

OS-Lab5

Kasra Noorbakhsh - Kourosch Sajjadi - Mehdi Jamalkhah

810100230 - 810100587 - 810100111

Contents

1	Introduction	2
1.1	VMA(Q1)	2
1.2	Hierarchical(Q2)	2
1.3	Entry(Q3)	2
2	XV6 Memory Management	3
2.1	kalloc()(Q4)	3
2.2	mappages()(Q5)	3
2.3	walkpgdir()(Q6)	3
2.4	allocvm() and mappages()(Q7)	3
2.5	exec()(Q8)	3
3	Implementation	4

1 Introduction

1.1 VMA(Q1)

In the Linux operating system, a virtual memory area (VMA) is a region of a process's virtual address space that is allocated and managed by the kernel. It represents a contiguous block of virtual memory that is mapped to either physical memory or swap space. VMAs are essential for efficient memory management in Linux, as they allow the kernel to keep track of which parts of a process's virtual memory are currently in use and which parts are not. This allows the kernel to page out unused memory to swap space when physical memory is low, and then page it back in when it is needed.

The xv6 operating system does not use the concept of VMAs. Instead, it uses a simpler memory management scheme that treats the entire virtual address space as a single contiguous block of memory. This can lead to inefficiencies in memory management, as the kernel must constantly page in and out memory from swap space.

1.2 Hierarchical(Q2)

Hierarchical paging save space in the physical memory by splitting the virtual-to-physical-address translation information into multiple smaller page tables. In this approach, page tables are organized into hierarchical layers where each layer points to the tables on the first layer below. We save space because page tables that aren't currently needed are swapped out to the disk.

1.3 Entry(Q3)

- **Address Information:**

- **Page Table Base Register (PTB) Index (20 bits):** This field identifies the index within the page directory table (PDT) that holds the address of the corresponding page table. The PDT maps virtual pages to physical frames, and the PTB index points to the appropriate page table entry (PTE) within the PDT.
- **Page Offset (12 bits):** This field specifies the offset within the virtual page that is being accessed. The offset is combined with the page table address obtained from the PTB index to form a complete physical memory address.

- **Protection and Control Information:**

- **Accessed (A) Bit (1 bit):** Indicates whether the page has been accessed recently. This information is used by the operating system to implement page replacement algorithms.
- **Dirty (D) Bit (1 bit):** Indicates whether the page has been modified since it was last loaded from disk. This information is used by the operating system to determine which pages need to be written back to disk when they are made unavailable.
- **Referenced (R) Bit (1 bit):** Indicates whether the page has been referenced recently. This information is also used by the operating system for page replacement decisions.
- **User/Supervisor (U/S) Bit (1 bit):** Controls whether the page can be accessed by user-mode or kernel-mode processes. This bit enforces memory protection between different security levels.
- **Write (W) Bit (1 bit):** Controls whether the page can be written to. This bit prevents unauthorized modifications to memory.
- **Present (P) Bit (1 bit):** Indicates whether the page is currently loaded in physical memory. If the Present bit is 0, the page is not present and the virtual address translation will fail.
- **Reserved (R) Bits (8 bits):** Leftover bits that are not currently used but may be repurposed in future memory management schemes.

2 XV6 Memory Management

2.1 kalloc()(Q4)

The xv6 kernel calls `kalloc` and `kfree` to allocate and free **physical memory** at run-time. The kernel uses run-time allocation for process memory and for these kernel data structures: kernel stacks, pipe buffers, and page tables.

2.2 mappages()(Q5)

The `mappages()` function in the xv6 operating system is responsible for mapping virtual pages to physical frames. parameters of `mappages` are `pde-t *pgdir, void *va, uint pa, int perm`. this function Creates translations from VA (virtual address) to pa (physical address) in existing page table `pgdir`.

2.3 walkpgdir()(Q6)

`walkpgdir` takes a page directory (first-level page table) and returns a pointer to the page table entry for a particular virtual address. It optionally will allocate any needed second-level page tables. parameters of `walkpgdir` are `walkpgdirpde-t *pgdir, const void *va, int alloc` Looks at virtual address `va`, finds where it should be mapped to according to page table `pgdir`, and returns the virtual address of the the index `il`. Returns address if successful, 0. If there is no mapping, then: if `alloc=1`, mapping is created (and address is returned) if `alloc=0`, return 0 this function simulates the translation and mapping between these 2 levels of page tables that hardware used TLB for it.

2.4 allocvm() and mappages()(Q7)

- **allocvm:** `allocvm` stands for Allocate User Virtual Memory. It's responsible for expanding the virtual address space of a process by allocating new page tables and physical memory pages. When a process needs more memory to accommodate its growing demands, it invokes `allocvm` to request the additional space.
- **mappages:** It's responsible for creating mappings between virtual addresses in a process's address space and the physical memory pages they represent. These mappings are stored in page tables, which are the data structures that translate virtual addresses to physical addresses.

When `allocvm` allocates additional page tables, it populates them with entries that map the newly allocated physical memory pages to the appropriate virtual addresses within the process's address space. This process of setting up mappings ensures that the process can access and use the newly allocated memory.

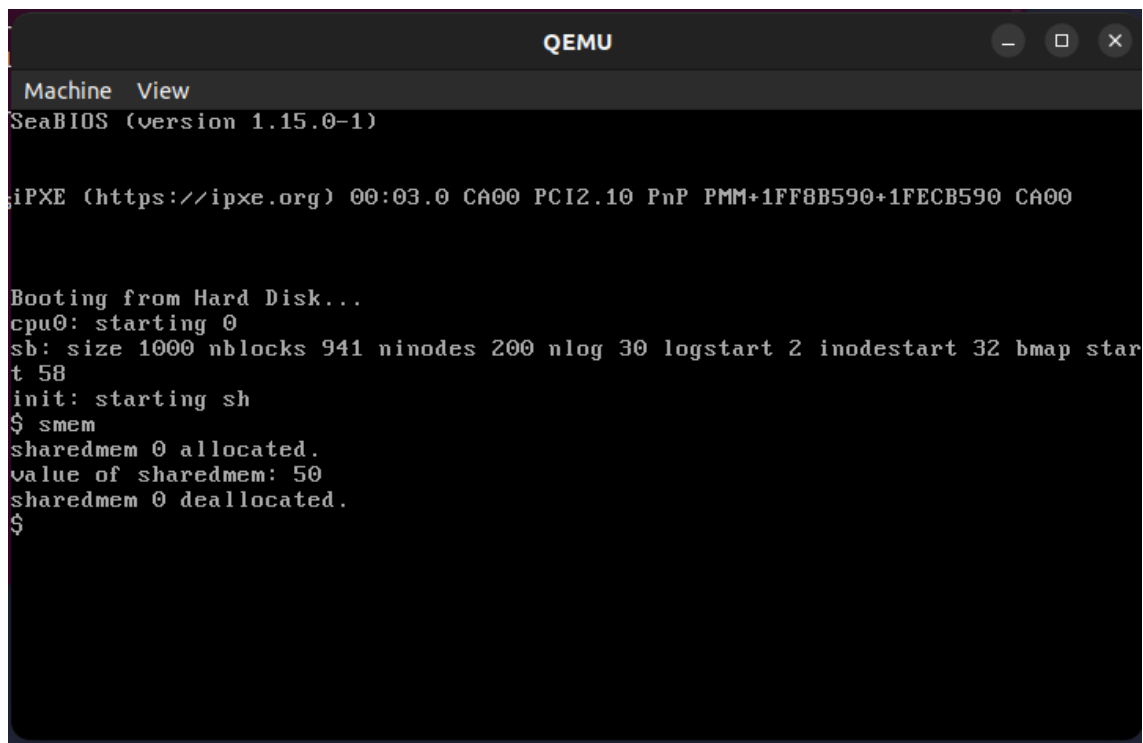
2.5 exec()(Q8)

- **Parsing the ELF file:** The `exec` system call first parses the ELF (Executable and Linkable Format) file to extract information about the program's layout.
- **Allocating memory:** allocates memory for the new program image by calling the `uvmmalloc` function. This function allocates a contiguous block of memory large enough to hold the entire program image
- **Loading program segments:** loads the program segments from the ELF file into the allocated memory. It uses the `loadseg` function to copy each segment from the file to the corresponding memory location specified in the ELF file.
- **Preparing the user stack:** The `exec` system call allocates a new stack page and pushes the program's arguments onto the stack.
- **Replacing the old image:** Finally, replaces the old process image with the new program image.

3 Implementation

For the `closed_shared_mem` and `open_shared_mem` we used the below strategy:

We used system calls (in xv6 only privileged mode can access) for a process to shrink or grow its memory. The system call is implemented by the function `growproc_smem`. If `n` is positive, `growproc_smem` allocates one or more physical pages and maps them at the top of the process's address space. If `n` is negative, `growproc_smem` un-maps one or more pages from the process's address space and frees the corresponding physical pages. To make these changes, xv6 modifies the process's page table. The process's page table is stored in memory, and so the kernel can update the table with ordinary assignment statements, which is what `allocuvmm_smem` and `deallocuvmm_smem` do.



```
QEMU
Machine View
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00

Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ smem
sharedmem 0 allocated.
value of sharedmem: 50
sharedmem 0 deallocated.
$
```

Figure 1: User-program(smem)

For the system call we forked 50 child's and opened the shared memory and increment it, by waiting we assure that the affects are visible.