**Final Project Report**

**Shared Memory & Semaphores Method**

**Description of design:**

This project uses a struct data type for the ring buffer, which contains an array to store all the shared data for future use. It also maintains two counting semaphores that are used for mutual exclusion between the producer and consumer processes. The consumer process is created by calling fork() in the main method. The producer process is allowed to write data to the ring buffer only when there is an empty slot, whereas the consumer is permitted to access the process to read when there is data in it.

**All system calls:**

System calls used by the main method:

(1) line 87 mmap():a system call used to map files or devices into memory. It is used for creating shared memory .

(2) line 96 fork():a system call used to create a new process by duplicating the existing process

(3) line 110 munmap():a system call used to unmap previously mapped regions of memory.

(4) line 100: sleep():This is not a direct system call, but it will invoke other system calls to do the suspension of current thread for a given number.

System calls used by Producer:

(1) line 30:fopen(): It is not a system call, but it wraps the open system call to open a file.

(2) line 44:fprintf():It is not a system call, but it wraps the system call to write data into a file.

(3) line 56: fclose():It is not a system call, but it wraps the system call to close an file.

System calls used by Consumer:

(1) line 65:fopen(): It is not a system call, but it wraps the open system call to open a file.

(2) line 75:fprintf():It is not a system call, but it wraps the system call to write data into a file.

(3) line 76: fclose():It is not a system call, but it wraps the system call to close an file.

**Source code** **(**For the purpose of screenshots, I set the buffer size to 10**)**:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <fcntl.h>

#include <sys/shm.h>

#include <sys/stat.h>

#include <sys/mman.h>

#include <time.h>

#include <unistd.h>

#include <semaphore.h>

#include <sys/wait.h>

#include <time.h>

#define BUFFER\_SIZE 10

struct RingBuffer{

int data[BUFFER\_SIZE];

int read\_index;

int write\_index;

sem\_t empty\_slots;

sem\_t filled\_slots;

};

struct RingBuffer \*buffer;

/\*----------Producer----------\*/

int producer(){

//Store all randomNumbers in to producer\_randomNumber.txt

FILE \*file;

file = fopen("producer\_randomNumber.txt","w");

if(file==NULL){

printf("fail to create a producer file!");

return -1;

}

int randomNUmbers[BUFFER\_SIZE];

srand(time(NULL));

for(int i = 0; i < BUFFER\_SIZE; i++) {

int randomNumber = rand() % 1000 + 1;

//array[i]=malloc(12\*sizeof(char));

//printf(array[i],"%d",randomNumber);

randomNUmbers[i]=randomNumber;

fprintf(file,"%d\n",randomNumber); //write to the .txt file

}

for(int i=0; i<BUFFER\_SIZE; i++){

sem\_wait(&buffer->empty\_slots); //if empty\_slots if >0,it will be deceased and access allowed; if ==0,Producer is blocked

buffer->data[buffer->write\_index] = randomNUmbers[i];

printf("Producer is producing item: %d\n",buffer->data[buffer->write\_index]);

buffer->write\_index = (buffer->write\_index + 1)%BUFFER\_SIZE; //because it is Ring buffer so next element is this way.

sem\_post(&buffer->filled\_slots);//increase the semaphore by one

}

fclose(file);

}

/\*----------Consumer----------\*/

int consumer(){

//Store all received randomNumbers in to consumer\_randomNumber.txt

FILE \*file;

file = fopen("consumer\_randomNumber","w");

if(file==NULL){

printf("fail to create a consumer file!");

return -1;

}

for(int i=0; i<BUFFER\_SIZE; i++){

sem\_wait(&buffer->filled\_slots); //if filled\_slots: >0, it will be decreased by one; ==0,no resources avalible, the consumer process will be blocked

int item = buffer->data[buffer->read\_index];

fprintf(file,"%d\n",item); //write to the .txt file

fclose(file);

printf("Consumer is reading data : %d\n",item);

buffer->read\_index = (buffer->read\_index+1)% BUFFER\_SIZE;

sem\_post(&buffer->empty\_slots); //incement the semaphore pointed to by empty\_slots by one

}}

int main(){

//allocate space for Ring buffer pointer

buffer = mmap(NULL,sizeof(struct RingBuffer),PROT\_READ|PROT\_WRITE, MAP\_SHARED|MAP\_ANONYMOUS,-1,0);

//Initialize index

buffer->read\_index =0;

buffer->write\_index =0;

//initialize semaphores

sem\_init(&buffer->empty\_slots,1,BUFFER\_SIZE);

sem\_init(&buffer->filled\_slots,1,0);

//Fork a child process whose pid == 0

pid\_t pid = fork();

if(pid<0){

fprintf(stderr,"fork fail.");

}else if(pid==0){

sleep(5);//Let consumer process delay 5 seconds

consumer();

}else{

producer();

wait(NULL); //wait for child to finish

}

//After both producer and consumer finished, it destroy the semaphores and unmap the shared memory

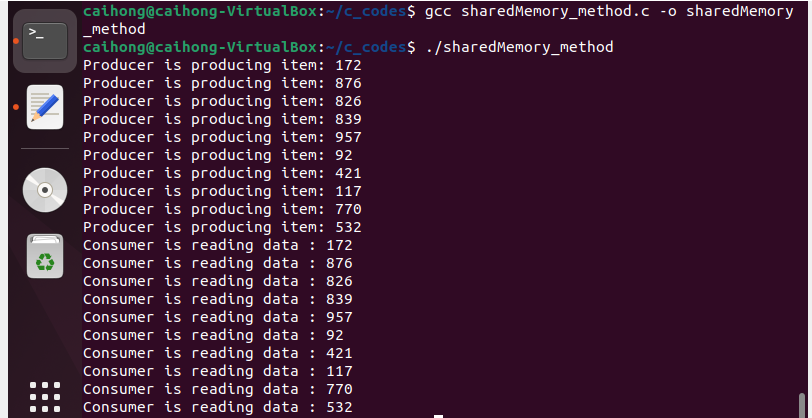
sem\_destroy(&buffer->empty\_slots);

sem\_destroy(&buffer->filled\_slots);

munmap(buffer,sizeof(struct RingBuffer()));

return 0;

**Result(**For the purpose of screenshots, I set the buffer size to 10**)**:



**Overall Discussion:**

The code provided in section 3.7.1 doesn't execute correctly if you attempt to run it sequentially on the Linux OS. The shared memory object is removed after the producer process finishes, so when you try to run the consumer process, it returns errors due to its inability to locate the shared memory. This is why I encapsulated the execution of both the producer and consumer processes within the main method. Here, the consumer process is created as the child process of the producer.

However, I did encounter a significant issue: process racing. Although I used semaphores to ensure that only one process at a time could access the shared buffer space, an issue arises if, like me, you prefer the idea of each process completing its task before the other process gets involved. In this scenario, you need to incorporate a delay on the consumer process. Interestingly, with or without the delay, the consumer seems capable of reading exactly the same amount of data as produced by the producer. I've increased the test quantity from 100 to 300,000, and it appears that no errors occur.

One point worth mentioning is that the semaphores provided by <semaphore.h> are very convenient to use. The **sem\_wait** function it offers automatically checks if the given integer counter is greater than or equal to zero. If the semaphore value is greater than zero at that moment, it allows the process to continue. If it's equal to zero, the process is blocked. Furthermore, **sem\_post** increments the semaphore by one after the process finishes the critical section.