Unit 3: Process Address Space

This slice set is based on Prof. Shih-Kun Huang's material and the book "Understanding the Linux Kernel"

Memory Addressing

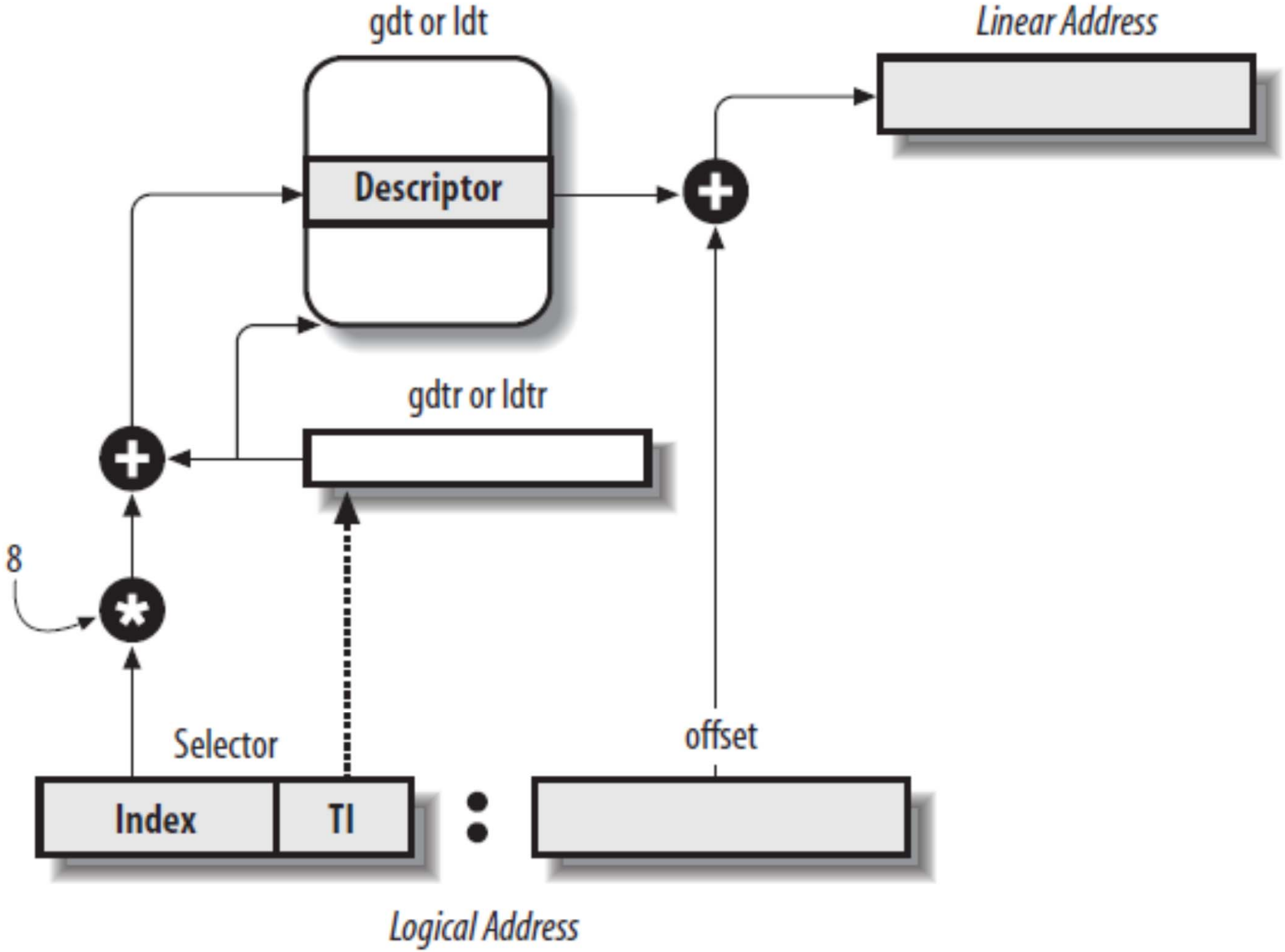
Three kinds of Addresses

- Logical Address
  - Each logical address consists of a segment and an offset
- Linear Address (virtual address)
  - 32-bits unsigned integer to address up to 4GB
- Physical Address
  - Used to address memory cells in memory chips

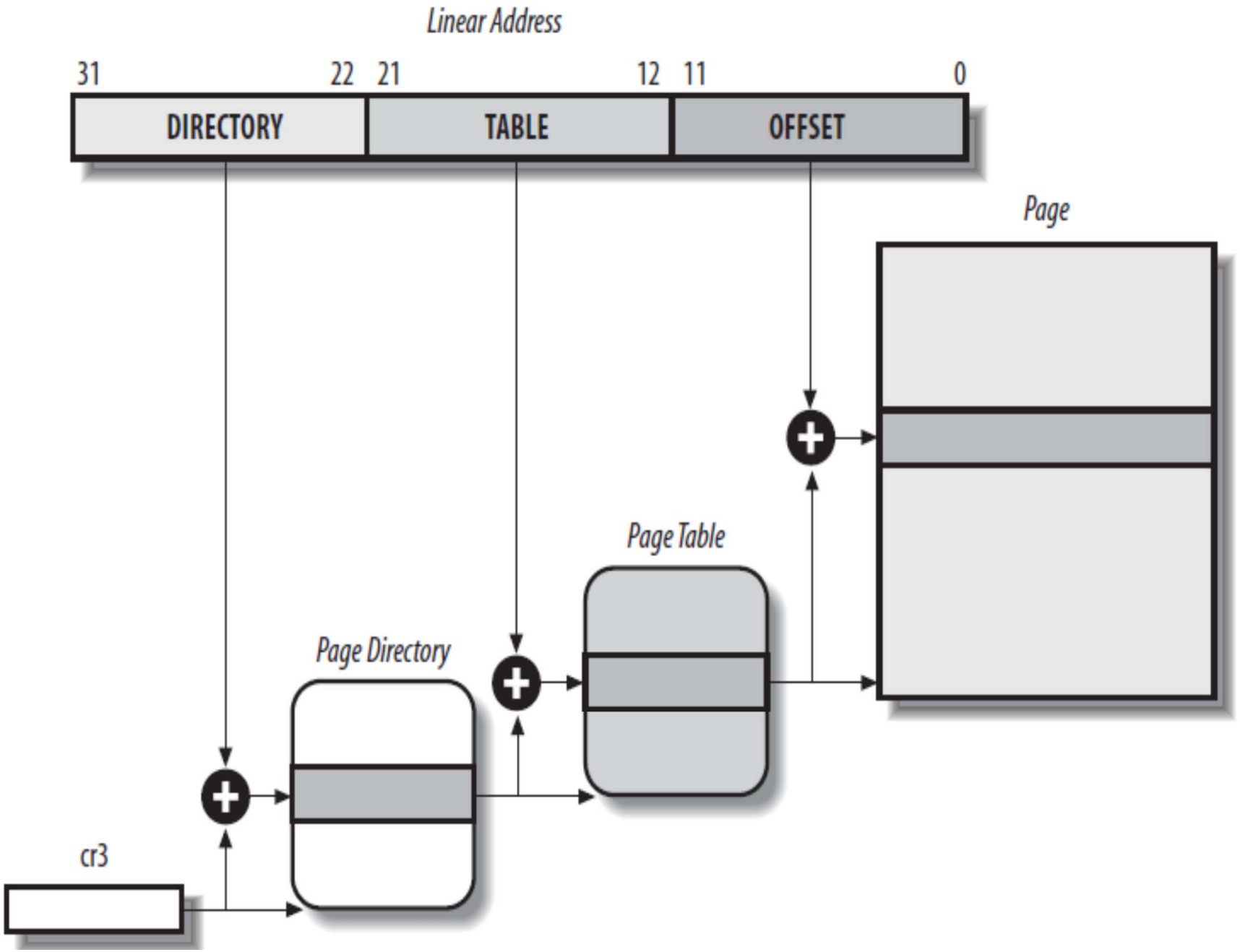
Logical Address translation

- Memory Management Unit (MMU) transforms a logical address into a linear address by Segmentation unit.

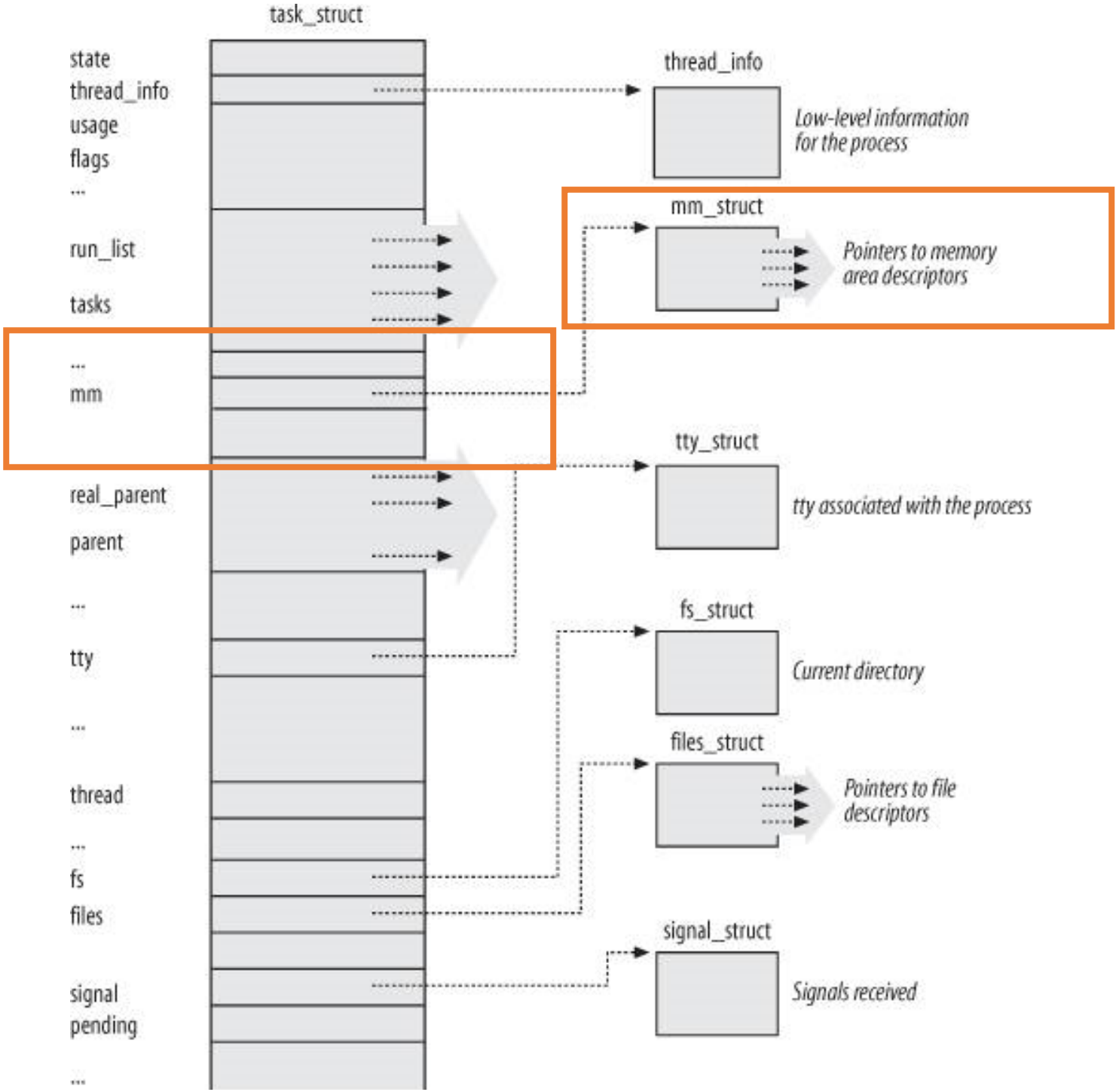
Segmentation


gdt or ldt
Linear Address
Descriptor
gdtr or ldtr
8
Selector
offset
Index
TI
Logical Address

Paging


Linear Address
31
22
21
12
11
0
DIRECTORY
TABLE
OFFSET
Page
Page Table
Page Directory
cr3

Process Address Space


task_struct
state
thread_info
usage
flags
...
run_list
tasks
...
mm
real_parent
parent
...
tty
...
thread
...
fs
files
signal
pending
...
thread_info
Low-level information for the process
mm_struct
Pointers to memory area descriptors
tty_struct
tty associated with the process
fs_struct
Current directory
files_struct
Pointers to file descriptors
signal_struct
Signals received

Linux Process Descriptor task_struct

pid

state

exit_state

stack

- thread_info
- Kernel mode stack
- current

thread

- thread_struct
- Hardware context
- Process switch

rt

- sched_rt_entity
- rq run_list

mm, active_mm

- mm_struct

parent, children, sibling

- list_head

Process Address Space

- When a process asks for dynamic memory (e.g., allocating its heap), it did not get all page frames. Instead, it gets the right to use a new range of linear memory
  - Called memory region
    - Initial linear address
    - Length
    - Access rights
- Page frames in memory regions are allocated and mapped on demand

Process Address Space

- Each process is with a different set of linear addresses
  - Address used by one process bears no relation to the address used by another process
  - Kernel dynamically modify a process address space by adding or removing intervals of linear addresses
- Memory regions vs. memory zones!
  - What is the difference?

Uses of memory regions

- Create a new process (fork)
  - New regions for the new process
- Load an entirely different program (exec)
  - Switching to a new set of memory regions
- Memory mapping on a file (mmap)
- Expanding a process's user-mode stack or heap (brk)
- Creating an IPC-shared memory region (shm)

Memory region related system calls

- brk(): changes the heap size of the process
- execve(): loads a new executable
- _exit(): terminate the current process
- fork(): create a new process
- mmap(), mmap2(): create a memory mapping for a file
- mremap(): expands or shrinks a memory region
- remap_file_pages(): a non-liner mapping for a file
- munmap(): destroys a memory mapping for a file
- shmat(): attaches a shared memory region
- shmdt(): detaches a shared memory region

Two types of invalid linear addresses

- Those caused by programming errors
  - Get an SIGSEGV
- Those caused by a missing page (page frame corresponding to that address has yet to be allocated)
  - Induced page faults exploited by the kernel to implement a demand paging

The Memory Descriptor

- All Information related to the process address space included in an object called the memory descriptor of type mm_struct.
  - The mm field of the process descriptor
  - mm_struct has a link list of vm_area_struct, each of which describes a memory region

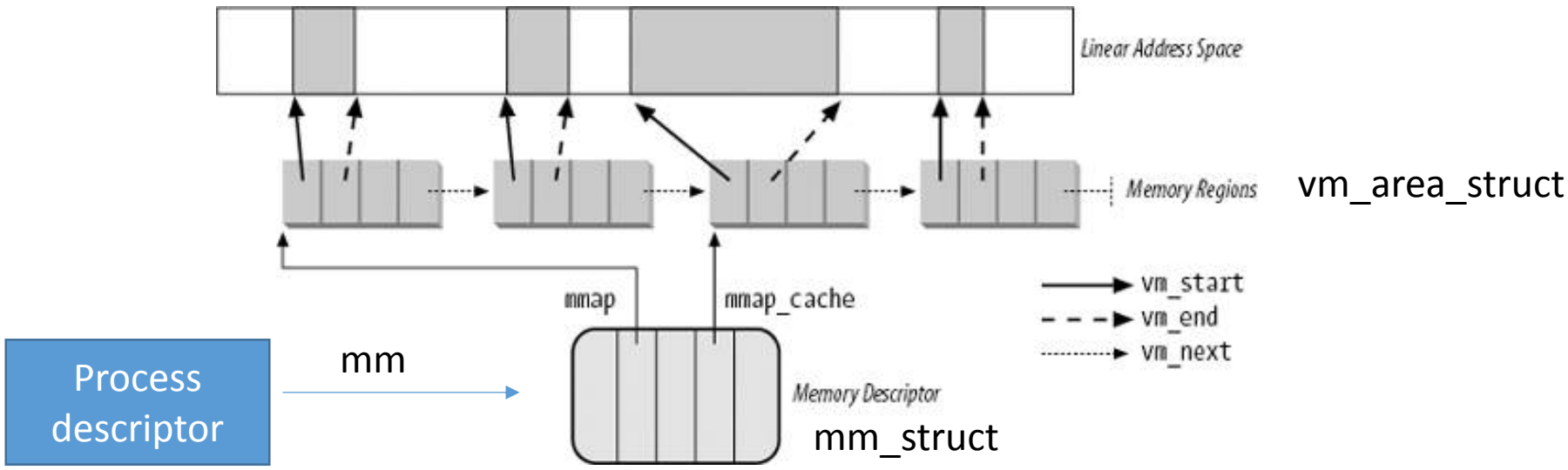


Table 9-2. The fields of the memory descriptor

Type	Field	Description
struct vm_area_struct *	mmap	Pointer to the head of the list of memory region objects
struct rb_root	mm_rb	Pointer to the root of the red-black tree of memory region objects
struct vm_area_struct *	mmap_cache	Pointer to the last referenced memory region object
unsigned long (*)()	get_unmapped_area	Method that searches an available linear address interval in the process address space
void (*)()	unmap_area	Method invoked when releasing a linear address interval
unsigned long	mmap_base	Identifies the linear address of the first allocated anonymous memory region or file memory mapping (see the section "Program Segments and Process Memory Regions" in Chapter 20)
unsigned long	free_area_cache	Address from which the kernel will look for a free interval of linear addresses in the process address space
pgd_t *	pgd	Pointer to the Page Global Directory
atomic_t	mm_users	Secondary usage counter
atomic_t	mm_count	Main usage counter
int	map_count	Number of memory regions
struct rw_semaphore	mmap_sem	Memory regions' read/write semaphore
spinlock_t	page_table_lock	Memory regions' and Page Tables' spin lock
struct list_head	mmlist	Pointers to adjacent elements in the list of memory descriptors

Linux Memory Descriptor mm_struct

mmap

- vm_area_struct
- Head of the list of virtual memory region

mm_rb

- struct rb_node mm_rb
- Root of the red-black tree of memory region

map_cache

- Last referenced memory region

get_unmapped_area

- Searches an available linear address interval

unmap_area

- Releasing a linear address interval

map_base

- First allocated anonymous memory region

pgd

- pgd_t: page table directory

mmlist

- list_head: adjacent elements in the list of memory descriptors

Memory descriptor (mm_struct)

- All memory descriptors are in a global doubly linked list
  - The first element of the list is the memory descriptor used by process 0
  - List is protected by mmlist_lock spin lock
  - *mmlist* refers to the next list element
- *map_count*: number of regions owned by the process
  - default max: 65535
  - /proc/sys/vm/max_map_count
- *mm_count*
  - main usage counter of a memory descriptor
- *mm_users*
  - number of lightweight processes sharing the mm_struct (clone())
  - For example, two threads sharing a mm_struct
    - mm_users =2, mm_count =1

The first task_struct and its mm_struct

Linux/init/init_task.c

struct task_struct init_task = INIT_TASK(init_task);

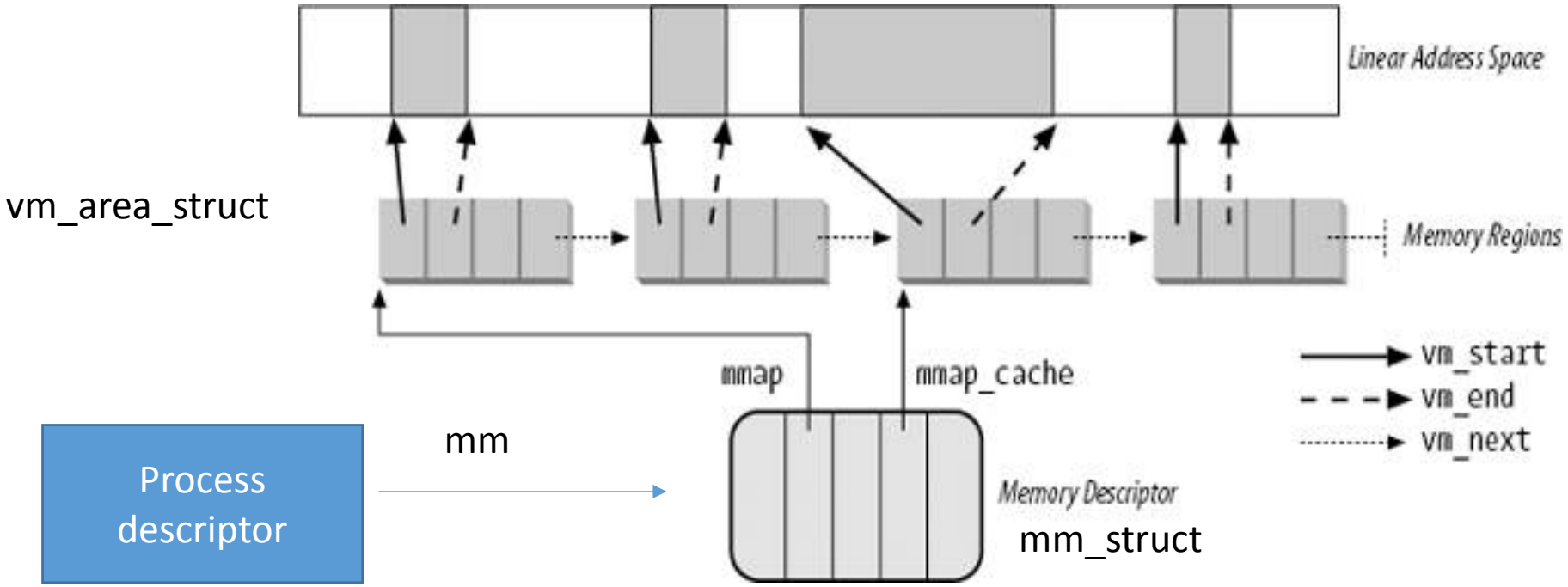
```
11 struct mm_struct init_mm = {
12       .mm_rb        = RB_ROOT,
13       .pgd          = swapper_pg_dir,
14       .mm_users     = ATOMIC_INIT(2),
15       .mm_count     = ATOMIC_INIT(1),
16       .mmap_sem     = __RWSEM_INITIALIZER(init_mm.mmap_sem),
17       .page_table_lock =  __SPIN_LOCK_UNLOCKED(init_mm.page_table_lock),
18       .mmlist       = LIST_HEAD_INIT(init_mm.mmlist),
19       .cpu_vm_mask    = CPU_MASK_ALL,
20 };
```

Memory Descriptors of Kernel Threads

- A kernel thread does not have its own address space and its PD's *mm* field is always NULL
  - How does it accesses linear addresses >= 3GB?
    - It never accesses linear addresses < 3GB
  - Why not borrow the memory descriptor from a regular process, because the page table entries >= 3GB of all processes are identical?

- The *mm* field points to the memory descriptor owned by the process, and the active_mm field of a process descriptor points to the memory descriptor that is used by the process
  - For regular processes, the two fields are the same
  - For kernel threads, mm=NULL and acrive_mm points to the mm field of the last running regular process

Memory Region Descriptors (again)



vm_area_struct

Table 9-3. The fields of the memory region object

Type	Field	Description
struct mm_struct *	vm_mm	Pointer to the memory descriptor that owns the region.
unsigned long	vm_start	First linear address inside the region.
unsigned long	vm_end	First linear address after the region.
struct vm_area_struct *	vm_next	Next region in the process list.
pgprot_t	vm_page_prot	Access permissions for the page frames of the region.
unsigned long	vm_flags	Flags of the region.
struct rb_node	vm_rb	Data for the red-black tree (see later in this chapter).
union	shared	Links to the data structures used for reverse mapping (see the section "Reverse Mapping for Mapped Pages" in Chapter 17).
struct list_head	anon_vma_node	Pointers for the list of anonymous memory regions (see the section "Reverse Mapping for Anonymous Pages" in Chapter 17).
struct anon_vma *	anon_vma	Pointer to the anon_vma data structure (see the section "Reverse Mapping for Anonymous Pages" in Chapter 17).
struct vm_operations_struct*	vm_ops	Pointer to the methods of the memory region.
unsigned long	vm_pgoff	Offset in mapped file (see Chapter 16). For anonymous pages, it is either zero or equal to vm_start/PAGE_SIZE (see Chapter 17).
struct file *	vm_file	Pointer to the file object of the mapped file, if any.
void *	vm_private_data	Pointer to private data of the memory region.
unsigned long	vm_truncate_count	Used when releasing a linear address interval in a non-linear file memory mapping.

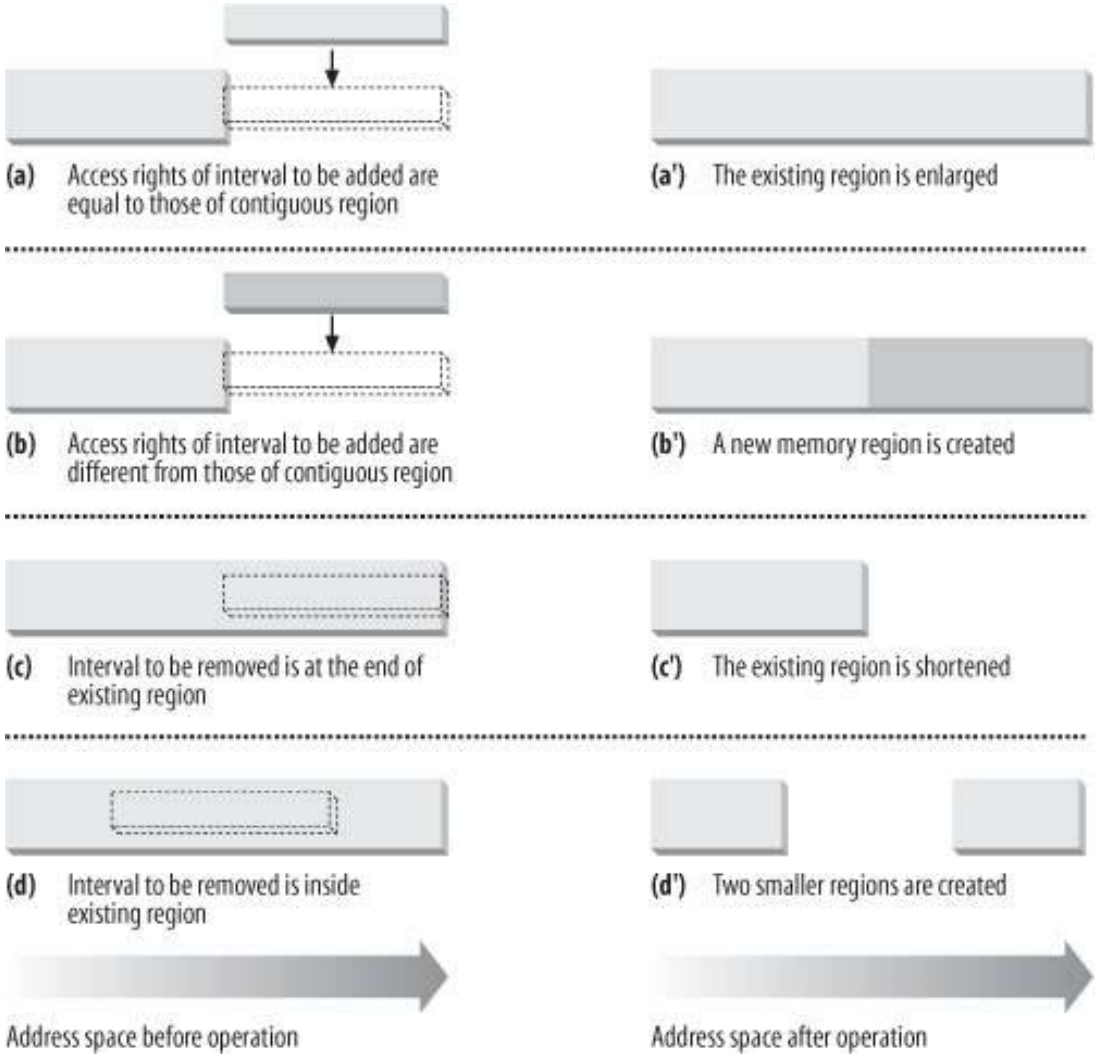
Memory region descriptor

- Each descriptor identified a linear address interval
  - *vm_start*: the first linear address of the interval
  - *vm_end*: first linear address outside of the interval
  - *vm_end-vm-start*: length of the memory region
- *vm_mm*: points to the *mm_struct* memory descriptor of the process that owns the region

Properties of memory regions

- Memory regions never overlap
- Kernel tries to merge regions when a new one is allocated right next to an existing one
  - Two adjacent regions can be merged if their *access rights* match

Adding or removing a linear address interval



Methods to Act on a Memory Region

- open: when the memory region added to a process
- Close: when the memory region removed from a process
- nopage: invoked by the page fault exception handler, when he page not in RAM, with linear address in the memory region
- populate: set the page table entries corresponding to the linear address of the memory region (pre-faulting)

Searching the memory region list

- A frequent operation is to find the correspond memory region of a linear address
  - E.g., finding out the region containing a faulting page

- Linear search? Definitely no.

- Linux organizes memory regions in a red-black
  - Each node has two children: left child and right child
  - Elements in the tree are sorted
    - For each node N, all elements of the subtree rooted at the left child of N precede N
    - All emements of the subttree rooted at the right child of N follow N

Red-black Trees (rbtree) in Linux

- A self-balancing binary search tree
  - Storing sortable key/value data pairs
- Faster real-time bounded worst case for insertion and deletion (at most two rotations for insertion and three rotations for deletion)
- Slower lookup time (still in O(log n)), compared with AVL trees
  - AVL trees vs. RB trees?

Use of red-black tree in the kernel

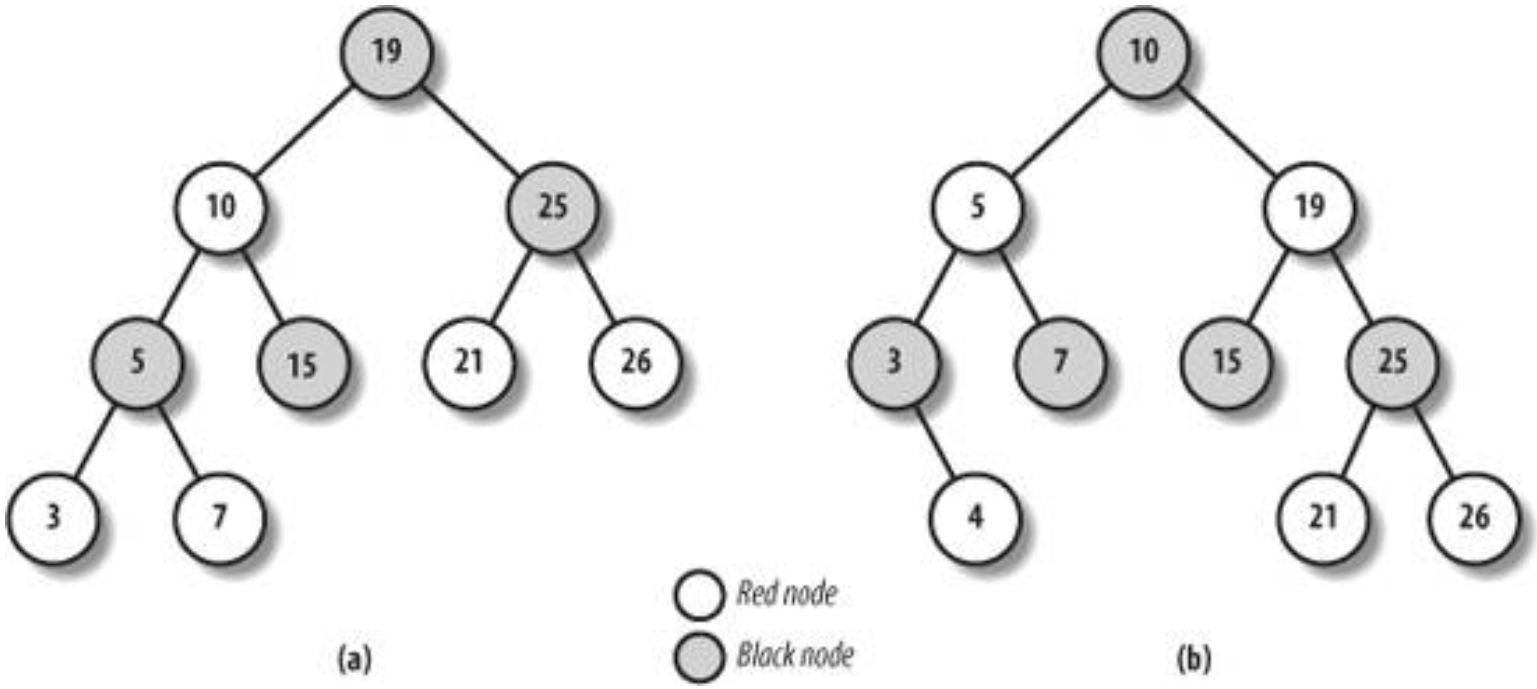
- I/O schedulers (deadline and CFQ to track requests)
- CD/DVD driver
- High-resolution timer to organize outstanding timer requests
- Ext3 filesystem tracks directory entries
- VMAs (virtual memory areas) tracking
- epoll file descriptors, cryptographic keys, network packets in “hierarchical token bucket” scheduler

Rules of red-black tree

- Every node must be either red of black
- The root of the tree must be black
- The children of a red node must be black
- Every path from a node to a descendant leaf must contain the same number of black nodes
  - Null pointers are counted as black nodes

- Every red-black tree with n internal nodes has a height of at most 2 x log(n+1)
- Searching is efficient: log(n)
- Each new node must be inserted as a leaf and colored red

Examples of red-black tress



Linux implementation

- “/lib/rbtree.c”
  - #include <linux/rbtree.h>
- Optimized for speed
  - One less layer of indirection (with better cache locality) than traditional tree implementation
- rb_node is embedded in the data structure
  - Traditional implementation: separate rb_node and data structures

rb_node and rb_root

```
35 struct rb_node {
36       unsigned long  __rb_parent_color;
37       struct rb_node *rb_right;
38       struct rb_node *rb_left;
39 } __attribute__((aligned(sizeof(long))));
40    /* The alignment might seem pointless, but allegedly CRIS needs it */
41
42 struct rb_root {
43       struct rb_node *rb_node;
44 };
45
```

container_of

```
struct mytype {
    struct rb_node node;
    char *keystring;
};
```

Embed rb_node in my data structure…

```
825 /**
826  * container_of -  cast a member of a structure out to the containing structure
827  * @ptr:        the pointer to the member.
828  * @type:       the type of the container struct this is embedded in.
829  * @member:     the name of the member within the struct.
830  *
831  */
832 #define container_of(ptr, type, member) ({                  \
833       const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
834       (type *)( (char *)__mptr - offsetof(type,member) );})
835
```

Search in rbtree

```
struct mytype *my_search(struct rb_root *root, char *string)
{
    struct rb_node *node = root->rb_node;

    while (node) {
        struct mytype *data = container_of(node, struct mytype, node);
        int result;

        result = strcmp(string, data->keystring);

        if (result < 0)
            node = node->rb_left;
        else if (result > 0)
            node = node->rb_right;
        else
            return data;
    }
    return NULL;
}
```

Insert in rbtree

```
int my_insert(struct rb_root *root, struct mytype *data)
{
    struct rb_node **new = &(root->rb_node), *parent = NULL;

    /* Figure out where to put new node */
    while (*new) {
        struct mytype *this = container_of(*new, struct mytype, node);
        int result = strcmp(data->keystring, this->keystring);

        parent = *new;
        if (result < 0)
            new = &((*new)->rb_left);
        else if (result > 0)
            new = &((*new)->rb_right);
        else
            return FALSE;
    }

    /* Add new node and rebalance tree. */
    rb_link_node(&data->node, parent, new);
    rb_insert_color(&data->node, root);

    return TRUE;
} ? end my_insert ?
```

vm_area_struct

```
134 struct vm_area_struct {
135       struct mm_struct * vm_mm;      /* The address space we belong to. */
136       unsigned long vm_start;       /* Our start address within vm_mm. */
137       unsigned long vm_end;         /* The first byte after our end address
138                                        within vm_mm. */
139
140       /* linked list of VM areas per task, sorted by address */
141       struct vm_area_struct *vm_next, *vm_prev;
142
143       pgprot_t vm_page_prot;        /* Access permissions of this VMA. */
144       unsigned long vm_flags;       /* Flags, see mm.h. */
145
146       struct rb_node vm_rb;
147 }
```

Data structures commonly used

- Linked list (wait queue, run queue, process organization, memory descriptor)
  - Doubly-linked list
- Balanced binary search tree (CFQ scheduling and memory regions)
  - Sortable key/value data pairs
  - Red-black tree and AVL tree
- Radix tree (page reclaiming)
  - Store sparse arrays
  - Long indexes
- Hash table (pid hash)
  - Specific size and hash function
  - Not sorted to be traversed

Memory region access rights

- Three kinds of flags associated with a page
- Read/Write,Present,Uesr/Supervisor in each page table
  - Used by hardware for memory protection
- Flags field of each page descriptor
  - Used by Linux for page frame management, including page frame reclaiming
  - PG_lru, PG_referenced, PG_dirty, PG_active ….
- With the pages of a memory region
  - Used by Linux for region access rights (in vm_flags)
  - Pages in the same region share the same access right

Page Table Flags vs. Protection Flags

- There are many flags in *vm_flags*, but there are only limited bits in x86 page table entries
  - R/W, User/Kernel, Present
  - The MMU do not understand all the page flags defined by the kernel
  - Translating the protection flags to page table bits is not straightforward
- The kernel tricks the MMU when setting the bits of page table entries, so that page fault interrupts can be generated as desired
  - For example, a copy-on-write page that is allowed to be written. But its page table write bit is cleared and write to it generates a page fault interrupt
  - By checking the memory region flag (with VM_SHARED set), the kernel can distinguish COW from writing read-only pages

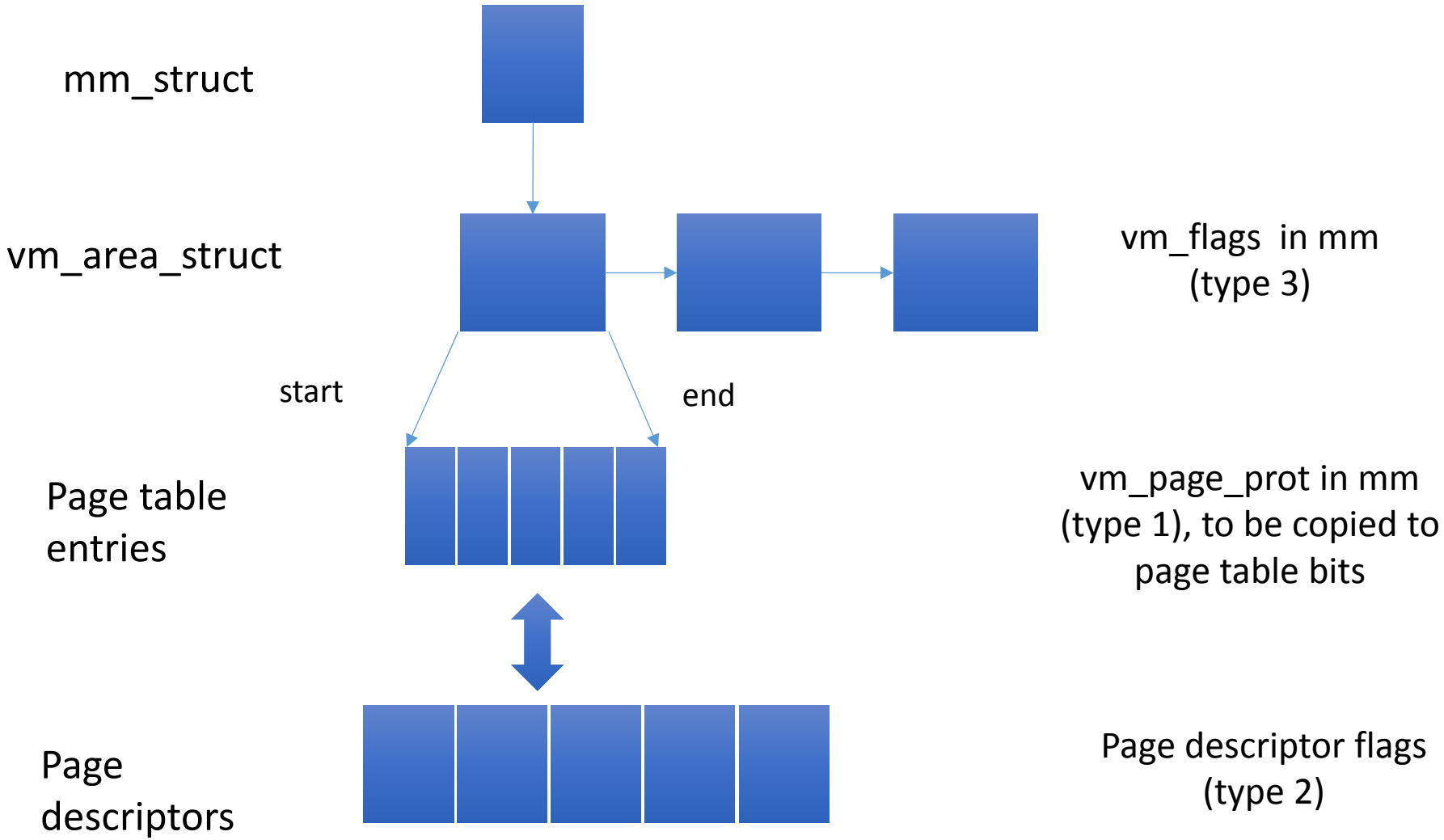

mm_struct
vm_area_struct
vm_flags in mm
(type 3)
start
end
Page table
entries
vm_page_prot in mm
(type 1), to be copied to
page table bits
Page
descriptors
Page descriptor flags
(type 2)

Table 9-5. The memory region flags

Flag name	Description
VM_READ	Pages can be read
VM_WRITE	Pages can be written
VM_EXEC	Pages can be executed
VM_SHARED	Pages can be shared by several processes
VM_MAYREAD	VM_READ flag may be set
VM_MAYWRITE	VM_WRITE flag may be set
VM_MAYEXEC	VM_EXEC flag may be set
VM_MAYSHARE	VM_SHARE flag may be set
VM_GROWSDOWN	The region can expand toward lower addresses
VM_GROWSUP	The region can expand toward higher addresses
VM_SHM	The region is used for IPC's shared memory
VM_DENYWRITE	The region maps a file that cannot be opened for writing
VM_EXECUTABLE	The region maps an executable file
VM_LOCKED	Pages in the region are locked and cannot be swapped out
VM_IO	The region maps the I/O address space of a device
VM_SEQ_READ	The application accesses the pages sequentially
VM_RAND_READ	The application accesses the pages in a truly random order
VM_DONTCOPY	Do not copy the region when forking a new process
VM_DONTEXPAND	Forbid region expansion through mremap( ) system call
VM_RESERVED	The region is special (for instance, it maps the I/O address space of a device), so its pages must not be swapped out
VM_ACCOUNT	Check whether there is enough free memory for the mapping when creating an IPC shared memory region (see Chapter 19)
VM_HUGETLB	The pages in the region are handled through the extended paging mechanism (see the section "Extended Paging" in Chapter 2)
VM_NONLINEAR	The region implements a non-linear file mapping

vm_area_struct->vm_flags

16 possible combinations

- Read,write, execute,share (16)
  - Both write and share, Read/Write bit set
  - Read or Execute, but no write or share, read/Write clear
  - NX supported, and no execute, NX bit set
  - Without any right, present bit cleared
- *vm_page_prot[]* is an array has 16 elements, each of which contains a bit pattern to be copied to page table entries
- When a new page is added to a memory region, its page table entry bits are set to *vm_page_prot*[vm_flags & 0x0f]

Memory region handling

- Helper functions that simplify the implementation of do_mmap() and do_munmap()
  - *find_vma*()
    - finding the closest region to a given address
  - *find_vma_intersection*()
    - finding a region that overlaps a given interval:
  - *get_unmpapped_area*()
    - finding a free interval:
  - *insert_vm_struct*()
    - inserting a region in the memory descriptor list:

find_vma()

```
vma = mm->mmap_cache;
if (vma && vma->vm_end > addr && vma->vm_start <= addr)
    return vma;

rb_node = mm->mm_rb.rb_node;
vma = NULL;
while (rb_node) {
    vma_tmp = rb_entry(rb_node, struct vm_area_struct, vm_rb);
    if (vma_tmp->vm_end > addr) {
        vma = vma_tmp;
        if (vma_tmp->vm_start <= addr)
            break;
        rb_node = rb_node->rb_left;
    } else
        rb_node = rb_node->rb_right;
}
if (vma)
    mm->mmap_cache = vma;
return vma;
```

find_vma_intersection()

```
vma = find_vma(mm, start_addr);
If (vma && end_addr <= vma->vm_start)
      vma = NULL;
return vma;
```

get_unmapped_area()

```
if (len > TASK_SIZE)
  return - ENDMEM;
addr = (addr + 0xff) & 0xfffff000;
if (addr && addr + len <= TASK_SIZE) {
    vma = find_vma(current->mm, addr);
    if (!vma || adddr + len <= vma->vm_start)
      return addr;
}
start_addr = addr = mm->free_area_cache;
for (vma = find_vma(current->mm,addr);;vma=vma->vm_next) {
      if (addr + len > TASK_SIZE) {
            ... ...
      }
      if (!vma || addr + len <= vma->vm_start) {
                  mm->free_area_cache = addr + len;
                  return addr;
      }
      addr = vma->vm_end;
}
```

More vma related functions

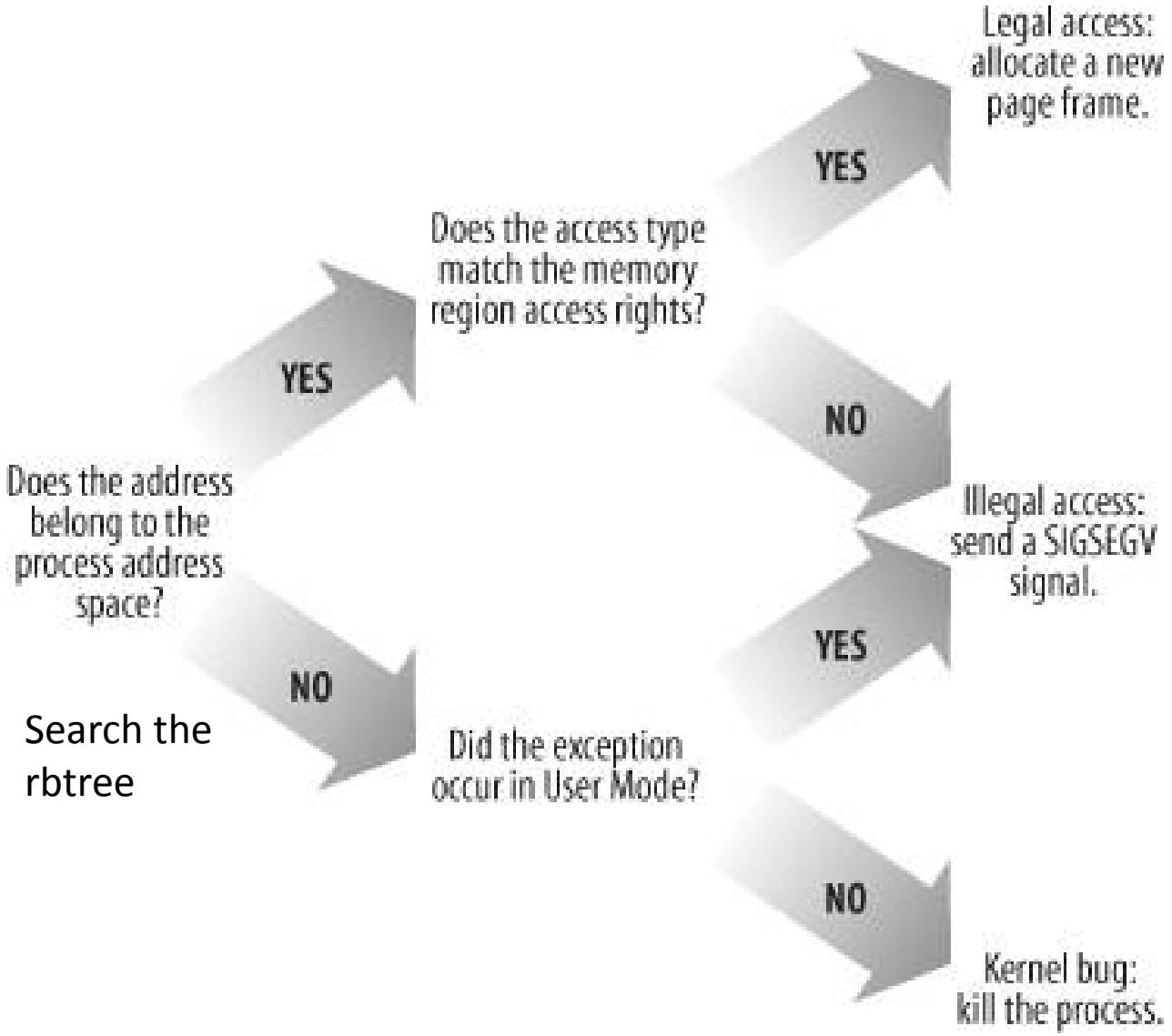
- insert_vm_struct()
  - inserting a region in the memory descriptor list
- do_mmap()
  - Allocating a linear address interval
- do_munmap()
  - Releasing a linear address interval
- split_vma()
  - Split a memory region intersecting a linear address interval into two smaller regions
- unmap_region()
  - Release the page frames in a list of memory regions

Page Fault Exception Handling

Page Fault Exception Handling

- the Linux Page Fault exception handler must distinguish exceptions caused by
  - programming errors
  - a reference to a page that legitimately belongs to the process address space but simply hasn't been allocated yet
- do_page_fault(), the Page Fault interrupt service routine, compares the faulty liner address against the memory regions of the current process

Overall scheme for the Page Fault handler



Search the rbtree

Parameters of do_page_fault()

- The regs address of a pt_regs structure containing the values of the microprocessor registers when the exception occurred
- A 3-bit error_code on the stack:
  - Bit 0=0 → the exception was caused by an access to a page that is not present (the Present flag in the Page Table entry is clear); otherwise, the exception was caused by an invalid access right.
  - Bit 1=0 → the exception was caused by a read or execute access; if set, the exception was caused by a write access.
  - Bit 2=0 → the exception occurred while the processor was in Kernel Mode; otherwise, it occurred in User Mode.

do_page_fault()

- CR2 register
  - Contains a value called Page Fault Linear Address (PFLA), which is the address of the instruction that caused the fault
  - Do_page_fault() first read cr2 and store the address in *address*

```
asm("movl %%cr2,%0":"=r" (address));
if (regs->eflags & 0x00020200) local_irq_enable( );
tsk = current;
```

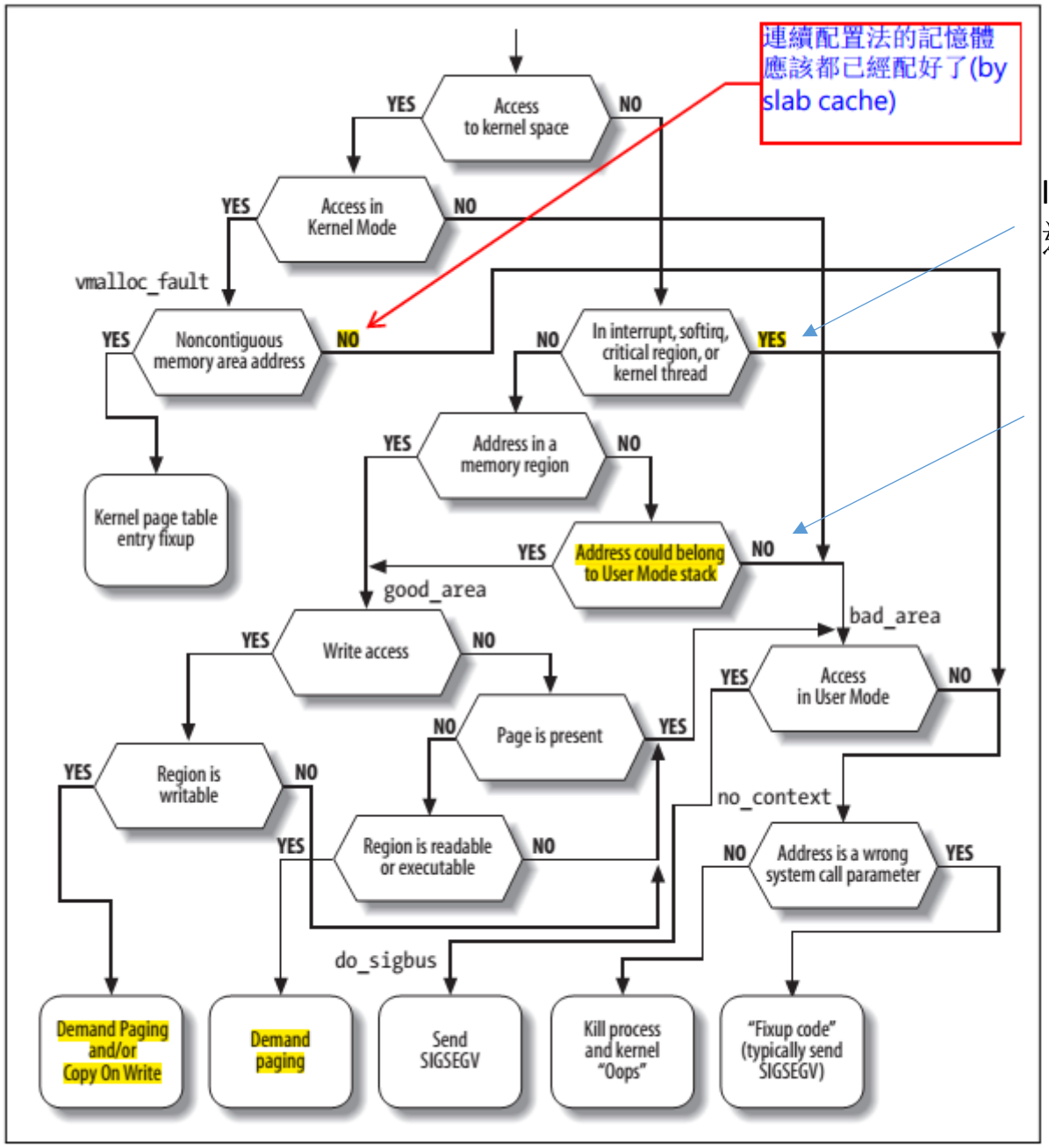


Figure 9-5. The flow diagram of the Page Fault handler

ISR 裡面 page fault 通常死定了!

Push [mem] 指令引起 page fault ，可是不在 的page 位於 mem 該位址而不是 stack 上

Faulty linear address check

- Are we in the kernel?
  - Yes. do_ page_fault( ) checks whether the faulty linear address belongs to the fourth gigabyte
    - If not, then the fault is likely caused by accessing to the vmalloc()'ed pages

```
info.si_code = SEGV_MAPERR;
if (address >= TASK_SIZE ) { // fourth gigabyte
 //nonexisting page frame
  if (!(error_code & 0x101)) goto vmalloc_fault;
  goto bad_area_nosemaphore;
}
```

Handling a Faulty Address Outside the Address Space

- Send a SIGSEGV signal to the faulting process
  - force_sig_info() makes sure that the process does not ignore/block the signal
  - Interestingly, a process can get rid of SIGSEGV by registering an empty handler of SIGSEGV

```
bad_area:
up_read(&tsk->mm->mmap_sem);
bad_area_nosemaphore:
if (error_code & 4) { /* User Mode */
    tsk->thread.cr2 = address;
    tsk->thread.error_code = error_code | (address >= TASK_SIZE);
    tsk->thread.trap_no = 14;
    info.si_signo = SIGSEGV;
    info.si_errno = 0;
    info.si_addr = (void *) address;
    force_sig_info(SIGSEGV, &info, tsk); return;
}
```

Handling a Faulty Address Inside the Address Space

Checks whether the read/write access matches the vm protection flags

```
good_area:
info.si_code = SEGV_ACCERR;
write = 0;
if (error_code & 2) { /* write access */
 if (!(vma->vm_flags & VM_WRITE)) goto bad_area; write++;
} else /* read access */
 if ((error_code & 1) || !(vma->vm_flags & (VM_READ | VM_EXEC)))
  goto bad_area;
```

If the access is granted, fault the page in. The page fault can either be major or minor

```
survivie:
ret = handle_mm_fault(tsk->mm, vma, address, write);
if (ret == VM_FAULT_MINOR|| ret == VM_FAULT_MAJOR) {
    if (ret == VM_FAULT_MINOR)
                tsk->min_flt++;
     else tsk_>maj_flt++;
     up_read(&tsk->mm->mmap_sem); // read-only access to the protected resources
}
```

Demand Paging

- How a page fault is handled depends on the prior use of the page
  - The page never accessed and not map a disk file
  - The page belong to a non-linear disk file
  - The page already accessed by the process , but temporarily saved to disk

```
entry = *pte;
if (! pte_present(entry)) {
    if (pte_none(entry)) //case 1
        return do_no_page(mm, vma, address, write_access, pte, pmd);
    if (pte_file(entry)) // case 2
        return do_file_page(mm, vma, address, write_acess, pte, pmd);
    return do_swap_page(mm, vma, address, pte, pmd, entry, write_access); // case 3
}
```

Demand Paging

- On the first case, the page has not been mapped
- vm->vm_ops->nopage
  - NULL: the page is an anonymous page
    - Will be handled by the virtual memory manager
  - Non-NULL: the page is an file-mapped page
    - Will be handled by file systems
- The page cache stores all the faulted-in pages

```
if (write_access) {
    pte_unmap(page_table);
    spin_unlock(&mm->page_table_lock);
    page = alloc_page(GFP_HIGHUSER | __GFP_ZERO);   ←(1)
    spin_lock(&mm->page_table_lock);
    page_table = pte_offset_map(pmd, addr);         ←(2)
    mm->rss++;
    entry = maybe_mkwrite(pte_mkdirty(mk_pte(page,
                                             vma->vm_page_prot)), vma);
    lru_cache_add_active(page);   ←(3)
    SetPageReferenced(page);      ←(3)
    set_pte(page_table, entry);   ←(4)
    pte_unmap(page_table);
    spin_unlock(&mm->page_table_lock);
    return VM_FAULT_MINOR;
}
```

(1) We are handling a page fault in the user space
(2) Get the page table containing the faulting page
(3) The page is added to the page cache as active and its reference flag is set
(4) Add the page to the page table

More to be addressed

- Original address space duplication way (fork())
  - Allocate page frames for the page tables of the child process
  - Allocate page frames for the pages of the child process
  - Initializing the page tables of the child process
  - Copying the pages of the parent process into the corresponding pages of the child process
- Copy on Write (COW)
  - Pages are shared between the parent and the child process
  - Whenever the parent or the child process attempts to write into a shared page frame, an exception occurs
  - Kernel duplicates the page into a new page frame and mark as writable;
    - the original page frame remains write-protected
- Managing the heap
  - Grow the heap by calling brk()

End of Unit 3