# Linux Kernel Module Development and Ioctl Implementation

## Overview

This report provides an in-depth examination of the processes involved in Linux kernel module development and the implementation of ioctl. The exploration is organized into three main sections: the creation of Linux Kernel Modules (LKMs) focused on process management and memory mapping, the development of an ioctl device driver for memory operations, and the use of the procfs filesystem to introduce new file entries. The report details the objectives, implementation steps, and results of these tasks.

## 1. Linux Kernel Modules (LKMs)

The initial phase of the project involved creating several LKMs that interact with the Linux kernel to perform specific functions:

- **1.1 Hello LKM!**
  A simple kernel module, `helloworld.ko`, was created to print "Hello World" when loaded and "Module Unloaded" upon unloading. This served as an introductory exercise to grasp the basics of loading and unloading kernel modules.
- **1.2 Delving into Linux**
  A series of LKMs were developed to provide deeper insights into the Linux kernel:
  - **lkm1.c**: Lists all running or runnable processes by iterating over the `task_struct` linked list, printing each process's ID (pid).
  - **lkm2.c**: Accepts a process ID as input and prints the pid and state of its child processes, using `task_struct` to access child processes.
  - **lkm3.c**: Takes a process ID and a virtual address as inputs. It checks if the virtual address is mapped and, if so, prints the pid, virtual address, and corresponding physical address by traversing memory management structures like `mm_struct` and `vm_area_struct`.
  - **lkm4.c**: For a given process ID, calculates the size of the allocated virtual address space and the mapped physical address space. A test program was created to allocate memory incrementally and observe memory statistics using this LKM.

  A shared Makefile or Kbuild file was used to compile all four modules, facilitating efficient management and compilation.

## 2. ioctl - A Single System Call for Multiple Purposes

The ioctl system call, crucial for low-level operations and device control, was used to develop a device driver handling memory operations and process management:

- **Memory Mapping:** The ioctl driver retrieves the physical address for a given virtual address and enables writing a byte value at a physical memory address. A user-space program was created to allocate memory, assign values, and use ioctl calls to access and modify physical addresses.

- **Process Management:** Another ioctl function was implemented to change the parent process of a specified process ID, allowing a central station process to be notified when a "soldier" process (a simulated Linux process) exits.

### 3. procfs - A Filesystem Without Files

This section explores the procfs filesystem, which facilitates communication between the kernel and user space:

- **hello_procfs.c:** This kernel module adds a new entry in the /proc filesystem named `/proc/hello_procfs`. Users can read a "Hello World!" message using the `cat` command. The module implements necessary file operations using `proc_create`, `single_open`, `seq_printf`, and `remove_proc_entry`.
- **get_pgfaults.c:** Introduces a new entry `/proc/get_pgfaults` that displays the total count of page faults since the system's boot. It uses the `all_vm_events` function to retrieve page fault information.

### Conclusion

This report highlights the significant learning experiences gained from exploring Linux internals through kernel module development and ioctl implementation. The tasks provided valuable insights into process management, memory mapping, and system calls, demonstrating the essential role of kernel modules and ioctl in Linux system programming.

---

## Virtualization in Cloud Computing and Hypervisor Implementation Using KVM

### Overview

Virtualization is a cornerstone of cloud computing, enabling efficient resource allocation and the simultaneous execution of multiple operating systems on a single physical machine. As cloud computing scales, the demand for reliable and efficient virtualization techniques increases. Traditional approaches like para- and full virtualization have limitations, particularly on x86 architectures, due to their complexity and challenges in emulating x86 instructions. To address these issues, hardware vendors have introduced native support for x86 virtualization.

Kernel-based Virtual Machine (KVM) leverages this hardware support to enable the creation of hypervisors within the Linux operating system. This report discusses the development of a user-space hypervisor using KVM, along with advanced hypercalls and scheduling techniques designed to optimize resource usage in a virtualized environment.

### 1. DIY Hypervisor with KVM

The hypervisor created in this project utilizes the KVM API to build, manage, and execute virtual machines (VMs). Below are the key steps involved in setting up and running a VM in long mode:

1. **Open /dev/kvm:** The process begins by opening the /dev/kvm device file to access KVM capabilities.

2. **Create VM with KVM_CREATE_VM:** A new virtual machine instance is created using the `KVM_CREATE_VM` ioctl call.
3. **Allocate Memory with KVM_SET_USER_MEMORY_REGION:** Memory is allocated to the VM by mapping user-space memory to the VM's address space using the `KVM_SET_USER_MEMORY_REGION` ioctl call.
4. **Create vCPU with KVM_CREATE_VCPU:** A virtual CPU (vCPU) is created for the VM using the `KVM_CREATE_VCPU` ioctl call.
5. **Setup vCPU Registers:** The vCPU's registers are initialized to ensure the guest starts in the correct mode (long mode).
6. **Run vCPU with KVM_RUN:** The `KVM_RUN` ioctl call is used to execute the vCPU, entering the main execution loop.
7. **Handle VM_EXIT:** The hypervisor manages VM exits, such as I/O operations, by interpreting the exit reason and taking appropriate actions.
8. **Print Output via Hypercalls:** Hypercalls are implemented to allow the guest to request I/O operations, like printing values or strings.
9. **Translate Addresses:** The `KVM_TRANSLATE` ioctl call converts guest virtual addresses to host virtual addresses, providing a view into the guest memory from the host.
10. **Graceful Shutdown:** The VM is shut down gracefully upon completion or receiving an exit signal, with all resources being freed.

**KVM API Overview**

- **KVM_CREATE_VM:** Initializes a new VM instance.
- **KVM_SET_USER_MEMORY_REGION:** Maps user-space memory to the VM's address space.
- **KVM_CREATE_VCPU:** Creates a vCPU for the VM, essential for executing instructions.
- **KVM_RUN:** Executes the vCPU, entering the VM's main execution loop and handling VM exits.
- **KVM_TRANSLATE:** Translates guest virtual addresses to host virtual addresses.

**2. Hypercalls and Scheduling in the Matrix Cloud**

In the Matrix Cloud, a custom KVM hypervisor is designed to efficiently manage two guest operating systems (Neo and Morpheus) on a single CPU. Neo's VM receives 70% of the CPU time, while Morpheus' VM receives 30%, based on a special Service Level Agreement (SLA). This scheduling approach ensures optimized resource utilization and service availability.

- **Controlled Scheduling:** The vCPUs are managed without using pthreads, ensuring they run alternately on the same core. This is done by modifying the main thread to switch between VMs on each KVM_EXIT_IO, ensuring controlled execution of VMs.
- **Timer-Based Preemption:** A timer-based mechanism is introduced to handle scenarios with no I/O operations. The `timer_create` function is used to issue a signal at specified intervals. Upon receiving the signal, the VM exits, and the user-space program schedules the other VM. The `KVM_SET_SIGNAL_MASK` ioctl call ensures proper signal handling within the KVM environment.

# Chapter 1

# Introduction to KSM

KSM stands for kernel-same page merging. It's a powerful tool used in Linux for de-duplication. It optimizes memory usage and improves the performance of the system.

## 1.1 Working of KSM

KSM works by continuously scanning the pages in the memory; if two pages are the same, then it performs de-duplication on them. It keeps only one copy of the page and lets the processes share it between them. Uses the COW flags on the shared pages. So whenever a process modifies a shared page kernel will create a duplicate copy and let that process modify that page.

## 1.2 Setup for experiments

We have used intel i5 13th generation CPU, 16 GB ram and given each VM 4 GB of RAM and 2 cores each.

## Chapter 2

# Running VMs without workloads

We have written a Python script that reads the proc files in "/sys/kernel/mm/ksm/" every 5 seconds and stores the results in the file. So, after starting the script, we started the first VM. KSM was on, so it started de-duplicating the pages that can be merged. this continued till some time, so the graph is flattened on that time. After this, we started another VM, this will again the shared pages as de-duplication is happening. This will also get flattened after some time because no pages will be left which are potential candidates for merging.

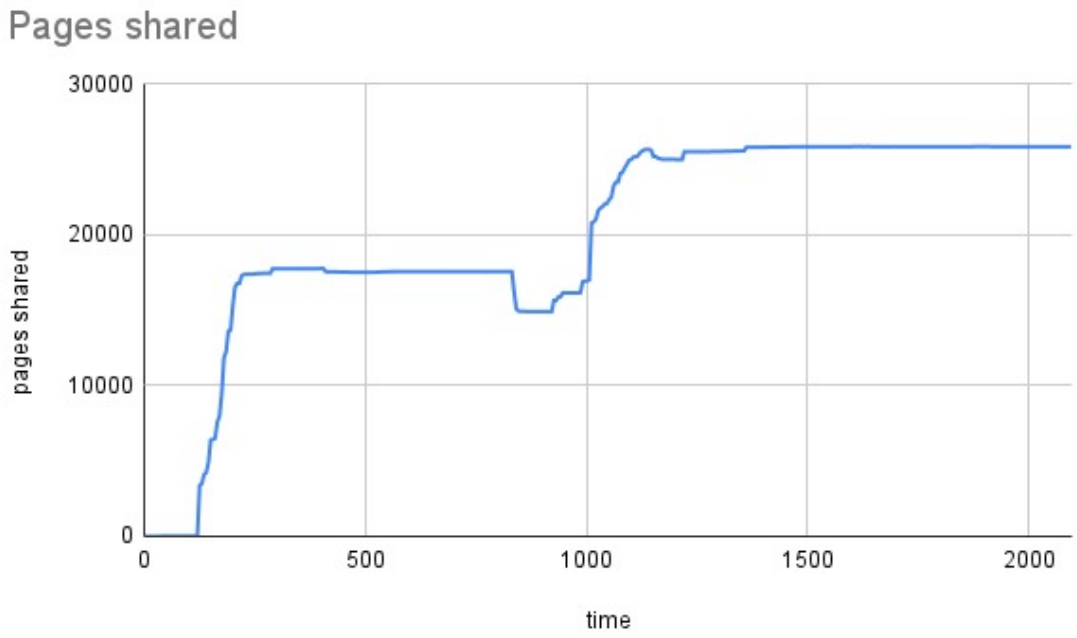


Figure 2.1: Plot between pages shared and time

Another plot is created for the general profit and the time. which shows how effective the KSM is. it has the same behaviour as the pages shared. after the first VM is started,

general profit keeps on increasing up to a certain limit, then flatterns out. When second VM starts, profit again starts increasing and flattens out after some time.

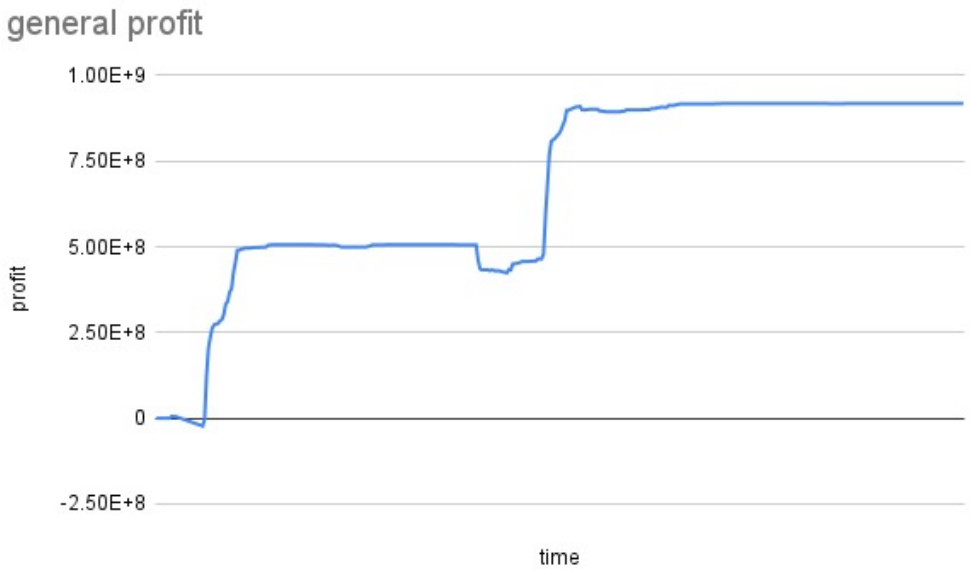


Figure 2.2: Plot between General profit and time