

第二章 实验四

POSIX 线程(pthread)

1. Pthread 概述

- POSIX 线程, 简称 Pthreads, 是线程的 POSIX(Portable Operating System Interface of UNIX, 缩写为 POSIX) 标准。该标准定义了创建和操纵线程的一整套 API。在类 Unix 操作系统(Unix、Linux、Mac OS X 等)中, 都使用 Pthreads 作为操作系统的线程。Windows 操作系统也可移植版 pthreads-win32。
- Pthreads 定义了一套 C 语言的类型、函数与变量, 它以 pthread.h 头文件和一个线程库实现。

2. pthread_create 创建线程函数简介

- pthread_create 函数声明

```
int pthread_create(pthread_t *tidp, const pthread_attr_t *attr, (void*) (*start_rtn)(void*), void *arg);
```
- pthread_create 参数
 - 1) 第一个参数为指向线程的标识符的指针。tidp 指向的内存单元被设置为新创线程的线程 ID。
 - 2) 第二个参数用来设置线程属性。attr 参数用来制定各种不同的线程属性。
 - 3) 第三个参数是线程运行函数的起始地址。新创线程从 start_rtn 函数的地址开始运行。
 - 4) 第四个函数是运行函数的参数。万能指针参数 arg。
- pthread_create 返回值
若线程创建成功, 则返回 0; 否则线程创建失败, 则返回出错编号。

3. semaphore.h 信号量头文件的函数简介

- 调用 `sem_init();` 初始化 sem_t 型变量, 并设置初始信号量。比如设置为 1。

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

 - 第二个参数 pshared 表示允许几个进程共享该信号量, 一般设 0 用于进程内的多线程共享, 要看是否支持进程共享。
 - 第三个参数 value 表示可用的资源的数目, 即信号灯的数目。例如打印机所以设成 1。
- 当调用 `sem_wait(sem_t *);` 信号量减一, 当信号量为 0 时, sem_wait(); 函数阻塞/等待; 信号量>0 时, 才进行。 (P 操作)

```
int sem_wait(sem_t *sem);
```
- 当调用 `sem_post(sem_t *);` 信号量加一。 (V 操作)

```
int sem_post(sem_t *sem);
```
- 当调用 `sem_destroy(sem_t *);` 销毁信号量

```
int sem_destroy(sem_t *sem);
```

4. pthread_mutex_t 互斥锁简介

- 互斥锁 pthread_mutex_t 使用过程中, 主要用 pthread_mutex_init, pthread_mutex_destroy, pthread_mutex_lock, pthread_mutex_unlock 这几个函数完成锁的初始化、销毁、上锁和解锁操作。

- 锁的创建：静态方式和动态方式
 - 使用宏 `PTHREAD_MUTEX_INITIALIZER` 来静态初始化锁：


```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```
 - 使用 `pthread_mutex_init` 函数动态创建锁，函数原型如下：


```
int pthread_mutex_init(pthread *mutex, const pthread_mutexattr_t *attr)
```
- 锁的销毁

调用 `pthread_mutex_destroy` 之后，可以释放锁占用的资源，前提是当前锁是没有被锁的状态。
- 锁的操作
 - 加锁 `int pthread_mutex_lock(pthread_mutex_t *mutex);` (P 操作)
 - 解锁 `int pthread_mutex_unlock(pthread_mutex_t *mutex);` (V 操作)
 - 测试加锁 `int pthread_mutex_trylock(pthread_mutex_t *mutex);`

注意：`pthread_mutex_trylock()` 语义与 `pthread_mutex_lock()` 类似，不同的是锁已经被占据时返回 `EBUSY` 而不是挂起等待。

5. pthread_join 等待结束函数简介

- `pthread_join` 用来等待一个线程的结束，函数声明：


```
int pthread_join(pthread thread, void **retval);
```
- `pthread_join` 参数
 - 1) 第一个参数 `thread` 是线程标识符，即线程 ID。
 - 2) 第二个参数 `retval` 是用户定义的指针，用来存储被等待线程的返回值。
- `pthread_join` 返回值


若成功，则返回 0；否则，返回出错编号。

6. VC++6.0 配置 pthread 库

[CodeBlocks 配置 pthread 库类似，具体配置步骤可以百度查找]



- 下载 PTHREAD 的 WINDOWS 开发包 `pthreads-w32-2-4-0-release.exe`（任何一个版本均可），解压到一个目录。 <http://sourceware.org/pthreads-win32/>
 下载如下版本：

05/27/2012 12:00 上午 1,228,939 [pthreads-w32-2-9-1-release.zip](#)
- 找到 `include` 文件夹，把它们添加到 `VC++6.0` 的头文件路径下面：






 `pthread.h`
 `sched.h`
 `semaphore.h` 三个头文件放置如下目录：

▶ 计算机 ▶ OS (C:) ▶ Program Files (x86) ▶ Microsoft Visual Studio ▶ VC98 ▶ Include ▶

- 找到 `lib` 文件夹，把它们添加到 `VC++6.0` 的静态链接库路径下面：

 `x64`
 `x86`

lib 文件夹下选择 x86 文件夹里 5 个文件

 `libpthreadGC2.a`
 `libpthreadGCE2.a`
 `pthreadVC2.lib`
 `pthreadVCE2.lib`
 `pthreadVSE2.lib`

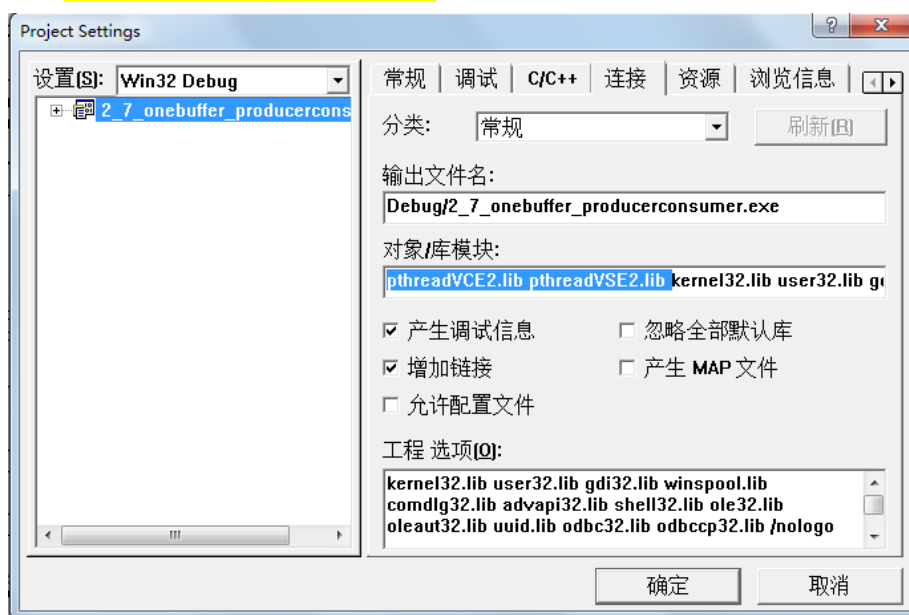
放置如下目录：

计算机 > OS (C:) > Program Files (x86) > Microsoft Visual Studio > VC98 > Lib

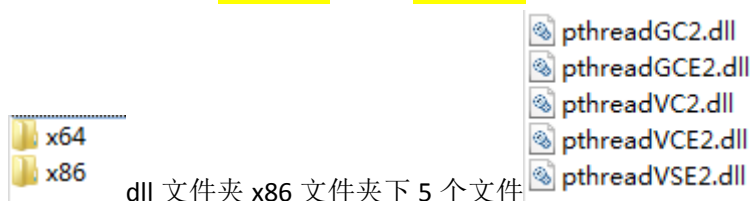
编译后再按照如下设置：

Project->Settings, 选择 Link 页面，然后将 lib 下的 *.lib 文件添加到 Object/library Modules, 各 lib 文件以空格隔开。

pthreadVCE2.lib pthreadVSE2.lib



- 将 dll 文件夹下的 *.dll 文件复制到工程目录下，即根目录。



复制到 *.cpp 程序运行的根目录。

7. 单个缓冲区的生产者与消费者实例

- 多线程并发应用程序有一个经典的模型，即生产者/消费者模型。系统中，产生数据的是生产者，处理数据的是消费者，**消费者和生产者通过一个缓冲区进行数据传递**。
 - 1) 生产者产生数据后提交到缓冲区，然后通知消费者可以从中取出数据进行处理。
 - 2) 消费者处理完数据后，通知生产者可以继续提供数据。
- 关键条件：**消费者和生产者这两个线程进行同步**。
 - 1) 只有缓冲区中有数据时，消费者才能够提取数据；
 - 2) 只有数据已被处理，生产者才能产生数据提交到缓冲区。
- 生产者/消费者模型的信号量和互斥锁的设置：
 - 1) 信号量 sem_t empty; //记录空缓冲区的个数
 - 2) 信号量 sem_t full; //记录装满数据缓冲区的个数
 - 3) 互斥锁 pthread_mutex_t mutex; //消费者和生产者一起去抢互斥锁，谁抢到了这

个锁谁就有资格对这个缓冲仓库进行相关操作。

- 使用 Microsoft Visual Studio C++ 6.0 完善程序 2_7_onebuffer_producerconsumer.cpp，运行读懂程序。

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <string.h>
#include <time.h>
#include <windows.h>

struct data          //信号量结构体
{
    sem_t empty;      //记录空缓冲区个数
    sem_t full;       //记录装满数据缓冲区个数
    int buffer;       //缓冲区
};

pthread_mutex_t mutex; //互斥锁

int num = 0; //记录缓冲区数据的个数

struct data sem;

void* Producer(void *arg)
{
    while(1)
    {
        Sleep(rand()%100);          //随机睡眠

        _____ //信号量的 P 操作
        _____ //互斥锁上锁

        num++;
        printf("Producer 生产了一条数据: %d\n 输入数据: ", num);
        scanf("%d", &sem.buffer);

        _____ //互斥锁解锁
        _____ //信号量的 V 操作
    }
}

void* Consumer(void *arg)
```

```

{
    while(1)
    {

        Sleep(rand()%100);           //随机睡眠

        _____ //信号量的 P 操作
        _____ //互斥锁上锁

        num--;
        printf("Consumer 消费了一条数据: %d\n", num);
        printf("消费数据: %d\n", sem.buffer);

        _____ //互斥锁解锁
        _____ //信号量的 V 操作
    }
}

int main()
{
    sem_init(&sem.empty, 0, 1);    //信号量初始化
    sem_init(&sem.full, 0, 0);

    pthread_mutex_init(&mutex, NULL); //互斥锁初始化

    pthread_t producid;
    pthread_t consumid;

    pthread_create(&producid, NULL, Producer, NULL); //创建生产者线程
    pthread_create(&consumid, NULL, Consumer, NULL); //创建消费者线程

    pthread_join(consumid, NULL); //线程等待，如果没有这一步，主程序会直接结束，
    //导致线程也直接退出。

    sem_destroy(&sem.empty);       //信号量的销毁
    sem_destroy(&sem.full);

    pthread_mutex_destroy(&mutex); //互斥锁的销毁

    return 0;
}

```



```

}Queue;

struct data          //信号量结构体
{
    sem_t empty;      //记录空缓冲区个数
    sem_t full;       //记录装满数据缓冲区个数
    Queue q;          //缓冲仓库：队列
};

```

```

pthread_mutex_t mutex; //互斥锁

```

```

struct data sem;

```

```

int InitQueue (Queue *q)  // 队列初始化

```

```

{
    if (q == NULL)
    {
        return FALSE;
    }

```

```

    q->front = 0;
    q->rear  = 0;

```

```

    return TRUE;
}

```

```

int QueueEmpty (Queue *q)      //判断空对情况

```

```

{
    完善程序;
}

```

```

int QueueFull (Queue *q)       //判断队满的情况

```

```

{
    完善程序;
}

```

```

int DeQueue (Queue *q, int *x) //出队函数

```

```

{
    完善程序;
}

```

```

int EnQueue (Queue *q, int x)  //进队函数

```

```

{
    完善程序;
}

```

```

}

void* Producer(void *arg)
{
    完善程序;
}

void* Consumer(void *arg)
{
    完善程序;
}

int main()
{
    sem_init(&sem.empty, 0, 10);    //信号量初始化
                                   //（做多容纳 10 条消息，容纳了 10 条生产者将不会生产消息）
    sem_init(&sem.full, 0, 0);

    pthread_mutex_init(&mutex, NULL); //互斥锁初始化

    InitQueue(&sem.q);    //队列初始化

    pthread_t producid;
    pthread_t consumid;

    pthread_create(&producid, NULL, Producer, NULL);    //创建生产者线程
    pthread_create(&consumid, NULL, Consumer, NULL);    //创建消费者线程

    pthread_join(consumid, NULL);    //线程等待，如果没有这一步，主程序会直接结束，
                                   //导致线程也直接退出。

    sem_destroy(&sem.empty);        //信号量的销毁
    sem_destroy(&sem.full);

    pthread_mutex_destroy(&mutex);    //互斥锁的销毁

    return 0;
}

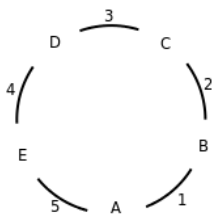
```



```
"D:\OS_pthread\Debug\2_8_nbuffer_producerconsumer.exe"
Producer生产了一条数据: 1
输入数据: 12
Producer生产了一条数据: 2
输入数据: 34
Producer生产了一条数据: 3
输入数据: 56
Producer生产了一条数据: 4
输入数据: 78
Consumer消费了一条数据: 1
消费数据: 12
Producer生产了一条数据: 5
输入数据: 90
Producer生产了一条数据: 6
输入数据: 11
Consumer消费了一条数据: 2
消费数据: 34
Producer生产了一条数据: 7
输入数据: -
```

9. 哲学家就餐问题

- 五个哲学家(不过我们写的程序可以有 N 个哲学家), 这些哲学家们只做两件事——思考和吃饭, 他们思考的时候不需要任何共享资源, 但是吃饭的时候就必须使用餐具, 而餐桌上的餐具是有限的, 原版的故事里, 餐具是叉子, 吃饭的时候要用两把叉子把面条从碗里捞出来。很显然把叉子换成筷子会更合理, 所以: 一个哲学家需要两根筷子才能吃饭。
- 现在引入问题的关键: 这些哲学家很穷, 只买得起五根筷子。他们坐成一圈, 两个人的中间放一根筷子。哲学家吃饭的时候必须同时得到左边和右边边的筷子。如果他身边的任何一位正在使用筷子, 那他只有等着。
- 假设哲学家的编号是 A、B、C、D、E, 筷子编号是 1、2、3、4、5, 哲学家和筷子围成一圈如下图所示:



- 使用 Microsoft Visual Studio C++ 6.0 编程: 程序 2_9_fivephilosophers.cpp。完善如下程序代码:

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <pthread.h>
#include <errno.h>
#include <math.h>
#include <windows.h>

//筷子作为mutex
pthread_mutex_t chopstick[6] ;
```

```

void *eat_think(void *arg)
{
    char phi = *(char *)arg;
    int left, right; //左右筷子的编号
    switch (phi) {
        case 'A':
            left = 5;
            right = 1;
            break;
        case 'B':
            left = 1;
            right = 2;
            break;
        case 'C':
            left = 2;
            right = 3;
            break;
        case 'D':
            left = 3;
            right = 4;
            break;
        case 'E':
            left = 4;
            right = 5;
            break;
    }

    //int i;
    for(;;) {
        Sleep(rand()%1000); //思考

        //补充拿起左右筷子的程序段

        printf("Philosopher %c is eating.\n", phi);
        Sleep(rand()%1000); //吃饭
        pthread_mutex_unlock(&chopstick[left]); //放下左手的筷子
        printf("Philosopher %c release chopstick %d\n", phi, left);
        pthread_mutex_unlock(&chopstick[right]); //放下右手的筷子
        printf("Philosopher %c release chopstick %d\n", phi, right);

    }
}

int main() {
    pthread_t A, B, C, D, E; //5个哲学家

```

```
int i;
for (i = 0; i < 5; i++)
pthread_mutex_init(&chopstick[i], NULL);
pthread_create(&A, NULL, eat_think, "A");
pthread_create(&B, NULL, eat_think, "B");
pthread_create(&C, NULL, eat_think, "C");
pthread_create(&D, NULL, eat_think, "D");
pthread_create(&E, NULL, eat_think, "E");

pthread_join(A, NULL);
pthread_join(B, NULL);
pthread_join(C, NULL);
pthread_join(D, NULL);
pthread_join(E, NULL);
return 0;
}
```