# CS 6V81-05: System Security and Malicious Code Analysis
# Understanding the Implementation of Virtual Memory

Zhiqiang Lin

Department of Computer Science
University of Texas at Dallas

February $29^{th}$, 2012



## Outline

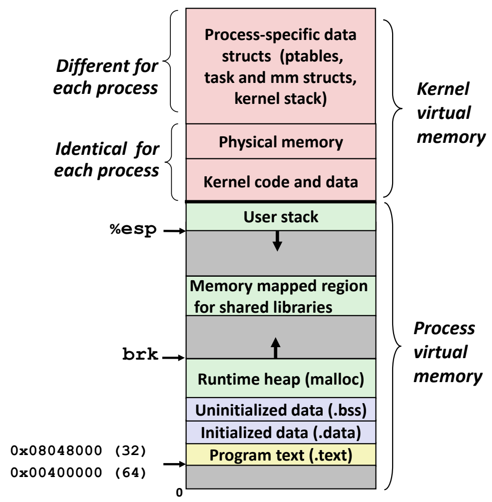


## Outline

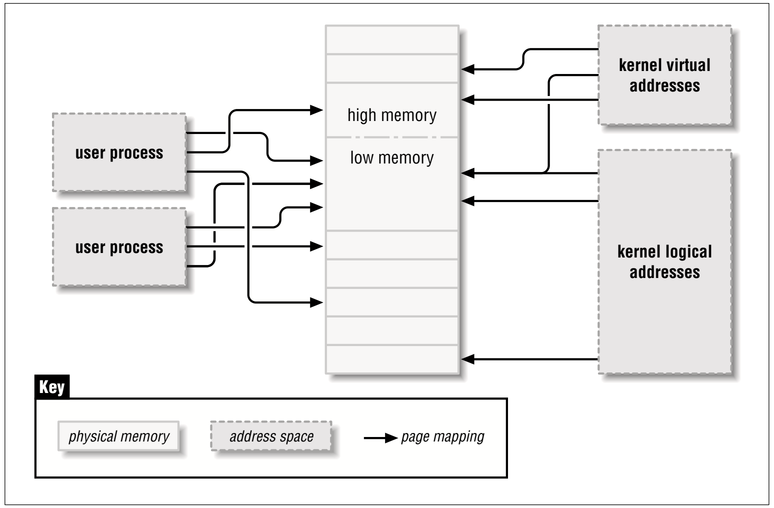




## Virtual Memory of a Linux Process

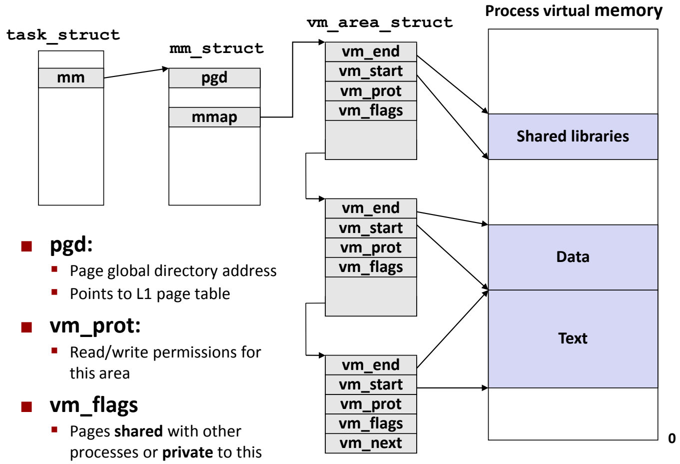


# Address type in Linux





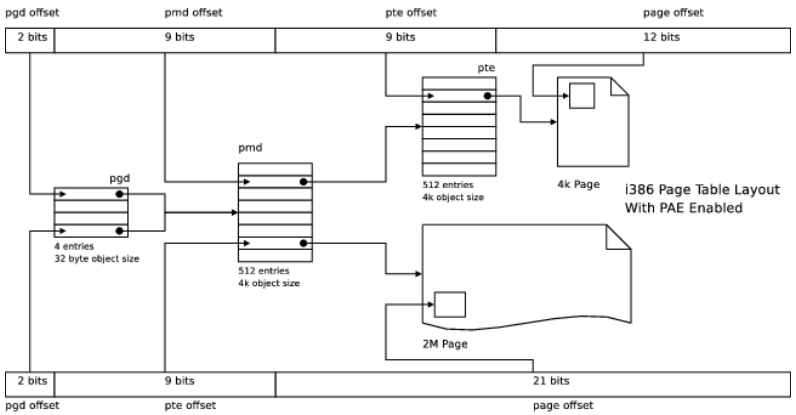

# Linux Organizes VM as Collection of “Areas”




- **pgd:**
  - Page global directory address
  - Points to L1 page table
- **vm_prot:**
  - Read/write permissions for this area
- **vm_flags**
  - Pages **shared** with other processes or **private** to this process



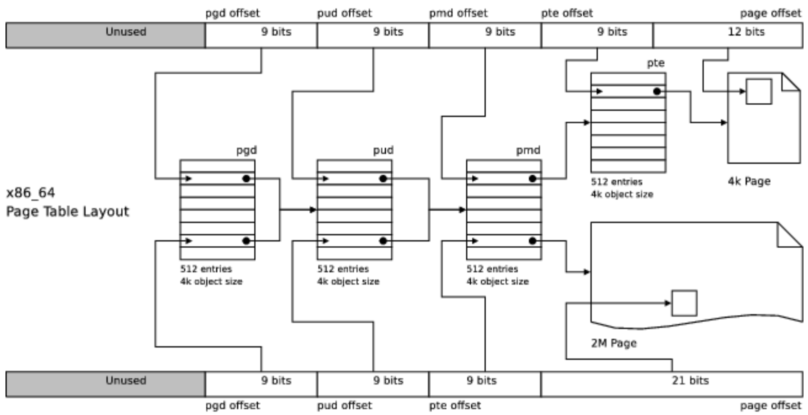
# i386+PAE




sources: http://linux-mm.org/PageTableStructure

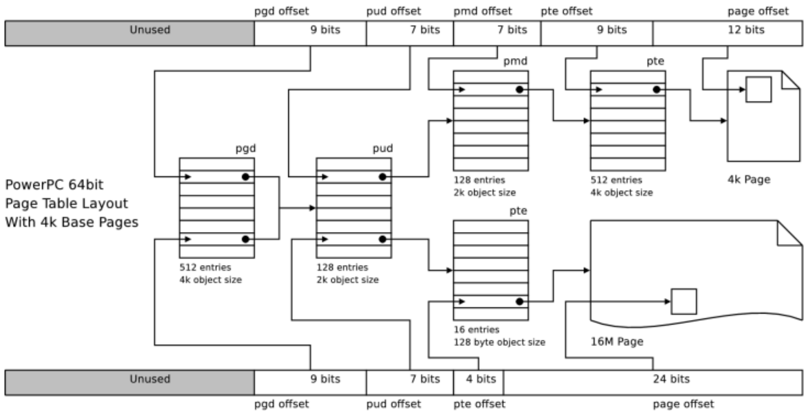


# x86_64




sources: http://linux-mm.org/PageTableStructure

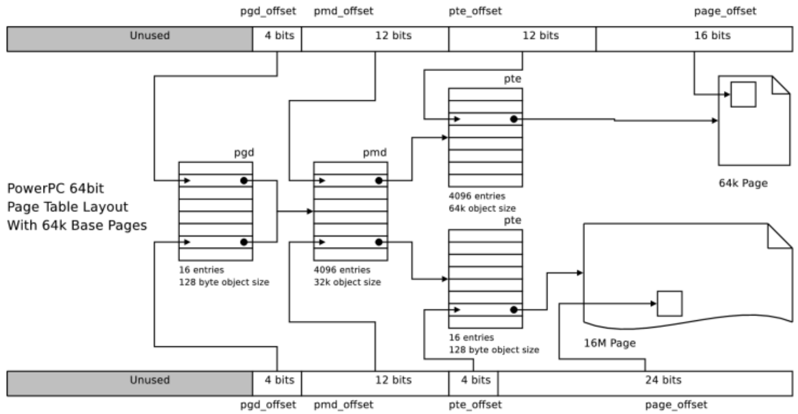
# Powerpc-4k




sources: http://linux-mm.org/PageTableStructure



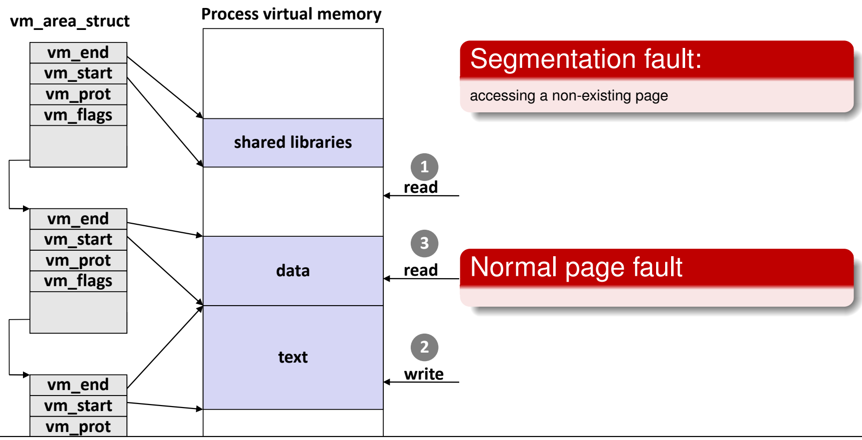
# Powerpc-64k




sources: http://linux-mm.org/PageTableStructure



# Simplifying Linking and Loading

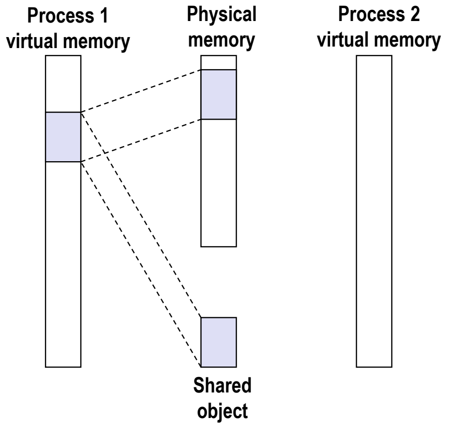





# Demand paging

- Key point: no virtual pages are copied into physical memory until they are referenced!
    - Known as demand paging
- Crucial for time and space efficiency

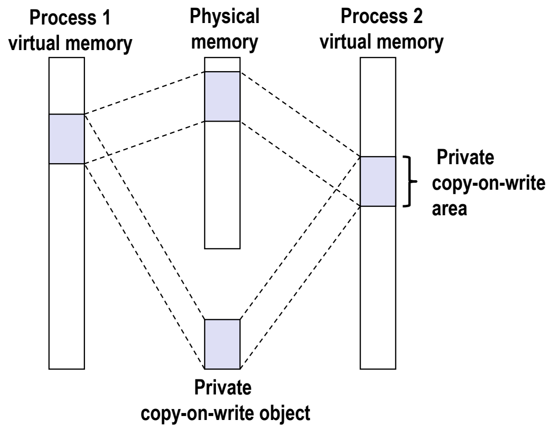
## Shared Objects




- Process 1 maps the shared object.
- Notice how the virtual addresses can be different.



## Private Copy-on-write (COW) Objects




- Two processes mapping a private copy-on-write (COW) object.
- Area flagged as private copy-on-write.
- PTEs in private areas are flagged as read-only
- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!

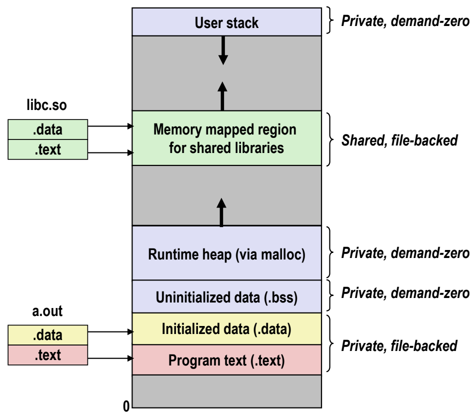


## Demand paging

- VM and memory mapping explain how fork provides private address space for each process.
- To create virtual address for new new process
  - Create exact copies of current mm_struct, vm_area_struct, and page tables.
  - Flag each page in both processes as read-only
  - Flag each vm_area_struct in both processes as private COW
- On return, each process has exact copy of virtual memory
- Subsequent writes create new pages using COW mechanism.



## The execve Function




- To load and run a new program a.out in the current process using execve:
- Free vm_area_struct's and page tables for old areas
- Create vm_area_struct's and page tables for new areas
  - Programs and initialized data backed by object files.
  - .bss and stack backed by anonymous files .
- Set PC to entry point in .text
  - Linux will fault in code and data pages as needed.

## User-Level Memory Mapping

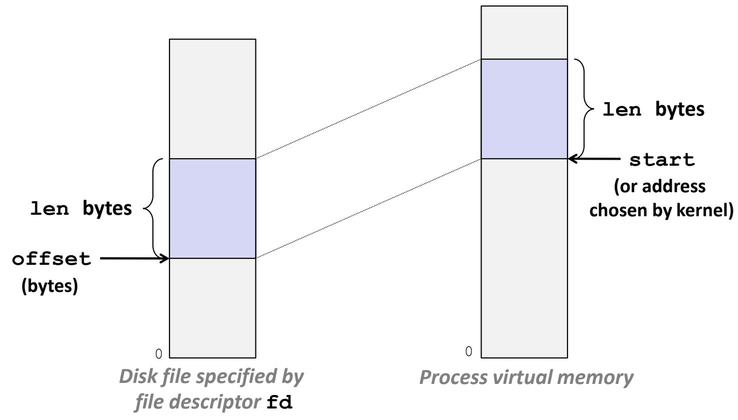
**void *mmap(void *start, int len, int prot, int flags, int fd, int offset)**

- Map len bytes starting at the offset of the file specified by file description fd, preferably at address start
  - **start:** may be 0 for "pick an address"
  - **prot:** PROT_READ, PROT_WRITE, ...
  - **flags:** MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...
- Return a pointer to start of mapped area (may not be start)



## User-Level Memory Mapping

**void *mmap(void *start, int len, int prot, int flags, int fd, int offset)**



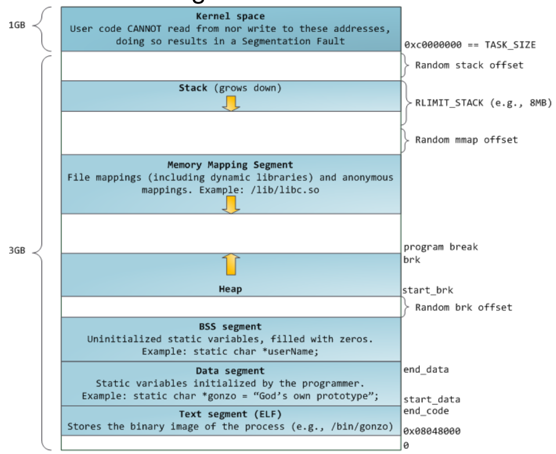

## Outline





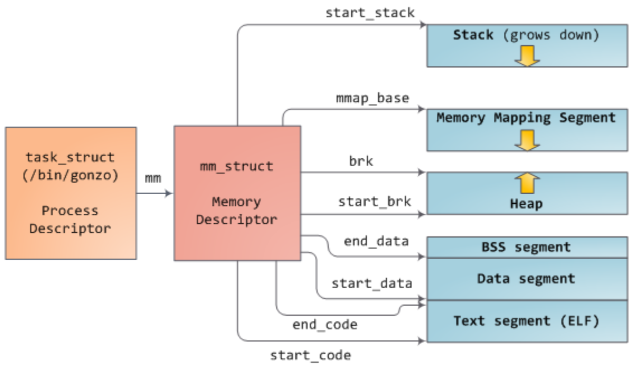
## Memory Management

**Memory management** is the heart of operating systems; Each process in a multi-tasking OS runs in its virtual address space.




Credit: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory

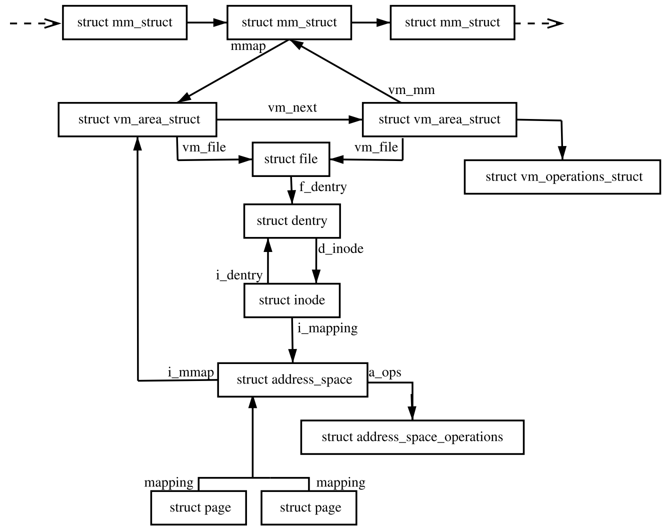
# How The Kernel Manages Process Memory




Credit: http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory



# Relevant data structures in address space






# mm_struct

```
struct mm_struct {
     [0] struct vm_area_struct *mmap;
     [4] struct rb_root mm_rb;
     [8] struct vm_area_struct *mmap_cache;
    [12] long unsigned int (*get_unmapped_area)(struct file *, long uns
    [16] void (*unmap_area)(struct mm_struct *, long unsigned int);
    [20] long unsigned int mmap_base;
    [24] long unsigned int task_size;
    [28] long unsigned int cached_hole_size;
    [32] long unsigned int free_area_cache;
    [36] pgd_t *pgd;
    [40] atomic_t mm_users;
    [44] atomic_t mm_count;
    [48] int map_count;
    [52] struct rw_semaphore mmap_sem;
    [80] spinlock_t page_table_lock;
    [96] struct list_head mmlist;
   [104] mm_counter_t _file_rss;
   [108] mm_counter_t _anon_rss;
   [112] long unsigned int hiwater_rss;
   [116] long unsigned int hiwater_vm;
```



# mm_struct

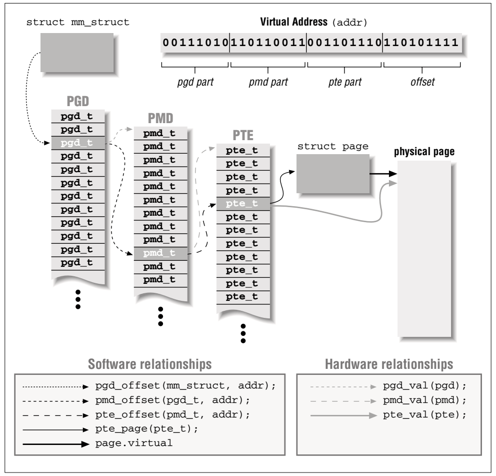
```
struct mm_struct {
  [120] long unsigned int total_vm;
  [124] long unsigned int locked_vm;
  [128] long unsigned int shared_vm;
  [132] long unsigned int exec_vm;
  [136] long unsigned int stack_vm;
  [140] long unsigned int reserved_vm;
  [144] long unsigned int def_flags;
  [148] long unsigned int nr_ptes;
  [152] long unsigned int start_code;
  [156] long unsigned int end_code;
  [160] long unsigned int start_data;
  [164] long unsigned int end_data;
  [168] long unsigned int start_brk;
  [172] long unsigned int brk;
  [176] long unsigned int start_stack;
  [180] long unsigned int arg_start;
  [184] long unsigned int arg_end;
  [188] long unsigned int env_start;
  [192] long unsigned int env_end;
```

## mm_struct

```
struct mm_struct {
  [196] long unsigned int saved_auxv[44];
  [372] unsigned int dumpable : 2;
  [376] cpumask_t cpu_vm_mask
  [380] mm_context_t context;
  [424] long unsigned int swap_token_time;
  [428] char recent_pagein;
  [432] int core_waiters;
  [436] struct completion *core_startup_done;
  [440] struct completion core_done;
  [468] rwlock_t ioctx_list_lock;
  [484] struct kioctx *ioctx_list;
}
SIZE: 488
```

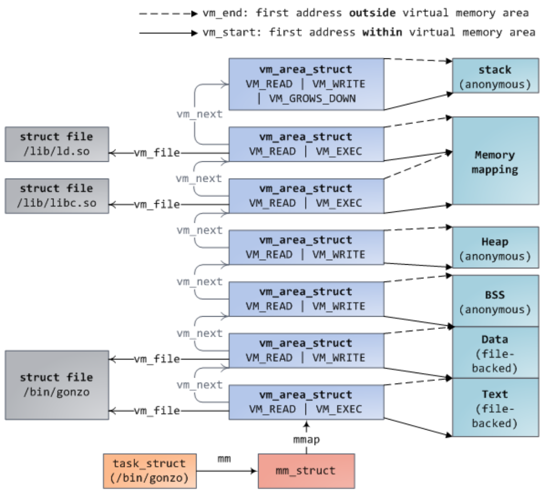


## Page Directory






## vm_area_struct



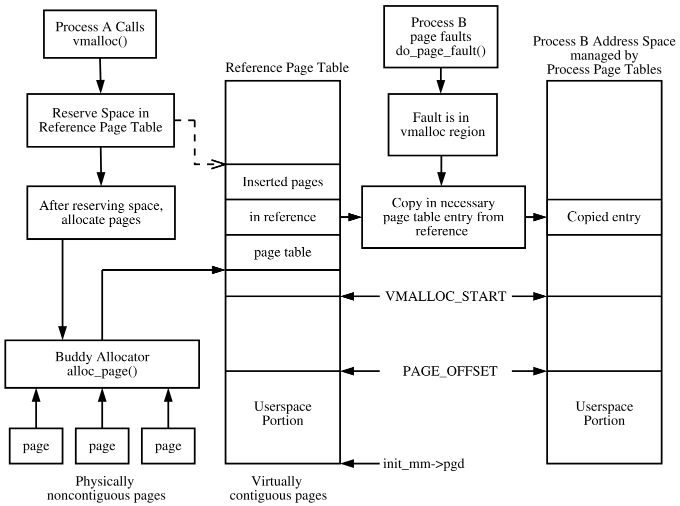

Credit: http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory



## vm_area_struct

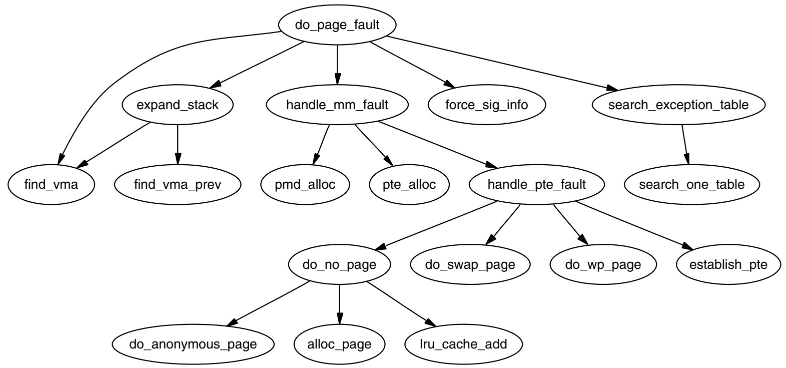
```
struct vm_area_struct {
   [0] struct mm_struct *vm_mm;
   [4] long unsigned int vm_start;
   [8] long unsigned int vm_end;
  [12] struct vm_area_struct *vm_next;
  [16] pgprot_t vm_page_prot;
  [20] long unsigned int vm_flags;
  [24] struct rb_node vm_rb;
       union {
           struct {...} vm_set;
           struct raw_prio_tree_node prio_tree_node;
  [36] } shared;
  [52] struct list_head anon_vma_node;
  [60] struct anon_vma *anon_vma;
  [64] struct vm_operations_struct *vm_ops;
  [68] long unsigned int vm_pgoff;
  [72] struct file *vm_file;
  [76] void *vm_private_data;
  [80] long unsigned int vm_truncate_count;
}
SIZE: 84
```

## Page Fault



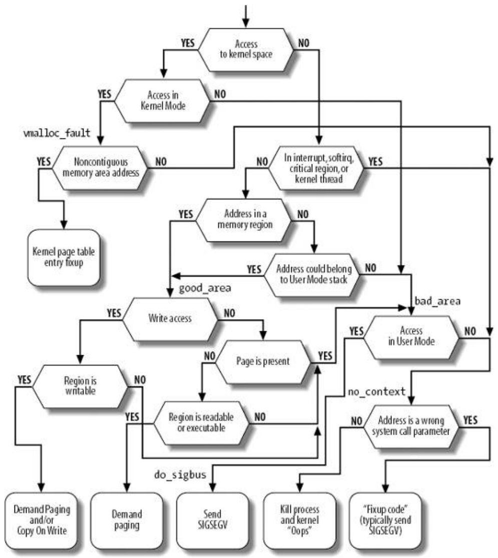



## do_page_fault






## do_page_fault




## Outline

1. Overview
2. Implementation
3. Summary

## Summary

- Memory management is crucial
  - Machine introspection
  - Memory forensics
  - Traversing kernel data structures to understand the memory
- Memory failures
- Exploits



## References


- Dening BEFORE MEMORY WAS VIRTUAL http://cs.gmu.edu/cne/pjd/PUBS/bvm.pdf
- Linux device drivers, Addison-wisely.
- http://www.cs.cmu.edu/afs/cs/academic/class/15213-f10/www/lectures/16-vm-systems.pdf
- Understanding Linux virtual memory manager http://www.kernel.org/doc/gorman/pdf/understand.pdf
- Understanding Linux kernel (3rd edition)
- http://linux-mm.org/PageTableStructure