## Solution for Critical Section

There are several approaches used for solving critical section:

1- Algorithmic Approach
2- Hardware Approach
3- Semaphore

### 1-Algorithmic Approaches

We first attempt to solve this for two processes, P0 and P1. They share some variables to synchronize. We fill in <CS Entry> and <CS Exit> from above with code that should satisfy the three conditions.

### Algorithmic Approaches for 2 Processes

### Critical Section Algorithm 1

• Shared data

```
int turn=0;
```

• Process Pi (define j = 1 − i, the other process)

```
while (1) {
        while (turn!=i); /* busy wait */

          /* critical section */

            turn=j;

          /* non-critical section */
      }
```

Note the semicolon at the end of the while statement's condition at the line labeled "busy wait" above. This means that Pi just keeps comparing turn to i over and over until it succeeds. This is sometimes called a ***spin lock***. For now, this is our only method of making one process wait for something to happen. This does satisfy mutual exclusion, but not progress (alternation is forced).

### Critical Section Algorithm 2

We’ll avoid this alternation problem by having a process wait only when the other has “indicated interest” in the critical section.

• Shared data
```
boolean flag[2];
flag[0]=false;
flag[1]=false;
```

• Process Pi
```
while (1) {
    flag[i]=true;
    while (flag[j]);

        /* critical section */

    flag[i]=false;

        /* non-critical section */
}
```

flag[i] set to true means that Pi is requesting access to the critical section. This one also satisfies mutual exclusion, but not progress. Both can set their flags, then both start waiting for the other to set flag[j] back to false. Not going to happen...
If we swap the order of the flag[i]=true; and while (flag[j]); statements, we no longer satisfy mutual exclusion.

**Critical Section Algorithm 3**
We combine the two previous approaches:

• Shared data
```
int turn=0;
boolean flag[2];
flag[0]=false;
flag[1]=false;
```

• Process Pi
```
while (1) {
    flag[i]=true;
    turn=j;
    while (flag[j] && turn==j);

        /* critical section */
```

```
                        flag[i]=false;

                          /* non-critical section */
}
```

So, we first indicate interest. Then we set turn=j;, effectively saying “no, you first” to the other process. Even if both processes are interested and both get to the while loop at the “same” time, only one can proceed. Whoever set turn first gets to go first. This one satisfies all three of our conditions. This is known as *Peterson’s Algorithm*.

## Algorithmic Approach for n Processes: Bakery algorithm

Can generalize this for n processes? The Bakery Algorithm. The idea is that each process, when it wants to enter the critical section, takes a number. Whoever has the smallest number gets to go in. This is more complex than the bakery ticket-splitters because two processes may grab the same number (to guarantee that they wouldn’t would require mutual exclusion – exactly the thing try to implement), and because there is no attendant to call out the next number – the processes all must come to agreement on who should proceed next into the critical section.
We break ties by allowing the process with the lower process identifier (PID) to proceed. For Pi, we call it i. This assumes PIDs from 0 to n − 1 for n processes, but this can be generalized. Although two processes that try to pick a number at about the same time may get the same number, we do guarantee that once a process with number k is in, all processes choosing numbers will get a number > k.
Notation used below: an ordered pair (number, pid) fully identifies a process’ number. We define a *lexicographic order* of these:

• (a, b) < (c, d) is a < c or if a = c and b < d

The algorithm:
• Shared data, initialized to 0’s and false

```
   boolean choosing[n];
   int number[n];
```

• Process Pi

```
     while (1) {
                 choosing[i]=true;
```

```
        number[i]=max(number[0],number[i],...,number[n-1])+1;
        choosing[i]=false;
        f or (j=0; j<n; j++) {
                    while (choosing[j]);
                        while ((number[j]!=0) &&
((number[j],j) < (number[i],i)));
                }

                            /* critical section */

                number[i]=0;

                    /* non-critical section */
    }
```

Before choosing a number, a process indicates that it is doing so. Then it looks at everyone else's number and picks a number one larger. Then it says it's done choosing. Then look at every other process. First, wait for that process not to be choosing. Then make sure we are allowed to go before that process. Once we have successfully decided that it's safe to go before every other process, then go!
To leave the CS, just reset the number back to 0.
we have a solution. But...problems:

1. That's a lot of code. Lots of while loops and for loops. Could be expensive if we're going to do this a lot.
2. If this is a highly popular critical section, the numbers might never reset, and we could overflow our integers. Unlikely, but think what could happen if we  did.
3. It's kind of inconvenient and in some circumstances, unreasonable, to have these arrays of n values. There may not always be n processes, as some may come and go.