**6. IPC:Semaphores**

**6.2** Write a program to illustrate the semaphore concept. Use fork sothat 2 process running simultaneously and communicate via semaphore.

**Objectives:**

1. To learn about IPC through semaphore.
2. Use of system call and IPC mechanism to write effective application programs.

**Theory:**

The problem describes two processes, the producer and the consumer, who share a common, fixed-size [buffer](https://en.wikipedia.org/wiki/Buffer_(computer_science)) used as a [queue.](https://en.wikipedia.org/wiki/Queue_(data_structure)) The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

**Data Dictionary:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Number** | **Variable/function** | **Data Type** | **Use** |
|  |  |  |  |
| 1 | Producers | pthread\_t | Process Thread of Producer |
|  |  |  |  |
| 2 | Consumer | pthread\_t | Process Thread of Consumer |
|  |  |  |  |
| 3 | buf\_mutex | sem\_t | To process wait condition |
|  |  |  |  |
| 4 | empty\_count | sem\_t | Keeps track of empty count |
|  |  |  |  |
| 5 | fill\_count | sem\_t | Keeps track of fill count |
|  |  |  |  |
| 6 | consumer | void | Used to regulate consumer action |
|  |  |  |  |
| 7 | producer | void | Used to regulate producer action |
|  |  |  |  |

Table 6.2 Data Dictonary

**Program :**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

#include <unistd.h>

#include <pthread.h>

#include <semaphore.h>

/\* use the pthread flag with gcc to compile this code

~$ gcc -pthread producer\_consumer.c -o producer\_consumer \*/

pthread\_t \*producers;

pthread\_t \*consumers;

sem\_t buf\_mutex,empty\_count,fill\_count;

int \*buf,buf\_pos =-1,prod\_count,con\_count,buf\_len; int produce(pthread\_t self){ int i = 0;

int p = 1 + rand()%40;

while(!pthread\_equal(\*(producers+i),self) && i < prod\_count){

i++;

}

printf("Producer %d produced %d \n",i+1,p);

return p;

}

void consume(int p,pthread\_t self){

int i = 0;

while(!pthread\_equal(\*(consumers+i),self) && i < con\_count){

i++;

}

printf("Buffer:");

for(i=0;i<=buf\_pos;++i)

printf("%d ",\*(buf+i));

printf("\nConsumer %d consumed %d \nCurrent buffer len: %d\n",i+1,p,buf\_pos);

}

void\* producer(void \*args){

while(1){

int p = produce(pthread\_self());

sem\_wait(&empty\_count);

sem\_wait(&buf\_mutex);

++buf\_pos; // critical section

\*(buf + buf\_pos) = p;

sem\_post(&buf\_mutex);

sem\_post(&fill\_count);

sleep(1 + rand()%3);

}

return NULL;

}

void\* consumer(void \*args){

int c;

while(1){

sem\_wait(&fill\_count);

sem\_wait(&buf\_mutex);

c = \*(buf+buf\_pos);

consume(c,pthread\_self());

--buf\_pos;

sem\_post(&buf\_mutex);

sem\_post(&empty\_count);

sleep(1+rand()%5);

}

return NULL;

}

int main(void){

int i,err;

srand(time(NULL));

sem\_init(&buf\_mutex,0,1);

sem\_init(&fill\_count,0,0);

printf("Enter the number of Producers:");

scanf("%d",&prod\_count);

producers = (pthread\_t\*) malloc(prod\_count\*sizeof(pthread\_t)); printf("Enter the number of Consumers:"); scanf("%d",&con\_count);

consumers = (pthread\_t\*) malloc(con\_count\*sizeof(pthread\_t));

printf("Enter buffer capacity:");

scanf("%d",&buf\_len);

buf = (int\*) malloc(buf\_len\*sizeof(int));

sem\_init(&empty\_count,0,buf\_len);

for(i=0;i<prod\_count;i++){

err = pthread\_create(producers+i,NULL,&producer,NULL); if(err != 0){

printf("Error creating producer %d: %s\n",i+1,strerror(err)); }else{

printf("Successfully created producer %d\n",i+1);

}

}

for(i=0;i<con\_count;i++){

err = pthread\_create(consumers+i,NULL,&consumer,NULL); if(err != 0){

printf("Error creating consumer %d: %s\n",i+1,strerror(err)); }else{

printf("Successfully created consumer %d\n",i+1); }

}

for(i=0;i<prod\_count;i++){

pthread\_join(\*(producers+i),NULL);

}

for(i=0;i<con\_count;i++){

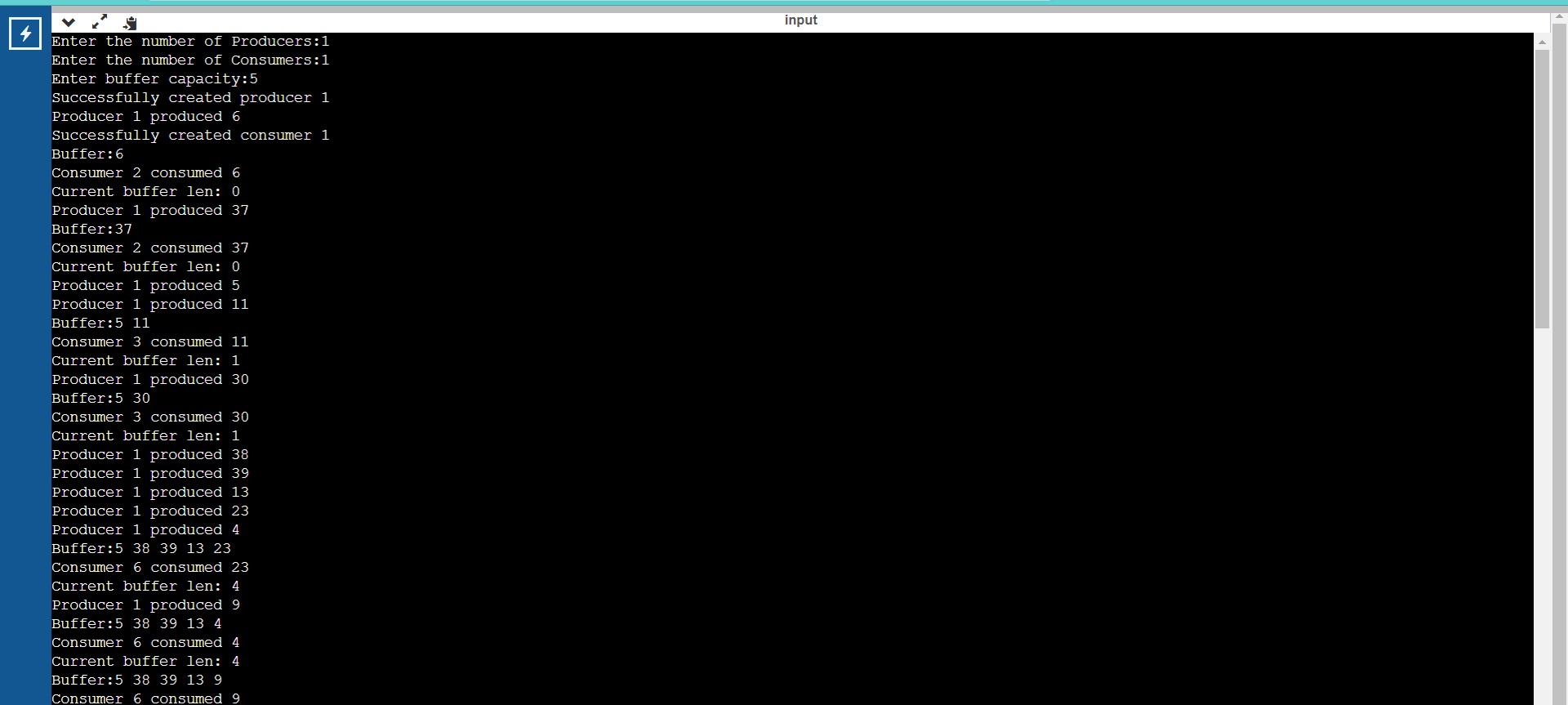
pthread\_join(\*(consumers+i),NULL);

}

return 0;

}

**Output:**



**Conclusion:**

1.Synchronization using IPC semaphores done to implement

2.Study Producer-Consumer problem.

**References:**

Dave’s Programming in C Tutorials