**Proof of a Solution to the Critical Section Problem**

**Peterson's Algorithm**

**Common data objects:**

int turn = 0;  
bool flag[2] = {false};

**Code:**

flag[i] = true;  
turn = j;  
while (flag[j] && turn == j)  
    ;  
 //  CS  
flag[i] = false;

**Idea:**

* flag[i] indicates position of Pi with respect to mutual exclusion
* turn resolves simultaneity conflicts
* some process sets turn last (before entering the while test)

**Proof:**

**Mutual Exclusion**   
Proof by contradiction: assume both P0 and P1 are in their CS   
- then flag[0] = flag[1] = true   
- the test for entry cannot have been true for both processes at the same time (because turn favors one);   
   therefore one process must have entered its CS first (without loss of generality, say P0)   
- but this means that P1 could not have found turn = 1 and therefore could not have entered its CS (i.e. contradiction)

**Blocking**   
Consider P0 blocked at the while loop:

**Progress**

*Case I: (Stuck)*   
P1 is not interested in entering its CS   
- then flag[1] = false   
- hence the while loop is false for P0 and it can go

*Case II: (Deadlock)*   
P1 is also blocked at the while loop   
- impossible, because turn = 0 or 1   
- hence the while loop is false for some process and it can go

**Bounded-Waiting**

*Case III: (Starvation)*   
P1 is executing its CS repeatedly   
- upon exiting its CS, P1 sets flag[1] = false   
- hence the while loop is false for P0 and it can go (sufficient?)

However, P1 may attempt to re-enter its CS before P0 has a chance to run.   
- but to re-enter, P1 sets flag[1] to true and sets turn to 0   
- hence the while loop is true for P1 and it waits   
- the while loop is now false for P0 and it can go

--

/\* Eisenberg-McGuire algorithm: a software approach to N-process

mutual exclusion.

For description of Eisenberg-McGuire algorithm, see page 261 of

"Concurrent Systems - Operating Systems, Database and Distributed

Systems: An Inegrated Approach / Jean Bacon -- 2nd Edition".

Copyrigh (c) 2001 Xiao Zhang \*/

#include <stdlib.h>

#include <pthread.h>

#include <iostream>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Eisenberg-McGuire's algorithm for N-process mutual exclusion \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class eis\_mcg\_mutex\_t {

private:

int n;

enum procphase { out\_cr, want\_cr, claim\_cr } \*procphase;

int turn;

public:

/\* Initialize the mutex data shared by N processes \*/

eis\_mcg\_mutex\_t(int nproc) {

n = nproc;

procphase = new enum procphase [n];

srand(time(0));

turn = (int) (1.0 \* n \* rand() / (RAND\_MAX + 1.0));

for (int i = 0; i < n; i++) procphase[i] = out\_cr;

}

~eis\_mcg\_mutex\_t() {

delete [] procphase;

}

/\* Entry protocol for process i \*/

void mutex\_lock(int i) {

procphase[i] = want\_cr;

int j = turn;

do {

while (j != i) {

if (procphase[j] == out\_cr) j = (j + 1) % n;

else j = turn;

}

procphase[i] = claim\_cr;

j = (j + 1) % n;

while (procphase[j] != claim\_cr) j = (j + 1) % n;

} while (!(j == i && (turn == i || procphase[turn] == out\_cr)));

turn = i;

}

/\* Exit protocol for process i \*/

void mutex\_unlock(int i) {

int j = (turn + 1) % n;

while (procphase[j] == out\_cr) j = (j + 1) % n;

turn = j;

procphase[i] = out\_cr;

}

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* To test the Eisenberg-McGuire's algorithm, we write a simple \*/

/\* program that creates N threads (processes) and then has each \*/

/\* thread increment a global variable `counter' NLOOP times. The \*/

/\* final value of `counter' is expected to be N \* NLOOP. \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define N 4 /\* number of threads \*/

#define NLOOP 1000 /\* number of times each thread loops \*/

int counter; /\* this is cremented by the threads \*/

eis\_mcg\_mutex\_t counter\_in\_use(N);

void \*doit(void \*arg)

{

int i, val;

int tid = (int)arg;

/\* Each thread fetches, prints and increments the counter NLOOP times.

The value of the counter should increase monotonically. \*/

for (i = 0; i < NLOOP; i++) {

/\* Replace pthread\_mutex\_lock() with Eisenberg-McGuire's

enter-critical-section procedure. \*/

counter\_in\_use.mutex\_lock(tid);

/\* Here is critical section \*/

val = counter;

counter = val + 1;

cout << tid << ": " << counter << endl;

/\* Replace pthread\_mutex\_unlock() with Eisenberg-McGuire's

leave-critical-section procedure. \*/

counter\_in\_use.mutex\_unlock(tid);

}

return NULL;

}

int main()

{

pthread\_t tid[N];

int i;

for (i = 0; i < N; i++) pthread\_create(&tid[i], NULL, doit, (void \*)i);

for (i = 0; i < N; i++) pthread\_join(tid[i], NULL);

return 0;

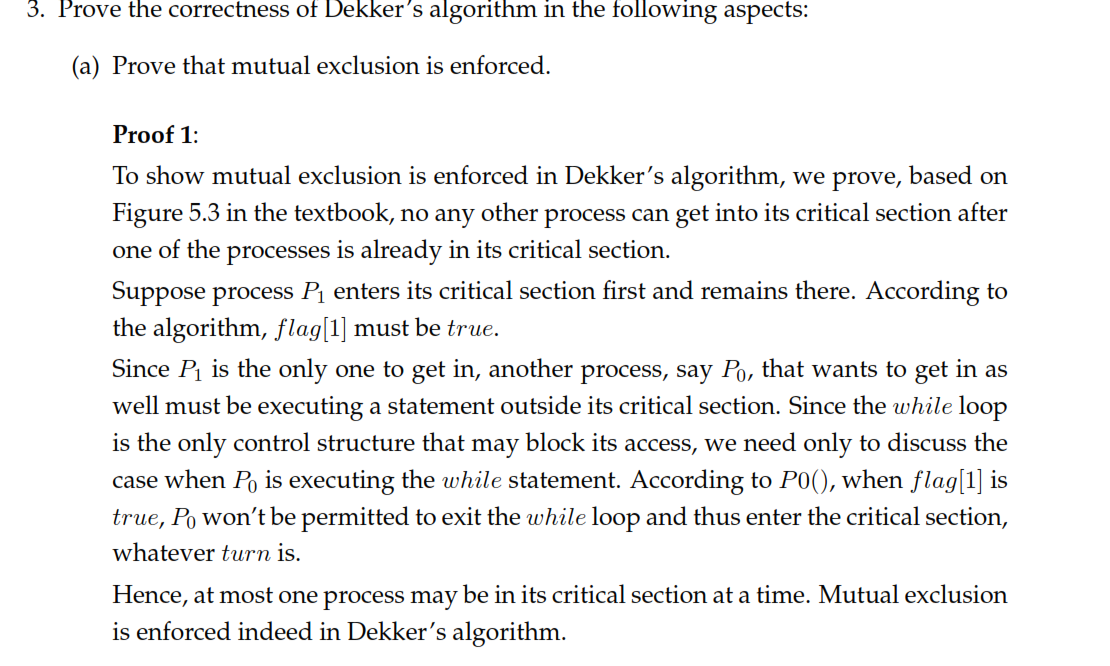
}

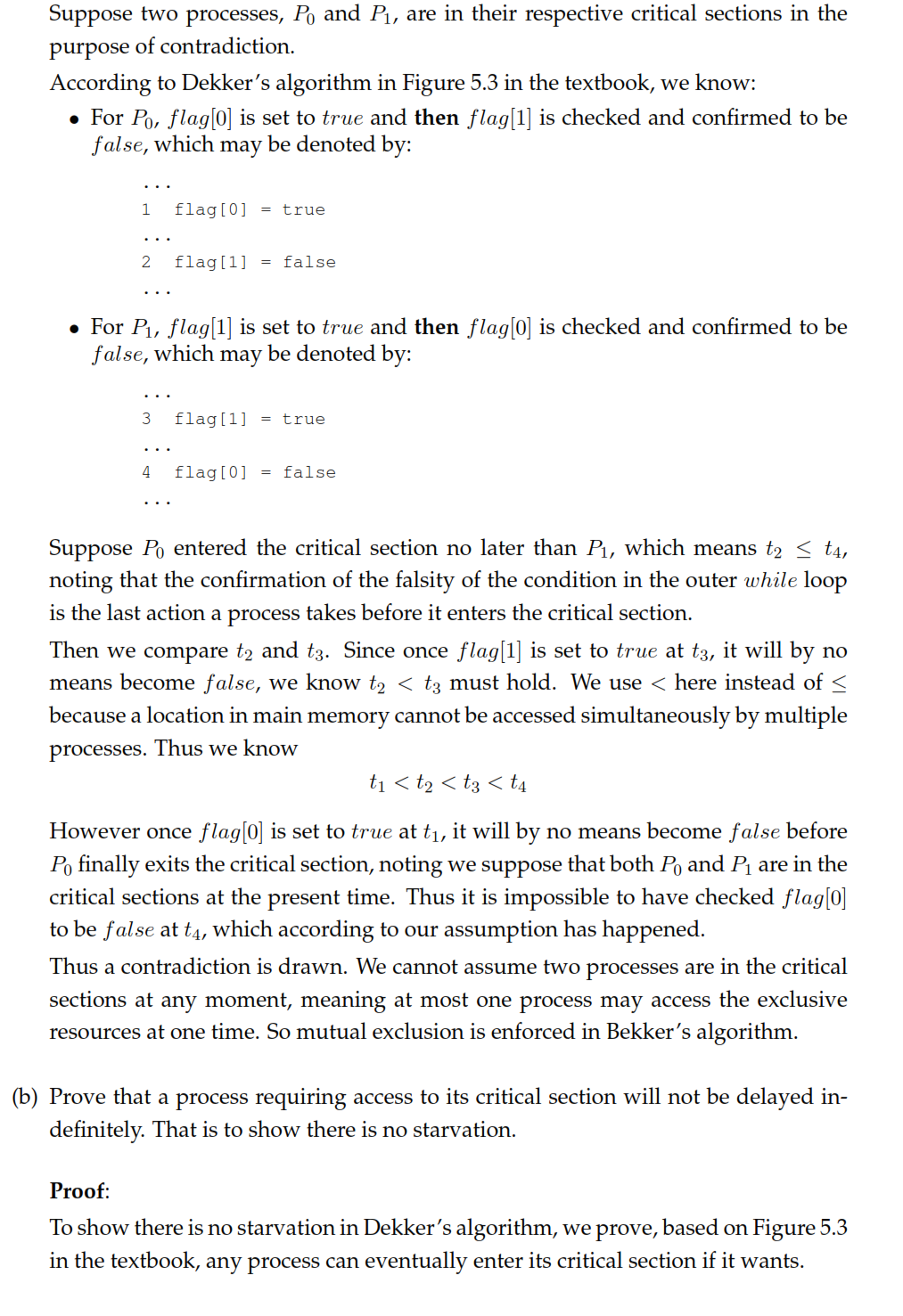
# Critical Section Problem: Dekker's Algorithm

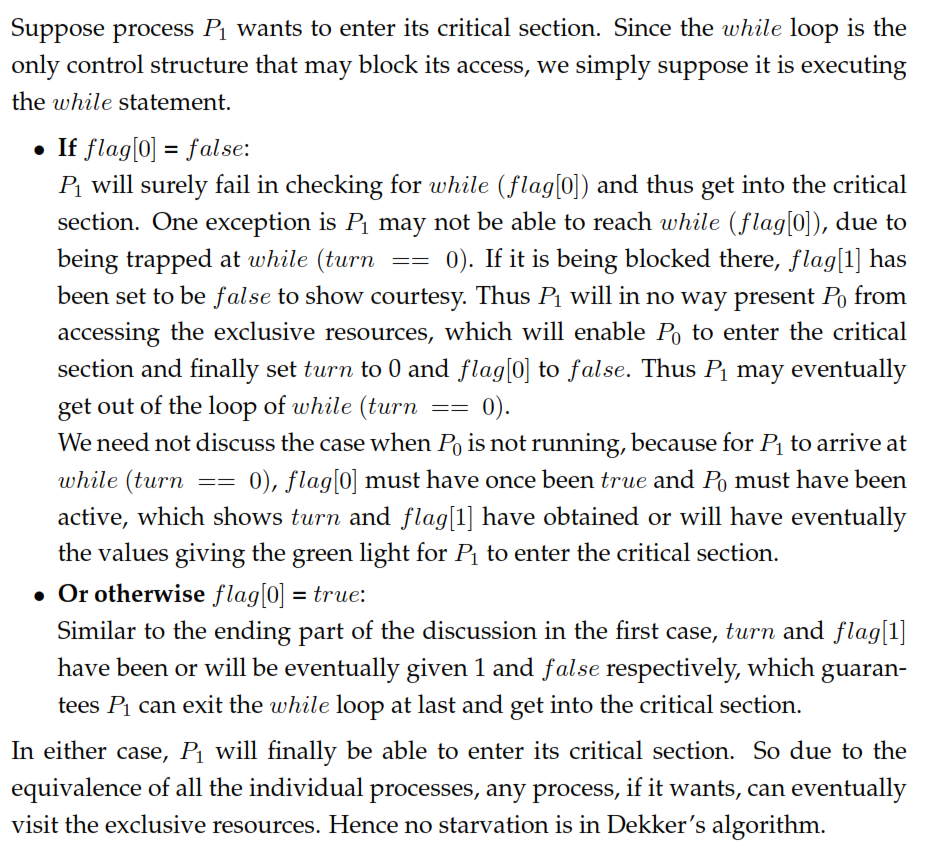
Dekker's algorithm was the first provably-correct solution to the critical section problem. It requires both an array of boolean values and an integer variable:

|  |
| --- |
| **var** flag: **array** [0..1] **of** boolean; turn: 0..1; |

|  |
| --- |
| **repeat**          flag[i] := true;         **while** flag[j] **do**                 **if** turn = j **then**                 **begin**                         flag[i] := false;                         **while** turn = j **do** no-op;                         flag[i] := true;                 **end**;                  critical section          turn := j;         flag[i] := false;                  remainder section  **until** false; |







CS 3750 Handout on an alternative to the Bakery Algorithm,

which does not use unique id numbers (at least not in the same

way as the Bakery Algorithm)

This Algorithm (Eisenberg and McGuire, 1972) Solves the

Critical Section Problem for N Processes.

Algorithm 5 :

The common data structures are:

var flag: array [0 .. n-1] of (idle, want-in, in-cs) ;

turn: 0 .. n-1 ;

All the elements of flag are initially idle, the initial value of "turn" is

immaterial (between 0 and n-1). The structure of process Pi is:

VAR j: 0..n ;

REPEAT

REPEAT

1 flag[i] := want-in ; (\* I tell everybody I want in \*)

(\* I wait until a sweep from "turn" around "clockwise" to me finds everyone

idle -- they have "priority" over me right now.\*)

2 j := turn ;

2 WHILE j <> i

2 DO IF flag[j] <> idle THEN j := turn ELSE j := j+1 mod n ;

(\* I tell everybody I am in the critical section. (but I'm NOT!) \*)

3 flag[i] := in-cs ;

(\* search the whole ring for someone besides me with flag = "in-cs" \*)

4 j := 0 ;

4 WHILE (j < n) AND ((j = i) OR (flag[j] <> in-cs)) DO j := j+1 ;

(\* if (the search fails) and ( (it's my turn) or (P\*, the process whos turn it

is, is idle)) then I go in to the CS. If not, I start over at 1. \*)

4 UNTIL (j >= n) AND ((turn = i) OR (flag[turn] = idle)) ;

5 turn := i ; (\* It is my turn, and I go in. \*)

6 .. CS ..

(\* I find the first process clockwise from "turn" who is not idle, and make it

be the turn of THAT process. (Note: "turn" is ME when I start this, and it

will still be MY turn after lines 7, if everyone else is idle.) \*)

7 j := turn+1 mod n ;

7 WHILE (flag[j] = idle) DO j := j+1 mod n ;

7 turn := j ;

8 flag[i] := idle ; (\* I tell everybody I am idle \*)

9 .. RS ..

UNTIL false ;

It is helpful to think of the processes as arranged by number in a circle like

this:

0

11 1

10 2

me 9 3

8 4

7 5 "turn"

6

Fig 1.

Proof of MUTUAL EXCLUSION in Algorithm 5:

As in the proof of algorithm 4, we assume that Pk enters its CS

at time Tk, REMAINS there for a "while" (0 time or more), and

is then joined there for the first time by Pm at time Tm. We

let Tm-set be the LAST time before Tm that Pm sets its flag to

"in-cs" in 3. Similarly, Tk-set is the LAST time before Tk

that Pk sets its flag to "in-cs".

----------------------------------------------------------------------

^ ^ ^

Tk-set Tk Tm

Fig. 2

Suppose that Tm-set < Tk-set (or that Tm-set = Tk-set). Then

during all the time between Tk-set and Tk (inclusive), Pm is

executing somewhere between 4 and 6, so flag[m]=in-cs during

all that time. On the other hand, since Pk is able to enter

the CS at time Tk, at SOME time between Tk-set and Tk, Pk

verifies in 4 that flag[m] <> in-cs! That is contradictory, so

we can conclude that Tk-set < Tm-set.

After Tk-set, there is no opportunity for Pk to change the

value of flag[k] until after Pk leaves the CS, some time after

Tm. Since Tm-set is after Tk-set, that implies that the value

of flag[k] is "in-cs" during all the time between Tm-set and

Tm. But then THAT contradicts the fact that in order for Pm to

enter the CS at time Tm, it must verify in 4 that flag[k] <>

in-cs at some time between Tm-set and Tm!

So there is NO chronological relationship among Tk, Tk-set, Tm,

and Tm-set that is logically consistent with their

definitions. We can therefore conclude that the assumption

upon which their definitions is based -- the coexistence of Pk

and Pm in the CS at time Tm -- is impossible. Proof of

PROGRESS in Algorithm 5:

Processes set their flag to "idle" in 8 before entering their

RS. The RS contains no code that sets any flags. As a result,

while executing in their RS, processes always have their flag

set to "idle". When the CS is empty and processes are

contending for entry, it is only the value of "turn", and flag

values <> idle that can prevent a contending process from

getting into its CS. Thus processes executing in their

remainder sections do not affect the decision.

Suppose that S is a subset of the n processes, and that P' is

the member of S whose number is closest to the value of "turn"

(measured by going clockwise to P' from "turn"). If the

members of S are contending to get into the CS, with no other

processes entering contention, and if the value of "turn" is

constant during this time, and if the decision as to who gets

into the CS is postponed long enough, then all the members of S

besides P' will eventually get stuck in the loop at 2.

After that, under the assumptions above, the only thing that

can postpone a decision to let P' into the CS is if a process

is still executing in the exit section (7-8) and has not yet

set its flag to "idle", with the result that P' is also being

forced to stay in the WHILE loop of 2. Of course, that process

will leave the exit section eventually (after executing a

bounded number of instructions).

Thus, if the postponement still persists, it can only then be

due to the fact that more processes are entering into

contention for the CS. (For example if turn = 4, P8 is stuck

in the WHILE loop in 2, P7 is entering the WHILE loop in 4, and

P5 begins to contend and succeeds in executing 1 before P7 gets

to the part of the WHILE loop in 4 where j = 5, then P5 has

effectively stopped P7 from entering the CS, and will enter

instead (unless P4 pulls that same trick on P5!).

But if the decision continues to be postponed long enough, then

sufficient time will pass so that ALL n processes are executing

in the entry section (1-4), and all the processes except P\*

(the process whose turn it is) are caught in the while loop of

2. Once that has occured, P\* will be able to get into the CS,

thus ending the postponement.

Proof of BOUNDED WAITING in Algorithm 5:

If there are processes contending to enter the CS, each process

Pi that leaves the CS advances "turn" to the NEXT CONTENDING

PROCESS after i in the cyclic ordering. This introduces a

"ratcheting" influence with a step of at least 1.

An argument very similar to one used in the Progress proof

above will show that if S is any subset of the n processes that

contains P\*, and if the processes in S contend for the CS until

one of them gets in, it will be P\* that gets in -- even if

other processes join the contention as it progresses.

If a process Pj begins to contend for the CS and then has to

yield to another process Pk, then Pk will designate its

successor (call it Pm) by setting "turn" after leaving the CS.

Since it will then be Pm's turn, Pm will be next to enter the

CS. Then, if there are still processes contending for the CS,

Pm will designate its successor into the CS by setting "turn".

This will go on at least as long as Pj remains in contention

for the CS.

After "turn" is set to k in 5, and as long as some process

remains "non-idle" in the entry section, line 5 will not change

the value of "turn" -- "turn" will only change in 7, where the

turn is given to the next "non-idle" process in clockwise

order. (Note that the first process to enter the CS after a

period of no contention MAY actually CHANGE the value of "turn"

in line 5, because P\* may have its flag set to "idle".)

Thus after process Pj has begun to contend to enter the CS and

has had to yield to Pk, each process that leaves the CS will

move the value of "turn" in a clockwise direction closer to j

by a step of at least one, WITHOUT GOING PAST j, and the result

will be that "turn" will be set to j after Pj has yielded to

another process no more than n-1 times, after which Pj will be

the next to enter the CS.

"PATHOLOGICAL" EXAMPLE:

0

7 1

6 2

5 3

4

Fig. 3

Initially all idle. Turn = 0.

P7, P5, P3, and P1 come into the entry section, in that order,

each getting to the end of part 2 and sleeping before the next

enters. Now any of P1, P3, P5, or P7 can wake up and execute

parts 3-5, set "turn", enter the CS, and then reset turn to the

value of the next contending process after it in the cyclic

ordering. If, say, P5 is the one to do that, then after P5

resets "turn", its value is 7. So the value of "turn" has been

allowed to jump from 0, to 5, to 7, completely bypassing 1 and

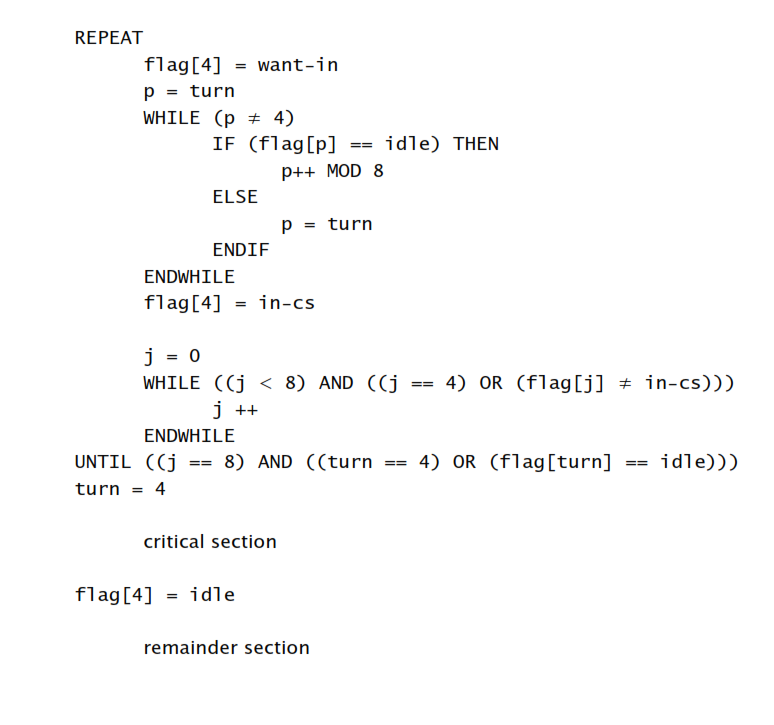
3, even though P1 and P3 are contending for the CS. Can this

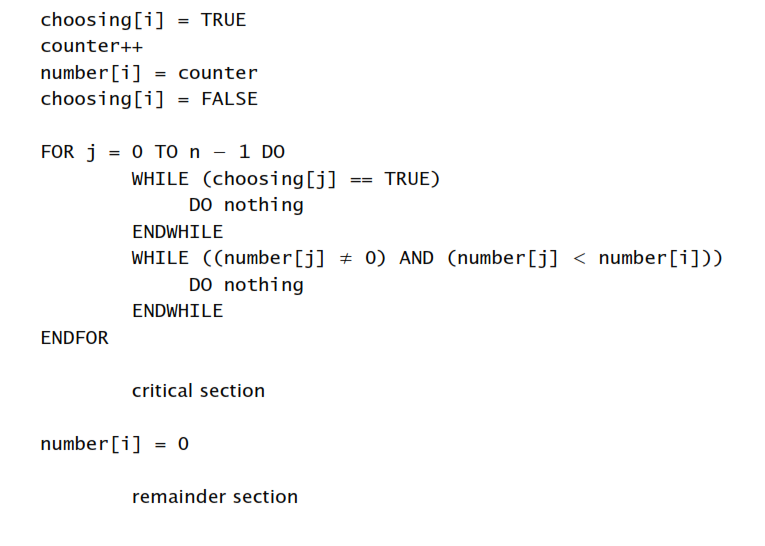
kind of jumping around of "turn" continue in such a way that P1

or P3 will have to yield the CS more than 7 times? The answer

is no. What prevents this from happening?

**Mcquire CSP Alogrithm**



**Lamport’s Bakery Algorithm for CSP**