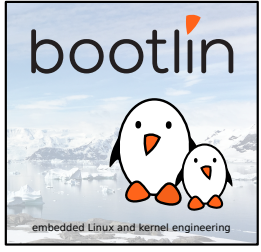
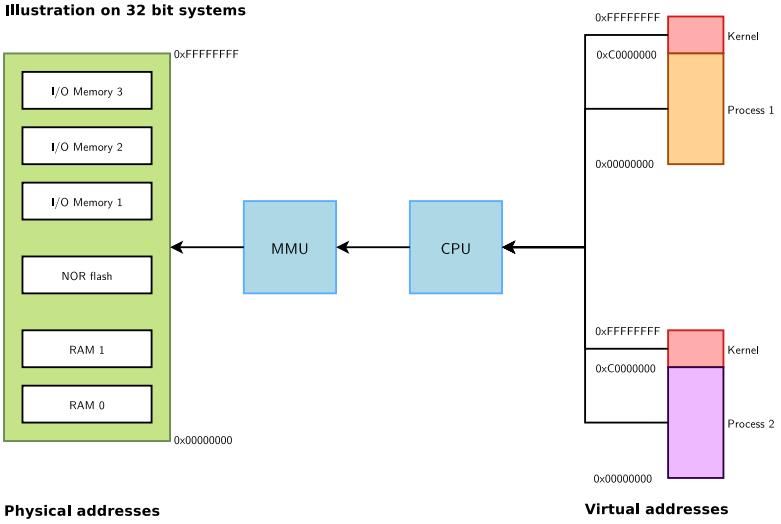
Memory Management

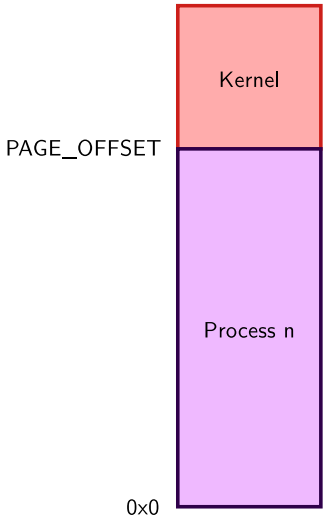


ReDS
Reconfigurable & embedded Digital Systems

Physical and virtual memory

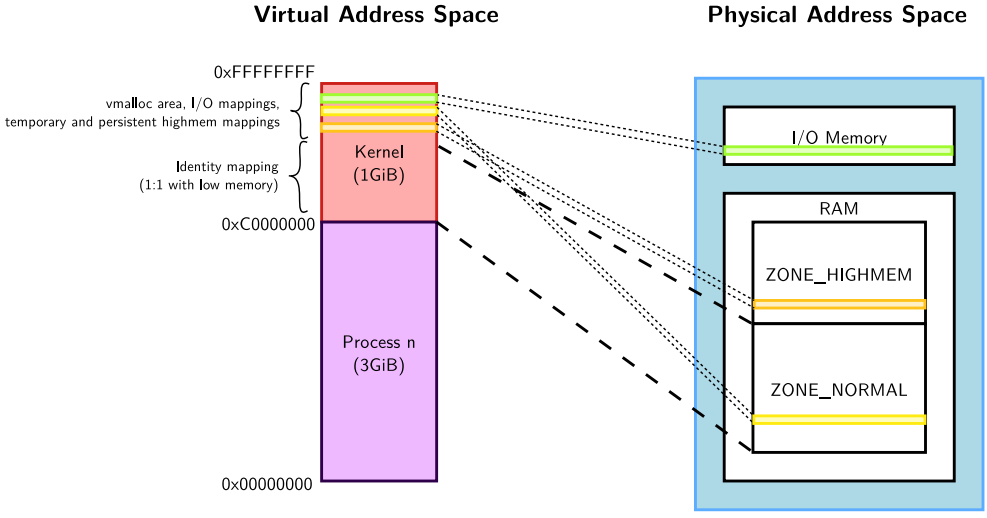


Virtual memory organization



- The top quarter reserved for kernel-space
  - Contains kernel code and core data structures
  - Allocations for loading modules
  - All kernel physical mappings
  - Identical in all address spaces
- The lower part is a per user process exclusive mapping
  - Process code and data (program, stack, ...)
  - Memory-mapped files
  - Each process has its own address space!
- The exact virtual mapping in-use is displayed in the kernel log early at boot time

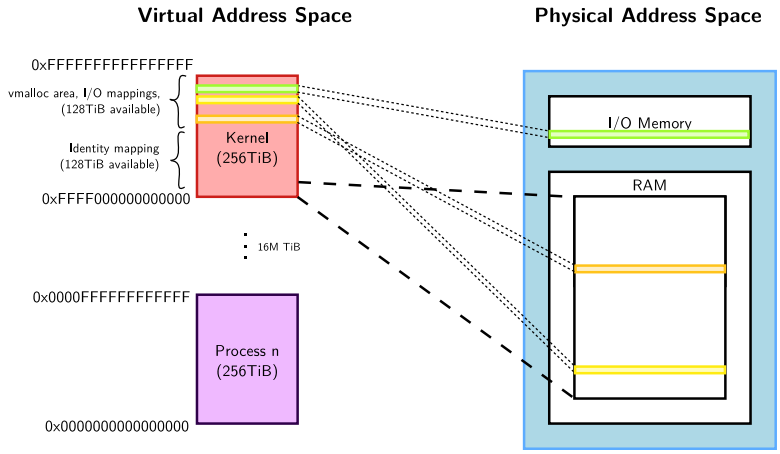
Physical/virtual memory mapping on 32-bit systems



32-bit systems limitations

- Only less than 1GB memory addressable directly through kernel virtual addresses
- 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 the 3GB/1GB memory split to 2GB/2GB or 1GB/3GB (`CONFIG_VMSPLIT_2G` or `CONFIG_VMSPLIT_1G`) $\Rightarrow$ reduce total user memory available for each process
  - Activate *highmem* support if available for your architecture:
    - Allows kernel to map parts of its non-directly accessible memory
    - Mapping must be requested explicitly
    - Limited addresses ranges reserved for this usage
- See Arnd Bergmann's *4GB by 4GB split* presentation (video and slides) at Linaro Connect virtual 2020.

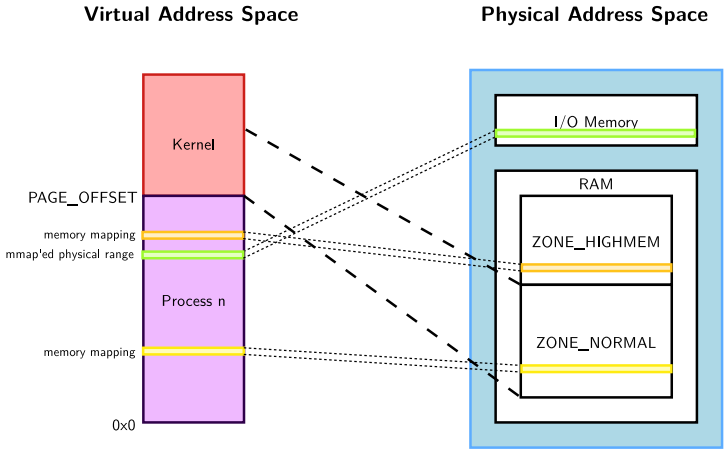
Physical/virtual memory mapping on 64-bit systems (4kiB-pages)



See also the documentation in `arch/x86/x86_64/mm`

User space virtual address space

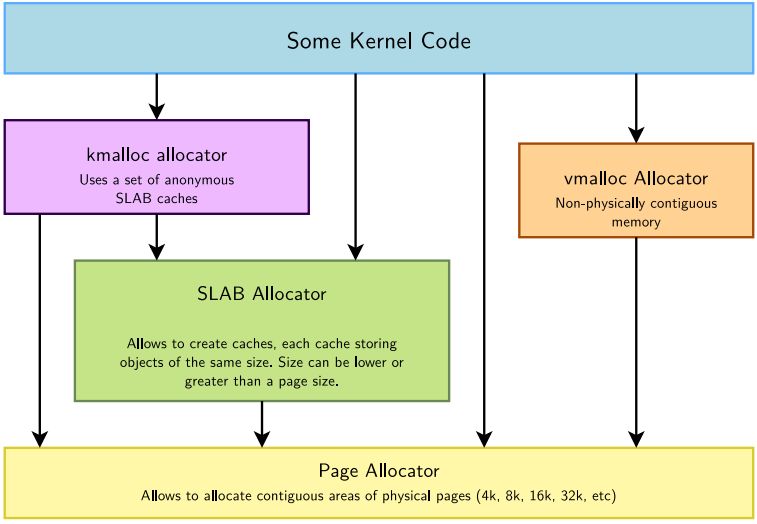
- When a process starts, the executable code is loaded in RAM and mapped into the process virtual address space.
- During execution, additional mappings can be created:
  - Memory allocations
  - Memory mapped files
  - `mmap`'ed areas
  - ...



Userspace memory allocations

- Userspace mappings can target the full memory
- When allocated, memory may not be physically allocated:
  - Kernel uses demand fault paging to allocate the physical page (the physical page is allocated when access to the virtual address generates a page fault)
  - ... or may have been swapped out, which also induces a page fault
    - See the `mlock`/`mlockall` system calls for workarounds
- User space memory allocation is allowed to over-commit memory (more than available physical memory) $\Rightarrow$ can lead to out of memory situations.
  - Can be prevented with the use of `/proc/sys/vm/overcommit_*`
- OOM killer kicks in and selects a process to kill to retrieve some memory. That's better than letting the system freeze.

Kernel memory allocators



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 contiguous in the kernel virtual address space, but also maps to physically contiguous pages. 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.
  - The *Contiguous Memory Allocator* (CMA) can be used to reserve a given amount of memory at boot (see https://lwn.net/Articles/486301/).

Page allocator API: get free pages

- **unsigned long** `get_zeroed_page(gfp_t gfp_mask)`
  - Returns the virtual address of a free page, initialized to zero
  - `gfp_mask`: see the next pages for details.
- **unsigned long** `__get_free_page(gfp_t gfp_mask)`
  - Same, but doesn't initialize the contents
- **unsigned long** `__get_free_pages(gfp_t gfp_mask, unsigned int order)`
  - Returns the starting virtual address of an area of several contiguous pages in physical RAM, with order being `log2(number_of_pages)`.Can be computed from the size with the `get_order()` function.

Page allocator API: free pages

- `void free_page(unsigned long addr)`
  - Frees one page.
- `void free_pages(unsigned long addr, unsigned int order)`
  - Frees multiple pages. Need to use the same order as in allocation.

Page allocator flags

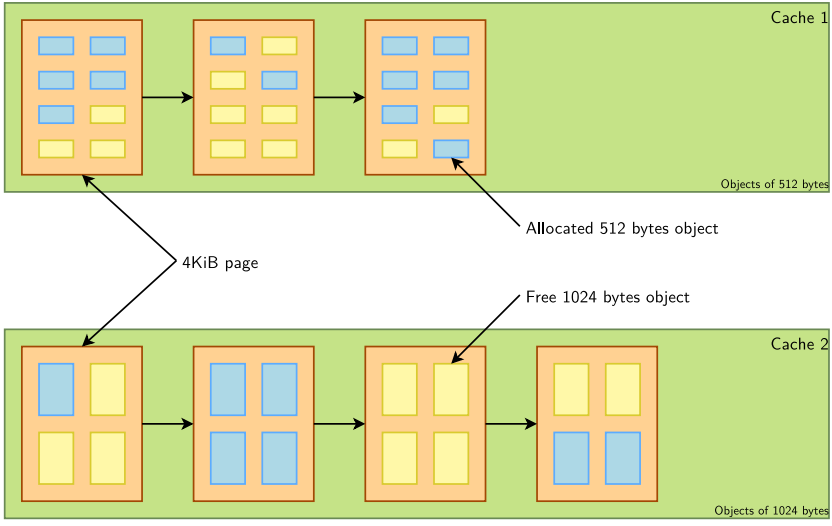
The most common ones are:

- GFP_KERNEL
  - Standard kernel memory allocation. The allocation may block in order to find enough available memory. Fine for most needs, except in interrupt handler context.
- GFP_ATOMIC
  - RAM allocated from code which is not allowed to block (interrupt handlers or critical sections). Never blocks, allows to access emergency pools, but can fail if no free memory is readily available.
- GFP_DMA
  - Allocates memory in an area of the physical memory usable for DMA transfers.
- Others are defined in include/linux/gfp_types.h.
  See also the documentation in core-api/memory-allocation

SLAB allocator 1/2

- The SLAB allocator allows to create *caches*, which contain a set of objects of the same size. In English, *slab* means *tile*.
- 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 instances in the kernel: directory entries, file objects, network packet descriptors, process descriptors, etc.
  - See `/proc/slabinfo`
- They are rarely used for individual drivers.
- See `include/linux/slab.h` for the API

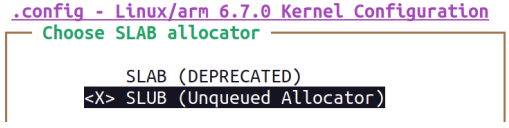
SLAB allocator 2/2



Different SLAB allocators

There are different, but API compatible, implementations of a SLAB allocator in the Linux kernel. A particular implementation is chosen at configuration time.

- `CONFIG_SLAB`: legacy but now deprecated (and removed in Linux 6.8)
- `CONFIG_SLUB`: the default allocated, scaling better and creating less fragmentation than previous implementations.
- `CONFIG_SLUB_TINY`: configure SLUB to achieve minimal memory footprint, sacrificing scalability, debugging and other features. Not recommended for systems with more than 16 MB of RAM.



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`, etc.) with the same semantics.
- Maximum sizes, on `x86` and `arm` (see `https://j.mp/YIGq6W`):
  - Per allocation: 4 MB
  - Total allocations: 128 MB
- 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, gfp_t 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_port_attr *tprops;
tprops = kmalloc(sizeof *tprops, GFP_KERNEL);
...
kfree(tprops);
```

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.

Device managed allocations

- The `probe()` function is typically responsible for allocating a significant number of resources: memory, mapping I/O registers, registering interrupt handlers, etc.
- These resource allocations have to be properly freed:
  - In the `probe()` function, in case of failure
  - In the `remove()` function
- This required a lot of failure handling code that was rarely tested
- To solve this problem, *device managed* allocations have been introduced.
- The idea is to associate resource allocation with the `struct device`, and automatically release those resources
  - When the device disappears
  - When the device is unbound from the driver
- Functions prefixed by `devm_`
- See `driver-api/driver-model/devres` for details

devm_kmalloc functions

Allocations with automatic freeing when the corresponding device or module is unprobed.

- `void *devm_kmalloc(struct device *dev, size_t size, gfp_t gfp);`
- `void *devm_kzalloc(struct device *dev, size_t size, gfp_t gfp);`
- `void *devm_kcalloc(struct device *dev, size_t n, size_t size, gfp_t flags);`
- `void *devm_kfree(struct device *dev, void *p);`
  Useful to immediately free an allocated buffer

For use in `probe()` functions, in which you have access to a `struct device` structure.

Device managed allocations: memory allocation example

- Normally done with `kmalloc(size_t, gfp_t)`, released with `kfree(void *)`
- Device managed with `devm_kmalloc(struct device *, size_t, gfp_t)`

Without devm functions

```
int foo_probe(struct platform_device *pdev)
{
        struct foo_t *foo = kmalloc(sizeof(struct foo_t),
                                    GFP_KERNEL);
        /* Register to framework, store
         * reference to framework structure in foo */
        ...
        if (failure) {
                kfree(foo);
                return -EBUSY;
        }
        platform_set_drvdata(pdev, foo);
        return 0;
}

void foo_remove(struct platform_device *pdev)
{
        struct foo_t *foo = platform_get_drvdata(pdev);
        /* Retrieve framework structure from foo
           and unregister it */
        ...
        kfree(foo);
}
```

With devm functions

```
int foo_probe(struct platform_device *pdev)
{
        struct foo_t *foo = devm_kmalloc(&pdev->dev,
                                         sizeof(struct foo_t),
                                         GFP_KERNEL);
        /* Register to framework, store
         * reference to framework structure in foo */
        ...
        if (failure)
                return -EBUSY;
        platform_set_drvdata(pdev, foo);
        return 0;
}

void foo_remove(struct platform_device *pdev)
{
        struct foo_t *foo = platform_get_drvdata(pdev);
        /* Retrieve framework structure from foo
           and unregister it */
        ...
        /* foo automatically freed */
}
```

vmalloc allocator

- The `vmalloc()` allocator can be used to obtain memory zones that are contiguous in the virtual addressing space, but not made out of physically contiguous pages.
- The requested memory size is rounded up to the next page (not efficient for small allocations).
- 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 `https://j.mp/YIGq6W` again), since physical memory fragmentation is not an issue.
- Not suitable for DMA purposes.
- API in `include/linux/vmalloc.h`
  - `void *vmalloc(unsigned long size);`
    - Returns a virtual address
  - `void vfree(void *addr);`

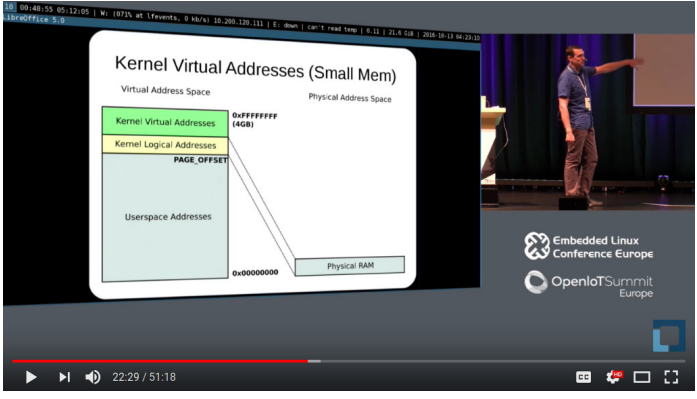
Kernel memory debugging

- KASAN (*Kernel Address Sanitizer*)
  - Dynamic memory error detector, to find use-after-free and out-of-bounds bugs.
  - Available on most architectures
  - See dev-tools/kasan for details.
- KFENCE (*Kernel Electric Fence*)
  - A low overhead alternative to KASAN, trading performance for precision. Meant to be used in production systems.
  - Available on most architectures.
  - See dev-tools/kfence for details.
- Kmemleak
  - Dynamic checker for memory leaks
  - This feature is available for all architectures.
  - See dev-tools/kmemleak for details.

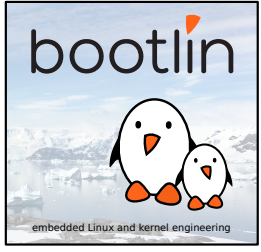
KASAN and Kmemleak have a significant overhead. Only use them in development!

Kernel memory management: resources

Virtual memory and Linux, Alan Ott and Matt Porter, 2016
Great and much more complete presentation about this topic
https://bit.ly/2Af1G2i (video: https://bit.ly/2Bwwv0C)



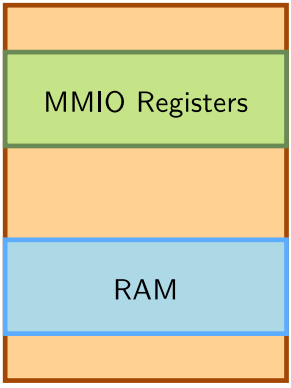
I/O Memory and Ports



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

- Memory-Mapped I/O (MMIO)
  - 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
- Port I/O (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 memory

- Tells the kernel which driver is using which I/O registers
- `struct resource *request_mem_region(unsigned long start, unsigned long len, char *name);`
- `void release_mem_region(unsigned long start, unsigned long len);`
- Allows to prevent other drivers from requesting the same I/O registers, but is purely voluntary.

/proc/iomem example - ARM 32 bit (BeagleBone Black, Linux 5.11)

```
40300000-4030ffff : 40300000.sram sram@0
44e00c00-44e00cff : 44e00c00.prm prm@c00
44e00d00-44e00dff : 44e00d00.prm prm@d00
44e00e00-44e00eff : 44e00e00.prm prm@e00
44e00f00-44e00fff : 44e00f00.prm prm@f00
44e01000-44e010ff : 44e01000.prm prm@1000
44e01100-44e011ff : 44e01100.prm prm@1100
44e01200-44e012ff : 44e01200.prm prm@1200
44e07000-44e07fff : 44e07000.gpio gpio@0
44e09000-44e0901f : serial
44e0b000-44e0bfff : 44e0b000.i2c i2c@0
44e10800-44e10a37 : pinctrl-single
44e10f90-44e10fcf : 44e10f90.dma-router dma-router@f90
48024000-48024fff : 48024000.serial serial@0
48042000-480423ff : 48042000.timer timer@0
48044000-480443ff : 48044000.timer timer@0
48046000-480463ff : 48046000.timer timer@0
48048000-480483ff : 48048000.timer timer@0
4804a000-4804a3ff : 4804a000.timer timer@0
4804c000-4804cfff : 4804c000.gpio gpio@0
48060000-48060fff : 48060000.mmc mmc@0
4819c000-4819cfff : 4819c000.i2c i2c@0
481a8000-481a8fff : 481a8000.serial serial@0
481ac000-481acfff : 481ac000.gpio gpio@0
481ae000-481aefff : 481ae000.gpio gpio@0
481d8000-481d8fff : 481d8000.mmc mmc@0
49000000-4900ffff : 49000000.dma edma3_cc
4a100000-4a1007ff : 4a100000.ethernet ethernet@0
4a101200-4a1012ff : 4a100000.ethernet ethernet@0
80000000-9fdfffff : System RAM
80008000-80cfffff : Kernel code
80e00000-80f3d807 : Kernel data
```

Mapping I/O memory in virtual memory

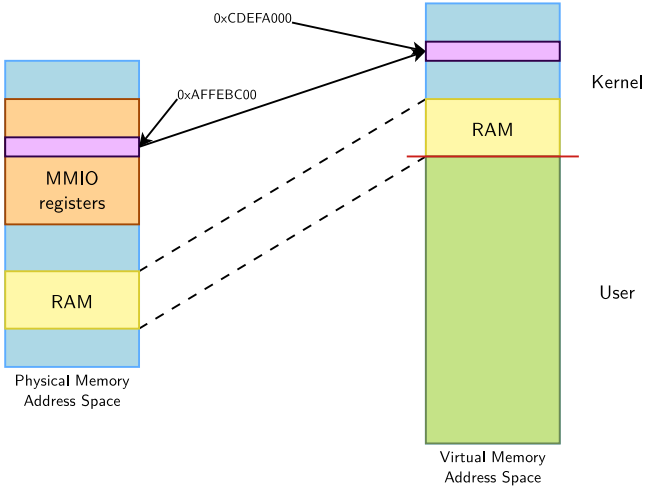
- Load/store instructions work with virtual addresses
- To access I/O memory, drivers need to have a virtual address that the processor can handle, because I/O memory is not mapped by default in virtual memory.
- The `ioremap` function satisfies this need:

```
#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(0xAFFEBC00, 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_ioremap_resource()`
  - Takes care of both the request and remapping operations!
- `devm_platform_ioremap_resource()`
  - Takes care of `platform_get_resource()`, `request_mem_region()` and `ioremap()`
  - Caution: unlike the other `devm_` functions, its first argument is of type `struct pdev`, not a pointer to `struct device`:
  - Example: `drivers/char/hw_random/st-rng.c`:

```
base = devm_platform_ioremap_resource(pdev, 0);
if (IS_ERR(base))
        return PTR_ERR(base);
```

Accessing MMIO devices: using accessor functions

- Directly reading from or writing to addresses returned by `ioremap()` (*pointer dereferencing*) may not work on some architectures.
- A family of architecture-independent accessor functions are available covering most needs.
- A few architecture-specific accessor functions also exists.

MMIO access functions

- `read[b/w/l/q]` and `write[b/w/l/q]` for access to little-endian devices, includes memory barriers
- `ioread[8/16/32/64]` and `iowrite[8/16/32/64]` are very similar to read/write but also work with port I/O (not covered in the course), includes memory barriers
- `ioread[8/16/32/64]be` and `iowrite[8/16/32/64]be` for access to big-endian devices, includes memory barriers
- `__raw_read[b/w/l/q]` and `__raw_write[b/w/l/q]` for raw access: no endianness conversion, no memory barriers
- `read[b/w/l/q]_relaxed` and `write[b/q/l/w]_relaxed` for access to little-endian devices, without memory barriers
- All functions work on a `void __iomem *`

MMIO access functions summary

Name	Device endianness	Memory barriers
`read/write`	little	yes
`ioread/iowrite`	little	yes
`ioreadbe/iowritebe`	big	yes
`__raw_read/__raw_write`	native	no
`read_relaxed/write_relaxed`	little	no

More details at https://docs.kernel.org/driver-api/device-io.html

Ordering

- Reads/writes to MMIO-mapped registers of given device are done in program order
- However reads/writes to RAM can be re-ordered between themselves, and between MMIO-mapped read/writes
- Some of the accessor functions include memory barriers to help with this:
  - Write operation starts with a write memory barrier which prior writes cannot cross
  - Read operation ends with a read memory barrier which guarantees the ordering with regard to the subsequent reads
- Sometimes compiler/CPU reordering is not an issue, in this case the code may be optimized by dropping the memory barriers, using the raw or relaxed helpers
- You can also add a memory barrier by hand if needed:
  - `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
- See `Documentation/memory-barriers.txt`

/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`, `riscv`, `powerpc`, `parisc`, `s390`: `CONFIG_STRICT_DEVMEM` option to restrict `/dev/mem` to non-RAM addresses, for security reasons (Linux 5.12 status). `CONFIG_IO_STRICT_DEVMEM` goes beyond and only allows to access *idle* I/O ranges (not appearing in `/proc/iomem`).