Rajshahi University of Engineering and Technology

Department of Computer Science and Engineering

kacchiOS Project Report

(Course Code: CSE 3202)

Course Title: Operating System Sessional

Submitted By:
Mohammad Muaz
Student ID: 2103129
Section: C

Submitted To:
Md. Farhan Shakib
Lecturer
Department of Computer Science and Engineering

Date of Submission: Jan 6, 2026

1 Introduction

An operating system is responsible for managing hardware resources and providing a controlled execution environment for programs. Understanding the internal working of an operating system requires hands-on experience with low-level mechanisms such as memory allocation, process scheduling, and context switching.

kacchiOS is developed as an educational operating system to explore these fundamental concepts. The system boots into a minimal kernel environment, supports dynamic memory allocation, creates and schedules multiple processes, and allows basic inter-process communication. The design follows a modular approach inspired by the XINU operating system.

2 Memory Management

2.1 Overview

Memory management in kacchiOS provides dynamic allocation and deallocation of kernel memory. The system supports both heap memory allocation and per-process stack allocation. The following components are implemented:

- Heap initialization (`meminit`)
- Heap allocation (`getmem`)
- Heap deallocation (`freemem`)
- Stack allocation (`getstk`)
- Stack deallocation (`freestk`)

2.2 Heap Initialization

The heap is initialized during kernel startup using the **`meminit`** function.The start of the heap is determined by the linker-provided symbol marking the end of the kernel, and the end of the heap is set to a fixed RAM boundary.A single large free memory block is created and inserted into a free list.

Data Structure

Free memory blocks are represented using a singly linked list:

```
struct memblk {
    struct memblk *mnext;
    uint32_t mlength;
};
```

The free list is maintained in sorted order by memory address to support efficient block coalescing.

2.3 Heap Allocation

The `getmem()` function is responsible for allocating heap memory on demand. It utilizes a **Best Fit** strategy to minimize external fragmentation.

Algorithm: Best Fit Allocation

1. **Size Alignment:** Round the requested size up to the nearest multiple of the system's block size.
2. **Search:** Traverse the free list to find a block where $size_{block} \geq size_{requested}$.
3. **Best Fit Selection:** Among all suitable blocks, select the one where the difference $(size_{block} - size_{requested})$ is minimal.
4. **Allocation:**
   - If an **exact match** is found: Remove the block from the free list and return it to the caller.
   - If a **larger block** is found: Split the block. Return the requested portion and keep the leftover portion in the free list.
5. **Termination:** If no suitable block exists after a full traversal, return `NULL`.

Data Structure

The heap is managed using a **singly linked list** of memory blocks (`memblk`). Each block records its size and pointer to next free block Best Fit helps reduce fragmentation compared to First Fit, making allocation efficient.

2.4 Heap Deallocation

The `freemem` function returns memory to the free list. The freed block is inserted at the correct position in the list, and adjacent free blocks are merged to reduce fragmentation.

2.5 Stack Management

Each process is allocated a private stack using `getstk` during process creation. Stacks grow downward, and memory is reclaimed using `freestk` when a process terminates.

3 Process Management

3.1 Overview

Process management provides abstractions for multi-tasking. Each process is represented by a Process Control Block (PCB) stored in proctab[].

3.2 Process Control Block

Each process is represented by a Process Control Block (PCB), stored in a global process table. The PCB stores information such as process ID, state, priority, stack pointer, and stack metadata.

```
1 struct pcb {
2     int pid;
3     int state;
4     int priority;
5     uint32_t *sp;
6     void *stack_base;
7     uint32_t stack_size;
8     ...
```

```
9 };
```

3.3 Process States

The following process states are supported:

- `PR_FREE` – Unused PCB
- `PR_READY` – Ready to run
- `PR_CURR` – Currently running
- `PR_BLOCKED` – Blocked (e.g., waiting for IPC)

3.4 Process Initialization

The `process_init` function initializes the process table and sets up the null process, which acts as the idle process and never terminates.

3.5 Process Creation

The `process_create` function performs the following steps:

1. Allocates a free process ID
2. Allocates a stack using `getstk`
3. Initializes the PCB fields
4. Constructs an initial stack frame

The initial stack frame is prepared such that the process starts execution at its entry function and returns safely to `process_exit` if it terminates.

3.6 Process Termination

When a process finishes execution, `process_exit` is called. This function frees the process stack, marks the PCB as free, and invokes the scheduler to select another process.

4 Scheduler

4.1 Scheduling Model

kacchiOS uses a **cooperative scheduling** model. Processes voluntarily yield the CPU using the `yield` function. No hardware timer interrupt is used.

4.2 Priority Scheduling

Each process has an associated priority. The scheduler selects the highest-priority ready process for execution. Separate ready queues are maintained for each priority level. FIFO way is used in each priority level.

4.3 Aging

To prevent starvation, an aging mechanism is implemented. Processes that remain in the ready state for extended periods gradually receive higher priority.

4.4 Scheduling Algorithm

The scheduler performs the following steps:

1. Applies aging to ready processes
2. Selects the highest-priority ready process
3. Updates process states
4. Performs a context switch

5 Context Switching

Context switching is implemented using x86 assembly. The `ctx_switch` routine saves the current process's CPU context, restores the next process's context, and transfers execution accordingly.

This mechanism enables multiple processes to share the CPU while maintaining independent execution states.

6 Inter-Process Communication

kacchiOS supports basic IPC through message passing.

6.1 Send

The `send` function delivers a message to a target process. If the receiver is blocked, it is immediately woken up.

6.2 Receive

The `receive` function blocks the calling process until a message is available, enabling synchronous communication between processes.

7 Testing and Validation

The operating system was tested using QEMU with serial output. Test cases included:

- Memory allocation and deallocation tests
- Multiple process creation and termination
- Cooperative multitasking behavior
- Priority scheduling and aging demonstration
- Context switch validation
- IPC blocking and wakeup behavior

All components operated correctly under the tested scenarios.

8 Conclusion

This project successfully demonstrates the implementation of essential operating system components in kacchiOS. Through this project, core concepts such as memory management, process scheduling, context switching, and inter-process communication were explored in a practical setting.

Future enhancements may include preemptive scheduling using timer interrupts, virtual memory support, and memory protection mechanisms.