Chapter - 6
Process Synchronization

Presented By

Narzu Tarannum (NTR)

Brac University

Process Synchronization: Objectives

- Concept of process synchronization.
- The critical-section problem, whose solutions can be used to ensure the consistency of shared data
- Software and hardware solutions of the critical-section problem
- Classical process-synchronization problems
- Tools that are used to solve process synchronization problems

What will Cover.....

- Process Synchronization basic Concepts
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classic problems of synchronization

Background

- A cooperating process is one that can affect or be affected by other processes executing in the system.
- Processes can execute concurrently or in parallel
- Concurrent or parallel access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Process Synchronization

- Process Synchronization means sharing system resources by processes in such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.
- Process Synchronization was introduced to handle problems that arose while multiple process executions.

Producer Consumer problem

- A producer process produces information that is consumed by a consumer process.
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers(actually support bounded buffer). This buffer will reside in a region of memory that shared by the producer and consumer processes.
- We can do so by having an integer count that keeps track of the number of full buffers.
- Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.
- The producer and consumer must be synchronized.

Producer Consumer problem

- The bounded buffer problem(Producer-Consumer Problem), is one of the classic problems of synchronization.
- The producer tries to insert data into an empty slot of the buffer.
- The consumer tries to remove data from a filled slot in the buffer.
- The producer must not insert data when the buffer is full.
- The consumer must not remove data when the buffer is empty.
- The producer and consumer should not insert and remove data simultaneously.

PRODUCER–CONSUMER PROBLEM

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

}
```

The producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently

DATA INTEGRITY PROBLEM

- Suppose the value of the variable counter is currently 5.
- The produce and consumer process execute the statements “counter++” and “counter--“ concurrently.
- Calling the execution of these two statements, the value of the variable counter may be 4, 5, 6!
- The only correct result, though is “counter==5” which is generated correctly if the producer and consumer execute separately.

DATA INTEGRITY PROBLEM

count++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

count-- could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

register1 and *register2* is local CPU registers.

Concurrent execution of "*counter++*" and "*counter--*" and allowing them to manipulate the counter variable create incorrect state.

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}

We would arrive at this incorrect state because we allowed both process to manipulate the variable counter concurrently

Race Condition

- A situation where several processes access and manipulate the same data concurrently
- Outcome of the execution depends on the particular order in which the access takes place, is called **race condition.**

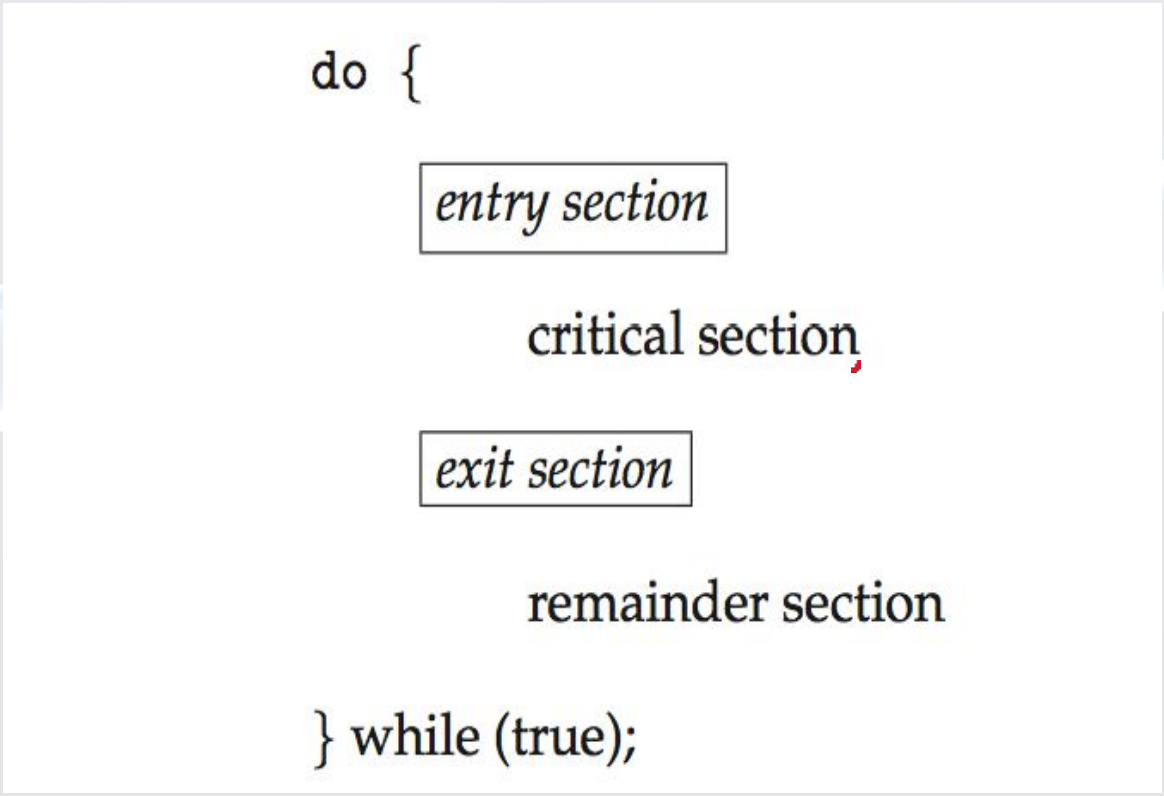
- To guard against the race condition above:
- We need to ensure that only one process at a time can be manipulating the variable **counter** (shared data).
- To make such a guarantee , we require that the processes be synchronized in some way.

Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - In which the process may be changing common variables, updating table, writing file, etc
  - When one process is executing in its critical section, no other process is to be allowed to execute in its critical section
  - That is no two processes are executing in their critical sections at the same time.
- ***Critical section problem*** is to design a protocol that the processes can use to cooperate
- Each process must ask permission to enter its critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i



Requirements of solution to the Critical-Section Problem

A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely
3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

   - Assume that each process executes at a nonzero speed

HARDWARE-BASED SOLUTION TO THE CRITICAL SECTION PROBLEM

- More solutions to the critical-section problem using techniques ranging from hardware to software-based APIs
- These solutions are based on the premise of **locking** — protecting critical regions through the use of locks.
- In a single-processor environment CS problem can be solved by preventing interrupts from occurring while a shared variable is being modified.
- For multiprocessor environment, we need different measures.
- Modern computer systems allow to test and modify the content of a word or to swap the contents of two words atomically – which is uninterruptable unit. We can use *test_and_set()* and *compare_and_swap()* instructions.

Synchronization of Hardware

- Many systems provide **hardware support** for critical section code
- Uniprocessors or single processor– could disable interrupts while modified a shared variable.
  - Currently running code would execute without preemption. So no unexpected modifications could be made to the shared variable.
  - This is the approach taken by non-preemptive kernels.
- Generally too inefficient on multiprocessor systems
  - Because, disabling interrupts on a multiprocessor can be time consuming.

- Modern machines provide special atomic hardware instructions: TestAndSet ; Compare and Swap ;
  - Atomic = non-interruptable
  - Either test memory word and set or modify the content of a word
  - Or swap contents of two memory words

test_and_set Instruction

- There is a shared lock variable which can take either of the two values, 0 or 1.
- Before entering into the critical section, a process inquires about the lock.
- If it is locked, it keeps on waiting till it becomes free.
- If it is not locked, it takes the lock and execute the critical section.

Definition:

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

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

test_and_set Instruction and its implementation

Mutual exclusion can be implemented by initializing a Boolean variable lock to false

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Figure 5.3 The definition of the test_and_set() instruction.

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Figure 5.4 Mutual-exclusion implementation with test_and_set().

COMPARE_AND_SWAP( ) INSTRUCTION

- Mutual exclusion can be achieved by declaring a global variable *lock* and initializing it to *0*
- First process that invokes this instruction will set *lock* to *1* and no other process can execute CS until this process updates it to *0* after CS execution.

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

Figure 5.5 The definition of the compare_and_swap() instruction.

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

Figure 5.6 Mutual-exclusion implementation with the compare_and_swap() instruction.

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest of these tools is" mutex lock" (Mutex = Mutual Exclusion)
- A process must acquire the lock before entering CS [*acquire()* function]
- A process must release the lock after exiting the CS [*release()* function]
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
  - But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Solution to Critical-section Problem Using mutex Locks

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

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}
```

```
release() {
    available = true;
}
```

- A simple tool requires-lock
- Race conditions are prevented by requiring that critical section be protected by locks.
- The Boolean variable available is shared between the processes and initially it is true.

MUTEX IMPLEMENTATION

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
int t_id[2]={1,2};
void *t_func(int *id);
int sum=0;
pthread_mutex_t mutex;
int main(){
        pthread_t t[2];
        pthread_mutex_init(&mutex,NULL);
        for(int i=0;i<2;i++){
            pthread_create(&t[i],NULL,(void *)t_func,&t_id[i]);
        }
        for(int i=0;i<2;i++){
                pthread_join(t[i],NULL);
        }
        pthread_mutex_destroy(&mutex);
        printf("Total count: %d\n",sum);
        return 0;
}
void *t_func(int *id){
        printf("Entered in Thread %d...\n",*id);
        for(int i=0;i<3;i++){
                pthread_mutex_lock(&mutex);
                sum++;
                pthread_mutex_unlock(&mutex);
        }
}
```

Declaring mutex variable

Initializing the mutex

Destroying the mutex after usage

Calling acquire func.

Calling release func.

Output:

```
Entered in Thread 2...
Entered in Thread 1...
Total count: 6
```

Semaphore

- Hardware based solution to the Critical Section (CS) problem are complicated for application programmer to use. To overcome this difficulty a synchronization tool called **semaphore** can be used.
- A **semaphore** is a protected integer variable that can facilitate and restrict access to shared resources in a multi-processing environment.
- This variable is used to solve critical section problem and to achieve process synchronization in multi-processing environment.

Semaphore Variables

- **Semaphore S** is an integer variable ; can not modify directly. Using two standard atomic operations we can modify wait() and signal()
- Originally called **P( )** and **V( )** (Less complicated)
  - Wait( ) or P( ) means to “test”
  - Signal( ) or V( ) means to “increment”

Semaphore Variable Definition

```
P(Semaphore S) {
        while (S ≤ 0)
        ; // No operation
        S = S-1;
}
```

```
V(Semaphore S) {
S = S+1;
}
```

- **P(S) and V(S) operations are atomic in nature that means when a process executing the process P(S) or V(S) that can not be Interrupted.**
- **All the modifications to the integer value of the semaphore in the wait() and signal() operations must be executed individually. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.**
- **Using semaphore we can implement mutual exclusion very easily.**

Types of semaphore

- **Binary Semaphore:** The value can range only between 0 and 1. This behaves similar to Mutex Lock and can Solves various synchronization problems.

- **Counting Semaphore**: The value can range over an unrestricted domain.
  - Used to control access to a given resource consisting of finite number of instances

implement mutual exclusion using Semaphore Variable

- **Binary semaphore** – If there is a single resource(CS) only , integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex lock.

```
P(Semaphore S) {
        while (S ≤ 0)
        ; // No operation
        S = S-1;
}
```

```
V(Semaphore S) {
S = S+1;
}
```

- Implementation with mutex lock:

```
mutex: Semaphore;     mutex = 1;

P_i:    P(mutex)
        CS          //Critical Section
        V(mutex)
        RS          //Remainder Section
```

Solve various synchronization problem using semaphore

- Binary semaphore (mutex) initialized to 1which provides mutual exclusion, used to solve CS problem.[example shown in previous slide]
- Foe solving various synchronization problems we need to initialize binary semaphore to 0:
- Here is an example of deciding the order of execution of process.
- Consider ***P*** $_1$ and ***P*** $_2$ are two processes that require ***S*** $_1$ to happen before ***S*** $_2$

P_1 has statement S_1
P_2 has statement S_2

Create a semaphore variable "**`sync`**" initialized to 0

P1:

```
S1;
signal(sync);
```

P2:

```
wait(sync);
S2;
```

Counting Semaphore

- **Counting semaphore** –The value can range over an unrestricted domain.
- Used to control access to a given resource consisting of finite number of instances. If there are more than one but limited resources.
- Initialized to the number of resources available, **S = n**

  Each process that wishes to use a resource performs a wait() operation

  S = S -1

  When a process releases a resource, it performs a signal() operation

  S = S +1

When **S** becomes **0**, all resources are being used
processes that wish to use a resource will block until **S>0**

Problem with Semaphore Implementation

- The main disadvantages of semaphore is it requires busy waiting.
- While a process is in it's critical section, any other process that tries to enter it's critical sections must loop continuously in the entry code. This is called **busy waiting** and it wastes CPU cycles. When a semaphore does this, it is called a **spinlock**.
- Busy waiting wastes CPU cycles that some other processes might be able to use productively.

Semaphore Implementation with no Busy waiting

- To avoid busy waiting each semaphore there is an associated waiting queue of process that are waiting to access the critical section.
- Rather than using a semaphore as a variable we can use it as a structure or record which have two fields:
  - value (of type integer)
  - list of the processes/ pointer to next record in the list

Declaration of semaphore as a structure

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

Declaration of semaphore as a record

```
semaphore = record
      value : integer;
      L : list of processor;
end;
```

Semaphore Implementation with no Busy waiting

- Two operations provide: block() & wakeup()
  - When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait
  - Rather than this busy waiting, the process can block itself which places it into a waiting queue associated with the semaphore
  - State of the process is switched to the waiting state and control is transferred to CPU scheduler which selects another process to execute.
  - It will be restarted when some other process executes a signal() operation
  - Restarted by a wakeup() operation that changes it from waiting state to ready state.
- "No busy waiting" means that whenever the process wakes up from waiting, the condition it was waiting for should hold. That is, it must not wake up, find the condition false, and again wait on the same semaphore.

Semaphore Implementation with no Busy waiting

```
Structure semaphore{
        int value;
        queue L;
        }
       wait (S){
              s.value=s.value-1;
               if (s.value <0)
                   {add this process S.L to waiting queue
                  block( );
                          }
                 }

                    Signal (S)
                    {
                     S.value=S.value+1;
                     if (S.value<=0)
                    { remove a process S.L from the waiting queue;
                                  wakeup( );  }
                      }
```

SEMAPHORE IMPLEMENTATION

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
int t_id[2]={1,2};
void *t_func(int *id);
int sum=0;
sem_t s;
int main(){
        pthread_t t[2];
        sem_init(&s,0,1);
        for(int i=0;i<2;i++){
            pthread_create(&t[i],NULL,(void *)t_func,&t_id[i]);
        }
        for(int i=0;i<2;i++){
                pthread_join(t[i],NULL);
        }
        sem_destroy(&s);
        printf("Total count: %d\n",sum);
        return 0;
}
void *t_func(int *id){
        printf("Entered in Thread %d...\n",*id);
        for(int i=0;i<3;i++){
                sem_wait(&s);
                sum++;
                sem_post(&s);
        }
}
```

Declaring semaphore variable

Initializing the semaphore

Destroying the semaphore after usage

Calling wait func.

Calling signal func.

Output:

```
Entered in Thread 2...
Entered in Thread 1...
Total count: 6
```

Classical Problems of Synchronization

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

Bounded-Buffer Problem

- The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time.
- The problem is to make sure that the producer won't try to add data into the buffer if it's full
- and that the consumer won't try to remove data from an empty buffer.
- The producer and consumer should not insert and remove data simultaneously.

Solution for Bounded-Buffer Problem

- The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again.
- In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

Inadequate implementation

- In the solution two library routines are used, sleep and wakeup. When sleep is called, the caller is blocked until another process wakes it up by using the wakeup routine. The global variable itemCount holds the number of items in the buffer.

Inadequate Solution/ Implementation

```
procedure producer(){
    while(true){
    item = produceItem();
    if(itemCount == BufferSize)
Sleep();
    putIntoBuffer(item);
    itemCount ++;
    if(itemCount == 1)
wakeup(consumer);
    }
}
```

```
procedure consumer(){
    while(true){
    if(itemCount == 0)
Sleep();
    item = removeFromBuffer();
    itemCount --;
    if(itemCount == BufferSize - 1)
wakeup(producer);
    }
}
```

DEADLOCK !!!

If, consumer is interrupted just after checking the itemCount value and before sleep() call, and producer produces an item it will call wakeup(). But, this call will be lost as no consumer in sleeping yet. And it will continue to produce items. Eventually, producer will sleep when the buffer is full. On the other hand, consumer will also stay in sleep as producer won't call wakeup().

Deadlock and Starvation

- Deadlock – The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by 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);

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

it contains a race condition

- Since both processes will sleep forever, we have run into a deadlock. This solution therefore is unsatisfactory

Solution for multiple producer and consumer using Semaphores

We will make use of three semaphores:

1. m(mutex), a binary semaphore which is used to acquire and release the lock.
2. empty, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
3. full, accounting semaphore whose initial value is zero.

Semaphore for multiple producer consumer

semaphore **mutex** = 1;

semaphore **full** = 0;

semaphore **empty** = BUFFER_SIZE;

```
procedure producer(){
do{

wait(empty); //wait until    empty>0 and then
decrement 'empty'
wait(mutex); //acquire lock

/* add data to buffer*/

signal(mutex); // release lock
signal(full); // increment 'full'
}
   } while(true)
```

```
procedure consumer(){
do{

wait(full);//wait until full>0 and then
decrement 'full'
wait(mutex); //acquire lock

/* remove data from buffer*/

signal(mutex); // release lock

signal(empty);// increment 'empty'

    }
  } while(true)
```

The Readers-Writers Problem

- There is a data area or Database that is shared among a number of concurrent processes.
- Some of these processes may want only to read the database, whereas others may want to update (write) the database.
- We distinguish between these two types of processes by referring to the former as **readers** and to the later as **writers**.
- Obviously if two readers access the data simultaneously, no adverse effects will result.
- If a writer and some other thread (either a reader or writer) access database the simultaneously, chaos may ensure.
- To ensure that these difficulties do not arrives, we require that the writers have exclusive access to the shared database.
- These synchronization problem is referred to as Readers-Writers Problems.

Readers-Writers Problem

- Data/ database is shared among number of processes
- Any number of readers may simultaneously read the data
- Only one writer can write to the data at a certain time
- If a writer is writing, no reader can read
- If at least one reader is reading, no writer can start writing
- Readers only read, Writers only write

Readers-Writers Problem

- The reader-writers has several variations, all involving priorities.
  - *first variation,* requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words no reader should wait for other reader to finish simply because a writer is waiting.
  - *second variation*- requires that, once a writer is ready, that writer performs it write as soon as possible. In other words if a writer is waiting to access the object, no new readers may start reading.

- Solution to either problem may create starvation.
  - First case: writer may starve
  - Second case: reader may starve

Readers-Writers Problem

- Solution to First Variation / case

- We will make use of two semaphores and integer variable
  - `semaphore rw_mutex = 1;`
  - `semaphore mutex = 1;`
  - `int read_count = 0;`
  - The semaphore rw_mutex is common to both reader and writer processes.
  - The mutex semaphore is used to ensure mutual exclusion when the variable read_count is updated. i.e. when any reader enters or exit from the critical section.
  - The read_count variable keeps track of how many processes are currently reading the object.

Readers-Writers Problem

- **writer** process has the following code skeleton -

```
do{

    /* write request for critical section*/
    wait(rw_mutex);

    /* writing is performed */

    // Leaves the critical section //
    signal(rw_mutex);
} while(true);
```

- If a writer is in the critical section and n readers are waiting
  - One reader is queued on rw_mutex
  - Other n-1 readers are queued on mutex
- When a writer executes signal(rw_mutex)
  - Resume the execution of either waiting readers or single waiting writer
- Multiple process can acquire rw_mutex in read mode
- Only one process can acquire rw_mutex in write mode

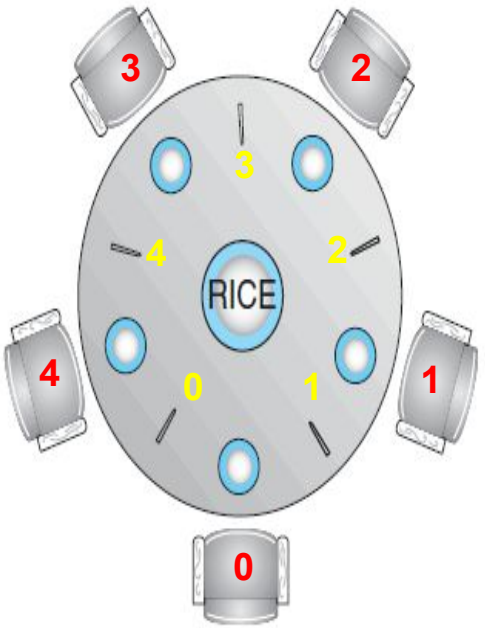
Readers-Writers Problem

- **reader** process has the following code skeleton -

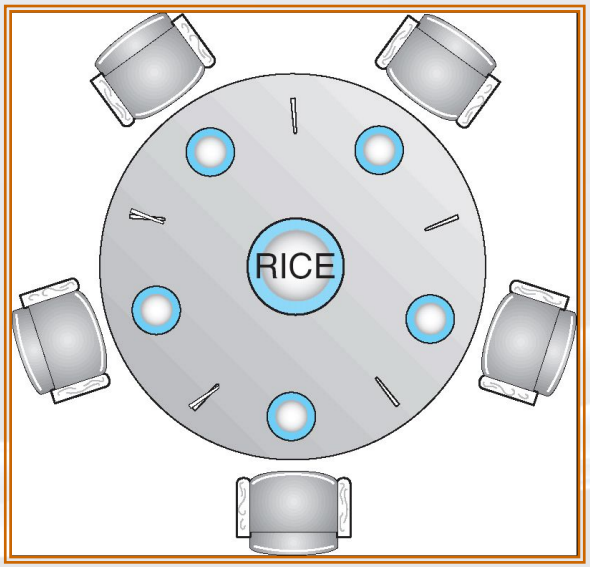
```
do{
    wait(mutex);
    read_count++;      // the number of readers has now increased by 1
    if(read_count == 1)
    wait(rw_mutex); //this ensure no writer can enter if there is even one reader
    signal(mutex); //Other readers can enter while this current reader is inside the critical section

/* reading is performed */

    wait(mutex);
    read_count--;            //reader wants to leave
    if(read_count == 0)     //no reader is left in the critical section.
    signal(rw_mutex);      //writers can enter
    signal(mutex)         //reader leaves
} while(true);
```

Dining-Philosophers Problem

- Problem statement
- Five silent philosophers sit at a round table with bowls of rice. Forks are placed between each pair of adjacent philosophers.
- Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when he has both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After he finishes eating, he needs to put down both forks so they become available to others. A philosopher can take the fork on his right or the one on his left as they become available, but cannot start eating before getting both of them.
- When she is finished eating, she puts down both chopsticks and starts thinking again



Dining-Philosophers Problem



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

Must satisfy mutual exclusion - no two philosopher can use the same fork at the same time.

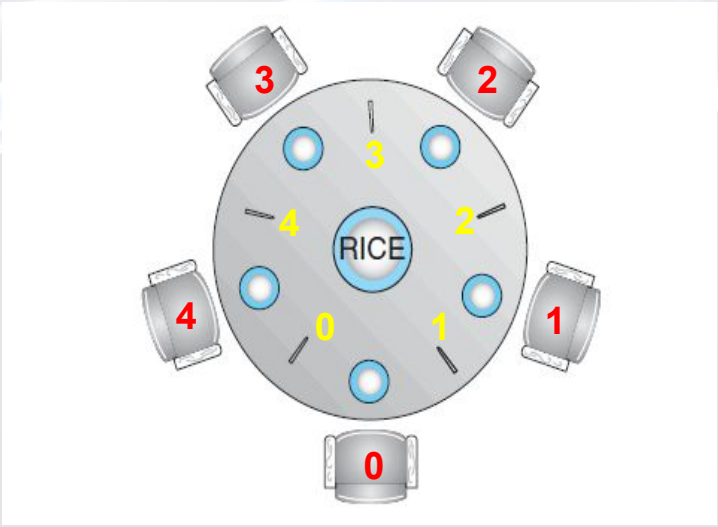
Solution to the problem

One of the simple solution is to use 5 semaphores for 5 chopsticks.

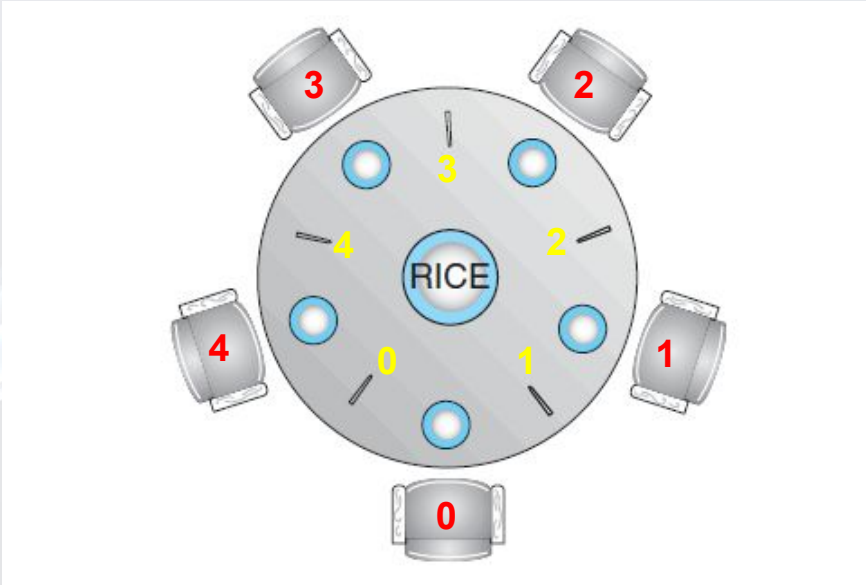
```
semaphore chopstick[5];
```

Structure of i^{th} philosopher's process can be -

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

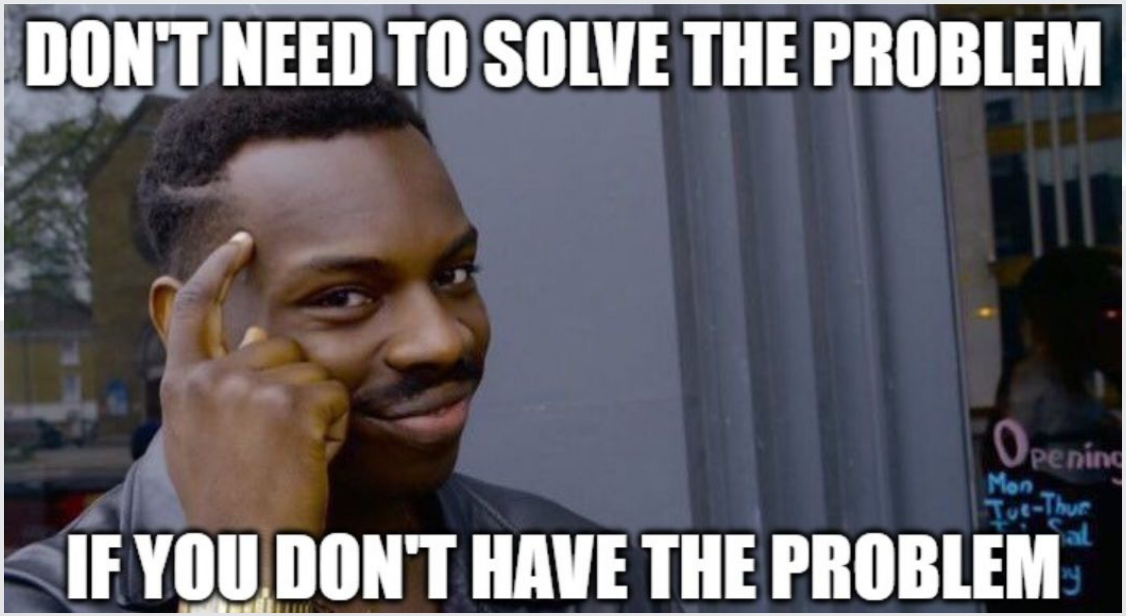


BUT GUESS WHAT'S COMING????



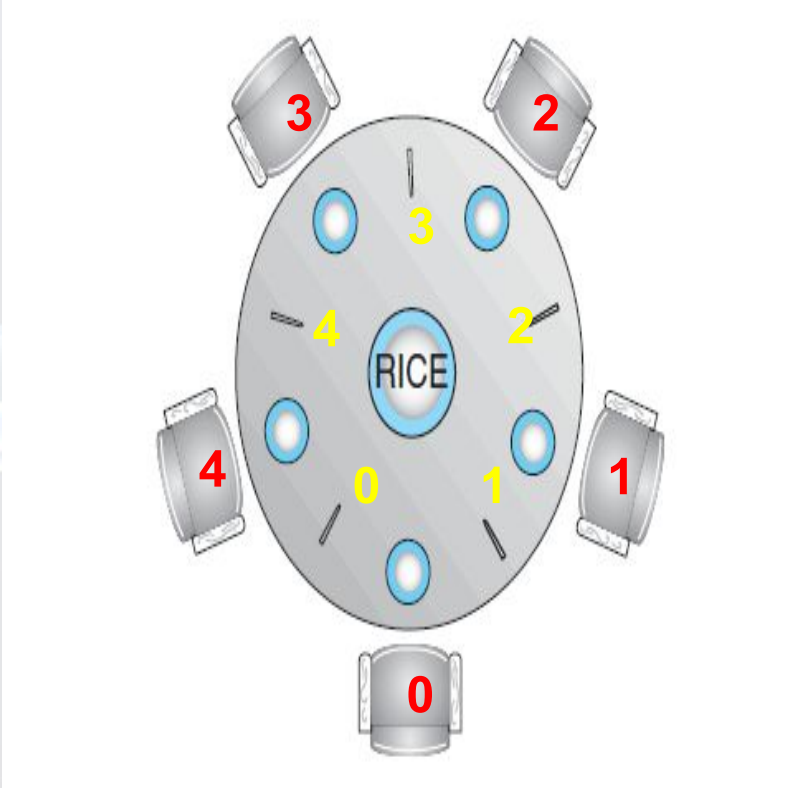
- In this situation it guarantees that no two neighbors are eating simultaneously. It could create a deadlock.
- If all the philosophers are hungry at the same time and grab their left chopstick by calling wait(chopstick[i]), No one will ever find a second chopstick for eating. All the elements of chopsticks will now be equal to zero. Consequently, they will die from starvation :P

How to prevent deadlock in this solution?



How to prevent deadlock in this solution?

- Allow at most four philosophers to be sitting simultaneously at the table
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution — that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.



Necessary Conditions for Deadlock

Necessary conditions for deadlocks are -

- Mutual Exclusion
- Hold and Wait
- No Preemption (cannot force to release the resource)
- Circular wait

If one of these are not present in a system, then the system is "**Deadlock Free**".

Mutual exclusion: At least one resource must be held in a non-sharable mode; only one process at a time can use a resource.
Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes.
No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task.
Circular wait: There must be a circular chain of two or more processes each of which is waiting for a resource held by the next member of the chain.

End of Chapter 6
Thank You