Lab 3 Report

Ramandeep Farmaha 20516974
Krishn Ramesh 20521942

Multi Process Method

N	B	P	C	Average Time (s)
100	4	1	1	0.000453
100	4	1	2	0.001289
100	4	1	3	0.001558
100	4	2	1	0.001650
100	4	3	1	0.001799
100	8	1	1	0.001354
100	8	1	2	0.001429
100	8	1	3	0.001487
100	8	2	1	0.001565
100	8	3	1	0.001745
398	8	1	1	0.002034
398	8	1	2	0.002375
398	8	1	3	0.002457
398	8	2	1	0.002576
398	8	3	1	0.002594

Multithreaded Method

N	B	P	C	Average Time (s)
100	4	1	1	0.000967
100	4	1	2	0.000901
100	4	1	3	0.001083
100	4	2	1	0.000982
100	4	3	1	0.001043
100	8	1	1	0.000638
100	8	1	2	0.000905
100	8	1	3	0.001064
100	8	2	1	0.000876
100	8	3	1	0.000346
398	8	1	1	0.000374
398	8	1	2	0.000481
398	8	1	3	0.000556
398	8	2	1	0.000628
398	8	3	1	0.002192

Specific Tuple (N = 398, B = 8, P = 1, C = 3)

Program ran 500 times

Multi Process Method:

Average Time (s): 0.002457
Standard Deviation (s): 0.000342

Multithreaded Method:

Average Time (s): 0.000556
Standard Deviation (s): 0.000068

The multithreaded method is significantly faster than the message queue method due to the lesser overhead of creating threads than processes, which also require maintaining their own individual resources and states. The multithreaded model is also much quicker because threads can directly communicate with each other, without requiring any additional data structures or mechanisms to transfer memory to one another.

Multi-process with message queues, albeit slower, decouple the producer and consumer logic, and are more stable than the multithreaded model. For example, in the multithreaded model if a producer thread seg faults, the entire process could shutdown. However, in the message queue model, if a producer thread seg faults, its process might shut down, but the other producer processes could take its place. Decoupling also allows for multiple servers to produce and consume messages, enabling parallelization across machines.

Appendix A - producer.c for Multi Process Method

```
/**
* @file:   produce.c
* @brief:  Implementing the producer-consumer paradigm with multiple processes and message queues.
* @date:   2017/11/22
* NOTES:
   Compile with: gcc produce.c -lrt -lm -o produce
*/


#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <time.h>
#include <stdbool.h>
#include <math.h>

const char* Q_NAME = "/sqrt_q";

int N; // Amount of numbers
int B; // Buffer size
int P; // Number of producers
int C; // Number of consumers

void producer(int id);
void consumer(int id);

int main(int argc, char *argv[])
{
   // Storing start time as t1
   struct timeval tv;
   double t1;
   double t2;
   gettimeofday(&tv, NULL);
   t1 = tv.tv_sec + tv.tv_usec/1000000.0;

   // Checking that all the args are passed in
   if ( argc !=5 ) {
      printf("Usage: %s <N> <B> <P> <C> \n", argv[0]);
      exit(1);
   }

   // Converting the args to ints
   N = atoi(argv[1]);
   B = atoi(argv[2]);
   P = atoi(argv[3]);
   C = atoi(argv[4]);
```

```
// Initializing the message queue attributes
mqd_t qdes;
struct mq_attr attr;
attr.mq_maxmsg = B;
attr.mq_msgsize = sizeof(int);
attr.mq_flags = 0;
attr.mq_curmsgs = 0;

// Creating the queue and setting the mode permissions to all users
qdes = mq_open(Q_NAME, (O_CREAT | O_RDWR), (S_IRWXU | S_IRWXG | S_IRWXO), &attr);

// If queue creation failed
if (qdes == -1) {
    perror("Main - mq_open() failed");
    return 1;
}

// Store producer and consumer pids
pid_t* prod_pids = malloc(P * sizeof(pid_t));
pid_t* con_pids = malloc(C * sizeof(pid_t));

int i;
// Create P producer processes
for (i = 0; i < P; i++) {
    prod_pids[i] = fork();
    if (prod_pids[i] < 0) {
        printf("Producer %d fork failed\n", i);
    } else if (prod_pids[i] == 0) {
        producer(i);
        break;
    }
}

// Create C consumer processes
for (i = 0; i < C; i++) {
    con_pids[i] = fork();
    if (con_pids[i] < 0) {
        printf("Consumer %d fork failed\n", i);
    } else if (con_pids[i] == 0) {
        consumer(i);
        break;
    }
}

// Wait for all producers to finish
int* ret_val;
for(i = 0; i < P; i++) {
    waitpid(prod_pids[i], ret_val, 0);
}

// Send -1 kill signals to the queue to terminate all consumers
int kill = -1;
for (i = 0; i < C; i++) {
    if (mq_send(qdes,(char *) &kill, sizeof(int), 0) == -1) {
```

```
            perror("Main - mq_send() failed");
            exit(1);
        }
    }

    // Wait for all consumers to finish
    for(i = 0; i < C; i++) {
        waitpid(con_pids[i], ret_val, 0);
    }

    // Close the queue and do cleanup
    if (mq_close(qdes) == -1) {
        perror("mq_close() failed");
        exit(2);
    }

    if (mq_unlink(Q_NAME) != 0) {
        perror("mq_unlink() failed");
        exit(3);
    }

    free(prod_pids);
    free(con_pids);

    // Store the finish time and print the execution time
    gettimeofday(&tv, NULL);

    t2 = tv.tv_sec + tv.tv_usec/1000000.0;
    printf("System execution time: %.6lf seconds\n", t2-t1);

    return 0;
}

void producer(int id) {
    int i;

    // Open queue for writing
    mqd_t q = mq_open(Q_NAME, O_WRONLY);
    if (q == -1) {
        perror("Producer - mq_open() failed");
        exit(1);
    }

    // Send ints to the queue according to i%P == id
    for (i = 0; i < N; i++) {
        if (i%P == id) {
            if (mq_send(q,(char *) &i, sizeof(int), 0) == -1) {
                perror("Producer - mq_send() failed");
                exit(1);
            }
        }
    }

    // Close the queue and exit the process
```

```
    mq_close(q);
    exit(0);
}


void consumer(int cid) {
    // Open the queue for reading
    mqd_t q = mq_open(Q_NAME, O_RDWR);
    if (q == -1) {
        perror("Consumer - mq_open() failed");
        exit(1);
    }

    int pt;
    int root;
    while (true){
        // If there's an error receiving from queue
        if (mq_receive(q, (char *) &pt, sizeof(int), NULL) == -1) {
            perror("mq_receive() failed");
            printf("Consumer: %d failed.\n", cid);
            exit(1);
        } else {
            // Else the consumer read the point correctly

            // Encounters kill signal so break out of loop
            if (pt < 0) {
                break;
            }

            // Calculate and print sqroot
            root = sqrt(pt);
            if (root*root == pt) {
                printf("%d %d %d\n", cid, pt, root);
            }
        }
    }

    // Close the queue and exit the process
    mq_close(q);
    exit(0);
}
```

Appendix B - producer.c for Multithreaded Method

```
/**
* @file:   produce.c
* @brief:  Implementing the producer-consumer paradigm with multiple threads and shared memory.
* @date:   2017/11/22
* NOTES:
   Compile with: gcc produce.c -pthread -lm -o produce
*/


#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <time.h>
#include <stdbool.h>
#include <math.h>
#include <pthread.h>
#include <semaphore.h>

int N; // Amount of numbers
int B; // Buffer size
int P; // Number of producers
int C; // Number of consumers

void *producer(void * arg);
void *consumer(void * arg);

// Shared circular buffer that holds the messages
int* shared_buffer;

// 2 semaphores to coordinate consumers and producers
sem_t items;
sem_t spaces;
// 1 mutex to protect access to shared memory
pthread_mutex_t mutex;

// Shared indexes that keep track of where the all the producers/consumers are writing/reading
int p_index = 0;
int c_index = 0;

int main(int argc, char *argv[])
{
   // Storing start time as t1
   struct timeval tv;
   double t1;
   double t2;
   gettimeofday(&tv, NULL);
   t1 = tv.tv_sec + tv.tv_usec/1000000.0;

   // Checking that all the args are passed in
```

```
if ( argc !=5 ) {
    printf("Usage: %s <N> <B> <P> <C> \n", argv[0]);
    exit(1);
}

// Converting the args to ints
N = atoi(argv[1]);
B = atoi(argv[2]);
P = atoi(argv[3]);
C = atoi(argv[4]);

// Setting the shared buffer to hold B integers
shared_buffer = malloc(B * sizeof(int));

// Storing thread ids of producers and consumers
pthread_t producers[P];
pthread_t consumers[C];

// Initializing items to 0 as no items produced to start off
sem_init(&items, 0, 0);
// Initializing spaces to B as all the buffer spaces empty to start off
sem_init(&spaces, 0, B);
// Initializing mutex
pthread_mutex_init(&mutex, NULL);

int i;
// create P producer threads
for (i = 0; i < P; i++) {
    int* id = malloc(sizeof(int));
    *id = i;
    int status = pthread_create(&producers[i], NULL, producer, id);
    if (status != 0) {
        printf("Producer %d thread create failed\n", i);
    }
}

// create C consumer threads
for (i = 0; i < C; i++) {
    int* id = malloc(sizeof(int));
    *id = i;
    int status = pthread_create(&consumers[i], NULL, consumer, id);
    if (status != 0) {
        printf("Consumer %d thread create failed\n", i);
    }
}

// Join all producers and consumers
void* ret_val;
for(i = 0; i < P; i++) {
    pthread_join(producers[i], &ret_val);
    free(ret_val);
}
for(i = 0; i < C; i++) {
    pthread_join(consumers[i], &ret_val);
```

```
        free(ret_val);
    }

    // Cleanup
    pthread_mutex_destroy(&mutex);
    sem_destroy(&items);
    sem_destroy(&spaces);

    free(shared_buffer);

    // Store the finish time and print the execution time
    gettimeofday(&tv, NULL);

    t2 = tv.tv_sec + tv.tv_usec/1000000.0;
    printf("System execution time: %.6lf seconds\n", t2-t1);

    return 0;
}

void *producer(void* arg) {
    int* pid = (int *) arg;

    int i;
    // Add ints to the buffer according to i%P == pid
    for (i = 0; i < N; i++) {
        if (i%P == *pid) {
            sem_wait(&spaces);
            pthread_mutex_lock(&mutex);

            // Write to p_index % B (circular buffer)
            shared_buffer[p_index % B] = i;
            p_index++;

            pthread_mutex_unlock(&mutex);
            sem_post(&items);
        }
    }

    // Cleanup and exit thread
    free(arg);
    pthread_exit(0);
}


void *consumer(void* arg) {
    int* cid = (int *) arg;

    int num;
    int root;
    while(true) {
        sem_wait(&items);
        pthread_mutex_lock(&mutex);

        // Read from c_index % B (circular buffer)
```

```
        num = shared_buffer[c_index % B];
        c_index++;

        // Already consumed N items so break
        if (c_index >= N-1) {
            break;
        }

        pthread_mutex_unlock(&mutex);
        sem_post(&spaces);

        // Calculate and print sqrt
        root = sqrt(num);
        if (root*root == num) {
            printf("%d %d %d\n", *cid, num, root);
        }
    }
    // Thread broke out of loop because N items have been read
    // Post semaphores and unlock mutex to let any stuck threads through
    sem_post(&spaces);
    sem_post(&items);
    pthread_mutex_unlock(&mutex);

    // Cleanup and exit thread
    free(arg);
    pthread_exit(0);
}
```