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**

5. After producer has produced a value, it calls signal() on produced semaphore. Thus the value of produced semaphore changes from ‘-1’ to ‘0’.
6. 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.
7. 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**

8. 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'.
9. 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.
10. 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**

11. 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.

Explaination:

Approach 1: Here's a pseudo code for implementing synchronization using one semaphore - mutex:

Produce.c

```
int i;
for(i = 1; i <= count; i++)
{
        wait(mutex);
        n = i;
        printf("produced: %d \n",n);
        signal(mutex);
}
```

Consume.c

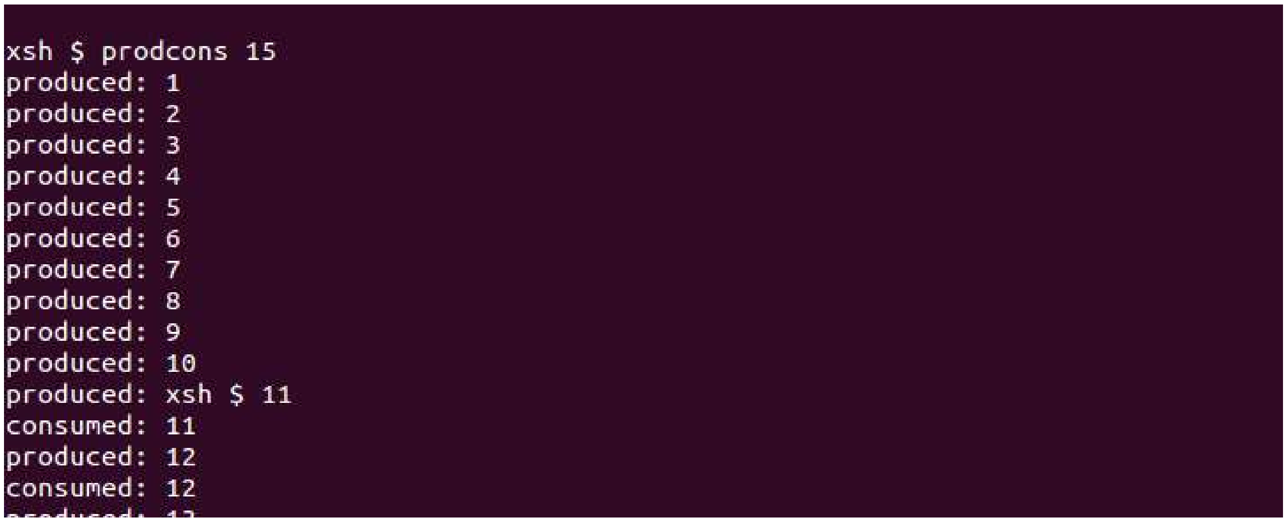
```
while (1)
{
        wait(mutex);
        printf("consumed: %d \n",n);
        if ( n == count){
                break;
        }
        signal(mutex);
}
```

As we can see, in the above pseudo code, there is just one semaphore "mutex" 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 "mutex" semaphore.
2. In case the initial value of mutex is initialized to 0, in case producer calls the wait() first on mutex, the value of the mutex would be decremented to -1 and if the consumer process calls wait() after producer, the value could be further decremented to -2. Thus the value becomes less than 0 and both the processes will be blocked and since there is none of the processes can call the signal anymore it

will be a deadlock. The processes will not be able to execute any further instructions as they both wait on the semaphore to be available again.

3. **Approach 2:** In case the value of mutex is initialized to 1. Any of the process may call the wait system call first and the first process to call the wait() will be able to execute the code in its critical section. Once, the execution of the critical section is completed and the signal() is called by the active process, both processes will again contend for the semaphore access.
4. Which process calls the wait() first, then depends on the scheduler and cannot be determined.
5. Hence, although the access to the shared resource is atomic i.e. only one process uses the shared resource ‘n’ using the mutex semaphore, the synchronicity of the two processes may not be achieved in the orderly way as it would be if two semaphores are used.
6. Here’s a screen shot of the output of processes using single semaphore for second case:



```
xsh $ prodcons 15
produced: 1
produced: 2
produced: 3
produced: 4
produced: 5
produced: 6
produced: 7
produced: 8
produced: 9
produced: 10
produced: xsh $ 11
consumed: 11
produced: 12
consumed: 12
```

As observed, there is no garbage character between produced and consumed due to atomicity of operation, however, the order of which process gets to execute in critical section cannot be determined and is dependent on the scheduler.

Approach 3: An alternative approach to use the single semaphore would be to use signaling at the end of producer loop and waiting at the beginning of consumer loop. Following pseudo code can be used to demonstrate the same:

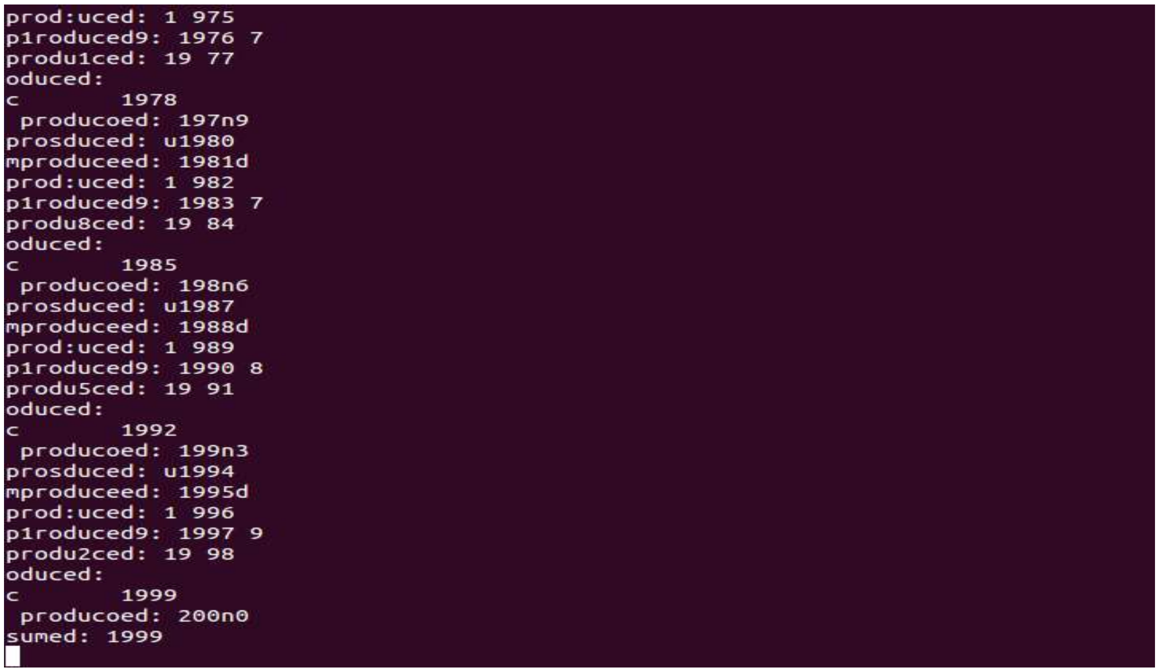
Produce.c:

```
1.    loop (i to n)
2.    n = i;
3.    printf("produced: %d \n",n);
4.    signal(consumed);
5.    loop end
```

Consume.c

```
1.    loop (i to n)
2.    wait(consumed);
3.    printf("consumed: %d \n",n);
4.    loop end
```

In this case, although the consumer will only consume the values after values are produced by the producer, there is no constraint or lock to stop the producer from producing n values at a stretch.
This will also result in printing the formatted output of producer and consumer to the screen in an interleaved manner and it may look like junk characters are produced.
Below is a screen print of the output of the above pseudo code:


```
prod:uced: 1 975
p1roduced9: 1976 7
produ1ced: 19 77
oduced:
c       1978
 producoed: 197n9
prosduced: u1980
mproduceed: 1981d
prod:uced: 1 982
p1roduced9: 1983 7
produ8ced: 19 84
oduced:
c       1985
 producoed: 198n6
prosduced: u1987
mproduceed: 1988d
prod:uced: 1 989
p1roduced9: 1990 8
produ5ced: 19 91
oduced:
c       1992
 producoed: 199n3
prosduced: u1994
mproduceed: 1995d
prod:uced: 1 996
p1roduced9: 1997 9
produ2ced: 19 98
oduced:
c       1999
 producoed: 200n0
sumed: 1999
```

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. create(function, size, priority, name, count, varArgs):

   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.
   - count specifies the number of arguments required for ‘function’.
   - varArgs specifies variable number of arguments which are actual parameters for function. The total arguments will be equal to count.

   This function returns the pid of the created process which is in the suspended state.
2. resume(pid):
   - The resume function accepts the process id of the process as an argument and resumes its execution (provided the process is in suspended state).
3. void producer(count, consumed, 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 semaphore, assigns incremental values to n starting from 1 with step count as 1 and then signals produced semaphore.
4. void consumer(count, consumed, produced)
   - This is the second method that is passed to the “create” 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.
5. semcreate(value):

   This is a system call which takes desired initial value as an argument and returns an integer identifier for the new semaphore.

6. Semdelete(sid32 semname)

   Function semdelete reverses the actions of semcreate. Semdelete takes the index of a semaphore as an argument and releases the semaphore table entry for subsequent use.

Contributions –

Akshay Kamath (akkamath)

1. Implemented semaphore changes in consume.c
2. Implementing changes for help and error handling.
3. Implementing and testing 3 variations of semaphore synchronization implementation in answer 2.
4. Fixing code issues and testing.
5. Implemented semdelete in producer.c and consumer.c
6. Assignment 3 report changes.

Sameedha Bairagi (sbairagi)

1. Implemented the semaphore changes in produce.c
2. Implemented method signature changes in main.c
3. Implemented method changes in header files.
4. Fixing code issues and testing
5. Assignment 3 report changes.