**ECE 254 Lab 3 Report**

Chris Fulton

Talal Kamran

Threads average:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | B | P | C | Time |
| 100 | 4 | 1 | 1 | 0.000867 |
| 100 | 4 | 1 | 2 | 0.000865 |
| 100 | 4 | 1 | 3 | 0.000974 |
| 100 | 4 | 2 | 1 | 0.000765 |
| 100 | 4 | 3 | 1 | 0.001045 |
| 100 | 8 | 1 | 1 | 0.000887 |
| 100 | 8 | 1 | 2 | 0.000849 |
| 100 | 8 | 1 | 3 | 0.000971 |
| 100 | 8 | 2 | 1 | 0.00092 |
| 100 | 8 | 3 | 1 | 0.001017 |
| 398 | 8 | 1 | 1 | 0.001969 |
| 398 | 8 | 1 | 2 | 0.001714 |
| 398 | 8 | 1 | 3 | 0.002118 |
| 398 | 8 | 2 | 1 | 0.001778 |
| 398 | 8 | 3 | 1 | 0.002051 |

Threads standard deviation:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | B | P | C | Time |
| 100 | 4 | 1 | 1 | 0.000083 |
| 100 | 4 | 1 | 2 | 0.000131 |
| 100 | 4 | 1 | 3 | 0.000101 |
| 100 | 4 | 2 | 1 | 0.000323 |
| 100 | 4 | 3 | 1 | 0.000257 |
| 100 | 8 | 1 | 1 | 0.000134 |
| 100 | 8 | 1 | 2 | 0.000115 |
| 100 | 8 | 1 | 3 | 0.000138 |
| 100 | 8 | 2 | 1 | 0.000147 |
| 100 | 8 | 3 | 1 | 0.000146 |
| 398 | 8 | 1 | 1 | 0.000226 |
| 398 | 8 | 1 | 2 | 0.000162 |
| 398 | 8 | 1 | 3 | 0.00017 |
| 398 | 8 | 2 | 1 | 0.000199 |
| 398 | 8 | 3 | 1 | 0.000239 |

Messages average:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | B | P | C | Time |
| 100 | 4 | 1 | 1 | 0.001492 |
| 100 | 4 | 1 | 2 | 0.001614 |
| 100 | 4 | 1 | 3 | 0.00172 |
| 100 | 4 | 2 | 1 | 0.001722 |
| 100 | 4 | 3 | 1 | 0.001876 |
| 100 | 8 | 1 | 1 | 0.001445 |
| 100 | 8 | 1 | 2 | 0.001694 |
| 100 | 8 | 1 | 3 | 0.001688 |
| 100 | 8 | 2 | 1 | 0.001727 |
| 100 | 8 | 3 | 1 | 0.00188 |
| 398 | 8 | 1 | 1 | 0.001338 |
| 398 | 8 | 1 | 2 | 0.00284 |
| 398 | 8 | 1 | 3 | 0.002787 |
| 398 | 8 | 2 | 1 | 0.002907 |
| 398 | 8 | 3 | 1 | 0.002971 |

Messages standard deviation:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | B | P | C | Time |
| 100 | 4 | 1 | 1 | 0.000676 |
| 100 | 4 | 1 | 2 | 0.000192 |
| 100 | 4 | 1 | 3 | 0.000088 |
| 100 | 4 | 2 | 1 | 0.000109 |
| 100 | 4 | 3 | 1 | 0.000095 |
| 100 | 8 | 1 | 1 | 0.000687 |
| 100 | 8 | 1 | 2 | 0.000171 |
| 100 | 8 | 1 | 3 | 0.000216 |
| 100 | 8 | 2 | 1 | 0.000109 |
| 100 | 8 | 3 | 1 | 0.000092 |
| 398 | 8 | 1 | 1 | 0.000718 |
| 398 | 8 | 1 | 2 | 0.000395 |
| 398 | 8 | 1 | 3 | 0.000125 |
| 398 | 8 | 2 | 1 | 0.000228 |
| 398 | 8 | 3 | 1 | 0.000136 |

Analysis:

POSIX messages have more built-in functionality and safety measures. They provide synchronization by blocking processes until read or write is possible, and they provide data safety by providing additional, specific data space. For these features, they tend to have more overhead, making them slower. As seen by the collected data, the average completion time for the task using separate processes and a message queue tended to be close to double to the time taken for threads and a shared memory buffer.

Threads with shared memory have are much less safe than POSIX message queues because the user is required to ensure the data safety and synchronization themselves. With shared memory, there are no safety features, so any mistake could easily cause data corruption. In exchange, shared memory tends to have much lower overhead and so higher speed and possible efficiency.

Appendix: “messages.c” Source Code

#include <stddef.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/wait.h>

#include <spawn.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <math.h>

#include "helpers.h"

int num\_producers\_done = 0;

int production\_done = 0;

struct for\_producer {

int num\_to\_prod; // How many values are to be produced

int prod\_ID; // ID of this producer

int num\_prod; // How many producers there are

mqd\_t mq; // Pointer to the mqueue

};

struct for\_consumer {

int cons\_ID; // ID of this consumer

mqd\_t mq; // Pointer to the mqueue

};

void\* producer( struct for\_producer\* data\_in )

{

struct for\_producer\* prod\_params;

// The number that will be produced

int produced;

unsigned prio = 1;

prod\_params = data\_in;

produced = prod\_params->prod\_ID;

printf("Initialized producer %d\n", prod\_params->prod\_ID);

while (produced < prod\_params->num\_to\_prod) {

if (0 == mq\_send(prod\_params->mq, &produced, sizeof(int), prio)) {

produced += prod\_params->num\_prod;

}

}

// This producer is done all of its production.

num\_producers\_done++;

// If all producers are done, then raise a flag for the consumers

if( num\_producers\_done == prod\_params->num\_prod ) {

production\_done = 1;

}

return 0;

}

void\* consumer( struct for\_consumer\* data\_in )

{

struct for\_consumer\* cons\_params;

unsigned prio = 1;

int consumed = 0;

double root;

cons\_params = data\_in;

printf("Initialized consumer %d\n", cons\_params->cons\_ID);

while( 1 ) {

if (-1 != mq\_receive( cons\_params->mq, &consumed, sizeof(int), &prio )) {

if( consumed != -1 ) {

root = sqrt( consumed );

if( round( root ) == root ) {

printf("%d %d %d \n", cons\_params->cons\_ID, (int)consumed, (int)root);

}

}

else {

break;

}

}

}

return 0;

}

// gcc -g -lpthread -lm -lrt -o produce messages.c helpers.c

///////////////////////////////////////////////////////////////////////

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

{

int n, b, p, c, i, j;

double start\_time, end\_time;

struct timeval tv;

struct for\_producer\* data\_prod;

struct for\_consumer\* data\_cons;

pid\_t\* prod\_processes;

pid\_t\* cons\_processes;

mqd\_t mq;

int wait\_status;

// Get command line arguments:

// n = number to produce

// b = buffer size

// p = number of producers

// c = number of consumers

if( argc < 5 ) {

return -1;

}

n = atoi( argv[1] );

b = atoi( argv[2] );

p = atoi( argv[3] );

c = atoi( argv[4] );

fflush(stdout);

// Initialize mailbox, get start time before creating producers and consumers

mq = init\_mqueue( b );

gettimeofday(&tv, NULL);

start\_time = tv.tv\_usec/1000000.0 + tv.tv\_sec;

printf("Initializing producers\n"); fflush(stdout);

// Initialize producers

data\_prod = (struct for\_producer\*)malloc( sizeof(struct for\_producer)\*p );

prod\_processes = (pid\_t\*)malloc( sizeof(pid\_t\*)\*p );

for (i = 0; i < p; i++) {

data\_prod[i].num\_to\_prod = n;

data\_prod[i].prod\_ID = i;

data\_prod[i].num\_prod = p;

data\_prod[i].mq = mq;

prod\_processes[i] = fork();

if (prod\_processes[i] < 0) {

fprintf(stderr, "Fork Failed");

return -1;

}

else if (prod\_processes[i] == 0) {

producer( &data\_prod[i] );

return 0;

}

}

printf("Initializing consumers\n"); fflush(stdout);

// Initialize consumers

data\_cons = (struct for\_consumer\*)malloc( sizeof(struct for\_consumer)\*p );

cons\_processes = (pid\_t\*)malloc( sizeof(pid\_t\*)\*p );

for (j = 0; j < c; j++) {

data\_cons[j].cons\_ID = j;

data\_cons[j].mq = mq;

cons\_processes[j] = fork();

if (cons\_processes[j] < 0) {

fprintf(stderr, "Fork Failed");

return -1;

}

else if (cons\_processes[j] == 0) {

consumer( &data\_cons[j] );

return 0;

}

}

printf("waiting for producers\n"); fflush(stdout);

// Wait for all producers to finish

for (j = 0; j < p; j++) {

waitpid(prod\_processes[j], &wait\_status, 0);

}

printf("sending kills \n"); fflush(stdout);

// Send "kill" messages to each of the consumers

int stop\_msg;

int prio;

stop\_msg = -1;

prio = 2;

for (j = 0; j < c; j++) {

if (0 == mq\_send(mq, (char\*)&stop\_msg, sizeof(int), prio) ){

printf("kill message %d sent \n", j);

}

}

printf("waiting for consumers\n"); fflush(stdout);

// Wait for all consumers to finish

for (j = 0; j < c; j++) {

waitpid(cons\_processes[j], &wait\_status, 0);

}

gettimeofday(&tv, NULL);

end\_time = tv.tv\_usec/1000000.0 + tv.tv\_sec;

printf("System execution time: %.6lf \n", end\_time - start\_time);

free(prod\_processes);

free(data\_prod);

free(cons\_processes);

free(data\_cons);

mq\_close(mq);

}

Appendix: “threads.c” Source Code

#include <stddef.h>

#include <stdlib.h>

#include <stdio.h>

#include <sys/wait.h>

#include <spawn.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <semaphore.h>

#include <math.h>

#include "helpers.h"

int num\_producers\_done = 0;

int production\_done = 0;

sem\_t\* BUFFER\_MUTEX;

struct for\_producer {

int num\_to\_prod; // How many values are to be produced

int prod\_ID; // ID of this producer

int num\_prod; // How many producers there are

int\* buffer; // Pointer to the buffer

};

struct for\_consumer {

int cons\_ID; // ID of this consumer

int\* buffer; // Pointer to the buffer

};

// gcc -g -lpthread -lm -lrt -o produce threads.c helpers.c

/\* A producer thread. Produces numbers and places them in a buffer.

Parameters:

num\_to\_produce is the number of integers the producer will produce

buffer is the buffer that this producer will place its numbers in

pid is the ID of this producer

num\_producers is the total number of producers

Return:

void

\*/

void\* producer( void\* data\_in )

{

struct for\_producer\* prod\_params;

// The number that will be produced

int produced;

prod\_params = (struct for\_producer\*)data\_in;

produced = prod\_params->prod\_ID;

while (produced < prod\_params->num\_to\_prod) {

//printf("num: %d, produced: %d \n", prod\_params->num\_to\_prod, produced);

sem\_wait( BUFFER\_MUTEX );

//printf("pushed value: %d \n", produced);

if( !(push( prod\_params->buffer, produced ) == -1) ) {

produced += prod\_params->num\_prod;

}

sem\_post( BUFFER\_MUTEX );

}

// This producer is done all of its production.

sem\_wait( BUFFER\_MUTEX );

num\_producers\_done++;

// If all producers are done, then raise a flag for the consumers

if( num\_producers\_done == prod\_params->num\_prod ) {

production\_done = 1;

}

sem\_post( BUFFER\_MUTEX );

pthread\_exit(0);

}

/\* A consumer thread. Consumes numbers from a buffer and prints their square root.

Parameters:

num\_to\_consume is the number of integers the producer will consume

buffer is the buffer that this producer will retrieve its numbers from

cid is the ID of this consumer

Return:

void

\*/

void\* consumer( void\* data\_in )

{

struct for\_consumer\* cons\_params;

double consumed = -1;

double root;

cons\_params = (struct for\_consumer\*)data\_in;

while( !(production\_done && consumed == -1) ) {

//printf("consumed: %d, cid: %d \n", consumed, cons\_params->cons\_ID);

sem\_wait( BUFFER\_MUTEX );

consumed = (double) pop( cons\_params->buffer );

sem\_post( BUFFER\_MUTEX );

if( consumed != -1 ) {

root = sqrt( consumed );

if( round( root ) == root ) {

printf("%d %d %d \n", cons\_params->cons\_ID, (int)consumed, (int)root);

}

}

}

pthread\_exit(0);

}

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

{

int n, b, p, c, i, j, error\_code;

double start\_time, end\_time;

struct timeval tv;

int\* buffer;

struct for\_producer\* data\_prod;

struct for\_consumer\* data\_cons;

pthread\_t\* prod\_threads;

pthread\_t\* cons\_threads;

// Check the command line arguments. If there aren't enough, error

if( argc < 5 ) {

return -1;

}

n = atoi( argv[1] );

b = atoi( argv[2] );

p = atoi( argv[3] );

c = atoi( argv[4] );

// Initialize buffer, get start time before creating producers and consumers

buffer = init\_buffer( b );

if (buffer == 0) {

return -1;

}

BUFFER\_MUTEX = malloc( sizeof(sem\_t) );

if ((error\_code = sem\_init(BUFFER\_MUTEX, 0, 1))) {

printf("Error initializing semaphore: error %d\n", error\_code);

return -1;

}

gettimeofday(&tv, NULL);

start\_time = tv.tv\_sec + tv.tv\_usec/1000000.0;

// Initialize producers

prod\_threads = (pthread\_t\*)malloc( sizeof(pthread\_t)\*p );

data\_prod = (struct for\_producer\*)malloc( sizeof(struct for\_producer)\*p );

for (i = 0; i < p; i++) {

data\_prod[i].num\_to\_prod = n;

data\_prod[i].prod\_ID = i;

data\_prod[i].num\_prod = p;

data\_prod[i].buffer = buffer;

pthread\_create( &prod\_threads[i], NULL, producer, (void\*)&data\_prod[i] );

}

// Initialize consumers

cons\_threads = (pthread\_t\*)malloc( sizeof(pthread\_t)\*c );

data\_cons = (struct for\_consumer\*)malloc( sizeof(struct for\_consumer)\*c );

for (j = 0; j < c; j++) {

data\_cons[j].cons\_ID = j;

data\_cons[j].buffer = buffer;

pthread\_create( &cons\_threads[j], NULL, consumer, (void\*)&data\_cons[j] );

}

// Wait for all consumers to finish.

// Consumers wait for producers so we don't need to wait for producers

for (j = 0; j < c; j++) {

pthread\_join( cons\_threads[j], NULL );

}

gettimeofday(&tv, NULL);

end\_time = tv.tv\_sec + tv.tv\_usec/1000000.0;

printf("System execution time: %.6lf \n", end\_time - start\_time);

// Free allocated memory

free(prod\_threads);

free(data\_prod);

free(cons\_threads);

free(data\_cons);

free(buffer);

}

Appendix: “helpers.c” Source Code

#include <stdlib.h>

#include <mqueue.h>

#include <stdio.h>

#define INDEX buffer[0]

/\* Initializes the mqueue

Parameters:

q\_size is the maximum size of the mqueue

Output:

A pointer to the mqueue. Returns nullptr on error.

\*/

mqd\_t init\_mqueue( int q\_size )

{

fflush(stdout);

mqd\_t qdes;

char \*qname = NULL;

mode\_t mode = S\_IRUSR | S\_IWUSR;

struct mq\_attr attr;

qname = "/stuff";

attr.mq\_maxmsg = q\_size;

attr.mq\_msgsize = sizeof(int);

attr.mq\_flags = 0; /\* a blocking queue \*/

fflush(stdout);

qdes = mq\_open(qname, O\_RDWR | O\_CREAT, mode, &attr);

if (qdes == -1 ) {

perror("mq\_open()");

exit(1);

}

return qdes;

}

/\* Initializes the buffer; simple function to create an int array to

act as the buffer.

Parameters:

buffer\_size is the number of integers the buffer should hold

Output:

A pointer to the buffer. Returns nullptr on error.

\*/

int\* init\_buffer( int buffer\_size )

{

int\* ptr;

// It only makes sense to allocate a buffer of positive size, so

// return error code if buffer\_size is not positive

if( buffer\_size <= 0 ) {

return 0;

}

// Each buffer array contains the number of ints specified plus one:

// the 0th position in the buffer is used as an index for the buffer

// to keep track of which slot we're at

ptr = (int\*) malloc( sizeof(int)\*(buffer\_size + 1) );

if ( 0 == ptr ){

return 0;

}

// Initialize the index of the buffer to 1, the first available position

ptr[0] = 1;

return ptr;

}

/\* pops a number from the given buffer. Moves backwards and then

pops the value at that location.

Parameters:

buffer is a pointer to the buffer that will be popd from

Output:

The number popd from the buffer. Returns -1 on error.

\*/

int pop( int\* buffer )

{

// We're at the first non-index location in the buffer,

// so there's nothing to pop

if ( 1 == INDEX ) {

return -1;

}

// Decrement the index, then take the value at the indexed location

else {

INDEX--;

//printf("succesful pop: buffer[%d] returned %d \n", INDEX+1, buffer[INDEX]);

return buffer[INDEX];

}

}

/\* pushs a number on the given buffer. pushs the value at the current

location then moves forwards.

Parameters:

buffer is a pointer to the buffer that will be pushd on

value is the value to be pushd on the buffer

Output:

Returns -1 on error, 0 otherwise.

\*/

int push( int\* buffer, int value )

{

int buffer\_size = sizeof(buffer)/sizeof(INDEX);

// We're at the end of the buffer, so there's no room to push

if ( buffer\_size == INDEX ) {

//printf("failed to succesfully push: %d \n", value);

return -1;

}

// Writes the value to the buffer at the current index, then increments the index

else {

buffer[INDEX] = value;

INDEX++;

//printf("succesfully pushed: %d, buffer[%d] is now %d \n", value, INDEX, buffer[INDEX-1]);

return 0;

}

}

Appendix: “helpers.h” Source Code

#include <mqueue.h>

int\* init\_buffer( int buffer\_size );

int pop( int\* buffer );

int push( int\* buffer, int value );

mqd\_t init\_mqueue( int q\_size );