Concurrent Computing November 12, 2024

Solutions to Exercise 6

Problem 1. The following algorithm implements a contention manager that transforms any obstruction-free algorithm into a wait-free one:

uses: $T[1, \dots, N]$—array of registers, $Executing[1, \dots, N]$—atomic wait-free snapshot object
initially: $T[1, \dots, N] \leftarrow \bot$, $Executing[1, \dots, N] \leftarrow \bot$

upon try_i **do**
  if $T[i] = \bot$ **then** $T[i] \leftarrow$ `GetTimestamp()`

  repeat
    $sact_i \leftarrow \{ p_j \mid T[j] \neq \bot \wedge p_j \notin \Diamond\mathcal{P}.suspected_i \}$
    $Executing.update(i, \bot)$
    $leader_i \leftarrow$ the process in $sact_i$ with the lowest timestamp $T[leader_i]$
    if $leader_i = i$ **then** $Executing.update(i, i)$
  until $Executing.scan()$ *contains only i and* $\bot, \forall$ *processes* $\in sact_i$

upon $resign_i$ **do**
  $T[i] \leftarrow \bot$
  $Executing.update(i, \bot)$

The algorithm uses a procedure `GetTimestamp()` that generates *unique* timestamps. We assume that if a process gets a timestamp t from `GetTimestamp()`, then no process can get a timestamp lower than t infinitely many times. Thus, we can easily implement `GetTimestamp()` using only registers (or even without using any shared objects). For example, we can use the output of a counter (see the lecture notes on how to implement a counter from registers) combined with a process id (to ensure that timestamps are unique). The algorithm also uses a wait-free, atomic snapshot object to store the process that should be executing next (or is currently executing) in order to avoid two processes executing concurrently.