



# Operating Systems CS F372

## Synchronization

**BIJU K RAVEENDRAN**

# Peterson's Solution – Algorithm 3

- Not guaranteed to work on modern architectures! (But good algorithmic description of solving the problem)
- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int **turn**;
  - Boolean **flag[2]**
- The variable **turn** indicates whose turn it is to enter the critical section.
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i]** = true implies that process  $P_i$  is ready!



# Algorithm 3

- Combined shared variables of algorithms 1 and 2.
- Process  $P_i$   
do {

```
flag [i] = true;  
turn = j;  
while (flag [j] && turn == j);  
    critical section  
flag [i] = false;  
    remainder section
```

```
}while(1);
```

# Algorithm 3

Shared variable Boolean flag[2] (flag[0] & flag[1] initial value false) and turn (initial value 0)

## Process 0

```
do {  
    flag[0]=true; turn = 1;  
    while (flag[1] && turn ==1);  
        // do nothing  
    <critical section>  
    flag[0]=false;  
    <reminder section >  
}while(1);
```

## Process 1

```
do {  
    flag[1]=true; turn = 0;  
    while (flag[0] && turn ==0);  
        // do nothing  
    <critical section>  
    flag[1]=false;  
    <reminder section >  
}while(1);
```

- Meets all three requirements
- Solves the critical-section problem for two processes.



# Peterson's Solution – Algorithm 3

- Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

$P_i$  enters CS only if:

either `flag[j] = false` or `turn = i`

2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

# Peterson's Solution – Algorithm 3

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
- Understanding why it will not work is also useful for better understanding race conditions.
- To improve performance, processors and/or compilers may reorder operations that have no dependencies.
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!



# Bakery Algorithm

- Critical section for 'n' processes
  - Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
  - If processes  $P_i$  and  $P_j$  receive the same number, if  $i < j$ , then  $P_i$  is served first; else  $P_j$  is served first.
  - The numbering scheme always generates numbers in non-decreasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...




# Bakery Algorithm

- Notation
  - lexicographical order(ticket #, process id #)
  - $(a,b) < (c,d)$  if  $a < c$  or if  $a = c$  and  $b < d$
  - $\max(a_0, \dots, a_{n-1})$  is a number,  $k$ , such that  $k \geq a_i$  for  $i = 0, \dots, n - 1$
- Shared data
  - `int number[n];` // initialized to 0



# Bakery Algorithm



```
do {  
    number[i] = max(number[0], number[1], ...,  
                    number [n - 1])+1;  
    for ( j = 0 ; j < n ; j++ ) {  
        while ( number[j] != 0) &&  
            ( number [ j , j ] ) < ( number [ i , i ] ) ;  
    }  
    CRITICAL SECTION  
    number[i] = 0 ;  
    REMAINDER SECTION  
} while(1) ;
```

# Synchronization Hardware

- Hardware is faster than the software & can have better efficiency
- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption until it invokes an operating system service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion
  - Processor is limited in its ability to interleave programs

