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;
}
```