Resource Allocation

Processes share common resources e.g. a shared memory where only one holder of the resource should be able to access the resource.

The problem can be described by a finite set of resources and a finite set of processes that compete for accessing their resources. Any solution of the problem is required to guarantee:

- MUTUAL EXCLUSION: no resource may be accessed by more than one process at any time
- NO STARVATION: as long as processes do not fail every process which is trying to access the resource will succeed in finite time.

Special Case: Mutual exclusion (1 resource ahares by everybody, complete communication graph)

The algorithm of Ricart and Agrawala

This algorithm resolves conflicts between processes by sending messages with time stamps.

For this purpose a process which changes its state to HUNGRY sends messages, called *requests*, to all its neighbours.

A *request* message includes information about the unique identifier of the sender and a time stamp.

The TIME STAMP is the local clock of the sending process when the message was created. It is increased when a message *request* is sent or received.

On the receiving of *request* a process, which is competing with another process for a resource, can distinguish whether receiver or sender were requesting "first" for a Resource assuming the lexical order of events.

The Protocol

A process p that receives a message *request* from process q does the following depending on its state:

- If p is thinking, then it sends a message *fork* to q. Sending a *fork* a process gives permission to the other process to access the resource and guarantees not to access the resource until it received itself a message *fork*.

- If p is hungry, then it is competing with q for the same resource. This means that p sent before a message *request* to q, so p concludes by using the lexical order of events which process sent its message first. If p sent its message "after" q sent its *fork* then it replies by sending message *fork*. Otherwise it delays sending message *fork* until it finished with accessing its critical section.

- If p is eating it also delays sending message *fork* until it finished with its critical section.

A process may access its critical section, when it received a message *fork* from all its neighbours.

When it is finished it sends to all neighbours which requested to access a resource messages *fork* and changes its state to *thinking*.

Correctness

Mutual Exclusion

Proof. Assume towards a contradiction that processes p and q were in their critical section at the same time. $\square$

In order to gain access to their critical section processes p and q send *request* messages to all their neighbours.

Let t_p denote the local clock time when process p sent its *request* messages

and

let t_q denote the local clock time when process q sent its *request* messages.

Due to the lexical order of time either $t\_p < t\_q$ or $t\_p > t\_q$ is true.

$t\_p < t\_q$:

- p received request by q after it send its request

- would have delayed replying the request *message* of process q until p finished accessing its critical section.

This implies that $t\_p > t\_q$ was valid....

Contradiction.

Correct (progress).

Proof. Assume towards a contradiction again that: □

there is a point in the execution in which some processes are hungry, and after this point no process can enter its critical section.

Then the system will reach a state in which no messages are sent any longer and no process changes its state.

Let p be a process in state *hungry* with smallest request time in this "permanent" state.

p sent requests and is waiting for at least one reply:

- All neighbours in state *thinking* must have answered the *request*
- All processes in state *hungry* will send messages *fork* to p

Contradiction.

Fairness - no Starvation

A hungry process will access the Critical Section in a finite time.

Proof. Receiving message *request* will make the local clock of a process at least one unit greater than the time stamp of message *request*. □

Therefore, a process which sends *request* messages to all its neighbours will loose at most once in a competition with each neighbour while it is trying to access a resource.

From the previous proof we also have that while you are *hungry* the system will be serving neighbours...

Complexities

Time Complexity: *O(n*k)*

We assume that it take at most k time units for a process to exit the critical section.

Communication Complexity: *2(n-1)*

Fault tolerance: None

A failure of a process might stop all other processes from being able to access the critical section.