EAST WEST UNIVERSITY

Summer 2025
CSE 325: Operating System

Section: 07

Group no: 07

Project Report: IELTS and GRE exam Problem

Submitted By-

Name	Student Id
Md. Rubaiyat Hasan	2023-3-60-273
Hafsa Ferdousi	2023-3-60-321
Nafisa Naower	2023-3-60-326
Inteshar Alam	2023-3-60-466

Submitted To-

Shatabdi Roy Moon

Lecturer

Department of Computer Science & Engineering

East West University

Submission Date

3rd September, 2025

Introduction:

This project demonstrates the practical application of core operating system principles to solve a complex real-world problem: the automated management of a large-scale standardized examination. We present a simulation system, implemented in C, that leverages processes, threads, semaphores, and mutex locks to manage the logistics of a mock IELTS and GRE exam for 300 students. The primary aim is to develop a robust program that automates student registration, room allocation, and attendance tracking while enforcing critical exam rules to ensure fairness and prevent disruptions.

The administration of such tests poses significant logistical challenges for academic institutions. Success hinges on the careful coordination of resources and strict compliance with protocols. Key challenges include the systematic distribution of hundreds of students across limited rooms without exceeding capacity, enforcing simultaneous start and end times, and accurately recording attendance—all within a secure and uninterrupted testing environment.

Conventional manual approaches to these tasks are often prone to human error, operational inefficiencies, and poor scalability. These shortcomings highlight the critical need for an automated, software-based solution capable of handling this complexity with precision, reliability, and speed. By integrating fundamental OS design principles, our system effectively models and controls the simultaneous actions of many entities (students) competing for shared resources (rooms), thereby mirroring the dynamics of a crowded exam hall.

The simulation relies on key operating system mechanisms:

· Processes to manage high-level administrative tasks, such as the initial allocation of students to rooms while strictly enforcing capacity limits.

· Threads to simulate the concurrent behavior of individual students, mimicking real-life actions like entering a room or finishing the exam.

· Semaphores to serve as synchronization gates, holding all student threads until a central start signal is given, ensuring a fair and coordinated beginning for all participants.

· Mutex Locks to protect critical operations—such as updating attendance records and regulating exits—guaranteeing that no student leaves before the official three-hour session concludes and preventing race conditions.

In essence, this project illustrates how synchronization techniques can coordinate concurrent processes, manage shared resources, and enforce strict timing constraints. The simulation concludes by generating a detailed attendance summary, providing a verifiable record of the examination and validating the robustness of the system's design against real-world concurrency challenges.

Projects Aims And Objectives:

1. Assigning each student a unique ID

2. Assign the student to a room

3. Ensure no overcapacity

4. One set per student

5. Enter only after the exam starts

6. No leaving before the exam ends

7. Attendance summary

Methodology:

This project simulates the execution of a large-scale examination by leveraging core operating system concurrency concepts. The design models real-world entities—students, rooms, and an exam controller—as software processes and threads, synchronized through semaphores, mutex locks, and condition variables to ensure correctness and avoid race conditions. The methodology is executed in the following sequential stages:

1. **Process-Based Student Allocation and Room Assignment**

· The parent process initiates the simulation by forking a dedicated child process via fork().

· This child process is solely responsible for allocating all 300 registered students (TOTAL_STUDENTS) into examination rooms with a fixed capacity of 30 (ROOM_CAPACITY) .

· Each student is assigned a unique identifier and systematically mapped to a specific room, guaranteeing no room exceeds its capacity and no student is duplicated or omitted.

· Upon completing the allocation, the child process terminates, and the parent process resumes control to manage the subsequent exam execution phase.

2. **Synchronized Exam Entry Using Semaphores**

· A global semaphore, exam_start, is initialized to zero. This creates a barrier, holding all student threads in a waiting state until the official exam begins.

· A controller thread releases this barrier at the designated start time by executing a post operation on the semaphore for each waiting student. This mechanism ensures a controlled, simultaneous entry for all participants, accurately simulating the start of a real exam.

3. **Fine-Grained Room Capacity Enforcement**

· Each examination room is protected by its own semaphore, initialized to the room's maximum capacity (30).

· Before a student thread can "enter" a room, it must successfully decrement (wait on) that room's specific semaphore. This guarantees that the physical constraint of room capacity is never violated, preventing overcrowding.

4. **Thread-Based Modeling of Student IDs**

· Each student is instantiated as a separate POSIX thread (pthread), allowing their actions to be modeled concurrently and independently.

- The lifecycle of a student thread follows a strict sequence:

1. Wait for Exam Start: Blocks on the global exam_start semaphore.
2. Gain Room Entry: Blocks on its assigned room's capacity semaphore.
3. Mark Attendance: Safely increments a shared attendance counter for its room. This operation is protected by a mutex lock to prevent data corruption.
4. Wait for Exam End: Remains in the exam hall until the official duration elapses.
5. Exit: Leaves the room upon receiving the end signal.

5. **Centralized Timing via the Controller Thread**

- A dedicated controller thread acts as the central timing authority for the entire examination.
- Its responsibilities include:
    1. Introducing a programmed delay before signaling the exam start.
    2. Releasing the global `exam_start` semaphore to commence the exam.
    3. Managing a shared countdown for the exam's duration (e.g., simulating a 3-hour exam in a reduced time scale).
    4. Updating a global exam_over flag and broadcasting a condition variable to all waiting student threads once the time has expired.

6. **Fair and Ordered Exam Conclusion**

- To prevent students from leaving prematurely and ensure a fair, simultaneous end, a mutex and condition variable pair is used.
- Student threads wait on this condition variable after finishing their work.

· Only when the controller thread broadcasts the signal that the exam is over are all student threads released simultaneously, ensuring a synchronized and race-condition-free conclusion.

7. **Attendance Verification and Reporting**

· After all student threads have exited, the parent process compiles the data collected during the simulation.

· A comprehensive summary report is generated, detailing:

· A per-room breakdown: the count of students present versus the room's capacity.

· A global summary: the total number of students who attended versus the total number registered.

Conclusion:

This project demonstrates that core operating-system primitives—processes, threads, semaphores, mutexes, and condition variables—can be composed to solve a real-world coordination problem cleanly and reliably. A forked allocator process assigns students to rooms; a controller thread governs exam timing; per-student threads model independent behavior; a start-gate semaphore guarantees a simultaneous commencement; per-room semaphores strictly enforce capacity; and a mutex+condition-variable pair ensures an orderly, race-free end to the exam. Empirically, the final attendance report confirms correctness (no over-capacity, no early start or early exit) and fairness across all rooms. In short, the simulator not only meets its functional goals but also showcases the practical power of synchronization and process

Source of code:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <semaphore.h>
#include <pthread.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <stdbool.h>

#define TOTAL_STUDENTS 300
#define ROOM_CAPACITY  30

typedef struct {
    int room_of_student[TOTAL_STUDENTS];
    int rooms;
} Shared;

static sem_t exam_start;
static sem_t *room_sem = NULL;
static pthread_mutex_t att_mtx = PTHREAD_MUTEX_INITIALIZER;
static int *room_attendance = NULL;
static int total_entered = 0;
```

```
static pthread_mutex_t exam_mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t  exam_over_cv = PTHREAD_COND_INITIALIZER;
static bool exam_over = false;

typedef struct {
    int id;
    int room;
} StudentArg;

typedef struct {
    int num_students;
    int duration_s;
    int start_delay_s;
} ControllerCfg;

// Assign students to rooms
static void allocator_child(Shared *sh, int N, int CAP) {
    int R = (N + CAP - 1) / CAP;
    sh->rooms = R;

    for (int i = 0; i < N; i++) {
        sh->room_of_student[i] = i / CAP;
    }

    for (int r = 1; r <= R; r++) {
```

```
        int start_id = (r - 1) * CAP + 1;
        int end_id   = r * CAP; if (end_id > N) end_id = N;
        printf("Students %d - %d -> Room %d\n", start_id, end_id, r);
    }
    _exit(0);
}

// Each student thread
static void *student_thread(void *vp) {
    StudentArg *arg = (StudentArg*)vp;
    int room = arg->room;
    free(arg);

    sem_wait(&exam_start);
    sem_wait(&room_sem[room]);

    pthread_mutex_lock(&att_mtx);
    room_attendance[room]++;
    total_entered++;
    pthread_mutex_unlock(&att_mtx);

    pthread_mutex_lock(&exam_mtx);
    while (!exam_over) pthread_cond_wait(&exam_over_cv, &exam_mtx);
    pthread_mutex_unlock(&exam_mtx);

    sem_post(&room_sem[room]);
```

```
    return NULL;
}

// Control exam timing
static void *controller_thread(void *vp) {
    ControllerCfg *cfg = (ControllerCfg*)vp;

    sleep(cfg->start_delay_s);
    for (int i = 0; i < cfg->num_students; i++) sem_post(&exam_start);
    printf(">>> Exam STARTED\n");

    sleep(cfg->duration_s);

    pthread_mutex_lock(&exam_mtx);
    exam_over = true;
    pthread_cond_broadcast(&exam_over_cv);
    pthread_mutex_unlock(&exam_mtx);
    printf(">>> Exam ENDED\n");

    return NULL;
}

int main(int argc, char **argv) {
    const int N   = TOTAL_STUDENTS;
    const int CAP = ROOM_CAPACITY;
```

```
int duration_s = 5, start_delay_s = 1;

Shared *sh = mmap(NULL, sizeof(Shared), PROT_READ | PROT_WRITE,
                  MAP_SHARED | MAP_ANONYMOUS, -1, 0);

if (fork() == 0) allocator_child(sh, N, CAP);

int st;
wait(&st);
int rooms = sh->rooms;

sem_init(&exam_start, 0, 0);

room_sem = calloc(rooms, sizeof(sem_t));
room_attendance = calloc(rooms, sizeof(int));

for (int r = 0; r < rooms; r++) sem_init(&room_sem[r], 0, CAP);

pthread_t ctrl_tid;
ControllerCfg cfg = { N, duration_s, start_delay_s };
pthread_create(&ctrl_tid, NULL, controller_thread, &cfg);

pthread_t *tids = malloc(sizeof(pthread_t) * N);
for (int i = 0; i < N; i++) {
    StudentArg *arg = malloc(sizeof(StudentArg));
    arg->id = i;
```

```
        arg->room = sh->room_of_student[i];
        pthread_create(&tids[i], NULL, student_thread, arg);
    }

    for (int i = 0; i < N; i++) pthread_join(tids[i], NULL);
    pthread_join(ctrl_tid, NULL);

    printf("\n=== Attendance Summary ===\n");
    for (int r = 0; r < rooms; r++) {
        printf("Room %2d: %3d / %3d\n", r+1, room_attendance[r], CAP);
    }
    printf("TOTAL: %d / %d\n", total_entered, N);

    for (int r = 0; r < rooms; r++) sem_destroy(&room_sem[r]);
    sem_destroy(&exam_start);

    munmap(sh, sizeof(Shared));
    free(room_sem);
    free(room_attendance);
    free(tids);

    return 0;
}
```

Compilation steps:

In the terminal, write

1. gcc -o project project.c

2. ./project.c

Output:

```
Students 1 - 30 -> Room 1
Students 31 - 60 -> Room 2
Students 61 - 90 -> Room 3
Students 91 - 120 -> Room 4
Students 121 - 150 -> Room 5
Students 151 - 180 -> Room 6
Students 181 - 210 -> Room 7
Students 211 - 240 -> Room 8
Students 241 - 270 -> Room 9
Students 271 - 300 -> Room 10
```

```
Exam starts in 1 seconds
>>> Exam STARTED
>>> Exam ENDED
```

```
=== Attendance Summary ===
Room  1:  30 /  30
Room  2:  30 /  30
Room  3:  30 /  30
Room  4:  30 /  30
Room  5:  30 /  30
Room  6:  30 /  30
Room  7:  30 /  30
Room  8:  30 /  30
Room  9:  30 /  30
Room 10:  30 /  30
TOTAL: 300 / 300
```