**DESIGN PROJECT REPORT**

**EEX5335**

**OPERATING SYSTEMS**

**MOBILE OPERATING SYSTEM**

BY

M.N.M. FAZEEL

220260734

SUBMITED TO

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

FACULTY OF ENGINEERING TECHNOLOGY

THE OPEN UNIVERSITY OF SRI LANKA

AT

COLOMBO

ON

18/12/2024

**Table of Contents**

[Introduction 3](#_Toc185448015)

[Assumptions 4](#_Toc185448016)

[Operating System Architecture 5](#_Toc185448017)

[System Architecture Diagram 5](#_Toc185448018)

[Core Components 5](#_Toc185448019)

[Implementation 6](#_Toc185448020)

[Libraries Used 6](#_Toc185448021)

[Custom Data Structures and Enumeration 7](#_Toc185448022)

[Global Kernal Instance 8](#_Toc185448023)

[Time Function 8](#_Toc185448024)

[OS Power Management 8](#_Toc185448025)

[Sensor Management 9](#_Toc185448026)

[Process Creation 9](#_Toc185448027)

[Security Token Generation 10](#_Toc185448028)

[Memory management 10](#_Toc185448029)

[Simulation 11](#_Toc185448030)

[Process Creation 11](#_Toc185448031)

[Simulating Sensors 11](#_Toc185448032)

[Simulating Process Scheduler 12](#_Toc185448033)

[Simulating Power State Transition 12](#_Toc185448034)

[Kernal Initialization 12](#_Toc185448035)

[Discussion 13](#_Toc185448036)

# **Introduction**

**Overview**:  
This project presents the design and implementation of a lightweight Mobile Operating System (MobileOS Kernel) designed for handheld devices like smartphones. The operating system prioritizes efficient process management, adaptive power saving, and scalability for different hardware.

**Objective**:  
To develop a kernel that demonstrates necessary features like process scheduling, sensor management, memory allocation, and power management.

# **Assumptions**

**Device**:  
The operating system is designed for a handheld device with the following hardware:

* **Processor**: ARM-based CPU
* **Memory**: 256 MB RAM
* **Storage**: 2 GB Flash Memory
* **Sensors**: Accelerometer, Light Sensor, GPS

**Usage Scenario**:  
The OS targets lightweight mobile devices for real-time applications, such as navigation, camera, and background task management.

**Requirements**:

* Support for multitasking and concurrency.
* Adaptive power modes to extend battery life.
* Security and token-based authentication.

# **Operating System Architecture**

## **System Architecture Diagram**

* The architecture is layered as follows:
  1. **Hardware Layer**: Manages device-specific drivers.
  2. **Kernel Layer**: Implements process scheduling, memory allocation, and sensor management.
  3. **Application Layer**: Hosts user-level applications (e.g., navigation, camera).

## **Core Components**

* **Scheduler**: A simple round-robin scheduler manages processes based on priority.
* **Memory Manager**: Adaptive memory allocation ensures optimal use of limited resources.
* **Sensor Manager**: Manages real-time sensor data acquisition.

# **Implementation**

## **Libraries Used**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <stdint.h>

#include <stdbool.h>

#include <string.h>

* #include<stdio.h>: Provides functions for input/output operations such as printing to the console and reading input
* #include<stdlib.h>: Provides utility functions for memory management, random number generation, and program control.
* malloc(): Allocates dynamic memory for sensor data buffers.
* free(): Deallocates memory
* rand(): Generates random numbers for simulating sensor data.
* #include<time.h>: Provides functions for working with time and date. time() used to retrieve the current Unix timestamp in seconds, in this code.
* #include<stdint.h>: Defines fixed-width integer types for portability and predictability across systems. uint8\_t 8-bit unsigned integers uint32\_t 32-bit unsigned integers.
* #include<string.h>: Provides functions for manipulating C strings and memory buffers. strncpy() used to copy process names safely to avoid buffer overflows. memset() used to Initializes memory structures.

## **Custom Data Structures and Enumeration**

*// Security Permissions*

typedef enum {

    PERM\_LOCATION,

    PERM\_CAMERA,

    PERM\_MICROPHONE,

    PERM\_STORAGE,

    PERM\_NETWORK,

    PERM\_CONTACTS,

    PERM\_SENSORS,

    PERM\_BACKGROUND\_PROCESS

} AppPermission;

*// Sensor Data Structure*

typedef struct {

    SensorType type;

    bool is\_active;

    void\* data\_buffer;

    uint16\_t sampling\_rate;

} SensorConfig;

* **Enumeration** **(enum{}):** User-defined data type that consists of a set of named integer constants. It is used to define variables that can only take one of the predefined values.
* **C structure** **(struct{}):** User-defined data type that groups together multiple related variables of different types into a single unit.

## **Global Kernal Instance**

*// Global Kernel Instance*

static MobileOSKernel mobile\_kernel;

* Used to define a static global variable of type MobileOSKernel within the scope of the source file where it's declared. By initiating MobileOSKernel like this it offers some benefits:
* **Encapsulation:** Keeps the implementation details hidden from other modules.
* **Optimization:** The system doesn’t need to repeatedly allocate new memory for the kernel state thus Reduces memory overhead.
* **Simplified Interface:** Allows for a single point of access and control over the mobile operating system's state.

## **Time Function**

uint32\_t system\_time() {

    return (uint32\_t)time(NULL); *// Use Unix timestamp in seconds*

}

* It retrieves the current system time and converts it into a Unix timestamp (number of seconds since the Unix epoch since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC)).

## **OS Power Management**

*// Power Management*

void power\_management(PowerManagementState new\_state) {

}

* This function manages the power state of the mobile operating system by modifying the kernel's power mode and adjusting the activity of processes and sensors according to the new state.
* Function takes in PowerManagementState as a parameter and adjust power states of the OS according to the power states. Defined power states are;
* POWER\_FULL: System is operating at full capacity with all resources available and active.
* POWER\_BATTERY\_SAVE: System reduces the activity of background processes and lowers the sampling rates of active sensors.
* POWER\_ULTRA\_BATTERY\_SAVE: System aggressively reduces power consumption by suspending non-critical processes and disabling most sensors and if priority of a process is lower than 2 it is set to the POWER\_SUSPEND state.

## **Sensor Management**

*// Sensor Management*

bool register\_sensor(SensorType type, uint16\_t sampling\_rate) {

}

* This function is responsible for registering a new sensor into the system's kernel. It assigns an available slot for the new sensor from a pool of predefined sensor slots if a free slot is found.
* Function takes in SensorType, sampling\_rate as parameters and perform bellow functions;
* If an inactive slot is found, the function fills it with the provided sensor type and sets its status to active (is\_active = true).
* It also sets the desired sampling\_rate for the sensor.
* A data buffer is allocated dynamically for the sensor using malloc(1024), which can be used to store the sensor data.

## **Process Creation**

*// Process Creation with Permissions*

uint32\_t create\_process(

}

* function is designed to create a new process in the system's kernel. It assigns an available process slot, initializes the process attributes, and assigns required permissions to the process.
* Parameters of the function:
* process\_name: Name of the process to be created.
* Priority: Priority level of the process (0 for low priority, 255 for high priority).
* required\_permissions: Array of permissions required by the process.
* permission\_count: Number of permissions in the required\_permissions array.
* Function loops through all available process slots (MAX\_PROCESSES) and checks if the process ID (pid) of the current slot is 0, indicating that the slot is available for a new process.
* If an available slot is found, the function:
  + Sets the process ID (pid) to the next available ID.
  + Copies the process name to the process structure.
  + Assigns the defined priority to the process.
  + Sets the last\_active\_timestamp to the current system time using the system\_time()function.
* Then it iterates over the provided required\_permissions array and sets the corresponding permissions in the priority array of the process structure to true.
* Function returns the process ID (pid) of the newly created process if creation is successful. Otherwise returns 0.

## **Security Token Generation**

*// Security Token Generation*

void generate\_security\_token() {

}

* function is designed to generate a cryptographic security token for the system. This function use rand() function to generate random integers but in real life we should generate random values using a cryptographic function.
* The function uses rand() to generate random values for each byte in the token array of mobile\_kernel.system\_token.
* SECURITY\_TOKEN\_LENGTH defines the size of the token, and the loop iterates over the range of this length to fill each byte with a random value.
* rand() % 256 ensures that the random values fall within the range of a byte (0 to 255).
* creation\_time: Sets the creation\_time attribute of the security token to the current system time using the system\_time() function.
* is\_valid: Marks the token as valid (true). This is used to indicate that the generated token is currently active and can be used for authentication.

## **Memory management**

*// Memory Management with Adaptive Allocation*

uint32\_t adaptive\_memory\_allocation(uint32\_t requested\_size) {

}

* This function is designed for managing memory allocation in the mobile operating system kernel. The function is work as follows;
  + The function first checks if the mobile\_kernel.available\_memory is sufficient to meet the requested\_size. If there’s enough available memory, it allocates the memory using malloc() and returns the pointer to the allocated memory.
  + If the available memory is insufficient, the function then attempts to free memory from low-priority processes that are in the POWER\_SUSPEND state.

# **Simulation**

## **Process Creation**

*// Creating multiple processes*

void create\_multiple\_processes() {

}

* This function creates multiple processes in the mobile OS kernel by utilizing the create\_process() function and then prints out the details of the created processes.
* Function defines arrays of permissions (AppPermission) that the processes require.
* Calls the create\_process() function for each process, passing the process name, priority, permissions, and the count of those permissions.
* Iterates through all possible process slots (MAX\_PROCESSES) in the mobile\_kernel.processes[] array. For each active process (pid != 0), it prints: pid, process\_name, priority.

## **Simulating Sensors**

*// Simulate sensor activity*

void simulate\_sensor\_activity() {

}

* Function mimics the behavior of sensors in the system by generating random data and storing it in each active sensor's data buffer. It then prints the simulated data to the console.
* Loops through all sensor slots in the mobile\_kernel.sensors[] array (MAX\_SENSORS).
* Checks if a sensor is active (is\_active is TRUE).
* Cast the data\_buffer pointer to an int\* since it is used to store integer values.
* Generate 10 random integer readings (values between 0 and 99) using the rand() function.
* Store these readings in the data\_buffer.
* Prints the type of the sensor (mobile\_kernel.sensors[i].type) and the simulated readings for that sensor.

## **Simulating Process Scheduler**

*// Simulating process sheduler*

void simulate\_scheduler() {

}

* This Function is a basic implementation of a process scheduler simulation. It iterates through the list of processes in the mobile\_kernel and simulates “running” each process by performing the following tasks:
* Iterates through all process slots (MAX\_PROCESSES) in the mobile\_kernel.processes[] array. For each active process (pid != 0).
* For each active process it prints: pid, process\_name, priority.
* Then it updates the last\_active\_timestamp field to the current time using the system\_time() function, which simulates the process being “run”.

## **Simulating Power State Transition**

*// Simulate power state transitions*

void test\_power\_state\_transitions() {

}

* This function is designed to simulate and verify the behavior of the power management system in the MobileOSKernel and it transitions. It simulates the system through different power states (POWER\_BATTERY\_SAVE and POWER\_ULTRA\_BATTERY\_SAVE) and checks how the system responds, focusing on updating sensor sampling rates in POWER\_BATTERY\_SAVE mode and suspending non-critical processes in POWER\_ULTRA\_BATTERY\_SAVE mode.
* Initially this function outputs the current state of the system to the terminal.
* Then by calling power\_management() function it changes the power state of the system and iterates through MAX\_SENSORS and MAX\_PROCESSES arrays prints the response of the systems to each power transitions.

## **Kernal Initialization**

*// Kernel Initialization*

void initialize\_mobile\_os() {

}

* The function initializes the **kernel state** and prepares the MobileOSKernel structure for operation. It sets up the initial system configuration, including memory management, power state, and security settings.
* Initially using memset() function set the entire MobileOSKernelblock to zero.
* Then it sets the power mode of the memory to POWER\_FULL.
* Finally, it specifies the total memory available in the system (256 MB in this example) and generates a security token.

# **Discussion**

In this project I have implemented a simple Kernal for a Mobile Operating System using C programming language. The complete code and instructions to run the code are uploaded to the GitHub.

The main challenge is to identify what the conditions are that I must fulfill and consider defining a Mobile Operating System. For that I have identified that to be a mobile operating system OS must critically balance the limited resources that are present in the small handheld device and manage the limited power capacity provided by the battery pack to get a long-lasting battery life. I have added some power states to this demonstration which I have inspired from my mobile phone, and I have added some optimizations to compensate for them.

Mobile SoCs are mostly ARM based so for this demonstration I considered a single core mobile SoC for simplicity, so I don’t need to consider multi-tasking and parallel processing much. To manage the processes of the system I have implemented a priority-based preemptive process scheduler which can handle up to 128 processes. Added maximum process count because I had to consider the mobile device limited power so, but with that limited power we can’t handle and allow many processes at once. That also can terminate processes if the power level is critical in order to save battery life.

For the security of the system, I have added process permissions to access security critical parts of the system like accessing connected sensors. Apart from that I have implemented a security token-based security mechanism but in the simulation I have only generated the token but using it to manage sessions.

Then finally, I have added some simulations to simulate the functions of this Kernal by providing required constraints and parameters to the system. I have also implemented a GUI by adding some buttons to call the simulate functions. That code is also in the same repository.

GitHub Repository: <https://github.com/FazeelNizam/MobileOSKernel>