北京理工大學

操作系统课程设计

实验三、生产者消费者问题	Experiment 3: Producer-Consumer Problem

学院：计算机学院
专业：计算机科学与技术
学生姓名：夏奇拉
学号：1820171025
班级：07111705

Purpose

1 Windows and Linux systems;

2 Create 4 buffers, initially "-";

3 Create 3 producers. For each producer:

-Wait for a random period of time within 3 seconds (including 3 seconds), and randomly add an initial of your name (uppercase) to the buffer. For example, Wang Quanyu's initials are W, Q, Y, that is, from these three letters each time take one;

- If the buffer is full, wait for the consumer to take the letters before adding them;

- Repeat 4 times;

4. Create 4 consumers, each consumer

-Wait a random amount of time to read letters from the buffer

- If the buffer is empty, wait for the producer to add letters before reading

- Repeat 3 times

5. The content of each operation needs to be printed

- Producer print: Letter the producer writes to the buffer

-Consumer Print: Letters taken by the consumer

- Need to print buffer content

- According to the principle of first consumption of goods produced first

Problem Discussion

In an operating system a producer is a process which is able to produce data. A consumer is a process that is able to consume data produced by a producer. Both producer and consumer share a common memory buffer. This buffer is a place of a certain size in the memory of the system which is used for storage. The producer produces the data into the buffer and the consumer consumes the data from the buffer.

The producer-consumer problem examines issues that might arise between the producer and consumer and presents a solutions. These problems include the following:

- the producer process should not produce data when the buffer is full
- the consumer process should not consume data when the shared buffer is empty.
- The access to the shared buffer should be mutually exclusive i.e. at a time only one process should be able to access the shared buffer and make changes to it.

For consistent data synchronization between the producer and the consumer, these problems need to be resolved.

Solution Algorithm

To solve the producer-consumer problem three semaphores are used. Semaphores are variables used to indicate the number of resources available in the system at a particular time. They are used to achieve process synchronization. The three semaphores used to solve the problem are:

1. *FULL:* The *FULL* variable is used to track the space filled in the buffer by the producer process. It is initialized to 0 initially as initially no space is filled by the producer processes**.**
2. *EMPTY:* The *EMPTY* variable is used to track the empty space in the buffer. The empty variable is initially initialized to the *BUFFER_SIZE* as initially, the whole buffer is empty.
3. *MUTEX*: *MUTEX* is used to achieve mutual exclusion. *MUTEX* ensures that at any particular time only the producer or the consumer is accessing the buffer. *MUTEX* is a binary semaphore variable that has a value of 0 or 1.

SIGNAL() and *WAIT()* operations are used in the three semaphores.

SIGNAL() increases the semaphore value by 1.

WAIT() decreases the semaphore value by 1.

The algorithm of the producer-consumer problem is as follows. I will implement this algorithm in both Windows and Linux.

```
SHARED MEMORY:
char buf[BUFFER_SIZE]
int in=0, out=0
semaphore full=0
semaphore empty=BUFFER_SIZE
semaphore mutex=1
```

```
PRODUCER
void Producer() {
        while(true) {
                wait(empty)
                wait(mutex)
                buf[in] = c
                in = (in+1) % BUFFER_SIZE
                signal(mutex)
                signal(full)
        }
}
```

```
CONSUMER
void Consumer() {
        while(true) {
                wait(full)
                wait(mutex)
                c = buf[out]
                out = (out+1) %
                BUFFER_SIZE
                signal(mutex)
                signal(empty)
                return c
        }
}
```

In this algorithm, the producer is running in a loop, which performs a computation to generate information. In this case it waits for a random period of time within 3 seconds (including 3 seconds) and randomly adds a single character from my initials XQL to the buffer.

The FIFO buffers are implemented as a BUFFER_SIZE-element character array with two indices.

IN: the write index indicates the next character to be written

OUT: the read index indicated the next character to be read

The *IN* and *OUT* indices are incremented modulo BUFFER_SIZE i.e the next element to be accessed after the BUFFER_SIZE+1 element is the 0^{th} element, hence the name “circular buffer”.

In the producer:

WAIT(EMPTY) - Before producing items, the producer process checks for the empty space in the buffer. If the buffer is full producer process waits for the consumer process to consume items from the buffer. so, the producer process executes *WAIT(EMPTY)* before producing any item.

WAIT(MUTEX) - Only one process can access the buffer at a time. So, once the producer process enters into the critical section of the code it decreases the value of *MUTEX* by executing *WAIT(MUTEX)* so that no other process can access the buffer at the same time.

BUF[IN] = C - adds the item to the buffer produced by the Producer process. once the Producer process reaches this point in the code, it is guaranteed that no other process will be able to access the shared buffer concurrently which helps in data consistency.

SIGNAL(MUTEX) - Now, once the Producer process added the item into the buffer it increases the *MUTEX* value by 1 so that other processes which were in a busy-waiting state can access the critical section.

SIGNAL(FULL) - when the producer process adds an item into the buffer spaces is filled by one item so it increases the Full semaphore so that it indicates the filled spaces in the buffer correctly.

In the consumer:

WAIT(FULL) - Before the consumer process starts consuming any item from the buffer it checks if the buffer is empty or has some item in it. So, the consumer process creates one more empty space in the buffer and this is indicated by the full variable. The value of the full variable decreases by one when the *WAIT(FULL)* is executed. If the Full variable is already zero i.e the buffer is empty then the consumer process cannot consume any item from the buffer and it goes in the busy-waiting state.

WAIT(MUTEX) - It does the same as explained in the producer process. It decreases the *MUTEX* by 1 and restricts another process to enter the critical section until the consumer process increases the value of *MUTEX* by 1.

C = BUF[OUT] - This function consumes an item from the buffer. when code reaches the this point it will not allow any other process to access the critical section which maintains the data consistency.

SIGNAL(MUTEX) - After consuming the item it increases the *MUTEX* value by 1 so that other processes which are in a busy-waiting state can access the critical section now.

SIGNAL(EMPTY) - when a consumer process consumes an item it increases the value of the *EMPTY* variable indicating that the empty space in the buffer is increased by 1.

Execution [Windows]

Set up Windows environment for compiling C program by following the steps listed here: Walkthrough: Compile a C program on the command line | Microsoft Learn.

Step 1: Create the producerconsumer.c program

In the Developer Command Prompt for Visual Studio 2022 create a C file using Notepad with the name parentprocess.c by running the following command:

```
> notepad producerconsumer.c
```

In the file, write the following code:

```
// BIT 100073007 Operating Systems Course Lab 3: Producer-Consumer
Problem
// WIN32

#include <stdio.h>   // printf(), fprintf()
#include <windows.h> //
#include <stdlib.h>  // rand()
#include <string.h>  // strlen()
#include <tchar.h>   // ?
#include <strsafe.h> // ?

#define BUFFER_SIZE 4

#define PRODUCERS 3
#define CONSUMERS 4

#define PRODUCER_ITERATIONS 4 // number of times producer loops
#define CONSUMER_ITERATIONS 3 // number of times consumer loops

// Function prototype
DWORD WINAPI Producer(LPVOID lpParam);
DWORD WINAPI Consumer(LPVOID lpParam);

// define threads
DWORD consumerId[CONSUMERS], producerId[PRODUCERS];
HANDLE producerthreads[PRODUCERS];
HANDLE consumerthreads[CONSUMERS];

// define semephore empty, full, mutex
HANDLE empty, full, mutex;

typedef struct
{
    char value[BUFFER_SIZE];
    int next_in, next_out;
} buffer_t;

buffer_t buffer;
```

```
// insertInitial function is used to add an initial to the buffer
int insertInitial(char initial, long int id)
{
    buffer.value[buffer.next_in] = initial;
    buffer.next_in = (buffer.next_in + 1) % BUFFER_SIZE;
    printf("producer %ld: produced %c\n", id, initial);

    printf("\t\t\t\t\t\tBuffer: ");
    for (int i = 0; i < BUFFER_SIZE; i++)
    {
        printf("%c ", buffer.value[i]);
    }
    printf("\n");

    return 0;
}

// consumeInitial function is used to consume an initial from the
buffer
int consumeInitial(char *initial, long int id)
{
    *initial = buffer.value[buffer.next_out];
    buffer.value[buffer.next_out] = '-';
    buffer.next_out = (buffer.next_out + 1) % BUFFER_SIZE;
    printf("\t\t\tconsumer %ld: consumed %c\n", id, *initial);

    printf("\t\t\t\t\t\tBuffer: ");

    for (int i = 0; i < BUFFER_SIZE; i++)
    {
        printf("%c ", buffer.value[i]);
    }
    printf("\n");

    return 0;
}

// Producer will iterate PRODUCER_ITERATIONS times and call the
insertInitial function to insert an initial to the buffer
// Producer argument param is an integer id of the producer used
to distiguish between the multiple producer threads
DWORD WINAPI Producer(LPVOID lpParam)
{

    char initials[] = "XQL";
    int length = strlen(initials);

    char initial;
    long int id = (long int)lpParam;

    int j = PRODUCER_ITERATIONS;
```

```
    while (j--)
    {
        // wait for random length of time from 0 to 3 seconds
        int randnum = rand() % 4; // range between 0 and 3

        // insert random initial into buffer
        int randInitial = rand() % length;

        WaitForSingleObject(empty, INFINITE);
        WaitForSingleObject(mutex, INFINITE);
        initial = initials[randInitial];
        if (insertInitial(initial, id))
            fprintf(stderr, "Error while inserting to buffer\n");

        ReleaseSemaphore(mutex, 1, NULL);
        ReleaseSemaphore(full, 1, NULL);
    }

    ExitThread(0);
}

// Consumer will iterate CONSUMER_ITERATIONS times and call the
consumeInitial function to insert an initial to the buffer
// Consumer argument param is an integer id of the consumer used
to distiguish between the multiple consumer threads
DWORD WINAPI Consumer(LPVOID lpParam)
{

    char initial;
    long int id = (long int)lpParam;

    int k = CONSUMER_ITERATIONS;

    while (k--)
    {
        Sleep((rand() % 6) * 1000);

        // read from buffer
        WaitForSingleObject(full, INFINITE);
        WaitForSingleObject(mutex, INFINITE);
        if (consumeInitial(&initial, id))
            fprintf(stderr, "Error while removing from buffer\n");
        ReleaseSemaphore(mutex, 1, NULL);
        ReleaseSemaphore(empty, 1, NULL);
    }

    ExitThread(0);
}

int _tmain()

{
    buffer.next_in = 0;
```

```
    buffer.next_out = 0;

    // Initialize buffer with '-'
    printf("\nInitialize buffer of size %d with '-' \n",
BUFFER_SIZE);

    printf("\t\t\t\t\t\tBuffer: ");
    for (int i = 0; i < BUFFER_SIZE; i++)
    {
        buffer.value[i] = '-';
        printf("%c ", buffer.value[i]);
    }
    printf("\n");

    // create producer and consumer threads
    long int i;

    printf("\nCreating %d Consumers and %d Producers\n\n",
CONSUMERS, PRODUCERS);

    full = CreateSemaphore(NULL, 0, BUFFER_SIZE, NULL);
    empty = CreateSemaphore(NULL, BUFFER_SIZE, BUFFER_SIZE, NULL);
    mutex = CreateSemaphore(NULL, 1, 1, NULL);

    if (full == NULL)
    {
        printf("CreateSemaphore error: %d\n", GetLastError());
        return 1;
    }

    if (empty == NULL)
    {
        printf("CreateSemaphore error: %d\n", GetLastError());
        return 1;
    }

    if (mutex == NULL)
    {
        printf("CreateSemaphore error: %d\n", GetLastError());
        return 1;
    }

    // create the producer threads
    for (i = 0; i < PRODUCERS; i++)
    {
        producerthreads[i] = CreateThread(NULL, 0, Producer,
(LPVOID)i, 0, &producerId[i]);

        if (producerthreads[i] == NULL)
        {
            printf("Error on line 197\n");
            ExitProcess(3);
        }
```

```
    }

    // create consumer threads
    for (i = 0; i < CONSUMERS; i++)
    {
        consumerthreads[i] = CreateThread(NULL, 0, Consumer,
(LPVOID)i, 0, &consumerId[i]);

        if (consumerthreads[i] == NULL)
        {
            printf("Error on line 197\n");
            ExitProcess(3);
        }
    }

    // wait for threads to complete
    for (i = 0; i < PRODUCERS; i++)
    {
        WaitForSingleObject(producerthreads[i], INFINITE);
    }

    for (i = 0; i < CONSUMERS; i++)
    {
        WaitForSingleObject(consumerthreads[i], INFINITE);
    }

    for (int i = 0; i < PRODUCERS; i++)
    {
        CloseHandle(producerthreads[i]);
    }

    for (int i = 0; i < CONSUMERS; i++)
    {
        CloseHandle(consumerthreads[i]);
    }

    CloseHandle(full);
    CloseHandle(empty);
    CloseHandle(mutex);

    return 0;
}
```

Creating a thread

CreateThread function creates a thread to execute within the virtual address spaced of the calling process. It accepts 6 arguments, 3 optional and 3 required.

1. [in, optional] LPSECURITY_ATTRIBUTES lpThreadAttributes: A pointer to a SECURITY_ATTRIBUTES structure that determines whether the returned handle can be inherited by child processes. If lpThreadAttributes is NULL, the handle cannot be inherited.

2. [in] SIZE_T dwStackSize: The initial size of the stack, in bytes. The system rounds this value to the nearest page. If this parameter is zero, the new thread uses the default size for the executable.
3. [in] LPTHREAD_START_ROUTINE lpStartAddress: A pointer to the application-defined function to be executed by the thread. This pointer represents the starting address of the thread.
4. [in, optional] __drv_aliasesMem LPVOID lpParameter: A pointer to a variable to be passed to the thread.
5. [in] DWORD dwCreationFlags: The flags that control the creation of the thread.
6. [out, optional] LPDWORD lpThreadId: A pointer to a variable that receives the thread identifier. If this parameter is NULL, the thread identifier is not returned.

The creating thread must specify the starting address of the code that the new thread is to execute. Typically, the starting address is the name of a function defined in the program code. This function takes a single parameter and returns a DWORD value. A process can have multiple threads simultaneously executing the same function.

DWORD

A DWORD is a 32-bit unsigned integer (range: 0 through 4294967295 decimal). Because a DWORD is unsigned, its first bit (Most Significant Bit (MSB)) is not reserved for signing.

This type is declared as follows:

typedef unsigned long DWORD, *PDWORD, *LPDWORD;

WINAPI

The calling convention for system functions.

LPVOID

A pointer to any type.

This type is declared in WinDef.h as follows:

typedef void *LPVOID;

LPARAM

A message parameter.

Creating a Semaphore

The CreateSemaphoreA function creates or opens a named or unnamed semaphore object. It accepts 4 parameters.

1. [in, optional] LPSECURITY_ATTRIBUTES lpSemaphoreAttributes: A pointer to a SECURITY_ATTRIBUTES structure. If this parameter is NULL, the handle cannot be inherited by child processes.
2. [in] LONG lInitialCount: The initial count for the semaphore object. This value must be greater than or equal to zero and less than or equal to lMaximumCount. The

state of a semaphore is signaled when its count is greater than zero and nonsignaled when it is zero. The count is decreased by one whenever a wait function releases a thread that was waiting for the semaphore. The count is increased by a specified amount by calling the ReleaseSemaphore function.

3. [in] LONG lMaximumCount: The maximum count for the semaphore object. This value must be greater than zero.
4. [in, optional] LPCSTR lpName: The name of the semaphore object.

WaitForSingleObject decrements the semaphore's count by one.

ReleaseSemaphore function increases the count of the specified semaphore object by a specified amount.

Step 2: Compile the C program

Use the following command to make executable version of the program.

```
> cl producerconsumer.c
```

This will output producerconsumer.exe

Step 4: Run the program

Use the following command to run the newly created executable .exe programs.

```
> producerconsumer
```

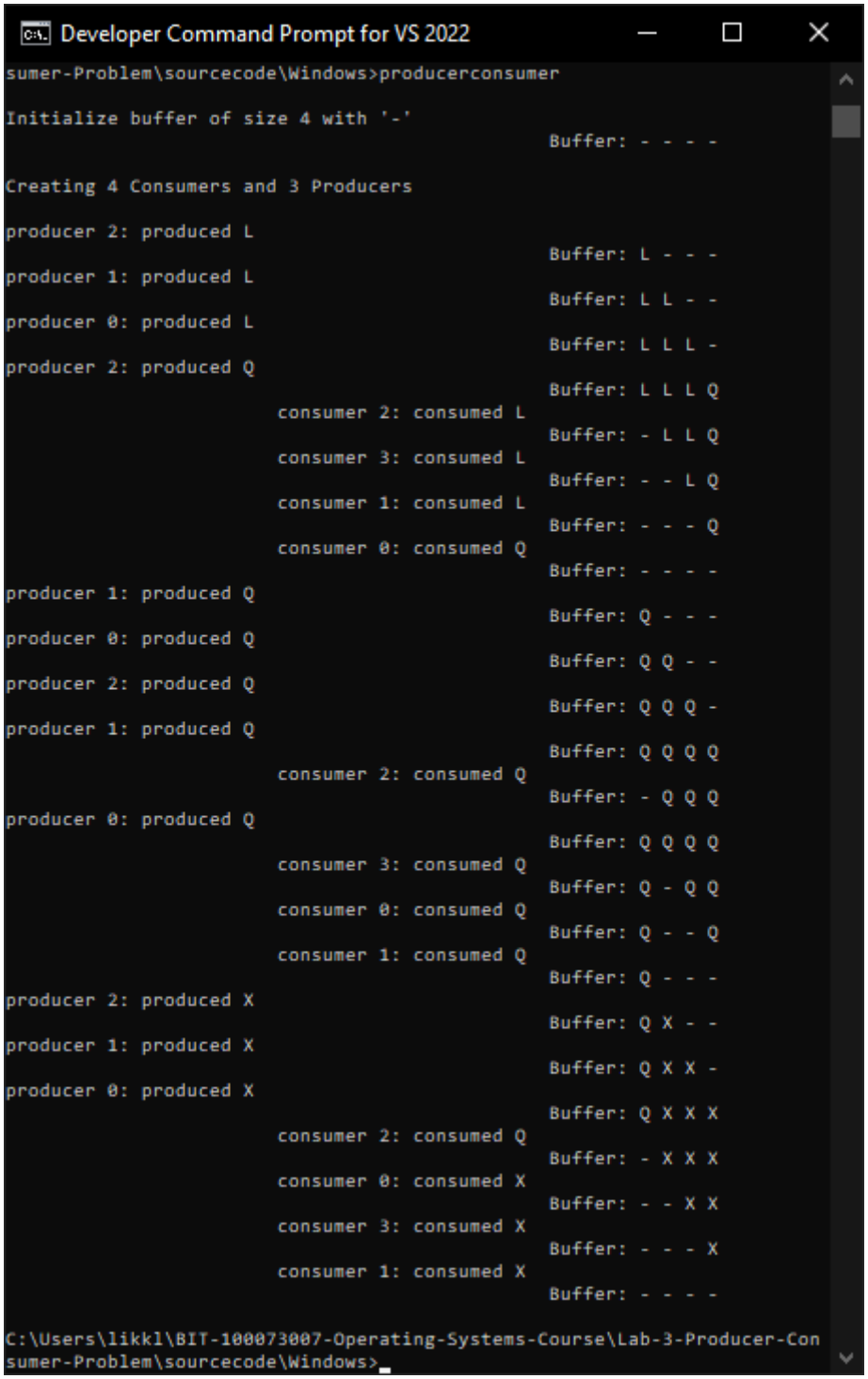
The following output is produced:

```
Initialize buffer of size 4 with '-'
                                                Buffer: - - - -
Creating 4 Consumers and 3 Producers
producer 2: produced L
                                                Buffer: L - - -
producer 1: produced L
                                                Buffer: L L - -
producer 0: produced L
                                                Buffer: L L L -
producer 2: produced Q
                                                Buffer: L L L Q
                        consumer 2: consumed L
                                                Buffer: - L L Q
                        consumer 3: consumed L
                                                Buffer: - - L Q
                        consumer 1: consumed L
                                                Buffer: - - - Q
                        consumer 0: consumed Q
```

```
                                                  Buffer: - - - -
producer 1: produced Q
                                                  Buffer: Q - - -
producer 0: produced Q
                                                  Buffer: Q Q - -
producer 2: produced Q
                                                  Buffer: Q Q Q -
producer 1: produced Q
                                                  Buffer: Q Q Q Q
                         consumer 2: consumed Q
                                                  Buffer: - Q Q Q
producer 0: produced Q
                                                  Buffer: Q Q Q Q
                         consumer 3: consumed Q
                                                  Buffer: Q - Q Q
                         consumer 0: consumed Q
                                                  Buffer: Q - - Q
                         consumer 1: consumed Q
                                                  Buffer: Q - - -
producer 2: produced X
                                                  Buffer: Q X - -
producer 1: produced X
                                                  Buffer: Q X X -
producer 0: produced X
                                                  Buffer: Q X X X
                         consumer 2: consumed Q
                                                  Buffer: - X X X
                         consumer 0: consumed X
                                                  Buffer: - - X X
                         consumer 3: consumed X
                                                  Buffer: - - - X
                         consumer 1: consumed X
                                                  Buffer: - - - -
```

Results and Analysis [Windows]

Running producerconsumer on the command line will start the producerconsumer program. The program creates 4 consumer threads and 3 producer threads. They share a buffer of size 3 which is initialized with “-”. the producer waits a random length of time between 3 seconds including 3 seconds then adds a random initial of my name XiaQiLa to the buffer. Each producer repeats this process 3 times. And prints the initials it adds to the buffer. Each consumer reads the buffer and prints the initial it consumes. After each operation, the buffer prints its contents.



Execution [Linux]

Step 1: Create C program

In the terminal, create a file using the nano editor with the name producerconsumer.c by running the following command:

```
$ nano producerconsumer.c
```

In the file we write the following code:

```
// BIT 100073007 Operating Systems Course Lab 3: Producer-Consumer
Problem

#include <stdio.h> // printf(), fprintf()
#include <stdlib.h> // rand()
#include <pthread.h> // pthread_...
#include <unistd.h> // sleep()
#include <string.h> // strlen()
#include <semaphore.h> // sem_...

#define BUFFER_SIZE 4

#define PRODUCERS 3
#define CONSUMERS 4

#define PRODUCER_ITERATIONS 4 // number of times producer loops
#define CONSUMER_ITERATIONS 3 // number of times consumer loops

pthread_t consumerId[CONSUMERS], producerId[PRODUCERS]; //define
threads

sem_t empty, full, mutex; //define 3 semaphores

typedef struct {
    char value[BUFFER_SIZE];
    int next_in, next_out;
} buffer_t;

buffer_t buffer;


// insertInitial function is used to add an initial to the buffer
int insertInitial(char initial, long int id)
{

    buffer.value[buffer.next_in] = initial;
    buffer.next_in = (buffer.next_in + 1) % BUFFER_SIZE;
    printf("producer %ld: produced %c\n", id, initial);

    printf("\t\t\t\t\t\tBuffer: ");
```

```
    for (int i = 0; i < BUFFER_SIZE; i++)
    {
        printf("%c ", buffer.value[i]);
    }
    printf("\n");

    return 0;
}

// consumeInitial function is used to consume an initial from the
buffer
int consumerInitial(char *initial, long int id)
{
     *initial = buffer.value[buffer.next_out];
     buffer.value[buffer.next_out] = '-';
     buffer.next_out = (buffer.next_out + 1) % BUFFER_SIZE;
     printf("\t\t\tconsumer %ld: consumed %c\n", id, *initial);

     printf("\t\t\t\t\t\tBuffer: ");

    for (int i = 0; i < BUFFER_SIZE; i++) {
        printf("%c ", buffer.value[i]);
    }
    printf("\n");

    return 0;
}

// Producer will iterate PRODUCER_ITERATIONS times and call the
insertInitial function to insert an initial to the buffer
// Producer argument param is an integer id of the producer used
to distiguish between the multiple producer threads
void *Producer(void *param)
{
    char initials[] = "XQL";
    int length = strlen(initials);

    char initial;
    long int id = (long int)param;

    int j = PRODUCER_ITERATIONS;
    while (j--)
    {
        // wait for random length of time from 0 to 3 seconds
        int randnum = rand() % 4; // range between 0 and 3

        // insert random initial into buffer

        int randInitial = rand() % length;

        sem_wait(&empty);
        sem_wait(&mutex);
        initial = initials[randInitial];
```

```
        if (insertInitial(initial, id))
            fprintf(stderr, "Error while inserting to buffer\n");

        sem_post(&mutex);
        sem_post(&full);
    }

    pthread_exit(0);
}

// Consumer will iterate CONSUMER_ITERATIONS times and call the
consumeInitial function to insert an initial to the buffer
// Consumer argument param is an integer id of the consumer used
to distiguish between the multiple consumer threads
void *Consumer(void *param)
{
    char initial;
    long int id = (long int)param;

    int k = CONSUMER_ITERATIONS;

    while (k--)
    {
     sleep(rand() % 6);

     // read from buffer
     sem_wait(&full);
     sem_wait(&mutex);
     if (consumerInitial(&initial, id))
          fprintf(stderr, "Error while removing from buffer\n");
     sem_post(&mutex);
     sem_post(&empty);
    }

    pthread_exit(0);
}

int main()
{
    buffer.next_in = 0;
    buffer.next_out = 0;

    // Initialize buffer with '-'

    printf("\nInitialize buffer of size %d with '-' \n",
BUFFER_SIZE);

    printf("\t\t\t\t\t\tBuffer: ");
    for (int i = 0; i < BUFFER_SIZE; i++) {
        buffer.value[i] = '-';
        printf("%c ", buffer.value[i]);
    }
    printf("\n");
```

```
    // create producer and consumer threads

    long int i;

    // create consumer threads
    printf("\nCreating %d Consumers and %d Producers\n\n",
CONSUMERS, PRODUCERS);

    sem_init(&full, 0, 0);
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&mutex, 0, 1);

        // Create the producer threads
    for (i = 0; i < PRODUCERS; i++)
        if (pthread_create(&producerId[i], NULL, Producer, (void
*)i) != 0) {
            perror("pthread_create");
            abort();
        }

    for (i = 0; i < CONSUMERS; i++)
        if (pthread_create(&consumerId[i], NULL, Consumer, (void
*)i) != 0) {
            perror("pthread_create");
            abort();
        }

        // Wait for threads to complete

    for (i = 0; i < PRODUCERS; i++)
        if (pthread_join(producerId[i], NULL) != 0) {
            perror("pthread_join");
            abort();
        }

    for (i = 0; i < CONSUMERS; i++)
        if (pthread_join(consumerId[i], NULL) != 0) {
            perror("pthread_join");
            abort();
        }

    return 0;
}
```

random length function

to generate random numbers the standard library functions RAND() and SRAND() are used. They are defined in the <stdlib.h> header file.

The syntax of RAND() is *int rand(void).* The function returns an integer value ranging from 0 and RAND_MAX. RAND_MAX is a symbolic constant defined in <stdlib.h> whose value ranges from 0 to 32767.

The “seed” is a starting point for the sequence. One would get the same sequence of numbers every time the program is run if they used the same seed. This is why it is a “pseudo-random” function.

By default the seed of the RAND() function is the set to 1.

RAND() modulo 4 sets the range of random number from 0 to 3.

```
#include <stdlib.h>

void getrand() {
        int x = rand() % 4 // range between 0 and 3
        return x
}
```

sleep function

The sleep() function suspends execution of the requesting thread until the number of real-time seconds provided by the argument seconds has passed or a signal is given to the calling thread with the action of invoking a signal-catching function or terminating the process has elapsed. The return value of the sleep function should be 0 if the specified period has passed.

Threads

Threads can be created in a POSIX system using PTHREAD. The header file <pthread.h> is needed

Thread ID

Each thread has an object of type pthread_t associated with it that tells its ID. The same pthread_t object cannot be used by multiple threads simultaneously. For multiple threads an array can be created where each element is an ID for a separate thread. *pthread_t id[2]*

Creating a thread

A thread is created and starts using the function *pthread_create()*. It takes four parameters:

- ID of type *pthread_t *:* reference (or pointer) to the ID of the thread
- Attributes of type *pthread_attr_t* *: used to set the attributes of a thread. NULL is fine in the case of this experiment
- Starting routine of type *void* *: the name of the function that the thread starts to execute. If the functions return type is *void* * then its name is simply written; otherwise it has to be type-cast to void *
- Arguments of type *void* *: this is the argument that the starting routine takes. If it takes multiple arguments a *struct* is used.

The return type of a starting routine and its argument is usually set to *void* *.

```
pthread_create(&id[0], NULL, printNumber, &arg);
```

Exiting a thread

pthread_exit() is used to exit a thread. The function is written at the end of the starting routine. Since a thread's local variables are destroyed when they exit, only references to global or dynamic variables are returned.

Waiting for a thread

A parent thread is made to wait for a child thread using pthread_join(). The two parameters of this function are:

- Thread ID of type *pthread_t*: the ID of the thread that the parent thread waits for.
- Reference to return value *void***: the value returned by the exiting thread is caught by this pointer

sleep

we have to include #include <unistd.h> to use sleep() in linux

Step 2: Compile the C program

Use the following command to make executable version of the two programs.

```
$ gcc producerconsumer.c -o producerconsumer
```

This will output producerconsumer.exe

Step 5: Run the program

Use the following command to run the newly created executable .exe programs.

```
$ ./producerconsumer
```

Results and Analysis [Linux]

The program successfully met all requirements of the experiment. It initialized a buffer of size 4 with “-”, created 4 consumers and 3 producers. The producer and consumer worked synchronously to read and write to the butter without overflow. The content of the buffer is printed after each computation from the producer and consumer threads.

References:

- https://www.scaler.com/topics/operating-system/producer-consumer-problem-in-os/
- https://computationstructures.org/lectures/synchronization/synchronization.html
- https://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html
- https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/what-is-modular-arithmetic
- https://www.educative.io/answers/how-to-create-a-simple-thread-in-c
- http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
- https://www.ibm.com/docs/en/zos/2.1.0?topic=functions-rand-generate-random-number
- https://medium.com/@sohamshah456/producer-consumer-programming-with-c-d0d47b8f103f
- https://faculty.cs.niu.edu/~hutchins/csci480/semaphor.htm
- https://learn.microsoft.com/en-gb/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread?redirectedfrom=MSDN
- https://en.wikibooks.org/wiki/Windows_Programming/Handles_and_Data_Types#DWORD,_WORD,_BYTE
- https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-dtyp/262627d8-3418-4627-9218-4ffe110850b2
- https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createsemaphorea