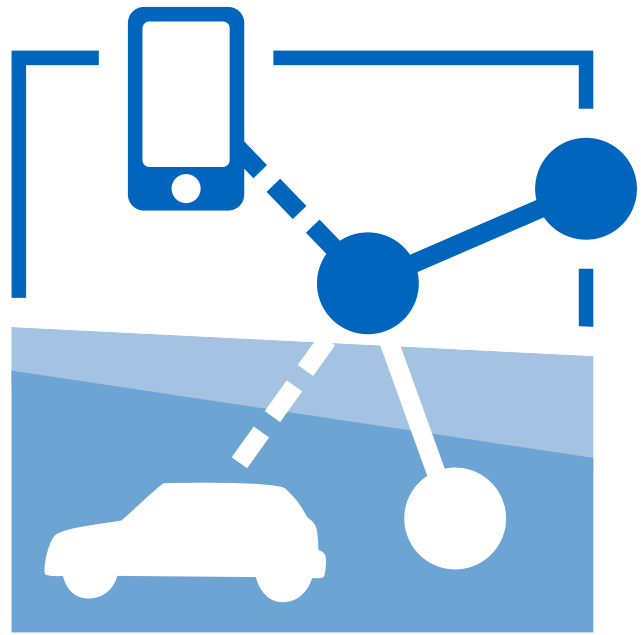
Linux and L4

Daniel Krefft



Latest update: April 17, 2015.
Document updates and sources:
`https://www.moodle.tum.de/course/view.php?id=17970`
Corrections, suggestions, contributions and translations are welcome!

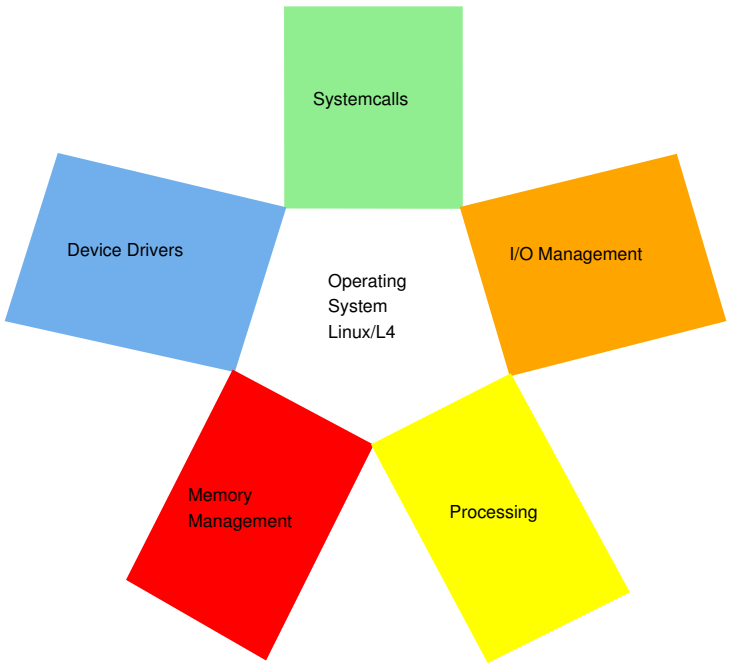
Rights to copy

Hyperlinks in the document

There are many hyperlinks in the document

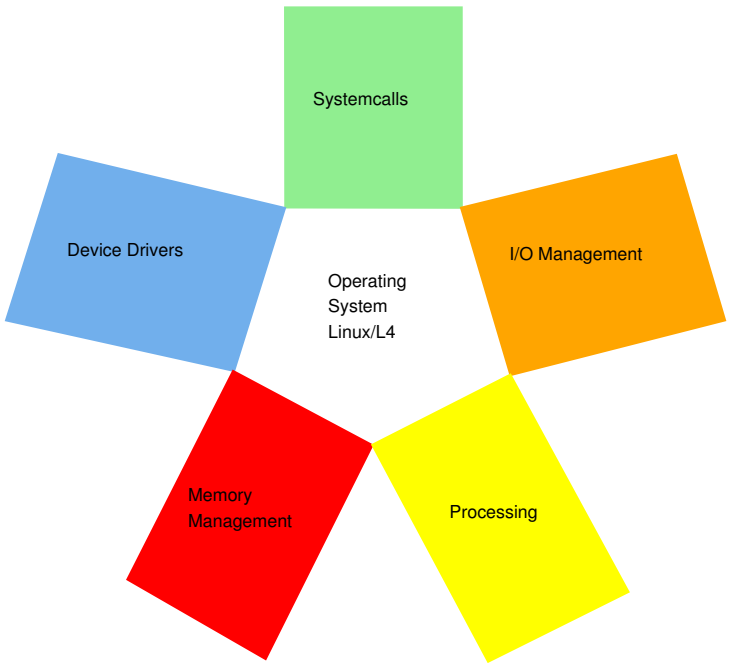
- Regular hyperlinks:
  `http://kernel.org/`
- Kernel documentation links:
  `Documentation/kmemcheck.txt`
- Links to kernel source files and directories:
  `drivers/input`
  `include/linux/fb.h`
- Links to the declarations, definitions and instances of kernel symbols (functions, types, data, structures):
  `platform_get_irq()`
  `GFP_KERNEL`
  `struct file_operations`

Simplified architecture of an operating system



- Interface for systemcalls
- I/O-Management
- Processing
- Memory management
- Device drivers

Simplified architecture of an operating system



- Interface for systemcalls
- I/O-Management
- Processing
- Memory management
- Device drivers (3rd session)

Systemcalls

- The main interface between the kernel and userspace is the set of system calls

I/O-Management

- Offers interface for proper hardware integration
- Provides common API for hardware access (device files)
  - Character Devices
  - Block Devices
  - Subsystems for integration of modern multimedia devices
- Realizes concept of files and folder over filesystem

Processing

- Distribute processing time between tasks and threads

Process

consists of code and data (const/stack)

Threads

share code segment and data segment **but** have own stack

Tasks

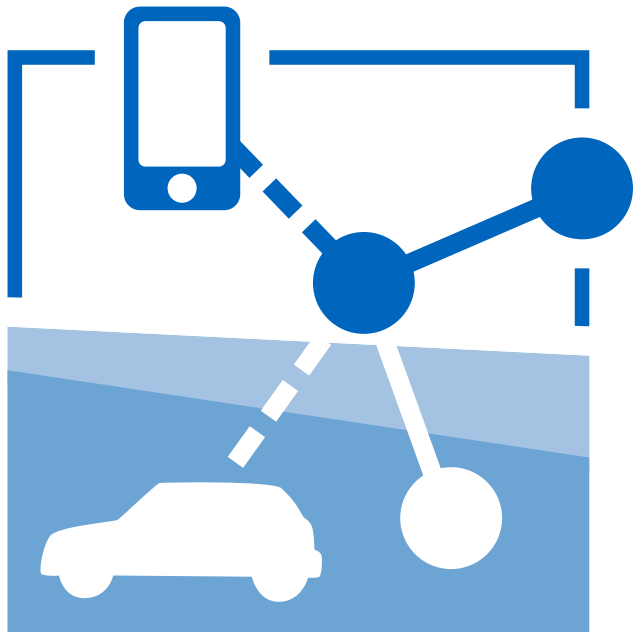
share code segment but have separate data segment

Memory management

- Separation of logical memory addresses from physical memory addresses
- Developer defines memory segments which can be checked for read/write access
- Enables distinction between *Kernelspace* and *Userspace*

Linux Kernel Introduction

Daniel Krefft



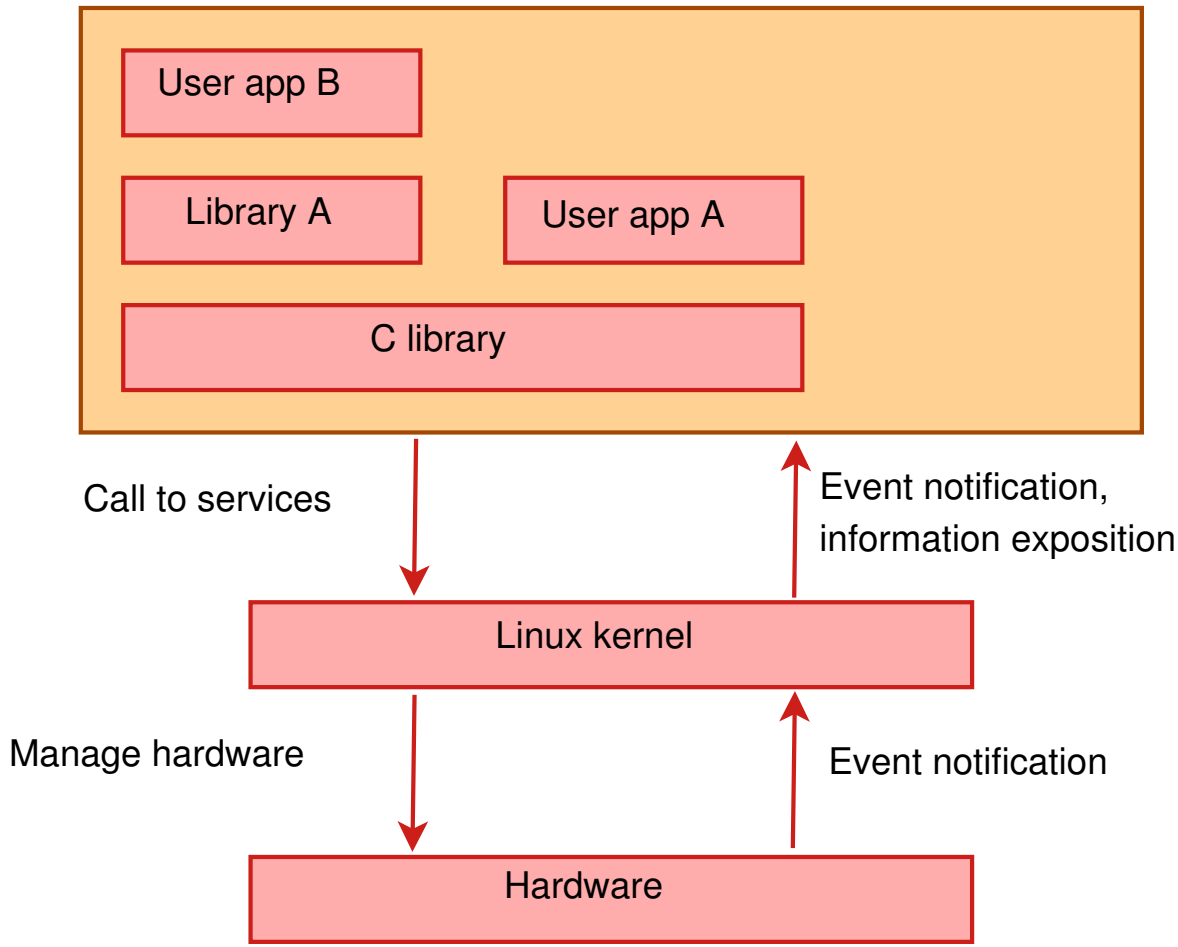
Linux Kernel Introduction

Linux features

History

- The Linux kernel is one component of a system, which also requires libraries and applications to provide features to end users.
- The Linux kernel was created as a hobby in 1991 by a Finnish student, Linus Torvalds.
  - Linux quickly started to be used as the kernel for free software operating systems
- Linus Torvalds has been able to create a large and dynamic developer and user community around Linux.
- Nowadays, hundreds of people contribute to each kernel release, individuals or companies big and small.

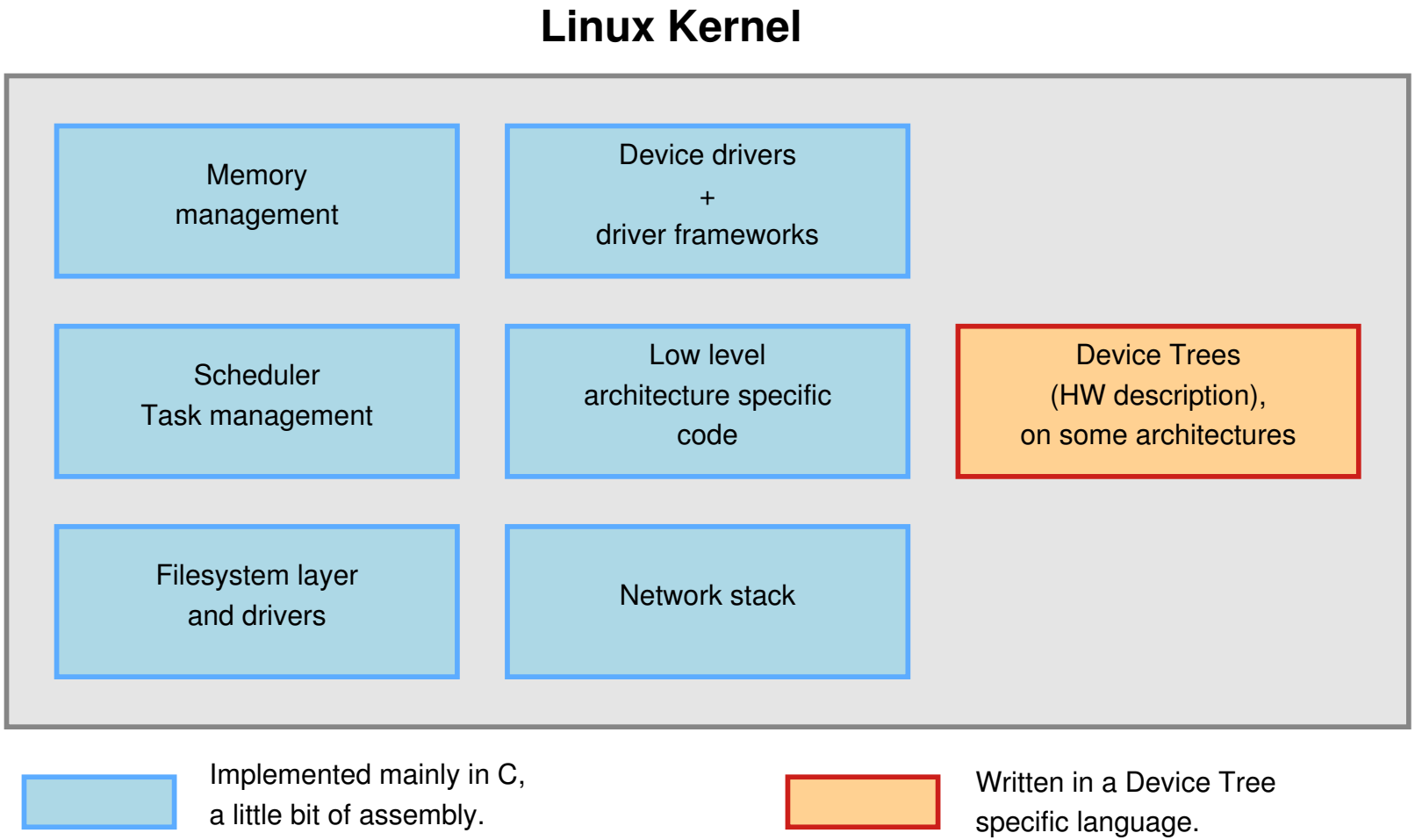
Linux kernel in the system



Linux kernel main roles

- **Manage all the hardware resources**: CPU, memory, I/O.
- Provide a **set of portable, architecture and hardware independent APIs** to allow userspace applications and libraries to use the hardware resources.
- **Handle concurrent accesses and usage** of hardware resources from different applications.
  - Example: a single network interface is used by multiple userspace applications through various network connections. The kernel is responsible to "multiplex" the hardware resource.

Inside the Linux kernel



Linux license

- The whole Linux sources are Free Software released under the GNU General Public License version 2 (GPL v2).
- For the Linux kernel, this basically implies that:
  - When you receive or buy a device with Linux on it, you should receive the Linux sources, with the right to study, modify and redistribute them.
  - When you produce Linux based devices, you must release the sources to the recipient, with the same rights, with no restriction..

Linux kernel key features

- Portability and hardware support. Runs on most architectures.
- Scalability. Can run on super computers as well as on tiny devices (4 MB of RAM is enough).
- Compliance to standards and interoperability.
- Exhaustive networking support.
- Security. It can't hide its flaws. Its code is reviewed by many experts.
- Stability and reliability.
- Modularity. Can include only what a system needs even at run time.
- Easy to program. You can learn from existing code. Many useful resources on the net.

Supported hardware architectures

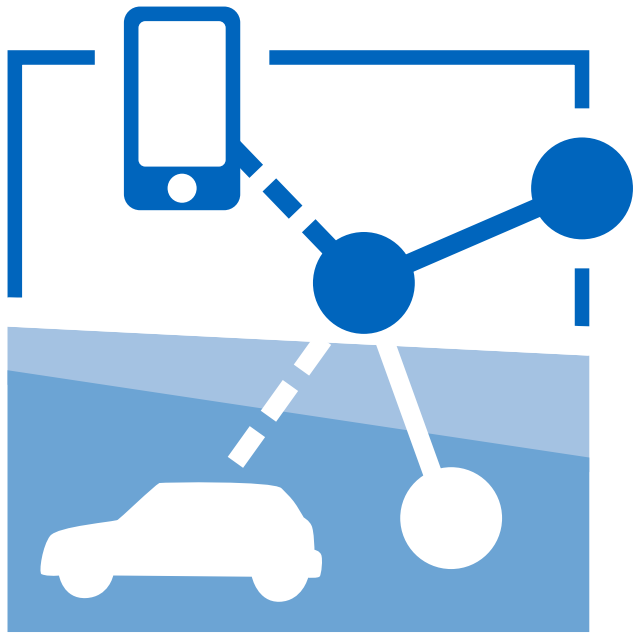
- See the `arch/` directory in the kernel sources
- Minimum: 32 bit processors, with or without MMU, and `gcc` support
- 32 bit architectures (`arch/` subdirectories)
  Examples: `arm`, `avr32`, `blackfin`, `m68k`, `microblaze`, `mips`, `score`, `sparc`, `um`
- 64 bit architectures:
  Examples: `alpha`, `arm64`, `ia64`, `sparc64`, `tile`
- 32/64 bit architectures
  Examples: `powerpc`, `x86`, `sh`
- Find details in kernel sources: `arch/<arch>/Kconfig`, `arch/<arch>/README`, or `Documentation/<arch>/`

Virtual filesystems

- Linux makes system and kernel information available in user-space through virtual filesystems.
- Virtual filesystems allow applications to see directories and files that do not exist on any real storage: they are created on the fly by the kernel
- The two most important virtual filesystems are
  - `proc`, usually mounted on `/proc`:
    Operating system related information (processes, memory management parameters...)
  - `sysfs`, usually mounted on `/sys`:
    Representation of the system as a set of devices and buses. Information about these devices.

System Calls

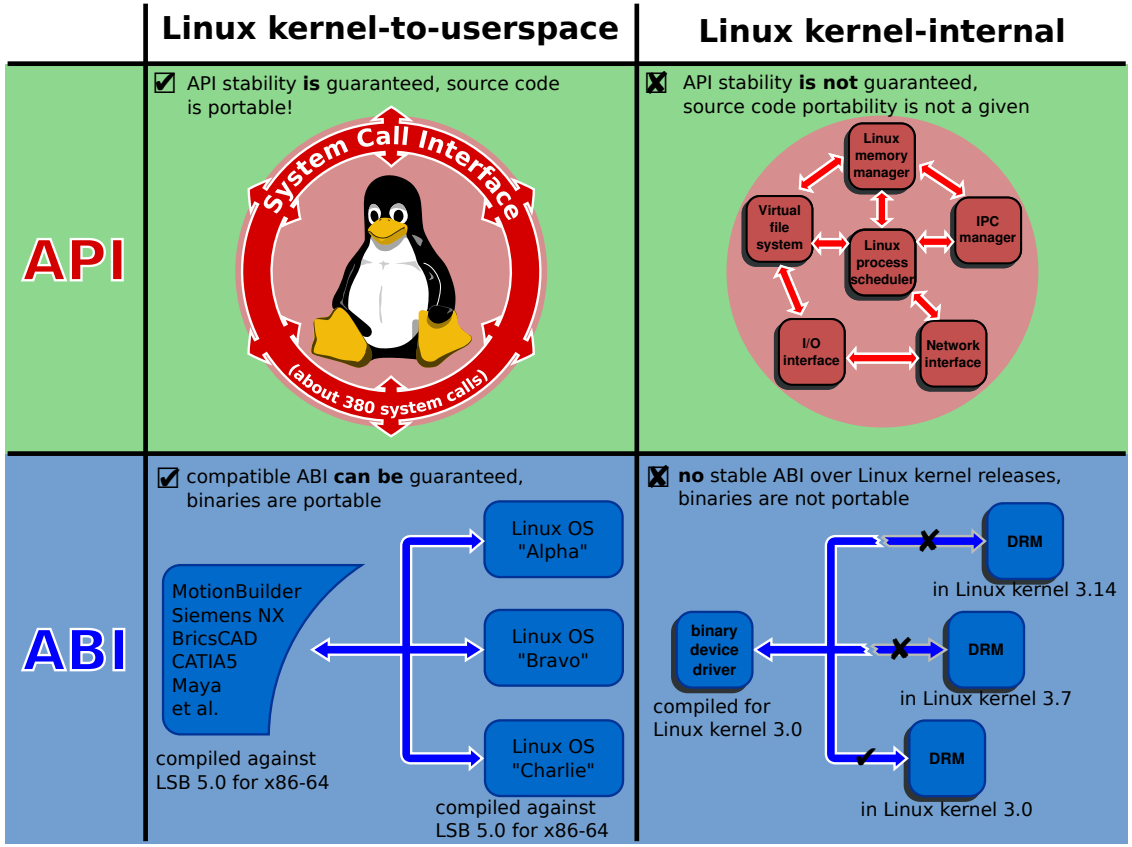
Daniel Krefft



System Calls

- The main interface between the kernel and userspace is the set of system calls
- About 300 system calls that provide the main kernel services
  - File and device operations, networking operations, inter-process communication, process management, memory mapping, timers, threads, synchronization primitives, etc.
- This interface is stable over time: only new system calls can be added by the kernel developers
- This system call interface is wrapped by the C library, and userspace applications usually never make a system call directly but rather use the corresponding C library function

System Calls

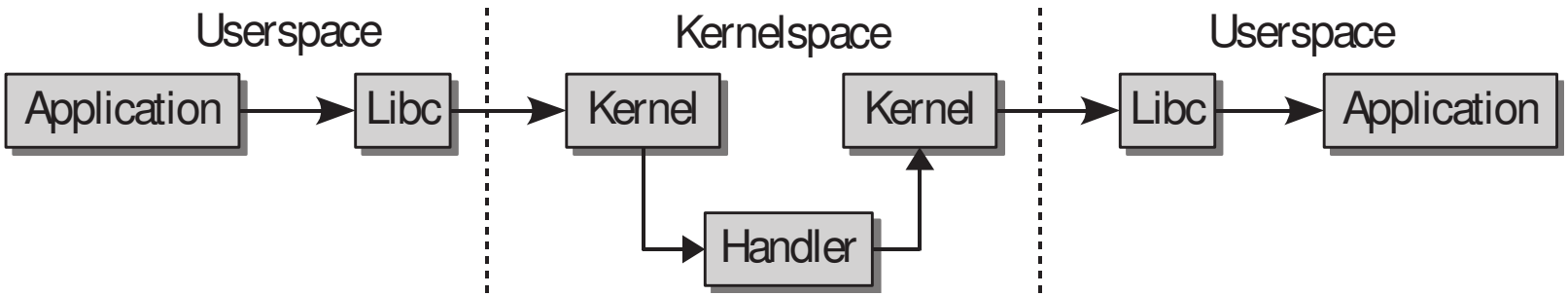


Available System Calls

There are several groups of system calls:

- **Process Management** Processes are at the center of the system, so it's not surprising that a large number of system calls are devoted to process management
- **Time Operations** Time operations are critical, not only to query and set the current system time, but also to give processes the opportunity to perform time-based operations
- **Scheduling** Scheduling-related system calls could be grouped into the process management category because all such calls logically relate to system tasks
- **Filesystem** All system calls relating to the filesystem apply to the routines of the VFS layer
- **Memory Management** Under normal circumstances, user applications rarely or never come into contact with memory management system calls because this area is completely shielded from the standard library

Implementation of System Calls



Example: Systemcalls

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <malloc.h>

int main() {
 int handle, bytes;
 void* ptr;
 handle = open("/tmp/test.txt", O_RDONLY);
 ptr = (void*)malloc(150);
 bytes = read(handle, ptr, 150);
 printf("%s", ptr);
 close(handle);
 return 0;
}
```

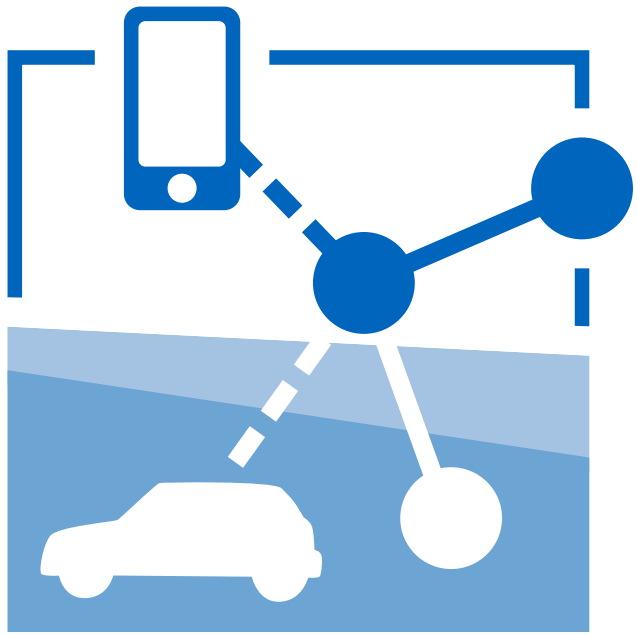
Examine the issued system calls via `strace`

Discussion & Questions

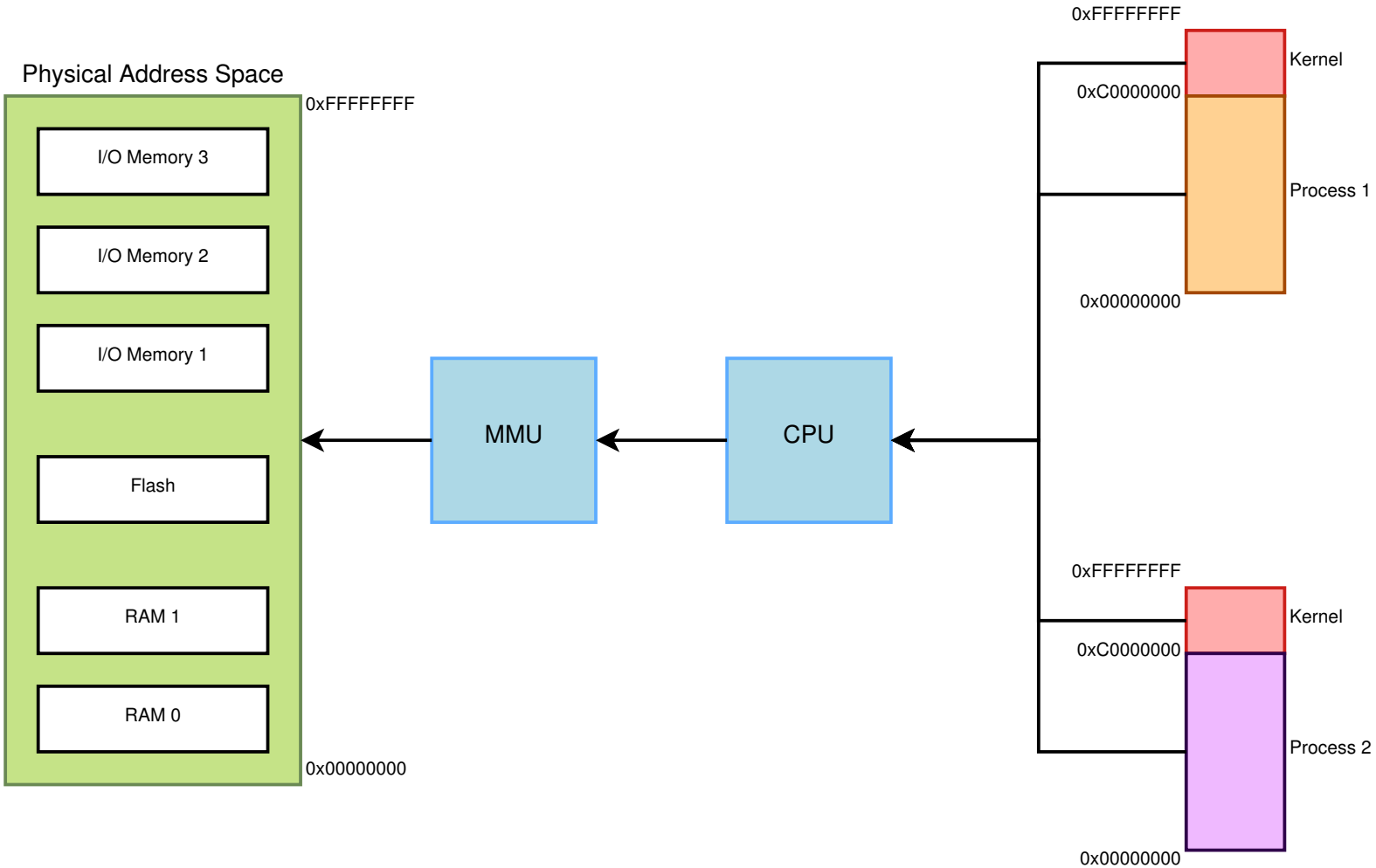
- Which error will arise in the example on slide 24
- Which system call is required for writing keylogger applications?
- Why do I need system calls?

Memory Management

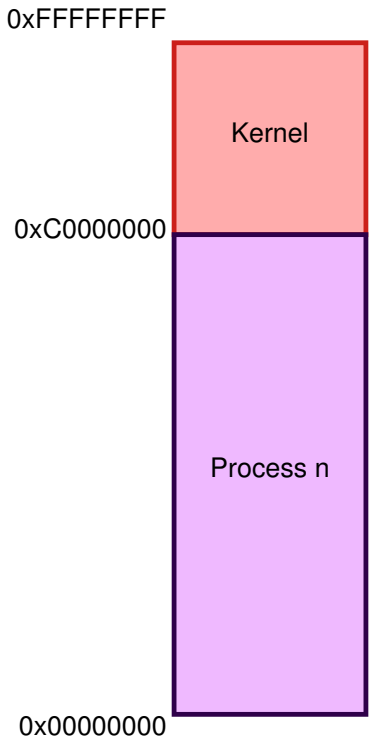
Daniel Krefft



Physical and Virtual Memory

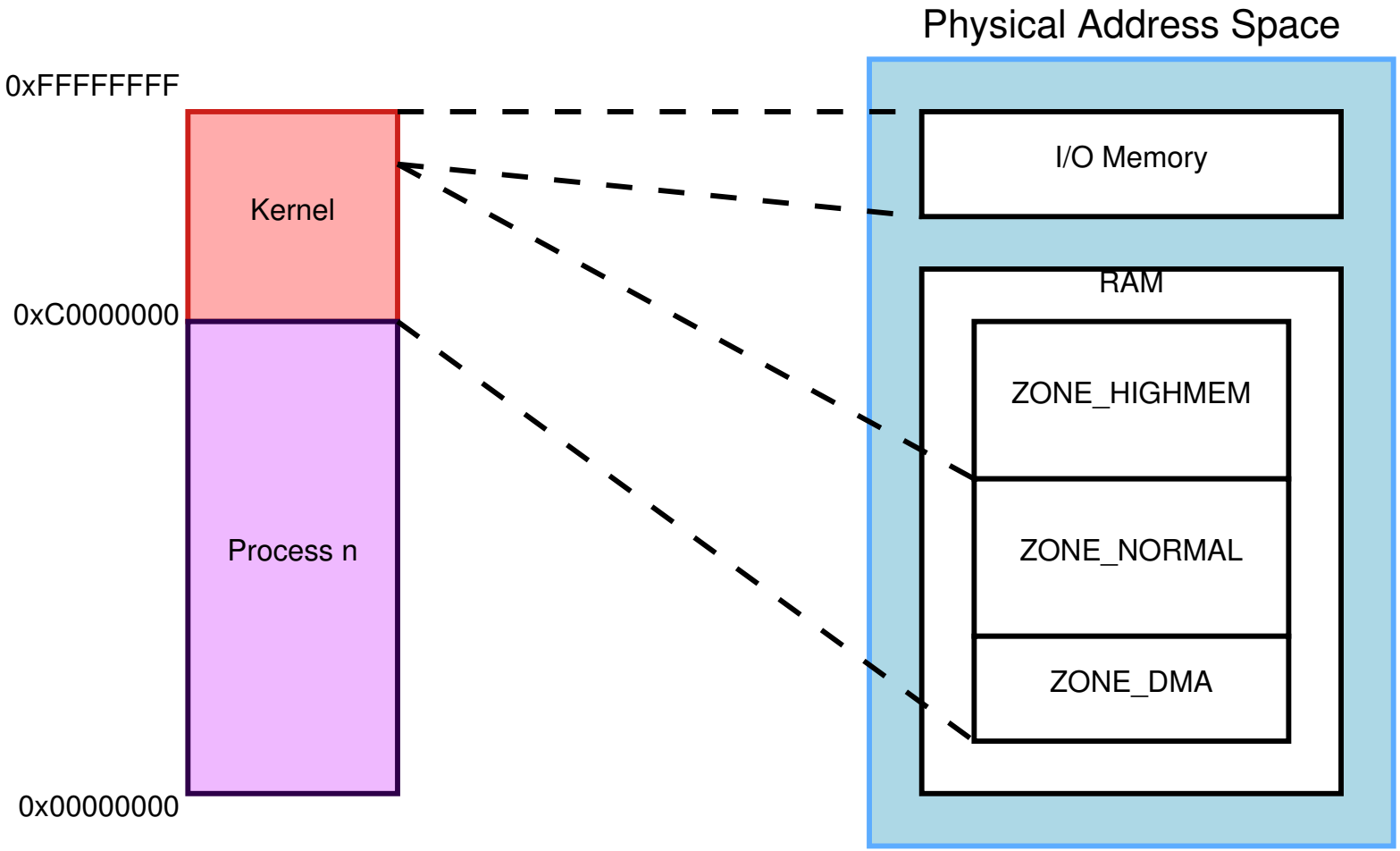


Virtual Memory Organization



- 1GB reserved for kernel-space
  - Contains kernel code and core data structures, identical in all address spaces
  - Most memory can be a direct mapping of physical memory at a fixed offset
- Complete 3GB exclusive mapping available for each user-space process
  - Process code and data (program, stack, ...)
  - Memory-mapped files
  - Not necessarily mapped to physical memory (demand fault paging used for dynamic mapping to physical memory pages)
  - Differs from one address space to another

Physical / virtual memory mapping



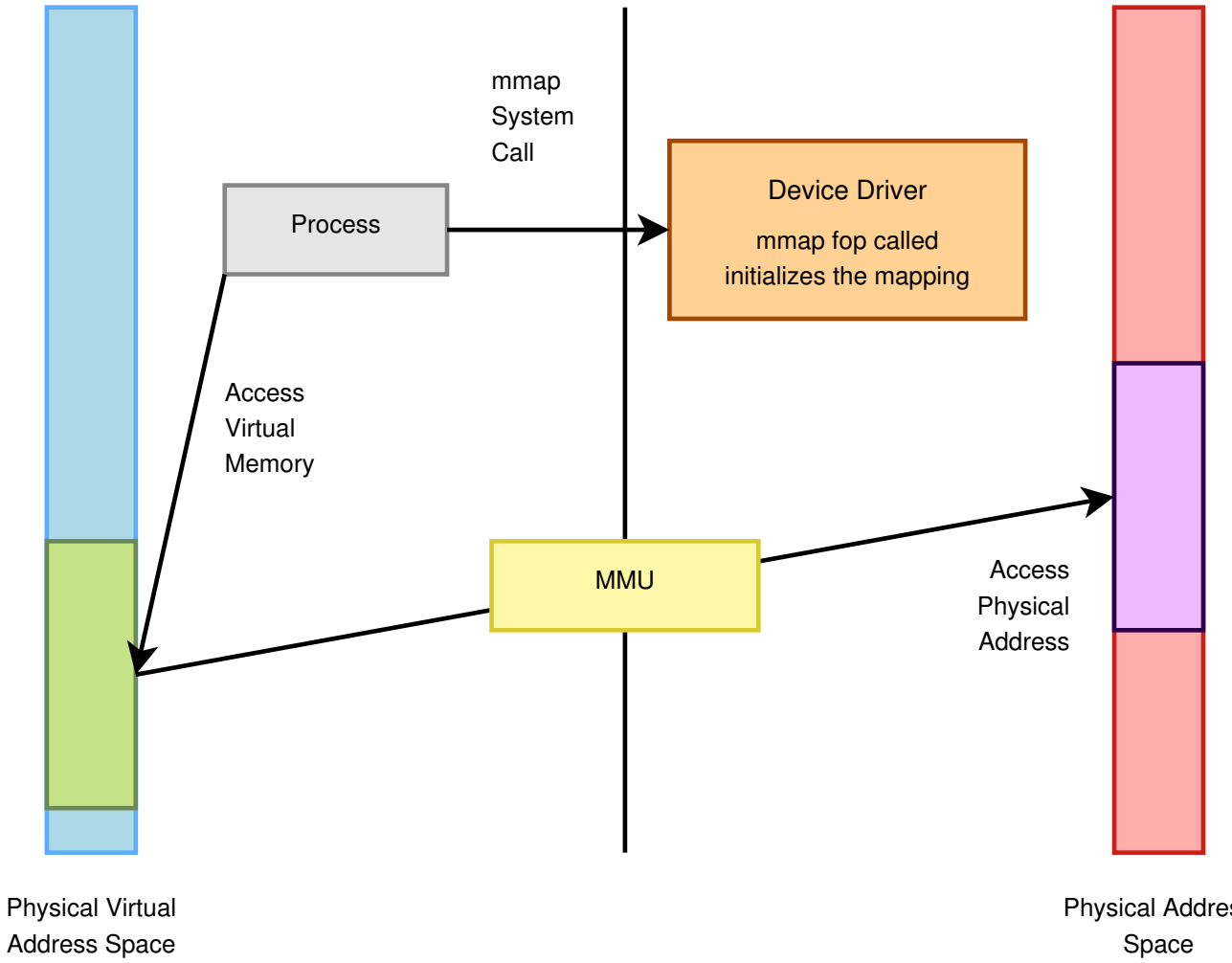
Accessing more physical memory

- Only less than 1GB memory addressable directly through kernel virtual address space
- If more physical memory is present on the platform, part of the memory will not be accessible by kernel space, but can be used by **user-space**
- To allow the kernel to access more physical memory:
  - Change 1GB/3GB memory split (2GB/2GB) (`CONFIG_VMSPLIT_3G`) $\Rightarrow$ reduces total memory available for each process
  - PAE (Physical Address Expansion) may be supported by your architecture (Allows accessing up to 64 GB)
- See `http://lwn.net/Articles/75174/` for useful explanations

Word about Memory Mappings

- `mmap` system call creates a new *memory mapping* in the calling process's virtual address space (user-space).
- Mappings fall into two categories:
  - *file mapping* maps a region of a file into the calling process's virtual memory
  - *anonymous mapping* doesn't have a corresponding file (pages are initialized to 0)
- The memory in one process's mapping may be shared with mappings in other processes (e.g. `fork`)
- Memory mappings serve a variety of purposes:
  - initialization of a process's text segment from the corresponding segment of an executable file
  - allocation of new (zero-filled) memory
  - file I/O (memory-mapped I/O)
  - inter- process communication (via a shared mapping).

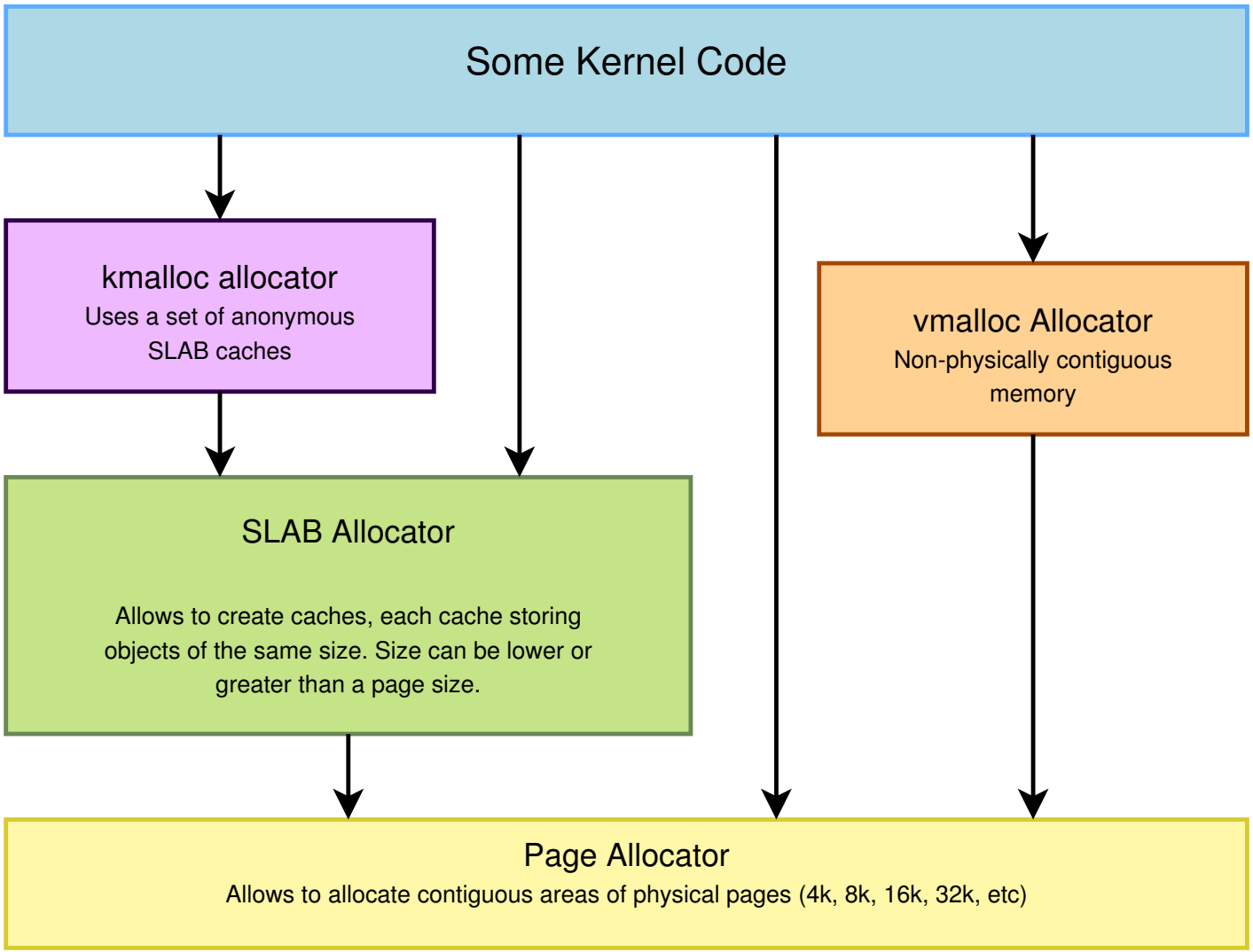
mmap Overview



Back to kernel memory

- Kernel memory allocators (see following slides) allocate *physical pages*, and kernel allocated memory cannot be swapped out, so no fault handling required for kernel memory.
- Most kernel memory allocation functions also return a kernel virtual address to be used within the kernel space.
- Kernel memory low-level allocator manages pages. This is the finest granularity (usually 4 KB, architecture dependent).
- However, the kernel memory management handles smaller memory allocations through its allocator (see *SLAB allocators* – used by `kmalloc()`).

Allocators in the Kernel



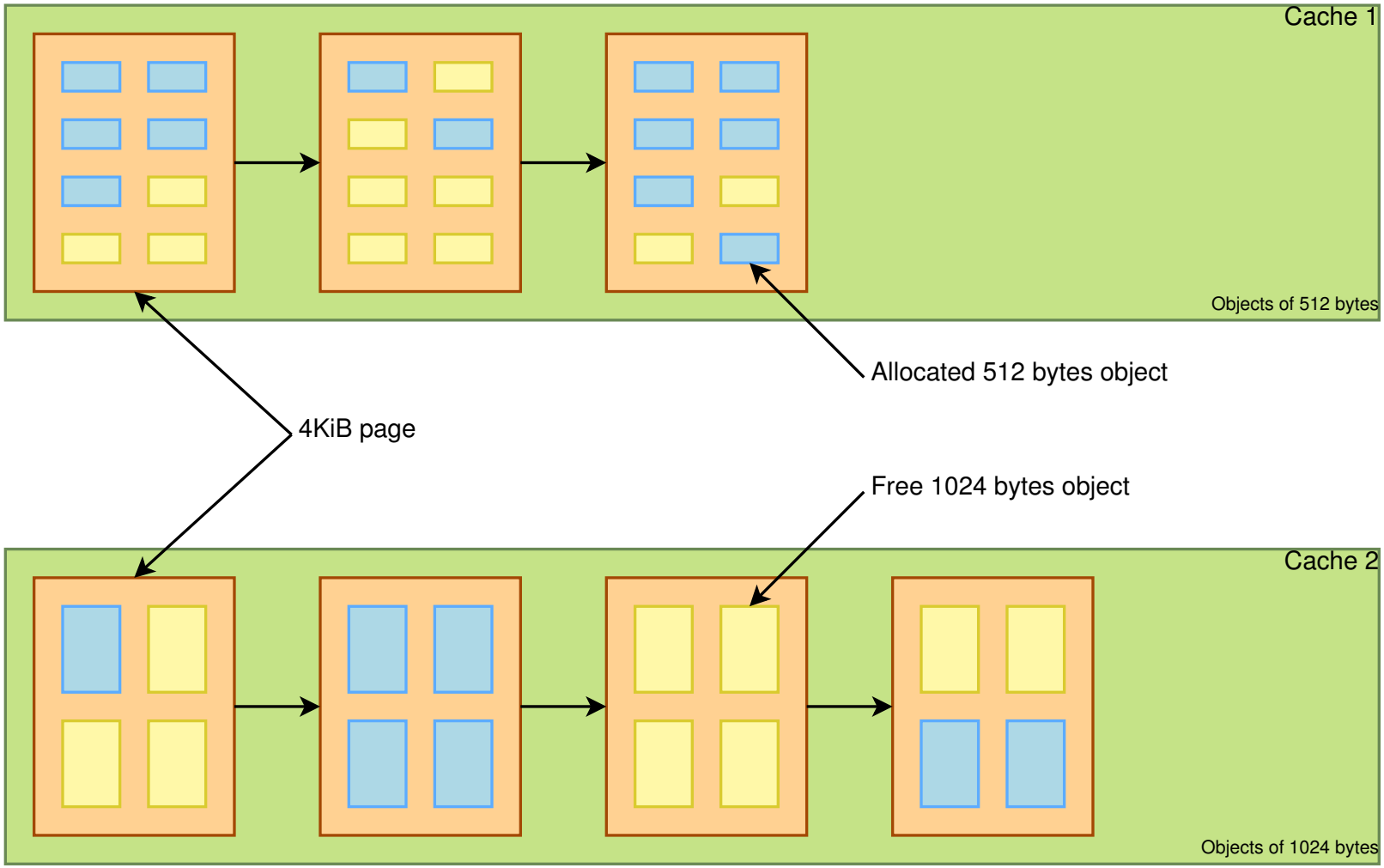
Page Allocator

- Appropriate for medium-size allocations
- A page is usually 4K, but can be made greater in some architectures (sh, mips: 4, 8, 16 or 64 KB, but not configurable in x86 or arm).
- Buddy allocator strategy, so only allocations of power of two number of pages are possible: 1 page, 2 pages, 4 pages, 8 pages, 16 pages, etc.
- Typical maximum size is 8192 KB, but it might depend on the kernel configuration.
- The allocated area is virtually contiguous (of course), but also physically contiguous. It is allocated in the identity-mapped part of the kernel memory space.
    - This means that large areas may not be available or hard to retrieve due to physical memory fragmentation.

SLAB Allocator 1/2

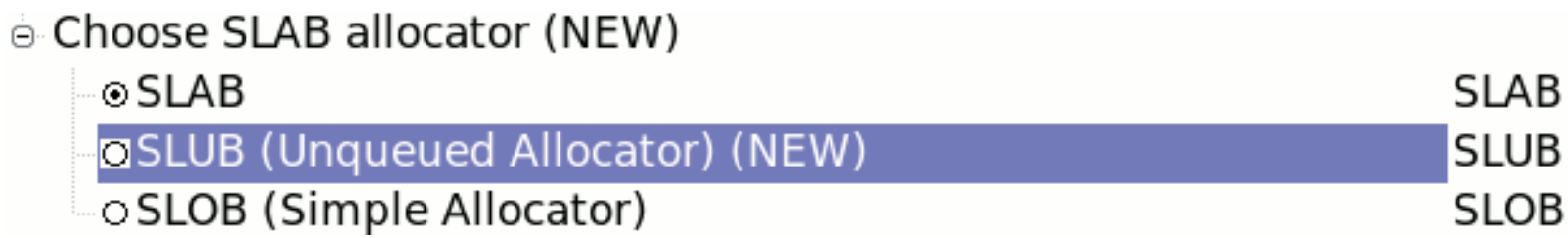
- The SLAB allocator allows to create caches, which contains a set of objects of the same size
- The object size can be smaller or greater than the page size
- The SLAB allocator takes care of growing or reducing the size of the cache as needed, depending on the number of allocated objects. It uses the page allocator to allocate and free pages.
- SLAB caches are used for data structures that are present in many many instances in the kernel: directory entries (inodes), file objects, network packet descriptors, process descriptors, etc. (See `/proc/slabinfo` via slabtop)
- They are rarely used for individual drivers.
- See `include/linux/slab.h` for the API

SLAB Allocator 2/2



Different SLAB Allocators

- There are three different, but API compatible, implementations of a SLAB allocator in the Linux kernel. A particular implementation is chosen at configuration time.
  - SLAB: legacy, well proven allocator.
    Still the default in most ARM `defconfig` files.
  - SLOB: much simpler. More space efficient but doesn't scale well. Saves a few hundreds of KB in small systems (depends on `CONFIG_EXPERT`)
    Linux 3.13 on ARM: used in 5 `defconfig` files
  - SLUB: more recent and simpler than SLAB, scaling much better (in particular for huge systems) and creating less fragmentation.
    Linux 3.13 on ARM: used in 0 `defconfig` files



kmalloc Allocator

- The kmalloc allocator is the general purpose memory allocator in the Linux kernel
- For small sizes, it relies on generic SLAB caches, named `kmalloc-XXX` in `/proc/slabinfo`
- For larger sizes, it relies on the page allocator
- The allocated area is guaranteed to be **physically contiguous**
- The allocated area size is rounded up to the size of the smallest SLAB cache in which it can fit (while using the SLAB allocator directly allows to have more flexibility)
- It uses the same flags as the page allocator (`GFP_KERNEL`, `GFP_ATOMIC`, `GFP_DMA`, etc.) with the same semantics.
- Should be used as the primary allocator unless there is a strong reason to use another one.

kmalloc API 1/2

- `#include <linux/slab.h>`
- `void *kmalloc(size_t size, int flags);`
  - Allocate `size` bytes, and return a pointer to the area (virtual address)
  - `size`: number of bytes to allocate
  - `flags`: same flags as the page allocator
- `void kfree(const void *objp);`
  - Free an allocated area
- Example: (`drivers/infiniband/core/cache.c`)

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

kmalloc API 2/2

- `void *kzalloc(size_t size, gfp_t flags);`
  - Allocates a zero-initialized buffer
- `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 *p, size_t new_size, gfp_t flags);`
  - Changes the size of the buffer pointed by `p` to `new_size`, by reallocating a new buffer and copying the data, unless `new_size` fits within the alignment of the existing buffer.

vmalloc Allocator

- The `vmalloc()` allocator can be used to obtain virtually contiguous memory zones, but not physically contiguous. The requested memory size is rounded up to the next page.
- The allocated area is in the kernel space part of the address space, but outside of the identically-mapped area
- Allocations of fairly large areas is possible (almost as big as total available memory, see `http://j.mp/YIGq6W` again), since physical memory fragmentation is not an issue, but areas cannot be used for DMA, as DMA usually requires physically contiguous buffers.
- API in `include/linux/vmalloc.h`
  - `void *vmalloc(unsigned long size);`
    - Returns a virtual address
  - `void vfree(void *addr);`

Kernel memory debugging

- `Kmemcheck`
  - Dynamic checker for access to uninitialized memory.
  - Only available on `x86` so far (Linux 3.10-rc5 status), but will help to improve architecture independent code anyway.
  - See `Documentation/kmemcheck.txt` for details.
- `Kmemleak`
  - Dynamic checker for memory leaks
  - This feature is available for all architectures.
  - See `Documentation/kmemleak.txt` for details.

Both have a significant overhead. Only use them in development!

Example: Kmemleak

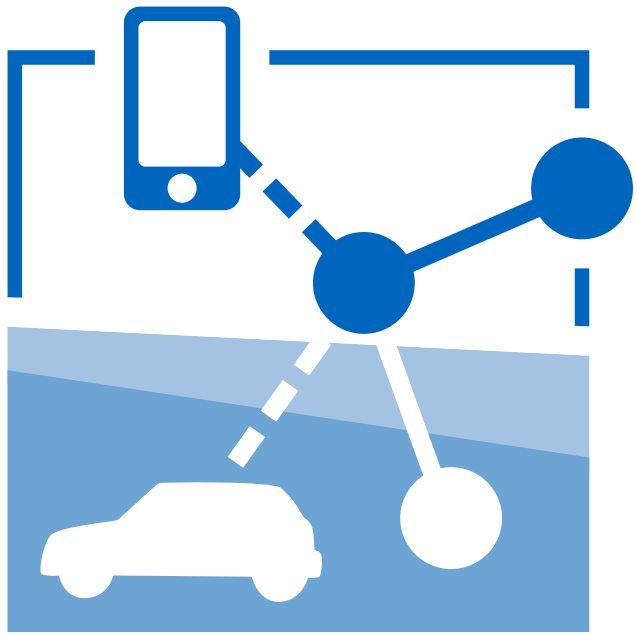
What the documentation says:
`Kmemleak` provides a way of detecting possible kernel memory leaks in a way similar to a tracing garbage collector, with the difference that the orphan objects are not freed but only reported via `/sys/kernel/debug/kmemleak`. A similar method is used by the **Valgrind tool** (memcheck –leak-check) to detect the memory leaks in user-space applications.
Kmemleak is supported on x86, arm, powerpc, sparc, sh, microblaze, ppc, mips, s390, metag and tile.

Discussion/Questions

- Why do I need anonymous memory mapping?
- Why do I need allocator objects?
- Why do I need PAE?
- What is MMU?
- Are systems without MMU available?
- What does SLAB stand for?

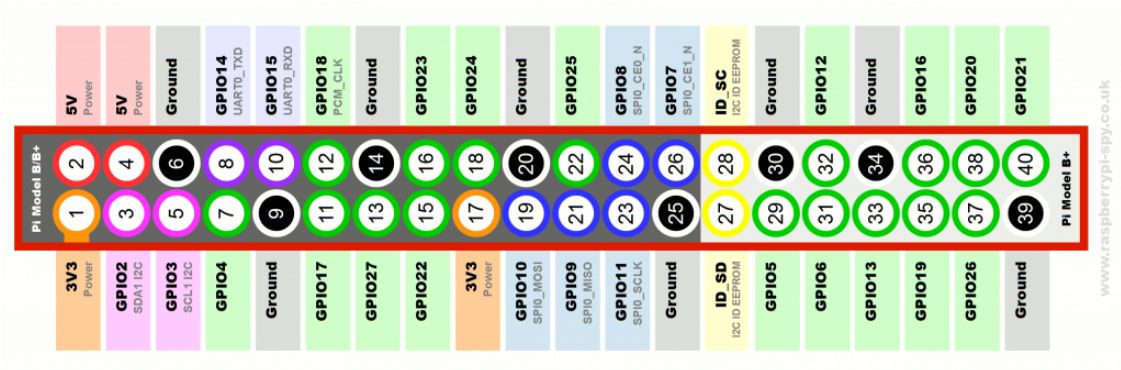
I/O Memory and Ports

Daniel Krefft



I/O Memory and Ports

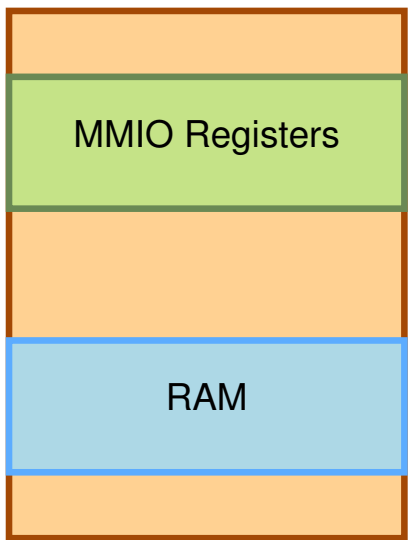
- Memory-mapped I/O (MMIO) and port-mapped I/O (PIO) are two complementary methods of performing input/output between the CPU and peripheral devices (e.g. UART, I^2C)
- General-Purpose Input (GPIO) pin's state may be exposed to the software developer through MMIO/PIO
- If you're using GPIO via sysfs you don't need to know little details like what gpio pin is mapped to what memory address.



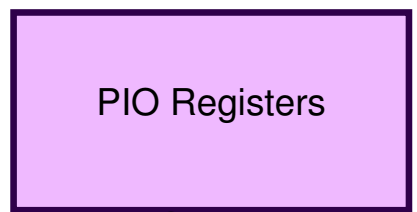
Port I/O vs. Memory-Mapped I/O

- MMIO (not to be confused with memory-mapped file I/O)
  - **Same address** bus to address memory and I/O devices
  - Access to the I/O devices using regular instructions
  - Most widely used I/O method across the different architectures supported by Linux
- PIO
  - **Different address** spaces for memory and I/O devices
  - Uses a special class of CPU instructions to access I/O devices
  - Example on x86: IN and OUT instructions

MMIO vs PIO



Physical Memory
address space, accessed with
normal load/store instructions



Separate I/O address space,
accessed with specific instructions

Requesting I/O ports

- Tells the kernel which driver is using which I/O ports
- Allows to prevent other drivers from using the same I/O ports, but is purely voluntary.
- ```
  struct resource *request_region(
        unsigned long start,
        unsigned long len,
        char *name);
  ```
- Tries to reserve the given region and returns `NULL` if unsuccessful.
- `request_region(0x0170, 8, "ide1");`
- ```
  void release_region(
      unsigned long start,
      unsigned long len);
  ```

/proc/ioports example (x86)

```
0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0070-0077 : rtc
0080-008f : dma page reg
00a0-00a1 : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : ide1
01f0-01f7 : ide0
0376-0376 : ide1
03f6-03f6 : ide0
03f8-03ff : serial
0800-087f : 0000:00:1f.0
...
```

Accessing I/O ports

- Functions to read/write bytes (`b`), word (`w`) and longs (`l`) to I/O ports:
  - `unsigned in[bwl](unsigned long port)`
  - `void out[bwl](value, unsigned long port)`
- And the strings variants: often more efficient than the corresponding C loop, if the processor supports such operations!
  - `void ins[bwl](unsigned port, void *addr, unsigned long count)`
  - `void outs[bwl](unsigned port, void *addr, unsigned long count)`
- Examples
  - read 8 bits
    - `oldlcr = inb(baseio + UART_LCR)`
  - write 8 bits
    - `outb(MOXA_MUST_ENTER_ENCHANCE, baseio + UART_LCR)`

Requesting I/O memory

- Functions equivalent to `request_region()` and `release_region()`, but for I/O memory.
-
```
struct resource *request_mem_region(
        unsigned long start,
        unsigned long len,
        char *name);
```
-
```
void release_mem_region(
        unsigned long start,
        unsigned long len);
```

/proc/iomem example

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

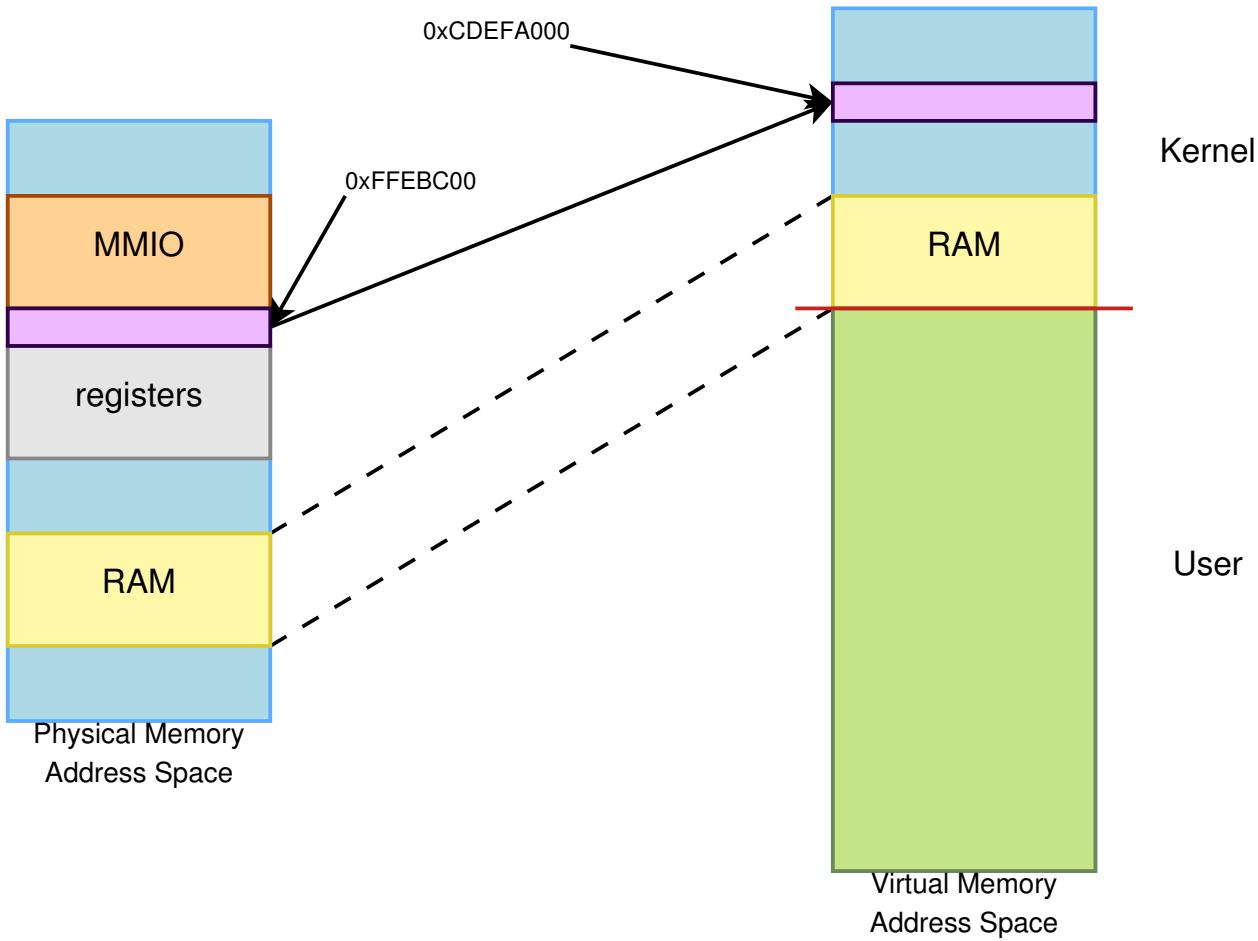
Mapping I/O memory in virtual memory

- Load/store instructions work with virtual addresses
- To access I/O memory, drivers need to have a virtual address that the processor can handle, because I/O memory is not mapped by default in virtual memory.
- The `ioremap` function satisfies this need:

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

void __iomem *ioremap(phys_addr_t phys_addr,
    unsigned long size);
void iounmap(void __iomem *addr);
```

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

ioremap()




```
ioremap(0xFFEBC00, 4096) = 0xCDEFA000
```

Managed API

Using `request_mem_region()` and `ioremap()` in device drivers is now deprecated. You should use the below "managed" functions instead, which simplify driver coding and error handling:

- `devm_ioremap()`
- `devm_iounmap()`
- `devm_request_and_ioremap()`
  - Takes care of both the request and remapping operations!

Accessing MMIO devices

- Directly reading from or writing to addresses returned by `ioremap()` (*pointer dereferencing*) may not work on some architectures.
- To do PCI-style, little-endian accesses, conversion being done automatically

```
unsigned read[bwl](void *addr);
void write[bwl](unsigned val, void *addr);
```

- To do raw access, without endianness conversion

```
unsigned __raw_read[bwl](void *addr);
void __raw_write[bwl](unsigned val, void *addr);
```

- Example
  - 32 bits write

```
__raw_writel(1 << KS8695_IRQ_UART_TX,
    membase + KS8695_INTST);
```

New API for mixed accesses

- A new API allows to write drivers that can work on either devices accessed over PIO or MMIO. A few drivers use it, but there doesn't seem to be a consensus in the kernel community around it.
- Mapping
  - For PIO: `ioport_map()` and `ioport_unmap()`. They don't really map, but they return a special iomem cookie.
  - For MMIO: `ioremap()` and `iounmap()`. As usual.
- Access, works both on addresses or cookies returned by `ioport_map()` and `ioremap()`
  - `ioread[8/16/32]()` and `iowrite[8/16/32]` for single access
  - `ioread[8/16/32]_rep()` and `iowrite[8/16/32]_rep()` for repeated accesses

Avoiding I/O access issues

- Caching on I/O ports or memory already disabled
- Use the macros, they do the right thing for your architecture
- The compiler and/or CPU can reorder memory accesses, which might cause troubles for your devices is they expect one register to be read/written before another one.
  - Memory barriers are available to prevent this reordering
  - `rmb()` is a read memory barrier, prevents reads to cross the barrier
  - `wmb()` is a write memory barrier
  - `mb()` is a read-write memory barrier
- Starts to be a problem with CPUs that reorder instructions and SMP.
- See `Documentation/memory-barriers.txt` for details

/dev/mem

- Used to provide user-space applications with direct access to physical addresses.
- Usage: open `/dev/mem` and read or write at given offset. What you read or write is the value at the corresponding physical address.
- Used by applications such as the X server to write directly to device memory.
- On `x86`, `arm`, `arm64`, `tile`, `powerpc`, `unicore32`, `s390`: `CONFIG_STRICT_DEVMEM` option to restrict `/dev/mem` non-RAM addresses, for security reasons (Linux 3.10 status).

Example: Port I/O via Kernel device driver

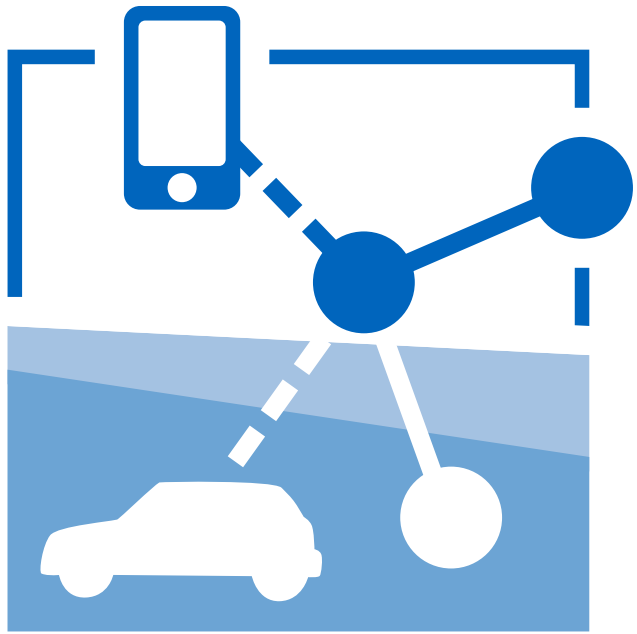
Linux provides the `pcimem` utility to allow reading from and writing to MMIO addresses. The Linux kernel also allows tracing MMIO access from kernel modules (drivers) using the kernel's `mmiotrace` debug facility. To enable this, the Linux kernel should be compiled with the corresponding option enabled.

Discussion & Questions

- Which other possibilities (beside MMIO/PIO) for input/output are available?

Processes, scheduling and interrupts

Daniel Krefft

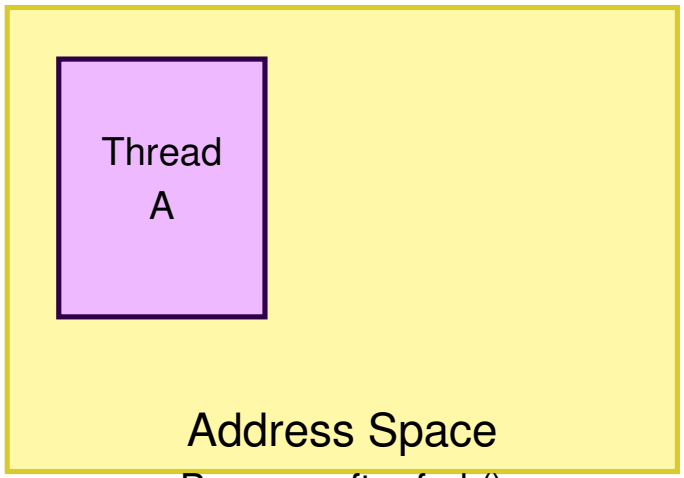


Process, thread?

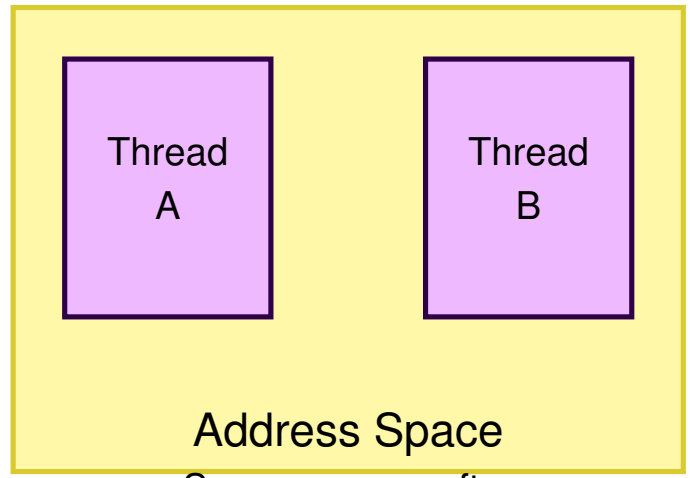
- Confusion about the terms *process*, *thread* and *task*
- In Unix, a process is created using `fork()` and is composed of
  - An address space, which contains the program code, data, stack, shared libraries, etc.
  - One thread, that starts executing the `main()` function.
  - Upon creation, a process contains one thread
  - Each process depends on a parent process in a hierarchical scheme (visualize via `pstree`)
- Additional threads can be created inside an existing process, using `pthread_create()`
  - They run in the same address space as the initial thread of the process
  - They start executing a function passed as argument to `pthread_create()`

Process, thread: kernel point of view

- The kernel represents each thread running in the system by a structure of type `struct task_struct` defined in <sched.h>
- From a scheduling point of view, it makes no difference between the initial thread of a process and all additional threads created dynamically using `pthread_create()`

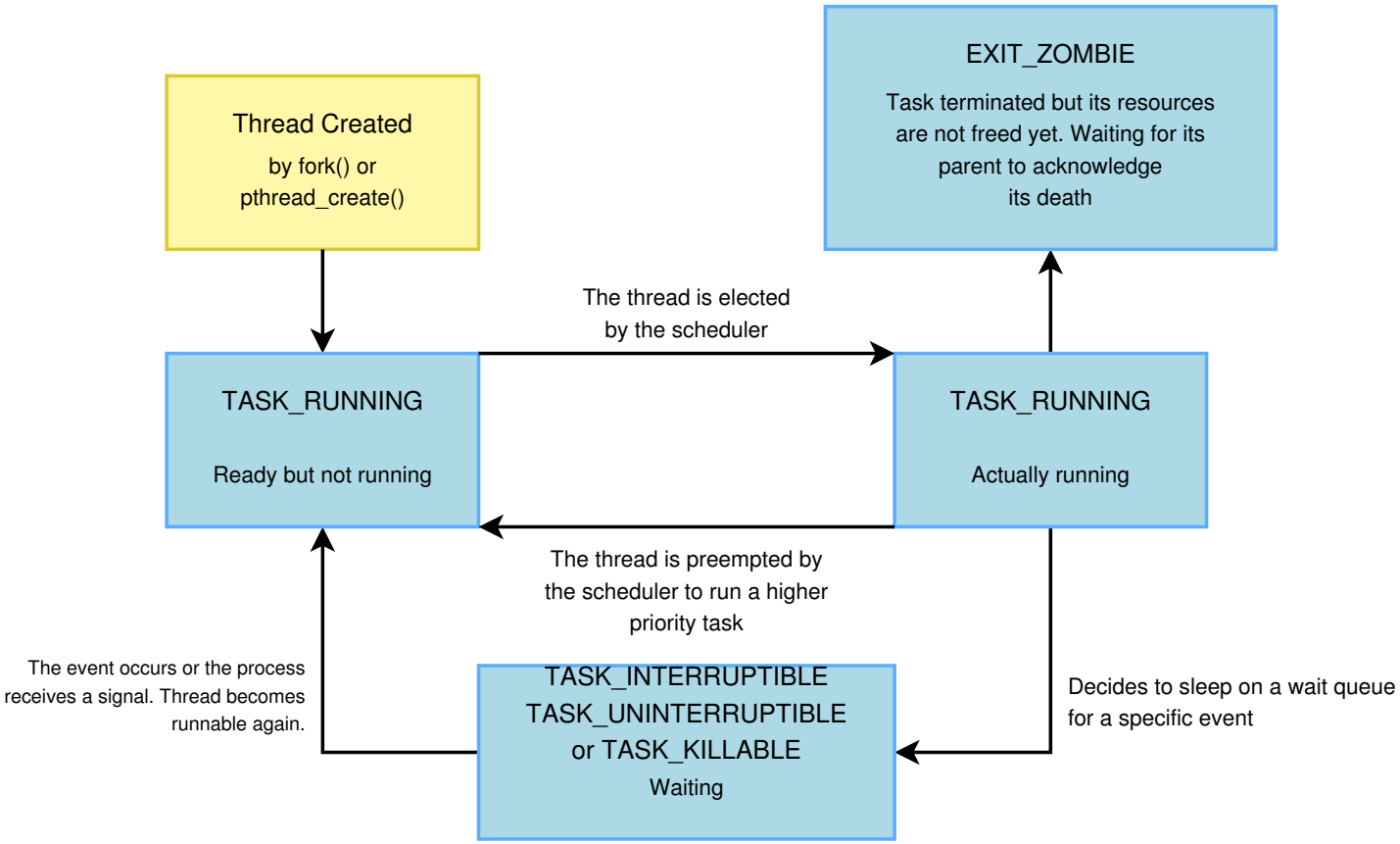


Process after fork()

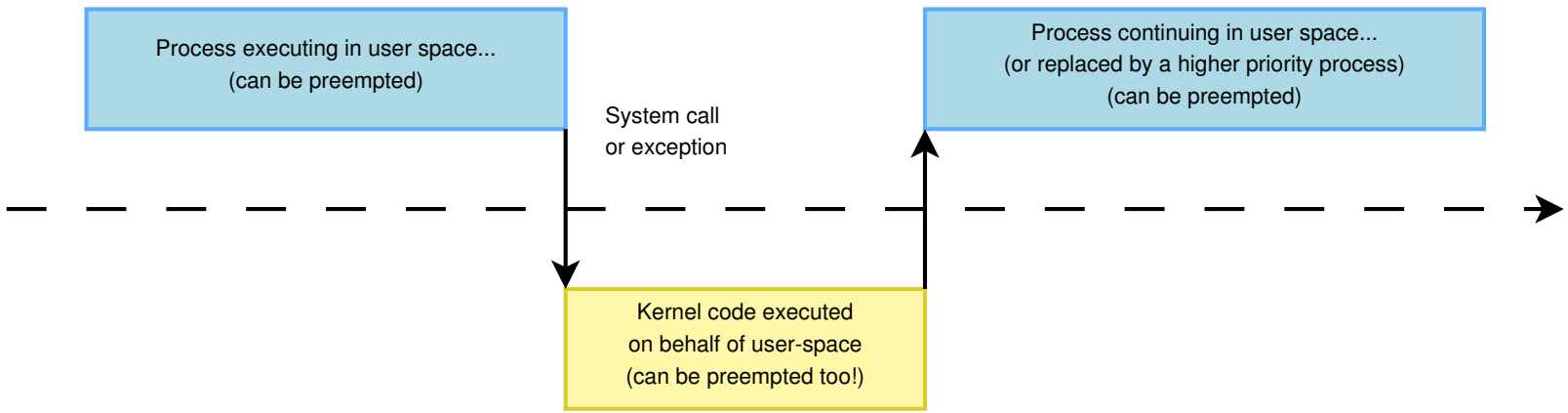


Same process after pthread_create()

A thread life



Execution of system calls

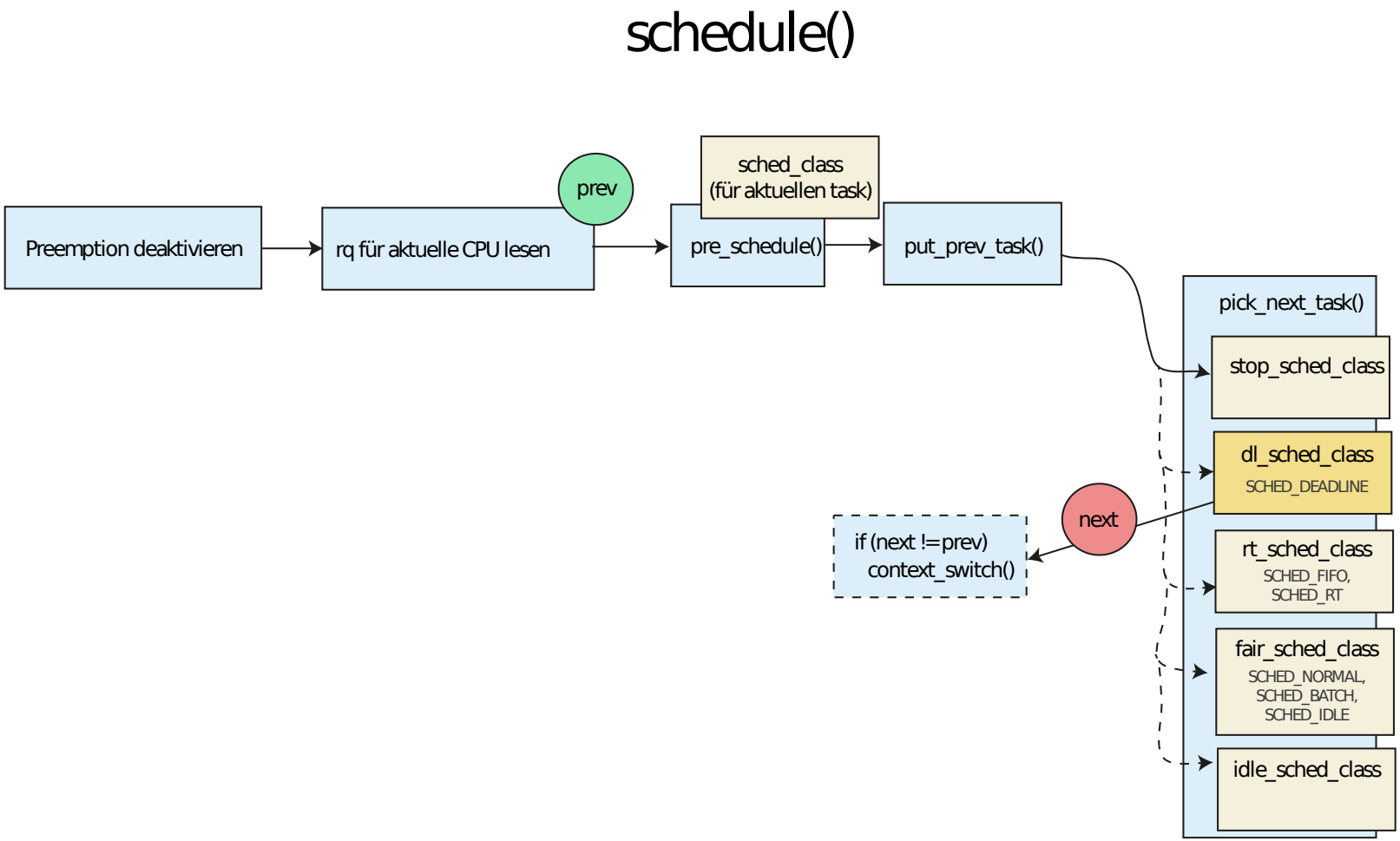


The execution of system calls takes place in the context of the thread requesting them.

Scheduling

- Method of sharing CPU time as fairly as possible AND
- Taking into account different task priorities
- Processes are associated with one of the following scheduling policies
  - SCHED_NORMAL (default) (Completely Fair Scheduler)
  - SCHED_BATCH
  - SCHED_IDLE
  - SCHED_FIFO (Realtime)
  - SCHED_RR (Realtime)
  - SCHED_DEADLINE (Earliest Deadline First)
- Scheduling policies are handled by several scheduling classes (hooks approach)
- `schedule` function is the starting point to an understandig of scheduling operations

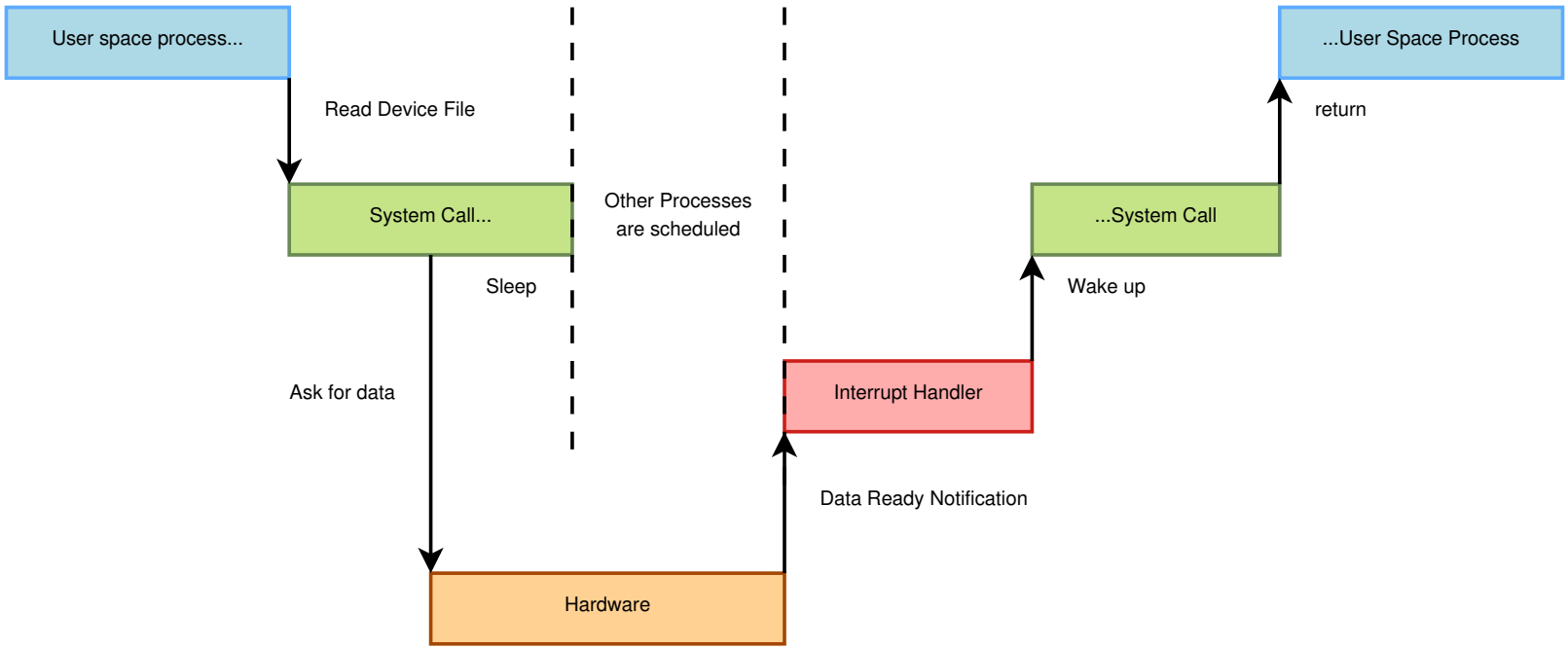
schedule() Function



Processes, scheduling and interrupts

Sleeping

Sleeping



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

How to sleep 1/3

- Must declare a wait queue
- A wait queue will be used to store the list of threads waiting for an event
  - Static queue declaration
    - useful to declare as a global variable
    - `DECLARE_WAIT_QUEUE_HEAD(module_queue);`
  - Or dynamic queue declaration
    - Useful to embed the wait queue inside another data structure

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

How to sleep 2/3

- Several ways to make a kernel process sleep
  - `void wait_event(queue, condition);`
    - Sleeps until the task is woken up and the given C expression is true. Caution: can't be interrupted (can't kill the user-space process!)
  - `int wait_event_killable(queue, condition);`
    - Can be interrupted, but only by a *fatal* signal (`SIGKILL`). Returns `-ERESTARSYS` if interrupted.
  - `int wait_event_interruptible(queue, condition);`
    - Can be interrupted by any signal. Returns `-ERESTARTSYS` if interrupted.

How to sleep 3/3

- `int wait_event_timeout(queue, condition, timeout);`
  - Also stops sleeping when the task is woken up and the timeout expired. Returns `0` if the timeout elapsed, non-zero if the condition was met.
- `int wait_event_interruptible_timeout(queue, condition, timeout);`
  - Same as above, interruptible. Returns `0` if the timeout elapsed, `-ERESTARTSYS` if interrupted, positive value if the condition was met.

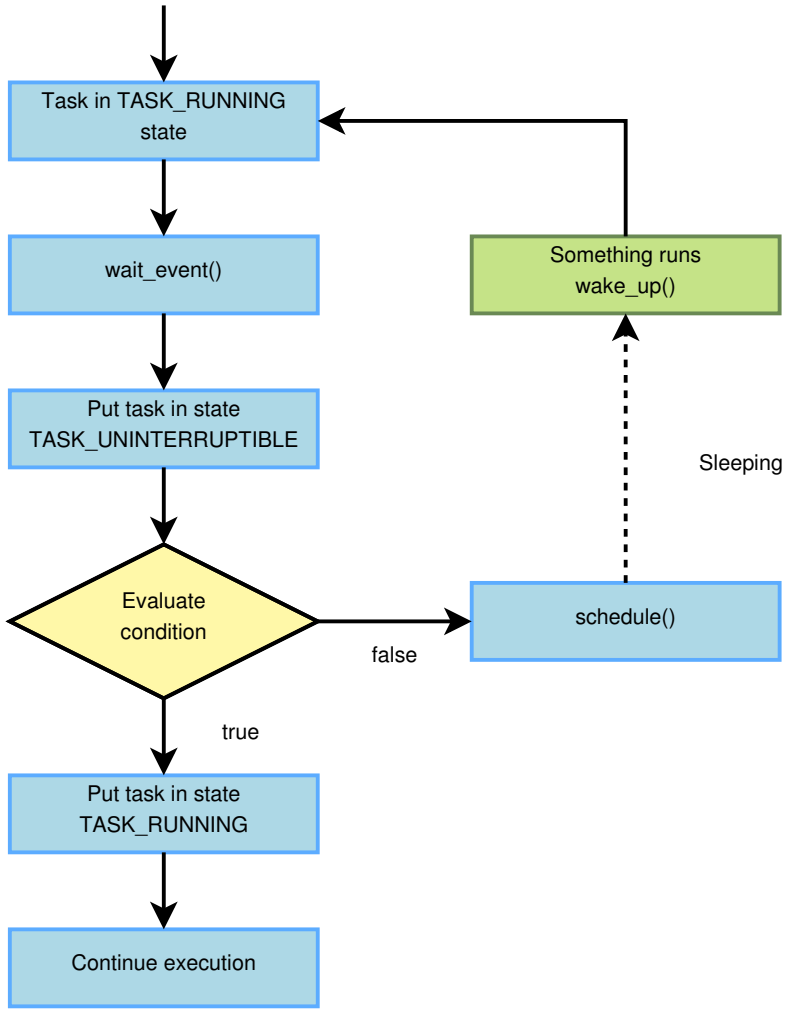
Waking up!

- Typically done by interrupt handlers when data sleeping processes are waiting for become available.
  - `wake_up(&queue);`
    - Wakes up all processes in the wait queue
  - `wake_up_interruptible(&queue);`
    - Wakes up all processes waiting in an interruptible sleep on the given queue

Exclusive vs. non-exclusive

- `wait_event_interruptible()` puts a task in a non-exclusive wait.
  - All non-exclusive tasks are woken up by `wake_up()` / `wake_up_interruptible()`
- `wait_event_interruptible_exclusive()` puts a task in an exclusive wait.
  - `wake_up()` / `wake_up_interruptible()` wakes up all non-exclusive tasks and only one exclusive task
  - `wake_up_all()` / `wake_up_interruptible_all()` wakes up all non-exclusive and all exclusive tasks
- Exclusive sleeps are useful to avoid waking up multiple tasks when only one will be able to "consume" the event.
- Non-exclusive sleeps are useful when the event can "benefit" to multiple tasks.

Sleeping and waking up - Implementation 1/2



Sleeping and waking up - Implementation 2/2

The scheduler doesn't keep evaluating the sleeping condition!

- `wait_event(queue, condition);`
  - The process is put in the `TASK_UNINTERRUPTIBLE` state.
- `wake_up(&queue);`
  - All processes waiting in `queue` are woken up, so they get scheduled later and have the opportunity to evaluate the condition again and go back to sleep if it is not met.

See `include/linux/wait.h` for implementation details.

Interrupt Management

Interrupts

- System calls are not the only way of switching between user and system mode
- Two types of interrupt are distinguished:
  - **Hardware Interrupts** are produced automatically by the system and connected peripherals
  - **SoftIRQs** are used to effectively implement deferred activities in the kernel itself

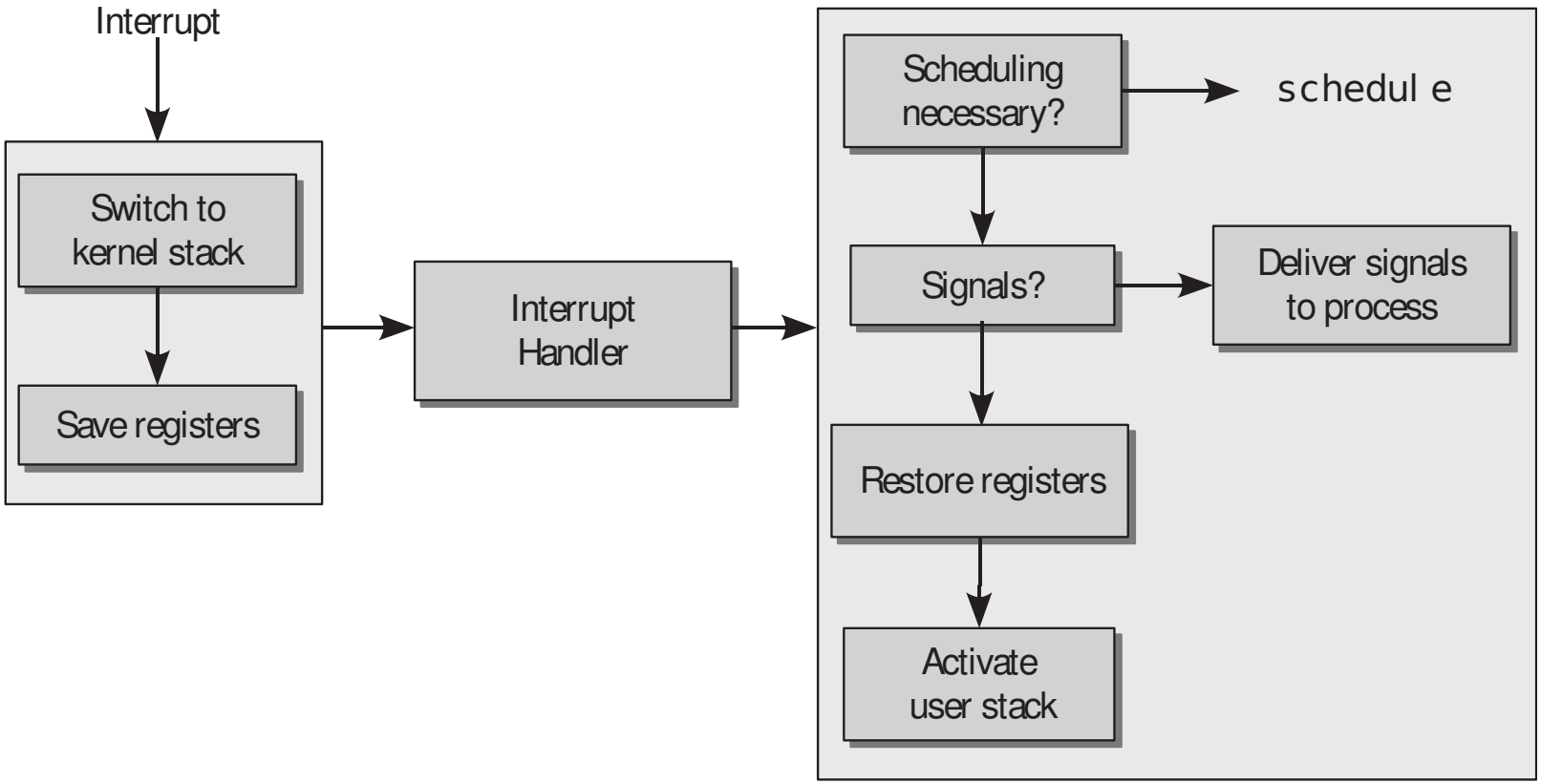
Example: /proc/interrupts

```
           CPU0       CPU1
39:        4          0          GIC     TWL6030-PIH
41:        0          0          GIC     l3-dbg-irq
42:        0          0          GIC     l3-app-irq
43:        0          0          GIC     prcm
44:        20294      0          GIC     DMA
52:        0          0          GIC     gpmc
...
IPI0:      0          0          Timer broadcast interrupts
IPI1:      23095      25663      Rescheduling interrupts
IPI2:      0          0          Function call interrupts
IPI3:      231        173        Single function call interrupts
IPI4:      0          0          CPU stop interrupts
LOC:       196407     136995     Local timer interrupts
Err:       0
```

Interrupt Types

- **Synchronous Interrupts** and **Exceptions** are produced by the CPU itself and are directed at the program currently executing
- **Asynchronous interrupts** are the classical interrupt type generated by peripheral devices and occur at arbitrary times

Interrupt Processing



Interrupt Handlers

- No guarantee in which address space the system will be in when the interrupt occurs: can't transfer data to and from user space
- Interrupt handler execution is managed by the CPU, not by the scheduler. Handlers can't run actions that may sleep, because there is nothing to resume their execution. In particular, need to allocate memory with `GFP_ATOMIC`.
- Interrupt handlers are run with all interrupts disabled. Therefore, they have to complete their job **quickly** enough, to avoiding **blocking interrupts** for too long.

How to support rapid processing?

Answer: Not every part of an ISR is equally important.

- **Critical** actions must be executed immediately following an interrupt. Otherwise, system stability or correct operation of the computer cannot be maintained. Other interrupts must be disabled when such actions are performed.
- **Noncritical** actions should also be performed as quickly as possible but with enabled interrupts (they may therefore be interrupted by other system events).
- **Deferrable** actions are not particularly important and need not be implemented in the interrupt handler. The kernel can delay these actions and perform them when it has nothing better to do (*tasklets*).

Typical Interrupt Handler's job

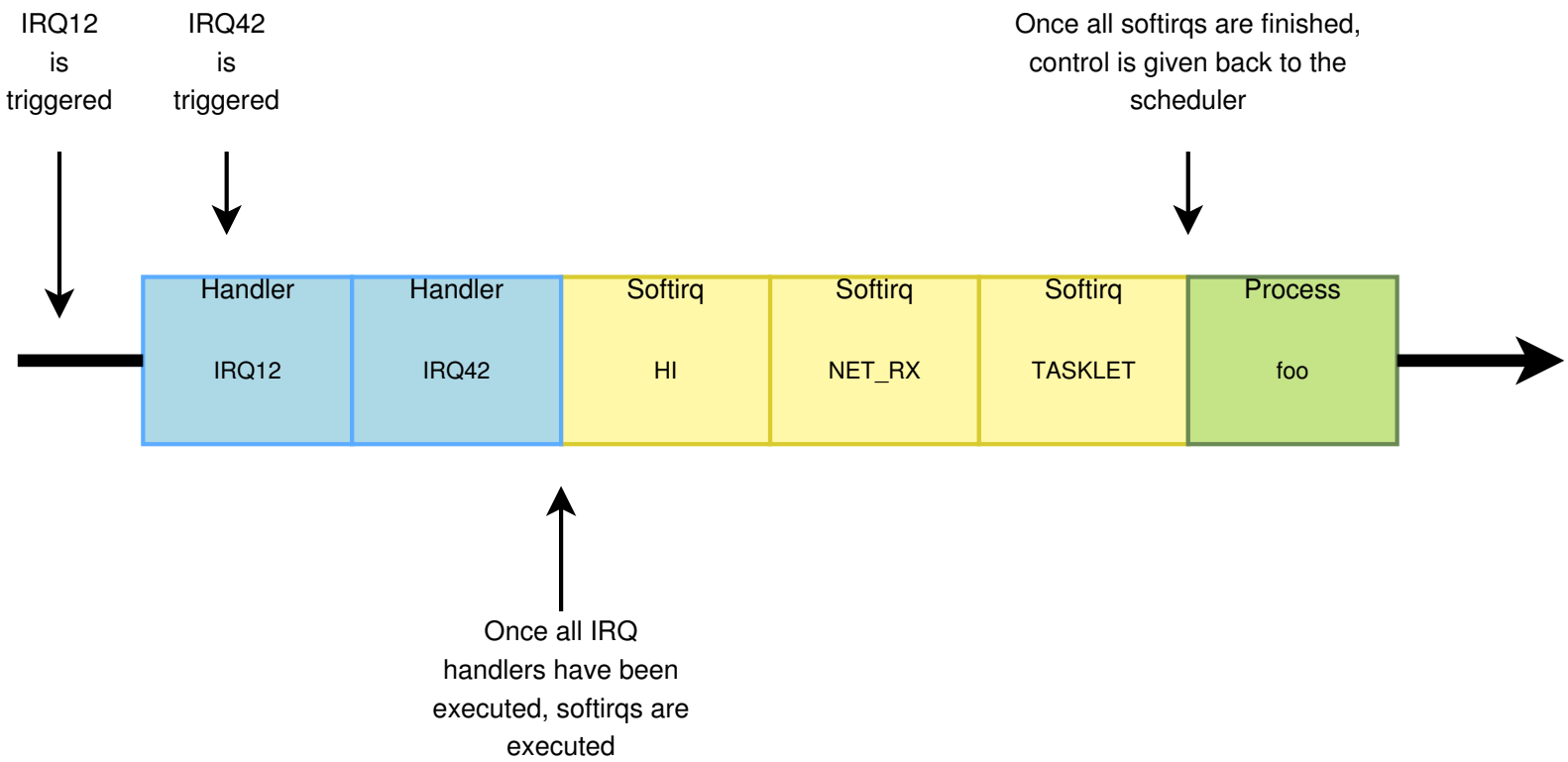
- Acknowledge the interrupt to the device (otherwise no more interrupts will be generated, or the interrupt will keep firing over and over again)
- Read/write data from/to the device
- Wake up any waiting process waiting for the completion of an operation, typically using wait queues

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

Top half and bottom half processing

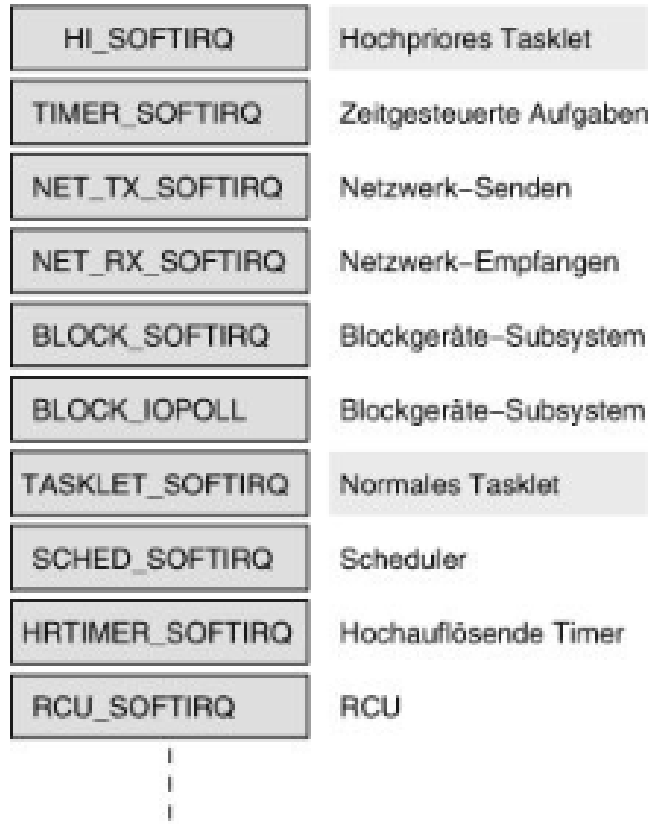
- Splitting the execution of interrupt handlers in 2 parts
  - Top half
    - This is the real interrupt handler, which should complete as quickly as possible since all interrupts are disabled. If possible, take the data out of the device and schedule a bottom half to handle it.
  - Bottom half
    - Is the general Linux name for various mechanisms which allow to postpone the handling of interrupt-related work. Implemented in Linux as softirqs, tasklets or workqueues.

Top half and bottom half diagram



Softirqs

- Softirqs are a form of bottom half processing
- The softirqs handlers are executed with all interrupts enabled, and a given softirq handler can run simultaneously on multiple CPUs
- They are executed once all interrupt handlers have completed, before the kernel resumes scheduling processes, so sleeping is not allowed.
- The number of softirqs is fixed in the system, so softirqs are not directly used by drivers, but by complete kernel subsystems (network, etc.)

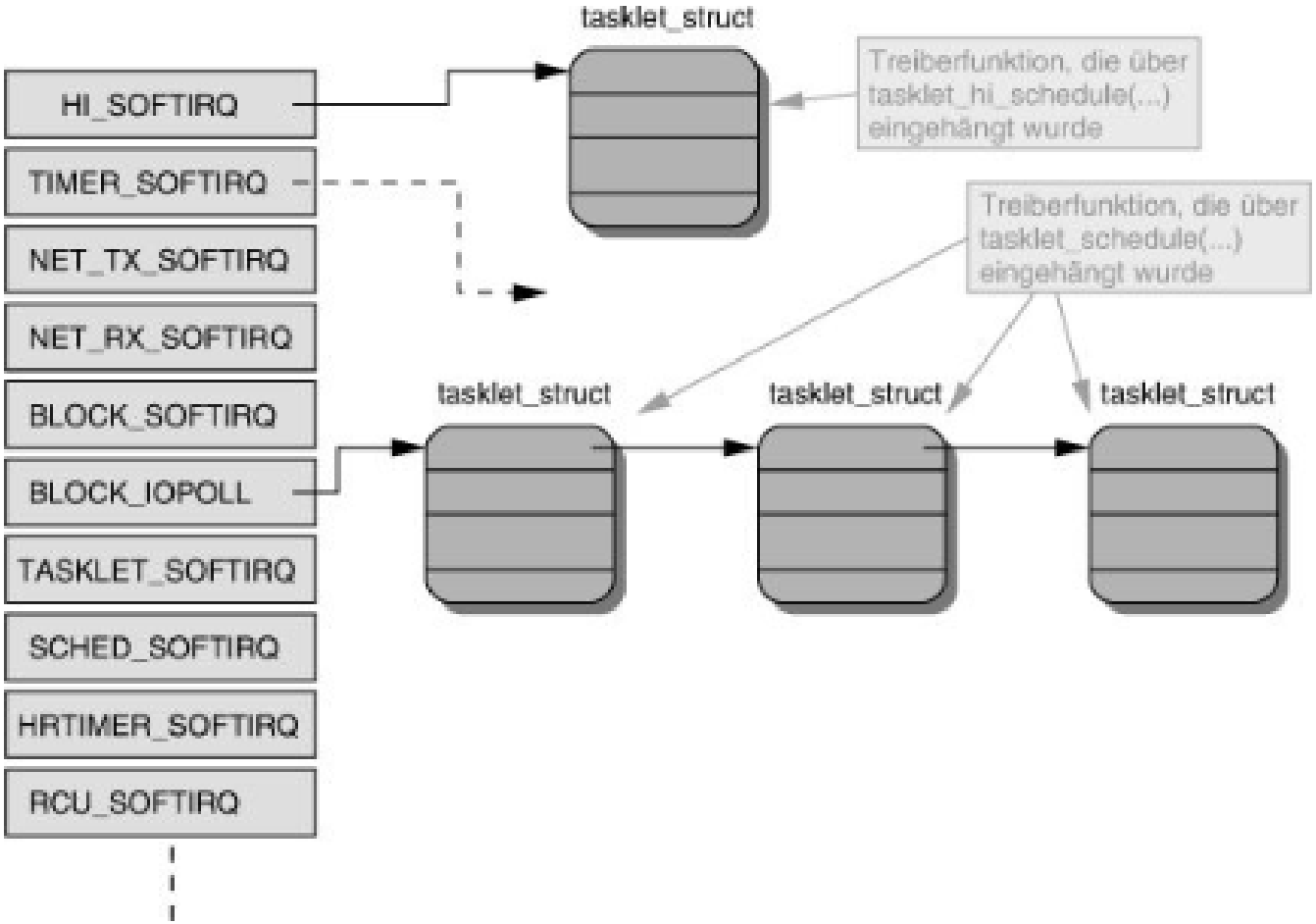


Tasklets

- Tasklets are executed within the `HI` and `TASKLET` softirqs. They are executed with all interrupts enabled, but a given tasklet is guaranteed to execute on a single CPU at a time.
- A tasklet can be declared statically with the `DECLARE_TASKLET()` macro or dynamically with the `tasklet_init()` function. A tasklet is simply implemented as a function. Tasklets can easily be used by individual device drivers, as opposed to softirqs.
- The interrupt handler can schedule the execution of a tasklet with
  - `tasklet_schedule()` to get it executed in the `TASKLET` softirq
  - `tasklet_hi_schedule()` to get it executed in the `HI` softirq (higher priority)

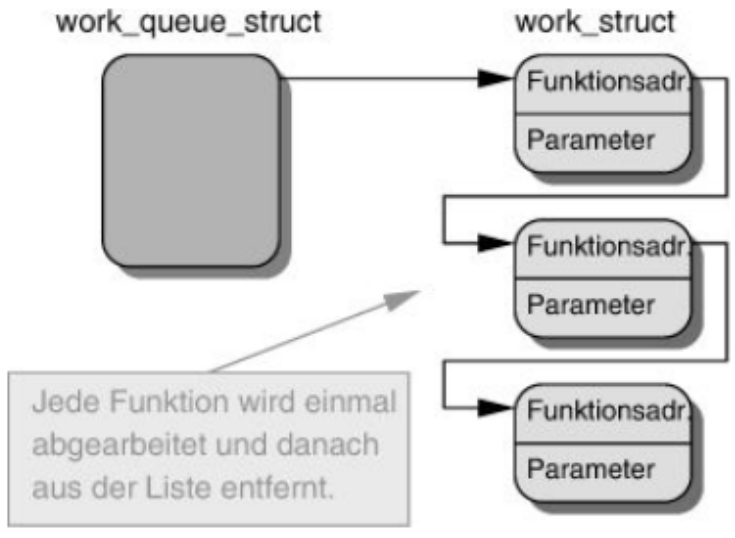
Tasklet Scheduling

- Schedule Tasklet in ISR with *tasklet_schedule* or *tasklet_hi_schedule*



Workqueues

- Workqueues are a general mechanism for deferring work. It is not limited in usage to handling interrupts.
- The function registered as workqueue is executed in a thread, which means:
  - Lower priority than tasklets (not critical)
  - All interrupts are enabled
  - Sleeping is allowed



Interrupt management summary

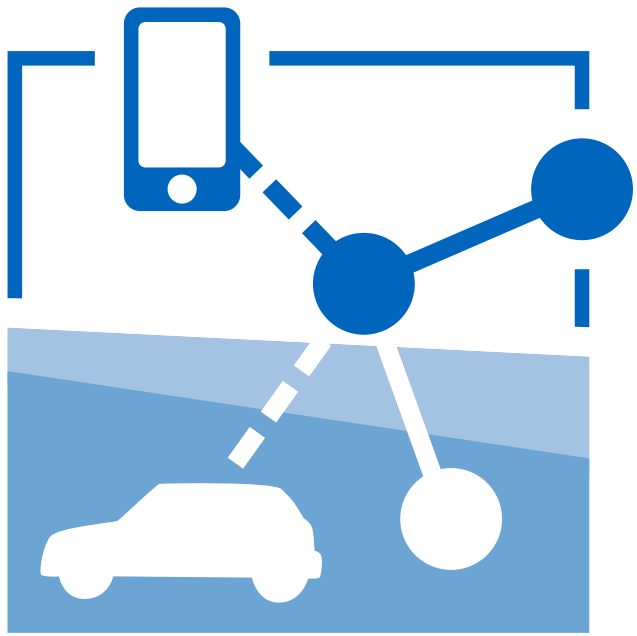
- Device driver
  - When the device file is first opened, register an interrupt handler for the device's interrupt channel.
- Interrupt handler
  - Called when an interrupt is raised.
  - Acknowledge the interrupt
  - If needed, schedule a tasklet taking care of handling data. Otherwise, wake up processes waiting for the data.
- Tasklet
  - Process the data
  - Wake up processes waiting for the data
- Device driver
  - When the device is no longer opened by any process, unregister the interrupt handler.

Discussion & Questions

No questions so far - great!

Concurrent Access to Resources: Locking

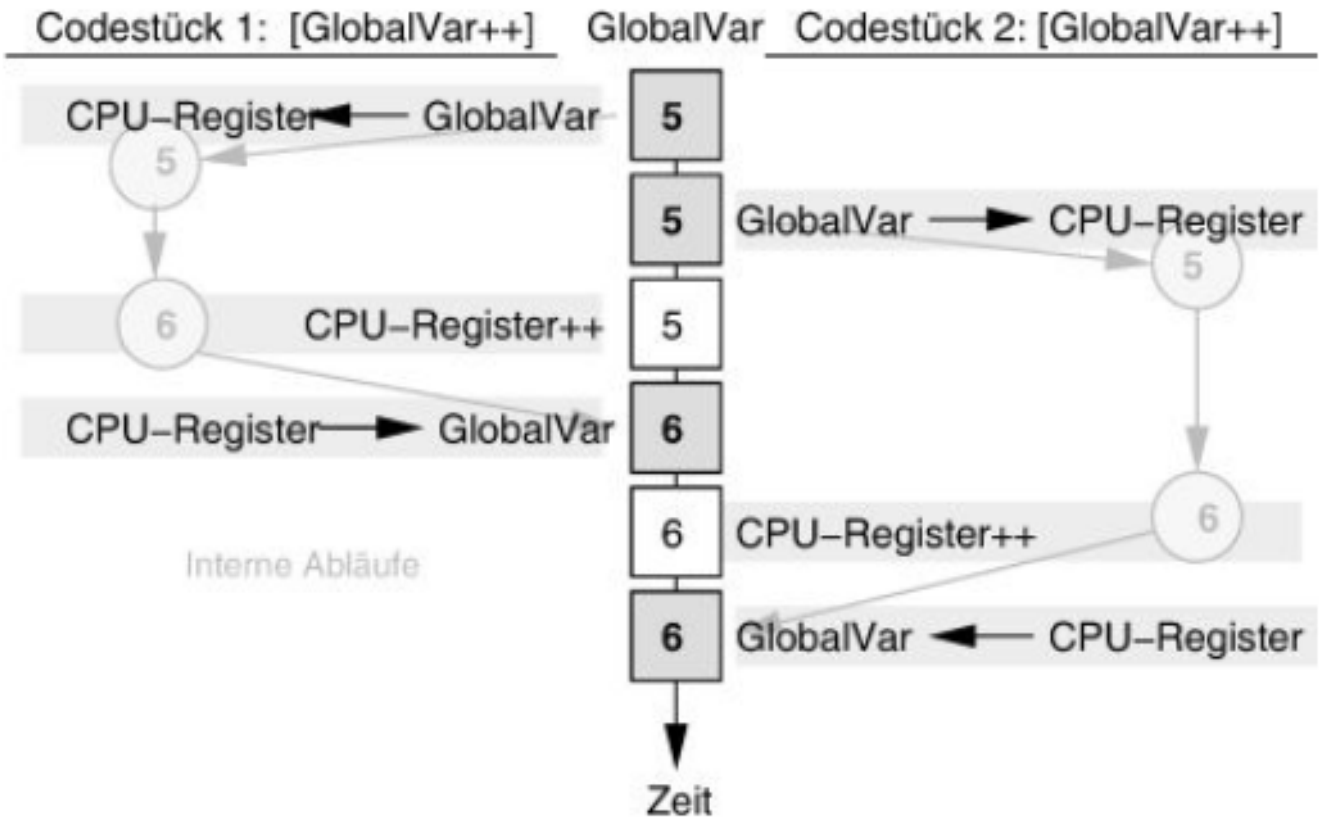
Daniel Krefft



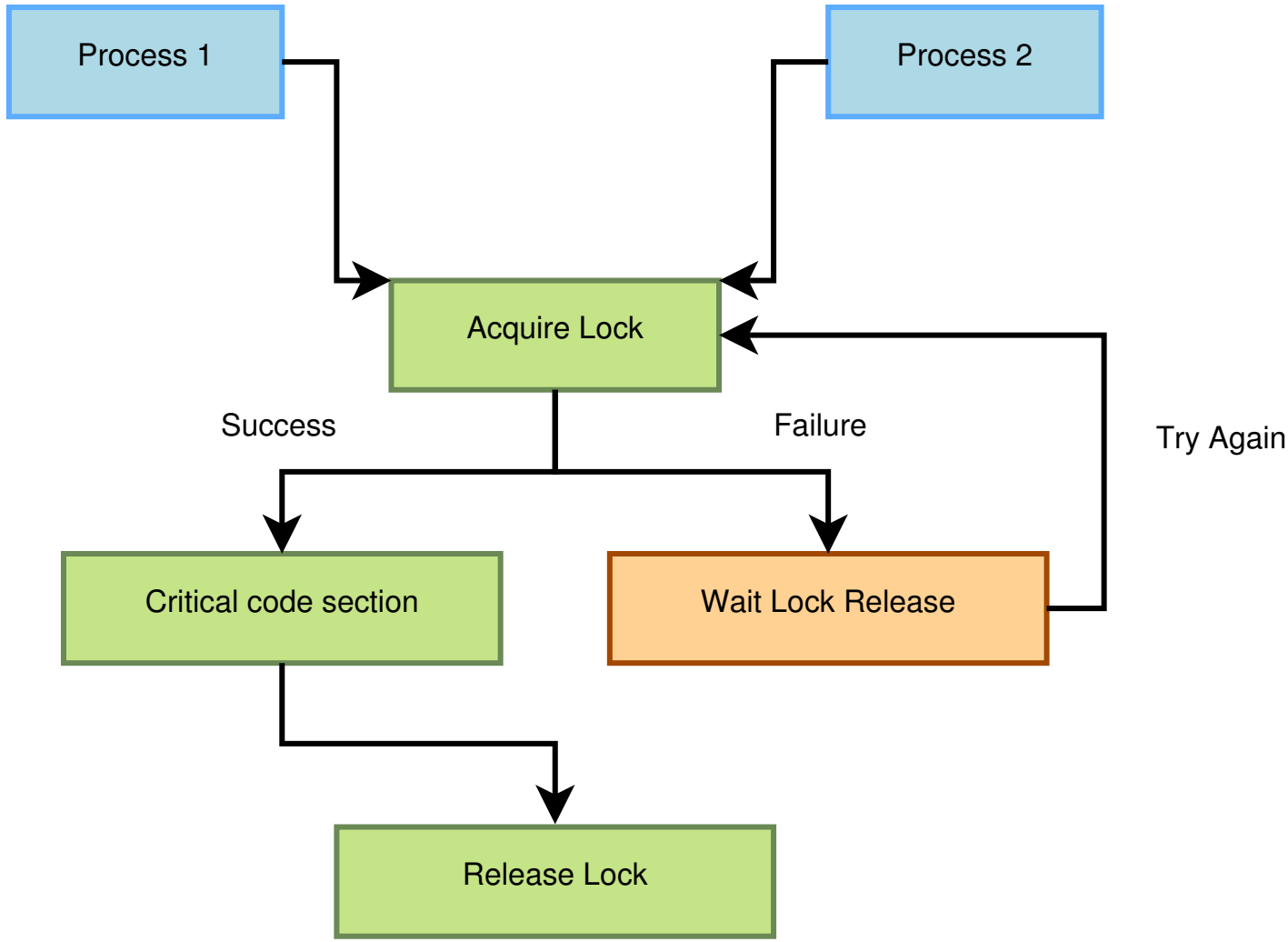
Sources of concurrency issues

- In terms of concurrency, the kernel has the same constraint as a multi-threaded program: its state is global and visible in all executions contexts
- Concurrency arises because of
  - *Interrupts*, which interrupts the current thread to execute an interrupt handler. They may be using shared resources.
  - *Kernel preemption*, if enabled, causes the kernel to switch from the execution of one system call to another. They may be using shared resources.
  - *Multiprocessing*, in which case code is really executed in parallel on different processors, and they may be using shared resources as well.
- The solution is to keep as much local state as possible and for the shared resources, use locking.

Critical sections: Race Condition



Concurrency protection with locks



Linux mutexes

- The kernel's main locking primitive
- The process requesting the lock blocks when the lock is already held. Mutexes can therefore only be used in contexts where sleeping is allowed.
- Mutex definition:
  - `#include <linux/mutex.h>`
- Initializing a mutex statically:
  - `DEFINE_MUTEX(name);`
- Or initializing a mutex dynamically:
  - `void mutex_init(struct mutex *lock);`

Locking and Unlocking Mutexes 1/2

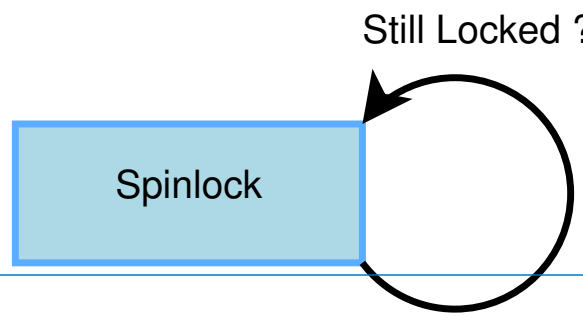
- `void mutex_lock(struct mutex *lock);`
  - Tries to lock the mutex, sleeps otherwise.
  - Caution: can't be interrupted, resulting in processes you cannot kill!
- `int mutex_lock_killable(struct mutex *lock);`
  - Same, but can be interrupted by a fatal (`SIGKILL`) signal. If interrupted, returns a non zero value and doesn't hold the lock. Test the return value!!!
- `int mutex_lock_interruptible(struct mutex *lock);`
  - Same, but can be interrupted by any signal.

Locking and Unlocking Mutexes 2/2

- `int mutex_trylock(struct mutex *lock);`
  - Never waits. Returns a non zero value if the mutex is not available.
- `int mutex_is_locked(struct mutex *lock);`
  - Just tells whether the mutex is locked or not.
- `void mutex_unlock(struct mutex *lock);`
  - Releases the lock. Do it as soon as you leave the critical section.

Spinlocks

- Locks to be used for code that is not allowed to sleep (interrupt handlers), or that doesn't want to sleep (critical sections). Be very careful not to call functions which can sleep!
- Originally intended for multiprocessor systems
- Spinlocks never sleep and keep spinning in a loop until the lock is available.
- Spinlocks cause kernel preemption to be disabled on the CPU executing them.
- The critical section protected by a spinlock is not allowed to sleep.



Initializing Spinlocks

- Statically
  - `DEFINE_SPINLOCK(my_lock);`
- Dynamically
  - `void spin_lock_init(spinlock_t *lock);`

Using Spinlocks 1/2

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

Using Spinlocks 2/2

- `void spin_lock_bh(spinlock_t *lock);`
- `void spin_unlock_bh(spinlock_t *lock);`
  - Disables software interrupts, but not hardware ones.
  - Useful to protect shared data accessed in process context and in a soft interrupt (*bottom half*).
  - No need to disable hardware interrupts in this case.
- Note that reader / writer spinlocks also exist.

Spinlock example

- Spinlock structure embedded into `struct uart_port`

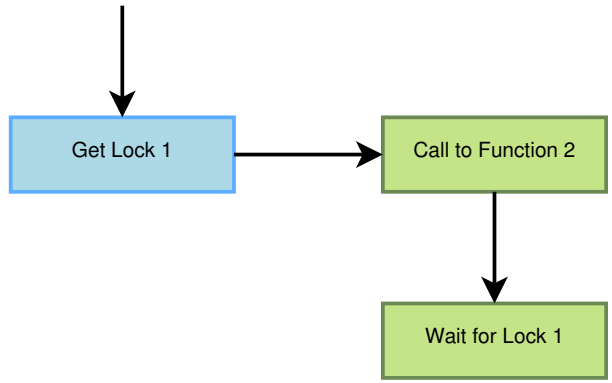
```
struct uart_port {
        spinlock_t lock;
        /* Other fields */
};
```

- Spinlock taken/released with protection against interrupts

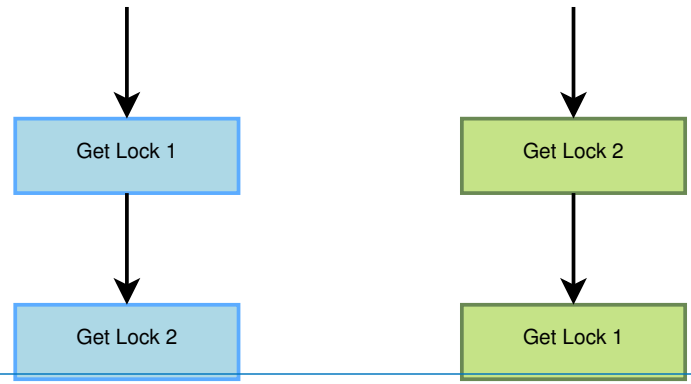
```
static unsigned int ulite_tx_empty
    (struct uart_port *port) {
        unsigned long flags;

        spin_lock_irqsave(&port->lock, flags);
        /* Do something */
        spin_unlock_irqrestore(&port->lock, flags);
}
```

Deadlock Situations

- They can lock up your system. Make sure they never happen!
- Don't call a function that can try to get access to the same lock

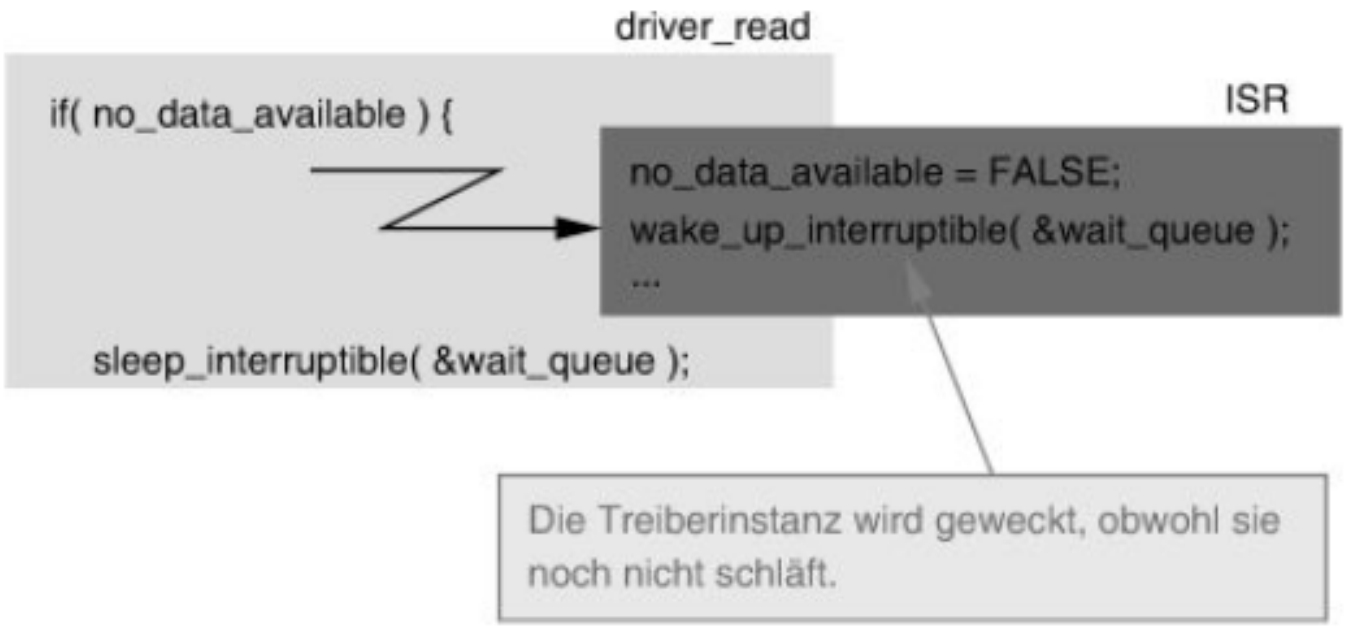


- Holding multiple locks is risky!



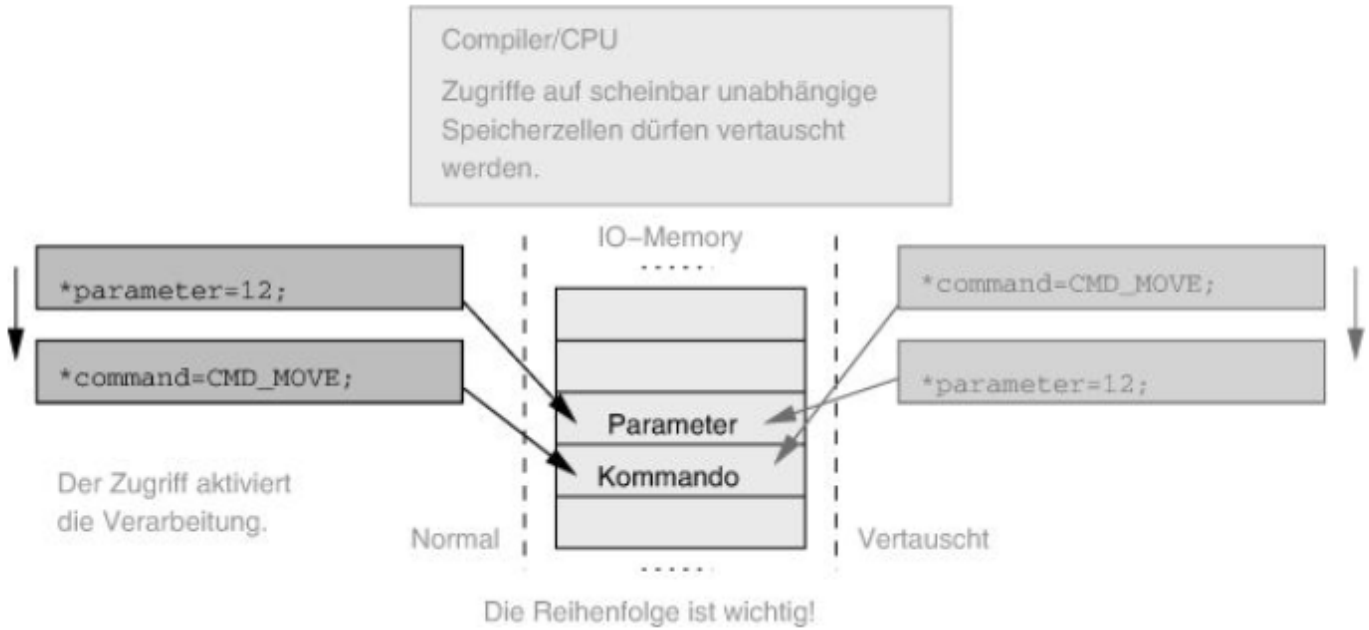
Critical sections: synchronisation

- Critical sections in synchronisation of independent instances
- *wait_event_interruptible*, *wait_event*



Critical sections: memory barriers

- Reordering by compiler and processor



Kernel lock validator

- From Ingo Molnar and Arjan van de Ven
  - Adds instrumentation to kernel locking code
  - Detect violations of locking rules during system life, such as:
    - Locks acquired in different order (keeps track of locking sequences and compares them).
    - Spinlocks acquired in interrupt handlers and also in process context when interrupts are enabled.
  - Not suitable for production systems but acceptable overhead in development.
- See `Documentation/lockdep-design.txt` for details

Critical sections: Overview

	Treiberinstanz	Kernel–Thread	Workqueue Event–Workqueue	Softirq	Tasklet Timer	Hardirq
Treiberinstanz	Mutex RT–Mutex RW–Semaphore Spinlock RW–Lock Seqlock	Mutex RT–Mutex RW–Semaphore Spinlock RW–Lock Seqlock	Spinlock RW–Lock Seqlock	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave
Kernel–Thread	Mutex RT–Mutex RW–Semaphore Spinlock RW–Lock Seqlock	Mutex RT–Mutex RW–Semaphore Spinlock RW–Lock Seqlock	Spinlock RW–Lock Seqlock	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave
Workqueue Event–Workqueue	Spinlock RW–Lock Seqlock	Spinlock RW–Lock Seqlock	Spinlock RW–Lock Seqlock	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave

Critical sections: Overview cont.

	Softirq	Tasklet Timer	Hardirq
Softirq	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock RW–Lock	Spinlock RW–Lock	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave
Tasklet Timer	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave	Spinlock–irqsave RW–Lock–irqsave	Spinlock RW–Lock	Spinlock RW–Lock 1)	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave
Hardirq	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock RW–Lock 2)

1) Nur bei unterschiedlichen Tasklets. Ein Tasklet läuft zu einem Zeitpunkt garantiert nicht zweimal.
2) Nur bei unterschiedlichen ISRs. Ein und dieselbe ISR läuft zu einem Zeitpunkt nicht zweimal ab.

Alternatives to Locking

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

Atomic Variables 1/2

- Useful when the shared resource is an integer value
- Even an instruction like `n++` is not guaranteed to be atomic on all processors!
- Atomic operations definitions
  - `#include <asm/atomic.h>`
- `atomic_t`
  - Contains a signed integer (at least 24 bits)
- Atomic operations (main ones)
  - Set or read the counter:
    - `void atomic_set(atomic_t *v, int i);`
    - `int atomic_read(atomic_t *v);`
  - Operations without return value:
    - `void atomic_inc(atomic_t *v);`
    - `void atomic_dec(atomic_t *v);`
    - `void atomic_add(int i, atomic_t *v);`
    - `void atomic_sub(int i, atomic_t *v);`

Atomic Variables 2/2

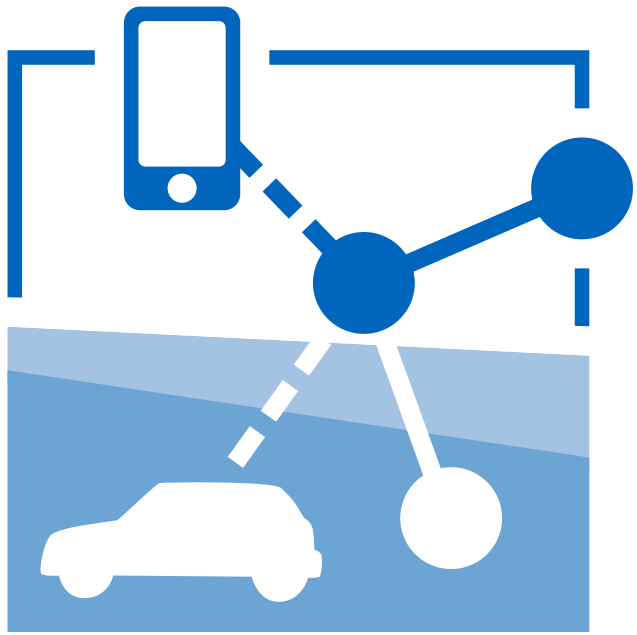
- Similar functions testing the result:
  - `int atomic_inc_and_test(...);`
  - `int atomic_dec_and_test(...);`
  - `int atomic_sub_and_test(...);`
- Functions returning the new value:
  - `int atomic_inc_return(...);`
  - `int atomic_dec_return(...);`
  - `int atomic_add_return(...);`
  - `int atomic_sub_return(...);`

Atomic Bit Operations

- Supply very fast, atomic operations
- On most platforms, apply to an unsigned long type.
- Apply to a void type on a few others.
- Set, clear, toggle a given bit:
  - `void set_bit(int nr, unsigned long * addr);`
  - `void clear_bit(int nr, unsigned long * addr);`
  - `void change_bit(int nr, unsigned long * addr);`
- Test bit value:
  - `int test_bit(int nr, unsigned long *addr);`
- Test and modify (return the previous value):
  - `int test_and_set_bit(...);`
  - `int test_and_clear_bit(...);`
  - `int test_and_change_bit(...);`

Kernel Resources

Daniel Krefft

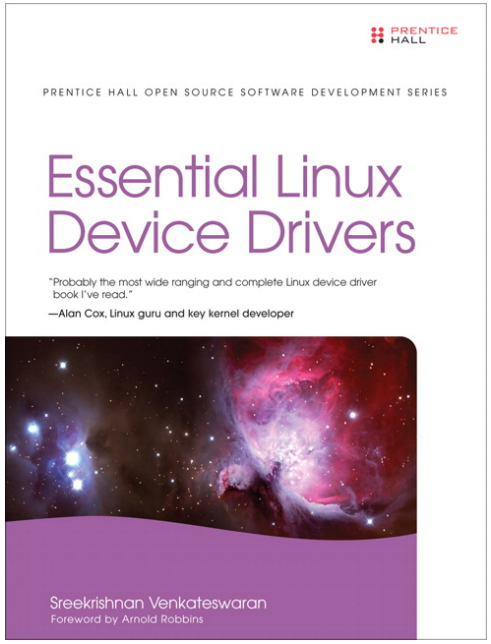


Kernel Development News

- Linux Weekly News
  - `http://lwn.net/`
  - The weekly digest off all Linux and free software information sources
  - In depth technical discussions about the kernel
  - Subscribe to finance the editors ($7 / month)
  - Articles available for non subscribers after 1 week.

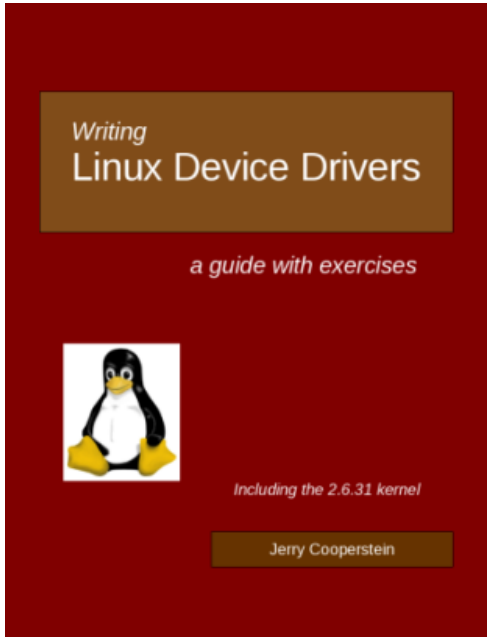
Useful Reading (1)

- Essential Linux Device Drivers, April 2008
  - `http://elinuxdd.com/`
  - By Sreekrishnan Venkateswaran, an embedded IBM engineer with more than 10 years of experience
  - Covers a wide range of topics not covered by LDD: serial drivers, input drivers, I2C, PCMCIA and Compact Flash, PCI, USB, video drivers, audio drivers, block drivers, network drivers, Bluetooth, IrDA, MTD, drivers in userspace, kernel debugging, etc.
  - *Probably the most wide ranging and complete Linux device driver book I've read* – Alan Cox



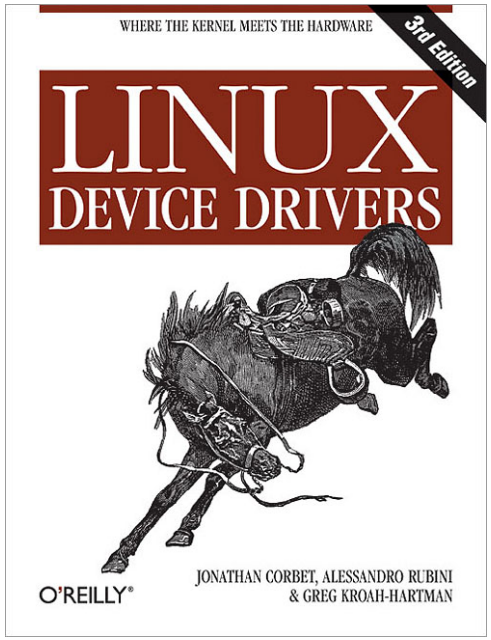
Useful Reading (2)

- Writing Linux Device drivers, September 2009
  - `http://www.coopj.com/`
  - Self published by Jerry Cooperstein
  - Available like any other book (Amazon and others)
  - Though not as thorough as the previous book on specific drivers, still a good complement on multiple aspects of kernel and device driver development.
  - Based on Linux 2.6.31
  - Multiple exercises. Updated solutions for 2.6.36.



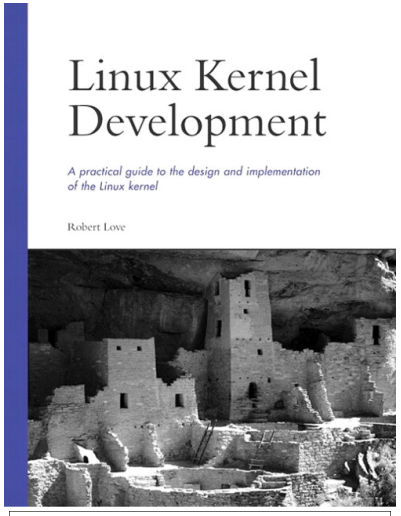
Useful Reading (3)

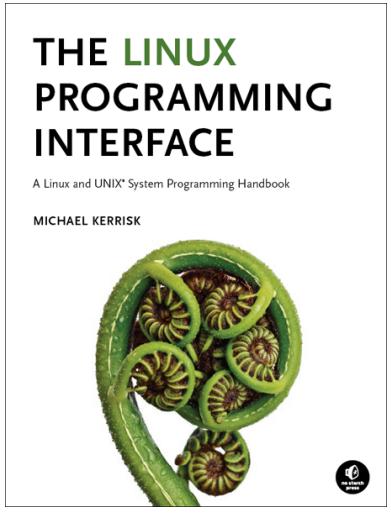
- Linux Device Drivers, 3rd edition, Feb 2005
  - `http://www.oreilly.com/catalog/linuxdrive3/`
  - By Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, O'Reilly
  - Freely available on-line! Great companion to the printed book for easy electronic searches!
  - `http://lwn.net/Kernel/LDD3/` (1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



Useful Reading (4)

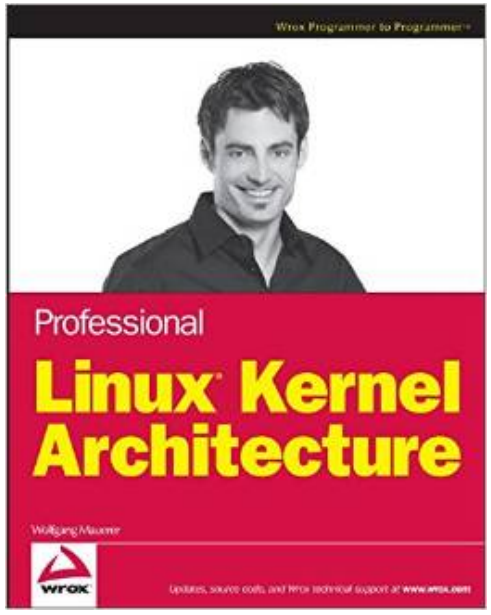
- Linux Kernel Development, 3rd Edition, Jun 2010
  - Robert Love, Novell Press
  - `http://free-electrons.com/redir/lkd3-book.html`
  - A very synthetic and pleasant way to learn about kernel subsystems (beyond the needs of device driver writers)
- The Linux Programming Interface, Oct 2010
  - Michael Kerrisk, No Starch Press
  - `http://man7.org/tlpi/`
  - A gold mine about the kernel interface and how to use it





Useful Reading (5)

- Professional Linux Kernel Architecture, October 2008
  - `http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470343435.html`
  - By Wolfgang Mauerer, John Wiley & Sons
  - Find an introduction to the architecture, concepts and algorithms of the Linux kernel in Professional Linux Kernel Architecture, a guide to the kernel sources and large number of connections among subsystems.



Useful Online Resources

- Kernel documentation (`Documentation/` in kernel sources)
  - Available on line: `http://free-electrons.com/kerneldoc/` (with HTML documentation extracted from source code)
- Linux kernel mailing list FAQ
  - `http://www.tux.org/lkml/`
  - Complete Linux kernel FAQ
  - Read this before asking a question to the mailing list
- Kernel Newbies
  - `http://kernelnewbies.org/`
  - Glossary, articles, presentations, HOWTOs, recommended reading, useful tools for people getting familiar with Linux kernel or driver development.
- Kernel glossary
  - `http://kernelnewbies.org/KernelGlossary`

International Conferences

- Embedded Linux Conference: `http://embeddedlinuxconference.com/`
  - Organized by the CE Linux Forum:
  - in California (San Francisco, April)
  - in Europe (October-November)
  - Very interesting kernel and userspace topics for embedded systems developers.
  - Presentation slides freely available
- Linux Plumbers: `http://linuxplumbersconf.org`
  - Conference on the low-level plumbing of Linux: kernel, audio, power management, device management, multimedia, etc.
- linux.conf.au: `http://linux.org.au/conf/`
  - In Australia / New Zealand
  - Features a few presentations by key kernel hackers.
- Don't miss our free conference videos on `http://free-electrons.com/community/videos/conferences/`

ARM resources

- ARM Linux project: `http://www.arm.linux.org.uk/`
  - Developer documentation: `http://www.arm.linux.org.uk/developer/`
  - linux-arm-kernel mailing list: `http://lists.infradead.org/mailman/listinfo/linux-arm-kernel`
  - FAQ: `http://www.arm.linux.org.uk/armlinux/mlfaq.php`
- Linaro: `http://linaro.org`
  - Many optimizations and resources for recent ARM CPUs (toolchains, kernels, debugging utilities...).
- ARM Limited: `http://www.linux-arm.com/`
  - Wiki with links to useful developer resources
- See our Embedded Linux course for details about toolchains: `http://free-electrons.com/training/embedded-linux/`