Write a program in C to implement Priority Queue & Round Robin

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct Process {
    int id;          // Process ID
    int priority;    // Process priority
    int burstTime;   // Burst time of the process
    struct Process* next;
} Process;

typedef struct PriorityQueue {
    Process* front;
} PriorityQueue;

// Function to create a new process
Process* createProcess(int id, int priority, int burstTime) {
    Process* newProcess = (Process*)malloc(sizeof(Process));
    newProcess->id = id;
    newProcess->priority = priority;
    newProcess->burstTime = burstTime;
    newProcess->next = NULL;
    return newProcess;
}

// Function to create a priority queue
PriorityQueue* createPriorityQueue() {
    PriorityQueue* pq = (PriorityQueue*)malloc(sizeof(PriorityQueue));
    pq->front = NULL;
    return pq;
}

// Function to insert a process into the priority queue
void insertProcess(PriorityQueue* pq, Process* newProcess) {
    if (pq->front == NULL || pq->front->priority > newProcess->priority) {
        newProcess->next = pq->front;
        pq->front = newProcess;
    } else {
        Process* current = pq->front;
        while (current->next != NULL && current->next->priority <= newProcess->priority) {
            current = current->next;
        }
        newProcess->next = current->next;
        current->next = newProcess;
    }
// Function to remove and return the highest priority process
Process* removeHighestPriorityProcess(PriorityQueue* pq) {
```

```
    if (pq->front == NULL) {
        return NULL;
    }
    Process* temp = pq->front;
    pq->front = pq->front->next;
    return temp;
}

// Function to implement Round Robin scheduling
void roundRobin(PriorityQueue* pq, int timeQuantum) {
    if (pq->front == NULL) {
        printf("No processes in the queue.\n");
        return;
    }

    Process* current = pq->front;
    while (current != NULL) {
        if (current->burstTime > timeQuantum) {
            printf("Process %d executed for %d units.\n", current->id, timeQuantum);
            current->burstTime -= timeQuantum;
            // Move to the end of the queue
            Process* temp = removeHighestPriorityProcess(pq);
            insertProcess(pq, temp);
        } else {
            printf("Process %d executed for %d units and finished.\n", current->id,
current->burstTime);
            current->burstTime = 0;  // Process is finished
            removeHighestPriorityProcess(pq); // Remove it from queue
        }
        current = pq->front; // Move to the next process in the queue
    }
}

int main() {
    PriorityQueue* pq = createPriorityQueue();
    int timeQuantum = 2; // Time quantum for Round Robin

    // Adding processes to the priority queue
    insertProcess(pq, createProcess(1, 1, 5)); // Process ID 1, Priority 1, Burst Time 5
    insertProcess(pq, createProcess(2, 3, 3)); // Process ID 2, Priority 3, Burst Time 3
    insertProcess(pq, createProcess(3, 2, 8)); // Process ID 3, Priority 2, Burst Time 8

    printf("Round Robin Scheduling:\n");
    roundRobin(pq, timeQuantum);

    // Free remaining processes in the queue
    while (pq->front != NULL) {
        Process* temp = removeHighestPriorityProcess(pq)      free(temp);
```

Write a C program to implement Dining Philosopher Problem

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_PHILOSOPHERS 5

pthread_mutex_t forks[NUM_PHILOSOPHERS];

void* philosopher(void* num) {
    int id = *(int*)num;
    while (1) {
        printf("Philosopher %d is thinking.\n", id);
        sleep(rand() % 3); // Simulate thinking time

        // Pick up the left fork
        pthread_mutex_lock(&forks[id]);

        // Pick up the right fork
        pthread_mutex_lock(&forks[(id + 1) % NUM_PHILOSOPHERS]);

        // Eating
        printf("Philosopher %d is eating.\n", id);
        sleep(rand() % 3); // Simulate eating time

        // Put down the right fork
        pthread_mutex_unlock(&forks[(id + 1) % NUM_PHILOSOPHERS]);

        // Put down the left fork
        pthread_mutex_unlock(&forks[id]);
    }
}

int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int philosopher_ids[NUM_PHILOSOPHERS];

    // Initialize mutexes for each fork
```

```
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_mutex_init(&forks[i], NULL);
    }

    // Create philosopher threads
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        philosopher_ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher,
&philosopher_ids[i]);
    }

    // Wait for philosopher threads to finish (they won't in this case)
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosophers[i], NULL);
    }

    // Clean up mutexes (this will not be reached in this example)
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_mutex_destroy(&forks[i]);
    }

    return 0;
}
```

Output run

```
gcc -pthread -o dining_philosophers dining_philosophers.c
./dining_philosophers
```

3.Write a C program to implement Producer-Consumer Problem

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>


#define BUFFER_SIZE 5

#define NUM_ITEMS 20
```

```
int buffer[BUFFER_SIZE];
int in = 0;  // Index for the next produced item
int out = 0; // Index for the next consumed item

sem_t empty; // Semaphore to count empty slots in the buffer
sem_t full;  // Semaphore to count full slots in the buffer
pthread_mutex_t mutex; // Mutex for mutual exclusion

void* producer(void* arg) {
    for (int i = 0; i < NUM_ITEMS; i++) {
        // Produce an item
        int item = rand() % 100;

        // Wait for an empty slot
        sem_wait(&empty);

        // Lock the buffer
        pthread_mutex_lock(&mutex);

        // Add the item to the buffer
        buffer[in] = item;
        printf("Producer produced: %d at index %d\n", item, in);
        in = (in + 1) % BUFFER_SIZE;

        // Unlock the buffer
        pthread_mutex_unlock(&mutex);
```

```c
        // Signal that a new item has been produced
        sem_post(&full);

        // Simulate production time
        sleep(rand() % 2);
    }
    return NULL;
}

void* consumer(void* arg) {
    for (int i = 0; i < NUM_ITEMS; i++) {
        // Wait for a full slot
        sem_wait(&full);

        // Lock the buffer
        pthread_mutex_lock(&mutex);

        // Remove the item from the buffer
        int item = buffer[out];
        printf("Consumer consumed: %d from index %d\n", item, out);
        out = (out + 1) % BUFFER_SIZE;

        // Unlock the buffer
        pthread_mutex_unlock(&mutex);

        // Signal that an item has been consumed
        sem_post(&empty);
```

```
        // Simulate consumption time
        sleep(rand() % 2);
    }
    return NULL;
}

int main() {
    pthread_t prod_thread, cons_thread;

    // Initialize semaphores and mutex
    sem_init(&empty, 0, BUFFER_SIZE); // Initially, the buffer is empty
    sem_init(&full, 0, 0);            // Initially, there are no full slots
    pthread_mutex_init(&mutex, NULL); // Initialize the mutex

    // Create producer and consumer threads
    pthread_create(&prod_thread, NULL, producer, NULL);
    pthread_create(&cons_thread, NULL, consumer, NULL);

    // Wait for the threads to finish
    pthread_join(prod_thread, NULL);
    pthread_join(cons_thread, NULL);

    // Clean up
    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);

    return 0;
```

```
}
```

Write a program in C to implement FCFS/SJF

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Process {
    int id;         // Process ID
    int burstTime;  // Burst time of the process
    int waitingTime; // Waiting time of the process
    int turnaroundTime; // Turnaround time of the process
} Process;

void calculateFCFS(Process processes[], int n) {
    int totalWaitingTime = 0;
    int totalTurnaroundTime = 0;

    processes[0].waitingTime = 0; // First process has no waiting time

    // Calculate waiting time and turnaround time for each process
    for (int i = 1; i < n; i++) {
        processes[i].waitingTime = processes[i - 1].waitingTime + processes[i - 1].burstTime;
    }

    for (int i = 0; i < n; i++) {
        processes[i].turnaroundTime = processes[i].waitingTime + processes[i].burstTime;
        totalWaitingTime += processes[i].waitingTime;
```

```
        totalTurnaroundTime += processes[i].turnaroundTime;
    }

    printf("\nFCFS Scheduling:\n");
    printf("Process ID\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burstTime,
processes[i].waitingTime, processes[i].turnaroundTime);
    }
    printf("Average Waiting Time: %.2f\n", (float)totalWaitingTime / n);
    printf("Average Turnaround Time: %.2f\n", (float)totalTurnaroundTime /
n);
}

void calculateSJF(Process processes[], int n) {
    // Sort processes based on burst time (SJF)
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].burstTime > processes[j + 1].burstTime) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }

    int totalWaitingTime = 0;
    int totalTurnaroundTime = 0;
```

```
    processes[0].waitingTime = 0; // First process has no waiting time

    // Calculate waiting time and turnaround time for each process
    for (int i = 1; i < n; i++) {
        processes[i].waitingTime = processes[i - 1].waitingTime + processes[i
- 1].burstTime;
    }

    for (int i = 0; i < n; i++) {
        processes[i].turnaroundTime = processes[i].waitingTime +
processes[i].burstTime;
        totalWaitingTime += processes[i].waitingTime;
        totalTurnaroundTime += processes[i].turnaroundTime;
    }

    printf("\nSJF Scheduling:\n");
    printf("Process ID\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burstTime,
processes[i].waitingTime, processes[i].turnaroundTime);
    }
    printf("Average Waiting Time: %.2f\n", (float)totalWaitingTime / n);
    printf("Average Turnaround Time: %.2f\n", (float)totalTurnaroundTime /
n);
}

int main() {
```

```c
    int n;

    printf("Enter the number of processes: ");
    scanf("%d", &n);

    Process* processes = (Process*)malloc(n * sizeof(Process));

    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1; // Process IDs start from 1
        printf("Enter burst time for process %d: ", processes[i].id);
        scanf("%d", &processes[i].burstTime);
    }

    calculateFCFS(processes, n);
    calculateSJF(processes, n);

    free(processes);
    return 0;
}
```

Here are shell scripts for each of the tasks you requested:

1. Check Whether a Given Number is Even or Not

```bash
read -p "Enter a number: " number

if (( number % 2 == 0 )); then
    echo "$number is even."
else
```

```
    echo "$number is odd."
```

2. Check Whether a Year is a Leap Year or Not

```
#!/bin/bash

read -p "Enter a year: " year

if (( (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0) )); then
    echo "$year is a leap year."
else
    echo "$year is not a leap year."
fi
```

3. Find the Factorial of a Given Number

```
#!/bin/bash

read -p "Enter a number: " number

factorial=1

for (( i=1; i<=number; i++ )); do
    factorial=$((factorial * i))
done

echo "Factorial of $number is $factorial."
```

4. Swap Two Integer Values

```
#!/bin/bash
```

```bash
read -p "Enter first integer: " a
read -p "Enter second integer: " b

echo "Before swapping: a = $a, b = $b"

# Swapping
temp=$a
a=$b
b=$temp
echo "After swapping: a = $a, b = $b"
```

1. Simulating the `cp` Command

```c
#include <stdio.h>
#include <stdlib.h>
void copyFile(const char *source, const char *destination) {
    FILE *srcFile = fopen(source, "rb");
    if (srcFile == NULL) {
        perror("Error opening source file");
        exit(EXIT_FAILURE);
    }
    FILE *destFile = fopen(destination, "wb");
    if (destFile == NULL) {
        perror("Error opening destination file");
        fclose(srcFile);
        exit(EXIT_FAILURE)
    char buffer[1024];
    size_t bytesRead;
    // Copying the file
    while ((bytesRead = fread(buffer, 1, sizeof(buffer), srcFile)) > 0) {
```

```
        fwrite(buffer, 1, bytesRead, destFile);
    }
    fclose(srcFile);
    fclose(destFile);
    printf("File copied from %s to %s\n", source, destination);
}
int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <source_file> <destination_file>\n", argv[0]);
        return EXIT_FAILURE;
    }
    copyFile(argv[1], argv[2]);
    return EXIT_SUCCESS;
}
```

Simulating the `grep` Command

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void grepFile(const char *filename, const char *pattern) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        exit(EXIT_FAILURE)
    char line[1024];
    int lineNumber = 0;
    int found = 0;
    while (fgets(line, sizeof(line), file)) {
```

```
        lineNumber++;
        if (strstr(line, pattern) != NULL) {
            printf("%d: %s", lineNumber, line);
            found = 1;
        }    if (!found) {
        printf("No matches found for pattern '%s' in file '%s'.\n", pattern,
filename);
    }
    fclose(file);
int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <filename> <pattern>\n", argv[0]);
        return EXIT_FAILURE
    grepFile(argv[1], argv[2]);
    return EXIT_SUCCESS;
}
```