```
{
   while (state[j]  == inside);  /* is the other one inside? */

   state[i] = inside;  /* get in and flip state */

   <<< critical section >>>

   state[i] = outside;  /* revert state */

   <<< code outside critical section >>>
}
```

This proposal has some nice features:

- No blocking occurs unless the other process is inside of the critical section (progress criteria is satisfied).
- To the extent allowed by the scheduler, there is a guarantee that both processes will eventually be able to enter the critical region.

But we still don't have a solution. To understand why this code is incorrect, we must remember two things:

- The currently running process can be pre-ememented at any time -- leaving the current activity incomplete
- Murphy's law will ensure that a context switch will occur at the most embarassing of all possible times. If there is an occasion for a context-switch to break our code, Mr. Murphy will find it.

Atomicity is the property of being executed as a single unit. This algorithm assumes that the test of (state[1] == inside) and the set of (state[0] = inside) are atomic. That is to say, this algorithm assumes that nothing can come in-between those two operations.

That assumption is inaccurate. A *race-condition* exists between testing and setting state. P_0 can be pre-empted between the two operations, by P_1. The result will be that P_1 will test state[0], find it false, and enter the critical section.

Consider the following trace:

1. P_0 finds (state[1] == outside)
2. The scheduler forces a context-switch
3. P_1 (finds state[0]==outside)
4. P_1 sets (state[0] = inside)
5. P_1 enters the critical section
6. The scheduler forces a context-switch
7. P_0 sets (state[1] = inside)
8. P_0 enters the critical section
9. **Both P_0 and P_1 are now in the critical section**

With both processes in the critical section, the mutual exclusion criteria has been violated.

Algorithm #3 (Incorrect)

Let's try again. This time, let's avoid the race-condition by expressing our intent first, and then checking the other process's state:

```
/* i is this process; j is the other process */

while (true)
{
   state[i] = interested;  /* declare interest */

   while (state[j]  == interested);  /* stay clear till safe */

   <<< critical section >>>

   state[i] = notinterested;  /* we’re done */

   <<< code outside critical section >>>
}
```

Okay. This does guarantee mutual exclusion, but not bounded wait. This approach allows a *livelock*. A *livelock* is a special type of deadlock, where the affected processes are consuming (wasting) CPU cycles by looping forever.

Consider the following trace:

1. P_0 sets state[0] to interested
2. A context-switch occurs
3. P_1 sets state[1] to interested
4. P_1 loops in while
5. A context-switch occurs
6. P_0 loops in while

Both P_0 and P_1 loop forever. This is the livelock.

Algorithm #4: Peterson's Algorithm (Correct)

This time, let's try using Algorithm #3, but taking turns to break ties:

```
/* i is this process; j is the other process */

while (true)
{
   state[i] = interested;  /* declare interest */
   turn = j;  /* be nice to other guy */

   while (state[j] == interested && turn == j);

   <<< critical section >>>

   state[i] = notinterested;  /* we’re done */

   <<< code outside critical section >>>
}
```

This code satisfies all three properties:

- mutual exclusion
- progress
- bounded wait