在线实验，请到PC端体验

多线程生产者消费者模型仿真停车场

一、实验简介

1.1 实验内容

通过《多线程生产者消费者模型仿真停车场》项目的学习，不仅可以实现一个基于生产者消费者模型的的停车场停车信息系统，还可以深入理解多线程的创建终止和同步过程，以及线程互斥访问共享数据的锁机制原理。如果对于锁这个操作不熟悉的同学可以参考

http://blog.csdn.net/u011857683/article/details/52336805 (http://blog.csdn.net/u011857683/article/details/52336805)

1.2 知识点

- 生产者消费者模型的概念
- 互斥量的使用，锁机制的实现方式
- `pthread_create`、`pthread_join`、`pthread_mutex_init`、`pthread_cond_init`、`pthread_barrier_wait`、`pthread_cond_wait` 的使用
- 描述停车场的数据结构

1.3 效果截图

- 编译

```
shiyanlou:~/ $ gcc ./Producer_consumer_model.c -o ./pxm -lpthread     [9:32:14]
```

- 运行（假设只有10个停车位）

```
shiyanlou:~/ $ ./pxm 10                                               [9:32:28]
```

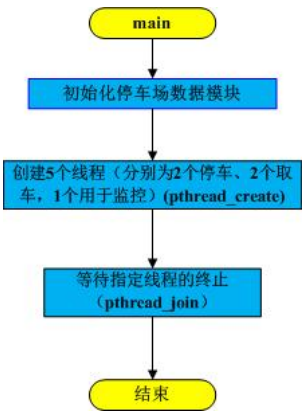
- 结果

```
Number of cars in carpark: 6
Delta: 0
Number of cars in carpark: 8
Delta: 0
Number of cars in carpark: 10
Delta: 0
Number of cars in carpark: 9
Delta: 0
Number of cars in carpark: 8
Delta: 0
Number of cars in carpark: 8
Delta: 0
Number of cars in carpark: 7
Delta: 0
Number of cars in carpark: 9
Delta: 0
Number of cars in carpark: 10
Delta: 0
Number of cars in carpark: 7
Delta: 0
Number of cars in carpark: 5
Delta: 0
Number of cars in carpark: 7
```

技术最有效的方式！ 开始实验

- 注：该结果说明数据在多线程共享的时候有效的保证了一致性。

1.4 设计流程



二、实验步骤

2.1 主要步骤一：main 函数的设计

`main` 函数完成的任务是接收命令行参数并检测其合法性。初始化 `ourpark` 数据结构。

2.1.1 多线程环境

在本项目中有两个生产者（往停车场停车）和两个消费者（从停车场取车）和 一个监督者（实时打印停车场消息）。用 5 个线程实现，代码如下：

```
pthread_create(&car_in, NULL, car_in_handler, (void *)&ourpark);  // 创建往停车场停车线程（生产者1)
pthread_create(&car_out, NULL, car_out_handler, (void *)&ourpark); // 创建从停车场取车线程（消费者1)
pthread_create(&car_in2, NULL, car_in_handler, (void *)&ourpark); // 创建往停车场停车线程（生产者2)
pthread_create(&car_out2, NULL, car_out_handler, (void *)&ourpark); // 创建从停车场取车线程（消费者2)
pthread_create(&m, NULL, monitor, (void *)&ourpark);  // 创建用于监控停车场状况的线程
```

2.1.2 描述停车场的数据结构

- 首先停车场应该包含停车空间，用数组 `carpark` 表示
- 停车场的车辆容量 `capacity`。 停车场现有车辆数目

- 停车场现有车辆数目 occupied
- 下一个进来的车的停车位置 nextin （用 carpark 数组代表的下标表示）
- 下一个取走的车的停车位置 nextout （用 carpark 数组代表的下标表示）
- 记录停车场进入车辆的总和 cars_in
- 记录从停车场开出去的车辆总和 cars_out
- 互斥量 lock ，保护该结构中的数据被线程互斥的方式使用
- 条件变量 space ，描述停车场是否有空位置
- 条件变量 car ，描述停车场是否有车

- 用于线程同步的线程屏障 bar
- 这部分的完整代码：

```
int main(int argc, char *argv[]) {

if (argc != 2) {
    printf("Usage: %s carparksize\n", argv[0]);
    exit(1);
}

cp_t ourpark;

initialise(&ourpark, atoi(argv[1])); // 初始化停车场数据结构

pthread_t car_in, car_out, m;  // 定义线程变量
pthread_t car_in2, car_out2;

pthread_create(&car_in, NULL, car_in_handler, (void *)&ourpark);  // 创建往停车场停车线程（生产者1）
pthread_create(&car_out, NULL, car_out_handler, (void *)&ourpark); // 创建从停车场取车线程（消费者1）
pthread_create(&car_in2, NULL, car_in_handler, (void *)&ourpark); // 创建往停车场停车线程（生产者2）
pthread_create(&car_out2, NULL, car_out_handler, (void *)&ourpark); // 创建从停车场取车线程（消费者2）
pthread_create(&m, NULL, monitor, (void *)&ourpark);  // 创建用于监控停车场状况的线程

// pthread_join 的第二个参数设置为 NULL，表示并不关心线程的返回状态，仅仅等待指定线程（第一个参数）的终止
pthread_join(car_in, NULL);
pthread_join(car_out, NULL);
pthread_join(car_in2, NULL);
pthread_join(car_out2, NULL);
pthread_join(m, NULL);

exit(0);
}
```

2.2 主要步骤二：生产者模型的设计

生产者在本项目中的就是往停车场停车的人，函数原型为： static void* car_in_handler(void *carpark_in)

2.2.1 锁机制

生产者往停车场停车需要读写 ourpark 数据，由于有2个生产者两个消费者，所以应该采用互斥的方式读写，所以要首先获取 ourpark 的锁。代码如下：

```
pthread_mutex_lock(&temp->lock);
```

当完成生产任务（停车），应该释放锁，代码如下：

```
pthread_mutex_unlock(&temp->lock);
```

2.2.2 条件变量

条件变量用来等待一个条件变为真。在生产者这边需要等待停车场有空位,即等待条件 temp->space 为真。如果暂时条件不为真，则需要释放锁，等待消费者的消费，然后再重新获取锁，代码如下：

```
pthread_cond_wait(&temp->space, &temp->lock);
```

当然，当生产者生产了产品（停车），同样也需要产生信号，告诉消费者有产品（车）可消费，代码如下：

```
pthread_cond_signal(&temp->car);
```

- 这部分的代码如下：

```
static void* car_in_handler(void *carpark_in) {

    cp_t *temp;
    unsigned int seed;
    temp = (cp_t *)carpark_in动手实践是学习 IT 技术最有效的方式！                开始实验

    // pthread_barrier_wait 函数表明，线程已完成工作，等待其他线程赶来
    pthread_barrier_wait(&temp->bar);
    while (1) {

        // 将线程随机挂起一段时间，模拟车辆到来的的随机性
        usleep(rand_r(&seed) % ONE_SECOND);

        pthread_mutex_lock(&temp->lock);

        // 循环等待直到有停车位
        while (temp->occupied == temp->capacity)
            pthread_cond_wait(&temp->space, &temp->lock);

        // 插入一个辆车（用随机数标识）
        temp->carpark[temp->nextin] = rand_r(&seed) % RANGE;

        // 各变量增量计算
        temp->occupied++;
        temp->nextin++;
        temp->nextin %= temp->capacity; // 循环计数车辆停车位置
        temp->cars_in++;

        // 可能有的人在等有车可取（线程），这是发送 temp->car 条件变量
        pthread_cond_signal(&temp->car);

        // 释放锁
        pthread_mutex_unlock(&temp->lock);

    }
    return ((void *)NULL);

}
```

2.3 实验主要步骤三： 消费者模型的设计

消费者在本项目中的就是从停车场取车的人，函数原型为： `static void* car_out_handler(void *carpark_in)`

2.3.1 锁机制

消费者从停车场取车需要读写 `ourpark` 数据，由于有2个生产者两个消费者，所以应该采用互斥的方式读写，所以要首先获取 `ourpark` 的锁。代码如下：

```
pthread_mutex_lock(&temp->lock);
```

当完成消费（取车），应该释放锁，代码如下：

```
pthread_mutex_unlock(&temp->lock);
```

2.3.2 条件变量

条件变量用来等待一个条件变为真。在消费者这边需要等待停车场有车,即等待条件 `temp->car` 为真。如果暂时条件不为真，则需要释放锁，等待生产者的生产，然后再重新获取锁，代码如下：

```
pthread_cond_wait(&temp->car, &temp->lock);
```

当然，当消费者消费了产品（取车），同样也需要产生信号，告诉生产者可以生产产品（停车），代码如下：

```
pthread_cond_signal(&temp->space);
```

- 这部分的完整代码如下：

```
static void* car_out_handler(void *carpark_out) {

    cp_t *temp;
    unsigned int seed;
    temp = (cp_t *)carpark_out;
    pthread_barrier_wait(&temp->bar);
    for (; ;) {
        // 将线程随机挂起一段时间，模拟车辆到来的的随机性
        usleep(rand_r(&seed) % ONE_SECOND);

        // 获取保护停车场结构的锁
        pthread_mutex_lock(&temp->lock);

        /* 获得锁后访问 temp->occupied 变量，此时如果车辆数为0（occupied ==0 ），
         pthread_cond_wait 进行的操作是忙等，释放锁（&temp->lock）供其它线程使用。
         直到 &temp->car 条件改变时再次将锁锁住 */
        while (temp->occupied == 0)
            pthread_cond_wait(&temp->car, &temp->lock);

        // 增加相应的增量
        temp->occupied--; // 现有车辆数目减1
        temp->nextout++;
        temp->nextout %= temp->capacity;
        temp->cars_out++;


        // 可能有的人在等有空空车位（线程），这是发送 temp->space 条件变量
        pthread_cond_signal(&temp->space);

        // 释放保护停车场结构的锁
        pthread_mutex_unlock(&temp->lock);

    }
    return ((void *)NULL);

}
```


2.4 监控程序

这部分主要用于实时的监控停车场状况，如停车场的车辆数目等。注意读数据时要上锁，读完数据要解锁。代码如下：

```
pthread_mutex_lock(&temp->lock);
pthread_mutex_unlock(&temp->lock);
```

- 这部分的完整代码如下：

```
// 监控停车场状况
static void *monitor(void *carpark_in) {

cp_t *temp;
temp = (cp_t *)carpark_in;

for (; ;) {
    sleep(PERIOD);

    // 获取锁
    pthread_mutex_lock(&temp->lock);

    /* 证明锁机制保证线程实现的生产者消费者模型正确的方式是：
     temp->cars_in - temp->cars_out - temp->occupied == 0，即总的进来的车 ==
     总的开出去的车 +  停车场现有的车 */
    printf("Delta: %d\n", temp->cars_in - temp->cars_out - temp->occupied);
    printf("Number of cars in carpark: %d\n", temp->occupied);

    // 释放锁
    pthread_mutex_unlock(&temp->lock);

}

return ((void *)NULL);
}
```

三、实验总结

“生产者／消费者模式”，这个模型在很多开发领域都能派上用场。在实际的软件开发过程中，经常会碰到如下场景：某个模块负责产生数据，这些数据由另一个模块来负责处理（此处的模块是广义的，可以是类、函数、线程、进程等）。产生数据的模块，就形象地称为生产者；而处理数据的模块，就称为消费者。本项目以停车场的示例描述多线程实现的“生产者／消费者模式”。

- 本项目的完整代码如下：

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

#define ONE_SECOND 1000000
#define RANGE 10
#define PERIOD 2
#define NUM_THREADS 4

// 定义数据结构描述停车场信息
typedef struct {
    int *carpark; // 用一个整数数组 buffer 模拟停车场停车位
    int capacity; // 停车场的车辆容量
    int occupied; // 停车场现有车辆数目
    int nextin;  // 下一个进来的车的停车位置（用 carpark 数组代表的下标表示）
    int nextout; // 下一个取走的车的停车位置（用 carpark 数组代表的下标表示）
    int cars_in; // 记录停车场进入车辆的总和
    int cars_out;  //记录从停车场开出去的车辆总和
    pthread_mutex_t lock; //互斥量，保护该结构中的数据被线程互斥的方式使用
    pthread_cond_t space; //条件变量，描述停车场是否有空位置
    pthread_cond_t car;   //条件变量，描述停车场是否有车
    pthread_barrier_t bar; //线程屏障
} cp_t;

static void * car_in_handler(void *cp_in);
static void * car_out_handler(void *cp_in);

static void * monitor(void *cp_in);
static void initialise(cp_t *cp, int size);


int main(int argc, char *argv[]) {

    if (argc != 2) {
        printf("Usage: %s carparksize\n", argv[0]);
        exit(1);
    }

    cp_t ourpark;

    initialise(&ourpark, atoi(argv[1])); // 初始化停车场数据结构

    pthread_t car_in, car_out, m;  // 定义线程变量
    pthread_t car_in2, car_out2;

    pthread_create(&car_in, NULL, car_in_handler, (void *)&ourpark);  // 创建往停车场停车线程（生产者1）
    pthread_create(&car_out, NULL, car_out_handler, (void *)&ourpark); // 创建从停车场取车线程（消费者1）
    pthread_create(&car_in2, NULL, car_in_handler, (void *)&ourpark); // 创建往停车场停车线程（生产者2）
    pthread_create(&car_out2, NULL, car_out_handler, (void *)&ourpark); // 创建从停车场取车线程（消费者2）
    pthread_create(&m, NULL, monitor, (void *)&ourpark);  // 创建用于监控停车场状况的线程

    // pthread_join 的第二个参数设置为 NULL，表示并不关心线程的返回状态，仅仅等待指定线程（第一个参数）的终止
    pthread_join(car_in, NULL);
    pthread_join(car_out, NULL);
    pthread_join(car_in2, NULL);
    pthread_join(car_out2, NULL);
    pthread_join(m, NULL);

    exit(0);
}

static void initialise(cp_t *cp, int size) {

    cp->occupied = cp->nextin = cp->nextout = cp->cars_in = cp->cars_out = 0;
    cp->capacity = size;  //设置停车场的大小

    cp->carpark = (int *)malloc(cp->capacity * sizeof(*cp->carpark));

    // 初始化线程屏障，NUM_THREADS 表示等待 NUM_THREADS = 4 个线程同步执行
    pthread_barrier_init(&cp->bar, NULL, NUM_THREADS);


    if (cp->carpark == NULL) {
        perror("malloc()");
        exit(1);
    }
```

```
    srand((unsigned int)getpid());

    pthread_mutex_init(&cp->lock, NULL); // 初始化停车场的锁
    pthread_cond_init(&cp->space, NULL); // 初始化描述停车场是否有空位的条件变量
    pthread_cond_init(&cp->car, NULL); // 初始化描述停车场是否有车的条件变量
}
```



```
static void* car_in_handler(void *carpark_in) {

    cp_t *temp;
    unsigned int seed;
    temp = (cp_t *)carpark_in;

    // pthread_barrier_wait 函数表明，线程已完成工作，等待其他线程赶来
    pthread_barrier_wait(&temp->bar);
    while (1) {

        // 将线程随机挂起一段时间，模拟车辆到来的的随机性
        usleep(rand_r(&seed) % ONE_SECOND);

        pthread_mutex_lock(&temp->lock);

        // 循环等待直到有停车位
        while (temp->occupied == temp->capacity)
            pthread_cond_wait(&temp->space, &temp->lock);

        // 插入一个辆车（用随机数标识）
        temp->carpark[temp->nextin] = rand_r(&seed) % RANGE;

        // 各变量增量计算
        temp->occupied++;
        temp->nextin++;
        temp->nextin %= temp->capacity; // 循环计数车辆停车位置
        temp->cars_in++;

        // 可能有的人在等有车可取（线程），这是发送 temp->car 条件变量
        pthread_cond_signal(&temp->car);

        // 释放锁
        pthread_mutex_unlock(&temp->lock);

    }
    return ((void *)NULL);

}

static void* car_out_handler(void *carpark_out) {

    cp_t *temp;
    unsigned int seed;
    temp = (cp_t *)carpark_out;
    pthread_barrier_wait(&temp->bar);
    for (; ;) {

        // 将线程随机挂起一段时间，模拟车辆到来的的随机性
        usleep(rand_r(&seed) % ONE_SECOND);

        // 获取保护停车场结构的锁
        pthread_mutex_lock(&temp->lock);

        /* 获得锁后访问 temp->occupied 变量，此时如果车辆数为0（occupied ==0 ），
         pthread_cond_wait 进行的操作是忙等，释放锁（&temp->lock）供其它线程使用。
         直到 &temp->car 条件改变时再次将锁锁住 */
        while (temp->occupied == 0)
            pthread_cond_wait(&temp->car, &temp->lock);

        // 增加相应的增量
        temp->occupied--; // 现有车辆数目减1
        temp->nextout++;
        temp->nextout %= temp->capacity;
        temp->cars_out++;


        // 可能有的人在等有空空车位（线程），这是发送 temp->space 条件变量
        pthread_cond_signal(&temp->space);

        // 释放保护停车场结构的锁
        pthread_mutex_unlock(&temp->lock);

    }
```

```
    return ((void *)NULL);

}

// 监控停车场状况
static void *monitor(void *carpark_in) {

    cp_t *temp;
    temp = (cp_t *)carpark_in;

    for (; ;) {
        sleep(PERIOD);

        // 获取锁
        pthread_mutex_lock(&temp->lock);

        /* 证明锁机制保证线程实现的生产者消费者模型正确的方式是:
         temp->cars_in - temp->cars_out - temp->occupied == 0，即总的进来的车 ==
         总的开出去的车 + 停车场现有的车 */
        printf("Delta: %d\n", temp->cars_in - temp->cars_out - temp->occupied);
        printf("Number of cars in carpark: %d\n", temp->occupied);

        // 释放锁
        pthread_mutex_unlock(&temp->lock);

    }

    return ((void *)NULL);
}
```


四、参考资料

- 《UNIX环境高级编程》 (https://book.douban.com/subject/1788421/)

课程教师

bof

共发布过**9**门课程

查看老师的所有课程 > (/teacher/165270)

前置课程

C 语言入门教程 (/courses/57)

Linux多线程编程入门指南 (/courses/731)

进阶课程

C 语言实现多线程排序 (/courses/603)

100 行 C++ 代码实现线程池 (/courses/565)