernel Internals
- Duplicating
- Character Devices
- Locking
- Communication
- Memory
- Interrupts

4 Realtime
- Realtime Preemption
- Xenomai

Dynamic Memory: `kmalloc()`

Kernel heap implementation

- Similar to userspace `malloc()`
- → Easy to use

```
#include <linux/slab.h>

void *kmalloc(size_t size, gfp_t flags);
```

- Memory internally/transparently managed as set of pages
- Pages are not necessarily contiguous
- `size` greater than page size might be more difficult to allocate

Dynamic Memory: `kmalloc()` Flags

Many flags to govern behaviour ...

- `GFP_KERNEL`: most commonly used
  - Might block (triggers swap activity, ...)
  - → Can only be called in process context
- `GFP_ATOMIC`: for use in non-process context
  - Scarce resource → use is discouraged

More ...

- LDD3
- `linux/gfp.h`

Dynamic Memory: More

Freeing memory

```
void kfree(const void *);
```

Allocating zeroed memory

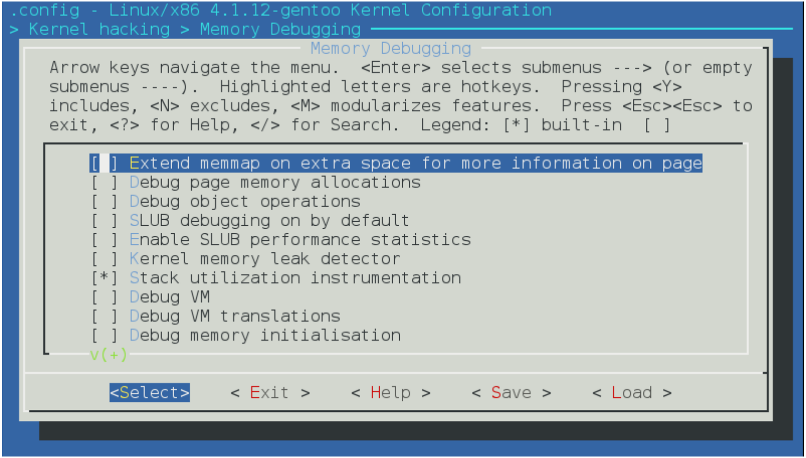
```
void *kzalloc(size_t size, gfp_t flags)
```

Freeing and zeroing memory

```
void kzfree(const void *);
```

Kernel hacking -¿ Memory Debugging

Dynamic Memory: Debugging



```
.config - Linux/x86 4.1.12-gentoo Kernel Configuration
> Kernel hacking > Memory Debugging ─────────────────────────────
                          Memory Debugging
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]

        [ ] Extend memmap on extra space for more information on page
        [ ] Debug page memory allocations
        [ ] Debug object operations
        [ ] SLUB debugging on by default
        [ ] Enable SLUB performance statistics
        [ ] Kernel memory leak detector
        [*] Stack utilization instrumentation
        [ ] Debug VM
        [ ] Debug VM translations
        [ ] Debug memory initialisation
        v(+)

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

I/O Memory

Device registers mapped into memory

- Access is transparent to software
- Just like ordinary memory
- … but the device listens
- → side effects

Implications

- Performance optimization are made at every level
  - Compiler may reorder memory access
  - CPU may reorder memory access
- → May twist order of access that's expected by device

I/O Memory: Reservation

Memory "regions"

- Reserved by drivers (physical address, length)
- Protection against accidental overlapping access
- Shows up in `/proc/iomem`
- No effect otherwise
    - Access works without
    - *But: no reason not to use it*

```
#include <linux/ioport.h>

struct resource *resource = request_mem_region(
   0x20200000, 180, "my-weird-driver");
release_mem_region(0x20200000, 180);
```

Making I/O Memory Accessible

I/O memory ...

- Not directly accessible (as is physical memory in general)
- Not managed by `struct page` (→ later)
- *I/O Memory Mapping* must be created

```
#include <asm/io.h>

void *base = ioremap(0x20200000, 180);
iounmap(base);
```

Accessing I/O Memory

Set of access functions that insert the right compiler and memory barriers ...

- Reading
    - `unsigned int ioread8(void *addr);`
    - `unsigned int ioread16(void *addr);`
    - `unsigned int ioread32(void *addr);`
- Writing
    - `void iowrite8(u8 value, void *addr);`
    - `void iowrite16(u16 value, void *addr);`
    - `void iowrite32(u32 value void *addr);`

... and a lot more

Overview

1. How Linux Boots
   - Overview
   - Mounting Filesystems
   - chroot
   - Bind and Move Mounts
   - Early Userspace
2. Kernel Management
   - Kernel Source
   - Kernel Modules
3. Kernel Internals
   - Duplicating
   - Character Devices
   - Locking
   - Communication
   - Memory
   - Interrupts
4. Realtime
   - Realtime Preemption
   - Xenomai

Interrupts

Interrupt facts

- Interrupt context is not *scheduled*
- No sleeping API calls allowed
- Not easily debugged
- Not easy in general
- No prioritization

But ...

- Threaded interrupt handlers
- ... thanks to `PREEMPT_RT` slowly being integrated in mainline

Interrupt Service Routine

```
static irqreturn_t my_isr(int irq, void *userdata)
{
    /* ... do something with device ... */
    return IRQ_HANDLED;
}
```

For hard ISRs (as opposed to threaded):

- `IRQ_HANDLED`, if interrupt is from device
    - Especially for shared interrupt lines
- `IRQ_NONE` otherwise

Requesting and Freeing Interrupts

```
int error = request_irq(irq_number, my_isr, IRQF_SHARED,
                        "my-super-driver", userdata);
```

- `my_isr` called as soon as interrupts happen
  - Attention: line is hot *immediately*
- `userdata`: “callback” argument to the ISR
- Interrupt shows up under `my-super-driver` in `/proc/interrupts`

```
free_irq(irq_number, userdata);
```

- Shared interrupts: `userdata` must not be NULL

Interrupt Flags

From `<linux/interrupts.h>`

- `IRQF_TRIGGER_RISING`
- `IRQF_TRIGGER_FALLING`
- `IRQF_TRIGGER_HIGH`
- `IRQF_TRIGGER_LOW`

Threaded Interrupts

Problem: an interrupt service routine must not sleep

- Many devices are on external buses like I2C or SPI
    - Interrupt triggered via GPIO line
    - Reading device state is slow
    - E.g. waits for I2C host controller interrupt
    - → sleeps
- Not being able to sleep is simply inconvenient

Solution before interrupts became threaded:

- Allocate a workqueue (`struct workqueue_struct`)
    - Basically a kernel worker thread
- Defer work there by enqueueing it in the ISR
- → Manual, verbose, error prone, duplicated code

Requesting Threaded Interrupts

Two interrupt service routines …

- "Hard" ISR (optional)
  - Decides whether work must be done → `return IRQ_WAKE_THREAD`
  - `IRQ_HANDLED` or `IRQ_NONE` otherwise
- "Threaded" ISR
  - Executed in process context → freedom!

```
error = request_threaded_irq(irq_number,
            my_hard_isr, my_threaded_isr,
            IRQF_SHARED, "my-super-driver", userdata);
```

Additional advantage

- Kernel thread shows up in ps output
- → *scheduled*
- → Reprioritizable!

Overview

FASCHINGBAUER

1. How Linux Boots
    - Overview
    - Mounting Filesystems
    - chroot
    - Bind and Move Mounts
    - Early Userspace
2. Kernel Management
    - Kernel Source
    - Kernel Modules
3. Kernel Internals
    - Duplicating
    - Character Devices
    - Locking
    - Communication
    - Memory
    - Interrupts
4. Realtime
    - Realtime Preemption
    - Xenomai

Overview

FASCHINGBAUER

1 How Linux Boots
  - Overview
  - Mounting Filesystems
  - chroot
  - Bind and Move Mounts
  - Early Userspace
2 Kernel Management
  - Kernel Source
  - Kernel Modules
3 Kernel Internals
  - Duplicating
  - Character Devices
  - Locking
  - Communication
  - Memory
  - Interrupts
4 Realtime
  - Realtime Preemption
  - Xenomai

Realtime in Mainline Linux

Mainline Linux has only *Soft Realtime* (via `SCHED_FIFO` and `SCHED_RR` and Priorities) → no guaranteed response times though

- Interrupt handler not prioritizable → arbitrary code (even realtime code) preempted by potentially unimportant code
- *Spinlocks* (`spinlock_t`) disable interrupts → not “preemptible”
- *Priority inversion* possible

Realtime Preemption Patch: Overview

FASCHINGBAUER

- Developed by Ingo Molnar (Scheduling) and Thomas Gleixner (Timer Infrastruktur, etc.)
- `http://rt.wiki.kernel.org`
- Separate patches for select kernel versions Kernelversionen
- ... or through Git, `git://git.kernel.org/pub/scm/linux/kernel/git/rt/linux-stable-rt.git`

Realtime Preemption Patch: Goals

Goals: solution of all problems

- Interrupt handler in per-interrupt kernel thread
    - ISR's prioritizable using established mechanisms
    - → by their PIDs
- Spinlocks and normal mutexes become *RT-Mutexes*
    - Priority inheritance
    - No spinlocks anymore → critical sections remain preemptible

Tools

- Setting realtime properties (interrupt threads *and* userland): `chrt`
- *CPU affinity*: `taskset`

Traps

- Swap, memory, code: `mlockall(MCL_CURRENT|MCL_FUTURE)`
- Stack prefaulting: `alloca()` and writing
- Too much realtime is bad → new dimension of bugs

Overview

FASCHINGBAUER

1. How Linux Boots
   - Overview
   - Mounting Filesystems
   - chroot
   - Bind and Move Mounts
   - Early Userspace
2. Kernel Management
   - Kernel Source
   - Kernel Modules
3. Kernel Internals
   - Duplicating
   - Character Devices
   - Locking
   - Communication
   - Memory
   - Interrupts
4. Realtime
   - Realtime Preemption
   - Xenomai

Notes