The Memory Management System

The kernel has two substantial jobs to tackle with respect to memory:
• Managing a potentially scarce resource pool of physical page frames.
• Managing virtual address translations both for user processes and kernel space.

In this unit, we will explore both of these issues with respect to the Linux 2.6 kernel running on a 32-bit X86 architecture.

The Page Frame Pool

A computer system has a certain amount of physical RAM. Some of that is taken up statically by the kernel, i.e. kernel text, data and bss. There are additional demands for physical memory:
• Dynamically allocated kernel data structures, e.g. the task_struct, kernel-mode stacks, in-core inodes and directory entries, pending signals, loadable kernel modules, and shared kernel/hardware data structures such as page tables.
• User process address space. We've seen that this can be broken down into anonymous regions, which are not persistent, and file-mapped regions where resident pages correspond to specific portions of specific, persistent files.
• I/O buffers. In some cases the I/O buffer corresponds to a portion of a file, but in other cases (e.g. network packet buffers) it does not.
• Filesystem block caches: Filesystems sometimes need to cache disk blocks which are not the actual contents of a given file, e.g. directory entries.

The demand for page frames ebbs and flows throughout the life of the system. At certain times, there may be abundant page frames which are empty and uncommitted, and therefore available for immediate allocation. At other times, the page frame pool may be depleted, causing pressure for page frames to be freed up to satisfy new demand. The kernel, through the Page Frame Reclamation Algorithm (PFRA), always tries to keep a reasonable number of free page frames, so that an allocation request does not fail or encounter excessive delay.

We will now consider how page frames are allocated by kernel routines, and how those page frames may be subdivided for efficient allocation of smaller objects.

Addressing Issues on X86

Recall that the kernel, in a 32-bit X86 environment, uses the last 1GB of virtual address space for itself, leaving the first 3GB to user-mode. Also recall that all kernel virtual addresses are shared, i.e. there is not a separate kernel virtual address space for each process or component of the kernel. Therefore, it would seem to be impossible for the

kernel to have more than 1GB of internal data structures.

This is addressed through an interesting strategy. Linux further subdivides both physical and kernel virtual space so that the first 896MB of physical page frames map **directly** to the first 896MB of the last GB of virtual address space. This greatly simplifies kernel code as a physical page number can be used directly to compute the virtual address and vice versa. The last 128MB of the kernel's virtual address space is used to establish **temporary mappings** to physical pages which are above 896MB. The kernel calls these two areas of physical memory **zones**, the former being the NORMAL zone and the latter the HIGHMEM zone. This klugery is not required for 64 bit architectures.

When allocating memory for internal data structures, the kernel uses the pool of pages in the NORMAL zone. Since user processes are not confined to the shared 1GB virtual address space, their pages always get pulled from the HIGHMEM zone first, and then, with reluctance, from the NORMAL zone if there is insufficient free memory in the HIGHMEM zone.

Page Descriptor Table

The kernel maintains a single, contiguous array of **page descriptor** structures, each of which represents one page frame:

```
struct page {
         unsigned long flags;                              /* bitwise flags */
         atomic_t _count;                        /* usage counter */
         atomic_t _mapcount;                     /* how many PTE mappings to this page */
         union {
                  unsigned long private;                   // buddy order, et al
                  struct address_space *mapping;           // reverse mapping
         };
         pgoff_t index;                          /* offset of page within mapping*/
         struct list_head lru;                   /* free/active lists */
         void *virtual;                          /* kernel virt address */
};
```

The page descriptor table, called `mem_map` (another term often used in UNIX kernels other than Linux is "core map") consumes a small amount of memory overhead. Specifically, on a 32-bit system, each `struct page` is 32 bytes long, so the overhead is 32/4096 or 0.7%. The page frame number is implicit by the index of the page descriptor within the `mem_map` array. For any given page, if it falls within the low-memory ("NORMAL") zone, the kernel can easily figure out the direct-mapped virtual address. Otherwise, if the page is currently mapped into kernel memory space, the `virtual` field says where.

A given page frame is either mapped (there is at least one PTE in somebody's virtual memory area, including the kernel's, that points to it) or it is free. To save space, some fields of the `struct page` are overloaded with mutually exclusive values. For example, the blind union private/mapping contains either information about the free page

or information about the reverse mapping if it is mapped. The `_mapcount` field will be 0 if the page is free and >0 if the page is mapped somewhere. The confusingly named `_count` field is a temporary usage counter that is incremented when certain memory management routines are doing something with that page, and is used as a form of locking to prevent pages from disappearing while kernel code is playing with them. The `flags` field contains bitwise flags of the form `PG_xxxx` which encode various status values. For example, the `PG_locked` flag is set when an I/O operation is in progress on that page.

Buddy System

The basic kernel page frame allocator is based on asking for one or more contiguous pages. Although most of the time kernel routines are asking for just one page, sometimes the kernel needs a rather large buffer which spans many pages, e.g. when setting up a Direct Memory Access transfer to/from an I/O device.

In general, memory allocation routines (e.g. `malloc()` in user mode) are faced with a tradeoff between space efficiency and time efficiency. If there is a request for say 7 contiguous "units" of memory, it is desirable in terms of space efficiency to satisfy that request with exactly 7 units. However, as memory is allocated and freed, **fragmentation** arises. There may be 3 units free, then 1 occupied unit, then 4 more free units. Even though there are 7 free units, they are not contiguous and thus do not satisfy the request.

In some allocation settings, e.g. on-disk filesystems, it is possible to de-fragment the system by swapping storage locations so as to group clumps of free units together. This is not possible in the C programming environment because once memory has been allocated, its raw address has been passed back as a pointer and there is no way to track down all references (including offset references) to that memory region, and thus no way to change the address of the region.

Given that fragmentation is unavoidable, the next best thing in terms of space efficiency is, given a request for N contiguous units, satisfy it with the smallest available contiguous region of size M units, M>=N. Now this leads to a time efficiency issue: how long will it take to search the list of available memory units to find this "best fit". In the case of the kernel, because memory allocation is a frequent and critical operation, it is desirable that this time be a constant one, not one which grows as the number of distinct free memory units increases because of fragmentation. Therefore, the Linux kernel makes a compromise, and adopts the well-known "buddy system" algorithm to give fairly decent space efficiency and lookup time which, although not strictly constant-time, has a small and fixed upper bound.

The "buddy system" algorithm maintains, for each of the memory allocation zones (e.g. NORMAL and HIGHMEM) an array of 11 pointers. The [0] element of this array points to a circular doubly-linked list of page descriptors for page frames which are free but not

contiguous with other free pages. The [1] element points to such a list for groups of 2 contiguous pages. The [2] element does the same for groups of 4 contiguous pages, and so on up to the [10] element which collects groups of 1024 contiguous pages, i.e. 4MB. These lists are chained through the `lru` field of the page descriptor (which has other uses when the page frame is not free) and furthermore the `private` field of the first page descriptor in a contiguous chunk holds the order of that chunk. Subsequent page descriptors in the chunk are not part of the list but their existence is inferred by their position in the `mem_map`.

When a routine in the kernel wants say one page frame, the buddy system tries to take that from the [0] order list of free page frames. If there are none, it grabs a chunk of two page frames from the [1] list and splits it, handing one back to the caller and moving the other to the [0] list. If the [1] list is empty, this process continues moving up the ladder, e.g. taking a clump of 4 contiguous frames, returning 1 to the caller, and splitting the remainder into a single free frame and a pair of frames.

When a page frame or contiguous group of frames is de-allocated by the kernel to be placed back in the free pool, the "buddy system" checks to see if the freed page(s) is/are a contiguous "buddy" of another free page. If so, they are merged to form a higher-order contiguous region. This process can continue up to order 10, i.e. coalescing into a chunk of 1,024 free pages.

Note that the buddy system trades CPU time efficiency for absolute memory efficiency, by introducing an additional restriction. A group of pages on the, e.g. [2] list, i.e. a group of 4 contiguous pages, will always start with a page number divisible by 4. If say page frame numbers 1024-1027 inclusive are free, these can be used to satisfy a request for 4 contiguous pages. However, if 1025-1028 inclusive are free, this does not satisfy the request, because the start of the free area, 1025, is not a multiple of 4. Likewise for all other groupings from 2 to 1024 pages.

Given a page frame number `a`, its order-`N` buddy with page frame number `b` therefore satisfies the expression:

```
b==a ^ (1<<N)
```

Any page frame number `a` of order-`N` has a parent of order `N+1` which lives at:

```
p=a & ~ (1<<N)
```

These simple bit-arithmetic relationships allow the deallocation process to be performed very quickly, without extensive data structures manipulation and resulting memory accesses.

Slab Allocator

The kernel subdivides page frames using a system called the **Slab Allocator** for small objects (e.g. allocating a new `task_struct`). This system is fairly complicated and will not be fully described here. The kernel grabs page frames and uses them as slabs,

storing an internal descriptor at the beginning of the slab to track how the page frame or frames is being subdivided. Because the kernel is a tightly-controlled piece of code and the type of data structures which will need to be allocated and released frequently is understood in advance, the slab allocator can be optimized so that it can quickly allocate and free objects of a given type.

When the kernel requires space for a new object of a certain type, it consults the slab associated with that type. The slab descriptor bitmap quickly determines if there is a free slot open, and if so where. If there are no free slots, the slab allocator goes back to the page frame allocator and asks for an additional page (not necessarily contiguous) which is subdivided into slots.

Likewise, de-allocation of an object allocated through the slab allocator means simply noting in the descriptor that the slot is free. If all slots in the page frame are free, the page frame itself is released to the free pool.

Managing Virtual Address Space

The kernel has the following challenges in managing the virtual address space of its processes:
• Laying out the virtual address space within the confines of the address size. On a 32-bit architecture, this means positioning the text, data, bss, stack, dynamic library and any mmap memory regions so they do not interfere with each other.
• When a page fault occurs, determining the mapping to backing store and paging-in the contents of the page (if needed).
• Handling copy-on-write and auto-grow regions.
• Determining when a protection or page fault is actually the result of errant program operation, and generating a signal.
• Coordinating the correspondence between memory-mapped file regions and access to file contents through the traditional read/write system calls.
• Keeping track of access frequency and selecting candidates for page reclamation.
• Knowing the reverse mapping from a physical page to the page table entry or entries which reference it, and consequently the file and offset of backing store, if applicable.
• Knowing when a page is "dirty" and arranging for it to be "sync’d" to backing store (either a file or swap space) before being reclaimed.
• Managing the backing store for anonymous pages ("swap space")

Virtual Addressing on the X86

The Linux kernel, being portable to many different architectures, supports up to a 4-level page structure. The highest level page table, the address of which is set by a CPU

register, is the Page Global Directory (PGD). Entries in the PGD are in the same format as our conceptual PTE entries, however the physical address field now contains the physical page frame number of the next level of page table. This level is called the Page Upper Directory (PUD). PUD entries, in turn, point to Page Middle Directory (PMD) tables, which finally point to the lowest-level Page Tables containing the PTEs ultimately used for translation.

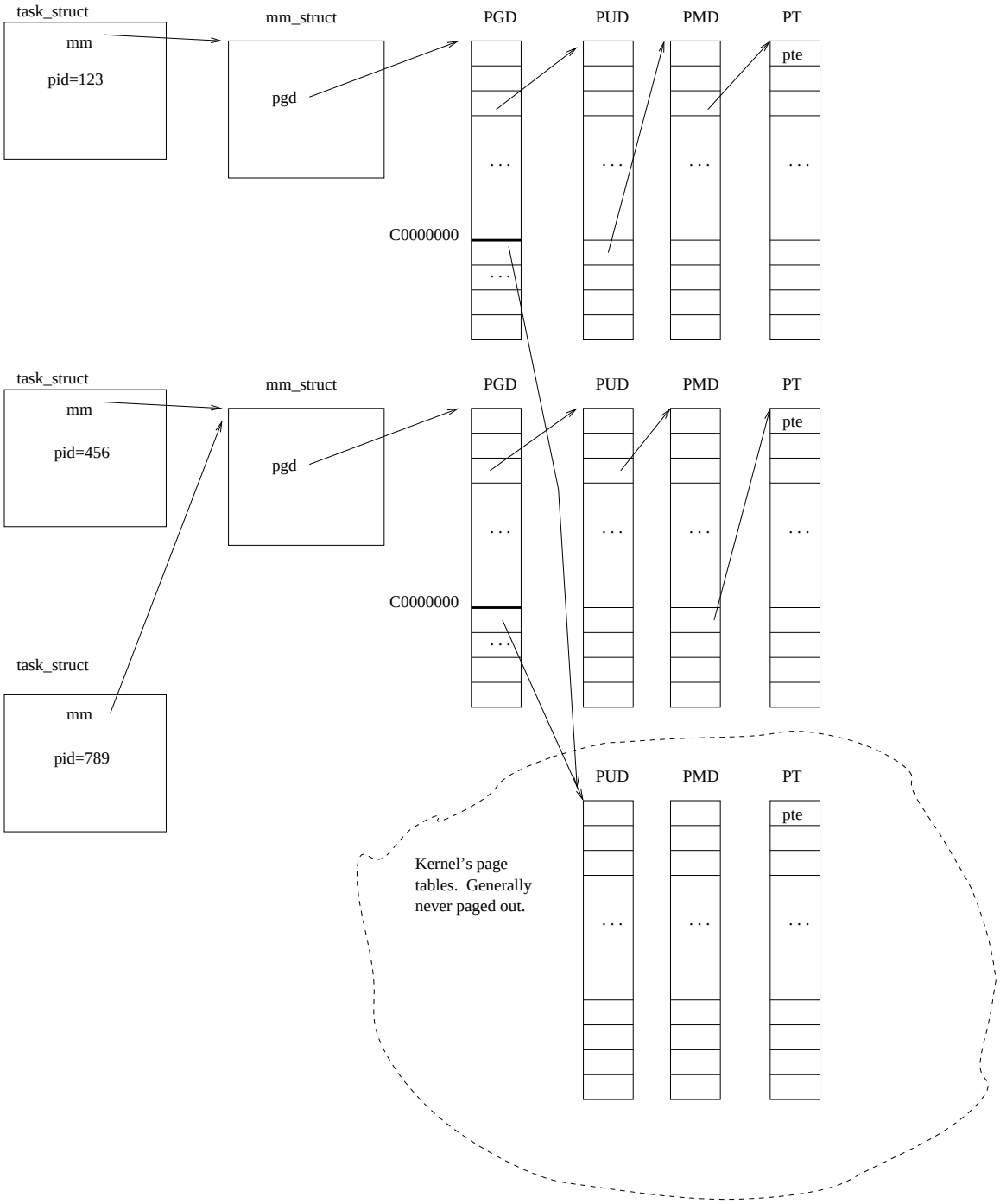
When running on the X86-32 architecture, the Linux kernel simply ignores the PUD and PMD, as the X86 is a 2-level paging structure. The most significant 10 bits of the 32-bit virtual address index the PGD. Each entry of the PGD spans 2^22 or 4MB of virtual address space and there are 1024 such entries. The PGD entries point to Page Tables, each containing 1024 entries spanning 4K each. The processor control register `%cr3` contains the *physical address* of the PGD, and is adjusted by the kernel whenever it is changing address spaces during a context switch.

Since a typical process address space is sparse, with large areas of undefined address space, many of the PGD entries will be empty (Present bit cleared). Recall that virtual address space (in a 32 bit X86 architecture) contains 3GB of user space, and the last 1GB is reserved for kernel address space and is common to all processes. Therefore, only one set of page tables needs to be maintained for kernel virtual addresses, and the last 256 PGD entries of each process point to these shared page tables. Kernel page tables are always resident to minimize the places where kernel code can encounter a page fault.

Furthermore, page tables (PUD/PMD/PT) for processes are not created in advance. They are dynamically allocated on demand through the page fault mechanism as the process uses its address space. This further reduces the overhead of the page tables.

Under the X86 64 bit architecture, registers are 64 bits wide. Addresses however are actually only 48 bits wide, and are sign-extended into the remaining 16 bits. The Linux kernel continues to use a 4K page size under this architecture, leaving 48-12=36 bits for the page number. These 36 bits are evenly divided 9|9|9|9. Thus the PGD has 512 entries, each pointing to a 512 entry PUD, pointing to a 512 entry PMD, and finally pointing to a Page Table containing 512 PTEs.

The diagram below depicts three tasks, (123/456/789) of which the last two share an entire address space (so they are two threads within the same multi-threaded process). Anywhere in kernel code (other than in an asynchronous interrupt context) `current->mm->pgd` contains the *virtual address* by which the PGD can be accessed.



Representing the process address space

The Linux kernel uses a number of data structures to keep track of the virtual address space of each process. As you should recall, a `task_struct` is used to represent each process (task) on the system. It contains pointers to subsidiary structures for various

aspects of the tasks's operating system context, among which is a pointer to a `struct mm_struct`, which is the top-level structure for representing the tasks's entire virtual address space. Normally there is one `mm_struct` for each task, but in multi-threaded programs, each new thread is spawned by means of the `clone` system call with the `CLONE_VM` flag set. In this case, multiple tasks will point to the same `mm_struct`.

```
struct mm_struct {
  struct vm_area_struct * mmap;                 /* head of list of regions */
  struct rb_root mm_rb;                         /* red-black tree for vma */
  struct vm_area_struct * mmap_cache;  /* last find_vma result */
  unsigned long (*get_unmapped_area)();         /* method to search for avail vma */
  void (*unmap_area)();                         /* method to release vm area */
  unsigned long mmap_base;             /* lowest virt addr */
  pgd_t * pgd;                                  /* ptr to Page Global Dir */
  atomic_t mm_users;                            /* How many users with user space? */
  atomic_t mm_count;                            /* mm_users + in-kernel users */
  int map_count;                       /* number of regions */
  struct rw_semaphore mmap_sem;                 /* protects list of regions */
  spinlock_t page_table_lock;                   /* Protects page tables and some counters */
  struct list_head mmlist;             /* linkage in list of all mm_struct */

  mm_counter_t _file_rss;              /* current RSS file-backed pages */
  mm_counter_t _anon_rss;              /* current RSS anon pages */
  unsigned long hiwater_rss;           /* High-watermark of RSS usage */
  unsigned long hiwater_vm;            /* High-water virtual memory usage */

  unsigned long total_vm;              /* size of entire vm space, in pages */
  unsigned long locked_vm;             /* number of locked-down pages */
  unsigned long shared_vm;                      /* number of shared mapped file pgs */
  unsigned long exec_vm;               /* number of executable pages */
  unsigned long stack_vm;              /* num pages in user-mode stack */
  unsigned long reserved_vm;           /* num special/reserved pages */
  unsigned long def_flags;             /* default access mode flags */
  unsigned long nr_ptes;               /* number of PTEs in use */
                  /* The following give the start and end of common regions */
  unsigned long start_code, end_code, start_data, end_data;
  unsigned long start_brk, brk, start_stack;
  unsigned long arg_start, arg_end, env_start, env_end;

  unsigned long saved_auxv[AT_VECTOR_SIZE]; /* for /proc/PID/auxv */

  /* Architecture-specific MM context */
  mm_context_t context;

  /* Token based thrashing protection. */
  unsigned long swap_token_time;
  char recent_pagein;
  /* ... some other stuff elided */
};
```

While that is quite a complicated structure, we see that it describes the lowest address of the virtual address space in question, as well as the start and end points of important parts

of memory. The mm_struct is on a global doubly-linked list of all such address spaces. Finally, hanging off the mm_struct is a singly-linked list of `struct vm_area_struct`, each of which describes one memory region:

```
struct vm_area_struct {
  struct mm_struct * vm_mm;  /* The address space we belong to. */
  unsigned long vm_start;    /* Our start address within vm_mm. */
  unsigned long vm_end;              /* The first byte after our end address */

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

  pgprot_t vm_page_prot;             /* PTE flags for this VMA */
  unsigned long vm_flags;            /* Flags, listed below. */

  struct rb_node vm_rb;                         /* Red/black trees */

  /* union below used to provide reverse mapping: given a particular
         range of bytes in a particular file, find all memory regions
         which map that part of the file */
  union {
         struct {
                  struct list_head list;
                  void *parent;      /* aligns with prio_tree_node parent */
                  struct vm_area_struct *head;
         } vm_set;

         struct raw_prio_tree_node prio_tree_node;
  } shared;

  /*
   * A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma
   * list, after a COW of one of the file pages.  A MAP_SHARED vma
   * can only be in the i_mmap tree.  An anonymous MAP_PRIVATE, stack
   * or bss vma (with NULL file) can only be in an anon_vma list.
   */
  struct list_head anon_vma_node;     /* list of shares */
  struct anon_vma *anon_vma; /* mapping for anonymous regions */

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

  /* Information about our backing store: */
  unsigned long vm_pgoff;            /* Offset in pages within file */
  struct file * vm_file;             /* File we map to (can be NULL). */
  void * vm_private_data;
  unsigned long vm_truncate_count;    /* truncate_count or restart_addr */
};
```

Because each `vm_area_struct` structure contains the start and end addresses of each memory region, the kernel during a page fault can quickly check a given virtual address and see if it is within any defined region, or falls outside of it. [Normally, in the latter case, the process will be given a SIGSEGV, however there is one important exception

noted below under "Stacks".] Because this is a frequently-performed operation, the Linux 2.6 optimizes this lookup by sorting the `vm_area_struct` structures using a binary search tree, which is kept balanced using a "red-black" tree coloring algorithm. The tree is rooted in the `mm_struct->mm_rb` and each VMA is represented by `vm_area_struct->vm_rb`.

You will also note that the protections associated with the region (read/write/execute) are stored in the `vm_area_struct`. From time to time, a page may be mapped in the region with a page table entry (PTE) that has a more restrictive protection. In particular, a page may be marked as read-only when it really should be read-write, in order to force "copy on write" (see later section). The `vm_page_prot` field describes the PTE entries which should be set when pages are mapped in to this region, while the bits in in `vm_flags` denote the "true" properties of the region, some of which are similar to the flags passed with the `mmap` system call (this is a partial list):

```
VM_READ             Read permission
VM_WRITE            Write permission
VM_EXEC             Execute permission
VM_SHARED           Region may be shared among multiple processes
VM_GROWSDOWN        Region automatically grows towards lower addresses
VM_EXECUTABLE       Region maps to an executable file
VM_LOCKED           Region should be locked in memory and never swapped out
VM_IO               Special region for memory-mapped I/O
VM_DONTCOPY         Discard this region when forking instead of copying
VM_DONTEXPAND       Disallow expansion of region with mremap
VM_DENYWRITE        Disallow over-write of active text pages
```

The `mmap` system call simply creates a new `vm_area_struct` and links it into the `struct mm`, after of course verifying the system call parameters, etc. Note that `mmap` does not actually map any pages in. It just establishes the validity of the new memory region. The actual mapping-in is performed by the page fault handler, to be discussed below. The `mremap` system call adjusts the start address and/or size of an existing region, and `mprotect` adjusts the flags. `munmap` deletes an existing region from the virtual memory space. Each of these system calls has an internalized version which is used within the kernel, e.g. during `exec` to clear out the old virtual address space and create the new address space for the new program.

Address space release (munmap)

When a `vm_area` is destroyed via the munmap operation, either via the system call or the internal version which is called during exec or exit, the kernel visits the page tables which are spanned by that region. For any PRESENT PTE, the PFN gets us to the struct page. The `mapcount` is decremented. If this was the last mapping to the page and the count is now 0, the page is moved to the free list. This process is recursive. If an entire page table worth of PTEs have been freed up, the page table itself can be returned to the page frame pool. Finally when the entire process address space is free, the PGD is freed.

Copy-On-Write and Fork

When the virtual address space is copied during fork, the kernel copies makes a deep copy of the struct mm in the parent to form the memory map of the child. Each PTE in the parent is duplicated to the child, and if the page in question is Present, the reference count of the corresponding struct page is also incremented.

For mappings that will be copy-on-write, the W bit is turned off in both the original parent PTE and the new child PTE. The Linux kernel also turns off the A bit in the child PTE. For regions that have the VM_SHARED property (i.e. MAP_SHARED memory mapped regions) the D bit is also turned off in the child. This is to avoid biasing the PFRA.

Page Fault Handling

The page fault exception is used by the CPU to indicate an access to an invalid or un-translated virtual address. However, this is usually not a real problem. The kernel enjoys handling page faults, and uses page faults to implement a number of features. On the X86-32 architecture, both page and protection faults come in as the same interrupt vector.

The kernel's page fault handler knows:
• The identity of the faulting process (`current` global variable)
• The attempted virtual address.
• The type of access (read/write).
• Whether the virtual page in question had a valid PTE
• The registers at the time of fault (including user/supervisor flag).

The page fault handler can have one of two general outcomes: Either the page fault is resolved, resulting in the establishment of a valid PTE (which has the appropriate R/W/X permissions for the access being attempted) pointing to a valid page frame with the correct data in it, or the page fault can not be resolved, resulting in the delivery of a SIGSEGV, SIGBUS or SIGKILL to the process. Let's take a look at the page fault handler code:

```
// from /usr/src/linux/arch/x86/mm/fault.c
/* do_page_fault is called from the assembly language entry.S
   file with some fairly complex assembly code.  The regs argument
   points to the saved registers area on the kernel mode stack, and
   error_code is the hardware fault code:
 *      bit 0 == 0 means no page found, 1 means protection fault
 *      bit 1 == 0 means read, 1 means write
 *      bit 2 == 0 means kernel, 1 means user-mode
 *      bit 3 == 1 means use of reserved bit detected
 *      bit 4 == 1 means fault was an instruction fetch
 */
```

```
fastcall void do_page_fault(struct pt_regs *regs,
                                    unsigned long error_code)
{
        struct task_struct *tsk;
        struct mm_struct *mm;
        struct vm_area_struct * vma;
        unsigned long address;
        int write, si_code;
        int fault;

        /* get the faulted address from the CR2 register*/
        address = read_cr2();
        tsk = current;
        si_code = SEGV_MAPERR;                  // default signal reason

        if (address >= 0xC0000000) {
                   /*... in here we check for bad accesses from kernel mode */
                   /*... if it is user mode, then this is always a bad access */
                goto bad_area_nosemaphore;
        }
        mm = tsk->mm;

          /* Some code elided here to avoid handling page faults during an
                   interrupt handler, atomic operation, or if this is
                   a kernel-mode task with no user address space */
          /* This next passage obtains the r/w lock on the process */
          /* address space.  See text about "exception tables" */
        if (!down_read_trylock(&mm->mmap_sem)) {
                if ((error_code & 4) == 0 &&             // Kernel mode
                    !search_exception_tables(regs->eip))
                        goto bad_area_nosemaphore;
                down_read(&mm->mmap_sem);
        }

          /* When we get here, we own the mmap reader/writer lock.  No other
          /* kernel paths can mess with the address space (e.g. a new mmap
          /* system call) until we are done */

          // find_vma finds memory region immediately above address, or
          // possibly including address.
        vma = find_vma(mm, address);
        if (!vma)           // Address is higher than end of highest region
                goto bad_area;
          // Note that vma could still refer to a region which is completely
          // above the fault address
        if (vma->vm_start <= address)
                goto good_area;
          if (!(vma->vm_flags & VM_GROWSDOWN))
                   goto bad_area;
          // If we get here, this is a stack (growsdown) memory region
        if (error_code & 4) {                   // in user mode
                   // Normally accessing an address below the current %esp
                   // register value is an error.  This next line provides
```

```
                    // an exception for entering a function with a large
                    // local variable space
                if (address + 65536 + 32 * sizeof(unsigned long) < regs->esp)
                        goto bad_area;
        }
          // If we get here, the faulted address is "near" the esp register
          // Try to expand the vma just above the fault to cover the fault
          // expand_stack could fail (e.g. out of memory, per-process
          // stack size limit reached
        if (expand_stack(vma, address))
                goto bad_area;
good_area:
          // We know here that the fault address lies within a valid region
        si_code = SEGV_ACCERR;                  // new default signal reason
        write = 0;
        switch (error_code & 3) {
                default:        /* 3: write, present */
                                /* fall through */
                case 2:         /* write, not present */
                        if (!(vma->vm_flags & VM_WRITE)) //write to r/o area,
                                goto bad_area;           // no good
                        write++;
                        break;
                case 1:         /* read, present */
                        goto bad_area;          // Some other prot fault???
                case 0:         /* read, not present */
                        if (!(vma->vm_flags & (VM_READ | VM_EXEC | VM_WRITE)))
                                goto bad_area;
        }

 survive:
          // handle_mm_fault attempts to resolve the page fault
          // It can return VM_FAULT_OOM, VM_FAULT_SIGBUS, VM_FAULT_MAJOR,
          // VM_FAULT_MINOR.  See text for discussion
        switch (handle_mm_fault(mm, vma, address, write)) {
                    case VM_FAULT_MINOR:
                             tsk->min_flt++;
                             break;
                    case VM_FAULT_MAJOR:
                             tsk->maj_flt++;
                             break;
                    case VM_FAULT_SIGBUS:
                             goto do_sigbus;
                case VM_FAULT_OOM:
                        goto out_of_memory;
                    default:
                             BUG();
        }

          // Fault has been resolved successfully.
          // Finally, we can release the mmap reader lock and return, which
          // causes control to return via assembly language to the
          // point of interruption (i.e. IRET instruction )
```

```
        up_read(&mm->mmap_sem);
        return;

bad_area:
          // Since we have determined that the attempted access was at
          // an invalid address, we don't care about the process address
          // space anymore, and can release the reader lock
        up_read(&mm->mmap_sem);

bad_area_nosemaphore:
        if (error_code & 4) {                   //Fault was in User Mode
                   /*...*/
                tsk->thread.cr2 = address;
                /* Kernel addresses are always protection faults */
                tsk->thread.error_code = error_code | (address >= 0xC0000000);
                tsk->thread.trap_no = 14;

                   /* Post a signal to the process.  The siginfo will
                    be filled in from the information in tsk->thread */
                force_sig_info_fault(SIGSEGV, si_code, address, tsk);
                return;                 // return to user mode
        }

no_context:
        /* See discussion in text about exception tables and "fixup"
        if (fixup_exception(regs))
                return;
          /*... Unhandled fault in kernel mode.  The current task is
                   terminated (even if SIGSEGV handler is established)
                   and a nasty message is printed to the console */
          do_exit(SIGKILL);             //Doesn't return

out_of_memory:
          // See text discussion about over-commit and what it means
          // to REALLY run out of memory.  If we reach this point, we
          // have little choice except to KILL the current process
        up_read(&mm->mmap_sem);
        if (is_init(tsk)) {  // Can't kill the init process !
                yield();
                down_read(&mm->mmap_sem);
                goto survive;                           // try again for a page frame
        }
        printk("VM: killing process %s\n", tsk->comm);
        if (error_code & 4)             // User Mode
                do_exit(SIGKILL);
        goto no_context;

do_sigbus:
        up_read(&mm->mmap_sem);

        /* Kernel mode? Handle exceptions or die */
        if (!(error_code & 4))          // Fault was in Kernel Mode
                goto no_context;
```

```
          // Post a SIGBUS to the process
        tsk->thread.cr2 = address;
        tsk->thread.error_code = error_code;
        tsk->thread.trap_no = 14;
        force_sig_info_fault(SIGBUS, BUS_ADRERR, address, tsk);
          return;            // return to user mode
}
```

We see that `do_page_fault` looks at the faulting address and determines if it falls within a valid memory region. If it does, and if the fault is not a "true" protection violation (e.g. writing to a region which does not have PROT_READ/VM_READ permission), the page fault is not a program error, but a benign part of normal virtual memory operations. We'll consider benign fault resolution shortly.

Stacks

When the stack grows e.g. through a push instruction, the stack pointer is decremented first and then the write takes place. If the stack pointer is now lower than the current start of the stack region, a fault will be incurred. If the faulted address is "close" to the stack pointer (an allowance is made for certain X86 instructions which access memory on the stack first, then adjust the stack pointer) and if the memory region just above the fauled address has the GROWSDOWN property, then `expand_stack` is called. This code (not shown) checks to see if the new stack size falls within ulimit resource limits, and if so the limits of that region are adjusted (as if `mremap` had been called) and the page fault resolution algorithm continues, considering the address to be a valid one now. This automagical stack growth is quite essential to transparent program execution.

Invalid Addresses and SIGSEGV

If, however, the faulted address is outside of all regions and is not associated with stack growth, or if the attempted access is a violation of the regions protections, a signal `SIGSEGV` is posted to the process using the kernel function `force_sig_info`. When a signal is posted in this manner, the kernel doesn't care if the process has that signal blocked or ignored: The user program attempted to access an illegal virtual memory address, and there is no point in letting it continue on its current path. The address will not magically become valid, and therefore the faulted instruction can never be satisfied and would loop forever.

Therefore, if SIGSEGV (or SIGBUS, or whatever other signal is being posted with `force_sig_info`) is currently set to SIG_IGN, the disposition is changed to SIG_DFL. The default action for these signals is to cause the process to exit, with a core dump.

If the signal has a handler, but the signal is currently being blocked, the signal is unblocked, the handler is reset to SIG_DFL, and once again the signal kills the process.

The reasoning is that a handler should not be invoked if the corresponding signal is being blocked.

It is, however, permissible for the signal to be *handled* as that will force execution to jump to another, presumably less flawed part of the program. This technique is often used in complex, modular applications (such as the open-source image editor *The Gimp*) to isolate a flawed module and prevent it from crashing the entire program and possibly leading to a loss of data.

Invalid addresses in Kernel Mode / The fixup table

Normally an invalid access while in kernel mode is a sign of a kernel bug, and page faults do not occur in kernel mode. Any kernel data structures are always mapped in. However, there are times when the kernel needs to access user memory, e.g. getting to the buffer in read and write system calls. The approach that the Linux kernel takes is to limit these accesses to a small number of locations within the kernel code. Each of these accesses is surrounded by special macros which record the program counter location associated with the access in a "fixup exceptions table". If a page fault arrives in kernel mode, and the program counter (`%eip`) matches one of these addresses, then it is a benign page fault which the kernel resolves. Otherwise, the kernel assumes that things have gone wrong, and it terminates the current process with SIGKILL and prints a nasty message to the console.

Therefore, we can conclude that an interrupt or exception handler will never be interrupted by an exception handler, other than this one circumstance: a system call interrupted by a page fault. Interrupt handlers must never cause page faults.

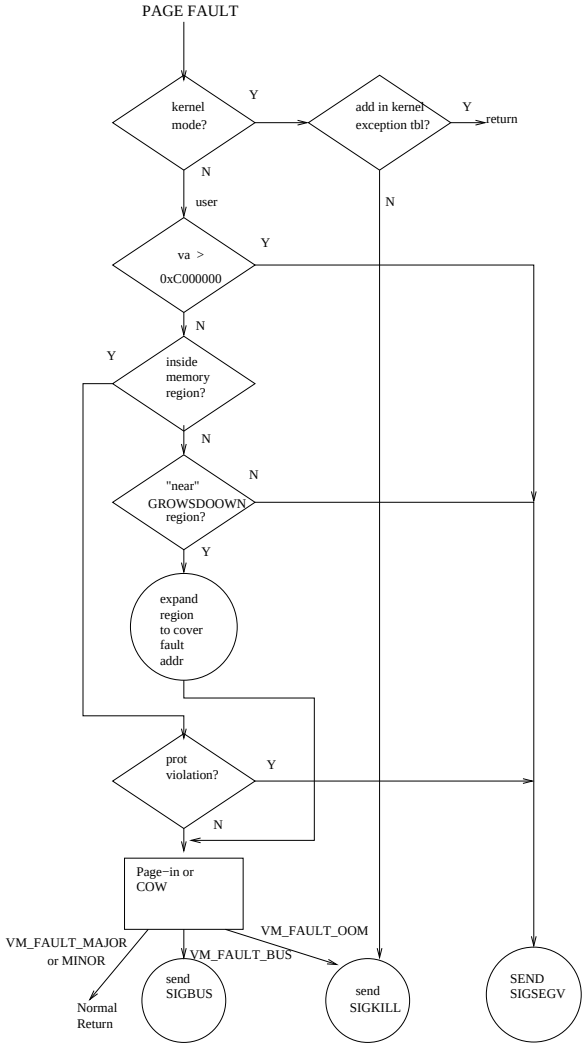
Benign Page Faults / Paging-in

Page faults which are within an existing region and which are not the result of protection violations are therefore not the result of any program error, and lead to the kernel routine `handle_mm_fault`. This routine can have one of four outcomes:

- `Minor fault`: The page fault is resolved quickly without requiring a disk I/O operation or putting the process to sleep. E.g. demand-paging new bss pages (just allocate a page, zero fill and return), paging-in file-mapped pages which are still cached in memory, or breaking up a Copy-on-Write shared page frame. Minor faults are good.
- `Major fault`: It was necessary to sleep pending an I/O operation to satisfy this fault. Major faults are not necessarily evil (e.g. demand paging an executable file from disk) but we would like to minimize their number.
- `Bus Error`: The kernel was unable to page-in the page because of an error other than being out of memory. E.g. the page is mapped to a disk file but an I/O error occurred trying to access the disk, or the mapping is no longer valid because the file has been truncated. A SIGBUS will be delivered to the process.

• `Out of Memory`: The kernel ran out of memory to allocate a free page frame or an internal data structure. This is very bad. Under ordinary circumstances, the kernel tries to prevent this from happening by maintaining a pool of free memory. If this does happen, the kernel has no choice but to terminate the current process. See also discussion below on "over-commit".

The Linux page fault handling algorithm is summarized by the following simplified flowchart:



Paging In Logic

When a page is not present, the PRESENT bit of the PTE is clear, and all other bits are don’t-cares, as far as the hardware is concerned. The Linux kernel overloads the remaining 31 bits of the PTE to encode certain information that is helpful for resolving the page fault, without having to rely on additional data structures. Let’s look at the

paging-in code:

```
// From /usr/src/linux/mm/memory.c
/* Some locking issues elided in handle_mm_fault and handle_pte_fault */

int handle_mm_fault(struct mm_struct *mm, struct vm_area_struct *vma,
                unsigned long address, int write_access)
{
        pgd_t *pgd;
        pud_t *pud;
        pmd_t *pmd;
        pte_t *pte;

        __set_current_state(TASK_RUNNING);

        count_vm_event(PGFAULT);

          /* PGD is always allocated.  */
        pgd = pgd_offset(mm, address);
          /* Allocate PUD corresponding to Virt Addr "address",
                    if not already allocated.  This can fail causing
                    VM_FAULT_OOM indication */
        pud = pud_alloc(mm, pgd, address);
        if (!pud)
                return VM_FAULT_OOM;
          /* Likewise for the PMD and the lowest-level Page Table */
          /* Recall that this is generic code...on the x86-32 architecture */
          /* there is no PUD or PMD, and pud_alloc/pmd_alloc are no-ops */
        pmd = pmd_alloc(mm, pud, address);
        if (!pmd)
               return VM_FAULT_OOM;
        pte = pte_alloc_map(mm, pmd, address);
        if (!pte)
                return VM_FAULT_OOM;

          /* Now pte points to the PTE for address */
        return handle_pte_fault(mm, vma, address, pte, pmd, write_access);
}


static inline int handle_pte_fault(struct mm_struct *mm,
                struct vm_area_struct *vma, unsigned long address,
                pte_t *pte, pmd_t *pmd, int write_access)
{
        pte_t entry;

        entry = *pte;
        if (!pte_present(entry)) {     // Check PRESENT bit
                if (pte_none(entry)) {// PTE bits are all 0
                   // This is the first time this virt page is being accessed
                        if (vma->vm_ops) {      //File-mapped
                                if (vma->vm_ops->fault || vma->vm_ops->nopage)
                                        // Read it in from a file
```

```
                                    return do_linear_fault(mm, vma, address,
                                            pte, pmd, write_access, entry);
                                // Don't worry about the nopfn stuff
                            if (vma->vm_ops->nopfn)
                                    return do_no_pfn(mm, vma, address, pte,
                                                     pmd, write_access);
                        }
                             // Allocate new zero-filled page for write_access
                             // or COW map to shared zero page for read access
                        return do_anonymous_page(mm, vma, address,
                                                 pte, pmd, write_access);
                }
                   /* Elided code on non-linear mappings */
                   // Must be anon page previously saved to swap space
                return do_swap_page(mm, vma, address,
                                    pte, pmd, write_access, entry);
        }

          // If we are here, the page is present, must be prot fault
          /* ... */
        if (write_access) {
               if (!pte_write(entry))           // WRITE PTE bit clear
                 {
                             /* COW: Allocate new page frame, memcpy old contents
                                into new pg frame, change faulting PTE to point
                                to new pg frame and turn on write-enable bit */
                        return do_wp_page(mm, vma, address,
                                          pte, pmd, ptl, entry);
                   }
                   // The following has to do with architectures that don't
                   // automatically update the DIRTY bit of the PTE on write
                   // access.  On these architectures (not X86) the kernel marks
                   // down pages to no permission to force page faults and
                   // allow it to determine which pages are being written to
                   // and/or read from.
                entry = pte_mkdirty(entry);     // set DIRTY bit
        }
          // Similar code for read access case
          entry=pte_mkyoung(entry);             // set ACCESSED bit
        return 0;
}
```

On-demand creation of Page Tables

Recall that the structure of the page table is multi-level and sparse. Only the top-level page global directory is pre-allocated. The kernel uses **lazy allocation** in creating the page upper directories, page middle directories, and page tables. Therefore, at this time, the kernel must allocate the directories/tables mapping the faulted virtual memory address, if they are not already allocated. When new page tables or page directories are allocated, they are set to 0.

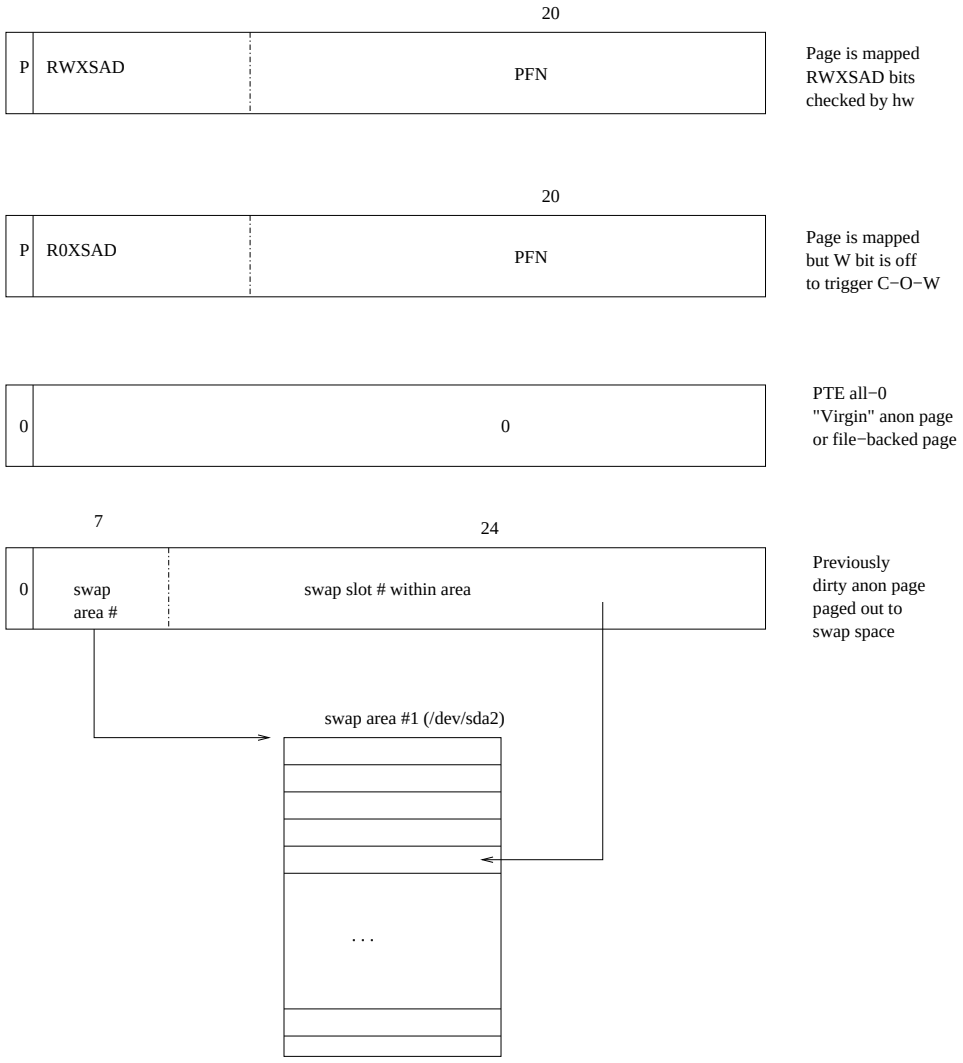
Paging-in from where?

To complete page fault resolution, the kernel must locate the canonical copy of the data corresponding to the virtual page which is faulted. In brief, this can fall into one of the following cases:

- The virtual page is file-mapped.
- The virtual page is anonymous and not previously written to.
- The virtual page is anonymous and previously dirtied, and swapped out.
- The virtual page is already present in physical memory, and this is a copy-on-write.

We will now discuss in detail each of these cases, among which `handle_pte_fault` discriminates by looking at the PTE and the attributes of the corresponding `vm_area_struct`:

- The PFN field is all zero and the Present bit is 0. If `vm_ops` is defined in the memory region descriptor in question, then this is a **file-mapped page**, i.e. the image of the page exists in the filesystem, and one of the page-in methods (e.g. `nopage`) will be called to provide it. This may result in an I/O operation (major page fault) to read the corresponding 4K chunk of the file in from disk, or that chunk may already be cached in physical memory (minor fault). See "File Mappings"
- The PFN field and Present bit are all 0 and there are no page-in methods associated with the memory region. This means that the page is anonymous but no swapped-out image of it exists. Either this virtual page has never before been accessed, or it was a zero-fill page that was never written to. This situation is resolvable as a minor fault: The kernel finds a page frame from the free pool and maps it to the faulted virtual page. If the faulted access was a write, the kernel 0-fills the page. However, for a read access, the kernel optimizes by mapping to a single, shared physical page containing all 0, setting the PTE protection bits to read-only. At a later time, if the process writes to this page, things will be straightened out by Copy-On-Write. This behavior of **zero-filling** new anonymous pages is very important for security. Otherwise, sensitive information from another process could be viewed when the page frame is recycled.
- The page frame number field of the PTE is non-zero, but the Present bit is 0 (and the Dirty bit is also 0). The kernel deliberately sets up this odd situation to indicate an anonymous page which has been `swapped out`. The page frame number does not correspond to a real address, but is used as an identifier to help the swap subsystem find the image of the page in the swap area. This will be discussed further under "Swap Space".
- The Present bit is 1. At this point in the code, we know that it must be a hardware protection violation which tripped the page fault, and we have previously checked that the attempted access is valid for this region. The kernel creates this contradiction to implement Copy-On-Write.



Copy-On-Write (COW)

There are several cases where the kernel must logically make a copy of a page. One common example is during `fork()`. Another is when a file is mapped MAP_PRIVATE. Actually allocating new page frames and copying, byte-for-byte, the entire contents of each page frame of the region being copied, would be a terrible performance penalty. Instead, the kernel continues to point both virtual mappings to the same page frame, but marks down the PTE entries in question to disallow write access.

As long as none of the processes sharing the mapping attempt to write, there is no need to do anything further. They will each see the correct data which are unchanged from the point of forking, in the first example case, or from the original contents of the file, in the second case. As soon as any process tries to write, it will be faulted. At this point the kernel, in routine `do_wp_page()` grabs a free page frame, does the memcpy of the old

page frame to the new, and turns on the write-enable bit of the PTE of the faulted process. The other process(es) sharing that mapping continue to see it read-only.

One important optimization on the above happens when a process forks and then either the parent or the child (but usually the child in the common UNIX programming idiom) execs. Then the shared memory region is unmapped from one of the processes. As we have seen, the kernel maintains a core map data structure giving the state of each page frame, and can track how many references there are to any given page frame. So the unmapping results in the reference count going back down to 1. When `do_wp_page()` sees this condition, it does not have to allocate a new page frame and memcpy. It merely changes the PTE to turn the write enable bit back on

For MAP_PRIVATE regions we have a hybrid. Consider, for example, the data region of a running program, which is mapped MAP_PRIVATE, PROT_READ|PROT_WRITE to an offset within the executable file where the data initializers are to be found. The region is write-enabled but the PTEs are marked down to force Copy-On-Write. If the process only reads a particular page in the data region, then that page is Demand-Paged in. Multiple processes running the same executable can share the mapping to the physical page frame containing the data. However, as soon as the process tries to write to a data region page, Copy-On-Write kicks in. A private copy is made for the writing process.

While the original page frame still corresponds to the contents of the file, and thus its reverse mapping remains file-backed, the private copy has broken the mapping, and thus is now considered anonymous. If this private copy were later to be paged-out, it will therefore wind up in swap space. The data region of the process, and indeed any writable MAP_PRIVATE region, can therefore contain both file-mapped and anonymous page frames at the same time. We'll see how the `mapping` field of the page descriptor is used to distinguish, on a page frame by page frame basis, this status.

Page Frame Reclamation

We can think of the pool of page frames as a cache for actively-used data which are otherwise stored on disk (in a file or in swap space). Linux uses a strategy of maintaining a pool of free page frames proactively. Consider what would happen if this strategy were not in place. We call the total number of page frames mapped to a process the **Resident Set Size (RSS)**. As processes request virtual memory and are allocated page frames, their RSS would grow, and the pool of free frames would shrink and shrink. Eventually, there will not be any frames available to satisfy a new request. This could block the process for a long time while a "victim" page is selected to be written back to backing store and re-assigned.

Instead, the Linux kernel uses a kernel task (*a kernel task aka kernel thread is like a user-mode process, but it never leaves kernel mode. It can make system calls, using a slightly*

modified, internalized syntax. Kernel tasks are scheduled by the scheduler, can go to sleep and wakeup, etc. They do not have their own mm_struct but "borrow" one from a user process, which is OK since they only access the shared kernel memory area) called `kswapd` to periodically scan the `mem_map`, looking for possible victims from which to steal back pages and thus reduce the total RSS. There are two parameters which can be tuned by the system administrator establishing low-water and high-water marks for free pages. `kswapd` works hard to keep the free page count above low-water, and goes to sleep when it reaches the high-water mark. To be more precise, each memory zone (e.g. NORMAL, HIGHMEM) has its own high and low water marks, but we will ignore this detail.

The Page Frame Reclamation Algorithm (PFRA) maintains two lists of non-free page frames: active and inactive. These lists are chained via the `lru` field of the page descriptor. Periodically, the PFRA scans the page descriptors on the active list with the desire to find hypoactive pages and move them to the inactive list. It uses the reverse mapping (see below) in the page descriptor to find the PTEs mapping to this physical page (when the page is shared there will be multiple PTE pointing to the same physical page). Each PTE is examined to see if the ACCESSED bit is set, then the ACCESSED bit is cleared. The PFRA also keeps a second bitwise flag, `PG_referenced`, in the `flags` field of the page descriptor, which stores the previous value of the ACCESSED PTE flag. (when there are multiple PTEs mapping the same page, PG_referenced is the OR of the ACCESSED PTE bits).

When an page which had `PG_referenced` set to 0 is visited again and the ACCESSED PTE flag(s) is (are all) still clear, it means the page has not been used in two successive scan passes. The PFRA will then move the page to the tail of the *inactive list*.

After the PFRA finishes hunting for active pages to move to the inactive list, it starts trying to pull victims off the inactive list, starting at the head of the list. I.e. it starts with the pages which have been sitting on the inactive list the longest. This is known as the **Least Recently Used (LRU)** ordering. The theory is that pages which have remained inactive for a long time will probably continue to be inactive.

For each inactive page scanned, the ACCESSED bit(s) is (are) checked again. Note that this bit is only cleared when the PFRA has visited the page from the *active* list, therefore if there has been any access whatsoever between the time that the page was moved to inactive and the time it was visited on the inactive list, this PTE bit will be set, and the page is just moved back to the active list. Otherwise, the page will be reclaimed. If the PTE indicates that the page is dirty, a write-back to backing store is begun. This operation might take a while, so `kswapd` continues to look at additional pages that are not dirty, until it has reclaimed the desired number of pages. Pages which are not dirty can be reclaimed by simply immediately unmapping the PTE(s). Pages which were in the process of being written back are unmapped when that writeback operation concludes.

Pages that the PFRA has reclaimed are then placed on the free list. Since they are placed at the tail of the free list (for a given buddy order) and the page allocator pulls from the front, this further enhances the notion of Least Recently Used. Pages which are on the free list and haven't been re-used but still contain valid images can be pulled back off so that a page fault can be resolved as a minor fault.

The PFRA is adaptive and if it feels that it isn't reclaiming enough memory, it increases the rate at which it scans active pages, thus making the activity timeout window shorter and increasing the pool of potential inactive pages. This is a delicate balancing act because the PFRA/`kswapd` consumes CPU resources in doing all of this scanning. Excessive scanning wastes CPU time, but insufficient reclamation can cause memory starvation and severe performance penalties.

Like scheduling, memory allocation and reclamation algorithms are the subject of extensive research and development.

Reverse Mapping

For any given page frame, there may be multiple mappings for it. The kernel must keep track of all of them and prevent inconsistent views. For example, a given page may appear as part of the virtual address space of one or more processes, correspond to a particular offset in a particular memory-mapped file, and furthermore correspond to a particular offset within the hard disk storing that file. The PFRA, when visiting page frames, must be able to find all existing mappings of that page so that it can locate and possibly modify the PTEs.

Either a page is mapped to a file, or it is anonymous. These two conditions are mutually exclusive. The `mapping` field of the page descriptor contains a pointer to one of two objects which will help with the reverse mapping. If the least significant bit is 1, the page is anonymous and `mapping &~01` points to a `struct anon_vma`. If the lsb is 0, the page maps to a file and `mapping` points to a `struct address_space` object for the file. This use of the least significant bit is a programming trick: because kernel pointers are always aligned on 4-byte boundaries, the least significant bit can be wasted for this purpose.

Anonymous Mappings

The `anon_vma` is simply the head of a doubly-linked circular list of memory region descriptors (`vm_area_struct`), chaining through the `anon_vma_node` pointers in the latter. Furthermore, the `vm_area_struct` contains a pointer (`anon_vma`) back to the `anon_vma` list head. More often than not, the list contains just one memory region descriptor. However, when a region becomes shared (e.g. after a fork) this list contains each region in each process which refers to it. This will also be the case for

MAP_SHARED regions that were inherited via fork. Therefore, each `anon_vma` structure represents one anonymous virtual memory region (which might be shared among multiple processes).

A process data region or other MAP_PRIVATE file mappings might find itself on the `anon_vma` list and also part of a file mapping, because some pages in the region still correspond to the file, but some have been written to making them anonymous. Within this region, mapped-in pages are either "virgin" and still correspond to the original file, or they have been dirtied and are now backed by swap space. This distinction is captured via the least significant bit of the `mapping` field of the page descriptor as described above.

The PFRA, after getting the `anon_vma` list start from the page descriptor, can walk the linked list anchored by the `struct anon_vma` and visit each memory region (`vm_area_struct`) which maps that page, which gives the starting virtual address of the that region. The `index` field of the page descriptor gives the offset, in pages, of that particular page within the region. The `pgd` field of the `struct mm_struct`, which in turn is accessible from the vm_area_struct, gives the address of the page global directory for the process. It is now a trivial matter to walk through the page table structure and find the PTE.

Swap Space and anonymous page-outs

When an anonymous page is selected for reclamation, the Dirty bit of the PTE is examined. If set, then the page has been written to and therefore must be saved to **swap space**. The kernel maintains one or more swap areas, which are either raw disk partitions, or files, in either case prepared with the system utility command `mkswap`, and then made available as part of swap space with the `swapon` command.

Anonymous pages aren't meant to be persistent, and so swap space is just a pool of disk space where we can stash them. There is nothing within a swap area which would tell one the identities of the page images. That information is kept by the kernel in RAM and simply discarded when the system halts. The kernel maintains an array of swap area descriptors, each of which includes a free map. Each slot of a free map gives the reference count of the corresponding page-sized slot of the corresponding swap area.

A particular page frame containing an anonymous page may be mapped by multiple PTEs in one or more processes. Therefore, when that page frame is swapped out, ALL of the PTE must be marked as not present. A free slot in some swap area is chosen (the algorithm to do so keeps things such as disk I/O performance in mind). Free slots are indicated by a swap map entry of 0. Once the slot is allocated, the swap map entry will reflect the number of mappings that had been pointing at that shared page frame.

Now the Linux kernel uses a slightly dirty trick. The swap area number (a 7-bit integer) combined with the slot number within the area form a unique identifier of the swap slot.

24 bits are allowed for the slot number, meaning the maximum size of one swap area is 2^24*2^12=2^36 or 64 GB. The total amount of swap space on the system is limited to 2^(36+7) = 8 TB.

Given this information, the page fault handler can, when the page is once again referenced, find the corresponding swap slot on disk and read it in. This swap identifier is stored in each PTE which had mapped the swapped-out page. There are 31 bits in the identifier and a PTE is 32 bits. The Linux kernel packs the bits into the PTE in such a way that the Present flag is always 0, and the remaining 31 bits are scattered between the PFN and the Flags fields. It isn't a valid PTE but since the Present bit is OFF, the hardware will never look at it or touch it.

File Mappings

For files which are mapped into memory, it is important to be able to locate quickly the page frame which currently contains a particular (page-aligned) offset in the file, and/or to determine that no such page is currently resident. Consider the `read` system call, the semantics of which must work correctly and transparently with memory-mapped files. `read` amounts to finding the page frame which holds that part of the file (if there is no such frame, ask the filesystem to bring it in from disk and sleep until that happens), and then copying the requested portion of that page into the user buffer.

Likewise, when faulting-in a memory-mapped file, it could be that the same (page-aligned) area of the file has recently been accessed from the same or different process using the read or write system call, in which case the fault can be resolved a minor fault by pointing the PTE at the existing page frame which contains that image.

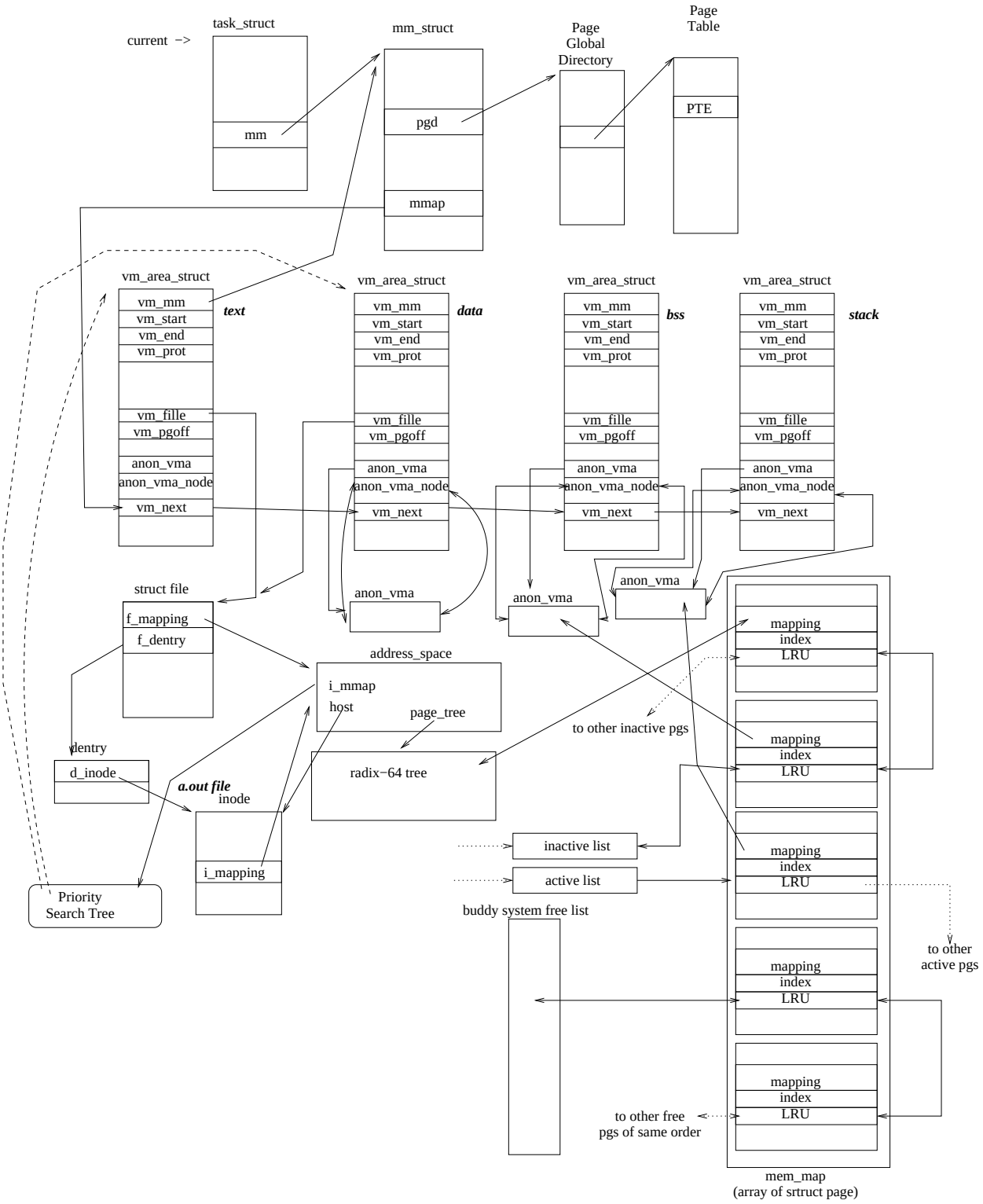
An `address_space` structure is used to provide this quick lookup. One of these is allocated for every open inode. It has many fields, but among them are two types of trees. One is a flexible radix-64 tree which, given an offset (in page-sized units) into the file, finds the struct page corresponding to that part of the file, or NULL if there is none such. Each node of the radix-64 tree has 64 pointers. If the file size is 256K or less (64*4096) then the entire file's address space is handled by a single-level tree. Otherwise, the tree grows. A two-level tree can handle (64*64*4096) or 16MB of file addressing, etc. up to 6 levels.

```
struct address_space {                  /* /usr/src/linux/include/linux/fs.h */
        struct inode            *host;          /* owner: inode, block_device */
        struct radix_tree_root  page_tree;      /* radix tree of all pages */
        spinlock_t              tree_lock;      /* and lock protecting it */
        unsigned int            i_mmap_writable;/* count VM_SHARED mappings */
        struct prio_tree_root   i_mmap;         /* tree of private and shared mappings */
        spinlock_t              i_mmap_lock;    /* protect tree, count, list */
        unsigned long           nrpages;        /* number of total pages */
          /* some other fields elided for simplicity */
```

```
};
```

While the radix tree is great for going from a known offset in the file to a page descriptor, the PFRA is trying to go a different way: find all PTEs which refer to this particular offset within the file. The `address_space` pointer of the `struct page` identifies the file, and the `index` field gives the offset in pages within the file. A simple approach would be to link all regions mapping to the file into a linked list, as is done for the `anon_vma` structure. However, while sharing of an anonymous region is usually limited to a small number of processes, memory-mapped files, especially those containing a common executable, are shared by many processes. There can also potentially be multiple mappings to different parts of one file (consider an a.out file where the .text and .data section are mapped separately).

Linux uses a Priority Search Tree for *each file*. It is access via the `address_space` structure's `i_mmap` pointer, and can quickly answer the following question: "given a particular offset within the file, tell me all memory regions which contain it." I.e. the PST sorts the mapping regions by their starting offset. The PFRA examines all memory region descriptors which map to the page. Each descriptor contains the starting virtual address of the region, and the corresponding offset within the file (this is in the `vm_pgoff` field). Now the offset of the page within the region can be readily calculated, and from there the PTE can be found.


task_struct
current ->
mm
mm_struct
pgd
mmap
Page
Global
Directory
Page
Table
PTE
vm_area_struct
vm_mm
vm_start
vm_end
vm_prot
vm_fille
vm_pgoff
anon_vma
anon_vma_node
vm_next
text
data
bss
stack
struct file
f_mapping
f_dentry
anon_vma
address_space
i_mmap
host
page_tree
radix−64 tree
dentry
d_inode
a.out file
inode
i_mapping
Priority
Search Tree
inactive list
active list
buddy system free list
mapping
index
LRU
to other inactive pgs
to other
active pgs
to other free
pgs of same order
mem_map
(array of srtruct page)

Resource Exhaustion

In an ideal world, systems would always have sufficient resources to handle the jobs which are thrown at them. But that world is not a terribly interesting one in terms of operating systems design. In reality, memory resource exhaustion is often a problem. We can consider two distinct problem areas: Insufficient physical memory, and insufficient swap space.

Thrashing / Swap Token

When memory becomes very scarce, the system can spend much of its time freeing up pages by swapping-out, followed by swapping-in when a process which has had its pages stolen gets scheduled and tries to run. This is an age-old problem known as **thrashing**. Linux tries to ameliorate it by using what Linux calls a "swap token". A process which holds the swap token is given a temporary reprieve by the PFRA. The swap token is given to a process based on heuristic rules...basically a process which has recently been victimized by the PFRA, has good priority and looks like it will do something useful if allowed to run.

This isn't necessarily a perfect solution. To contrast, the Solaris kernel takes a different approach. When memory runs low, a kernel thread called the "swapper" picks entire processes to victimize and steals all of the process' pages. A process thus swapped-out is basically put to sleep and therefore doesn't get scheduled. When things get calmer, the process is released from the swapper's grasp and is allowed to run and fault its pages back in.

Memory over-commit and the out-of-memory killer

Just like an airline which over-books seats, it is possible for the kernel to hand out more virtual memory than it has physical memory and swap to cover. If it happens that all of the memory is demanded at once and there are insufficient free swap slots to page-out to, the kernel will have no choice but to kill processes in order to regain memory. There is an Out-Of-Memory killer which is triggered when the system becomes desperately low of memory. It picks large processes which haven't been running too long.

This might seem like a design flaw. After all, the programmer successfully allocated memory using malloc (which in turn used `brk` or `mmap`), so why has the kernel gone back on its promise and why should a process be terminated because of something which is not its fault?

On the other hand, the most conservative approach, which guarantees that one will never run out of memory, is to reserve a swap space slot for each virtual anonymous page allocated. Then there will always be a place to page-out any resident page and we can

never get into this trouble. The problem with this approach is that processes will start getting memory allocation failures long before the system has reached a high probability of running out of memory, unless the administrator has configured a very large amount of swap space. The reason: most virtual anonymous pages are never actually paged-out. Consider a large process which forks and then execs. Most of the virtual address space of the child will be discarded.

Older Linux kernels took the first approach. Solaris kernels tend towards to more conservative approach. Modern (2.6+) Linux kernels allow the administrator to tune the memory over-commit policy.

```
The following is verbatim from \fCDocumentation/vm/overcommit-accounting\fP:

The Linux kernel supports the following overcommit handling modes

0       -       Heuristic overcommit handling. Obvious overcommits of
                address space are refused. Used for a typical system. It
                ensures a seriously wild allocation fails while allowing
                overcommit to reduce swap usage.  root is allowed to
                allocate slighly more memory in this mode. This is the
                default.

1       -       Always overcommit. Appropriate for some scientific
                applications.

2       -       Don't overcommit. The total address space commit
                for the system is not permitted to exceed swap + a
                configurable percentage (default is 50) of physical RAM.
                Depending on the percentage you use, in most situations
                this means a process will not be killed while accessing
                pages but will receive errors on memory allocation as
                appropriate.

The overcommit policy is set via the sysctl 'vm.overcommit_memory'.

The overcommit percentage is set via 'vm.overcommit_ratio'.

The current overcommit limit and amount committed are viewable in
/proc/meminfo as CommitLimit and Committed_AS respectively.

Gotchas
-------

The C language stack growth does an implicit mremap. If you want absolute
guarantees and run close to the edge you MUST mmap your stack for the
largest size you think you will need. For typical stack usage this does
not matter much but it's a corner case if you really really care

In mode 2 the MAP_NORESERVE flag is ignored.


How It Works
```

```
------------

The overcommit is based on the following rules

For a file backed map
        SHARED or READ-only     -       0 cost (the file is the map not swap)
        PRIVATE WRITABLE        -       size of mapping per instance

For an anonymous or /dev/zero map
        SHARED                  -       size of mapping
        PRIVATE READ-only       -       0 cost (but of little use)
        PRIVATE WRITABLE        -       size of mapping per instance

Additional accounting
        Pages made writable copies by mmap
        shmfs memory drawn from the same pool
```