Some more C functions for you: `sigprocmask pthread_sigmask pthread_self() atexit sigaction`

Candidate # 0 (Review) Protect our critical section with a mutex. But how should it work!?

pthread_mutex_lock(p_mutex_t* m) { while(m->lock) {}; m->lock = 1;}
pthread_mutex_unlock(p_mutex_t* m) { m->lock = 0; }

Problems?

Candidate # 1

wait until your flag is lowered raise my flag // *Do Critical Section stuff* lower my flag	wait until your flag is lowered raise my flag // *Do Critical Section stuff* lower my flag

// Threads do other stuff and then will repeat at sometime in the future

Problems with 1b? Fix?

Candidate #2

raise my flag wait until your flag is lowered // *Do Critical Section stuff* lower my flag	raise my flag wait until your flag is lowered // *Do Critical Section stuff* lower my flag

// Threads do other stuff and then will repeat at sometime in the future

Problems with 2?

Candidate #3

wait until my turn (turn==id?) // *Do Critical Section stuff* turn = *yourid*	wait until my turn (turn==id?) // *Do Critical Section stuff* turn = *yourid*

// Threads do other stuff and then will repeat at sometime in the future

Problems with 3?

Three desirable properties of the solution to the Critical Section Problem?

Candidate #4

raise my flag if your flag is raised, wait until my turn // *Do Critical Section stuff* turn = *yourid* lower my flag	raise my flag if your flag is raised, wait until my turn // *Do Critical Section stuff* turn = *yourid* lower my flag

// Threads do other stuff and then will repeat at sometime in the future
Problems with 4?

Candidate #5

raise my flag
while(your flag is raised) :
  if it's your turn to win :
  lower my flag
  wait while your turn
  raise my flag
// *Do Critical Section stuff*
set your turn to win
lower my flag
	raise my flag
	while(your flag is raised) :
	  if it's your turn to win :
	  lower my flag
	  wait while your turn
	  raise my flag
	// *Do Critical Section stuff*
	set your turn to win
	lower my flag

MYTHS ABOUT THE MUTUAL EXCLUSION PROBLEM

G.L. PETERSON
Department of Computer Science, University of Rochester, Rochester, NY 14627, U.S.A.

Received 13 January 1981; revised version received 30 March 1981

Recently in these pages appeared a report by Doran and Thomas [2] which gave partially simplified versions of Dekker-like solutions to the two process mutual exclusion problem with busy-waiting. This report presents a truly simple solution to the problem and attempts in a small way to dispel some myths that seem to have arisen concerning the problem.

Briefly, the *mutual exclusion problem* for two processes is to find sections of code (*trying protocol, exit protocol*) for each of two asynchronous processes to use when trying to enter and upon exiting their designated critical sections. The protocols must preserve mutual exclusion and not have deadlock or lockout. *Mutual exclusion* means that both processes can never be in their critical sections at the same time. No *deadlock* or *lockout* means that no process waits forever inside a protocol. More formal definitions can be found in [5] and elsewhere.

The original solution due to Dekker is discussed at length by Dijkstra in [1]. Of the many reformulations given since, perhaps the best appears in [3]. (Unfortunately the authors believe their correct solution is incorrect.) The solutions of Doran and Thomas are slight improvements which eliminate the 'loop inside a loop' structure of the previously published solutions. The solution presented here has an extremely simple structure and, as shown later, is easy to prove correct.

The protocols of P_1 and P_2 are given in Fig. 1. Q1 and Q2 are initially *false* and TURN may start as either 1 or 2. (The busy wait loop '*wait until* Boolean' is just another way of saying "*repeat*/* empty statement/* *until* Boolean". The Boolean formula is *not* evaluated atomically.)

As can be seen, the algorithm has a very simple structure. This results in an easy proof of correctness. First, neither process can be locked out. Consider P_1, it has only one wait loop, and assume it can be forced to remain there forever. After a finite amount of time, P_2 will be doing one of three general things: not trying to enter, waiting in its protocol, or repeatedly cycling through its protocols. In the first case, P_1 notes that Q2 is *false* and proceeds. The second case is impossible due to TURN being either 1 or 2; and one of the processes will proceed. In the third case P_2 will quickly set TURN to 2 and never change it back to 1, allowing P_1 to proceed.

If mutual exclusion were not preserved and both processes could somehow end up in their critical sections at the same time, then we have Q1 = Q2 = *true*. Their tests in their wait loops just prior to entering their critical sections at this point could not have been at approximately the same time as TURN would have been favorable to only one of the processes and the other part of the test would have failed for both. This

```
/*trying protocol for P1*/
Q1 := true;
TURN := 1;
wait until not Q2 or TURN = 2;
Critical Section;
/*exit protocol for P1*/
Q1 := false.
```

```
/*trying protocol for P2*/
Q2 := true;
TURN := 2;
wait until not Q1 or TURN = 1;
Critical Section;
/*exit protocol for P2*/
Q2 := false.
```