SWE3015: Operating System Project

VIRTUAL MEMORY IMPLEMENTATION
Process Address Space

- Linux 32 bit address space layout

```
<table>
<thead>
<tr>
<th>PAGE_OFFSET</th>
<th>0xc0000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel virtual memory (code, data, heap, stack)</td>
<td></td>
</tr>
<tr>
<td>user stack (created at runtime)</td>
<td></td>
</tr>
<tr>
<td>memory mapped region for shared libraries</td>
<td></td>
</tr>
<tr>
<td>run-time heap (managed by malloc)</td>
<td></td>
</tr>
<tr>
<td>read/write segment (.data, .bss)</td>
<td></td>
</tr>
<tr>
<td>read-only segment (.init, .text, .rodata)</td>
<td></td>
</tr>
<tr>
<td>unused</td>
<td></td>
</tr>
</tbody>
</table>
```
• Linux 64 bit address space layout
Process Address Space

• Memory areas
  - The intervals of legal addresses in process address space.
    - Text section: code
    - Data section: initialized global variables
    - BSS section: uninitialized global variables
    - User-space stack
    - Heap
    - Shared libraries (text, data, bss for each shlib)
    - Memory mapped files
    - Shared memory segments
Process Address Space

- Paging: three-level address translation
  - In i386, the size of Page Middle Directory (PMD) is 1, if the physical address extension (PAE) flag is disabled.
  - 512 In x86_64, 4-level page table is appeared (add PUD)
Process Address Space

```
struct mm_struct

Virtual Address (addr)

00111010 110110011 001101110 110101111

pgd part  pmd part  pte part  offset

PGD

PMD

PTE

Software relationships

Hardware relationships

pgd_offset(mm_struct, addr);
pmd_offset(pgd_t, addr);
pte_offset(pmd_t, addr);
pte_page(pte_t);
page.virtual

pgd_val(pgd);
pmd_val(pmd);
pte_val(pte);
```
Page Table Functions

• PTE functions
  – MK function: `pte_mkproperty(pte_t pte)`
    • Set indicated property (e.g. write protect)
  – Flag checker: `pte_flag (pte_t pte)`
    • Return True(1) if flag is set, otherwise return False(0)
  – Get PFN: `pte_pfn(pte_t pte)`

• There are similar functions for PGD, PMD, and PUD
  – See `<arch/x86/include/asm/pgtable.h>`
Page Table Functions Examples

- Code snippet in <mm/ksm.c>
  - KSM provides memory deduplication based on page tables

```c
if (pte_dirty(pte))
    set_page_dirty(page);
entry = pte_mkc lean(pte_wrprotect(entry));
set_p te_at_notify(mm, addr, ptep, entry);

<mm/ksm.c/write_pro tect_page()>
```

```c
static inline pte_t pte_wrprotect(pte_t pte)
{
    return pte_clear_flags(pte, _PAGE_RW);
}

<arch/x86/include/asm/pgtable.h>
```
Virtual Memory

• Swapping
  – Physical memory ↔ Swap area
  – Store anonymous pages to swap storage to reclaim physical memory pages (swap out)
    • Swapped in when an attempt is made to access it

– Anonymous pages
  • Memory pages which is not related to file
  • Stack, heap, BSS
  • (written) program data
  • Shared memory, mmap() with MAP_ANON
Virtual Memory

• Demand Paging
  – Physical memory ↔ File system
  – Copying a disk page into physical memory only if an attempt is made to access it and that page is not already in memory
    • Program code
    • (initial) program data
    • Memory-mapped files
    • Swapped-out pages
  – Virtually mapped but not allocated physical memory until access
Virtual Memory

• Page cache
  – A cache of physical storage pages
  – The page cache holds
    • Pages containing data of regular files
    • Pages containing directories
    • Pages containing data directly read from block device files
    • Pages containing data of user mode processes that have been swapped out on disk
    • Pages belonging to files of special filesystems (e.g., shm)
  – Each page included in the page cache contains data belonging to some file.
Virtual Memory

• Page fault
  – Page fault mainly occurs due to
    • Not-present pages (especially for demand paging)
    • Protection violation (especially for copy-on-write)

  – Major page fault
    • If the kernel need to access the disk to make the page available.

  – Minor page fault
    • If the kernel only need to allocate pages in RAM without reading anything from disk.
SWE3015: Operating System Project

MEMORY MANAGEMENT IN LINUX KERNEL
• struct vm_area_struct `<linux/mm.h>

  – Non-overlapping regions, each representing a continuous, page-aligned subset of the virtual address space.

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct mm_struct *</td>
<td>associated mm_struct</td>
</tr>
<tr>
<td>unsigned long</td>
<td>VMA start, inclusive</td>
</tr>
<tr>
<td>unsigned long</td>
<td>VMA end, exclusive</td>
</tr>
<tr>
<td>struct vm_area_struct *</td>
<td>list of VMA’s</td>
</tr>
<tr>
<td>pgprot_t</td>
<td>access permissions</td>
</tr>
<tr>
<td>unsigned long</td>
<td>VMA flags</td>
</tr>
<tr>
<td>struct rb_node</td>
<td>VMA’s node in the tree</td>
</tr>
<tr>
<td>struct vm_operations_struct *</td>
<td>associated ops</td>
</tr>
<tr>
<td>struct file *</td>
<td>mapped file, if any.</td>
</tr>
<tr>
<td>unsigned long</td>
<td>offset within the file</td>
</tr>
</tbody>
</table>

...
• Linked list (via mm -> mmap)
  – Simple and efficient for traversing of all elements
  – Sorted by ascended address (linked via vma -> vm_next)
• Red-black tree (via mm -> mm_rb)
  – A type of balanced binary tree.
    • The root & all leaves are black.
    • Both children of every red node are black.
    • All paths from any given node to its leaf nodes contain the same number of black nodes.
  – Searching, insertion, deletion: $O(\log(n))$
  – Used when locating a specific VMA in the address space.
- **VMA flags** `<linux/mm.h>`

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM_READ / VM_WRITE / VM_EXEC</td>
<td>Pages can be read from / written to / executed.</td>
</tr>
<tr>
<td>VM_SHARED</td>
<td>Pages are shared.</td>
</tr>
<tr>
<td>VM_MAYREAD / VM_MAYWRITE / VM_MAYEXEC / VM_MAYSHARE</td>
<td>VM_READ / VM_WRITE / VM_EXEC / VM_SHARE flag can be set.</td>
</tr>
<tr>
<td>VM_GROWSDOWN / VM_GROWSUP</td>
<td>The area can grow downward / upward.</td>
</tr>
<tr>
<td>VM_SHM</td>
<td>The area is used for shared memory</td>
</tr>
<tr>
<td>VM_DENYWRITE / VM_EXECUTABLE</td>
<td>The area maps an unwritable file / an executable file</td>
</tr>
<tr>
<td>VM_LOCKED</td>
<td>The pages in this area are locked.</td>
</tr>
<tr>
<td>VM_IO</td>
<td>The area maps a device’s I/O space.</td>
</tr>
<tr>
<td>VM_RESERVED</td>
<td>This area must not be swapped out.</td>
</tr>
<tr>
<td>VM_SEQ_READ / VM_RAND_READ</td>
<td>The pages seem to be accessed sequentially / randomly.</td>
</tr>
</tbody>
</table>
• VMA operations
  – struct vm_operations_struct <linux/mm.h>
    • fault function called by page fault handler
      – See handle_pte_fault in mm/memory.c

<table>
<thead>
<tr>
<th>Name</th>
<th>Caller Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>void open</td>
<td>When given memory area is added</td>
</tr>
<tr>
<td>void close</td>
<td>When given memory area is removed</td>
</tr>
<tr>
<td>int fault</td>
<td>Invoked by page fault handler (see above description)</td>
</tr>
<tr>
<td>int access</td>
<td>When get_user_pages() fails</td>
</tr>
<tr>
<td>int page_mkwrite</td>
<td>Invoked by page fault handler (e.g. process accesses read-only page)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
• `/proc/<pid>/maps <fs/proc/task_mmu.c>`
  – start – end permission offset major:minor inode file

```bash
hahaman5@ubuntu:~$ cat /proc/3650/maps
00400000-0040b000 r-xp 00000000 fc:00 130830
0060a000-0060d000 r--p 0000a000 fc:00 130830
0060b000-0060c000 rw-p 0000b000 fc:00 130830
019af000-019d0000 rw-p 00000000 00:00 0
7f76e538b000-7f76e5654000 r--p 00000000 fc:00 7793
7f76e5654000-7f76e5813000 r-xp 00000000 fc:00 147750
7f76e5813000-7f76e5a12000 --p 001bf000 fc:00 147750
7f76e5a12000-7f76e5a16000 r--p 001be000 fc:00 147750
7f76e5a16000-7f76e5a18000 rw-p 001c2000 fc:00 147750
7f76e5a18000-7f76e5a1d000 rw-p 00000000 00:00 0
7f76e5a1d000-7f76e5a40000 r-xp 00000000 fc:00 147729
7f76e5c35000-7f76e5c38000 rw-p 00000000 00:00 0
7f76e5c3d000-7f76e5c3f000 rw-p 00000000 00:00 0
7f76e5c3f000-7f76e5c40000 r--p 00220000 fc:00 147729
7f76e5c40000-7f76e5c42000 rw-p 00230000 fc:00 147729
7ffffff2f000-7fffffff6250000 rw-p 00000000 00:00 0
7ffffff2fe000-7fffffff6400000 r-xp 00000000 00:00 0
fffffffffffffff601000 r-xp 00000000 00:00 0
```

```bash
hahaman5@ubuntu:~$ cat &
[1] 3650
```

```bash
hahaman5@ubuntu:~$ cat /proc/3650/maps
```

```bash
[1]+  Stopped  cat
hahaman5@ubuntu:~$ cat
```
Memory Descriptor

- struct mm_struct <linux/sched.h>
  - Contains all the information related to the process address space.
- Threads share a memory descriptor.
- Doubly linked via the mmlist field.
- No mm_struct for kernel thread

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct vm_area_struct * mmap;</td>
<td>list of memory areas (VMAs)</td>
</tr>
<tr>
<td>struct rb_root mm_rb;</td>
<td>red-black tree of memory areas</td>
</tr>
<tr>
<td>pgd_t * pgd;</td>
<td>page global directory</td>
</tr>
<tr>
<td>atomic_t mm_users;</td>
<td>address space users</td>
</tr>
<tr>
<td>atomic_t mm_count;</td>
<td>reference count for mm_struct</td>
</tr>
<tr>
<td>int map_count;</td>
<td>number of memory areas</td>
</tr>
<tr>
<td>struct list_head mmlist;</td>
<td>list of all mm_structs</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Memory Descriptor

- Allocating/freeing a memory descriptor
  - `allocate_mm()` `<kernel/fork.c>`
    - `kmem_cache_alloc(mm_cachep, GFP_KERNEL)`
  - `free_mm(mm)`
    - `kmem_cache_free(mm_cachep, (mm))`

```c
COPY_MM() {

  ... if (clone_flags & CLONE_VM) {
    atomic_inc(&current->mm->mm_users);
    tsk->mm = current->mm;
  } else { // process creation
    tsk->mm = allocate_mm();
    mm_init(tsk->mm);
    ... 
  }
}
```
Memory Descriptor

- **Kernel threads**
  - Kernel threads do not have a process address space and therefore do not have a memory descriptor.
  - Kernel threads use the memory descriptor of whatever task ran previously.
    - Task -> active_mm: address space referenced by a process
    - When a kernel thread is scheduled (task->mm == NULL), the kernel keeps the previous process’s address space loaded.
    - The kernel updates the active_mm field.
    - The kernel thread can then use the previous process’s page tables as needed.
  - Kernel threads use only the information pertaining to kernel memory. (same for all processes)
Address Space Structure

- task_struct (.a.out)
  - mm
  - mm_struct
- Struct file /lib/ld.so
- Struct file /lib/ld.so
- Struct file ./a.out

- vm_area_struct
  - VM_READ | VM_WRITE
  - VM_GROWS_DOWN
- vm_area_struct
  - VM_READ | VM_EXEC
- vm_area_struct
  - VM_READ | VM_WRITE
- vm_next
- vm_area_struct
  - VM_READ | VM_WRITE
- vm_area_struct
  - VM_READ | VM_WRITE
- vm_area_struct
  - VM_READ | VM_WRITE
- vm_area_struct
  - VM_READ | VM_WRITE

Stack
- (anonymous)
Memory Mapping
- (file-backed)
Heap
- (anonymous)
- .bss
  - (anonymous)
- .data
  - (file-backed)
- .text
  - (file-backed)
Page Fault Handling

- `<arch/x86/mm/fault.c>` `do_page_fault()`