ADVANCED OPERATING SYSTEMS (FA15-BL-CSCI-P536-36187)

Assignment 3 - Producer Consumer using Semaphores

Pralhad Sapre & Srivatsan Iyer

How exactly is synchronization achieved using semaphore in our assignment?

In our assignment we have two semaphores defined as

```
sid32 produced, consumed;
```

Now notice the initialization of these two semaphores as

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

The reason why consumed is initialized to 1 is because the first statement of the producer's loop is to wait on the consumed semaphore. A value of 1 releases the wait and the production can begin. When the producer is done the produced semaphore is signalled using signal(). This releases the wait(produced) statement which is the first statement in the loop of the consumer. Thus a production opens the door for consumption and when the consumption is done a signal(consumed) opens the producer for production again.

This to and fro signalling ensures that every object produced is consumed before any other object is produced. We illustrate the code for producer and consumer below.

```
void consumer(int count) {
    for (;;) {
        wait(produced);
        kprintf("Consumed: %d\n", n);

        if(n==count) {
               signal(consumed);
               break;
        }

        signal(consumed);
    }
    semdelete(produced);
    semdelete(consumed);
}

void producer(int count) {
    int i = 0;
    for (i = 1; i<=count; i++) {
        wait(consumed);
        n = i;
        kprintf("Produced: %d\n", n);
        signal(produced);
    }
}
```

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

The above synchronization **can** be **partially achieved** using a single semaphore. The exact kind of synchronization (i.e. to have consumer wait until producer produces), however, can not be possible. In the below given process, the consumer relies on a flag variable, and as one can note below, the consumer might have to wake up periodically to check for presence of anything consumable.

We achieve it using a similar scheme of a variable which indicates whether the last produced value has been consumed. The overall template of the producer and consumer is similar in terms of semaphores surrounding the critical section. However both producer and consumer wait() and signal() a common semaphore. The logic lies in between these two primitives which checks

- The check: If its a producer, has the last generated value being consumed? If not just signal() the semaphore.
- If its a consumer, if a value has been generated consume it and flip the flag (variable) to enable to production logic to be executed. If you have reached the last value 'n', break the loop and delete the semaphore.

The check is done using a variable which is declared as "volatile" which forces the compiler to avoid any optimizations on the variable and compulsorily check the value of the variable each time.

Here is the core piece of code in consumer.c

```
#include <xinu.h>
#include <prodcons.h>

extern volatile int cur_value_consumed;

void consumer(int count) {
    for (;;) {
        wait(produced);
        if (cur_value_consumed == 0) {
                printf("Consumed: %d\n", n);
                cur_value_consumed = 1;
                if(n==count) {
                        signal(produced);
                        //deleting the semaphore after the last signal from producer
                        printf("Deleting the single semaphore");
                        semdelete(produced);

                        break;
                }
        }
        signal(produced);
    }
}
```

Here is the core piece of code in producer.c

```
#include <xinu.h>
#include <prodcons.h>

volatile int cur_value_consumed = -1;

void producer(int count) {
    cur_value_consumed = 1;
    int i = 0;
    for (i = 1; i<=count;) {
        wait(produced);
        if (cur_value_consumed == 1) {
                n = i;
                printf("Produced: %d\n", i);
                cur_value_consumed = 0;
                i++;
        }
        signal(produced);
    }
}
```

Tasks done by us and functions in the project

Principally we have modified these files

produce.c - to include the usage of semaphores with wait() and signal() system calls

consume.c - to include the wait(), signal() and semdelete() system calls

xsh_prodcons.c - it has the semcreate() primitive and the definition of produced and consumed semaphores of type “sid32”

prodcons.c - To have extern declarations of the produced, consumed semaphores

Tasks

Pralhad Sapre - Implementation using a single semaphore

Srivatsan Iyer - Implementation using two semaphores