Memory Mapping

Sarah Diesburg
COP5641

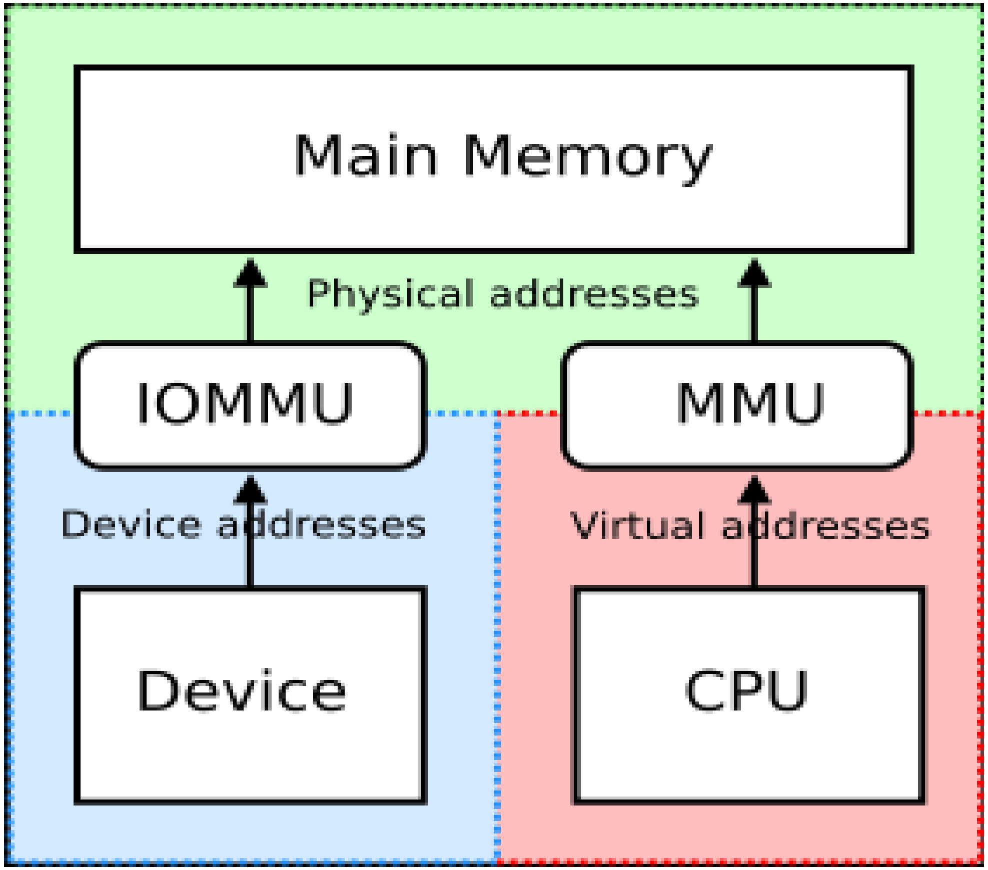
Memory Mapping

- Translation of address issued by some device (e.g., CPU or I/O device) to address sent out on memory bus (physical address)
- Mapping is performed by memory management units

Memory Mapping

- CPU(s) and I/O devices may have different (or no) memory management units
  - No MMU means direct (trivial) mapping
- Memory mapping is implemented by the MMU(s) using page (translation) tables stored in memory
- The OS is responsible for defining the mappings, by managing the page tables

Memory Mapping

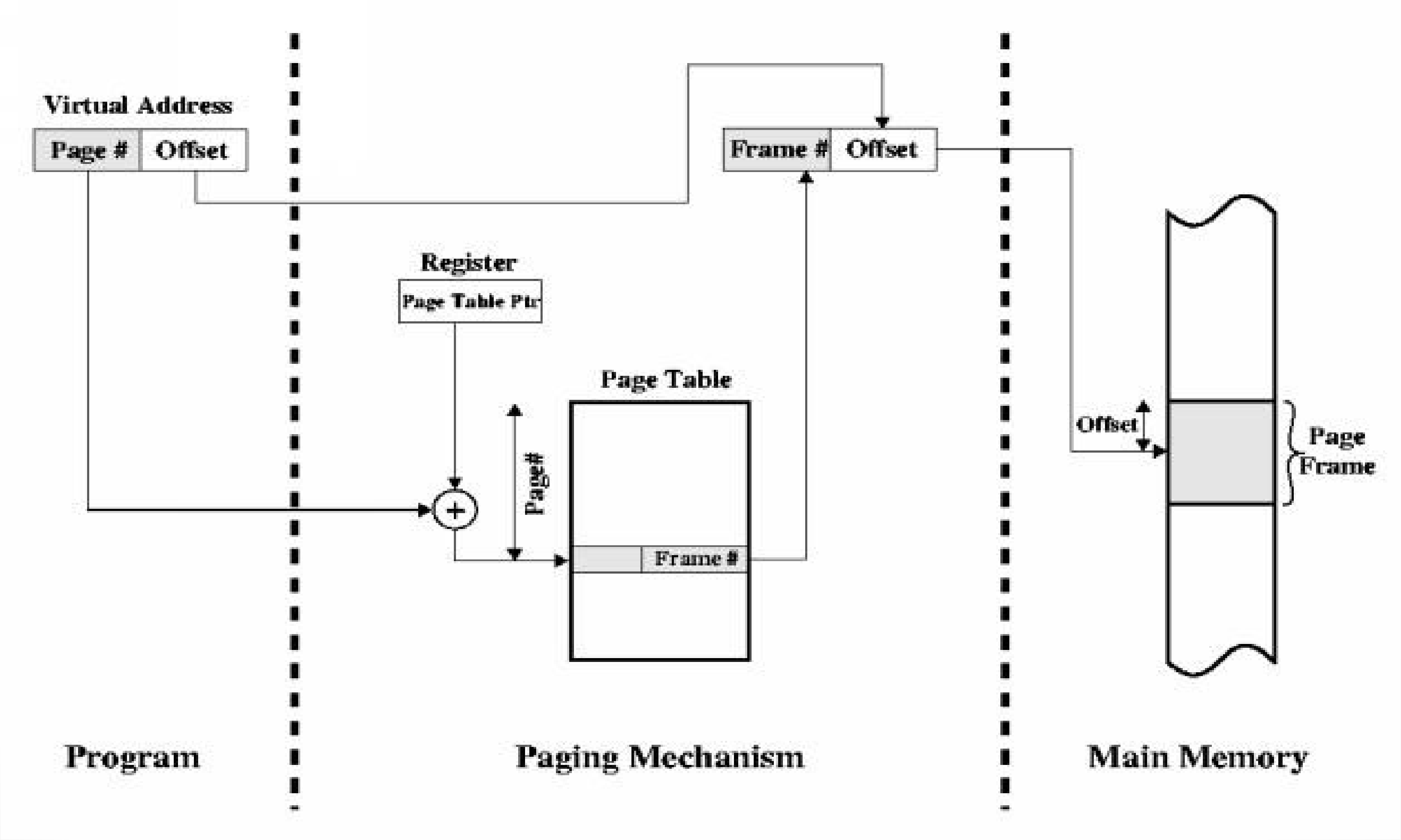


AGP and PCI Express graphics cards us a Graphics Remapping Table (GART), which is one example of an IOMMU. See Wiki article on IOMMU for more detail on memory mapping with I/O devices. http://en.wikipedia.org/wiki/IOMMU

Memory Mapping

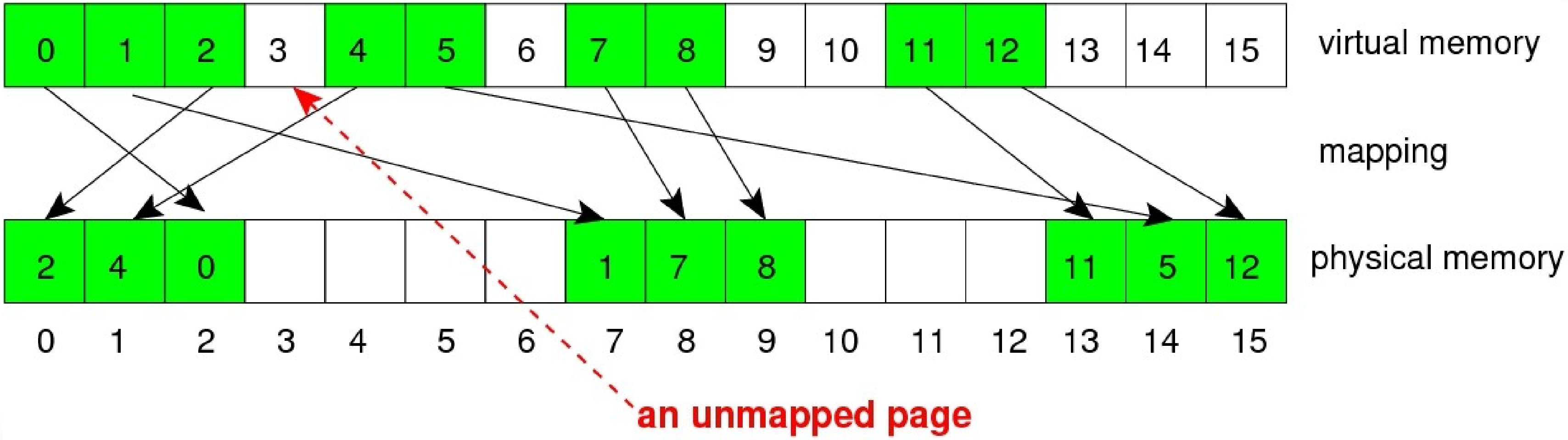
- Typically divide the virtual address space into pages
  - Usually power of 2
- The offset (bottom n bits) of the address are left unchanged
- The upper address bits are the virtual page number

Address Mapping Function (Review)



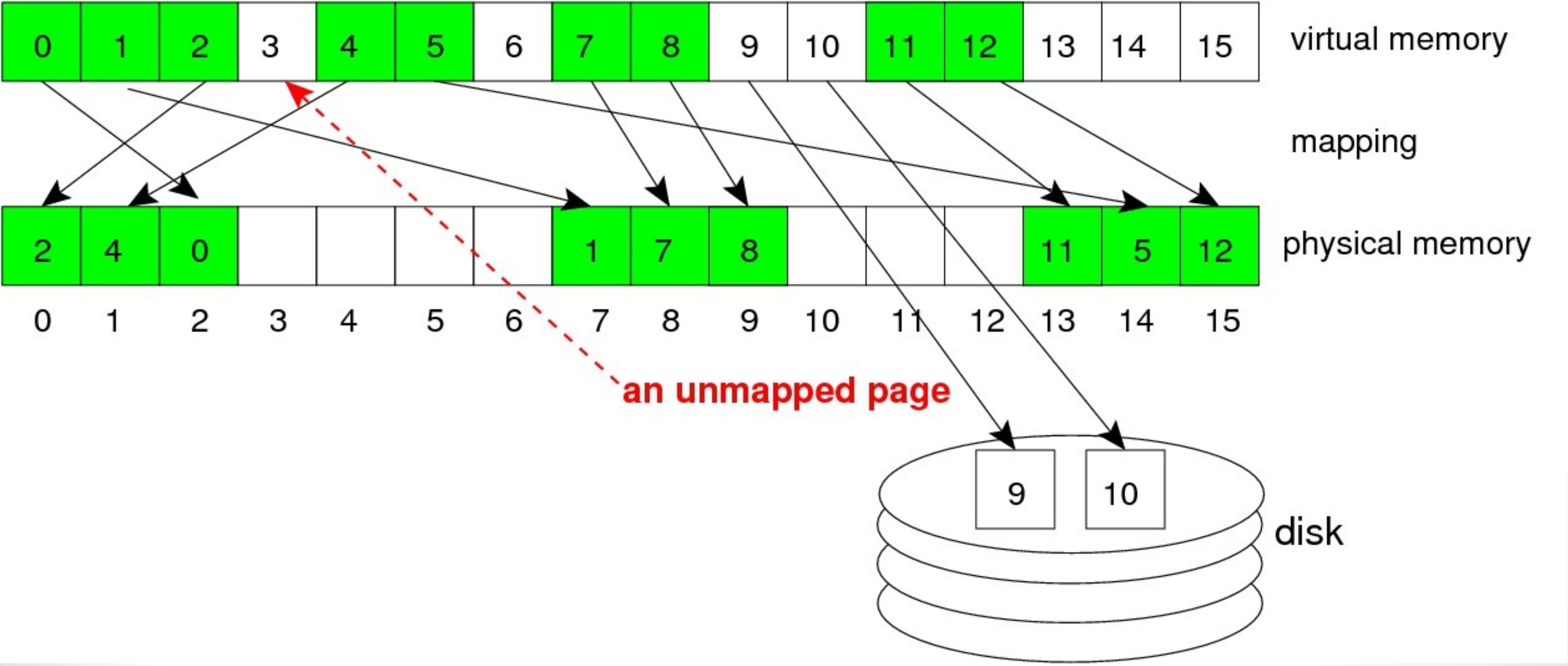
Unmapped Pages

- The mapping is sparse. Some pages are unmapped.



Unmapped Pages

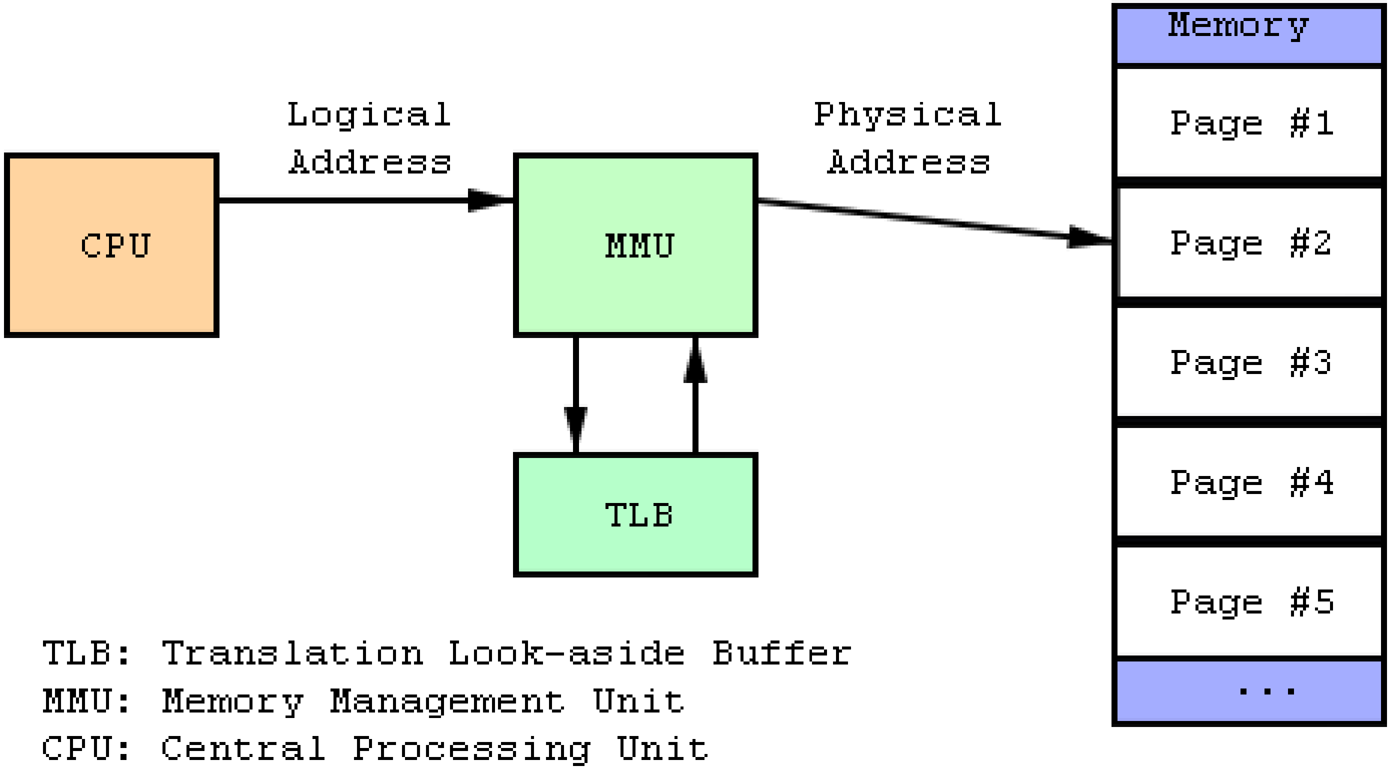
- Pages may be mapped to locations on devices and others to both



MMU Function

- MMU translates virtual page numbers to physical page numbers via *Translation Lookaside Buffer (TLB)*
- If TLB lacks translation, slower mechanism is used with page tables
- The physical page number is combined with the page offset to give the complete physical address

MMU Function



TLB: Translation Look-aside Buffer
MMU: Memory Management Unit
CPU: Central Processing Unit

MMU Function

- Computes address translation
- Uses special associative cache (TLB) to speed up translation
- Falls back on full page translation tables, in memory, if TLB misses
- Falls back to OS if page translation table misses
  - Such a reference to an unmapped address causes a page fault

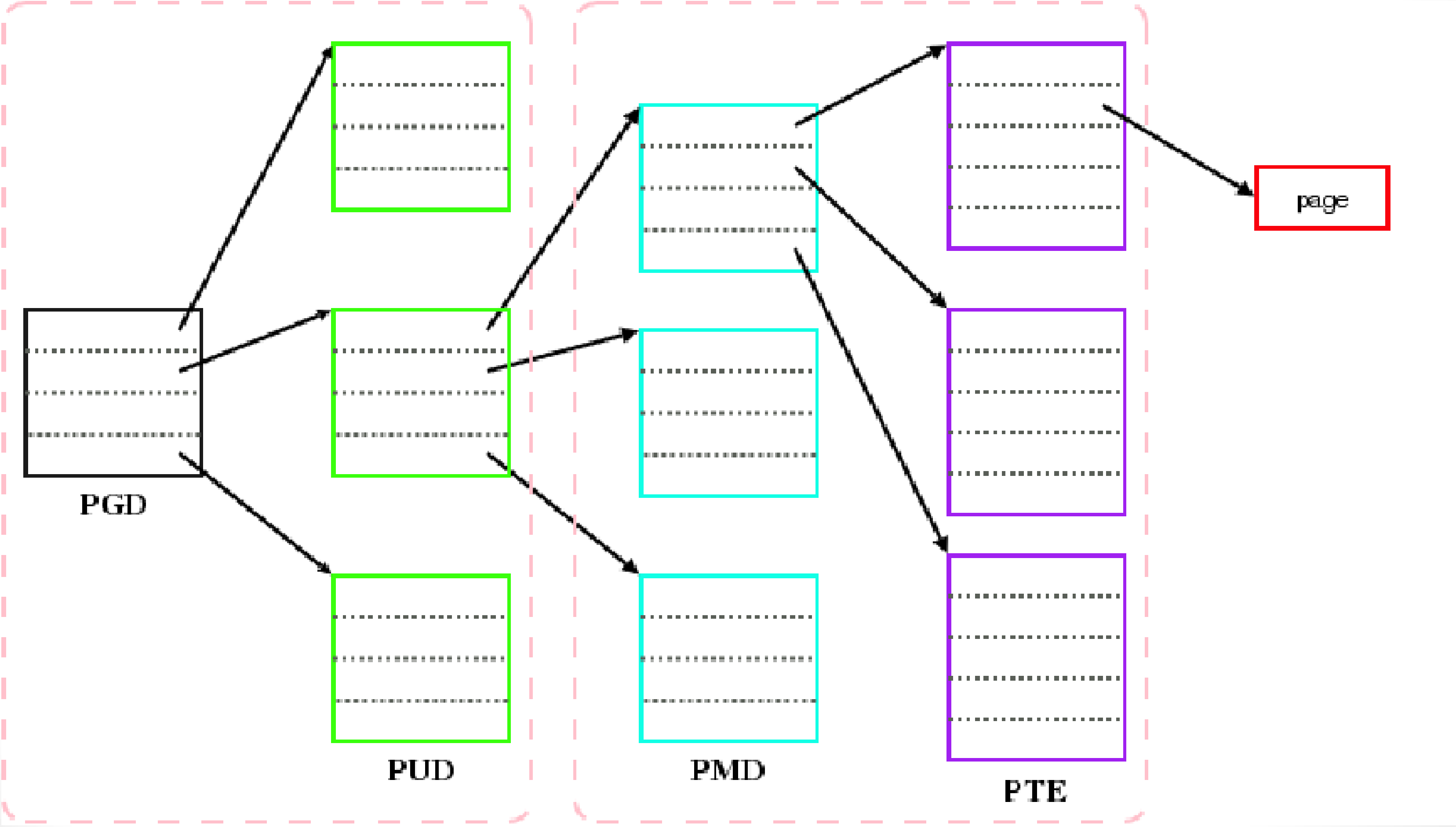
MMU Function

- If page fault caused by a CPU (MMU)
  - Enters the OS through a trap handler
- If page fault caused by an I/O device (IOMMU)
  - Enters the OS through an interrupt handler
- What reasons could cause a page fault?

Possible Handler Actions

1. Map the page to a valid physical memory location
    - May require creating a page table entry
    - May require bringing data in to memory from a device
2. Treat the event as an erro (e.g., SIG_SEGV)
3. Pass the exception on to a device-specific handler
    - The device's fault method

Linux Page Tables 4-levels

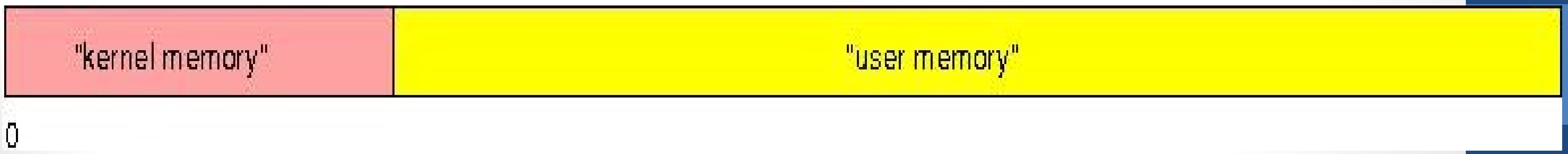


Linux Page Tables

- Logically, Linux now has four levels of page tables:
  - PGD - top level, array of pgd_t items
  - PUD - array of pud_t items
  - PMD - array of pmd_t items
  - PTE - bottom level, array of pte_t items
- On architectures that do not require all four levels, inner levels may be collapsed
- Page table lookups (and address translation) are done by the hardware (MMU) so long as the page is mapped and resident
- Kernel is responsible for setting the tables up and handling page faults
- Table are located in struct mm object for each process

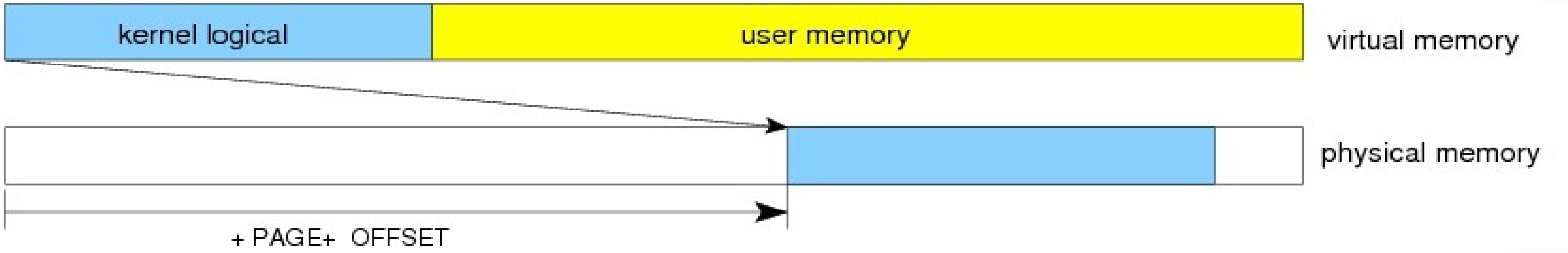
Kernel Memory Mapping

- Each OS process has its own memory mapping
- Part of each virtual address space is reserved for the kernel
- This is the same range for every process
- So, when a process traps into the kernel, there is no change of page mappings
- This is called "kernel memory"
- The mapping of the rest of the virtual address range varies from one process to another



Kernel Logical Addresses

- Most of the kernel memory is mapped linearly onto physical addresses
- Virtual addresses in this range are called kernel logical addresses



- Examples of PAGE_OFFSET values:
  - 64-bit X86: 0xffffffff80000000
  - ARM & 32-bit X86: CONFIG_PAGE_OFFSET
  - default on most architectures = 0xc0000000

Kernel Logical Addresses

- In user mode, the process may only access addresses less than 0xc0000000
  - Any access to an address higher than this causes a fault
- However, when user-mode process begins executing in the kernel (e.g. system call)
  - Protection bit in CPU changed to supervisor mode
  - Process can access addresses above 0xc0000000

Kernel Logical Addresses

- Mapped using page table by MMU, like user virtual addresses
- But mapped linearly 1:1 to contiguous physical addresses
- __pa(x) adds PAGE_OFFSET to get physical address associated with virtual address
- __va(x) subtracts PAGE_OFFSET to get virtual address associated with physical address
- All memory allocated by kmalloc() with GFP_KERNEL fall into this category

Page Size Symbolic Constants

- PAGE_SIZE
  - value varies across architectures and kernel configurations
  - code should never use a hard-coded integer literal like 4096
- PAGE_SHIFT
  - the number of bits to right shift to convert virtual address to page number
  - and physical address to page frame number

struct page

- Describes a page of physical memory.
- One exists for each physical memory page
- Pointer to struct page can be used to refer to a physical page
- members:
  - atomic_t count = number of references to this page
  - void * virtual = virtual address of the page, if it is mapped (in the kernel memory space) / otherwise NULL
  - flags = bits describing status of page
    - PG_locked - (temporarily) locked into real memory (can't be swapped out)
    - PG_reserved - memory management system "cannot work on the page at all"
    - ... and others

struct page pointers ↔ virtual addresses

- struct page *virt_to_page(void *kaddr);
  - Given a kernel logical address, returns associated struct page pointer
- struct page *pfn_to_page(int pfn);
  - Given a page frame number, returns the associated struct page pointer
- void *page_address(struct page *page);
  - Returns the kernel virtual address, if exists.

kmap() and kunmap()

- kmap is like page_address(), but creates a "special" mapping into kernel virtual memory if the physical page is in high memory
  - there are a limited number of such mappings possible at one time
  - may sleep if no mapping is currently available
  - not needed for 64-bit model
- kunmap() - undoes mapping created by kmap()
  - Reference-count semantics

Some Page Table Operations

- *pgd_val()* - fetches the unsigned value of a PGD entry
- *pmd_val()* - fetches the unsigned value of a PMD entry
- *pte_val()* - fetches the unsigned value of PTE
- *mm_struct* - per-process structure, containing page tables and other MM info

Some Page Table Operations

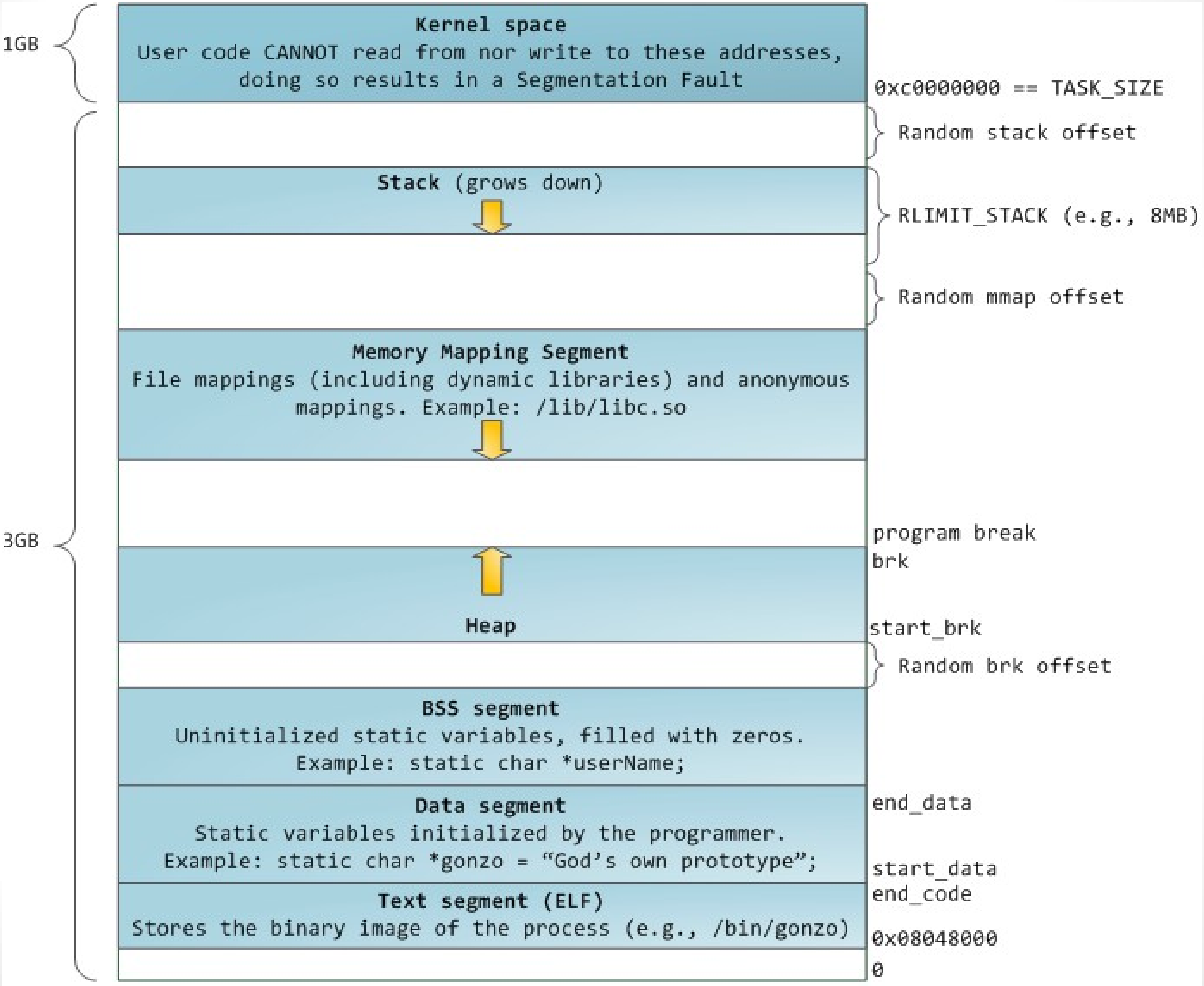
- *pgd_offset()* - pointer to the PGD entry of an address, given a pointer to the specified *mm_struct*
- *pmd_offset()* - pointer to the PMD entry of an address, given a pointer to the specified PGD entry
- *pte_page()* - pointer to the *struct page()* entry corresponding to a PTE
- *pte_present()* - whether PTE describes a page that is currently resident

Some Page Table Operations

- Device drivers should not need to use these functions because of the generic memory mapping services described next

Virtual Memory Areas

- A range of contiguous VM is represented by an object of type struct vm_area_struct.
- Used by kernel to keep track of memory mappings of processes
- Each is a contract to handle the VMem→PMem mapping for a given range of addresses
- Some kinds of areas:
    - Stack, memory mapping segment, heap, BSS, data, text



1GB

Kernel space

User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault

`0xc0000000 == TASK_SIZE`

Random stack offset

Stack (grows down)

RLIMIT_STACK (e.g., 8MB)

Random mmap offset

Memory Mapping Segment

File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

3GB

program break

brk

Heap

start_brk

Random brk offset

BSS segment

Uninitialized static variables, filled with zeros.

Example: static char *userName;

end_data

Data segment

Static variables initialized by the programmer.

Example: static char *gonzo = “God’s own prototype”;

start_data

end_code

Text segment (ELF)

Stores the binary image of the process (e.g., /bin/gonzo)

0x08048000

0

Virtual Memory Regions

- Stack segment
  - Local variable and function parameters
  - Will dynamically grow to a certain limit
  - Each thread in a process gets its own stack
- Memory mapping segment
  - Allocated through mmap()
  - Maps contents of file directly to memory
    - Fast way to do I/O
  - Anonymous memory mapping does not correspond to any files
    - Malloc() may use this type of memory if requested area large enough

Virtual Memory Segments

- Heap
  - Meant for data that must outlive the function doing the allocation
  - If size under MMAP_THRESHOLD bytes, malloc() and friends allocate memory here
- BSS
  - "block started by symbol“
  - Stores uninitialized static variables
  - Anonymous (not file-backed)

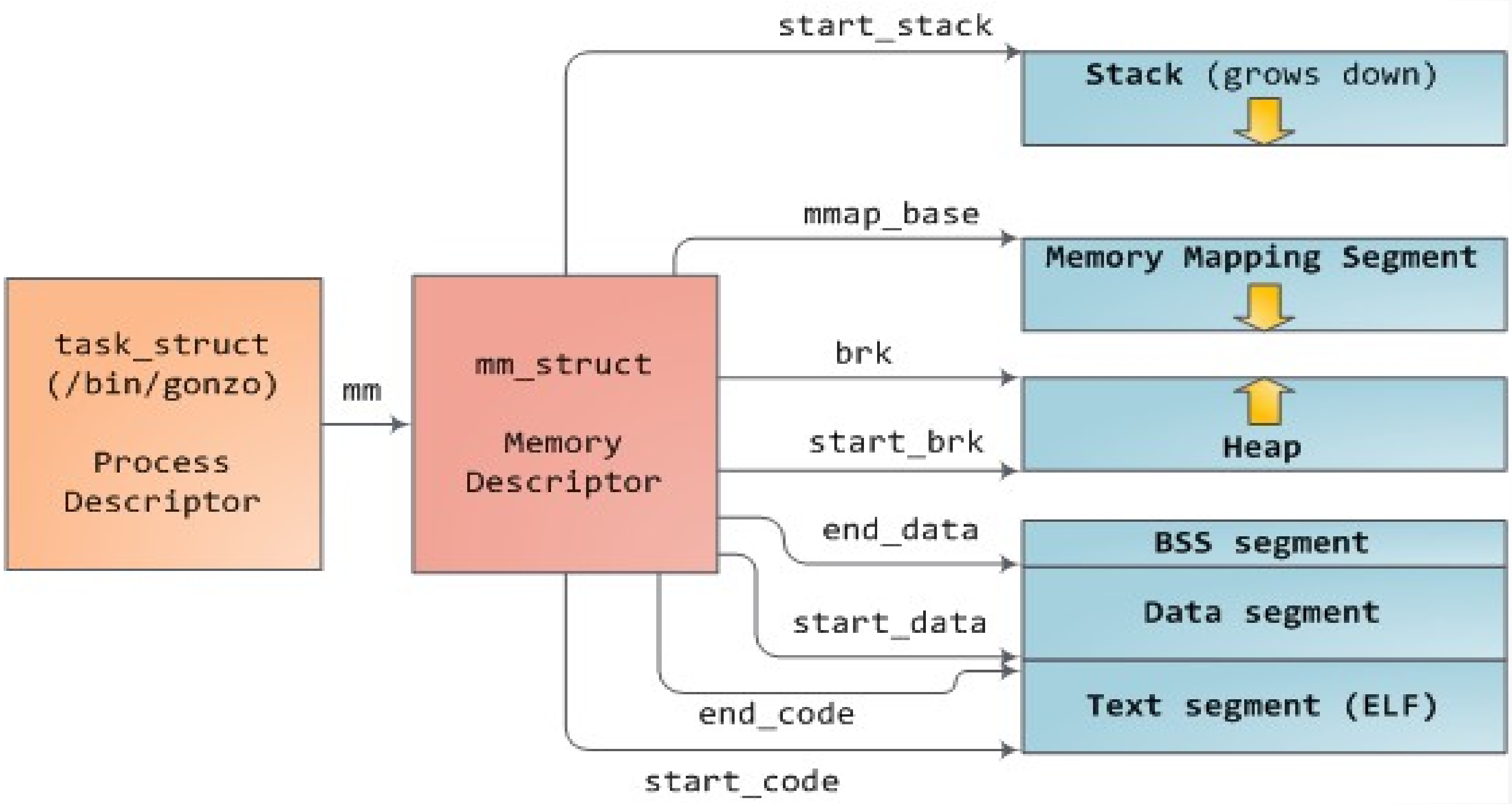
Virtual Memory Segments

- Data
  - Stores static variables initialized in source code
  - Not anonymous
- Text
  - Read-only
  - Stores code
  - Maps binary file in memory

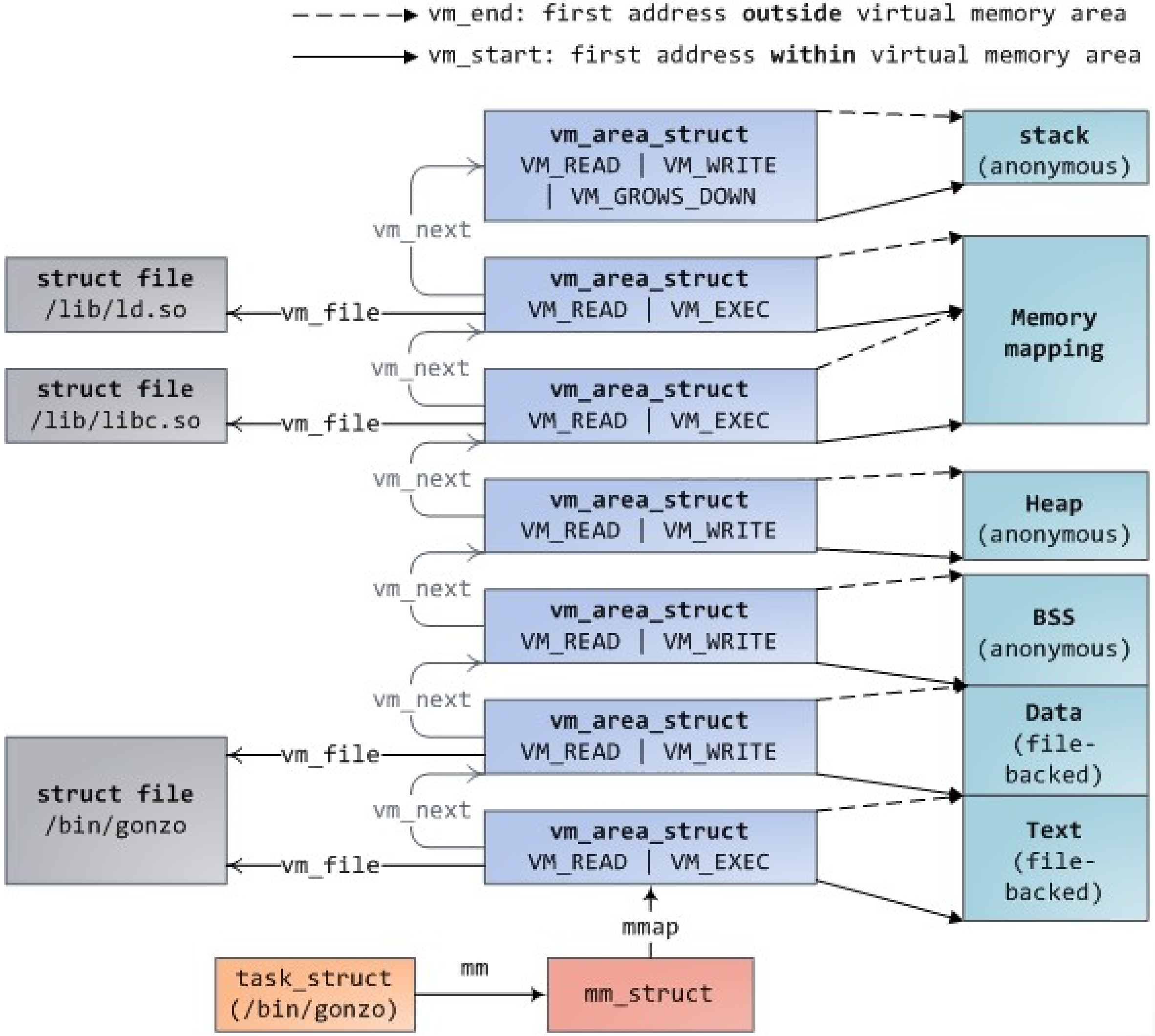
Process Memory Map

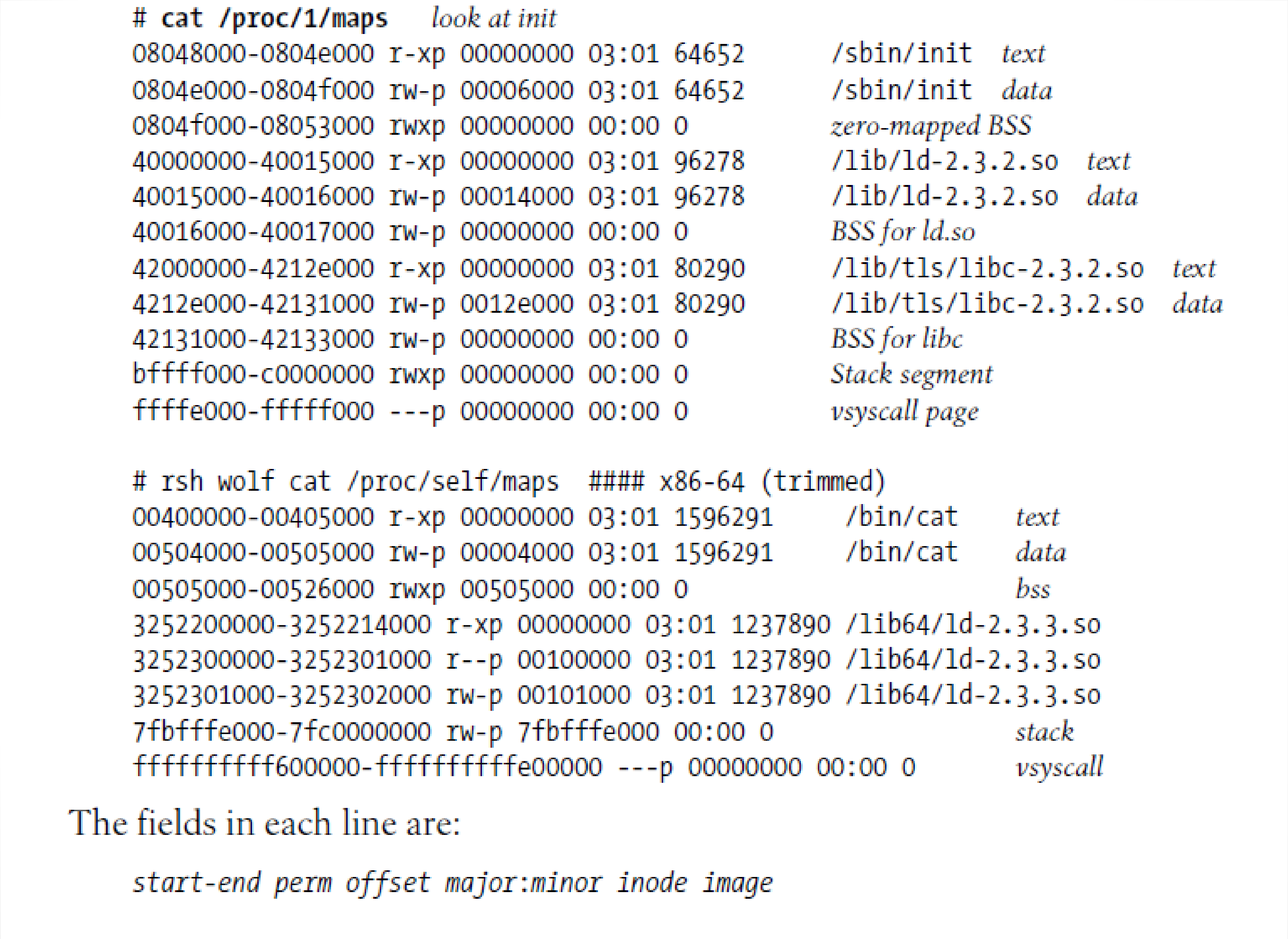
- struct mm_struct - contains list of process' VMAs, page tables, etc.
- accessible via current-> mm
- The threads of a process share one struct mm_struct object

Virtual Memory Regions



Virtual Memory Area Mapping Descriptors





```
# cat /proc/1/maps    look at init
08048000-0804e000 r-xp 00000000 03:01 64652       /sbin/init   text
0804e000-0804f000 rw-p 00006000 03:01 64652       /sbin/init   data
0804f000-08053000 rwxp 00000000 00:00 0           zero-mapped BSS
40000000-40015000 r-xp 00000000 03:01 96278       /lib/ld-2.3.2.so   text
40015000-40016000 rw-p 00014000 03:01 96278       /lib/ld-2.3.2.so   data
40016000-40017000 rw-p 00000000 00:00 0           BSS for ld.so
42000000-4212e000 r-xp 00000000 03:01 80290       /lib/tls/libc-2.3.2.so   text
4212e000-42131000 rw-p 0012e000 03:01 80290       /lib/tls/libc-2.3.2.so   data
42131000-42133000 rw-p 00000000 00:00 0           BSS for libc
bffff000-c0000000 rwxp 00000000 00:00 0           Stack segment
ffffe000-fffff000 ---p 00000000 00:00 0           vsyscall page

# rsh wolf cat /proc/self/maps  #### x86-64 (trimmed)
00400000-00405000 r-xp 00000000 03:01 1596291     /bin/cat     text
00504000-00505000 rw-p 00004000 03:01 1596291     /bin/cat     data
00505000-00526000 rwxp 00505000 00:00 0                        bss
3252200000-3252214000 r-xp 00000000 03:01 1237890 /lib64/ld-2.3.3.so
3252300000-3252301000 r--p 00100000 03:01 1237890 /lib64/ld-2.3.3.so
3252301000-3252302000 rw-p 00101000 03:01 1237890 /lib64/ld-2.3.3.so
7fbfffe000-7fc0000000 rw-p 7fbfffe000 00:00 0                  stack
ffffffffff600000-ffffffffffe00000 ---p 00000000 00:00 0        vsyscall
```

The fields in each line are:

```
start-end perm offset major:minor inode image
```

struct vm_area_struct

- Represents how a region of virtual memory is mapped
- Members include:
  - vm_start, vm_end - limits of VMA in virtual address space
  - vm_page_prot - permissions (p = private, s = shared)
  - vm_pgoff - of memory area in the file (if any) mapped

struct vm_area_struct

- vm_file - the struct file (if any) mapped
- provides (indirect) access to:
  - major, minor - device of the file
  - inode - inode of the file
  - image - name of the file
- vm_flags - describe the area, e.g.,
  - VM_IO - memory-mapped I/O region will not be included in core dump
  - VM_RESERVED - cannot be swapped
- vm_ops - dispatching vector of functions/methods on this object
- vm_private_data - may be used by the driver

vm_operations_struct.vm_ops

- void *open (struct vm_area_struct *area);
  - allows initialization, adjusting reference counts, etc.;
  - invoked only for additional references, after mmap(), like fork()
- void *close (struct vm_area_struct *area);
  - allows cleanup when area is destroyed;
  - each process opens and closes exactly once
- int fault (struct vm_area_struct *vma, struct vm_fault *vmf);
  - general page fault handler;

Uses of Memory Mapping by Device Drivers

- A device driver is likely to use memory mapping for two main purposes:
  1. To provide user-level access to device memory and/or control registers
     - For example, so an Xserver process can access the graphics controller directly
  2. To share access between user and device/kernel I/O buffers, to avoid copying between DMA/kernel buffers and userspace

The mmap() Interfaces

- User-level API function:
  - void *mmap (caddr_t start, size_t len, int prot, int flags, int fd, off_t offset);
- Driver-level file operation:
  - int (*mmap) (struct file *filp, struct vm_area_struct *vma);

Implementing the mmap() Method in a Driver

1. Build suitable page tables for the address range two ways:
   a) Right away, using remap_pfn_range or vm_insert_page
   b) Later (on demand), using the fault() VMA method
2. Replace vma->vm_ops with a new set of operations, if necessary

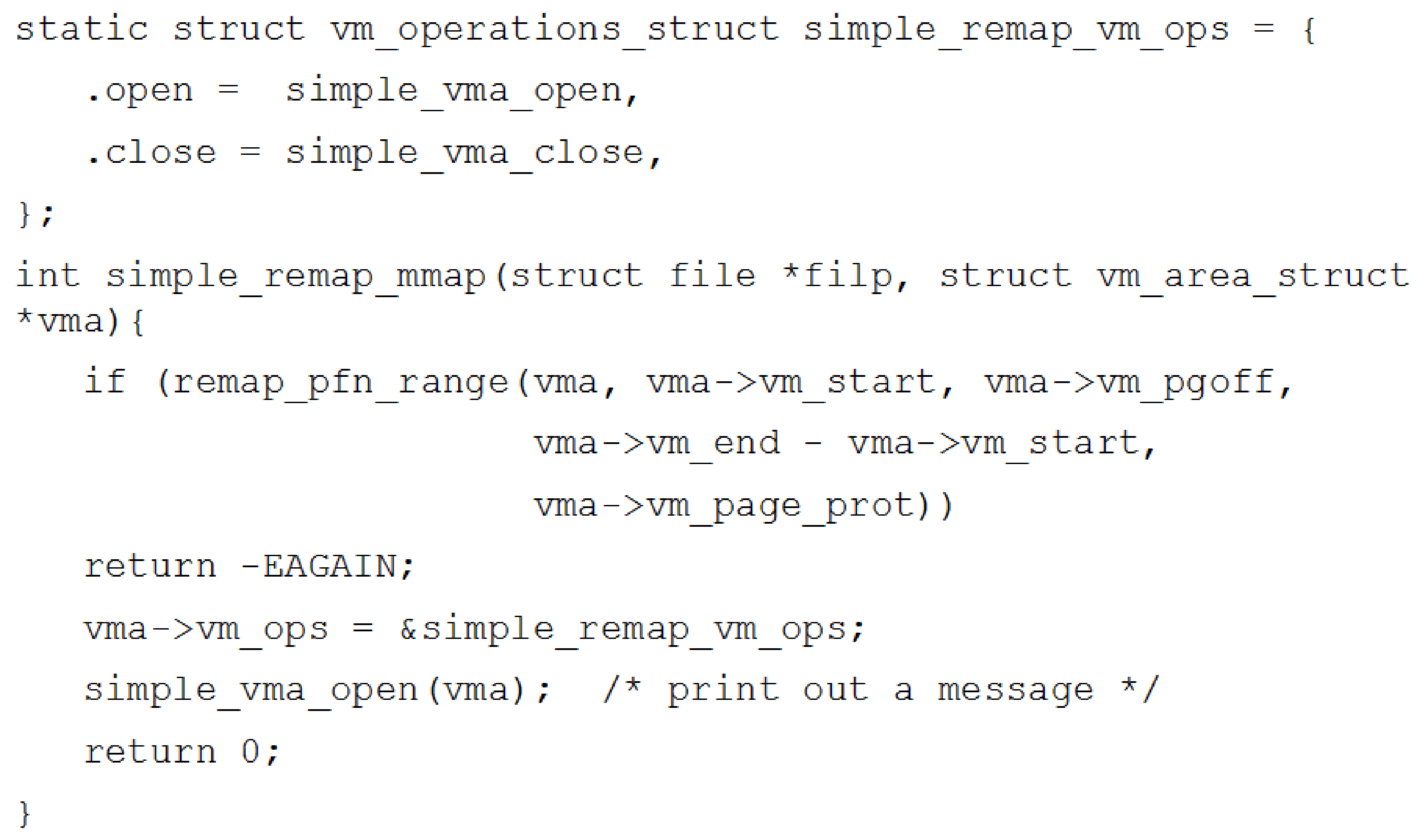
The remap_pfn_range() Kernel Function

- Use to remap to system RAM
  - int remap_pfn_range (struct vm_area_struct *vma, unsigned long addr, unsigned long pfn, unsigned long size, pgprot_t prot);
- Use to remap to I/O memory
  - int io_remap_pfn_range(struct vm_area_struct *vma, unsigned long addr ,unsigned long phys_addr, unsigned long size, pgprot_t prot);

The remap_pfn_range() Kernel Function

- vma = virtual memory are to which the page range is being mapped
- addr = target user virtual address to start at
- pfn = target page frame number of physical address to which mapped
  - normally vma->vm_pgoff>>PAGE_SHIFT
  - mapping targets range (pfn<<PAGE_SHIFT) .. (pfn<<PAGE_SHIFT)+size
- prot = protection
  - normally the same value as found in vma->vm_page_prot
  - may need to modify value to disable caching if this is I/O memory

The remap_pfn_range() Kernel Function



```
static struct vm_operations_struct simple_remap_vm_ops = {
   .open =  simple_vma_open,
   .close = simple_vma_close,
};
int simple_remap_mmap(struct file *filp, struct vm_area_struct
*vma){
   if (remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff,
                       vma->vm_end - vma->vm_start,
                       vma->vm_page_prot))
   return -EAGAIN;
   vma->vm_ops = &simple_remap_vm_ops;
   simple_vma_open(vma);  /* print out a message */
   return 0;
}
```

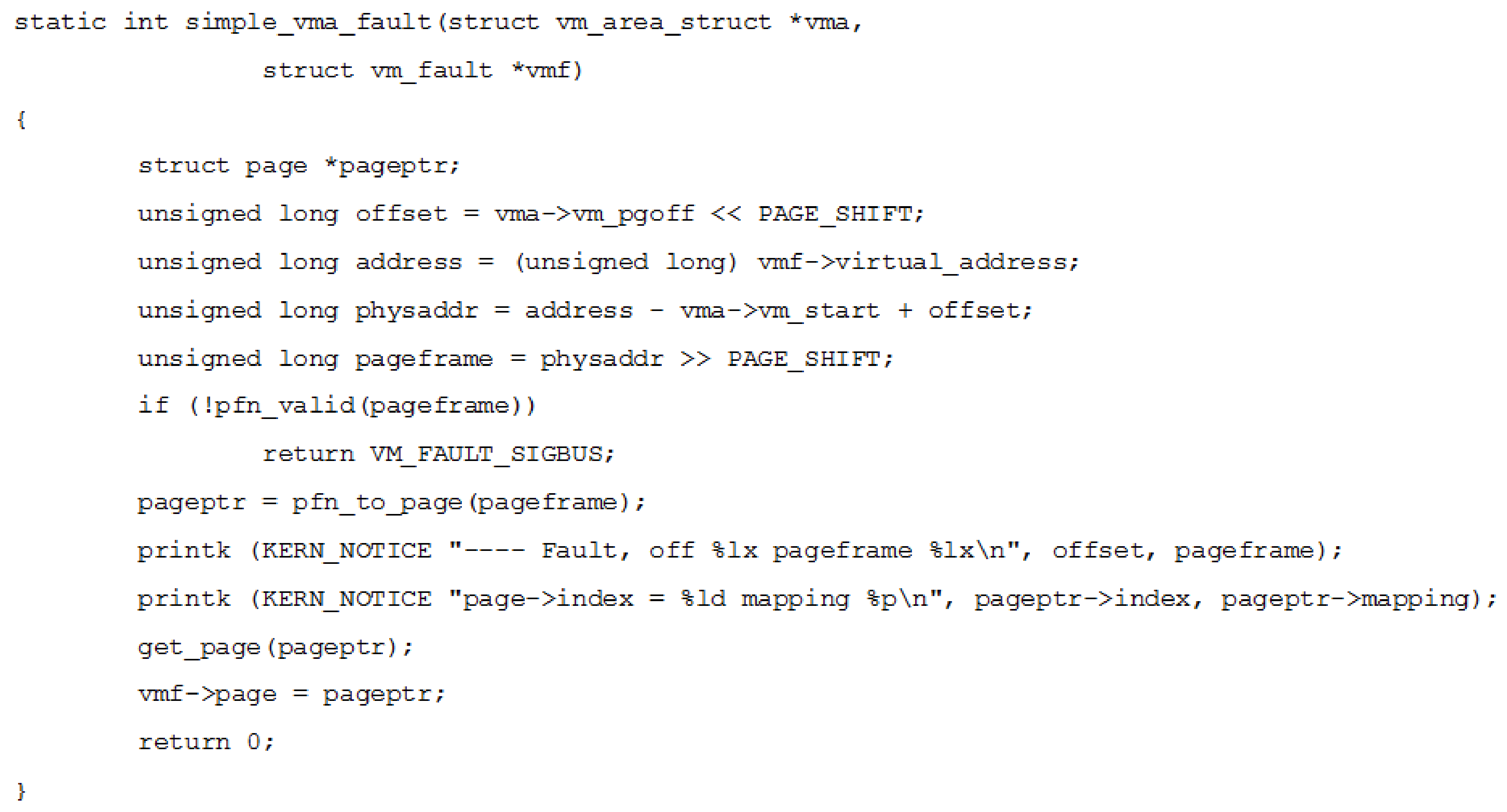
Using fault()

- LDD3 discusses a nopage() function that is no longer in the kernel
  - Race conditions
- Replaced by fault()
- http://lwn.net/Articles/242625/

Using fault()

- struct page (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);
- vmf - is a struct vm_fault, which includes:
  - flags
    - FAULT_FLAG_WRITE indicates the fault was a write access
    - FAULT_FLAG_NONLINEAR indicates the fault was via a nonlinear mapping
  - pgoff - logical page offset, based on vma
  - virtual_address - faulting virtual address
  - page - set by fault handler to point to a valid page descriptor; ignored if VM_FAULT_NOPAGE or VM_FAULT_ERROR is set

Using fault()



```
static int simple_vma_fault(struct vm_area_struct *vma,
                struct vm_fault *vmf)
{
        struct page *pageptr;
        unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
        unsigned long address = (unsigned long) vmf->virtual_address;
        unsigned long physaddr = address - vma->vm_start + offset;
        unsigned long pageframe = physaddr >> PAGE_SHIFT;
        if (!pfn_valid(pageframe))
                return VM_FAULT_SIGBUS;
        pageptr = pfn_to_page(pageframe);
        printk (KERN_NOTICE "---- Fault, off %lx pageframe %lx\n", offset, pageframe);
        printk (KERN_NOTICE "page->index = %ld mapping %p\n", pageptr->index, pageptr->mapping);
        get_page(pageptr);
        vmf->page = pageptr;
        return 0;
}
```

A Slightly More Complete Example

- See ldd3/sculld/mmap.c
- http://www.cs.fsu.edu/~baker/devices/notes/sculld/mmap.c

Remapping I/O Memory

- remap_pfn_to_page() cannot be used to map addresses returned by ioremap() to user space
- instead, use io_remap_pfn_range() directly to remap the I/O areas into user space