```python
class ContiguousAllocation:
    def __init__(self, total_blocks):
        self.total_blocks = total_blocks
        self.blocks = [None] * total_blocks

    def allocate(self, file_name, size):
        for i in range(self.total_blocks - size + 1):
            if all(self.blocks[i + j] is None for j in range(size)):
                for j in range(size):
                    self.blocks[i + j] = file_name
                print(f"File '{file_name}' allocated at blocks {i} to {i + size - 1}")
                return
        print(f"Not enough space to allocate file '{file_name}'")

    def deallocate(self, file_name):
        for i in range(self.total_blocks):
            if self.blocks[i] == file_name:
                self.blocks[i] = None
        print(f"File '{file_name}' deallocated.")

    def display(self):
        print("Disk blocks:", self.blocks)


contiguous = ContiguousAllocation(10)
contiguous.allocate("File1", 3)
contiguous.allocate("File2", 4)
contiguous.deallocate("File1")
contiguous.display()
```

```python
import socket
import signal
import os

def handle_client(client_socket):
    data = client_socket.recv(1024)
    if data:
        client_socket.send(data)
    client_socket.close()

def server():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(('localhost', 8080))
    server_socket.listen(5)

    signal.signal(signal.SIGINT, lambda signum, frame: os._exit(0))

    while True:
        client_socket, addr = server_socket.accept()
        handle_client(client_socket)

if __name__ == '__main__':
    server()
```

```python
class LinkedAllocation:
    def __init__(self, total_blocks):
        self.total_blocks = total_blocks
        self.blocks = [None] * total_blocks

    def allocate(self, file_name, size):
        prev_block = None
        for i in range(self.total_blocks):
            if self.blocks[i] is None:
                if prev_block is None:
                    self.blocks[i] = [file_name, None]
                else:
                    self.blocks[prev_block][1] = i
                    self.blocks[i] = [file_name, None]
                prev_block = i
                size -= 1
                if size == 0:
                    print(f"File '{file_name}' allocated.")
                    return
        print(f"Not enough space to allocate file '{file_name}'")

    def deallocate(self, file_name):
        for i in range(self.total_blocks):
            if self.blocks[i] and self.blocks[i][0] == file_name:
                next_block = self.blocks[i][1]
                while next_block is not None:
                    self.blocks[i] = None
                    i = next_block
                    next_block = self.blocks[i][1]
                self.blocks[i] = None
        print(f"File '{file_name}' deallocated.")

    def display(self):
        print("Disk blocks:", self.blocks)


linked = LinkedAllocation(10)
linked.allocate("File1", 3)
linked.allocate("File2", 2)
linked.deallocate("File1")
linked.display()
```

```python
import socket

def client():
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect(('localhost', 8080))

    client_socket.send(b'Hello, Server!')
    data = client_socket.recv(1024)
    print(f'Received from server: {data.decode()}')
    client_socket.close()

if __name__ == '__main__':
    client()
```

```python
class IndexedAllocation:
    def __init__(self, total_blocks):
        self.total_blocks = total_blocks
        self.blocks = [None] * total_blocks
        self.index_blocks = {}

    def allocate(self, file_name, size):
        index_block = []
        for i in range(self.total_blocks):
            if self.blocks[i] is None:
                index_block.append(i)
                if len(index_block) == size:
                    self.index_blocks[file_name] = index_block
                    for block in index_block:
                        self.blocks[block] = file_name
                    print(f"File '{file_name}' allocated with index block {index_block}.")
                    return
        print(f"Not enough space to allocate file '{file_name}'")

    def deallocate(self, file_name):
        if file_name in self.index_blocks:
            index_block = self.index_blocks[file_name]
            for block in index_block:
                self.blocks[block] = None
            del self.index_blocks[file_name]
            print(f"File '{file_name}' deallocated.")
        else:
            print(f"File '{file_name}' not found.")

    def display(self):
        print("Disk blocks:", self.blocks)
        print("Index blocks:", self.index_blocks)


indexed = IndexedAllocation(10)
indexed.allocate("File1", 3)
indexed.allocate("File2", 4)
indexed.deallocate("File1")
indexed.display()
```

```python
class BankersAlgorithm:
  def __init__(self, num_processes, num_resources):
    self.num_processes = num_processes
    self.num_resources = num_resources
    self.available = [0] * num_resources
    self.maximum = [[0] * num_resources for _ in range(num_processes)]
    self.allocation = [[0] * num_resources for _ in range(num_processes)]
    self.need = [[0] * num_resources for _ in range(num_processes)]

  def input_resources(self):
    print("\nEnter the available instances of each resource:")
    for i in range(self.num_resources):
      self.available[i] = int(input(f"Resource {i}: "))

    print("\nEnter the maximum demand of each process for each resource:")
    for i in range(self.num_processes):
      print(f"Process {i}:")
      for j in range(self.num_resources):
        self.maximum[i][j] = int(input(f"Maximum demand for Resource {j}: "))

    print("\nEnter the current allocation of resources to each process:")
    for i in range(self.num_processes):
      print(f"Process {i}:")
      for j in range(self.num_resources):
        self.allocation[i][j] = int(input(f"Allocated for Resource {j}: "))

    for i in range(self.num_processes):
      for j in range(self.num_resources):
        self.need[i][j] = self.maximum[i][j] - self.allocation[i][j]

  def is_safe(self):
    work = self.available[:]
    finish = [False] * self.num_processes
    safe_sequence = []

    while len(safe_sequence) < self.num_processes:
      progress_made = False
      for i in range(self.num_processes):
        if not finish[i] and all(self.need[i][j] <= work[j] for j in range(self.num_resources)):
          for j in range(self.num_resources):
            work[j] += self.allocation[i][j]
          safe_sequence.append(i)
          finish[i] = True
          progress_made = True
          break
      if not progress_made:
        return False, []

    return True, safe_sequence
```

```python
def request_resources(self, process_id, request):
    for i in range(self.num_resources):
      if request[i] > self.need[process_id][i]:
        print(f"Error: Process {process_id} has exceeded its maximum claim.")
        return False

    for i in range(self.num_resources):
      if request[i] > self.available[i]:
        print(f"Resources not available for process {process_id}.")
        return False

    for i in range(self.num_resources):
      self.available[i] -= request[i]
      self.allocation[process_id][i] += request[i]
      self.need[process_id][i] -= request[i]

    is_safe, safe_sequence = self.is_safe()

    if is_safe:
      print(f"Resources allocated to process {process_id}. Safe sequence: {safe_sequence}")
      return True
    else:
      for i in range(self.num_resources):
        self.available[i] += request[i]
        self.allocation[process_id][i] -= request[i]
        self.need[process_id][i] += request[i]
      print("System is not in a safe state. Request denied.")
      return False


def main():
  num_processes = int(input("Enter the number of processes: "))
  num_resources = int(input("Enter the number of resource types: "))

  bankers_algo = BankersAlgorithm(num_processes, num_resources)
  bankers_algo.input_resources()

  process_id = int(input("\nEnter the process ID that is requesting resources: "))
  request = []
  print("Enter the request for resources (one number per resource type):")
  for i in range(num_resources):
    request.append(int(input(f"Resource {i}: ")))

  bankers_algo.request_resources(process_id, request)


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

```python
import threading
import time
import random

class Philosopher(threading.Thread):
  def __init__(self, id, left_fork, right_fork):
    threading.Thread.__init__(self)
    self.id = id
    self.left_fork = left_fork
    self.right_fork = right_fork

  def run(self):
    while True:
      self.think()
      self.dine()

  def think(self):
    print(f"Philosopher {self.id} is thinking.")
    time.sleep(random.uniform(1, 3))

  def dine(self):
    self.pick_up_forks()
    print(f"Philosopher {self.id} is eating.")
    time.sleep(random.uniform(1, 2))
    self.put_down_forks()

  def pick_up_forks(self):
    print(f"Philosopher {self.id} is trying to pick up forks.")
    self.left_fork.acquire()
    print(f"Philosopher {self.id} picked up left fork.")
    self.right_fork.acquire()
    print(f"Philosopher {self.id} picked up right fork.")

  def put_down_forks(self):
    self.left_fork.release()
    print(f"Philosopher {self.id} put down left fork.")
    self.right_fork.release()
    print(f"Philosopher {self.id} put down right fork.")

def main():
  num_philosophers = 5
  forks = [threading.Lock() for _ in range(num_philosophers)]
  philosophers = []
  for i in range(num_philosophers):
    left_fork = forks[i]
    right_fork = forks[(i + 1) % num_philosophers]
    philosopher = Philosopher(i, left_fork, right_fork)
    philosophers.append(philosopher)
  for philosopher in philosophers:
    philosopher.start()
  time.sleep(10)
  for philosopher in philosophers:
    philosopher.join()

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

```python
import threading
import time
import random

class Buffer:
    def __init__(self, size):
        self.size = size
        self.buffer = []
        self.lock = threading.Lock()
        self.empty = threading.Condition(self.lock)
        self.full = threading.Condition(self.lock)

    def add(self, item):
        with self.lock:
            while len(self.buffer) == self.size:
                self.empty.wait()
            self.buffer.append(item)
            print(f"Produced: {item}")
            self.full.notify()

    def remove(self):
        with self.lock:
            while len(self.buffer) == 0:
                self.full.wait()
            item = self.buffer.pop(0)
            print(f"Consumed: {item}")
            self.empty.notify()
            return item

class Producer(threading.Thread):
    def __init__(self, buffer):
        threading.Thread.__init__(self)
        self.buffer = buffer

    def run(self):
        while True:
            item = random.randint(1, 100)
            self.buffer.add(item)
            time.sleep(random.uniform(0.5, 1.5))

class Consumer(threading.Thread):
    def __init__(self, buffer):
        threading.Thread.__init__(self)
        self.buffer = buffer

    def run(self):
        while True:
            item = self.buffer.remove()
            time.sleep(random.uniform(1, 2))

def main():
    buffer_size = 5
    buffer = Buffer(buffer_size)

    producer = Producer(buffer)
    consumer = Consumer(buffer)

    producer.start()
    consumer.start()

    producer.join()
    consumer.join()

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

```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;
}
```

```python
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)
```

```python
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}")
```

```python
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}")
```

```python
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()
```

```python
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()
```

```python
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)
```