Paging

Advanced Operating Systems

Overview

- Page fault handling
- VMAs
- Memory mappings
- User-level page fault handling
- Transparent huge pages

Page Faults

- Page fault: a special kind of exception
  - Interrupt vector 14
- Why page fault?
  - The MMU wants (software) attention!
  - Key to implement demand paging
- Page fault types:
  - Invalid: Access to an invalid address
    - e.g., write to a read-only page
  - Major: Access is valid, but page is not in memory
    - e.g., file or swapped pages
  - Minor: Access is valid and page is in memory
    - e.g., first access to anonymous memory

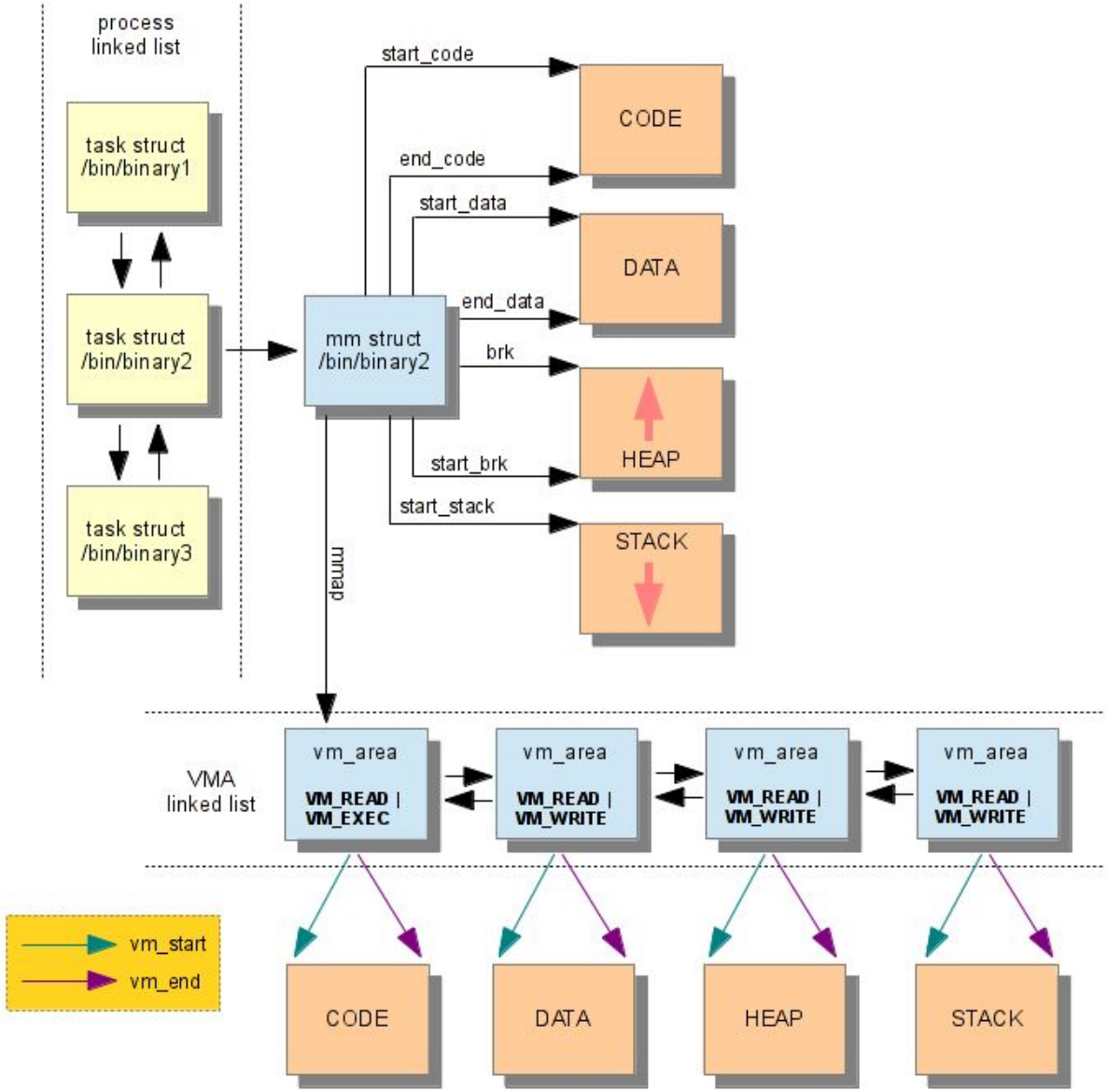
Page Faults: Kernel vs. User

Kernel page fault

- Invalid:
  - Kernel address: oops!
  - User address: checks (e.g., `copy_to/from_user`)
- Major: if kernel is paged out (e.g., Windows)
- Minor: lazy allocations (e.g., old vmalloc)

User page fault

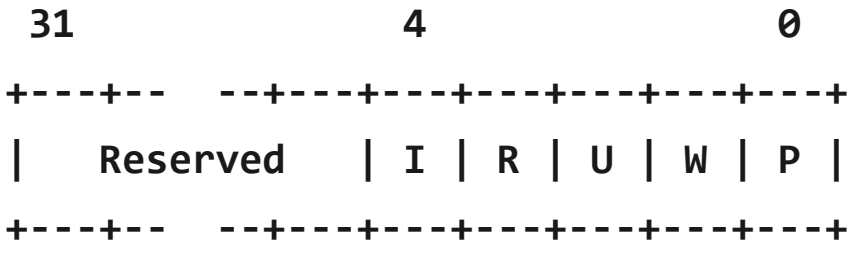
- Everything goes!
- How can we figure out what happened?



MM Data Structures in the Kernel

Page Faults: Fault Info

- How to retrieve page fault information ?
  - CR2 register contains the faulting virtual address
  - The error code on the stack contains:



    - P = PTE Present Bit Set
    - W = Write Access
    - U = User Access
    - R = PTE Reserved Bits Set
    - I = Instruction Fetch Access

Page Faults: User Thread Info

- How to retrieve the faulting user thread?
  - Get the current `task` (`get_current()`)
- Generic implementation:

```
 union thread_union {
#ifndef CONFIG_ARCH_TASK_STRUCT_ON_STACK
    struct task_struct task;
#endif
#ifndef CONFIG_THREAD_INFO_IN_TASK
    struct thread_info thread_info;
#endif
    unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

Page Faults: User Thread Info

- How to retrieve the faulting user thread?
  - Get the current `task` (`get_current()`)
- x86:

```
DECLARE_PER_CPU(struct task_struct *, current_task);

static __always_inline struct task_struct *get_current(void)
{
    return this_cpu_read_stable(current_task);
}

#define __percpu_seg      gs
```

Page Faults: User Address Space Info

- How to retrieve the faulting address space?
  - Get task's `mm` (`get_current()->mm`)

```
struct mm_struct {
  struct vm_area_struct *mmap; /* list of VMAs */
  struct rb_root mm_rb;
  u32 vmacache_seqnum;         /* per-thread cache */
  unsigned long mmap_base;     /* base of mmap area */
  pgd_t * pgd;
  atomic_t mm_users;           /* users including user space */
  spinlock_t page_table_lock;  /* Protects page tables */
  struct rw_semaphore mmap_lock;
  struct list_head mmlist;     /* List of mm's. */
  unsigned long start_code, end_code, start_data, end_data;
  unsigned long start_brk, brk, start_stack;
  struct linux_binfmt *binfmt;
};
```

Page Faults: VMA Info

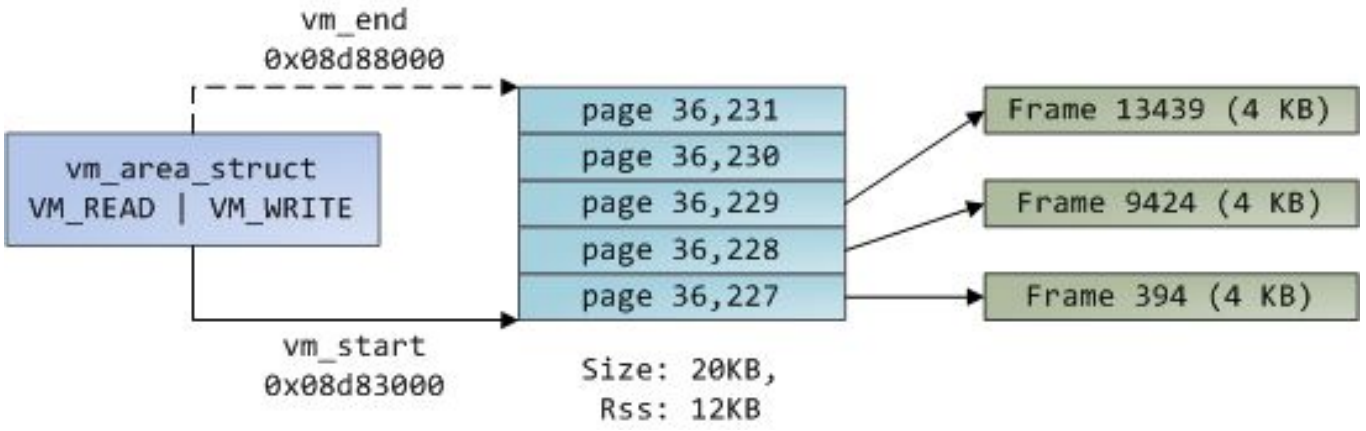
- How to retrieve the faulting VMA?
  - Contiguous virtual region with given properties
  - Look up on rb tree (`find_vma(mm, address)`)

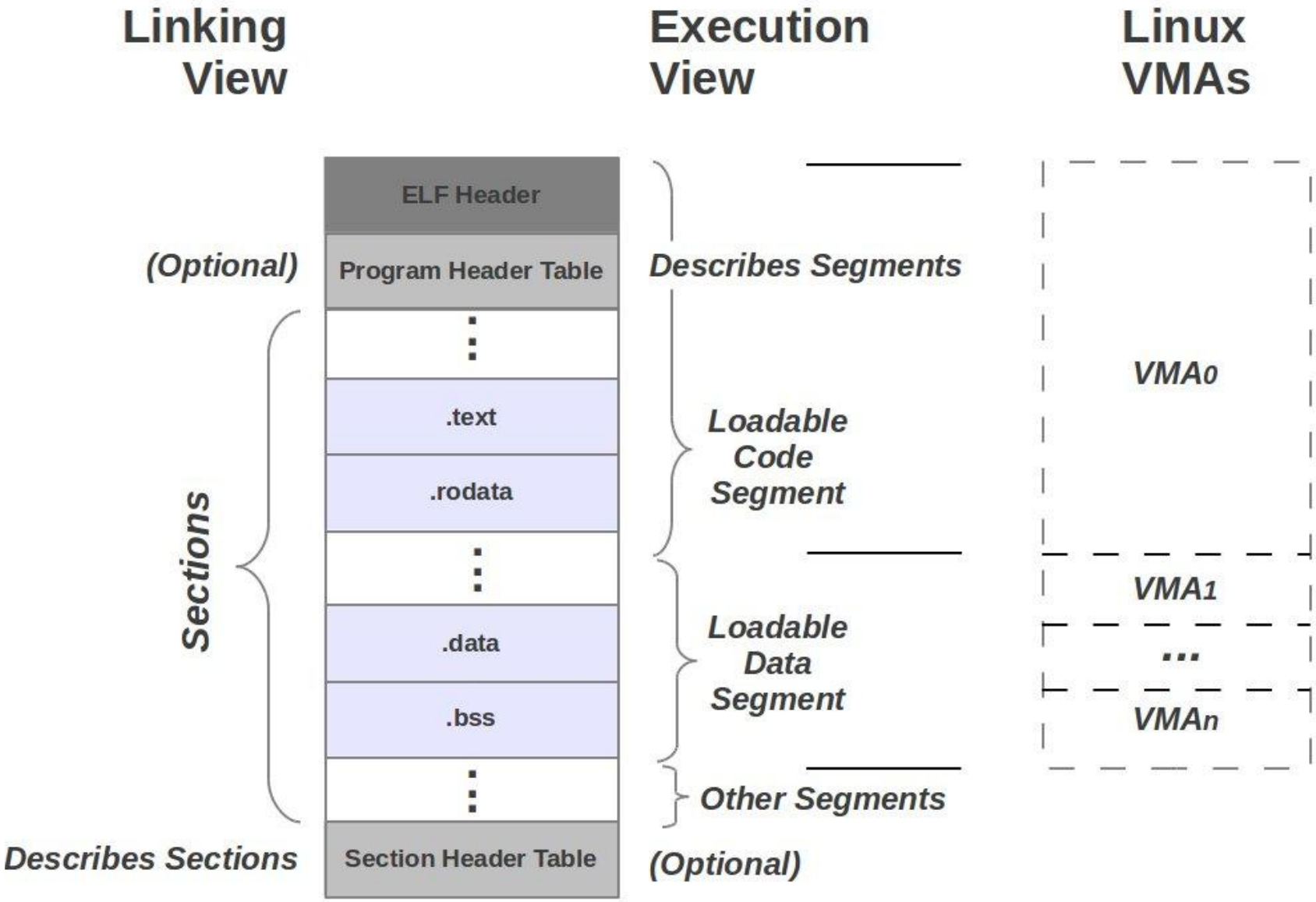
```
struct vm_area_struct {
    unsigned long vm_start; /* Our start address. */
    unsigned long vm_end;   /* Our end address+1 */
    struct vm_area_struct *vm_next, *vm_prev;
    struct rb_node vm_rb;
    struct mm_struct *vm_mm; /* Address space. */
    pgprot_t vm_page_prot;   /* Access permissions. */
    unsigned long vm_flags;  /* Flags, see mm.h. */
    struct list_head anon_vma_chain; /* Serialized by
                                      * page_table_lock */
    unsigned long vm_pgoff; /* Offset within file */
    struct file * vm_file; /* File we map to. */
};
```

Page Faults: PTE Info

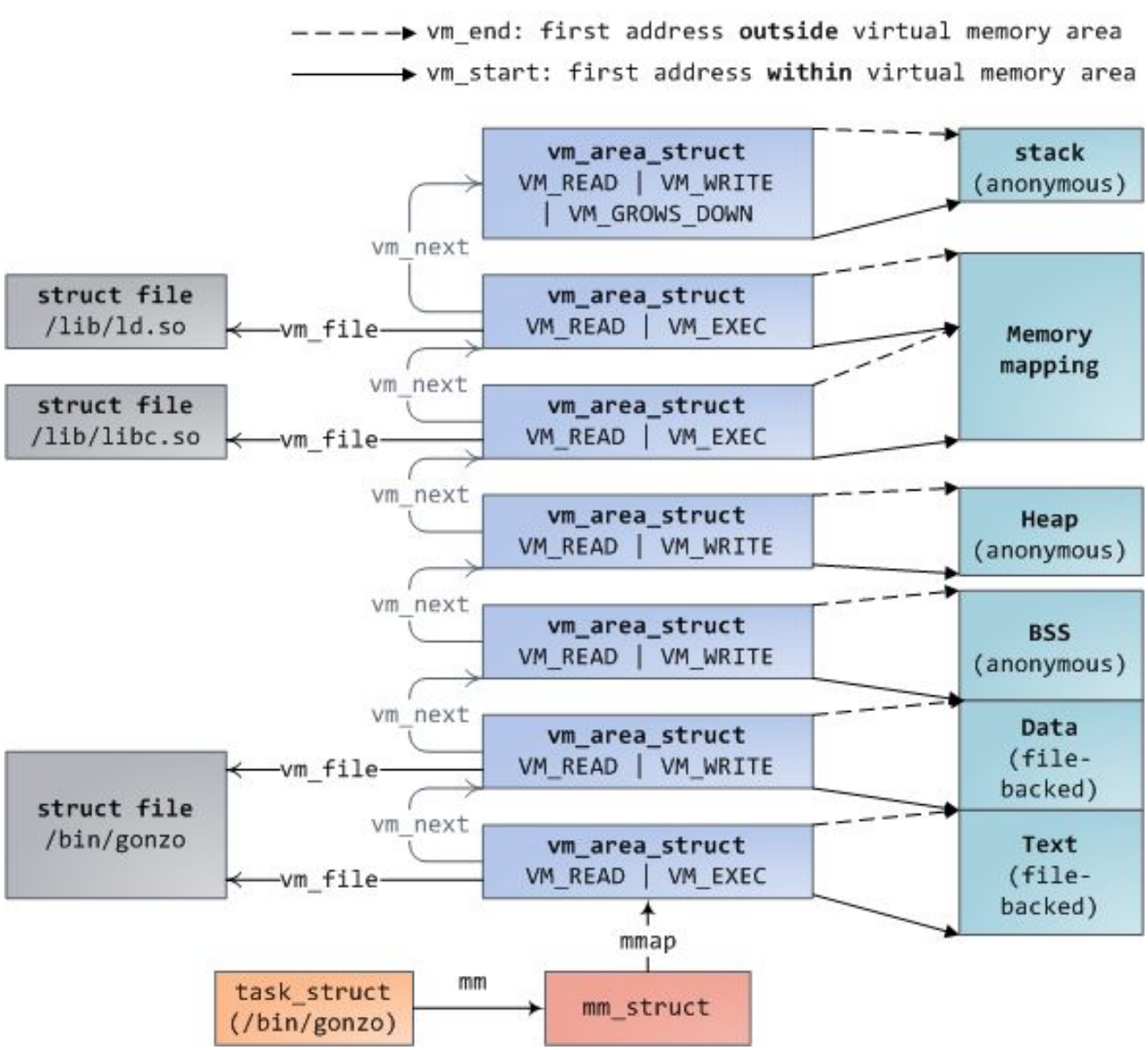
- How to retrieve the faulting PTE?
  - Walk page tables from faulting address

```
pgd_t * pgd = pgd_offset(mm, address);
p4d_t * p4d = p4d_alloc(mm, pgd, address);
pud_t * pud = pud_alloc(mm, p4d, address);
pmd_t * pmd = pmd_alloc(mm, pud, address);
pte_t * pte = pte_alloc(mm, pmd, address);
```



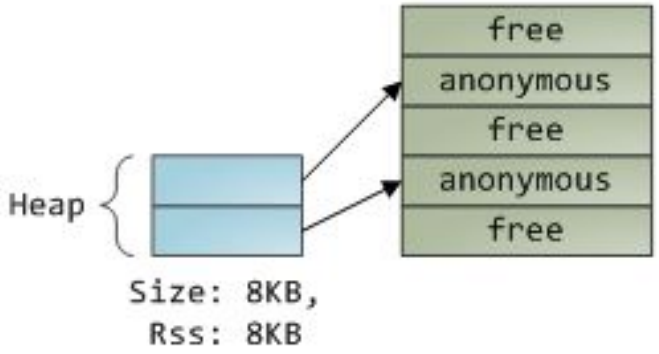


Linking View vs. Execution View vs. VMA View

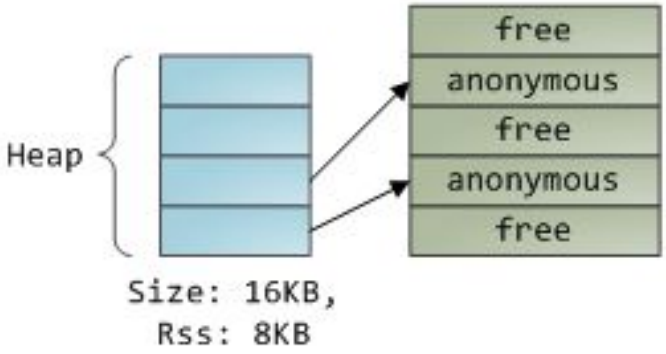


VMA: Anonymous vs. File-backed Memory

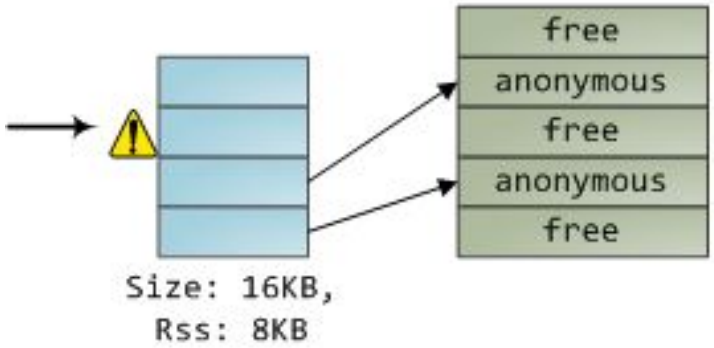
1. Program calls brk() to grow its heap



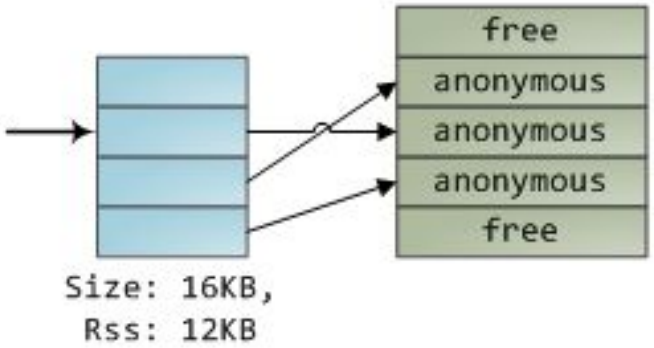
2. brk() enlarges heap VMA.
New pages are **not** mapped onto physical memory.



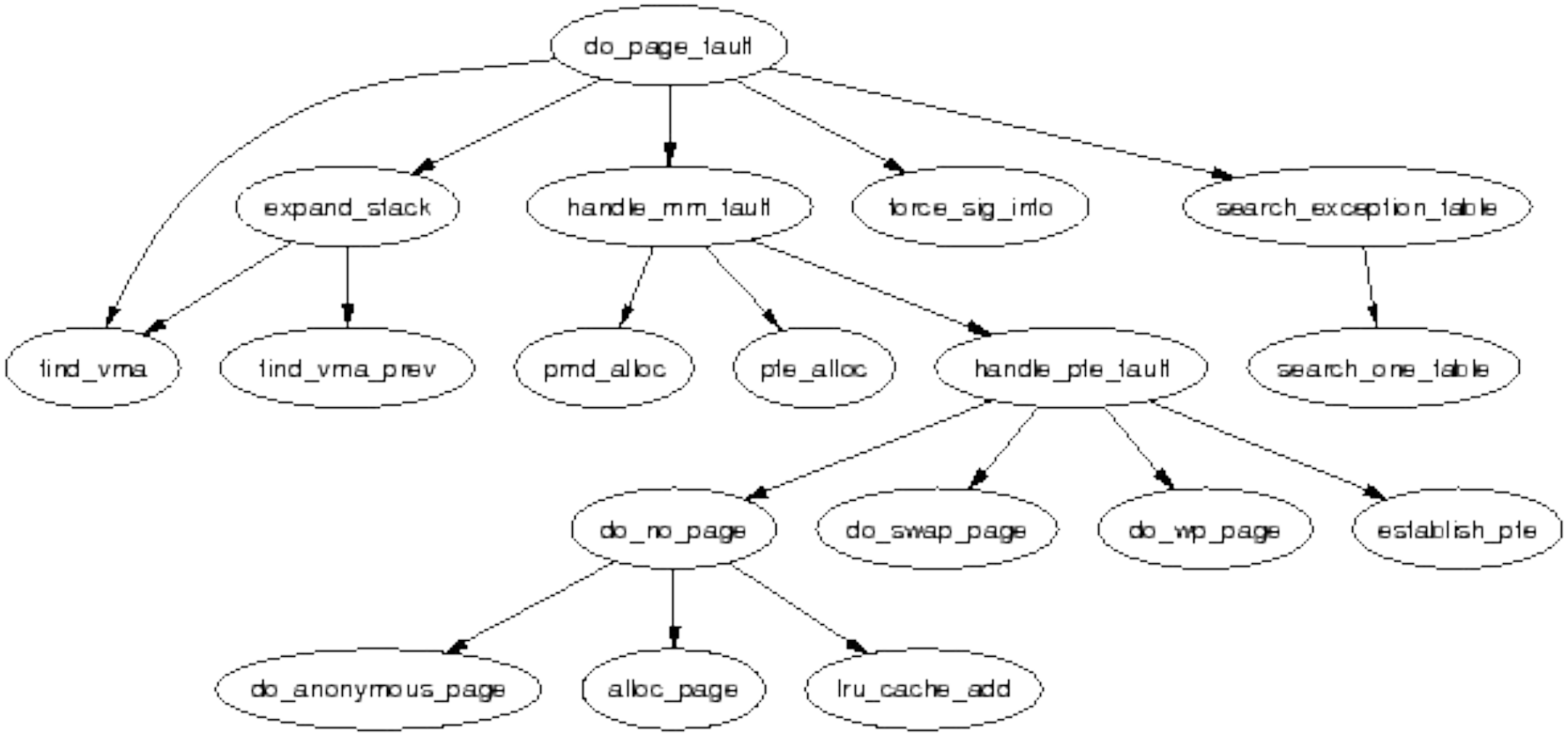
3. Program tries to access new memory.
Processor page faults.



4. Kernel assigns page frame to process, creates PTE, resumes execution. Program is unaware anything happened.



Demand Paging Example



Page Fault Handling Callgraph

VMA Operations: `mmap`

- `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`
- Maps a new region in the address space
  - PROT_NONE | READ | WRITE | EXECUTE
  - MAP_PRIVATE | SHARED | ANONYMOUS
  - MAP_FIXED | POPULATE | GROWSDOWN...
- Kernel:
  - Finds fitting hole in `mm` (`get_unmmaped_area`)
  - Simply creates and links new VMA if we are lucky
  - Faults in new page frames if requested

VMA Operations: `munmap`

- `int munmap(void *addr, size_t length);`
- Unmaps a region in the address space
- Kernel:
  - Simply unlinks and deletes a VMA if we are lucky
  - Removes mapped page frames (`zap_pte_range`)
  - Removes allocated page tables (`free_pgtables`)
  - Flushes TLB entries

Paging in OpenLSD

Lab 4: Don’t stop me now
(paging)

Core:

- Implement page fault handling. Use VMAs for permission checking and demand paging
- Implement `mmap`, `munmap`

Bonuses:

- Huge pages, `mprotect`, `madvise`, `vdso`

VMA Operations: `mprotect`

- `int mprotect(void *addr, size_t len, int prot);`
- Changes a region's protection bits
- Kernel:
  - Simply updates `vma->vm_page_prot` if we are lucky
  - Updates the affected PTEs (`change_protection`)
  - TLB entries?

VMA Operations: `madvise`

- `int madvise(void *addr, size_t length, int advice);`
- Gives the kernel advice on a region, e.g.:
  - MADV_DONTNEED (get rid of page frames)
  - MADV_WILLNEED (readahead)
- Kernel:
  - Simply updates VMA properties if we are lucky
  - Updates the affected PTEs as necessary

VMA Operations: Beyond `madvise`

- `madvise` can specify simple paging policies
- What if we want more complex ones? E.g.:
  - Lazy restore for checkpoint/restore mechanisms
  - Post-copy live migration of virtual machines
- Userfaultfd (UFFD):
  - User-level page fault handling
  - Allows arbitrary paging policies in userland:
    - Application thread touches non-present page
    - Kernel upcalls UFFD thread with page fault info
    - UFFD thread handles page fault and returns
    - Application thread can resume execution

VMA Operations: userfaultfd

```
/* Create and enable userfaultfd object */
if ((uffd = userfaultfd(O_CLOEXEC | O_NONBLOCK)) == -1)
    errExit("userfaultfd");
if (ioctl(uffd, UFFDIO_API, &uffdio_api) == -1)
    errExit("ioctl-UFFDIO_API");

/* Create a private anonymous mapping. */
if ((p = mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, -1, 0)) ==

    errExit("mmap");

/* Register the memory range of the mapping. */
if (ioctl(uffd, UFFDIO_REGISTER, &uffdio_register) == -1)
    errExit("ioctl-UFFDIO_REGISTER");

/* Create a thread that will process the userfaultfd events */
if ((s = pthread_create(&thr, NULL, fault_handler_thread, (void *) uffd)))
    errExit("pthread_create");
```

VMA Operations: Find and Split

- Problem: `mmap`, `munmap`, `mprotect`, `madvise` work on arbitrary address ranges
- Address range != VMA
- We need to map address ranges → VMAs
  - `find_vma` (similar to page fault handling)
- We need to split VMAs as needed
  - `split_vma`
  - e.g., `mprotect` on a VMA subrange
    - May end up with 2 splits
    - 3 VMAs as a result

VMA Operations: Merge

- Problem: split and create operations may results in more and more VMAs
  - `find_vma` gets slower and slower
- VMA cache mitigates the impact, but:
  - Cache misses are still relatively frequent
  - VMAs still consume kernel resources
  - Limiting is needed (/proc/sys/vm/max_map_count)
- Solution:
  - VMA merging (`vma_merge`)
  - Merge any adjacent VMAs with same properties
  - `mmap` is VMA-merge friendly (buffer-allocator-like)

VMA Operations: Huge Pages

- How do we use huge pages?
- mmap support for hugetlbfs:
  - HUGETLB|HUGE_2MB|HUGE_1GB
- Problems:
  - Requires application awareness
  - Static limit to the number of huge pages available
  - Can’t swap or mix with regular pages within VMA
- `Libhugetlbfs`
  - Overrides mmap & friends to allocate huge pages
  - More transparency, but perhaps too much
  - Can’t solve the other problems

Transparent Huge Pages

- Transparent Huge Pages (THPs) designed to overcome the limitations of hugetlbfs
  - Key application: virtualization (KVM)
- Idea: Kernel transparently allocates THPs whenever possible (e.g., alignment is right)
- Applications need not be THP-aware at all
- In practice application awareness helps:
  - Align mapped regions to THP boundary
  - Don’t THP if you expect to use few bytes in a region
    - `madvise`’s `MADV_NOHUGEPAGE` flag
    - `MADV_HUGEPAGE` when default off

THP Operations

- Map:
  - Allocate compound pages and map them as THPs
  - Compound page: group of phys-contiguous pages
- Collapse:
  - Replace a number of regular pages with THP
- Split:
  - Split a THP into a number of regular pages
- Compact:
  - Memory compaction to allocate compound pages
  - Needed when memory gets severely fragmented
  - How can memory get fragmented with buddy?

THP Operations: Map

- When to map?
  - Page fault (PF) time, periodically, or never
    - Currently at PF time (`do_huge_pmd_*_page`)
    - May increase PF latency
- Under which conditions?
  - Mapped huge page within owning VMA (alignment)
  - Huge pmd available (availability)
  - Owning VMA larger than threshold (thresholding)
    - Currently only alignment and availability
- What memory types?
  - Currently anonymous and tmpfs/shmem memory
    - KVM happens to use anon memory for guests

THP Operations: Collapse

- When to collapse?
  - Page fault (PF) time, periodically, or never
    - Currently periodically (khugepaged)
    - Scans N pages every K milliseconds
- Under which conditions?
  - Mapped huge page within owning VMA (alignment)
  - More than N page frames allocated (thresholding)
    - Currently only alignment
- Same memory types as Map
- How?
  - Replace pages with a compound page
  - Replace PTEs with huge pmd

THP Operations: Split

- When to split?
  - `split_vma`, swap (4K-only), DONTNEED(4K), etc.
    - Originally, all except DONTNEED (`split_huge_pmd`)
    - Also as needed for THP-unaware kernel code
    - Collapse will eventually fix this
- The jemalloc memory leak
  - DONTNEED key to reclaim unused memory
  - With THP this doesn't work
  - Prompted changes to jemalloc to `MADV_NOHUGEPAGE` certain memory regions
  - Fixed in recent kernels

THP Operations: Compact

- When to compact?
  - Direct compaction (failed alloc), periodic compaction
    - Currently both are implemented
- `kcompactd`
  - Kernel thread for periodic memory compaction
  - Direct memory compaction waits for `kcompactd`
  - Used by PF handling code and `khugepaged`

THP Tunables

 # THP

```
$ echo [always|madvise|never] >
/sys/kernel/mm/transparent_hugepage/enabled
$ echo [always|defer|madvise|defer+madvise|never] >
/sys/kernel/mm/transparent_hugepage/defrag
```

 # khugepaged

```
$ echo [0|1] >
/sys/kernel/mm/transparent_hugepage/khugepaged/defrag
$ echo $value >
.../pages_to_scan|scan_sleep_millisecs|alloc_sleep_millisecs
```

THP Stats

```
$ grep Anon /proc/meminfo
AnonPages:         703044 kB
AnonHugePages:     389120 kB

$ grep thp /proc/vmstat
thp_fault_alloc 1408
thp_fault_fallback 0
thp_collapse_alloc 194
thp_collapse_alloc_failed 0
thp_split_page 119
thp_zero_page_alloc 0
thp_zero_page_alloc_failed 0
```

References

[1] Bovet, Daniel P., and Marco Cesati. Understanding the Linux Kernel, 2005.

[2] Gorman, Mel. Understanding the Linux Virtual Memory Manager, 2004.

[3] “Linux Cross Reference,” n.d. https://elixir.bootlin.com/linux/latest/source.

[4] “Transparent Hugepage Support,” n.d. https://www.kernel.org/doc/Documentation/vm/transhuge.txt.