操作系统(D)

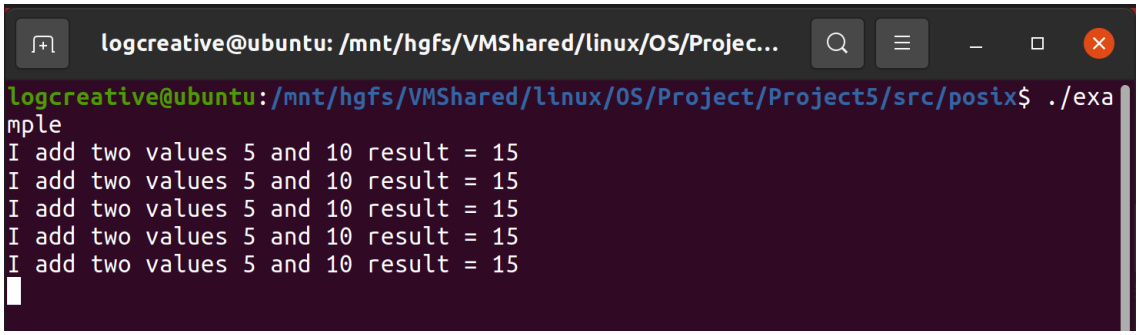
项目 5

李子龙 518070910095

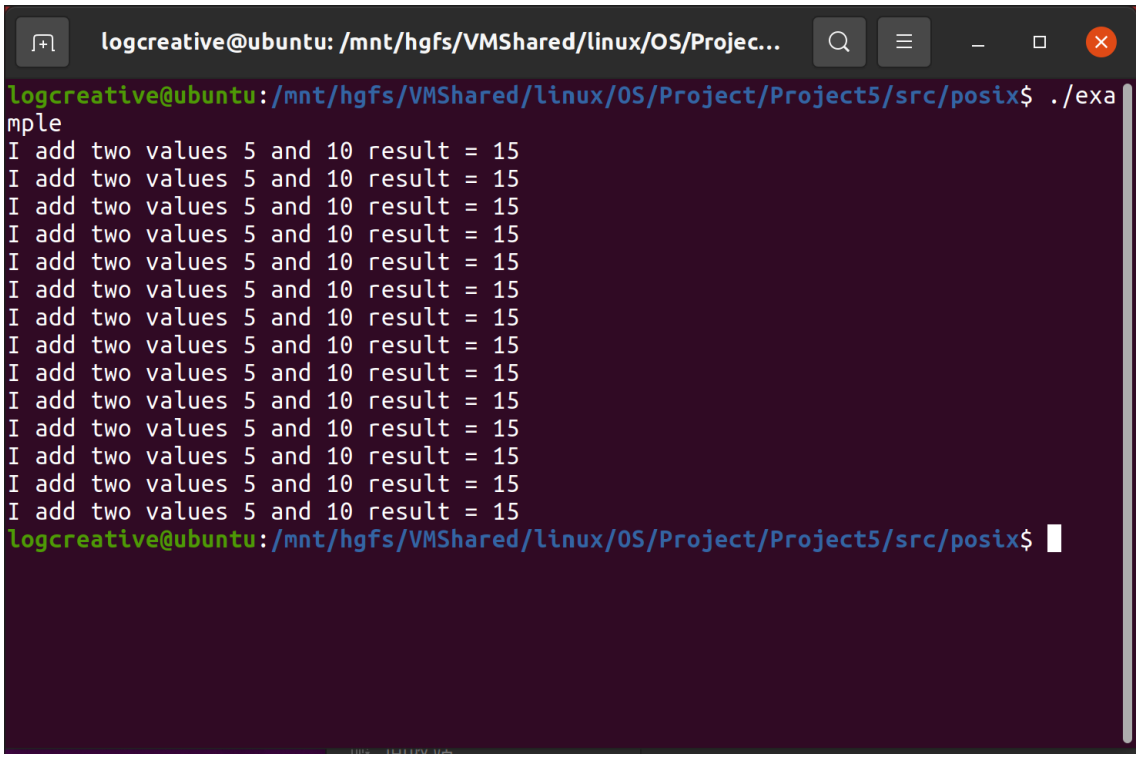
2021 年 3 月 27 日

一 设计线程池

开始时刻，输入5个线程，线程池处理这5个线程。



后来，输入10个线程，但是线程池上限为9个线程，所以又只处理了9个线程。



1. `pool_init()` 中初始化一个互斥锁和一个信号量。首先全局定义了两者后，再初

始化 NUMBER_OF_THREADS = 3 个 worker 线程。注意此处将 sem_submit 初始化为 0。

```
// mutex
pthread_mutex_t queue_mutex;

// semaphore
sem_t sem_submit;

// initialize the thread pool
void pool_init(void)
{
    // mutual-exclusion locks
    pthread_mutex_init(&queue_mutex, NULL);

    // semaphores
    sem_init(&sem_submit, 0, 0);

    for(int i = 0; i < NUMBER_OF_THREADS; ++i)
        pthread_create(&bee[i],NULL,worker,NULL);
}
```

2. pool_submit() 需要使用队列存储任务。这里采用了循环队列。注意循环队列的容量是数据总量 - 1，也就是最多有 9 个任务可以在线程池中。

```
// the work queue
task workqueue[QUEUE_SIZE];

int front = 0, rear = 0;

// insert a task into the queue
// returns 0 if successful or 1 otherwise,
int enqueue(task t)
{

    pthread_mutex_lock(&queue_mutex);

    int res = 0;
    if((rear + 1) % QUEUE_SIZE == front) res = 1;
    else {
        rear = (rear + 1) % QUEUE_SIZE;
        workqueue[rear] = t;
    }

    pthread_mutex_unlock(&queue_mutex);

    return res;
}

// remove a task from the queue
task dequeue()
{

    pthread_mutex_lock(&queue_mutex);

    front = (front + 1) % QUEUE_SIZE;
```

```
        task taskfront = workqueue[front];

        pthread_mutex_unlock(&queue_mutex);

        return taskfront;
    }
```

3. worker() 进程，根据线程池的定义，一旦有可用进程就会从队列中弹出一个进程执行，并将需要服务的请求传递给它。一旦线程完成了服务，它会返回到池中再等待操作。如果池内没有可用线程，那么会等待，直到有空线程为止。这里使用一个信号量管理临界区入口。

```
// the worker thread in the thread pool
void *worker(void *param)
{
    while(TRUE){
        sem_wait(&sem_submit);
        // execute the task
        task worktodo = dequeue();
        execute(worktodo.function, worktodo.data);
    }

    pthread_exit(0);
}
```

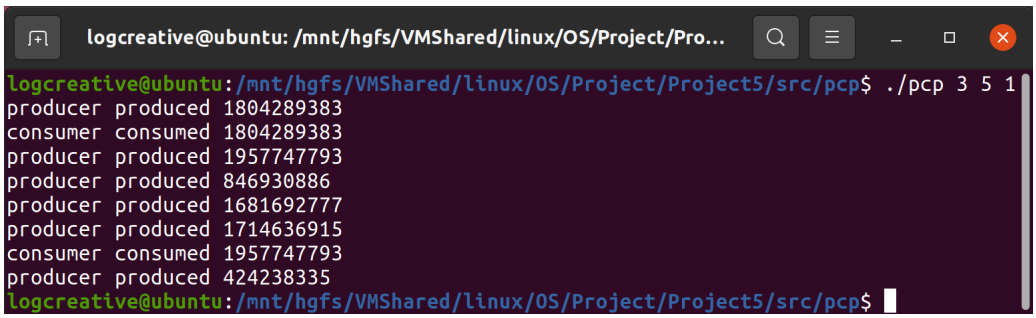
4. 为了防止对队列的同时操作，设置了相关互斥锁，在第 2. 点可见使用了 queue_mutex 进行管理。

5. pool_shutdown() 会首先对每一个线程进行线程撤销，最后进行线程合并。信号量是一个线程撤销点。

```
// shutdown the thread pool
void pool_shutdown(void)
{
    for(int i = 0; i < NUMBER_OF_THREADS; ++i)
        pthread_cancel(bee[i]);
    for(int i = 0; i < NUMBER_OF_THREADS; ++i)
        pthread_join(bee[i],NULL);
}
```

二 生产者–消费者问题

生产者比消费者多的情况:



消费者比生产者多的情况：

两个情况都没有报错，说明互斥锁与信号量运行正常。

1. 缓冲区

   头文件定义如下：

Listing 1: src/pcp/buffer.h

```h
typedef int buffer_item;
#define BUFFER_SIZE 5

int insert_item(buffer_item item);
int remove_item(buffer_item *item);
```

   实现如下：

Listing 2: src/pcp/buffer.c

```c
#include "buffer.h"

buffer_item buffer[BUFFER_SIZE];

int front = 0, rear = 0;

int insert_item(buffer_item item){
    if((rear+1) % BUFFER_SIZE == front) return -1;
    rear = (rear + 1) % BUFFER_SIZE;
    buffer[rear] = item;
    return 0;
}

int remove_item(buffer_item *item){
    if(front==rear) return -1;
    front = (front + 1) % BUFFER_SIZE;
    *item = buffer[front];
    return 0;
}
```

   这里使用了一个循环队列，同上一题。

2. 主函数。

```c
int main(int argc, char *argv[]){
    if(argc!=4){
        fprintf(stderr,"Three parameters are required!\n");
```

```
        return 1;
    }

    int sleep_amount = atoi(argv[1]);
    int p_count = atoi(argv[2]);
    int c_count = atoi(argv[3]);

    buffer_init();

    pthread_t* pbee = (pthread_t *) malloc(p_count*(sizeof(pthread_t)));
    for(int i = 0; i < p_count; ++i)
        pthread_create(&pbee[i], NULL, producer, NULL);

    pthread_t* cbee = (pthread_t *) malloc(c_count*(sizeof(pthread_t)));
    for(int j = 0; j < c_count; ++j)
        pthread_create(&cbee[j], NULL, consumer, NULL);

    sleep(sleep_amount);

    return 0;
}
```

按照要求的几点进行：

1. 获取相关参数。
2. 初始化缓冲区。
3. 创建生产者线程。
4. 创建消费者线程。
5. 主线程休眠以观察生产者与消费者的行为。
6. 退出。

3. 缓冲区初始化函数。

```
pthread_mutex_t mutex;
sem_t empty;
sem_t full;

void buffer_init(){
    pthread_mutex_init(&mutex,NULL);
    sem_init(&empty, 0, BUFFER_SIZE - 1);
    sem_init(&full, 0, 0);
}
```

定义了一个互斥锁，两个信号量。由于循环队列的容量是大小 - 1，所以 `empty` 信号量被初始化为 `BUFFER_SIZE - 1`。

4. 生产者函数。这里认为生产需要花费 1 秒。

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

    while(TRUE){
        sleep(1);
```

```
        item = rand();

        sem_wait(&empty);
        pthread_mutex_lock(&mutex);

        int error = 0;

        if(insert_item(item)){
            fprintf(stderr,"FULL!\n");
            error = 1;
        }
        else
            fprintf(stdout,"producer produced %d\n", item);

        pthread_mutex_unlock(&mutex);
        if(!error) sem_post(&full);
    }
}
```

5. 消费者函数。这里认为消费需要花费 1 秒。

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

    while(TRUE){
        sem_wait(&full);
        pthread_mutex_lock(&mutex);

        int error = 0;
        if(remove_item(&item)){
            fprintf(stderr, "EMPTY!\n");
            error = 1;
        }
        else
            fprintf(stdout, "consumer consumed %d\n", item);

        pthread_mutex_unlock(&mutex);
        if(!error) sem_post(&empty);

        sleep(1);
    }
}
```

A 第一小项全部代码

Listing 3: src/posix/threadpool.c

```
/**
 * Implementation of thread pool.
 */

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

```c
#include <stdio.h>
#include <semaphore.h>
#include "threadpool.h"

#define QUEUE_SIZE 10
#define NUMBER_OF_THREADS 3

#define TRUE 1

// this represents work that has to be
// completed by a thread in the pool
typedef struct
{
    void (*function)(void *p);
    void *data;
}
task;

// the work queue
task workqueue[QUEUE_SIZE];

int front = 0, rear = 0;

// the worker bee
pthread_t bee[NUMBER_OF_THREADS];

// mutex
pthread_mutex_t queue_mutex;

// semaphore
sem_t sem_submit;

// insert a task into the queue
// returns 0 if successful or 1 otherwise,
int enqueue(task t)
{

    pthread_mutex_lock(&queue_mutex);

    int res = 0;
    if((rear + 1) % QUEUE_SIZE == front) res = 1;
    else {
        rear = (rear + 1) % QUEUE_SIZE;
        workqueue[rear] = t;
    }

    pthread_mutex_unlock(&queue_mutex);

    return res;
}

// remove a task from the queue
task dequeue()
{

    pthread_mutex_lock(&queue_mutex);
```

```
    front = (front + 1) % QUEUE_SIZE;
    task taskfront = workqueue[front];

    pthread_mutex_unlock(&queue_mutex);

    return taskfront;
}

// the worker thread in the thread pool
void *worker(void *param)
{
    while(TRUE){
        sem_wait(&sem_submit);
        // execute the task
        task worktodo = dequeue();
        execute(worktodo.function, worktodo.data);
    }

    pthread_exit(0);
}

/**
 * Executes the task provided to the thread pool
 */
void execute(void (*somefunction)(void *p), void *p)
{
    (*somefunction)(p);
}

/**
 * Submits work to the pool.
 */
int pool_submit(void (*somefunction)(void *p), void *p)
{
    task worktodo;
    worktodo.function = somefunction;
    worktodo.data = p;
    int res = enqueue(worktodo);
    if (!res) sem_post(&sem_submit);
    return res;
}

// initialize the thread pool
void pool_init(void)
{
    // mutual-exclusion locks
    pthread_mutex_init(&queue_mutex, NULL);

    // semaphores
    sem_init(&sem_submit, 0, 0);

    for(int i = 0; i < NUMBER_OF_THREADS; ++i)
        pthread_create(&bee[i],NULL,worker,NULL);
}
```

```
// shutdown the thread pool
void pool_shutdown(void)
{
    for(int i = 0; i < NUMBER_OF_THREADS; ++i)
        pthread_cancel(bee[i]);
    for(int i = 0; i < NUMBER_OF_THREADS; ++i)
        pthread_join(bee[i],NULL);
}
```

B 第二小项全部代码

Listing 4: src/pcp/Makefile

```
# makefile for Producer-Consumer Problem
#

CC=gcc
CFLAGS=-Wall
PTHREADS=-lpthread

all: pcp.o buffer.o
    $(CC) $(CFLAGS) -o pcp pcp.o buffer.o $(PTHREADS)

pcp.o: pcp.c
    $(CC) $(CFLAGS) -c pcp.c $(PTHREADS)

buffer.o: buffer.c buffer.h
    $(CC) $(CFLAGS) -c buffer.c $(PTHREADS)

clean:
    rm -rf *.o
    rm -rf pcp
```

Listing 5: src/pcp/pcp.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include "buffer.h"

#define TRUE 1

pthread_mutex_t mutex;
sem_t empty;
sem_t full;

void buffer_init(){
    pthread_mutex_init(&mutex,NULL);
    sem_init(&empty, 0, BUFFER_SIZE - 1);
    sem_init(&full, 0, 0);
}
```

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

    while(TRUE){
        sleep(1);
        item = rand();

        sem_wait(&empty);
        pthread_mutex_lock(&mutex);

        int error = 0;

        if(insert_item(item)){
            fprintf(stderr,"FULL!\n");
            error = 1;
        }
        else
            fprintf(stdout,"producer produced %d\n", item);

        pthread_mutex_unlock(&mutex);
        if(!error) sem_post(&full);
    }
}

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

    while(TRUE){
        sem_wait(&full);
        pthread_mutex_lock(&mutex);

        int error = 0;
        if(remove_item(&item)){
            fprintf(stderr, "EMPTY!\n");
            error = 1;
        }
        else
            fprintf(stdout, "consumer consumed %d\n", item);

        pthread_mutex_unlock(&mutex);
        if(!error) sem_post(&empty);

        sleep(1);
    }
}

int main(int argc, char *argv[]){
    if(argc!=4){
        fprintf(stderr,"Three parameters are required!\n");
        return 1;
    }

    int sleep_amount = atoi(argv[1]);
    int p_count = atoi(argv[2]);
    int c_count = atoi(argv[3]);
```

```
    buffer_init();

    pthread_t* pbee = (pthread_t *) malloc(p_count*(sizeof(pthread_t)));
    for(int i = 0; i < p_count; ++i)
        pthread_create(&pbee[i], NULL, producer, NULL);

    pthread_t* cbee = (pthread_t *) malloc(c_count*(sizeof(pthread_t)));
    for(int j = 0; j < c_count; ++j)
        pthread_create(&cbee[j], NULL, consumer, NULL);

    sleep(sleep_amount);

    return 0;
}
```