COM301P

OS Lab Assignment 7

Animesh Kumar
CED18I065
15th November, 2020

Q1: Simulate the Producer Consumer code discussed in the class.

- Algorithm:
    - The Producer will be creating new items in a time period of 1 sec and the consumer will consume in a time period of 5 sec. This is done to simulate the asynchronous access to the shared buffer.
    - There will be a controller thread also, it will terminate both the producer and consumer threads after 60 sec.
    - Below pseudocode is the same as taught in class.

Producer:

```
while (true) {
        // item not produced
        while ((in+1) % BS = = out)
        // do nothing as the Buffer is Full
        buffer [in] = next-produced-item;
        in = ( in + 1) % BS ;
}
```

Consumer:

```
while (true) {
        // item not consumed
        while (in = = out)
        //do nothing as Buffer is Empty
        next-consumed-item = buffer(out]:
        out = (out + l ) % BS ;
}
```

- C Program:

```c
#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>
```

```c
#include<string.h>
#include<sys/wait.h>
#include<time.h>
#include<pthread.h>
#include<signal.h>
#include<stdlib.h>

#define BUFF_SIZE 5

int Buffer[BUFF_SIZE];
int in_ptr = 0;
int out_ptr = 0;

void* runner_producer(void * params);
void* runner_consumer(void * params);
void* runner_controller(void * params);

void kill_handler(int signum);
int ProduceItem();
void PrintBuffer();


int main() {
```

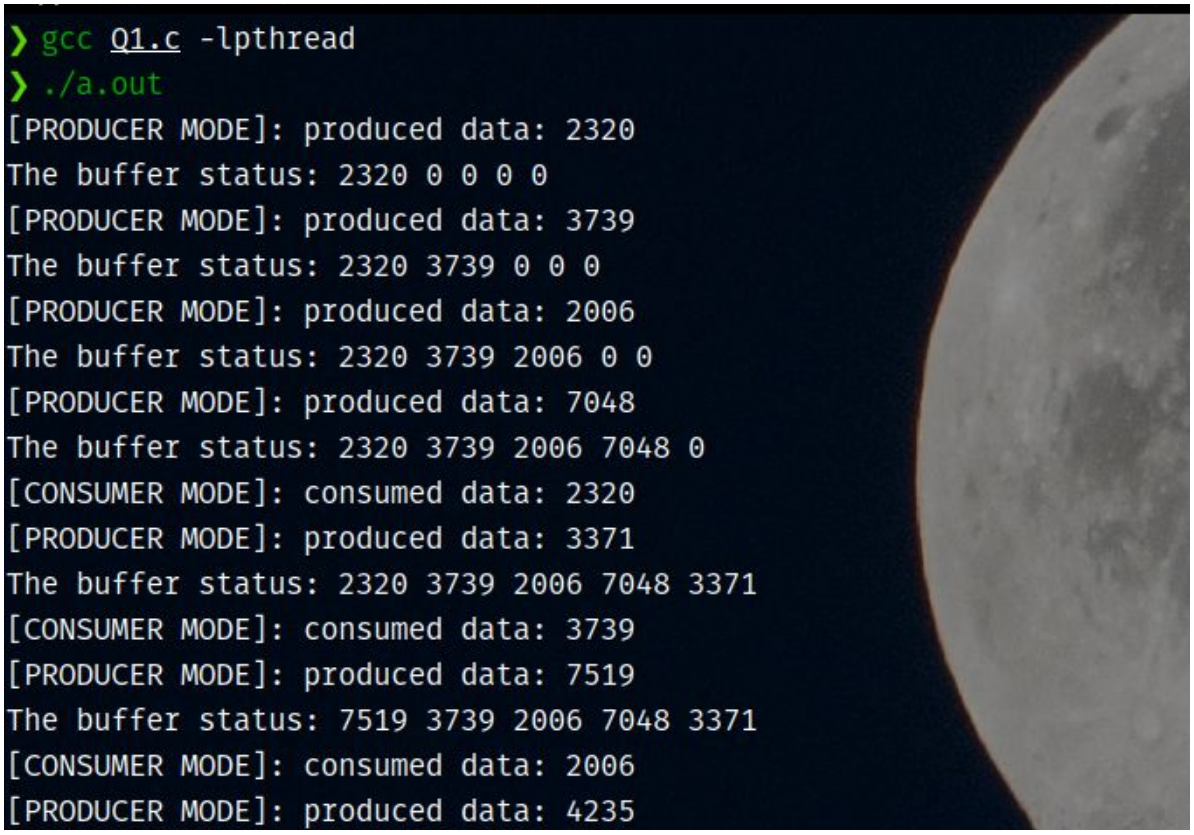
```
 pthread_t tids[3];
 pthread_create(&tids[0], NULL, runner_producer, NULL);
 pthread_create(&tids[1], NULL, runner_consumer, NULL);
 pthread_create(&tids[2], NULL, runner_controller, &tids);
 pthread_join(tids[2], NULL);
 return 0;
}


// the producer thread
void* runner_producer(void * params) {
 while(1) {
   int new_item = ProduceItem();
   sleep(1);
   while((in_ptr + 1) % BUFF_SIZE == out_ptr) { // buffer full
     // do nothing
   }
   Buffer[in_ptr] = new_item;
   printf("[PRODUCER MODE]: produced data: %d\n", Buffer[in_ptr]);
   PrintBuffer();
   in_ptr = (in_ptr + 1) % BUFF_SIZE;
 }
 pthread_exit(NULL);
}
```

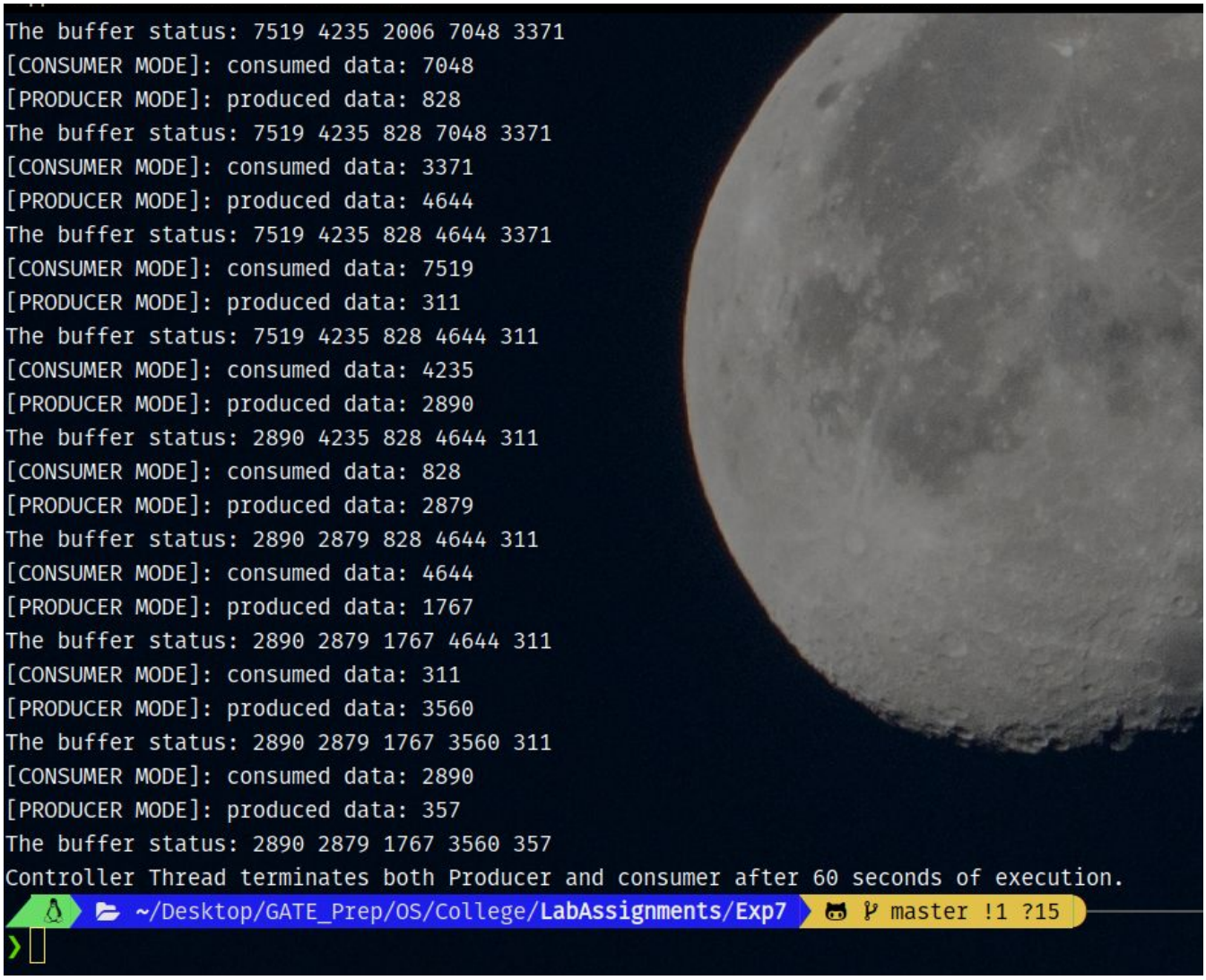
```c
// The consumer thread
void* runner_consumer(void * params) {
 while(1) {
   while(in_ptr == out_ptr) {  // buffer empty
     // do nothing
   }
   sleep(5);
   printf("[CONSUMER MODE]: consumed data: %d\n", Buffer[out_ptr]);
   out_ptr = (out_ptr + 1) % BUFF_SIZE;
 }
 pthread_exit(NULL);
}

// Controls the two threads and terminates them
void* runner_controller(void * params) {
 pthread_t * tids = params;
 signal(SIGUSR1, kill_handler);
 sleep(60);
 pthread_kill(tids[0], SIGUSR1);
 pthread_kill(tids[1], SIGUSR1);
 pthread_join(tids[0], NULL);
 pthread_join(tids[1], NULL);
```

```
 printf("Controller Thread terminates both Producer and consumer after 60 seconds of execution.\n");

 pthread_exit(NULL);

}


// Kills the thread when SIGUSR1 is recieved

void kill_handler(int signum) {

 pthread_exit(NULL);

}


// produces a new item randomly and return

int ProduceItem() {

 srand(time(0));

 return ((rand() % 10000) + 100);

}


// Prints the current status of the buffer

void PrintBuffer() {

 printf("The buffer status: ");

 fflush(stdout);

 for(int i = 0; i < BUFF_SIZE; i++)

   printf("%d ", Buffer[i]);

 printf("\n");
```

```
}
```

- Output:


❯ gcc Q1.c -lpthread
❯ ./a.out
[PRODUCER MODE]: produced data: 2320
The buffer status: 2320 0 0 0 0
[PRODUCER MODE]: produced data: 3739
The buffer status: 2320 3739 0 0 0
[PRODUCER MODE]: produced data: 2006
The buffer status: 2320 3739 2006 0 0
[PRODUCER MODE]: produced data: 7048
The buffer status: 2320 3739 2006 7048 0
[CONSUMER MODE]: consumed data: 2320
[PRODUCER MODE]: produced data: 3371
The buffer status: 2320 3739 2006 7048 3371
[CONSUMER MODE]: consumed data: 3739
[PRODUCER MODE]: produced data: 7519
The buffer status: 7519 3739 2006 7048 3371
[CONSUMER MODE]: consumed data: 2006
[PRODUCER MODE]: produced data: 4235



Q2: Extend the producer consumer simulation in Q1 to sync access of critical data using Peterson's algorithm.

- Algorithm:

  To solve the CS problem faced in Q1, we shall use the solution provided by peterson. For the producer Pi and the consumer Pj, the code of processor Pi, critical section handling is as below

  while(TRUE) {

  flag [j] = TRUE;

```
        turn = i;
        while (flag[i] && turn == i);
        //critical section
        flag[j] = FALSE;
        //remainder section
}
```

- The consumer Pj will also access the CS in the similar way as above
  - With above locking and unlocking mechanisms using flag[] array data structure and turn variable, we shall modify the C program written for Q1.
  - All other set up including the 'controller' thread, is the same as in Q1.
- C Program:

```c
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<string.h>
#include<sys/wait.h>
#include<time.h>
#include<pthread.h>
#include<signal.h>
#include<stdlib.h>


#define BUFF_SIZE 5
#define PRODUCER 0
#define CONSUMER 1
```

```
int Buffer[BUFF_SIZE];
int in_ptr = 0;
int out_ptr = 0;
int flag[2];
int turn;

void* runner_producer(void * params);
void* runner_consumer(void * params);
void* runner_controller(void * params);

void kill_handler(int signum);
int ProduceItem();
void PrintBuffer();



int main() {
 pthread_t tids[3];
 flag[PRODUCER] = 0;
 flag[CONSUMER] = 0;
 pthread_create(&tids[0], NULL, runner_producer, NULL);
 pthread_create(&tids[1], NULL, runner_consumer, NULL);
 pthread_create(&tids[2], NULL, runner_controller, &tids);
```

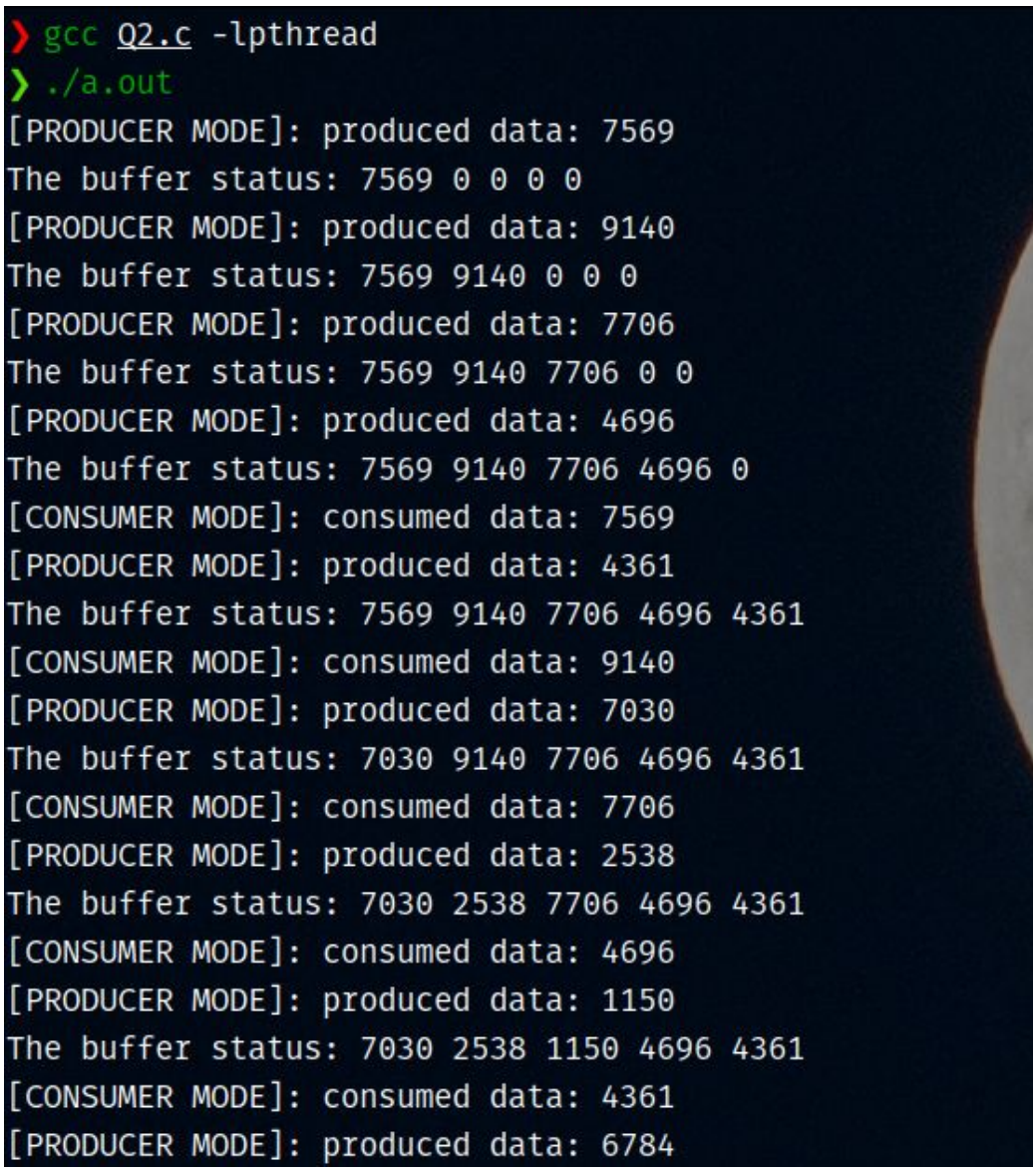
```
 pthread_join(tids[2], NULL);
 return 0;
}


// the producer thread
void* runner_producer(void * params) {
 while(1) {
   int new_item = ProduceItem();
   sleep(1);
   while((in_ptr + 1) % BUFF_SIZE == out_ptr) { // buffer full
     // do nothing
   }
   /*****************************************************************/
   // Entry Section
   flag[PRODUCER] = 1;
   turn = CONSUMER;
   while(flag[CONSUMER] && (turn == CONSUMER)){
     // busy waiting
   }
   Buffer[in_ptr] = new_item;    // Critical section
   flag[PRODUCER] = 0;        // Exit Section
   /*****************************************************************/
   printf("[PRODUCER MODE]: produced data: %d\n", Buffer[in_ptr]);
```

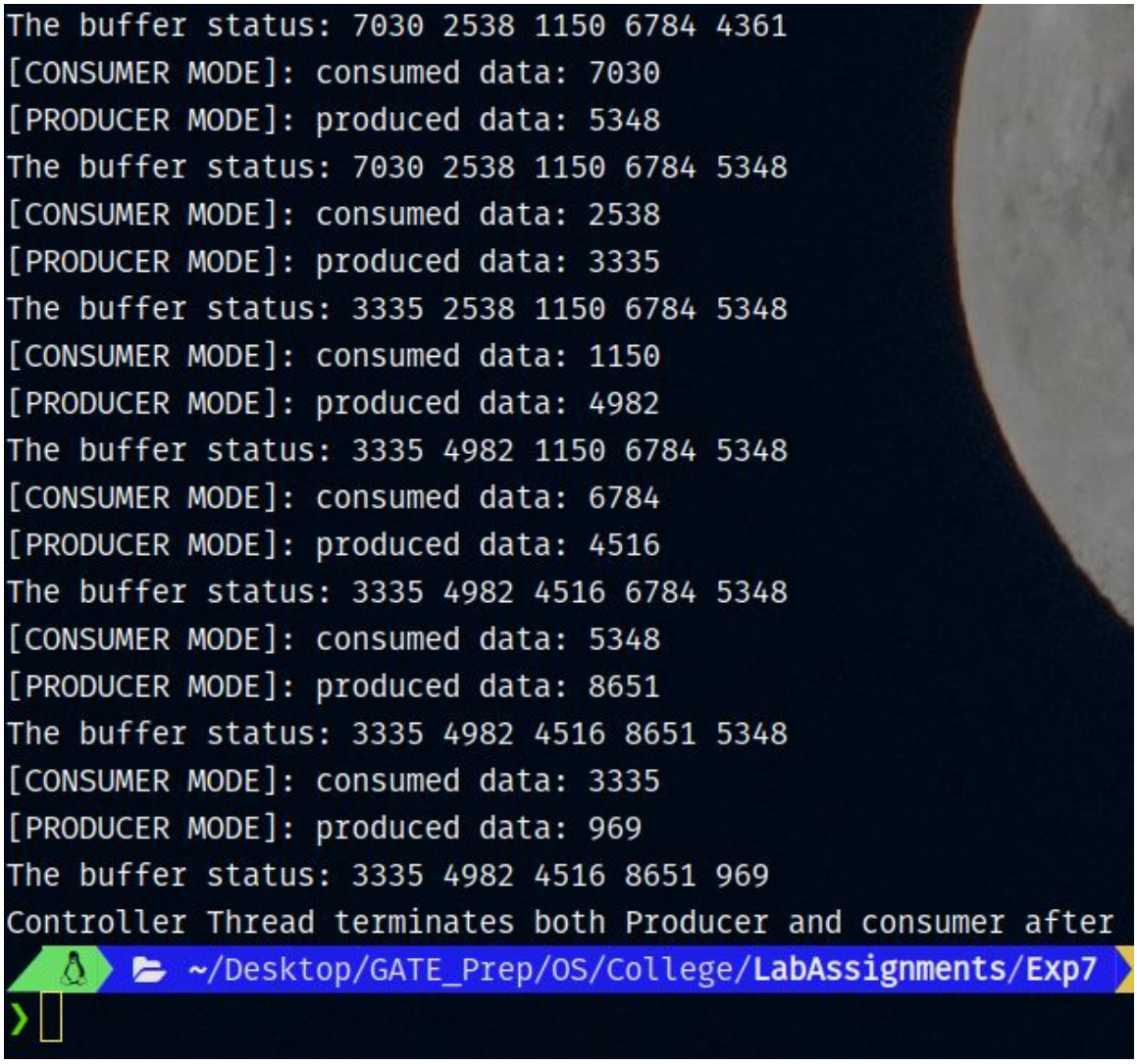
```
   PrintBuffer();
   in_ptr = (in_ptr + 1) % BUFF_SIZE;
 }
 pthread_exit(NULL);
}

// The consumer thread
void* runner_consumer(void * params) {
 while(1) {
   while(in_ptr == out_ptr) {  // buffer empty
     // do nothing
   }
   sleep(5);
   /*****************************************************************/
   // Entry section
   flag[CONSUMER] = 1;
   turn = PRODUCER;
   while(flag[PRODUCER] && (turn == PRODUCER)) {
     // busy waiting
   }
   // Critical Section
   printf("[CONSUMER MODE]: consumed data: %d\n", Buffer[out_ptr]);
   flag[CONSUMER] = 0; // Exit Section
```

```
   /*****************************************************************/
   out_ptr = (out_ptr + 1) % BUFF_SIZE;
 }
 pthread_exit(NULL);
}

// Controls the two threads and terminates them
void* runner_controller(void * params) {
 pthread_t * tids = params;
 signal(SIGUSR1, kill_handler);
 sleep(60);
 pthread_kill(tids[0], SIGUSR1);
 pthread_kill(tids[1], SIGUSR1);
 pthread_join(tids[0], NULL);
 pthread_join(tids[1], NULL);
 printf("Controller Thread terminates both Producer and consumer after 60
seconds of execution.\n");
 pthread_exit(NULL);
}

// Kills the thread when SIGUSR1 is recieved
void kill_handler(int signum) {
 pthread_exit(NULL);
```

```
}


// produces a new item randomly and return
int ProduceItem() {
 srand(time(0));
 return ((rand() % 10000) + 100);
}


// Prints the current status of the buffer
void PrintBuffer() {
 printf("The buffer status: ");
 fflush(stdout);
 for(int i = 0; i < BUFF_SIZE; i++)
   printf("%d ", Buffer[i]);
 printf("\n");
}
```

- Output:


gcc Q2.c -lpthread
./a.out
[PRODUCER MODE]: produced data: 7569
The buffer status: 7569 0 0 0 0
[PRODUCER MODE]: produced data: 9140
The buffer status: 7569 9140 0 0 0
[PRODUCER MODE]: produced data: 7706
The buffer status: 7569 9140 7706 0 0
[PRODUCER MODE]: produced data: 4696
The buffer status: 7569 9140 7706 4696 0
[CONSUMER MODE]: consumed data: 7569
[PRODUCER MODE]: produced data: 4361
The buffer status: 7569 9140 7706 4696 4361
[CONSUMER MODE]: consumed data: 9140
[PRODUCER MODE]: produced data: 7030
The buffer status: 7030 9140 7706 4696 4361
[CONSUMER MODE]: consumed data: 7706
[PRODUCER MODE]: produced data: 2538
The buffer status: 7030 2538 7706 4696 4361
[CONSUMER MODE]: consumed data: 4696
[PRODUCER MODE]: produced data: 1150
The buffer status: 7030 2538 1150 4696 4361
[CONSUMER MODE]: consumed data: 4361
[PRODUCER MODE]: produced data: 6784


The buffer status: 7030 2538 1150 6784 4361
[CONSUMER MODE]: consumed data: 7030
[PRODUCER MODE]: produced data: 5348
The buffer status: 7030 2538 1150 6784 5348
[CONSUMER MODE]: consumed data: 2538
[PRODUCER MODE]: produced data: 3335
The buffer status: 3335 2538 1150 6784 5348
[CONSUMER MODE]: consumed data: 1150
[PRODUCER MODE]: produced data: 4982
The buffer status: 3335 4982 1150 6784 5348
[CONSUMER MODE]: consumed data: 6784
[PRODUCER MODE]: produced data: 4516
The buffer status: 3335 4982 4516 6784 5348
[CONSUMER MODE]: consumed data: 5348
[PRODUCER MODE]: produced data: 8651
The buffer status: 3335 4982 4516 8651 5348
[CONSUMER MODE]: consumed data: 3335
[PRODUCER MODE]: produced data: 969
The buffer status: 3335 4982 4516 8651 969
Controller Thread terminates both Producer and consumer after
~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp7

Q3 and Q4: Dictionary Problem: Let the producer set up a dictionary of at least 20 words with three attributes (Word, Primary meaning, Secondary meaning) and let the consumer search for the word and retrieve its respective primary and secondary meaning.

and

Extend Q3 to avoid duplication of dictionary entries and implement an efficient binary search on the consumer side in a multithreaded fashion.

- Algorithm:

  My interpretation is the Producer takes a word's info from the user, adds it into the dictionary and then the consumer takes over and asks for a word to search in the stdin. Then it searches the word, relevant output and hands over the control to the producer.

  - Struct word will be used to store word, its primary meaning and its secondary meaning.
  - Struct block will be used to pass data around various threads while implementing a multithreaded version of the binary search.
  - There will be a global array of words of SIZE shared among all the threads.
  - The main thread shall create two more threads for Producer and consumer separately.
  - Creating and destroying the mutex is the responsibility of the main thread.

- For synchronization I used pthread mutex lock and unlock.
- To interleave the two threads precisely alternatively, I use two variables allow_consumer and allow_producera to context switch the two threads after one iteration each. That is, accessing the CS is in the order -

  Producer->Consumer->Producer->Consumer...

The Producer Thread:

Until the dictionary size is not exhausted, do -

- (Entry section) Busy waits on the variable *allow_producer* and then waits at the mutex, then locks it and enters the critical section
- (Critical Section) Takes in a word's information including primary and secondary meaning via stdin
  - **Then checks for duplication**, if the entered word already exists in the dictionary using multithreaded efficient binary search. If the word is new, it is added to the dictionary.
- (Exit Section) Unlocks the mutex and then sets the *'allow_consumer'* variable.

The Consumer Thread:

while(TRUE):

- (Entry section) Busy waits on the variable *allow_consumer* and then waits at the mutex, then locks it and enters the critical section
- (Critical Section) Takes in a keyword whose information to be searched via stdin
  - if the word is *'exit'* - the consumer terminates the producer thread via pthread_kill and also terminates itself.
  - else retrieves the information using multithreaded efficient binary search from the dictionary and prints it on the stdout.
- (Exit Section) Unlocks the mutex and then sets the *'allow_producer'* variable.

The binary search thread(runner)

- The dictionary array is by default unsorted in nature.
- if the mid of the array is same as the keyword to be searched

        - Print the meanings and exit
    - else search the keyword both in the left and right halves of the dictionary array.

- C Program:

```
#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

#include<string.h>

#include<sys/wait.h>

#include<time.h>

#include<pthread.h>

#include<signal.h>

#include<stdlib.h>


#define SIZE 30


// A single word info

struct word {

 char self[20];     // the word

 char mean1[20];    // the first meaning

 char mean2[20];    // the second meaning

};


// A package to pass data among threads

struct block {
```

```
 int l;                // left end of Words[]
 int h;                // right end of Words[]
 char keyword[20];       // keyword to be searched
};


struct word Words[SIZE]; // the dictionary
int real_size = 0; // the real size of the dictionary at current time
int found = 0;
_Bool allow_producer = 1;
_Bool allow_consumer = 0;
int duplicate_check = 0;


pthread_mutex_t lock;


void* runner_producer(void * params);
void* runner_consumer(void * params);
void* runner(void * params);


int add_word();
void kill_handler(int signum);
_Bool is_duplicate_entry(char * keyword);
void print_dict();
```

```
int main() {
 // Mutex variable intialization
 if (pthread_mutex_init(&lock, NULL) != 0) {
   printf("\n[X]mutex init has failed...\n");
   exit(1);
 }
 // Creating producer and consumer threads
 pthread_t tid_producer, tid_consumer;
 pthread_create(&tid_producer, NULL, runner_producer, NULL);
 pthread_create(&tid_consumer, NULL, runner_consumer, &tid_producer);
 pthread_join(tid_producer, NULL);
 pthread_join(tid_consumer, NULL);
 // Destroying the mutex variable
 pthread_mutex_destroy(&lock);
 return 0;
}


// The producer thread routine which parallely adds content to the
Dictionary
void* runner_producer(void * params) {
 for(real_size = 0; ; real_size++) {
   while(!allow_producer) {  // Busy waiting for consumer to signal
     // wait
```

```
   }
   pthread_mutex_lock(&lock);
   // Critical section starts
   if(add_word() < 0){
     fprintf(stderr, "Dictionary Space Exhausted\n  >>No new words can be
added\n  >>...\n");
     break;
   }
   // Critical section ends
   pthread_mutex_unlock(&lock);

   allow_producer = !(allow_producer);
   allow_consumer = !(allow_consumer);
 }
 print_dict();
 pthread_exit(NULL);
}

// handles the signal SIGUSR1 to kill the producer thread by the consumer
thread
void kill_handler(int signum) {
 pthread_exit(NULL);
}
```

```
// The consumer thread routine, which searches in the dictionary
void* runner_consumer(void * params) {
 pthread_t * tid_producer = params;
 signal(SIGUSR1, kill_handler);
 pthread_t tid;
 struct block args;
 args.l = 0;
 while(1) {
   while(!allow_consumer) { // Busy waiting for producer to signal
     // wait
   }
   found = 0;
   pthread_mutex_lock(&lock);
   // Critical Section starts
   args.h = real_size - 1;
   printf("\n[CONSUMER MODE]: Enter key to be searched(enter 'exit' to
quit the search operation): ");
   fflush(stdout);
   scanf("%s", args.keyword);
   if(strcmp(args.keyword, "exit") == 0) {
     pthread_kill(*tid_producer, SIGUSR1); // kill the producer before
terminating
     break;
   }
```

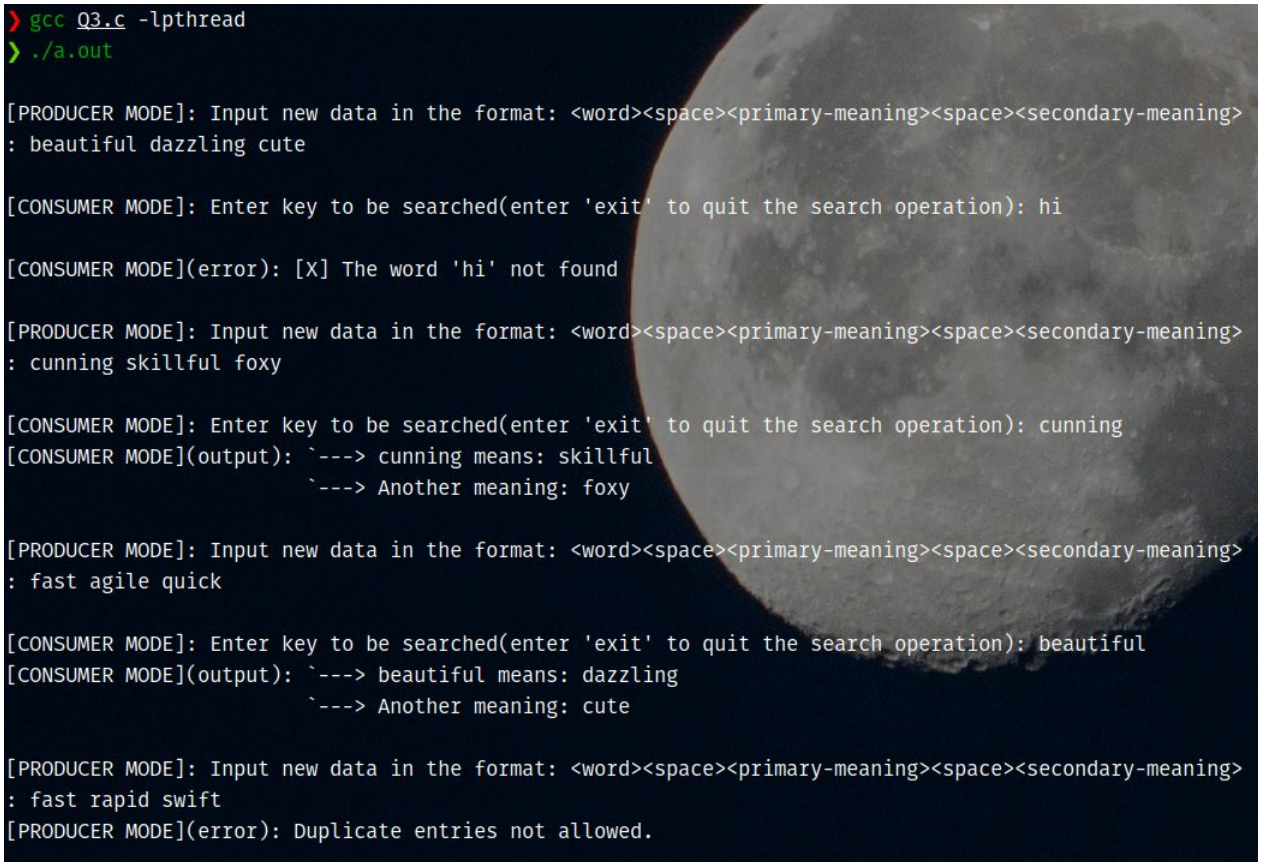
```
   pthread_create(&tid, NULL, runner, &args);
   pthread_join(tid, NULL);
   if(!found)
     printf("\n[CONSUMER MODE](error): [X] The word '%s' not found\n",
args.keyword);
   // Critical section ends
   pthread_mutex_unlock(&lock);
   allow_consumer = !(allow_consumer);
   allow_producer = !(allow_producer);
 }
 pthread_exit(NULL);
}


// adds a word in the dictionary(from stdin)
int add_word() {
 if(real_size == SIZE)    // Dictionary space exhausted
   return -1;
 struct word new;
 printf("\n[PRODUCER MODE]: Input new data in the format:
<word><space><primary-meaning><space><secondary-meaning>\n: ");
 scanf("%s %s %s", new.self, new.mean1, new.mean2);
 if(is_duplicate_entry(new.self))
   printf("[PRODUCER MODE](error): Duplicate entries not allowed.\n");
 else  // in case of new entry
```

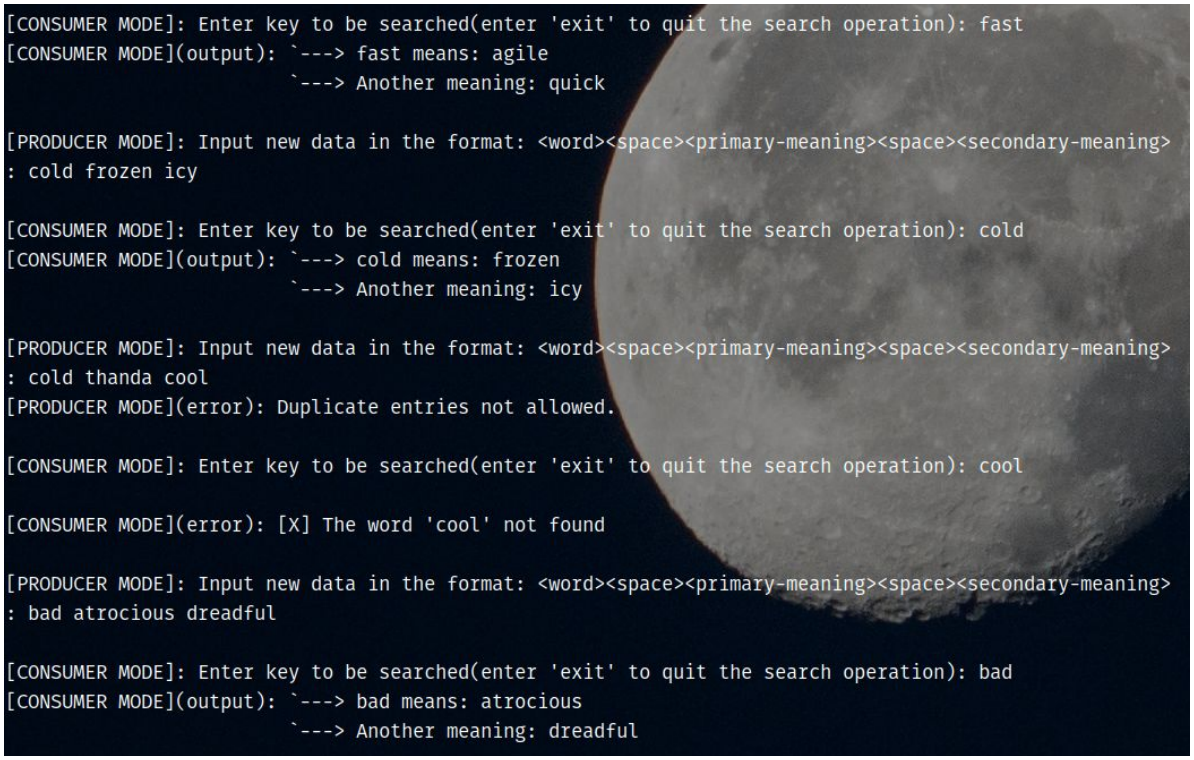
```
   Words[real_size] = new;
 return 1;
}


// Returns 1 if the keyword already exists in our dictionary
_Bool is_duplicate_entry(char * keyword) {
 found = 0;
 duplicate_check = 1;
 struct block args;
 args.l = 0;
 args.h = real_size - 1;
 strcpy(args.keyword, keyword);
 pthread_t tid;
 pthread_create(&tid, NULL, runner, &args);
 pthread_join(tid, NULL);
 duplicate_check = 0;
 return found; // whether keyword found or not
}


// Thread routine which implements an efficient binary search using
multithreading
void* runner(void * params) {
 struct block * args = params;
```

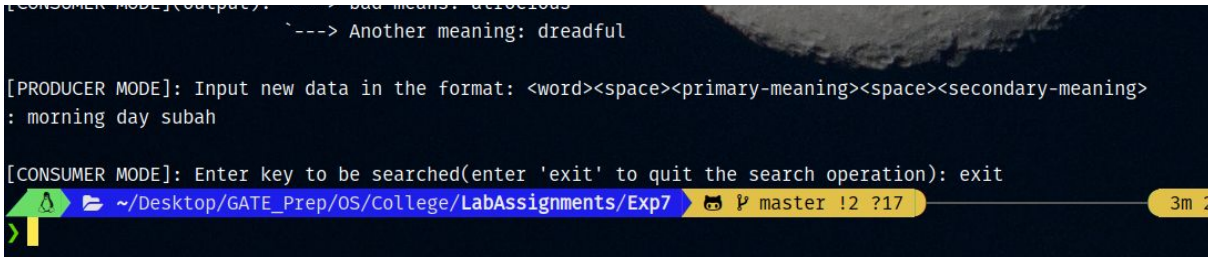
```
if(args->l > args->h)      // Base case
  pthread_exit(NULL);


int mid = (args->l + args->h) / 2;
if(strcmp(Words[mid].self, args->keyword) == 0) {
  if(!duplicate_check)
    printf("[CONSUMER MODE](output): `---> %s means: %s\n
`---> Another meaning: %s\n", Words[mid].self, Words[mid].mean1,
Words[mid].mean2);
  found = 1;
  pthread_exit(NULL);
}


// Setting arguments to pass to both the right and left threads
pthread_t tid1, tid2;
struct block left, right;
left.l = args->l;
left.h = mid - 1;
strcpy(left.keyword, args->keyword);


right.l = mid + 1;
right.h = args->h;
strcpy(right.keyword, args->keyword);
```

```
 pthread_create(&tid1, NULL, runner, &left); // search 'keyword' in left
half

 pthread_create(&tid2, NULL, runner, &right); // search 'keyword' in right
half

 pthread_join(tid1, NULL);

 pthread_join(tid2, NULL);

 pthread_exit(NULL);

}


// prints the dictionary

void print_dict() {

 printf("**********The Dictionary********** \n");

 for (int i = 0; i < real_size; i++)

   printf("W: %s, mean1: %s, mean2: %s\n", Words[i].self, Words[i].mean1,
Words[i].mean2);

 printf("************************************************** \n");

}
```

- Output:


❯ gcc Q3.c -lpthread
❯ ./a.out
[PRODUCER MODE]: Input new data in the format: <word><space><primary-meaning><space><secondary-meaning>
: beautiful dazzling cute
[CONSUMER MODE]: Enter key to be searched(enter 'exit' to quit the search operation): hi
[CONSUMER MODE](error): [X] The word 'hi' not found
[PRODUCER MODE]: Input new data in the format: <word><space><primary-meaning><space><secondary-meaning>
: cunning skillful foxy
[CONSUMER MODE]: Enter key to be searched(enter 'exit' to quit the search operation): cunning
[CONSUMER MODE](output): `---> cunning means: skillful
`---> Another meaning: foxy
[PRODUCER MODE]: Input new data in the format: <word><space><primary-meaning><space><secondary-meaning>
: fast agile quick
[CONSUMER MODE]: Enter key to be searched(enter 'exit' to quit the search operation): beautiful
[CONSUMER MODE](output): `---> beautiful means: dazzling
`---> Another meaning: cute
[PRODUCER MODE]: Input new data in the format: <word><space><primary-meaning><space><secondary-meaning>
: fast rapid swift
[PRODUCER MODE](error): Duplicate entries not allowed.





Q5: Trace of Dekker's Solution to Critical Section Problem:

Problem Definition:

There are two threads T1 and T2. T1 is the writer thread which increments the shared variable Buffer by 1 and T2 is the reader thread which reads the variable Buffer.

Solution:

Clearly, the shared variable lies in the critical section of both the threads. Therefore we solve this problem using Dekker's solution.

The Basic Idea: Each of the threads:

- **START:** sets their flag to TRUE, expressing the desire to enter CS.
- while(other thread has desire to enter CS)
    - if(it is turn of other thread)
        - suppress the desire to enter CS
        - wait for the turn of other thread getover
        - After wait is over, express the desire to enter CS again
        - Since it is current thread's turn the outer while loop breaks
- Enter CS
- Set turn to other thread
- kill your old desire to enter CS, since the desire is fulfilled already.
- *goto* **START**

Initial Setup:

```
#define PRODUCER 0
#define CONSUMER 1
#define TRUE 0
#define FALSE 1
```

Along with Buffer, there will be two more shared data structures:

```
int flag[2];      // desire to enter CS
int turn = PRODUCER;
int Buffer = 0;

flag[PRODUCER] = FALSE;
flag[CONSUMER] = FALSE;
```

Thread 1 Pseudocode: (T1)

```
while(1) {
   // Entry Section
 1.    flag[PRODUCER] = TRUE;
 2.    while(flag[CONSUMER]) {
 3.      if(turn == CONSUMER) {
 4.        flag[PRODUCER] = FALSE;
 5.        while(turn == CONSUMER){
 6.          // busy wait
```

```
7.        }
8.        flag[PRODUCER] = TRUE;
9.      }
10.   }
11.   // Critical section
12.   Buffer++;
13.   // Exit section
14.   turn = CONSUMER;
15.   flag[PRODUCER] = FALSE;
16. }
```

Thread 2 Pseudocode: (T2)

```
while(1) {
   // Entry Section
1.    flag[CONSUMER] = TRUE;
2.    while(flag[PRODUCER]) {
3.      if(turn == PRODUCER) {
4.        flag[CONSUMER] = FALSE;
5.        while(turn == PRODUCER){
6.          // busy wait
7.        }
8.        flag[CONSUMER] = TRUE;
9.      }
10.   }
11.   // Critical section
12.   printf("%d", Buffer);
13.   // Exit section
14.   turn = PRODUCER;
15.   flag[CONSUMER] = FALSE;
16. }
```

Trace:

					flag[] values	
Serial #	Thread 1 (line # executed)	Thread 2 (line # executed)	Buffer	turn	PRODUCER	CONSUMER
0	-	-	0	PRODUCER	FALSE	FALSE
1	1	-	0	PRODUCER	TRUE	FALSE
2	-	1	0	PRODUCER	TRUE	TRUE
3	2 (stuck in outer while)	-	0	PRODUCER	TRUE	TRUE
4	stuck in outer while	2-3-4-5 (Busy waits)	0	PRODUCER	TRUE	FALSE
5	12(outer while breaks, enters the CS)	Busy waits	1	PRODUCER	TRUE	FALSE
6	14-15(exits the CS)	Busy waits	1	CONSUMER	FALSE	FALSE
7	-	8-9-10-11-12 (Exits the busy wait and enters CS)	1	CONSUMER	FALSE	TRUE
8	1, 2 (stuck in outer while)	-	1	CONSUMER	TRUE	TRUE
9	stuck in outer while	14-15(exits the CS)	1	PRODUCER	TRUE	FALSE
10	12(outer while breaks, enters the CS)	-	2	PRODUCER	TRUE	FALSE
11	-	1-2(stuck in outer while)	2	PRODUCER	TRUE	TRUE

12	14-15-16(exits the CS)	stuck in while	2	CONSUMER	FALSE	TRUE
	...					

and so on...

This is the way the two threads will be synchronised using Dekker's algorithm and the shared data Buffer will be shared safely among the two threads without letting the race condition or deadlock or starvation occur.

Thanks,

Animesh Kumar
CED18I065