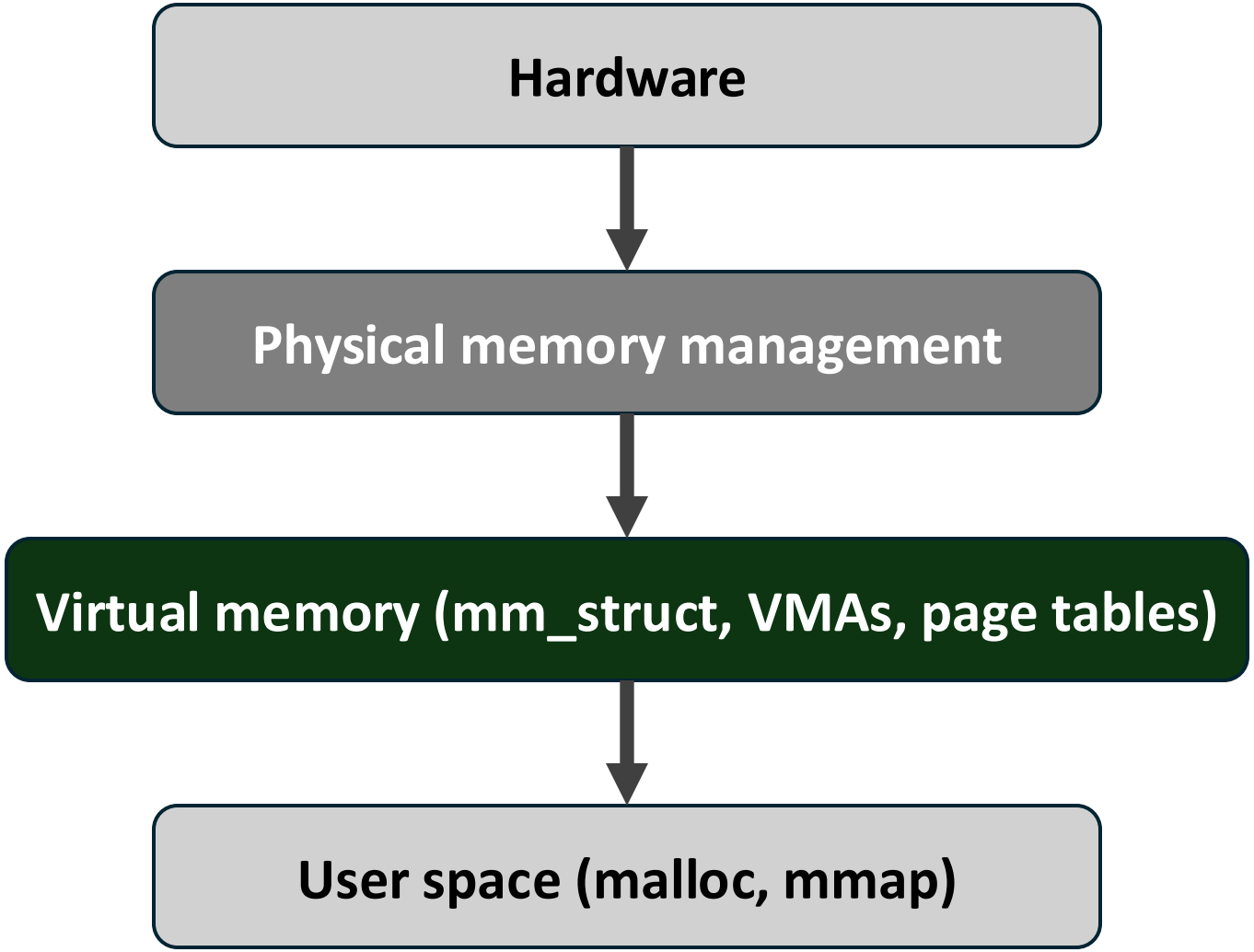
# CS 477
# Advanced Operating System

## Lecture 10: Virtual Memory Management

# Memory management



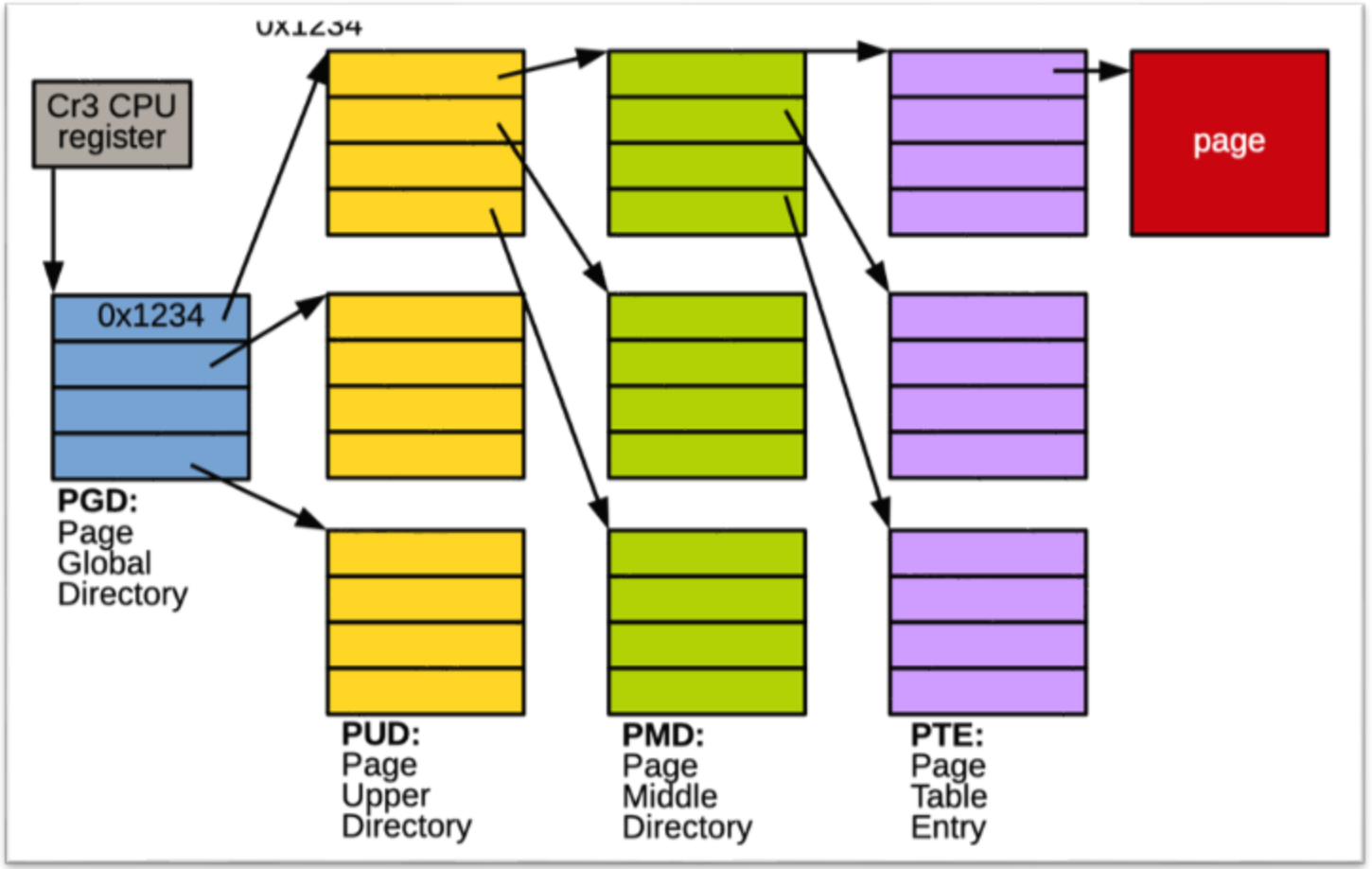
# Today's agenda

- Page tables
- Address space
- Memory descriptor: **`struct mm_struct`**
- Virtual memory area (VMA)
- VMA manipulation

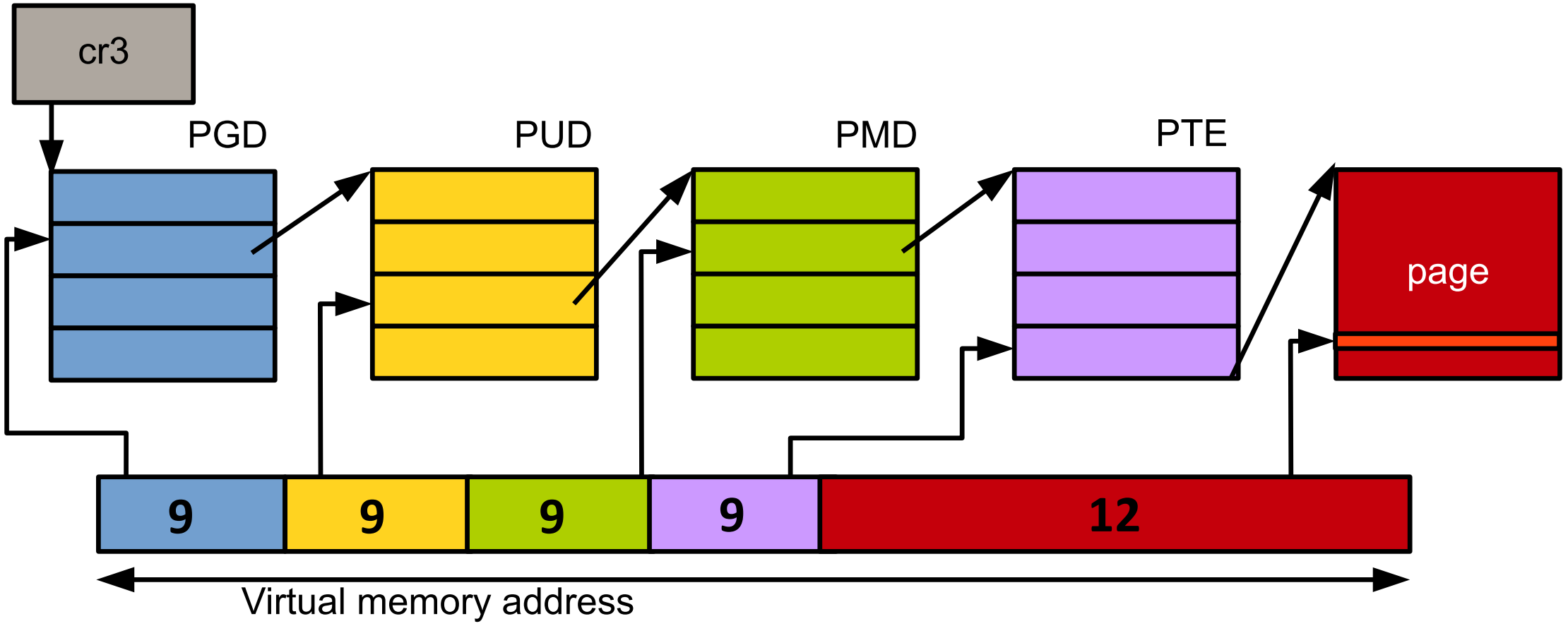
# Page table

- Linux enables paging early in the boot process
  - Page table maintains the mapping to obtain virtual to physical page
- OS sets the page table
  - MMU does the translation according to the page tables' content
- VMAs define the address space and is sparsely populated
  - One address space per process → one page table per process
  - Lots of “empty” areas

# Page tables

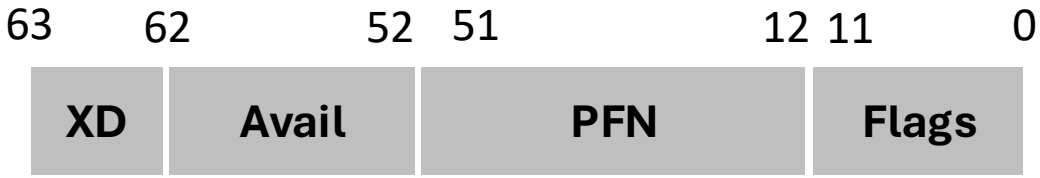

0x1234
Cr3 CPU register
0x1234
PGD:
Page
Global
Directory
PUD:
Page
Upper
Directory
PMD:
Page
Middle
Directory
PTE:
Page
Table
Entry
page

# Virtual to physical translation using MMU




**4-level page table**

# Page table entry format




Key flags

- **P (present)**: Page is in memory (bit 0)
- **R/W (Read/Write):** Writable if set (bit 1)
- **U/S (User/Supervisor):** User accessible if set (bit 2)
- **A (Accessed):** Set by CPU when page accessed (bit 5)
- **D (Dirty):** Set by CPU when page is written (bit 6)
- **XD (Execute Disable):** NX bit (bit 63)
- **PFN:** Physical frame number (bits 51-12)
- Physical address = PFN << 12

# Page table allocation: lazy is better

- Naïve approach: Allocate all 4 levels of the entire address space
  - 48-bit address space = 256 TB
  - Each PTE covers 4KB
  - Need 256TB/4KB = 64 B PTEs
  - Memory requirements:
    - 64B*8 = 512 GB per process
    - + PMD/PUD/PGD tables
- **Problem:** Most address space is unused

# Allocate on demand: allocate pages when needed

```
// During page fault:
do_page_fault(address = 0x00007FFE12345000)
{
  pgd = pgd_offset(mm, address); // PGD exists (mm->pgd)
  if (pgd_none(*pgd)) {
    pud = pud_alloc(mm, pgd, address); // ← Allocate PUD table
    if (!pud) return -ENOMEM;
  }
  pud = pud_offset(pgd, address);
  if (pud_none(*pud)) {
    pmd = pmd_alloc(mm, pud, address); // ← Allocate PMD table
    if (!pmd) return -ENOMEM;
  }
  pmd = pmd_offset(pud, address);
  if (pmd_none(*pmd)) {
    pte = pte_alloc(mm, pmd, address); // ← Allocate PTE table
    if (!pte) return -ENOMEM;
  }
  // Now we have a full path to PTE!
}
```

# Example: fresh process

- Right after fork():

```
mm->pgd → [PGD table – 512 entries]
                        ↓
                  All NULL (empty)
```

- After first page fault at 0x00007FFE12345000:

```
mm->pgd → [PGD table]
          ↓
          [entry 0] → [PUD table]
                      ↓
                      [entry 0] → [PMD table]
                                  ↓
                                  [entry 511] → [PTE table]
                                                ↓
                                                [entry 508] → Physical page!
```

Total memory used:

- 1 PGD (already exists): 4 KB
- 1 PUD table: 4KB
- 1 PMD table: 4KB
- 1 PTE table: 4KB
- Total: **16 KB vs. 512 GB**

# Viewing page table memory

- Checking page table overheads

```
# Per-process page table memory
cat /proc/PID/status | grep VmPTE
VmPTE: 348 kB  # Page tables for this process

# System-wide
cat /proc/meminfo | grep PageTables
PageTables: 45678 kB # All process page tables
```

- Typical overhead: 0.1-1% of total memory

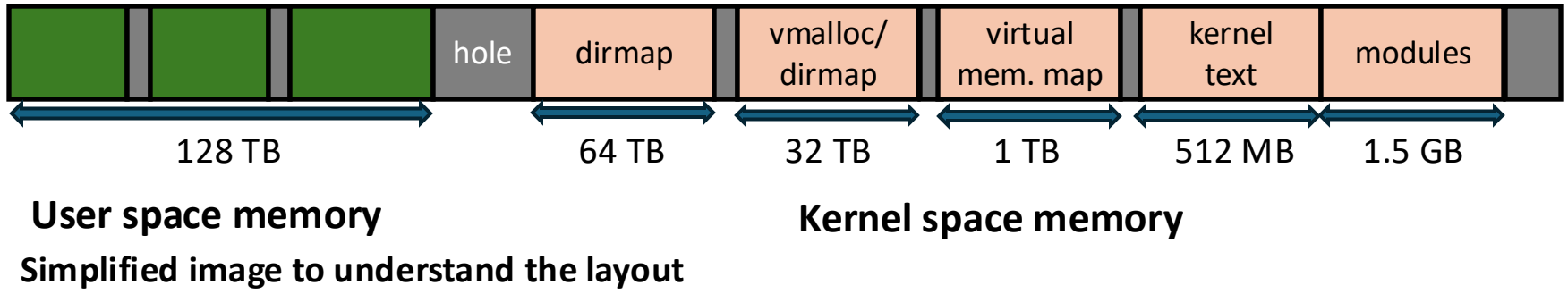
**Q. Why is this critical?**

- Sparse address spaces are cheap
- Page tables grow as VMAs are populated
- Graceful degradation under memory pressure

# Address space: range

- Memory region that a process can access
  - An illusion to access 100% of the system memory
  - Virtual memory enables accessing memory region larger than physical limit

- Defined by the process page table



# Address space access

- Memory address → index within the address space
  - Identify a specific byte
- Each process has access to **a flat 32/64-bit address space**
  - Not segmented
  - CS/DS/ES/SS bases = 0x0
    - All limits are ignored

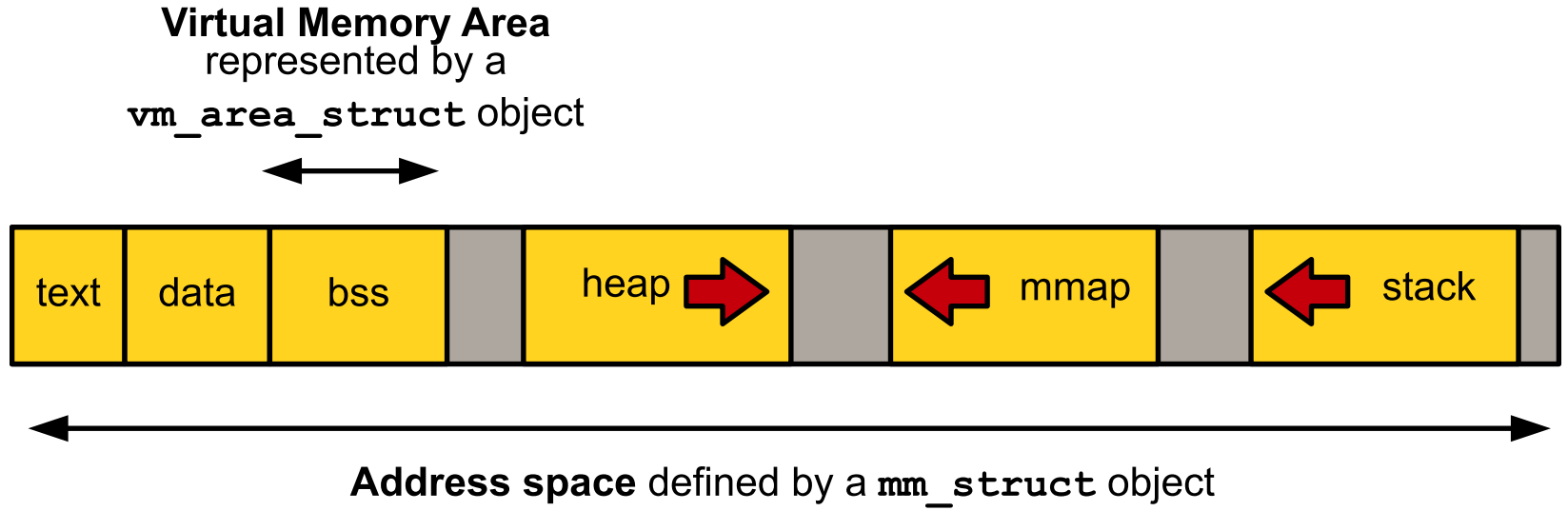
# Address space: permission

- Virtual memory area (VMA)
  - Interval of addresses that a process has the right to access
  - Dynamically add/remove to/from the address space
  - Associated permissions: **read, write, execute**
  - Illegal access → segmentation fault

```
$ cat /proc/1/maps      # or sudo pmap 1
55fe3bf02000-55fe3bff9000 r-xp 00000000 fd:00 1975429  /usr/lib/systemd/systemd
55fe3bffa000-55fe3c021000 r--p 000f7000 fd:00 1975429  /usr/lib/systemd/systemd
55fe3c021000-55fe3c022000 rw-p 0011e000 fd:00 1975429  /usr/lib/systemd/systemd
55fe3db4a000-55fe3ddfd000 rw-p 00000000 00:00 0        [heap]
7f7522769000-7f7522fd9000 rw-p 00000000 00:00 0
7f7523150000-7f7523265000 r-xp 00000000 fd:00 1979800  /usr/lib64/libm-2.25.so
```

# Address space is a collection of VMAs

- Mapping of the executable file code (**text section**)
- Mapping of the executable file initialized variables (**data section**)
- Mapping of the zero page for uninitialized variables (**bss section**)
- Mapping of the **zero page for the user-space stack**
- **Text, data, bss for each shared library** used
- Memory-mapped files, shared memory segment, anonymous mappings (malloc)



# Example of allocated address space

```
└─$ cat /proc/self/maps
5e164b673000-5e164b675000 r--p 00000000 103:03 104203765                  /usr/bin/cat
5e164b675000-5e164b67b000 r-xp 00002000 103:03 104203765                  /usr/bin/cat
5e164b67b000-5e164b67d000 r--p 00008000 103:03 104203765                  /usr/bin/cat
5e164b67d000-5e164b67e000 r--p 00009000 103:03 104203765                  /usr/bin/cat
5e164b67e000-5e164b67f000 rw-p 0000a000 103:03 104203765                  /usr/bin/cat
5e1662b55000-5e1662b76000 rw-p 00000000 00:00 0                           [heap]
74e263400000-74e263976000 r--p 00000000 103:03 104203199                  /usr/lib/locale/locale-archive
74e263a00000-74e263a28000 r--p 00000000 103:03 104206541                  /usr/lib/x86_64-linux-gnu/libc.so.6
74e263a28000-74e263bbd000 r-xp 00028000 103:03 104206541                  /usr/lib/x86_64-linux-gnu/libc.so.6
74e263bbd000-74e263c0c000 r--p 001bd000 103:03 104206541                  /usr/lib/x86_64-linux-gnu/libc.so.6
74e263c0c000-74e263c10000 r--p 0020b000 103:03 104206541                  /usr/lib/x86_64-linux-gnu/libc.so.6
74e263c10000-74e263c12000 rw-p 0020f000 103:03 104206541                  /usr/lib/x86_64-linux-gnu/libc.so.6
74e263c12000-74e263c1f000 rw-p 00000000 00:00 0
74e263d25000-74e263d6a000 rw-p 00000000 00:00 0
74e263d7d000-74e263d7f000 rw-p 00000000 00:00 0
74e263d7f000-74e263d81000 r--p 00000000 00:00 0                           [vvar]
74e263d81000-74e263d83000 r--p 00000000 00:00 0                           [vvar_vclock]
74e263d83000-74e263d85000 r-xp 00000000 00:00 0                           [vdso]
74e263d85000-74e263d86000 r--p 00000000 103:03 104206538                  /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
74e263d86000-74e263db4000 r-xp 00001000 103:03 104206538                  /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
74e263db4000-74e263dbf000 r--p 0002f000 103:03 104206538                  /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
74e263dbf000-74e263dc1000 r--p 0003a000 103:03 104206538                  /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
74e263dc1000-74e263dc2000 rw-p 0003c000 103:03 104206538                  /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
74e263dc2000-74e263dc3000 rw-p 00000000 00:00 0
7ffd93731000-7ffd93752000 rw-p 00000000 00:00 0                           [stack]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                   [vsyscall]
```

# Memory descriptor: `mm_struct`

Represents a process entire virtual address space

```c
/* include/linux/mm_types.h (Linux 6.x) */
struct mm_struct {
  struct maple_tree mm_mt; /* VMA tree */
  unsigned long mmap_base;  /* Base of mmap area */
  unsigned long task_size; /* Size of task vm space */
  pgd_t *pgd; /* Page tables */
  atomic_t mm_users;

  unsigned long  start_code; /* start addr. of code */
  unsigned long  end_code; /* end addr. of code */
  unsigned long  start_data; /* start addr. of data */
  unsigned long  end_data; /* end addr. of data */
  unsigned long  start_brk; /* start addr. of heap */
  unsigned long  end_brk; /* end addr. of heap */
  unsigned long  total_vm; /* total mapped pages */
  unsigned long  locked_vm; /* # of locked pages */

  unsigned long  flags; /* arch-specific flags*/
  atomic_t mm_count;
};
```

- **`mm_mt`**: VMA lookup and range performance (~40% space reduced)
  - RCU-friendly, better cache locality
  - Scalable data structure
- **`mm_users`**: # processes using the address space
- **`mm_count`**: reference count
  - +1 if **`mm_users`** > 0
  - +1 if the kernel uses address space
  - **`mm_struct`** freed→ **`mm_count`** = 0

# `mm_struct` life cycle: where is it tracked?

Memory descriptor (mm) is part of the **`task_struct`**

```
struct task_struct {
  /* ... */
  struct mm_struct *mm; /* Address space */
  struct mm_struct *active_mm; /* Active address space */
  /* ... */
};
```

Finding the mm that is in use:

```
// Current process's mm
struct mm_struct *mm = current->mm;
```

# `mm_struct` life cycle: allocation

## Creation (`fork()`)

```
fork()
    → copy_mm()
    → dup_mm()
        → Allocate new mm_struct (from slab cache)
        → Copy VMAs
        → Setup new page tables
        → Return new mm
```

## Sharing (threads):

```
clone(CLONE_VM)
   → Parent and child share same mm_struct
   → mm_users++
```

# `mm_struct` life cycle: deallocation

## Two-level reference counting for safe cleanup

Process exits (or last thread terminates)

↓

`do_exit()`

↓

`exit_mm()`

↓

`mmput()` – Decrement **`mm_users`**

```
{if (atomic_dec_and_test(&mm→mm_users)) __mmput(mm); }
```

↓

`__mmput()` – Final cleanup

- Tear down VMAs
- Free page tables
- Release resources

`mmdrop(mm)` – Decrement **`mm_count`**

```
{ if (atomic_dec_and_test(&mm→mm_count)) free_mm(mm) → return to slab cache
}
```

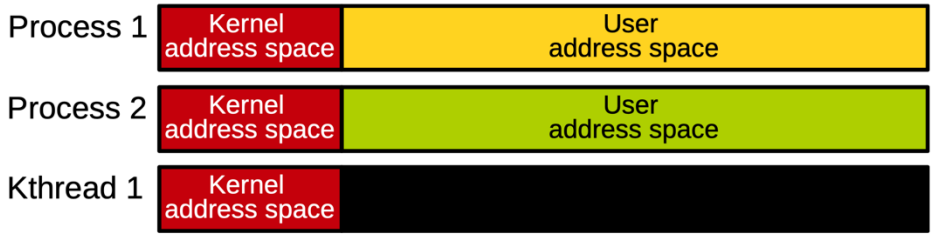
**`mm_users`** = 0: No more user processes
→ tear down the user space mm

**`mm_count`** = 0: No kernel references
→ free the structure
*Kernel may hold references even if all the processes exit (e.g., lazy TLB)*

# `mm_struct` and kernel threads

Kernel threads do not have a user-space address space

```
// Kernel thread
current->mm == NULL // No user address space
```




**But,** they still need to access the kernel address space

- ***Borrow previous process's `mm`***

```
// When kernel thread is scheduled:
if (next->mm == NULL) {
  next->active_mm = prev->mm; // Borrow it
  // Keep same page tables loaded
  // (Kernel mappings are the same anyway across all tasks!)
}
```

  - Kernel thread never accesses user addresses
  - No need to flush TLB on switch to the kernel thread

# Virtual memory area (VMA)

Contiguous range of virtual address with same permissions (per line)

```
└─$ cat /proc/self/maps
5e164b673000-5e164b675000 r--p 00000000 103:03 104203765                 /usr/bin/cat
5e164b675000-5e164b67b000 r-xp 00002000 103:03 104203765                 /usr/bin/cat
5e164b67b000-5e164b67d000 r--p 00008000 103:03 104203765                 /usr/bin/cat
5e164b67d000-5e164b67e000 r--p 00009000 103:03 104203765                 /usr/bin/cat
5e164b67e000-5e164b67f000 rw-p 0000a000 103:03 104203765                 /usr/bin/cat
5e1662b55000-5e1662b76000 rw-p 00000000 00:00 0                          [heap]
74e263400000-74e263976000 r--p 00000000 103:03 104203199                 /usr/lib/locale/locale-archive
74e263a00000-74e263a28000 r--p 00000000 103:03 104206541                 /usr/lib/x86_64-linux-gnu/libc.so.6
74e263a28000-74e263bbd000 r-xp 00028000 103:03 104206541                 /usr/lib/x86_64-linux-gnu/libc.so.6
74e263bbd000-74e263c0c000 r--p 001bd000 103:03 104206541                 /usr/lib/x86_64-linux-gnu/libc.so.6
74e263c0c000-74e263c10000 r--p 0020b000 103:03 104206541                 /usr/lib/x86_64-linux-gnu/libc.so.6
74e263c10000-74e263c12000 rw-p 0020f000 103:03 104206541                 /usr/lib/x86_64-linux-gnu/libc.so.6
74e263c12000-74e263c1f000 rw-p 00000000 00:00 0
74e263d25000-74e263d6a000 rw-p 00000000 00:00 0
74e263d7d000-74e263d7f000 rw-p 00000000 00:00 0
74e263d7f000-74e263d81000 r--p 00000000 00:00 0                          [vvar]
74e263d81000-74e263d83000 r--p 00000000 00:00 0                          [vvar_vclock]
```

```
# r = read
# w = write
# x = execute
# s = shared
# p = private (copy on write)
```

Each VMA can be:

- Anonymous (heap, stock) or file-backed
- Private (COW) or shared (IPC)
- Different permissions

# VMA representation: `vm_area_struct`

```
/* include/linux/mm_types.h */
struct vm_area_struct {
  unsigned long vm_start; /* Start address */
  unsigned long vm_end; /* End address (exclusive) */

  struct mm_struct *vm_mm; /* Parent mm */

  pgprot_t vm_page_prot; /* Access permissions */
  unsigned long vm_flags; /* Flags */

  struct file *vm_file; /* Mapped file (or NULL) */
  unsigned long vm_pgoff; /* Offset in file (pages) */

  const struct vm_operations_struct *vm_ops; /* Operations */
  void *vm_private_data; /* Private data (driver specific) */
};
```

- size: **`[vm_start - vm_end)`**
- Address space pointed by vm_mm
- Each VMA is unique to **`mm_struct`**:
  - 2 process mapping the same file will have 2 diff **`mm_struct`** and two different **`vm_area_struct`**
  - 2 threads sharing an **`mm_struct`** object also share the same **`vm_area_struct`** objects

# VMA permissions and flags

- Basic permission

```
VM_READ // Readable
VM_WRITE // Writable
VM_EXEC // Executable
VM_SHARED // Shared (vs. private COW)
```

- Special flags

```
VM_GROWSDOWN // Stack (grows toward lower addresses)
VM_GROWSUP // Heap (uncommon)
VM_LOCKED // Pages pinned in memory (mlock)
VM_IO // Device I/O memory
VM_DONTCOPY // Don't copy on fork()
VM_DONTEXPAND // Cannot grow via mremap()
VM_SEQ_READ // Sequential access pattern
VM_RAND_READ // Random access pattern
```

# VMA types

- Anonymous VMA (heap, stack)

```
vm_file = NULL
vm_flags = VM_READ | VM_EXEC // code
vm_flags = VM_READ | VM_WRITE // stack
```

- No backing file
- Pages allocated on demand
- Initialized to zero

- File-backed VMA (executable, libraries)

```
vm_file = filename
vm_pgoff = <some value>;
```

- Reads from the disk on page fault
- Can be shared between processes

- Shared VMA (IPC)

```
vm_flags = VM_SHARED
```

- Changes visible to other processes

- System-call enabled (madvise)

```
vm_flags = VM_SEQ_READ
OR
vm_flags = VM_RAND_READ
```

- Used mostly for prefetching file data
- Instructs pre-fetching algorithm to increase or decrease its prefetch window

# VMA operations

```
struct vm_operations_struct {
  void (*open)(struct vm_area_struct *vma);
  void (*close)(struct vm_area_struct *vma);

  // Page fault handler
  vm_fault_t (*fault)(struct vm_fault *vmf);

  // Make page writable (COW)
  vm_fault_t (*page_mkwrite)(struct vm_fault *vmf);
  /* ... */
};
```

- Examples:
  - Anonymous VMA: **`fault()`** allocates zero page
  - File VMA: **`fault()`** reads from disk
  - Device memory: **`fault()`** maps hardware registers

# VMA lookup scalability

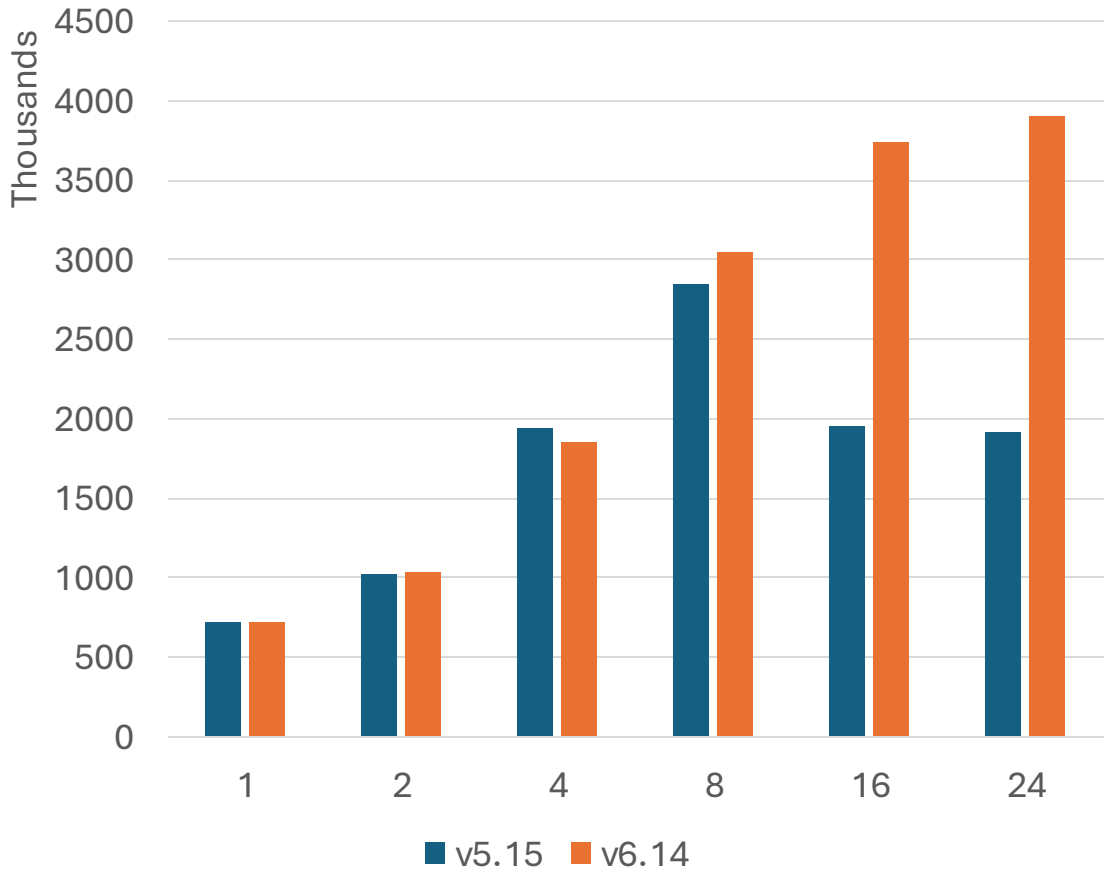
- The scalability problem:
  - **`find_vma()`** is on the critical path of every page fault

```
do_page_fault()
    ↓
vma = find_vma(mm, addr); // ← Must be fast!
   ↓
check permissions
   ↓
handle_mm_fault()
```

- Modern applications: 10K+ VMAs (JVM, databases)
- High-page fault rates: 100K+ faults/sec
- Multicore scalability: many threads fault simultaneously

# VMA lookup scalability comparison

- **Linux ≤6.0: Linked list + red-black tree**
  - O(log n) lookup in rb-tree
  - BUT: mmap_sem (reader-writer lock) contention
- **Linux ≥6.1: Maple tree**
  - Integrated RCU-friendly design
  - Single structure, better locality
  - Per-VMA locking (6.x)



# VMA lookup implementation (simplified for context)

```c
/* Find VMA containing address */
struct vm_area_struct *find_vma(struct mm_struct *mm, unsigned long addr) {
  struct ma_state mas;
  struct vm_area_struct *vma;

  /* Initialize maple tree state */
  mas_init(&mas, &mm->mm_mt, addr); /* Walk maple tree (RCU-friendly) */
  rcu_read_lock();
  vma = mas_walk(&mas);
  rcu_read_unlock();

  if (!vma || vma->vm_start > addr)
    return NULL;
  return vma;
}
```

**Still O(log n), but:**

- RCU read-side is lockless
- Better cache behavior
- Per-VMA locking possible

# VMA manipulation API: finding a VMA

```c
/* Find VMA containing address */
struct vm_area_struct *find_vma(struct mm_struct *mm,
                                unsigned long addr);

/* Find VMA intersection with range */
struct vm_area_struct *find_vma_intersection(
              struct mm_struct *mm,
              unsigned long start_addr,
              unsigned long end_addr);

/* Iterate over VMAs in range */
VMA_ITERATOR(vmi, mm, start);
struct vm_area_struct *vma;

for_each_vma(vmi, vma) {
  /* Process each VMA */
}
```

# VMA manipulation API: VMA lookup in page fault

```
// Page fault at address 0x7ffe12345000
do_page_fault() {
  unsigned long address = read_cr2();

  // Faulting address
  struct mm_struct *mm = current->mm;
  struct vm_area_struct *vma;

  // Find VMA containing this address vma = find_vma(mm, address);
  if (!vma || vma->vm_start > address)
    goto bad_area; // SIGSEGV!

  // Check permissions
  if (error_code & PF_WRITE && !(vma->vm_flags & VM_WRITE))
    goto bad_area; // SIGSEGV!

  // Handle the fault
  handle_mm_fault(vma, address, flags);
}
```

# VMA manipulation API: create VMA (`do_mmap()`)

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

- **`file`**: File to map (or NULL for anonymous)
- **`addr`**: Requested address (or 0 for kernel choice)
- **`len`**: Length in bytes
- **`prot`**: Protection (PROT_READ, PROT_WRITE, PROT_EXEC)
- **`flags`**: MAP_PRIVATE, MAP_SHARED, MAP_ANONYMOUS, etc.

- Returns the starting address of new mapping

# VMA manipulation API: Protection flags

| Flag (user exposed) | Effect | VM flag |
|---|---|---|
| **`PROT_READ`** | Readable | **`VM_READ`** |
| **`PROT_WRITE`** | Writable | **`VM_WRITE`** |
| **`PROT_EXEC`** | Executable | **`VM_WRITE`** |
| **`PROT_NONE`** | No access | (none) |

Common combinations:

- **`.text`**: **`PROT_READ | PROT_EXEC`**
- **`.data`**: **`PROT_READ | PROT_WRITE`**
- Stack: **`PROT_READ | PROT_WRITE`**

# VMA manipulation API: Mapping flags

| Flag | Effect |
|---|---|
| **MAP_PRIVATE** | Private COW mapping |
| **MAP_SHARED** | Shared mapping (IPC) |
| **MAP_ANONYMOUS** | No file backing |
| **MAP_FIXED** | Use exact address |
| **MAP_GROWSDOWN** | Stack semantics |
| **MAP_LOCKED** | Locked pages in memory |
| **MAP_POPULATE** | Prefault pages immediately |
| **MAP_HUGETLB** | Use huge pages |

Must specify either **MAP_PRIVATE** or **MAP_SHARED**

# Allocating memory region: `mmap()` system call

- **User space interface**

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

- **Example**

```
// Anonymous mapping (heap-like)
void *p = mmap(NULL, 4096, PROT_READ | PROT_WRITE,
                           MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

// File mapping (shared library)
void *p = mmap(NULL, size, PROT_READ | PROT_EXEC, MAP_PRIVATE, fd, 0);

// Shared memory (IPC)
void *p = mmap(NULL, size, PROT_READ | PROT_WRITE,
                           MAP_SHARED | MAP_ANONYMOUS, -1, 0);
```

# VMA merging

- **Kernel tries to merge adjacent VMAs with the same permission**

```
Before: [VMA1: 0x1000-0x2000, rw-] [VMA2: 0x2000-0x3000, rw-]
After: [VMA: 0x1000-0x3000, rw-] ← Single VMA!
```

- Benefits:
  - Fewer VMA objects
  - Faster lookup
  - Less memory overhead
- Can merge if:
  - Adjacent addresses
  - Same permission
  - Same file/offset (or both anonymous)
  - Same flags

# Removing VMAs: `do_munmap()`

```
/* mm/mmap.c */
int do_munmap(struct mm_struct *mm, unsigned long start, size_t len)
```

System call:

```
int munmap(void *addr, size_t len);
```

- What does it do?
  1. Find VMAs in range [start,  start + len)
  2. Unmap pages (free page tables)
  3. Delete VMA objects from maple tree
  4. Merge adjacent VMAs if possible

# VMA splitting

- munmap() may split a VMA:

```
Original VMA: [VMA: 0x1000-0x4000, rw-]

munmap(0x2000, 0x1000):

[VMA1: 0x1000-0x2000, rw-] [VMA2: 0x3000-0x4000, rw-]
                          ↑ hole ↑
```

Kernel must:

1. Allocate new vm_area_struct
2. Adjust address ranges
3. Update maple tree

# Other VMA operations

```c
/* Change protection */
int mprotect(void *addr, size_t len, int prot);

/* Expand/shrink mapping */
void *mremap(void *old_addr, size_t old_size, size_t new_size, int flags);

/* Lock pages in memory */
int mlock(void *addr, size_t len);
int munlock(void *addr, size_t len);

/* Give advice about usage */
int madvise(void *addr, size_t len, int advice);
```

# Putting it all together

# Page fault flow

User program accesses address 0x7FFE12345000

↓

MMU attempts translation

↓

PTE not present → Page fault exception

↓

`do_page_fault()`

↓

`find_vma(mm, 0x7FFE12345000)` ← Search maple tree

↓

Found VMA [0x7FFE12340000-0x7FFE12350000, rw-]

↓

Check permissions (write fault, VMA is writable → OK)

↓

`handle_mm_fault()`

↓

Allocate page tables (if needed - lazy allocation!)

↓

Allocate physical page (alloc_pages - from local NUMA node!)

↓

Install PTE (update page tables)

↓

Return to user (instruction retries successfully)

# Copy-on-write (COW)

- **`fork()`** shares memory initially

```
fork()
  → Copy mm_struct
  → Copy VMAs (maple tree copy)
  → Copy page tables
  → Mark all pages read-only (in both parent and child)
  → Increment page refcounts
```

- On first write

```
Write to COW page
  → Page fault (write to read-only page)
  → Check: refcount > 1?
  → Yes: Allocate new page, copy contents
  → Install new page with write permission
  → Decrement refcount on old page
```

Fast **`fork`**, saves memory

# Example: process startup

```
execve("/bin/ls", ...)
  ↓
// Kernel creates new mm_struct with maple tree
mm = mm_alloc();
mt_init(&mm->mm_mt);

// Create VMAs for executable segments
do_mmap(exe_file, 0x400000, text_size, PROT_READ|PROT_EXEC,
                            MAP_PRIVATE, 0); // .text
do_mmap(exe_file, 0x600000, data_size, PROT_READ|PROT_WRITE,
                            MAP_PRIVATE, offset); // .data

// Create anonymous VMA for heap
do_mmap(NULL, heap_start, 0, PROT_READ|PROT_WRITE,
                            MAP_PRIVATE|MAP_ANONYMOUS, 0); // Create stack VMA
do_mmap(NULL, stack_start, stack_size, PROT_READ|PROT_WRITE,
                            MAP_PRIVATE|MAP_GROWSDOWN, 0);

// Load page tables into CR3
switch_mm(mm);
```

# Security: enforcing permissions

- Hardware enforces page permissions

```
User tries: ((char*)0xFFFFFFFF80000000) = 0; // Kernel address
MMU checks PTE U/S bit → SIGSEGV!
```

- Cannot bypass
  - User/supervisor bit checked on every memory access
  - NX (no execute) bit prevents code execution on data pages
  - Page tables themselves are protected
- **KPTI** (kernel page table isolation):
  - Separate page tables for user/kernel mode
  - Mitigates meltdown/Spectre attacks
  - ~5-10% performance cost

# Further reading

- Supporting bigger and heterogeneous memory efficiently
  - AutoNUMA, Transparent Hugepage Support, Five-level page tables
  - Heterogeneous memory management
- Optimization for virtualization
  - Kernel same-page merging (KSM)
  - MMU notifier
- Linux kernel virtual memory map
- Kernel page table isolation
- Addressing Meltdown and Spectre in the kernel
- Meltdown and Spectre