Name: Arunima Chaudhuri
Roll no: 23CSM1R04

Producer Consumer Example

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define QUEUESIZE 10
#define LOOP 20
void *producer (void *args);
void *consumer (void *args);
typedef struct {
int buf[QUEUESIZE];
long head, tail;
int full, empty;
pthread_mutex_t *mut;
pthread_cond_t *notFull, *notEmpty;
} queue;
queue *queueInit (void);
void queueDelete (queue *q);
void queueAdd (queue *q, int in);
void queueDel (queue *q, int *out);
int main ()
{
queue *fifo;
pthread_t pro, con;
fifo = queueInit ();
if (fifo ==  NULL) {
fprintf (stderr, "main: Queue Init failed.\n");
exit (1);
}
pthread_create (&pro, NULL, producer, fifo);
pthread_create (&con, NULL, consumer, fifo);
pthread_join (pro, NULL);
pthread_join (con, NULL);
queueDelete (fifo);
return 0;
}
void *producer (void *q)
{
queue *fifo;
int i;
fifo = (queue *)q;
for (i = 0; i < LOOP; i++) {
```

```
pthread_mutex_lock (fifo->mut);
while (fifo->full) {
printf ("producer: queue FULL.\n");
pthread_cond_wait (fifo->notFull, fifo->mut);
}
queueAdd (fifo, i);
pthread_mutex_unlock (fifo->mut);
pthread_cond_signal (fifo->notEmpty);
usleep (100000);
}
for (i = 0; i < LOOP; i++) {
pthread_mutex_lock (fifo->mut);
while (fifo->full) {
printf ("producer: queue FULL.\n");
pthread_cond_wait (fifo->notFull, fifo->mut);
}
queueAdd (fifo, i);
pthread_mutex_unlock (fifo->mut);
pthread_cond_signal (fifo->notEmpty);
usleep (200000);
}
return (NULL);
}
void *consumer (void *q)
{
queue *fifo;
int i, d;
fifo = (queue *)q;
for (i = 0; i < LOOP; i++) {
pthread_mutex_lock (fifo->mut);
while (fifo->empty) {
printf ("consumer: queue EMPTY.\n");
pthread_cond_wait (fifo->notEmpty, fifo->mut);
}
queueDel (fifo, &d);
pthread_mutex_unlock (fifo->mut);
pthread_cond_signal (fifo->notFull);
printf ("consumer: recieved %d.\n", d);
usleep(200000);
}
for (i = 0; i < LOOP; i++) {
pthread_mutex_lock (fifo->mut);
while (fifo->empty) {
printf ("consumer: queue EMPTY.\n");
pthread_cond_wait (fifo->notEmpty, fifo->mut);
}
```

```
queueDel (fifo, &d);
pthread_mutex_unlock (fifo->mut);
pthread_cond_signal (fifo->notFull);
printf ("consumer: recieved %d.\n", d);
usleep (50000);
}
return (NULL);
}
#ifdef 0
typedef struct {
int buf[QUEUESIZE];
long head, tail;
int full, empty;
pthread_mutex_t *mut;
pthread_cond_t *notFull, *notEmpty;
} queue;
#endif
queue *queueInit (void)
{
queue *q;
q = (queue *)malloc (sizeof (queue));
if (q == NULL) return (NULL);
q->empty = 1;
q->full = 0;
q->head = 0;
q->tail = 0;
q->mut = (pthread_mutex_t *) malloc (sizeof (pthread_mutex_t));
pthread_mutex_init (q->mut, NULL);
q->notFull = (pthread_cond_t *) malloc (sizeof (pthread_cond_t));
pthread_cond_init (q->notFull, NULL);
q->notEmpty = (pthread_cond_t *) malloc (sizeof (pthread_cond_t));
pthread_cond_init (q->notEmpty, NULL);

return (q);
}
void queueDelete (queue *q)
{
pthread_mutex_destroy (q->mut);
free (q->mut);
pthread_cond_destroy (q->notFull);
free (q->notFull);
pthread_cond_destroy (q->notEmpty);
free (q->notEmpty);
free (q);
}
void queueAdd (queue *q, int in)
```

```
{
q->buf[q->tail] = in;
q->tail++;
if (q->tail == QUEUESIZE)
q->tail = 0;
if (q->tail == q->head)
q->full = 1;
q->empty = 0;
return;
}
void queueDel (queue *q, int *out)
{
*out = q->buf[q->head];
q->head++;
if (q->head == QUEUESIZE)
q->head = 0;
if (q->head == q->tail)
q->empty = 1;
q->full = 0;
return;
}
```

Output

```
producer: recieved 0.
consumer: recieved 0.
producer: recieved 1.
consumer: recieved 1.
producer: recieved 2.
consumer: recieved 2.
producer: recieved 3.
consumer: recieved 3.
```

Readers-Writers Problem

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define MAXCOUNT 5
#define READER1  50000
#define READER2 100000
#define READER3400000
#define READER4 800000
```

```
#define WRITER1  150000
typedef struct {
pthread_mutex_t *mut;
int writers;
int readers;
int waiting;
pthread_cond_t *writeOK, *readOK;
} rwl;
rwl *initlock (void);
void readlock (rwl *lock, int d);
void writelock (rwl *lock, int d);
void readunlock (rwl *lock);
void writeunlock (rwl *lock);
void deletelock (rwl *lock);
typedef struct {
rwl *lock;
int id;
long delay;
} rwargs;
rwargs *newRWargs (rwl *l, int i, long d);
void *reader (void *args);
void *writer (void *args);
static int data = 1;
int main ()
{
pthread_t r1, r2, r3, r4, w1;
rwargs *a1, *a2, *a3, *a4, *a5;
rwl *lock;
lock = initlock ();
a1 = newRWargs (lock, 1, WRITER1);
pthread_create (&w1, NULL, writer, a1);
a2 = newRWargs (lock, 1, READER1);
pthread_create (&r1, NULL, reader, a2);
a3 = newRWargs (lock, 2, READER2);
pthread_create (&r2, NULL, reader, a3);
a4 = newRWargs (lock, 3, READER3);
pthread_create (&r3, NULL, reader, a4);
a5 = newRWargs (lock, 4, READER4);
pthread_create (&r4, NULL, reader, a5);
pthread_join (w1, NULL);
pthread_join (r1, NULL);
pthread_join (r2, NULL);
pthread_join (r3, NULL);
pthread_join (r4, NULL);
free (a1); free (a2); free (a3); free (a4); free (a5);
return 0;
```

```
}

rwargs *newRWargs (rwl *l, int i, long d)
{
rwargs *args;
args = (rwargs *)malloc (sizeof (rwargs));
if (args == NULL) return (NULL);
args->lock = l; args->id = i; args->delay = d;
return (args);
}
void *reader (void *args)
{
rwargs *a;
int d;
a = (rwargs *)args;
do {
readlock (a->lock, a->id);
d = data;
usleep (a->delay);
readunlock (a->lock);
printf ("Reader %d : Data = %d\n", a->id, d);
usleep (a->delay);
} while (d != 0);
printf ("Reader %d: Finished.\n", a->id);
return (NULL);
}
void *writer (void *args)
{
rwargs *a;
int i;
a = (rwargs *)args;
for (i = 2; i < MAXCOUNT; i++) {
writelock (a->lock, a->id);
data = i;
usleep (a->delay);
writeunlock (a->lock);
printf ("Writer %d: Wrote %d\n", a->id, i);
usleep (a->delay);
}
printf ("Writer %d: Finishing...\n", a->id);
writelock (a->lock, a->id);
data = 0;
writeunlock (a->lock);
printf ("Writer %d: Finished.\n", a->id);
return (NULL);
}
```

```
rwl *initlock (void)
{
rwl *lock;
lock = (rwl *)malloc (sizeof (rwl));
if (lock == NULL) return (NULL);
lock->mut = (pthread_mutex_t *) malloc (sizeof (pthread_mutex_t));
if (lock->mut == NULL) { free (lock); return (NULL); }
lock->writeOK = (pthread_cond_t *) malloc (sizeof (pthread_cond_t));
if (lock->writeOK == NULL) { free (lock->mut); free (lock); return
(NULL); }
lock->readOK =
(pthread_cond_t *) malloc (sizeof (pthread_cond_t));
if (lock->writeOK == NULL) {
free (lock->mut); free (lock->writeOK);
free (lock); return (NULL);
}

pthread_mutex_init (lock->mut, NULL);
pthread_cond_init (lock->writeOK, NULL);
pthread_cond_init (lock->readOK, NULL);
lock->readers = 0;
lock->writers = 0;
lock->waiting = 0;
return (lock);
}
void readlock (rwl *lock, int d)
{
pthread_mutex_lock (lock->mut);
if (lock->writers || lock->waiting) {
do {
printf ("reader %d blocked.\n", d);
pthread_cond_wait (lock->readOK, lock->mut);
printf ("reader %d unblocked.\n", d);
} while (lock->writers);
}
lock->readers++;
pthread_mutex_unlock (lock->mut);
return;
}
void writelock (rwl *lock, int d)
{
pthread_mutex_lock (lock->mut);
lock->waiting++;
while (lock->readers || lock->writers) {
printf ("writer %d blocked.\n", d);
pthread_cond_wait (lock->writeOK, lock->mut);
```

```
printf ("writer %d unblocked.\n", d);
}
lock->waiting--;
lock->writers++;
pthread_mutex_unlock (lock->mut);
return;
}
void readunlock (rwl *lock)
{
pthread_mutex_lock (lock->mut);
lock->readers--;
pthread_cond_signal (lock->writeOK);
pthread_mutex_unlock (lock->mut);
}
void writeunlock (rwl *lock)
{
pthread_mutex_lock (lock->mut);
lock->writers--;
pthread_cond_broadcast (lock->readOK);
pthread_mutex_unlock (lock->mut);
}
void deletelock (rwl *lock)
{
pthread_mutex_destroy (lock->mut);
pthread_cond_destroy (lock->readOK);
pthread_cond_destroy (lock->writeOK);
free (lock);
return;
}
```

Output

```
reader 1 blocked.
writer 1 blocked.
reader 1 unblocked.
Reader 1: Data = 2
reader 1 blocked.
writer 1 unblocked.
Reader 1: Data = 2
Reader 1: Finished.
Writer 1: Wrote 2
Writer 1: Wrote 3
reader 2 blocked.
writer 1 blocked.
Reader 2 : Data = 3
```

```
Reader 2 : Data = 3
Reader 2 : Finished.
writer 1 unblocked.
Reader 1: Data = 3
Reader 1: Data = 3
Reader 1: Finished.
writer 1 blocked.
writer 1 unblocked.
Writer 1: Finished.
Reader 3 : Data = 3
Reader 3 : Data = 3
Reader 3 : Finished.
reader 2 blocked.
reader 4 blocked.
reader 2 unblocked.
reader 4 unblocked.
Reader 2 : Data = 3
Reader 4 : Data = 3
Reader 4 : Data = 3
Reader 2 : Data = 3
Reader 4 : Finished.
Reader 2 : Finished.
```

Sleeping Barber

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

#define NUM_CHAIRS 5

int num_customers = 20;  // Number of customers to simulate

sem_t customers;
sem_t barber;
sem_t access_seats;
int waiting = 0;

void *barber_thread(void *arg) {
    while (1) {
        sem_wait(&customers);
```

```
        sem_wait(&access_seats);
        waiting--;
        sem_post(&barber);
        sem_post(&access_seats);

        // Cut hair (service the customer)
        printf("Barber is cutting hair.\n");
        sleep(1);

        printf("Barber has finished cutting hair.\n");
    }
}

void *customer_thread(void *arg) {
    int id = *((int *)arg);
    sleep(1);  // Customers arrive at random times.

    sem_wait(&access_seats);
    if (waiting < NUM_CHAIRS) {
        printf("Customer %d sits down in the waiting room.\n", id);
        waiting++;
        sem_post(&customers);
        sem_post(&access_seats);
        sem_wait(&barber);
        printf("Customer %d is getting a haircut.\n", id);
    } else {
        printf("Customer %d leaves because the waiting room is full.\n",
id);
        sem_post(&access_seats);
    }
}

int main() {
    pthread_t barber_t;
    pthread_t customers_t[num_customers];
    int customer_ids[num_customers];

    sem_init(&customers, 0, 0);
    sem_init(&barber, 0, 0);
    sem_init(&access_seats, 0, 1);

    pthread_create(&barber_t, NULL, barber_thread, NULL);

    for (int i = 0; i < num_customers; i++) {
        customer_ids[i] = i;
        pthread_create(&customers_t[i], NULL, customer_thread,
```

```
&customer_ids[i]);
    }

    for (int i = 0; i < num_customers; i++) {
        pthread_join(customers_t[i], NULL);
    }

    // Terminate the barber thread after all customers are done
    pthread_cancel(barber_t);
    pthread_join(barber_t, NULL);

    sem_destroy(&customers);
    sem_destroy(&barber);
    sem_destroy(&access_seats);

    return 0;
}
```

Output

```
Customer 0 sits down in the waiting room.
Barber is cutting hair.
Customer 1 sits down in the waiting room.
Barber has finished cutting hair.
Customer 2 sits down in the waiting room.
Barber is cutting hair.
Customer 3 sits down in the waiting room.
Barber has finished cutting hair.
Customer 4 sits down in the waiting room.
Barber is cutting hair.
Customer 5 leaves because the waiting room is full.
Barber has finished cutting hair.
Customer 6 sits down in the waiting room.
Barber is cutting hair.
Customer 7 sits down in the waiting room.
Barber has finished cutting hair.
Customer 8 sits down in the waiting room.
Barber is cutting hair.
Customer 9 leaves because the waiting room is full.
Barber has finished cutting hair.
Customer 10 sits down in the waiting room.
Barber is cutting hair.
Customer 11 sits down in the waiting room.
Barber has finished cutting hair.
Customer 12 sits down in the waiting room.
Barber is cutting hair.
```

```
Customer 13 leaves because the waiting room is full.
Barber has finished cutting hair.
Customer 14 sits down in the waiting room.
Barber is cutting hair.
Customer 15 leaves because the waiting room is full.
Barber has finished cutting hair.
Customer 16 sits down in the waiting room.
Barber is cutting hair.
Customer 17 leaves because the waiting room is full.
Barber has finished cutting hair.
Customer 18 sits down in the waiting room.
Barber is cutting hair.
Customer 19 leaves because the waiting room is full.
Barber has finished cutting hair.
```

Dining Philosophers

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_PHILOSOPHERS 5
#define EATING_TIME 2
#define THINKING_TIME 1

pthread_mutex_t forks[NUM_PHILOSOPHERS];

void *philosopher(void *arg) {
    int id = *(int *)arg;
    int left_fork = id;
    int right_fork = (id + 1) % NUM_PHILOSOPHERS;

    while (1) {
        // Thinking
        printf("Philosopher %d is thinking\n", id);
        sleep(THINKING_TIME);

        // Pick up forks
        pthread_mutex_lock(&forks[left_fork]);
        pthread_mutex_lock(&forks[right_fork);

        // Eating
        printf("Philosopher %d is eating\n", id);
        sleep(EATING_TIME);
```

```
        // Release forks
        pthread_mutex_unlock(&forks[left_fork]);
        pthread_mutex_unlock(&forks[right_fork);
    }
    return NULL;
}

int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int philosopher_ids[NUM_PHILOSOPHERS];

    // Initialize forks (mutexes)
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_mutex_init(&forks[i], NULL);
    }

    // Create philosopher threads
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        philosopher_ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher,
&philosopher_ids[i]);
    }

    // Join philosopher threads (this will never be reached as they run
forever)
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosophers[i], NULL);
    }

    // Destroy forks (mutexes)
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_mutex_destroy(&forks[i]);
    }

    return 0;
}
```

Output

```
Philosopher 0 is thinking
Philosopher 3 is thinking
Philosopher 1 is thinking
Philosopher 4 is thinking
Philosopher 2 is thinking
```

```
Philosopher 0 is eating
Philosopher 4 is eating
Philosopher 1 is eating
Philosopher 2 is eating
Philosopher 3 is eating
Philosopher 0 is thinking
Philosopher 1 is thinking
Philosopher 4 is thinking
Philosopher 3 is thinking
Philosopher 2 is thinking
Philosopher 0 is eating
```