ECE 254: Operating Systems and Systems Programming

Lab 5 Report

Group 01
By: Rushan Yogaratnam & Ameen Patel
2B Computer Engineering
University of Waterloo
Spring 2014

Introduction:

In this lab two solutions are developed for the producer consumer problem. One implementation involves the use of multiple processes with a POSIX message queue while the other implementation uses multiple threads and a shared global data structure. Having implemented the producer consumer problem under the two implementation schemes a timing analysis was conditioned to better analyze the performance characteristics of both implementations and the tradeoffs involved with each approach.

Important Timing Considerations:

In this particular implementation a linked list is used as a buffer. This works by having the producer appending to the head of the linked list and the consumers consuming from the head of the linked list. Since both the insert and consume operations involve the head they are O (1) operations. In addition a semaphore is used in the producer and consumer to enforce that the linked list maintains a maximum size of B, size of buffer.

In terms of timing the results may be slightly higher than expected as the linked list uses heap allocated memory by making calls to malloc(). However the results still show the trend that the threading results are faster than the process results. Note that the heap allocated memory is de-allocated by the consumer using free().

Note that all timing data was collected over 400 runs, X = 400

Process Implementation:

Average System Execution Time				
N	**B**	**P**	**C**	**Time (seconds)**
100	4	1	1	0.002053
100	4	1	2	0.002345
100	4	1	3	0.002837
100	4	2	1	0.002337
100	4	3	1	0.002627
100	8	1	1	0.001824
100	8	1	2	0.00249
100	8	1	3	0.00278
100	8	2	1	0.002333
100	8	3	1	0.002972
398	8	1	1	0.002109
398	8	1	2	0.002577
398	8	1	3	0.002959
398	8	2	1	0.002499
398	8	3	1	0.003122

Threading Implementation:

Average System Execution Time				
N	**B**	**P**	**C**	**Time (seconds)**
100	4	1	1	0.000451
100	4	1	2	0.00048
100	4	1	3	0.00066
100	4	2	1	0.000561
100	4	3	1	0.000688
100	8	1	1	0.000475
100	8	1	2	0.000693
100	8	1	3	0.000742
100	8	2	1	0.000517
100	8	3	1	0.000635
398	8	1	1	0.000753
398	8	1	2	0.000746
398	8	1	3	0.000854
398	8	2	1	0.000873
398	8	3	1	0.000968

Timing Data For (N ,B ,P ,C) = (398, 8 , 1, 3) , X = 400		
Implementation	**Average System Execution Time (s)**	**Standard Deviation (s)**
Multi-Process	0.002959	0.000726
Threaded	0.000854	0.000561

Notice that the average system execution time of the threaded implementation is 3.46 times faster than the multi-process implementation.

Discussion:

From the timing analysis it is clear that the multi-threaded implementation significantly outperformed the multi-process implementation in terms of speed. When looking at the average system initialization time at (N,B,P,C) = (398,8,1,3) we see that the multi-threaded implementation is 3.46 times faster than the multi-process implementation. From looking at the table of average system execution times it can be seen that implementation using threads is consistently faster in every case. This can be attributed to the fact that threads are much lighter than processes thus allowing the OS to switch between threads much faster than it can switch between processes. Another contributing factor is that on ecelinux the max POSIX message queue size is 10. This significantly slows down the process implementation as the buffer fills up and becomes empty rather fast.

Advantages and Disadvantages:

One reason to favor using multiple threads instead of multiple processes is because inter process communication between thread is easier between threads than it in between processes. We did not have to run into this issue as we were fortunate enough to use the POSIX message queue, which provided the sufficient mechanism to enable IPC, without it IPC would be very complicated between processes. Another reason to favor threads over processes is because context switching in threading is much faster than context-switching in processes. The OS is able to switch threads much faster than it is able to switch between processes. Threads are also take less time to initialize than a process which requires in this case requires a call to fork and exec. The timing analysis further support this argument as the Producer Consumer threading implementation was significantly faster than the process implementation.

The problem with threads is that they operate in the same virtual address space and therefore are using the same global data-structure. This causes non-determinism as the concurrent threads are accessing the same shared mutable variables. In order to co-ordinate their execution and prevent race conditions and deadlocks we must use synchronization primitives such as mutex's and semaphores. In that sense the threaded implementation is much more difficult to debug and maintain as we must always be concerned with race conditions and deadlock. For the process implementation with the POSIX message queue the blocking and unblocking in done for us and so there is less work to be done as we don't have to worry about deadlocks and race conditions. One issue with the POSIX message queue is that the max number of messages that can fit into the queue is set by the system. On ecelinux the max queue size is 10, this is a limiting factor of the POSIX message queue. Overall the threading approach leads to faster switching between threads and is faster overall at the cost of race conditions and deadlocks, while the process implementation is less susceptible to race conditions in deadlock but comes at the cost of being slow to switch between processes.

Source Code Listing:

.c source files:

```c
/*
 * common.c
 *
 * ECE254 Group 01
 * By : Rushan Yogaratnam and Ameen Patel
 * University of Waterloo Computer Engineering
 * Spring 2014
 *
 * Implementation of common.h
 *
 */

#include <mqueue.h>
#include <stdlib.h>

//constant queue_name for both producer and consumer.
const char* queue_name = "/mailbox_ece254_ryogarat";

//name is unique to avoid conflicting with other students.
const char* consumer_sem_name = "named_sem_ryogarat_cons_sem";



//implemenation of process_arguments, used to ensure
//the command line arguments are valid.
int process_arguments(int argc, char* argv[], int * queue_size,
        int * message_count, int * producer_count, int * consumer_count) {

    if (argc < 5) {
        return 1;
    } else {
        *message_count = atoi(argv[1]);
        *queue_size = atoi(argv[2]);
        *producer_count = atoi(argv[3]);
        *consumer_count = atoi(argv[4]);

        return ((*message_count <= 0 || *queue_size <= 0 || *producer_count
<= 0
                || *consumer_count <= 0) ? 1 : 0);
    }

}
```

```c
/*
 * producerConsumerParent.c
 * ECE254 Group 01
 * By : Rushan Yogaratnam and Ameen Patel
 * University of Waterloo Computer Engineering
 * Spring 2014
 *
 * Producer Consumer Process Implementation:
 * This is the parent process which fork's and
 * exec's the producer and consumer processes
 * as child processes.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <sys/time.h>
#include <time.h>
#include <semaphore.h>
#include <string.h>
#include "producer.h"
#include "common.h"

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

    int queue_size;
    int message_count;
    int num_producers;
    int num_consumers;

    //validate the command line arguments
    if (process_arguments(argc, argv, &queue_size, &message_count,
            &num_producers, &num_consumers))
    {
        printf("Invalid  arguments provided\n");
        return 1;
    }

    //set queue attributes
    struct mq_attr queue_attributes;
    queue_attributes.mq_maxmsg = queue_size;
    queue_attributes.mq_msgsize = sizeof(int);
    queue_attributes.mq_flags = 0;

    mqd_t queue_descriptor;
    mode_t permissions = S_IRUSR | S_IWUSR;

    //attempt to open queue and perform error checking
    queue_descriptor = mq_open(queue_name, O_RDWR | O_CREAT, permissions,
            &queue_attributes);
```

```
    if (queue_descriptor == -1) {
        printf("error creating queue %s\n", strerror(errno));
        return 1;
    }

    //used by the consumers to determine wether or not
    //they should continue consumption
    sem_t *consumer_sem = sem_open(consumer_sem_name, O_RDWR | O_CREAT,
            permissions, message_count);

    if (consumer_sem == SEM_FAILED) {
        printf("failed to create consumer semaphore\n");
    }

    //get time before first fork
    double time_before_first_fork = get_time_in_seconds();

    //spawn producer processes
    int i;
    for (i = 0; i < num_producers; ++i) {

        spawn_child("./producer", argv, i, num_producers);
    }

    //spawn consumer processes
    int j;
    for (j = 0; j < num_consumers; ++j) {

        spawn_child("./consumer", argv, j, num_consumers);
    }

    int status, pid;

    //busy loop and wait for all of the child
    //processes to complete execution.

    while ((pid = wait(&status)) != -1) {
    }

    double time_after_last_consumed = get_time_in_seconds();

    double execution_time = time_after_last_consumed -
time_before_first_fork;
    printf("System execution time: %f seconds\n", execution_time);

    //Tidy up queues and semaphores

    //close queue.
    if (mq_close(queue_descriptor) == -1) {
        perror("mq_close failed");
        exit(2);
    }

    //mark queue for deletion.
    if (mq_unlink(queue_name) != 0) {
        perror("mq_unlink failed");
        exit(3);
```

```c
    }

    if (sem_close(consumer_sem) == -1) {
        perror("consumption semaphore failed to close");
        exit(2);
    }

    if (sem_unlink(consumer_sem_name) == -1) {
        perror("failed to unlink consumer semaphore");
        exit(3);
    }

    return 0;

}

int spawn_child(char* program, char **arg_list, int p_id, int childCount) {

    arg_list[0] = program;

    //as part of the arguments to the exec'd process
    //send the assigned ID, p_id for producers c_id for consumers.
    char pid[15];
    sprintf(pid, "%d", p_id);
    arg_list[2] = pid;

    pid_t child_pid;
    child_pid = fork();

    if (child_pid > 0) {
        return child_pid;
    } else if (child_pid < 0) {
        printf("error creating child process %s\n", strerror(errno));
        return -1;
    } else {
        execvp(program, arg_list);
        printf("error occurred in execvp %s\n", strerror(errno));
        abort();
    }

}

double get_time_in_seconds() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (tv.tv_sec + tv.tv_usec / 1000000.0);

}
```

```c
/*
 * producer.c
 * ECE254 Group 01
 * By : Rushan Yogaratnam and Ameen Patel
 * University of Waterloo Computer Engineering
 * Spring 2014
 *
 * Producer Consumer Process Implementation:
 * This is the producer child process that
 * the parent process forks.
 *
 */
#include <mqueue.h>
#include <stdio.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include <semaphore.h>
#include <string.h>
#include "producer.h"
#include "common.h"

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

    //this is the assigned producer id, between 0 and P-1.
    int pid = atoi(argv[2]);

    //the number of producer, P
    int producer_count = atoi(argv[3]);

    //number of messages to produce
    int production_count = atoi(argv[1]);

    //attempt to open the queue and perform error handling
    mqd_t queue_descriptor;
    queue_descriptor = mq_open(queue_name, O_RDWR);

    if (queue_descriptor == -1) {
        printf("error opening queue in producer %s\n", strerror(errno));
        return 1;
    }

    //produce only the set of elements that satisfy i%num_producers = pid.
    int i;
    for (i = pid; i < production_count; i += producer_count) {

        //send a message to the queue, blocks if queue is full.
        int message = i;
        if (mq_send(queue_descriptor, (char*) &message, sizeof(int), 0) == -
1) {
            printf("P: pid %d send failed %s \n", getpid(), strerror(errno));
            return 1;
        }
    }
```

```c
    //close the queue
    if (mq_close(queue_descriptor) == -1) {
        perror("mq_close failed");
        exit(2);
    }

    return 0;

}
```

```c
/*
 * consumer.c
 * ECE254 Group 01
 * By : Rushan Yogaratnam and Ameen Patel
 * University of Waterloo Computer Engineering
 * Spring 2014
 *
 *Producer Consumer process implemenation:
 * This is the consumer process that the
 * parent process creates.
 *
 */
#include <mqueue.h>
#include <stdio.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include <semaphore.h>
#include <string.h>
#include <math.h>
#include "common.h"

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

    //the assigned consumer id, between 0 and C-1.
    int c_id = atoi(argv[2]);

    //open the queue and perform error handling

    mqd_t queue_descriptor;
    queue_descriptor = mq_open(queue_name, O_RDONLY);

    if (queue_descriptor == -1) {
        printf("error opening queue in consumer %s\n", strerror(errno));
        return 1;
    }

    //open a descriptor for the consumer semaphore.
    sem_t *consumer_sem;
    consumer_sem = sem_open(consumer_sem_name, 0);
```

```
    if (consumer_sem == SEM_FAILED) {
        printf("error opening semaphore in consumer %s\n", strerror(errno));
        return 1;
    }

    //infinite loop and keep consuming elements.
    //the consumer exits when consumer_sem
    //indicates that there are no more items to be expected.
    while (1) {

        //decrement the consumer semaphore.
        //when this reaches 0 all callers will
        //stop consuming.

        if (sem_trywait(consumer_sem) == -1) {
            break;
        }

        //recieve a message, this will block if the queue is empty.
        int message;
        if (mq_receive(queue_descriptor, (char*) &message, sizeof(int), 0)==
-1) {

            printf("failed to receive message %s \n", strerror(errno));
            return 1;
        } else {
            int root = sqrt(message);

            if ((root * root) == message) {
                printf("%i %i %i\n", c_id, message, root);
            }
        }

    }

    //close the descriptor to the queue.
    if (mq_close(queue_descriptor) == -1) {
        perror("mq_close failed");
        exit(2);
    }

    //close the descriptor to the semaphore
    if (sem_close(consumer_sem) == -1) {
        perror("sem_close failed in consumer");
        exit(2);
    }

    return 0;

}
```

```c
/*
 * producerConsumer.c
 * ECE254 Group 01
 * By :  Rushan Yogaratnam and Ameen Patel
 * University of Waterloo Computer Engineering
 * Spring 2014
 *
 *Producer Consumer Threading Implementation:
 * This process has the main thread create
 * new threads for the producers and consumers
 * and passes the messages to a global data-structure.
 *
 * Data is passed to a linked list , but items
 * are only added to the head and are removed
 * from the head. This makes the run-time of the linked
 * list operation O(1).
 *
 *A counting semaphore is used to ensure the buffer
 *size is fixed, hence the linked list does not overflow.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <time.h>
#include <stdlib.h>
#include <math.h>
#include "producerConsumer.h"
#include "common.h"

//List node for the queue
struct queue_element {
    int value;
    struct queue_element* next;
};

//struct for the thread parameters
//this is used to pass the producer
//and consume id's to threads.
struct thread_params {
    int id;
};

//global buffer
struct queue_element* buffer;

//semaphore for the critical sections.
sem_t buff_lock;
//semaphore for the number of messages produces.
sem_t count;
//semaphore for the size of the buffer.
sem_t buff_size;
//semaphore for the consumer
sem_t con_num;
```

```
//these are declared global to allow
//the producers to read them.
int production_count;
int producer_count;

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

    int consumer_count;
    int buffer_size;

    if (process_arguments(argc, argv, &buffer_size, &production_count,
            &producer_count, &consumer_count)) {
        printf("Invalid arguments\n");
        return 1;
    }

    //initailze the buffer to null.
    //the buffer is dynamically built up
    //using the add and remove operations.
    //the size is enforced by a semaphore.
    buffer = NULL;

    //initalize semaphores
    sem_init(&buff_lock, 0, 1);
    sem_init(&count, 0, 0);
    sem_init(&buff_size, 0, buffer_size);
    sem_init(&con_num, 0, production_count);

    //array of id's used for joining
    pthread_t p_thread_id[producer_count];
    pthread_t c_thread_id[consumer_count];

    //creates structs to pass to threads
    struct thread_params p_id[producer_count];
    struct thread_params c_id[consumer_count];

    //get time before first fork
    double time_before_first_thread_created = get_time_in_seconds();

    //creates producer threads
    int i;
    for (i = 0; i < producer_count; ++i) {
        p_id[i].id = i;
        pthread_create(&(p_thread_id[i]), NULL, &producer, &(p_id[i]));
    }

    //create consumer threads
    int c;
    for (c = 0; c < consumer_count; ++c) {
        c_id[c].id = c;
        pthread_create(&(c_thread_id[c]), NULL, &consumer, &(c_id[c]));
    }

    int j;
    for (j = 0; j < producer_count; ++j) {
        pthread_join(p_thread_id[j], NULL);
    }
```

```
    int k;
    for (k = 0; k < consumer_count; ++k) {
        pthread_join(c_thread_id[k], NULL);
    }

    double time_after_last_consumed = get_time_in_seconds();
    double execution_time = time_after_last_consumed
            - time_before_first_thread_created;

    printf("System execution time: %f seconds\n", execution_time);

    //clean up semaphores
    sem_destroy(&buff_lock);
    sem_destroy(&count);
    sem_destroy(&buff_size);
    sem_destroy(&con_num);

    return 0;
}

/*
 Function is used to add an element to the linked list.
 If the linkedlist is null it creates
 the linked list and sets the value of the head.
 All further calls appends the values to the head
 of the linked list.
 */
void add_to_buffer(int value) {

    if (buffer == NULL) {
        buffer = malloc(sizeof(struct queue_element));
        buffer->value = value;
        buffer->next = NULL;
    } else {
        struct queue_element* new_head = malloc(sizeof(struct
queue_element));
        new_head->next = buffer;
        new_head->value = value;
        buffer = new_head;
    }
}

/*
 Function takes an item from the head of the linked list
 reassigned the head and deletes the node from the heap.
 This throws an error if the linked list is null.
 If the message obtained is a square number it is printed out.
 */
void consume_from_buffer(int * c_id) {

    if (buffer == NULL) {
        printf("error failed to read from queue, queue is empty\n");
    } else {

        //get the current head of the buffer
        struct queue_element* current_element;
```

```
        current_element = buffer;

        int val = current_element->value;

        //reassign the buffer head
        buffer = buffer->next;
        //delete the previous buffer node.
        free(current_element);

        int root = sqrt(val);
        if (val == (root * root)) {
            printf("%i %i %i\n", *c_id, val, root);
        }
    }

}

/*
 Producer thread function.
 producer producers the set of integers
 that satisfy i%num_producers = p_id;

 Produer first waits till the buffer is not full.
 Producer locks the buffer.
 Producer inserts a value into a buffer.
 Producer unlocks the buffer.
 Producer notifies the buffer is not empty, by calling
 sem_post(&count).
 */
void* producer(void* producer_params) {

    struct thread_params *params = (struct thread_params*) producer_params;
    int p_id = params->id;

    int i;
    for (i = p_id; i < production_count; i += producer_count) {

        //wait until buffer is not full
        sem_wait(&buff_size);

        //toggle the lock
        sem_wait(&buff_lock);

        //add item to buffer
        add_to_buffer(i);

        //release the lock
        sem_post(&buff_lock);

        //notify that the buffer is not empty.
        sem_post(&count);

    }
    return NULL;
}

/*
```

```
 Consumer thread function.
 Consumer consumes all integers until none are avaliable.
 If the buffer if empty it blocks.

 Consumer waits until buffer is not empty.
 Consumer locks the buffer.
 Consumer retrieves item from the buffer
 Consumer unlocks the buffer.
 Consumer notifies that the buffer is not full,by calling
 sem_post(&buff_size)

 */
void* consumer(void* consumer_params) {

    struct thread_params *params = (struct thread_params*) consumer_params;
    int c_id = params->id;

    while (1) {

        //If no more items are to be consumed
        //exit the thread function
        if (sem_trywait(&con_num)) {
            break;
        }

        //wait until the buffer is not empty.
        sem_wait(&count);

        //lock the buffer
        sem_wait(&buff_lock);

        //consume an item from the buffer
        consume_from_buffer(&c_id);

        //unlock the buffer
        sem_post(&buff_lock);

        //notify that the buffer is not full.
        sem_post(&buff_size);
    }
    return NULL;
}

double get_time_in_seconds() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (tv.tv_sec + tv.tv_usec / 1000000.0);

}
```

Header Files:

```c
/*
 * common.h
 *
 * ECE254 Group 01
 * By : Rushan Yogaratnam and Ameen Patel
 * University of Waterloo Computer Engineering
 * Spring 2014
 *
 * The purpose of this header file is to
 * declare common constants and functions
 * that are used between both the producer and the consumer.
 */

#ifndef COMMON_H_
#define COMMON_H_

//global queue_name
extern const char* queue_name;

//global semaphore queue_name
extern const char* consumer_sem_name;

//Provider error checking on the command line arguments,
// if they are invalid -1 is return. The last two parameters
//are pointers to queue size and process count, these are set
//if valid

int process_arguments(int argc, char* argv[], int * queue_size,
        int * message_count, int * producer_count, int * consumer_count);

//This method provides a convenient way to get the current time in seconds
double get_time_in_seconds();

#endif
```

```c
/*
```

```
 * Producer.h
 * ECE254 Group 01
 * By : Rushan Yogaratnam and Ameen Patel
 * University of Waterloo Computer Engineering
 * Spring 2014
 *
 *
 * These are a bunch of helper functions.
 * The are just functions which
 * make the code cleaner by providing abstraction.
 *
 */

#ifndef PRODUCER_H_
#define PRODUCER_H_


//spawns the child process and sets the time before forking,
//which is the last parameter.
int spawn_child(char*, char **, int, int);

#endif /* PRODUCER_H_ */

/*
 * producerConsumer.h
 * ECE254 Group 01
 * By : Rushan Yogaratnam and Ameen Patel
 * University of Waterloo Computer Engineering
 * Spring 2014
 *
 *
 * These are a bunch of helper functions.
 * The are just functions which
 * make the code cleaner by providing abstraction.
 *
 */
#ifndef PRODUCERCONSUMER_H_
#define PRODUCERCONSUMER_H_

//Thread function for the producer threads
void* producer(void* unused);

//Thread functionf for the consumer threads
void* consumer(void* unused);

#endif /* PRODUCERCONSUMER_H_ */
```