1. 并行:多个执行命令同时在执行

并发:多个执行命令交替来执行

2. 线程:进程中的一个执行单元 分配CPU的基本单位

线程的组成:线程栈(线程局部的临时资源)

内核对象(操作系统定义的)包括 计数器 挂起计数器 信号 等等

3. 程序运行的四个阶段：预处理、编译、汇编、链接

预处理：解析 #ifdef、#ifndef 、 #define、#include、对文件进行序号标识、对#pragma保留不作处理

编译：对预处理的文件进行词法分析、语法分析、语义校验、编译优化、生成汇编代码

汇编：将汇编代码转化为机器代码

链接：将机器码语言文件加载为.exe文件加载到内存中

4. 进程的五种状态，主要是就绪、堵塞、运行。

5. C++创建线程的两个函数：

_beginthreadex(参数)：这是C++运行库里的函数，先创建内存块，再调用createthread()函数。与之对应的函数是_endthradex(销毁内存块)。

createthread(参数)：这是windows的底层API的函数，有可能会导致内存泄漏。

6. 进程间的同步：

https://www.52pojie.cn/forum.php?mod=viewthread&tid=997742

原子操作：同一时刻只允许一个线程访问资源。

(1).同一进程间的线程同步控制：

通过关键字：::InterlockedDecrement(参数); 参数类型是volatile的，可以实现对特殊地址得稳定访问，防止编译优化。

通过关键资源(临界区)：同一时刻只允许同一个进程中某一个线程访问某个代码段。

关键资源的定义：

```
CRITICAL_SECTION m_cs;
```

关键资源的开始和结束标志：

```
//进入关键资源的函数
EnterCriticalSection(&pthis->m_cs);
//离开关键资源的函数
LeaveCriticalSection(&pthis->m_cs);
```

关键资源的三种方式：

直接堵塞、旋转锁、异步处理。

(2).跨进程的线程控制：

互斥量的方式：同一时刻只允许一个线程访问代码段，安全性更高，可以跨进程。

事件、信号量(信号量可以指定个数)也能跨进程。

7.线程池：预先创建一些线程(一般是CPU核数的两倍)，使其处于睡眠状态(阻塞、挂起、睡眠)，当有任务的时候，再唤醒线程。

8.进程和线程补充

网址：https://www.52pojie.cn/forum.php?mod=viewthread&tid=981120&extra=page%3D1%26filter%3Dtypeid%26typeid%3D28

线程（英语：thread）是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。通俗来讲，对于二进制学习，进程就是映射到内存空间的 4GB，而线程则对应为 EIP（对于 x86 系统来讲）。首先，在 Win32 桌面应用程序中线程的创建是相当常见的。一个设计得体的 GUI 程序的每一个操作都应该新开线程进行操作，避免主界面的卡死。

8.1线程的创建

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes, //安全属性
  SIZE_T dwStackSize,   //线程堆栈大小
  LPTHREAD_START_ROUTINE lpStartAddress,    //线程执行函数的指针
  LPVOID lpParameter,   //执行函数的参数
  DWORD dwCreationFlags,    //线程调度
  LPDWORD lpThreadId    //线程ID
);
```

调用 CreateThread 函数来创建线程，返回值为创建线程的句柄。该函数参数 lpThreadAttributes 为创建线程的安全属性（线程为内核对象，创建内核对象的特征属性），通常设置为 NULL；dwStackSize 为创建线程分配的堆栈大小，一般设置为 0，由系统进行分配；lpStartAddress 为线程的执行函数指针，该函数需符合线程的创建规范：

```
//线程函数
DWORD WINAPI ThreadProc(LPVOID lpParameter){
    //函数操作
    return 0;
}
//创建线程
HANDLE hThread = ::CreateThread(NULL, 0, ThreadProc, NULL, 0, NULL);
```

lpParameter 为线程执行函数的参数，对应执行函数的接收参数，该参数可以是数字，也可以是指向包含其他信息的一个数据结构的指针，执行函数接收后再做相应的处理；dwCreationFlags 为线程创

建的调度参数，设置为 0 即创建线程后便立即调度，设置为 CREATE_SUSPENDED 宏（0x00000004）创建后为挂起状态，等待恢复调用；lpThreadId 为创建线程的 ID 名，方便跨进程调用（内核对象都可以根据 ID 名来跨进程调用）。编码过程，养成好习惯很重要，创建线程后，等待线程结束后，需关闭线程 (::CloseHandle(hThread);)，避免程序长期运行造成内核泄露。

8.2线程的控制

```
//挂起线程：
::SuspendThread(hThread);

//恢复线程：
::ResumeThread(hThread);

//终止线程：
//方式一：
::ExitThread(DWORD dwExitCode);         //结束码
//方式二：
通过线程函数返回，自然结束
//方式三：
::TerminateThread(hThread,dwExitCode);  //线程句柄，结束码

::WaitForSingleObject(hThread,INFINITE);    //线程句柄，等待时间

//判断线程是否结束，成功返回TRUE,失败返回FALSE
BOOL GetExitCodeThread(
  HANDLE hThread,       //线程句柄
  LPDWORD lpExitCode    //结束码
);
```

挂起和恢复线程都没什么好讲的。终止线程，推荐使用方式二来自然终止，以便再结束使用调用析构函数、内存释放、句柄销毁等操作。方式一 ExitThread 方法为同步调用终止线程，终止后会释放整个堆栈空间；方式三 TerminateThread 方法为异步调用终止线程，终止后不会释放堆栈空间。关于同步和异步，同步操作进程会等待线程的结束才结束后面的操作，异步则不会，因此在调用 TerminateThread 来结束线程时，需要配合使用等待函数 WaitForSingleObject(hThread,INFINITE) 来等待结束，避免堆栈混淆。参数 INFINITE 宏（0xFFFFFFFF）为一直等待直到线程结束，也可设置具体的等待时长。

8.3线程结构体的结构

在多线程程序中，线程之间的切换是随时都在发生，具体到毫秒级的更替。从一个线程切换到另一个线程运行时，前一个线程便处于挂起状态，那么这时就需要一个结构体来存储当前挂起线程的相关参数，以便线程切换回来时，之前操作保留下来的值和参数都还存在，避免线程运行出错。CONTEXT

结构便是做这个工作的，每一个线程被挂起时操作系统便会存储一份该线程的 CONTEXT 结构，该结构体具体如下：

```
typedef struct _CONTEXT {

    //
    // The flags values within this flag control the contents of
    // a CONTEXT record.
    //
    // If the context record is used as an input parameter, then
    // for each portion of the context record controlled by a flag
    // whose value is set, it is assumed that that portion of the
    // context record contains valid context. If the context record
    // is being used to modify a threads context, then only that
    // portion of the threads context will be modified.
    //
    // If the context record is used as an IN OUT parameter to capture
    // the context of a thread, then only those portions of the thread's
    // context corresponding to set flags will be returned.
    //
    // The context record is never used as an OUT only parameter.
    //

    DWORD ContextFlags;

    //
    // This section is specified/returned if CONTEXT_DEBUG_REGISTERS is
    // set in ContextFlags.  Note that CONTEXT_DEBUG_REGISTERS is NOT
    // included in CONTEXT_FULL.
    //

    DWORD   Dr0;
    DWORD   Dr1;
    DWORD   Dr2;
    DWORD   Dr3;
    DWORD   Dr6;
    DWORD   Dr7;

    //
    // This section is specified/returned if the
```

```
38      // ContextFlags word contians the flag CONTEXT_FLOATING_POINT.
39      //
40
41      FLOATING_SAVE_AREA FloatSave;
42
43      //
44      // This section is specified/returned if the
45      // ContextFlags word contians the flag CONTEXT_SEGMENTS.
46      //
47
48      DWORD   SegGs;
49      DWORD   SegFs;
50      DWORD   SegEs;
51      DWORD   SegDs;
52
53      //
54      // This section is specified/returned if the
55      // ContextFlags word contians the flag CONTEXT_INTEGER.
56      //
57
58      DWORD   Edi;
59      DWORD   Esi;
60      DWORD   Ebx;
61      DWORD   Edx;
62      DWORD   Ecx;
63      DWORD   Eax;
64
65      //
66      // This section is specified/returned if the
67      // ContextFlags word contians the flag CONTEXT_CONTROL.
68      //
69
70      DWORD   Ebp;
71      DWORD   Eip;
72      DWORD   SegCs;              // MUST BE SANITIZED
73      DWORD   EFlags;             // MUST BE SANITIZED
74      DWORD   Esp;
75      DWORD   SegSs;
```

```
76
```

```
    //
    // This section is specified/returned if the ContextFlags word
    // contains the flag CONTEXT_EXTENDED_REGISTERS.
    // The format and contexts are processor specific
    //

    BYTE    ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];
} CONTEXT;
```

从结构体中清晰可见，CONTEXT 结构中主要保存着直接和 CPU 有关的寄存器信息。其中 ContextFlags 为指定具体进行查询的字段。 可进行查询字段包括：CONTEXT_DEBUG_REGISTERS 调式寄存器、CONTEXT_FLOATING_POINT 浮点寄存器、CONTEXT_SEGMENTS 段寄存器、CONTEXT_INTEGER 通用数据寄存器、CONTEXT_CONTROL 控制寄存器组、CONTEXT_EXTENDED_REGISTERS 扩展寄存器组等，其他也可设置为 CONTEXT_ALL 查询所有字段，CONTEXT_FULL 查询除去 CONTEXT_DEBUG_REGISTERS 调式寄存器其他的所有字段。
具体到 CONTEXT 结构的实际运用，可在逆向时对程序进行 HOOK 操作，通过该结构来获取某一线程挂起时操作系统存储的 CONTEXT 结构，修改结构的相关参数（如 EIP），再将修改后的结构体扔给线程，如若修改 EIP 为 Payload 的地址，便可达到对该程序的 HOOK 操作。

```
//手动挂起某一线程
::SuspendThread(hThread);

//创建 CONTEXT 结构体
CONTEXT context

//设置要获取字段
context.ContextFlags = CONTEXT_CONTROL;

//获取挂起线程操作系统存储的 CONTEXT 结构
BOOL ok = ::GetThreadContext(hThread,&context);

//修改EIP
context.Eip = xxx;        //Payload 地址

//写入到挂起的线程，替换原有 CONTEXT 结构体中的本字段值
::SetThreadContext(hThread,&context);

//恢复线程
```

```
::ResumeThread(hThread);
```

8.3线程的安全问题

```
#include <iostream>
#include <windows.h>
using namespace std;

int cnt = 0;    //全局共有计数变量

DWORD WINAPI ThreadProc(LPVOID Parameter) {
        DWORD id = (DWORD)Parameter;
        for (int i = 0; i < 1000; i++) {
                int temp = cnt;
                Sleep(1);
                temp++;
                cnt = temp;
                cout << "thread" << id << ": " << cnt << endl;
        }
        return 0;
}

int main(){
        HANDLE thr[2];
  //创建线程
        thr[0] = ::CreateThread(NULL, 0, ThreadProc, (LPVOID)1, 0, NULL);
        thr[1] = ::CreateThread(NULL, 0, ThreadProc, (LPVOID)2, 0, NULL);
        //等待创建两个线程结束
        if(!::WaitForMultipleObjects(2,thr,TRUE,INFINITE)) {
                CloseHandle(thr[0]);
                CloseHandle(thr[1]);
                cout << "Final: " << cnt << endl;
        }
        return 0;
}
```

如上代码所示，在主函数中创建了两个线程，执行的操作都是对全局共有的计数变量来进行 1000 次 +1 操作，线程执行函数中通过 Sleep(1) 对线程进行耗时操作，在线程执行的生命周期里的任一时间点可能已经到了线程切换的时间。假使切换时间在 Sleep 时间中，此时线程挂起 temp 获取了 cnt 的值存入了 CONTEXT 结构；在下一个线程执行时 temp 就获取了上一线程同样的 cnt 值，待上一线程恢

复，CONTEXT　中存储的 cnt　值便已不是实时的 cnt 值，程序最终的计算结果便会发生错误。运行该程序可以发现最终的结果小于 2000，并不是原本程序设计所期望的 2000，在线程中的 cout 输出过程中也会发生中断现象。（结果受具体不同的 CPU 运算速度影响，如若在其他同学 PC 上未发生错误，可调节 Sleep 的时间值以达到计算错误。

8.3线程的安全的控制

在上文的线程安全问题中，由于多个线程同时请求同一公有共享资源，导致线程的抢占，前一个线程还没有执行结束就切换到了下一个线程，最终导致多线程程序最终运行的结果不能与实际设计的相对应，造成程序的运行结果有问题。

当多个线程同时对一全局共有资源进行操作时，如若不进行相应的权限控制，便会引发线程安全问题，使得程序最终运行结果和预期设计不符合。

关于解决线程安全问题，于是提出了相应的线程互斥、线程同步的设计。一般来说，习惯使用线程互斥来解决共有资源的抢占问题，使用线程同步来进行线程之间的协同调用。所谓线程互斥即在使用某一公有资源时，某一线程率先拿到使用权，那么另外一个线程只能等待前一线程执行结束才能获取公有资源；线程同步即在多线程程序中，多个线程协同工作，比如前一线程执行结束给下一线程发送信号，下一线程接到信号后便开始执行，这样使得多线程相互之间分工明确，协同工作，互不干扰。

一般来说，一般习惯使用临界区、互斥体来进行线程的互斥操作，使用事件和信号量来进行线程同步控制。

- 线程互斥

1、临界区

对于多线程对全局公有变量抢占的问题，使用临界区来进行控制是最常用的方法。相对与下文提及的互斥体（内核对象），临界区的效率会更高些。

```
#include <iostream>
#include <windows.h>
using namespace std;

int cnt = 0;          //全局公有计数变量
CRITICAL_SECTION cs;  //创建临界区结构体

DWORD WINAPI ThreadProc(LPVOID Parameter) {
        DWORD id = (DWORD)Parameter;
        for (int i = 0; i < 1000; i++) {
                EnterCriticalSection(&cs);  //获取临界区权限
                int med = cnt;
                Sleep(1);
                med++;
                cnt = med;
                cout << "thread" << id << ": " << cnt << endl;
```

```
                LeaveCriticalSection(&cs);  //释放临界区权限
        }
        return 0;
}

int main(){
        ::InitializeCriticalSection(&cs); //初始化临界区
        HANDLE hthr[2];
        hthr[0] = ::CreateThread(NULL, 0, ThreadProc, (LPVOID)1, 0, NULL);
        hthr[1] = ::CreateThread(NULL, 0, ThreadProc, (LPVOID)2, 0, NULL);

        if(!::WaitForMultipleObjects(2,hthr,TRUE,INFINITE)) {
                CloseHandle(hthr[0]); //销毁线程句柄
                CloseHandle(hthr[1]); //销毁线程句柄
                DeleteCriticalSection(&cs); //关闭临界区句柄
                cout << "Final: " << cnt << endl;
        }

        return 0;
}
```

上面代码是对 0x04 线程安全问题的代码使用临界区进行控制的改进代码，通过临界区的控制，其最终结果与程序设计 2000 的设计相符合。所谓互斥，即获得了相关的权限了你才能对目标资源进行使用，否则就需要等待上一个线程释放使用权限，等待线程拿到线程后才能进行自己的操作。

临界区的创捷首先按需要进行创建相应的结构体，使用时需要进行初始化工作。对于使用的互斥来进行线程控制也是一门艺术，获取/释放权限一般在对全局公有变量的前后使用，避免使用互斥来限制了过多了区域，造成了更多的运算消耗。

- 互斥体（内核对象）

互斥体为内核对象，使用方式与临界区大致相同。

```
#include <iostream>
#include <windows.h>
using namespace std;

int cnt = 0;         //全局共有计数变量
HANDLE hmutex;         //互斥体句柄

DWORD WINAPI ThreadProc(LPVOID Parameter) {
        DWORD id = (DWORD)Parameter;
        for (int i = 0; i < 1000; i++) {
```

```
                WaitForSingleObject(hmutex,INFINITE);        //获取互斥体权限
                int med = cnt;
                Sleep(1);
                med++;
                cnt = med;
                cout << "thread" << id << ": " << cnt << endl;
                ReleaseMutex(hmutex);        //释放互斥体权限
        }
        return 0;
}

int main() {
        //创建互斥体（内核对象）
        hmutex = CreateMutex(NULL,        //LPSECURITY_ATTRIBUTES lpMutexAttributes, 指向
                             FALSE,        //BOOL bInitialOwner, 初始化互斥对象的所有者
                             NULL        //LPCTSTR lpName, 互斥体ID的指针
                              );
        HANDLE thr[2];
        thr[0] = ::CreateThread(NULL, 0, ThreadProc, (LPVOID)1, 0, NULL);
        thr[1] = ::CreateThread(NULL, 0, ThreadProc, (LPVOID)2, 0, NULL);

        if (!::WaitForMultipleObjects(2, thr, TRUE, INFINITE)) {
                CloseHandle(thr[0]);
                CloseHandle(thr[1]);
                CloseHandle(hmutex);        //关闭互斥体句柄
                cout << "Final: " << cnt << endl;
        }

        return 0;
}
```

以上代码是对 0x04 线程安全问题的代码使用互斥体进行控制的改进代码，其使用方式与临界区差不多。不过相比临界区，互斥体为内核对象，直接创建后不需要初始化，作为内核对象，互斥体允许程序间跨进程进行调用。

```
//进程一：创建互斥体
HANDLE g_hMutex = CreateMutex(NULL,FALSE, "TestMutex");

//进程二：
```

```
HANDLE g_hMutex = OpenMutex(MUTEX_ALL_ACCESS,FALSE, "TestMutex"); //打开进程一的互斥体对象
WaitForSingleObject(g_hMutex,INFINITE);

//此处写进程二逻辑代码

ReleaseMutex(g_hMutex);
```

对于线程互斥总结的使用方法：

	初始化	获取权限	释放权限	销毁
临界区	InitializeCriticalSection	EnterCriticalSection	LeaveCriticalSection	DeleteCriticalSection
互斥体	CreateMutex	WaitForSingleObject	ReleaseMutex	CloseHandle

互斥体与临界区的区别：

① 临界区只能用于单个进程间的线程控制，互斥体可跨进程调用；

② 互斥体可以设定等待超时，但临界区不能；

③ 线程意外终结时，Mutex 可以避免无限等待；

④ 在执行效率上，临界区优于互斥体，一般做线程互斥优先考虑使用临界区。

- 线程同步

3、事件（内核对象）

和互斥体一样，事件也是一个内核对象。不同于互斥体，事件主要用于做线程同步控制。事件创建之后，一次只能让一个线程获取信号，这一点不同于信号量，因此事件的作用显得过于局限，没有信号量灵活。

```
#include <iostream>
#include <windows.h>
using namespace std;

int cnt = 0;         //全局共有计数变量
HANDLE hEvent;         //事件句柄

DWORD WINAPI ThreadProc(LPVOID Parameter) {
        DWORD id = (DWORD)Parameter;
        for (int i = 0; i < 1000; i++) {
                WaitForSingleObject(hEvent, INFINITE);         //等待事件信号
                int med = cnt;
                Sleep(1);
                med++;
```

```
15                  cnt = med;
16                  cout << "thread" << id << ": " << cnt << endl;
17                  SetEvent(hEvent);         //重新激发信号，信号 +1
18          }
19          return 0;
20  }
21
22  int main() {
23          //创建互斥体（内核对象）
24          hEvent = CreateEvent( NULL,         //事件内核对象安全属性
25                                FALSE,        //信号是否接受触变
26                                TRUE,         //事件创建时信号状态，TRUE为有信号，FALSE无信号
27                                NULL          //事件对象ID
28                                 );
29          HANDLE thr[2];
30          thr[0] = ::CreateThread(NULL, 0, ThreadProc, (LPVOID)1, 0, NULL);
31          thr[1] = ::CreateThread(NULL, 0, ThreadProc, (LPVOID)2, 0, NULL);
32
33          if (!::WaitForMultipleObjects(2, thr, TRUE, INFINITE)) {
34                  CloseHandle(thr[0]);
35                  CloseHandle(thr[1]);
36                  CloseHandle(hEvent);        //关闭事件句柄
37                  cout << "Final: " << cnt << endl;
38          }
39
40          return 0;
41  }
```

以上代码是对 0x04 线程安全问题的代码使用事件进行控制的改进代码，不同于临界区和互斥体，线程互斥只能限制对公有变量使用权限的限定，不限定线程之间怎么进行调用（即抢占使用，可能线程1一直抢占成功，也可能线程2一直抢占成功，线程调用无规律）。但通过事件的线程同步控制两个线程之间将轮流有序地进行调用，线程1过后便是线程2，或者线程2过后就是线程1，感兴趣的同学可以自行进行测试，加深理解。

在此重点提及一下事件的创建函数 CreateEvent 的 4 个参数，参数一和常规内核对象一样，为安全属性，一般直接设定为 NULL；参数二表示线程是否接受触变，为 TRUE 时使用 WaitForSingleObject 等待到信号后不改变状态；为 FALSE 时改变状态，如若设置为TRUE，可使用 ResetEvent(hEvent) 来使信号量 -1 从而使另外一个线程拿不到信号达到阻塞等待，接收到信号的线程在执行完相关的操作后，调用 SetEvent(hEvent) 来激活信号，使信号+1，这时正在阻塞的另外一个线程拿到信号便可运行；参数三设定为事件创建时的信号状态，参数设定为 TRUE 时，事件一创建便为

通知状态，为 FALSE 时为未通知状态，则需要调用 SetEvent(hEvent) 来进行手动设置激发信号，参数四为事件对象的 ID 名。

4、信号量（内核对象）

信号量也是一个内核对象，主要用于做线程同步控制，不同于事件，信号量主要控制在量，可以同时释放多个信号让多个线程来接收，当信号达到设定的阈值便不再增加信号量。可以理解为信号量设定后拥有一个计数器，比如最高同时释放2个信号量，创建时激发了一个信号（+1），当一个线程拿到信号时信号释放（-1），其他的线程此时便处于阻塞状态，如果该线程执行完后激发了两个信号（+2），那么其他阻塞的线程便会有两个能拿到信号开始执行。如果该设定的信号量后面激发了三个信号（+3）也只能激活两个线程，信号量最多只能达到阈值的个数。

```
#include <iostream>
#include <windows.h>
using namespace std;

int cnt = 0;         //全局共有计数变量
HANDLE hSemaphore;         //信号量对象的句柄

DWORD WINAPI ThreadProc(LPVOID Parameter) {
        DWORD id = (DWORD)Parameter;
        for (int i = 0; i < 1000; i++) {
                WaitForSingleObject(hSemaphore, INFINITE);         //等待信号量对象的信号
                int med = cnt;
                Sleep(1);
                med++;
                cnt = med;
                cout << "thread" << id << ": " << cnt << endl;
                ReleaseSemaphore(hSemaphore,1,NULL);         //重新激发信号，信号+1,参数三[
        }
        return 0;
}

int main() {
        //创建信号量（内核对象）
        hSemaphore = CreateSemaphore(NULL,     //信号量内核对象安全属性
                                     1,     //初始资源数量。0时不发送信号
                                     1,     //最大并发数量（阈值），初始资源数量 <= 阈值
                                     NULL   //信号量对象ID
                                     );

        HANDLE thr[2];
```

```
30          thr[0] = ::CreateThread(NULL, 0, ThreadProc, (LPVOID)1, 0, NULL);
31          thr[1] = ::CreateThread(NULL, 0, ThreadProc, (LPVOID)2, 0, NULL);
32
33          if (!::WaitForMultipleObjects(2, thr, TRUE, INFINITE)) {
34                  CloseHandle(thr[0]);
35                  CloseHandle(thr[1]);
36                  CloseHandle(hSemaphore);        //关闭信号量对象的句柄
37                  cout << "Final: " << cnt << endl;
38          }
39
40          return 0;
41  }
```

以上代码是对 线程安全问题的代码使用信号量进行线程同步控制的改进代码，创建新信号阈值为1，激发1个信号（可以理解为事件相似，事件为信号量的一个子集）。与事件一样，通过信号量的线程同步控制两个线程之间将轮流有序地进行调用。

对于线程同步总结的使用方法：

	初始化	信号计数 -1	信号计数 +1	销毁
事件	CreateEvent	ResetEvent / WaitForSingleObject	SetEvent	CloseHandle
信号量	CreateSemaphore	WaitForSingleObject	ReleaseSemaphore	CloseHandle

事件与信号量的区别：

① 二者都是内核对象；

② 事件为信号量的一种特殊形式；

③ 信号量适用面更广，可用于相当复杂的线程同步控制。