CS307 - Operating Systems
PA4 Report
Extending a Virtual Memory Implementation with Paging with Semaphores

Barış Pome
Student ID: 31311
Sabancı University

Fall 2024-2025

Introduction

This assignment focuses on extending the functionality of a simple virtual memory (VM) system by implementing paging and adding new system calls for process management. The primary goal is to separate the virtual address space (VAS) from the physical memory through address translation using page tables. Additionally, the implementation supports multiple processes by introducing key system calls such as `yield` (to facilitate context switching) and `brk` (to dynamically allocate or deallocate heap memory).

The enhancements were implemented within the provided `vm.c` file, strictly adhering to the boundaries defined in the assignment. These modifications include the introduction of new methods, helper functions, and updates to existing system calls and memory access routines. This report details the overall approach, key implementation decisions, and helper methods introduced to improve code clarity and efficiency.

The implementation emphasizes modularity and correctness, with helper functions created for tasks such as managing page table entries, handling bitmaps for memory allocation, and performing address translations. These functions not only simplify the main implementation but also ensure compliance with the assignment's specifications for managing virtual and physical memory effectively.

1 Program Overview

The program enhances an existing virtual memory implementation by introducing a paging mechanism and system calls to manage multiple processes. Key features include:

- **Paging Implementation**: Virtual memory addresses are translated to physical memory using page tables. Each virtual address is divided into a Virtual Page Number (VPN) and an offset, ensuring efficient memory management.
- **System Calls**:
  - `yield`: Facilitates context switching between processes.
  - `brk`: Dynamically allocates or deallocates heap memory during runtime.
- **Memory Access Modifications**: Updates to `mr` and `mw` functions ensure memory accesses are translated correctly, handling segmentation faults and access violations effectively.

- **Helper Functions**: Helper functions were developed to manage page table entries, process metadata, and bitmap operations for free and allocated frames. These functions improve modularity and readability.

The program supports process creation, memory allocation, and dynamic heap management while adhering to the constraints of the provided framework. Testing was performed to validate the correctness of each component.

2 Detailed Explanation of the Code

This section provides a comprehensive explanation of the functions implemented, divided into helper functions, core methods, and an overall description of the code functionality.

2.1 Helper Functions

The following helper functions were implemented to enhance modularity and readability:

- **create_page_table_entry**: This function creates a page table entry by combining the frame number, read permission, and write permission. It sets the valid bit to indicate that the page is active.
- **is_page_valid**: Checks whether a given page table entry is valid by inspecting its valid bit.
- **has_read_permission**: Determines whether a given page table entry has read permission by inspecting its corresponding bit.
- **has_write_permission**: Determines whether a given page table entry has write permission by inspecting its corresponding bit.
- **get_frame_number**: Extracts the frame number from a page table entry by shifting out irrelevant bits.
- **set_frame_used** and **set_frame_free**: These functions manipulate the free bitmap to mark frames as used or free, respectively. They operate on the OS bitmap region.
- **is_frame_free**: Checks if a frame is available by evaluating the bitmap for free frames.

- **get_pcb_base** and **get_page_table_base**: These functions compute the base addresses for the Process Control Block (PCB) and page table for a given process ID.
- **is_process_terminated**: Determines if a process has terminated by checking if its PID field in the PCB is set to a special value (`0xffff`).
- **get_virtual_page_number** and **get_page_offset**: Extract the virtual page number and page offset from a virtual address.
- **get_physical_address**: Combines a frame number and offset to compute the corresponding physical memory address.

These helper functions simplify memory management, address translation, and process handling.

2.2 Core Methods

The following core methods were implemented or modified:

- **initOS**: Initializes the OS memory by setting the bitmap for free frames, clearing the process count, and preparing status registers.
- **createProc**: Creates a new process by assigning a unique process ID, initializing its PCB, allocating memory for its code and heap segments, and loading segment data from files.
- **loadProc**: Loads a process into the CPU registers by setting the Program Counter (`RPC`) and the Page Table Base Register (`PTBR`) based on the PCB values.
- **allocMem**: Allocates memory for a virtual page by finding a free frame in the bitmap, marking it as used, and creating a valid page table entry with appropriate permissions.
- **freeMem**: Frees a virtual page by marking its frame as available in the bitmap and invalidating its page table entry.
- **tbrk**: Implements the `brk` system call, allowing dynamic allocation or deallocation of heap pages based on the `R0` register value.
- **tyld**: Implements the `yield` system call, saving the current process state, finding the next runnable process, and switching the CPU context.

- **thalt**: Implements the `halt` system call, terminating the current process by freeing all allocated pages, marking the PCB as terminated, and switching to the next runnable process.
- **mr** and **mw**: Modified memory read and write functions to incorporate address translation, permission checks, and segmentation fault handling.

These methods form the core functionality for managing virtual memory and processes.

2.3 What Does the Code Do?

The implemented code extends the basic virtual memory system to include paging and process management features. It enables:

- Efficient memory allocation and deallocation using page tables and bitmaps.
- Address translation to map virtual addresses to physical memory.
- Dynamic heap management via the `brk` system call.
- Context switching between multiple processes using the `yield` system call.
- Proper cleanup of terminated processes and memory resources through the `halt` system call.
- Robust error handling for segmentation faults and invalid memory access during read and write operations.

Overall, the code implements a modular, extensible virtual memory system that adheres to the assignment's requirements and supports key operating system concepts such as paging and process management.

3 Conclusion

In this assignment, the virtual memory system was successfully extended to include paging and process management capabilities. The implementation introduced a paging mechanism to enable address translation and efficient memory allocation while supporting dynamic heap management. Key system calls such as `yield` and `brk` were developed to handle process context switching and memory management dynamically.

The use of helper functions improved code readability and modularity, while robust error handling ensured reliability in addressing segmentation faults and invalid memory accesses. Overall, the assignment met the objectives and provided a deeper understanding of core operating system concepts, such as memory management, paging, and system call design.