Linked List Exclusion: An Approach to Critical Section Mutual Exclusion Using Single Index Process Queues

Karl A. Morris
Florida International University
09/16/09

Abstract

The current methods utilized in implementing mutual exclusion on critical sections require excessive communication overhead via broadcasting to processes when seeking access to a critical section, or confirming that the process is not in use by the responding process. Additionally, current implementations utilize queues of N length to track access requests for the critical section.

We propose an algorithm that utilizes a queue of length 1 to realize sequential process access to a critical section by dynamically building a linked list, where effective pointers are built by queuing the request of a single subsequent process. The reduction in queue size will not only reduce the amount of storage required per process for implementing mutual exclusion, but will also aid in realizing faster hand over between processes by reducing the overall communication time between subsequent processes in the sequence.

Introduction

An increasingly non-trivial amount of time is spent during message passing between processes. This problem is exacerbated when broadcasting and queuing is utilized to achieve a particular outcome such as Critical Section Mutual Exclusion. A simpler mechanism could increase the efficiency of the necessary interprocess communication by reducing the size of the queues and utilizing a mechanism of direct communication when possible.

The Approach

To access a critical section, a process (P_1) broadcasts to all running processes (P_n). Once received, P_n responds **iff** they do not have a pending request **and** have not already queued a request. P_n queues a request **iff** they have a pending request **and** have not already queued a request. A process maintains its Request logical timestamp for fault tolerance purposes which will be detailed later on.

P_1 may accesses the critical section when it receives N-1 responses. It is queued when it receives N-2 responses. Upon exiting the critical section a process responds to the process request it has stored in its queue and clears the queue.

Table 1.Process request and response

Time/Process	P_1	P_2	P_3	P_4	P_5
t_1	req	ok	ok	ok	ok
t_2	queue	req	ok	ok	ok
t_3	ok	queue	req	ok	ok
t_4	ok	ok	queue	req	ok
t_5	ok	ok	ok	queue	req

The results of this are illustrated in Figure 1. Each process is able to queue the request of the most recent request following its own. A logical linked list results where processes are released using a Fist In First Out method. All new requests will be queued by the process with the preceding request.

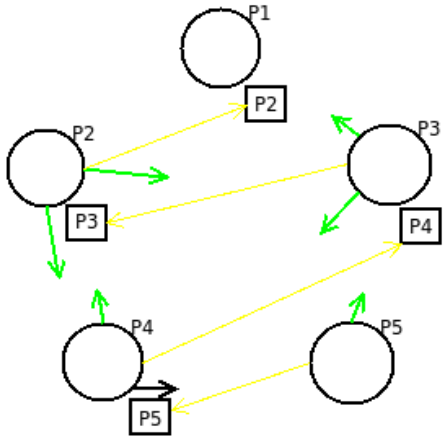

Figure 1. Process message queuing

Mitigating Deadlock

Deadlock may occur if a process blindly queued the first request received subsequent to its own request. A simple example of deadlock is if two processes both submit a request at relatively the same time. This may result in each process queuing its sister process's request, thereby eliminating the possibility of receiving a response. To correct this issue we implement a Lamport logical clock to ascertain the timing of each event. When process P_2 receives a request from process P_1 that is older than the its own pending request, it does not queue the request, but instead respond to P_1. P_1, upon receiving P_2's request, which is older, queues the request as normal and continues to wait for responses from all remaining processes.

Fault Tolerance

The death of a process would result in a deadlock situation where all subsequent processes would become zombies. To correct this problem we will implement a previsioning mechanism to remove the dead process reference from the queue.

When a process dies, all remaining processes broadcast the time of their last request (P_1.req). Each process then listens for incoming messages from the remaining processes (P_n.req). When a process receives its first response, it compares it to its own request time.

If P_1.req > P_2.req

    P_1.res(P_2)

else

    P_1.enqueue(P_2.req)

This continues until it finds a message which is after its own and that message is queued.

After its first queued message, it compares all subsequent messages both to its own request time and the request time it has queued as depicted below.

```
If P1.req > Pn.req
        P1.res(Pn)
else
        If P1.queue() < Pn.req
                P1.res(Pn)
        else
                P1.queue() > Pn.req
                        P1.res(P1.dequeue)
                        P1.enqueue(Pn.req)
```

This continues until each process receives at least N-2 responses. The process that receives N-1 responses is first in the linked list and may access the critical section. Processes that did not have pending requests at the time of the election would submit a sentinel value such as '0' or '-1' to indicate this. Receiving processes would ignore these values, but increment their internal counters to ensure they inevitably record N-1 responses.

Acknowledgements

We would like to thank Yali Wu and Juan Carlos Martinez for their invaluable contribution to the modeling and testing of the Linked List Exclusion algorithm.

References

1. Lamport, L.: **Time, clocks, and the ordering of events in a distribution system**. Communications of the ACM, Volume 21, Issue 7, ISSN:0001-0782, 1978