Linux kernel module memory and user-space interaction

Kathryn Hampton

Advanced Linux Kernel

June 21, 2014

Topics

- Recap: Linux memory
- vmalloc()
- kmalloc()
- mmap()
- mmap: kmalloc
- mmap: vmalloc, 3 ways
- module access to user space
- Appendices (a.k.a. ‘stuff’)

Note that where there are topics that have any processor dependencies this discussion assumes x86 with virtually addressed TLB and physically addressed caches

Simplistic recap: Linux memory

- There are three addressing modes:
  - Logical: ‘what you use in asm code’
    - On x86, segment:offset, where ‘segment’ is a hw addressing component
    - Linux uses the ‘long flat’ model where segments are not used for address separation/generation
  - Linear: a.k.a virtual, what Linux uses to do business
  - Physical: what goes out on the memory bus to select chips
- Physical memory is carved into ‘page’-size frames
  - The page descriptor structure (struct page) has information about the state of each frame (mm_types.h)
  - Frames are numbered from 0
    - n * (page-size) == (n << PAGE_SHIFT) == paddr of the frame
  - page-frame-numbers (pfn’s) are indices into the array of struct page
- The page frames and descriptors that describe them are ‘always’ there, but may or may not be addressable/accessible from running code

*Physical = Linear on a system with no mmu

Recap continued

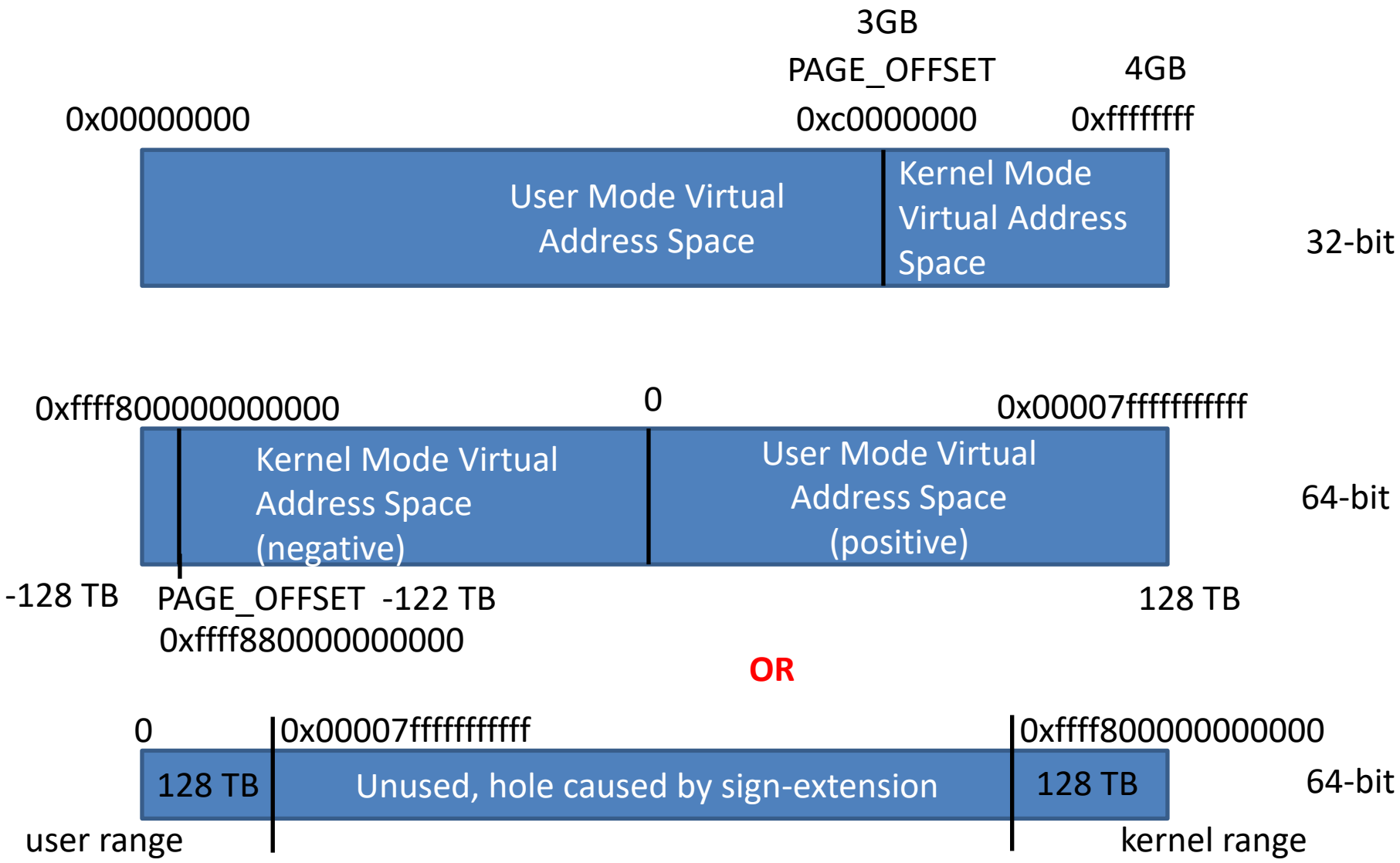
- Virtual addresses are only meaningful/usable when they have page frames attached to them.
  - This process is called ‘mapping’ and the associations are maintained in page tables
- Linux partitions virtual address space into user-space and kernel-space ranges
  - Page tables don’t change when trapping from user mode to kernel mode
  - Kernel addresses map to the same page frames in all processes, whereas the same user space virtual addresses may map to different page frames in every process
  - If the overall address space is small, the kernel range is limited so that the user range can be larger
    - For a 32-bit system the defaults are 3GB for user space and 1 GB for kernel space (0xc00000000-0xffffffff)

Kernel address space

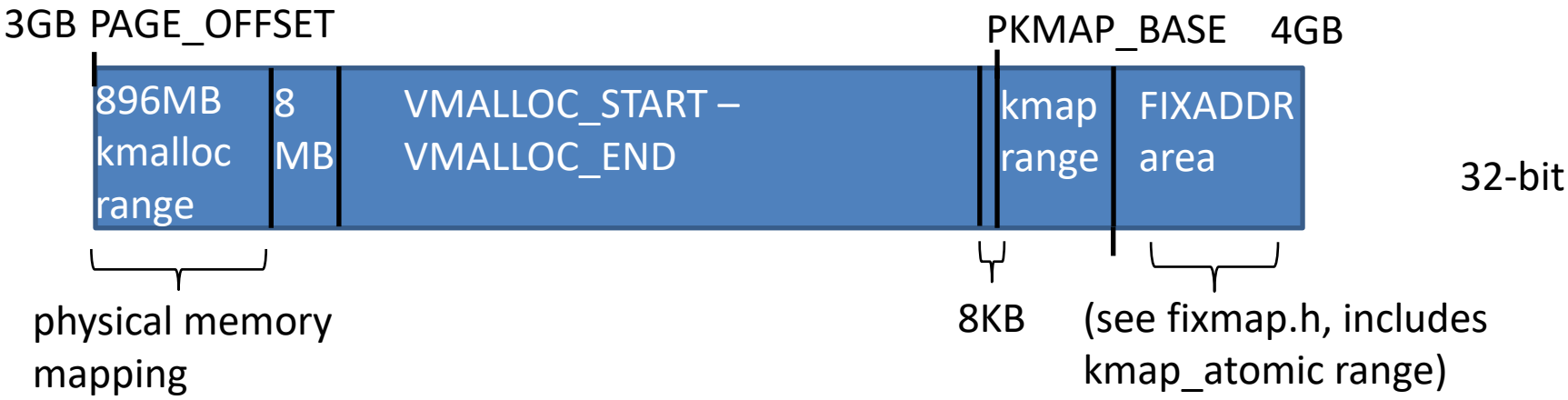
- Linux permanently maps some contiguous part of the kernel virtual address space to a contiguous range of physical addresses
  - Generally 896MB for 32-bit
    - Page frames beyond that range are considered ‘high memory’
  - For 32-bit, paddr = vaddr – offset*, vaddr = paddr + offset. Minor changes for 64-bit
- The rest of the kernel address space range is used for addressing high memory, device memory, …
  - vmalloc() and kmap() addresses are allocated from those address ranges
  - These addresses may be limited and have to be reused whenever the kernel needs to access unmapped page frames

*PAGE_OFFSET, 0xc000000 for 32-bit

Kernel/User space address division

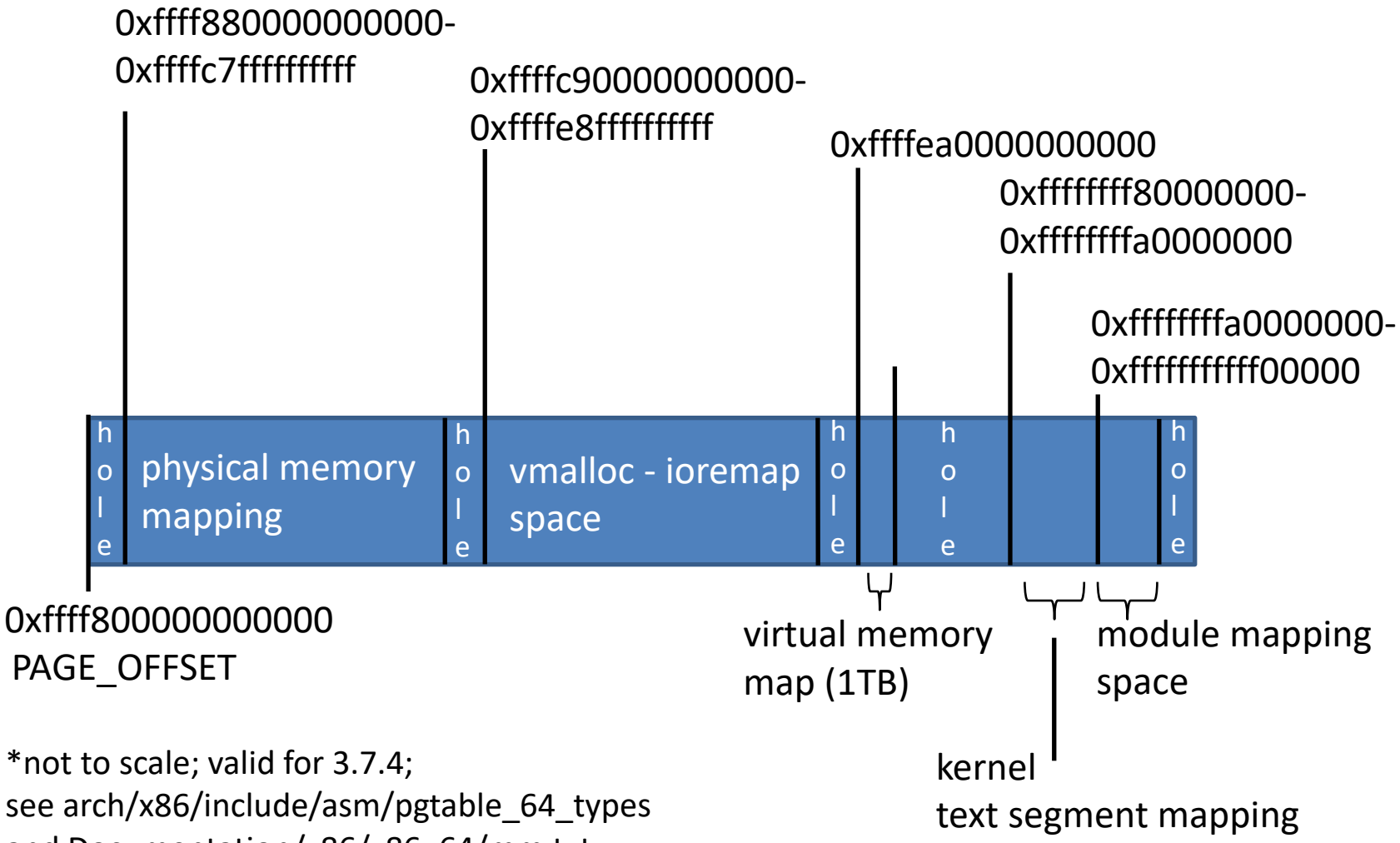


Kernel virtual address ranges
32-bit*



*very configuration dependent

Kernel virtual address ranges – 64-bit (128 TB)*



*not to scale; valid for 3.7.4;
see arch/x86/include/asm/pgtable_64_types
and Documentation/x86/x86_64/mm.txt

Kernel memory operations*

- kmalloc(size_): allocate a contiguous address range of size_ bytes and allocate contiguous physical memory to back it
  - Allocation is made from previously established caches of fixed size chunks
  - may not be page-frame aligned
  - chunk returned may be larger than size requested
- vmalloc(size_): allocate a contiguous address range of size_ bytes and allocate enough page frames to back it
  - Address returned is guaranteed page-frame aligned
  - Page frames are allocated individually and not contiguously
- kmap(struct page *page): allocate a kernel virtual address to map a page frame so that it can be accessed
  - Used for example to map a page of highmem allocated to a user process so that it can be accessed by a kernel routine
  - Doesn't allocate any page frames
  - Used for *brief* mappings because the addresses in this range are *severely* limited (so call kunmap() ASAP!)
  - also see kmap_atomic, which uses a *different* address range…

*this is just a basic subset…

vmalloc()

- Used to get access to page frames that aren't in the 'identity mapped' range
- Important files:
  - vmalloc.c
  - vmalloc.h
  - mm_types.h (page table management, struct vm_area_struct*, more…)
- The vmalloc code is highly self-contained
  - Similar but separate from code that manages user space mappings

*Used later for mmap of vmalloc'd allocations

Both kernel and user space virtual mapping information has to be tracked

pgtable_32_types.h | slub_def.h | slab.h | slab.c | slub.c | tboot.c | rbtree.h | sched.h | mm_types.h

(Unknown Scope)

```
226 struct vm_area_struct {
227     struct mm_struct * vm_mm;    /* The address space we belong to. */
228     unsigned long vm_start;      /* Our start address within vm_mm. */
229     unsigned long vm_end;        /* The first byte after our end address
230                                     within vm_mm. */
231
232     /* linked list of VM areas per task, sorted by address */
233     struct vm_area_struct *vm_next, *vm_prev;
234
235     pgprot_t vm_page_prot;       /* Access permissions of this VMA. */
236     unsigned long vm_flags;      /* Flags, see mm.h. */
237
238     struct rb_node vm_rb;
239
240     /*
241      * For areas with an address space and backing store,
242      * linkage into the address_space->i_mmap interval tree, or
243      * linkage of vma in the address_space->i_mmap_nonlinear list.
244      */
245     union {
246         struct {
247             struct rb_node rb;
248             unsigned long rb_subtree_last;
249         } linear;
250         struct list_head nonlinear;
251     } shared;
252
253     /*
254      * A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma
255      * list, after a COW of one of the file pages.  A MAP_SHARED vma
256      * can only be in the i_mmap tree.  An anonymous MAP_PRIVATE, stack
257      * or brk vma (with NULL file) can only be in an anon_vma list.
258      */
259     struct list_head anon_vma_chain; /* Serialized by mmap_sem &
260                       * page_table_lock */
261     struct anon_vma *anon_vma;  /* Serialized by page_table_lock */
262
263     /* Function pointers to deal with this struct. */
264     const struct vm_operations_struct *vm_ops;
265
266     /* Information about our backing store: */
267     unsigned long vm_pgoff;      /* Offset (within vm_file) in PAGE_SIZE
268                       units, *not* PAGE_CACHE_SIZE */
269     struct file * vm_file;       /* File we map to (can be NULL). */
270     void * vm_private_data;      /* was vm_pte (shared mem) */
271
272 #ifndef CONFIG_MMU
273     struct vm_region *vm_region;    /* NOMMU mapping region */
274 #endif
275 #ifdef CONFIG_NUMA
276     struct mempolicy *vm_policy;    /* NUMA policy for the VMA */
277 #endif
```

pgtable_32_types.h | slub_def.h | slab.h | slab.c | slub.c | rbtree.h | mm_types.h

(Unknown Scope)

```
311 struct mm_struct {
312     struct vm_area_struct * mmap;       /* list of VMAs */
313     struct rb_root mm_rb;
314     struct vm_area_struct * mmap_cache; /* last find_vma result */
315 #ifdef CONFIG_MMU
316     unsigned long (*get_unmapped_area) (struct file *filp,
317                 unsigned long addr, unsigned long len,
318                 unsigned long pgoff, unsigned long flags);
319     void (*unmap_area) (struct mm_struct *mm, unsigned long addr);
320 #endif
321     unsigned long mmap_base;        /* base of mmap area */
```

For a user-space process the vm areas are stored in both the process' mm->mmap list and mm->rb red-black tree

But for vmalloc() the kernel keeps a global LIST_HEAD and RB_ROOT internal to vmalloc.c (see next page)

Flags are important for mapping! More later!

sys_x86_64.c | syscalls.h | dma-mapping-common.h | util.c | mm.h | vmalloc.c | mmap.c

(Unknown Scope)

```
1704  *  __vmalloc_node  -  allocate virtually contiguous memory
1705  *  @size:      allocation size
1706  *  @align:     desired alignment
1707  *  @gfp_mask:  flags for the page level allocator
1708  *  @prot:      protection mask for the allocated pages
1709  *  @node:      node to use for allocation or -1
1710  *  @caller:    caller's return address
1711  *
1712  *  Allocate enough pages to cover @size from the page level
1713  *  allocator with @gfp_mask flags.  Map them into contiguous
1714  *  kernel virtual space, using a pagetable protection of @prot.
1715  */
1716 static void *__vmalloc_node(unsigned long size, unsigned long align,
1717                             gfp_t gfp_mask, pgprot_t prot,
1718                             int node, const void *caller)
1719 {
1720     return __vmalloc_node_range(size, align, VMALLOC_START, VMALLOC_END,
1721                             gfp_mask, prot, node, caller);
1722 }
1723
1724 void *__vmalloc(unsigned long size, gfp_t gfp_mask, pgprot_t prot)
1725 {
1726     return __vmalloc_node(size, 1, gfp_mask, prot, -1,
1727                             __builtin_return_address(0));
1728 }
1729 EXPORT_SYMBOL(__vmalloc);
1730
1731 static inline void *__vmalloc_node_flags(unsigned long size,
1732                             int node, gfp_t flags)
1733 {
1734     return __vmalloc_node(size, 1, flags, PAGE_KERNEL,
1735                             node, __builtin_return_address(0));
1736 }
1737
1738 /**
1739  *  vmalloc  -  allocate virtually contiguous memory
1740  *  @size:      allocation size
1741  *  Allocate enough pages to cover @size from the page level
1742  *  allocator and map them into contiguous kernel virtual space.
1743  *
1744  *  For tight control over page level allocator and protection flags
1745  *  use __vmalloc() instead.
1746  */
1747 void *vmalloc(unsigned long size)
1748 {
1749     return __vmalloc_node_flags(size, -1, GFP_KERNEL | __GFP_HIGHMEM);
1750 }
1751 EXPORT_SYMBOL(vmalloc);
```

sys_x86_64.c | syscalls.h | util.c | mm.h | vmalloc.c | mmap.c | tlb.c

(Unknown Scope)

```
252 struct vmap_area {
253     unsigned long va_start;
254     unsigned long va_end;
255     unsigned long flags;
256     struct rb_node rb_node;      /* address sorted rbtree */
257     struct list_head list;       /* address sorted list */
258     struct list_head purge_list;    /* "lazy purge" list */
259     struct vm_struct *vm;
260     struct rcu_head rcu_head;
261 };
262
263 static DEFINE_SPINLOCK(vmap_area_lock);
264 static LIST_HEAD(vmap_area_list);
265 static struct rb_root vmap_area_root = RB_ROOT;
```

vmalloc.c global data for Tracking kernel vma's

Functions collectively set up parameters and flags for __vmalloc_node_range(), where real work gets done

START HERE

sys_x86_64.c | syscalls.h | dma-mapping-common.h | util.c | mm.h | vmalloc.c | mmap.c

(Unknown Scope)

```
1646  *  __vmalloc_node_range  -  allocate virtually contiguous memory
1647  *  @size:      allocation size
1648  *  @align:     desired alignment
1649  *  @start:     vm area range start
1650  *  @end:       vm area range end
1651  *  @gfp_mask:  flags for the page level allocator
1652  *  @prot:      protection mask for the allocated pages
1653  *  @node:      node to use for allocation or -1
1654  *  @caller:    caller's return address
1655  *
1656  *  Allocate enough pages to cover @size from the page level
1657  *  allocator with @gfp_mask flags.  Map them into contiguous
1658  *  kernel virtual space, using a pagetable protection of @prot.
1659  */
1660 void *__vmalloc_node_range(unsigned long size, unsigned long align,
1661             unsigned long start, unsigned long end, gfp_t gfp_mask,
1662             pgprot_t prot, int node, const void *caller)
1663 {
1664     struct vm_struct *area;
1665     void *addr;
1666     unsigned long real_size = size;
1667
1668     size = PAGE_ALIGN(size);
1669     if (!size || (size >> PAGE_SHIFT) > totalram_pages)
1670         goto fail;
1671
1672     area = __get_vm_area_node(size, align, VM_ALLOC | VM_UNLIST,
1673                 start, end, node, gfp_mask, caller);
1674     if (!area)
1675         goto fail;
1676
1677     addr = __vmalloc_area_node(area, gfp_mask, prot, node, caller);
1678     if (!addr)
1679         return NULL;
1680
1681     /*
1682      * In this function, newly allocated vm_struct is not added
1683      * to vmlist at __get_vm_area_node(). so, it is added here.
1684      */
1685     insert_vmalloc_vmlist(area);
1686
1687     /*
1688      * A ref_count = 3 is needed because the vm_struct and vmap_area
1689      * structures allocated in the __get_vm_area_node() function contain
1690      * references to the virtual address of the vmalloc'ed block.
1691      */
1692     kmemleak_alloc(addr, real_size, 3, gfp_mask);
1693
1694     return addr;
1695
1696 fail:
1697     warn_alloc_failed(gfp_mask, 0
```

sys_x86_64.c | syscalls.h | util.c | mm.h | vmalloc.c | mmap.c | tlb.c

(Unknown Scope)

```
252 struct vmap_area {
253     unsigned long va_start;
254     unsigned long va_end;
255     unsigned long flags;
256     struct rb_node rb_node;     /* address sorted rbtree */
257     struct list_head list;      /* address sorted list */
258     struct list_head purge_list;    /* "lazy purge" list */
259     struct vm_struct *vm;
260     struct rcu_head rcu_head;
261 };
262
263 static DEFINE_SPINLOCK(vmap_area_lock);
264 static LIST_HEAD(vmap_area_list);
265 static struct rb_root vmap_area_root = RB_ROOT;
```

Allocates the address range and saves it where it can be tracked

Allocates the pages to back the address and sets up the page tables *

Saves the vm_struct in the vmalloc vmlist

vmalloc.h

(Unknown Scope)

```
27  struct vm_struct {
28      struct vm_struct    *next;
29      void                *addr;
30      unsigned long       size;
31      unsigned long       flags;
32      struct page         **pages;
33      unsigned int        nr_pages;
34      phys_addr_t         phys_addr;
35      const void          *caller;
36  };
37
```

sys_x86_64.c | syscalls.h | dma-mapping-common.h | util.c | mm.h | vmalloc.c | mmap.c | mmu

(Unknown Scope)

```
1323  static struct vm_struct *__get_vm_area_node(unsigned long size,
1324          unsigned long align, unsigned long flags, unsigned long start,
1325          unsigned long end, int node, gfp_t gfp_mask, const void *caller)
1326  {
1327      struct vmap_area *va;
1328      struct vm_struct *area;
1329
1330      BUG_ON(in_interrupt());
1331      if (flags & VM_IOREMAP) {
1332          int bit = fls(size);
1333
1334          if (bit > IOREMAP_MAX_ORDER)
1335              bit = IOREMAP_MAX_ORDER;
1336          else if (bit < PAGE_SHIFT)
1337              bit = PAGE_SHIFT;
1338
1339          align = 1ul << bit;
1340      }
1341
1342      size = PAGE_ALIGN(size);
1343      if (unlikely(!size))
1344          return NULL;
1345
1346      area = kzalloc_node(sizeof(*area), gfp_mask & GFP_RECLAIM_MASK, node);
1347      if (unlikely(!area))
1348          return NULL;
1349
1350      /*
1351       * We always allocate a guard page.
1352       */
1353      size += PAGE_SIZE;
1354
1355      va = alloc_vmap_area(size, align, start, end, node, gfp_mask);
1356      if (IS_ERR(va)) {
1357          kfree(area);
1358          return NULL;
1359      }
1360
1361      /*
1362       * When this function is called from __vmalloc_node_range,
1363       * we do not add vm_struct to vmlist here to avoid
1364       * accessing uninitialized members of vm_struct such as
1365       * pages and nr_pages fields. They will be set later.
1366       * To distinguish it from others, we use a VM_UNLIST flag.
1367       */
1368      if (flags & VM_UNLIST)
1369          setup_vmalloc_vm(area, va, flags, caller);
1370      else
1371          insert_vmalloc_vm(area, va, flags, caller);
1372
1373      return area;
```

```
423  found:
424      if (addr + size > vend)
425          goto overflow;
426
427      va->va_start = addr;
428      va->va_end = addr + size;
429      va->flags = 0;
430      __insert_vmap_area(va);
431      free_vmap_cache = &va->rb_node;
432      spin_unlock(&vmap_area_lock);
433
434      BUG_ON(va->va_start & (align-1));
435      BUG_ON(va->va_start < vstart);
436      BUG_ON(va->va_end > vend);
437
438      return va;
```

Finds an available virtual address range starting with ‘addr’ and returns an initialized struct vmap_area

sys_x86_64.c | syscalls.h | util.c | mm.h | vmalloc.c | mmap.c | tlb.c | pgtable.c | stackprote

(Unknown Scope)

```
1290  static void setup_vmalloc_vm(struct vm_struct *vm, struct vmap_area *va,
1291                  unsigned long flags, const void *caller)
1292  {
1293      vm->flags = flags;
1294      vm->addr = (void *)va->va_start;
1295      vm->size = va->va_end - va->va_start;
1296      vm->caller = caller;
1297      va->vm = vm;
1298      va->flags |= VM_VM_AREA;
1299  }
1300
1301  static void insert_vmalloc_vmlist(struct vm_struct *vm)
1302  {
1303      struct vm_struct *tmp, **p;
1304
1305      vm->flags &= ~VM_UNLIST;
1306      write_lock(&vmlist_lock);
1307      for (p = &vmlist; (tmp = *p) != NULL; p = &tmp->next) {
1308          if (tmp->addr >= vm->addr)
1309              break;
1310      }
1311      vm->next = *p;
1312      *p = vm;
1313      write_unlock(&vmlist_lock);
1314  }
1315
1316  static void insert_vmalloc_vm(struct vm_struct *vm, struct vmap_area *va,
1317                  unsigned long flags, const void *caller)
1318  {
1319      setup_vmalloc_vm(vm, va, flags, caller);
1320      insert_vmalloc_vmlist(vm);
1321  }
1322
```

kmalloc()

- kmalloc allocates physically contiguous memory from the slab caches initialized by start_kernel()
  - Can grow the cache if it needs to
  - May sleep if allocation isn't available
  - Allocations aren't page aligned and must be adjusted for mmap()
- Physically-contiguous memory == contiguous pages/pfn's
  - Important for mmap() of kmalloc() allocations

mmap()

- mmap() attaches page frames to user-space virtual addresses
- mmap() has as an argument the file descriptor of an open file
  - The fd may resolve to a file system or to a kernel module (or neither for the heap malloc() case)
- The physical memory that is mapped may be
  - For page frames acquired by kmalloc() or vmalloc()
  - For device resources assigned a physical address
  - For pages in the page cache filled with data from an on-disk file
  - Whatever the file system or kernel module that implements the file_operations mmap() method supplies for a page fault
- Important files: mm/mmap.c, mm/util.c , mm/memory.c (for remap_pfn_range)

```
fd = open("/dev/mmapper_k", O_RDWR | O_SYNC);
if( fd == -1) {
        printf("open error...\n");
        return -1;
}

mptr = mmap(0, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

```
/* read from mmap memory */
    printf("mptr is %p\n", mptr);
    memset(buffer, 0, size);            /* Clear the buffer */
    memcpy(buffer, mptr, 78);
buffer[79] = '\n';
buffer[80] = '\0';
    printf("mmap:  '%s'\n", buffer);
    printf("mptr is %p\n", mptr);
```

util.c | mm.h | vmalloc.c | mmap.c* | tlb.c | pgtable.c | stackprotector.h | common.c

(Unknown Scope)

```
1002  unsigned long do_mmap_pgoff(struct file *file, unsigned long addr,
1003              unsigned long len, unsigned long prot,
1004              unsigned long flags, unsigned long pgoff)
1005  {
1006      struct mm_struct * mm = current->mm;
1007      struct inode *inode;
1008      vm_flags_t vm_flags;
1009
1010      (SOME PARAM VALIDATION and ERROR CHECKING)
1011
1012      /* Obtain the address to map to. we verify (or select) it and ensure
1013       * that it represents a valid section of the address space.
1014       */
1015      addr = get_unmapped_area(file, addr, len, pgoff, flags);
1016      if (addr & ~PAGE_MASK)
1017          return addr;
1018
1019      (MORE CHECKING)
1020
1021      inode = file ? file->f_path.dentry->d_inode : NULL;
1022      if (file) {
1023          switch (flags & MAP_TYPE) {
1024          case MAP_SHARED:
1025              if ((prot&PROT_WRITE) && !(file->f_mode&FMODE_WRITE))
1026                  return -EACCES;
1027
1028              (MORE CHECKING)
1029
1030              vm_flags |= VM_SHARED | VM_MAYSHARE;
1031              if (!(file->f_mode & FMODE_WRITE))
1032                  vm_flags &= ~(VM_MAYWRITE | VM_SHARED);
1033              /* fall through */
1034          case MAP_PRIVATE:
1035              if (!(file->f_mode & FMODE_READ))
1036                  return -EACCES;
1037              if (file->f_path.mnt->mnt_flags & MNT_NOEXEC) {
1038                  if (vm_flags & VM_EXEC)
1039                      return -EPERM;
1040                  vm_flags &= ~VM_MAYEXEC;
1041              }
1042
1043              if (!file->f_op || !file->f_op->mmap)
1044                  return -ENODEV;
1045              break;
1046          default:
1047              return -EINVAL;
1048          }
1049      } else {
1050          (ANON CASE)
1051      }
1052      return mmap_region(file, addr, len, flags, vm_flags, pgoff);
1053  }
```

Dump_stack with kprobe:

```
[<c0525581>] ? do_mmap_pgoff+0x1/0x2d0
[<c0514564>] ? vm_mmap_pgoff+0x64/0x90
[<c0523ed2>] ? sys_mmap_pgoff+0x42/0x100
[<c097d477>] ? syscall_call+0x7/0xb
```

Validate parameters, set file=fget(fd),
Take the mm->mmap_sem lock

Get a virtual address from the process address space

The struct file must have a registered mmap method

Validate the address, allocate and Initialize the vma

```
1310      vma->vm_mm = mm;
1311      vma->vm_start = addr;
1312      vma->vm_end = addr + len;
1313      vma->vm_flags = vm_flags;
1314      vma->vm_page_prot = vm_get_page_prot(vm_flags);
1315      vma->vm_pgoff = pgoff;
1316      INIT_LIST_HEAD(&vma->anon_vma_chain);
```

Then call the mmap() method…

```
1329          vma->vm_file = get_file(file);
1330          error = file->f_op->mmap(file, vma);
1331          if (error)
1332              goto unmap_and_free_vma;
```

mmap: kmalloc

- The address returned by kmalloc() may not be page-aligned, so adjust accordingly – mmap requires this

Adjust for page alignment

```
static int __init mmap_kmalloc_init (void) {
        int i;

        kmalloc_ptr = kmalloc(LEN + PAGE_SIZE, GFP_KERNEL);
        if (!kmalloc_ptr) {
                printk("kmalloc failed\n");
                return -ENOMEM;
        }
        printk("kmalloc_ptr at 0x%p \n", kmalloc_ptr);

        //#define PAGE_MASK (~(PAGE_SIZE-1))
        //page-align the area to map

        kmalloc_area = (char *)(((unsigned long)kmalloc_ptr + PAGE_SIZE -1) & PAGE_MASK);

        printk("kmalloc_area :0x%p \t physical Address 0x%lx)\n", kmalloc_area,
                        (unsigned long)virt_to_phys((void *)(kmalloc_area)));

        //initialize the memory for the user app to read
        for (i = 0; i<LEN; i++) {
            kmalloc_area[i] = '0' + (i % 10);
        };

        return 0;
}
```

mmap: the kernel module

The sample module was created for an assignment in the Advanced Linux Drivers class and subsequently extended as an experiment. It allocates and initializes two buffers, one using kmalloc, the other using vmalloc.

The driver must set up a mmap method…

```
static struct file_operations mmapper_fops = {
    open:    mmapper_open,
    release: mmapper_release,
    mmap:    mmapper_mmap,

//#if LINUX_VERSION_CODE >= KERNEL_VERSION(3,0,0)
        unlocked_ioctl:   mmapper_ioctl,
//#else
//        ioctl: mapper_ioctl,
//#endif
    owner:   THIS_MODULE
};
```

```
static int mmapper_mmap(struct file *filp, struct vm_area_struct *vma)
{
    struct mmapper_dev *thisMPR = (struct mmapper_dev *)filp->private_data;
    int ret;

    if (!thisMPR) {
        printk(KERN_ALERT "mmapper mmap: No device instance for minor %d\n",
            iminor(filp->f_dentry->d_inode));
        return -ENODEV;
    }

    //ensure no other operations on the device are in flight
    if ((ret = mutex_lock_interruptible(&thisMPR->mtx)) !=0) {
        printk(KERN_ALERT "mmapper - unable to take lock in mmap, ret= %x\n", ret);
        return ret;
    }

    if (thisMPR->dev_buf == kmalloc_area)
        ret = mmap_kmalloc(thisMPR, vma);
    else if (thisMPR->dev_buf == vmalloc_ptr)
        ret = mmap_vmalloc(thisMPR, vma);
    else {
```

mmap of a kmalloc allocation

```
static int mmap_kmalloc(struct mmapper_dev *thisMPR,
            struct vm_area_struct * vma) {
        int ret = 0;
        unsigned long length = vma->vm_end - vma->vm_start;

        /* Restrict it to size of device memory */
        if (length > thisMPR->dev_len)
                return -EIO;

        ret = remap_pfn_range(
           vma,
           vma->vm_start,
               virt_to_phys((void*)((unsigned long)kmalloc_area)) >> PAGE_SHIFT,
               length,
           vma->vm_page_prot
             );
        if(ret != 0) {
                printk(KERN_ALERT "Unable to map k
            "length %d, ret %x\n", vma->vm_start,
        }
        return 0;
}
```

Called with the mm->mmap_sem held (way back in vm_mmap_pgoff)

Contiguous memory, only need the first pfn

The semaphore was taken in the call to vm_mmap_pgoff()

memory.c | Header1.h* | mmap.c | util.c | mm.h | vmalloc.c | mmap.c | tlb.c

(Unknown Scope)

```
2286
2287 /**
2288  * remap_pfn_range - remap kernel memory to userspace
2289  * @vma: user vma to map to
2290  * @addr: target user address to start at
2291  * @pfn: physical address of kernel memory
2292  * @size: size of map area
2293  * @prot: page protection flags for this mapping
2294  *
2295  *  Note: this is only safe if the mm semaphore is held when called.
2296  */
2297 int remap_pfn_range(struct vm_area_struct *vma, unsigned long addr,
2298             unsigned long pfn, unsigned long size, pgprot_t prot)
2299 {
```

memory.c* | Header1.h* | mmap.c | util.c | mm.h | vmalloc.c | mmap.c | tlb.c | pgtable.c

(Unknown Scope)

```
2297  int remap_pfn_range(struct vm_area_struct *vma, unsigned long addr,
2298              unsigned long pfn, unsigned long size, pgprot_t prot)
2299  {
2300      pgd_t *pgd;
2301      unsigned long next;
2302      unsigned long end = addr + PAGE_ALIGN(size);
2303      struct mm_struct *mm = vma->vm_mm;
2304      int err;
2305
2306      /*
2307       * Physically remapped pages are special. Tell the rest of the world about it:
2308       *   VM_IO tells people not to look at these pages
2309       *  (accesses can have side effects).
2310       *   VM_PFNMAP tells the core MM that the base pages are just
2311       *  raw PFN mappings, and do not have a "struct page" associated with them.
2312       *   VM_DONTEXPAND
2313       *      Disable vma merging and expanding with mremap().
2314       *   VM_DONTDUMP
2315       *      Omit vma from core dump, even when VM_IO turned off.
2316       *
2317       * There's a horrible special case to handle copy-on-write behaviour that some
2318       * programs depend on. We mark the "original" un-COW'ed pages by matching them
2319       * up with "vma->vm_pgoff". See vm_normal_page() for details.
2320       */
2321      if (is_cow_mapping(vma->vm_flags)) {
2322          if (addr != vma->vm_start || end != vma->vm_end)
2323              return -EINVAL;
2324          vma->vm_pgoff = pfn;
2325      }
2326
2327      err = track_pfn_remap(vma, &prot, pfn, addr, PAGE_ALIGN(size));
2328      if (err)
2329          return -EINVAL;
2330
2331      vma->vm_flags |= VM_IO | VM_PFNMAP | VM_DONTEXPAND | VM_DONTDUMP;
2332
2333      BUG_ON(addr >= end);
2334      pfn -= addr >> PAGE_SHIFT;
2335      pgd = pgd_offset(mm, addr);
2336      flush_cache_range(vma, addr, end);
2337      do {
2338          next = pgd_addr_end(addr, end);
2339          err = remap_pud_range(mm, pgd, addr, next,
2340                  pfn + (addr >> PAGE_SHIFT), prot);
2341          if (err)
2342              break;
2343      } while (pgd++, addr = next, addr != end);
2344
2345      if (err)
2346          untrack_pfn(vma, pfn, PAGE_ALIGN(size));
2347      return err;
2348  }
```

Flags are important.
VM_PFNMAP says the vma is ineligible for swapping.

Older kernels used VM_RESERVED but that has been deprecated

NOP on x86

Build the page tables with the Kmalloc pfns

mmap: vmalloc, approach 1

- Loop over all pages and map each individually

```
long length = vma->vm_end - vma->vm_start;
unsigned long start = vma->vm_start;
/* loop over all pages, map it page individually */
while (length > 0) {
    pfn = vmalloc_to_pfn(vmalloc_area_ptr);
    if ((ret = remap_pfn_range(vma, start, pfn, PAGE_SIZE,
            PAGE_SHARED)) < 0) {
        return ret;
    }
    start += PAGE_SIZE;
    vmalloc_area_ptr += PAGE_SIZE;
    length -= PAGE_SIZE;
}
```

http://zakirsblog.blogspot.com/2010/03/mmap.html

mmap: vmalloc, approach 2

- Minimal work in mmap method
- Maps on page faults when the user process tries to access the virtual address

```
static int __init mmap_vmalloc_init (void) {

        int i;

    // Allocate  memory  with vmalloc.  It is already page aligned
        vmalloc_ptr = ((char *)vmalloc(LEN));
        if (!vmalloc_ptr) {
                printk("vmalloc failed\n");
                return -ENOMEM;
        }
        printk("vmalloc_ptr at 0x%p \n", vmalloc_ptr);

    for (i = 0; i<LEN; i++) {
        vmalloc_ptr[i] = 'a' + (i % 26);
    };

    return 0;
}
```

```
static struct vm_operations_struct mmapper_vmem_ops = {
    .fault = file_vma_fault,
};

int mmap_vmalloc(struct mmapper_dev *thisMPR,
            struct vm_area_struct *vma)
{
        long length = vma->vm_end - vma->vm_start;

        // Restrict it to size of device memory
        if (length > thisMPR->dev_len)
                return -EIO;

    vma->vm_ops = &mmapper_vmem_ops;
    vma->vm_private_data = vmalloc_ptr;
    vma->vm_pgoff = (unsigned long) vmalloc_ptr >> PAGE_SHIFT;
#if LINUX_VERSION_CODE >= KERNEL_VERSION(3,4,0)
    vma->vm_flags |= VM_DONTEXPAND | VM_DONTDUMP | VM_PFNMAP;
#else
    vma->vm_flags |= VM_DONTEXPAND | VM_RESERVED;
#endif
        printk("Mapping vmalloc_area_ptr: 0x%p \n", vmalloc_ptr);

        return 0;
}
```

When the fault happens …

Page fault handling

- Since there is no pte for the address, a page fault occurs when the user mmapping process access it
- The fault handling path leads to the module fault handler

memory.c* | sg.c | sched.h | memcontrol.c | compaction.c | exec.c | signal.c | ttm

(Unknown Scope)

```
3210  static int __do_fault(struct mm_struct *mm, struct vm_area_struct *vma,
3211          unsigned long address, pmd_t *pmd,
3212          pgoff_t pgoff, unsigned int flags, pte_t orig_pte)
3213  {
3214      pte_t *page_table;
3215      spinlock_t *ptl;
3216      struct page *page;
3217      struct page *cow_page;
3218      pte_t entry;
3219      int anon = 0;
3220      struct page *dirty_page = NULL;
3221      struct vm_fault vmf;
3222      int ret;
3223      int page_mkwrite = 0;
3224
3225      ...
3226
3227      vmf.virtual_address = (void __user *)(address & PAGE_MASK);
3228      vmf.pgoff = pgoff;
3229      vmf.flags = flags;
3230      vmf.page = NULL;
3231
3232      ret = vma->vm_ops->fault(vma, &vmf);
3233
3234      ...
```

… now goes on to build the page table entry and complete the faulting operation

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

    printk(KERN_ALERT "in vmalloc fault for addr %lx\n", vmf->pgoff << PAGE_SHIFT);
    page = vmalloc_to_page((void *)(vmf->pgoff << PAGE_SHIFT));
    if (!page)
        return VM_FAULT_SIGBUS;

    get_page(page);
    vmf->page = page;
    printk(KERN_ALERT "returning page  %x\n", (unsigned int)page);
    return 0;
}
```

mtdchar.c | mem.c | pgtable.h | memory.c | mmap.c | util.c | mm.h | vmalloc.c | mmap

(Unknown Scope)

```c
 * Walk a vmap address to the struct page it maps.
 */
struct page *vmalloc_to_page(const void *vmalloc_addr)
{
    unsigned long addr = (unsigned long) vmalloc_addr;
    struct page *page = NULL;
    pgd_t *pgd = pgd_offset_k(addr);

    /*
     * XXX we might need to change this if we add VIRTUAL_BUG_ON for
     * architectures that do not vmalloc module space
     */
    VIRTUAL_BUG_ON(!is_vmalloc_or_module_addr(vmalloc_addr));

    if (!pgd_none(*pgd)) {
        pud_t *pud = pud_offset(pgd, addr);
        if (!pud_none(*pud)) {
            pmd_t *pmd = pmd_offset(pud, addr);
            if (!pmd_none(*pmd)) {
                pte_t *ptep, pte;

                ptep = pte_offset_map(pmd, addr);
                pte = *ptep;
                if (pte_present(pte))
                    page = pte_page(pte);
                pte_unmap(ptep);
            }
        }
    }
    return page;
}
EXPORT_SYMBOL(vmalloc_to_page);
```

Finds the page in the vmalloc mapping and returns it to be used in the user-space mmap

Mmap: vmalloc, approach 3

- Combining vmalloc_user with remap_vmalloc range
- The new, 'preferred' approach for mapping vmalloc allocations to user space

The only change, but an important one!

```
static int __init mmap_vmalloc_init (void) {

        int i;

    // Allocate  memory  with vmalloc. It is already page aligned
        vmalloc_ptr = ((char *)vmalloc_user(LEN));
        if (!vmalloc_ptr) {
                printk("vmalloc failed\n");
                return -ENOMEM;
        }
        printk("vmalloc_ptr at 0x%p \n", vmalloc_ptr);

    for (i = 0; i<LEN; i++) {
        vmalloc_ptr[i] = 'a' + (i % 26);
    };

    return 0;
```

Note the flags

pgtable.h | memory.c | mmap.c | util.c | mm.h | vmalloc.c | mm

Scope)

```
 */
void *vmalloc_user(unsigned long size)
{
    struct vm_struct *area;
    void *ret;

    ret = __vmalloc_node(size, SHMLBA,
                GFP_KERNEL | __GFP_HIGHMEM | __GFP_ZERO,
                PAGE_KERNEL, -1, __builtin_return_address(0));
    if (ret) {
        area = find_vm_area(ret);
        area->flags |= VM_USERMAP;
    }
    return ret;
}
EXPORT_SYMBOL(vmalloc_user);
```

```c
int mmap_vmalloc(struct mmapper_dev *thisMPR,
            struct vm_area_struct *vma)
{
        long length = vma->vm_end - vma->vm_start;
        int ret;

        // Restrict it to size of device memory
        if (length > thisMPR->dev_len)
                return -EIO;

        ret = remap_vmalloc_range(vma, vmalloc_ptr, 0)

        printk("Mapping vmalloc_area_ptr: 0x%p , status %x\n",
           vmalloc_ptr, ret);

        return ret;
}
```

Much simpler!

Get the vmalloc mapping

Flags are important…
vmalloc() won't set VM_USERMAP

Page at a time, extract from the vmalloc and plug into the user table

mem.c | pgtable.h | memory.c | mmap.c | util.c | mm.h | vmalloc.c | mmap.c

(Unknown Scope)

```c
 2119  *  remap_vmalloc_range  -  map vmalloc pages to userspace
 2120  *  @vma:       vma to cover (map full range of vma)
 2121  *  @addr:      vmalloc memory
 2122  *  @pgoff:     number of pages into addr before first page to map
 2123  *
 2124  *  Returns:    0 for success, -Exxx on failure
 2125  *
 2126  *  This function checks that addr is a valid vmalloc'ed area, and
 2127  *  that it is big enough to cover the vma. Will return failure if
 2128  *  that criteria isn't met.
 2129  *
 2130  *  Similar to remap_pfn_range() (see mm/memory.c)
 2131  */
 2132 int remap_vmalloc_range(struct vm_area_struct *vma, void *addr,
 2133                         unsigned long pgoff)
 2134 {
 2135     struct vm_struct *area;
 2136     unsigned long uaddr = vma->vm_start;
 2137     unsigned long usize = vma->vm_end - vma->vm_start;
 2138
 2139     if ((PAGE_SIZE-1) & (unsigned long)addr)
 2140         return -EINVAL;
 2141
 2142     area = find_vm_area(addr);
 2143     if (!area)
 2144         return -EINVAL;
 2145
 2146     if (!(area->flags & VM_USERMAP))
 2147         return -EINVAL;
 2148
 2149     if (usize + (pgoff << PAGE_SHIFT) > area->size - PAGE_SIZE)
 2150         return -EINVAL;
 2151
 2152     addr += pgoff << PAGE_SHIFT;
 2153     do {
 2154         struct page *page = vmalloc_to_page(addr);
 2155         int ret;
 2156
 2157         ret = vm_insert_page(vma, uaddr, page);
 2158         if (ret)
 2159             return ret;
 2160
 2161         uaddr += PAGE_SIZE;
 2162         addr += PAGE_SIZE;
 2163         usize -= PAGE_SIZE;
 2164     } while (usize > 0);
 2165
 2166     vma->vm_flags |= VM_DONTEXPAND | VM_DONTDUMP;
 2167
 2168     return 0;
 2169 }
```

100 %

Module access to user space

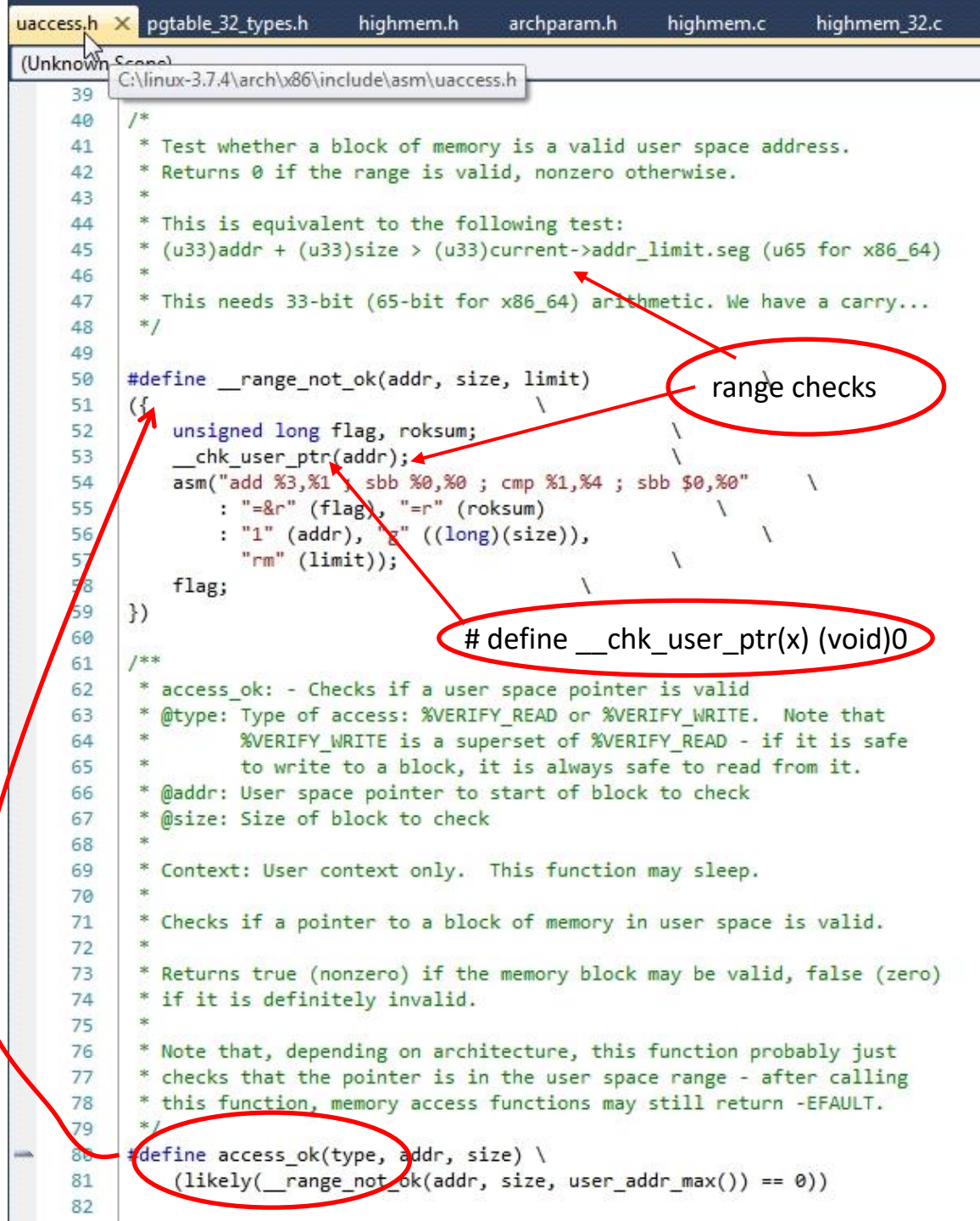
- mmap() allows user space access to kernel addresses.
- In process context, the kernel has access to the user space addresses but cannot trust them
  - They might be bogus or have no backing pages
- The kernel might have to access process user pages when not in that process's address space
  - Remember, every process has a different set of VMAs
- As with all memory access, implementation is a mix of abstracted and architecturally specific parts

Process context syscalls

- e.g. handling file_ops read/write/ioctl… involving reads and writes of user-supplied buffers
- Kernel can't trust user virtual addresses
  - Must use copy_to/from_user(), put/get_user(), access_ok() and variations…
    - copy_to_user(), copy_from_user() call access_ok() and __copy_*_user().
    - put_user() and get_user() copy using n-byte data types (1, 2, 4…
    - access_ok validates the user address+size for multiple, cheaper accesses to the buffer during operation of the function
      - E.g. multiple subsequent calls to __put_user or __copy_to_user don't call access_ok()

access_ok()

- Called at the beginning of many other copy/put operations
- Range checks that the address to (address+size-1) is in the valid range for the calling process
- Does not guarantee the page is really there though




```
39
40  /*
41   * Test whether a block of memory is a valid user space address.
42   * Returns 0 if the range is valid, nonzero otherwise.
43   *
44   * This is equivalent to the following test:
45   * (u33)addr + (u33)size > (u33)current->addr_limit.seg (u65 for x86_64)
46   *
47   * This needs 33-bit (65-bit for x86_64) arithmetic. We have a carry...
48   */
49
50  #define __range_not_ok(addr, size, limit)                       \
51  ({                                                              \
52      unsigned long flag, roksum;                                 \
53      __chk_user_ptr(addr);                                       \
54      asm("add %3,%1 ; sbb %0,%0 ; cmp %1,%4 ; sbb $0,%0"         \
55          : "=&r" (flag), "=r" (roksum)                           \
56          : "1" (addr), "g" ((long)(size)),                       \
57            "rm" (limit));                                        \
58      flag;                                                       \
59  })
60
61  /**
62   * access_ok: - Checks if a user space pointer is valid
63   * @type: Type of access: %VERIFY_READ or %VERIFY_WRITE.  Note that
64   *        %VERIFY_WRITE is a superset of %VERIFY_READ - if it is safe
65   *        to write to a block, it is always safe to read from it.
66   * @addr: User space pointer to start of block to check
67   * @size: Size of block to check
68   *
69   * Context: User context only.  This function may sleep.
70   *
71   * Checks if a pointer to a block of memory in user space is valid.
72   *
73   * Returns true (nonzero) if the memory block may be valid, false (zero)
74   * if it is definitely invalid.
75   *
76   * Note that, depending on architecture, this function probably just
77   * checks that the pointer is in the user space range - after calling
78   * this function, memory access functions may still return -EFAULT.
79   */
80  #define access_ok(type, addr, size) \
81      (likely(__range_not_ok(addr, size, user_addr_max()) == 0))
82
```

Range Checking doesn't eliminate page faults. The kernel handles them as previously discussed.

uaccess.h

(Unknown Scope)

```
547
548 #define put_user_try        uaccess_try
549 #define put_user_catch(err) uaccess_catch(err)
550
```

Same for put and get

http://lkml.iu.edu/hypermail/linux/kernel/0901.2/03293.html

From: Hiroshi Shimamoto <h-shimamoto@xxxxxxxxxxxx>

Impact: introduce new uaccess exception handling framework

Introduce {get|put}_user_try and {get|put}_user_catch as new uaccess exception handling framework.
{get|put}_user_try begins exception block and {get|put}_user_catch(err) ends the block and gets err if an exception occured in {get|put}_user_ex() in the block. The exception is stored thread_info->uaccess_err.

See slide 40 'syscall handling' for an example of the framework in use.

Vbox\linux-2.6.32.61\ uaccess.h

Unknown Scope)

C:\linux-3.7.4\arch\x86\include\asm\uaccess.h

```
466
467 /*
468  * uaccess_try and catch
469  */
470 #define uaccess_try do {                                    \
471     current_thread_info()->uaccess_err = 0;                 \
472     stac();                                                 \
473     barrier();
474
475 #define uaccess_catch(err)                                  \
476     clac();                                                 \
477     (err) |= (current_thread_info()->uaccess_err ? -EFAULT : 0);    \
478 } while (0)
479
```

Call to allow override of Supervisor mode access protection

ia32_signal.c | uaccess.h | compiler.h | uaccess_64.h | usercopy_32.c | uaccess.h | uaccess.h

(Unknown Scope)

C:\linux-3.7.4\include\asm-generic\uaccess.h

```
242
243 static inline long copy_from_user(void *to,
244         const void __user * from, unsigned long n)
245 {
246     might_sleep();
247     if (access_ok(VERIFY_READ, from, n))
248         return __copy_from_user(to, from, n);
249     else
250         return n;
251 }
252
253 static inline long copy_to_user(void __user *to,
254         const void *from, unsigned long n)
255 {
256     might_sleep();
257     if (access_ok(VERIFY_WRITE, to, n))
258         return __copy_to_user(to, from, n);
259     else
260         return n;
261 }
```

putuser.S | uaccess_32.h | ia32_signal.c | uaccess.h | uaccess_64.h* | usercopy_32.c

(Unknown Scope)

```
122
123 static __always_inline __must_check
124 int __copy_to_user(void __user *dst, const void *src, unsigned size)
125 {
126     int ret = 0;
127
128     might_fault();
129     if (!__builtin_constant_p(size))
130         return copy_user_generic((__force void *)dst, src, size);
131     switch (size) {
132     case 1:__put_user_asm(*(u8 *)src, (u8 __user *)dst,
133                   ret, "b", "b", "iq", 1);
134         return ret;
135     case 2:__put_user_asm(*(u16 *)src, (u16 __user *)dst,
136                   ret, "w", "w", "ir", 2);
137         return ret;
138     case 4:__put_user_asm(*(u32 *)src, (u32 __user *)dst,
139                   ret, "l", "k", "ir", 4);
140     <more...>
141
142         return ret;
143     default:
144         return copy_user_generic((__force void *)dst, src, size);
145     }
146 }
```

copy_from_user()

↓

__copy_from_user ← arch/x86/include/asm/uaccess_32.h

↓

__copy_from_user_||() ← arch/x86/lib/usercopy_32.c

Guarantees that the pages
ARE available and copies the bytes

32-bit & 64-bit: similar but different...

put/get user() are used to copy small amounts of data, chars, ints etc.

```
#define __put_user_x(size, x, ptr, __ret_pu)\
asm volatile("call __put_user_" #size : "=a" (__ret_pu)\
    : "0" ((typeof(*(ptr)))(x)), "c" (ptr) : "ebx")
```

From x86/lib/putuser.S :

```
ENTRY(__put_user_4)
ENTER
mov TI_addr_limit(%_ASM_BX),%_ASM_BX
sub $3,%_ASM_BX
cmp %_ASM_BX,%_ASM_CX
jae bad_put_user
ASM_STAC
3:movl %eax,(%_ASM_CX)
xor %eax,%eax
EXIT
```

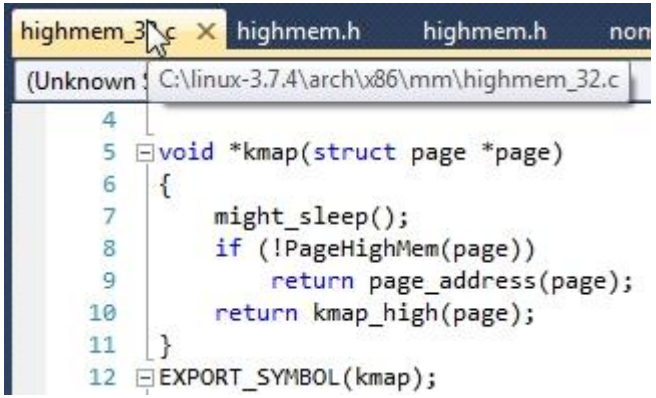
uaccess_32.h | ia32_signal.c | uaccess.h* | compiler.h | uaccess_64.h | usercopy_32.c

(Unknown Scope)

```
241
242 /**
243  * put_user: - Write a simple value into user space.
244  * @x:   Value to copy to user space.
245  * @ptr: Destination address, in user space.
246  *
247  * Context: User context only.  This function may sleep.
248  *
249  * This macro copies a single simple value from kernel space to user
250  * space.  It supports simple types like char and int, but not larger
251  * data types like structures or arrays.
252  *
253  * @ptr must have pointer-to-simple-variable type, and @x must be
254  * assignable
255  * to the result of dereferencing @ptr.
256  *
257  * Returns zero on success, or -EFAULT on error.
258  */
259 #define put_user(x, ptr)                    \
260 ({                                  \
261     int __ret_pu;                       \
262     __typeof__(*(ptr)) __pu_val;                \
263     __chk_user_ptr(ptr);                    \
264     might_fault();                      \
265     __pu_val = x;                       \
266     switch (sizeof(*(ptr))) {               \
267     case 1:                             \
268         __put_user_x(1, __pu_val, ptr, __ret_pu);   \
269         break;                          \
270     case 2:                             \
271         __put_user_x(2, __pu_val, ptr, __ret_pu);   \
272         break;                          \
273     case 4:                             \
274         __put_user_x(4, __pu_val, ptr, __ret_pu);   \
275         break;                          \
276     case 8:                             \
277         __put_user_x8(__pu_val, ptr, __ret_pu);     \
278         break;                          \
279     default:                            \
280         __put_user_x(X, __pu_val, ptr, __ret_pu);   \
281         break;                          \
282     }                                   \
283     __ret_pu;                           \
284 })
285
```

Accessing another user-space context: kmap()

- Modules operating in process context share the page tables of the calling user
- Getting access to a different user process context isn't recommended but can be necessary
  - Example: cat /proc/<pid>/cmdline, which reads the command and arguments for the specified pid from the stack of its primary thread
- The trick is to get a valid vaddr backed by the appropriate physical pages
- The kmap() call accomplishes this
  - Only needed for 32-bit with highmem
  - 64-bit returns page_address(page)



Task_struct for the target pid

proc_pid_cmdline(struct task_struct, *task, buffer)

↓

access_process_vm()*

↓

access_remote_vm()

↓

__access_remote_vm()

Maps one page at a time

mm_struct for the pid user-space

Sets flags and calls __get_user_pages

kmap(): allocate an address for this page

memory.c | base.c

(Unknown Scope)

```
3832  static int __access_remote_vm(struct task_struct *tsk, struct mm_struct *mm,
3833          unsigned long addr, void *buf, int len, int write)
3834  {
3835      struct vm_area_struct *vma;
3836      void *old_buf = buf;
3837
3838      down_read(&mm->mmap_sem);
3839      /* ignore errors, just check how much was successfully transferred */
3840      while (len) {
3841          int bytes, ret, offset;
3842          void *maddr;
3843          struct page *page = NULL;
3844
3845          ret = get_user_pages(tsk, mm, addr, 1,
3846                  write, 1, &page, &vma);
3847          if (ret <= 0) {
3848              /*
3849               * Check if this is a VM_IO | VM_PFNMAP VMA, which
3850               * we can access using slightly different code.
3851               */
3852  #ifdef CONFIG_HAVE_IOREMAP_PROT
3853              vma = find_vma(mm, addr);
3854              if (!vma || vma->vm_start > addr)
3855                  break;
3856              if (vma->vm_ops && vma->vm_ops->access)
3857                  ret = vma->vm_ops->access(vma, addr, buf,
3858                                  len, write);
3859              if (ret <= 0)
3860  #endif
3861                  break;
3862              bytes = ret;
3863          } else {
3864              bytes = len;
3865              offset = addr & (PAGE_SIZE-1);
3866              if (bytes > PAGE_SIZE-offset)
3867                  bytes = PAGE_SIZE-offset;
3868
3869              maddr = kmap(page);
3870              if (write) {
3871                  copy_to_user_page(vma, page, addr,
3872                                    maddr + offset, buf, bytes);
3873                  set_page_dirty_lock(page);
3874              } else {
3875                  copy_from_user_page(vma, page, addr,
3876                                      buf, maddr + offset, bytes);
3877              }
3878              kunmap(page);
3879              page_cache_release(page);
```

100 %

cacheflush.h | putuser.S | uaccess_32.h | ia32_signal.c | uaccess.h | usercopy_32.c

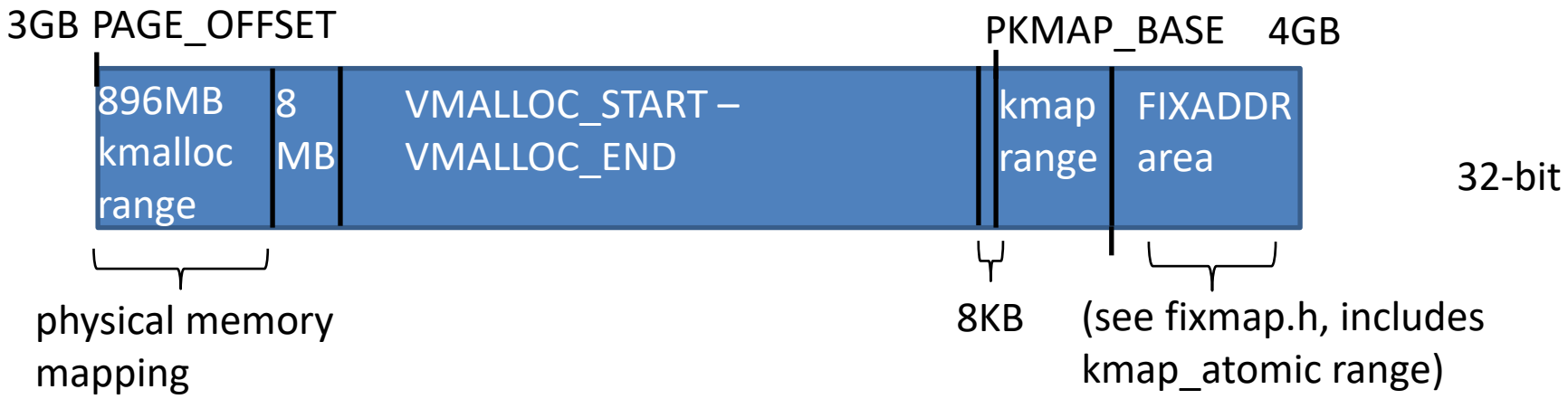
(Unknown Scope)

```
22  #define flush_icache_user_range(vma,pg,adr,len) do { } while (0)
23  #define flush_cache_vmap(start, end)        do { } while (0)
24  #define flush_cache_vunmap(start, end)      do { } while (0)
25
26  #define copy_to_user_page(vma, page, vaddr, dst, src, len) \
27      do { \
28          memcpy(dst, src, len); \
29          flush_icache_user_range(vma, page, vaddr, len); \
30      } while (0)
```

* Call isn't exported to modules! But it's underlying components ARE.

kmap() and friends

- Virtual address for kmap() come from a limited allocation of possible virtual addresses
- kmap() CAN block waiting for a big enough address range to become available
- kmap_atomic () won't block but uses an even smaller range of addresses



kmap(struct page *page)

↓

kmap_high()

↓

map_new_virtual()

highmem.c | highmem_32.c | highmem.h | highmem.h | nommu.c | memory.c | base.c

(Unknown Scope)

```
161 static inline unsigned long map_new_virtual(struct page *page)
162 {
163     unsigned long vaddr;
164     int count;
165
166 start:
167     count = LAST_PKMAP;
168     /* Find an empty entry */
169     for (;;) {
170         last_pkmap_nr = (last_pkmap_nr + 1) & LAST_PKMAP_MASK;
171         if (!last_pkmap_nr) {
172             flush_all_zero_pkmaps();
173             count = LAST_PKMAP;
174         }
175         if (!pkmap_count[last_pkmap_nr])
176             break;  /* Found a usable entry */
177         if (--count)
178             continue;
179
180         /*
181          * Sleep for somebody else to unmap their entries
182          */
183         {
184             DECLARE_WAITQUEUE(wait, current);
185
186             __set_current_state(TASK_UNINTERRUPTIBLE);
187             add_wait_queue(&pkmap_map_wait, &wait);
188             unlock_kmap();
189             schedule();
190             remove_wait_queue(&pkmap_map_wait, &wait);
191             lock_kmap();
192
193             /* Somebody else might have mapped it while we slept */
194             if (page_address(page))
195                 return (unsigned long)page_address(page);
196
197             /* Re-start */
198             goto start;
199         }
200     }
201     vaddr = PKMAP_ADDR(last_pkmap_nr);
202     set_pte_at(&init_mm, vaddr,
203            &(pkmap_page_table[last_pkmap_nr]), mk_pte(page, kmap_prot));
204
205     pkmap_count[last_pkmap_nr] = 1;
206     set_page_address(page, (void *)vaddr);
207
208     return vaddr;
209 }
210
```

Might sleep

Kernel user space access for signals

- This section discusses memory access, not the signal handling subsystem
  - A topic for another investigation…
- Divided into two categories
  - Synchronous: sigaction(), kill(), sigwaitinfo()
    - Standard syscall handling eg copy_from_user()
  - Asynchronous: kernel delivery of a signal to a handler
    - Trickier and architecture specific
    - For x86, starts with entry32.s/entry64.s calls to do_notify_resume() in arch/x86/kernel
- Surprise: The implementation is architecture-dependent

syscall handling

sys_ia32.c | syscalls.h | entry_32.S | syscalls.h | signal.c | signal.c | ia32_signal.c | usercop

(Unknown Scope)

C:\linux-3.7.4\arch\x86\kernel\signal.c

```
559 asmlinkage int
560 sys_sigaction(int sig, const struct old_sigaction __user *act,
561           struct old_sigaction __user *oact)
562 {
563     struct k_sigaction new_ka, old_ka;
564     int ret = 0;
565
566     if (act) {
567         old_sigset_t mask;
568
569         if (!access_ok(VERIFY_READ, act, sizeof(*act)))
570             return -EFAULT;
571
572         get_user_try {
573             get_user_ex(new_ka.sa.sa_handler, &act->sa_handler);
574             get_user_ex(new_ka.sa.sa_flags, &act->sa_flags);
575             get_user_ex(mask, &act->sa_mask);
576             get_user_ex(new_ka.sa.sa_restorer, &act->sa_restorer);
577         } get_user_catch(ret);
578
579         if (ret)
580             return -EFAULT;
581         siginitset(&new_ka.sa.sa_mask, mask);
582     }
583
584     ret = do_sigaction(sig, act ? &new_ka : NULL, oact ? &old_ka : NULL);
585
586     if (!ret && oact) {
587         if (!access_ok(VERIFY_WRITE, oact, sizeof(*oact)))
588             return -EFAULT;
589
590         put_user_try {
591             put_user_ex(old_ka.sa.sa_handler, &oact->sa_handler);
592             put_user_ex(old_ka.sa.sa_flags, &oact->sa_flags);
593             put_user_ex(old_ka.sa.sa_mask.sig[0], &oact->sa_mask);
594             put_user_ex(old_ka.sa.sa_restorer, &oact->sa_restorer);
595         } put_user_catch(ret);
596
597         if (ret)
598             return -EFAULT;
599     }
600
601     return ret;
602 }
```

Some architectures use copy_to/from_user to read and return the sigaction structures.

Copy the sigaction structures into the kernel*

Process the syscall

Copy the old sigaction Structure back to The user

* “the new uaccess exception handling framework” see notes page

Invoking a signal handler

do_notify_resume()

The kernel is about to return to user space and current() is valid

do_signal(struct pt_regs *regs)

Calls get_signal_to_deliver() to retrieve a signal with a handler from the task_struct

handle_signal(unsigned long sig, siginfo_t *info, struct k_sigaction *ka, struct pt_regs *regs)

setup_rt_frame(int sig, struct k_sigaction *ka, siginfo_t *info, struct pt_regs *regs)

static int __setup_rt_frame(int sig, struct k_sigaction *ka, siginfo_t *info, sigset_t *set, struct pt_regs *regs)

Sets up the stack for the task signal handler

-internal.h | signal.c* | entry_32.S | uaccess.h | compat.c | signalfd.c | signal.c | exit.c

known Scope)

```
342 static int __setup_rt_frame(int sig, struct k_sigaction *ka, siginfo_t *info,
343                sigset_t *set, struct pt_regs *regs)
344 {
345     struct rt_sigframe __user *frame;
346     void __user *restorer;
347     int err = 0;
348     void __user *fpstate = NULL;
349
350     frame = get_sigframe(ka, regs, sizeof(*frame), &fpstate);
351
352     if (!access_ok(VERIFY_WRITE, frame, sizeof(*frame)))
353         return -EFAULT;
354
355     put_user_try {
356         put_user_ex(sig, &frame->sig);
357         put_user_ex(&frame->info, &frame->pinfo);
358         put_user_ex(&frame->uc, &frame->puc);
359
360         /* Create the ucontext.  */
361         if (cpu_has_xsave)
362             put_user_ex(UC_FP_XSTATE, &frame->uc.uc_flags);
363         else
364             put_user_ex(0, &frame->uc.uc_flags);
365         put_user_ex(0, &frame->uc.uc_link);
366         put_user_ex(current->sas_ss_sp, &frame->uc.uc_stack.ss_sp);
367         put_user_ex(sas_ss_flags(regs->sp),
368                 &frame->uc.uc_stack.ss_flags);
369         put_user_ex(current->sas_ss_size, &frame->uc.uc_stack.ss_size);
370
371         /* Set up to return from userspace.  */
372         restorer = VDSO32_SYMBOL(current->mm->context.vdso, rt_sigreturn);
373         if (ka->sa.sa_flags & SA_RESTORER)
374             restorer = ka->sa.sa_restorer;
375         put_user_ex(restorer, &frame->pretcode);
376
377         /*
378          * This is movl $__NR_rt_sigreturn, %ax ; int $0x80
379          * WE DO NOT USE IT ANY MORE! It's only left here for historical
380          * reasons and because gdb uses it as a signature to notice
381          * signal handler stack frames.
382          */
383         put_user_ex(*((u64 *)&rt_retcode), (u64 *)frame->retcode);
384     } put_user_catch(err);
385
386     err |= copy_siginfo_to_user(&frame->info, info);
387     err |= setup_sigcontext(&frame->uc.uc_mcontext, fpstate,
388                 regs, set->sig[0]);
389     err |= __copy_to_user(&frame->uc.uc_sigmask, set, sizeof(*set));
```

Gets a frame on the correct stack

And copies the signal info to the stack

nal.c | fpu-internal.h | signal.c | uaccess.h | tlbflush.h | pgtable.c | tlbflush.h

nknown Scope)

```
2664 int copy_siginfo_to_user(siginfo_t __user *to, siginfo_t *from)
2665 {
2666     int err;
2667
2668     if (!access_ok (VERIFY_WRITE, to, sizeof(siginfo_t)))
2669         return -EFAULT;
2670     if (from->si_code < 0)
2671         return __copy_to_user(to, from, sizeof(siginfo_t))
2672             ? -EFAULT : 0;
2673     /*
2674      * If you change siginfo_t structure, please be sure
2675      * this code is fixed accordingly.
2676      * Please remember to update the signalfd_copyinfo() function
2677      * inside fs/signalfd.c too, in case siginfo_t changes.
2678      * It should never copy any pad contained in the structure
2679      * to avoid security leaks, but must copy the generic
2680      * 3 ints plus the relevant union member.
2681      */
2682     err = __put_user(from->si_signo, &to->si_signo);
2683     err |= __put_user(from->si_errno, &to->si_errno);
2684     err |= __put_user((short)from->si_code, &to->si_code);
2685     switch (from->si_code & __SI_MASK) {
2686     case __SI_KILL:
2687         err |= __put_user(from->si_pid, &to->si_pid);
```

Appendices

- How Linux configures page tables for 32/64-bit builds
- vmalloc() and the tlb flush question
- Does this code work?
- Can kernel-mode code survive a page fault?

Page table layout

- Linux uses a 4-level abstraction for its paging table layout, with the page table entry (pte) and the following directory levels:
  - Page global (pgd)
  - Page middle (pmd)
  - Page upper (pud)
  - On an x86-64 machine the bits-per-level are 9-9-9-9
- But 4 levels are not needed for smaller physical addresses such as 32-bit or PAE
- So, how is this handled?

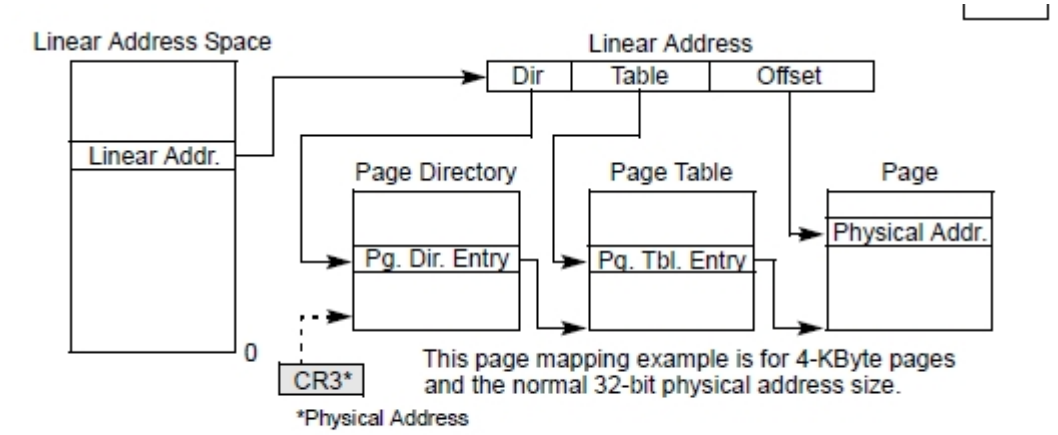


Figure 2-1. IA-32 System-Level Registers and Data Structures

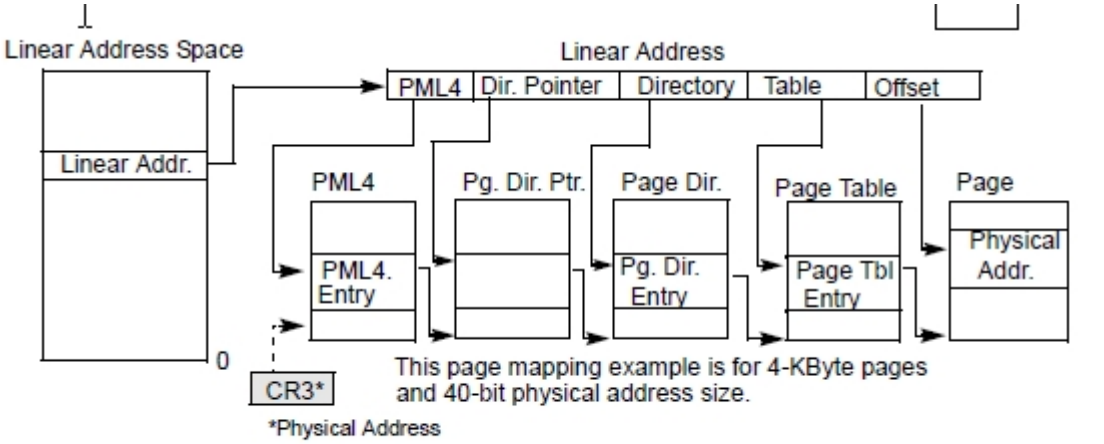


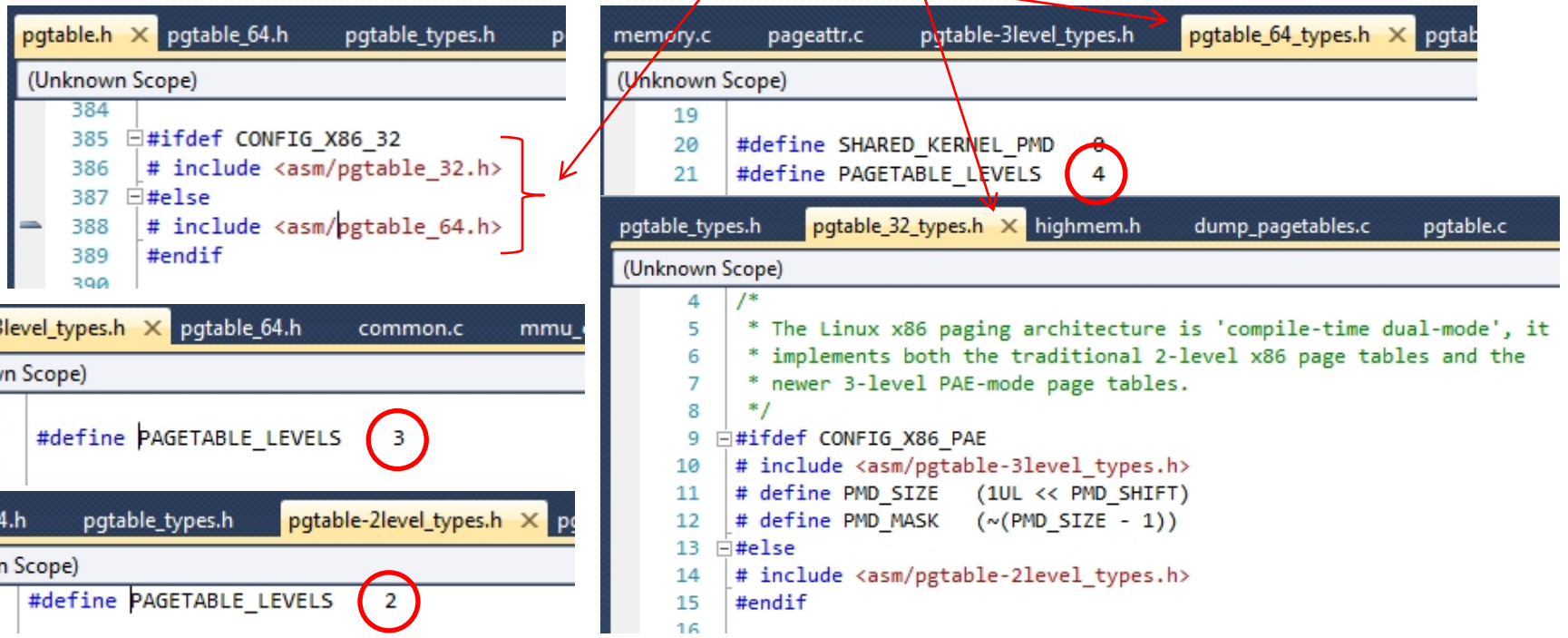
Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode

Figures are from:
Intel® 64 and IA-32 Architectures
Software Developer's Manual vol.3A

How Linux configures page tables for 32/64-bit builds

- Important header files:
  - pgtable.h, pgtable_types.h, pgtable_32_types.h, pgtable_64_types.h

Include asm/pgtable_32_types and asm/pgtable_64_types respectively



PAGETABLE_LEVELS

- Determine whether calls and paging structures exist
- 64-bit is 4 level. For x86_32 there is further selection for levels 2 and 3

pgtable_types.h | pgtable.h | 4level-fixup.h | pgtable-nopud.h

(Unknown Scope)

```
210
211 #if PAGETABLE_LEVELS > 3
212 typedef struct { pudval_t pud; } pud_t;
213
214 static inline pud_t native_make_pud(pmdval_t val)
215 {
216     return (pud_t) { val };
217 }
218
219 static inline pudval_t native_pud_val(pud_t pud)
220 {
221     return pud.pud;
222 }
223 #else
224 #include <asm-generic/pgtable-nopud.h>
225
226 static inline pudval_t native_pud_val(pud_t pud)
227 {
228     return native_pgd_val(pud.pgd);
229 }
230 #endif
```

```
232 #if PAGETABLE_LEVELS > 2
233 typedef struct { pmdval_t pmd; } pmd_t;
234
235 static inline pmd_t native_make_pmd(pmdval_t val)
236 {
237     return (pmd_t) { val };
238 }
239
240 static inline pmdval_t native_pmd_val(pmd_t pmd)
241 {
242     return pmd.pmd;
243 }
244 #else
245 #include <asm-generic/pgtable-nopmd.h>
246
247 static inline pmdval_t native_pmd_val(pmd_t pmd)
248 {
249     return native_pgd_val(pmd.pud.pgd);
250 }
251 #endif
252
```

Folding levels

- How is this used?

pgtable-2level_types.h | pgalloc.h | pgtable-nopud.h | pgtable-nopmd.h

(Unknown Scope)

```
9
10  #define __PAGETABLE_PMD_FOLDED
```

pgtable.h | memory.c | mm.h*

(Unknown Scope)

```
1181 #ifdef __PAGETABLE_PUD_FOLDED
1182 static inline int __pud_alloc(struct mm_struct *mm, pgd_t *pgd,
1183                         unsigned long address)
1184 {
1185     return 0;
1186 }
1187 #else
1188 int __pud_alloc(struct mm_struct *mm, pgd_t *pgd,
1189     unsigned long address);
1190 #endif
1191
1192 #ifdef __PAGETABLE_PMD_FOLDED
1193 static inline int __pmd_alloc(struct mm_struct *mm, pud_t *pud,
1194                         unsigned long address)
1195 {
1196     return 0;
1197 }
1198 #else
1199 int __pmd_alloc(struct mm_struct *mm, pud_t *pud,
1200     unsigned long address);
1201 #endif
```

pgtable-2level_types.h | pgalloc.h | pgtable-nopud.h | pgtable-nopr

(Unknown Scope)

```
5
6   #define __PAGETABLE_PUD_FOLDED
7
```

pgtable.h | memory.c* | mm.h

(Unknown Scope)

```
3584 #ifndef __PAGETABLE_PUD_FOLDED
3585 /*
3586  * Allocate page upper directory.
3587  * We've already handled the fast-path in-line.
3588  */
3589 int __pud_alloc(struct mm_struct *mm, pgd_t *pgd,
3590     unsigned long address)
3591 {
3592     pud_t *new = pud_alloc_one(mm, address);
3593     if (!new)
3594         return -ENOMEM;
3595
3596     smp_wmb(); /* See comment in __pte_alloc */
3597
3598     spin_lock(&mm->page_table_lock);
3599     if (pgd_present(*pgd))  /* Another has populat
3600         pud_free(mm, new);
3601     else
3602         pgd_populate(mm, pgd, new);
3603     spin_unlock(&mm->page_table_lock);
3604     return 0;
3605 }
3606 #endif /* __PAGETABLE_PUD_FOLDED */
```

Question: Will the TLB be flushed after setting up the new page tables for vmalloc()?

Answer: Maybe.

On standard 32-bit or 64-bit systems. vmalloc.c flushes the TLB when FREEING virtual addresses. The newly allocated virtual addresses won't be loaded into the TLB until they are first accessed. But the allocation can force a purge of addresses previously identified as freed ('lazily' freed), resulting in a call to __flush_tlb_all on x86.

There is a call path that starts with map_vm_area() and might get to a call to pud_populate() if there is a PGD (page global directory) entry change and CONFIG_x86_PAE is set *. In this case flush_tlb_mm() is called with &init_mm as the struct mm pointer argument (for the vmalloc() case). This is significant because flush_tlb_mm() will only cause a flush to happen if the mm struct matches current->mm, so no flush will happen for a call from vmalloc().

I was not able to find any other paths from the vmalloc() that might result in a tlb flush. CAVEAT: This is based on the standard unmodified 3.7.4 kernel.

* https://lkml.org/lkml/2011/3/15/516 discusses this: [PATCH]x86: flush tlb if PGD entry is changed in i386 PAE mode

Does this code work?

Yes.

This is code widely used as a teaching example. It allows the driver to blink LEDs on the keyboard by calling into another driver. (But I consider this bogus because the underlying ioctl handler expects a user-mode caller.)

```
struct timer_list my_timer;
struct tty_driver *my_driver;
/*
 * Function my_timer_func blinks the keyboard LEDs periodically by invoking
 * command KDSETLED of ioctl() on the keyboard driver.
 */

static void my_timer_func(unsigned long ptr)
{
    int *pstatus = (int *)ptr;

    if (*pstatus == ALL_LEDS_ON)
        *pstatus = RESTORE_LEDS;
    else
        *pstatus = ALL_LEDS_ON;

    (my_driver->ioctl) (vc_cons[fg_console].d->vc_tty, NULL, KDSETLED,
                        *pstatus);

    my_timer.expires = jiffies + BLINK_DELAY;
    add_timer(&my_timer);
}

static int __init kbleds_init(void)
{
    int i;

....                // DELETED TO FIT PAGE

    my_driver = vc_cons[fg_console].d->vc_tty->driver;
    printk(KERN_INFO "kbleds: tty driver magic %x\n", my_driver->magic);

    /*
     * Set up the LED blink timer the first time
     */
    init_timer(&my_timer);
    my_timer.function = my_timer_func;
    my_timer.data = (unsigned long)&kbledstatus;
    my_timer.expires = jiffies + BLINK_DELAY;
    add_timer(&my_timer);

    return 0;
}
```

http://linux.die.net/lkmpg/x1194.html

But what about this?

The call resolves to a case statement in vt_do_kdskled(). This can ONLY work if the module has a user_mode address to return the value into.

```
struct timer_list my_timer;
struct tty_driver *my_driver;
/*
 * Function my_timer_func blinks the keyboard LEDs periodically by invoking
 * command KDSETLED of ioctl() on the keyboard driver.
 */

static void my_timer_func(unsigned long ptr)
{
    int *pstatus = (int *)ptr;

    if (*pstatus == ALL_LEDS_ON)
        *pstatus = RESTORE_LEDS;
    else
        *pstatus = ALL_LEDS_ON;

    (my_driver->ioctl) (vc_cons[fg_console].d->vc_tty, NULL, KDGETLED,
                        *pstatus);

    my_timer.expires = jiffies + BLINK_DELAY;
    add_timer(&my_timer);
}
```

A kernel-mode address

```
/* the ioctls below only set the lights, not the functions */
/* for those, see KDGKBLED and KDSKBLED above */
case KDGETLED:
        ucval = getledstate();
        return put_user(ucval, (char __user *)arg);

case KDSETLED:
        if (!perm)
                return -EPERM;
        setledstate(kbd, arg);
        return 0;
}
```

So, no, it does not.

Can kernel-mode code survive a page fault?

- According to most classes and books I have seen, no
  - From ‘Essential Linux Device Drivers’: “User mode code is allowed to page fault, however, whereas kernel mode code isn’t.”
- But think about it. What happens when a driver writes to a user-mode address mapped to a page that exists but hasn’t been faulted-in?

The answer is: Maybe

- Consider a module that
  - supports mmap
  - supports the .fault method, and
  - allocates memory that can be mapped
- Assume the user code maps the memory and issues an ioctl to the driver that causes the driver to access that memory using the user mode virtual address
- What happens?

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

    printk(KERN_ALERT "In vmalloc fault for addr %lx\n", vmf->pgoff << PAGE_SHIFT);
    page = vmalloc_to_page((void *)(vmf->pgoff << PAGE_SHIFT));
    if (!page)
        return VM_FAULT_SIGBUS;

    get_page(page);
    vmf->page = page;
    printk(KERN_ALERT "returning page  %x\n", (unsigned int)page);
    return 0;
}


static struct vm_operations_struct mmapper_vmem_ops = {
    .fault = file_vma_fault,
};

int mmap_vmalloc(struct mmapper_dev *thisMPR,
            struct vm_area_struct *vma)
{
        long length = vma->vm_end - vma->vm_start;

        // Restrict it to size of device memory
        if (length > thisMPR->dev_len)
                return -EIO;

        thisMPR->map_addr=vma->vm_start;
    printk(KERN_ALERT "will mmap %p\n", (void *)(vma->vm_start));


    vma->vm_ops = &mmapper_vmem_ops;
    vma->vm_pgoff = (unsigned long) vmalloc_ptr >> PAGE_SHIFT;
#if LINUX_VERSION_CODE >= KERNEL_VERSION(3,4,0)
    vma->vm_flags |= VM_DONTEXPAND | VM_DONTDUMP;
#else
    vma->vm_flags |= VM_DONTEXPAND | VM_RESERVED;
#endif
        printk("Mapping vmalloc_area_ptr: 0x%p \n", vmalloc_ptr);

        return 0;
}
```

4. The fault handler associates the page frame with the faulting address

1. The mmap() handler gets invoked

2. The mmap handler saves and prints the mapped virtual address

3. And sets up the page fault handler

```
//#if LINUX_VERSION_CODE >= KERNEL_VERSION(3,0,0)
static long mmapper_ioctl(
//#else
//static int mmapper_ioctl(struct inode *inode,
//#endif
        struct file *filp,
                unsigned int cmd, unsigned long arg)
{
    long ret;

    struct mmapper_dev *thisMPR = (struct mmapper_dev *)filp->private_data;
    //int ret;

    if (!thisMPR) {
        printk(KERN_ALERT "mmapper ioctl: No device instance for minor %d\n",
            iminor(filp->f_dentry->d_inode));
        return -ENODEV;
    }

        // hack test
        if (thisMPR->cioctls == 1) {
                printk(KERN_ALERT "Testing fault handler: 0x%p \n",
                        (void *)thisMPR->map_addr);

        ret = put_user('X', (char *)(thisMPR->map_addr));
        printk(KERN_ALERT "Testing fault handler: Ret is %d\n", ret);
```

```
[ 1426.121946] In mmapper ioctl cmd:1 - arg: 3215035528
[ 1426.121949] mmapper in ioctl for minor 1, thisMPR f83f708c, use_cnt = 1 ret = 0
[ 1426.121951] will mmap b76db000
[ 1426.121952] Mapping vmalloc_area_ptr: 0xf8519000
[ 1426.121954] mmapper in mmap for minor 1, thisMPR f83f708c, use_cnt = 1 ret = 0
[ 1426.121958] Testing fault handler: 0xb76db000
[ 1426.121960] In vmalloc fault for addr f8519000
[ 1426.121961] returning page  c1ed7aa0
[ 1426.121964] Testing fault handler: Ret is 0
[ 1426.121965] In mmapper ioctl cmd:1 - arg: 3215035528
[ 1426.121966] mmapper in ioctl for minor 1, thisMPR f83f708c, use_cnt = 1 ret = 0
[ 1426.121977] In vmalloc fault for addr f851a000
[ 1426.121978] returning page  c1ed7a80
[ 1426.121980] In vmalloc fault for addr f851b000
[ 1426.121981] returning page  c1efa4e0
[ 1426.121983] In vmalloc fault for addr f851c000
```

```
hacktest
hacktesti ret = 0
mmapper_v mptr is 0xb76db000 for size 16384
mmapper_v mmap:  'Xbcdefghijklmnopqrstuvwxyzabcdefg
```

The ioctl handler overwrites the first character at the saved user virtual address 0xb76db000

The first page faults in and the ioctl completes. When the user accesses the other pages, they fault in individually

dmesg output shows put_user() is successful, as is the ioctl

User program output shows the letter a overwritten with the 'X'