University of Waterloo
Faculty of Engineering
Department of Electrical and Computer Engineering

Lab 3 Report
(Group 68)

Prepared by
Zhidong Zhang
20619543
z498zhan@uwaterloo.ca
2B Computer Engineering
and
Shuyu Lyu
20622527
s88lu@edu.uwaterloo.ca
2B Computer Engineering

23 November 2017

Timing analysis:

- Inter-thread communication with shared memory:

N	B	P	C	Time
100	4	1	1	0.000851
100	4	1	2	0.000810
100	4	1	3	0.000765
100	4	2	1	0.000811
100	4	3	1	0.000875
100	8	1	1	0.000618
100	8	1	2	0.000768
100	8	1	3	0.000884
100	8	2	1	0.000742
100	8	3	1	0.000848
398	8	1	1	0.001157
398	8	1	2	0.001531
398	8	1	3	0.002034
398	8	2	1	0.001436
398	8	3	1	0.001640

- Inter-process communication with message queue:

N	B	P	C	Time
100	4	1	1	0.002457
100	4	1	2	0.002921
100	4	1	3	0.003117
100	4	2	1	0.002837
100	4	3	1	0.003001
100	8	1	1	0.002627
100	8	1	2	0.002866
100	8	1	3	0.003081
100	8	2	1	0.002585
100	8	3	1	0.002997
398	8	1	1	0.003643
398	8	1	2	0.003577
398	8	1	3	0.002983
398	8	2	1	0.003408
398	8	3	1	0.004146

- (N, B, P, C) = (398, 8, 1, 3) ran 500 times
    1. Thread:
       average timing = 0.002034s, standard deviation = 0.000100s
    2. Process:
       average timing = 0.002983s, standard deviation = 0.001662s

Analysis:

From the above tables, it is obvious that multi-thread approach has a lower average time than multi-process approach.

By running 500 times with (N, B, P, C) = (398, 8, 1, 3), we can also confirm that using thread is faster than using process (the average time of using thread is less than the average time of using process by 0.000949s). Besides, multi-thread approach has a much lower standard deviation than multi-process approach (standard deviation of multi-thread is 0.000100s and standard deviation of multi-thread is 0.001662s), which also means the inter-thread communication approach performs more consistently than the inter-process communication approach.

The advantage of multi-thread approach is that using thread is faster and has more consistent performance. Creating new threads and cleaning up threads are much faster than creating new process and cleaning up processes. Besides, it takes less time to switch between two threads within the same process. Additionally, thread can communicate directly through shared memory, however, process can only communicate by using IPC communication, which is slower than using shared memory. The disadvantage of multi-thread approach is that using threads required more careful synchronization. Since threads share the same memory, using semaphore and mutex to synchronize the access to the shared data becomes more important, which adds more complexity to the solution. However, processes are independent and they don't have shared memory, therefore, synchronization is not needed as much.

To conclude, multi-thread approach are faster and more consistent than multi-process approach, while multi-process approach requires less synchronization and has less complexity.

Appendix A:

Source code of 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>

int N,B,P,C;
int total_consumed;
int *sharedbuffer;
int bufferCount;
//Declare all the semaphore
sem_t spaces, items, lock;

void *producer(void *arg) {
	int producerId;
	int count = 0;

	producerId = *(int *)arg;
	free(arg);

	while(1) {
		//Produce item
		int item = producerId + count * P;
		if(item >= N) {
			break;
		}
		//Wait for spaces
		sem_wait(&spaces);
		//Lock the modification
		sem_wait(&lock);
		//Add item
		sharedbuffer[bufferCount] = item;
		bufferCount++;
		count++;
		//signal lock
		sem_post(&lock);
		//signal items
		sem_post(&items);
	}
```

```
        pthread_exit(0);
}

void *consumer(void *arg) {
        int consumerId;
        int item;
        int itemSqrt;

        consumerId = *(int *)arg;
        free(arg);

        while(total_consumed < N) {
                sem_wait(&items);
                sem_wait(&lock);
                //pop
                if(bufferCount <= 0) {
                        sem_post(&lock);
                        sem_post(&items);
                        break;
                }
                bufferCount--;
                item = sharedbuffer[bufferCount];
                total_consumed++;
                sem_post(&lock);
                sem_post(&spaces);
                //Signal other consumer
                if (total_consumed >= N){
                        sem_post(&items);
                }
                //Consume item
                itemSqrt = sqrt(item);
                if(itemSqrt * itemSqrt == item) {
                        printf("%d %d %d \n", consumerId, item, itemSqrt);
                }
        }

        pthread_exit(0);
}

int main(int argc, const char * argv[]){
        if (argc < 5) {
                printf("Incorrect number of arguments.\n");
                exit(1);
        }
        N = atoi(argv[1]);
        B = atoi(argv[2]);
        P = atoi(argv[3]);
        C = atoi(argv[4]);

        total_consumed = 0;

        //Init shared buffer using array
        sharedbuffer = (int *) malloc(sizeof(int) *B);
```

```
	//Init semaphore
	sem_init(&spaces, 0, B);
	sem_init(&items, 0, 0);
	sem_init(&lock, 0, 1);

	pthread_t *producer_t = malloc(sizeof(pthread_t) * P);
	pthread_t *consumer_t = malloc(sizeof(pthread_t) * C);

	struct timeval tv;
	double t1;
	double t2;

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

	int i;
	for (i = 0; i < P; i++) {
		int *id = malloc(sizeof(int));
		*id = i;
		pthread_create(&producer_t[i], NULL, producer, id);
	}

	for (i = 0; i < C; i++) {
		int *id = malloc(sizeof(int));
		*id = i;
		pthread_create(&consumer_t[i], NULL, consumer, id);
	}

	for (i = 0; i < P; i++) {
		pthread_join(producer_t[i], NULL);
	}

	for (i = 0; i < C; i++) {
		pthread_join(consumer_t[i], NULL);
	}

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

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

	//Clean up
	free(producer_t);
	free(consumer_t);
	free(sharedbuffer);
	sem_destroy(&items);
	sem_destroy(&spaces);
	sem_destroy(&lock);

	return 0;
}
```

Appendix B:

Source code of inter-process communication:

1. Source code of producer:

```
#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>

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

	int itemNum, producerNum, consumerNum, bufferSize, producerId;

	itemNum = atoi(argv[1]);
	producerId = atoi(argv[2]);
	producerNum = atoi(argv[3]);
	consumerNum = atoi(argv[4]);

	mqd_t mq;
	char  *name = "/MessageQueue_z498zhan";
	//Open the message queue
	mq  = mq_open(name, O_RDWR);
	if (mq == -1 ) {
		perror("mq failed to open");
		exit(1);
	}

	int count = 0;

	while(1) {
		//Produce item
		int item = producerId + count * producerNum;
		if(item >= itemNum) {
			break;
		}
		//send item
		if(mq_send(mq, (char*)&item, sizeof(int), 0) == -1){
			perror("Sending item to mq failed\n");
		}
```

```
                count++;
        }

        if (mq_close(mq) == -1){
                perror("mq failed to close");
                exit(2);
        }

        return 0;
}
```

2. Source code of consumer

```
#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>

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

        int itemNum, producerNum, consumerNum, bufferSize, consumerId;

        itemNum = atoi(argv[1]);
        consumerId = atoi(argv[2]);
        producerNum = atoi(argv[3]);
        consumerNum = atoi(argv[4]);

        mqd_t mq;
        char *name = "/MessageQueue_z498zhan";

        //Open the message queue
        mq = mq_open(name, O_RDONLY);
        if (mq == -1 ) {
                perror("mq open failed.\n");
                exit(1);
        }

        mqd_t cq;
        char *cname = "/CountQueue_z498zhan";

        //Open the count queue for counting the recieved items
        cq = mq_open(cname, O_RDWR);
        if (cq == -1 ) {
                perror("cq open failed.\n");
                exit(1);
```

```
	}

	int item;
	int itemSqrt;
	int received_count;
	while(true) {
		//Recieve count from cq
		if(mq_receive(cq, (char *)&received_count, sizeof(int), 0) == -1){
			perror("Failed receiving item from cq\n");
			return 1;
		}

		//if all item are recieved, signal other process and break
		if(received_count == itemNum){
			if(mq_send(cq, (char *)&received_count, sizeof(int), 0) == -1){
				perror("Failed sending item to cq\n");
			}
			break;
		}

		//increment received_count
		if (received_count < itemNum){
			received_count++;
			if(mq_send(cq, (char *)&received_count, sizeof(int), 0) == -1){
				perror("Failed sending item to cq\n");
			}
		}

		//Recieve item from message queue
		if(mq_receive(mq, (char *)&item, sizeof(int), 0) == -1){
			perror("Failed receiving item from mq\n");
			return 1;
		}

		//Consume item
		itemSqrt = sqrt(item);
		if(itemSqrt * itemSqrt == item) {
			printf("%d %d %d \n", consumerId, item, itemSqrt);
		}

	}

	if (mq_close(mq) == -1) {
		perror("mq failed to close");
		exit(2);
	}

	if (mq_close(cq) == -1) {
		perror("cq failed to close");
		exit(2);
	}

	return 0;
}
```

3. Source code of the main function that create processes

```c
#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>

int N,B,P,C;

int newProcess(char* program, char** arguments) {
	pid_t pid;
	pid = fork();

	if(pid < 0) {
		fprintf (stderr, "an error occurred in fork\n");
		abort ();
	}
	else if (pid == 0){
		//Child process
		//Replace the current child program with program
		execvp (program, arguments);
		//The execvp function returns only if an error occurs.
		fprintf (stderr, "an error occurred in execvp\n");
		abort();
	}
	else {
		//Parent process
		return pid;
	}
}



int main(int argc, char * argv[]) {
	if (argc < 5) {
		printf("Incorrect number of arguments. \n");
		exit(1);
	}

	N = atoi(argv[1]);
	B = atoi(argv[2]);
	P = atoi(argv[3]);
```

```
C = atoi(argv[4]);

//Initialize message queue
mqd_t mq;
char *name = "/MessageQueue_z498zhan";
mode_t mode = S_IRUSR | S_IWUSR;
struct mq_attr attr;

attr.mq_maxmsg  = B;
attr.mq_msgsize = sizeof( int );
attr.mq_flags   = 0;

mq_unlink(name);

mq  = mq_open(name, O_RDWR | O_CREAT, mode, &attr);
if (mq == -1 ) {
        perror("mq open failed");
        exit(1);
}

//Initialize count queue
mqd_t cq;
char * cname = "/CountQueue_z498zhan";
mode_t mode_c = S_IRUSR | S_IWUSR;
struct mq_attr c_attr;

c_attr.mq_maxmsg = 1;
c_attr.mq_msgsize = sizeof(int);
c_attr.mq_flags = 0;


mq_unlink(cname);

cq = mq_open(cname, O_RDWR | O_CREAT, mode_c, &c_attr);
if (cq == -1){
        perror("cq failed to open");
        exit(1);
}

int received_count = 0;
if(mq_send(cq, (char*)&received_count, sizeof(int), 0) == -1){
        perror("cq failed to send");
}

struct timeval tv;
double t1;
double t2;

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

int i, j;
for(i = 0; i < P; i++) {
        int length = snprintf( NULL, 0, "%d", i );
```

```
                char id_str[length+1];
                sprintf(id_str, "%d", i);
                argv[2] = id_str;
                newProcess("./producer", argv);
        }

        for(j = 0; j < C; j++) {
                int length = snprintf( NULL, 0, "%d", j );
                char id_str[length+1];
                sprintf(id_str, "%d", j);
                argv[2] = id_str;
                newProcess("./consumer", argv);
        }

        int status = 0;
        //wait until all child processes return
        while (wait(&status) > 0);

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

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

        if(mq_close(mq) == -1){
                perror("mq failed to close");
                exit(2);
        }

        if(mq_unlink(name) != 0) {
                perror("mq failed to unlink");
                exit(3);
        }

         if(mq_close(cq) == -1){
                perror("cq failed to close");
                exit(2);
         }

         if(mq_unlink(cname) != 0){
                perror("cq failed to unlink");
                exit(3);
         }

        return 0;
}
```