System Programming Lecture - Report 7

Student Number: 201520740

Name: LIN WEI

Exercise 1

Program

main.c

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <pthread.h>

#define ADD_ENTRY_NUM 30
#define DELETE_ENTRY_NUM 10
#define ENQUEUE_THREAD_NUM 2
#define DEQUEUE_THREAD_NUM 6

/* Global mutex for structured code locking. */
pthread_mutex_t list_lock = PTHREAD_MUTEX_INITIALIZER;

struct entry {
	struct entry *next;
	void *data;
};
struct entry *entry;

struct list {
	struct entry *head;
	struct entry **tail;
};
```

```
struct list *list;

struct list *
list_init(void)
{
	struct list *list;

	list = malloc(sizeof *list);
	if (list == NULL)
		return (NULL);
	list->head = NULL;
	list->tail = &list->head;
	return (list);
}

int
list_enqueue(struct list *list, void *data)
{
	struct entry *e;
	char *d = (char *) data;

	e = malloc(sizeof *e);
	if (e == NULL)
		return (1);
	e->next = NULL;
	e->data = data;

	pthread_mutex_lock(&list_lock);
	*list->tail = e;
	list->tail = &e->next;
	printf("Enqueued [%s]\n", d);
	pthread_mutex_unlock(&list_lock);
```

```
        return (0);
}

struct entry *
list_dequeue(struct list *list)
{
        struct entry *e;
        char * d;

        pthread_mutex_lock(&list_lock);
        if (list->head == NULL)
        {
           printf("Dequeue failed: empty list\n");
                pthread_mutex_unlock(&list_lock);
           return(NULL);
        }
        e = list->head;
        list->head = e->next;
        if (list->head == NULL)
              list->tail = &list->head;
        d = (char *)e->data;
        printf("Dequeued [%s]\n", d);
        pthread_mutex_unlock(&list_lock);

        return (e);
}

struct entry *
list_traverse(struct list *list, int (*func)(void *, void *), void *user)
{
        struct entry **prev, *n, *next;
```

```
    pthread_mutex_lock(&list_lock);

if (list == NULL)
    {
            pthread_mutex_unlock(&list_lock);
            return (NULL);
    }


prev = &list->head;
for (n = list->head; n != NULL; n = next) {
      next = n->next;
      switch (func(n->data, user)) {
      case 0:
            /* continues */
            prev = &n->next;
            break;
      case 1:
            /* delete the entry */
            *prev = next;
            if (next == NULL)
                  list->tail = prev;
                        pthread_mutex_unlock(&list_lock);
            return (n);
      case -1:
      default:
            /* traversal stops */
                        pthread_mutex_unlock(&list_lock);
            return (NULL);
      }
}
```

```
            pthread_mutex_unlock(&list_lock);
        return (NULL);
}

int
print_entry(void *e, void *u)
{
        printf("%s\n", (char *)e);
        return (0);
}

int
delete_entry(void *e, void *u)
{
        char *c1 = e, *c2 = u;

        return (!strcmp(c1, c2));
}

/* Thread of enqueue. */
void * enqueue(void *arg)
{
        int i;
        char buffer[100];
        for(i = 0; i < ADD_ENTRY_NUM; i++)
        {
                sprintf(buffer, "%d", i);
                list_enqueue(list, strdup(buffer));
        }
}
```

```
/* Thread of dequeue. */
void * dequeue(void *arg)
{
	int i;
	for(i = 0; i < DELETE_ENTRY_NUM; i++)
	{
		list_dequeue(list);
	}
}

int
main()
{
	pthread_t enq[ENQUEUE_THREAD_NUM], deq[DEQUEUE_THREAD_NUM];
	int i;

	list = list_init();

	    /* Create enqueue and dequeue threads. */
	    for (i = 0; i < ENQUEUE_THREAD_NUM || i < DEQUEUE_THREAD_NUM; i++)
	    {
			if(i < ENQUEUE_THREAD_NUM)
				pthread_create(&enq[i], NULL, enqueue, NULL);
			if(i < DEQUEUE_THREAD_NUM)
				pthread_create(&deq[i], NULL, dequeue, NULL);
	    }

	for(i = 0; i < ENQUEUE_THREAD_NUM; i++)
		pthread_join(enq[i], NULL);

	for(i = 0; i < DEQUEUE_THREAD_NUM; i++)
		pthread_join(deq[i], NULL);
```

```
        /* Delete the entry with data "2". */
        entry = list_traverse(list, delete_entry, "2");
    if (entry != NULL) {
          free(entry->data);
          free(entry);
    }

        /* Traverse the list. */
        list_traverse(list, print_entry, NULL);
    free(list);
    return (0);
}
```

Result

```
lw@lw-VirtualBox:~/Documents/Report7/Exercise1$ ./a.out
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
```

```
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
```

Dequeue failed: empty list

Enqueued [0]

Enqueued [1]

Enqueued [2]

Enqueued [3]

Enqueued [4]

Enqueued [5]

Enqueued [6]

Enqueued [7]

Enqueued [8]

Enqueued [9]

Enqueued [10]

Enqueued [11]

Enqueued [12]

Enqueued [13]

Enqueued [14]

Enqueued [15]

Enqueued [16]

Enqueued [17]

Enqueued [18]

Enqueued [19]

Enqueued [20]

Enqueued [21]

Enqueued [22]

Enqueued [23]

Enqueued [24]

Enqueued [25]

Enqueued [26]

Enqueued [27]

Enqueued [28]

Enqueued [29]

Dequeued [0]

Dequeued [1]

Dequeued [2]

Dequeued [3]

Dequeued [4]

Dequeued [5]

Dequeued [6]

Dequeued [7]

Dequeued [8]

Dequeued [9]

Enqueued [0]

Enqueued [1]

Enqueued [2]

Enqueued [3]

Enqueued [4]

Enqueued [5]

Enqueued [6]

Enqueued [7]

Enqueued [8]

Enqueued [9]

Enqueued [10]

Enqueued [11]

Enqueued [12]

Enqueued [13]

Enqueued [14]

Enqueued [15]

Enqueued [16]

Enqueued [17]

Enqueued [18]

Enqueued [19]

Enqueued [20]

Enqueued [21]

Enqueued [22]

Enqueued [23]

Enqueued [24]

Enqueued [25]

Enqueued [26]

Enqueued [27]

Enqueued [28]

Enqueued [29]

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

0

1

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

Consideration on Exercise 1

As we can see, all these threads ran correctly. First, the dequeue threads tried to delete entries but failed because the list was empty. After the enqueue threads have added some entries into the list, the remaining dequeue threads began to delete entries from the list. Finally, other enqueue threads added more entries into the list. And after the program deleted the entry with data”2”, *list_traverse* function showed there was no data “2” among the remaining entries.

Exercise 2

Program

main.c

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <pthread.h>

#define ADD_ENTRY_NUM 30
#define DELETE_ENTRY_NUM 10
#define ENQUEUE_THREAD_NUM 2
#define DEQUEUE_THREAD_NUM 6

struct entry {
        struct entry *next;
        void *data;
};
struct entry *entry;

struct list {
        struct entry *head;
        struct entry **tail;
            /* Mutex for structured data locking. */
            pthread_mutex_t list_lock;
};
struct list *list;

struct list *
list_init(void)
{
```

```
        struct list *list;

        list = malloc(sizeof *list);
        if (list == NULL)
                return (NULL);
        list->head = NULL;
        list->tail = &list->head;
            /* Init of the mutex. */
            pthread_mutex_init(&list->list_lock, NULL);
        return (list);
}

int
list_enqueue(struct list *list, void *data)
{
        struct entry *e;
        char *d = (char *) data;

        e = malloc(sizeof *e);
        if (e == NULL)
                return (1);
        e->next = NULL;
        e->data = data;

        pthread_mutex_lock(&list->list_lock);
        *list->tail = e;
        list->tail = &e->next;
        printf("Enqueued [%s]\n", d);
        pthread_mutex_unlock(&list->list_lock);

        return (0);
```

```
}

struct entry *
list_dequeue(struct list *list)
{
        struct entry *e;
        char * d;

        pthread_mutex_lock(&list->list_lock);
        if (list->head == NULL)
        {
           printf("Dequeue failed: empty list\n");
                pthread_mutex_unlock(&list->list_lock);
           return(NULL);
        }
        e = list->head;
        list->head = e->next;
        if (list->head == NULL)
                list->tail = &list->head;
        d = (char *)e->data;
        printf("Dequeued [%s]\n", d);
        pthread_mutex_unlock(&list->list_lock);

        return (e);
}

struct entry *
list_traverse(struct list *list, int (*func)(void *, void *), void *user)
{
        struct entry **prev, *n, *next;
```

```
    pthread_mutex_lock(&list->list_lock);

if (list == NULL)
    {
        pthread_mutex_unlock(&list->list_lock);
        return (NULL);
    }


prev = &list->head;
for (n = list->head; n != NULL; n = next) {
    next = n->next;
    switch (func(n->data, user)) {
    case 0:
        /* continues */
        prev = &n->next;
        break;
    case 1:
        /* delete the entry */
        *prev = next;
        if (next == NULL)
            list->tail = prev;
                pthread_mutex_unlock(&list->list_lock);
        return (n);
    case -1:
    default:
        /* traversal stops */
                pthread_mutex_unlock(&list->list_lock);
        return (NULL);
    }
}
```

```
            pthread_mutex_unlock(&list->list_lock);
        return (NULL);
}

int
print_entry(void *e, void *u)
{
        printf("%s\n", (char *)e);
        return (0);
}

int
delete_entry(void *e, void *u)
{
        char *c1 = e, *c2 = u;

        return (!strcmp(c1, c2));
}

/* Thread of enqueue. */
void * enqueue(void *arg)
{
        int i;
        char buffer[100];
        for(i = 0; i < ADD_ENTRY_NUM; i++)
        {
                sprintf(buffer, "%d", i);
                list_enqueue(list, strdup(buffer));
        }
}
```

```
/* Thread of dequeue. */
void * dequeue(void *arg)
{
	int i;
	for(i = 0; i < DELETE_ENTRY_NUM; i++)
	{
		list_dequeue(list);
	}
}

int
main()
{
	pthread_t enq[ENQUEUE_THREAD_NUM], deq[DEQUEUE_THREAD_NUM];
	int i;

	list = list_init();

	    /* Create enqueue and dequeue threads. */
	    for (i = 0; i < ENQUEUE_THREAD_NUM || i < DEQUEUE_THREAD_NUM; i++)
	    {
			if(i < ENQUEUE_THREAD_NUM)
				pthread_create(&enq[i], NULL, enqueue, NULL);
			if(i < DEQUEUE_THREAD_NUM)
				pthread_create(&deq[i], NULL, dequeue, NULL);
	    }

	for(i = 0; i < ENQUEUE_THREAD_NUM; i++)
		pthread_join(enq[i], NULL);
```

```
    for(i = 0; i < DEQUEUE_THREAD_NUM; i++)
        pthread_join(deq[i], NULL);

        /* Delete the entry with data "2". */
        entry = list_traverse(list, delete_entry, "2");
    if (entry != NULL) {
        free(entry->data);
        free(entry);
    }

        /* Traverse the list. */
        list_traverse(list, print_entry, NULL);

    free(list);
    return (0);
}
```

Result

```
lw@lw-VirtualBox:~/Documents/Report7/Exercise2$ ./a.out
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
```

Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list
Dequeue failed: empty list

Dequeue failed: empty list

Dequeue failed: empty list

Dequeue failed: empty list

Dequeue failed: empty list

Dequeue failed: empty list

Dequeue failed: empty list

Dequeue failed: empty list

Enqueued [0]

Enqueued [1]

Enqueued [2]

Enqueued [3]

Enqueued [4]

Enqueued [5]

Enqueued [6]

Enqueued [7]

Enqueued [8]

Enqueued [9]

Enqueued [10]

Enqueued [11]

Enqueued [12]

Enqueued [13]

Enqueued [14]

Enqueued [15]

Enqueued [16]

Enqueued [17]

Enqueued [18]

Enqueued [19]

Enqueued [20]

Enqueued [21]

Enqueued [22]

Enqueued [23]

Enqueued [24]

Enqueued [25]

Enqueued [26]

Enqueued [27]

Enqueued [28]

Enqueued [29]

Dequeued [0]

Dequeued [1]

Dequeued [2]

Dequeued [3]

Dequeued [4]

Dequeued [5]

Dequeued [6]

Dequeued [7]

Dequeued [8]

Dequeued [9]

Enqueued [0]

Enqueued [1]

Enqueued [2]

Enqueued [3]

Enqueued [4]

Enqueued [5]

Enqueued [6]

Enqueued [7]

Enqueued [8]

Enqueued [9]

Enqueued [10]

Enqueued [11]

Enqueued [12]

Enqueued [13]

Enqueued [14]

Enqueued [15]

Enqueued [16]

Enqueued [17]

Enqueued [18]

Enqueued [19]

Enqueued [20]

Enqueued [21]

Enqueued [22]

Enqueued [23]

Enqueued [24]

Enqueued [25]

Enqueued [26]

Enqueued [27]

Enqueued [28]

Enqueued [29]

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

0

1

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

Consideration on Exercise 2

All these threads ran correctly and the result was the same as exercise 1.By these two exercises, I have gotten the difference between code locking and data locking in programming.

Review of this lecture

In this lecture, I have learned the basic of code locking, data locking and the deadlock problem. I find I always tend to forget unlocking the mutex after the *return* statement within some *if* blocks. We really should take care of this kind of miss because it may cause the deadlock problem when all the threads are sharing one Mutex lock.