Linux Internals

Liran Ben Haim
liran@mabel-tech.com

MABEL Technology

Tech The Best

Rights to Copy

What is Linux?

- Linux is a kernel that implements the POSIX and Single Unix Specification standards which is developed as an open-source project.
  - Usually when one talks of “installing Linux”, one is referring to a Linux distribution.
- A distribution is a combination of Linux and other programs and library that form an operating system.
  - There exists many such distribution for various purposes, from high-end servers to embedded systems.
  - They all share the same interface, thanks to the LSB standard.
- Linux runs on 21 platforms and supports implementations ranging from ccNUMA super clusters to cellular phones and micro controllers.
- Linux is 18 years old, but is based on the 40 years old Unix design philosophy.

What is Open Source?

Open Source is a way to develop software application in a distributed fashion that allows cooperation of multiple bodies to create the end product.

- They don't have to be from the same company or indeed, any company.

With Open Source software the source code is published and any one can use, learn, distribute, adapt and sell the program.

An Open Source program is protected under copyright law and is licensed to it's users under a software license agreement.

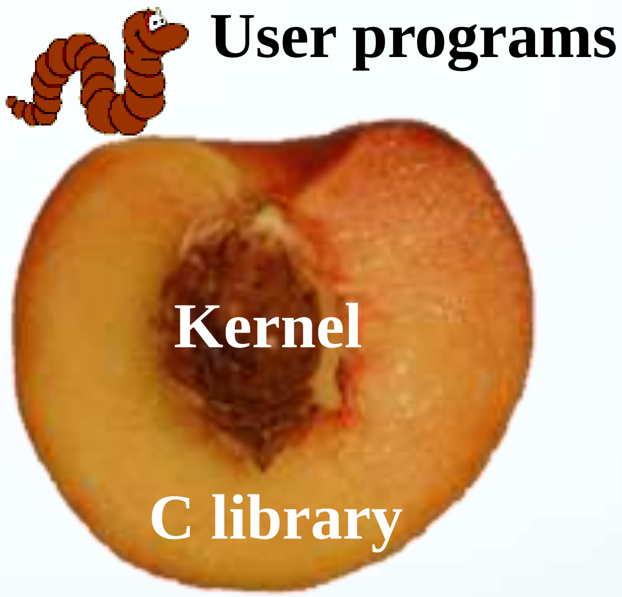
- It is **NOT** software in the public domain.

When making use of Open Source software it is imperative to understand what license governs the use of the work and what is and what is not allowed by the terms of the license.

- The same thing is true for ANY external code used in a product.

Layers in a Linux System

- Kernel
- Kernel Modules
- C library
- System libraries
- Application libraries
- User programs

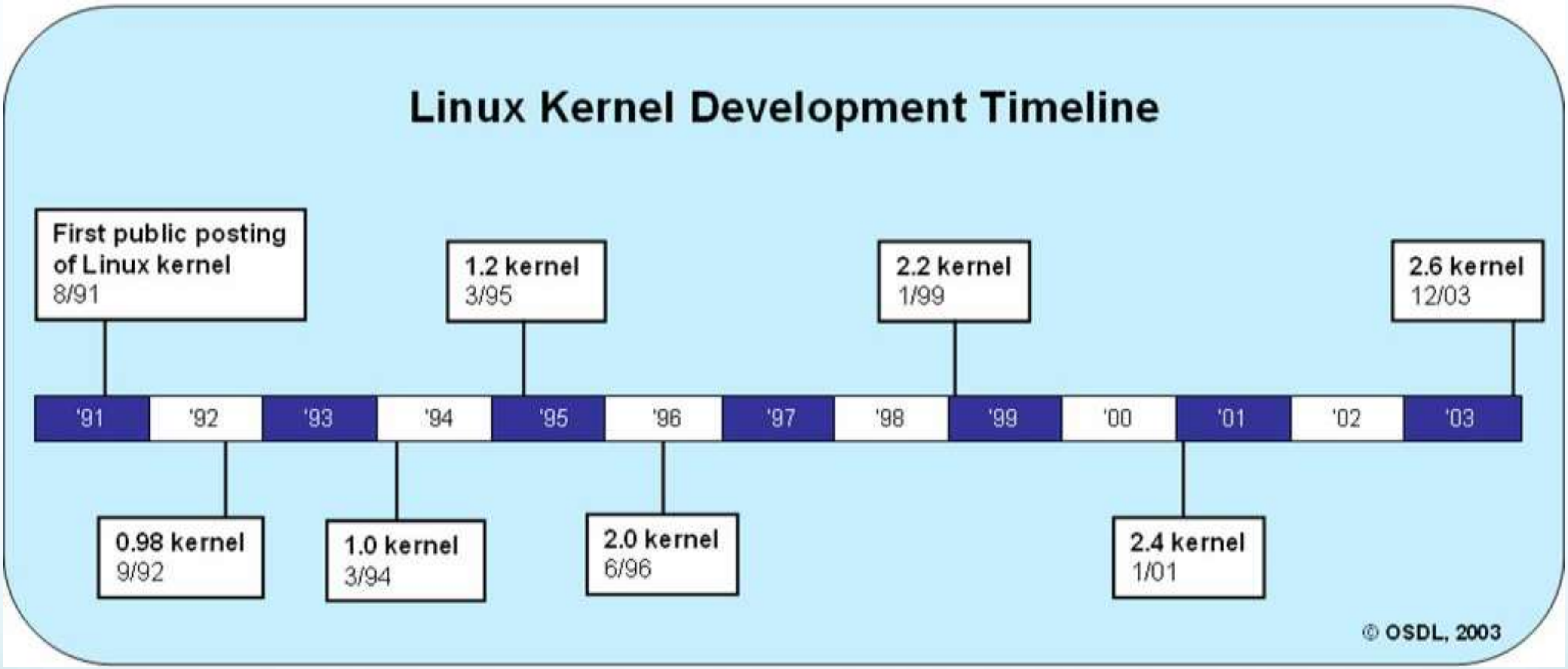


Development

Kernel Overview

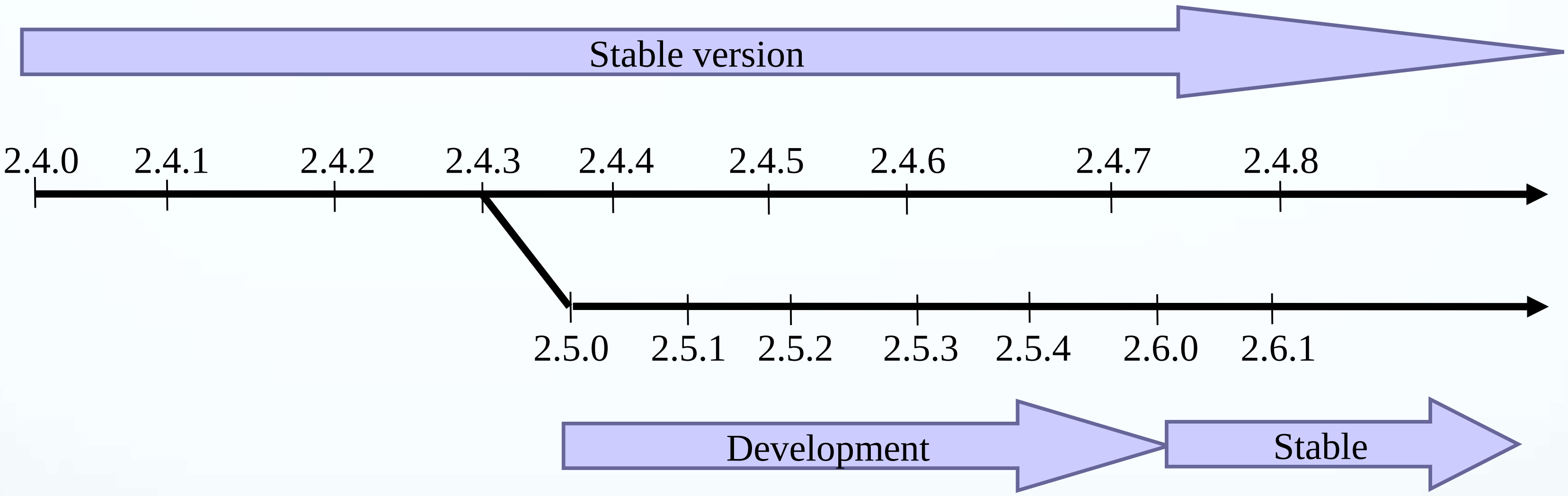
Linux Features

Timeline



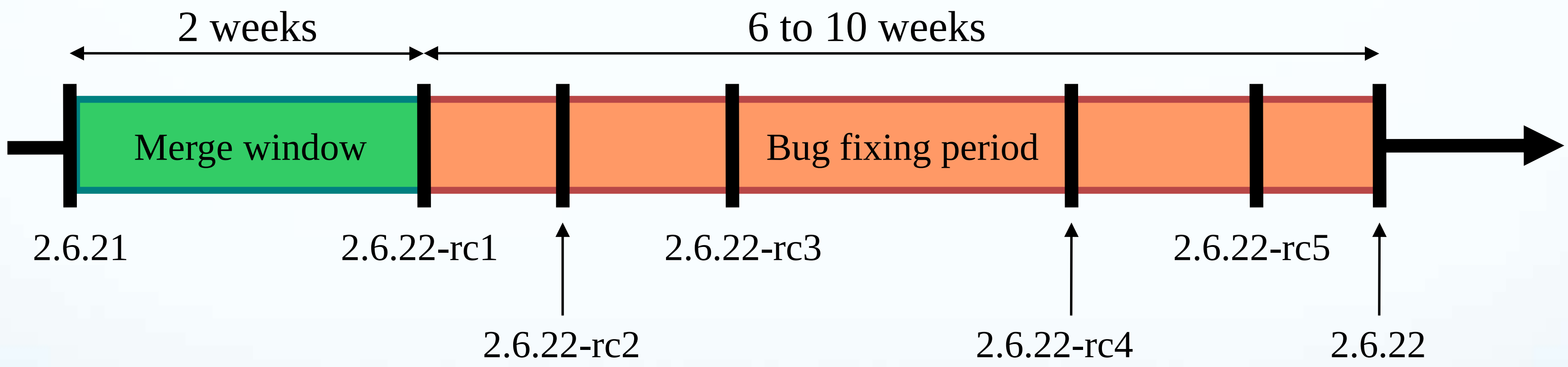
Linux Kernel Development Timeline
First public posting of Linux kernel 8/91
1.2 kernel 3/95
2.2 kernel 1/99
2.6 kernel 12/03
'91 '92 '93 '94 '95 '96 '97 '98 '99 '00 '01 '02 '03
0.98 kernel 9/92
1.0 kernel 3/94
2.0 kernel 6/96
2.4 kernel 1/01
© OSDL, 2003

Until 2.6



Note: in reality, many more minor versions exist inside the stable and development branches

From 2.6 onwards



Linux Kernel Key Features

Portability and hardware support
Runs on most architectures.

Scalability
Can run on super computers as well as on tiny devices
(4 MB of RAM is enough).

Compliance to standards and interoperability.

Exhaustive networking support.

Security
It can't hide its flaws. Its code is reviewed by many experts.

Stability and reliability.

Modularity
Can include only what a system needs even at run time.

Easy to program
You can learn from existing code. Many useful resources on the net.

No stable Linux internal API

Of course, the external API must not change (system calls, /proc, /sys), as it could break existing programs. New features can be added, but kernel developers try to keep backward compatibility with earlier versions, at least for 1 or several years.

The internal kernel API can now undergo changes between two 2.6.x releases. A stand-alone driver compiled for a given version may no longer compile or work on a more recent one.
See Documentation/stable_api_nonsense.txt
in kernel sources for reasons why.

Whenever a developer changes an internal API, (s)he also has to update all kernel code which uses it. Nothing broken!

Works great for code in the mainline kernel tree.
Difficult to keep in line for out of tree or closed-source drivers!

Supported Hardware

See the arch/ directory in the kernel sources

Minimum: 32 bit processors, with or without MMU

32 bit architectures (arch/ subdirectories)
 alpha, arm, cris, frv, h8300, i386, m32r, m68k, m68knommu, mips, parisc, ppc, s390, sh, sparc, um, v850, xtensa

64 bit architectures:
 ia64, mips64, ppc64, sh64, sparc64, x86_64

See arch/<arch>/Kconfig, arch/<arch>/README, or Documentation/<arch>/ for details

MABEL Technology
Tech The Best

What's new in each Linux release?

```
commit 3c92c2ba33cd7d666c5f83cc32aa590e794e91b0
Author: Andi Kleen <ak@suse.de>
Date:   Tue Oct 11 01:28:33 2005 +0200

    [PATCH] i386: Don't discard upper 32bits of HWCR on K8

    Need to use long long, not long when RMWing a MSR. I think
    it's harmless right now, but still should be better fixed
    if AMD adds any bits in the upper 32bit of HWCR.

    Bug was introduced with the TLB flush filter fix for i386

    Signed-off-by: Andi Kleen <ak@suse.de>
    Signed-off-by: Linus Torvalds <torvalds@osdl.org>
...
```

The official list of changes for each Linux release is just a huge list of individual patches!

Very difficult to find out the key changes and to get the global picture out of individual changes.

MABEL Technology
Tech The Best

What's new in each Linux release?

Fortunately, a summary of key changes with enough details is available on http://wiki.kernelnewbies.org/LinuxChanges

For each new kernel release, you can also get the changes in the kernel internal API: http://lwn.net/Articles/2.6-kernel-api/

What's next?
Documentation/feature-removal-schedule.txt lists the features, subsystems and APIs that are planned for removal (announced 1 year in advance).

Linux Internals

Kernel overview

Kernel user interface

Mounting virtual filesystems

Linux makes system and kernel information available in user-space through virtual filesystems (virtual files not existing on any real storage). No need to know kernel programming to access this!

Mounting /proc:

mount -t proc none /proc

Mounting /sys:

mount -t sysfs none /sys

Filesystem type

Raw device
or filesystem image
In the case of virtual
filesystems, any string is fine

Mount point

MABEL Technology
Tech The Best

Kernel userspace interface

- A few examples:

/proc/cpuinfo: processor information

/proc/meminfo: memory status

/proc/version: version and build information

/proc/cmdline: kernel command line

/proc/<pid>/environ: calling environment

/proc/<pid>/cmdline: process command line

... and many more! See by yourself!

man 5 proc

Userspace interface documentation

Lots of details about the /proc interface are available in
Documentation/filesystems/proc.txt
(almost 2000 lines) in the kernel sources.

You can also find other details in the proc manual page:
man proc

See the New Device Model section for details about /sys

Linux Internals

Compiling and booting Linux

Getting the sources

Linux kernel size

Linux 2.6.16 sources:

Raw size: 260 MB (20400 files, approx 7 million lines of code)

bzip2 compressed tar archive: 39 MB (best choice)

gzip compressed tar archive: 49 MB

Minimum compiled Linux kernel size (with Linux-Tiny patches)

approx 300 KB (compressed), 800 KB (raw)

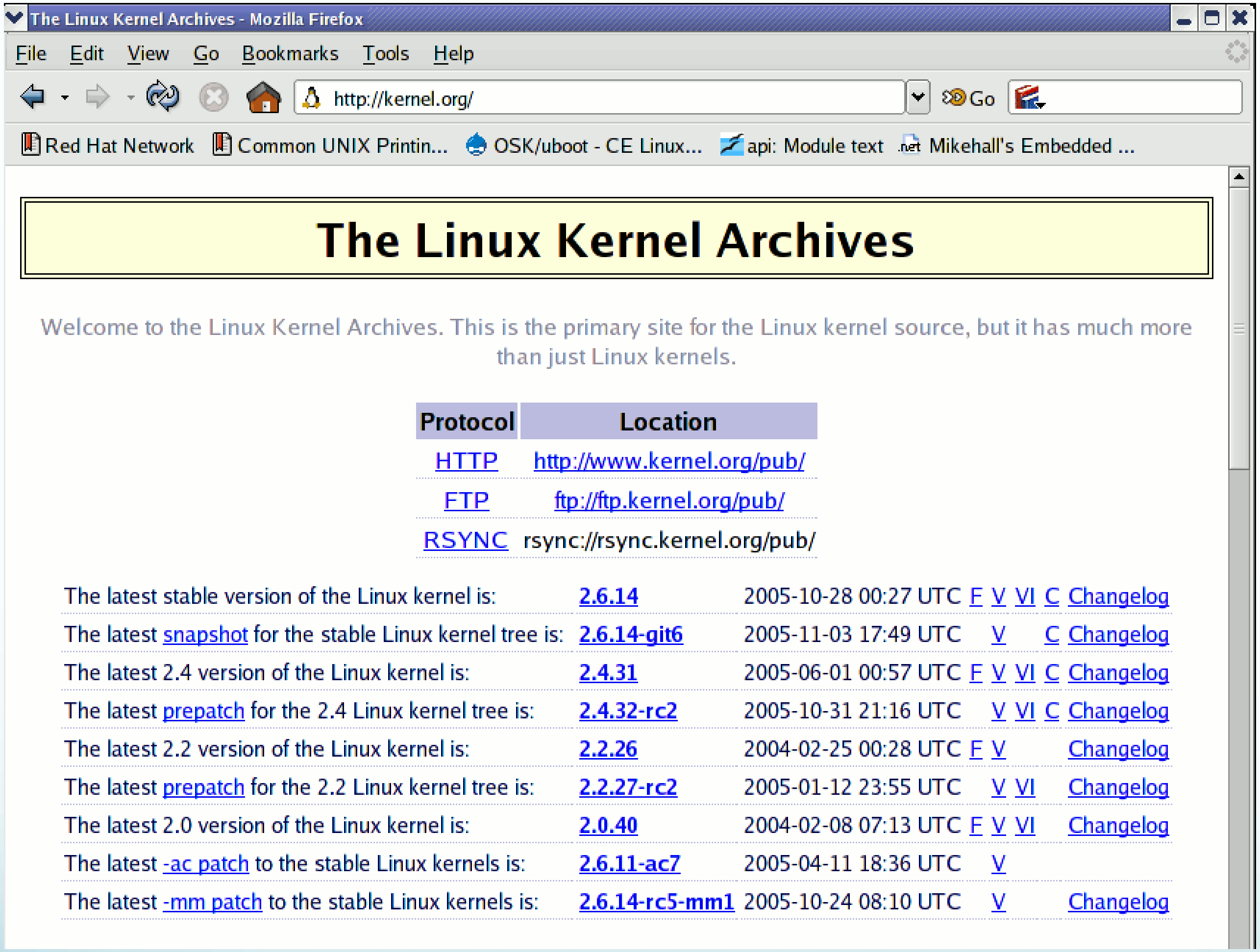
Why are these sources so big?

Because they include thousands of device drivers, many network protocols, support many architectures and filesystems...

The Linux core (scheduler, memory management...) is pretty small!

MABEL Technology
Tech The Best

kernel.org



The Linux Kernel Archives

Welcome to the Linux Kernel Archives. This is the primary site for the Linux kernel source, but it has much more than just Linux kernels.

Protocol	Location
HTTP	http://www.kernel.org/pub/
FTP	ftp://ftp.kernel.org/pub/
RSYNC	rsync://rsync.kernel.org/pub/

The latest stable version of the Linux kernel is:	**2.6.14**	2005-10-28 00:27 UTC	F	V	VI	C	Changelog
The latest snapshot for the stable Linux kernel tree is:	**2.6.14-git6**	2005-11-03 17:49 UTC		V		C	Changelog
The latest 2.4 version of the Linux kernel is:	**2.4.31**	2005-06-01 00:57 UTC	F	V	VI	C	Changelog
The latest prepatch for the 2.4 Linux kernel tree is:	**2.4.32-rc2**	2005-10-31 21:16 UTC		V	VI	C	Changelog
The latest 2.2 version of the Linux kernel is:	**2.2.26**	2004-02-25 00:28 UTC	F	V			Changelog
The latest prepatch for the 2.2 Linux kernel tree is:	**2.2.27-rc2**	2005-01-12 23:55 UTC		V	VI		Changelog
The latest 2.0 version of the Linux kernel is:	**2.0.40**	2004-02-08 07:13 UTC	F	V	VI		Changelog
The latest -ac patch to the stable Linux kernels is:	**2.6.11-ac7**	2005-04-11 18:36 UTC		V			
The latest -mm patch to the stable Linux kernels is:	**2.6.14-rc5-mm1**	2005-10-24 08:10 UTC		V			Changelog

Getting Linux sources: 2 possibilities

- Full sources

The easiest way, but longer to download.

Example:
http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.14.1.tar.bz2

Or patch against the previous version

Assuming you already have the full sources of the previous version

Example:
http://kernel.org/pub/linux/kernel/v2.6/patch-2.6.14.bz2 (2.6.13 to 2.6.14)
http://kernel.org/pub/linux/kernel/v2.6/patch-2.6.14.7.bz2 (2.6.14 to 2.6.14.7)

Using the patch command

- The patch command applies changes to files in the current directory:

Making changes to existing files

Creating or deleting files and directories

patch usage examples:

patch -p<n> < diff_file

cat diff_file | patch -p<n>

bzcat diff_file.bz2 | patch -p<n>

zcat diff_file.gz | patch -p<n>

n: number of directory levels to skip in the file paths

You can reverse a patch with the -R option

Embedded Linux Driver Development

Kernel Overview

Kernel Code

Linux Sources Structure (1)

arch/<arch>	Architecture specific code
arch/<arch>/mach-<mach>	Machine / board specific code
COPYING	Linux copying conditions (GNU GPL)
CREDITS	Linux main contributors
crypto/	Cryptographic libraries
Documentation/	Kernel documentation. Don't miss it!
drivers/	All device drivers (drivers/usb/, etc.)
fs/	Filesystems (fs/ext3/, etc.)
include/	Kernel headers
include/asm-<arch>	Architecture and machine dependent headers
include/linux	Linux kernel core headers
init/	Linux initialization (including main.c)
ipc/	Code used for process communication

Linux Sources Structure (2)

kernel/	Linux kernel core (very small!)
lib/	Misc library routines (zlib, crc32...)
MAINTAINERS	Maintainers of each kernel part. Very useful!
Makefile	Top Linux makefile (sets arch and version)
mm/	Memory management code (small too!)
net/	Network support code (not drivers)
README	Overview and building instructions
REPORTING-BUGS	Bug report instructions
scripts/	Scripts for internal or external use
security/	Security model implementations (SELinux...)
sound/	Sound support code and drivers
usr/	Early user-space code (initramfs)

LXR: Linux Cross Reference

- http://sourceforge.net/projects/lxr
- Generic source indexing tool and code browser

Web server based
 Very easy and fast to use

Identifier or text search available

Very easy to find the declaration, implementation or usages of symbols

Supports C and C++

Supports huge code projects such as the Linux kernel (260 M in Apr. 2006)

Takes a little bit of time and patience to setup (configuration, indexing, server configuration).

Initial indexing quite slow:
 Linux 2.6.11: 1h 40min on P4 M 1.6 GHz, 2 MB cache

You don't need to set up LXR by yourself. Use our http://lxr.free-electrons.com server! Other servers available on the Internet: http://free-electrons.com/community/kernel/lxr/

MABEL Technology
Tech The Best

Implemented in C

Implemented in C like all Unix systems.
(C was created to implement the first Unix systems)

A little Assembly is used too:
CPU and machine initialization, critical library routines.

See http://www.tux.org/lkml/#s15-3
for reasons for not using C++
(main reason: the kernel requires efficient code).

Compiled with GNU C

Need GNU C extensions to compile the kernel.
So, you cannot use any ANSI C compiler!

Some GNU C extensions used in the kernel:

- Inline C functions
- Inline assembly
- Structure member initialization
  in any order (also in ANSI C99)
- Branch annotation (see next page)

Help gcc Optimize Your Code!

Use the likely and unlikely statements (include/linux/compiler.h)

Example:

```
if (unlikely(err)) {
    ...
}
```

The GNU C compiler will make your code faster for the most likely case.

Used in many places in kernel code! Don't forget to use these statements!

No C library

The kernel has to be standalone and can't use user-space code.
User-space is implemented on top of kernel services, not the opposite.
Kernel code has to supply its own library implementations (string utilities, cryptography, uncompression...)

So, you can't use standard C library functions in kernel code. (printf(), memset(), malloc()...).
You can also use kernel C headers.

Fortunately, the kernel provides similar C functions for your convenience, like printk(), memset(), kmalloc()...

Managing Endianess

- Linux supports both little and big endian architectures

Each architecture defines __BIG_ENDIAN or __LITTLE_ENDIAN in <asm/byteorder.h>
Can be configured in some platforms supporting both.

To make your code portable, the kernel offers conversion macros (that do nothing when no conversion is needed).
Most useful ones:

```
u32 cpu_to_be32(u32);     // CPU byte order to big endian
u32 cpu_to_le32(u32); // CPU byte order to little endian
u32 be32_to_cpu(u32);     // Little endian to CPU byte order
u32 le32_to_cpu(u32); // Big endian to CPU byte order
```

MABEL Technology
Tech The Best

Kernel Coding Guidelines

Never use floating point numbers in kernel code. Your code may be run on a processor without a floating point unit (like on arm). Floating point can be emulated by the kernel, but this is very slow.

Define all symbols as static, except exported ones (avoid name space pollution)

All system calls return negative numbers (error codes) for errors:

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

See Documentation/CodingStyle for more guidelines

Kernel Stack

- Very small and fixed stack.
  - 2 page stack (8k), per task.
  - Or 1 page stack, per task and one for interrupts.
  - Chosen in build time via menu.
  - Not for all architectures
- For some architectures, the kernel provides debug facility to detect stack overruns.

Example - Linked Lists

Many constructs use doubly-linked lists.

List definition and initialization:

struct list_head mylist = **LIST_HEAD_INIT**(mylist);

or

LIST_HEAD(mylist);

or

INIT_LIST_HEAD(&mylist);

List Manipulation

List definition and initialization:

void **list_add**(struct list_head *new, struct list_head *head);

void **list_add_tail**(struct list_head *new, struct list_head *head);

void **list_del**(struct list_head *entry);

void **list_del_init**(struct list_head *entry);

void **list_move**(struct list_head *list, struct list_head *head);

void **list_add_tail**(struct list_head *list, struct list_head *head);

List Manipulation (cont.)

List splicing and query:

void **list_splice**(struct list_head *list, struct list_head *head);

void **list_add_splice_init**(struct list_head *list, struct list_head *head);

void **list_empty**(struct list_head *head);

In 2.6, there are variants of these API's for RCU protected lists (see section about Locks ahead).

List Iteration

Lists also have iterator macros defined:

list_for_each(pos, head);

list_for_each_prev(pos, head);

list_for_each_safe(pos, n, head);

list_for_each_entry(pos, head, member);

Example:

struct mydata *pos;

list_for_each_entry(pos, head, dev_list) {

pos->some_data = 0777;

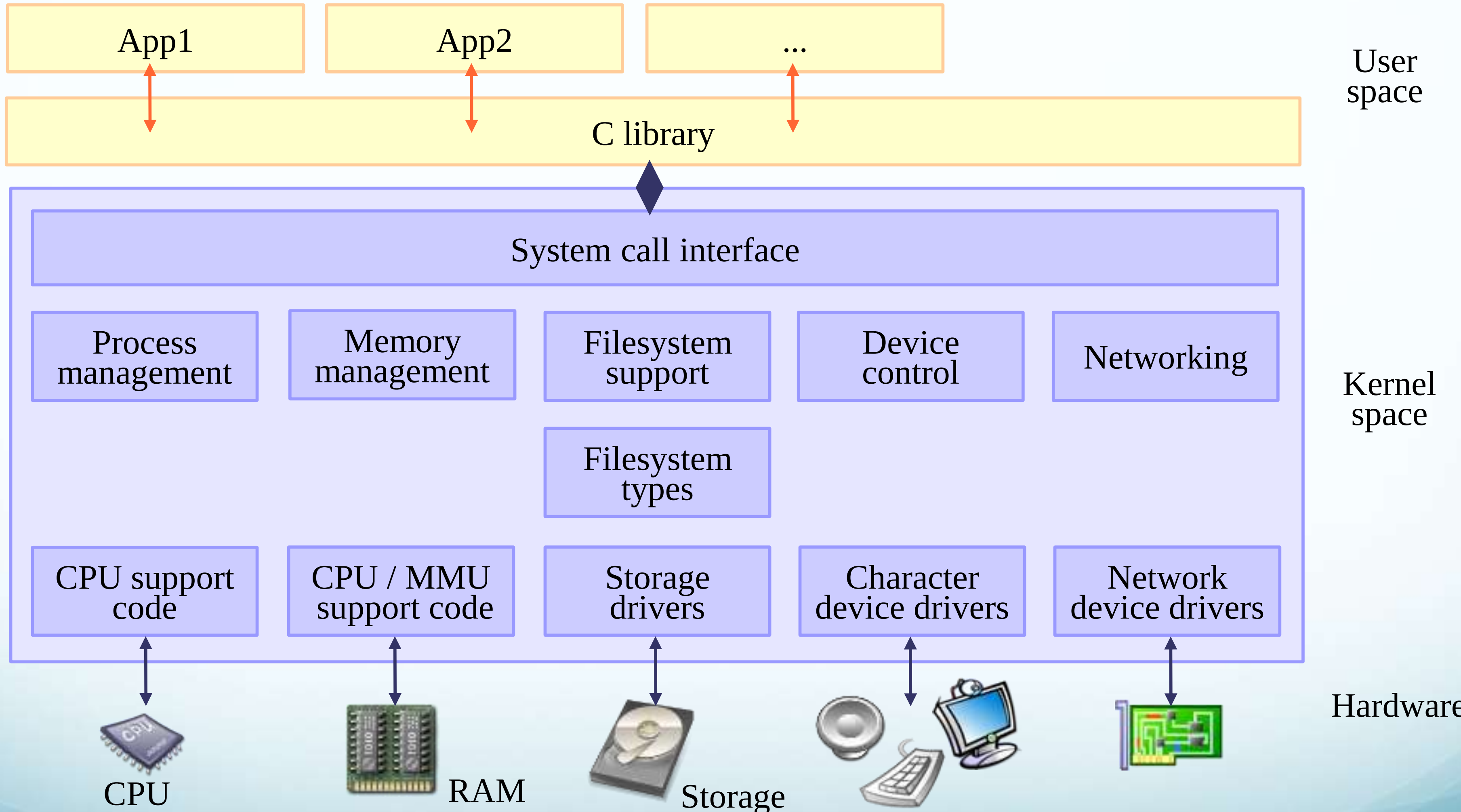
}

Embedded Linux Driver Development

Kernel Overview

Supervisor mode and the sys call interface

Kernel Architecture



MABEL Technology
Tech The Best

Kernel Mode vs. User Mode

All modern CPUs support a dual mode of operation:

- User mode, for regular tasks.
- Supervisor (or privileged) mode, for the kernel.

The mode the CPU is in determines which instructions the CPU is willing to execute:

- “Sensitive” instructions will not be executed when the CPU is in user mode.

The CPU mode is determined by one of the CPU registers, which stores the current “Ring Level”

- 0 for supervisor mode, 3 for user mode, 1-2 unused by Linux.

The System Call Interface

When a user space tasks needs to use a kernel service, it will make a “System Call”.

The C library places parameters and number of system call in registers and then issues a special trap instruction.

The trap atomically changes the ring level to supervisor mode and the sets the instruction pointer to the kernel.

The kernel will find the required system called via the system call table and execute it.

Returning from the system call does not require a special instruction, since in supervisor mode the ring level can be changed directly.

System Calls

- System calls provide a layer between the hardware and user-space processes
  - Abstracted hardware interface for user-space
  - Ensure system security and stability
  - Provide the only entry point to the kernel (from software)
- The system call interface is part of the C library
- Usually return 0 on success and negative value for error
- Context switch

System Calls

- More than 300 (architecture dependent)
  - some calls aren't implemented for all platforms and call sys_ni_ syscall
- Each system call has a unique number
  - usually Defined in unistd*.h, for example
    - linux/include/asm-mips/unistd.h
    - linux/arch/arm/include/asm/unistd.h
    - linux/arch/sparc/include/asm/unistd_32.h
- Naming convention: sys_[name]
- Invoked using software interrupt mechanism (platform depended)

Make a call

Using software interrupt instruction (int, syscall, swi etc.)

Input and output parameters using hardware registers

Example (MIPS)

```
int mygetpid()
{
    asm volatile(
            “li $v0,4020\n\t”
            “syscall\n\t”
    );
}
```

System call table

Architecture depended table to hold system calls (function pointers)

Can be found in linux/arch/[arch]/kernel/...

```
.macro  syscalltable
sys     sys_syscall             8       /* 4000 */
sys     sys_exit                1
sys     sys_fork                0
sys     sys_read                3
sys     sys_write               3
sys     sys_open                3       /* 4005 */
sys     sys_close               1
sys     sys_waitpid             3
sys     sys_creat               2
sys     sys_link                2
sys     sys_unlink              1       /* 4010 */
sys     sys_execve              0
sys     sys_chdir               1
sys     sys_time                1
sys     sys_mknod               3
sys     sys_chmod               2       /* 4015 */
sys     sys_lchown              3
sys     sys_ni_syscall          0
```

System Initialization - Example(MIPS)

The first phase in system initialization is platform depended code – linux/arch/mips/kernel/head.S

Do some basic hardware setup including TLB

Call BSP specific code (kernel_entry_setup)

Call Platform in-depended code start_kernel

linux/init/main.c

To setup exception handlers (including software interrupt) it calls to trap_init

The software interrupt handler is handle_sys

linux/arch/mips/kernel/scall32-o32.S

```
NESTED(handle_sys, PT_SIZE, sp)
        .set    noat
        SAVE_SOME
        TRACE_IRQS_ON_RELOAD
        STI
        .set    at

        lw      t1, PT_EPC(sp)          # skip syscall on return

        subu    v0, v0, __NR_O32_Linux  # check syscall number
        sltiu   t0, v0, __NR_O32_Linux_syscalls + 1
        addiu   t1, 4                   # skip to next instruction
        sw      t1, PT_EPC(sp)
        beqz    t0, illegal_syscall

        sll     t0, v0, 3
        la      t1, sys_call_table
        addu    t1, t0
        lw      t2, (t1)                # syscall routine
        lw      t3, 4(t1)               # >= 0 if we need stack arguments
        beqz    t2, illegal_syscall

        sw      a3, PT_R26(sp)          # save a3 for syscall restarting
        bgez    t3, stackargs

stack_done:
        lw      t0, TI_FLAGS($28)       # syscall tracing enabled?
        li      t1, _TIF_SYSCALL_TRACE | _TIF_SYSCALL_AUDIT
        and     t0, t1
        bnez    t0, syscall_trace_entry # -> yes

        jalr    t2                      # Do The Real Thing (TM)

        li      t0, -EMAXERRNO - 1      # error?
        sltu    t0, t0, v0
        sw      t0, PT_R7(sp)           # set error flag
        beqz    t0, 1f

        negu    v0                      # error
        sw      v0, PT_R0(sp)           # set flag for syscall
                                        # restarting
1:      sw      v0, PT_R2(sp)           # result
```

Linux Error Codes

- Try to report errors with error numbers as accurate as possible! Fortunately, macro names are explicit and you can remember them quickly.

Generic error codes:
include/asm-generic/errno-base.h

Platform specific error codes:
include/asm/errno.h

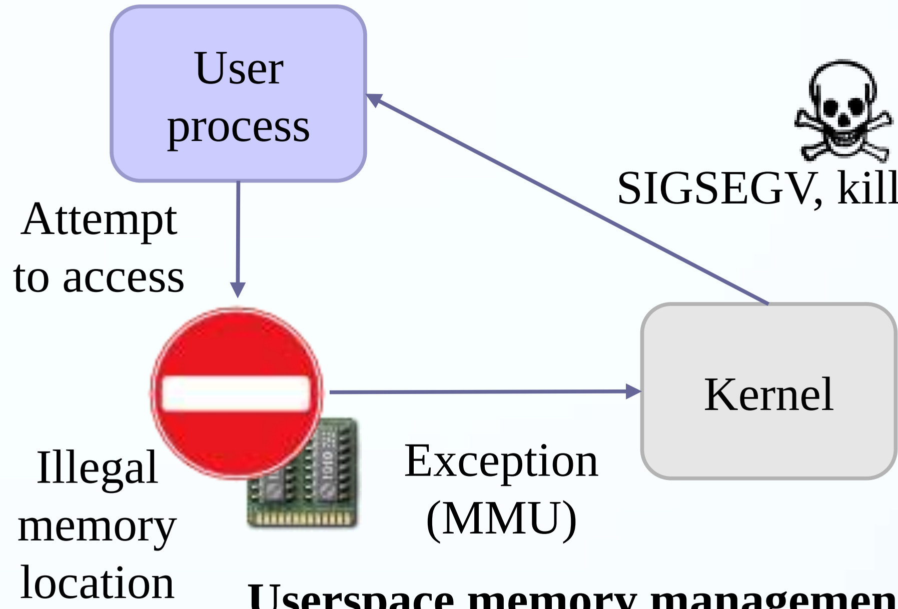
Kernel memory constraints

- Who can look after the kernel?

No memory protection
Accessing illegal memory locations result in (often fatal) kernel oopses.

Fixed size stack (8 or 4 KB)
Unlike in userspace,
no way to make it grow.

Kernel memory can't be swapped out (for the same reasons).



Userspace memory management
Used to implement:
- memory protection
- stack growth
- memory swapping to disk

Linux Internals

Compiling and booting Linux

Kernel configuration

Kernel configuration overview

Makefile edition

Setting the version and target architecture if needed

Kernel configuration: defining what features to include in the kernel:

make [config|xconfig|gconfig|menuconfig|oldconfig]

Kernel configuration file (Makefile syntax) stored in the .config file at the root of the kernel sources

Distribution kernel config files usually released in /boot/

Makefile changes

To identify your kernel image with others build from the same sources, use the EXTRAVERSION variable:

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 15
EXTRAVERSION = -acme1
```

uname -r will return:
2.6.15-acme1

make xconfig

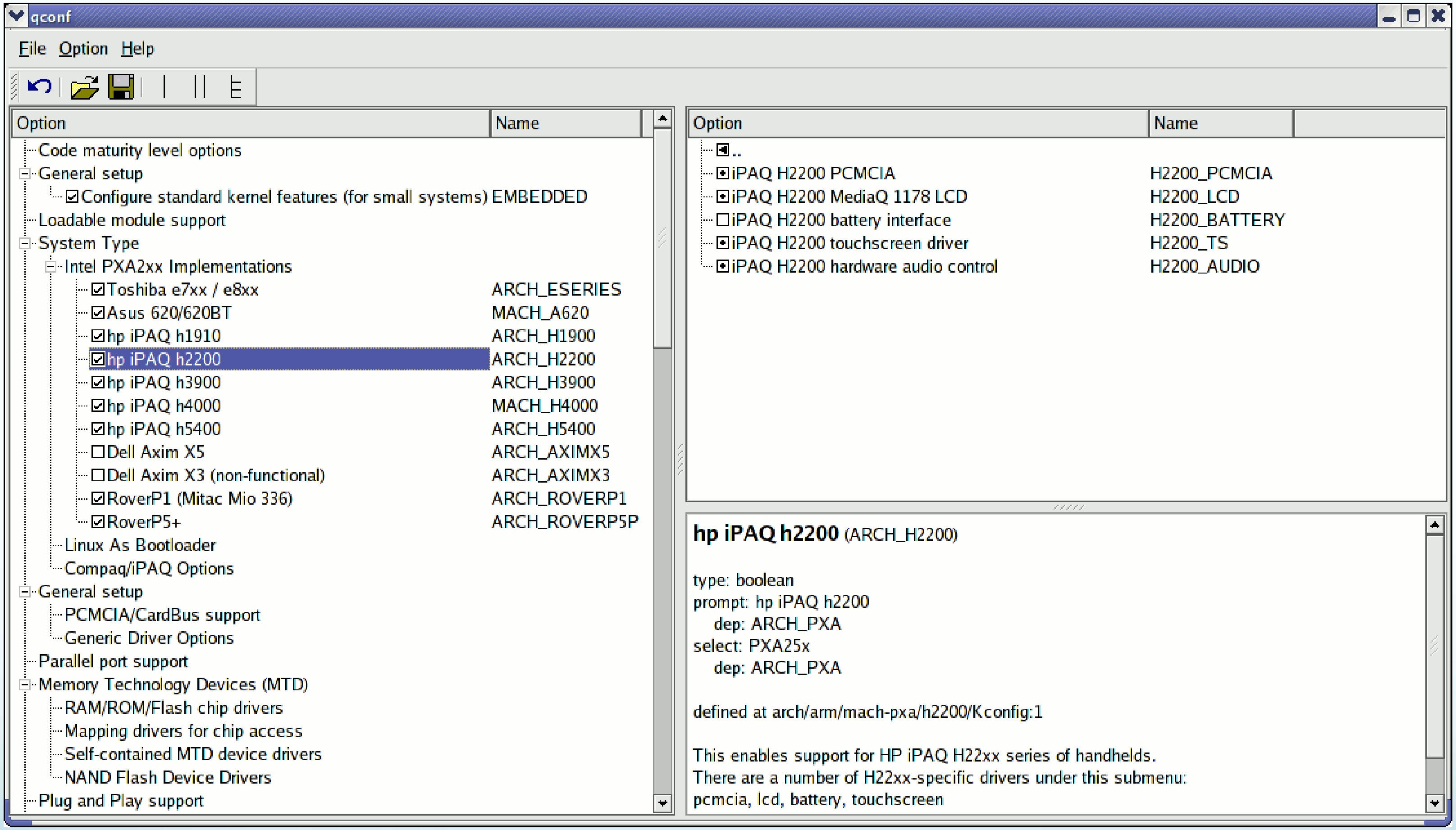
- make xconfig

New Qt configuration interface for Linux 2.6.
Much easier to use than in Linux 2.4!

Make sure you read
help -> introduction: useful options!

File browser: easier to load configuration files

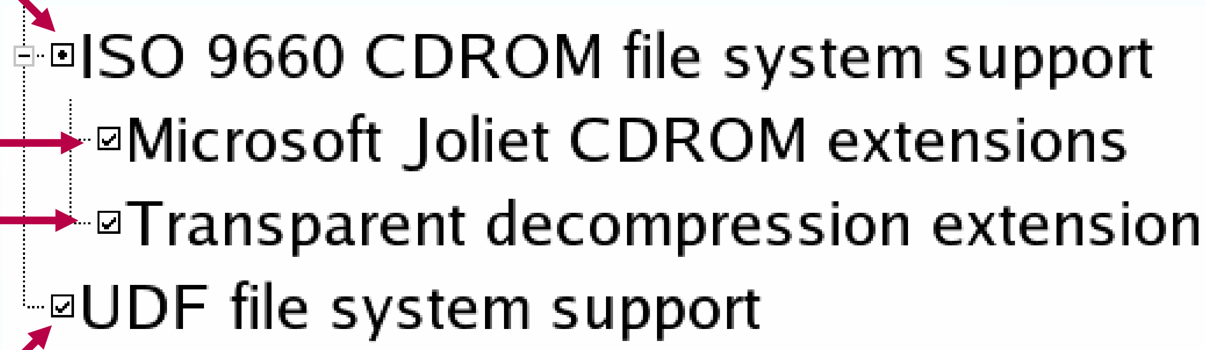
make xconfig



Compiling statically or as a module

Compiled as a module (separate file)
CONFIG_ISO9660_FS=m

Driver options
CONFIG_JOLIET=y
CONFIG_ZISOFS=y



Compiled statically in the kernel
CONFIG_UDF_FS=y

make config / menuconfig / gconfig

make config

Asks you the questions 1 by 1. Extremely long!

make menuconfig

Same old text interface as in Linux 2.4.
Useful when no graphics are available.
Pretty convenient too!

make gconfig

New GTK based graphical configuration interface.
Functionality similar to that of make xconfig.

make oldconfig

- make oldconfig

Needed very often!

Useful to upgrade a .config file from an earlier kernel release

Issues warnings for obsolete symbols

Asks for values for new symbols

If you edit a .config file by hand, it's strongly recommended to run make oldconfig afterwards!

make allnoconfig

- make allnoconfig

Only sets strongly recommended settings to y.

Sets all other settings to n.

Very useful in embedded systems to select only the minimum required set of features and drivers.

Much more convenient than unselecting hundreds of features one by one!

make help

- make help

Lists all available make targets

Useful to get a reminder, or to look for new or advanced options!

Linux Internals

Compiling and booting Linux

Compiling the kernel

Compiling and installing the kernel

- Compiling step

make

Install steps (logged as root!)

make install

make modules_install

Generated files

- Created when you run the make command

vmlinux
Raw Linux kernel image, non compressed.

arch/<arch>/boot/zImage (default image on arm)
zlib compressed kernel image

arch/<arch>/boot/bzImage (default image on i386)
Also a zlib compressed kernel image.
Caution: bz means “big zipped” but not “bzip2 compressed”!
(bzip2 compression support only available on i386 as a tactical patch. Not very attractive for small embedded systems though: consumes 1 MB of RAM for decompression).

MABEL Technology
Tech The Best

Files created by make install

/boot/vmlinuz-<version>
Compressed kernel image. Same as the one in arch/<arch>/boot

/boot/System.map-<version>
Stores kernel symbol addresses

/boot/initrd-<version>.img (when used by your distribution)
Initial RAM disk, storing the modules you need to mount your root filesystem. make install runs mkinitrd for you!

/etc/grub.conf or /etc/lilo.conf
make install updates your bootloader configuration files to support your new kernel! It reruns /sbin/lilo if LILO is your bootloader.

Not relevant for embedded systems.

MABEL Technology
Tech The Best

Files created by make modules_install (1)

- /lib/modules/<version>/: Kernel modules + extras

build/
Everything needed to build more modules for this kernel: Makefile,
.config file, module symbol information (module.symVers),
kernel headers (include/ and include/asm/)

kernel/
Module .ko (Kernel Object) files, in the same directory structure as in the sources.

Files created by make modules_install (2)

- /lib/modules/<version>/ (continued)

modules.alias
    Module aliases for module loading utilities. Example line:
    alias sound-service-?-0 snd_mixer_oss

modules.dep
    Module dependencies (see the Loadable kernel modules section)

modules.symbols
    Tells which module a given symbol belongs to.

All the files in this directory are text files.
    Don't hesitate to have a look by yourself!

Linux Internals

Compiling and booting Linux

Overall system startup

Linux booting sequence

Bootloader
- Executed by the hardware at a fixed location in ROM / Flash
- Initializes support for the device where the kernel image is found (local storage, network, removable media)
- Loads the kernel image in RAM
- Executes the kernel image (with a specified command line)

Kernel
- Uncompresses itself
- Initializes the kernel core and statically compiled drivers (needed to access the root filesystem)
- Mounts the root filesystem (specified by the `init` kernel parameter)
- Executes the first userspace program

First userspace program
- Configures userspace and starts up system services

Initramfs features and advantages (1)

Root file system built in in the kernel image (embedded as a compressed cpio archive)

Very easy to create (at kernel build time). No need for root permissions (for mount and mknod).

Compared to init ramdisks, just 1 file to handle.

Always present in the Linux 2.6 kernel (empty by default).

Just a plain compressed cpio archive. Neither needs a block nor a filesystem driver.

Initramfs features and advantages (2)

ramfs: implemented in the file cache.
- No duplication in RAM, no filesystem layer to manage.
- Just uses the size of its files. Can grow if needed.

Loaded by the kernel earlier.
- More initialization code moved to user-space!

Simpler to mount complex filesystems from flexible userspace scripts rather than from rigid kernel code. More complexity moved out to user-space!

No more magic naming of the root device.
- pivot_root no longer needed.

Initramfs features and advantages (3)

Possible to add non GPL files (firmware, proprietary drivers) in the filesystem. This is not linking, just file aggregation (not considered as a derived work by the GPL).

Possibility to remove these files when no longer needed.

Still possible to use ramdisks.

More technical details about initramfs:
see Documentation/filesystems/ramfs-rootfs-initramfs.txt
and Documentation/early-userspace/README in kernel sources.

See also http://www.linuxdevices.com/articles/AT4017834659.html for a nice overview of initramfs (by Rob Landley, new Busybox maintainer).

Linux Internals

Compiling and booting Linux

Kernel booting

Kernel command line parameters

- As most C programs, the Linux kernel accepts command line arguments

Kernel command line arguments are part of the bootloader configuration settings.

Useful to configure the kernel at boot time, without having to recompile it.

Useful to perform advanced kernel and driver initialization, without having to use complex user-space scripts.

Kernel command line example

- HP iPAQ h2200 PDA booting example:

- root=/dev/ram0 \ Root filesystem (first ramdisk)
  rw \ Root filesystem mounting mode
  init=/linuxrc \ First userspace program
  console=ttyS0,115200n8 \ Console (serial)
  console=tty0 \ Other console (framebuffer)
  ramdisk_size=8192 \ Misc parameters...
  cachepolicy=writethrough

- Hundreds of command line parameters described on Documentation/kernel-parameters.txt

Booting variants

XIP (Execute In Place)

The kernel image is directly executed from the storage

Can be faster and save RAM
However, the kernel image can't be compressed

No initramfs / initrd

Directly mounting the final root filesystem
(root kernel command line option)

No new root filesystem

Running the whole system from the initramfs

rootfs on NFS

- Once networking works, your root filesystem could be a directory on your GNU/Linux development host, exported by NFS (Network File System). This is very convenient for system development:

Makes it very easy to update files (driver modules in particular) on the root filesystem, without rebooting. Much faster than through the serial port.

Can have a big root filesystem even if you don't have support for internal or external storage yet.

The root filesystem can be huge. You can even build native compiler tools and build all the tools you need on the target itself (better to cross-compile though).

First user-space program

Specified by the init kernel command line parameter

Executed at the end of booting by the kernel

Takes care of starting all other user-space programs
(system services and user programs).

Gets the 1 process number (pid)
- Parent or ancestor of all user-space programs
- The system won't let you kill it.

Only other user-space program called by the kernel:
/sbin/hotplug

The init program

/sbin/init is the second default init program

Takes care of starting system services, and eventually the user interfaces (sshd, X server...)

Also takes care of stopping system services

Lightweight, partial implementation available through busybox

See the Init runlevels annex section for more details about starting and stopping system services with init.

However, simple startup scripts are often sufficient in embedded systems.

Linux Internals

Driver Development

Loadable Kernel Modules

Loadable Kernel Modules (1)

Modules: add a given functionality to the kernel (drivers, filesystem support, and many others).

Can be loaded and unloaded at any time, only when their functionality is need. Once loaded, have full access to the whole kernel. No particular protection.

Useful to keep the kernel image size to the minimum (essential in GNU/Linux distributions for PCs).

Loadable Kernel Modules (2)

Useful to support incompatible drivers (either load one or the other, but not both).

Useful to deliver binary-only drivers (bad idea) without having to rebuild the kernel.

Modules make it easy to develop drivers without rebooting: load, test, unload, rebuild, load...

Modules can also be compiled statically into the kernel.

Hello Module

MABEL Technology
Tech The Best

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

static int __init hello_init(void)
{
   printk(KERN_ALERT "Good morrow");
   printk(KERN_ALERT "to this fair assembly.\n");
   return 0;
}

static void __exit hello_exit(void)
{
   printk(KERN_ALERT "Alas, poor world, what treasure");
   printk(KERN_ALERT "hast thou lost!\n");
}

module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Greeting module");
MODULE_AUTHOR("William Shakespeare");
```

__init:
removed after initialization (static kernel or module).

__exit: discarded when module compiled statically into the kernel.

Example available on http://free-electrons.com/doc/c/hello.c

Module License Usefulness

Used by kernel developers to identify issues coming from proprietary drivers, which they can't do anything about.

Useful for users to check that their system is 100% free.

Useful for GNU/Linux distributors for their release policy checks.

Possible Module License Strings

Available license strings explained in include/linux/module.h

GPL

GNU Public License v2 or later

GPL v2

GNU Public License v2

GPL and additional rights

Dual BSD/GPL

GNU Public License v2 or BSD license choice

Dual MPL/GPL

GNU Public License v2 or Mozilla license choice

Proprietary

Non free products

Compiling a Module

The below Makefile should be reusable for any Linux 2.6 module.

Just run make to build the hello.ko file

Caution: make sure there is a [Tab] character at the beginning of the $(MAKE) line (make syntax)

```
# Makefile for the hello module

obj-m := hello.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
	$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
```

[Tab]!
(no spaces)

Either
- full kernel source directory (configured and compiled)
- or just kernel headers directory (minimum needed)

Example available on http://free-electrons.com/doc/c/Makefile

Kernel Log

Printing to the kernel log is done via the printk() function.

The kernel keeps the messages in a circular buffer (so that doesn't consume more memory with many messages).

Kernel log messages can be accessed from user space through system calls, or through /proc/kmsg

Kernel log messages are also displayed in the system console.

printk()

The printk function:

- Similar to stdlib's printf(3)
- No floating point format.
- Log message are prefixed with a “<0>”, where the number denotes severity, from 0 (most severe) to 7.
- Macros are defined to be used for severity levels: KERN_EMERG, KERN_ALERT, KERT_CRIT, KERN_ERR, KERN_WARNING, KERN_NOTICE, KERN_INFO, KERN_DEBUG.
- Usage example:

printk(KERN_DEBUG “Hello World number %d\n”, num);

Accessing the Kernel Log

Many ways are available!

Watch the system console

syslogd/klogd
- Daemon gathering kernel messages
- in /var/log/messages
- Follow changes by running:
- tail -f /var/log/messages
- Caution: this file grows!
- Use logrotate to control this

dmesg
- Found in all systems
- Displays the kernel log buffer

logread
- Same. Often found in small embedded systems with no /var/log/messages or no dmesg. Implemented by Busybox.

cat /proc/kmsg
- Waits for kernel messages and displays them.
- Useful when none of the above user space programs are available (tiny system)

Using the Module

- Need to be logged as root

Load the module:

insmod ./hello.ko

You will see the following in the kernel log:

Good morrow
to this fair assembly

Now remove the module:

rmmod hello

You will see:

Alas, poor world, what treasure
hast thou lost!

Module Utilities (1)

modinfo <module_name>

modinfo <module_path>.ko

Gets information about a module: parameters, license, description. Very useful before deciding to load a module or not.

insmod <module_name>

insmod <module_path>.ko

Tries to load the given module, if needed by searching for its .ko file throughout the default locations (can be redefined by the MODPATH environment variable).

Module Utilities (2)

modprobe <module_name>

Most common usage of modprobe: tries to load all the modules the given module depends on, and then this module. Lots of other options are available.

lsmod

Displays the list of loaded modules

Compare its output with the contents of /proc/modules!

Module Utilities (3)

rmmod <module_name>

Tries to remove the given module

modprobe -r <module_name>

Tries to remove the given module and all dependent modules
(which are no longer needed after the module removal)

MABEL Technology
Tech The Best

Module Dependencies

Module dependencies stored in
/lib/modules/<version>/modules.dep

They don't have to be described by the module writer.

They are automatically computed during kernel building from module exported symbols. module2 depends on module1 if module2 uses a symbol exported by module1.

You can update the modules.dep file by running (as root)
depmod -a [<version>]

MABEL Technology
Tech The Best

Linux Internals

Driver Development
Module Parameters

Hello Module with Parameters

```
/* hello_param.c */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/moduleparam.h>

MODULE_LICENSE("GPL");

/* A couple of parameters that can be passed in: how many times we say
   hello, and to whom */

static char *whom = "world";
module_param(whom, charp, 0);

static int howmany = 1;
module_param(howmany, int, 0);

static int __init hello_init(void)
{
    int i;
    for (i = 0; i < howmany; i++)
            printk(KERN_ALERT "(%d) Hello, %s\n", i, whom);
    return 0;
}

static void __exit hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel %s\n", whom);
}

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

Thanks to Jonathan Corbet for the example!

Example available on http://free-electrons.com/doc/c/hello_param.c

Passing Module Parameters

Through insmod or modprobe:

insmod ./hello_param.ko howmany=2 whom=universe

Through modprobe
after changing the /etc/modprobe.conf file:

options hello_param howmany=2 whom=universe

Through the kernel command line, when the module is built statically into the kernel:

options hello_param.howmany=2 hello_param.whom=universe

module name

module parameter name

module parameter value

Declaring a Module Parameter

- #include <linux/moduleparam.h>

- module_param(
    name, /* name of an already defined variable */
    type, /* either byte, short, ushort, int, uint, long, ulong, charp, bool or invbool (checked at compile time!) */
    perm /* for /sys/module/<module_name>/<param> 0: no such module parameter value file */
  );

- <u>Example</u>

- int irq=5;
  module_param(irq, int, S_IRUGO);

Declaring a Module Parameter Array

- #include <linux/moduleparam.h>

- module_param_array(
```
    name,        /* name of an already defined array */
    type,        /* same as in module_param */
    num,         /* address to put number of elements in the array, or NULL */
    perm         /* same as in module_param */
);
```

- <u>Example</u>

- static int count;
  static int base[MAX_DEVICES] = { 0x820, 0x840 };
  module_param_array(base, int, &count, 0);

MABEL Technology
Tech The Best

Linux Internals

Driver Development
Memory Management

Physical Memory

In ccNUMA[1] machines:

- The memory of each node is represented in pg_data_t
- These memories are linked into pgdat_list

In uniform memory access systems:

- There is just one pg_data_t named contig_page_data

If you don't know which of these is your machine, you're using a uniform memory access system :-)

[1] **ccNUMA**: Cache Coherent Non Uniform Memory Access

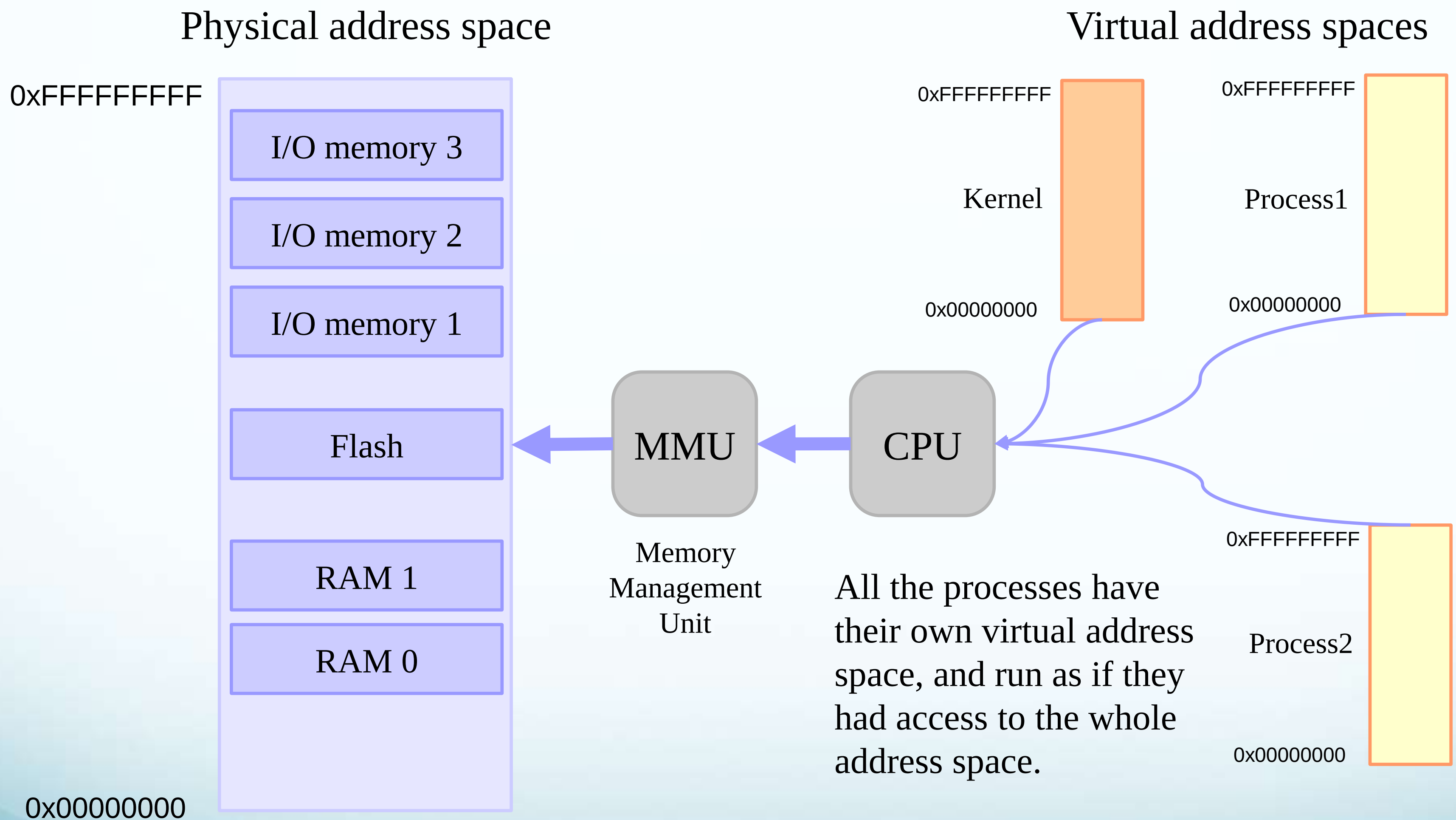
Memory Zones

Each pg_data_t is split to three zones

Each zone has different properties:

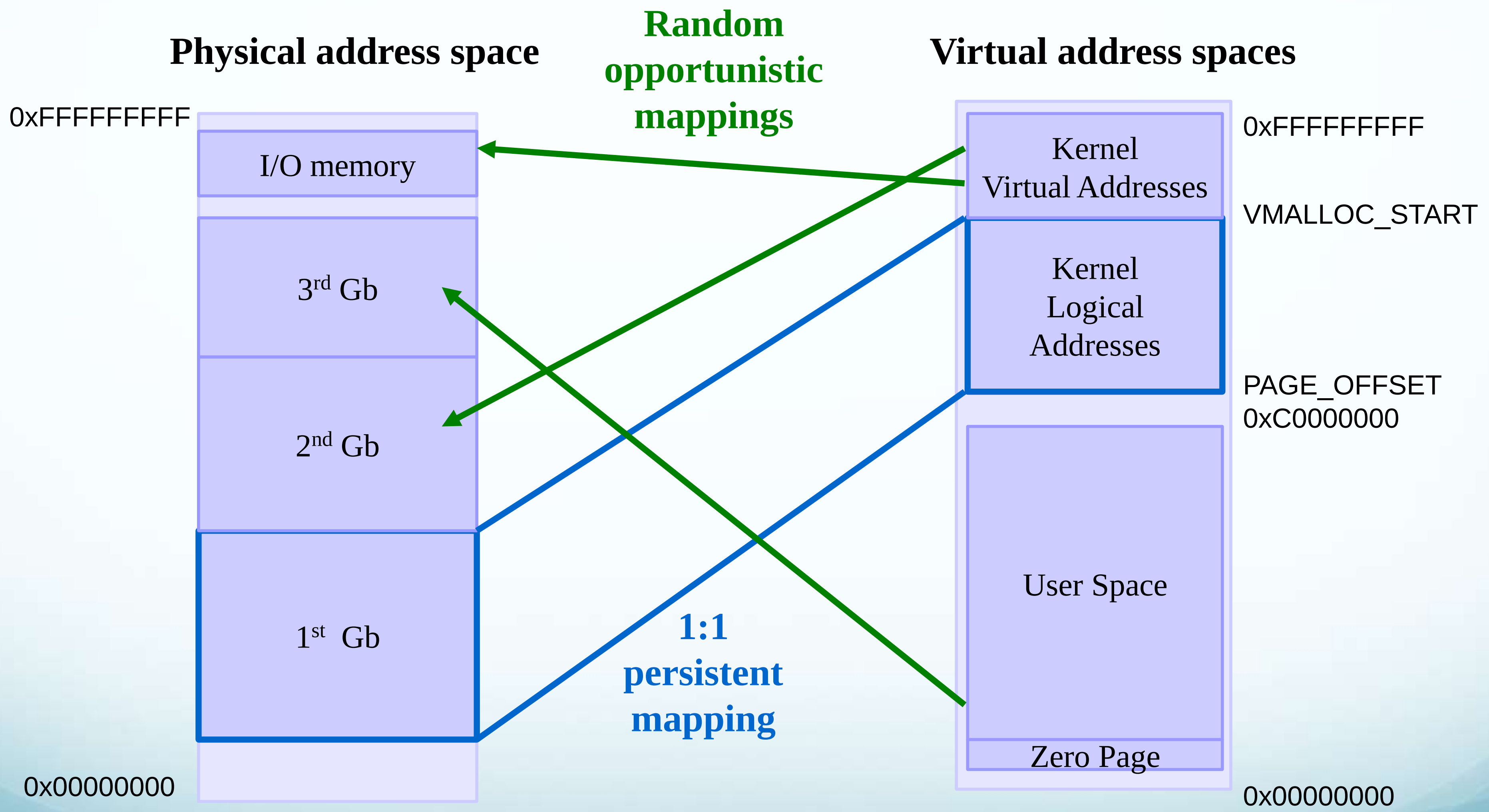
- **ZONE_DMA**
  - DMA operations on address limited busses is possible
- **ZONE_NORMAL**
  - Maps directly to linear addressing (<~1Gb on i386)
  - Always mapped to kernel space.
- **ZONE_HIMEM**
  - Rest of memory.
  - Mapped into kernel space on demand.

Physical and virtual memory



All the processes have their own virtual address space, and run as if they had access to the whole address space.

3:1 Virtual Memory Map



Address Types

Physical address

- Physical memory as seen from the CPU, with out MMU[1] translation.

Bus address

- Physical memory as seen from device bus.
- May or may not be virtualized (via IOMMU, GART, etc).

Virtual address

- Memory as seen from the CPU, with MMU[1] translation.

[1] **MMU**: Memory Management Unit

Translation Macros

bus_to_phys(address)

phys_to_bus(address)

phys_to_virt(address)

virt_to_phys(address)

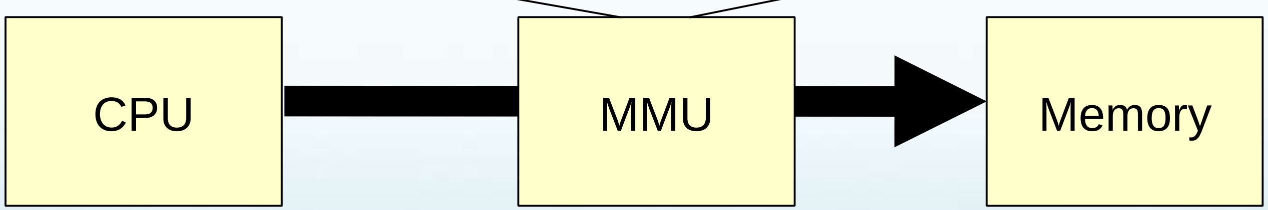
bus_to_virt(address)

virt_to_bus(address)

...

The MMU

Task	Virtual	Physical	Permission
12	0x8000	0x5340	RWX
12	0x8001	0x1000	RX
15	0x8000	0x3390	RX



kmalloc and kfree

Basic allocators, kernel equivalents of glibc's malloc and free.

```
static inline void *kmalloc(size_t size, int flags);
```

size: number of bytes to allocate
flags: priority (see next page)

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

Example:

```
data = kmalloc(sizeof(*data), GFP_KERNEL);
...
kfree(data);
```

kmalloc features

Quick (unless it's blocked waiting for memory to be freed).

Doesn't initialize the allocated area.
You can use kcalloc or kzalloc to get zeroed memory.

The allocated area is contiguous in physical RAM.

Allocates by 2^n sizes, and uses a few management bytes.
So, don't ask for 1024 when you need 1000! You'd get 2048!

Caution: drivers shouldn't try to kmalloc
more than 128 KB (upper limit in some architectures).

Main kmalloc flags (1)

- Defined in include/linux/gfp.h (GFP: get_free_pages)

GFP_KERNEL
Standard kernel memory allocation. May block. Fine for most needs.

GFP_ATOMIC
Allocated RAM from interrupt handlers or code not triggered by user processes. Never blocks.

GFP_USER
Allocates memory for user processes. May block. Lowest priority.

Main kmalloc flags (2)

Extra flags (can be added with |)

__GFP_DMA
Allocate in DMA zone

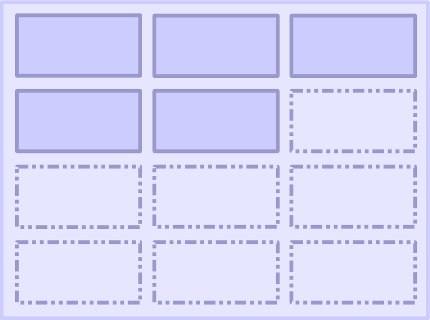
__GFP_REPEAT
Ask to try harder. May still block, but less likely.

__GFP_NOFAIL
Must not fail. Never gives up.
Caution: use only when mandatory!

__GFP_NORETRY
If allocation fails, doesn't try to get free pages.

Example:
GFP_KERNEL | __GFP_DMA

Slab caches



- Also called *lookaside caches*

Slab: name of the standard Linux memory allocator

Slab caches: Objects that can hold any number of memory areas of the same size.

Optimum use of available RAM and reduced fragmentation.

Mainly used in Linux core subsystems: filesystems (open files, inode and file caches...), networking... Live stats on /proc/slabinfo.

May be useful in device drivers too, though not used so often. Linux 2.6: used by USB and SCSI drivers.

MABEL Technology
Tech The Best

Slab cache API (1)

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

Creating a cache:

```
cache = kmem_cache_create (
    name,                      /* Name for /proc/slabinfo */
    size,                      /* Cache object size */
    flags,                     /* Options: alignment, DMA... */
    constructor,/* Optional, called after each allocation */
    );
```

Slab cache API (2)

Allocating from the cache:

object = kmem_cache_alloc (cache, flags);

Freing an object:

kmem_cache_free (cache, object);

Destroying the whole cache:

kmem_cache_destroy (cache);

More details and an example in the Linux Device Drivers book: http://lwn.net/images/pdf/LDD3/ch08.pdf

Memory pools

- Useful for memory allocations that cannot fail

Kind of lookaside cache trying to keep a minimum number of pre-allocated objects ahead of time.

Use with care: otherwise can result in a lot of unused memory that cannot be reclaimed! Use other solutions whenever possible.

Memory pool API (1)

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

Mempool creation:

```
mempool = mempool_create (
    min_nr,
    alloc_function,
    free_function,
    pool_data);
```

Memory pool API (2)

Allocating objects:

```
object = mempool_alloc (pool, flags);
```

Freeing objects:

```
mempool_free (object, pool);
```

Resizing the pool:

```
status = mempool_resize (
                         pool, new_min_nr, flags);
```

Destroying the pool (caution: free all objects first!):

```
mempool_destroy (pool);
```

Memory pools using slab caches

Idea: use slab cache functions to allocate and free objects.

The mempool_alloc_slab and mempool_free_slab functions supply a link with slab cache routines.

So, you will find many code examples looking like:

```
cache = kmem_cache_create (...);
pool = mempool_create (
                min_nr,
                mempool_alloc_slab,
                mempool_free_slab,
                cache);
```

Allocating by pages

- More appropriate when you need big slices of RAM:

unsigned long get_zeroed_page(int flags);
Returns a pointer to a free page and fills it up with zeros

unsigned long __get_free_page(int flags);
Same, but doesn't initialize the contents

unsigned long __get_free_pages(int flags, unsigned long order);
Returns a pointer on a memory zone of several contiguous pages in physical RAM.
order: $\log_2$(<number_of_pages>)
maximum: 8192 KB (MAX_ORDER=11 in linux/mmzone.h)

The basic system allocator that all other rely on.

MABEL Technology
Tech The Best

Freeing pages

```
void free_page(unsigned long addr);

void free_pages(unsigned long addr,
                unsigned long order);
```

Need to use the same order as in allocation.

The Buddy System

Kernel memory page allocation follows the “Buddy” System.

Free Page Frames are allocated in powers of 2:

- If suitable page frame is found, allocate.
- Else: seek higher order frame, allocate half, keep “buddy”

When freeing page frames, coalescing occurs.

vmalloc

- vmalloc can be used to obtain contiguous memory zones in virtual address space (even if pages may not be contiguous in physical memory).

```
void *vmalloc(unsigned long size);

void vfree(void *addr);
```

Memory utilities

```
void * memset(void * s, int c, size_t count);
```

Fills a region of memory with the given value.

```
void * memcpy(void * dest,
              const void *src,
              size_t count);
```

Copies one area of memory to another.
Use memmove with overlapping areas.

Lots of functions equivalent to standard C library ones defined in include/linux/string.h

Memory management - Summary

- Small allocations

kmalloc, kzalloc (and kfree!)

slab caches

memory pools

- Bigger allocations

__get_free_page[s], get_zeroed_page, free_page[s]

vmalloc, vfree

Libc like memory utilities

memset, memcopy, memmove...

MABEL Technology
Tech The Best

Linux Internals

Driver Development

I/O Memory and Ports

Requesting I/O Ports

/proc/ioports example

```
0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0060-006f : keyboard
0070-0077 : rtc
0080-008f : dma page reg
00a0-00a1 : pic2
00c0-00df : dma2
00f0-00ff : fpu
0100-013f : pcmcia_socket0
0170-0177 : ide1
01f0-01f7 : ide0
0376-0376 : ide1
0378-037a : parport0
03c0-03df : vga+
03f6-03f6 : ide0
03f8-03ff : serial
0800-087f : 0000:00:1f.0
0800-0803 : PM1a_EVT_BLK
0804-0805 : PM1a_CNT_BLK
0808-080b : PM_TMR
0820-0820 : PM2_CNT_BLK
0828-082f : GPE0_BLK
...
```

```
struct resource *request_region(
    unsigned long start,
    unsigned long len,
    char *name);
```

Tries to reserve the given region and returns NULL if unsuccessful. Example:

```
request_region(0x0170, 8, "ide1");
```

```
void release_region(
    unsigned long start,
    unsigned long len);
```

See include/linux/ioport.h and kernel/resource.c

MABEL Technology
Tech The Best

Reading/Writing on I/O Ports

- The implementation of the below functions and the exact *unsigned* type can vary from architecture to architecture!

- <u>bytes</u>
unsigned inb(*unsigned* port);
void outb(*unsigned* char byte, *unsigned* port);

- <u>words</u>
unsigned inw(*unsigned* port);
void outw(*unsigned* char byte, *unsigned* port);

- <u>"long" integers</u>
unsigned inl(*unsigned* port);
void outl(*unsigned* char byte, *unsigned* port);

Reading/Writing Strings on I/O Ports

- Often more efficient than the corresponding C loop, if the processor supports such operations!

- <u>byte strings</u>
  void insb(*unsigned* port, void *addr, unsigned long count);
  void outsb(*unsigned* port, void *addr, unsigned long count);

- <u>word strings</u>
  void insw(*unsigned* port, void *addr, unsigned long count);
  void outsw(*unsigned* port, void *addr, unsigned long count);

- <u>long strings</u>
  void inbsl(*unsigned* port, void *addr, unsigned long count);
  void outsl(*unsigned* port, void *addr, unsigned long count);

Requesting I/O Memory

/proc/iomem example

```
00000000-0009efff : System RAM
0009f000-0009ffff : reserved
000a0000-000bffff : Video RAM area
000c0000-000cffff : Video ROM
000f0000-000fffff : System ROM
00100000-3ffadfff : System RAM
  00100000-0030afff : Kernel code
  0030b000-003b4bff : Kernel data
3ffae000-3fffffff : reserved
40000000-400003ff : 0000:00:1f.1
40001000-40001fff : 0000:02:01.0
  40001000-40001fff : yenta_socket
40002000-40002fff : 0000:02:01.1
  40002000-40002fff : yenta_socket
40400000-407fffff : PCI CardBus #03
40800000-40bfffff : PCI CardBus #03
40c00000-40ffffff : PCI CardBus #07
41000000-413fffff : PCI CardBus #07
a0000000-a0000fff : pcmcia_socket0
a0001000-a0001fff : pcmcia_socket1
e0000000-e7ffffff : 0000:00:00.0
e8000000-efffffff : PCI Bus #01
  e8000000-efffffff : 0000:01:00.0
...
```

Equivalent functions with the same interface

```
struct resource *request_mem_region(
        unsigned long start,
        unsigned long len,
        char *name);

void release_mem_region(
        unsigned long start,
        unsigned long len);
```

Mapping I/O Memory into Virtual Memory

To access I/O memory, drivers need to have a virtual address that the processor can handle.

The ioremap() functions satisfy this need:

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

   void *ioremap(unsigned long phys_addr,
                 unsigned long size);
   void iounmap(void *address);
```

Caution: check that ioremap doesn't return a NULL address!

Accessing I/O Memory

Directly reading from or writing to addresses returned by ioremap() ("pointer dereferencing") may not work on some architectures.

Use the below functions instead. They are always portable and safe:

unsigned int ioread8(void *addr); (same for 16 and 32)
void iowrite8(u8 value, void *addr); (same for 16 and 32)

To read or write a series of values:

void ioread8_rep(void *addr, void *buf, unsigned long count);
void iowrite8_rep(void *addr, const void *buf, unsigned long count);

Other useful functions:

void memset_io(void *addr, u8 value, unsigned int count);
void memcpy_fromio(void *dest, void *source, unsigned int count);
void memcpy_toio(void *dest, void *source, unsigned int count);

Linux Internals

Driver Development

Character Drivers

Usefulness of Character Drivers

Except for storage device drivers, most drivers for devices with input and output flows are implemented as character drivers.

So, most drivers you will face will be character drivers
You will regret if you sleep during this part!

Character device files

Accessed through a sequential flow of individual characters

Character devices can be identified by their c type (ls -l):

```
crw-rw---- 1 root uucp   4,  64 Feb 23 2004 /dev/ttyS0
crw--w---- 1 jdoe tty  136,   1 Feb 23 2004 /dev/pts/1
crw------- 1 root root  13,  32 Feb 23 2004 /dev/input/mouse0
crw-rw-rw- 1 root root   1,   3 Feb 23 2004 /dev/null
```

Example devices: keyboards, mice, parallel port, IrDA, Bluetooth port, consoles, terminals, sound, video...

Device major and minor numbers

- As you could see in the previous examples,
  device files have 2 numbers associated to them:

First number: *major* number

Second number: *minor* number

Major and minor numbers are used by the kernel to bind a driver to the device file. Device file names don't matter to the kernel!

To find out which driver a device file corresponds to, or when the device name is too cryptic, see Documentation/devices.txt.

MABEL Technology
Tech The Best

Information on Registered Devices

- Registered devices are visible in /proc/devices:

Character devices:	Block devices:
1 mem	1 ramdisk
4 /dev/vc/0	3 ide0
4 tty	8 sd
4 ttyS	9 md
5 /dev/tty	22 ide1
5 /dev/console	65 sd
5 /dev/ptmx	66 sd
6 lp	67 sd
7 vcs	68 sd
10 misc	69 sd
13 input	
14 sound	
...	

Major number

Registered name

Can be used to find free major numbers

Device file creation

Device files are not created when a driver is loaded.

They have to be created in advance:
    mknod /dev/<device> [c|b] <major> <minor>

Examples:
    mknod /dev/ttyS0 c 4 64
    mknod /dev/hda1 b 3 1

Creating a Character Driver

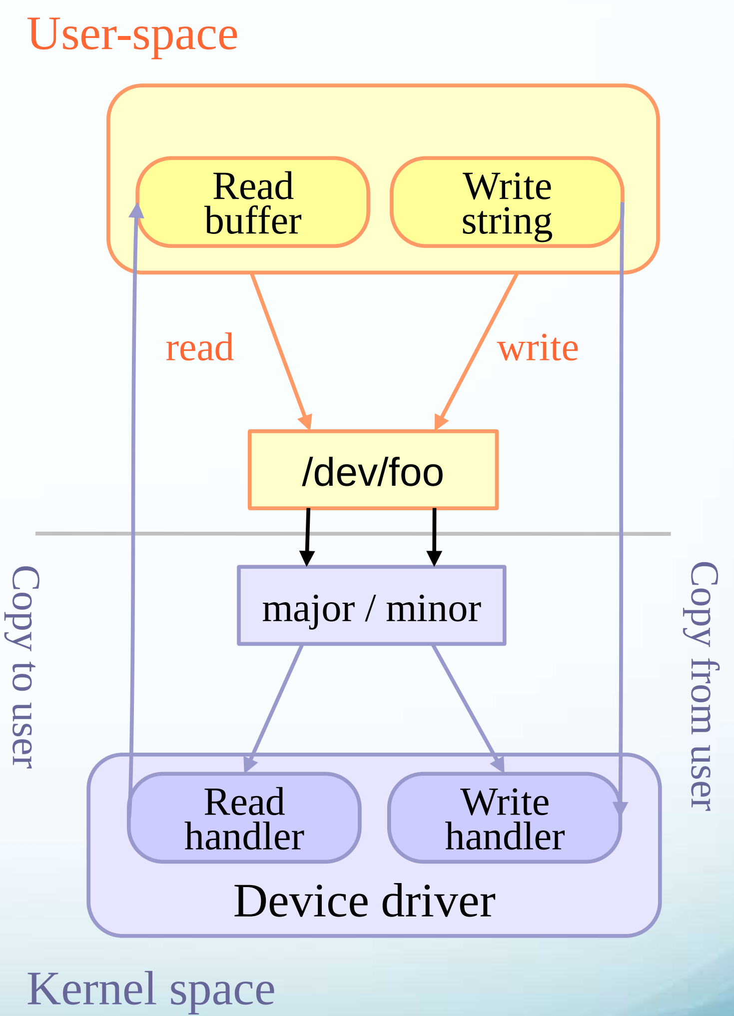
- User-space needs

The name of a device file in /dev to interact with the device driver through regular file operations (open, read, write, close...)

The kernel needs

To know which driver is in charge of device files with a given major / minor number pair

For a given driver, to have handlers ("*file operations*") to execute when user-space opens, reads, writes or closes the device file.



Declaring a Character Driver

- Device number registration

Need to register one or more device numbers (major/minor pairs), depending on the number of devices managed by the driver.

Need to find free ones!

File operations registration

Need to register handler functions called when user space programs access the device files: open, read, write, ioctl, close...

dev_t Structure

- Kernel data structure to represent a major/minor pair.

Defined in <linux/kdev_t.h>
  Linux 2.6: 32 bit size (major: 12 bits, minor: 20 bits)

Macro to create the structure:
  MKDEV(int major, int minor);

Macros to extract the numbers:
  MAJOR(dev_t dev);
  MINOR(dev_t dev);

Allocating Fixed Device Numbers

- #include <linux/fs.h>

- ```
  int register_chrdev_region(
      dev_t from,              /* Starting device number */
      unsigned count,          /* Number of device numbers */
      const char *name);       /* Registered name */
  ```

- Returns 0 if the allocation was successful.

- <u>Example</u>

- ```
  if (register_chrdev_region(MKDEV(202, 128),
                                          acme_count, "acme")) {
  printk(KERN_ERR "Failed to allocate device number\n");
  ...
  ```

MABEL Technology
Tech The Best

Dynamic Allocation of Device Numbers

- Safer: have the kernel allocate free numbers for you!
- #include <linux/fs.h>
-
```
int alloc_chrdev_region(
    dev_t *dev,                /* Output: starting device number */
    unsigned baseminor,        /* Starting minor number, usually 0 */
    unsigned count,            /* Number of device numbers */
    const char *name);         /* Registered name */
```
- Returns 0 if the allocation was successful.
- <u>Example</u>
-
```
if (alloc_chrdev_region(&acme_dev, 0, acme_count, "acme")) {
printk(KERN_ERR "Failed to allocate device number\n");
...
```

File Operations (1)

- Before registering character devices, you have to define file_operations (called *fops*) for the device files.
  Here are the main ones:

```
int (*open)(
    struct inode *, /* Corresponds to the device file */
    struct file *); /* Corresponds to the open file descriptor */
```

Called when user-space opens the device file.

```
int (*release)(
    struct inode *,
    struct file *);
```

Called when user-space closes the file.

The file Structure

- Is created by the kernel during the open call. Represents open files. Pointers to this structure are usually called "*fips*".

mode_t f_mode;
The file opening mode (FMODE_READ and/or FMODE_WRITE)

loff_t f_pos;
Current offset in the file.

struct file_operations *f_op;
Allows to change file operations for different open files!

struct dentry *f_dentry
Useful to get access to the inode: filp->f_dentry->d_inode.

To find the minor number use:
MINOR(flip->f_dentry->d_inode->i_rdev)

File Operations (2)

```
ssize_t (*read)(
      struct file *,          /* Open file descriptor */
      char *,                 /* User-space buffer to fill up */
      size_t,                 /* Size of the user-space buffer */
      loff_t *);              /* Offset in the open file */
```

Called when user-space reads from the device file.

```
ssize_t (*write)(
      struct file *,       /* Open file descriptor */
      const char *,        /* User-space buffer to write to the device */
      size_t,              /* Size of the user-space buffer */
      loff_t *);           /* Offset in the open file */
```

Called when user-space writes to the device file.

Exchanging Data With User-Space (1)

- In driver code, you can't just memcpy between an address supplied by user-space and the address of a buffer in kernel-space!

Correspond to completely different address spaces (thanks to virtual memory).

The user-space address may be swapped out to disk.

The user-space address may be invalid (user space process trying to access unauthorized data).

Exchanging Data With User-Space (2)

- You must use dedicated functions such as the following ones in your read and write file operations code:
- include <asm/uaccess.h>
- unsigned long copy_to_user(void __user *to,
  const void *from,
  unsigned long n);

  unsigned long copy_from_user(void *to,
  const void __user *from,
  unsigned long n);
- Make sure that these functions return 0!
  Another return value would mean that they failed.

File Operations (3)

int (*ioctl) (struct inode *, struct file *,
                 unsigned int, unsigned long);

Can be used to send specific commands to the device, which are neither reading nor writing (e.g. formatting a disk, configuration changes).

int (*mmap) (struct file *,
                 struct vm_area_struct);

Asking for device memory to be mapped into the address space of a user process

struct module *owner;

Used by the kernel to keep track of who's using this structure and count the number of users of the module. Set to THIS_MODULE.

Read Operation Example

```
static ssize_t
acme_read(struct file *file, char __user *buf, size_t count, loff_t * ppos)
{
   /* The hwdata address corresponds to a device I/O memory area */
   /* of size hwdata_size, obtained with ioremap() */
   int remaining_bytes;

   /* Number of bytes left to read in the open file */
   remaining_bytes = min(hwdata_size - (*ppos), count);

   if (remaining_bytes == 0) {
           /* All read, returning 0 (End Of File) */
           return 0;
   }

   if (copy_to_user(buf /* to */, *ppos+hwdata /* from */, remaining_bytes)) {
      return -EFAULT;

   /* Increase the position in the open file */
   *ppos += remaining_bytes;
   return remaining_bytes;
}
```

Read method

Piece of code available on
http://free-electrons.com/doc/c/acme_read.c

Write Operation Example

```
static ssize_t
acme_write(struct file *file, const char __user *buf, size_t count, loff_t * ppos)
{
   /* Assuming that hwdata corresponds to a physical address range */
   /* of size hwdata_size, obtained with ioremap() */

   /* Number of bytes not written yet in the device */
   remaining_bytes = hwdata_size - (*ppos);

   if (count > remaining_bytes) {
             /* Can't write beyond the end of the device */
             return -EIO;
   }

   if (copy_from_user(*ppos+hwdata /* to */, buf /* from */, count)) {
      return -EFAULT;

   /* Increase the position in the open file */
   *ppos += count;
   return count;
}
```

Write method

Piece of code available on
http://free-electrons.com/doc/c/acme_write.c

MABEL Technology
Tech The Best

File Operations Definition Example

- <u>Defining a file_operations structure</u>
- include <linux/fs.h>
- static struct file_operations acme_fops =

```
{
    .owner = THIS_MODULE,
    .read = acme_read,
    .write = acme_write,
};
```

- You just need to supply the functions you implemented! Defaults for other functions (such as open, release...)
  are fine if you do not implement anything special.

Character Device Registration (1)

The kernel represents character drivers using the cdev structure.

Declare this structure globally (within your module):

```
#include <linux/cdev.h>
static struct cdev *acme_cdev;
```

In the init function, allocate the structure and set its file operations:

```
acme_cdev = cdev_alloc();
acme_cdev->ops = &acme_fops;
acme_cdev->owner = THIS_MODULE;
```

Character Device Registration (2)

Now that your structure is ready, add it to the system:

```
int cdev_add(
    struct cdev *p,        /* Character device structure */
    dev_t dev,             /* Starting device major / minor number */
    unsigned count);       /* Number of devices */
```

Example (continued):

```
if (cdev_add(acme_cdev, acme_dev, acme_count)) {
printk (KERN_ERR “Char driver registration failed\n”);
...
```

Character Device Unregistration

First delete your character device:

```
void cdev_del(struct cdev *p);
```

Then, and only then, free the device number:

```
void unregister_chrdev_region(dev_t from, unsigned count);
```

Example (continued):

```
cdev_del(acme_cdev);
unregister_chrdev_region(acme_dev, acme_count);
```

Char Driver Example Summary (1)

```
static void *hwdata;
static hwdata_size=8192;

static int acme_count=1;
static dev_t acme_dev;

static struct cdev *acme_cdev;

static ssize_t acme_write(...) {...}

static ssize_t acme_read(...) {...}

static struct file_operations acme_fops =
{
        .owner = THIS_MODULE,
        .read = acme_read,
        .write = acme_write,
};
```

MABEL Technology

Tech The Best

```
static int __init acme_init(void)
{
    int err;
    hwdata = ioremap(PHYS_ADDRESS,
        hwdata_size);

    if (!acme_buf) {
       err = -ENOMEM;
       goto err_exit;
    }

    if (alloc_chrdev_region(&acme_dev, 0,
                 acme_count, "acme")) {
       err=-ENODEV;
       goto err_free_buf;
    }

    acme_cdev = cdev_alloc();

    if (!acme_cdev) {
       err=-ENOMEM;
       goto err_dev_unregister;
    }

    acme_cdev->ops = &acme_fops;
    acme_cdev->owner = THIS_MODULE;
```

```
if (cdev_add(acme_cdev, acme_dev,
                 acme_count)) {
       err=-ENODEV;
       goto err_free_cdev;
    }

    return 0;

err_free_cdev:
    kfree(acme_cdev);
err_dev_unregister:
    unregister_chrdev_region(
       acme_dev, acme_count);
err_free_buf:
    iounmap(hwdata);
err_exit:
    return err;
}

static void __exit acme_exit(void)
{
    cdev_del(acme_cdev);
    unregister_chrdev_region(acme_dev,
                        acme_count);
    iounmap(hwdata);
}
```

Show how to handle errors and deallocate resources in the right order!

Character Driver Summary

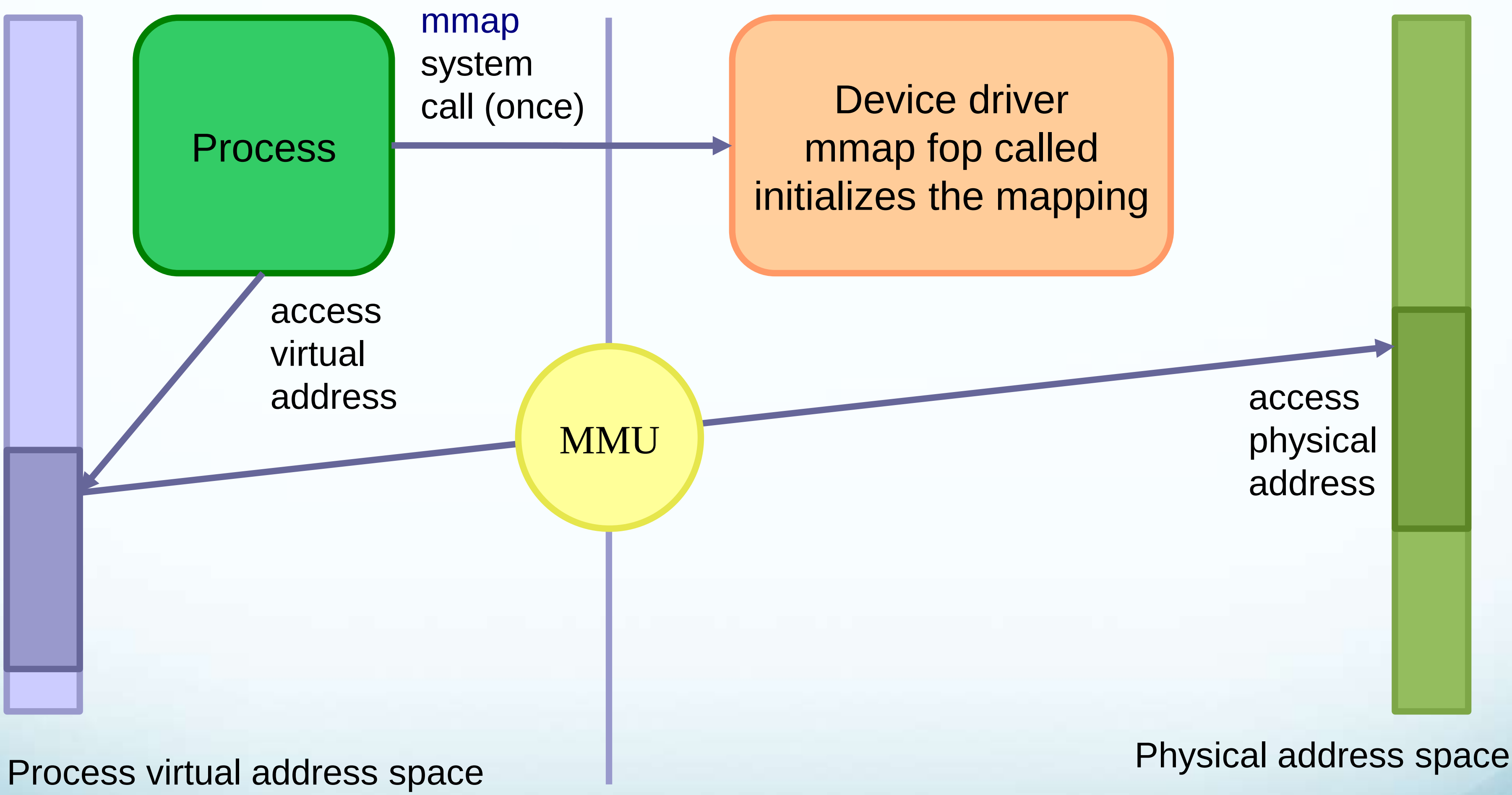
Character driver writer - Define the file operations callbacks for the device file: `read`, `write`, `ioctl`... - In the module init function, get major and minor numbers with `alloc_chrdev_region()`, init a `cdev` structure with your file operations and add it to the system with `cdev_add()`. - In the module exit function, call `cdev_del()` and `unregister_chrdev_region()`	Kernel
System administration - Load the character driver module - In `/proc/devices`, find the major number it uses. - Create the device file with this major number The device file is ready to use!	User-space
System user - Open the device file, read, write, or send ioctl's to it.	User-space
Kernel - Executes the corresponding file operations	Kernel

MABEL Technology

Tech The Best

Moving data between user and kernel

mmap overview



How to Implement mmap() - Kernel-Space

Character driver: implement a mmap file operation and add it to the driver file operations:

```
int (*mmap)(
    struct file *,              /* Open file structure */
    struct vm_area_struct       /* Kernel VMA structure */
);
```

Initialize the mapping.

Can be done in most cases with the remap_pfn_range() function, which takes care of most of the job.

remap_pfn_range()

pfn: page frame number.
The most significant bits of the page address
(without the bits corresponding to the page size).

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

int remap_pfn_range(
   struct vm_area_struct *,              /* VMA struct */
   unsigned long virt_addr,      /* Starting user virtual address */
   unsigned long pfn,       /* pfn of the starting physical address */
   unsigned long size,           /* Mapping size */
   pgprot_t                      /* Page permissions */
);
```

PFN: Page Frame Number, the number of the page (0, 1, 2, ...).

Simple mmap() Implementation

```
static int acme_mmap(
    struct file *file, struct vm_area_struct *vma)
{
    size = vma->vm_end - vma->vm_start;

    if (size > ACME_SIZE)
    return -EINVAL;

    if (remap_pfn_range(vma,
                        vma->vm_start,
                        ACME_PHYS >> PAGE_SHIFT,
                        size,
                        vma->vm_page_prot))
        return -EAGAIN;
    return 0;
}
```

mmap summary

The device driver is loaded.
    It defines an mmap file operation.

A user space process calls the mmap system call.

The mmap file operation is called.
    It initializes the mapping using the device physical address.

The process gets a starting address to read from and write to
    (depending on permissions).

The MMU automatically takes care of converting the process
    virtual addresses into physical ones.

Direct access to the hardware!
    No expensive read or write system calls!

Sending Signal to user

```
struct siginfo info;
info.si_signo = SIG_TEST; // define as SIGRTMIN+5
info.si_code = SI_QUEUE; // emulate sigqueue
info.si_int = 1234; // signal data
struct task_struct *t;
t = find_task_by_pid_type(PIDTYPE_PID, pid);
send_sig_info(SIG_TEST, &info, t);
```

Using Virtual file systems

- Proc

```
 myproc = create_proc_entry(MYNAME,0,NULL);
myproc->read_proc = myread;
myproc->write_proc = mywrite;
```

- Sysfs

```
kobj_set_kset_s(&myfs_subsys, fs_subsys);
myfs_subsys.kobj.ktype = &mytype;
err = subsystem_register(&myfs_subsys);
```

- DebugFS

```
dir = debugfs_create_dir("mydirectory", NULL);
file = debugfs_create_file("myfile", 0644, dir,
                           &file_value, &my_fops);
```

Socket Based

- AF_INET (with UDP)

```
sock_create(PF_INET, SOCK_DGRAM, IPPROTO_UDP, &clientsocket)
```

- `AF_PACKET`
- `AF_NETLINK`

  `genlmsg_new`: allocates a new skb that will be sent to the user space.

  `genlmsg_put`: fills the generic netlink header.

  `nla_put_string`: write a string into the message.

  `genlmsg_end`: finalize the message

  `genlmsg_unicast`: send the message back to the user space program.

Linux Internals

Driver Development

Debugging

MABEL Technology
Tech The Best

Debugging Tools

- strace(1)
- ltrace(1)
- POSIX Threads Trace Toolkit
- Dmalloc
- Valgrind

Valgrind Tools

- Memcheck – detects memory management problems
- Cachegrind – cache profiler for I1, D1 and L2 CPU caches
- Helgrind – multi-threading data race detector
- Callgrind – heavyweight profiler
- Massif – heap profiler

Splint

- http://splint.org/, from the University of Virginia
- GPL tool for statically checking C programs for security vulnerabilities and coding mistakes
- Today's lint program for GNU/Linux.
  The successor of LClint.
- Very complete manual and documentation
- Doesn't support C++

Oprofile

- http://oprofile.sourceforge.net
- A system-wide profiling tool
- Can collect statistics like the top users of the CPU.
- Works without having the sources.
- Requires a kernel patch to access all features, but is already available in a standard kernel.
- Requires more investigation to see how it works.
- Ubuntu/Debian packages:
  oprofile, oprofile-gui

Kernel Debugging with printk

- Universal debugging technique used since the beginning of programming (first found in cavemen drawings)
- Printed or not in the console or /var/log/messages according to the priority. This is controlled by the loglevel kernel parameter, or through /proc/sys/kernel/printk (see Documentation/sysctl/kernel.txt)
- Available priorities (include/linux/kernel.h):

```
#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 */
```

Debugging with the Magic Key

- You can have a “magic” key to control the kernel.
- To activate this feature, make sure that:
  - Kernel configuration CONFIG_MAGIC_SYSRQ enabled.
  - Enable it at run time:

echo “1” > **/proc/sys/kernel/sysrq**

- The key is:
  - PC Console: **SysRq**
  - Serial Console: Send a **BREAK**
  - From shell (2.6 only): echo t > **/proc/sysrq-trigger**

Debugging with the Magic Key Cont.

- Together with the magic key, you use the following:
  - **b**: hard boot (no sync, no unmount)
  - **s**: sync
  - **u**: Remount all read-only.
  - **t**: task list (proccess table).
  - **1-8**: Set console log level.
  - **e**: Show Instruction Pointer.
  - And more... press **h** for help.
- Programmers can add their own handlers as well.
- See Documentation/sysrq.txt for more details.

Debugging with /proc or /sys (1)

- Instead of dumping messages in the kernel log, you can have your drivers make information available to user space
- Through a file in /proc or /sys, which contents are handled by callbacks defined and registered by your driver.
- Can be used to show any piece of information about your device or driver.
- Can also be used to send data to the driver or to control it.
- Caution: anybody can use these files.
  You should remove your debugging interface in production!

Debugging with /proc or /sys (2)

- Examples
  - cat /proc/acme/stats (dummy example)
    Displays statistics about your acme driver.
  - cat /proc/acme/globals (dummy example)
    Displays values of global variables used by your driver.
  - echo 600000 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed
    Adjusts the speed of the CPU (controlled by the cpufreq driver).

Debugfs

- A virtual filesystem to export debugging information to user-space.
- Kernel configuration: DEBUG_FS
  Kernel hacking -> Debug Filesystem
- Much simpler to code than an interface in /proc or /sys. The debugging interface disappears when Debugfs is configured out.
- You can mount it as follows:
  sudo mount -t debugfs none /mnt/debugfs
- First described on http://lwn.net/Articles/115405/
- API documented in the Linux Kernel Filesystem API: http://free-electrons.com/kerneldoc/latest/DocBook/filesystems/index.html

Debugging with ioctl

- Can use the ioctl() system call to query information about your driver (or device) or send commands to it.
- This calls the ioctl file operation that you can register in your driver.
- Advantage: your debugging interface is not public. You could even leave it when your system (or its driver) is in the hands of its users.

KGDB

- (from version 2.6.26)

- The execution of the kernel is fully controlled by gdb from another machine, connected through a serial line.

- Can do almost everything, including inserting breakpoints in interrupt handlers.

Using KGDB

- Compile the kernel with:
  - Debugging information
  - Serial IO drivers
  - KGDB support
- Add to kernel command line:
  - kgdbwait
  - Kgdboc / kgdboe
- Run cross debugger with vmlinux file
  - (gdb) target remote /dev/ttyS1
    (gdb) b sys_read
    (gdb) c
- To break in the debugger
  - echo “g”>/proc/sysrq-trigger

Debugging Kerenl Module

- insmod ./your_module.ko
- Find out sections
  - cat /sys/module/[name]/sections/.text
  - cat /sys/module/[name]/sections/.data
  - cat /sys/module/[name]/sections/.bss
- Break in the debugger and load symbols
  - (gdb) add-symbol-file /path/to/module.ko [text] \
    - -s .bss [bss address] -s .data [data address]

MABEL Technology
Tech The Best

Debugging With Virtualization

- Debug the kernel like a user space application
- User Mode Linux
  - Compile the kernel with um architecture
  - Run and debug like a user space process
- QEMU (http://wiki.qemu.org/Main_Page)
  - Many platforms
  - Run with -s -S
  - Debug vmlinux
  - (gdb) target remote 127.0.0.1:1234

ftrace - Kernel function tracer

- New infrastructure that can be used for debugging or analyzing latencies and performance issues in the kernel.
- Developed by Steven Rostedt. Merged in 2.6.27.
  For earlier kernels, can be found from the rt-preempt patches.
- Very well documented in Documentation/ftrace.txt
- Negligible overhead when tracing is not enabled at run-time.
- Can be used to trace any kernel function!
- See our video of Steven's tutorial at OLS 2008:
  http://free-electrons.com/community/videos/conferences/

Using ftrace

- Tracing information available through the debugfs virtual fs (CONFIG_DEBUG_FS in the Kernel Hacking section)
- Mount this filesystem as follows:
  mount -t debugfs nodev /debug
- When tracing is enabled (see the next slides), tracing information is available in /debug/tracing.
- Check available tracers in /debug/tracing/available_tracers

Scheduling latency tracer

- CONFIG_SCHED_TRACER (Kernel Hacking section)

- Maximum recorded time between waking up a top priority task and its scheduling on a CPU, expressed in µs.

- Check that wakeup is listed in /debug/tracing/available_tracers

- To select, reset and enable this tracer:
  echo wakeup > /debug/tracing/current_tracer
  echo 0 > /debug/tracing/tracing_max_latency
  echo 1 > /debug/tracing/tracing_enabled

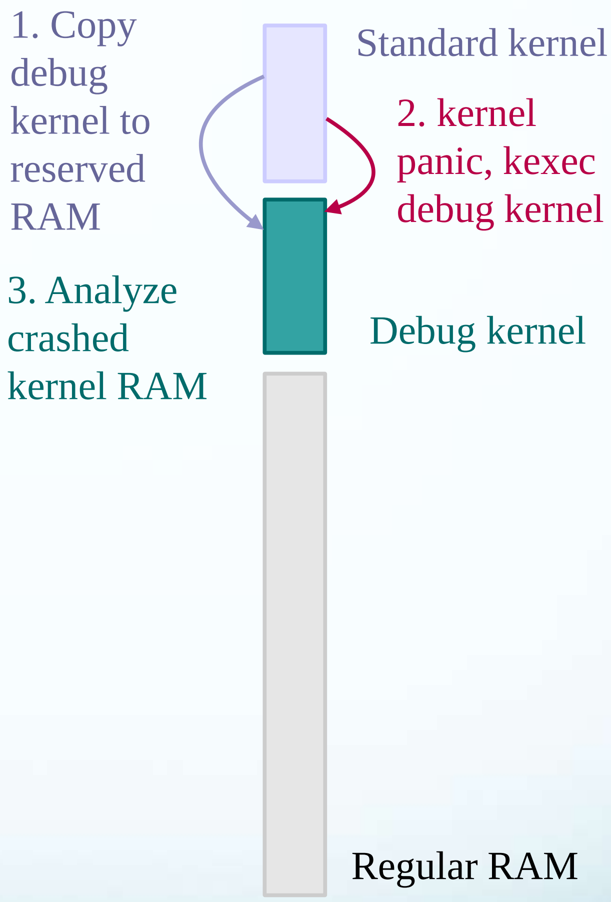
- Let your system run, in particular real-time tasks.
  Example: chrt -f 5 sleep 1

- Disable tracing:
  echo 0 > /debug/tracing/tracing_enabled

- Read the maximum recorded latency and the corresponding trace:
  cat /debug/tracing/tracing_max_latency

kexec/kdump

- kexec system call: makes it possible to call a new kernel, without rebooting and going through the BIOS / firmware.
- Idea: after a kernel panic, make the kernel automatically execute a new, clean kernel from a reserved location in RAM, to perform post-mortem analysis of the memory of the crashed kernel.
- See Documentation/kdump/kdump.txt in the kernel sources for details.



Debugging with SystemTap

- http://sourceware.org/systemtap/
- Infrastructure to add instrumentation to a running kernel: trace functions, read and write variables, follow pointers, gather statistics...
- Eliminates the need to modify the kernel sources to add one's own instrumentation to investigated a functional or performance problem.
- Uses a simple scripting language. Several example scripts and probe points are available.
- Based on the Kprobes instrumentation infrastructure. See Documentation/kprobes.txt in kernel sources. Linux 2.6.26: supported on most popular CPUs (arm included in 2.6.25). However, lack of recent support for mips (2.6.16 only!).

SystemTap script example (1)

```
#! /usr/bin/env stap
# Using statistics and maps to examine kernel memory allocations

global kmalloc

probe kernel.function("__kmalloc") {
        kmalloc[execname()] <<< $size
}

# Exit after 10 seconds
probe timer.ms(10000) { exit () }

probe end {
        foreach ([name] in kmalloc) {
                printf("Allocations for %s\n", name)
                printf("Count:   %d allocations\n", @count(kmalloc[name]))
                printf("Sum:     %d Kbytes\n", @sum(kmalloc[name])/1000)
                printf("Average: %d bytes\n", @avg(kmalloc[name]))
                printf("Min:     %d bytes\n", @min(kmalloc[name]))
                printf("Max:     %d bytes\n", @max(kmalloc[name]))
                print("\nAllocations by size in bytes\n")
                print(@hist_log(kmalloc[name]))
                printf("-------------------------------------------------------\n\n");
        }
}
```

SystemTap script example (2)

```
#! /usr/bin/env stap

# Logs each file read performed by each process

probe kernel.function ("vfs_read")
{
  dev_nr = $file->f_dentry->d_inode->i_sb->s_dev
  inode_nr = $file->f_dentry->d_inode->i_ino
  printf ("%s(%d) %s 0x%x/%d\n",
      execname(), pid(), probefunc(), dev_nr, inode_nr)
}
```

- Nice tutorial on http://sources.redhat.com/systemtap/tutorial.pdf

Kernel markers

- Capability to add static markers to kernel code, merged in Linux 2.6.24 by Matthieu Desnoyers.
- Almost no impact on performance, until the marker is dynamically enabled, by inserting a probe kernel module.
- Useful to insert trace points that won't be impacted by changes in the Linux kernel sources.
- See marker and probe example in samples/markers in the kernel sources.

See http://en.wikipedia.org/wiki/Kernel_marker

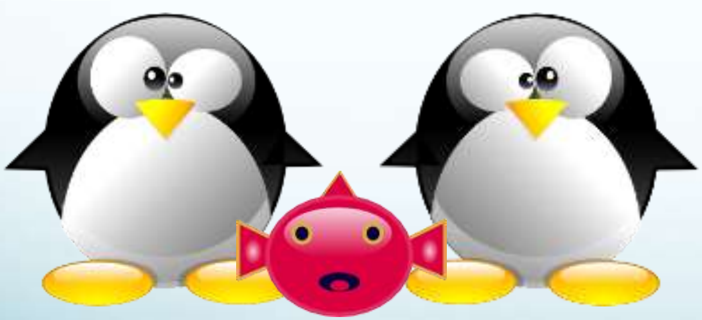
LTTng

- http://ltt.polymtl.ca/
- The successor of the Linux Trace Toolkit (LTT)
- Toolkit allowing to collect and analyze tracing information from the kernel, based on kernel markers and kernel tracepoints.
- So far, based on kernel patches, but doing its best to use in-tree solutions, and to be merged in the future.
- Very precise timestamps, very little overhead.
- Useful guidelines in http://ltt.polymtl.ca/svn/trunk/lttv/QUICKSTART

MABEL Technology
Tech The Best

Driver Development

Locking



Sources of concurrency issues

- The same resources can be accessed by several kernel processes in parallel, causing potential concurrency issues

Several user-space programs accessing the same device data or hardware. Several kernel processes could execute the same code on behalf of user processes running in parallel.

Multiprocessing: the same driver code can be running on another processor. This can also happen with single CPUs with hyperthreading.

Kernel preemption, interrupts: kernel code can be interrupted at any time (just a few exceptions), and the same data may be access by another process before the execution continues.

Avoiding concurrency issues

Avoid using global variables and shared data whenever possible
(cannot be done with hardware resources)

Don't make resources available to other kernel processes until they are ready to be used.

Use techniques to manage concurrent access to resources.

See Rusty Russell's Unreliable Guide To Locking Documentation/DocBook/kernel-locking/
in the kernel sources.

Linux mutexes

The main locking primitive since Linux 2.6.16.

Better than counting semaphores when binary ones are enough.

Mutex definition:

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

Initializing a mutex statically:

```
DEFINE_MUTEX(name);
```

Or initializing a mutex dynamically:

```
void mutex_init(struct mutex *lock);
```

locking and unlocking mutexes

void mutex_lock (struct mutex *lock);
Tries to lock the mutex, sleeps otherwise.
Caution: can't be interrupted, resulting in processes you cannot kill!

int mutex_lock_killable (struct mutex *lock);
Same, but can be interrupted by a fatal (SIGKILL) signal. If interrupted, returns a non zero value and doesn't hold the lock. Test the return value!!!

int mutex_lock_interruptible (struct mutex *lock);
Same, but can be interrupted by any signal.

int mutex_trylock (struct mutex *lock);
Never waits. Returns a non zero value if the mutex is not available.

int mutex_is_locked(struct mutex *lock);
Just tells whether the mutex is locked or not.

void mutex_unlock (struct mutex *lock);
Releases the lock. Do it as soon as you leave the critical section.

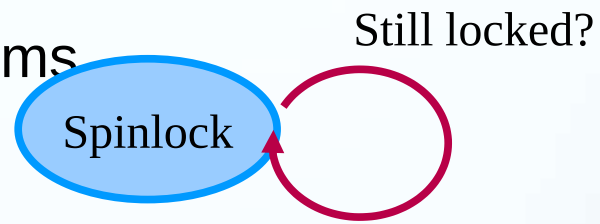
Reader / writer semaphores

- Allow shared access by unlimited readers, or by only 1 writer. Writers get priority.
- void init_rwsem (struct rw_semaphore *sem);
- void down_read (struct rw_semaphore *sem);
  int down_read_trylock (struct rw_semaphore *sem);
  int up_read (struct rw_semaphore *sem);
- void down_write (struct rw_semaphore *sem);
  int down_write_trylock (struct rw_semaphore *sem);
  int up_write (struct rw_semaphore *sem);
- Well suited for rare writes, holding the semaphore briefly. Otherwise, readers get *starved*, waiting too long for the semaphore to be released.

Spinlocks

Locks to be used for code that is not allowed to sleep (interrupt handlers), or that doesn't want to sleep (critical sections). Be very careful not to call functions which can sleep!

Originally intended for multiprocessor systems



Spinlocks never sleep and keep spinning in a loop until the lock is available.

Spinlocks cause kernel preemption to be disabled on the CPU executing them.

Initializing spinlocks

Static

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
```

Dynamic

```
void spin_lock_init (spinlock_t *lock);
```

Using spinlocks (1)

- Several variants, depending on where the spinlock is called:

void spin_[un]lock (spinlock_t *lock);

Doesn't disable interrupts. Used for locking in process context (critical sections in which you do not want to sleep).

void spin_lock_irqsave / spin_unlock_irqrestore (spinlock_t *lock, unsigned long flags);

Disables / restores IRQs on the local CPU.

Typically used when the lock can be accessed in both process and interrupt context, to prevent preemption by interrupts.

Using spinlocks (2)

void spin_[un]lock_bh (spinlock_t *lock);

Disables software interrupts, but not hardware ones. Useful to protect shared data accessed in process context and in a soft interrupt (“bottom half”). No need to disable hardware interrupts in this case.

Note that reader / writer spinlocks also exist.

Avoiding Coherency Issues

Hardware independent

```
#include <asm/kernel.h>
void barrier(void);
```

Only impacts the behavior of the
compiler. Doesn't prevent reordering
in the processor!

Hardware dependent

```
#include <asm/system.h>
void rmb(void);
void wmb(void);
void mb(void);
```

Safe on all architectures!

Alternatives to Locking

- As we have just seen, locking can have a strong negative impact on system performance. In some situations, you could do without it.

By using lock-free algorithms like Read Copy Update (RCU).
RCU API available in the kernel
(See http://en.wikipedia.org/wiki/RCU).

When available, use atomic operations.

Atomic Variables

Useful when the shared resource is an integer value

Even an instruction like n++ is not guaranteed to be atomic on all processors!

Header

#include <asm/atomic.h>

Type

atomic_t
contains a signed integer (use 24 bits only)

Atomic operations (main ones)

Set or read the counter:

```
atomic_set(atomic_t *v, int i);
int atomic_read(atomic_t *v);
```

Operations without return value:

```
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_ *v);
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);
```

Similar functions testing the result:

```
int atomic_inc_and_test(...);
int atomic_dec_and_test(...);
int atomic_sub_and_test(...);
```

Functions returning the new value:

```
int atomic_inc_and_return(...);
int atomic_dec_and_return(...);
int atomic_add_and_return(...);
int atomic_sub_and_return(...);
```

Atomic Bit Operations

Supply very fast, atomic operations

On most platforms, apply to an unsigned long type.
Apply to a void type on a few others.

Set, clear, toggle a given bit:

```
void set_bit(int nr, unsigned long *addr);
void clear_bit(int nr, unsigned long *addr);
void change_bit(int nr, unsigned long *addr);
```

Test bit value:

```
int test_bit(int nr, unsigned long *addr);
```

Test and modify (return the previous value):

```
int test_and_set_bit(...);
int test_and_clear_bit(...);
int test_and_change_bit(...);
```

MABEL Technology

Tech The Best

How Time Flys

Timer Frequency

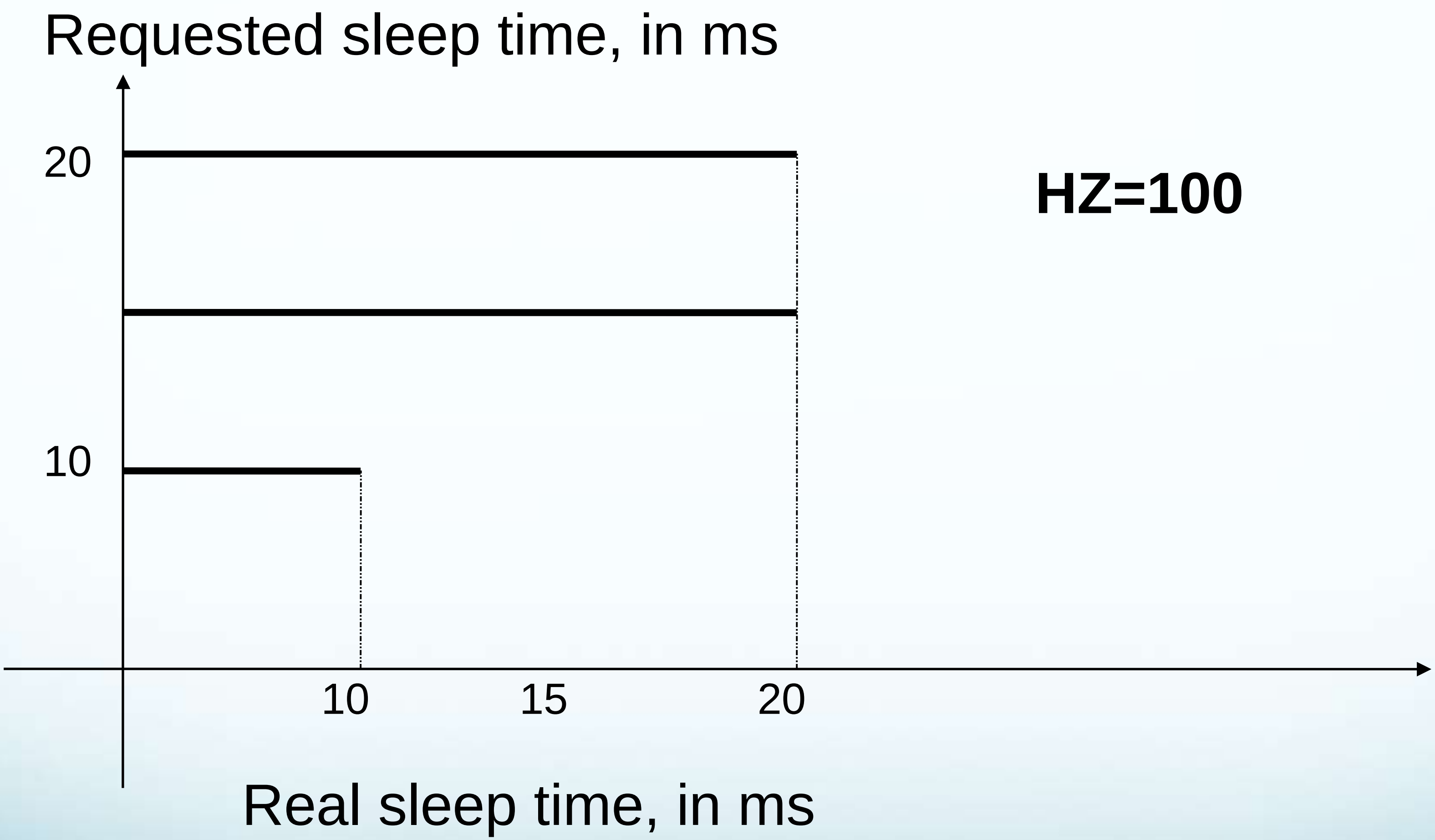
- Timer interrupts are raised every HZ th of second (= 1 *jiffy*)

HZ is now configurable (in Processor type and features):
100 (i386 default), 250 or 1000.
Supported on i386, ia64, ppc, ppc64, sparc64, x86_64
See kernel/Kconfig.hz.

Compromise between system responsiveness and global throughput.

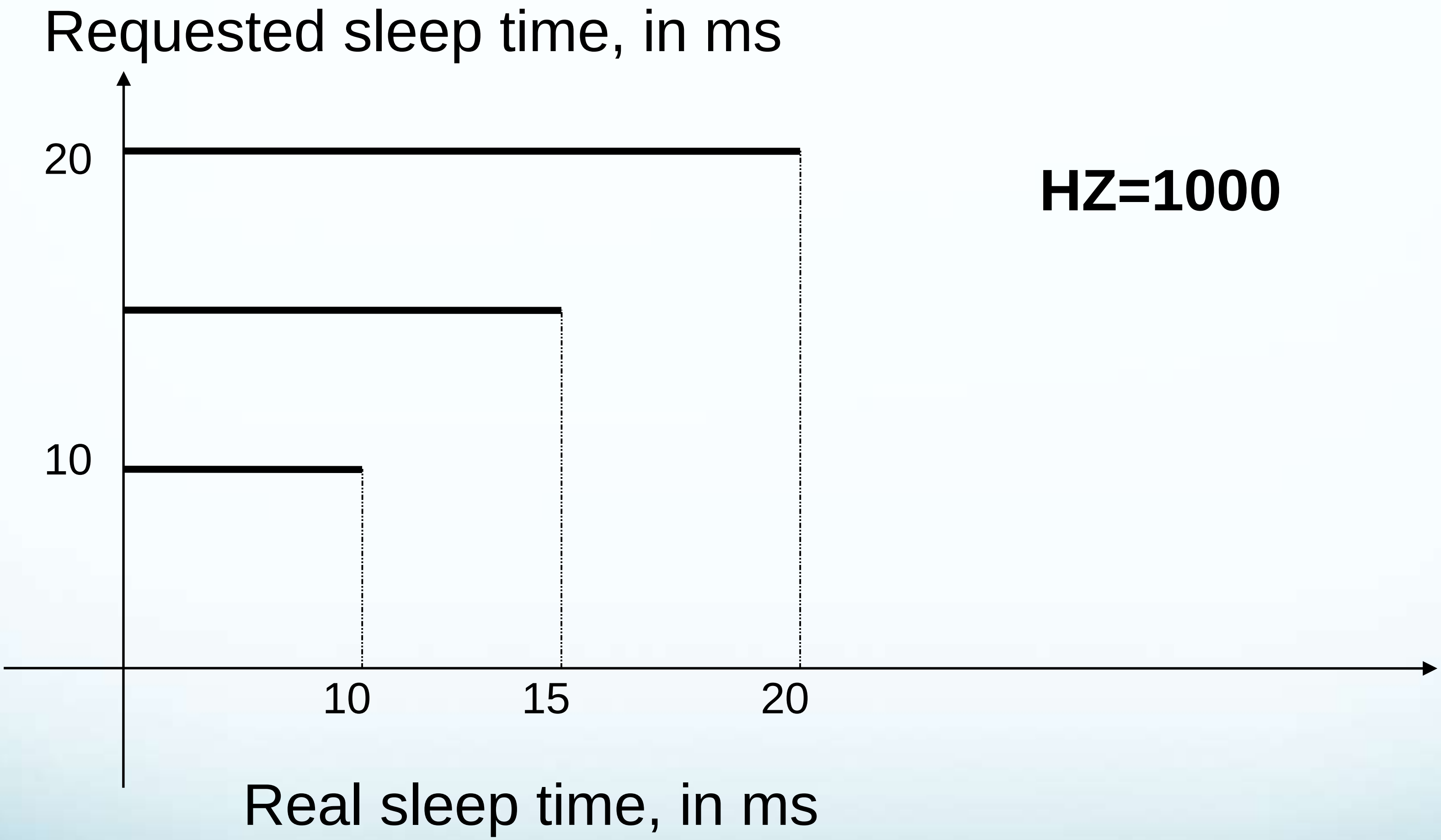
Caution: not any value can be used. Constraints apply!

The Effect of Timer Frequency



HZ=100

The Effect of Timer Frequency cont.



High-Res Timers and Tickless Kernel

The **high-res timers** feature enables POSIX timers and *nanosleep()* to be as accurate as the hardware allows (around 1usec on typical hardware) by using non RTC interrupt timer sources if supported by hardware.

- This feature is transparent - if enabled it just makes these timers much more accurate than the current HZ resolution.

The **tickless kernel** feature enables 'on-demand' timer interrupts.

- On x86 test boxes the measured effective IRQ rate drops to to 1-2 timer interrupts per second.

Timers

A timer is represented by a *timer_list* structure:

```
struct timer_list {
	/* ... */
	unsigned long expires;				/* In Jiffies */
	void (*function )(unsigned int);
	unsigned long data;					/* Optional */		);
```

Timer Operations

Manipulated with:

```
void init_timer(struct timer_list *timer);
void add_timer(struct timer_list *timer);
void init_timer_on(struct timer_list *timer, int cpu);
void del_timer(struct timer_list *timer);
void del_timer_sync(struct timer_list *timer);
void mod_timer(struct timer_list *timer, unsigned long expires);
void timer_pending(const struct timer_list *timer);
```

MABEL Technology
Tech The Best

Driver Development

Processes and Scheduling

Processes and Threads – a Reminder

A process is an instance of a running program.

- Multiple instances of the same program can be running. Program code ("text section") memory is shared.
- Each process has its own data section, address space, open files and signal handlers.

A thread is a single task in a program.

- It belongs to a process and shares the common data section, address space, open files and pending signals.
- It has its own stack, pending signals and state.

It's common to refer to single threaded programs as processes.

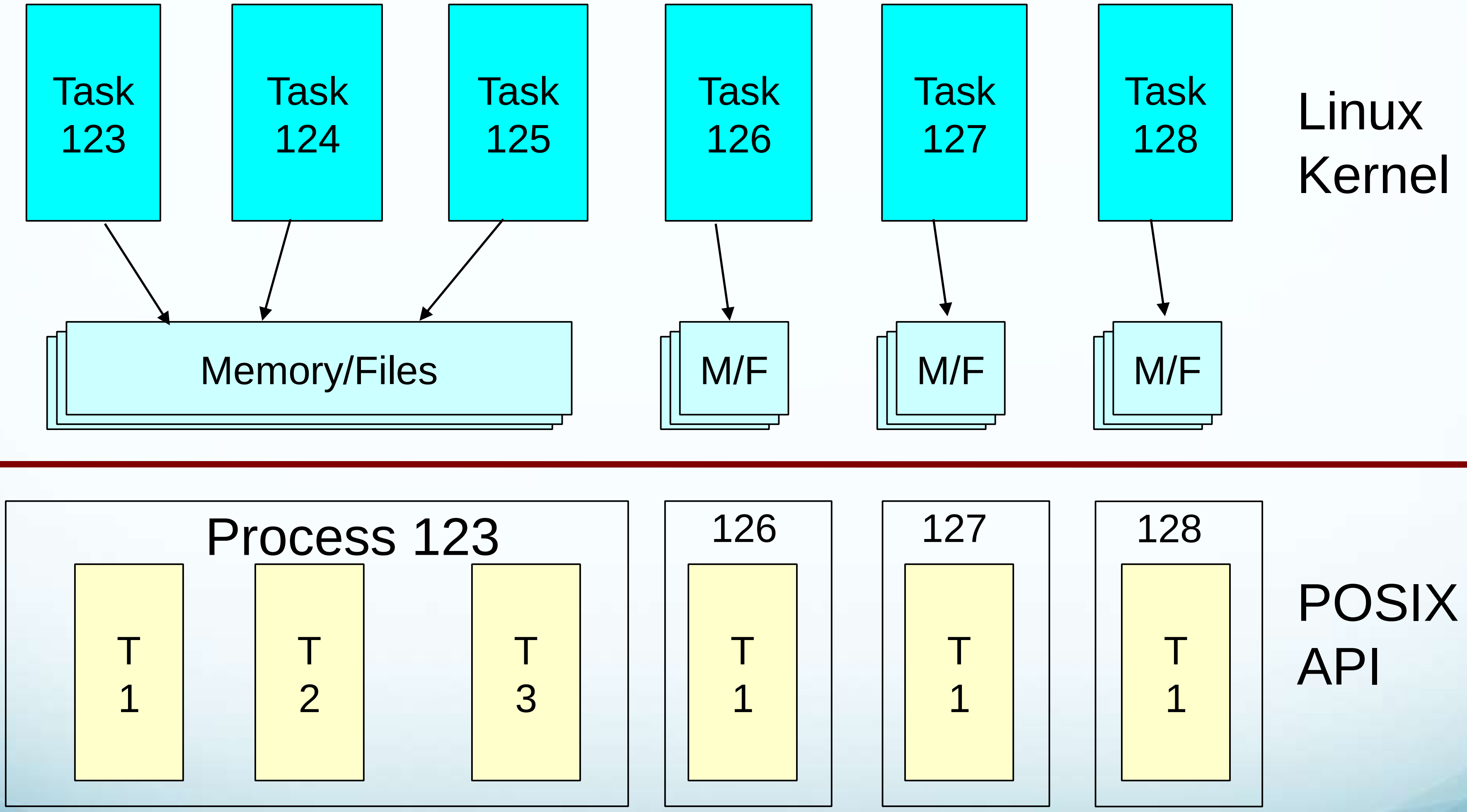
The Kernel and Threads

In 2.6 an explicit notion of processes and threads was introduced to the kernel.

Scheduling is done on a thread by thread basis.

The basic object the kernel works with is a task, which is analogous to a thread.

Thread vs. Process vs. Task



task_struct

Each task is represented by a task_struct.

The task is linked in the task tree via:

parent Pointer to its parent

children A linked list

sibling A linked list

task_struct contains many fields:

comm: name of task

priority, rt_priority: nice and real-time priorities

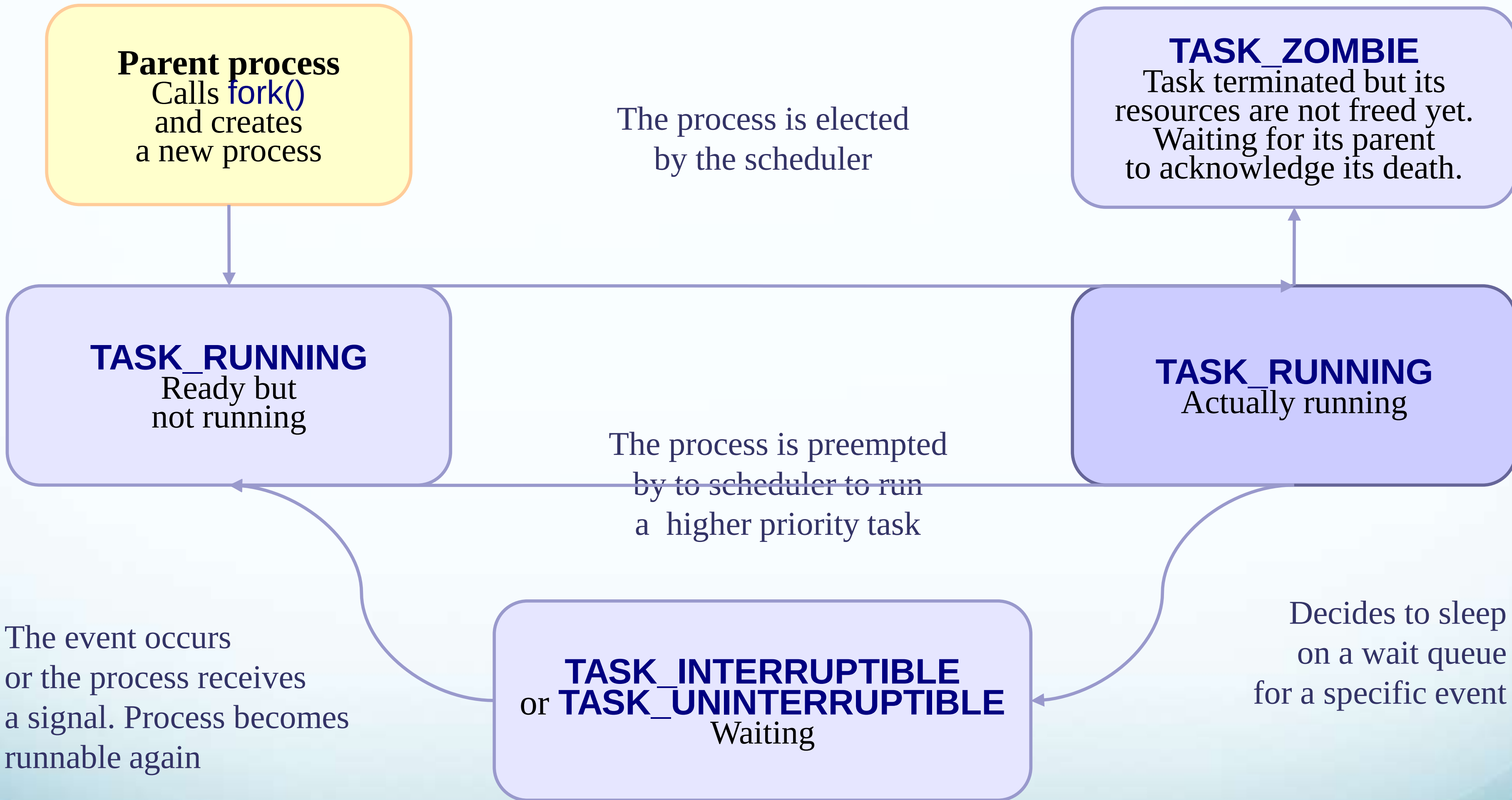
uid, euid, gid, egid: task's security credentials

Current Task

- current points to the current process task_struct
  - When applicable – not valid in interrupt context.
- Current is a macro that appears to the programmer as a magical global variable which updated each context switch.
  - Real value is either in register or computed from start of stack register value.
- On SMP machine current will point to different structure on each CPU.

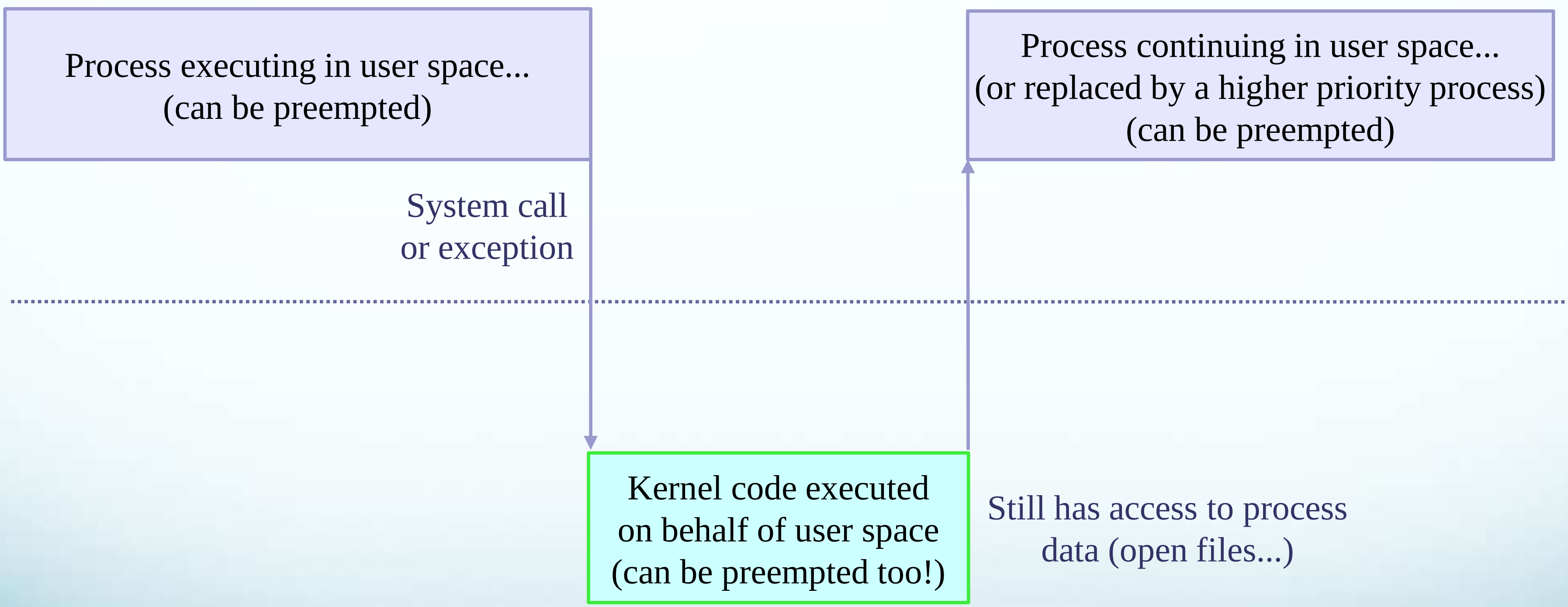
MABEL Technology
Tech The Best

A Process Life



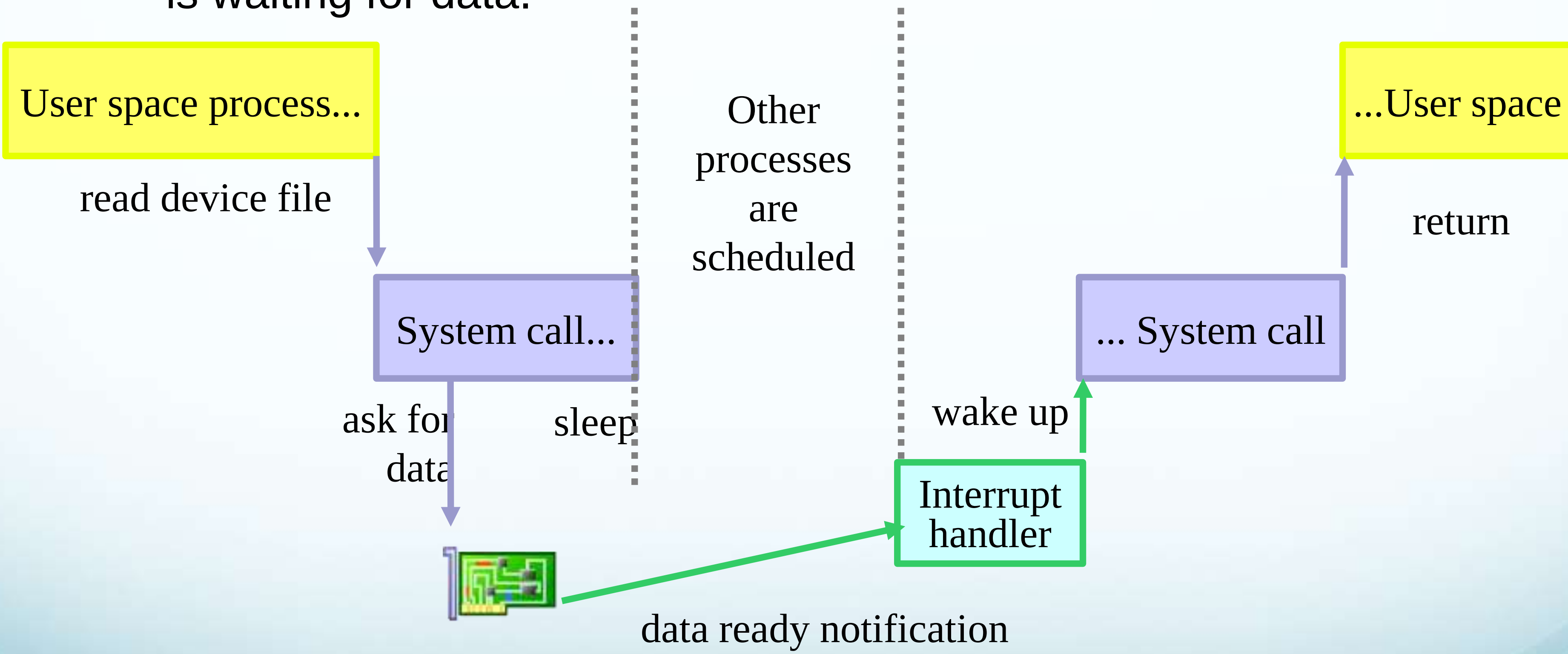
Process Context

User-space programs and system calls are scheduled together:



Sleeping

- Sleeping is needed when a process (user space or kernel space)
  is waiting for data.



How to sleep (1)

- Must declare a wait queue

Static queue declaration

```
DECLARE_WAIT_QUEUE_HEAD (module_queue);
```

Or dynamic queue declaration

```
wait_queue_head_t queue;
   init_waitqueue_head(&queue);
```

How to sleep (2)

- Several ways to make a kernel process sleep

wait_event(queue, condition);
    Sleeps until the given C expression is true.
    Caution: can't be interrupted (can't kill the user-space process!)

wait_event_killable(queue, condition); (Since Linux 2.6.25)
    Sleeps until the given C expression is true.
    Can only be interrupted by a “fatal” signal (SIGKILL)

wait_event_interruptible(queue, condition);
    Can be interrupted by any signal

wait_event_timeout(queue, condition, timeout);
    Sleeps and automatically wakes up after the given timeout.

wait_event_interruptible_timeout(queue, condition, timeout);
    Same as above, interruptible.

How to sleep - Example

- From drivers/ieee1394/video1394.c
- ```
  wait_event_interruptible(
      d->waitq,
      (d->buffer_status[v.buffer]
       == VIDEO1394_BUFFER_READY)
  );

  if (signal_pending(current))
      return -EINTR;
  ```

Currently running process
  ```

Waking up!

- Typically done by interrupt handlers when data sleeping processes are waiting for are available.

wake_up(queue);
Wakes up all the waiting processes on the given queue

wake_up_interruptible(queue);
Wakes up only the processes waiting in an interruptible sleep on the given queue

For all processes waiting in queue, condition is evaluated. When it evaluates to true, the process is put back to the TASK_RUNNING state, and the need_resched flag for the current process is set.

When is Scheduling Run?

- Each process has a need_resched flag which is set:

After a process exhausted its time slice.

After a process with a higher priority is awakened.

This flag is checked (possibly causing the execution of the scheduler):

When returning to user-space from a system call.

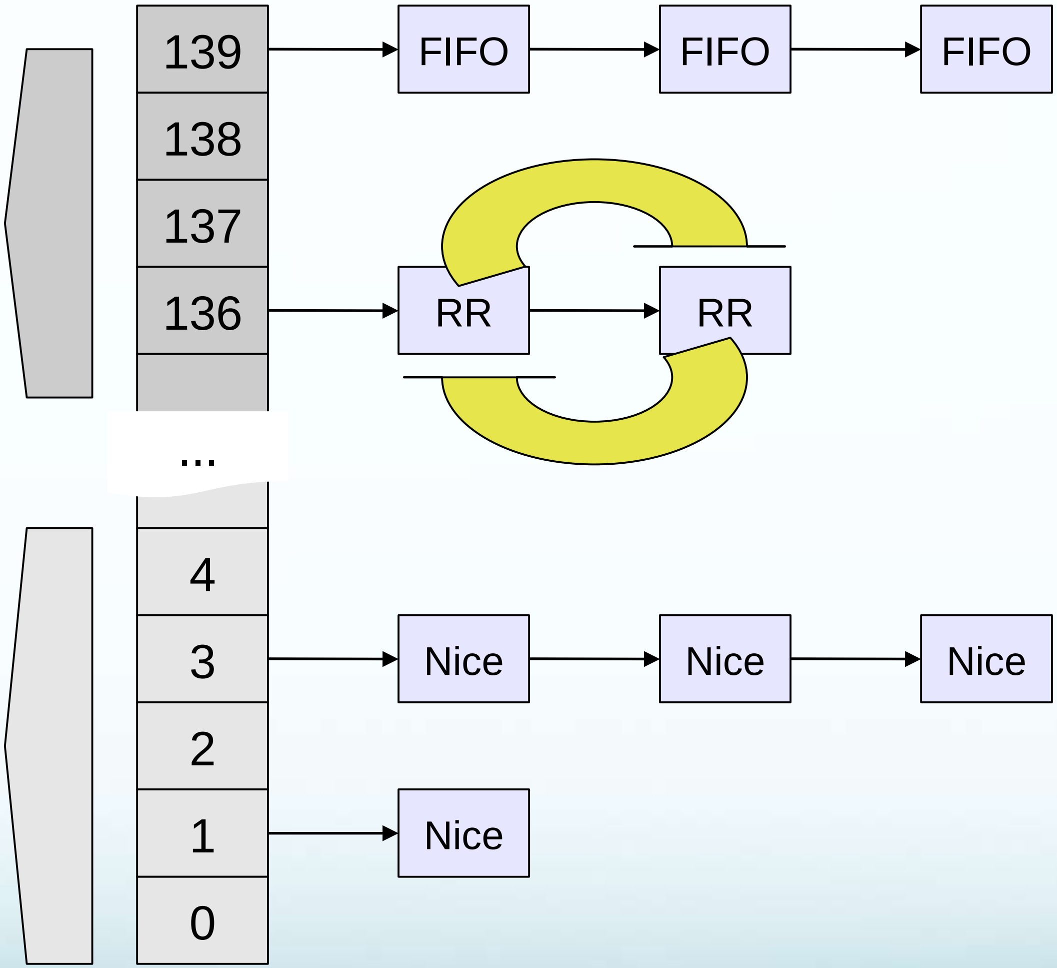
When returning from an interrupt handler (including the CPU timer).

Scheduling also happens when kernel code explicitly calls schedule() or executes an action that sleeps.

The Run Queues

Real Time
SCHED_FIFO
SCHED_RR
40 - 139

Nice
SCHED_OTHER
SCHED_BATCH
1 - 39



FIFO tasks run until they yield, block, exit or preempted by higher priority task.

RR tasks run until they yield, block, exit, preempted by higher priority task or run out of time slice, in which case next task is of the same priority.

Nice tasks run until they yield, block, exit preempted by higher priority task or run out of time slice, in which case next time might be of lower priority.

Dynamic Priorities

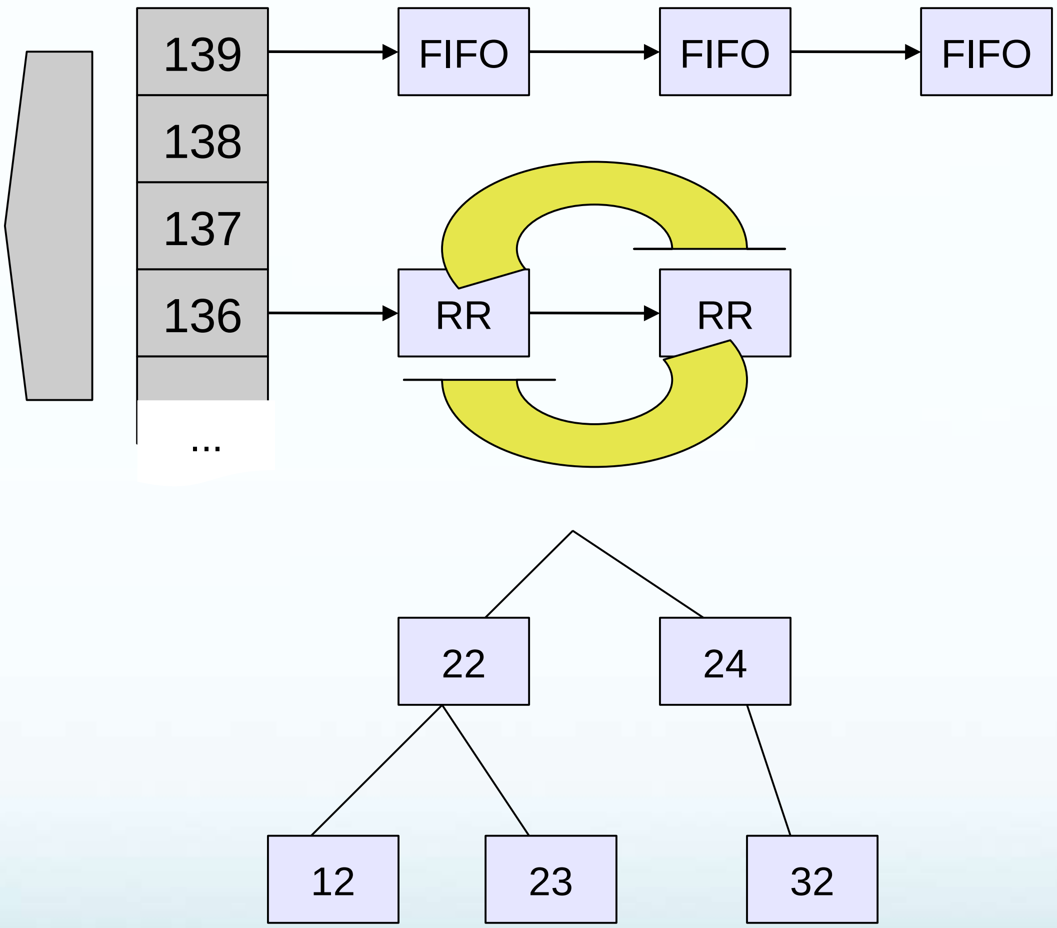
- Only applies to regular processes.

For a better user experience, the Linux scheduler boots the priority of interactive processes (processes which spend most of their time sleeping, and take time to exhaust their time slices). Such processes often sleep but need to respond quickly after waking up (example: word processor waiting for key presses).
Priority bonus: up to 5 points.

Conversely, the Linux scheduler reduces the priority of compute intensive tasks (which quickly exhaust their time slices).
Priority penalty: up to 5 points.

CFS Work Queues and Tree

Real Time
SCHED_FIFO
SCHED_RR
40 - 139



FIFO tasks run until they yield, block, exit or preempted by higher priority task.

RR tasks run until they yield, block, exit, preempted by higher priority task or run out of time slice, in which case next task is of the same priority.

Nice
SCHED_OTHER
SCHED_BATCH
1 - 39

Nice tasks run until they yield, block, exit preempted by higher priority task or run when another task difference from it's fair share is bigger.

MABEL Technology
Tech The Best

The CFS Data structure

The CFS holds all nice level tasks in a red-black tree, sorted according to the time the task needs to run to be balanced minus it's fair share of the CPU.

Therefore, the leftmost task in the tree (smallest value) is the one which the scheduler should pick next.

An adjustable granularity time guarantees against too often task switches.

This red-black tree algorithm is O(log n), which is a small drawback, considering the previous scheduler was O(1).

Driver Development

Interrupt Management

Interrupt handler constraints

Not run from a user context:
Can't transfer data to and from user space (need to be done by system call handlers)

Interrupt handler execution is managed by the CPU, not by the scheduler. Handlers can't run actions that may sleep, because there is nothing to resume their execution. In particular, need to allocate memory with GFP_ATOMIC.

Have to complete their job quickly enough:
they shouldn't block their interrupt line for too long.

Registering an interrupt handler (1)

- Defined in include/linux/interrupt.h

```
int request_irq(                                  Returns 0 if successful
      unsigned int irq,                           Requested irq channel
      irqreturn_t handler,                        Interrupt handler
      unsigned long irq_flags,                    Option mask (see next page)
      const char * devname,                       Registered name
      void *dev_id);                              Pointer to some handler data
                                                  Cannot be NULL and must be
   unique for shared irqs!

void free_irq( unsigned int irq, void *dev_id);
```

dev_id cannot be NULL and must be unique for shared irqs.
Otherwise, on a shared interrupt line,
free_irq wouldn't know which handler to free.

Registering an interrupt handler (2)

- irq_flags bit values (can be combined, none is fine too)

IRQF_DISABLED

"Quick" interrupt handler. Run with all interrupts disabled on the current cpu (instead of just the current line). For latency reasons, should only be used when needed!

IRQF_SHARED

Run with interrupts disabled only on the current irq line and on the local cpu. The interrupt channel can be shared by several devices. Requires a hardware status register telling whether an IRQ was raised or not.

IRQF_SAMPLE_RANDOM

Interrupts can be used to contribute to the system entropy pool used by /dev/random and /dev/urandom. Useful to generate good random numbers. Don't use this if the interrupt behavior of your device is predictable!

Information on installed handlers

- /proc/interrupts

	CPU0		
0:	5616905	XT-PIC	timer # Registered name
1:	9828	XT-PIC	i8042
2:	0	XT-PIC	cascade
3:	1014243	XT-PIC	orinoco_cs
7:	184	XT-PIC	Intel 82801DB-ICH4
8:	1	XT-PIC	rtc
9:	2	XT-PIC	acpi
11:	566583	XT-PIC	ehci_hcd, uhci_hcd, yenta, radeon@PCI:1:0:0
12:	5466	XT-PIC	i8042
14:	121043	XT-PIC	ide0
15:	200888	XT-PIC	ide1
NMI:	0		Non Maskable Interrupts
ERR:	0		Spurious interrupt count

The interrupt handler's job

Acknowledge the interrupt to the device (otherwise no more interrupts will be generated)

Read/write data from/to the device

Wake up any waiting process waiting for the completion of this read/write operation:
wake_up_interruptible(&module_queue);

Interrupt handler prototype

- irqreturn_t (*handler) (

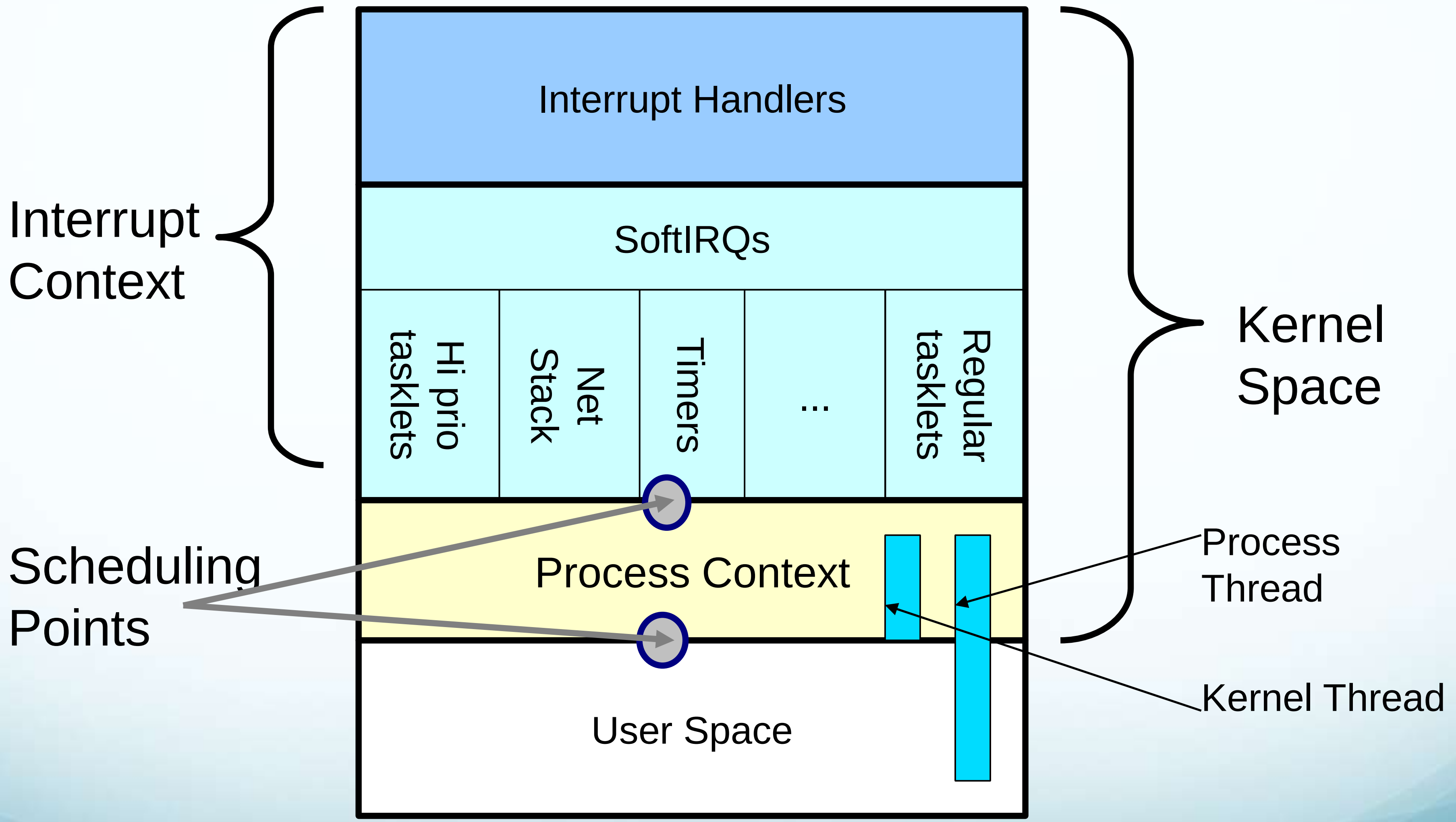
```
  int,              // irq number of the current interrupt
  void *dev_id,     // Pointer used to keep track
                    // of the corresponding device.
                    // Useful when several devices
                    // are managed by the same module
);
```

- Return value:

IRQ_HANDLED: recognized and handled interrupt

IRQ_NONE: not on a device managed by the module. Useful to share interrupt channels and/or report spurious interrupts to the kernel.

Linux Contexts



Top half and bottom half processing (1)

- Splitting the execution of interrupt handlers in 2 parts

Top half: the interrupt handler must complete as quickly as possible. Once it acknowledged the interrupt, it just schedules the lengthy rest of the job taking care of the data, for a later execution.

Bottom half: completing the rest of the interrupt handler job. Handles data, and then wakes up any waiting user process.
Best implemented by *tasklets* (also called *soft irqs*).

Softirq

A fixed set (max 32) of software interrupts (prioritized):

HI_SOFTIRQ	Runs low latency tasklets
TIMER_SOFTIRQ	Runs timers
NET_TX_SOFTIRQ	Network stack Tx
NET_RX_SOFTIRQ	Network stack Rx
SCSI_SOFTIRQ	SCSI sub system
TASKLET_SOFTIRQ	Runs normal tasklets

Activated on return from interrupt (in do_IRQ())

Can run concurrently on SMP systems (even the same softirq).

top half and bottom half processing (2)

Declare the tasklet in the module source file:

```
DECLARE_TASKLET (module_tasklet,    /* name */
                 module_do_tasklet, /* function */
                 data               /* params */
);
```

Schedule the tasklet in the top half part (interrupt handler):

```
tasklet_schedule(&module_tasklet);
```

Note that a tasklet_hi_schedule function is available to define high priority tasklets to run before ordinary ones.

By default, tasklets are executed right after all top halves (hard irqs)

Handling Floods

Normally, pending softirqs (including tasklets) will be run after each interrupt.

- A pending softirq is marked in a special bit field.
- The function that handles this is called *do_softirq()* and it is called by *do_IRQ()* function.

If after *do_softirq()* called the handler for that softirq, the softirq is still pending (the bit is on), it will **not** call the softirq again.

Instead, a low priority kernel thread, called *ksoftirqd*, is woken up. It will execute the softirq handler **when it is next scheduled.**

Work Queues

Each work queue has a kernel thread (task) per CPU.

- Since 2.6.6 also a single threaded version exists.

Code in a work queue:

- Has a process context.
- May sleep.

New work queues may be created/destroyed via:

- struct workqueue_struct ***create_workqueue**(const char *name);
- struct workqueue_struct ***create_singlethread_workqueue**(const char *name);
- void **destroy_workqueue**(const char *name);

Working the Work Queue

Declare a work structure:

DECLARE_WORK(work, func, data);

INIT_WORK(work, func, data);

Queue the work:

int **queue_work**(struct workqueue_struct *wq, struct work_struct *work);

int **queue_delayed_work**(struct workqueue_struct *wq, struct work_struct *work, unsigned long delay);

Wait for all work to finish:

int **flush_workqueue**(struct workqueue_struct *wq);

The Default Work Queue

One “default” work queue is run by the *events* kernel thread (also known as the *keventd_wq* in the sources).

For the *events* work queue, we have the more common:

- int **schedule_work**(struct work_struct *work);
- int **schedule_delayed_work**(struct work_struct *work, unsigned long delay);
- int **cancel_delayed_work**(struct work_struct *work);
- int **flush_scheduled_work**(void);
- int **current_is_keventd**(void);

Disabling interrupts

- May be useful in regular driver code...

Can be useful to ensure that an interrupt handler will not preempt your code (including kernel preemption)

Disabling interrupts on the local CPU:

```
unsigned long flags;
local_irq_save(flags);           // Interrupts disabled
...
local_irq_restore(flags); // Interrupts restored to their previous state.
```

Note: must be run from within the same function!

Masking out an interrupt line

- Useful to disable interrupts **on a particular line**

void disable_irq (unsigned int irq);
Disables the irq line for all processors in the system.
Waits for all currently executing handlers to complete.

void disable_irq_nosync (unsigned int irq);
Same, except it doesn't wait for handlers to complete.

void enable_irq (unsigned int irq);
Restores interrupts on the irq line.

void synchronize_irq (unsigned int irq);
Waits for irq handlers to complete (if any).

Checking interrupt status

- Can be useful for code which can be run from both process or interrupt context, to know whether it is allowed or not to call code that may sleep.

irqs_disabled()
Tests whether local interrupt delivery is disabled.

in_interrupt()
Tests whether code is running in interrupt context

in_irq()
Tests whether code is running in an interrupt handler.

Interrupt Management Summary

- Device driver

When the device file is first open, register an interrupt handler for the device's interrupt channel.

Interrupt handler

Called when an interrupt is raised.

Acknowledge the interrupt.

If needed, schedule a tasklet or work queue taking care of handling data.

Otherwise, wake up processes waiting for the data.

- Tasklet

Process the data.

Wake up processes waiting for the data.

Device driver

When the device is no longer opened by any process, unregister the interrupt handler.

MABEL Technology
Tech The Best

Linux Internals

Driver development

DMA

DMA situations

- Synchronous

A user process calls the read method of a driver. The driver allocates a DMA buffer and asks the hardware to copy its data. The process is put in sleep mode.

The hardware copies its data and raises an interrupt at the end.

The interrupt handler gets the data from the buffer and wakes up the waiting process.

- Asynchronous

The hardware sends an interrupt to announce new data.

The interrupt handler allocates a DMA buffer and tells the hardware where to transfer data.

The hardware writes the data and raises a new interrupt.

The handler releases the new data, and wakes up the needed processes.

Memory constraints

Need to use contiguous memory in physical space

Can use any memory allocated by kmalloc (up to 128 KB) or __get_free_pages (up to 8MB)

Can use block I/O and networking buffers, designed to support DMA.

Can not use vmalloc memory (would have to setup DMA on each individual page)

Reserving memory for DMA

- To make sure you've got enough RAM for big DMA transfers...
  Example assuming you have 32 MB of RAM, and need 2 MB for DMA:

Boot your kernel with mem=30
  The kernel will just use the first 30 MB of RAM.

Driver code can now reclaim the 2 MB left:

```
dmabuf = ioremap (
        0x1e00000,      /* Start: 30 MB */
        0x200000        /* Size: 2 MB */
        );
```

Memory synchronization issues

- Memory caching could interfere with DMA

Before DMA to device:
Need to make sure that all writes to DMA buffer are committed.

After DMA from device:
Before drivers read from DMA buffer, need to make sure that memory caches are flushed.

Bidirectional DMA
Need to flush caches before and after the DMA transfer.

Linux DMA API

- The kernel DMA utilities can take care of:

Either allocating a buffer in a cache coherent area,

Or make sure caches are flushed when required,

Managing the DMA mappings and IOMMU (if any)

See Documentation/DMA-API.txt
for details about the Linux DMA generic API.

Most subsystems (such as PCI or USB) supply their own DMA API, derived from the generic one. May be sufficient for most needs.

Coherent or streaming DMA mappings

Coherent mappings

- Can simultaneously be accessed by the CPU and device.
- So, have to be in a cache coherent memory area.
- Usually allocated for the whole time the module is loaded.
- Can be expensive to setup and use.

Streaming mappings (recommended)

- Set up for each transfer.
- Keep DMA registers free on the physical hardware registers. Some optimizations also available.

Allocating coherent mappings

- The kernel takes care of both the buffer allocation and mapping:
- include <asm/dma-mapping.h>
- ```
  void *                                   /* Output: buffer address */
  dma_alloc_coherent(
      struct device *dev,       /* device structure */
      size_t size,              /* Needed buffer size in bytes */
      dma_addr_t *handle,       /* Output: DMA bus address */
      gfp_t gfp                 /* Standard GFP flags */
  );
  ```
- ```
  void dma_free_coherent(struct device *dev,
  size_t size, void *cpu_addr, dma_addr_t handle);
  ```

DMA pools (1)

dma_alloc_coherent usually allocates buffers with __get_free_pages (minimum: 1 page).

You can use DMA pools to allocate smaller coherent mappings:

<include linux/dmapool.h>

Create a dma pool:

```
struct dma_pool *
dma_pool_create (
    const char *name,         /* Name string */
    struct device *dev,       /* device structure */
    size_t size,              /* Size of pool buffers */
    size_t align,             /* Hardware alignment (bytes) */
    size_t allocation         /* Address boundaries not to be crossed */
);
```

DMA pools (2)

Allocate from pool

```
void * dma_pool_alloc (
    struct dma_pool *pool,
    gfp_t mem_flags,
    dma_addr_t *handle
);
```

Free buffer from pool

```
void dma_pool_free (
                        struct dma_pool *pool,
                        void *vaddr,
                        dma_addr_t dma);
```

Destroy the pool (free all buffers first!)

```
void dma_pool_destroy (struct dma_pool *pool);
```

Setting up streaming mappings

- Works on buffers already allocated by the driver
- <include linux/dmapool.h>
- ```
  dma_addr_t dma_map_single(
  struct device *,               /* device structure */
  void *,                        /* input: buffer to use */
  size_t,                        /* buffer size */
  enum dma_data_direction        /* Either DMA_BIDIRECTIONAL,
                                    DMA_TO_DEVICE or
  DMA_FROM_DEVICE */
  );
  ```
- void dma_unmap_single(struct device *dev, dma_addr_t handle, size_t size, enum dma_data_direction dir);
  ```

DMA streaming mapping notes

When the mapping is active: only the device should access the buffer (potential cache issues otherwise).

The CPU can access the buffer only after unmapping!

Another reason: if required, this API can create an intermediate *bounce buffer* (used if the given buffer is not usable for DMA).

Possible for the CPU to access the buffer without unmapping it, using the dma_sync_single_for_cpu() (ownership to cpu) and dma_sync_single_for_device() functions (ownership back to device).

The Linux API also support scatter / gather DMA streaming mappings.

filesystems

- Block devices

Floppy or hard disks (SCSI, IDE)

Compact Flash (seen as a regular IDE drive)

RAM disks

Loopback devices

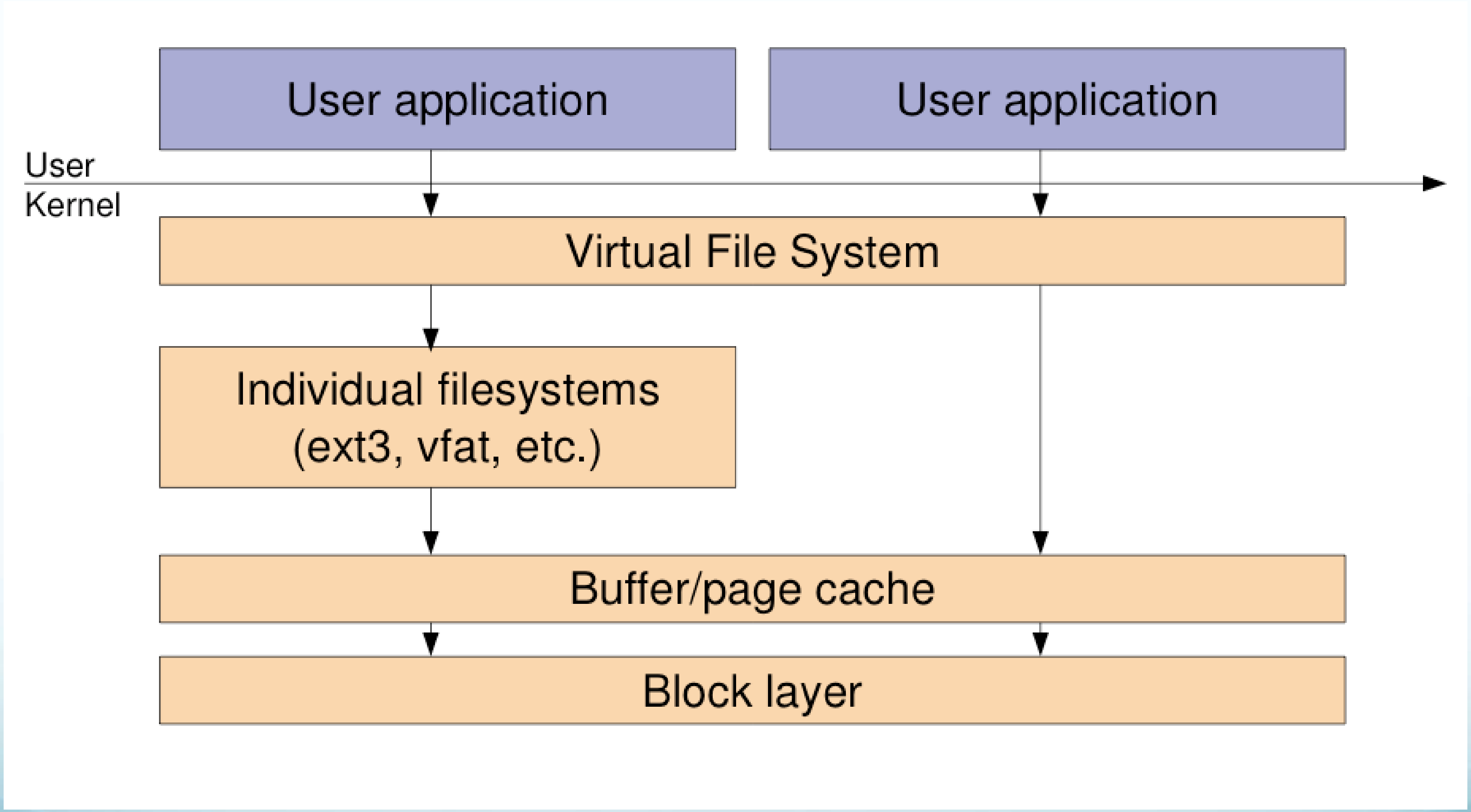
- Memory Technology Devices (MTD)

Flash, ROM or RAM chips

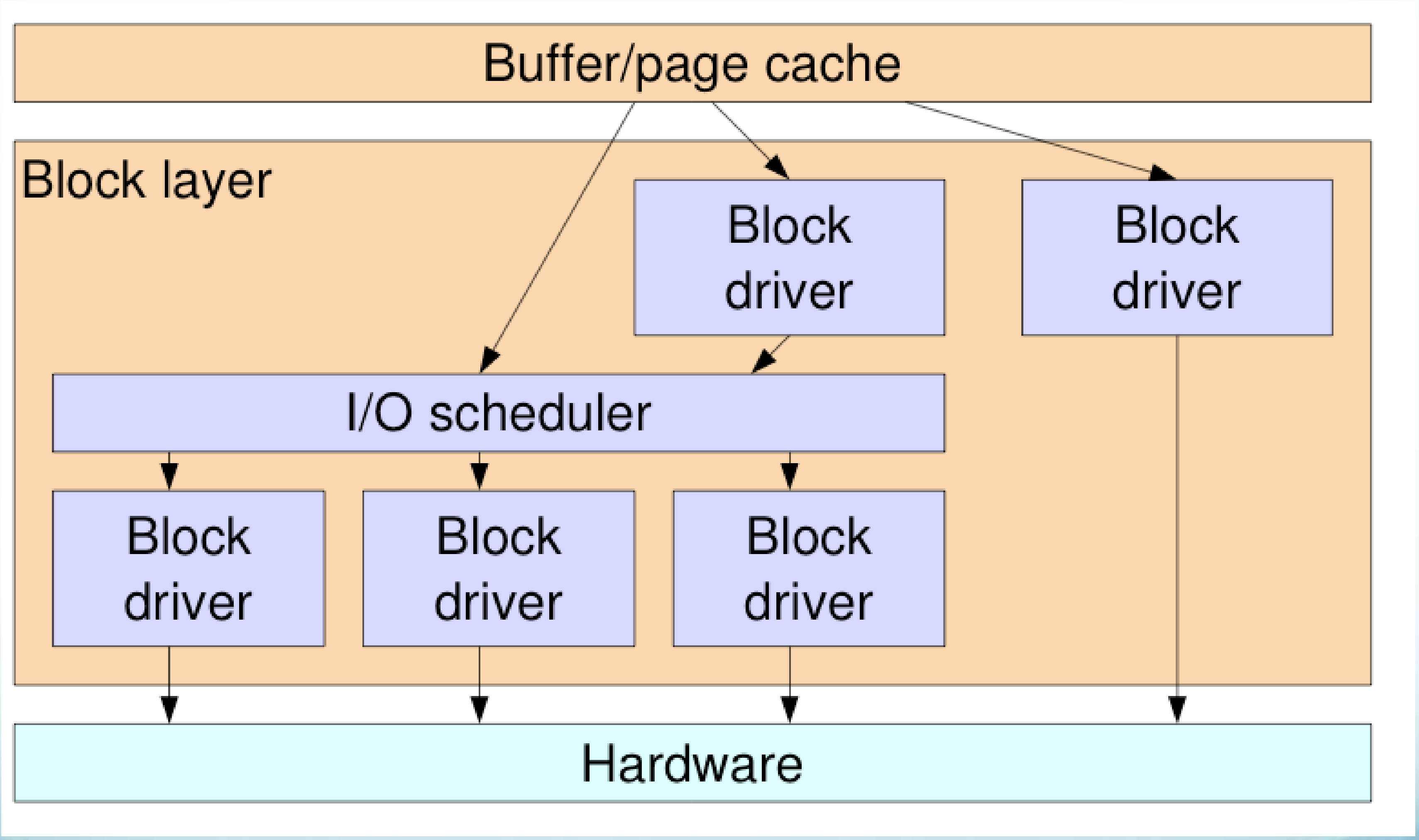
MTD emulation on block devices

Filesystems are either made for block or MTD storage devices. See Documentation/filesystems/ for details.

General Architecture

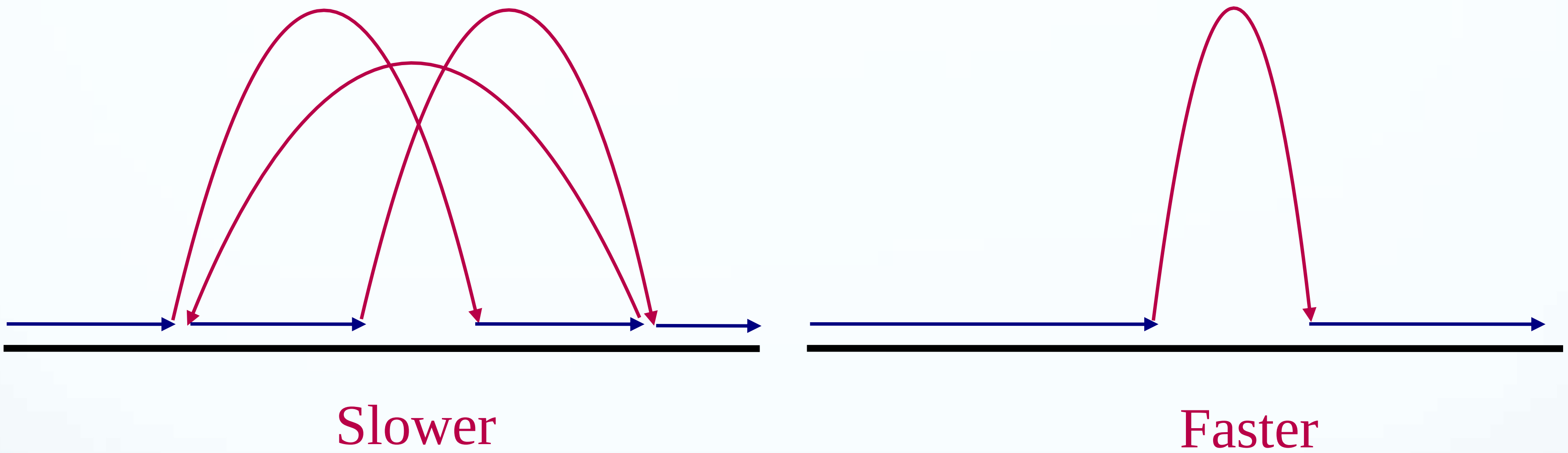


The Block Layer



I/O schedulers

Mission of I/O schedulers: re-order reads and writes to disk to minimize disk head moves (time consuming!)



2.4 has one fixed: the Linus Elevator.

2.6 has modular IO scheduler No-op, Elevator, Antciptory, Deadline, CFQ

VFS

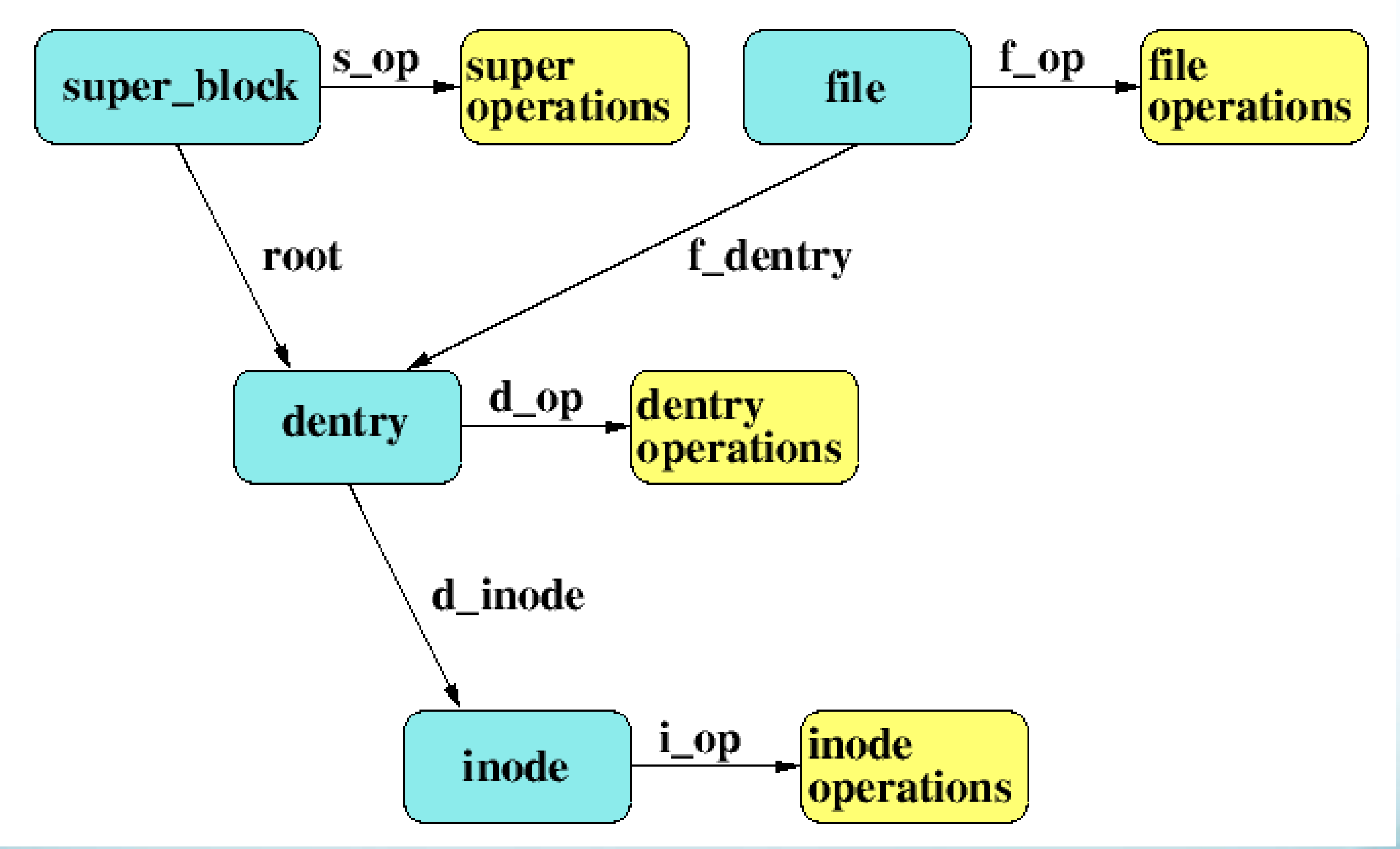
Linux provides a unified Virtual File System interface:

- The VFS layer supports abstract operations.
- Specific file systems implements them.

The major VFS abstract objects:

- **super_block** Represent a file system
- **Dentry** A directory entry. Maps names to inodes
- **inode** A file inode. Contains persistent information
- **file** An open file (file descriptor). Refers to dentry.

VFS Structures



Writing a file system module

- Implement the following structures:

  file_system_type

  super_operations

  file_operations

  inode_operations

  address_space_operations

Call register_filesystem on load time

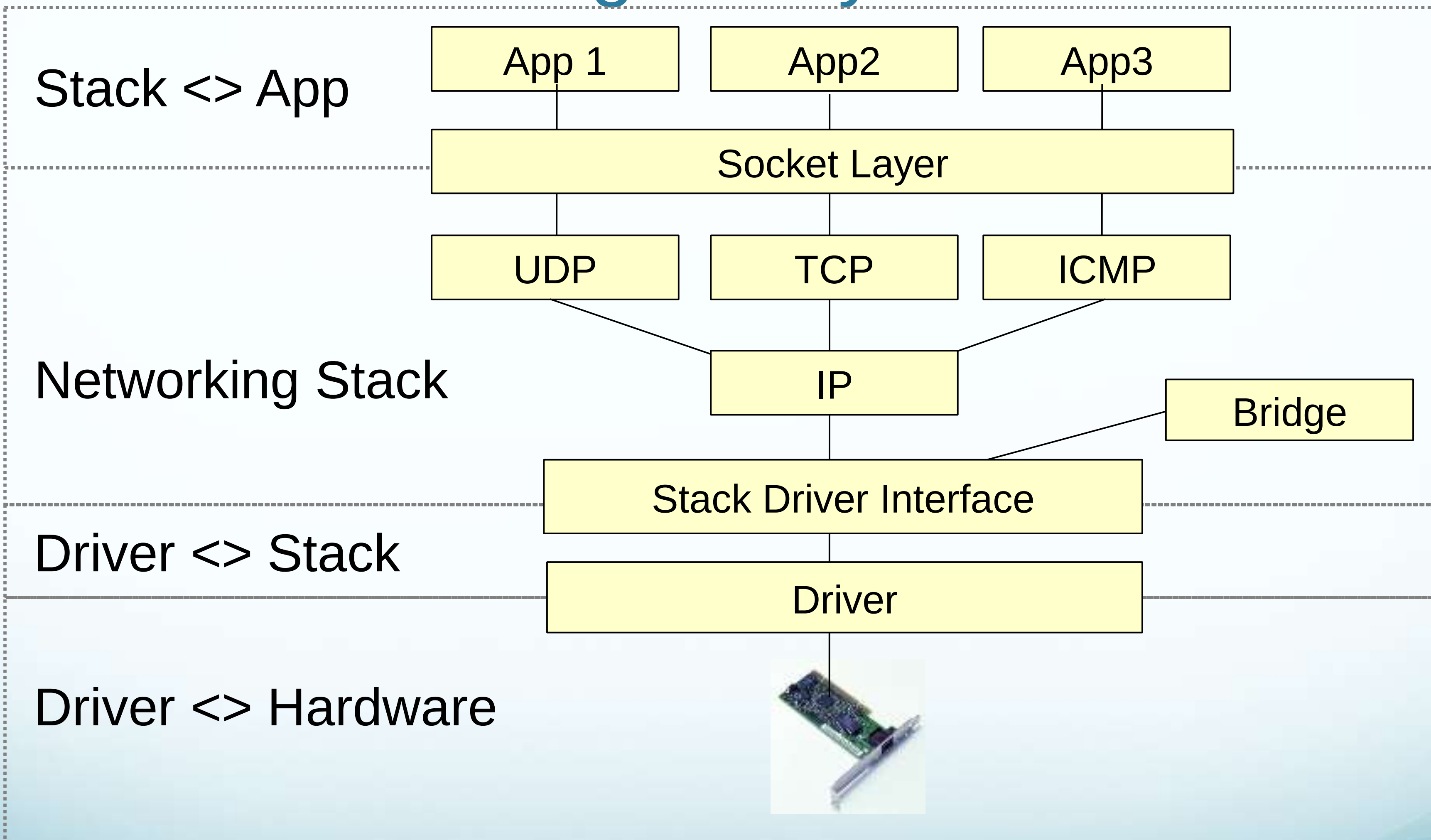
The network subsystem

MABEL Technology
Tech The Best

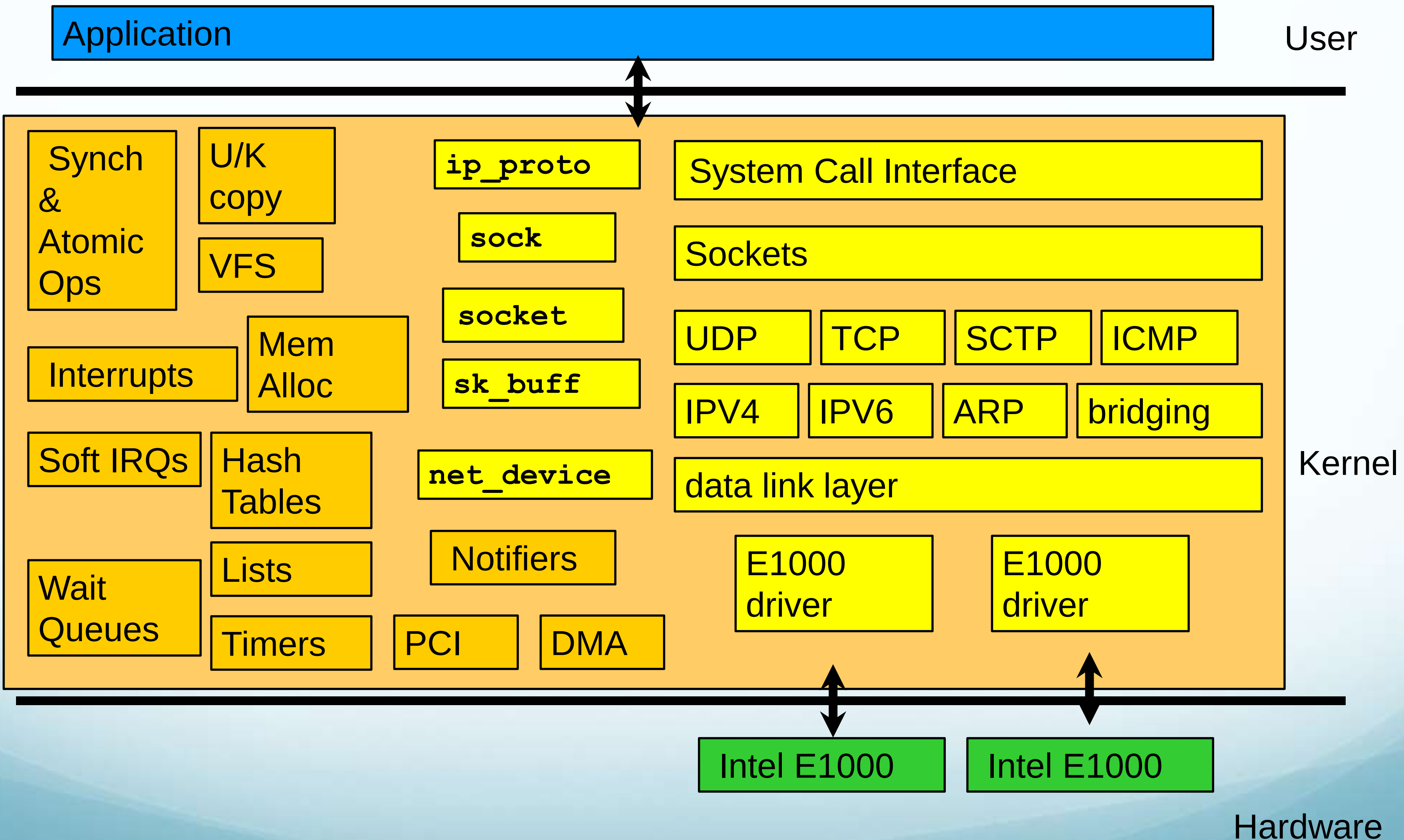
Network Systems Design and Implementation

- The Linux Networking Stack
  - + Mature (over 15 years old)
  - + *Open* (we can look at it)
  - + Relevant (large market share)
  - + Feature-rich (millions of LOCs)
    - Hundreds of protocols, devices, features
  - - Evolving (changes every day)

Linux networking Subsystem Overview



Network Subsystem



Network-specific facilities

- `sk_buff`:
  - Core networking data structure for managing data (i.e., packets)
- `net_device:`
  - Core data structure that represents a network interface (e.g., an Intel E1000 Ethernet NIC).
- `proto_ops:`
  - Data structure for different IP protocol families
    - `SOCK_STREAM, SOCK_DGRAM, SOCK_RAW`
    - Virtual functions for `bind()`, `accept()`, `connect()`, etc.
- `struct sock/ struct socket`:
  - Core data structures for representing sockets

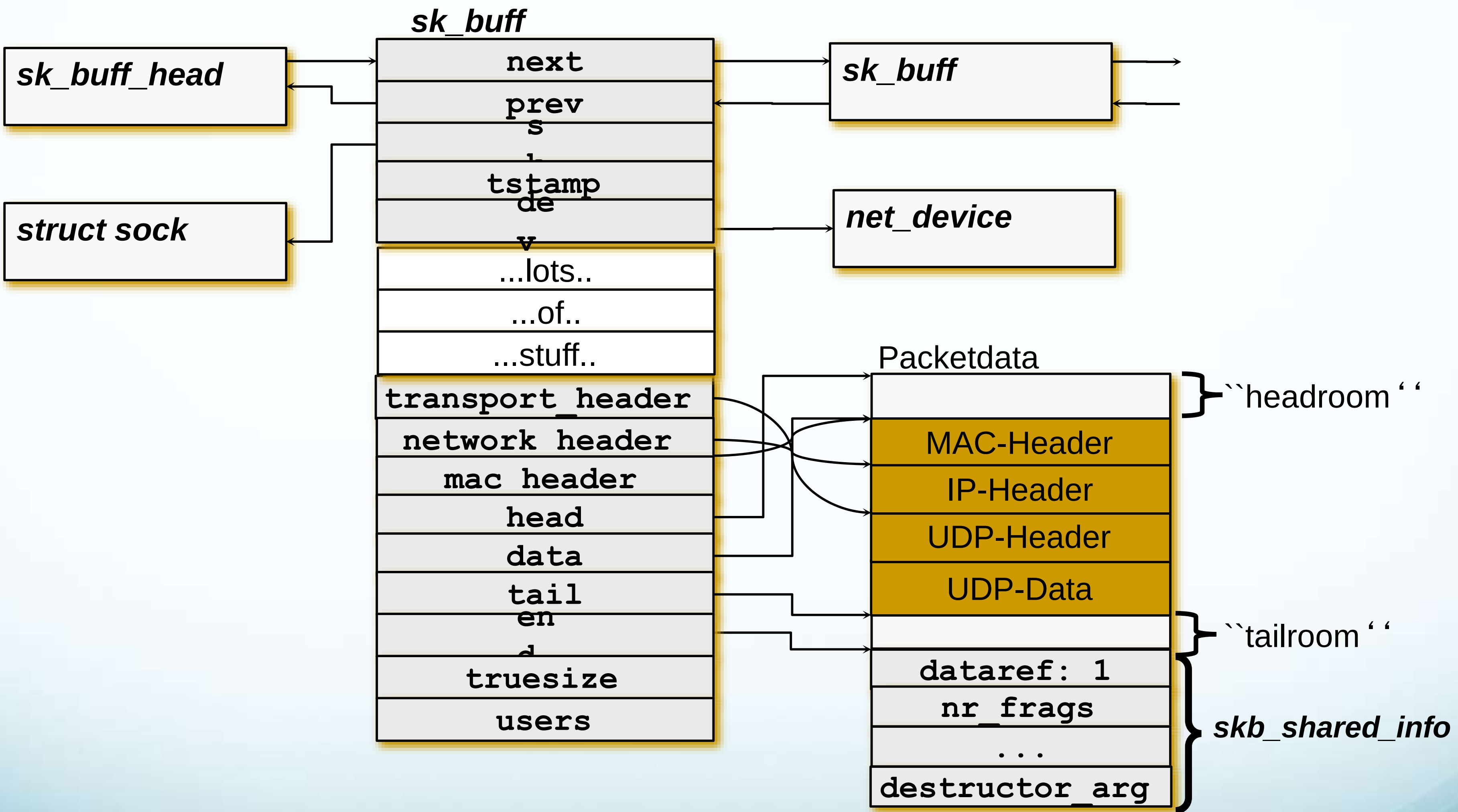
Socket Buffers (1)

- We need to manipulate packets through the stack
- This manipulation involves efficiently:
  - Adding protocol headers/trailers down the stack.
  - Removing protocol headers/trailers up the stack.
  - Concatenating/separating data.
- Each protocol should have convenient access to header fields.
- To do all this the kernel provides the sk_buff structure.

Socket Buffers (2)

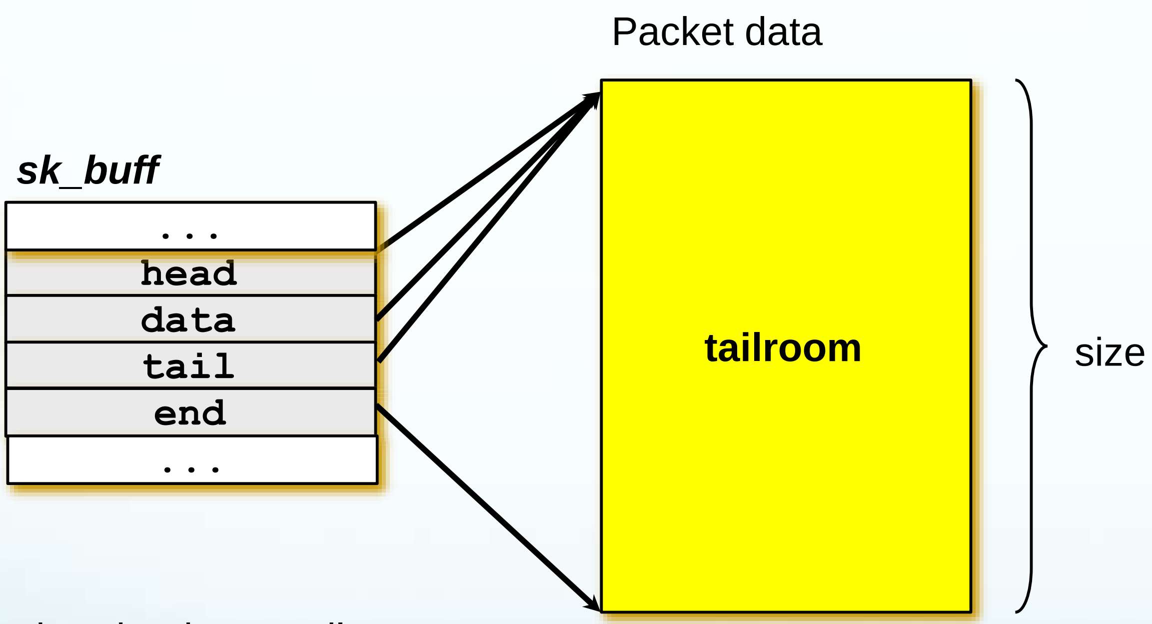
- Created when an application passes data to a socket or when a packet arrives at the network adaptor (*dev_alloc_skb()* is invoked).
- Packet headers of each layer are
  - Inserted in front of the payload on send
  - Removed from front of payload on receive
- The packet is (hopefully) copied only twice:
  1. Once from the user address space to the kernel address space via an explicit copy
  2. Once when the packet is passed to or from the network adaptor (usually via DMA)

Structure of sk_buff



linux-2.6.31/include/linux/skbuff.h

sk_buff after alloc_skb(size)

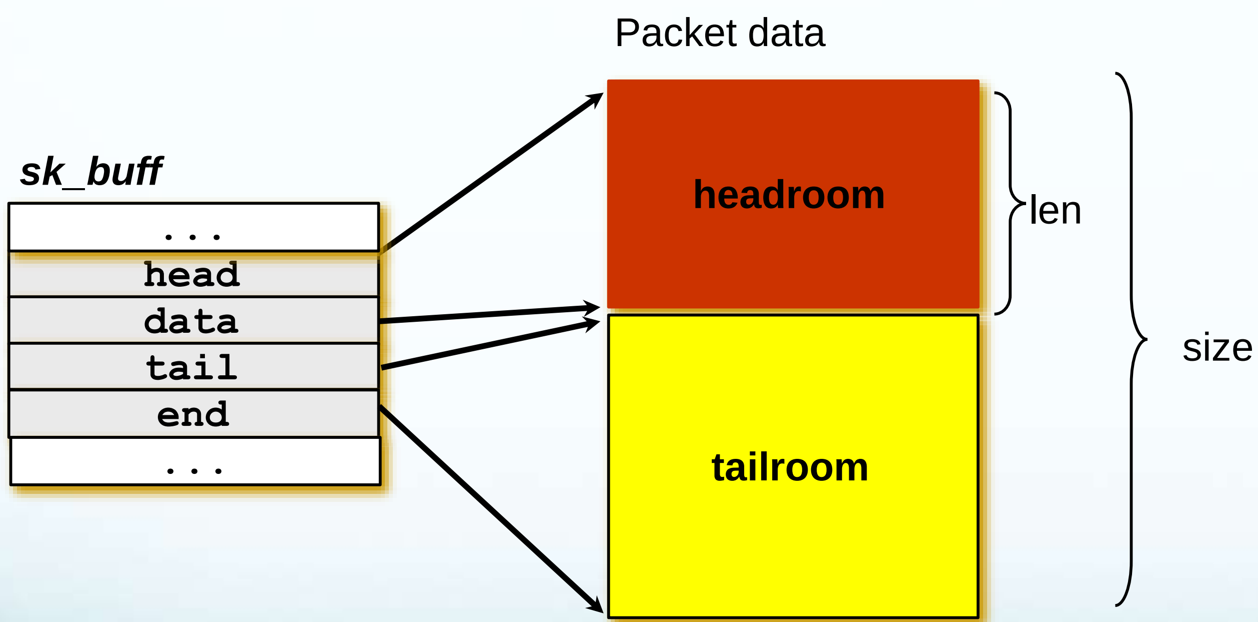


head = data = tail

end = tail + size

len = 0

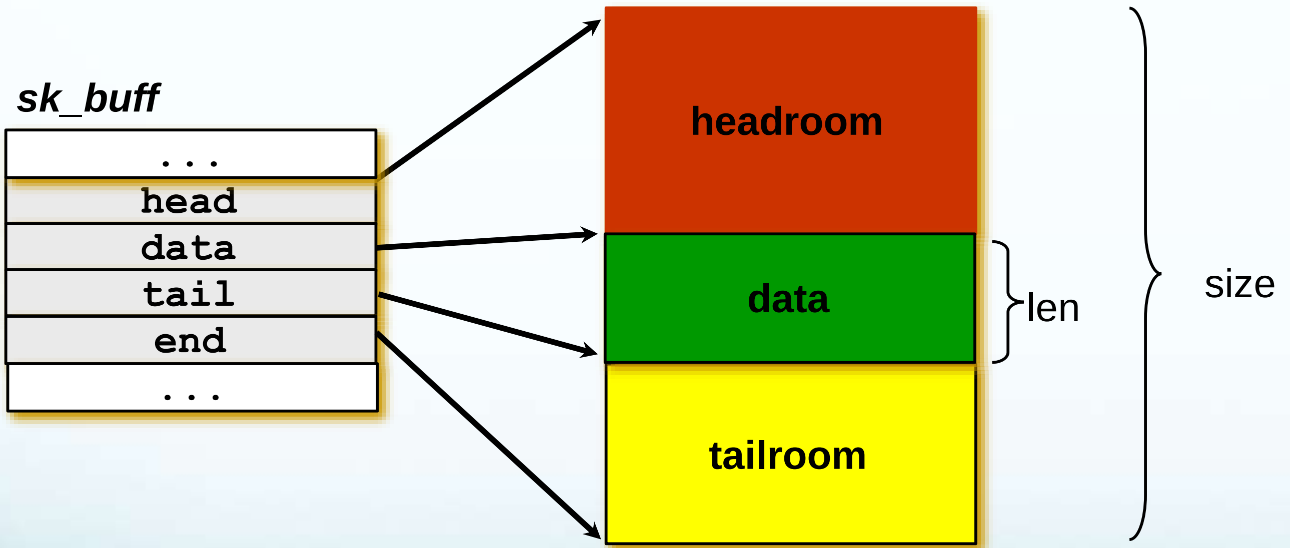
sk_buff after skb_reserve(len)



data += len
tail += len

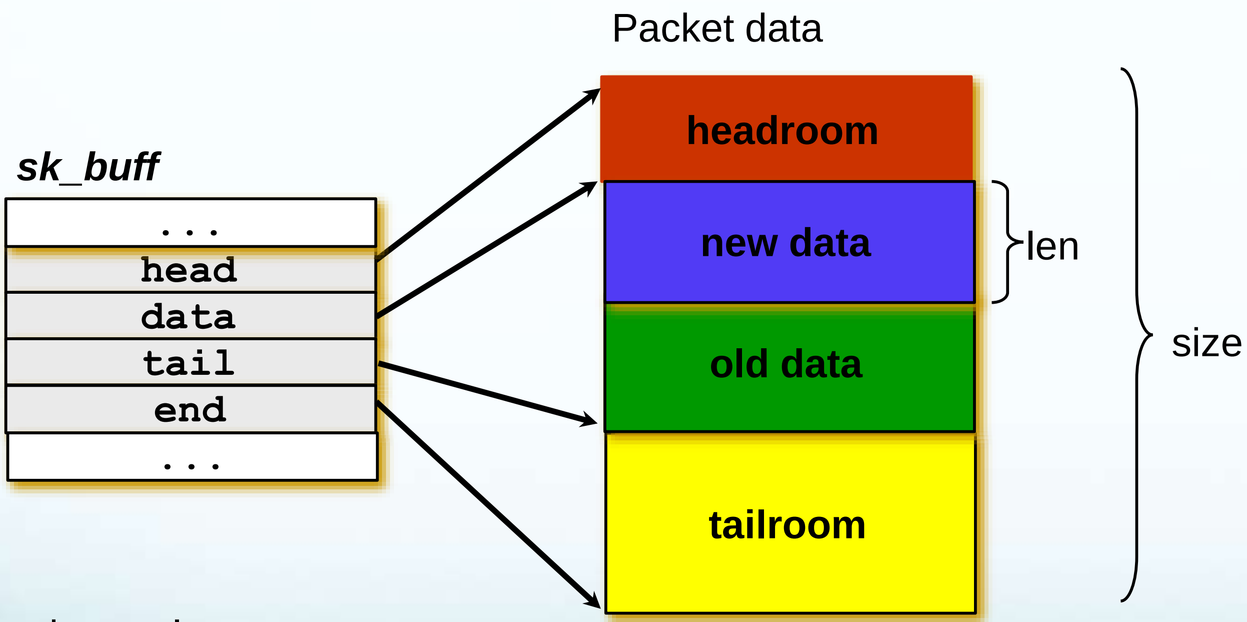
sk_buff after skb_put(len)

Packet data



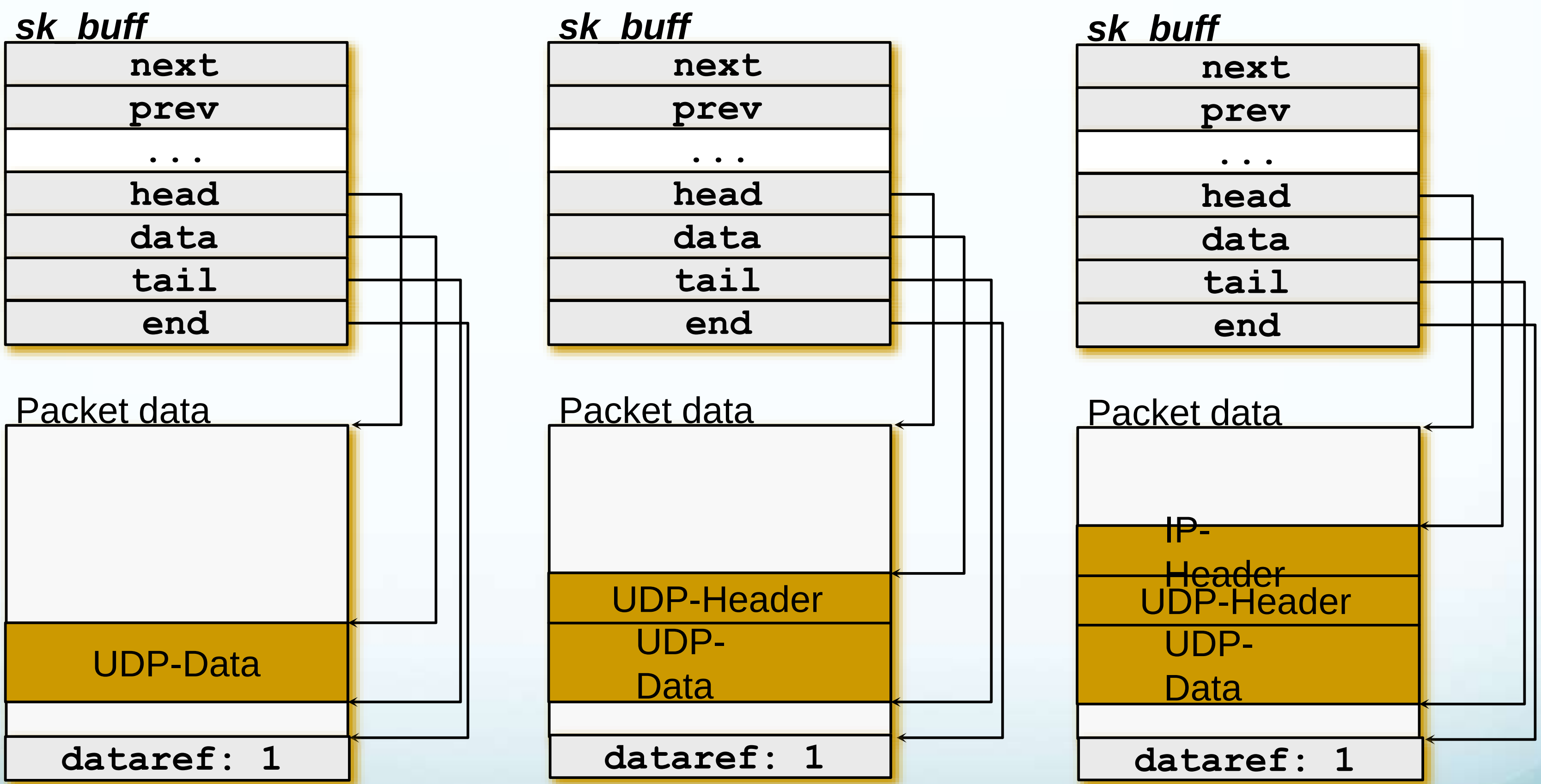
tail += len
len += len

sk_buff after skb_push(len)



data -= len
len += len

Changes in *sk_buff* as a Packet Traverses Across the Stack



Parameters of *sk_buff* Structure

- *sk*: points to the socket that created the packet (if available).
- *tstamp*: specifies the time when the packet arrived in the Linux (using *ktime*)
- *dev:* states the current network device on which the socket buffer operates. If a routing decision is made, dev points to the network adapter on which the packet leaves.
- *_skb_dst:* a reference to the adapter on which the packet leaves the computer
- *cloned:* indicates if a packet was cloned.

Parameters of *sk_buff* Structure

- *pkt_type*: specifies the type of a packet
  - PACKET_HOST: a packet sent to the local host
  - PACKET_BROADCAST: a broadcast packet
  - PACKET_MULTICAST: a multicast packet
  - PACKET_OTHERHOST: a packet not destined for the local host, but received in the promiscuous mode.
  - PACKET_OUTGOING: a packet leaving the host
  - PACKET_LOOKBACK: a packet sent by the local host to itself.

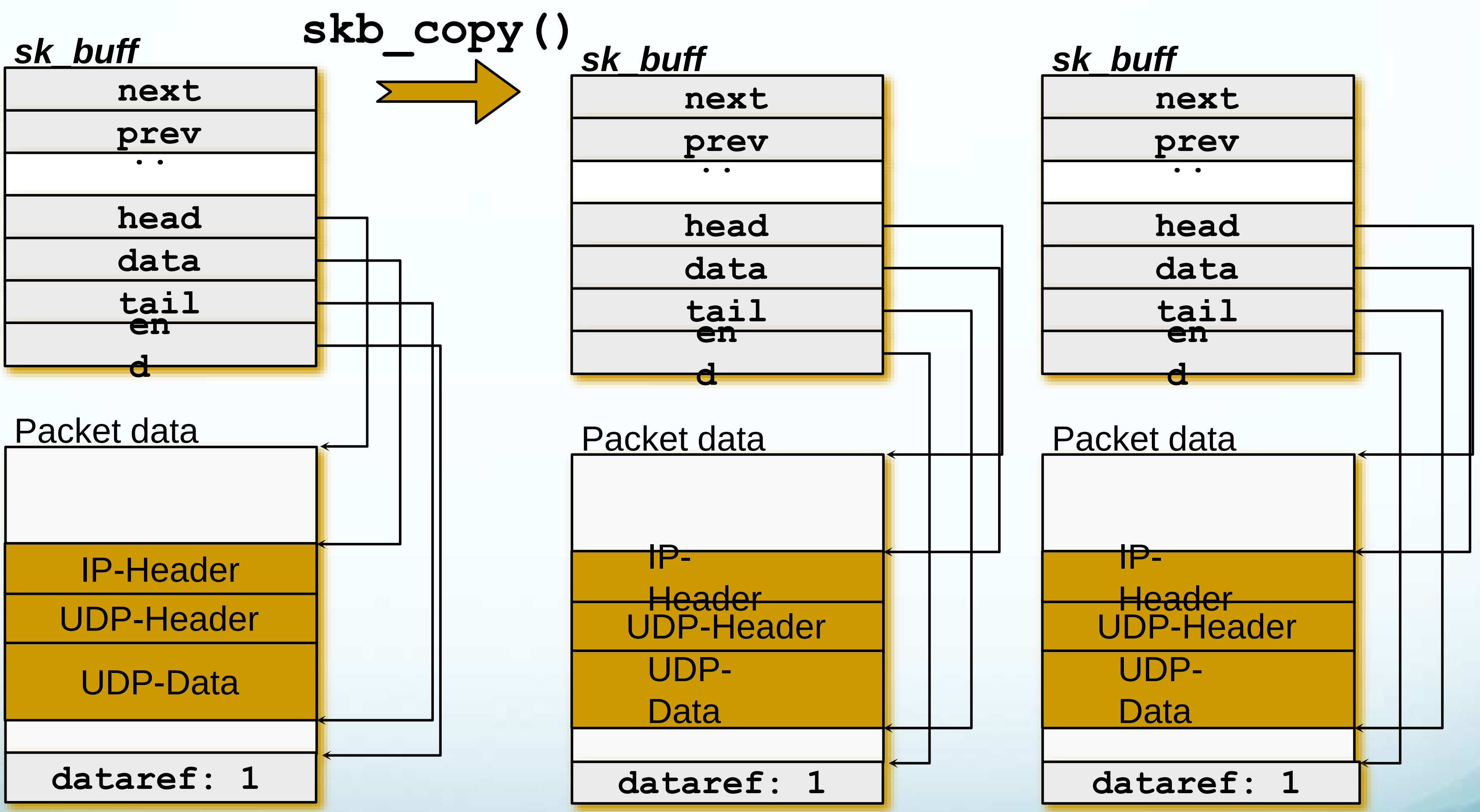
Creating Socket Buffers

- *alloc_skb(size, gfp_mask)*
  - Tries to reuse a *sk_buff* in the *skb_fclone_cache* queue; if not successful, tries to obtain a packet from the central socket-buffer cache (*skbuff_head_cache*) with *kmem_cache_alloc().*
  - If neither is successful, then invoke *kmalloc()* to reserve memory*.*
- *dev_alloc_skb(size)*
  - Same as alloc_skb but uses GFP_ATOMIC and reserves 32 bytes of headroom
- *netdev_alloc_skb(device, size)*
  - Same as dev_alloc_skb but uses a particular device (i.e., NUMA machines)

Creating Socket Buffers (2)

- *skb_copy(skb,gfp_mask):* creates a copy of the socket buffer *skb*, copying both the *sk_buff* structure and the packet data.
- *skb_copy_expand(skb,newheadroom, newtailroom, gfp_mask):* creates a new copy of the socket buffer and packet data, and in addition, reserves a larger space before and after the packet data.

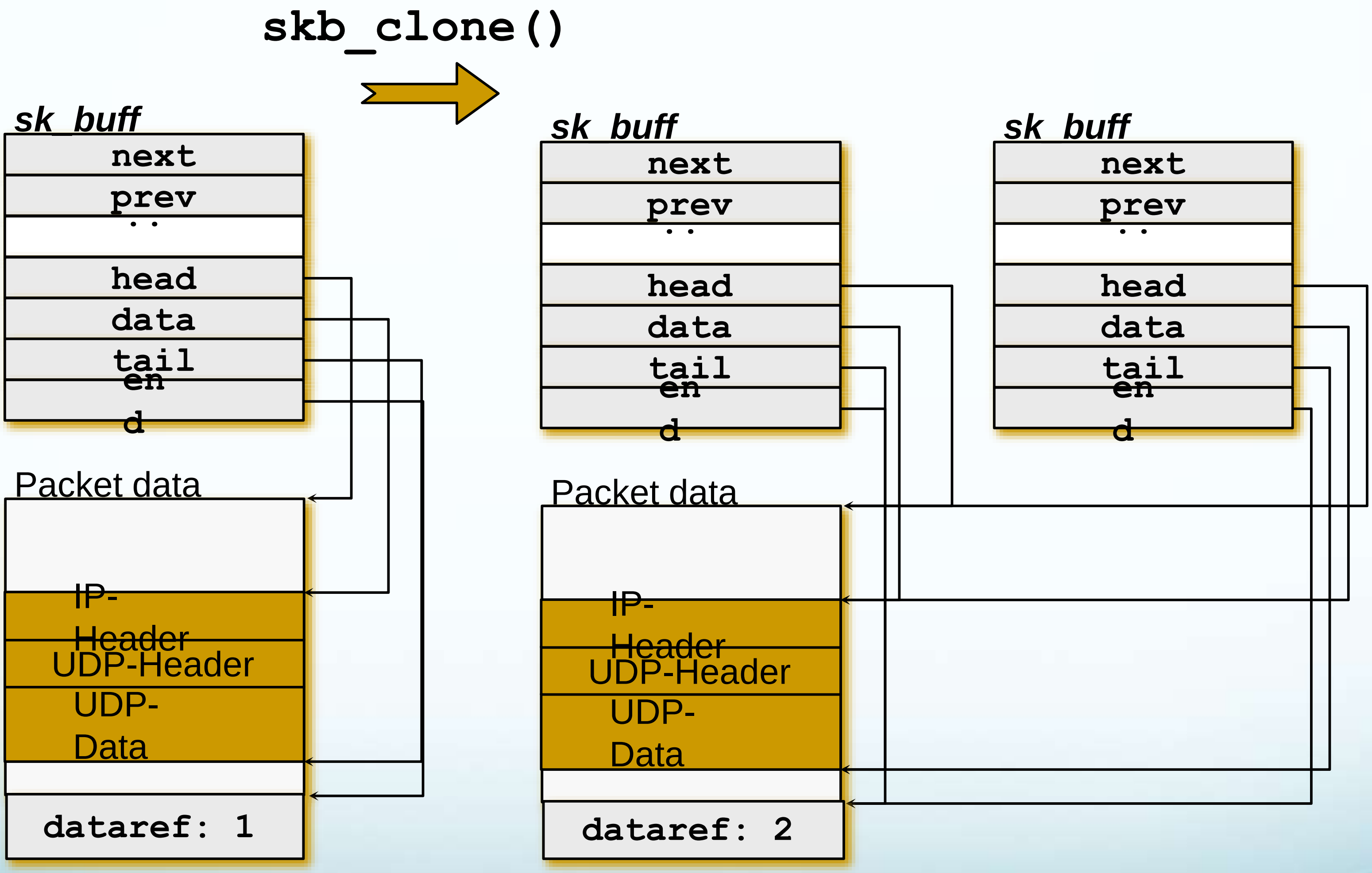
Copying Socket Buffers



Cloning Socket Buffers

- *skb_clone():* creates a new socket buffer sk_buff, but not the packet data. Pointers in both sk_buffs point to the same packet data space.
  - Used all over the place, e.g., *tcp_transmit_skb().*

Cloning Socket Buffers



Releasing Socket Buffers

- *kfree_skb():* decrements reference count for skb. If null, free the memory.
  - Used by the kernel, not meant to be used by drivers
- *dev_free_skb():*
  - For use by drivers in non-interrupt context
- *dev_free_skb_irq():*
  - For use by drivers in interrupt context
- *dev_free_skb_any():*
  - For use by drivers in any context

Manipulating sk_buffs

- *skb_put(skb,len):* appends data to the end of the packet; increments the pointer *tail* and *skbàlen* by *len*; need to ensure the *tailroom* is sufficient.
- *skb_push(skb,len):* inserts data in front of the packet data space; decrements the pointer *data* by *len*, and increment *skbàlen* by *len;* need to check the *headroom* size.
- *skb_pull(skb,len):* truncates *len* bytes at the beginning of a packet.
- *skb_trim(skb,len):* trim skb to len bytes (if necessary)

Manipulating sk_buffs (2)

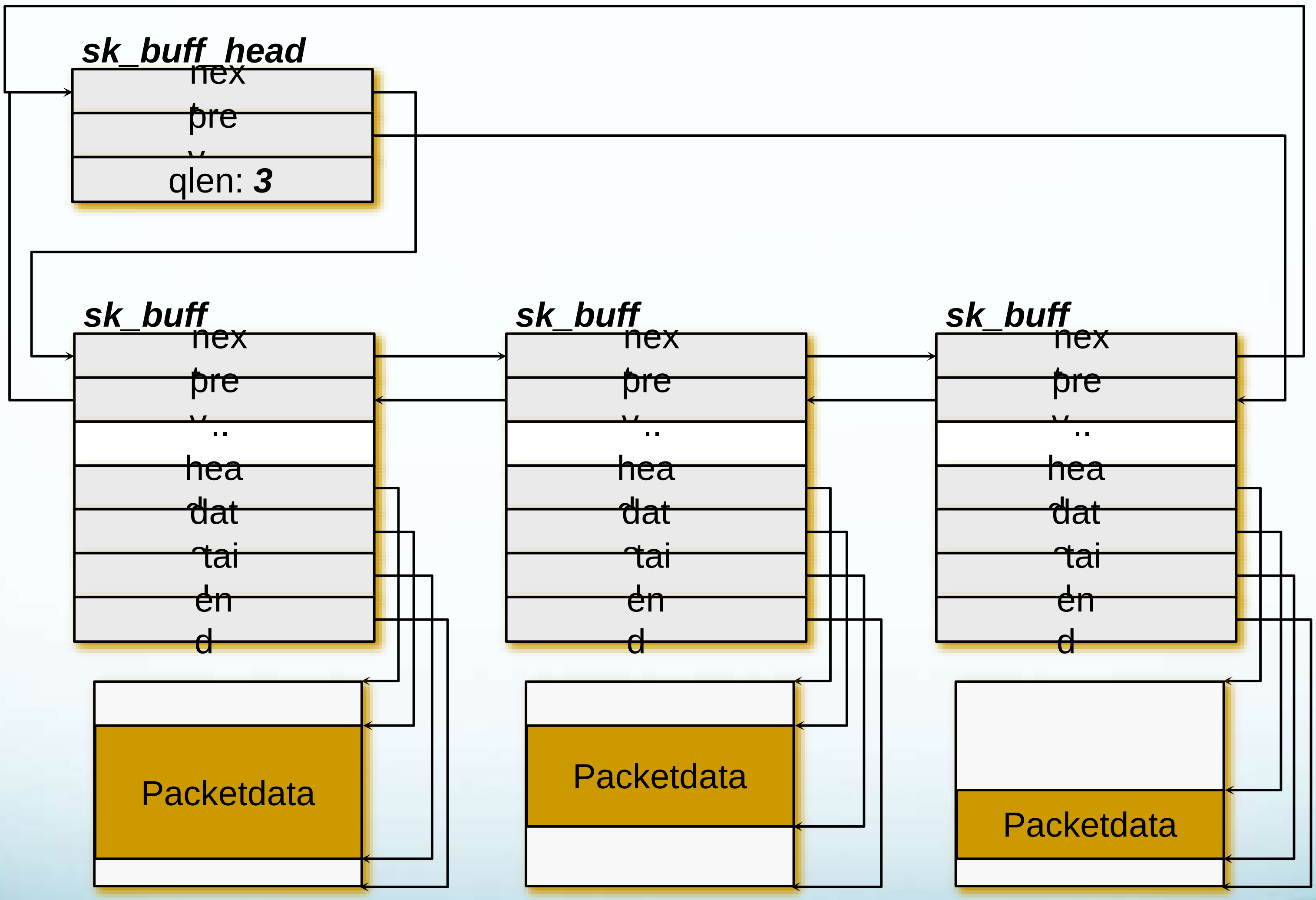
- *skb_tailroom(skb):* returns the size of the tailroom (in bytes).
- *skb_headroom(skb):* returns the size of the headroom *(data-head)*
- *skb_realloc_headroom(skb,newheadroom)* creates a new socket buffer with a headroom of size *newheadroom.*
- *skb_reserve(skb,len):* increases headroom by *len* bytes.

Socket Buffer Queues

Socket buffers are arranged in a dual-concatenated ring structure.

```
struct sk_buff_head {
    struct sk_buff  *next;
    struct sk_buff  *prev;
    __u32  qlen;
    spinlock_t      lock;
};
```

Socket Buffer Queues



Managing Socket Buffer Queues

- *skb_queue_head_init(list):* initializes an *skb_queue_head* structure
  - prev = next = self; qlen = 0;
- *skb_queue_empty(list):* checks whether the queue *list* is empty*;* checks if list == list->next
- *skb_queue_len(list):* returns length of the queue.
- *skb_queue_head(list, skb):* inserts the socket buffer *skb* at the head of the queue and increment *listàqlen* by one.
- *skb_queue_tail(list, skb):* appends the socket buffer *skb* to the end of the queue and increment *listàqlen* by one.

Managing Socket Buffer Queues

- *skb_dequeue(list):* removes the top skb from the queue and returns the pointer to the skb.
- *skb_dequeue_tail(list):* removes the last packet from the queue and returns the pointer to the packet.
- *skb_queue_purge():* empties the queue list; all packets are removed via *kfree_skb().*
- *skb_insert(oldskb, newskb, list):* inserts *newskb* <u>in front of</u> *oldskb* in the queue of *list*.
- *skb_append(oldskb, newskb, list):* inserts *newskb* <u>behind</u> *oldskb* in the queue of *list.*

MABEL Technology
Tech The Best

Managing Socket Buffer Queues

- *skb_unlink(skb, list):* removes the socket buffer *skb* from queue *list* and decrement the queue length.
- *skb_peek(list):* returns a pointer to the first element of a list, if this list is not empty; otherwise, returns *NULL*.
  - Leaves buffer on the list
- *skb_peek_tail(list):* returns a pointer to the last element of a queue; if the list is empty, returns *NULL.*
  - Leaves buffer on the list

sk_buff Alignment

- CPUs often take a performance hit when accessing unaligned memory locations.
- Since an Ethernet header is 14 bytes, network drivers often end up with the IP header at an unaligned offset.
- The IP header can be aligned by shifting the start of the packet by 2 bytes. Drivers should do this with:
  - skb_reserve(NET_IP_ALIGN);
- The downside is that the DMA is now unaligned. On some architectures the cost of an unaligned DMA outweighs the gains so NET_IP_ALIGN is set on a per arch basis.

MABEL Technology Tech The Best

Network Devices

- An interface between software-based protocols and network adapters (hardware).
- Two major functions:
  - Abstract from the technical properties of network adapters (that implement different layer-1 and layer-2 protocols and are manufactured by different vendors).
  - Provide a uniform interface for access by protocol instances.
- Represented in Linux by a **`struct net_device`**
  - **`include/linux/netdevice.h`**

Struct net_device_ops

- The methods of a network interface. The most important ones:
  - ndo_init(), called once when the device is registered
  - ndo_open(), called when the network interface is up'ed
  - ndo_close(), called when the network interface is down'ed
  - ndo_start_xmit(), to start the transmission of a packet
  - ndo_tx_timeout(), callback for when tx doesn’t progress in time
- And others:
  - ndo_get_stats(), to get statistics
  - ndo_do_ioctl(), to implement device specific operations
  - ndo_set_rx_mode(), to select promiscuous, multicast, etc.
  - ndo_set_mac_address(), to set the MAC address
  - ndo_set_multicast_list(), to set multicast filters
- The netdev_ops field in the struct net_device structure must be set to point to the struct net_device_ops structure.

net_device members

- ***char name[IFNAMSIZ]*** - name of the network device, e.g., *eth0-eth4, lo* (loopback device)
- ***unsigned int mtu*** – Maximum Transmission Unit: the maximum size of frame the device can handle.
- ***unsigned int irq*** – irq number.
- ***unsigned char *dev_addr*** : hw MAC address.
- ***int promiscuity –*** a counter of the times a NIC is told to set to work in promiscuous mode; used to enable more than one sniffing client.
- ***struct net_device_stats stats*** – statistics
- ***struct net_device_ops *netdev_ops*** – netdev ops

MABEL Technology
Tech The Best

net_device->flags

- flags: properties of the network device
  - *IFF_UP:* the device is on.
  - *IFF_BROADCAST:* the device is broadcast-enabled.
  - *IFF_DEBUG:* debugging is turned on.
  - *IFF_LOOKBACK:* the device is a loopback network device.
  - *IFF_POINTTOPOINT:* this is a point-to-point connection.
  - *IFF_PROMISC:* this flag switches the promiscuous mode on.
  - *IFF_MULTICAST*: activates the receipt of multicast packets.
  - *IFF_NOARP*: doesn’t support ARP

net_device->features

- features: features of the network device
  - *NETIF_F_SG:* supports scatter-gather.
  - *NETIF_F_IP_CSUM:* supports TCP/IP checksum offload.
  - *NETIF_F_NO_CSUM:* checksum not needed (loopback).
  - *NETIF_F_HW_CSUM:* supports all checksums.
  - *NETIF_F_FRAGLIST:* supports scatter-gather.
  - *NETIF_F_HW_VLAN_TX:* hardware support for VLANs.
  - *NETIF_F_HW_VLAN_RX:* hardware support for VLANs.
  - *NETIF_F_GSO:* generic segmentation offload
  - *NETIF_F_GRO:* generic receive offload.
  - *NETIF_F_LRO:* large receive offload.

net_device allocation

- Allocated using:
  - struct net_device ***alloc_netdev**(size, mask, setup_func);
    - size – size of our private data part
    - mask – a naming pattern (e.g. "eth%d")
    - setup_func – A function to prepare the rest of the net_device.
- And deallocated with
  - void **free_netdev**(struct *net_device);
- For Ethernet we have a specialized version:
  - struct net_device ***alloc_etherdev**(size);
    - which calls alloc_netdev(size, "eth%d", ether_setup);

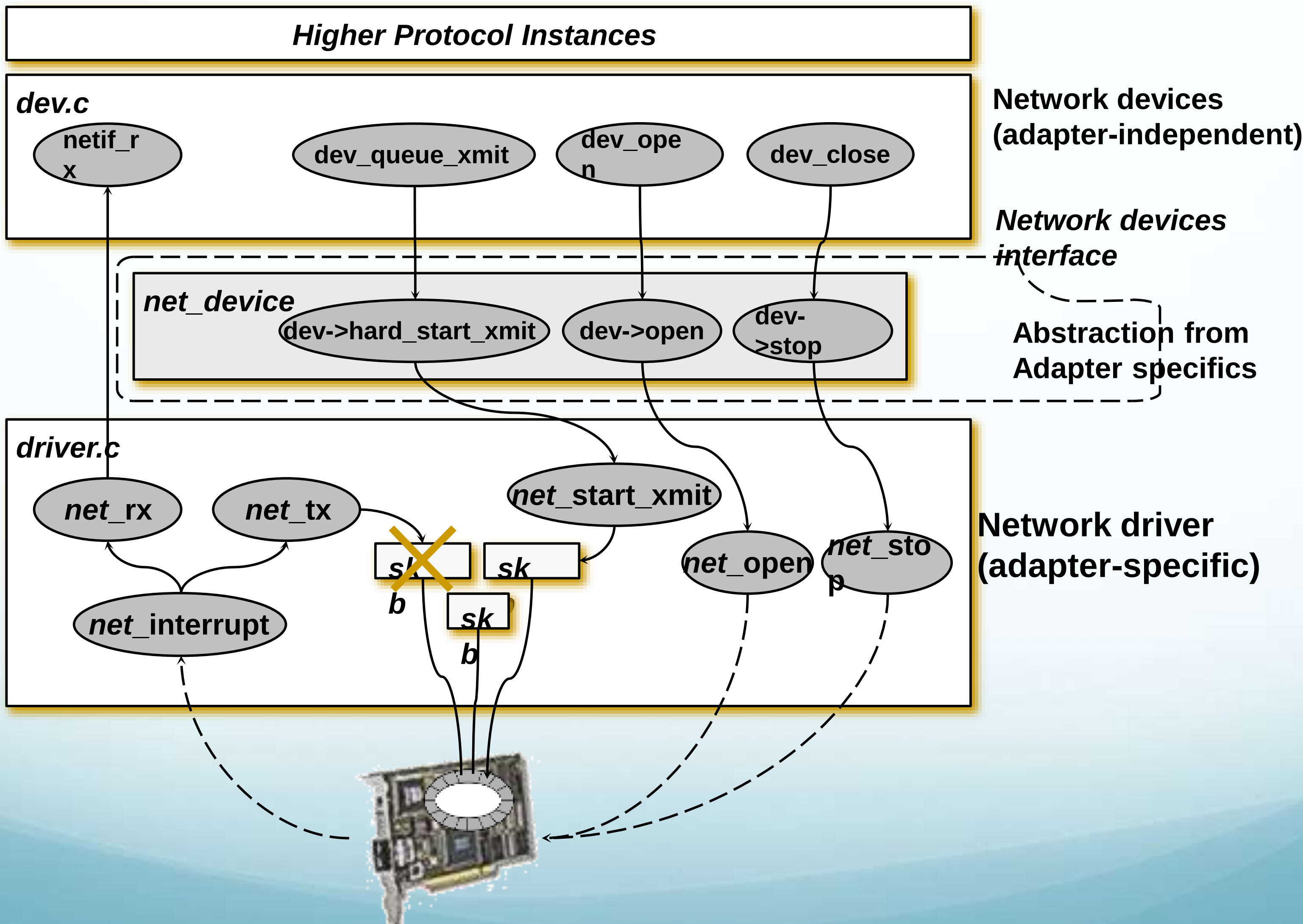
net_device registration

- Registered via:
  - int **register_netdev**(struct net_device *dev);
  - **int unregister_netdev**(struct net_device dev);

Utility Functions

- netif_start_queue()
  - Tells the kernel that the driver is ready to send packets
- netif_stop_queue()
  - Tells the kernel to stop sending packets. Useful at driver cleanup of course, but also when all transmission buffers are full.
- netif_queue_stopped()
  - Tells whether the queue is currently stopped or not
- netif_wake_queue()
  - Wakeup a queue after a netif_stop_queue(). The kernel will resume sending packets

Network Device Interface



MABEL Technology
Tech The Best

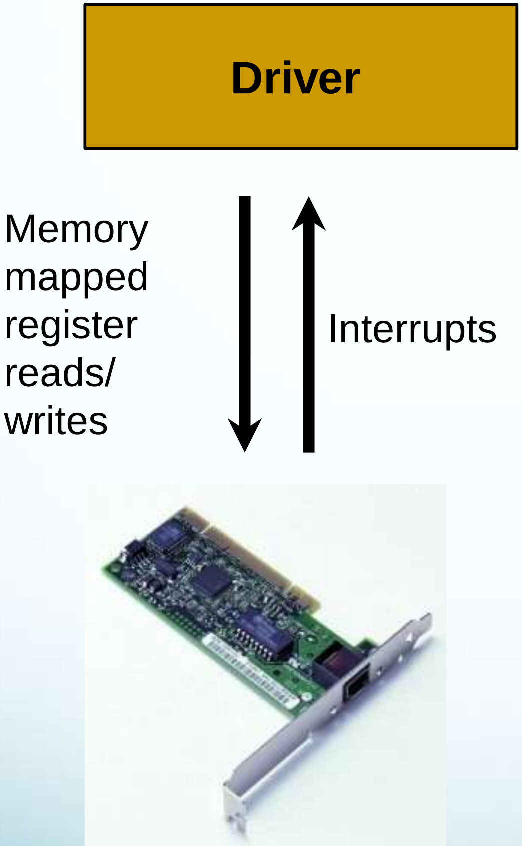
Device Layer vs. Device Driver

- Linux tries to abstract away the device specifics using the **`struct net_device`**
- Provides a generic device layer in
  - **`linux/net/core/dev.c`** and
  - **`include/linux/netdevice.h`**
- Device drivers are responsible for providing the appropriate virtual functions
  - E.g., **`dev->netdev_ops->ndo_start_xmit`**
- Device layer calls driver layer and vice-versa
- Execution spans interrupts, syscalls, and softirqs

Network Process Contexts

- Hardware interrupt
  - Received packets (upcalls)
- Process context
  - System calls (downcalls)
- Softirq context
  - NET_RX_SOFTIRQ for received packets (upcalls)
  - NET_TX_SOFTIRQ for delayed sending packets (downcalls)

Device Driver HW Interface



- Driver talks to the device:
  - Writing commands to memory-mapped *control status registers*
  - Setting aside buffers for packet transmission/reception
  - Describing these buffers in *descriptor rings*
- Device talks to driver:
  - Generating *interrupts* (both on send and receive)
  - Placing values in control status registers
  - DMA’ing packets to/from available buffers
  - Updating status in descriptor rings

NIC IRQ

- The NIC registers an interrupt handler with the IRQ with which the device works by calling **`request_irq()`**.
  - This interrupt handler is the one that will be called when a frame is received
  - The same interrupt handler *may* be called for other reasons (depends, NIC-dependent)
    - Transmission complete, transmission error
  - Newer drivers (e.g., e1000e) seem to use Message Sequenced Interrupts (MSI), which use different interrupt numbers
- Device drivers can release an IRQ using **`free_irq`**.

MABEL Technology Tech The Best

Packet Reception with NAPI

- Originally, Linux took one interrupt per received packet
  - This could cause excessive overhead under heavy loads
- NAPI: "New API"
- With NAPI, interrupt notifies softnet layer (NET_RX_SOFTIRQ) that packets are available
- Driver requirements:
  - Ability to turn receive interrupts off and back on again
  - A ring buffer
  - A poll function to pull packets out
- Most drivers support this now.

MABEL Technology
Tech The Best

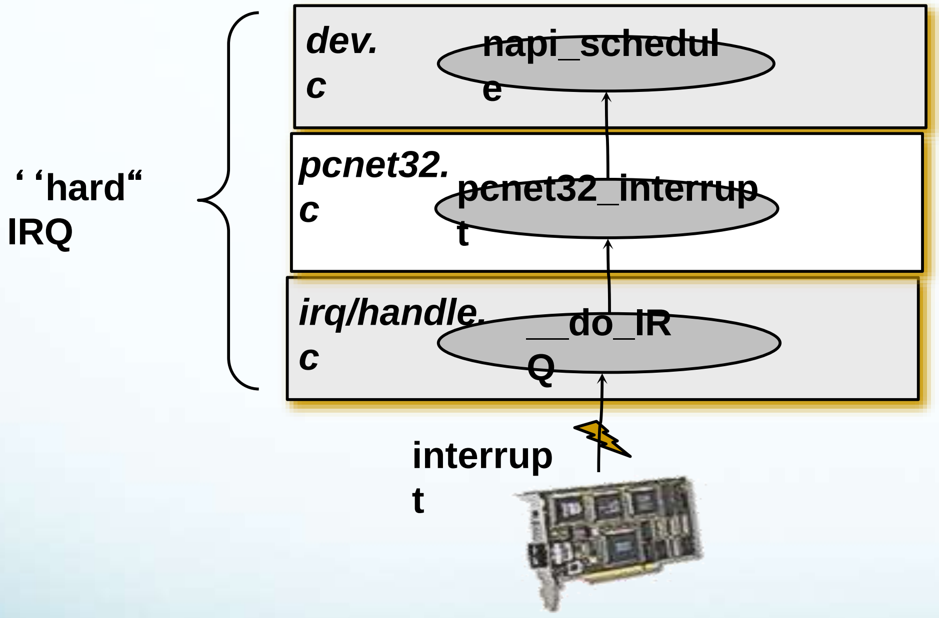
Reception: NAPI mode (1)

- NAPI allows dynamic switching:
  - To polled mode when the interrupt rate is too high
  - To interrupt-driven when load is low
- In the network interface private structure, add a **`struct napi_struct`**
- At driver initialization, register the NAPI poll operation:
  `netif_napi_add(dev, &bp->napi, my_poll, 64);`
  - **`dev`** is the network interface
  - **`&bp->napi`** is the struct napi_struct
  - **`my_poll`** is the NAPI poll operation
  - **`64`** is the *weight* that represents the importance of the network interface. It is related to the threshold below which the driver will return back to interrupt mode.

Reception: NAPI mode (2)

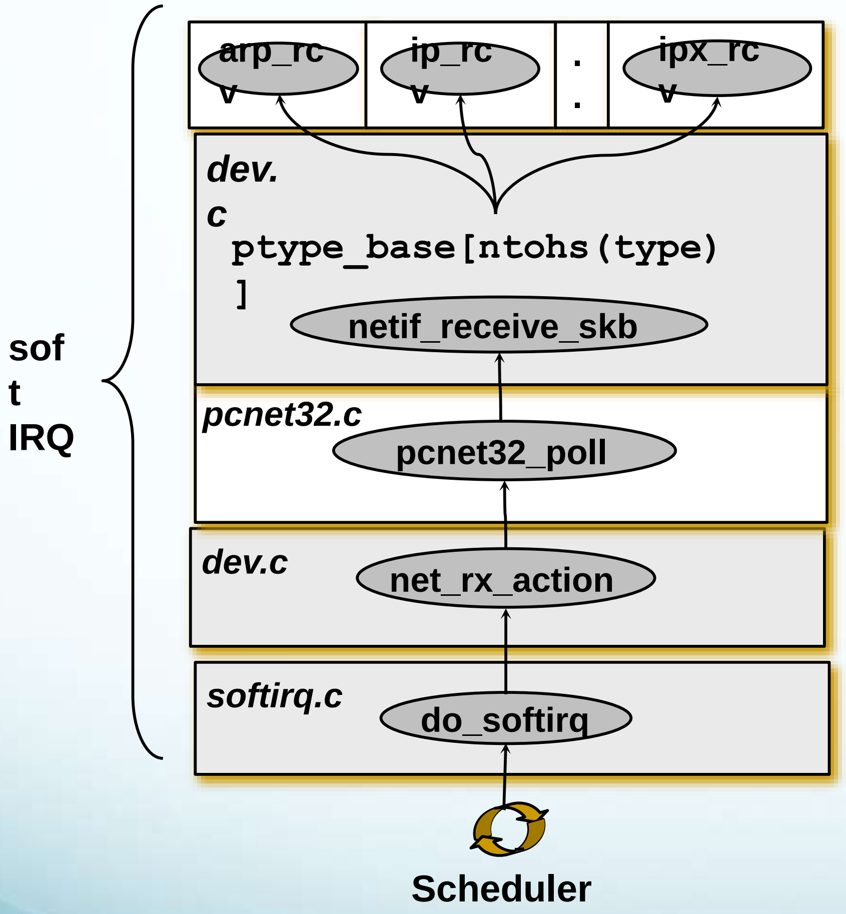
- In the interrupt handler, when a packet has been received:
  - `if (napi_schedule_prep(&bp->napi)) {`
  - `/* Disable reception interrupts */`
  - `__napi_schedule(& bp->napi);`
  - `}`
  - The kernel will call our **`poll()`** operation regularly
- The **`poll()`** operation has the following prototype:
  - **`static int my_poll(struct napi_struct *napi, int budget)`**
- It must receive at most budget packets and push them to the network stack using **`netif_receive_skb()`**.
- If fewer than budget packets have been received, switch back to interrupt mode using **`napi_complete(& bp->napi)`** and reenable interrupts
- Poll function must return the number of packets received

Receiving Data Packets (1)



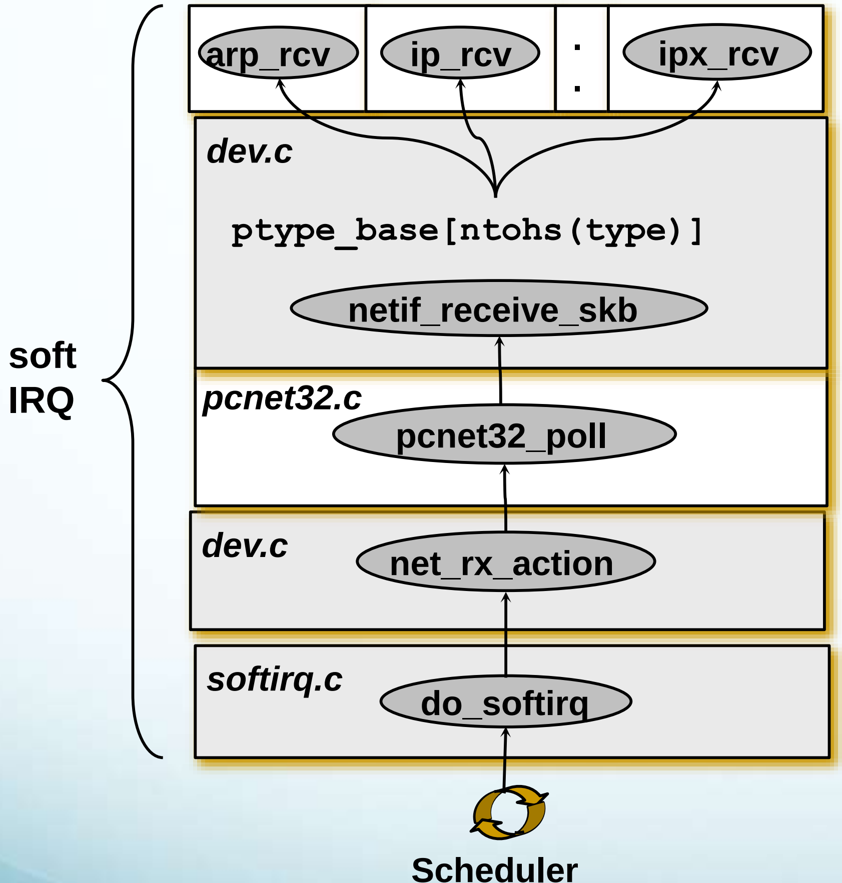
- HW interrupt invokes **`__do_IRQ`**
- **`__do_IRQ`** invokes each handler for that IRQ:
  - **`action->handler(irq, action->dev_id);`**
- **`pcnet_32_interrupt`**
  - Acknowledge intr ASAP
  - Checks various registers
  - Calls **`napi_schedule`** to wake up NET_RX_SOFTIRQ

Receiving Data Packets (2)



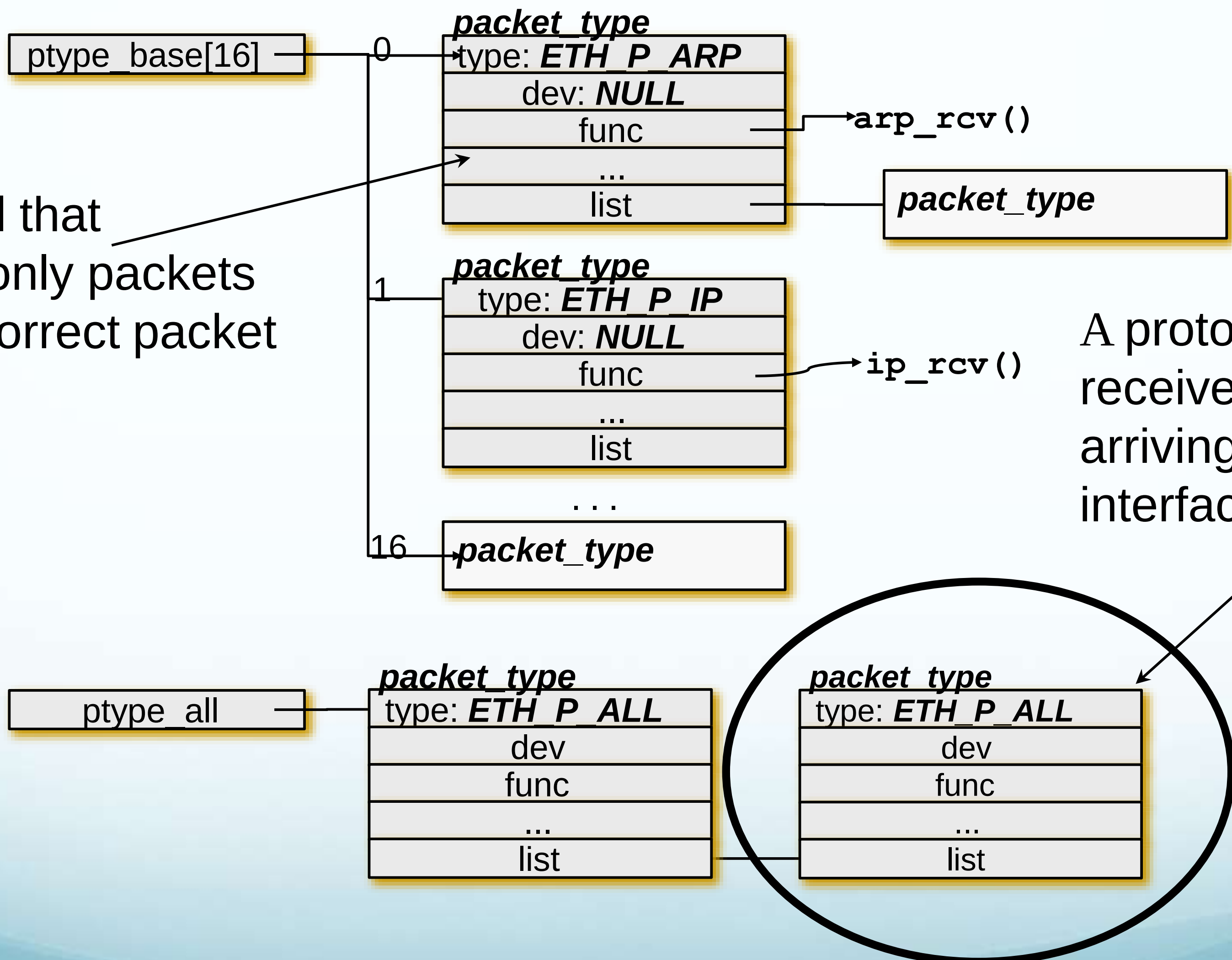
- Immediately after the interrupt, **`do_softirq`** is run
  - Recall softirqs are per-cpu
- For each napi struct in the list (one per dev)
  - Invoke **`poll`** function
  - Track amount of work done (packets)
  - If work threshold exceeded, wake up softirqd and break out of loop

Receiving Data Packets (3)



- Driver poll function:
  - *may* call **`dev_alloc_skb`** and copy
    - pcnet32 does, e1000 doesn't.
  - *Does* call **`netif_receive_skb`**
  - Clears tx ring and frees sent skbs
- **`netif_receive_skb:`**
  - Calls **`eth_type_trans`** to get packet type
    - **`skb_pull`** the ethernet header (14 bytes)
    - Data now points to payload data (e.g., IP header)
  - Demultiplexes to appropriate receive function based on header type

Packet Types Hash Table



A protocol that receives only packets with the correct packet identifier

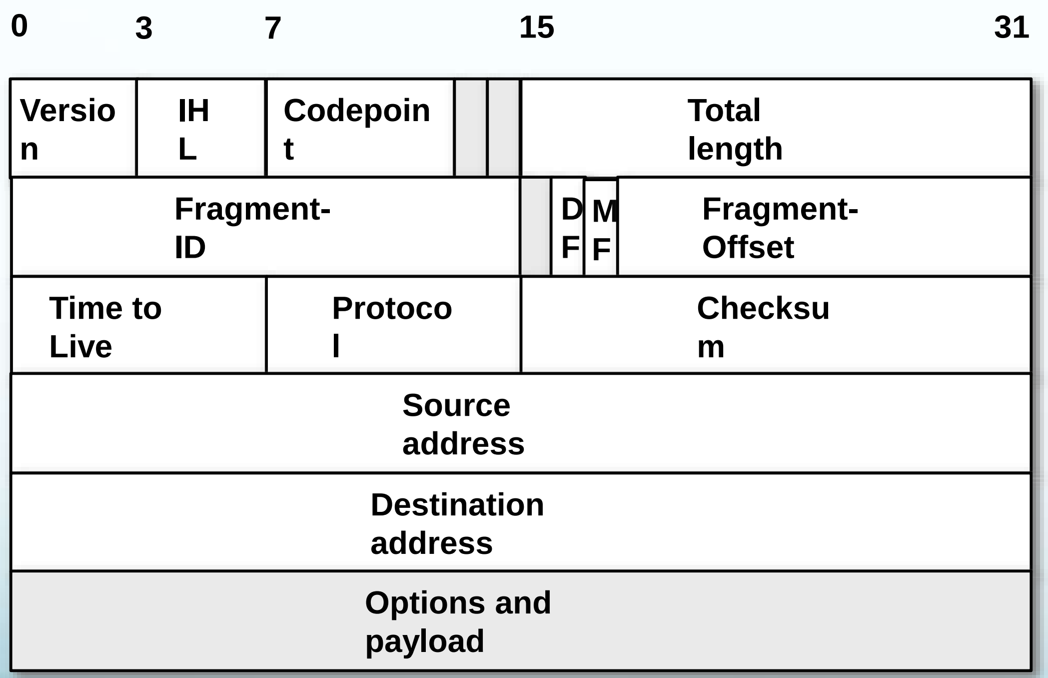
A protocol that receives all packets arriving at the interface

Queuing Ops

- enqueue()
  - Enqueues a packet
- dequeue()
  - Returns a pointer to a packet (skb) eligible for sending; NULL means nothing is ready
- pfifo – 3 band priority fifo
  - Enqueue function is **`pfifo_fast_enqueue`**
  - Dequeue function is **`pfifo_fast_dequeue`**

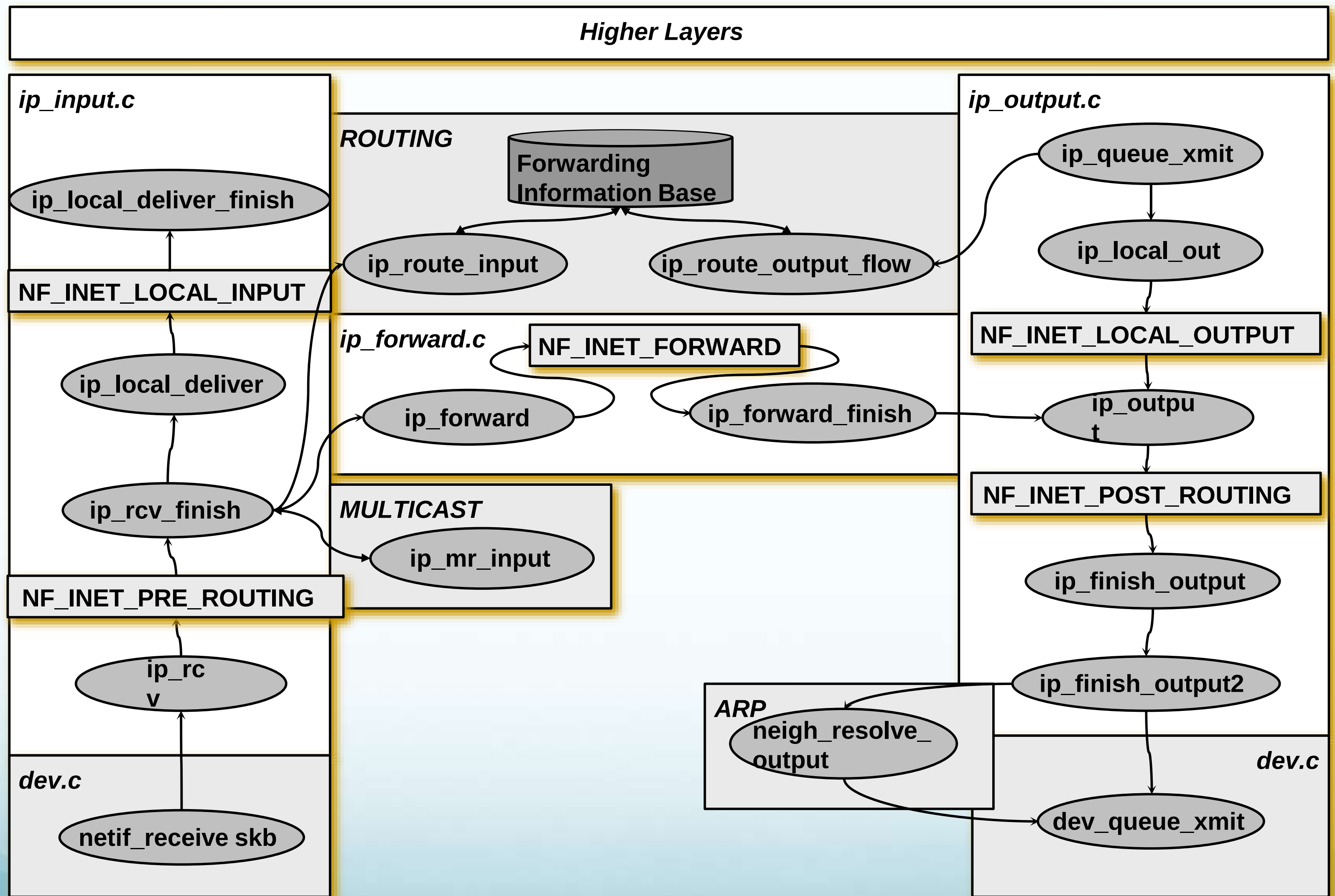
IP

IP-packet format



- Encapsulate/ decapsulate transport-layer messages into IP datagrams
- Routes datagrams to destination
- Handle static and/or dynamic routing updates
- Fragment/ reassemble datagrams
- Unreliably

IP Implementation Architecture



Sources of IP Packets

1. Packets arrive on an interface and are passed to the *ip_rcv()* function.
2. TCP/UDP packets are packed into an IP packet and passed down to IP via *ip_queue_xmit().*
3. The IP layer generates IP packets itself:
    1. Multicast packets
    2. Fragmentation of a large packet
    3. ICMP/IGMP packets.

What is Netfilter?

- A framework for packet “mangling”
- A protocol defines "hooks" which are well-defined points in a packet's traversal of that protocol stack.
  - IPv4 defines 5
  - Other protocols include IPv6, ARP, Bridging, DECNET
- At each of these points, the protocol will call the netfilter framework with the packet and the hook number.
- Parts of the kernel can register to listen to the different hooks for each protocol.
- When a packet is passed to the netfilter framework, it will call all registered callbacks for that hook and protocol.

MABEL Technology
Tech The Best

Netfilter IPv4 Hooks

- NF_INET_PRE_ROUTING
  - Incoming packets pass this hook in ip_rcv() before routing
- NF_INET_LOCAL_IN
  - All incoming packets addressed to the local host pass this hook in ip_local_deliver()
- NF_INET_FORWARD
  - All incoming packets not addressed to the local host pass this hook in ip_forward()
- NF_INET_LOCAL_OUT
  - All outgoing packets created by this local computer pass this hook in ip_build_and_send_pkt()
- NF_INET_POST_ROUTING
  - All outgoing packets (forwarded or locally created) will pass this hook in ip_finish_output()

Netfilter Callbacks

- Kernel code can register a call back function to be called when a packet arrives at each hook. and are free to manipulate the packet.
- The callback can then tell netfilter to do one of five things:
  - NF_DROP: drop the packet; don't continue traversal.
  - NF_ACCEPT: continue traversal as normal.
  - NF_STOLEN: I've taken over the packet; stop traversal.
  - NF_QUEUE: queue the packet (usually for userspace handling).
  - NF_REPEAT: call this hook again.

IPTables

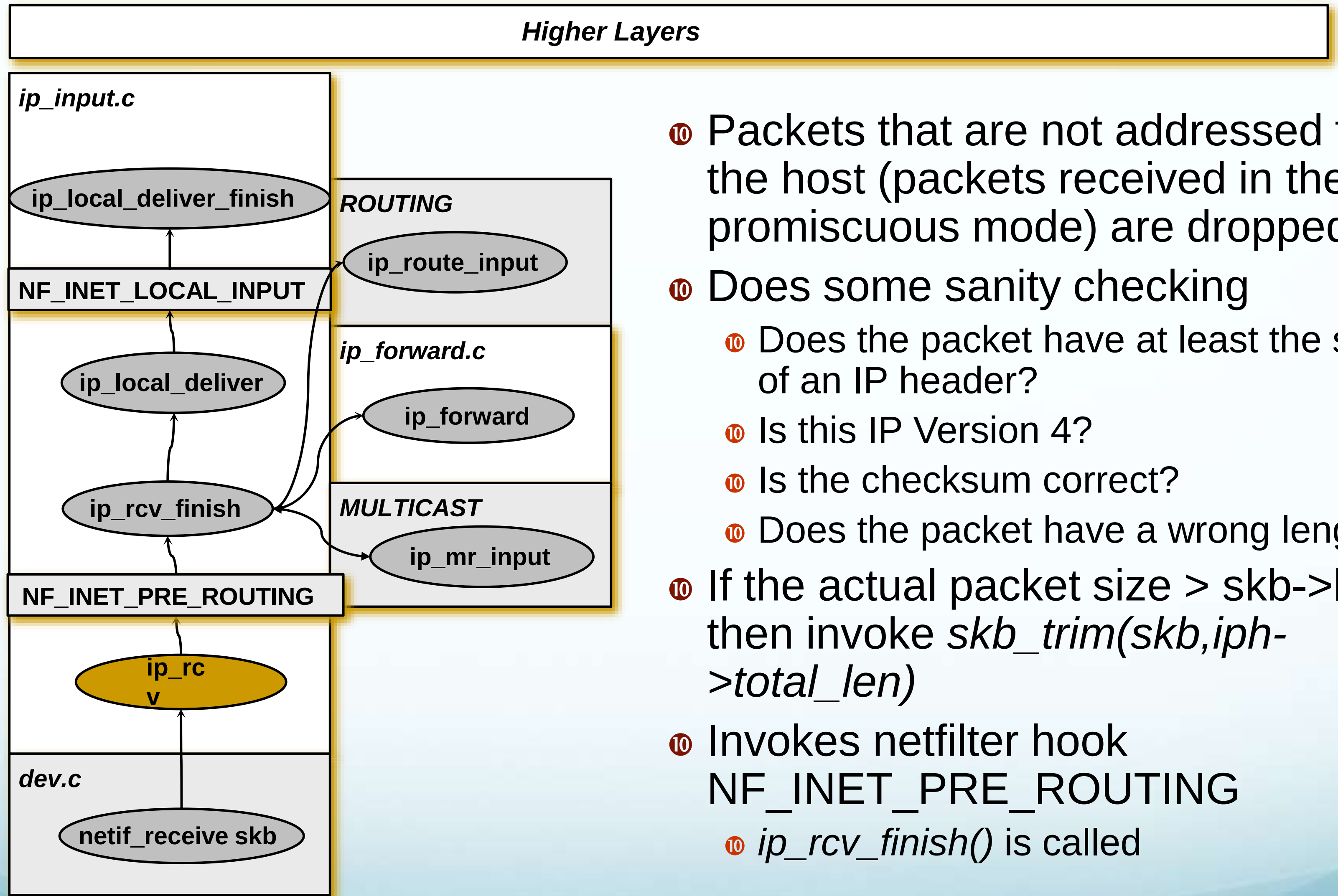
- A packet selection system called IP Tables has been built over the netfilter framework.
- It is a direct descendant of ipchains (that came from ipfwadm, that came from BSD's ipfw), with extensibility.
- Kernel modules can register a new table, and ask for a packet to traverse a given table.
- This packet selection method is used for:
  - Packet filtering (the `filter' table),
  - Network Address Translation (the `nat' table) and
  - General preroute packet mangling (the `mangle' table).

MABEL Technology
Tech The Best

Naming Conventions

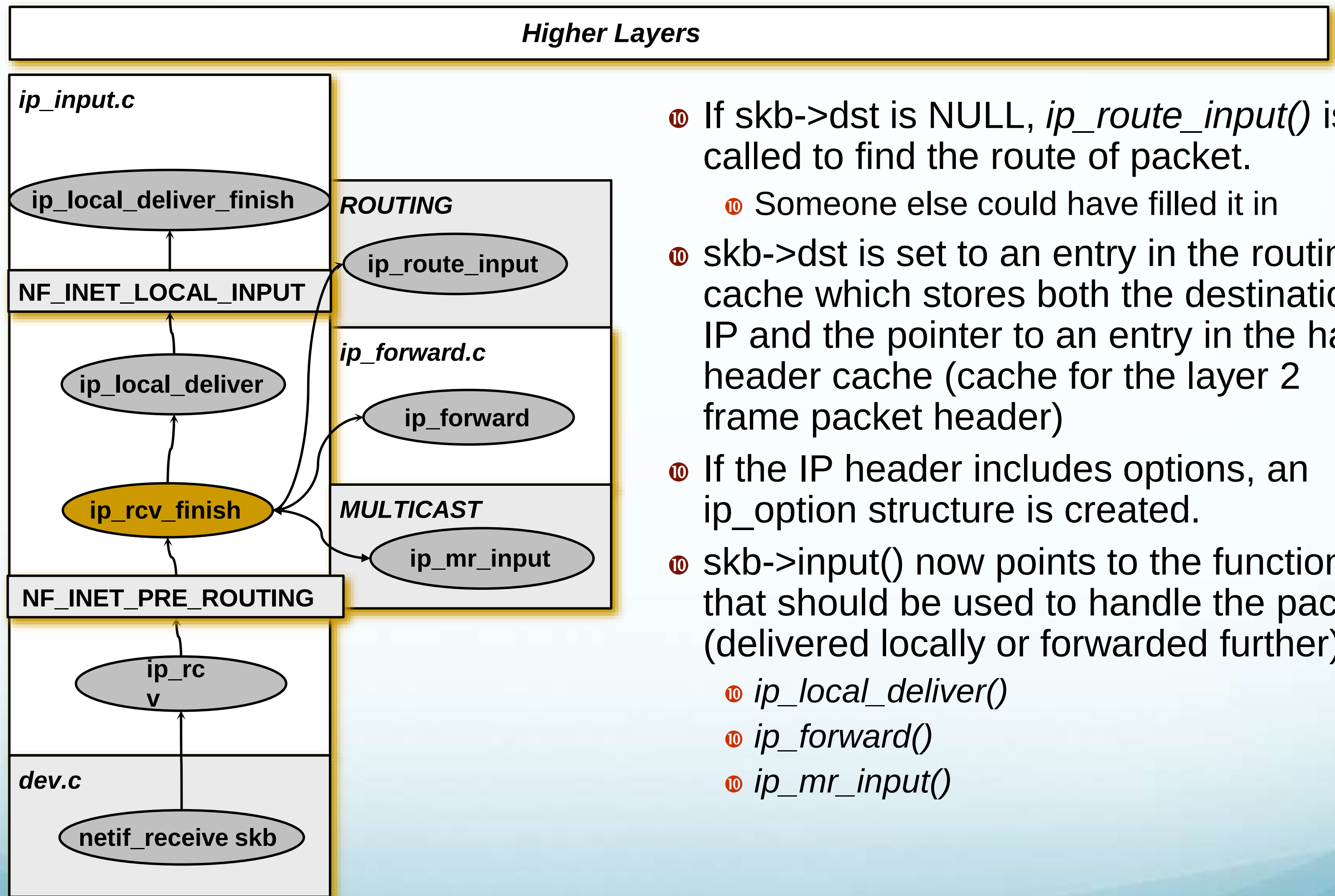
- Methods are frequently broken into two stages (where the second has the same name with a suffix of *finish* or *slow*, is typical for networking kernel code.)
  - E.g., ip_rcv, ip_rcv_finish
- In many cases the second method has a “slow” suffix instead of “finish”; this usually happens when the first method looks in some cache and the second method performs a lookup in a more complex data structure, which is slower.

Receive Path: ip_rcv



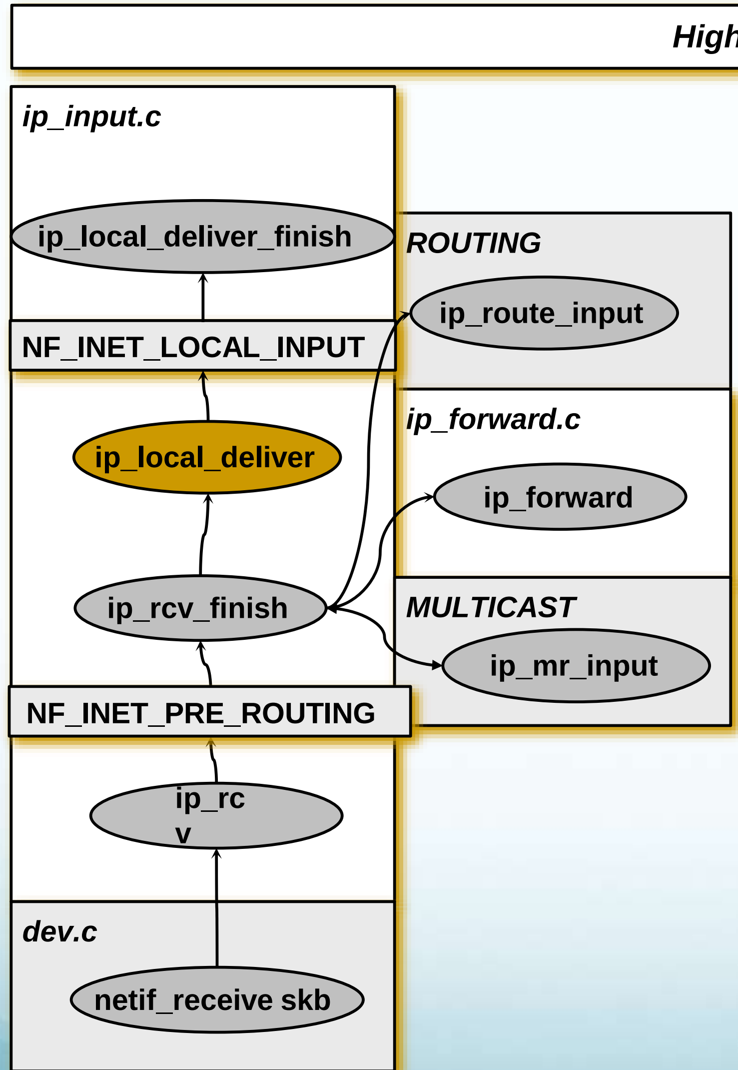
- Packets that are not addressed to the host (packets received in the promiscuous mode) are dropped.
- Does some sanity checking
  - Does the packet have at least the size of an IP header?
  - Is this IP Version 4?
  - Is the checksum correct?
  - Does the packet have a wrong length?
- If the actual packet size > skb->len, then invoke *skb_trim(skb,iph->total_len)*
- Invokes netfilter hook NF_INET_PRE_ROUTING
  - *ip_rcv_finish()* is called

Receive Path: ip_rcv_finish



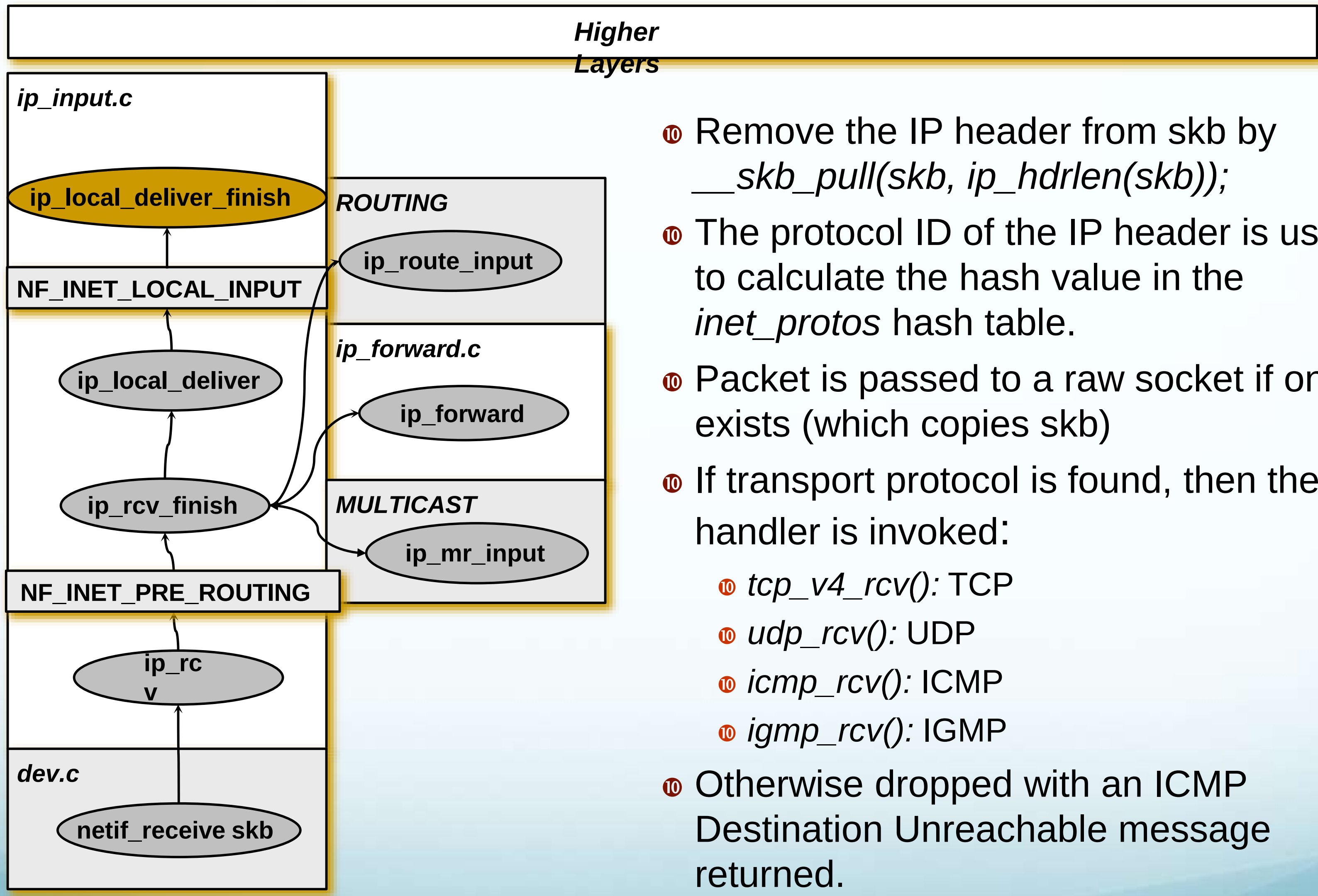
- If skb->dst is NULL, *ip_route_input()* is called to find the route of packet.
  - Someone else could have filled it in
- skb->dst is set to an entry in the routing cache which stores both the destination IP and the pointer to an entry in the hard header cache (cache for the layer 2 frame packet header)
- If the IP header includes options, an ip_option structure is created.
- skb->input() now points to the function that should be used to handle the packet (delivered locally or forwarded further):
  - *ip_local_deliver()*
  - *ip_forward()*
  - *ip_mr_input()*

Receive Path: ip_local_deliver



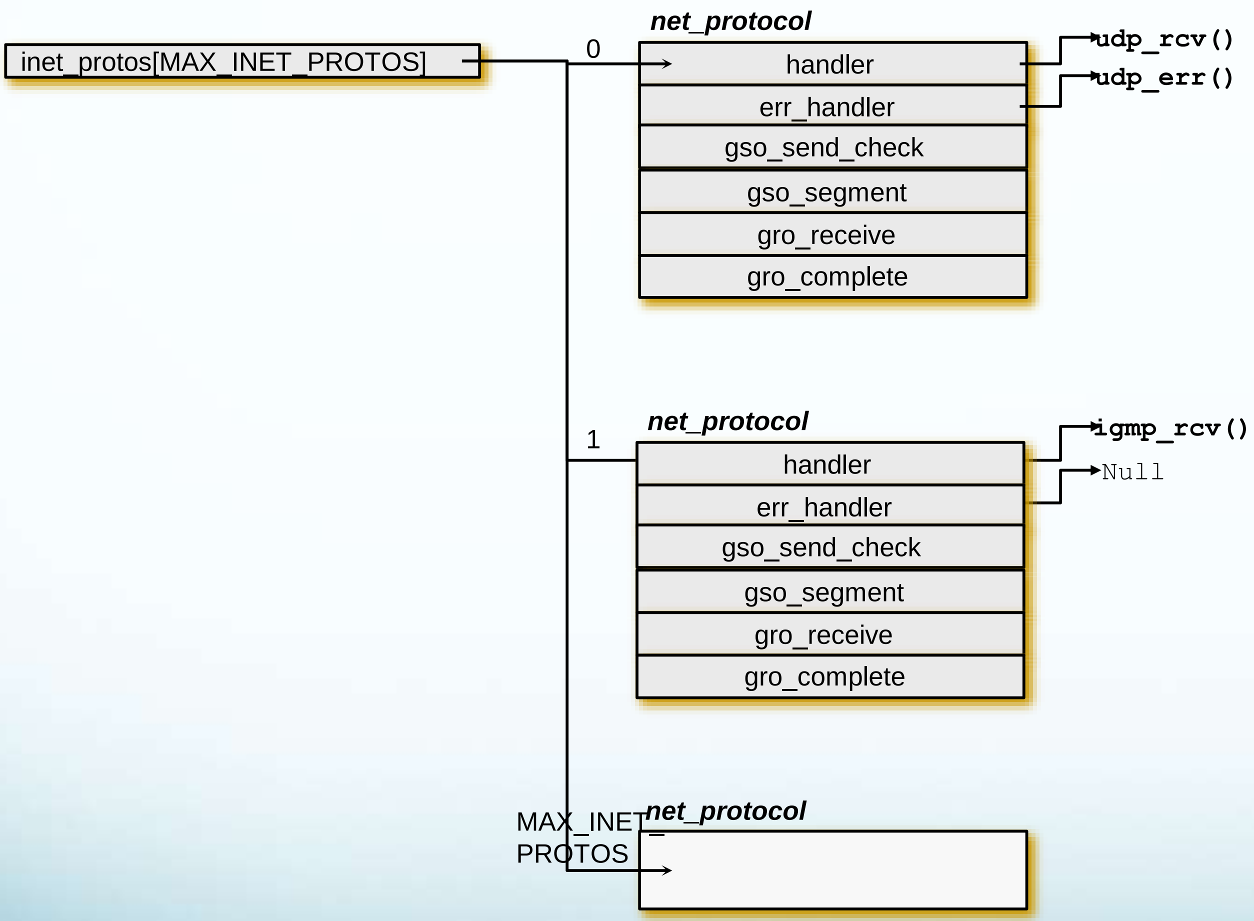
- The only task of *ip_local_deliver(skb)* is to re-assemble fragmented packets by invoking *ip_defrag().*
- The netfilter hook NF_INET_LOCAL_IN is invoked.
- This in turn calls ip_local_deliver_finish

Recv: ip_local_deliver_finish



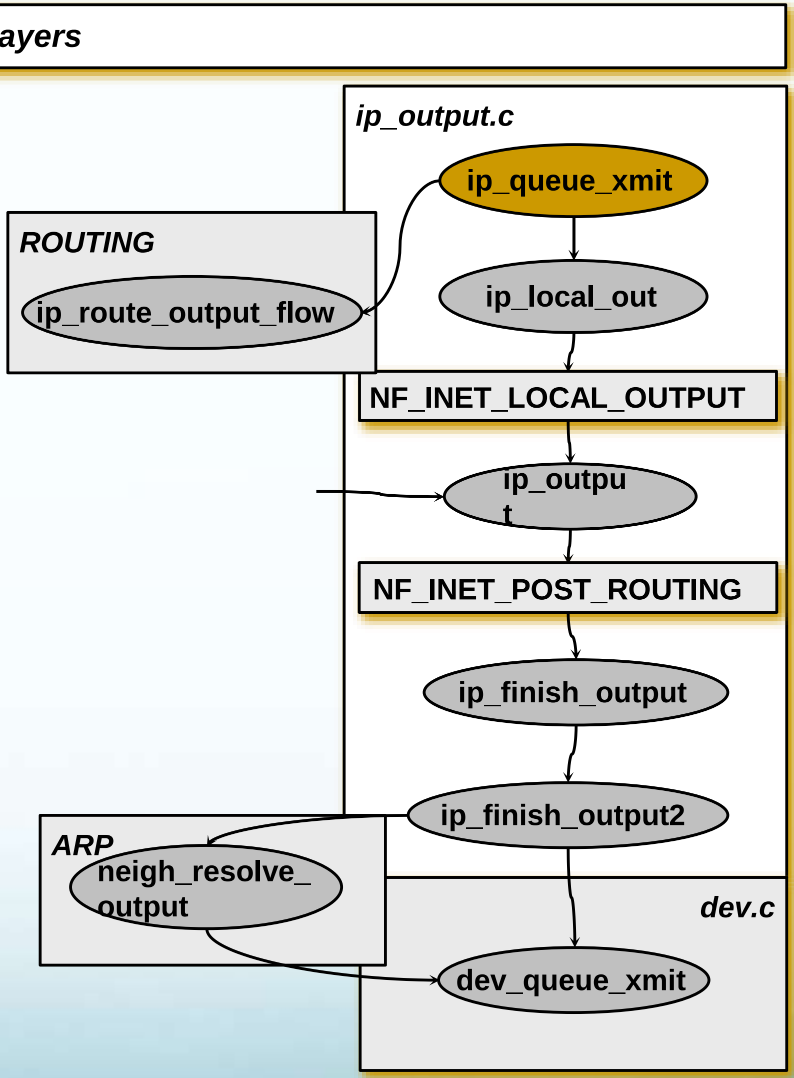
- Remove the IP header from skb by *__skb_pull(skb, ip_hdrlen(skb));*
- The protocol ID of the IP header is used to calculate the hash value in the *inet_protos* hash table.
- Packet is passed to a raw socket if one exists (which copies skb)
- If transport protocol is found, then the handler is invoked:
  - *tcp_v4_rcv():* TCP
  - *udp_rcv():* UDP
  - *icmp_rcv():* ICMP
  - *igmp_rcv():* IGMP
- Otherwise dropped with an ICMP Destination Unreachable message returned.

Hash Table *inet_protos*



Send Path: ip_queue_xmit (1)

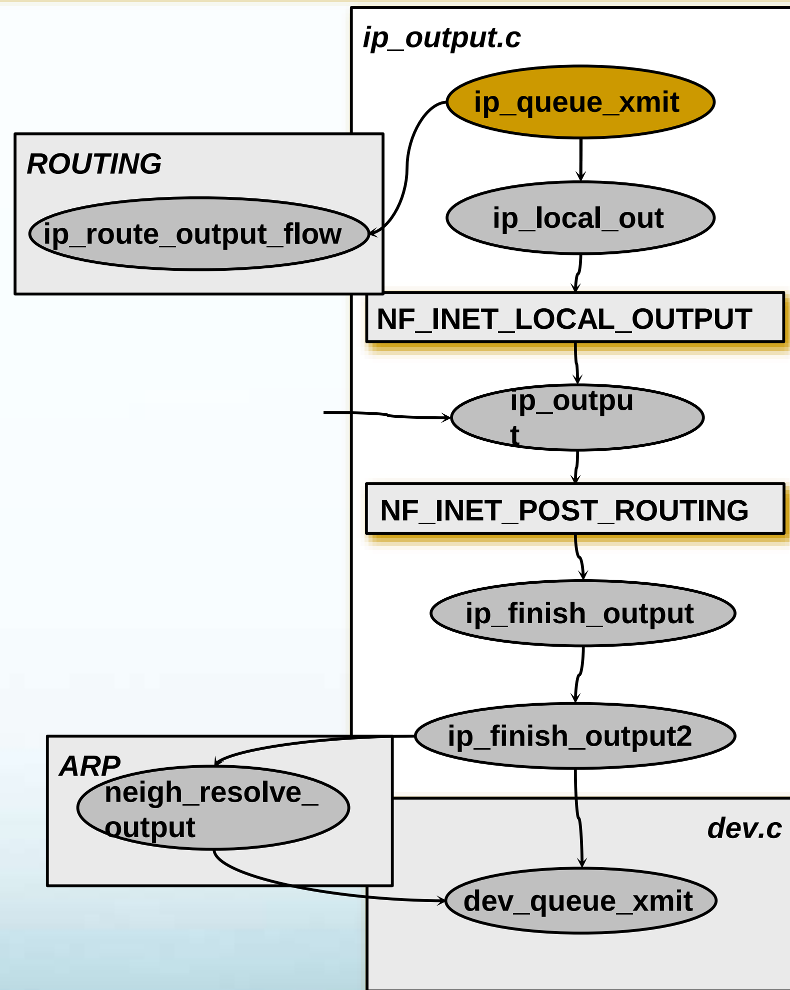
- *skb->dst* is checked to see if it contains a pointer to an entry in the routing cache.
  - Many packets are routed through the same path, so storing a pointer to an routing entry in skb->dst saves expensive routing table lookup.
- If route is not present (e.g., the first packet of a socket), then *ip_route_output_flow()* is invoked to determine a route.



Send Path: ip_queue_xmit (2)

Higher Layers

- Header is pushed onto packet
  - *skb_push(skb, sizeof(header + options);*
- The fields of the IP header are filled in (version, header length, TOS, TTL, addresses and protocol).
- If IP options exist, *ip_options_build()* is called.
- *Ip_local_out()* is invoked.

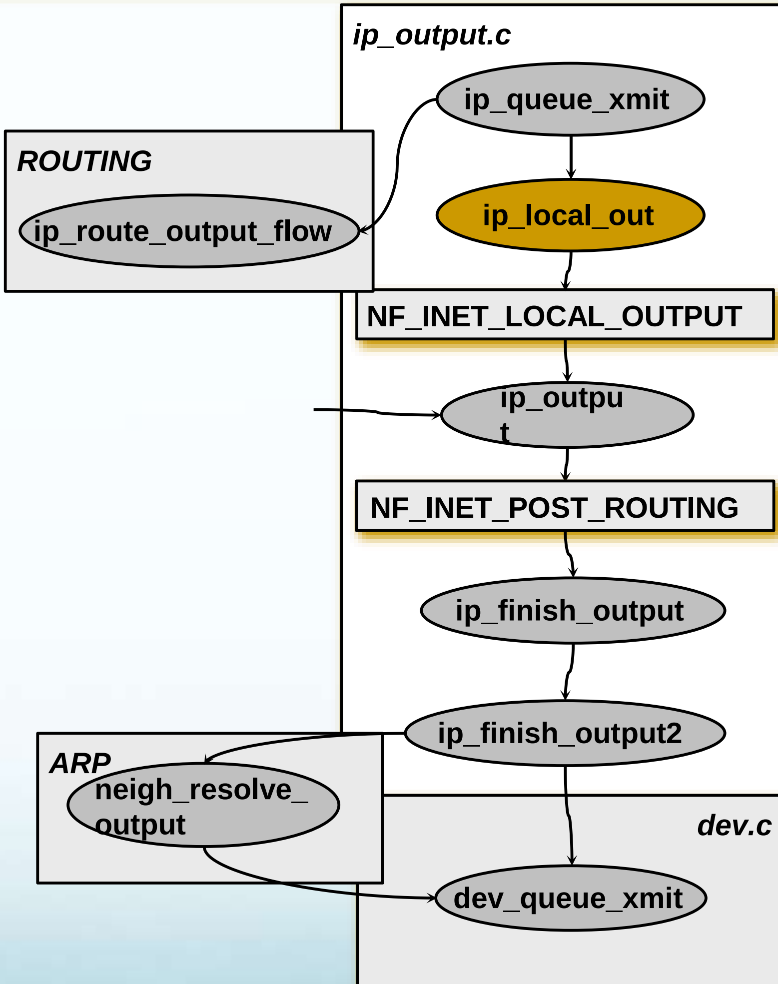


MABEL Technology
Tech The Best

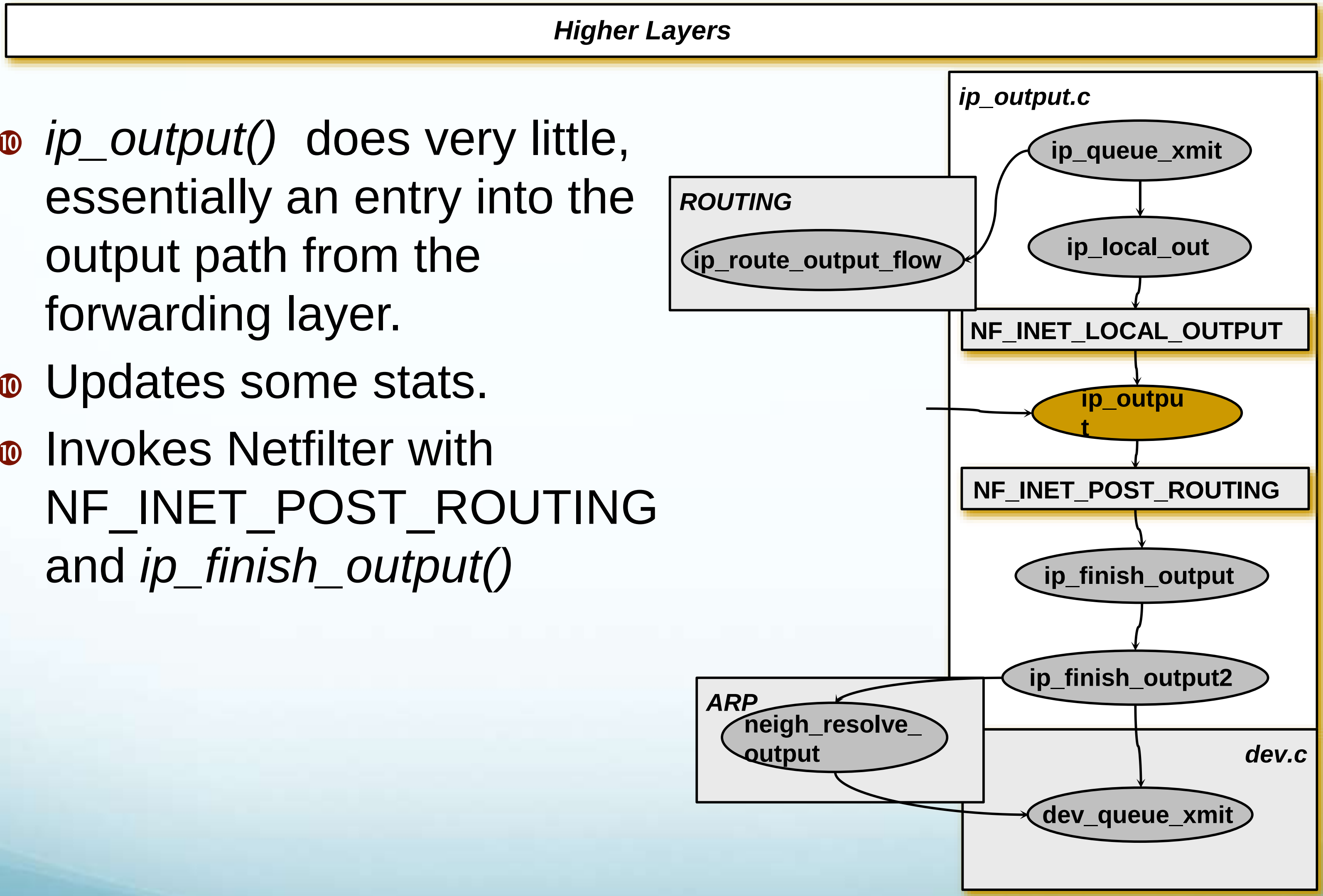
Send Path: ip_local_out

Higher Layers

- The checksum is computed
  - *ip_send_check(iph)*
- Netfilter is invoked with NF_INET_LOCAL_OUTPUT using *skb->dst_output()*
  - This is *ip_output()*
- If the packet is for the local machine:
  - *dst->output = ip_output*
  - *dst->input = ip_local_deliver*
  - *ip_output()* will send the packet on the loopback device
  - Then we will go into *ip_rcv()* and *ip_rcv_finish()* , but this time dst is NOT null; so we will end in *ip_local_deliver()* .



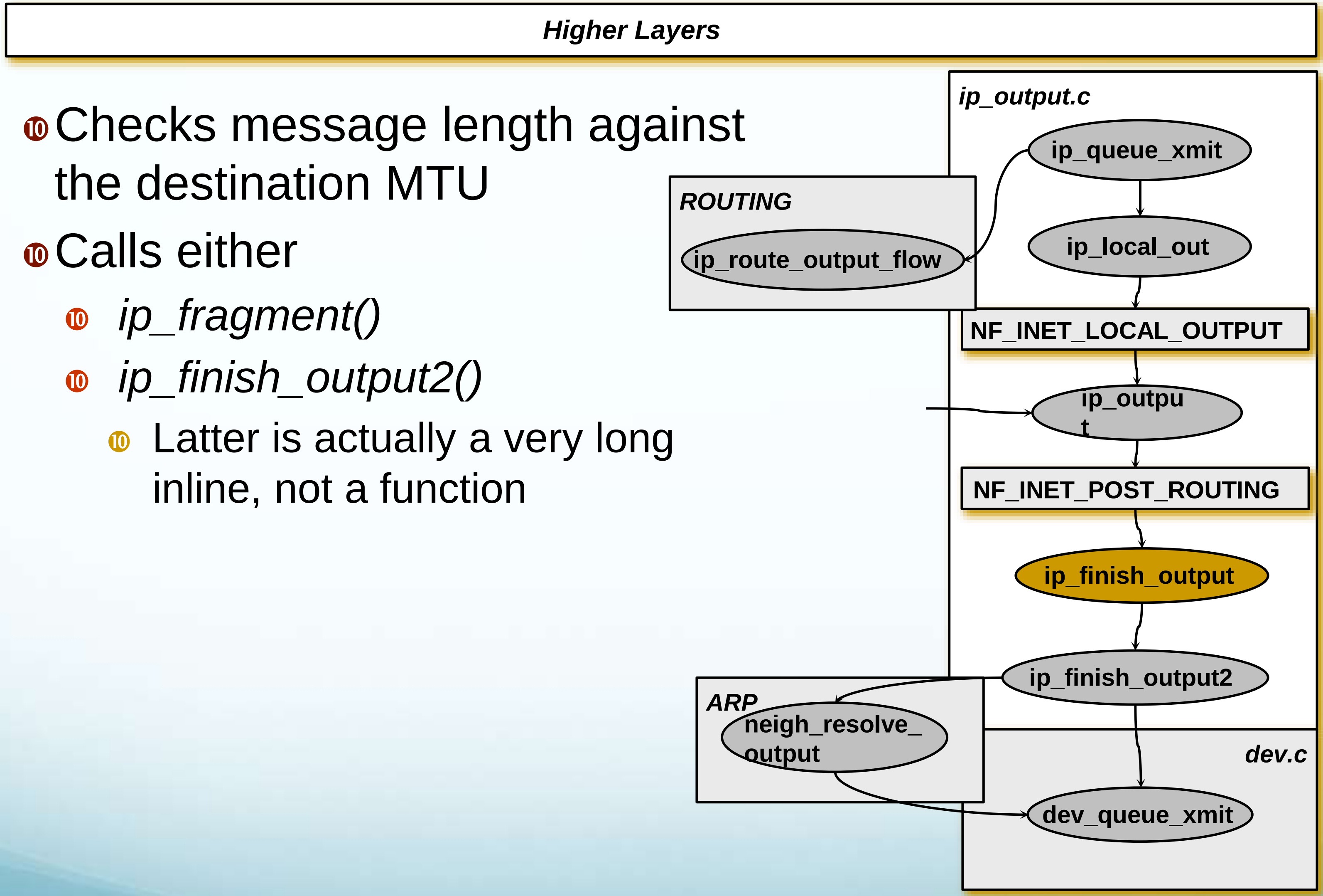
Send Path: ip_output



- *ip_output()* does very little, essentially an entry into the output path from the forwarding layer.
- Updates some stats.
- Invokes Netfilter with NF_INET_POST_ROUTING and *ip_finish_output()*

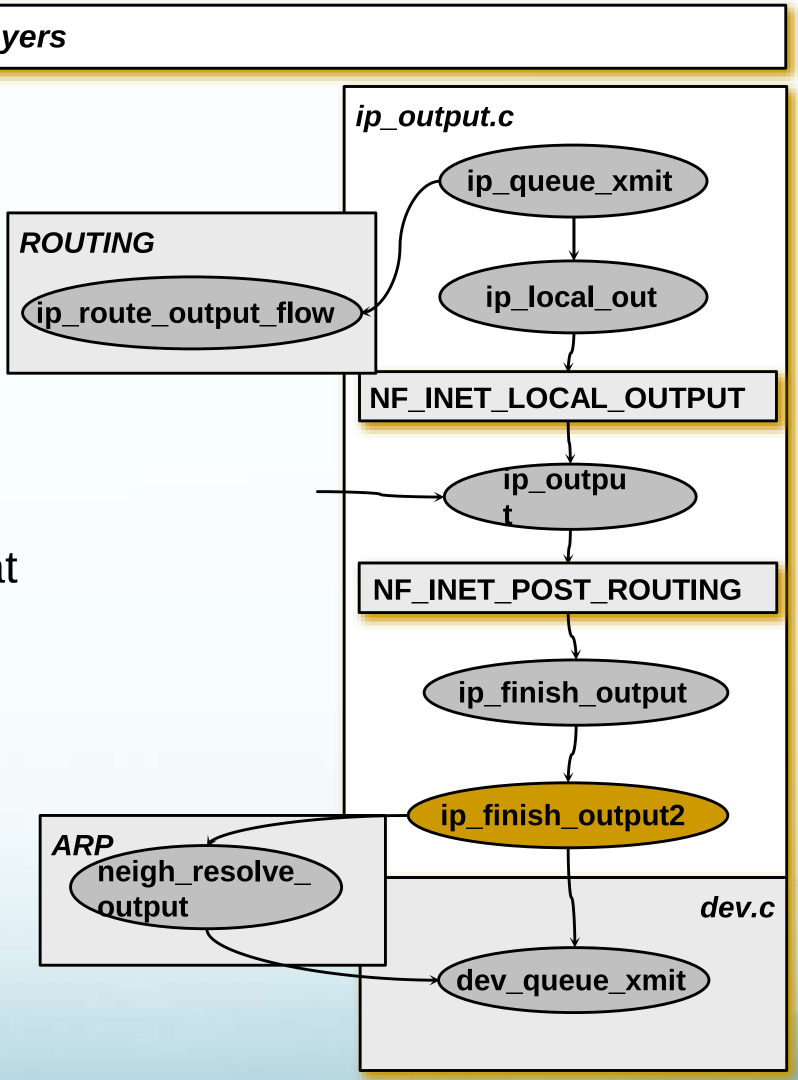
Send Path: ip_finish_output

- Checks message length against the destination MTU
- Calls either
  - *ip_fragment()*
  - *ip_finish_output2()*
    - Latter is actually a very long inline, not a function

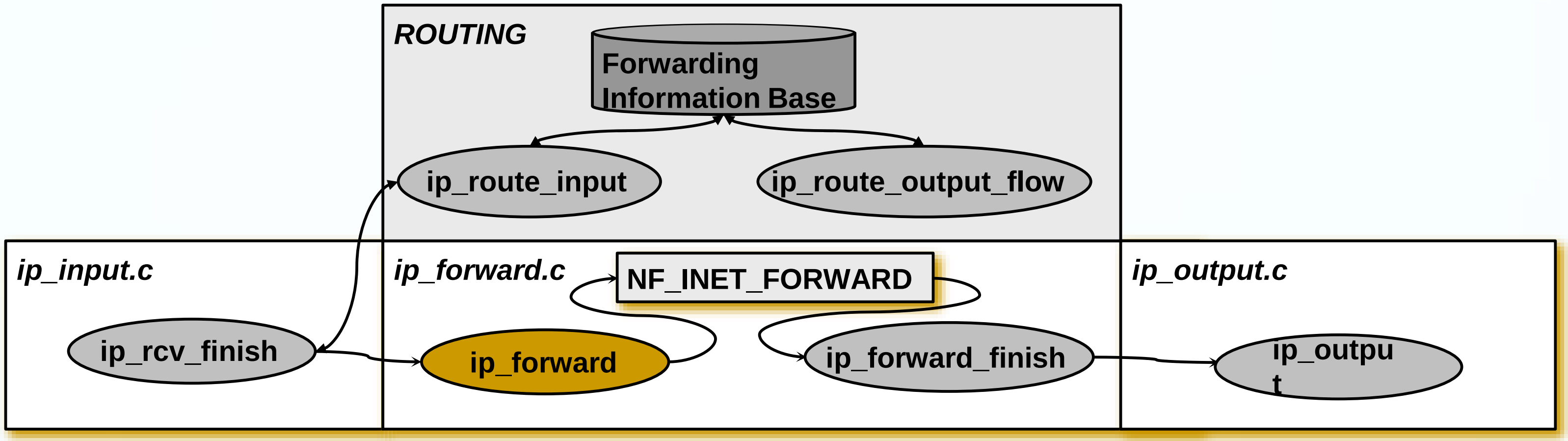


Send Path: ip_finish_output2

- Checks skb for room for MAC header. If not, call *skb_realloc_headroom().*
- Send the packet to a neighbor by:
  - *dst->neighbour->output(skb)*
  - *arp_bind_neighbour()* sees to it that the L2 address (a.k.a. the mac address) of the next hop will be known.
- These eventually end up in *dev_queue_xmit()* which passes the packet down to the device.

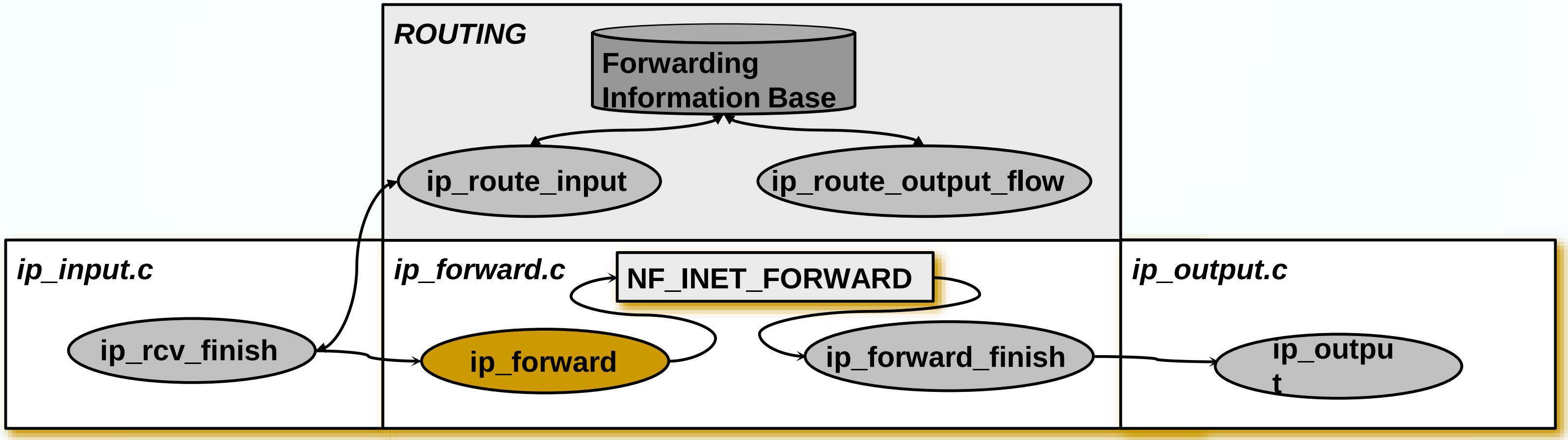


Forwarding: ip_forward (1)



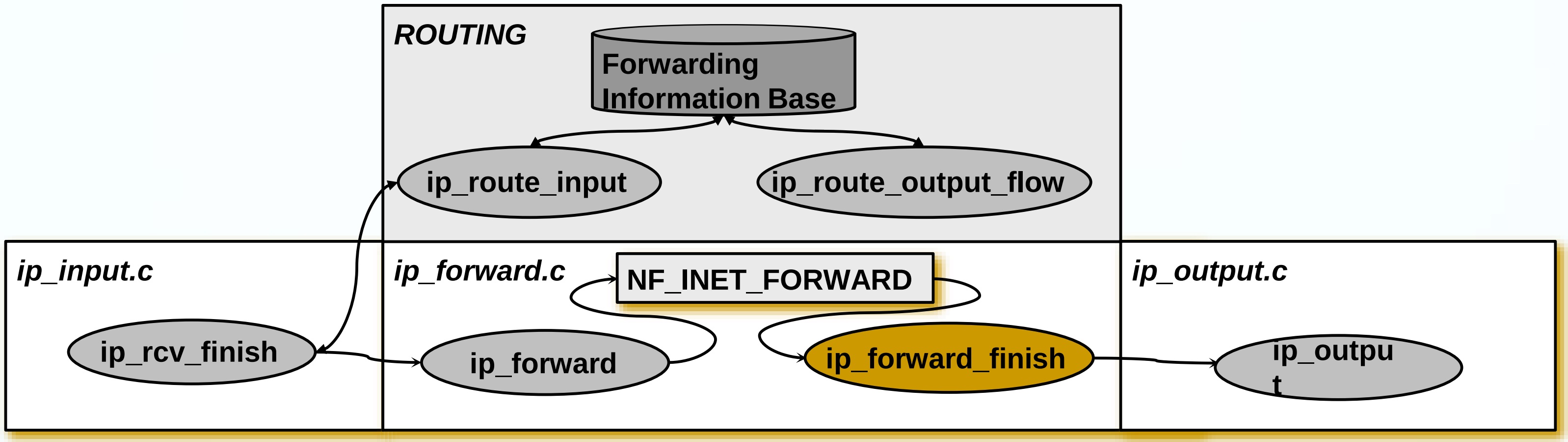
- Does some validation and checking, e.g.,:
  - If skb->pkt_type != PACKET_HOST, drop
  - If *TTL <= 1*, then the packet is deleted, and an ICMP packet with *ICMP_TIME_EXCEEDED* set is returned.
  - If the packet length (including the MAC header) is too large (*skb->len > mtu)* and no fragmentation is allowed (Don't fragment bit is set in the IP header), the packet is discarded and the ICMP message with *ICMP_FRAG_NEEDED* is sent back.

Forwarding: ip_forward (2)



- *skb_cow(skb,headroom)* is called to check whether there is still sufficient space for the MAC header in the output device. If not, *skb_cow()* calls *pskb_expand_head()* to create sufficient space.
- The *TTL* field of the IP packet is decremented by 1.
  - *ip_decrease_ttl()* also incrementally modifies the header checksum.
- The netfilter hook NF_INET_FORWARDING is invoked.

Forwarding: ip_forward_finish



- Increments some stats.
- Handles any IP options if they exist.
- Calls the destination output function via skb->dst->output(skb) – which is *ip_output()*

What UDP Does

UDP packet format

0 3 7	15		31
Source Port (16)		**Destination Port (16)**	
Length (16)		**Checksum (16)**	
Dat a			

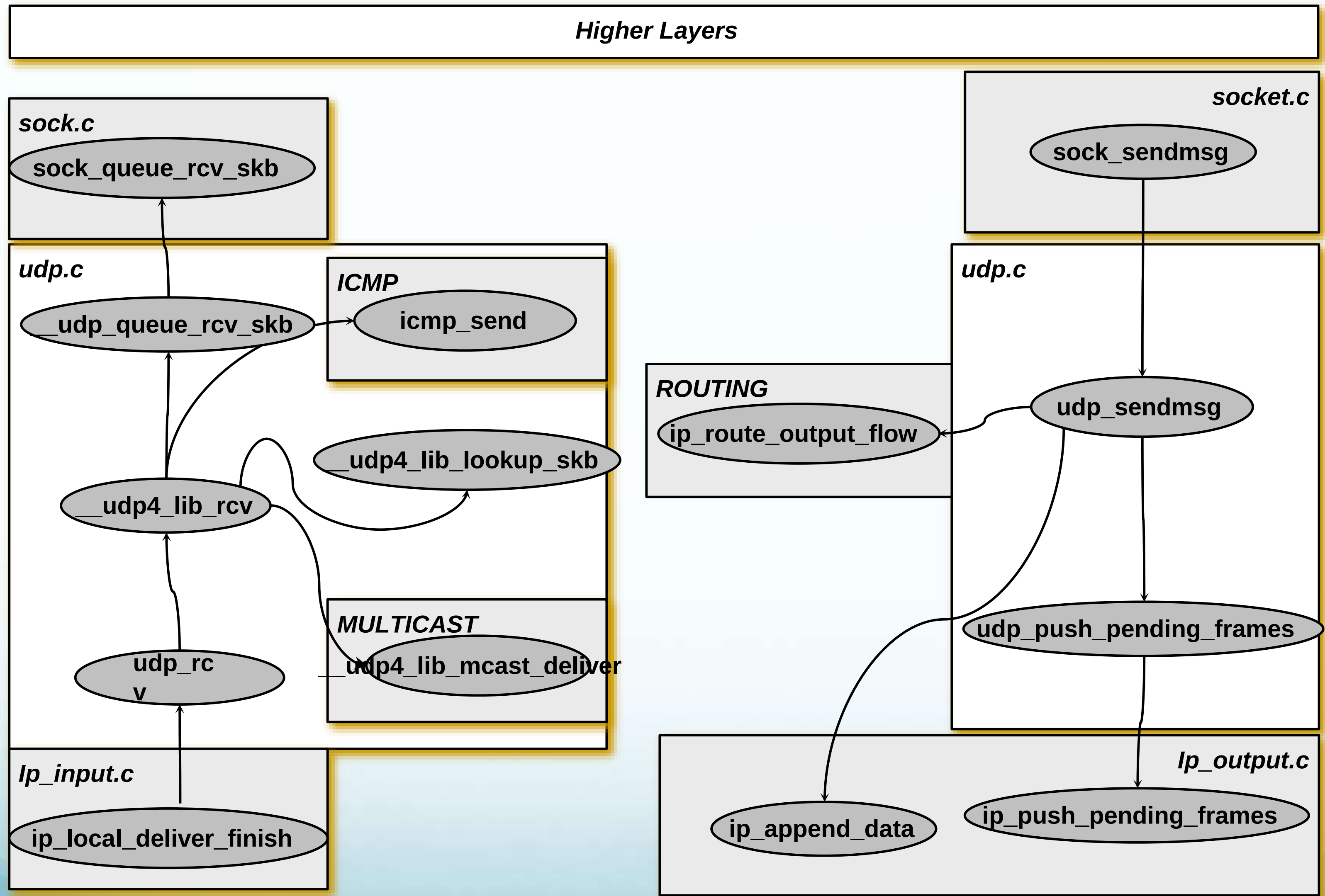
- RFC 768
- IP Proto 17
- Connectionless
- Unreliable
- Datagram
- Supports multicast
- Optional checksum
- Nice and simple.
- Yet still 2187 lines of code!

UDP Header

The udp header: *include/linux/udp.h*

```
struct udphdr {
__be16 source;
__be16 dest;
__be16 len;
__sum16 check;
};
```

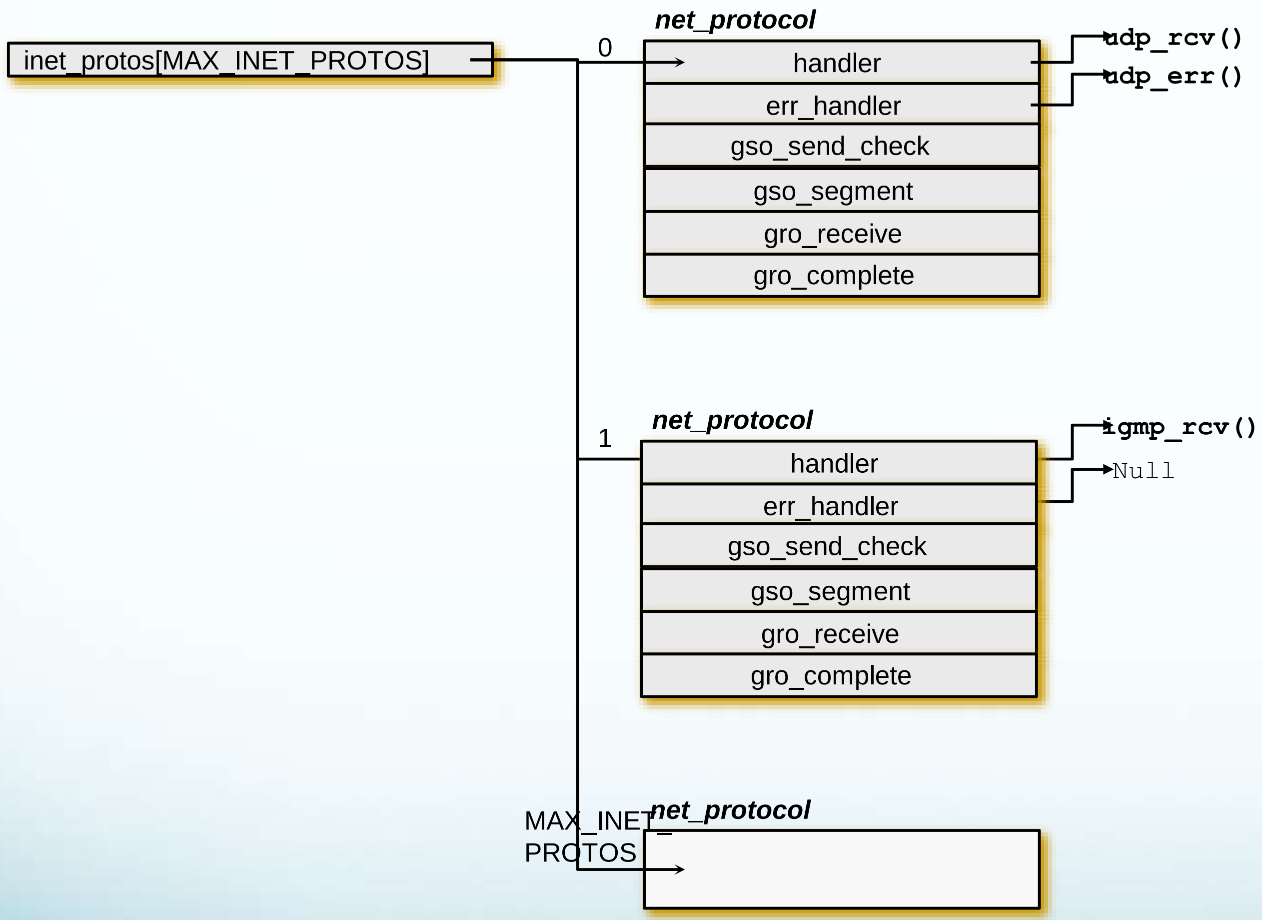
UDP Implementation Design



Receiving packets in UDP

- From **user space**, you can receive udp traffic with three system calls:
  - *recv() (when the socket is connected).*
  - *recvfrom()*
  - *recvmsg()*
- All three are handled by *udp_rcv()* in the kernel.

Recall IP’s *inet_protos*



Sending packets in UDP

- From **user space**, you can send udp traffic with three system calls:
  - *send() (when the socket is connected).*
  - *sendto()*
  - *sendmsg()*
- All three are handled by *udp_sendmsg()* in the kernel.
  - *udp_sendmsg()* is much simpler than the tcp parallel method , *tcp_sendmsg().*
  - *udp_sendpage()* is called when user space calls sendfile() (to copy a file into a udp socket).
    - sendfile() can be used also to copy data between one file descriptor and another.
    - *udp_sendpage() invokes udp_sendmsg().*

BSD Socket API

- Originally developed by UC Berkeley at the dawn of time
- Used by 90% of network oriented programs
- Standard interface across operating systems
- Simple, well understood by programmers

MABEL Technology
Tech The Best

User Space Socket API

- socket() / bind() / accept() / listen()
  - Initialization, addressing and hand shaking
- select() / poll() / epoll()
  - Waiting for events
- send() / recv()
  - Stream oriented (e.g. TCP) Rx / Tx
- sendto() / recvfrom()
  - Datagram oriented (e.g. UDP) Rx / TX
- close(), shutdown()
  - Closing down an association

Socket() System Call

- Creating a socket **from user space** is done by the *socket()* system call:
  - **`int socket (int family, int type, int protocol);`**
  - On success, a file descriptor for the new socket is returned.
  - For open() system call (for files), we also get a file descriptor as the return value.
  - “Everything is a file” Unix paradigm.
- The first parameter, family, is also sometimes referred to as “domain”.

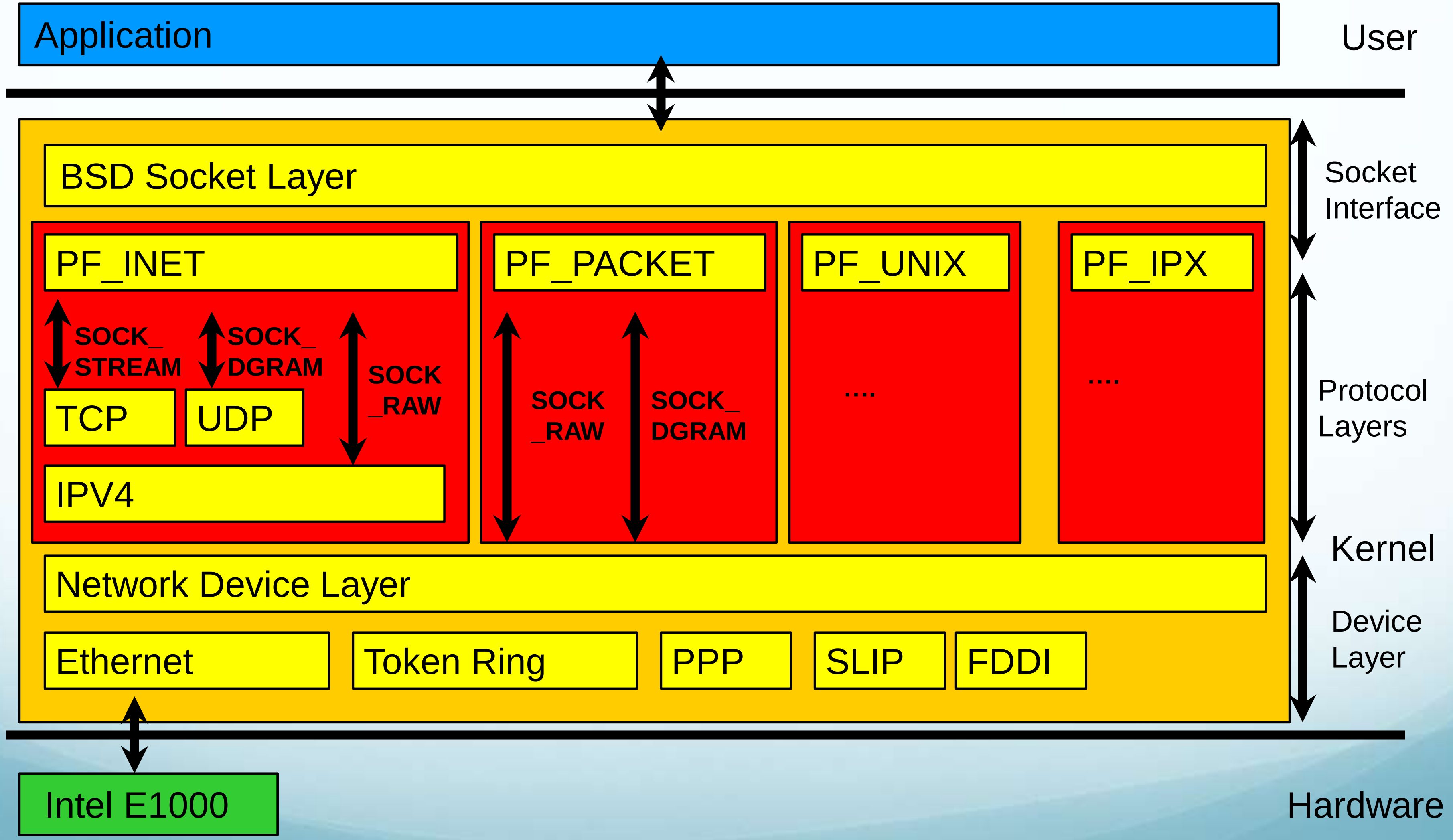
Socket(): Family

- A family is a ***suite of protocols***
- Each family is a subdirectory of linux/net
  - E.g., linux/net/ipv4, linux/net/decnet, linux/net/packet
- IPv4: PF_INET
- IPv6: PF_INET6.
- Packet sockets: PF_PACKET
  - Operate at the device driver layer.
  - pcap library for Linux uses PF_PACKET sockets
  - pcap library is in use by sniffers such as tcpdump.
- Protocol Family == Address Family
  - PF_INET == AF_INET (in /include/linux/socket.h)

Socket(): Type

- SOCK_STREAM and SOCK_DGRAM are the mostly used types.
  - SOCK_STREAM for TCP, SCTP
  - SOCK_DGRAM for UDP.
  - SOCK_RAW for RAW sockets.
  - There are cases where protocol can be either SOCK_STREAM **or** SOCK_DGRAM; for example, Unix domain socket (AF_UNIX).

Socket Layer Architecture



Key Concepts

- Function pointer tables ("ops")
  - In-kernel interfaces for socket functions
    - Binding between BSD sockets and AF_XXX families
    - Binding between AF_INET and transports (TCP, UDP)
- Socket data structures
  - struct socket (BSD socket)
  - struct sock (protocol family socket, network state)
    - struct packet_sock (PF_PACKET)
    - struct inet_sock (PF_INET)
      - struct udp_sock
      - struct tcp_sock

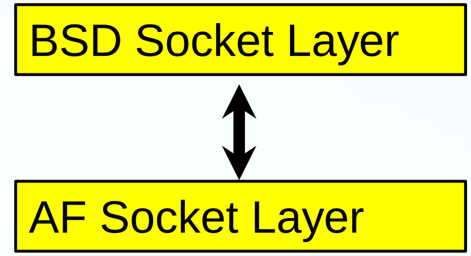
Socket Data Structures

- For every socket which is created by a user space application, there is a corresponding struct **socket** and struct **sock** in the kernel.
- These are **confusing**.
- struct **socket:** include/linux/net.h
  - Data common to the ***BSD socket layer***
  - Has only 8 members
  - Any variable **"sock"** always refers to a struct socket
- struct **sock :** include/net/sock.h
  - Data common to the ***Network Protocol layer (i.e., AF_INET)***
  - has more than 30 members, and is one of the biggest structures in the networking stack.
  - Any variable **"sk"** always refers to a struct sock.

BSD Socket <- -> AF Interface

- Main data structures
  - struct **net_proto_family**
  - struct **proto_ops**
- Key function
- `sock_register(struct net_proto_family *ops)`
- Each address family:
  - Implements the struct **net _proto_family.**
  - Calls the function **sock_register**( ) when the protocol family is initialized.
  - Implement the struct **proto_ops** for binding the BSD socket layer and protocol family layer.

net_proto_family



- Describes each of the supported protocol families
  - `struct net_proto_family {`
    - `int family;`
    - `int (*create)(struct net *net, struct socket *sock, int protocol, int kern);`
    - `struct module *owner;`
  - `}`
- Specifies the handler for socket creation
  - create() function is called whenever a new socket of this type is created

INET and PACKET proto_family

```
static const struct net_proto_family
inet_family_ops = {
     .family = PF_INET,
     .create = inet_create,
     .owner = THIS_MODULE,          /* af_inet.c */
};
static const struct net_proto_family
packet_family_ops = {
     .family = PF_PACKET,
     .create = packet_create,
     .owner = THIS_MODULE,          /* af_packet.c
*/
};
```

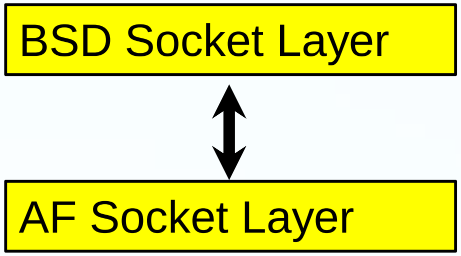
proto_ops

BSD Socket Layer

AF Socket Layer

- Defines the **binding** between the **BSD socket layer** and **address family (AF_*) layer**.
- The **proto_ops** tables contain function exported by the AF socket layer to the BSD socket layer
- It consists of the **address family type** and a set of pointers to **socket operation routines** specific to a particular address family.

struct proto_ops




```
struct proto_ops {
        int                 family;
        struct module       *owner;
        int                 (*release);
        int                 (*bind);
        int                 (*connect);
        int                 (*socketpair);
        int                 (*accept);
        int                 (*getname);
        unsigned int        (*poll);
        int                 (*ioctl);
        int                 (*compat_ioctl);
        int                 (*listen);
        int                 (*shutdown);
        int                 (*setsockopt);
        int                 (*getsockopt);
        int                 (*compat_setsockopt);
        int                 (*compat_getsockopt);
        int                 (*sendmsg);
        int                 (*recvmsg);
        int                 (*mmap);
        ssize_t             (*sendpage);
        ssize_t             (*splice_read);
};
```

include/linux/net.h

PF_PACKET

BSD Socket Layer

↕

AF Socket Layer

```
static const struct proto_ops packet_ops = {
        .family =       PF_PACKET,
        .owner =        THIS_MODULE,
        .release =      packet_release,
        .bind =         packet_bind,
        .connect =      sock_no_connect,
        .socketpair =   sock_no_socketpair,
        .accept =       sock_no_accept,
        .getname =      packet_getname,
        .poll =         packet_poll,
        .ioctl =        packet_ioctl,
        .listen =       sock_no_listen,
        .shutdown =     sock_no_shutdown,
        .setsockopt =   packet_setsockopt,
        .getsockopt =   packet_getsockopt,
        .sendmsg =      packet_sendmsg,
        .recvmsg =      packet_recvmsg,
        .mmap =         packet_mmap,
        .sendpage =     sock_no_sendpage,
};
```

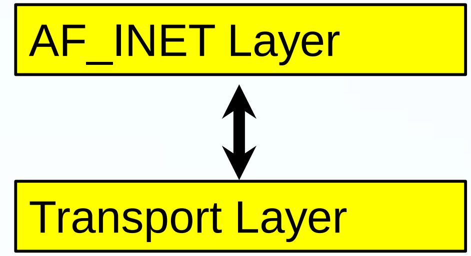
net/packet/af_packet.c

PF_INET proto_ops

	inet_stream_ops (TCP)	inet_dgram_ops (UDP)	inet_sockraw_ops (RAW)
.family	PF_INET	PF_INET	PF_INET
.owner	THIS_MODULE	THIS_MODULE	THIS_MODULE
.release	inet_release	inet_release	inet_release
.bind	inet_bind	inet_bind	inet_bind
.connect	inet_stream_connect	inet_dgram_connect	inet_dgram_connect
.socketpair	sock_no_socketpair	sock_no_socketpair	sock_no_socketpair
.accept	inet_accept	sock_no_accept	sock_no_accept
.getname	inet_getname	inet_getname	inet_getname
.poll	tcp_poll	udp_poll	datagram_poll
.ioctl	inet_ioctl	inet_ioctl	inet_ioctl
.listen	inet_listen	sock_no_listen	sock_no_listen
.shutdown	inet_shutdown	inet_shutdown	inet_shutdown
.setsockopt	sock_common_setsockopt	sock_common_setsockopt	sock_common_setsockopt
.getsockopt	sock_common_getsockop	sock_common_getsockop	sock_common_getsockop
.sendmsg	tcp_sendmsg	inet_sendmsg	inet_sendmsg
.recvmsg	sock_common_recvmsg	sock_common_recvmsg	sock_common_recvmsg
.mmap	sock_no_mmap	sock_no_mmap	sock_no_mmap
.sendpage	tcp_sendpage	inet_sendpage	inet_sendpage
.splice_read	tcp_splice_read	--	--

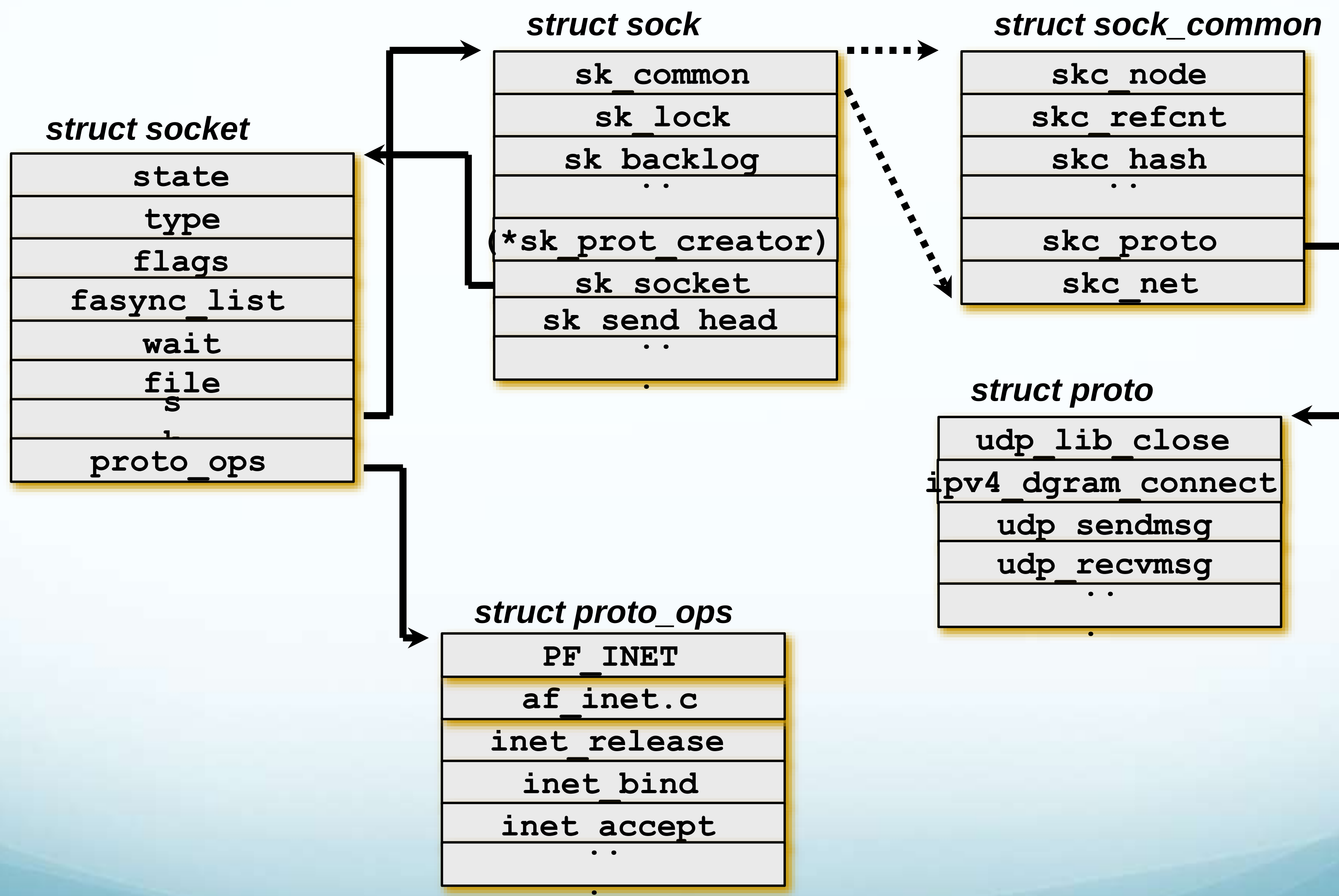
net/ipv4/af_inet.c

AF_INET ←→ Transport API



- struct inet_protos
  - Interface between IP and the transport layer
  - Is the **upcall** binding from IP to transport
  - Method for demultiplexing IP packets to proper transport
- struct proto
  - Defines interface for individual protocols (TCP, UDP, etc)
  - Is the **downcall** binding for AF_INET to transport
  - Transport-specific functions for socket API
- struct inet_protosw
  - Describes the PF_INET protocols
  - Defines the different SOCK types for PF_INET
  - SOCK_STREAM (TCP), SOCK_DGRAM (UDP), SOCK_RAW

Relationships



References

- *«Essential Linux Device Drivers»*, chapter 15
- *«Linux Device Drivers»*, chapter 17 (a little bit old)
- Documentation/networking/netdevices.txt
- Documentation/networking/phy.txt
- include/linux/netdevice.h, include/linux/ethtool.h, include/linux/phy.h, include/linux/sk_buff.h
- And of course, drivers/net/ for several examples of drivers
- Driver code templates in the kernel sources:
  drivers/usb/usb-skeleton.c
  drivers/net/isa-skeleton.c
  drivers/net/pci-skeleton.c
  drivers/pci/hotplug/pcihp_skeleton.c

Advice and Resources

Getting Help and Contributions

Sites

http://www.kernel.org

http://elinux.org

http://free-electrons.com

http://rt.wiki.kernel.org

http://kerneltrap.com

Information Sites (1)

- Linux Weekly News
  http://lwn.net/

The weekly digest off all Linux and free software information sources.

In-depth technical discussions about the kernel.

Subscribe to finance the editors ($5 / month).

Articles available for non-subscribers after 1 week.

LWN.net
Your Linux info source

Useful Reading (1)

- Linux Device Drivers, 3rd edition, Feb 2005

By Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, O'Reilly
http://www.oreilly.com/catalog/linuxdrive3/

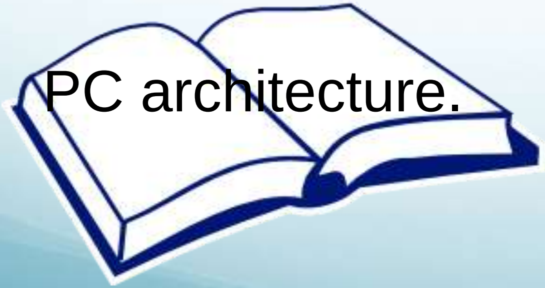
Freely available on-line!
Great companion to the printed book for easy electronic searches!
http://lwn.net/Kernel/LDD3/ (1 PDF file per chapter)
http://free-electrons.com/community/kernel/ldd3/ (single PDF file)

A must-have book for Linux device driver writers!

Useful Reading (2)

Linux Kernel Development, 2nd Edition, Jan 2005
Robert Love, Novell Press
http://rlove.org/kernel_book/
A very synthetic and pleasant way to learn about kernel subsystems (beyond the needs of device driver writers)

Understanding the Linux Kernel, 3rd edition, Nov 2005
Daniel P. Bovet, Marco Cesati, O'Reilly
http://oreilly.com/catalog/understandlk/
An extensive review of Linux kernel internals, covering Linux 2.6 at last.
Unfortunately, only covers the PC architecture.



Useful Reading (3)

Building Embedded Linux Systems, 2nd edition, August 2008
Karim Yaghmour, Jon Masters, Gilad Ben Yossef, Philippe Gerum, O'Reilly Press

http://www.oreilly.com/catalog/belinuxsys/

See http://www.linuxdevices.com/articles/AT2969812114.html for more embedded Linux books.

Useful On-line Resources

Linux kernel mailing list FAQ

http://www.tux.org/lkml/

Complete Linux kernel FAQ.

Read this before asking a question to the mailing list.

Kernel Newbies

http://kernelnewbies.org/

Glossaries, articles, presentations, HOWTOs, recommended reading, useful tools for people getting familiar with Linux kernel or driver development.

International Conferences (1)

- Useful conferences featuring Linux kernel presentations

Linux Symposium (July): http://linuxsymposium.org/
Lots of kernel topics.

Fosdem: http://fosdem.org (Brussels, February)
For developers. Kernel presentations from well-kno
hackers.

CE Linux Forum: http://celinuxforum.org/
Organizes several international technical conf
particular in California (San Jose) and in Japan. Now open to non CELF members!
Very interesting kernel topics for embedded systems developers.

MABEL Technology

Tech The Best

International Conferences (2)

linux.conf.au: http://conf.linux.org.au/ (Australia/New Zealand)
Features a few presentations by key kernel hackers.

Linux Kongress (Germany, September/October)
http://www.linux-kongress.org/
Lots of presentations on the kernel but very expensive registration fees.

Don't miss our free conference videos on http://free-electrons.com/community/videos/conferences/!

MABEL Technology
Tech The Best

Use the Source, Luke!

- Many resources and tricks on the Internet find you will, but solutions to all technical issues only in the Source lie.

Thanks to LucasArts