**Assignment 3: Deadlock Avoidance**

Atuhaire Ambala and Ricardo Escarcega

Prof. Isac Artzi

CST 315

June 2nd 2024

**Assignment 3: Deadlock Avoidance**

In a multi-threaded environment, several processes compete for access to a shared resource, such as a file or a specific memory address. This competition can lead to situations where some processes are starved of the resource they need, potentially leading to performance bottlenecks and inefficient resource utilization.

To mitigate this issue, a timer-based solution is proposed, where each process has a predetermined time interval to access the resource. If a process cannot access the resource within this time frame, it is considered starved, terminated, and then restarted. This mechanism ensures that no process waits indefinitely for the resource, thereby preventing deadlock situations.

The challenge is to implement this solution in C, ensuring that the system efficiently handles multiple processes, logs activities, and effectively prevents starvation. The implementation should track and report which processes are running, the resources they are trying to access, the availability of these resources, and whether any process experiences starvation.

**Implementation Approach in C**

1. Initialization:
   1. Define constants for the number of processes, timeout duration, and maximum retries.
   2. Initialize a mutex to control access to the shared resource.
2. Process Data Structure:
   1. Create a structure to hold process-specific data, including process ID, retry count, and start time.
3. Thread Function:
   1. Each thread simulates a process trying to access the shared resource.
   2. If the resource is available, the thread locks the mutex, uses the resource, and then unlocks the mutex.
   3. If the resource is not available, the thread checks if it has been waiting longer than the timeout period. If so, it increments its retry count and restarts the timer.
4. Main Function:
   1. Create and start multiple threads (processes).
   2. Wait for all threads to complete execution.
   3. Destroy the mutex at the end.

**Flowchart**

Mermaid.js syntax

**flowchart** TD

A[Start] **-->** B[Initialize mutex]

B **-->** C[Create threads]

C **-->** D{Thread Function}

D **-->** E[Record start time]

E **-->** F[Try to lock mutex]

F **-->**|Success| G[Access resource]

G **-->** H[Release resource]

H **-->** I[Exit thread]

F **-->**|Failure| J[Check timeout]

J **-->**|Timeout exceeded| K[Increment retries]

K **-->** L[Restart timer]

J **-->**|Timeout not exceeded| M[Wait and retry]

M **-->** F

C **-->** N[Wait for threads to complete]

N **-->** O[Destroy mutex]

O **-->** P[End]

**A diagram of a process

Description automatically generated**

**Code**

**A screen shot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**A black background with orange text

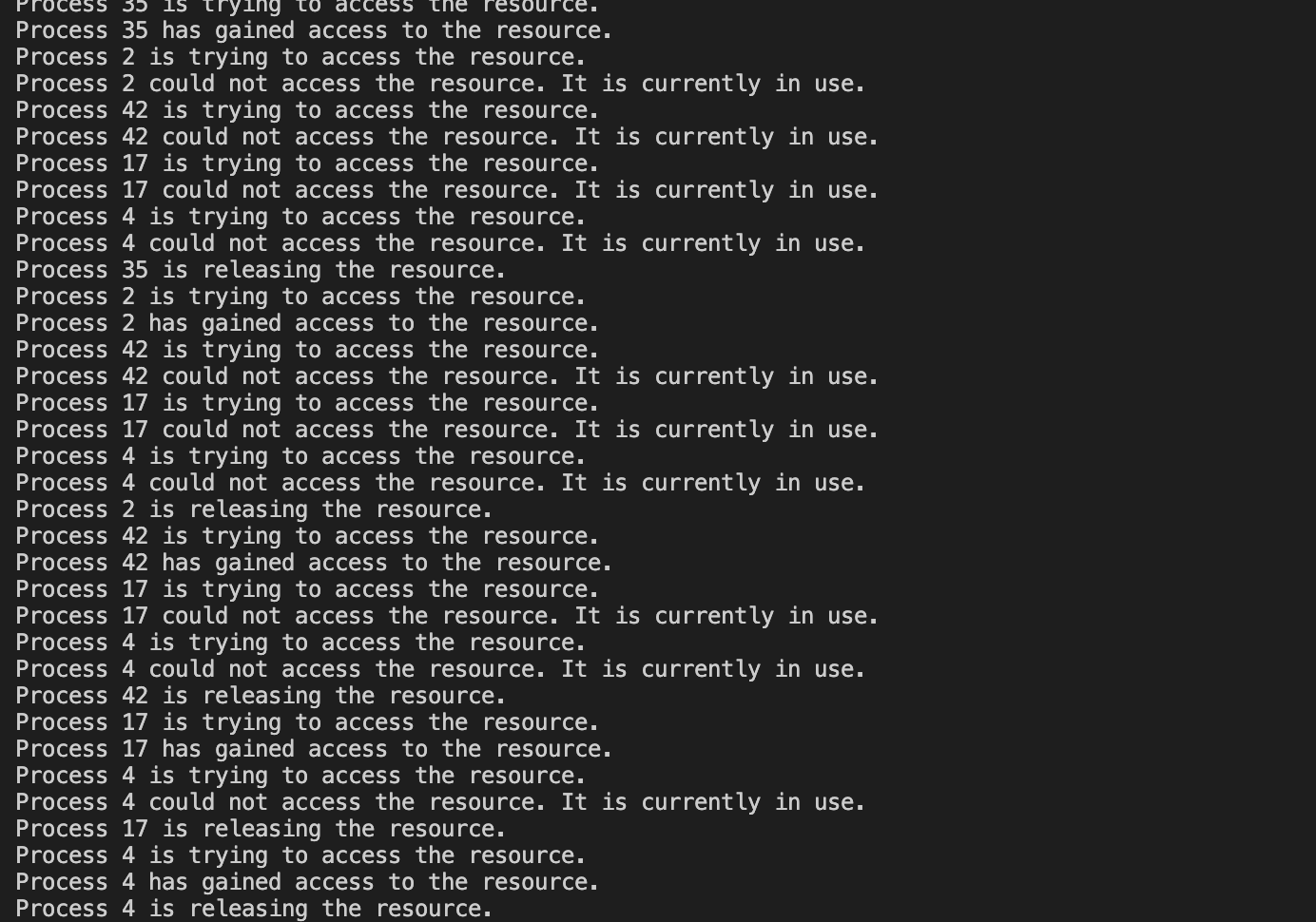
Description automatically generated**

**A computer screen shot of a program code

Description automatically generated**

**Execution**

TIMEOUT = 5 and NUM\_PROCESSES=50



TIMEOUT = 1 and NUM\_PROCESSES=30

A screenshot of a computer screen

Description automatically generated

**Testing and Validation**

To ensure the robustness and correctness of the timer-based starvation prevention mechanism, the implementation must undergo thorough testing and validation. The primary objective is to verify that the system efficiently handles multiple processes, prevents starvation, and manages resource contention without causing deadlocks. The testing process involves creating a controlled environment with a specified number of processes competing for the shared resource. By varying the number of processes and adjusting the timeout duration, we can observe the system's behavior under different load conditions. The validation process includes checking the activity logs to ensure that:

1. Resource Access: Processes can successfully access and release the resource.
2. Starvation Handling: Processes that experience starvation are correctly terminated and restarted.
3. Log Accuracy: The activity log accurately reflects the status of each process, including attempts to access the resource, successful accesses, releases, and starvation events.

Additionally, edge cases such as extremely high contention (many processes with a very short timeout) and low contention (few processes with a long timeout) are tested to ensure the system's stability and performance across a range of scenarios. By systematically testing and validating the implementation, we can confirm that the solution is both effective and reliable in preventing process starvation while managing shared resource access in a multi-threaded environment. Examples of our testing can be found in the screenshots above

Analysis of Program

We can observe several key points about the behavior of the system by comparing the system with a timeout of 5 seconds and 50 processes vs a timeout of 1 seconds and 30 processes:

1. **Reduced Starvation**: The increased timeout allows processes enough time to access the resource, preventing frequent starvation and restarts.
2. **Sequential Access**: Processes are able to access the resource in a somewhat sequential manner, though many processes still experience contention.

**Efficiency and Scalability**

**Efficiency**

* **Resource Utilization**: The resource is utilized more effectively since processes have sufficient time to complete their operations before being terminated due to starvation.
* **Overhead Reduction**: The number of restarts due to starvation is significantly reduced, decreasing the overhead associated with frequent process termination and restart.

**Scalability**

* **High Number of Processes**: The system can handle a larger number of processes without significant issues related to starvation, as seen in the activity log.
* **Contention Management**: While the system manages contention better with a longer timeout, there is still visible contention when multiple processes attempt to access the resource simultaneously.

**Limitations**

1. **Contention Remains**: Despite the increased timeout, there is still significant contention for the resource, causing many processes to wait.
2. **Potential Delays**: Processes may still experience delays due to the time spent waiting for the resource to become available, especially as the number of processes increases.

**Alternative Approaches for Improvement**

Given that the current solution with a 5-second timeout works well but still exhibits contention, we can consider alternative approaches to further improve efficiency and scalability. The first example would be to Implement a priority-based scheduling mechanism where processes that have been waiting longer are given higher priority to access the resource. This ensures fairer access by prioritizing waiting processes and reduces the chance of prolonged contention and starvation. Second would be a token-passing mechanism where only the process holding the token can access the resource. This would eliminate contention by ensuring exclusive access to the resource. Lastly would be to use an exponential backoff strategy where processes wait for an increasing amount of time before retrying to access the resource.

**Conclusion**

The timer-based approach with a longer timeout of 5 seconds effectively reduces starvation and overhead, making it a viable solution for scenarios with high contention. However, to further improve resource utilization and reduce contention, implementing alternative strategies like priority-based scheduling, token-based access, or an exponential backoff strategy can provide additional benefits.