	*Akademia Górniczo-Hutnicza w Krakowie*		
Faculty: ESA	Academic Year 2019/2020	Year of study	Fields of science:
Student name:	**Ioannis Manousaridis**		
Course:	**Real Time Operating Systems in Control Applications**		
Exercise nr:	**12P**		
Exercise subject:	**Exercise RTOS -12P. FreeRTOS – synchronization 2**		
Lecturer: dr inż. Grzegorz Wróbel			
Date: 00.00.2020			Grade:

1 Exercise purpose

The purpose of this exercise is to familiarize with the synchronization or protection of resources against simultaneous access of many processes which can be implemented using various tools: inter-task communication, semaphores, mutexes. In this lab the problem of producer-consumer is studied.

2 Assumptions / Theory

The Producer-Consumer problem is a classic problem this is used for multi-process synchronization i.e. synchronization between more than one processes.

In the producer-consumer problem, there is one Producer that is producing something and there is one Consumer that is consuming the products produced by the Producer. The producers and consumers share the same memory buffer that is of fixed-size.

The job of the Producer is to generate the data, put it into the buffer, and again start generating data. While the job of the Consumer is to consume the data from the buffer.

The following are the problems that might occur in the Producer-Consumer:

1. The producer should produce data only when the buffer is not full. If the buffer is full, then the producer shouldn't be allowed to put any data into the buffer.
2. The consumer should consume data only when the buffer is not empty. If the buffer is empty, then the consumer shouldn't be allowed to take any data from the buffer.
3. The producer and consumer should not access the buffer at the same time.

The above three problems can be solved with the help of semaphores. In the producer-consumer problem, three semaphore variables are used:

- Semaphore S: This semaphore variable is used to achieve mutual exclusion between processes. By using this variable, either Producer or Consumer will be allowed to use or access the shared buffer at a particular time. This variable is set to 1 initially.
- Semaphore E: This semaphore variable is used to define the empty space in the buffer. Initially, it is set to the whole space of the buffer i.e. "n" because the buffer is initially empty.
- Semaphore F: This semaphore variable is used to define the space that is filled by the producer. Initially, it is set to "0" because there is no space filled by the producer initially.

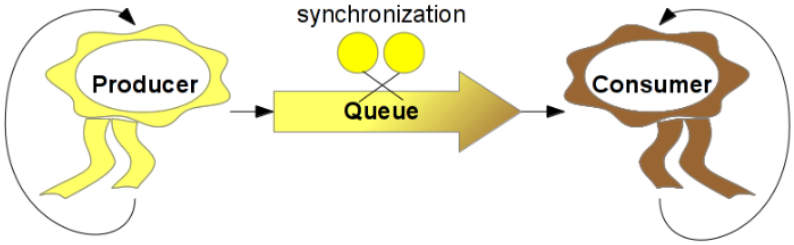

Figure 9: Producer-Consumer example

3 Description of implementation

There are many ways to solve this problem but in this case semaphores will be used. As it can be seen in figure 2, there are two blue boxes, two tasks the producer and the consumer. The producer generates random one digit number and place them into the buffer, if it is empty. To check if the buffer has an empty space a counting semaphore called full is used.

On the other side, the produce reads data from the buffer when he gets access to it if it is not empty. This procedure go on forever.

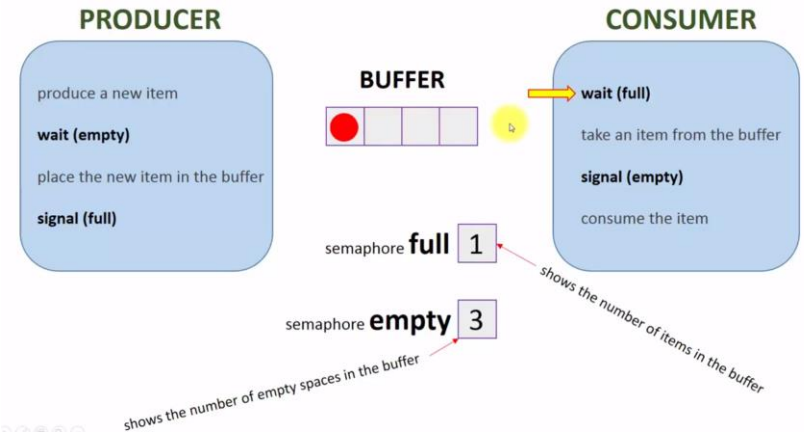

Figure 2: Producer – Consumer tasks communicating through a queue.

4 Code

Below the code is being explained.

```
#define N 5

#define PRODUCER_DELAY 1000
#define CONSUMER_DELAY 1000


xSemaphoreHandle empty, full;
xSemaphoreHandle mutex;
xTaskHandle  producerHandler;
xTaskHandle consumerHandler;
int buffer[N];
int in = 0, out = 0;
```

In the beginning we need to initialize the producer and consumer delays as well as the semaphores and the task handlers.

```
int main(int argc, char *argv[])
{
	printf("Author: Ioannis Manousaridis \n\n");
	full = xSemaphoreCreateCounting(N, 0);
	empty = xSemaphoreCreateCounting(N, N);
	mutex = xSemaphoreCreateMutex();
	if ((empty == NULL) || (full == NULL))
		printf("Error while creating counting Semaphores\n");
	else if (mutex == NULL)
		printf("Error while creating mutex\n");
	else
	{
		xTaskCreate(producer, "producer", 1000, NULL, tskIDLE_PRIORITY,
&producerHandler);
		xTaskCreate(consumer, "consumer", 1000, NULL, tskIDLE_PRIORITY,
&consumerHandler);

	}

	vTaskStartScheduler();

	/* The following line should never be reached */
	while (1)
	{
		printf("Insufficient heap memory!\n");
	};
	return 0;
}
```

In the main function are created the semaphores. One mutex semaphore and two counting semaphores, one for the full and one for the empty using the xSemaphoreCreateCounting. These semaphores are used for the access to the resources of the buffer as it was explained before.

Finally, in the end, we have the two functions tasks of the producer and the consumer which produce and consumes data from the queue while updating accordingly the semaphores' values.

```
void producer(void *pvParameters)
{
	int count = 0;
	int item;
	while (1)
	{
		item = rand() % 10;
		if (xSemaphoreTake(empty, PRODUCER_DELAY) != pdTRUE)
			printf("Producer unable to take empty, buffer is full\n");
		else if (xSemaphoreTake(mutex, PRODUCER_DELAY) != pdTRUE)
			printf("Producer unable to take mutex\n");
		else
		{
			printf("Producer sending #%d item %6d\n", count, item);
			buffer[in] = item;
			in = (in + 1) % N;
			count++;
			if (xSemaphoreGive(mutex) != pdTRUE)
				printf("Producer unable to give mutex\n");
			if (xSemaphoreGive(full) != pdTRUE)
				printf("Producer unable to give full\n");
			vTaskDelay(PRODUCER_DELAY);
		}

	}

}

void consumer(void *pvParameters)
{
	int item;
	while (1)
	{
		if (xSemaphoreTake(full, CONSUMER_DELAY) != pdTRUE)
			printf("Consumer unable to take full, buffer is empty\n");
		else if (xSemaphoreTake(mutex, CONSUMER_DELAY) != pdTRUE)
			printf("Consumer unable to take mutex\n");
		else
		{
			item = buffer[out];
			printf("Consumer received item: %6d\n", item);
			out = (out + 1) % N;
			if (xSemaphoreGive(mutex) != pdTRUE)
				printf("consumer unable to give mutex\n");
			if (xSemaphoreGive(empty) != pdTRUE)
				printf("Consumer unable to give empty\n");
			vTaskDelay(CONSUMER_DELAY);
		}
	}
}
```

5 Conclusions

You can see the results for the different problems below.

1. Speed of work for the producer and consumer is the same.

In this case the producer is placing an item in the queue and the consumer is taking after the item almost immediately. Consequently, the queue has only one item every time and never becomes full.

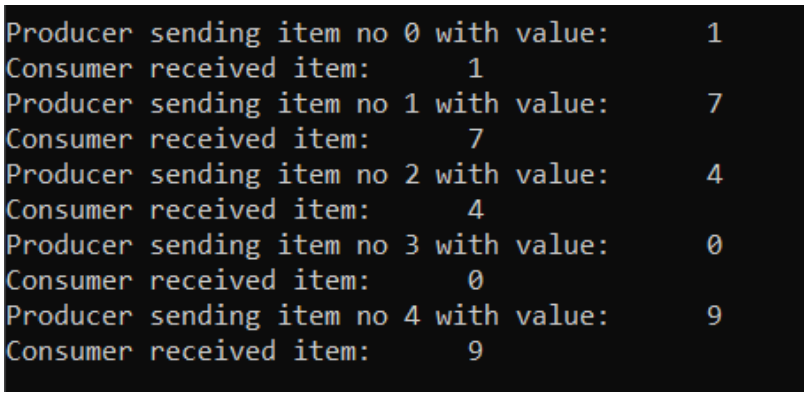

Figure 3. Results when producer and consumer have the same delay.

2. Producer is faster than consumer.

In this case the producer is placing an item in the queue and the consumer is taking after the item but much slower than the consumer. Sometimes it can be seen in figure 4 that the producer will place two items and the consumer will take only. If the programs run for a long time, eventually the queue will be filled.

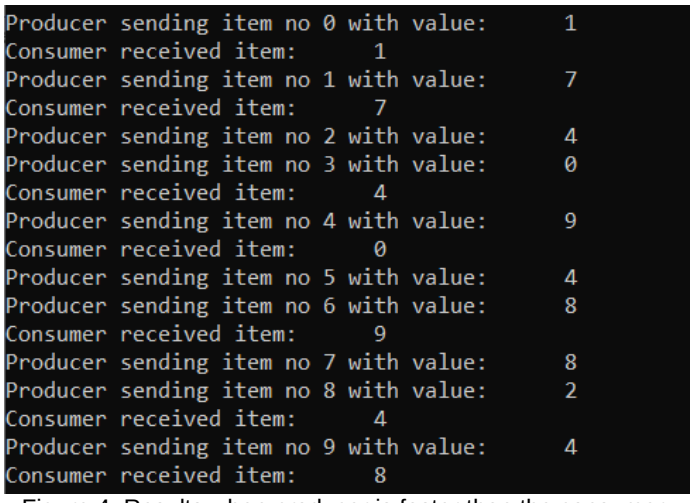

Figure 4. Results when producer is faster than the consumer.

3. The producer is slower than the consumer.

When the producer is slower than the consumer, every item that will be placed in the buffer it will be taken immediately from the consumer. In this case, again, the queue will have max of one item stored.

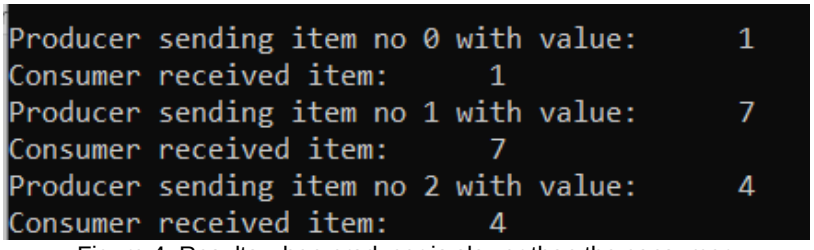

Figure 4. Results when producer is slower than the consumer.

To conclude, synchronizations is possible in the consumer-producer problem with the use of semaphores.