**ECE 254: Operating Systems and Systems Programming**

**Lab 5: Report**

Group 34

Tianyi Zhang and Kwok Yin Timothy Tong

2B Computer Engineering

University of Waterloo

Spring 2014

**Collected Data:**

**Average System Execution Time for Threads**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **B** | **P** | **C** | **Time** |
| 100 | 4 | 1 | 1 | 0.000652 |
| 100 | 4 | 1 | 2 | 0.001489 |
| 100 | 4 | 1 | 3 | 0.00199 |
| 100 | 4 | 2 | 1 | 0.001065 |
| 100 | 4 | 3 | 1 | 0.001147 |
| 100 | 8 | 1 | 1 | 0.000783 |
| 100 | 8 | 1 | 2 | 0.001488 |
| 100 | 8 | 1 | 3 | 0.001916 |
| 100 | 8 | 2 | 1 | 0.001046 |
| 100 | 8 | 3 | 1 | 0.001114 |
| 398 | 8 | 1 | 1 | 0.001747 |
| 398 | 8 | 1 | 2 | 0.003062 |
| 398 | 8 | 1 | 3 | 0.003914 |
| 398 | 8 | 2 | 1 | 0.001793 |
| 398 | 8 | 3 | 1 | 0.002024 |

**Table 1.1**: Average system execution time for threads in seconds for varying N. B, P, C collected over 800 runs

**Average System Execution Time for Processes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **B** | **P** | **C** | **Time** |
| 100 | 4 | 1 | 1 | 0.002295 |
| 100 | 4 | 1 | 2 | 0.002751 |
| 100 | 4 | 1 | 3 | 0.003469 |
| 100 | 4 | 2 | 1 | 0.002856 |
| 100 | 4 | 3 | 1 | 0.00341 |
| 100 | 8 | 1 | 1 | 0.002447 |
| 100 | 8 | 1 | 2 | 0.002847 |
| 100 | 8 | 1 | 3 | 0.003422 |
| 100 | 8 | 2 | 1 | 0.002917 |
| 100 | 8 | 3 | 1 | 0.003394 |
| 398 | 8 | 1 | 1 | 0.002904 |
| 398 | 8 | 1 | 2 | 0.003479 |
| 398 | 8 | 1 | 3 | 0.003933 |
| 398 | 8 | 2 | 1 | 0.003904 |
| 398 | 8 | 3 | 1 | 0.004302 |

**Table 1.2**: Average system execution time for processes in seconds for varying N. B, P, C values collected over 800 runs

**Timing Data Comparisons for a Given Sample Set**

|  |  |  |
| --- | --- | --- |
|  | **Average System Execution Time** | **Standard Deviation** |
| **Threads** | 0.003914 | 0.000969 |
| **Processes** | 0.003933 | 0.000547 |

**Table 1.3**: Timing data comparison for both implementations for (N,B,P,C) = (398, 8, 1, 3) in seconds collected over 800 runs

DISCUSSION

As can be seen from the data in Table 1.1 and Table 1.2, the average system execution time for threading is consistently faster in every case than the multi-process implementation. More specifically, if we look at Table 1.3 for a specific sample set of (N,B,P,C) = (398, 8, 1, 3), we can see that threads in this case are in fact \_\_\_\_\_\_\_\_ times faster on average than multi-processes over a 800 run sample. We conclude from the above observation that multi-threading is indeed significantly faster in terms of timing performance than multi-processes execution.

This makes a lot of sense and can be explained by the fact that threads require less overhead to establish and terminate due to fewer copying of memory, hence the faster performance. Another notable explanation can be credited to the fact that program context is maintained in a thread due to shared memory in CPU cache as opposed to being reloaded every for a process switch. This allows the OS to switch much faster for threads resulting in the increased performance. Lastly, the message queue for the process implementation is limited in its size and capacity. For ecelinux, the maximum queue size for messages in POSIX standards is 10. This means the performance for processes will be comparatively reduced due to increased frequency of blocking for consumers and producers, hence more overhead and reduced speed.

When it comes to context switching, utilizing threads would be advantageous because it takes much less time for the operating system to switch threads than processes as all threads are in the same process in this case. In addition, it would make sense to think that processes are initialized much slower, as the fork() call has the responsibility of cloning the entire process and subsequently, the exec() call has to bring the newly spawned child process to the current running point. As our data suggests, the threading implementation took less time to finish running in comparison to the process implementation.

Nevertheless, the process implementation also has its own upsides. Clearly, the whole implementation for the process version has a better defined structure. It consists of 3 components – a producer, a consumer, and a program that spawns them and records time. This allows for a better readability and code organization. In addition, this implementation uses a simple POSIX queue, which has concurrency features already built in. However on the other hand in the threading solution, we had to design and create our own with multiple semaphores in order to prevent race conditions from happening. During the implementation process, we had to watch out for deadlocks and starvations. Therefore, the multithreading solution was overall more difficult and time-consuming to debug.

One more thing worth mentioning is that the maximum size of the POSIX queue (mqueue) is fixed and determined by the operating system (10 on ECELINUX). Therefore, this would be a disadvantage for the process implementation, as one would definitely get more flexibility in terms of the buffer size with the threading version.

APPENDIX

**Code – Process implementation:**

/\*

\* producer.c

\* ECE254 Group 34

\* By: Tianyi Zhang and Kwok Yin Timothy Tong

\* University of Waterloo Computer Engineering

\* Fall 2015

\*/

#include <stdio.h>

#include <stdlib.h>

#include <mqueue.h>

const char\* qname = "/mailbox\_t94zhang";

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

// format should be ./produce <N> <B> <P> <C>

// ie. early return for invalid input

if (argc != 5) {

exit(1);

}

// number of numbers produced in a set

int N = atoi(argv[1]);

// number of producers in total

int P = atoi(argv[3]);

// argv[2] is a char pointer, take its value

int pid = \*argv[2];

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// open up the queue

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

mqd\_t qdes = mq\_open(qname, O\_RDWR);

// check if queue was opened successfully

if (qdes == -1 ) {

perror("mq\_open() failed in producer");

exit(1);

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// make producers which statisfy condition

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int msg = pid;

for (msg; msg < N; msg+=P) {

// Requirement: message / P = pid

int sendMsg = mq\_send(qdes, (char\*) &msg, sizeof(int), 0);

// check if mq\_send() succeeded

if (sendMsg == -1) {

perror("mq\_send() failed in producer");

}

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// final checks and cleaning up

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

if (mq\_close(qdes) == -1) {

perror("mq\_close() failed in producer");

exit(2);

}

return 0;

}

/\*

\* consumer.c

\* ECE254 Group 34

\* By : Tianyi Zhang and Kk Yin Timothy Tong

\* University of Waterloo Computer Engineering

\* Fall 2015

\*

\*/

#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 <math.h>

#include <semaphore.h>

const char\* qname = "/mailbox\_t94zhang";

const char\* sem\_name = "sem\_consumer\_t94zhang";

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

{

// format should be ./produce <N> <B> <P> <C>

// ie. early return for invalid input

if ( argc != 5 ) {

exit(1);

}

mqd\_t qdes; // queue\_descriptor

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

struct mq\_attr attr; // queue attributes

// unique id assigned for this consumer

// argv[2] is a char pointer, take its value

// in C int is recognized as char

int id = \*argv[2];

// initialize a blocking queue

attr.mq\_maxmsg = id;

attr.mq\_msgsize = sizeof(int);

attr.mq\_flags = 0;

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// open up the queue

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

qdes = mq\_open(qname, O\_RDWR | O\_CREAT, mode,

&attr);

// check if queue was opened successfully

if (qdes == -1 ) {

perror("mq\_open() in consumer");

exit(1);

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// open up the semaphore

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

sem\_t \*c\_semaphore = sem\_open(sem\_name, 0);

if (c\_semaphore == SEM\_FAILED) {

perror("sem\_open() in consumer");

exit(1);

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// wait for all items to be consumed

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int msg; // message to be received

while(sem\_trywait(c\_semaphore) != -1) { // decrement & locks the semaphore

// exit loop if and only if all callers stopped consuming

// ie. semaphore counter reaches 0

// returns the number of byes in the recieved msg.

// valid msg if return value > 0

int isReceived = mq\_receive(qdes, (char\*) &msg, sizeof(int),

0);

if (isReceived){

// find perfect square

if (((int)sqrt(msg) \* (int)sqrt(msg)) == msg){

printf("%i %i %i\n", id, msg, (int)sqrt(msg));

}

}

else

{

perror("mq\_receive() in consumer");

exit(1);

}

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// final checks and cleaning up

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

if (mq\_close(qdes) == -1) {

perror("mq\_close() failed in consumer");

exit(2);

}

if (sem\_close(c\_semaphore) == -1) {

perror("sem\_close failed in consumer");

exit(2);

}

return 0;

}

/\*

\* processes\_main.c

\* ECE254 Group 34

\*

\* By: Tianyi Zhang and Kwok Yin Timothy Tong

\* University of Waterloo Computer Engineering

\* Fall 2015

\*/

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

#include <mqueue.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <sys/time.h>

#include <time.h>

#include <semaphore.h>

double get\_time();

//constant queue\_name for both producer and consumer.

const char\* qname = "/mailbox\_t94zhang";

const char\* sem\_name = "sem\_consumer\_t94zhang";

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

{

// format should be ./produce <N> <B> <P> <C>

// ie. early return for invalid input

if (argc != 5) {

exit(1);

}

// number of integers the producer should produce

int N = atoi(argv[1]);

// number of integers the message queue can hold

int B = atoi(argv[2]);

// number of producers

int P = atoi(argv[3]);

// number of consumers

int C = atoi(argv[4]);

// check for incorrect parameters

if (N < 1 || B < 1 || P < 1 || C < 1){

exit(1);

}

mqd\_t qdes; // queue\_descriptor

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

struct mq\_attr attr; // queue attributes

// initialize a blocking queue

attr.mq\_maxmsg = B;

attr.mq\_msgsize = sizeof(int);

attr.mq\_flags = 0;

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// open up the queue

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

qdes = mq\_open(qname, O\_RDWR | O\_CREAT, mode,

&attr);

// check if queue was opened successfully

if (qdes == -1 ) {

perror("mq\_open() in process main");

exit(1);

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// open up the semaphore

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

sem\_t \*sem;

sem = sem\_open(sem\_name, O\_RDWR | O\_CREAT,

mode, N);

// check if semaphore was opened successfully

if (sem == SEM\_FAILED ) {

perror("sem\_open() in process main");

exit(1);

}

// start tracking execution time

double t\_before\_fork = get\_time();

pid\_t pid\_child;

int p,c;

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// spawn producer processes

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

argv[0] = "producer";

// loop to generate P producer processes

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

// properly cast int to char via pointer

// note: in C int is recognized as char

char\* producer\_id = (char\*)&p;

// assign unique producer id

argv[2] = producer\_id;

pid\_child = fork();

// check if forking was successful

if (pid\_child < 0){

perror("fork()");

exit(1);

} else if (pid\_child == 0) {

// produce and send items

execvp("./producer", argv);

}

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// spawn consumer processes

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

argv[0] = "consumer";

// loop to generate C consumer processes

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

// properly cast int to char via pointer

// note: in C int is recognized as char

char\* consumer\_id = (char\*)&c;

// assign unique consumer id

argv[2] = consumer\_id;

pid\_child = fork();

// check if forking was successful

if (pid\_child < 0){

perror("fork()");

exit(1);

} else if (pid\_child == 0) {

// receive and consume items

execvp("./consumer", argv);

}

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// output timing results and data

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int status\_child, child\_pid;

// loop infinitely until all children are depleted

for(;;){

// wait for all children to finish

child\_pid = wait(&status\_child);

if (child\_pid == -1) {

// stop timer as execution is finished

double t\_last\_consumed = get\_time();

// output results for analysis

printf("System execution time: %f seconds\n",

t\_last\_consumed - t\_before\_fork);

// break out of loop; all children finished

break;

}

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// final checks and cleaning up

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// close queue

if (mq\_close(qdes) == -1) {

perror("mq\_close() failed in process main");

exit(2);

}

// remove queue

if (mq\_unlink(qname) == -1) {

perror("mq\_unlink() failed in process main");

exit(3);

}

if (sem\_close(sem) == -1) {

perror("sem\_close() failed in process main");

exit(2);

}

if (sem\_unlink(sem\_name) == -1) {

perror("sem\_unlink() failed in process main");

exit(3);

}

return 0;

}

/\* Helper function to get time in seconds \*/

double get\_time() {

struct timeval tv;

gettimeofday(&tv, NULL);

return (tv.tv\_sec + tv.tv\_usec / 1000000.0);

}

**Code – Threading implementation:**

/\*

\* processes\_main.c

\* ECE254 Group 34

\* By: Tianyi Zhang and Kwok Yin Timothy Tong

\* University of Waterloo Computer Engineering

\* Fall 2015

\*/

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <time.h>

#include <semaphore.h>

#include <mqueue.h>

#include <math.h>

void \* produce(void \* id);

void \* consume(void \* id);

double get\_time();

struct Buffer\_node {

struct Buffer\_node \* next;

int val;

};

// Global vars

int N, B, P, C;

int count = 0; // size of linked list

int items\_produced = 0;

int items\_consumed = 0;

struct Buffer\_node \* head; // head of the linked list

struct Buffer\_node \* tail; // tail of the linked list

sem\_t \*sem\_crit\_region,

\*sem\_space\_avail,

\*sem\_item\_avail;

char \* sem\_crit\_reg\_name = "sem\_crit";

char \* sem\_space\_avail\_name = "sem\_space";

char \* sem\_item\_avail\_name = "sem\_item";

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

double t\_before;

double t\_after;

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

// format should be ./produce <N> <B> <P> <C>

if (argc != 5) {

exit(1);

}

// number of integers the producer should produce

N = atoi(argv[1]);

// number of integers the message queue can hold

B = atoi(argv[2]);

// number of producers

P = atoi(argv[3]);

// number of consumers

C = atoi(argv[4]);

// check for incorrect parameters

if (N < 1 || B < 1 || P < 1 || C < 1){

exit(1);

}

sem\_crit\_region = sem\_open(sem\_crit\_reg\_name, O\_RDWR | O\_CREAT,

mode, 1);

sem\_space\_avail = sem\_open(sem\_space\_avail\_name, O\_RDWR |

O\_CREAT, mode, B);

sem\_item\_avail = sem\_open(sem\_item\_avail\_name, O\_RDWR |

O\_CREAT, mode, 0);

if(sem\_item\_avail == SEM\_FAILED || sem\_space\_avail ==

SEM\_FAILED || sem\_crit\_region == SEM\_FAILED){

perror("sem\_open");

}

pthread\_t producers\_threads[P];

pthread\_t consumers\_threads[C];

t\_before = get\_time();

int p, c;

// SPAWN PRODUCER THREADS ==> generates messages

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

int \* producer\_id = malloc(sizeof(int));

\*producer\_id = p;

pthread\_create(&(producers\_threads[p]), NULL, produce,

producer\_id);

}

// SPAWN CONSUMER THREADS ==> consumes messages

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

int \* consumer\_id = malloc(sizeof(int));

\*consumer\_id = c;

pthread\_create(&(consumers\_threads[c]), NULL, consume,

consumer\_id);

}

int p\_j, c\_j;

// Join all producer threads

for (p\_j = 0; p\_j < P; p\_j++) {

pthread\_join(producers\_threads[p\_j], NULL);

}

// Join all consumer threads

for (c\_j = 0; c\_j < C; c\_j++) {

pthread\_join(consumers\_threads[c\_j], NULL);

}

t\_after = get\_time();

printf("System execution time: %f seconds\n",

t\_after - t\_before);

// Cleanup – closing and unlinking semaphores

if(sem\_close(sem\_crit\_region) != 0 ||

sem\_close(sem\_space\_avail) != 0 ||

sem\_close(sem\_item\_avail) != 0){

perror("sem\_close");

}

if (sem\_unlink(sem\_crit\_reg\_name) != 0 ||

sem\_unlink(sem\_space\_avail\_name) != 0 ||

sem\_unlink(sem\_item\_avail\_name) != 0) {

perror("sem\_unlink");

}

return 0;

}

/\* Helper function to get time in seconds \*/

double get\_time() {

struct timeval tv;

gettimeofday(&tv, NULL);

return (tv.tv\_sec + tv.tv\_usec / 1000000.0);

}

/\* Routine of producer \*/

void \*produce(void \*id){

int producer\_id = \*((int \*)id);

int i;

for (i = producer\_id; i < N && items\_produced < N; i += P) {

int message = i;

// Create a node for the newly generated message

struct Buffer\_node \* new\_node = malloc(sizeof(struct

Buffer\_node));

new\_node->next = NULL;

new\_node->val = message;

// Wait till space is available

sem\_wait(sem\_space\_avail);

sem\_wait(sem\_crit\_region);

/\* Critical region \*/

// Append new message node to tail

if (head == NULL) {

head = new\_node;

tail = new\_node;

} else {

tail->next = new\_node;

tail = new\_node;

}

count++;

items\_produced++;

/\* End of critical region \*/

sem\_post(sem\_crit\_region);

// Increment count of item\_avail semaphore so consumers can

get through.

sem\_post(sem\_item\_avail);

}

}

/\* Routine of consumer \*/

void \*consume(void \*id){

int consumer\_id = \*((int \*)id);

int message;

while (items\_consumed < N) {

sem\_trywait(sem\_item\_avail);

sem\_wait(sem\_crit\_region);

/\* Critical region \*/

if(head != NULL){

message = head->val; // take from the head

if (count == 1) {

free(head);

head = NULL;

tail = NULL;

} else {

struct Buffer\_node \* tmp = head;

head = head->next;

tmp = NULL;

}

count--;

items\_consumed++;

if (sqrt(message) == (int)(sqrt(message))) {

printf("%d %d %d\n\r", consumer\_id, message,

(int)(sqrt(message)));

}

}

/\* End of critical region \*/

sem\_post(sem\_crit\_region);

sem\_post(sem\_space\_avail);

}

}