软件安全

第1章 并发

徐国胜
北京邮电大学

本讲内容安排

1. 并发机制
2. 常见错误
3. 缓解策略
4. 缓解陷阱
5. 著名漏洞

1.1 多线程

- 并发
  - 一种系统属性，是指系统中几个计算同时执行，并可能彼此交互。
  - 一个并发程序通常使用顺序线程和(或)进程的一些组合来执行计算，其中每个线程和进程执行可以在逻辑上并行执行的计算。这些进程和(或)线程可以在单处理器系统上使用分时抢占式的方式(用一种时间分片的方法使每个线程和(或)进程中的执行步骤交错进行)、在多核多处理器系统中，或者在一个分布式计算系统中执行。
  - 多个控制流并发执行是现代计算环境的重要组成部分。

- 多线程与并发
  - 多线程不一定是并发的，一个多线程程序可以以不并发的方式执行。
  - 一个多线程程序分成可以并发执行的两个或更多线程。每个线程都作为一个单独的程序，但所有线程都在相同的内存中工作，并共享相同的内存。
  - 线程之间的切换速度比进程间切换更快，多个线程可以在多个CPU 上并行执行，以提高性能收益。
  - 单CPU支持多线程（超线程，hyperthreading）

- 多线程安全
  - 多线程因执行次序易产生潜在灾难性问题
  - 一个单线程的程序是完全不会产生任何额外线程的程序。
    - 单线程程序通常并不需要担心同步，并可以受益于强大的单核处理器
    - 单线程不能从多核的性能中受益，因为所有的指令必须在一个处理器的单个线程上按顺序运行。
    - 单线程程序可能有并发问题（如图7-1）

```
01 char *err_msg;
02 #define MAX_MSG_SIZE = 24;
03 void handler(int signum) {
04                 strcpy(err_msg , "SIGINT encountered.");
05 }
06
07 int main(void){
08                 signal(SIGINT, handler);
09                 err_msg = (char*)malloc(MAX_MSG_SIZE);
10 if(err_msg == NULL){
11 /*处理错误条件*/
12 }
13 strcpy(err_msg , "No errors yet.");
14 /* 主代码循环*/
15 return 0;
16 }
```

此程序只使用一个线程，但它采用了两个控制流：一个使用main()函数，另一个使用handler() 函数

如果在调用malloc()的过程中调用信号处理程序，该程序可能崩溃。

多流程的单线程

1.2 并行与并发

- 并行与并发
  - 并发和并行不等价，所有的并行程序都是并发的，但不是所有的并发程序都是并行的。这意味着并发程序既可以用交错、时间分片的方式执行又可以并行执行，如图7.1 和图7.2 所示。
  - 并行计算是“同时使用多台电脑资源解决计算问题”。把问题分解成几部分，再细分成一系列指令。然后来自各部分的指令在不同的CPU 并行运行，实现并行计算。每个部分都必须独立于其他部分并可同时解，最终的结果是多个CPU 比单个CPU 可以在更短的时间内解决问题。

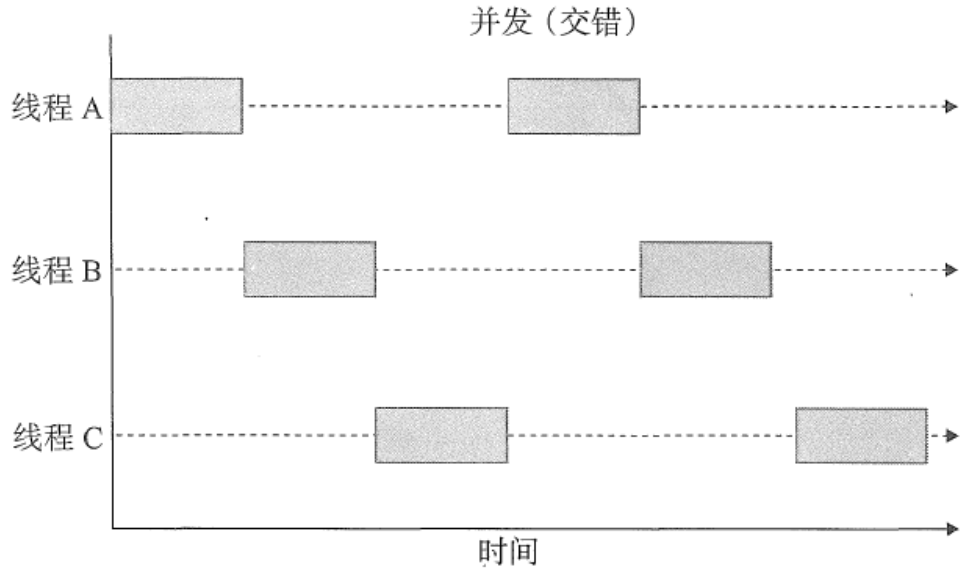

图 7.1　并发交错

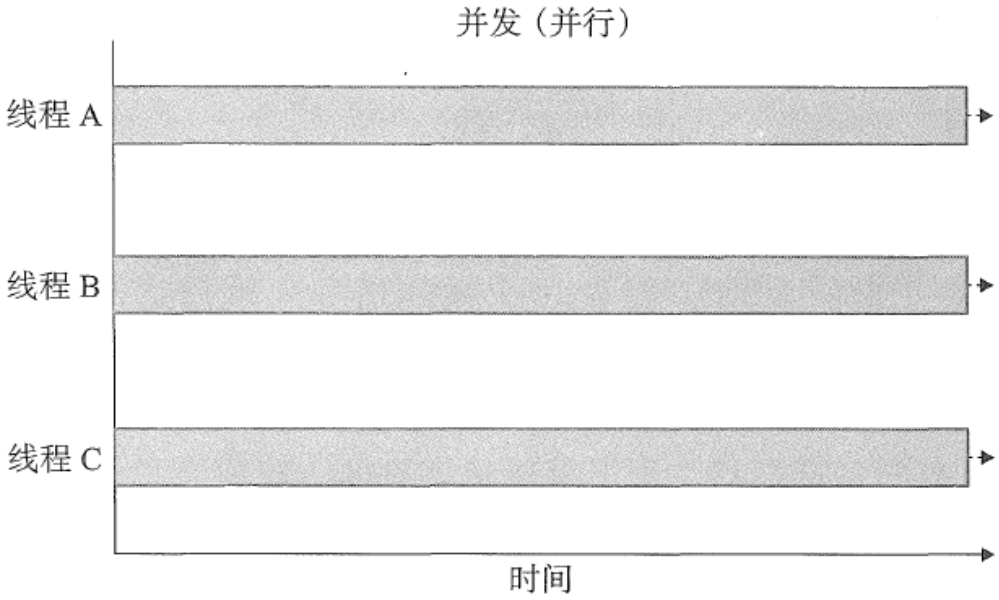

图 7.2　并发并行（需要多核或者多处理器）

1.3 数据并行与任务并行

- 数据并行
  - 并行包括数据并行(*data* parallelism) 和任务并行(*task parallelism)*
  - 如图7.3 所示，数据并行把问题分解成数据段，且并行应用一个函数，在这种情况下，可以利用存储在数组中的字符。数据并行可用于在比顺序处理更短的时间内处理计算单元，它是高性能计算的基础。
  - 例如，要计算一个两维数组的总和，顺序的解决方案将线性地遍历并累加每一个数组项。数据并行把问题划分成单个行，并行地汇总每一行以得到一个子求和的列表，最后汇总每个子求和，从而缩短总的计算时间。

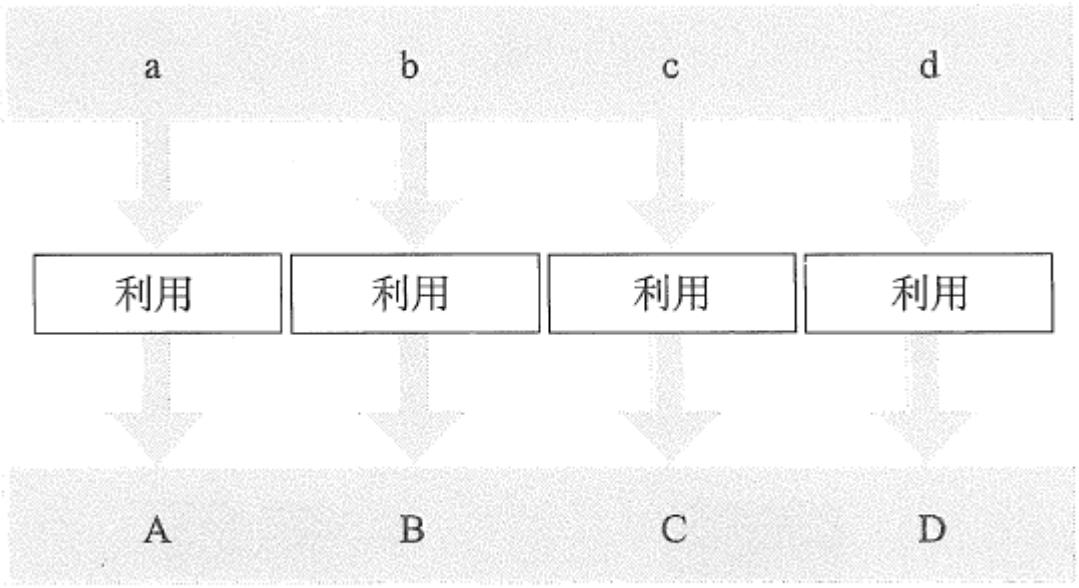

图 7.3　数据并行（资料来源：[Reinders 2007]）

- 任务并行
  - 如图7.4所示，任务并行性指将一个问题分解成可以共享数据的不同任务。各任务在同一时间执行，但执行不同的功能。
  - 因为这种类型的并行性的任务数量是固定的，所以它具有有限的可扩展性。
  - 它由主流操作系统和多种编程语言支持，一般用来提高程序的相应能力。
  - 例如，通过给一个计算分配独立的任务，就能同时计算数据集的平均值、最小值、二进制或和几何平均数。

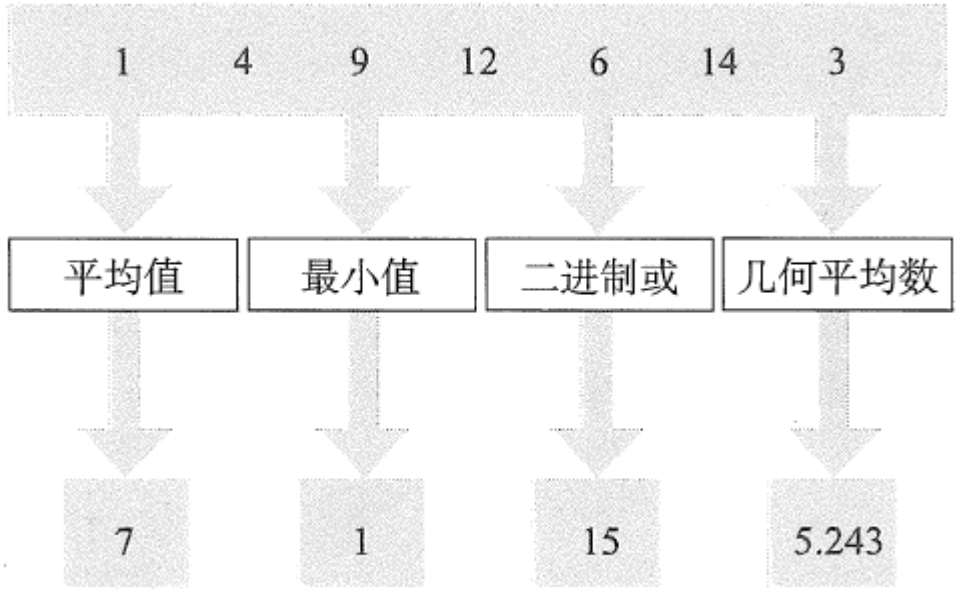

图 7.4 任务并行（资料来源：[Reinders 2007]）

1.4 并行的性能目标

- 并行的性能目标：并行度(parallelism)
  - 用来表示工作(所有指令花费的总时间)跨度(执行最长的并行执行路径或关键路径所花费的时间)比。
  - 所得到的值是沿关键路径的每个步骤完成的平均值，并且是任意数量的处理器可能获得的最大加速比。
  - 可实现的并行度受限于程序结构，依赖于它的关键路径和工作量。

- 图7.5显示了一个现有的程序，它的工作时间是20秒，跨度是10秒。在使用超过两个处理器时，工作跨度比提供的性能增益很少。

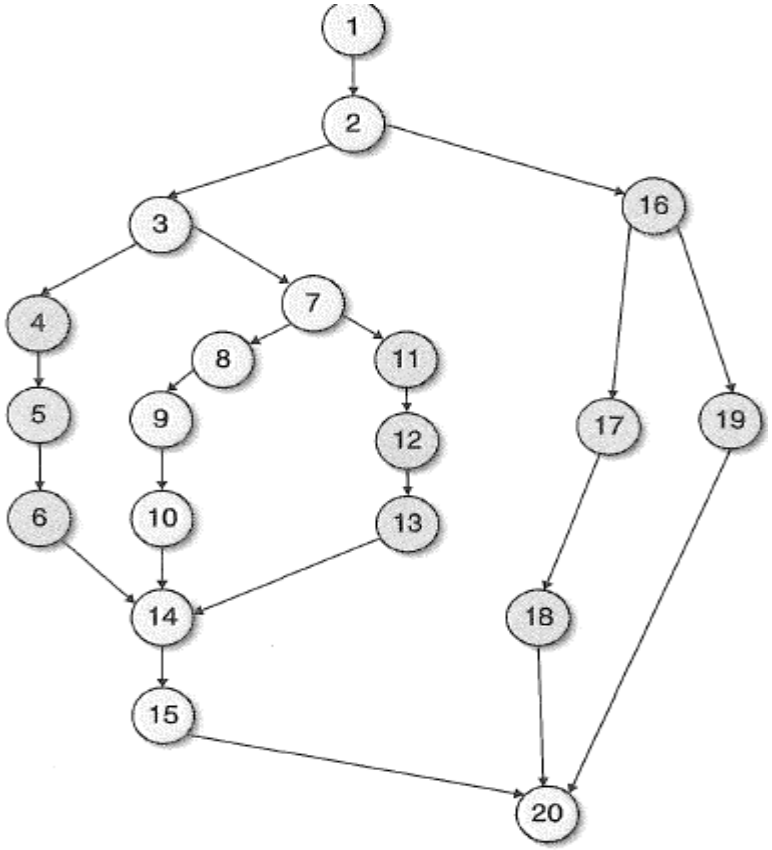

图 7.5　可实现的并行度受限于结构

（资料来源：摘自 [Leiserson 2008]）

- 能够并行地执行的计算越多，优势就越大。这种优势有一个上限，这个上限近似于工作跨度比。然而，重组要并行化的代码是一个昂贵且危险的任务，并且即使在理想的情况下，它也有限制，在如图7.6所示的这个例子中，需要执行的工作是82 秒，跨度是10秒。因此，使用超过8个处理器能获得的性能改进很少。

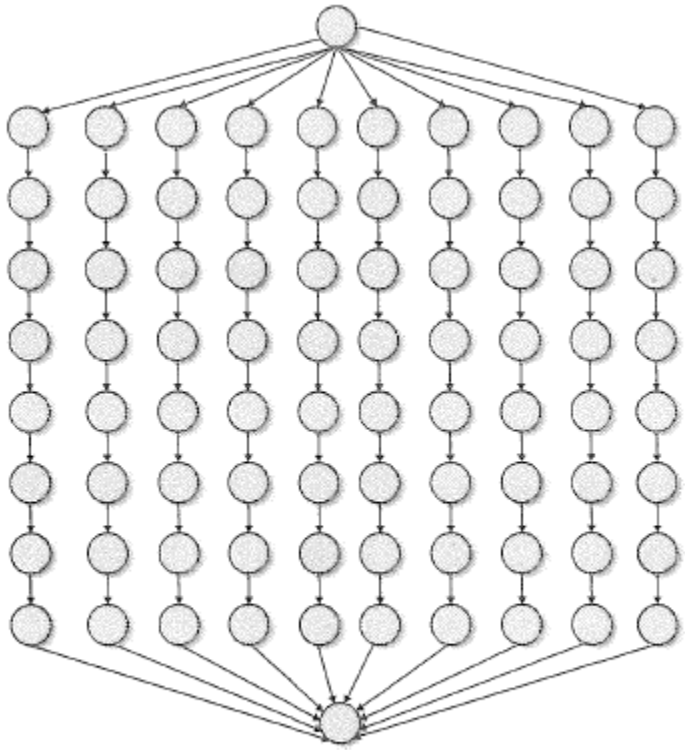

图 7.6　并行的限制（资料来源：[Leiserson 2008]）

本讲内容安排

1. 并发机制
2. 常见错误
3. 缓解策略
4. 缓解陷阱
5. 著名漏洞

2.1 竞争条件

- 竞争条件概念
  - 不受控制的并发可能会导致不确定的行为(即对相同的一组输入，一个程序可能表现出不同的行为)。
  - 在任何情况下，取决于哪个线程首先完成，只要两个线程可以产生不同的行为，都会产生竞争条件。
- 竞争条件存在的三个属性
  - 1.并发属性：至少有两个必须同时执行的控制流。
  - 2.共享对象属性：两个并发流都必须访问一个共享的竞争对象。
  - 3.改变状态属性：至少有一个控制流一定会改变竞争对象的状态。

- 竞争条件的特点
  - 特点：难以察觉、重现和消除，并可能导致错误，如数据损坏或崩溃。
  - 成因：竞争条件是运行时环境导致的，这个运行时环境包括必须对共享资源的访问进行控制的操作系统，特别是通过进程调度进行控制的。

- 例子1：银行取钱
  - 假设丈夫和妻子同时分别从不同的ATM(自动柜员机)尝试从他们的联合储蓄账户中提取所有的钱。两人都查余额，然后取出这一数额。所期望的行为是，允许一个人提取余额，而另一人发现，余额为$0。然而，当双方都看到相同的初始余额且允许两人取出这一金额时，可能就会出现竞争条件。允许竞争条件的代码甚至无法检测到账户余额出现透支!
  - 对于储蓄账户的竞争条件，丈夫和妻子代表并发流，共享对象是储蓄账户，而改变状态属性则通过提款来满足。包含这样的竞争条件的储蓄账户软件可能会被能够协调几个行动者同时使用ATM 的政击者利用。

- 例子2：非线程安全的C++代码示例

```
1 static atomic<int> UnitsSold = 0;
2
3 void IncrementUnitsSold(void) {
4        UnitsSold = UnitsSold + 1;
5 }
```

如果两个单独的线程都调用函数IncrementUnitsSold(),可能会出现如表7.1 所示的竞争条件。

表 7.1　竞争条件示例

时间	线程 1	线程 2
T0	进入 IncrementUnitsSold( ) 函数	
T1		进入 IncrementUnitsSold( ) 函数
T2		载入 (UnitsSold = 0)
T3	载入 (UnitsSold = 0)	
T4		递增 (UnitsSold = 1)
T5		保存 (UnitsSold = 1)
T6	递增 (UnitsSold = 1)	
T7	保存 (UnitsSold = 1)	
T8	返回	
T9		返回

两个线程调用函数后，变量UnitsSold应设置为2，但它却设置为1，这是因为对变量UnitsSold的不同步访问产生的固有的争用条件所致。

2.2 损坏的值

- 例子，8位存储平台上写入16位数值，图表7.2展示了可能的执行场景

```
1 short int x = 0;
2
3 //线程1                    //线程2
4 x = 100;                   x = 300;
```

表 7.2　执行场景

时间	线程 1	线程 2	x
T0		x.low = 44; // 300 % 256	44
T1	x.low = 100;		100
T2	x.high = 0;		100
T3		x.high = 1; // floor(300 / 256)	356

- 防止此类数据损坏最常见的缓解措施是使x成为原子类型。这样做会保证两次写人不能交织进行，并且要求，一旦这些线程已完成x会设置为100或300。

2.3 易变的对象

- 编译器的优化机制
  - 在省略volatile限定符的情况下，除了可能的别名，可以假定指定位置的内容是不变的。（volatile类型限定符告诉标准C语言实现，某些对象可以用实现不能控制的方式改变数值，即易失对象。volatile最常见的用法是对特殊内存地址提供可靠的访问，计算机硬件或中断处理器之类的异步进程使用这些特殊内存地址。）例如下面的程序依赖于接收的一个SIGINT信号来切换一个标志，以终止循环。
  - 然而，因为在main()中的interrupted变量没有声明为volatile，所以对该变量的读取可能被编译器优化掉，导致尽管在信号处理程序中对该变量赋了值，而循环可能永远不会终止。
  - 例如，当使用GCC-O优化标志编译时，该程序甚至接到一个SIGINT也不会终止。

```
01 #include <signal.h>
02
03 sig_atomic_t interrupted; /*错误未声明为volatile*/
04
05 void sigint_handler(int signum) {
06              interrupted = 1; /* 赋值可能是在main()不可见的*/
07 }
08
09 int main(void) {
10              signal(SIGINT, sigint_handler);
11              while (!interrupted) { /* 循环可能永远不会终止*/
12              /* 做一些工作*/
13 }
14 return 0;
15 }
```

- 变量易失性申明（volatitle）
  - 以下例子通过在interrupted 变量声明中加入volatile限定，就保证while循环的每次迭代以及信号处理程序都从原来的地址访问它

```
01  #include <signal.h>
02
03 volatitle sig_atomic_t interrupted; /*错误未声明为volatile*/
04
05 void sigint_handler(int signum) {
06    interrupted = 1; /* 赋值可能是在main()不可见的*/
07 }
08
09 int main(void) {
10    signal(SIGINT, sigint_handler);
11    while (!interrupted) { /* 循环可能永远不会终止*/
12    /* 做一些工作*/
13     }
14     return 0;
15 }
```

变量易失性申明（volatitle）的风险

- 当一个变量被声明为volatile时就会禁止编译器对该内存位置的读取和写入顺序进行重新排列。
- 但是，编译器可能对这些读取、写入和对其他的内存位置的读取和写入的相对顺序进行重新排列。
- 下面的程序片段试图使用volatile变量将非易失性数据结构的状态发送信号到另一个线程。

```
1 volatile int buffer_ready;
2 char buffer[BUF_SIZE];
3
4 void buffer_init() {
5   for (size_t 0; < BUF_SIZE; i++)
6       buffer[i] = 0;
7   buffer_ready = 1;
8 }
```

此处的for 循环，既不访问volatile位置也不执行任何有副作用的操作。因此，编译器可以自由地把循环移动到保存buffer_ready的语句下面，从而挫败开发人员的意图。

- 具有volatitle类型限定符的对象不保证多个线程之间的同步，不防止并发内存访问，也不保证对对象的原子性访问。
- 认为volatile类型限定符保证了原子性、可见性或内存访问顺序是一个误解。

本讲内容安排

1. 并发机制
2. 常见错误
3. 缓解策略
4. 缓解陷阱
5. 著名漏洞

3.1 内存模型

- 一个标准化的线程平台内存模型比以前的内存模型要复杂得多
  - C/C++的内存模型必须提供线程安全性，同时仍然允许细粒度访问硬件，特别是一个平台可能会提供的任何低级别的线程原语。
  - 很容易相信，当两个线程禁止同时访问同一个对象时，程序已经是正确的;然而，因为编译器重新排序(compiler reordering)和可见性(visibility)，这样的程序仍然是危险的。

- 编译器重新排序
  - 在重组程序方面，编译器具有非常大的自由度。

- 理想情况下，当两个线程都完成时，r1和r2都设置为1

- 这段代码对竞争条件缺乏保护，因此，线程1有可能在线程2开始之前完成，在这种情况下，r1仍然会是0；同样，r2保持0也是可能的。
- 还有第4个可能的场景：两个线程可能都完成，但r1和r2仍设直为0! 这种情况是可能的，只是因为允许编译器和处理器对每个线程中的事件重新排序，规定每个线程的行为必须与按指定顺序执行的动作一致。因此，表7.3中所示的执行顺序是有效的。

```
1 int x = 0,  y = 0,  r1 = 0, r2 = 0;
2
3 //线程1                  //线程2
4  x = 1;                 y = 1;
5 r1 = y;                 r2 = x;
```

- ς

表 7.3 执行顺序

时间	线程 1	线程 2
T0	int tmp1 = y; //0	
T1		int tmp2 = x; //0
T2	x = 1;	
T3		y = 1;
T4	r1 = tmp1; //0	
T5		r2 = tmp2; //0

- 可见性
  - 即使编译器以这种方式避免了对语句重新排序（编译以某种形式禁止了这种重排），但硬件配置仍然可能允许这种情况发生。
  - 例如，如果每个线程都由一个单独的处理器执行，每个处理器都有一级高速缓存RAM，镜像一般的内存，那么可能会发生这种情况（高级硬件可以有两级甚至三级从RAM 隔离处理器的缓存内存）：线程1 有可能在线程2读取x的值之前，把x设值为1，但更新后的x值未能在线程2 读取该值之前到达线程2的缓存，从而导致线程2读取x为0的过时的值。

- 数据竞争(Data Race)
  - 如果某个程序在不同的线程中包含两个相互矛盾的动作，其中至少有一个不是原子的，并且两者都不在另一个之前发生，那么执行这个程序包含数据竞争。
  - 任何这样的数据竞争都会导致未定义的行为。
    - 如果两个表达式中一个修改某一内存位置，而另一个读取或修改相同的内存位置，那么这两个表达式求值发生冲突。

- 发生在之前(Happens-Before)
  - 如果两个动作涉及共享数据并由不同的线程执行，这两个动作必须同步。例如，如果第一个线程的动作完成后，它把一个互斥量解锁，并且第二个线程在开始行动之前，把同一个互斥量加锁，那么就建立了相应的"发生在之前"关系。
  - 同样，如果一个原子的值由第一个线程写入， 并随后被第二个线程读出，那么第一个动作发生在第二个动作之前。
  - 最后，如果两个动作在一个线程中发生，那么第一个动作发生在第二个动作之前。

3.2 同步原语

- 可以使用同步原语(synchronization primitive) 建立一个"发生在之前"的关系。
  - c和c++都支持几种不同类型的同步原语，包括互斥变量(mutex variable)、条件变量(condition variable)和锁变量(lock variable)。底层操作系统还支持额外的同步原话，如信号量(semaphore)、管道(pipe)、命名管道(named pipe)和临界区对象(critical section object)。
  - 在竞争窗口之前获取同步对象，然后在窗口结束后释放它，使竞争窗口中关于使用相同的同步机制的其他代码是原子的。
  - 竞争窗口最终成为一个代码临界区。所有临界区对执行临界区的线程以外的所有适当的同步线程都是原子的。

锁机制

- 防止临界区并发执行的策略中，大多数涉及锁机制；
- 锁机制导致一个或多个线程等待，直到另一个线程退出临界区。

```
01  #include <thread>
02  #include <iostream>
03  using namespace std;
04
05  int shared_data = 0;
06
07  void thread_function(int id) {
08    shared_data = id; // shared_data 的竞争窗口开始
09    cout << "Thread " << id << " set shared value to "
10         << shared_data << endl;
11    usleep(id * 100);
12    cout << "Thread " << id << " has shared value as "
13         << shared_data << endl;
14       // shared_data 的竞争结束窗口结束
15  }
16
17  int main(void) {
18    const size_t thread_size = 10;
19    thread threads[thread_size];
20
21    for (size_t i = 0; i < thread_size; i++)
22      threads[i] = thread(thread_function, i);
23
24    for (size_t i = 0; i < thread_size; i++)
25      threads[i].join();
26    // main() 继续之前，等待直到线程完成
```

没有锁机制，不能防止每次调用thread_function()所看到的shared_data不一致

```
01  int shared_lock = 0;
02
03  void thread_function(int id) {
04    while (shared_lock)  // shared_lock 的竞争窗口由此开始
05      sleep(1);
06    shared_lock = 1;     // shared_lock 的竞争窗口到此为止
07    shared_data = id;    // shared_data 的竞争窗口由此开始
08    cout << "Thread " << id << " set shared value to "
09         << shared_data << endl;
10    usleep(id * 100);
11    cout << "Thread " << id << " has shared value as "
12         << shared_data << endl;
13       // shared_data 的竞争窗口到此为止
14    shared_lock = 0;
15  }
```

当该锁由一个线程锁定时，其他线程都不可以进入竞争窗口，直到第一个线程清除锁为止。这使得shared_lock上的竞争窗口互斥。

此方案引入了第二个竞争窗口，即这个共享锁本身的竞争窗口。可能发生下面的情况，即两个线程同时发现锁是空闲的，然后两个线程都把该锁锁定，并继续进入共享数据上的竞争窗口。

试图使用一个对象作为锁的程序员需要一个对象来防止其本身的竞争窗口

互斥量（mutex）

- 最简单的一种锁机制是称为互斥量的一个对象。
- 互斥量有两种可能的状态：锁定和解锁。
  - 一个线程锁定一个互斥量后，任何后续试图锁定该互斥量的线程都将被阻止，直到此互斥量被解锁为止。
  - 当互斥量解锁后，阻塞线程可以恢复执行，并锁定互斥量以继续。
- 互斥量可确保一次只有一个线程可以运行花括号内的代码，从而使程序是线程安全的。互斥量不与任何其他数据关联，只是作为锁对象。

如下图所示，可以锁定和解锁C++的互斥量

- 当对已经锁定的互斥量执行lock()操作时，该函数会被阻塞直到当前持有该锁的线程释放它。
- try_lock()方法试图锁定互斥量，但如果该互斥量已经锁定，它就立即返回，以允许线程执行其他操作。

```
01  mutex shared_lock;
02
03  void thread_function(int id) {
04    shared_lock.lock();
05    shared_data = id;
06    cout << "Thread " << id << " set shared value to "
07         << shared_data << endl;
08    usleep(id * 100);
09    cout << "Thread " << id << " has shared value as "
10         << shared_data << endl;
11    shared_lock.unlock();
12  }
```

互斥量的其他情况

- C++还支持定时的互斥量
  - 它提供try_lock_for() 和try_lock_until()方法。这些方法会被阻塞，直到互斥量成功锁定或经过指定长度的时间。所有其他方法的行为与普通的互斥量相同。
- C++还支持递归互斥量
  - 这些互斥量的行为也像普通的互斥量一样，除了它们允许单个线程不止一次地获取锁，而中间不用解锁。多次锁定一个互斥量的线程，必须解锁相同的次数，之后此互斥量才可以被任何其他线程锁定。非递归互斥量在没有干预解锁时不能被同一个线程多次锁定。
- c++支持既是定时又是递归的互斥量

原子操作

- 原子操作是不可分割的，一个原子操作不能被任何其他的操作中断，当正在执行原子操作时，它访问的内存，也不可以被任何其他机制改变。因此，必须在一个原子操作运行完成后，其他任何事物才能出问该操作所使用的内存，原子操作不能被划分成更小的部分。简单的机器指令，例如，装载一个寄存器，可能是不可中断的。被一个原子加载访问的内存位置不可以由其他任何线程访问，直到此原子操作完成。
- 原子对象是保证它执行的所有操作都是原子的任何对象。通过对某个对象上的所有操作施加原子性，一个原子对象不会被同时读取或写人破坏。原子对象不存在数据竞争，虽然它们仍然可能会受到竞争条件的影响。

3.3 不可变数据结构

- 正如已经看到的，只有当两个或多个线程共享数据而且至少一个线程试图对数据进行修改时，才可能有竞争条件。提供线程安全的一种常用的方法是简单地防止线程修改共享数据，在本质上，即是使数据只读。保护不可改变的共享数据不需要锁。
- 有几个技巧使共享数据只读。一旦初始化，一些类根本无法提供任何方法来修改它们的数据。可以安全地在线程之间共享这些类的对象。
- 在C 和C++中一种常见的战术是声明一个共享对象为const(常量)。

- 另一种方法是复制一个线程可能要修改的任何对象。再次，在这种情况下，所有共享对象都是只读的，任何需要修改一个对象的线程都创建一个共享对象的私有副本，其后只能用它的副本工作。因为副本是私有的，所以共享的对象仍然是不变的。

3.4 并发代码属性
线程安全

- 线程安全函数的使用可以帮助消除竞争条件。根据定义，一个线程安全函数通过锁或其他互斥机制来防止共享资源被并发访问。因此，一个线程安全的函数可以同时被多个线程调用，而不用担心。
- 如果一个函数不使用静态数据或共享资源，它明显是线程安全的。但是，使用全局数据会提高线程安全性，这是一个危险信号，任何全局数据的使用都必须同步，以避免出现竞争状况。为了使一个函数成为线程安全的，它必须同步访问共享资源。特定数据的访问或整个库可以锁定。然而，在库上使用全局锁会导致争用(contention) (在本节后面介绍)。

可重入

- 可重入(reentrant) 函数也可以减轻并发编程错误。函数是可重人的，是指相同函数的多个实例可以同时运行在相同的地址空间中，而不会创建潜在的不一致的状态。
- IBM定义的可重入函数，是指**它在连续调用时不持有静态数据，也不会返回一个指向静态数据的指针**。因此，可重入函数使用的所有数据都由调用者提供，并且可重入函数不能调用不可重入函数。可重入函数可以中断，并重新进入(reentered) 而不会丢失数据的完整性，因此，可重人函数是线程安全的。

线程安全与可重入

线程安全和可重人的概念有类似之处，但存在一些重要的区别。可重入函数一定也是线程安全的，但线程安全的函数却可能无法重人。例如，从多个线程调用时，下面的函数是线程安全的，但不可重入。

increment_counter()函数可以安全地被多个线程调用，因为这里用互斥量来同步访问共享的counter变量。然而，如果此函数被中断处理程序调用、中断并重入，那么第2 次调用就会出现死锁。

```
01  #include <pthread.h>
02
03  int increment_counter () {
04    static int count = 0;
05    static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
06
07    pthread_mutex_lock(&mutex);
08    count++;
09    int result = count;
10    pthread_mutex_unlock(&mutex);
11
12    return result;
13  }
```

本讲内容安排

1. 并发机制
2. 常见错误
3. 缓解策略
4. 缓解陷阱
5. 著名漏洞

4.1 并发实现的常见错误

- 没有用锁保护共享数据(即数据竞争)
- 当锁确实存在时，不使用锁访问共享数据
- 过早释放锁
- 对操作的一部分获取正确的锁，释放它，后来再次取得它，然后又释放它，而正确的做法是一直持有该锁
- 在想要用局部变量时，意外地通过使用全局变量共享数据
- 在不同的时间对共享数据使用两个不同的锁
- 由下列情况引起死锁
  - 不恰当的锁定序列(加锁和解锁序列必须保持一致)
  - 锁定机制使用不当或错误选择
  - 不释放锁或试图再次获取已经持有的锁

其他一些常见的并发陷阱（1）

- 缺乏公平——所有线程没有得到平等的机会来获得处理。
- 饥饿——当一个线程霸占共享资源、阻止其他线程使用时发生。
- 活锁——线程继续执行，但未能获得处理。
- 假设线程将（盲目的认为）
  - 以一个特定的顺序运行
  - 不能同时运行
  - 同时运行
  - 在一个线程结束前获得处理

其他一些常见的并发陷阱（2）

- 假设一个变量不需要锁定，因为开发人员认为只有一个线程写入它且所有其他线程都读取它。这还假定该变量上的操作是原子的。
- 使用非线程安全库。如果一个库能保证由多个线程同时访问时不会产生数据竞争，那么认为它是线程安全的。
- 依托测试，以找到数据竞争和死锁。
- 内存分配和释放问题。当内存在一个线程中分配而在另一个线程中释放时，这些问题可能出现，不正确的同步可能会导致内存仍然被访问时被释放。

4.2 死锁

- 同步原语消除竞争窗口
  - 通过使冲突的竞争窗口互斥，使得一旦一个临界区开始执行时，没有额外的线程可以执行，直到前一个线程退出临界区为止，从而消除竞争条件。
  - 例如，储蓄账户的竞争条件，可以通过把账户余额查询和提款组合成单个原子事务来消除。
- 同步原语的不正确使用可能会导致死锁
  - 当两个或多个控制流以彼此都不可以继续执行的方式阻止对方时，就会发生死锁。
  - 特别是，对于一个并发执行流的循环，如果其中在循环中的每个流都已经获得了导致在循环中随后的流悬停的同步对象，则会发生死锁。

右面的程序说明了死锁的概念

- 此代码将产生一个固定数量的线程，每个线程都修改一个值，然后读取它。
- 虽然通常一个锁就足够了，但是每个线程都用一个锁守卫共享数据值。每个线程都必须获得两个锁，然后才能再访问该值。
- 如果一个线程首先获得锁0，第二个线程获得锁1，那么程序将出现死锁。

```
01  #include <iostream>
02  #include <thread>
03  #include <mutex>
04  using namespace std;
05
06  int shared_data = 0;
07  mutex *locks = NULL;
08  int thread_size;
09
10  void thread_function(int id) {
11    if (id % 2)
12      for (int i = 0; i < thread_size; i++)
13        locks[i].lock();
14    else
15      for (int i = thread_size - 1; i >= 0; i--)
16        locks[i].lock();
17
18    shared_data = id;
19    cout << "Thread " << id << " set data to " << id << endl;
20
21    if (id % 2)
22      for (int i = thread_size - 1; i >= 0; i--)
23        locks[i].unlock();
24    else
25      for (int i = 0; i < thread_size; i++)
26        locks[i].unlock();
27  }
28
29  int main(int argc, char** argv) {
30    thread_size = atoi(argv[1]);
31    thread* threads = new thread[thread_size];
32    locks = new mutex[thread_size];
33
34    for (int i = 0; i < thread_size; i++)
35      threads[i] = thread(thread_function, i);
36
37    for (int i = 0; i < thread_size; i++)
38      threads[i].join();
39    // main() 继续之前，等待直到线程完成
40    delete[] locks;
41    delete[] threads;
42
43    return 0;
44  }
```

- 当前面的代码有5个线程运行时，下面是一个示例输出。在这种情况下，在仅允许一个线程完成后，该程序出现死锁。
  - thread 0 set data to 0 (线程0把数据设置为0)
  - *这里的一个也是可能的结果！！*

- 每个线程都以同一顺序获取锁，可以消除潜在的死锁。下面的程序无论创建多少线程都不会出现死锁。
- 当用五个线程运行这个程序时，下面是产生的一个示例输出。
  - Thread 0 set data to 0
  - Thread 4 set data to 4
  - Thread 2 set data to 2
  - Thread 3 set data to 3
  - Thread 1 set data to 1

```
01  void thread_function(int id) {
02    for (int i = 0; i < thread_size; i++)
03      locks[i].lock();
04
05    shared_data = id;
06    cout << "Thread " << id << " set data to " << id << endl;
07
08    for (int i = thread_size - 1; i >= 0; i--)
09      locks[i].unlock();
10  }
```

- 例7.1中的代码代表两个银行账户，它们之间可以接受现金转账。一个线程把钱从第一个账户转移到第二个账户，而另一个线程把第二个账户的钱转移到第一个账户。

例 7.1　锁定顺序不当造成死锁

```
01  #include <thread>
02  #include <iostream>
03  #include <mutex>
04
05  using namespace std;
06
07  int accounts[2];
08  mutex locks[2];
09
10  void thread_function(int id) { // id 是 0 或 1
11    // 我们把钱从我们的账户转移到其他账户
12    int amount = (id + 2) * 10;
13    lock_guard<mutex> this_lock(locks[id]);
14    lock_guard<mutex> that_lock(locks[!id]);
15    accounts[id] -= amount;
16    accounts[!id] += amount;
17    cout << "Thread " << id << " transferred $" << amount
18         << " from account " << id << " to account " << !id << endl;
19  }
20
21  int main(void) {
22    const size_t thread_size = 2;
23    thread threads[thread_size];
24
25    for (size_t i = 0; i < 2; i++)
26      accounts[i] = 100;
27    for (size_t i = 0; i < 2; i++)
28      cout << "Account " << i << " has $" << accounts[i] << endl;
29
30    for (size_t i = 0; i < thread_size; i++)
31      threads[i] = thread(thread_function, i);
32
33    for (size_t i = 0; i < thread_size; i++)
34      threads[i].join();
35    // main() 继续之前，等待直到线程完成
36
37    for (size_t i = 0; i < 2; i++)
38      cout << "Account " << i << " has $" << accounts[i] << endl;
39    return 0;
40  }
```

- 这个程序通常能完成转账，使得第一个账户有$110的净余额，而第二个账户有$90的净余额。然而，当线程1锁定第一个互斥量且线程2锁定第二个互斥量时，该程序可能会出现死锁。然后线程1必须阻止，直到线程2的互斥量释放，且线程2必须阻止，直到线程1的互斥量释放。因为任何一种条件都不可能发生，所以该程序将出现死锁。
- 为了缓解这个问题，账号应以一个一致的顺序锁定，如下面的代码所示。

- 在这个解决方案中，对两个账户的ID进行比较，ID较低的账户在ID较高的账户之前显式锁定。因此，无论哪个账户是资金的给予者或接收者，每个线程都会先锁定第一个互斥量，再锁定第二个互斥量。
- 如果线程1不是最后一个完成运行的线程，那么例7.2 对thread_function()的修改会导致死锁这是因为线程1永远不会真正解锁它的互斥量，导致其他线程无法获取它。

```
01  void thread_function(int id) { // id 是 0 或 1
02    // 我们把钱从我们的账户转移到其他账户
03    int amount = (id + 2) * 10;
04    int lo_id = id;
05    int hi_id = !id;
06    if (lo_id > hi_id) {
07      int tmp = lo_id;
08      lo_id = hi_id;
09      hi_id = tmp;
10    }
11    // 现在 lo_id < hi_id
12
13    lock_guard<mutex> this_lock(locks[lo_id]);
14    lock_guard<mutex> that_lock(locks[hi_id]);
15    accounts[id] -= amount;
16    accounts[!id] += amount;
17    cout << "Thread " << id << " transferred $" << amount
18         << " from account " << id << " to account " << !id << endl;
19  }
```

例 7.2　不释放锁造成死锁

```
01  void thread_function(int id) {
02    shared_lock.lock();
03    shared_data = id;
04    cout << "Thread " << id << " set shared value to "
05         << shared_data << endl;
06    // 做其他工作
07    cout << "Thread " << id << " has shared value as "
08         << shared_data << endl;
09    if (id != 1) shared_lock.unlock();
10  }
```

- 死锁(和其他的数据竞争)可能对以下条件敏感：
  - 处理器速度
  - 进程或线程调度算法的变动
  - 在执行的时候，强加的不同内存限制
  - 任何异步事件中断程序执行的能力
  - 其他并发执行进程的状态

4.3 过早释放锁

- 考虑在例7.3运行的一个线程数组的代码。每个线程都把一个共享变量设置为它的线程编号，然后打印出共享变量的值。为了防止数据竞争，每个线程都锁定一个互斥量，以使变量被正确地设置。

例 7.3　过早释放锁

```
01  #include <thread>
02  #include <iostream>
03  #include <mutex>
04
05  using namespace std;
06
07  int shared_data = 0;
08  mutex shared_lock;
09
10  void thread_function(int id) {
11    shared_lock.lock();
12    shared_data = id;
13    cout << "Thread " << id << " set shared value to "
14         << shared_data << endl;
15    shared_lock.unlock();
16    // do other stuff
17    cout << "Thread " << id << " has shared value as "
18         << shared_data << endl;
19  }
20
21  int main(void) {
22    const size_t thread_size = 3;
23    thread threads[thread_size];
24
25    for (size_t i = 0; i < thread_size; i++)
26      threads[i] = thread(thread_function, i);
27
28    for (size_t i = 0; i < thread_size; i++)
29      threads[i].join();
30    // main() 继续之前，等待直到线程完成
31
```

- 遗憾的是，当共享变量的每一个写操作都由互斥量所保护时，随后的读取是不受保护的。使用三个线程调用此程序产生下面的输：
  - Thread 0 set shared value to 0
  - Thread 0 has shared value as 0
  - Thread 1 set shared value to 1
  - Thread 2 set shared value to 2
  - Thread 1 has shared value as 2
  - Thread 2 has shared value as 2
  - Done

- 读取和写人共享数据都必须受到保护，以确保每一个线程读取到它写入的相同的值。将临界区扩展为包括读取值，此代码就呈现为线程安全的，如下所示。

```
01  void thread_function(int id) {
02    shared_lock.lock();
03    shared_data = id;
04    cout << "Thread " << id << " set shared value to "
05         << shared_data << endl;
06    // do other stuff
07    cout << "Thread " << id << " has shared value as "
08         << shared_data << endl;
09    shared_lock.unlock();
10  }
```

- 下面是这段代码的正确输出的一个示例。需要注意的是，线程的顺序仍然可以有所不同，但每个线程都正确地打印出线程编号。
  - Thread 0 set shared value to 0
  - Thread 0 has shared value as 0
  - Thread 1 set shared value to 1
  - Thread 1 set shared value to 1
  - Thread 2 has shared value as 2
  - Thread 2 has shared value as 2
  - Done

4.4 争用

- 当一个线程试图获取另一个线程持有的锁时，就会发生锁争用。有些锁争用是正常的，这表明，锁正在"工作"以防止竞争条件。过多的锁争用会导致性能不佳。
- 减少持有锁的时间量或通过降低每个锁保护的粒度或资源量，可以解决锁争用导致的性能差的问题。持有锁的时间越长，另一个线程尝试获取锁，并被迫等待的概率将越大。反之，减少持有锁的持续时间就减少了争用。例如，不会作用于共享资源的代码，不需要在临界区之内得到保护，并可以与其他线程并行运行。在一个临界区之内执行一个阻塞操作延伸了临界区的持续时间，从而增加了潜在的争用。在临界区之内的阻塞操作也可能导致死锁。在临界区之内执行阻塞操作几乎始终是一个严重的错误。

- 锁的粒度也可以影响争用。增加由一个单一锁保护的共享资源的数量，或扩大共享资源的范围(例如，锁定整个表以访问一个单元格)，将使在同一时间多个线程尝试访问该资源的概率增大。
- 在选择锁的数量时，增加锁的开销和减少锁争用之间有一个权衡。更细的粒度(每个保护少量的数据)需要更多的锁，使得锁本身的开销增加。额外的锁也会增加死锁的风险。锁一般是相当快的，但是，当然单个执行线程运行速度会比没有锁更慢。

本讲内容安排

1. 并发机制
2. 常见错误
3. 缓解策略
4. 缓解陷阱
5. 著名漏洞

5.1 在多核动态随机访问存储器系统中的DoS攻击

- 现在的随机存取存储器(DRAM)内存系统不区分运行在不同的核上的不同线程的内存访问请求。这种区分的缺乏使得多核系统容易受到利用内存系统中的不公平发起的攻击。这个不公平，允许一个线程使用一个特定的内存访问模式，比来自其他线程的内存访问请求优先获得对内存的访问。这种优先访问导致其他线程访问内存的时间变长。针对在目前的DRAM内存系统中为何一个线程可以拒绝另一个线程的服务，给出了两个主要的原因：

- 1.不公平的行命中优先(row-hit-first) 调度:访问导致行命中(称为一个高的行缓冲位置)的线程与访问导致行冲突的线程相比，前一个获得更高的优先级。因此，如果碰巧访问相同的DRAM 条带，那么一个具有较高的行缓冲位置的应用程序(例如，一个通过内存流的应用程序)可以明显延缓另一个具有较低的行缓冲位置的应用程序。
- 2.不公平的最旧优先(oldest-first) 调度:最旧优先调度隐含着赋予那些可以以更快速率产生内存请求的线程更高的优先级。这种侵略性的线程可以用它们的请求淹没内存系统，以比内存系统更快的速度服务。因此，侵略性的线程可以以它们的请求填满存储系统的缓冲区，而内存访问密集程度较低(less-memory-intensive) 的线程被内存系统阻塞，直到所有的从侵略性的线程较早到达的请求都得到服务为止。

- 在多核系统上使用这些内存访问技术，使得攻击者可以拒绝或延缓其他线程的内存访问。预计随着时间的推移，硬件的这类问题将得到纠正，以确保公平。

5.2系统调用包装器中的并发漏洞

- 系统调用插入是用于提高操作系统安全策略的一个内核扩展技术(被商业杀毒软件广泛使用)，但在与当前的操作系统结合时，它变得脆弱，并可能导致权限提升和绕过审计，下面的系统调用包装器都已知存在了并发的漏洞。

- 通用软件包装器工具包(Generic Software Wrappers Toolkit ,GSWKT)(CVE-2007-4302):在GSWTK中某些系统调用包装器中的多竞争条件允许本地用户挫败系统调用插入，并可能获得权限或绕过审计。
- Systrace：Systrace是多种操作平台的访问控制系统。sysjail实用程序是使用Systrace框架的一种遏制设施。sudo 实用程序是一种权限管理工具；一个只在CVS预发布的sudo版本包括一个基于Systrace 的监控模式。因为Systrace 实现了自己的系统调用包装器，所以它容易发生多个并发漏洞。sudo(监控模式)和sysjail都使用了此功能。
- Cerb CerbNG (CVE-2007-4303): (1) 一定的规则和(2) 在应用于FreeBSD 4.8中的CerbNG 虚拟机保护的参数复制中，多竞争条件允许本地用户挫败系统调用插人，并可能获得权限或绕过审计，如通过修改log-exec.cb 中的命令行来展示。

- 已确定在系统调用包装器中存在三种形式的并发漏洞:
  - 1.包装器逻辑中的同步错误导致不正确的操作，如不正确地锁定数据。
  - 2. 在复制系统调用参数时，包装器和内核之间缺乏同步导致数据竞争，使得包装器和内核处理的参数不同。
  - 3. 在解释系统调用参数时，包装器和内核之间缺乏同步导致数据竞争。

- 在论文“ Exploiting Concurrency Vulnerabilities in System Call Wrappers” (系统调用包装器中的并发漏洞利用)中Robert Watson 观察到最常见的可识别和可利用的漏洞可分为三[Watson 2007]：

- 1.检查时间到使用时间(time-of-check-to-time-of-use , TOCTTOU; 也被称为检查时间，使用时间，或TOCTOU) 漏洞，其中访问控制检查与它们所保护的操作是非原子的，允许攻击者违反访问控制规则。
- 2.审核时间到使用时间(time-of-audit-to-time-of-use , TOATTOU) 漏洞，其中因为非原子性，跟踪偏离实际访问，违反精度要求。这允许攻击者掩盖活动，避免触发人侵检测软件(IDS) 。
- 3.替换时间到使用时间(time-of-replacement-to-time-of-use , TORTTOU) 漏洞，对于包装器是唯一的，其中攻击者在包装器替换系统调用参数后，但在此内核访问它们之前修改这些参数，从而违反安全策略。

谢谢大家