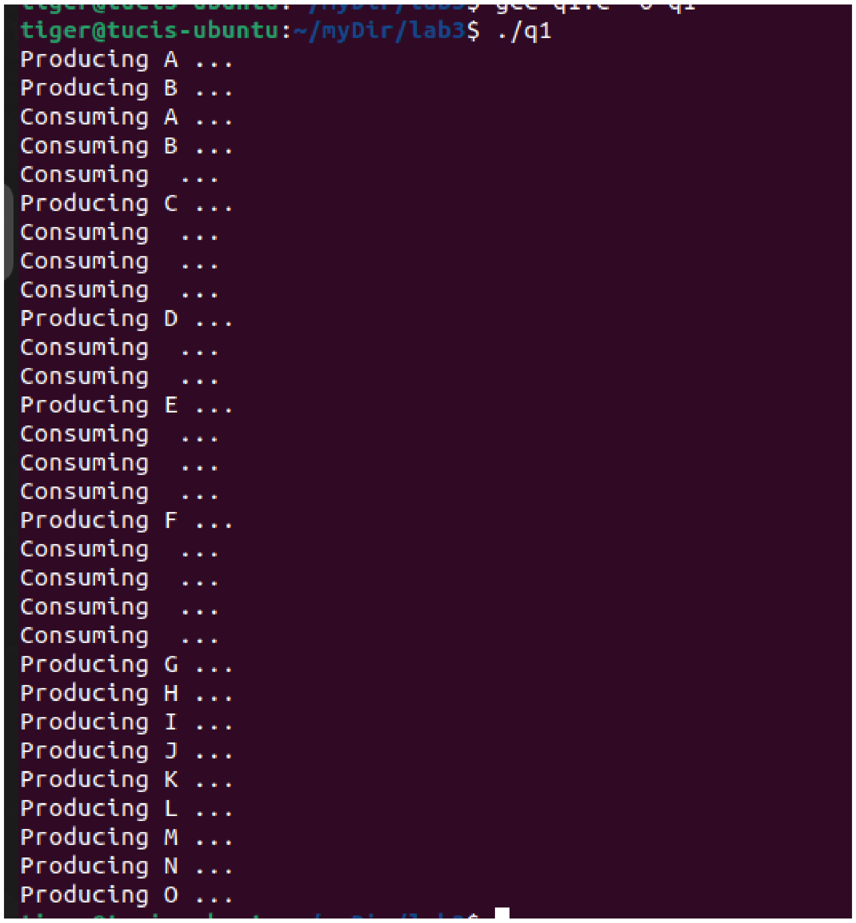
Synchronization Lab

1. Code output:



```
tiger@tucis-ubuntu:~/myDir/lab3$ ./q1
Producing A ...
Producing B ...
Consuming A ...
Consuming B ...
Consuming  ...
Producing C ...
Consuming  ...
Consuming  ...
Consuming  ...
Producing D ...
Consuming  ...
Consuming  ...
Producing E ...
Consuming  ...
Consuming  ...
Consuming  ...
Producing F ...
Consuming  ...
Consuming  ...
Consuming  ...
Consuming  ...
Producing G ...
Producing H ...
Producing I ...
Producing J ...
Producing K ...
Producing L ...
Producing M ...
Producing N ...
Producing O ...
```

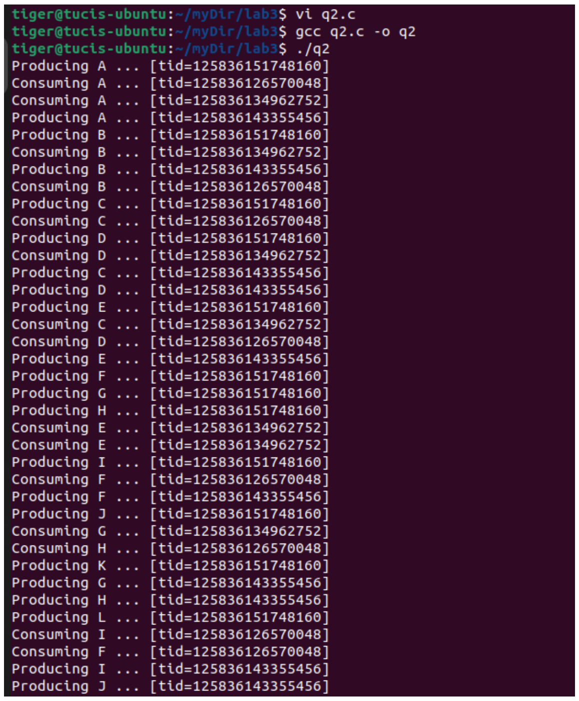
Each line comes from one of two threads:
Producing X ... = producer wrote a letter into the buffer.
Consuming X ... = consumer read a letter from the buffer.
The many lines that look like Consuming ... (with nothing between Consuming and the dots) mean the consumer read the NUL character '\0'. The global buffer is empty, so the consumer often runs before the producer writes to that slot and "consumes" an empty value. Printing '\0' with %c shows up as a blank. The odd ordering (sometimes A then B, sometimes several blanks consume in a row) is because the two threads sleep for random intervals and there's no coordination for scheduling. Near the end can only Producing G ..., Producing H ..., … Producing O ... with no matching consumes. That's because the consumer has already completed its fixed 15 reads and exited, while the producer is still finishing its 15 writes.

2. Code:
#include <pthread.h>
#include <semaphore.h>

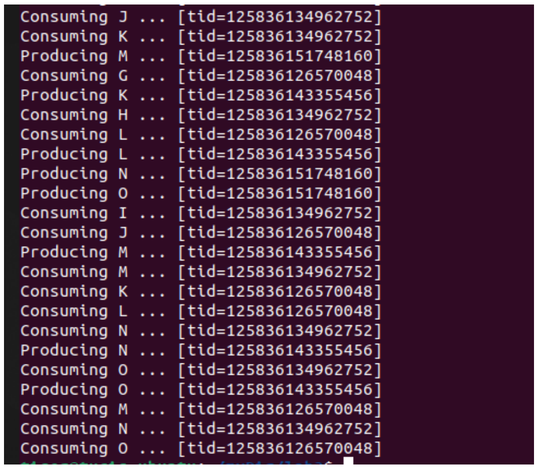
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#define BUFF_SIZE 20          // size of the circular buffer
#define NUM_PROD  2           // number of producer threads
#define NUM_CONS  2           // number of consumer threads
#define NITEMS_P  15          // items produced by each producer
#define NITEMS_C  15          // items consumed by each consumer (match totals)

static char buffer[BUFF_SIZE];
static int  nextIn  = 0;
static int  nextOut = 0;

static sem_t empty_slots;     // counts free slots in buffer
static sem_t full_slots;      // counts filled slots in buffer
static sem_t mutex;           // mutual exclusion on buffer

// Put one item into the buffer (blocks if full)
static void Put(char item)
{
   sem_wait(&empty_slots);           // wait for at least one empty slot
   sem_wait(&mutex);                 // enter critical section

   buffer[nextIn] = item;
   nextIn = (nextIn + 1) % BUFF_SIZE;

   sem_post(&mutex);                 // leave critical section
   sem_post(&full_slots);            // we just made one slot full

   printf("Producing %c ... [tid=%lu]\n", item, (unsigned long)pthread_self());
   fflush(stdout);
}

static void *Producer(void *arg)
{
   long id = (long)arg;
   (void)id;
   for (int i = 0; i < NITEMS_P; i++) {
      sleep(rand() % 6);            // random delay
      char item = (char)('A' + i % 26);
      Put(item);
   }
```

```
    return NULL;
}

// Get one item from the buffer (blocks if empty)
static char Get(void)
{
    sem_wait(&full_slots);            // wait for at least one full slot
    sem_wait(&mutex);                 // enter critical section

    char item = buffer[nextOut];
    nextOut = (nextOut + 1) % BUFF_SIZE;

    sem_post(&mutex);                 // leave critical section
    sem_post(&empty_slots);            // just freed one slot

    printf("Consuming %c ... [tid=%lu]\n", item, (unsigned long)pthread_self());
    fflush(stdout);
    return item;
}

static void *Consumer(void *arg)
{
    (void)arg;
    for (int i = 0; i < NITEMS_C; i++) {
        sleep(rand() % 6);            // random delay
        (void)Get();
    }
    return NULL;
}

int main(void)
{
    srand((unsigned)time(NULL));

    // initialize semaphores
    sem_init(&mutex,       0, 1);
    sem_init(&empty_slots, 0, BUFF_SIZE);
    sem_init(&full_slots,  0, 0);

    // start producers and consumers
    pthread_t prod[NUM_PROD], cons[NUM_CONS];
    for (long i = 0; i < NUM_PROD; i++)
        pthread_create(&prod[i], NULL, Producer, (void*)i);
    for (long i = 0; i < NUM_CONS; i++)
```

```
        pthread_create(&cons[i], NULL, Consumer, (void*)i);

    // wait for all threads
    for (int i = 0; i < NUM_PROD; i++) pthread_join(prod[i], NULL);
    for (int i = 0; i < NUM_CONS; i++) pthread_join(cons[i], NULL);

    // cleanup
    sem_destroy(&mutex);
    sem_destroy(&empty_slots);
    sem_destroy(&full_slots);

    return 0;
}
```
Output:

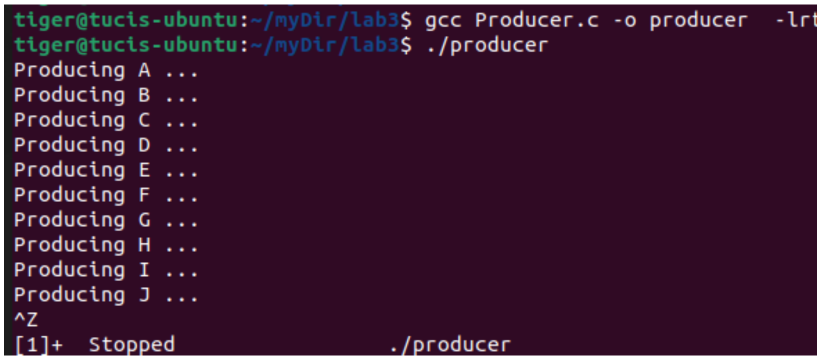


```
tiger@tucis-ubuntu:~/myDir/lab3$ vi q2.c
tiger@tucis-ubuntu:~/myDir/lab3$ gcc q2.c -o q2
tiger@tucis-ubuntu:~/myDir/lab3$ ./q2
Producing A ... [tid=125836151748160]
Consuming A ... [tid=125836126570048]
Consuming A ... [tid=125836134962752]
Producing A ... [tid=125836143355456]
Producing B ... [tid=125836151748160]
Consuming B ... [tid=125836134962752]
Producing B ... [tid=125836143355456]
Consuming B ... [tid=125836126570048]
Producing C ... [tid=125836151748160]
Consuming C ... [tid=125836126570048]
Producing D ... [tid=125836151748160]
Consuming D ... [tid=125836134962752]
Producing C ... [tid=125836143355456]
Producing D ... [tid=125836143355456]
Producing E ... [tid=125836151748160]
Consuming C ... [tid=125836134962752]
Consuming D ... [tid=125836126570048]
Producing E ... [tid=125836143355456]
Producing F ... [tid=125836151748160]
Producing G ... [tid=125836151748160]
Producing H ... [tid=125836151748160]
Consuming E ... [tid=125836134962752]
Consuming E ... [tid=125836134962752]
Producing I ... [tid=125836151748160]
Consuming F ... [tid=125836126570048]
Producing F ... [tid=125836143355456]
Producing J ... [tid=125836151748160]
Consuming G ... [tid=125836134962752]
Consuming H ... [tid=125836126570048]
Producing K ... [tid=125836151748160]
Producing G ... [tid=125836143355456]
Producing H ... [tid=125836143355456]
Producing L ... [tid=125836151748160]
Consuming I ... [tid=125836126570048]
Consuming F ... [tid=125836126570048]
Producing I ... [tid=125836143355456]
Producing J ... [tid=125836143355456]
```

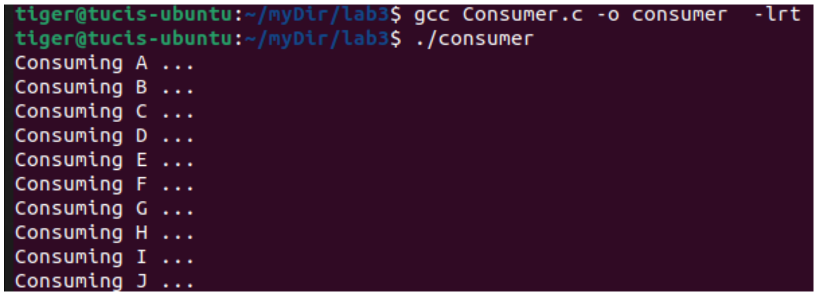


```
Consuming J ... [tid=125836134962752]
Consuming K ... [tid=125836134962752]
Producing M ... [tid=125836151748160]
Consuming G ... [tid=125836126570048]
Producing K ... [tid=125836143355456]
Consuming H ... [tid=125836134962752]
Consuming L ... [tid=125836126570048]
Producing L ... [tid=125836143355456]
Producing N ... [tid=125836151748160]
Producing O ... [tid=125836151748160]
Consuming I ... [tid=125836134962752]
Consuming J ... [tid=125836126570048]
Producing M ... [tid=125836143355456]
Consuming M ... [tid=125836134962752]
Consuming K ... [tid=125836126570048]
Consuming L ... [tid=125836126570048]
Consuming N ... [tid=125836134962752]
Producing N ... [tid=125836143355456]
Consuming O ... [tid=125836134962752]
Producing O ... [tid=125836143355456]
Consuming M ... [tid=125836126570048]
Consuming N ... [tid=125836134962752]
Consuming O ... [tid=125836126570048]
```

3. (1) output:



```
tiger@tucis-ubuntu:~/myDir/lab3$ gcc Producer.c -o producer  -lrt
tiger@tucis-ubuntu:~/myDir/lab3$ ./producer
Producing A ...
Producing B ...
Producing C ...
Producing D ...
Producing E ...
Producing F ...
Producing G ...
Producing H ...
Producing I ...
Producing J ...
^Z
[1]+  Stopped                 ./producer
```



```
tiger@tucis-ubuntu:~/myDir/lab3$ gcc Consumer.c -o consumer  -lrt
tiger@tucis-ubuntu:~/myDir/lab3$ ./consumer
Consuming A ...
Consuming B ...
Consuming C ...
Consuming D ...
Consuming E ...
Consuming F ...
Consuming G ...
Consuming H ...
Consuming I ...
Consuming J ...
```

(2) Code:

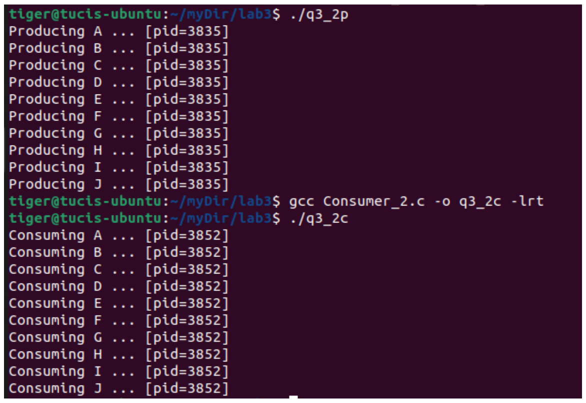
```
//Producer_2.c
#include <stdio.h>
#include <sys/types.h>
#include <semaphore.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/shm.h>

#define BUFF_SIZE 20


typedef struct {
char buffer[BUFF_SIZE];
int nextIn;
int nextOut;
} shared_data;

shared_data *shm, *s;
char sem_name1[] = "mutex";
char sem_name2[] = "empty_slots";
char sem_name3[] = "full_slots";
```

```
sem_t *empty_slots;
sem_t *full_slots;
sem_t *mutex;

void Put(char item)
{
  sem_wait(empty_slots);
  sem_wait(mutex);
  s->buffer[s->nextIn] = item;
  s->nextIn = (s->nextIn + 1) % BUFF_SIZE;
  sem_post(mutex);
  printf("Producing %c ... [pid=%d]\n", item, (int)getpid());
  sem_post(full_slots);
}

void Producer()
{
    int i;

    for(i = 0; i < 10; i++)
    {
      sleep(rand()%3);
      Put((char)('A'+ i % 26));
    }
}


void main()
{
        mutex=sem_open(sem_name1, O_CREAT,0644, 1);
        full_slots=sem_open(sem_name3, O_CREAT,0644, 0);
        empty_slots=sem_open(sem_name2, O_CREAT,0644, 10);

        //allocate the shared memory segment
        key_t key;
        key = 1234;
        //create the segment
        int shmid;

        if ((shmid = shmget(key, sizeof(shared_data), IPC_CREAT |0666)) <0)
        {
                perror("Shmget");
                exit(1);
        }
        //attach to the segment
        if ((shm = (shared_data *) shmat(shmid, NULL, 0))==(shared_data *) -1)
        {
                perror("Shmat");
                exit(1);
        }

        s=shm;
        s->nextIn = 0;

        Producer( );
    //detach
        shmdt((void *) shm);
}
// Consumer_2.c
#include<stdlib.h>
#include<unistd.h>
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/shm.h>

#define BUFF_SIZE 20

char buffer[BUFF_SIZE];
int nextIn = 0;
int nextOut = 0;

char sem_name1[] = "mutex";
char sem_name2[] = "empty_slots";
char sem_name3[] = "full_slots";
sem_t *empty_slots;
sem_t *full_slots;
sem_t *mutex;

typedef struct {
char buffer[BUFF_SIZE];
int nextIn;
int nextOut;
} shared_data;

shared_data *shm, *s;
```

```
void Get(char item)
{
 sem_wait(full_slots);
 sem_wait(mutex);
 item = s->buffer[s->nextOut];
 s->nextOut = (s->nextOut + 1) % BUFF_SIZE;
 sem_post(mutex);
 printf("Consuming %c ... [pid=%d]\n", item, (int)getpid());
 sem_post(empty_slots);
}

void Consumer()
{
   int i;
   char  item;
   for(i = 0; i < 10; i++)
   {
     sleep(rand()%9);
     Get(item);
   }
}

void main()
{
            mutex=sem_open(sem_name1, O_CREAT,0644, 1);
            full_slots=sem_open(sem_name3, O_CREAT,0644, 0);
            empty_slots=sem_open(sem_name2, O_CREAT,0644, 10);

            //allocate the shared memory segment
            key_t key;
            key = 1234;
            //locate the segment
            int shmid;
            if ((shmid = shmget(key, sizeof(shared_data),0666)) <0)
            {
                        perror("Shmget");
                        exit(1);
            }
            //attach to the segment
            if ((shm = (shared_data *) shmat(shmid, NULL, 0))==(shared_data *) -1)
            {
                        perror("Shmat");
                        exit(1);
```

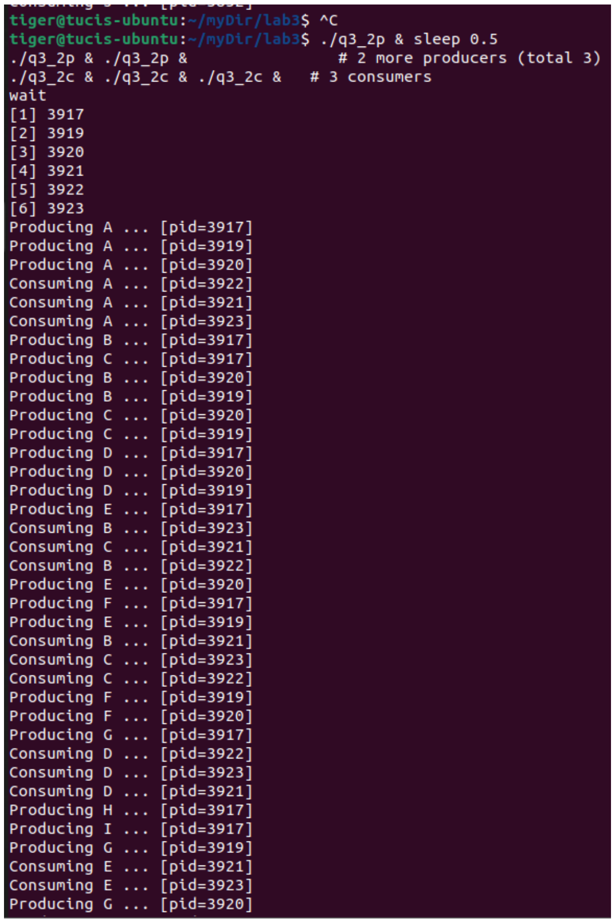
```
        }

        s=shm;
        s->nextOut = 0;
        Consumer();
        shmdt((void *) shm);

}
```

Output:

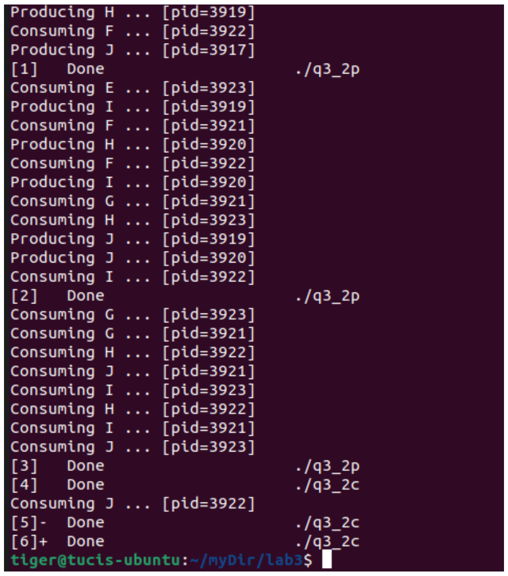


```
tiger@tucis-ubuntu:~/myDir/lab3$ ./q3_2p
Producing A ... [pid=3835]
Producing B ... [pid=3835]
Producing C ... [pid=3835]
Producing D ... [pid=3835]
Producing E ... [pid=3835]
Producing F ... [pid=3835]
Producing G ... [pid=3835]
Producing H ... [pid=3835]
Producing I ... [pid=3835]
Producing J ... [pid=3835]
tiger@tucis-ubuntu:~/myDir/lab3$ gcc Consumer_2.c -o q3_2c -lrt
tiger@tucis-ubuntu:~/myDir/lab3$ ./q3_2c
Consuming A ... [pid=3852]
Consuming B ... [pid=3852]
Consuming C ... [pid=3852]
Consuming D ... [pid=3852]
Consuming E ... [pid=3852]
Consuming F ... [pid=3852]
Consuming G ... [pid=3852]
Consuming H ... [pid=3852]
Consuming I ... [pid=3852]
Consuming J ... [pid=3852]
```

(3) Output:



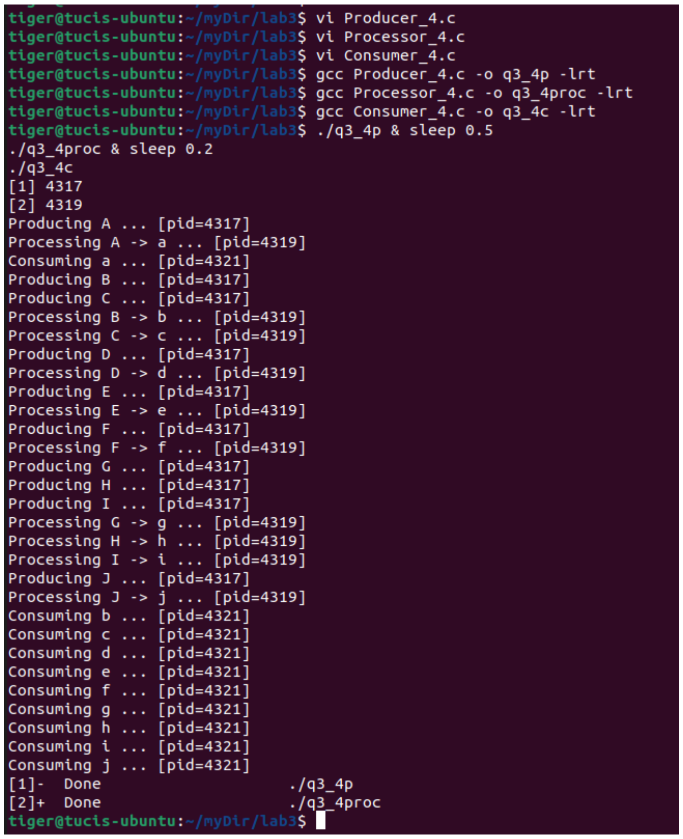
```
tiger@tucis-ubuntu:~/myDir/lab3$ ^C
tiger@tucis-ubuntu:~/myDir/lab3$ ./q3_2p & sleep 0.5
./q3_2p & ./q3_2p &                 # 2 more producers (total 3)
./q3_2c & ./q3_2c & ./q3_2c &   # 3 consumers
wait
[1] 3917
[2] 3919
[3] 3920
[4] 3921
[5] 3922
[6] 3923
Producing A ... [pid=3917]
Producing A ... [pid=3919]
Producing A ... [pid=3920]
Consuming A ... [pid=3922]
Consuming A ... [pid=3921]
Consuming A ... [pid=3923]
Producing B ... [pid=3917]
Producing C ... [pid=3917]
Producing B ... [pid=3920]
Producing B ... [pid=3919]
Producing C ... [pid=3920]
Producing C ... [pid=3919]
Producing D ... [pid=3917]
Producing D ... [pid=3920]
Producing D ... [pid=3919]
Producing E ... [pid=3917]
Consuming B ... [pid=3923]
Consuming C ... [pid=3921]
Consuming B ... [pid=3922]
Producing E ... [pid=3920]
Producing F ... [pid=3917]
Producing E ... [pid=3919]
Consuming B ... [pid=3921]
Consuming C ... [pid=3923]
Consuming C ... [pid=3922]
Producing F ... [pid=3919]
Producing F ... [pid=3920]
Producing G ... [pid=3917]
Consuming D ... [pid=3922]
Consuming D ... [pid=3923]
Consuming D ... [pid=3921]
Producing H ... [pid=3917]
Producing I ... [pid=3917]
Producing G ... [pid=3919]
Consuming E ... [pid=3921]
Consuming E ... [pid=3923]
Producing G ... [pid=3920]
```



```
Producing H ... [pid=3919]
Consuming F ... [pid=3922]
Producing J ... [pid=3917]
[1]   Done                    ./q3_2p
Consuming E ... [pid=3923]
Producing I ... [pid=3919]
Consuming F ... [pid=3921]
Producing H ... [pid=3920]
Consuming F ... [pid=3922]
Producing I ... [pid=3920]
Consuming G ... [pid=3921]
Consuming H ... [pid=3923]
Producing J ... [pid=3919]
Producing J ... [pid=3920]
Consuming I ... [pid=3922]
[2]   Done                    ./q3_2p
Consuming G ... [pid=3923]
Consuming G ... [pid=3921]
Consuming H ... [pid=3922]
Consuming J ... [pid=3921]
Consuming I ... [pid=3923]
Consuming H ... [pid=3922]
Consuming I ... [pid=3921]
Consuming J ... [pid=3923]
[3]   Done                    ./q3_2p
[4]   Done                    ./q3_2c
Consuming J ... [pid=3922]
[5]-  Done                    ./q3_2c
[6]+  Done                    ./q3_2c
tiger@tucis-ubuntu:~/myDir/lab3$ 
```

We can run several producer and consumer processes, and they will initially run and print interleaved lines with the shared memory and named semaphores allow cross-process coordination, but the current code will not run properly with many processes without fixes. Each new producer sets nextIn = 0 and each consumer sets nextOut = 0, so starting additional processes can reset these shared indices while others are active, corrupting the buffer. The semaphore that counts free slots is also initialized to 10 even though the buffer holds 20, which halves capacity and makes blocking more likely.

(4) Output:



```
tiger@tucis-ubuntu:~/myDir/lab3$ vi Producer_4.c
tiger@tucis-ubuntu:~/myDir/lab3$ vi Processor_4.c
tiger@tucis-ubuntu:~/myDir/lab3$ vi Consumer_4.c
tiger@tucis-ubuntu:~/myDir/lab3$ gcc Producer_4.c -o q3_4p -lrt
tiger@tucis-ubuntu:~/myDir/lab3$ gcc Processor_4.c -o q3_4proc -lrt
tiger@tucis-ubuntu:~/myDir/lab3$ gcc Consumer_4.c -o q3_4c -lrt
tiger@tucis-ubuntu:~/myDir/lab3$ ./q3_4p & sleep 0.5
./q3_4proc & sleep 0.2
./q3_4c
[1] 4317
[2] 4319
Producing A ... [pid=4317]
Processing A -> a ... [pid=4319]
Consuming a ... [pid=4321]
Producing B ... [pid=4317]
Producing C ... [pid=4317]
Processing B -> b ... [pid=4319]
Processing C -> c ... [pid=4319]
Producing D ... [pid=4317]
Processing D -> d ... [pid=4319]
Producing E ... [pid=4317]
Processing E -> e ... [pid=4319]
Producing F ... [pid=4317]
Processing F -> f ... [pid=4319]
Producing G ... [pid=4317]
Producing H ... [pid=4317]
Producing I ... [pid=4317]
Processing G -> g ... [pid=4319]
Processing H -> h ... [pid=4319]
Processing I -> i ... [pid=4319]
Producing J ... [pid=4317]
Processing J -> j ... [pid=4319]
Consuming b ... [pid=4321]
Consuming c ... [pid=4321]
Consuming d ... [pid=4321]
Consuming e ... [pid=4321]
Consuming f ... [pid=4321]
Consuming g ... [pid=4321]
Consuming h ... [pid=4321]
Consuming i ... [pid=4321]
Consuming j ... [pid=4321]
[1]-  Done                    ./q3_4p
[2]+  Done                    ./q3_4proc
tiger@tucis-ubuntu:~/myDir/lab3$ 
```

Code:

```
//Producer
#include <stdio.h>
#include <sys/types.h>
#include <semaphore.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/shm.h>

#define BUFF_SIZE 20

typedef struct {
   char buffer[BUFF_SIZE];
   int  nextIn;
   int  nextProc;
   int  nextOut;
} shared_data;

shared_data *shm, *s;

char sem_name_mutex[]   = "mutex";
char sem_name_empty[]   = "empty_slots";
char sem_name_prod[]    = "produced_slots";
char sem_name_ready[]   = "processed_slots";

sem_t *empty_slots;
sem_t *produced_slots;
sem_t *processed_slots;
sem_t *mutex;

static void Put(char item)
{
   sem_wait(empty_slots);
   sem_wait(mutex);

   s->buffer[s->nextIn] = item;
   s->nextIn = (s->nextIn + 1) % BUFF_SIZE;

   sem_post(mutex);
   printf("Producing %c ... [pid=%d]\n", item, (int)getpid());
   sem_post(produced_slots);                // item is now available for PROCESSOR
}
```

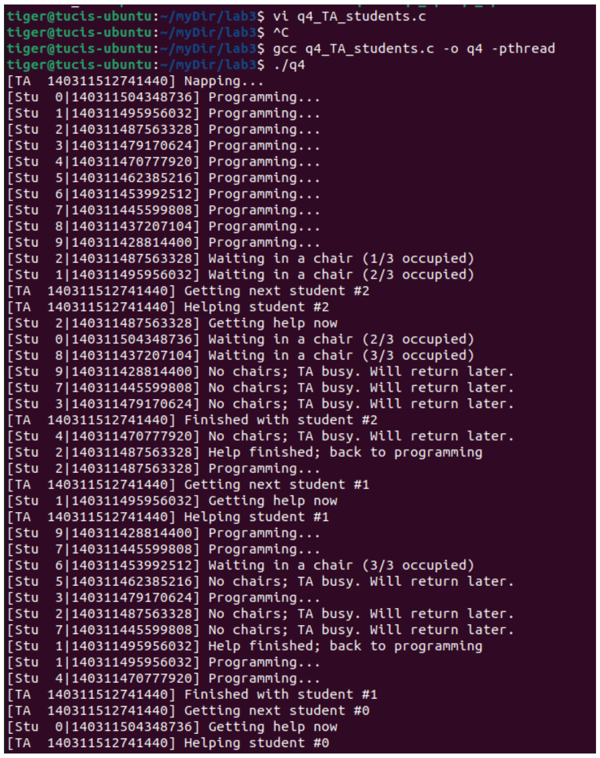
```
static void Producer(void)
{
    for (int i = 0; i < 10; i++) {
        sleep(rand() % 3);
        Put((char)('A' + (i % 26)));
    }
}

int main(void)
{
    // open/create semaphores
    mutex          = sem_open(sem_name_mutex, O_CREAT, 0644, 1);
    processed_slots= sem_open(sem_name_ready, O_CREAT, 0644, 0);
    produced_slots = sem_open(sem_name_prod,  O_CREAT, 0644, 0);
    empty_slots    = sem_open(sem_name_empty, O_CREAT, 0644, 10);

    // allocate the shared memory segment
    key_t key = 1234;
    int shmid;
    if ((shmid = shmget(key, sizeof(shared_data), IPC_CREAT | 0666)) < 0) {
        perror("Shmget");
        exit(1);
    }
    if ((shm = (shared_data *)shmat(shmid, NULL, 0)) == (shared_data *)-1) {
        perror("Shmat");
        exit(1);
    }

    s = shm;
    s->nextIn   = 0;
    // nextProc/nextOut set by the other processes

    Producer();

    // detach
    shmdt((void *)shm);
    return 0;
}
//Processor
#include <stdio.h>
#include <sys/types.h>
#include <semaphore.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/shm.h>
#include <ctype.h>

#define BUFF_SIZE 20

typedef struct {
   char buffer[BUFF_SIZE];
   int  nextIn;
   int  nextProc;
   int  nextOut;
} shared_data;

shared_data *shm, *s;

char sem_name_mutex[]   = "mutex";
char sem_name_empty[]   = "empty_slots";
char sem_name_prod[]    = "produced_slots";
char sem_name_ready[]   = "processed_slots";

sem_t *empty_slots;
sem_t *produced_slots;
sem_t *processed_slots;
sem_t *mutex;

static void ProcessOne(void)
{
   sem_wait(produced_slots);      // wait for item produced
   sem_wait(mutex);

   char in = s->buffer[s->nextProc];
   char out = (char)tolower((unsigned char)in);  // convert case
   s->buffer[s->nextProc] = out;                 // write back in place
   s->nextProc = (s->nextProc + 1) % BUFF_SIZE;

   sem_post(mutex);
   printf("Processing %c -> %c ... [pid=%d]\n", in, out, (int)getpid());
   sem_post(processed_slots);     // now ready for consumer
}

static void Processor(void)
{
   for (int i = 0; i < 10; i++) { // process same number as produced/consumed
       sleep(rand() % 3);
       ProcessOne();
   }
}

int main(void)
{
   // open existing semaphores (they are created by Producer)
   mutex           = sem_open(sem_name_mutex, O_CREAT, 0644, 1);
   processed_slots = sem_open(sem_name_ready, O_CREAT, 0644, 0);
   produced_slots  = sem_open(sem_name_prod,  O_CREAT, 0644, 0);
   empty_slots     = sem_open(sem_name_empty, O_CREAT, 0644, 10);

   // attach to shared memory created by Producer
   key_t key = 1234;
   int shmid;
   if ((shmid = shmget(key, sizeof(shared_data), 0666)) < 0) {
       perror("Shmget");
       exit(1);
   }
   if ((shm = (shared_data *)shmat(shmid, NULL, 0)) == (shared_data *)-1) {
       perror("Shmat");
       exit(1);
   }

   s = shm;
   s->nextProc = 0;

   Processor();

   shmdt((void *)shm);
   return 0;
}
// Consumer
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/shm.h>
```

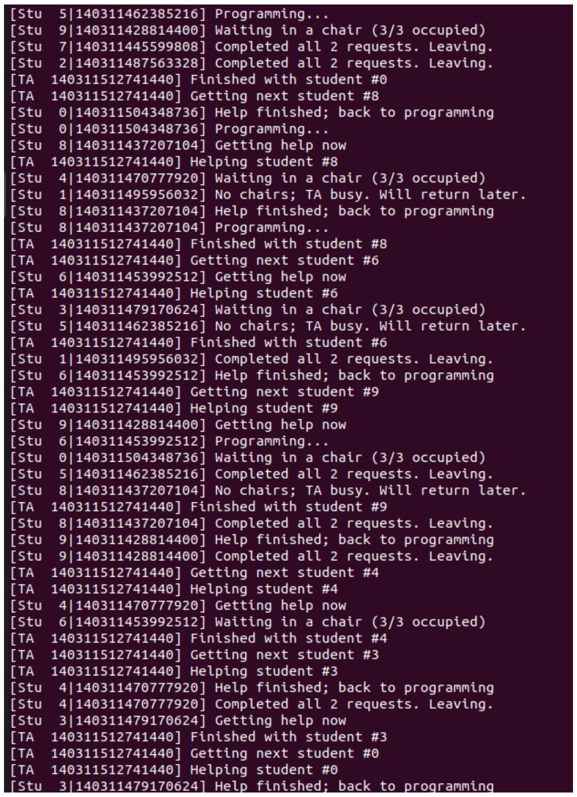
```
#define BUFF_SIZE 20

typedef struct {
   char buffer[BUFF_SIZE];
   int  nextIn;
   int  nextProc;
   int  nextOut;
} shared_data;

shared_data *shm, *s;

char sem_name_mutex[]   = "mutex";
char sem_name_empty[]   = "empty_slots";
char sem_name_prod[]    = "produced_slots";
char sem_name_ready[]   = "processed_slots";

sem_t *empty_slots;
sem_t *produced_slots;          // not used here but opened for completeness
sem_t *processed_slots;
sem_t *mutex;

static void Get(char item)
{
   sem_wait(processed_slots);          // wait until item has been processed
   sem_wait(mutex);

   item = s->buffer[s->nextOut];
   s->nextOut = (s->nextOut + 1) % BUFF_SIZE;

   sem_post(mutex);
   printf("Consuming %c ... [pid=%d]\n", item, (int)getpid());
   sem_post(empty_slots);              // slot becomes free again
}

static void Consumer(void)
{
   char item;
   for (int i = 0; i < 10; i++) {
      sleep(rand() % 9);
      Get(item);
   }
}

int main(void)
```

```
{
   mutex          = sem_open(sem_name_mutex, O_CREAT, 0644, 1);
   processed_slots = sem_open(sem_name_ready, O_CREAT, 0644, 0);
   produced_slots  = sem_open(sem_name_prod,  O_CREAT, 0644, 0);
   empty_slots     = sem_open(sem_name_empty, O_CREAT, 0644, 10);

   // attach to shared memory created by Producer
   key_t key = 1234;
   int shmid;
   if ((shmid = shmget(key, sizeof(shared_data), 0666)) < 0) {
      perror("Shmget");
      exit(1);
   }
   if ((shm = (shared_data *)shmat(shmid, NULL, 0)) == (shared_data *)-1) {
      perror("Shmat");
      exit(1);
   }

   s = shm;
   s->nextOut = 0;

   Consumer();

   shmdt((void *)shm);
   return 0;
}
```

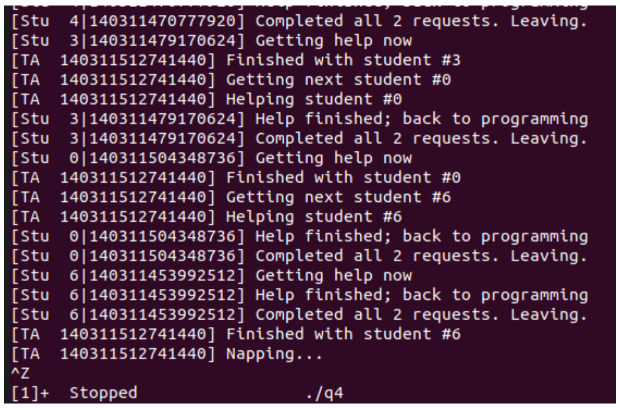
4. Output:



```
tiger@tucis-ubuntu:~/myDir/lab3$ vi q4_TA_students.c
tiger@tucis-ubuntu:~/myDir/lab3$ ^C
tiger@tucis-ubuntu:~/myDir/lab3$ gcc q4_TA_students.c -o q4 -pthread
tiger@tucis-ubuntu:~/myDir/lab3$ ./q4
[TA  140311512741440] Napping...
[Stu  0|140311504348736] Programming...
[Stu  1|140311495956032] Programming...
[Stu  2|140311487563328] Programming...
[Stu  3|140311479170624] Programming...
[Stu  4|140311470777920] Programming...
[Stu  5|140311462385216] Programming...
[Stu  6|140311453992512] Programming...
[Stu  7|140311445599808] Programming...
[Stu  8|140311437207104] Programming...
[Stu  9|140311428814400] Programming...
[Stu  2|140311487563328] Waiting in a chair (1/3 occupied)
[Stu  1|140311495956032] Waiting in a chair (2/3 occupied)
[TA  140311512741440] Getting next student #2
[TA  140311512741440] Helping student #2
[Stu  2|140311487563328] Getting help now
[Stu  0|140311504348736] Waiting in a chair (2/3 occupied)
[Stu  8|140311437207104] Waiting in a chair (3/3 occupied)
[Stu  9|140311428814400] No chairs; TA busy. Will return later.
[Stu  7|140311445599808] No chairs; TA busy. Will return later.
[Stu  3|140311479170624] No chairs; TA busy. Will return later.
[TA  140311512741440] Finished with student #2
[Stu  4|140311470777920] No chairs; TA busy. Will return later.
[Stu  2|140311487563328] Help finished; back to programming
[Stu  2|140311487563328] Programming...
[TA  140311512741440] Getting next student #1
[Stu  1|140311495956032] Getting help now
[TA  140311512741440] Helping student #1
[Stu  9|140311428814400] Programming...
[Stu  7|140311445599808] Programming...
[Stu  6|140311453992512] Waiting in a chair (3/3 occupied)
[Stu  5|140311462385216] No chairs; TA busy. Will return later.
[Stu  3|140311479170624] Programming...
[Stu  2|140311487563328] No chairs; TA busy. Will return later.
[Stu  7|140311445599808] No chairs; TA busy. Will return later.
[Stu  1|140311495956032] Help finished; back to programming
[Stu  1|140311495956032] Programming...
[Stu  4|140311470777920] Programming...
[TA  140311512741440] Finished with student #1
[TA  140311512741440] Getting next student #0
[Stu  0|140311504348736] Getting help now
[TA  140311512741440] Helping student #0
```



```
[Stu  5|140311462385216] Programming...
[Stu  9|140311428814400] Waiting in a chair (3/3 occupied)
[Stu  7|140311445599808] Completed all 2 requests. Leaving.
[Stu  2|140311487563328] Completed all 2 requests. Leaving.
[TA  140311512741440] Finished with student #0
[TA  140311512741440] Getting next student #8
[Stu  0|140311504348736] Help finished; back to programming
[Stu  0|140311504348736] Programming...
[Stu  8|140311437207104] Getting help now
[TA  140311512741440] Helping student #8
[Stu  4|140311470777920] Waiting in a chair (3/3 occupied)
[Stu  1|140311495956032] No chairs; TA busy. Will return later.
[Stu  8|140311437207104] Help finished; back to programming
[Stu  8|140311437207104] Programming...
[TA  140311512741440] Finished with student #8
[TA  140311512741440] Getting next student #6
[Stu  6|140311453992512] Getting help now
[TA  140311512741440] Helping student #6
[Stu  3|140311479170624] Waiting in a chair (3/3 occupied)
[Stu  5|140311462385216] No chairs; TA busy. Will return later.
[TA  140311512741440] Finished with student #6
[Stu  1|140311495956032] Completed all 2 requests. Leaving.
[Stu  6|140311453992512] Help finished; back to programming
[TA  140311512741440] Getting next student #9
[TA  140311512741440] Helping student #9
[Stu  9|140311428814400] Getting help now
[Stu  6|140311453992512] Programming...
[Stu  0|140311504348736] Waiting in a chair (3/3 occupied)
[Stu  5|140311462385216] Completed all 2 requests. Leaving.
[Stu  8|140311437207104] No chairs; TA busy. Will return later.
[TA  140311512741440] Finished with student #9
[Stu  8|140311437207104] Completed all 2 requests. Leaving.
[Stu  9|140311428814400] Help finished; back to programming
[Stu  9|140311428814400] Completed all 2 requests. Leaving.
[TA  140311512741440] Getting next student #4
[TA  140311512741440] Helping student #4
[Stu  4|140311470777920] Getting help now
[Stu  6|140311453992512] Waiting in a chair (3/3 occupied)
[TA  140311512741440] Finished with student #4
[TA  140311512741440] Getting next student #3
[TA  140311512741440] Helping student #3
[Stu  4|140311470777920] Help finished; back to programming
[Stu  4|140311470777920] Completed all 2 requests. Leaving.
[Stu  3|140311479170624] Getting help now
[TA  140311512741440] Finished with student #3
[TA  140311512741440] Getting next student #0
[TA  140311512741440] Helping student #0
[Stu  3|140311479170624] Help finished; back to programming
```



```
[Stu  4|140311470777920] Completed all 2 requests. Leaving.
[Stu  3|140311479170624] Getting help now
[TA  140311512741440] Finished with student #3
[TA  140311512741440] Getting next student #0
[TA  140311512741440] Helping student #0
[Stu  3|140311479170624] Help finished; back to programming
[Stu  3|140311479170624] Completed all 2 requests. Leaving.
[Stu  0|140311504348736] Getting help now
[TA  140311512741440] Finished with student #0
[TA  140311512741440] Getting next student #6
[TA  140311512741440] Helping student #6
[Stu  0|140311504348736] Help finished; back to programming
[Stu  0|140311504348736] Completed all 2 requests. Leaving.
[Stu  6|140311453992512] Getting help now
[Stu  6|140311453992512] Help finished; back to programming
[Stu  6|140311453992512] Completed all 2 requests. Leaving.
[TA  140311512741440] Finished with student #6
[TA  140311512741440] Napping...
^Z
[1]+  Stopped                 ./q4
```

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
#include <time.h>

#define NUM_STUDENTS        10
#define NUM_CHAIRS          3
#define REQS_PER_STUDENT    2   // how many times each student asks for help

//  waiting-room state (protected by mutex)
static int queue[NUM_CHAIRS];   // circular queue of waiting student IDs
static int q_head = 0, q_tail = 0, q_count = 0;

//semaphores
static sem_t mutex;             // protects queue and counters
static sem_t students_sem;      // counts waiting students (wakes TA)
static sem_t help_sem[NUM_STUDENTS]; // student waits here until TA summons
static sem_t done_sem[NUM_STUDENTS]; // student waits here until TA is done

// push a student ID into the circular queue requires mutex held
static void q_push(int id) {
    queue[q_tail] = id;
    q_tail = (q_tail + 1) % NUM_CHAIRS;
    q_count++;
}
// pop the next student ID from the circular queue (requires mutex held)
static int q_pop(void) {
    int id = queue[q_head];
    q_head = (q_head + 1) % NUM_CHAIRS;
    q_count--;
    return id;
}

// TA thread
static void* ta_thread(void* arg) {
    (void)arg;
    const unsigned long tid = (unsigned long)pthread_self();

    // total number of help sessions must complete for a clean shutdown
    const int total_sessions = NUM_STUDENTS * REQS_PER_STUDENT;
    int served = 0;

    while (served < total_sessions) {
        // If nobody is waiting, block (nap) until a student arrives.
        if (q_count == 0) {
            printf("[TA  %lu] Napping...\n", tid);
            fflush(stdout);
        }
        sem_wait(&students_sem);     // woken up when a student posts here

        // Take next student from the waiting room critical section.
        sem_wait(&mutex);
        int id = q_pop();
        sem_post(&mutex);

        printf("[TA  %lu] Getting next student #%d\n", tid, id);
        fflush(stdout);

        // Call the student into the office.
        sem_post(&help_sem[id]);

        // Help the student for 1–2 seconds at random.
        printf("[TA  %lu] Helping student #%d\n", tid, id);
        fflush(stdout);
        sleep((rand() % 2) + 1);     // 1..2 seconds

        // Tell the student we are done.
        sem_post(&done_sem[id]);
        printf("[TA  %lu] Finished with student #%d\n", tid, id);
        fflush(stdout);

        served++;
    }
    printf("[TA  %lu] Office hours over. Done for the day.\n", tid);
    fflush(stdout);
    return NULL;
}

// Student threads
static void* student_thread(void* arg) {
    int id = (int)(long)arg;
    const unsigned long tid = (unsigned long)pthread_self();

    for (int r = 0; r < REQS_PER_STUDENT; r++) {
        // Programing for 1–3 seconds before needing help.
        printf("[Stu %2d|%lu] Programming...\n", id, tid);
        fflush(stdout);
        sleep((rand() % 3) + 1);     // 1..3 seconds

        // Try to sit in the hallway critical section for chair count/queue.
        sem_wait(&mutex);
        if (q_count < NUM_CHAIRS) {
            q_push(id);
            printf("[Stu %2d|%lu] Waiting in a chair (%d/%d occupied)\n",
                   id, tid, q_count, NUM_CHAIRS);
            fflush(stdout);

            // Notify TA that a student is waiting.
            sem_post(&students_sem);
            sem_post(&mutex);

            // Wait until TA calls me in.
            sem_wait(&help_sem[id]);
            printf("[Stu %2d|%lu] Getting help now\n", id, tid);
            fflush(stdout);

            // Wait until TA finishes.
            sem_wait(&done_sem[id]);
            printf("[Stu %2d|%lu] Help finished; back to programming\n", id, tid);
            fflush(stdout);
        } else {
            // No chair—leave and try again later.
            printf("[Stu %2d|%lu] No chairs; TA busy. Will return later.\n", id, tid);
            fflush(stdout);
            sem_post(&mutex);
            // brief pause before trying again on the next loop iteration
            sleep(1);
        }
    }

    printf("[Stu %2d|%lu] Completed all %d requests. Leaving.\n",
           id, tid, REQS_PER_STUDENT);
    fflush(stdout);
    return NULL;
}

int main(void) {
    srand((unsigned)time(NULL));

    // Init semaphores
    sem_init(&mutex, 0, 1);
    sem_init(&students_sem, 0, 0);
    for (int i = 0; i < NUM_STUDENTS; i++) {
        sem_init(&help_sem[i], 0, 0);
        sem_init(&done_sem[i], 0, 0);
    }

    // Create threads
    pthread_t ta;
    pthread_t students[NUM_STUDENTS];
    pthread_create(&ta, NULL, ta_thread, NULL);
    for (int i = 0; i < NUM_STUDENTS; i++) {
        pthread_create(&students[i], NULL, student_thread, (void*)(long)i);
    }

    // Join threads
    for (int i = 0; i < NUM_STUDENTS; i++) pthread_join(students[i], NULL);
    pthread_join(ta, NULL);

    // Cleanup
    sem_destroy(&mutex);
    sem_destroy(&students_sem);
    for (int i = 0; i < NUM_STUDENTS; i++) {
        sem_destroy(&help_sem[i]);
        sem_destroy(&done_sem[i]);
    }
    return 0;
}
```