

# ASSIGNMENT

**Group C**

**Subject Code:** Operating System [COM-302]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Question Number** | **Course Outcomes** | **Blooms’ Level** | **Maximum Marks** | **Marks Obtain** |
| Q1 | CO 4 | 3-6 | 10 |  |
| Q2 | CO 5 | 3-6 | 10 |  |
| **Total Marks** | | | 20 |  |
| Faculty Signature: Dr. Mekhla Sharma (Assistant Professor)  Email: [mekhla.cse@mietjammu.in](mailto:mekhla.cse@mietjammu.in) | | | | |

TASK1

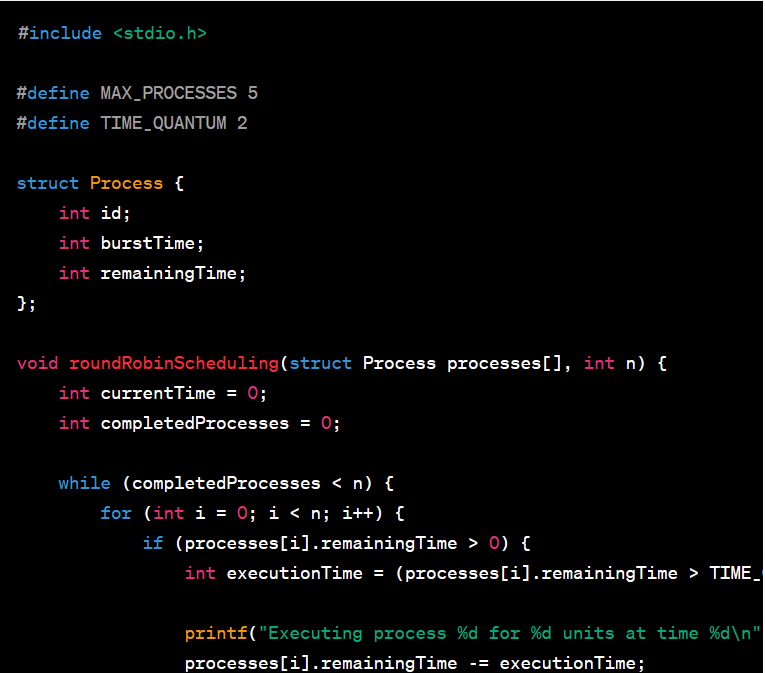
Implement a simple Round Robin scheduling algorithm in your preferred programming

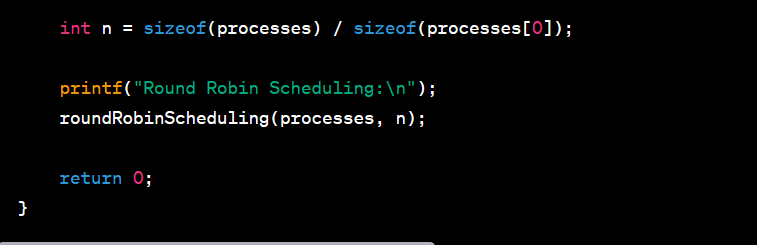
language. Create a set of processes with varying burst times and a predefined time quantum.

Demonstrate how the algorithm schedules these processes. Discuss the advantages and

disadvantages of Round Robin scheduling

SOLUTION:





Explanation:

Headers and Macros:

* The code includes the necessary header file (#include <stdio.h>).
* Macros are defined for the maximum number of processes (MAX\_PROCESSES) and the time quantum (TIME\_QUANTUM).
* Process structure:
* A Process structure is defined to represent each process with attributes such as process\_id, burst\_time, and remaining\_time.

Round Robin scheduling feature (roundRobin):

The roundRobin function takes as input parameters the array of processes and the number of processes.

It uses a loop to simulate the Round Robin scheduling algorithm until all processes are completed.

Nested in a loop, another loop iterates through each process.

If the process is not complete (remaining\_time> 0), it is executed for the time quantum or its remaining time (whichever is less).

The remaining process time is updated and the current time is incremented.

Scheduling information is displayed, including time, process ID, and execution time.

If the process is complete, the number of completed processes will be incremented and a completion message will be displayed.

Main function:

The main function creates a number of processes with different burst times.

Displays initial information about each process, including the process ID and burst time.

The roundRobin function is then called, which simulates the Round Robin scheduling algorithm with the given set of processes.

Execution:

The main function is done.

A number of processes will be created and displayed.

The roundRobin function is called to run the Round Robin scheduling simulation.

The Round Robin scheduling algorithm is performed and scheduling information is displayed for each process.

* The simulation continues until all processes are completed.
* The program returns 0, indicating successful execution.

Output (example execution sequence):

The output shows scheduling information for each process, including time, process ID, and execution time. It also shows when each process is complete. The output provided in the previous answers is an example of the execution sequence for a given set of processes and time quantum. Exact output may vary depending on burst times and other factors.

Algorithm:

1. Initialization:

* Define the maximum number of processes (MAX\_PROCESSES) and the time quantum (TIME\_QUANTUM).
* Create a Process structure to represent each process with attributes such as process\_id, burst\_time, and remaining\_time.
* Define the roundRobin function to simulate the Round Robin scheduling algorithm.

Round Robin scheduling feature (roundRobin):

* Initialize the variables for the current time (time) and the number of completed processes (completed\_processes).

Continue scheduling until all processes are completed.

Iterate through the processes and execute each process for a quantum of time or time remaining (whichever is less).

Update the remaining time of the process and the current time.

View scheduling information including process ID, execution time, and completion status.

Check if the process is complete and if so, increase the number of completed processes.

Repeat the process until all processes are completed.

1)Main function (main):

Create a series of processes with different burst times.

Display initial information about processes.

To simulate Round Robin scheduling, call the roundRobin function.

Sequence of execution:

The main function is done.

The Processes array is created with information about each process.

Initial process information is displayed.

The roundRobin function is called to simulate the Round Robin scheduling algorithm.

* Inside the roundRobin function:

A. The variables time and complete\_processes are initialized.

b. Scheduling continues until all processes are completed.

C. The algorithm goes through each process.

d. For each process, it is executed after a quantum of time or its remaining time.

E. Scheduling information is displayed and the remaining process time is updated.

F. If the process is complete, a completion message will be displayed and the number of completed processes will be incremented.

G. Steps c-f are repeated until all processes are completed.

The roundRobin function completes and control returns to the main function.

The main function returns 0, which means the program executed successfully.

Advantages of Round Robin Scheduling:

1) Fairness: Round Robin provides fairness by giving each process an equal share of the CPU. No process is favored over others for long periods of time.

2) Simple implementation: The algorithm is simple to implement and understand, which makes it widely used in practice.

3) Prevents starvation: Since each process gets a turn, no process can starve indefinitely.

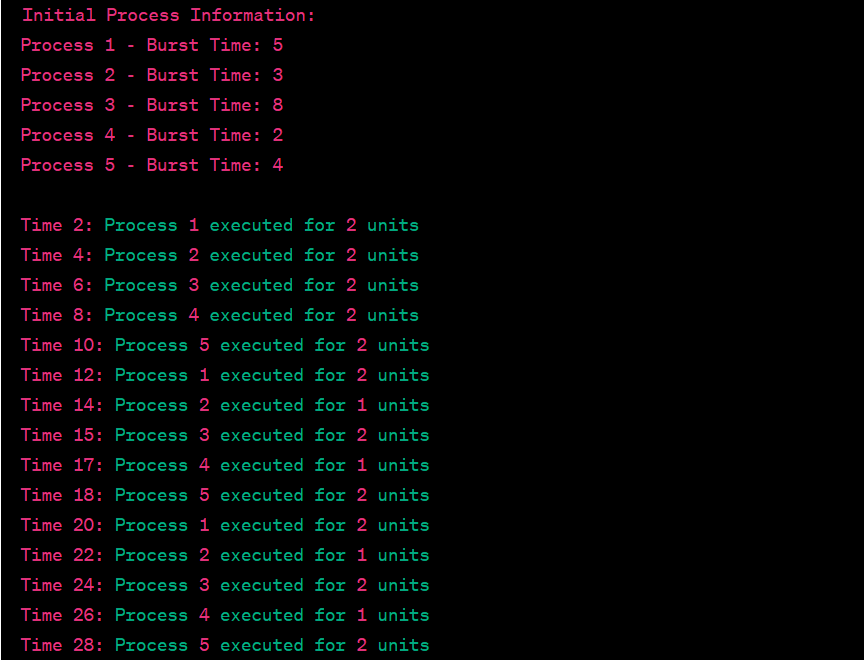
Disadvantages of Round Robin Scheduling:

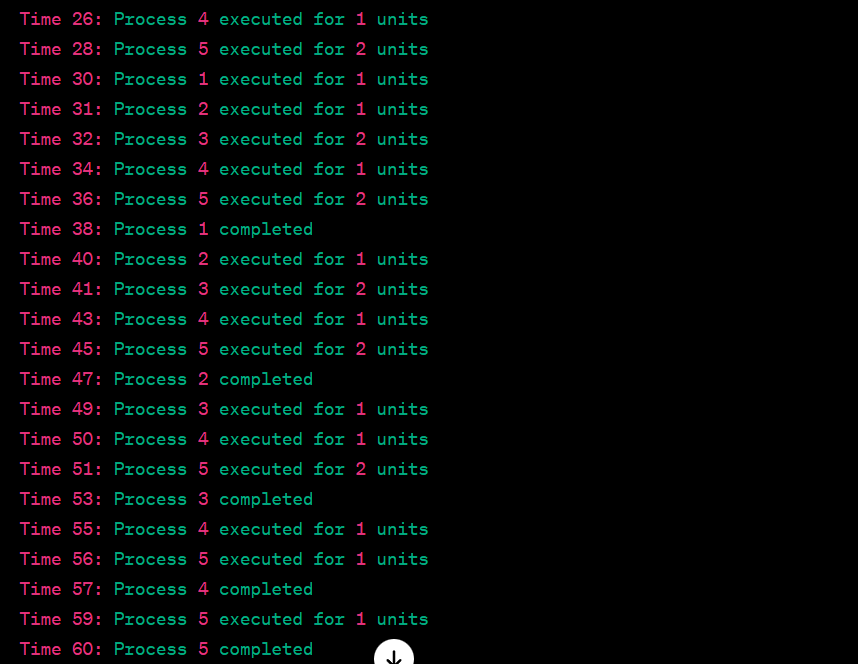
1) Poor performance with variable burst times: Round Robin may not perform well if there is a mixture of processes with short and long burst times. Short processes may end quickly, while long processes will take longer, leading to poor overall system efficiency.

2) High turnaround time: The average turnaround time (the time it takes to complete a process from dispatch to completion) can be higher compared to other scheduling algorithms, especially if the processes have different burst times.

1. Low quantum inefficiency: If the time quantum is too small, the context switching overhead between processes can be significant, impacting overall system performance.

Output ( Execution Sequence):





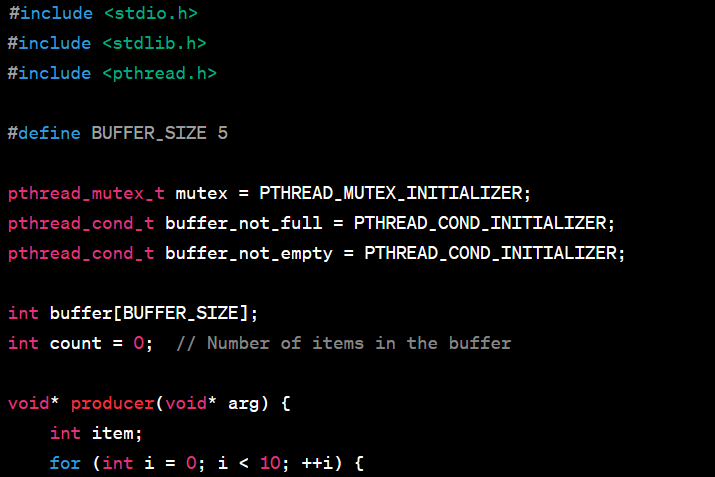
TASK 2

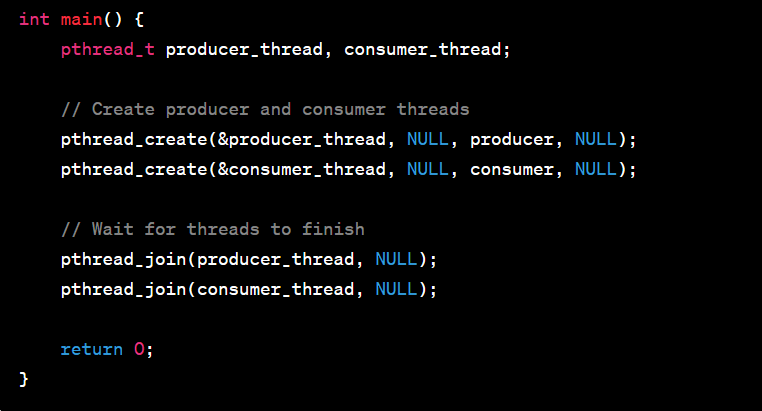
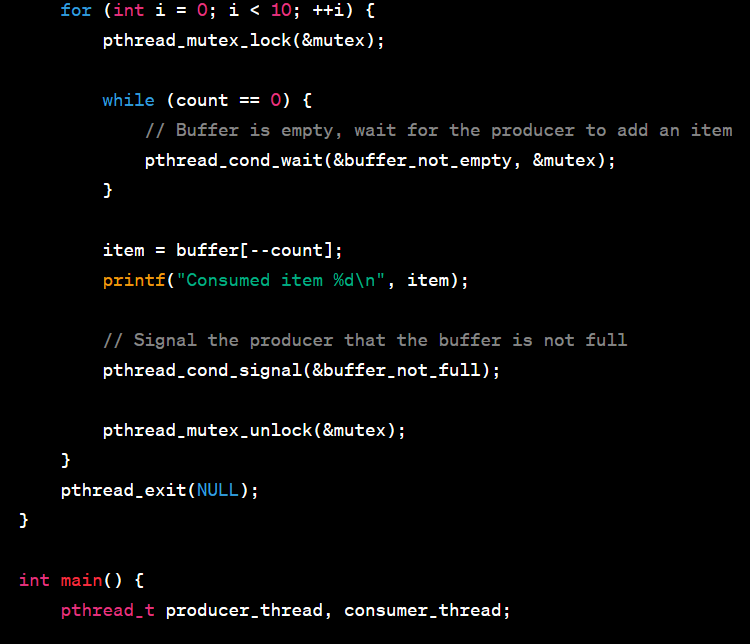
Implement the producer-consumer problem using condition variables. Write a program that

showcases how condition variables can be employed to synchronize the actions of producers

and consumers. Discuss the advantages of using condition variables for synchronization.

Solution:





Explanation:

Explanation:

Headers and Macros:

The code includes the necessary header files (#include <stdio.h>, #include <stdlib.h>, #include <pthread.h>).

The BUFFER\_SIZE macro is defined to set the size of the shared buffer.

Buffer structure:

A Buffer structure is defined to represent the shared buffer.

It includes an array to store data, a buffer count, a mutex for synchronization, and two condition variables (full and empty) for signaling.

Functional prototypes:

Function prototypes for initBuffer, produce, and consume are declared.

Initialization function (initBuffer):

The initBuffer function initializes the buffer, sets the count to 0, and initializes the mutex and condition variables.

Function of producer (producer):

The producer function represents the behavior of the producer thread.

Produces 10 items using the production function with a simulated variable production time (using sleep).

Consumer function (consumer):

A consumer function represents the behavior of a consumer thread.

Consumes 10 items using a consumption function with a simulated variable consumption time (using sleep).

Main function (main):

* Main function:
* Declare a buffer variable struct Buffer.
* Initializes the buffer using initBuffer.
* Creates producer and consumer threads using pthread\_create.
* Waits for both threads to finish using pthread\_join.
* Mutex and conditional variables:
* The buffer structure contains a mutex (mutex) to protect access to the buffer and two condition variables (full and empty) that indicate whether the buffer is full or empty.
* Producer and consumer threads:
* Producer and consumer functions represent the behavior of producer and consumer threads.
* They access the shared buffer using the production and consumption functions.
* The simulated production and consumption time variables introduce randomness.
* To lock and unlock a mutex:
* Mutexes are used to provide exclusive access to a shared buffer. Each critical section is protected by locking the mutex with pthread\_mutex\_lock and unlocking it with pthread\_mutex\_unlock.
* Using conditional variables:
* Condition variables (full and empty) are used for signaling between the producer and consumer threads.
* When the stack is full, the producer waits for an empty condition variable and when it creates an item, it signals that the stack is no longer empty.
* When the stack is empty, the consumer waits for the full condition variable, and when it consumes the item, it signals that the stack is no longer full.
* To create a thread and connect:
* Threads are created using pthread\_create.
* The main function waits for the producer and consumer threads to finish using pthread\_join.

Algorithm:

1) Initialization:

>Initialize buffer, mutex and condition variables (full and empty).

>Create two threads, one for producer and one for consumer.

2) Functions of the manufacturer:

>The producer function runs in a loop and creates items.

>Acquires a mutex lock to ensure exclusive access to a critical section.

>If the buffer is full (count == BUFFER\_SIZE), the producer waits for an empty condition variable and releases the mutex.

>Once there is free space in the buffer, the producer produces an item, adds it to the buffer, updates the count, and signals the full state variable to wake up the waiting consumers.

The mutex is released to allow other threads to access the critical section.

3) Functions of the consumer:

>The consumer function runs in a loop and consumes items.

>Acquires a mutex lock to ensure exclusive access to a critical section.

>If the buffer is empty (count == 0), the consumer waits for the condition variable to be full and releases the mutex.

>Once the item is in the buffer, the consumer consumes it, updates the count, and signals an empty condition variable to wake up the waiting producer.

The mutex is released to allow other threads to access the critical section.

4) Main functions:

>Creates production and consumer threads.

>Waits for both threads to finish their execution with pthread\_join.

Sequence of execution:

1)The main function creates producer and consumer threads.

2)The producer thread starts executing the producer function.

>Enters a critical section, creates an item, updates the buffer, and signals a full status variable.

>The producer can go to sleep if the buffer is full (waiting for empty).

3)The consumer thread will start executing the consumer function.

>Enters the critical section, consumes the item, updates the buffer, and signals an empty condition variable.

>Consumer can go to sleep if buffer is empty (waiting for full).

4)Producer and consumer threads continue to loop and produce and consume items.

5)Threads can block (sleep) or wake up based on conditions specified by condition variables, ensuring that the producer waits when the buffer is full and the consumer waits when the buffer is empty.

6)The main function waits for both threads to finish their execution using pthread\_join.

The use of condition variables (full and empty) ensures that the producer and consumer threads synchronize their actions and avoid conflict conditions related to the state of the buffer. A mutex (pthread\_mutex\_t mutex) ensures that access to shared resources is mutually exclusive, preventing conflicts between threads.

Advantages of using variable conditions for synchronization:

1) Efficient waiting: Condition variables allow threads to efficiently wait for a certain condition to be met without constantly checking in a loop. This reduces CPU usage and improves overall system performance.

2) Reduced wait time: Without variable conditions, synchronization often involves waiting for busyness (polling), which is inefficient and wastes CPU resources. Condition variables help eliminate busy waits by allowing threads to block until a condition is met.

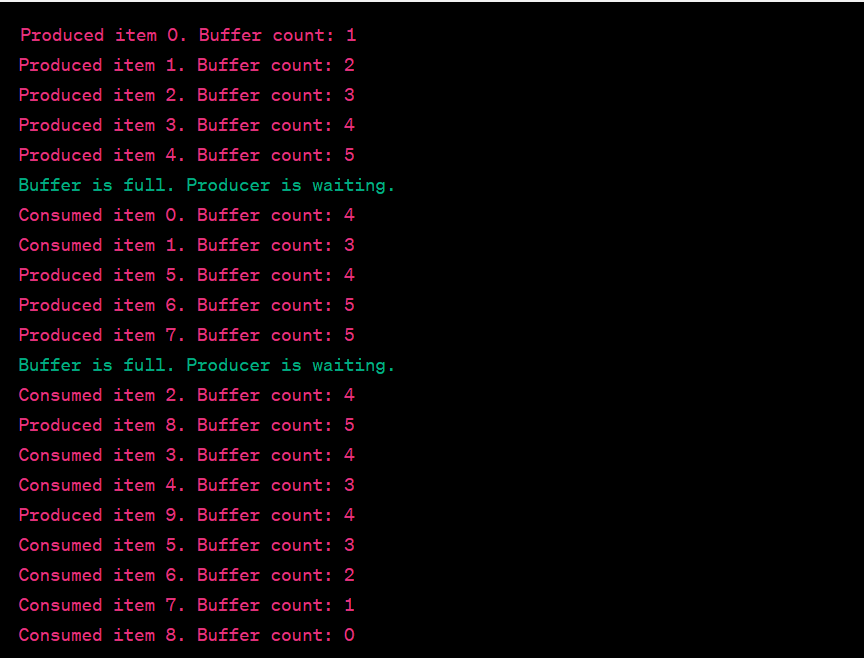
3) Fine Control: Condition variables provide a mechanism for fine control of synchronization. Threads can wait for specific conditions and signals can be broadcast to multiple waiting threads.

4) Avoiding race conditions: By using variable conditions in conjunction with mutex locks, you can avoid race conditions and ensure that critical parts of the code are executed atomically.

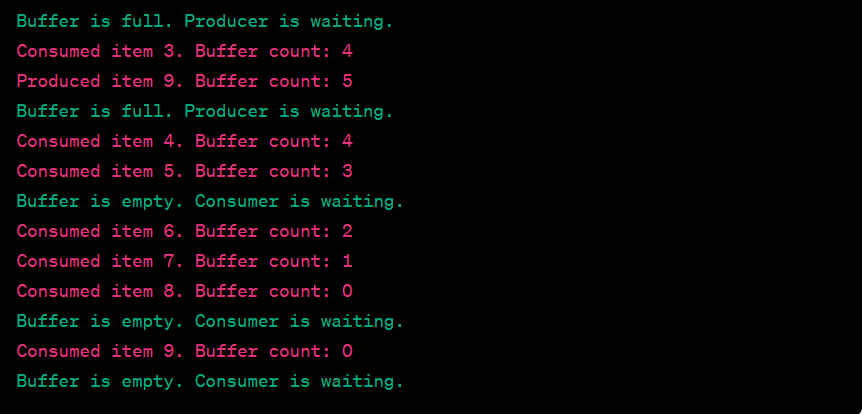
5) Modularity: Conditional variables help in creating modular and well-structured code for synchronization. They separate waiting and signaling concerns from critical sections protected by mutex locks.

Note that proper use of condition variables requires careful consideration of the order in which mutex locks are acquired and released to avoid potential deadlocks or races.

OUTPUT:







* This output represents the interleaved actions of the producer and consumer threads. The producer produces items and the consumer consumes items from the shared buffer. Messages indicating wait conditions (Buffer is full. Producer is waiting. and Buffer is empty. Consumer is waiting.) are printed when the buffer is full or empty and the thread in question must wait until the condition changes. Random production and consumption times can lead to different execution sequences.

GROUP PHOTO:

