Linux Systems Internals and Programming (contd…)

---

2. Segmentation and Paging

2.1 Role of Segmentation

Definition:
Segmentation is a memory management technique that divides a program's memory into variable-sized segments. Each segment corresponds to a logical part of the program, such as code, data, stack, or heap.

Key Features:

- Provides **logical division** of memory.
- Each segment has a **base address** and **limit** (size).
- Segments allow **protection** by setting different access permissions.

Real-world Example:
Consider a C program with the following memory segments:

- **Code Segment (Text Segment)** – Stores the compiled machine code.
- **Data Segment** – Contains global and static variables.
- **Stack Segment** – Stores local variables and function call information.
- **Heap Segment** – Used for dynamic memory allocation.

💡 **Why is segmentation useful?**
Segmentation enables efficient memory allocation and protection by restricting access to certain regions (e.g., user processes cannot modify the kernel segment).

2.2 Paging Mechanisms for Memory Isolation

Definition:
Paging is a memory management scheme that eliminates fragmentation by dividing memory into fixed-sized blocks called **pages** (in RAM) and **frames** (in physical memory).

Mechanism:

1. The process address space is divided into **pages** of equal size (e.g., 4 KB).
2. The physical memory is divided into **frames** of the same size.
3. The OS maintains a **Page Table** that maps virtual page numbers (VPNs) to physical frame numbers (PFNs).

Key Benefits:
✅ No external fragmentation (unlike segmentation).

✅ Enables **virtual memory**, allowing processes to run even if they are not fully in RAM.
✅ Provides **memory isolation**, preventing one process from accessing another's memory.

💡 **Practical Example – Page Tables in Linux**
Linux maintains **multi-level page tables**:

- **Page Global Directory (PGD)** → Top-level table.
- **Page Middle Directory (PMD)** → Intermediate table.
- **Page Table Entry (PTE)** → Maps a virtual page to a physical frame.

2.3 Address Translation by the MMU

Memory Management Unit (MMU):
The MMU is a hardware component that performs **address translation** from virtual addresses (used by applications) to physical addresses (used by hardware).

Steps in Address Translation:

1. CPU generates a **virtual address** (VA).
2. MMU looks up the **Page Table** to find the corresponding **physical address** (PA).
3. If the page is **not present in memory** (Page Fault), the OS loads it from disk.
4. The MMU updates the **TLB (Translation Lookaside Buffer)** for fast lookups.

Practical Example in C – Simulating Paging:

```
#include <stdio.h>

#define PAGE_SIZE 4096  // 4 KB pages
#define FRAME_COUNT 8    // Physical memory has 8 frames

int page_table[FRAME_COUNT] = {2, 4, 7, 1, 5, 3, 6, 0};  // Mapping of virtual pages to
physical frames

void translate_address(int virtual_page) {
   if (virtual_page >= FRAME_COUNT) {
      printf("Page Fault! Virtual page %d not found.\n", virtual_page);
      return;
   }
   int physical_frame = page_table[virtual_page];
   printf("Virtual Page %d → Physical Frame %d\n", virtual_page, physical_frame);
}

int main() {
   translate_address(3);
   translate_address(5);
   translate_address(10);  // Simulating a page fault
   return 0;
```

```
}
```

📌 **Output Example:**

Virtual Page 3 → Physical Frame 1
Virtual Page 5 → Physical Frame 3
Page Fault! Virtual page 10 not found.

---

3. User-Space to Kernel-Space Transition

3.1 Transition Mechanism

3.1.1 Understanding User-Space to Kernel-Space Transition

Definition:
A system transitions from **user mode** (restricted operations) to **kernel mode** (privileged operations) when an application needs access to hardware or sensitive resources.

How does this happen?

- **System Calls (Syscalls)** – The primary method for safe transition.
- **Interrupts & Exceptions** – Hardware triggers requiring OS intervention.
- **Traps & Faults** – Special CPU instructions that cause controlled transitions.

🔹 **Example:** When a program executes `printf()`, it internally calls the **write syscall**, which interacts with the OS to write data to the terminal.

3.1.2 Role of System Calls (Syscalls)

Definition:
A system call is a controlled way for user applications to request privileged operations from the OS.

Steps in a Syscall Execution:

1. The application invokes a library function (e.g., `open()` in `stdio.h`).
2. The function triggers a **software interrupt** (e.g., `int 0x80` in x86, `syscall` in x86_64).
3. The CPU switches to **kernel mode**, and the OS handles the request.
4. The OS performs the operation (e.g., reading a file).
5. Control returns to the user-space application.

Practical Example – Calling a Syscall in C:

```
#include <stdio.h>
#include <unistd.h>

int main() {
    char buffer[50];
    int bytes = read(STDIN_FILENO, buffer, sizeof(buffer));
    write(STDOUT_FILENO, buffer, bytes);
    return 0;
}
```

Here, `read()` and `write()` use **syscalls** internally.

3.2 Privilege Escalation

3.2.1 How Syscalls Enable Privilege Escalation

Privilege Escalation:
This occurs when a process gains higher privileges than intended (e.g., a user process obtaining root privileges).

💡 **How it happens?**

1. **Legitimate Escalation** – Running `sudo` to gain admin access.
2. **Exploit-Based Escalation** – Using buffer overflow attacks or misconfigured permissions.

🔹 **Example:**

```
sudo su  # Elevates to root privileges
```

3.2.2 Syscall Handling in the Kernel

How Linux Handles a Syscall?

1. The syscall number is stored in **registers** (e.g., `rax` in x86_64).
2. The CPU executes the **syscall instruction**, transferring control to the kernel.
3. The **system call dispatcher** looks up the syscall table.
4. The corresponding kernel function executes the requested operation.
5. The result is returned to the user process.

Example – Listing Syscalls in Linux:

```
cat /proc/syscall
strace ls  # Shows syscalls used by 'ls' command
```

Additional Resources

📌 **Reference Books & Articles:**

1. **"Understanding the Linux Kernel"** – Daniel P. Bovet, Marco Cesati
2. **"Operating System Concepts"** – Abraham Silberschatz

📌 **Online Resources:**

- Linux Kernel Docs: https://www.kernel.org/doc/
- OSDev Wiki: https://wiki.osdev.org
- IBM Developer: https://developer.ibm.com
