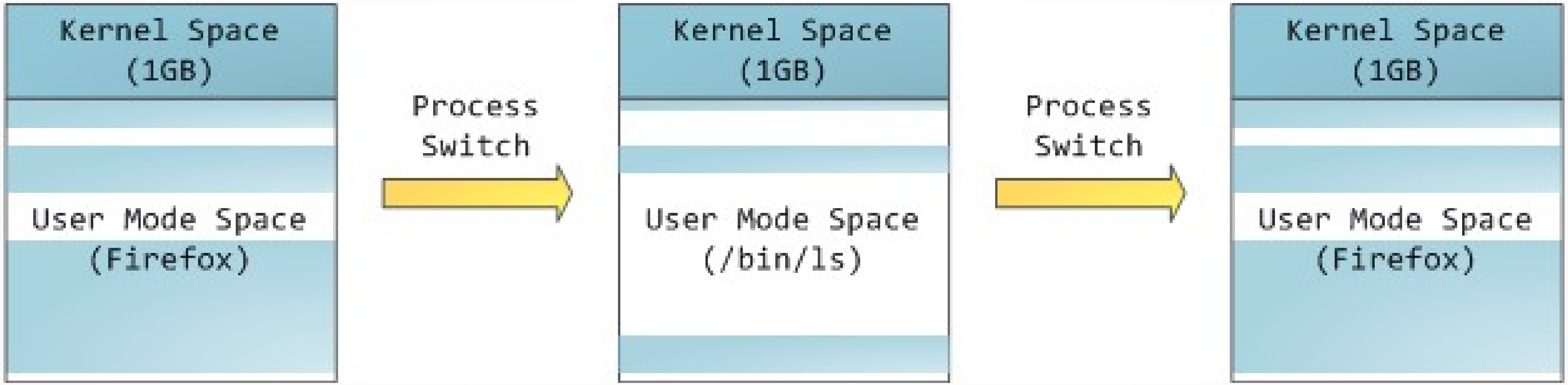
Memory management in Linux

Stefan Strogin, 2018-2019

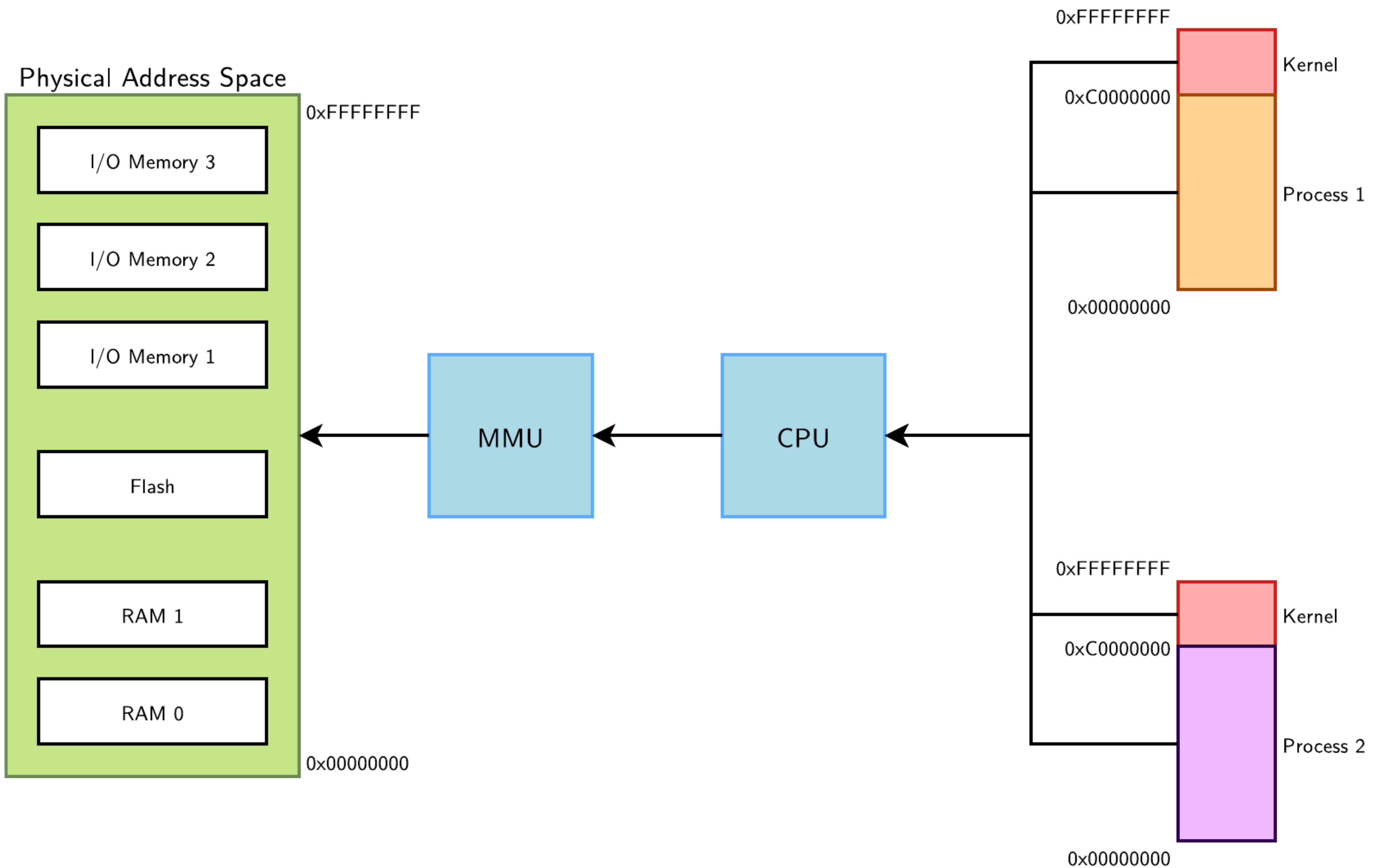
Memory management tasks

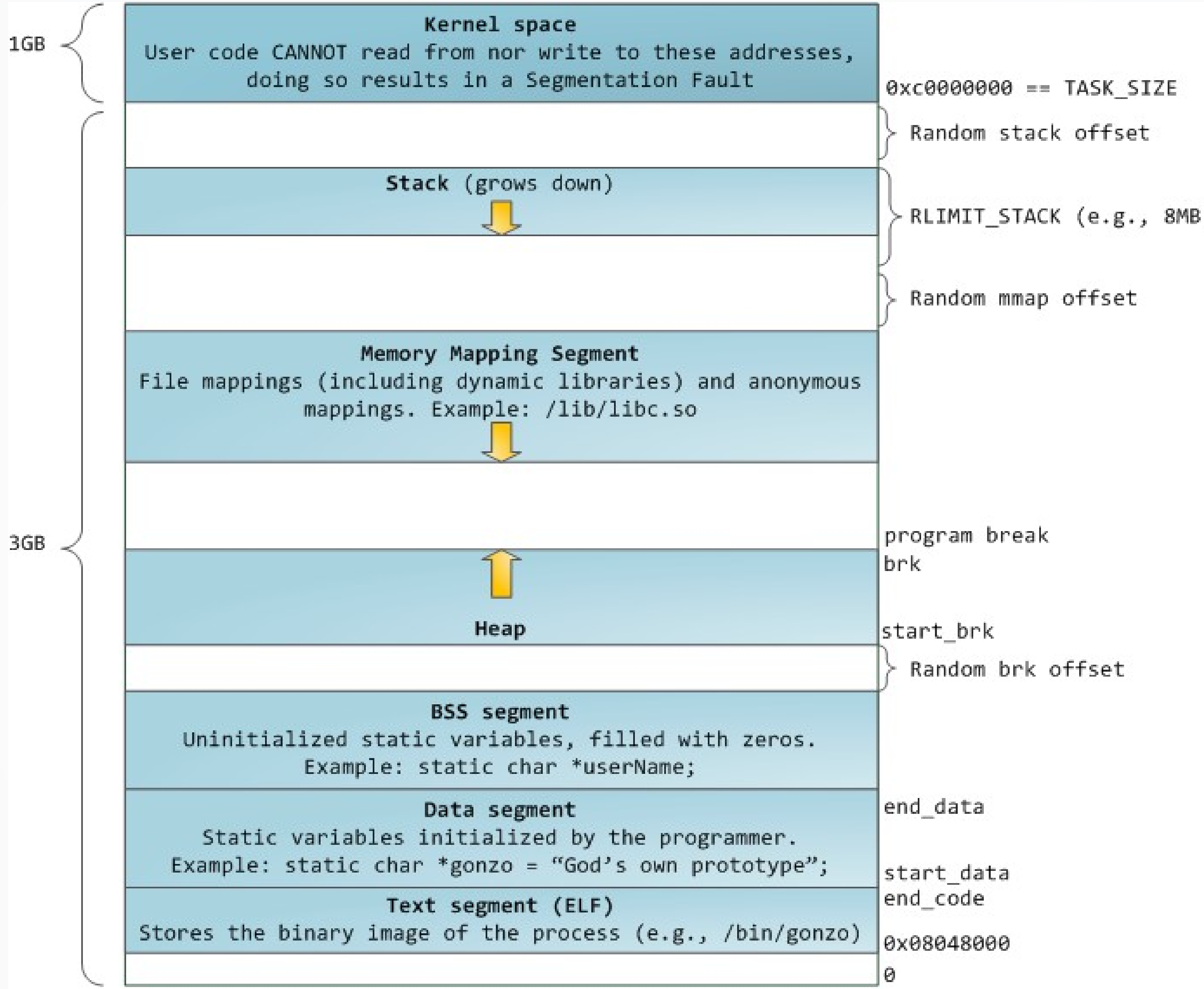
- Physical memory management
- Virtual memory allocation
- Physical / virtual memory mapping, PTE management
- Memory allocation for kernel needs

Virtual memory organization



- Kernel code and data are always addressable, ready to handle interrupts or system calls at any time
- Mapping for the user-mode portion of the address space changes whenever a process switch happens


Physical Address Space
0xFFFFFFFF
I/O Memory 3
I/O Memory 2
I/O Memory 1
Flash
RAM 1
RAM 0
0x00000000
MMU
CPU
0xFFFFFFFF
Kernel
0xC0000000
Process 1
0x00000000
0xFFFFFFFF
Kernel
0xC0000000
Process 2
0x00000000



1GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
3GB
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
end_data
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = “God’s own prototype”;
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

Stack

- If a process pushes data beyond stack size, the CPU will trigger a page fault.
- The page fault handler detects the address is just beyond the stack, and allocates a new page to extend the stack.
- See __do_page_fault() in arch/{x86,arm,arm64...}/mm/fault.c, then expand_stack() and acct_stack_growth()
- If the stack size is more than RLIMIT_STACK, a SIGSEGV signal is generated (segmentation fault).

mmap()

- mmap() is the standard way to allocate large amounts of memory from user space.
- Can map contents of files directly to memory, which is used for example for loading dynamic libraries.
- MAP_ANONYMOUS flag causes mmap() to allocate normal memory for the process. The MAP_SHARED flag can make the allocated pages shareable with other processes.
- If you request more than M_MMAP_THRESHOLD (128 kB by default, adjustable via mallopt()) via malloc(), anonymous mapping is used instead of heap memory.

Heap

- Allocating from heap is the standard way to allocate small amounts of memory from user space: malloc() and friends in C, ‘new’ keyword in C++, etc.
- If there is enough space in the heap to satisfy a memory request, it can be handled by the language runtime without kernel involvement. Otherwise the heap is enlarged via sbrk()/brk() (see do_brk() in mm/mmap.c).
- Large allocations use mmap() instead of brk().
- May become fragmented eventually:

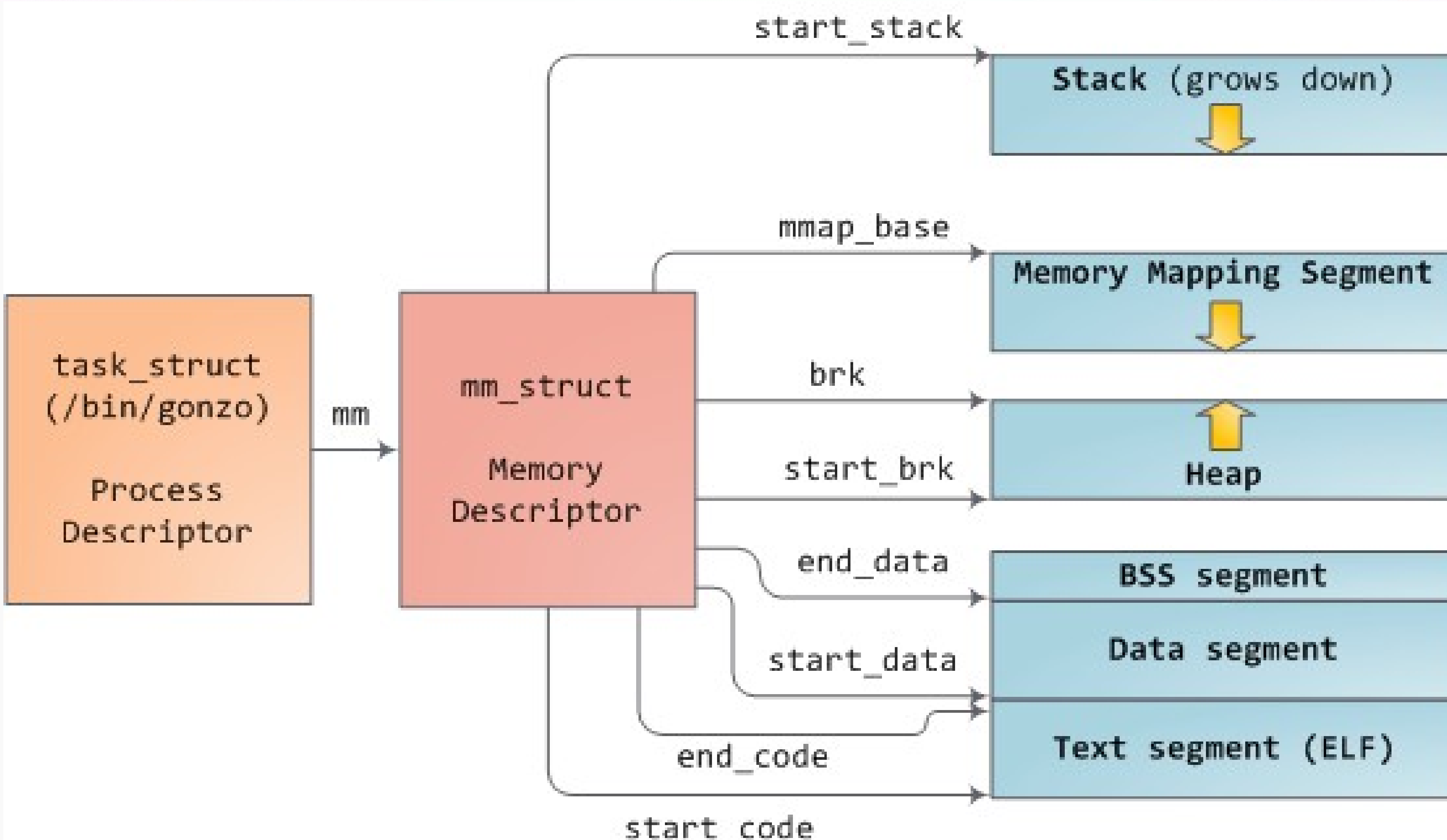
/proc/<pid>/maps

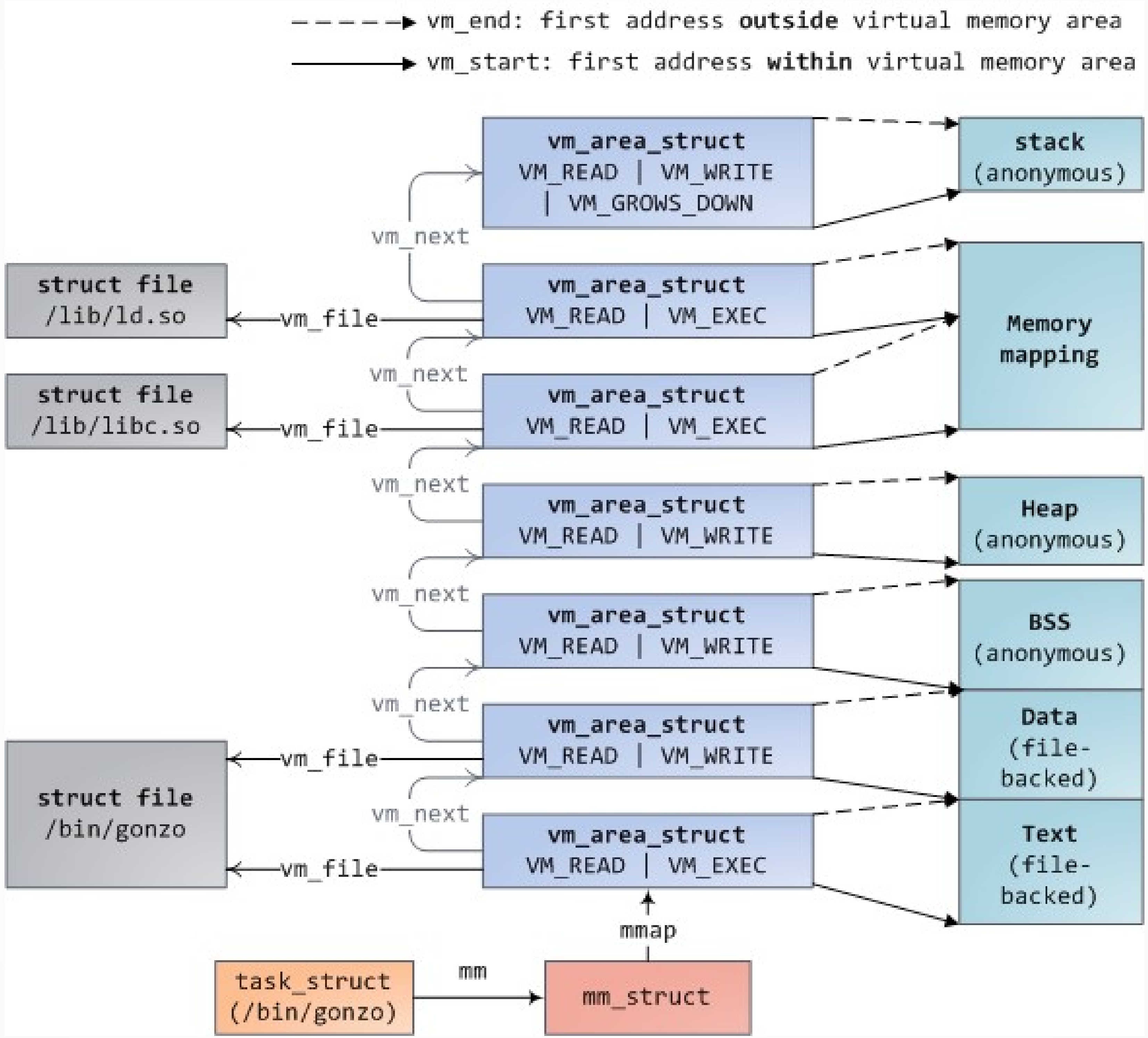
- The /proc/PID/maps file containing the currently mapped memory regions and their access permissions.
  The format is:
  address perms offset dev inode pathname

```
~/ cat /proc/self/maps
55de9ca91000-55de9ca99000 r-xp 00000000 fc:00 23855162                   /bin/cat
55de9cc98000-55de9cc99000 r--p 00007000 fc:00 23855162                   /bin/cat
55de9cc99000-55de9cc9a000 rw-p 00008000 fc:00 23855162                   /bin/cat
55de9df89000-55de9dfaa000 rw-p 00000000 00:00 0                          [heap]
7f32494c1000-7f32497da000 r--p 00000000 fc:00 5902488                    /usr/lib64/locale/locale-archive
7f32497da000-7f3249993000 r-xp 00000000 fc:00 25668388                   /lib64/libc-2.26.so
7f3249993000-7f3249b92000 ---p 001b9000 fc:00 25668388                   /lib64/libc-2.26.so
7f3249b92000-7f3249b96000 r--p 001b8000 fc:00 25668388                   /lib64/libc-2.26.so
7f3249b96000-7f3249b98000 rw-p 001bc000 fc:00 25668388                   /lib64/libc-2.26.so
7f3249b98000-7f3249b9c000 rw-p 00000000 00:00 0
7f3249b9c000-7f3249bc1000 r-xp 00000000 fc:00 25669421                   /lib64/ld-2.26.so
7f3249d81000-7f3249d83000 rw-p 00000000 00:00 0
7f3249d9e000-7f3249dc0000 rw-p 00000000 00:00 0
7f3249dc0000-7f3249dc1000 r--p 00024000 fc:00 25669421                   /lib64/ld-2.26.so
7f3249dc1000-7f3249dc2000 rw-p 00025000 fc:00 25669421                   /lib64/ld-2.26.so
7f3249dc2000-7f3249dc3000 rw-p 00000000 00:00 0
7ffe92c6e000-7ffe92c8f000 rw-p 00000000 00:00 0                          [stack]
7ffe92cf3000-7ffe92cf6000 r--p 00000000 00:00 0                          [vvar]
7ffe92cf6000-7ffe92cf8000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

See https://www.kernel.org/doc/Documentation/filesystems/proc.txt

Managing virtual memory. mm_struct

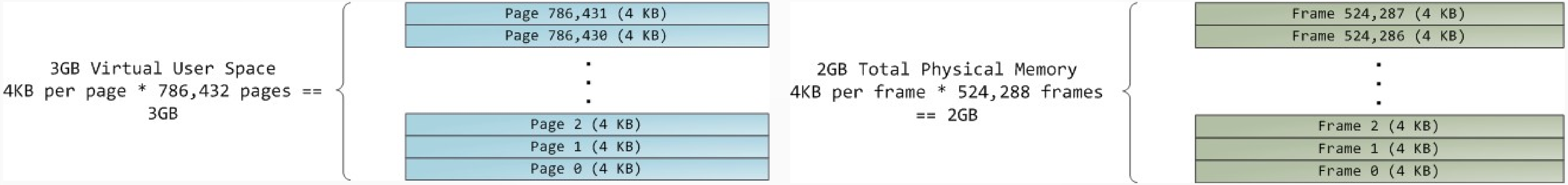




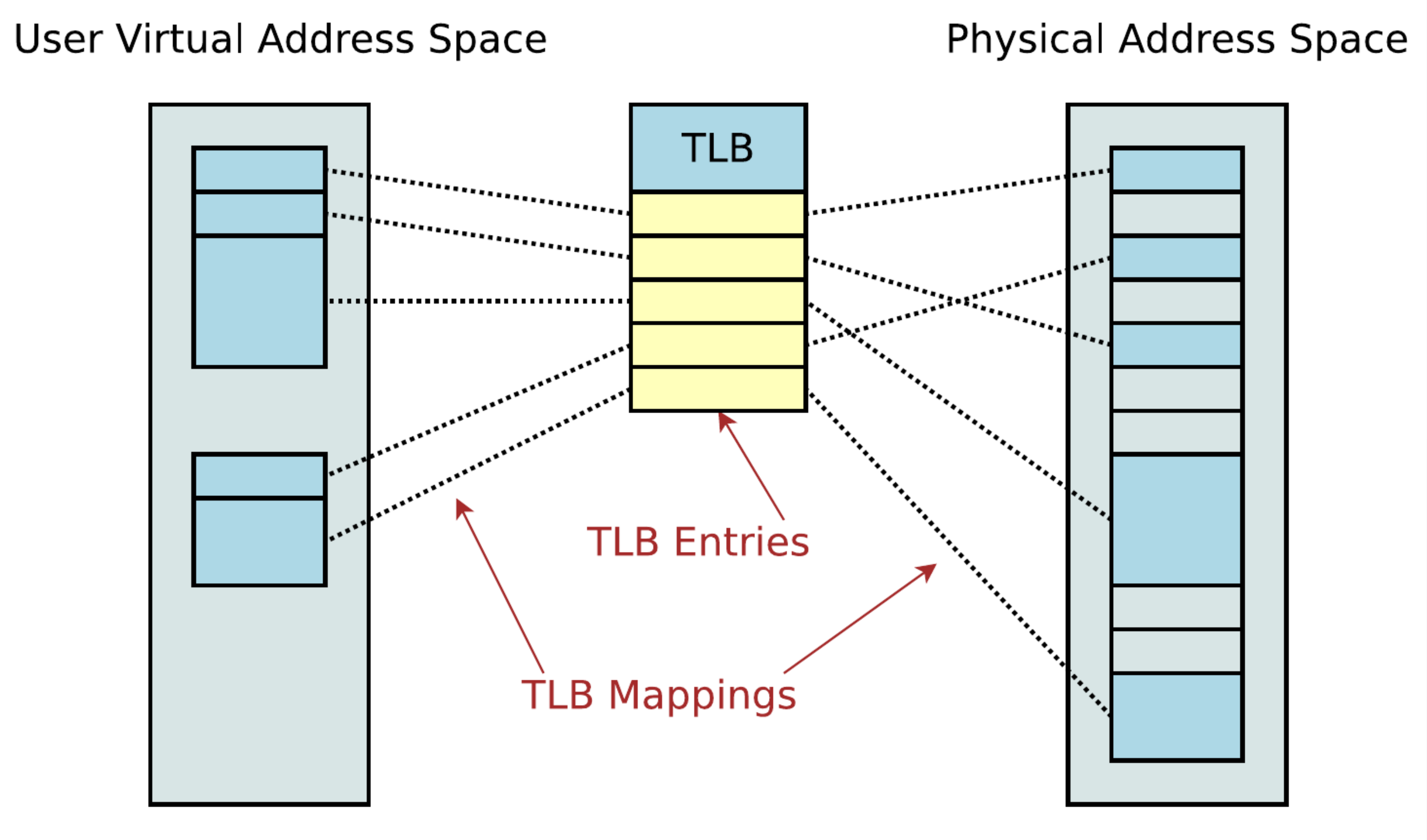
vm_end: first address outside virtual memory area
vm_start: first address within virtual memory area
vm_area_struct
VM_READ | VM_WRITE | VM_GROWS_DOWN
stack (anonymous)
vm_next
struct file /lib/ld.so
vm_file
vm_area_struct
VM_READ | VM_EXEC
Memory mapping
vm_next
struct file /lib/libc.so
vm_file
vm_area_struct
VM_READ | VM_EXEC
vm_next
vm_area_struct
VM_READ | VM_WRITE
Heap (anonymous)
vm_next
vm_area_struct
VM_READ | VM_WRITE
BSS (anonymous)
vm_next
vm_area_struct
VM_READ | VM_WRITE
Data (file-backed)
struct file /bin/gonzo
vm_file
vm_next
vm_area_struct
VM_READ | VM_EXEC
Text (file-backed)
vm_file
mmap
task_struct (/bin/gonzo)
mm
mm_struct

Translation of virtual memory to physical memory

- The 4GB virtual address space is divided into pages (4 KB by default, 2 MB or 1 GB is also supported on x86_64).
- The size of a VMA must be a multiple of page size.
- MMU uses TLB or page tables to translate a virtual address into a physical memory address.



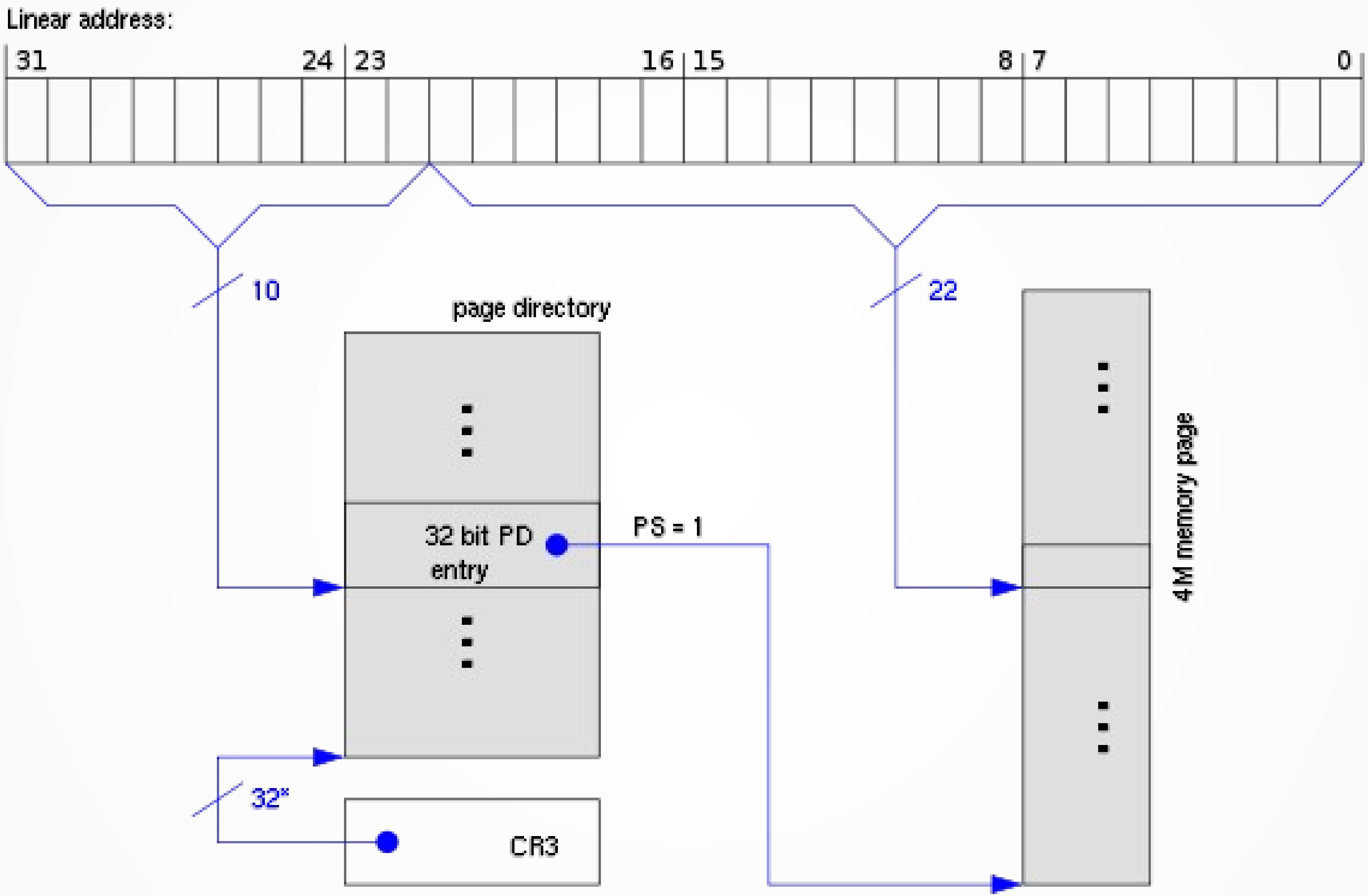
TLB mappings



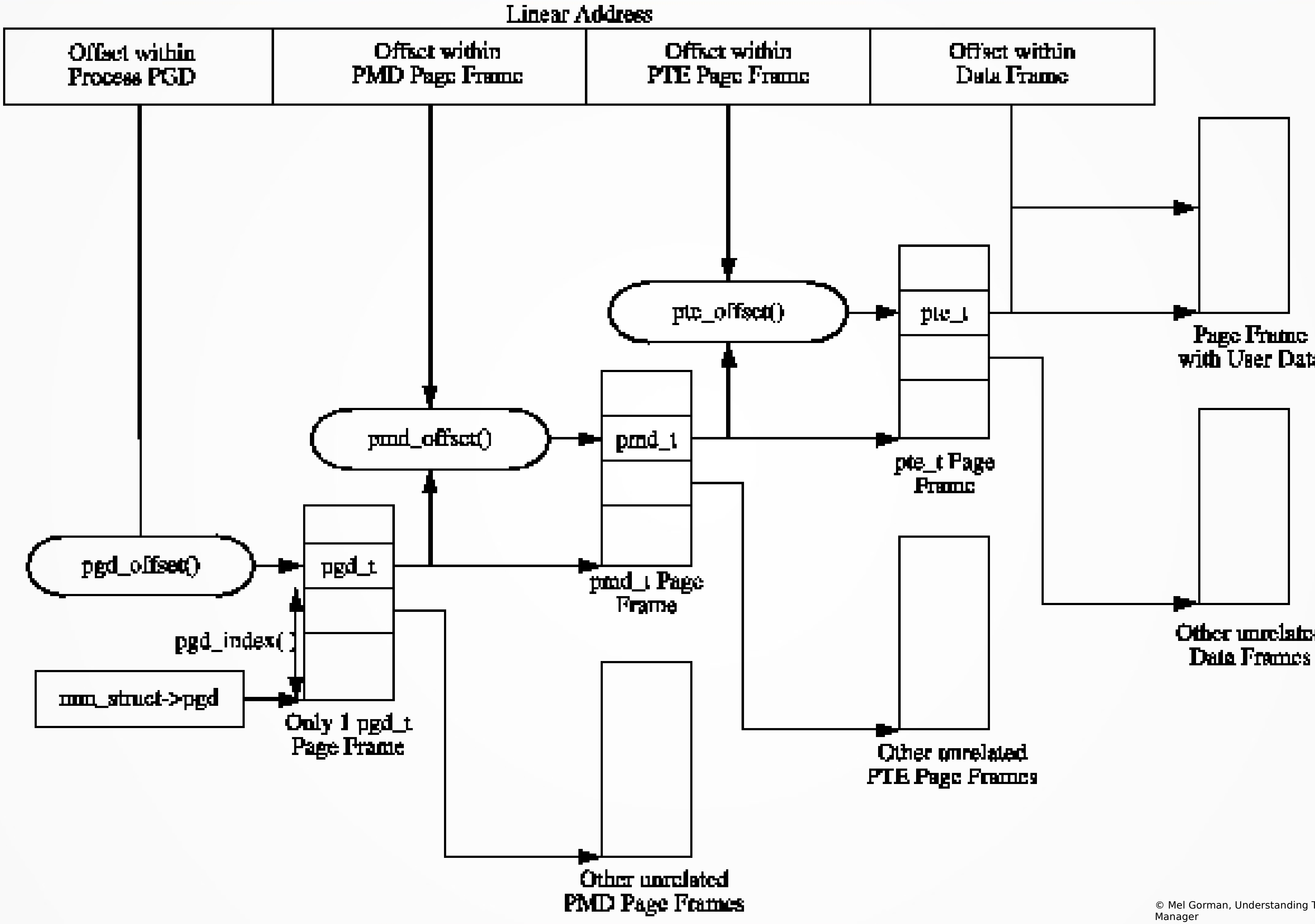
- Context switch loads loads new PDG pointer to CR3 and flushes TLB.

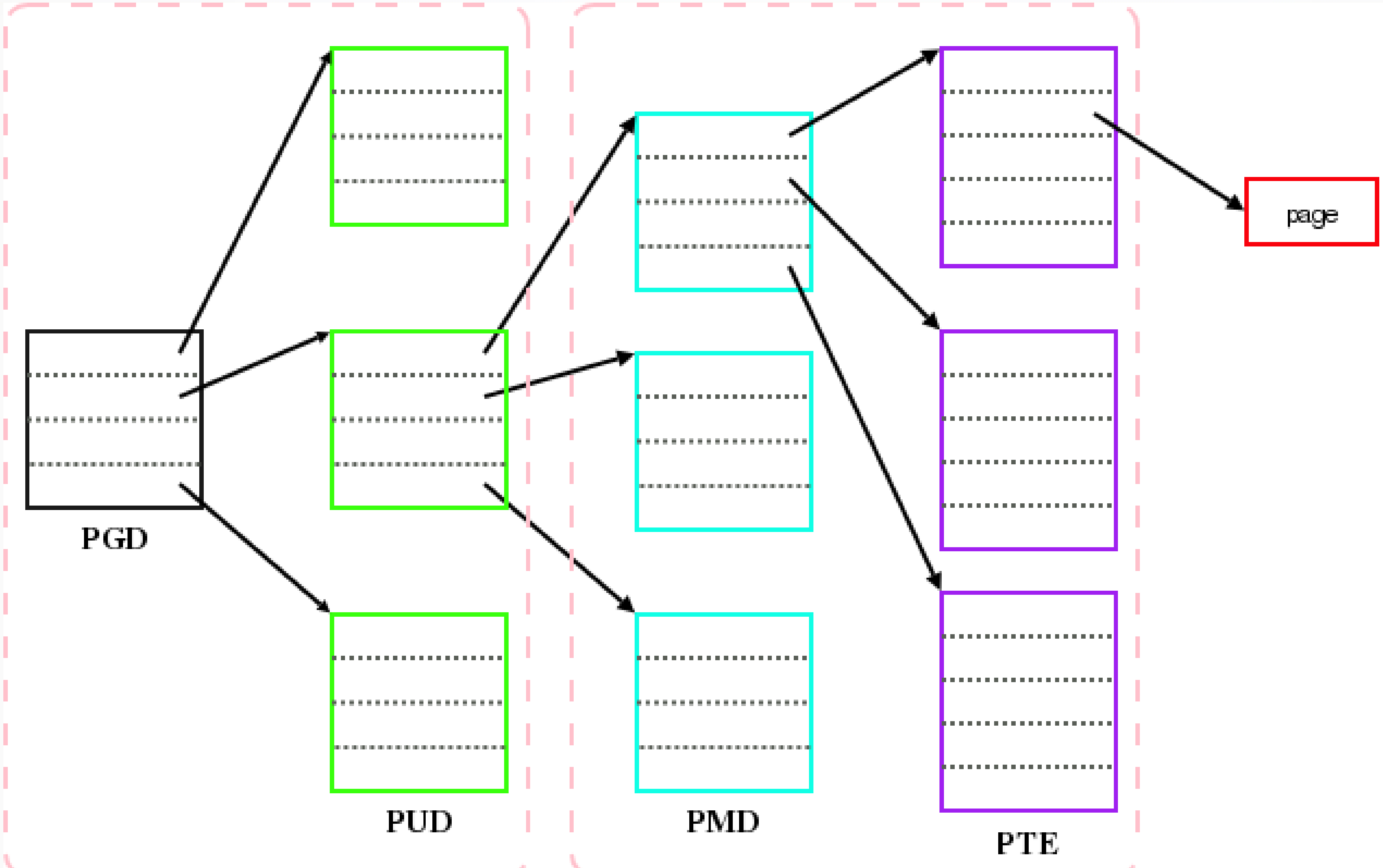
Page tables

- Each process has a pointer (mm_struct->pgd) to its own Page Global Directory (PGD) which is a physical page frame.
- The page tables are loaded differently depending on the architecture. On the x86, the process page table is loaded by copying mm_struct->pgd into the cr3 register.



*) 32 bits aligned to a 4-KByte boundy


Linear Address
Offset within Process PGD
Offset within PMD Page Frame
Offset within PTE Page Frame
Offset within Data Frame
pte_offset()
pte_t
Page Frame with User Data
pmd_offset()
pmd_t
pte_t Page Frame
pgd_offset()
pgd_t
pmd_t Page Frame
Other unrelated Data Frames
pgd_index( )
mm_struct->pgd
Only 1 pgd_t Page Frame
Other unrelated PTE Page Frames
Other unrelated PMD Page Frames

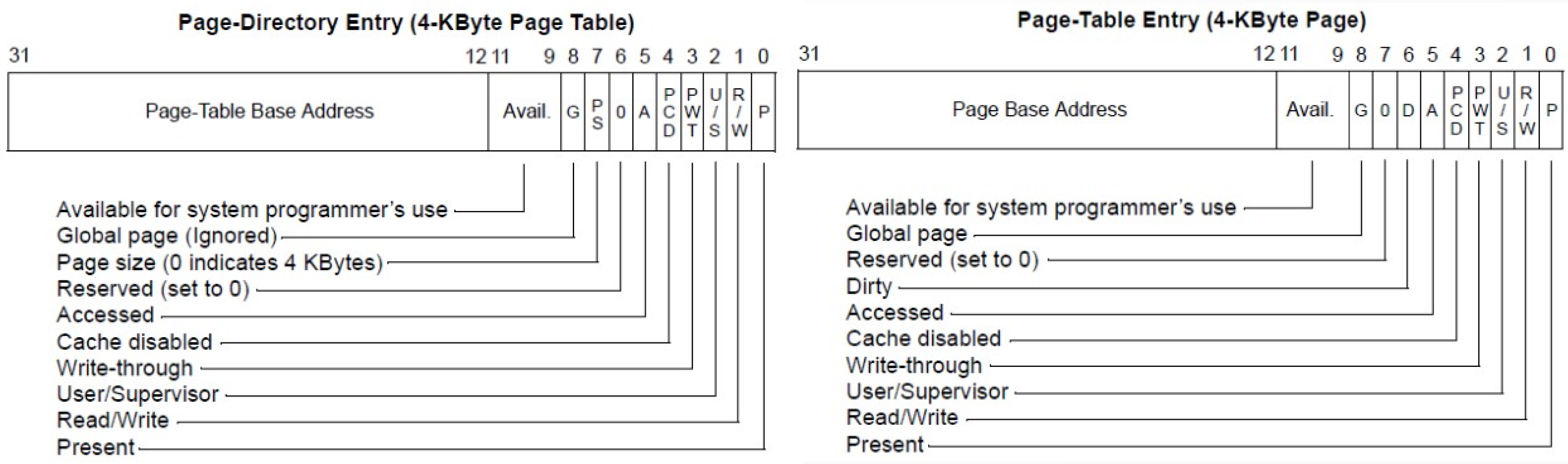


Architecture	Bits used			
	PGD	PUD	PMD	PTE
i386	22-31			12-21
i386 (PAE mode)	30-31		21-29	12-20
x86-64	39-46	30-38	21-29	12-20

ARM: it’s complicated…
See https://elinux.org/Tims_Notes_on_ARM_memory_allocation

Page tables

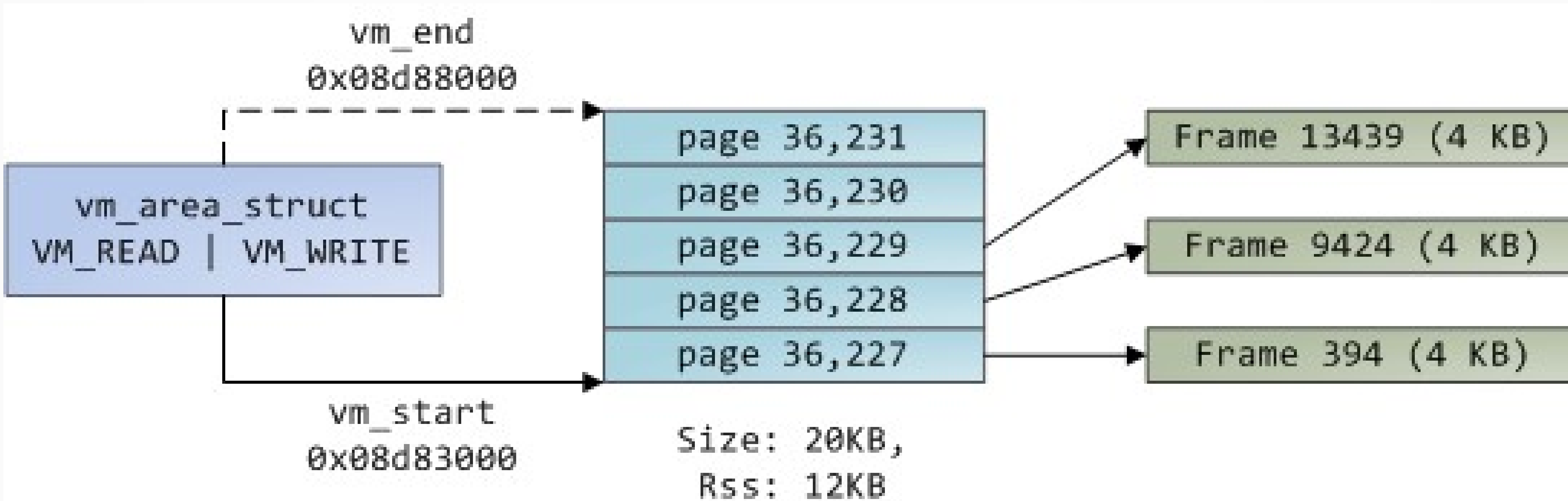
- Page directory and page table entries:



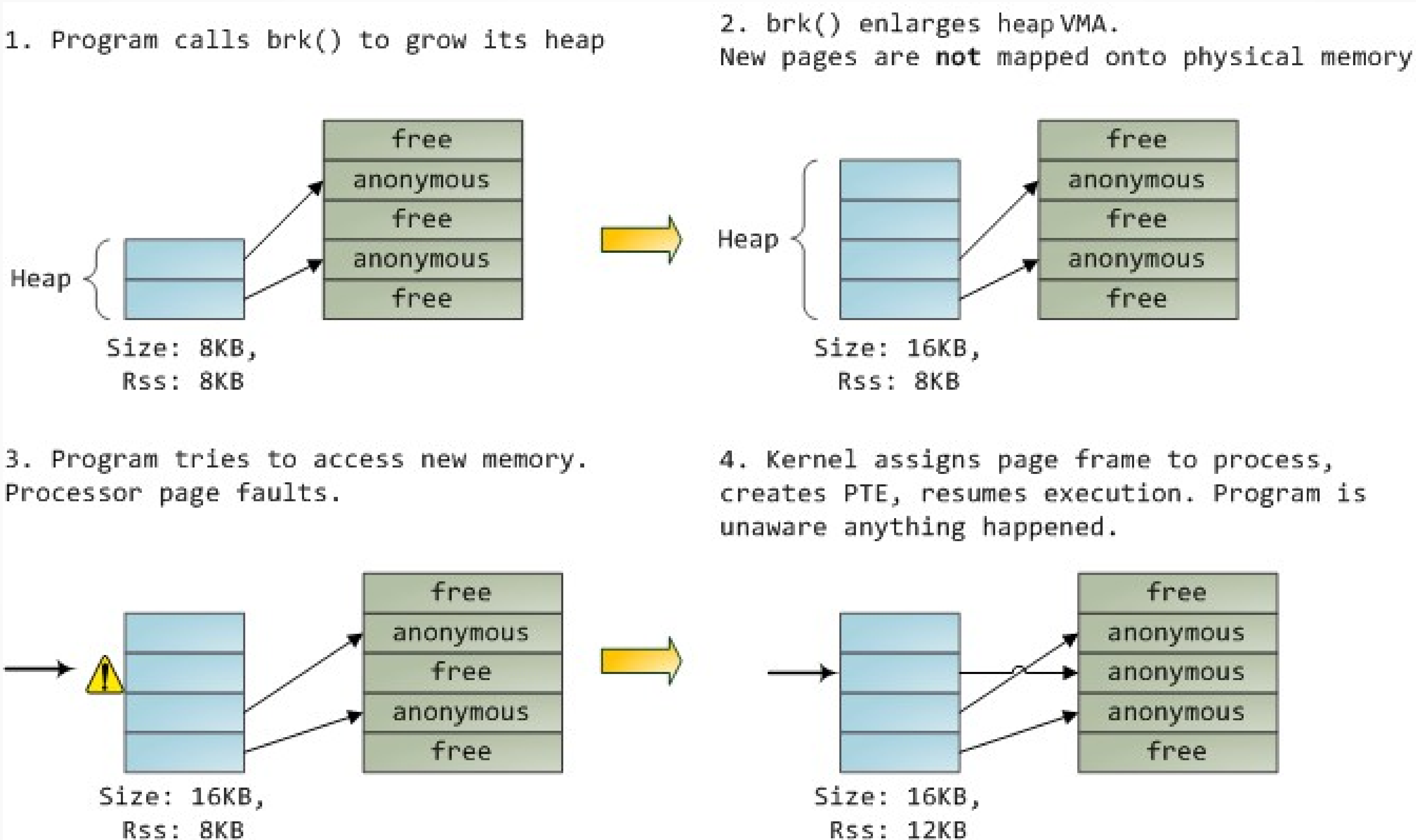
Page frames

- In Linux each page frame is tracked by a descriptor and several flags.
- Physical memory is managed with the buddy memory allocator.
- A page frame is free if it's available for allocation via the buddy system.
- An allocated page frame might be **anonymous** (holding program data), or **page cache** (holding data stored in a file or block device).

Arrows represent page table entries mapping pages onto page frames



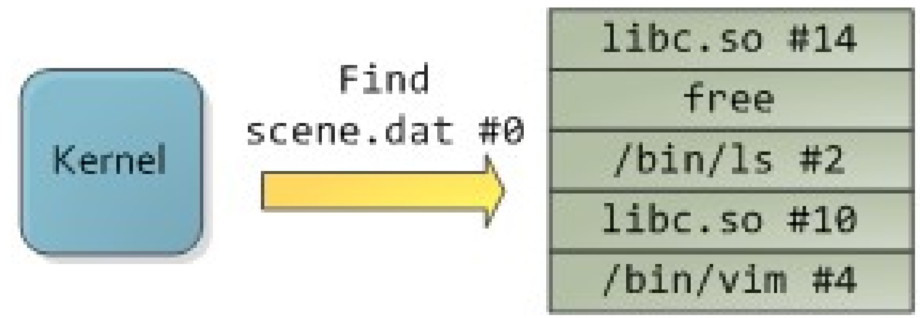
Some virtual pages lack arrows; this means their corresponding PTEs have the **Present** flag clear


1. Program calls brk() to grow its heap
Heap
free
anonymous
free
anonymous
free
Size: 8KB,
Rss: 8KB
2. brk() enlarges heap VMA.
New pages are **not** mapped onto physical memory.
Heap
free
anonymous
free
anonymous
free
Size: 16KB,
Rss: 8KB
3. Program tries to access new memory.
Processor page faults.
free
anonymous
free
anonymous
free
Size: 16KB,
Rss: 8KB
4. Kernel assigns page frame to process,
creates PTE, resumes execution. Program is
unaware anything happened.
free
anonymous
anonymous
anonymous
free
Size: 16KB,
Rss: 12KB

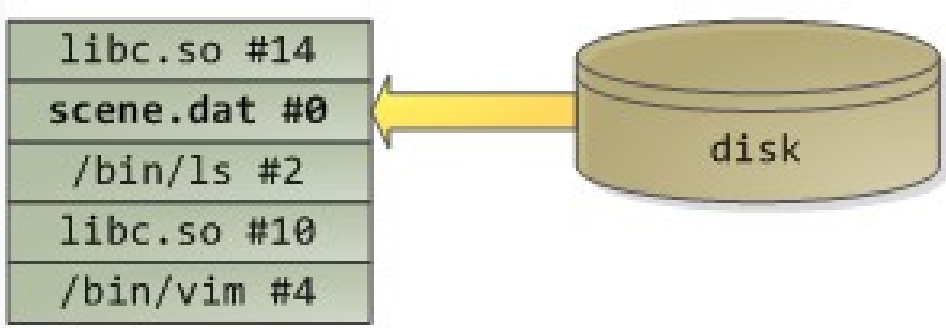
Page cache

1. Render asks for 512 bytes of scene.dat starting at offset 0.

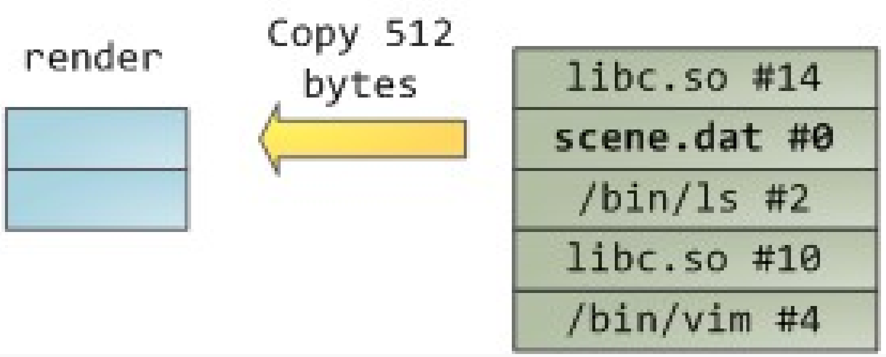
2. Kernel searches the page cache for the 4KB chunk of scene.dat satisfying the request. Suppose the data is not cached.



3. Kernel allocates page frame, initiates I/O requests for 4KB of scene.dat starting at offset 0 to be copied to allocated page frame

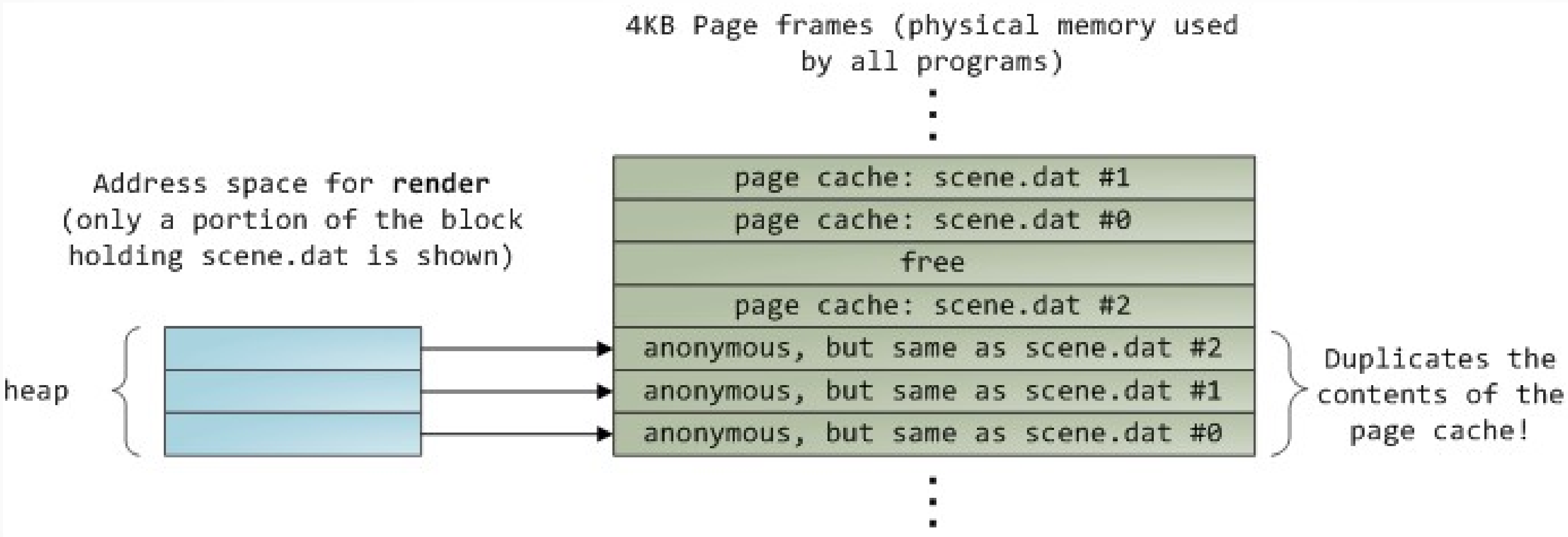


4. Kernel copies the requested 512 bytes from page cache to user buffer, read() system call ends.

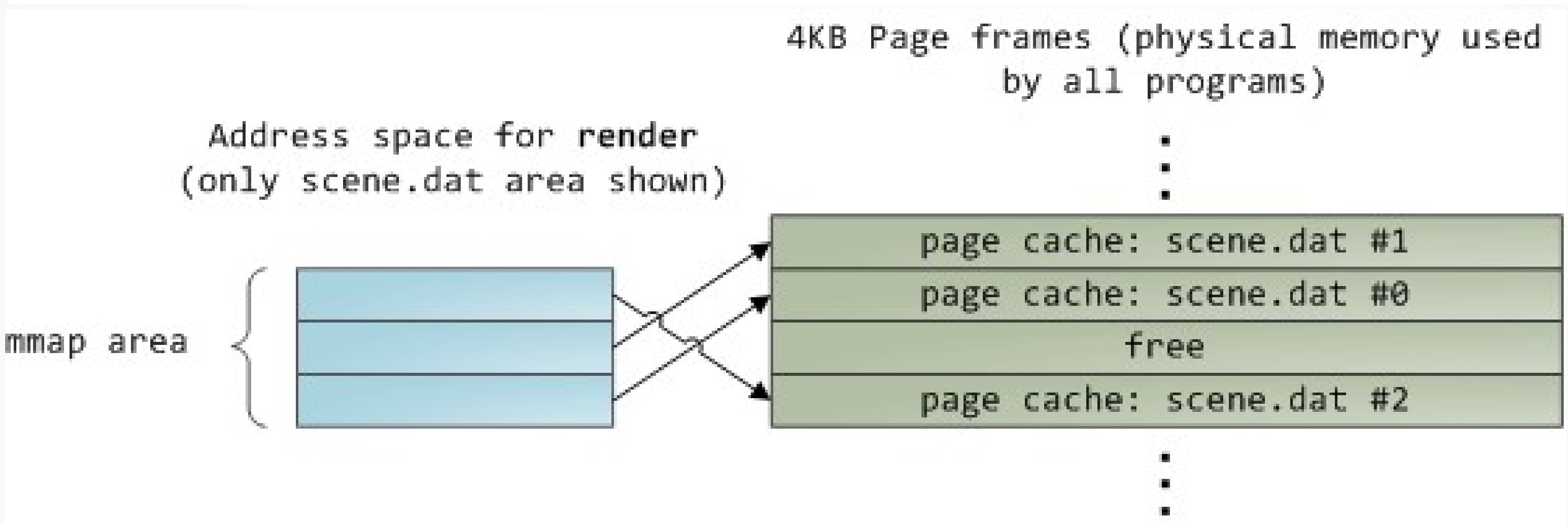


Page cache and memory-mapped files

read():



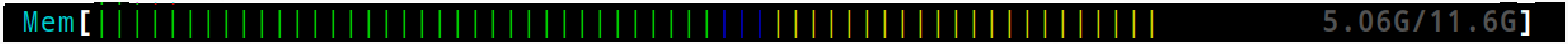
mmap():



Freeing/reclaiming page-cached memory

- When a process is terminated, the pages storing the mapped file are not freed.
- As long as there's enough free physical memory, the page cache won't be freed.
- It is not dependent on a particular process, page cache is a system-wide resource.

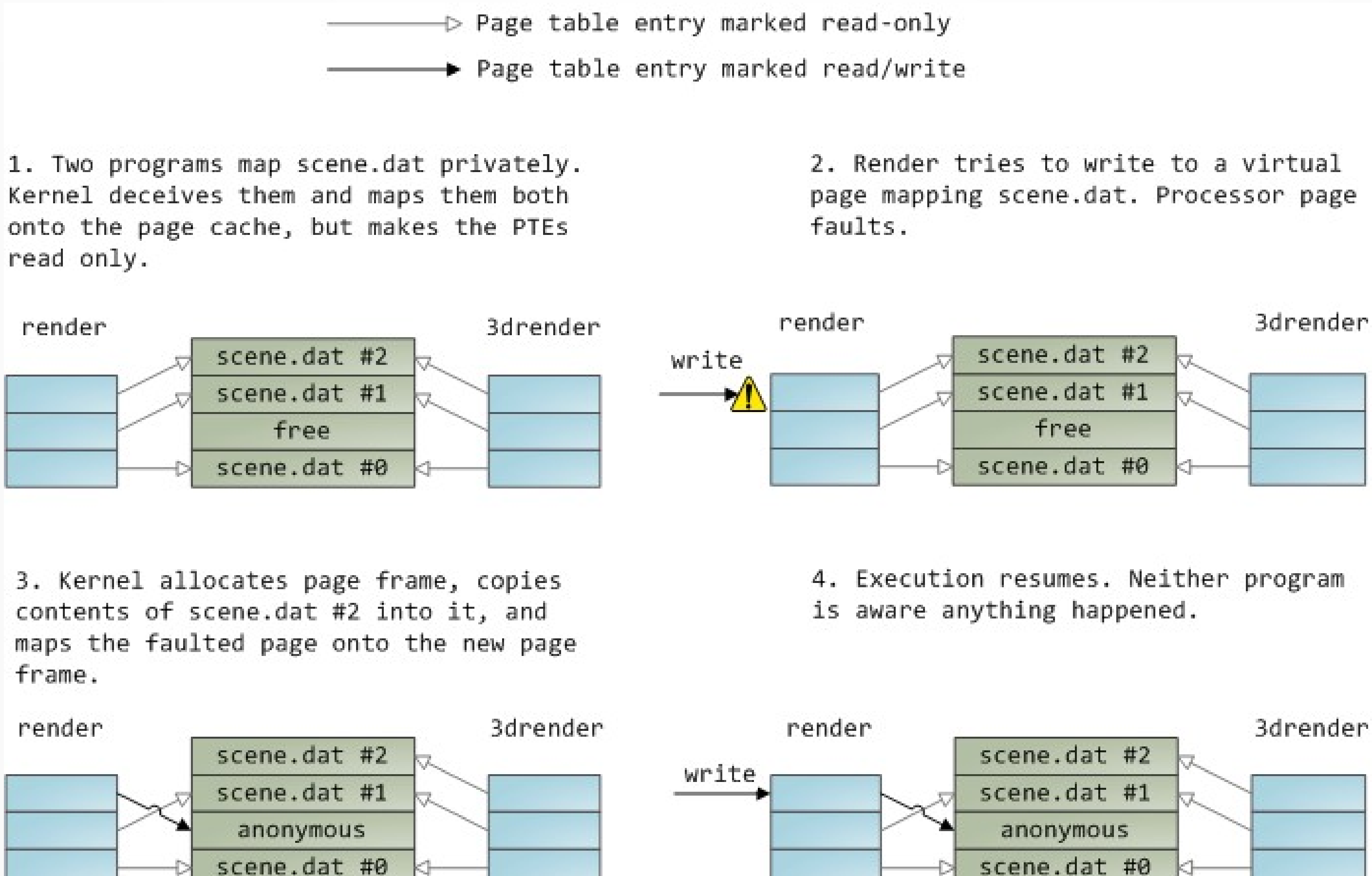
htop:



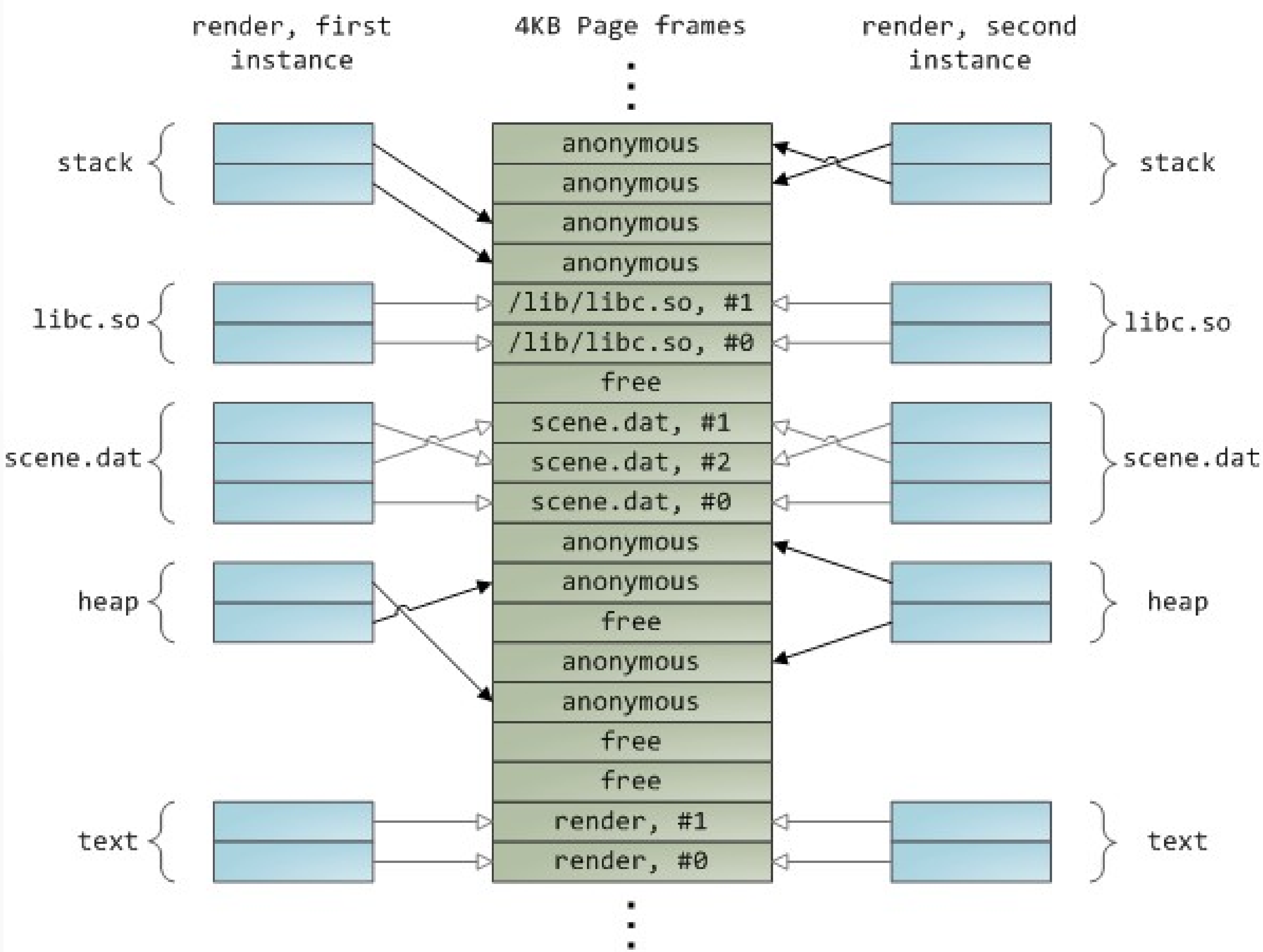

```
~/ free -m
              total        used        free      shared  buff/cache   available
Mem:          11906        4539        3274         516        4092        8981
Swap:             0           0           0
```

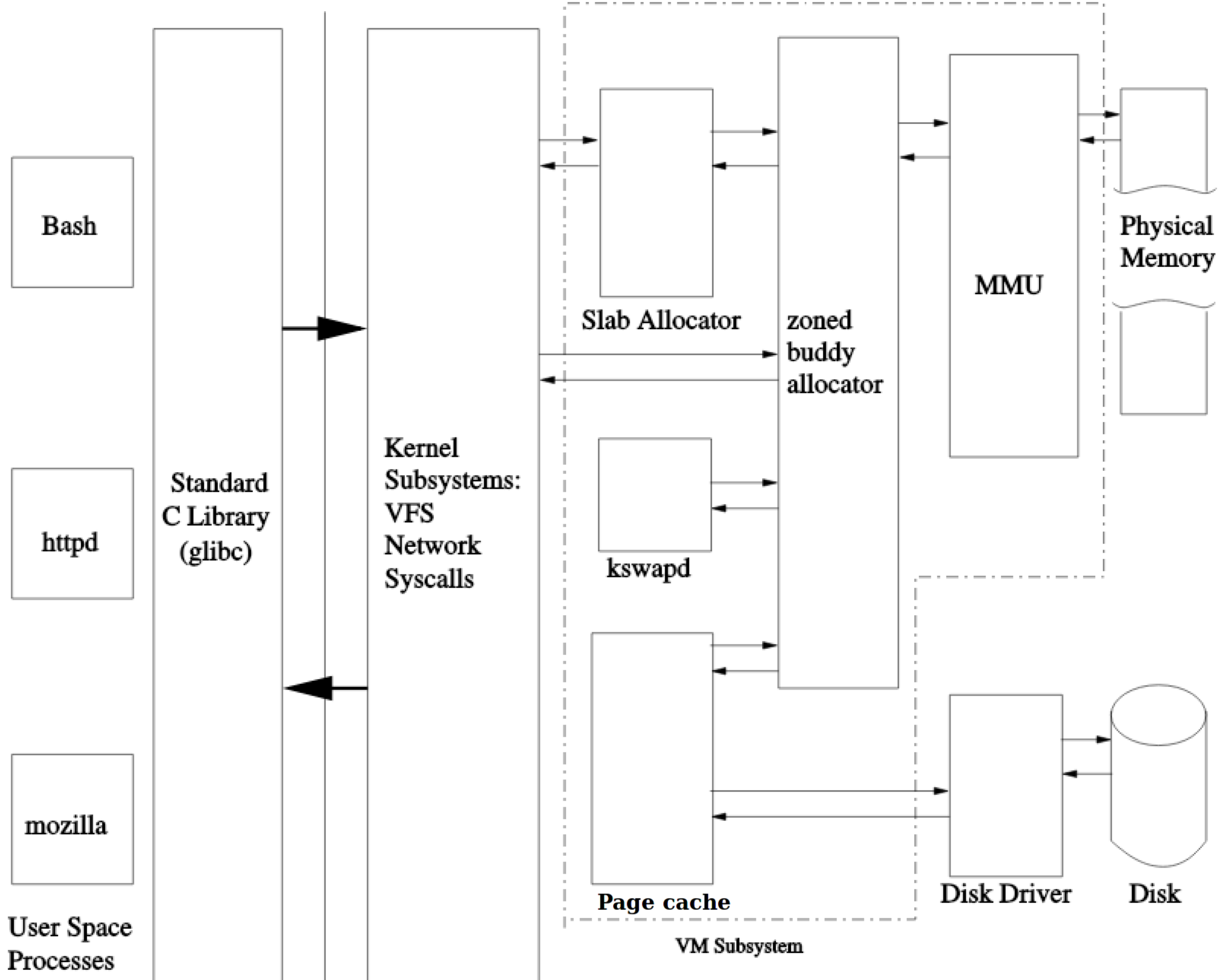
Private mappings. Copy-on-write

MAP_PRIVATE. The file is mapped copy-on-write, and any changes made in memory by this process are not reflected in the actual file, or in the mappings of other processes



Shared mappings




Bash
httpd
mozilla
User Space
Processes
Standard
C Library
(glibc)
Kernel
Subsystems:
VFS
Network
Syscalls
Slab Allocator
zoned
buddy
allocator
kswapd
Page cache
MMU
Physical
Memory
Disk Driver
Disk
VM Subsystem