***SUBJECT: OPERATING SYSTEMS***

**NAME:** ANEEQ ASGHAR

**Roll number:** 17L-4525

**Section:** EE-6A

**Semester:** Spring 2020

**Synchronization problem in the context of multiprocessor systems**

Synchronization refers to the operation or activity of two or more things at the same time or rate. There are two types of synchronization related concepts in Computer Science:

1. Process Synchronization
2. Data Synchronization

Process Synchronization is a way to coordinate processes that use shared data, multiple processes are to join up or handshake at a certain point, in order to reach an agreement or commit to a certain sequence of action. It is sharing system resources by processes in such a way that, simultaneous access to shared data is handled thereby minimizing the chance of inconsistent data. To keep up information consistency requests systems to guarantee synchronized execution of collaborating forms.

There are two types of execution techniques of a process namely serial and parallel execution.

Parallel processing means that at least two microprocessors handle parts of an overall task. The concept is pretty simple: A computer scientist divides a complex problem into component parts using special software specifically designed for the task. Parallel execution is the ability to apply multiple CPU and I/O resources to the execution of a single database operation.

Serial processing means strictly sequential, without overlap of the successive processing times on objects or distinct subsystems. The next object begins processing only when the previous one is completed.

**Types of processes:**

Processes are divided into two types on the basis of process synchronization namely cooperative and independent processes.

1. Independent Process: Execution of one process does not affect the execution of other processes.
2. Cooperative Process: Execution of one process affects the execution of other processes.

Process Synchronization is an approach to organize forms that utilization shared information. It happens in a working framework among participating procedures. Collaborating forms are forms that share assets. While executing numerous simultaneous procedures, process synchronization assists with keeping up shared information consistency and participating procedure execution. Procedures must be planned to guarantee that simultaneous access to shared information doesn't make irregularities. Information irregularity can bring about what is known as a race condition. A race condition happens when at least two activities are executed simultaneously, not planned for the best possible grouping, and not left in the basic segment accurately.

**Why process synchronization is a necessity?**

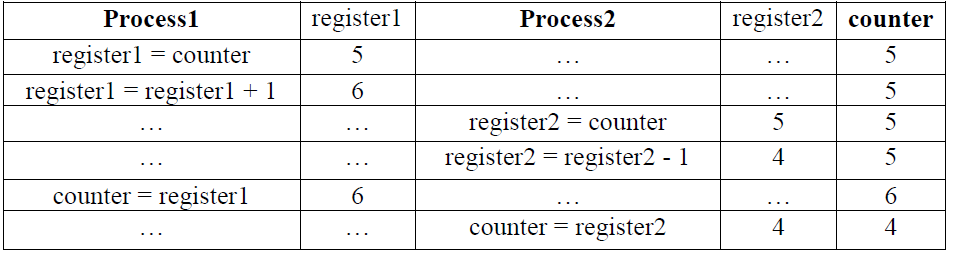
The requirement for synchronization begins when procedures need to execute simultaneously. The fundamental reason for synchronization is the sharing of assets without obstruction utilizing common avoidance. It is necessary only for cooperative process which share information.

**Impacts of not synchronizing processes:**

Three main issues which arise if we do not synchronize the process are inconsistency, starvation and deadlock. Undesirable output is received (Inconsistency) when processes aren’t synchronized. Starvation means that a process is not getting CPU for a very long period of time and Deadlock means that a condition in which processes are not getting their desired resources because other processes is not releasing that resource.

Process synchronization is extremely important in order to avoid inconsistency, starvation and deadlock. An example of Co-operative process in which we have a variable which is shared among two processes and we get two different answers that is starting execution from the PROCESS-I, we get the value of variable equal to 4 and when we start execution from PROCESS-II, we get the value of variable equal to 6. (In this example, we are considering **uniprocessor system**)

A variable counter is set to 5.



Actually, we are incrementing the value of variable using one process (PROCESS-I) and decrementing the value of variable using second process (PROCESS-II). If the processes would be synchronized, we would have the value of variable equal to 5 that is when we increment we will get 6 and when we will decrement we will get again 5. But, we get the wrong output just because of sleep command. So, here we have an error because processes are not synchronized.

**OUTPUT:**

**Value of counter is equal to 5 in case of synchronized process.**

**Value of counter is equal to 4 in case of synchronized process.**

After adding 1 and subtracting 1 from the counter we should get the value of 5 which was original value of counter but due to parallel execution of both the processes we got a value of 4 as the system was not synchronized. This results in a race condition.

**Race condition:**

A race condition is actually a situation that several tasks access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access take place.

To avoid shared resources being used simultaneously by multiple processes, data that is shared between multiple processes must be placed in a critical region. A critical section is a segment of code that can be accessed by only one signal process at a certain instance in time. This section consists of shared data resources that need to be accessed by other processes. Only one process can be executed inside the critical section at a time. Other processes waiting to execute their critical sections have to wait until the current process finishes executing its critical section.

**Solution to race condition:**

To avoid race condition we need Mutual Exclusion. Mutual Exclusion is some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same things.

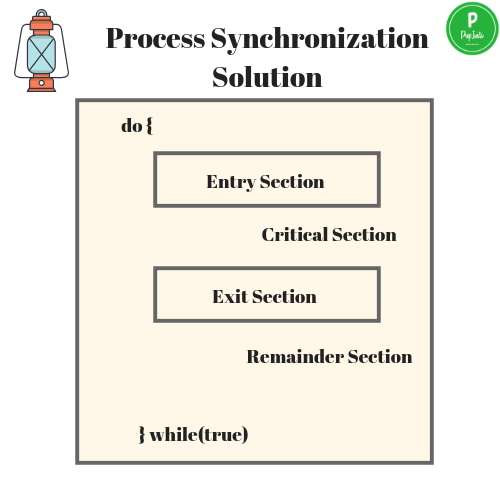
(Rules for avoiding Race Condition):

1. No two processes may be simultaneously inside their critical regions. (**Mutual Exclusion)**
2. No assumptions may be made about speeds or the number of CPUs.
3. **Progress**: No process running outside its critical region may block other processes.
4. **Bounded Wait**: No process should have to wait forever to enter its critical region.

There are four sections of a program:

1. Non Critical Section
2. Entry Section
3. Critical Section
4. Exit Section

We have a segment in the program called Remainder Section or Non-Critical Section. This area can be utilized at the same time by various procedures since it has no impact on the synchronization in light of the fact that there aren’t any mutual assets present in this segment. Entry Section in the code which permits to enter a process into a basic area and it procures a lock. This area doesn't permit different procedure to clear the passage segment code to accomplish common avoidance. While, Exit Section is a code which expels the lock and let different procedures which are holding on to enter the basic area.



**Solving Synchronization problem in processes using semaphores:**

A semaphore is a variable or abstract data type used to control access of a common resource by multiple processes in a concurrent system such as a multitasking operating system. A semaphore is simply a variable. This variable is used to solve critical section problems and to achieve process synchronization in the multiprocessing environment.

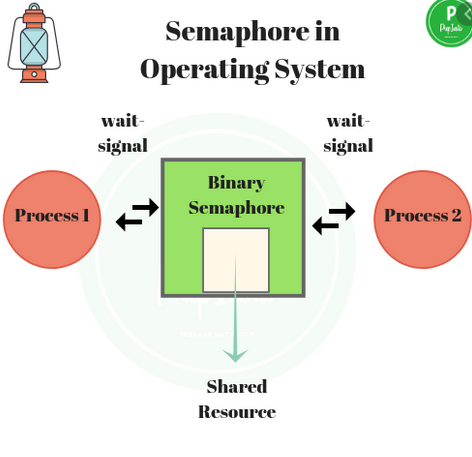
Semaphores may be binary (0/10), or counting. Every semaphore variable, s, it is initialized to some positive value

* 1 for a binary semaphore
* N > 1 for a counting semaphore

**Binary semaphore:**

A binary semaphore is restricted to values of zero or one, while a counting semaphore can assume any nonnegative integer value. A binary semaphore can be used to control access to a single resource. In particular, it can be used to enforce mutual exclusion for a critical section in user code. Value of semaphore s is 1 when it is in the unlocked state and value of s is equal to zero when the semaphore is locked or in the critical region of the code.

There are two operations associated with the binary semaphore to check whether the semaphore is in the locked or unlocked state.



1. P(s)

It tests s; if positive, resets s to 0 and proceed; otherwise, put the executing process to the back of a waiting queue for s.

1. V(s)

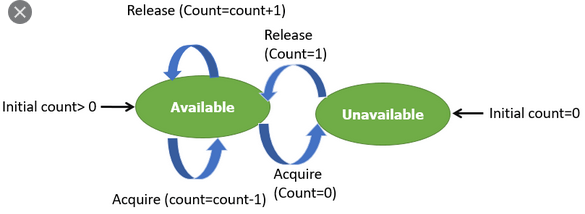
Set s to 1 and wake up a process in the waiting queue for s

**Counting Semaphore:**

Counting semaphore uses a count that helps task to be acquired or released numerous times. The binary semaphores are quite similar to counting semaphores, but their value is restricted to 0 and 1. A counting semaphore, s, is used for producer/consumer sync. Multiple processes can enter the critical region depending on the number of available resources, n.

n== the count of available resources

0==no resource (locking consumers out)



There are two operations associated with the binary semaphore to check whether the semaphore is in the locked or unlocked state.

1. P(s)

It tests s; if positive, decrements s and proceed. Otherwise, put the executing process to the back of a waiting queue for s

1. V(s)

It Increments s; wakes up a process, if any, in the waiting queue for s.

A Process P is executed inside the critical section is shown below:

***//some code***

***P(s);***

***//critical section(cs)***

***//exit from cs***

***V(s);***

***//remaining code***

There are two main operations of semaphores to enter and exit the critical region respectively. The functions of semaphores are:

* **wait( )**
* **post( )**

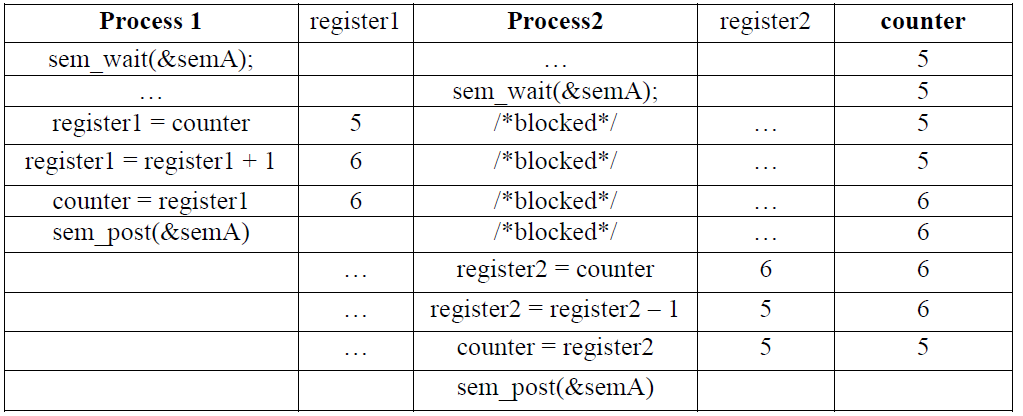
|  |  |
| --- | --- |
| Operations of Entry Section | Operations of Exit Section |
| wait () | post () |
| P () | V () |
| down () | up () |

The operations “wait (), p (), or down ()” are used to decrement the value of the semaphore variable from one to zero so that semaphore enter in the lock state and for it to enter the critical region and “post (), v (), or up ()” operations are used to increment the value of the semaphore from zero to one so that semaphore enter in the unlock state to exit from the critical section of the program.

In LINUX sem\_wait( ) and sem\_post( ) are used for the process to enter and exit from the critical region respectively. It is represented in the table below that how does the wait( ) and post( ) command works.

|  |  |
| --- | --- |
| **Code of Entry Section** | **Code of Exit Section** |
| **// *wait () function for counting semaphore***  wait (semaphore s)  {  s.value = s.value-1;  if(s.value < = 0)  {  // Put process in suspended list  block();  }  else  {  // add the process to critical section  }  } | **// *wait () function for binary semaphore***  wait (semaphore s)  {  if(s.value = = 1)  {  s.value = 0;  // add the process to critical section  }  else  {  // block the process and place in the  // in the suspended list  block();  }  } |
| **// *post () function for counting semaphore***  post (semaphore s)  {  s.value = s.value+1;  if (s.value < = 0)  {  // remove process p from queue  wakeup(p);  }  } | **// *post () function for binary semaphore***  post (semaphore s)  {  if (s.value = = 0)  {  // remove process p from queue  s.value = 1;  wakeup(p);  }  } |

An example to resolve race condition using binary semaphores is as follows:



Here, process 1 enters the critical region and process 2 is blocked until process 1 leaves the critical region and hence, after the process 1 leaves the critical region as indicated by the sem\_post( ) command process 2 enters the critical region and therefore, we get the accurate value of counter which is 5.

Here, I used this concept of semaphore in process synchronization to implement the Dining Philosopher’s problem as an example to further enhance my skill in using semaphores to avoid race conditions.

**Dining Philosopher’s Problem**

The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pick up the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.

Here, forks are the resource rather than the noodles hence, both the left and right forks should be available for the philosopher to eat food. Here, I implemented the following pseudo code for each of the process/philosopher:

*void philosopher(void)*

*{*

*while(TRUE)*

*{*

*thinking ();*

*take\_fork (i);*

*take\_fork ((i+1)%N);*

*eating();*

*put\_fork (i);*

*put\_fork ((i+1)%N);*

*}*

*}*

Here, I implemented the dining philosopher’s problem for five philosophers.

**Rules:**

* Philosopher should pick left fork first followed by right fork.
* Philosopher should put left fork first followed by right fork.

**Problem:**

If more than one philosopher wants to eat, race condition may occur.

**Solution using semaphore:**

Size of the array of the semaphores is equal to the number of forks. In our case size of the semaphore array is 5 ~ sem[5].

Semaphores are used to handle forks. According to the rules mentioned:

|  |  |  |
| --- | --- | --- |
| **Process** | **Left fork** | **Right fork** |
| **Process 0** | S[0] | S[1] |
| **Process 1** | S[1] | S[2] |
| **Process 2** | S[2] | S[3] |
| **Process 3** | S[3] | S[4] |
| **Process 4** | S[4] | S[0] |

p0 p1 p2 p3 p4

/ \ / \ / \ / \ / \

f0 f1 f1 f2 f2 f3 f3 f4 f4 f0

Process/ Philosopher 0 requires fork 0 and 1 for it to enter in the critical region/eat. Here, fork maps onto the semaphores.

S0 1 F0

S1 1 F1

S2 1 F2

S3 1 F3

S4 1 F4

When philosopher 0 wants to eat it’s left and right fork which maps onto semaphores are semaphore 0 and semaphore 1. Here, N is the total number of philosophers 5 in our case.

P(0) ->s(0)

p (0) -> s(0+1 % 5)= s(1)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Philosopher** | **p0** | **p1** | **p2** | **p3** | **p4** |
| **State** | Eating | Thinking | thinking | thinking | Thinking |

S0 0 F0

S1 0 F1

S2 1 F2

S3 1 F3

S4 1 F4

Semaphore 0 and 1 are in locked state.

Now, process 1 wants to eat and the required forks or semahores for it:

P(1) ->s(1)

p (1) -> s(0+2 % 5)= s(2)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Philosopher** | **p0** | **p1** | **p2** | **p3** | **p4** |
| **State** | Eating | Blocked | thinking | thinking | Thinking |

S0 0 F0

S1 0 F1

S2 0 F2

S3 1 F3

S4 1 F4

Semaphore 0,1 and 2 are in locked state. Here, process 1 has took his left fork but cannot take his right fork and hence, goes to the waiting queue.

Now, process 0 finished eating and hence, semaphore 0 and 1 are again in the unlock state after the post command is executed.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Philosopher** | **p0** | **p1** | **p2** | **p3** | **p4** |
| **State** | Thinking | Blocked | thinking | thinking | Thinking |

S0 1 F0

S1 1 F1

S2 0 F2

S3 1 F3

S4 1 F4

Now, semaphore 1 again goes to the lock state as the left fork of philosopher 1 is taken by the philosopher and it moves from blocked to eating state as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Philosopher** | **p0** | **p1** | **p2** | **p3** | **p4** |
| **State** | Thinking | eating | thinking | thinking | Thinking |

S0 1 F0

S1 1 F1

S2 0 F2

S3 1 F3

S4 1 F4

The following algorithm is applied while solving the dining philosopher’s problem using semaphores:

*void philosopher(void)*

*{*

*while(TRUE)*

*{*

*thinking ();*

*take\_fork (Si);*

*take\_fork (S(i+1)%N);*

*eating();*

*put\_fork (Si);*

*put\_fork (S(i+1)%N);*

*}*

*}*

Here, deadlock condition may arise depending on the scheduling algorithm. Deadlock condition occurs if all the processes occupy there left fork and therefore, not a single process can go into the eat condition. To remove the deadlock last philosopher must pick his right fork first followed by the left fork. In our case last philosopher is philosopher 4 and hence, philosopher 4 picks 1 its right fork first (Semaphore 0) followed by its left fork (Semaphore 4) as follows:

|  |  |  |
| --- | --- | --- |
| **Process** | **Fork** | **Fork** |
| **Process 0** | S[0] | S[1] |
| **Process 1** | S[1] | S[2] |
| **Process 2** | S[2] | S[3] |
| **Process 3** | S[3] | S[4] |
| **Process 4** | S[0] (Right fork) | S[4] (Left fork) |

Here, I implemented Dining Philosopher’s problem using both threads and processes. The solution developed using threads was a single terminal solution whereas, solution developed using processes was a multiple terminal solution.

***Explanation of implementing solution using threads:***

1. Start.

2. Declare and initialize the thread variables (philosophers) as required.

3. Declare and initialize the mutex variables (forks) as required.

4. Create the threads representing philosophers.

5. Wait until the threads finish execution.

6. Stop.

In the thread function the important task is to handle two conditions , first if the count of the philosopher which comes in is less than the total philosopher , then there is no issue we can simply get it into the thinking state and then it takes the left fork than right fork and then it can eat and free the forks by put functions , but the second one is the most important if the count is equal to the total philosophers then it should take the right fork first followed by the left fork as stated in the code below.

***Explanation of implementing solution using processes:***

1. Create a number of shared memories using the shmget function in the server code depending upon the number of philosopher’s. Make a number of pointer to mutex variables of sem\_t data type depending upon on the number of philosophers.
2. Attach the shared memory using the shmat and typecast its datatype to sem\_t and make it equal to the pointer to mutex variables.
3. Initialize the semaphores to zero and go to sleep for 45 seconds so that all of the philosopher’s eat/go to the critical region and hence, detach the shared memory and destroy the shared memory.
4. In the client code receive the shared memory produced in the server code by using the key=0 in the shmget function.
5. Enter the philosopher which wants to eat.
6. Use inter-process communication using unnamed pipes to read from a file whether the process have already eaten food. If the process had already eaten food leave from the process and return that the specific philosopher had already eaten food.
7. If all the philosophers had eaten food then return that all the philosophers have eaten food and exit from the code.
8. If the philosopher entered is less than the number of total philosopher’s perform the usual method of acquiring forks by picking the left fork followed by the right fork.
9. If the philosopher entered is equal to the number of total philosopher’s deadlock condition arises and hence, pick the right fork followed by the left fork.
10. Make sure you write the philosopher in the file when it’s eating food to determine that the specific philosopher have already eaten food.
11. Detach the shared memory of the client.
12. Run the server code followed by the client code.
13. Here, I used a client server model because we do not want that every time I run the code new shared memories are produced and hence, we are using the shared memory from the server side.

**Pros and cons of solution using semaphores :**

**Advantages :**

The basic advantage of using the semaphores is that there is no spinning , the CPU is not idle and it is using the resources every time and the other processes are in waiting state. The semaphores are machine independent; it allows the more flexible use of resource because multiple threads can enter the critical area.

**Disadvantages:**

The basic disadvantage is that we have to take care of the order of the functions and moreover we have to exclude the issue of deadlock and exclusion should be followed carefully. In large codes it is sometimes not possible to use semaphores. Then there is a queue like situation which can create issue if a process which is in the last needs to be processed earlier like in this issue that the last philosopher may sometimes need to eat first but it is processed in the end. First, if there are philosophers in your operating system, it’s probably not very efficient. Philosophers are not known for executing quickly. Quite the opposite in fact. And if you have forks in your CPU, then something is really, really wrong.

**IMPROVEMENTS:**

We can use monitor base solution which is entirely deadlock free solution for this problem.

**References:**

1. https://www.techopedia.com/definition/13390/synchronization-dot-net
2. <https://study.com/academy/lesson/process-synchronization-in-operating-systems-definition-mechanisms.html>
3. <https://prepinsta.com/operating-systems/process-synchronization/>