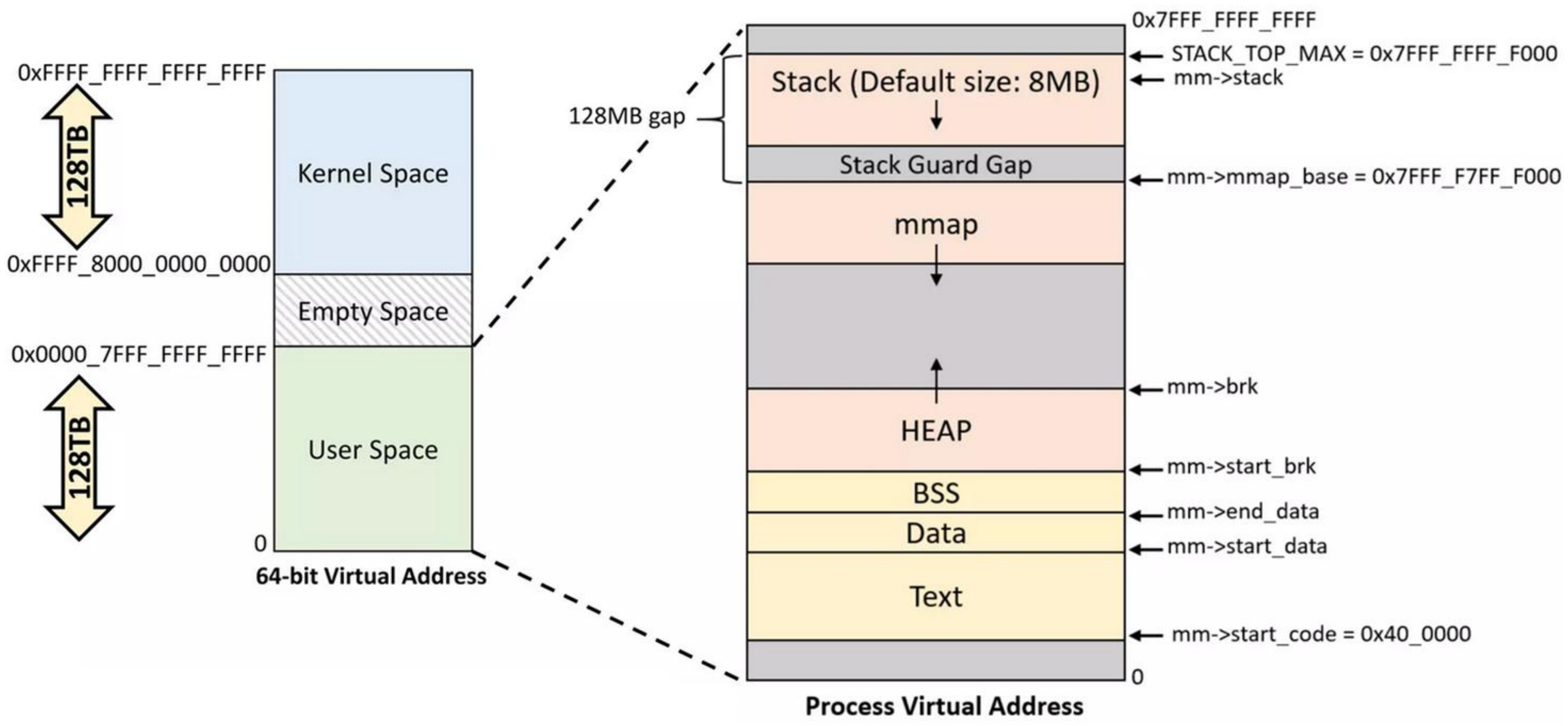
Process address space

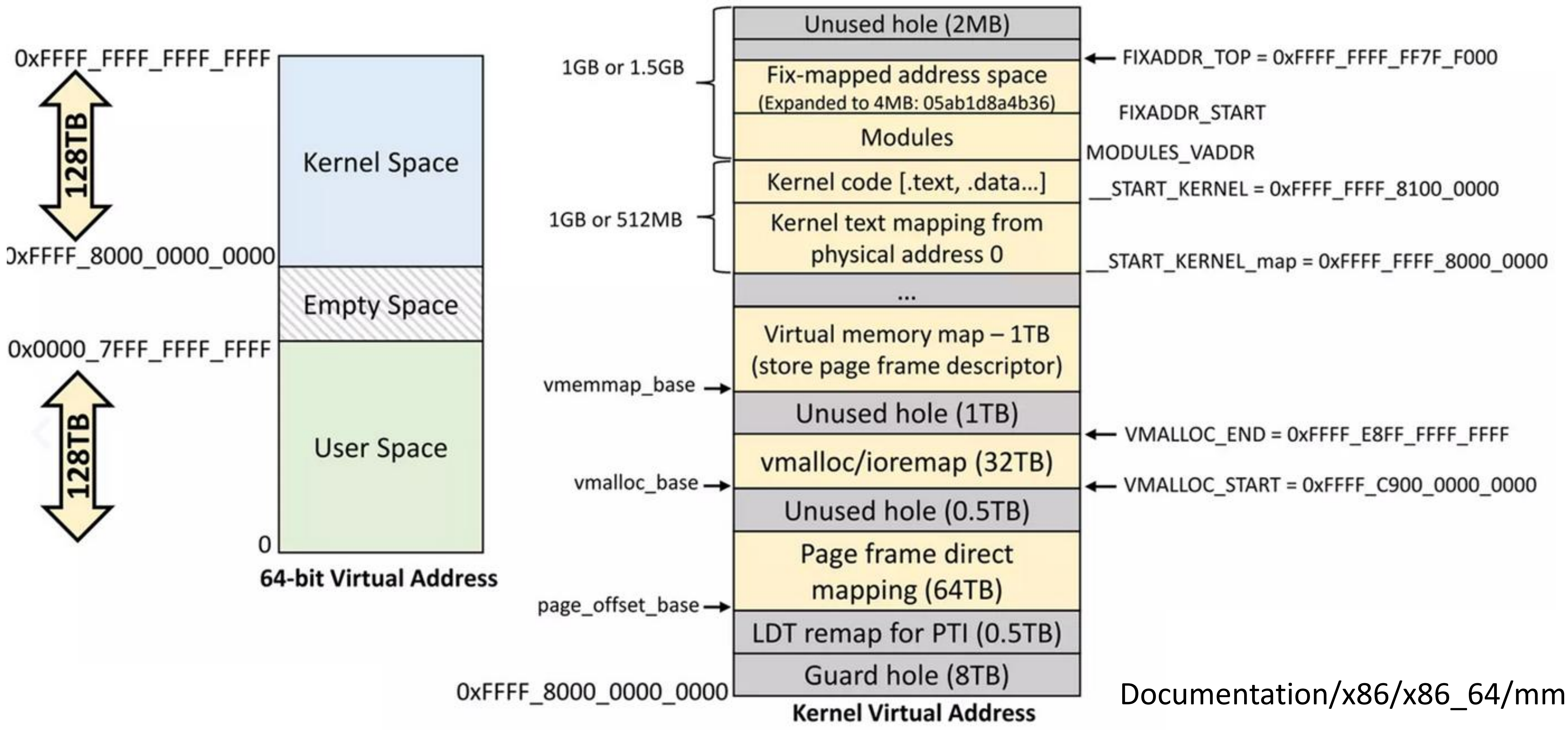
Table of contents

Process virtual address space in x86_64



(source: Adrian Huang, Process address space, 2022)

Kernel virtual address space in x86_64



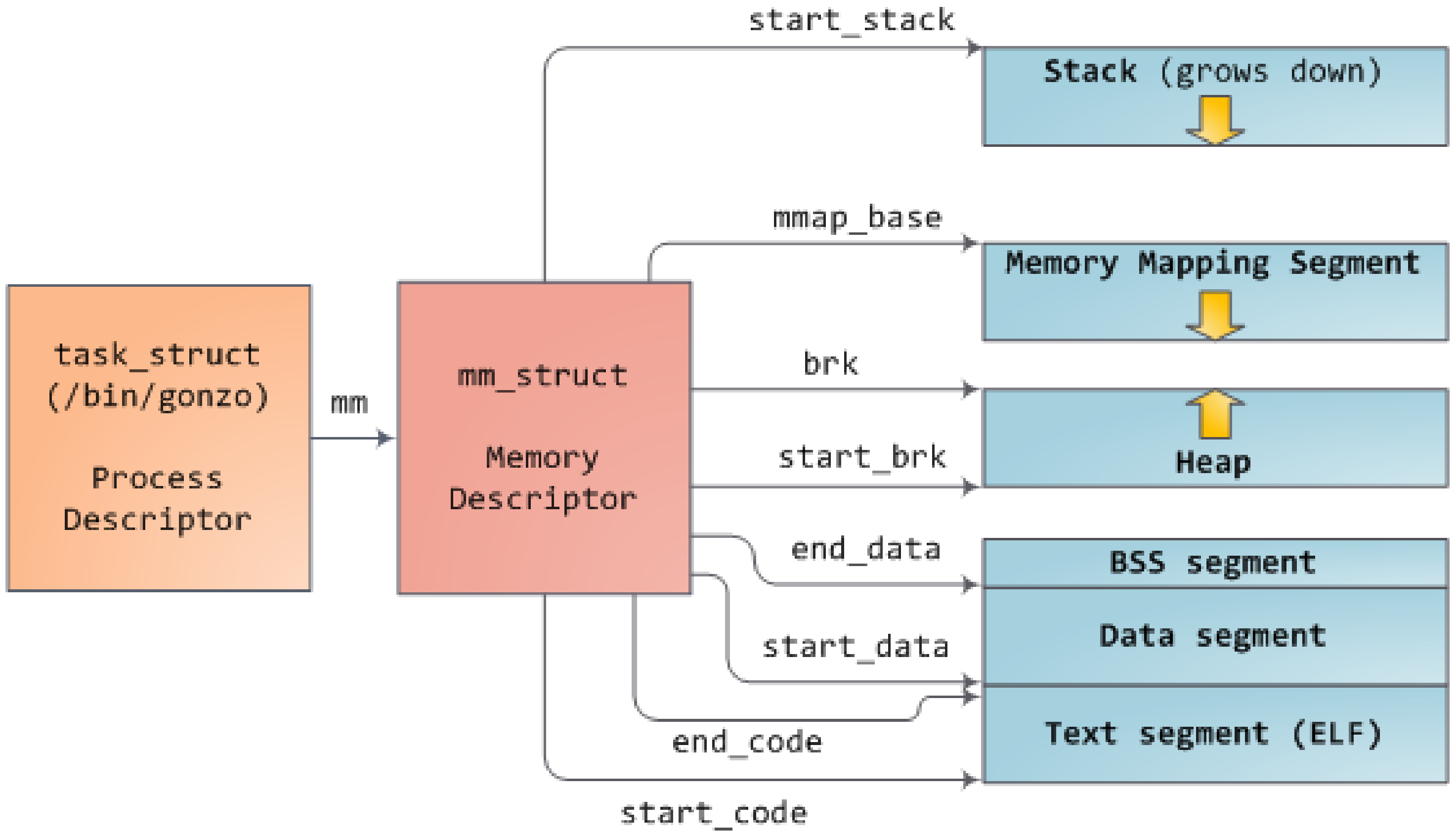
0xFFFF_FFFF_FFFF_FFFF
128TB
0xFFFF_8000_0000_0000
0x0000_7FFF_FFFF_FFFF
128TB
0
Kernel Space
Empty Space
User Space
64-bit Virtual Address
1GB or 1.5GB
1GB or 512MB
vmemmap_base
vmalloc_base
page_offset_base
0xFFFF_8000_0000_0000
Unused hole (2MB)
Fix-mapped address space
(Expanded to 4MB: 05ab1d8a4b36)
Modules
Kernel code [.text, .data...]
Kernel text mapping from physical address 0
...
Virtual memory map – 1TB (store page frame descriptor)
Unused hole (1TB)
vmalloc/ioremap (32TB)
Unused hole (0.5TB)
Page frame direct mapping (64TB)
LDT remap for PTI (0.5TB)
Guard hole (8TB)
Kernel Virtual Address
FIXADDR_TOP = 0xFFFF_FFFF_FF7F_F000
FIXADDR_START
MODULES_VADDR
__START_KERNEL = 0xFFFF_FFFF_8100_0000
__START_KERNEL_map = 0xFFFF_FFFF_8000_0000
VMALLOC_END = 0xFFFF_E8FF_FFFF_FFFF
VMALLOC_START = 0xFFFF_C900_0000_0000

Documentation/x86/x86_64/mm.rst

(source: Adrian Huang, Process address space, 2022)

Process memory descriptor

Each task_struct structure has a **mm** field which points to the **process memory descriptor** – a **mm_struct** structure describing the process address space.



The task_struct and mm_struct (source: Duarte, Software Illustrated)

Process memory descriptor

```
#define VMACACHE_BITS 2
#define VMACACHE_SIZE (1U << VMACACHE_BITS)

struct  vmacache {
      u64 seqnum;
      struct vm_area_struct *vmas[VMACACHE_SIZE];
};

struct task_struct {
     ....
      struct mm_struct     *mm;
     /* Per-thread vma caching: */
     struct vmacache       vmacache;
    .....
};
```

```
struct mm_struct {
   struct  maple_tree mm_mt;
   u64   vmacache_seqnum;
   unsigned long (*get_unmapped_area)
                 (struct file *filp, unsigned long addr,
                  unsigned long len, unsigned long pgoff, unsigned long flags);
   unsigned long  mmap_base;
   unsigned long  task_size;
   pgd_t  *pgd;
   atomic_t   mm_users;
   atomic_t   mm_count;
   int  map_count;
   spinlock_t   page_table_lock;
   struct rw_semaphore  mmap_lock;

   unsigned long  start_code, end_code;
   unsigned long  start_data, end_data;
   unsigned long  start_brk, brk, start_stack;
   unsigned long  arg_start, arg_end, env_start, env_end;
   unsigned long  total_vm;
   .....
};
```

Process memory descriptor

mm_mt: pointer to the **maple tree** root (replaced red-black tree **mm_rb** and linear list **mmap**);

vmacache_seqnum: per-thread VMA cache (*why per-thread? why seqnum?*),

mmap_base: points to the place from which the dynamic memory area mapping starts;
the **get_unmapped_area()** function finds the right place for the new mapping,

task_size: the size of the process address space (usually TASK_SIZE),

pgd: Page Global Directory,

mm_users: number of processes sharing this structure (*who can share?*),

map_count: number of VMA areas,

mmap_lock: semaphore to protect the structure,

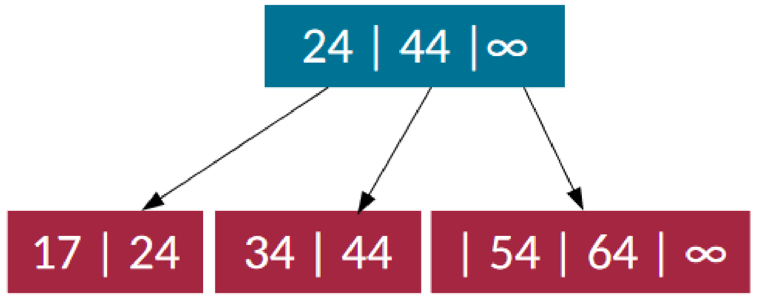
start_code, **end_code**: the beginning (end) address of the code section,

start_data, **end_data**: the beginning (end) address of the data section,

start_brk, **brk**: the beginning (end) address of the heap area,

total_vm: total number of pages used by the process.

vm_area_struct – maple tree



- **Maple tree** replaced **red-black tree** and a **list**.
- They belong to the **B-tree family**, their nodes can contain **more than two elements** — up to 16 elements in leaf nodes, or ten elements in internal nodes.
- There is less need to create new nodes, as nodes may include empty slots that can be filled over time without additional allocations.
- Each node holds keys, called **pivots**, that separate the node into subtrees. A subtree before a given key contains only values lower or equal to the key, while subtree after the key contains only values higher than the key (and lower than the next key). It speeds up searching.
- Each node requires at most **256 bytes**, which is a **multiple of popular cache line sizes**. The increased number of elements in a node and the cache-aligned size means fewer cache misses when traversing the tree.
- Maple trees can track gaps, store ranges, and be implemented using **read-copy-update** (RCU).

- The Linux Maple Tree, Metthew Wilcox, Embedded Linux Conference, 2019.
- Finer-grained kernel address-space layout randomization, Jake Edge, February 2020.
- Maple tree RFC patches sent out as new data structure to help with Linux performance, Michael Larabel, December 2020.
- Introducing maple trees, Marta Rybczyńska, LWN, February 2021.
- Maple Tree, Liam Howlett, Linux Plumbers Conference, 2022.

Per-thread VMA cache

There are **sequence numbers** stored in both **struct mm_struct** (one per address space) and in **struct task_struct** (one per thread).

The purpose of the sequence numbers is to ensure that the cache **does not return stale results**.

Any **change** to the **address space** (the addition or removal of a VMA, for example) causes the **per-address-space sequence number to be incremented**.

Every attempt to **look up an address in the per-thread cache** first checks the sequence numbers; if they do not match, the cache is deemed to be invalid and will be reset.

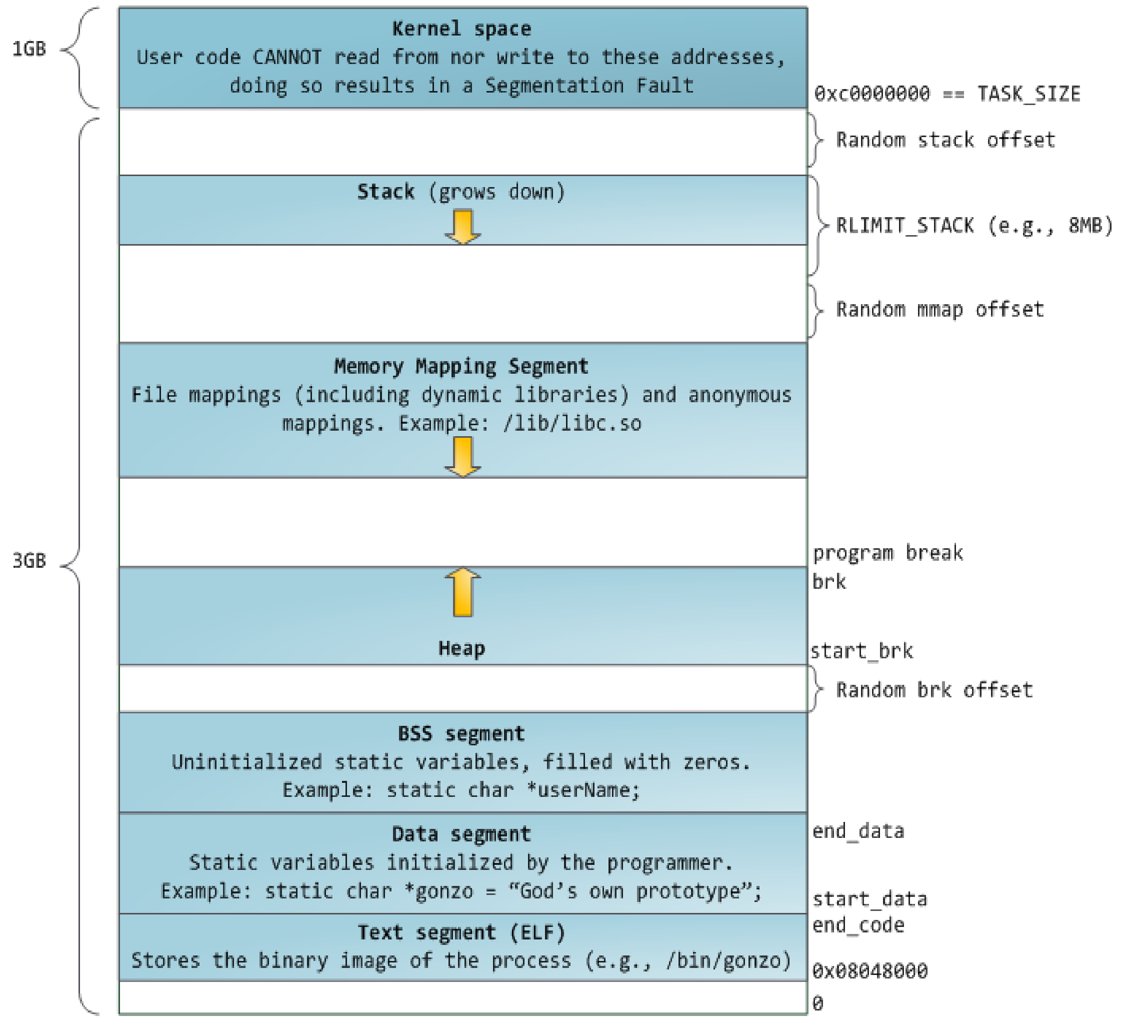
Address-space changes are relatively rare in most workloads, so the invalidation of the cache should not happen too often.

Every call to **find_vma()** first does a **linear search** through the **cache.** Should the VMA be found, the work is done; otherwise, a **traversal of the maple tree** will be required. In this case, the result of the **lookup** will be **stored back** into the **cache**. That is done by **overwriting** the **entry indexed** by the **lowest bits** of the **page-frame number** associated with the **original virtual address**.

Change in cache hit ratio depends on the workload: from **51%** to **73%**, but also from **1%** to **99,97%** (multithreaded web server).

Additional reading: Optimizing VMA caching (Jonathan Corbet, May 2014)

Process address space



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;
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

Process address space (source: Duarte, Software Illustrated)

The **BSS segment** and the **data segment** contain static (global) variables in C: BSS – **uninitialized**, data segment – **initialized**.

The BSS area is **anonymous**, it does not map any file.

The data segment is **not anonymous**, it maps the fragment of the binary program image. This is **private mapping**, i.e. changes in this memory area are not transferred to the file.

New mappings are created starting from the address **mm_struct→ mmap_base.**

The layout of the process address space can be read from the **/proc/<process_pid>/maps** file or by function **pmap <process_pid>**.

All process memory descriptors are linked in the list – using **mmlist** field (*why do we need a maple tree and a list?*).

Process address space

In the past, the **initial virtual addresses of the segments** had the same value for virtually all processes. Now, **randomizing the address space** is popular for **security** reasons. Linux randomizes the **stack**, the file **mapping segment**, and the **heap**, adding offset to their start address.

Attempting to place more data on the stack than the space allocated results in a page fault that is handled by the **expand_stack()** function, which checks whether the stack extension is acceptable. If the stack size **does not exceed RLIMIT_STACK** (usually 8 MB), then the stack is expanded, otherwise segmentation fault is generated. **The stack is only expanded, never decreased**. A dynamic stack extension is **the only case** where a reference to an unmapped memory area can be properly handled.

Below the stack there is an area for mapping files with the **mmap()** function.

You can also create **anonymous mapping** that does not match any files. For example, the C library supports a **large memory block request** (larger than MMAP_THRESHOLD, 128 KB by default) using **malloc()** to create such an anonymous mapping instead of allocating memory on the heap.

The system function **brk()** is used to expand the **heap**.

vm_area_struct

The process address space consists of many **disjoint contiguous memory areas**. Each of them is described by the **vm_area_struct** structure.

```
struct vm_area_struct {
        unsigned long  vm_start;
        unsigned long  vm_end;
        struct mm_struct   *vm_mm;
        pgprot_t  vm_page_prot;
        unsigned long  vm_flags;
        /*
         * For areas with an address space and backing store,
         * linkage into the address_space->i_mmap
         * interval tree
         */
        struct {
            struct rb_node  rb;
            unsigned long  rb_subtree_last;
        } shared;
```

vm_start: VMA start address,

vm_end: address of the first byte AFTER the end of VMA,

vm_mm: the address space to which this VMA area belongs,

vm_page_prot: access rights to this VMA,

vm_flags: flags defining the properties of the area

Relation to **address_space** will be covered later.

The number of such areas for one process usually fluctuates within **6**, although in certain situations it can reach **3000**.

vm_area_struct – continued

```
  /*
   * A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma list,
   * after a COW of one of the file pages.  A MAP_SHARED vma
   * can only be in the i_mmap tree.  An anonymous MAP_PRIVATE, stack
   * or brk vma (with NULL file) can only be in an anon_vma list.
   */
  struct list_head anon_vma_chain;    /* Serialized by mmap_lock & page_table_lock */
  struct anon_vma *anon_vma;          /* Serialized by page_table_lock */
  struct vm_operations_struct *vm_ops;
  unsigned long vm_pgoff;             /* Offset (within vm_file) in PAGE_SIZE units */
  struct file *vm_file;               /* File we map to (can be NULL) */
   ......
};
```

anon_vma_chain and **anon_vma**: used to handle **shared** pages from **anonymous** mappings,

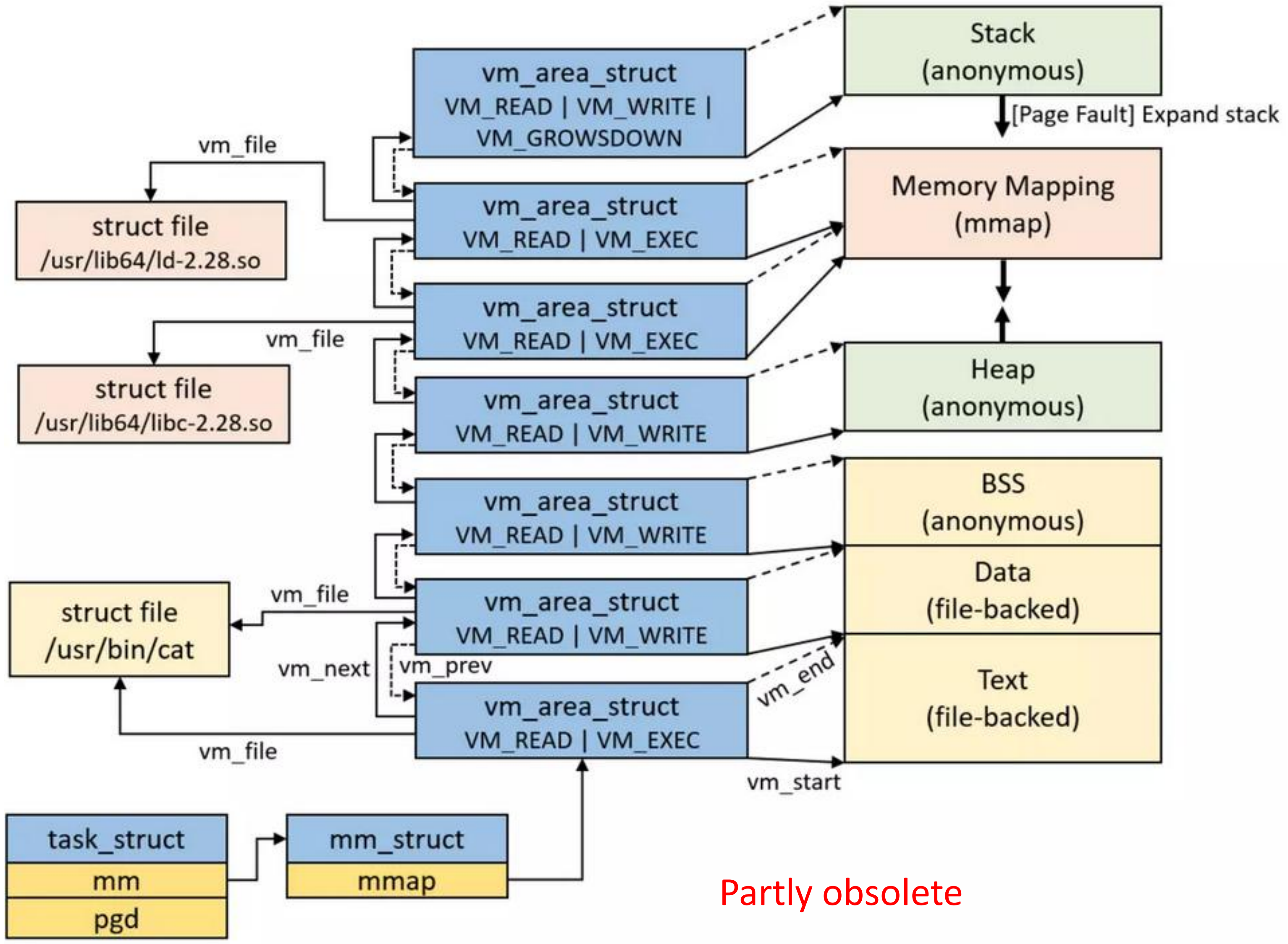
vm_ops: a set of pointers for functions performing **operations** on this VMA

vm_pgoff: if the area belongs to a **mapped file**, it is an offset in the file (in the number of pages),

vm_file: if the area belongs to a **mapped file**, then this is the pointer to this file,

Process address space

```
jmd@students:~$ ps
    PID TTY          TIME CMD
2513716 pts/51   00:00:00 bash
3322045 pts/51   00:00:00 ps
jmd@students:~$ cat /proc/2513716/maps
```



Partly obsolete

(source: Adrian Huang, Process address space, 2022)

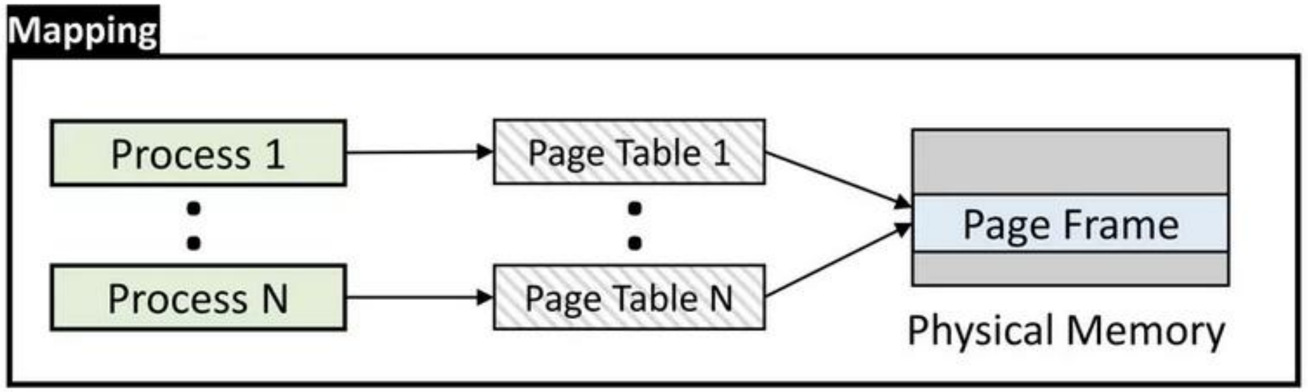
vm_area_struct – reverse mapping

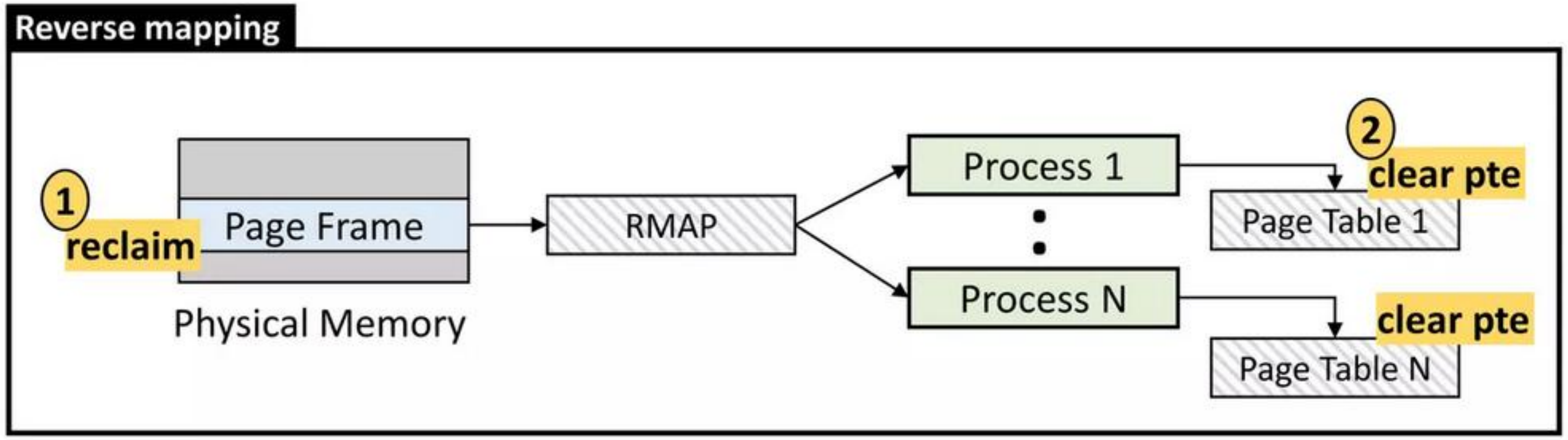
Structures used for **reverse mapping**. Depending on the origin of the area, they are:

- **Tree of intervals**, containing areas included in various processes, coming from the **same file**. The maple tree binds the areas belonging to one process. You also need a **reverse look at the processes to which a specific piece of the file has been mapped**. This is done by the **struct shared**.
- **List of anonymous areas** that match the same pages in memory if the area comes from anonymous mapping. Fields **anon_vma_chain** and **anon_vma** are used for that (described in the include/linux/rmap.h file).

  The purpose of this structure is to make it easier for the kernel to **free memory when swapping is required**. By following the list, the kernel can quickly find **all mappings for a given page**, unmap them, and swap the page out.

Mapping & reverse mapping

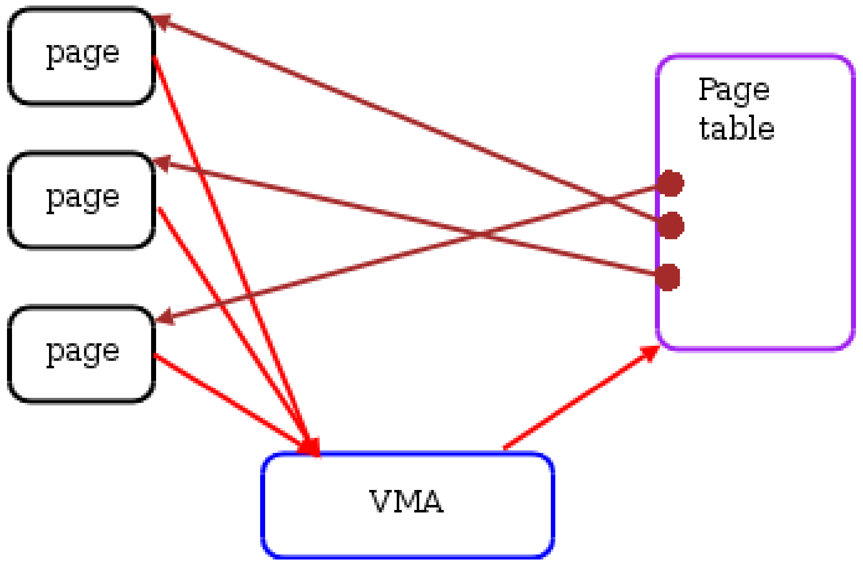


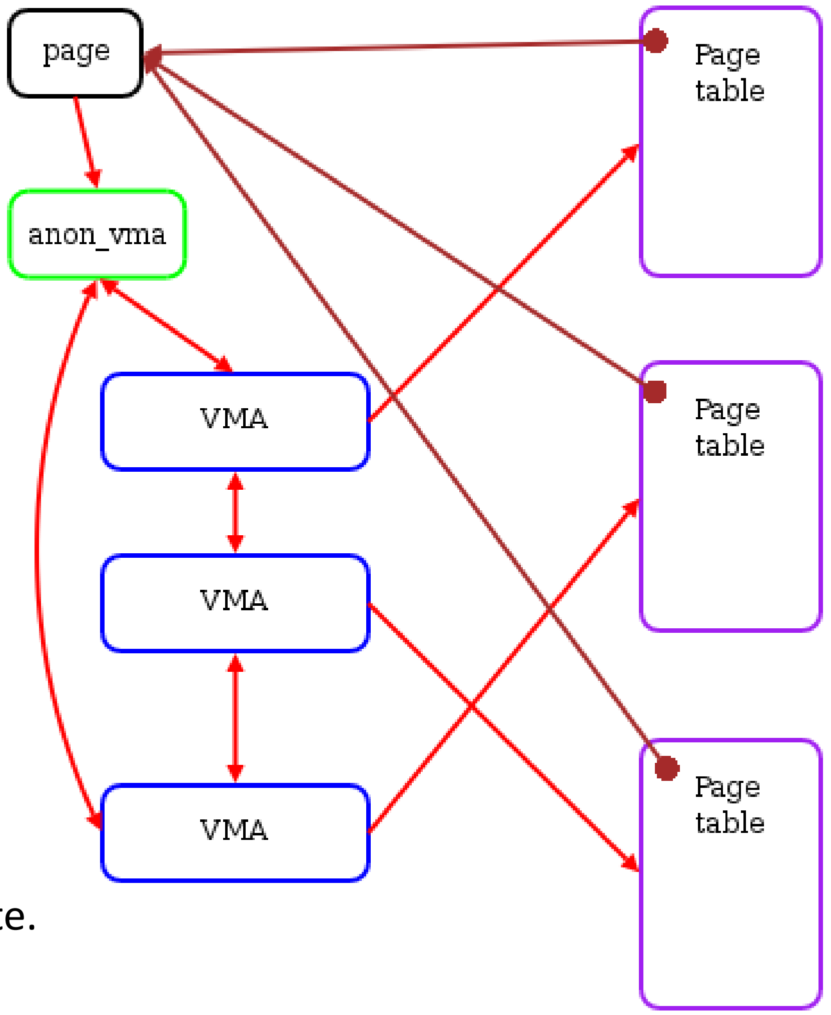


(source: Adrian Huang, Reverse mapping (rmap), 2022)

Anonymous reverse mapping

A **process** has several **non-shared**, **anonymous pages** in the same virtual memory area.



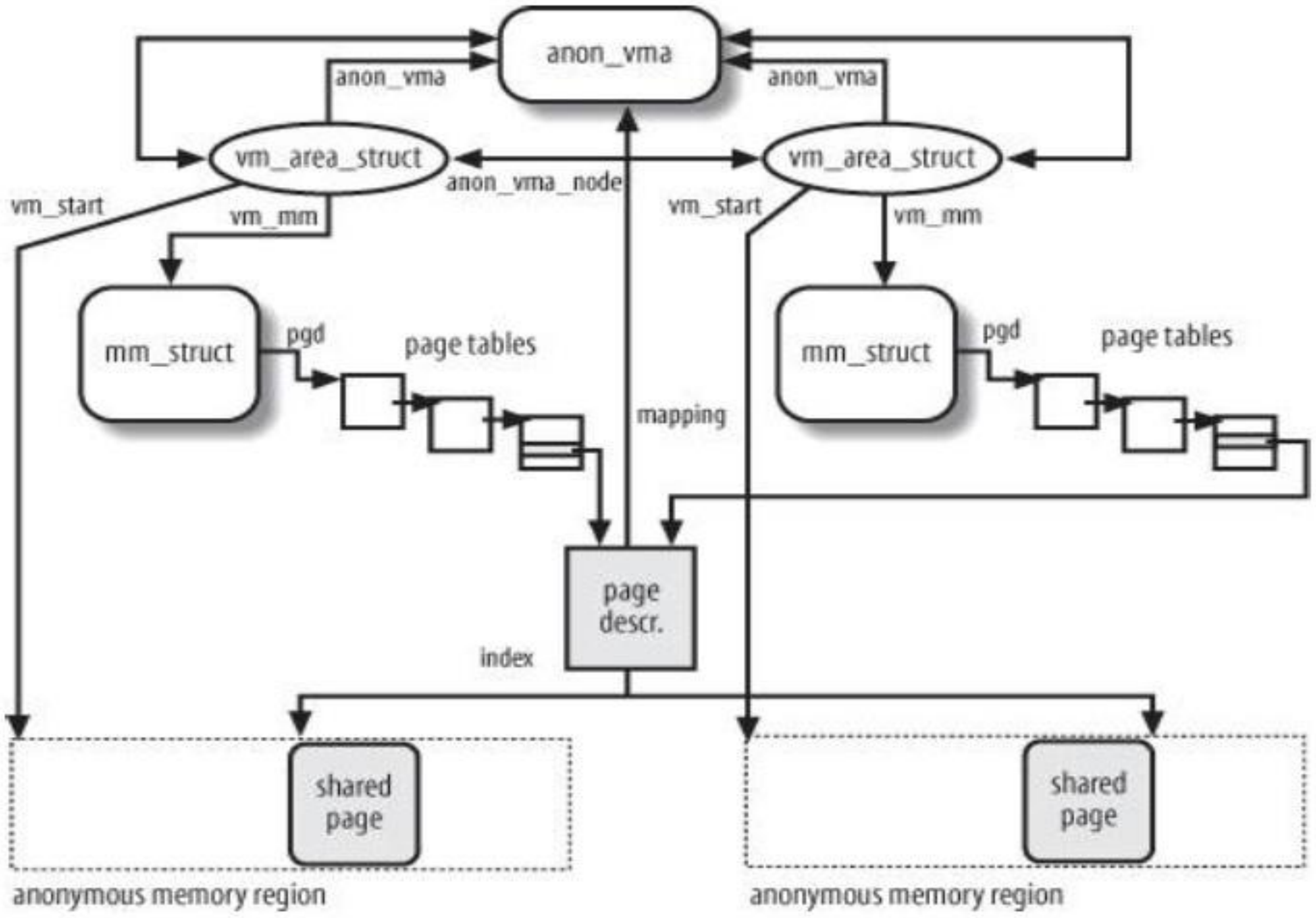


When the **process forks**, there will be **multiple page tables** pointing to the same **anonymous pages** and a single VMA pointer will no longer be adequate.

The **new structure anon_vma** takes care of that.

Anonymous reverse mappings (source: Jonathan Corbet, January 2004)

Anonymous reverse mapping



Anonymous reverse mappings (source: Bovet, Cesati, Understanding the Linux Kernel

vm_operations

Each virtual memory area can be assigned a **set of operations specific to pages of this area** when writing them to disk, reading, handling page faults, etc.

When performing such operations, Linux first checks whether the corresponding function is defined and if so, it is performed.

If not, then the **default operation** is performed.

In practice, special operations are defined for **shared memory pages** and for **disk file images**.

```
struct vm_operations_struct {
    void (*open)(struct vm_area_struct * area);
    void (*close)(struct vm_area_struct * area);
    int (*fault)(struct vm_fault *vmf);
    ...
```

open() – invoked when **adding** the area to the process address space;
close() – invoked when **removing** an area from the process address space;
fault() – invoked by the **page fault** handler.

Allocation and release of process address space areas

Functions associated with the **allocation** and **release** of process address space areas (only some are listed):

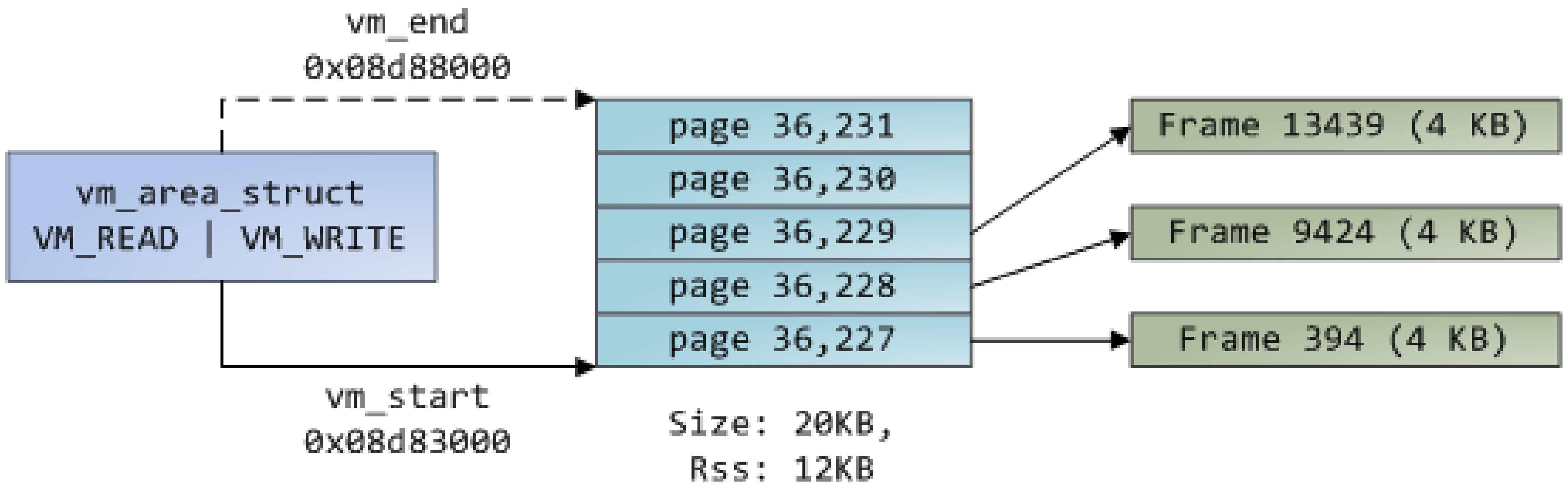
- **brk()** – changes the size of the process heap,
- **execve()** – loads a new executable file, changes the address space of the process,
- **_exit()** – terminates the process and destroys its address space,
- **fork()** – creates a new process, and so the new address space,
- **mmap()** – maps the file to memory, increasing the address space of the process,
- **munmap()** – destroys the file mapping, reduces the address space of the process,
- **shmat()** – attaches a shared memory area,
- **shmdt()** – disconnects the shared memory area.

Each area consists of **pages** with **consecutive numbers**.

When the kernel adds the process page, in the corresponding **position of the page table** it sets the flags according to the contents of the **vm_page_prot** field in the corresponding VMA (there are some exceptions such as **Copy On Write**).

The relationship of **vm_page_prot** with **protection bits in the page table** position is described in supplement.

Extending the process heap



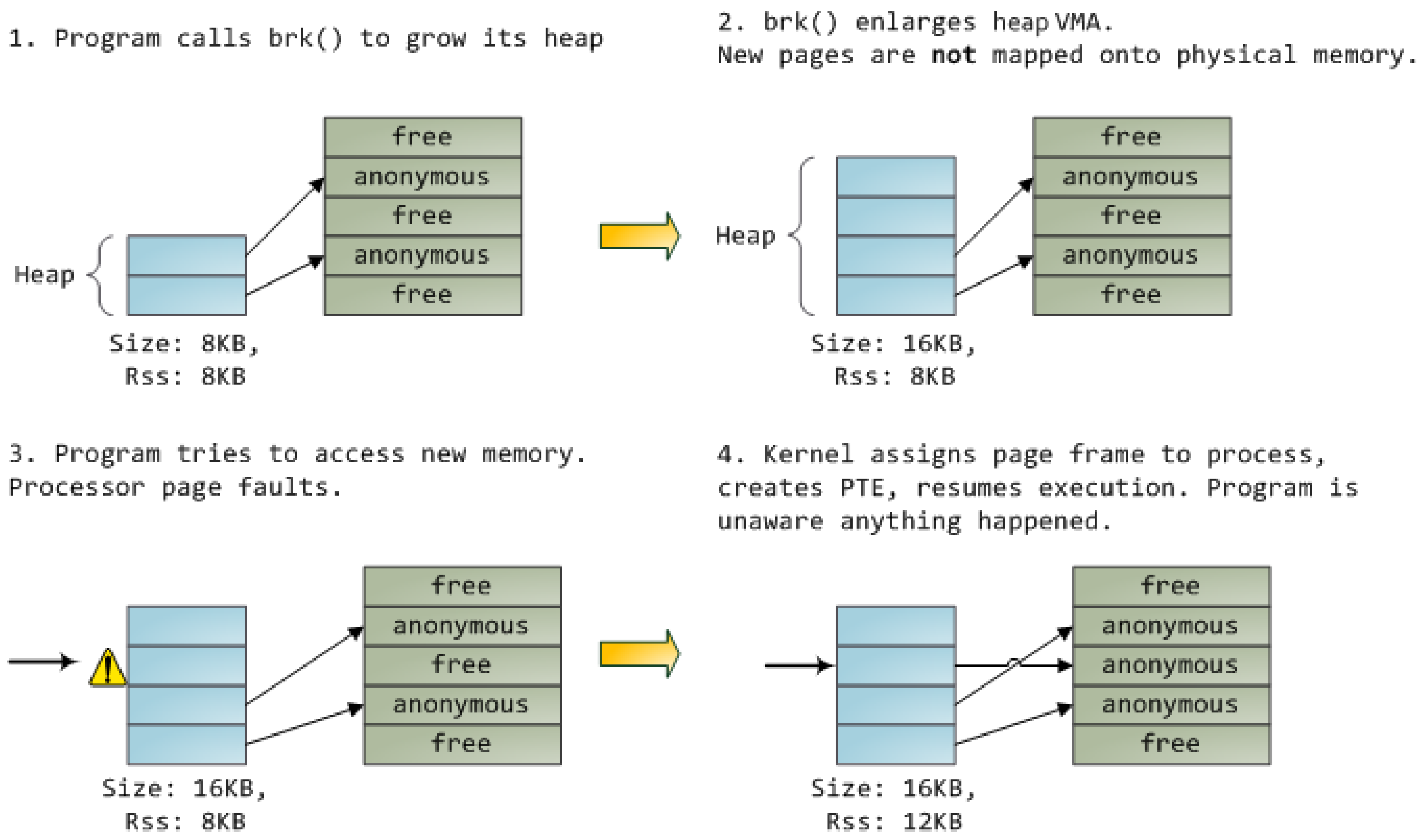
Heap of the user process (source: Duarte, Software Illustrated)

Extending the process heap

Blue rectangles are **pages in the VMA area**, arrows are the positions of **page tables** that map pages to page frames. No arrow means a flag **Present** equal to zero. The page has never been brought to memory, or has been swapped out. Access to these pages will generate a **page fault**, although the pages are described by some VMA.

VMA is a form of **contract** between the **program** and the **kernel**.

The program asks for memory allocation, the kernel pretends to have allocated (only PTE knows the truth), the actual work will be performed in the future by the **page fault handler**.


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

Memory allocation on the heap (source: Duarte, Software Illustrated)

Mapping files to memory

The **do_mmap()** function creates a new linear address interval for the process.

It does not have to be related to the creation of a new VMA (adjacent areas with the same permissions are combined into one).

If the parameters **file** and **offset** are different from NULL, they indicate the area of the **file** with which the newly created memory area will be associated.

Otherwise, an **anonymous** area will be created.

The **do_munmap()** function is used to remove an existing mapping from the process's address space.

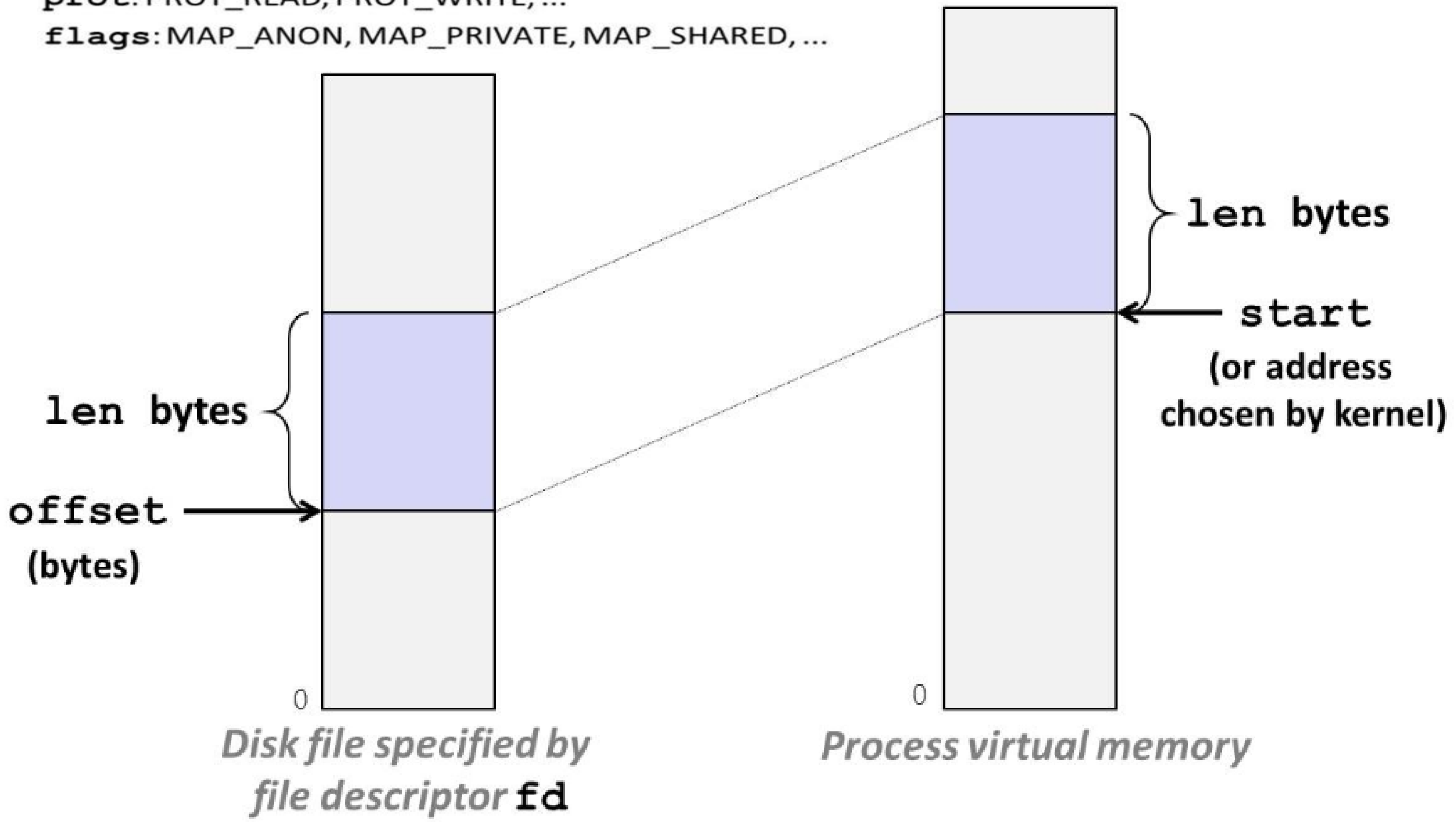
```
unsigned long do_mmap(struct file *file, unsigned long addr,
    unsigned long len, unsigned long prot, unsigned long flags, unsigned long pgoff, ...)

int do_munmap(struct mm_struct *mm, unsigned long start, size_t len, struct list_head *uf)
```

User-Level Memory Mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

`prot`: PROT_READ, PROT_WRITE, ...

`flags`: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...



len bytes
offset
(bytes)
0
Disk file specified by file descriptor fd
len bytes
start
(or address chosen by kernel)
0
Process virtual memory

Creating the process address space fork()

It would seem that the natural way to **duplicate the process** is the following scheme: the **data** area of the parent process is **copied**, and the **code** area is **shared** by both processes, parent and child.

Such an algorithm is very **expensive** for **longer programs** that can have a part of the address space on disk at a given moment.

In older Unix systems, there were two functions: **fork()** and **vfork()**.

The function **fork()** worked as described, and **vfork()** was called only if the next child statement was to call **exec()**. In the second case, the **address space was not copied.**

Linux offers better solution. A **copy on write** is used, the address space is copied only if one of the two processes tries to write something. Only the read-only memory, e.g. the program code, will always be shared.

The file **include/linux/syscalls.h** contains the following header declarations:

```
asmlinkage long sys_fork(void);
asmlinkage long sys_vfork(void);
asmlinkage long sys_clone(...);
```

Creating the process address space

fork()

/kernel/fork.c
contains
the following definitions
(simplified form).

```
SYSCALL_DEFINE0(fork)
{
        struct kernel_clone_args args = {
            .exit_signal = SIGCHLD,
        };
        return kernel_clone(&args);
}
SYSCALL_DEFINE0(vfork)
{
        struct kernel_clone_args args = {
            .flags                  = CLONE_VFORK | CLONE_VM,
            .exit_signal    = SIGCHLD,
        };
        return kernel_clone(&args);
}
SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp,
             int __user *, parent_tidptr, int __user *, child_tidptr, unsigned long, tls)
{
        struct kernel_clone_args args = {
        ...
        };
        return kernel_clone(&args);
}
/*
 * Ok, this is the main fork-routine.
 * It copies the process, and if successful kick-starts
 * it and waits for it to finish using the VM if required.
 */
pid_t kernel_clone(struct kernel_clone_args *args)
```

Creating the process address space fork() → copy_process()

The actual work is done in **kernel_clone()**, which calls **copy_process(),** and after return **wakes up** and **starts a new process**.

Basically, the **child process** should be done first (*why?*)

The **copy_process()** function performs the following actions:

1. Invokes **dup_task_struct()**, which creates a **new kernel mode stack**, a **new thread_info structure**, and a **new task_struct**.
2. Clears or sets new values for different fields in the **process descriptor**.
3. Sets the status **TASK_UNINTERRUPTIBLE** for the new process.
4. Invokes **alloc_pid**, to assign **PID** to the new process.
5. Depending on the flags set, shares or duplicates

– open file descriptors (**copy_files()**),

– file system context (**copy_fs()**),

– signal handlers (**copy_sighand()**),

– structures describing virtual memory of the parent process (**copy_mm()**).

Creating the process address space
fork() → copy_process()

```
/* copy all the process information */
        if ((retval = copy_semundo(clone_flags, p)))
                goto bad_fork_cleanup_audit;
        if ((retval = copy_files(clone_flags, p)))
                goto bad_fork_cleanup_semundo;
        if ((retval = copy_fs(clone_flags, p)))
                goto bad_fork_cleanup_files;
        if ((retval = copy_sighand(clone_flags, p)))
                goto bad_fork_cleanup_fs;
        if ((retval = copy_signal(clone_flags, p)))
                goto bad_fork_cleanup_sighand;
        if ((retval = copy_mm(clone_flags, p)))
                goto bad_fork_cleanup_signal;
        if ((retval = copy_namespaces(clone_flags, p)))
                goto bad_fork_cleanup_mm;
        if ((retval = copy_io(clone_flags, p)))
                goto bad_fork_cleanup_namespaces;
        retval = copy_thread(clone_flags, stack_start, stack_size, p, regs);
        ...
```

Creating the process address space fork() → clone_flags

Interesting flags **in** **clone_flags** include

CLONE_FILES,

CLONE_SIGHAND,

CLONE_VM,

which determine whether a given resource should be **copied** or **cloned** (that is, properly **shared** by both processes).

For example, if you call **kernel_clone()** with the **CLONE_FILES** flag set, only the **reference count** to the files structure that will be shared is **increased**.

```
#define CSIGNAL          0x000000ff  /* signal mask to be sent at exit */
#define CLONE_VM         0x00000100  /* set if VM shared between processes */
#define CLONE_FS         0x00000200  /* set if fs info shared between processes */
#define CLONE_FILES      0x00000400  /* set if open files shared between processes */
#define CLONE_SIGHAND    0x00000800  /* set if signal handlers and blocked signals shared */
#define CLONE_PTRACE     0x00002000  /* set if we want to let tracing continue on the child too */
#define CLONE_VFORK      0x00004000  /* set if the parent wants the child to wake it up on mm_release */
#define CLONE_PARENT     0x00008000  /* set if we want to have the same parent as the cloner */
#define CLONE_THREAD     0x00010000  /* Same thread group? */
...
```

Creating the process address space fork() → copy _mm()

Kernel_clone() passes **clone_flags** to the function **copy_mm()**.

- **CLONE_VM** flag is **set**, the function **increases** the **reference count** for the **mm_struct** and **copies** the pointer to that structure. Both processes have the right to **write** to the same memory.

- **CLONE_VM** flag is **cleared**, it creates a **new structure** describing the process virtual memory (structure **mm_struct** assigned by the function **allocate_mm()**); creates a **new page directory** (**mm_init() → pgd_alloc()**), fills it with zeros, and the space 3-4 GB maps to the kernel memory (but remember about PTI).

  Then **copy_mm()** copies the entire **maple tree** from the parent process to the child process (function **dup_mmap()**). This is not just a copy, other actions must be done as well.

  If the area comes from a **memory mapped file**, only the **reference count** for the pages in this area is **increased**, and the area itself is added to the structure of the **shared areas**.

  If it is a **normal area**, the function **copy_page_range()** is called. First, it checks if we can **write** in the area and whether it is **not shared**. Then, in the loop **for each page** belonging to the area, the **RW flag is set to zero**. Then, the **page reference count increases**.

  If one of the processes wants to save something to such a page, a **page protection fault** will be reported. The system will check if the **vm_area_struct** to which the page is assigned has a **write flag set** and if so, the **page will be copied**.

  In the case of the **code area**, only the **reference count** of the memory **increases**, which means that it is released only after the two processes have finished their work.

Implementation of threads and different variants of clone()

fork() is implemented as a system function call **clone()** in which the parameter flags specifies both **SIGCHLD** and **all the cloning flags zeroed**, and the parameter that is to point to the child stack has the value of the **current stack pointer of the parent process**.

The **parent** process and the **child** process **share the user-mode stack temporarily**, but thanks to the **COW** mechanism, they will get **separate copies** of the **stack** at the first attempt to **write**.

vfork() is implemented as a system function call **clone()** in which the flags parameter specifies both **SIGCHLD** and flags **CLONE_VM** and **CLONE_VFORK**, and the parameter that is to point to the child stack has the value of the **current stack pointer of the parent process**.

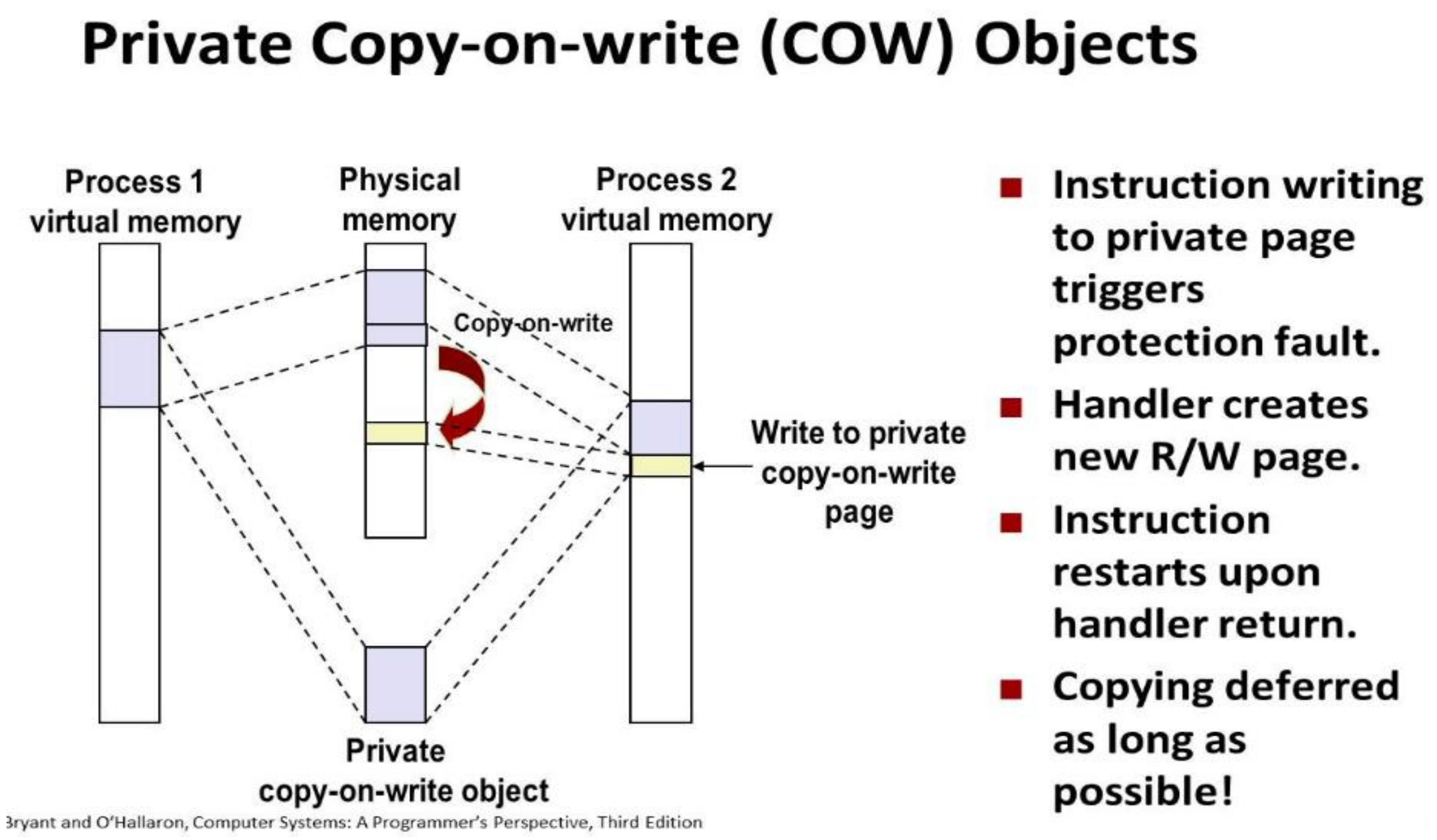
vfork() does **not copy** the **table entries** of the parent process, a child process **shares** the **address space** with the parent process.

The child process is performed as the only thread in the address space of the parent process, which is **blocked until the child completes exec()** or **exit()**, to prevent the parent process from erasing the data needed for the child process.

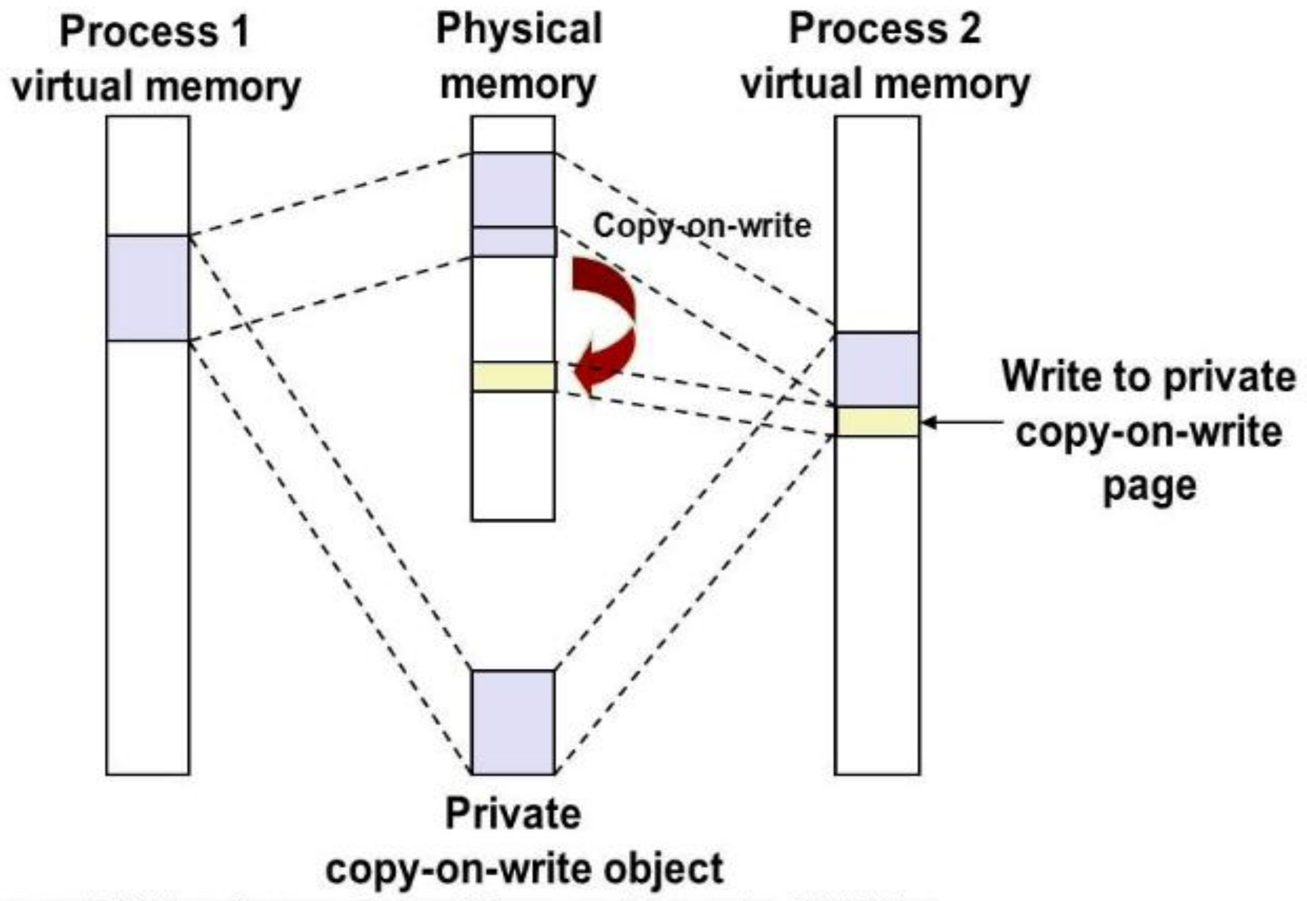
Nowadays, when **fork()** uses **COW** and starts the **child BEFORE** the **parent** process, the only profit from **vfork()** is to **not copy** the **page tables**.

In the future, Linux will probably make **COW** for **page tables**, and then there will be no profit.

COW in fork()



Private Copy-on-write (COW) Objects



- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Handling page faults

The kernel must distinguish the exceptions caused by **programming errors** from referring to a **page belongs** to the **process address space**, but has not yet been allocated.

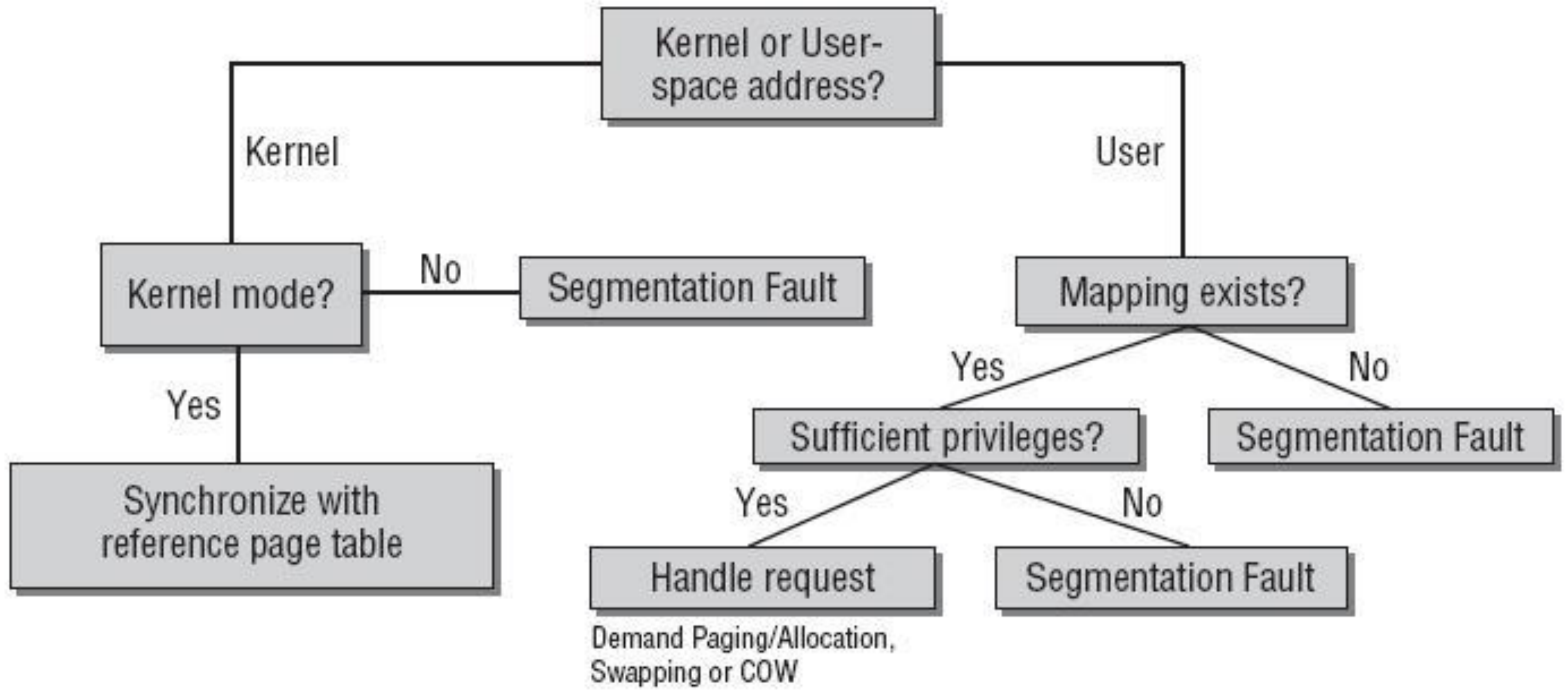
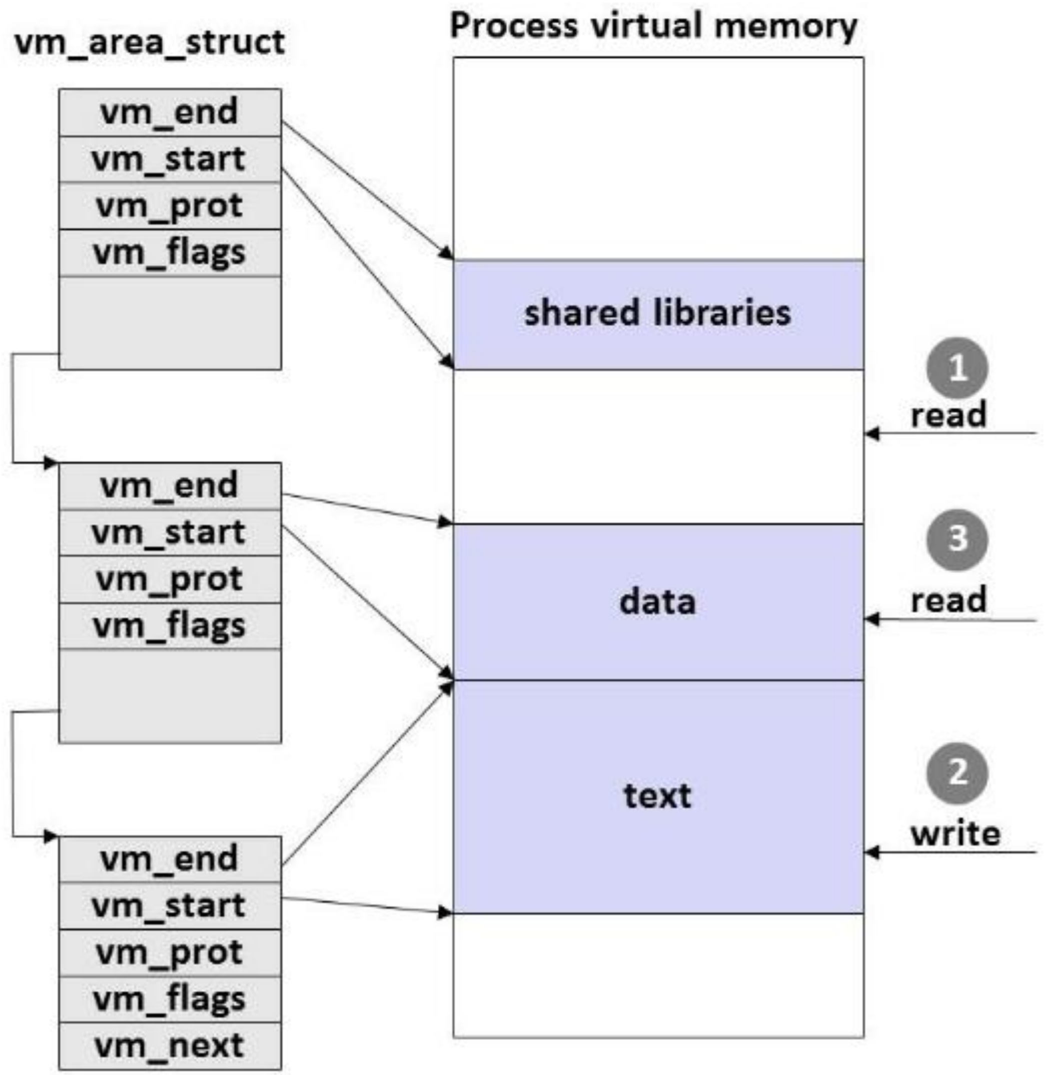


Figure 4-17: Potential options for handling page faults.

Options for handling page faults (source: Mauerer, Professional Linux Kernel Architecture)

Linux Page Fault Handling



Segmentation fault:
accessing a non-existing page

Normal page fault

Protection exception:
e.g., violating permission by writing to a read-only page (Linux reports as Segmentation fault)

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Handling page faults

The error handling of **page fault** largely depends on architecture.

For i386, the wrong address (address) is passed to page fault handler **do_page_fault()** via register **cr2**.

After verifying that a valid **memory context exists**, a **find_vma()** function is invoked, which looks through the **maple tree** for the appropriate memory area descriptor (**vm_area_struct**).

```
extern struct vm_area_struct * find_vma(struct mm_struct * mm, unsigned long addr);
```

If **no VMA** exists for the given address, the **SIGSEGV** (segmentation fault) signal is sent.

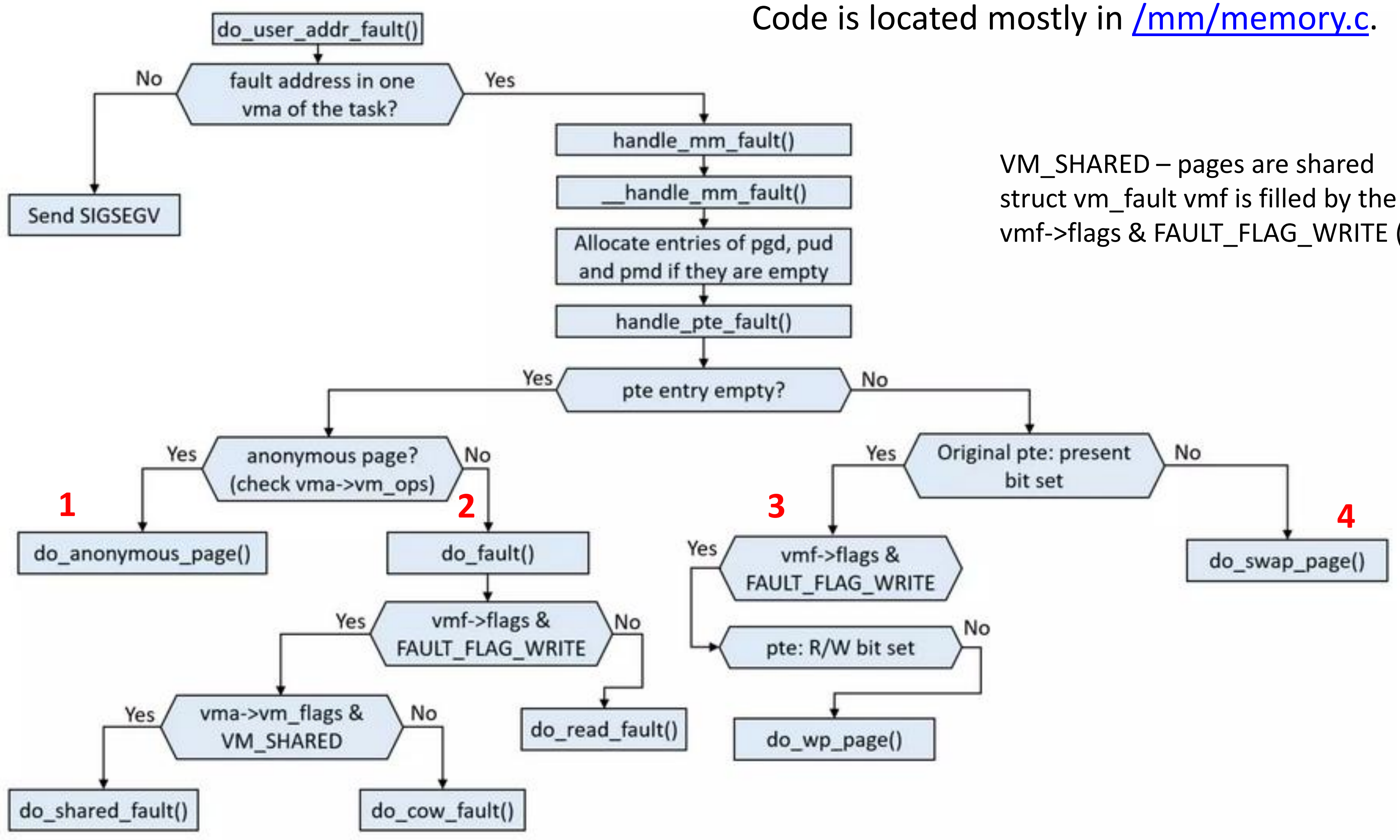
If the **bottom of the stack** has been found (an area has been found whose end address is greater than address A which caused the error, address A does not belong to this area, however the area has the **VM_GROWSDOWN** flag set), then the **stack will be expanded** (call **expand_stack()**).

If the address is **correct**, then the **access rights** to the page are checked, and then it calls the **handle_mm_fault()** function, which will bring the page.

Page fault handler for userspace address

Code is located mostly in /mm/memory.c.

VM_SHARED – pages are shared
struct vm_fault vmf is filled by the pagefault handler
vmf->flags & FAULT_FLAG_WRITE (type of access)



(source: Adrian Huang, Memory Mapping Implementation (mmap), 2022)

Additional reading

- Linux Memory Managment Frequently Asked Questions (Rob Landley).
- Priority trees in the Linux kernel (Michał Jastrzębski) – were replaced (2012) by interval trees. The interval tree is based on the augmented rbtree.
- mm: per-thread VMA caching (Davidlohr Bueso, February 2014)
- Another attempt at speculative page-fault handling (Nur Hussein, April 2017)
- Microsoft Research: A fork() in the road, A. Baumann, J. Appavoo, O. Krieger, T. Roscoe, HotOS '19, May 13–15, 2019, Bertinoro, Italy (paper, presentation).
- On-demand-fork: a microsecond fork for memory-intensive and latency-sensitive applications (paper), On-demand-fork (presentation from EuroSys 2021)
- How to get rid of mmap_sem?, Jonathan Corbet, May 2019.

  Renamed to **mmap_lock** (v5.8), part of an effort to wrap the lock in an API, hoping to ease its replacement in the future.