University of Waterloo

Faculty of Engineering

Department of Electrical and Computer Engineering

ECE 254

Laboratory 3

Prepared by
Jinming Zhang
UW Student ID Number: 20613667
UW User ID: Jm3zhang @edu.uwaterloo.ca
2B Computer Engineering
and
Lun Jing
UW Student ID Number: 20558988
UW User ID: l7jing@edu.uwaterloo.ca
2B Computer Engineering

26 November 2017

Inter-Thread Communication with Shared Memory Average Timing Measurement Data:

N	B	P	C	Time
100	4	1	1	0.000914
100	4	1	2	0.000818
100	4	1	3	0.000919
100	4	2	1	0.000907
100	4	3	1	0.000933
100	8	1	1	0.000641
100	8	1	2	0.000745
100	8	1	3	0.000862
100	8	2	1	0.000728
100	8	3	1	0.000962
398	8	1	1	0.001148
398	8	1	2	0.001534
398	8	1	3	0.002002
398	8	2	1	0.001417
398	8	3	1	0.001762

Inter-Process Communication with Shared Memory Average Timing Measurement Data:

N	P	B	C	Time
100	4	1	1	0.001535
100	4	1	2	0.001360
100	4	1	3	0.001428
100	4	2	1	0.001336
100	4	3	1	0.000881
100	8	1	1	0.001393
100	8	1	2	0.001269
100	8	1	3	0.001309

100	8	2	1	0.001212
100	8	3	1	0.000978
398	8	1	1	0.002991
398	8	1	2	0.002884
398	8	1	3	0.003064
398	8	2	1	0.002356
398	8	3	1	0.001493

(N,B,P,C) = (398, 8, 1, 3) data from running program 500 times on Linux platform:

- Inter-Thread Communication
  - ⇒ Average timing measurement data: 0.002002 seconds
  - ⇒ Standard deviation: 0.000121
- Inter-Process Communication
  - ⇒ Average timing measurement data: 0.003064 seconds
  - ⇒ Standard deviation: 0.000842

Timing Analysis:

As we can see from the timing Analysis, the time needed for the Average Timing Measurement using Inter-Thread Communication with Shared Memory approach

seems less than the Average Timing Measurement using Inter-Process Communication with Shared Memory approach.

A better picture for this Timing Analysis is to compare the time at one for the specific input. As the above date shows, we can see when we pick N to be 398, B to be 8, P to be 1 and C to be 3, the data we have for using Inter-Thread Communication is definitely shorter than using the Inter-Process Communication. Average timing measurement for Inter-Thread Communication is 0.002002 seconds and it is shorter than the average timing for Inter-Process Communication which is 0.003064 seconds. The Standard deviation for Inter-Thread Communication is 0.000121 and it is smaller than the Standard deviation for Inter-Process Communication which is 0.000842.

From the timing Analysis, it shows one of the advantage of using Inter-Thread Communication instead of using the Inter-Process Communication and it is the big reason why choose threads rather than creating a new process because of the superior performance. Creating a new thread is much faster than creating a new process and as the prof mentioned in the lecture, creating a new thread is in fact 10 times faster than creating a new process. Terminating and cleaning up a thread is also faster than terminating and cleaning up a process. Also as the prof mentioned in lectures, it takes less time to switch between two threads within the same process (because less data needs to be stored/restored), the reason of that is because threads share the same memory space. If two threads are communicating, they do not have to use any of the IPC mechanisms, because they can just communicate directly. If we analyze the time with respect to the standard deviation, we can see the Inter-Thread

Communication is more stable then the Inter-Process Communication, because the standard deviation for threads is smaller than processes. A reason of that might be threads have the advantage of background working. Threads can run something in the background to keep the program responsive which is better than process since process is not responsive when some parts of the program is blocked.

To conclude this analysis, using Inter-Thread Communication to solve the lab given problem (square root a produced number) is better than using Inter-Process Communication. It is because Inter-Thread Communication has a better performance than Inter-Process Communication since it is faster and more stable.

Appendix A:

Source code of the producer and consumer using Inter-Thread Communication

```
#include <stdbool.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <pthread.h>
#include <semaphore.h>
#include <signal.h>
#include <math.h>

// global variable
sem_t spaces;
sem_t items;
int BUFFER_SIZE;
int INT_NEED_PRODUCED;
```

```
int NUM_OF_CONSUMERS;
int NUM_OF_PRODUCERS;
int *buffer;
int produced_counter;
int consumed_counter;
int buffer_index;
pthread_mutex_t mutex;
struct timeval tv;
double t1;
double t2;

//producer
void *producer( void* arg){
     // get the pid
     int pid = *(int*)arg;
     // while condition meets, produce
     while(INT_NEED_PRODUCED > produced_counter && pid < INT_NEED_PRODUCED)
{
            int product;
            product = produced_counter;
            // decide which producer produce which
            if(product % NUM_OF_PRODUCERS == pid){
                // semaphore for spaces
                sem_wait( &spaces );
                // critical section since the buffer can't be used in
parallel
                pthread_mutex_lock(&mutex);
                buffer[buffer_index] = product;
                buffer_index ++;
                // counter for the number of product
                produced_counter ++;
                pthread_mutex_unlock(&mutex);
                sem_post( &items );
            }
     }
     pthread_exit( NULL );
}

//comsumer
void* consumer( void* arg ) {
     // get the consumer id
     int cid = *(int*)arg;
     // products in buffer
     int product_int;
```

```
      // square root of the product
      int result;
      while(buffer_index >= 0) {
         // semaphore for items
         sem_wait( &items );
         // critical section since the buffer can't be used in parallel
         pthread_mutex_lock(&mutex);
         buffer_index --;
         // this will prevent the deadlock, where the produce finished all
the produce and there are more there 1 items the consumers need to consume
         if(buffer_index < 0){
             pthread_mutex_unlock(&mutex);
             sem_post( &items );
             pthread_exit( NULL );
         }
         product_int = buffer[buffer_index];
         // counter for consumed items
         consumed_counter ++;
         // this will prevent the deadlock, where the produce finished all
the produce and there are more there 1 items the consumers need to consume
         if(consumed_counter > INT_NEED_PRODUCED - 1){
             sem_post( &items );
         }
         pthread_mutex_unlock(&mutex);
         sem_post( &spaces );

         result = sqrt(product_int);
         // check if it is a perfect root, if it is, then print
         if(product_int == (result * result)){
             printf("%d %d %d\n", cid, product_int, result);
         }
      }
      pthread_exit( NULL );
}
//main
int main(int argc,  char const *argv[])
{
      //initial the counters
      buffer_index = 0;
      produced_counter = 0;
      consumed_counter = 0;

      //N the number of integers the producers should produce in total 1
      //B be the buffer size 2
```

```
//P be the number of producers 3
//C be the number of consumers 4
// check conditions
if (argc != 5){
   printf("Input Error!/n");
   return -1;
}

// get B
BUFFER_SIZE = atoi(argv[2]);
if(BUFFER_SIZE < 0){
   printf("Buffer Size Error!/n");
   return -1;
}
// get P and C
NUM_OF_PRODUCERS = atoi(argv[3]);
NUM_OF_CONSUMERS = atoi(argv[4]);

// check conditions
if(NUM_OF_PRODUCERS < 1 ){
   printf("Number of Producer Error!/n");
   return -1;
}

if(NUM_OF_CONSUMERS < 1 ){
   printf("Number of Consumers Error!/n");
   return -1;
}

if(NUM_OF_CONSUMERS > 1 && NUM_OF_PRODUCERS > 1){
   printf("Bad Case, Producer and Consumer Both Bigger Than 1!/n");
   return -1;
}

// get N
INT_NEED_PRODUCED = atoi(argv[1]);
if(INT_NEED_PRODUCED < 0){
   printf("Number of Integer Need to Produce Error!/n");
   return -1;
}

// creates the buffer and PC id arrays
buffer = (int *) malloc(sizeof(int) *BUFFER_SIZE);
int *pid_array = (int *) malloc(sizeof(int) *NUM_OF_PRODUCERS);
```

```
int *cid_array = (int *) malloc(sizeof(int) *NUM_OF_CONSUMERS);
int i;
for ( i = 0; i < BUFFER_SIZE; i++ ) {
   buffer[i] = -1;
}
for ( i = 0; i < NUM_OF_PRODUCERS; i++ ) {
   pid_array[i] = i;
}
for ( i = 0; i < NUM_OF_CONSUMERS; i++ ) {
   cid_array[i] = i;
}

//initiallize mutex
pthread_mutex_init(&mutex, NULL);
pthread_mutex_init(&mutex, NULL);
//initiallize sem
sem_init( &spaces, 0, BUFFER_SIZE );
sem_init( &items, 0, 0 );
//array for threads
pthread_t producer_array[NUM_OF_PRODUCERS];
pthread_t consumer_array[NUM_OF_CONSUMERS];

//starting point
gettimeofday(&tv, NULL);
t1 = tv.tv_sec + tv.tv_usec/1000000.0;

//PC
//create threads
int a;
for(a = 0; a < NUM_OF_PRODUCERS; a++){
   pthread_create(&producer_array[a], NULL, producer, &pid_array[a]);
}
for(a = 0; a < NUM_OF_CONSUMERS; a++){
   pthread_create(&consumer_array[a], NULL, consumer, &cid_array[a]);
}

//Join
for(a = 0; a < NUM_OF_PRODUCERS; a++){
   pthread_join(producer_array[a], NULL);
}
for(a = 0; a < NUM_OF_CONSUMERS; a++){
   pthread_join(consumer_array[a], NULL);
}
```

```
    //ending point
    gettimeofday(&tv, NULL);
    t2 = tv.tv_sec + tv.tv_usec/1000000.0;
    printf("System execution time: %.6lf\n", t2-t1);

    // freeing
    buffer = NULL;
    free( buffer );
    free( pid_array );
    free( cid_array );
    sem_destroy( &spaces );
    sem_destroy( &items );
    pthread_mutex_destroy( &mutex );

    return 0;
}
```

Appendix B:

Source code of the producer and consumer using Inter-Process Communication

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <unistd.h>
#include <mqueue.h>

struct timeval tv;
double t1;
double t2;

mqd_t producer;
mqd_t consumer;
mqd_t publicQueue;

int Number, Bsize, Pnum, Cnum;
int empty = -1;
```

```
int Produce(int id)
{
  int currentMsgNum, spaceAvalibal, msg, index;
  msg = 1;
  index = 0;
  while(1)
  {
     //printf("P %d - wait \n", id);
    mq_receive(producer, (char *)&spaceAvalibal, sizeof(int), 0);
     //printf("P %d - get \n", id);
    mq_receive(publicQueue, (char *)&currentMsgNum, sizeof(int), 0);
     //current count
    if (currentMsgNum > 0) {
         msg = id + index*Pnum;
         //printf("%d \n", msg);
         index++;              // create some msg
         currentMsgNum--;       // global counter --
         mq_send(publicQueue, (char *)&currentMsgNum, sizeof(int), 0);
         mq_send(consumer, (char *)&msg, sizeof(int), 0);
    }
      else if (currentMsgNum <= 0){
         mq_send(producer, (char *)&spaceAvalibal, sizeof(int), 0);  // Let
the other producers get chance to stop
         mq_send(consumer, (char *)&empty, sizeof(int), 0);
         mq_send(publicQueue, (char *)&currentMsgNum, sizeof(int), 0);
         break;
      }

  }
     //printf(" finished producer %d \n", id);
  return 0;
}

int Consume(int id)
{
  int msg, spaceAvalibal, output;
  spaceAvalibal = 1;
  while (1) {
       //printf("C %d - wait \n", id);
    mq_receive(consumer, (char *)&msg, sizeof(int), 0);
     //printf("C %d - get \n", id);
    if (msg != empty) {
```

```
        output = sqrt(msg);
        if(output * output == msg){
            printf("%d %d %d\n", id, msg, output);
        }
        mq_send(producer, (char *)&spaceAvalibal, sizeof(int), 0);

    } else if(msg == empty){
        mq_send(consumer, (char *)&msg, sizeof(int), 0);
        break;
      }

  }
     //printf(" finished consumer %d \n", id);
  return 0;
}

int main(int argc, char *argv[])
{
     //printf("start\n");
    //  A valid posix queue name must start with a "/".

    // int Number, Bsize, Pnum, Cnum;
    int P_pid, C_pid, Pid;              // producer Pid and consumer Pid, and
a public used Pid
    int currentMsgNum;             // receive the number of how many msg
left to be produce in Global Counter
    int i;

    char *pq_name = "/promail1";
      char *cq_name = "/conmail1";
      char *gq_name = "/pmail1";

      // unlink all the queues to make sure they are cleaned.
    mq_unlink("/promail1");
    mq_unlink("/conmail1");
    mq_unlink("/pmail1");


    if (argc != 5 || atoi(argv[1]) < 0 || atoi(argv[2]) < 0 || atoi(argv[3])
< 0 || atoi(argv[4]) < 0) {
        perror("Invalid arguments \n");
        return 0;
    }
```

```
    Number = atoi(argv[1]);  //number of messages the producers should
produce
    Bsize = atoi(argv[2]);   //buff_er size
    Pnum = atoi(argv[3]);    //number of producers
    Cnum = atoi(argv[4]);   // number of consumers


    struct mq_attr attr; // queue attr for Producer and Receiver
    attr.mq_maxmsg  = Bsize;
    attr.mq_msgsize = sizeof(int);
    attr.mq_flags   = 0; /* a blocking queue  */

      mode_t mode = S_IRUSR | S_IWUSR;

   // printf("initial the three queues \n");
    producer = mq_open(pq_name, O_RDWR | O_CREAT, mode, &attr);
    if (producer == -1 ) {
         perror("mq_open1() failed");
         return 0;
      }
    consumer = mq_open(cq_name, O_RDWR | O_CREAT, mode, &attr);
    if (consumer == -1 ) {
         perror("mq_open2() failed");
         return 0;
      }


      struct mq_attr attr2; // queue attr for Global Queue
    attr2.mq_maxmsg  = 1;
    attr2.mq_msgsize = sizeof(int);
    attr2.mq_flags   = 0;    /* a blocking queue  */


    publicQueue = mq_open(gq_name, O_RDWR | O_CREAT, mode, &attr2);
    if (publicQueue == -1 ) {
         perror("mq_open3() failed");
         return 0;
      }

    //start time
      //printf("Timer start \n");
    gettimeofday(&tv, NULL);
       t1 = tv.tv_sec + tv.tv_usec/1000000.0;
```

```
//printf("initial the buffer sized box \n");
// set Buffer sized msg box
for (i = 0; i < Bsize; i++) {
  if(( mq_send(producer, (char *)&i, sizeof(int), 0)) == -1){
     perror("mq_send() for producer failed");
  }

}

//printf("initial the global counter \n");
// set total number of msgs - global counter
if(( mq_send(publicQueue, (char *)&Number, sizeof(int), 0)) == -1 ){
  perror("mq_send() for publicQueue failed");
}




  //printf("start fork produce processes\n");
for(P_pid = 0; P_pid < Pnum; P_pid++)
{
  Pid = fork();
  if (Pid == 0) {
    Produce(P_pid);
     return 0;
  } else if (Pid < 0) {
    printf("Fail to fork produce process %d \n", P_pid );
     return 0;
  }
}

  //printf("start fork consumer processes\n");
for(C_pid = 0; C_pid < Cnum; C_pid++)
{
  Pid = fork();
  if (Pid == 0) {
    Consume(C_pid);
     return 0;
  } else if (Pid < 0) {
    printf("Fail to fork consume process %d \n", P_pid );
     return 0;
  }
}
```

```
while (1) {
  //Polling: keep tracking the number of msg left in the global Counter
  mq_receive(publicQueue, (char *)&currentMsgNum, sizeof(int), 0); //get
  if (currentMsgNum > 0) {
    mq_send(publicQueue, (char *)&currentMsgNum, sizeof(int), 0); //set
    //continue...
  } else if (currentMsgNum <= 0) {
      mq_send(publicQueue, (char *)&currentMsgNum, sizeof(int), 0);
    break;
    // finished
  }
}



  if (mq_close(producer) == -1){
     perror("mq_close() producer failed");
     return 0;
  }
  if (mq_close(consumer) == -1){
     perror("mq_close() consumer failed");
     return 0;
  }
  if (mq_close(publicQueue) == -1){
     perror("mq_close() publicQueue failed");
     return 0;
  }

  if (mq_unlink(pq_name) != 0){
     perror("mq_unlink() producer failed");
     return 0;
  }

  if (mq_unlink(cq_name) != 0){
     perror("mq_unlink() consumer failed");
     return 0;
  }

  if (mq_unlink(gq_name) != 0){
     perror("mq_unlink() publicQueue failed");
     return 0;
  }



//end time
```

```
  gettimeofday(&tv, NULL);
      t2 = tv.tv_sec + tv.tv_usec/1000000.0;

  double runtime = t2 - t1;
  printf("System execution time: %.6lf seconds \n", runtime);
    // printf("%.6lf \n", runtime);
  return 0;
}
```