```python
import threading
import time

def task1():
    for i in range(5):
        print(f"Task 1: {i}")
        time.sleep(1)

def task2():
    for i in range(5):
        print(f"Task 2: {i}")
        time.sleep(1)

t1 = threading.Thread(target=task1)
t2 = threading.Thread(target=task2)

t1.start()
t2.start()

t1.join()
t2.join()
```

```python
import threading
import time

def download_data():
    print("Downloading data...")
    time.sleep(3)  # Simulating download delay
    print("Download complete.")

def process_data():
    print("Processing data...")
    time.sleep(2)  # Simulating processing delay
    print("Processing complete.")

download_thread =
threading.Thread(target=download_data)
process_thread = threading.Thread(target=process_data)

download_thread.start()
time.sleep(1)
process_thread.start()

download_thread.join()
process_thread.join()

print("Both tasks (download and process) are completed.")
```

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

int main() {
    int pipefd[2];
    pid_t pid;
    char buffer[100];

    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(1);
    }

    pid = fork();

    if (pid < 0) {
        perror("fork");
        exit(1);
    } else if (pid == 0) {
        close(pipefd[0]);
        const char message[] = "Hello from the child process!";
        write(pipefd[1], message, strlen(message) + 1);
        close(pipefd[1]);
        exit(0);
    } else {
        close(pipefd[1]);
        read(pipefd[0], buffer, sizeof(buffer));
        printf("Parent received: %s\n", buffer);
        close(pipefd[0]);
    }

    return 0;
}
```

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

sem_t semaphore;

void* task(void* arg) {
    sem_wait(&semaphore);
    printf("%s acquired semaphore.\n", (char*)arg);
    sleep(2);
    printf("%s releasing semaphore.\n", (char*)arg);
    free(arg);
    sem_post(&semaphore);
    return NULL;
}

int main() {
    pthread_t threads[4];
    sem_init(&semaphore, 0, 2);

    for (int i = 0; i < 4; i++) {
        char* name = malloc(20);
        if (name == NULL) {
            perror("malloc");
            exit(1);
        }
        sprintf(name, "Thread-%d", i + 1);
        if (pthread_create(&threads[i], NULL, task, name) != 0) {
            perror("pthread_create");
            exit(1);
        }
    }

    for (int i = 0; i < 4; i++) {
        pthread_join(threads[i], NULL);
    }

    sem_destroy(&semaphore);
    return 0;
}
```

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

int main() {
    int fd[2];
    char buffer[128];

    if (pipe(fd) == -1) {
        perror("pipe");
        exit(1);
    }

    pid_t pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(1);
    }

    if (pid == 0) {
        close(fd[1]);
        read(fd[0], buffer, sizeof(buffer));
        printf("Child received: %s\n", buffer);
        close(fd[0]);
    } else {
        close(fd[0]);
        const char *message = "Hello from the parent process!";
        write(fd[1], message, strlen(message) + 1);
        close(fd[1]);
    }

    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>

#define SHM_NAME "/my_shm"
#define SIZE 4096
int main() {
    int shm_fd;
    void *ptr;

    shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);

    if (shm_fd < 0) {
        perror("shm_open");
        exit(1);
    }

    if (ftruncate(shm_fd, SIZE) == -1) {
        perror("ftruncate");
        exit(1);
    }

    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (ptr == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }

    sprintf(ptr, "Hello from shared memory!");

    if (munmap(ptr, SIZE) == -1) {
        perror("munmap");
        exit(1);
    }

    if (shm_unlink(SHM_NAME) == -1) {
        perror("shm_unlink");
        exit(1);
    }

    return 0;
}
```

```
import pandas as pd #sjf
import matplotlib.pyplot as plt

def sjf(processes, burst_time, arrival_time):
    n = len(processes)
    waiting_time = [0] * n
    turn_around_time = [0] * n
    completion_time = [0] * n
    gantt_chart = []
    time = 0
    temp = sorted([(arrival_time[i], burst_time[i], processes[i]) for i in range(n)], key=lambda x: x[0])

    while temp:
        ready_queue = [p for p in temp if p[0] <= time]
        if ready_queue:
            ready_queue.sort(key=lambda x: x[1])
            process = ready_queue.pop(0)
            gantt_chart.append(process[2])
            time += process[1]
            temp.remove(process)
            completion_time[processes.index(process[2])] = time
        else:
            time += 1

    for i in range(n):
        turn_around_time[i] = completion_time[i] - arrival_time[i]
        waiting_time[i] = turn_around_time[i] - burst_time[i]

    avg_waiting_time = sum(waiting_time) / n
    avg_turn_around_time = sum(turn_around_time) / n
    throughput = n / (max(completion_time) - min(arrival_time))

    df = pd.DataFrame({
        'ProcessID': processes,
        'Arrival Time': arrival_time,
        'Burst Time': burst_time,
        'Completion Time': completion_time,
        'Turnaround Time': turn_around_time,
        'Waiting Time': waiting_time,
        'Response Time': waiting_time
    })

    print(df)
    print(f"Average Waiting Time: {avg_waiting_time}")
    print(f"Average Turnaround Time: {avg_turn_around_time}")
    print(f"Throughput: {throughput}")

    plt.figure(figsize=(10, 6))
    plt.title('Shortest Job First Scheduling')
    plt.barh(processes, gantt_chart)
    plt.xlabel('Time')
    plt.show()

n = int(input("Enter the number of processes: "))
processes = []
arrival_time = []
burst_time = []

for i in range(n):
    processes.append(f"P{i+1}")
    arrival_time.append(int(input(f"Enter arrival time for process P{i+1}: ")))
    burst_time.append(int(input(f"Enter burst time for process P{i+1}: ")))

sjf(processes, burst_time, arrival_time)
```

```
def LRU(pages, capacity):
    memory = []
    page_faults = 0
    for page in pages:
        if page not in memory:
            if len(memory) < capacity:
                memory.append(page)
            else:
                memory.remove(memory[0])
                memory.append(page)
            page_faults += 1
        else:
            memory.remove(page)
            memory.append(page)
    return page_faults

pages = list(map(int, input("Enter the sequence of pages: ").split()))
capacity = int(input("Enter the capacity of memory: "))
page_faults = LRU(pages, capacity)
print(f"Page Replacement: LRU")
print(f"Total Page Faults: {page_faults}")
```

```
def FIFO(pages, capacity):
    memory = []
    page_faults = 0
    for page in pages:
        if page not in memory:
            if len(memory) < capacity:
                memory.append(page)
            else:
                memory.pop(0)
                memory.append(page)
            page_faults += 1
    return page_faults

pages = list(map(int, input("Enter the sequence of pages: ").split()))
capacity = int(input("Enter the capacity of memory: "))
page_faults = FIFO(pages, capacity)
print(f"Page Replacement: FIFO")
print(f"Total Page Faults: {page_faults}")
```

```
import matplotlib.pyplot as plt #fcfs
def calculate_times(processes):
    n = len(processes)
    completion_time = [0] * n
    turnaround_time = [0] * n
    waiting_time = [0] * n
    response_time = [0] * n
    completion_time[0] = processes[0][1] + processes[0][2]
    for i in range(1, n):
        completion_time[i] = max(completion_time[i - 1], processes[i][1]) + processes[i][2]
        turnaround_time[i] = completion_time[i] - processes[i][1]
        waiting_time[i] = turnaround_time[i] - processes[i][2]
        response_time[i] = waiting_time[i]
    avg_waiting_time = sum(waiting_time) / n
    avg_turnaround_time = sum(turnaround_time) / n
    throughput = n / completion_time[-1]
    return (completion_time, turnaround_time, waiting_time, response_time,
            avg_waiting_time, avg_turnaround_time, throughput)
def plot_gantt_chart(processes, completion_time):
    fig, gnt = plt.subplots()
    for i, (pid, arrival_time, burst_time) in enumerate(processes):
        start_time = completion_time[i] - burst_time
        gnt.broken_barh([(start_time, burst_time)], (10 * i, 9), edgecolor='black', color='skyblue')
    plt.grid(True)
    plt.show()

def main():
    print("Enter number of processes:")
    n = int(input().strip())
    processes = []
    print("Enter Process ID, Arrival Time, Burst Time:")
    for _ in range(n):
        pid, arrival_time, burst_time = input().split()
        processes.append((int(pid), int(arrival_time), int(burst_time)))
    processes.sort(key=lambda x: x[1])
    (completion_time, turnaround_time, waiting_time, response_time,
     avg_waiting_time, avg_turnaround_time, throughput) = calculate_times(processes)

    print("Process\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting Time\tResponse Time")
    for i in range(n):
        print(f"{processes[i][0]}\t{processes[i][1]}\t\t{processes[i][2]}
                        \t\t{completion_time[i]}\t\t{turnaround_time[i]}\t\t{waiting_time[i]}\t\t{response_time[i]}")
    print(f"\nAverage Waiting Time: {avg_waiting_time}")
    print(f"Average Turnaround Time: {avg_turnaround_time}")
    print(f"Throughput: {throughput}")
    plot_gantt_chart(processes, completion_time)

if __name__ == "__main__":
    main()
```

```
import pandas as pd #rr
import matplotlib.pyplot as plt

def round_robin(processes, burst_time, arrival_time, quantum):
    n = len(processes)
    waiting_time = [0] * n
    turn_around_time = [0] * n
    completion_time = [0] * n
    remaining_burst = burst_time.copy()
    gantt_chart = []
    time = 0
    queue = []

    while True:
        done = True
        for i in range(n):
            if remaining_burst[i] > 0:
                done = False
                if remaining_burst[i] > quantum:
                    time += quantum
                    remaining_burst[i] -= quantum
                    gantt_chart.append(processes[i])
                else:
                    time += remaining_burst[i]
                    waiting_time[i] = time - arrival_time[i] - burst_time[i]
                    remaining_burst[i] = 0
                    gantt_chart.append(processes[i])
        if done:
            break

    for i in range(n):
        turn_around_time[i] = burst_time[i] + waiting_time[i]

    completion_time = [waiting_time[i] + burst_time[i] + arrival_time[i] for i in range(n)]

    avg_waiting_time = sum(waiting_time) / n
    avg_turn_around_time = sum(turn_around_time) / n
    throughput = n / (max(completion_time) - min(arrival_time))

    df = pd.DataFrame({
        'ProcessID': processes,
        'Arrival Time': arrival_time,
        'Burst Time': burst_time,
        'Completion Time': completion_time,
        'Turnaround Time': turn_around_time,
        'Waiting Time': waiting_time,
        'Response Time': waiting_time
    })

    print(df)
    print(f"Average Waiting Time: {avg_waiting_time}")
    print(f"Average Turnaround Time: {avg_turn_around_time}")
    print(f"Throughput: {throughput}")

    plt.figure(figsize=(10, 6))
    plt.title('Round Robin Scheduling')
    plt.barh(processes, gantt_chart)
    plt.xlabel('Time')
    plt.show()


n = int(input("Enter the number of processes: "))
processes = []
arrival_time = []
burst_time = []

for i in range(n):
    processes.append(f"P{i+1}")
    arrival_time.append(int(input(f"Enter arrival time for process P{i+1}: ")))
    burst_time.append(int(input(f"Enter burst time for process P{i+1}: ")))

quantum = int(input("Enter the time quantum: "))

round_robin(processes, burst_time, arrival_time, quantum)
```