Seminar

Real-Time Embedded Linux

Lecturer

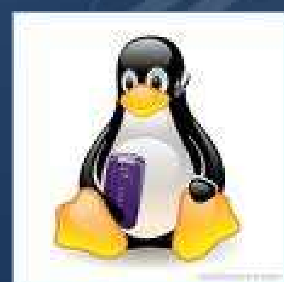
Ziv Erez

Ziv is a senior lecturer at Logtel.
Ziv Erez is an accomplished software professional with over 16 years of experience in the high-tech industry, mainly with fabless ASIC companies. He specializes in kernel-level and device drivers development for modern operating systems, including Windows, Linux and embedded systems. Ziv has managed Linux and Windows driver development teams both domestic and off-shore. His specific areas of expertise are Wi-Fi, Wireless Sensor Networks (WSN), and USB 2.0.

Allon Herman

Allon Herman has an experience of many years in the field of UNIX / Linux.
He has been working as a freelance since 1992.
Allon has worked with multiple leading Israeli and international organizations during his career like Orbotech (as Linux Mentor), Intel (Drivers development) and Go Networks (Integration of wireless systems using Embedded Linux).
He wrote multiple UNIX and Linux device drivers for controlling communication devices, medical devices and others.
He has acted as a Linux Mentor, consulting to organizations in process of migration their engineering environments to Linux.
Allon has dealt with the planning and deploying engineering servers, engineering support and configuration management, including ClearCase, Perforce and SVN.
Allon has instructed many courses in Israel and abroad in parallel to the activities mentioned herein. The course included the following fields: Shell Scripting, PERL, Make, UNIX / Linux System Programming, UNIX (various flavors) and Linux System Administration, UNIX and Linux Kernel Programming.

Linux Kernel & Device Drivers



By Allon Herman & Ziv Erez

Q4' 2021

About the Copyright

Covering Variety of Advanced Technology areas

Software

Telecom

Hardware

Currently:
- Automotive
- LBS
- Smart city

Internet of Things
& related industries

Data Science:
Big data & AI

LOGTEL

Knowledge By All Means

Courses

Nextgen Seminars

Web info

Conferences

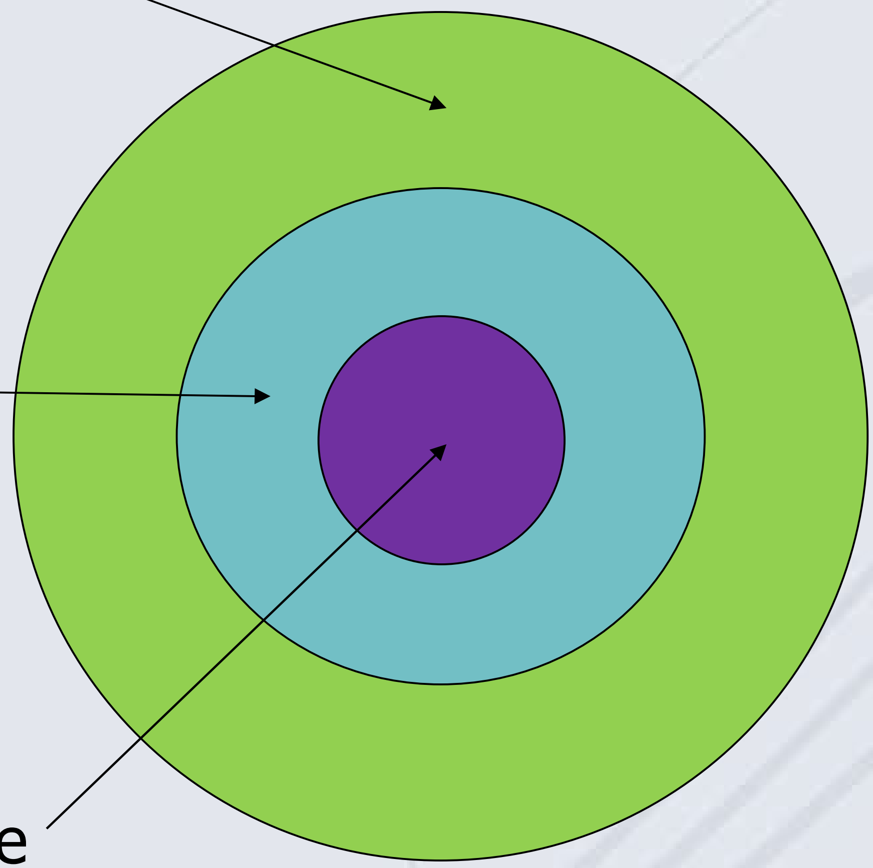
Consultation & projects

Linux Kernel & Device Drivers

Chapter 1 - Introduction

Linux System Overview

- User Land
- The Kernel
- The Hardware



Roles of The Kernel

- Manage hardware resources.
- Manage user processes.
- Grant user processes access to hardware resources.
- Protect hardware resources.
- Protect user process resources.

Supporting Multiple Architectures

- The Linux kernel is designed to support multiple architectures.
- Linux kernel CPU architecture support is done in an architecture specific section of the kernel.
- These sections provide a uniform API, so the rest of the kernel is "CPU architecture blind"

Supported Architectures

- The Linux kernel supports a wide variety of CPU’s

- The following CPU’s are among the most widely used CPU’s:
  - `X86`
  - `X86_64`
  - `Power PC`
  - `Arm`

- The CPU’s listed above are a fraction of the variety of CPU’s the Linux supports.

The Kernel Sub-Directories

Documentation

- The `Documentation` directory contains reference documentation of the kernel's API.

- The documents may be formatted in several ways, to suit the readers needs:
  - Manual Pages
  - Post Script
  - PDF
  - HTML

Documentation - Continued

- Building the documents in HTML format:

```
$ cd <kernel-source-dir>
$ make htmldocs
```

- Reading the documentation:

```
$ firefox <kernel-source-dir>/Documentation/DocBook/index.html
```

- The documentation is useful, though terse, and incomplete.

arch

- The arch directory contains the processor specific support.
- Every family of CPU's supported by Linux has a sub directory under arch. (*e.g.* `arch/x86`)
- The CPU family specific sub directory provides:
  - Architecture specific header files, that define architecture dependent macros and constants, and architecture specific inline functions.
  - The low level kernel API, that is implemented in an architecture specific manner, partially in assembly.

arch - Continued

- Example – The system calls’ interface:
  - The assembly function ENTRY defines the kernel system call interface.
  - X86_64 implementation in `arch/x86/entry/entry_64.S`
  - Arm implementation in `arch/arm/kernel/entry-common.S`
  - C code that implements the actual system call is architecture independent (*e.g.* fork in `kernel/fork.c`)

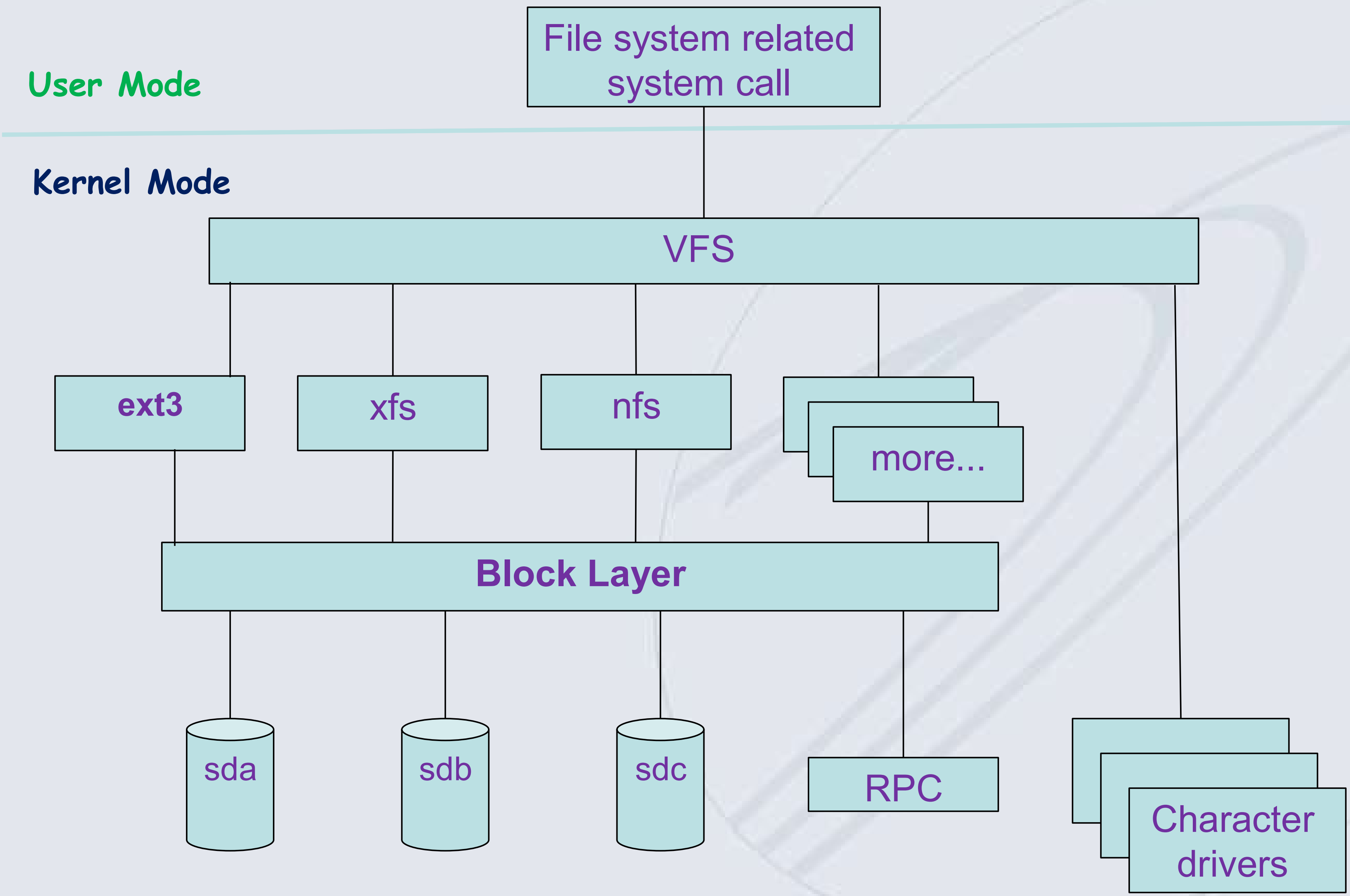
fs

- The sub directory `fs` contains the support for file systems.
- It implements the file system related system calls.
- It implements the VFS.

fs - Continued

- It's sub directories implement the various file systems:
  - General purpose - `ext4`
  - Compatibility - `NTFS`
  - Scalable - `XFS`
  - Clustered (SAN) - `OCFS2`
  - Distributed - `Lustre`
  - Flash Optimized - `cramfs`

Understanding VFS



mm

- The sub directory mm contains the high level implementation of memory management.
- Low level implementation of memory management is in `arch/<cpu-family>/mm`
- The main elements in **mm** are:
  - Kernel memory allocators
  - Swapping / paging
  - Page tables management
- It supports MMU emulation for hardware that lacks MMU (*e.g.* nios).

block

- The sub directory block implements:
  - The API used by block devices
  - I/O schedulers
  - File mapping infrastructure
- File systems are highly dependent on the block layer.

drivers

- The sub directory drivers has almost 90 sub directories, that group the Linux device drivers into “families”:
  - Vendor specific device drivers - nubus, sbus, etc.
  - Infrastructure drivers that provide a uniform API that is use by other device drivers - SCSI, USB, etc.
  - Families of drivers that provide API’s that are used by the rest of the kernel - char, block, net, etc.
  - Virtualization infrastructure - xen.
  - More...

include

- The sub directory include contains sub directories that contain architecture independent include files.
- Copies of the sub directories of include exists in **`/usr/include`**.
- These copies are used to maintain a common language between user applications and the kernel in terms of:
  - Constant definitions
  - Macro definitions
  - Structure definitions

include - Continued

- Sections of include files that are intended to be used only by the kernel are marked as following:

```
#ifdef __KERNEL__
. . . .
#endif /* __KERNEL__ */
```

net

- he sub directory net implements the network related functionality.
- The socket subsystem infrastructure is implemented in net/core.
- Other subdirectories implement network protocols (at all levels):
  - Link layer - ethernet, mac80211, ax25
  - Network and transport layers - ipv4, decent, appletalk, etc
  - Presentation layer - sunrpc
  - Application <-> kernel connectivity - netlink
  - Tunneling - l2tp

crypto

- The sub directory crypto implements multiple encryption algorithms.

- The encryption algorithms are used in:
  - Secured network connections - IPsec
  - Transparent encryption of storage devices - dm-crypt
  - Module signing

init

- The sub directory init implements the kernel initialization:
  - Load and mount of the ram disk
  - mount of the root file system
  - hand crafting of `init`† (PID 1)

Recent distro's are using `systemd` instead of "good old init"

lib

- The sub directory lib contains utility functions that are equivalent to functions that are typically found in libc.

- Example:
  - String manipulations functions - **`strcpy`**, **`strcmp`**, etc.
  - Text formatting - **`sprintf`**.
  - more...

Linux Kernel & Device Drivers

Chapter 2 - The Kernel Architecture

Linux Kernel Versions

- 0.11 - December 1991
- 1.0 - March 1994
- 1.2 - March 1995
- 2.0 - June 1996
- 2.2 - January 1999
- 2.4 - January 2001
- 2.6 - December 2003
- 3.0 - July 2011

- Current – 5.14.9 (30-Sep-2021)

The Kernel Release Model

- Up to 2.6:
  - Odd versions were development versions.
  - Even versions were production versions.
  - The release of a new production version was the conclusion accord of a development version.

- 2.6 upwards:
  - New versions are released once every 2-3 months.
  - New releases include both bug fixes and new features.
  - Long term releases continue to be maintained, although new versions have already been released. Long term releases are typically used with long term supported distributions:
    - 2.6.32.X - SLES 11.X, RHEL 6.X, ...

The Kernel Release Model - Continued

- Kernel versions moved from 2.6.X to 3.X to designate 20 years since the first kernel release.

- The recent move from 3.X to 4.X, does not show any significant change:
  - It is the outcome of a poll among kernel developers.
  - It signifies 500,000 GIT commits (the change from 2.6 to 3.0 was after 250,000 GIT commits).

Kernel General Properties

- Main kernel properties:
  - SMP Support.
  - Soft real time (pre 2.6 versions did not properly support real time).
  - Hard real time available as a kernel patch (not available for all releases).
  - POSIX (IEEE 1003) compliant.
  - A modular kernel.
  - A wide range of supported platforms.
  - Open source.

System Calls

- System calls grant user mode programs access to protected system resources.

- System call are the means to request services from the kernel.

- System calls may be grouped, per the areas that they deal with:
  - File System – `open, close, read, write, ...`
  - Memory Allocation – `brk, sbrk`
  - Inter Process Communication – `shmget, semop, msgctl, ...`
  - Process – `fork, execv, ...`
  - Network – `socket, bind, connect, send, ...`

Executing A System Call

- User code calls a run time library function.
- The runtime library function sets a CPU register to the ID of the system call.
- The runtime library function issues a software interrupt.†
- Interrupt handler users the register set earlier as an index in the system calls lookup table.
- The interrupt handler copies system calls parameters from user address space to kernel address space.†

The actual implementation is architecture dependent

Executing A System Call (Cont'd)

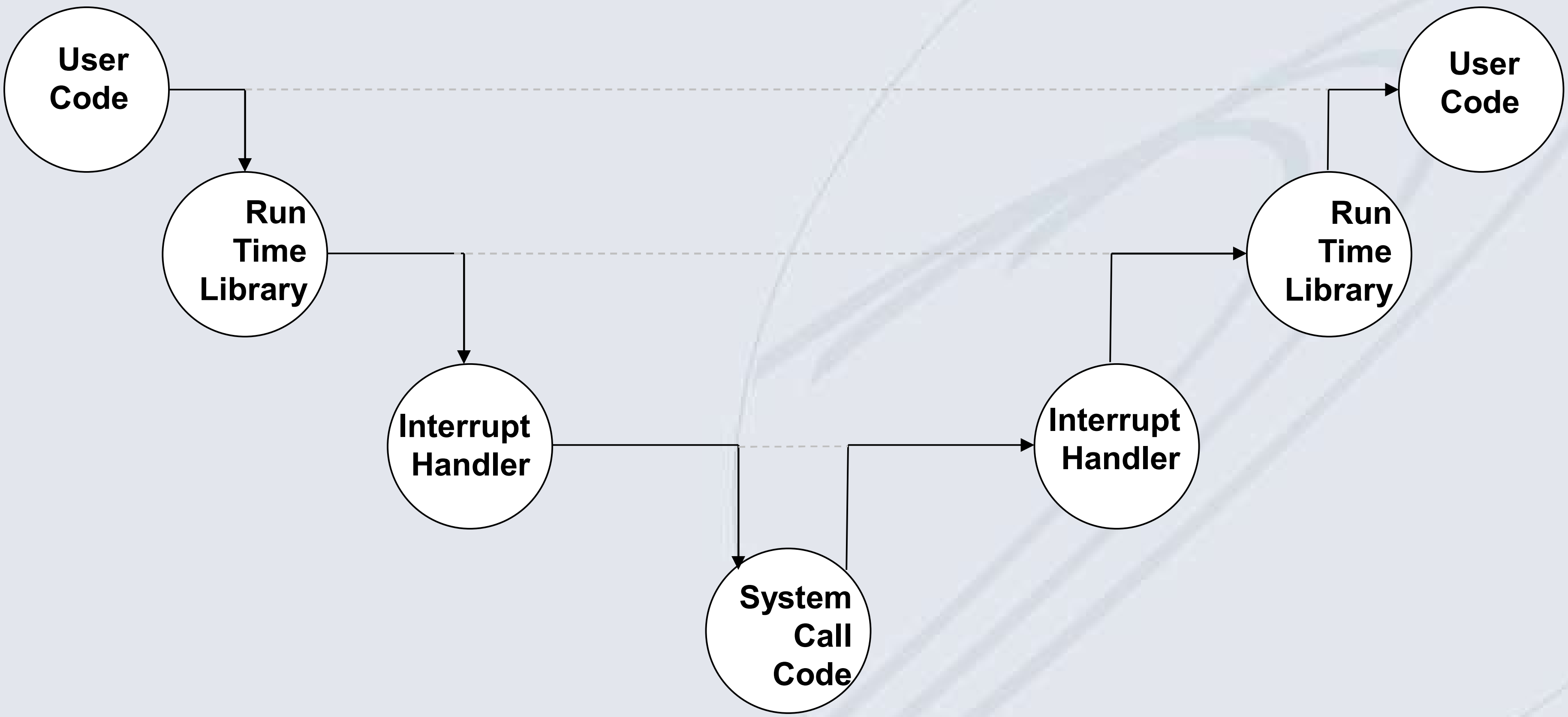
- The interrupt handler calls the kernel function that was picked from the lookup table.†
- The kernel system call code executes and fulfils the request.
- The software interrupt handler copies the execution results (return value and error code if relevant), to CPU registers.†
- CPU returns to user mode, and control is returned to the runtime library function that issued the software interrupt.

† The actual implementation is architecture dependent

Executing A System Call (Cont'd)

- The runtime library copies error code from the CPU register to `errno`, and returns the return code that was stored by the kernel code in another CPU register.
- The user program continues to execute.

System Call Execution



System Calls - Exercise

- Write two short C programs, both which execute the command `ls`:
  - Implement the first program using the `system` C library function.
  - Implement the second program using the `fork` and `exec` system calls.
- Can you tell which of the two runs more efficiently?

- Tip:
  - Choose the `exec` system call that you find most convenient.

The Task Scheduler

- The old implementation of the task scheduler wasn’t designed to take advantage of multi core and multiprocessor architectures. Hence, it suffered from several limitations:
  - It had an O(*n*) complexity (*n* being the number of tasks in the run queue), because its algorithm required a scan of the entire run queue, each time a task was scheduled.
  - SMP performance was poor, and NUMA even worse, because it used a single run queue, and processes were prone to being migrated among CPU’s.
  - The scheduler used a single run queue lock. Thus, if the scheduler was scheduling a task to run on one CPU, tasks could no be scheduled on other CPU’s at the same time.

The 2.6 Scheduler

- One of the most notable changes in kernel 2.6 is the scheduler.

- The old pre-2.6 scheduler has been completely rewritten.

- The new scheduler has an O(*1*) complexity, meaning that scheduling time is fixed, regardless of how many processes are currently executing.

The 2.6 Scheduler - Continued

- It has a per CPU run queue, which allows a per CPU lock, rather than the old big-lock.
- The per CPU run queues guarantee the CPU affinity of tasks and prevent them from migrating among CPU’s.
- The run queues are implemented as FIFO queues.
- The time quantum, and the priority of tasks is calculated when the task is preempted.

The 2.6 Scheduler (continued)

- When that task is preempted it is placed in a per CPU expired queue.
- The expired queue and the run queue are swapped, when the run queue becomes empty.
- The run queue is divided into 140 sections (per the 140 supported priority levels).
- Finding a task to execute becomes a function of the number of priorities rather than tasks. Thus, taking it to O(*1*).

3.8 - Further Advancement

- The 2.6 scheduler is good at keeping physical CPU affinity, but allows process migration among cores.

- Migration of processes among cores introduces a significant execution time overhead on systems with multiple cores per CPU, especially NUMA based system.

- Linux 3.8 introduced enhanced NUMA support, that is good at core affinity.

Avoiding Starvation of Tasks

- The 2.6 scheduler exercises the following measures to avoid starvation of tasks:
  - An interactiveness metric is calculated for each process, to determine if it is I/O bound, or CPU bound.
  - CPU bound processes are penalized in their priority.
  - I/O bound processes receive a priority bonus.

Avoiding Starvation of Tasks

- The 2.6 scheduler performs load balancing between CPU's:
  - A kernel task checks that all CPU's are load balanced, once every 200*ms*.
  - Tasks are migrated among CPU's to achieve load balance, if the CPU's are found to be not load balanced.
  - The down side of load balancing is that the new CPU's cache is "cold".
  - Special consideration should be taken on NUMA systems.

Process Transition States

- Created –
  - Being created is an intermediate state. It lasts until the process becomes run-able.
  - During this state, the system allocates the resources required for managing it.
  - The parent process’ environment is copied to the process being created.

Process Transition States - (Cont'd)

- Ready to Run –
  - A process is ready to run once it's creation is complete, and there is no resource on which it is blocked.
  - The task scheduler elects processes to run from the ready to run list of processes based on their scheduling priority and scheduling policy.

Process Transition States (Cont'd)

- Kernel Running –
  - Processes always become kernel running when they are elected to run.
  - When a process is kernel running the CPU is in supervisor mode.
  - Processes become kernel running whenever they issue a system call, until its completion.
  - Processes become kernel running whenever the processor/core on which the are executing is interrupted, for the entire duration of its processing.

Process Transition States (Cont'd)

- When a process is kernel running it executes kernel code and uses the kernels page table.
- Processes that are kernel running use a process specific kernel stack.†
- The time that a process spends in kernel running state, is on account of its time quantum, even when it is for the purpose of servicing an interrupt.

†Recent kernels implement a per-core interrupt handling stack.

Process Transition States (Cont'd)

- User Running –
  - When a process is user running the CPU is in user mode.
  - When a process is user running it executes the instructions of the executable image associated with the process.
  - User running processes have no direct access to resources owned by other processes.
  - User running processes have no direct access to kernel resources.

Process Transition States (Cont'd)

- Preempted –
  - Processes are preempted when their time slice expires.
  - If there is no process more eligible to run than the preempted process, the preempted process will be granted an additional time slice.
  - If there is a process more eligible to run that the preempted process, it will run and the preempted process will be ready to run.

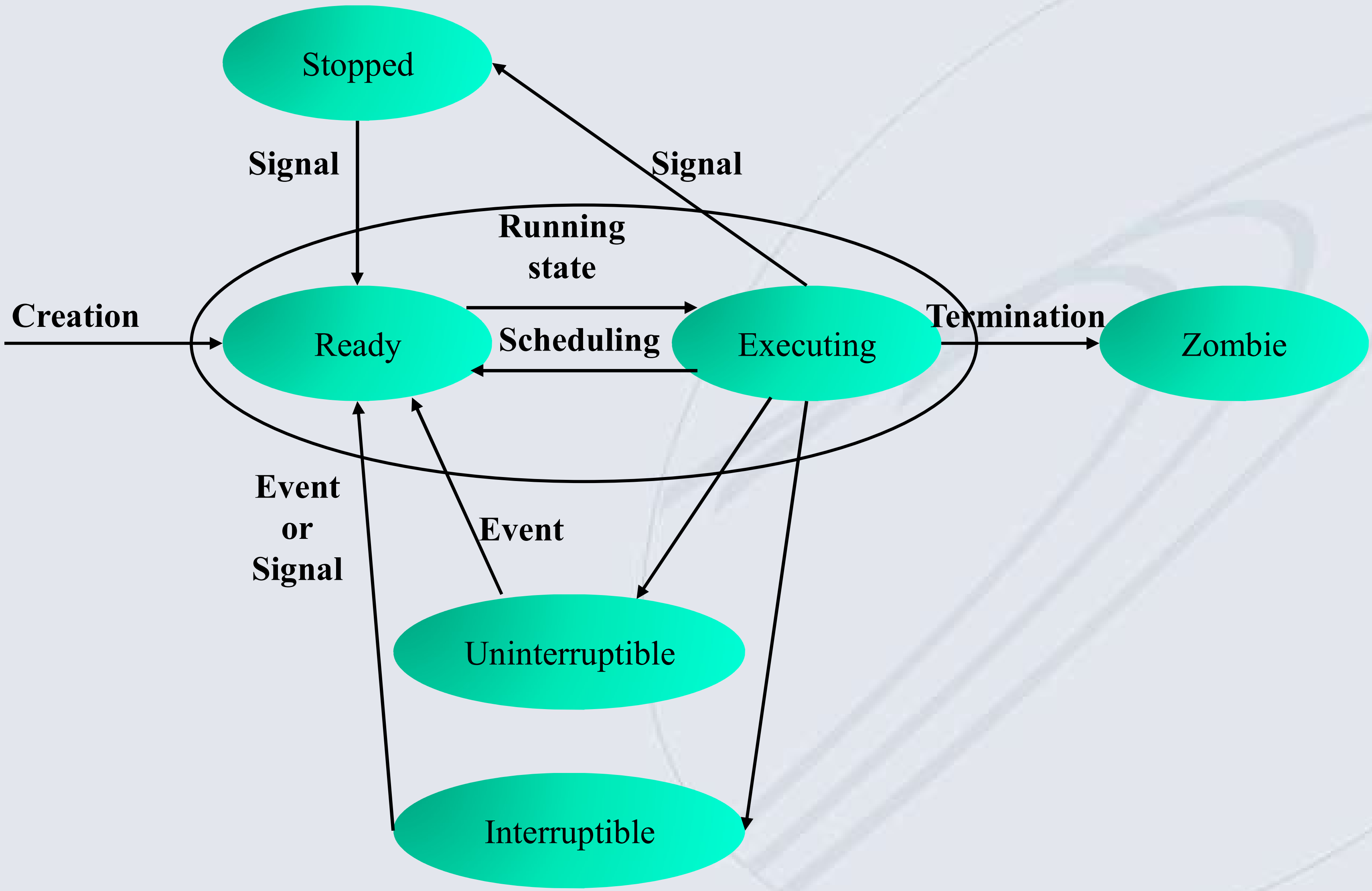
Process Transition States (Cont'd)

- Asleep (Awaiting an event) –
  - Processes go to sleep while they are awaiting an event (e.g. keyboard input), or waiting for a resource to become available (e.g. enough memory to satisfy an allocation request).
  - At first, processes are asleep in the memory. Gradually, if a process has slept for an extended period, and there is not enough memory to meet the requirements of a running process, it will be swapped out.
  - When an event / resource on which a process has slept incurs / becomes available, the process is waken up.
  - When a process that has been swapped out is waken up, it will be swapped in to allow its execution.

Process Transition States (Cont'd)

- Zombie –
  - When a process exits, it relinquishes all of its resources, other that its task_struct. At this time, the process is a zombie.
  - Zombies disappear when their exit status code is collected by their parent process. When a zombie disappears, its task_struct and PID are released.

Process Life Cycle



53

Processes - Exercise

1. Take a snapshot of the processes that are currently executing.
2. Can you tell how many processes exist in each run state?
3. Hint, ps’s formatting options can be useful, in conjunction with the commands sort, and, wc.

Block Devices

- Block devices are devices that perform I/O in blocks of a fixed size.

- Typically, physical blocks may be randomly accessed.

- Block devices may be characterized as rotating media, in many cases (disk drives, DVD’s, etc.).

Block Devices - Continued

- Access time to a physical block is influenced by the mechanical characteristics of the device, and by the physical block that was previously accessed.

- Classic Unix systems use elevator sort, to order the I/O requests, thus minimizing the mechanical overhead, and providing the best average performance.

Elevator Sort Not As Good As It Seems

- While elevator sort grants the best average performance, it might have some severe drawbacks:
  - Physical blocks near the perimeters, or the rotation axis of a drive are likely find their way to the bottom of the I/O queue.
  - What is the benefit of elevator sort when using a logical device (software, or hardware implemented)?
  - What if we need a guaranteed I/O rate for a real time application?

I/O Scheduling

- I/O schedulers have been introduced to Linux in kernel 2.6

- I/O schedulers provide the means for optimizing I/O performance on specific block devices.

- The I/O scheduler chosen should be based on the physical characteristics of the block device, and on its intended use.

CFQ Scheduling

- CFQ – Completely Fair Queuing.
- A per process I/O queue is maintained.
- I/O bandwidth is distributed equally between all I/O requests.
- Based on elevator sort.
- The default in most distributions.
- Most suitable for general purpose systems.

Deadline Scheduling

- Uses a deadline algorithm to minimize latency for any I/O request.
- Round robin is used to avoid starvation of processes, and to guarantee fairness between processes.
- The behavior is near real time.

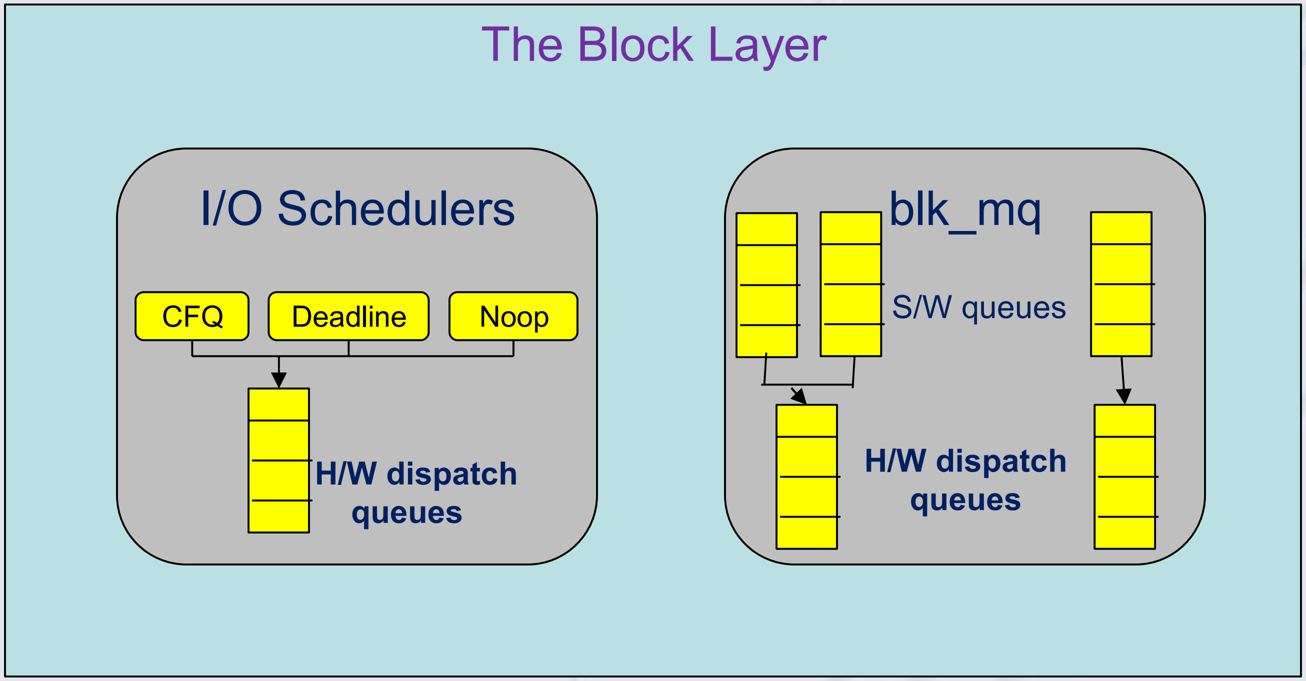
Anticipatory Scheduling

- Introduces a delay prior to serving I/O requests.
- The delay allows ordering of the requests in a seek optimized manner, and aggregating of requests that relate to adjacent blocks.
- Might cause a higher I/O latency.
- Intended for use on systems with small, or slow disks.

NOOP Scheduling

- Simply accumulates the requests in a FIFO queue.
- Intended for use in the following cases:
  - The underlying hardware has a smart controller capable of doing hardware optimization.
  - When the underlying hardware is a ram disk, or a flash drive, typically on embedded systems.

Further Improvement of The Block Layer



The Block Layer - blk_mq

- Disk I/O shaping by schedulers, as previously discussed, is done based on the physical characteristics of rotational devices.

- Performance characteristics of rotational devices may be summarized by the following statement:

  - Random accesses that require head movement are slow. Sequential accesses that only require disk platter rotation are fast.

- The introduction of SSD's exhibits a little latency difference between sequential and random accesses.

The Block Layer - blk_mq

- The difference of the hardware characteristics between SSD's and HDD's makes latencies of SSD's 4 orders of magnitude lower than these of HDD's.

- While the typical latency of an SSD is a few tens of usecs, the typical latency of an HDD is a few tens of msecs.

- The outcome is that an SSD is capable of performing far more IOPS than an HDD.

- The increase in number of cores and CPU sockets, along with the higher SSD performance turns the block layer into a bottleneck.

The Block Layer - blk_mq

- Block layer performance issues:
  - A per-device request queue is shared by all cores. Request queue consistency is achieved by locks. The latency of waiting for request queue locks increases, as the number of IOPS increases.
  - I/O request completion is typically signaled by an interrupt. The increase in IOPS translates into an increase in the interrupt rate.
  - NUMA architecture poses even a higher latency due to the need to access memory of remote nodes (both request queues, and locks).
- The performance issues above turn the block layer into a non-scalable subsystem.

The Block Layer - blk_mq

- The multi-queue block layer has been designed to deal with performance issues, while keeping it as device independent as possible.

- blk-mq requirements:

  - Fairness - Multiple processes may be accessing a single storage device. Yet, none of these processes should be starved.

  - Accounting - Monitoring and debugging of storage devices should be easy to perform by system administrators.

  - Staging - A staging area is needed within the block layer, to improve performance and guarantee fairness of service.

The Block Layer - blk_mq

- blk_mq architecture:
  - Software staging queues - Each CPU socket, or even core, has its own software staging queue. Remote memory accesses are eliminated. Cache misses are limited to misses on L1/L2. Locks are only awaited by processes from within the same CPU socket (less processes awaiting locks combined with faster lock access).
  - Hardware dispatch queues - an intermediate layer that queues IO's scheduled for dispatch to the device. Multiple hardware dispatch queues may exists per device, depending on the concurrency of the device.

The Block Layer - blk_mq

- The implementation of blk_mq is regarded complete only from Linux 4.17 onwards.

- The first I/O scheduler associated with blk_mq became an integral part of mainline kernels on release 4.12 (summer 2017).

- BFQ - Budget Fair Queueing:

  - Based on CFQ, allocates each process a 'budget' of sectors.
  - Throughput may be distributed across multiple queues.
  - Provides high throughput along with low latency.

I/O Scheduling - Exercise

1. Find out what are your disk devices (fdisk –l can be useful)
2. For each device examine /sys/block/<*device*>/queue/scheduler and determine which policies it supports, and how is it set now.
3. Important, meta devices are based on the policy of the underlying physical devices.

Linux Kernel Preemption

- Any user process that is executing in user mode is preemptive.
- Whenever the time quantum of a user process expires, it may be preempted.
- However till kernel 2.6, once a user process entered kernel mode, by issuing a system call, it could not be preempted. Thus completely locking the system, unless voluntarily giving up CPU.

Linux Kernel Preemption

- Kernel 2.6 removed the “big kernel lock”, in favor of resource specific locks.

- The scheduler introduced in kernel 2.6 allows preemption of kernel processes and user processes while executing in kernel mode.

NPTL

- NPTL – Native POSIX Threads Library.
- NPTL has been the model for implementing threads in Linux since the introduction of kernel 2.6.
- The implementation of NPTL:
  - 1-1 model – Each user mode thread corresponds to a kernel thread.
  - Resource Sharing – Threads are creating using `clone` rather than `fork`, thus reducing resource usage overhead and creation overhead.

NPTL - Continued

- `Futex` – Fast user space locking. This is a Linux specific system call that provides the infrastructure for locks and mutexes in NPTL.
- All threads of a process share the same PID.
- Signals are normally posted to the process as a whole. Thread specific API allows posting a signal to a specific thread.

BSD vs. Linux

- Licensing:
  - Linux is licensed under GPL.
  - BSD uses the traditional BSD license that implies, that any one who creates an application based on BSD may close its code (as Apple did in OS X).
- Origins:
  - Linux is a system that was essentially built from scratch to meet Unix standards.
  - BSD is based on the legacy systems that were developed by CSRG in Berkeley till the mid 1990’s.
- Supported platforms:
  - Linux supports more CPU families than BSD does.
  - Examples: alpha and parisc are supported by Linux, but not by BSD.

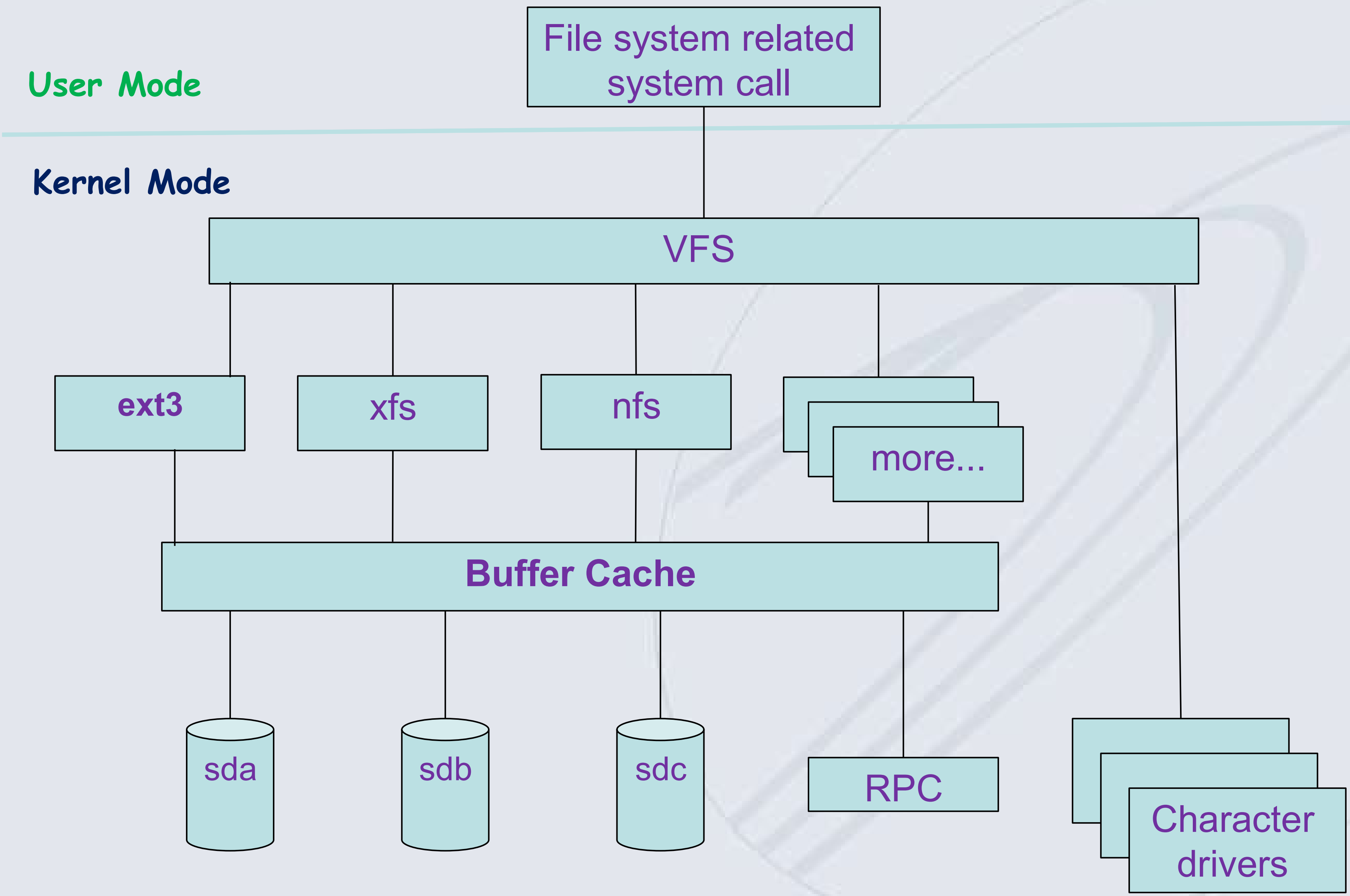
BSD vs. Linux

- Supported hardware:
  - Linux supports more hardware devices than does BSD.
  - Example: amso1100 infiniband adapter, not supported by BSD (per the list in their web site).
- New interfaces:
  - Linux has introduced API's beyond the standard UNIX API's.
  - BSD has not necessarily incorporated these API's.
  - Example: BSD has ALSA compatibility utilities, but has not incorporated it into the kernel.
- File systems:
  - Linux supports a richer set of file systems than BSD does.
  - Example: clustered file systems are not supported on BSD (GFS2, OCFS2).

Linux Kernel & Device Drivers

Chapter 3 - The Kernel Perspective

The I/O Subsystem



Operating On Files

- The user perspective of the file system is rather simple:
  - A uniform name space regardless of the actual device, or file system implementation.
  - The file system has a hierarchic structure that provides the means to address any file in an absolute, or a relative path.

```
$ pwd
/home/johndoe
absolute path .....          $ ls /home/janedoe
relative path .....          $ ls ../janedoe
```

  - A file may be simultaneously accessed by multiple processes.

Operating On Files - Continued

- The implementation of this “simple” user perspective is not as simple as it seems. It raises many questions:
  - What is the significance of the file descriptor (a non negative integer)? How does it correspond to a file?
  - How does the system call code identify the file system module that should handle a file operation?
  - What happens when an open file is being removed?
- The following slides address the questions above and many others.

The File Structure

- Every open instance of a file has a file structure.
- Each task holds a list of pointers to file structures, each file structure corresponds to a file currently open by the task.
- The file descriptor of an open file, is the index of it's file structure in the task's `files_struct` (list of pointers to file structures).
- The following file descriptors have special significance:
  - 0 - stdin
  - 1 - stdout
  - 2 - stderr

Multiple Opens of The Same File

- Files may be opened simultaneously by multiple tasks.

- The file structure maintains the common state of the file for the process that has opened it, and child processes that inherited its descriptor:
  - A reference count.
  - The current position in the file.
  - A lock† for preventing simultaneous operations on the file.

- Each open operation increments the inode reference count.

- Each close operation decrements the inode reference count.

is is not a general purpose lock. There are certain limitations on its use.

File System Deduction

- When a file is opened, The type of it's containing file system is deduced.
- The `file_operations`† field of the file structure is set accordingly.
- `file_operations` is a structure that holds pointers to back-end functions of the file system's system calls.
- The VFS implementation of file system related system calls, call the back-end functions to do the actual file system work.

† The details of `file_operations` are discussed later.

Removing Open Files

- Removing an open file is like pulling the carpet under the feet of any task that is accessing it...
- Linux avoids this situation, that implies run time errors, by taking the following approach:
  - The file's directory entry is removed immediately.
  - Yet, the actual file removal is deferred till the number of tasks holding it open is down to 0.
  - When the reference count of the file is down to 0, it will be removed.

What Makes The Removal Scenario Possible?

- If a file doesn't have a directory entry, how can it still be accessed?
- The Linux directory entry links a name to a file. Yet, it does not describe it.
- Linux uses a structure named inode to access and manage files.
- In fact the inode records all the information about a file except its name.

The Inode

- The kernel has two perspectives of an inode:
  - An in-core inode.
  - A physical (on-disk) inode.
- The in-core inode exists only during runtime of tasks that access it corresponding file.
- The information within the in-core inode is used by the VFS, and is file system independent.

The Inode

- It stores information such as file mode, time stamps, size, etc.

- The on-disk inode holds information regarding the file's layout on the disk, besides the information above.

- The on-disk inode is stored on the disk and resident as long as its corresponding file exists.

File System Structures and Drivers

- The in-core inode, provides fields that are significant in conjunction with device files:
  - `i_mode` - The file's type and access permissions.
  - `i_rdev` - The device number, of the corresponding device.
  - `i_dev` - The device number, of the device on which the file is stored.

- The device number is used by VFS to assign the `file_operations` structure of the corresponding device, to the file structure.

Exercise

1. Write a C program that uses the system call `write` to print "hello world" on the standard output (use of *printf* is not allowed here).
2. Alter the previous program, to create a file named `hello.txt` and save the output within it.
3. `/proc/<pid>/fdinfo/<fd>`, were `<pid>` is the process id of a currently executing task, and `<fd>` is the file descriptor of one of it's open files, exposes the `f_mode` field of it's file structure.
   Use the information from:
   `/usr/include/asm-generic/fcntl.h` to describe the flags with which descriptors 1, and 3 of process 1 were opened.

Introduction to sysfs

- The sysfs pseudo file system was introduced with the 2.6 kernel series.

- It addresses three main purposes:

  - Provide access to device information.

  - Allow dynamic configuration of components of the I/O subsystem.

  - Support dynamic creation and deletion of device files.

The Gory Details

- The sysfs pseudo file system is mounted on **/sys**.

- The top level sub directories of **/sys** represent various perspectives of the system's hardware:

```
# ls /sys
block   class   devices   fs          kernel   power
        bus     dev       firmware    hypervisor   module
```

- The next slides will provide a brief discussion of each perspective.

/sys/block

- The `/sys/block` directory represents the block devices that are currently active in the system.

- Each sub directory of `/sys/block` provides the information and configuration interfaces of a block device (either physical, or virtual).

- Example:

```
# ls /sys/block/sda
alignment_offset  device             inflight  removable  size       trace
bdi               discard_alignment  power     ro         slaves     uevent
capability        ext_range          queue     sda1       stat
dev               holders            range     sda2       subsystem
# cat /sys/block/sda/removable
0
# cat /sys/block/sr0/removable
1
```

/sys/bus

- The `/sys/bus` directory provides the hardware tree perspective.

- Each sub directory of `/sys/bus`, provides the information about a hardware bus and the devices that are attached to it:

```
# ls /sys/bus
acpi   event_source   hid   mdio_bus   pci           pcmcia     pnp    sdio    ssb
bcma   firewire       i2c   mmc        pci_express   platform   scsi   serio   usb
```

- Example, the PCI bus:

```
# ls /sys/bus/pci
devices    drivers_autoprobe    rescan                slots
drivers    drivers_probe        resource_alignment    uevent
```

/sys/class

- The sub directory `/sys/class` provides a device class perspective.

- The devices are classified per their functionality:
  - Memory
  - Net
  - Sound
  - SCSI disks (SAS, and SATA are also considered as SCSI)

- Example, network devices:

```
# ls /sys/class/net
eth0  lo  pan0  wlan0
```

/sys/dev

- The sub directory `/sys/dev` provides the major:minor (device number) perspective.

- It has sub directories block and char that represent the generic device types supported by Linux.

- Each device that is currently present has a representation in one of the sub directories mentioned above.

- A different view of `sda`:

```
# ls /sys/dev/block/8:0
alignment_offset  device             inflight  removable  size        trace
bdi               discard_alignment  power     ro         slaves      uevent
capability        ext_range          queue     sda1       stat
dev               holders            range     sda2       subsystem
```

/sys/fs

- The sub directory `/sys/fs` provides a file system perspective.
- The sub directories of `/sys/fs` are file systems.
- Each sub directory of a file system represents a device that is currently mounted using the respective file system module.
- Example, mounted devices that are using `ext4`:

```
# ls /sys/fs/ext4
dm-0  dm-2  features  sda1
```

/sys/module

- The sub directory `/sys/module` provides the modules perspective.
- It has a sub directory for each module, that contains it's parameters, and configuration interface.
- Example, serial port parameters:

```
# ls module/8250/parameters
nr_uarts   probe_rsa   share_irqs   skip_txen_test
```

Exercise

1. identify the name of the disk device that your system is using (`fdisk -l` can be helpful).
2. Use `/sys` to determine if it is removable or not.
3. Use `/sys` to report the capacity of the device.

The task_struct

- Each process has a `task_struct`, that describes its credentials, status and is used as a hub to access all the information related to it.

- The fields of the `task_struct` describe the following properties of a process:
  - The state of the process.
  - Signals received that are pending handling.

The task_struct - Continued

- The process' execution domain *("personality", ABI).*
- The CPU on which the processes is executing, or was recently executed.
- The process' scheduling priority (static and dynamic).
- The process' scheduling policy.
- The set of CPU's on which the process may execute.

The task_struct - Continued

- The duration of the process' time quantum.
- Pointer to the page table of the process.
- The process' binary format (*ELF, COFF, etc.*).
- The signal on which the process exited, and its exit status code (after it has exited).
- The process's *PID* and *PGRP*.
- Pointer to the process' parent and real parent.
- Pointers to the list of the process' siblings and children.

The task_struct - Continued

- The process’ real time priority.
- The process’ execution time statistics.
- The process’ real/effective/saved *UID* and *GID*.
- The process’ groups affiliation.
- The process’ resource limits.
- The *TTY* with which the process is associated.

The task_struct - Continued

- Pointer to the `files_struct` that describes the files currently open by the process.

- The process' signal reaction table and signal mask.

The Wait Queue

- The kernel maintains numerous wait queues, on which blocked processes are queued.

- The wait queues hold the “altruistic” process that have voluntarily given up their CPU (awaiting in event to incur).

- The tasks in the wait queues are not run-able because they are awaiting events.

- task on wait queues are excluded from the scheduling effort.

/proc

- Linux provides the pseudo file system `/proc` for the purpose of viewing process information.

- Each currently executing process has a sub directory in `/proc` named by it's PID

```
# ls /proc/1
attr          coredump_filter  io          mountstats     pagemap      stack
autogroup     cpuset           limits      net            personality  stat
auxv          cwd              loginuid    ns             root         statm
cgroup        environ          maps        numa_maps      sched        status
clear_refs    exe              mem         oom_adj        schedstat    syscall
cmdline       fd               mountinfo   oom_score      sessionid    task
comm          fdinfo           mounts      oom_score_adj  smaps        wchan
```

- The contents of the files / sub directories of each of the process specific directories provide information from the processes `task_struct` and it's sub structures.

Exercise

- Choose an arbitrary process, and explore it’s properties:
  - What the command the invoked it.
  - How many files does it have currently open, and what are they?
  - What are the files that are currently mapped to its address space?

Using Floating Point

- Use of floating point operations within the Linux kernel is discouraged.

- In case it cannot be avoided, the following considerations should be taken:
  - It is emulated on many architectures.
  - The kernel configuration should enable the floating point emulation, when using float on an architecture that emulates it.
  - When running on x86 architecture, the module should be compiled with the `-mhard-float` gcc flag. `kernel_fpu_begin()` should be called before the code that uses floating point starts, and `kernel_fpu_end()` should be called at the end of that section.

Linux Kernel & Device Drivers

Chapter 4 - Loadable kernel Modules

Loadable Kernel Modules

- Loadable Kernel Modules (LKM) are object-code files (not complete executables) that can be dynamically linked to a running kernel.
- LKMs add functionality to a running kernel.
- Can be loaded and unloaded at run-time, as needed.
- Once loaded, LKMs become part of the kernel space, with full access privileges.
- Modules may be compiled statically into the kernel or as LKMs (depends on .config).

Modules dependencies

- Modules may depend on one another: module B depends on module A, if B uses a symbol exported by A.
- Dependencies are computed automatically during kernel build and written to:
  /lib/modules/<version>/modules.dep
- Dependencies can be updated using :

  depmod -a [<version>]

Modules utilities

- Listing loaded modules: lsmod
- Also available via /proc/modules
- Loading a module: insmod <module_name>.ko
- Example: `insmod floppy.ko`
- Unloading a module: rmmod <module_name>
- Example: `rmmod floppy`

Modules utilities - 2

- modprobe <module_name>

  Loads a module and its dependencies.

  Example: `modprobe lp`

- modprobe -r <module_name>

  Removes a module with its dependencies (if no longer used).

  Example: `modprobe -r lp`

- modinfo <module_path>[.ko]

  Gets information about a module (parameters, license, description, dependencies…)

  Example: `modinfo floppy.ko`

Module information

- modinfo <name>.ko prints information about the module.
- Example:

```
filename:    ./hello2.ko
author:      My name
license:     GPL
srcversion:  3B31B0BDC5F09A167C8B780
depends:
vermagic:    2.6.32-22-generic-pae SMP mod_unload
             modversions 586TSC
parm:        whom:charp
parm:        howmany:int
parm:        numbers:array of int
```

Troubleshooting loading issues

- While loading, modules sometimes emit diagnostics messages to the kernel log.
- The kernel keeps a cyclic buffer for log messages.
- Use dmesg to display the log.

Exercise - Module Utilities

1. Who uses the module ‘dm_mod’?
2. Which modules does ‘dm_mirror depend on?
3. Unload modules ‘dm_log’ and ‘dm_mirror’ with their dependencies.
   Use ‘lsmod’ to view the difference.
4. Load module ‘dm_mirror’ with its’ dependencies.
   Use ‘lsmod’ to view the difference.

Building Kernel Modules

Building modules

- Building modules requires to have a configured and built kernel tree.
- Modules won't load into a different kernels (version mismatch error on insmod).
- Modules are built using make files, compatible with the kernel's build system (a recursive set of make files).
- See kernel/Documetation/kbuild for details.

“Hello world” Example

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

static int __init hello_init(void)
{
    printk(KERN_ALERT “Hello world!\n");
    return 0;
}

static void __exit hello_exit(void)
{
    printk(KERN_ALERT “Goodbye!\n");
}

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

```
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION(“Hello module");
MODULE_AUTHOR(“My Name");
```

- “**__init**” functions are removed from RAM after module’s loading.
- “**__exit**” is left out when module is compiled statically.
- **printk** is the kernel’s equivalent of printf. The message is emitted to the kernel log.

Introducing printk

- printk is the most basic module debug mechanism.
- Syntax: printk (<priority> <format> , <args>)
- Available priorities (include/linux/kernel.h):
  - KERN_EMERG "<0>" /* system is unusable */
  - KERN_ALERT "<1>" /* action must be taken immediately */
  - KERN_CRIT "<2>" /* critical conditions */
  - KERN_ERR "<3>" /* error conditions */
  - KERN_WARNING "<4>" /* warning conditions */
  - KERN_NOTICE "<5>" /* normal but significant condition */
  - KERN_INFO "<6>" /* informational */
  - KERN_DEBUG "<7>" /* debug-level messages */
- Usually kept in /var/log/messages.
- Visible with the dmesg command.

Module license

- Linux kernel is an open-source, GPL-licensed software.
- Kernel developers are very sensitive to licensing.
- Therefore, each module must declare its' license.
- Examples:
  - GPL
  - Dual BSD/GPL
  - Dual MIT/GPL
  - GPL v2
  - Dual BSD/GPL
  - Proprietary
- 'Tainted kernel' warning is issued if some loaded module have non GPL-compatible license.

Module Makefile template

```
# Makefile for the 'hello' module

obj-m := hello.o

KDIR  ?= /lib/modules/$(shell uname -r)/build

PWD := $(pwd)

default:

	$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) M=$(PWD) modules
```

- Note the last line must begin with [TAB].
- Run make to build 'hello.ko' file.
- Compiled module names end with .ko
- You can build against a different kernel tree by defining KDIR externally.

Multiple source files

```
# Makefile for the 'hello' module – multiple source files

objm := hello.o

hello-objs := src1.o  src2.o  src3.o

KDIR  ?= /lib/modules/$(shell uname r)/build

PWD := $(pwd)

default:

    $(MAKE) C $(KDIR) SUBDIRS=$(PWD) M=$(PWD) modules
```

Exercise - ‘Hello’ module

1. Change into the exercises directory.
2. Compile the “hello” module against your kernel tree.
3. Display the module information using ‘modinfo’.
4. Load the module using ‘insmod’.
5. Observe the kernel log messages using ‘dmesg’.
6. Unload the module using ‘rmmod’ and observe the kernel log again.
7. Add a ‘MODULE_DESCRIPTION’ to the code.
8. Change the license to “Proprietary”.
9. Compile and reload module. Is there a difference?

Module Preliminaries

Handling initialization errors

- Initialization errors may prevent module from loading.
- In this case, __init function should return an error code instead of 0 (success).

```
int  __init my_init(void)
{
 item1 = allocate_thing(arguments);
if (!item1)
  return -ENOMEM;
}
```

- Take care to undo any initialization steps before returning.
- One way to do this is to call your __exit function.

Linux error codes

- In the Linux kernel, error codes are negative numbers defined in ***<linux/errno.h>***

```
#define EPERM      1    /* Operation not permitted */
#define ENOENT     2    /* No such file or directory */
#define ESRCH      3    /* No such process */
#define EINTR      4    /* Interrupted system call */
#define EIO        5    /* I/O error */
#define ENXIO      6    /* No such device or address */
#define E2BIG      7    /* Arg list too long */
#define ENOEXEC    8    /* Exec format error */
#define EBADF      9    /* Bad file number */
#define ECHILD     10   /* No child processes */
#define EAGAIN     11   /* Try again */
#define ENOMEM     12   /* Out of memory */
#define EACCES     13   /* Permission denied */
#define EFAULT     14   /* Bad address */
```

Kernel Symbol Table

- insmod resolves undefined symbols against the kernel's public **symbols table**.
- The table contains the addresses of global kernel items - functions and variables - needed by other modules.
- The table is useful when debugging kernel 'oops'.
- To view the symbol table: cat /proc/kallsyms
- See 'man nm' for explanation of the display.
- Also installed at: /boot/System.map-<kernel_ver>

Exporting Symbols

- When a module is loaded, any symbol exported by the module becomes part of the kernel symbol table.
- Usually, modules don't export any symbols unless other modules need to use it.
- To export symbols, use the following macros:
  - EXPORT_SYMBOL(name)
  - EXPORT_SYMBOL_GPL(name) - makes the symbol available to GPL-licensed modules only.
- Use these macros outside of any function.
- Other modules must declare the variables as 'extern'.

Linux Kernel & Device Drivers

Chapter 5 – Character Device Drivers

Modules & Device Drivers

- Most LKMs implement device drivers / file-system drivers.
- Three basic types of device drivers:
  1. Character device driver
  2. Block device driver
  3. Network interface driver

Character devices

- A **character (char) device** can be accessed as a stream of bytes (like a file).
- Examples: The text console (*/dev/console*), serial ports (*/dev/ttyS0)*
- Char driver usually implements at least the open, close, read and write system calls.
- Char devices are accessed via file-system nodes, such as /dev/tty1 and /dev/lp0.
- Character devices can be identified by the initial ’c’ letter (ls -l):

```
crw-rw---- 1 root uucp   4,  64 Feb 23 2004 /dev/ttyS0
crw------- 1 root root  13,  32 Feb 23 2004 /dev/input/mouse0
```

- Device nodes must be created using mknod command.

Block devices

- A **block device** is a device (e.g., a disk) that can host a file-system.
- Block devices are accessed through data blocks of a given size.
- Blocks can be accessed in any order.
- Linux allows applications to read/write to block devices like char devices; however, their kernel/driver interface completely differs.
- Like a char device, each block device is accessed through a file-system node in the */dev* directory.
- Block devices can be identified by the initial 'b' letter (ls -l):

```
brw-rw----   1 root disk     3,   1 Feb 23  2004 hda1
brw-rw----   1 jdoe floppy   2,   0 Feb 23  2004 fd0
```

Network interface drivers

- Any network transaction is made through an interface: a device able to exchange data with other hosts.
- Usually, an interface is a hardware device, but it might also be a pure software device, like the loopback interface.
- A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel.
- A **network driver** only handles packets.
- Network drivers have a packet-oriented kernel interface.
- Network drivers do not reflect as file nodes under /dev.

Device major and minor numbers

- Device files are identified by 2 numbers: major & minor numbers.
- These numbers are used by the kernel to bind a driver to the device file.
- Device file names don't matter to the kernel!

```
brw-rw----   1 root disk     3,    1 Feb 23  2004 hda1
brw-rw----   1 jdoe floppy   2,    0 Feb 23  2004 fd0
```

- To see which major numbers have been assigned, look at kernel/Documentation/devices.txt.

mknod command

- Device files are not created when a driver is loaded.
- They have to be created in advance:

  mknod /dev/<device> [c|b] <major> <minor>

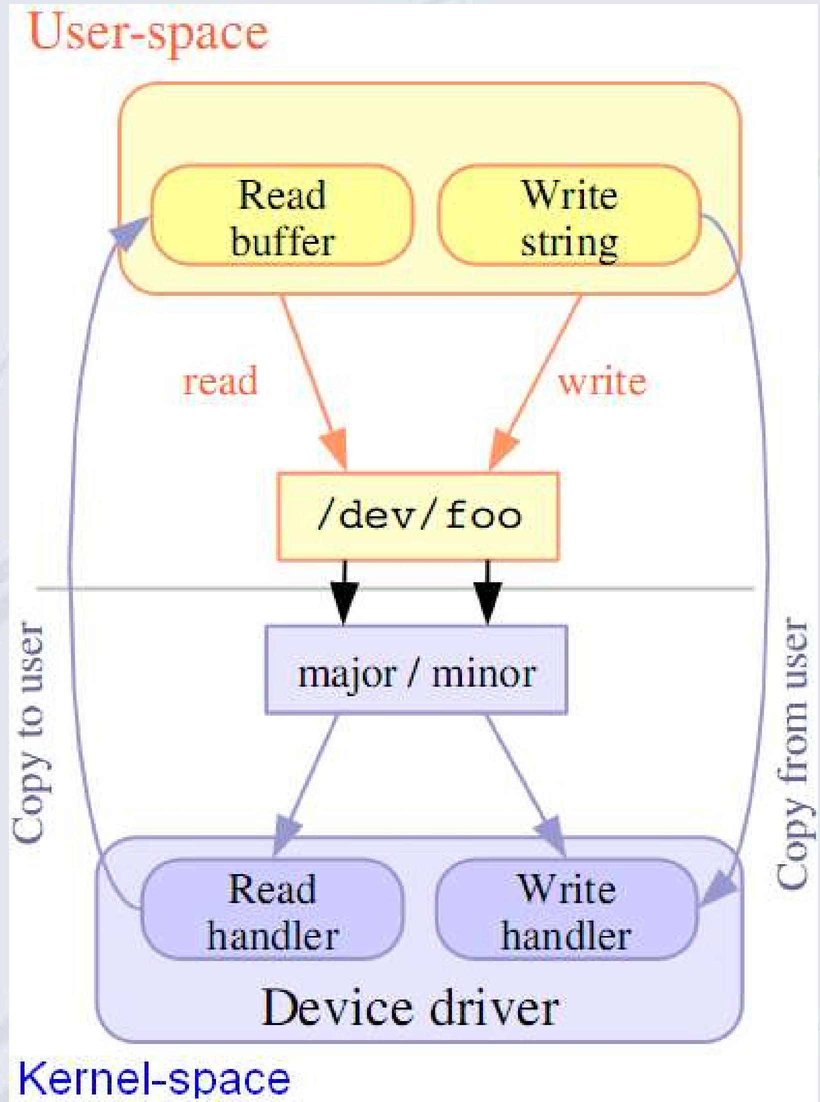
- Examples:

  mknod /dev/ttyS0 c 4 64

  mknod /dev/hda1 b 3 1

Character device driver

- The most common type of driver.
- Device name (in /dev) is needed so user applications can use the driver.
- The kernel associates a driver with a device file using the **major/minor numbers.**
- The driver provides handler functions for file operations (open, read, write, ioctl, close, etc.)



Source: http://www.free-electrons.com

Device numbers - dev_t

- Check for free major numbers in /proc/devices.
- Major/Minor pair (**device number**) is represented by dev_t type.
- Requires #include <linux/kdev_t.h>
- 32 bit size: major = 12 bits, minor = 20 bits
- To create a device number: MKDEV(int major, int minor)
- To extract the minor and major numbers:

  MAJOR(dev_t dev)

  MINOR(dev_t dev)

Static device numbers allocation

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

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

- Returns 0 if the allocation was successful.

- Example:

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

Dynamic device numbers allocation

```
int alloc_chrdev_region(dev_t *dev,  // Output: starting device number
    unsigned baseminor,              /* Starting minor number, usually 0 */
    unsigned count,                  /* Number of device numbers */
    const char *name)                /* Registered name */
```

- Returns 0 if the allocation was successful.
- You'll have to create the device node manually or by **udev**.
- Example:

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

Freeing device numbers

- Regardless of how you allocate your device numbers, you should free them when they are no longer in use, by calling:

```
void unregister_chrdev_region(dev_t first, unsigned int count)
```

- Usually in your module's cleanup function.

Creating device nodes

- Device files have to be created in advance:

  mknod /dev/<device> [c|b] <major> <minor>

- Example:

  mknod /dev/ttyS0 c 4 64

File operations

- The file_operations structure (**fops**), defined in *<linux/fs.h>*, sets up operations on a char device.
- It's made of a collection of function pointers.
- The operations mostly implement system calls (open, read, ioctl, etc.) - about 25 operations.
- Each field must point to the function in the driver that implements a specific operation, or left NULL for unsupported operations.
- Each open file has its own set of functions (via pointer).
- We can consider the file as an "object" and the file operation functions as its "methods".

struct file_operations

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*lock) (struct file *, int, struct file_lock *);
    …
```

- List at http://www.makelinux.net/ldd3/chp-3-sect-3.shtml

The file structure

- The file structure represents an open file. struct file is created by the kernel during the open call.
- Structure contains:

mode_t f_mode

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

unsigned int f_flags

- File flags (O_CREAT, O_APPEND, etc.)

loff_t f_pos

- Current offset in the file.

struct file_operations *f_op

- Allows to change file operations for different open files!

struct dentry *f_dentry

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

File operations (2)

```
ssize_t (*read) (struct file *,     /* Open file descriptor */
    __user 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 program reads from the device file.

```
ssize_t (*write) (struct file *,    /* Open file descriptor */
    __user 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 program writes to the device file.

Kernel-User space data exchange

- In driver code, you can't just memcpy between an address supplied by user-space and the address of a buffer in kernel-space!
- Completely different address spaces, due 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).
- Use the following functions from <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 they return 0 - otherwise, they failed!

File operations (3)

```
ssize_t (*open) (struct inode *,       /* inode descriptor */
    struct file *)                     /* Open file descriptor */
```

- Called when user-space program opens the device file.

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

- Called when user-space program closes the file.

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

- Can be used to send specific commands to the device, neither reading nor writing (e.g. your own commands).
- See include/asm-generic/ioctl.h for details on numbering scheme.

Character device registration

The kernel represents character drivers with a cdev structure:

1. Declare this structure globally (within your module):

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

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

```
my_cdev = cdev_alloc();
my_cdev->ops = &my_fops;
my_cdev->owner = THIS_MODULE;
```

3. Then 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 */
```

Character device example (1)

```
static dev_t acme_dev = MKDEV(92,0);
static struct cdev acme_cdev;

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

static int __init acme_init(void)
{
    int err;
    acme_buf = vmalloc (acme_bufsize);

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

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

    cdev_init(&acme_cdev, &acme_fops);

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

     return 0;

    err_dev_unregister:
         unregister_chrdev_region(acme_dev,
       acme_count);
    err_free_buf:
         vfree (acme_buf);
    err_exit:
         return err;
}
```

Source: http://www.free-electrons.com

Character device unregistration

1. First delete your character device:

   void cdev_del (struct cdev *p)

2. Then, and only then, free the device number:

   void unregister_chrdev_region (dev_t from, unsigned count)

- Example (continued):

```
static void __exit acme_exit(void)
{
    cdev_del (&acme_cdev);
    unregister_chrdev_region(acme_dev, acme_count);
    vfree (acme_buf);
}
```

Character device example (2)

```
static ssize_t acme_read (struct file *file, char
      __user *buf, size_t count, loff_t * ppos)
{
   int remaining_size, transfer_size;

   remaining_size = acme_buffer_filled - (int)
      (*ppos); // bytes left to transfer
   if (remaining_size == 0) {
       /* All read, returning 0 (End Of File) */
      return 0;
   }

   /* Size of this transfer */
   transfer_size = min(remaining_size, (int)
      count);

   if (copy_to_user(buf /* to */, acme_buf +
      *ppos /* from */, transfer_size)) {
      return -EFAULT;
   } else {
      /* Increase the position in the open file */
      *ppos += transfer_size;
      return transfer_size;
   }
}
```

```
static ssize_t acme_write(struct file *file, const
      char __user *buf, size_t count, loff_t * ppos)
      {
   int remaining_bytes;

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

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

   if (copy_from_user(acme_buf + *ppos /* to */,
      buf /* from */, count)) {
      return -EFAULT;
   } else {
        // Increase the position in the open file
      *ppos += count;
        // Increase the position in the device buffer
       acme_buffer_filled = count;
      return count;
   }
}
```

Source: http://www.free-electrons.com

Exercise - Character device

1. Compile the sample driver ‘acme.c’ and install it. View the major number registered in /proc/devices.
2. Create an appropriate device node and access the driver.
3. Modify the code to dynamically allocate its’ buffer upon ‘write’ file operation.
4. Implement the ‘open’ file operation, so it distinguishes ‘append’ from ‘create’ file mode.
5. Hint: Check the file structure for O_CREAT and O_APPEND flags.
6. In case of create, existing data buffers should be freed.
7. In case of append, consecutive writes should add to the current buffer contents.

ioctl

- IOCTL = “Input and Output Control”
- Used for sending “out-of-band” commands to a device
- Steps to use IOCTL:
  1. Create IOCTL command in driver
  2. Write IOCTL function in the driver
  3. Create IOCTL command in a Userspace application
  4. Use the IOCTL system call in a Userspace

Write IOCTL function in the driver

```
static long etx_ioctl (struct file *file, unsigned int cmd, unsigned long arg)
{
        switch (cmd)  {
                case WR_VALUE:
                        if (copy_from_user(&value ,(int32_t*) arg, sizeof(value)) )        {
                                pr_err("Data Write : Err!\n");
                        }
                        break;
                case RD_VALUE:
                        if (copy_to_user((int32_t*) arg, &value, sizeof(value)) )          {
                                pr_err("Data Read : Err!\n");
                        }
                        break;
                default:
                        pr_info("Default\n");
                        break;
        }
        return 0;
}
```

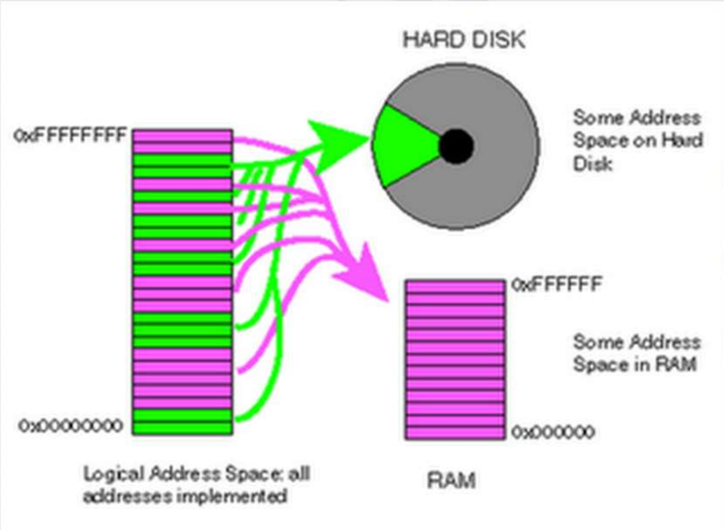
ioctl command in userspace app.

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

#define WR_VALUE  _IOW ('a','a',int32_t*)
#define RD_VALUE  _IOR  ('a','b',int32_t*)

int fd;
int32_t value, number;

fd = open("/dev/etx_device", O_RDWR);
if (fd < 0) {
        printf("Cannot open device file...\n");
        return 0;
}

ioctl (fd, WR_VALUE, (int32_t*) &number);

ioctl (fd, RD_VALUE, (int32_t*) &value);
printf("Value is %d\n", value);
```

Exercise - ioctl

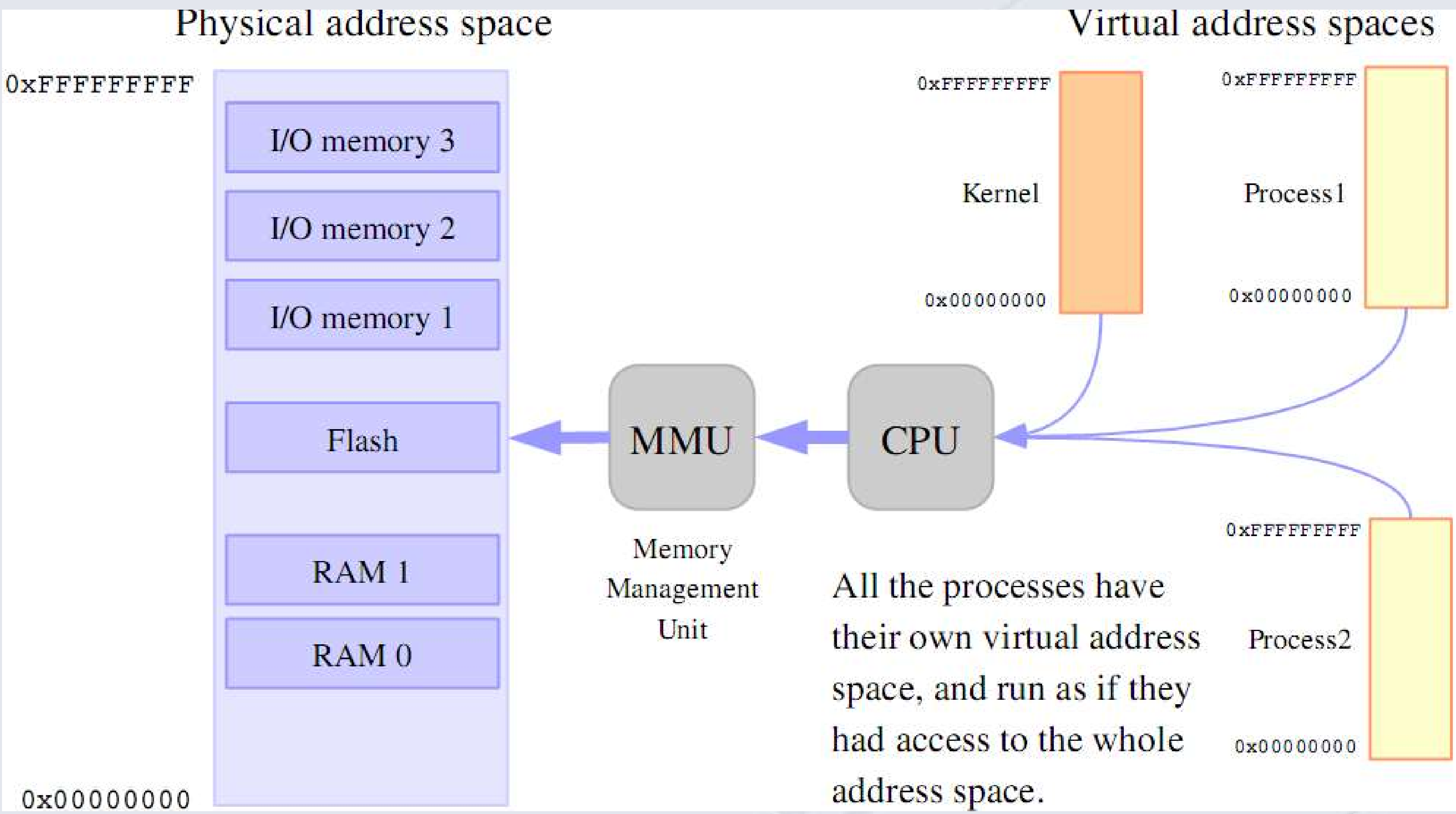
1. Compile the sample driver ‘driver.c’ and install it.
2. View the major number registered in ‘dmesg’.
3. Create an appropriate device node to access the driver.
4. Complete the implementation of “etx_ioctl” function in the driver:
   - ‘write’ command saves an integer from userspace in a variable
   - ‘read’ command returns the saved integer
5. Compile the ‘test_app.c’ user application: `gcc -o test_app test_app.c`
6. Run ./test_app and send ioctl’s to the driver.

Virtual Memory Overview



http://techtitude.blogspot.com/2009/10/virtual-memory-concept.html

Physical and Virtual Memory



Source: http://www.free-electrons.com

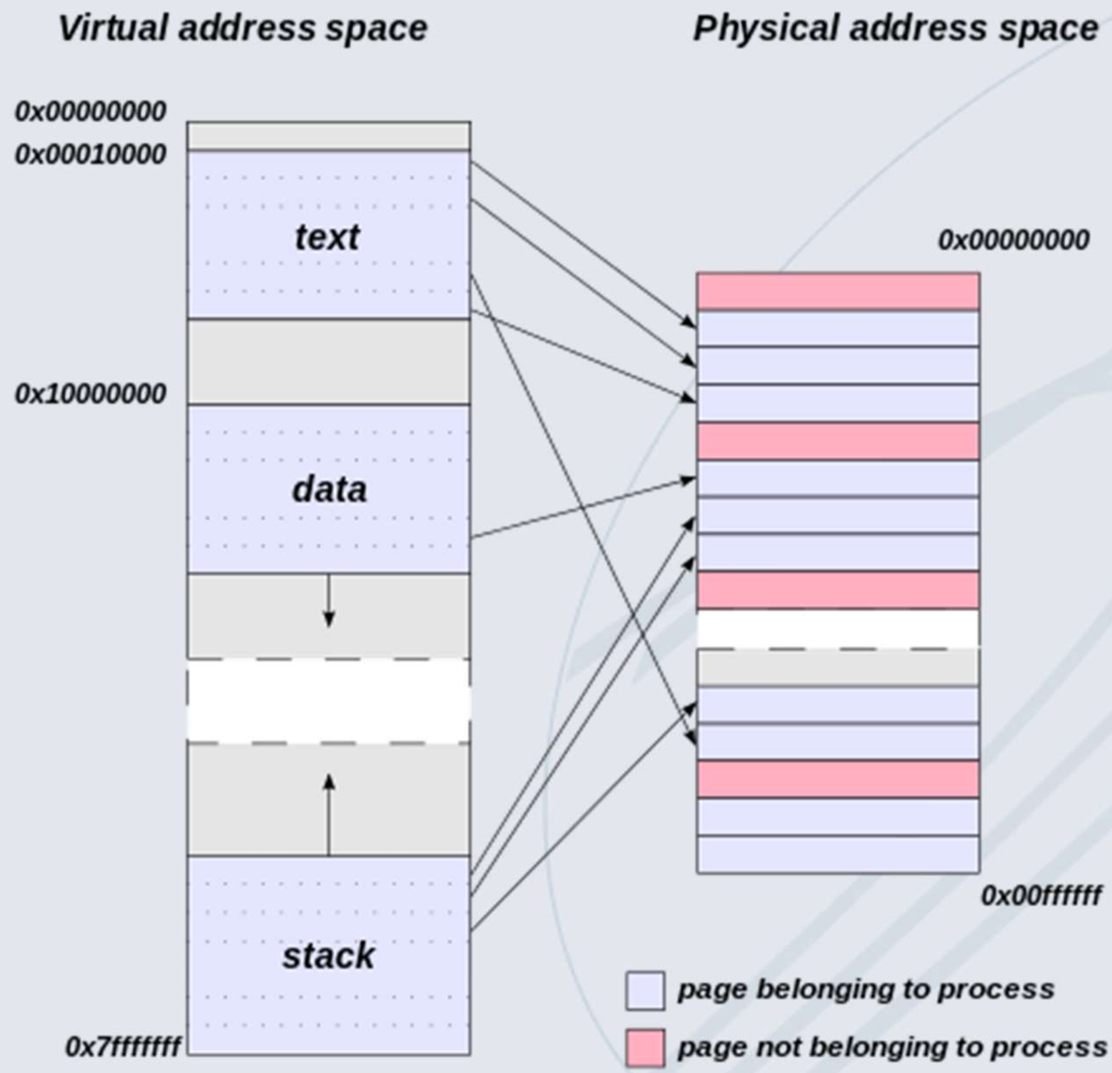
Memory Management Concepts

- **Physical address:** Corresponds to the electrical signals sent along the address pins of the microprocessor to the memory bus.
- **Virtual address:** A single unsigned integer, that can be used to address a memory cell. Each program runs with virtual addresses.
- **Virtual memory**: The MMU hardware maps virtual addresses to physical addresses.
- **Memory protection**: Processes can only access their own address space. Errors are detected by hardware, kernel kills the process.
- Addresses are grouped in fixed-length intervals called **Pages** (almost always 4KB each).
- **On-Demand Paging**: The kernel only loads the parts of executables and libraries as they are actually used.

Page Frames & Swapping

- **Pages** (groups of data) are distinguished from **page frames** (groups of memory addresses).
- **Swapping** is used to dynamically allocate physical pages to processes.
- A page may be stored in a page frame, then saved to disk and later reloaded in a different page frame.
- Access to a swapped page causes an exception, being handled by the kernel's memory management (mm) code.
- On PC, usually Linux has a dedicated swap partition (or swap file).

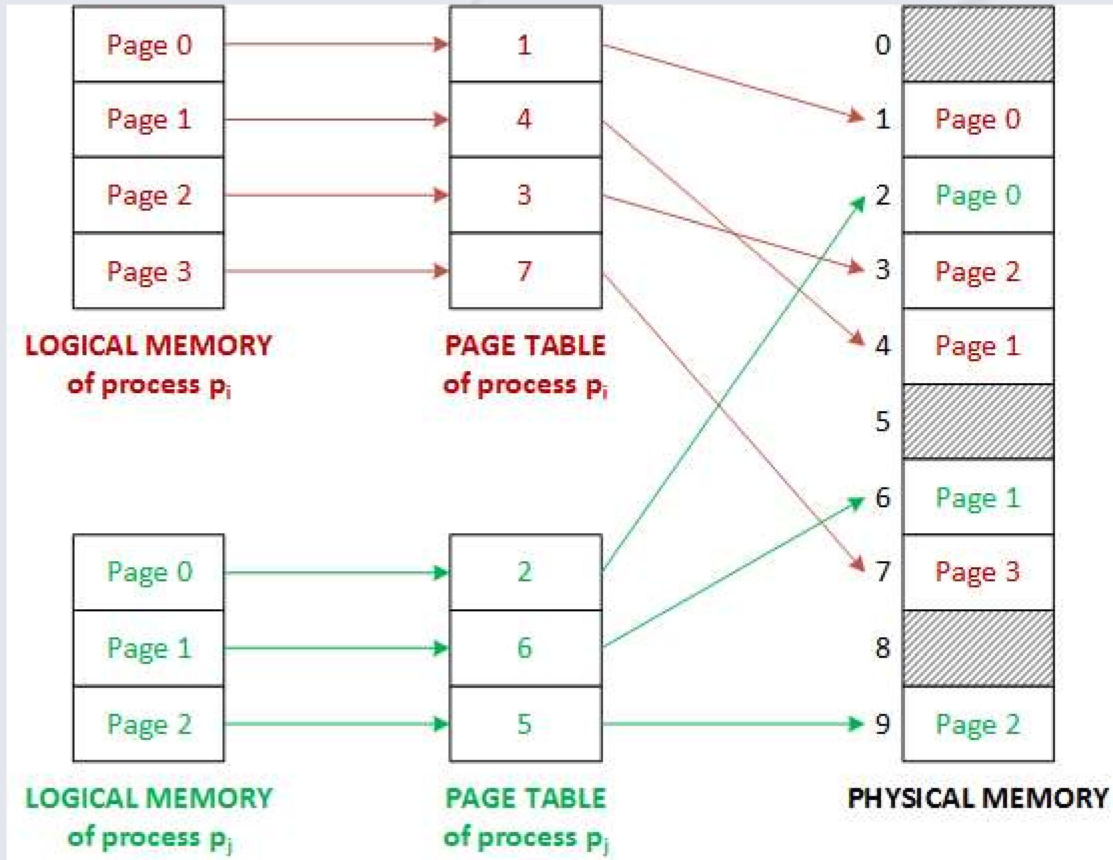
Process Memory Map



Source: https://en.wikipedia.org/wiki/Page_table

Page Tables

- Data structures mapping virtual to physical addresses are called **Page Tables**.
- They are stored in main memory and must be properly initialized by the kernel.



Source: https://i.stack.imgur.com/e2NNe.png

Memory Protection

- For each page, the page tables store permissions flags that determine:
  - **Read/Write flag**: Contains the access rights (Read/Write or Read) of the page.
  - **User/Supervisor flag**: Contains the privilege level required to access the page.
- Each process has different page tables.
- Upon context switch, pointers to the page tables are changed so process is limited to access only its own page frames.

Kernel Memory Allocation

kmalloc and kfree

- Basic allocators, kernel equivalents of ’malloc’ and ’free’.
- Requires #include <linux/slab.h>
- static inline void * kmalloc(size_t size, int flags)
  - size: number of bytes to allocate
  - flags: priority
- void kfree (const void *objp)
- Example:

```
work = kmalloc (sizeof (*work), GFP_ATOMIC);
...
kfree(work);
```

kmalloc flags

- Defined in include/linux/gfp.h (GFP: __get_free_pages)
- GFP_KERNEL
  - Standard kernel memory allocation.
  - May block - caution!!!
  - Fine for most needs.
- GFP_ATOMIC
  - Allocated RAM from interrupt handlers or code not triggered by user processes.
  - Never blocks - if allocation fails, returns with an error.

Other allocation functions

- static inline void *kzalloc (size_t size, gfp_t flags)

  Zeroes the allocated buffer.

- static inline void *kcalloc (size_t n, size_t size, gfp_t flags)

  Allocates memory for an array of n elements of size size, and zeroes its contents.

- void * krealloc (const void *, size_t size, gfp_t flags)

  Changes the size of the given buffer.

vmalloc

- If you do not need physically contiguous memory, use vmalloc():

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

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

- You'll get contiguous virtual addresses, not suitable for DMA.
- To free the memory, use vfree(void *mem).
- Example:

```
p = vmalloc(sizeof (struct data));
if (!p)
    /* error */
…
vfree(p);
```

Memory functions

- Lots of functions equivalent to standard C library ones defined in include/linux/string.h.
- Examples:
  - void * memset (void * s, int c, size_t count)
    - Fills a region of memory with the given value.
  - void * memcpy (void * dest, const void *src, size_t count)
    - Copies one area of memory to another.
    - Use memmove with overlapping areas.

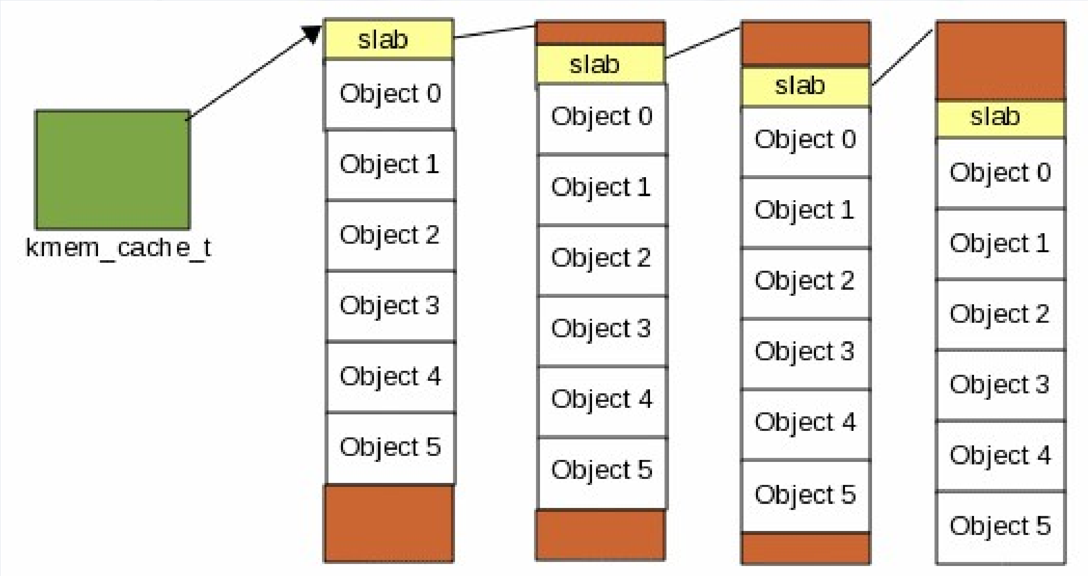
Exercise - Memory Allocation

1. Write a module that finds out the largest memory block that can allocated in the kernel.
2. The module shall try to allocate increasing amounts, until allocation fails.
3. Once failed, the module shall print the attempted size.
4. Try to allocate with kmalloc and then with vmalloc.
   Is there a difference?

Fixed-size Memory Manager (SLAB/SLUB/SLOB)

The SLAB Allocator

- The **SLAB Allocator** is a memory manager for small, fixed-size allocations.
- Mainly used by Linux core subsystems: filesystems (open files, inode and file caches...), networking, USB / SCSI drivers...
- Live stats on /proc/slabinfo.



Source: http://www.secretmango.com/jimb/Whitepapers/slabs/slab.html

SLAB Cache Creation

- Modules can declare caches:

```
static struct kmem_cache *my_cachep;

struct kmem_cache * kmem_cache_create (const char *name,
    size_t size, size_t offset, unsigned long flags;
    void (*ctor)(void*, struct kmem_cache *, unsigned long))
```

- **name -** identifies the cache in /proc/slabinfo
- **size -** the size of an object in this cache
- **offset -** offset of the first object in the page; usually 0
- **flags** - bit mask of:
  - SLAB_NO_REAP - protects the cache from being reduced
  - SLAB_HWCACHE_ALIGN - each data object shall be aligned to a cache line
  - SLAB_CACHE_DMA - each data object to shall be allocated in ZONE_DMA

SLAB Cache Removal

- To remove an allocated cache:
  void **kmem_cache_destroy** (struct kmem_cache *cachep)

- Usually called by kernel modules when they are unloaded.
- The cache must be empty before this function is called.

SLAB Cache Example (1)

```
static struct kmem_cache *my_cachep;

static void init_my_cache (void )
{
   my_cachep = kmem_cache_create(
                  "my_cache",            /* Name */
                  32,                    /* Object Size */
                  0,                     /* Alignment */
                  SLAB_HWCACHE_ALIGN,    /* Flags */
                  NULL);          /* Constructor */

   return;
}

static void remove_my_cache (void)
{
  if (my_cachep)
                kmem_cache_destroy( my_cachep );
  return;
}
```

kmem_cache_alloc / kmem_cache_free

- To allocate an object from a cache, use:

  void* **kmem_cache_alloc** (struct kmem_cache *cachep,
  gfp_t flags)

  - **cachep** - previously returned by kmem_cache_create
  - **flags** - same as for kmalloc (GFP_KERNEL, GFP_ATOMIC)
  - **kmem_cache_zalloc()** - clear the returned object (memset).

- To return an object to the cache, use:

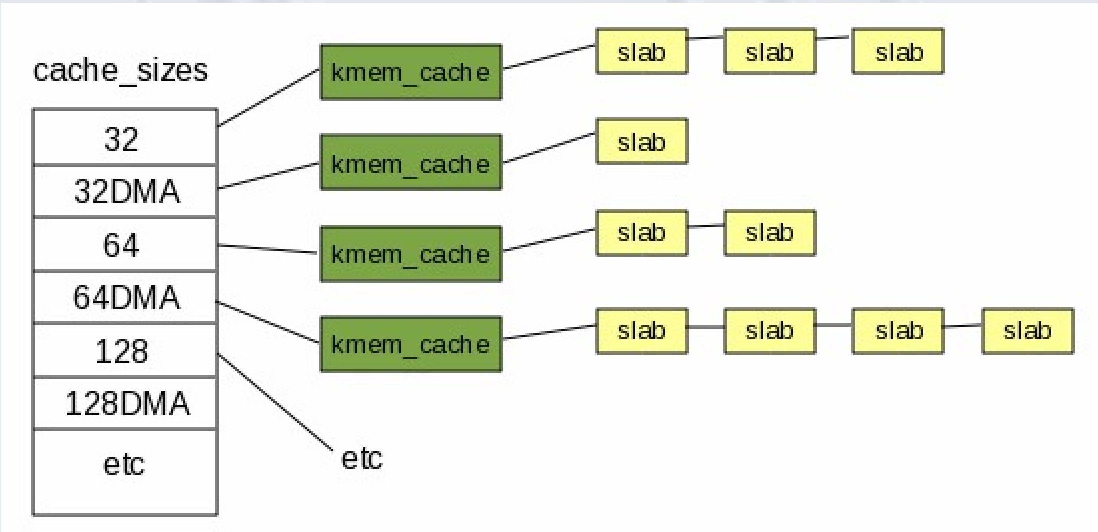
  void **kmem_cache_free** (struct kmem_cache *cachep,
  void *objp)

SLAB Cache Example (2)

```
int slab_test (void)
{
  void *object;

  printk( "Cache name is %s\n", my_cachep->name );
  printk( "Cache object size is %d\n", kmem_cache_size( my_cachep ) );

  object = kmem_cache_alloc (my_cachep, GFP_KERNEL );

  if (object) {

    kmem_cache_free( my_cachep, object );

  }

  return 0;
}
```

SLAB and kmalloc/kfree

- The kernel ties kmalloc() / kfree() calls into the slab allocator.
- On initialization, the kernel asks the slab allocator to create several caches of varying sizes.
- Caches for generic objects of 32, 64, 128, 256, all the way to 131072 bytes are created.
- When a module calls kmalloc(), the **cache_sizes** array is searched to find the cache with the size appropriate to fit the requested object.



Source: http://www.secretmango.com/jimb/Whitepapers/slabs/slab.html

SLOB & SLUB Allocators

- **SLOB** (**S**imple **L**ist **O**f **B**locks) allocator replaces 'slab' for small systems.
- SLOB is a traditional heap implementation, in a single linked-list of pages and 'first-fit' policy.
- It has much lower overhead, but scales poorly and does not handle fragmentation as well.

- **SLUB** (**S**imple **L**ist of **U**nqueued **B**locks) is a drop-in replacement (same API), with different implementation better for multi-CPU machines.
- SLUB scales better, but is considered slower than SLAB.
- SLUB is the default allocator!

Exercise – Slab allocator

1. Check your kernel .config – which allocator is used?
2. Write a module that accepts two parameters: Size & Count
3. The module shall create a slab cache for objects of size ‘Size’.
4. The module shall allocate ‘Count’ objects from the slab.
5. The module shall free the cache upon exit.
6. Load your module with different sizes.
7. Observe the information in /proc/slabinfo or slabtop after every attempt. Try to identify ‘your’ cache.

I/O Memory & Ports

I/O Ports

- /proc/ioports:

```
0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0060-0060 : keyboard
0064-0064 : keyboard
0070-0071 : rtc0
0080-008f : dma page reg
00a0-00a1 : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : 0000:00:1f.1
  0170-0177 : ata_piix
01f0-01f7 : 0000:00:1f.1
…
```

- To reserve an I/O region:

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

- Example:

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

- To release:

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

- See include/linux/ioport.h

Accessing I/O Ports

- Platform-specific implementation!
- Bytes:

  unsigned **inb** (unsigned port);

  void **outb** (unsigned char byte, unsigned port);
- Words:

  unsigned **inw** (unsigned port);

  void **outw** (unsigned char byte, unsigned port);
- Long integers:

  unsigned **inl** (unsigned port);

  void **outl** (unsigned char byte, unsigned port);

String I/O Access

- Requires processor support!
- Byte strings:

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

  void **outsb** (unsigned port, void *addr, unsigned long count);
- Word strings:

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

  void **outsw** (unsigned port, void *addr, unsigned long count);
- Long strings:

  void **insl** (unsigned port, void *addr, unsigned long count);

  void **outsl** (unsigned port, void *addr, unsigned long count);

I/O Memory

- /proc/iomem:

```
00000000-0000ffff : reserved
00010000-0009fbff : System RAM
0009fc00-0009ffff : reserved
000a0000-000bffff : Video RAM
   area
000c0000-000ccfff : Video ROM
000e4000-000fffff : reserved
  000f0000-000fffff : System
   ROM
00100000-5ffaffff : System RAM
  00100000-005b5530 : Kernel
   code
  005b5531-007dc367 : Kernel
   data
 ...
```

- To reserve a region:

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

- To release:

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

Mapping I/O Memory

```
#include <asm/io.h>
…
void *ioremap (unsigned long phys_addr, unsigned long size);
```

- Returns the equivalent virtual address.

```
void iounmap (void *address);
```

- Releases the virtual address mapping.

Accessing I/O Memory

- Directly reading/writing from/to I/O re-mapped addresses is architecture-dependent. It may not work as expected!
- Portable and safe I/O access functions:

  unsigned int **ioread8/16/32** (void *addr);

  void **iowrite8/16/32** (u8 value, void *addr);

- 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);

- Also:

  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);

Memory Address Translations

- Requires #include <asm/io.h>

```
unsigned long virt_to_phys (void *x);
```

- Translates a (kernel) virtual address to a physical address.
- May not provide bus mappings for DMA transfers.

```
void *phys_to_virt (unsigned long x);
```

- Translates a physical address to a virtual (kernel) address.
- For platforms with IOMMU:

```
bus_addr = virt_to_bus (virt_addr);
virt_addr = bus_to_virt (bus_addr);
```

- Translates DMA-bus addresses to virtual addresses and back (see kernel/Documentation/DMA-API.txt)

LOGTEL
It's more than knowledge

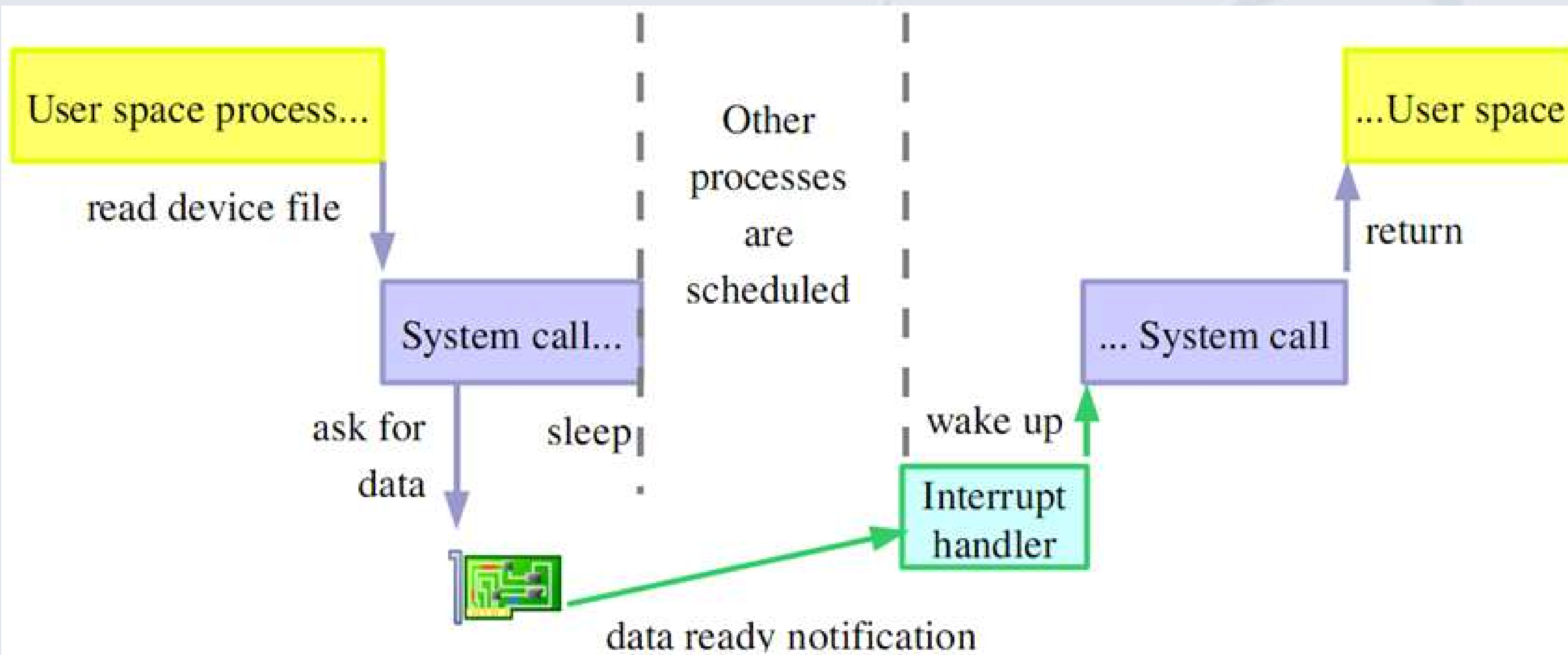
Linux Kernel & Device Drivers

Chapter 6 – Kernel Space Considerations

Sleeping in the Kernel

Sleeping

- Needed when a process is waiting for data (or other event).



Source: http://free-electrons.com

How to sleep (1)

- First, declare a wait queue, either statically:

  DECLARE_WAIT_QUEUE_HEAD (module_queue);

  - or dynamically:

  wait_queue_head_t queue;

  init_waitqueue_head(&queue);

- Several ways to make a kernel process sleep:

  wait_event (queue, condition);

  - Sleeps until the given C expression is true.
  - Can't be interrupted, i.e. by killing the process in user-space.
  - Process goes to TASK_UNINTERRUPTIBLE state.

How to sleep (2)

wait_event_interruptible (queue, condition);

- Can be interrupted (TASK_INTERRUPTIBLE state).

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.

- Example:

```
wait_event_interruptible (d->waitq, (d-
    >buffer_status[v.buffer] == VIDEO1394_BUFFER_READY) );

if (signal_pending(current))

    return -EINTR;
```

Waking up

wake_up (queue);

- Wakes up all the waiting processes on the given queue.

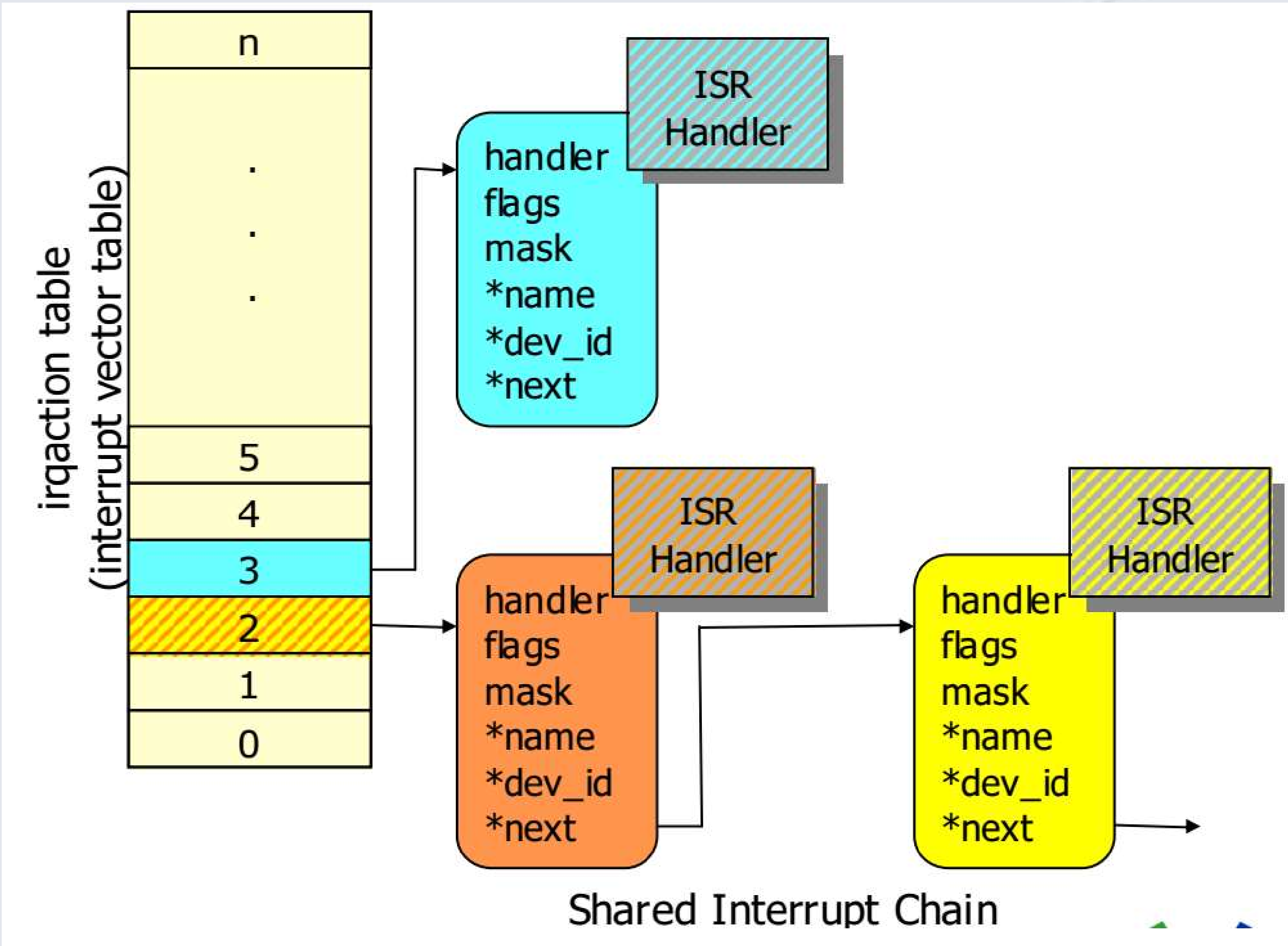
wake_up_interruptible (queue);

- For processes waiting using wait_event_interruptible.

- Typically done by interrupt handlers when data becomes available.
- Upon the event, conditions for all processes waiting in queue is evaluated.
- When it evaluates to true, the process is put back to the TASK_RUNNING state.

Interrupt Handling

Interrupt Handlers are Chained



Source: http://elinux.org/images/b/b6/InterruptThreads_Anderson.pdf

Interrupts information

- /proc/interrupts

```
       CPU0
0:     5616905          XTPIC  timer
1:        9828          XTPIC  i8042
2:           0          XTPIC  cascade
3:     1014243          XTPIC  orinoco_cs
7:         184          XTPIC  Intel 82801DBICH4
8:           1          XTPIC  rtc
9:           2          XTPIC  acpi
11:     566583          XTPIC  ehci_hcd, uhci_hcd,
uhci_hcd, uhci_hcd, yenta, radeon@PCI:1:0:0
12:       5466          XTPIC  i8042
14:     121043          XTPIC  ide0
15:     200888          XTPIC  ide1
```

Registering an interrupt handler

- Defined in <include/linux/interrupt.h>

```
int request_irq (
    unsigned int irq,            // Requested irq channel
    irqreturn_t handler,         // Interrupt handler
    unsigned long irq_flags,     // Option mask (see next page)
    const char * devname,        // Registered name
    void *dev_id);               // Pointer to some handler data.
                                   // Can't be NULL; must be unique
                                   // for shared irqs!
```

- Returns 0 if successful.

- To unhook a handler:
  void **free_irq** (unsigned int irq, void *dev_id);

irq_flags

Typical irq_flags bit values (can be OR'd):

- IRQF_DISABLED
  - "Quick" interrupt handler. Run with all interrupts disabled on the current CPU (instead of just the current line).
- 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.
- IRQF_TRIGGER_*
  - IRQF_TRIGGER_NONE, IRQF_TRIGGER_RISING, IRQF_TRIGGER_FALLING, IRQF_TRIGGER_HIGH, IRQF_TRIGGER_LOW

Interrupt handler's tasks

1. Clear the interrupt flag of the device.
2. Read/write data from/to the device.
3. Wake up processes waiting for this operation:

```
wake_up_interruptible (&module_queue);
```

Interrupt handler prototype

```
irqreturn_t (*handler) (
  int irq,          // irq number of the current interrupt.
  void *dev_id);    // Pointer used to keep track of the specific device.
                    // Useful when module manages several devices.
```

Return value:

- IRQ_HANDLED: recognized and handled interrupt.
- IRQ_NONE: Not handled by the module. Useful for shared interrupts and/or to report spurious interrupts.
- IRQ_WAKE_THREAD: For threaded interrupts.

Disabling Interrupts

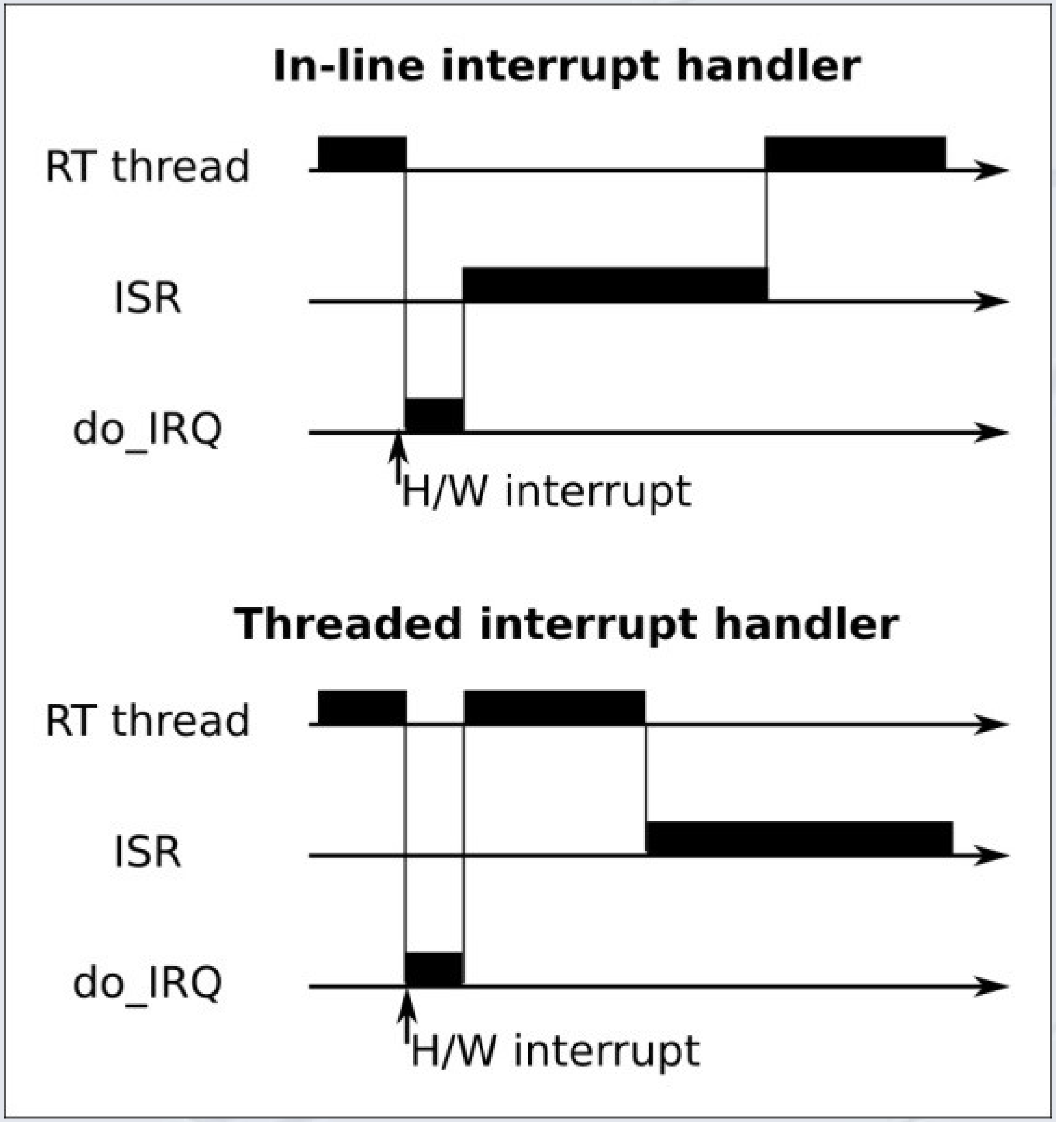
- Disabling interrupts on the local CPU:

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

- Must be run from within the same function!

Threaded Interrupts

- Comes from the PREEMPT_RT patch.
- All IRQs execute in high-priority kernel threads: Highest priority wins.
- Same priority threads will run in the order they were scheduled.
- Threads are created automatically by **request_irq()**
- Your ISR shall be called by the kernel thread.
- Only another ISR or higher-priority thread will preempt you.



https://www.packtpub.com/mapt/book/networking_and_servers/9781784392536/14/ch14lvl1sec151/threaded-interrupt-handlers

Threaded Interrupts API

```
int request_threaded_irq (unsigned int irq,     // irq number
        irq_handler_t handler,                  // ISR ‘fast’
        irq_handler_t thread_fn,                // Thread function
        unsigned long irqflags,                 // IRQF_* Flags
        const char *devname,                    // Registered name
        void *dev_id)                           // Unique pointer
```

- typedef **irqreturn_t** (*irq_handler_t) (int irq, void *data);

- Note: Fast handler should return IRQ_WAKE_THREAD.

Exercise – Interrupts

1. Compile the module from “interrupt.c”.
2. Display /proc/interrupts and select a used IRQ line.
3. Insert the module passing the IRQ number you chose:

```
insmod interrupt.ko irq=NN
```

4. Display ‘dmesg’ several times and observe module messages.
5. Remove module from kernel.

Exercise – Interrupts (2)

1. Use the source file “ktime.c” as an example of how to get the current time.
2. Modify your “interrupt.c” module to calculate and print the time elapsed between each consecutive interrupts.
3. Compile the module.
4. Display /proc/interrupts and select a used IRQ line.
5. Insert the module passing the IRQ number you chose:

   ```
   insmod interrupt.ko irq=NN
   ```
6. Display dmesg several times and observe module messages.
7. Remove module from kernel.

Exercise – Interrupts (3)

1. Convert your "interrupt.c" module to use threaded interrupt.
2. Calculate and print the time elapsed between the "fast" handler and the "thread_fn" (latency).
3. Compile the module.
4. Display /proc/interrupts and select a used IRQ line.
5. Insert the module passing the IRQ number you chose:

   ```
   insmod interrupt.ko irq=NN
   ```

6. Display dmesg several times and observe module messages.
7. Remove module from kernel.

Exercise – Sleeping

1. Modify your “acme.c” module as following:
   - In “open” handler: register an interrupt handler.
   - In “read” handler: suspend the process before return.
   - In ISR: count 30 interrupts, then wake-up the process.
   - In “module_exit”: unregister the interrupt handler (if hooked).
2. Compile and load the module.
3. Create a device node for it (mknod).
4. Read from the device node. What happens?
5. Remove module from kernel.

Top and Bottom halves

Top half / bottom half concept

Splitting the execution of interrupt handlers in 2 parts:

1. Top half:

- The interrupt handler; completes as quickly as possible.
- It saves somewhere in RAM all information needed later for completely handling the event.
- Schedules the rest of the job for later execution.
- Before terminating, re-enables interrupt notifications for the local CPU.

2. Bottom half:

- Completing the rest of the interrupt handling.
- Handles data, and then wakes up any waiting user process.
- Best implemented by **tasklets**.

tasklets (1)

- **tasklets** are executed right after all interrupt handlers.
- One tasklet runs only on one CPU.
- Different tasklets may run simultaneously on different CPUs
- To declare a tasklet:

```
DECLARE_TASKLET (module_tasklet,        // Name
    module_do_tasklet,                  // Function
    tasklet_data);                      // tasklet argument
```

tasklets (2)

- To schedule a tasklet (in the interrupt handler):

  tasklet_schedule (&module_tasklet);

  tasklet_hi_schedule (&module_tasklet);

  - Defines high priority tasklets that run first.

- tasklet prototype:

void (***func**) (unsigned long);   // declared in DECLARE_TASKLET

Tasklet example

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/interrupt.h>

char my_tasklet_data[]="my_tasklet_function was
      called";

/* Bottom Half Function */
void my_tasklet_function( unsigned long data )
{
      printk( "%s\n", (char *)data );
      return;
}

DECLARE_TASKLET( my_tasklet,
      my_tasklet_function, (unsigned long)
      &my_tasklet_data );
```

```
int init_module( void )
{
      /* Schedule the Bottom Half */
      tasklet_schedule( &my_tasklet );
      return 0;
}


void cleanup_module( void )
{
      /* Stop the tasklet before we exit */
      tasklet_kill( &my_tasklet );
      return;
}
```

Source: http://public.dhe.ibm.com/software/dw/linux/l-tasklets/l-tasklets-pdf.pdf

Exercise – Tasklets

1. Modify your "interrupt.c" module as following:
   - In the interrupt handler, schedule a tasklet for later execution.
   - In the tasklet function, calculate and print the time difference between ISR and tasklet.
2. Compile and load the module.
3. Watch dmesg output.

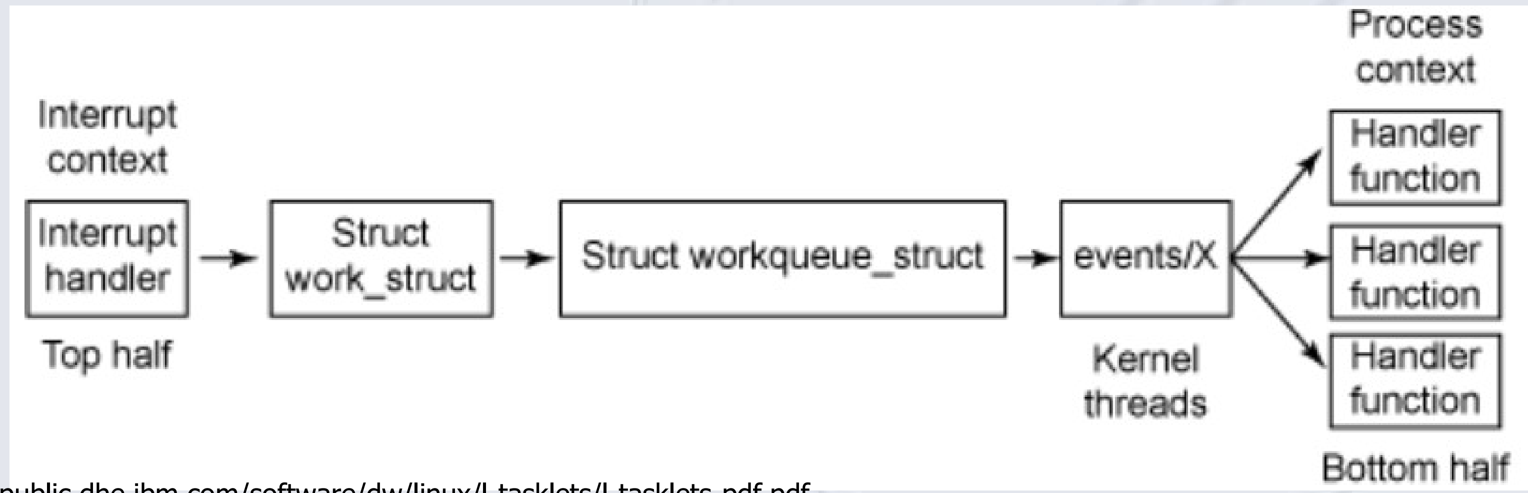
Hints:

- Use the function **ktime_get_ns().**
- Don't forget to cancel the tasklet and unhook the interrupt handler in your module_cleanup!

Deferring work

Work queues

- **Work queues** is another mechanism for deferring work.
- Unlike tasklets, work queues are managed by a kernel thread, so they can sleep.
- Work queues provide a flexible API that permits queuing of multiple work items.
- Work queues have greater latency than tasklets.
- Work items are associated with the specific CPU.



Source: http://public.dhe.ibm.com/software/dw/linux/l-tasklets/l-tasklets-pdf.pdf

Kernel-global work queue functions

int **schedule_work** (struct work_struct *work);

int **schedule_work_on** (int cpu, struct work_struct *work);

int **scheduled_delayed_work** (struct delayed_work *dwork, unsigned long delay);

int **scheduled_delayed_work_on** (int cpu, struct delayed_work *dwork, unsigned long delay);

Work queue API (1)

- See kernel/workqueue.c and include/linux/workqueue.h

- Work queue construction & destruction:

  struct workqueue_struct ***create_workqueue** (name);

  void **destroy_workqueue** (struct workqueue_struct *);

- Work initialization macros:

  INIT_WORK(work, func);

  INIT_DELAYED_WORK (work, func);

  INIT_DELAYED_WORK_DEFERRABLE (work, func);

Work queue API (2)

- Work queueing:

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

  int **queue_work_on** (int cpu, struct workqueue_struct *wq, struct work_struct *work)

  int **queue_delayed_work** (struct workqueue_struct *wq, struct delayed_work *dwork, unsigned long delay)

  int **queue_delayed_work_on** (int cpu, struct workqueue_struct *wq, struct delayed_work *dwork, unsigned long delay)

- To find out whether a work item is currently pending:

  work_pending (work)

  delayed_work_pending (work)

Work queue flushing

int **flush_work** (struct work_struct *work)

- Flushes a specific, pending work item from the queue.
- Blocks until operation completes.

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

- Flushes all pending work items on a given work queue.
- Blocks until operation completes.

void **flush_scheduled_work** (void)

- Flushes the kernel-global work queue.

Work cancellation

int **cancel_work_sync** (struct work_struct *work)

- Cancel work item, if not already executing.
- If the work item is already in progress, blocks until the callback has finished.

int **cancel_delayed_work_sync** (struct delayed_work *dwork)

- Cancels a delayed work item.

Work queues example (1)

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

static struct workqueue_struct *my_wq;

typedef struct {
  struct work_struct my_work;
  int    x;
} my_work_t;

my_work_t *work, *work2;

static void my_wq_function( struct
      work_struct *work)
{
  my_work_t *my_work = (my_work_t *)work;
  printk( "my_work.x %d\n", my_work->x );
  kfree( (void *)work );
  return;
}
```

```
int init_module( void )
{
  int ret;
  my_wq = create_workqueue("my_queue");

  if (my_wq) {
    /* Queue some work (item 1) */
    work = (my_work_t *)
      kmalloc(sizeof(my_work_t), GFP_KERNEL);

    if (work) {
      INIT_WORK( (struct work_struct *)work,
      my_wq_function );

      work->x = 1;
      ret = queue_work( my_wq, (struct
      work_struct *)work );
    }
```

Source: http://public.dhe.ibm.com/software/dw/linux/l-tasklets/l-tasklets-pdf.pdf

Work queues example (2)

```
/* Queue some additional work (item 2) */
    work2 = (my_work_t *)
      kmalloc(sizeof(my_work_t), GFP_KERNEL);

    if (work2) {
      INIT_WORK( (struct work_struct *)work2,
      my_wq_function );

      work2->x = 2;

      ret = queue_work( my_wq, (struct
      work_struct *)work2 );
    }
  }
  return 0;
}
```

```
void cleanup_module( void )
{
  flush_workqueue( my_wq );

  destroy_workqueue( my_wq );

  return;
}
```

Source: http://public.dhe.ibm.com/software/dw/linux/l-tasklets/l-tasklets-pdf.pdf

Exercise – Work queues

1. Modify your “interrupt.c” module to use a work queue:
   - In the interrupt handler, create a work item and queue it.
   - In the work item function, calculate and print the time difference between ISR and work item execution.
2. Compile and load the module.
3. Watch dmesg output.

Hints:

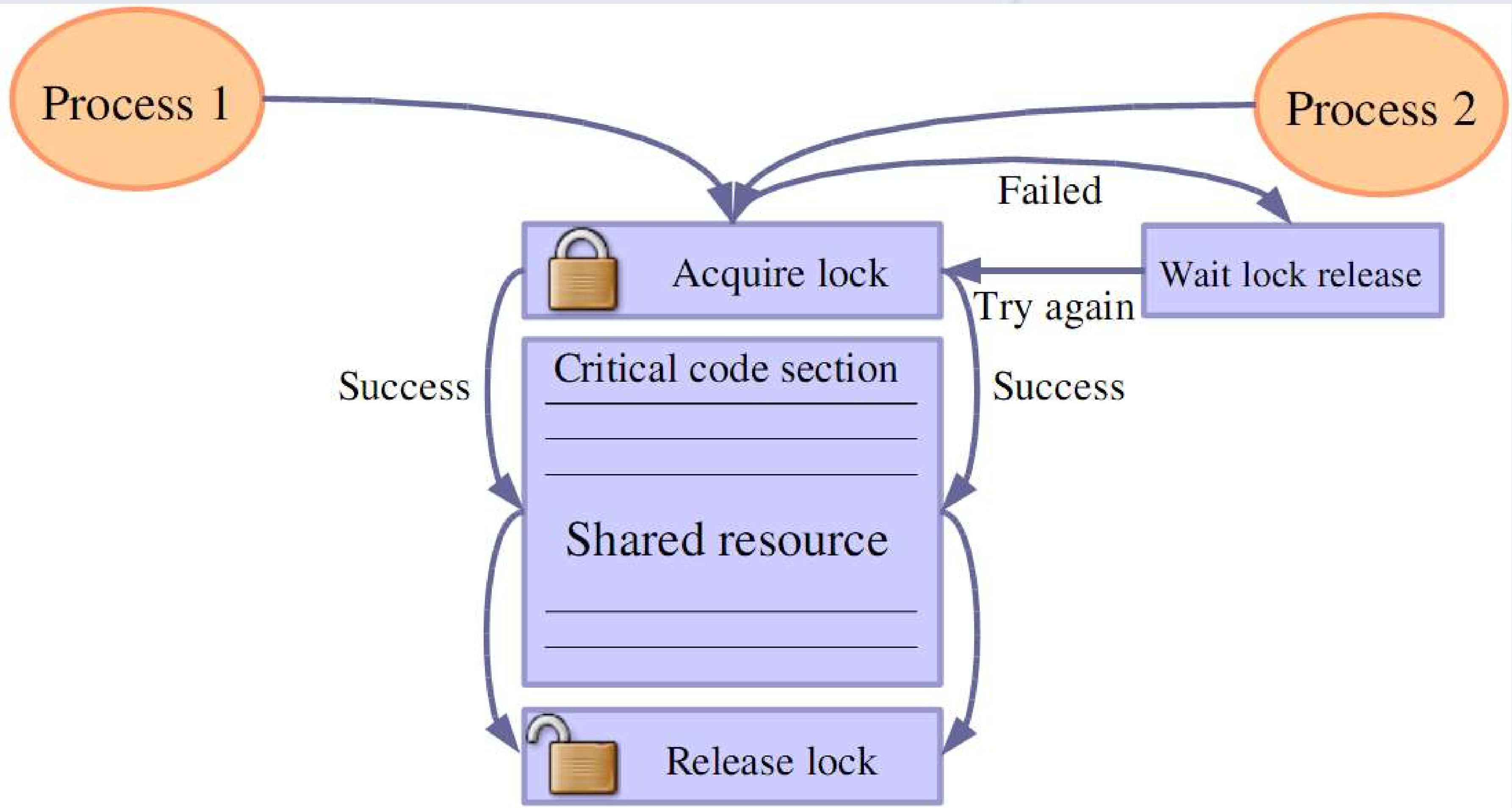
- Use the function **ktime_get_ns()**
- Don’t forget to flush the queue and unhook the interrupt handler in your module_cleanup!

Synchronization

Simultaneous resources access

- Several user-space programs accessing the same device, data or hardware.
- Several kernel processes executing the same code on behalf of parallel user processes.
- The same code can be running on another processor.
- Kernel code can be interrupted at any time, and the same data may be access by another process before execution continues.
- Kernel code may be preempted.

Locking principle



Source: http://free-electrons.com

Linux Locking primitives

- Linux offers several locking primitives:
  - Semaphores
  - Mutexes
  - Spinlocks

- Atomic variables are not locks, but help avoid synchronization issues.

Semaphores

- Semaphores are sleeping locks: they cause a task to sleep on contention, instead of spin.
- It is safe to block while holding a semaphore.
- Semaphores are used when the lock-held time may be long.
- Can be used to synchronize user contexts, whereas spinlocks cannot.
- The semaphore structure contain:
  - Pointer to a **wait queue** - a list of processes blocking on the semaphore.
  - **Usage count** - the number of concurrently allowed holders.

Semaphores (2)

- If count is negative, the semaphore is unavailable and the absolute value of the usage count is the number of processes blocked on the wait queue.
- Semaphores are manipulated via two methods:
  - down - attempts to acquire the semaphore and blocks if it fails.
  - up - releases the semaphore, waking up any tasks blocked along the way.

Initialization semaphores

- Requires #include <linux/semaphore.h>
- Initialization:

```
void sema_init (struct semaphore *sem, int val);
```

- Example:

```
struct semaphore mr_sem;
sema_init (&mr_sem, 2);     // usage count is 2
```

Acquiring semaphores

- Attempt to acquire a semaphore:

  int **down_interruptible** (struct semaphore*);

  - Returns 0 on success (semaphore acquired).
  - If not, process goes to TASK_INTERRUPTIBLE.
  - If sleeping is interrupted by a signal, returns -EINTR.

  int **down** (struct semaphore*);

  - If semaphore is not available, process goes to TASK_UNINTERRUPTIBLE.

- Trying to acquire semaphore (without sleeping):

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

  - Returns 0 if acquired successfully or 1 if it can’t be acquired.

Releasing semaphores

- Releasing a semaphore:

  void **up** (struct semaphore*);

- Example:

```
struct semaphore mr_sem;
sema_init(&mr_sem, 1);      /* usage count is 1 */
if (down_interruptible(&mr_sem))
      /* semaphore not acquired; received a signal ... */
/* critical region (semaphore acquired) ... */
up(&mr_sem);
```

Exercise – Semaphores

1. Modify your “acme.c” module as following:
   - In the “read” handler, block on a semaphore.
   - In the “write” handler, release the semaphore.
2. Compile and load the module.
2. Create a device node for it (mknod).
3. Read from the device node. What happens?
4. Write to the device node. What happens?
5. Try reading from several processes simultaneously. What happens?

6. Remove module from kernel.

Mutexes

- Can be viewed as non-counting semaphores.
- Requires `#include <linux/mutex.h>`
- Initializing a mutex statically:

```
DEFINE_MUTEX(name);
```

- Initializing a mutex dynamically:

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

Mutex API

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

- Tries to lock the mutex, sleeps otherwise.
- Process goes to TASK_UNINTERRUPTIBLE state.

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

- If interrupted, returns a non-zero value and doesn't hold the lock.
- Test the return value!!!

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);

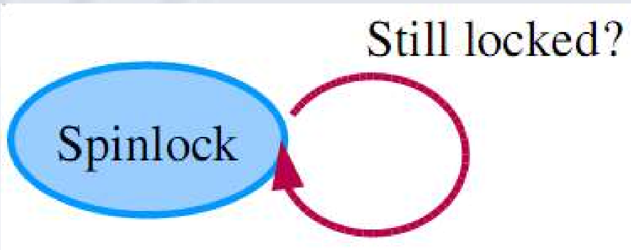
- Just tells whether the mutex is locked or not.

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

- Releases the lock.

Spinlocks

- Originally intended for multiprocessor systems.
- Used for code that can’t sleep (interrupt handlers), or doesn't want to sleep (critical sections).
- Be very careful not to call functions which can sleep!
- Spinlocks are not interruptible, don't sleep and keep spinning in a loop until the lock is available.
- Spinlocks disable kernel preemption and/or interrupts on the local CPU.
- Never re-acquire a spinlock you already hold, or you will deadlock!



Source: http://free-electrons.com

Spinlock initialization

- Requires #include <linux/spinlock.h>
- Initializing spinlocks statically:

  spinlock_t my_spinlock = SPIN_LOCK_UNLOCKED;

  or

  DEFINE_SPINLOCK (my_spinlock);
- Dynamically:

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

Source: http://free-electrons.com

Using spinlocks

void **spin_lock** (spinlock_t *lock);

void **spin_unlock** (spinlock_t *lock);

- Used for protecting critical sections in process context.
- Does not disable interrupts.

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

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

- Disables / restores IRQs on the local CPU.
- Used to prevent preemption by interrupts (when code can be accessed in both process and interrupt context).

Spinlock usage Example

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;
unsigned long flags;
spin_lock_irqsave (&mr_lock, flags);
/* critical section ... */
spin_unlock_irqrestore (&mr_lock, flags);
```

- While holding a spinlock, never call any function that:
  - touches user memory
  - kmalloc() with the GFP_KERNEL flag
  - uses any semaphore functions
  - uses any of the scheduler functions

Atomic variables

- Useful when the shared resource is an integer value.
- Even an instruction like n++ is not guaranteed to be atomic.
- Requires #include <asm/atomic.h>
- Type atomic_t contains a signed integer.
- Set / Read operations:
  - atomic_set (atomic_t * int);
  - int atomic_read (atomic_t *);
- Operations without return value:
  - void atomic_inc (atomic_t *);
  - void atomic_dec (atomic_t *);
  - void atomic_add (int, atomic_t *);
  - void atomic_sub (int, atomic_t *);

Atomic variables (2)

- Similar functions testing the result (returns true if the result is zero) :
  - int atomic_inc_and_test (atomic_t *);
  - int atomic_dec_and_test (atomic_t *);
  - int atomic_sub_and_test (atomic_t *);

- Functions returning the new value:
  - int atomic_inc_and_return (atomic_t *);
  - int atomic_dec_and_return (atomic_t *);
  - int atomic_add_and_return (atomic_t *);
  - int atomic_sub_and_return (atomic_t *);

Atomic bit operations

- 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 (returns the previous value):
  - int test_and_set_bit (int nr, unsigned long *addr);
  - int test_and_clear_bit (int nr, unsigned long *addr);
  - int test_and_change_bit (int nr, unsigned long *addr);

Time in the kernel

HZ and jiffies

- Timer interrupts occur every 1 / HZ of second (= 1 **jiffy**)
- HZ is now configurable (in 'Processor type and features'):
  - 100, 250 (i386 default), 300 or 1000 (other architectures)
  - See kernel/Kconfig.hz.
- Compromise between system responsiveness and global throughput.
- The global variable jiffies represents the number of timer ticks since the machine started. It increments with each timer interrupt.
- Read jiffies with get_jiffies_64 function.
- Convert to milliseconds with jiffies_to_msecs or to microseconds with jiffies_to_usecs.
- Requires #include <linux/jiffies.h>

Using jiffies

```
int time_after (unsigned long a, unsigned long b);
int time_before (unsigned long a, unsigned long b);
int time_after_eq (unsigned long a, unsigned long b);
int time_before_eq (unsigned long a, unsigned long b);
```

- These Boolean expressions compare jiffies in a safe way, without problems in case of counter overflow and without the need to access jiffies_64.

Converting jiffies

- To exchange time representations with user space programs, jiffies are commonly converted to:

  struct timespec (seconds and nanoseconds)

  struct timeval (seconds and microseconds)

- Conversion functions defined in <linux/time.h>:

  unsigned long **timespec_to_jiffies** (struct timespec *value);

  void **jiffies_to_timespec** (unsigned long jiffies, struct timespec *value);

  unsigned long **timeval_to_jiffies** (struct timeval *value);

  void **jiffies_to_timeval** (unsigned long jiffies, struct timeval *value);

Timers

- The kernel provides asynchronous timers, that run after their defined delay.
- Timers run in atomic context (e.g. can't sleep).
- Timers require #include <linux/timer.h>
- Timer structure:

```
struct timer_list {
    /* ... */
    unsigned long expires;              // expressed in jiffies
    void (*function)(unsigned long);    // timer callback function
    unsigned long data;                 // callback argument
};
```

Timers API

- Timer structure initialization - static:

  struct timer_list **TIMER_INITIALIZER** (_function, _expires, _data);

- Dynamic initialization:

  void **init_timer** (struct timer_list *timer);

  void **setup_timer** (struct timer_list * timer, void (*function)(unsigned long), unsigned long data)

- Adding a new timer:

  void **add_timer** (struct timer_list * timer);

- Removing a timer:

  int **del_timer** (struct timer_list * timer);

Timers API (2)

int **mod_timer** (struct timer_list *, unsigned long expires);

- Updates the expiration time of a timer.

int **del_timer_sync** (struct timer_list *);

- Guarantees that when it returns, the timer function is not running on any CPU.
- Can sleep if it is called from a non-atomic context but busy waits in other situations.
- Be very careful about calling it while holding locks; if the timer function attempts to obtain the same lock, the system can deadlock.

int **timer_pending** (const struct timer_list *);

- Returns true/false indicating whether the timer is scheduled.

Timers example

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/timer.h>

static struct timer_list my_timer;

void my_timer_callback( unsigned long data )
{
  printk( "my_timer_callback called (%ld).\n",
      jiffies );
}

int init_module( void ) {
  int ret;

  printk("Timer module installing\n");
  setup_timer ( &my_timer, my_timer_callback,
              0 );
```

```
  printk( "Starting timer to fire in 200ms
      (%ld)\n", jiffies );

  ret = mod_timer( &my_timer, jiffies +
      msecs_to_jiffies(200) );

  if (ret) printk("Error in mod_timer\n");

  return 0;
}

void cleanup_module( void )
{
  int ret;
  ret = del_timer( &my_timer );
  if (ret)
      printk("The timer is still in use...\n");
  printk("Timer module uninstalling\n");
  return;
}
```

Source: http://www.ibm.com/developerworks/linux/library/l-timers-list

High-resolution timers

- hrtimers were introduced in mainstream kernel 2.6.21 (2007).
- Depend on **CONFIG_HIGH_RES_TIMERS**.
- hrtimers operate at the granularity of nanoseconds (architecture-dependent).
- hrtimers are available to kernel code, and to user-space applications (via nanosleep, itimers, and POSIX-timers).
- Time is now represented in a data type called ktime_t, using nanosecond base.
  - plain nanosecond value on 64 bit CPU
  - (seconds, nanoseconds) pair on 32 bit
- struct hrtimer defines the timer (callback function, expiration time, etc.)

High-resolution timers (2)

hrtimers rely on several defined clocks (see include/linux/timer.h):

- CLOCK_REALTIME
  - System-wide clock measuring the time in seconds and nanoseconds since 1/1/1970, 00:00.
  - Can be modified.
  - Accuracy: 1/HZ.
- CLOCK_MONOTONIC
  - System-wide clock measuring the time in seconds and nanoseconds since system boot.
  - Cannot be modified, so can be used for accurate time measurement.
  - Accuracy: 1/HZ

High-resolution timers (3)

```
#include <linux/hrtimer.h>
#include <linux/ktime.h>
…
void hrtimer_init (struct hrtimer *time, clockid_t which_clock, enum
    hrtimer_mode mode );

int hrtimer_start (struct hrtimer *timer, ktime_t time, const enum
    hrtimer_mode mode);
```

- **mode** may be :
  - HRTIMER_MODE_ABS = Time value is absolute
  - HRTIMER_MODE_REL = Time value is relative to now

hrtimers example (1)

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/hrtimer.h>
#include <linux/ktime.h>

MODULE_LICENSE("GPL");

#define MS_TO_NS(x) (x * 1E6L)

static struct hrtimer hr_timer;

enum hrtimer_restart my_hrtimer_callback (struct
      hrtimer *timer )
{
      printk( "my_hrtimer_callback called
            (%ld).\n", jiffies );
      ktime_t now = ktime_get();
      hrtimer_forward (&hr_timer,  now,
            MS_TO_NS(delay_in_ms));
      return HRTIMER_RESTART;
}
```

```
int init_module( void )
{
  ktime_t ktime;
  unsigned long delay_in_ms = 200L;

  ktime = ktime_set( 0, MS_TO_NS(delay_in_ms)
      );

  hrtimer_init(&hr_timer, CLOCK_MONOTONIC,
      HRTIMER_MODE_REL );

  hr_timer.function = &my_hrtimer_callback;

  hrtimer_start( &hr_timer, ktime,
      HRTIMER_MODE_REL );

  return 0;
}
```

Source: http://www.ibm.com/developerworks/linux/library/l-timers-list

Cancelling hrtimers

- Once an hrtimer has started, it can be cancelled using:

  int **hrtimer_cancel** (struct hrtimer *timer);

  int **hrtimer_try_to_cancel** (struct hrtimer *timer);

- hrtimer_cancel attempts to cancel the timer, but if it has already fired, it will wait for the callback function to finish.
- hrtimer_try_to_cancel attempts to cancel the timer, but will return failure if the timer has fired.

hrtimers example (2)

```
void cleanup_module( void )
{
  int ret;

  ret = hrtimer_cancel( &hr_timer );
  if (ret)
      printk("The timer was still in use...\n");

  printk("HR Timer module uninstalling\n");

  return;
}
```

Source: http://www.ibm.com/developerworks/linux/library/l-timers-list

Exercise – Timers

1. Modify your “interrupt.c” module as following:
   - Collect statistics of interrupts rate (interrupts / second).
   - Calculate minimum, maximum and average rate.
   - Print the statistics in “module_cleanup”.
2. Compile and load the module.
3. Wait a few seconds and unload the module.
4. Watch dmesg output.

Hints:

- Use a timer.
- Don’t forget to cancel the timer and unhook the interrupt handler in your module_cleanup!