**Lab2 Report**

**1. Linux Time Measurement Data (see section 3 for important notes)**

Inter-process Communication with Message Queue Approach:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Linux Message Passing** | | | | | |
| **P** | **C** | **N** | **B** | **Time(s)** | **Standard Deviation** |
| 1 | 1 | 100 | 4 | 99.09929 | 0.000404669 |
| 2 | 1 | 100 | 4 | 99.09812 | 0.000425769 |
| 3 | 1 | 100 | 4 | 99.09958 | 0.000304665 |
| 1 | 2 | 100 | 4 | 99.09945 | 0.000404669 |
| 1 | 3 | 100 | 4 | 99.10039 | 0.001028979 |
| 1 | 1 | 100 | 8 | 99.09949 | 0.000604669 |
| 2 | 1 | 100 | 8 | 99.09845 | 0.000293869 |
| 3 | 1 | 100 | 8 | 99.09911 | 0.000294657 |
| 1 | 2 | 100 | 8 | 99.09945 | 0.000429487 |
| 1 | 3 | 100 | 8 | 99.09916 | 0.000739403 |
| 1 | 1 | 398 | 8 | 397.3983 | 0.000928929 |
| 2 | 1 | 398 | 8 | 397.3985 | 0.00039574 |
| 3 | 1 | 398 | 8 | 397.3972 | 0.000302947 |
| 1 | 2 | 398 | 8 | 397.3979 | 0.00101938 |
| 1 | 3 | 398 | 8 | 397.3985 | 0.000429486 |

Inter-thread Communication with Shared Memory Approach:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Linux Shared Memory** | | | | | |
| **P** | **C** | **N** | **B** | **Time(s)** | **Standard Deviation** |
| 1 | 1 | 100 | 4 | 99.095637 | 0.000154986 |
| 2 | 1 | 100 | 4 | 99.0968816 | 0.000209527 |
| 3 | 1 | 100 | 4 | 99.099852 | 0.001215869 |
| 1 | 2 | 100 | 4 | 48.0975055 | 0.001724498 |
| 1 | 3 | 100 | 4 | 48.0963575 | 0.000158937 |
| 1 | 1 | 100 | 8 | 99.096105 | 0.00020204 |
| 2 | 1 | 100 | 8 | 99.096014 | 0.001334298 |
| 3 | 1 | 100 | 8 | 99.096247 | 0.00059592 |
| 1 | 2 | 100 | 8 | 48.098059 | 0.000176728 |
| 1 | 3 | 100 | 8 | 48.095279 | 0.00015893 |
| 1 | 1 | 398 | 8 | 397.384899 | 0.00061842 |
| 2 | 1 | 398 | 8 | 397.383297 | 0.00089592 |
| 3 | 1 | 398 | 8 | 397.38512 | 0.000158924 |
| 1 | 2 | 398 | 8 | 179.392891 | 0.00109592 |
| 1 | 3 | 398 | 8 | 179.392891 | 0.00118892 |

With the message passing approach, we have noticed that the time needed to finish sending and receiving N integers does not noticeably change with the increase of producers or consumers. However, the time significantly increases as the number of integers required to transmit (N) increase from 100 to 398.

On the other hand, although the time needed to finish sending and receiving N integers does not noticeably change as the number of producers increase, the time does significantly decrease when the number of producers increases from 1 to 2. The time also significantly increases as the number of integers required to transmit (N) increase from 100 to 398.

The advantage of using message queue is that it is easier to implement than using shared memory since we do not have to care about shared memory protection. The disadvantage of using message queue is that it is generally slower than using shared memory and this is particularly obvious when we have more than one consumer.

On the other hand, the disadvantage of using shared memory is that it is more complex to implement than using message queue because we have to take care of shared memory protection in our program, and deadlock may occur if semaphores and mutexes are not used correctly. The advantage of using shared memory is that it is faster than using message queues and it allows convenient communication between threads.

**2. Keil Time Measurement Data (see section 3 for important notes)**

Inter-process Communication with Mailbox Approach:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Keil Message Passing(Sys Time Tick set to 10us)** | | | | |
| **P** | **C** | **N** | **B** | **Time(s)** |
| 1 | 1 | 100 | 4 | 0.03613 |
| 2 | 1 | 100 | 4 | 0.03673 |
| 3 | 1 | 100 | 4 | 0.03656 |
| 1 | 2 | 100 | 4 | 0.0157 |
| 1 | 3 | 100 | 4 | 0.00559 |
| 1 | 1 | 100 | 8 | 0.03604 |
| 2 | 1 | 100 | 8 | 0.0367 |
| 3 | 1 | 100 | 8 | 0.037 |
| 1 | 2 | 100 | 8 | 0.01711 |
| 1 | 3 | 100 | 8 | 0.01163 |
| 1 | 1 | 398 | 8 | 0.14366 |
| 2 | 1 | 398 | 8 | 0.14652 |
| 3 | 1 | 398 | 8 | 0.14725 |
| 1 | 2 | 398 | 8 | 0.06777 |
| 1 | 3 | 398 | 8 | 0.04672 |

Inter-thread Communication with Shared Memory Approach:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Keil Shared Memory** | | | | |
| **P** | **C** | **N** | **B** | **Time(s)** |
| 1 | 1 | 100 | 4 | 0.03409 |
| 2 | 1 | 100 | 4 | 0.03403 |
| 3 | 1 | 100 | 4 | 0.03406 |
| 1 | 2 | 100 | 4 | 0.01708 |
| 1 | 3 | 100 | 4 | 0.01159 |
| 1 | 1 | 100 | 8 | 0.0341 |
| 2 | 1 | 100 | 8 | 0.03398 |
| 3 | 1 | 100 | 8 | 0.03397 |
| 1 | 2 | 100 | 8 | 0.01711 |
| 1 | 3 | 100 | 8 | 0.01161 |
| 1 | 1 | 398 | 8 | 0.13558 |
| 2 | 1 | 398 | 8 | 0.13502 |
| 3 | 1 | 398 | 8 | 0.13539 |
| 1 | 2 | 398 | 8 | 0.06777 |
| 1 | 3 | 398 | 8 | 0.04532 |

Standard deviation calculated for all cases running on the Keil LPC1768 board is always zero because no matter how many times we run the program, it still gives the same timing measurement.

With both approaches, the time needed to finish sending and receiving N integers does not noticeably change as the number of producer increases, but it does decrease as the number of consumer increases. In general, the program with the shared memory approach is a little faster than with the message passing approach.

The advantage of using the message passing approach on the Keil LPC1768 board to solve the problem is that we do not have to care about shared memory protection. The disadvantage of this approach is that it is more complex to implement and generally slower than the shared memory approach.

On the other hand, the advantage of using the shared memory approach is that it is easier to implement on the Keil board (just need some mutexes and semaphores, a lot easier to implement than using mailboxes), and it is generally faster than using the other approach. The disadvantage of using this approach is that deadlock may occur if mutexes and semaphores are not used properly, so we have to be very careful when using the shared memory approach.

**3. Important Notes**

Linux part:

* For the message queue approach, time is measured by subtracting the start time of program from the end time of the program.
* For the shared memory approach, timing measurement data is printed out at the end of the program.
* For both approaches, producers will be put to sleep for 1 second after appending an integer to the message queue/shared buffer.

Keil part:

* For both approaches, timing measurement data is calculated by multiplying the timer tick value printed on the LCD screen by 0.00001 (10us).

**Appendix**

1. Linux inter-process communication with message queue.

#include <string.h>

#include <stdio.h>

#include <sys/types.h>

#include <stdlib.h>

#include <stdbool.h>

#include <mqueue.h>

#include <time.h>

#include <signal.h>

#include <sys/stat.h>

#include <math.h>

#define P 1

#define C 1

#define N 100 // The number of messages the sender is going to send.

#define B 4 // Number of messages the mqueue can hold.

struct timeval tv;

double t1;

double t2;

void producer(int pid) {

if (pid == 0) {

gettimeofday(&tv, NULL); // Measure time. (get t1)

t1 = tv.tv\_sec + tv.tv\_usec/1000000.0; // Convert time to seconds.

printf("start time:%f\n", t1);

}

mqd\_t qdes; // Declare message queue descriptors(handler).

char qname[] = "/msgqueuetwo"; //Queue name.

mode\_t mode = S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IWGRP | S\_IROTH | S\_IWOTH; // Mode of file.

struct mq\_attr attr; // Queue attribute.

attr.mq\_maxmsg = B; // Set message queue size.

attr.mq\_msgsize = sizeof(int); // Set message size.

attr.mq\_flags = 0;

qdes = mq\_open(qname, O\_RDWR | O\_CREAT, mode, &attr); // Create and open the message queue.

int i = 0;

// Produce integer.

while(i < N) {

if (i%P == pid) {

if (mq\_send(qdes, (char \*)&i, sizeof(int), 0) == -1) {

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

} else {

printf("sending %i\n", i);

}

}

i++;

sleep(1);

}

}

void consumer(int cid) {

//printf("in consumer.\n");

mqd\_t qdes;

char qname[] = "/msgqueuetwo";

mode\_t mode = S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IWGRP | S\_IROTH | S\_IWOTH;

struct mq\_attr attr;

attr.mq\_maxmsg = B;

attr.mq\_msgsize = sizeof(int);

attr.mq\_flags = 0;

qdes = mq\_open(qname, O\_RDONLY, mode, &attr);

if (qdes == -1 ) {

perror("mq\_open()");

exit(1);

}

double root = 0;

while(true) {

int count = 0;

int i;

struct timespec ts = {time(0) + 2, 0};

// Receive messages.

if (mq\_timedreceive(qdes, (char \*) &i, sizeof(int), 0, &ts) == -1) {

count++;

} else {

printf("%i is consumed.\n", i);

root = sqrt((double)i);

if ((root - (int)root) == 0) {

printf("cid:%d, i:%d, root:%f\n", cid, i, root);

}else{

printf("root:%f\n", root);

}

}

if (i == (N-1)) {

gettimeofday(&tv, NULL); // Measure time. (get t2)

t2 = tv.tv\_sec + tv.tv\_usec/1000000.0; // Convert time to seconds.

printf("end time:%f\n", t2);

break;

}

if (count == 3) {

break;

}

}

}

int main() {

pid\_t ppid[P];

pid\_t cpid[C];

pid\_t parentpid = getpid();

int i;

int j;

// Create producers and consumers.

for(i = 0; i < (P - 1); i++) {

ppid[i] = fork();

if (ppid[i] < 0) {

printf("Child not created.\n");

} else if (ppid[i] == 0) {

producer(i+1);

break;

} else {

}

}

if (getpid() == parentpid) {

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

cpid[j] = fork();

if (cpid[j] < 0) {

printf("Child not created.\n");

} else if (cpid[j] == 0) {

consumer(j);

break;

} else {

}

}

}

if (getpid() == parentpid) {

producer(0);

}

return 0;

}

2. Linux inter-thread communication with shared memory.

#include <pthread.h>

#include <stdio.h>

#include <semaphore.h>

#define B 4

#define N 100

#define P 1

#define C 1

int buffer[B];

int index\_in = 0;

int index\_out = 0;

int sent;

int receive;

struct timeval tv;

double t1;

double t2;

sem\_t sem\_mutex; // buffer semaphore.

sem\_t b; // number of messages currently in queue.

sem\_t e; // number of empty spaces in queue.

// Producer 0.

void \*Producer0() {

int pid = 0; // Producer id.

int i = 0;

while(sent < N) {

if (i%P == pid) {

sem\_wait(&e); // Check if buffer is empty.

sem\_wait(&sem\_mutex); // Lock buffer.

// Put integer in buffer.

buffer[index\_in] = i;

index\_in = (index\_in + 1)%B;

sent++;

sem\_post(&b); // More integers in buffer.

sem\_post(&sem\_mutex); // Unlock buffer.

}

i++;

sleep(1);

}

}

// Producer 1.

void \*Producer1() {

int pid = 1;

int i = 0;

while(sent < N) {

if (i%P == pid) {

sem\_wait(&e);

sem\_wait(&sem\_mutex);

buffer[index\_in] = i;

index\_in = (index\_in + 1)%B;

sent++;

sem\_post(&b);

sem\_post(&sem\_mutex);

}

i++;

sleep(1);

}

}

// Producer 2.

void \*Producer2() {

int pid = 2;

int i = 0;

while(sent < N) {

if (i%P == pid) {

sem\_wait(&e);

sem\_wait(&sem\_mutex);

buffer[index\_in] = i;

index\_in = (index\_in + 1)%B;

sent++;

sem\_post(&b);

sem\_post(&sem\_mutex);

}

i++;

sleep(1);

}

}

void \*Consumer0(){

int cid = 0;

double i = 0;

double root = 0;

while (receive < N) {

sem\_wait(&b);

sem\_wait(&sem\_mutex);

i = buffer[index\_out];

index\_out = (index\_out + 1)%B;

receive++;

sem\_post(&e);

sem\_post(&sem\_mutex);

//printf("received:%d\n", receive);

root = sqrt(i);

if ((root - (int)root) == 0) {

// print cid, root, i.

printf("cid:%d i:%f root:%f\n", cid, i, root);

} else {

// print root.

printf("root:%f\n", root);

}

}

if (receive <= N) {

gettimeofday(&tv, NULL); // Measure time. (get t1)

t2 = tv.tv\_sec + tv.tv\_usec/1000000.0; // Convert time to seconds.

}

sleep(2);

}

void \*Consumer1(){

int cid = 1;

double i = 0;

double root = 0;

while (receive < N) {

sem\_wait(&b);

sem\_wait(&sem\_mutex);

i = buffer[index\_out];

index\_out = (index\_out + 1)%B;

receive++;

sem\_post(&e);

sem\_post(&sem\_mutex);

//printf("received:%d", receive);

root = sqrt(i);

if ((root - (int)root) == 0) {

// print cid, root, i.

printf("cid:%d i:%f root:%f\n", cid, i, root);

} else {

// print root.

printf("root:%f\n", root);

}

}

if (receive <= N) {

gettimeofday(&tv, NULL); // Measure time. (get t1)

t2 = tv.tv\_sec + tv.tv\_usec/1000000.0; // Convert time to seconds.

}

sleep(2);

}

void \*Consumer2(){

int cid = 2;

double i = 0;

double root = 0;

while (receive < N) {

sem\_wait(&b);

sem\_wait(&sem\_mutex);

i = buffer[index\_out];

index\_out = (index\_out + 1)%B;

receive++;

sem\_post(&e);

sem\_post(&sem\_mutex);

//printf("received:%d", receive);

root = sqrt(i);

if ((root - (int)root) == 0) {

// print cid, root, i.

printf("cid:%d i:%f root:%f\n", cid, i, root);

} else {

// print root.

printf("root:%f\n", root);

}

}

if(receive <= N) {

gettimeofday(&tv, NULL); // Measure time. (get t1)

t2 = tv.tv\_sec + tv.tv\_usec/1000000.0; // Convert time to seconds.

}

sleep(2);

}

int main(){

gettimeofday(&tv, NULL); // Measure time. (get t1)

t1 = tv.tv\_sec + tv.tv\_usec/1000000.0; // Convert time to seconds.

pthread\_t ptid0,ctid0;

pthread\_t ptid1, ptid2;

pthread\_t ctid1, ctid2;

sent = 0;

receive = 0;

//initialize the semaphores

sem\_init(&sem\_mutex,0,1);

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

sem\_init(&e, 0, B);

//creating producer and consumer threads

if(pthread\_create(&ptid0, NULL, Producer0, NULL)){

printf("\n ERROR creating Producer0");

exit(1);

}

if (P >= 2) {

if(pthread\_create(&ptid1, NULL, Producer1, NULL)){

printf("\n ERROR creating Producer1");

exit(1);

}

}

if (P == 3) {

if(pthread\_create(&ptid2, NULL, Producer2, NULL)){

printf("\n ERROR creating Producer2");

exit(1);

}

}

if(pthread\_create(&ctid0, NULL, Consumer0, NULL)) {

printf("\n ERROR creating Consumer0");

exit(1);

}

if (C >= 2) {

if(pthread\_create(&ctid1, NULL, Consumer1, NULL)) {

printf("\n ERROR creating Consumer1");

exit(1);

}

}

if (C == 3) {

if(pthread\_create(&ctid2, NULL, Consumer2, NULL)) {

printf("\n ERROR creating Consumer2");

exit(1);

}

}

// wait for the producers to finish

if(pthread\_join(ptid0, NULL)){

printf("\n ERROR joining thread");

exit(1);

}

if (P >= 2) {

if(pthread\_join(ptid1, NULL)){

printf("\n ERROR joining thread");

exit(1);

}

}

if (P == 3) {

if(pthread\_join(ptid2, NULL)){

printf("\n ERROR joining thread");

exit(1);

}

}

// wait for consumers to finish

if(pthread\_join(ctid0, NULL)) {

printf("\n ERROR joining thread");

exit(1);

}

if (C >= 2) {

if(pthread\_join(ctid1, NULL)){

printf("\n ERROR joining thread");

exit(1);

}

}

if (C == 3) {

if(pthread\_join(ctid2, NULL)) {

printf("\n ERROR joining thread");

exit(1);

}

}

printf("time: %f seconds\n", (t2-t1));

sem\_destroy(&sem\_mutex);

sem\_destroy(&b);

sem\_destroy(&e);

//exit the main thread

pthread\_exit(NULL);

return 1;

}

3. Keil inter-process communication with mailbox.

#include <LPC17xx.h>

#include <RTL.h>

#include "GLCD.h"

#include <stdio.h>

#include <math.h>

int P = 1;

int C = 1;

int n = 100; // Number of messages.

int b = 4; // Mailbox size.

int sent; // Number of sent messages.

int receive; // Number of received messages.

int tick; // System timer tick count. For time measurement.

os\_mbx\_declare (MsgBox, 1); // Declare an RTX mailbox with b entries

U32 mpool[1\*(2\*sizeof(U32))/4 + 3]; // Reserve a memory for b messages

OS\_SEM s; // Control access to the mailbox.

OS\_SEM m; // Number of messages in the mailbox.

OS\_SEM e; // Number of empty spaces in the mailbox.

\_\_task void rec\_task0 (void) {

/\* This task will receive a message. \*/

int cid = 0; // Consumer id.

U32 \*rptr;

int rec;

double i = 0;

double root = 0;

char stringBuffer[20];

while(receive < n) {

os\_sem\_wait(m, 0xf); // Wait for messages.

os\_sem\_wait(s, 0xf);

// Receive message.

rec = os\_mbx\_wait (MsgBox, (void\*\*)&rptr, 0x0);

if (rec == OS\_R\_OK || rec == OS\_R\_MBX) {

i = rptr[0];

\_free\_box (mpool, rptr);

receive++;

os\_sem\_send(e); // More empty space in mailbox.

}

os\_sem\_send(s);

// Calculate and print outputs.

root = sqrt(i);

if ((root - (int)root) == 0) {

// print cid, root, i.

sprintf(stringBuffer, "cid:%d i:%f root:%f", cid, i, root);

GLCD\_DisplayString(0, 0, 1, stringBuffer);

} else {

// print root.

sprintf(stringBuffer, "root:%f", root);

GLCD\_DisplayString(0, 0, 1, stringBuffer);

}

os\_dly\_wait(30);

}

// Measure time.

tick = os\_time\_get();

//sprintf(stringBuffer, "tick:%d", tick);

//GLCD\_DisplayString(0, 0, 1, stringBuffer);

os\_tsk\_delete\_self ();

}

\_\_task void rec\_task1 (void) {

/\* This task will receive a message. \*/

int cid = 1; // Consumer id.

U32 \*rptr;

int rec;

double i = 0;

double root = 0;

char stringBuffer[20];

while(receive < n) {

os\_sem\_wait(m, 0xf); // Wait for messages.

os\_sem\_wait(s, 0xf);

// Receive message.

rec = os\_mbx\_wait (MsgBox, (void\*\*)&rptr, 0x0);

if (rec == OS\_R\_OK || rec == OS\_R\_MBX) {

i = rptr[0];

\_free\_box (mpool, rptr);

receive++;

os\_sem\_send(e); // More empty spaces.

}

os\_sem\_send(s);

// Calculate and print output.

root = sqrt(i);

if ((root - (int)root) == 0) {

// print cid, root, i.

sprintf(stringBuffer, "cid:%d i:%f root:%f", cid, i, root);

GLCD\_DisplayString(0, 0, 1, stringBuffer);

} else {

// print root.

sprintf(stringBuffer, "root:%f", root);

GLCD\_DisplayString(0, 0, 1, stringBuffer);

}

os\_dly\_wait(30);

}

tick = os\_time\_get();

//sprintf(stringBuffer, "tick:%d", tick);

//GLCD\_DisplayString(0, 0, 1, stringBuffer);

os\_tsk\_delete\_self ();

}

\_\_task void rec\_task2 (void) {

/\* This task will receive a message. \*/

int cid = 2; // Consumer id.

U32 \*rptr;

int rec;

double i = 0;

double root = 0;

char stringBuffer[20];

while(receive < n) {

os\_sem\_wait(m, 0xf); // Wait for messages.

os\_sem\_wait(s, 0xf);

// Receive message.

rec = os\_mbx\_wait (MsgBox, (void\*\*)&rptr, 0x0);

if (rec == OS\_R\_OK || rec == OS\_R\_MBX) {

i = rptr[0];

\_free\_box (mpool, rptr);

receive++;

os\_sem\_send(e);

}

os\_sem\_send(s);

// Calculate and print output.

root = sqrt(i);

if ((root - (int)root) == 0) {

// print cid, root, i.

sprintf(stringBuffer, "cid:%d i:%f root:%f", cid, i, root);

GLCD\_DisplayString(0, 0, 1, stringBuffer);

} else {

// print root.

sprintf(stringBuffer, "root:%f", root);

GLCD\_DisplayString(0, 0, 1, stringBuffer);

}

os\_dly\_wait(30);

}

// Measure time.

tick = os\_time\_get();

//sprintf(stringBuffer, "tick:%d", tick);

//GLCD\_DisplayString(0, 0, 1, stringBuffer);

os\_tsk\_delete\_self ();

}

\_\_task void send\_task0 (void) {

/\* This task will send a message. \*/

int pid = 0; // Producer id.

U32 \*mptr;

char stringBuffer[20];

int i = 0;

while(sent < n) {

os\_sem\_wait(e, 0xf); // Wait for an empty space.

os\_sem\_wait(s, 0xf);

// Produce message.

if (i%P == pid) {

mptr = \_alloc\_box (mpool); //Allocate a memory for the message

mptr[0] = i; //Set the message content.

mptr[1] = 0;

// Send message.

if (os\_mbx\_send (MsgBox, mptr, 0x0) == OS\_R\_TMO) {

\_free\_box (mpool, mptr);

} else {

sent++;

os\_sem\_send(m);

}

}

os\_sem\_send(s);

i++;

}

os\_tsk\_delete\_self ();

}

\_\_task void send\_task1 (void) {

/\* This task will send a message. \*/

int pid = 1; // Producer id.

U32 \*mptr;

char stringBuffer[20];

int i = 0;

while(sent < n) {

os\_sem\_wait(e, 0xf); // Wait for an empty space.

os\_sem\_wait(s, 0xf);

// Produce message.

if (i%P == pid) {

mptr = \_alloc\_box (mpool); //Allocate a memory for the message

mptr[0] = i; //Set the message content.

mptr[1] = 0;

// Send message.

if (os\_mbx\_send (MsgBox, mptr, 0x0) == OS\_R\_TMO) {

\_free\_box (mpool, mptr);

} else {

sent++;

os\_sem\_send(m);

}

}

os\_sem\_send(s);

i++;

}

os\_tsk\_delete\_self ();

}

\_\_task void send\_task2 (void) {

/\* This task will send a message. \*/

int pid = 2; // Producer id.

U32 \*mptr;

char stringBuffer[20];

int i = 0;

while(sent < n) {

os\_sem\_wait(e, 0xf); // Wait for empty space.

os\_sem\_wait(s, 0xf);

// Produce message.

if (i%P == pid) {

mptr = \_alloc\_box (mpool); //Allocate a memory for the message

mptr[0] = i; //Set the message content.

mptr[1] = 0;

// Send message.

if (os\_mbx\_send (MsgBox, mptr, 0x0) == OS\_R\_TMO) {

\_free\_box (mpool, mptr);

} else {

sent++;

os\_sem\_send(m);

}

}

os\_sem\_send(s);

i++;

}

os\_tsk\_delete\_self ();

}

\_\_task void initialize\_task (void) {

// Initialize semaphores.

os\_sem\_init(s, 1);

os\_sem\_init(m, 0);

os\_sem\_init(e, b);

sent = 0;

receive = 0;

os\_mbx\_init (MsgBox, sizeof(MsgBox)); // Initialize mailbox.

// Create tasks.

os\_tsk\_create (send\_task0, 1);

if (P >= 2) {

os\_tsk\_create (send\_task1, 1);

}

if (P >= 3) {

os\_tsk\_create (send\_task2, 1);

}

os\_tsk\_create (rec\_task0, 1);

if (C >= 2) {

os\_tsk\_create (rec\_task1, 1);

}

if (C >= 3) {

os\_tsk\_create (rec\_task2, 1);

}

tick = os\_time\_get();

os\_tsk\_delete\_self();

}

void main (void) {

SystemInit();

GLCD\_Init();

GLCD\_Clear(White);

\_init\_box (mpool, sizeof(mpool), sizeof(U32));

os\_sys\_init(initialize\_task);

}

4. Keil inter-thread communication with shared memory.

#include <LPC17xx.h>

#include <RTL.h>

#include "GLCD.h"

#include <stdio.h>

#include <math.h>

int P = 1; // Number of producers.

int C = 1; // Number of consumers.

int index\_in = 0; // Insert index.

int index\_out = 0; // Remove index.

int B = 4; // Buffer size.

int buffer[4];// Create fixed buffer.

int N = 100; // Max. number of integers.

int sent; // Sent count.

int receive; // Receive count.

int tick = 0;

// Create semaphore s, n, r, b, e.

OS\_SEM s;

OS\_SEM n;

OS\_SEM r;

OS\_SEM b;

OS\_SEM e;

\_\_task void rec\_task0 (void) {

char stringBuffer[20];

int cid = 0; // Consumer id.

int i = 0;

double root = 0;

while (receive < N) {

os\_sem\_wait(b, 0xf); // Check if there is something in the buffer.

os\_sem\_wait(s, 0xf); // Check if another task is using the buffer.

// consume number in buffer.

i = buffer[index\_out];

index\_out = (index\_out + 1)%B;

os\_sem\_wait(r, 0xf); // Global variable lock.

receive++;

os\_sem\_send(r); // Global variable unlock.

os\_sem\_send(s); // Unlock buffer.

os\_sem\_send(e); // More empty space.

root = sqrt(i);

if ((root - (int)root) == 0) {

// print cid, root, i.

sprintf(stringBuffer, "cid:%d i:%d root:%f", cid, i, root);

GLCD\_DisplayString(0, 0, 1, stringBuffer);

} else {

// print root.

sprintf(stringBuffer, "root:%f", root);

GLCD\_DisplayString(0, 0, 1, stringBuffer);

}

// Sleep for 0.5 seconds.

os\_dly\_wait(30);

}

tick = os\_time\_get();

//sprintf(stringBuffer, "tick:%d", tick);

//GLCD\_DisplayString(0, 0, 1, stringBuffer);

os\_tsk\_delete\_self();

}

\_\_task void rec\_task1 (void) {

char stringBuffer[20];

int cid = 1; // Consumer id.

int i = 0;

double root = 0;

while (receive < N) {

os\_sem\_wait(b, 0xf); // Check if there is something in the buffer.

os\_sem\_wait(s, 0xf); // Check if another task is using the buffer.

// consume number in buffer.

i = buffer[index\_out];

index\_out = (index\_out + 1)%B;

os\_sem\_wait(r, 0xf); // Global variable lock.

receive++;

os\_sem\_send(r); // Global variable unlock.

os\_sem\_send(s); // Unlock buffer.

os\_sem\_send(e); // More empty space.

root = sqrt(i);

if ((root - (int)root) == 0) {

// print cid, root, i.

sprintf(stringBuffer, "cid:%d i:%d root:%f", cid, i, root);

GLCD\_DisplayString(0, 0, 1, stringBuffer);

} else {

// print root.

sprintf(stringBuffer, "root:%f", root);

GLCD\_DisplayString(0, 0, 1, stringBuffer);

}

// Sleep for 0.5 seconds.

os\_dly\_wait(30);

}

tick = os\_time\_get();

//sprintf(stringBuffer, "tick:%d", tick);

//GLCD\_DisplayString(0, 0, 1, stringBuffer);

os\_tsk\_delete\_self();

}

\_\_task void rec\_task2 (void) {

char stringBuffer[20];

int cid = 2; // Consumer id.

int i = 0;

double root = 0;

while (receive < N) {

os\_sem\_wait(b, 0xf); // Check if there is something in the buffer.

os\_sem\_wait(s, 0xf); // Check if another task is using the buffer.

// consume number in buffer.

i = buffer[index\_out];

index\_out = (index\_out + 1)%B;

os\_sem\_wait(r, 0xf); // Global variable lock.

receive++;

os\_sem\_send(r); // Global variable unlock.

os\_sem\_send(s); // Unlock buffer.

os\_sem\_send(e); // More empty space.

root = sqrt(i);

if ((root - (int)root) == 0) {

// print cid, root, i.

sprintf(stringBuffer, "cid:%d i:%d root:%f", cid, i, root);

GLCD\_DisplayString(0, 0, 1, stringBuffer);

} else {

// print root.

sprintf(stringBuffer, "root:%f", root);

GLCD\_DisplayString(0, 0, 1, stringBuffer);

}

// Sleep for 0.5 seconds.

os\_dly\_wait(30);

}

tick = os\_time\_get();

//sprintf(stringBuffer, "tick:%d", tick);

//GLCD\_DisplayString(0, 0, 1, stringBuffer);

os\_tsk\_delete\_self();

}

\_\_task void send\_task0 (void) {

int pid = 0;

int i = 0;

while (sent < N) {

// Produce integer.

if (i%P == pid) {

os\_sem\_wait(e, 0xf); // Wait for empty space.

os\_sem\_wait(s, 0xf); // Check if another task is using the buffer.

// put number in buffer.

buffer[index\_in] = i;

index\_in = (index\_in + 1)%B;

os\_sem\_wait(n, 0xf); // Global variable lock.

sent++;

os\_sem\_send(n); // Global variable unlock.

os\_sem\_send(s); // Unlock buffer.

os\_sem\_send(b); // More messages in buffer.

// Sleep for 0.5 seconds.

//os\_dly\_wait(30);

}

i++;

}

os\_tsk\_delete\_self();

}

\_\_task void send\_task1 (void) {

int pid = 1;

int i = 0;

while (sent < N) {

if (i%P == pid) {

os\_sem\_wait(e, 0xf);

os\_sem\_wait(s, 0xf);

// put number in buffer.

buffer[index\_in] = i;

index\_in = (index\_in + 1)%B;

os\_sem\_wait(n, 0xf);

sent++;

os\_sem\_send(n);

os\_sem\_send(s);

os\_sem\_send(b);

// Sleep for 0.5 seconds.

//os\_dly\_wait(30);

}

i++;

}

os\_tsk\_delete\_self();

}

\_\_task void send\_task2 (void) {

int pid = 2;

int i = 0;

while (sent < N) {

if (i%P == pid) {

os\_sem\_wait(e, 0xf);

os\_sem\_wait(s, 0xf);

// put number in buffer.

buffer[index\_in] = i;

index\_in = (index\_in + 1)%B;

os\_sem\_wait(n, 0xf);

sent++;

os\_sem\_send(n);

os\_sem\_send(s);

os\_sem\_send(b);

// Sleep for 0.5 seconds.

//os\_dly\_wait(30);

}

i++;

}

os\_tsk\_delete\_self();

}

\_\_task void initialize\_task (void) {

tick = os\_time\_get();

sent = 0;

receive = 0;

os\_sem\_init(s, 1); // s locks the buffer. Initiate it with 1.

os\_sem\_init(n, 1); // n locks the variable "sent". Initiate it with 1.

os\_sem\_init(r, 1); // r locks the variable "receive". Initiate it with 1.

os\_sem\_init(b, 0); // b is the number of messages currently in the buffer. Initiate it with 0.

os\_sem\_init(e, B); // e is the number of empty spaces in the buffer. Initiate it with buffer size.

// Initialize all tasks.

os\_tsk\_create(send\_task0, 1);

if (P >= 2) {

os\_tsk\_create(send\_task1, 1);

}

if (P == 3) {

os\_tsk\_create(send\_task2, 1);

}

os\_tsk\_create(rec\_task0, 1);

if (C >= 2) {

os\_tsk\_create(rec\_task1, 1);

}

if (C == 3) {

os\_tsk\_create(rec\_task2, 1);

}

os\_tsk\_delete\_self();

}

void main (void) {

SystemInit();

GLCD\_Init();

GLCD\_Clear(White);

os\_sys\_init(initialize\_task);

}