**Assignment 3 Report**

**How exactly synchronization is achieved using semaphore in our assignment?**

**Ans:** Two semaphores are used – produced and consumed which works as follows –

1. Produced and consumed semaphores are initialized to ‘0’ and ‘1’ respectively.
2. Now, both producer and consumer start executing in separated processes independently.
3. When consumer process runs for the first time, it calls wait() on produced semaphore. Decrementing the value from ‘0’ to ‘-1’. Since, this is a value below 0, the consumer process is put to waiting state by XINU.
4. When producer runs for first time, it calls wait() on consumed semaphore, decrementing its value from ‘1’ to ‘0’. Since this is a non-negative number, the producer can produce the value of n by incrementing it and printing it.

Steps 1-4 can be summarized using the following table: Fig 1.1

|  |  |  |
| --- | --- | --- |
| **Semaphore** | **consumed** | **produced** |
| **Initial Value** | consumed = 1 | produced = 0 |
| **After Wait()** | consumed = 0 | produced = -1 |
| **Used in Process** | Producer | Consumer |
| **Process State** | **Running** | **Blocked** |

1. After producer has produced a value, it calls signal() on produced semaphore. Thus the value of produced semaphore changes from ‘-1’ to ‘0’.
2. Producer again goes to produce a value and calls a wait() on consumed, thus the value of consumed is decremented from ‘0’ to –‘1’. Now the producer process gets blocked as the value of consumed semaphore is negative.
3. Since the value of produced semaphore is non-negative ‘0’ (after step 5), the consumer process is not blocked anymore and can consume the value of n and print it.

The state of the system (steps 5- 7) can be summarized using the following table Fig 1.2:

|  |  |  |
| --- | --- | --- |
| **Semaphore** | **consumed** | **produced** |
| **Initial Value** | consumed = 0 | produced = -1 |
| **After Signal(produced)** | consumed = 0 | produced = **0** |
| **After wait(consumed)** | consumed = **-1** | produced = 0 |
| **Used in Process** | Producer | Consumer |
| **Process State** | **Blocked** | **Running** |

1. After consuming the value of n, the consumer process calls the signal() on consumed semaphore. Thus the value of consumed semaphore is incremented to ‘0’ from ‘-1’.
2. The consumer process calls a wait() on produced semaphore once again making its value -1. Thus blocking the execution of the process until the value of produced semaphore is incremented again by the producer process.
3. As before, since the value of consumed semaphore is now non-negative ‘0’ (after step 8.), the process starts its execution and produces the value of n.

Steps 8 – 10 can be summarized as follows Fig 1.3:

|  |  |  |
| --- | --- | --- |
| **Semaphore** | **consumed** | **produced** |
| **Initial Value** | consumed = -1 | produced = 0 |
| **After Signal(consumed)** | consumed = **0** | produced = 0 |
| **After wait(produced)** | consumed = 0 | produced = **-1** |
| **Used in Process** | Producer | Consumer |
| **Process State** | **Running** | **Blocked** |

1. The state of producer and consumer process in Fig 1.3 are now similar to that in Fig 1.1. Thus, the two processes will re-run in a similar synchronized way as described above until the maximum value of n is produced and consumed.

**Can the above synchronization be achieved with just one semaphore? Why or why not?**

No, the above synchronization cannot be achieved with one semaphore.

Here’s a pseudo code for implementing synchronization using one semaphore - testSem:

**Produce.c**

1. int i;

2. for(i = 1; i <= count; i++)

3. {

4. wait(testSem);

5. n = i;

6. printf("produced: %d \n",n);

7. signal(testSem);

8. }

**Consume.c**

1.while (1)

2. {

3. wait(testSem);

4. printf("consumed: %d \n",n);

5. if ( n == count){

6. break;

7. }

8. signal(testSem);

9. }

As we can see, in the above pseudo code, there is just one semaphore “testSem” that is being used in both the processes to synchronize the production and consumption of ‘n’.

1. When both processes, producer and consumer start together, they would call wait() on the “testSem” semaphore.
2. In case the initial value of testSem is 1, after the wait system call, the testSem would decremented to 0, thus the calling process

**Source Code Changes:**

**prodcons.h**

#include <xinu.h>

#include <stddef.h>

#include <stdio.h>

/\*Global variable for producer consumer\*/

extern int n; /\*this is just declaration\*/

extern sid32 produced, consumed;

/\*function Prototype\*/

void consumer(int count, sid32 consumed, sid32 produced);

void producer(int count, sid32 consumed, sid32 produced);

**produce.c**

#include <prodcons.h>

void producer(int count, sid32 consumed, sid32 produced)

{

//Code to produce values less than equal to count,

int i;

for(i = 1; i <= count; i++)

{

wait(consumed);

n = i;

printf("produced: %d \n",n);

signal(produced);

}

}

**consume.c**

#include <prodcons.h>

void consumer(int count, sid32 consumed, sid32 produced)

{

while (1){

wait(produced);

printf("consumed: %d \n",n);

if ( n == count){

break;

}

signal(consumed);

}

}

**xsh\_prodcons.c**

#include <prodcons.h>

#include <ctype.h>

int n ; //Definition for global variable 'n'

/\*Now global variable n will be on Heap so it is accessible all the processes i.e. consume and produce\*/

sid32 produced,consumed;

shellcmd xsh\_prodcons(int nargs, char \*args[])

{

//Argument verifications and validations

int count = 2000; //local varible to hold count

int i = 0;

consumed = semcreate(1);

produced = semcreate(0);

// Initialise the value of n to 0, since this is an extern variable, it may start with the previous value

/\* Output info for '--help' argument \*/

if (nargs == 2 && strncmp(args[1], "--help", 7) == 0)

{

printf("Usage: %s\n\n", args[0]);

printf("Description:\n");

printf("\tProducer Consumer Example using semaphore synchronization.\n");

printf("Options (one per invocation):\n");

printf("\t--help\tdisplay this help and exit\n");

return 0;

}

/\* Check argument count \*/

/\* If argument count is greater than 2, then there are too many arguments\*/

if (nargs > 2)

{

fprintf(stderr, "%s: too many arguments\n", args[0]);

return 1;

}

/\* If argument count is equal to 2, then assign args[1] to count variable \*/

if (nargs == 2)

{

// Parse through the array of parameters and return 1 if there is a character other than a number.

for(i = 0; args[1][i] != '\0'; i++ )

{

if (isdigit(args[1][i]) == 0)

{

fprintf(stderr, "%s: input parameter should be an integer.\n", args[0]);

return 1;

}

}

// Else, it can be safely converted to a number.

count = atoi(args[1]);

}

if(count == 0){

fprintf(stderr, "Count should be greater than zero.\n");

return 1;

}

//create the process producer and consumer and put them in ready queue.

//Look at the definitions of function create and resume in exinu/system folder for reference.

resume( create(producer, 1024, 20, "producer", 3, count, consumed, produced) );

resume( create(consumer, 1024, 20, "consumer", 3, count, consumed, produced) );

}

**Function Descriptions:**

* 1. resume():
  2. create(function, size, priority, name, args, hhkhkkj):

The create system call is used to create a new process that will execute instructions written in the ‘function’ specified in the first argument.

Following is the argument description -

* size specifies the stack size, generally in bytes.
* Priority specifies the priority of the process.
* Name specifies identifying name for the new process.
* args specifies the number of arguments required for ‘function’.
* hhkhkkj specifies the actual parameter that is passed to the process. i.e. count
* This function returns the pid of the created process.
* The created process is in the suspended state.
* The resume function accepts the pid of the process and resumes the execution of the process.
* void producer(int count)
  1. void producer(int count, sid32 consumed, sid32 produced)
* This is the first method passed to the create system call.
* The producer method accepts 3 arguments- count, consumed semaphore and produced semaphore. It puts a wait on consumed sempahore, assigns incremental values to n starting from 1 with step count as 1 and then signals produced semaphore.
  1. void consumer(int count, sid32 consumed, sid32 produced)
* This is the second method that is passed to the system call.
* The consumer method accepts 3 arguments – count, consumed semaphore and produced semaphore. It puts a wait on produced semaphore, prints values of ‘n’ until it reaches its maximum limit i.e. count and signals consumed semaphore.
  1. semcreate(int semValue):

**Contributions –**

**Akshay Kamath (akkamath)**

**Sameedha Bairagi (sbairagi)**