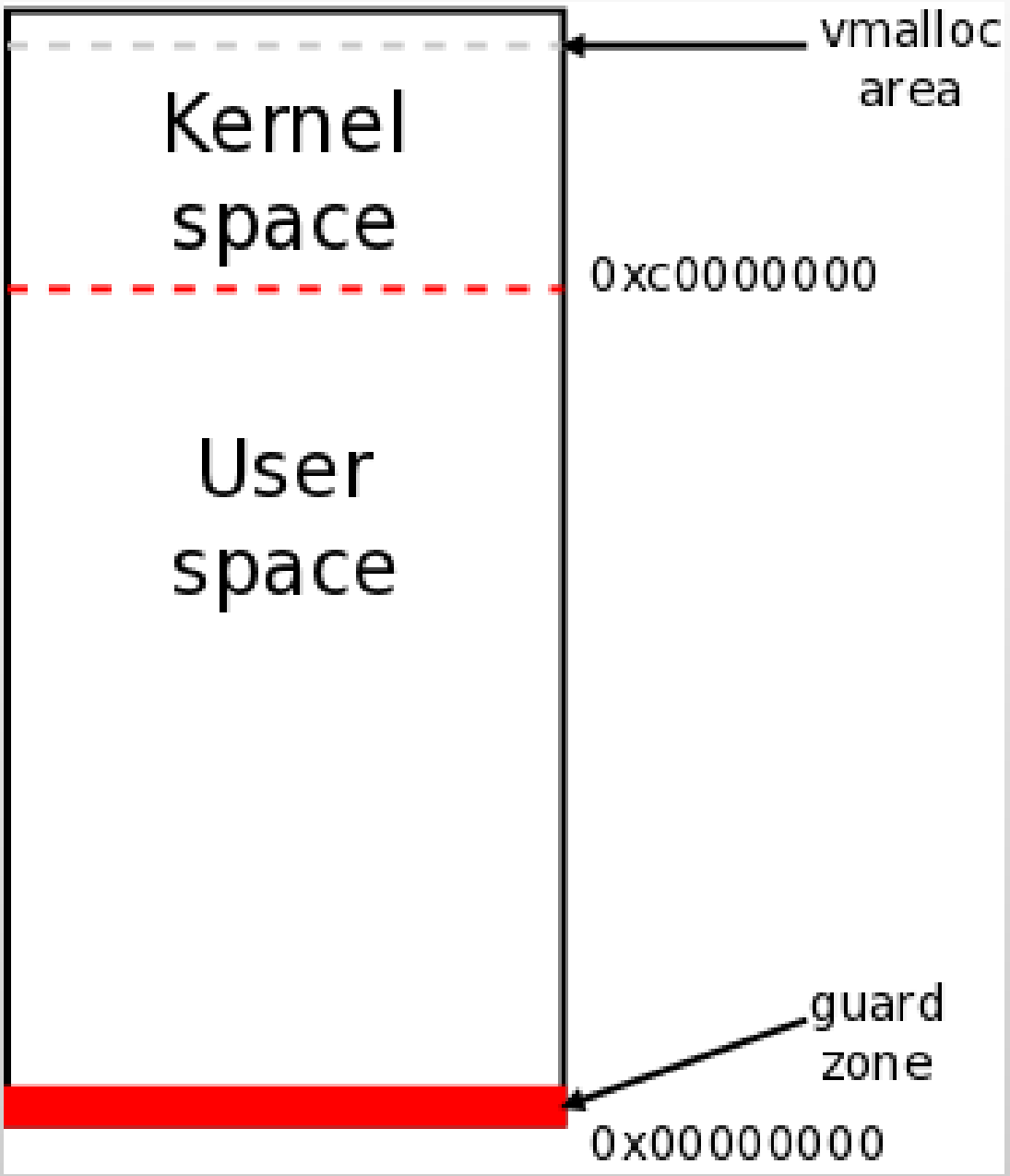
Process memory management

Typical 32-bit virtual address space layout



Implications

- Kernel space is always mapped
  - Inaccessible from user mode
- User space is mapped in kernel mode
  - But not directly accessible
- Limits:
  - Kernel: 1GB virtual address space
  - User: 3GB virtual address space

An aside

/proc/sys/vm/mmap_min_addr

- The lowest address a process can map
- Should not be zero!

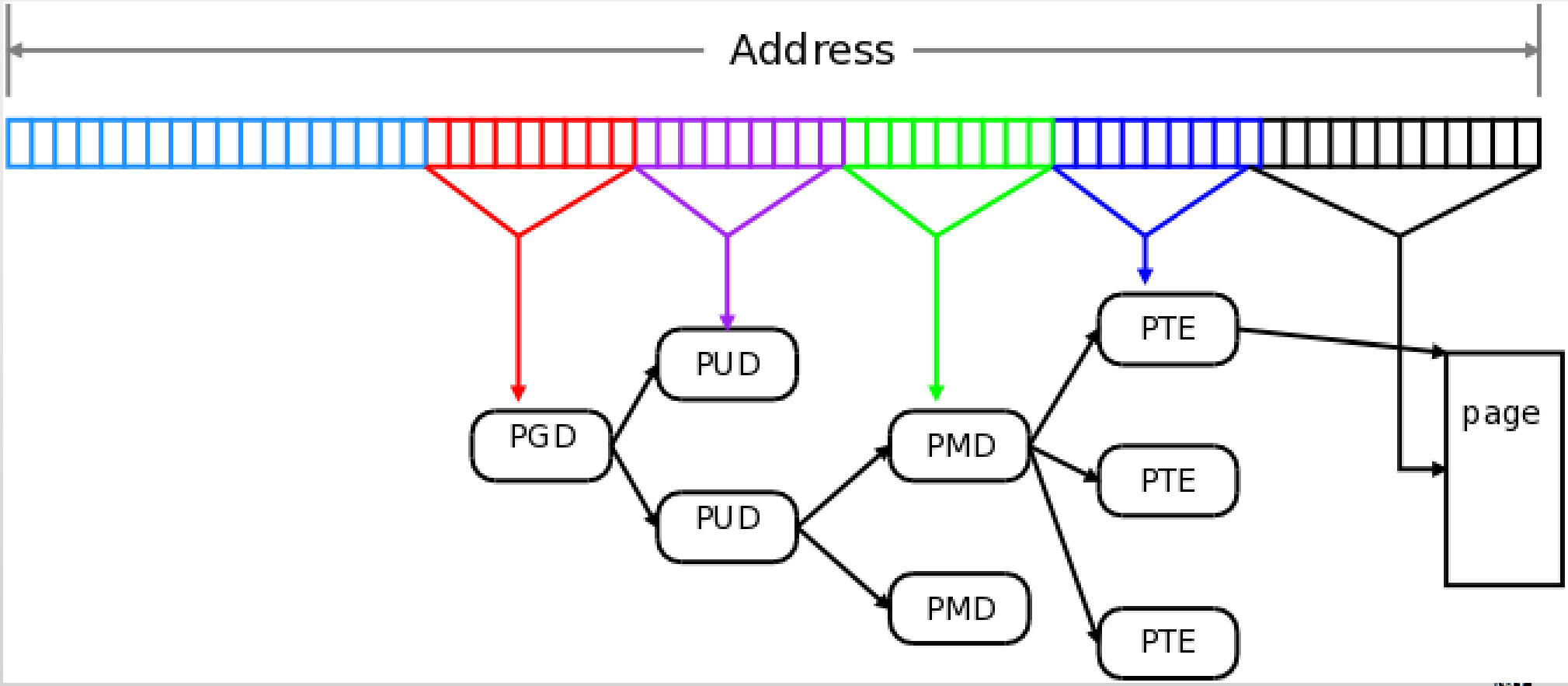
struct mm_struct

The core process VM data structure
- Pointed to from task_struct
- Defined in <linux/mm_types.h>

What's found here
- Page table pointer
- VMA list
- Memory use stats
- AIO context info
- OOM killer information
- ...

Page tables

64-bit x86 page table layout



User-space memory is complex

It contains:

- Executable text areas
- Heap data areas
- Named data areas
- Stacks
- Device memory mappings
- ...

All of these may be shared

Page tables

...implement these areas

- Convert virtual to physical addresses
- Contain access permissions
- Mark non-present pages

More is needed, though.

Virtual memory areas

A VMA describes a memory area

- Address range
- Access permissions
- Backing store
- Access operations
- ...

A simple program

```
#include <unistd.h>
#include <stdio.h>

main(int artc, char **argv)
{
     printf ("Hello\n");
     pause();
}
```

Its VMAs

```
002c8000-002c9000 r-xp 00000000 00:00 0          [vdso]
008ac000-008cc000 r-xp 00000000 08:01 260681     /lib/ld-2.12.90.so
008cc000-008cd000 r--p 0001f000 08:01 260681     /lib/ld-2.12.90.so
008cd000-008ce000 rw-p 00020000 08:01 260681     /lib/ld-2.12.90.so
008d0000-00a5d000 r-xp 00000000 08:01 260688     /lib/libc-2.12.90.so
00a5d000-00a5e000 ---p 0018d000 08:01 260688     /lib/libc-2.12.90.so
00a5e000-00a60000 r--p 0018d000 08:01 260688     /lib/libc-2.12.90.so
00a60000-00a61000 rw-p 0018f000 08:01 260688     /lib/libc-2.12.90.so
00a61000-00a64000 rw-p 00000000 00:00 0
08048000-08049000 r-xp 00000000 08:03 2114849    pause
08049000-0804a000 rw-p 00000000 08:03 2114849    pause
b77c1000-b77c2000 rw-p 00000000 00:00 0
b77de000-b77e0000 rw-p 00000000 00:00 0
bfb8b000-bfbac000 rw-p 00000000 00:00 0          [stack]
```

(As seen in /proc/pid/maps)

struct vm_area_struct

(<linux/mm_types.h>)

```
struct vm_area_struct {
     struct mm_struct *vm_mm;
     unsigned long vm_start, vm_end;
     struct vm_area_struct *vm_next, *vm_prev;
     pgprot_t vm_page_prot;
     unsigned long vm_flags;
     struct file *vm_file;
     struct vm_operations_struct *vm_ops;
     /* ... */
};
```

struct vm_operations_struct

(<linux/mm.h>)
Defined by anything that manages VMAs

```
struct vm_operations_struct {
    void (*open) (struct vm_area_struct *area);
    void (*close) (struct vm_area_struct *area);
    int (*fault) (struct vm_fault *vmf);
    /* ... */
};
```

How VMAs are made

A call to mmap() with:

- An ordinary file
- /dev/zero
- A device special file
- A kernel virtual file (perf, for example)

The kernel will:

- Create a new VMA
- Find the right mmap() function
- Pass the VMA to it

New VMA, option 1

If the VMA refers to device memory

The mmap() function need only make PTEs

To do that:

```
remap_pfn_range(struct vm_area_struct *vma,
                unsigned long start,
                unsigned long start_pfn,
                unsigned long size,
                pgprot_t prot);
```

New VMA, option 2

Remember the relevant information
Store the vm_operations_struct pointer
    (in the VMA)

Use open() and close() to track usage

Use fault() to map each page on first reference

Handling faults

```
static int videobuf_vm_fault(struct vm_area_struct *vma,
                             struct vm_fault *vmf)
{
      struct page *page;

      page = alloc_page(GFP_USER | __GFP_DMA32);
      if (!page)
            return VM_FAULT_OOM;
      clear_user_highpage(page,
                          (unsigned long)vmf->virtual_address);
      vmf->page = page;
      return 0;
}
```

Page fault handling (simplified)

Find mm_struct from task_struct
Look up the VMA for the faulting address
    (an rbtree exists for that purpose)
Check protections
Call the associated fault() method
    Set the PTE entry accordingly

Relevant system calls

execve()
mmap()
munmap()
mprotect()
mlock()
mlockall()
brk()
shmat()

Accessing user-space memory

Do not dereference user-space pointers

- Even if it “just works” much of the time

Why?

- Pages might not be present
  - (Instant kernel oops)
- Page protections might be bypassed

Moving user-space data

```
int access_ok(int type, void __user *addr,
              unsigned long len);
/* This is a weak (but necessary) check! */

unsigned long copy_from_user(void *dest,
                const void __user *source,
                unsigned long len);

unsigned long copy_to_user(void __user *dest,
                const void *source,
                unsigned long len);

/* Return value is number of UNCOPIED bytes */

/* Lots of variants, naturally */
```

get_user_pages()

To pin user-space memory:

```
int get_user_pages(struct task_struct *task,
            struct mm_struct *mm,
            unsigned long start,
            int nr_pages,
            int write,
            int force_write,
            struct page **pages,
            struct vm_area_struct **vmas);

/* Returns number of pages */
```

Questions?