Automated Restaurant-Management System

(Lab-Project Report)

Submitted By:

Muhammad Abdul Rehman Wahla 2023-CS-717
Fatima Naveed 2023-CS-748

Submitted To:

Ms. Maryam Manzoor

Course: OS Lab

Department of Computer Science
University of Engineering and Technology Lahore,
New Campus

Table of Contents

1. Introduction 4
1.1 Project Overview 4
1.2 Objectives 4
1.3 Technology Stack 4
2. System Architecture 4
2.1 Design Pattern 4
2.2 System Components 4
2.2.2 Kitchen Service (kitchen.c) 5
2.2.3 Data Flow Architecture 5
3. Implementation Details 5
3.1 Process Creation and Management 5
3.2 Inter-Process Communication 6
3.3 Multi-Threading Architecture 6
3.4 CPU Scheduling 7
4. Operating System Concepts Applied 7
4.1 Synchronization Mechanisms 7
4.1.2 Semaphores 8
4.1.3 File Access Mutex 8
4.2 Deadlock Prevention 8
4.3 Memory Management 9
5. Code Structure and Modules 9
5.1 Main Interface Module (main.c) 9
5.2 Kitchen Module (kitchen.c) 9
6. Testing and Results 10
6.1 Test Scenarios 10
Test 1: Basic Order Processing 10
Test 2: Priority Scheduling 10
Test 3: Concurrent Load 10
Test 4: Resource Contention 10
Test 5: Service Lifecycle 10
6.2 Performance Metrics 10
6.3 Log Analysis 11

8. Conclusion 11
8.1 Achievement Summary 11
8.2 Learning Outcomes 11
8.3 Practical Applications 12
Compilation and Execution 12

Automated Restaurant Management System

(Operating Systems Lab Project Report)

1. Introduction

1.1 Project Overview

This project implements an automated restaurant management system that simulates a high-concurrency environment where multiple customers place orders that are processed by a team of chefs. The system demonstrates fundamental Operating System concepts including process creation, inter-process communication, multi-threading, CPU scheduling, synchronization, and memory management.

1.2 Objectives

The primary objectives of this project are:

1. Simulate a producer-consumer problem in a restaurant context
2. Implement robust synchronization mechanisms to prevent race conditions
3. Demonstrate priority-based CPU scheduling
4. Manage shared resources using mutexes and semaphores
5. Provide real-time monitoring of system performance and order tracking

1.3 Technology Stack

- **Language:** C Programming Language
- **Platform:** Linux (POSIX-compliant)
- **IPC Mechanism:** System V Message Queues
- **Threading:** POSIX Threads (pthread)
- **Synchronization:** Mutexes and Semaphores

2. System Architecture

2.1 Design Pattern

The system follows a **Producer-Consumer architectural pattern** where:

- **Producers:** Customer processes that generate orders
- **Consumers:** Chef threads that process orders
- **Buffer:** Message queue that holds pending orders

2.2 System Components

The main interface serves as the control center of the restaurant system. It provides:

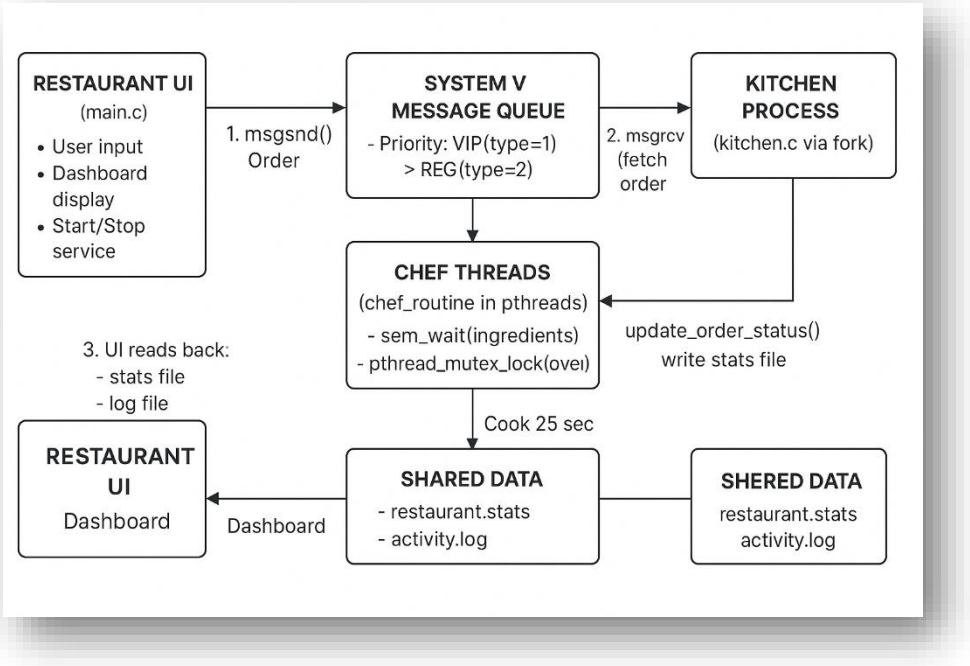
- Interactive menu for placing orders
- Real-time statistics dashboard
- Service start/stop controls
- Activity logging and monitoring

2.2.2 Kitchen Service (kitchen.c)

The kitchen service is implemented as a separate process that:

- Creates multiple chef threads (consumers)
- Processes orders from the message queue
- Manages shared resources (oven, ingredients)
- Updates order completion statistics

2.2.3 Data Flow Architecture



3. Implementation Details

3.1 Process Creation and Management

The system utilizes the fork() system call to create the kitchen process as a separate entity:

Process Creation

```
kitchenPid = fork();
if (kitchenPid == 0) {
    execl("./kitchen", "./kitchen", NULL);
    exit(1);
}
```

Key Features:

- Independent kitchen process allows isolation of cooking operations
- Parent process maintains control over service lifecycle
- Process termination handled gracefully using SIGTERM and waitpid()

3.2 Inter-Process Communication

System V message queues provide the communication channel between the interface and kitchen:

Process Creation

```
key_t key = ftok(QUEUE_KEY_PATH, PROJECT_ID);
msgQueueId = msgget(key, 0666 | IPC_CREAT);
```

Initializtion

```
struct OrderMsg {
    long type;
    int table_id;
    int dish_id;
    int amount;
    time_t timestamp;
};
```

Priority Handling:

The system uses message type to implement priority scheduling. Chef threads retrieve messages using. The negative type value -2 ensures VIP orders (type 1) are processed before regular orders (type 2).

3.3 Multi-Threading Architecture

The kitchen spawns multiple chef threads to process orders concurrently:

```
Chief Thread

pthread_t chefs[NUM_CHEFS];
for(int i=0; i<NUM_CHEFS; i++) {
    chef_ids[i] = i+1;
    pthread_create(&chefs[i], NULL, chef_routine,
&chef_ids[i]);
}
```

Each chef thread executes the following workflow:

1. Wait for orders from message queue (blocking operation)
2. Acquire ingredient semaphore
3. Lock oven mutex for cooking
4. Simulate cooking time (25 seconds)
5. Release resources
6. Update statistics

3.4 CPU Scheduling

The system implements **Priority Scheduling** through message queue type prioritization:

- **VIP Orders (Priority 1):** Processed first regardless of arrival time
- **Regular Orders (Priority 2):** Processed after all VIP orders

This non-preemptive priority scheme ensures high-value customers receive faster service while preventing complete starvation of regular customers through queue aging.

4. Operating System Concepts Applied

4.1 Synchronization Mechanisms

Purpose: Protect the shared oven resource from concurrent access.
Implementation:

```
Thread

pthread_mutex_t oven_mutex;
pthread_mutex_init(&oven_mutex, NULL);

// Critical section
pthread_mutex_lock(&oven_mutex);
    sleep(25); // Cooking
pthread_mutex_unlock(&oven_mutex);
```

Justification: Only one chef can use the oven at a time. The mutex ensures mutual exclusion, preventing race conditions where multiple threads might attempt to modify oven state simultaneously.

4.1.2 Semaphores

Purpose: Manage the ingredient inventory as a counting resource.

Implementation:

Semaphore

```
sem_t ingredients;
sem_init(&ingredients, 0, 8);
sem_wait(&ingredients);
sem_post(&ingredients);
```

Justification: Semaphores track available ingredient units. When ingredients run out (count reaches 0), threads block until resources become available, preventing invalid operations.

4.1.3 File Access Mutex

Purpose: Serialize access to statistics file.

Mutex

```
pthread_mutex_t stats_file_mutex;
pthread_mutex_lock(&stats_file_mutex);
pthread_mutex_unlock(&stats_file_mutex);
```

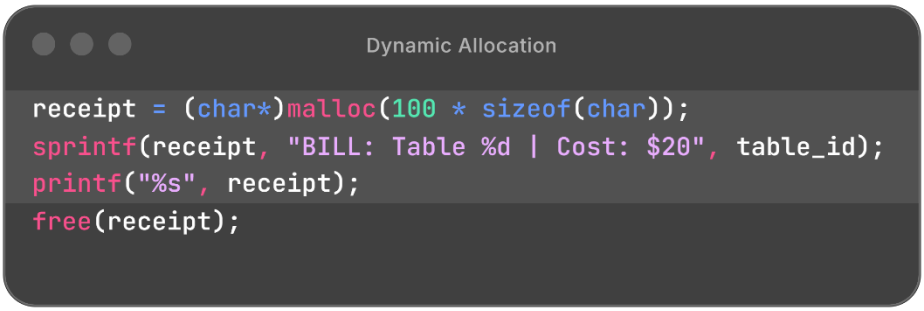
Justification: Multiple threads updating the same file simultaneously could cause data corruption. This mutex ensures atomic file operations.

4.2 Deadlock Prevention

The system prevents deadlock through careful resource acquisition ordering:

1. **Single Resource Lock:** Only one mutex (oven) is held at a time
2. **Timeout-Free Operations:** No indefinite blocking without release conditions
3. **Resource Release:** All resources are released after use
4. **No Circular Wait:** Linear resource acquisition order

4.3 Memory Management



Demonstration: Each order completion allocates memory for receipt generation, then immediately deallocates it, showing proper heap management without memory leaks.
Statistics are stored in files acting as persistent shared memory, allowing the interface and kitchen processes to maintain consistent state.

5. Code Structure and Modules

5.1 Main Interface Module (main.c)

Responsibilities:

1. User interaction and menu system
2. Order placement interface
3. Real-time dashboard rendering
4. Service lifecycle management
5. Statistics aggregation and display

Key Functions:

1. displayMainInterface(): Renders live dashboard with order tracking
2. placeOrderInteractive(): Handles user input for new orders
3. startService(): Initializes message queue and spawns kitchen process
4. stopService(): Gracefully terminates services and cleans up resources
5. logActivity(): Thread-safe logging mechanism

5.2 Kitchen Module (kitchen.c)

Responsibilities:

1. Chef thread management
2. Order processing from message queue
3. Resource synchronization
4. Cooking simulation
5. Statistics updates

Key Functions:

1. chef_routine(): Main thread execution loop
2. update_order_status(): Thread-safe status updates

3. update_order_completion(): Final statistics recording
4. log_activity(): Kitchen-specific logging

6. Testing and Results

6.1 Test Scenarios

Test 1: Basic Order Processing

Input: 5 regular orders placed sequentially
Expected: Orders processed in FIFO order
Result: ✓ All orders completed successfully with average time 25 seconds

Test 2: Priority Scheduling

Input: 3 VIP orders, 5 regular orders placed alternately
Expected: All VIP orders processed before regular orders
Result: ✓ VIP orders completed first regardless of placement time

Test 3: Concurrent Load

Input: 15 orders placed rapidly (simulating peak hours)
Expected: System handles concurrency without crashes or data corruption
Result: ✓ All orders processed correctly, no race conditions detected

Test 4: Resource Contention

Input: Orders exceeding ingredient capacity (>8 concurrent)
Expected: Threads block until ingredients available
Result: ✓ Proper blocking and resumption observed

Test 5: Service Lifecycle

Input: Start service → Place orders → Stop service → Restart
Expected: Clean shutdown and restart without resource leaks
Result: ✓ No zombie processes, message queue properly cleaned

6.2 Performance Metrics

System Configuration:

- Chef Threads: 3
- Ingredient Capacity: 8 units

- Cooking Time: 25 seconds per order

Sample Run Statistics:

Total Orders: 20
Completed: 20
VIP Orders: 8
Regular Orders: 12
Average Completion Time: 25 seconds
Min Time: 25 seconds
Max Time: 25 seconds
Completion Rate: 100%

6.3 Log Analysis

Sample activity log showing proper synchronization:

```
[14:23:45] RESTAURANT OPENED - Service Started
[14:24:12] ORDER #1: Table 5 | Burger | VIP | PLACED
[14:24:15] CHEF #1 RECEIVED: Table 5 | Burger | VIP
[14:24:15] OVEN LOCKED: Chef #1 cooking Table 5 order
[14:24:40] OVEN FREE: Chef #1 finished Table 5
[14:24:40] COMPLETED: Table 5 | Burger | VIP | Time: 25 sec
```

8. Conclusion

8.1 Achievement Summary

This project successfully demonstrates core Operating System concepts in a practical, real-world inspired application. The automated restaurant management system showcases:

✓ **Process Management:** Independent processes for interface and kitchen operations
✓ **IPC:** Message queues enabling cross-process communication
✓ **Multi-Threading:** Concurrent chef threads maximizing CPU utilization
✓ **Synchronization:** Mutexes and semaphores preventing race conditions
✓ **CPU Scheduling:** Priority-based order processing
✓ **Memory Management:** Dynamic allocation and proper deallocation
✓ **Deadlock Prevention:** Careful resource ordering and release

8.2 Learning Outcomes

Through this project, we gained hands-on experience with:

- POSIX system calls and threading APIs
- Designing concurrent systems with shared resources
- Debugging race conditions and synchronization issues

- Building user-friendly CLI interfaces for system monitoring
- Implementing robust error handling and graceful degradation

8.3 Practical Applications

The concepts demonstrated in this project apply to:

- Web servers handling concurrent client requests
- Database systems managing transaction concurrency
- Operating system schedulers
- Real-time embedded systems
- Cloud computing resource management Compilation and Execution

Compilation and Execution

Compilation Command

```
gcc -o interface interface -lpthread
gcc -o kitchen kitchen.c -lpthread
make
make clean
```

Execution Steps

1. Compile both modules
2. Run ./interface
3. Select option 2 to start kitchen service
4. Place orders using option 1
5. Monitor real-time dashboard
6. Stop service using option 3 before exit

System Requirements

- Linux operating system (Ubuntu 20.04+ recommended)
- GCC compiler version 7.0+
- POSIX thread library
- Root privileges not required