Process Synchronization

Process Synchronization

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Producer Consumer Problem Revisited

- Modifying the Bounded Buffer Approach for Producer and Consumer Problem.

- Having a **`counter`** that keeps track of the number of full buffers.

Producer Consumer Problem Revisited

- One possibility is to add an integer variable counter, initialized to 0.
- Counter is
  - incremented every time
    - we add a new item to the buffer
  - decremented every time
    - we remove one item from the buffer.

Producer Consumer Problem Revisited

- The code for the producer process can be modified as follows:

```
while (true) {
      /* produce an item in nextProduced */
      while (counter == BUFFER_SIZE)
         ; /* do nothing */
      buffer[in] = nextProduced;
      in = (in + 1) % BUFFER_SIZE;
      counter++;
}
```

Earlier Approach-

```
item next_produced;
while (true) {
/* produce an item in next produced */
while (((in + 1) % BUFFER_SIZE) == out)
                        ; /* do nothing */
buffer[in] = next_produced;
in = (in + 1) % BUFFER_SIZE;
}
```

Producer Consumer Problem Revisited

- The code for the consumer process can be modified as follows:

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

Earlier Approach-

```
item next_consumed;
while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
}
```

Producer Consumer Problem Revisited

- The producer and consumer routines are correct separately,
  - May not function correctly when executed concurrently.

- **Suppose that the value of the variable counter is currently 5**

Producer Consumer Problem Revisited

- The producer and consumer processes execute
  - the statements "counter++" and "counter--" concurrently.

- Result= counter may be 4, 5, or 6!

- The only correct result, counter == 5, **which is generated correctly**
  - **if the producer and consumer execute separately.**

Producer Consumer Problem Revisited

- " counter++" may be implemented in machine language as –

  register1 = counter
  register1 = register1 + 1
  counter= register1

- register1 is one of the local CPU registers.

- "counter--" is implemented as follows:

  register2 = counter
  register2 = register2 - 1
  counter= register2

- register2 is one of the local CPU registers.

Producer Consumer Problem Revisited

- The concurrent execution of "counter++" and "counter--" is equivalent to
  - A sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order.
- But the order within each high-level statement is preserved

Producer Consumer Problem Revisited

- One such interleaving is-

T_0:	*producer*	execute	$register_1 = \text{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \text{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\text{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\text{counter} = register_2$	$\{counter = 4\}$

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Producer Consumer Problem Revisited

- Result=> "counter == 4“, incorrect state
  - indicating that four buffers are full,
- Correct state=five buffers are full.

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Producer Consumer Problem Revisited

- If the order of the statements at T4 and *T5* is reversed, we would arrive at the incorrect state
  - Result=> "counter == 6“, incorrect state

T_0:	*producer*	execute	$register_1 = \text{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \text{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\text{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\text{counter} = register_2$	$\{counter = 4\}$

Producer Consumer Problem Revisited

- Incorrect state as
  - we allowed both processes to manipulate the variable counter concurrently.

Producer Consumer Problem Revisited

- When several processes access and manipulate the same data concurrently

- The outcome of the execution depends on
  - the particular order in which the access takes place,

- Also called a Race Condition.

Producer Consumer Problem Revisited

- To guard against the race condition,
  - Only one process at a time can be manipulating the variable counter.
  - Thus the processes must be synchronized

The Critical Section Problem

The Critical Section Problem

- Consider a system consisting of *n* processes {Po, P1 , ... , *Pn-1*}.

Critical Section-

- Each process has a segment of code, called a critical section in which the process may be
  - changing common variables,
  - updating a table,
  - writing a file, and so on.

The Critical Section Problem

- When one process is executing in its critical section,
  - No other process is to be allowed to execute in its critical section.
  - Execution of Critical Sections by the processes is mutually exclusive
- ***Critical-section problem*** **is to design a protocol that the processes can use to cooperate.**

The Critical Section Problem

- **Entry section**
  - Each process must request permission to enter its critical section. The section of code implementing this request is the entry section.

- **Exit Section**
  - The critical section may be followed by an exit section.

- **Remaining Section**
  - The remaining code is the remaining section.

The Critical Section Problem

General Structure of a Typical Process Pi

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (TRUE);
```

The Critical Section Problem

Solution ≡ Must Satisfy three requirements-

1) **Mutual Exclusion**
2) **Progress**
3) **Bounded Waiting**

The Critical Section Problem

1) Mutual Exclusion

If Process Pi is executing in its critical section, then no other processes can be executing in their critical sections

The Critical Section Problem

2) Progress

- If no process is executing in its critical section and
  - some processes wish to enter their critical sections

- Then only those processes
  - that are not executing in their remainder sections
  - can participate in deciding which will enter its critical section next.
- This selection cannot be postponed indefinitely.

The Critical Section Problem

2) Progress

- Only those processes interested in entering CS , should compete to enter CS.

- Only those processes wishing to enter CS, should compete for CS

The Critical Section Problem

3) Bounded Waiting

- There exists
  - a bound, or
  - limit,
  - on the number of times that other processes are allowed to enter the critical sections
- after a process has made a request to enter its critical section and before that request is granted.

The Critical Section Problem

3) Bounded Waiting

- Max After a bound/time limit , after which the process definitely will get a chance to enter CS

The Critical Section Problem

Assumption-

- Each process is executing at a nonzero speed.
- No assumption concerning the relative speed of the *n* processes.

Critical-Section Handling in OS

Two approaches are used to handle critical sections in OS depending on if kernel is-

- **Preemptive**
- **Non-preemptive**

Critical-Section Handling in OS

- **Preemptive-**
  - allows a process to be preempted while it is running in kernel mode.

- **Non-preemptive-**
  - does not allow a process running in kernel mode to be preempted;
  - a kernel-mode process will run until it
    - exits kernel mode,
    - blocks, or
    - voluntarily yields control of the CPU.

Critical-Section Handling in OS

- **Non-preemptive kernel-**
  - is essentially free from race conditions on kernel data structures,
  - as only one process is active in the kernel at a time.

- **Preemptive kernels-**
  - must be carefully designed to ensure that shared kernel data are free from race conditions.
  - are especially difficult to design for SMP architectures,
  - since in these environments it is possible for two kernel-mode processes
  - to run simultaneously on different processors.

Critical-Section Handling in OS

- Why would anyone favor a preemptive kernel over a nonpreemptive one? ?

Critical-Section Handling in OS

- Why would anyone favor a preemptive kernel over a nonpreemptive one?

- A preemptive kernel
  - is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel.
  - may be more responsive, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes.

Solutions to The Critical Section Problem

- **Software Based Solutions**
- **Hardware Based Solutions**

Software Synchronization

Software Based Solutions to The Critical Section Problem

- **Software Based Solutions**
  - **Two process Solution**
    - **Algorithm 1**
    - **Algorithm 2**
    - **Algorithm 3/Peterson's Solution**

  - **Multiple Process Solution**

Software Based Solutions to The Critical Section Problem

- Two process Solution
  - Assume that the load and store machine-language instructions are atomic;
  - that is, cannot be interrupted

Software Based Solutions to The Critical Section Problem

- **Algorithm 1**

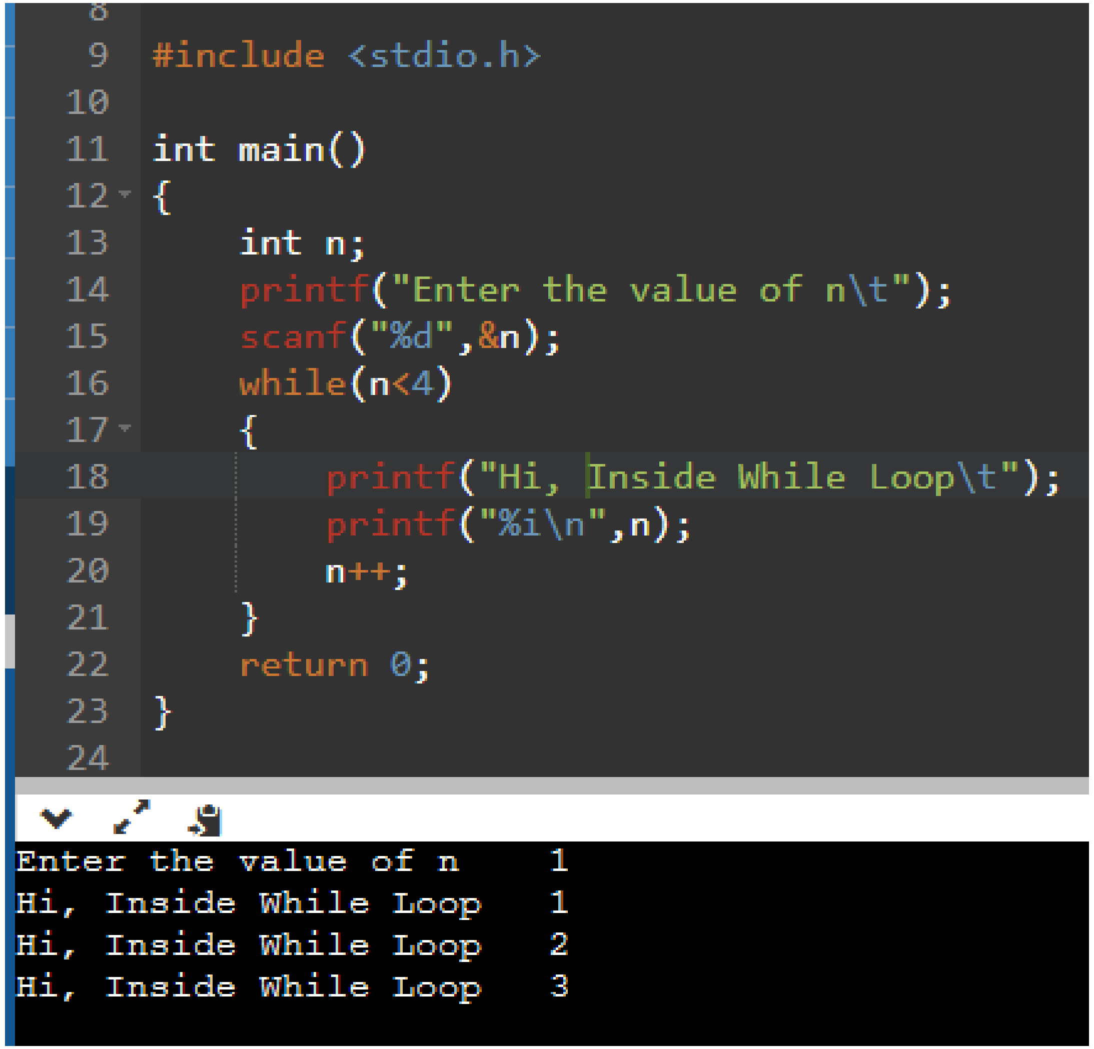
```
do {
    while (turn != i);
        critical section
    turn = j;
        remainder section
} while (1);
```

The structure of process P_i in algorithm 1.

Working of While Loop without semicolon;

Example-

```
#include <stdio.h>
int main()
{
        int n = 0;
        printf("enter value of n");
        scanf("%d",&n);
        while(n < 4)
        {
                printf("Hi, Inside while loop\n");
                printf("%i\n", n);
                n++;
        }
}
```



Working of While Loop with;

- Generally we Dont write a semicolon after the condition in while loop
- The problem is that the loop body becomes a semicolon, which is a do nothing statement.
- This while loop has no body

Working of While Loop with;

- **Trapped in while loop as Condition is true**
- **Executes while loop with No body**

- Out of while loop as Condition is False
- Enters CS

```c
#include <stdio.h>

int main()
{
    int n=0;
    printf("Enter Value of n \t");
    scanf("%d",&n);
    while(n<4);
    printf("Inside Critical Section\n");
    return 0;
}

```

```
Enter Value of n        7
Inside Critical Section


...Program finished with exit code 0
```

Working of While Loop with;

Example-

```
#include <stdio.h>
int main()
{
    int n = 0;
    printf("enter value of n");
    scanf("%d",&n);
    while(n < 4);
    {
        printf("Hi, Inside while loop\n");
        printf("%i\n", n);
        n++;
    }
}
```

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Working of While Loop with;

```
                                    Online C Co
                          Code, Compile, Run and
Write your code in this editor and pres

*************************************

#include <stdio.h>

int main()
{
    int n = 0;
    printf("enter value of n");
    scanf("%d",&n);
  while(n < 4);
  {
    printf("Hi, Inside while loop\n");
    printf("%i\n", n);
    n++;
  }
```

```
enter value of n 7
Hi, Inside while loop
7


...Program finished with exit code 0
Press ENTER to exit console.
```

- Condition =False,
- Print statement Executes

```
                             Online C Compiler.
                   Code, Compile, Run and Debug C
Write your code in this editor and press "Run"

*********************************************

#include <stdio.h>

int main()
{
   int n = 0;
   printf("enter value of n");
   scanf("%d",&n);
  while(n < 4);
  {
    printf("Hi, Inside while loop\n");
    printf("%i\n", n);
    n++;
  }
```

```
enter value of n 3
```

- Condition =True,
- Empty statement Executes
- Infinite Loop
- Print statement doesn't execute

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Working of While Loop with;

```
Run   Debug   Stop   Share   Sa
main.c
                              Online C Co
                    Code, Compile, Run and
Write your code in this editor and pres

**************************************

#include <stdio.h>

int main()
{
    int n = 0;
    printf("enter value of n");
    scanf("%d",&n);
  while(n < 4);
  {
    printf("Hi, Inside while loop\n");
    printf("%i\n", n);
    n++;
  }
```

```
enter value of n 7
Hi, Inside while loop
7


...Program finished with exit code 0
Press ENTER to exit console.
```

- Condition =False,
- Control comes out of Loop
- Critical Section executes

```
main.c
                                   Online C Compiler.
                         Code, Compile, Run and Debug C
Write your code in this editor and press "Run"

*************************************************

#include <stdio.h>

int main()
{
    int n = 0;
    printf("enter value of n");
    scanf("%d",&n);
  while(n < 4);
  {
    printf("Hi, Inside while loop\n");
    printf("%i\n", n);
    n++;
  }
```

```
enter value of n 3
```

- Condition =True,
- Empty statement Executes
- Process gets trapped in Infinite Loop
- Does not enter Critical Section

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Software Based Solutions to The Critical Section Problem

- **Algorithm 1**

```
do {
      while (turn != i);
          critical section
      turn = j;
          remainder section
} while (1);
```

The structure of process P_i in algorithm 1.

Entry Code

Exit Code

- **Acts like a trap**
- **Stopping processes to enter into the Critical Section**
- **Turn=Shared Common Integer/Global Integer turn initialized to 0/1**

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Algorithm 1 :
Mutual Exclusion Check

If P0 is executing critical section,
Can another process P1 enter the critical section or not?

Algorithm 1

CS=critical section

- **Algorithm 1- Lets initialize turn with 0**

P0, i=0	P1, i=1

do{
while(turn!=0);
        critical section
turn=1;
remainder section
}while(1);
	do{
	while(turn!=1);
	        critical section
	turn=0;
	remainder section
	}while(1);

```
do {
    while (turn != i);
        critical section
    turn = j;
        remainder section
} while (1);
```

The structure of process P_i in algorithm 1.

Can P1 enter CS while P0 is in CS ?

Can P1 enter CS while P0 is in CS-No

- **Algorithm 1**

P0, i=0

do{
while(turn!=0);
      critical section
turn=1;
remainder section
}while(1);

Lets initialize turn with 0

- For P0,Lets take turn=0,
- while condition =false
- P0 enters CS
- While P0 is Inside CS, Still turn=0
- P1 tries to enter the CS

P1, i=1

do{
while(turn!=1);
      critical section
turn=0;
remainder section
}while(1);

- P1 tries to enter CS
- Still turn=0,
- while condition =true
- P1 gets trapped in an infinite loop
- P1 is unable to enter CS

Can P1 enter CS while P0 is in Remainder Section ?

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Can P1 enter CS while P0 is in Remainder Section -Yes

- **Algorithm 1**

P0, i=0

do{
while(turn!=0);
critical section
turn=1;
remainder section
}while(1);

CS=critical section

- For P0,
- After CS
- Exit Code makes turn=1
- Starts executing Remainder Section
- Now can P1 enter CS

P1, i=1

do{
while(turn!=1);
critical section
turn=0;
remainder section
}while(1);

- P1 tries to enter CS
- Now turn=1,
- while condition =false
- P1 enters CS

Algorithm 1 :
Mutual Exclusion Check

Satisfied!!!!!

Can P0 enter CS immediately again after completing RS?

Can P0 enter CS immediately again after completing RS?-No

- **Algorithm 1**

P0, i=0

do{
while(turn!=0);
critical section
turn=1;
remainder section
}while(1);

CS=critical section

- After 1st run
- turn becomes =1,

For P0

- while condition =true
- P0 gets trapped in an infinite loop

- P0 is unable to enter CS

Solutions to The Critical Section Problem

- **Algorithm 1**

- If turn == i, then process Pi executes in critical section

  - turn=0, Po executes CS
  - turn=1, P1 executes CS

Algorithm 1 :
Progress Requirement Check

Algorithm 1 : Progress Requirement Check

- **After P0 comes out then P1 gets control to go inside**
- **After P1 comes out then P0 gets control to go outside.**

- **P0 $\rightarrow$ P1**

Algorithm 1 : Progress Requirement Check

- **It is Strict Alteration**
  - **Case 1-If a process doesn't want to go to CS, still it goes due to alteration.**
  - **Case 2-If P1 doesn't want to go in CS and P0 wants to go in CS, still P0 won't get chance.**

- **This Solution is not following progress**

Algorithm 1 : Bounded Waiting Check

- Process 0 can go directly and Process 1 can go after process 0 into critical section. So, Bounded waiting is satisfied.

Solutions to The Critical Section Problem

- **Algorithm 1**

- Mutual Exclusion is preserved
  - Ensures that only one process at a time can be in its critical section.
- Does not satisfy, Progress Requirement
- Bounded Waiting Criteria is satisfied

Algorithm 2

Algorithm 2

- The two processes share boolean array:
  - Boolean flag[2]

- 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!

- Boolean array can be initialized to false.

Boolean Array flag[2]

[0]	[1]
F	F

- As the process wants to enter CS, its cell can be made True.

Solutions to The Critical Section Problem

Algorithm 2

```
do {
    flag[i] = true;
    while (flag[j]);

        critical section

    flag[i] = false;

        remainder section

} while (1);
```

The structure of process P_i in algorithm 2.

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Algorithm 2

- **Algorithm 2**

P0,i=0	P1,i=1

do{
flag[0]=true;
while(flag[1]);
critical section
flag[0]=false;
remainder section
}while(1);
	do{
	flag[1]=true;
	while(flag[0]);
	critical section
	flag[1]=false;
	remainder section
	}while(1);

```
do {

    flag[i] = true;
    while (flag[j]);

        critical section

    flag[i] = false;

        remainder section

} while (1);
```

The structure of process P_i in algorithm 2.

Algorithm 2

- **Algorithm 2**

P0	P1

do{
    flag[0]=true;
    while(flag[1]);
    critical section
    flag[0]=false;
    remainder section
}while(1);
	do{
	    flag[1]=true;
	    while(flag[0]);
	    critical section
	    flag[1]=false;
	    remainder section
	}while(1);

Boolean Array flag[2]

[0]	[1]
F	F

Algorithm 2 :
Mutual Exclusion Check

If P0 is executing critical section,
Can another process P1 enter the critical section or not?

If P0 is executing CS,
Can another process P1 enter the CS or not? No

- **Algorithm 2**

P0

do{
flag[0]=true;
while(flag[1]);
critical section
flag[0]=false;
remainder section
}while(1);

- For P0
- It wants to enter CS, so sets flag[0]=T
- It checks If his friend P1 wants to go,
- As flag[1]=F
- Control Comes out of while loop
- P0 executes CS

P1

do{
flag[1]=true;
while(flag[0]);
critical section
flag[1]=false;
remainder section
}while(1);

```
do {
    flag[i] = true;
    while (flag[j]);

        critical section

    flag[i] = false;

        remainder section

} while (1);
```

The structure of process P_i in algorithm 2.

[0]	[1]
T	F

Boolean Array flag[2]

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

If P0 is executing CS,
Can another process P1 enter the CS or not? No

- **Algorithm 2**

P0

do{
flag[0]=true;
while(flag[1]);
critical section
flag[0]=false;
remainder section
}while(1);

- For P0
- It wants to enter CS, so sets flag[0]=T
- It checks If his friend P1 wants to go,
- As flag[1]=F
- Control Comes out of while loop
- P0 executes CS

P1

do{
flag[1]=true;
while(flag[0]);
critical section
flag[1]=false;
remainder section
}while(1);

- P1 tries to enter CS
- Sets flag[1]=T,
- while condition =true
- P1 gets trapped in an infinite loop
- P1 is unable to enter CS

[0]	[1]
T	F

Boolean Array flag[2]

[0]	[1]
T	T

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Can P1 enter CS while P0 is in Remainder Section ?

Can P1 enter CS while P0 is in Remainder Section ?Yes

- **Algorithm 2**

P0

do{
flag[0]=true;
while(flag[1]);
critical section
flag[0]=false;
remainder section
}while(1);

- For P0
- After finishing CS
- Sets flag[0]=false
- Continues with RS

P1

do{
flag[1]=true;
while(flag[0]);
critical section
flag[1]=false;
remainder section
}while(1);

- P1 tries to enter CS
- sets flag[1]=T,
- while condition =false
- P1 enters CS

[0]	[1]
F	T

Boolean Array flag[2]

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Algorithm 2 :
Mutual Exclusion Check

Satisfied!!!!!

Algorithm 2:
Progress Requirement Check

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Can P0 enter CS immediately again after completing RS?

Can P0 enter CS immediately again after completing RS? Yes

- **Algorithm 2**

P0

do{
flag[0]=true;
while(flag[1]);
critical section
flag[0]=false;
remainder section
}while(1);

- If His friend P1 is not interested in CS and flag[1]=F
- and If P0 wants to go again , is interested, so set flag[0]=T
- while loop is false
- So P0 can enter CS any number of times

[0]	[1]
T	F
F	F

Boolean Array flag[2]

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Algorithm 2 : Progress Requirement Check

- **If P1 wants to go in CS again and immediately after executing RS, it can enter again if P2 is not interested in CS**

- **Whichever process is interested , while others are not, gets to enter CS**

- **Progress till now.**

Algorithm 2 : Progress Requirement Check

- **Contradiction crops up……………..Now…**

If both P0,P1 want to enter CS

Concurrent Execution!!!

- **Algorithm 2**

P0

do{
    flag[0]=true;
    while(flag[1]);
    critical section
    flag[0]=false;
    remainder section
}while(1);

- For P0

- Gets trapped in While
- Infinite Loop
- P0 Doesn't enter CS

P1

do{
    flag[1]=true;
    while(flag[0]);
    critical section
    flag[1]=false;
    remainder section
}while(1);

- For P1

- Gets trapped in While
- Infinite Loop
- P1 Doesn't enter CS

[0]	[1]
T	T

Boolean Array flag[2]

Algorithm 2 : Progress Requirement Check

- **If Both P0,P1 want to enter CS, Both go in infinte loop, No one gets CS**

- **Thus, No Progress.**

Solutions to The Critical Section Problem

- **Algorithm 2**

- Mutual Exclusion is preserved
- Does not satisfy, Progress Requirement.

Solutions to The Critical Section Problem

- **Now Lets us combine the concept of**

- **both Algorithm 1 and Algorithm 2**

Algorithm 3

- Algorithm 3/Peterson's Solution
- 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

- **Algorithm 3/Peterson’s Solution**

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

6.2 The structure of process P_i in Peterson's solution.

Algorithm 3

- **Algorithm 3**

P0	P1

do{
    flag[0]=true;
    turn=1
    while(turn==1 && flag[1]==T);
    critical section
    flag[0]=false;
    remainder section
}while(1);
	do{
	    flag[1]=true;
	    turn=0
	    while(turn==0 && flag[0]==T);
	    critical section
	    flag[1]=false;
	    remainder section
	}while(1);

```
do {

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

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

6.2 The structure of process P_i in Peterson's solution.

Algorithm 3

- **Algorithm 3**

P0	P1

do{
flag[0]=true;
turn=1
while(turn==1 &&
flag[1]==T);
critical section
flag[0]=false;
remainder section
}while(1);
	do{
	flag[1]=true;
	turn=0
	while(turn==0 && flag[0]==T);
	critical section
	flag[1]=false;
	remainder section
	}while(1);

turn=0/1

Boolean Array flag[2]

[0]	[1]
F	F

Algorithm 3 :
Mutual Exclusion Check

If P0 is executing critical section,
Can another process P1 enter the critical section or not?

If P0 is executing CS, Can another process P1 enter the CS or not? No

- **Case 1**

P0

do{
flag[0]=true;
turn=1
while(turn==1 &&
flag[1]==T);
critical section
flag[0]=false;
remainder section
}while(1);

- For P0
- Sets flag as true
- turn=1
- P1 is not interested
- while (T&&F);
- while(F);
- exits while loop
- P0 executes CS

P1

do{
flag[1]=true;
turn=0
while(turn==0 && flag[0]==T);
critical section
flag[1]=false;
remainder section
}while(1);

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
        critical section
    flag[i] = FALSE;
        remainder section
} while (TRUE);
```

[0]	[1]
T	F

urn=1

oolean Array flag[2]

If P0 is executing CS, Can another process P1 enter the CS or not? No

- **Case 2**

P0

do{
flag[0]=true;
turn=1
while(turn==1 &&
flag[1]==T);
critical section
flag[0]=false;
remainder section
}while(1);

- For P0
- Interested
- Already Inside CS

P1

do{
flag[1]=true;
turn=0
while(turn==0 && flag[0]==T);
critical section
flag[1]=false;
remainder section
}while(1);

- P1 tries to enter CS
- set flag[1]=T,
- turn=0
- while(T&&T);
- while(T);
- gets trapped
- goes in Infinte loop

[0]	[1]
T	T

urn=0
3oolean Array flag[2]

Can P1 enter CS while P0 is in Remainder Section ?

Can P1 enter CS while P0 is in Remainder Section ?Yes

- **Case 2**

P0

do{
flag[0]=true;
turn=1
while(turn==1 &&
flag[1]==T);
critical section
flag[0]=false;
remainder section
}while(1);

- For P0
- after P0 comes out of CS
- Sets flag to False

P1

do{
flag[1]=true;
turn=0
while(turn==0 && flag[0]==T);
critical section
flag[1]=false;
remainder section
}while(1);

- P1 tries to enter CS
- set flag[1]=T,
- turn=0
- while(T&&F);
- while(F);
- control comes out of while loop
- P1 Executes CS

[0]	[1]
F	T

urn=0

Boolean Array flag[2]

Algorithm 3 :
Mutual Exclusion Check

Satisfied!!!!!

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya
TRUST

Algorithm 3:
Progress Requirement Check

K J Somaiya College of Engineering

If both P0,P1 want to enter CS

- **Algorithm 3**

P0

```
do{
      flag[0]=true;
      turn=1
      while(turn==1 &&
flag[1]==T);
      critical section
       flag[0]=false;
      remainder section
}while(1);
```

2

- For P0
- Now P0 is also interested
- sets flag as tru
and turn=1
- while(T&&T);
  while(T);
  P0 gets
trapped
- Now Context Switch
- Doesn't enter CS

P1

```
do{
      flag[1]=true;
      turn=0
      while(turn==0 && flag[0]==T);
      critical section
       flag[1]=false;
      remainder section
while(1);
```

1

- For P1
- Suppose P1 is interested
- sets flag as true and turn=0
- Context Switch occurs
- As P0 had set turn as 1
- Now P1 tries
- while(F&&T);
- while(F);
- P1 enters CS

3

[0]	[1]
T	T

urn=0/1

Boolean Array flag[2]

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Algorithm 3 :
Progress Check

Satisfied!!!!!

Algorithm 3 :
Bounded Waiting Check

Pi will enter the critical section (progress) after at most one entry by Pj (bounded waiting) i.e.
P0 will enter the critical section (progress) after at most one entry by P1 (bounded waiting).

Every process gets a fair chance.

Satisfied!!!!!

Solutions to The Critical Section Problem

Algorithm 3/Peterson’s Solution

- 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

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Hardware Synchronization

Hardware Based Solutions to The CS Problem

- Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures.

Hardware Synchronization

- **Hardware Solutions- Simple tool-a lock.**
- Race conditions are prevented by requiring that critical regions be protected by locks.

Hardware Synchronization

- A process must acquire a lock before entering a critical section;
- It releases the lock when it exits the critical section.

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

Solution to the critical-section problem using locks.

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Hardware Synchronization

- Special Atomic Hardware Instructions-
  - Atomic = Non-Interruptable

1) **Test Memory word and Set value-Test and Set()**
2) **Swap contents of two memory words-Swap()**

test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
  {
      boolean rv = *target;
      *target = TRUE;
      return rv:
  }
```

- Executed atomically
  - If 2 test and set instructions are executed simultaneously , they will be executed sequentially in some arbitary order.

test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
  {
      boolean rv = *target;
      *target = TRUE;
      return rv:
  }
```

- Returns the original value of passed parameter i.e False
- Set the new value of passed parameter to “TRUE”.

Mutual-exclusion implementation with TestAndSet ()

- Implements mutual exclusion
- Lock=Shared /Global Boolean variable , initialized to FALSE

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Mutual-exclusion implementation with TestAndSet ()

- Solution:

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

      // critical section

   lock = FALSE;

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

Mutual-exclusion implementation with TestAndSet ()

P0

```
do{
      while(TestAndSet(&lock));
       critical section
       lock=FALSE;
       remainder section
}while(1);
```

- P0 tries to enter CS
- T&S returns False and set lock=True
- While(False);
- P0 comes out of while
- Enter CS

- Now P1 tries to enter CS
- P0 is still inside CS and lock=true
- Test&Set Returns the original value of passed parameter i.e. True
- While(True);
- Goes in Infinte Loop/Do Nothing
- so P1 cannot enter CS

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Mutual-exclusion implementation with TestAndSet ()

P0

do{
    while(TestAndSet(&lock));
    critical section
    lock=FALSE;
    remainder section
}while(1);

- After completing CS
- P0 makes lock=false
- Exits CS
- Enters RS

- Now P1 can enter CS as lock is false now
- Comes out of while loop
- Executes CS

swap Instruction

Definition:

```
void Swap(boolean *a, boolean *b) {
   boolean temp = *a;
   *a = *b;
   *b = temp;
}
```

- Swaps the contents of 2 memory word
- Executed atomically=Non-interruptable

Mutual-exclusion implementation with swap ()

```
do {
   key = TRUE;
   while (key == TRUE)
      Swap(&lock, &key);

      // critical section

   lock = FALSE;

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

Mutual-exclusion implementation with the Swap() instruction.

- Solution:
- Key=Local Variable=Each process has its own Key
- Lock=Global Variable=Common for all Processes
- Both Initialized to False

Mutual-exclusion implementation with swap ()

- Solution:

```
do {
   key = TRUE;
   while (key == TRUE)
      Swap(&lock, &key);

      // critical section

   lock = FALSE;

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

Mutual-exclusion implementation with the Swap() instruction.

- Initilally (L,K)=(F,F)
- P0 makes key=true so
- (L,K)=(F,T)
- While key is true
- Swap values of Key and Lock so
- (L,K)=(T,F)
- Key becomes false, exits while loop
- Enters CS

- Now P1 tries to enter CS
- P1's own key is also True Initially
- Global Lock is True as P0 is in CS
- Key=true so
- (L,K)=(T,T)
- After Swapping also,
- Key will always be True, So trapped in while loop
- P1 will not be able to enter CS

Mutual-exclusion implementation with swap ()

- Solution:

```
do {
   key = TRUE;
   while (key == TRUE)
      Swap(&lock, &key);

      // critical section

   lock = FALSE;

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

Mutual-exclusion implementation with the Swap() instruction.

- P0
- Now, Lock=False and P0 exits CS
- P0 enters RS

- Now P1 tries to enter CS
- So Pair (L,K)=(F,T)
- The value will be swapped
- Key will become False
- Pair(L,K)=(T,F)
- Control comes out of while
- Enters CS

Hardware Synchronization

1) **Test Memory word and Set value-Test and Set()**
2) **Swap contents of two memory words-Swap()**

These algorithms do not satisfy the bounded waiting requirement

Bounded Waiting implementation with test and set ()

- Solution:

```
do {
   waiting[i] = TRUE;
   key = TRUE;
   while (waiting[i] && key)
      key = TestAndSet(&lock);
   waiting[i] = FALSE;

      // critical section

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;

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

Entry Code

Exit Code

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Bounded Waiting implementation with test and set ()

```
do {
   waiting[i] = TRUE;
   key = TRUE;
   while (waiting[i] && key)
      key = TestAndSet(&lock);
   waiting[i] = FALSE;

      // critical section

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;

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

- Solution: Satisfies all 3 critical section requirements
- Shared Data strcutures/Global variables
  - boolean waiting[n];
  - boolean lock;
- Both initialized to false

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Bounded Waiting implementation with test and set ()

```
do {
   waiting[i] = TRUE;
   key = TRUE;
   while (waiting[i] && key)
      key = TestAndSet(&lock);
   waiting[i] = FALSE;

      // critical section

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;

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

- Lock is initialized to False
- Pi can enter its critical section only if either waiting[i]==false or key==false
- The value of Key can become false only if Test&Set() is executed
- The 1st process to execute the Test&Set() will set key==false and lock to True
- waiting[i] of Pi=False
- All other processes must wait

Bounded Waiting implementation with test and set ()

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

        // critical section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;

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

- Lock=false
- For Pi , waiting and Key both are True
- while Condtn=True
    - Inside While Loop,
    - Key=T&S(&Lock)
    - So Key=False(Returns Original Value of Lock=Fa
    - Lock=True
- exit form While loop as Key=false
- Waiting of Pi finishes=false
- Enters Critical Section
- Now Pj tries to enter Critical Section
- waiting and key Both are True,
- Lock=True
- while conditn=True
    - Inside While Loop
    - Key=T&S(&Lock)
    - Key=True
    - Lock=True
    - Trapped in while loop

Bounded Waiting implementation with test and set ()

- The hardware-based solutions to the CSP are complicated for application programmers to use.

- Soln= Semaphore

Semaphore

- A synchronization tool
- A semaphore S is an integer variable that,
  - apart from initialization,
  - is accessed only through two standard atomic operations:
    - wait () and
    - signal ()

Semaphore

- wait () operation = originally termed P
  - from the Dutch *proberen,*
  - *Meaning* "to test“ or “to attempt”
- signal() operation = originally called V
  - from *verhogen,*
  - *Meaning* "to increment"

Semaphore

- Classical Definition of Wait-

```
wait(S) {
while S <= 0
;  // no-op
s--;
}
```

- The testing of the integer value of S (S<=0), as well as its possible modification (S--), must be executed without interruption.

Semaphore

- Classical Definition of Signal-

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

Semaphore

- All modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly.
  - That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

Usage of Semaphore

- To Deal with n-process Critical Section Problem, i.e. Mutual Exclusion
  - binary semaphores to deal with the critical-section problem for multiple processes.
  - binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion

- To Solve Synchronization Problem

Types of Semaphore

- Types of Semaphores
  - Counting
  - Binary

Types of Semaphore

- Counting Semaphore-
  - The value of a counting semaphore can range over an unrestricted domain.

- Binary Semaphore-
  - The value of a binary semaphore can range only between 0 and 1.
  - Also known as mutex locks,
  - As they provide mutual exclusion.

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Counting Semaphore

- Used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available.

Counting Semaphore

- Each process that wishes to use a resource performs a wait() operation on the semaphore ,
  - thereby decrementing the count.

- When a process releases a resource, it performs a signal() operation
  - incrementing the count.

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Counting Semaphore

- **When the count for the semaphore goes to 0, all resources are being used.**
  - **After that, processes that wish to use a resource will block until the count becomes greater than 0.**

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

ISRO | ISRO CS 2017 – May | Question 78

At particular time, the value of a counting semaphore is 10, it will become 7 after:
(a) 3 V operations
(b) 3 P operations
(c) 5 V operations and 2 P operations
(d) 2 V operations and 5 P operations

Which of the following option is correct?
(A) Only (b)
(B) Only(d)
(C) Both (b) and (d)
(D) None of these

At particular time, the value of a counting semaphore is 10, it will become 7 after:
(a) 3 V operations
(b) 3 P operations
(c) 5 V operations and 2 P operations
(d) 2 V operations and 5 P operations
Which of the following option is correct?
(A) Only (b)
(B) Only(d)
(C) Both (b) and (d)
(D) None of these

Answer: (C)
Explanation: P: Wait operation decrements the value of the counting semaphore by 1.
V: Signal operation increments the value of counting semaphore by 1.
Current value of the counting semaphore = 10
a) after 3 P operations, value of semaphore = 10-3 = 7
d) after 2 v operations, and 5 operations value of semaphore = 10 + 2 – 5 = 7
Hence option (C) is correct.

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

UGC-NET | UGC NET CS 2018 July – II | Question 51

At a particular time of computation, the value of a counting semaphore is 10. Then 12 P operations and “x” V operations were performed on this semaphore. If the final value of semaphore is 7, x will be:

(A) 8
(B) 9
(C) 10
(D) 11

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

UGC-NET | UGC NET CS 2018 July – II | Question 51

At a particular time of computation, the value of a counting semaphore is 10. Then 12 P operations and “x” V operations were performed on this semaphore. If the final value of semaphore is 7, x will be:

(A) 8
(B) 9
(C) 10
(D) 11

Answer: (B)

Explanation: Intially the value of a counting semaphore is 10 Now 12 P operation are performed.
Now counting semaphore value = -2
“x” V operations were performed on this semaphore and final value of counting semaphore = 7
i.e x + (-2) = 7
x = 9.
So, option (C) is correct.

Binary Semaphore

Mutual Exclusion using Binary Semaphore

- Used to deal with the critical-section problem for multiple processes.
- Processes share a semaphore, mutex, initialized to 1.

```
do {
   wait(mutex);

      // critical section

   signal(mutex);

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

```
wait(S) {
while S <= 0
;  // no-op
s--;
}
```

- Process P0 tries to enter the CS

P0

while(mutex<=0);

mutex=1 so Condition is False

Comes out of while loop

mutex - -

so mutex=0

CS

```
do {
   wait(mutex);

      // critical section

   signal(mutex);

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

```
wait(S) {
while S <= 0
;  // no-op
s--;
}
```

Now P1 tries to enter CS

while(mutex<=0);

condition is True , so P1 gets trapped in a Do Nothing Loop

P1 cannot enter CS ============>ME preserved

Synchronization using Semaphore

- We can also use semaphores to solve various synchronization problems.
- Consider two concurrently running processes:
  - P1 with a statement S1
  - P2 with a statement S2 .
  - Suppose we require that S2 be executed only after S1 has completed.

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Synchronization using Semaphore

- Let P1 and P2 share a common semaphore synch, initialized to 0
- Statements inserted in P1

  S1;

  signal(synch) ;
- Statements inserted in P2

  wait(synch);

  S2;

```
wait(S) {
while S <= 0
;  // no-op
s--;
}
```

Synchronization using Semaphore

- P1

    S1;
    signal(synch) ;

- P2

    wait(synch);
    S2;

- Because synch is initialized to 0,
- P2 will execute S2 only after P1 has invoked signal (synch), which is after statement S1 has been executed.
    - Else P2 will caught in infinte loop inside wait() fn

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Semaphore

- Must guarantee that no two processes can execute the wait() and signal() on the same semaphore at the same time

- Thus, the implementation becomes the critical section problem
  - the wait and signal code are placed in the critical section

Busy Waiting

- While a process is in its critical section,
  - any other process that tries to enter its critical section must loop continuously in the entry code=>Infinite Loop
  - This continual looping is clearly a problem in a real multiprogramming system where a single CPU is shared among many processes.
  - Busy waiting wastes CPU cycles that some other process might be able to use productively.

Busy Waiting

- Disadvantage
  - Busy Waiting

- Software Solutions i.e. Algo 1, Algo2, Peterson’s Solution and Semaphore definition
  - all suffer from Busy Waiting

Busy Waiting

- While a process is in its critical section,
  - This type of semaphore is also called a Spinlock
  - The process "spins" while waiting for the lock.
  - Advantage of Spinlocks-
    - **No context switch is required when a process must wait on a lock, and a context switch may take considerable time.**

Busy Waiting

- Thus, when locks are expected to be held for short times, spinlocks are useful;

- Often employed on multiprocessor systems where
  - one thread can "spin" on one processor while
  - another thread performs its critical section on another processor.

Busy Waiting

- Little busy waiting if critical section rarely occupied

- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

Semaphore Definition

- Define a semaphore as a "C' struct

- Each semaphore has
  - an integer value
  - a list of processes-"list"

```
typedef struct {
        int value;
        struct process *list;
} semaphore;
```

Semaphore Definition

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Semaphore Definition

- Wait()
  - When a process must wait on a semaphore, it is added to the list of processes.
- Signal()
  - A signal() operation removes one process from the list of waiting processes and awakens that process

```
typedef struct {
      int value;
      struct process *list;
} semaphore;
```

Implementation with no busy waiting

- We modify the definition of the wait() and signal() semaphore operations.

- When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait.

- Rather than busy waiting, the process can *block* itself.

Implementation with no busy waiting

- Two operations:
  - **block**
  - **wakeup**

Implementation with no busy waiting

- Two operations:
  - **block** –
  - The block() operation suspends the process that invokes it.
  - place the process invoking the operation on the waiting queue associated with the semaphore
  - The state of the process is switched to the waiting state.
  - Control transferred to the CPU scheduler, which selects another process to execute.

Implementation with no busy waiting

wakeup –

- A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation.

- The process is restarted by a wakeup () operation

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Implementation with no busy waiting

wakeup –

- The wakeup(P) operation resumes the execution of a blocked process P.
- Remove one of processes in the waiting queue and place it in the ready queue
- Changes the process state from waiting to ready
- These two operations are provided by the operating system as basic system calls.

Implementation with no busy waiting

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

SOMAIYA VIDYAVIHAR UNIVERSITY

K J Somaiya College of Engineering

Somaiya TRUST

Implementation with no busy waiting

- Semaphore values may be negative,
- If a semaphore value is negative, its magnitude is the no of processes waiting on that semaphore.

- Binary Semaphore values are never negative under the classical definition of semaphores with busy waiting.

Implementation with no busy waiting

- The list of waiting processes can be easily implemented by
  - a link field in each process control block (PCB).
- Each semaphore contains
  - an integer value and
  - a pointer to a list of PCBs.

Implementation with no busy waiting

- One way to add and remove processes from the list
  - so as to ensure bounded waiting is to use a FIFO queue,
  - where the semaphore contains both head and tail pointers to the queue.
  - In general, the list can use *any* queueing strategy.

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Deadlock and Starvation

- The event in question is execution of signal operation
- Let S and Q be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
```

Deadlock and Starvation

P_0 P_1

```
        wait(S);                    wait(Q);
       wait(Q);                     wait(S);
...                    ...
       signal(S);                          signal(Q);
          signal(Q);                       signal(S);
```

- P0 executes wait(S) then P1 executes wait(Q)
- When P0 executes wait(Q), it must wait until P1 executes signal(Q)
- When P1 executes wait(S), it must wait until P0 executes signal (S)

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Deadlock and Starvation

- Deadlock-Every process in the set is waiting for an event that can be caused only by another waiting process in the set

- Starvation – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended

Classical Problems of Synchronization

Classical problems used to test newly-proposed synchronization schemes

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Semaphore Solution to Bounded-Buffer Problem

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Bounded-Buffer Problem

- *n* buffers, each can hold one item
- Semaphore mutex ->Binary Semaphore
  - Provides mutual exclusion for access to the buffer pool
  - initialized to the value 1
- Semaphore full ->Counting Semaphore
  - Counts no of full buffers
  - initialized to the value 0
- Semaphore empty ->Counting Semaphore
  - Counts no of empty buffers
  - initialized to the value n

  - Assuming Buffer is empty

Bounded-Buffer Problem

- The structure of the producer process

```
do {
        ...
       /* produce an item in next_produced */
        ...
     wait(empty);
     wait(mutex);
        ...
       /* add next produced to the buffer */
        ...
     signal(mutex);
     signal(full);
 } while (true);
```

- Semaphore mutex
  - initialized to the value 1
- Semaphore full
  - initialized to the value 0
- Semaphore empty
  - initialized to the value n

```
wait(S) {
while S <= 0
; // no-op
s--;
}
```

Bounded-Buffer Problem

- The structure of the producer process

Mutual Exclusion?

```
do {
        ...
       /* produce an item in next_produced */
        ...
     wait(empty);
     wait(mutex);
         ...
       /* add next produced to the buffer */
         ...
     signal(mutex);
     signal(full);
 } while (true);
```

```
wait(S) {
while S <= 0
;  // no-op
s--;
}
```

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Bounded-Buffer Problem

- The structure of the consumer process

```
do {
    wait(full);
    wait(mutex);
        ...
    /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
    /* consume the item in next consumed */
        ...
 } while (true);
```

- Semaphore mutex
  - initialized to the value 1
- Semaphore full
  - initialized to the value 0
- Semaphore empty
  - initialized to the value n

```
wait(S) {
while S <= 0
;  // no-op
S--;
}
```

Bounded-Buffer Problem

- The structure of the consumer process

Mutual Exclusion?

```
do {
    wait(full);
    wait(mutex);
        ...
    /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
    /* consume the item in next consumed */
        ...
 } while (true);
```

```
wait(S) {
while S <= 0
;  // no-op
s--;
}
```

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Readers-Writers Problem

Readers-Writers Problem

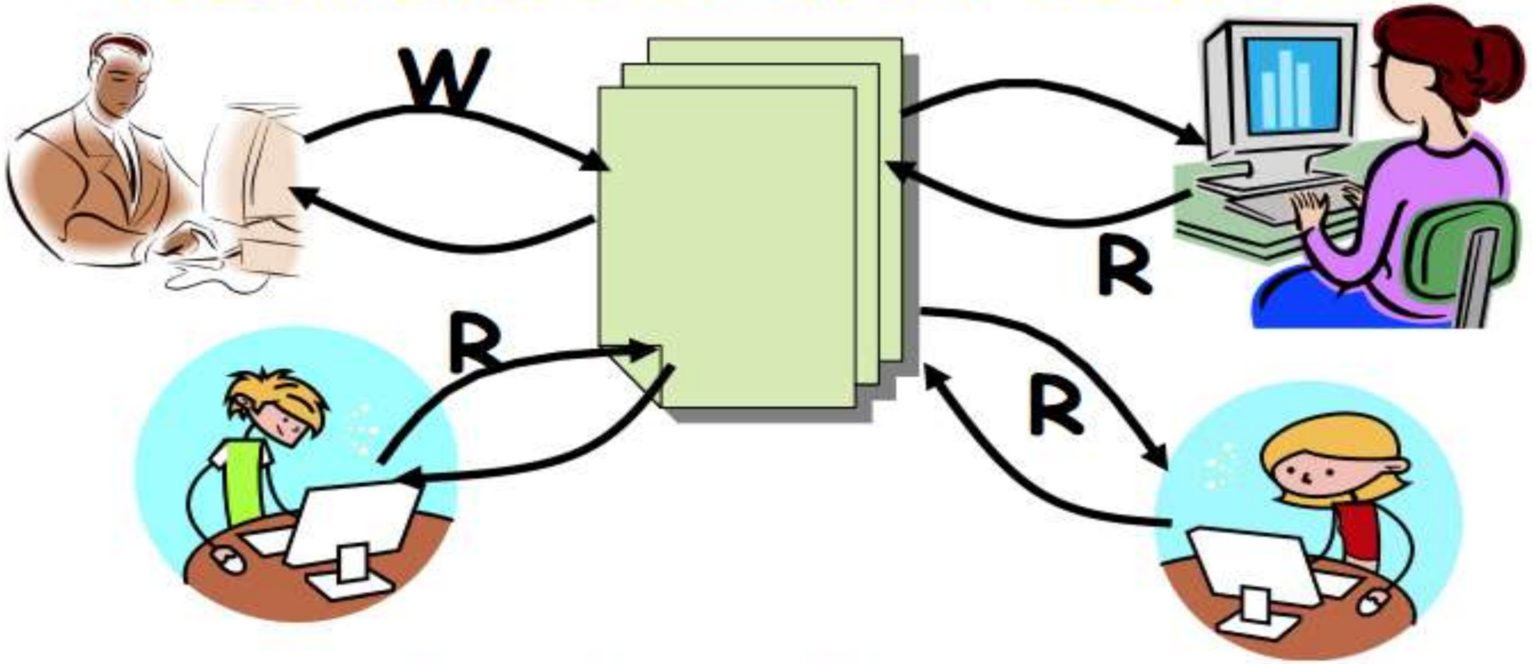
- Process Synchronization Problem

- A Database/Data set/Object/File/Record is shared among a number of concurrent processes

  - Readers – only read the data set; they do ***not*** perform any updates

  - Writers – can both read and write i.e. update

Readers/Writers Problem



Motivation: Consider a shared database

- Two classes of users:
  - » Readers – never modify database
  - » Writers – read and modify database

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Readers-Writers Problem

Problem –

- Allow multiple readers to read at the same time, no adverse effects
- If a writer and some other process(reader/writer) access shared object simultaneously chaos may ensue

Requirements

  - Writers must have exclusive access to the shared object

Readers-Writers Problem

Problem –

- R-W : Problem
- W-R : Problem
- W-W : Problem
- R-R : No Problem

Reader Writer Problem

- Several variations– all involving priorities

- First Reader Writer Problem-
  - No reader will be kept waiting unless a writer has already obtained permission to use the shared object.
  - No reader should wait for other readers to finish simply because a writer is waiting

- Second Reader Writer Problem-
  - Once a writer is ready , that writer performs its write as soon as possible
  - If a writer is waiting to access the object, no new readers may start reading

Solution to First Reader Writer Problem

- Shared Data
  - Data set
  - Semaphore **rw_mutex**
  - Semaphore **mutex**
  - Integer **read_count**

- Semaphore **rw_mutex**
  - Initialized to 1
  - **Common to both reader and writers**
  - **Mutual Exclusion semaphore for the writers**
  - **Also used by 1st or Last Reader that enters or exits CS**
  - **It is not used by readers who enter or exit while other readers are in their critical sections**
- Semaphore **mutex**
  - Initialized to 1
  - **To ensure Mutual Exclusion when variable readcount is updated**
- Integer **read_count**
  - **Initialized to 0**
  - **Keeps a track of how many processes are currently reading the object**

Readers-Writers Problem

- The structure of a writer process

```
do{
        wait(rw_mutex);
            ...
        /* writing is performed */
            ...
      signal(rw_mutex);
   } while (true);
```

- Semaphore **rw_mutex**
  - **Mutual Exclusion for the writers**
  - initialized to 1

```
wait(S) {
while S <= 0
; // no-op
S--;
}
```

Writer process:
- Writer requests the entry to critical section.
- If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
- If allowed, Performs writing
- It exits the critical section.

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Readers-Writers Problem

- The structure of a writer process
- **If a writer is in the critical section and n readers are waiting,**
  - **then one reader is queued on `rw_mutex`, and n- 1 readers are queued on mutex.**
- When a writer executes signal (**`rw_mutex`**),
  - **It resumes the execution of either the waiting readers or a single waiting writer.**
  - **The selection is made by the scheduler**

```
do{
            wait(rw_mutex);

               ...
            /* writing is performed */

               ...
          signal(rw_mutex);
     } while (true);
```

- Semaphore **`rw_mutex`**
  - **Mutual Exclusion for the writers**
  - initialized to 1

Readers-Writers Problem

- The structure of a reader process

```
do {
       wait(mutex);
       read_count++;
       if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
        ...
       /* reading is performed */
        ...
    wait(mutex);
       read count--;
       if (read_count == 0)
           signal(rw_mutex);
    signal(mutex);
} while (true);
```

- Semaphore **`rw_mutex`**
  - **`Mutual Exclusion for the writers`**
  - initialized to 1
- Semaphore **`mutex`**
  - **`ME for readcount update`**
  - initialized to 1
- **`read_count`** initialized to 0

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Readers-Writers Problem

- The structure of a reader process

```
do {
       wait(mutex);
       read_count++;
       if (read_count == 1)

        wait(rw_mutex);

     signal(mutex);

         ...
       /* reading is performed */

         ...
     wait(mutex);
       read count--;
       if (read_count == 0)

           signal(rw_mutex);

     signal(mutex);
} while (true);
```

Reader process:

- Reader requests the entry to critical section.
- If allowed:
  - it increments the count of number of readers inside the critical section.
  - If this reader is the first reader entering, it locks the rw_mutex semaphore to restrict the entry of writers if any reader is inside.
  - **It then, signals mutex as any other reader is allowed to enter while others are already reading.**

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Readers-Writers Problem

- The structure of a reader process

```
do {
      wait(mutex);
      read_count++;
      if (read_count == 1)
       wait(rw_mutex);
    signal(mutex);
        ...
      /* reading is performed */
        ...
    wait(mutex);
      read count--;
      if (read_count == 0)
          signal(rw_mutex);
    signal(mutex);
} while (true);
```

Reader process:

- After performing reading, it exits the critical section.
- **When exiting, it checks if no more reader is inside, it signals the semaphore "rw_mutex" as now, writer can enter the critical section.**
- If not allowed, it keeps on waiting.

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

- The structure of a reader process

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
         wait(rw_mutex); //First reader sets rw_mutex=0
                          //So writer cannot enter CS
            //Following Readers need not enter if statement

signal(mutex);
           ...
        /* reading is performed */
           ...
        wait(mutex);
           read count--;
           if (read_count == 0)
              signal(rw_mutex); //now writers can write
                        //accessed by last reader
        signal(mutex);
    } while (true);
```

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

R-W Problem

- The structure of a reader process

```
do {
      wait(mutex);
      read_count++;
      if (read_count == 1)
       wait(rw_mutex);
    signal(mutex);
        ...
      /* reading is performed */
        ...
    wait(mutex);
      read count--;
      if (read_count == 0)
         signal(rw_mutex);
    signal(mutex);
} while (true);
```

Reader

- Initially readcount=0
- First Reader R1 tries to enter
- wait mutex so mutex=0
- readcount=1
- if readcount is one i.e. First reader
- wait rw_mutex so rw_mutex=0
- readcount updation finished so signal mutex, mutex=1
- Reader enters CS
- “Reader R1 is reading”

Writer

- Now Writer tries to enter
- It executes his code
- wait(rw_mutex)
- gets trapped in infinite loop as rw_mutex was already 0
- Writer cannot enter CS

```
do{
          wait(rw_mutex);
               ...
          /* writing is performed */
               ...
          signal(rw_mutex);
     } while (true);
```

W-R Problem

- The structure of a reader process

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
         wait(rw_mutex);
      signal(mutex);
          ...
        /* reading is performed */
          ...
      wait(mutex);
        read count--;
        if (read_count == 0)
           signal(rw_mutex);
      signal(mutex);
} while (true);
```

Writer1

- Initally rw_mutex=1
- First Writer W1 tries to enter CS
- wait operation
- rw_mutex becomes 0
- Writer W1 enters CS

Reader1

- Initially mutex=1,readcount=0
- Now reader R1 tries to enter CS
- wait mutex
- mutex=0
- readcount=1
- In if section,
- wait(rw_mutex)
- gets trapped in infinite loop
- Reader Cannot enter CS

```
do{
          wait(rw_mutex);
               ...
          /* writing is performed */
               ...
          signal(rw_mutex);
     } while (true);
```

W-W Problem

- The structure of a reader process

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
         wait(rw_mutex);
      signal(mutex);
          ...
        /* reading is performed */
          ...
      wait(mutex);
        read count--;
        if (read_count == 0)
           signal(rw_mutex);
      signal(mutex);
} while (true);
```

Writer1

- Initally rw_mutex=1
- First Writer W1 tries to enter CS
- wait operation
- rw_mutex becomes 0
- Writer W1 enters CS

Writer2

- Another Writer W2 tries to enter CS
- wait operation
- rw_mutex was already 0
- Thus trapped in Infinite loop
- Writer2 Cannot enter CS

```
do{
        wait(rw_mutex);
           ...
        /* writing is performed */
           ...
        signal(rw_mutex);
   } while (true);
```

```
wait(S) {
while S <= 0
;  // no-op
S--;
}
```

SOMAIYA VIDYAVIHAR UNIVERSITY

K J Somaiya College of Engineering

Somaiya TRUST

R-R Problem

- The structure of a reader process

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
         wait(rw_mutex);
      signal(mutex);
           ...
        /* reading is performed */
           ...
      wait(mutex);
        read count--;
        if (read_count == 0)
           signal(rw_mutex);
      signal(mutex);
} while (true);
```

Reader1

- Initially readcount=0
- First Reader R1 tries to enter
- wait mutex so mutex=0
- readcount=1
- if readcount is one i.e. First reader
- wait rw_mutex so rw_mutex=0
- readcount updation finished so signal mutex,
- mutex=1
- Reader enters CS

Reader2

- Reader R2 tries to enter CS
- wait mutex so mutex=0
- readcount=2
- Does not enter If section
- readcount updation finished so signal mutex,
- mutex=1
- Reader R2 Enters CS

```
do{
          wait(rw_mutex);
               ...
          /* writing is performed */
               ...
          signal(rw_mutex);
    } while (true);
```

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Readers-Writers Problem

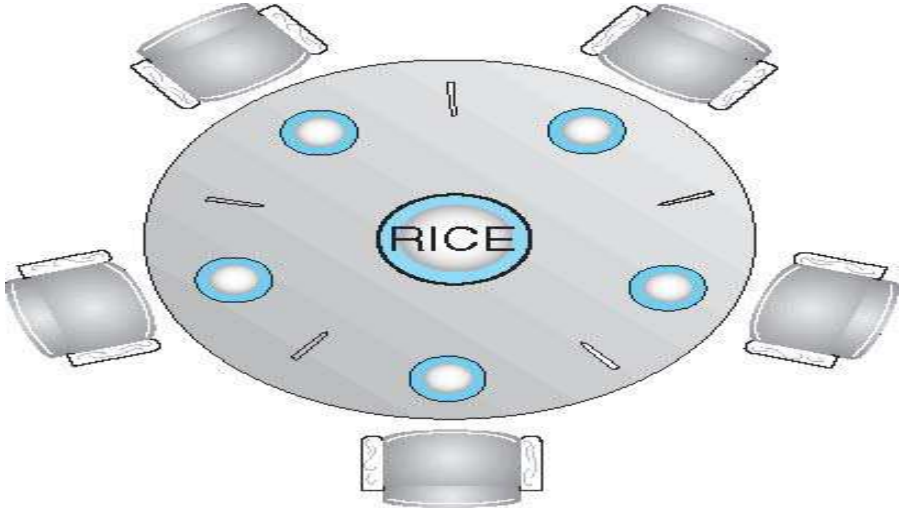
- Both Soln may lead to Starvation
- First Reader Writer Problem-
  - Writers may starve

- Second Reader Writer Problem-
  - Readers may starve

- For this reason, other variants of the problem have been proposed.

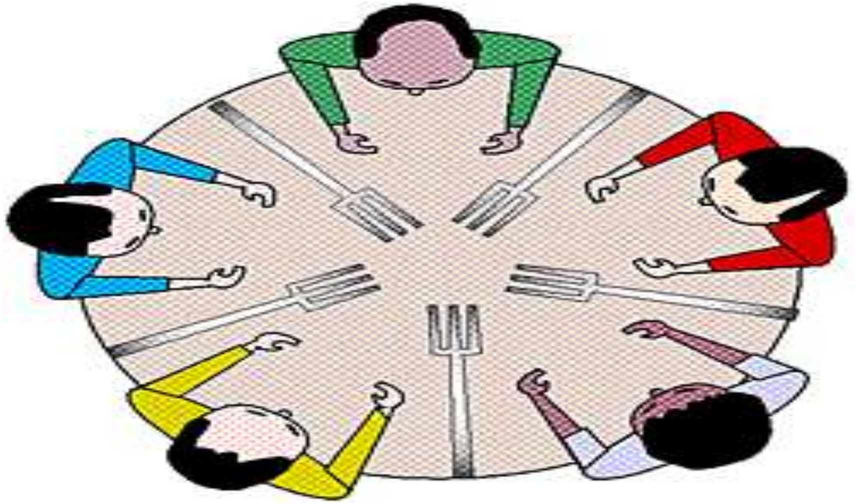
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem



- 5 Philosophers spend their lives alternating thinking and eating

Dining-Philosophers Problem



- When thinking
  - Don’t interact with their neighbors
- When hungry-
  - A philosopher needs both their right and left chopstick to eat.
  - A hungry philosopher may only eat if there are both chopsticks available

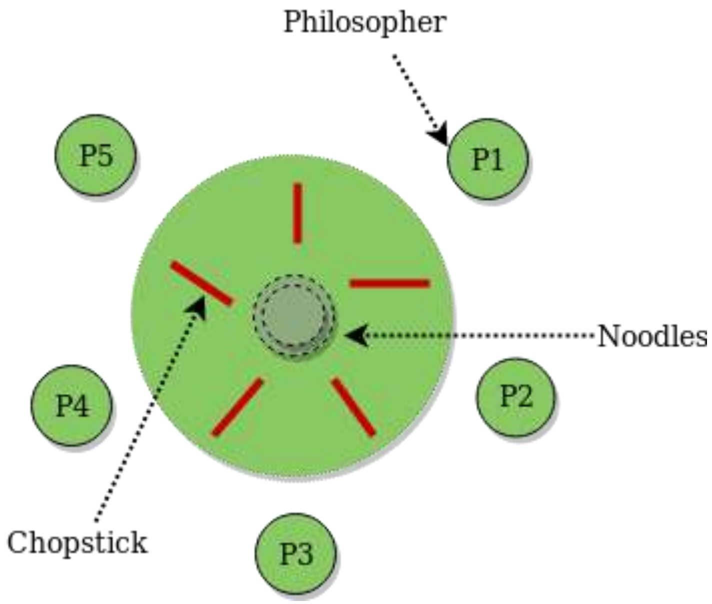
Dining-Philosophers Problem

- Occasionally try to pick up 2 chopsticks that are closest to her to eat from bowl
  - Chopsticks that are between her and her left and right neighbor
  - Pick up only one at a time
  - Need both to eat, then release both when done

Dining-Philosophers Problem

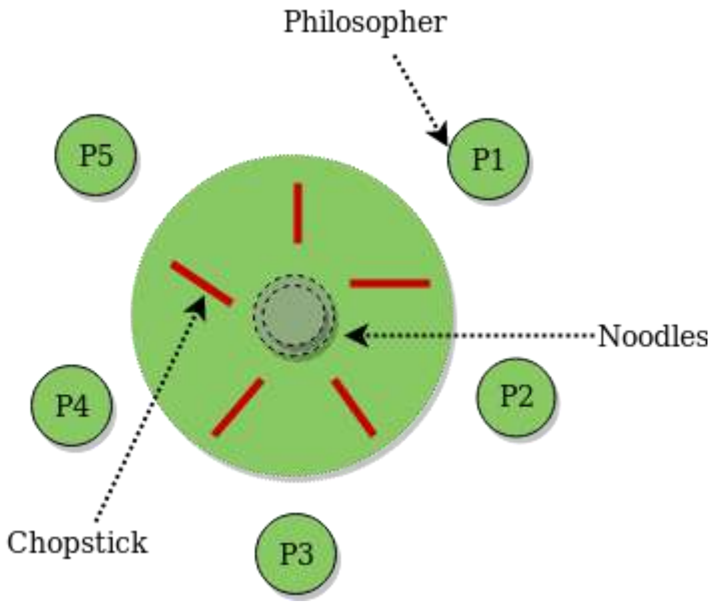
- Classic Synchronization problem
- Example of large class of concurrency control problems
- Represents the need to allocate several resources among several processes
  - in a deadlock free and starvation free manner

Dining-Philosophers Problem



- 5 Philosophers share a common circular table
  - Surrounded by 5 chairs=each belonging to one philosopher
  - Center of table ->Bowl of rice
  - Five single chopsticks

Dining-Philosophers Problem



- Shared data?
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Semaphore Solution for Dining-Philosophers Problem

- Grab the chopstick –By Executing wait operation on the semaphore
- Release the chopstick-By executing the signal operation on the appropriate semaphore
- Array of Semaphores chopstick [5] initialized to 1

Semaphore Solution for Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
     wait (chopstick[i] );
      wait (chopStick[ (i + 1) % 5] );

                 //  eat

      signal (chopstick[i] );
      signal (chopstick[ (i + 1) % 5] );

                  //  think

} while (TRUE);
```

Philosopher i has picked up the chopsticks on his sides.

Then the eating function is performed.

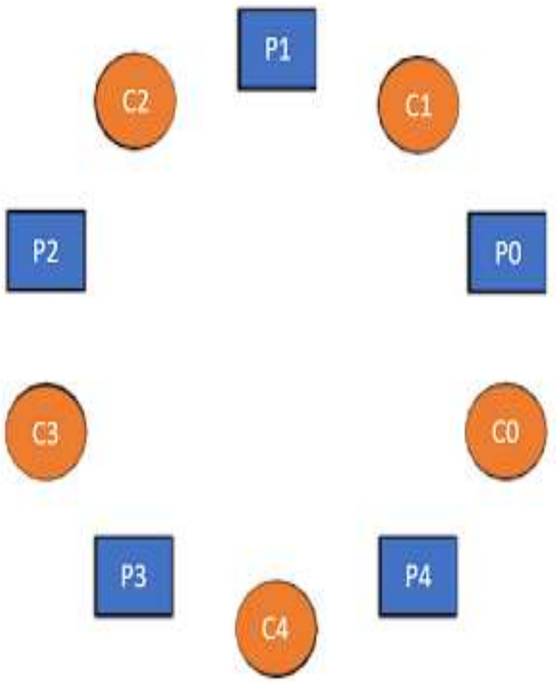
Philosopher i has eaten and put down the chopsticks on his sides.

Then the philosopher goes back to thinking.

Semaphore Solution for Dining-Philosophers Problem Algorithm

```
do {
    wait (chopstick[i] );
  wait (chopStick[ (i + 1)
  % 5] );

            //  eat

  signal (chopstick[i] );
  signal (chopstick[ (i +
  1) % 5] );

                //  think

} while (TRUE);
```

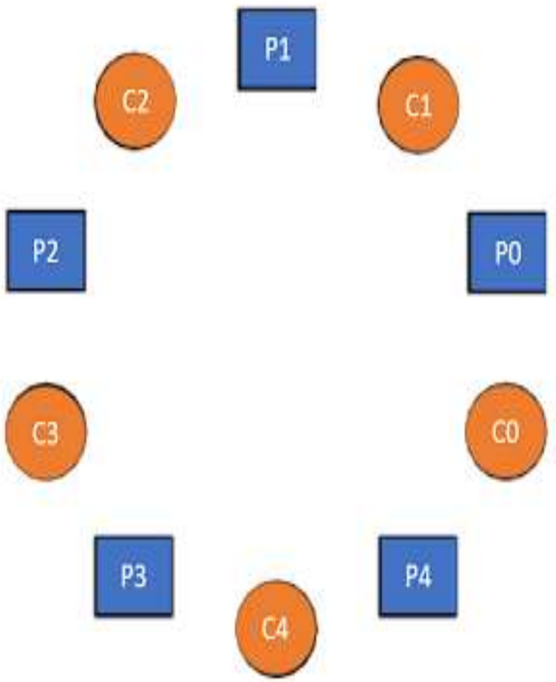
```
do {
i=0
    wait (chopstick[0] );
  wait (chopStick[1] );

          //  eat

  signal (chopstick[0] );
  signal (chopstick[1] );

              //  think

} while (TRUE);
```



P = Philosopher
C = Chopstick

Semaphore Solution for Dining-Philosophers Problem Algorithm

do {
i=1
wait (chopstick[1]);
wait (chopStick[2]);
// eat
signal (chopstick[1]);
signal (chopstick[2]);
// think
} while (TRUE);
	do {
	i=2
	wait (chopstick[2]);
	wait (chopStick[3]);
	// eat
	signal (chopstick[2]);
	signal (chopstick[3]);
	// think
	} while (TRUE);

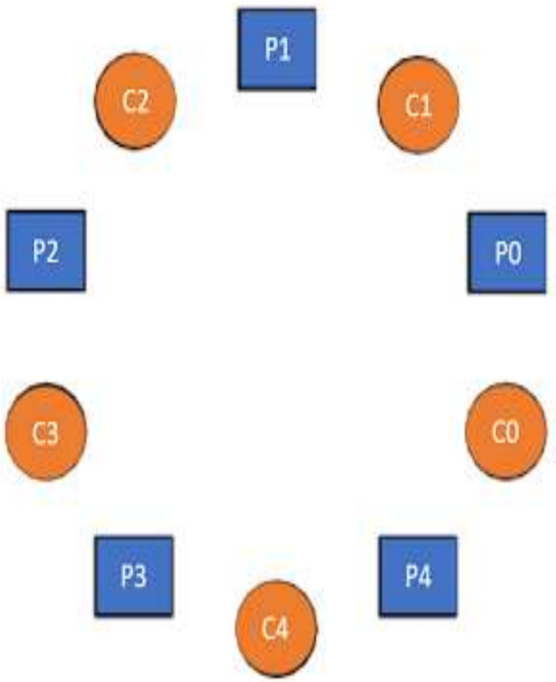


P = Philosopher
C = Chopstick

Semaphore Solution for Dining-Philosophers Problem Algorithm

```
do {
i=3
    wait (chopstick[3] );
  wait (chopStick[4] );

             //  eat

  signal (chopstick[3] );
  signal (chopstick[4] );

                 //  think

} while (TRUE);
```

```
do {
i=4
    wait (chopstick[4] );
  wait (chopStick[4+1%5=0] );

             //  eat

  signal (chopstick[4] );
  signal (chopstick[0] );

                 //  think

} while (TRUE);
```



P = Philosopher
C = Chopstick

Semaphore Solution for Dining-Philosophers Problem Algorithm

What is the problem with this algorithm?

Semaphore Solution for Dining-Philosophers Problem Algorithm

- Algorithm Guarantees that No two neighbors are eating simultaneously

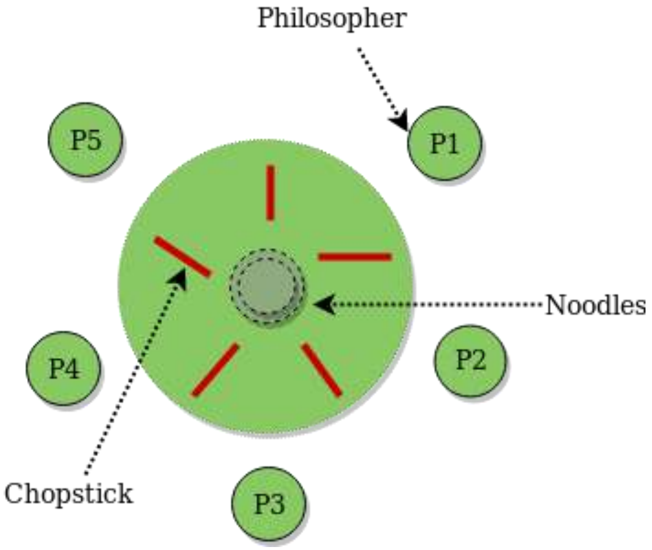
Still Must be rejected

- Why?

Semaphore Solution for Dining-Philosophers Problem Algorithm

Why?

- Possibility of deadlock
- If all 5 philosophers are hungry simultaneously and each grabs left chopstick
- All elements of chopsticks will be =0
- When each philosopher tries to grab her right chopstick, delayed forever



SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Semaphore Solution for Dining-Philosophers Problem Algorithm

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section)
  - Use an asymmetric solution --
    - Odd-numbered philosopher picks up first the left chopstick and then the right chopstick.
    - Even-numbered philosopher picks up first the right chopstick and then the left chopstick

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

- Deadlock and starvation are possible.

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Q1)The program follows to use a shared binary semaphore T.

Process A

int Y;

A1: Y = X*2;

A2: X = Y;

signal(T);

Process B

int Z;

B1: wait(T);

B2: Z = X+1;

X = Z;

T is set to 0 before either process begins execution and, as before, X is set to 5.
Now, how many different values of X are possible after both processes finish executing?
a) one
b) two
c) three
d) four

Q1)The program follows to use a shared binary semaphore T.

Process A

int Y;

A1: Y = X*2;

A2: X = Y;

signal(T);

Process B

int Z;

B1: wait(T);

B2: Z = X+1;

X = Z;

T is set to 0 before either process begins execution and, as before, X is set to 5.
Now, how many different values of X are possible after both processes finish executing?
a) one
b) two
c) three
d) four

Explanation: The semaphore T ensures that all the statements from A finish execution before B begins.

So now there is only one way in which statements from A and B can be interleaved:
A1 A2 B1 B2: X = 11.

Ans c)

Q2) The bounded buffer problem is also known as _____________

a) Readers – Writers problem

b) Dining – Philosophers problem

c) Producer – Consumer problem

d) None of the mentioned

Q2) The bounded buffer problem is also known as ____________
a) Readers – Writers problem
b) Dining – Philosophers problem
c) Producer – Consumer problem
d) None of the mentioned

Answer: c

Q3) All processes share a semaphore variable mutex, X initialized to 1.

Each process must execute wait(mutex) before entering the critical section and signal(mutex) afterward.

Suppose a process executes in the following manner:

signal(mutex)

______

critical section

____

wait(mutex);

In this situation _____________________

a) a deadlock will occur

b) processes will starve to enter critical section

c) several processes maybe executing in their critical section

d) all of the mentioned

Q3) All processes share a semaphore variable mute, X initialized to 1.

Each process must execute wait(mutex) before entering the critical section and signal(mutex) afterward.

Suppose a process executes in the following manner:

signal(mutex)

______

critical section

____

wait(mutex);

In this situation _____________________

a) a deadlock will occur

b) processes will starve to enter critical section

c) several processes maybe executing in their critical section

d) all of the mentioned

Answer: c

Q4) All processes share a semaphore variable mutex, X initialized to 1.

Each process must execute wait(mutex) before entering the critical section and signal(mutex) afterward.

Suppose a process executes in the following manner:

waitl(mutex)

______

critical section

____

wait(mutex);

In this situation _____________________

a) a deadlock will occur

b) processes will starve to enter critical section

c) several processes maybe executing in their critical section

d) all of the mentioned

Q4) All processes share a semaphore variable mutex initialized to 1.

Each process must execute wait(mutex) before entering the critical section and signal(mutex) afterward.

Suppose a process executes in the following manner:

waitl(mutex)

______

critical section

____

wait(mutex);

In this situation _____________________

a) a deadlock will occur

b) processes will starve to enter critical section

c) several processes maybe executing in their critical section

d) all of the mentioned

Answer: a

Q5) In the bounded buffer problem, there are the empty and full semaphores that _____________

a) count the number of empty and full buffers

b) count the number of empty and full memory spaces

c) count the number of empty and full queues

d) none of the mentioned

Q5) In the bounded buffer problem, there are the empty and full semaphores that _____________

a) count the number of empty and full buffers

b) count the number of empty and full memory spaces

c) count the number of empty and full queues

d) none of the mentioned

Answer: a

Q6) The wait operation of the semaphore basically works on the basic ________ system call.
a) stop()
b) block()
c) hold()
d) wait()

Q6) The wait operation of the semaphore basically works on the basic ________ system call.

a) stop()
b) block()
c) hold()
d) wait()

Answer: b

Monitors

Monitors

- A high-level synchronization construct

- *Set of programmer defined operators*

Syntax of Monitors

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

     Initialization code (…) { … }
  }
```

Monitors

- *Declaration of variables*
  - *whose value define the state of an instance of the type*

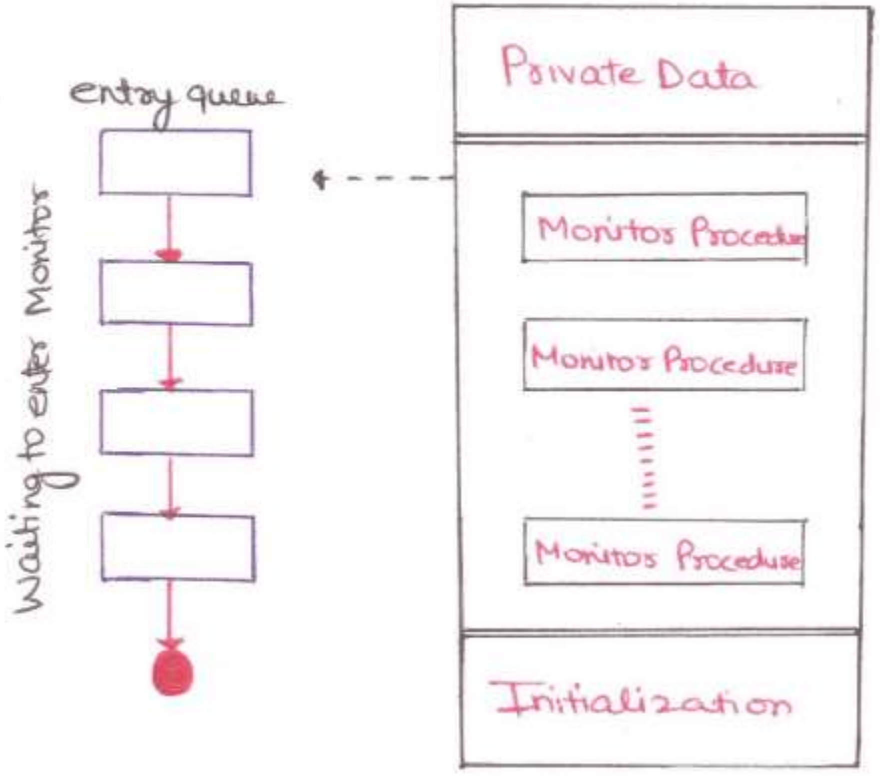
- *Bodies of procedures*
  - *that implement operations on the type*

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

     Initialization code (…) { … }
}
```

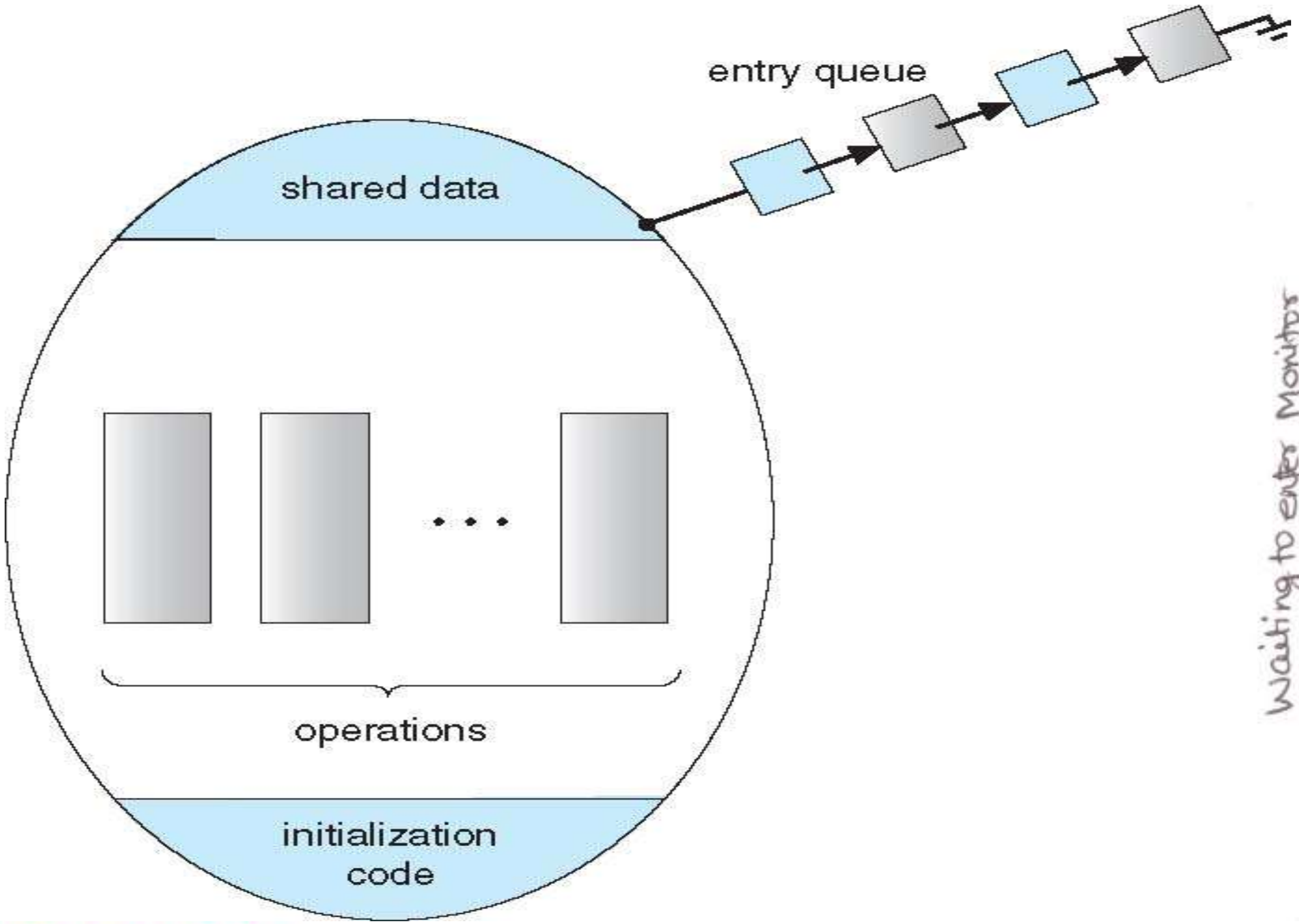
Monitors

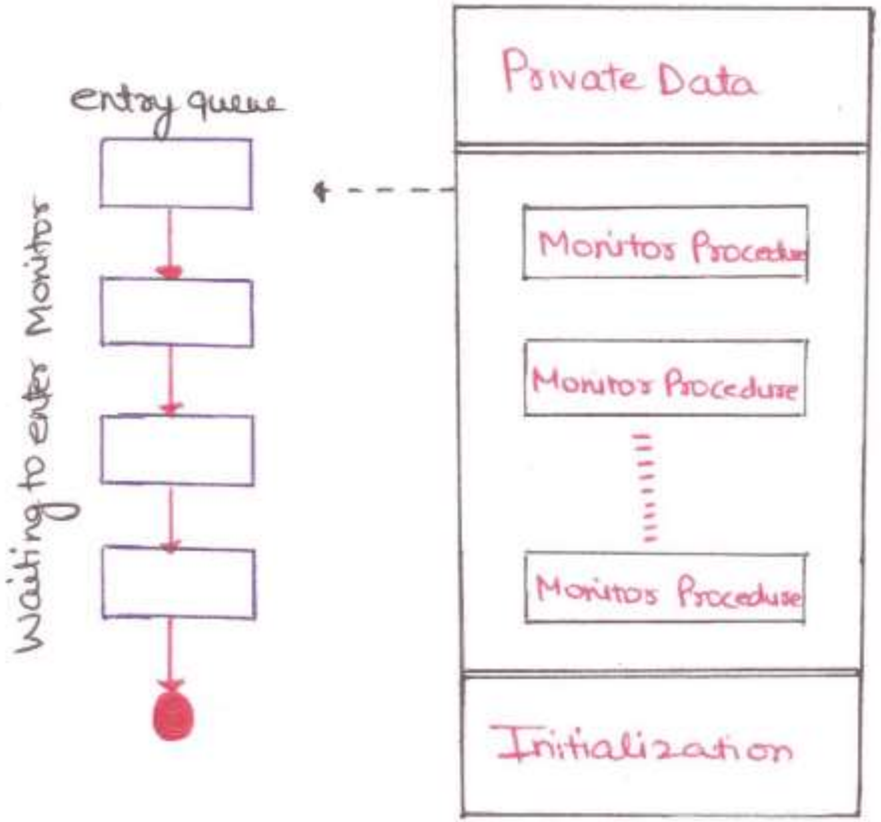
- Internal variables only accessible by code within the procedure

- Procedure defined within the monitor can access only those variables
  - declared locally within the monitor and
  - its formal parameters.



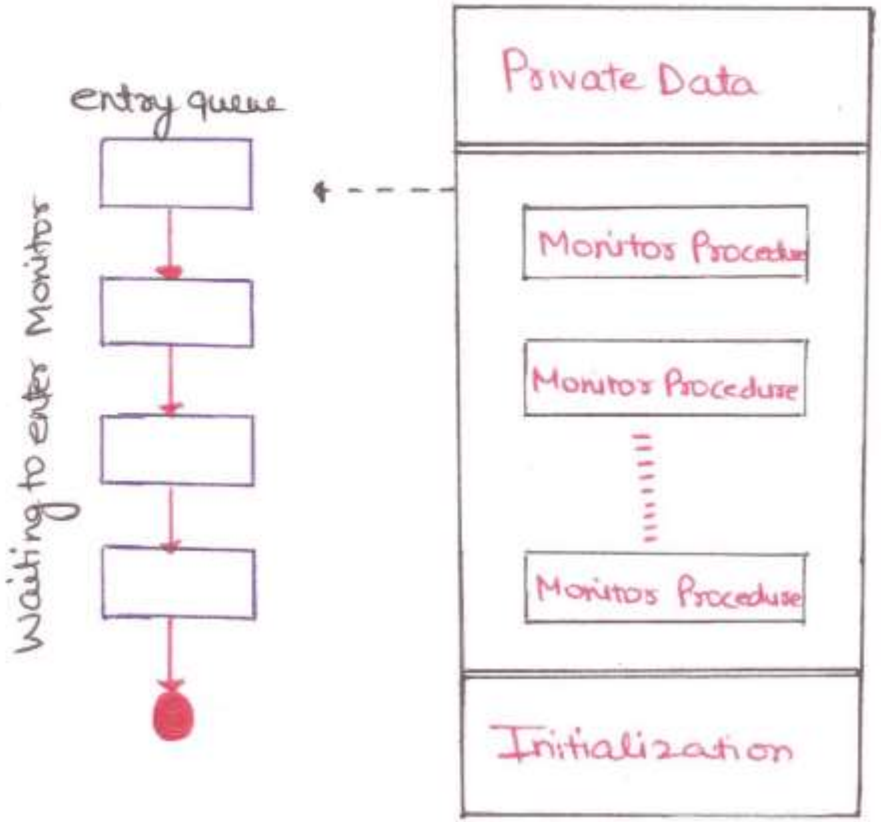
Schematic view of a Monitor





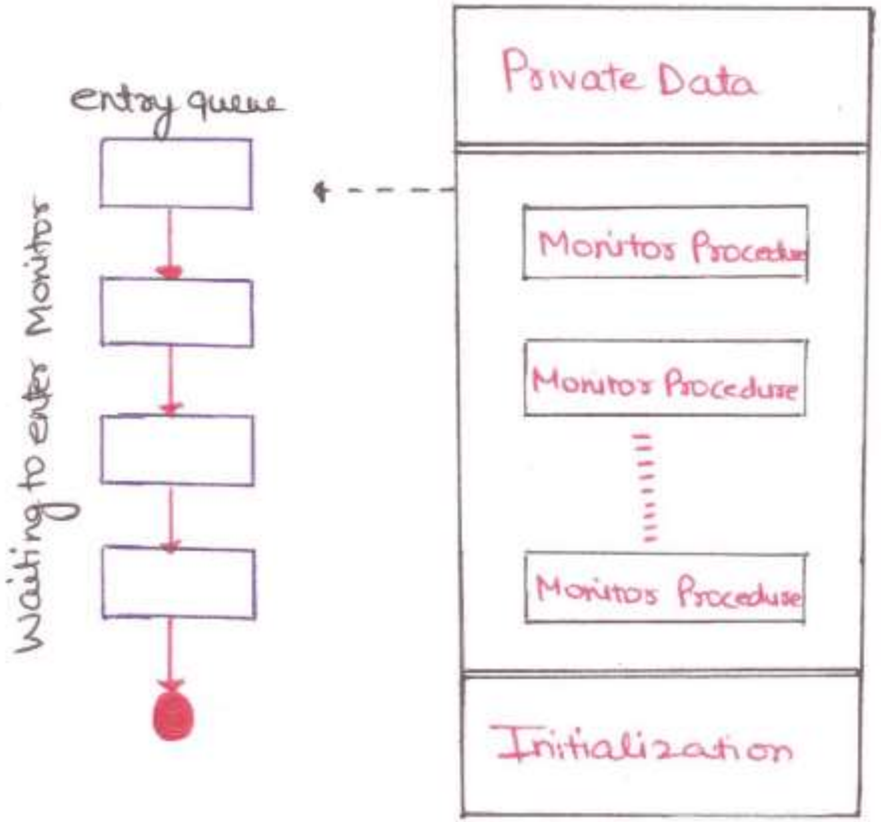
Monitors

- It is the collection of condition variables and procedures combined together in a special kind of module or a package.



Monitors

- The processes running outside the monitor can't access the internal variable of the monitor but can call procedures of the monitor.



Monitors

- Only one process at a time can execute code inside monitors.

- Monitor ensures that
  - Only one process may be active within the monitor at a time

Monitors

- Prgmr does not need to
  - code the synchronization constraint explicitly
- But not powerful enough to model some synchronization schemes
- Need to define additional condition construct

Condition Variables

- **Prgmr can define one or more condition variables**

  condition x, y;

- Only Two operations are allowed on a condition variable:
  - **x.wait()**
  - **x.signal()**

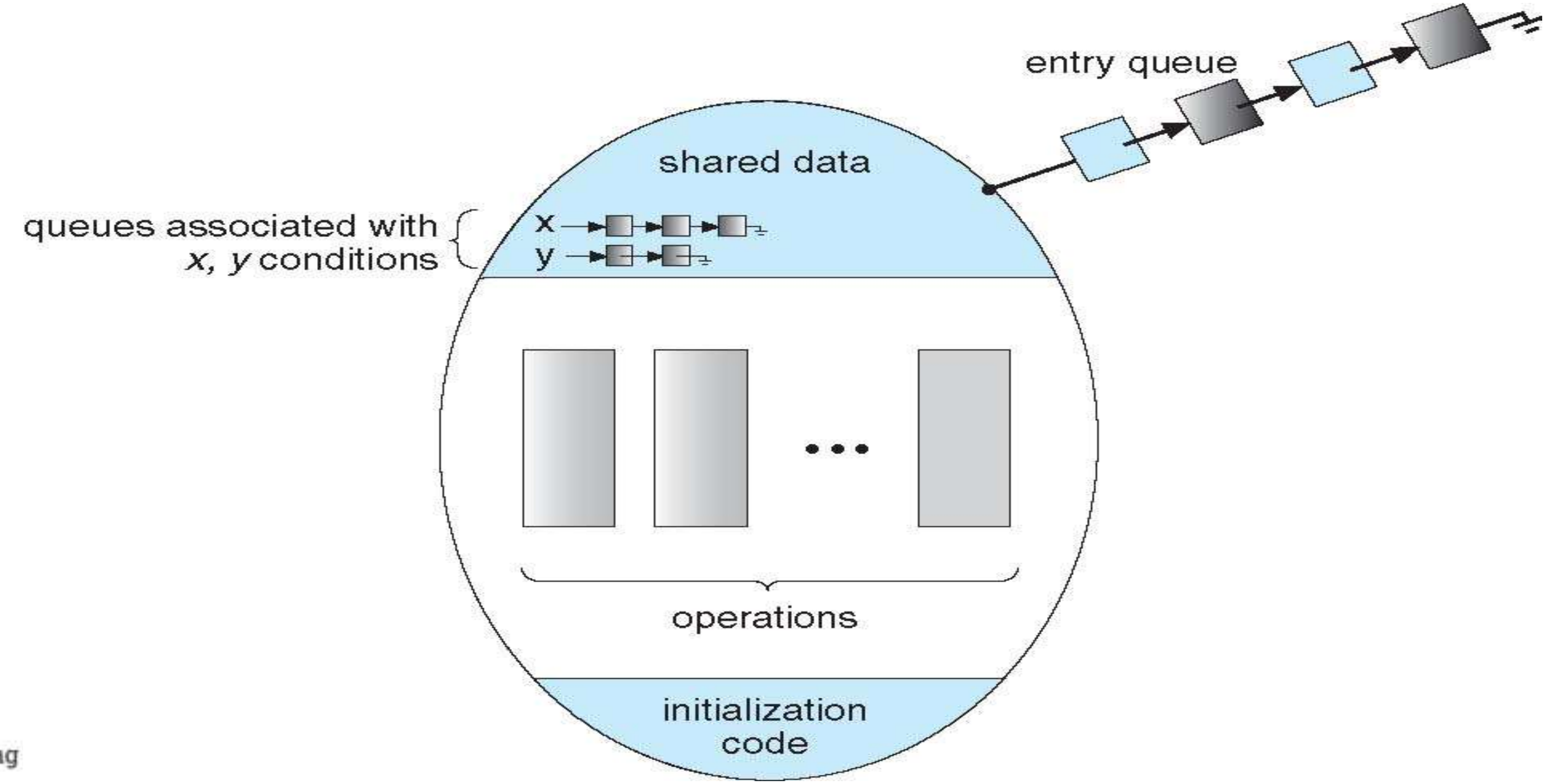
```
Monitor Demo //Name of Monitor
{
variables;
condition variables;

procedure p1 {....}
prodecure p2 {....}


}
```

Syntax of Monitor

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Condition Variables

- **`x.wait()`** –
  - Process performing wait operation on any condition variable are suspended.
    - suspended until **`x.signal()`**
  - The suspended processes are placed in block queue of that condition variable.



Condition Variables

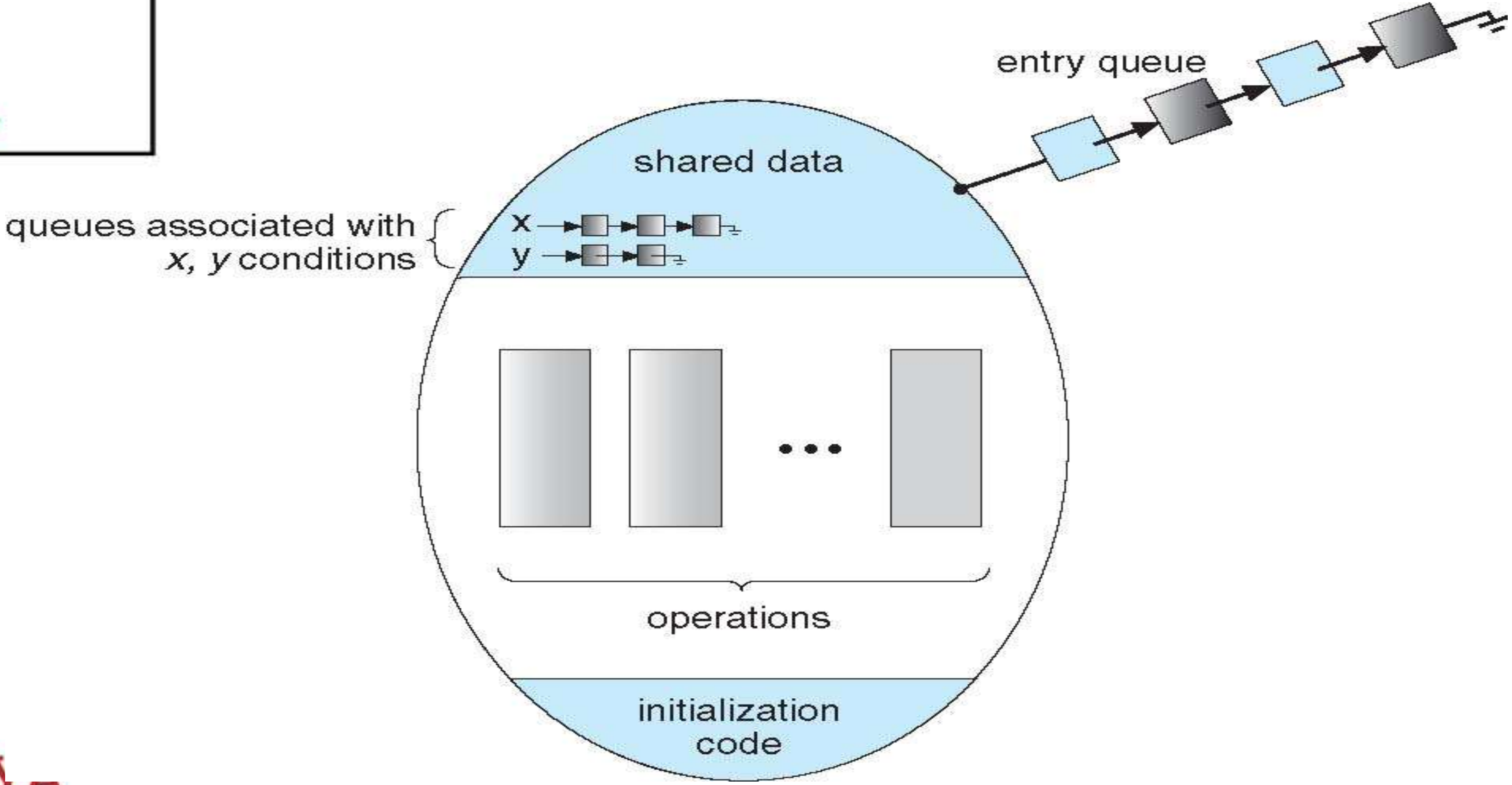
- **`x.signal()`** –
  - When a process performs signal operation on condition variable, one of the blocked processes is given chance.
  - resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`process is suspended`**, then it has no effect on the variable
    - State of x, As if the operation was never executed
    - In contrast to semaphore, state of semaphore always gets affected

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Monitor with Condition Variables

```
Monitor Demo //Name of Monitor
{
variables;
condition variables;

procedure p1 {....}
prodecure p2 {....}


}
```

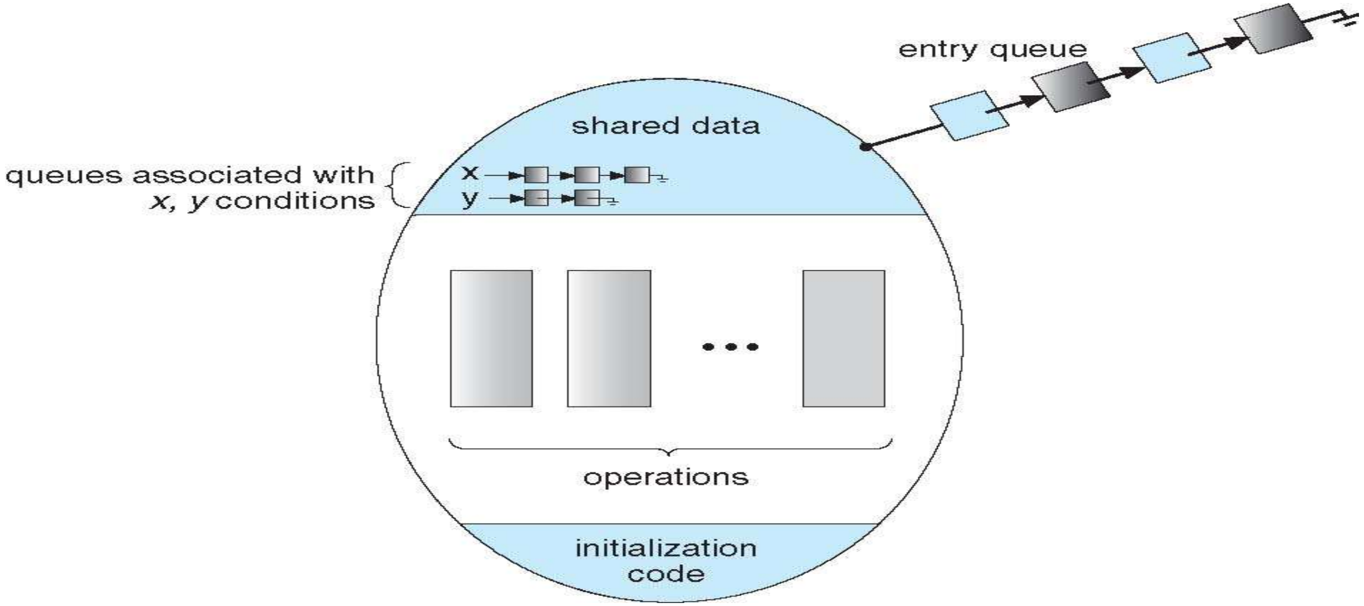
Syntax of Monitor



SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Condition Variables Choices

- If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`,
  - Suspended process Q associated with condition x is invoked
  - what should happen next?



Condition Variables Choices

- Both Q and P cannot execute in parallel.
  - If Q is resumed, then P must wait
  - Otherwise both P and Q will be active simultaneously within the monitor

Condition Variables Choices

- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition

Condition Variables Choices

- Since P was already executing in the monitor,
  - Option 2 seems more reasonable
  - However, If P continues, the logical condition for which Q was waiting may no longer hold by the time Q is resumed
- Both have pros and cons – language implementer can decide
- Monitors implemented in Concurrent Pascal compromise
  - P executing signal immediately leaves the monitor, Q is resumed
- Some languages that do support monitors are Java, C#, Visual Basic, Ada, Mesa and concurrent Euclid.

Monitors

- **Advantages of Monitor:**
  - Make parallel programming easier and
  - less error prone than using techniques such as semaphore.

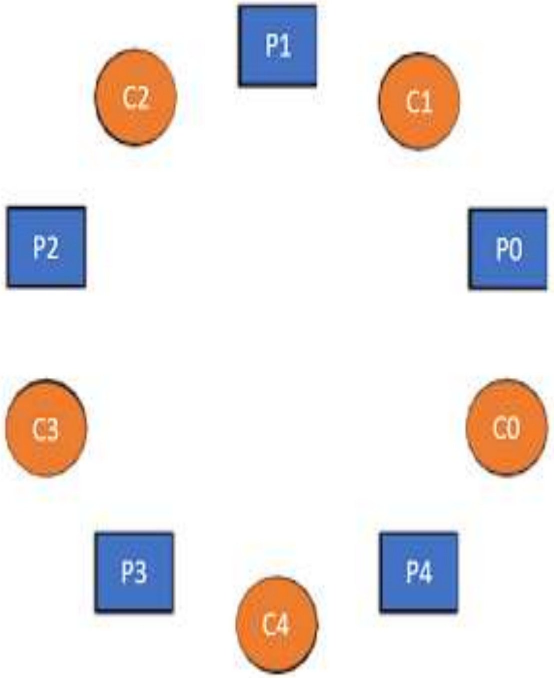
- **Disadvantages of Monitor:**
  - Monitors have to be implemented as part of the programming language .
  - The compiler must generate code for them.
  - This gives the compiler the additional burden of having to know what operating system facilities are available to control access to critical sections in concurrent processes.

Monitor Solution to Dining Philosophers

- Deadlock free solution to Dining Philosophers Problem

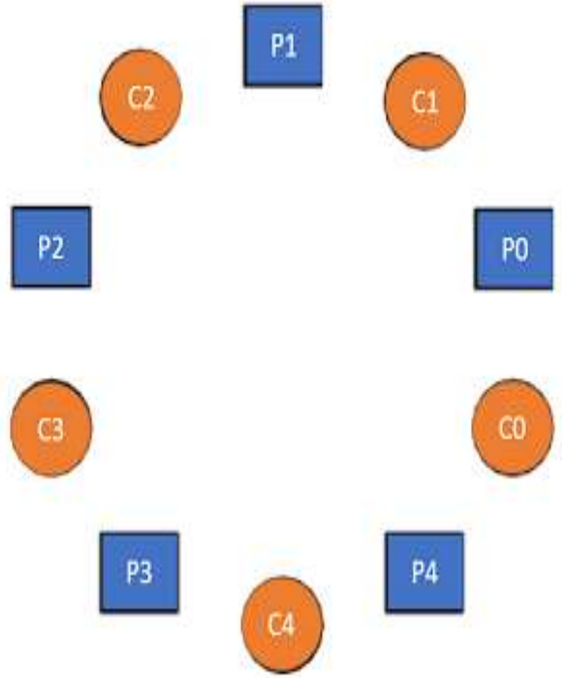
- To distinguish amongst the 3 states in which the philosopher
  - **enum { THINKING; HUNGRY, EATING) state [5] ;**

- Philosopher i can set the variable state [i] = EATING only
  - **if her two neighbors are not eating**
  - **(state [(i +4) % 5] ! = EATING) and (state [(i +1) % 5] != EATING)**



Monitor Solution to Dining Philosophers

- Philosopher i can set the variable state [i] = EATING only
  - **if her two neighbors are not eating**
  - **(state [(i +4) % 5] ! = EATING) and (state [(i +1) % 5] != EATING)**
- i=3,P3
- state(i+1), state P4,Right Neighbour
- state(i+4)%5,7%5,state P2,Left Neighbour



SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Monitor Solution to Dining Philosophers

- Also need to declare
  - condition self[5];
  - in which i^{th} philosopher **can delay herself when she is hungry but is unable to obtain the chopsticks she needs.**
- Distribution of chopsticks is controlled by the monitor DiningPhilosophers

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
enum { THINKING; HUNGRY, EATING) state [5] ;
condition self [5];

void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
         self[i].wait;
}
```

Pickup chopsticks

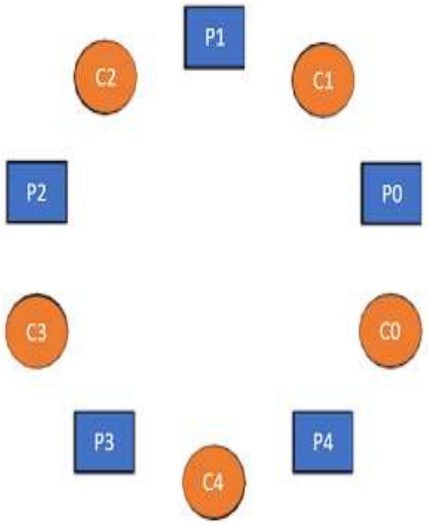
- If unable to eat, waits to be signaled
- Philosopher can delay herself when she is hungry but is unable to obtain the chopsticks she needs.

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{

    void putdown (int i) {
        state[i] = THINKING;
                // test left and right neighbors
         test((i + 4) % 5);
         test((i + 1) % 5);
}
```

Put down chopsticks

if right neighbor R=(i+1)%5 is hungry and both of R’s neighbors are not eating, set R’s state to eating and wake up neighbour R by signaling



SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Monitor Solution to Dining Philosophers

```
void test (int i) {
        if ((state[(i + 4) % 5] != EATING) && (state[i] == HUNGRY) &&(state[(i + 1) % 5] !=
  EATING) ) {
             state[i] = EATING ;
      self[i].signal () ;
        }
    }

       initialization_code() {
       for (int i = 0; i < 5; i++)
       state[i] = THINKING;
     }
}
```

- If her two neighbors are not eating and she is hungry
  - I.E. if my left and right neighbors are not eating
- Set her state as eating

- signal() has no effect during Pickup(),
- but is important to wake up waiting hungry philosophers during Putdown()

Solution to Dining Philosophers

- Each philosopher *i* invokes the operations `pickup()` and `putdown()` in the following sequence:

```
DiningPhilosophers.pickup(i);

      EAT

DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

Solution to Dining Philosophers

- Each philosopher, before starting to eat, must invoke the operation pickup().
- This act may result in the suspension of the philosopher process.
- After the successful completion of the operation, the philosopher may eat.
- After eating, philosopher invokes putdown() and start to think

Solution to Dining Philosophers

- Execution of Pickup(), Putdown() and test() are all mutually exclusive, i.e. only one at a time can be executing
- No 2 neighbors are eating simultaneously
- So No deadlock, but starvation is possible