Operating Systems (UNIT-3)

Process Synchronization

Cooperating processes : The Process that can affect or be affected by the other process executing in the system.

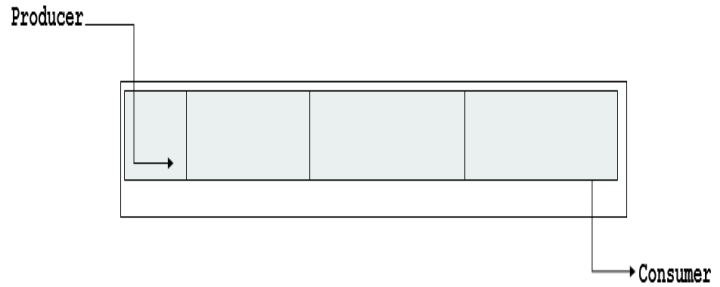
Cooperating Process : Share the data directly or they can be in the shared memory.

Concurrent access to shared data may result in data inconsistency

Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Concurrent access to the shared memory may result in Data Inconsistency (If the two or more process try to access the same data in the shared memory, there will be a problem- to address this issue we need Process Synchronization).

For Example



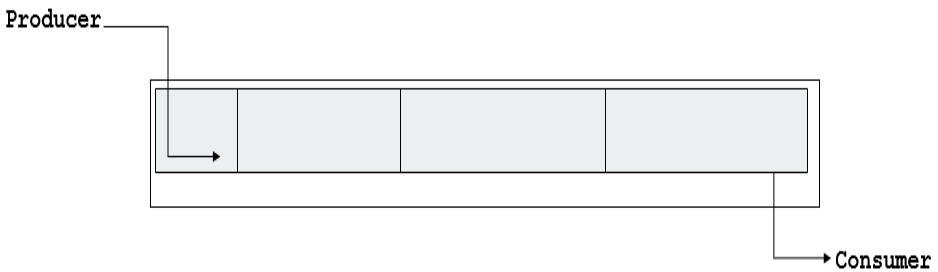
Bounded Buffer Problem also termed as the Producer Consumer Problem.

There is buffer of n slots, and each slot have capable of storing one unit of Data.

Process Synchronization

We need to have the synchronization between producer and consumer i.e Consumer should not try to consume the data that has not yet produced.

Illustration of the problem:
Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an **integer `counter`** that keeps track of the number of full buffers.
Initially, **`counter`** 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



Producer and Consumer Code

- **Producer Code.**

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

      while (counter ==
BUFFER_SIZE) ;
            /* do nothing */
      buffer[in] =
next_produced;
      in = (in + 1) %
BUFFER_SIZE;
      counter++;
}
```

- **Consumer Code**

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

Producer Consumer Problem

- Let us assume the value of Counter is 5.
- Now producer and consumer processes concurrently execute the statements “counter++” and “counter--”
- The value of the counter may be 4, 5 or 6, 5….. Which is the correct value in this if there is a synchronization between process and consumer.
- The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.
- Do we really get the value of Counter = 5 if both producer and consumer execute concurrently.

Producer Consumer Problem

- When the counter statement is executed at the **machine level** it will be executed as below.
- Let us take the value of the counter = 5
- “counter++” may be implemented in machine language.
- register1 = counter = (register1 =5 & counter = 5)
- register1 = register1 + 1 = (register1 =6 & counter = 5)
- counter = register1 = counter = 6
- First the value of counter will be stored in register, register will be incremented and then that will stored back to counter
- Counter--” may be implemented in machine language.
- register1 = counter = (register1 =5 & counter = 5)
- register1 = register1 - 1 = (register1 =4& counter = 5)
- counter = register1 = counter = 4

Race Condition

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

A situation like this, where several processes access and manipulate the same d
concurrently and the outcome of the execution depends on the particular order in wh
the access takes place, is called a race condition

Critical Section

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each Process have segment of code called Critical Section
- **Critical Section** : Segment of the code of the process, in which the process will be changing the data in the shared region of the memory.
- I.E whenever the process is accessing the shared memory and doing some changes, the code responsible for that is called **Critical Section**
- Each process has **critical section** segment of code
  - Process may be changing **common variables, updating table, writing file**, etc.
  - When one process in critical section, no other may be in its critical section.
- ***Critical section problem*** is to design protocol that the processes can use in order to be synchronized (Process should not change the shared data concurrently) .

Critical Section

- Rules to Operate in Critical Section
- Each process must ask permission to enter critical section and the The section of code implementing this request is the **entry section.**
- The critical section may be followed by an **exit section**.
- The remaining code is the **remainder section.**
- General structure of process P_i

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

Solution Critical Section - must satisfy the following threerequirements

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 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, and this selection cannot be postponed indefinitely.

- 3. **Bounded Waiting** - There exists a bound, or limit, 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.

- If Bounded waiting is not there it will lead to the Starvation

Two approaches depending on if kernel is preemptive or non- preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

Two approaches depending on if kernel is preemptive or non- preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

Peterson's Solution

- Software based solution for Critical Section problem.
- It illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.
- Peterson's solution is restricted to **two processes** that alternate execution

between their critical sections and remainder sections. Lets say P_i and P_j

- Peterson's solution requires the two processes to share **two data items**:
- int turn;
- boolean flag[2];
- The variable turn indicates whose turn it is to enter its critical section. That is,if turn == i, then process *Pi* is allowed to execute in its critical section.
- Theflag array is used to indicate if a process is ready to enter its critical section. For example, if flag[i] is true, this value indicates that *Pi* is ready to enterits critical section.

Peterson's Solution- Algorithm

```
do {
        flag[i] = true; # Process i is
  ready to enter the critical section
       turn = j; # But if the other process
wishes to enter the critical section, it can
do so
    while (flag[j] && turn = = j);#(If this
  condition is true means it will remain in
  while loop only)
#We have semicolon, if the condition is true
  Pi will stay in while loop only.
                critical section
    flag[i] = false;
                remainder section
   } while (true);
```

The structure of process P_i in Peterson's solution

```
do {
        flag[j] = true; # Process j is
  ready to enter the critical section
       turn = i; # But if the other
process wishes to enter the critical
section, it can do so
    while (flag[i] && turn = = i);
                critical section
    flag[j] = false;
                remainder section
  } while (true);
```

The structure of process P_j in Peterson's solution

Peterson's Solution

- We now prove that this solution is correct. We need to show that:
- **1.** Mutual exclusion is preserved.
- **Pi will enter into crticial section only if one among these is falls(flag[j] && turn = = j) and if one among these are falls pj will not be able to enter**
- **2.** The progress requirement is satisfied.
- **Pi and Pj were not in reminder section and one among them took the decision of who will enter into section.**
- **3.** The bounded-waiting requirement is met
- There is no indefinite wait as soon as one process enters critical section, the other process is also allowed.

Synchronization Hardware

- Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures.
- All the hardware solutions below based on idea of **locking**
  - Protecting critical regions via locks

test_and_set Instruction

- Hardware solution for synchronization problem
- It takes two values 0 and 1.
- 0 means unlock and 1 means lock.
- Process will enquire about lock before entering into critical section.
- If the Lock value is 1 means it will wait for the lock to become zero.
- If the Lock value is 0 means it will set the lock and enter into critical section

test_and_set Instruction

Definition:

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

It has Boolean return type i.e : It returns only true or false

It takes the Boolean variable (Target and rv)

Target is shared lock variable that takes the value 0 and 1.

It will be assigned to rv and that will be returned.

Atomic Operation : The whole process of test and set will happen in single machine cycle.

test_and_set Instruction

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

- Solution:

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */
           /* critical section */
    lock = false;
           /* remainder section
*/
 } while (true);
```

- 1. Initially the lock is always set to zero.
- 2. test and set (&lock) will call the Boolean function test and set
- 3. Lock value is target value
- 4. rv = 0 and it will be returned
- 5. target is again set back to true and 0 will be returned
- 5. while(0) – Condition become false and the loop is broke and process will enter the critical section
- 6. So 0 implies that the critical section is unlocked
- 7. When the process is in Critical section, remember the target value was set to 1 in setp5
- 8. Now if again some process comes it will be while (1)– So it will wait
- 9. Now when the process comes out of critical section lock is set to false and now while becomes 0, so now the other process will get a chance to enter
```

Compare and Swap Instruction

Defn of Swap Instruction : void swap (boolean*a, boolean*b)

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

Code for Mutual Exclusion

```
do{ key = true
  while (key ==true);
   swap(&lock,&key)
   # Critical section
   lock = false
}while(True)
```

- 1. Initially the lock is always set to zero.
- 2. Each process have key values
- 3. Key is set to true
- 4.After swapping key becomes false(b) and lock becomes true(a)
- 5.It comes to the while loop and now key is false , so the condition fails, hence it enters the critical section
- 6.Once the process comes out of the critical section it will make lock as zero, so other process can come.
- ***7.Both Test and set and compare and swap provide the Mutual Exclusion but not Bounded waiting***

Mutex Locks

- It takes two values 0 and 1.
- 0 means unlock and 1 means lock.
- If it is zero means, process can acquire the lock and change the value of lock to 1 and enter to critical section.
- Once the process is finished executing in the critical section it will release the lock.
-

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

Mutex Locks

- The Boolean variable acquire is shared between the process and initially it is zero

```
acquire() {
        while (lock! = 0)
            ; /* busy wait */
        available = false;
    }
release() {
        available = true;
    }
```

Semaphores

- **Semaphore is integer variable which is non negative and shared between the variables (Resource) .**
- **Semaphore *S* – integer variable**
- **Can only be accessed via two indivisible operations**
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait() operation`**

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
} (It will check S is less than or equal to Zero, it will stay in while loop, but
  if S becomes greater than Zero, it will break the loop and decrement the S)
```

- It tells the process that, whether the resource(S) is available, if it is not available it will wait, if it is available that resource will be taken by the process and it will decrement one resource value.

Semaphores

- **Definition of the signal() operation**

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

  - **Once the process is completed using the resource, it will return back the resource, hence the S is incremented.**

  - **Wait and Signal Operations are executed indivisibly(If the one process modifies the semaphores value, no other process can simultaneously modify that semaphore value**

Types of Semaphores

Binary Semaphore : The value of the binary semaphore can range between 0 and 1. They are also called as mutex locks, as they are the locks that provide the Mutex exclusion.

(If the value is 0, it means some process is already using the resource, if the value is 1, the resource is available for the process to use.)

Counting Semaphore : Its value can range over an unrestricted domain, it is used to control the access to resources, that have multiple instances.

Lets assume that we have three process P1,P2,P3 and there is resource with two instances (means that resource can be used by two process at the same time)

```
wait(S) {
    while (S >= 0)
       ; // busy wait
    S--;
```

S is set to 2, after first loop, it will become S=1, P1 take the process, after second loop P2 take the process, at third iteration, S will be zero, P3 will wait.

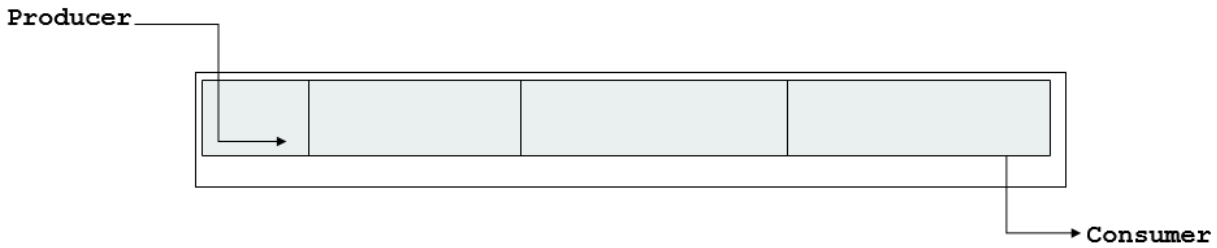
Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded Buffer Problem

Bounded Buffer Problem also termed as the Producer Consumer Problem.

There is buffer of n slots, and each slot have capable of storing one unit of Data.
There are the two process
Producer : Produce the Data and Store in the Buffer
Consumer : Remove the data from the Buffer



Rules:

1. **Producer must not insert the data when the Buffer is full.**
2. **Consumer should not retrieve the data when the Buffer is empty.**
3. **Producer and Consumer should not insert and remove the data simultaneously.**

Solution using Semaphore

- We will make use of 3 Semaphores
- 1. Binary Semaphore (Mutex) : Used to acquire and release Lock
- 2. Empty Semaphore : Counting Semaphore, whose initial value is number of free slots in the buffer, since initially the buffer is empty its value is equal the number of slots in the buffer
- 3.Full Semaphore : Counting Semaphore, whose initial value is Zero

(It is used to track how many number of slots are full, basically it will keep track of filled slots in the buffer)

Solution using Semaphore

- We will make use of 3 Semaphores
- 1. Binary Semaphore (Mutex) : Used to acquire and release Lock
- 2. Empty Semaphore : Counting Semaphore, whose initial value is number of free slots in the buffer, since initially the buffer is empty its value is equal the number of slots in the buffer
- 3.Full Semaphore : Counting Semaphore, whose initial value is Zero

(It is used to track how many number of slots are full, basically it will keep track of filled slots in the buffer)

Producer and Consumer Code

Producer Code

```
do
{
wait(empty); #wait until
empty is > 0  and decrement empty
wait (mutex);acquire lock
#adding of data happen
signal(mutex); release lock
signal (Full);Increment Full
}while (True)
```

Consumer Code

```
do
{
wait(full); #wait until
full is > 0  and decrement
wait (mutex);acquire lock
#adding of data happen
signal(empty); release lock
signal (empty);Increment Full
}while (True)
```

Important methods that can be used with semaphore in c

- Some **sem_init** -> Initialise the semaphore to some initial value
  - **sem_wait** -> Same as wait() operation
  - **sem_post** -> Same as Signal() operation
  - **sem_destroy** -> Destroy the semaphore to avoid memory leak
- One can use include header file and declare a mutex of type pthread_mutex_t in c.
- Some important methods that can be used with semaphore in c
  - **pthread_mutex_init** -> Initialise the mutex
  - **pthread_mutex_lock()** -> Same as wait() operation
  - **pthread_mutex_unlock()** -> Same as Signal() operation
  - **pthread_mutex_destroy()** -> Destroy the mutex to avoid memory leak

Important methods that can be used with semaphore in c

- **pthread_mutex_init** -> Initialise the mutex #This initializes *mutex with the attributes specified by attr. If attr is NULL, a default set of attributes is used. The initial state of *mutex will be "initialized and unlocked".

- **sem_init :** This initializes the semaphore *sem. The initial value of the semaphore will be value. If pshared is 0, the semaphore is shared among all threads of a process (and hence need to be visible to all of them such as a global variable). If pshared is not zero, the semaphore is shared but should be in shared memory
- int sem_init(sem_t * sem, int pshared, unsigned int value)
- **pthread_mutex_lock()** -> Same as wait() operation
- **pthread_mutex_unlock()** -> Same as Signal() operation
- **pthread_mutex_destroy()** -> Destroy the mutex to avoid memory leak

Producer Code

Producer Code

```
do
{
wait(empty); #wait until
empty is > 0  and decrement empty
wait (mutex);acquire lock
#adding of data happen
signal(mutex); release lock
signal (Full);Increment Full
}while (True)
```

Producer Code

```
int waittime,item,i;
        item=rand()%5;
        waittime=rand()%5;
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);
        printf("\nProducer has produced item: %d\n",item);
        write(item);
pthread_mutex_unlock(&mutex);
        sem_post(&full);
```

Producer and Consumer Code

Producer Code

```
do
{
wait(empty); #wait until
empty is > 0  and decrement empty
wait (mutex);acquire lock
#adding of data happen
signal(mutex); release lock
signal (Full);Increment Full
}while (True)
```

Consumer Code

```
do
{
wait(full); #wait until
full is > 0  and decrement empty
wait (mutex);acquire lock
#adding of data happen
signal(empty); release lock
signal (empty);Increment Full
}while (True)
```

Reader Writer Problem

- A database is shared between the concurrent process
- Some of these process may want to only read and some may want to update(Read and Write)
- Those Process who wants to only read - Readers
- Those process who wants to Read and Write – Writers
- If the two readers access the data at same time then there will be no problem
- But if two writers want to access the data at same time then there may be chaos
- To Solve this we should make sure that – Writers get the exclusive access to the shared database system

Reader Writer Problem

To Solve this we will use two semaphores and 1 Integer

- Mutex : (Semaphore) Initialised to 1 – Which is used to maintain the mutual exclusion, when the read count variable is updated i.e when any reader enter and exit from the database
- Wrt(Semaphore) - : Initialised to 1 – common to both reader and writer process and which is used to write.
- Read count (Variable) – Initialised to 0 – Keeps track of how many process are reading the data

Reader and Writer Code

Writer Code

```
do
{
#Writer request for the access
wait(wrt); - #Performs the write and during this time no reader or writer will be allowed
#Performs the write operation
Signal(wrt);
}while (True)
```

Reader Code

do

{

wait(mutex); #wait until

readcount++ // The number of readers are increase by 1

if(readcount ==1) // If I have at least one reader, we should not allow the writers to come , so this will be achieved by calling wrt

Wait(wrt)

Signal(mutex) // other readers can enter while one reader is in the data system

Perform the Read operation

Now read wants to leave

Wait(mutex)

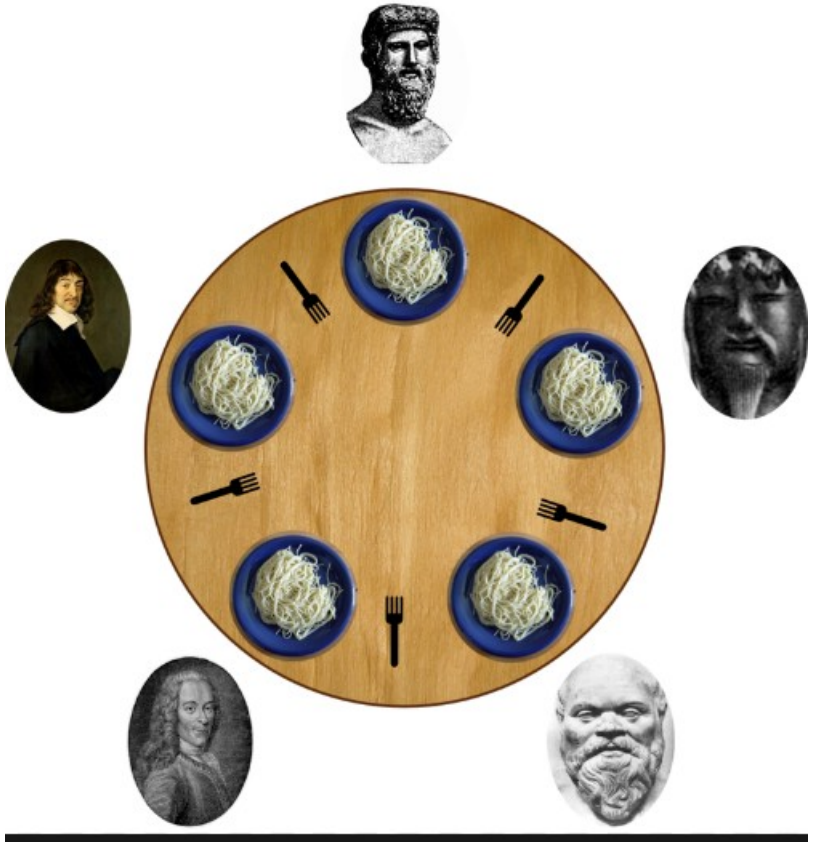
Readcount--

if(readcount ==0) // If I don't have any readers, writer can come

signal(wrt) #

Signal(mutex) // It will free the mutex

Dining Philosopher Problem



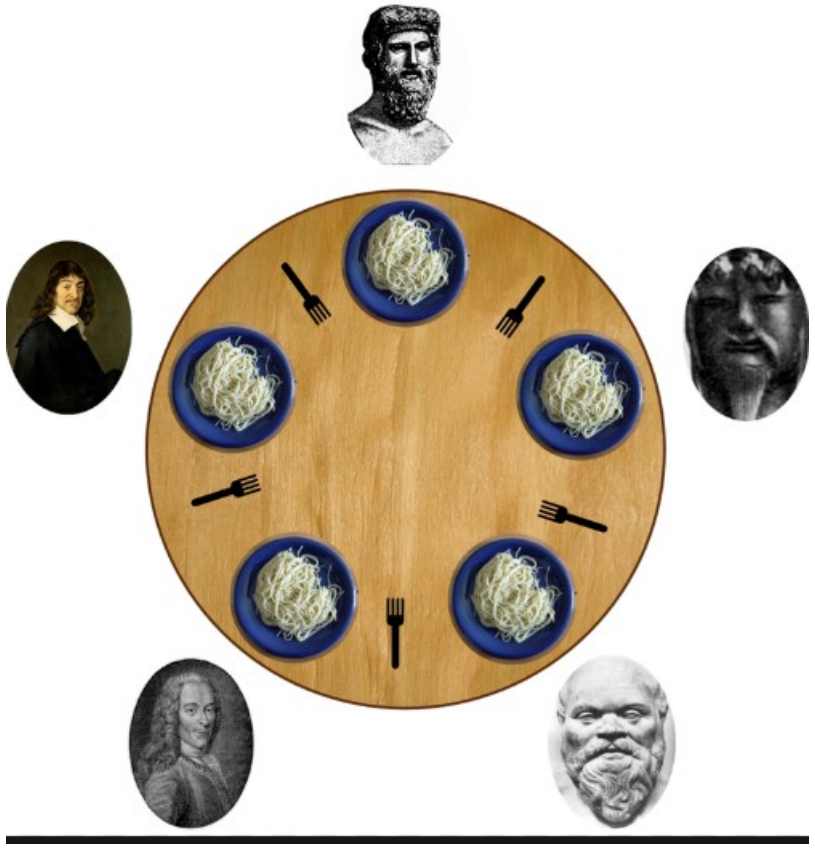
```
do {
.
takefork(i)
takefork(i+1%N))

Eat();

putfork(i)
Putfork((i+1)%N))
```

- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining Philosopher Problem- Some General Solutions



- 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 -- an 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.

Dining Philosopher Problem – Solution using Semaphore

We will take 5 Semaphores and all are initialised to 1

```
do {
.
wait (takefork(Si))
wait(takefork(Si+1%N))

Eat();

signal(putfork(Si))
Signal(putfork((Si+1)%N))
}
while(true
```

Deadlock and Starvation(Drawbacks of Semaphores

- **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);`

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**