Paging in RISC-V (SV39)

In this chapter, we introduce **virtual memory** to Novix OS by implementing the **SV39 paging system** — the 39-bit virtual memory scheme used by RISC-V 64-bit systems.
Paging provides **isolation**, **protection**, and **address space translation** for every process.

Concept Overview

In **SV39**, each virtual address (VA) is divided into:

- **9 bits VPN[2]**
- **9 bits VPN[1]**
- **9 bits VPN[0]**
- **12 bits page offset**

This structure creates a **3-level page table**:

- **Level 2 (Root)** → Page directory pointer
- **Level 1 (Middle)** → Page directory
- **Level 0 (Leaf)** → Page table entries (PTEs)

Each PTE describes the mapping between a **virtual page** and a **physical frame** and includes permission bits:

Flag	Meaning
V	Valid entry
R	Read
W	Write
X	Execute
U	User accessible
G	Global
A	Accessed
D	Dirty

Function: paging_init

```c
void paging_init(void);
```

Purpose

Initializes the kernel's root page table and performs **identity mapping** of the kernel memory.

Detailed Steps

1. Allocate one page for the root page table:

   ```c
   paddr_t pt_page = alloc_pages(1);
   kernel_pagetable = (pagetable_t)pt_page;
   ```

2. Identity-map all kernel memory from `__kernel_base` to `__heap_end`:

   ```c
   map_page(kernel_pagetable, pa, pa, PTE_V | PTE_R | PTE_W | PTE_X, 0);
   ```

3. Map MMIO regions for devices such as **VirtIO**:

   ```c
   map_mmio_range(kernel_pagetable, VIRTIO_MMIO_BASE, VIRTIO_MMIO_BASE,
                  VIRTIO_MMIO_SIZE * VIRTIO_MMIO_MAX_DEVICES,
                  PTE_V | PTE_R | PTE_W);
   ```

4. Load this new page table into the CPU:

   ```
   set_active_pagetable((uintptr_t)kernel_pagetable);
   ```

5. Log success:

   ```
   uart_printf("[paging_init] kernel page table initialized at : 0x%lx\n",
               (uintptr_t)kernel_pagetable);
   ```

Function: alloc_pages

```
paddr_t alloc_pages(uint64_t n);
```

Purpose

Allocates one or more contiguous physical pages.
This is the physical memory backend used by the paging subsystem.

Algorithm

- For a single page:
  - Try to reuse one from the **free list** (`alloc_free_list()`)
  - Otherwise use the **bump allocator** (`next_paddr`)
- For multiple pages:
  - Allocate sequentially via bump pointer
  - Zero-initialize using `memset()`
  - Track each allocation with `track_alloc()`

Function: free_page / free_pages_range

```
void free_page(paddr_t pa);
void free_pages_range(paddr_t pa, size_t npages);
```

Purpose

Releases one or more physical pages and returns them to the free list.

Description

- Adds the page's address to the free list.
- Increments `g_free_pages` counter.
- Tracks the deallocation for debugging purposes (`track_free()`).

Function: map_page

```
void map_page(pagetable_t root_table, uint64_t va, uint64_t pa,
              uint64_t flags, int debug_flag);
```

Purpose

Creates a mapping from **virtual address (VA)** to **physical address (PA)**.

Detailed Description

1. Extract indices:

   ```
   vpn2 = (va >> 30) & 0x1FF;
   vpn1 = (va >> 21) & 0x1FF;
   vpn0 = (va >> 12) & 0x1FF;
   ```

2. Traverse or allocate new page tables at each level (L2 → L1 → L0).

3. Insert the final **leaf PTE** at level 0:

   ```
   pt[vpn0] = ((pa >> 12) << 10) | flags | PTE_V;
   ```

4. Optional UART log when `debug_flag` is true.

Function: unmap_page

```
void unmap_page(pagetable_t root_table, vaddr_t va);
```

Purpose

Removes a mapping from a virtual address.

Behavior

- Traverses the same 3-level page table.
- Clears the leaf PTE.
- Executes `sfence_vma()` to flush the TLB.

Function: set_active_pagetable

```
void set_active_pagetable(uintptr_t new_pagetable);
```

Purpose

Switches the CPU's currently active page table.

Mechanism

- Writes the `satp` CSR with the new page table physical address:

  ```
  WRITE_CSR(satp, MAKE_SATP(new_pagetable));
  sfence_vma();
  ```

Function: map_mmio_range

```
void map_mmio_range(pagetable_t pt, uint64_t pa_start,
                    uint64_t va_start, uint64_t len, uint64_t perm);
```

Purpose

Maps **memory-mapped I/O (MMIO)** regions such as device buffers.

Implementation

Loops over each page of the MMIO region and calls `map_page()` with the specified permissions.

Function: walk_page

```
paddr_t walk_page(pagetable_t pagetable, vaddr_t va);
```

Purpose

Traverses a page table and returns the physical address mapped to a virtual address.

Return Value

- Physical address (if mapping exists)
- `0` if unmapped or invalid

Function: walk

```
pte_t* walk(pagetable_t pagetable, uint64_t va, int alloc);
```

Purpose

Locates or creates the PTE for a given virtual address.
If `alloc` is true, missing page tables are allocated dynamically.

Function: walkaddr

```
paddr_t walkaddr(pagetable_t pagetable, vaddr_t va);
```

Purpose

Translates a user-space virtual address to a physical address.

Safety

- Rejects kernel-space addresses.
- Verifies the PTE's validity and permission bits.
- Returns 0 on failure.

Function: free_pagetable

```
void free_pagetable(pagetable_t pt, int lvl);
```

Purpose

Recursively frees all page-table pages belonging to a process.

Behavior

- Traverses all 512 entries per level.
- Frees only **user pages** (PTE_U set).
- Skips kernel mappings.
- Clears PTE entries after freeing.

Function: debug_pagetable

```
void debug_pagetable(const char *who);
```

Purpose

Prints the currently active page table's physical address by reading the `satp` CSR.

Function: page_allocator_free_pages

```
uint64_t page_allocator_free_pages(void);
```

Purpose

Returns the count of currently free physical pages (`g_free_pages`).

Function: kernel_main

```
void kernel_main(void);
```

Purpose

Initializes all major kernel subsystems, including paging.

Steps

1. **Clear BSS**
   ```
   bss_init();
   ```
2. **Boot message**
   ```
   uart_cls();
   uart_printf("Novix RISC-V 64 OS, (c) NovixManiac, Version 0.0.1\n\n");
   ```
3. **Initialize subsystems**
   ```
   trap_init();
   stack_init();
   ram_init();
   heap_init();
   ```
4. **Enable Paging**
   ```
   paging_init();
   ```
5. Enter infinite loop.

Summary

At this stage, **Novix OS** now has a functioning **virtual memory system** powered by the **SV39** paging scheme. You have implemented:

- Multi-level page tables (L2 → L1 → L0)
- Physical page allocation and freeing
- Page mapping, unmapping, and lookup
- Kernel identity mapping
- MMIO region handling
- Page-table switching via `satp`

This is one of the most important architectural milestones — from now on, every address in Novix OS goes through the paging layer, providing the basis for **process isolation**, **system calls**, and **virtual address spaces**.