Process Address Space

Paolo Calafiura <pcalafiura@lbl.gov>
June 18 2003

Disclaimer

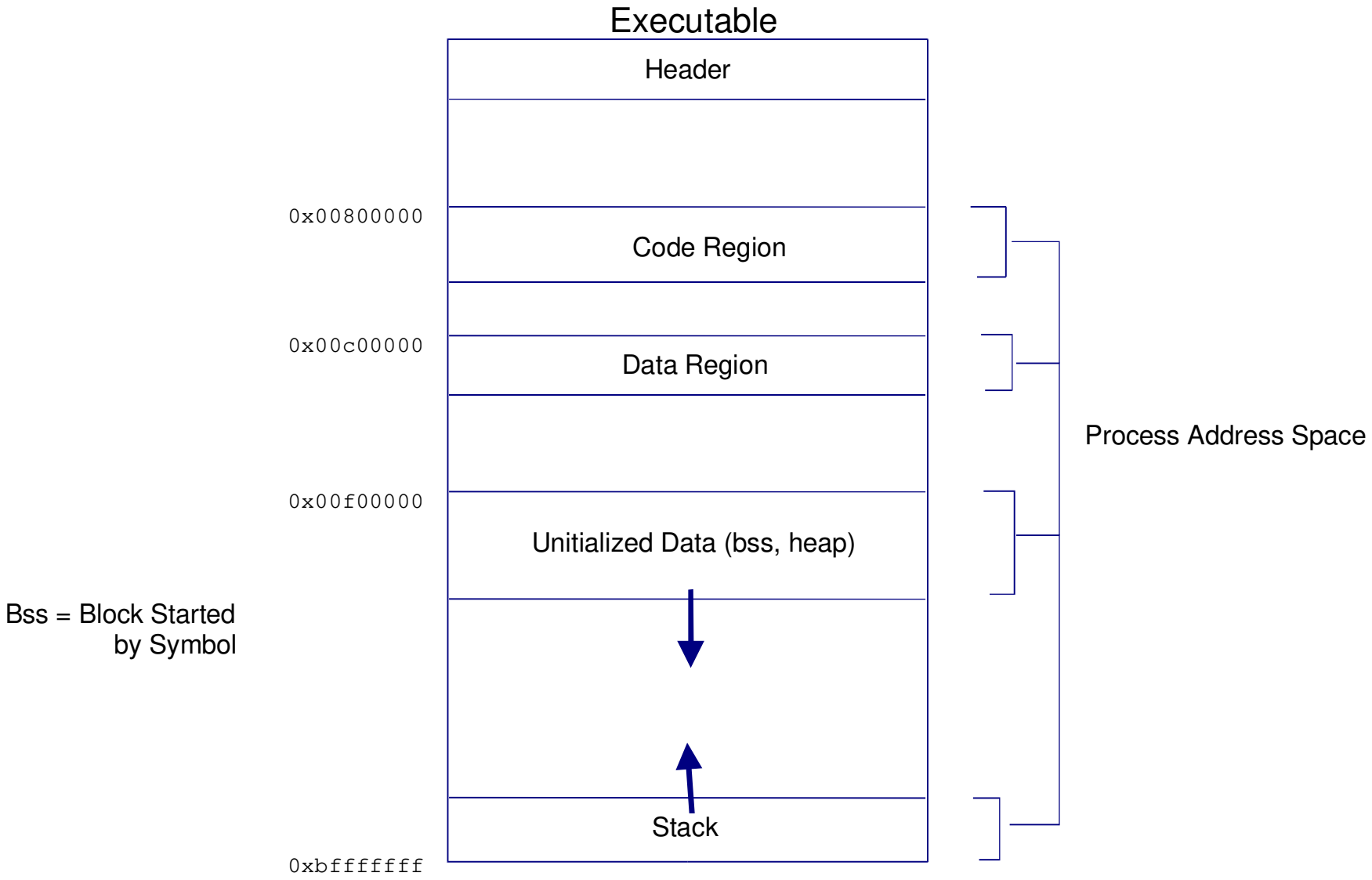
I don't really know what I am talking about

- But I am used to this, so don't worry...

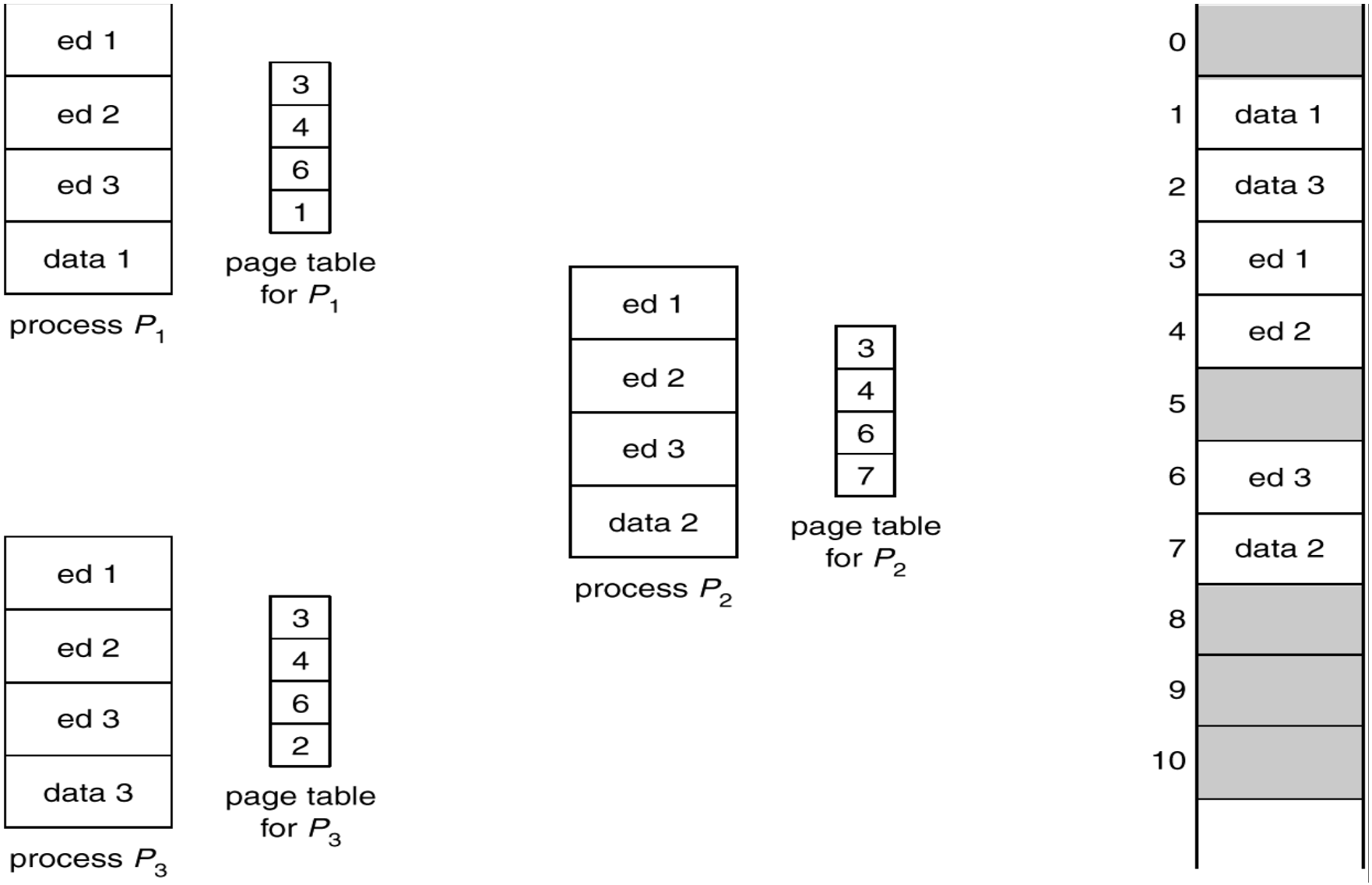
Overview

- Process Address Space
  - Executable memory regions
  - Process Memory Allocation
- How Kernel Manages the Address Space
  - VM Architecture
  - Memory Descriptors, Regions
  - Page Fault Handling
    - Demand paging, COW

Process Address Space



Shared Code Pages



Stack vs Heap Allocation

```
char buff[5000];
transmogrify(buff);
```

- Stack allocation
  - Local (automatic) vars
  - Nested functions
  - Kernel managed
    - Initial size (e.g. 4K)
    - Incremented (e.g by 2K) on stack push via page fault

```
char* p(new char[5000]);
delete p;
```

- new/delete, malloc/free
  - Use sys_brk() internally
    - Align to page
    - Check limits, intersections
    - Allocate linear addr interval

Other Memory Requests

- An executable is launched when the user types a command at the console
  - Kernel assigns one or more memory regions and maps the process address space to them
- Release an address space and/or load a fresh one for a new executable (_exit(), fork(), execve())
- mmap() address space to file (and munmap())
- Create a shared memory region (shmat()/shmdt())

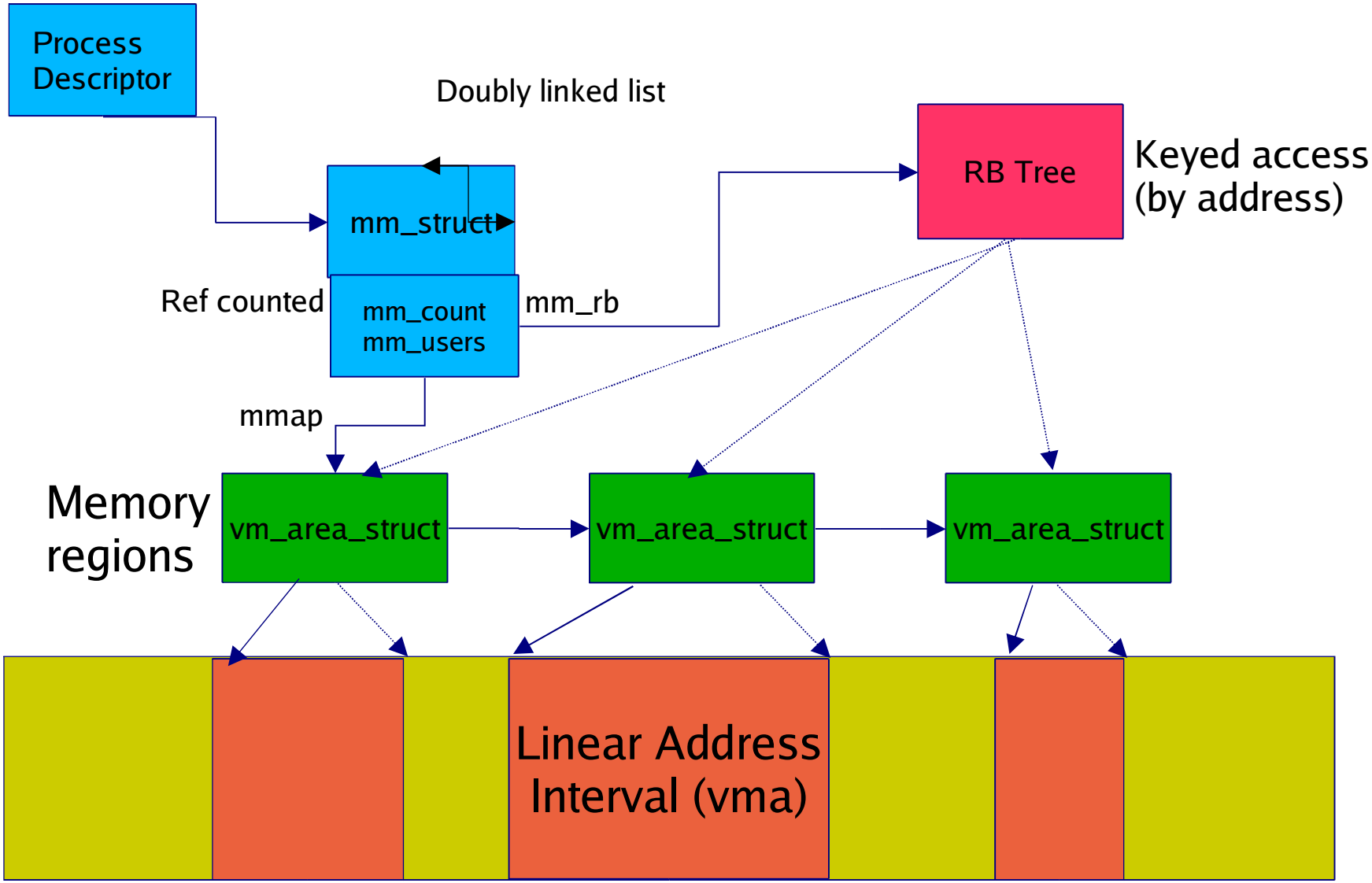
Kernel Role

- Allocate and deallocate RAM pages on behalf of the process
- Load pages reading from disk the process data
- Keep track of the pages allocated for each process and of their linear address mapping

Kernel Strategy

- Virtual Memory Abstraction
  - Hide address binding from process
    - A process sees memory as a flat address range
- Page Fault Mechanism
  - Assume user processes to be
    - Dynamic (expect memory requests at all times)
    - Unreliable (verify every memory request)
  - Optimize RAM usage rather than access time
    - Serve memory requests at the latest possible moment (on memory access)

Memory Management



Memory Descriptor

```
struct mm_struct {

  struct vm_area_struct * mmap;
   /* list of VMAs */

  rb_root_t mm_rb; /* root RB tree */

  struct vm_area_struct * mmap_cache;
   /* last find_vma result */

  atomic_t mm_users;
   /* How many users with user space? */

  atomic_t mm_count;
   /* How many references to "struct
   mm_struct" (users count as 1) */

  struct list_head mmlist;
   /* List of all active mm's. ... */

  unsigned long start_code, end_code,
   start_data, end_data;

  unsigned long start_brk, brk, start_stack;

  ....

 };
```

- One per process
  - manage memory regions
    - iterate over all regions
    - access one by address
  - Ref counted by
    - number of users
    - number of kernel refs (all user procs count 1)
  - linked list
  - Exec region pointers to
    - text, data, heap, stack

Memory Region

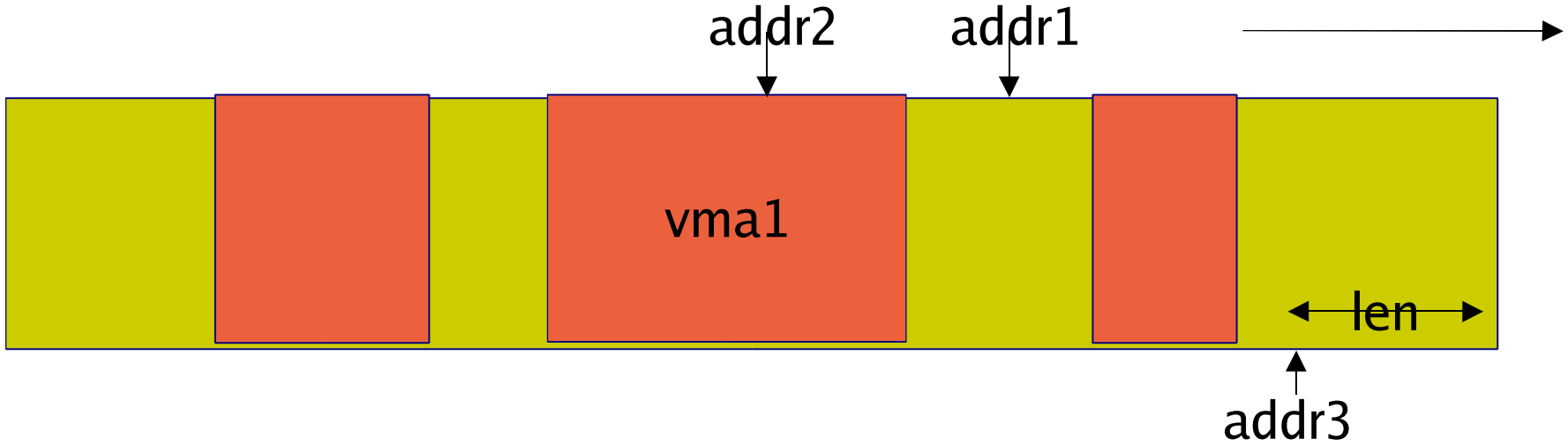
```
struct vm_area_struct {

  struct mm_struct * vm_mm;
   /* The address space we belong to. */

  unsigned long vm_start, vm_end;
   /* Start/end address within vm_mm. */

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

  rb_node_t vm_rb;

  pgprot_t vm_page_prot;
   /* Access permissions of this VMA. */

  unsigned long vm_flags;
   /* VMA Flags */

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

- One per vma per process
  vma is page fault "unit"
    - shared lib, executable
- vma offsets
- Linked list sorted by address
- node of rb tree
- vma protection and flags
    - READ, WRITE, EXEC,...
- pointer to vma ops
    - open (add vma)
    - close (remove vma)
    - nopage (used by Page Fault)

Memory Region Helpers

Functions to support vma (de)allocation

- vma1 = find_vma(mm, addr1);
- vma1 = find_vma_intersection(mm, addr2);

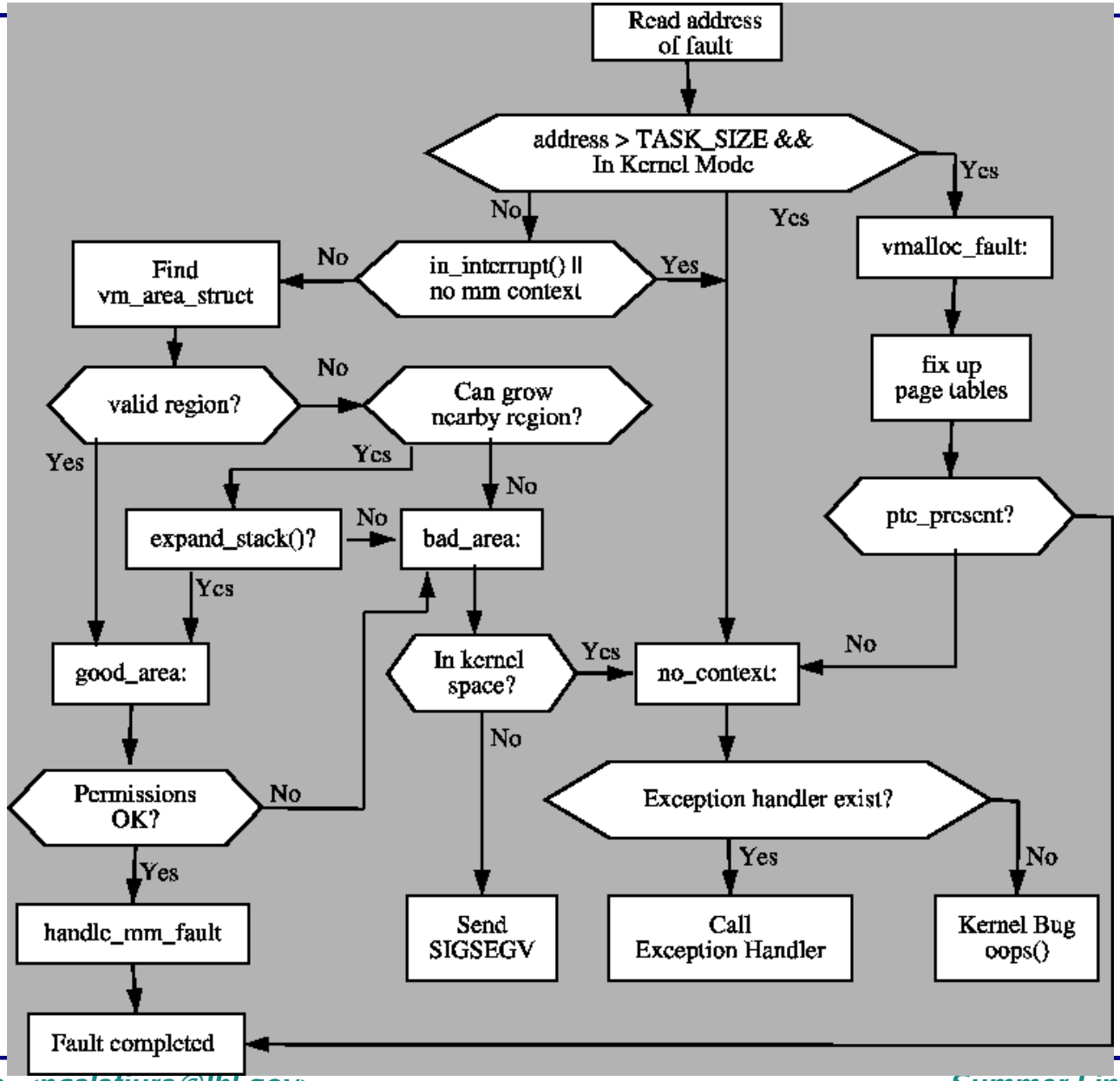


- addr3 = arch_get_unmapped_area(0,addr1,len,0,flags);
  (assume no file mapping)
  - page aligned address for a vma of size len
- insert_vm_struct(mm, vma)

Page Faulting

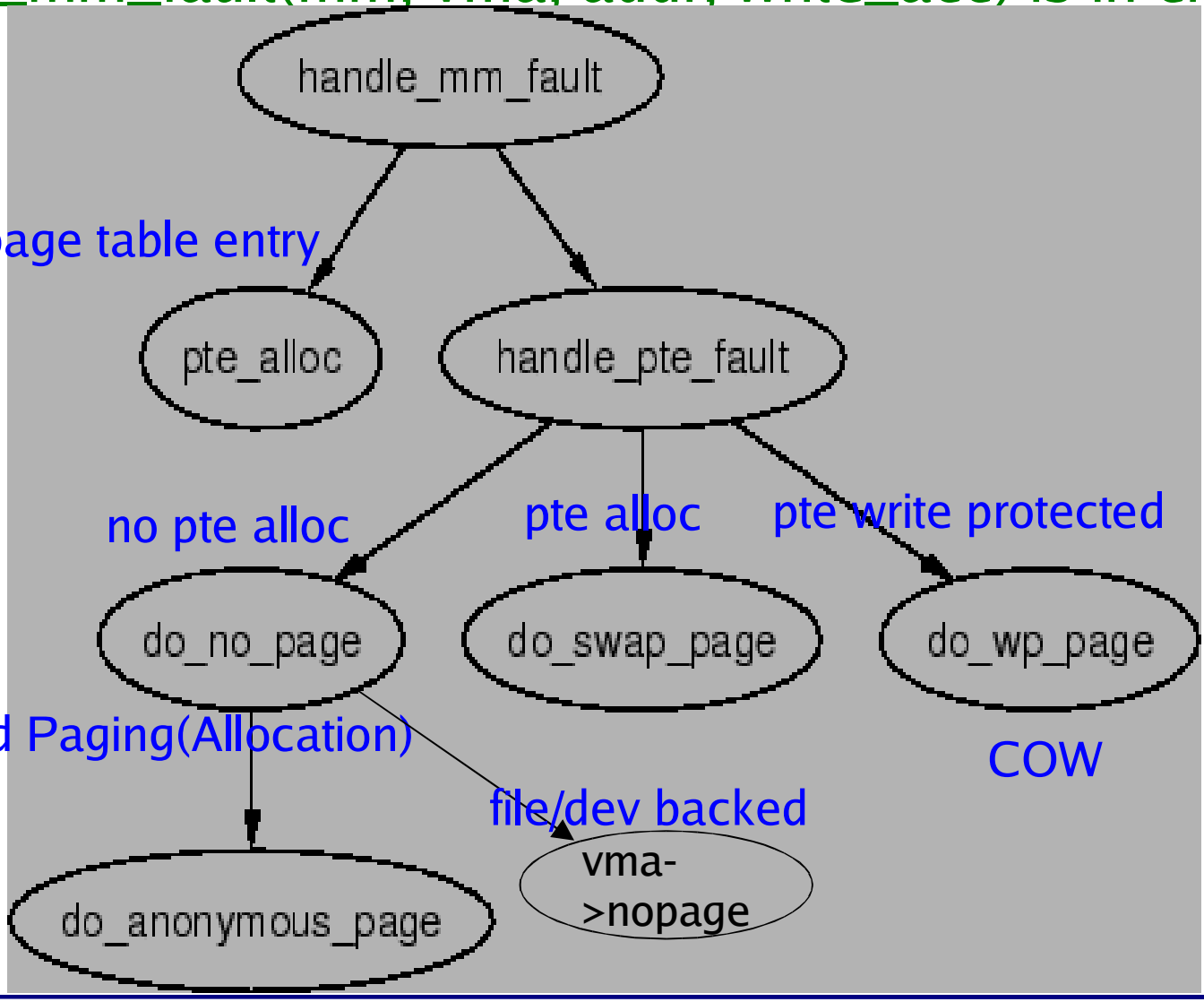
- Linux, like most op sys, uses *Demand Fetching* for non-resident pages. For example
  - user requested vma (no associated page yet) – Minor PF
  - page swapped out to disk – Major PF (process sleep)
  - region is invalid or wrong permissions – Error PF
- Hardware raises a page fault. Kernel traps it and invokes handler (for x86 do_page_fault())
- Handler is given
  - address of fault
  - page not found or invalid access right
  - read/exec or write access
  - User or Kernel Mode

do_page_fault()



Handling a Valid Page Fault

handle_mm_fault(mm, vma, addr, write_acc) is in charge



Demand Paging: do_anonymous_page()


```
/* Read-only mapping of ZERO_PAGE. */
entry = pte_wrprotect(mk_pte(ZERO_PAGE(addr), vma->vm_page_prot));

/* ..except if it's a write access */
if (write_access) {
        struct page *page;

        /* Allocate our own private page. */
        spin_unlock(&mm->page_table_lock);

        page = alloc_page(GFP_HIGHUSER);
        if (!page)
                goto no_mem;
        clear_user_highpage(page, addr);

        spin_lock(&mm->page_table_lock);
        if (!pte_none(*page_table)) {
                page_cache_release(page);
                spin_unlock(&mm->page_table_lock);
                return 1;
        }
        mm->rss++;
        flush_page_to_ram(page);
        entry = pte_mkwrite(pte_mkdirty(mk_pte(page, vma->vm_page_prot)));
        lru_cache_add(page);
        mark_page_accessed(page);
}

set_pte(page_table, entry);

/* No need to invalidate - it was non-present before */
update_mmu_cache(vma, addr, entry);
spin_unlock(&mm->page_table_lock);
return 1;       /* Minor fault */
```

if read-only map to ZERO_PAGE

- statically allocated
- non-writable, use COW

if writable allocate

fill with 0s

mark as writable & dirty

prepare for swap

Copy On Write

- efficient process fork():
  only child-modified pages (often none) are copied
- each pte is refcounted
- do_wp_page()
  - check pte refcount > 1
  - copies old page frame into new one
    - special handling for ZERO_PAGE (avoid cache trashing)
  - create new writable pte
  - decrement old pte refcount

References

- Bovet & Cesati, chapter 8
- M. Gorman Thesis, chapter 5

  www.csn.ul.ie/~mel/projects/vm/
- V. Asta, Op Sys Lecture Notes (in Italian!)

  www.axdigital.com/~vito/sotv/dispense/Disp_P1_06.sxi.gz
- J. Levine, Linkers and Loaders

  www.iecc.com/linker/