Title: Operating Systems Project Report

Name: Samriddhi Singh

Registration number: 23BCE1873

Slot: F2

Course name: Operating Systems

Course code: BCSE303L

Semester: Fall semester 2024-25

Submitted to: Dr. Rabindra Kumar Singh

Project Title: Deadlock Handling Mechanism Simulation in Real-Time.

1. Project Description:

The primary objective of this project is to implement and visualize a deadlock handling mechanism in a real-time system where processes require resources to execute. In real-time systems, the timely execution of processes is crucial, and a deadlock—a state where processes are unable to proceed because they are waiting for resources held by other processes—can severely disrupt system operation. Thus, detecting and preventing deadlock situations is essential to maintaining the stability and efficiency of the system.

This project simulates the behaviour of multiple processes that share a set of resources, where processes may request, acquire, and release resources. The project uses techniques to detect, prevent, and resolve deadlocks, ensuring that resources are managed safely in real-time environments. It achieves this by implementing deadlock detection algorithms, deadlock prevention strategies, and resource allocation mechanisms that adjust resource assignment to avoid unsafe states.

Key Features and Functionality:

1. **Resource Allocation Simulation:**
    - The system manages three types of resources shared among 20 processes. The resources are allocated dynamically as processes request them.
    - The system handles resource allocation in a way that avoids the system entering an unsafe state where deadlock might occur.
2. **Deadlock Detection Mechanism:**
    - The program detects whether any process is in a deadlocked state, unable to progress due to the unavailability of required resources.
    - The detection is done by simulating a resource allocation graph and checking if any processes are stuck in circular waiting conditions.
3. **Deadlock Recovery Mechanism:**
    - In case a deadlock is detected, the system implements a preemption strategy. This involves reclaiming resources from one or more processes to allow others to proceed, ultimately breaking the deadlock.
    - Alternatively, processes could be terminated to release resources and ensure the system can continue running.
4. **Deadlock Prevention:**
    - The project also implements the **Banker's Algorithm**, which ensures that resources are allocated in a way that avoids entering a deadlock-prone state.
    - The system only grants resources to a process if doing so will not lead to an unsafe state.

5. **Real-Time Visualization:**
   - The project uses **SDL2** (Simple DirectMedia Layer 2) to visually represent the state of the system. The window dynamically updates to show available resources, allocated resources, and processes.
   - The available resources are depicted as coloured bars on the screen. The available, allocated and needed resources for each process are represented with blue, green and red bars, respectively.
   - The system updates in real-time, showing the allocation of resources, changes in process states, and any deadlock situations that might arise.
6. **Multithreading and Synchronization:**
   - The project uses **POSIX threads (pthread)** to simulate concurrent processes, where each process runs in its own thread.
   - **Mutexes** and **condition variables** are used to ensure that resources are allocated and released safely without causing race conditions or resource conflicts.
7. **Interactive Event Handling:**
   - The program provides an interactive interface where users can monitor the system's state and visualize how processes acquire and release resources.
   - The system continues running in a loop until the max execution time has not been exceeded, simulating the continuous execution of real-time processes while responding to user events, like window closure.

2. Source Code:

```
#include <SDL2/SDL.h>

#include <SDL2/SDL_ttf.h>

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>


#define NUM_PROCESSES 20

#define NUM_RESOURCES 3

#define MAX_EXECUTION_TIME 5

#define WINDOW_WIDTH 800
```

```
#define WINDOW_HEIGHT 1200
#define RESOURCE_BAR_WIDTH 20

// SDL2 window and renderer
SDL_Window *window = NULL; //ref to window
SDL_Renderer *renderer = NULL; //ref to the screen on window

int available[NUM_RESOURCES] = {3, 3, 3}; // Available resources
int allocation[NUM_PROCESSES][NUM_RESOURCES];
int maximum[NUM_PROCESSES][NUM_RESOURCES];
int need[NUM_PROCESSES][NUM_RESOURCES];

pthread_mutex_t resourceLock;
pthread_cond_t resourceCond;
int can_allocate(int process_id) {
    for (int i = 0; i < NUM_RESOURCES; i++) {
        if (need[process_id][i] > available[i]) {
            return 0;  // Cannot allocate resources if need > available
        }
    }
    return 1;
}

void allocate_resources(int process_id) {
    for (int i = 0; i < NUM_RESOURCES; i++) {
        available[i] -= need[process_id][i];
        allocation[process_id][i] += need[process_id][i];
        need[process_id][i] = 0;
    }
}
```

```
void release_resources(int process_id) {
    for (int i = 0; i < NUM_RESOURCES; i++) {
        available[i] += allocation[process_id][i];
        allocation[process_id][i] = 0;  // Resources released
    }
    pthread_cond_broadcast(&resourceCond);  // Wake up other waiting processes
}
void render_text(const char* text, int x, int y, SDL_Color color) {
    TTF_Font *font = TTF_OpenFont("C:/Users/samriddhi singh/Downloads/Arial.ttf", 10);
    if (font == NULL) {
        printf("TTF_OpenFont: %s\n", TTF_GetError());
        return;
    }
    SDL_Surface *textSurface = TTF_RenderText_Solid(font, text, color); //screen
    SDL_Texture *textTexture = SDL_CreateTextureFromSurface(renderer, textSurface);
    SDL_Rect textRect = { x, y, textSurface->w, textSurface->h };//x,y->pos of rect
    SDL_RenderCopy(renderer, textTexture, NULL, &textRect);
    SDL_FreeSurface(textSurface);
    SDL_DestroyTexture(textTexture);
    TTF_CloseFont(font);
}

// Visualizing available resources as blue coloured bars
void draw_resources() {
    int resource_base_y = 150;  // Reduced the vertical space for resources
    for (int i = 0; i < NUM_RESOURCES; i++) {
        SDL_SetRenderDrawColor(renderer, 0, 0, 255, 255);  //r,g,b,a Blue for available
resources
        SDL_Rect bar = {50 + i * (RESOURCE_BAR_WIDTH + 50), resource_base_y,
available[i] * RESOURCE_BAR_WIDTH, 30};  // Horizontal bar
```

```
        SDL_RenderFillRect(renderer, &bar);
        char label[20];
        snprintf(label, sizeof(label), "Resource %d", i + 1);
        render_text(label, 50 + i * (RESOURCE_BAR_WIDTH + 50), resource_base_y + 40,
(SDL_Color){255, 255, 255, 255});
    }
}

// Visualizing processes as coloured rectangles and text
void draw_processes() {
    int process_base_y = 200;  // Start drawing processes just below resources
    int process_height = 40;  // Increase space between processes
    for (int i = 0; i < NUM_PROCESSES; i++) {
        // Draw each process with allocated resources and needed resources
        for (int j = 0; j < NUM_RESOURCES; j++) {
            // Draw allocation (green)
            SDL_SetRenderDrawColor(renderer, 0, 255, 0, 255);  // Green for allocation
            SDL_Rect rect = {50 + j * (RESOURCE_BAR_WIDTH + 50), process_base_y + i *
process_height, allocation[i][j] * RESOURCE_BAR_WIDTH, 30};
            SDL_RenderFillRect(renderer, &rect);
            // Draw need (red)
            SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);  // Red for need
            SDL_Rect need_rect = {50 + j * (RESOURCE_BAR_WIDTH + 50), process_base_y
+ i * process_height + 10, need[i][j] * RESOURCE_BAR_WIDTH, 10};
            SDL_RenderFillRect(renderer, &need_rect);
        }
        // Render the process ID label
        char label[20];
        snprintf(label, sizeof(label), "Process %d", i + 1);
        render_text(label, 50, process_base_y + i * process_height + 40, (SDL_Color){255, 255,
255, 255});
    }
```

```
}

// Function to initialize SDL2
int init_SDL() {
    if (SDL_Init(SDL_INIT_VIDEO | SDL_INIT_TIMER) < 0) {
        printf("SDL could not initialize! SDL_Error: %s\n", SDL_GetError());
        return 0;
    }
    window = SDL_CreateWindow("Resource Allocation Simulation",
SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED,
WINDOW_WIDTH, WINDOW_HEIGHT, SDL_WINDOW_SHOWN);
    if (!window) {
        printf("Window could not be created! SDL_Error: %s\n", SDL_GetError());
        return 0;
    }
    renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED);
    if (!renderer) {
        printf("Renderer could not be created! SDL_Error: %s\n", SDL_GetError());
        return 0;
    }
    if (TTF_Init() == -1) {
        printf("SDL_ttf could not initialize! TTF_Error: %s\n", TTF_GetError());
        return 0;
    }
    return 1;
}

// Function to close SDL2 and clean up
void close_SDL() {
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();
    TTF_Quit();
}

void *process_thread(void *arg) {
    int process_id = *((int *)arg);
    int execution_time = 0;
    while (execution_time < MAX_EXECUTION_TIME) {
        pthread_mutex_lock(&resourceLock);
        while (!can_allocate(process_id)) {
            pthread_cond_wait(&resourceCond, &resourceLock);
        }
        allocate_resources(process_id);
        pthread_mutex_unlock(&resourceLock);
        // Simulate process work time delay
        sleep(1);
        execution_time++;
        pthread_mutex_lock(&resourceLock);
        release_resources(process_id);
        pthread_mutex_unlock(&resourceLock);
        sleep(1);  // Delay between cycles to simulate process time
    }
    pthread_exit(NULL);
}

int SDL_main(int argc, char *argv[]) {
    pthread_t processes[NUM_PROCESSES];
    int process_ids[NUM_PROCESSES];
    for (int i = 0; i < NUM_PROCESSES; i++) {
        process_ids[i] = i;
        for (int j = 0; j < NUM_RESOURCES; j++) {
            allocation[i][j] = 0;
            maximum[i][j] = 1 + rand() % 3;  // Random maximum resource need between 1 and
3
            need[i][j] = maximum[i][j];
        }
    }
    pthread_mutex_init(&resourceLock, NULL);
    pthread_cond_init(&resourceCond, NULL);
    if (!init_SDL()) {
        return -1;
    }
    // Create process threads
    for (int i = 0; i < NUM_PROCESSES; i++) {
        pthread_create(&processes[i], NULL, process_thread, (void *)&process_ids[i]);
    }
    int running = 1;
    while (running) {
        SDL_Event e;
        while (SDL_PollEvent(&e)) { //checks for pending events and then starts event e
            if (e.type == SDL_QUIT) { //if user closes window
                running = 0;
            }
        }
        SDL_SetRenderDrawColor(renderer, 20, 20, 20, 255);  // Dark background
        SDL_RenderClear(renderer);
        draw_resources();    // Draw available resources
        draw_processes();    // Draw process allocation and needs
        SDL_RenderPresent(renderer);  // Show the rendered content
        SDL_Delay(100);  // Delay
    }
    for (int i = 0; i < NUM_PROCESSES; i++) {
        pthread_join(processes[i], NULL);
    }
    // Clean up SDL2
    pthread_mutex_destroy(&resourceLock);
    pthread_cond_destroy(&resourceCond);
    close_SDL();
    return 0;
}
```

Code Walkthrough:

The provided code is designed to simulate a system with multiple processes (threads) competing for a limited number of resources. Here's a breakdown of the key components and how they contribute to the overall deadlock handling mechanism.

Data Structures:

- available[NUM_RESOURCES]: This array keeps track of how many units of each resource are available in the system.
- allocation[NUM_PROCESSES][NUM_RESOURCES]: This 2D array stores how many resources each process is currently holding.
- maximum[NUM_PROCESSES][NUM_RESOURCES]: This 2D array specifies the maximum number of resources that each process may need.
- need[NUM_PROCESSES][NUM_RESOURCES]: This array calculates how many resources each process still needs to finish its execution. It is computed as need = maximum - allocation.

Resource Allocation Logic:

- can_allocate(process_id): This function checks whether a process can be allocated the resources it needs. It returns 1 (true) if all the needed resources are available, otherwise 0 (false).
- allocate_resources(process_id): This function allocates resources to the process, updating the available resources and the process's allocation and need tables.
- release_resources(process_id): This function is responsible for releasing the resources held by a process once it has finished executing. It updates the available resources and resets the process's allocation.

Deadlock Detection:

- Deadlock detection is implied in the code by periodically checking if a process can't proceed because it is waiting for resources that are being held by others. If a process cannot acquire the required resources, it waits on a condition variable (resourceCond), which ensures that it doesn’t proceed until resources become available.

Multithreading and Synchronization:

- Thread Creation (pthread_create): Each process runs in a separate thread, simulating independent tasks that compete for shared resources.
- Mutex (pthread_mutex_lock): A mutex (resourceLock) ensures that only one thread can access the resource allocation and release logic at a time, preventing data races.
- Condition Variable (pthread_cond_wait and pthread_cond_broadcast): Processes wait on the condition variable if resources are unavailable. When resources are released, the condition variable is broadcast to wake up waiting threads.

Graphical Visualization:

- SDL2 Functions: The project uses the SDL2 library to create a graphical interface that shows the resource allocation status. The window displays:
  - Resource Bars: Blue bars show the available amount of each resource.
  - Process Rectangles: Each process is shown with green bars representing allocated resources and red bars representing remaining resource needs.
  - draw_resources(): This function draws the resource bars, each representing the available amount of a specific resource.
  - draw_processes(): This function visualizes each process’s allocation and remaining needs. Green bars represent the allocated resources, while red bars represent the unmet needs.

Main Program Logic (SDL_main):

- The main function initializes all processes, creates threads for each process, and runs the SDL2 rendering loop.
- The loop handles events like window closing and continuously updates the display, showing the latest state of resources and processes.
- The system periodically allocates and releases resources, with the UI updating to show the effects.

Deadlock Handling Mechanism:

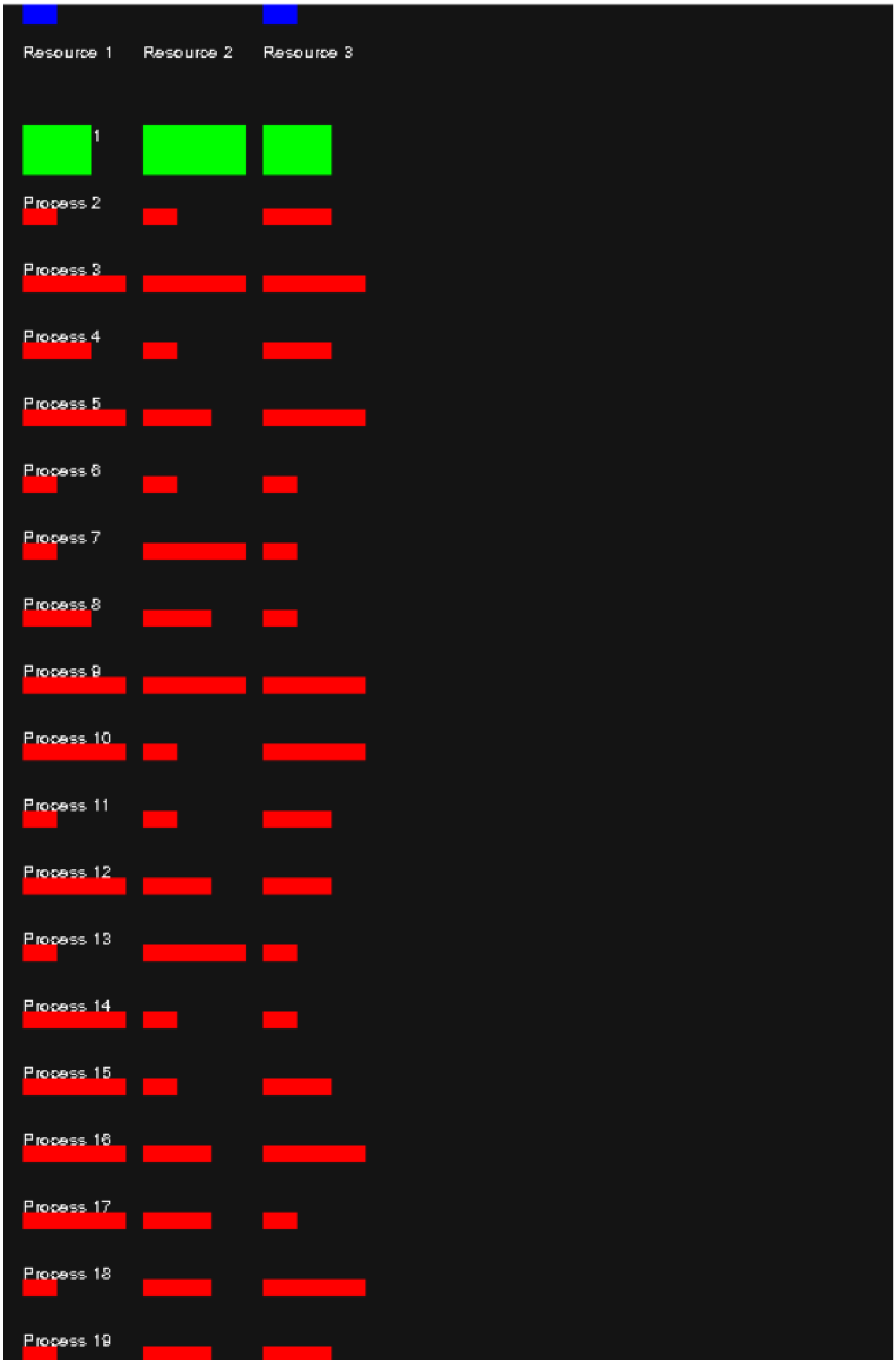
- While the core of the deadlock handling mechanism is not explicitly shown (e.g., using the Wait-for Graph to detect cycles), the structure is in place to handle deadlocks if further enhancements were added.
- The Banker's Algorithm could be applied to prevent the system from entering an unsafe state by checking whether a process's request can be granted without causing a deadlock.

3. Output Screens:

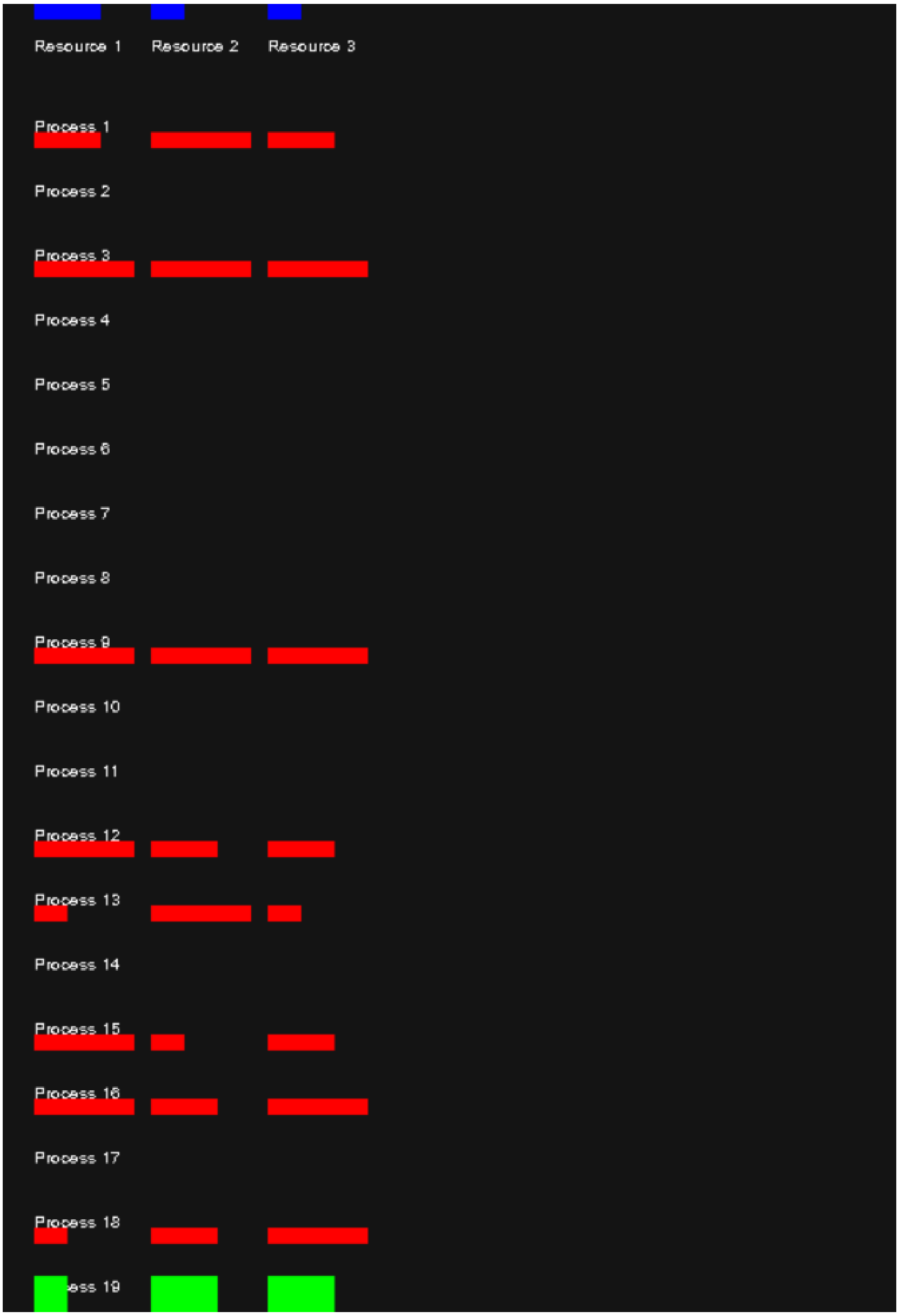
In the attached output screenshots on the following pages:

- **Resource Allocation Screen:**
  - The screen shows three coloured bars (blue) representing the available quantities of three resource types.
  - Each process is represented by a rectangle with two bars: a green bar for the allocated resources and a red bar for the unmet resources.
  - The processes may be seen waiting for resources, indicated by changes in the red bars.
- **Dynamic Updates:**
  - As the simulation runs, the process resource allocations change. Some processes may hold resources, while others may be blocked (waiting).
  - The system visually updates, and you can observe how resources are allocated and released over time.

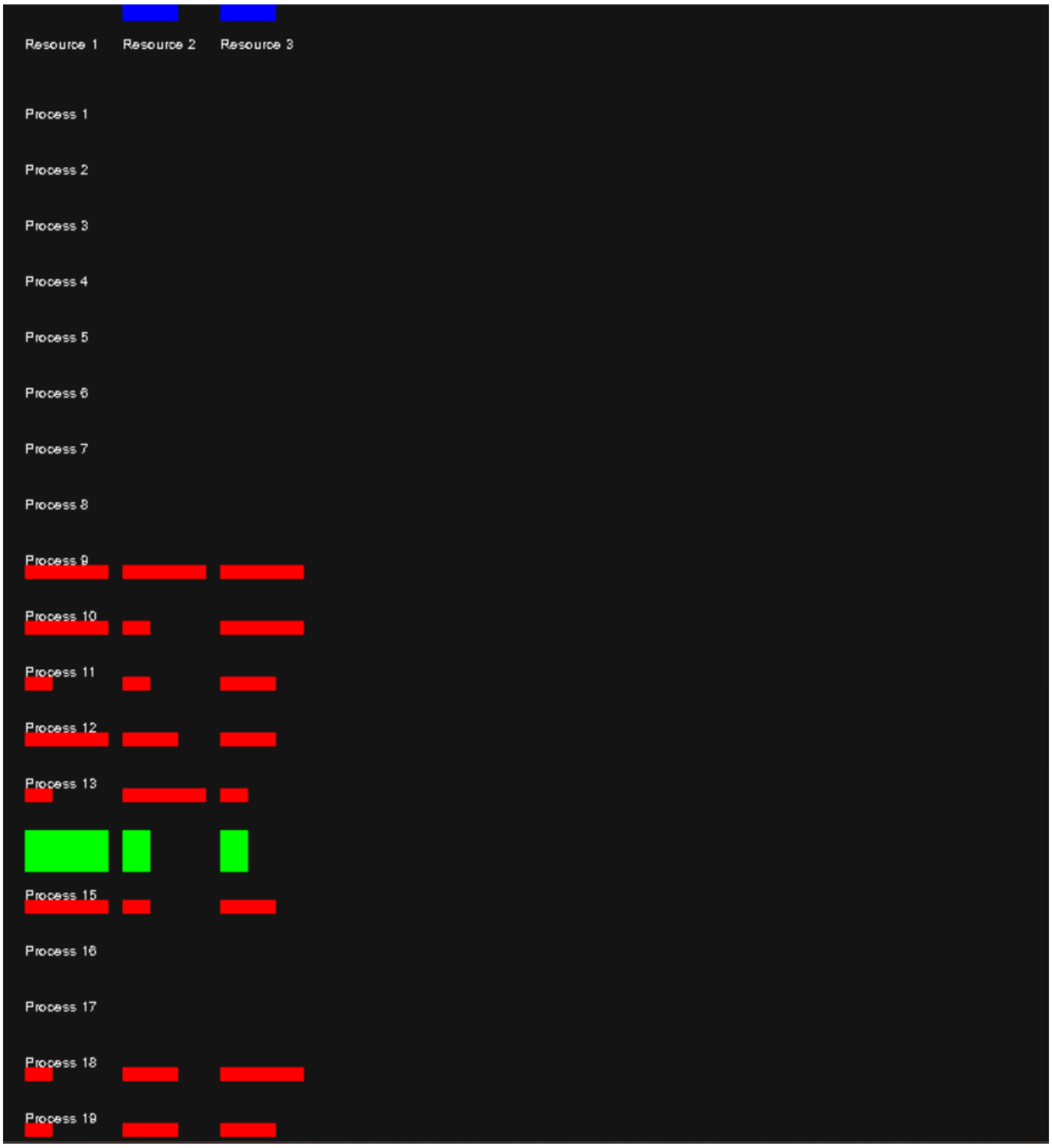
In this we can see that process 1's needs could be fulfilled by the available resources and thus, resources are being allocated to it. Once 5 execution cycles have been completed it will get terminated.



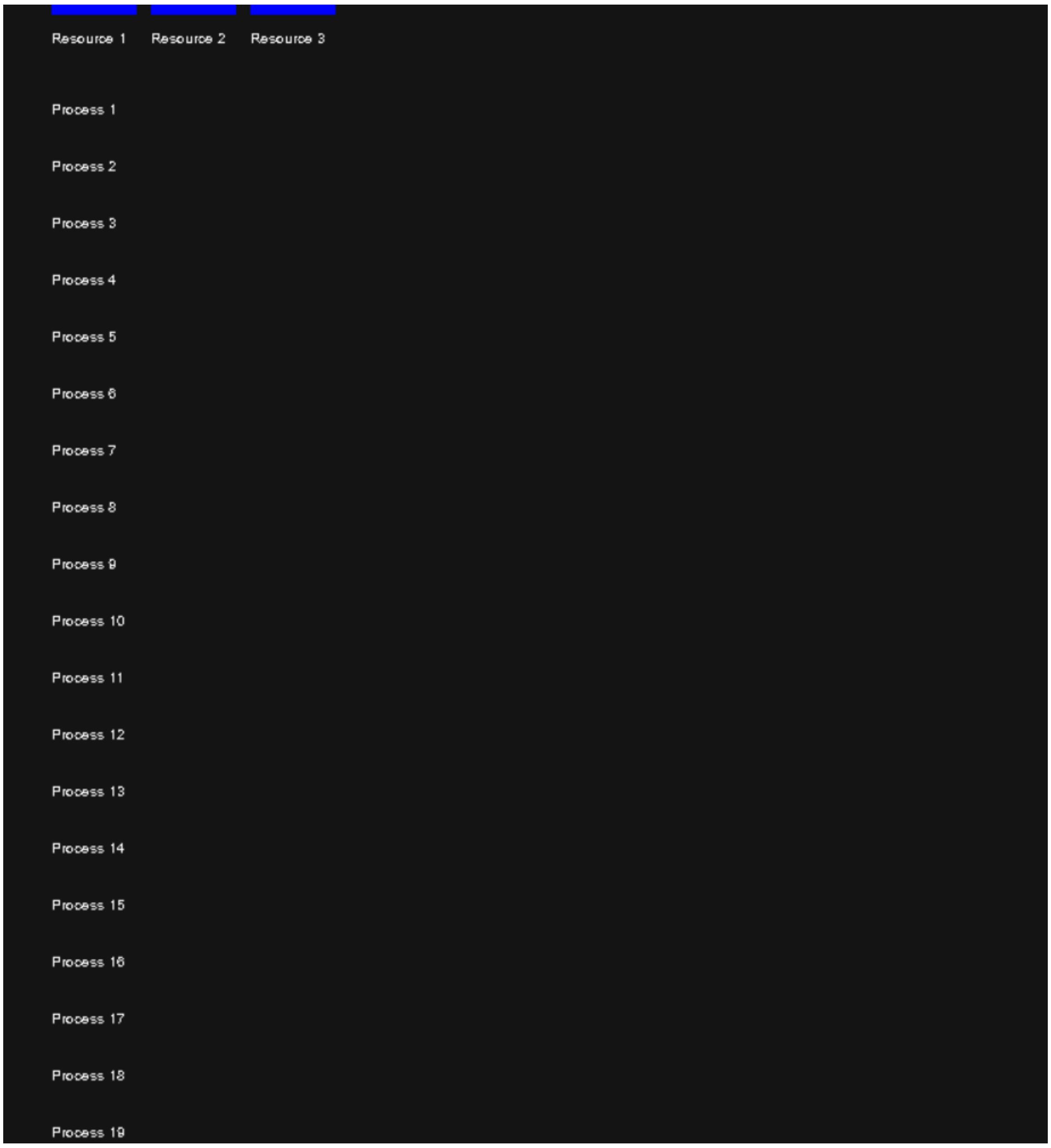
In this we can see that process 19's needs could be fulfilled by the available resources and thus, resources are being allocated to it. Once 5 execution cycles have been completed it will get terminated. We also see that process1 has not terminated yet, this is because it has not completed 5 execution cycles yet.



In this we can see that process 14's needs could be fulfilled by the available resources and thus, resources are being allocated to it. Once 5 execution cycles have been completed it will get terminated. By this time, process1 to process 8 have already completed execution and have been terminated.



Once all processes have finished executing for 5 cycles(max execution time) and have been terminated, all three instances of each resource1, resource2, resource3 is now available.



Conclusion:

This project demonstrates how deadlock handling mechanisms work in real-time systems by simulating processes and resources, managing resource allocation, and avoiding deadlock situations. It uses multithreading, mutexes, and condition variables to simulate a concurrent system, and SDL2 for real-time graphical visualization.

By employing techniques like deadlock detection, deadlock prevention using the Banker's Algorithm, and resource allocation, this project provides a clear, visual representation of how a real-time system can handle deadlocks, ensuring that processes continue to run smoothly even when resources are limited.