**Classical Problems of Synchronization**

* Bounded – Buffer Problem.
* Readers –Writers Problem.
* Dining – Philosophers Problem.

**Bounded – Buffer Problem**

* N buffers, each can hold one item.
* Semaphore **mutex** initialized to the value 1.
* Semaphore **full** initialized to the value 0.
* Semaphore **empty** initialized to the value N.

**Bounded – Buffer Problem Implementation**

The structure of the producer process:

|  |  |
| --- | --- |
| while (true) {  // produce an item  wait (empty);// N  wait (mutex);// 1  // add the item to the buffer  signal (mutex);  signal (full);// 0  } | wait( S) {  S.value --;  if ( S.value < 0 ) {  add this process to S.List ;  block( );  }  } |
| signal ( S) {  S.value ++;  if (S.value <=0 ) {  remove a process P from S.list;  wakeup( P );  }  } |

The structure of the consumer process:

|  |  |
| --- | --- |
| while (true) {  wait (full);  wait (mutex);  // remove an item from buffer  signal (mutex);  signal (empty);  } | wait( S) {  S.value --;  if ( S.value < 0 ) {  add this process to S-> List ;  block( );  }  } |
| signal ( S) {  S.value ++;  if (S.value <=0 ) {  remove a process P from S.list;  wakeup( P );  }  } |

**Readers-Writers Problem**

**Writers**: - they can both read and write.

**A data set:** -itis shared among a number of concurrent processes.

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

**Problem**: - allow multiple readers to read at the same time, only one single writer can access the shared data at the same time.

**Shared Data: -** they are

* + Data set
  + Semaphore mutex initialized to 1.
  + Semaphore wrt initialized to 1.
  + Integer read count initialized to 0.

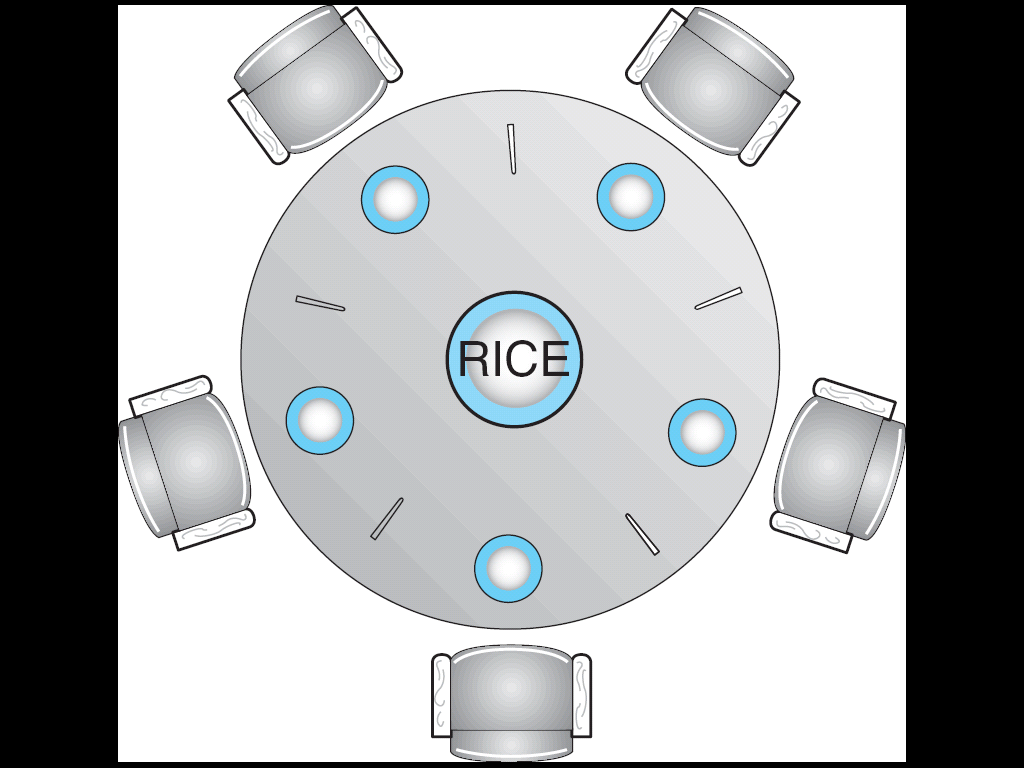
The structure of a writer process

|  |  |
| --- | --- |
| while (true) {  wait (wrt) ;//1    // writing is performed  signal (wrt) ;  } | acquire( S) {  S.value --;  if ( S.value < 0 ) {  add this process to S-> List ;  block( );  }  } |
| release ( S) {  S.value ++;  if (S.value <=0 ) {  remove a process P from S.list;  wakeup( P );  }  } |

The structure of a reader process

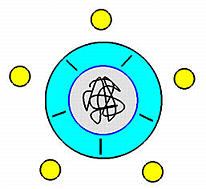
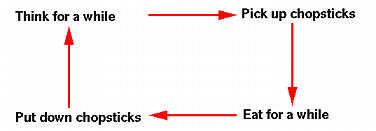
|  |  |
| --- | --- |
| while (true) {  wait (mutex) ; // 1  readcount ++ ; // 0  if (readcount == 1)  wait (wrt) ;  // update by writer process  signal (mutex)  // reading is performed  wait (mutex) ;  readcount - - ;  if (readcount == 0)  signal (wrt) ;  signal (mutex) ;  } | wait( S) {  S.value --;  if ( S.value < 0 ) {  add this process to S.List ;  block( );  }  } |
| signal ( S) {  S.value ++;  if (S.value <=0 ) {  remove a process P from S.list;  wakeup( P );  }  } |
| Note: for first operation, no reading in case of writing and no writing in case of reading. | |

**Dining-Philosophers Problem**

****

**Shared data: -** they are

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

The structure of Philosopher *i*:

|  |  |
| --- | --- |
| While (true) {  wait ( chopstick[i] );  wait ( chopstick[ (i + 1) % 5] );    // eat  signal ( chopstick[i] );  signal (chopstick[ (i + 1) % 5] );    // think  } | wait( S) {  S.value --;  if ( S.value < 0 ) {  add this process to S. List ;  block( );  }  } |
| signal ( S) {  S.value ++;  if (S.value <=0 ) {  remove a process P from S.list;  wakeup( P );  }  } |

**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. (Circular Waiting for resources)

**Starvation**: indefinite blocking .a process may never be removed from the semaphore queue in which it is suspended.

Figure 1: there is no deadlock

Figure 2: there is no deadlock, but process A waiting for Process B

Figure 3: there is deadlock, because that the process A waiting for Process B and process B waiting for A