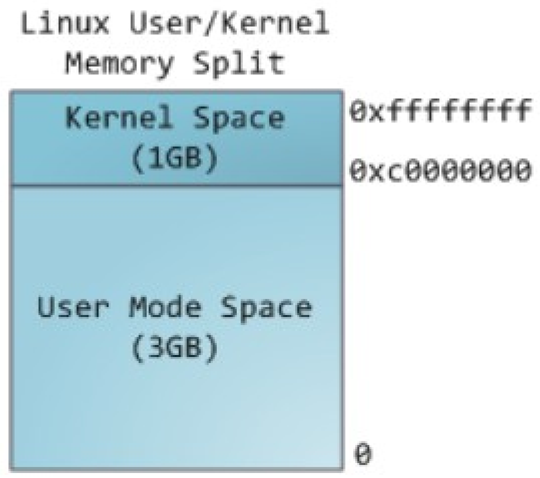
Memory management

Virtual address space

- Each process in a multi-tasking OS runs in its own memory sandbox called the virtual address space.
- In 32-bit mode this is a 4GB block of memory addresses.
- These virtual addresses are mapped to physical memory by page tables, which are maintained by the operating system kernel and consulted by the processor.
- Each process has its own set of page tables.
- Once virtual addresses are enabled, they apply to all software running in the machine, including the kernel itself.
- Thus a portion of the virtual address space must be reserved to the kernel

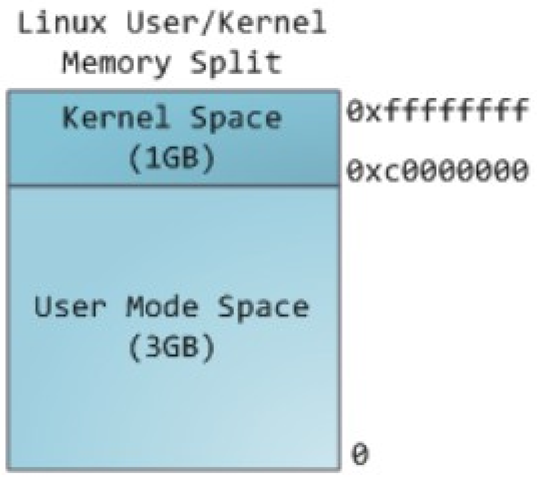


Kernel and user space

- Kernel might not use 1 GB much physical memory.
- It has that portion of address space available to map whatever physical memory it wishes.
- Kernel space is flagged in the page tables as exclusive to privileged code (ring 2 or lower), hence a page fault is triggered if user-mode programs try to touch it.
- In Linux, kernel space is constantly present and maps the same physical memory in all processes.
- Kernel code and data are always addressable, ready to handle interrupts or system calls at any time.
- By contrast, the mapping for the user-mode portion of the address space changes whenever a process switch happens

Kernel virtual address space

- Kernel address space is the area above CONFIG_PAGE_OFFSET.
- For 32-bit, this is configurable at kernel build time. The kernel can be given a different amount of address space as desired.
- Two kinds of addresses in kernel virtual address space
  - Kernel logical address
  - Kernel virtual address



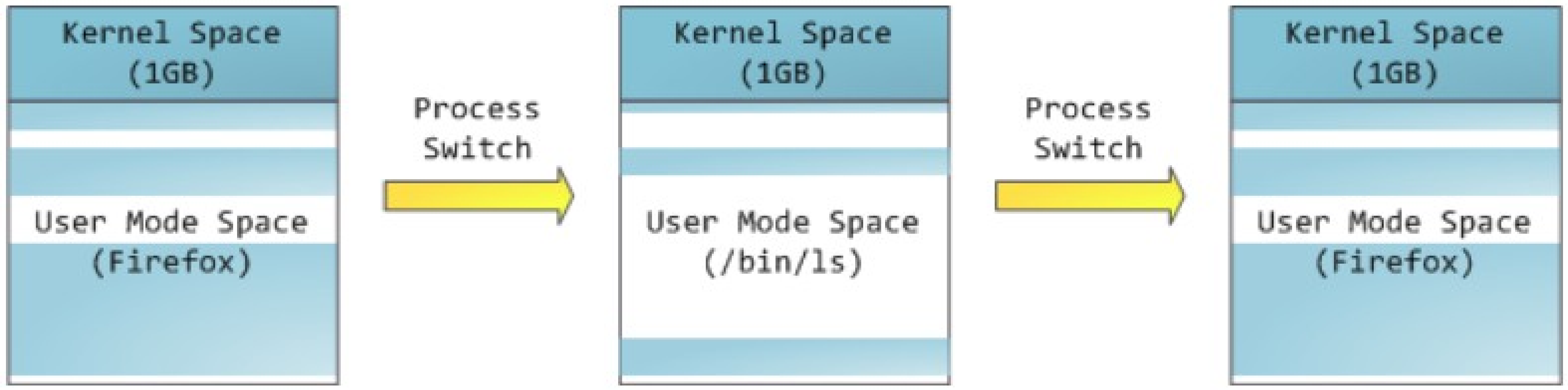
Kernel logical address

- Allocated with kmalloc()
- Holds all the kernel data structures
- Can never be swapped out
- Virtual addresses are a fixed offset from their physical addresses.
  - Virt: 0xc0000000 → Phys: 0x00000000
- This makes converting between physical and virtual addresses easy. Macros that do that
  - _pa(x)
  - _va(x)
- Virtually-contiguous regions are by nature also physically contiguous.
- Makes them suitable for DMA transfers.

Kernel virtual address

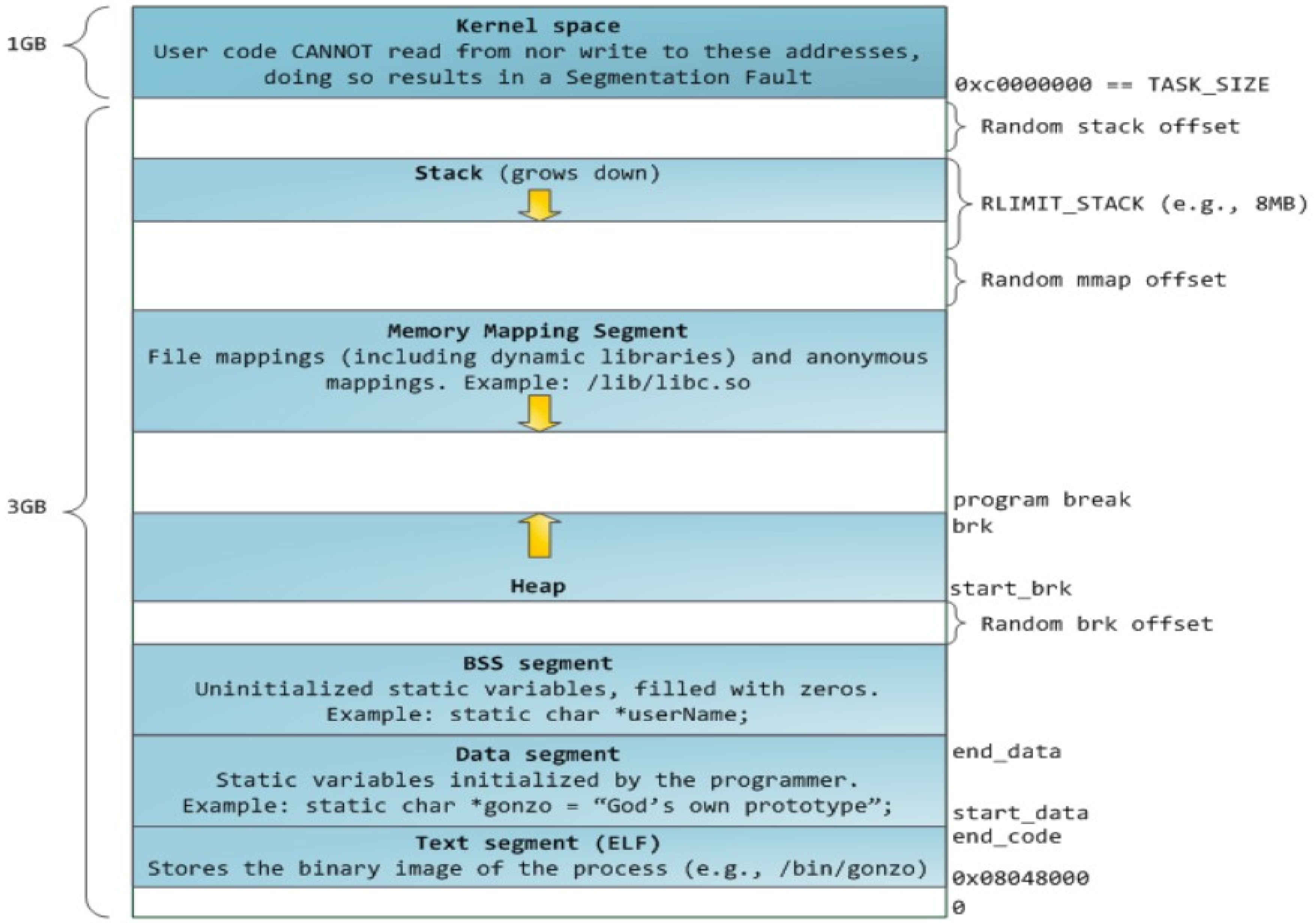
- Allocated with vmalloc()
- Contiguous virtual addresses can go to non-contiguous physical addresses
- Easier to allocate, especially for large buffers

Mapping changes on context switch



- Blue regions represent virtual addresses that are mapped to physical memory, whereas white regions are unmapped.
- In the example above, Firefox has used far more of its virtual address space due to its legendary memory hunger.

Memory segments


1GB
3GB
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
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
end_data
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

Virtual address randomization

- An exploit often needs to reference absolute memory locations: an address on the stack, the address for a library function, etc.
- Remote attackers must choose this location blindly, counting on the fact that address spaces of all processes are the same.
- When they are, the attack is successful.
- Thus address space randomization has become popular.
- Linux randomizes the stack, memory mapping segment, and heap by adding offsets to their starting addresses.
- Unfortunately the 32-bit address space is pretty tight, leaving little room for randomization and hampering its effectiveness.

Stack segment

- The topmost segment in the process address space is the stack, which stores local variables and function parameters in most programming languages.
- Calling a method or function pushes a new stack frame onto the stack.
- The stack frame is destroyed when the function returns.
- This simple design, possible because the data obeys strict LIFO order, means that no complex data structure is needed to track stack contents - a simple pointer to the top of the stack will do.
- Pushing and popping are thus very fast and deterministic.
- Also, the constant reuse of stack regions tends to keep active stack memory in the cpu caches, speeding up access.
- Each thread in a process gets its own stack.

Growing the stack segment

- It is possible to exhaust the area mapping the stack by pushing more data than it can fit.
- This triggers a page fault that is handled in Linux by expand_stack(), which in turn calls acct_stack_growth() to check whether it's appropriate to grow the stack.
- If the stack size is below RLIMIT_STACK (usually 8MB), then normally the stack grows and the program continues merrily, unaware of what just happened. This is the normal mechanism whereby stack size adjusts to demand.
- However, if the maximum stack size has been reached, we have a stack overflow and the program receives a Segmentation Fault.
- While the mapped stack area expands to meet demand, it does not shrink back when the stack gets smaller.
- Dynamic stack growth is the only situation in which access to an unmapped memory region, might be valid. Any other access to unmapped memory triggers a page fault that results in a Segmentation Fault. Some mapped areas are read-only, hence write attempts to these areas also lead to segfaults.

Memory mapping segment

- Below the stack, we have the memory mapping segment.
- Here the kernel maps contents of files directly to memory.
- Any application can ask for such a mapping via the Linux mmap() system call.
- Memory mapping is a convenient and high-performance way to do file I/O, so it is used for loading dynamic libraries.
- It is also possible to create an anonymous memory mapping that does not correspond to any files, being used instead for program data.
- In Linux, if you request a large block of memory via malloc(), the C library will create such an anonymous mapping instead of using heap memory. 'Large' means larger than MMAP_THRESHOLD bytes, 128 kB by default and adjustable via mallopt().

Heap segment

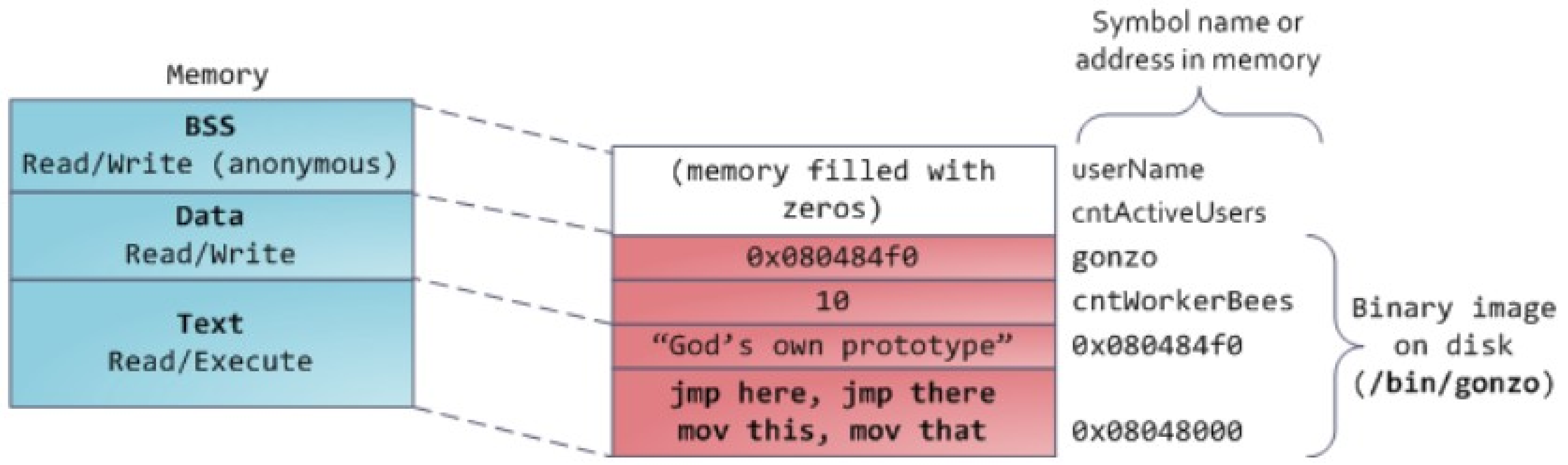
- Heap comes next in our plunge into address space.
- The heap provides runtime memory allocation, like the stack.
- Heap is meant for data that must outlive the function doing the allocation, unlike the stack.
- Most languages provide heap management to programs. Satisfying memory requests is thus a joint affair between the language runtime and the kernel.
- In C, the interface to heap allocation is malloc() and friends.
- In a garbage-collected language like C# the interface is the new keyword.
- 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 the brk() system call (implementation) to make room for the requested block.
- Heap management is complex, requiring sophisticated algorithms that strive for speed and efficient memory usage in the face of our programs' chaotic allocation patterns. The time needed to service a heap request can vary substantially.

BSS and data segment

- BSS and data segments store contents for static (global) variables in C.
- BSS
  - stores the contents of uninitialized static variables, whose values are not set by the programmer in source code.
  - The BSS memory area is anonymous: it does not map any file.
  - If you say static int cntActiveUsers, the contents of cntActiveUsers live in the BSS.
- The data segment
  - holds the contents for static variables initialized in source code.
  - This memory area is not anonymous. It maps the part of the program's binary image that contains the initial static values given in source code.
  - So if you say static int cntWorkerBees = 10, the contents of cntWorkerBees live in the data segment and start out as 10.
  - Even though the data segment maps a file, it is a private memory mapping, which means that updates to memory are not reflected in the underlying file.
  - This must be the case, otherwise assignments to global variables would change your on-disk binary image.

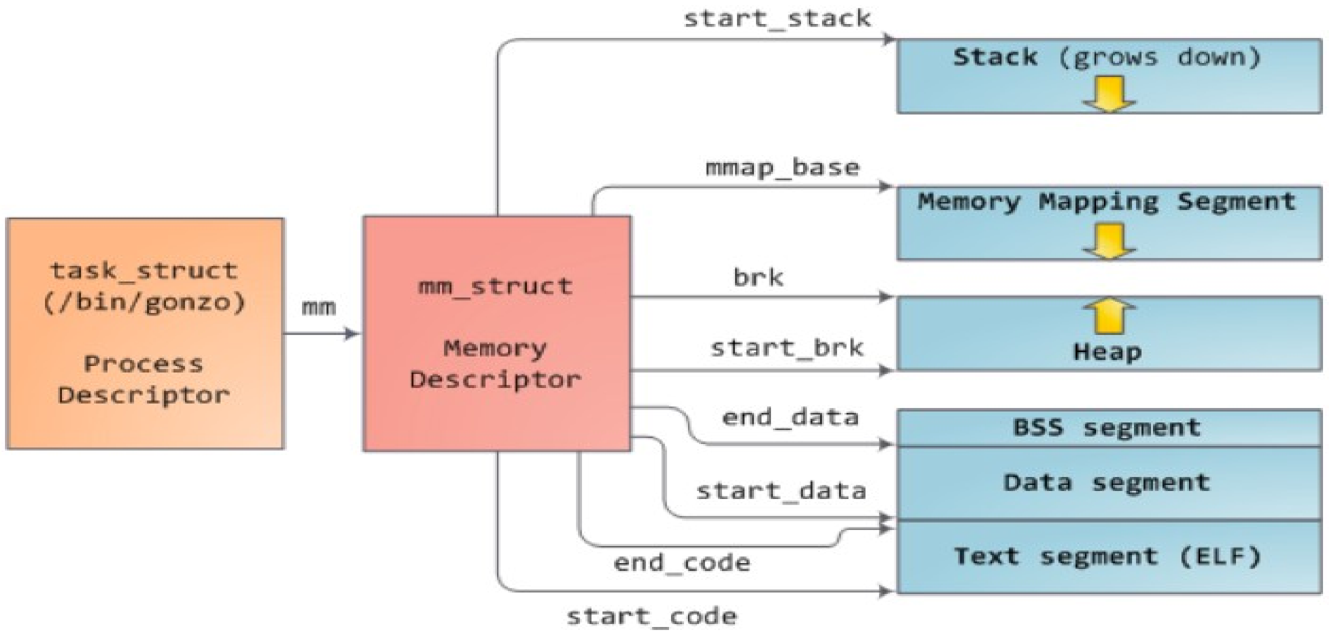
Text segment

- Text segment is
  - read-only.
  - stores all of your code in addition to tidbits like string literals.
  - maps your binary file in memory.
  - writes to this area earn your program a Segmentation Fault. This helps prevent pointer bugs

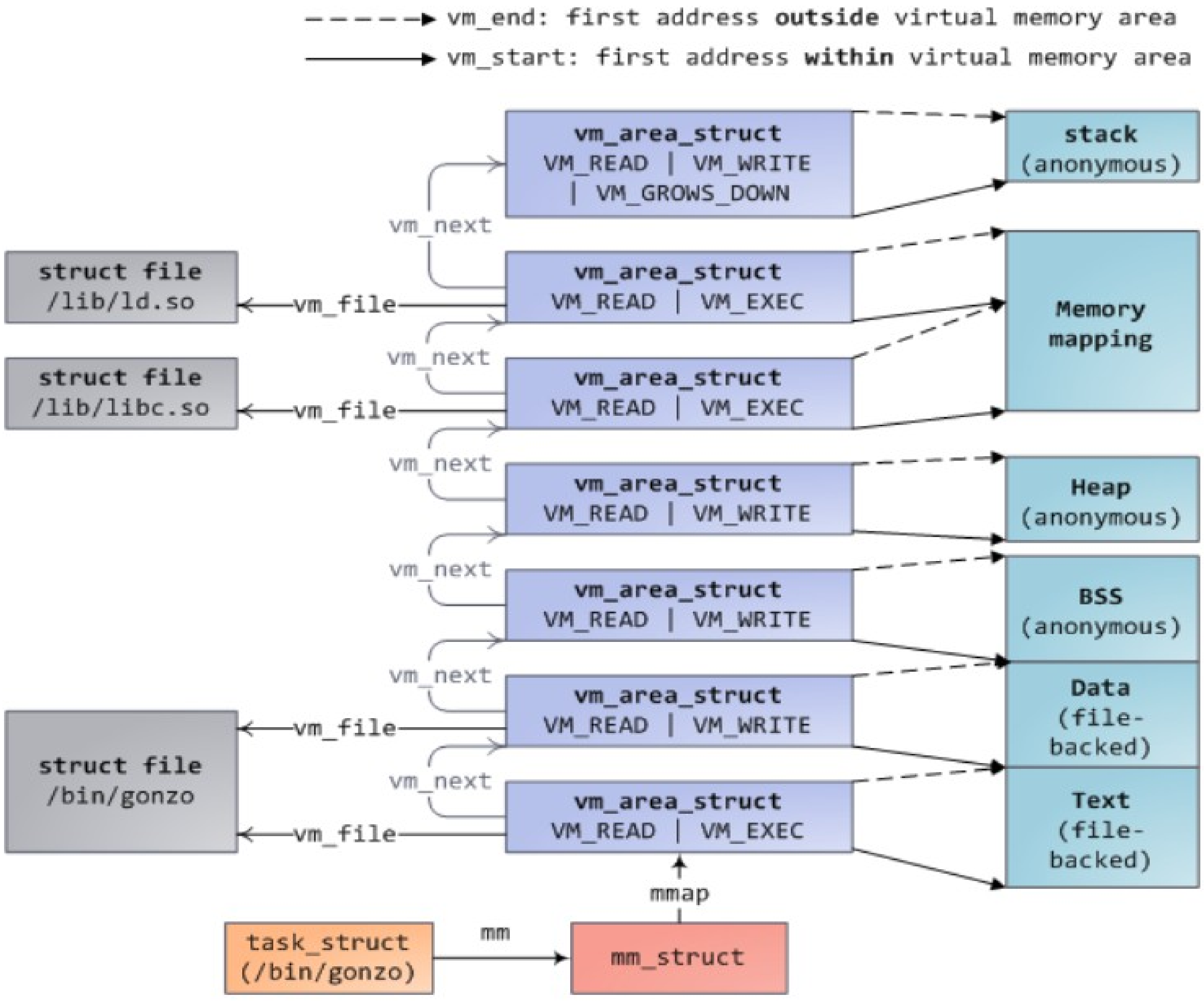


How kernel manages process memory

- Linux processes are implemented in the kernel as instances of task_struct, the process descriptor.
- The mm field in task_struct points to the memory descriptor, mm_struct, which is an executive summary of a program's memory.



Virtual memory area



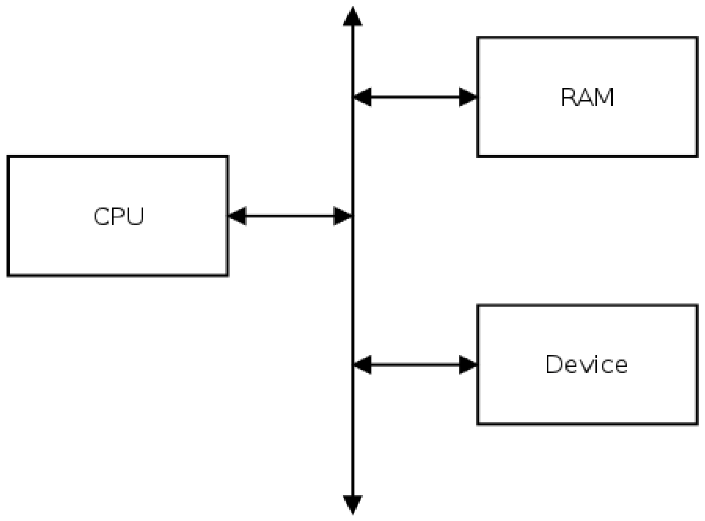
VMA information

- Each virtual memory area (VMA) is a contiguous range of virtual addresses; these areas never overlap.
- An instance of vm_area_struct fully describes
  - a memory area, including its start and end addresses
  - flags to determine access rights and behaviors
  - the vm_file field to specify which file is being mapped by the area, if any. A VMA that does not map a file is anonymous.
- Each memory segment above (e.g., heap, stack) corresponds to a single VMA, with the exception of the memory mapping segment. This is not a requirement, though it is usual in x86 machines.
- VMAs do not care which segment they are in.

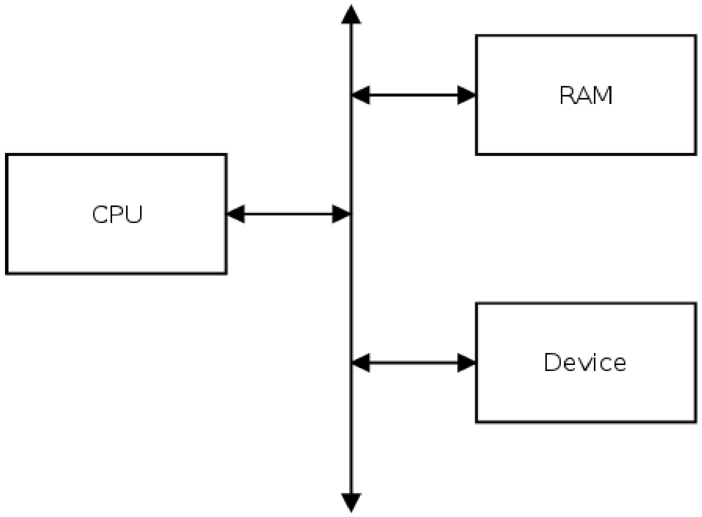
VMA data structures

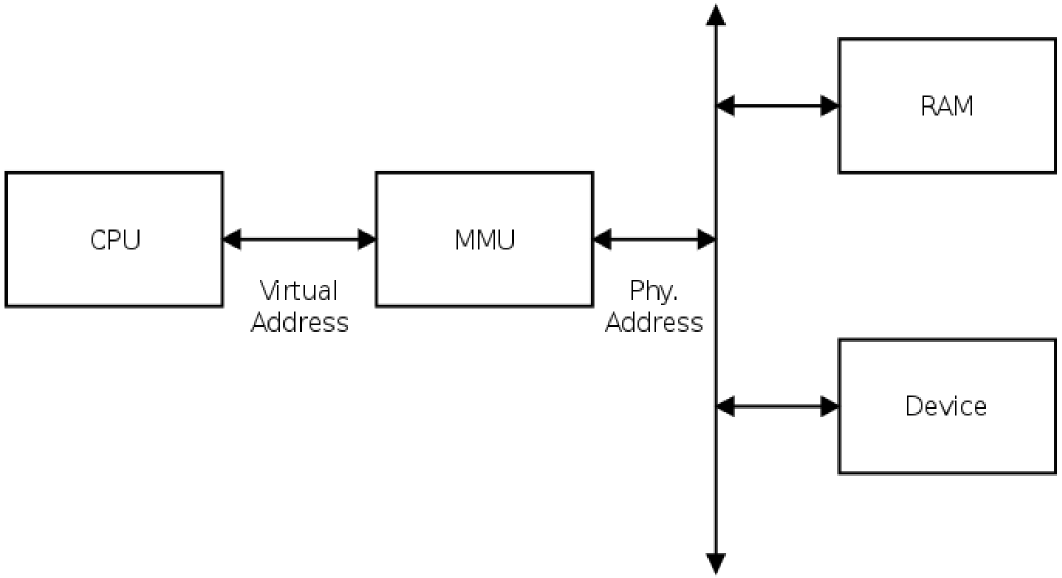
- A program's VMAs are stored in its memory descriptor
  - as a linked list in the mmap field, ordered by starting virtual address
  - as a red-black tree rooted at the mm_rb field.
- When you read file /proc/pid_of_process/maps, the kernel is simply going through the linked list of VMAs for the process and printing each one.
- The red-black tree allows the kernel to search quickly for the memory area covering a given virtual address.

Virtual address to physical address

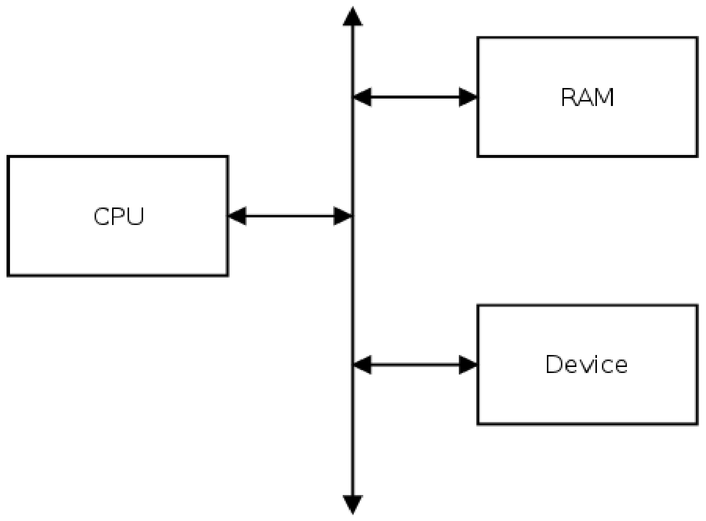


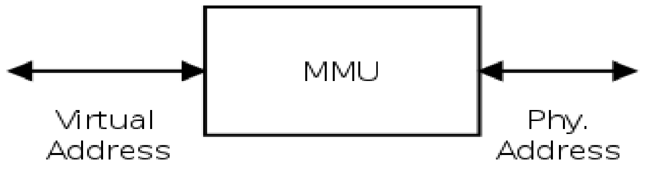
Virtual address to physical address



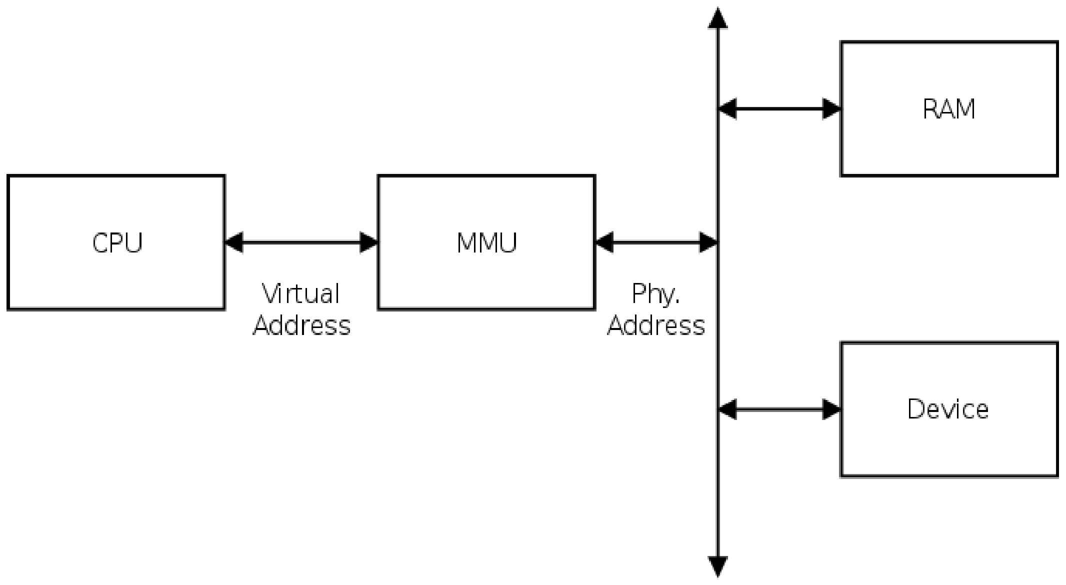


Virtual address to physical address

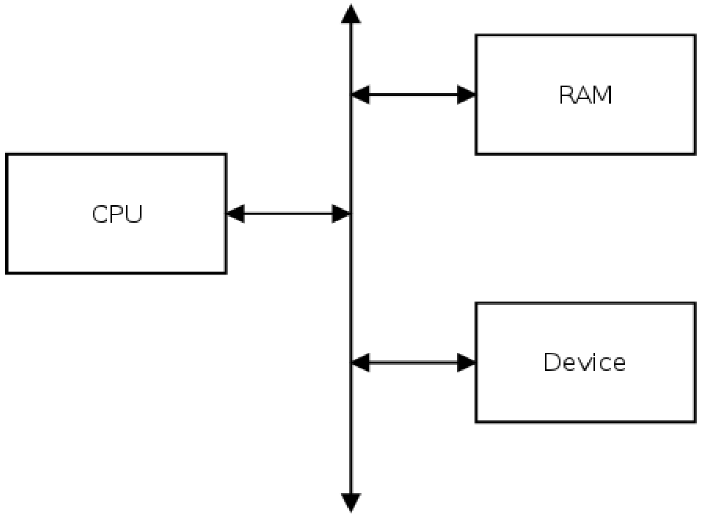


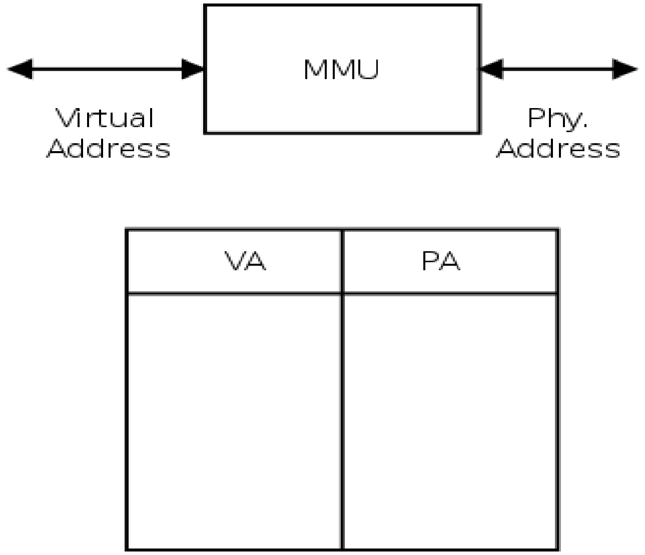


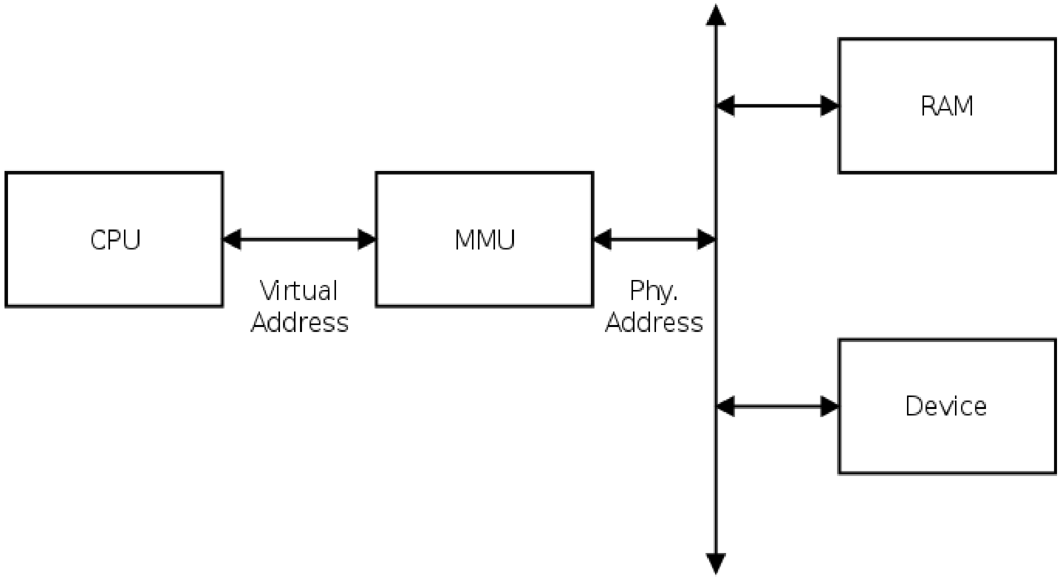
VA	PA

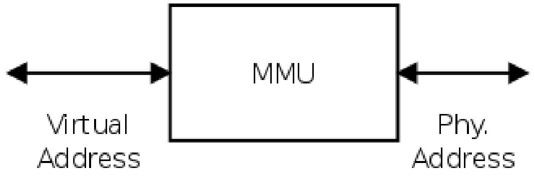


Virtual address to physical address





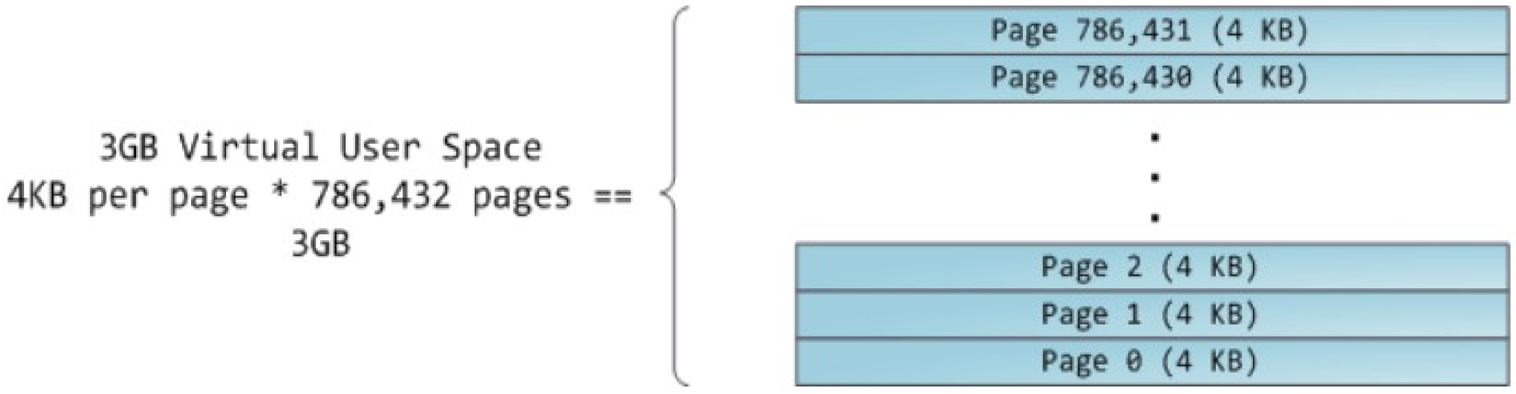




VP	PP
0	1
1	2
2	0

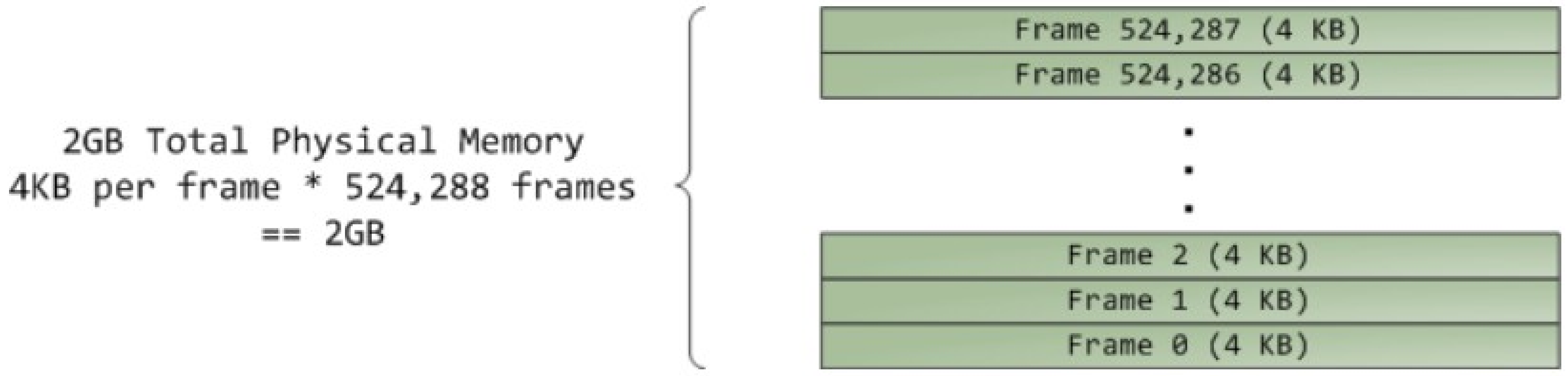
Virtual pages

- The 4GB virtual address space is divided into pages.
- Linux maps the user portion of the virtual address space using 4KB pages.
- Bytes 0-4095 fall in page 0, bytes 4096-8191 fall in page 1, and so on.
- The size of a VMA must be a multiple of page size.



Physical page frame

- This physical address space is broken down by the kernel into page frames.
- The processor doesn't know or care about frames, yet they are crucial to the kernel because the page frame is the unit of physical memory management.
- Linux uses 4KB page frames in 32-bit mode. In the kernel, the abbreviation pfn, for page frame number, is often used to refer to refer to physical page frames.
- In Linux, physical memory is managed with the buddy memory allocation technique, hence a page frame is free if it's available for allocation via the buddy system.

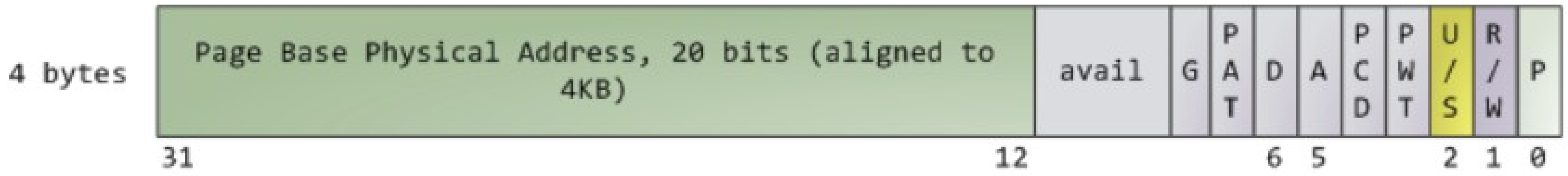


A page table per process

- Each process has its own set of page tables – created at time of process creation and modified over its lifetime.
- Per process page tables are large and stored in RAM.
- Linux stores a pointer to a process' page tables in the pgd field of the memory descriptor.

A page table entry (PTE)

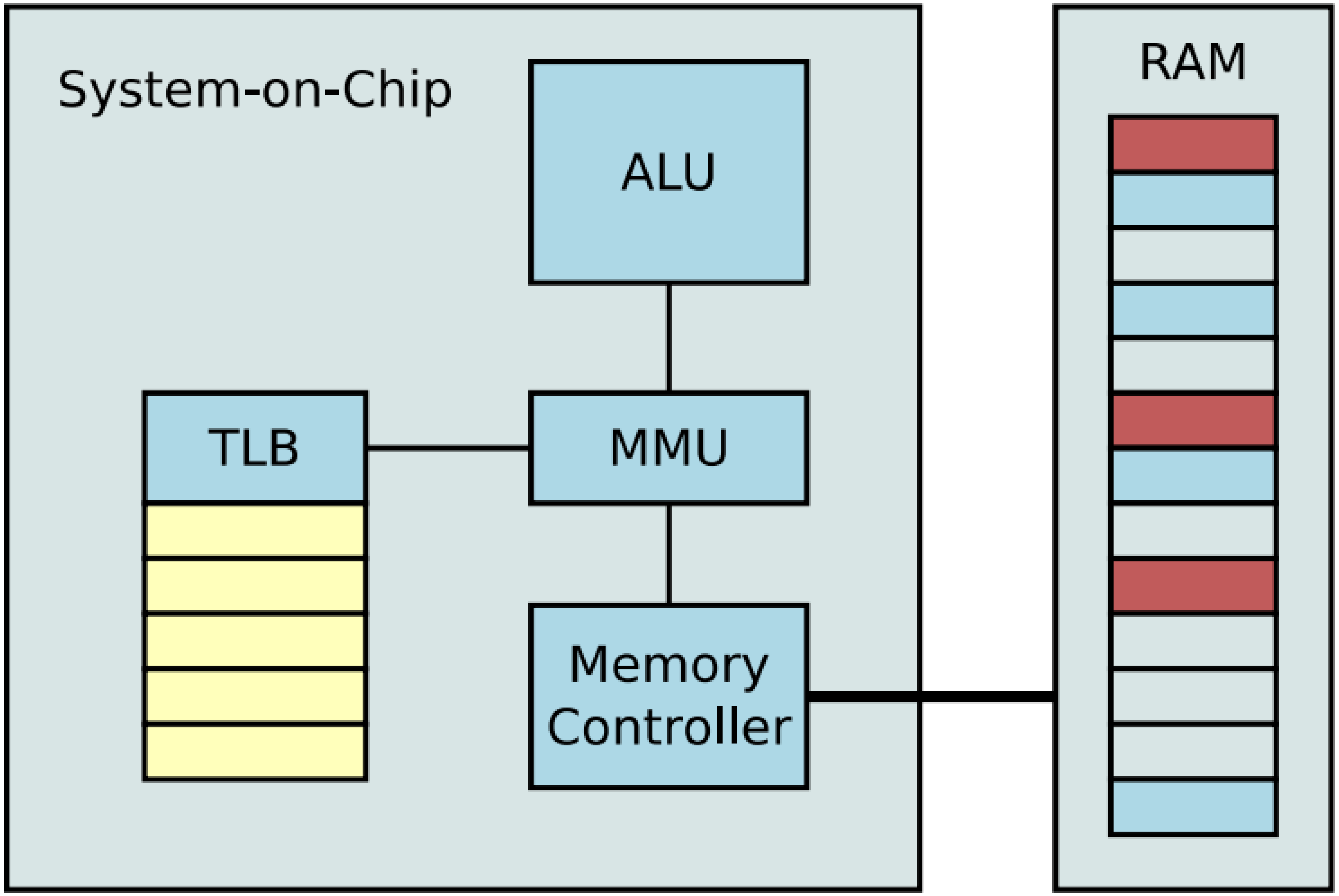
- Bit P tells the processor whether the virtual page is present in physical memory. If 0, accessing the page triggers a page fault. When 0, the kernel can do whatever it pleases with the remaining fields.
- The R/W flag stands for read/write; if clear, the page is read-only.
- Flag U/S stands for user/supervisor; if clear, then the page can only be accessed by the kernel. These flags are used to implement the read-only memory and protected kernel space we saw before.
- Bits D and A are for dirty and accessed. A dirty page has had a write, while an accessed page has had a write or read. Both flags are sticky: the processor only sets them, they must be cleared by the kernel.
- Finally, the PTE stores the starting physical address that corresponds to this page, aligned to 4KB.



MMU: hardware based address translation

- MMU is programmed with page table details to translate a virtual address into a physical memory address.
- A special register, the ***page table base register (PTBR)***, points to the beginning of the page table of the currently scheduled process.
- Whenever a process switch occurs, PTBR is changed.

Part of page table cached in TLB



Kernel involvement in address translation

- Once the page table for the scheduled process is appropriately set up, every user space process memory access (instruction fetch, data read/write, stack, etc) goes only through the MMU.
- The hardware handles it all, without need to switch into the kernel on every access.
- Kernel is notified only in case of exceptions called page faults.

3 cases of page faults and kernel involvement

(a) **Illegal access:** If there's no suitable VMA, no contract covers the attempted memory access and the process is punished by Segmentation Fault.

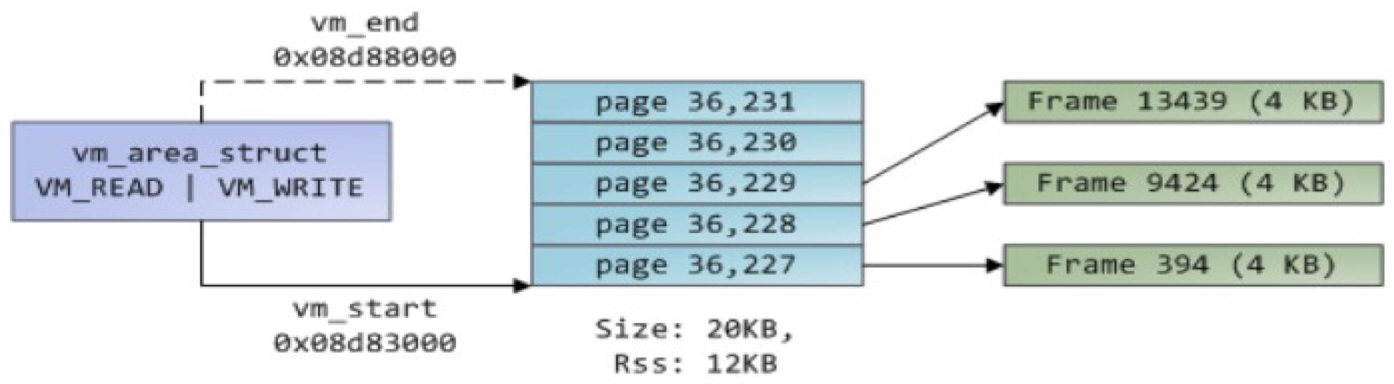
(b) **Legal access, mapped page, swapped out:** The PTE for a swapped out page, for example, has 0 in the Present flag but is not blank. Instead, it stores the swap location holding the page contents, which must be read from disk and loaded into a page frame by do_swap_page() in what is called a major fault.

(c) **Legal access, unmapped page:** When a VMA is found the kernel must handle the fault by looking at the PTE contents and the type of VMA. In our case, the PTE shows the page is not present. In fact, our PTE is completely blank (all zeros), which in Linux means the virtual page has never been mapped. This must be handled by do_anonymous_page(), which allocates a page frame and makes a PTE to map the faulted virtual page onto the freshly allocated frame.

Lazy Allocation: agreed upon (VMA) vs. actually done (PTE)

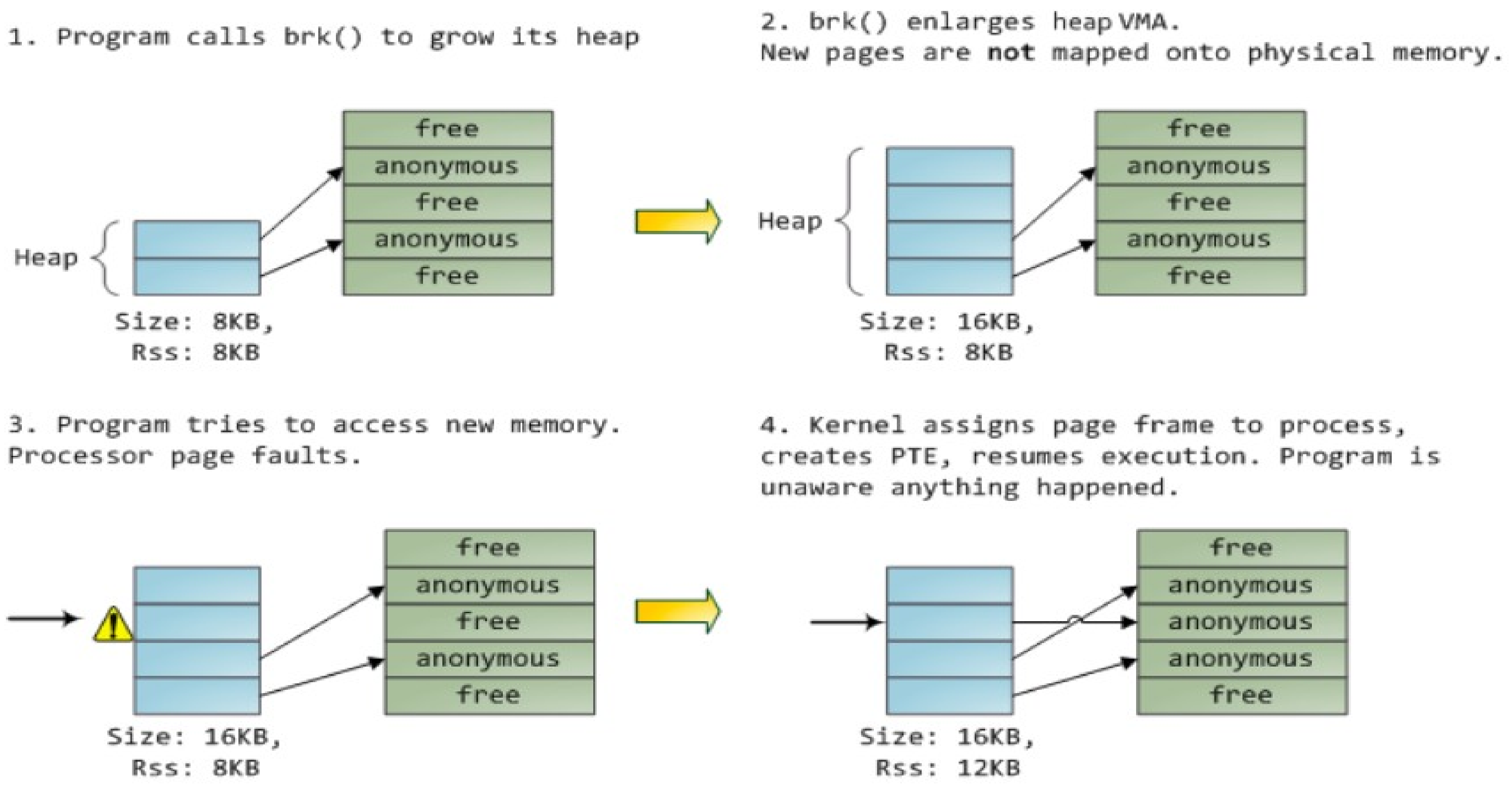
- A VMA is like a contract between your program and the kernel.
- You ask for something to be done (memory allocated, a file mapped, etc.), the kernel says "sure", and it creates or updates the appropriate VMA.
- But it does not actually honor the request right away, it waits until a page fault happens to do real work.
- The rule is that VMAs record what has been agreed upon, while Page table entries (PTE) reflect what has actually been done by the lazy kernel.

VMA, PTE and page frames



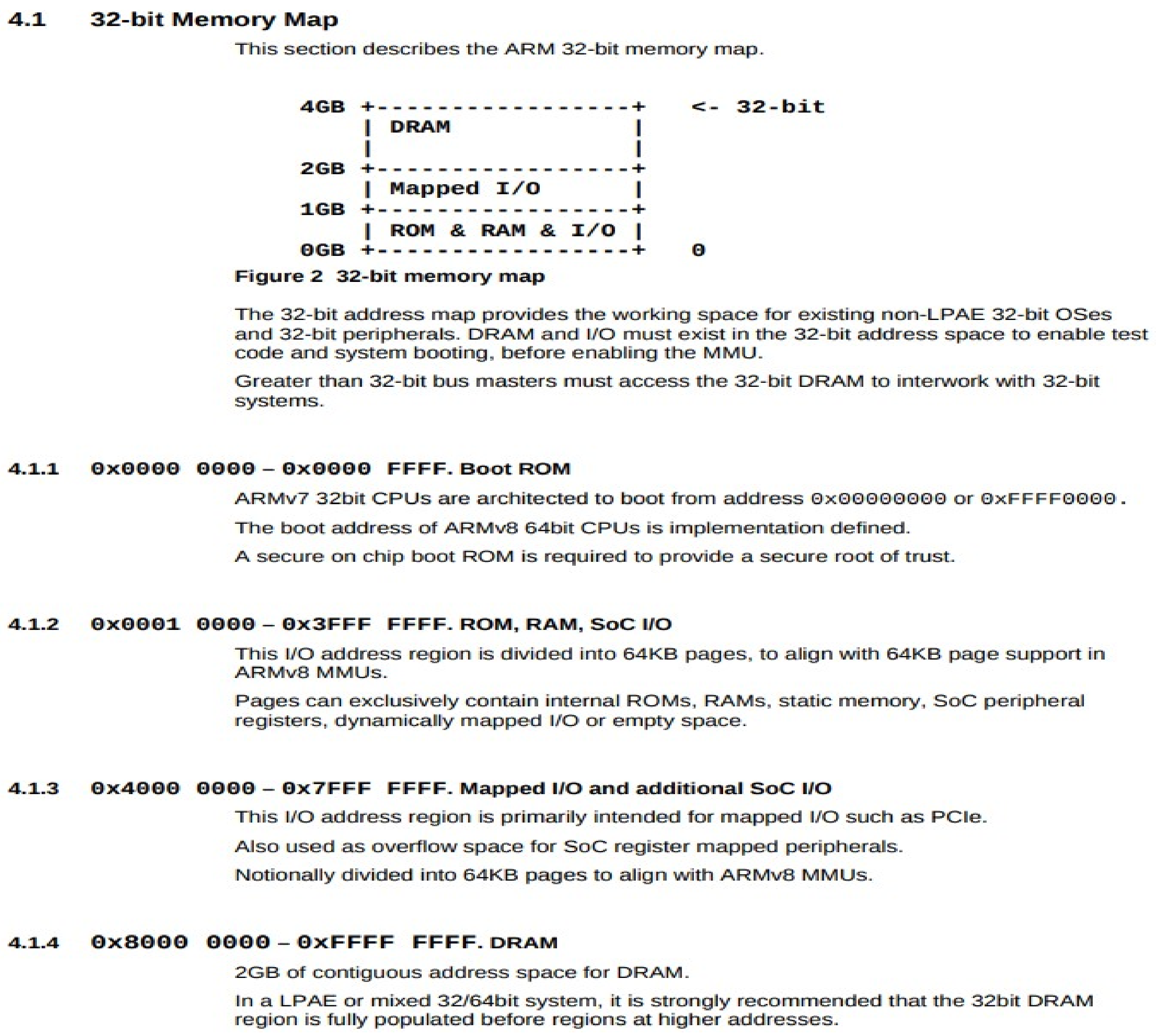
- Blue rectangles represent pages in the VMA range, while arrows represent page table entries mapping pages onto page frames. Some virtual pages lack arrows; this means their corresponding PTEs have the Present flag clear. This could be because the pages have never been touched or because their contents have been swapped out.
- In either case access to these pages will lead to page faults, even though they are within the VMA.

Heap allocation example


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

When the program asks for more memory via the brk() system call, the kernel simply updates the heap VMA and calls it good. No page frames are actually allocated at this point and the new pages are not present in physical memory.

Embedded systems specific



4.1 32-bit Memory Map

This section describes the ARM 32-bit memory map.

```
4GB +-----------------+    <- 32-bit
    | DRAM            |
    |                 |
2GB +-----------------+
    | Mapped I/O      |
1GB +-----------------+
    | ROM & RAM & I/O |
0GB +-----------------+    0
```

Figure 2 32-bit memory map

The 32-bit address map provides the working space for existing non-LPAE 32-bit OSes and 32-bit peripherals. DRAM and I/O must exist in the 32-bit address space to enable test code and system booting, before enabling the MMU.

Greater than 32-bit bus masters must access the 32-bit DRAM to interwork with 32-bit systems.

4.1.1 0x0000 0000 – 0x0000 FFFF. Boot ROM

ARMv7 32bit CPUs are architected to boot from address `0x00000000` or `0xFFFF0000`.

The boot address of ARMv8 64bit CPUs is implementation defined.

A secure on chip boot ROM is required to provide a secure root of trust.

4.1.2 0x0001 0000 – 0x3FFF FFFF. ROM, RAM, SoC I/O

This I/O address region is divided into 64KB pages, to align with 64KB page support in ARMv8 MMUs.

Pages can exclusively contain internal ROMs, RAMs, static memory, SoC peripheral registers, dynamically mapped I/O or empty space.

4.1.3 0x4000 0000 – 0x7FFF FFFF. Mapped I/O and additional SoC I/O

This I/O address region is primarily intended for mapped I/O such as PCIe.

Also used as overflow space for SoC register mapped peripherals.

Notionally divided into 64KB pages to align with ARMv8 MMUs.

4.1.4 0x8000 0000 – 0xFFFF FFFF. DRAM

2GB of contiguous address space for DRAM.

In a LPAE or mixed 32/64bit system, it is strongly recommended that the 32bit DRAM region is fully populated before regions at higher addresses.