Parul® University

Operating System (203105203)

Prof. Umang panchal, Prof Atul Kumar Assistant Professor
Computer Engineering

Parul®
University

DIGITAL LEARNING CONTENT

CHAPTER-3

Inter-Process Communication

Parul®
University

DIGITAL LEARNING CONTENT

Topics

I. Critical Section
II. Race Conditions
III. Mutual Exclusion
IV. Hardware Solution.
V. Strict Alternation.
VI. Peterso's Solution,
VII. The Producer\Consumer Problem
VIII. Semaphores
IX. Event Counters.
X. Monitors.
XI. Classical IPC Problems: Readers & Writer Problem.
XII. Dinning Philosopher Problem

Parul®
University

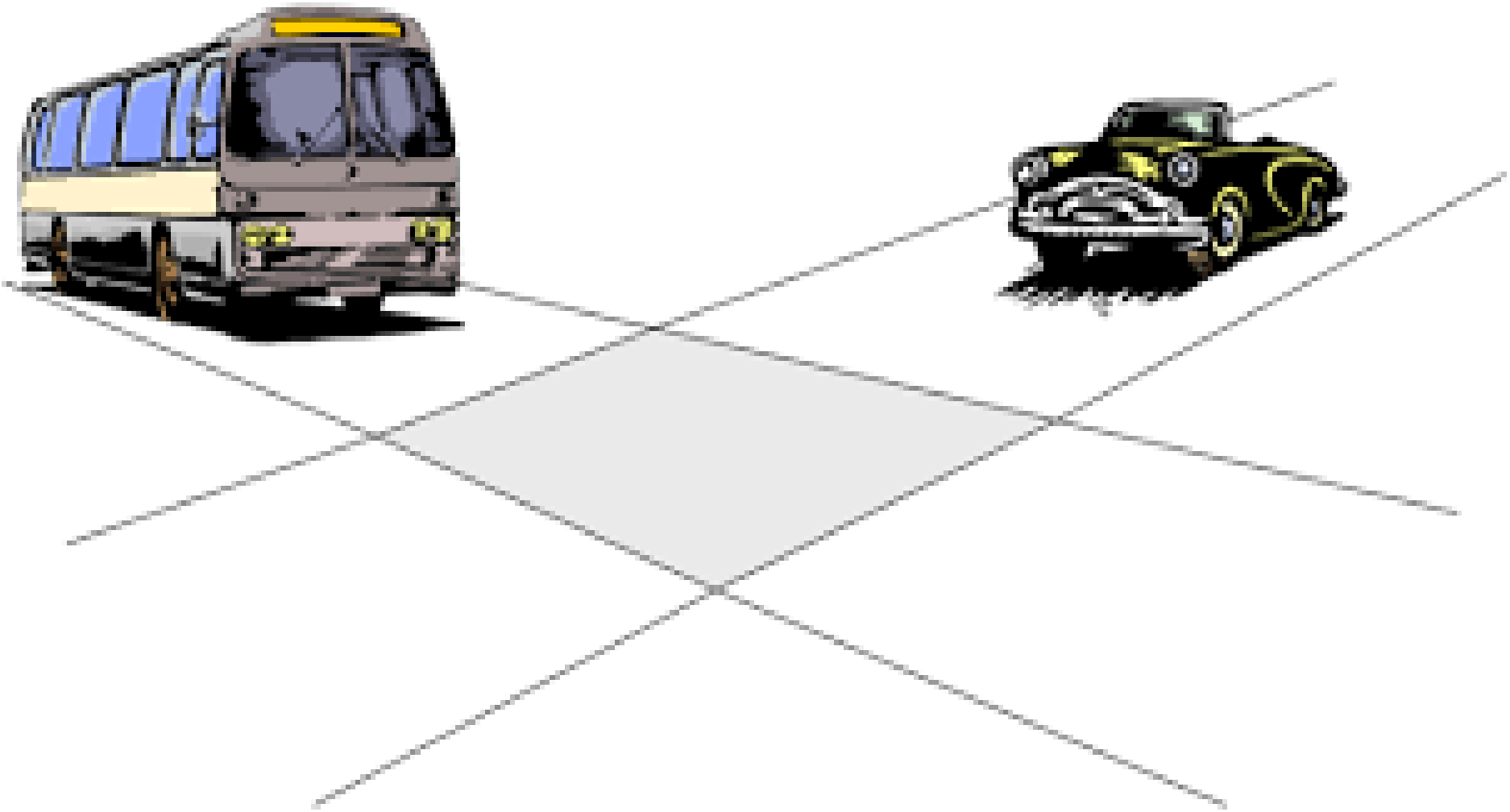
Introduction To IPC

IPC refers to a mechanism which allows communication between two or more processes to perform their actions simultaneously.

Processes that are executing concurrently in a program are of two types-

(i) **Independent Processes:-**These are those processes which does not dependent on execution of other process in same program. Independent processes don't share data with another process.

(ii) **Co-operating Processes:**- These are the processes which can affect or get affected by other processes during execution. Co-operating processes can share data with another processes in simultaneous execution.



Parul® University

Why we need IPC?

There are several reasons which allows processes to co-operate:-

(i) Information sharing

(ii) Computation speedup

(iii) Modularity

(iv) Convenience

Race condition

Race condition is a situation arise due to concurrent execution of more than one processes which are accessing and manipulating the same shared data and the result of execution depends upon the specific order where the access take place.

Race condition leads to inconsistency which is totally undesirable.

Reasons for Race Condition

1. Exact instruction execution order cannot be predicted
2. Resource (file, memory, data etc…) sharing.

Example of race condition

Assume that two threads each increment the value of a global integer variable by 1. Ideally, the following sequence of operations would take place:

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

In the case shown above, the final value is 2, as expected. However, if the two threads run simultaneously without locking or synchronization, the outcome of the operation could be wrong. The alternative sequence of operations below demonstrates this scenario:

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

In this case, the final value is 1 instead of the correct result of 2. This occurs because here the increment operations are not mutually exclusive. Mutually exclusive operations are those that cannot be interrupted while accessing some resource such as a memory location.

Parul® University

DIGITAL LEARNING CONTENT

Critical section

Critical section is piece of code which contains some shared code of variable which is accessible by each process concurrently in order to complete execution. There must be only one process is allowed at a time in critical section otherwise more than one access may lead to inconsistency. concurrent accesses to shared resources can lead to unexpected or erroneous behaviour , so part of the program where the shared resource is accessed is protected. This protected section is the critical section or critical region.

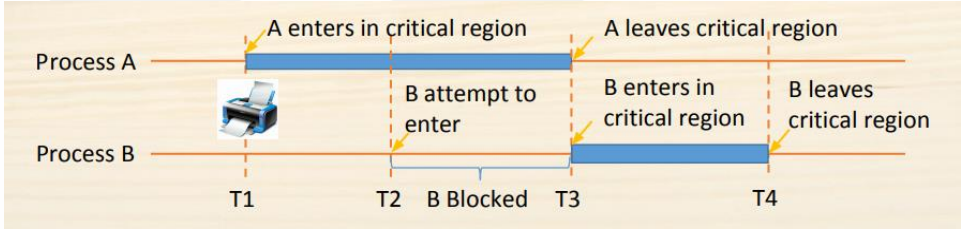


Figure :1 [image source google]

Need of critical section

Process A have to read variable ‘x’ and process B has to write to the same variable ‘x’ at the same time
if A needs to read the updated value of ‘x’, executing Process A and Process B at the same time may not give required results. To prevent this, variable ‘x’ is protected by a critical section. First, B gets the access to the section. Once B finishes writing the value, A gets the access to the critical section and variable ‘x’ can be read.

Process A:

```
// Process A
.
.
b = x + 5;              // instruction executes at time = Tx
.
```

Process B:

```
// Process B
.
.
x = 3 + z;              // instruction executes at time = Tx
.
```

Solution to critical section

Brute-Force approach by using locking mechanism

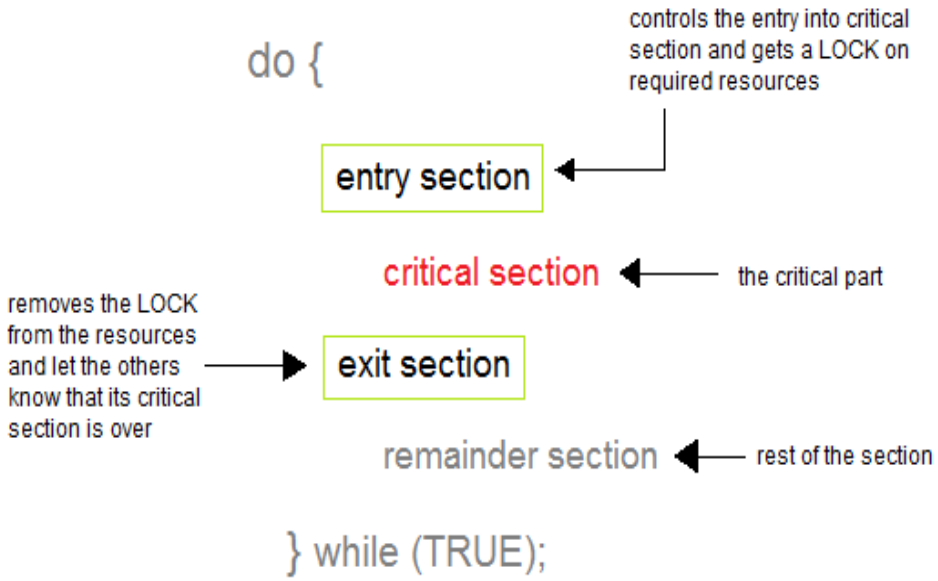


Figure :2 [image source google]

To critical section (Continued)

(i**) Mutual Exclusion: Out** of a group of cooperating processes, only one process can be in its critical section at a given point of time.

(ii) **Progress: If** no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

(iii) **Bounded Waiting: After** a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, system must grant the process permission to get into its critical section.

Solution to synchronization problem

There are some mechanisms have been introduced for synchronization problem which are as follows-

(i) Hardware solution

(ii) Software solution

(iii) Strict alteration

Solution to synchronization problem(Cont....)

Hardware Approach to synchronization problem:-

(i) Exclusive access to memory location always assumed

(ii) Test-and-Set: special machine-level instruction

(iii) Swap: atomically swaps contents of two words

Solution to synchronization problem(Cont....)

Test-and-Set Instruction

(i) hardware assistance for process synchronization .

(ii) a special hardware instruction that does two operations atomically . i.e., both operations are executed or neither is

(a) sets the result to current value

(b) changes current value to true

(c) when describing machine language (CPU) operations, the verb 'set' means 'set to true'

Solution to synchronization problem(Cont....)

Test-and-Set Instruction (Cont.)

```
do {
   while (TestAndSetLock(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
}while (TRUE);
```

Figure :3 [image source google]

Solution to synchronization problem(Cont....)

Mutual-exclusion implementation with the Swap()

- A global Boolean variable lock is declared and is initialized to false and each process has a local Boolean variable key

```
while (true) {
    key = TRUE;
    while ( key == TRUE)
        Swap (&lock, &key );
            //   critical section
    lock = FALSE;
            //     remainder section
}
```

The above both algorithms do not satisfy the bounded-waiting requirement

Figure :3 [image source google]

Solution to synchronization problem(Contd…)

Software Approach to synchronization problem***(Peterson's Solution)***:-
It is a two process solution.
We must assume that LOAD and STORE instructions are atomic and can not be interrupted in between.
The two processes will share two variables
(i)**int** turn :- is used to indicate that whose turn is there to enter in critical section.(ii)**Boolean** interested[2]:- is used to indicate whether a process is ready to enter in critical section the interested[i] = true {means Pi is ready}

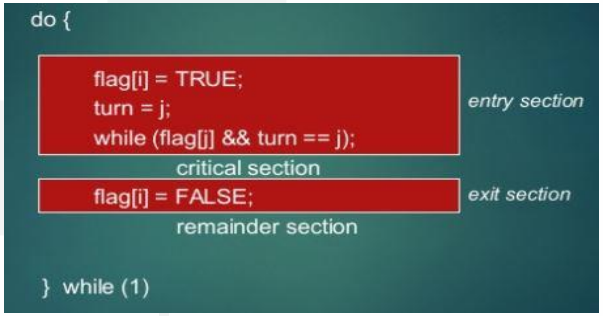


Figure :4 [image source google]

Parul® University

DIGITAL LEARNING CONTENT

solution to synchronization problem(Cont....)

Hardware Approach to synchronization problem***(Peterson's Solution)***:-

```
Entry section(process)
{
1. int  other;
2. other = 1- process;
3. interested [ process] = TRUE;
4. turn = process
5. while (interested[other]==TRUE && turn
== process);
}
```

CS

```
Exit section(process)
{
6. interested[process] == False;
}
```

solution to synchronization problem(Cont.…)

Analysis:-

(i) Mutual Exclusion is assured as only one process can access the critical section at any time.

(ii) Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.

(iii) Bounded Waiting is preserved as every process gets a fair chance.

Disadvantages:-

(i) It involves Busy waiting.

(ii) It is limited to 2 processes.

Parul® University

DIGITAL LEARNING CONTENT

solution to synchronization problem(Cont....)

Strict Alteration:-

Turn Variable or Strict Alternation Approach is the software mechanism implemented at user mode. It is a busy waiting solution which can be implemented only
for two processes. In this approach,
A turn variable is used which is actually

This approach can only be used for only two processes. In general, let the two processes be Pi and Pj. They share a variable called turn variable. The pseudo code of the program can be given as following.

```
Non - CS
while (turn ! = i);
Critical Section
turn = j;
Non - CS
Non - CS
while (turn ! = j);
Critical Section
turn = i ;
Non - CS
```

Figure :6[image source google]

PRODUCER AND CONSUMER PROBLEM

It is multi-process synchronization problem.

It is also known as bounded buffer problem.

This problem describes two processes producer and consumer, who share common, fixed size buffer.

- Producer process will Produce some data item and put it into buffer.
- Consumer process will consume this data item (remove it from the buffer).

Condition for inconsistency:-

a) Producer must not try to produce any data item to buffer if buffer size i full.

b) Consumer must not try to consume any data item if buffer is empty.

PRODUCER AND CONSUMER PROBLEM

Solution for Producer:-

a) Producer either go to sleep or discard data if the buffer is full.
b) Once the consumer removes an item from the buffer, it notifies the producer to put the data into buffer. (by using some synchronization tool)

PRODUCER AND CONSUMER PROBLEM

Solution for Consumer:-

a) Consumer can go to sleep if the buffer is empty.

b) Once the producer puts data into buffer, it notifies the consumer to remove (use) data item from buffer. (By using some synchronization tool)

	Producer	Consumer
Empty	YES	NO
Filled	NO	YES
Partially filled	YES	YES

Figure :7[image source google]

SEMAPHORE

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait() and signal() that are used for process synchronization.

The definitions of wait() and signal() are as follows

Wait:-The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

```
wait(S)
{
while (S<=0);
 S--;
}
```

SEMAPHORE

Signal:- The signal operation increments the value of its argument S.

```
signal(S)
{ S++;
 }
```

Types of Semaphores:-

There are two main types of semaphores i.e. counting semaphores and binary semaphores. these are given as follows

Semaphore

Counting Semaphores:- These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count I decremented.

Binary Semaphores:- The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

Monitor

- A more elevated level synchronization primitive.
- A monitor is a bunch of techniques, variables, and information structures that are completely assembled in an exceptional sort of module or bundle.
- Procedures may call the strategies in a monitor at whatever point they need to, however, they can't directly get to the monitor's interior data structures from methodology announced outside the monitor.
- Monitor have a significant property for accomplishing common rejection: just one procedure can be dynamic in a monitor at any moment.

- At the point when a procedure calls a monitor technique, the initial hardly any directions of the method will verify whether some other procedure is as of now dynamic inside the monitor .
- Assuming this is the case, the calling procedure will be suspended until the different procedure has left the monitor. On the off chance that no different procedure is utilizing the monitor, the calling procedure may enter.

Producer consumer problem using monitor

- The arrangement proposes condition factors, alongside two procedures on them, wait and signal.
- At the point when a monitor method finds that it can't proceed (e.g., the maker finds the buffer full), it does a lookout for some condition variable, full.
- This activity causes the calling procedure to block. It additionally permits another procedure that had been recently block from entering the screen to enter now.

Producer consumer problem using monitor

- This different procedure the customer, can awaken its dozing accomplice by doing a sign on the condition variable that its accomplice is looking out for.
- To abstain from having two dynamic procedures in the monitor simultaneously a sign explanation may show up just as the last proclamation in a monitor method.
- On the off chance that a sign is done on a condition variable on which a few procedures are pausing, just one of them, controlled by the framework scheduler, is retrieve.

Producer consumer problem using monitor

```
monitor Producer_Consumer
{
    int item_Count;

    condition full;
    condition empty;


    procedure add(item)
    {
        while (item_Count == Buffer_S)

        wait(full);
    }

    putItemIntoBuffer(item);

    item_count =item_count +1;


    if (item_count==1)
    {
        notify(empty);
    }
}
```

Figure :8

```
Procedure remove()
{
    while (item_Count == 0)
    {
        wait(empty);
    }


    item = removeItemFromBuffer();

    item_Count = item_Count - 1;

    if(item_Count == Buffer_S - 1)
    {
        notify(full);
    }
}
```

Figure :9

Reader's & Writer Problem

- Reader's writer problem is a one of the example of a classic synchronization problem.
- Consider below situation when resource shared between number of people.
- If one user is accessing the file so at the same time no other user can read and write that same file. Otherwise change will not reflects.
- However different users can read same file simultaneously.

Reader's & Writer Problem

- This situation consider as reader's and writer's problem
- Lets take a look into problem statement more.
- Two operation we consider here one is Read operation and another is write operation.
- We have two user consider here one is reader and another one is writer.
- If one reader is using file and one writer come to access that same file is a problem (R + W = Problem)
- If one writer is using file and one reader come to to access same file is a problem (W + R = Problem)

Parul®
University

Reader's & Writer Problem

- If one writer is using file and another write come to access same file is a problem. (W + W = Problem)
- If one reader is using file and another reader come to access same file so no problem (R + R = No problem).
- This four case we have to consider.
- Here we use semaphore because here synchronization requires to take care of given cases.

Reader's & Writer Problem

```
1  int read_count = 0
2
3  semaphore mutex = 1
4  semaphore b =1
5
6  void reader(void)
7  {
8      while(true)
9      {
10         down(mutex);
11
12         read_count = read_count+1;
13         if(read_count== 1) then down(b)
14
15         up(mutex)
16
17         readDataBase();
18
19         down(mutex)
20
21         read_count =read_count-1;
22
23         if(read_count== 0)then up (b)
24         up(mutex);
25
26         use_read_data();
27     }
28 }
29
```

Figure :10

```
1  Void write(void)
2  {
3      while(true)
4      {
5          down(b);
6
7          readDataBase();
8      }
9  }
```

Figure :11

Dining Philosopher's Problem

As we take a look in given figure five philosopher (P0,P1,P2,P3,P4) seating on dining table and given chopsticks (C0,C1,C2,C3,C4) to eat food on table .each philosopher requires tow chopsticks at a same time to eat food

At any instant, a philosopher is either eating or thinking. If philosopher is eating have to pick chopstick one from their left and one from their right.

If philosopher is thinking it down the chopstick on table.

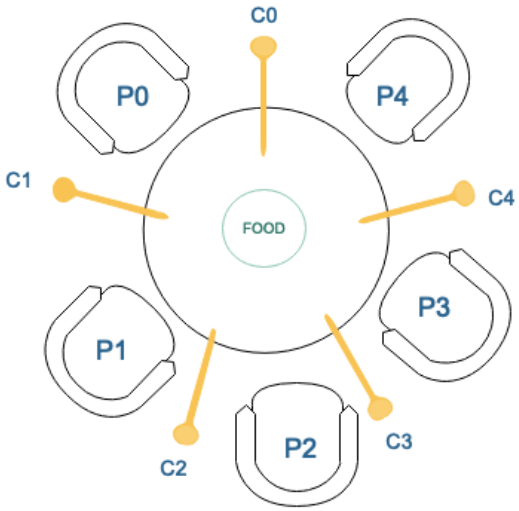


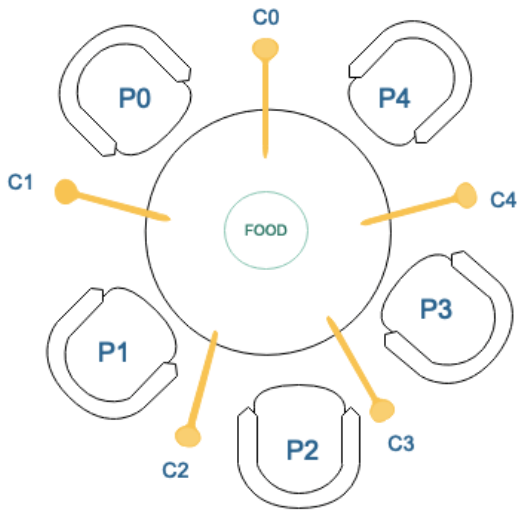
Figure :12 dining philosopher's problem [1]

Dining Philosopher's Problem

For example philosopher P0 want to eat food it require two chop stick C0 and C1 as consider to below here i = 0 for p0 as 0^{th} position.

take_chopstick(i); = C0

take_chopstick((i+1) % N); =(0+1)mod5 = C1



Dining Philosopher's Problem

Now discuss case 1: In which one by one philosopher is eating and thinking . which is best case. So if philosopher P0 is eating with chopstick C0 & C1 then put the chopstick and then other P1 philosopher is come to eat which require chopstick C1 and C2 .as we see chopstick is now available for philosopher P1 because philosopher P0 already put down chopstick C1 after eating. So in this case no problem occur

```
1  void philisopher (void)
2  {
3      while(true)
4      {
5          thinking();
6
7          take_chopstick(i);
8          take_chopstick((i+1) % N);
9          eat();
10
11         put_chopstick(i);
12         put_chopstick((i+1) % N);
13     }
14 }
```

Figure :13

Dining Philosopher's Problem

Now lets discuss Case 2: In which if philosopher P0 is eating with chopstick C0 & C1 and at the same time philosopher P1 enter to eat so it require chopstick C1 & C2 . Now in this case C1 chopstick not available for Philosopher P1 because it taken by philosopher P0 already.

To deal with is situation Binary semaphore use

When one philosopher eat the food with needed chopstick and **case 2** situation occur it block that philosopher. otherwise it allow another philosopher if its require chopstick available on table so at some time multiple philosopher come and eat and put chopstick if their required chopstick available on the table .

Parul® University

DIGITAL LEARNING CONTENT

Dining Philosopher's Problem

Consider this given table in this it shows which semaphore require by each philosophers.

P0	S0,S1
P1	S1,S2
P2	S2,S3
P3	S3,S4
P4	S4,S0

Table:1

```
1  void philisopher (void)
2  {
3      while(true)
4      {
5          thinking();
6
7          wait(take_chopstick(Si));
8          wait(take_chopstick((Si+1) % N));
9          eat();
10
11         signal( put_chopstick(i));
12         signal( put_chopstick((i+1) % N));
13     }
14 }
15
16
17
18
```

Figure :14

Dining Philosopher's Problem

- Lets consider now at starting condition all semaphore are 1.
- Now if Philosopher P0 is using S0 and S1 semaphore at present so it become 0 Until Philosopher put it down . So another philosopher who want to use common semaphore not allow at time.

Consider case 3: If all five philosopher comes at time and pick their first semaphore (S0,S1,S2,S3,S4) shown in Table 3 so for all philosopher second semaphore for will not available.

S0	S1	S2	S3	S4
1	1	1	1	1
0	0			

Table:2

Parul® University

DIGITAL LEARNING CONTENT

Dining Philosopher's Problem

This situation consider as a deadlock. To avoid this situation philosopher P4 need to swipe their semaphore as below Table 4. So when P4 comes it will not allow Due to not availability of their first semaphore. So S4 will remain 1 and P3 will use S4 & S3 for eat and put it back. So again S3 & S4 will available as show in Table 6 so further that P2 will use S2 & S3 and run further process .

P0	S0,S1
P1	S1,S2
P2	S2,S3
P3	S3,S4
P4	S4,S0

Table:3

P0	S0,S1
P1	S1,S2
P2	S2,S3
P3	S3,S4
P4	S0,S4

Table:4

S0	S1	S2	S3	S4
0	0	0	0	1

Table:5

S0	S1	S2	S3	S4
0	0	0	1	1

Table:6

S0	S1	S2	S3	S4
0	0	1	1	1

Table:7

Parul® University

Reference

[1] https://www.geeksforgeeks.org/dining-philosopher-problem-using-semaphores/

DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in