EE5903 RTS
Chapter 6
Real-Time Process Synchronization

Bharadwaj Veeravalli

elebv@nus.edu.sg

Contents

- Quick review on RTOS – Process/tasks & threads
- Understanding the need for Synchronization
- Producer-Consumer Paradigm
- Software Solutions
- Hardware Solution
- Semaphores & their use;
- Use of Semaphores to solve - Critical Section Problem, Producer-Consumer Problem
- Deadlock & Starvation
- Example – Dining Philosopher's Problem
- RTS Resource Sharing Protocols

Quick review on RTOS – process/tasks/threads

A process (synonymously called " task " throughout in earlier chapters) is an abstraction of a running program and is the logical unit of a work schedulable by the real-time operating system (RTOS).
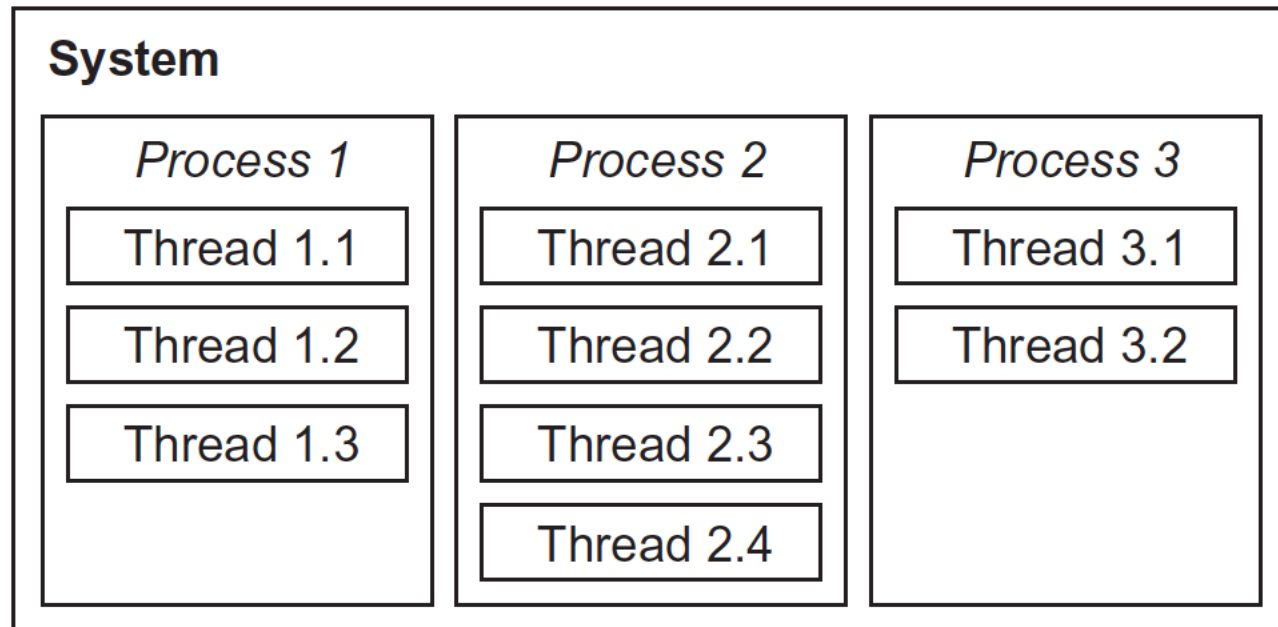
A process is usually represented by a private data structure that contains at least *an identity, priority level, state of execution* (e.g., running, ready, or suspended), *and resources* associated with the process.

A thread is a lightweight process that must reside within some regular process and make use of the resources of that particular process only.

> -- Multiple threads that reside logically within the same process may share resources with each other.
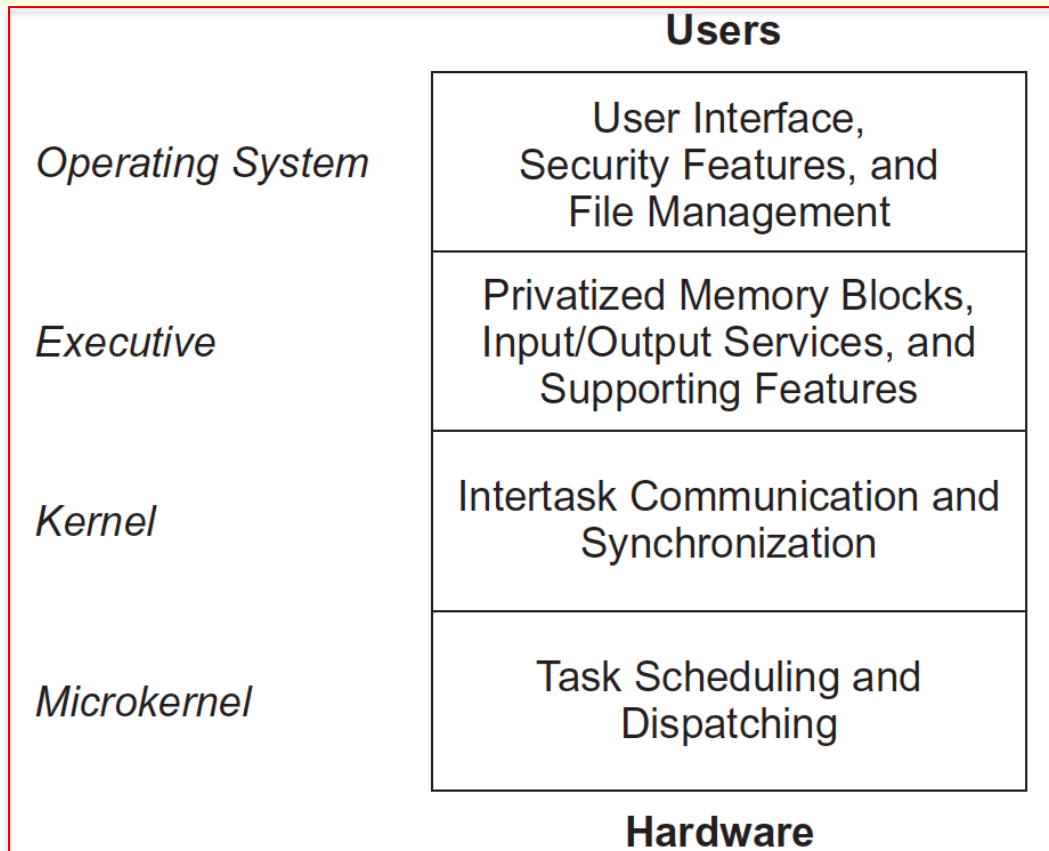
Quick review on RTOS– process/tasks/threads

Thus, processes are active participants of system-level multitasking, whereas threads can be seen as members of process-level multitasking. Figure below shows the hierarchy.

System		
Process 1	*Process 2*	*Process 3*
Thread 1.1	Thread 2.1	Thread 3.1
Thread 1.2	Thread 2.2	Thread 3.2
Thread 1.3	Thread 2.3	
	Thread 2.4	

Quick review on RTOS – process/tasks/threads

RTOS provide three essential functions with respect to software tasks:

- Scheduling, Dispatching, Inter-task Communication and Synchronization.

	Users	
Operating System	User Interface, Security Features, and File Management	
Executive	Privatized Memory Blocks, Input/Output Services, and Supporting Features	
Kernel	Intertask Communication and Synchronization	
Microkernel	Task Scheduling and Dispatching	
	Hardware	

Need for Synchronization

Multitasking systems are usually concerned with resource sharing.

In most cases, the resources can only be used by a single task at a time, and use of the resources cannot be interrupted. (**)

Such resources are said to be serially reusable, and they include certain peripherals, shared memory, and also the CPU.

Consequently, when processes / tasks compete for resources and execute a "shared logic" on a "shared resource", inconsistencies may result in the generated output values, if they are not properly scheduled!

This calls for proper *synchronization mechanisms* between processes / tasks & threads, whichever granularity is of concern.

(**) – Note that priority of a task is not of a concern here while it is using a resource. When a task uses a resource in an exclusive mode, it can never be interrupted regardless of its priority.

6

Producer-Consumer Paradigm

- This is an example of a situation where two competing processes attempt to access a shared variable and hence end up with inconsistent values, leading to an abnormal behavior!

- A Producer is a process which generates relevant outputs and a Consumer is a process that attempts to consume those outputs generated by the producer.

Producer-Consumer Paradigm (Cont'd)

- Variables required to define the problem

```
#define BUFFER_SIZE 10
typedef struct {
    DATA             data;
} item;
item   buffer[BUFFER_SIZE];
int    in = 0;                 // Location of next input to buffer
int    out = 0;                // Location of next removal from buffer
int    counter = 0;            // Number of buffers currently full
```

in

out
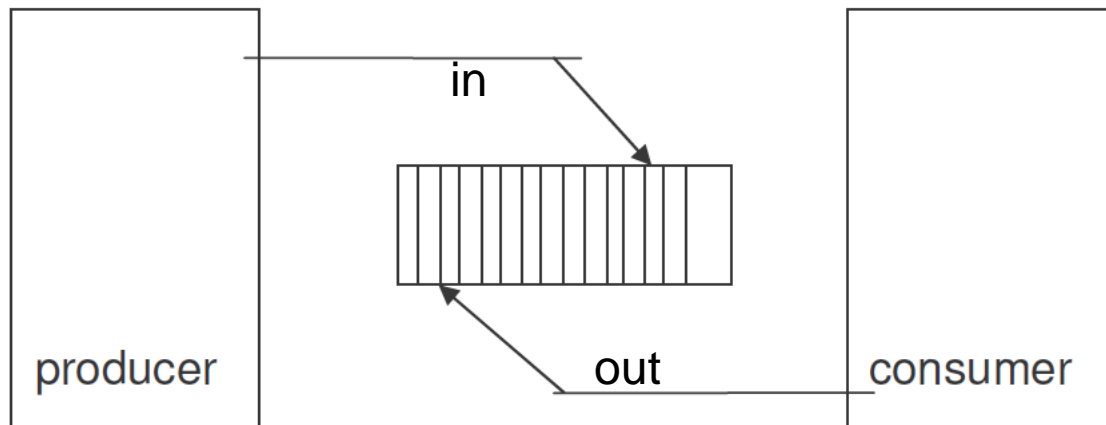
producer

consumer

Note:
counter – shared variable

Producer-Consumer Paradigm (Cont'd)

Code for Producer:

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced; /* Else */
        in = (in + 1) % BUFFER_SIZE;
        counter++; /* counter – freely  updated without  any bounds;
}
```

Producer-Consumer Paradigm (Cont'd)

Code for Consumer:

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
         counter--;
        /* consume the item in next_consumed */
}
```

Producer-Consumer Paradigm (Cont'd)

- Potential problem – Race Condition *- Where?*
- At the <u>lower levels of execution</u> (Assembly level) Counter++ and Counter-- statements are implemented using 3 machine instructions as shown below:

Register1 = counter   // current value of the counter

Register1 = Register1+1

counter = Register1

Register2 = counter   // It is possible that system uses Reg1 alone!

Register2 = Register2 – 1

counter = Register2   *What exactly is the problem?*

Producer-Consumer Paradigm (Cont'd)

Let us run and observe the sequence of actions taking place.

Sample runs to demonstrate inconsistency and inaccuracy. Assume at this current time t, we have **5** items in the buffer currently;

Specifically pay attention to where context-switch happens and the impact of context-switch on the outcome of the results as seen by the two processes.

Assume that Prod() has just generated an item and loaded to the buffer and it needs to perform count++;

When a context switch happens, Cons() will retrieve an item from the buffer and it needs to perform count--;

Producer-Consumer Paradigm (Cont'd)

Run trace (from machine instruction execution perspective):

TO;	Producer	Execute	register1 = counter	register1 = 5
T1;	Producer	Execute	register1 = register1 + 1	register1 = 6
T2;	Consumer	Execute	register2 = counter	register2 = 5
T3;	Consumer	Execute	register2 = register2 - 1	register2 = 4
T4;	Producer	Execute	counter = register1	counter = 6
T5;	Consumer	Execute	counter = register2	counter = 4

1) After T1 context switch happens;
2) After T3 execution context switch happens;
3) After T4 context switch happens;

Clearly the above working style and the final result is unacceptable!

Q: Can we restrict the order in which the variables (shared ones) are accessed without much compromising the speed performance?

Producer-Consumer Paradigm – Control strategy on Critical Sections of the codes

- **Solution** - To introduce a control mechanism to allow access to the shared variables for all the competing processes;

- **Critical Section (CS)** – A section of the code that has shared variables and/or lines of code that will be used by all competing processes in a cooperative execution environment;

- As seen earlier on the potential problems, we attempt to set forth some **rules** that all processes have to follow during shared variables/code access

Producer-Consumer Paradigm – Control strategy on Critical Sections of the codes (Cont'd)

Design Rule 1:

- A CS part of the code must have the following three sections.

A Critical Section Environment contains:

Entry Section	Code requesting entry into the critical section.
Critical Section	Code in which only one process can execute at any one time.
Exit Section	The end of the critical section, releasing or allowing others in.
Remainder Section	Rest of the code AFTER the critical section.

Producer-Consumer Paradigm – Control strategy on Critical Sections of the codes (Cont'd)

Design Rule 2:

■Use of Rule 1 must enforce all the 3 following properties to hold at all times for all competing processes.

(a) Mutual Exclusion – No more than one process can be present in the CS at any time;

(b) Progress – If no one is in the CS section then the selection of the next process that can enter CS cannot be delayed indefinitely;

(c) Bounded waiting time - No indefinite waiting of any process to gain access to CS

CS 3 properties - description

Mutual Exclusion – If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

■Points to ponder (Design considerations):

➢ Critical sections better be focused and short.

➢ Better to avoid infinite loop(s) in the CS

➢ If a process somehow halts/waits in its critical section, it must not interfere with other processes.

CS 3 properties - description

Progress – If no process is executing in its critical section and there exist some processes that wish to enter CS, then *the selection of the process that will enter the critical section next cannot be postponed indefinitely*:

- If only one process wants to enter, it should be allowed.

- If two or more processes want to enter, one of them should succeed.
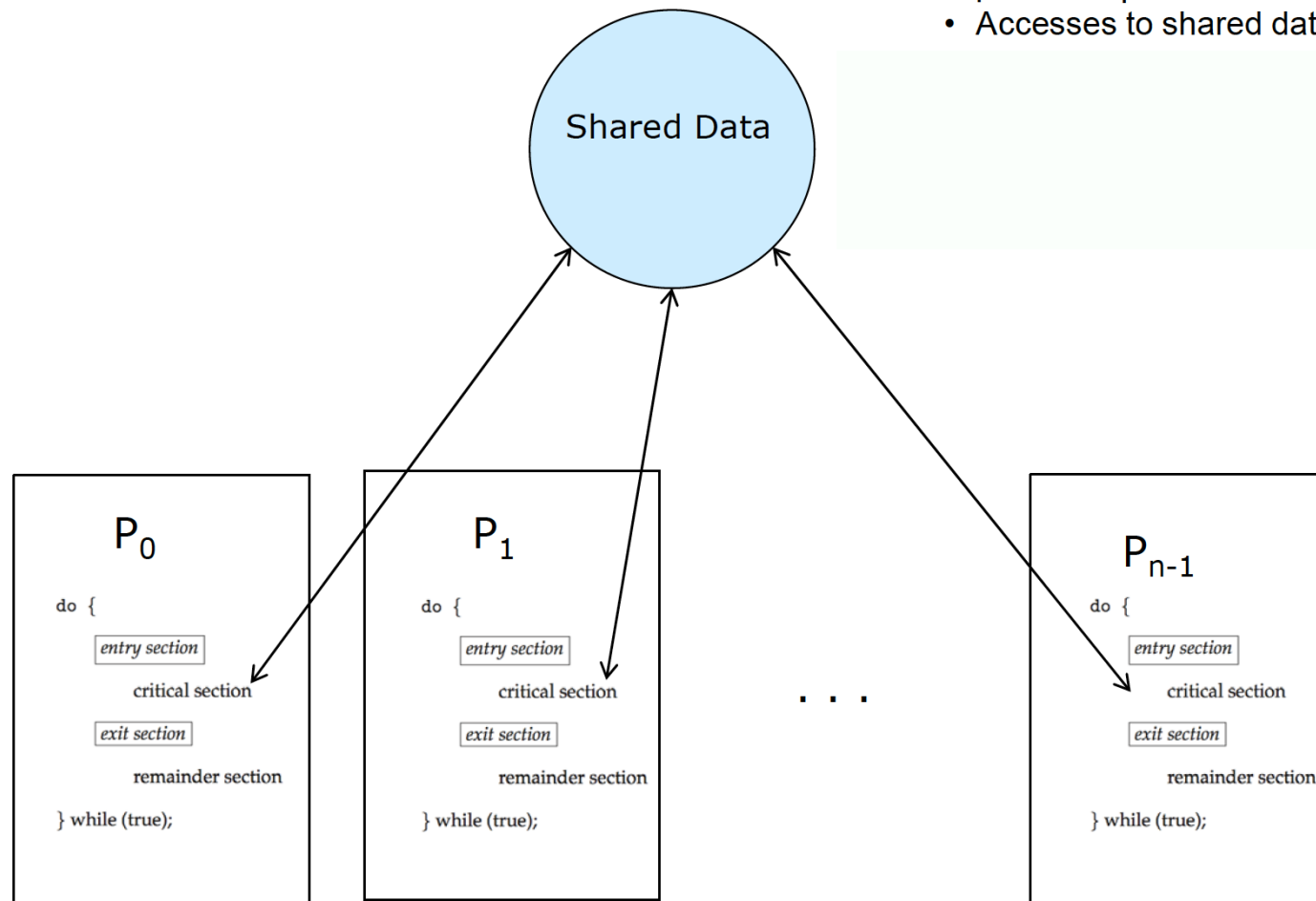
CS 3 properties - description

Bounded Waiting – A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

- Assume that each process executes at a nonzero speed.
- No assumption concerning relative speed of the n processes.

Note: Progress and Bounded waiting are distinct and must not be confused.

Producer-Consumer Paradigm – Control strategy on Critical Sections of the codes (Cont'd)

- Each process P_i has its own critical section
  - Accesses to shared data only in CS

Shared Data

P_0

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

P_1

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

. . .

P_{n-1}

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Software Solution: Attempt -1

■ Solution for two Processes i and j

We use a shared variable "turn" to exercise control - "turn" variable lets us to decide who goes next!

Note: Once turn is set by a process it is locked until it is released by that process!

```
turn = i; // proc i qualifies to enter its CS
Process i:  // This is code for process i
while (true) {
    while(turn != i) no_op; // Entry Code
     CS section here;
     turn = j; // Exit code
    Remainder section;
}
```

In Process j, this statement will be while(turn != j)

Q 6.1: *What is the problem with this Solution 1?*

21

Problem with Solution 1:

- *Satisfies mutual exclusion by providing toggled access;* Progress & (bounded waiting time, in a way) property is NOT satisfied; How?

- Simple Logic: If Pj exits by setting turn to i and terminates, it cannot enter again, because it sets turn to i and leaves; If Pj terminates/suspends before exit code there is no way to set turn to i.

- Thus Process i cannot enter again! Thus, an irrelevant or non-existing process can block other processes from entering a critical section. So *progress of process i is inhibited*;

Software Solution: Attempt -2

■ We use a <u>shared array</u> of variables flag[] to exercise control; flag[i]=true (Proc i is ready and can enter CS)

// flag[i] – indicates the state of P_i

Process **i**: // code for process i

```
while (true) {
    flag[i] = true;
    while(flag[ j ])
        no_op; // i will wait for j
    CS section here /*else */
    flag[i] = false;
    Remainder section
}
```

Flag[i] is an expression of its interest!

In Process j, these statements will be flag[j]=true and while(flag[i])

Q 6.2: *What is the problem with this Solution 2?*

23

(c) Bharadwaj V 2022

Problem with Solution 2

- Can mutually block indefinitely. *Mainly bounded waiting time property is not satisfied!*

- Both i and j can set flag[] to true (after the main while(true) statement) and then both get blocked and remain busy-waiting;

Software Solution: Attempt -3

■ We use a shared array of variables flag[] to exercise control; flag[i]=true means Proc i can enter CS;

Process **i**: // code for process i
while (true) {

       while(flag[j]) *no_op*;

       flag[i] = true;

       // CS section here

       flag[i] = false;

       // remainder section here

}

Check flag j first and then express your interest!

Q 6.3: *What is the problem with this Solution 3?*

Problem with Solution 3

■ *Does not satisfy Mutual Exclusion requirement (dangerous!)*

■ Suppose i decides to enter CS; Currently flag[i]=false;

■ So it checks flag [j] is true or not;

■ Suppose flag[j] is false;   Now Proc j decides to enter again. Then Proc i is about to execute flag[i] = true;

■ At this time instant, j checks flag[i] and finds as false;

■ So it decides to enter and sets flag[j] =true and enters its CS section;

■ Proc i also sets flag[i] = true (as it has earlier checked flag[j] as False) and it also enters its CS!!

Q: How about fusing attempts 1 & 3?

Software Solution: Attempt – 4 (Peterson's Solution)

■ We use both turn & flag[] to exercise control;

```
do {

    flag[i] = true;    Express interest;

    turn = j;  But I will give you a chance!    Entry Code

    while (flag[j] && turn == j);

    Critical section

    flag[i] = false;    Exit Code

            remainder section

} while (true);
```

flag[] is set by the respective process

Process i

Software Solution: Attempt – 4 (Peterson's Solution)

- **Q. 6.4:** *Verify if this solution satisfies all the three rules; We will show one of them -* Mutual Exclusion property

```
do {
    flag[i] = true;                    Entry
    turn = j;                          Code
    while (flag[j] && turn == j);
            critical section
    flag[i] = false;
            remainder section
} while (true);
```

When P_i wants to enter, following are possible:

1) If P_j is outside CS then flag[j] is FALSE and P_i may enter.
2) If P_j is in its CS, eventually it will set flag[j] to FALSE and P_i may enter.
3) If P_j is in the entry section and flag[j] is still false, P_i may enter CS if it reaches ***while*** first. Otherwise, P_j enters and P_i may enter after P_j sets flag[j] to FALSE and exits.

Thus, P_i may need to wait for at most one round.

Lamport's Bakery Algorithm

Solution for **n** competing processes

Each process gets a unique token (ID) and sets a boolean variable flag[] to express its interest to enter its CS; (*Same as most solutions seen before*)

Algorithm uses this pair: (token, process ID)

Algorithm's Key Idea: Before entering its CS, the process receives a number (=token); Process that holds the smallest token number enters the CS;

Just like in a bakery, get a number at the entrance, serve according to it!

Contention may arise - If processes **i** and **j** receive the same token number, if **i ≤ j**, then **P$_i$** is served first; else **P$_j$** is served first; This is like assigning a Priority to a process; That is, we will say that **i** has higher **priority** than **j**

Lamport's Bakery Algorithm

```
public Bakery(int n)
{
    flag = new boolean[n];
    label = new label[n];
    for(i=0;i<n;i++){
        flag[i] = F; // initialization
        label[i]=0; // initialization
            }
}
```

```
public void lock() {
    flag[i] = true; // expression of interest
    label[i] = max(label[0],…,label[n-1]) + 1;
    while (flag[k] && ((label[i], i) > (label[k],k));
    // waits in this while loop for one of
        the conditions to fail to access CS
}

public void unlock() {
            flag[i] = false;
}
```

Note: label is our token mentioned earlier;

Execution details of Bakery algorithm

- Main flow is on FCFS basis;

- A process takes a token before it enters it CS and waits for other processes that have lower token ID to complete their CS work;

- Bakery() part of the Code:

    - n: # of processes;

    - flag array - Boolean array which is used by a process to express its interest to enter CS;

    - label array - used to schedule between the interested processes; Lower label process gets higher priority;

Execution details of Bakery algorithm

■ lock() part of the Code:

This is for locking purposes; When a process i needs to access CS it calls this lock() procedure;

     - flag[i] is set to true;

Example: Say process 3 wants to enter; Set flag[3] = T;

Then assign label[3] in the next line as follows:

     - Take maximum of all the current label values + 1,   which is the new label value to process 3;

Looping: See if any other process k (whose flag is true) & has a label lower than its value (lexicographically)

Execution details of Bakery algorithm

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

- If such a condition satisfies then it waits in this while() condition; Note that due to any concurrency issues, if label values are identical for two processes then process # will be checked.

- Continuing our example, now, 3 can enter CS region; In the mean time, suppose 4 wants to enter! Label value becomes 2; Process 4 *waits in the while()loop* (*why?*) until process 3 sets its flag to F.

- After a process completes its CS work, it calls unlock() and sets its flag to F;

Execution details of Bakery algorithm

Q 6.5: Consider the following competing processes and discuss the cases and demonstrate Bakery algorithm. Start from the initial state and demonstrate the workings.

- Process arrival sequence is given in the following time order:

PID	Flag	Label	CS

1) P3 arrives enters CS;

2) P4 arrives;

3) P3 exits;

4) P1 wants to enter CS;

5) P2 also wants to enter CS

6) P4 completes CS

7) P3 wants to enter now;

8) P1 completes CS

9) P1 and P4 enter CS at the same time and accesses the label[] line in lock() simultaneously;

Record your observations

Drawbacks in Software solutions

- Software solutions are very delicate ☺.
- Processes that need to enter their critical section may busy wait (consuming processor time needlessly).
  - If critical sections are long, it would be more efficient to block processes that are waiting.