ABSTRACT

The Printer Queue Management System is a project that simulates how print jobs are handled by a computer system. In this project, a C program manages the job scheduling and printing process, while a Python Tkinter interface lets users easily submit and monitor print jobs. The goal is to show how print jobs can be organized, processed, and tracked in real time using simple operating system techniques.

When a user submits a print job, they provide details like the number of pages, lines, and characters in the job, as well as a priority level. The system then calculates how long the job should take by using fixed speeds for printing pages, processing characters, and printing lines. If the estimated time exceeds a set deadline, the job is rejected so that only tasks that can be completed on time are allowed. This helps ensure that the printer works efficiently and doesn't get overwhelmed by jobs that take too long.

Overall, the project shows that a printer queue can be effectively managed by combining low-level programming with a user-friendly interface. The system clearly demonstrates operating system concepts like multitasking, scheduling, and synchronization in a real-world scenario. It also lays a solid foundation for future improvements, such as implementing more advanced scheduling methods or adjusting resource allocation when many jobs are in the queue. This project not only provides a practical simulation of a real printing system but also serves as an educational tool for understanding how operating systems manage multiple tasks concurrently.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Overview

The Printer Queue Management System is a project that simulates how a computer handles print jobs in a practical, real-world scenario. This system is designed to mimic the entire printing process—from the moment a user submits a job until it is processed and completed by the printer. It serves as an educational tool to demonstrate core operating system concepts such as multitasking, process synchronization, and deadline scheduling.

The project is divided into two main components: the backend and the frontend. The backend is written in the C programming language. It is responsible for receiving, scheduling, and processing print jobs using multiple threads. Each thread represents a printer that handles one job at a time. To avoid conflicts when multiple threads access shared data, the backend uses synchronization mechanisms like mutex locks and semaphores

On the other side, the frontend is developed using Python with Tkinter. This graphical user interface allows users to input details for their print jobs, such as job ID, the number of pages, lines, and characters involved, as well as the priority of the job. Once the jobs are entered, they are displayed in a list for review. When the user submits the print jobs, the frontend communicates with the backend using interprocess communication techniques. This integration helps in managing the jobs effectively and in providing real-time visual feedback to the user, making the complex process user-friendly and accessible.

Additionally, the system continuously monitors the progress of each job. The backend updates the job status every second, reflecting how many pages, lines, or characters have been processed. If a job takes too long, it is cancelled to maintain system responsiveness. In summary, the Printer Queue Management System effectively bridges the gap between low-level system programming and user-friendly application design. By simulating the scheduling and execution of print jobs, the project demonstrates the practical application of operating system principles in a controlled environment. It lays a solid foundation for further enhancements, such as adding advanced scheduling algorithms or improving the user interface to support more complex operations, making it a robust and versatile tool for both learning and demonstration.

1.2 Objective

The Printer Queue Management System project was conceived with the aim of simulating real-world print job management in an operating system environment. The overall objective is not only to build a functioning simulation but also to highlight important operating system concepts like multitasking, process synchronization, and deadline scheduling. The system brings together a low-level C-based backend that handles the core processing of print jobs, and a higher-level Python Tkinter frontend that provides an intuitive user interface. Through this project, we seek to deliver a practical example of how multiple interacting components can be brought together to form a cohesive system.

Key Objectives

1. Simulate Real-World Printer Operations

Our project calculates the required processing time using fixed speeds for pages, lines, and characters. By comparing these times against a predetermined deadline, the system decides whether a job should be accepted or terminated. This simulation demonstrates the challenges that a real printer controller might face in a busy office environment.

2. Demonstrate Multithreading and Concurrent Processing

The backend of the system is written in C, leveraging the power of POSIX threads to simulate multiple printers working concurrently. A fundamental objective is to show how multithreading can be used to run several processes at the same time without one process interfering with another.

3. Enforce Deadlines and Manage Resource Constraints:

To achieve efficient and low-level control over the pipe operations, the communication logic is Through this mechanism, the system learns to allocate resources efficiently and ensures that high-priority jobs are never bogged down by tasks that exceed their allotted time. In doing so, it shows how resource management can be balanced with performance requirements.

4. Integrate a User-Friendly Interface with Complex Backend Logic:

The project includes the development of a user-friendly GUI that dynamically adapts based on the The objective is to bridge low-level system programming with high-level user interaction. Users can input job details—such as the number of pages, lines, and characters, as well as job type and priority—through a simple graphical interface, and then view real-time status updates as the backend processes the print jobs.

5. Demonstrate Practical OS Concepts:

Through the use of job queues, sorting algorithms, and timed simulations, the system explains complex ideas such as how an operating system prioritizes tasks and manages competing resources in a busy environment**.**

6. Accurate Calculation of Print Job Time:

This involves computing times based on three different factors—pages, characters, and lines—and picking the maximum value to determine if the job can finish within the acceptable deadline. This step ensures that only those jobs that can realistically be completed are added to the queue. By doing so, the system avoids unnecessary processing and ensures fairness in resource allocation.

7. Effective Scheduling Through Priority Sorting:

Another important goal is to implement a scheduling algorithm that orders print jobs based on both their priority and their estimated print time. Older or simpler jobs may be processed quicker, but if two jobs have the same priority, the one with the lesser estimated time should ideally be handled first.

8. Safe Concurrency Management:

Mutex locks and semaphores are deployed exactly for this purpose. Achieving safe concurrency not only prevents race conditions but also avoids potential system crashes or data corruption, thus mimicking the careful resource management found in real operating systems.

9. Real-Time Feedback and Monitoring:

A further critical objective is to provide continuous, real-time feedback during the print job processing. The system should update the user every second on how much of the job is completed, and any job that doesn't meet its deadline is immediately stopped.

10. Modular Design for Future Scalability

Finally, a key objective is to design the system in a modular manner so that it can be extended in the future. The separation of the backend processing from the frontend interface illustrates how a modular approach can help maintain and upgrade systems over time.

1.3 Scope of the Project

The Printer Queue Management System is designed as a simulation of print job management in a computing environment. It models the entire process of receiving, scheduling, processing, and monitoring print jobs as they move through a simulated printer queue. The primary aim of the project is to demonstrate key operating system principles—such as multitasking, synchronization, and deadline enforcement—using both low-level C programming for system operations and a high-level Python interface for user interaction. The scope of this project defines what the system is intended to do, under what conditions it works, and the boundaries within which it operates.

Job Submission and Queuing

At the heart of the system is the `PrintJob` data structure, which encapsulates all relevant parameters for a print task:

- **Job ID:** Unique identifier for the print job.
- **Page Count:** The total number of pages to be printed.
- **Line Count:** Total output lines.
- **Character Count:** Number of characters to be processed.
- **Category and Priority:** User-defined classifications, which influence scheduling decisions.

Jobs are submitted via a user interface or command-line input. The backend reads the input, parses the parameters, and validates that the numerical inputs are within acceptable ranges. Once validated, the system computes an estimated process time using fixed speed constants:

- **Page Speed (pages/sec)**
- **Line Speed (lines/sec)**
- **Character Speed (characters/sec)**

Each of these speeds is utilized to calculate the time required for job completion. The calculated time is compared against a predefined deadline (e.g., 20 seconds). If the estimated processing time exceeds the allotted time, the job is immediately rejected to maintain system responsiveness.

Multi-threading and Synchronization

The backend is implemented using C and leverages POSIX threads for concurrent processing:

- **Job Submission Thread:** Continuously reads and queues validated print jobs.
- **Printer Threads:** Each printer thread waits on the `printer` semaphore to signal job availability. Upon receiving a job, a thread dequeues it from the ready queue and simulates the printing process using interval-based progression (typically updating every second).

Thread synchronization is critical. The system uses:

- **Mutex Locks:** (e.g., `queue_mutex`, `print_mutex`) to protect shared resources (job queues, output streams) against concurrent access. This prevents data races and ensures that only one thread modifies a shared structure at any given instance.
- **Semaphores:** Employed to signal print job availability. For example, the semaphore value increases when a new job is added and a printer thread waits on the semaphore to ensure that it processes tasks in a timely manner.
- **Condition Variables:** Used in job submission to notify the system when a new array of jobs has been queued, ensuring that waiting printer threads continue processing without idle time.

Future Enhancements and Scalability

- **Dynamic Scheduling Algorithms:** Future work can introduce dynamic real-time scheduling, incorporating adaptive priority adjustments based on system load or job complexity.

- **Enhanced Error Handling:** A robust error recovery mechanism can be implemented to allow for job retries or alternative resource allocation strategies when jobs exceed deadlines.

- **Distributed Deployment:** Although currently designed for a single-machine simulation, the architecture can be extended to support networked environments where print jobs are distributed across multiple systems.

- **GUI Enhancements:** Future iterations may add graphical analytics, system performance charts, and interactive feedback mechanisms that provide real-time monitoring of concurrency metrics and thread performance.

CHAPTER 2

PROBLEM DEFINITION

Modern operating systems must manage multiple tasks concurrently while ensuring that critical processes complete within defined time constraints. In environments where numerous print jobs are submitted simultaneously—such as busy offices, educational institutions, and public service centers—it becomes challenging to efficiently schedule these jobs without causing undue delays or resource conflicts. The Printer Queue Management System project addresses this challenge by simulating a real-world printing scenario, in which each print job is processed against strict deadlines while sharing limited printer resources.

In traditional printing environments, print jobs differ in size, complexity, and urgency. Some documents require only a few seconds to print, while others may involve thousands of pages. Without an effective scheduling algorithm, large or low-priority jobs can monopolize printer resources, causing high-priority or smaller tasks to be delayed. This inefficiency not only results in longer wait times but can also lead to system bottlenecks, ultimately reducing productivity. The core problem, therefore, is how to design a system that dynamically batches, prioritizes, and processes print jobs in a way that minimizes waiting time and ensures that high-priority tasks are serviced promptly—all while respecting a strict overall deadline for each job.

The project simulates the processing of print jobs by using predetermined speeds for pages, lines, and characters, thereby estimating the processing time for each job. This estimated time is then compared against a set deadline. If the computed processing time exceeds the deadline, the job is rejected to ensure that the system does not become overloaded with jobs that cannot be completed in time. This approach models real-world scenarios where deadlines are critical and immediate action is required to maintain system efficiency.

Furthermore, integrating a user-friendly front end with a performance-critical backend presents additional challenges. Users require a clear and interactive graphical user interface (GUI) to submit print jobs, monitor their progress, and view system feedback. To bridge the gap between a high-level GUI (developed in Python using Tkinter) and the low-level C-based backend, a reliable inter-process communication mechanism is required. This integration must ensure that user inputs are transmitted correctly to the backend, and that real-time feedback from the job processing is accurately displayed.

CHAPTER 3

SYSTEM REQUIREMENTS

3.1 Hardware Requirements

For optimal performance, the system should be deployed on a standard desktop or laptop computer with the following minimum and recommended specifications:

- **Processor:**
    - **Minimum:** A dual-core processor (e.g., Intel Core i3 or equivalent) is sufficient for basic operations.
- **Memory (RAM):**
    - **Minimum:** 4 GB of RAM is required to run the operating system and handle the thread-based simulation without significant slowdowns.
- **Storage:**
    - **Minimum:** At least 100 MB of free disk space for the executable files, source code, and related libraries.

3.2 Software Requirements

- **Compiler and Development Tools (for the C Backend):**

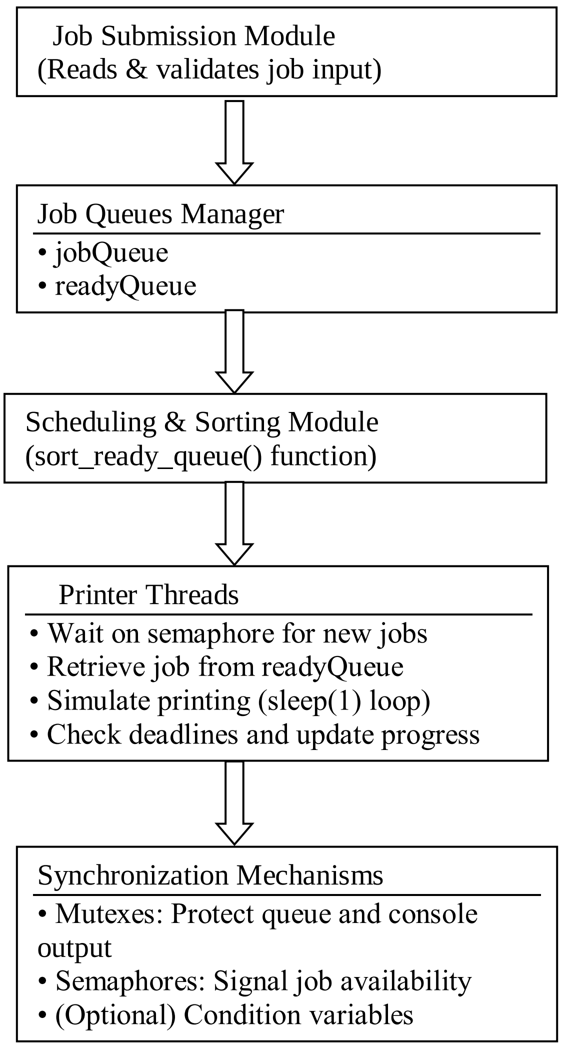
    - **Minimum:** GCC or a compatible C compiler (such as MinGW) that supports POSIX threads, semaphores, and mutex operations.
    - **Recommended:** An integrated development environment (IDE) such as Visual Studio Code, Code::Blocks, or any other capable IDE that supports debugging and profiling to assist in multi-threaded programming.

- **Programming Language Requirements:**

    - **C Language:** The backend is written primarily in C. As such, a properly configured C compilation environment with support for POSIX libraries is required. This includes standard libraries for threading (pthread), synchronization primitives, and standard I/O.
    - **Python Environment:** Python 3.8 or later is required to run the Tkinter-based GUI. Additionally, the Python installation should include the Tkinter module for GUI functionality,

CHAPTER 4

SYSTEM ARCHITECTURE



Job Submission Module
(Reads & validates job input)
Job Queues Manager
• jobQueue
• readyQueue
Scheduling & Sorting Module
(sort_ready_queue() function)
Printer Threads
• Wait on semaphore for new jobs
• Retrieve job from readyQueue
• Simulate printing (sleep(1) loop)
• Check deadlines and update progress
Synchronization Mechanisms
• Mutexes: Protect queue and console output
• Semaphores: Signal job availability
• (Optional) Condition variables

Fig 4.1 Detailed Backend Architecture

Explanation:

- **Job Submission Module:** Handles reading user input, performs validations, and calculates print time estimates.
- **Job Queues Manager:** Uses two arrays—`jobQueue` for all submissions and `readyQueue` for jobs ready to be processed after scheduling.
- **Scheduling & Sorting Module:** Orders jobs based on priority and processing times so that critical tasks are serviced first.
- **Synchronization Mechanisms:** Mutex locks and semaphores ensure safe, concurrent access to shared data, avoiding race conditions and ensuring proper thread coordination.

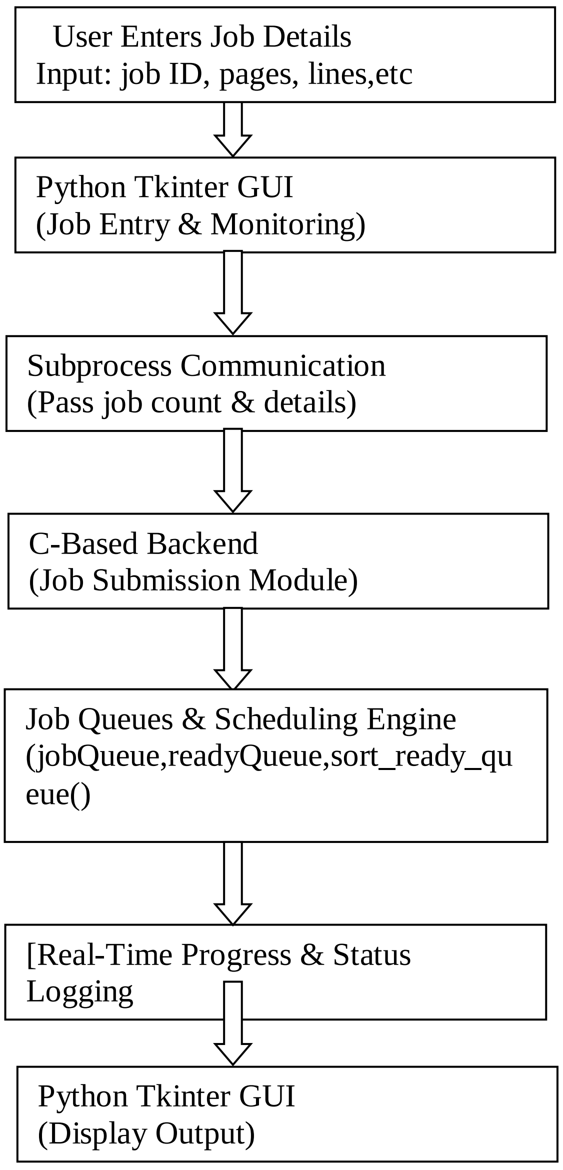

User Enters Job Details
Input: job ID, pages, lines,etc
Python Tkinter GUI
(Job Entry & Monitoring)
Subprocess Communication
(Pass job count & details)
C-Based Backend
(Job Submission Module)
Job Queues & Scheduling Engine
(jobQueue,readyQueue,sort_ready_queue()
[Real-Time Progress & Status Logging
Python Tkinter GUI
(Display Output)

Fig 4.2 End to End Data Flow Diagram

Explanation:

- **Job Entry:** Users input job details via the Tkinter GUI.
- **Data Formatting & Submission:** The frontend formats the data and submits it through the subprocess module to the backend.
- **Job Reception:** The C backend's Job Submission Module reads and validates this data, enqueuing jobs.
- **Queue Management & Scheduling:** Jobs are stored and sorted in queues, ready to be processed.

4.1 Module Description

1. utility.c (Utility Functions Module – Thread-Safe I/O and Job Formatting) Purpose: This module provides core helper functions for the entire backend. It ensures that all console output is performed in a thread-safe manner, converts numeric job parameters into human-readable strings, and formats job details for display. These functions are essential for debugging, proper logging, and consistent output when multiple threads are active.

Key Functions:

1. **safe_print:**
   - Provides a thread-safe method for output by locking around the printf operations.
   - Uses `pthread_mutex_lock` and `pthread_mutex_unlock` to prevent simultaneous writes to the console, ensuring that log messages and progress updates are legible.
2. **getTypeString:**
   - Converts numerical job categories into descriptive strings (e.g., "Newspaper", "Magazine/Book", "Advertisement").
   - Handles special cases, such as when a job ID equals –1 (used as a termination signal).
3. **printJob:**
   - Formats the contents of a `PrintJob` structure into a human-readable string.
   - Consolidates all key fields including job ID, page count, line count, character count, priority, and type into a single output line.
4. **displayQueues:**
   - Iterates through the main job queue and the ready queue, printing their current contents.
   - Locks the queue during read operations to prevent concurrent modifications, ensuring an accurate snapshot of the system's state.

2. job_submission.c (Job Submission Module – Input Handling and Job Enqueuing) Purpose: This module handles all aspects of job submission. It reads user input from the standard input, validates job parameters, calculates the estimated processing time using defined speeds, and enqueues each valid job into the system's job queue and ready queue. It further monitors for jobs that exceed the allowed deadline and signals printer threads when jobs are available.

Key Functions:

1. **job_submission:**

- **Input Handling:** Uses functions like `scanf` to read the number of jobs and each job's parameters (job ID, pages, lines, category, priority, character count).
- **Time Estimation:** Computation of three potential job times:
  - Page processing time: `pages / PRINTER_SPEED`
  - Character processing time: `char_count / CHAR_SPEED`
  - Line processing time: `lines / LINE_SPEED`
  - Selects the maximum of these as the overall estimated processing time.
- **Deadline Check:** If the estimated time exceeds `PRINT_DEADLINE`, the job is rejected and an appropriate message is logged.
- **Job Enqueuing:** Inserts validated jobs into both `jobQueue` and `readyQueue` while protecting these data structures with the `queue_mutex`.
- **Termination Sentinel:** After processing the input, posts termination sentinel jobs (with job ID = –1) to signal the printer threads for proper shutdown.

3. printer.c (Printer Function Module – Simulated Printing and Deadline Enforcement)
Purpose: This module is responsible for simulating the printing process. Printer threads continuously retrieve jobs from the ready queue, simulate the real-time progression of printing, and ensure that each job adheres to the defined deadline. It logs progress at each step and aborts jobs that exceed their allotted processing time.

Key Functions:

1. **printer_function:**
   - **Job Retrieval:** Uses `sem_wait` to block until a job is available, and then locks the `queue_mutex` to safely remove the next job from the `readyQueue`.
   - **Termination Check:** Checks if the retrieved job is a termination signal (job ID equals –1); if so, the thread logs the event and gracefully shuts down.
   - **Job Processing Simulation:**
     - Computes and re-checks the estimated processing time for pages, characters, and lines.
     - Uses a loop with `sleep(1)` to mimic a one-second printing interval, incrementally updating the number of pages, lines, and characters processed.
     - Logs detailed progress messages that include the elapsed time and current job status.

- **Deadline Enforcement:** If the job is not completed within the `PRINT_DEADLINE`, the function logs an abort message, indicating that the job has been terminated prematurely.
- **Output Update:** After each job, the function updates the display of queues and semaphore states using `displayQueues` and `printSemaphoreStates` to facilitate debugging and real-time monitoring.

4. queue_sorting.c (Queue Sorting Module – Priority-Based Job Scheduling) Purpose: This module is critical for ensuring that jobs in the ready queue are processed in an optimal order based on user-specified priority and the estimated processing time. By sorting the ready queue, the system ensures that higher-priority or less resource-intensive jobs are handled first, improving overall system responsiveness.

Key Functions:

1. **sort_ready_queue:**
   - Implements a bubble sort algorithm that iterates over the ready queue and compares adjacent jobs.
   - Compares jobs based on:
     - **Priority:** Jobs with lower numerical priority values (indicating higher priority) are preferred.
     - **Processing Time:** For jobs with equal priority, the one with a lower estimated processing time is given precedence.
   - Excludes termination sentinel jobs from the sorting process to ensure they remain correctly positioned.
   - Swaps jobs if the ordering criteria are not met, resulting in a sorted queue that is used by the printer threads.

5. synchronization.c (Synchronization Module – Managing Concurrent Access) Purpose: This module encapsulates the initialization, use, and cleanup of all synchronization primitives within the backend. It is vital in protecting shared resources such as the job queues and output operations. This module ensures that multiple threads can operate safely without data corruption, race conditions, or deadlocks.

Key Features and Functions:

1. **Mutex Initialization and Usage:**
   - Several mutexes, including `print_mutex`, `queue_mutex`, and `submission_mutex`, are initialized at program startup.

  - These mutexes are used to ensure that when one thread is modifying shared data (e.g., adding or removing jobs from queues or writing to the console), no other thread can interfere until the operation completes.

2. **Semaphore Management:**
   - The `printer` semaphore is created and initialized with a value of 0.
   - This semaphore is used to control access to the printer threads; every time a job is submitted or a termination signal is enqueued, the semaphore is incremented to signal that a job is available.
3. **Condition Variable:**
   - A condition variable (`submission_cv`) is used in conjunction with a mutex (`submission_mutex`) to signal the completion of job submission.
   - This ensures that any waiting threads are notified promptly once all jobs have been submitted.
4. **Resource Cleanup:**
   - Upon program termination, all synchronization primitives are properly destroyed (e.g., using `pthread_mutex_destroy`, `sem_destroy`, and `pthread_cond_destroy`) to prevent resource leaks and maintain system hygiene.

CHAPTER 5

IMPLEMENTATION

# Backendd.c

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdarg.h>
#include <string.h>
#include <math.h>
#include <stdint.h>

#define MAX_JOBS 10
#define NUM_PRINTERS 1

// Speed definitions
#define PRINTER_SPEED 1        // pages per second
#define CHAR_SPEED 3500             // characters per second
#define LINE_SPEED 70             // lines per second

// Deadline (seconds allowed per job)
#define PRINT_DEADLINE 20


typedef struct {
    int job_id;
    int pages;
    int lines;
    int category;
    int priority;
    int char_count;
```

```c
} PrintJob;


PrintJob jobQueue[MAX_JOBS];
int jobSubCount = 0;
PrintJob readyQueue[MAX_JOBS + 10];
int readyCount = 0;

pthread_mutex_t print_mutex;
pthread_mutex_t queue_mutex;
sem_t printer;
pthread_mutex_t submission_mutex;
pthread_cond_t submission_cv;
int submitted = 0;

// ------------------------- Utility Functions -------------------------

void safe_print(const char *format, ...) {
    va_list args;
    pthread_mutex_lock(&print_mutex);
    va_start(args, format);
    vprintf(format, args);
    fflush(stdout);
    va_end(args);
    pthread_mutex_unlock(&print_mutex);
}

const char* getTypeString(int category, int job_id) {
    if (job_id == -1)
        return "TERMINATE";
    static char buf[24];
    switch (category) {
        case 1: snprintf(buf, sizeof(buf), "Newspaper"); break;
        case 2: snprintf(buf, sizeof(buf), "Magazine/Book"); break;
        case 3: snprintf(buf, sizeof(buf), "Advertisement"); break;
        default: snprintf(buf, sizeof(buf), "Unknown");
    }
    return buf;
}

void printJob(PrintJob job, char *dest, size_t destSize) {
    snprintf(dest, destSize, "(ID:%d, Pages:%d, Lines:%d, Chars:%d, Priority:%d,
Type:%s)",
                job.job_id, job.pages, job.lines, job.char_count, job.priority,
getTypeString(job.category, job.job_id));
}

void displayQueues() {
    pthread_mutex_lock(&queue_mutex);
    safe_print("\n==================== Queues ====================\n");
    safe_print("Job Queue: [");
    char buf[128];
    for (int i = 0; i < MAX_JOBS; i++) {
        if (i < jobSubCount) {
            printJob(jobQueue[i], buf, sizeof(buf));
            safe_print("%s", buf);
        } else {
            safe_print("None");
        }
        if (i < MAX_JOBS - 1)
            safe_print(", ");
    }
    safe_print("]\n");

    safe_print("Ready Queue: [");
    for (int i = 0; i < readyCount; i++) {
        printJob(readyQueue[i], buf, sizeof(buf));
        safe_print("%s", buf);
        if (i < readyCount - 1)
            safe_print(", ");
    }
    safe_print("]\n");
    safe_print("===============================================\n");
    pthread_mutex_unlock(&queue_mutex);
}


void* job_submission(void* arg) {
    int num_jobs;
    if (scanf("%d", &num_jobs) != 1) {
        safe_print("Invalid input for number of jobs.\n");
        return NULL;
    }

    for (int i = 0; i < num_jobs; i++) {
        PrintJob newJob;
            if  (scanf("%d  %d  %d  %d  %d  %d",  &newJob.job_id,  &newJob.pages,
&newJob.lines,
                   &newJob.category, &newJob.priority, &newJob.char_count) != 6)
{
            safe_print("Invalid job input.\n");
            return NULL;
        }

        // Calculate estimated printing times.
        float est_page_time = (float)newJob.pages / PRINTER_SPEED;
        float est_char_time = (float)newJob.char_count / CHAR_SPEED;
        float est_line_time = (float)newJob.lines / LINE_SPEED;

        float estimated_time = est_page_time;
        if (est_char_time > estimated_time)
            estimated_time = est_char_time;
        if (est_line_time > estimated_time)
            estimated_time = est_line_time;
        // Check against the allowed deadline.
        if (estimated_time > PRINT_DEADLINE) {
              safe_print("Job ID %d terminated: Estimated time %.2f sec exceeds 
deadline %d sec.\n",
                         newJob.job_id, estimated_time, PRINT_DEADLINE);
            continue;  // Skip this job; do not add it to the queue.
        }

        pthread_mutex_lock(&queue_mutex);
        if (jobSubCount < MAX_JOBS) {
            jobQueue[jobSubCount++] = newJob;
            readyQueue[readyCount++] = newJob;
        } else {
                safe_print("Job Queue is full! Cannot add Job ID: %d\n", 
newJob.job_id);
        }
        pthread_mutex_unlock(&queue_mutex);

        char jobInfo[128];
        printJob(newJob, jobInfo, sizeof(jobInfo));
        safe_print("\nJob Submitted: %s\n", jobInfo);
        displayQueues();
        sem_post(&printer);
    }

    // Post a termination sentinel for each printer thread.
    for (int j = 0; j < NUM_PRINTERS; j++) {
        pthread_mutex_lock(&queue_mutex);
        PrintJob term = { .job_id = -1, .pages = -1, .lines = -1, .priority = -
1, .category = -1, .char_count = -1 };
        readyQueue[readyCount++] = term;
        pthread_mutex_unlock(&queue_mutex);
        sem_post(&printer);
    }
    safe_print("\nSubmitted %d jobs for processing...\n", num_jobs);

    pthread_mutex_lock(&submission_mutex);
    submitted = 1;
    pthread_cond_signal(&submission_cv);
    pthread_mutex_unlock(&submission_mutex);

    return NULL;
}

void printSemaphoreStates() {
    int val;
    sem_getvalue(&printer, &val);
    safe_print("� Semaphore value: %d\n", val);
}

void sort_ready_queue() {
    int limit = readyCount;
    // Exclude termination sentinel if present.
    if (readyCount > 0 && readyQueue[readyCount - 1].job_id == -1)
        limit = readyCount - 1;

    for (int i = 0; i < limit - 1; i++) {
        for (int j = 0; j < limit - i - 1; j++) {
            float est_time1 = (float)readyQueue[j].pages / PRINTER_SPEED;
            float char_time1 = (float)readyQueue[j].char_count / CHAR_SPEED;
            float line_time1 = (float)readyQueue[j].lines / LINE_SPEED;
            if (char_time1 > est_time1) est_time1 = char_time1;
            if (line_time1 > est_time1) est_time1 = line_time1;

            float est_time2 = (float)readyQueue[j+1].pages / PRINTER_SPEED;
            float char_time2 = (float)readyQueue[j+1].char_count / CHAR_SPEED;
            float line_time2 = (float)readyQueue[j+1].lines / LINE_SPEED;
            if (char_time2 > est_time2) est_time2 = char_time2;
            if (line_time2 > est_time2) est_time2 = line_time2;
            if (readyQueue[j].priority > readyQueue[j+1].priority ||
                (readyQueue[j].priority == readyQueue[j+1].priority && est_time1
> est_time2)) {
                PrintJob temp = readyQueue[j];
                readyQueue[j] = readyQueue[j + 1];
                readyQueue[j + 1] = temp;
            }
        }
    }
}

void* printer_function(void* arg) {
    long printer_id = (long)(intptr_t)arg;

    while (1) {
        sem_wait(&printer);

        pthread_mutex_lock(&queue_mutex);
        sort_ready_queue();
        if (readyCount <= 0) {
            pthread_mutex_unlock(&queue_mutex);
            continue;
        }
        PrintJob jobToPrint = readyQueue[0];
        // Shift the ready queue.
        for (int i = 0; i < readyCount - 1; i++)
            readyQueue[i] = readyQueue[i + 1];
        readyCount--;
        pthread_mutex_unlock(&queue_mutex);

        // Check for termination signal.
        if (jobToPrint.job_id == -1) {
             safe_print("\n⍰ [Printer %ld] Received termination signal. Shutting
down.\n", printer_id);
```

```
            break;
        }

        // Calculate the estimated printing times for each factor.
        float est_page_time = (float)jobToPrint.pages / PRINTER_SPEED;
        float est_char_time = (float)jobToPrint.char_count / CHAR_SPEED;
        float est_line_time = (float)jobToPrint.lines / LINE_SPEED;
        float estimated_time = est_page_time;
        if (est_char_time > estimated_time) estimated_time = est_char_time;
        if (est_line_time > estimated_time) estimated_time = est_line_time;

        // Check against the allowed deadline.
        if (estimated_time > PRINT_DEADLINE) {
              safe_print("\n⏱ [Printer %ld] Job ID:%d estimated time %.2f sec
exceeds deadline (%d sec).\n",
                                    printer_id, jobToPrint.job_id, estimated_time,
PRINT_DEADLINE);
            displayQueues();
            printSemaphoreStates();
            continue; // Skip this job.
        }

         safe_print("\n [Printer %ld] Starting Job ID:%d - Pages:%d, Lines:%d,
Chars:%d, Type:%s, Priority:%d\n",
                                 printer_id, jobToPrint.job_id, jobToPrint.pages,
jobToPrint.lines, jobToPrint.char_count,
                             getTypeString(jobToPrint.category, jobToPrint.job_id),
jobToPrint.priority);

        // Simulate the printing process second-by-second.
           int printed_pages = 0, processed_chars = 0, printed_lines = 0,
seconds_elapsed = 0;
while ((printed_pages < jobToPrint.pages ||
        processed_chars < jobToPrint.char_count ||
        printed_lines < jobToPrint.lines) &&
        seconds_elapsed < PRINT_DEADLINE)
{
    sleep(1);
    seconds_elapsed++;

    if (printed_pages < jobToPrint.pages) {
        printed_pages += PRINTER_SPEED;
        if (printed_pages > jobToPrint.pages)
            printed_pages = jobToPrint.pages;
    }
    if (processed_chars < jobToPrint.char_count) {
        processed_chars += CHAR_SPEED;
        if (processed_chars > jobToPrint.char_count)
            processed_chars = jobToPrint.char_count;
    }
    if (printed_lines < jobToPrint.lines) {
        printed_lines += LINE_SPEED;
        if (printed_lines > jobToPrint.lines)
            printed_lines = jobToPrint.lines;
    }

     safe_print("[Printer  %ld]  Job  ID:%d  Progress:  %d/%d  pages,  %d/%d  lines,
%d/%d chars (Elapsed: %d sec)\n",
               printer_id, jobToPrint.job_id, printed_pages, jobToPrint.pages,
                            printed_lines,   jobToPrint.lines,   processed_chars,
jobToPrint.char_count, seconds_elapsed);
}

// If not done within the deadline, terminate the job.
if (printed_pages < jobToPrint.pages ||
    processed_chars < jobToPrint.char_count ||
    printed_lines < jobToPrint.lines  ) {
    safe_print("\n⚠ [Printer %ld] Deadline exceeded! Job ID:%d aborted.\n",
               printer_id, jobToPrint.job_id);
} else {
    safe_print("\n✅ [Printer %ld] Completed Job ID:%d in %d sec.\n",
               printer_id, jobToPrint.job_id, seconds_elapsed);
}
        displayQueues();
        printSemaphoreStates();
    }
    return NULL;
}

int main() {
    pthread_t submission_thread, printer_threads[NUM_PRINTERS];

    // Initialize mutexes, condition variables, and semaphore.
    pthread_mutex_init(&print_mutex, NULL);
    pthread_mutex_init(&queue_mutex, NULL);
    pthread_mutex_init(&submission_mutex, NULL);
    pthread_cond_init(&submission_cv, NULL);
    sem_init(&printer, 0, 0);

    // Create printer threads.
    for (long i = 0; i < NUM_PRINTERS; i++) {
            pthread_create(&printer_threads[i],    NULL,    printer_function,
(void*)(intptr_t)i);

    }
    // Create the job submission thread.
    pthread_create(&submission_thread, NULL, job_submission, NULL);

    // Wait for the submission and printer threads.
    pthread_join(submission_thread, NULL);
    for (int i = 0; i < NUM_PRINTERS; i++) {
        pthread_join(printer_threads[i], NULL);
    }

    // Cleanup.
    sem_destroy(&printer);
    pthread_mutex_destroy(&print_mutex);
    pthread_mutex_destroy(&queue_mutex);
    pthread_mutex_destroy(&submission_mutex);
    pthread_cond_destroy(&submission_cv);

    return 0;
}
```

#frontend.py

```
import tkinter as tk
from tkinter import ttk, messagebox
from tkinter.scrolledtext import ScrolledText
import subprocess
import threading
job_list = []
def add_job():

    job_id = entry_job_id.get().strip()
    pages = entry_pages.get().strip()
    lines = entry_lines.get().strip()
    char_count = entry_char_count.get().strip()
    category = category_var.get().strip()
    priority = priority_var.get().strip()


    if not job_id or not pages or not lines or not char_count or not category or
not priority:
        messagebox.showwarning("Input Error", "All fields must be filled!")
        return


    try:
        job_id_int = int(job_id)
        pages_int = int(pages)
        lines_int = int(lines)
        char_count_int = int(char_count)
    except ValueError:
        messagebox.showerror("Input Error", "Job ID, Pages, Lines, and Character Count must be numbers!")
        return

    try:
        # The combo boxes are expected to start with a number.
        category_num = int(category.split()[0])
        priority_num = int(priority.split()[0])
    except Exception:
        messagebox.showerror("Input Error", "Category and Priority should start with a number.")
        return

    # Create a job tuple with 6 elements.
    job = (job_id_int, pages_int, lines_int, category_num, priority_num, char_count_int)
    job_list.append(job)
    listbox_jobs.insert(tk.END,
              f"ID:{job_id_int} | Pages:{pages_int} | Lines:{lines_int} | Chars:{char_count_int} | Type:{category_num} | Priority:{priority_num}")

    # Clear entries for next input.
    entry_job_id.delete(0, tk.END)
    entry_pages.delete(0, tk.END)
    entry_lines.delete(0, tk.END)
    entry_char_count.delete(0, tk.END)
    category_var.set('')
    priority_var.set('')

def append_output(text):
    def inner():
        output_text.insert(tk.END, text)
        output_text.see(tk.END)
    root.after(0, inner)

def execute_backend(jobs_to_submit):

    if not jobs_to_submit:
        messagebox.showwarning("No Job", "No job to submit!")
        return

    # Prepare input data; first line is the number of jobs.
    input_data = f"{len(jobs_to_submit)}\n"
    for job in jobs_to_submit:
                                                      input_data            +=
f"{job[0]}\n{job[1]}\n{job[2]}\n{job[3]}\n{job[4]}\n{job[5]}\n"

    try:
        process = subprocess.Popen(
            ["printer_queue.exe"],
            stdin=subprocess.PIPE,
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE,
            text=True,
            encoding='utf-8',
            errors='replace'
        )

        def read_output(pipe):
            for line in iter(pipe.readline, ''):
                append_output(line)
            pipe.close()

        # Capture backend output in separate threads.
```

```
                threading.Thread(target=read_output,   args=(process.stdout,),
daemon=True).start()
                threading.Thread(target=read_output,   args=(process.stderr,),
daemon=True).start()

        process.stdin.write(input_data)
        process.stdin.close()
        process.wait()

    except Exception as e:
        root.after(0, lambda: messagebox.showerror("Execution Error", str(e)))
def submit_job():
    """Submit a single selected job from the listbox."""
    selected_indices = listbox_jobs.curselection()
    if not selected_indices:
           messagebox.showwarning("Selection  Error",  "Please  select  a  job  to
submit!")
        return
    index = selected_indices[0]
    job = job_list.pop(index)
    listbox_jobs.delete(index)
    threading.Thread(target=execute_backend, args=([job],), daemon=True).start()
def submit_all_jobs():
    """Submit all jobs currently in the GUI queue."""
    if not job_list:
        messagebox.showwarning("No Jobs", "No jobs to submit!")
        return
    jobs_to_submit = job_list.copy()
    job_list.clear()
    listbox_jobs.delete(0, tk.END)
            threading.Thread(target=execute_backend,    args=(jobs_to_submit,),
daemon=True).start()
#------------------------- Tkinter Layout -------------------------
root = tk.Tk()
root.title("Printer Queue Management")
```

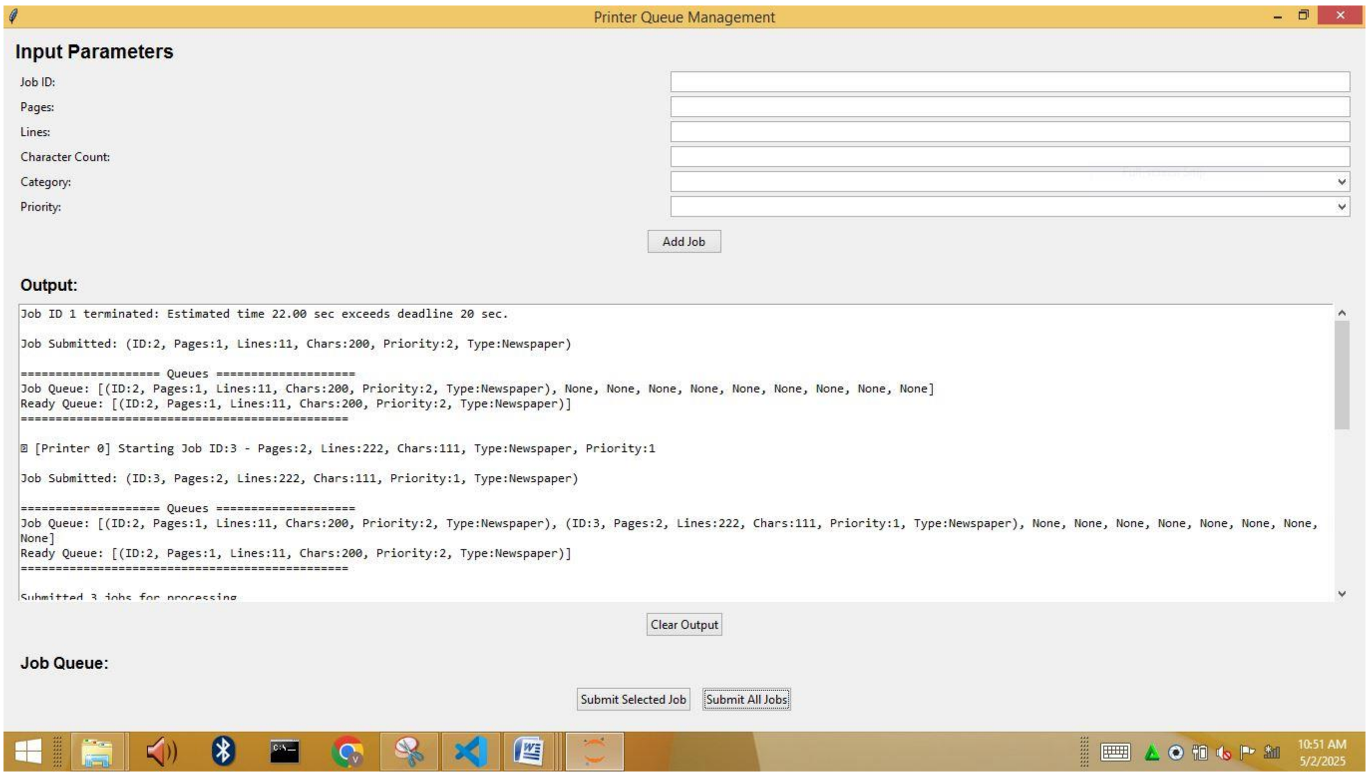
```
try:
    root.state('zoomed')
except Exception:
    root.geometry("1200x800")
main_frame = ttk.Frame(root, padding="10")
main_frame.grid(row=0, column=0, sticky="nsew")
root.columnconfigure(0, weight=1)
root.rowconfigure(0, weight=1)
# Configure two columns.
main_frame.columnconfigure(0, weight=1)
main_frame.columnconfigure(1, weight=1)
ttk.Label(main_frame, text="Input Parameters", font=("Arial", 14, "bold")).grid(row=0, column=0, columnspan=2, sticky="w", pady=(0,5))
ttk.Label(main_frame, text="Job ID:").grid(row=1, column=0, sticky="w", padx=5, pady=2)
entry_job_id = ttk.Entry(main_frame, width=20)
entry_job_id.grid(row=1, column=1, sticky="ew", padx=5, pady=2)
ttk.Label(main_frame, text="Pages:").grid(row=2, column=0, sticky="w", padx=5, pady=2)
entry_pages = ttk.Entry(main_frame, width=20)
entry_pages.grid(row=2, column=1, sticky="ew", padx=5, pady=2)
ttk.Label(main_frame, text="Lines:").grid(row=3, column=0, sticky="w", padx=5, pady=2)
entry_lines = ttk.Entry(main_frame, width=20)
entry_lines.grid(row=3, column=1, sticky="ew", padx=5, pady=2)
ttk.Label(main_frame, text="Character Count:").grid(row=4, column=0, sticky="w", padx=5, pady=2)
entry_char_count = ttk.Entry(main_frame, width=20)
entry_char_count.grid(row=4, column=1, sticky="ew", padx=5, pady=2)
ttk.Label(main_frame, text="Category:").grid(row=5, column=0, sticky="w", padx=5, pady=2)
category_var = ttk.Combobox(main_frame, values=["1 - Newspaper", "2 - Magazine/Book", "3 - Advertisement"], width=18)
category_var.grid(row=5, column=1, sticky="ew", padx=5, pady=2)
```

```
ttk.Label(main_frame, text="Priority:").grid(row=6, column=0, sticky="w", padx=5, pady=2)
priority_var = ttk.Combobox(main_frame, values=["1 - High", "2 - Medium", "3 - Low"], width=18)
priority_var.grid(row=6, column=1, sticky="ew", padx=5, pady=2)
btn_add_job = ttk.Button(main_frame, text="Add Job", command=add_job)
btn_add_job.grid(row=7, column=0, columnspan=2, pady=10)
ttk.Label(main_frame, text="Output:", font=("Arial", 12, "bold")).grid(row=8, column=0, columnspan=2, sticky="w", padx=5, pady=(10,2))
# Increase height of the output text area.
output_text = ScrolledText(main_frame, height=40, font=("Consolas", 10))
output_text.grid(row=9, column=0, columnspan=2, sticky="nsew", padx=5, pady=5)

btn_clear_output = ttk.Button(main_frame, text="Clear Output", command=lambda: output_text.delete(1.0, tk.END))
btn_clear_output.grid(row=10, column=0, columnspan=2, pady=5)


ttk.Label(main_frame, text="Job Queue:", font=("Arial", 12, "bold")).grid(row=11, column=0, columnspan=2, sticky="w", padx=5, pady=(10,2))
frame_listbox = ttk.Frame(main_frame)
frame_listbox.grid(row=12, column=0, columnspan=2, sticky="nsew", padx=5, pady=5)
frame_listbox.columnconfigure(0, weight=1)
frame_listbox.rowconfigure(0, weight=1)
scrollbar_jobs = ttk.Scrollbar(frame_listbox, orient=tk.VERTICAL)
listbox_jobs = tk.Listbox(frame_listbox, height=5, yscrollcommand=scrollbar_jobs.set, font=("Consolas", 10))
scrollbar_jobs.config(command=listbox_jobs.yview)
scrollbar_jobs.grid(row=0, column=1, sticky="ns")
listbox_jobs.grid(row=0, column=0, sticky="nsew")
submit_frame = ttk.Frame(main_frame)
submit_frame.grid(row=13, column=0, columnspan=2, pady=10)
btn_submit_job = ttk.Button(submit_frame, text="Submit Selected Job", command=submit_job)
```

```
btn_submit_job.pack(side=tk.LEFT, padx=5)
btn_submit_all    =    ttk.Button(submit_frame,    text="Submit    All    Jobs",
command=submit_all_jobs)
btn_submit_all.pack(side=tk.LEFT, padx=5)
main_frame.rowconfigure(9, weight=2)
main_frame.rowconfigure(12, weight=1)
root.mainloop()
```

CHAPTER 6

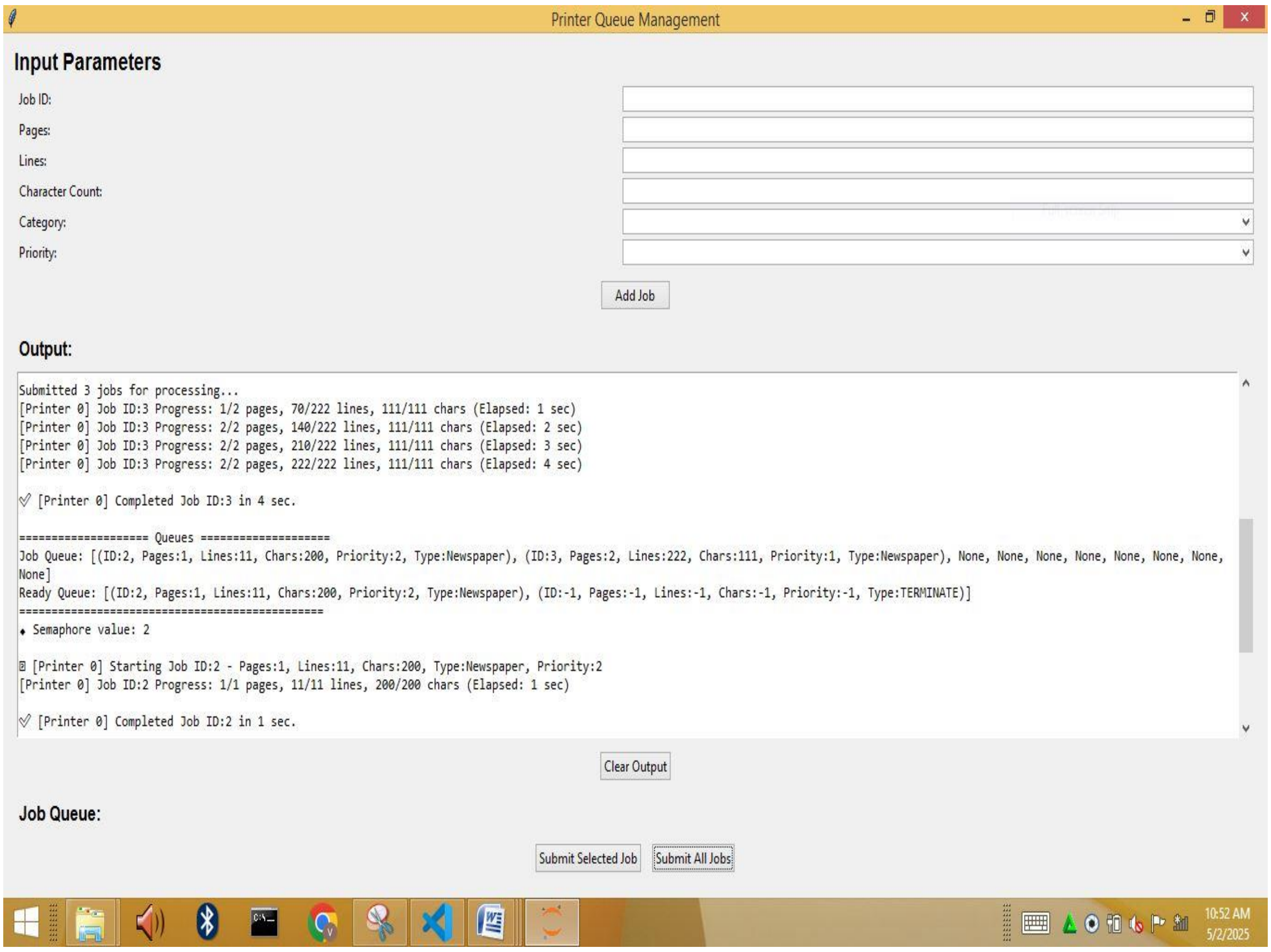
RESULTS



```
Job ID 1 terminated: Estimated time 22.00 sec exceeds deadline 20 sec.

Job Submitted: (ID:2, Pages:1, Lines:11, Chars:200, Priority:2, Type:Newspaper)

=================== Queues ===================
Job Queue: [(ID:2, Pages:1, Lines:11, Chars:200, Priority:2, Type:Newspaper), None, None, None, None, None, None, None, None, None]
Ready Queue: [(ID:2, Pages:1, Lines:11, Chars:200, Priority:2, Type:Newspaper)]
==============================================

▫ [Printer 0] Starting Job ID:3 - Pages:2, Lines:222, Chars:111, Type:Newspaper, Priority:1

Job Submitted: (ID:3, Pages:2, Lines:222, Chars:111, Priority:1, Type:Newspaper)

=================== Queues ===================
Job Queue: [(ID:2, Pages:1, Lines:11, Chars:200, Priority:2, Type:Newspaper), (ID:3, Pages:2, Lines:222, Chars:111, Priority:1, Type:Newspaper), None, None, None, None, None, None, None,
None]
Ready Queue: [(ID:2, Pages:1, Lines:11, Chars:200, Priority:2, Type:Newspaper)]
==============================================

Submitted 3 jobs for processing
```

Fig 6.1 Adding printjobs in the jobqueue

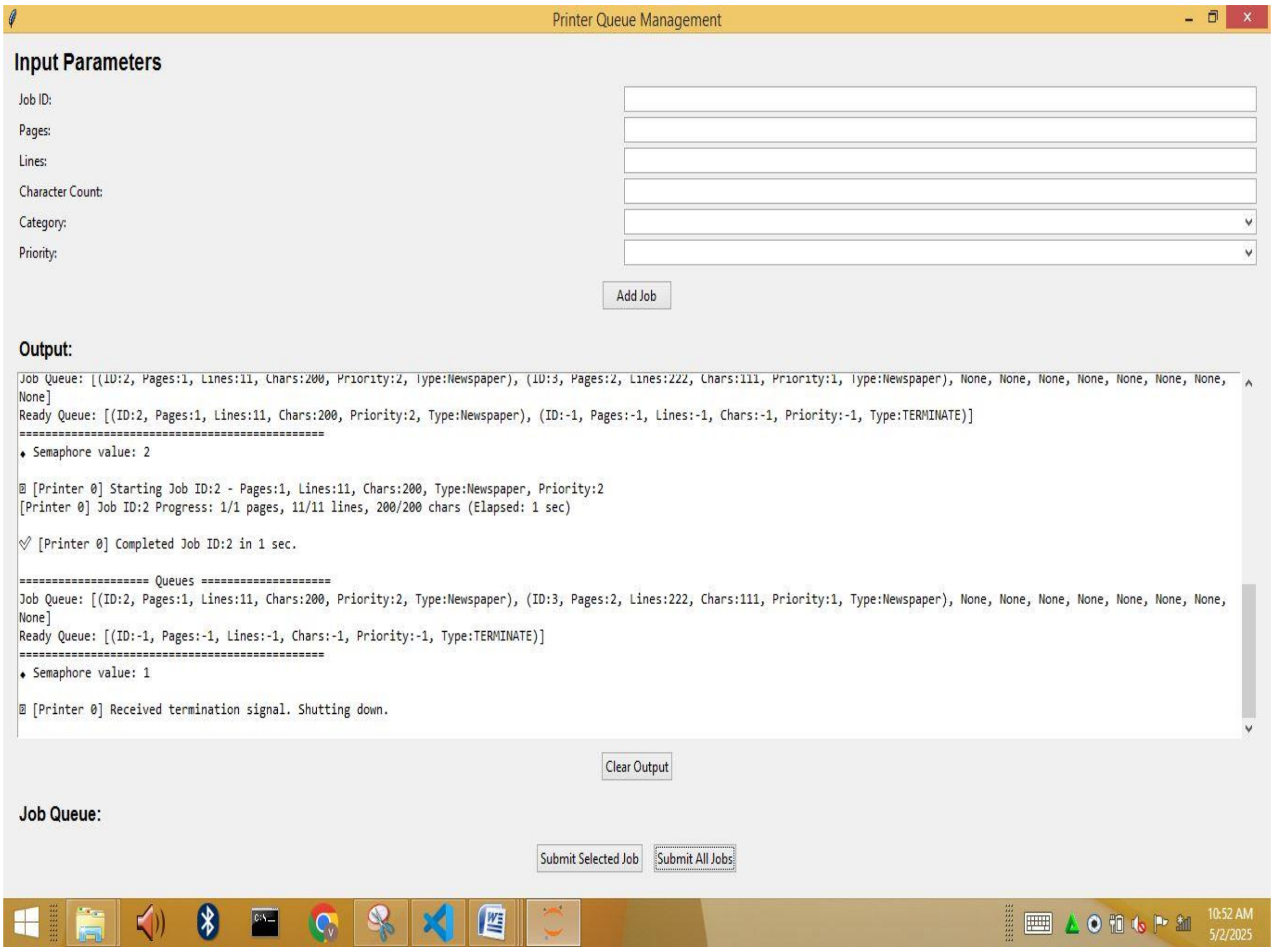
In Figure 6.1 ,the tkinter gui interface allows to enter the print jobs to add the jobs to the job queue the printjobs upon selecting the details of the jobs are added untill the submit all button is clicked .the printjobs are processed after the click of the submit all button .In this scenario the printjobs of pages 22 with priority 2, jobid of 2 with pages 1 of priority 2 , jobid of 3 with pages 2 of priority 1 are added in the job queue. From the above diagram we infer that if a printjob with estimated time(seconds) > deadline(seconds) that printjob is terminated . here the jobid 1 when it is added to job queue it gets terminated because the estimated time is 22.00 sec and deadline is 20.00 sec.



```
Submitted 3 jobs for processing...
[Printer 0] Job ID:3 Progress: 1/2 pages, 70/222 lines, 111/111 chars (Elapsed: 1 sec)
[Printer 0] Job ID:3 Progress: 2/2 pages, 140/222 lines, 111/111 chars (Elapsed: 2 sec)
[Printer 0] Job ID:3 Progress: 2/2 pages, 210/222 lines, 111/111 chars (Elapsed: 3 sec)
[Printer 0] Job ID:3 Progress: 2/2 pages, 222/222 lines, 111/111 chars (Elapsed: 4 sec)

✅ [Printer 0] Completed Job ID:3 in 4 sec.

==================== Queues ====================
Job Queue: [(ID:2, Pages:1, Lines:11, Chars:200, Priority:2, Type:Newspaper), (ID:3, Pages:2, Lines:222, Chars:111, Priority:1, Type:Newspaper), None, None, None, None, None, None, None,
None]
Ready Queue: [(ID:2, Pages:1, Lines:11, Chars:200, Priority:2, Type:Newspaper), (ID:-1, Pages:-1, Lines:-1, Chars:-1, Priority:-1, Type:TERMINATE)]
================================================
♦ Semaphore value: 2

▯ [Printer 0] Starting Job ID:2 - Pages:1, Lines:11, Chars:200, Type:Newspaper, Priority:2
[Printer 0] Job ID:2 Progress: 1/1 pages, 11/11 lines, 200/200 chars (Elapsed: 1 sec)

✅ [Printer 0] Completed Job ID:2 in 1 sec.
```

Fig 6.2 Execution of the printjobs

In Figure 6.2 , the jobid 3 with pages 2 of priority 1 is executed first then the jobid 2 with pages 1 of priority2.because the jobid with highest priority is executed first then the lower priority jobs. We can also infer that printer can print only 70 lines in a second so the jobid 3 with pages 2 takes 4 seconds to complete .



```
Job Queue: [(ID:2, Pages:1, Lines:11, Chars:200, Priority:2, Type:Newspaper), (ID:3, Pages:2, Lines:222, Chars:111, Priority:1, Type:Newspaper), None, None, None, None, None, None, None,
None]
Ready Queue: [(ID:2, Pages:1, Lines:11, Chars:200, Priority:2, Type:Newspaper), (ID:-1, Pages:-1, Lines:-1, Chars:-1, Priority:-1, Type:TERMINATE)]
===============================================
⬧ Semaphore value: 2

▯ [Printer 0] Starting Job ID:2 - Pages:1, Lines:11, Chars:200, Type:Newspaper, Priority:2
[Printer 0] Job ID:2 Progress: 1/1 pages, 11/11 lines, 200/200 chars (Elapsed: 1 sec)

✅ [Printer 0] Completed Job ID:2 in 1 sec.

==================== Queues ====================
Job Queue: [(ID:2, Pages:1, Lines:11, Chars:200, Priority:2, Type:Newspaper), (ID:3, Pages:2, Lines:222, Chars:111, Priority:1, Type:Newspaper), None, None, None, None, None, None, None,
None]
Ready Queue: [(ID:-1, Pages:-1, Lines:-1, Chars:-1, Priority:-1, Type:TERMINATE)]
===============================================
⬧ Semaphore value: 1

▯ [Printer 0] Received termination signal. Shutting down.
```

Fig 6.3 Termination of the backend

In Figure 6.3, infer that the jobid 2 with pages 1 lines11,char 200 takes 1 second to complete its printing job. after processing of all jobs in the job queue and in the ready queue the printjob with values -1 is added to terminate the backend c program . to terminate the execution of the backend c program when the page with value -1 is used .

CHAPTER 7

CONCLUSION AND FUTURE WORK

In conclusion, our Printer Queue Management System successfully demonstrates the key ideas behind job scheduling, multithreading, and synchronization in an operating system environment. The project shows how multiple print jobs can be handled simultaneously by a system that uses a C-based backend for processing and a Python-based GUI for user interaction. Through this project, we learned how to manage concurrent access to shared resources and enforce deadlines on tasks to ensure efficient processing.

The results of our simulation are clear. When print jobs are submitted, the system first checks the input parameters such as the number of pages, lines, and characters. It then calculates how long the job should take by comparing the estimated times based on fixed speeds. If the job takes too long to complete, it is rejected with an appropriate message. This behavior mimics a practical printing environment, where complex tasks need to be managed carefully to avoid delays.

The backend program uses multiple threads—a submission thread to handle incoming jobs and one or more printer threads to process them. The use of mutexes and semaphores ensured that the job queues were updated correctly and that the log messages were printed without interruption. The system also features a job sorting function that arranges pending jobs by priority and estimated processing time.

While our system is designed for educational purposes and simulates print job processing rather than controlling a real printer, it clearly demonstrates important operating system concepts in a simple way. The separation of concerns between the C backend and the Python GUI helped us focus on both efficient processing as well as user interaction. This design also leaves room for future improvements, such as adding more complex scheduling strategies or enhancing the graphical interface with more detailed reporting tools.

Overall, the project met its objectives of simulating a multi-threaded scheduling environment and enforcing deadlines to handle print jobs. The results indicate that the project is a strong educational tool for understanding how operating systems manage multiple tasks and handle concurrent processes. The feedback logs, progress updates, and correct handling of failed jobs all contribute to the clear demonstration of the system’s functionality.

REFERENCES

[1] Silberschatz, A., Galvin, P. B., & Gagne, G. (2014). Operating System Concepts (9th ed.). Wiley.

[2] Kerrisk, M. (2010). The Linux Programming Interface. No Starch Press.

[3] Python Software Foundation. (2023). Tkinter — Python interface to Tcl/Tk. Retrieved from https://docs.python.org/3/library/tkinter.html

[4] cppreference.com. . (2023). C Standard Library. Retrieved from https://en.cppreference.com/w/c

[5] Pthreads Programming Tutorials. (n.d.). POSIX Threads Programming. Retrieved from https://computing.llnl.gov/tutorials/pthreads/