实验3 线程与线程池

互联网络程序设计实验

一、实验目的

- 封装线程
- 封装线程互斥变量
- 封装线程条件变量
- 封装线程池

二、实验原理

- 线程是程序执行的基本单位，或者说其是调度的基本单位。在日常的编程中，总会遇到为了充分利用资源，或者为了充分发挥程序高性能而让程序产生线程，以达到在一个进程有多个控制流，并发或者并行执行的情况。
- 线程池是使用线程的一种常用方式，使用它可以避免线程大量创建与消亡带来的性能损失，以及将阻塞的事务交给线程池完成，而使主线程不产生阻塞来达到高性能的目的。

三、问题描述

```
lang@liang:~/linux/thread$ ./pthread_mutex
5000
lang@liang:~/linux/thread$ ./pthread_mutex
5049
lang@liang:~/linux/thread$ ./pthread_mutex
5094
lang@liang:~/linux/thread$ ./pthread_mutex
7250
lang@liang:~/linux/thread$ ./pthread_mutex
10000
lang@liang:~/linux/thread$
```

```c
#include <stdio.h>
#include <pthread.h>
// pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
int g_val = 0;
void* add(void *argv)
{
    for(int i = 0 ; i < 5000; ++i)
    {
        // g_val++;
        // pthread_mutex_lock(&lock);
        int tmp = g_val;
        g_val = tmp+1;
        // pthread_mutex_unlock(&lock);
    }
}

int main(int argc, char const *argv[])
{
    pthread_t id1,id2;

    pthread_create(&id1,NULL,add,NULL);
    pthread_create(&id2,NULL,add,NULL);

    pthread_join(id1,NULL);
    pthread_join(id2,NULL);

    printf("%d\n",g_val);
    return 0;
}
```

- 在上面代码中，我们执行两个线程分别对全局变量累加5000次，但是得到的结果却是不确定的。这是因为，在多线程程序中，线程调度使得线程间进行切换执行，如果当线程1将数据从内存读入CPU正在准备累加时，调度器切换线程2执行，此时，线程2获取的值是未累加的。那么，当两个线程都执行完本次累加后，实际值只增加了1。所以就会产生多次执行，结果不确定性。那么解决这个问题，就需要互斥操作了。

四、互斥量

- 通过互斥量实现线程锁，在每个线程累加之前，进行临界资源的锁操作，在结束时解锁，那么就能保证目标的实现了。


```c
#include <stdio.h>
#include <pthread.h>
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
int g_val = 0;
void* add(void *argv)
{
    for(int i = 0 ; i < 5000; ++i)
    {
        // g_val++;
        pthread_mutex_lock(&lock);
        int tmp = g_val;
        g_val = tmp+1;
        pthread_mutex_unlock(&lock);
    }
}

int main(int argc, char const *argv[])
{
    pthread_t id1,id2;

    pthread_create(&id1,NULL,add,NULL);
    pthread_create(&id2,NULL,add,NULL);

    pthread_join(id1,NULL);
    pthread_join(id2,NULL);

    printf("%d\n",g_val);
    return 0;
}
```

五、条件变量

- 假设我们现在需要做一个生产者消费者模型，生产者对带有头节点的链表头插方式push_front生产数据，消费者调用pop_front消费数据。而生产者可能动作比较慢，这时就会有问题。
- 生产者生产一个数据时间，消费者可能迫切需要。因此，一直轮询申请锁资源，以便进行消费。所以就会产生多次不必的锁资源申请释放动作，影响系统性能。

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
typedef struct node
{
    int _data;
    struct node *_next;
}node_t,* node_p,**node_pp;

node_p head = NULL;

node_p alloc_node(int data)
{
    node_p ret = (node_p)malloc(sizeof(node_t));
    ret->_data = data;
    ret->_next = NULL;
    return ret;
}

void init(node_pp phead)
{
    *phead = alloc_node(0);
}
```

```c
void push_front(node_p head,int data)
{
    node_p tmp = alloc_node(data);
    tmp->_next = head->_next;
    head->_next = tmp;
}

void pop_front(node_p head, int * pdata)
{
    if(head->_next!=NULL)
    {
        node_p tmp = head->_next;
        head->_next = tmp->_next;

        *pdata = tmp->_data;
        free(tmp);
    }
}

void show(node_p head)
{
    node_p cur = head->_next;
    while(cur)
    {
        printf("%d->", cur->_data);
        cur = cur->_next;
    }
    printf("\n");
}
```

```c
//消费者
void * consumer(void *argv)
{
    int data;
    while(1)
    {
        pthread_mutex_lock(&lock);
        // while(head->_next==NULL)
        if(head->_next==NULL)
        {
            printf("producter is not ready\n");
            // pthread_cond_wait(&cond,&lock);
            // break;
        }
        else{
        printf("producter is ready...\n");
        pop_front(head,&data);
        printf("%s  data = %d \n",__func__, data);
        }
        pthread_mutex_unlock(&lock);

        sleep(1);
    }
}
```

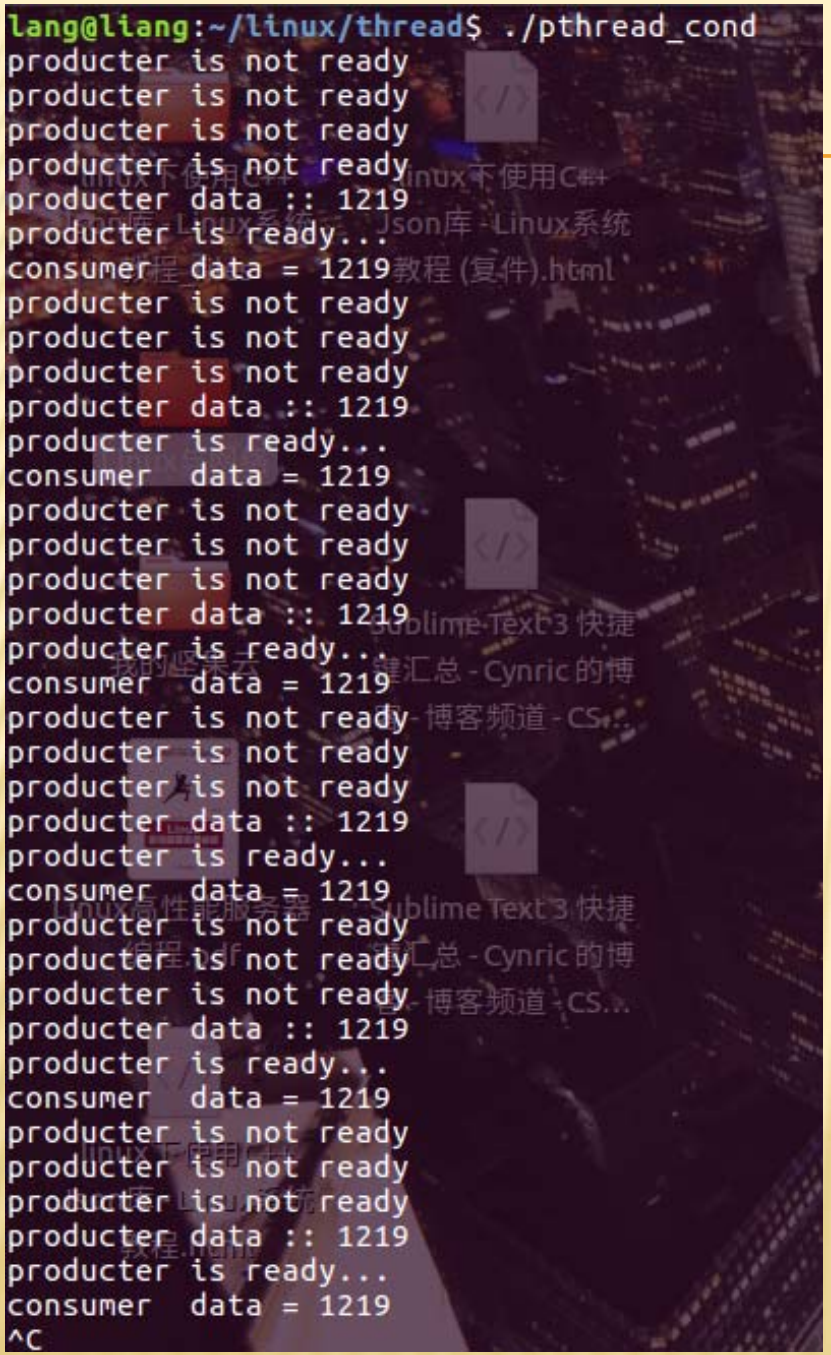
```c
void * producter(void * argv)
{
    int data = rand()%1234;
    while(1)
    {
        sleep(4);
        pthread_mutex_lock(&lock);
        push_front(head,data);
        printf("%s data :: %d\n",__func__, data);
        pthread_mutex_unlock(&lock);
        // pthread_cond_signal(&cond);
    }
}

int main(int argc, char const *argv[])
{
    init(&head);

    pthread_t id1,id2;

    pthread_create(&id1,NULL,consumer,NULL);
    pthread_create(&id2,NULL,producter,NULL);

    pthread_join(id1,NULL);
    pthread_join(id2,NULL);
}
```



- 我们发现。生产者生产一个数据之后，消费者总是会多次进行锁资源申请并尝试消费数据。那么，解决这一问题的方案就是：条件变量。

五、添加条件变量

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
typedef struct node
{
    int _data;
    struct node *_next;
}node_t,* node_p,**node_pp;

node_p head = NULL;

node_p alloc_node(int data)
{
    node_p ret = (node_p)malloc(sizeof(node_t));
    ret->_data = data;
    ret->_next = NULL;
    return ret;
}

void init(node_pp phead)
{
    *phead = alloc_node(0);
}
```

```c
void push_front(node_p head,int data)
{
    node_p tmp = alloc_node(data);
    tmp->_next = head->_next;
    head->_next = tmp;
}

void pop_front(node_p head, int * pdata)
{
    if(head->_next!=NULL)
    {
        node_p tmp = head->_next;
        head->_next = tmp->_next;

        *pdata = tmp->_data;
        free(tmp);
    }
}

void show(node_p head)
{
    node_p cur = head->_next;
    while(cur)
    {
        printf("%d->", cur->_data);
        cur = cur->_next;
    }
    printf("\n");
}
```

```c
//消费者
void * consumer(void *argv)
{
    int data;
    while(1)
    {
        pthread_mutex_lock(&lock);
        while(head->_next==NULL)
        // if(head->_next==NULL)
        {
            printf("producter is not ready\n\n");
            pthread_cond_wait(&cond,&lock);
            break;
        }
        // else{
        // printf("producter is ready...\n");
        pop_front(head,&data);
        printf("%s  data = %d \n",__func__, data);
        // }
        pthread_mutex_unlock(&lock);

        sleep(1);
    }
}
```

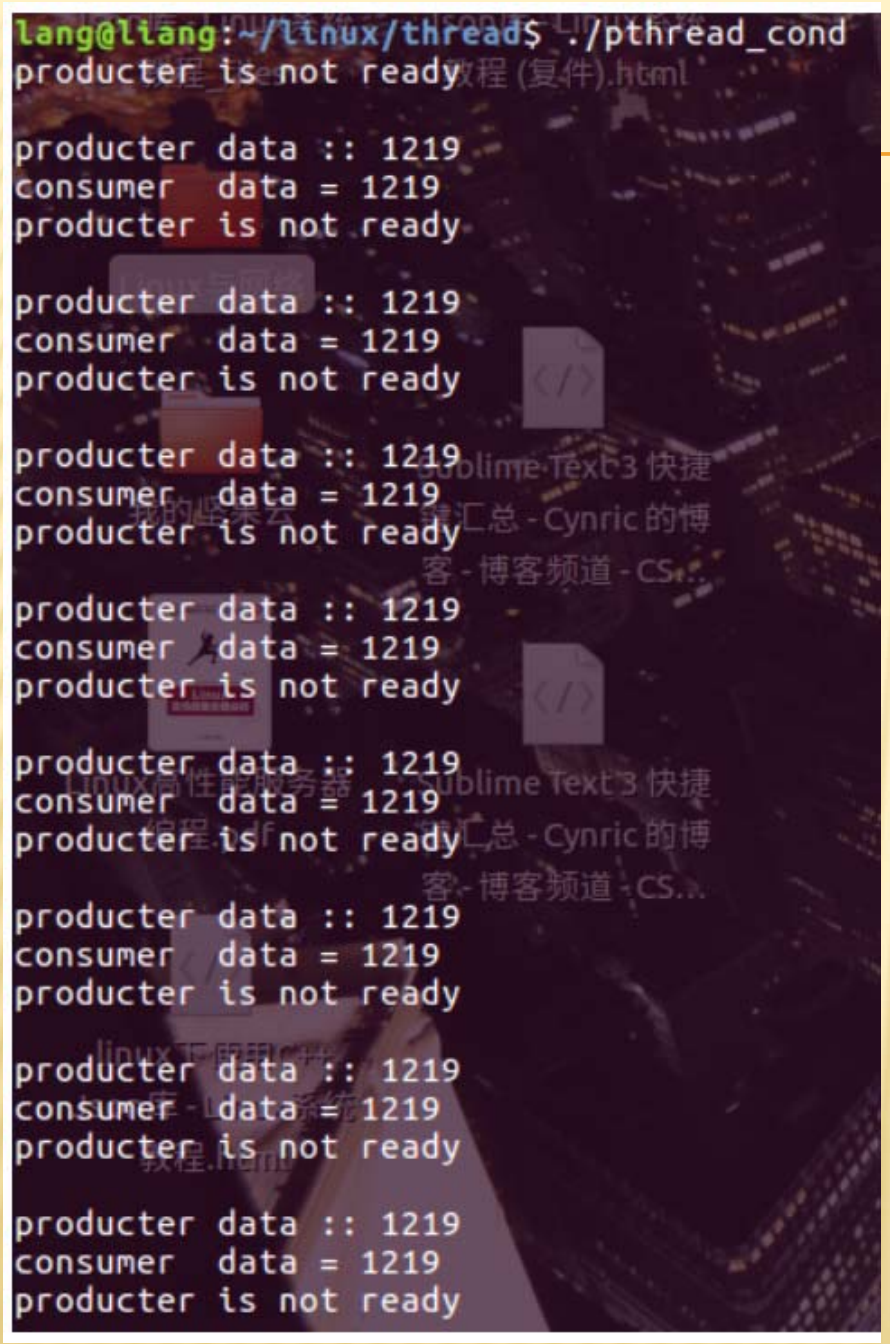
```c
void * producter(void * argv)
{
    int data = rand()%1234;
    while(1)
    {
        sleep(4);
        pthread_mutex_lock(&lock);
        push_front(head,data);
        printf("%s data :: %d\n",__func__, data);
        pthread_mutex_unlock(&lock);
        pthread_cond_signal(&cond); //条件变量v操作
    }
}

int main(int argc, char const *argv[])
{
    init(&head);

    pthread_t id1,id2;

    pthread_create(&id1,NULL,consumer,NULL);
    pthread_create(&id2,NULL,producter,NULL);

    pthread_join(id1,NULL);
    pthread_join(id2,NULL);
}
```



- 由图可以看出，这下我们的消费者不再进行过多次没必要的轮寻访问，当生产者生产数据时，告诉消费者可以进行消费了，那么消费者进行消费。

- 一个Condition Variable总是和一个Mutex搭配使用的。一个线程可以调用pthread_cond_wait在一个Condition Variable上阻塞等待，这个函数做以下三步操作:
  1. 释放Mutex
  2. 阻塞等待
  3. 当被唤醒时,重新获得Mutex并返回

六、LINUX上常用的线程系统调用

- 创建线程

```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict

attr, void *(*start_routine)(void*), void *restrict arg);
```

- 线程退出：

```
#include <pthread.h>

void pthread_exit(void *value_ptr);
```

- 终止线程：

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

- 初始化和销毁线程属性结果：

```
#include <pthread.h>

int pthread_attr_destroy(pthread_attr_t *attr);

int pthread_attr_init(pthread_attr_t *attr);
```

- 得到和设置线程栈大小：

```
#include <pthread.h>

int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,

size_t *restrict stacksize);

int     pthread_attr_setstacksize(pthread_attr_t     *attr,     size_t

stacksize);
```

- 得到和设置线程优先级：

```
#include <pthread.h>
int   pthread_attr_getschedparam(const   pthread_attr_t   *restrict
attr, struct sched_param *restrict param);
int    pthread_attr_setschedparam(pthread_attr_t    *restrict    attr,
const struct sched_param *restrict param);
```

- 分离线程:

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

- 等待线程退出:

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

- 给线程发送信号：

```
#include <signal.h>

int pthread_kill(pthread_t thread, int sig);
```

- 初始化和释放锁：

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);

int  pthread_mutex_init(pthread_mutex_t  *restrict  mutex,  const

pthread_mutexattr_t *restrict attr);

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- 上锁和解锁：

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- 初始化和释放条件变量：

```
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);

int   pthread_cond_init(pthread_cond_t   *restrict   cond,   const

pthread_condattr_t *restrict attr);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

等待和唤醒条件变量：

```
#include <pthread.h>
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
pthread_mutex_t *restrict mutex, const struct timespec *restrict
abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond,
pthread_mutex_t *restrict mutex);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

七、实验内容

- 编写程序，封装线程和线程池，用boost测试库进行测试。

The End