CS6465: Advanced Operating System Implementation

Lecture 2: Process Memory

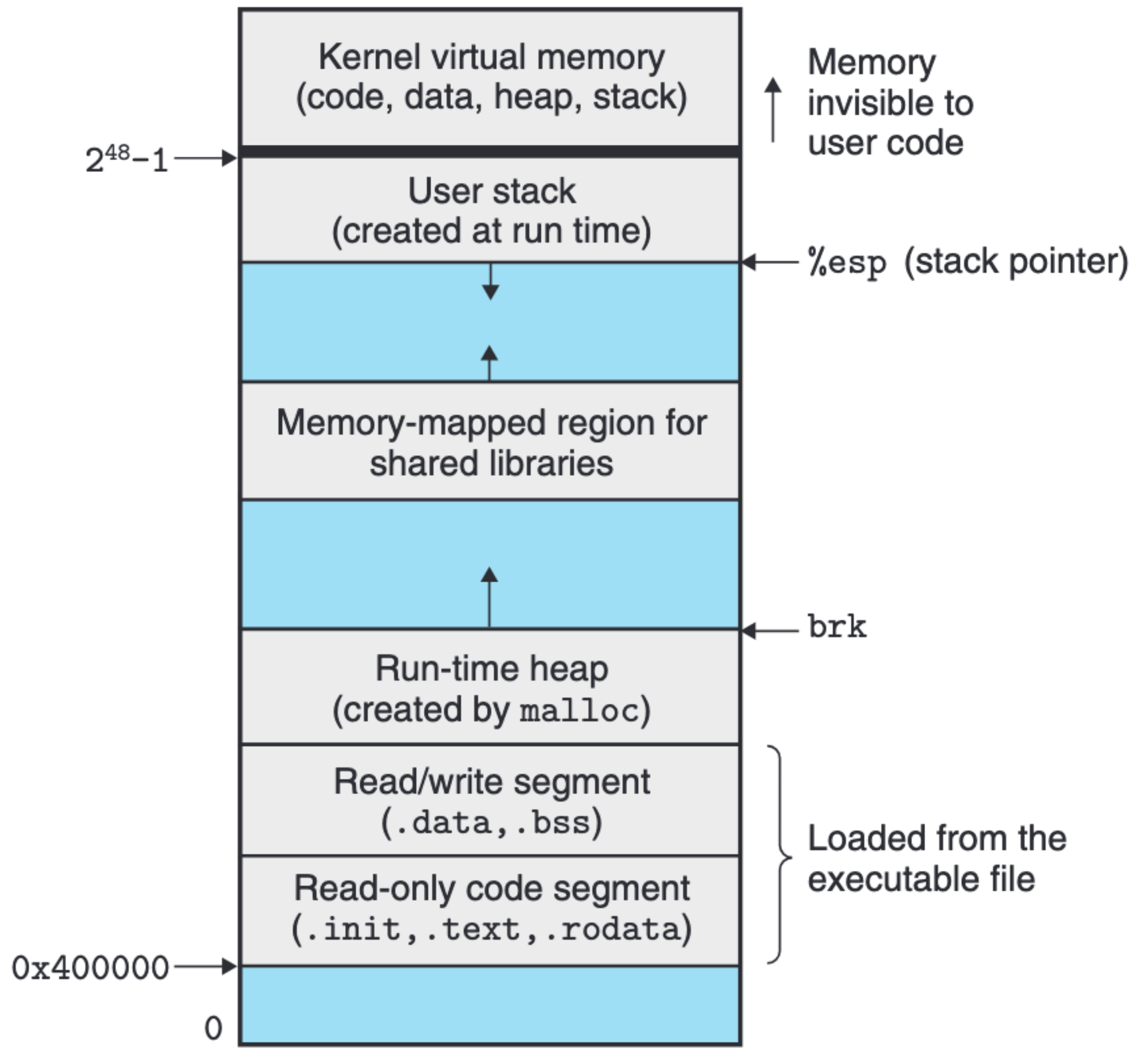
Anton Burtsev

August 2024

Virtual process memory

- Each process has a private address space
- But only a small portion of this address space (3GBs on 32bit machines, and 128TB or 64PT on 64bit machines) is used by an application

Linux Process Memory Layout



https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt

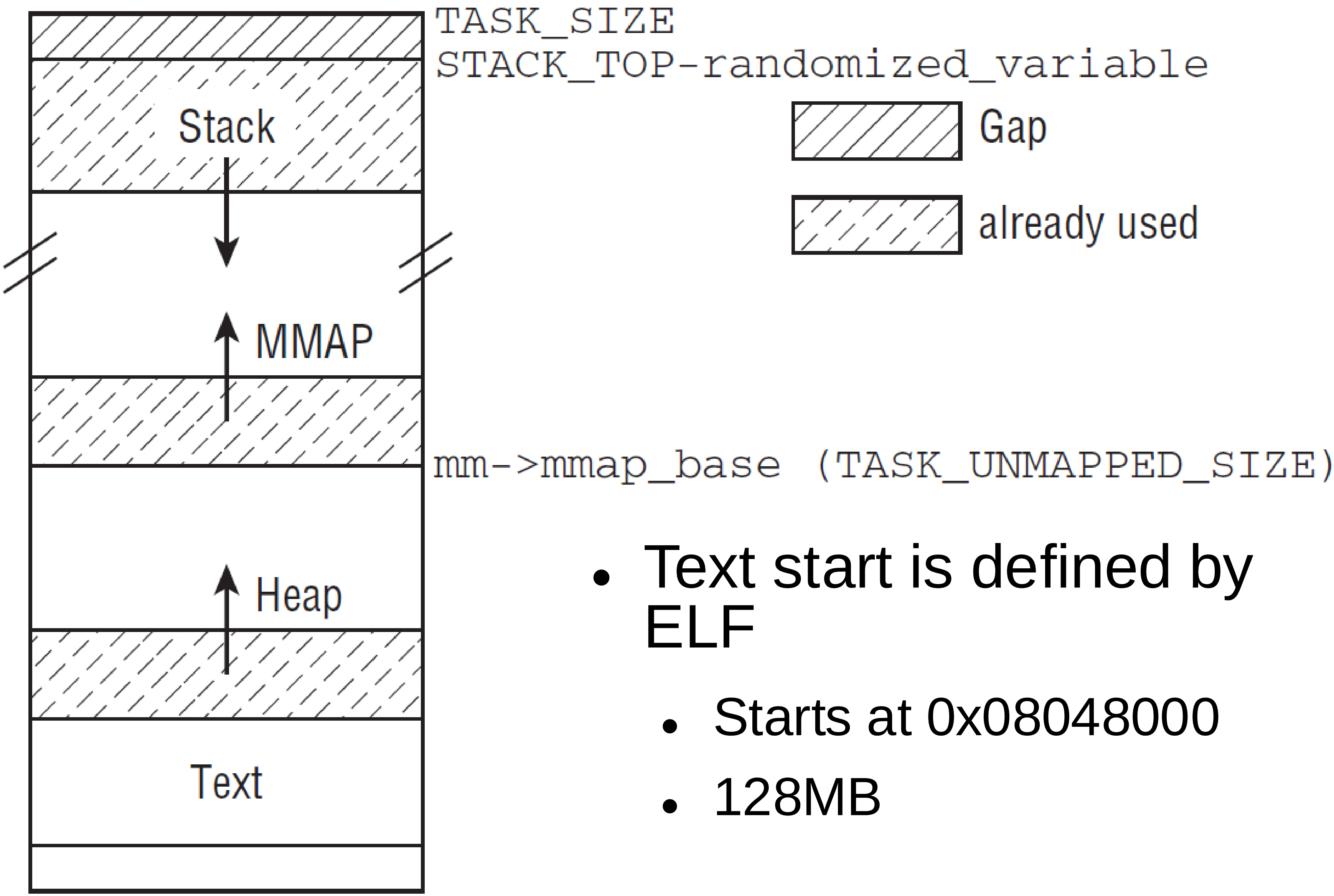
Start addr	Offset	End addr	Size	VM area description
0000000000000000	0	00ffffffffffffff	64 PB	user-space virtual memory, different per mm
0000800000000000	+64 PB	ffff7fffffffffff	~16K PB	... huge, still almost 64 bits wide hole of non-canonical virtual memory addresses up to the -64 PB starting offset of kernel mappings.
				Kernel-space virtual memory, shared between all processes:
ff00000000000000	-64 PB	ff0fffffffffffff	4 PB	... guard hole, also reserved for hypervisor
ff10000000000000	-60 PB	ff10ffffffffffff	0.25 PB	LDT remap for PTI
ff11000000000000	-59.75 PB	ff90ffffffffffff	32 PB	direct mapping of all physical memory (page_offset_base)
ff91000000000000	-27.75 PB	ff9fffffffffffff	3.75 PB	... unused hole
ffa0000000000000	-24 PB	ffd1ffffffffffff	12.5 PB	vmalloc/ioremap space (vmalloc_base)
ffd2000000000000	-11.5 PB	ffd3ffffffffffff	0.5 PB	... unused hole
ffd4000000000000	-11 PB	ffd5ffffffffffff	0.5 PB	virtual memory map (vmemmap_base)
ffd6000000000000	-10.5 PB	ffdeffffffffffff	2.25 PB	... unused hole
ffdf000000000000	-8.25 PB	fffffdffffffffff	~8 PB	KASAN shadow memory
				Identical layout to the 47-bit one from here on:
fffffc0000000000	-4 TB	fffffdffffffffff	2 TB	... unused hole
				vaddr_end for KASLR
fffffe0000000000	-2 TB	fffffe7fffffffff	0.5 TB	cpu_entry_area mapping
fffffe8000000000	-1.5 TB	fffffeffffffffff	0.5 TB	... unused hole
ffffff0000000000	-1 TB	ffffff7fffffffff	0.5 TB	%esp fixup stacks
ffffff8000000000	-512 GB	ffffffeeffffffff	444 GB	... unused hole
ffffffef00000000	-68 GB	fffffffeffffffff	64 GB	EFI region mapping space
ffffffff00000000	-4 GB	ffffffff7fffffff	2 GB	... unused hole
ffffffff80000000	-2 GB	ffffffff9fffffff	512 MB	kernel text mapping, mapped to physical address 0
ffffffff80000000	-2048 MB			
ffffffffa0000000	-1536 MB	fffffffffeffffff	1520 MB	module mapping space
ffffffffff000000	-16 MB			
FIXADDR_START	~-11 MB	ffffffffff5fffff	~0.5 MB	kernel-internal fixmap range, variable size and offset
ffffffffff600000	-10 MB	ffffffffff600fff	4 kB	legacy vsyscall ABI
ffffffffffe00000	-2 MB	ffffffffffffffff	2 MB	... unused hole

Process memory

- Memory of different kinds
  - Code, data, heap, stack
  - Shared libraries
  - Memory mapped files
  - Shared memory regions
  - Copy-on-write regions after the fork
  - Paged out infrequently used pages
- The kernel needs data structures to manage these holes

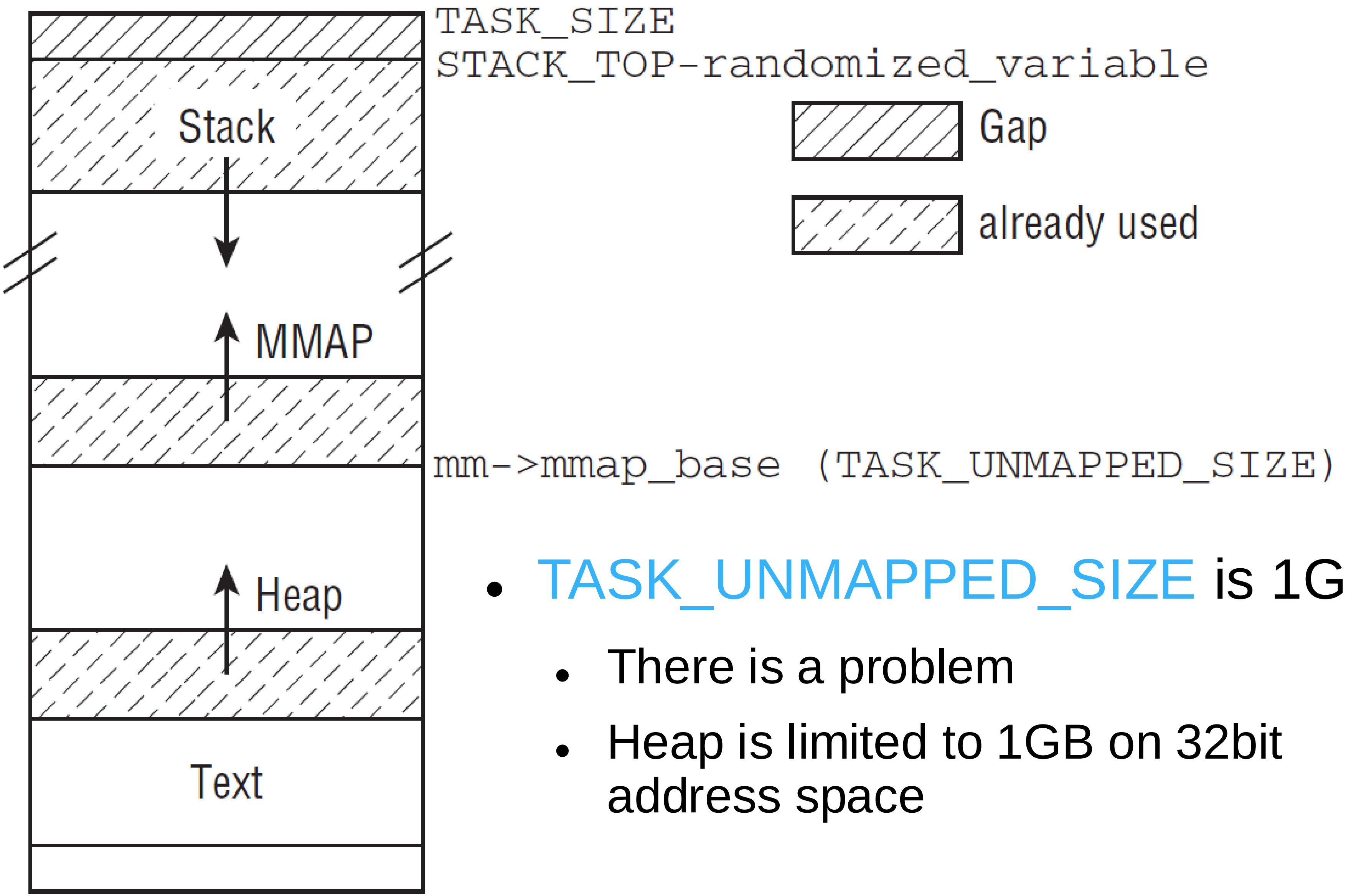
Detour: Layout of a small 32bit address space

Discussion: 32 bits



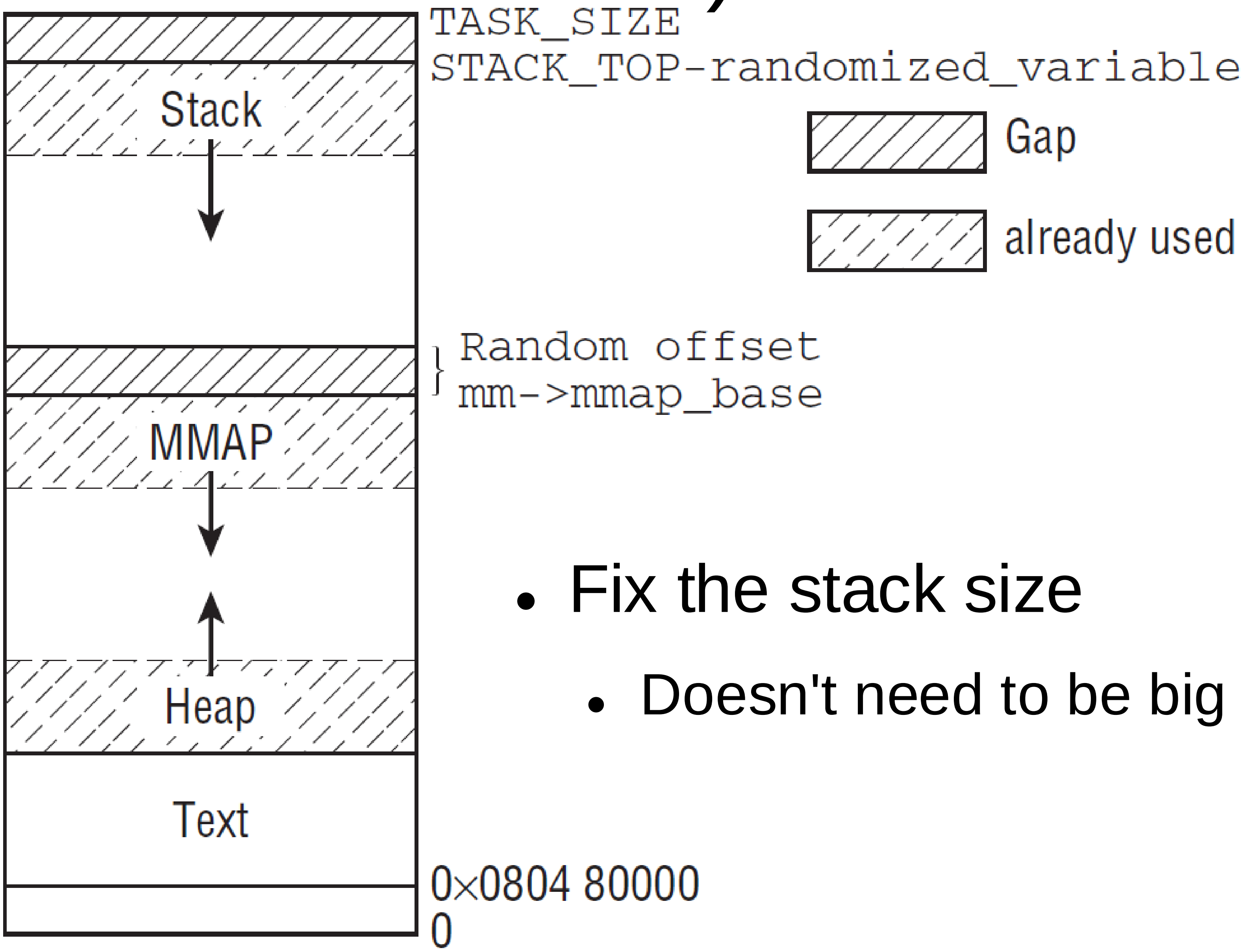
- Text start is defined by ELF
  - Starts at 0x08048000
  - 128MB

Discussion: 32 bits



- TASK_UNMAPPED_SIZE is 1GB
  - There is a problem
  - Heap is limited to 1GB on 32bit address space

Alternative address space layout (32 bits)

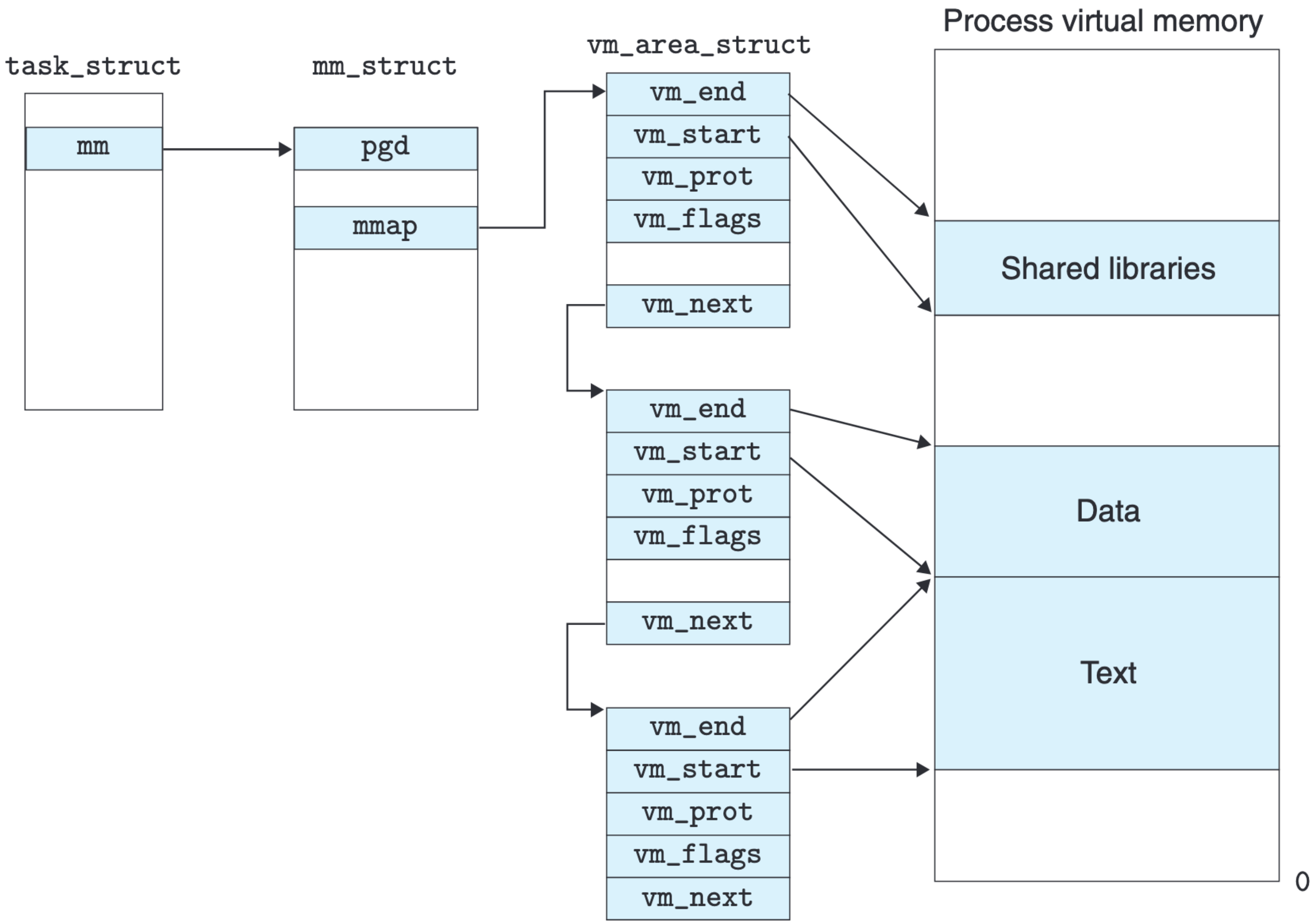


- Fix the stack size
  - Doesn't need to be big

Process memory

- Kernel doesn't trust the user
  - Needs data structures to manage different memory
  - Each address space access is verified

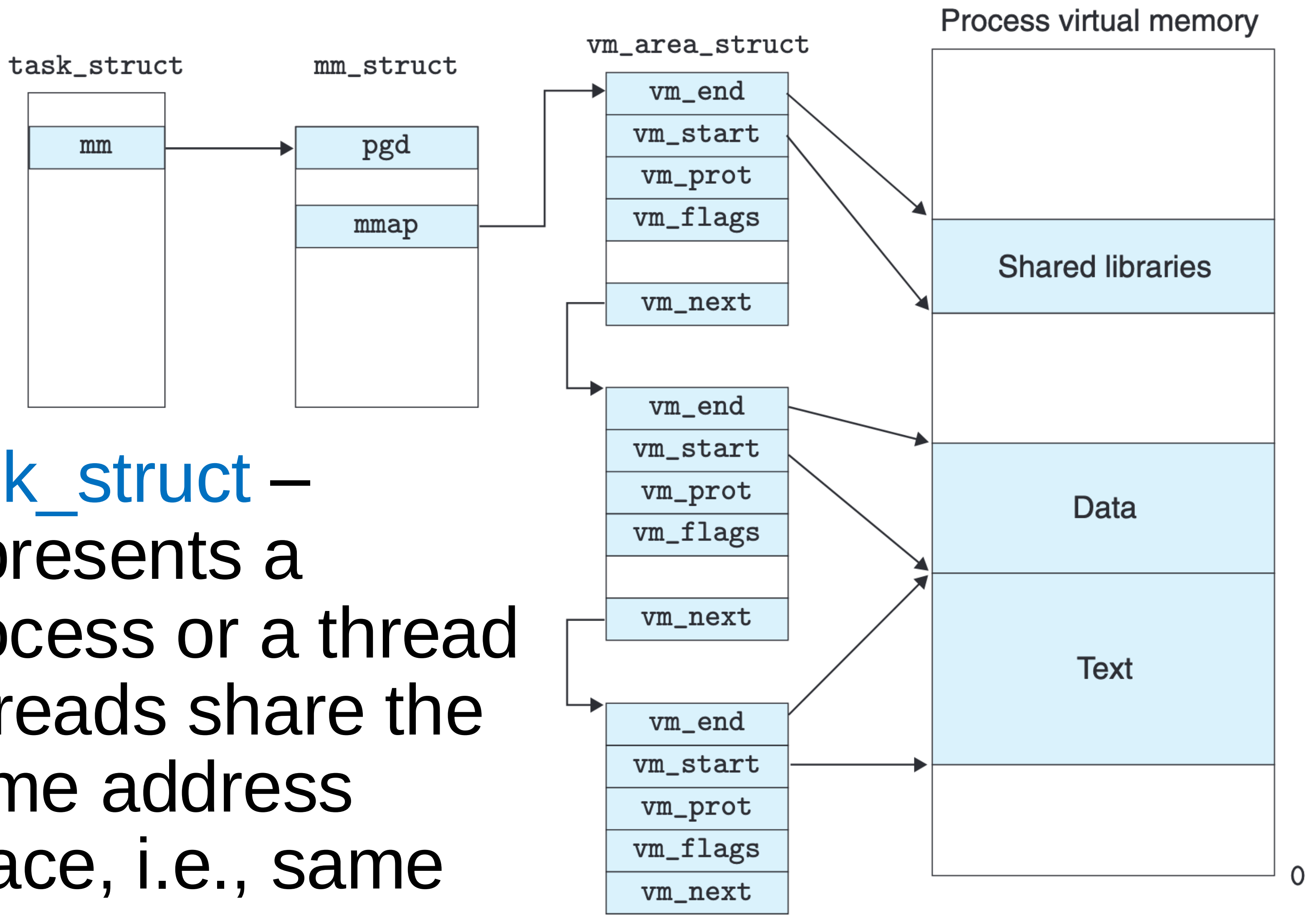
Data structures



Operations

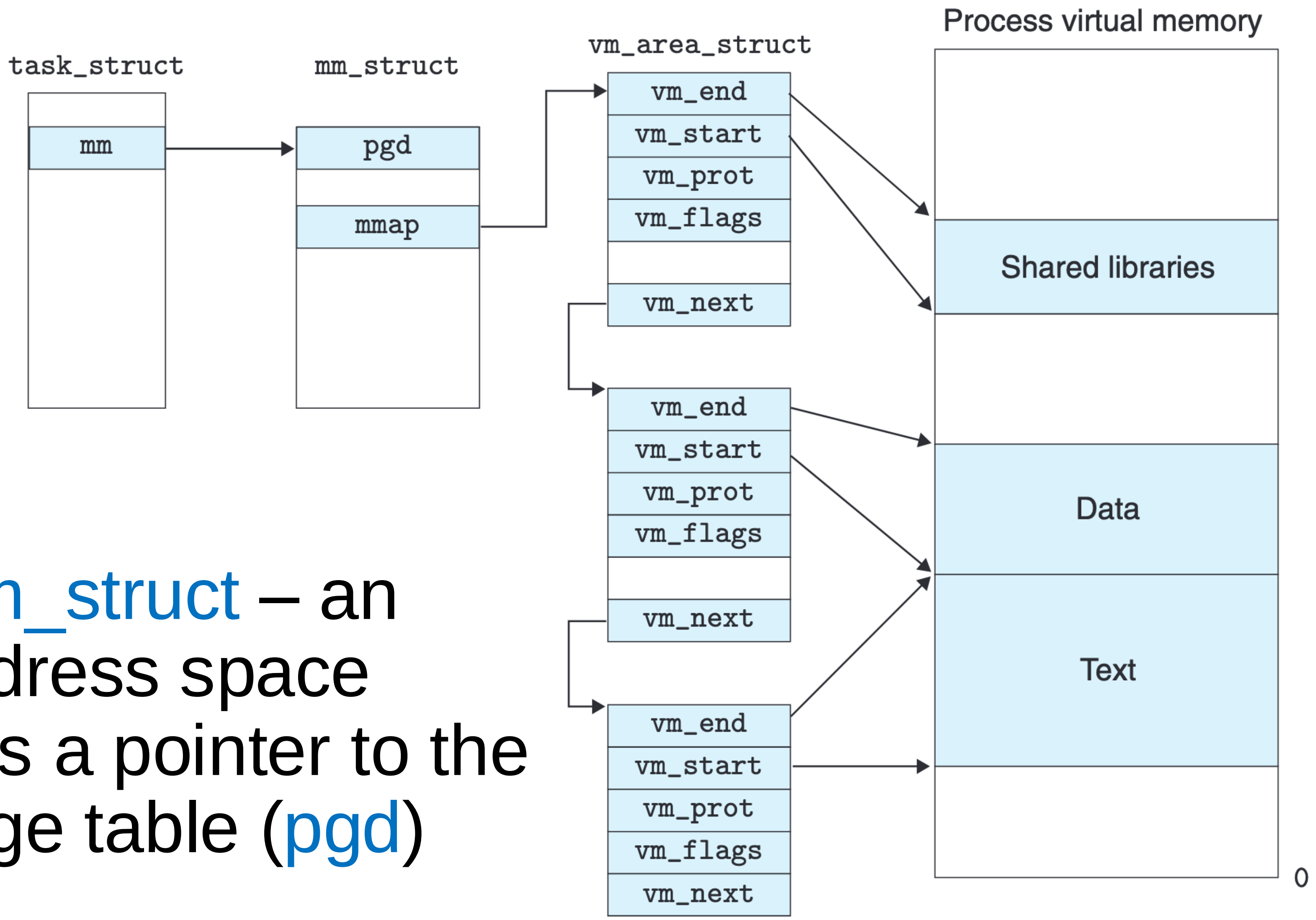
- Find available virtual addresses
  - To map something new, e.g., a new shared library, a file, etc.
- Page in a page on a page fault
  - Figure out the state of the page and allocate it or read it back
- Page out a page
  - Identify idle pages and move them to swap
- Support copy-on-write (COW) for fork()

Data structures



task_struct –
represents a process or a thread
Threads share the same address space, i.e., same mm_struct

Data structures



mm_struct – an address space
Has a pointer to the page table (pgd)

```
struct mm_struct {
    struct vm_area_struct * mmap; // list of all vm_areas
    rb_root_t mm_rb;              // red-black tree
    struct vm_area_struct * mmap_cache; // last looked up vm area (cached)
    pgd_t * pgd;                  // root of the page table
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
    unsigned long arg_start, arg_end, env_start, env_end;
    unsigned long rss, total_vm, locked_vm;
    unsigned long def_flags;
    unsigned long cpu_vm_mask;
    unsigned long swap_address;
    ...
};
```

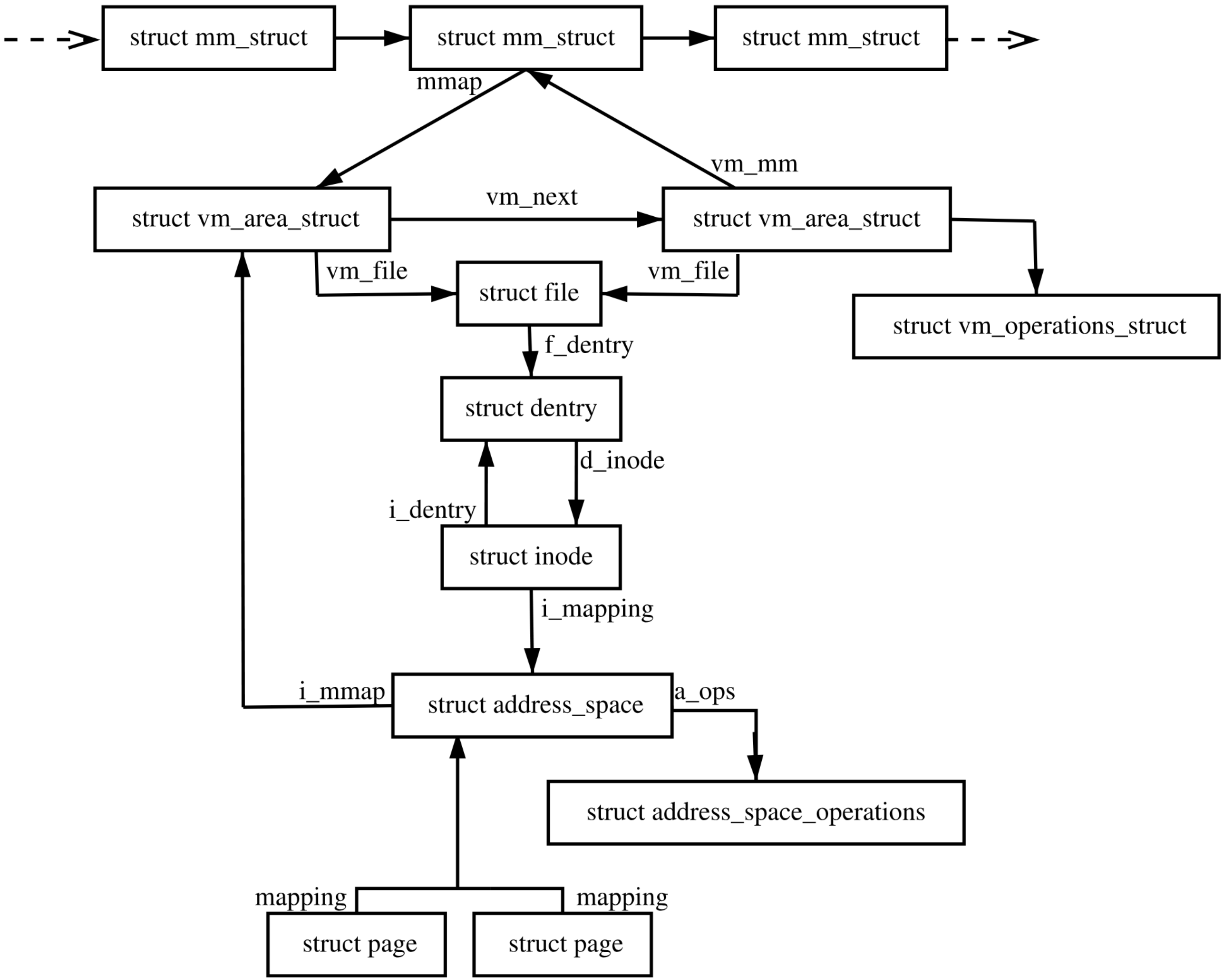
https://pdos.csail.mit.edu/~sbw/links/gorman_book.pdf

```
struct vm_area_struct {
    struct mm_struct * vm_mm;   // mm_struct we belong to
    unsigned long vm_start;     // start virtual address
    unsigned long vm_end;       // end virtual address

    /* linked list of VM areas per task, sorted by address */
    struct vm_area_struct *vm_next;

    pgprot_t vm_page_prot;
    unsigned long vm_flags;

    rb_node_t vm_rb;            // node of the RB tree

    struct vm_area_struct *vm_next_share;
    struct vm_area_struct **vm_pprev_share;

    /* Function pointers to deal with this struct. */
    struct vm_operations_struct * vm_ops;

    /* Information about our backing store: */
    unsigned long vm_pgoff;
    struct file * vm_file;
    unsigned long vm_raend;
    void * vm_private_data;
};
```

https://pdos.csail.mit.edu/~sbw/links/gorman_book.pdf

https://elixir.bootlin.com/linux/v6.10.7/source/include/linux/mm_types.h#L648

Data structures



```
struct vm_operations_struct {
void (*open)(struct vm_area_struct * area);
void (*close)(struct vm_area_struct * area);
struct page * (*nopage)(struct vm_area_struct * area,
                unsigned long address, int unused);
};
```

- nopage() – handles a page fault
- For example, filemap nopage() will locate the page in the page cache or read it in from disk

https://pdos.csail.mit.edu/~sbw/links/gorman_book.pdf

```
struct vm_area_struct {
    struct mm_struct * vm_mm;   // mm_struct we belong to
    unsigned long vm_start;     // start virtual address
    unsigned long vm_end;       // end virtual address
    /* linked list of VM areas per task, sorted by address */
    struct vm_area_struct *vm_next;
    pgprot_t vm_page_prot;
    unsigned long vm_flags;
    rb_node_t vm_rb;            // node of the RB tree
    struct vm_area_struct *vm_next_share;
    struct vm_area_struct **vm_pprev_share;
    /* Function pointers to deal with this struct. */
    struct vm_operations_struct * vm_ops;
    /* Information about our backing store: */
    unsigned long vm_pgoff;
    struct file * vm_file;      // will lead to an "address space"
    unsigned long vm_raend;
    void * vm_private_data;
};
```

https://pdos.csail.mit.edu/~sbw/links/gorman_book.pdf

https://elixir.bootlin.com/linux/v6.10.7/source/include/linux/mm_types.h#L648

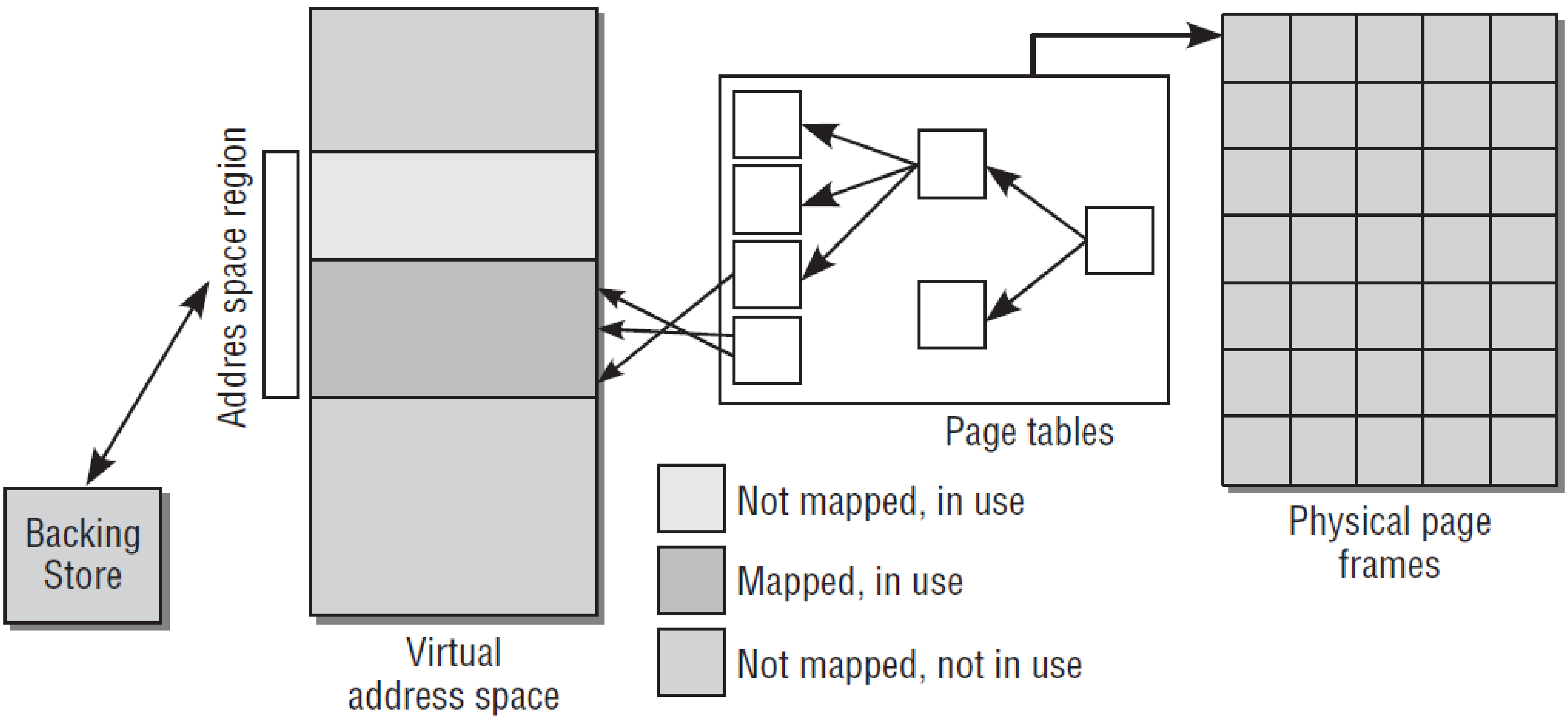
Operations

- Find available virtual addresses
  - To map something new, e.g., a new shared library, a file, etc.
- get_unmapped_area() – can be architecture specific, but at a high level uses vma->vm_next to iterate through the address space
- https://elixir.bootlin.com/linux/v6.10.7/source/mm/mmap.c#L1586

Operations

- Page in a page on a page fault
    - Figure out the state of the page and allocate it or read it back

Demand paging



- Allocation and filling pages with data on demand

Demand paging

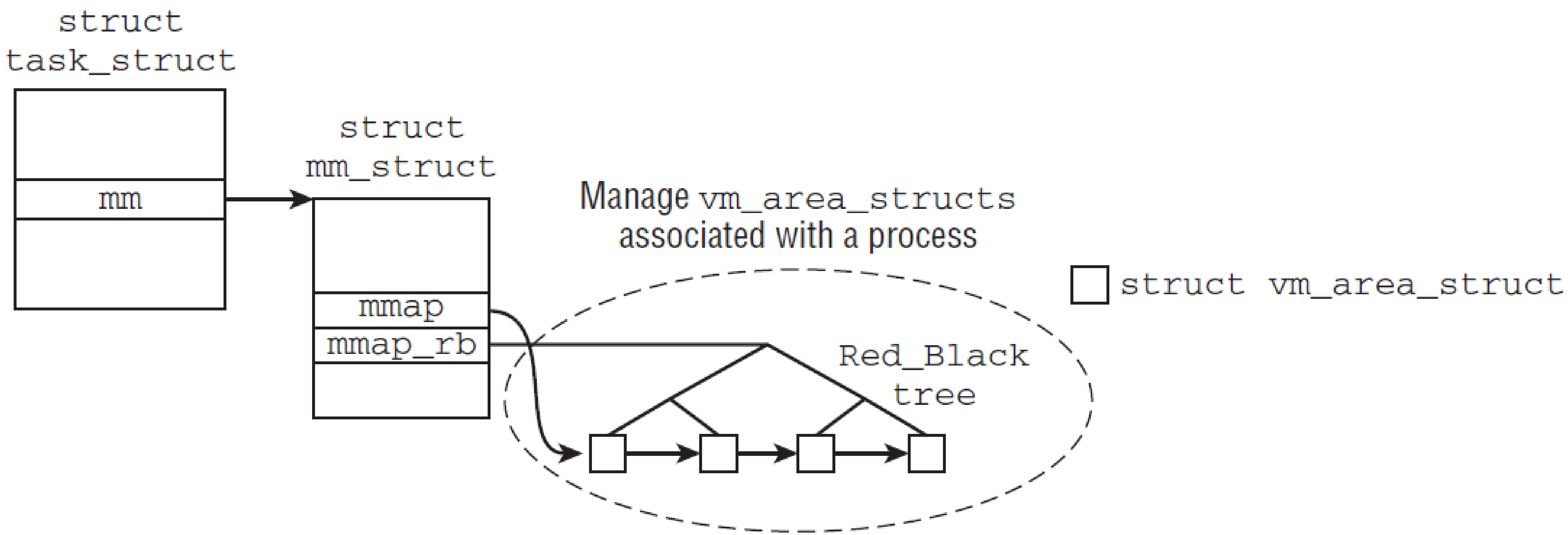
- A process tries to access a part of the address space which cannot be resolved through page tables
- Processor triggers a page fault
- The kernel runs through the process address space data structures
  - Find appropriate backing store
- Kernel allocates and fills the physical page with data from the backing store
- The page is mapped into the address space of a process by updating the page tables

Page fault possible reasons

Exception	Type	Action
Region valid, but page not allocated	Minor	Allocate a page frame from the physical page allocator.
Region not valid but is beside an expandable region like the stack	Minor	Expand the region and allocate a page.
Page swapped out, but present in swap cache	Minor	Re-establish the page in the process page tables and drop a reference to the swap cache.
Page swapped out to backing storage	Major	Find where the page with information is stored in the PTE and read it from disk.
Page write when marked read-only	Minor	If the page is a COW page, make a copy of it, mark it writable and assign it to the process. If it is in fact a bad write, send a `SIGSEGV` signal.
Region is invalid or process has no permissions to access	Error	Send a `SEGSEGV` signal to the process.
Fault occurred in the kernel portion address space	Minor	If the fault occurred in the `vmalloc` area of the address space, the current process page tables are updated against the master page table held by `init_mm`. This is the only valid kernel page fault that may occur.
Fault occurred in the userspace region while in kernel mode	Error	If a fault occurs, it means a kernel system did not copy from userspace properly and caused a page fault. This is a kernel bug that is treated quite severely.

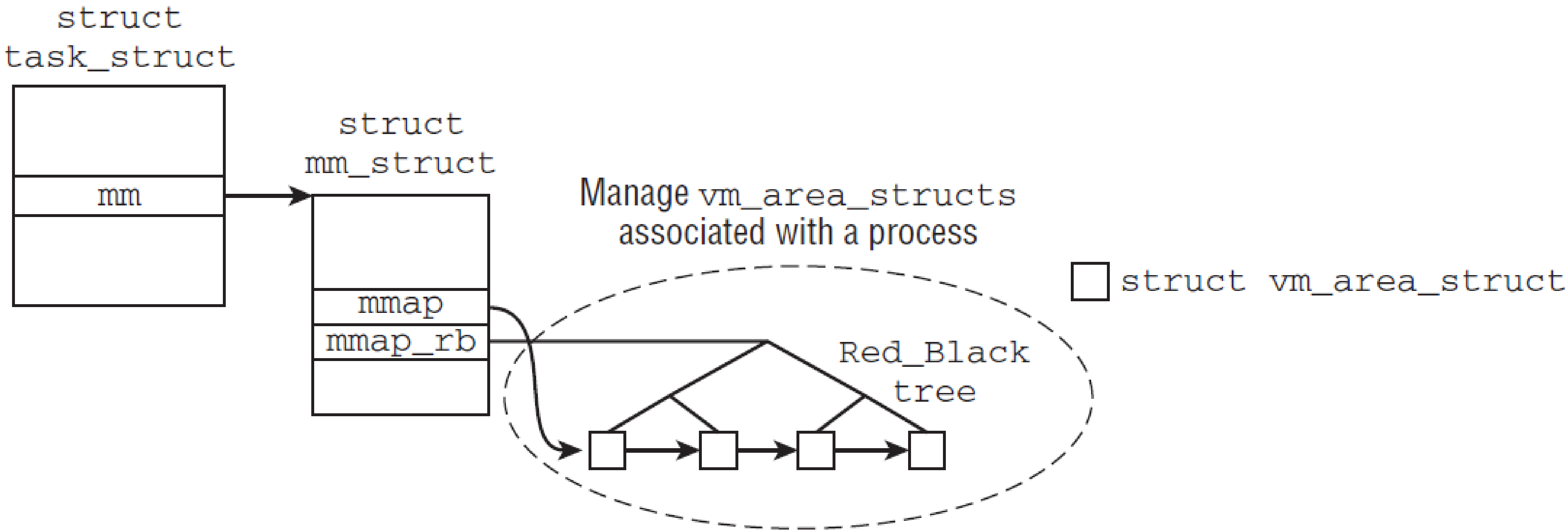
Memory map and vm areas

- All areas are kept as
  - Linked list
  - Red-black tree



Pagefault

- For the current process
  - Represented with the task_struct
  - Walk the mm->mmap_rb to locate a vm_area_struct for the faulting virtual address



Page fault: high-level

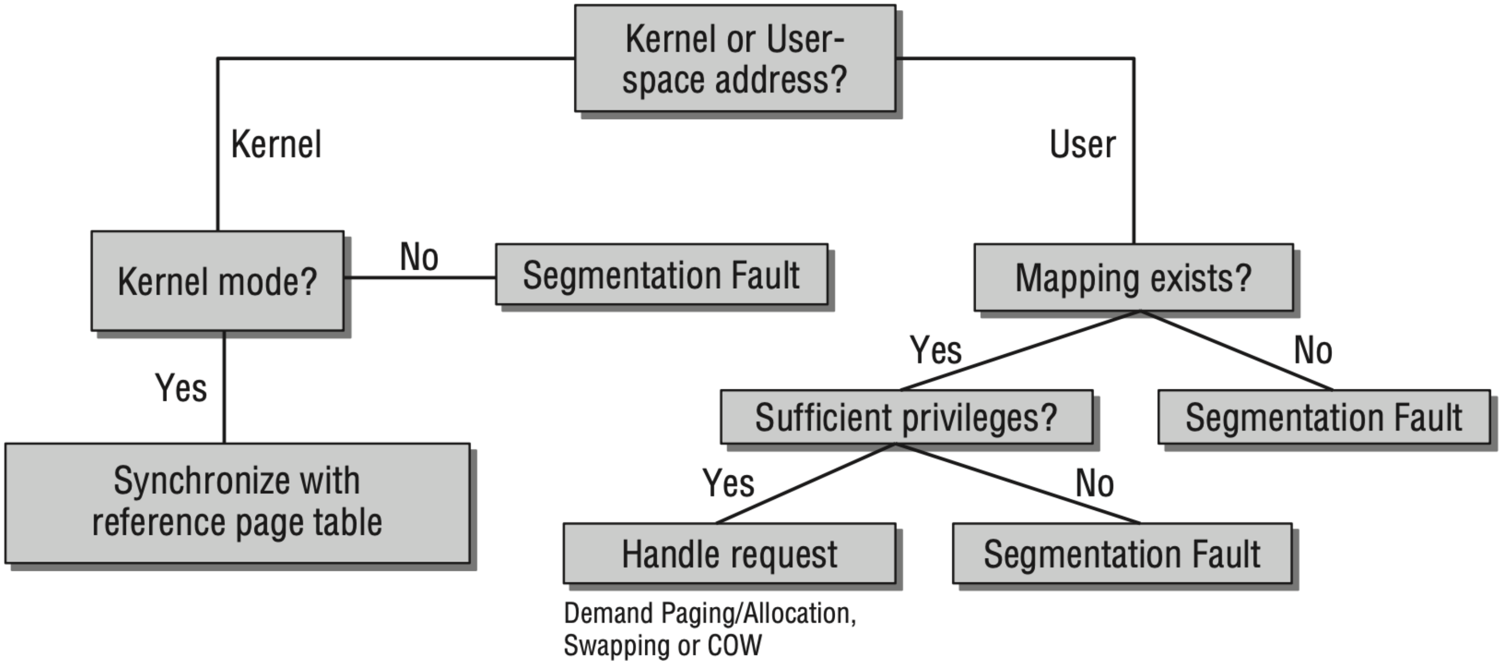
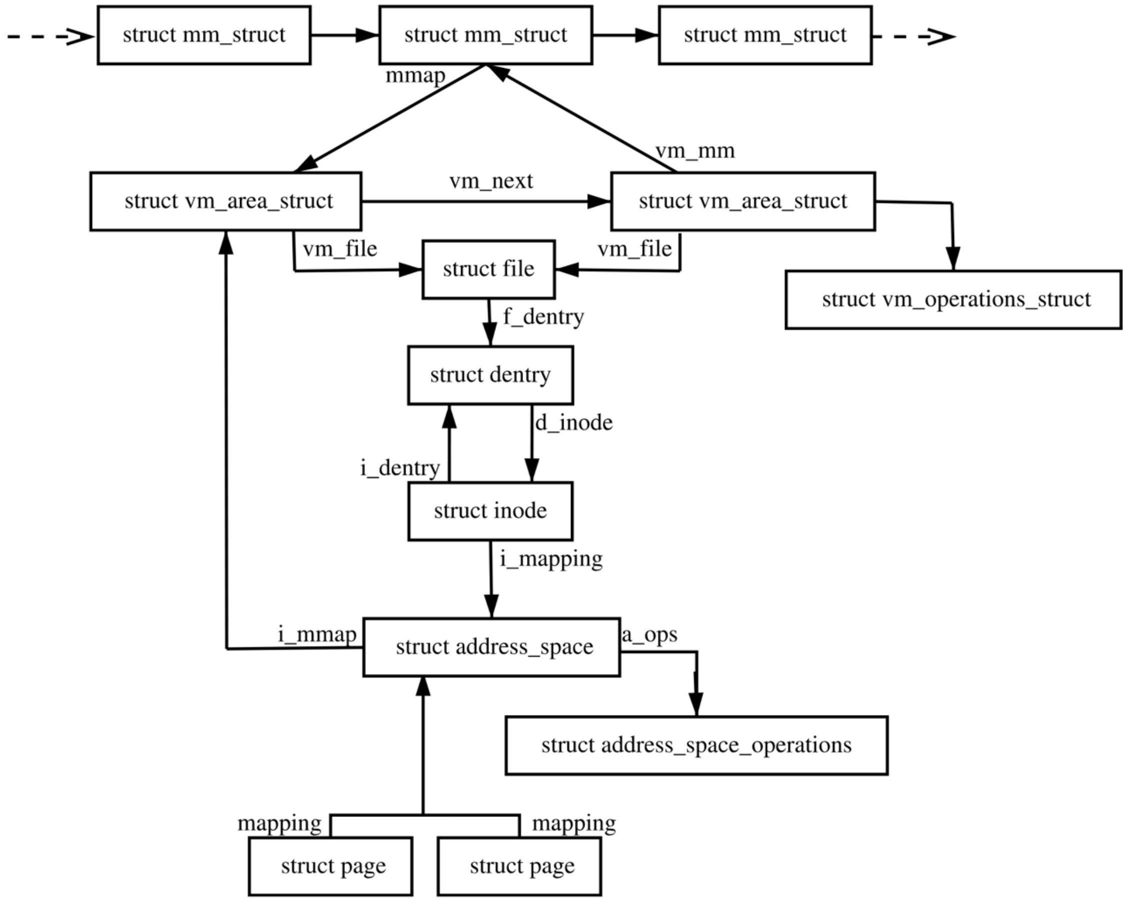


Figure 4-17: Potential options for handling page faults.

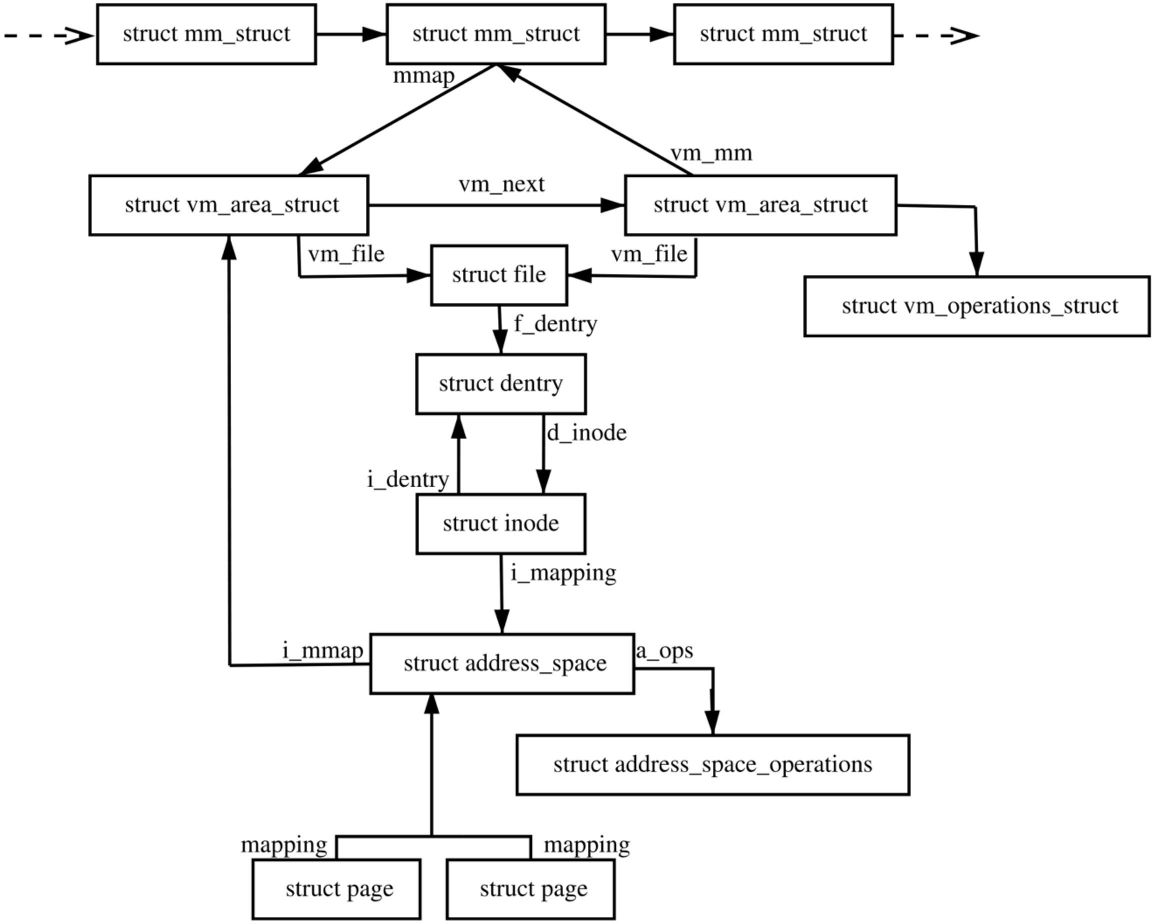
Pagefault (2)

- Each vm_area_struct has a pointer to a vm_file backing this area



Pagefault (3)

- Each address_space has a set of function calls to read data from a backing device



Operations

- Page out a page
  - Identify idle pages and move them to swap

More information

- The RB tree is sufficient to look up a page on a page fault
- More information is needed however for
  - Finding which file backs up each memory area
  - Finding all virtual address spaces in which each page is mapped
    - This is used for swapping out
    - Taking a page (not frequently used) and unmapping it from all address spaces

Additional data structures

- Pages represent either
  - Anonymous pages
    - Not backed up by files, e.g., allocated by mmap() for use by malloc(), i.e., heap
  - Region in a file or a block device
    - Each process has a private file pointer (struct file)
    - Files point to inodes (struct inode)

Reverse mapping

- Connection between a page and all address spaces it is mapped into
  - Used for swapping
  - Each page maintains a counter for the number of times it’s mapped

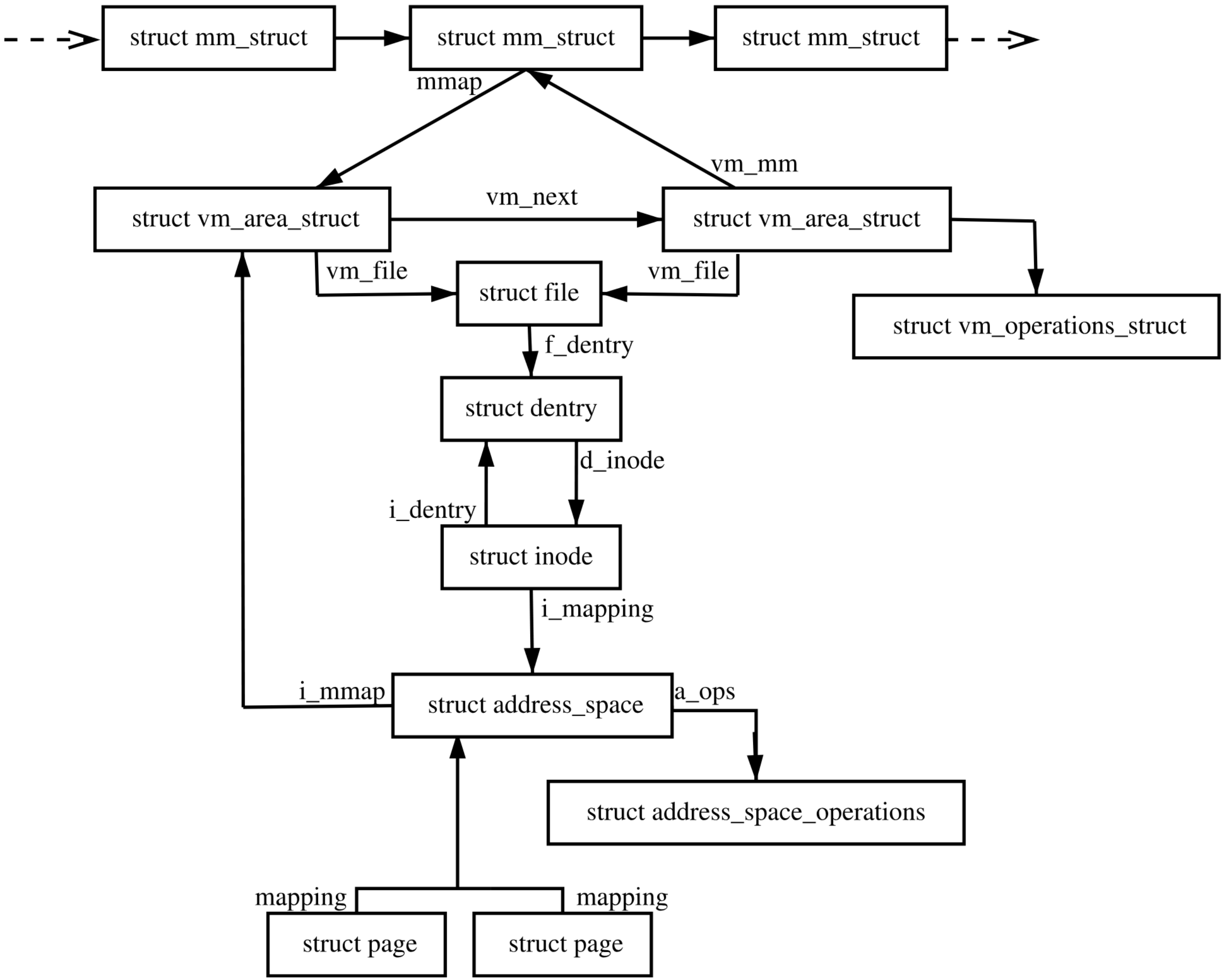
mm.h

```
struct page {
....
       atomic_t _mapcount;              /* Count of ptes mapped in mms,
                                         * to show when page is mapped
                                         * & limit reverse map searches.
                                         */
...
};
```

Reverse mapping

- Anonymous and file-backed pages are handled differently

Data structures



```
union {
    struct {
        unsigned long private;          /* Mapping-private opaque data:
                                         * usually used for buffer_heads
                                         * if PagePrivate set; used for
                                         * swp_entry_t if PageSwapCache;
                                         * indicates order in the buddy
                                         * system if PG_buddy is set.
                                         */
        struct address_space *mapping;  /* If low bit clear, points to
                                         * inode address_space, or NULL.
                                         * If page mapped as anonymous
                                         * memory, low bit is set, and
                                         * it points to anon_vma object:
                                         * see PAGE_MAPPING_ANON below.
                                         */
    };

    struct kmem_cache *slab; /* SLUB: Pointer to slab */
    struct page *first_page; /* Compound tail pages */
};
```

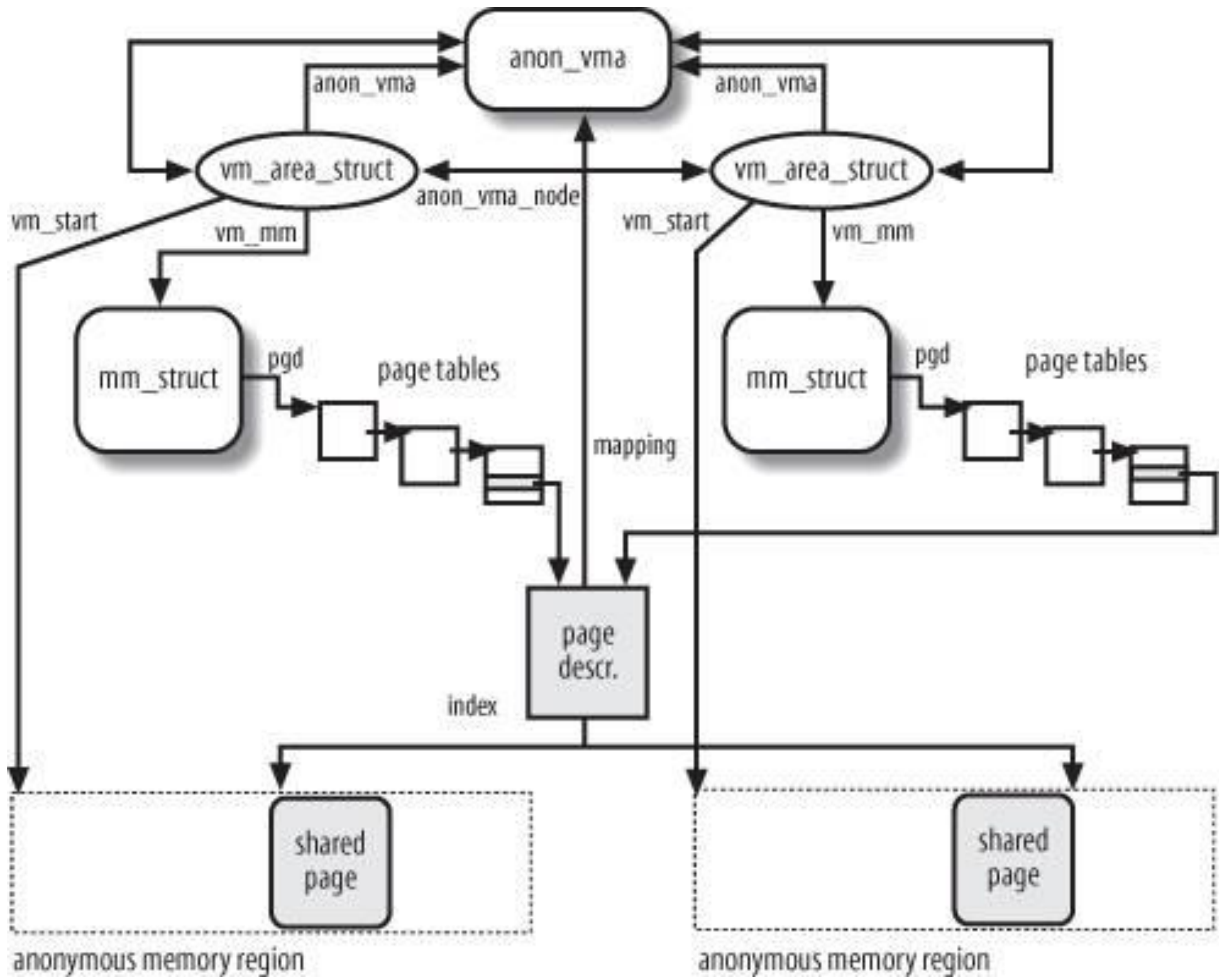
- `mapping` specifies the address space in which a page frame is located. `index` is the offset within the mapping. Address spaces are a very general concept used, for example, when reading a file into memory. An address space is used to associate the file contents (data) with the areas in memory into which the contents are read. By means of a small trick,[7] `mapping` is able to hold not only a pointer, but also information on whether a page belongs to an anonymous memory area that is not associated with an address space. If the bit with numeric value 1 is set in `mapping`, the pointer does *not* point to an instance of `address_space` but to another data structure (`anon_vma`) that is important in the implementation of reverse mapping for anonymous pages; this structure is discussed in Section 4.11.2. Double use of the pointer is possible because `address_space` instances are always aligned with `sizeof(long)`; the least significant bit of a pointer to this instance is therefore 0 on all machines supported by Linux.

Reverse mapping for anonymous pages

- Frequently shared between parent and child processes
  - COW on fork()
  - And even MAP_SHARED + MAP_ANONYMOUS
- Kernel creates an anon_vma data structure
- Maintains all vm_area_struct on a linked list
- Finding a page table entry requires scanning the list
  - The number of anonymous shares is not very large

Data structures after the fork()

Parent and child have the same page mapped



Reverse mapping for mapped pages

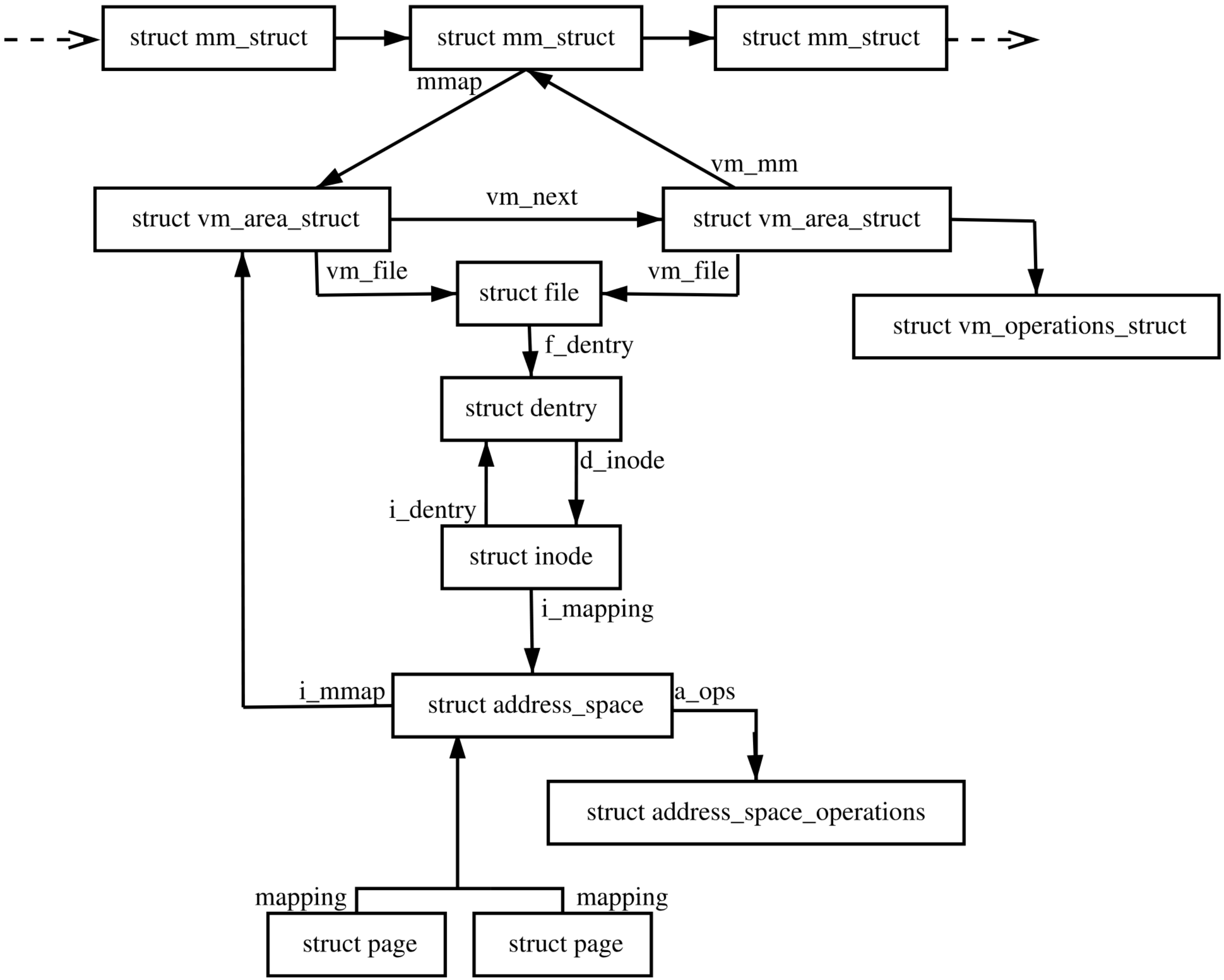
Some libraries, e.g., libc are mapped in every process in the system

Scanning a linked list is prohibitive

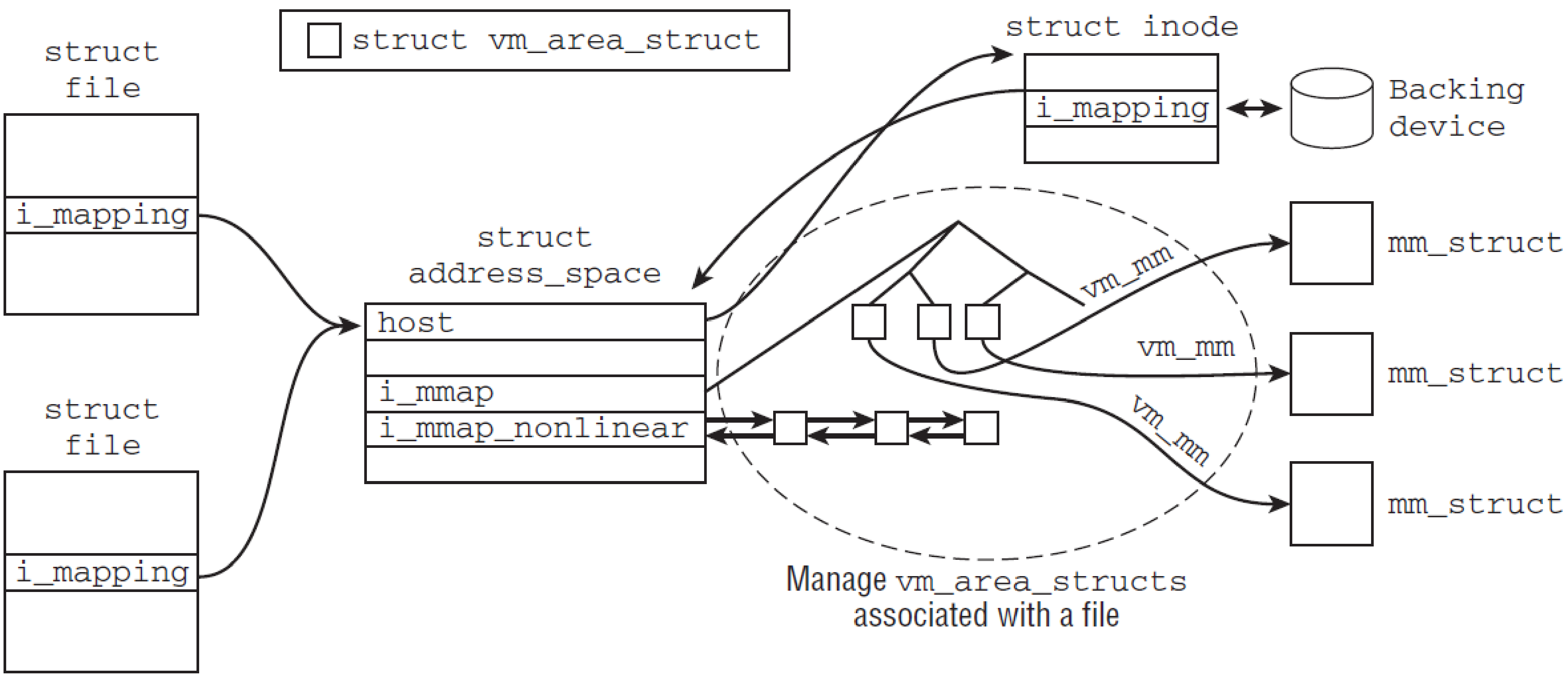
Priority search tree for every file

Stored in the i_mmap field of the address_space

Data structures



Additional data structures



Priority tree

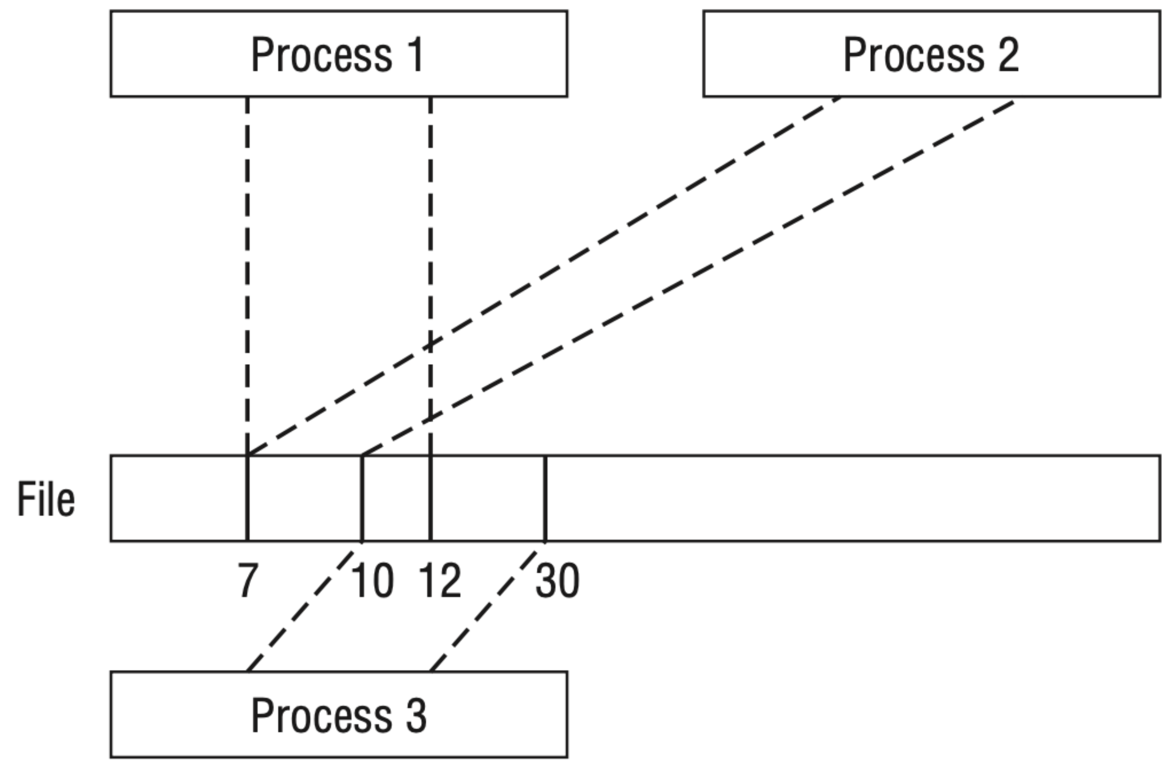


Figure 4-8: Multiple processes can map identical or overlapping regions of a file into their virtual address space.

Queries about the intervals

Priority search tree (PST)
A data structure that represents a set of overlapping intervals
Fast queries about overlapping intervals

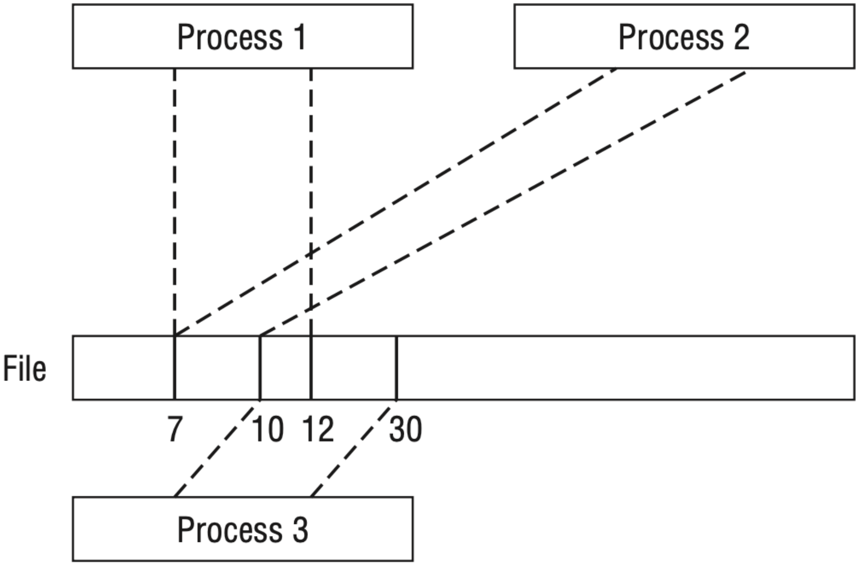
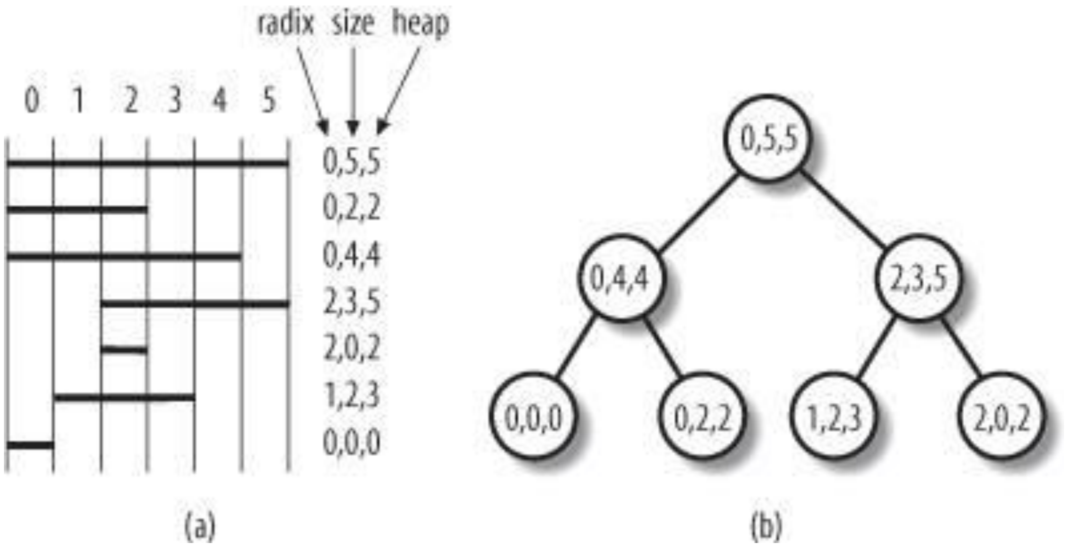


Figure 4-8: Multiple processes can map identical or overlapping regions of a file into their virtual address space.

Example

- Which interval contains “5”
- Start at root (0,5,5)
- Found one!
- Descent into (0,4,4)
- Too small, terminate
- Descent into (2,3,5)
- Found another one!
- Check (1,2,3) and (2,0,2) but both are negative



Thank you!