ASSIGNMENT

BY

AARAB MANHAS
2022a1r044
3rd Semester
Computer Science Engineering (CSE)
GROUP G (2022A1R039-2022A1R044)

Model Institute of Engineering and Technology (Autonomous)
(Permanently Affiliated to the University of Jammu, Accredited by NAAC with "A" Grade)
Jammu, India
2023

Subject Code: COM-302

Due Date: 4 - December - 2023

Question Number	Course Outcomes	Blooms' Level	Maximum Marks	Marks Obtain
Q 1	CO1, CO2 & CO5	3-6	10	
Q2	CO3, CO4	3-6	10	
Total Marks			20	
Faculty Name: Prof. Mekhla Sharma				

TASK-1

Q1. Design a program that implements Round Robin scheduling Algorithm. Create a set of processes with specified quantum time and demonstrate how the operating system schedules these processes. Implement and analyse the algorithm with at least 3 different specified quantum time.

ANS. Round Robin scheduling algorithm is a CPU scheduling algorithm that assigns a fixed time quantum to each process in a cyclic way. The algorithm is pre-emptive, meaning that the CPU can be taken away from a process even if it has not completed its execution. Here is an implementation of the algorithm in Python:

CODE:

```
def round_robin(processes, quantum):
    n = len(processes)
    burst_time = [process [1] for process in processes]
    waiting_time = [0] * n
    turnaround_time = [0] * n
    remaining_time = burst_time.copy()
    time = 0
    while True:
        done = True
        for i in range(n):
            if remaining_time[i] > 0:
                done = False
                if remaining_time[i] > quantum:
                    time += quantum
                    remaining_time[i] -= quantum
                else:
                    time += remaining_time[i]
                    waiting_time[i] = time - burst_time[i]
                    remaining_time[i] = 0
                    turnaround_time[i] = waiting_time[i] + burst_time[i]
        if done:
            break
    return waiting_time, turnaround_time

# Sample usage
processes = [("P1", 5), ("P2", 4), ("P3", 2), ("P4", 1)]
quantum_times = [1, 2, 3]
```

```
for quantum in quantum_times:
    waiting_time, turnaround_time = round_robin(processes, quantum)
    print(f"Quantum time: {quantum}")
    print(f"Waiting time: {waiting_time}")
    print(f"Turnaround time: {turnaround_time}")
```

OUTPUT:

```
Quantum time: 1
Waiting time: [7, 7, 5, 3]
Turnaround time: [12, 11, 7, 4]
Quantum time: 2
Waiting time: [7, 7, 4, 6]
Turnaround time: [12, 11, 6, 7]
Quantum time: 3
Waiting time: [6, 8, 6, 8]
Turnaround time: [11, 12, 8, 9]
```

- The execution sequence and a detailed explanation of code and algorithms used.

The code you provided is an implementation of the Round Robin scheduling algorithm.

Step by step Explanation:

- n = len(processes): Get the number of processes from the input.
- burst time = [process [1] for process in processes]: Extract the burst times of the processes from the input and store them in the burst time list.
- waiting time = [0] * n: Initialize a list to store the waiting time for each process, initially set to 0 for all processes.
- turnaround time = [0] * n: Initialize a list to store the turnaround time for each process, initially set to 0 for all processes.
- remaining time = burst _time. Copy (): Create a copy of the burst times to keep track of the remaining time for each process.
- time = 0: Initialize the current time to 0.
- while True: Start an infinite loop to simulate the scheduling until all processes are executed.
- done = True: Flag to check if all processes are done.
- for i in range (n): Iterate through each process.
- if remaining time[i] > 0: Check if the remaining time for the process is greater than 0, indicating the process is not finished.

- done = False: Set done to False, indicating there are still processes to execute.
- if remaining time[i] > quantum: If the remaining time for the process is greater than the time quantum, execute for the time quantum.
- time += quantum: Increment the total time by the time quantum.
- remaining time[i] -= quantum: Subtract the time quantum from the remaining time of the current process.
- else: If the remaining time is less than or equal to the time quantum, execute the remaining time.
- time += remaining time[i]: Increment the total time by the remaining time of the current process.
- waiting time[i] = time - burst time[i]: Calculate the waiting time for the current process.
- remaining time[i] = 0: Set the remaining time for the current process to 0, as it has finished execution.
- turnaround time[i] = waiting time[i] + burst time[i]: Calculate the turnaround time for the current process.
- if done: If all processes are done, exit the loop.
- break: Break out of the infinite loop.
- return waiting time, turnaround time: Return the lists of waiting time and turnaround time for each process.

In summary, this code simulates the execution of processes using the Round Robin scheduling algorithm, updating the waiting time, turnaround time, and remaining time for each process until all processes are completed. The function then returns the waiting time and turnaround time for each process.

Here's an analysis report for the code:

Round Robin Scheduling Algorithm:

- The code implements the Round Robin scheduling algorithm, a widely used algorithm in operating systems for process scheduling.
- Round Robin is a pre-emptive scheduling algorithm where each process is assigned a fixed time slot or quantum. When a process's time quantum expires, it is moved to the back of the queue, and the next process in the queue gets a chance to execute.

- **Function Overview:**

1. The round robin function takes a list of processes and a time quantum as input and returns lists of waiting time and turnaround time for each process.
2. It uses a while loop to simulate the execution of processes until all processes are completed.
3. The algorithm iterates through each process in a round-robin fashion, deducting the quantum time from the remaining time of each process until they complete.

- **Data Structures:**

1. The code uses lists to store burst times, waiting times, turnaround times, and remaining times for each process.
2. The processes list contains tuples with process names and their burst times.
3. The burst time, waiting time, turnaround time, and remaining time lists are used to store relevant information for each process.

- **Time Complexity:**

The time complexity of the algorithm is O(n^2), where 'n' is the number of processes. This is because, in the worst case, the algorithm may iterate through all processes for each time quantum in the while loop.

- **Sample Usage:**

The code demonstrates the usage of the round robin function with a sample set of processes and different time quantum values.
The waiting time and turnaround time are printed for each case.

- **Output Interpretation**:

The output provides insights into the performance of the Round Robin algorithm with different time quantum values.
It helps in understanding how the choice of the time quantum impacts waiting times and turnaround times for the given set of processes.

- **Possible Enhancements**:

The code assumes that the input processes are sorted by arrival time. If not, a sorting step might be added.
The code could be extended to include additional metrics or visualization for a more comprehensive analysis.

- **Conclusion:**

The Round Robin scheduling algorithm is a simple and widely used approach for time-sharing systems.
The provided code offers a basic implementation and can be further customized or extended based on specific requirements or additional features needed in a scheduling system.
This analysis provides an overview of the code's functionality, its time complexity, and potential areas for improvement or extension.

TASK-2

Q2. Design and implement various methods for IPC, such as message passing or shared memory, to facilitate communication between processes in the operating system.

ANS.

Inter-Process Communication (IPC) is a crucial aspect of modern operating systems and distributed systems. It allows multiple processes to interact and share resources. Here are some methods for implementing IPC:

1. **Shared Memory:**

Shared memory is a technique where multiple processes can access a common memory region. This method is efficient for data transfer as there is no overhead of message passing. However, it requires synchronization mechanisms to avoid race conditions and deadlocks.

2. **Message Passing:**

Message passing is a technique where processes exchange messages through queues or channels. This method provides a decoupled communication mechanism, where sender and receiver do not have to be active simultaneously. It also supports asynchronous communication, making it suitable for distributed systems.

3. **Pipes:**

Pipes are a form of message passing that allows processes to communicate through a unidirectional channel. Pipes are commonly used in shell scripts and command pipelines to pass data between commands.

4. **Signals:**

Signals are asynchronous events that interrupt the execution of a process. Signals are used to notify processes of exceptional events such as termination, interrupts, or errors. Signals can also be used for synchronization purposes by sending signals between processes.

5. **Sockets:**

Sockets are network-based IPC mechanisms that allow processes running on different machines to communicate over a network. Sockets support various protocols such as TCP/IP, UDP, and Unix domain sockets for local communication.

6. **Semaphores:**

Semaphores are synchronization primitives used to control access to shared resources by multiple processes. Semaphores can be used in shared memory or message passing systems to prevent race conditions and deadlocks.

7. **Mutexes:**

Mutexes (mutual exclusion locks) are synchronization primitives used to ensure exclusive access to shared resources by a single process at a time. Mutexes can be implemented using semaphores or hardware-based locking mechanisms for faster performance.

8. **Condition Variables:**

Condition variables are synchronization primitives used to wait for specific conditions before proceeding with execution. Condition variables can be used with mutexes to implement wait-notify mechanisms for efficient resource sharing in multi-threaded applications.

These methods can be implemented using various programming languages such as C, Python, or Java, depending on the specific requirements of the application or operating system being developed.

Here are the steps to implement different methods of IPC.

1. Message Passing:

a. Define the message structure:
The message structure should contain the necessary data that needs to be passed between the processes. The message structure can be defined using a struct or a class, depending on the programming language being used.

b. Create message queues:
Each process should create its own message queue using the operating system's IPC API. The queue should have a maximum size, which determines how many messages it can hold at a time.

c. Send messages:
To send a message, the process should first obtain a handle to the destination process's message queue using the operating system's IPC API. Then, it can write the message into the queue using the write () function provided by the API. The sending process should also specify the priority of the message, which determines its order of execution in the destination process's queue.

d. Receive messages:
To receive messages, the process should use the read () function provided by the operating system's IPC API to read messages from its own message queue. The receiving process should also specify a timeout value, which determines how long it should wait for a message to arrive before returning an error.

e. Deleting message queues:
When a process is finished with its communication needs, it should delete its own message queue using the operating system's IPC API to free up resources.

2. **Shared Memory**:

a. Allocate shared memory:

The operating system provides an API for allocating shared memory segments that can be accessed by multiple processes simultaneously. Each process should obtain a handle to the shared memory segment using this API.

b. Map shared memory:
Each process should map the shared memory segment into its own address space using the operating system's IPC API. This allows processes to access data in the shared memory segment as if it were located in their own memory space.

c. Read and write shared memory:
Processes can read and write data in the shared memory segment using standard memory access instructions provided by their programming language or CPU architecture. Since multiple processes have access to this shared resource, synchronization mechanisms like locks or semaphores may be required to prevent race conditions and data corruption issues.

d. Unmap shared memory:
When a process is finished with its communication needs, it should unmap the shared memory segment from its address space using the operating system's IPC API to free up resources and prevent any potential conflicts with other processes that may still be accessing it.

3. **Pipes:**

- Creation: Create a pipe using the pipe system call.
- Communication: One process writes to the pipe (write system call), and the other reads from it (read system call).

4. **Signals:**

- Signal Definition: Define signal handlers using the signal function.
- Signal Sending: Send signals using the kill system call or other signal-generating functions.
- Signal Handling: Handle signals in the appropriate signal handler function.

5. **Sockets:**

- Creation: Create a socket using the socket system call.
- Binding: Bind the socket to an address and port using the bind system call.

- Listening/Connecting: For servers, listen for incoming connections (listen), and for clients, initiate a connection (connect).
- Data Transfer: Use send and recv for communication over the socket.

6. **Semaphores:**
   - Initialization: Initialize a semaphore using the sem_init function.
   - Operations: Perform semaphore operations such as sem_wait (decrement), sem_post (increment), and sem_destroy (destroy).

7. **Condition Variables:**

   - Initialization: Initialize a condition variable using pthread_cond_init.
   - Waiting: Use pthread_cond_wait to wait on a condition variable.
   - Signaling: Use pthread_cond_signal to wake up one waiting thread or pthread_cond_broadcast to wake up all waiting threads.
   - Destruction: Destroy the condition variable using pthread_cond_destroy.

8. **Mutexes (Mutual Exclusion):**

   - Initialization: Initialize a mutex using the pthread_mutex_init function.
   - Locking/Unlocking: Use pthread_mutex_lock to acquire the lock and pthread_mutex_unlock to release it.
   - Destruction: Destroy the mutex using pthread_mutex_destroy.