CS5460/6460: Operating Systems

Lecture 22: Virtual process memory

Anton Burtsev
April, 2014

Virtual process memory

- Each process has a private 4GB address space
  - All addressable memory (32bits)
    - Well, 3GBs out of 4GBs on Linux
  - Isolated from other processes
- But only a small portion of 3GBs is actually used by an application

Process memory

- Only a small portion of 3GBs is actually used by an application
- Memory of different kinds (which?)

Process memory

- Only a small portion of 3GBs is actually used by an application
- Memory of different kind
  - 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

Process memory

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

<mm_types.h>

```
struct mm_struct {
  ...
  unsigned long (*get_unmapped_area) (struct file *filp,
            unsigned long addr, unsigned long len,
            unsigned long pgoff, u
  ...
  unsigned long mmap_base; /* base
  unsigned long task_size; /* size of t          space */
  ...
  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;
  ...
}
```

- Start and end for code and data
  - Size doesn't change once ELF is mapped

```
<mm_types.h>
struct mm_struct {
   ...
   unsigned long (*get_unmapped_area) (struct file *filp,
               unsigned long addr, unsigned long len,
               unsigned long pgoff, unsigned long flags);
   ...
   unsigned long mmap_base; /* base
   unsigned long task_size; /* size of
   ...
   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;
   ...
}
```

- Start of the heap
- Brk – end of the heap

```
<mm_types.h>
struct mm_struct {
    ...
    unsigned long (*get_unmapped_area) (struct file *filp,
                unsigned long addr, uns
                unsigned long pgoff, un
    ...
    unsigned long mmap_base; /* base
    unsigned long task_size; /* size of                space */
    ...
    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;
    ...
}
```

- Start and end of the program arguments
- Start and end of the environment
- Both mapped at the topmost area of the stack

<mm_types.h>

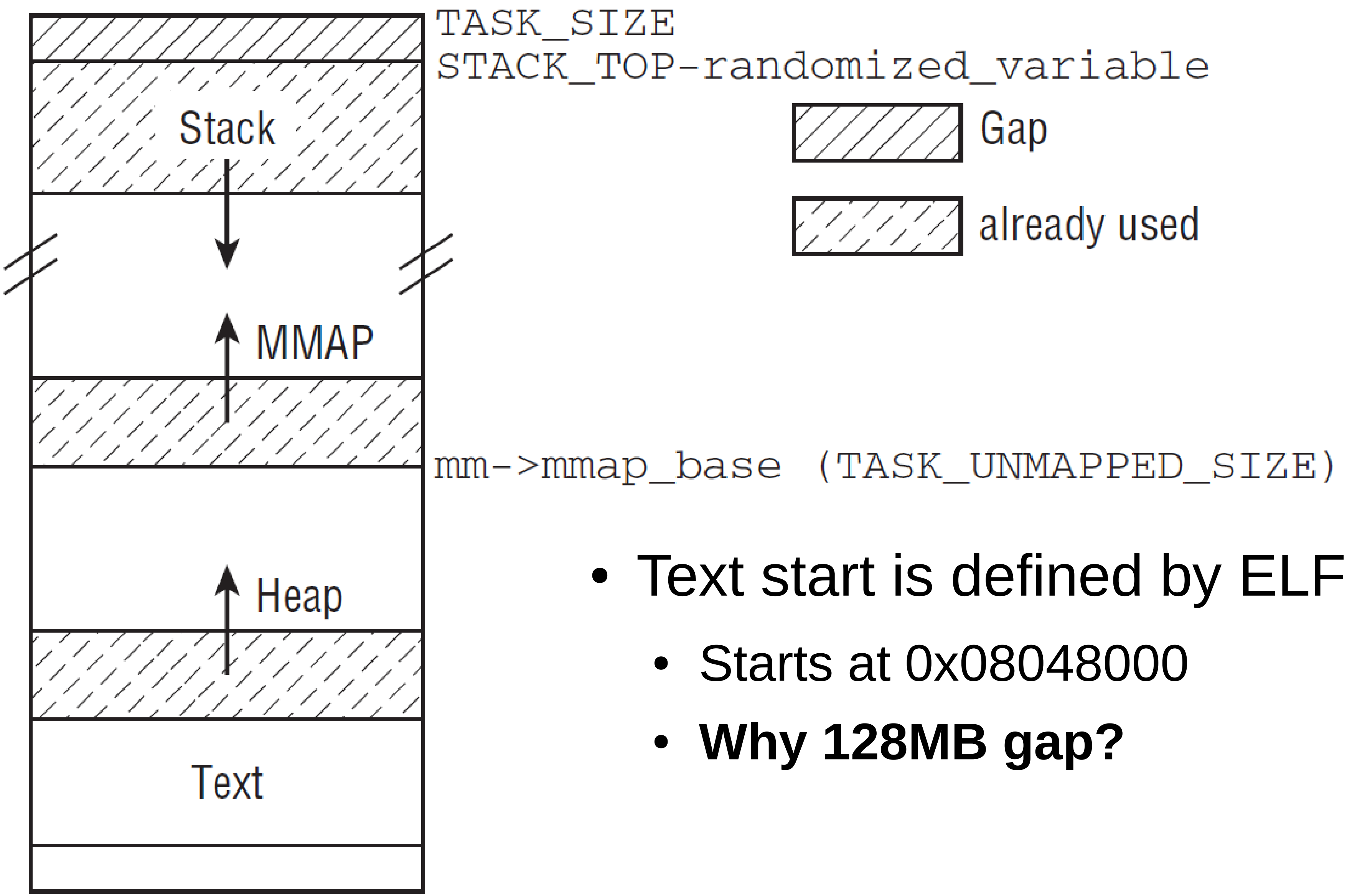
```
struct mm_struct {
  ...
  unsigned long (*get_unmapped_area) (struct file *filp,
            unsigned long addr, unsigned long len,
            unsigned long pgoff, unsigned long flags);
  ...
  unsigned long mmap_base; /* base of mmap area */
  unsigned long task_size; /* size of task vm space */
  ...
  unsigned long start_code, end_co
  unsigned long start_brk, brk, star
  unsigned long arg_start, arg_end,
  ...
}
```

- Start point for the memory mappings in the address space

```
<mm_types.h>
struct mm_struct {
    ...
    unsigned long (*get_unmapped_area) (struct file *filp,
                unsigned long addr, unsigned long len,
                unsigned long pgoff, unsigned long flags);
    ...
    unsigned long mmap_base; /* base o   ap area */
    unsigned long task_size; /* size of t      space */
    ...
    unsigned long start_code, end_code
    unsigned long start_brk, brk, star
    unsigned long arg_start, arg_end,
    ...
}
```

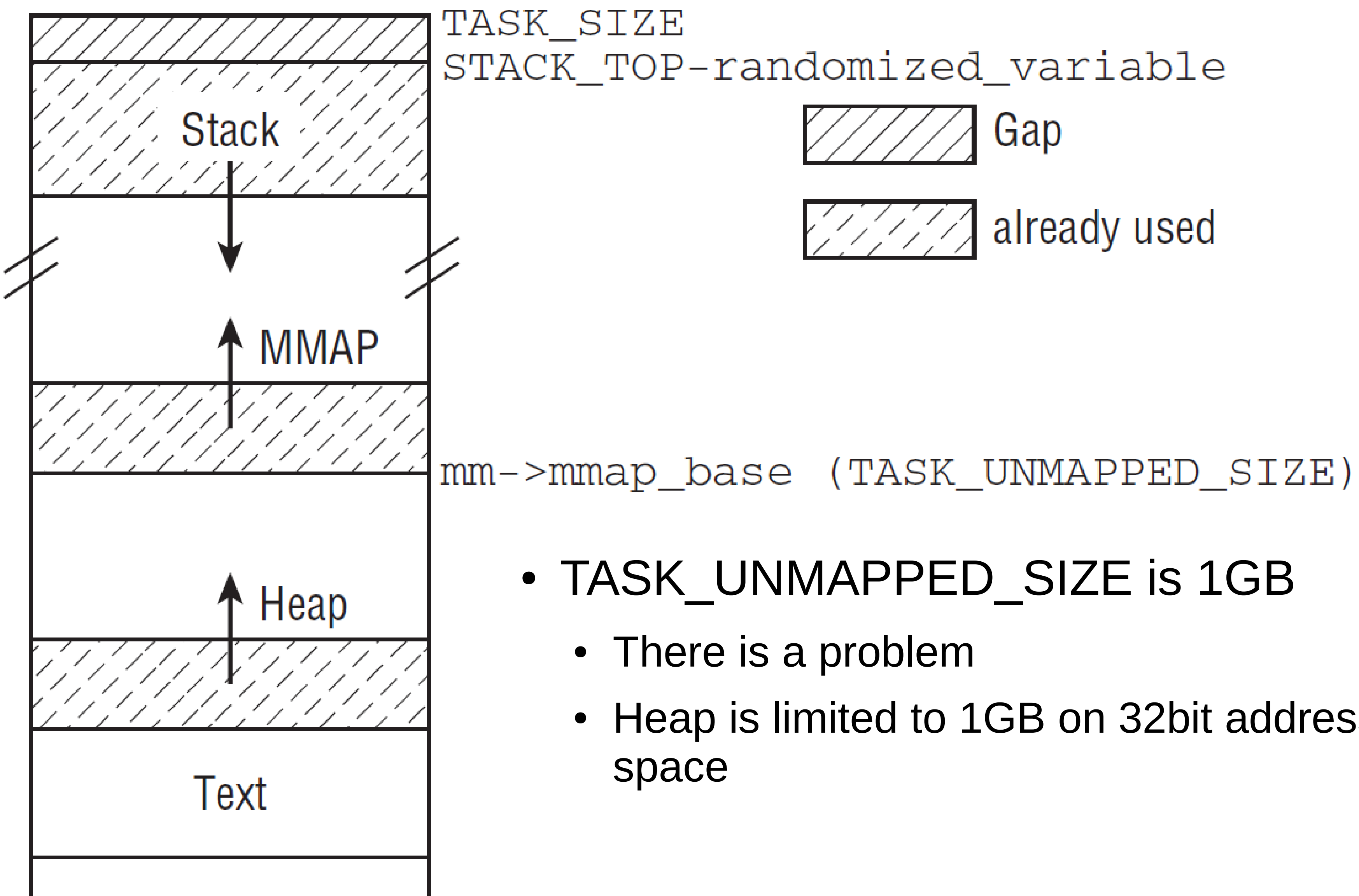
- Get suitable location for the next mapping in the mmap area

Address space layout



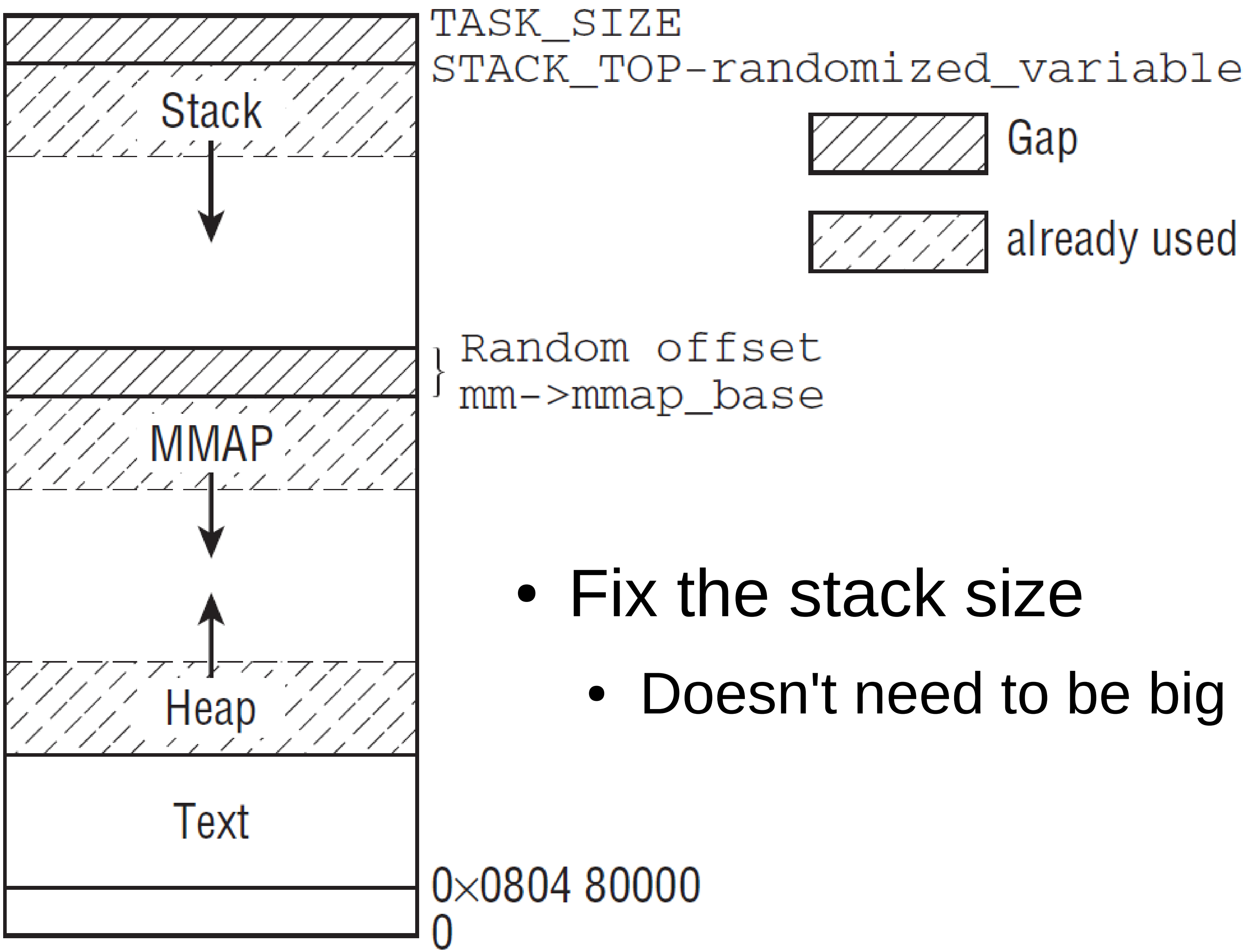
- Text start is defined by ELF
  - Starts at 0x08048000
  - **Why 128MB gap?**

Address space layout



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

Alternative address space layout



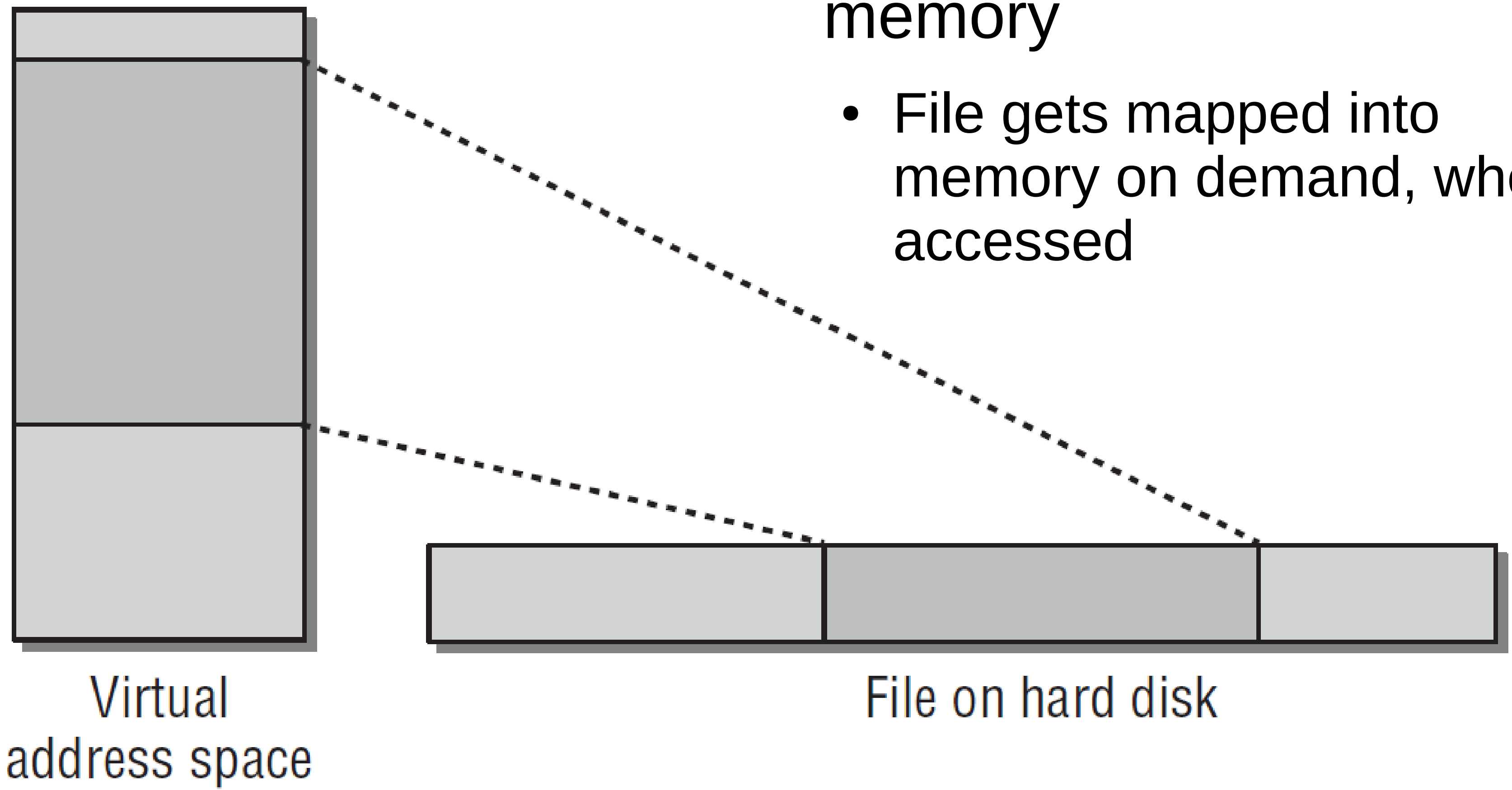
- Fix the stack size
  - Doesn't need to be big

Memory mapping

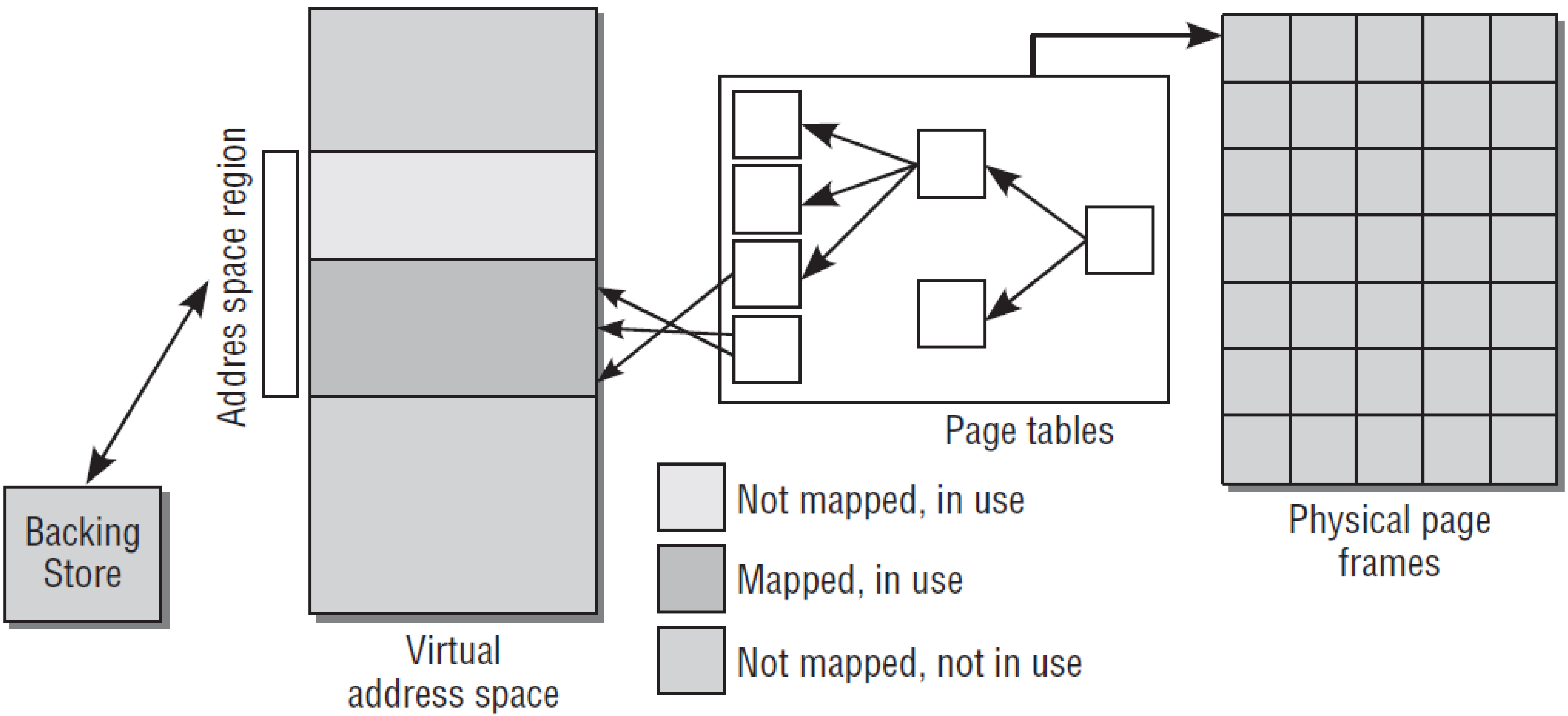
- In a typical system total size of all virtual address spaces of all processes is much larger than available physical memory
  - Only some parts of the virtual address space are backed by physical pages
  - Kernel keeps information about pages associated with parts of virtual address space
- Can you think of a system/setup when this is not true?

Example: editing of a large file

- Only small part of a file is actually mapped into memory
  - File gets mapped into memory on demand, when accessed



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

Map of the process virtual memory

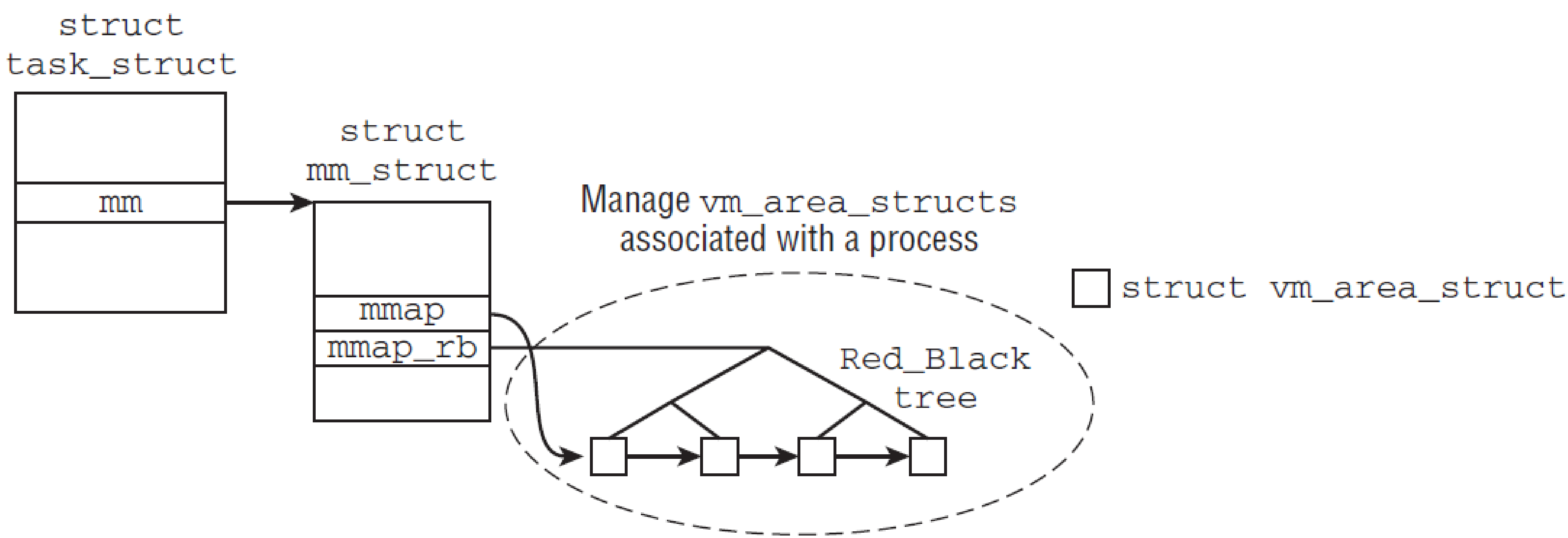
```
<mm_types.h>
struct mm_struct {
  struct vm_area_struct * mmap; /* list of VMAs  */
  struct rb_root mm_rb;
  struct vm_area_struct * mmap_cache; /* last find_vma result */
...
}
```

- Each memory area of process virtual address space is described as

```
struct vm_area_struct {
  struct mm_struct * vm_mm; /* The address space we belong to. */
  unsigned long vm_start;   /* Our start address within vm_mm. */
  unsigned long vm_end;     /* The first byte after our end address
                               within vm_mm. */
```

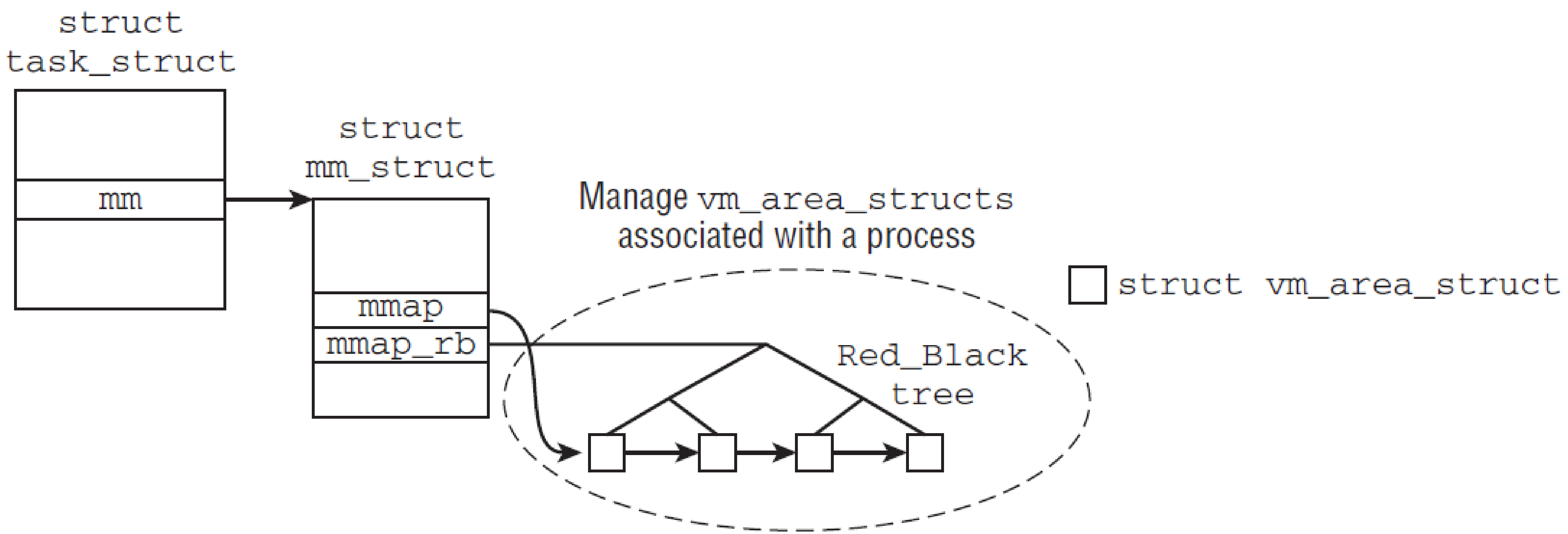
Map of the process virtual memory

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



Page fault

- These data structures are sufficient to find a region for the page which is missed in memory



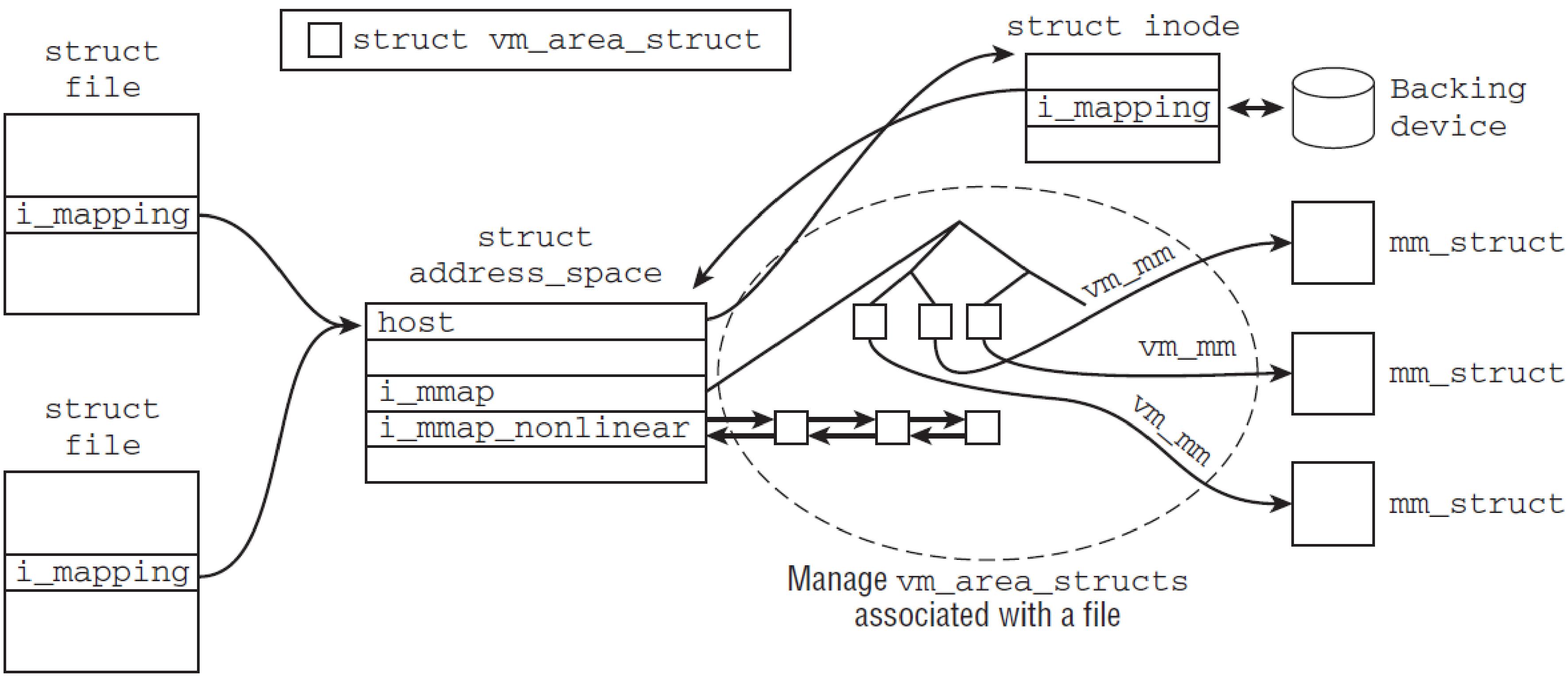
More information

- 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. heap
  - Region in a file or a block device
    - Each process has a private file pointer (`struct file`)
    - Files point to inodes (`struct inode`)

Additional data structures



Additional data structures (definitions)

<fs.h>

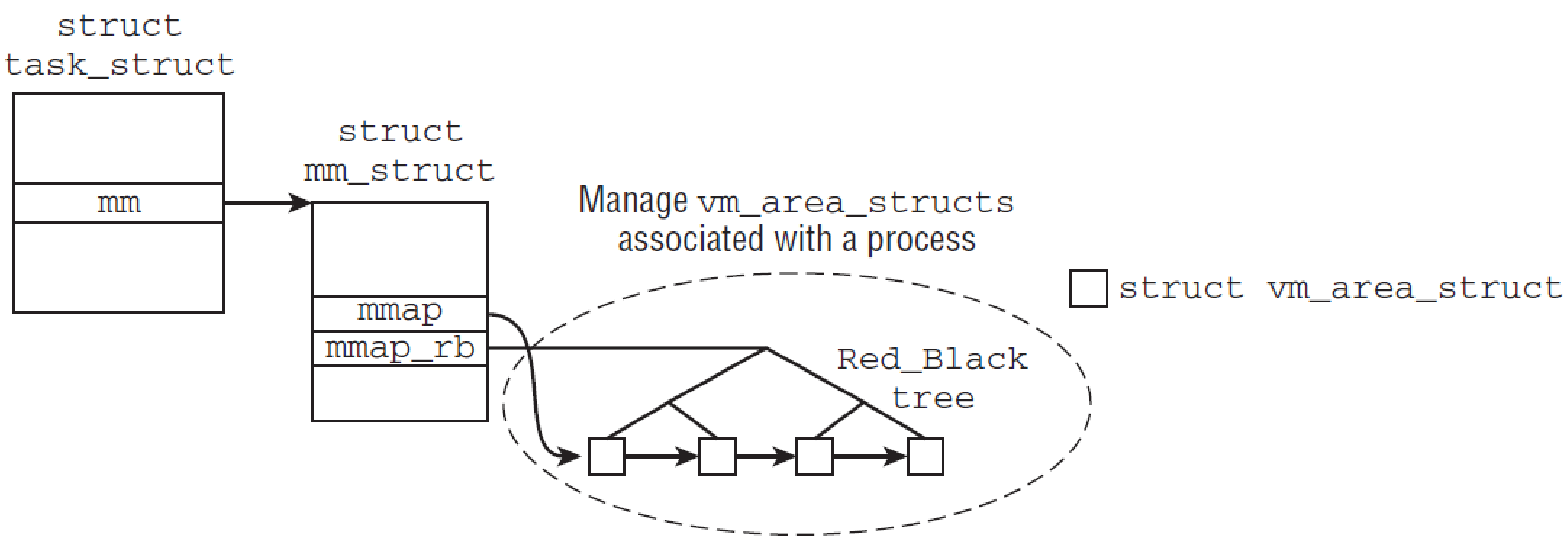
```
struct address_space {
  struct inode *host;                /* owner: inode, block_device */
  ...
  struct prio_tree_root i_mmap;      /* tree of private and shared
                                          mappings */
  struct list_head i_mmap_nonlinear; /*list VM_NONLINEAR mappings */
  ...
}

struct file {
  ...
  struct address_space *f_mapping;
  ...
}

struct inode {
...
struct address_space *i_mapping;
...
}
```

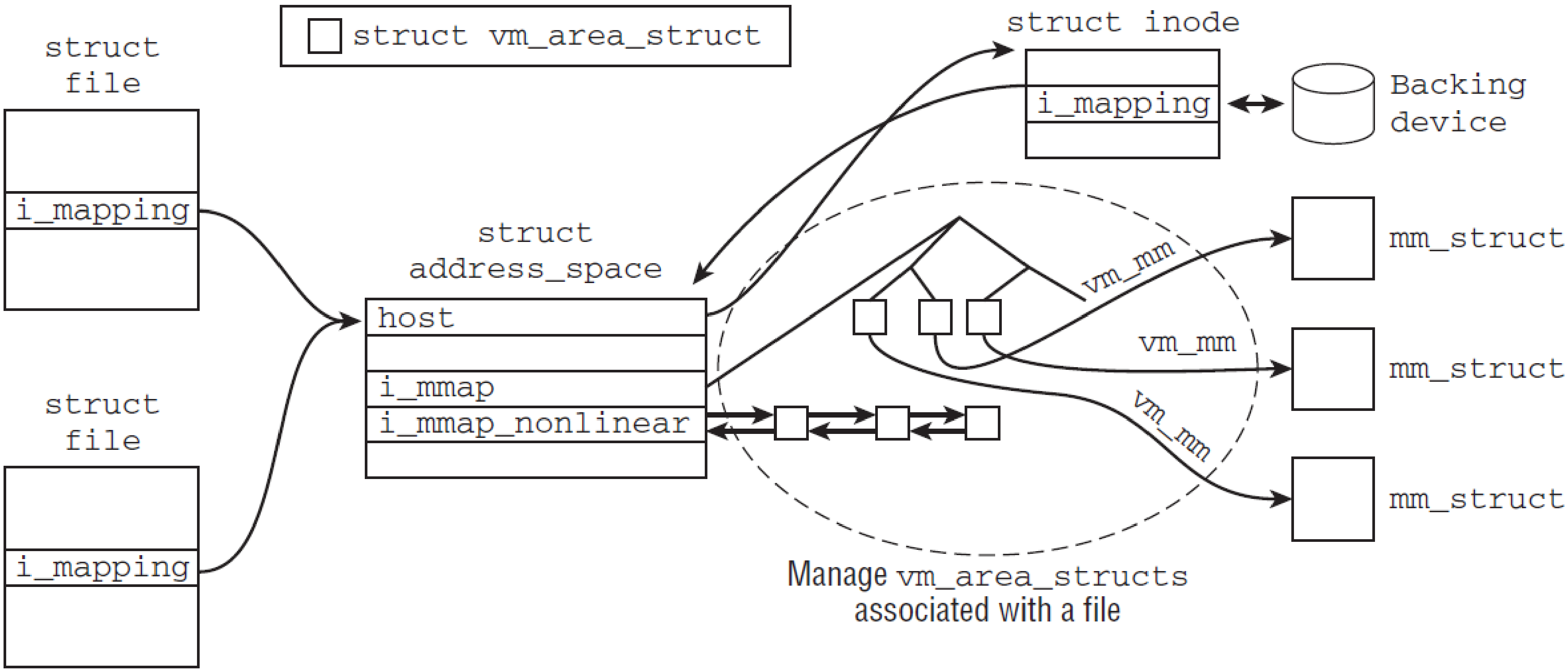
Pagefault

- 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



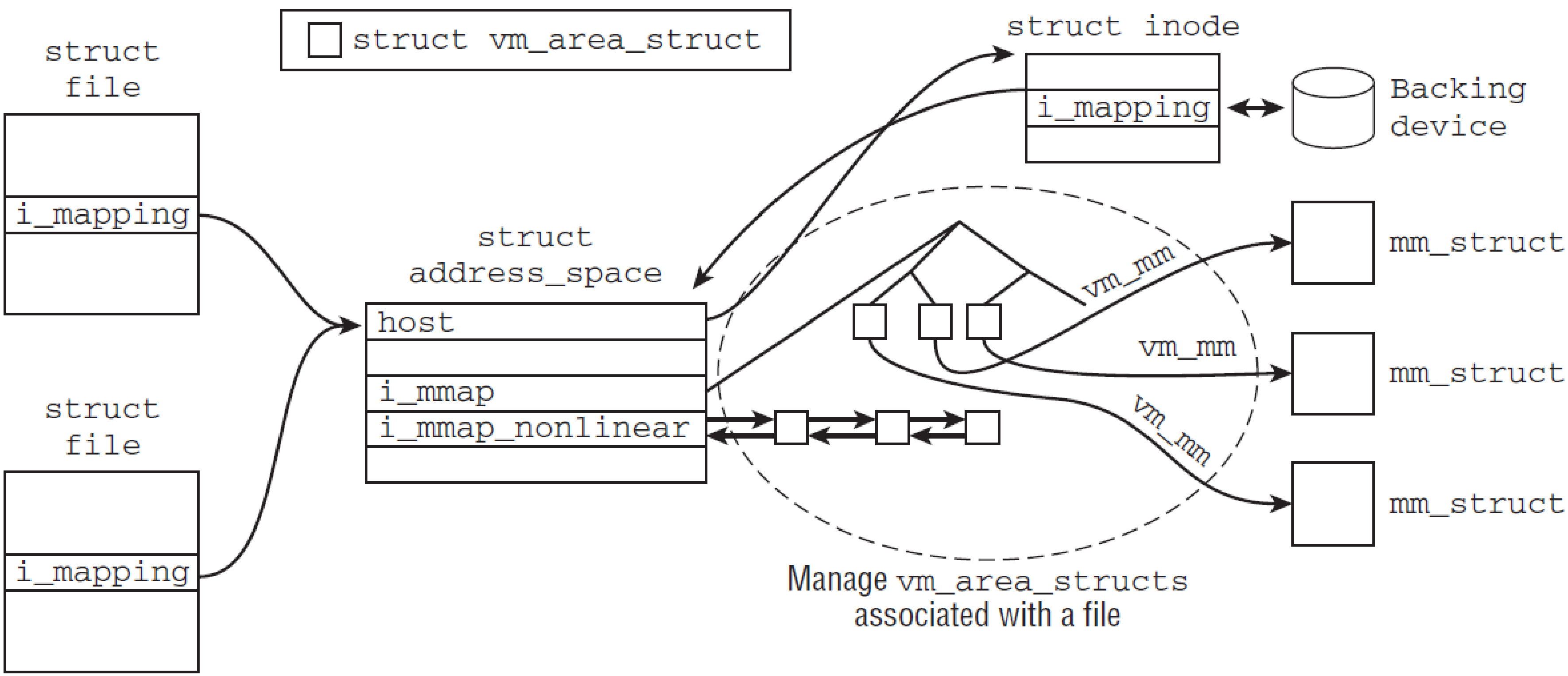
Pagefault (2)

- 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



Conclusion

- Virtual to physical mapping
  - Page tables
- Virtual to file mapping
  - struct address_space
- Page to address spaces mapping
  - Reverse mapping
  - Next time!

Thank you!

Reverse mapping

- Connection between a page and all address spaces it is mapped into
  - Used for swapping