第十课　线程同步

一、竞争与同步

当多个线程同时访问其所共享的进程资源时，
需要相互协调，以防止出现数据不一致、
不完整的问题。这就叫线程同步。

范例：vie.c

理想中的原子++：

线程1		线程2		内存
指　令	寄存器	指　令	寄存器	g_cn
读内存	0			0
算加法	1			0
写内存	1			1
		读内存	1	1
		算加法	2	1
		写内存	2	2

现实中的非原子++：

线程1		线程2		内存
指　令	寄存器	指　令	寄存器	g_cn
读内存	0			0
		读内存	0	0
算加法	1			0
		算加法	1	0
写内存	1			1
		写内存	1	1

二、互斥量

```
int pthread_mutex_init (pthread_mutex_t* mutex,
    const pthread_mutexattr_t* mutexattr);
```

亦可

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_lock (pthread_mutex_t* mutex);

int pthread_mutex_unlock (pthread_mutex_t* mutex);
```

```
int pthread_mutex_destroy (pthread_mutex_t* mutex);
```

1) 互斥量被初始化为非锁定状态；

2) 线程1调用pthread_mutex_lock函数，立即返回，
   互斥量呈锁定状态；

3) 线程2调用pthread_mutex_lock函数，阻塞等待；

4) 线程1调用pthread_mutex_unlock函数，
   互斥量呈非锁定状态；

5) 线程2被唤醒，从pthread_mutex_lock函数中返回，
   互斥量呈锁定状态；

...

范例：mutex.c

三、信号量

信号量是一个计数器，用于控制访问有限共享资源的线程数。

```
#include <semaphore.h>

// 创建信号量
int sem_init (sem_t* sem, int pshared,
    unsigned int value);
```

sem   - 信号量ID，输出。

pshared - 0表示该信号量用于线程间的同步。
          非0表示该信号量用于进程间的同步。
          对于后者，信号量对象被存储在共享内存中。

value   - 信号量初值。

成功返回0，失败返回-1。

```
// 信号量减1，不够减即阻塞
int sem_wait (sem_t* sem);

// 信号量减1，不够减即返回-1，errno为EAGAIN
int sem_trywait (sem_t* sem);

// 信号量减1，不够减即阻塞，
// 直到abs_timeout超时返回-1，errno为ETIMEDOUT
int sem_timedwait (sem_t* sem,
    const struct timespec* abs_timeout);

struct timespec {
    time_t tv_sec;  // Seconds
    long   tv_nsec; // Nanoseconds [0 - 999999999]
};
```

```
// 信号量加1
int sem_post (sem_t* sem);

// 销毁信号量
int sem_destroy (sem_t* sem);
```

范例：sem.c

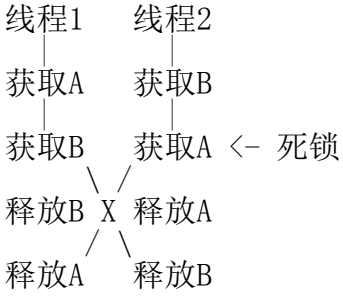
注意：

1) 信号量APIs没有声明在pthread.h中，
   而是声明在semaphore.h中，失败也不返回错误码，
   而是返回-1，同时设置errno。

2) 互斥量任何时候都只允许一个线程访问共享资源，
   而信号量则允许最多value个线程同时访问共享资源，
   当value为1时，与互斥量等价。

范例：pool.c

四、死锁问题



范例：dead.c

五、条件变量

生产者消费者模型

生产者：产生数据的线程。
消费者：使用数据的线程。

通过缓冲区隔离生产者和消费者，与二者直连相比，
避免相互等待，提高运行效率。

生产快于消费，缓冲区满，撑死。
消费快于生产，缓冲区空，饿死。

条件变量可以让调用线程在满足特定条件的情况下暂停。

```
int pthread_cond_init (pthread_cond_t* cond,
    const pthread_condattr_t* attr);
```

亦可

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

// 使调用线程睡入条件变量cond，同时释放互斥锁mutex
int pthread_cond_wait (pthread_cond_t* cond,
    pthread_mutex_t* mutex);

int pthread_cond_timedwait (pthread_cond_t* cond,
    pthread_mutex_t* mutex,
    const struct timespec* abstime);

struct timespec {
    time_t tv_sec;  // Seconds
    long   tv_nsec; // Nanoseconds [0 - 999999999]
};

// 从条件变量cond中唤出一个线程，
// 如果该线程能够获得原先的互斥锁，
// 则从pthread_cond_wait/pthread_cond_timedwait函数中返回，
// 否则继续阻塞于等待加锁
int pthread_cond_signal (pthread_cond_t* cond);

// 从条件变量cond中唤出所有线程，
// 如果其中的某个线程能够获得原先的互斥锁，
// 则从pthread_cond_wait/pthread_cond_timedwait函数中返回，
// 其余线程继续阻塞于等待加锁
int pthread_cond_broadcast (pthread_cond_t* cond);

int pthread_cond_destroy (pthread_cond_t* cond);
```

范例：cond.c

注意：一个线程被从条件变量中唤出以后，
在其重新获得先前被释放的互斥锁之前，
其它线程有可能执行锁区间的代码。
因此从pthread_cond_wait/pthread_cond_timedwait函数返回以后，
有必要对导致该线程睡入条件变量的条件再做一次判断。

范例：bc.c (if->while)

六、哲学家就餐问题

1965年，著名计算机科学家艾兹格·迪科斯彻，提出并解决了一个他称之为哲学家就餐的同步问题。从那时起，每个发明同步原语的人，都希望通过解决哲学家就餐问题来展示其同步原语的精妙之处。

这个问题可以简单地描述如下：

五个哲学家围坐在一张圆桌周围，每个哲学家面前都有一盘面条。由于面条很滑，所以需要一双筷子才能夹住。相邻两个盘子之间放有一根筷子。哲学家的生活中有两种交替活动时段：即吃饭和思考。当一个哲学家觉得饿了时，他就试图分两次去取其左边和右边的筷子，每次拿一根，不分次序。如果成功地得到了两根筷子，就开始吃面条，吃完后放下筷

子继续思考。

图示：dining.png

关键问题是：能为每一个哲学家写一段描述其行为的程序，且决不会死锁吗？

提示：

如果五位哲学家同时拿起各自左边的筷子，那么就没有人能够拿到他们各自右边的筷子，于是发生了死锁。

如果每位哲学家在拿到左边的筷子后，发现其右边的筷子不可用，就先放下左边的筷子，等待一段时间，再重新尝试，那么就可以保证其它哲学家有同时获得两根筷子的机会。但在某一瞬间，所有的哲学家都同时拿起左筷，看到右筷不可用，又都同时放下左筷，等一会儿，又都同时拿起左筷，如此重复下去。虽然程序在不停运行，但都无法取得进展，于是发生了活锁。

思路：

解决问题的关键在于，必须保证任意一位哲学家只有在其左右两个邻居都没有在进餐时，才允许其进入进餐状态。这样做不仅不会发生死锁，而且对于任意数量的哲学家都能获得最大限度的并行性。

范例：dining.c