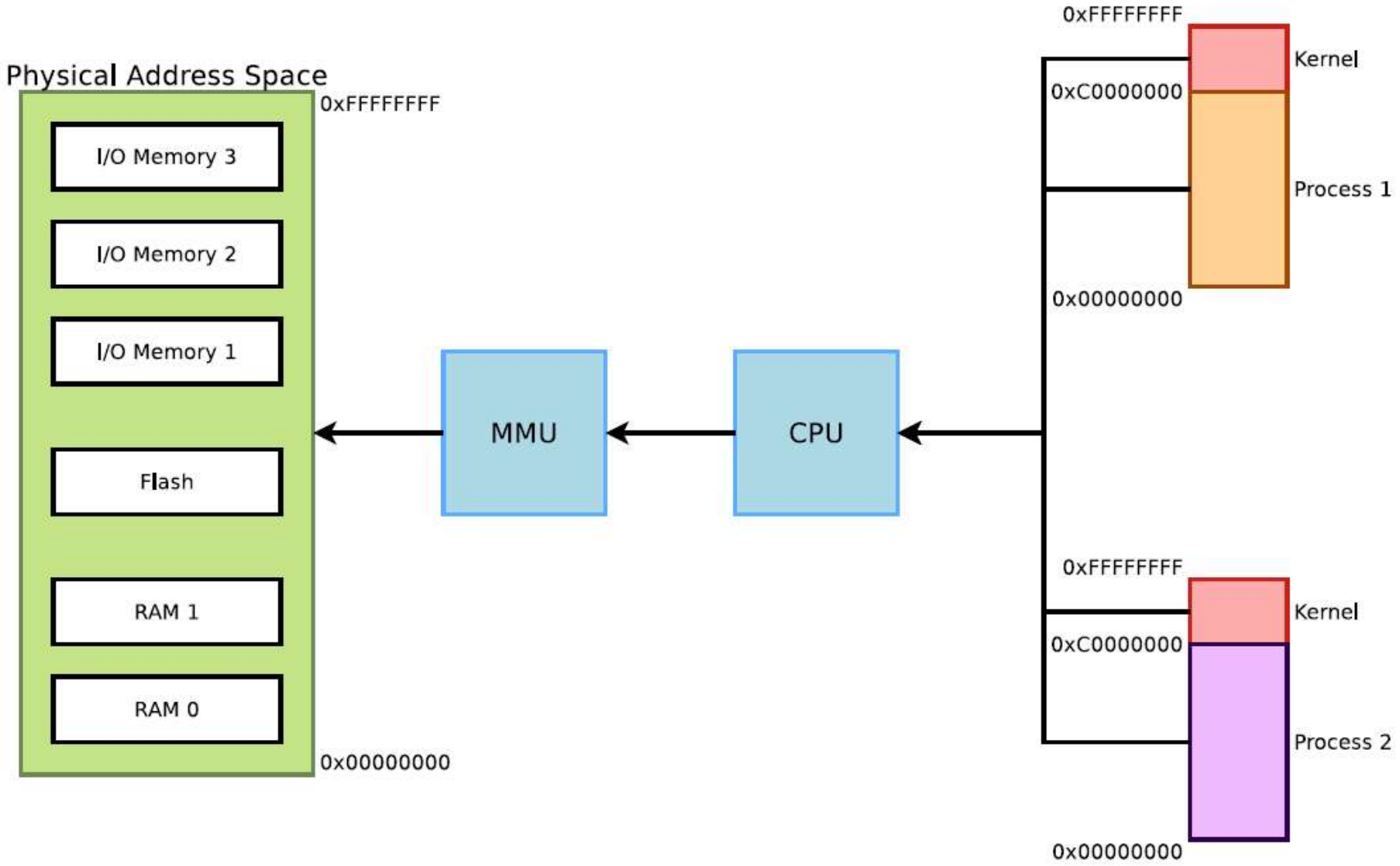
Rights to copy

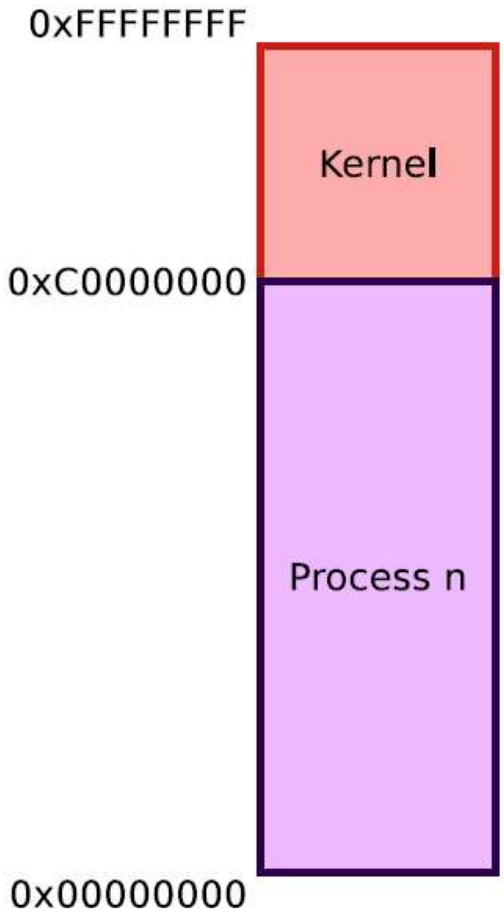
7. Memory Management

Physical and Virtual Memory

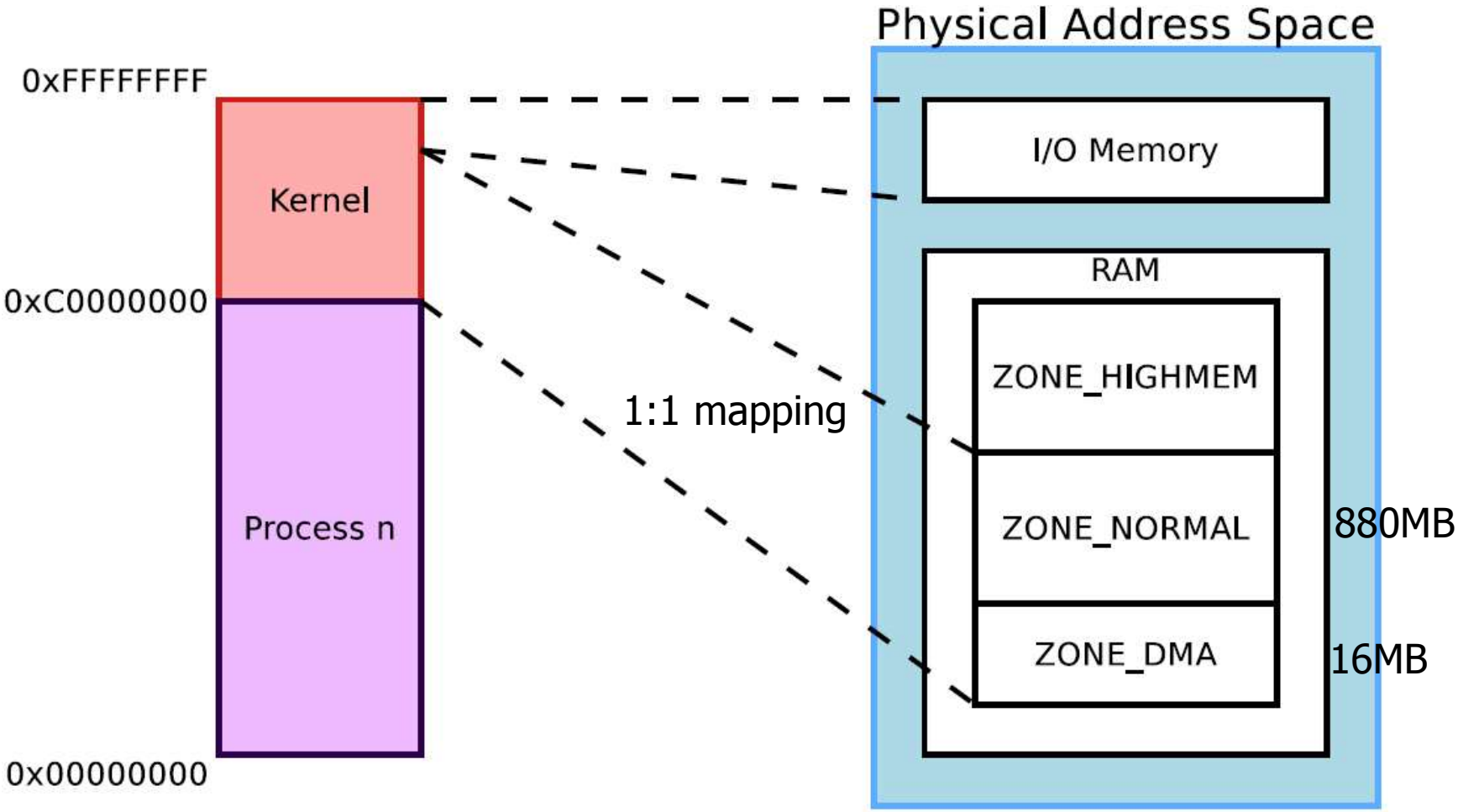


Virtual Memory Organization

- 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

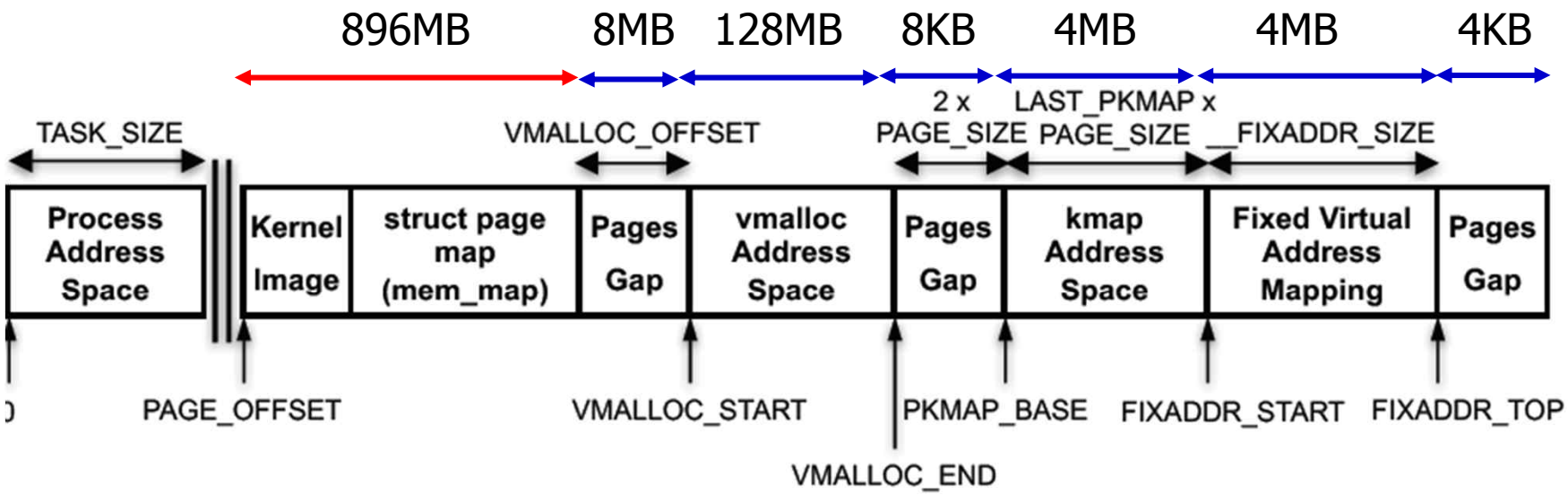


Many old devices (ISA bus in particular) could only use 24 bits for DMA addresses, and were thus limited to the bottom 16MB of memory

Accessing more physical memory

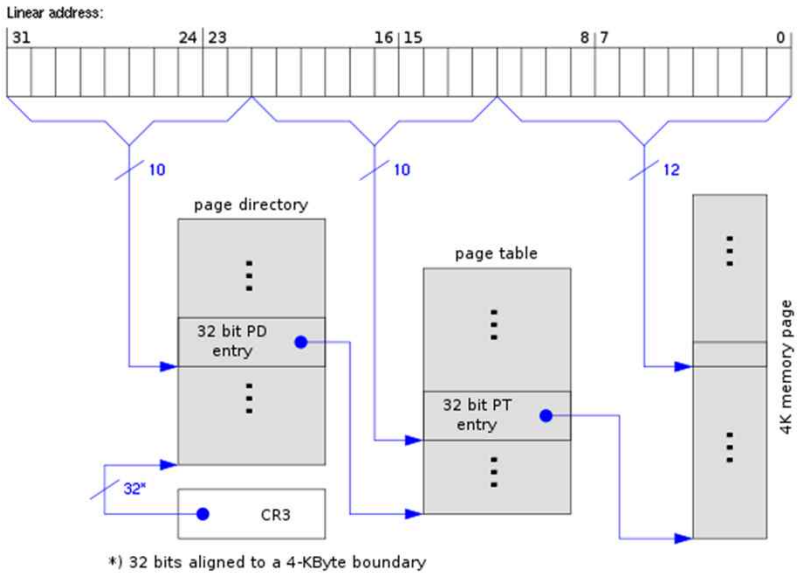
- Only less than 1GB memory addressable directly through kernel virtual address space
- If more physical memory is present on the platform, part of the memory will not be accessible by kernel space, but can be used by user-space
- To allow the kernel to access more physical memory:
  - Change 1GB/3GB memory split (2GB/2GB) (CONFIG_VMSPLIT_3G ➔ CONFIG_VMSPLIT_2G)
    - ➔ reduces total memory available for each process
  - Change for a 64 bit architecture
  - Activate highmem support if available for your architecture:
    - Allows kernel to map parts of its non-directly accessible memory (kmap, vmap, fixmap)
    - Mapping must be requested explicitly
    - Limited addresses ranges reserved for this usage

Kernel Address Space



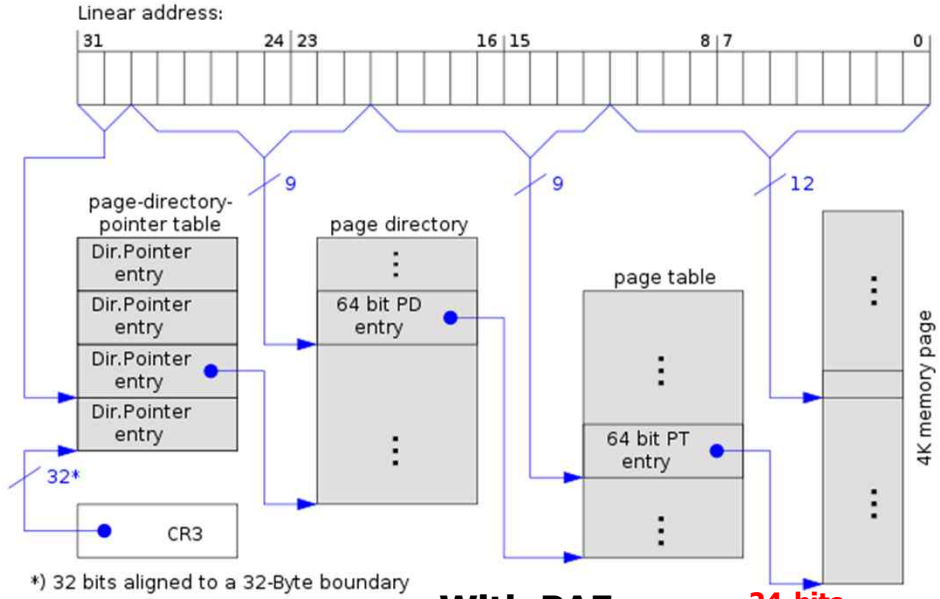
Accessing even more physical memory!

- If your 32 bit platform hosts more than 4GB, they just cannot be mapped
- IA's PAE (Physical Address Extension), ARM Cortex-A15 LPAE allows to access a physical address space larger than 4GB.
- Adds some address extension bits used to index memory areas
- Allows accessing up to 64 GB of physical memory on x86
- Note that each user-space process is still limited to a 3 GB memory space



No PAE

$2^{10} * 2^{10} * 2^{12}$ bytes = 4 GB



With PAE

$2^{24} * 2^{12}$ bytes = 64 GB

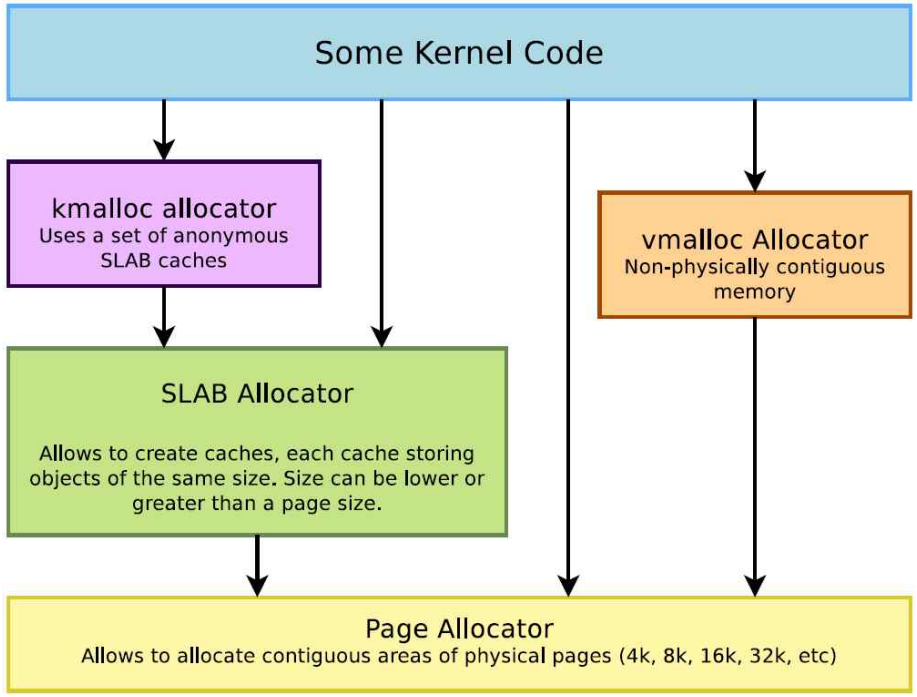
User-Space Memory

- New user-space memory is allocated either from the already allocated process memory, or using the mmap system call
- Note that memory allocated 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
- User space memory allocation is allowed to over-commit memory (more than available physical memory)

  ➔ can lead to out of memory
- OOM killer kicks in and selects a process to kill to retrieve some memory.
  - better than letting the system freeze.

Kernel Memory

- Kernel memory allocators allocate physical pages, and kernel allocated memory cannot be swapped out ➔ no fault handling required
- 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.
  - The finest granularity (usually 4 KB, architecture dependent).
- However, the kernel memory management handles smaller memory allocations through its allocator (SLAB allocators used by kmalloc()).



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
  - 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 (2^{11} pages), depend on the kernel configuration.
- The allocated area is virtually contiguous, but also physically contiguous.
  - large areas may not be available or hard to retrieve due to physical memory fragmentation.
  - See /proc/buddyinfo

Page Allocator APIs

- unsigned long get_zeroed_page(int flags)
  - Returns the virtual address of a free page, initialized to zero
- unsigned long __get_free_page(int flags)
  - Same, but doesn't initialize the contents
- unsigned long __get_free_pages(int flags, unsigned int order)
  - Returns 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

- 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

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.

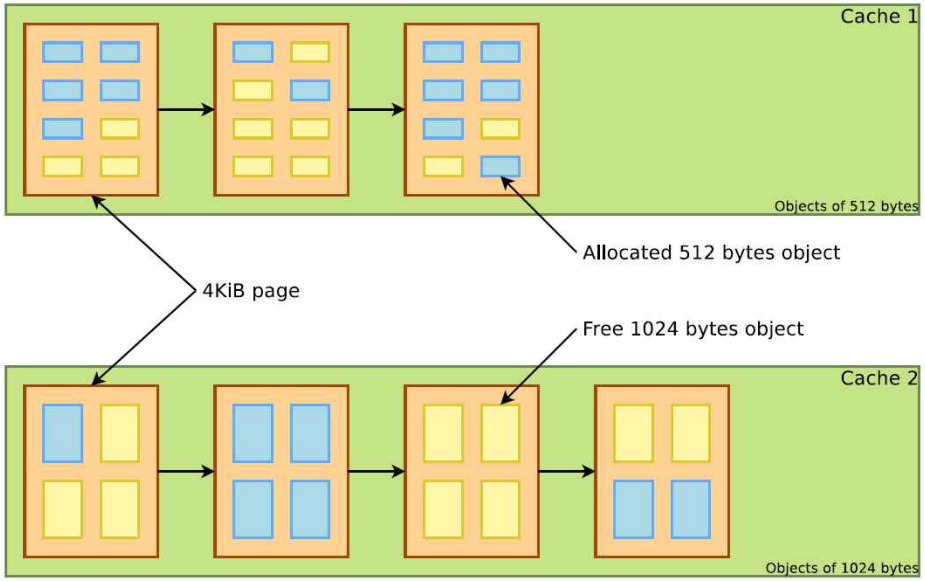
GFP_ATOMIC
RAM allocated from code which is not allowed to block (interrupt handlers or critical sections). Never blocks, allows to access emergency pools, but can fail if no free memory is readily available.

GFP_DMA
Allocates memory in an area of the physical memory usable for DMA transfers.

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 instances in the kernel
  - directory entries, file objects, network packet descriptors, process descriptors, etc.
  - rarely used for individual drivers.
- See /proc/slabinfo
- See include/linux/slab.h for the API

SLAB Allocator




```
% cat /proc/slabinfo
slabinfo - version: 1.1
kmem_cache            60      78    100    2    2    1
blkdev_requests     5120    5120     96  128  128    1
mnt_cache             20      40     96    1    1    1
inode_cache         7005   14792    480 1598 1849    1
dentry_cache        5469    5880    128  183  196    1
filp                 726     760     96   19   19    1
buffer_head        67131   71240     96 1776 1781    1
vm_area_struct      1204    1652     64   23   28    1
...
size-8192              1      17   8192    1   17    2
size-4096             41      73   4096   41   73    1
...
```

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.
  - Linux 4.20 on ARM: used in 48 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 4.20 on ARM: used in 7 defconfig files
- SLUB: more recent and simpler than SLAB, scaling much better (in particular for huge systems) and creating less fragmentation.
  - Linux 4.20 on ARM: used in 0 defconfig files

```
defconfig
# CONFIG_SLAB is not set
CONFIG_SLUB=y
# CONFIG_SLOB is not set
```

kmalloc Allocator

- The kmalloc allocator is the general purpose memory allocator in the Linux kernel
- For small sizes, it relies on generic SLAB caches
- 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.
- Maximum sizes, on x86 and arm
  - Per allocation: 4 MB
  - Total allocations: 128 MB

kmalloc APIs

- #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
- 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.

Example: (drivers/infiniband/core/cache.c)

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

devm_kmalloc functions

- Automatically free the allocated buffers when the corresponding device or module is unprobed.
  - void *devm_kmalloc(struct device *dev, size_t size, int flags);
  - void *devm_kzalloc(struct device *dev, size_t size, int flags);
  - 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.

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), 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

- KASAN https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html
  - Dynamic memory error detector, to find use-after-free and out-of-bounds bugs.
  - Only available on x86_64, arm64, s390 and xtensa so far (Linux 4.20 status), but will help to improve architecture independent code anyway.
  - See dev-tools/kasan for details.
- Kmemleak
  - Dynamic checker for memory leaks
  - This feature is available for all architectures.
  - CONFIG_DEBUG_KMEMLEAK

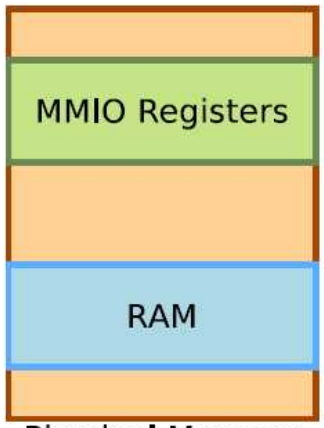
    # mount -t debugfs nodev /sys/kernel/debug/

    # cat /sys/kernel/debug/kmemleak
  - See dev-tools/kmemleak for details.
- Both have a significant overhead. Only use them in development!

I/O Memory and Ports

Port I/O vs. 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
- 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



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 - ARM (Raspberry Pi, Linux 4.14)

```
00000000-3b3fffff : System RAM
00008000-00affffff : Kernel code
00c00000-00d468af : Kernel data
3f006000-3f006fff : dwc_otg
3f007000-3f007eff : /soc/dma@7e007000
3f00b840-3f00b84e : /soc/vchiq
3f00b880-3f00b8bf : /soc/mailbox@7e00b880
3f100000-3f100027 : /soc/watchdog@7e100000
3f101000-3f102fff : /soc/cprman@7e101000
3f200000-3f2000b3 : /soc/gpio@7e200000
3f201000-3f201fff : /soc/serial@7e201000
3f201000-3f201fff : /soc/serial@7e201000
3f202000-3f2020ff : /soc/mmc@7e202000
3f212000-3f212007 : /soc/thermal@7e212000
3f215000-3f215007 : /soc/aux@0x7e215000
3f980000-3f98ffff : dwc_otg
```

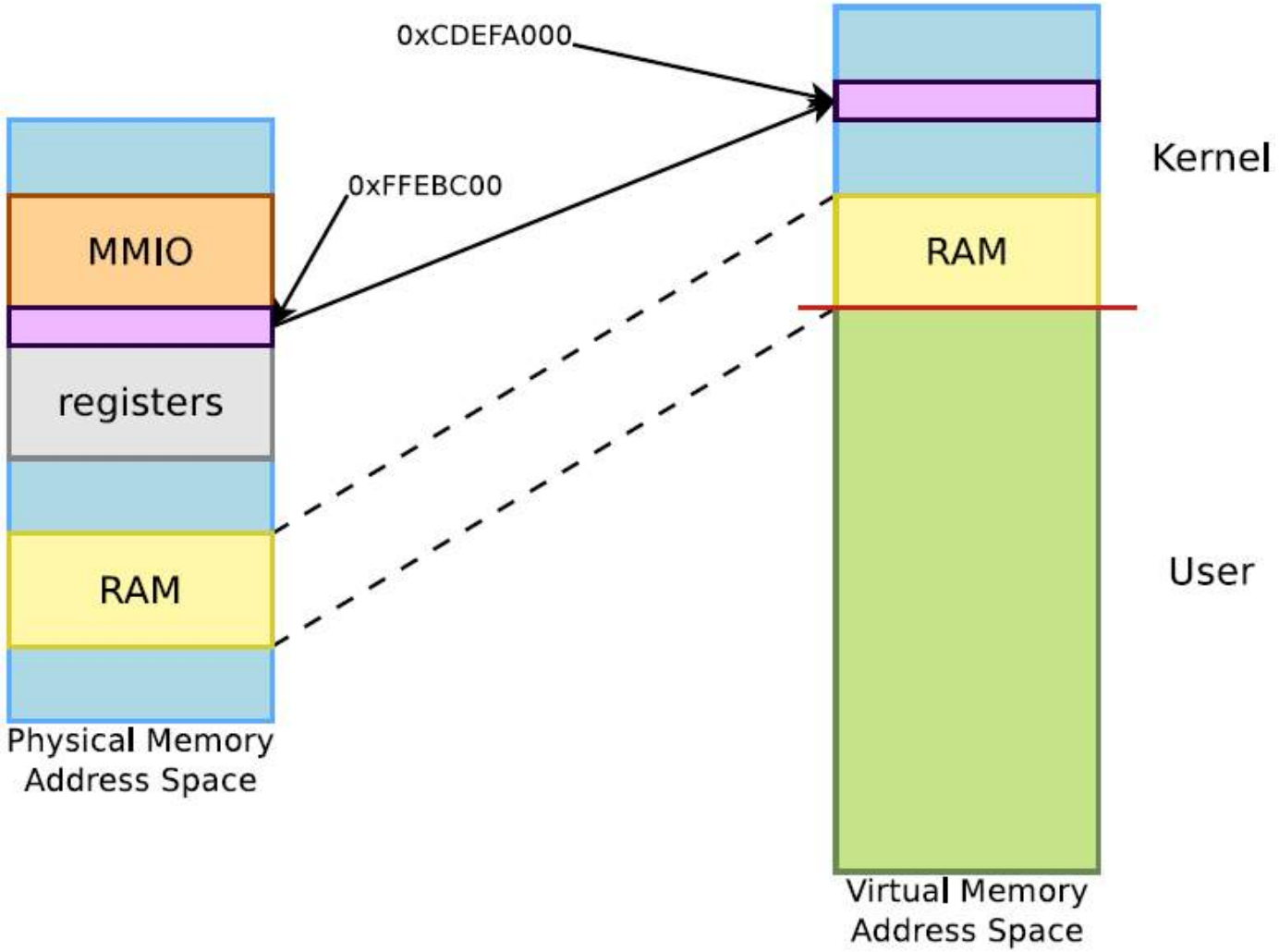
Mapping I/O memory in virtual memory

- 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.
- Useful when physical memory or I/O Address is larger than virtual address space size.(0xffffffff)
- The ioremap function satisfies this need:
- #include <asm/io.h>

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

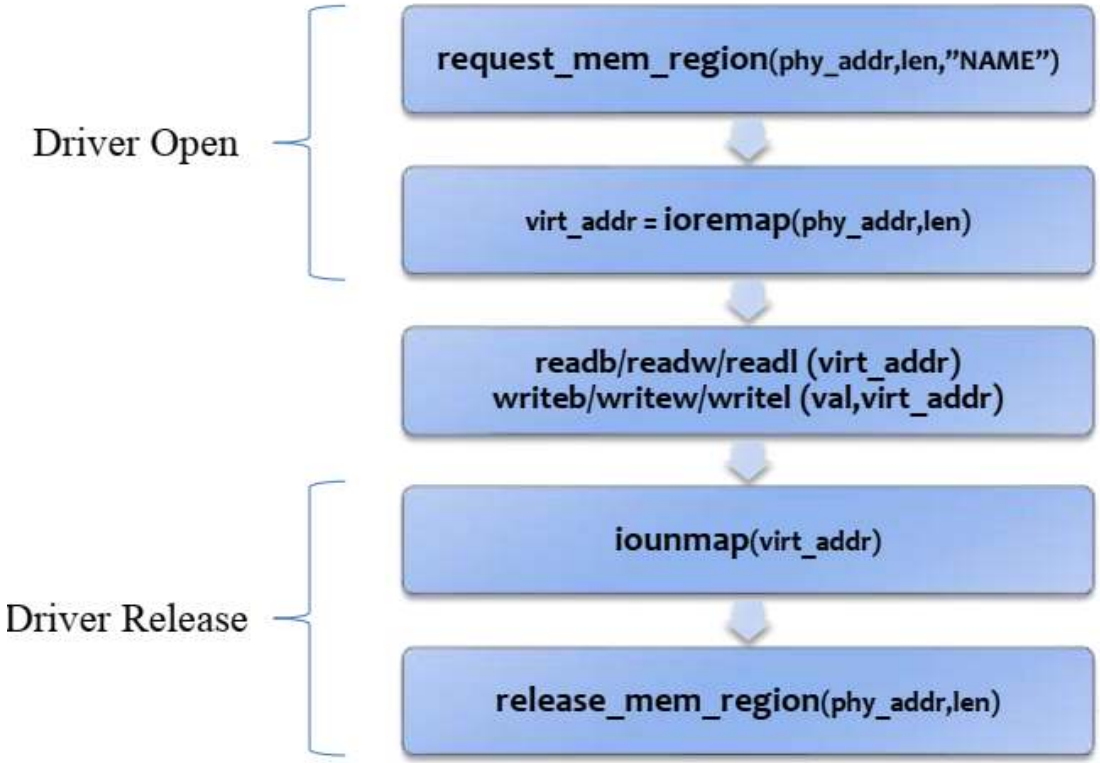
  void iounmap(void __iomem *addr);
- Caution: check that ioremap() doesn't return a NULL address!

ioremap()




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

Flow of I/O Memory Map Access




```
readb(addr);
readw(addr);
readl(addr);
writeb(val,addr);
writew(val,addr);
writel(val,addr);
memcpy_fromio(buffer,addr, len);
memcpy_toio(addr,buffer,len);
memset_io(addr,val,len);
```

You should not directly access addresses returned by ioremap as if they were pointer to virtual memory address.
The read and write functions are defined to be ordered.
Guarantee read/write ordering

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!

Avoiding I/O access issues

- Caching on I/O ports or memory already disabled
  - non-cacheable, non-bufferable
  - volatile
- 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.

```
while (count--) {
    writeb(*(ptr++), address);
    wmb();
}
```

/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, powerpc, s390 and unicore32: CONFIG_STRICT_DEVMEM option to restrict /dev/mem to non-RAM addresses, for security reasons (Linux 4.20 status).

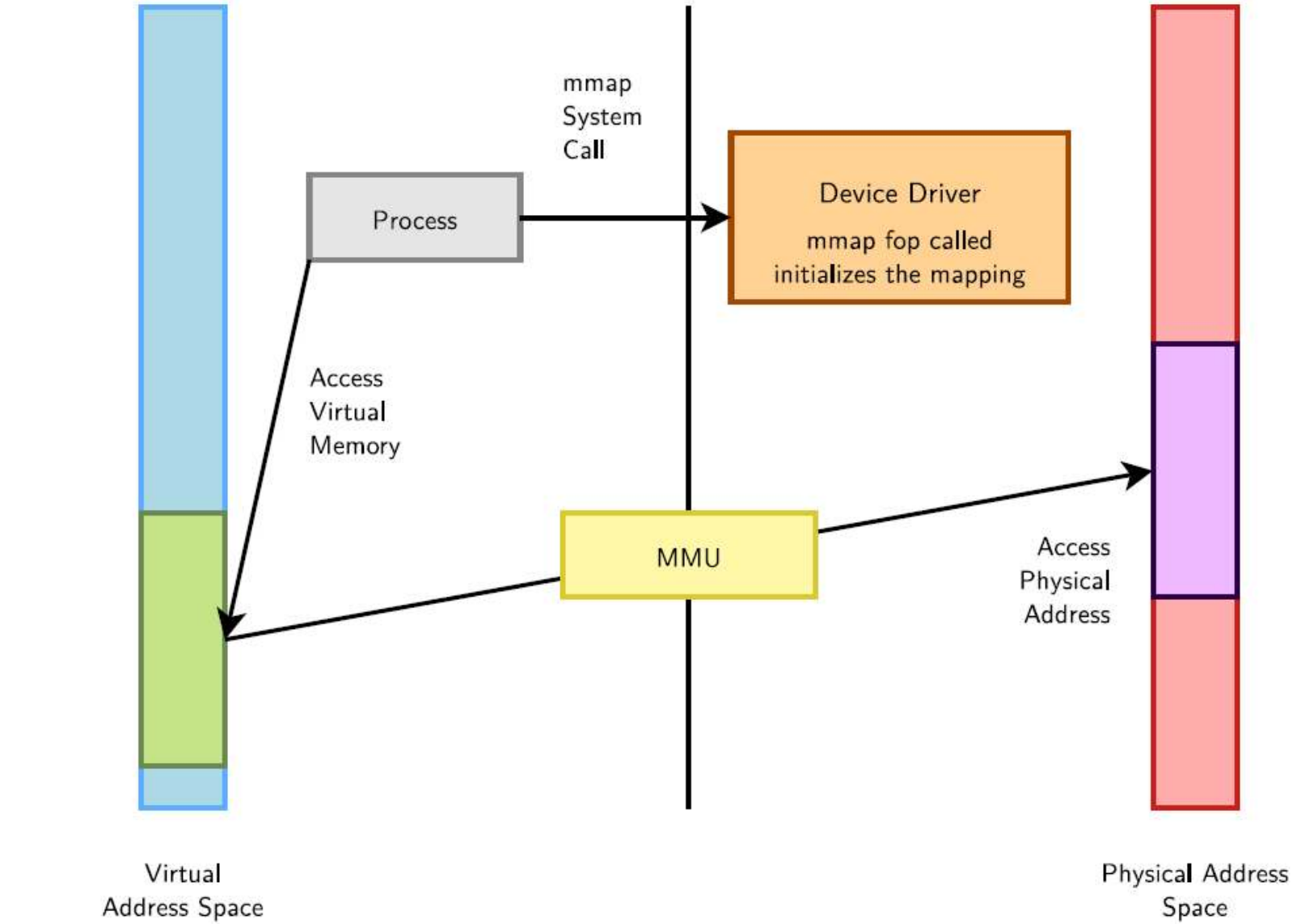
mmap

- Possibility to have parts of the virtual address space of a program mapped to the contents of a file
- Particularly useful when the file is a device file
- Allows to access device I/O memory and ports without having to go through (expensive) read, write or ioctl calls
- One can access to current mapped files by two means:
  - /proc/<pid>/maps
  - pmap <pid>

/proc/<pid>/maps

```
       start-end          perm offset major:minor inode  mapped file name
...
7f4516d04000-7f4516d06000 rw-s 1152a2000 00:05 8406   /dev/dri/card0
7f4516d07000-7f4516d0b000 rw-s 120f9e000 00:05 8406   /dev/dri/card0
...
7f4518728000-7f451874f000 r-xp 00000000 08:01 268909  /lib/x86_64-linux-gnu/libexpat.so.1.5.2
7f451874f000-7f451894f000 ---p 00027000 08:01 268909  /lib/x86_64-linux-gnu/libexpat.so.1.5.2
7f451894f000-7f4518951000 r--p 00027000 08:01 268909  /lib/x86_64-linux-gnu/libexpat.so.1.5.2
7f4518951000-7f4518952000 rw-p 00029000 08:01 268909  /lib/x86_64-linux-gnu/libexpat.so.1.5.2
...
7f451da4f000-7f451dc3f000 r-xp 00000000 08:01 1549    /usr/bin/Xorg
7f451de3e000-7f451de41000 r--p 001ef000 08:01 1549    /usr/bin/Xorg
7f451de41000-7f451de4c000 rw-p 001f2000 08:01 1549    /usr/bin/Xorg
...
```

mmap Overview



How to Implement mmap - User Space

- Open the device file
- Call the mmap system call (see man mmap for details):

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

- You get a virtual address you can write to or read from.
- prot: memory protection
  - PROT_EXEC: Pages may be executed.
  - PROT_READ: Pages may be read.
  - PROT_WRITE: Pages may be written.
  - PROT_NONE: Pages may not be accessed.
- flag: MAP_SHARED, MAP_PRIVATE, MAP_LOCKED, …

mmap() - Example

```
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
int main( int argc,char **argv ) {
        int fd ;
        int filesize= getpagesize();
        void *virt_addr;
        if ((fd = open( “test.bin”, O_RDONLY)) < 0)
                perror("open error”) ;
        virt_addr = mmap(0, filesize, PROT_READ,
                                        MAP_SHARED | MAP_LOCKED, fd , 0) ;
        if (virt_addr == MAP_FAILED) perror("mmap error”) ;
        *(unsigned long*)virt_addr = 0x12345678;
        msync(virt_addr,filesize,MS_SYNC)
        munmap(virt_addr,filesize)
}
```

How to Implement mmap - Kernel Space

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

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

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

remap_pfn_range()

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

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

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

Simple mmap implementation

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

      if (size > ACME_SIZE)
          return -EINVAL;

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

      return 0;
    }
```

devmem2

- https://bootlin.com/pub/mirror/devmem2.c , by Jan-Derk Bakker
- Very useful tool to directly peek (read) or poke (write) I/O addresses mapped in physical address space from a shell command line!
  - Useful when debugging embedded boards
  - Very useful for early interaction experiments with a device, without having to code and compile a driver.
  - Uses mmap to /dev/mem.
  - Examples (b: byte, h: half, w: word)
    - devmem2 0x000c0004 h (reading)
    - devmem2 0x000c0008 w 0xffffffff (writing)
  - devmem is now available in BusyBox, making it even easier to use.

mmap Summary

- The device driver is loaded. It defines an mmap file operation.
- A user space process calls the mmap system call.
- The mmap file operation is called.
- It initializes the mapping using the device physical address.
- The process gets a starting address to read from and write to (depending on permissions).
- The MMU automatically takes care of converting the process virtual addresses into physical ones.
- Direct access to the hardware without any expensive read or write system calls