Lab 5 Report
By: Sudhanva Huruli and Aayushya Agarwal

N	B	P	C	Time
100	4	1	1	0.002514
100	4	1	2	0.003048
100	4	1	3	0.003731
100	4	2	1	0.003183
100	4	3	1	0.003638
100	8	1	1	0.002507
100	8	1	2	0.003072
100	8	1	3	0.003661
100	8	2	1	0.003046
100	8	3	1	0.003698
398	8	1	1	0.003037
398	8	1	2	0.003857
398	8	1	3	0.004295
398	8	2	1	0.003755
398	8	3	1	0.004094

Message Queue
(above) X = 500

Shared Memory
(below) X=500

N	B	P	C	Time
100	4	1	1	0.000429
100	4	1	2	0.000513
100	4	1	3	0.000584
100	4	2	1	0.000512
100	4	3	1	0.000677
100	8	1	1	0.000369
100	8	1	2	0.000445
100	8	1	3	0.000502
100	8	2	1	0.000528
100	8	3	1	0.000641
398	8	1	1	0.000738
398	8	1	2	0.000725
398	8	1	3	0.000803
398	8	2	1	0.000816
398	8	3	1	0.00098

The information below illustrates the average time and standard deviation at N = 398, B = 8, P = 1 and C = 3 and X = 500 for the message queue version.

at (N,B,P,C) = (398,8,1,3)	
average time	0.004295

standard deviation	0.000543

The information below illustrates the average time and standard deviation at N = 398, B = 8, P = 1 and C = 3 and X = 500 for the shared memory version.

average time	0.000803
standard deviation	0.000565

As you can see the shared memory version is much faster than the message queue version for the given testing conditions. If we take the condition above (N = 398, B = 8, P = 1 and C = 3 and X = 500), we see that the average time for the shared memory is more than 10th of the average time for message queue. Also we can see that the standard deviation is the same, so that means in general, the shared memory is a lot faster than the standard deviation.

Advantages and Disadvantages

We found, in general that the shared memory was faster than the message queue. The advantage of using shared memory over message queues is that the shared memory uses threads. This is an advantage because threads can switch a lot easier, take less time to be initialized and can share data easier. Threads are a lot more lightweight, so they can be opened, closed, and share data a lot easier than processes. Moreover, another advantage with the shared memory version was that the different threads are localized and as a result switching between them is quicker. Additionally, the circular queue used in the shared memory version is local to the process and as a result data accesses are faster. One of the disadvantages of shared memory is that we (as programmers) have to worry about race conditions. Furthermore, we need to lock and unlock critical resources on our own and ensure the blocking of threads appropriately.

The advantage of using the message queue version is that the message queue takes care of blocking processes. Therefore, we do not need to have as many semaphores as the previous version. Furthermore, sending data between processes is a lot easier. The disadvantage of the shared memory version is that it is slower. It is slower because the data is not local to the process. As a result, data accesses will take a considerably more amount of time. Additionally, spawning entirely new processes for each consumer or producer is taxing on the system. Closing each process will similarly be relatively heavy compared to the thread version.

Lab 5 Report
By: Sudhanva Huruli and Aayushya Agarwal

Appendix A

Shared Memory Version

```
/**
 * @file:   code/lab5/mqueue/sender.c
 * @brief:  send random points to a POSIX queue. Type 'q' to terminate.
 * @date:   2014/06/25
 * NOTES:
 * To execute: <executable> <arg1>, where arg1 is a posix queue name.
 * EXAMPLE: ./sender.out /mailbox1_userid
 * A valid posix queue name must start with a "/".
 * Execute command "man mq_overview" for details.
 * Check /dev/mqueue to clean up the queue if sender fails to do so.
 */

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <pthread.h>
#include <time.h>
#include "common.h"
#include "point.h"
#include <wait.h>

#include <semaphore.h>

#define _XOPEN_SOURCE 600

int capacity = 0;
int P = 0;
int C = 0;
int buffer_size = 0;
struct mq_attr attr;

// These two structs are used to send parameters to the producer and consumer
methods
struct producer_parms
{
        /* The ID of the producer */
        int producer_id;
};
```

```
struct consumer_params
{
            /* The ID of the consumer */
            int consumer_id;
};

struct buffer_queue {
   int front,rear;
   int capacity;
   int *array;
};


/* -- This queue was found off the internet -- */
/* Reference: http://stackoverflow.com/questions/20619234/circular-queue-
implementation */
struct buffer_queue* q(int size)
{
    struct buffer_queue *q=malloc(sizeof(struct buffer_queue));
    if(!q)
        return NULL;
    q->capacity=size;
    q->front=-1;
    q->rear=-1;
    q->array=malloc(q->capacity*sizeof(int));
    if(!q->array)
       return NULL;
    return q;
};

int isemptyqueue(struct buffer_queue *q) {
   return(q->front==-1);
}

int isfullqueue(struct buffer_queue *q) {
   return((q->rear+1)%q->capacity==q->rear);
}

void enqueue(struct buffer_queue *q,int x) {

   if(isfullqueue(q))
      printf("queue overflow\n");
   else{
      q->rear=(q->rear+1)%q->capacity;
      q->array[q->rear]=x;
      if(q->front==-1) {
```

```
      q->front=q->rear;
    }
  }
}

int dequeue(struct buffer_queue *q) {
  int data=0;

  if(isemptyqueue(q)) {
    printf("queue underflow");
    return 0;
  }
  else {
    data=q->array[q->front];
    if(q->front==q->rear)
      q->front=q->rear=-1;
    else
      q->front=(q->front+1)%q->capacity;
  }

  return data;
}
/* -- This queue was found off the internet -- */

struct buffer_queue *shared_queue = NULL;

// declaring semaphores and mutex
sem_t consumers;
sem_t producers;
sem_t buffers;
pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;

// Consumer
int receiver(void* parameters)
{
        // local consumer params struct to contain our ID
        struct consumer_params* c = (struct consumer_params*) parameters;
        //run consumers until we are done consuming
        while(1){

                // do a sem wait here
                // total = consumers, count = producers, buffer = buffers
                // if we are done consuming then we break out of this consumer
                if (sem_trywait(&consumers) == -1){
                        break;
                }
		// must wait on producers to produce something before we start
consuming
		sem_wait(&producers);

		// mutex since our queue is a critical resource
		pthread_mutex_lock(&job_queue_mutex);
		int initial_buffer_number = dequeue(shared_queue);
		pthread_mutex_unlock(&job_queue_mutex);

		// calculate square root of the the number.
		// if our double subtract the integer is extremely small then its an
integer
		double sqrt_number = sqrt(initial_buffer_number);
		if (sqrt_number-floor(sqrt_number) < 1e-8){
			printf("%d %d %d \n", c->consumer_id, initial_buffer_number,
(int) sqrt(initial_buffer_number));
		}

		// we want to do a sem_post to the buffers semaphore since one thing
got read off the buffer
		sem_post (&buffers);
	}
	return NULL;
}

// use message queue for the counter - Consumers coordinate together. Producers
do not need to coordinate.
// may produce and may consume.
// advanced linux book for the second part
int sender(void* parameters){
	int i;
	struct producer_parms* p = (struct producer_parms*) parameters;

	// we produce starting at our producer id and keep going till we are less than
N.
	// We only produce multiples of P starting from our ID
	for (i=p->producer_id; i < capacity; i+=P){

		// must have buffer space
		sem_wait(&buffers);

		// mutex : Queue is a critical resource
		pthread_mutex_lock(&job_queue_mutex);
		//printf("Hello am here on iteration %d\n", i);
		enqueue(shared_queue,i);
		//printf("Sending: %d\n",i);
		pthread_mutex_unlock(&job_queue_mutex);

		// we have started producing and another consumer may start
running
		sem_post(&producers);
	}
	return NULL;
}


int main(int argc, char *argv[])
{

	int i;
	struct timeval tv;
	double t1;
	double t2;


	// 5 args must be there
	if (argc != 5){
		printf("Your command needs to look like: ./produce <N> <B> <P>
<C>\n");
		exit(1);
	}

	/* Initializing our N, B, P, C values*/
	capacity = atoi(argv[1]);
	P = atoi(argv[3]);
	C = atoi(argv[4]);
	buffer_size = atoi(argv[2]);

	if (buffer_size == 0 || P == 0 || C == 0){
		printf("Your buffer size, number of consumers and number of
producers cannot be 0.\n");
		exit(1);
	}

	// init our shared queue
	shared_queue = q(buffer_size);

	// declaring semaphores and mutex
	sem_init (&consumers, 0, capacity); // consumer semaphore to know when
we are done
```

```
	sem_init (&producers, 0, 0); // set at 0 so consumers are initially blocked
until a producer produces
	sem_init (&buffers, 0, buffer_size); // semaphore to make sure there is buffer
space

	// we will have P prods and C consumers
	pthread_t producer_thread_id[P];
	pthread_t consumer_thread_id[C];

	// our time before we start
	gettimeofday(&tv, NULL);
	t1 = tv.tv_sec + tv.tv_usec/1000000.0;

	// we will produce P producers and C consumers
	for (i=0; i < P; i++){
		struct producer_parms *pp = malloc(sizeof(struct producer_parms));
		pp->producer_id = i;
		//printf("Your i value is %d", i);
		pthread_create (&producer_thread_id[i], NULL, &sender, pp);
	}

	for (i=0; i < C; i++){
		struct consumer_params *cc = malloc(sizeof(struct
consumer_params));
		cc->consumer_id = i;
		pthread_create (&consumer_thread_id[i], NULL, &receiver, cc);
	}


	// must join it P and C times
	for (i=0; i < P; i++){
		pthread_join (producer_thread_id[i], NULL);
	}

	for (i=0; i < C; i++){
		pthread_join (consumer_thread_id[i], NULL);
	}

	// output the total time it took
	gettimeofday(&tv, NULL);
	t2 = tv.tv_sec + tv.tv_usec/1000000.0;
	printf("System execution time: %.6lf seconds\n",t2-t1);


	return 0;
}
```

Lab 5 Report
By: Sudhanva Huruli and Aayushya Agarwal

Message Queue Version

```
/**
 * @file:   code/lab5/mqueue/top.c
 * @brief:  send random points to a POSIX queue. Type 'q' to terminate.
 * @date:   2014/06/25
 * NOTES:
 * To execute: <executable> <arg1>, where arg1 is a posix queue name.
 * EXAMPLE: ./sender.out /mailbox1_userid
 * A valid posix queue name must start with a "/".
 * Execute command "man mq_overview" for details.
 * Check /dev/mqueue to clean up the queue if sender fails to do so.
 */

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <time.h>
#include "common.h"
#include "point.h"
#include <wait.h>
#include <semaphore.h>

int spawn (char* program, char** arg_list)
{
  pid_t child_pid;

  /* Duplicate this process.  */
  child_pid = fork ();
  if (child_pid != 0)
    /* This is the parent process.  */
    return child_pid;
  else {
    /* Now execute PROGRAM, searching for it in the path.  */
    execvp (program, arg_list);
    /* The execvp function returns only if an error occurs.  */
    fprintf (stderr, "an error occurred in execvp\n");
    abort ();
  }
}

int main(int argc, char *argv[])
{
	int child_status;/* this variable is only used for wait*/
```

```
mqd_t qdes; /*message queue for data*/
sem_t *sem; /*semaphore that counts number consumed*/
int i,j;

/*time values for timing analysis*/
struct timeval tv;
double t1;
double t2;

/*msesage queue attributes*/
char *qname = NULL;
mode_t mode = S_IRUSR | S_IWUSR;
struct mq_attr attr;

/* ensuring that the five arguments required are coming in*/
if ( argc !=5 ) {
        printf("Usage: %s <qname>\n", argv[0]);
        printf("The qname must start with a \"/\". \n");
        printf("You must specify your N value");
        printf("You must specify your B value");
        exit(1);
}

/* capacity = N*/
int capacity = atoi(argv[1]);
int B = atoi(argv[2]); /*size of queue*/
int P = atoi(argv[3]); /*number of producers*/
int C = atoi(argv[4]); /*number of consumers*/

if( B <= 0 ) {
        printf("The value B must be greater than 0");
        exit(1);
}

if( P <= 0 ) {
        printf("The value P must be greater than 0");
        exit(1);
}

if( C <= 0 ) {
        printf("The value C must be greater than 0");
        exit(1);
}

qname = "/mailbox1_sshuruli";/*default mailbox name*/
```

```
	/*max queue size*/
	attr.mq_maxmsg  = B;
	attr.mq_msgsize = sizeof(struct point);
	attr.mq_flags   = 0;

	qdes  = mq_open(qname, O_RDWR | O_CREAT, mode, &attr);
	if (qdes == -1 ) {
		printf("Opening the queue failed in the top.c");
		perror("mq_open() failed");
		exit(1);
	}

	/*initializing semaphore with total number of numbers created*/
	sem = sem_open("/num_consumed", O_CREAT | O_EXCL, 0644, capacity);
	if (sem == SEM_FAILED)
	{
		printf ("Could not open semaphore --- errno \n");
	}

	/* building our receivers process's argument list*/
	char* arg_list_reciever[] = {
		argv[2],
		"0",
		NULL

	};
	/*building our producers process's argument list*/
	char* arg_list_producer[] = {
		argv[1],
		argv[3],
		"0",
		argv[2],
		NULL
	};
	/*time before first child is spawned*/
	gettimeofday(&tv, NULL);
	t1 = tv.tv_sec + tv.tv_usec/1000000.0;

	/*spawning C consumers*/
	for(j=0;j<C;j++) {

		char str[15];
		sprintf(str, "%d", j);
		arg_list_reciever[1] = str;
		//receiver process
		spawn("./receiver.out", arg_list_reciever);
	}

	/*spawning P producers*/
	for(i=0;i<P;i++) {
		char str[15];
		sprintf(str, "%d", i);
		arg_list_producer[2] = str;

		spawn("./sender.out", arg_list_producer);
	}

	/* before unlinking queue we must wait for child to finish*/
	for (i = 0; i < (C+ P); i ++){
		wait(&child_status);
	}
	if (WIFEXITED (child_status)){

		/*unlinking and closing message queue*/
		if (mq_close(qdes) == -1){
			perror("mq_close() failed");
			exit(2);
		}
		if (mq_unlink(qname) != 0){
			perror("mq_unlink() failed");
			exit(3);
		}

		/*closing and unlinking semaphore*/
		sem_close(sem);
		sem_unlink("num_consumed");

		/*time after all numbers are consumed*/
		gettimeofday(&tv, NULL);
		t2 = tv.tv_sec + tv.tv_usec/1000000.0;

	}
	else{
		printf ("the child process exited abnormally\n");
	}

	printf("System execution time: %.6lf seconds\n",t2-t1);

	return 0;
}
```

Lab 5 Report
By: Sudhanva Huruli and Aayushya Agarwal

```
/**
 * @file:   code/lab5/mqueue/sender.c
 * @brief:  send random points to a POSIX queue. Type 'q' to terminate.
 * @date:   2014/06/25
 * NOTES:
 * To execute: <executable> <arg1>, where arg1 is a posix queue name.
 * EXAMPLE: ./sender.out /mailbox1_userid
 * A valid posix queue name must start with a "/".
 * Execute command "man mq_overview" for details.
 * Check /dev/mqueue to clean up the queue if sender fails to do so.
 */

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <time.h>
#include "common.h"
#include "point.h"
#include <wait.h>

int main(int argc, char *argv[])
{

	mqd_t qdes;

	int i;

	char *qname = NULL;
	mode_t mode = S_IRUSR | S_IWUSR;
	struct mq_attr attr;
	struct point pt;

		/* ensuring that the four arguments required are coming in*/
	if ( argc !=4 ) {
		printf("You must specify your N value\n");
		printf("You must specify your P value");
		printf("You must specify your id value");
		printf("You must specify your B value");
		exit(1);
	}

	/* capacity = N*/
	int capacity = atoi(argv[0]);
	int P=atoi(argv[1]); /*number of producers*/
	int id=atoi(argv[2]);  /*id of producer*/
	int queue_size = atoi(argv[3]);		/*queue size*/

	/*default mailbox name*/
	qname = "/mailbox1_sshuruli";
	attr.mq_maxmsg  = queue_size;
	attr.mq_msgsize = sizeof(struct point);
	attr.mq_flags   = 0;		/* a blocking queue  */

	/*opening message queue*/
	qdes  = mq_open(qname, O_RDWR, mode, &attr);
	if (qdes == -1 ) {
		printf("Error in opening the mailbox in the producer.");
		perror("mq_open()");
		exit(1);
	}

	for(i=id; i< capacity; i+=P) {
		set_position(i, &pt);

		/* creating number and sending it in the message queue*/
		if (mq_send(qdes, (char *)&pt, sizeof(struct point), 0) == -1) {
			fprintf(stderr, "%s\n","Error in sending");
			perror("mq_send() failed");
		}

	}
	/*closing message queue*/
	if (mq_close(qdes) == -1){
		perror("mq_close() failed");
		exit(2);
	}

	return 0;
}
```

Lab 5 Report
By: Sudhanva Huruli and Aayushya Agarwal

```
/**
 * @file:   code/lab5/mqueue/receiver.c
 * @brief:  receive a message from POSIX queue. Typle Ctrl-C to quit
 * @date:   2014/06/25
 * NOTES:
 * To execute: <executable> <arg1>, where arg1 is a posix queue name.
 * EXAMPLE: ./receiver.out /mailbox1_userid
 * A valid posix queue name must start with a "/".
 * Execute command "man mq_overview" for details.
 */

#include <stdbool.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <signal.h>
#include "common.h"
#include "point.h"
#include <math.h>
#include <semaphore.h>

#define _XOPEN_SOURCE 600

bool g_continue = true;

void sig_handler(int sig)
{
	g_continue = false;
}

int main(int argc, char *argv[])
{
	mqd_t qdes;
	char  *qname = NULL;
	mode_t mode = S_IRUSR | S_IWUSR;
	struct mq_attr attr;

	/* Open a preexisting semaphore. */
	sem_t *sem;

	/* ensuring that the two arguments required are coming in*/
	if ( argc !=2 ) {
		printf("You must specify your id value");
		printf("You must specify your B value");
		exit(1);
	}
	int queue_size = atoi(argv[0]);		/*message queue size*/
	int id = atoi(argv[1]);		/*id of consumer*/

	sem = sem_open("/num_consumed", 0); /* Open a preexisting semaphore. */
	if(sem==SEM_FAILED){
		printf("Error opening sem\n");
	}

	/* setting a default name for our mailbox*/
	qname = "/mailbox1_sshuruli";
	attr.mq_maxmsg  = queue_size;
	attr.mq_msgsize = sizeof(struct point);
	attr.mq_flags   = 0;	/* a blocking queue  */

	/*opening message queue*/
	qdes  = mq_open(qname, O_RDONLY, mode, &attr);
	if (qdes == -1 ) {
		printf("Error in opening the mailbox in the consumer.");
		perror("mq_open()");
		exit(1);
	}

	signal(SIGINT, sig_handler); /* install Ctl-C signal handler */

	/* we will only consume up until all the items have been consumed*/
	while(sem_trywait(sem)!=-1){

		struct point pt;
		struct timespec ts = {time(0) + 5, 0};

		/* only block for a limited time if the queue is empty */
		if (mq_timedreceive(qdes, (char *) &pt, sizeof(struct point), 0, &ts) ==
-1) {
			perror("mq_timedreceive() failed");
			printf("Type Ctrl-C and wait for 5 seconds to terminate.\n");
		}
		else {

			/*consuming integer and finding square root*/
			int x = get_x_coord(pt);
			double sqrt_num = sqrt(x);

			/*if the square root number is an integer, print the result*/
```

```
				if(sqrt_num-floor(sqrt_num) < 1e-8) {
					printf("%d %d %d \n",id, x, (int) sqrt_num);
				}

		}

	}

	/*close the semaphore*/
	sem_close(sem);

	/* we must close the connection to the queue still*/
	if (mq_close(qdes) == -1) {
		perror("mq_close() failed");
		exit(2);
	}

	return 0;
}
```