Busy Waiting as synchronization

It is hard to discover a solution to the critical section problem through trial and error. However, theoreticians were already developing algorithms assuming that synchronization was possible. To specify algorithms, the following statements were "invented" to help specify the "blocking" needed to specify concurrent programs and to allow "atomic" actions:

await – specifies that the process somehow is blocked at this point
the angle brackets < > which specify the actions we want performed as an "atomic" action

With these, we can now specify a pseudo-code solution for the critical section problem:

```
boolean in1=false, in2=false;
// MUTEX: not (in1 and in2)

P1::
   while (true) {
      < await (! in2) { in1 = true; } >
      critical_section;
      in1 = false;
      non_critical_section;
   }
```

```
P2::
   while (true) {
      < await (! in1) { in2 = true; } >
      critical_section;
      in2 = false;
      non_critical_section;
   }
```

We see with the above that we can easily specify a solution to the critical section problem. This solution employs 2 variables. To generalize, we'd have to use *n* variables. However, there are only two states that we are interested in: some process is in the CS, or no process is in the CS. One variable should therefore be necessary to distinguish between these two states, independent of the number of processes.

```
boolean lock= false;

P1::
   while (true) {
      < await (! lock) { lock = true; } >
      critical_section;
      lock = false;
      non_critical_section;
   }
```

```
P2::
   while (true) {
      < await (! lock) {  lock = true; } >
      critical_section;
      lock = false;
      non_critical_section;
   }
```

The significance of this change is that almost all machines, especially multiprocessors, have some special instruction that can be used to implement the conditional atomic actions above. As an example, the instruction TS (Test and Set).

TestAndSet

The TS instruction takes two boolean arguments: a shared lock and a local condition code cc. As an atomic action, TS sets cc to the value of lock, then sets lock to true:

```
TS(lock, cc):  < cc = lock; lock = true >
```

Using TS, we can implement a solution to the CS problem.

```
boolean lock = false;
```

```
P1::
     bool cc;
     while (true) {
        non_critical_section;
        TS(lock, cc)
        while (cc) TS(lock, cc);
        critical_section;
        lock = false;
     }
```

```
P2::
     bool cc;
     while (true) {
        non_critical_section;
        TS(lock, cc)
        while (cc) TS(lock, cc);
        critical_section;
        lock = false;
     }
```

When a solution to the CS problem uses an instruction like TS, the scheduling must be strongly fair to ensure eventual entry. This is a strong requirement; two or more processes could always be contending for entry. The spin lock does not does not control the order in which delayed processes enter their critical sections if two or more are trying to do so.

The two process tie-breaker algorithm (Peterson's algorithm)

```
bool in1 = false, in2= false;
int last= 1;
```

```
P1::
   while (true) {
      non critical section;
      // entry protocol
      in1 = true; last = 1;
      < await not in2 or last == 2 >
      critical section
      in1= false;
   }
```

```
P2::
   while (true) {
      non critical section;
     // entry protocol
      in2= true; last = 2;
      < await not in1 or last == 1 >
      critical section;
      in2= false;
   }
```

The problem here is that each "await" statement references the variables altered in the other process. In this particular case, however, it is not necessary that the delay conditions be evaluated atomically for the following informal reasons:

Consider the await statement in P1. If the delay condition is true, either in2 is false or last is 2. The only way P2 could make the delay condition false is if it executes the first statement in its entry protocol, which sets in2 to true. But then the next action of P2 is to make the condition true again by setting last to 2. Thus, when P1 is in its critical section, it can be assured the P2 will not be in its critical section. The argument for P2 is symmetrical → thus the following solution:

```
bool in1= false, in2= false; int last = 1;
```

```
P1::
   while (true) {
      non critical section
      // enter CS
      in1= true; last= 1;
      while (in2 and last == 1)
         ;
      critical section
      // exit CS
      in1= false;
   }
```

```
P2::
   while (true) {
      non critical section
      // enter CS
      in2= true; last = 2;
      while (in1 and last == 2)
         ;
      critical section
      // exit CS
      in2= false;
   }
```

See also Sections 5.4 and 5.5 of the JR Programming Language Book: The Bakery Algorithm