Bakery Algorithm

Multiple Process Solutions

- Peterson’s solution solves the critical-section problem for two processes, in software
- For multiple processes we have “Bakery Algorithm”
- Used in bakeries etc

Bakery Algorithm

- The basic idea is that of a bakery
- On entering the bakery, Customers take tokens
- Whoever has the lowest token gets service next.
- “Service" means entry to the critical section.

Bakery Algorithm

- **`int token[n];`**
- **`token[0]`** = token given to Process 0
- **`token[1]`** = token given to Process 1
- …
- …
- **`token[n-1]`** = token given to Process n – 1

Algorithm

```
while(TRUE)
{
 //1. Receive a token
 token[OwnID]= max(token[0                   [n-1])+1;
 //2. Wait for turn
 for (OthersID = 0;OthersID<n;OthersID++)
   while(token[OthersID]!=0 &&(token[OthersID],OthersID)<
   (token[OwnID],OwnID));
 //3. Enter Critical section
 critical_section();
 //4. Leave Critical Section
 token[OwnID] = 0;
}
```

Why multiple waits?

Because, have to wait for multiple processes

Receive a token

- Initially token[0] .. token[n-1] are set to zero
- Process i chooses token[i] as
  - **`max(token[0],token[1],...,token[n-1]) + 1;`**
- Let n = 5;
- Let the order of execution be P0,P3,P4,P1,P2,P3,P4…
- P0 gets token[0] = max(0,0,0,0,0) + 1 = 0+1=1
- P3 gets token[3] = max(1,0,0,0,0) + 1 = 1+1=2
- P4 gets token[4] = max(1,0,0,2,0) + 1 = 2+1=3
- P1 gets token[1] = max(1,0,0,2,3) + 1 = 3+1=4
- P2 gets token[2] = max(1,4,0,2,3) + 1 = 4+1=5
- P3 gets token[3] = max(1,4,5,2,3) + 1 = 5+1=6
- P4 gets token[4] = max(1,4,5,6,3) + 1 = 6+1=7

Wait for turn

- Pi waits until it has the lowest token of all the processes waiting to enter the critical section.
- Bakery Algorithm does not guarantee that two processes do not receive the same token
- In case of a tie, the process with the lowest ID is served first.

```
for (OthersID = 0;OthersID < n ; OthersID++)
   while(token[OthersID]!=0 &&(token[OthersID],OthersID)<
   (token[OwnID],OwnID));
```

- (a,b) < (c,d) = TRUE if a < c or if both a = c and b < d
- token[OwnID] = 0 => Process is not trying to enter the critical section

```
while(TRUE)
{//1. Receive a token
 token[OwnID]=   max(token[0],token[1],..,token[n-1])+1;
```

token[0] = 1

```
//2. Wait for turn
  for (OthersID = 0;Ot                   ID++)
```

T && F

```
       while(token[OthersID]!=0
&&(token[OthersID],OthersID)< (token[OwnID],OwnID));
//3. Enter Critical Section
  critical_section();
```

Timeout

```
while(TRUE)
{//1. Receive a token
 token[OwnID]=   max(token[0],token[1],..,token[n-1])+1;
```

token[1] = 1

```
//2. Wait for turn
  for (OthersID = (                ersID++)
```

F && T

```
       while(token[OthersID]!=0 &&(token[OthersID],OthersID)<
(token[OwnID],OwnID));
//3. Enter Critical Section
  critical_section();
```

Bakery Algorithm

```
while(TRUE)
{
 //1. Receive a token
 choosing[OwnID] = true;
 token[OwnID]= max(token[0],token[1],..,token[n-1])+1;
 choosing[OwnID] = false;
 //2. Wait for turn
 for (OthersID = 0;OthersID<n;OthersID++)
  while(choosing[OthersID]);
  while(token[OthersID]!=0 &&(token[OthersID],OthersID)<
  (token[OwnID],OwnID));
 //3. Enter Critical section
 critical_section();
 //4. Leave Critical Section
 token[OwnID] = 0;
}
```

Bakery Algorithm

- The reason for `choosing` is to prevent the second **while** loop being *entered* when process $P_{OthersID}$ is setting its `token[OthersID]`.
- `choosing[OthersID]` is true if $P_{OthersID}$ is choosing a token.
- If a process $P_{OthersID}$ is choosing a token when Pi tries to look at it, Pi waits until $P_{OthersID}$ has done so before looking