

Project 5: Designing a Thread Pool and Producer-Consumer Problem

CS307-Operating System (D), CS356-Operating System Course Design, Chentao Wu, Spring 2020.

Name: 方泓杰(Hongjie Fang) Student ID: 518030910150 Email: galaxies@sjtu.edu.cn

1 Designing a Thread Pool

Thread pools were introduced in Section 4.5.1 in textbook. When thread pools are used, a task is submitted to the pool and executed by a thread from the pool. Work is submitted to the pool using a queue, and an available thread removes work from the queue. If there are no available threads, the work remains queued until one becomes available. If there is no work, threads await notification until a task becomes available. This project involves creating and managing a thread pool, and it may be completed using either Pthreads and POSIX synchronization or Java.

Solution. I use Pthread and POSIX synchronization in C programming language to complete the project. Here are my detailed solutions.

- I implement a linked queue to store the waiting work, so the length of the queue is basically unlimited and we can allow more work submitted in the same time.
- We use a mutex lock `queue_mutex` in every queue operation to prevent racing conditions in the queue.
- We use a semaphore `wait_sem` to represent the number of waiting work in the waiting queue.
- Every time we submit a work, we will first put it in the waiting queue. And then we use the semaphore `wait_sem` to awake a idle thread to execute it. If the threads are all busy, then the work will be waiting in the queue.
- For every thread (worker), we use the semaphore to let it wait for an available task. If it get an available task, it will update the queue and execute the task. The routine of every thread is basically an infinite loop repeating checking available work and executing it.
- When shutting down the thread pool, I use a flag `shutdown_flag` to notify every thread, and we will unlock the semaphore to let every thread exit from the infinite loop. And we use `pthread_join` to gather all threads together in the end.

Here is the specific implementation of the thread pool (`threadpool.c`).

```
1 #include <pthread.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <semaphore.h>
6 #include "threadpool.h"
7
8 #define QUEUE_SIZE 20
9 #define NUMBER_OF_THREADS 3
10
11 #define TRUE 1
12
```

```

13 // this represents work that has to be
14 // completed by a thread in the pool
15 typedef struct {
16     void (*function)(void *p);
17     void *data;
18 } task;
19
20 // the work queue
21 struct queue_node {
22     task worktodo;
23     struct queue_node *nxt;
24 } *head, *tail;
25
26 // the worker bee
27 pthread_t bee[NUMBER_OF_THREADS];
28
29 // the mutex lock
30 pthread_mutex_t queue_mutex;
31
32 // semaphore
33 sem_t wait_sem;
34
35 // shutdown flag
36 int shutdown_flag;
37
38 // Insert a task into the queue.
39 // returns 0 if successful or 1 otherwise.
40 int enqueue(task t) {
41     tail -> nxt = (struct queue_node *) malloc (sizeof(struct queue_node));
42
43     // fail to allocate space
44     if (tail -> nxt == NULL) return 1;
45
46     tail = tail -> nxt;
47     tail -> worktodo = t;
48
49     return 0;
50 }
51
52 // Remove a task from the queue.
53 task dequeue() {
54     if (head == tail) {
55         fprintf(stderr, "Error: No more work to do!\n");
56         exit(1);
57     }
58
59     static struct queue_node *tmp;
60     tmp = head;
61     head = head -> nxt;
62     free(tmp);

```

```

63
64     return head -> worktodo;
65 }
66
67 // the worker thread in the thread pool.
68 void *worker(void *param) {
69     static task tsk;
70     while(TRUE) {
71         // wait for available task
72         sem_wait(&wait_sem);
73
74         if (shutdown_flag) break;
75
76         pthread_mutex_lock(&queue_mutex);
77         tsk = dequeue();
78         pthread_mutex_unlock(&queue_mutex);
79
80         // execute the task
81         execute(tsk.function, tsk.data);
82     }
83     pthread_exit(0);
84 }
85
86 // Executes the task provided to the thread pool.
87 void execute(void (*somefunction)(void *p), void *p) {
88     (*somefunction)(p);
89 }
90
91 // Submits work to the pool.
92 int pool_submit(void (*somefunction)(void *p), void *p) {
93     static task tsk;
94     tsk.function = somefunction;
95     tsk.data = p;
96
97     pthread_mutex_lock(&queue_mutex);
98     int res = enqueue(tsk);
99     pthread_mutex_unlock(&queue_mutex);
100
101     // success, update the semaphore.
102     if (res == 0)
103         sem_post(&wait_sem);
104     return res;
105 }
106
107 // Initialize the thread pool.
108 void pool_init(void) {
109     static int err;
110
111     shutdown_flag = 0;
112

```

```

113 // initialize the queue
114 head = (struct queue_node *) malloc (sizeof(struct queue_node));
115 if (head == NULL) {
116     fprintf(stderr, "Error: queue initialization error!\n");
117     exit(1);
118 }
119 head -> nxt = NULL;
120 tail = head;
121
122 // create the mutex lock
123 err = pthread_mutex_init(&queue_mutex, NULL);
124 if (err) {
125     fprintf(stderr, "Error: pthread mutex initialization error!\n");
126     exit(1);
127 }
128
129 // create the semaphore
130 err = sem_init(&wait_sem, 0, 0);
131 if (err) {
132     fprintf(stderr, "Error: semaphore initialization error!\n");
133     exit(1);
134 }
135
136 // create the threads
137 for (int i = 0; i < NUMBER_OF_THREADS; ++ i) {
138     err = pthread_create(&bee[i], NULL, worker, NULL);
139     if (err) {
140         fprintf(stderr, "Error: pthread create error!\n");
141         exit(1);
142     }
143 }
144 fprintf(stdout, "[Log] Create the threads successfully!\n");
145 }
146
147 // shutdown the thread pool
148 void pool_shutdown(void) {
149     static int err;
150
151     shutdown_flag = 1;
152
153     // release the semaphore
154     for (int i = 0; i < NUMBER_OF_THREADS; ++ i)
155         sem_post(&wait_sem);
156
157     // join the threads
158     for (int i = 0; i < NUMBER_OF_THREADS; ++ i) {
159         err = pthread_join(bee[i], NULL);
160         if (err) {
161             fprintf(stderr, "Error: pthread join error!\n");
162             exit(1);

```

```

163     }
164 }
165 fprintf(stdout, "[Log] Join the threads successfully!\n");
166
167 // destroy the mutex lock
168 err = pthread_mutex_destroy(&queue_mutex);
169 if (err) {
170     fprintf(stderr, "Error: pthread mutex destroy error!\n");
171     exit(1);
172 }
173
174 // destroy the semaphore
175 err = sem_destroy(&wait_sem);
176 if (err) {
177     fprintf(stderr, "Error: semaphore destroy error!\n");
178     exit(1);
179 }
180 }

```

I write a `client.c` program to help testing the thread pool. In the testing program, we construct 50 pieces of work, and each of them is adding two random values together. We will send them to the thread pool and wait some time in order to let all the work finish executing. You can refer to the code [src/threadpool/client.c](#) to see the details.

I also write a Makefile file to help testing. We only need to enter the following instructions in the terminal and we can test the program.

```

1 make
2 ./example

```

Here are the executing result (Fig. 1 and Fig. 2). Here we only show the beginning logs and ending logs of the testing program, since the full version containing all executing logs of the 50 pieces of work is too long.

```

galaxies@ubuntu:~/CS307-Projects/Project5/threadpool$ make
gcc -Wall -c client.c -lpthread
gcc -Wall -c threadpool.c -lpthread
gcc -Wall -o example client.o threadpool.o -lpthread
galaxies@ubuntu:~/CS307-Projects/Project5/threadpool$ ./example
[Log] Create the threads successfully!
I add two values 83 and 86 result = 169
I add two values 77 and 15 result = 92
I add two values 93 and 35 result = 128
I add two values 86 and 92 result = 178
I add two values 49 and 21 result = 70
I add two values 62 and 27 result = 89
I add two values 90 and 59 result = 149
I add two values 63 and 26 result = 89
I add two values 40 and 26 result = 66
I add two values 72 and 36 result = 108
I add two values 11 and 68 result = 79
I add two values 67 and 29 result = 96
I add two values 82 and 30 result = 112
I add two values 62 and 23 result = 85
I add two values 67 and 35 result = 102
I add two values 29 and 2 result = 31
I add two values 22 and 58 result = 80
I add two values 69 and 67 result = 136
I add two values 93 and 56 result = 149

```

Figure 1: The beginning logs of the testing program

```

I add two values 24 and 95 result = 119
I add two values 82 and 45 result = 127
I add two values 14 and 67 result = 81
I add two values 34 and 64 result = 98
I add two values 43 and 50 result = 93
I add two values 87 and 8 result = 95
I add two values 76 and 78 result = 154
I add two values 88 and 84 result = 172
I add two values 3 and 51 result = 54
I add two values 54 and 99 result = 153
I add two values 32 and 60 result = 92
I add two values 76 and 68 result = 144
I add two values 39 and 12 result = 51
I add two values 26 and 86 result = 112
I add two values 94 and 39 result = 133
[Log] Join the threads successfully!
galaxies@ubuntu:~/CS307-Projects/Project5/threadpools$

```

Figure 2: The ending logs of the testing program

In summary, that's the full C implementation and the testing process of the thread pool. \square

2 Producer-Consumer Problem

In Section 7.1.1 in textbook, we presented a semaphore-based solution to the producer-consumer problem using a bounded buffer. In this project, you will design a programming solution to the bounded-buffer problem using the producer and consumer processes. The solution presented in Section 7.1.1 uses three semaphores, `empty` and `full`, which count the number of empty and full slots in the buffer, and `mutex`, which is a binary (or mutual-exclusion) semaphore that protects the actual insertion of removal of items in the buffer. For this project, you will use standard counting semaphores for `empty` and `full` and a mutex lock, rather than a binary semaphore, to represent `mutex`. The producer and consumer - running as separate threads - will move items to and from a buffer that is synchronized with the `empty`, `full` and `mutex` structures. You can solve the problem using either Pthreads or the Windows API.

Solution. I use Pthread in C programming language in Linux system to complete the project. Here are my detailed solutions.

- We use a semaphore `empty` to represent the number of empty blocks in the buffer. We use a semaphore `full` to represent the number of full blocks in the buffer. And we also use a mutex lock `mutex` to prevent racing conditions among producers and consumers.
- We let the producers to produce a piece of data in random time period (randomly from 1 second to 3 seconds); we also let the consumers to consume a piece of data in random time period (randomly from 1 second to 3 seconds).
- We use a circular queue taught in the data structure class to implement the buffer module and we use the bounded-buffer solution in textbook to implement the producers and consumers.
- The program will run for T seconds with n producers and m consumers, where T , n and m are provided as arguments. We will read them from the arguments to the program and create $(n + m)$ threads representing each producer and each consumer. After T seconds, we will terminate all the threads and gather them together and exit the program.

Here is the specific implementation of the buffer module (`buffer.h` and `buffer.c`).

```

1 # define BUFFER_SIZE 5
2 typedef int buffer_item;

```

```

3 int insert_item(buffer_item item);
4 int remove_item(buffer_item *item);
5 void buffer_initialization();

```

```

1 # include "buffer.h"
2
3 // buffer implementation: circular queue
4 buffer_item buf[BUFFER_SIZE + 1];
5 int head, tail;
6
7 // insert an item to the buffer
8 int insert_item(buffer_item item) {
9     if ((tail + 1) % (BUFFER_SIZE + 1) == head) return -1;
10    tail = (tail + 1) % (BUFFER_SIZE + 1);
11    buf[tail] = item;
12    return 0;
13 }
14
15 // remove an item from the buffer
16 int remove_item(buffer_item *item) {
17     if (head == tail) return -1;
18     head = (head + 1) % (BUFFER_SIZE + 1);
19     *item = buf[head];
20     return 0;
21 }
22
23 void buffer_initialization() {
24     head = 0;
25     tail = 0;
26 }

```

Here is the specific implementation of the producer-consumer program (`producer_consumer.c`).

```

1 # include <stdio.h>
2 # include <unistd.h>
3 # include <stdlib.h>
4 # include <pthread.h>
5 # include <semaphore.h>
6 # include "buffer.h"
7
8 # define TRUE 1
9
10 // semaphore empty, full
11 sem_t empty, full;
12 // mutex lock mutex
13 pthread_mutex_t mutex;
14 // terminate flag
15 int terminate_flag;
16
17 // producer thread
18 void *producer(void *param) {

```

```

19  buffer_item item;
20
21  while(TRUE) {
22      static int time_period;
23      time_period = rand() % 3;
24      sleep(time_period);
25
26      // generate an item
27      item = rand();
28
29      sem_wait(&empty);
30      pthread_mutex_lock(&mutex);
31      if (terminate_flag) break;
32
33      if (insert_item(item)) {
34          fprintf(stderr, "Error: producer can not insert the item!\n");
35          exit(1);
36      }
37      else fprintf(stdout, "[Log] Producer produced item %d.\n", item);
38
39      pthread_mutex_unlock(&mutex);
40      sem_post(&full);
41  }
42  pthread_mutex_unlock(&mutex);
43  pthread_exit(0);
44 }
45
46 // consumer thread
47 void *consumer(void *param) {
48     buffer_item item;
49
50     while(TRUE) {
51         static int time_period;
52         time_period = rand() % 3;
53         sleep(time_period);
54
55         sem_wait(&full);
56         pthread_mutex_lock(&mutex);
57         if (terminate_flag) break;
58
59         if (remove_item(&item)) {
60             fprintf(stderr, "Error: consumer can not remove the item!\n");
61             exit(1);
62         }
63         else fprintf(stdout, "[Log] Consumer consumed item %d.\n", item);
64
65         pthread_mutex_unlock(&mutex);
66         sem_post(&empty);
67     }
68     pthread_mutex_unlock(&mutex);

```



```

69     pthread_exit(0);
70 }
71
72 int main(int argc, char *argv[]) {
73     pthread_t *producer_t, *consumer_t;
74     int total_time, producer_number, consumer_number;
75     static int err;
76
77     // extract the arguments
78     if(argc != 4) {
79         fprintf(stderr, "Error: invalid arguments.\n");
80         exit(1);
81     }
82     total_time = atoi(argv[1]);
83     producer_number = atoi(argv[2]);
84     consumer_number = atoi(argv[3]);
85
86     // initialization
87     buffer_initialization();
88     terminate_flag = 0;
89     // |-- create the mutex lock
90     err = pthread_mutex_init(&mutex, NULL);
91     if (err) {
92         fprintf(stderr, "Error: pthread mutex initialization error!\n");
93         exit(1);
94     }
95     // |-- create the semaphore
96     err = sem_init(&empty, 0, BUFFER_SIZE);
97     if (err) {
98         fprintf(stderr, "Error: semaphore initialization error!\n");
99         exit(1);
100    }
101    err = sem_init(&full, 0, 0);
102    if(err) {
103        fprintf(stderr, "Error: semaphore initialization error!\n");
104        exit(1);
105    }
106
107    // create threads
108    producer_t = (pthread_t *) malloc (sizeof(pthread_t) * producer_number);
109    consumer_t = (pthread_t *) malloc (sizeof(pthread_t) * consumer_number);
110
111    for (int i = 0; i < producer_number; ++ i)
112        pthread_create(&producer_t[i], NULL, &producer, NULL);
113    for (int i = 0; i < consumer_number; ++ i)
114        pthread_create(&consumer_t[i], NULL, &consumer, NULL);
115
116    // sleep some time
117    sleep(total_time);
118

```

```

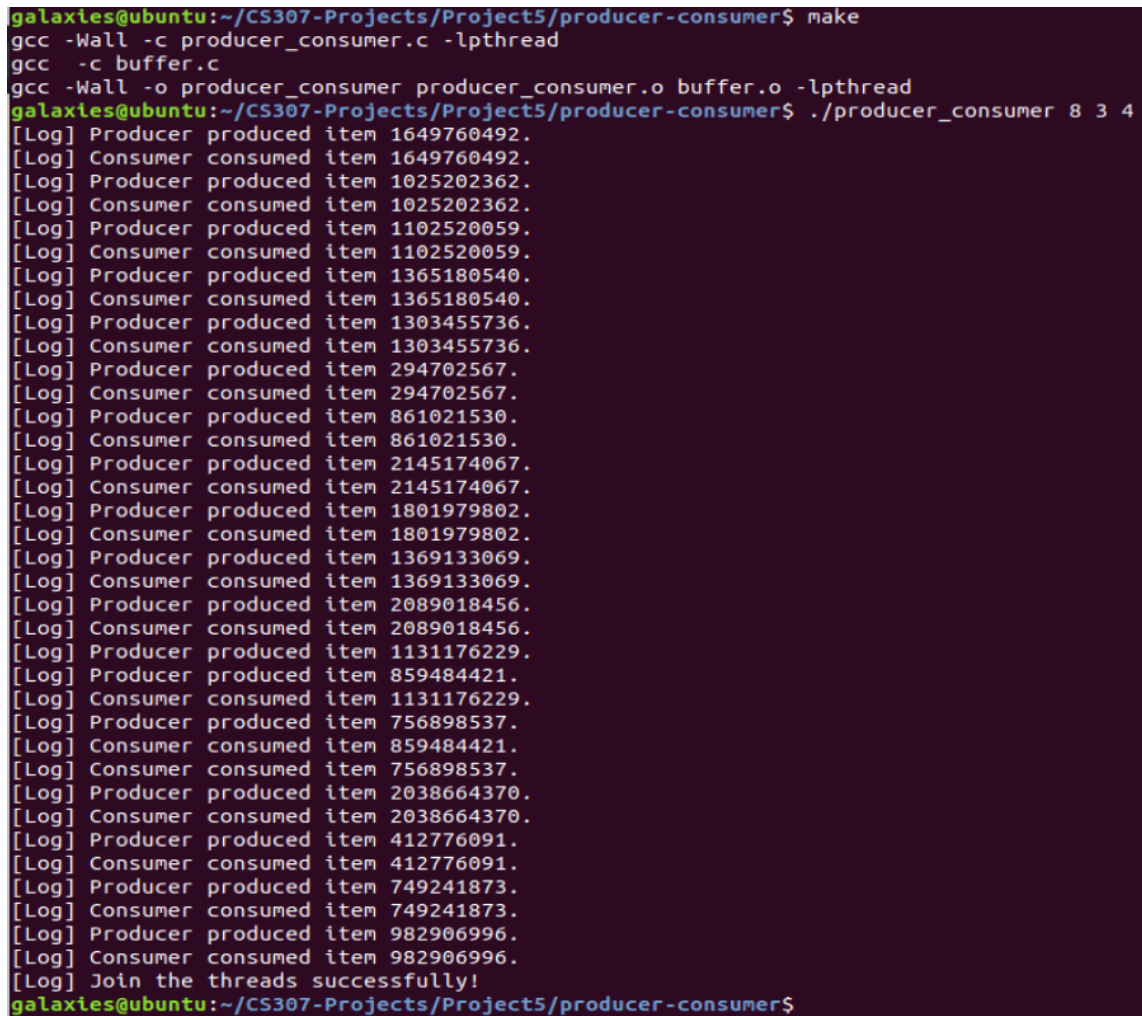
119 // terminate
120 terminate_flag = 1;
121 // |-- terminate the threads
122 for (int i = 0; i < producer_number; ++ i)
123     sem_post(&empty);
124 for (int i = 0; i < consumer_number; ++ i)
125     sem_post(&full);
126 // |-- join the threads
127 for (int i = 0; i < producer_number; ++ i) {
128     err = pthread_join(producer_t[i], NULL);
129     if (err) {
130         fprintf(stderr, "Error: pthread join error!\n");
131         exit(1);
132     }
133 }
134 for (int i = 0; i < consumer_number; ++ i) {
135     err = pthread_join(consumer_t[i], NULL);
136     if (err) {
137         fprintf(stderr, "Error: pthread join error!\n");
138         exit(1);
139     }
140 }
141 fprintf(stdout, "[Log] Join the threads successfully!\n");
142 // |-- destroy the mutex lock
143 err = pthread_mutex_destroy(&mutex);
144 if (err) {
145     fprintf(stderr, "Error: pthread mutex destroy error!\n");
146     exit(1);
147 }
148 // |-- destroy the semaphores
149 err = sem_destroy(&empty);
150 if (err) {
151     fprintf(stderr, "Error: semaphore destroy error!\n");
152     exit(1);
153 }
154 err = sem_destroy(&full);
155 if (err) {
156     fprintf(stderr, "Error: semaphore destroy error!\n");
157     exit(1);
158 }
159 // |-- free the spaces
160 free(producer_t);
161 free(consumer_t);
162
163 return 0;
164 }

```

I write a Makefile file to help testing the program. We only need to enter the following instructions in the terminal and we can test the program with arguments $T = 8$, $n = 3$ and $m = 4$, that is, 3 producers and 4 consumers in 8 seconds. You can also try your own arguments yourselves.

```
1 make
2 ./producer_consumer 8 3 4
```

Here is the execution result of the producer-consumer program (Fig. 3).



```
galaxies@ubuntu:~/CS307-Projects/Project5/producer-consumer$ make
gcc -Wall -c producer_consumer.c -lpthread
gcc -c buffer.c
gcc -Wall -o producer_consumer producer_consumer.o buffer.o -lpthread
galaxies@ubuntu:~/CS307-Projects/Project5/producer-consumer$ ./producer_consumer 8 3 4
[Log] Producer produced item 1649760492.
[Log] Consumer consumed item 1649760492.
[Log] Producer produced item 1025202362.
[Log] Consumer consumed item 1025202362.
[Log] Producer produced item 1102520059.
[Log] Consumer consumed item 1102520059.
[Log] Producer produced item 1365180540.
[Log] Consumer consumed item 1365180540.
[Log] Producer produced item 1303455736.
[Log] Consumer consumed item 1303455736.
[Log] Producer produced item 294702567.
[Log] Consumer consumed item 294702567.
[Log] Producer produced item 861021530.
[Log] Consumer consumed item 861021530.
[Log] Producer produced item 2145174067.
[Log] Consumer consumed item 2145174067.
[Log] Producer produced item 1801979802.
[Log] Consumer consumed item 1801979802.
[Log] Producer produced item 1369133069.
[Log] Consumer consumed item 1369133069.
[Log] Producer produced item 2089018456.
[Log] Consumer consumed item 2089018456.
[Log] Producer produced item 1131176229.
[Log] Producer produced item 859484421.
[Log] Consumer consumed item 1131176229.
[Log] Producer produced item 756898537.
[Log] Consumer consumed item 859484421.
[Log] Consumer consumed item 756898537.
[Log] Producer produced item 2038664370.
[Log] Consumer consumed item 2038664370.
[Log] Producer produced item 412776091.
[Log] Consumer consumed item 412776091.
[Log] Producer produced item 749241873.
[Log] Consumer consumed item 749241873.
[Log] Producer produced item 982906996.
[Log] Consumer consumed item 982906996.
[Log] Join the threads successfully!
galaxies@ubuntu:~/CS307-Projects/Project5/producer-consumer$
```

Figure 3: The execution result of the producer-consumer program

In summary, that's the full C implementation of the producer-consumer problem.

□

3 Personal Thoughts

The first project makes me understand the thread pool better, and the second one improves my understanding about some classical process synchronization methods. Both of them allow me to understand the basic tools such as Pthreads, mutex lock and semaphores in POSIX implementation better. I also implement the thread pool and the producer-consumer problem we learnt in the class myself, which benefits me a lot in programming field.

By the way, you can find all the source codes in the “src” folder. You can also refer to [my github](#) to see my codes of this project, and they are in the Project5 folder.