

Programming Embedded Linux

Liran Ben Haim
liran@mabel-tech.com

Rights to Copy

- This kit contains work by the following authors:

- © Copyright 2004-2009 **Michael Opdenacker /Free Electrons**
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>
- © Copyright 2009-2017 **Bina Ltd.**
info@bna.co.il
<http://www.bna.co.il>

- 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>

What is Linux?



- ▼ Linux is a kernel that implements the POSIX and Single Unix Specification standards which is developed as an open-source project.
- ▼ Usually when one talks of “installing Linux”, one is referring to a Linux distribution.

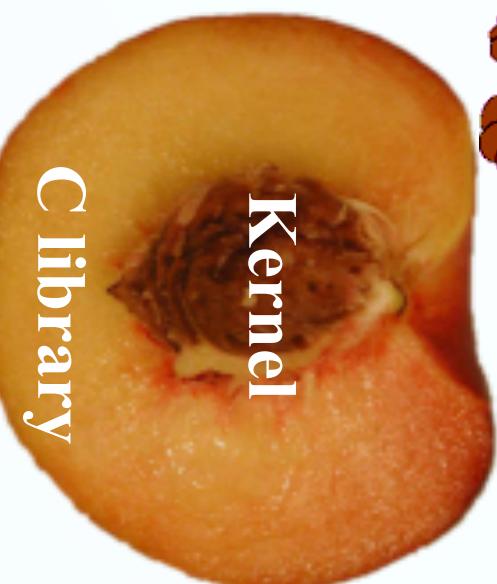
- ▼ A distribution is a combination of Linux and other programs and library that form an operating system.
- ▼ There exists many such distribution for various purposes, from high-end servers to embedded systems.
- ▼ They all share the same interface, thanks to the LSB standard.
- ▼ Linux runs on 21 platforms and supports implementations ranging from ccNUMA super clusters to cellular phones and micro controllers.
- ▼ Linux is 18 years old, but is based on the 40 years old Unix design philosophy.

Layers in a Linux System



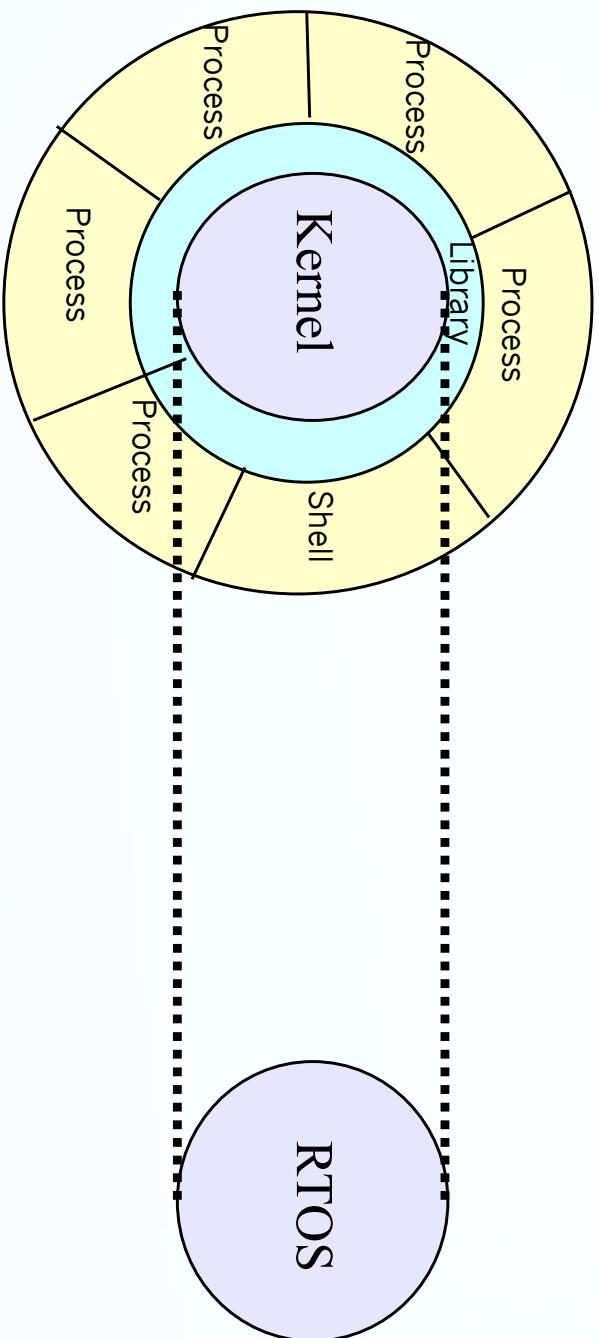
C library

Kernel



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

Linux VS. Legacy RTOS

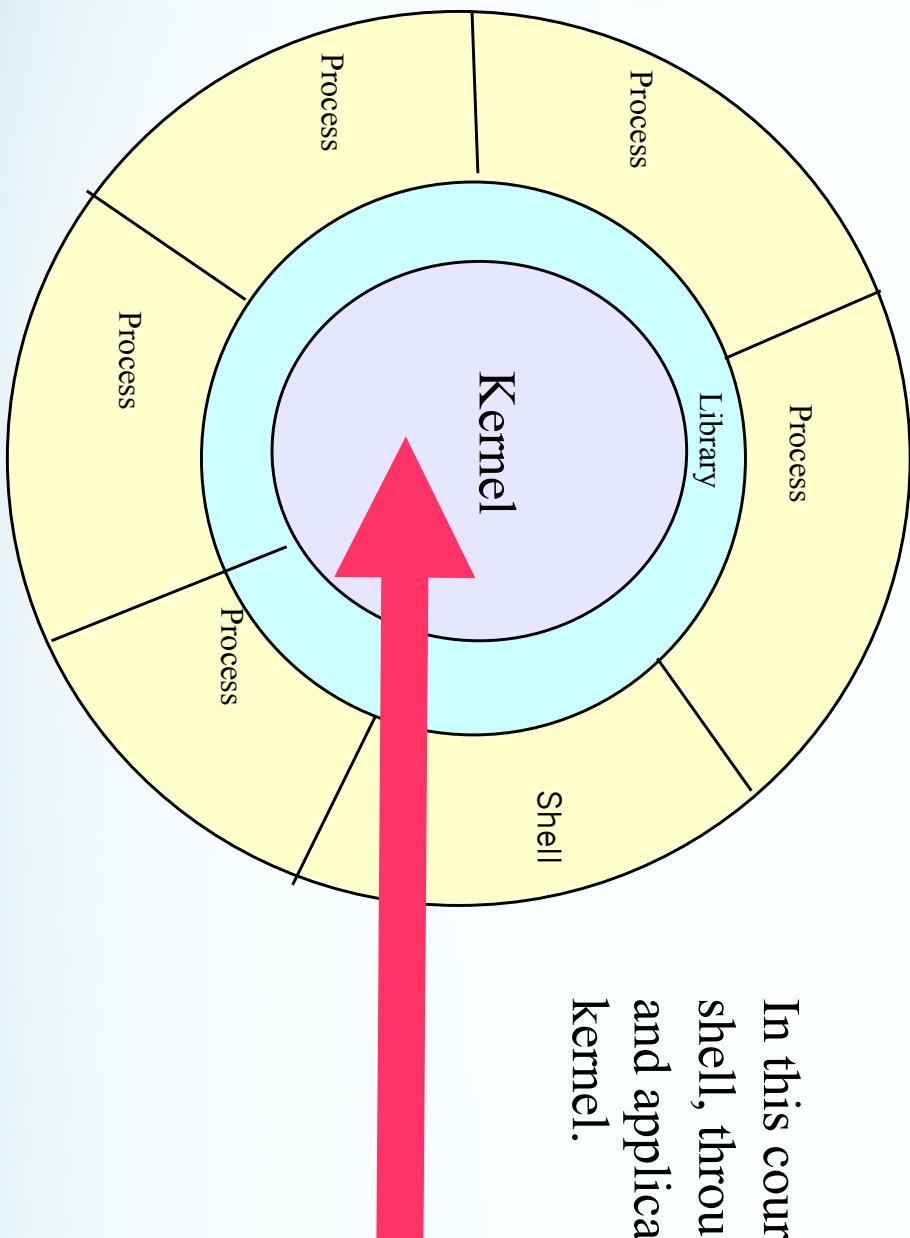


RTOS are like the Linux kernel:

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.

Agenda

▼ Day 1

- ▼ Files, the shell, admin, boot

▼ Day 2

- ▼ Application development

▼ Day 4

- ▼ Memory Management
- ▼ Character devices

- ▼ Hardware access
- ▼ Locking

▼ Day 5

- ▼ Linux kernel
- ▼ Writing a kernel module

- ▼ Time
- ▼ Interrupts
- ▼ Networking

- ▼ Additional resources

Embedded Linux Driver Development

The Basics

Everything is a file

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:
 - README
 - .bashrc
 - Buglist
 - index.htm
 - index.html.old
 - Windows

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!

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

GNU/Linux Filesystem Structure (2)

<code>/lost+found</code>	Corrupt files the system tried to recover
<code>/media/</code>	Mounted file systems
<code>(/mnt)</code>	
<code>/media/windows/</code>	Specific tools installed by the sysadmin
<code>/opt/</code>	
<code>/proc/</code>	Access to system information
<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)

`/tmp/`
Temporary files

`/usr/`
Regular user tools (not essential to the system)

`/usr/bin/, /usr/lib/, /usr/sbin...`

`/usr/local/`
Specific software installed by the sysadmin

(often preferred to `/opt/`)

`/var/`
Data used by the system or system servers

`/var/log/, /var/spool/mail` (incoming mail),

`/var/spool/lpd` (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...)

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
- ▶ **man -k [keyword]**
 - ▶ Search keyword in all man pages

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 -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
- ◀ **ls -t** (time)
Lists the most recent files
first

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

`./`

- ▼ 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).

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

▼ 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.

Listing All Processes

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

▼ **ps aux (procps version) OR ps (Busybox version)**

Lists all the processes running on the system

▼

```
ps aux   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	2.7	16.0	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
- ▶ **killall [-<signal>] <command>**
Kills all the jobs running <command>. Example:
`killall bash`

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)

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_entrepri\$e/**
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:
 - ▶ 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.
- ▶ Or 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)

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

- ▶ `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]

 - 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

- ▼ `ls *.txt | less`
- ▼ `cat *.log | grep -i error | sort`
- ▼ `grep -ri error . | grep -v "ignored" | sort -u \ > serious_errors.log`
- ▼ `cat /home/*/homework.txt | sort | more`
- ▼ This one of the most powerful features in Unix shells!
- ▼ Grep searches file, sort sorts them. More info at the appendixes.

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:

```
mp4player 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

```
mkfs.ext3 ./disk.img  
mount -t ext3 -o loop ./disk1.img /mnt/image
```

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.

The Unix and GNU/Linux Command-Line

System Administration Basics

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...`)
- ▶ Mount options for each device can be stored in the `/etc/fstab` file.

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

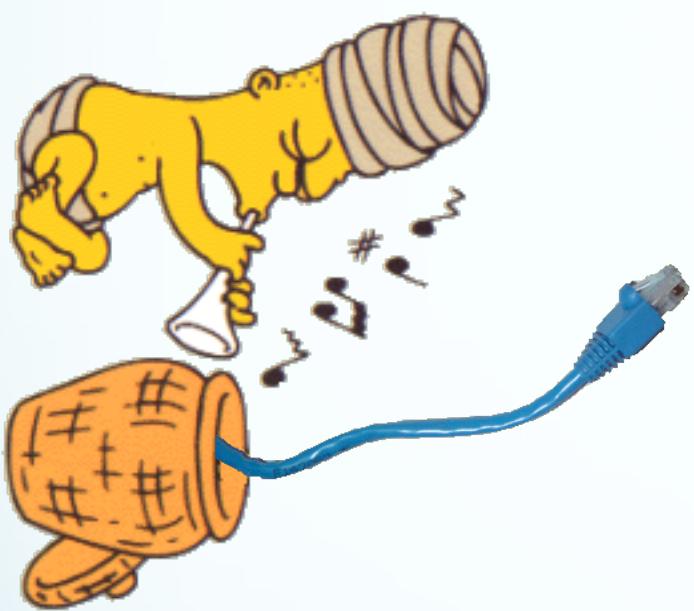
▶ `umount /mnt/usbdisk`

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 `lsof` command (`list open files`) to view which processes still have open files in the mounted partition.

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.

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!

Network Testing

► ping freshmeat.net

ping 192.168.1.1

Tries to send packets to the given machine and get acknowledgement 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!

The Best Tech PROC File System 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
- ▶ ...

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
- ▼ Programmatically, 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.

Sysfs

- ▶ Sysfs is a virtual file system that represents the hardware and drivers in the system:
- ▶ Devices existing in the system: their power state, the bus they are attached to, and the driver responsible for them.
- ▶ The system bus structure: which bus is connected to which bus (e.g. USB bus controller on the PCI bus), existing devices and devices potentially accepted (with their drivers)
- ▶ Available device drivers: which devices they can support, and which bus type they know about.
- ▶ The various kinds ("classes") of devices: **input**, **net**, **sound**... Existing devices for each class.

Using SysFS

▼ For example:

▼ View all PCI devices:

```
ls /sys/devices/pci0000\::00/
```

```
0000:00:00.0 0000:00:1a.1 0000:00:1c.2 0000:00:1d.2 0000:00:1f.3 ...
```

▼ Check which interrupt is used by a certain device:

```
cat /sys/devices/pci0000\::00/0000:00:1c.4/irq
```

23

▼ Sysfs documentation:

▼ [Documentation/driver-model/](#)

▼ [Documentation/filesystems/sysfs.txt](#)

Embedded Linux Driver Development

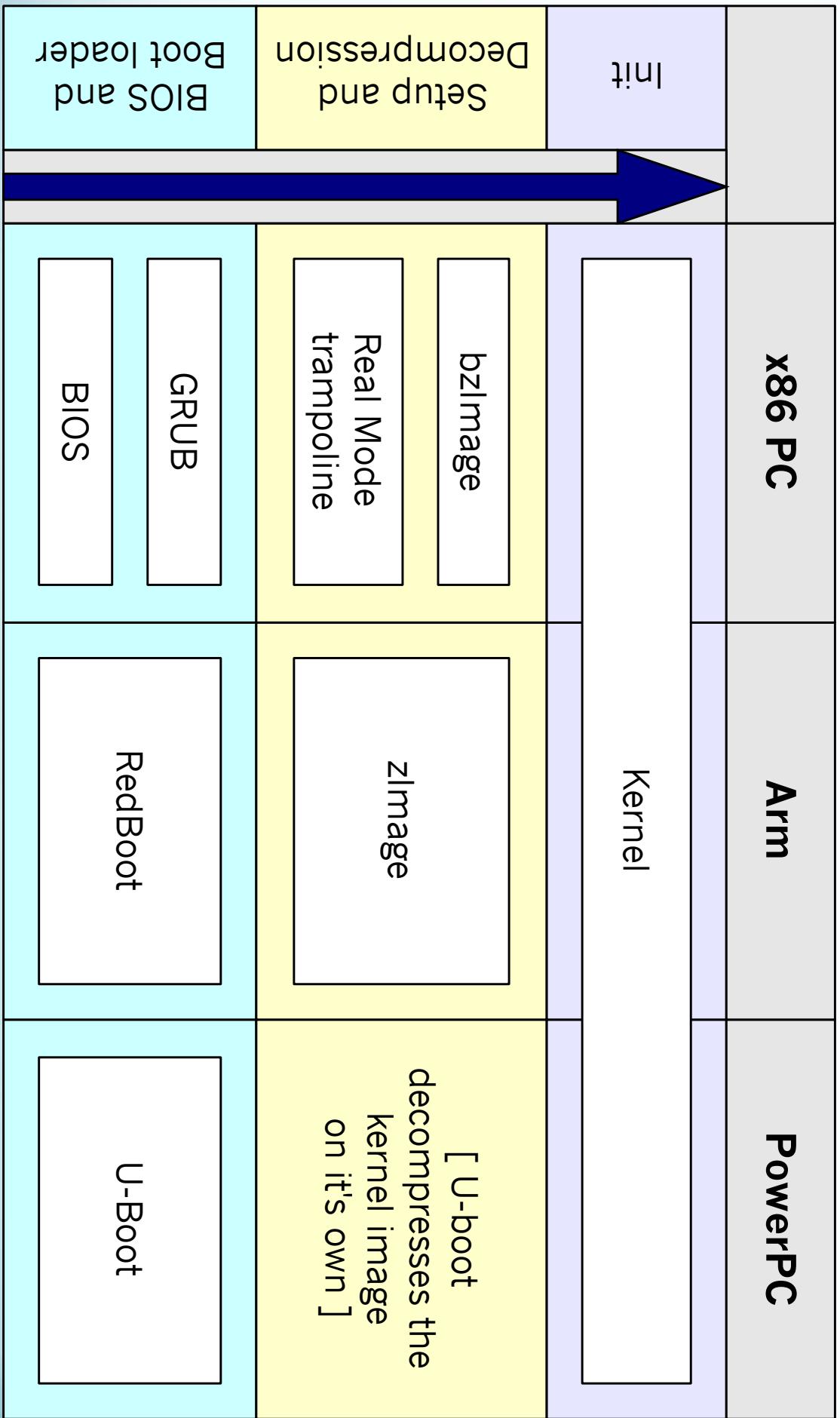
System Administration Basics

Boot Sequence

Linux Boot Process

- ▶ BIOS and/or boot loader initializes hardware.
- ▶ Boot loader loads kernel image into memory.
 - ▶ Boot loader can get kernel image from flash, HD, network.
- ▶ Possibly also loads a file system to RAM.
- ▶ Boot loader or kernel decompress compressed kernel.
- ▶ Kernel performs internal (hash table, lists etc.) and hardware (device driver) setup.
- ▶ Kernel finds and mounts the root file system.
- ▶ Kernel executes the “/sbin/init” application.

Boot Sequences



Root File System Options

- ▶ Many storage devices available:
 - ▶ On flash (NAND / NOR)
 - ▶ On CompactFlash / SDD
 - ▶ On Disk (SCSI / PATA / SATA)
 - ▶ In RAM (INITRamDisk or INITial RAM FileSystem)
 - ▶ From network (NFS)
- ▶ Type to use either hard coded in kernel config or passed to kernel by boot loader in the kernel command line string.

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.
 - ▶ The 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

Startup the system

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

Format:

```
::sysinit:/bin/mount -t proc proc /proc
::sysinit:/bin/mount -a
::sysinit:/sbin/ifconfig lo 127.0.0.1 up
```

id: runlevel : action : command

Put a getty on the serial port

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

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

Start system loggers

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

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

Stuff to do before rebooting

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

command: shell command to execute

Coding Embedded Linux Applications

Writing Applications

A Simple Makefile

```
CC=/usr/local/arm/2.95.3/bin/arm-linux-gcc
CFLAGS=-g
LDFLAGS=-lpthread
.PHONY: clean all

all: testapp

test1.o: test1.c test1.h
    $(CC) $(CFLAGS) -c test1.c

test2.o: test2.c test2.h
    $(CC) $(CFLAGS) -c test2.c

testapp: test1.o test2.o
    $(CC) $(LDFLAGS) test1.o test2.o -o testapp

clean:
    @rm -f *.o *~ 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;
}
```

Opening Files

- ▼ To work with a file we need to open it and get a file descriptor:

```
int fd;  
  
fd = open("/dev/drv_ctl", O_RDWR);  
  
▼ open() returns the file descriptor or -1 for failure.  
▼ In case of failure the special variable errno can be  
used to get the error code.  
  
if(-1 == fd)  
  
fprintf(stderr, "Error %s", strerror(errno));
```

File Flags

- ▼ Various file status flags can be passed to open()
- ▼ **O_NONBLOCK** - open file in non-blocking mode.
- ▼ **O_SYNC** – disable OS caching of writes to the file.
- ▼ **O_ASYNC** – send notification when data is available.
- ▼ File creation permission can also be set in open.
See man open(2) for more flags.

- ▼ File status flags can also be changed after open(2) using fcntl(2):

```
fcntl(fd, F_SETFL, O_NONBLOCK);
```

File Operations

- ▼ **read, write**: read or write A buffer of data to a file
- ▼ **pread, pwrite**: read or write from specified file offset.
- ▼ **readv, writev**: like read/write, but with an array of buffers
- ▼ **select, poll, epoll**: block for event on one or more files
- ▼ **ioctl**: I/O control, exchange a buffer with device driver
- ▼ **fsync, fdatasync**: flush kernel cache of driver to hardware
- ▼ **mmap**: map resource to memory (will be described later).

Read Operations

- `ssize_t read(int fd, void *buf, size_t count);`
- ▶ **count:** maximal number of bytes to read.
- ▶ **buf:** pointer to buf to copy/transfer data to.
- ▶ **fd:** file descriptor
- ▶ return value is how many bytes were transferred (can be less than requested) or -1 for error (errno has error code).
- ▶ Position in file starts from 0 and advances with read/writes.
- ▶ You can change position with lseek(2) or use pread(2).

ioctl Operations

- ▼ ioctl operations send command to driver and optionally sends or gets a long or pointer to buffer.

```
int ioctl(int fd, int request, unsigned long param);
```
- ▼ **request:** integer defined by driver.
- ▼ **param:** long or pointer to buffer (usually a structure).
- ▼ can be either in, out or in/out operation.
- ▼ ioctl are for when nothing else fits - don't use ioctl(2) if other file operations is better suitable (read, write ...)

Working with processes

- ◀ A processes is a container which holds a running program.
- ◀ Each container is a sand box which has its own set of file descriptors and virtual memory space.
- ◀ When a process ends all the resources it uses (memory, files etc.) are automatically collected by the system.
- ◀ You first create a new process, then set it's properties.
- ◀ When a new process is created it starts to run automatically.
- ◀ You create new processes using the fork(2) call.

Physical and Virtual Memory

Virtual address spaces

0xFFFFFFFF

0xFFFFFFFF

0xFFFFFFFF

I/O memory 3

I/O memory 2

I/O memory 1

Flash

Kernel

Process2

0x00000000

0x00000000

MMU

CPU

Memory Management Unit

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

0x00000000

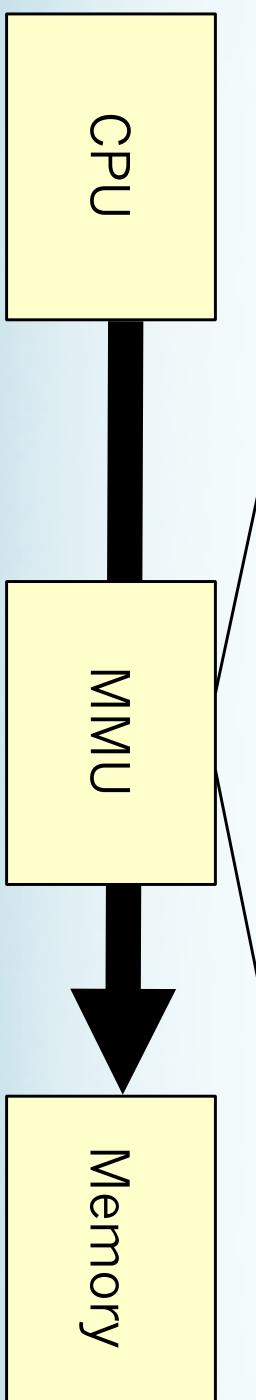
0x00000000

The Memory Management Unit

- ▶ The tables that describes the virtual to physical translations are called page tables. They reside in system memory.

- ▶ Each entry describes a slice of memory called a page.

Context	Virtual	Physical	Permission
12	0x8000	0x5340	RWX
15	0x8000	0x3390	RX



Process VMA

- You can view a user application mapping from the proc file system:

```

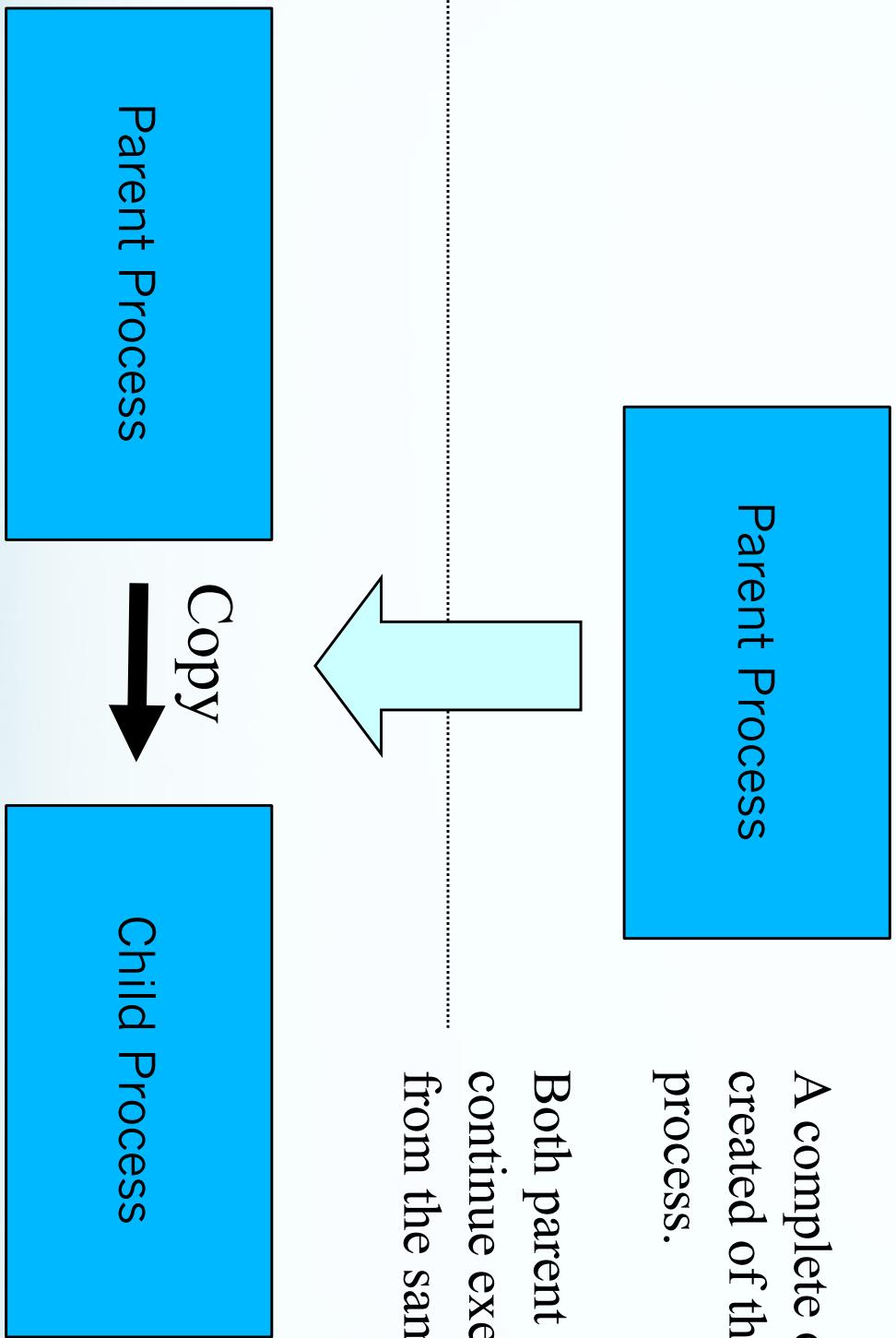
• > cat /proc/1/maps (init process)
    start           end             perm offset      major:minor inode
00771000-0077f000 r-xp 00000000 03:05 1165839  /lib/libselinux.so.1
0077f000-00781000 rw-p 0000d000 03:05 1165839  /lib/libselinux.so.1
0097d000-00992000 r-xp 00000000 03:05 1158767  /lib/ld-2.3.3.so
00992000-00993000 r-p 00014000 03:05 1158767  /lib/ld-2.3.3.so
00993000-00994000 rw-p 00015000 03:05 1158767  /lib/tls/libc-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
fefd4000-f0000000 rw-p fefd4000 00:00 0
fffffe000-ffffff000 ---p 00000000 00:00 0

```

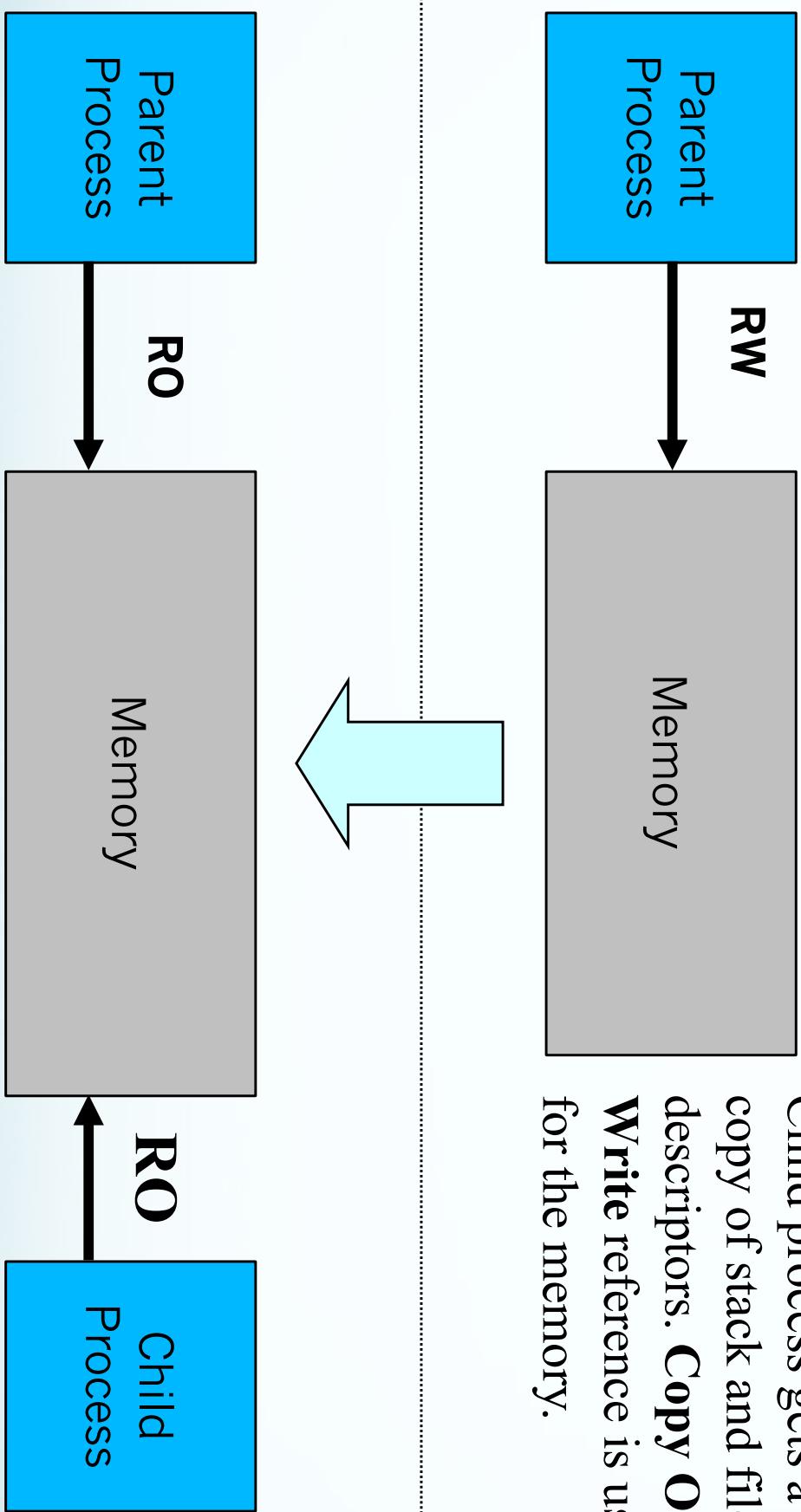
Creating Processes Using **fork()**

```
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

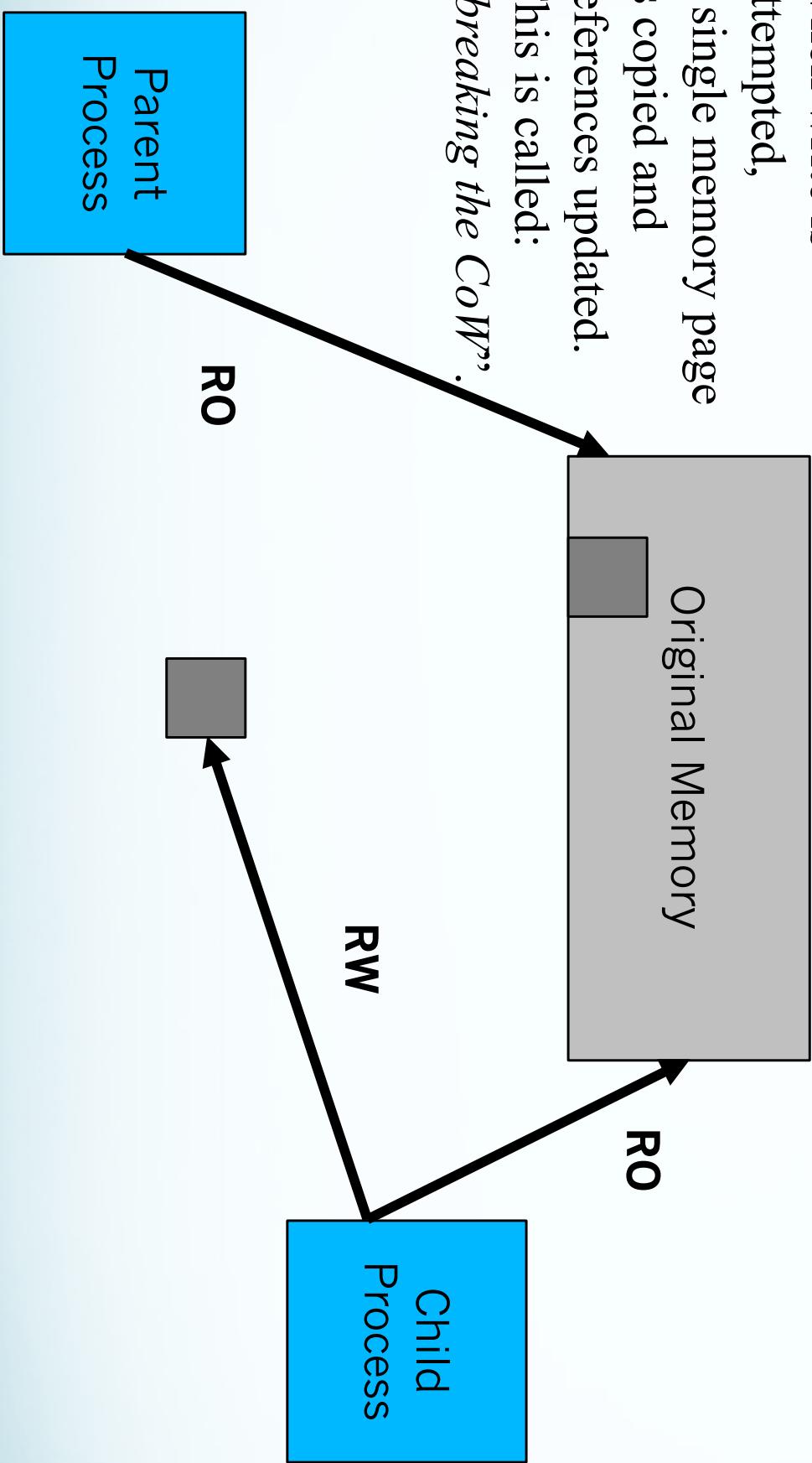


How fork() Really Works



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.

```
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);  
}
```

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

```
}
```

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 it's 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.

Mapping Memory Using mmap()

- `void *mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);`
- ▶ `mmap()` asks the kernel to change a process page table.
- ▶ The `mmap()` function asks to map `length` bytes starting at offset `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.

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.

Protecting Memory

- ▼

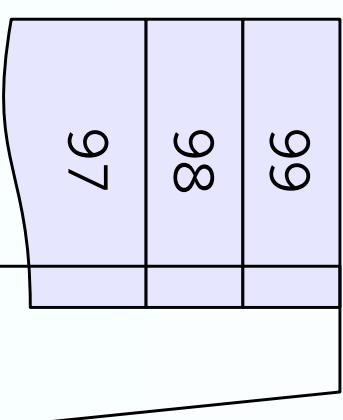
```
int mprotect(const void *addr, size_t len,  
int prot);
```
- ▼ The function `mprotect()` specifies the desired protection for the memory page(s) containing part or all of the address range.
- ▼ If an access is disallowed by the protection given it, the program receives a `SIGSEGV` signal.
- ▼ `prot` is a bitwise-or of the following values:
 - ◀ `PROT_NONE`
 - ◀ `PROT_READ`
 - ◀ `PROT_WRITE`
 - ◀ `PROT_EXEC`
- ▼ Useful to find bugs (accessing a wrong memory region, stack etc.)

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.
- ▶ Not for threads!!!
- ▶ This behavior can be limited by setting a resource limit on the stack size.
 - ▶ See `setrlimit()`
 - ▶ `ulimit -s [size]/unlimited`

Linux Priorities

Real Time priority



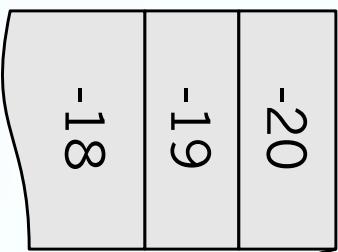
Real time processes

SCHED_FIFO

SCHED_RR

...

Nice level



...

-20

-19

-18

...

0

Non real-time processes

SCHED_OTHER

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().
```

Guarantee Real Time Response

- ▼ To get deterministic real time response from a Linux process, make sure to:
- ▼ Put the process in a real time scheduling domain using `sched_setscheduler()`
- ▼ Use `mlockall()` to lock all process memory, both current and future.
- ▼ 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.

Real Time Example

#define MY_PRIORITY (49)

#define MAX_SAFE_STACK (8*1024)

- ▶ Some defines.

```
void stack_prefault(void) {  
    unsigned char dummy[MAX_SAFE_STACK];  
    memset(&dummy, 0, MAX_SAFE_STACK);  
}
```

- ▶ This make sure the stack is allocated.

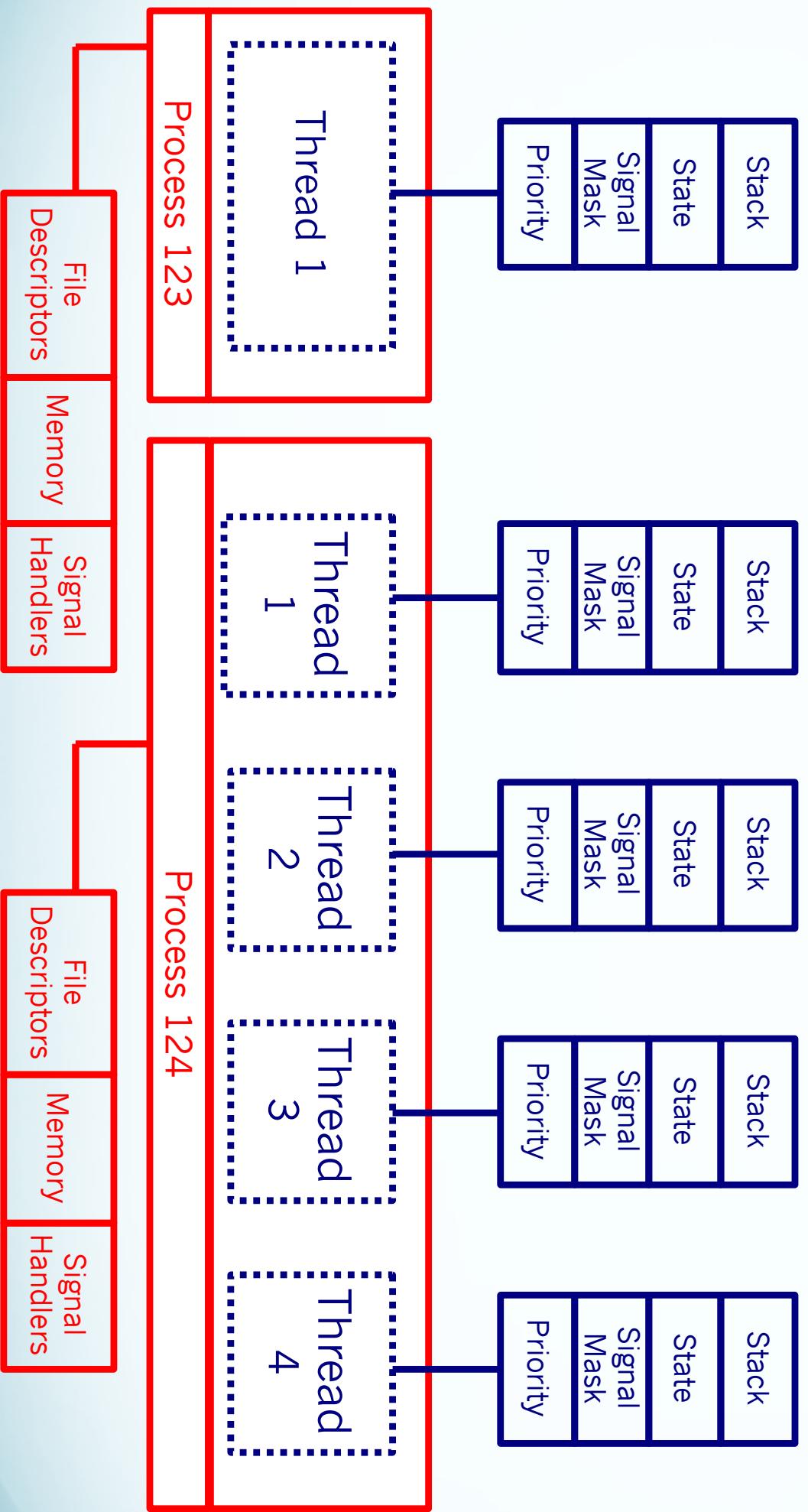
```
int main(int argc, char* argv[]) {  
    struct sched_param param;  
    param.sched_priority = MY_PRIORITY;  
    // Continue on next slide... ==>
```

Real Time Example

- ▶ Set ourselves as a real time task with FIFO scheduling
- ▶ Lock all memory allocation in physical memory, current and future.
- ▶ Pre-fault the stack.
- ▶ It's now OK to do real time stuff.

```
if(sched_setscheduler(0, SCHED_FIFO,  
&param) == -1) {  
    exit(1);  
}  
  
if(mlockall(MCL_CURRENT|MCL_FUTURE) == -  
1) {  
    exit(2);  
}  
  
stack_prefault();  
  
do_interesting_stuff();  
}
```

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 the context switch time are the same.
- ▼ To use threads in your code:
 - ◀ `#include <pthread.h>`
- ▼ In your Makefile:
 - ◀ Add `-lpthread` to LDFLAGS

Creating Threads

- Function: **pthread_create()**

```
int pthread_create(pthread_t *thread, 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)*.
- ▶ For SCHED_OTHER policy use *setpriority(2)*

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()**
 - ▼ 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

Action	Main Thread	Other Thread
<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()*.

Threads Considered Harmful

- ▶ Threads were created to allow for a light-weight process on OS were process creation time is very high but Linux processes are light weight, thanks to CoW.
- ▶ Threads share too much and using them nulls the advantage of having protection between tasks.
Partial sharing can easily be done using shared memory between processes.
- ▶ There is nothing you can do with threads you can't do with processes.

Linux IPC

▼ Inter Process (and thread)

Communication

- ▼ Mutexes
- ▼ Condition Variables
- ▼ Semaphores
- ▼ Shared memory
- ▼ Message Queues
- ▼ Pipes
- ▼ Unix Domain Sockets
- ▼ Signals

Creating a Mutex

- Function: **pthread_mutex_init()**
 - ▼

```
int pthread_mutex_init(pthread_mutex_t *mutex, const
pthread_mutexattr_t *attr);
```
 - ▼

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```
 - ▼ 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.

Mutex Attributes

- ▶ `int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);`
- ▶ `int pthread_mutexattr_init(pthread_mutexattr_t *attr);`
 - ▶ Init and destroy attribute structure.
- ▶ `int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);`
- ▶ Permit a mutex to be operated upon by any task that has access to the memory where the mutex is allocated.
- ▶ More ahead...

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

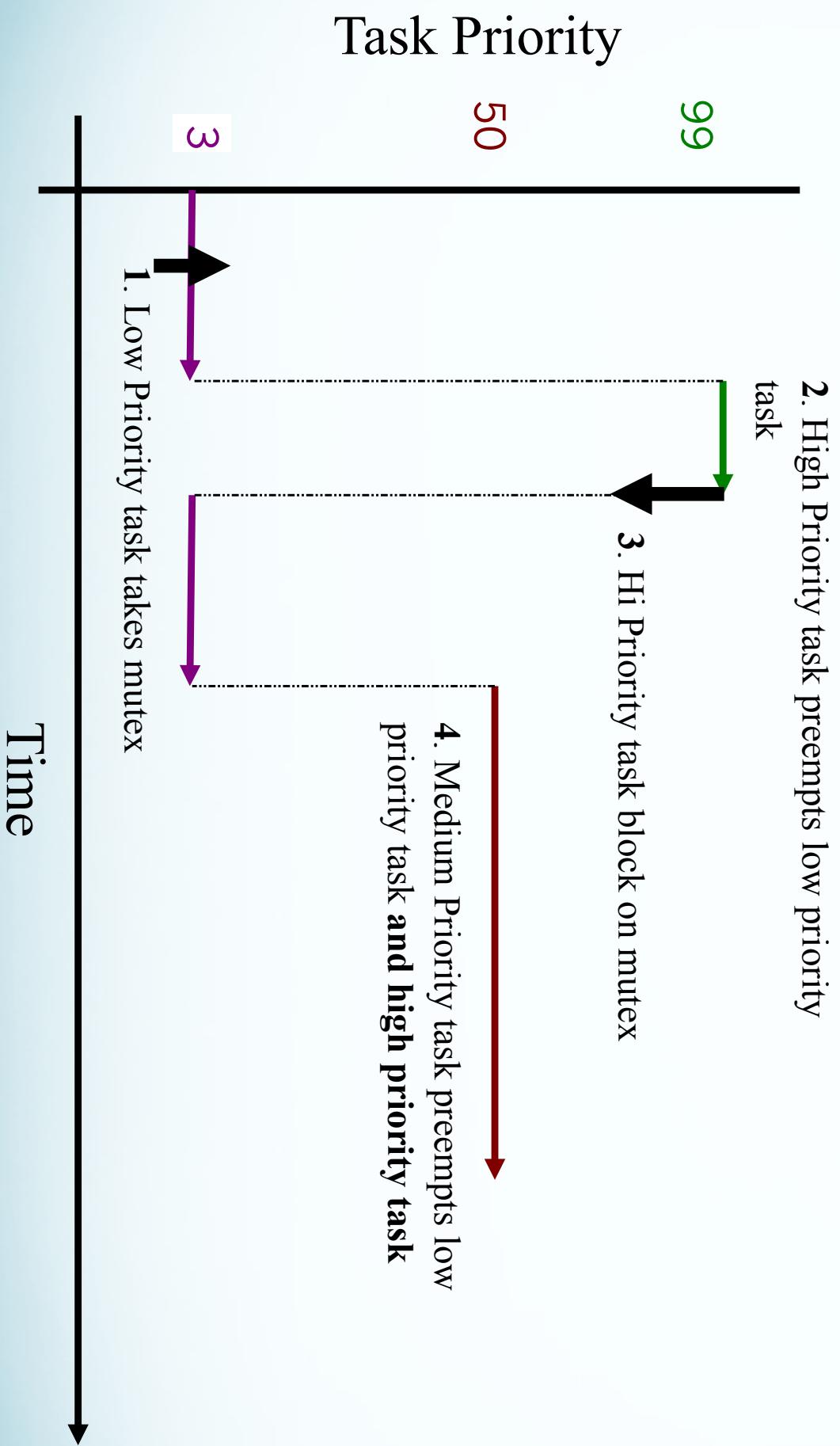
- Function: **pthread_mutex_timedlock()**

- ```
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.

# Setting Mutex Priority Protocol

- ```
int pthread_mutexattr_setprotocol
(pthread_mutexattr_t *attr, int protocol);
```
- ▼ Set priority protocol: PTHREAD_PRIO_NONE, PTHREAD_PRIO_INHERIT PTHREAD_PRIO_PROTECT

- int pthread_mutexattr_setprioceiling(pthread_mutex_attr_t *restrict mutex, int prioceiling, int *oldceiling);
- ▼ **prioceiling:** which priority ceiling to use.
- ▼ Can also set in run time with pthread_mutex_setprotocol(3) and pthread_mutex_setprioceiling(3) calls.

Condition Variables

- ▶ Condition variables are queues where tasks can wait blocking until an a certain condition becomes true.
- ▶ A condition is a predicate, like in: if($x==0$) ...
- ▶ Condition variables use mutexes to protect the testing and setting of the condition.
- ▶ The condition variable API allows a task to **safely** test if the condition is true, blocking if it isn't, without race conditions.
- ◀ Use of condition variables can help avoid many subtle bugs.

Condition Variables API

- `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 (fFlag == 0)
        pthread_cond_wait(cond, lock);
    fFlag = 0;
    pthread_mutex_unlock(lock);
}

void my_post_event(pthread_mutex_t *lock, pthread_cond_t *cond) {
    pthread_mutex_lock(lock);
    fFlag = 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 controls whether the semaphore can be used from different processes via shared memory.

Even More Locks

- ▶ **POSIX spin locks**

- ▶ `pthread_spin_lock()` ...

- ▶ **Reader Writer locks**

- ▶ `pthread_rwlock_rdlock()`

- ▶ `pthread_rwlock_wrlock()` ...

POSIX Shared Memory

- `int 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()`!

POSIX Message Queue

- ▶ POSIX message queues allow processes to exchange data in the form of messages.
- ▶ Message queues are created and opened using `mq_open(3)`;
- ▶ The returns a queue descriptor (`mqd_t`).
- ▶ Each queue is identified by a name of the form `/somename`.
- ▶ Messages are transferred to and from a queue using `mq_send(3)` and `mq_receive(3)`.
- ▶ Queue are closed using `mq_close(3)`, and deleted using `mq_unlink(3)`.

Getting A Queue

- ▶ #include <mqqueue.h>
- ▶ mqd_t mq_open(const char *name, int oflag,
mode_t mode, struct mq_attr *attr);
- ▶ **O_RDONLY, O_WRONLY, O_RDWR**.
- ▶ **O_NONBLOCK**: Open the queue in non-blocking
mode.
- ▶ **O_CREAT**: Create the message queue if it does not
exist.
- ▶ **O_EXCL**: If a queue already exists, then fail.
- ▶ The mode argument provides permissions.

Attributes

- ▶ mqd_t mq_getattr(mqd_t mqdes, struct mq_attr *attr);
- ▶ mqd_t mq_setattr(mqd_t mqdes, struct mq_attr *newattr, struct mq_attr *oldattr);
- ▶ Retrieve and modify attributes of the queue referred to by the descriptor mqdes.

```
struct mq_attr {  
  
    long mq_flags;           /* Flags: 0 or O_NONBLOCK */  
  
    long mq_maxmsg;         /* Max. # of messages on queue */  
  
    long mq_msgsize;        /* Max. message size (bytes) */  
  
    long mq_curmsgs;        /* # of messages currently in queue */  
};
```

Sending Messages

- ▶ `mqd_t mq_send(mqd_t mqdes, const char *msg_ptr,
size_t msg_len, unsigned msg_prio);`
- ▶ `mqd_t mq_timedsend(mqd_t mqdes, const char
*msg_ptr, size_t msg_len, unsigned msg_prio, const
struct timespec *abs_timeout);`
- ▶ `mq_send()` adds the message `msg_ptr` to the
queue.
- ▶ `msg_prio` specifies the message priority.
 - ▶ FIFO is used for same priority messages.
- ▶ If queue is full send is blocked, unless
`O_NONBLOCK` used.

Receiving Messages

- ▶ `mqd_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned *msg_prio);`
- ▶ `mqd_t mq_timedreceive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned *msg_prio, const struct timespec *abs_timeout);`
- ▶ `mq_receive()` removes the oldest message with the highest priority from the queue, and places it in the buffer.
- ▶ If `prio` is not `NULL`, then it is used to return the priority.
- ▶ If the queue is empty, `mq_receive()` blocks unless `O_NONBLOCK` was used.

Message Notification

- ▶ `mqd_t mq_notify(mqd_t mqdes, const struct sigevent *notification);`
- ▶ `mq_notify()` allows the process get an asynchronous notification when a new message arrives.
- ▶ Use a NULL notification to un-register.
- ▶ Notification only occurs for new messages on empty queue.
- ▶ `mq_receive` takes precedence over notifications.
- ▶ Notification occurs once: after a notification is delivered, the notification registration is removed.

Notifications

```
struct sigevent {  
    int     sigev_notify;      /* Notification method */  
    int     sigev_signo;       /* Notification signal */  
    union   sigval sigev_value; /* Data passed with notification */  
    void   (*sigev_notify_function) (union sigval);  
        /* Function for thread notification */  
    void   *sigev_notify_attributes;  
        /* Thread function attributes */  
};
```

▼ Notification methods:

- ▼ **SIGEV_NONE**: no notifications
- ▼ **SIGEV_SIGNAL**: send signal `sigev_signo` with value `sigev_value`.
- ▼ **SIGEV_THREAD**: start thread with function `sigev_notify_function` and `sigev_value` parameter

POSIX Pipes

- `int pipe(int filedes[2]);`
- ▶ `pipe(2)` creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by `filedes`.
- ▶ `filedes[0]` is for reading, `filedes[1]` is for writing.
- int `mkfifo(const char *pathname, mode_t mode);`
- ▶ A `fifo(7)` is a special file that behaves like a pipe.
- ▶ Since `fifo(7)` exists in the file system, it can be opened by two non related processes.

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, "/tmp/socket0");  
  
bind(sock, (struct sockaddr *) &server, sizeof(struct  
sockaddr_un));
```

- ▶ Exists in a stream (like TCP) and datagram (like UDP) flavors.
- ▶ UDS can be used to pass file descriptors between processes.

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.
- ▼ Two default signal handlers exist:
 - ▼ **SIG_IGN**: Ignore the specified signal.
 - ▼ **SIG_DFL**: Go back to default behaviour.
- ▼ Or you can register a call back function that gets called when the process receives that 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

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

- SIGSTP 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)

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!

Async. Signal Safety

- ▶ Must be very careful to write signal handler only with async safe code.
- ▶ For example, no printf(), malloc(), any function that takes lock or functions calling them.
- ▶ Can only use POSIX async-signal safe functions
- ▶ See signal(7) for the list.
- ▶ Typical signal handler just sets a flag.

POSIX Timers Overview

- ▶ #include <signal.h>
- ▶ #include <time.h>
- ▶ int timer_create(clockid_t clockid, struct sigevent *restrict evp, timer_t *restrict timerid);
- ▶ int timer_gettime(timer_t timerid, int flags, const struct itimerspec *restrict value, struct itimerspec *restrict ovalue);
- ▶ int timer_gettime(timer_t timerid, struct itimerspec *value);
- ▶ int timer_getoverrun(timer_t timerid);
- ▶ int timer_delete(timer_t timerid);

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.

Build for Debugging

- ▼ In order to effectively debug a program, we need to build the program with debug information
- ▼ Debug information saves the relation between the program binary and its source and is required for proper GDB operation
- ▼ To build a binary with debug information, pass the `-g` argument to GCC
- ▼ Usually passed via the CFLAGS Makefile variable

Remote Debug Example

- ▶ On the target:

- ▶ **\$ `gdbserver /dev/ttys0 my_prog 12 3`**

- ▶ Load `my_prog` with parameters 12 3 and wait for the debugger to connect on the first serial port

- ▶ **\$ `gdbserver 0.0.0.0:9999 my_prog 12 3`**

- ▶ Load `my_prog` with parameters 12 3 and wait for the debugger to connect on TCP port 9999

- ▶ **\$ `gdbserver 0.0.0.0:9999 -attach 112`**

- ▶ Attach agent to the process with PID 112 and wait for the debugger to connect on TCP port 9999

Remote Debug Example

- ▶ On the host:
 - ▶ **\$ gdb my_prog**
 - ▶ Start GDB
 - ▶ (gdb) set solib-absolute-prefix /dev/null
 - ▶ (gdb) set solib-search-path /path/to/target/libs
 - ▶ Set host path to target libraries with debug information
 - ▶ **\$ target remote 192.1.2.3:9999**
 - ▶ Connect to GDBServer on IP 192.1.2.3 port 9999

Remote Debugging Tips

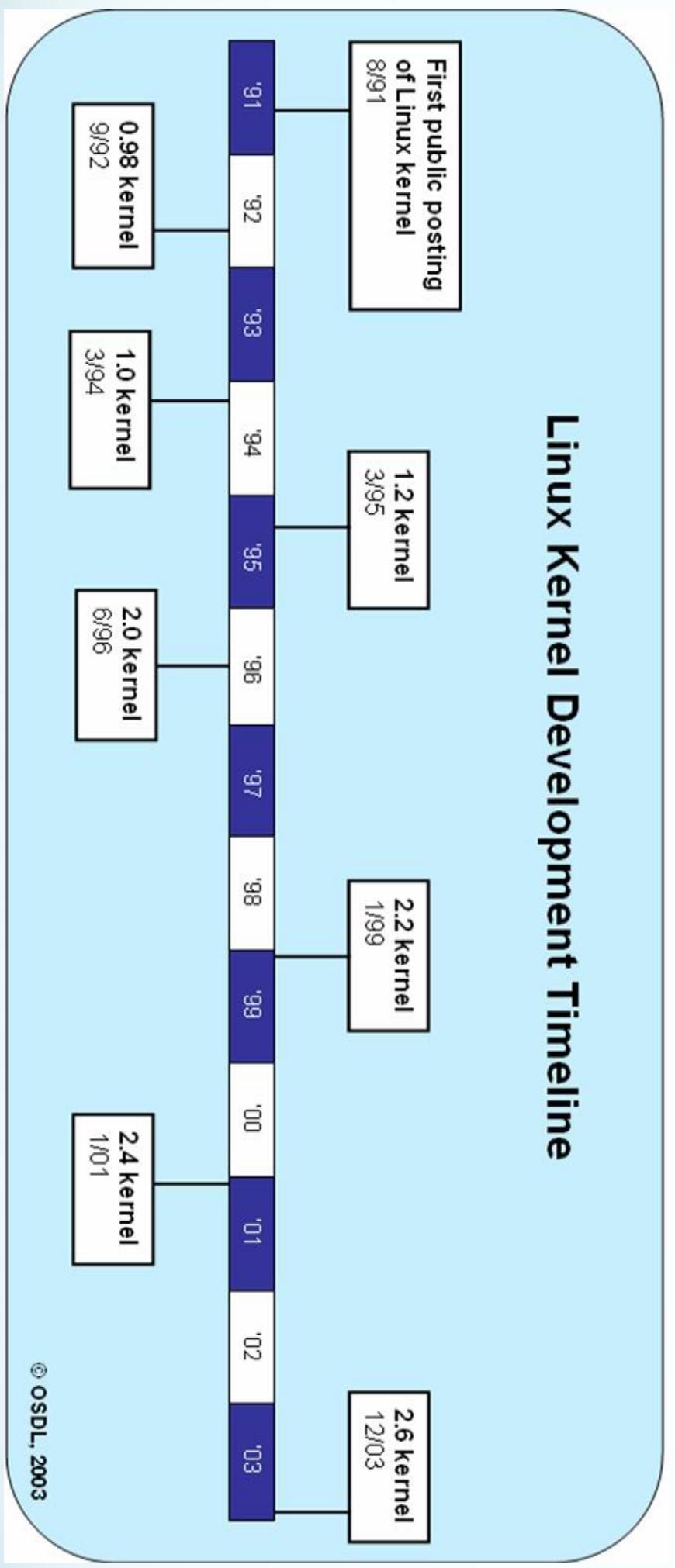
- ▼ Your first automatic breakpoint will be before `main()` in the guts of the C library
 - ▼ So just do `break main` and `continue`
- ▼ If the path to the copy of target libraries on the host is wrong or the target and host files do not match, you will see many strange errors
- ▼ If the target GDBserver is not installed or the file `libpthread_db.so` is missing, you will not be able to see or debug threads

Embedded Linux Driver Development

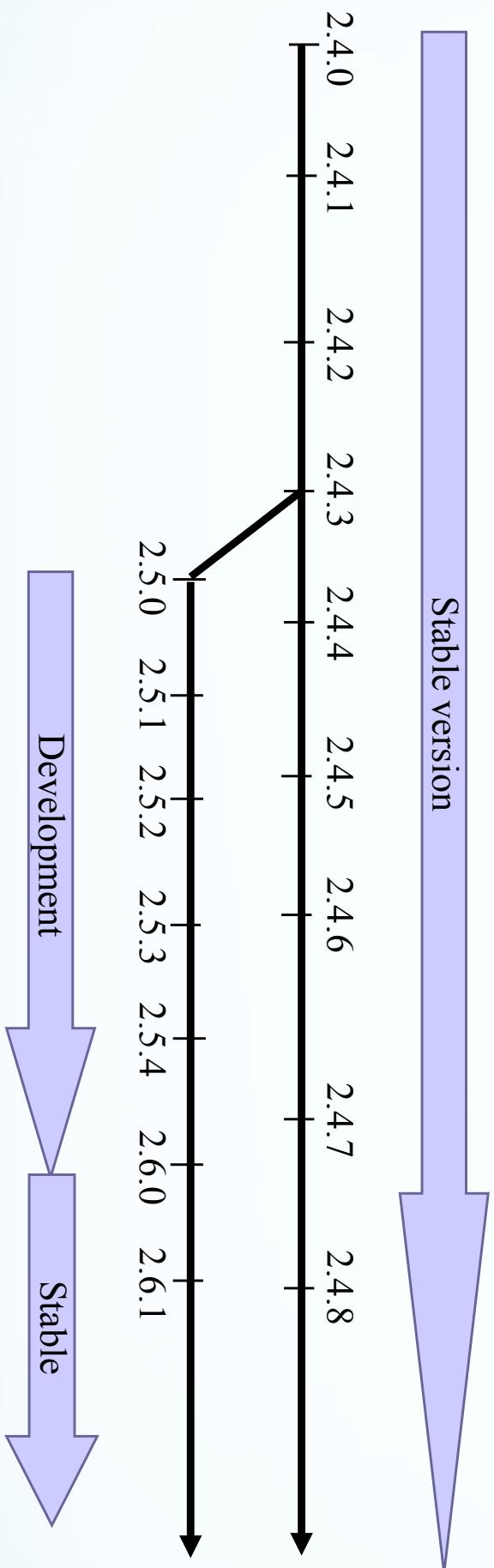
- Kernel Overview
- Linux Features

Linux Kernel Development Timeline

Linux Kernel Development Timeline

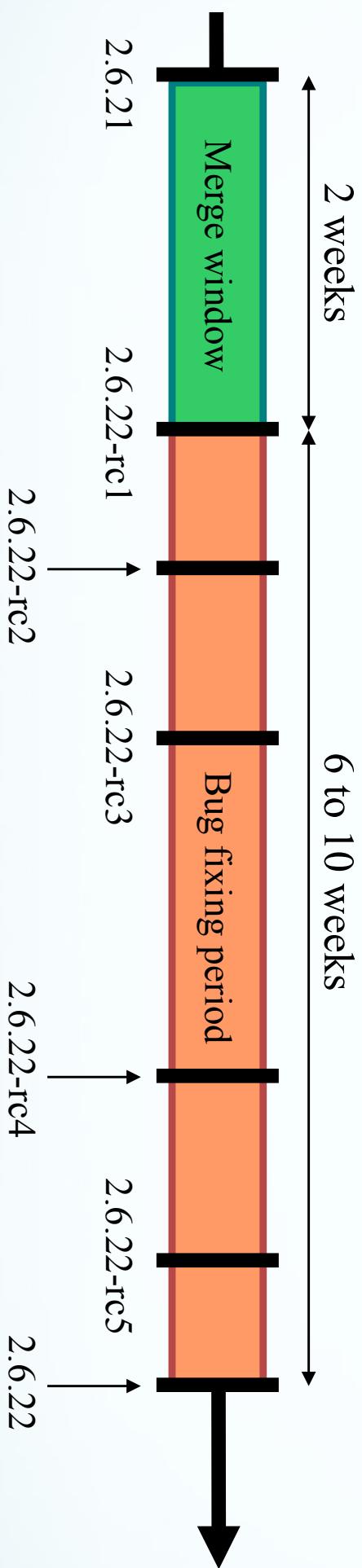


Until 2.6



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

From 2.6 onwards



Linux Kernel Key Features

- ▼ Portability and hardware support
Runs on most architectures.
- ▼ Scalability
Can run on super computers as well as on tiny devices (4 MB of RAM is enough).
- ▼ 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.
- ▼ Compliance to standards and interoperability.
- ▼ Exhaustive networking support.
- ▼ Easy to program
You can learn from existing code. Many useful resources on the net.

No stable Linux internal API

- ▶ Of course, the external API must not change (system calls, `/proc`, `/sys`), as it could break existing programs. New features can be added, but kernel developers try to keep backward compatibility with earlier versions, at least for 1 or several years.
- ▶ The internal kernel API can now undergo changes between two `2.6.x` releases. A stand-alone driver compiled for a given version may no longer compile or work on a more recent one.
See [Documentation/stable api nonsense.txt](#)
in kernel sources for reasons why.
- ▶ Whenever a developer changes an internal API, (s)he also has to update all kernel code which uses it. Nothing broken!
- ▶ Works great for code in the mainline kernel tree.
Difficult to keep in line for out of tree or closed-source drivers!

Supported Hardware 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

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

Linux Sources Structure (2)

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

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 **printf()**, **memset()**, **kmalloc()** ...

Managing Endianess

- Linux supports both little and big endian architectures
- ▶ Each architecture defines `__BIG_ENDIAN` or `__LITTLE_ENDIAN` in `<asm/byteorder.h>`
 - Can be configured in some platforms supporting both.
- ▶ To make your code portable, the kernel offers conversion macros (that do nothing when no conversion is needed). Most useful ones:

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

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>
```

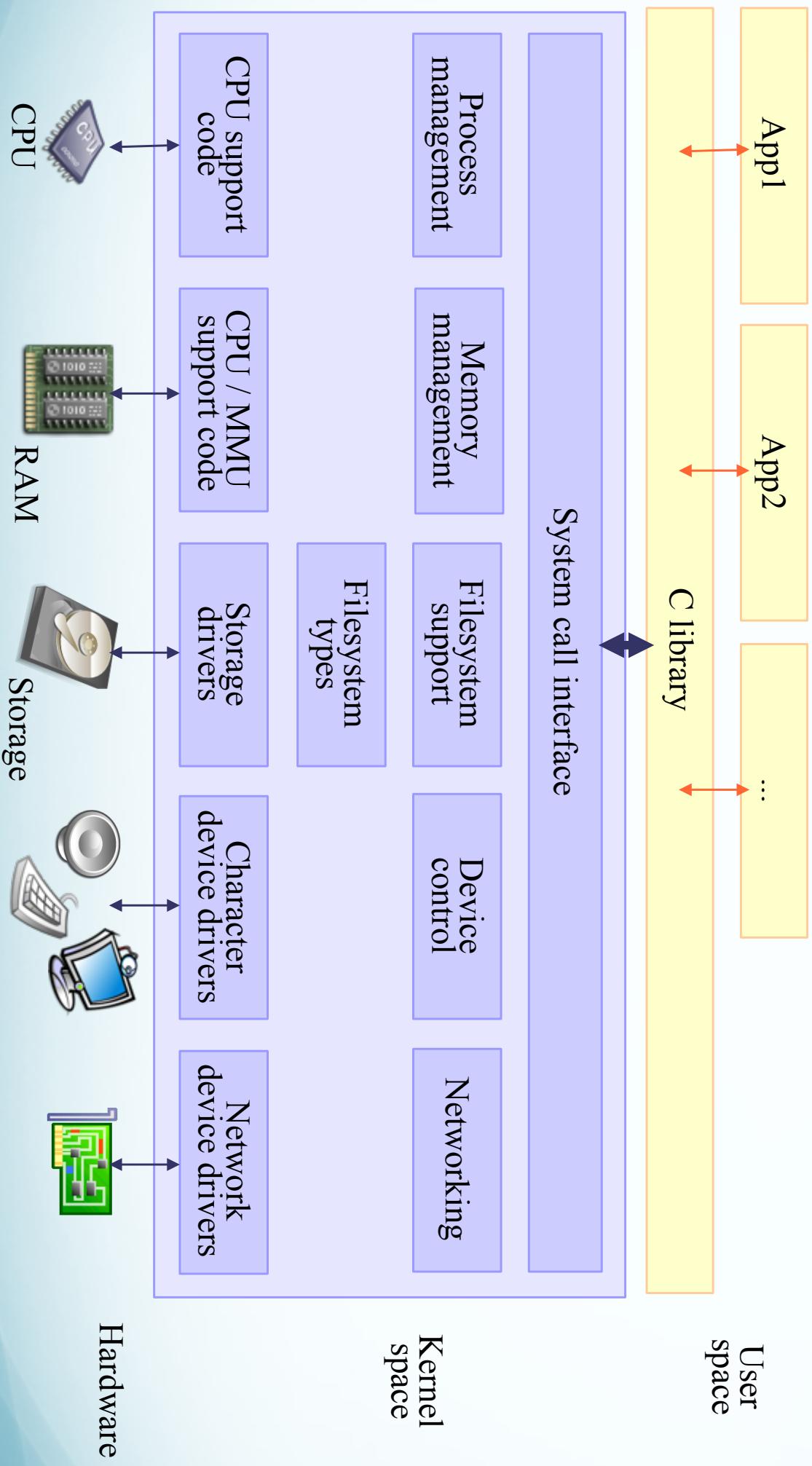
- See [Documentation/CodingStyle](#) for more guidelines

Embedded Linux Driver Development

Kernel Overview

Supervisor mode and the sys call interface

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 tasks needs to use a kernel service, it will make a “System Call”.
- ▶ The C library places parameters and number of system call in registers and then issues a special trap instruction.
- ▶ The trap atomically changes the ring level to supervisor mode and the sets the instruction pointer to the kernel.
- ▶ The kernel will find the required system called via the system call table and execute it.
- ▶ Returning from the system call does not require a special instruction, since in supervisor mode the ring level can be changed directly.

Make a call

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

- ▶ Input and output parameters using hardware registers

- ▶ Example (MIPS)

```
int mygetpid()
```

```
{
```

```
asm volatile(
```

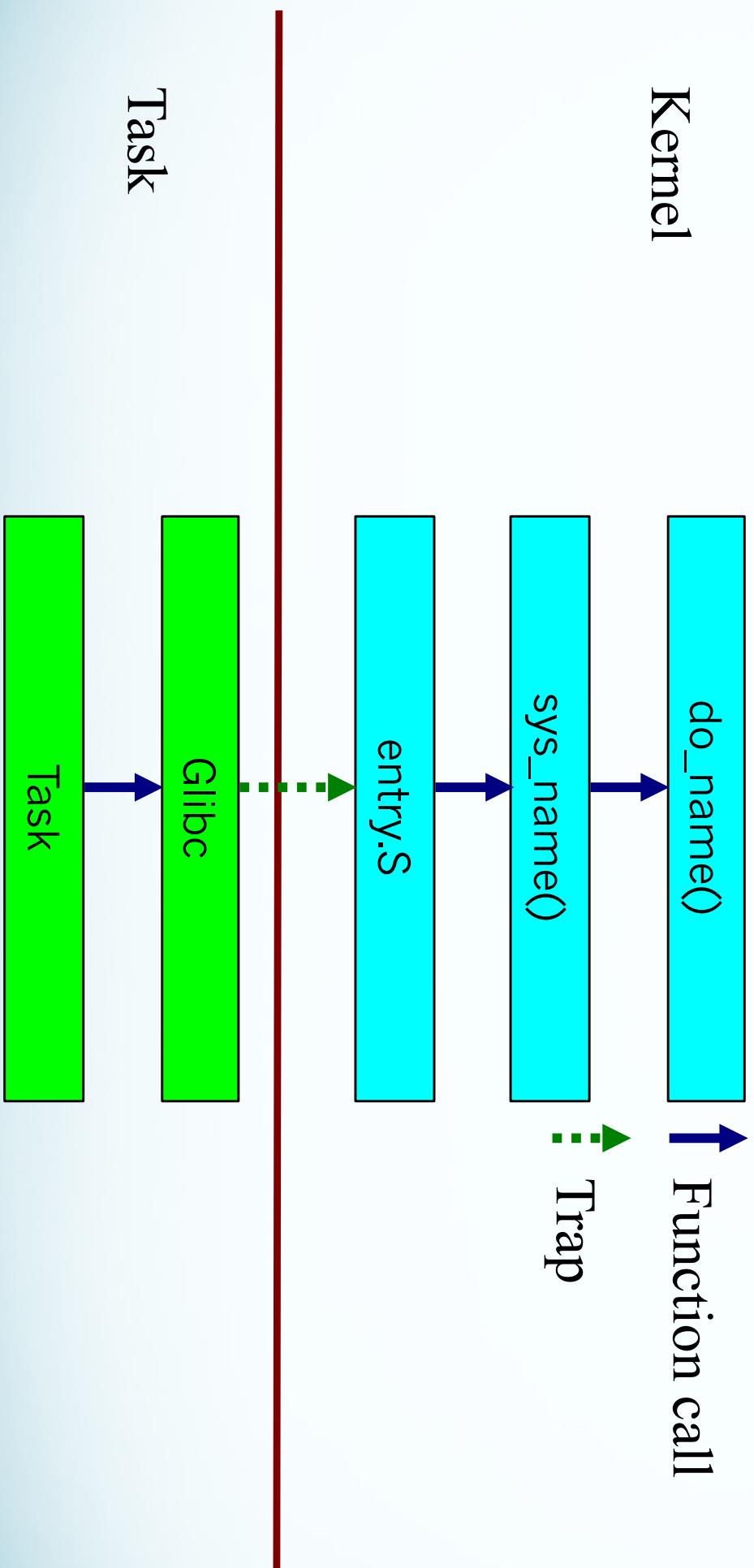
```
“li $v0,4020\n\t”
```

```
“syscall\n\t”
```

```
);
```

```
}
```

Linux System Call Path



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`

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");
}

```

__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**
 - ▶ **Proprietary**
Non free products
- ▶ **Dual BSD/GPL**
GNU Public License v2
or BSD license choice
- ▶ **Dual MPL/GPL**
GNU Public License v2
or Mozilla license choice

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
```

Either

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

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

Kernel Log

- ▶ Printing to the kernel log is done via the `printf()` 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.

printf()

- ▶ The **printf** 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:
- ```
printf(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

▼ **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)

▼ **dmesg**

Found in all systems

Displays the kernel log buffer

# Using the Module

- Need to be logged as **root**
- ▼ Load the module:  
`insmod ./hello.ko`
- ▼ You will see the following in the kernel log:  
**Good morrow**  
to this fair assembly
- ▼ Now remove the module:  
`rmmod hello`
- ▼ You will see:  
**Alas, poor world, what treasure  
hast thou lost!**

# Module Utilities (1)

## modinfo <module\_name>

**modinfo <module\_path>.ko**

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

## insmod <module\_name>

**insmod <module\_path>.ko**

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

# Module Utilities (2)

## modprobe <module\_name>

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

## lsmod

Displays the list of loaded modules  
Compare its output with the contents of  
`/proc/modules!`

# Module Utilities (3)

## ▶ `rmmod <module_name>`

Tries to remove the given module

## ▶ `modprobe -r <module_name>`

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

# 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>]`

# Embedded Linux Driver Development

Driver Development  
Module Parameters

# Hello Module with Parameters

Thanks to  
Jonathan Corbet  
for the example!

```
/* 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);
```

Example available on [http://free-electrons.com/doc/c/hello\\_param.c](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
```

# 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!) */`  
    `/* for /sys/module/<module_name>/<param>`  
        `0: no such module parameter value file */`  
    `);`
- Example
- `int irq=5;`  
`module_param(irq, int, S_IRUGO);`

# 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

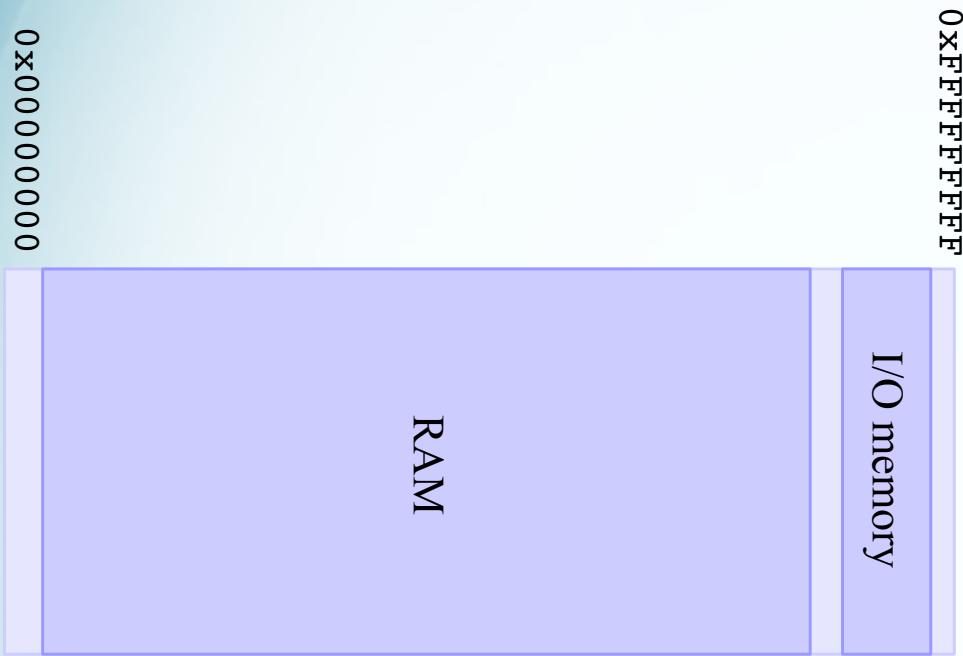
```
static int count;
static int base[MAX_DEVICES] = { 0x820, 0x840 };
module_param_array(base, int, &count, 0);
```

# Embedded Linux Driver Development

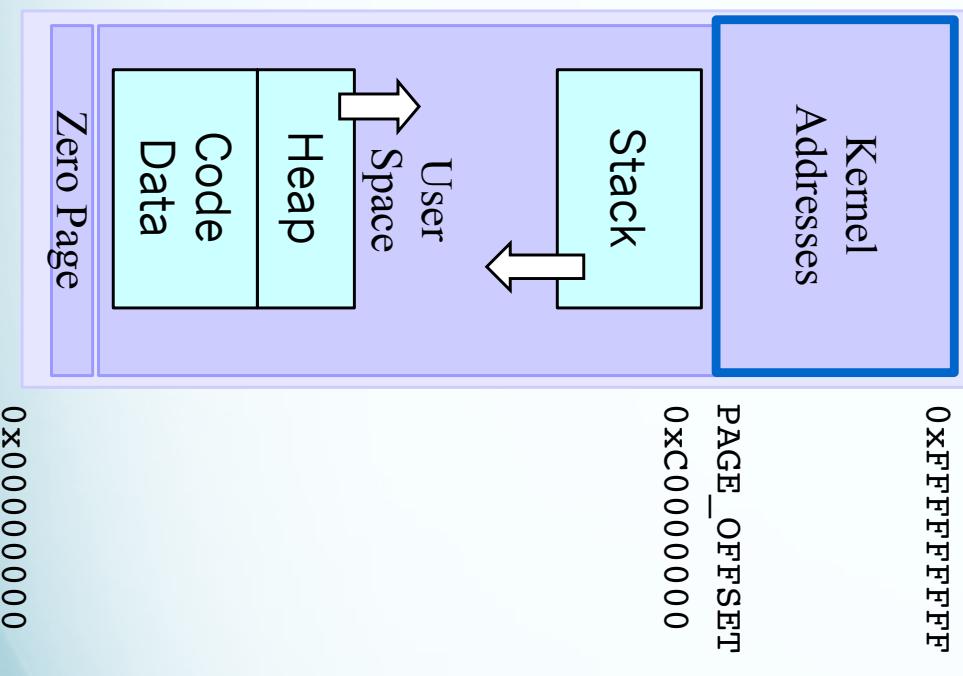
Driver Development  
Memory Management

# 3:1 Virtual Memory Map

## Physical address space



## Virtual address space



# Address Types

## Physical address

- Physical memory as seen from the CPU, without MMU<sup>1</sup> translation.

## Bus address

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

## Virtual address

- Memory as seen from the CPU, with MMU<sup>1</sup> translation.

<sup>1</sup> 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.

# 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 (explained in a few pages)

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

- ▶ Example: ([drivers/infiniband/core/cache.c](#))  
struct ib\_update\_work \*work;  
work = kmalloc(sizeof \*work, GFP\_ATOMIC);  
...  
kfree(work);

# Kmalloc features

- ▶ Quick (unless it's blocked waiting for memory to be freed).
- ▶ Doesn't initialize the allocated area.
- ▶ 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).
- ▶ Minimum allocation: 32 or 64 bytes (page size dependent).



# Main kmalloc flags (1)

- Defined in [include/linux/gfp.h](#) (GFP: [get free pages](#))
- ▼ [GFP\\_KERNEL](#)  
Standard kernel memory allocation. May block. Fine for most needs.
- ▼ [GFP\\_ATOMIC](#)  
RAM allocated from code which is not allowed to block (interrupt handlers) or which doesn't want to block (critical sections). Never blocks.
- ▼ [GFP\\_USER](#)  
Allocates memory for user processes. May block. Lowest priority.

# Allocating by pages

- More appropriate when you need big slices of RAM:
- ▼ A page is usually **4K**, but can be made greater in some architectures (**sh**, **mips**: **4**, **8**, **16** or **64K**, but not configurable in **i386** or **arm**).
- ▼ **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 int order);**  
Returns a pointer on an area of several contiguous pages in physical RAM.  
**order:  $\log_2(<\text{number\_of\_pages}>)$**   
If variable, can be computed from the size with the **get\_order** function.  
Maximum: 8192 KB (**MAX\_ORDER=11** in **include/linux/mmzone.h**),  
except in a few architectures when overwritten with **CONFIG\_FORCE\_MAX\_ZONEORDER**.

# Freeing pages

- ▶ `void free_page(unsigned long addr);`
  - ▶ `void free_pages(unsigned long addr,  
                      unsigned int 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, sizet count);`  
Fills a region of memory with the given value.
- ◀ `void * memcpy(void * dest, const void *src, sizet count);`  
Copies one area of memory to another.  
Use memmove with overlapping areas.
- ◀ Lots of functions equivalent to standard C library ones defined in include/linux/string.h and in include/linux/kernel.h (sprintf, etc.)

# Memory Management - Summary

- Small allocations
  - Bigger allocations
    - ▶ `kmalloc`, `kzalloc`  
(and `kfree!`)
    - ▶ `_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

# Requesting I/O Memory

/proc/iomem example

00000000-0009efff : System RAM

0009f000-0009ffff : reserved

000a0000-000bffff : Video RAM area

000c0000-000cffff : Video ROM

000f0000-000fffff : System ROM

00100000-3ffadfff : System RAM

00100000-0030afff : Kernel code

0030b000-003b4bff : Kernel data

3ffae000-3fffffff : reserved

40000000-400003ff : 0000:00:1f.1

40001000-40001fff : 0000:02:01.0

40001000-40001fff : yenta\_socket

40002000-40002fff : 0000:02:01.1

40002000-40002fff : yenta\_socket

40400000-407fffff : PCI CardBus

#03

40800000-40bfffff : PCI CardBus

#03

40c00000-40ffffff : PCI CardBus

#07

41000000-413fffff : PCI CardBus #07

a0000000-a0000fff : pcmcia\_socket0

a0001000-a0001fff : pcmcia\_socket1

e0000000-e7ffffff : 0000:00:00.0

e8000000-efffffff : PCI Bus #01

e8000000-efffffff : 0000:01:00.0

► Equivalent functions with the same interface

► struct resource \*request\_mem\_region(

    unsigned long start,

    unsigned long len,

    char \*name);

► void release\_mem\_region(

    unsigned long start,

    unsigned long len);

# Mapping I/O Memory into Virtual Memory

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

- ▶ The `ioremap()` functions satisfy this need:

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

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

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

# Accessing I/O Memory

- ▶ Directly reading from or writing to addresses returned by **ioremap()** ("pointer dereferencing") may not work on some architectures.
- ▶ Use the below functions instead. They are always portable and safe:

```
unsigned int ioread8(void *addr); (same for 16 and 32)
void iowrite8(u8 value, void *addr); (same for 16 and 32)
```
- ▶ To read or write a series of values:

```
void ioread8_rep(void *addr, void *buf, unsigned long count);
void iowrite8_rep(void *addr, const void *buf, unsigned long count);
```
- ▶ Other useful functions:

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

# Embedded Linux Driver Development

Driver Development

Character Drivers

# Drivers

- ◀ Except for storage device drivers, most drivers for devices with input and output flows are implemented as character drivers.
- ◀ So, most drivers you will face will be character drivers
- You will regret if you sleep during this part!



# Character device files

- ▼ Accessed through a sequential flow of individual characters
- ▼ Character devices can be identified by their **c** type (**ls -l**):

```
crw-rw---- 1 root uucp 4, 64 Feb 23 2004 /dev/ttys0
crw--w---- 1 jdoe tty 136, 1 Feb 23 2004 /dev/pts/1
crw----- 1 root root 13, 32 Feb 23 2004
/dev/input/mouse0
crw-rw-rw- 1 root root 1, 3 Feb 23 2004 /dev/null
```
- ▼ Example devices: keyboards, mice, parallel port, IrDA, Bluetooth port, consoles, terminals, sound, video...

# Device major and minor numbers

- As you could see in the previous examples, device files have 2 numbers associated to them:
  - First number: *major* number
  - Second number: *minor* number
- Major and minor numbers are used by the kernel to bind a driver to the device file. Device file names don't matter to the kernel!
- To find out which driver a device file corresponds to, or when the device name is too cryptic, see [Documentation/devices.txt](#).

# Devices

- Registered devices are visible in `/proc/devices`:

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

Can be used to find free major numbers



# Device file creation

- ▼ Device files are not created when a driver is loaded.
- ▼ They have to be created in advance:  
`mknod /dev/<device> [c|b] <major> <minor>`
- ▼ Examples:  
`mknod /dev/ttyS0 c 4 64`  
`mknod /dev/hd1 b 3 1`

# Creating a Character Driver

## User-space

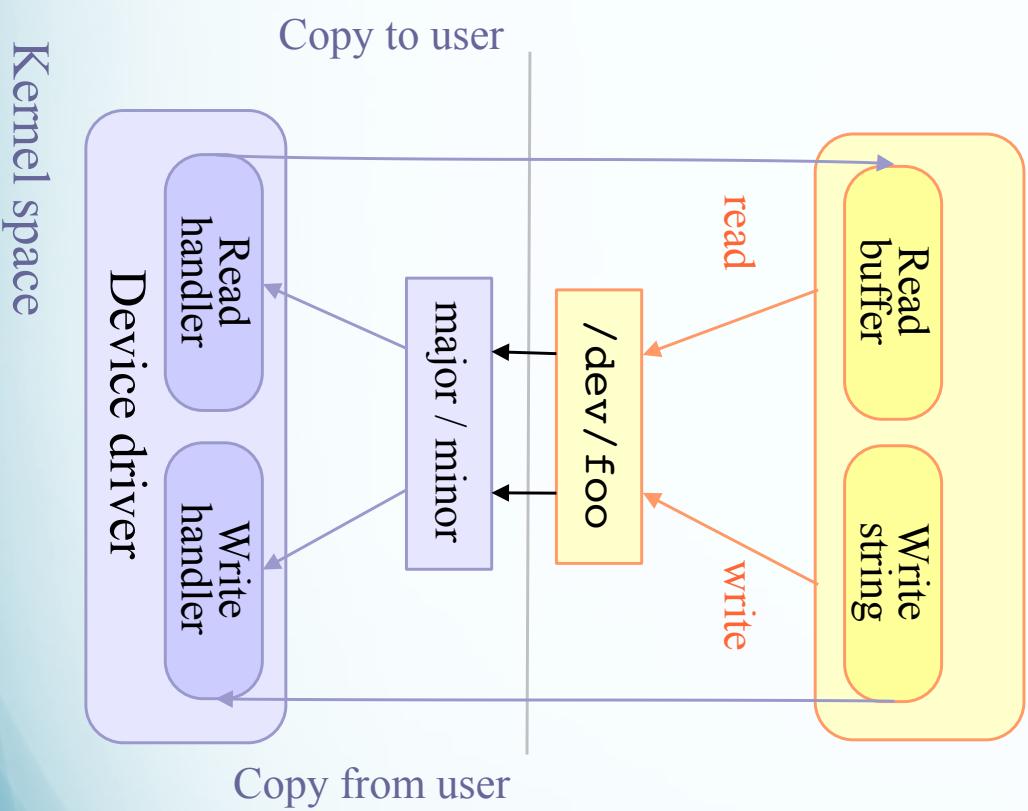
- User-space needs

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

## The kernel needs

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

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



# Declaring a Character Driver

- Device number registration

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

## File operations registration

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

# dev\_t Structure

- Kernel data structure to represent a major/minor pair.
- ▼ Defined in <linux/kdev\_t.h>  
Linux 2.6: 32 bit size (major: 12 bits, minor: 20 bits)
- ▼ Macro to create the structure:  
`MKDEV( int major, int minor );`
- ▼ Macros to extract the numbers:  
`MAJOR(dev_t dev);`  
`MINOR(dev_t dev);`

# Numbers

- `#include <linux/fs.h>`
- ```
int register_chrdev_region(
    dev_t from,
    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");
...
}
```

# Numbers

- Safer: have the kernel allocate free numbers for you!
- ```
#include <linux/fs.h>
```
- ```
int alloc_chrdev_region(/* Output: starting device
 dev_t *dev,
 number */ /* Starting minor number, usually 0
 unsigned baseminor, /* 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");
...
}
```

File Operations (1)

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

- ▼ **int (*open)(**
 struct inode *, /* Corresponds to the device file */
 struct file *); /* Corresponds to the open file
descriptor */
- Called when user-space opens the device file.
- ▼ **int (*release)(**
 struct inode *,
 struct file *);
- Called when user-space closes the file.

The file Structure

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

▼ `mode_t f_mode;`

The file opening mode (`FMODE_READ` and/or `FMODE_WRITE`)

▼ `loff_t f_pos;`

Current offset in the file.

▼ `struct file_operations *f_op;`

Allows to change file operations for different open files!

▼ `struct dentry *f_dentry`

Useful to get access to the inode: `filp->f_dentry->d_inode.`

▼ To find the minor number use:

`MINOR(filp->f_dentry->d_inode->i_rdev)`

File Operations (2)

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

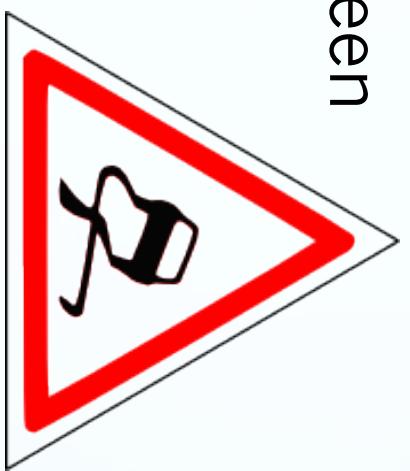
Called when user-space reads from the device file.

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

Called when user-space writes to the device file.

Exchanging Data With User-Space

- 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

- 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 (*unlocked_ioctl) (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 hldata address corresponds to a device I/O memory area */
    /* of size hldata_size, obtained with ioremap() */

    int remaining_bytes;

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

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

    if (copy_to_user(buf /* to */, *ppos+hldata /* 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 hwdta corresponds to a physical address range */
    /* of size hwdta_size, obtained with ioremap() */

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

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

    if (copy_from_user(*ppos+hwdta /* 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

- 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.

(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;
```

(2)

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

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

- ▶ Example (continued):

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

Character Device Unregistration

- ▶ First delete your character device:
`void cdev_del(struct cdev *p);`
- ▶ Then, and only then, free the device number:
`void unregister_chrdev_region(dev_t from,
unsigned count);`
- ▶ Example (continued):
`cdev_del(acme_cdev);
unregister_chrdev_region(acme_dev,
acme_count);`

Char Driver Example Summary (1)

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

static int acme_count=1;
static dev_t acme_dev;

static struct cdev *acme_cdev;

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

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

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

Char Driver Example Summary (2)

```

static int __init acme_init(void)
{
    int err;
    hldata = ioremap(PHYS_ADDRESS,
                      hdata_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;
    }

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

```

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()`

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!

System user

- Open the device file, read, write, or send ioctl's to it.

Kernel

- Executes the corresponding file operations

Kernel

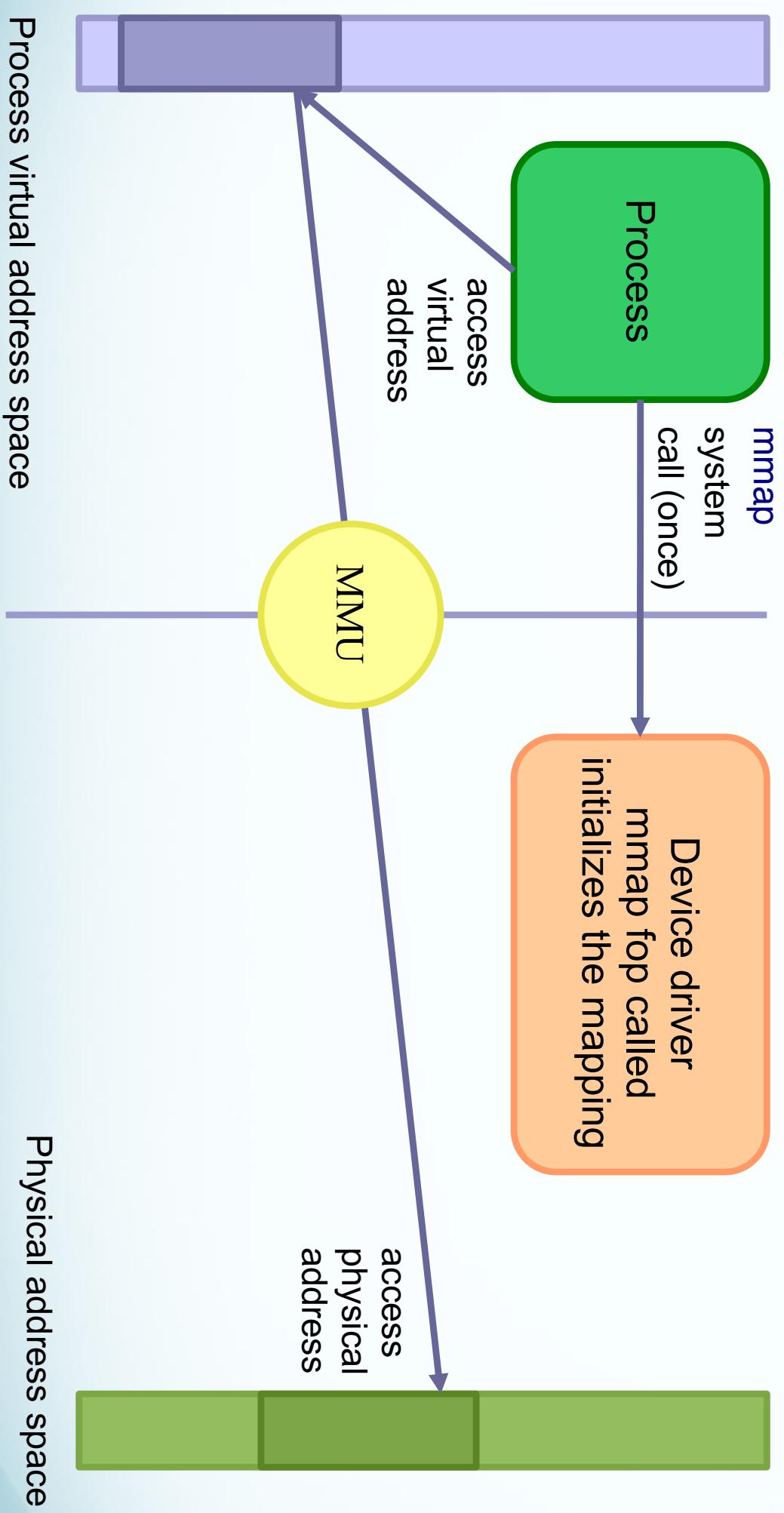
User-space

Kernel

Embedded Linux Driver Development

Driver Development
mmap()

mmap overview



How to Implement mmap() - Kernel-Space

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

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

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

remap_pfn_range()

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

- ▶ `#include <linux/mm.h>`

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

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

Implementation

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

 if (size > ACME_SIZE)
 return -EINVAL;

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

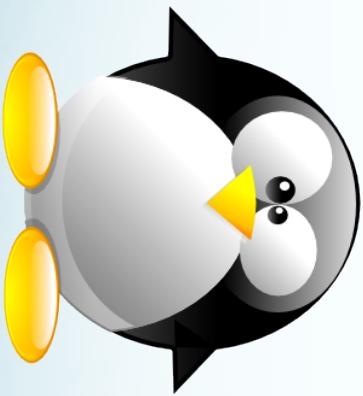
mmap Summary

- ▶ The device driver is loaded.
It defines an `mmap` file operation.
 - ▶ A user space process calls the `mmap` system call.
 - ▶ The `mmap` file operation is called.
It initializes the mapping using the device physical address.
 - ▶ The process gets a starting address to read from and write to (depending on permissions).
 - ▶ The MMU automatically takes care of converting the process virtual addresses into physical ones.
- Direct access to the hardware!
No expensive `read` or `write` system calls!

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.

Kgdb

- ▼ Inside the kernel from 2.6.26
- ▼ The execution of the kernel is fully controlled by **gdb** from another machine, connected through a serial line.
- ▼ Can do almost everything, including inserting breakpoints in interrupt handlers.
- ▼ In version 2.6.35 we can find also kdb
- ▼ The key difference between Kgdb and KDB is that using Kgdb requires an additional computer to run a gdb frontend, and you can do source level debugging. KDB, on the other hand, can be run on the local machine and can be used to inspect the system, but it doesn't do source-level debugging

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.

Kernel Oops Example

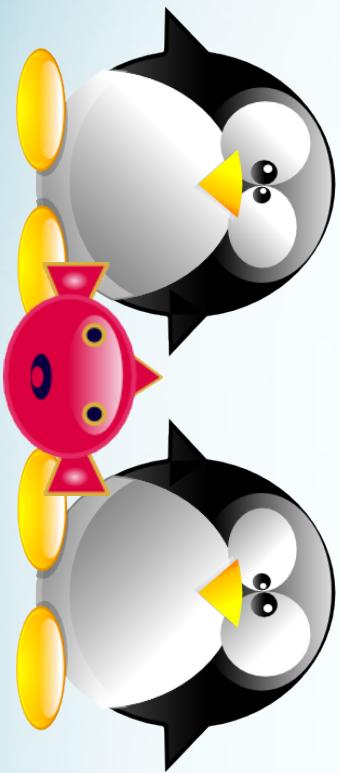
```
unable to handle kernel NULL pointer dereference at virtual address
printing eip: d8d4d545
*pde = 00000000
Oops: 0000
CPU:    0
EIP: 0010:[<d8d4d545>] Tainted: PF
EFLAGS: 00013286
eax: c8b078c0 ebx: c8b078c0 ecx: 00000019 edx: c8b078c0
esi: 00000000 edi: 00000000 ebp: bfffff75c esp: c403bf84
ds: 0018 es: 0018 ss: 0018
Process vmware (pid: 5194, stackpage=c403b000)
Stack:[c8b078c0 00000000 bfffff35c 00000000 c0136d64 c403bfa4 c8b078c0
Call trace:[sys_fstat64+100/112] [filp_close+53/112] [sys_close+67/96]

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
Locking



Linux mutexes

- ▶ The main locking primitive since Linux 2.6.16.
Better than counting semaphores when binary ones are enough.
- ▶ Mutex definition:
`#include <linux/mutex.h>`
- ▶ Initializing a mutex statically:
`DEFINE_MUTEX(name);`
- ▶ Or initializing a mutex dynamically:
`void mutex_init(struct mutex *lock);`

locking and unlocking

mutexes

▼ **void mutex_lock (struct mutex *lock);**

Tries to lock the mutex, sleeps otherwise.

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

▼ **int mutex_lock_killable (struct mutex *lock);**

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

▼ **int mutex_lock_interruptible (struct mutex *lock);**

Same, but can be interrupted by any signal.

▼ **int mutex_trylock (struct mutex *lock);**

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

▼ **int mutex_is_locked(struct mutex *lock);**

Just tells whether the mutex is locked or not.

▼ **void mutex_unlock (struct mutex *lock);**

Releases the lock. Do it as soon as you leave the critical section.

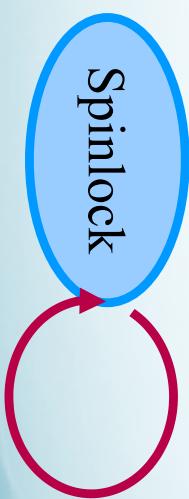
Spinlocks

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

- ▶ Originally intended for multiprocessor systems

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

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



Still locked?

Initializing spinlocks

- ▶ Static

```
spinlock_t my_lock =  
SPIN_LOCK_UNLOCKED;
```
- ▶ Dynamic

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

Using spinlocks (1)

- Several variants, depending on where the spinlock is called:
 - ▼ **void spin_[un]lock (spinlock_t *lock);**
Doesn't disable interrupts. Used for locking in process context (critical sections in which you do not want to sleep).
 - ▼ **void spin_lock_irqsave /
spin_unlock_irqrestore (spinlock_t
*lock, unsigned long flags);**
Disables / restores IRQs on the local CPU.
Typically used when the lock can be accessed in both process and interrupt context, to prevent preemption by interrupts.

Using spinlocks (2)

► `void spin_unlock_bh (spinlock_t *lock);`

Disables software interrupts, but not hardware ones.

Useful to protect shared data accessed in process context and in a soft interrupt ("bottom half"). No need to disable hardware interrupts in this case.

Note that reader / writer spinlocks also exist.

Avoiding Coherence Issues

Hardware independent

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

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

Hardware dependent

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

Safe on all architectures!

Alternatives to Locking

- As we have just seen, locking can have a strong negative impact on system performance. In some situations, you could do without it.
- By using lock-free algorithms like Read Copy Update (RCU). RCU API available in the kernel (See <http://en.wikipedia.org/wiki/RCU>).
- When available, use atomic operations.

Atomic Variables

- ▶ Useful when the shared resource is an integer value
- ▶ Even an instruction like `n++` is not guaranteed to be atomic on all processors!
- Header
- ▶ `#include <asm/atomic.h>`
- Type
- ▶ `atomic_t` contains a signed integer (use 24 bits only)
- Atomic operations (main ones)
- ▶ Set or read the counter:


```
atomic_set(atomic_t *v, int i);
int atomic_read(atomic_t *v);
```
- ▶ Operations without return value:


```
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_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(...);
```

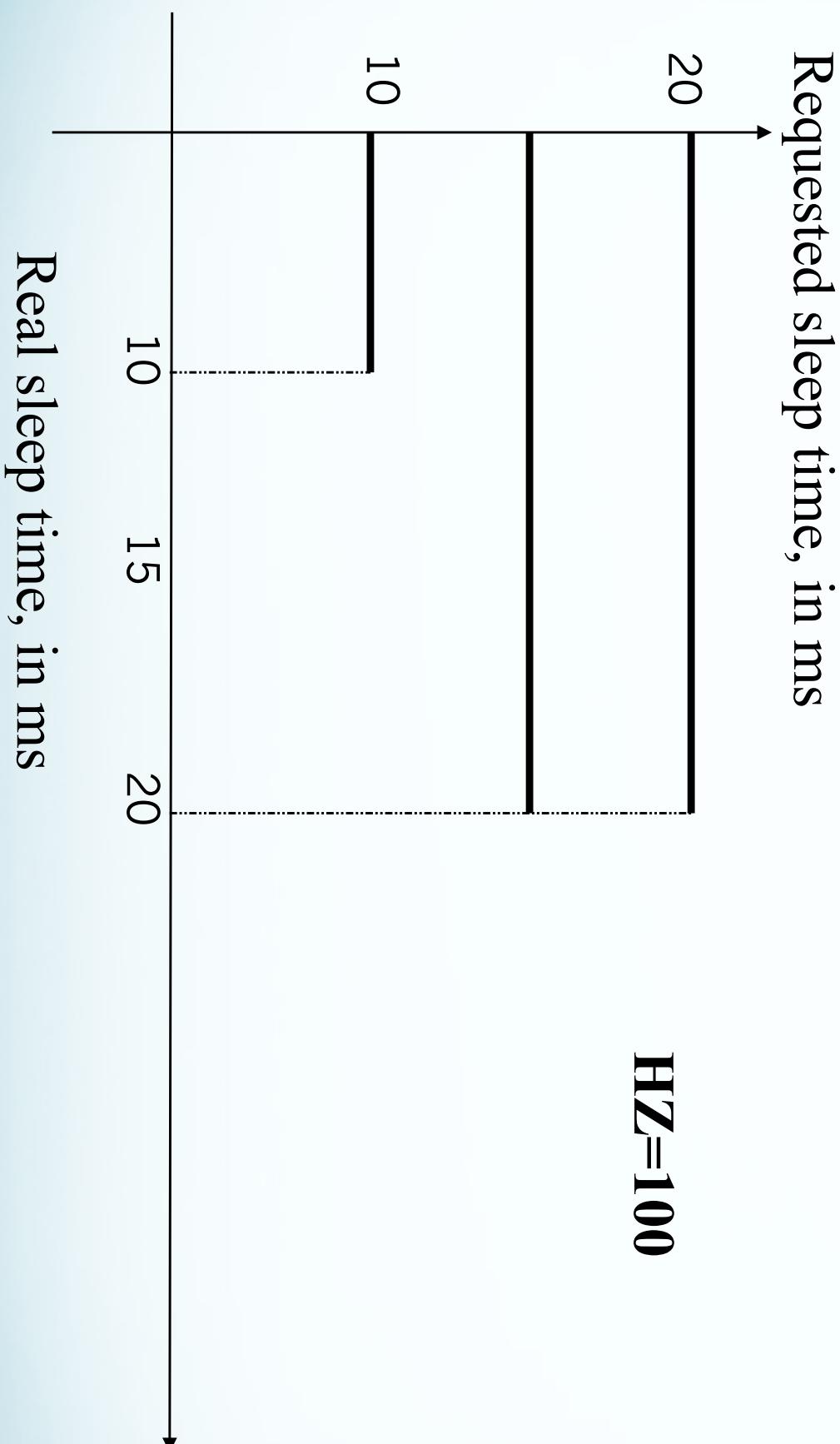
Embedded Linux Driver Development

How Time Flies

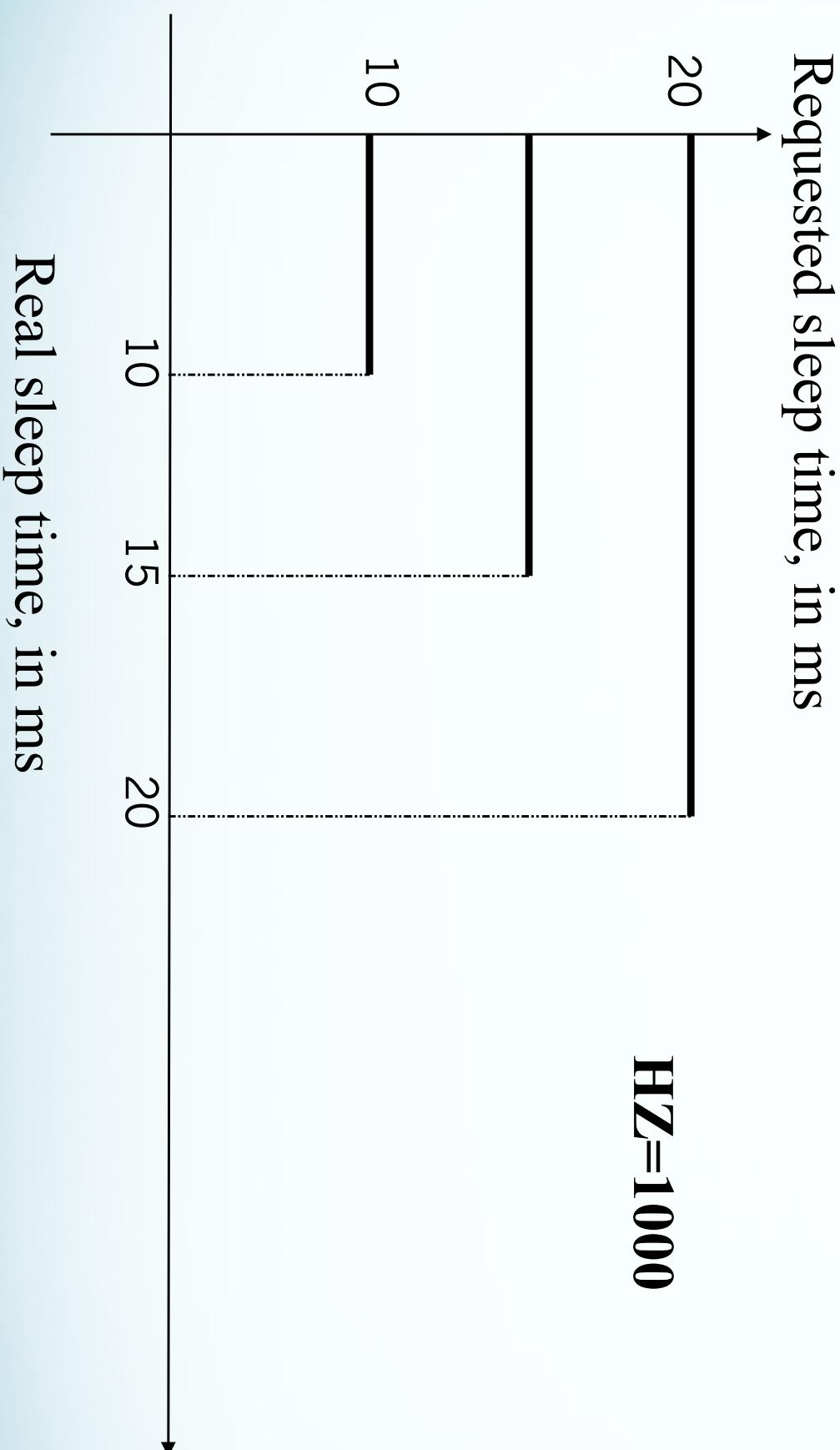
Timer Frequency

- 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.



Kernel (2.6.20)

High-Res Timers and Tickless

- ▶ 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 1-2 timer interrupts per second.

- ▶ 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);`

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.

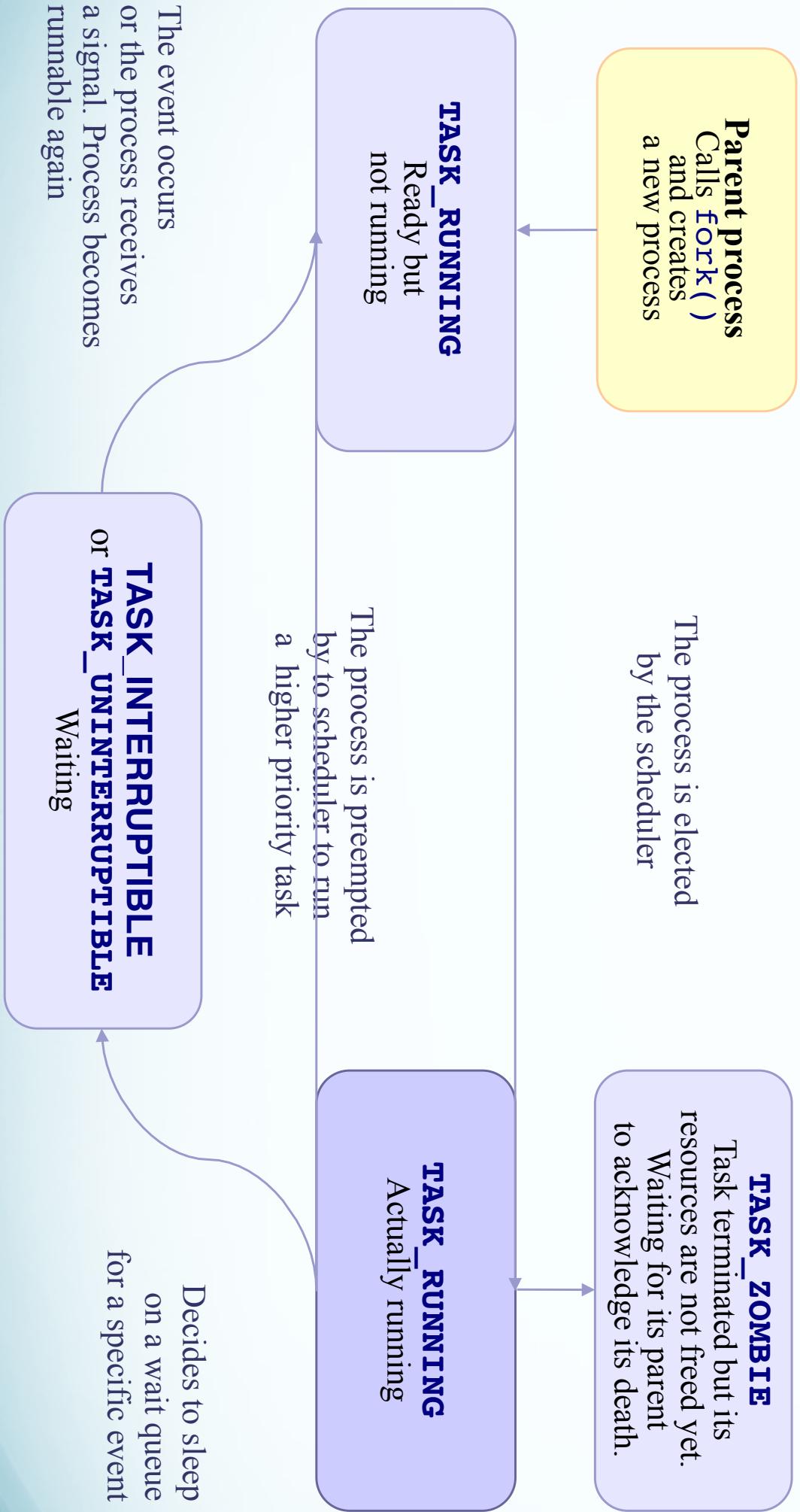
task_struct

- ▼ 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

Current Task

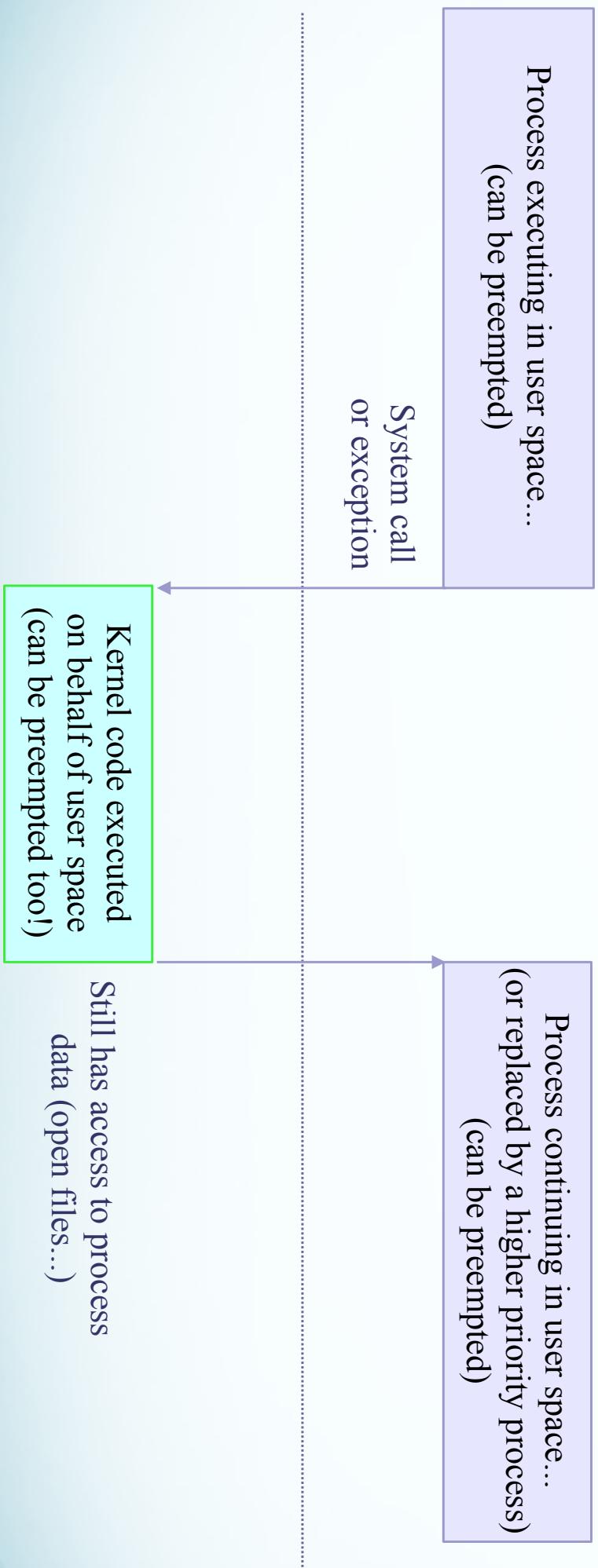
- ▶ **current** points to the current process **task_struct**
- ▶ When applicable – not valid in interrupt context.
- ▶ Current is a macro that appears to the programmer as a magical global variable which updated each context switch.
- ▶ Real value is either in register or computed from start of stack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

A Process Life



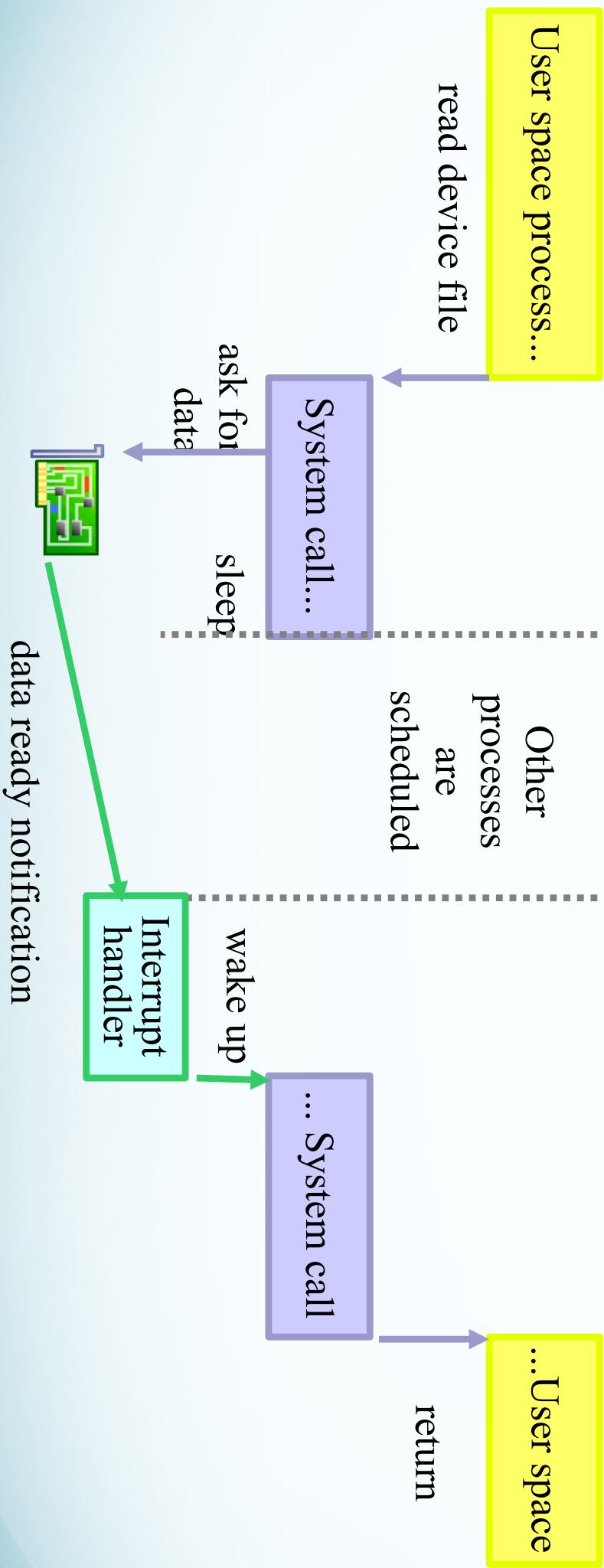
Process Context

User-space programs and system calls are scheduled together:



Sleeping

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



How to Sleep (1)

- Must declare a wait queue

- ▼ Static queue declaration

DECLARE_WAIT_QUEUE_HEAD (module_queue) ;

- ▼ Or dynamic queue declaration

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

How to Sleep (2)

- Several ways to make a kernel process sleep
 - ▶ **wait event(queue, condition);**
Sleeps until the given C expression is true.
Caution: can't be interrupted (can't kill the user-space process!)
 - ▶ **wait event killable(queue, condition); (Since Linux 2.6.25)**
Sleeps until the given C expression is true.
Can only be interrupted by a "fatal" signal (**SIGKILL**)
 - ▶ **wait event interruptible(queue, condition);**
Can be interrupted by any signal
 - ▶ **wait event timeout(queue, condition, timeout);**
Sleeps and automatically wakes up after the given timeout.
 - ▶ **wait event interruptible timeout(queue, condition, timeout);**
Same as above, interruptible.

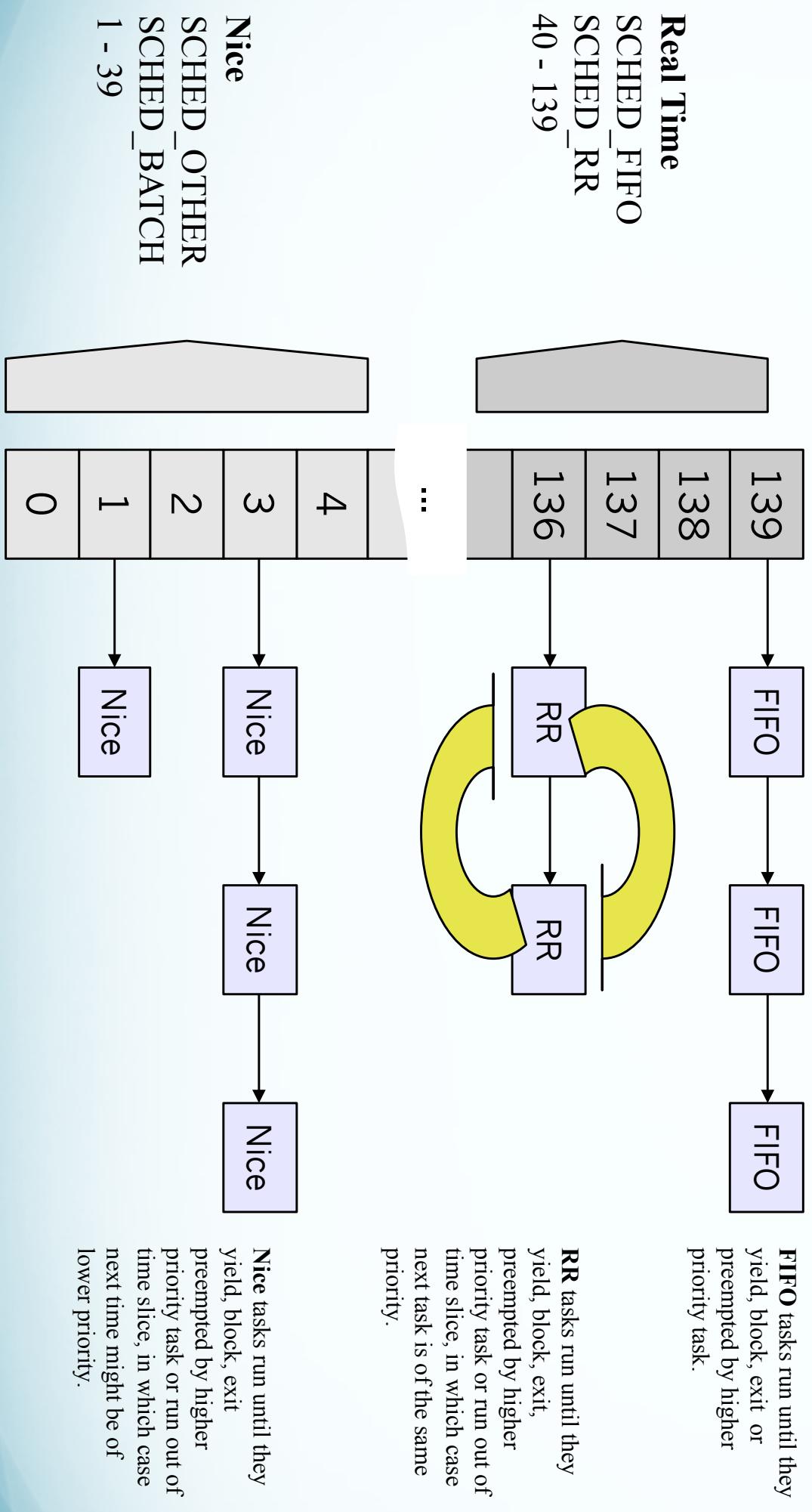
Waking Up!

- Typically done by interrupt handlers when data sleeping processes are waiting for are available.
- ▶ `wake_up(queue);`
Wakes up all the waiting processes on the given queue
- ▶ `wake_up_interruptible(queue);`
Wakes up only the processes waiting in an interruptible sleep on the given queue
- ▶ For all processes waiting in `queue`, condition is evaluated.
When it evaluates to true, the process is put back to the `TASK_RUNNING` state, and the `need_resched` flag for the current process is set.

When is Scheduling Run?

- Each process has a `need_resched` flag which is set:
 - ◀ After a process exhausted its time slice.
 - ◀ After a process with a higher priority is awakened.
- This flag is checked (possibly causing the execution of the scheduler):
 - ◀ When returning to user-space from a system call.
 - ◀ When returning from an interrupt handler (including the CPU timer).
- Scheduling also happens when kernel code explicitly calls `schedule()` or executes an action that sleeps.

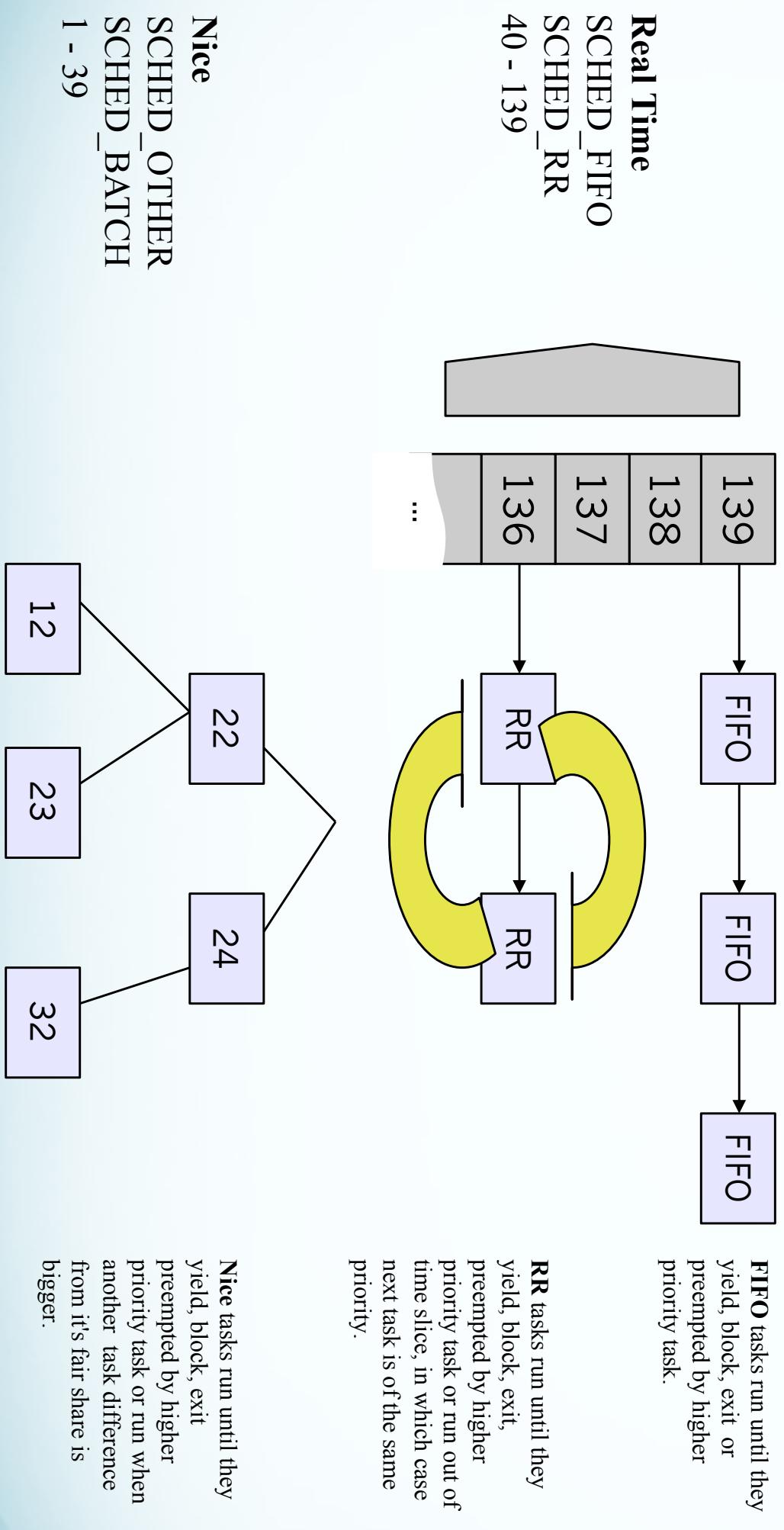
The Run Queues



Dynamic Priorities

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

CFS Work Queues and Tree



Nice tasks run until they yield, block, exit preemted by higher priority task or run when another task difference from its fair share is bigger.

The CFS Data structure

- ▼ The CFS holds all nice level tasks in a red-black tree, sorted according to the time the task needs to run to be balanced minus it's fair share of the CPU.
- ▼ Therefore, the leftmost task in the tree (smallest value) is the one which the scheduler should pick next.
- ▼ An adjustable granularity time guarantees against too often task switches.
- ▼ This red-black tree algorithm is $O(\log n)$, which is a small drawback, considering the previous scheduler was $O(1)$.

Embedded Linux Driver Development

Driver Development
Interrupt Management

Interrupt handler constraints

- ▶ Not run from a user context:
- ▶ Can't transfer data to and from user space (need to be done by system call handlers)
- ▶ Interrupt handler execution is managed by the CPU, not by the scheduler. **Handlers can't run actions that may sleep**, because there is nothing to resume their execution. In particular, need to allocate memory with `GFP_ATOMIC`.
- ▶ Have to complete their job quickly enough: they shouldn't block their interrupt line for too long.

Registering an interrupt handler (1)

- Defined in [include/linux/interrupt.h](#)

▼ `int request_irq(`

successful

unsigned int irq,

irqreturn_t handler,

unsigned long irq_flags,

const char * devname,

void *dev_id);

Returns 0 if

Requested irq channel

Interrupt handler

Option mask (see next page)

Registered name

Pointer to some handler data

Cannot be NULL and must be unique

for shared irqs!

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

▼ `dev_id` cannot be NULL and must be unique for shared irqs.

Otherwise, on a shared interrupt line,
`free_irq` wouldn't know which handler to free.

Registering an interrupt handler (2)

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

▼ IRQF DISABLED

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

▼ IRQF_SHARED

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

▼ IRQF_SAMPLE_RANDOM

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

Information On installed handlers

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

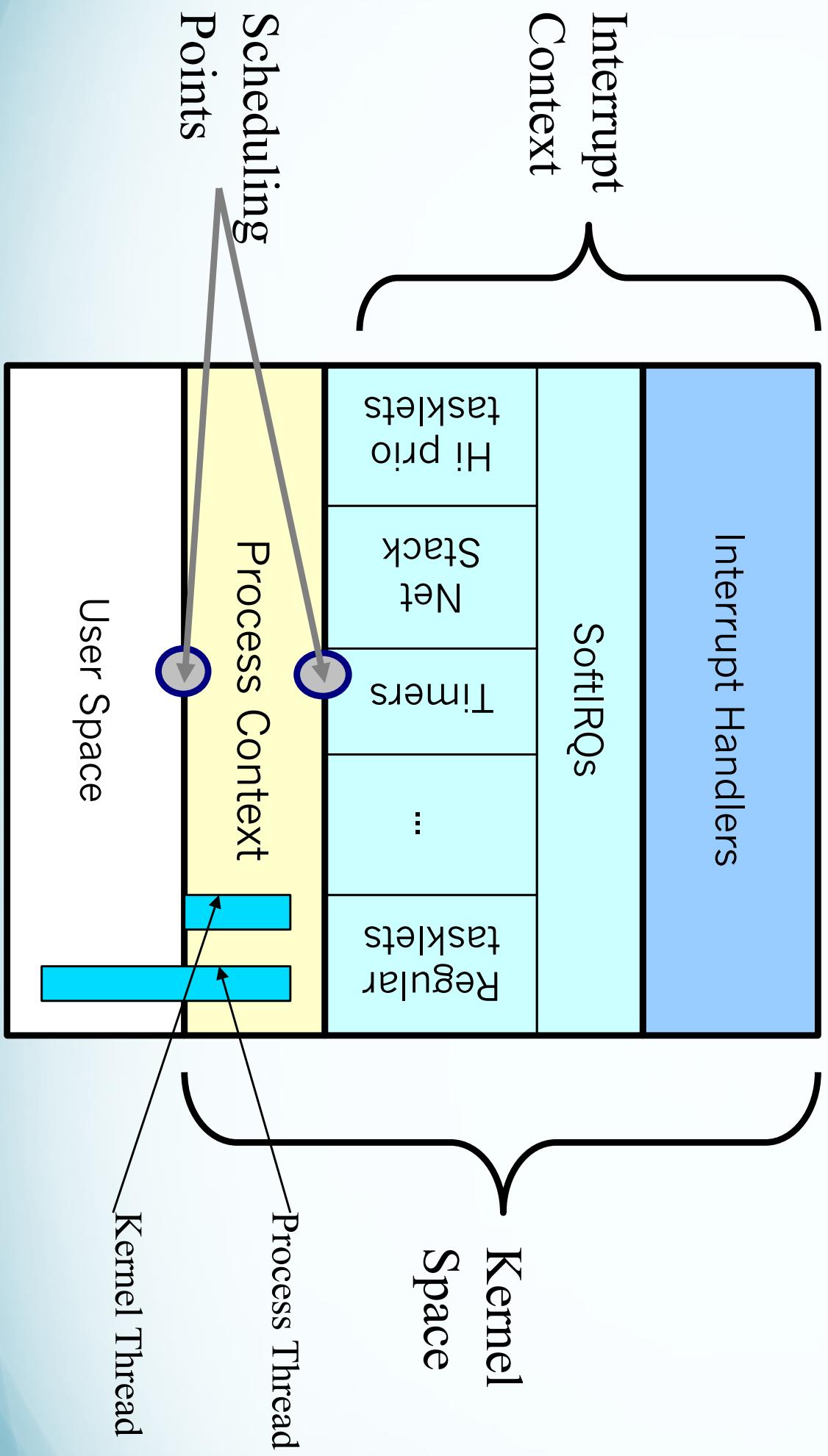
Interrupt handler prototype

- irqreturn_t (*handler) (
 int, // irq number of the current interrupt
 void *dev_id, // Pointer used to keep track
 // of the corresponding device.
 // Useful when several devices
 // are managed by the same module
);
- Return value:
- ▼ IRQ_HANDLED: recognized and handled interrupt
- ▼ IRQ_NONE: not on a device managed by the module. Useful to share interrupt channels and/or report spurious interrupts to the kernel.

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);`

Linux Contexts



Top half and bottom half processing (1)

- Splitting the execution of interrupt handlers in 2 parts
 - ▼ *Top half*: the interrupt handler must complete as quickly as possible. Once it acknowledged the interrupt, it just schedules the lengthy rest of the job taking care of the data, for a later execution.
 - ▼ *Bottom half*: completing the rest of the interrupt handler job. Handles data, and then wakes up any waiting user process.
Best implemented by *tasklets* (also called *soft irqs*).

top half and bottom half processing (2)

- ▶ Declare the tasklet in the module source file:

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

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

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

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

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

Handling Floods

- ▼ Normally, pending softirqs (including tasklets) will be run after each interrupt.
- ▼ A pending softirq is marked in a special bit field.
- ▼ The function that handles this is called `do_softirq()` and it is called by `do_IRQ()` function.
- ▼ If after `do_softirq()` called the handler for that softirq, the softirq is still pending (the bit is on), it will **not** call the softirq again.
- ▼ Instead, a low priority kernel thread, called `ksoftirqd`, is woken up. It will execute the softirq handler **when it is next scheduled**.

Disabling interrupts

- May be useful in regular driver code...

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

- Disabling interrupts on the local CPU:

```
unsigned long flags;  
local irq save(flags); // Interrupts disabled
```

...

```
local irq restore(flags); // Interrupts restored to their previous state.
```

Note: must be run from within the same function!

Masking Out an interrupt line

- Useful to disable interrupts **on a particular line**
 - ▶ **void disable_irq (unsigned int irq);**
Disables the irq line for all processors in the system.
Waits for all currently executing handlers to complete.
 - ▶ **void disable_irq_nosync (unsigned int irq);**
Same, except it doesn't wait for handlers to complete.
 - ▶ **void enable_irq (unsigned int irq);**
Restores interrupts on the irq line.
 - ▶ **void synchronize_irq (unsigned int irq);**
Waits for irq handlers to complete (if any).

Checking interrupt status

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

◀ irqs_disabled()

Tests whether local interrupt delivery is disabled.

◀ in_interrupt()

Tests whether code is running in interrupt context

◀ in_irq()

Tests whether code is running in an interrupt handler.

Interrupt Management Summary

- Device driver

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

Interrupt handler

- ▼ Called when an interrupt is raised.

- ▼ Acknowledge the interrupt.

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

- ▼ Otherwise, wake up processes waiting for the data.

- Tasklet

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

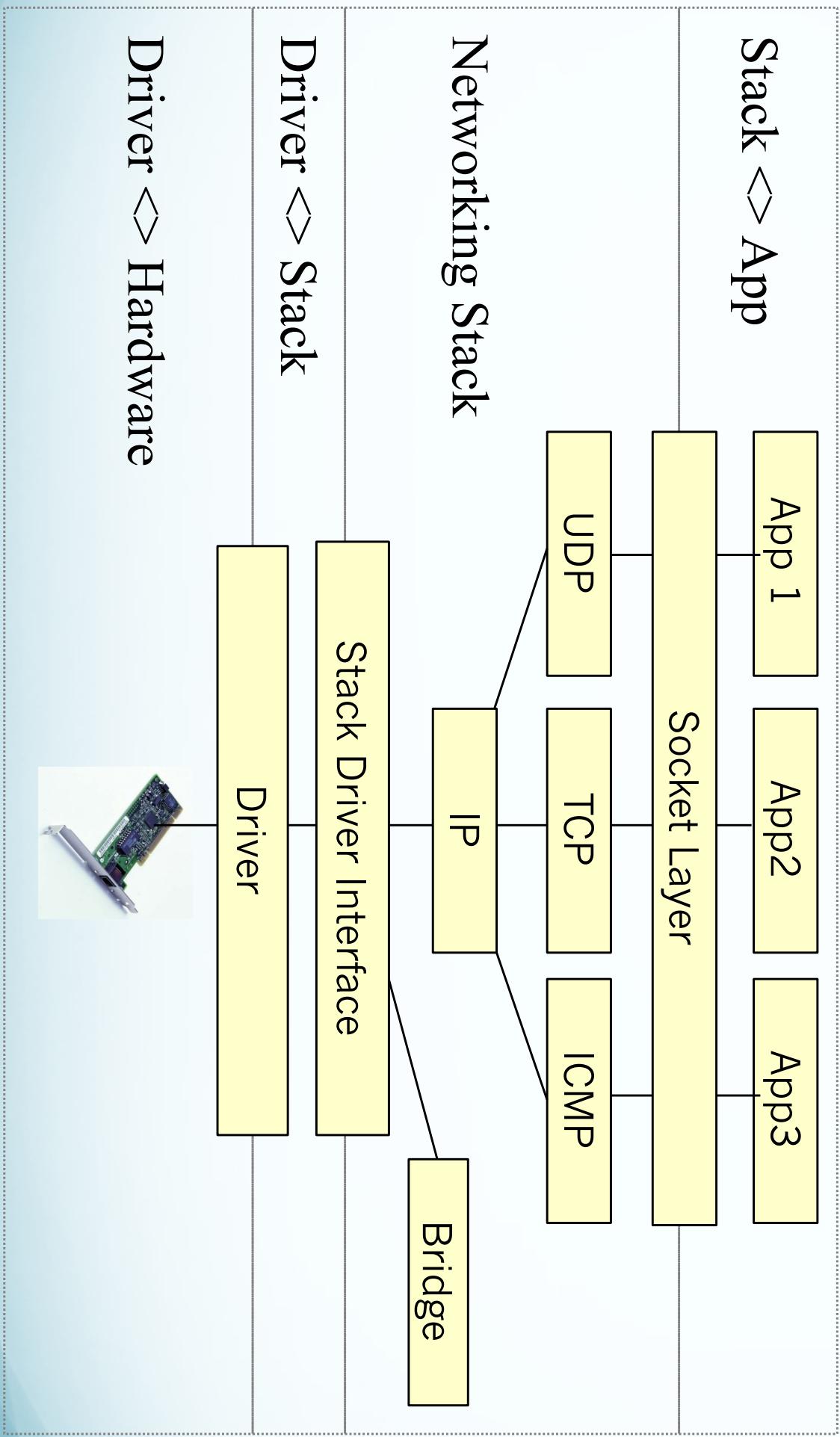
Device driver

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

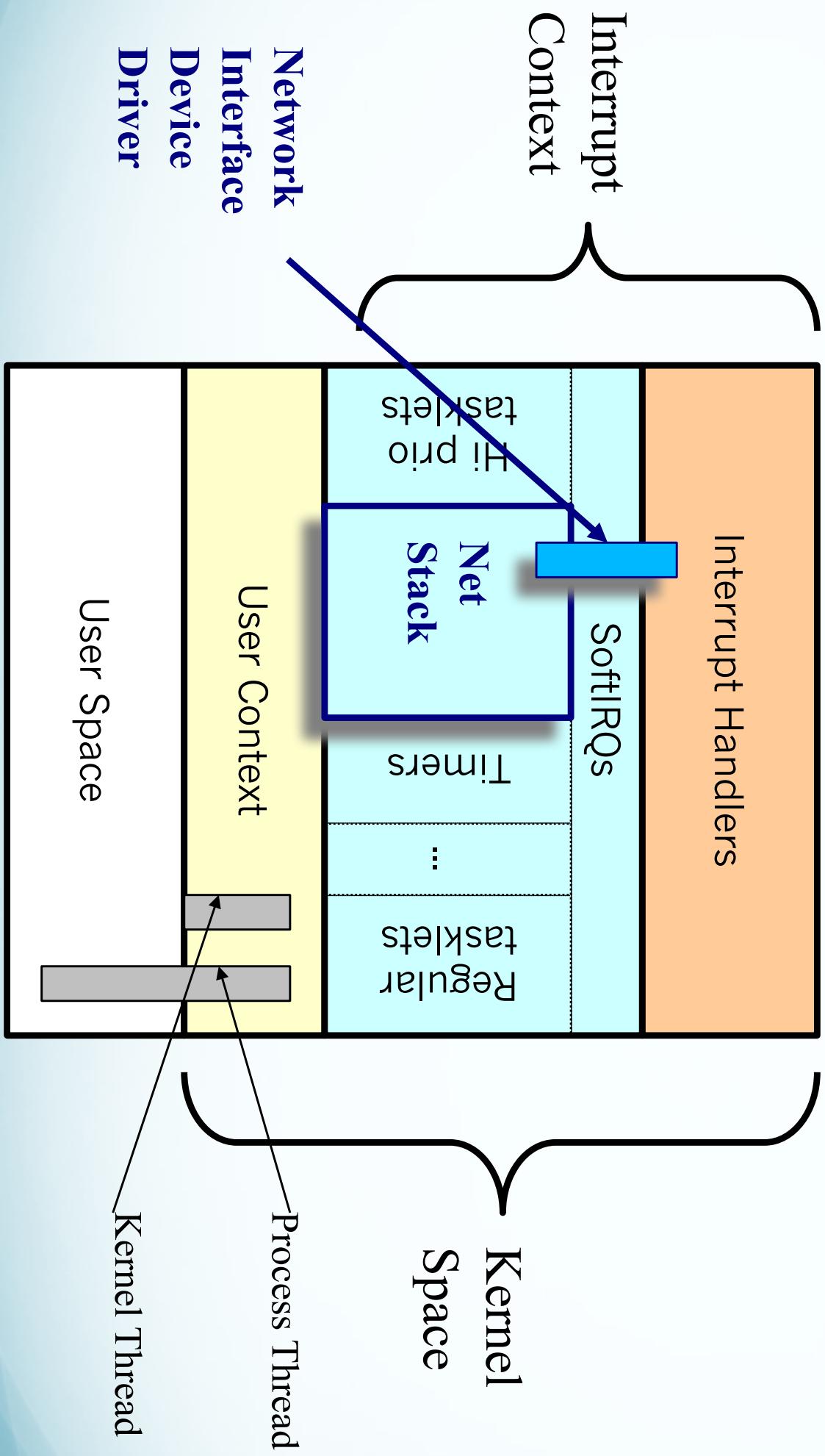
Embedded Linux Driver Development

The Network Subsystem
and Network Device Drivers

Linux networking Subsystem



Linux Contexts



BSD Sockets Interface

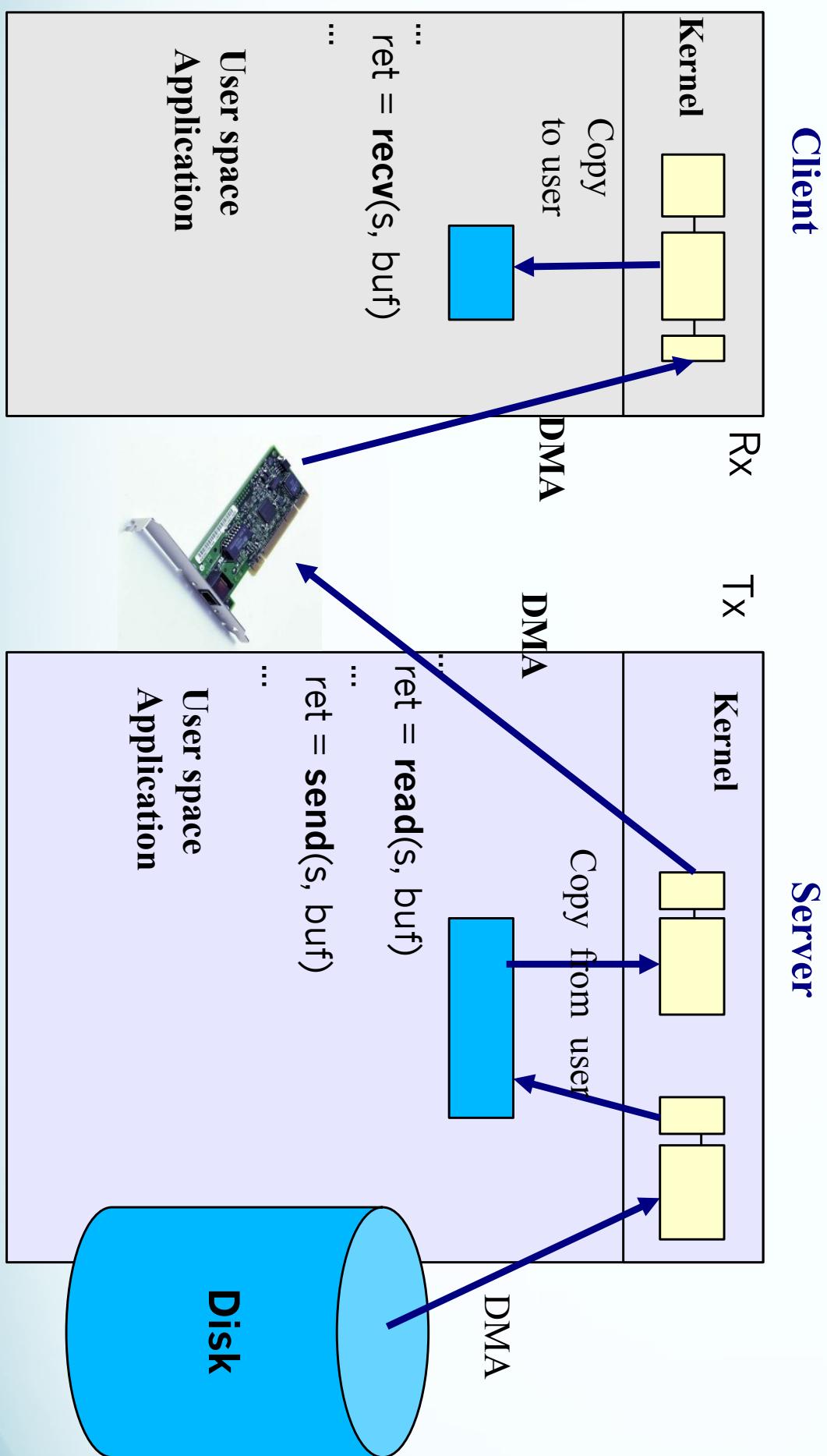
- ▼ User space network interface:
 - ▼ socket() / bind() / accept() / listen()
 - ▼ Initialization, addressing and hand shaking
 - ▼ select() / poll() / epoll()
 - ▼ Waiting for events
- ▼ send() / recv()
- ▼ Stream oriented (e.g. TCP) Rx / Tx
- ▼ sendto() / recvfrom()
- ▼ Datagram oriented (e.g. UDP) Rx / Tx

BSD Sockets Interface

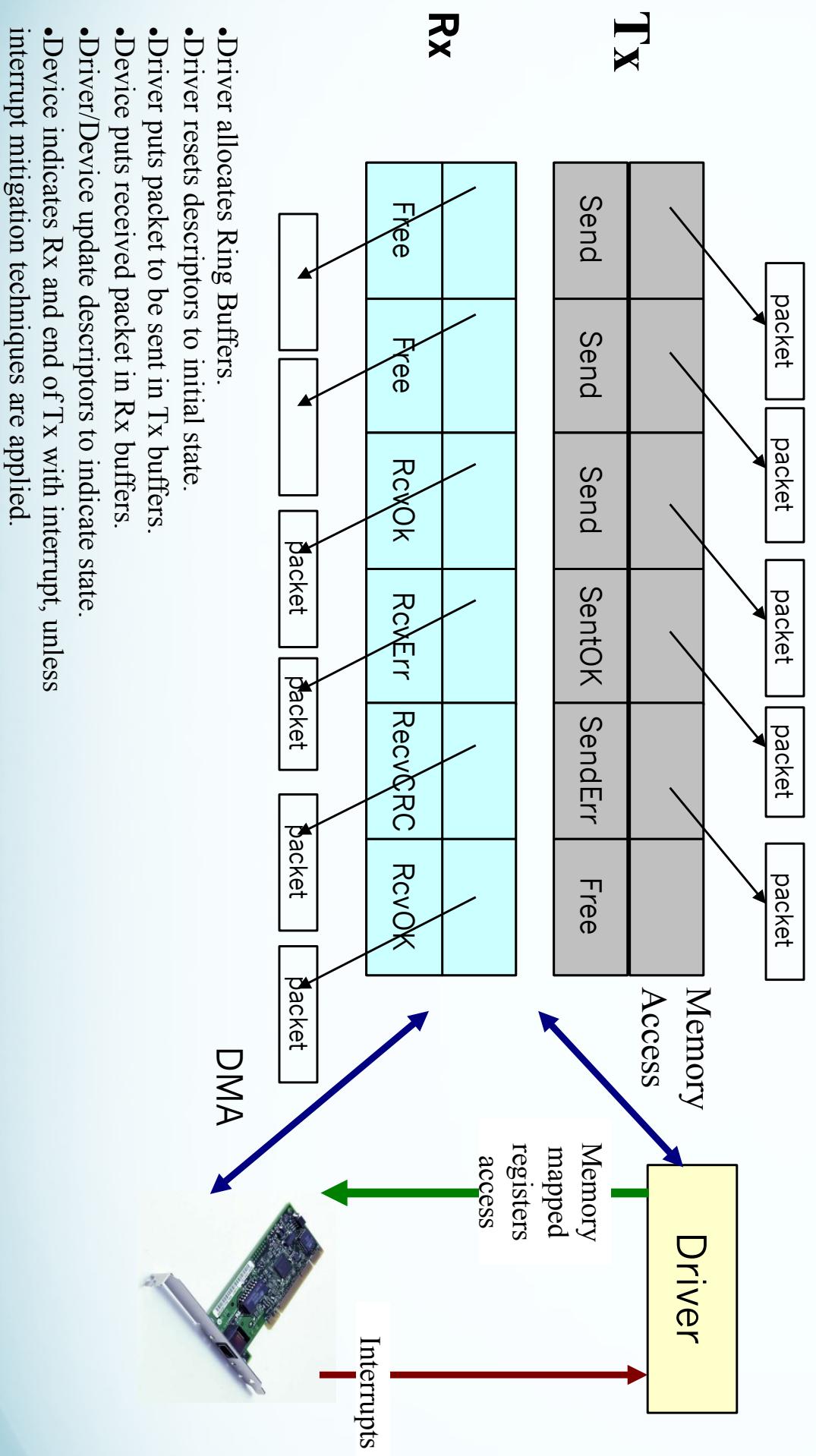
Properties

- ▶ Originally developed by UC Berkeley research at the dawn of time
- ▶ Used by 90% of network oriented programs
- ▶ Standard interface across operating systems
- ▶ Simple, well understood by programmers
- ▶ Context switch for every Rx/Tx
- ▶ Buffer copied from/to user space to/from kernel

Simple Client/Server Copies



Hardware Interface



Packet Representation

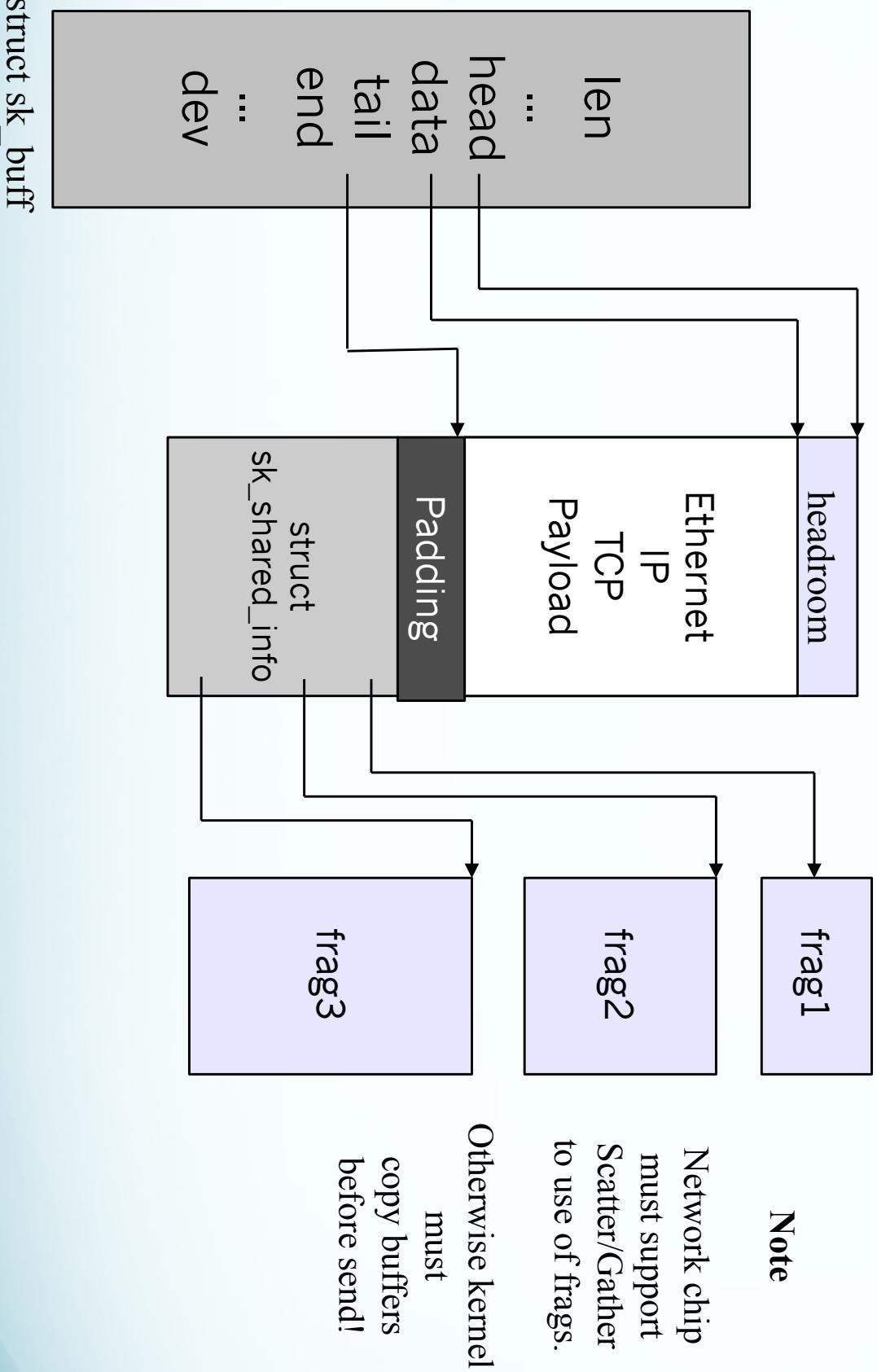
- ▼ We need to manipulate packets through the stack
- ▼ This manipulation involves efficiently:
 - ▼ Adding protocol headers/trailers down the stack.
 - ▼ Removing protocol headers/trailers up the stack.
- ▼ Packets can be chained together.
- ▼ Each protocol should have convenient access to header fields.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

Socket Buffers

- ▼ The sk_buff structure represents a single packet.
- ▼ This structure is passed through the protocol stack.
- ▼ It holds pointers to a buffers with the packet data.
- ▼ It holds many type of other information:
 - ▼ Data size.
 - ▼ Incoming device.
 - ▼ Priority.
 - ▼ Security ...

Tech The Best

Socket Buffer Diagram



Socket Buffer Operations

- ▼ **skb_put**: add data to a buffer.
- ▼ **skb_push**: add data to the start of a buffer.
- ▼ **skb_pull**: remove data from the start of a buffer.
- ▼ **skb_headroom**: returns free bytes at buffer head.
- ▼ **skb_tailroom**: returns free bytes at buffer end.
- ▼ **skb_reserve**: adjust headroom.
- ▼ **skb_trim**: remove end from a buffer.

Operation Example: `skb_put`

- `unsigned char *skb_put`
(`struct sk_buff * skb, unsigned int len`)
- ▼ Adds data to a buffer:
 - ▼ `skb`: buffer to use
 - ▼ `len`: amount of data to add
- ▼ This function extends the used data area of the buffer.
- ▼ If this would exceed the total buffer size the kernel will panic.
- ▼ A pointer to the first byte of the extra data is returned.

Socket Buffer Allocations

- ◀ **dev_alloc_skb**: allocate an skbuff for Rx
- ◀ **netdev_alloc_skb**: allocate an skbuff for Rx, on a specific device.
- ◀ Allocate a new sk_buff and assign it a usage count of one.
- ◀ The buffer has unspecified headroom built in. Users should allocate the headroom they think they need without accounting for the built in space. The built in space is used for optimizations
- ◀ NULL is returned if there is no free memory.
- ◀ Although these functions allocates memory it can be called from an interrupt.

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, NET_IP_ALIGN);
```

Network Device Allocation

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

```
struct net_device *alloc_netdev(size)
```
- ▼ *size* – size of our private data
- ▼ And deallocated with:

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

- ▼ The *net_device* struct 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.
 - ▼ *tx_timeout()* - called when a timeout occurs ...

Network Device Registration

- ▶ `register_netdev(struct net_device)`
- ▶ Used to register the device with the stack
- ▶ Usually done in a bus probe function when new device is discovered.

Packet Reception

- ▶ The driver allocates an skb and sets up a descriptor in the ring buffers for the hardware.
- ▶ The driver Rx interrupt handler calls `netif_rx(skb)`.
- ▶ `netif_rx` deposits the `sk_buff` in the per-cpu input queue, and marks the `NET_RX_SOFTIRQ` to run.
- ▶ At SoftIRQ processing time, `net_rx_action()` is called by `NET_RX_SOFTIRQ`, which calls the driver `poll()` method to feed the packet up.
- ▶ Normally `poll()` is set to `process_backlog()` by `net_dev_init()`.

Packet Transmission

- ▶ Each network device defines a method:

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

- ▶ This function is indirectly called from the NET_TX_SOFTIRQ
- ▶ Call are serialized via the lock dev->xmit_lock_owner
- ▶ The driver manages the transmit queue during **interface up and downs** or to **signal back pressure** using the following functions:

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

NAPI

- ▼ Adaptive interrupt mitigation and scaling in software
- ▼ Alternate between interrupt and polling according to need:
 - ▼ Use interrupts to detect “first packet”.
 - ▼ Move to polling to get packets.
 - ▼ Return to interrupt when no traffic.
- ▼ It is used by defining a new method:

```
int (*poll) (struct net_device *dev, int *budget);
```

- ▼ which is 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 it's poll method.
- ▼ The Poll method
 - ▼ Scans receive ring buffers, feeding packets to the stack via: **netif_receive_skb(skb)**.
 - ▼ If work finished within budget parameter, re-enables interrupts and calls **netif_rx_complete(dev)**
 - ▼ Else, stack will call poll method again.

Embedded Linux driver development

Advice and Resources

Getting Help and Contributions

Sites

- ▼ <http://www.kernel.org>
- ▼ <http://elinux.org>
- ▼ <http://free-electrons.com>
- ▼ <http://rt.wiki.kernel.org>
- ▼ <http://kerneltrap.com>

Information Sites (1)

- Linux Weekly News
<http://lwn.net/>
- ▼ The weekly digest off all Linux and free software information sources.
- ▼ In-depth technical discussions about the kernel.
- ▼ Subscribe to finance the editors (\$5 / month).
- ▼ Articles available for non-subscribers after 1 week.

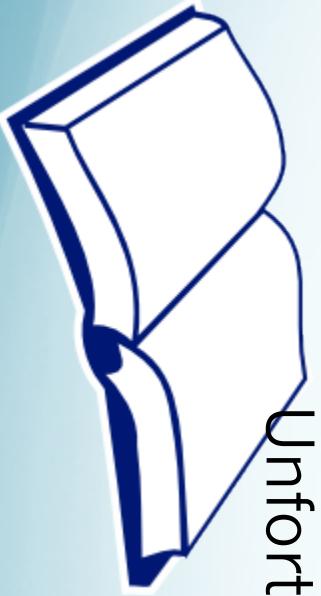


Your Linux info source

Useful Reading (1)

- Linux Device Drivers, 3rd edition, Feb 2005
 - ▶ By Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, O'Reilly
<http://www.oreilly.com/catalog/linuxdrive3/>
 - ▶ Freely available on-line!
Great companion to the printed book for easy electronic searches!
<http://lwn.net/Kernel/LDD3/> (1 PDF file per chapter)
<http://free-electrons.com/community/kernel/ldd3/> (single PDF file)
- A must-have book for Linux device driver writers!

Useful Reading (2)

- 
- ▶ Linux Kernel Development, 2nd Edition, Jan 2005
Robert Love, Novell Press
http://rllove.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/underlk/>
An extensive review of Linux kernel internals, covering Linux 2.6 at last.
Unfortunately, only covers the PC architecture.

Useful Reading (3)



- ▶ Building Embedded Linux Systems, 2nd edition, August 2008
Karim Yaghmour, Jon Masters, Gilad Ben Yossef, Philippe Gerum, O'Reilly Press
 - ▶ <http://www.oreilly.com/catalog/belinuxsys/>
- See
<http://www.linuxdevices.com/articles/AT2969812114.html>
for more embedded Linux books.

Useful On-line Resources

▼ Linux kernel mailing list FAQ

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

Complete Linux kernel FAQ.

Read this before asking a question to the mailing list.

▼ Kernel Newbies

<http://kernelnewbies.org/>

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



International Conferences

- Useful conferences featuring Linux kernel presentations
- ▼ Linux Symposium (July): <http://linuxsymposium.org/>
Lots of kernel topics.
- ▼ Fosdem: <http://fosdem.org> (Brussels, February)
For developers. Kernel presentations from well-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

- ▼ 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

Labs

Using QEMU

- ▶ To run qemu
 - # cd ~/armsystem
- ▶ To compile for ARM
 - arm-none-linux-gnueabi-gcc -g3 -o myapp ./source.c
- ▶ Root directory for the target is located in
 - ~/armsystem/outfs