THE OS AS A GATEKEEPER

Ch5 - Process Synchronization

Operating Systems

Objective

- To discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space so that data is consistently maintained.

Concept of concurrency

- Existence of several simultaneous processes.

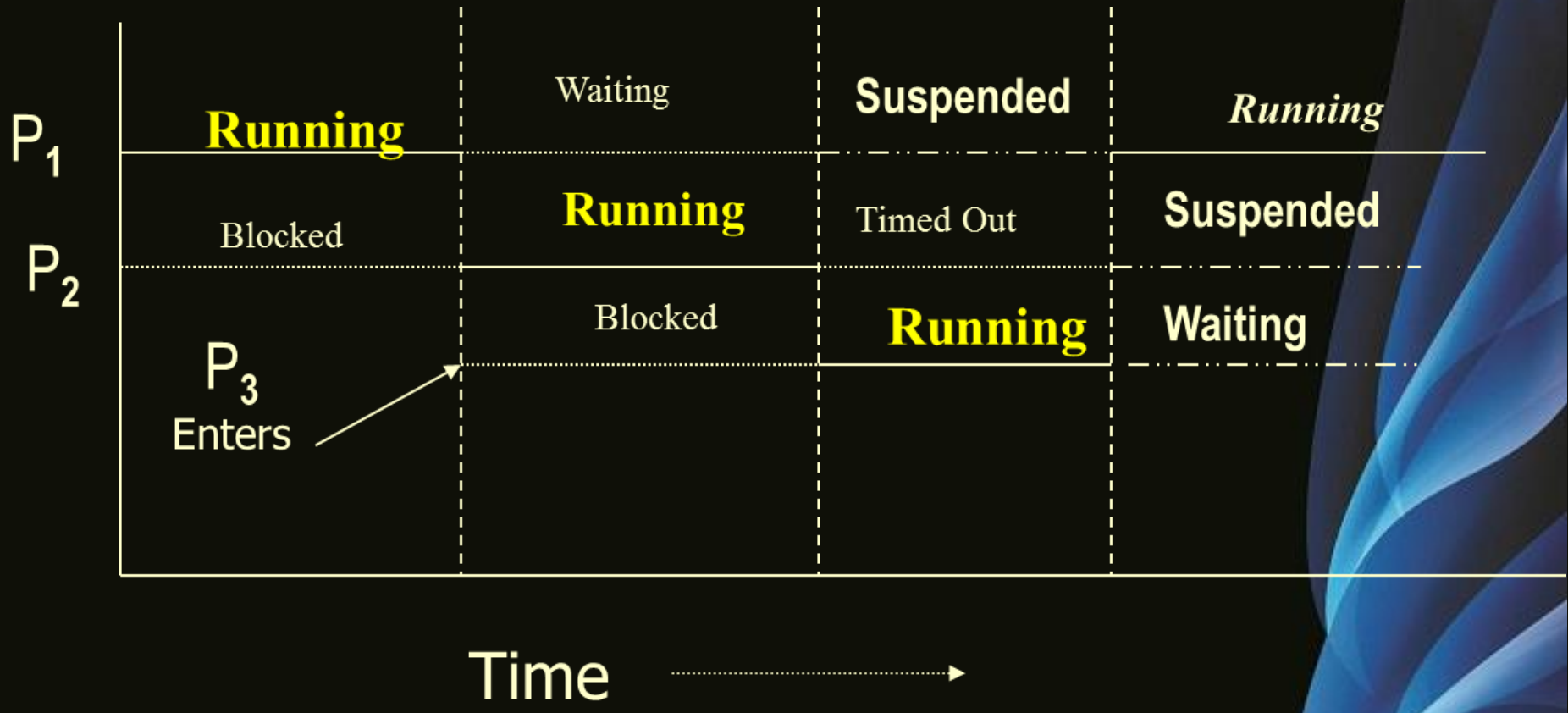
May be observed in situations like:

- Multi-user OS where different users execute programs concurrently
- Single-user OS where input/output devices go simultaneously with program execution
- Some processors whose activities are concurrent, example, fetch next instruction is done simultaneously with the execution of an instruction.

Definition: Concurrent Processes

- A set of processes P_1 , P_2……, P_i , ….. P_n are said to be **CONCURRENT** if before any one among these processes gets terminated (normally or enforced) some other process in the set *starts* running.

The Typical Timing Sequence



Review on Bounded Buffer

PRODUCER

```
while (true) {

    /*  produce an item and put in
                nextProduced  */

     while (count == BUFFER_SIZE)
                do nothing;

      buffer [in] = nextProduced;

      in = (in + 1) % BUFFER_SIZE;

      count++;
}
```

CONSUMER

```
while (true)  {

     while (count == 0)
                do nothing

     nextConsumed =  buffer [out];

     out = (out + 1) % BUFFER_SIZE;

      count--;

    /*  consume the item in
                nextConsumed  */
}
```

Atomic Operation

- The statements count ++ and count -- must be performed atomically.
- Atomic operation means an operation that completes in its entirety without interruption.
- May not function correctly when executed concurrently.

Atomic Operation

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language may get interleaved
- Interleaving depends upon how the producer and consumer processes are scheduled.

Atomic Operation

- count++ could be implemented as:

```
MOV AX, COUNT      register1 = count
INC AX             register1 = register1 + 1
MOV COUNT, AX      count = register1
```

- count-- could be implemented as:

```
MOV BX, COUNT      register2 = count
DEC BX             register2 = register2 - 1
MOV COUNT, BX      count = register2
```

Now consider this interleaving…

- *Consider this execution interleaving with "count = 5" initially:*
  - S0: producer execute register1 = count {register1 = 5}
  - S1: producer execute register1 = register1 + 1 {register1 = 6}
  - S2: consumer execute register2 = count {register2 = 5}
  - S3: consumer execute register2 = register2 - 1 {register2 = 4}
  - S4: producer execute count = register1 {count = 6 }
  - S5: consumer execute count = register2 {count = 4}

Race Condition

- Situation where several processes access and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

- To prevent race conditions, concurrent processes must be **synchronized**.

Precedence Constraints Problem

- Determine the values of X when statements of A and B are interleaved with the ff. sequence of execution:

Shared var **X:** integer {initialized to 10}

Process A	Process B
Int M;	Integer W;
A1: M = X * 2;	B1: W = X + 3
A2: M = M + 3;	B2: W = W * 2
A3: X = M;	B3: X = W

a. A1 A2 A3 B1 B2 B3

b. A1 B1 A2 B2 A3 B3

c. A1 B1 B2 B3 A2 A3

d. B1 B2 B3 A1 A2 A3

Solution A:

Shared var **X:** integer {initialized to 10}

Process A	Process B
Int M;	Integer W;
A1: M = X * 2;	B1: W = X + 3
A2: M = M + 3;	B2: W = W * 2
A3: X = M;	B3: X = W

a. A1 A2 A3 B1 B2 B3

A1: M = X * 2 => 10 * 2 = 20

A2: M = M + 3 => 20 + 3 = 23

A3: X = M => X = 23

B1: W = X + 3 => 23 + 3 = 26

B2: W = W * 2 => 26 * 2 = 52

B3: X = W => **X = 52**

Solution B:

Shared var **X:** integer {initialized to 10}

Process A	Process B
Int M;	Integer W;
A1: M = X * 2;	B1: W = X + 3
A2: M = M + 3;	B2: W = W * 2
A3: X = M;	B3: X = W

b. A1 B1 A2 B2 B3 A3

A1: M = X * 2 => 10 * 2 = 20

B1: W = X + 3 => 23 + 3 = 26

A2: M = M + 3 => 20 + 3 = 23

B2: W = W * 2 => 26 * 2 = 52

B3: X = W => X = 52

A3: X = M => **X = 23**

The Critical Section (CS) Problem

- Consider *n* processes all competing to use some shared data.
- Each process has a code segment, called *critical section* in which shared data is accessed.
- Ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical condition.

The Critical Section (CS) Problem

- Execution of critical sections by the processes is mutually exclusive in time.

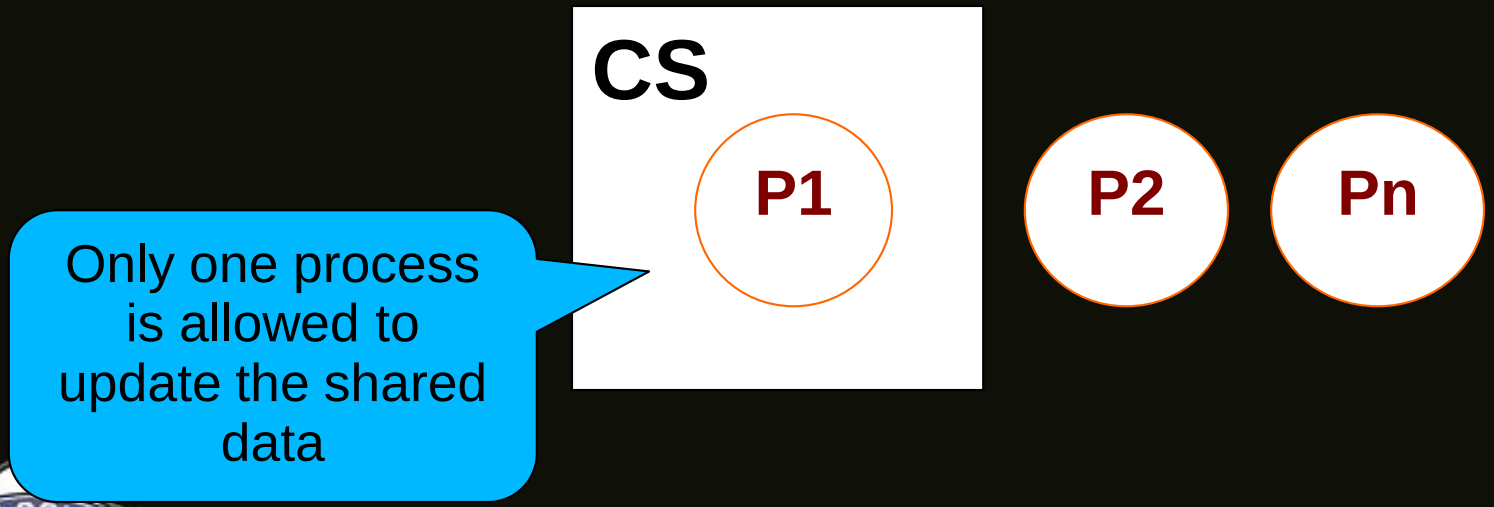
- Design protocol:
  - Each process must request permission to enter its CS (entry section)
  - The CS is followed by an exit section
  - The remaining code is in the remainder section.

Solution to Critical-Section Problem

1. Mutual Exclusion
2. Progress
3. Bounded Waiting

Solution to Critical-Section Problem

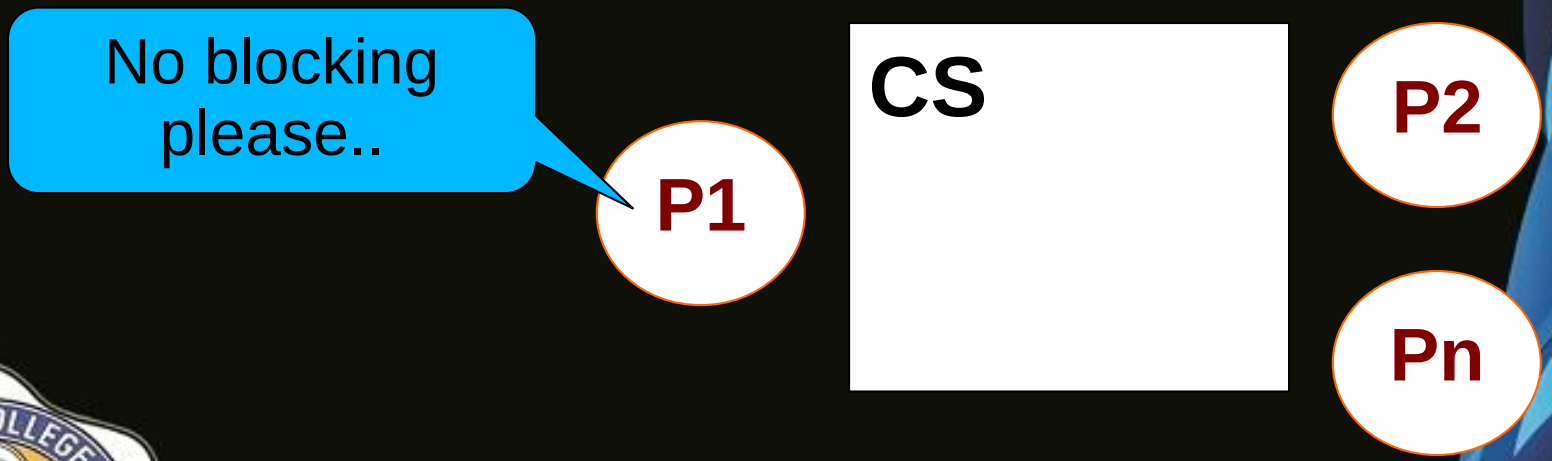
1. **Mutual Exclusion**
   - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections



Solution to Critical-Section Problem

2. **Progress**

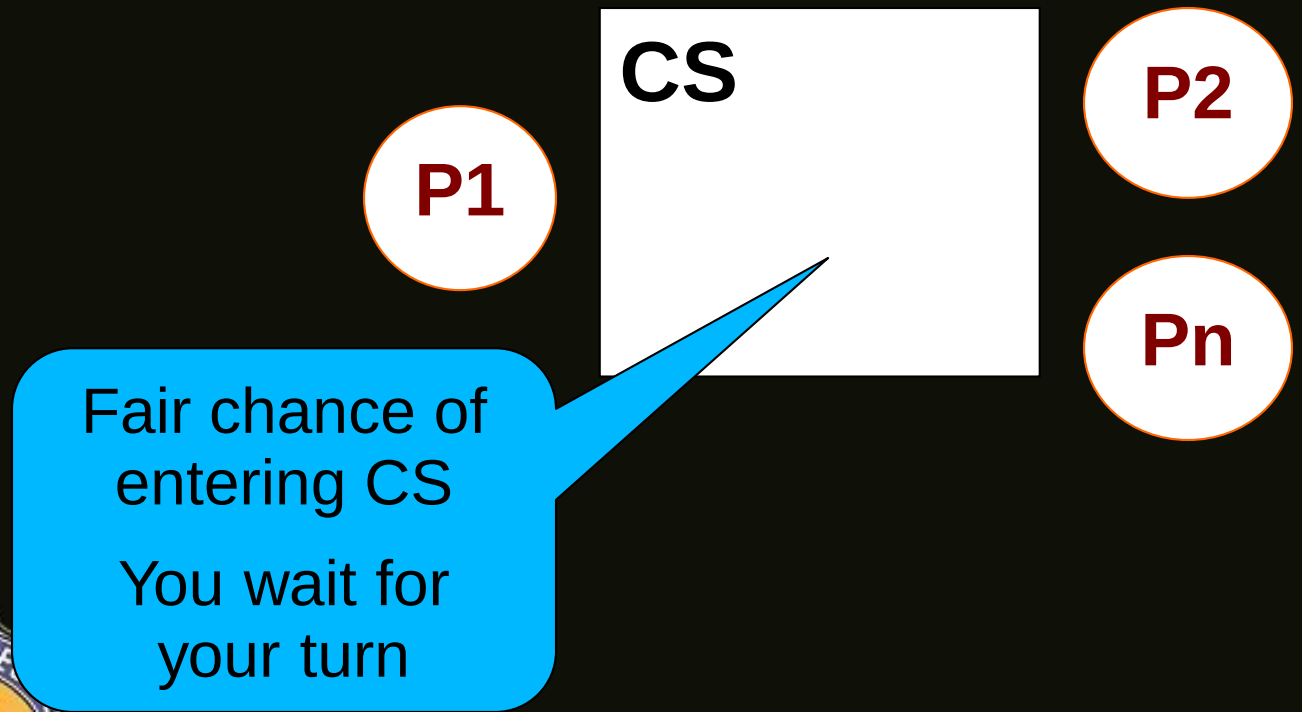
   - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.



Solution to Critical-Section Problem

3. **Bounded Waiting**

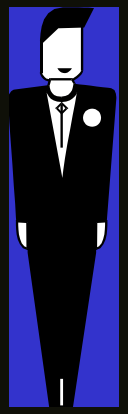
   - No process should wait arbitrarily long to enter its CS.



Note:

- No assumptions are made about relative process speeds or number of CPU's
- A process in one CS should not block others in entering a different CS.

Sample Scenario – Milk Issue



Time	Person A	Person B
3:00	Look in fridge, *Out of milk*	----------
3:05	Leave for store	----------
3:10	Arrive at store	Look in fridge, *Out of Milk*
3:15	Buy Milk	Leave for store
3:20	Leave store	Arrive at store
3:25	Arrive home, put milk away	Buy milk
3:30	----------	Leave Store
3:35	----------	Arrive home, oops

Observation: Milk issue

- Someone gets milk, but not EVERYONE (too much milk!)
- This shows that when cooperating processes are not synchronized, they may face unexpected “timing errors”

- ***Mutual Exclusion*** - ensures that only one process (or person) is doing a thing at one time, thus avoiding data inconsistency. All others should be prevented from modifying shared data until the current process finishes.

  - E.g. only one person buys milk at a time.

Solution 1 – Attempt to computerize Milk Buying

```
PROCESS A AND B

If (NOmilk)
{
    if (NOnote)
    {
        Leave note;
        Buy milk;
        Remove note;
    }
}
```

This solution works for some people because the first three lines are performed **ATOMICALLY** but does not work otherwise.

What happens when the 2 processes execute concurrently?

Observation on Solution 1

```
PROCESS A AND B

If (NOmilk)
{
    if (NOnote)
    {
        Leave note;
        Buy milk;
        Remove note;
    }
}
```

What happens when the 2 processes execute concurrently?

Both processes A and B will:

- Leave note and
- Buy milk

Therefore,

Mutual exclusion is NOT met.

Solution 2 – Attempt to use two notes

PROCESS A

```
Leave (NOTEa);

If (NO_NOTEb)
{
    if (NOmilk) Buy Milk;
}

Remove (NOTEa);
```

PROCESS B

```
Leave (NOTEb);

If (NO_NOTEa)
{
    if (NOmilk) Buy Milk;
}

Remove (NOTEb);
```

WHAT CAN YOU SAY ABOUT THIS SOLUTION?

Observation on Solution 2

PROCESS A

Leave (**NOTEa**);

If (**NO_NOTEb**)

if (**NOmilk**) Buy Milk;

Remove (**NOTEa**);

PROCESS B

Leave (**NOTEb**);

If (**NO_NOTEa**)

if (**NOmilk**) Buy Milk;

Remove (**NOTEb**);

If both processes execute at the same time, both will:

- Leave the note, and
- Remove the note at the same time

Thus, **the Progress criterion is NOT satisfied**…

Solution 3:

PROCESS A

Leave (**NOTEa**);

If (**NOnoteB**)
{
    if (**NOmilk**) Buy Milk;
}

Remove (**NOTEa**);

PROCESS B

Leave (**NOTEb**)

While (**NOTEa**) do
    nothing;

if (**NOmilk**) Buy Milk;

Remove (**NOTEb**);

In case of tie, who will buy first?

Observation on Solution 3

- Process B will always be the first to buy the milk, in case of a tie.
- Process B consumes CPU cycles while waiting.
- More complicated if extended to many processes.

Thus, **Bounded Waiting criterion is violated.**

Algorithm Template

```
do {
        ENTRY_SECTION

        CRITICAL_SECTION

        EXIT_SECTION

        Remainder_Section

} while (TRUE)
```

ENTRY SECTION – Each process must request permission to enter its CS

EXIT SECTION – Process leaves the CS thereby informing other processes that the CS is free for some other process to enter

Two Process Solution

Algo 1: Strict Alternation Shared var ***Turn*** <either a or b>

PROCESS A	PROCESS B
repeat	repeat
While (turn==B) do nothing;	While (turn==A) do nothing;
<critical section A>	<critical section B>
Turn = B;	Turn = A;
until false	until false

Which of the following requirement is violated?

Mutual Exclusion Progress Bounded Waiting

Two Process Solution

Algo 1: Strict Alternation Shared var ***Turn*** <either a or b>

PROCESS A	PROCESS B
repeat	repeat
While (turn==B) do nothing;	While (turn==A) do nothing;
<critical section A>	<critical section B>
Turn = B;	Turn = A;
until false	until false

Which of the following requirement is violated?

Progress

Algorithm 2

Shared Var ***pAinside***, ***pBinside***; FALSE

PROCESS A

```
While (TRUE)  {
    While (pBinside) do
                    nothing;

    pAinside = True;

    <critical section A>

    pAinside = False;
}
```

PROCESS B

```
While (TRUE)  {
    While (pAinside) do
                    nothing;

    pBinside = True;

    <critical section B>

    pBinside = False;
}
```

Which of the following requirement is violated?

Mutual Exclusion Progress Bounded Waiting

Algorithm 2

Shared Var ***pAinside***, ***pBinside***; FALSE

PROCESS A

```
While (TRUE)  {
    While (pBinside) do
                        nothing;

    pAinside = True;

    <critical section A>

    pAinside = False;
}
```

PROCESS B

```
While (TRUE)  {
    While (pAinside) do
                        nothing;

    pBinside = True;

    <critical section B>

    pBinside = False;
}
```

Which of the following requirement is violated?

Mutual Exclusion

Algorithm 3

Shared Var ***pAtrying***, ***pBtrying*** : FALSE

PROCESS A

```
While (TRUE)  {
    pAtrying = True;
    While (pBtrying) do
                    nothing;
    <critical section A>
    pAtrying = False;
}
```

PROCESS B

```
While (TRUE)  {
    pBtrying = True;
    While (pAtrying) do
                    nothing;
    <critical section B>
    pBtrying = False;
}
```

Which of the following requirement is violated?

Mutual Exclusion Progress Bounded Waiting

Algorithm 3

Shared Var ***pAtrying***, ***pBtrying*** : FALSE

PROCESS A

```
While (TRUE)  {
    pAtrying = True;

    While (pBtrying) do
                        nothing;

    <critical section A>

    pAtrying = False;
}
```

PROCESS B

```
While (TRUE)  {
    pBtrying = True;

    While (pAtrying) do
                        nothing;

    <critical section B>

    pBtrying = False;
}
```

Which of the following requirement is violated?

Progress

Dekker's Algorithm - Elegant solution to mutual exclusion problem

Shared var pAtrying pBtrying = False

PROCESS A

```
While (TRUE)  {
    pAtrying = True;
    While (pBtrying)
        if (turn==B)  {
            pAtrying = False;
            while (turn==B) do nothing;
            pAtrying = True
        }
    <critical section>
    turn==B;
    pAtrying = False;
}
```

PROCESS B

```
While (TRUE)  {
    pBtrying = True;
    While (pAtrying)
        if (turn==A) {
            pBtrying = False;
            while (turn==A) do nothing;
            pBtrying = True
        }
    <critical section>
    turn==A;
    pBtrying = False;
}
```

Peterson's Algorithm: Much simpler than Dekker's Algorithm

Shared var pAtrying pBtrying = False, Turn =a or b

PROCESS A

```
While (TRUE)
{
    pAtrying = True;
    turn= B;
    While (pBtrying && turn ==B)
            do nothing;

    <critical section A>

    pAtrying = False;
}
```

PROCESS B

```
While (TRUE)
{
    pBtrying = True;
    turn= A;
    While (pAtrying && turn ==A)
            do nothing;

    <critical section A>

    pBtrying = False;
}
```

Note

- Both **Dekker's and Peterson's algorithms** are correct but they only work for 2 processes, similar to the last solution of the "too-much-milk" problem.

Multiple Process Synchronization Solution

- Hardware Support – special instructions
  - Many CPU's today provide hardware instructions to read, modify and store a word **atomically**. Most common instructions with this capability are:
    - **TAS –** (Test and Set) – Motorolla 68k
    - **CAS** – (Compare and Swap) – IBM 370 and Motorola 68k
    - **XCHG** – (exchange) –x86

Mutual Exclusion with Test-&-Set (TAS)

```
boolean TestAndSet
(boolean &target)

{
    boolean rv = target;
    target = true;
    return rv;
}
```

Shared data:

boolean lock = false;

Process *Pi*

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

    <critical section>

    lock = false;
    remainder section
}
```

Mutual Exclusion with SWAP and XCHG

Atomically swap two variables.

```
void Swap(boolean &a,
          boolean &b)

{
    boolean temp = a;
    a = b;
    b = temp;
}
```

Shared data (initialized to **false**):

```
    boolean lock;
    boolean waiting[n];
```

Process *Pi*

```
do {
key = true;
    while (key == true)
        Swap(lock,key);
    critical section
    lock = false;
    remainder section
}
```

Multiple Process Synchronization Solution

- The basic idea is to be able to read the contents of a variable (memory location) and set it to something else *all in one execution cycle* hence not interruptible.
- The use of these special instructions makes the programming task easier and improves efficiency.

Semaphores

Semaphores

- Common synchronization construct in OS kernels
- Like a traffic light : STOP and GO
- Similar to mutex but more general

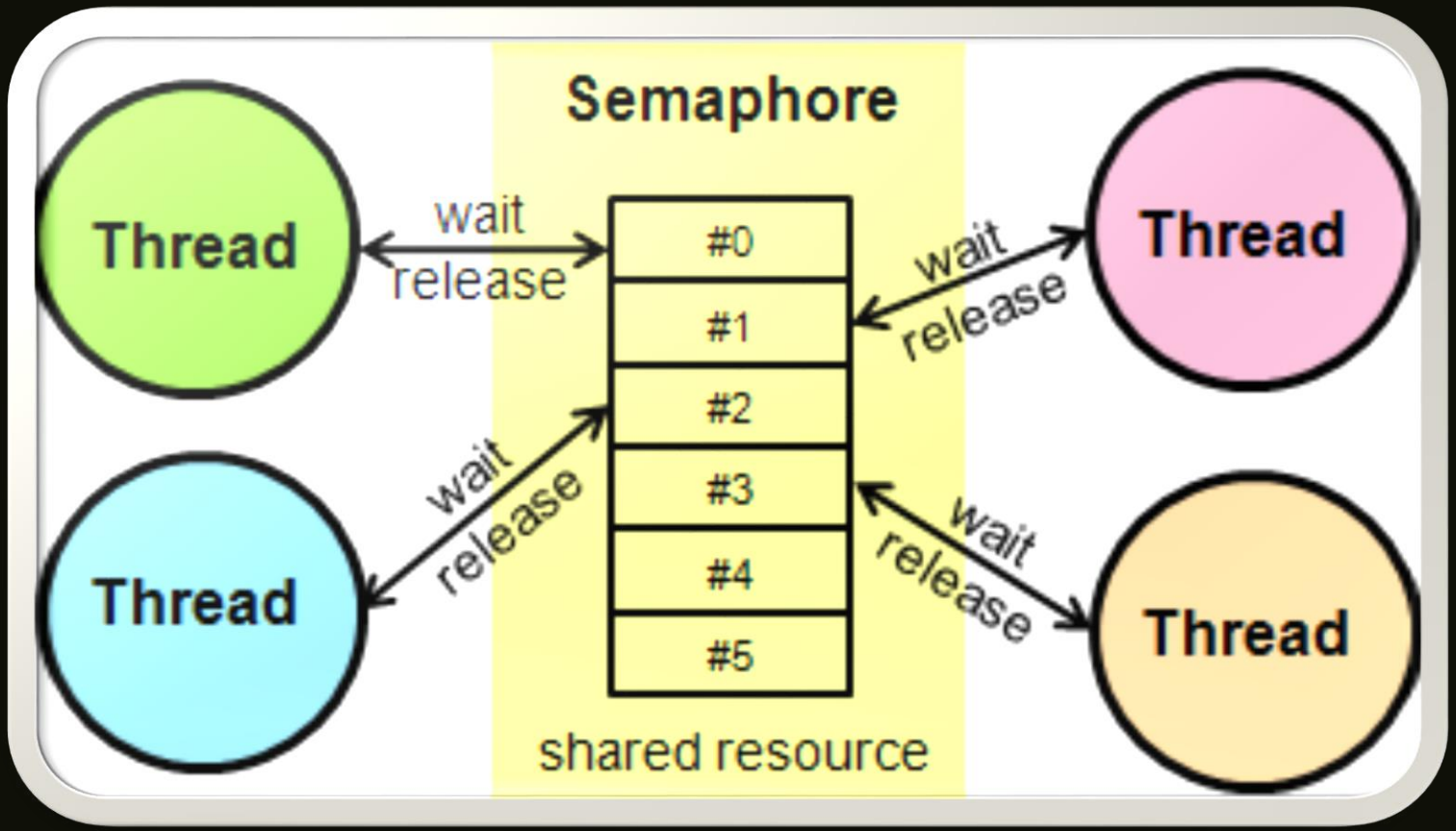
Mutual Exclusion (mutex)

- Either it allows the thread to obtain the lock and proceed with the CS or be blocked until the mutex becomes free
- Acquire and Release
- CS – changing variables, updating a table, writing a file, etc.

Semaphores

- are used to manage and protect access to shared resources.
- Semaphores are very similar to Mutexes. Whereas a Mutex permits just one thread to access a shared resource at a time, a semaphore can be used to permit a fixed number of threads to access a pool of shared resources.
- Using semaphores, access to a group of identical peripherals can be managed

Semaphores


Semaphore
Thread
wait
release
#0
#1
#2
#3
#4
#5
Thread
wait
release
Thread
wait
release
Thread
wait
release
shared resource

Semaphores

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard atomic operations modify S: wait() and signal()
  - Originally called P() and V()

Two Atomic Operations

- *Wait (S)*
  - While s ≤ 0 do no_op;
  - S--

PROBEREN:

"to test"

- *Signal (S)*
  - S++

VERHOGEN:

release

"to increment"

A Semaphore...

- Semaphor S == integer value (count-based sync)
- on init (initialize S)

  Assigned a max value (+ integer) maximum count
- As thread arrives

  on try (wait)

  if non-zero -> decrement and proceed (counting S)
- if initialized with 1

  semaphor == mutex (binary semaphore)

  on exit (signal)

  increment

Two Atomic Operations

Example: S=5

```
wait (S) {
        while (S <=0)
        ; // busy wait
        S--
}
signal (S) {
        S++;
}
```

Resources	Process
S1	P1
S2	P2
S3	P3
S4	P4
S5	P5

Solution to too much milk using semaphore

```
Semaphore OkToBuyMilk = 1;

Wait (OkToBuyMilk);
If (NoMilk)
    {
        Buy Milk;
    }
Signal (OkToBuyMilk)
```

Types of semaphores

- **Binary Semaphors**
  - semaphore with an integer value of 0 or 1
  - behaves similarly with mutex
- **Counting Semaphors**
  - semaphore with an integer value ranging between 0 and an arbitrarily large number.
  - can be used to control access to a given resource consisting of a finite number of instances.
  - also known as general semaphore.

Semaphore Usage

- Can solve various synchronization problems

Sample Problem: Consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2.

P_1 and P_2 that require S_1 to happen before S_2

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Semaphore Implementation

- Must guarantee that no two processes can execute **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - 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

- 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
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Example #1:

- There are 6 processes waiting in the ready queue (P1,P2…P6)
- Let us assume that
  - P5 can only be executed if P1 and P2 have already completed their tasks
  - P3 will only be executed if P6 is finished

Synchronize the execution of the 2 CPUs:

CPU1:	**CPU2:**
P1;	P4;
P2;	P5;
P3;	P6;

Solution:

- P5 can only be executed if P1 and P2 have already completed their tasks.
- P3 will only be executed if P6 is finished

Synchronize the execution of the 2 CPUs:

CPU1:	**CPU2:**
P1;	P4;
	wait (s1);
P2;	P5;
signal (s1);	
wait (s2);	
P3;	P6;
	signal (s2);

Prerequites of process P5

Has prerequites before execution

Has prerequites before execution

Prerequiste of Process P3

Possible uses of semaphores

- Mutual Exclusion
  - Initializes the semaphore to **one (1)**

    - Example:

```
Wait (Mutex);

   < Critical section; >

Signal (Mutex);
```

Possible uses of semaphores

- Synchronization of cooperating processes (signalling)
  - initializes the semaphore to **zero**.

- Example

P1:	P2
…	…
…	…
A	wait (flag)
Signal (flag)	B

Possible uses of semaphores

- Managing multiple instances of a resource
  - Initializes the semaphore to the **number of instances**.

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
- 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);`

Deadlock and Starvation

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

Classical Problems of Synchronization

Classical Synchronization Problem:

- Bounded - Buffer
- Readers - Writers
- Dining Philosophers
- Sleeping - Barbers

Bounded-Buffer Problem:

- Two processes share a common, fixed size buffer. One of them, the **producer** puts information into the buffer, and the other, the **consumer,** takes out information from the buffer.
- When the producer wants to put a new item in the buffer but it is already full then allow the producer to sleep to be awakened when the consumer has removed one or more items.
- Similarly to the consumer, it goes to sleep if the buffer is empty until the producer puts something in the buffer.

shared data

semaphore full = 0, empty = n, mutex = 1;

Producer:

```
do {
    …
    produce an item in nextp
    …
    wait ( empty );
    wait ( mutex );
     …
    add nextp to buffer
     …
    signal ( mutex );
    signal ( full );
 } while (TRUE);
```

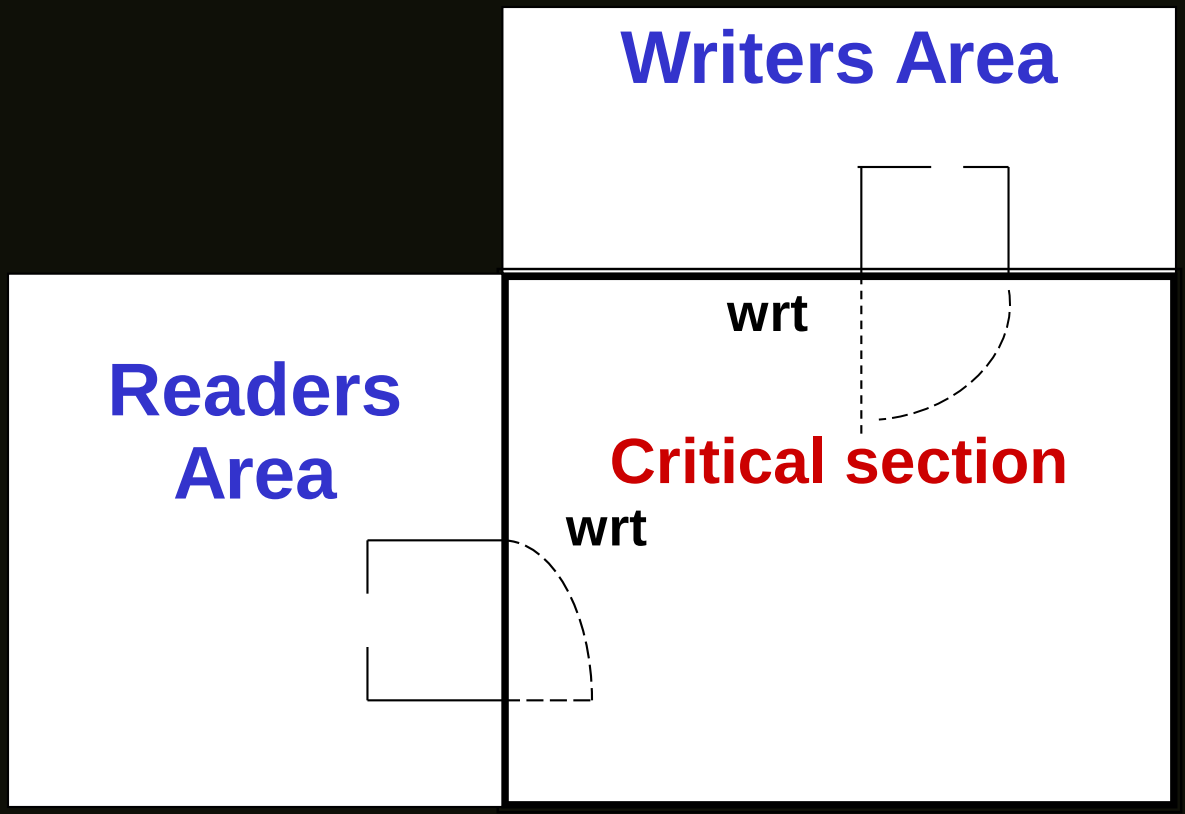
Consumer:

```
do {
    wait ( full )
    wait ( mutex );
     …
    remove item from buffer to
nextc
     …
    signal ( mutex );
    signal ( empty );
     …
    consume item in nextc
     …
  } while (TRUE);
```

Readers-Writers Problem

- This problem models access to a database with many competing processes wishing to read and write into it (imagine an airline reservation system).
- It is possible to have many processes reading the database at the same time (for query purposes). But if one process is writing (modifying), no other process shall access to the database, not even readers. With this condition in mind, how will you program the reader and writers?

Illustration: Readers and Writers



Algorithm #1:

Readers have higher priority than writers

If a writer appears while several readers are in the database, the writer must wait. If new readers keep appearing so that there is always at least one reader in the database, the writer must keep waiting until no more readers are interested in the database.

Solution to Readers/Writers: Algorithm #1

shared data **semaphore mutex = 1, wrt = 1;**

int readcount = 0;

Writers:

```
do {
    wait(wrt);
     ...
    writing is performed
     ...
    signal(wrt);

} while (TRUE)
```

Readers:

```
do {
      wait ( mutex );
      readcount + +;
      if ( readcount == 1 )     wait ( wrt );
      signal ( mutex );
       …
      reading is performed
       …
      wait ( mutex );
      readcount - -;
      if (readcount == 0 )      signal ( wrt );
      signal ( mutex ):
} while (TRUE)
```

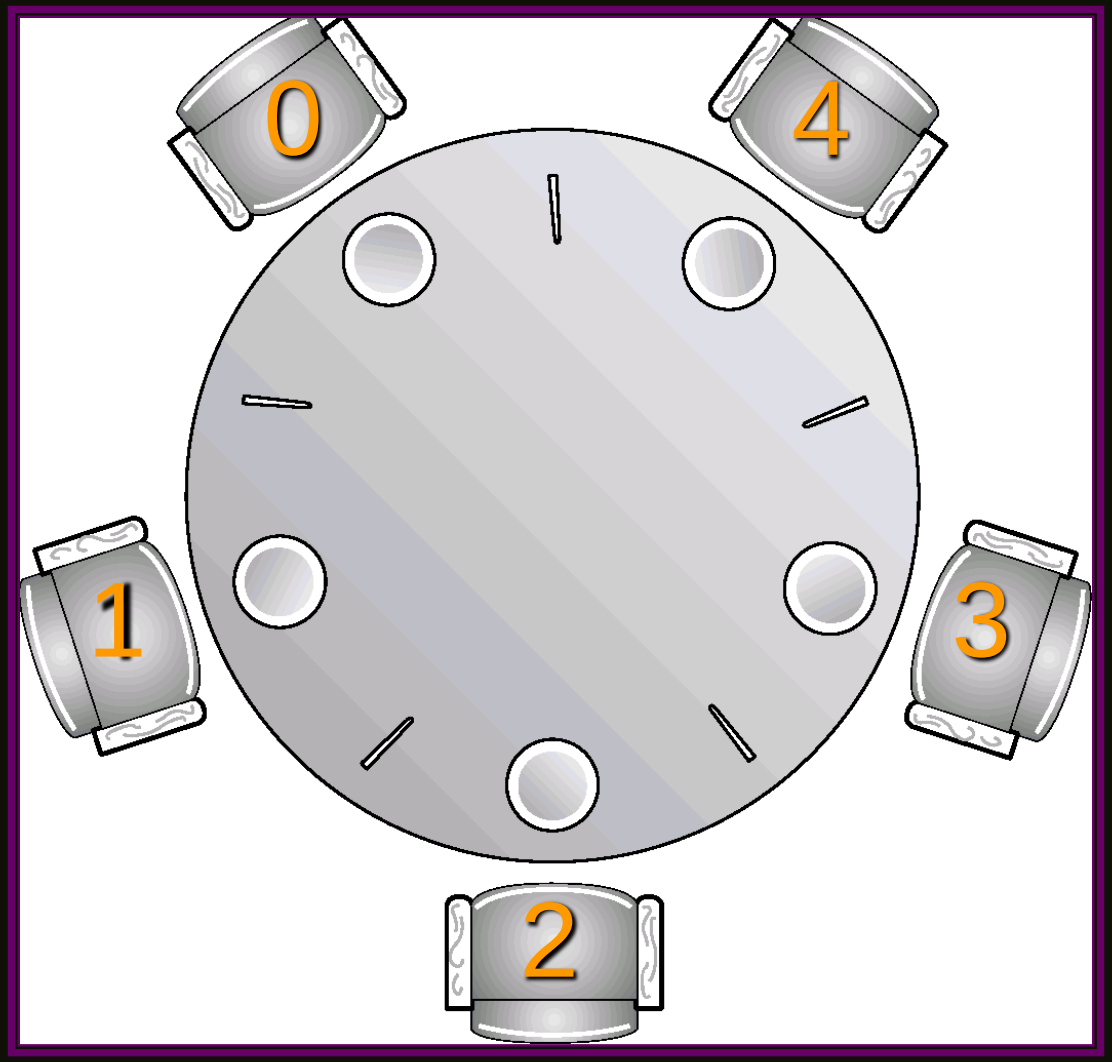
Dining-Philosophers Problem

- Five philosophers spend their lives thinking and eating.
- Philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher.
- At the center of the table, there is a bowl of rice, and the table is laid with five single chopsticks.

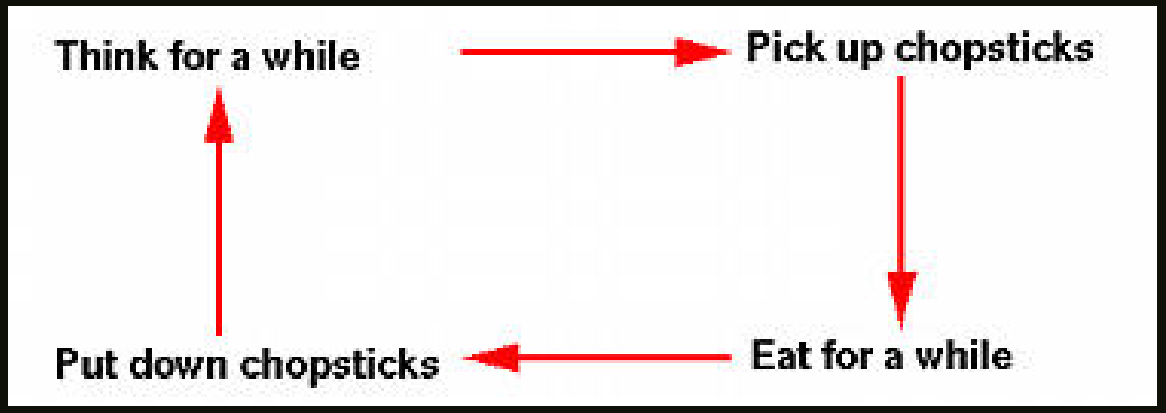
Dining-Philosophers cont...

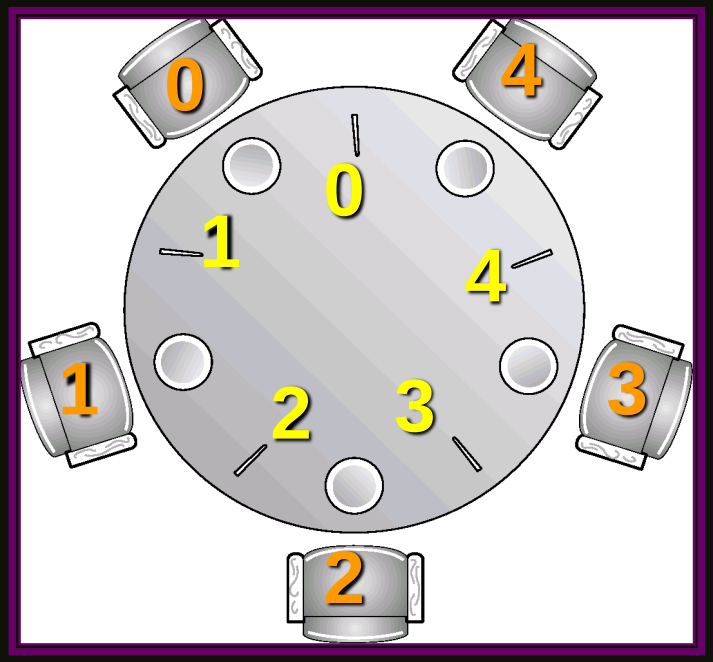
- When a philosopher thinks, he does not interact with his colleagues.
- When a philosopher gets hungry, he tries to pick up two chopsticks that are closest to him.
- Philosopher could pick up only one chopstick at a time.
- Obviously, he cannot pick up a chopstick that is already in the hand of his neighbor.
- When a hungry philosopher has his both chopsticks, he eats without releasing them.
- When he is finished, he puts down the chopsticks and starts thinking again.

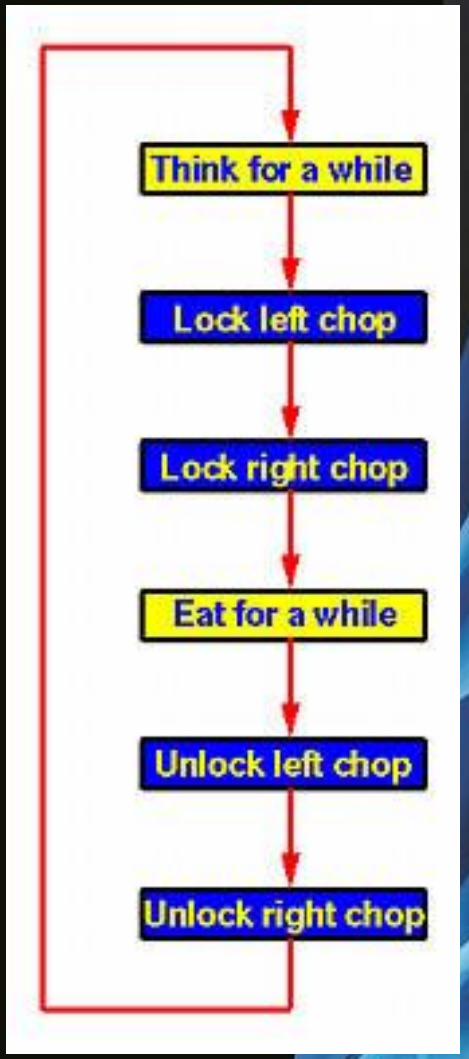
Dining-Philosophers diagram



Dining-Philosophers analysis

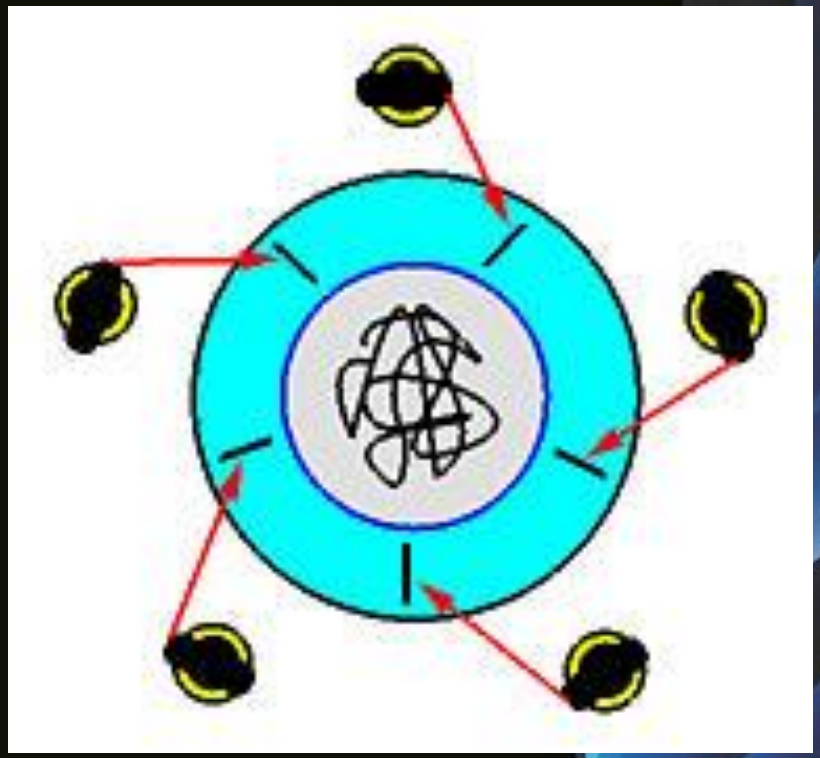






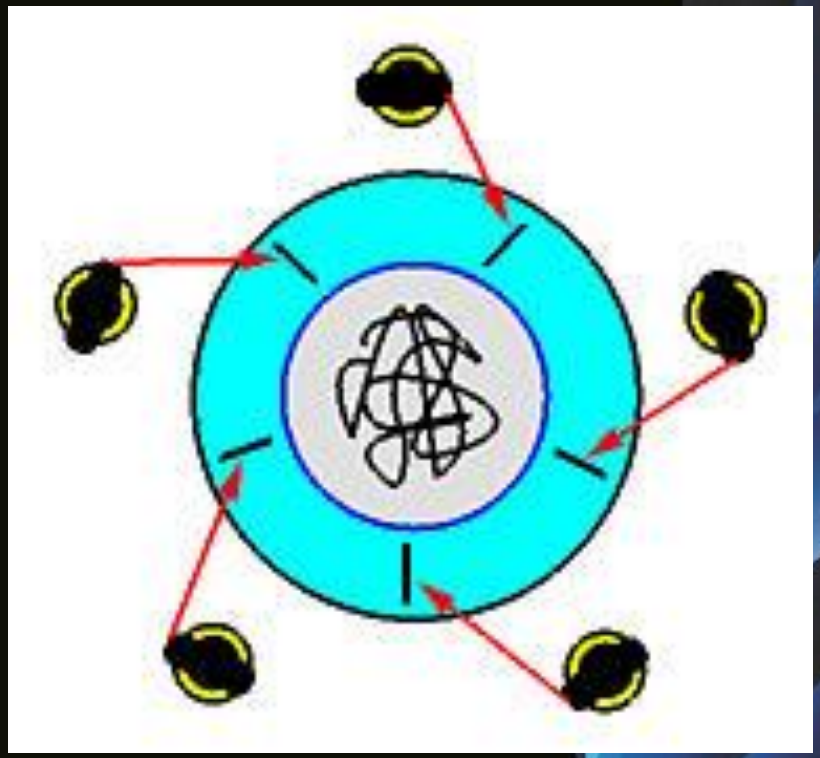
Dining-Philosophers analysis

What if every philosopher sits down about the same time and picks up his left chopstick as shown in the following figure?
In this case, all chopsticks are locked and none of the philosophers can successfully lock his right chopstick. As a result, we have a circular waiting (i.e., every philosopher waits for his right chopstick that is currently being locked by his right neighbor), and hence a deadlock occurs.



Dining-Philosophers analysis

Imagine that two philosophers are fast thinkers and fast eaters. They think fast and get hungry fast. Then, they sit down in opposite chairs as shown below. Because they are so fast, it is possible that they can lock their chopsticks and eat. After finish eating and before their neighbors can lock the chopsticks and eat, they come back again and lock the chopsticks and eat. In this case, the other three philosophers, even though they have been sitting for a long time, they have no chance to eat. This is a starvation.



Solution for Dining Philosophers…

Shared data:

```
state [ 5 ] of (thinking, hungry, eating);
                                  // init to all thinking//
self [ 5 ]: semaphore;            { init to all 0 }
mutex: semaphore;                 { init to 1 }
left = (i + 4) % 5;               { left neighbor }
right = (i + 1) % 5;              { right neighbor }
```

Solution for Dining Philosophers…

```
Philosopher  ( i ) {
 state [ i ] = hungry;
 wait ( mutex );
      test [ i ];
 signal (mutex);
 if (state [ i ] != eating )
          wait ( self [ i ] );
      … eating …
      wait ( mutex );
          putdown [ i ];
      signal ( mutex );
}
```

```
void test ( int i )  {
     if ( (state [ left ] ! = eating) &&
        (state [ i ] == hungry) &&
        (state [ right ] != eating))  {
                state [ i ] = eating;
                signal (self [ i ] );
         }
 }
void putdown ( int i )  {
      state [ i ] = thinking;
      wait ( self [ i ] );
           // test left and right neighbors
      test ( left  );
      test ( ( right );
}
```

Sleeping – Barbers Problem:

- A barber shop consists of a waiting room with 5 chairs and the barber room containing the barber chair.
- If no customer to serve, the barber goes to sleep to be awakened by the customer.
- If a customer enters the barber shop but all chairs are occupied, the customer leaves the shop; otherwise he sits in one of the free chairs.
- Write the program to coordinate the actions of the barber and the customer.

Sleeping – Barbers Solution

Shared data: **customer = 0, barber = 0, mutex = 1;**
int waiting = 0; CHAIR = 5;

Barber:

```
while (TRUE) {
        wait ( customer );
        wait ( mutex );
          waiting - -;
          signal ( barber );
        signal ( mutex );
        ....
        cut hair();
}
```

Customer:

```
while ( TRUE )  {
   wait ( mutex );
   if ( waiting < CHAIR)  {
          waiting + +;
          signal ( customer );
          signal ( mutex );
          wait ( barber );
          .....
          get haircut();
     } else signal ( mutex);
}
```

THANK YOU