EI338 Computer Systems Engineering

Project 5

November 29, 2019

Zhihui Xie, 517030910356

The environment used in this project is **Deepin 15.11**, the latest version of an open source operating system based on Debian's stable branch. The kernel version is **Linux version 4.15.0**.

Exercise I Designing a Thread Pool

In this exercise, we will design a thread pool. 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.

We implement the thread pool in C. For thread management, we use the execution model **POSIX Threads (Pthreads)**.

Basic Structure

Put aside details in the client program, the implement of the thread pool involves the following activities:

- The *pool_init()* function will create the threads at startup as well as initialize mutual-exclusion locks and semaphores.
- The *pool_submit()* function is partially implemented and currently places the function to be executed—as well as its data— into a task struct.
- The *worker()* function is executed by each thread in the pool, where each thread will wait for available work.
- The *pool_shutdown()* function will cancel each worker thread and then wait for each thread to terminate by calling *pthread_join()* .

Besides, functions *enqueue()* and *dequeue()* are also needed to complete for managing the queue.

Implementation

enqueue() & dequeue()

The queue for the pool is realized as a circular queue. We use a mutex for synchronization.

```c
// insert a task into the queue
// returns 0 if successful or 1 otherwise,
int enqueue(task t) {
    pthread_mutex_lock(&mutex);

    if ((rear + 1) % QUEUE_SIZE == front) {
        pthread_mutex_unlock(&mutex);

        return 1;
    }

    worktodos[rear] = t;
    rear = (rear + 1) % QUEUE_SIZE;

    pthread_mutex_unlock(&mutex);

    return 0;
}

// remove a task from the queue
task *dequeue() {
    pthread_mutex_lock(&mutex);

    task *task_tmp;

    if (front == rear) {
        pthread_mutex_unlock(&mutex);

        return NULL;
    }

    task_tmp = &worktodos[front];
    front = (front + 1) % QUEUE_SIZE;

    pthread_mutex_unlock(&mutex);
    return task_tmp;
}
```

Listing 1: enqueue() & dequeue()

worker()

worker() is assigned to each thread and each thread will wait for available work. A semaphore is used to suggest whether there exists available work.

```
void *worker(void *param) {
    task *todo;

    // execute the task
    while(TRUE) {
        sem_wait(&sem);

        todo = dequeue();

        execute(todo->function, todo->data);
    }
}
```

Listing 2: worker()

pool_submit()

When a function as well as its data packed in a struct 'task' is submitted to the thread pool, the task will be enqueued and the semaphore' signal function will be called.

```
int pool_submit(void (*somefunction)(void *p), void *p) {
    task todo;
    int flag;

    todo.function = somefunction;
    todo.data = p;

    // add the task to the queue
    flag = enqueue(todo);

    if (flag)
        return 1;

    sem_post(&sem);
    return 0;
}
```

Listing 3: pool_submit()

pool_init()

First of all, we of course need to ensure that the pool is not empty by creating some threads. Mutex locks and semaphores are also initialized here for further usage.

```
void pool_init(void) {
    // create the threads at startup
    for (int i = 0; i < NUMBER_OF_THREADS; ++i) {
        pthread_create(&bees[i], NULL, worker, NULL);
    }

    // initialize the queue
    front = rear = 0;

    // initialize mutual-exclusion locks and semaphores
    sem_init(&sem, 0, 0);
    pthread_mutex_init(&mutex, NULL);
}
```

Listing 4: pool_init()

pool_cancel()

By calling *pool_cancel()*, we terminate threads so that everthing goes back to the main thread.

```
void pool_shutdown(void) {
    // cancel each worker thread and wait for each thread to terminate
    for (int i = 0; i < NUMBER_OF_THREADS; ++i) {
        pthread_cancel(bees[i]);
        pthread_join(bees[i], NULL);
    }
}
```

Listing 5: pool_cancel()

Result

To test whether our program is correct, we design the client as follows (*void add(void *param)* and *void subtract(void *param)* are simple functions doing easy calculation.):

```c
int main(void) {
    int sleep_time;

    // create some work to do
    struct data work;
    work.a = 5;
    work.b = 10;

    // initialize the thread pool
    pool_init();

    // submit the works to the queue
    for (int i = 0; i < 10; ++i) {
        pool_submit(&add, &work);
        pool_submit(&subtract, &work);
    }

    // may be helpful
    sleep_time = 1;
    printf("Sleep time: %d\n", sleep_time);
    sleep(sleep_time);

    pool_shutdown();

    return 0;
}
```

Listing 6: client.c

And experimental results go like this:

Figure 1: The Thread Pool Test

Exercise II The Producer – Consumer Problem

This time, we are going to design a programming solution to the bounded-buffer problem using the producer and consumer processes.

We use standard counting semaphores for **empty** and **full** and a mutex lock, rather than a binary semaphore, to represent mutex. The problem is solved using **Pthreads**.

Basic Structure

There are mainly two aspects to be consider: the buffer and the producer & consumer threads.

The Buffer Internally, the buffer will consist of a fixed-size array of type buffer item (which will be defined using a *typedef*). The array of buffer item objects will be manipulated as a circular queue. The buffer will be manipulated with two functions, *insert_item()* and *remove_item()*, which are called by the producer and consumer threads, respectively.

The Producer & Consumer Threads The producer thread will alternate between sleeping for a random period of time and inserting a random integer into the buffer. Random numbers will be produced using the *rand()* function, which produces random integers between 0 and *RAND_MAX*. The consumer will also sleep for a random period of time and, upon awakening, will attempt to remove an item from the buffer.

Implementation

The Buffer

The buffer is implemented as a circular queue. It will be initialized with *buffer_init()*, and then manipulated with *insert_item()* and *remove_item()*.

buffer_init() At the beginning, the circular queue is empty. The mutex lock and two semaphores are also initialized here.

```
void buffer_init() {
    front = rear = 0;

    // initialize mutual-exclusion locks and semaphores
    sem_init(&full, 0, 0);
    sem_init(&empty, 0, BUFFER_SIZE);

    pthread_mutex_init(&mutex, NULL);

    printf("buffer initialized\n");
}
```

Listing 7: buffer_init()

insert_item() & remove_item() Every time before we insert an item into the buffer, we need to synchronize both the mutex lock **mutex** and the semaphore **empty**. And after exiting the critical section, we call *pthread_mutex_unlock()* as well as *sem_post()*. *remove_item()* is very much the same.

```
int insert_item(buffer_item item) {
    sem_wait(&empty);
    pthread_mutex_lock(&mutex);

    if ((rear + 1) % BUFFER_SIZE == front) {
        pthread_mutex_unlock(&mutex);

        // return -1 indicating an error condition
        return -1;
    }

    buffer[rear] = item;
    rear = (rear + 1) % BUFFER_SIZE;

    pthread_mutex_unlock(&mutex);
    sem_post(&full);

    // return 0 if successful, otherwise
    return 0;
}

int remove_item(buffer_item *item) {
    sem_wait(&full);
    pthread_mutex_lock(&mutex);

    if (front == rear) {
        pthread_mutex_unlock(&mutex);

        // return -1 indicating an error condition
        return -1;
    }

    // place it in item
    *item = buffer[front];

    front = (front + 1) % BUFFER_SIZE;

    pthread_mutex_unlock(&mutex);
    sem_post(&empty);

    // return 0 if successful, otherwise
    return 0;
}
```

The Producer & Consumer Threads

These threads are assigned with tasks *producer()* and *consumer()*, which is implemented as follows:

producer() The producer thread will alternate between sleeping for a random period of time and inserting a random integer into the buffer. To avoid it from sleeping too much, we set the maximal sleeping time as 2s.

```
void *producer(void *param) {
    buffer_item item;

    while (TRUE) {
        // sleep for a random period of time
        sleep(rand() % MAX_SLEEP_TIME);

        // generate a random number
        item = rand();

        if (insert_item(item))
            fprintf(stderr, "producer produced error");
        else
            printf("producer produced %d\n",item);
    }
}
```

Listing 8: producer()

consumer() *consumer()* is very similar to *consumer()*.

```
void *consumer(void *param) {
    buffer_item item;

    while (TRUE) {
        // sleep for a random period of time
        sleep(rand() % MAX_SLEEP_TIME);
        if (remove_item(&item))
            fprintf(stderr, "consumer consumed error");
        else
            printf("consumer consumed %d\n",item);
    }
}
```

Listing 9: consumer()

Result

We set the sleeping time for *main()* to wait as 5s, and use 2 producer threads as well as 3 consuemr threads. The experimental result is presented below.

Figure 2: The Procuder-Consumer Problem Test