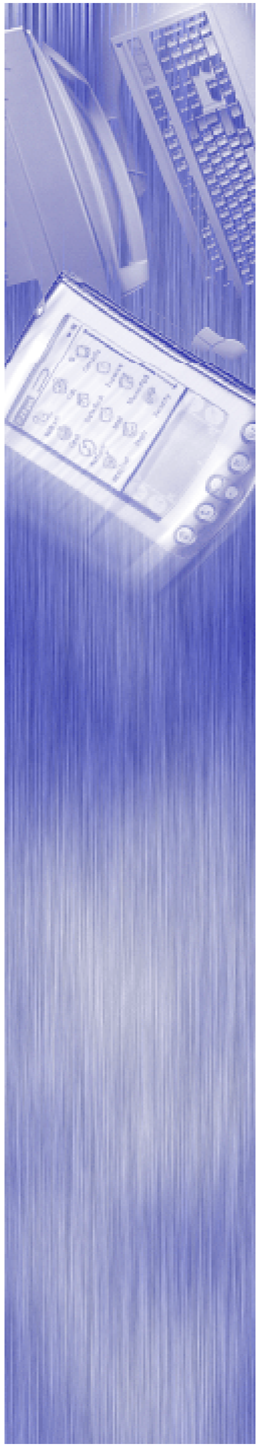


CS632/SEP564: Embedded Operating Systems (Fall 2008)

Memory Management

KAIST

Memory Management

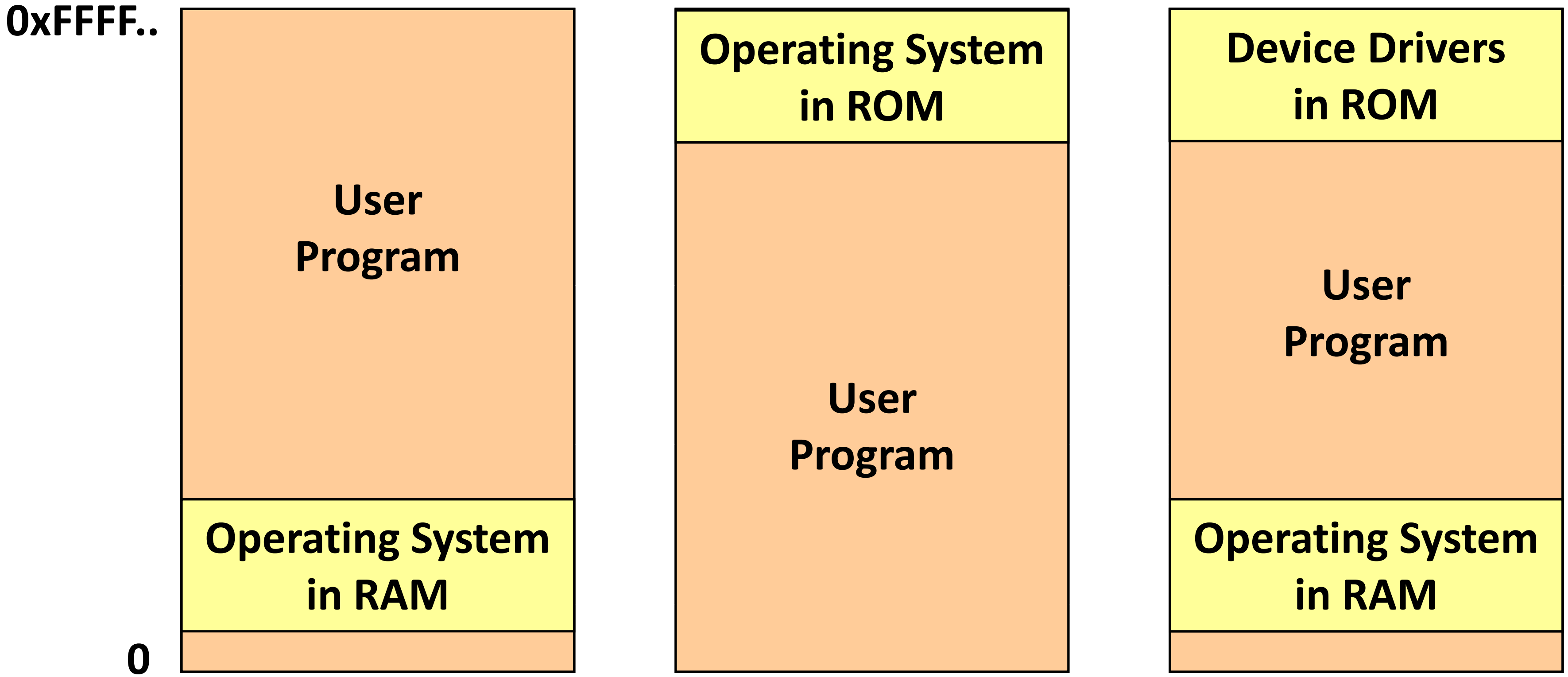
- **Goals**
  - To provide a convenient abstraction for programming
  - To allocate scarce memory resources among competing processes to maximize performance with minimal overhead
  - To provide isolation between processes
- **Why is it so difficult?**

Hardware Support

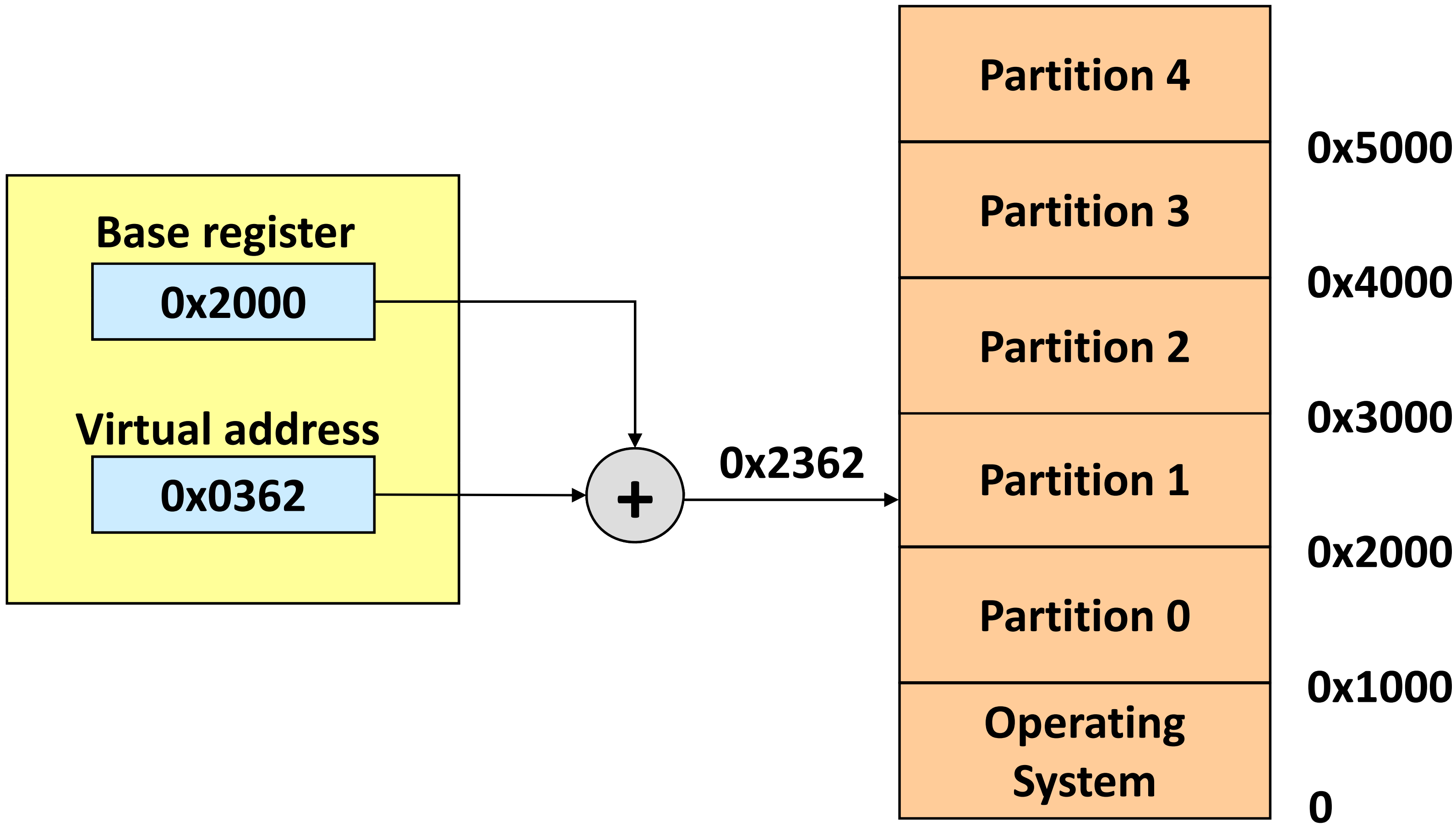
- **None**
  - CPU directly accesses physical memory
- **Memory Protection Unit (MPU)**
  - CPU directly accesses physical memory with memory protection
- **Memory Management Unit (MMU)**
  - Fully implements virtual memory
  - Linux runs with MMU-enabled CPUs

Single/Batch Programming

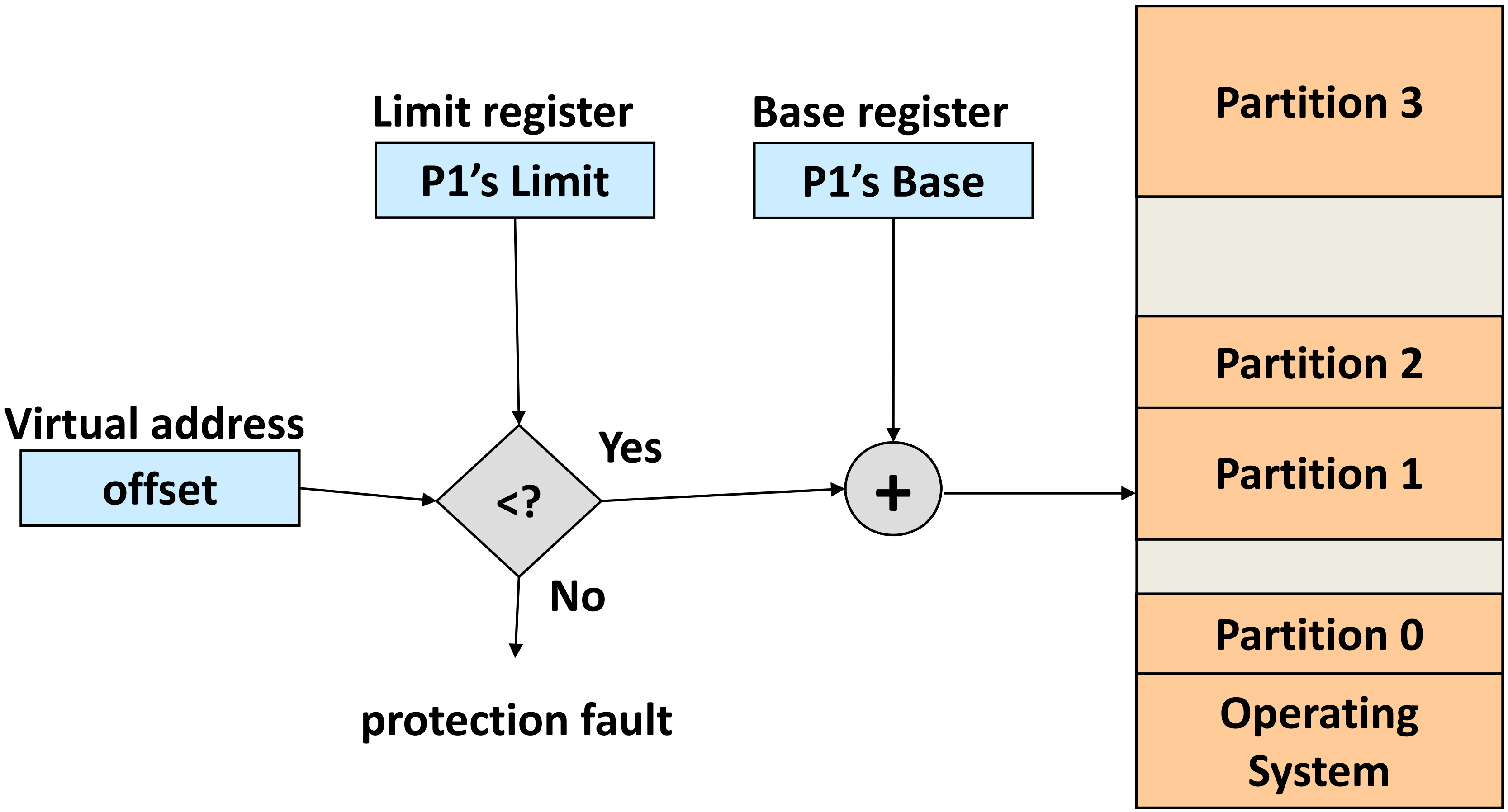
- **An OS with one user process**
  - Programs use physical addresses directly.
  - OS loads job, runs it, unloads it.



Fixed Partitions



Variable Partitions

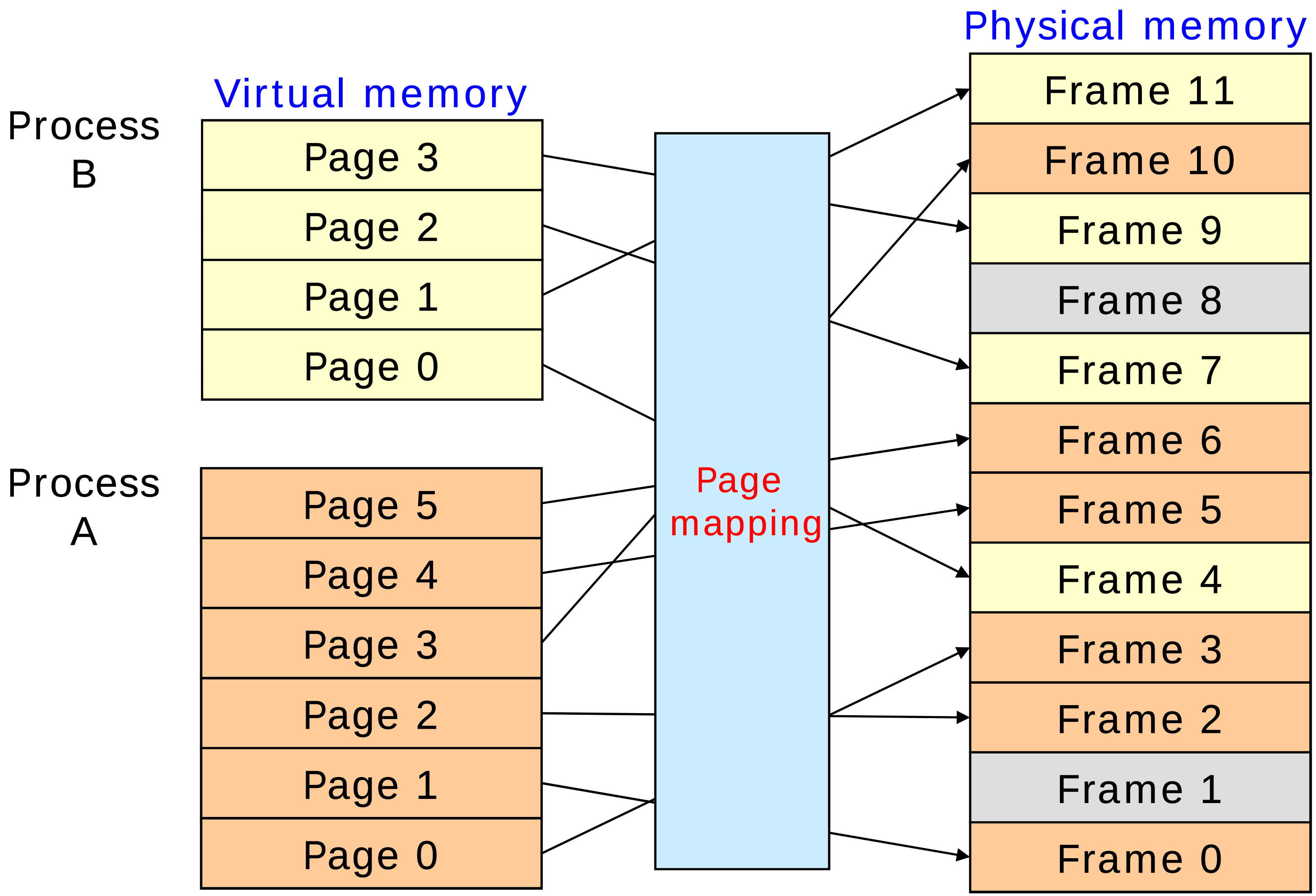


Virtual Memory

- **Why?**
  1.
  2.
  3.

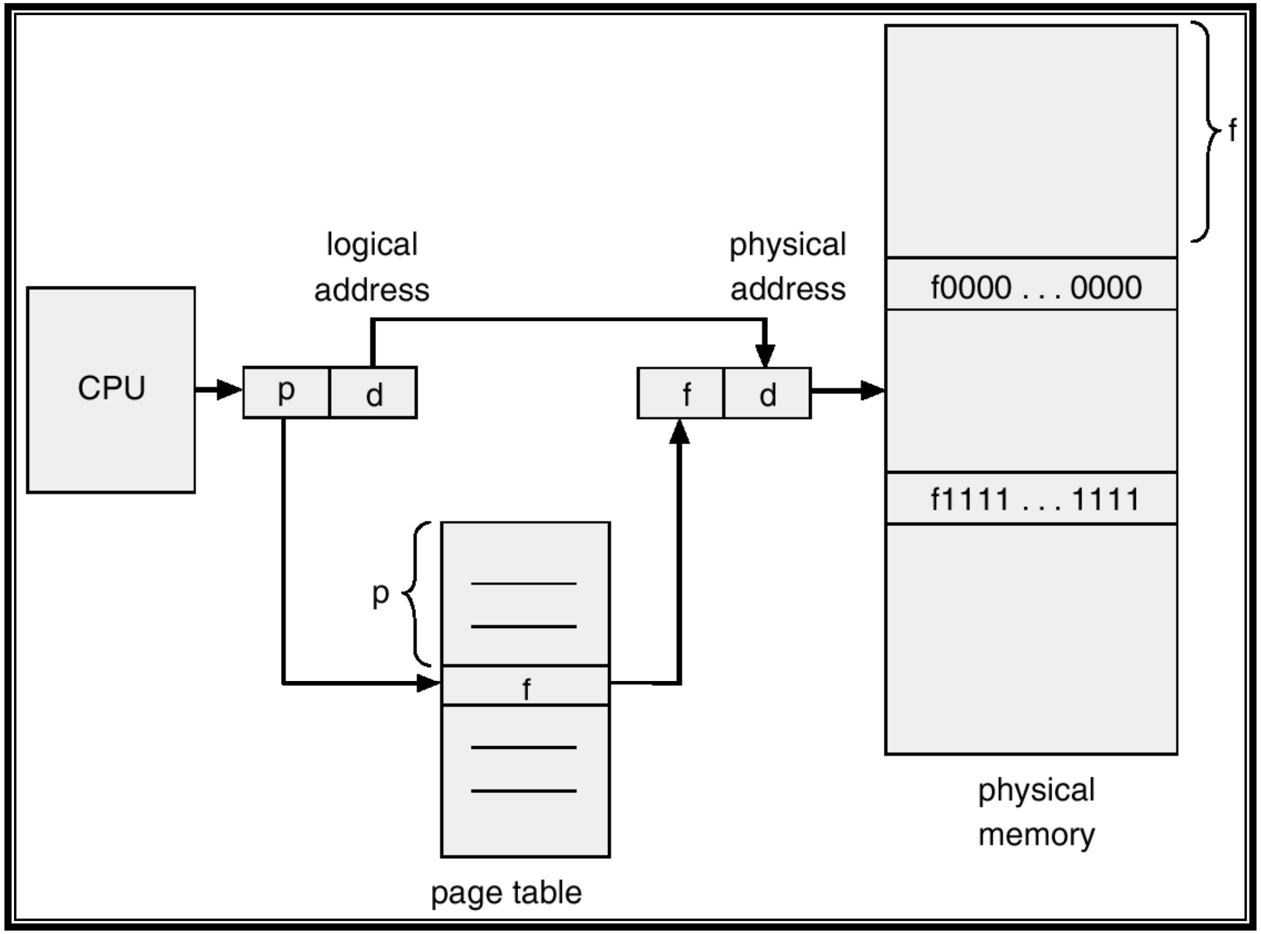
- **How?**
  - MMU (Memory Management Unit)
  - Address translation
  - Demand paging
  - Page tables

Paging (1)


Physical memory
Virtual memory
Process B
Page 3
Page 2
Page 1
Page 0
Process A
Page 5
Page 4
Page 3
Page 2
Page 1
Page 0
Page mapping
Frame 11
Frame 10
Frame 9
Frame 8
Frame 7
Frame 6
Frame 5
Frame 4
Frame 3
Frame 2
Frame 1
Frame 0

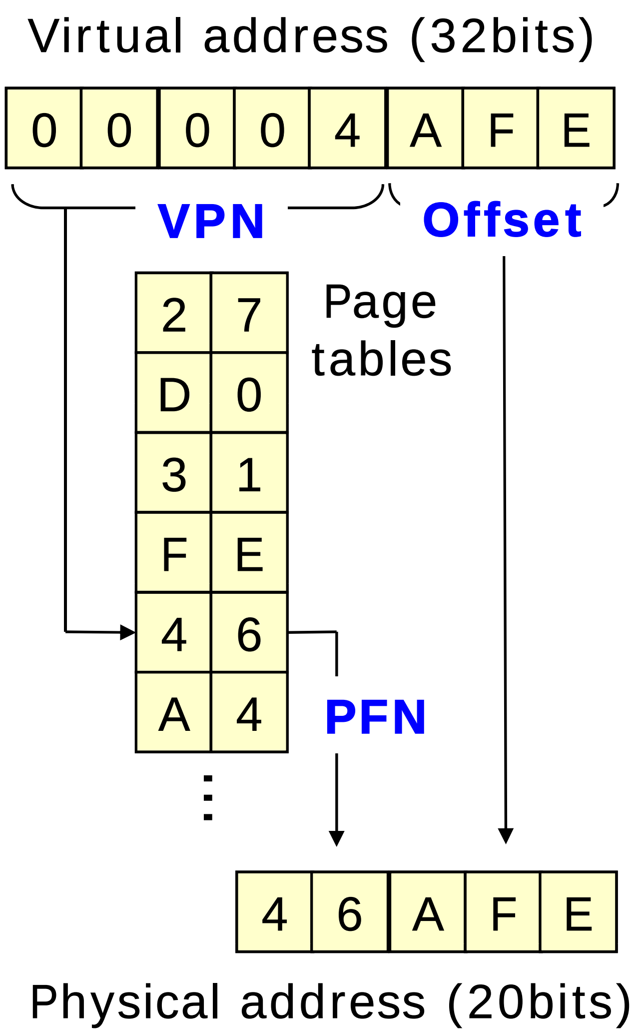
Paging (2)

- **Address translation architecture**



Paging (3)

- **Paging example**
  - Virtual address: 32 bits
  - Physical address: 20 bits
  - Page size: 4KB
  - Offset: 12 bits
  - VPN: 20 bits
  - Page table entries: 2^{20}



Paging (4)

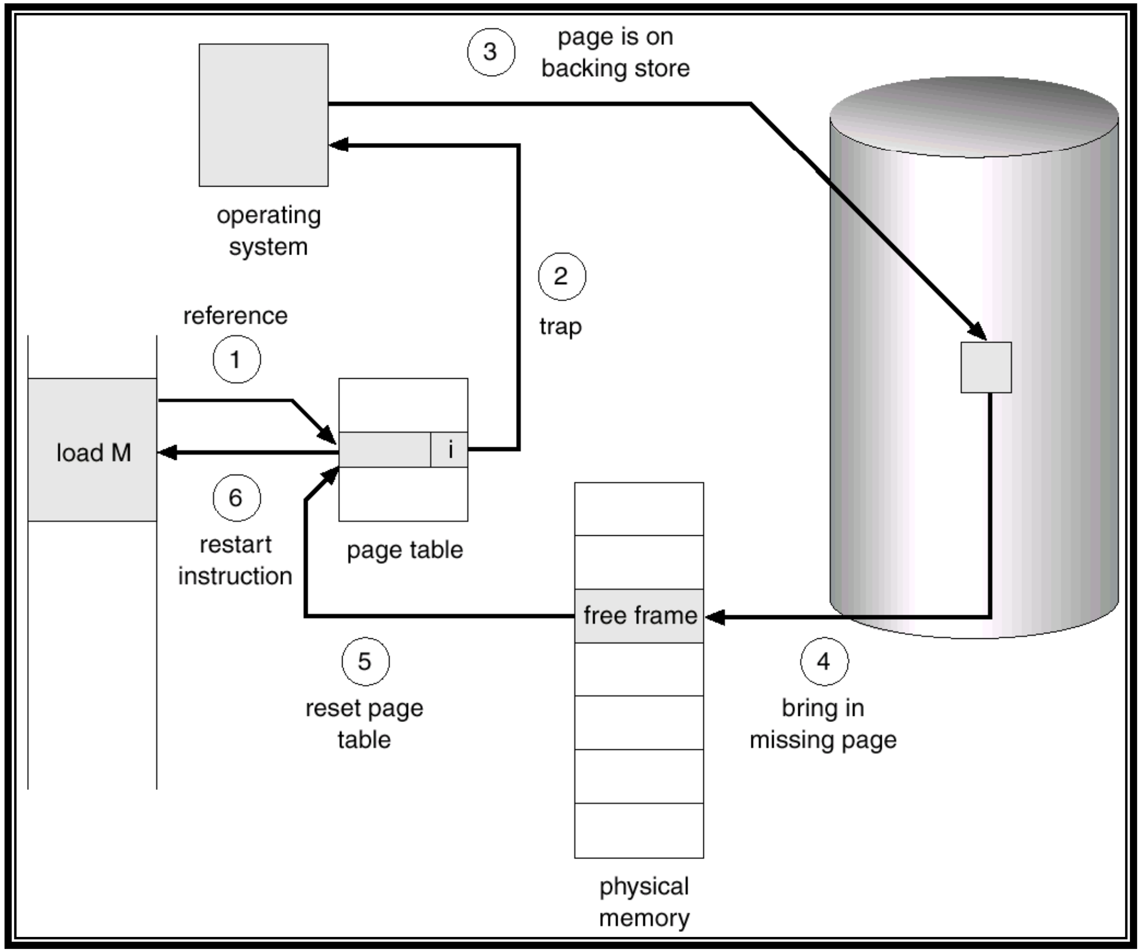
- **Page Table Entries (PTEs)**

1	1	1	2	20
V	R	M	Prot	Page Frame Number (PFN)

- Valid bit (V) says whether or not the PTE can be used.
  – It is checked each time a virtual address is used.
- Reference bit (R) says whether the page has been accessed.
  – It is set when a read or write to the page occurs.
- Modify bit (M) says whether or not the page is dirty.
  – It is set when a write to the page occurs.
- Protection bits (Prot) control which operations are allowed on the page.
  – Read, Write, Execute, etc.
- Page frame number (PFN) determines physical page.

Paging (5)

- **Handling a page fault**

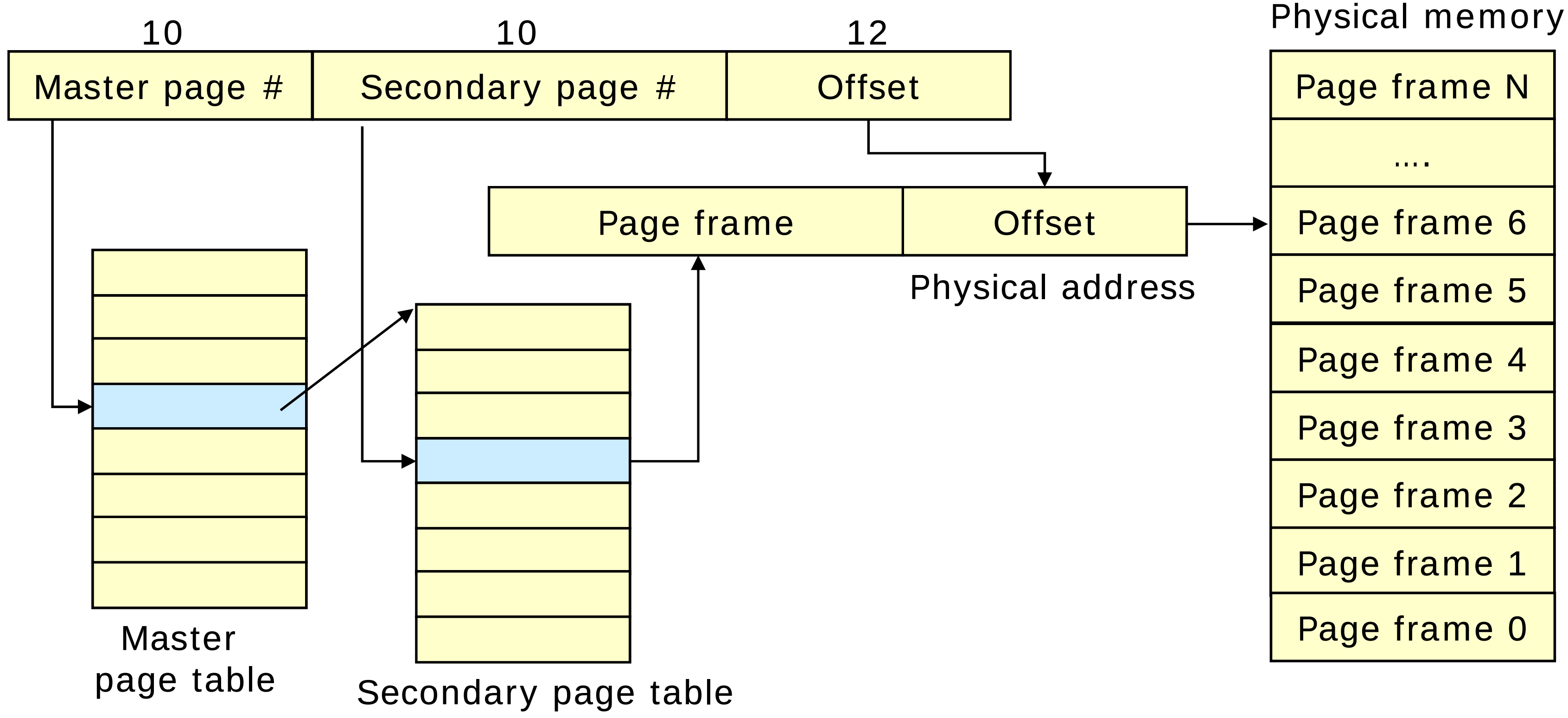


Problems

- **Space overhead**
  - Page tables
- **Time overhead**
  - Address translation

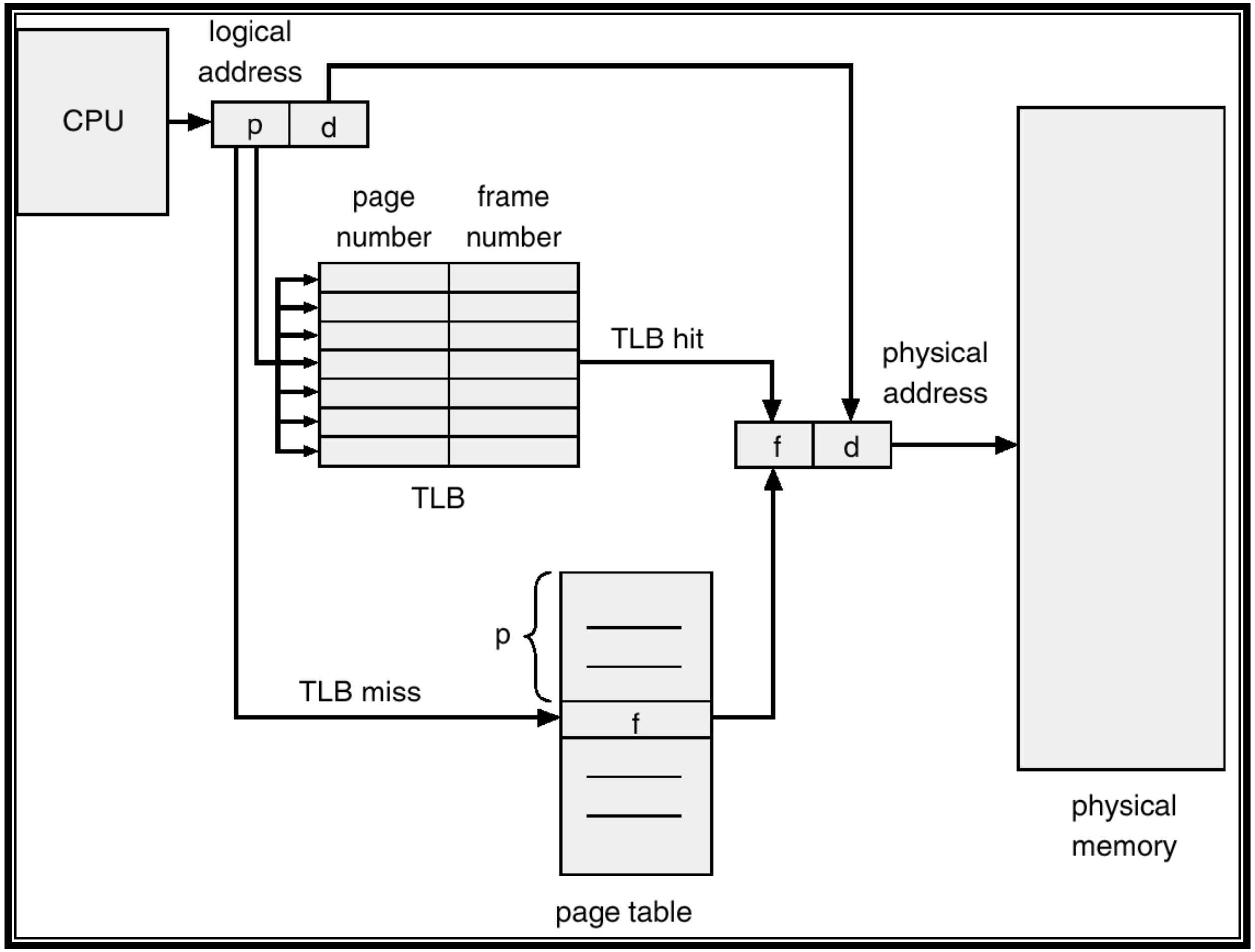
Multi-level Page Tables

- **Example: Two-level Page Tables**
  - 32-bit address space, 4KB pages, 4bytes/PTE
  - Want master page table in one page



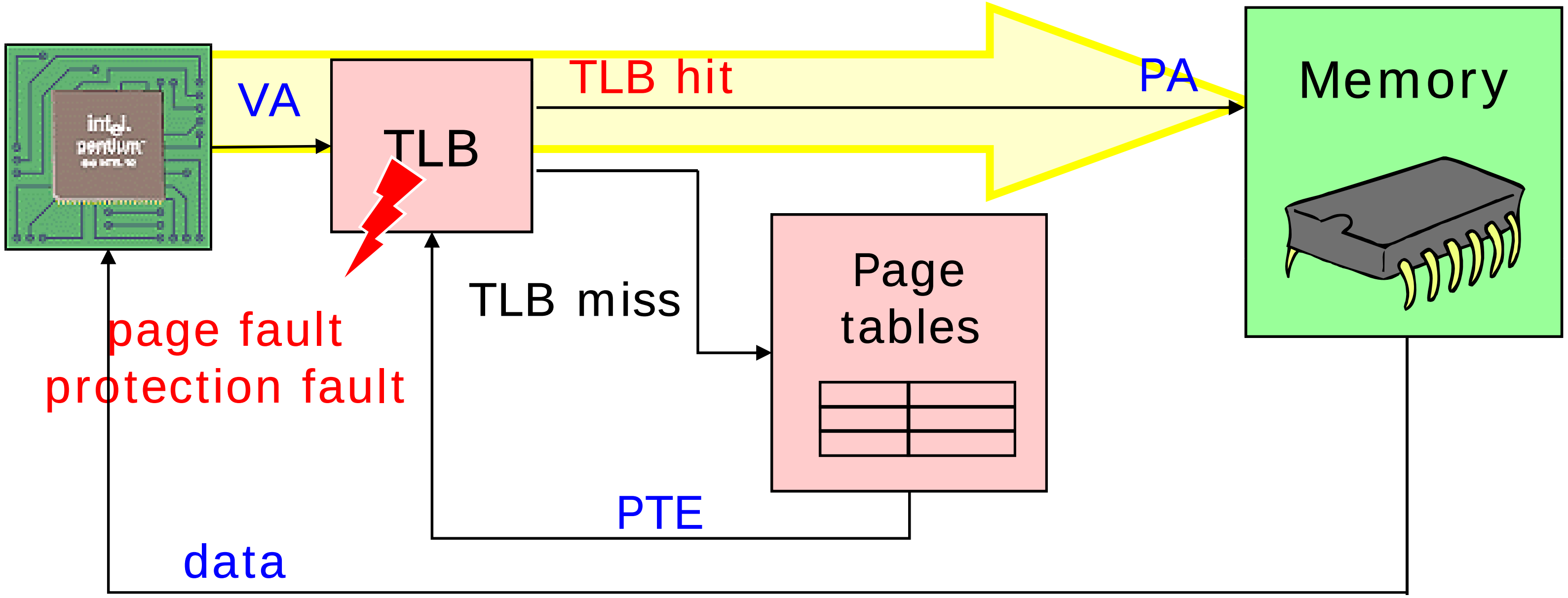
TLBs

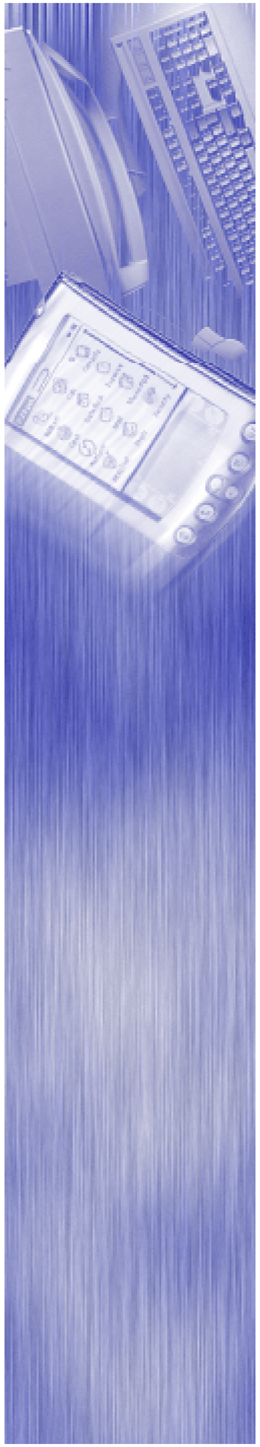
- **Address translation with TLB**



Memory Reference

- **Situation**
  - Process is executing on the CPU, and it issues a read to a (virtual) address.

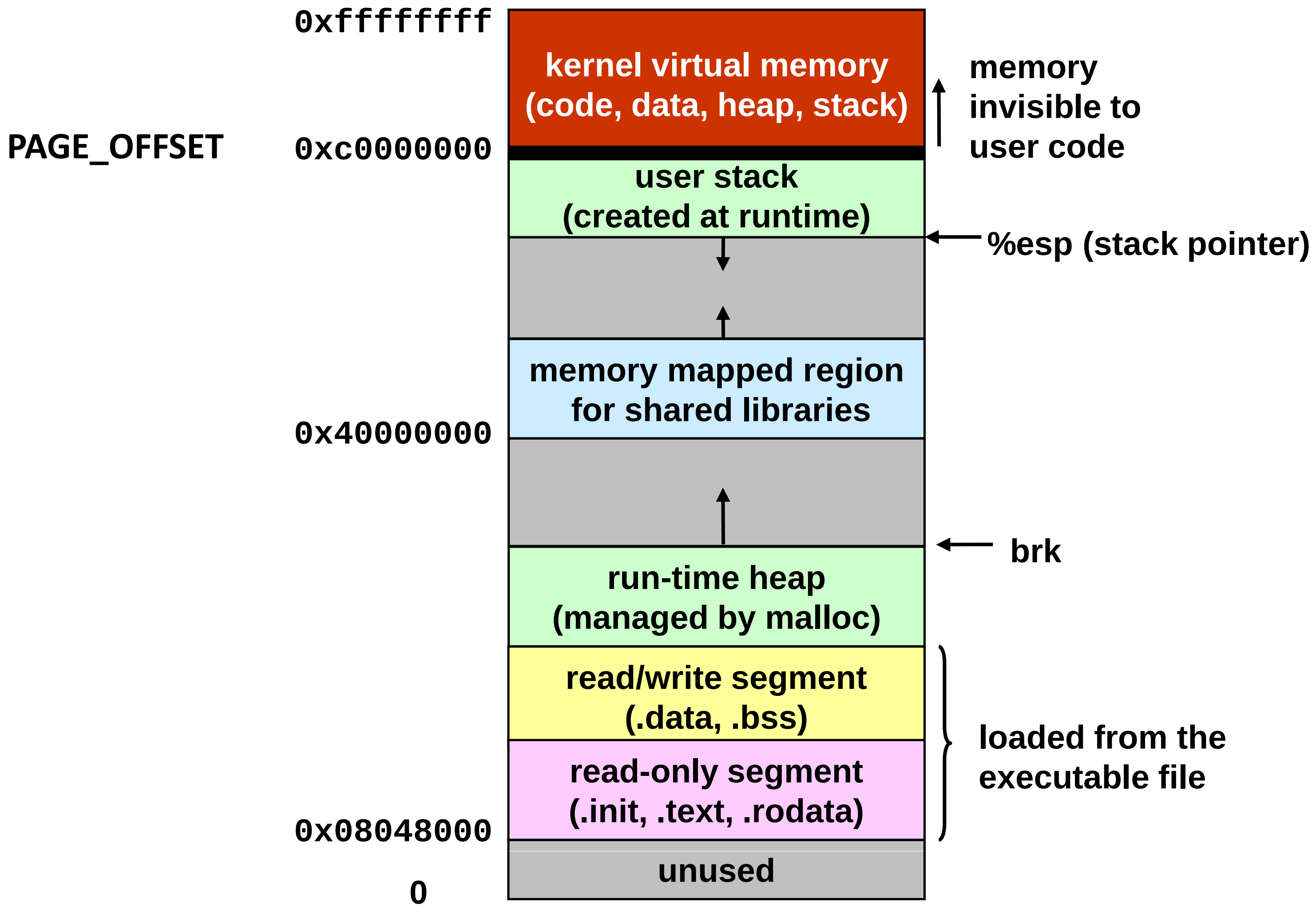




CS632/SEP564: Embedded Operating Systems (Fall 2008)

Virtual Memory Implementation

Process Address Space (1)

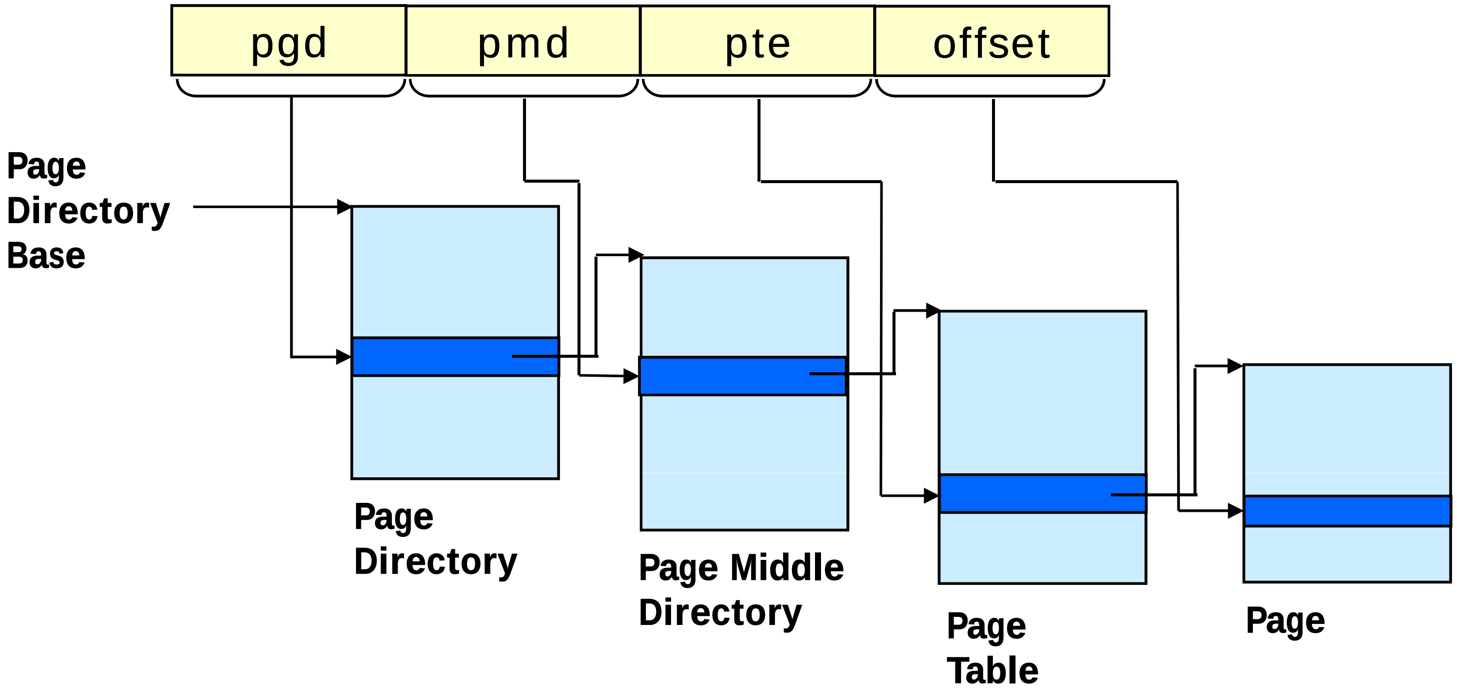


Process Address Space (2)

- **Memory areas**
  - The intervals of legal addresses in process address space.
  - Text section: code
  - Data section: initialized global variables
  - BSS section: uninitialized global variables
  - User-space stack
  - Heap
  - Shared libraries (text, data, bss for each shlib)
  - Memory mapped files
  - Shared memory segments

Process Address Space (3)

- **Paging: three-level address translation**
  - In i386, the size of Page Middle Directory (PMD) is 1, if the physical address extension (PAE) flag is disabled.



Virtual Memory (1)

- **Demand paging**
  - Physical memory ↔ File system
  - Pages are backed by files
    - Program code
    - (Initial) program data
    - Memory-mapped files, ...
- **Swapping**
  - Physical memory ↔ Swap area
  - Anonymous pages
    - Stack, heap, BSS
    - (Written) program data
    - Shared memory, mmap() with MAP_ANON, ...

Virtual Memory (2)

- **Page cache**
  - A cache of physical pages
  - The page cache holds
    - Pages containing data of regular files
    - Pages containing directories
    - Pages containing data directly read from block device files
    - Pages containing data of user mode processes that have been swapped out on disk
    - Pages belonging to files of special filesystems (e.g., shm)
  - Each page included in the page cache contains data belonging to some file.

Virtual Memory (3)

- **Page fault**
  - Page fault mainly occurs due to
    - Not-present pages
    - Protection violation (especially for copy-on-write)
  - Major page fault
    - If the kernel need to access the disk to make the page available.
  - Minor page fault
    - If the kernel only need to allocate pages in RAM without reading anything from disk.

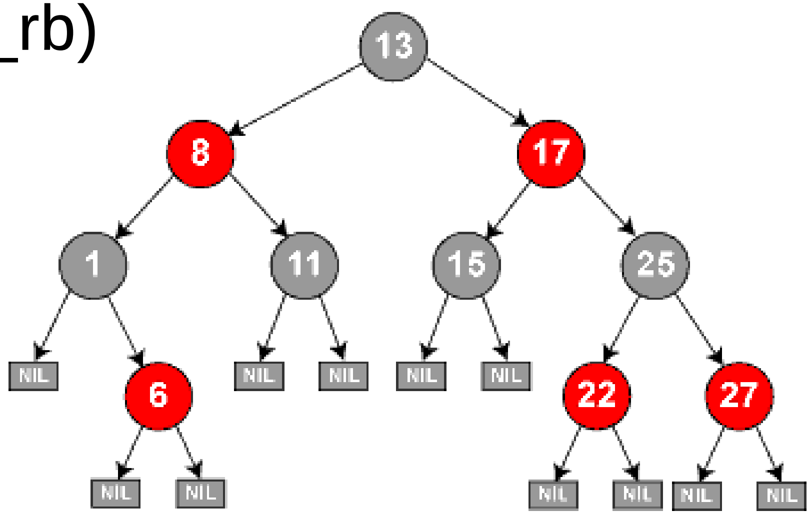
VMA (1)

- **struct vm_area_struct** <linux/mm.h>
  - Nonoverlapping regions, each representing a continuous, page-aligned subset of the virtual address space.

struct mm_struct *	vm_mm;	associated mm_struct
unsigned long	vm_start;	VMA start, inclusive
unsigned long	vm_end;	VMA end, exclusive
struct vm_area_struct *	vm_next;	list of VMA's
pgprot_t	vm_page_prot;	access permissions
unsigned long	vm_flags;	VMA flags
struct rb_node	vm_rb;	VMA's node in the tree
struct vm_operations_struct *	vm_ops;	associated ops
struct file *	vm_file;	mapped file, if any.
unsigned long	vm_pgoff;	offset within the file
...		

VMA (2)

- **Organization of VMAs**
  - Linked list (via mm→mmap)
    - – Simple and efficient for traversing of all elements
    - – Sorted by ascended address (linked via vma→vm_next)
  - Red-black tree (via mm→mm_rb)
    - – A type of balanced binary tree.
      - » The root & all leaves are black.
      - » Both children of every red node are black.
      - » All paths from any given node to its leaf nodes contain the same number of black nodes.
    - – Searching, insertion, deletion: O(log(n))
    - – Used when locating a specific VMA in the address space.



VMA (3)

- **VMA flags**

Flag	Description
VM_READ / VM_WRITE / VM_EXEC	Pages can be read from / written to / executed.
VM_SHARED	Pages are shared.
VM_MAYREAD / VM_MAYWRITE / VM_MAYEXEC / VM_MAYSHARE	VM_READ / VM_WRITE / VM_EXEC / VM_SHARE flag can be set.
VM_GROWSDOWN / VM_GROWSUP	The area can grow downward / upward.
VM_SHM	The area is used for shared memory
VM_DENYWRITE / VM_EXECUTABLE	The area maps an unwritable file / an executable file
VM_LOCKED	The pages in this area are locked.
VM_IO	The area maps a device's I/O space.
VM_RESERVED	This area must not be swapped out.
VM_SEQ_READ / VM_RAND_READ	The pages seem to be accessed sequentially / randomly.

VMA (4)

VMA operations

- struct vm_operations_struct <linux/mm.h>
- void open(struct vm_area_struct *area);
  - Invoked when the given VMA is added to an address space.
- void close (struct vm_area_struct *area);
- struct page *nopage(struct vm_area_struct *area,
  unsigned long address, int unused);
  - Invoked by the page fault handler when a page that is not present in physical memory is accessed.
- int populate(struct vm_area_struct *area,
  unsinged long address, unsigned long len,
  pgprot_t prot, unsigned long pgoff, int nonblock);

VMA (5)

- **/proc/<pid>/maps**
  - start – end permission offset major:minor inode file

```
cafe.kaist.ac.kr - PuTTY
[dev1:/home/jinsoo-81] cat &
[1] 6117
[dev1:/home/jinsoo-82] cat /proc/6117/maps
00544000-0055d000 r-xp 00000000 08:02 2498325    /lib/ld-2.4.so
0055d000-0055e000 r--p 00018000 08:02 2498325    /lib/ld-2.4.so
0055e000-0055f000 rw-p 00019000 08:02 2498325    /lib/ld-2.4.so
00561000-0068d000 r-xp 00000000 08:02 2498326    /lib/libc-2.4.so
0068d000-00690000 r--p 0012b000 08:02 2498326    /lib/libc-2.4.so
00690000-00691000 rw-p 0012e000 08:02 2498326    /lib/libc-2.4.so
00691000-00694000 rw-p 00691000 00:00 0
08048000-0804d000 r-xp 00000000 08:02 2171880    /bin/cat
0804d000-0804e000 rw-p 00004000 08:02 2171880    /bin/cat
0804e000-0806f000 rw-p 0804e000 00:00 0          [heap]
b7d8a000-b7db7000 r--p 011e8000 08:02 1753893    /usr/lib/locale/locale-archive
b7db7000-b7fb7000 r--p 00000000 08:02 1753893    /usr/lib/locale/locale-archive
b7fb7000-b7fb8000 rw-p b7fb7000 00:00 0
b7fc1000-b7fc2000 r--p 029ec000 08:02 1753893    /usr/lib/locale/locale-archive
b7fc2000-b7fc7000 r--p 011e0000 08:02 1753893    /usr/lib/locale/locale-archive
b7fc7000-b7fc8000 rw-p b7fc7000 00:00 0
bfa27000-bfa3c000 rw-p bfa27000 00:00 0          [stack]
ffffe000-fffff000 ---p 00000000 00:00 0          [vdso]
[1]  + Suspended (tty input)         cat
[dev1:/home/jinsoo-83]
```

VMA (6)

- **pmap**

cafe.kaist.ac.kr - PuTTY

```
[dev1:/home/jinsoo-98] cat &
[1] 6652
[dev1:/home/jinsoo-99] pmap 6652
6652:   cat
00544000    100K r-x--  /lib/ld-2.4.so
0055d000      4K r----  /lib/ld-2.4.so
0055e000      4K rw---  /lib/ld-2.4.so
00561000   1200K r-x--  /lib/libc-2.4.so
0068d000     12K r----  /lib/libc-2.4.so
00690000      4K rw---  /lib/libc-2.4.so
00691000     12K rw---    [ anon ]
08048000     20K r-x--  /bin/cat
0804d000      4K rw---  /bin/cat
0804e000    132K rw---    [ anon ]
b7ce8000    180K r----  /usr/lib/locale/locale-archive
b7d15000   2048K r----  /usr/lib/locale/locale-archive
b7f15000      4K rw---    [ anon ]
b7f1f000      4K r----  /usr/lib/locale/locale-archive
b7f20000     20K r----  /usr/lib/locale/locale-archive
b7f25000      4K rw---    [ anon ]
bfe7c000     84K rw---    [ stack ]
ffffe000      4K -----    [ anon ]
 total     3840K
[1]  + Suspended (tty input)         cat
[dev1:/home/jinsoo-100]
```

Memory Descriptor (1)

- **struct mm_struct** <linux/sched.h>
  - Contains all the information related to the process address space.
  - Threads share a memory descriptor.
  - Doubly linked via the mmlist field.

struct vm_area_struct *	mmap;	list of memory areas (VMAs)
struct rb_root	mm_rb;	red-black tree of memory areas
pgd_t *	pgd;	page global directory
atomic_t	mm_users;	address space users
atomic_t	mm_count;	reference count for mm_struct
int	map_count;	number of memory areas
struct list_head	mmlist;	list of all mm_structs
...		

Memory Descriptor (2)

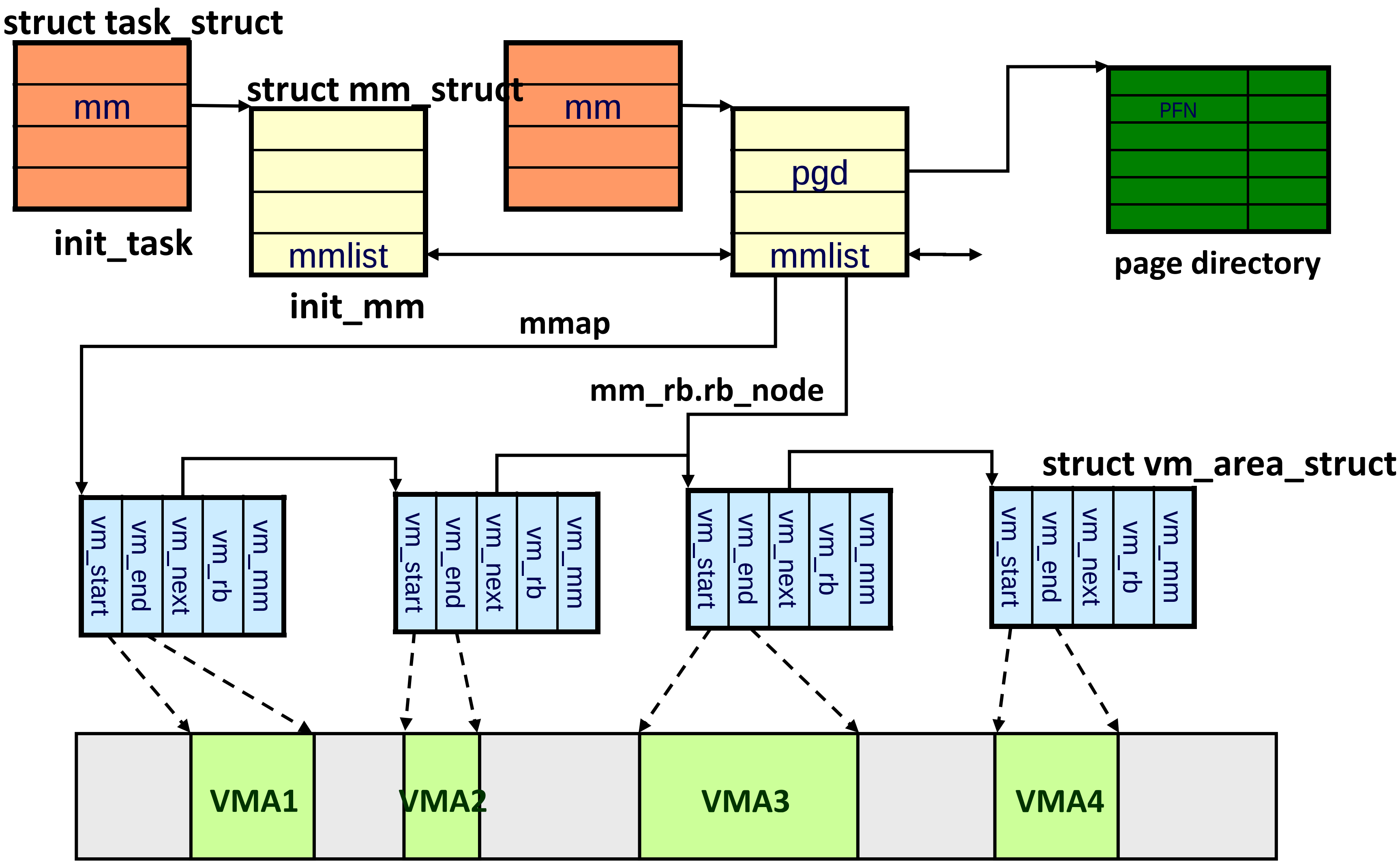
- **Allocating / freeing a memory descriptor**
  - allocate_mm() <kernel/fork.c>
    = kmem_cache_alloc(mm_cachep, GFP_KERNEL)
  - free_mm(mm)
    = kmem_cache_free(mm_cachep, (mm))

```
copy_mm()  {                                        <kernel/fork.c>
    …
    if (clone_flags & CLONE_VM) {                   // thread creation
       atomic_inc(&current->mm->mm_users);
       tsk->mm = current->mm;
    } else {                                        // process creation
       tsk->mm = allocate_mm();
       mm_init(tsk->mm);
       …
    }
}
```

Memory Descriptor (3)

- **Kernel threads**
  - Kernel threads do not have a process address space and therefore do not have a memory descriptor.
  - Kernel threads use the memory descriptor of whatever task ran previously.
    - task→active_mm: address space referenced by a process
    - When a kernel thread is scheduled (task→mm == NULL), the kernel keeps the previous process's address space loaded.
    - The kernel updates the active_mm field.
    - The kernel thread can then use the previous process's page tables as needed.
  - Kernel threads use only the information pertaining to kernel memory. (same for all processes)

Memory Descriptor (4)



do_mmap() (1)

- unsigned long **do_mmap (**struct file *file,
  unsigned long addr, unsigned long len,
  unsigned long prot, unsigned long flag,
  unsigned long offset**);**
  - **file** and **offset**:
    - Specified when the memory region is backed by a file (file-backed mapping)
    - file==NULL and offset==0 for anonymous mapping
  - **addr**: linear address where the search for a free interval must start.
  - **prot**: the access rights of the pages in the region
  - **flag**: the memory region flags

do_mmap() (2)

- **Protection**
  - PROT_READ (= VM_READ)
  - PROT_WRITE (= VM_WRITE)
  - PROT_EXEC (= VM_EXEC)
  - PROT_NONE: none of access rights
- **Flags**
  - MAP_SHARED: shared among several processes
  - MAP_PRIVATE: private to this process
  - MAP_FIXED: must be exactly the specified address
  - MAP_ANONYMOUS: no file is associated
  - MAP_POPULATE: pre-allocate the page frames

do_mmap() (3)

- **Implementation for anonymous mapping**
  - Check the parameters
  - Obtain a linear address interval for the new region
    - get_unmapped_area()
  - Compute VM flags based on **prot** and **flags** parameters
  - Locate a VMA structure that precedes the new interval
    - find_vma_prepare()
  - Check against address space limit
  - Check if an old private anonymous mapping can be expanded.
    - They should have exactly the same flags.
    - If so, merge two VMAs using vma_merge()

do_mmap() (4)

- **Implementation (cont'd)**
  - Allocate a vm_area_struct for the new memory region
    - kmem_cache_alloc()
  - Initialize the vm_area_struct
  - Insert the new region in the memory region list and red-black tree
    - vma_link()
  - Increase the accounting information
    - mm->total_vm
  - If VM_LOCKED is set, allocate all the pages of the memory region and lock them in RAM
    - make_pages_present()
  - Return the linear address of the new memory region

do_munmap() (1)

- int **do_munmap (**struct mm_struct *mm,
  unsigned long start, size_t len**);**
  - **mm**: the process's memory descriptor
  - **start**: the starting address of the interval
  - **len**: the length of the interval

  - Phase 1: Scan the list of memory regions owned by the process and unlink all regions included in the linear address space of the process address space.
  - Phase 2: update the process page tables and remove the memory regions

do_munmap() (2)

- **Phase 1 implementation**
  - Check the parameters
  - Locate the first memory region that ends after the linear address interval to be deleted.
    - find_vma_prev()
  - Split the first memory region if the start address lies inside the region
    - split_vma()
  - Split the last memory region if the linear address ends inside the region
    - split_vma()

do_munmap() (3)

- **Phase 2 implementation**
  - Remove the memory regions included in the linear address interval from the process's linear address space
    - detach_vmas_to_be_unmapped()
  - Clear the page table entries covering the linear address interval and free the corresponding page frames
    - unmap_region()
  - Release the descriptors of the memory regions and adjust the accounting information
    - remove_vma_list()

Page Fault Handling