mCertiKOS Lab 1: Memory Management Analysis

The CS422 Assignment 1 focuses on memory management within the mCertiKOS operating system, a research-oriented educational kernel designed to demonstrate operating system principles. This comprehensive analysis examines the lab structure, component architecture, and implementation requirements for understanding the foundational memory management systems.

Understanding the mCertiKOS Architecture

The mCertiKOS operating system lab is divided into three major components, each focusing on different aspects of systems programming and kernel development. The first part introduces students to x86 assembly language, the QEMU emulator, and the PC bootstrap process. This foundational knowledge provides critical context for understanding how the kernel initializes and interacts with hardware. While no code implementation is required for this section, mastering these concepts is essential for successful completion of later parts.

The lab environment utilizes Git for version control, allowing students to track changes, commit completed work, and submit assignments. This infrastructure mirrors professional software development practices and encourages systematic implementation and testing. The repository structure maintains different branches for each lab assignment, ensuring students can progress through increasingly complex implementations while maintaining access to previous work.

When successfully built and executed, the kernel initializes device drivers and data structures before entering a simple monitor interface. This interface supports basic commands such as "help" and "kerninfo," providing information about the kernel's memory footprint and symbol locations. Though limited in functionality, this environment demonstrates that the kernel is running directly on simulated hardware, illustrating the low-level nature of operating system development.

PC Bootstrap and Memory Architecture

A critical component of understanding operating system functionality is familiarity with the PC bootstrap process and physical memory layout. The lab provides detailed insights into how a computer initializes from power-on, including the BIOS startup

sequence and transition to the bootloader. This process reveals the hierarchical nature of system initialization, where increasingly complex code components take control as the system boots.

The physical address space of a PC follows a standardized layout with specific regions allocated for different purposes. The lab output demonstrates this through the BIOS-e820 memory map, which identifies usable RAM regions (0x00000000-0x0009fbff and 0x00100000-0x7fffdfff) separated by reserved memory segments. Understanding this memory map is crucial for implementing proper memory management, as it defines the boundaries within which the kernel can operate[1].

The bootloader, residing in the boot directory, consists of two stages (boot0 and boot1) that handle the transition from BIOS execution to kernel initialization. These components must operate within strict size constraints, with boot0 limited to 446 bytes and boot1 to 31744 bytes. This limitation emphasizes the importance of efficient coding practices in early-stage boot components.

Physical Memory Management Framework

The core focus of the lab appears to be implementing the physical memory management layers for the mCertiKOS kernel. This subsystem is organized into multiple interconnected modules housed within the kern/pmm directory. These modules implement a layered approach to memory management, with each layer building upon functionality provided by lower layers.

The physical memory management implementation is divided into at least three key components:

1. MATIntro (Memory Allocation Table Introduction): This component likely establishes the fundamental data structures and concepts for physical memory management.
2. MATInit (Memory Allocation Table Initialization): This module handles the initialization of memory structures, preparing them for operational use.
3. MATOp (Memory Allocation Table Operations): This component implements the core functionality for memory allocation and deallocation operations[1].

This modular design follows principles of layered abstraction, allowing each component to focus on specific aspects of memory management while building upon previously established functionality. Such organization improves maintainability and allows for systematic testing of each layer's implementation.

Kernel Architecture and Components

The mCertiKOS kernel is structured into several logical subsystems, each responsible for specific functionality. The compilation output reveals key components including device drivers (video, console, serial), library functions (string manipulation, debugging, printing), and initialization code. This modular approach allows for clear separation of concerns and systematic development.
Device drivers in the kern/dev directory handle hardware-specific operations, abstracting device details to provide consistent interfaces for the rest of the kernel. This abstraction is particularly important for console output, which the kernel directs to both the virtual VGA display and the simulated PC's serial port, enabling flexible debugging approaches.

Library functions in kern/lib provide essential utilities such as string manipulation, formatting, and debugging support. These components form the foundational tools that higher-level kernel functions utilize to implement more complex behavior. The seg.c and x86.c modules specifically deal with x86 architecture details, encapsulating platform-specific code.

Development Environment and Tools

The lab utilizes industry-standard development tools, particularly the QEMU emulator for simulating a complete PC environment. This approach offers significant advantages over development on physical hardware, particularly for debugging purposes. QEMU can be integrated with the GNU Debugger (GDB), allowing students to set breakpoints and step through code execution, even during the early boot process[1].
The compilation process utilizes standard GNU tools including the assembler (as) for assembly files, compiler (cc) for C files, and linker (ld) to create the final executable objects. The make system orchestrates this process, building the bootloader and kernel components before combining them into a disk image (certikos.img) that QEMU can boot from.
A significant aspect of the development environment is the ability to interact with the kernel through either the virtual VGA display or the serial console. This flexibility allows developers to work in various environments, including remote SSH sessions where graphical output may not be available or convenient.

Understanding mCertiKOS : Memory Management in Lab 2

This report provides a comprehensive analysis of the Lab 2 assignment for CS422, which focuses on implementing critical memory management components in the mCertiKOS operating system. The assignment builds upon previous work on physical memory management and introduces advanced concepts related to virtual memory management. While implementation is not required at this time, a thorough understanding of the underlying concepts, architecture, and required functions is essential for future development.

Container Management System

The first part of the assignment continues work on physical memory management by implementing a container management layer. Containers are fundamental objects within the mCertiKOS architecture that serve a critical role in resource tracking and process management. The container abstraction provides a structured approach to memory allocation and resource distribution within the operating system framework. Containers in mCertiKOS are specifically designed to track resource usage of processes and maintain parent-child relationships between them. This mechanism is crucial for preventing resource exhaustion attacks, where malicious processes might attempt to consume all available system resources. Currently, the container system primarily tracks page allocations, but its architecture allows for potential extension to monitor other resources such as CPU time. Each process within the system is assigned a unique identifier (ID) in the range [0, NUM_IDS), with ID 0 being reserved exclusively for the kernel itself.

The container associated with each ID maintains several important fields: quota (maximum number of pages allowed), usage (currently allocated pages), parent (ID of the parent), n_children (number of children), and used (boolean indicating whether the ID is in use). This structure establishes a hierarchical relationship between processes, enabling parent processes to distribute resources to their children. The root container (ID 0) serves as the foundation of this tree structure and is the only container without a parent.

Throughout system execution, the container mechanism is utilized in two primary scenarios: during page allocation requests (such as handling page faults or malloc system calls) and when spawning new IDs (where parent IDs must distribute quota to newly created children). To ensure proper resource management, the system maintains a critical "Soundness" invariant: the sum of available quotas (quota minus usage) across all used IDs must never exceed the total number of pages available for allocation.

Container Functions

The MContainer layer requires implementation of several essential functions:

1. container_init - Initializes the container system
2. container_get_parent - Retrieves the parent ID of a given container
3. container_get_nchildren - Returns the number of children for a container
4. container_get_quota - Retrieves the quota assigned to a container
5. container_get_usage - Gets the current resource usage of a container
6. container_can_consume - Checks if a container can consume additional resources
7. container_split - Divides resources between parent and child containers
8. container_alloc - Allocates resources to a container
9. container_free - Releases resources from a container[1]

These functions collectively provide the foundation for resource management within the mCertiKOS system, enabling controlled allocation and deallocation of memory resources while maintaining system integrity through adherence to the Soundness invariant.

Virtual Memory Management

The second and more substantial part of the assignment focuses on implementing page-based virtual memory management for mCertiKOS. This component is responsible for configuring the Memory Management Unit (MMU) of the CPU to properly translate between virtual and physical memory addresses, a fundamental aspect of modern operating systems.

Address Translation in x86 Architecture

Understanding virtual memory management requires a thorough grasp of the x86 protected-mode memory management architecture, which encompasses both segmentation and page translation mechanisms. In x86 terminology, a virtual address

consists of a segment selector and an offset within the segment. This virtual address undergoes segment translation to produce a linear address, which then undergoes page translation to yield the physical address that ultimately accesses RAM[1].
In the context of C programming within mCertiKOS, pointers represent the "offset" component of virtual addresses. Although the x86 architecture utilizes segmentation, mCertiKOS effectively neutralizes its impact by configuring the Global Descriptor Table (GDT) with segment base addresses set to 0 and limits set to 0xffffffff, resulting in linear addresses that match the virtual address offsets. This configuration allows the development focus to remain on page translation rather than segmentation[1].
Once the processor enters protected mode with paging enabled, all memory references are interpreted as virtual addresses and translated by the MMU. This translation mechanism means that all C pointers are virtual addresses, requiring the kernel to handle various address types carefully. The kernel frequently needs to manipulate addresses as opaque values or integers without dereferencing them, particularly within the physical memory allocator.

Identity Mapping in mCertiKOS

A significant challenge in kernel development arises when the kernel needs to access memory using physical addresses, which cannot be directly used once paging is enabled. mCertiKOS addresses this challenge through identity mapping, where a virtual address maps to the identical physical address. Specifically, mCertiKOS reserves page structure index 0 for the kernel process and configures it as an identity map. This approach allows the kernel to access physical addresses by switching to page structure 0, where virtual addresses match their corresponding physical addresses.

Page Structure Implementation

The page-based virtual memory implementation requires configuring several layers that collectively establish the translation from virtual to physical addresses. These layers include:

MPTIntro Layer

This introductory layer establishes the foundation for page table management. It defines the structures and initial settings needed for manipulating page tables[1].

MPTOp Layer

The MPTOp layer provides basic operations for page table manipulation, including functions to modify entries, set permissions, and manage page table structures[1].

MPTComm Layer

This layer implements common page table operations that are frequently used throughout the kernel, abstracting the lower-level details and providing a more convenient interface[1].

MPTKern Layer

The MPTKern layer specifically handles page table operations for the kernel space, ensuring proper memory management for kernel processes[1].

MPTNew Layer

This layer is responsible for creating new page table structures, particularly when spawning new processes that require their own virtual memory space[1].

Testing Methodology

The assignment includes a comprehensive testing framework to validate the correctness of implementations. Tests can be executed using the command `make TEST=1`, which runs a series of predefined test cases for each layer. The framework checks whether implementations maintain the required invariants and correctly handle various edge cases[1].

Additionally, students are encouraged to develop their own test cases to thoroughly challenge their implementations. These custom tests should be incorporated into the provided test functions in each layer's test.c file

Understanding mCertiKOS Lab 3: Trap Handling and Process Management

This comprehensive overview examines the structure and requirements of Lab 3 for the CS422 course, focused on implementing key kernel facilities in mCertiKOS to enable user-mode processes and handle traps, interrupts, and exceptions. The assignment builds upon previous work and introduces critical operating system concepts related to process management and protected mode transitions.

Lab Overview and Structure

Lab 3 is divided into two interconnected parts that collectively enable mCertiKOS to run multiple protected user-mode processes and handle various system events. The first part focuses on implementing thread and process management facilities, while the second part deals with trap handling mechanisms. This lab represents a significant milestone in transforming mCertiKOS into a functional operating system capable of running and managing user-level processes[1].

The assignment builds upon previous labs, requiring students to merge their existing Lab 2 code with the new Lab 3 codebase using Git. The lab environment includes several sample user processes that demonstrate memory isolation between processes. These processes include an idle process and three test processes (ping, pong, and ding) that show how each process owns its own virtual address space with full memory isolation and protection.

User Process Design in mCertiKOS

In the mCertiKOS architecture, every user-level process has a corresponding kernel service thread. This design creates a one-to-one mapping between user processes and their kernel handlers. When a process makes a system call or yields control, it traps into its corresponding kernel thread, which then manages the context switch process. This

involves switching page structures, saving process states, scheduling the next thread, and eventually restoring the appropriate context to resume execution[1].
The lab introduces the `startuser` monitor task, which initiates the idle process. This idle process then spawns three user processes that demonstrate memory isolation. Since preemption is not yet implemented, these processes must explicitly yield control to allow other processes to run, highlighting one of the limitations that will be addressed in future lab assignments.

Part 1: Thread and Process Management

The first part of the lab introduces several layers that collectively build the process management infrastructure:

PKCtxIntro Layer - Kernel Context Management

This layer introduces the concept of kernel thread context, which includes six register values that need to be saved and restored when switching between kernel threads. The most critical implementation in this layer is the `cswitch` assembly function, which must carefully save the current thread's state and restore the new thread's state.

PKCtxNew Layer - Creating New Kernel Contexts

This layer implements functionality for creating new kernel contexts for child processes. The `kctx_new` function must be implemented to initialize a new kernel context structure with appropriate values to enable a new thread to execute properly when scheduled.

Thread Control Block (TCB) Management

The PTCBIntro layer introduces Thread Control Blocks, which are data structures that store information about each thread. The PTCBInit layer builds upon this by implementing the `tcb_init` function to initialize all TCBs. This initialization process is crucial for establishing the foundation of the thread management system.

Thread Queue Management

Thread queues are essential for scheduling and maintaining thread states. The PTQueueIntro layer introduces these queues, while the PTQueueInit layer implements functions to initialize and manipulate them. Four key functions need to be implemented: `tqueue_init`, `tqueue_enqueue`, `tqueue_dequeue`, and `tqueue_remove`. These functions collectively enable thread scheduling and management.

Thread Operations and Process Creation

The PThread layer implements core thread operations, including `thread_spawn` for creating new threads and `thread_yield` for voluntarily releasing the CPU to another thread. The PProc layer then introduces functions for creating user-level processes, building upon the thread management capabilities.

Part 2: Trap Handling

The second part of the lab focuses on implementing exception and system call handling mechanisms, allowing the kernel to recover control from user-mode code when necessary. This implementation is fundamental to the operation of any protected-mode operating system[1].

Protected Control Transfer Concepts

The lab introduces key concepts related to interrupts and exceptions, which are both forms of protected control transfers. These mechanisms allow the processor to switch from user mode to kernel mode securely, without giving user-mode code any opportunity to interfere with kernel operations or other environments[1].
In Intel's terminology, the lab distinguishes between:

- Interrupts: Protected control transfers caused by asynchronous events external to the processor
- Exceptions: Protected control transfers caused synchronously by the currently running code

Understanding these concepts requires studying the x86 interrupt and exception mechanism as detailed in the Intel documentation. The processor's interrupt/exception mechanism is designed to ensure these transfers are genuinely protected, preventing user code from compromising system security or stability.

Testing and Evaluation

The lab includes built-in tests that can be run using the `make TEST=1` command. These tests verify the correct implementation of each layer. Students are also encouraged to write their own test cases to thoroughly validate their implementations and challenge their understanding of the concepts.

Understanding mCertiKOS Lab 4: Implementing Concurrency and Multiprocessor Support

This report provides a comprehensive analysis of the mCertiKOS Lab 4 assignment, which focuses on transforming the operating system from a single-processor cooperative multitasking system into a fully-featured multiprocessor preemptive system with synchronization capabilities. The lab is structured as a progressive implementation of multiple concurrent computing concepts, ultimately building towards a complete solution for the classic producer-consumer problem.

Assignment Structure and Overview

The Lab 4 assignment is divided into four interconnected parts, each building upon the previous one to incrementally add advanced operating system functionality. Students are required to modify the existing operating system kernel to support these new features, with the freedom to make arbitrary changes to the kernel source code. The assignment emphasizes the complexity of debugging concurrent systems and strongly recommends testing each component thoroughly before proceeding to the next implementation phase.

Unlike previous labs in the course, this assignment requires starting from a fresh codebase rather than building upon previous work. This is because the fundamental architectural changes needed for multiprocessor support necessitate significant modifications to existing code structures. The lab provides specific instructions for creating a new branch called 'lab4' based on the instructor-provided origin/lab4 branch[1].

The assignment includes a user-process testing framework that allows students to verify their implementations at each stage. This framework consists of two types of user programs ("ping" and "pong") that call system calls for producing and consuming values at different speeds, revealing different behavioral characteristics as more concurrency features are implemented.

Test Environment and Expected Behavior Progression

The lab includes an initial test setup consisting of user programs that demonstrate the progression of system capabilities as students implement each part. The key test applications are found in user/pingpong/ping.c and user/pingpong/pong.c, which call system calls produce and consume at different speeds. These system calls are designed to print out 5 numbers in sequence using a loop.
In the initialization process (defined in kern/init/init.c), the system creates two producer processes on CPU #1 and two consumer processes on CPU #2. This setup is specifically designed to reveal different levels of output interleaving as students implement more concurrency features.
The expected behavioral progression through the implementation phases is carefully described:

1. Initial state: Only one user process on each processor runs, with perfectly ordered outputs
2. After Part 1 (multiprocessor support): Producer and consumer outputs interleave arbitrarily, but still only one process per CPU executes
3. After Part 2 (preemptive scheduler): Both processes on each CPU execute, with potential alternation of outputs
4. After Part 3 (kernel preemption): Numbers within individual system calls become randomly interleaved due to preemption during kernel execution

This progressive behavior provides a clear indication of successful implementation at each stage, making debugging and verification more manageable.

Part 1: Multiprocessor Support Implementation

The first major component of this lab focuses on implementing "symmetric multiprocessing" (SMP) in mCertiKOS. In this architecture, all CPUs have equivalent access to system resources such as memory and I/O buses. While functionally identical, processors are classified into two categories during the boot process:

1. Bootstrap Processor (BSP): Responsible for initializing the system and booting the operating system

2. Application Processors (APs): Activated by the BSP only after the operating system is up and running

The hardware and BIOS determine which processor serves as the BSP. Prior to this lab, all existing mCertiKOS code has been running exclusively on the BSP. The implementation requires understanding the role of Local APIC (LAPIC) units, which are responsible for delivering interrupts throughout the system and providing each CPU with a unique identifier.
The lab provides LAPIC driver implementation in kern/dev/lapic.h and kern/dev/lapic.c, though students are not expected to fully understand this low-level code. Before booting the Application Processors, the Bootstrap Processor must collect system information including the total number of CPUs, their APIC IDs, and the MMIO address of the LAPIC unit.

Part 2: Preemptive Scheduler Implementation

The second part of the lab addresses a limitation observed in the initial test setup: only one process per CPU ever executes. This occurs because the current scheduler implements cooperative multitasking, where processes must explicitly yield control of the CPU through a system call. As demonstrated in user/lib/entry.S, user programs no longer wrap execution with yield calls, instead simply spinning once their main function returns.
To address this limitation, students need to implement a preemptive scheduler that can interrupt running processes based on timer events rather than relying on explicit cooperation. After successful implementation, both user processes on each CPU should execute, with their outputs potentially alternating as processes get preempted during execution.
This preemptive capability is fundamental to modern operating systems and represents a significant enhancement to the basic functionality of mCertiKOS.

Part 3: Kernel Preemption

The third component builds upon the preemptive scheduler by enabling interrupts within specific kernel sections. Initially, even with the preemptive scheduler implemented, system call handlers like sys_produce and sys_consume execute without interruption because interrupts are disabled when trapping into the kernel (as seen in the _alltraps function in kern/dev/idt.S).
This part requires students to enable interrupts in system call handlers for producer and consumer operations, allowing timer interrupts to occur during kernel execution. After successfully implementing this feature, the printed numbers should appear randomly

interleaved, reflecting the reality that even kernel code can be preempted during execution[1].
This phase introduces the complexity of ensuring kernel code remains correct when execution can be interrupted at almost any point, requiring careful management of shared data structures.

Part 4: Producer-Consumer Problem Implementation

The final part of the lab challenges students to design and implement the classic producer-consumer problem (also known as the bounded-buffer problem) using shared objects and condition variables. This synchronization problem is fundamental in concurrent programming, requiring careful coordination between threads that produce data and threads that consume it.
For this part, students are given significant freedom to design their kernel structure, including adding, modifying, or deleting directories and files as needed. The assignment specifies that students must ensure all new files are added to the git repository before committing or pushing their work.
No specific test scripts are provided for this part of the assignment, encouraging students to develop their own comprehensive test cases to validate their implementation. This approach prepares students for real-world software engineering practices where developing thorough test suites is essential for reliable software.