Operating Systems 2024 Spring Term Project

Team Members: Ömer Faruk Aydın B201202023

Github Link: https://github.com/ofaydn/dispatcherSystem

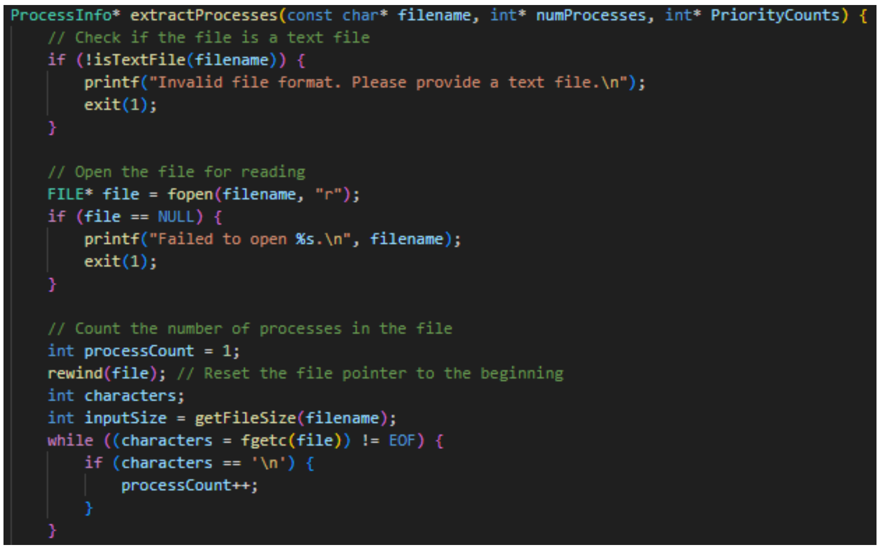
Main idea:

Main idea of this project is to develop a dispatcher system that will evaluate processes obtained from the input and out them in correct algorithms to maintain stability in a CPU working environment. Project is coded in C language. Main functions of this dispatcher system are FCFS, SJF and RR algorithms.

- ### Getting the Input

Getting input is one of the most important parts of this dispatcher system since if there is no process to evaluate, program would be meaningless. In order to make program more manageable and error-free. Functions for managing input are:

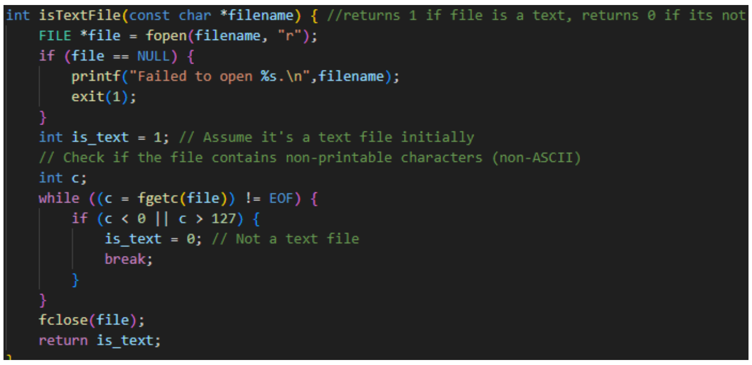
1. ProcessInfo* extractProcesses(const char* filename, int* numProcesses, int* PriorityCounts)



```c
ProcessInfo* extractProcesses(const char* filename, int* numProcesses, int* PriorityCounts) {
    // Check if the file is a text file
    if (!isTextFile(filename)) {
        printf("Invalid file format. Please provide a text file.\n");
        exit(1);
    }

    // Open the file for reading
    FILE* file = fopen(filename, "r");
    if (file == NULL) {
        printf("Failed to open %s.\n", filename);
        exit(1);
    }

    // Count the number of processes in the file
    int processCount = 1;
    rewind(file); // Reset the file pointer to the beginning
    int characters;
    int inputSize = getFileSize(filename);
    while ((characters = fgetc(file)) != EOF) {
        if (characters == '\n') {
            processCount++;
        }
    }
```

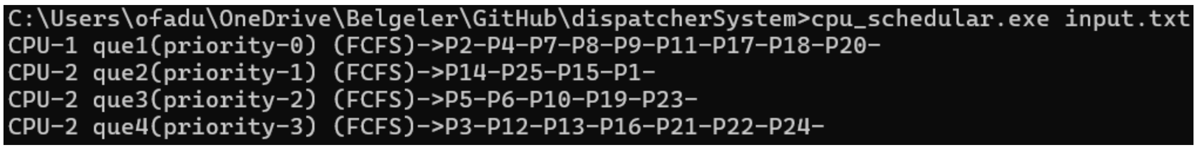
This is just beginning of the extraction process. This function returns array of structs which is structure of a process. First steps of this function are checking the input file then opening it. Then determining the numbers of processes. I will be explaining isTextFile and getFileSize functions and then continue to explain extractProcesses functions.

2. int isTextFile(const char *filename)

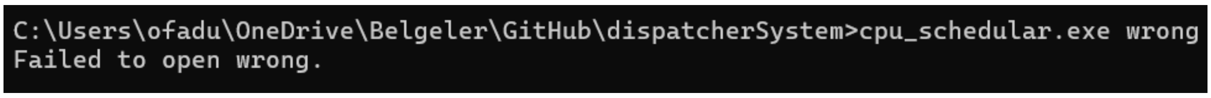


```c
int isTextFile(const char *filename) { //returns 1 if file is a text, returns 0 if its not
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        printf("Failed to open %s.\n",filename);
        exit(1);
    }
    int is_text = 1; // Assume it's a text file initially
    // Check if the file contains non-printable characters (non-ASCII)
    int c;
    while ((c = fgetc(file)) != EOF) {
        if (c < 0 || c > 127) {
            is_text = 0; // Not a text file
            break;
        }
    }
    fclose(file);
    return is_text;
```

This function simply checks the input whether it is okay to take this file as an input or not. After file operations closes file. Here is an example when getting proper input type and one with improper input type.

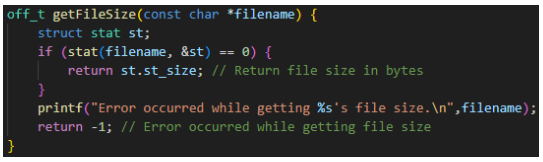


```
C:\Users\ofadu\OneDrive\Belgeler\GitHub\dispatcherSystem>cpu_schedular.exe input.txt
CPU-1 que1(priority-0) (FCFS)->P2-P4-P7-P8-P9-P11-P17-P18-P20-
CPU-2 que2(priority-1) (FCFS)->P14-P25-P15-P1-
CPU-2 que3(priority-2) (FCFS)->P5-P6-P10-P19-P23-
CPU-2 que4(priority-3) (FCFS)->P3-P12-P13-P16-P21-P22-P24-
```



```
C:\Users\ofadu\OneDrive\Belgeler\GitHub\dispatcherSystem>cpu_schedular.exe wrong
Failed to open wrong.
```

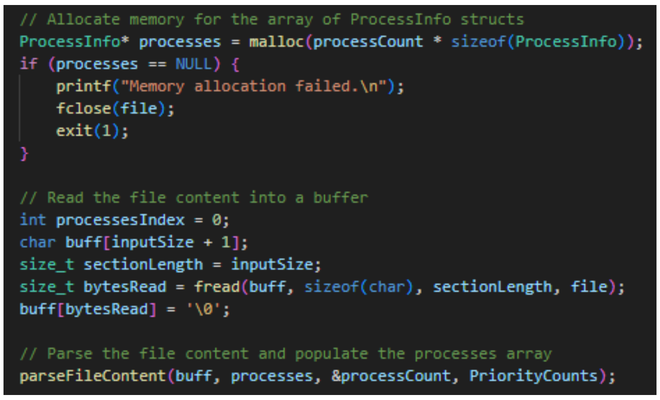
3. off_t getFileSize()



```c
off_t getFileSize(const char *filename) {
    struct stat st;
    if (stat(filename, &st) == 0) {
        return st.st_size; // Return file size in bytes
    }
    printf("Error occurred while getting %s's file size.\n",filename);
    return -1; // Error occurred while getting file size
}
```

This function gets the size of the input file. This value will be used when counting the number of processes.

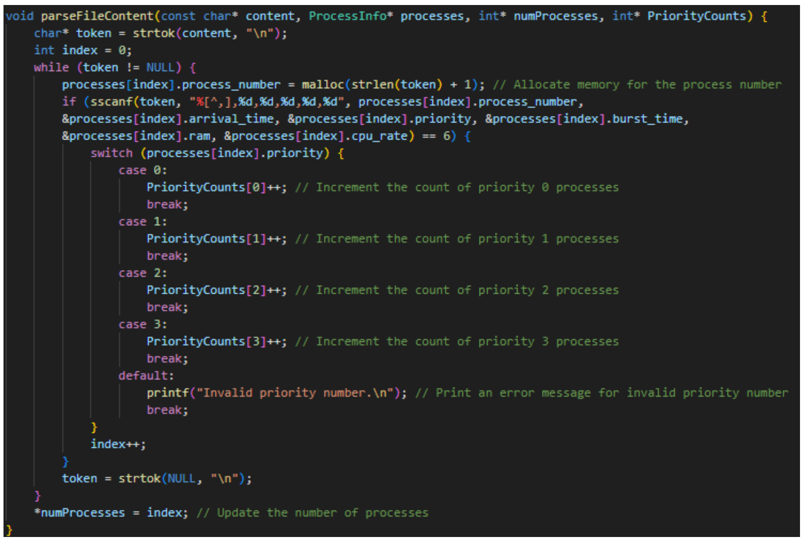
4. Continuing to extractProcesses



```c
// Allocate memory for the array of ProcessInfo structs
ProcessInfo* processes = malloc(processCount * sizeof(ProcessInfo));
if (processes == NULL) {
    printf("Memory allocation failed.\n");
    fclose(file);
    exit(1);
}

// Read the file content into a buffer
int processesIndex = 0;
char buff[inputSize + 1];
size_t sectionLength = inputSize;
size_t bytesRead = fread(buff, sizeof(char), sectionLength, file);
buff[bytesRead] = '\0';

// Parse the file content and populate the processes array
parseFileContent(buff, processes, &processCount, PriorityCounts);
```

After determining the numbers of the processes and validating the input, program allocates memory for our process array by using processCount and size of the ProcessInfo struct. Then proceeds to parse that input file into our array.

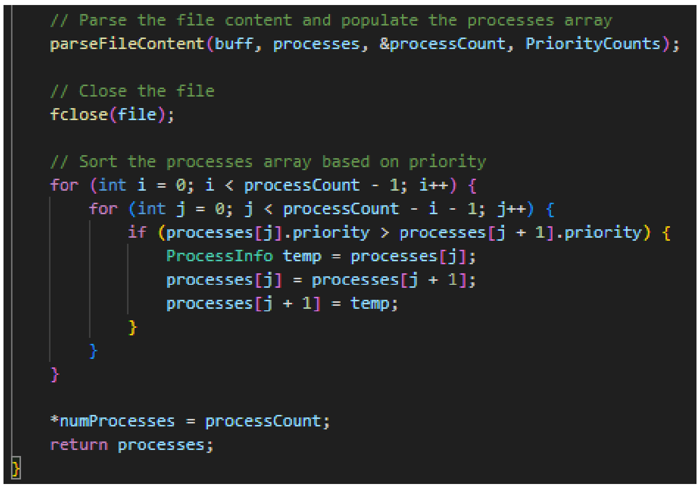
5. Void parseFileContent(cons char* content, ProcessInfo* processes, int* numProcesses, int* PriorityCounts)



```c
void parseFileContent(const char* content, ProcessInfo* processes, int* numProcesses, int* PriorityCounts) {
    char* token = strtok(content, "\n");
    int index = 0;
    while (token != NULL) {
        processes[index].process_number = malloc(strlen(token) + 1); // Allocate memory for the process number
        if (sscanf(token, "%[^,],%d,%d,%d,%d,%d", processes[index].process_number,
        &processes[index].arrival_time, &processes[index].priority, &processes[index].burst_time,
        &processes[index].ram, &processes[index].cpu_rate) == 6) {
            switch (processes[index].priority) {
                case 0:
                    PriorityCounts[0]++; // Increment the count of priority 0 processes
                    break;
                case 1:
                    PriorityCounts[1]++; // Increment the count of priority 1 processes
                    break;
                case 2:
                    PriorityCounts[2]++; // Increment the count of priority 2 processes
                    break;
                case 3:
                    PriorityCounts[3]++; // Increment the count of priority 3 processes
                    break;
                default:
                    printf("Invalid priority number.\n"); // Print an error message for invalid priority number
                    break;
            }
            index++;
        }
        token = strtok(NULL, "\n");
    }
    *numProcesses = index; // Update the number of processes
}
```

This function does more than just parsing input to processes array. It also tracks the processes based on their priority number. This priority count array later will be used when grouping processes.

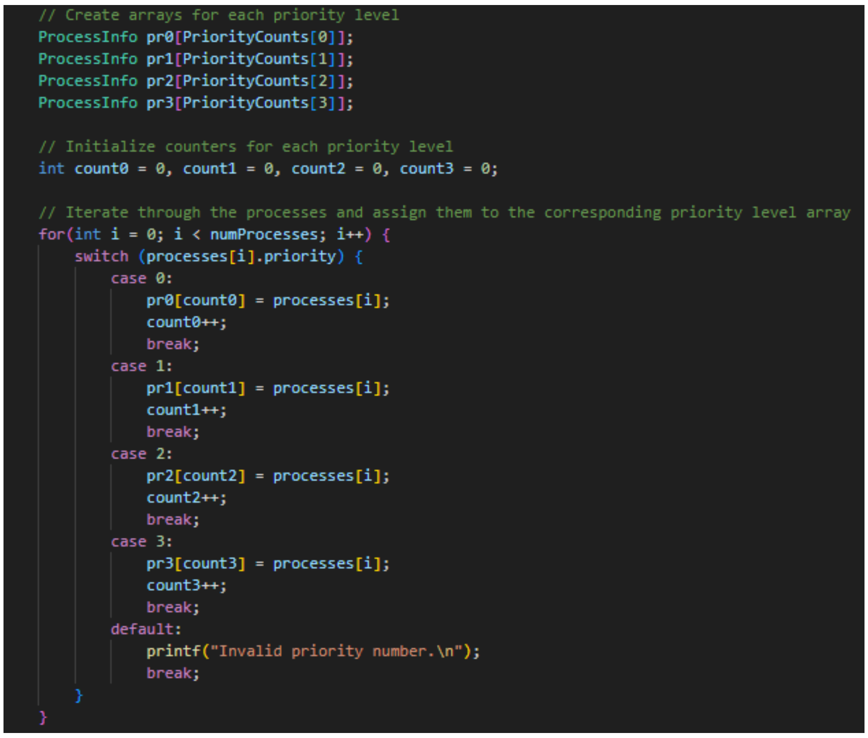
6. After parsing input file content to the process array, program closes the file. After that it sorts the process array based on their priorities. At the end of the function it returns the process array.



```c
    // Parse the file content and populate the processes array
    parseFileContent(buff, processes, &processCount, PriorityCounts);

    // Close the file
    fclose(file);

    // Sort the processes array based on priority
    for (int i = 0; i < processCount - 1; i++) {
        for (int j = 0; j < processCount - i - 1; j++) {
            if (processes[j].priority > processes[j + 1].priority) {
                ProcessInfo temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }

    *numProcesses = processCount;
    return processes;
}
```

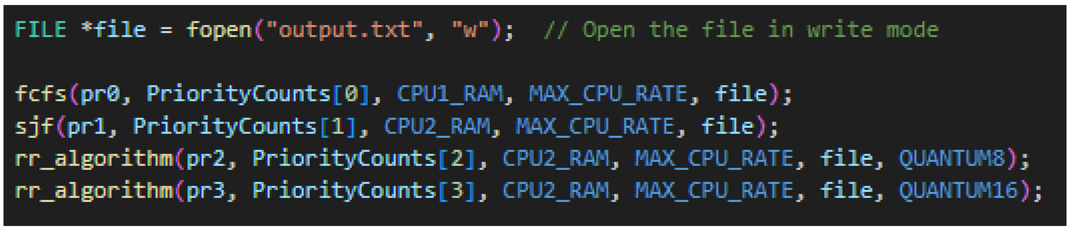
- **Processing the input**

1. After successfully getting the input from the input file and parsing it to the process array. Program defines 4 different process arrays based on their priorities. Program does these using values that generated previously.



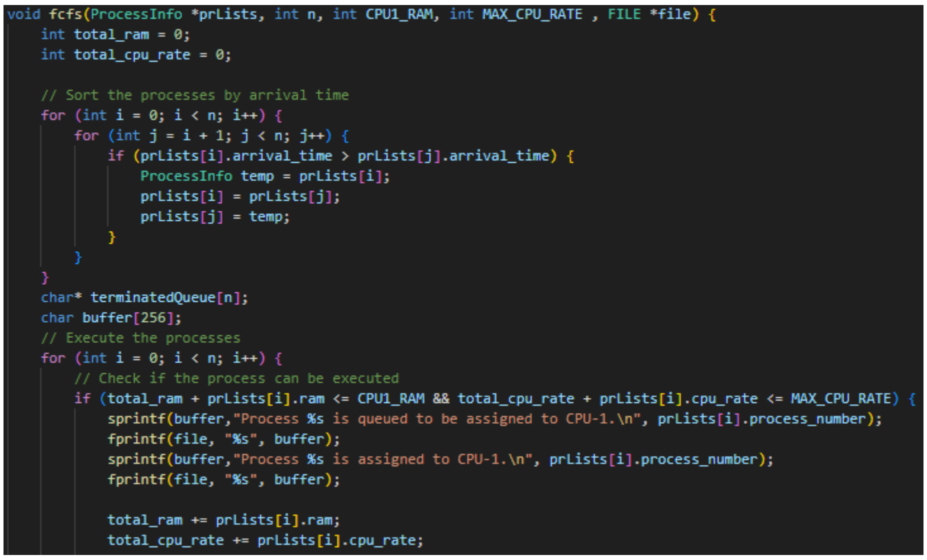
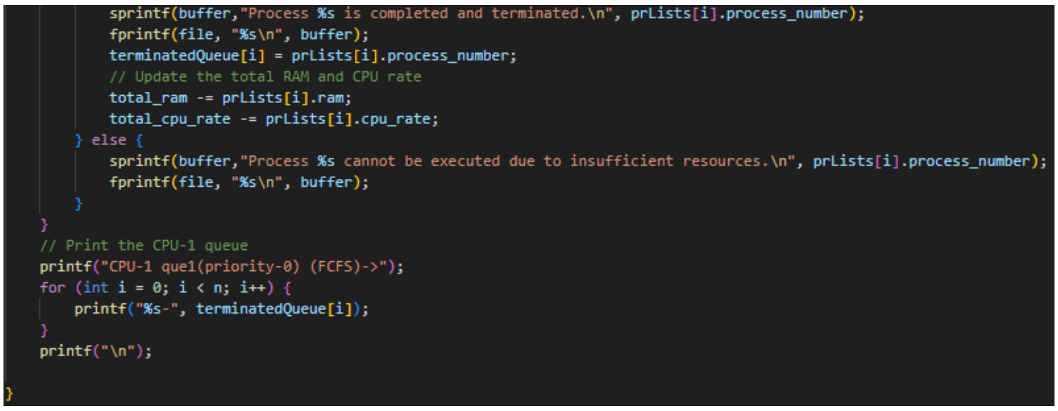
```c
    // Create arrays for each priority level
    ProcessInfo pr0[PriorityCounts[0]];
    ProcessInfo pr1[PriorityCounts[1]];
    ProcessInfo pr2[PriorityCounts[2]];
    ProcessInfo pr3[PriorityCounts[3]];

    // Initialize counters for each priority level
    int count0 = 0, count1 = 0, count2 = 0, count3 = 0;

    // Iterate through the processes and assign them to the corresponding priority level array
    for(int i = 0; i < numProcesses; i++) {
        switch (processes[i].priority) {
            case 0:
                pr0[count0] = processes[i];
                count0++;
                break;
            case 1:
                pr1[count1] = processes[i];
                count1++;
                break;
            case 2:
                pr2[count2] = processes[i];
                count2++;
                break;
            case 3:
                pr3[count3] = processes[i];
                count3++;
                break;
            default:
                printf("Invalid priority number.\n");
                break;
        }
    }
```

2. After making proper ordering based on priority number, its time to perform algorithms based on processes' priority numbers. Since the processes with the highest priority must be sorted according to the FCFS algorithm, the program processes them first. Then SJF and Round Robin algorithm with 8 quantum time and finally Round Robin algorithm with 16 quantum time.


```
FILE *file = fopen("output.txt", "w");  // Open the file in write mode

fcfs(pr0, PriorityCounts[0], CPU1_RAM, MAX_CPU_RATE, file);
sjf(pr1, PriorityCounts[1], CPU2_RAM, MAX_CPU_RATE, file);
rr_algorithm(pr2, PriorityCounts[2], CPU2_RAM, MAX_CPU_RATE, file, QUANTUM8);
rr_algorithm(pr3, PriorityCounts[3], CPU2_RAM, MAX_CPU_RATE, file, QUANTUM16);
```

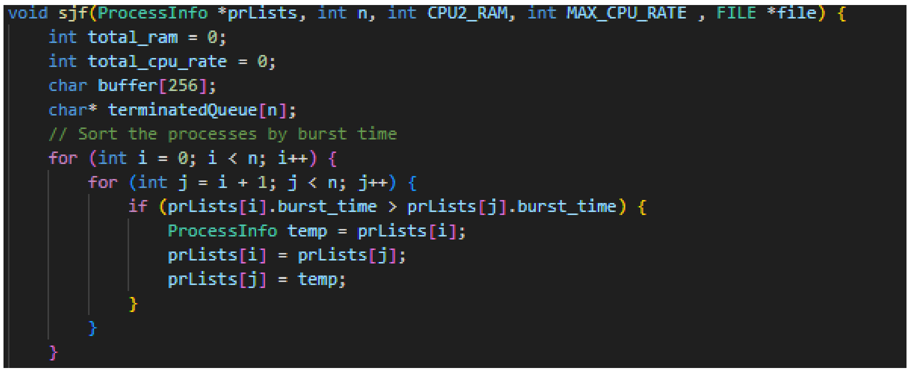
a. **FCFS Algorithm**

This algorithm takes an array of ProcessInfo structures, the number of processes, the maximum RAM and CPU rate for CPU1 because priority 0 processes will be assigned to CPU-1, and a file pointer for logging. The function first sorts the processes by their arrival time. Then, it iterates over the sorted processes, checking if each process can be executed based on the available RAM and CPU rate. If a process can be executed, it is assigned to CPU1, its resource usage is added to the total, and a log message is written to the file. After execution, the process is terminated, its resource usage is subtracted from the total, and it is added to the terminatedQueue. If a process cannot be executed due to insufficient resources, a log message is written to the file. Finally, the function prints the list of processes that were assigned to CPU1 and terminated.

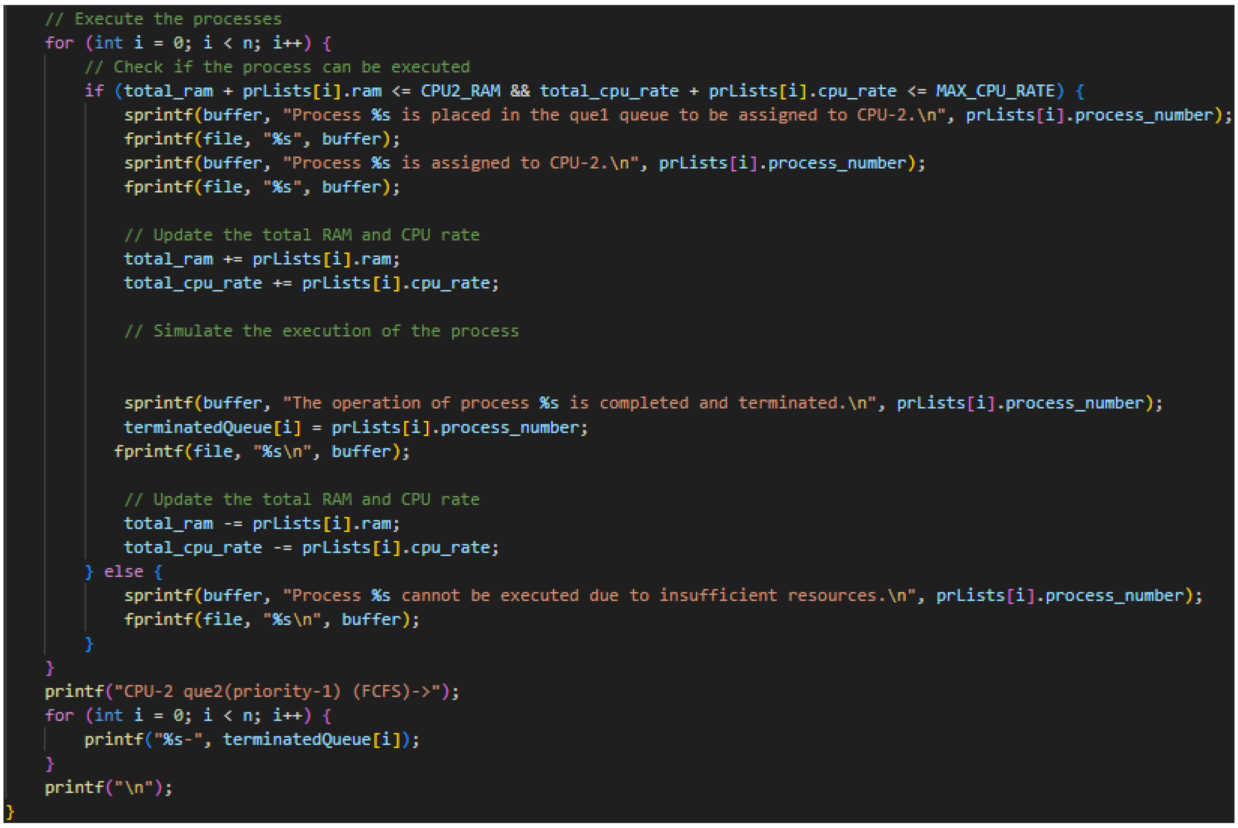

```
void fcfs(ProcessInfo *prLists, int n, int CPU1_RAM, int MAX_CPU_RATE , FILE *file) {
    int total_ram = 0;
    int total_cpu_rate = 0;

    // Sort the processes by arrival time
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (prLists[i].arrival_time > prLists[j].arrival_time) {
                ProcessInfo temp = prLists[i];
                prLists[i] = prLists[j];
                prLists[j] = temp;
            }
        }
    }
    char* terminatedQueue[n];
    char buffer[256];
    // Execute the processes
    for (int i = 0; i < n; i++) {
        // Check if the process can be executed
        if (total_ram + prLists[i].ram <= CPU1_RAM && total_cpu_rate + prLists[i].cpu_rate <= MAX_CPU_RATE) {
            sprintf(buffer,"Process %s is queued to be assigned to CPU-1.\n", prLists[i].process_number);
            fprintf(file, "%s", buffer);
            sprintf(buffer,"Process %s is assigned to CPU-1.\n", prLists[i].process_number);
            fprintf(file, "%s", buffer);

            total_ram += prLists[i].ram;
            total_cpu_rate += prLists[i].cpu_rate;
```

```
            sprintf(buffer,"Process %s is completed and terminated.\n", prLists[i].process_number);
            fprintf(file, "%s\n", buffer);
            terminatedQueue[i] = prLists[i].process_number;
            // Update the total RAM and CPU rate
            total_ram -= prLists[i].ram;
            total_cpu_rate -= prLists[i].cpu_rate;
        } else {
            sprintf(buffer,"Process %s cannot be executed due to insufficient resources.\n", prLists[i].process_number);
            fprintf(file, "%s\n", buffer);
        }
    }
    // Print the CPU-1 queue
    printf("CPU-1 que1(priority-0) (FCFS)->");
    for (int i = 0; i < n; i++) {
        printf("%s-", terminatedQueue[i]);
    }
    printf("\n");

}
```

b. **SJF Algorithm**

Execution of SJF algorithm is pretty similar to FCFS algorithm. Except this time it sorts he processes based on their burst time and decides to start with the shortest one. It sorts processes using Bubble Sort algorithm.



```c
void sjf(ProcessInfo *prLists, int n, int CPU2_RAM, int MAX_CPU_RATE , FILE *file) {
    int total_ram = 0;
    int total_cpu_rate = 0;
    char buffer[256];
    char* terminatedQueue[n];
    // Sort the processes by burst time
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (prLists[i].burst_time > prLists[j].burst_time) {
                ProcessInfo temp = prLists[i];
                prLists[i] = prLists[j];
                prLists[j] = temp;
            }
        }
    }
```

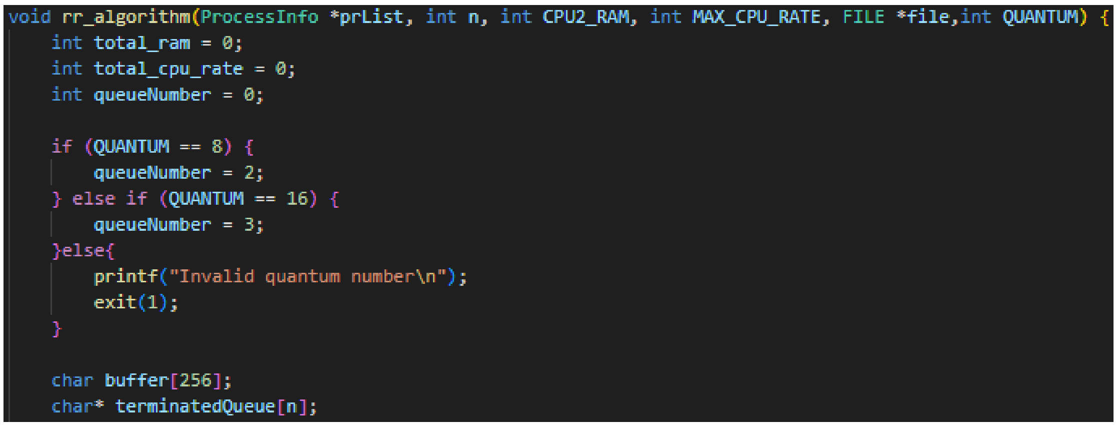


```c
    // Execute the processes
    for (int i = 0; i < n; i++) {
        // Check if the process can be executed
        if (total_ram + prLists[i].ram <= CPU2_RAM && total_cpu_rate + prLists[i].cpu_rate <= MAX_CPU_RATE) {
            sprintf(buffer, "Process %s is placed in the que1 queue to be assigned to CPU-2.\n", prLists[i].process_number);
            fprintf(file, "%s", buffer);
            sprintf(buffer, "Process %s is assigned to CPU-2.\n", prLists[i].process_number);
            fprintf(file, "%s", buffer);

            // Update the total RAM and CPU rate
            total_ram += prLists[i].ram;
            total_cpu_rate += prLists[i].cpu_rate;

            // Simulate the execution of the process


            sprintf(buffer, "The operation of process %s is completed and terminated.\n", prLists[i].process_number);
            terminatedQueue[i] = prLists[i].process_number;
           fprintf(file, "%s\n", buffer);

            // Update the total RAM and CPU rate
            total_ram -= prLists[i].ram;
            total_cpu_rate -= prLists[i].cpu_rate;
        } else {
            sprintf(buffer, "Process %s cannot be executed due to insufficient resources.\n", prLists[i].process_number);
            fprintf(file, "%s\n", buffer);
        }
    }
    printf("CPU-2 que2(priority-1) (FCFS)->");
    for (int i = 0; i < n; i++) {
        printf("%s-", terminatedQueue[i]);
    }
    printf("\n");
}
```

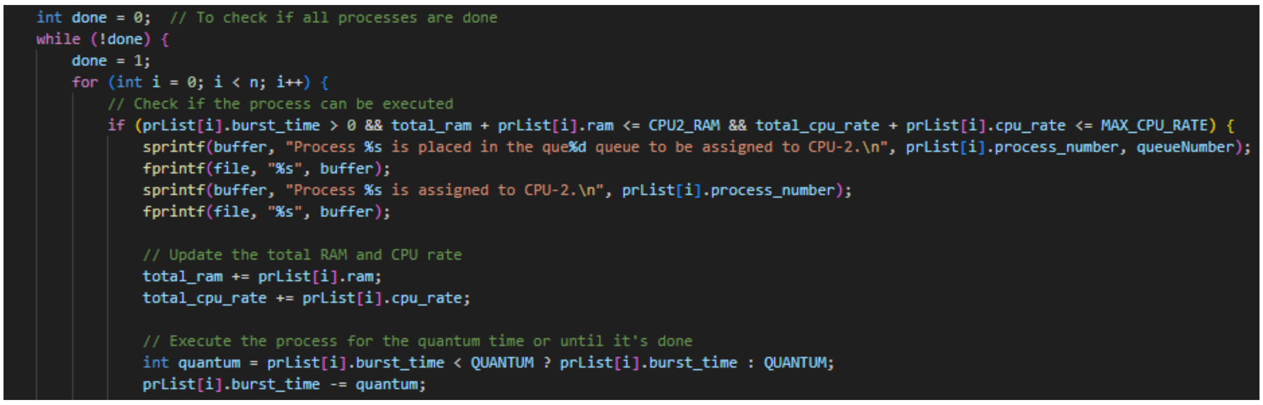
At the end the day, SJF assigns processes to CPU-2.

c. Round Robin Algorithm

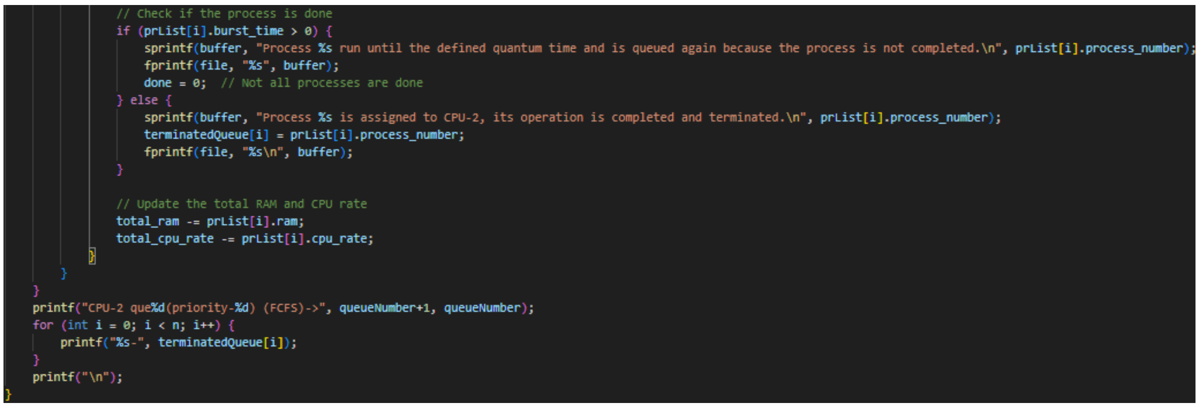
It takes an array of ProcessInfo structures, the number of processes, the maximum RAM and CPU rate for CPU2, a file pointer for logging, and a quantum time. The function first determines the queue number based on the quantum time. With this approach I've saved myself trouble of writing this function again for the other quantum times. Then, it enters a loop that continues until all processes are done. In each iteration of the loop, it checks each process to see if it can be executed based on its burst time and the available RAM and CPU rate. If a process can be executed, it is assigned to CPU2, its resource usage is added to the total, and a log message is written to the file. The process is then executed for the quantum time or until it's done. If the process is not done after the quantum time, it is queued again, and the loop will continue. If the process is done, it is added to the terminatedQueue and a log message is written to the file. After each process execution, its resource usage is subtracted from the total. Finally, the function prints the list of processes that were assigned to CPU2 and terminated.



```c
void rr_algorithm(ProcessInfo *prList, int n, int CPU2_RAM, int MAX_CPU_RATE, FILE *file,int QUANTUM) {
    int total_ram = 0;
    int total_cpu_rate = 0;
    int queueNumber = 0;

    if (QUANTUM == 8) {
        queueNumber = 2;
    } else if (QUANTUM == 16) {
        queueNumber = 3;
    }else{
        printf("Invalid quantum number\n");
        exit(1);
    }

    char buffer[256];
    char* terminatedQueue[n];
```



```c
    int done = 0;  // To check if all processes are done
    while (!done) {
        done = 1;
        for (int i = 0; i < n; i++) {
            // Check if the process can be executed
            if (prList[i].burst_time > 0 && total_ram + prList[i].ram <= CPU2_RAM && total_cpu_rate + prList[i].cpu_rate <= MAX_CPU_RATE) {
                sprintf(buffer, "Process %s is placed in the que%d queue to be assigned to CPU-2.\n", prList[i].process_number, queueNumber);
                fprintf(file, "%s", buffer);
                sprintf(buffer, "Process %s is assigned to CPU-2.\n", prList[i].process_number);
                fprintf(file, "%s", buffer);

                // Update the total RAM and CPU rate
                total_ram += prList[i].ram;
                total_cpu_rate += prList[i].cpu_rate;

                // Execute the process for the quantum time or until it's done
                int quantum = prList[i].burst_time < QUANTUM ? prList[i].burst_time : QUANTUM;
                prList[i].burst_time -= quantum;
```



```c
                // Check if the process is done
                if (prList[i].burst_time > 0) {
                    sprintf(buffer, "Process %s run until the defined quantum time and is queued again because the process is not completed.\n", prList[i].process_number);
                    fprintf(file, "%s", buffer);
                    done = 0;  // Not all processes are done
                } else {
                    sprintf(buffer, "Process %s is assigned to CPU-2, its operation is completed and terminated.\n", prList[i].process_number);
                    terminatedQueue[i] = prList[i].process_number;
                    fprintf(file, "%s\n", buffer);
                }

                // Update the total RAM and CPU rate
                total_ram -= prList[i].ram;
                total_cpu_rate -= prList[i].cpu_rate;
            }
        }
    }
    printf("CPU-2 que%d(priority-%d) (FCFS)->", queueNumber+1, queueNumber);
    for (int i = 0; i < n; i++) {
        printf("%s-", terminatedQueue[i]);
    }
    printf("\n");
}
```

- **Outputs**

  Its easy to implement fcfs and sjf algorithms because it doesn't require much computation unlike round robin. If a process didn't terminate in quantum time, it gets queued again and waits for the next time to terminate. These operations needs constant check. To show you how round robin works I will give an example. Consider this input:

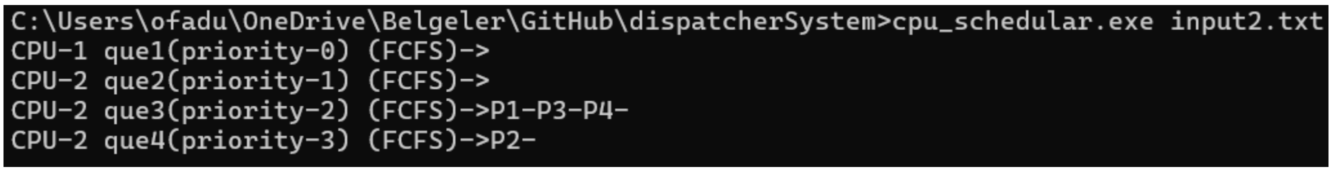
  P1,0, 2, 15, 574, 4

  P2,0, 3, 17, 574, 4

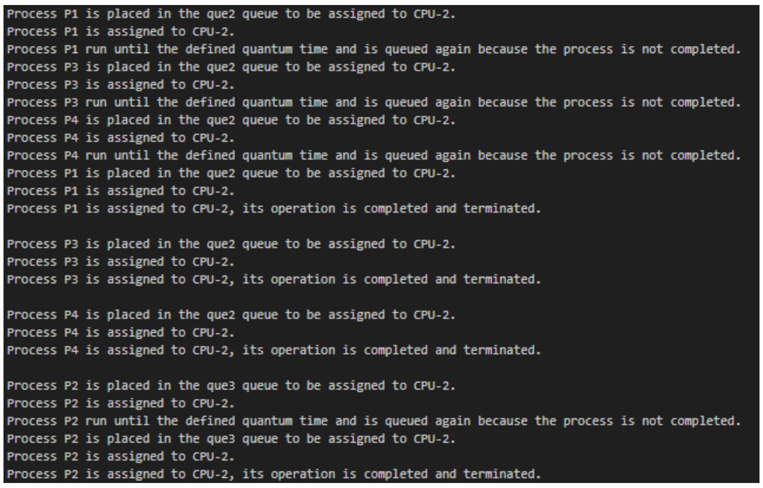
  P3,0, 2, 14, 574, 4

  P4,0, 2, 14, 574, 4

  Output of this input:



```
C:\Users\ofadu\OneDrive\Belgeler\GitHub\dispatcherSystem>cpu_schedular.exe input2.txt
CPU-1 que1(priority-0) (FCFS)->
CPU-2 que2(priority-1) (FCFS)->
CPU-2 que3(priority-2) (FCFS)->P1-P3-P4-
CPU-2 que4(priority-3) (FCFS)->P2-
```



```
Process P1 is placed in the que2 queue to be assigned to CPU-2.
Process P1 is assigned to CPU-2.
Process P1 run until the defined quantum time and is queued again because the process is not completed.
Process P3 is placed in the que2 queue to be assigned to CPU-2.
Process P3 is assigned to CPU-2.
Process P3 run until the defined quantum time and is queued again because the process is not completed.
Process P4 is placed in the que2 queue to be assigned to CPU-2.
Process P4 is assigned to CPU-2.
Process P4 run until the defined quantum time and is queued again because the process is not completed.
Process P1 is placed in the que2 queue to be assigned to CPU-2.
Process P1 is assigned to CPU-2.
Process P1 is assigned to CPU-2, its operation is completed and terminated.

Process P3 is placed in the que2 queue to be assigned to CPU-2.
Process P3 is assigned to CPU-2.
Process P3 is assigned to CPU-2, its operation is completed and terminated.

Process P4 is placed in the que2 queue to be assigned to CPU-2.
Process P4 is assigned to CPU-2.
Process P4 is assigned to CPU-2, its operation is completed and terminated.

Process P2 is placed in the que3 queue to be assigned to CPU-2.
Process P2 is assigned to CPU-2.
Process P2 run until the defined quantum time and is queued again because the process is not completed.
Process P2 is placed in the que3 queue to be assigned to CPU-2.
Process P2 is assigned to CPU-2.
Process P2 is assigned to CPU-2, its operation is completed and terminated.
```

As you can see it firstly starts processing P1, but quantum time is 8 so it doesn't have enough time to terminate it in one cycle, now its P3's turn but same thing happens here as well and then P4 goes through same events. When it's finally P1's turn it succefully terminates the process and others follow.

In conclusion, the dispatcher system developed in this project aims optimizing CPU resource management through the implementation of FCFS, SJF, and RR algorithms. The system adeptly handles process evaluation, ensuring that each task is allocated to the appropriate algorithm based on its priority, thereby maintaining a stable and efficient CPU working environment.