P536 : Assignment 2

Report

Group Members:
Rajasi Rane (Username : rrane)
Ankita Shetty (Username : shettya)

Questions:

Q1. How exactly is synchronization achieved using semaphore in our assignment?

The main task of our assignment is to solve the producer consumer problem using semaphores. In order to do so, we create two semaphores using semcreate system call as follows

consumed = semcreate(1)
produced = semcreate(0)

It takes an argument i.e. count, which gets assigned to the semaphore. In addition to this count, the semaphore also maintains a list to store all the blocked processes. The semcreate system call returns an ID of the semaphore created, which we save in '*produced*' and '*consumed*'.

On calling '*prodcons <<count>>*', it calls the producer process with count as an argument.

Producer Process:

```
for(i=1, i<=count, i++)
{
        wait(consumed);
        n=i;    //produce n
        signal(produced);
}
```

Consumer Process:

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

1. When the *producer process* begins execution, '*wait*' system call is executed on semaphore '*consumed*'. It decrements the count value and checks if it is a positive or a negative value. Since on decrementing, the value is now 0, the '*wait'* system call does not block the current *producer process*, thus producing 'n'.
2. Once the *producer produces* a value, it signals the *consumer process* to consume that value using the *'signal'* system call on the semaphore *'produced'*. It increments the count of *'produced'* and will also check for any waiting processes in the *'produced'* semaphores' process list. If a waiting process is found, it puts that process in the ready queue.

In this case since there is no process waiting, it increments the count and executes the *'wait'* system call for the next iteration, thus decrementing value of *'consumed'* to -1. Since the value is now negative, the *'wait'* system call blocks the calling process i.e. the *producer process* by doing the following:

- It adds the process to the semaphores' (*produced*) process list
- It changes the state of the process to *PR_WAIT*
- It calls *resched* that switches the control to a ready process

3. Now the next ready process i.e. *consumer process* executes.
   Here the *'wait'* system call executes for *'produced'* and decrements its value to 0. Again the calling process is not blocked and the *consumer* consumes the produced value. Then it calls the *'signal'* system call on *'consumed'*, which will:
   - Increment the value of *'consumed'*
   - It will look up for any waiting processes in the process list of *'consumed'* and put them in the ready queue if there are any.

   In this case, the *producer process* will resume its execution, since it was in the process list of *'consumed'*.

Producer and consumer processes continue execution in this fashion until it reaches the maximum value given by 'count'. As it can be seen in the above-mentioned execution of the program; whenever a value is produced, it gets consumed before the next value can be produced. This is how synchronization is achieved using semaphores in our program.

Q2. Can the above synchronization be achieved with just one semaphore? Why or why not?

According to our understanding, the synchronization needed in the producer consumer problem cannot be achieved using a single semaphore. However, it may work in some cases where the processes are not affected by other processes' execution.

- Consider a scenario where multiple processes are writing into a shared buffer. In this case mutual exclusion can be achieved using a single semaphore.
  Here each process will call *'wait'* on the semaphore before executing the critical section and call *'signal'* on the semaphore after the execution of the critical section is complete. This is how synchronization can be achieved, for this scenario, using a single semaphore.

In the producer consumer problem, the producer produces multiple values and the consumer must consume all the produced values as and when they are produced. If we try to use a single semaphore for this problem, i.e. by calling *'wait'* on the semaphore before executing the critical section and calling 'signal' on the same semaphore after the critical section, there are two possible cases:

1. When we set the count of semaphore to 0
   In this case, producer calls *'wait'* on this semaphore decrementing its value, which now becomes negative thus blocking the producer. The control is then transferred to the consumer, where it again calls *'wait'* on the same semaphore

further decrementing its value. As there is no *'signal'* call on the semaphore both processes remain in the waiting state.

2. When we set the count of semaphore to 1
   In this case, even after calling *'wait'* on the semaphore, its count remains non-negative and thus the producer will continue producing values till it reaches the maximum count, in the iteration, since there are no waiting processes. Once all the values are produced, the control switches to the next process i.e. 'consumer'. This is nothing but 'busy waiting' where the producer will produce all values and the consumer will consume only the last value.

Thus, there is no synchronization between the produced & consumed values.

Files created:

- prodcons.h
- xsh_prodcons.c
- producer.c
- consumer.c

Tasks performed by:

- **Rajasi Rane**:
  Updated prodcons.h and xsh_prodcons.c files.

- **Ankita Shetty**:
  Modified the producer and consumer function.

Both members worked on creating this report and also modified the shprototypes file under include directory and shell.c under shell directory to add 'prodcons' as a command. The team also tried to implement synchronization using a single semaphore, which needs a buffer to work as expected.

References:

Operating System Design: The Xinu Approach, 2E