Synchronization

A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes may either directly share a logical address space (that is, both code and data), or be allowed to share data only through files or messages. Concurrent access to shared data may result in data inconsistency. We discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

Producer Consumer Problem

- A producer process produces information that is consumed by a consumer process. Example, a complier may produce assembly code which is consumed by an assembler.
- One solution to the producer – consumer problem uses shared memory.
- To allow producer & consumer process to run concurrently, we must have a buffer that reside in a region of shared memory.
- A producer can produce one time while the consumer is consuming another item. The producer & consumer must be synchronized so that the consumer does not try to consume an item that has not yet been produced.
- Two types of buffers are used:

1- Unbounded Buffer-

- No practical limit on the size of the buffer.
- The consumer may have to wait for new items, but the producer can always produces new items.

2- Bounded Buffer-

- Assumes a fixed buffer size.
- The consumer must wait if buffer is empty & the producer must wait if the buffer is full.

```
# define BUFFER_SIZE 10

typedef struct
{
        _ _ _ _
}item;
item buffer [BUFFER_SIZE];
int in =0;   // in points to next free position in buffer
int out =0;  // out points to first full position in buffer
```

- The buffer is empty when in ==out;
- The buffer is full when ((in +1) % BUFFER_SIZE)==out.
- The procedure process has a local variable *nextProduced* in which the new item to be produced is stored.
- The consumer process has a local variable *nextConsumed* in which the item to be consumed is stored.

- *Counter* is incremented every time we add a new item to the buffer & decremented every time we remove one item from the buffer

Code for Producer Process

```
while (true)
{
        /* produce an item is nextProduced*/
        while (counter== BUFFER_SIZE)
                ; /* do nothing */
        buffer [in] = nextProduced;
        in= (in+1) % BUFFER_SIZE;
        counter++;
}
```

Code for Consumer Process

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

- The statement counter ++ & counter- - may be implemented in machine language as

counter ++:

$register_1$ = counter
$register_1$= $register_1$ + 1
counter= $register_1$

counter--:

$register_2$ =counter
$register_2$= $register_2$-1
counter= $register_2$

- The $register_1$ & $register_2$ are local CPU register. Even though $regiter_1$ & $register_2$ may be the same physical register.
- These two processes will result correctly when they execute one by one. But they may not function correctly when executed concurrently.

Example:

set counter=5

T_0: producer execute $register_1$= counter	[$register_1$=5]
T_1: producer execute $register_1$=$register_1$+1	[$register_1$=6]
T_2: consumer execute $register_2$= counter	[$register_2$=5]
T_3: consumer execute $register_2$= $register_2$-1	[$register_2$=4]
T_4: producer execute counter =$register_1$	[counter=6]
T_5: consumer execute counter= $register_2$	[counter=4]

- This interleaving gives a wrong output as counter=4, whereas the correct output is counter=5. This incorrect output has generated because we allow both the process to manipulate the global variable *counter* concurrently. A situation like this is called race condition.

Race condition: A situation where several processes access & manipulate the same data concurrently & the final value of the shared data depends upon which process finishes last. To prevent race condition concurrent process must be synchronized.

Example of race condition: A kernel data structure that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed. If two processes were to open files simultaneously, the separate updates to this list could result in a race condition.

Questions asked in semester exam:

Question: State the Producer-consumer problem. Give a solution to the solution using semaphores.

[2016-2017][10 Marks]

Question: Explain the principle of concurrency.

[2014-2015][5 Marks]

Question: What is Bounded Buffer problem? Discuss briefly with the help of Producer and Consumer Process.

[2014-2015] [10 Marks]

Question: State and describe the Producer-Consumer problem with its suitable solution.

[2013-2014] [5 Marks]

Question: Discuss message-passing system. Explain how message passing can be used to solve to buffer Producer/Consumer problem with infinite buffer.

[2012-2013] [10 Marks]

Question: State the finite buffer Producer-Consumer Problem. Give solution of the problem using semaphores.

[2011-2012] [10 Marks]

Question: Write an algorithm to explain the producer/consumer using semaphores.

[2010-2011] [10 Marks]

Question: How concurrency problems are solved with Producer and Consumer problem?

[2009-2010] [5 Marks]

Question: Define a critical section problem and its solution by using Semaphore. Use this approach to solve Producer/Consumer problem.

[2006-2007] [5 Marks]