- **Project Title:** kacchiOS: A Minimal Baremetal Operating System
- **Course:** Operating Systems
- **Submitted by:** Group A7
- **Date:**

1. Abstract

kacchiOS is a minimal, 32-bit baremetal operating system designed to demonstrate the fundamental concepts of kernel development. Built from scratch for the x86 architecture, it implements a multiboot-compliant bootloader, a basic physical memory manager, cooperative multitasking, and a serial I/O driver. This report details the architectural decisions and implementation strategies used to meet the core requirements of an educational OS.

2. System Overview

The system is a monolithic kernel that boots via GRUB/QEMU. It does not rely on standard libraries (freestanding implementation).

- **Architecture:** x86 (IA-32)
- **Boot Protocol:** Multiboot 1 Specification
- **Language:** C and x86 Assembly
- **Output:** Serial Port (COM1) redirection

3. Implementation Details

3.1 Bootstrapping The entry point is defined in `boot.S`. The code initializes a 16KB kernel stack (`stack_bottom` to `stack_top`), clears the BSS section, and transfers control to the C kernel via `kmain`.

3.2 Memory Management Memory management is handled by `memory.c`.

- **Strategy:** A "Bump Allocator" (Sequential Fit) is used.
- **Implementation:** The kernel tracks the end of the kernel image (`__kernel_end`). The `kalloc(size_t size)` function returns the current free address and advances the pointer, ensuring 4KB page alignment. This is used specifically for allocating process stacks.

3.3 Process Management The system supports multitasking via a Process Control Block (PCB) structure defined in `process.h`, which tracks the Stack Pointer (`esp`), Process ID (`pid`), and State (`state`).

- **Creation:** `process_create` allocates a new 4KB stack for the process. It manually constructs an initial stack frame containing a return address to `process_exit` and the entry point of the function.
- **Termination:** When a process function returns, it "returns" into `process_exit`, which marks the process as `TERMINATED` and yields the CPU.

3.4 Scheduling and Context Switching

- **Policy:** The scheduler (`schedule()` in `process.c`) employs a Round-Robin algorithm. It iterates through the process list starting from the current PID to find the next task in the `READY` state.
- **Context Switch:** The actual register swapping is performed in `switch.S`. The `context_switch` function saves the registers `ebx`, `esi`, `edi`, and `ebp` onto the old stack and loads the new stack pointer.
- **Cooperative Multitasking:** Processes voluntarily give up control using `yield()`, which calls the scheduler.

3.5 User Interface (Shell) A basic shell is implemented in `kernel.c`. It runs as a separate process, reading input from the serial port. It supports basic character echoing, backspacing, and command execution (echoing typed lines).

4. Results and Output

The system successfully boots and initializes the scheduler.

- **Initialization:** The kernel prints the banner and initializes memory/process tables.
- **Multitasking:** Three tasks run concurrently:
    1. **Shell Task:** Accepts user input.
    2. **Background Task (Task A):** Prints a running status message periodically.
    3. **Finite Task:** Runs for 3 iterations and then self-terminates.
- **Termination:** The Finite Task successfully exits, proving the `process_exit` logic works.

4.1 Output Analysis

The screenshot above captures the live execution of kacchiOS on the QEMU emulator. It provides visual validation of several key kernel features implemented in the code:

1. **Preemptive-style Interleaving (Cooperative Multitasking):**
   - The output shows `[Background Task] Running...` and `[Finite Task] I am alive!` appearing mixed with the shell prompt `kacchiOS>`.
   - For example, the line `this is group a[Finite Task] I am alive!` demonstrates that while the user was typing "this is group a", the scheduler performed a context switch to the Finite Task, which printed its message before control returned to the shell. This confirms that `schedule()` and `yield()` are successfully arbitrating CPU time between multiple processes.
2. **Process Termination Logic:**

    - After the Finite Task prints "I am alive!" three times (matching the loop `for (int i = 0; i < 3; i++)` in `kernel.c`), the message `Process Terminated.` appears.
    - This confirms that the stack manipulation in `process_create` correctly set the return address to `process_exit`. When `task_finite` finished, it automatically jumped to `process_exit`, which set its state to `TERMINATED` and removed it from the run queue.
3. **Shell and Serial I/O:**
    - The successful echoing of user commands (`You typed: hello`, `You typed: the bg task runs infinitely`) verifies that the Serial Driver (`serial.c`) is correctly polling the COM1 port using `inb(COM1 + 5)` and handling interrupts/status bits correctly to receive keyboard input.

5. Future Work

Future improvements planned for kacchiOS include:

- Implementing `kfree()` for memory deallocation.
- Adding a Programmable Interval Timer (PIT) driver for preemptive scheduling.
- Implementing Inter-Process Communication (IPC) mechanisms.

6. Conclusion

kacchiOS successfully demonstrates the core "Must Include" requirements of the project. It establishes a functional environment where memory is managed, processes are created and destroyed, and a scheduler arbitrates CPU time, providing a solid foundation for further OS research.