

Programming Embedded Linux®

Covers Linux version 2.6.24

```
Version 1.4
pgd=00000000
<1>[4d1b65e8] *pgd=00000000[4d1b65e8] *pgd:
Internal error: Oops: f5 [#1]
Internal error: Oops: f5 [#1]
Modules linked in:Modules linked in: hx4700_udc
CPU: 0
CPU: 0
PC is at set_pxa_fb_info+0x2c/0x44
PC is at set_pxa_fb_info+0x2c/0x44
LR is at hx4700_udc_init+0x1c/0x38 [hx4700_udc]
LR is at hx4700_udc_init+0x1c/0x38 [hx4700_udc]
pc : [<c00116c8>] lr : [<bf00901c>] Not tainted
sp : c076df78 ip : 60000093 fp : c076df84
pc : [<c00116c8>] lr : [<bf00901c>] Not tainted
```



Codefidence
a name you can trust™

מכללת ה-ים

Rights to Copy



Attribution – ShareAlike 2.0

You are free

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions



Attribution. You must give the original author credit.



Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

License text: <http://creativecommons.org/licenses/by-sa/2.0/legalcode>

This kit contains work by the following authors:

© Copyright 2004-2006

Michael Opdenacker

michael@free-electrons.com

<http://www.free-electrons.com>

© Copyright 2003-2006

Oron Peled

oron@actcom.co.il

<http://www.actcom.co.il/~oron>

© Copyright 2004–2008

Codefidence Ltd.

info@codefidence.com

<http://www.codefidence.com>

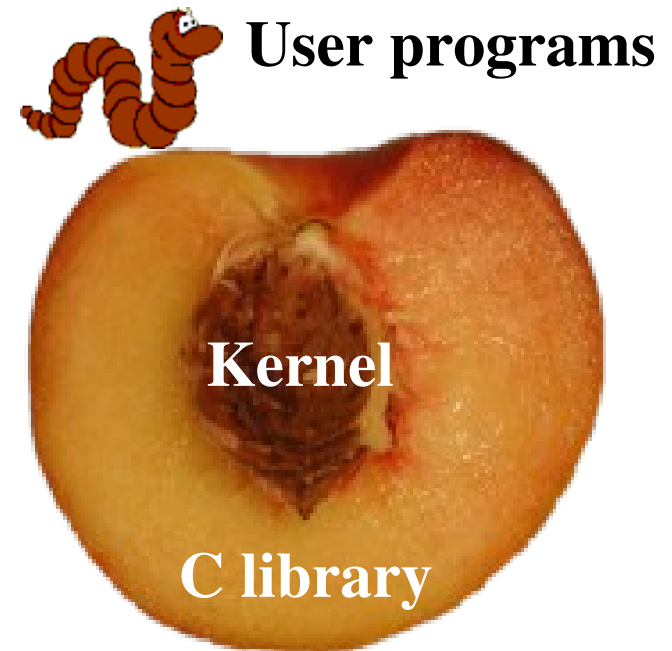
What is Linux?



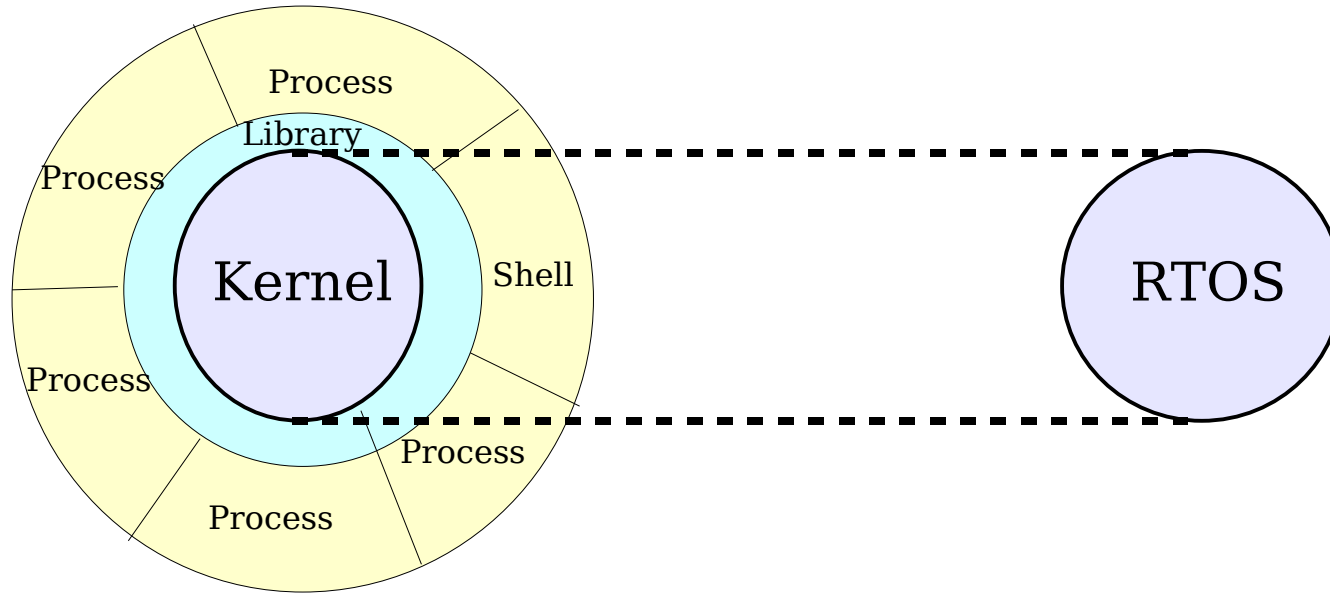
- ▶ Linux is a kernel that implements the POSIX and Single Unix Specification standards which is developed as an open-source project.
 - ▶ Usually when one talks of “installing Linux”, one is referring to a Linux distribution.
- ▶ A distribution is a combination of Linux and other programs and library 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 24 main platforms and supports applications ranging from ccNUMA super clusters to cellular phones and micro controllers.
- ▶ Linux is 15 years old, but is based on the 40 years old Unix design philosophy.

Layers in a Linux System

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



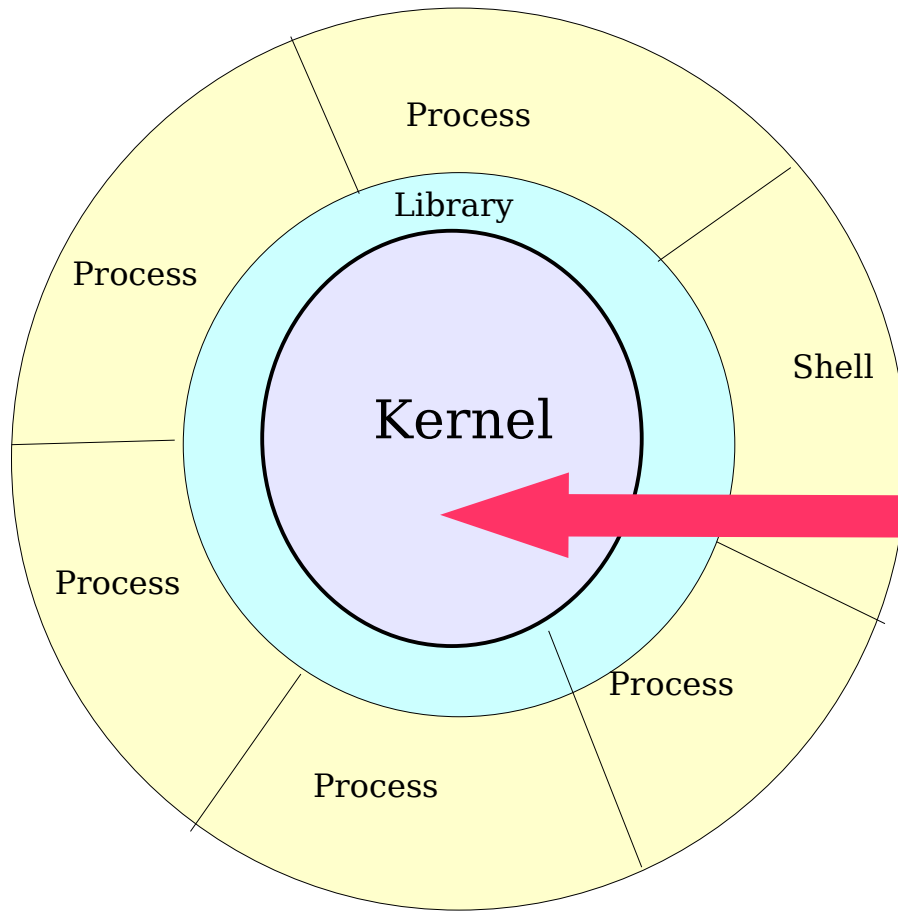
Linux vs. Legacy RTOS



RTOS are like the Linux kernel:
Single program with single memory space that
manages memory, scheduling and interrupts.

Linux also has user tasks that run in their own
memory space. One of them is the shell.

This Course



In this course we will go from the shell, through the system libraries and applications and unto the kernel.

Table of Content

▶ Basic Interaction

- ▶ Files and file system
- ▶ The shell
- ▶ Processes
- ▶ Setting up networking

▶ Application Programming

- ▶ Makefile
- ▶ Processes
- ▶ Threads
- ▶ IPC

▶ Kernel

- ▶ Source layout
- ▶ Coding convention
- ▶ Versions
- ▶ Kernel modules
- ▶ Memory Management
- ▶ Character device
- ▶ Scheduling
- ▶ Soft and hard interrupts
- ▶ Networking

The Unix and GNU/Linux Command-Line



The Unix Filesystem

Everything is a File

Almost everything in Unix is a file!

- ▶ Regular files

- ▶ Directories

Directories are just files
listing a set of files

- ▶ Symbolic links

Files referring to the name of
another file

- ▶ Devices and peripherals

Read and write from devices as
with regular files

- ▶ Pipes

Used to cascade programs

```
cat *.log | grep error
```

- ▶ Sockets

Inter process communication

Filenames

File name features since the beginning of Unix:

- ▶ Case sensitive.
 - ▶ No obvious length limit.
 - ▶ Can contain any character (including whitespace, except /).
- File types stored in the file (“magic numbers”).
- File name extensions not needed and not interpreted. Just used for user convenience.

- ▶ File name examples:

<code>README</code>	<code>.bashrc</code>	<code>Windows Buglist</code>
<code>index.htm</code>	<code>index.html</code>	<code>index.html.old</code>

File Paths

A *path* is a sequence of nested directories with a file or directory at the end, separated by the `/` character.

- ▶ Relative path: `documents/fun/microsoft_jokes.html`
Relative to the current directory
- ▶ Absolute path: `/home/bill/bugs/crash9402031614568`
- ▶ `/` : *root directory*.

Start of absolute paths for all files on the system (even for files on removable devices or network shared).

GNU/Linux Filesystem Structure (1)

Not imposed by the system. Can vary from one system to the other, even between two GNU/Linux installations!

<code>/</code>	Root directory
<code>/bin/</code>	Basic, essential system commands
<code>/boot/</code>	Kernel images, initrd and configuration files
<code>/dev/</code>	Files representing devices
	<code>/dev/hda</code> : first IDE hard disk
<code>/etc/</code>	System configuration files
<code>/home/</code>	User directories
<code>/lib/</code>	Basic system shared libraries

GNU/Linux Filesystem Structure (2)

<code>/lost+found</code>	Corrupt files the system tried to recover
<code>/mnt/</code>	Mounted filesystems <code>/mnt/usbdisk/</code> , <code>/mnt/windows/</code> ...
<code>/opt/</code>	Specific tools installed by the sysadmin <code>/usr/local/</code> often used instead
<code>/proc/</code>	Access to system information <code>/proc/cpuinfo</code> , <code>/proc/version</code> ...
<code>/root/</code>	root user home directory
<code>/sbin/</code>	Administrator-only commands
<code>/sys/</code>	System and device controls (cpu frequency, device power, etc.)

GNU/Linux Filesystem Structure (3)

<code>/tmp/</code>	Temporary files
<code>/usr/</code>	Regular user tools (not essential to the system) <code>/usr/bin/</code> , <code>/usr/lib/</code> , <code>/usr/sbin...</code>
<code>/usr/local/</code>	Specific software installed by the sysadmin (often preferred to <code>/opt/</code>)
<code>/var/</code>	Data used by the system or system servers <code>/var/log/</code> , <code>/var/spool/mail</code> (incoming mail), <code>/var/spool/lpd</code> (print jobs)...

The Unix and GNU/Linux Command-Line



Shells and File Handling

Command-Line Interpreters

- ▶ Shells: tools to execute user commands.
- ▶ Called “shells” because they hide the details on the underlying operating system under the shell's surface.
- ▶ Commands are entered using a text terminal: either a window in a graphical environment, or a text-only console.
- ▶ Results are also displayed on the terminal. No graphics are needed at all.
- ▶ Shells can be scripted: provide all the resources to write complex programs (variable, conditionals, iterations...)

ls Command

Lists the files in the current directory, in alphanumeric order, except files starting with the “.” character.

▶ **ls -a** (all)

Lists all the files (including . * files)

▶ **ls -l** (long)

Long listing (type, date, size, owner, permissions)

▶ **ls -t** (time)

Lists the most recent files first

▶ **ls -S** (size)

Lists the biggest files first

▶ **ls -r** (reverse)

Reverses the sort order

▶ **ls -ltr** (options can be combined)

Long listing, most recent files at the end

Filename Pattern Substitutions

Better introduced by examples:

▶ `ls *txt`

The shell first replaces `*txt` by all the file and directory names ending by `txt` (including `.txt`), except those starting with `.`, and then executes the `ls` command line.

▶ `ls -d .*`

Lists all the files and directories starting with `.`

`-d` tells `ls` not to display the contents of directories.

▶ `ls ?.log`

Lists all the files which names start by 1 character and end by `.log`

Special Directories (1)

`./`

- ▶ The current directory. Useful for commands taking a directory argument. Also sometimes useful to run commands in the current directory (see later).
- ▶ So `./readme.txt` and `readme.txt` are equivalent.

`../`

- ▶ The parent (enclosing) directory. Always belongs to the `.` directory (see `ls -a`). Only reference to the parent directory.
- ▶ Typical usage:
`cd ..`

The `cd` and `pwd` Commands

▶ `cd <dir>`

Change the current directory to `<dir>`.

▶ `pwd`

Displays the current directory ("working directory").

The cp Command

▶ `cp <source_file> <target_file>`

Copies the source file to the target.

▶ `cp file1 file2 file3 ... dir`

Copies the files to the target directory (last argument).

▶ `cp -i` (interactive)

Asks for user confirmation if the target file already exists

▶ `cp -r <source_dir> <target_dir>` (recursive)

Copies the whole directory.

The mv and rm Commands

▶ `mv <old_name> <new_name>` (move)

Renames the given file or directory.

▶ `mv -i` (interactive)

If the new file already exists, asks for user confirm

▶ `rm file1 file2 file3 ...` (remove)

Removes the given files.

▶ `rm -i` (interactive)

Always ask for user confirm.

▶ `rm -r dir1 dir2 dir3` (recursive)

Removes the given directories with all their contents.

Creating and Removing Directories

▶ `mkdir dir1 dir2 dir3 ...` (make dir)

Creates directories with the given names.

▶ `rmdir dir1 dir2 dir3 ...` (remove dir)

Removes the given directories

Safe: only works when directories are empty.

Alternative: `rm -r` (doesn't need empty directories).

Displaying File Contents

Several ways of displaying the contents of files.

▶ `cat file1 file2 file3 ...` (concatenate)

Concatenates and outputs the contents of the given files.

▶ `more file1 file2 file3 ...`

After each page, asks the user to hit a key to continue.

Can also jump to the first occurrence of a keyword
(`/` command).

▶ `less file1 file2 file3 ...`

Does more than `more` with less.

Doesn't read the whole file before starting.

Supports backward movement in the file (`?` command).

The Unix and GNU/Linux Command-Line



Task Control

Full Control Over Tasks

- ▶ Since the beginning, Unix supports true preemptive multitasking.
- ▶ Ability to run many tasks in parallel, and abort them even if they corrupt their own state and data.
- ▶ Ability to choose which programs you run.
- ▶ Ability to choose which input your programs takes, and where their output goes.

Processes

“Everything in Unix is a file
Everything in Unix that is not a file is a process”

Processes

- ▶ Instances of a running programs
- ▶ Several instances of the same program can run at the same time
- ▶ Data associated to processes:
Open files, allocated memory, process id, parent, priority, state...

Running Jobs in Background

Same usage throughout all the shells.

- ▶ Useful:

- ▶ For command line jobs which output can be examined later, especially for time consuming ones.

- ▶ To start graphical applications from the command line and then continue with the mouse.

- ▶ Starting a task: add `&` at the end of your line:

```
find_prince_charming --cute --clever --rich &
```

Background Job Control

▶ jobs

Returns the list of background jobs from the same shell

```
[1]-  Running ~/bin/find_meaning_of_life --without-god &  
[2]+  Running make mistakes &
```

▶ fg

fg %<n>

Puts the last / nth background job in foreground mode

▶ Moving the current task in background mode:

[Ctrl] Z

bg

▶ kill %<n>

Aborts the nth job.

Job Control Example

```
> jobs
[1]-  Running ~/bin/find_meaning_of_life --without-god &
[2]+  Running make mistakes &

> fg
make mistakes

> [Ctrl] Z
[2]+  Stopped make mistakes

> bg
[2]+  make mistakes &

> kill %1
[1]+  Terminated ~/bin/find_meaning_of_life --without-god
```

Listing All Processes

... whatever shell, script or process they are started from

▶ `ps -ux`

Lists all the processes belonging to the current user

▶ `ps -aux` (Note: `ps -edf` on System V systems)

Lists all the processes running on the system

▶ `ps -aux`

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
bart	3039	0.0	0.2	5916	1380	pts/2	S	14:35	0:00	/bin/bash
bart	3134	0.0	0.2	5388	1380	pts/3	S	14:36	0:00	/bin/bash
bart	3190	0.0	0.2	6368	1360	pts/4	S	14:37	0:00	/bin/bash
bart	3416	0.0	0.0	0	0	pts/2	R	15:07	0:00	[bash] ...

▶

PID:	Process id
VSZ:	Virtual process size (code + data + stack)
RSS:	Process resident size: number of KB currently in RAM
TTY:	Terminal
STAT:	Status: R (Runnable), S (Sleep), D (Uninterrupted sleep), Z (Zombie), T(Traced)

Live Process Activity

- ▶ **top** – Displays most important processes, sorted by cpu percentage

```
top - 15:44:33 up 1:11, 5 users, load average: 0.98, 0.61, 0.59
Tasks: 81 total, 5 running, 76 sleeping, 0 stopped, 0 zombie
Cpu(s): 92.7% us, 5.3% sy, 0.0% ni, 0.0% id, 1.7% wa, 0.3% hi, 0.0% si
Mem: 515344k total, 512384k used, 2960k free, 20464k buffers
Swap: 1044184k total, 0k used, 1044184k free, 277660k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
3809	jdoe	25	0	6256	3932	1312	R	93.8	0.8	0:21.49	bunzip2
2769	root	16	0	157m	80m	90m	R	2.7	16.0	5:21.01	X
3006	jdoe	15	0	30928	15m	27m	S	0.3	3.0	0:22.40	kdeinit
3008	jdoe	16	0	5624	892	4468	S	0.3	0.2	0:06.59	autorun
3034	jdoe	15	0	26764	12m	24m	S	0.3	2.5	0:12.68	kscd
3810	jdoe	16	0	2892	916	1620	R	0.3	0.2	0:00.06	top

- ▶ You can change the sorting order by typing

M: Memory usage, **P**: %CPU, **T**: Time.

- ▶ You can kill a task by typing **k** and the process id.

Killing Processes (1)

► `kill <pids>`

Sends a termination signal (SIGTERM) to the given processes. Lets processes save data and exit by themselves. Should be used first.

Example:

```
kill 3039 3134 3190 3416
```

► `kill -9 <pids>`

Sends an immediate termination signal (SIGKILL). The system itself terminates the processes. Useful when a process is really stuck (doesn't answer to `kill -1`).

► `kill -9 -1`

Kills all the processes of the current user. `-1`: means all processes.

Killing Processes (2)

▶ `killall [-<signal>] <command>`

Kills all the jobs running `<command>`. Example:

`killall bash`

Sequential Commands

- ▶ Can type the next command in your terminal even when the current one is not over.
- ▶ Can separate commands with the `;` symbol:
`echo -n "I love thee"; sleep 5; echo " not"`
- ▶ Conditionals: use `||` (or) or `&&` (and):
`more God || echo "Sorry, God doesn't exist"`
Runs `echo` only if the first command fails

`ls mydir && cat mydir/*`

Only cats the directory contents if the `ls` command succeeds (means read access).

The Unix and GNU/Linux Command-Line



System Administration Basics

Shutting Down

▶ `shutdown -h +5` (`-h`: halt)

Shuts the system down in 5 minutes.

Users get a warning in their consoles.

▶ `shutdown -r now` (`-r`: reboot)

▶ `init 0`

Another way to shutdown (used by `shutdown`).

▶ `init 6`

Another way to reboot (used by `shutdown`).

▶ `[Ctrl][Alt][Del]`

Also works on GNU/Linux (at least on PCs!).

Network Setup (1)

► `ifconfig -a`

Prints details about all the network interfaces available on your system.

► `ifconfig eth0`

Lists details about the `eth0` interface

► `ifconfig eth0 192.168.0.100`

Assigns the `192.168.0.100` IP address to `eth0` (1 IP address per interface).

► `ifconfig eth0 down`

Shuts down the `eth0` interface (frees its IP address).



Network Setup (2)

▶ `route add default gw 192.168.0.1`

Sets the default route for packets outside the local network. The gateway (here `192.168.0.1`) is responsible for sending them to the next gateway, etc., until the final destination.

▶ `route`

Lists the existing routes

▶ `route del default`
`route del <IP>`

Deletes the given route

Useful to redefine a new route.

Network Testing

► `ping freshmeat.net`
`ping 192.168.1.1`

Tries to send packets to the given machine and get acknowledgment packets in return.

```
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.  
64 bytes from 192.168.1.1: icmp_seq=0 ttl=150 time=2.51 ms  
64 bytes from 192.168.1.1: icmp_seq=1 ttl=150 time=3.16 ms  
64 bytes from 192.168.1.1: icmp_seq=2 ttl=150 time=2.71 ms  
64 bytes from 192.168.1.1: icmp_seq=3 ttl=150 time=2.67 ms
```

- When you can ping your gateway, your network interface works fine.
- When you can ping an external IP address, your network settings are correct!

Network Setup Summary

Only for simple cases with one interface, no DHCP server...

▶ Connect to the network (cable, wireless card or device...)

▶ Identify your network interface:
`ifconfig -a`

▶ Assign an IP address to your interface (assuming `eth0`)
`ifconfig eth0 192.168.0.100` (example)

▶ Add a route to your gateway (assuming `192.168.0.1`) for packets outside the network:
`route add default gw 192.168.0.1`

Name Resolution

- ▶ Your programs need to know what IP address corresponds to a given host name (such as `kernel.org`)
- ▶ Domain Name Servers (DNS) take care of this.
- ▶ You just have to specify the IP address of 1 or more DNS servers in your `/etc/resolv.conf` file:
`nameserver 217.19.192.132`
`nameserver 212.27.32.177`
- ▶ The changes takes effect immediately!

Mounting Devices (1)

- ▶ To make filesystems on any device (internal or external storage) visible on your system, you have to *mount* them.
- ▶ The first time, create a mount point in your system:
`mkdir /mnt/usbdisk` (example)
- ▶ Now, mount it:
`mount -t vfat /dev/sda1 /mnt/usbdisk`
`/dev/sda1`: physical device
`-t`: specifies the filesystem (format) type
(`ext2`, `ext3`, `vfat`, `reiserfs`, `iso9660`...)

Mounting Devices (2)

- ▶ Lots of mount options are available, in particular to choose permissions or the file owner and group... See the `mount` manual page for details.
- ▶ Mount options for each device can be stored in the `/etc/fstab` file.
- ▶ You can also mount a filesystem image stored in a regular file (*loopback devices*)
 - ▶ Useful to access the contents of an ISO cdrom image without having to burn it.
 - ▶ Useful to create a Linux partition on a hard disk with only Windows partitions

```
cp /dev/sda1 usbkey.img  
mount -o loop -t vfat usbkey.img /mnt/usbdisk
```

Listing Mounted Filesystems

► Just use the `mount` command with no argument:

```
/dev/hda6 on / type ext3 (rw,noatime)
none on /proc type proc (rw,noatime)
none on /sys type sysfs (rw)
none on /dev/pts type devpts (rw,gid=5,mode=620)
usbfs on /proc/bus/usb type usbfs (rw)
/dev/hda4 on /data type ext3 (rw,noatime)
none on /dev/shm type tmpfs (rw)
/dev/hda1 on /win type vfat (rw,uid=501,gid=501)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
```

► Or display the `/etc/mtab` file

(same result, updated by `mount` and `umount` each time they are run)

Unmounting Devices

- ▶ `umount /mnt/usbdisk`

Commits all pending writes and unmounts the given device, which can then be removed in a safe way.

- ▶ To be able to unmount a device, you have to close all the open files in it:

- ▶ Close applications opening data in the mounted partition

- ▶ Make sure that none of your shells have a working directory in this mount point.

- ▶ You can run the `lsuf` command (list open files) to view which processes still have open files in the mounted partition.

Creating Filesystems

Examples

▶ `mkfs.ext2 /dev/sda1`

Formats your USB key (`/dev/sda1`: 1st partition raw data) in ext2 format.

▶ `mkfs.ext2 -F disk.img`

Formats a disk image file in ext2 format

▶ `mkfs.vfat -v -F 32 /dev/sda1` (`-v`: verbose)

Formats your USB key back to FAT32 format.

▶ `mkfs.vfat -v -F 32 disk.img`

Formats a disk image file in FAT32 format.

Blank disk images can be created as in the below example:

```
dd if=/dev/zero of=disk.img bs=1024 count=65536
```

The Unix and GNU/Linux Command-Line



Help

Command Help

Some Unix commands and most GNU/Linux commands offer at least one help argument:

▶ `-h`

(`-` is mostly used to introduce 1-character options)

▶ `--help`

(`--` is always used to introduce the corresponding “long” option name, which makes scripts easier to understand)

You also often get a short summary of options when you input an invalid argument.

Manual Pages

▶ `man [section] <keyword>`

▶ Displays one or several manual pages for `<keyword>` from optional `[section]`.

▶ `man fork`

▶ Man page of the *fork()* system call

▶ `man fstab`

▶ Man page of the fstab configuration file

▶ `man printf`

▶ Man of *printf()* shell command

▶ `man 3 printf`

▶ Man of *printf()* library function

Coding Embedded Linux Applications



Writing Applications

A Simple Makefile

Makefile
variables

`CC=/usr/local/arm/2.95.3/bin/arm-linux-gcc`

`CFLAGS=-g`

`LDFLAGS=-lpthreads`

Always build the
clean and **all**
targets, if asked

`.PHONY: clean all`

To build target
all build target
test

`all: test`

These must be a
TAB, not a
SPACE!

`test.o: test.c test.h`

To compile **test.o**
from **test.c** and
test.h, run the
following command

`$(CC) $(CFLAGS) -C test.c`

`test: test.o`

To link **test** from
test.o, run the
following
command

`$(CC) $(LDFLAGS) test.o -o test`

To **clean**, run the
following
command

`clean:`

`@rm -f *.o *~ test`

A Not-So-Simple Makefile

```
.PHONY: clean all
SRC := $(shell ls *.c)
EXE := test
OBJ := $(SRC:.c=.o)
LDFLAGS := -lsqlite3
CFLAGS := -g
all: $(EXE)
%.o : %.c
    $(COMPILE.c) -MD -o $@ $<
    @cp $*.d $*.P; \
    sed -e 's/#.*//' -e 's/^[^:]*: *//' -e 's/ *\\$$$/ ' \
    -e '/^$$/ d' -e 's/$$/ :/' < $*.d >> $*.P; \
    rm -f $*.d
$(EXE): $(OBJ)
    $(CC) $(LDFLAGS) -o $@ $(OBJ)
clean:
    rm -f $(EXE) $(OBJ) $(SRC:.c=.P) *~
-include $(SRC:.c=.P)

# Source: http://make.paulandlesley.org/autodep.html
```

This Makefile automatically compiles all files ending in `.c` in the current directory, computes dependency information for them and links them into an executable called **test**.

Hello World!

Linux Application API is ISO C and POSIX!

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int i=1;
    printf("Hello World! %d\n", i);
    return 0;
}
```

Creating Processes Using fork()

```
pid_t pid;

if (pid = fork()) {

    int status;

    printf("I'm the parent!\n");

    wait(&status);

    if (WIFEXITED(status))

        printf("Child exist with status of %d\n", WEXITSTATUS(status));

} else {

    printf("I'm the child!\n");

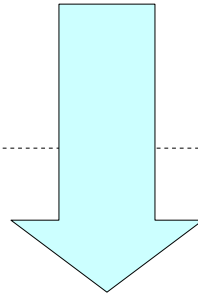
    exit(0);

}
```

How fork() Seems to Work

Parent Process

A complete copy is created of the parent process.



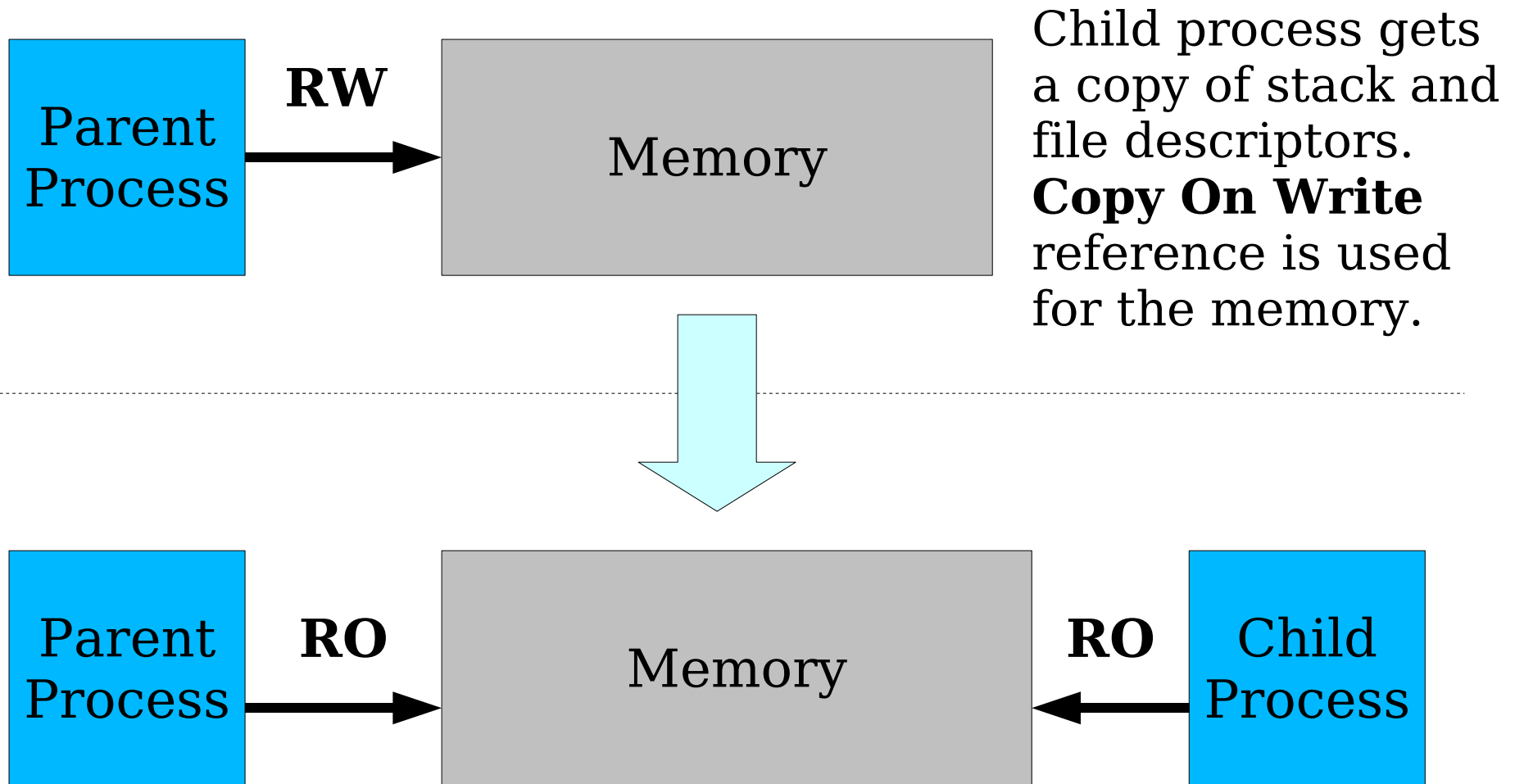
Both parent and child continue execution from the same spot.

Parent Process

Copy
→

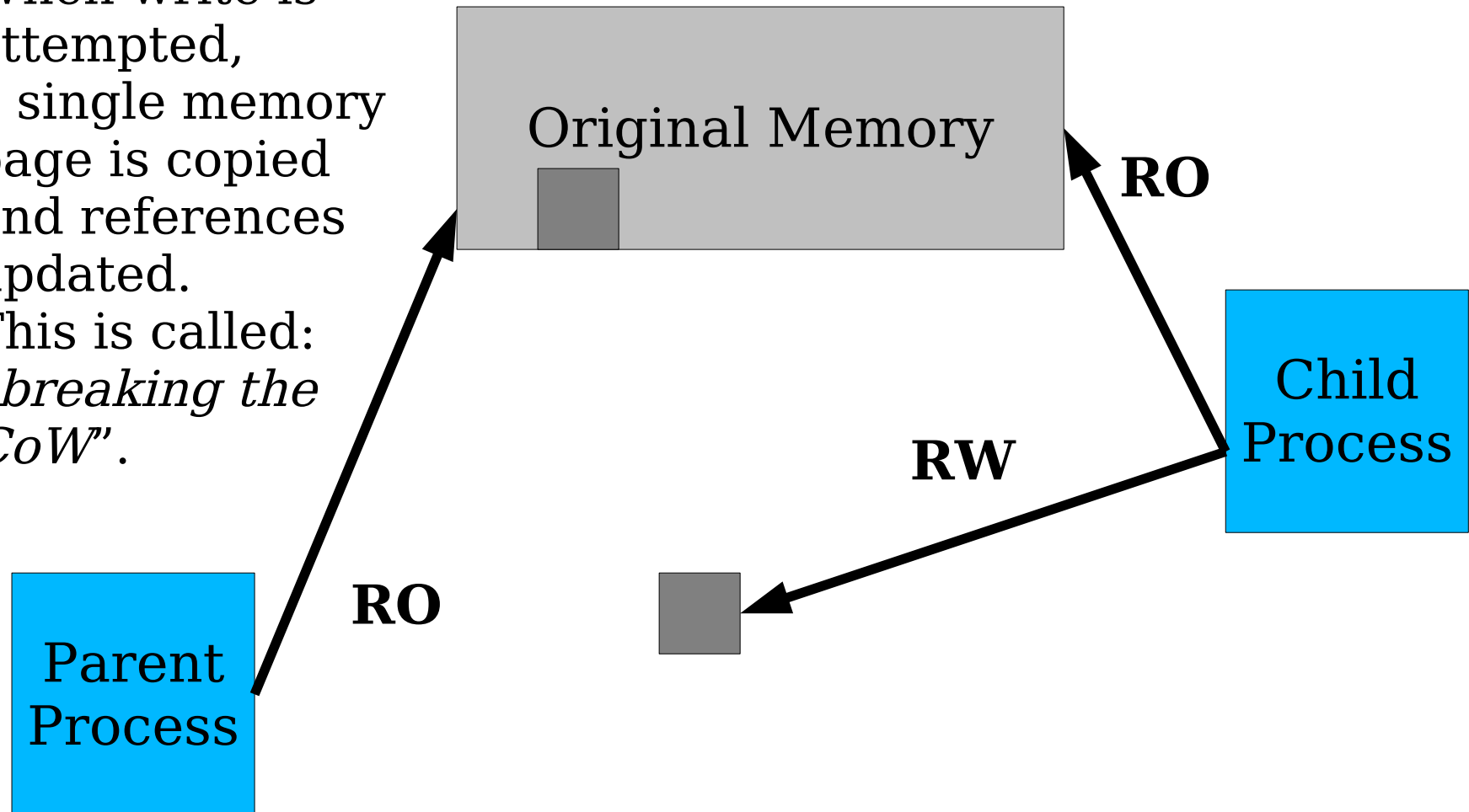
Child Process

How fork() Really Works



What Happens During Write?

When write is attempted, a single memory page is copied and references updated. This is called: *“breaking the CoW”*.



Creating Processes (2)

This call will replace the program memory (code and data) with the image from storage at the specified path.

Open file descriptors, priority and other properties remains the same.

```
pid_t pid;  
  
if (pid = fork()) {  
    int status;  
  
    printf("I'm the parent!\n");  
  
    wait(&status);  
  
} else {  
  
    printf("I'm the child!\n");  
  
    execve("/bin/ls", argv, envp);  
  
}
```

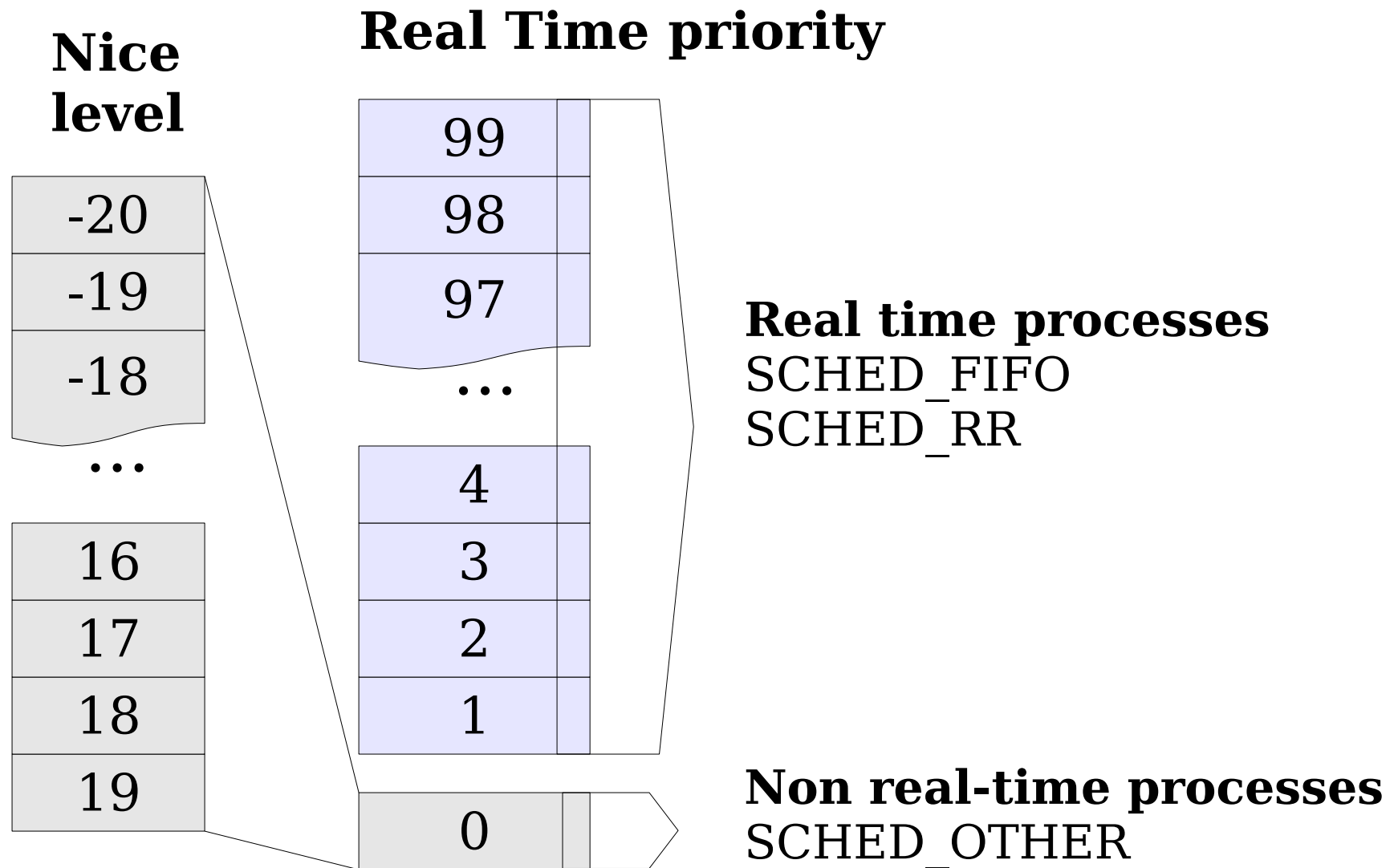
Exiting Processes

- ▶ A process exists when either of the following occurs:
 - ▶ Return from the main function.
 - ▶ Calling the *exit()* function.
 - ▶ Exception (more on those later).
- ▶ A process should return an exit status to its parent process
- ▶ Upon exit:
 - ▶ All exit handlers are called (use *atexit()* to register them)
 - ▶ All memory, file descriptors and other resources are released by the system.

Linux Process Stack

- ▶ Linux process stack is auto expanding.
- ▶ A default stack (8Mb) is allocated at process creation.
- ▶ By default, use of additional stack will automatically trigger allocation of more stack space.
- ▶ This behavior can be limited by setting a resource limit on the stack size.
 - ▶ See *setrlimit()*

Linux Priorities



POSIX Priorities (2)

- ▶ **SCHED_OTHER**: Default Linux time-sharing scheduling
 - ▶ Priority 0 is reserved for it.
 - ▶ Fair, no starvation.
 - ▶ Use this for any non time critical tasks.
- ▶ **SCHED_FIFO**: First In-First Out scheduling
 - ▶ Priorities 1 – 99.
 - ▶ Preemptive.
- ▶ **SCHED_RR**: Round Robin scheduling
 - ▶ Like SCHED_FIFO + time slice

Changing Real Time Priorities

```
int sched_setscheduler(pid_t pid, int policy, const struct sched_param *p);
```

```
struct sched_param {  
    int sched_priority  
};
```

► *sched_setscheduler()* sets both the scheduling policy and the associated parameters for the process identified by *pid*. If *pid* equals zero, the scheduler of the calling process will be set. The interpretation of the parameter *p* depends on the selected policy. Currently, the following three scheduling policies are supported under Linux: `SCHED_FIFO`, `SCHED_RR`, and `SCHED_OTHER`;

► There is also a *sched_getscheduler()*.

Locking Memory

▶ `int mlock(const void *addr, size_t len);`

▶ *mlock* disables paging for the memory in the range starting at *addr* with length *len* bytes.

▶ `int mlockall(int flags);`

▶ *mlockall()* disables paging for all pages mapped into the address space of the calling process.

▶ *MCL_CURRENT* locks all pages which are currently mapped into the address space of the process.

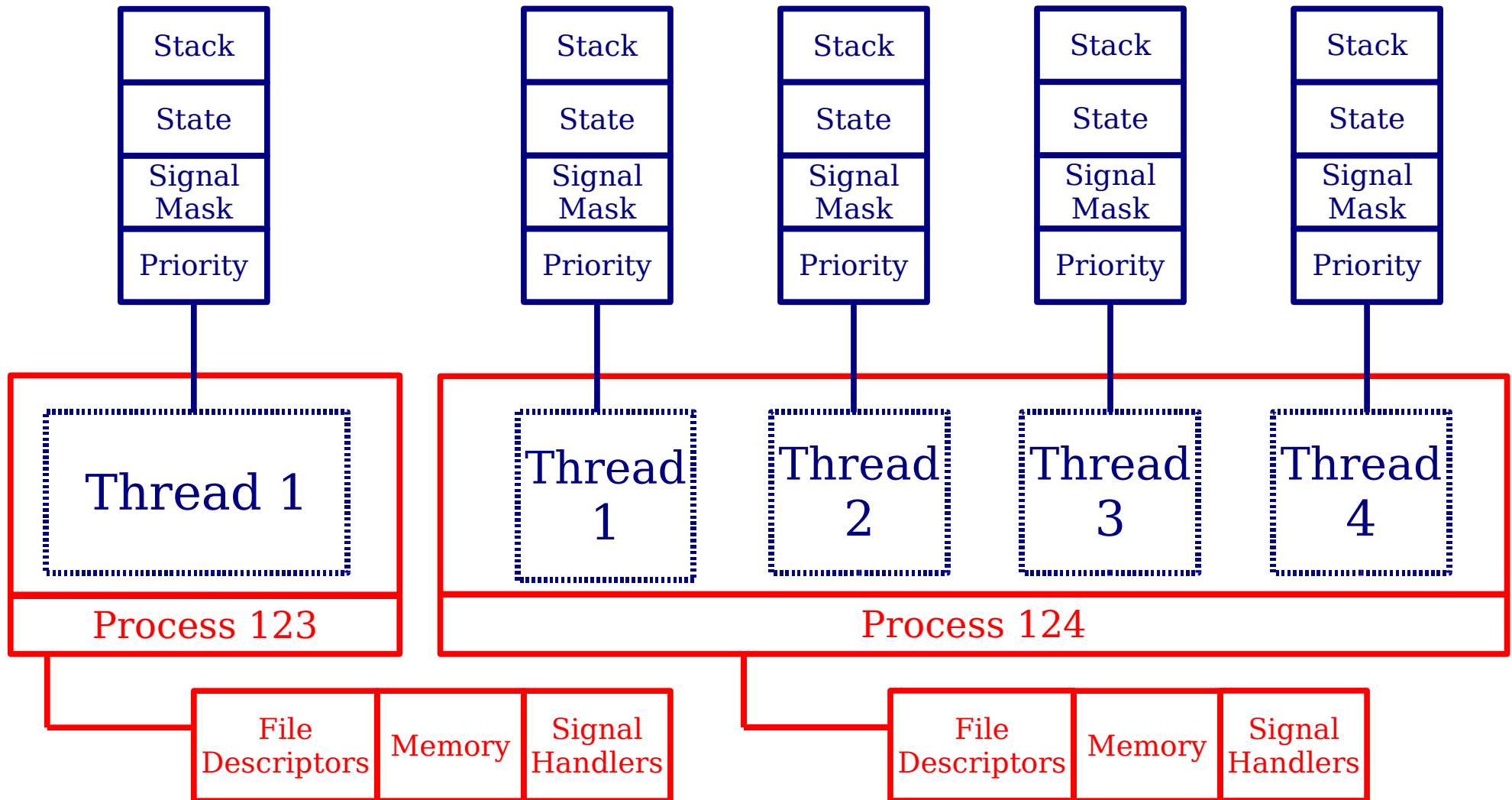
▶ *MCL_FUTURE* locks all pages which will become mapped into the address space of the process in the future. These could be for instance new pages required by a growing heap and stack as well as new memory mapped files or shared memory regions.

▶ **Must lock memory to guarantee real time responses!**

Real Time Responses

- ▶ To get deterministic real time response from a Linux process, make sure to:
 - ▶ Put the process in a real time scheduling domain.
 - ▶ *mlockall()* process memory.
 - ▶ Pre-fault stack pages
 - ▶ To do this call a dummy function that allocates on stack an automatic variable big enough for your entire future stack usage and writes to it.

Threads and Processes



POSIX Threads

- ▶ Linux uses the POSIX Threads threading mechanism.
- ▶ Linux threads are Light Weight Processes – each thread is a task scheduled by the kernel's scheduler.
- ▶ Process creation time is roughly double than that of a thread's.
 - ▶ But Linux process creation time is relatively low...
- ▶ To use threads in your code:
 - ▶ `#include <pthread.h>`
- ▶ In your Makefile:
 - ▶ Add `-lpthread` to `CFLAGS`.

Creating Threads

Function: **pthread_create()**

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
void *(*start_routine)(void *), void * arg);
```

- ▶ The *pthread_create()* routine creates a new thread within a process. The new thread starts in the start routine *start_routine* which has a start argument *arg*. The new thread has attributes specified with *attr*, or default attributes if *attr* is NULL.
- ▶ If the *pthread_create()* routine succeeds it will return 0 and put the new thread ID into *thread*, otherwise an error number shall be returned indicating the error.

Creating Thread Attributes

Function: **pthread_attr_init()**

```
int pthread_attr_init(pthread_attr_t *attr);
```

- ▶ Setting attributes for threads is achieved by filling a thread attribute object *attr* of type *pthread_attr_t*, then passing it as a second argument to *pthread_create(3)*. Passing NULL is equivalent to passing a thread attribute object with all attributes set to their default values.
- ▶ *pthread_attr_init()* initializes the thread attribute object *attr* and fills it with default values for the attributes.
- ▶ Each attribute *attrname* can be individually set using the function *pthread_attr_setattrname()* and retrieved using the function *pthread_attr_getattrname()*.

Destroying Thread Attributes

Function: **pthread_attr_destroy()**

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

- ▶ *pthread_attr_destroy()* destroys a thread attribute object, which must not be reused until it is reinitialized. *pthread_attr_destroy()* does nothing in the LinuxThreads implementation.
- ▶ Attribute objects are consulted only when creating a new thread. The same attribute object can be used for creating several threads. Modifying an attribute object after a call to *pthread_create()* does not change the attributes of the thread previously created.

Detach State

Thread Attribute: **detachstate**

- ▶ Control whether the thread is created in the joinable state or in the detached state. The default is joinable state.
- ▶ In the joinable state, another thread can synchronize on the thread termination and recover its termination code using *pthread_join(3)*, but some of the thread resources are kept allocated after the thread terminates, and reclaimed only when another thread performs *pthread_join(3)* on that thread.
- ▶ In the detached state, the thread resources are immediately freed when it terminates, but *pthread_join(3)* cannot be used to synchronize on the thread termination.
- ▶ A thread created in the joinable state can later be put in the detached thread using *pthread_detach(3)*.

Sched Policy

Thread Attribute: **schedpolicy**

- ▶ Select the scheduling policy for the thread: one of `SCHED_OTHER` (regular, non-realtime scheduling), `SCHED_RR` (realtime, round-robin) or `SCHED_FIFO` (realtime, first-in first-out).
- ▶ Default value: `SCHED_OTHER`.
- ▶ The realtime scheduling policies `SCHED_RR` and `SCHED_FIFO` are available only to processes with superuser privileges.
- ▶ The scheduling policy of a thread can be changed after creation with *`pthread_setschedparam(3)`*.

Sched Param

Thread Attribute: **schedparam**

- ▶ Contain the scheduling parameters (essentially, the scheduling priority) for the thread.
- ▶ Default value: priority is 0.
- ▶ This attribute is not significant if the scheduling policy is `SCHED_OTHER`; it only matters for the realtime policies `SCHED_RR` and `SCHED_FIFO`.
- ▶ The scheduling priority of a thread can be changed after creation with *`pthread_setschedparam(3)`*.

Inherit Sched

Thread Attribute: **inheritsched**

- ▶ Indicate whether the scheduling policy and scheduling parameters for the newly created thread are determined by the values of the schedpolicy and schedparam attributes (PTHREAD_EXPLICIT_SCHED) or are inherited from the parent thread (value PTHREAD_INHERIT_SCHED).
- ▶ Default value: PTHREAD_EXPLICIT_SCHED.

Destroying Threads

Function: **pthread_exit()**

`void pthread_exit(void *status);`

- ▶ The *pthread_exit()* routine terminates the currently running thread and makes *status* available to the thread that successfully joins, *pthread_join()*, with the terminating thread. In addition, *pthread_exit()* executes any remaining cleanup handlers in the reverse order they were pushed, *pthread_cleanup_push()*, after which all appropriate thread specific destructors are called.
- ▶ An implicit call to *pthread_exit()* is made if any thread, other than the thread in which *main()* was first called, returns from the start routine specified in *pthread_create()*.

Thread and Process Termination

<i>Action</i>	<i>Main Thread</i>	<i>Other Thread</i>
<code>pthread_exit()</code>	Thread terminates	Thread terminates
<code>exit()</code>	All threads terminate	All threads terminate
<i>Thread function returns</i>	All threads terminate	Thread terminates

Waiting For a Thread to Finish

Function: **pthread_join()**

`int pthread_join(pthread_t thread, void **status);`

- ▶ If the target thread `thread` is not detached and there are no other threads joined with the specified thread then the *pthread_join()* function suspends execution of the current thread and waits for the target thread to terminate. Otherwise the results are undefined.
- ▶ On a successful call *pthread_join()* will return 0, and if `status` is non NULL then `status` will point to the status argument of *pthread_exit()*. On failure *pthread_join()* will return an error number indicating the error.
- ▶ Also exists *pthread_tryjoin_np()* and *pthread_timedjoin_np()*.

Canceling Threads

Function: **pthread_cancel()**

`int pthread_cancel(pthread_t thread);`

- ▶ Cancellation is the mechanism by which a thread can terminate the execution of another thread. More precisely, a thread can send a cancellation request to another thread. Depending on its settings, the target thread can then either ignore the request, honor it immediately, or defer it till it reaches a cancellation point.
- ▶ When a thread eventually honors a cancellation request, it performs as if `pthread_exit(PTHREAD_CANCELED)` has been called at that point.
- ▶ `pthread_cancel()` sends a cancellation request to the thread denoted by the *thread* argument.

Cancel State

Function: **pthread_setcancelstate()**

`int pthread_setcancelstate(int state, int *oldstate);`

- ▶ *pthread_setcancelstate()* changes the cancellation state for the calling thread - that is, whether cancellation requests are ignored or not.
- ▶ The *state* argument is the new cancellation state: either `PTHREAD_CANCEL_ENABLE` to enable cancellation, or `PTHREAD_CANCEL_DISABLE` to disable cancellation (cancellation requests are ignored).
- ▶ If *oldstate* is not `NULL`, the previous cancellation state is stored in the location pointed to by *oldstate*, and can thus be restored later by another call to *pthread_setcancelstate()*.

Cancel Type

Function: **pthread_setcanceltype()**

`int pthread_setcanceltype(int type, int *oldtype);`

- ▶ *pthread_setcanceltype()* changes the type of responses to cancellation requests for the calling thread: asynchronous (immediate) or deferred.
- ▶ The `type` argument is the new cancellation type: either `PTHREAD_CANCEL_ASYNCHRONOUS` to cancel the calling thread as soon as the cancellation request is received, or `PTHREAD_CANCEL_DEFERRED` to keep the cancellation request pending until the next cancellation point. If `oldtype` is not `NULL`, the previous cancellation state is stored in the location pointed to by *oldtype*, and can thus be restored later by another call to *pthread_setcanceltype()*.

Cancellation Point Defaults

- ▶ Threads are always created by *pthread_create(3)* with cancellation enabled and deferred. That is, the initial cancellation state is `PTHREAD_CANCEL_ENABLE` and the initial type is `PTHREAD_CANCEL_DEFERRED`.

Cancellation Points

- ▶ Cancellation points are those points in the program execution where a test for pending cancellation requests is performed and cancellation is executed if positive.
- ▶ The following POSIX threads functions are cancellation points:
pthread_join(3), *pthread_cond_wait(3)*, *pthread_cond_timedwait(3)*,
pthread_testcancel(3), *sem_wait(3)*, *sigwait(3)*.
- ▶ All other POSIX threads functions are guaranteed not to be cancellation points. That is, they never perform cancellation in deferred cancellation mode.
- ▶ A number of system calls (basically, all system calls that may block, such as *read(2)*, *write(2)*, *wait(2)*, etc.) and library functions that may call these system calls (e.g. *fprintf(3)*) are cancellation points.

Test Cancel

Function: **pthread_testcancel()**

void pthread_testcancel(void);

- ▶ *pthread_testcancel()* does nothing except testing for pending cancellation and executing it. Its purpose is to introduce explicit checks for cancellation in long sequences of code that do not call cancellation point functions otherwise.

Linux IPC

- ▶ POSIX IPC
- ▶ SysV IPC
- ▶ Pipes
- ▶ Unix Domain Sockets
- ▶ Signals
- ▶ *TCP/IP Sockets*

Creating a Mutex

Function: **pthread_mutex_init()**

```
int pthread_mutex_init(pthread_mutex_t *mutex, const  
pthread_mutex_attr *attr);
```

- ▶ The *pthread_mutex_init()* routine creates a new mutex, with attributes specified with *attr*, or default attributes if *attr* is NULL.
- ▶ If the *pthread_mutex_init()* routine succeeds it will return 0 and put the new mutex ID into *mutex*, otherwise an error number shall be returned indicating the error.

Destroying a Mutex

Function: **pthread_mutex_destroy()**

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- ▶ The *pthread_mutex_destroy()* routine destroys the mutex specified by *mutex*.
- ▶ If the *pthread_mutex_destroy()* routine succeeds it will return 0, otherwise an error number shall be returned indicating the error.

Locking Mutexes

Function: **pthread_mutex_lock()**

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- ▶ The *pthread_mutex_lock()* routine shall lock the mutex specified by *mutex*. If the mutex is already locked, the calling thread blocks until the mutex becomes available.
- ▶ If the *pthread_mutex_lock()* routine succeeds it will return 0, otherwise an error number shall be returned indicating the error.

Function: **pthread_mutex_trylock()**

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- ▶ The *pthread_mutex_trylock()* routine shall lock the mutex specified by *mutex* and return 0, otherwise an error number shall be returned indicating the error. In all cases the *pthread_mutex_trylock()* routine will not block the current running thread.

Locking Mutexes

Function: **pthread_mutex_timedlock()**

```
int pthread_mutex_timedlock(pthread_mutex_t *mutex, struct timespec *restrict  
    abs_timeout);
```

- ▶ The *pthread_mutex_timedlock()* function shall lock the mutex object referenced by *mutex*. If the mutex is already locked, the calling thread shall block until the mutex becomes available as in the *pthread_mutex_lock()* function. If the mutex cannot be locked without waiting for another thread to unlock the mutex, this wait shall be terminated when the specified timeout expires.
- ▶ The timeout shall expire when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

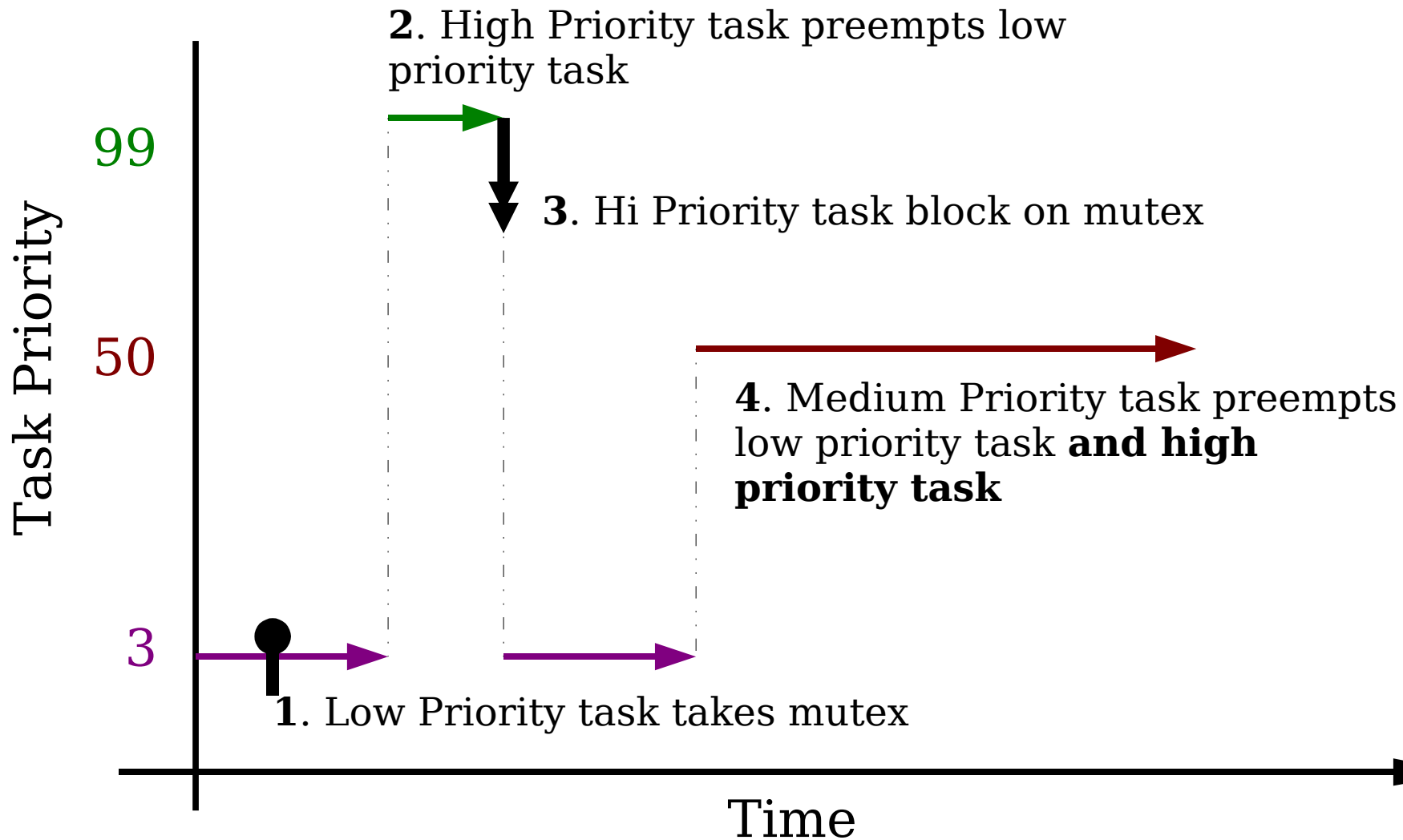
Unlocking Mutexes

Function: **pthread_mutex_unlock()**

`int pthread_mutex_unlock(pthread_mutex_t *mutex);`

- ▶ If the current thread is the owner of the mutex specified by *mutex*, then the *pthread_mutex_unlock()* routine shall unlock the mutex. If there are any threads blocked waiting for the mutex, the scheduler will determine which thread obtains the lock on the mutex, otherwise the mutex is available to the next thread that calls the routine *pthread_mutex_lock()*, or *pthread_mutex_trylock()*.
- ▶ If the *pthread_mutex_unlock()* routine succeeds it will return 0, otherwise an error number shall be returned indicating the error.

Priority Inversion



Priority Inheritance and Ceilings

- ▶ Priority inheritance and ceilings are methods to protect against priority inversions.
- ▶ Linux only got support for them in 2.6.18.
- ▶ Patches to add support for older version exists.
 - ▶ Embedded Linux vendors usually provide patched kernels.
- ▶ If the kernel version you're using is not patched, make sure to protect against this scenario in design
 - ▶ One possible way: raise the priority of each tasks trying to grab a mutex to the maximum priority of all possible contenders.

POSIX Condition Variables

```
int pthread_cond_wait(pthread_cond_t *cond,  
    pthread_mutex_t *mutex);
```

```
int pthread_cond_timedwait(pthread_cond_t *cond,  
    pthread_mutex_t *mutex, const struct timespec  
    *abstime);
```

```
int pthread_cond_signal(pthread_cond_t *cond);
```

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- ▶ The first two function calls are used for waiting on the condition var.
- ▶ The latter two function calls are used to wake up waiting tasks:
 - ▶ *pthread_cond_signal()* wakes up one single task that is waiting for the condition variable.
 - ▶ *pthread_cond_broadcast()* wakes up all waiting tasks.

Condition Variable Usage Example

```
void my_wait_for_event(pthread_mutex_t *lock, pthread_cond_t *cond) {
    pthread_mutex_lock(lock);
    while (flag == 0)
        pthread_cond_wait(cond, lock);
    flag = 1;
    pthread_mutex_unlock(lock);
}

void my_post_event(pthread_mutex_t *lock, pthread_cond_t *cond) {
    pthread_mutex_lock(lock);
    flag = 1;
    pthread_cond_signal(cond);
    pthread_mutex_unlock(lock);
}
```

The condition variable function calls are used within an area protected by the mutex that belong to the condition variable. The operating system releases the mutex every time it blocks a task on the condition variable; and it has locked the mutex again when it unblocks the calling task from the signaling call.

POSIX Semaphores

POSIX Semaphores also available:

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem, int *sval);
int sem_destroy(sem_t *sem);
```

The *pshared* flag is meaningless in Linux and must be 0.

Shared Memory Using mmap()

`void *mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);`

- ▶ The *mmap()* function asks to map *length* bytes starting at *offset* from the file (or device) specified by the file descriptor *fd* into process virtual memory at a kernel chosen address, but preferably *start* with protection of *prot*.
- ▶ The actual place where the object is mapped is returned by *mmap()* (a pointer).
- ▶ The return value in case of an error is `MAP_FAILED` (-1) and **NOT 0!**

MMAP Flags

- ▶ **MAP_SHARED:** Share this mapping with all other processes that map this object. Storing to the region is equivalent to writing to the file. The file may not actually be updated until *msync(2)* or *munmap(2)* are called.
- ▶ **MAP_FIXED:** Do not select a different address than the one specified. If the specified address cannot be used, *mmap()* will fail. If **MAP_FIXED** is specified, *start* must be a multiple of the page size.
- ▶ **MAP_PRIVATE:** Create a private copy-on-write mapping. Stores to the region do not affect the original file. It is unspecified whether changes made to the file after the *mmap()* call are visible in the mapped region.
- ▶ **MAP_ANONYMOUS:** The mapping is not backed by any file; the *fd* and *offset* arguments are ignored.

POSIX Shared Memory

`void *shm_open(const char *name, int oflag, mode_t mode);`

- ▶ *shm_open()* creates and opens a new, or opens an existing, POSIX shared memory object. A POSIX shared memory object is in effect a handle which can be used by unrelated processes to *mmap(2)* the same region of shared memory. The *shm_unlink()* function performs the converse operation, removing an object previously created by *shm_open()*.
- ▶ The operation of *shm_open()* is analogous to that of *open(2)*. *name* specifies the shared memory object to be created or opened. For portable use, *name* should have an initial slash (/) and contain no embedded slashes.
- ▶ Tip: make sure to reserve the size of the shared memory object using *ftruncate()*!

SysV Mailboxes

```
key_t ftok(const char *pathname, int proj_id);
```

```
int msgget(key_t key, int msgflg);
```

```
int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);
```

```
ssize_t msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp,  
int msgflg);
```

- ▶ *ftok()* generates a key from the *pathname* and project ID *proj_id*.
- ▶ *msgget()* creates a mailbox from the key.
- ▶ *msgsnd()* sends a message and *msgrcv()* retrieves it.
- ▶ The different flags can control messages types (priorities), if receiving is a blocking or non-blocking operations and mailbox permissions.

SysV Mailbox Messages

```
struct msgbuf {  
    long mtype;    /* message type, must be > 0 */  
    char mtext[1]; /* message data */  
};
```

- ▶ The *mtext* field is an array (or other structure) whose size is specified by *msgsz* parameter.
- ▶ The *mtype* field must have a positive integer value that can be used by the receiving process for message selection.

SysV Mailbox Message Types

- ▶ For *msgrcv()*, the *msgtyp* argument specifies the type of message requested as follows:
 - ▶ If *msgtyp* is 0, then the first message in the queue is read.
 - ▶ If *msgtyp* is greater than 0, then the first message on the queue of type *msgtyp* is read, unless MSG_EXCEPT was asserted in *msgflg*, in which case the first message on the queue of type not equal to *msgtyp* will be read.
 - ▶ If *msgtyp* is less than 0, then the first message on the queue with the lowest type less than or equal to the absolute value of *msgtyp* will be read.

SysV Mailboxes Flags

- ▶ The *msgflg* argument asserts none, one or more (or-ing them) of the following flags:
 - ▶ **IPC_NOWAIT** For immediate return if no message of the requested type is on the queue. The system call fails with *errno* set to ENOMSG.
 - ▶ **MSG_EXCEPT** Used with *msgtyp* greater than 0 to read the first message on the queue with message type that differs from *msgtyp*.
 - ▶ **MSG_NOERROR** To truncate the message text if longer than *msgsz* bytes.

POSIX Pipes

`int pipe(int fildes[2]);`

- ▶ *pipe()* creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by *fildes*. *fildes[0]* is for reading, *fildes[1]* is for writing.

`FILE *popen(const char *command, const char *type);`

- ▶ The *popen()* function opens a process by creating a pipe, forking, and invoking the shell. Since a pipe is by definition unidirectional, the *type* argument may specify only reading or writing, not both; the resulting stream is correspondingly read-only or write-only.
- ▶ A named version of pipes, called **fifo** also exists.

UNIX Domain Sockets

- ▶ Files in the file system that acts like sockets.
- ▶ All normal socket operations applies.

```
struct sockaddr_un server;
```

```
int sock;
```

```
sock = socket(AF_UNIX, SOCK_STREAM, 0);
```

```
server.sun_family = AF_UNIX;
```

```
strcpy(server.sun_path, SOCKET_NAME);
```

```
bind(sock, (struct sockaddr *) &server, sizeof(struct sockaddr_un));
```

- ▶ UDS can be used to pass file descriptors between processes using a special socket option operation.

Signals

- ▶ Signals are asynchronous notifications sent to a process by the kernel or another process
- ▶ Signals interrupt whatever the process was doing at the time to handle the signal.
- ▶ Each signal may have a **signal handler**, which is a function that gets called when the process receives that signal.
- ▶ Two default signal handlers also exist:
 - ▶ **SIG_IGN**: Causes the process to ignore the specified signal.
 - ▶ **SIG_DFL**: Causes the system to set the default signal handler for the given signal.

Regular Signals

- **SIGHUP** Hangup detected on controlling terminal
- **SIGINT** Interrupt from keyboard
- **SIGQUIT** Quit from keyboard
- **SIGILL** Illegal Instruction
- **SIGABRT** Abort signal from abort(3)
- **SIGFPE** Floating point exception
- **SIGKILL** Kill signal
- **SIGSEGV** Invalid memory reference
- **SIGPIPE** Broken pipe: write to pipe with no readers
- **SIGALRM** Timer signal from alarm(2)
- **SIGTERM** Termination signal
- **SIGUSR1** User-defined signal 1
- **SIGUSR2** User-defined signal 2
- **SIGCHLD** Child stopped or terminate
- **SIGCONT** Continue if stopped
- **SIGSTOP** Stop process
- **SIGTSTP** Stop typed at tty
- **SIGTTIN** tty input for background process
- **SIGTTOU** tty output for background process
- **SIGBUS** Bus error (bad memory access)
- **SIGPOLL** Pollable event (Sys V). Synonym of SIGIO
- **SIGPROF** Profiling timer expired
- **SIGSYS** Bad argument to routine (SVID)
- **SIGTRAP** Trace/breakpoint trap
- **SIGURG** Urgent condition on socket (4.2 BSD)
- **SIGIO** I/O now possible (4.2 BSD)

See *signal(7)* for default behaviors.

There are two signal handlers you cannot modify or ignore – SIGKILL and SIGSTOP.

Limitations of Regular Signals

▶ Too few signals

- ▶ Only 2 signals available for user defined purposes: USR1, USR2. All the rest have predefined meaning.

▶ No queuing

- ▶ If the same signal is sent multiple times until a task processes it, it is only delivered once.

▶ No priority

- ▶ Multiple signals delivered according to order sent.

▶ Portability

- ▶ Sys V signal handlers are “popped” before executed (sadly, also a race-condition).

▶ Signal Masking

- ▶ Installing the same signal handler for multiple signals may cause re-entrancy.

▶ No passing of a value with the signal

Real Time Signals

- ▶ Additional 32 signals from SIGRTMIN to SIGRTMAX
- ▶ No predefined meaning...
 - ▶ But LinuxThreads lib makes use of the first 3.
- ▶ Multiple instances of the same signals are queued.
- ▶ Value can be sent with the signal.
- ▶ Priority is guaranteed:
 - ▶ Lowest number real time signals are delivered first. Same signals are delivered according to order they were sent.
 - ▶ Regular signals have higher priority then real time signals.

Signal Action

`int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`

- ▶ Register a signal handler.
 - ▶ *signum*: signal number.
 - ▶ *act*: pointer to new *struct sigaction*.
 - ▶ *oldact*: pointer to buffer to be filled with current *sigaction* (or NULL, if not interested).

Signal Action cont.

- ▶ The *sigaction* structure is defined as something like:

```
▶ struct sigaction {  
    void (*sa_handler)(int);  
    void (*sa_sigaction)(int, siginfo_t *, void *);  
    sigset_t sa_mask;  
    int sa_flags;  
    ...  
}
```

- ▶ *sa_mask* gives a mask of signals which should be blocked during the execution of the signal handler.

- ▶ The signal which triggered the handler will also be blocked, unless the SA_NODEFER or SA_NOMASK flags are used.

Real Time Signals Flags

- ▶ *sa_flags* can be used to pass flags to change behavior:
 - ▶ **SA_ONESHOT**: Restore the signal action to the default state once the signal handler has been called.
 - ▶ **SA_RESTART**: Make blocking system calls restart automatically after a signal is received.
 - ▶ **SA_NODEFER**: Do not prevent the signal from being received from within its own signal handler.
 - ▶ **SA_SIGINFO**: The signal handler takes 3 arguments, not one. In this case, *sa_sigaction* should be set instead of *sa_handler*.
 - ▶ For details about *siginfo_t* structure, see *sigaction(2)*.

Sending Signals

`int sigqueue(pid_t pid, int sig, const union sigval value);`

- ▶ Queue signal to process.
- ▶ *pid* is the process ID to send the signal to.
- ▶ *sig* is the signal number.
- ▶ *sigval* is:

```
union sigval {  
    int  sival_int;  
    void *sival_ptr;  
};
```

- ▶ The *sigval* is available to the handler via the *sig_value* field of *siginfo_t*.

Signal Masking

- ▶ The *sigprocmask()* call is used to change the list of currently blocked signals.

`int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`

- ▶ The behavior of the call is dependent on the value of *how*, as follows:

- ▶ **SIG_BLOCK:** The set of blocked signals is the union of the current set and the set argument.
- ▶ **SIG_UNBLOCK:** The signals in set are removed from the current set of blocked signals.
- ▶ **SIG_SETMASK:** The set of blocked signals is set to the argument set.

Signal Sets

▶ These functions allow the manipulation of POSIX signal sets:

▶ **int sigemptyset(sigset_t *set);**

▶ Initializes the signal set given by *set* to empty, with all signals excluded from the set.

▶ **int sigfillset(sigset_t *set);**

▶ Initializes set to full, including all signals.

▶ **int sigaddset(sigset_t *set, int signum);**

▶ **int sigdelset(sigset_t *set, int signum);**

▶ Add and delete respectively signal *signum* from set.

Signals & Threads

- ▶ Signal masks are **per thread**.
- ▶ Signal handlers are **per process**.
- ▶ Exception signals (SIGSEGV, SIGBUS...) will be caught by **thread doing the exception**.
- ▶ Other signals will be caught by **any thread in the process** whose mask does not block the signal – use *pthread_sigmask()* to modify the thread's signal mask.
- ▶ **Tip:** Use a “signal handler” thread that does *sigwait(3)* to make thread catching less random!

Processes Timers

- ▶ `#include <sys/time.h>`
- ▶ `int getitimer(int which, struct itimerval *value);`
- ▶ `int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue);`
- ▶ The system provides each process with three interval timers, each decrementing in a distinct time domain.
- ▶ When any timer expires, a signal is sent to the process, and the timer (potentially) restarts.

Timers Expiry

- ▶ The timer interval is controlled by the following structures:

```
▶ struct itimerval {  
    struct timeval it_interval; /* next value */  
    struct timeval it_value;    /* current value */  
};  
  
▶ struct timeval {  
    long tv_sec;                /* seconds */  
    long tv_usec;              /* microseconds */  
};
```

- ▶ Timers decrement from it_value to zero, generate a signal, and reset to it_interval.
- ▶ A timer which is set to zero (it_value is zero or the timer expires and it_interval is zero) stops.

Time Domains

▶ **ITIMER_REAL**

- ▶ Decrements in real time, and delivers SIGALRM upon expiration.

▶ **ITIMER_VIRTUAL**

- ▶ Decrements only when the process is executing, and delivers SIGVTALRM upon expiration.

▶ **ITIMER_PROF**

- ▶ Decrements both when the process executes and when the system is executing on behalf of the process.
- ▶ SIGPROF is delivered upon expiration.

More on Timers

- ▶ Timers will never expire before the requested time, instead expiring some short, constant time afterwards, dependent on the system timer resolution.
- ▶ If the timer expires while the process is active (always true for ITIMER_VIRT), the signal will be delivered immediately when generated.
- ▶ Otherwise, the delivery will be offset by a small time dependent on the system load.
- ▶ An alternate interface called `timer_create()` allows specifying which signal will be sent (utilize real time signals) and spawning of a thread on timer expiry.

Debugging

- ▶ GDB: The GNU Debugger
 - ▶ GDB runs on the host
 - ▶ Either standalone or via a graphical front end like Eclipse/CDT.
 - ▶ GDBserver runs on the target
 - ▶ GDBserver can attach to an already running processes or start new processes under the debugger.
 - ▶ GDB talks to GDBserver via TCP or UART
 - ▶ Need to have the executable with debug information on the host.
 - ▶ Also, system and application dynamic libraries, if used.
 - ▶ No need to have debug symbols in executable on target.
 - ▶ See http://www.codefidence.com/sourcedrop/course/gdb_with_embedded_linux_cheat_sheet.pdf

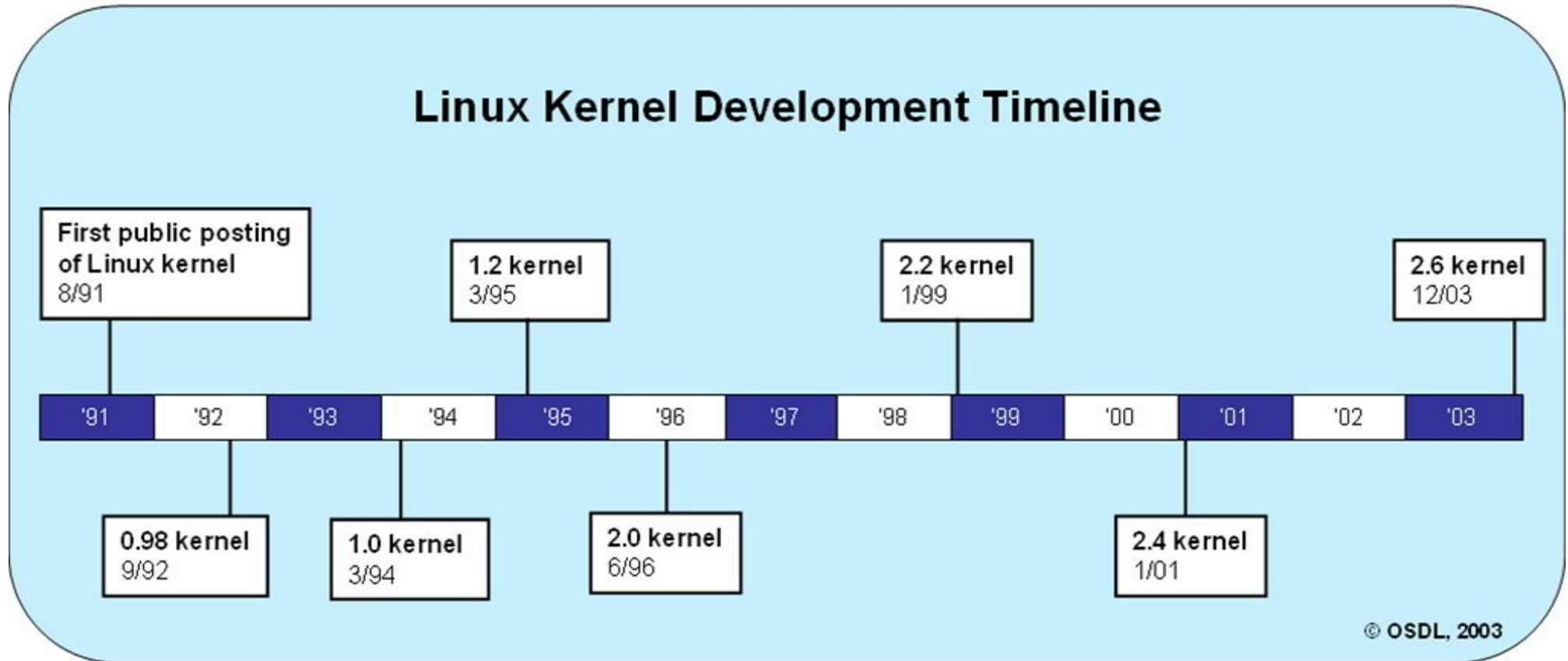
Embedded Linux Driver Development



Kernel Overview

Linux Features

Linux Kernel Development Timeline




Linux Stable Releases

Major versions

- ▶ 1 major version every 2 or 3 years

Examples: 1.0, 2.0, 2.4, 2.6



Even number

Stable releases

- ▶ 1 stable release every 1 or 2 months

Examples: 2.0.40, 2.2.26, 2.4.27, 2.6.7 ...

Stable release updates (since March 2005)

- ▶ Updates to stable releases up to several times a week
Address only critical issues in the latest stable release

Examples: 2.6.11.1 to 2.6.11.7

Linux Development and Testing Releases

Testing releases

- ▶ Several testing releases per month, before the next stable one.
You can contribute to making kernel releases more stable by testing them!

Example: `2.6.12-rc1`

Development versions

- ▶ Unstable versions used by kernel developers before making a new stable major release

Examples: `2.3.42`, `2.5.74`



Odd number

Continued Development in Linux 2.6

- ▶ Since 2.6.0, kernel developers have been able to introduce lots of new features one by one on a steady pace, without having to make major changes in existing subsystems.
- ▶ Opening a new Linux 2.7 (or 2.9) development branch will be required only when Linux 2.6 is no longer able to accommodate key features without undergoing traumatic changes.
- Thanks to this, more features are released to users at a faster pace.
- However, the internal kernel API can undergo changes between two 2.6.x releases. A module compiled for a given version may no longer compile or work on a more recent one.

2.4 vs. 2.6

Linux 2.4

- Mature
- But developments stopped; very few developers willing to help.
- Now obsolete and lacks recent features.
- Still fine if you get your sources, tools and support from commercial Linux vendors.

Linux 2.6

- 3.5 years old stable Linux release!
- Support from the Linux development community and all commercial vendors.
- Now mature and more exhaustive. Most drivers upgraded.
- Cutting edge features and increased performance.

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.

Supported Hardware Architectures

- ▶ 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

Embedded Linux Driver Development



Kernel Overview

Kernel Code

Linux Sources Structure (1)

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

Linux Sources Structure (2)

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

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()`...

Kernel Stack

- ▶ Very small and fixed stack.

- ▶ 2 page stack (8k), per task.

- ▶ Or 1 page stack, per task and one for interrupts.

2.6 ▶ Chosen in build time via menu.

- ▶ Not for all architectures

- ▶ For some architectures, the kernel provides debug facility to detect stack overruns.

Managing Endianness

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
```

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 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)

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);
```


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



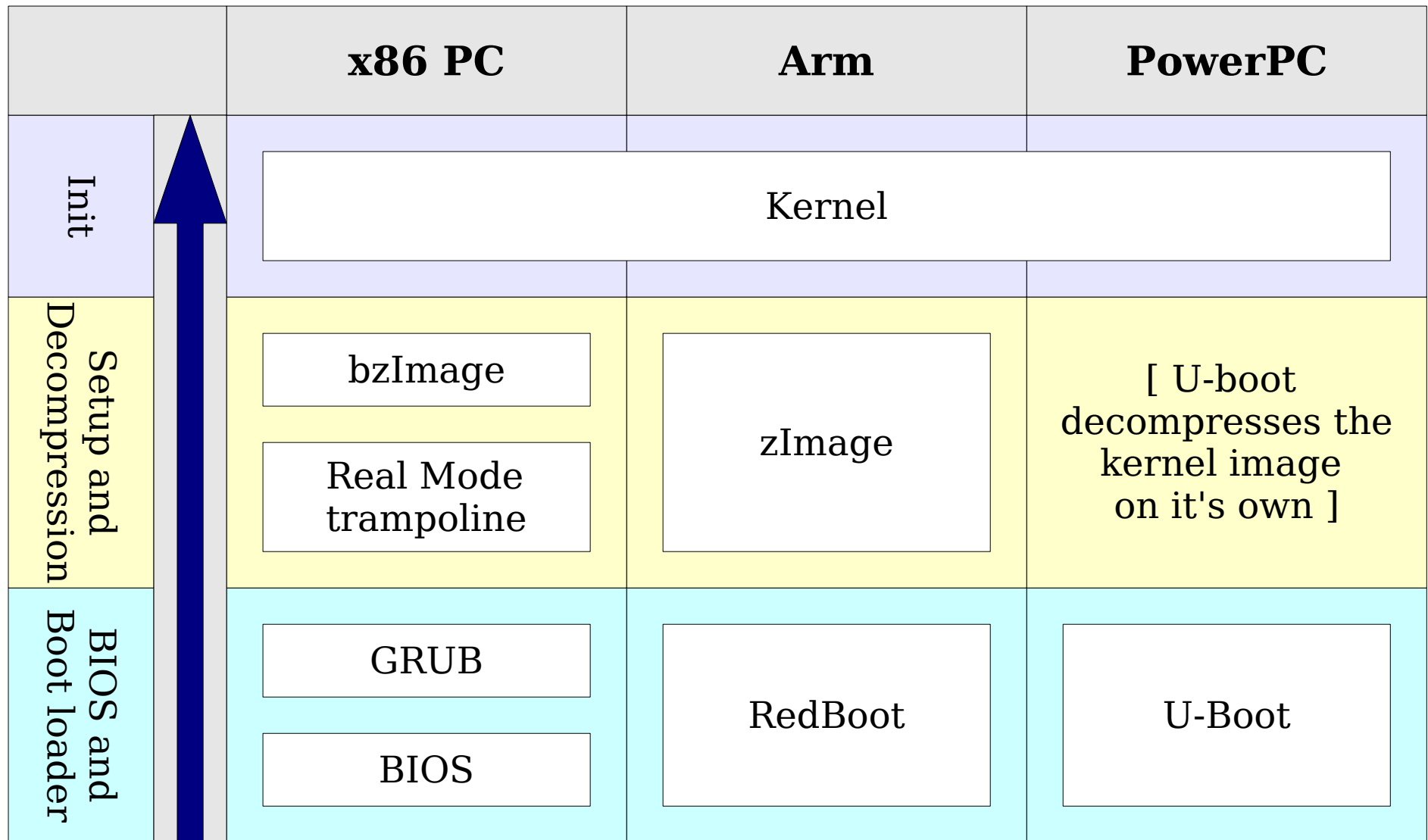
Kernel Overview

Boot Sequence

Linux Boot Process

-
- ▶ BIOS and/or bootloader initializes hardware.
 - ▶ Bootloader loads kernel image into memory.
 - ▶ Bootloader can get kernel image from flash, HD, network.
 - ▶ Possibly also loads a filesystem to RAM in the form of initrd or initramfs archives.
 - ▶ Bootloader or kernel decompress compressed kernel.
 - ▶ Kernel performs internal (hash table, lists etc.) and hardware (device driver) setup.
 - ▶ Kernel finds and mounts the root filesystem.
 - ▶ Kernel executes the “/sbin/init” application.
-

Boot Sequences



Root Filesystem Options

	External FS	Initrd Style	Initramfs style
Kernel Version	2.4 and 2.6	2.4 and 2.6	2.6 only
Storage format	Filesystem on storage device or network	Filesystem image	CPIO archive
Provided by	Passed to kernel as parameter ("root=")	Loaded and location provided by boot loader	Like initrd or statically linked into kernel image
Stored in	On device	Fixed allocation RAM disk	Dynamically allocated page cache
Run time Location	Copied to page cache	Copied to page cache	Already in page cache

init – the First Process

- ▶ The first (and only) process the kernel starts is *init*.
 - ▶ By default it is searched in /sbin/init
 - ▶ Can be overridden by the kernel parameter “init=”.
- ▶ *init* has three roles:
 - ▶ To setup the system configuration and start applications.
 - ▶ To shut down the system applications.
 - ▶ To serve as the parent process of all child processes whose parent has exited.
- ▶ The default init implementation reads its instructions from the file /etc/inittab

inittab

This is an Busybox style inittab (for an example for Sys V inittab see the Appendix.)

Format:

id: runlevel : **action** : command

id: To which device should std input/output go (empty means the console)

runlevel: ignored. For compatibility with Sys V init.

action: one of sysinit, respawn, askfirst, wait, shutdown and once

command: shell command to execute

Startup the system

```
::sysinit:/bin/mount -o remount,rw /  
::sysinit:/bin/mount -t proc proc /proc  
::sysinit:/bin/mount -a  
::sysinit:/sbin/ifconfig lo 127.0.0.1 up
```

Put a getty on the serial port

```
::respawn:/sbin/getty -L ttyS1 115200 vt100
```

Start system loggers

```
null::respawn:/sbin/syslogd -n -m 0  
null::respawn:/sbin/klogd -n
```

Stuff to do before rebooting

```
null::shutdown:/usr/bin/killall klogd  
null::shutdown:/usr/bin/killall syslogd  
null::shutdown:/bin/umount -a -r
```

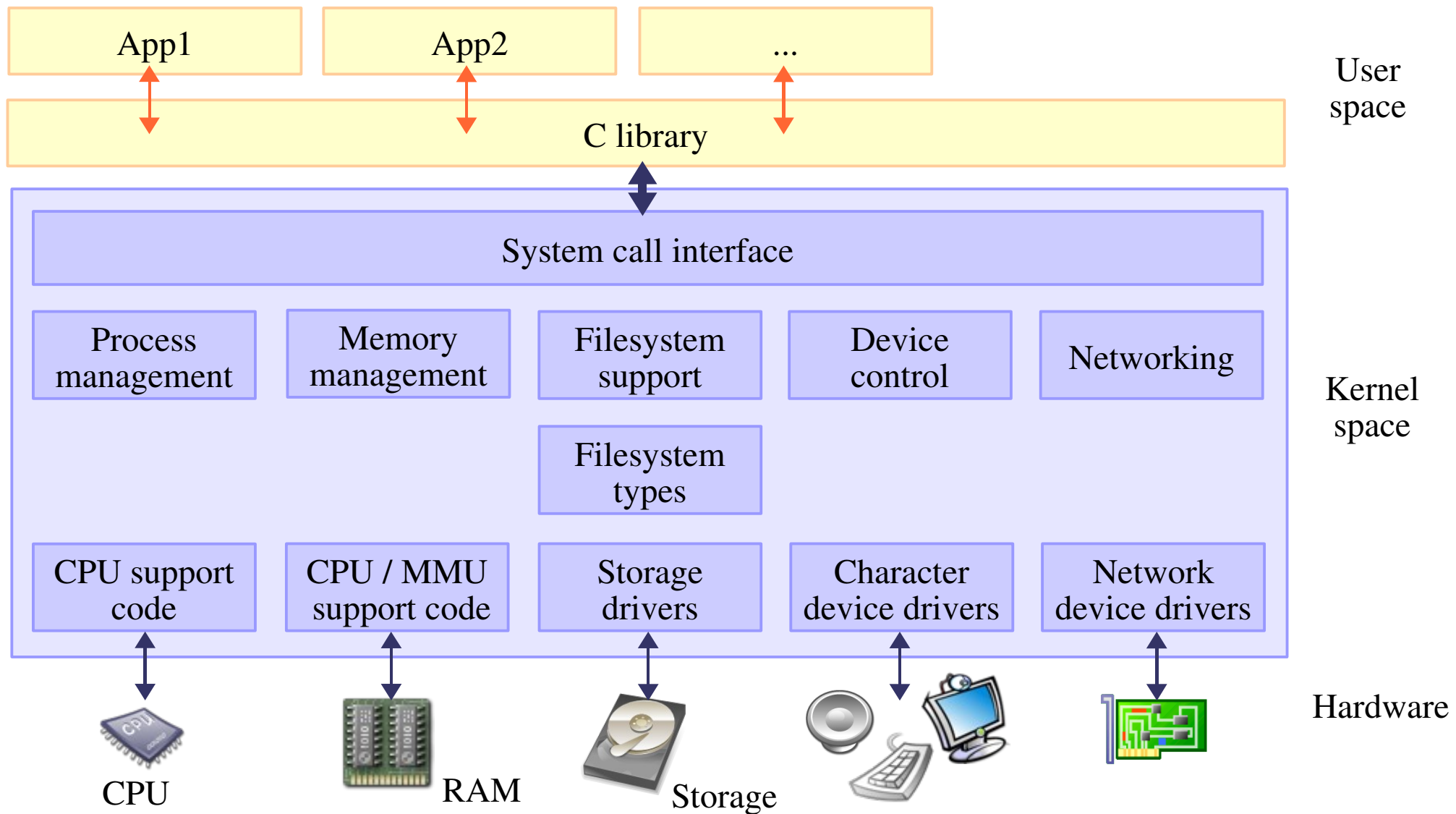

Embedded Linux Driver Development



Kernel Overview

Kernel Subsystems

Kernel Architecture



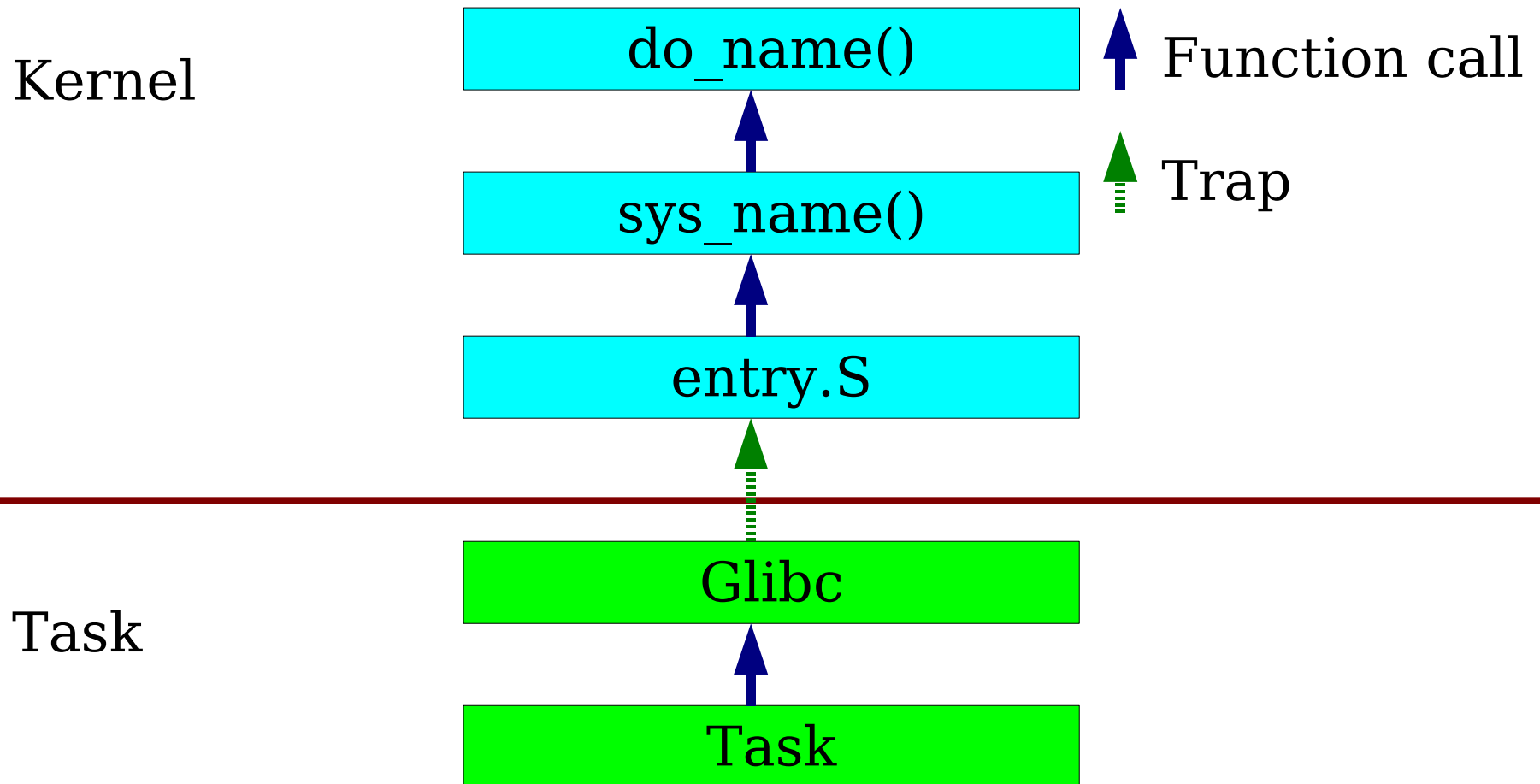
Kernel-Mode vs. User-Mode

- ▶ 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 task 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 then sets the instruction pointer to the kernel.
- ▶ The kernel will find the required system call 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.

Linux System Call Path



Embedded Linux Driver Development



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

```
/* 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>

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 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>]`

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)

Embedded Linux Driver Development



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
```



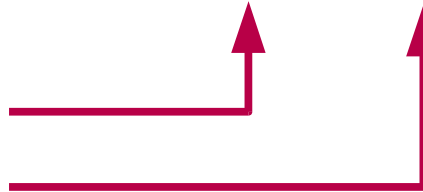
- ▶ Through `modprobe`

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 */
);
```

Example

```
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 */
);
```

Example

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

Embedded Linux Driver Development



Driver Development Using the proc Filesystem Interface

Proc Filesystem Interface

- ▶ `/proc` is a virtual filesystem that exports kernel internal structures to user-space
 - ▶ `/proc/cpuinfo`: processor information
 - ▶ `/proc/meminfo`: memory status
 - ▶ `/proc/version`: version and build information
 - ▶ `/proc/cmdline`: kernel command line
 - ▶ `/proc/<pid>/fd`: process used file descriptors
 - ▶ `/proc/<pid>/cmdline`: process command line
 - ▶ `/proc/sys/kernel/panic`: time in second until reboot in case of fatal error
 - ▶ ...

User-space 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`

Hello Module with proc File

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

#define MYNAME "driver/my_proc_file"

static struct proc_dir_entry *my_proc_dir = NULL;

int mymodule_proc_read(char *page, char **start, off_t off,
    int count, int *eof, void *data) {

    int len = 0;

    len += sprintf(page, "io=%d\n", io);
    len += sprintf(page + len, "irq=%d\n", irq);
    if (len <= off+count)
        *eof = 1;
    *start = page + off;
    len -= off;
    if (len > count) len = count;
    if (len < 0) len = 0;
    return len;
}

int __init startup_mymodule(void) {
    my_proc_dir = create_proc_entry(MYNAME, 0, NULL);
    my_proc_dir->read_proc = mymodule_proc_read;
    return 0;
}

void __exit shutdown_mymodule(void) {
    remove_proc_entry(MYNAME, NULL);
}
```

The proc file name

The proc_dir_entry struct

Callback function

page is the buffer we write to

Setting *eof to 1 means end of file.

start is set to a pointer to where we wrote

second parameters is file permission, last parameter is a handle of directory.

Don't forget to check for NULL here! **178**

Some More proc Details

- ▶ You can also register a `proc_write()` callback.
- ▶ `proc_dir_entry` has a data field. The kernel does not use it, but whatever you set there will be returned to you as the last parameter of the callback.
- ▶ The permissions (2nd) parameter of `create_proc_entry()` is the same as the mode flags of the `open(2)` system call.
 - ▶ 0 means use the system wide defaults.
- ▶ The directory handle (3rd) parameter of `create_proc_entry()` is an address of `proc_dir_entry` for a proc directory.
 - ▶ NULL means the proc root directory.
- ▶ For large and complex files use the `seq_file` wrapper.

Using a proc File

- ▶ Once the module is loaded, you can access the registered proc file:
 - ▶ From the shell:
 - ▶ **Read** `cat /proc/driver/my_proc_file`
 - ▶ **Write** `echo "123" > /proc/driver/my_proc_file`
 - ▶ Programatically, using `open(2)`, `read(2)` `write(2)` and related functions.
- ▶ You can't delete, move or rename a proc file.
- ▶ proc files usually don't have reported size.

Embedded Linux Driver Development



Driver Development Memory Management

Physical Memory

- ▶ In ccNUMA¹ machines:
 - ▶ The memory of each node is represented at *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

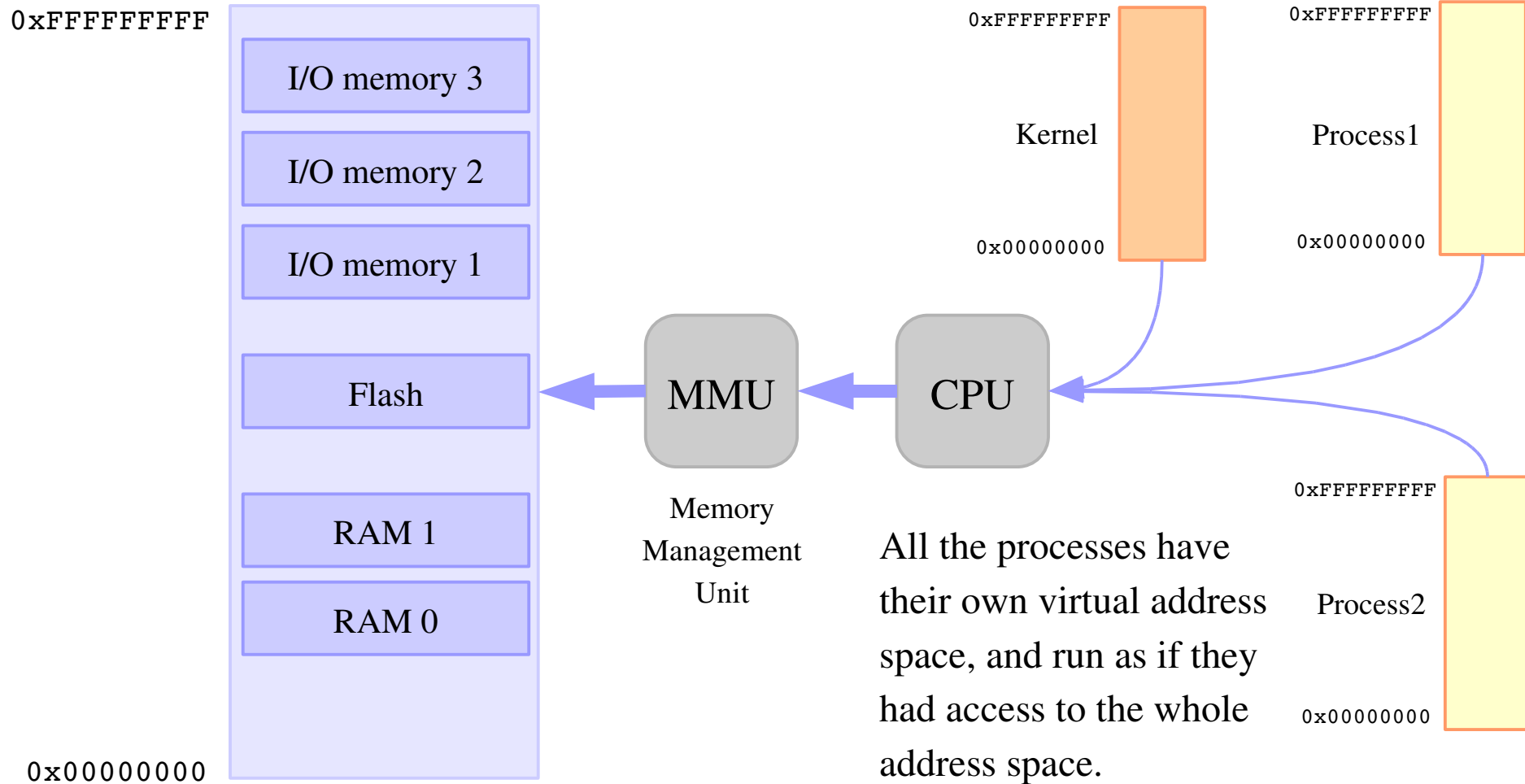
- ▶ Each *pg_data_t* is split to three zones
- ▶ Each zone has different properties:
 - ▶ **ZONE_DMA**
 - ▶ DMA operations on address limited buses is possible.
 - ▶ **ZONE_NORMAL**
 - ▶ Maps directly to linear addressing (<~1Gb on i386)
 - ▶ Always mapped to kernel space.
 - ▶ **ZONE_HIMEM**
 - ▶ Rest of memory.
 - ▶ Mapped into kernel-space on demand.

Physical and Virtual Memory

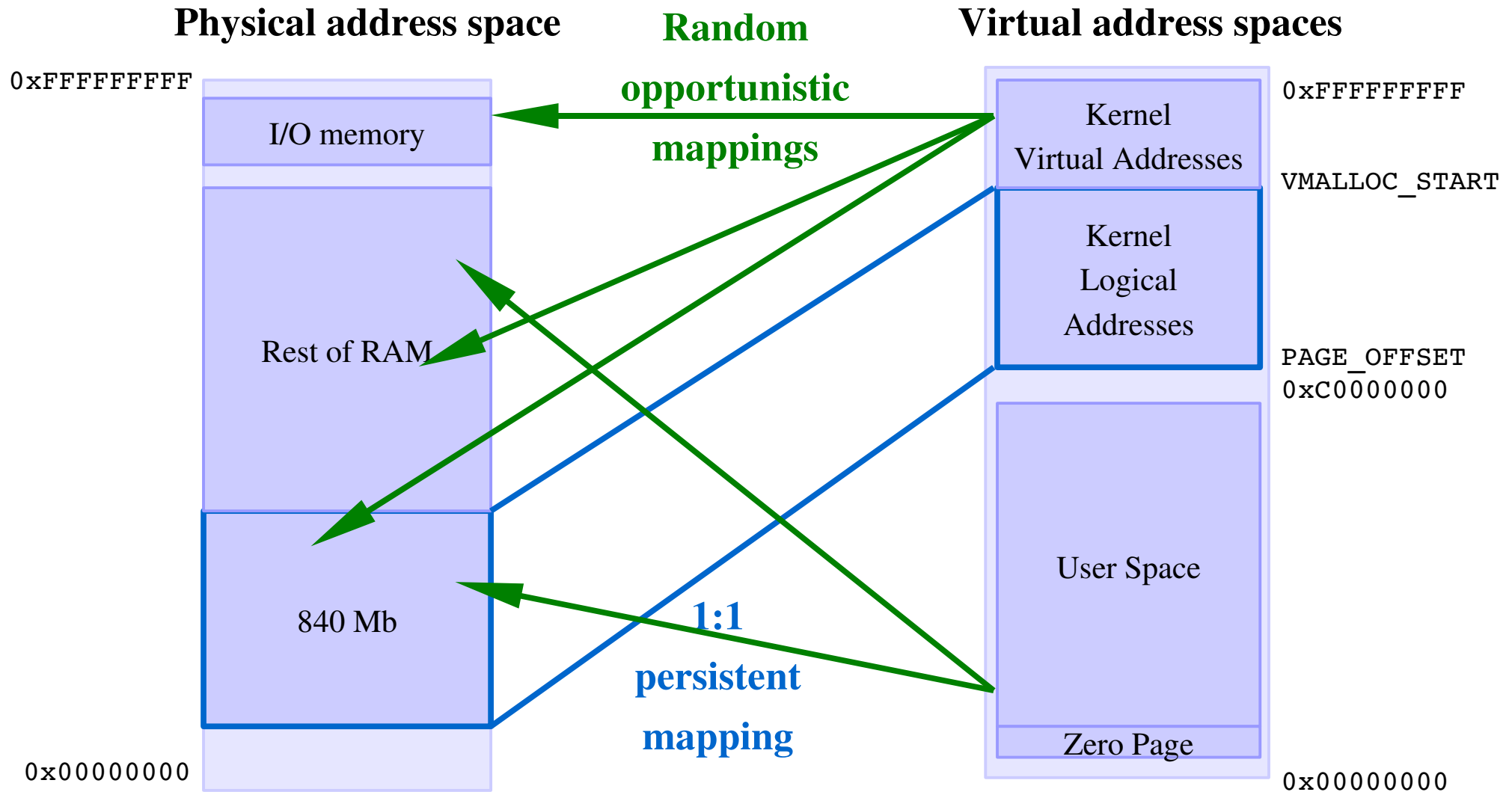


Physical address space

Virtual address spaces



3:1 Virtual Memory Map



Address Types

▶ **Physical address**

- ▶ Physical memory as seen from the CPU, without MMU¹ 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 MMU¹ translation.

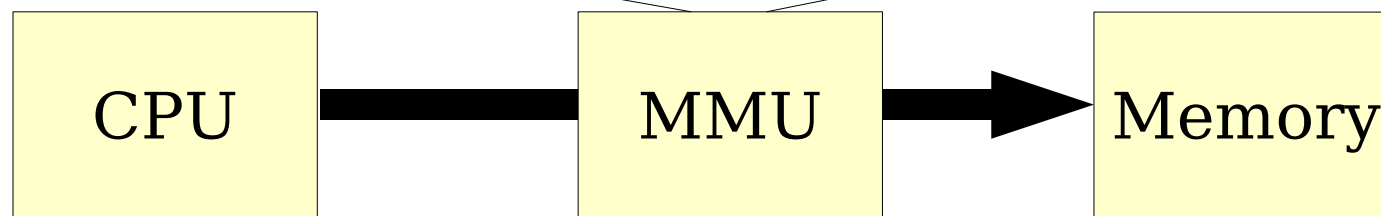
¹ **MMU**: Memory Management Unit

Address 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)`
- ▶ Where * are different bus names.

The Memory Management Unit

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



Translation Look-aside Buffers

- ▶ The tables that describes the virtual to physical translations are called page tables.
- ▶ They reside in system memory.
- ▶ The MMU caches the content of these tables in a CPU local cache dubbed the TLB, or Translation Look-aside Buffer.
- ▶ Making the changes to the page tables often requires TLB flushes.
 - ▶ For example, while context switching between two processes.
- ▶ TLB flushes are expensive, at least on some architectures.

kmalloc() and kfree()

- ▶ Basic allocators, kernel equivalents of `glibc`'s `malloc()` and `free()`.
- ▶ `#include <linux/slab.h>`
- ▶ `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).



Memory Allocation 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.

Memory Allocation 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.

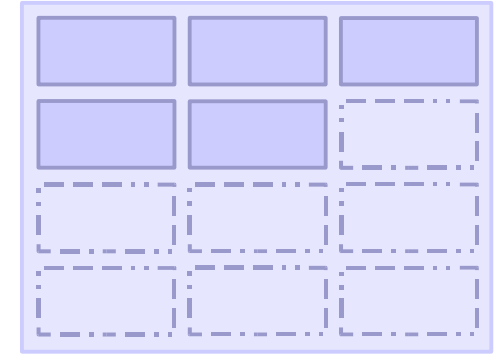
▶ Example:

GFP_KERNEL | __GFP_DMA

Slab Caches

Also called *look-aside 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.



Slab Cache API (1)

▶ `#include <linux/slab.h>`

▶ Creating a private cache:

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

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 look-aside 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);
```

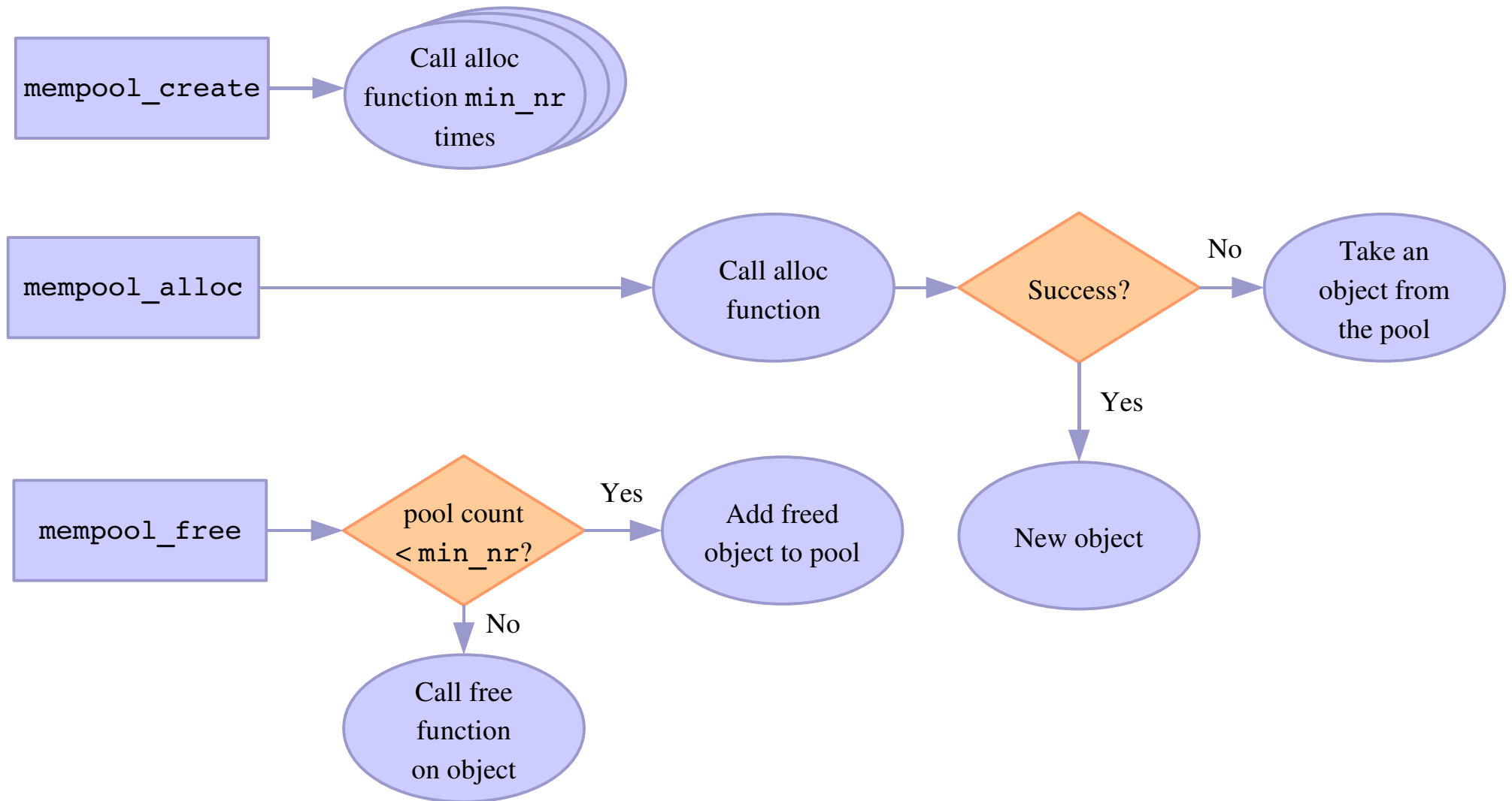
- ▶ Resizing the pool:

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

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

```
mempool_destroy(pool);
```

Memory Pool Implementation



Memory Pools Using Slab Caches

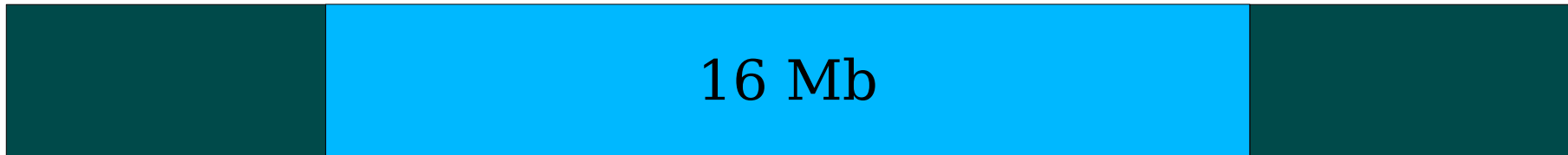
- ▶ Idea: use slab cache functions to allocate and free objects.
- ▶ The `mempool_alloc_slab` and `mempool_free_slab` functions supply a link with slab cache routines.
- ▶ So, you will find many code examples looking like:

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

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.

Buddy System 1



We need 8 Mb of memory, but don't find an exact match.

We do have a block of 16 Mb memory though.

Buddy System 2



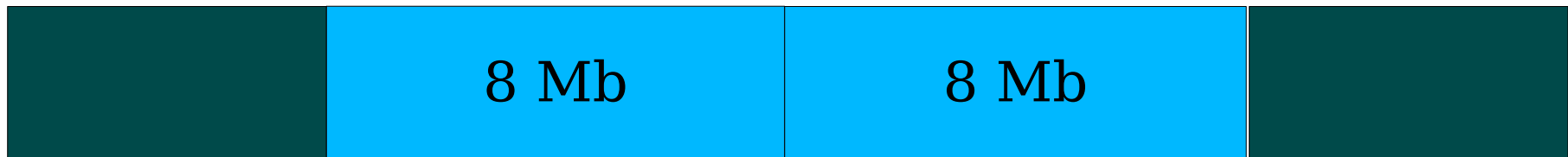
So we'll split the 16 Mb into two 8 Mb areas.

Buddy System 3



We'll use 8 Mb and keep the rest
as a free block of 8 Mb.

Buddy System 4



When the allocated memory has
been freed...

Buddy System 5



16 Mb

We can once again combine the two blocks in a single 16 Mb free block.

Because of the order of 2 allocation, it's easy to spot our “buddy”.

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`)

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.

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()` for 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`, `memcpy`,
`memmove`...

Embedded Linux Driver Development



Driver Development I/O Memory and Ports

Requesting I/O Ports

`/proc/ioprots` 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`

Reading/Writing on I/O Ports

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

bytes

```
unsigned inb(unsigned port);  
void outb(unsigned char byte, unsigned port);
```

words

```
unsigned inw(unsigned port);  
void outw(unsigned char byte, unsigned port);
```

"long" integers

```
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!

byte strings

```
void insb(unsigned port, void *addr, unsigned long count);  
void outsb(unsigned port, void *addr, unsigned long count);
```

word strings

```
void insw(unsigned port, void *addr, unsigned long count);  
void outsw(unsigned port, void *addr, unsigned long count);
```

long strings

```
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

```
00000000-0009ffff : System RAM
0009ffff-0009ffff : reserved
000a0000-000bffff : Video RAM area
000c0000-000cffff : Video ROM
000f0000-000fffff : System ROM
00100000-3ffadfff : System RAM
    00100000-0030ffff : 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!

Differences from Standard Memory

- ▶ Reads and writes on memory can be cached.
- ▶ The compiler may choose to write the value in a CPU register, and may never write it in main memory.
- ▶ The compiler may decide to optimize or reorder read and write instructions.

Avoiding I/O Access Issues

- ▶ Caching on I/O ports or memory already disabled, either by the hardware or by Linux init code.
- ▶ Memory barriers are supplied to avoid reordering:

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!

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);
```

Embedded Linux Driver Development



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!



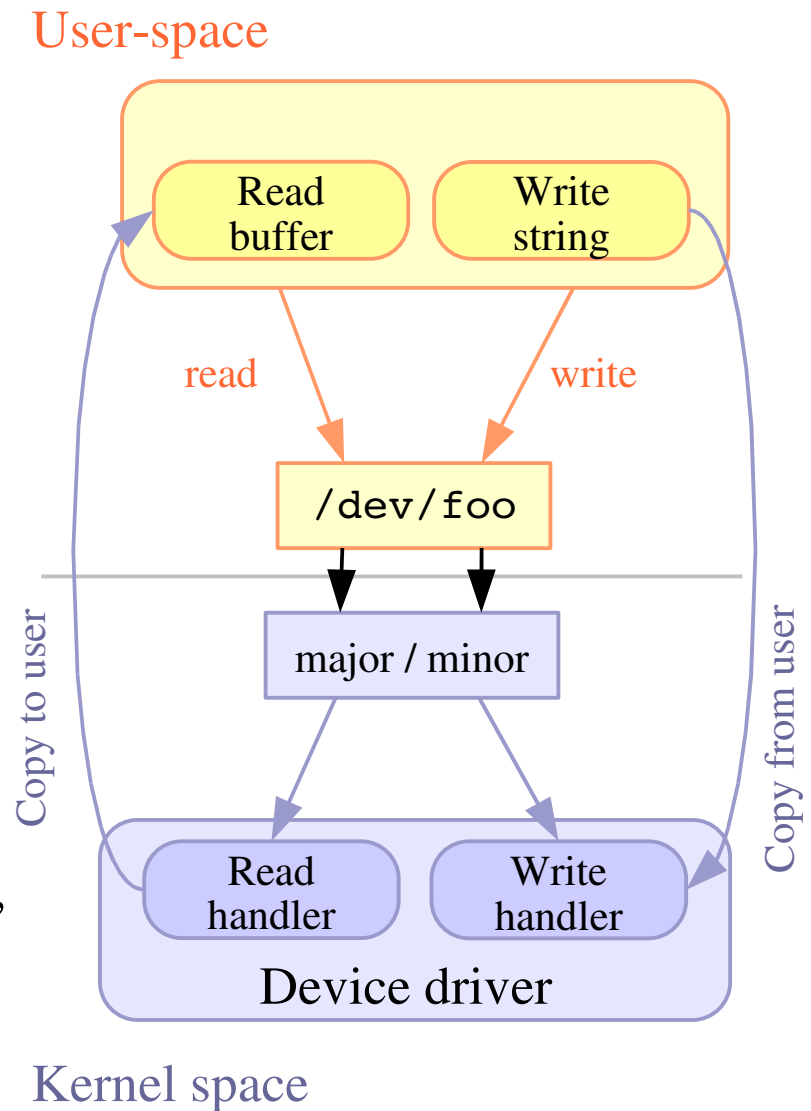
Creating a Character Driver

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`...

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

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.

Example

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

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.

Example

```
if (alloc_chrdev_region(&acme_dev, 0, acme_count, "acme")) {
    printk(KERN_ERR "Failed to allocate device number\n");
    ...
}
```

Creating Device Files

- ▶ Issue: you can no longer create `/dev` entries in advance!
You have to create them on the fly after loading the driver according to the allocated major number.
- ▶ Tip: read `/proc/devices` after module loads to find out which major number your driver got.
- ▶ Trick: the script loading the module can then use `/proc/devices`:

```
rm -f /dev/my_dev_file
insmod my_driver.ko
cat /proc/devices
mknod /dev/my_dev_file c 254 0
```

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 "*files*".

▶ `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`.

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

File Operations Definition Example

Defining a file operations structure

```
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);
```

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`

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,
};
```

Char Driver Example Summary (2)

```
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.

Kernel

- Executes the corresponding file operations

Kernel

Embedded Linux Driver Development



Driver Development mmap()

mmap() (1)

Possibility to have parts of the virtual address space of a program mapped to the contents of a file!

```
> cat /proc/1/maps (init process)
```

start	end	perm	offset	major:minor	inode	mapped file name
00771000-0077f000	r-xp	00000000	03:05	1165839	/lib/libselinux.so.1	
0077f000-00781000	rw-p	0000d000	03:05	1165839	/lib/libselinux.so.1	
0097d000-00992000	r-xp	00000000	03:05	1158767	/lib/ld-2.3.3.so	
00992000-00993000	r--p	00014000	03:05	1158767	/lib/ld-2.3.3.so	
00993000-00994000	rw-p	00015000	03:05	1158767	/lib/ld-2.3.3.so	
00996000-00aac000	r-xp	00000000	03:05	1158770	/lib/tls/libc-2.3.3.so	
00aac000-00aad000	r--p	00116000	03:05	1158770	/lib/tls/libc-2.3.3.so	
00aad000-00ab0000	rw-p	00117000	03:05	1158770	/lib/tls/libc-2.3.3.so	
00ab0000-00ab2000	rw-p	00ab0000	00:00	0		
08048000-08050000	r-xp	00000000	03:05	571452	/sbin/init (text)	
08050000-08051000	rw-p	00008000	03:05	571452	/sbin/init (data, stack)	
08b43000-08b64000	rw-p	08b43000	00:00	0		
f6fdf000-f6fe0000	rw-p	f6fdf000	00:00	0		
fef4000-ff000000	rw-p	fef4000	00:00	0		
ffffe000-ffffff00	---p	00000000	00:00	0		

mmap() (2)

Particularly useful when the file is a device file!

Allows to access device I/O memory and ports without having to go through (expensive) `read`, `write` or `ioctl` calls!

`X server` example (maps excerpt)

start	end	perm	offset	major:minor	inode	mapped file name
08047000	081be000	r-xp	00000000	03:05	310295	/usr/X11R6/bin/Xorg
081be000	081f0000	rw-p	00176000	03:05	310295	/usr/X11R6/bin/Xorg
...						
f4e08000	f4f09000	rw-s	e0000000	03:05	655295	/dev/dri/card0
f4f09000	f4f0b000	rw-s	4281a000	03:05	655295	/dev/dri/card0
f4f0b000	f6f0b000	rw-s	e8000000	03:05	652822	/dev/mem
f6f0b000	f6f8b000	rw-s	fcff0000	03:05	652822	/dev/mem

A more user friendly way to get such information: `pmap <pid>`

How to Implement mmap() - User-Space

► Open the device file

► Call the `mmap` system call (see `man mmap` for details):

```
void *mmap(  
    void *start,      /* Often 0, preferred starting address */  
    size_t length,    /* Length of the mapped area */  
    int prot ,        /* Permissions: read, write, execute */  
    int flags,        /* Options: shared mapping, private copy... */  
    int fd,           /* Open file descriptor */  
    off_t offset      /* Offset in the file */  
);
```

► Read from the return virtual address or write to it.

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;
}
```

Embedded Linux Driver Development



Driver Development Debugging



Usefulness of a Serial Port

- ▶ Most processors feature a serial port interface (usually very well supported by Linux). Just need this interface to be connected to the outside.
- ▶ Easy way of getting the first messages of an early kernel version, even before it boots. A minimum kernel with only serial port support is enough.
- ▶ Once the kernel is fixed and has completed booting, possible to access a serial console and issue commands.
- ▶ The serial port can also be used to transfer files to the target.

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 /proc or /sys

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!

kgdb Kernel Patch

<http://kgdb.linsyssoft.com/>

- ▶ The execution of the patched kernel is fully controlled by **gdb** from another machine, connected through a serial line.
- ▶ Can do almost everything, including inserting breakpoints in interrupt handlers.
- ▶ Supported architectures: **i386**, **x86_64**, **ppc** and **s390**.



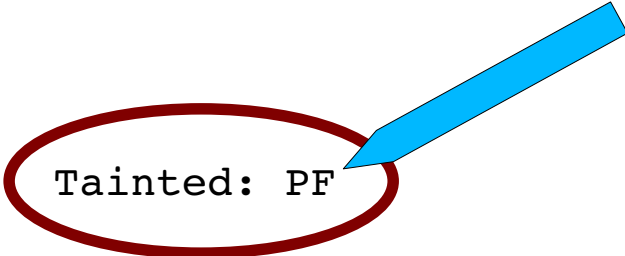
Kernel Oops

- ▶ Caused by an exception in the kernel code itself.
- ▶ The kernel issues a diagnostic message, called an Oops!. The message contains debug information, such as register content, stack trace, code dump etc.
- ▶ After the message is sent to the log and console -
 - ▶ From within a task – the kernel kills the task.
 - ▶ From interrupt – the kernel panics and may reboot automatically if given a-priori the **panic=secs** parameter.
- ▶ In 2.4, stack trace addresses can be translated via the **ksymoops** utility. In 2.6, with the CONFIG_KALLSYMS option on, the kernel does translation internally.

Kernel Oops Example

```
Unable to handle kernel NULL pointer dereference at virtual address
00000014
printing eip: d8d4d545
*pde = 00000000
Oops: 0000
CPU:      0
EIP:      0010:[<d8d4d545>]  Tainted: PF
EFLAGS: 00013286
eax: c8b078c0   ebx: c8b078c0   ecx: 00000019   edx: c8b078c0
esi: 00000000   edi: 00000000   ebp: bffff75c   esp: c403bf84
ds: 0018   es: 0018   ss: 0018
Process vmware (pid: 5194, stackpage=c403b000)
Stack: c8b078c0 00000000 bffff35c 00000000 c0136d64 c403bfa4 c8b078c0
c0130185
Call Trace: [sys_stat64+100/112] [filp_close+53/112]
[sys_close+67/96] [system_call+51/56]

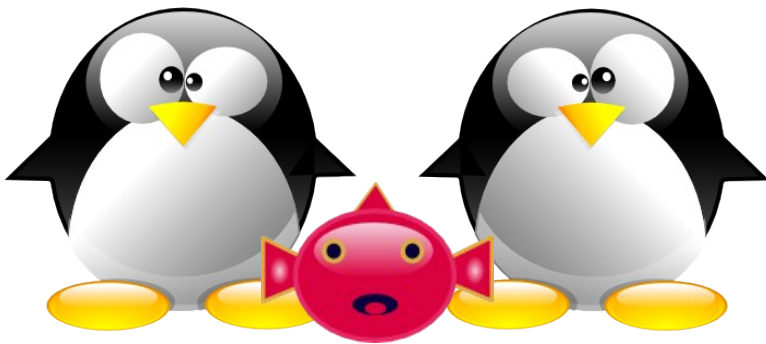
Code: f6 46 14 01 74 1c 83 c4 f4 8b 06 50 e8 da 62 43 e7 83 c4 f8
```



Embedded Linux Driver Development



Driver Development Concurrent Access to Resources



Sources of Concurrency Issues

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

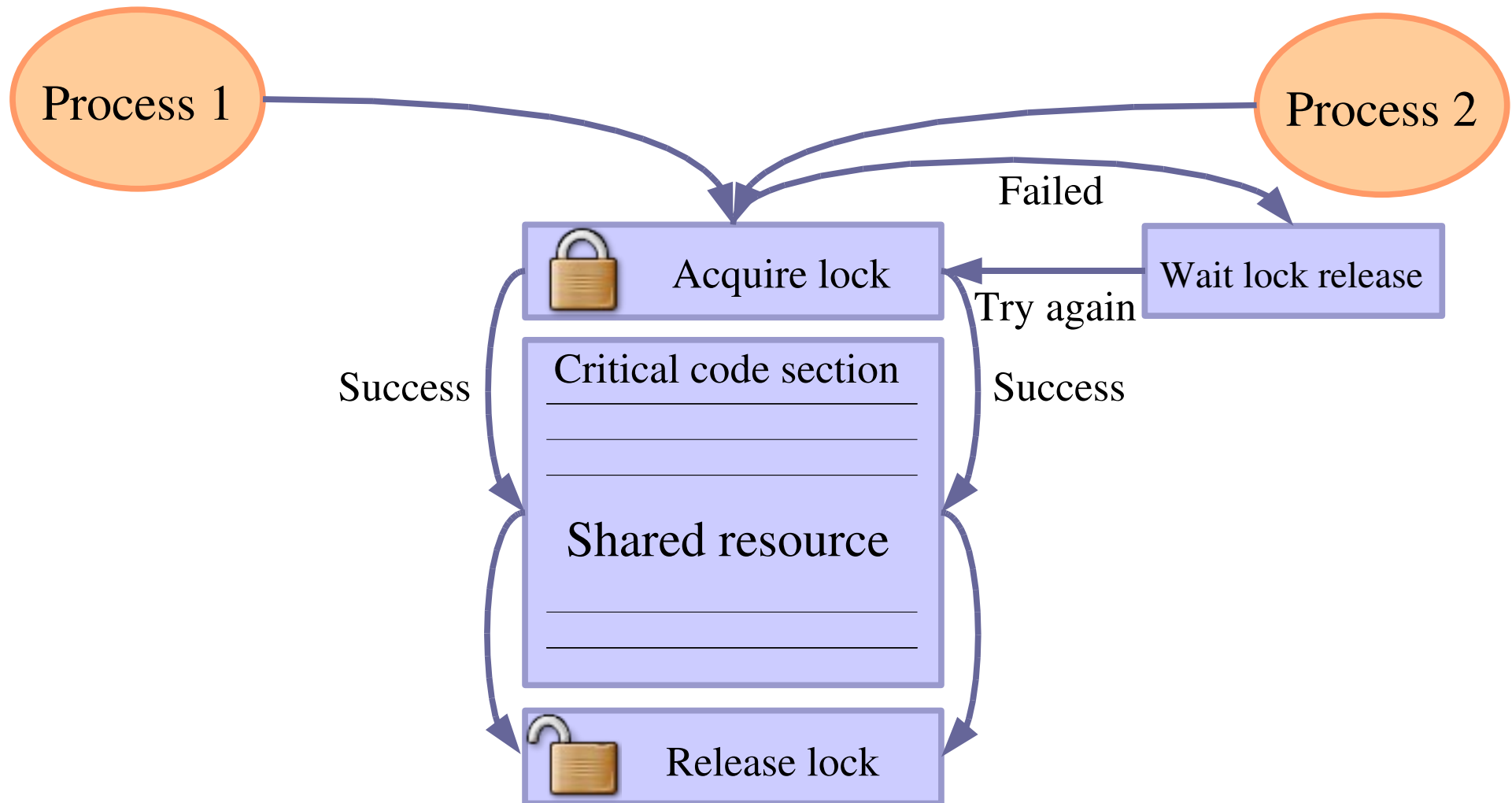
- ▶ Several user-space programs accessing the same device data or hardware. Several kernel processes could execute the same code on behalf of user processes running in parallel.
- ▶ Multi processor (SMP): the same driver code can be running on another processor. This can also happen with single CPUs 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.

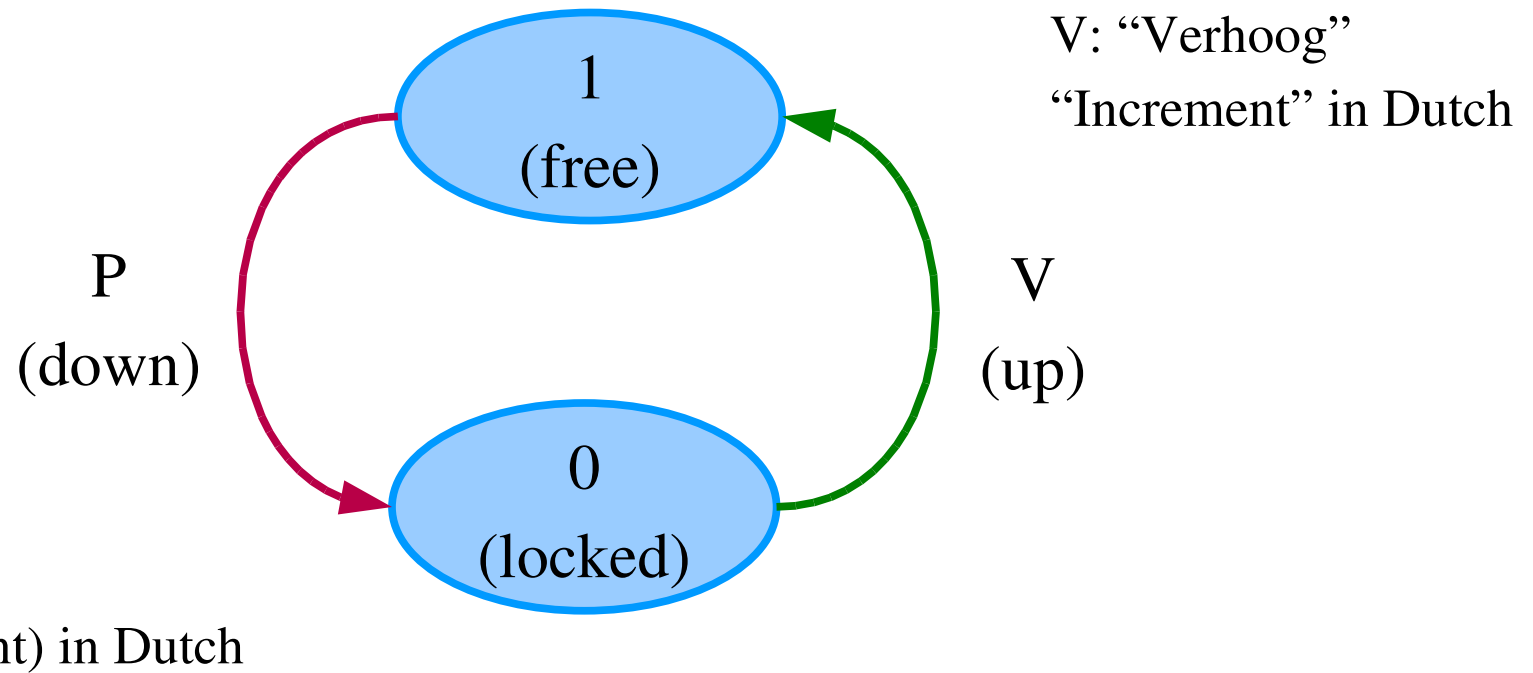
See Rusty Russell's Unreliable Guide To Locking
[Documentation/DocBook/kernel-locking/](#)
in the kernel sources.

Concurrency Protection With Semaphores



Kernel Semaphores

Also called “mutexes” (Mutual Exclusion)



Initializing a Semaphore

► Statically

```
DECLARE_MUTEX(name);  
DECLARE_MUTEX_LOCKED(name);
```

► Dynamically

```
void init_MUTEX(struct semaphore *sem);  
void init_MUTEX_LOCKED(struct semaphore *sem);
```

Locking and Unlocking Semaphores

► `void down(struct semaphore *sem);`

Decrements the semaphore if set to 1, waits otherwise.

Caution: can't be interrupted, causing processes you cannot kill!

► `int down_interruptible(struct semaphore *sem);`

Same, but can be interrupted. If interrupted, returns a non zero value and doesn't hold the semaphore. Test the return value!!!

► `int down_trylock(struct semaphore *sem);`

Never waits. Returns a non zero value if the semaphore is not available.

► `void up(struct semaphore *sem);`

Releases the semaphore. Make sure you do it as soon as possible!

Reader/Writer Semaphores

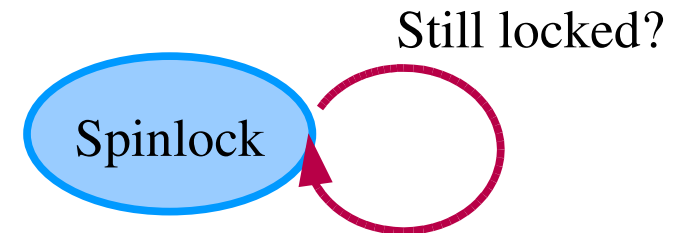
Allow shared access to unlimited readers, or to only one writer. Writers get priority.

```
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 can't sleep (critical sections, interrupt handlers... Be very careful not to call functions which can sleep!
- ▶ Intended for multiprocessor systems
- ▶ Spinlocks are not interruptible, don't sleep and keep spinning in a loop until the lock is available.
- ▶ Spinlocks cause kernel preemption to be disabled on the CPU executing them, similar to VxWorks's task lock.
- ▶ May require interrupts to be disabled too.



Initializing Spinlocks

▶ Static

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
```

▶ Dynamic

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

Using Spinlocks

```
void spin_[un]lock(spin_lock_t *lock);
```

```
void spin_[un]lock_irq[save|restore](spin_lock_t  
    *lock, unsigned long flags);
```

Disables IRQs on the local CPU

```
void spin_[un]lock_irq(spin_lock_t *lock);
```

Disables IRQs without saving flags. When you're sure that nobody already disabled interrupts.

```
void spin_[un]lock_bh(spin_lock_t *lock);
```

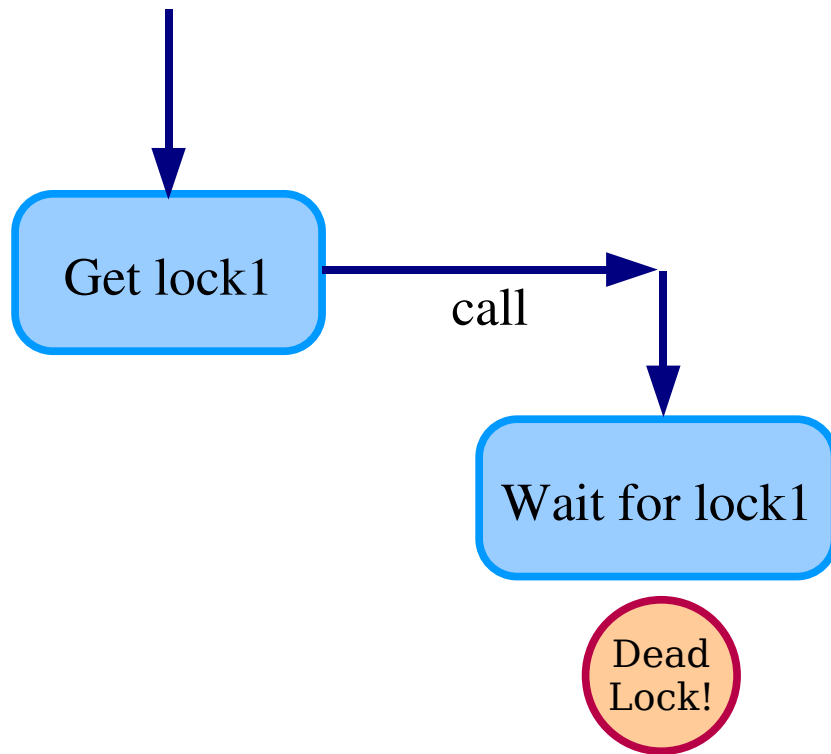
Disables software interrupts, but not hardware ones

Note that reader/writer spinlocks also exist.

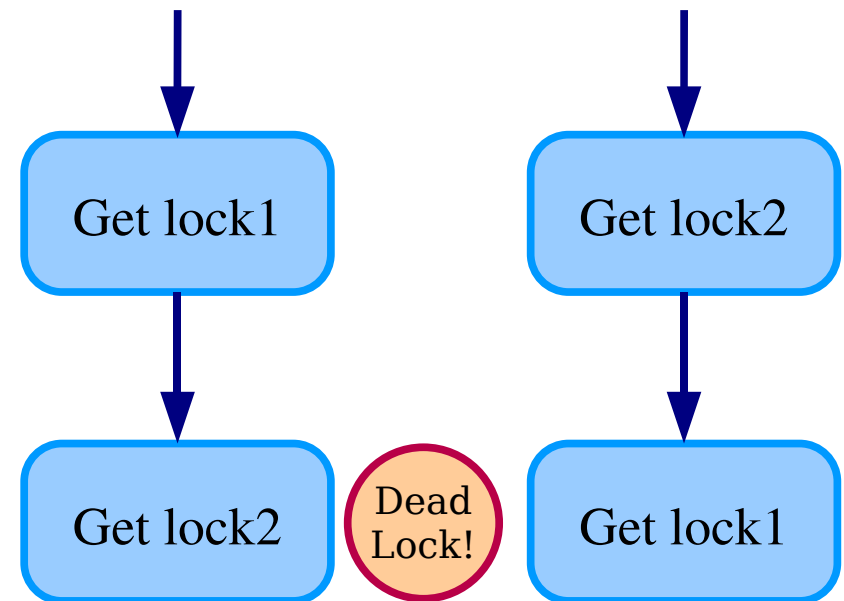
Deadlock Situations

They can lock up your system. Make sure they never happen!

Don't call a function that can try to get access to the same lock



Holding multiple locks is risky!



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_t *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(...);
```

Embedded Linux Driver Development



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.

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

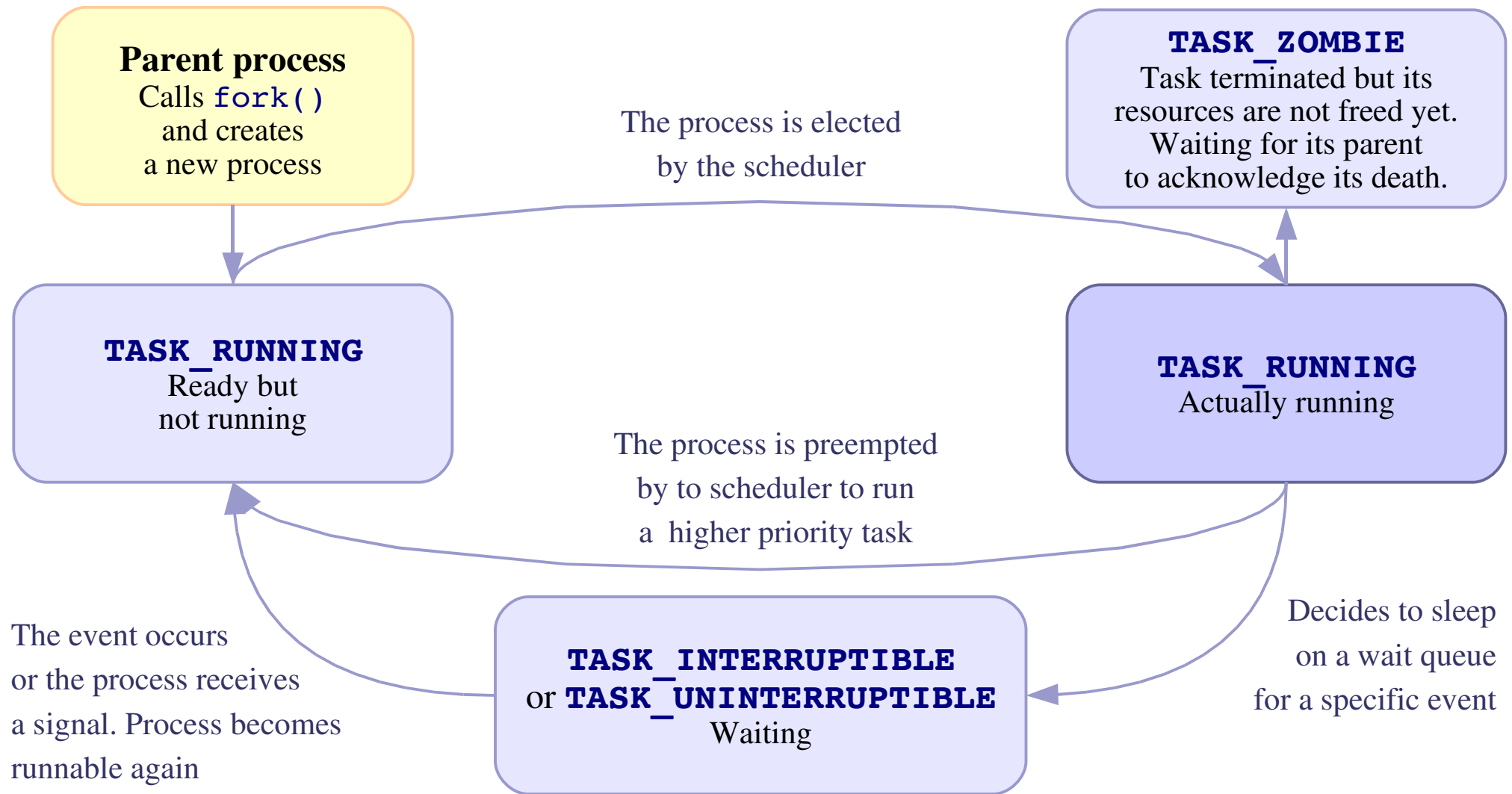
Task Identifiers

- ▶ Each `task_struct` has the following identities:
 - ▶ **PID** Globally unique ID. Different one for each thread.
 - ▶ **TGID** Thread Group ID. Returned to user-space as `getpid()`
 - ▶ Shared by all threads of a process.
 - ▶ For single threaded process == PID.
 - ▶ **PGID** Process Group ID (Posix.1).
 - ▶ **SID** Session ID (Posix.1).

Current Task

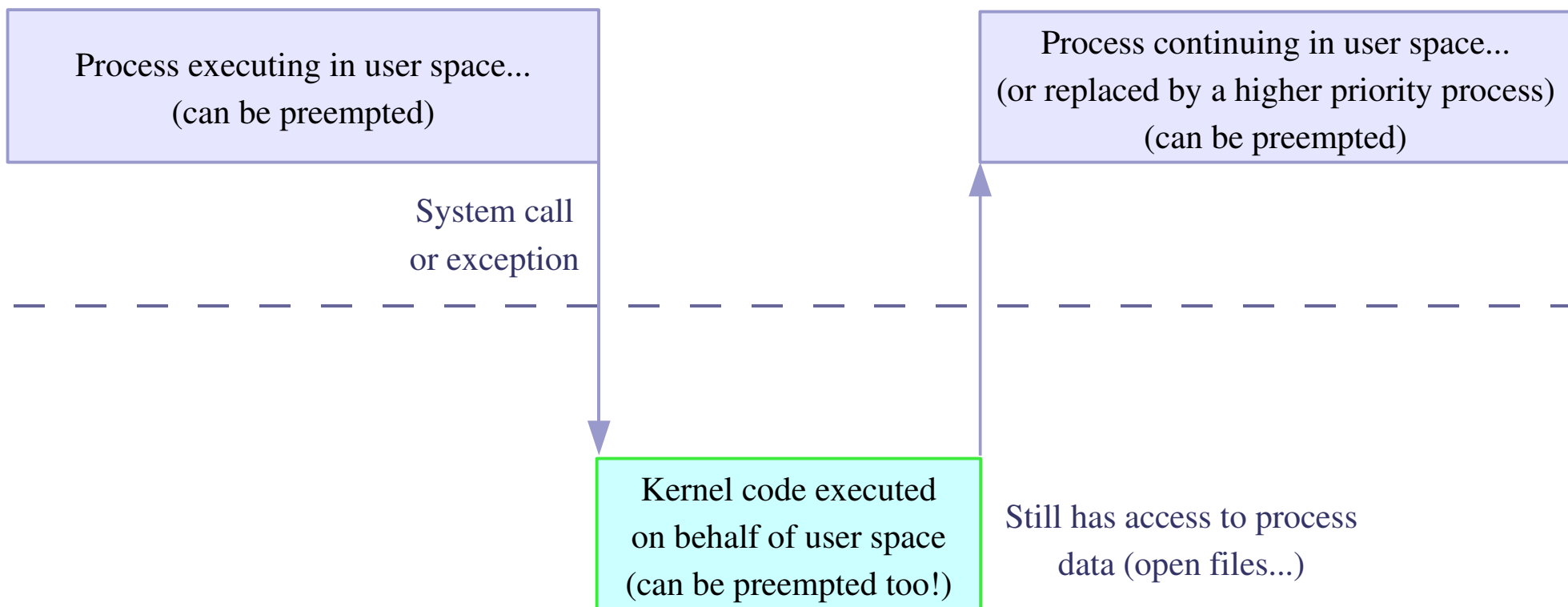
- ▶ `current` points to the current process `task_struct`
 - ▶ When applicable – not valid in interrupt context.
- ▶ Current is a macro that appears to the programmer as a magical global variable which updated each context switch.
 - ▶ Real value is either in register or computed from start of stack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

A Process Life



Process Context

User-space programs and system calls are scheduled together:



Kernel Threads

- ▶ The kernel does not only react to user-space programs (system calls, exceptions) or hardware events (interrupts). It also runs its own “processes”.
- ▶ Kernel threads are standard processes scheduled and preempted in the same manner as any other processes (you can view them with `top` or `ps`!). They just have no special address space and usually run forever.
- ▶ Kernel thread examples:
 - ▶ `pdfflush`: regularly flushes “dirty” memory pages to disk (file changes not committed to disk yet).
 - ▶ `ksoftirqd`: manages soft irqs.

Timer Frequency

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

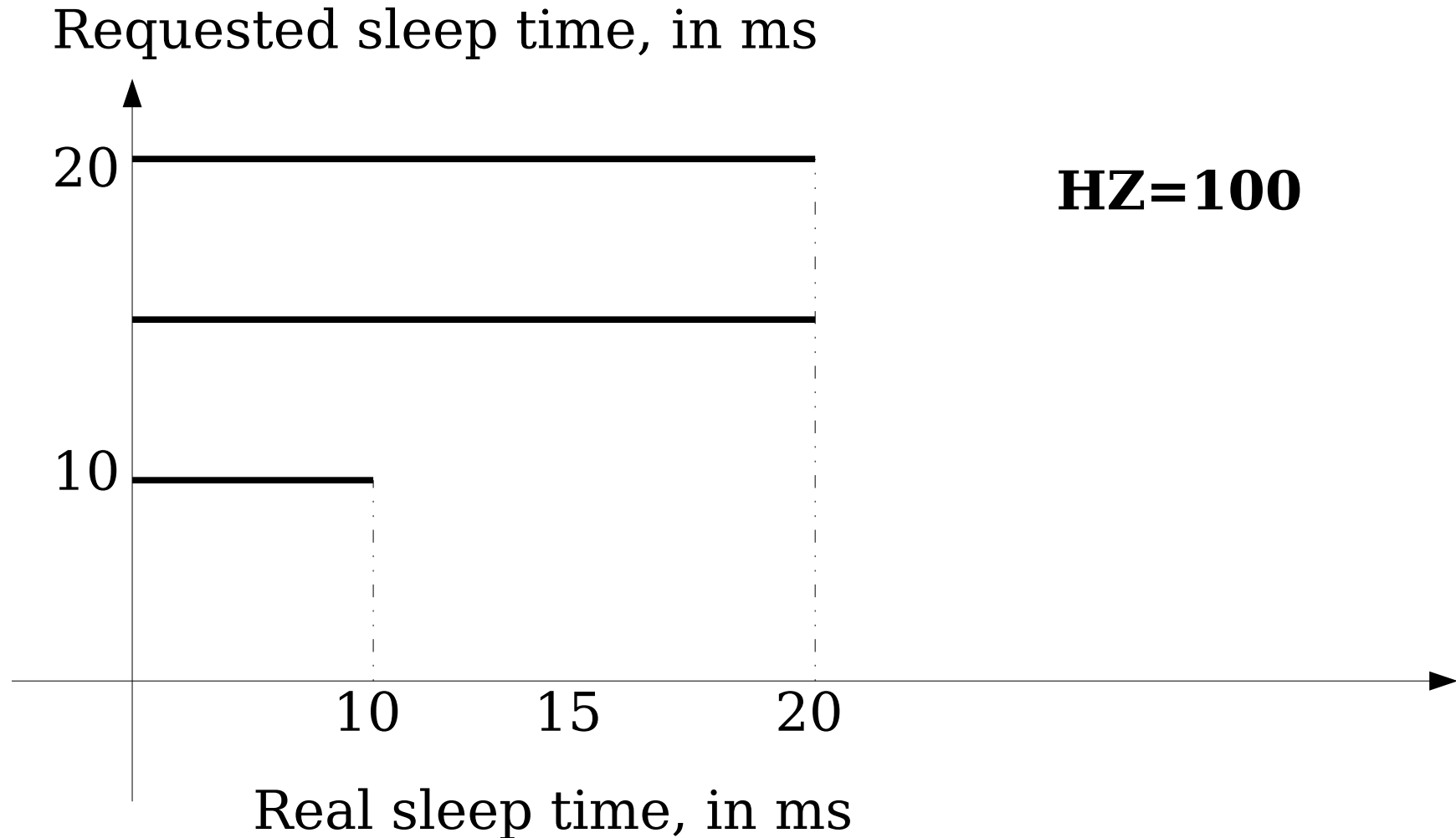
- ▶ `HZ` is now configurable (in `Processor type and features`):
`100` (`i386` default), `250` or `1000`.

Supported on `i386`, `ia64`, `ppc`, `ppc64`, `sparc64`, `x86_64`

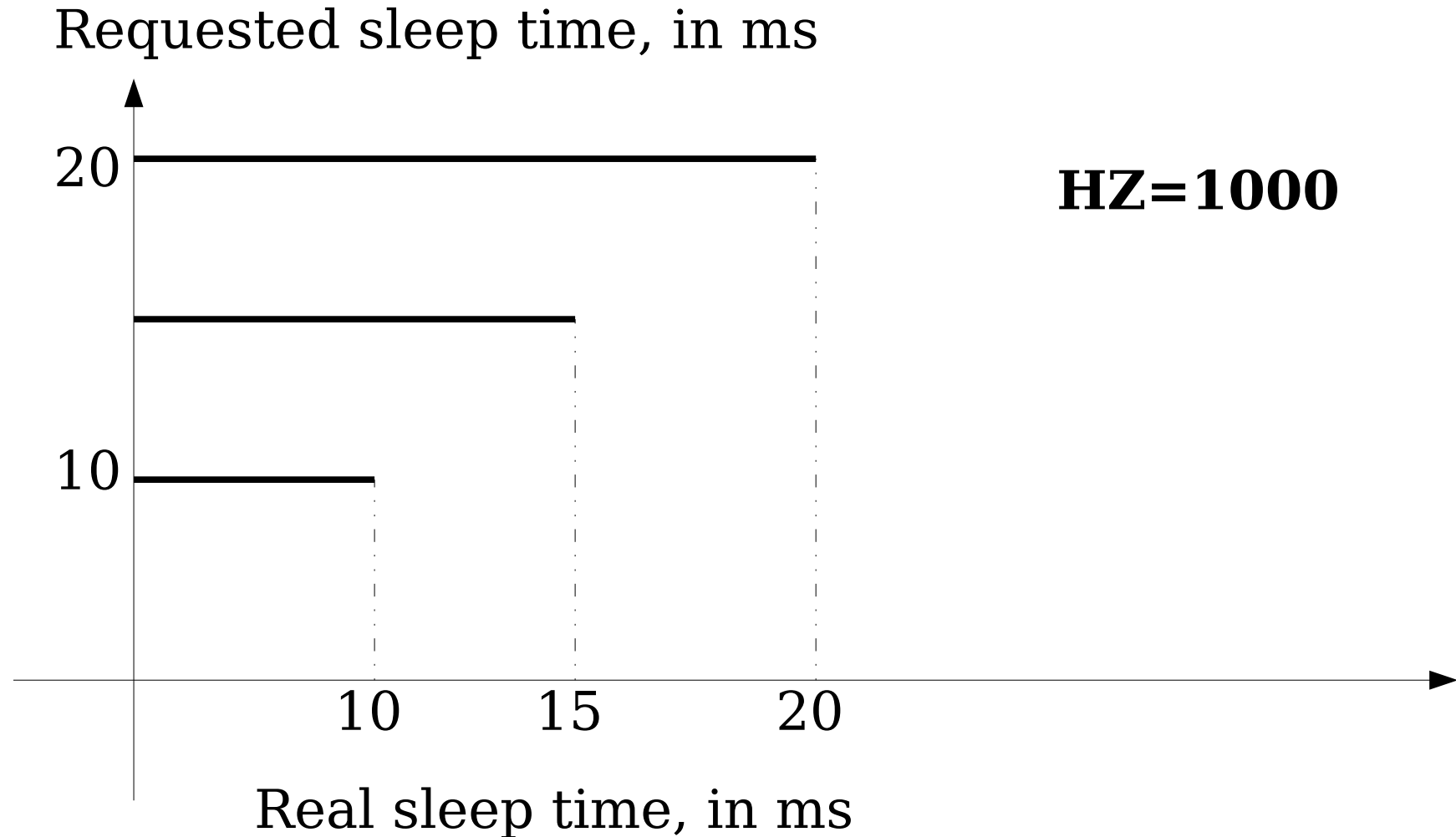
See `kernel/Kconfig.hz`.

- ▶ Compromise between system responsiveness and global throughput.
- ▶ Caution: not any value can be used. Constraints apply!

The Effect of Timer Frequency



The Effect of Timer Frequency cont.




High-Res Timers and Tickless Kernel (2.6.20)

- ▶ 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.
- ▶ Both require outside kernel patches.

O(1) Scheduler

- ▶ The O(1) scheduler was the algorithm used from the beginning of the 2.6 kernel up till (and including) 2.6.22.
- ▶ The kernel maintains 2 priority arrays: the *active* and the *expired* array.
- ▶ Each array contains 140 entries (100 real-time priorities + 40 regular ones), 1 for each priority, each containing a list of processes with the same priority.
- ▶ The arrays are implemented in a way that makes it possible to pick a process with the highest priority in **constant** time (regardless of the number of running processes).

Choosing and Expiring Processes

- 
- ▶ The scheduler finds the highest process priority
 - ▶ It executes the first process in the priority queue for this priority.
 - ▶ If the process is a non real time one, once the process has exhausted its time slice, it is moved to the expired array. Otherwise, it's run until it yields the CPU[*].
 - ▶ The scheduler gets back to selecting another process with the highest priority available, and so on...
 - ▶ Once the active array is empty, the 2 arrays are swapped!
Again, everything is done in constant time!

[*] SCHED_RR tasks will go to the back of the list for a specific priority when their time slice expires.

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.

Time Slices

The scheduler also prioritizes high priority (but non-real-time) processes by giving them a bigger time slice.

- ▶ Initial process time slice: parent's time slice split in 2 (otherwise process would cheat by forking).
- ▶ Minimum priority: 5 ms or 1 jiffy (whichever is larger)
- ▶ Default priority in jiffies: 100 ms
- ▶ Maximum priority: 800 ms

Note: actually depends on [HZ](#).

See [kernel/sched.c](#) for details.

Dynamic Priorities

Only applies to regular processes.

- ▶ For a better user experience, the Linux scheduler boosts 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.

Problems With the O(1) Scheduler

- ▶ The O(1) scheduler uses heuristics for deciding when to move a process to the expired queue.
- ▶ It is sometimes wrong, and is also subject to a number of attacks which can “fool” it and get more CPU cycles (fifty.c, thud.c, chew.c, ring-test.c, massive_intr.c).
- ▶ Also as a result of the heuristics, the code is complicated – hard to understand and to maintain.
- ▶ Therefore, the so-called Completely Fair Scheduler (CFS) was introduced by Red Hat's Ingo Molnar in kernel 2.6.23. For more information see the annexes.

Embedded Linux Driver Development



Driver Development Sleeping

How to Sleep (1)

Sleeping is needed when a user process is waiting for data which are not ready yet. The process then puts itself in a waiting queue.

▶ `#include <linux/wait.h>`

▶ Static queue declaration:

```
DECLARE_WAIT_QUEUE_HEAD(module_queue);
```

▶ Dynamic queue declaration:

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


How to Sleep (2)

There are several ways to make a kernel process sleep:

▶ `wait_event(queue, condition);`

Sleeps until the given boolean expression is true.

Caution: can't be interrupted (i.e. by killing the client process in user-space)

▶ `wait_event_interruptible(queue, condition);`

Can be interrupted

▶ `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.

Waking Up!

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

▶ `wake_up(&queue);`

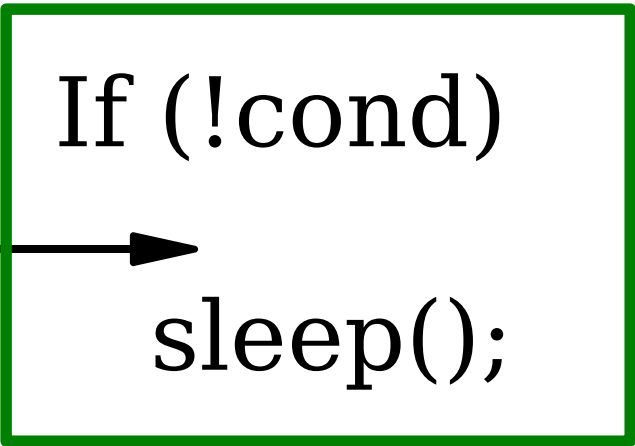
Wakes up all the waiting processes on the given queue

▶ `wake_up_interruptible(&queue);`

Does the same job. Usually called when processes waited using `wait_event_interruptible()`.

Atomically Going to Sleep

We must have a **queue** so we can put the task on it before checking the condition to avoid race.



```
If (!cond)
sleep();
```

We must have **cond** to support the event happening before going to sleep.

This is a naïve implementation of *wait_for_event()* used to explain why we need both **queue** and **cond**.

Embedded Linux Driver Development



Driver Development Interrupt Management

Need for Interrupts

- ▶ Internal processor interrupts used by the processor, for example for multi-task scheduling.
- ▶ External interrupts needed because most internal and external devices are slower than the processor. Better not keep the processor waiting for input data to be ready or data to be output. When the device is ready again, it sends an interrupt to get the processor attention again.

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(
    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);
```

Registering an Interrupt Handler (2)

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

▶ `SA_INTERRUPT`

"Quick" interrupt handler. Run with all interrupts disabled on the current CPU. Shouldn't need to be used except in specific cases (such as timer interrupts)

▶ `SA_SHIRQ`

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.

▶ `SA_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!

When to Register the Handler

- ▶ Either at driver initialization time:
consumes lots of IRQ channels!
- ▶ Or at device open time (first call to the **open** file operation):
better for saving free IRQ channels.

Need to count the number of times the device is opened, to be able to free the IRQ channel when the device is no longer in use.

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
12:         5466       XT-PIC  i8042
14:        121043      XT-PIC  ide0
15:        200888      XT-PIC  ide1
NMI:              0      # Non Maskable Interrupts
ERR:              0
```

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 */  
    void *dev_id, /* Pointer used to keep track of the  
                  corresponding device. Useful  
                  when several devices are  
                  managed by the same module */  
    struct pt_regs *regs /* CPU register snapshot, rarely  
                          needed */  
);
```

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.

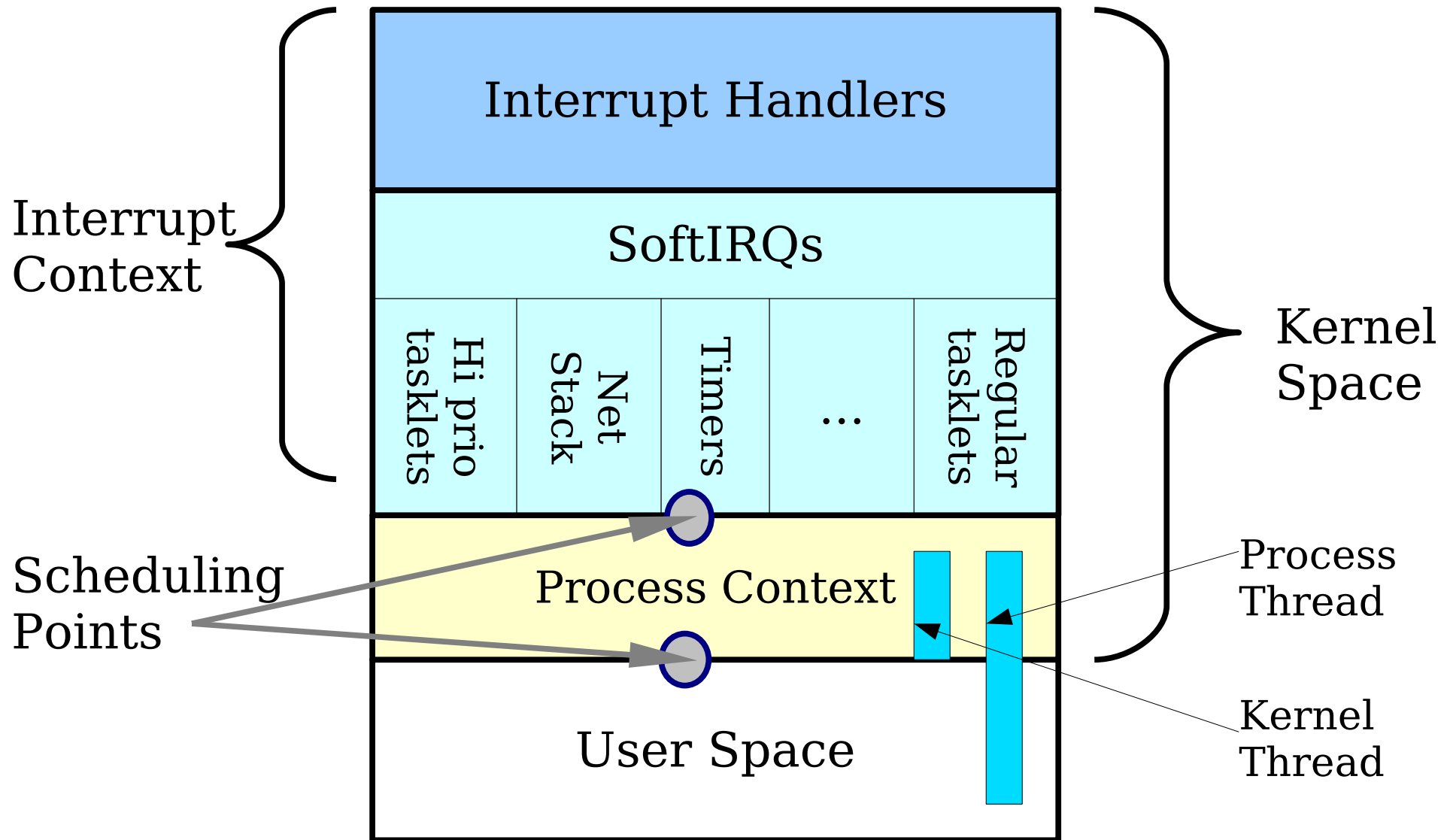
Top-Half and Bottom-Half Processing (1)

- ▶ Top-half and bottom-half is a logical paradigm to divide the tasks that needs to be performed when an interrupt occurs:
- ▶ **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 handling job in a different context. Handles data, and then wakes up any waiting user process.
- ▶ Can be implemented by *tasklets*, *timers* or *work queues* or *user-space tasks*.

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

Linux Contexts



Handling Floods

- ▶ Normally, pending softirqs 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**.

Tasklets

- ▶ Are run from softirqs (normal or low-latency)
- ▶ Each tasklet runs only on a single CPU (serialization)
- ▶ You can initialize a tasklet via:

```
init tasklet_init(struct tasklet_struct *t
                  void (*func)(unsigned long),
                  unsigned long data));
```

- ▶ Or declare the tasklet in the module source file:

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

Tasklet Scheduling and Killing

- ▶ Scheduling a tasklet in the top half part (interrupt handler):
 - ▶ For regular tasklets:
`tasklet_schedule(&module_tasklet);`
 - ▶ Or for low latency tasklets (runs first):
`tasklet_hi_schedule();`
 - ▶ If this tasklet was already scheduled – it is run only once.
 - ▶ If this tasklet was already running – it is rescheduled for later.
- ▶ On module exit, the tasklet should be killed:
 - ▶ `tasklet_kill(&module_tasklet);`

Tasklet Masking

- ▶ Tasklets may be temporarily disabled/enabled:
 - ▶ `tasklet_enable(&module_tasklet);`
 - ▶ `tasklet_disable(&module_tasklet);`
 - ▶ `tasklet_hi_enable(&module_tasklet);`
 - ▶ `tasklet_disable_nosync(&module_tasklet);`

Timers

- ▶ Run via softirq like tasklets...
- ▶ But at a specific time.
- ▶ 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);`

Work Queues (2.6 only)

- ▶ 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

▶ Work is delivered to a work queue via:

▶ **DECLARE_WORK**(work, func, data);

▶ **INIT_WORK**(work, func, data);

▶ **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);

▶ **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 (this includes 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 device.

▶ `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/Work Queue

- ▶ Process the data.
- ▶ Wake up processes waiting for the data.

Device driver

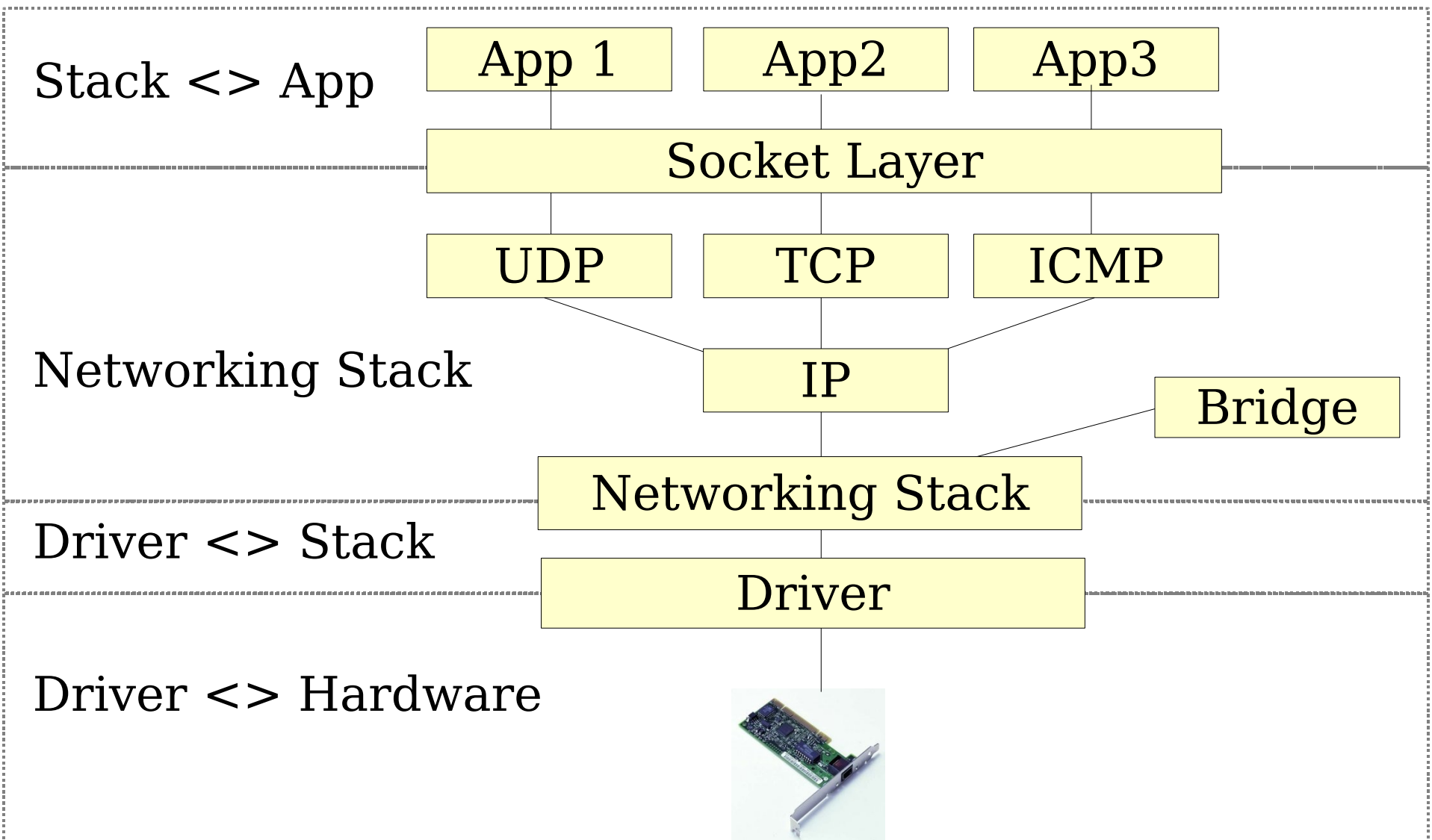
- ▶ When the device is no longer opened by any process, unregister the interrupt handler.

Embedded Linux Driver Development

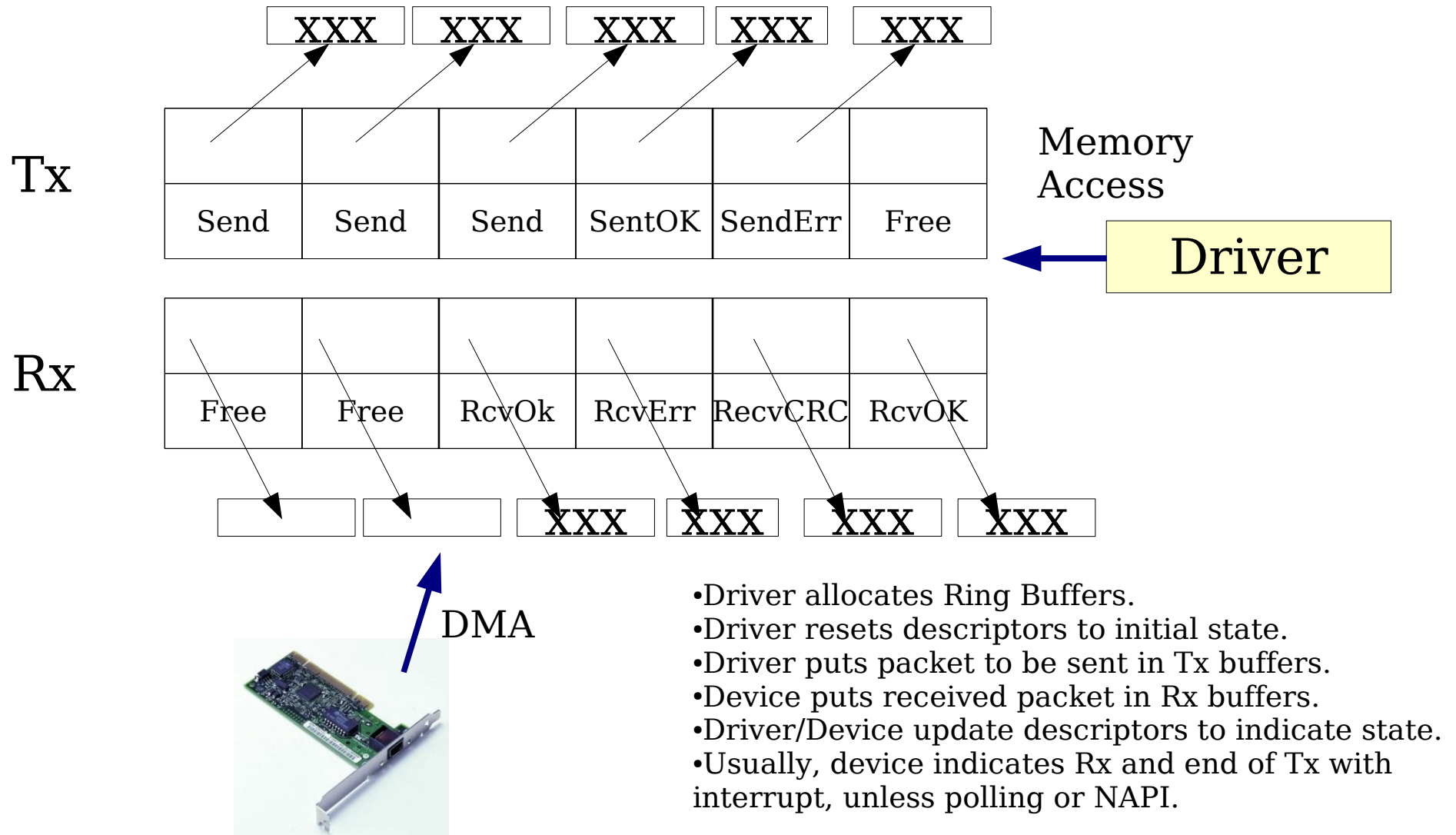


The Network Subsystem and Network Device Drivers

A View of the Linux Networking Subsystem



Network Device Driver Hardware Interface

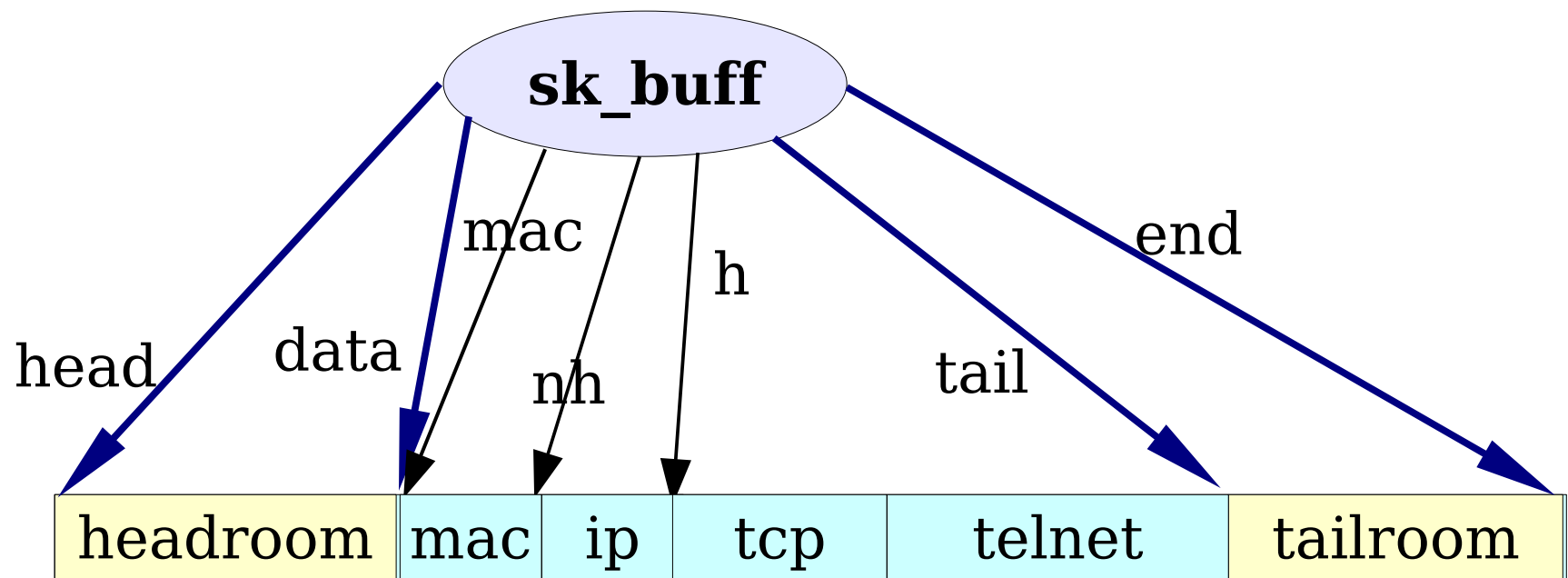


Socket Buffers

- ▶ We need to manipulate packets through the network stack.
- ▶ This manipulation involves efficiently:
 - ▶ Adding protocol headers/trailers down the stack.
 - ▶ Removing protocol headers/trailers up the stack.
- ▶ Packets can be chained together.
- ▶ Each protocol should have convenient access to header fields.
- ▶ To do all this the kernel provides the *sk_buff* structure.

struct sk_buff

- ▶ An *sk_buff* represents a single packet.
- ▶ This structure is passed through the protocol stack.
- ▶ It holds pointers to a buffer with the packet data:



sk_buff Manipulation (1)

- ▶ Manipulate an *sk_buff*:

```
unsigned char *skb_push(struct sk_buff *skb,  
    unsigned int len);
```

- ▶ data -= len

```
unsigned char *skb_pull(struct sk_buff *skb,  
    unsigned int len);
```

- ▶ data += len

sk_buff Manipulation (2)

- ▶ Manipulate an *sk_buff*:

- ▶ `int skb_headroom(const struct sk_buff *skb);`

- ▶ data - head

- ▶ `int skb_tailroom(const struct sk_buff *skb);`

- ▶ end - tail

- ▶ `int skb_reserve(const struct sk_buff *skb,
unsigned int len);`

- ▶ tail = (data += len)

sk_buff Allocation

- ▶ Low level allocation is done via:

```
struct sk_buff *alloc_skb(unsigned int  
    size, int gfp_mask);
```

- ▶ But it is better to use the wrapper:

```
struct sk_buff *dev_alloc_skb(unsigned int  
    size);
```

- ▶ Which reserves some space for optimization.

sk_buff Allocation Example

- ▶ Immediately after allocation, we should reserve the needed headroom:

```
struct sk_buff *skb;  
skb = dev_alloc_skb(1500);  
if (unlikely(!skb))  
    break;  
/* Mark as being used by this device */  
skb->dev = dev;  
/* Align IP on 16 byte boundaries */  
skb_reserve(skb, 2);
```

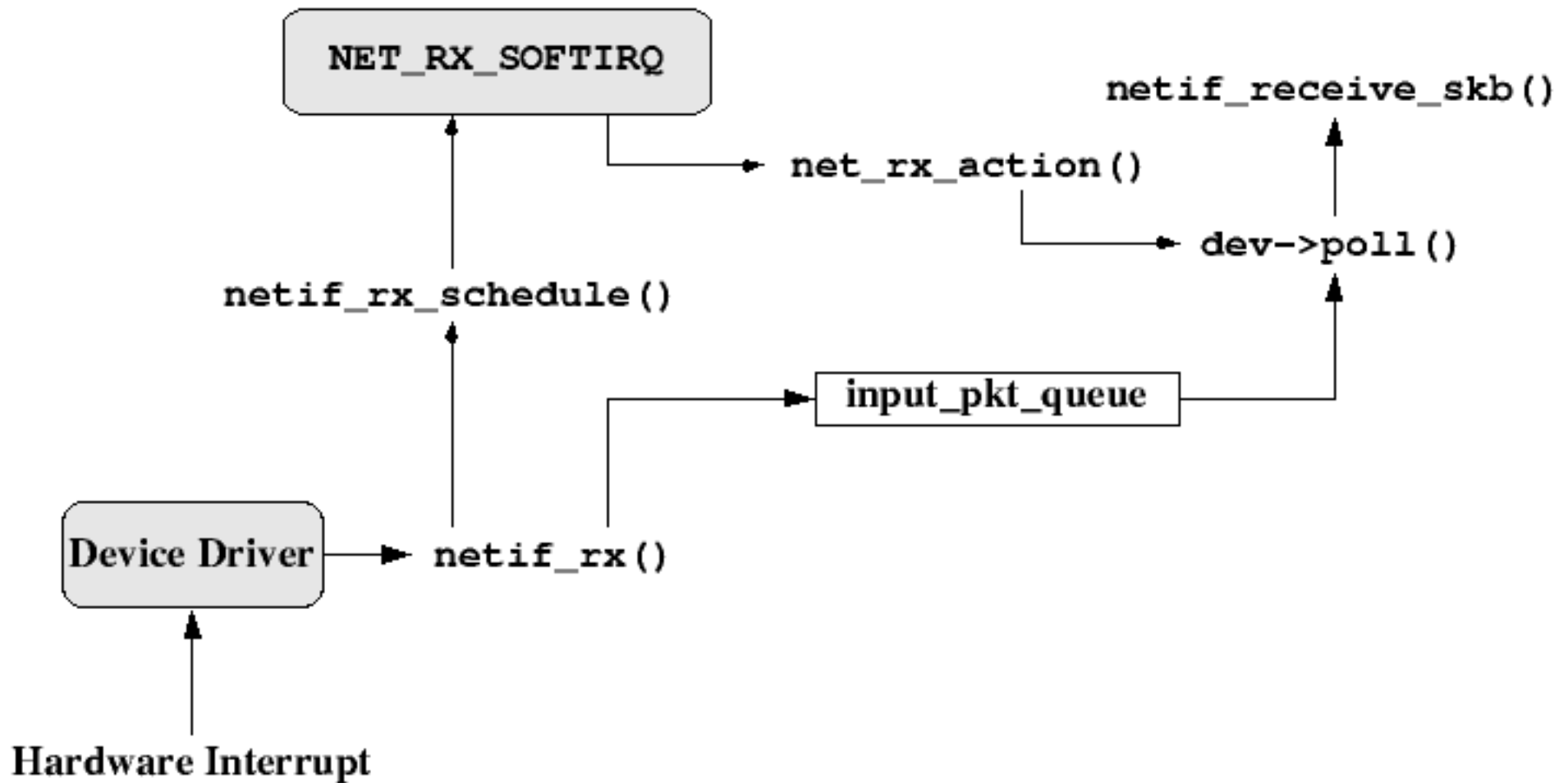
Softnet

- ▶ Was introduced in kernel 2.4.x.
- ▶ Parallelize packet handling on SMP machines.
- ▶ Packet transmit/receive is handled via two softirqs:
 - ▶ **NET_TX_SOFTIRQ** feeds packets from the network stack to the driver.
 - ▶ **NET_RX_SOFTIRQ** feeds packets from the driver to the network stack.
 - ▶ Like any other softirq, these are called on return from interrupt or via the low priority ksoftirqd kernel thread.
- ▶ Transmit/receive queues are stored in per-CPU *softnet_data*.

Packet Reception

- ▶ The driver:
 - ▶ Allocates an *skb*.
 - ▶ sets up a descriptor in the ring buffers for the hardware.
- ▶ The driver Rx interrupt handler calls *netif_rx(skb)*.
- ▶ *netif_rx(skb)*
 - ▶ Deposits the *sk_buff* in the per-CPU input queue.
 - ▶ Marks the NET_RX_SOFTIRQ to run.
- ▶ Later, *net_rx_action()* is called by NET_RX_SOFTIEQ, which calls the driver's *poll()* method to feed the packet up.
 - ▶ Normally *poll()* is set to *process_backlog()* by *net_dev_init()*.

Packet Reception Overview



Packet Transmission

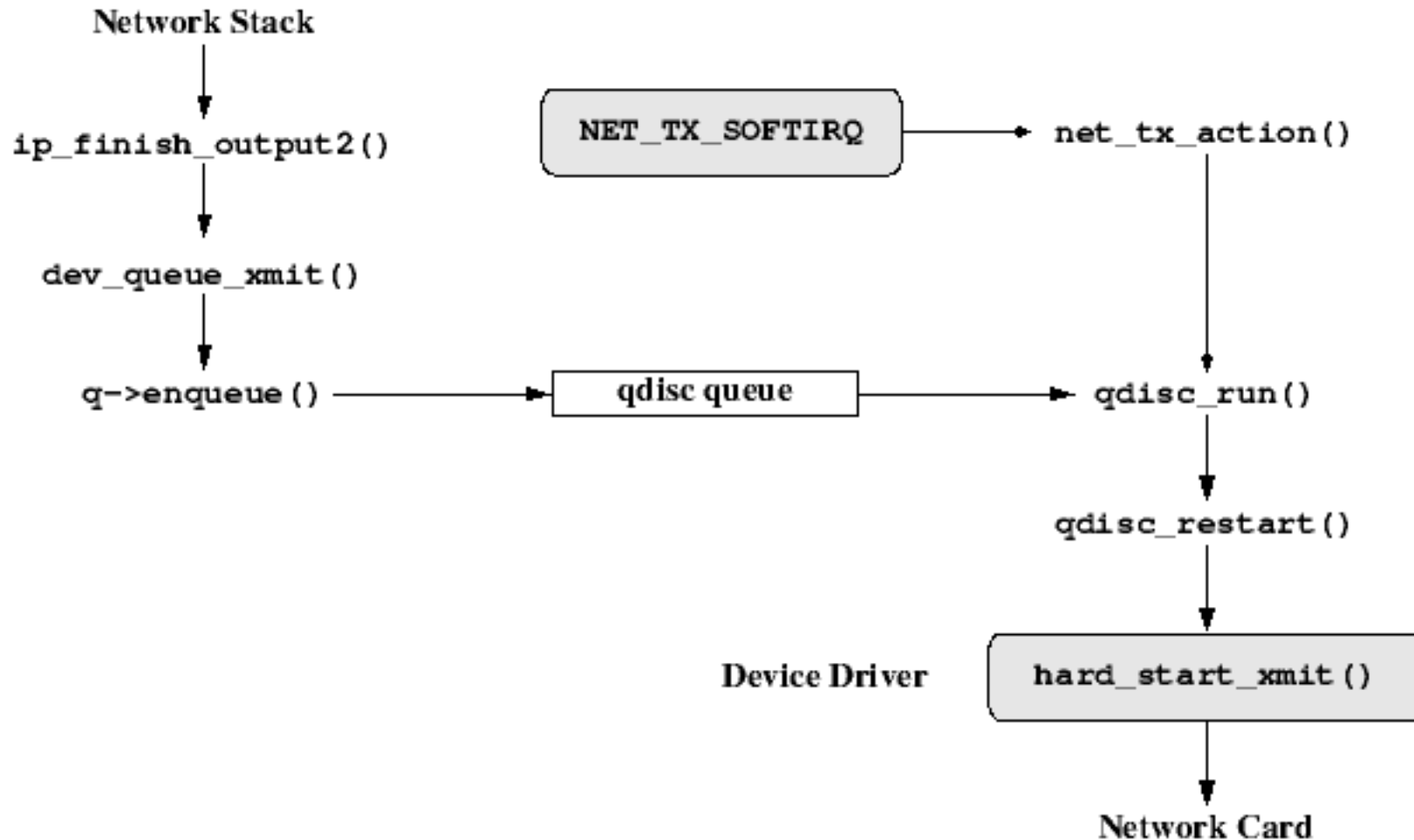
- ▶ Each network device defines a method:

```
int (*hard_start_xmit)(struct sk_buff *skb, struct
    net_device *dev);
```

- ▶ This method is indirectly called from NET_TX_SOFTIRQ
- ▶ Calls to this method are serialized via dev->xmit_lock_owner
- ▶ The driver can manage the transmit queue:

```
void netif_start_queue(struct net_device *net);
void netif_stop_queue(struct net_device *net);
void netif_wake_queue(struct net_device *net);
int netif_queue_stopped(struct net_device *net);
```

Packet Transmission Overview



Network Device Allocation

- ▶ Each network device is represented by a *struct net_device*.
- ▶ These structures are allocated using:

```
struct net_device *alloc_netdev(size, mask,  
                                setup_func);
```

- ▶ *size* – size of our priv data part
 - ▶ *mask* – a naming pattern (e.g. “eth%d”)
 - ▶ *setup_func* – A function to prepare the rest of net_device.
- ▶ And deallocated with:

```
void free_netdev(struct *net_device);
```

Network Device Allocation (cont.)

- ▶ For Ethernet device drivers, we have a short version:

```
struct net_device *alloc_etherdev(size);
```

- ▶ which calls *alloc_netdev(size, “eth%d”, ether_setup);*

Network Device Registration

- ▶ A network device driver provides interface to the network stack.
- ▶ It does not have or use major/minor numbers, like character devices and has no /dev file.
- ▶ A network driver is represented by a *struct net_device*.
- ▶ The structure is registered with the kernel via:

```
int register_netdev(struct net_device *dev);  
int unregister_netdev(struct net_device *dev);
```
- ▶ After filling in some important bits...

Network Device Initialization

- ▶ The *net_device* should be filled with numerous methods:
 - ▶ *open()* – request resources, register interrupts, start queues.
 - ▶ *stop()* – deallocates resources, unregister irq, stop queue.
 - ▶ *get_stats()* – report statistics.
 - ▶ *set_multicast_list()* – configure device for multicast.
 - ▶ *do_ioctl()* – device specific IOCTL function.
 - ▶ *change_mtu()* – Control device MTU setting.
 - ▶ *hard_start_xmit()* – called by the stack to initiate Tx.

Network Device Initialization (Cont.)

- ▶ Also, the *dev->flags* should be set according to device capabilities:
 - ▶ *IFF_MULTICAST* – device support multicast.
 - ▶ *IFF_NOARP* – device does not support the ARP protocol.

NAPI

- ▶ Network “**New API**”.
- ▶ Optional – provides interrupt mitigation under high load.
- ▶ Requirements:
 - ▶ A DMA ring buffer.
 - ▶ Ability to turn off receive interrupts.
- ▶ It is used by defining a new method:


```
int (*poll) (struct net_device *dev, int *budget);
```
- ▶ Called by the network stack periodically when signaled by the driver to do so.

NAPI (cont.)

- ▶ When a receive interrupt occurs, driver:
 - ▶ Turns off receive interrupts.
 - ▶ Calls *netif_rx_schedule(dev)* to get stack to start calling its poll method.
- ▶ Poll method:
 - ▶ Scans receive ring buffers, feeding packets to the stack via: *netif_receive_skb(skb)*.
 - ▶ If work finished within the budget parameter, re-enables interrupts and calls *netif_rx_complete(dev)*.
 - ▶ Else, the stack will call the *poll()* method again.

Embedded Linux driver development



Advice and Resources Getting Help and Contributions

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.



Information Sites (2)

KernelTrap

<http://kerneltrap.org/>



- ▶ Forum website for kernel developers.
- ▶ News, articles, white-papers, discussions, polls, interviews.
- ▶ Perfect if a digest is not enough!

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, April 2003
Karim Yaghmour, O'Reilly



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

Not very fresh, but doesn't depend too much on kernel versions.

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.



CE Linux Forum Resources



CE Linux Forum

CE Linux Forum's Wiki

is full of useful resources for embedded systems developers:

- ▶ Kernel patches not available in mainstream yet.
- ▶ Many howto documents of all kinds.
- ▶ Details about ongoing projects, such as reducing kernel size, boot time, or power consumption.
- ▶ Contributions are welcome!

<http://tree.celinuxforum.org/CelfPubWiki>

International Conferences (1)

Useful conferences featuring Linux kernel presentations

- ▶ Ottawa Linux Symposium (July): <http://linuxsymposium.org/>

Right after the (private) kernel summit.



Lots of kernel topics. Many core kernel hackers still present.

- ▶ Fosdem: <http://fosdem.org> (Brussels, February)



For developers. Kernel presentations from well-known kernel hackers.

- ▶ CE Linux Forum: <http://celinuxforum.org/>



Organizes several international technical conferences, in particular in California (San Jose) and in Japan. Now open to non CELF members!
Very interesting kernel topics for embedded systems developers.

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/>!

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

Annexes



Additional Materials For Your Reference

Embedded Linux Driver Development



Annexes

More Shell Tips & Tricks

The head and tail Commands

▶ `head [-<n>] <file>`

Displays the first <n> lines (or 10 by default) of the given file.

Doesn't have to open the whole file to do this!

▶ `tail [-<n>] <file>`

Displays the last <n> lines (or 10 by default) of the given file.

No need to load the whole file in RAM! Very useful for huge files.

▶ `tail -f <file>` (follow)

Displays the last 10 lines of the given file and continues to display new lines when they are appended to the file.

Very useful to follow the changes in a log file, for example.

▶ Examples

```
head windows_bugs.txt
```

```
tail -f outlook_vulnerabilities.txt
```

Environment Variables

- ▶ Shells let the user define *variables*.
They can be reused in shell commands.
Convention: lower case names
- ▶ You can also define *environment variables*: variables that are also visible within scripts or executables called from the shell.
Convention: upper case names.
- ▶ **env**
Lists all defined environment variables and their value.

Shell Variables Examples

Shell variables (bash)

▶ `projdir=/home/marshall/coolstuff`
`ls -la $projdir; cd $projdir`

Environment variables (bash)

▶ `cd $HOME`

▶ `export DEBUG=1`
`./find_extraterrestrial_life`
(displays debug information if `DEBUG` is set)

Measuring Elapsed Time

▶ `time find_expensive_housing --near`
`<...command output...>`

`real` `0m2.304s` (actual elapsed time)

`user` `0m0.449s` (CPU time running program code)

`sys` `0m0.106s` (CPU time running system calls)

$\text{real} = \text{user} + \text{sys} + \text{waiting}$

$\text{waiting} = \text{I/O waiting time} + \text{idle time (running other tasks)}$

The grep Command

▶ `grep <pattern> <files>`

Scans the given files and displays the lines which match the given pattern.

▶ `grep error *.log`

Displays all the lines containing `error` in the `*.log` files.

▶ `grep -i error *.log`

Same, but case insensitive.

▶ `grep -ri error .`

Same, but recursively in all the files in `.` and its subdirectories.

▶ `grep -v info *.log`

Outputs all the lines in the files except those containing `info`.

The sort Command

▶ `sort <file>`

Sorts the lines in the given file in character order and outputs them.

▶ `sort -r <file>`

Same, but in reverse order.

▶ `sort -ru <file>`

u: unique. Same, but just outputs identical lines once.

▶ More possibilities described later!

Symbolic Links

A symbolic link is a special file which is just a reference to the name of another one (file or directory):

- ▶ Useful to reduce disk usage and complexity when 2 files have the same content.
- ▶ Example:
`anakin_skywalker_biography -> darth_vador_biography`
- ▶ How to identify symbolic links:
 - ▶ `ls -l` displays `->` and the linked file name.
 - ▶ GNU `ls` displays links with a different color.

Creating Symbolic Links

- ▶ To create a symbolic link (same order as in `cp`):

```
ln -s file_name link_name
```

- ▶ To create a link with to a file in another directory, with the same name:

```
ln -s ../README.txt
```

- ▶ To create multiple links at once in a given directory:

```
ln -s file1 file2 file3 ... dir
```

- ▶ To remove a link:

```
rm link_name
```

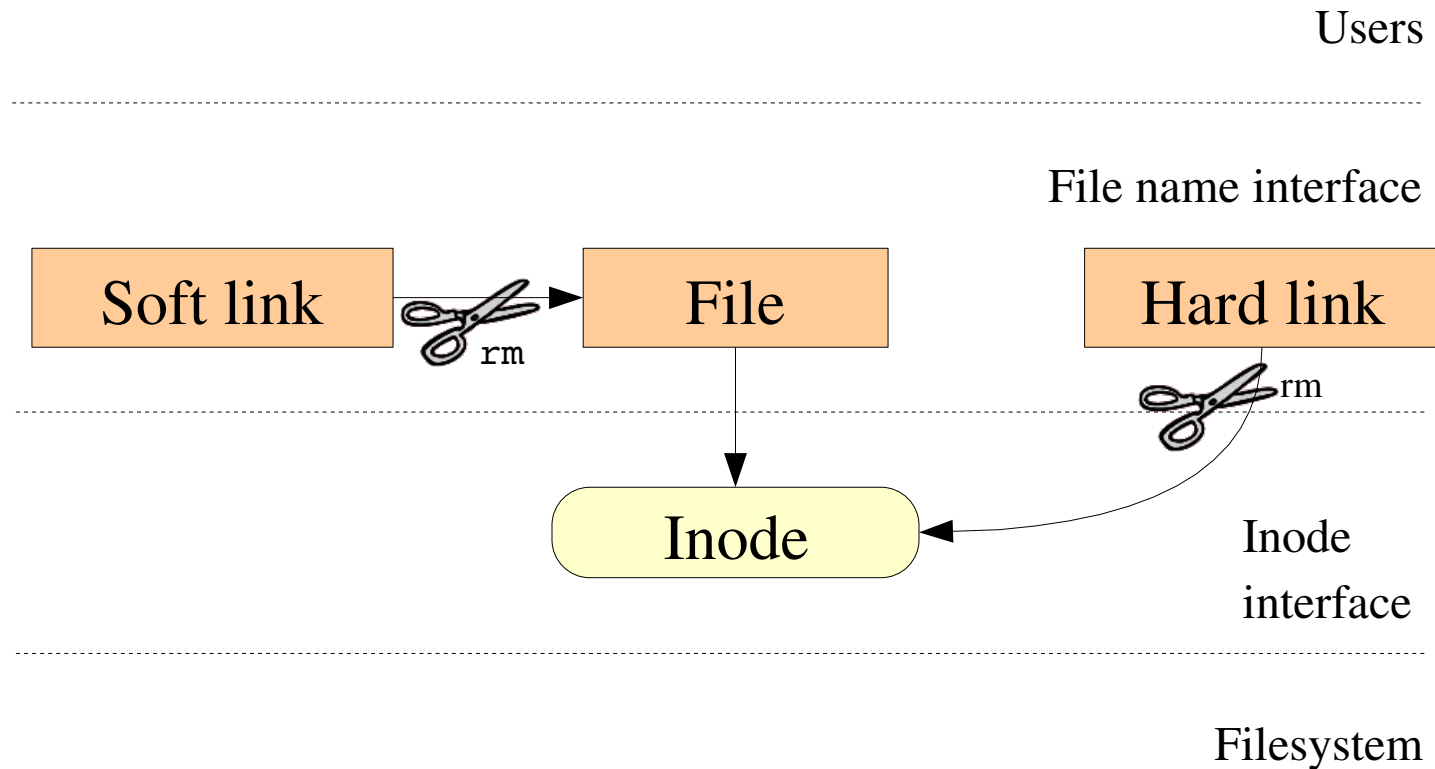
Of course, this doesn't remove the linked file!

Hard Links

- ▶ The default behavior for `ln` is to create *hard links*.
- ▶ A *hard link* to a file is a regular file with exactly the same physical contents.
- ▶ While they still save space, hard links can't be distinguished from the original files.
- ▶ If you remove the original file, there is no impact on the hard link contents.
- ▶ The contents are removed when there are no more files (hard links) to them.

Files Names and inodes

Makes hard and symbolic (soft) links easier to understand!



File Ownership

▶ `chown -R sco /home/linux/src` (`-R`: recursive)

Makes user `sco` the new owner of all the files in `/home/linux/src`.

▶ `chgrp -R empire /home/askywalker`

Makes `empire` the new group of everything in `/home/askywalker`.

▶ `chown -R borg:aliens usss_entreprise/`
`chown` can be used to change the owner and group at the same time.

File Access Rights

Use `ls -l` to check file access rights

3 types of access rights

- ▶ Read access (**r**)
- ▶ Write access (**w**)
- ▶ Execute rights (**x**)

3 types of access levels

- ▶ User (**u**): for the owner of the file
- ▶ Group (**g**): each file also has a “group” attribute, corresponding to a given list of users
- ▶ Others (**o**): for all other users

Access Right Constraints

- ▶ **x** without **r** is legal but is useless...
You have to be able to read a file to execute it.
- ▶ Both **r** and **x** permissions needed for directories:
x to enter, **r** to list its contents.
- ▶ You can't rename, remove, copy files in a directory if you don't have **w** access to this directory.
- ▶ If you have **w** access to a directory, you CAN remove a file even if you don't have write access to this file (remember that a directory is just a file describing a list of files). This even lets you modify (remove + recreate) a file even without **w** access to it.

Access Rights Examples

▶ `-rw-r--r--`

Readable and writable for file owner, only readable for others

▶ `-rw-r-----`

Readable and writable for file owner, only readable for users belonging to the file group.

▶ `drwx-----`

Directory only accessible by its owner.

▶ `-----r-x`

File executable by others but neither by your friends nor by yourself. Nice protections for a trap...



chmod: Changing Permissions

▶ `chmod <permissions> <files>`

2 formats for permissions:

▶ Octal format (abc):

$a, b, c = r*4 + w*2 + x$ (r, w, x: booleans)

Example: `chmod 644 <file>`

(rw for u, r for g and o)

▶ Or symbolic format. Easy to understand by examples:

`chmod go+r`: add read permissions to group and others.

`chmod u-w`: remove write permissions from user.

`chmod a-x`: (a: all) remove execute permission from all.

More chmod (1)

```
chmod -R a+rX linux/
```

Makes `linux` and everything in it available to everyone!

- ▶ **R**: apply changes recursively.
 - ▶ **X**: **x**, but only for directories and files already executable
- Very useful to open recursive access to directories, without adding execution rights to all files.

More chmod (2)

`chmod a+t /tmp`

- ▶ `t`: (sticky). Special permission for directories, allowing only the directory and file owner to delete a file in a directory.
- ▶ Useful for directories with write access to anyone, like `/tmp`.
- ▶ Displayed by `ls -l` with a `t` character.

Standard Output

More about command output.

- ▶ All the commands outputting text on your terminal do it by writing to their *standard output*.
- ▶ Standard output can be written (redirected) to a file using the `>` symbol
- ▶ Standard output can be appended to an existing file using the `>>` symbol

Standard Output Redirection Examples

- ▶ `ls ~saddam/* > ~gwb/weapons_mass_destruction.txt`
- ▶ `cat obiwan_kenobi.txt > starwars_biographies.txt`
`cat han_solo.txt >> starwars_biographies.txt`
- ▶ `echo "README: No such file or directory" > README`

Useful way of creating a file without a text editor.

Nice Unix joke too in this case.



Standard Input

More about command input:

▶ Lots of commands, when not given input arguments, can take their input from *standard input*.

▶ `sort`
`windows`
`linux`
`[Ctrl][D]`
`linux`
`windows`

`sort` takes its input from the standard input: in this case, what you type in the terminal (ended by `[Ctrl][D]`)

▶ `sort < participants.txt`

The standard input of `sort` is taken from the given file.

Pipes

▶ Unix pipes are very useful to redirect the standard output of a command to the standard input of another one.

▶ Examples

▶ `cat *.log | grep -i error | sort`

▶ `grep -ri error . | grep -v "ignored" | sort -u \> serious_errors.log`

▶ `cat /home/*/homework.txt | grep mark | more`

▶ This one of the most powerful features in Unix shells!

The tee Command

`tee [-a] file`

► The `tee` command can be used to send standard output to the screen and to a file simultaneously.

► `make | tee build.log`

Runs the `make` command and stores its output to `build.log`.

► `make install | tee -a build.log`

Runs the `make install` command and appends its output to `build.log`.

Standard Error

- ▶ Error messages are usually output (if the program is well written) to *standard error* instead of standard output.
- ▶ Standard error can be redirected through `2>` or `2>>`
- ▶ Example:
`cat f1 f2 nofile > newfile 2> errfile`
- ▶ Note: `1` is the descriptor for standard output, so `1>` is equivalent to `>`.
- ▶ Can redirect both standard output and standard error to the same file using `&>` :
`cat f1 f2 nofile &> wholefile`

Special Devices (1)

Device files with a special behavior or contents:

▶ `/dev/null`

The data sink! Discards all data written to this file.

Useful to get rid of unwanted output, typically log information:

```
mplayer black_adder_4th.avi &> /dev/null
```

▶ `/dev/zero`

Reads from this file always return `\0` characters

Useful to create a file filled with zeros:

```
dd if=/dev/zero of=disk.img bs=1k count=2048
```

See `man null` or `man zero` for details.

Special Devices (2)

▶ `/dev/random`

Returns random bytes when read. Mainly used by cryptographic programs. Uses interrupts from some device drivers as sources of true randomness (“entropy”).

Reads can be blocked until enough entropy is gathered.

▶ `/dev/urandom`

For programs for which pseudo random numbers are fine.

Always generates random bytes, even if not enough entropy is available (in which case it is possible, though still difficult, to predict future byte sequences from past ones).

See `man random` for details.

Embedded Linux driver development



Annexes

Init Runlevels (Sys V Init)

System V Init Runlevels (1)

- ▶ Introduced by System V Unix
Much more flexible than in BSD.
- ▶ Make it possible to start or stop different services for each runlevel.
- ▶ Correspond to the argument given to `/sbin/init`.
- ▶ Runlevels defined in `/etc/inittab`.

`/etc/inittab` excerpt:

```
id:5:initdefault:
```

```
# System initialization.
```

```
si::sysinit:/etc/rc.d/rc.sysinit
```

```
10:0:wait:/etc/rc.d/rc 0
```

```
11:1:wait:/etc/rc.d/rc 1
```

```
12:2:wait:/etc/rc.d/rc 2
```

```
13:3:wait:/etc/rc.d/rc 3
```

```
14:4:wait:/etc/rc.d/rc 4
```

```
15:5:wait:/etc/rc.d/rc 5
```

```
16:6:wait:/etc/rc.d/rc 6
```

System V Init Runlevels (2)

Standard levels:

- ▶ `init 0`
Halt the system.
- ▶ `init 1`
Single user mode for maintenance.
- ▶ `init 6`
Reboot the system.
- ▶ `init S`
Single user mode for maintenance.
Mounting only /. Often identical to 1

Customizable levels: 2, 3, 4, 5

- ▶ `init 3`
Often multi-user mode, with only command-line login.
- ▶ `init 5`
Often multi-user mode, with graphical login.

Init Scripts

According to `/etc/inittab` settings, `init <n>` runs:

- ▶ First `/etc/rc.d/rc.sysinit` for all runlevels
- ▶ Then scripts in `/etc/rc<n>.d/`
- ▶ Starting services (`1, 3, 5, S`):
runs `S*` scripts with the `start` option.
- ▶ Killing services (`0, 6`):
runs `K*` scripts with the `stop` option.
- ▶ Scripts are run in file name lexical order
Just use `ls -l` to find out the order!

/etc/init.d

- ▶ Repository for all available init scripts.
- ▶ `/etc/rc<n>.d/` only contains links to the `/etc/init.d/` scripts needed for runlevel `n`
- ▶ `/etc/rc1.d/` example (from Fedora Core 3):

```
K01yum -> ../init.d/yum
K02cups-config-daemon -> ../init.d/cups-
config-daemon
K02haldaemon -> ../init.d/haldaemon
K02NetworkManager ->
../init.d/NetworkManager
K03messagebus -> ../init.d/messagebus
K03rhnsd -> ../init.d/rhnsd
K05anacron -> ../init.d/anacron
K05atd -> ../init.d/atd
```

```
S00single -> ../init.d/single
S01sysstat -> ../init.d/sysstat
S06cpuspeed -> ../init.d/cpuspeed
```

Handling Init Scripts by Hand

Simply call the `/etc/init.d` scripts!

▶ `/etc/init.d/sshd start`

Starting sshd: [OK]

▶ `/etc/init.d/nfs stop`

Shutting down NFS mountd: [FAILED]

Shutting down NFS daemon:

[FAILED] Shutting down NFS quotas:

[FAILED]

Shutting down NFS services: [OK]

▶ `/etc/init.d/pcmcia status`

cardmgr (pid 3721) is running...

▶ `/etc/init.d/httpd restart`

Stopping httpd: [OK]

Starting httpd: [OK]

Embedded Linux Driver Development



Annexes

The Completely Fair Scheduler

The Completely Fair Scheduler

- ▶ The so-called Completely Fair Scheduler (CFS) was introduced by Red Hat's Ingo Molnar in kernel 2.6.23.
- ▶ As Ingo puts it: “CFS basically models an ideal, precise multi-tasking CPU”.
- ▶ CFS picks tasks to run according to the *p->wait_runtime* value, which stands for the amount of time (in nano-seconds) that the task should now run on the CPU for it to become completely fair and balanced.
- ▶ CFS also uses the *rq->fair_clock* value to track the CPU time a runnable task would have fairly gotten; For example, if there are four tasks in the system, the fair clock will increase at $\frac{1}{4}$ speed of the actual wall time.

The CFS Algorithm

- ▶ The CFS holds all tasks in a red-black tree, sorted according to “ $rq \rightarrow fair_clock - p \rightarrow wait_runtime$ ”.
- ▶ Therefore, the leftmost task in the tree (smallest value) is the one which the scheduler should pick next.
- ▶ This red-black tree algorithm is $O(\log n)$, which is a small drawback, considering the previous scheduler was $O(1)$.

Lab Information

▶ **Login:**

- ▶ User 'gby', password 'qwerty' or user 'root', password 'secret'

▶ **Running the Eclipse IDE:**

- ▶ /opt/course/scripts/eclipse &

▶ **Running the QEMU Emulator:**

- ▶ /opt/course/scripts/run-qemu &

▶ **The emulated board files are at:**

- ▶ /opt/course/emulation/root

▶ **The kernel module template is at:**

- ▶ /opt/course/skeleton/

Copyrights and Trademarks

© Copyright 2006-2004, Michael Opdenacker

© Copyright 2003-2006, Oron Peled

© Copyright 2004-2008 Codefidence Ltd.

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1, 2003 Edition, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 6, Copyright (C) 2001-2003 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html>.

Portions © Xavier Leroy <Xavier.Leroy@inria.fr>

Tux Image Copyright: (C) 1996 Larry Ewing

Linux is a registered trademark of Linus Torvalds.

All other trademarks are property of their respective owners.

Used and distributed under a Creative Commons Attribution-ShareAlike 2.0 license