Assignment 3. Producer - Consumer Problem (Due: Monday, November 21, 2016)

In Section 5.7.1, we had presented a semaphore-based solution to the producer-consumer problem using a bounded buffer. In this project, we will design a programming solution to the bounded-buffer problem using the producer and consumer processes shown in Figures 5.9 and 5.10 (textbook). The solution presented in Section 5.7.1 uses three semaphores: `empty` and `full`, which count the number of empty and full slots in the buffer, and `mutex`, which is a binary (or mutual-exclusion) semaphore that projects the actual insertion or removal of items in the buffer. For this project, standard counting semaphores will be used for `empty` and `full`, and a mutex lock, rather than a binary semaphore, will be used to represent `mutex`. The producer and consumer - running as separate threads - will move items to and from a buffer that is synchronized with these `empty`, `full`, and `mutex` structures. You can solve the problem using either Pthreads or the Win32 API.

The Buffer

Internally, the buffer will consist of a fixed-size array of type `buffer_item` (which will be defined using a `typedef`). The array of `buffer_item` objects will be manipulated as a circular queue. The definition of `buffer_item`, along with the size of the buffer, can be stored in a header file such as the following:

```
/* buffer.h */
typedef int buffer_item;
#define BUFFER_SIZE 5
```

The buffer will be manipulated with two functions, `insert_item()` and `remove_item()`, which are called by the producer and consumer threads, respectively. A skeleton outlining these functions appears in Figure 1.

```
#include “buffer.h”

/* the buffer */
buffer_item buffer[BUFFER_SIZE];

int insert_item(buffer_item item) {
    /*insert item into buffer
     return 0 if successful, otherwise
     return -1 indicating an error condition */
}

int remove_item(buffer_item *item) {
    /* remove an object from buffer
     placing it in item
     return 0 if successful, otherwise
     return -1 indicating an error condition */
}
```

Figure 1. A skeleton program.

The `insert_item()` and `remove_item()` functions will synchronize the producer and consumer using the algorithms outlined in Figures 6.10 and 6.11 (textbook). The buffer will also require an initialization function that initializes the mutual-exclusion object `mutex` along with the `empty` and `full` semaphores.

The `main()` function will initialize the buffer and create the separate producer and consumer threads. Once it has created the producer and consumer threads, the `main()` function will sleep for a period of time and, upon awakening, will terminate the application. The `main()` function will be passed three parameters on the command line:

a. How long to sleep before terminating
b. The number of producer threads
c. The number of consumer threads

A skeleton for this function appears in Figure 2.

```
#include "buffer.h"

int main(int argc, char *argv[]) {
    /*1. Get command line arguments argv[1], argv[2], argv[3] */
    /*2. Initialize buffer */
    /*3. Create producer thread(s) */
    /*4. Create consumer thread(s) */
    /*5. Sleep */
    /*6. Exit */
}
```

Figure 2. A skeleton program

Producer and Consumer Threads

The producer thread will alternate between sleeping for a random period of time and inserting a random integer into the buffer. Random numbers will be produced using the `rand()` function, which producers random integers between `0` and `RAND_MAX`. The consumer will also sleep for a random period of time and, upon awakening, will attempt to remove an item from the buffer. An outline of the producer and consumer threads appears in Figure 3.

Pthreads Thread Creation

Creating threads using the Pthreads API is discussed in Chapter 4. Please refer to that chapter for specific instructions regarding creation of the producer and consumer using Pthreads.

```
#include <stdlib.h> /* required for rand() */
#include "buffer.h"

void *producer(void *param) {
   buffer_item item;

   while (TRUE) {
      /* sleep for a random period of time */
      sleep(...);
      /* generate a random number */
      item = rand();
      if (insert_item(item))
         fprintf("report error condition");
      else
         printf("producer produced %d\n", item);
   }
}

void *consumer(void *param) {
   buffer_item item;

   while (TRUE) {
      /* sleep for a random period of time */
      sleep(...);
      if (remove_item(&item))
         fprintf("report error condition");
      else
         printf("consumer consumed %d\n", item);
   }
}
```

Figure 3 An outline of the producer and consumer threads.

```
#include <pthread.h>
pthread_mutex_t mutex;

/* create the mutex lock */
pthread_mutex_init(&mutex, NULL);

/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/*** critical section ***/

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

Figure 4 Code sample.

Pthread Mutex Locks

The code sample depicted in Figure 6.30 (textbook) illustrates how mutex locks available in the Pthread API can be used to protect a critical section.

Pthread uses the `pthread_mutex_t` data type for mutex locks. A mutex is created with the `pthread_mutex_init(&mutex, NULL)` function, with the first parameter being a pointer to the mutex. By passing `NULL` as a second parameter, we initialize the mutex to its default attributes. The mutex is acquired and released with the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions. If the mutex lock is unavailable when `pthread_mutex_lock()` is invoked, the calling thread is blocked until the owner invokes `pthread_mutex_unlock()`. All mutex functions return a value of 0 with correct operation; if an error occurs, these functions return a nonzero error code.

Pthreads Semaphores

Pthreads provides two types of semaphores - named and unnamed. For this project, we use unnamed semaphores. The code below illustrates how a semaphore is created:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 5 */
sem_init(&sem, 0, 5);
```

The `sem_init()` creates and initializes a semaphore. This function is passed three parameters:

a. A pointer to the semaphore
b. A flag indicating the level of sharing
c. The semaphore's initial value

```
#include <semaphore.h>
sem_t mutex;

/* Create the semaphore */
sem_init(&mutex, 0, 1);

/* acquire the semaphore */
sem_wait(&mutex);

/*** critical section ***/

/* release the semaphore */
sem_post(&mutex);
```

Figure 5 AAA5.

In this example, by passing the flag 0, we are indicating that this semaphore can only be shared by threads belonging to the same process that created the semaphore. A nonzero value would allow other processes to access the semaphore as well. In this example, we initialize the semaphore to value 5.

In Section 5.6.1, we described the classical `wait()` and `signal()` semaphore operations. Pthreads names the `wait()` and `signal()` operations `sem_wait()` and `sem_post()`, respectively. The code example shown in Figure 5 creates a binary semaphore `mutex` with an initial value of 1 and illustrates its use in protecting in a critical section.