

Notes on Dijkstra's Algorithm

These are some notes on Dijkstra's mutual exclusion algorithm (CACM 8(9), Sept 1965, p. 569).

Below, I give Dijkstra's mutual exclusion algorithm using more modern notation. (The *Interested* array is the negation of Dijkstra's *b* array and the *Passed* array is the negation of Dijkstra's *c* array.) N is the number of processes in the system. The shared registers $Passed[i]$ and $Interested[i]$ can be written by process i and read by every process. The shared register K can be read and written by every process, and stores an integer value, where $1 \leq K \leq N$ (initially arbitrary). All boolean variables are initially false.

Shared variables:

$Interested[1..N]$ % Boolean entries indicate which processes are interested in using the resource
 $Passed[1..N]$ % Boolean entries indicate which processes have passed Phase 1
 K % integer is the name of a process allowed to try for the resource next

Local variable:

$done$ % Boolean variable that is used to determine when the process can exit the outer loop

Code for process i :

```

 $Interested[i] \leftarrow true$ 
loop
  loop % Entry Phase 1
    exit when  $K = i$ 
    if not  $Interested[K]$  then % try to go next
       $K \leftarrow i$ 
    end loop
     $Passed[i] \leftarrow true$  % begin Entry Phase 2
     $done \leftarrow true$ 
    for  $j \leftarrow 1..N$  except  $i$  % check if anyone else has passed Phase 1
      if  $Passed[j]$  then % go back to Phase 1
         $Passed[i] \leftarrow false$ 
         $done \leftarrow false$ 
      end for
    exit when done
  end loop
CRITICAL SECTION
 $Passed[i] \leftarrow false$ 
 $Interested[i] \leftarrow false$ 
REMAINDER SECTION

```

We now sketch the proof of correctness of the algorithm.

Exclusion property: No two processes are ever in the critical section simultaneously. The argument for this property is based entirely on the *Passed* array. (It has nothing to do

with the *Interested* array or K .) To derive a contradiction, suppose two agents i and j are in the critical section simultaneously. Then, consider the last time each of them set its entry of *Passed* to true prior to entering the critical section. Without loss of generality, suppose i did this after j did. Then, when i reads *Passed*[j] during Phase 2, it will see that *Passed*[j] is true, and it will go back to Phase 1. This contradicts the fact that i gets into the critical section.

Progress property: If anybody wants to enter the critical section, eventually somebody does. Suppose some processes are trying to get to the critical section (*i.e.*, in the big loop) and no process is in the critical section. We must show that some process eventually enters the critical section. Eventually K will get set to the name of one of the processes that is trying to enter the critical section. The value of K may change a few times (for example if several processes are about to write their own ids into K) but, eventually, K will stop changing until some progress is made because *Interested*[K] will be true. Once K has settled down to a particular value, that process must just wait until all other processes get kicked out of phase 2. Those processes won't be able to re-enter phase 2. So the process whose id is in K will eventually get into the critical section.

Notice that this algorithm allows starvation: one process may be left out of the critical section forever while another enters it infinitely many times.