ECE 254 lab 4 report

Inter-thread performance

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | B | P | C | Time |
| 100 | 4 | 1 | 1 | 0.00084 |
| 100 | 4 | 1 | 2 | 0.000753 |
| 100 | 4 | 1 | 3 | 0.000957 |
| 100 | 4 | 2 | 1 | 0.000874 |
| 100 | 4 | 3 | 1 | 0.001084 |
| 100 | 8 | 1 | 1 | 0.000703 |
| 100 | 8 | 1 | 2 | 0.000762 |
| 100 | 8 | 1 | 3 | 0.000872 |
| 100 | 8 | 2 | 1 | 0.000804 |
| 100 | 8 | 3 | 1 | 0.000979 |
| 398 | 8 | 1 | 1 | 0.001316 |
| 398 | 8 | 1 | 2 | 0.00178 |
| 398 | 8 | 1 | 3 | 0.002047 |
| 398 | 8 | 2 | 1 | 0.001564 |
| 398 | 8 | 3 | 1 | 0.001918 |

Table for inter-thread: Timing measurement data table for given (N, B, P, C) values

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

Average time = 0.002047 s

Standard deviation = 0.000319994 s

Inter-process performance

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | B | P | C | Time |
| 100 | 4 | 1 | 1 | 0.000915 |
| 100 | 4 | 1 | 2 | 0.001003 |
| 100 | 4 | 1 | 3 | 0.001003 |
| 100 | 4 | 2 | 1 | 0.001164 |
| 100 | 4 | 3 | 1 | 0.001184 |
| 100 | 8 | 1 | 1 | 0.001025 |
| 100 | 8 | 1 | 2 | 0.001253 |
| 100 | 8 | 1 | 3 | 0.001355 |
| 100 | 8 | 2 | 1 | 0.001147 |
| 100 | 8 | 3 | 1 | 0.001171 |
| 398 | 8 | 1 | 1 | 0.001634 |
| 398 | 8 | 1 | 2 | 0.001845 |
| 398 | 8 | 1 | 3 | 0.002071 |
| 398 | 8 | 2 | 1 | 0.001954 |
| 398 | 8 | 3 | 1 | 0.001949 |

Table for inter-process: Timing measurement data table for given (N, B, P, C) values

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

Average time = 0. 002071s

Standard deviation = 0.000604

# Comparison of the two approaches

It can be seen that the multi-threads approach has a slightly lower average runtime and standard deviation in the test case we are looking at. This indicates that multi-thread basically run faster than the multi-processes and the result is more consistence for multi-thread.

The main reason behind this is that multi-thread’s context switching is faster than the one for multi-processes.

Secondly, this difference is more obvious when there is an imbalance between the number of producers and consumers. In this case, the producers and consumers get blocked more often and amplify the difference of speed in context switching.

Thirdly, since multi-threads use shared variable, it is easier and faster the access the variable in order to identify the end of execution and the blocking mechanism.

All of the above reason aligns with the results listed above.

# Appendix

## Source code for multi-thread approach

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <semaphore.h>

#include <math.h>

*//* #define DEBUG true;

*//* Global variables

int NUM\_INT;

int BUFFER\_SIZE;

int NUM\_PROD;

int NUM\_CON;

int counter = 0;

int \*buffer;

int \*pid;

int \*cid;

int buf\_index = -1;

int ctotal = 0;

int cnum;

sem\_t spaces;

sem\_t items;

pthread\_mutex\_t prod\_mutex;

pthread\_mutex\_t con\_mutex;

pthread\_mutex\_t buffer\_mutex;

struct timeval tv;

double t1;

double t2;

int produce(int pid)

{

#ifdef DEBUG

printf("Producer %d produced %d.\n", pid, counter + 1);

#endif

return counter++;

}

void consume(int cid, int value, int ctotal)

{

#ifdef DEBUG

printf("Consumer %d consumed %d.\n", cid, ctotal);

#endif

int sqrt\_value = sqrt((double)value);

if (sqrt\_value \* sqrt\_value == value)

{

printf("%d %d %d\n", cid, value, sqrt\_value);

}

}

void \*producer(void \*arg)

{

int \*pid = (int \*)arg;

while (1)

{

pthread\_mutex\_lock(&prod\_mutex);

*//* When all items are produced

if (counter == NUM\_INT)

{

pthread\_mutex\_unlock(&prod\_mutex);

break;

}

*//* Produce item if mod value is pid

else if (counter % NUM\_PROD == \*pid)

{

int v = produce(\*pid);

pthread\_mutex\_unlock(&prod\_mutex);

sem\_wait(&spaces);

pthread\_mutex\_lock(&buffer\_mutex);

buffer[++buf\_index] = v;

pthread\_mutex\_unlock(&buffer\_mutex);

sem\_post(&items);

}

else

{

pthread\_mutex\_unlock(&prod\_mutex);

}

}

pthread\_exit(NULL);

}

void \*consumer(void \*arg)

{

int \*cid = (int \*)arg;

while (1)

{

pthread\_mutex\_lock(&con\_mutex);

*//* Reduce number of consumers if number of remaining items is less than number of remaining consumers

if (NUM\_INT - ctotal < cnum)

{

cnum--;

pthread\_mutex\_unlock(&con\_mutex);

break;

}

pthread\_mutex\_unlock(&con\_mutex);

sem\_wait(&items);

pthread\_mutex\_lock(&buffer\_mutex);

int temp = buffer[buf\_index];

buffer[buf\_index--] = -1;

ctotal++;

pthread\_mutex\_unlock(&buffer\_mutex);

sem\_post(&spaces);

consume(\*cid, temp, ctotal);

}

pthread\_exit(NULL);

}

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

{

*//* Check the number of arguments

if (argc != 5)

{

printf("Error! Wrong number of arguments.\n");

return -1;

}

*//* Assign arguments to global variables

NUM\_INT = atoi(argv[1]);

BUFFER\_SIZE = atoi(argv[2]);

NUM\_PROD = atoi(argv[3]);

NUM\_CON = cnum = atoi(argv[4]);

*//* Check if arguments are valid

if (NUM\_INT <= 0 || BUFFER\_SIZE <= 0 || NUM\_PROD <= 0 || NUM\_CON <= 0)

{

printf("Error! Invalid arguments.\nN: %d, B: %d, P: %d, C: %d.\n", NUM\_INT, BUFFER\_SIZE, NUM\_PROD, NUM\_CON);

return -1;

}

*//* Set producer counter

counter = 0;

*//* Allocate and assign buffer

buffer = malloc(BUFFER\_SIZE \* sizeof(int));

int i, j, k;

*//* Initialize buffer

for (i = 0; i < BUFFER\_SIZE; i++)

{

buffer[i] = -1;

}

*//* Initialize semaphores and mutexes

sem\_init(&spaces, 0, BUFFER\_SIZE);

sem\_init(&items, 0, 0);

pthread\_mutex\_init(&prod\_mutex, NULL);

pthread\_mutex\_init(&con\_mutex, NULL);

pthread\_mutex\_init(&buffer\_mutex, NULL);

*//* Initialize thread variables

pid = malloc(NUM\_PROD \* sizeof(int));

cid = malloc(NUM\_CON \* sizeof(int));

pthread\_t prod[NUM\_PROD];

pthread\_t con[NUM\_CON];

*//* Initialize timer

gettimeofday(&tv, NULL);

t1 = tv.tv\_sec + tv.tv\_usec / 1000000.0;

*//* Create producer threads

for (j = 0; j < NUM\_PROD; j++)

{

pid[j] = j;

pthread\_create(&prod[j], NULL, producer, &pid[j]);

}

*//* Create consumer threads

for (k = 0; k < NUM\_CON; k++)

{

cid[k] = k;

pthread\_create(&con[k], NULL, consumer, &cid[k]);

}

*//* Wait for producer threads exit

for (j = 0; j < NUM\_PROD; j++)

{

pthread\_join(prod[j], NULL);

}

*//* Wait for consumer threads exit

for (k = 0; k < NUM\_CON; k++)

{

pthread\_join(con[k], NULL);

}

*//* Calculate time after all consumers exit

gettimeofday(&tv, NULL);

t2 = tv.tv\_sec + tv.tv\_usec / 1000000.0;

*//* Print time

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

*//* Deallocate pointers

free(buffer);

free(pid);

free(cid);

*//* Destroy semaphores and mutexes

sem\_destroy(&spaces);

sem\_destroy(&items);

pthread\_mutex\_destroy(&prod\_mutex);

pthread\_mutex\_destroy(&con\_mutex);

pthread\_mutex\_destroy(&buffer\_mutex);

pthread\_exit(0);

}

## Source code for multi-thread approach

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <math.h>

#include <sys/types.h>

#include <mqueue.h>

// #define DEBUG true

#define LINUX true

int NUM\_INT, BUFFER\_SIZE, NUM\_PROD, NUM\_CON, child\_pid, status, i;

int ioCounter = 0, multiProcess = 0;

struct timeval tv;

double t1;

double t2;

#ifdef LINUX

// Initialize the message queue

int spaceQueueMsgSize = 1, itemQueueMsgSize;

mqd\_t spaceQueue, itemQueue;

char spaceQueueName[] = "/254\_harry\_space", itemQueueName[] = "/254\_harry\_item";

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

struct mq\_attr attr;

pid\_t child\_pid;

#endif

int signalSpaceQueue(){

if (mq\_send(spaceQueue, &NUM\_INT, spaceQueueMsgSize, 0) != -1) return 1;

perror("Send to space queue failed");

return 0;

}

int waitSpaceQueue(){

if (mq\_receive(spaceQueue, &NUM\_INT, spaceQueueMsgSize, 0) != -1) return 1;

perror("Receive from space queue failed");

return 0;

}

int signalItemQueue(int\* ptr){

if (mq\_send(itemQueue, ptr, itemQueueMsgSize, 0) != -1) return 1;

perror("Send to item queue failed");

return 0;

}

int waitItemQueue(int\* ptr){

if (mq\_receive(itemQueue, ptr, itemQueueMsgSize, 0) != -1) return 1;

perror("Receive from item queue failed");

return 0;

}

int produce(int pid, int i) {

#ifdef DEBUG

printf("Producer %d produced %d.\n", pid, (pid + NUM\_PROD \* i));

#endif

// generate a INT i where i%P = pid

// E.g.: pid = 3 with 7 producers -> 3,10,17,...

return (pid + NUM\_PROD \* i);

}

//int consume(int cid, int value, int ctotal) {

int consume(int cid, int value) {

#ifdef DEBUG

printf("Consumer %d consumed.\n", cid);

#endif

// find the square root in the buffer and if the number is prefect square,

// print: cid value squareRootOfValue

int sqrt\_value = sqrt((double)value);

if (sqrt\_value \* sqrt\_value == value) {

printf("%d %d %d\n", cid, value, sqrt\_value);

}

}

void \*producer(int pid) {

int newValue, i = 0;

do {

newValue = produce(pid, i);

waitSpaceQueue();

signalItemQueue(&newValue);

ioCounter--;

i++;

} while (ioCounter);

}

void \*consumer(int cid) {

int newValue;

do {

waitItemQueue(&newValue);

signalSpaceQueue();

consume(cid, newValue);

ioCounter--;

} while (ioCounter);

}

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

/\*

#ifdef DEBUG

NUM\_PROD = 5;

NUM\_CON = 6;

BUFFER\_SIZE = 10;

NUM\_INT = 20;

#endif

\*/

// Check the number of arguments

if (argc != 5) {

printf("Error! Wrong number of arguments.\n");

return -1;

}

// Assign arguments to global variables

NUM\_INT = atoi(argv[1]);

BUFFER\_SIZE = atoi(argv[2]);

NUM\_PROD = atoi(argv[3]);

NUM\_CON = atoi(argv[4]);

// Check if arguments are valid

if (NUM\_INT <= 0 || BUFFER\_SIZE <= 0 || NUM\_PROD <= 0 || NUM\_CON <= 0) {

printf("Error! Invalid arguments.\nN: %d, B: %d, P: %d, C: %d.\n", NUM\_INT, BUFFER\_SIZE, NUM\_PROD, NUM\_CON);

return -1;

}

// Check if the buffer size is enough

if (BUFFER\_SIZE > 9) {

printf("Error! Invalid arguments.\nB: %d. \n",BUFFER\_SIZE);

return -1;

}

#ifdef LINUX

itemQueueMsgSize = sizeof(NUM\_INT);

attr.mq\_maxmsg = BUFFER\_SIZE;

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

attr.mq\_msgsize = spaceQueueMsgSize; // notification queue require no msg size

spaceQueue = mq\_open(spaceQueueName, O\_RDWR | O\_CREAT, mode, &attr);

attr.mq\_msgsize = itemQueueMsgSize;

itemQueue = mq\_open(itemQueueName, O\_RDWR | O\_CREAT, mode, &attr);

if (!itemQueue || !spaceQueue) {

perror("SETUP: Filling space queue failed");

exit(1);

}

for (i = 0; i < BUFFER\_SIZE; i++) {

// filling up the whole space queue

if (!signalSpaceQueue()) {

perror("SETUP: Filling space queue failed");

exit(2);

}

}

#endif

// Initialize timer

gettimeofday(&tv, NULL);

t1 = tv.tv\_sec + tv.tv\_usec / 1000000.0;

// Create consumers

for (i = 0; i < NUM\_CON; i++){

child\_pid = fork ();

if (child\_pid == 0) {

/\* This is the consumer process. \*/

#ifdef DEBUG

printf("%dth Consumer \n", i+1);

#endif

ioCounter = NUM\_INT / NUM\_CON;

ioCounter = (NUM\_INT % NUM\_CON) > i ? ++ioCounter : ioCounter;

consumer(i);

exit(0);

}

}

// Create producers

for (i = 0; i < NUM\_PROD; i++){

child\_pid = fork ();

if (child\_pid == 0) {

/\* This is the producers process. \*/

#ifdef DEBUG

printf("%dth Producer \n", i+1);

#endif

ioCounter = NUM\_INT / NUM\_PROD;

ioCounter = (NUM\_INT % NUM\_PROD) > i ? ++ioCounter : ioCounter;

producer(i);

exit(0);

}

}

// Wait for all the processes to be finished

multiProcess = NUM\_PROD + NUM\_CON;

/\* This is the parent process. \*/

while (multiProcess > 0) {

child\_pid = wait(&status);

#ifdef DEBUG

printf("Child with PID %ld exited with status 0x%x.\n", (long)child\_pid, status);

#endif

--multiProcess;

}

gettimeofday(&tv, NULL);

t2 = tv.tv\_sec + tv.tv\_usec / 1000000.0;

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

#ifdef DEBUG

printf("all processes done and exited\n");

#endif

#ifdef LINUX

// Deleting the queues

if (mq\_close(spaceQueue) == -1 || mq\_close(itemQueue) == -1 ) {

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

exit(3);

}

#ifdef DEBUG

printf("all queues are closed \n");

#endif

if (mq\_unlink(spaceQueueName) != 0 || mq\_unlink(itemQueueName) != 0 ) {

perror("mq\_unlink() failed");

exit(4);

}

#ifdef DEBUG

printf("all queues are closed \n");

#endif

#endif

return 0;

}