西安电子科技大学 XIDIAN UNIVERSITY | 广州研究院 Guangzhou institute of technology

ThreadSanitizer

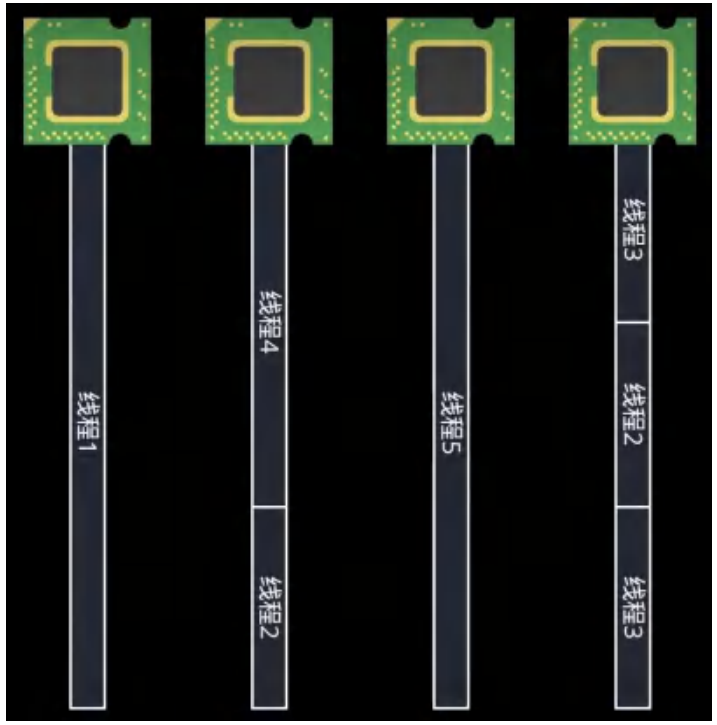
data race detection in practice

汇报人：林志伟

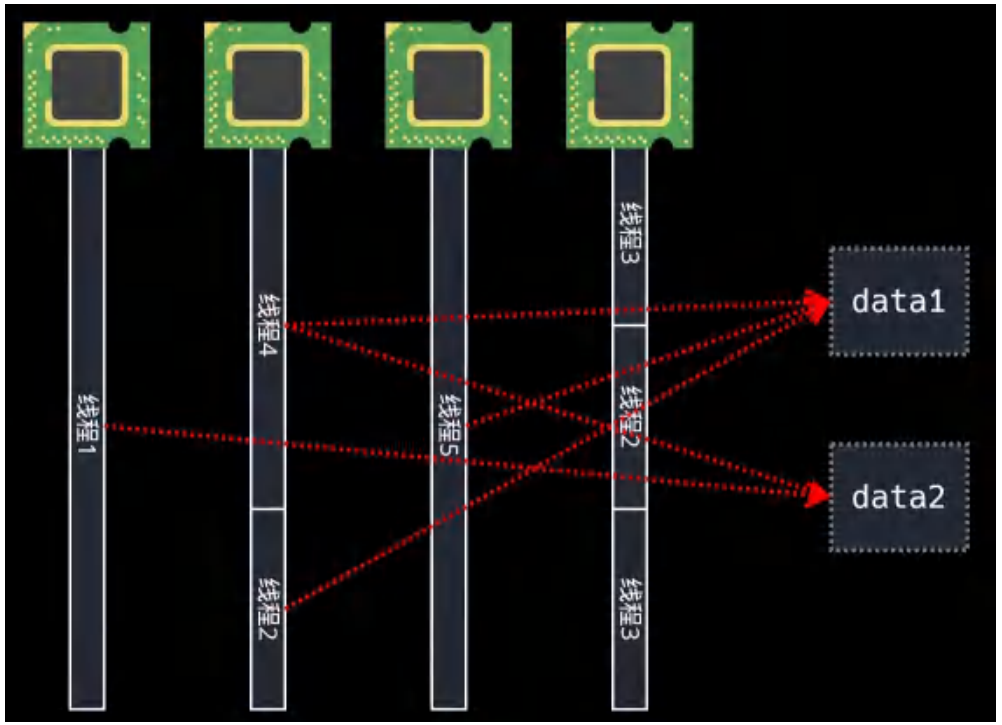
时间：2025.09.18

背景介绍

现代计算机普遍是多核 CPU，为了更好利用硬件性能，程序员会使用多线程来并行执行任务以提高运行效率。



如果多个线程同时访问同一个内存，并且这些访问没有互相协调，就会出现不可预测的行为。



数据竞争：多个线程并发访问同一共享内存地址，且至少其中一个访问操作为写操作的情况。

背景介绍

```cpp
int n = 0; // 全局共享变量

void increase_number() {
    for (int i = 0; i < 1000000; i++) {
        n++;
    }
}

int main() {
    // 创建 10 个线程
    for (int i = 0; i < 10; i++) {
        std::thread(increase_number);
    }

    // ...
}
```

无法得到正确结果10000000，并且每次得到的结果是随机的，结果具有不确定性。

n++ 不是原子操作，这条语句被翻译成机器代码的时候会有三条指令

```
00007FF627F1196F  mov         eax,dword ptr [n (07FF627F21340h)]
00007FF627F11975  inc         eax
00007FF627F11977  mov         dword ptr [n (07FF627F21340h)],eax
```

不发生数据竞争

发生数据竞争

背景介绍

- 数据竞争导致的漏洞往往难以发现，因为它们仅在极为特殊的条件下才会出现，且难以复现。换言之，即便所有测试都顺利通过，也无法保证程序中不存在数据竞争。由于数据竞争可能导致数据损坏或段错误，因此，拥有能定位现有数据竞争、并在新数据竞争随源代码出现时及时捕获的工具至关重要。

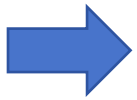
- 早期常用的检测工具是 Valgrind，开启检测会带来 10× 到 50× 的时间开销，同时还伴随极高的内存占用。这种成本在小型实验程序中尚且可以接受，但在复杂的工业级应用或大型测试场景下几乎不可用。

- 为了在工程实践中更高效地发现并发错误，Google 开发了 Sanitizer 系列工具，其中就包括 ThreadSanitizer（TSan），TSan 包含 V1 和 v2 两个版本，现在常用的 v2 版本在真实应用中的典型运行时间开销约为 5×–15×，内存开销为 5×–10×，这使得它能够在更多的真实工程环境中得到实际使用。

TSan - 使用方式

TSan 是一款检测数据竞争的工具，主要由编译器检测模块和运行时库组成，只需要在 gcc/clang 编译选项加上 -fsanitize=thread 就可以开启检查功能，并且具有详细的报错信息。

```
$ cat simple_race.cc
#include <pthread.h>
#include <stdio.h>

int Global;

void *Thread1(void *x) {
  Global++;
  return NULL;
}

void *Thread2(void *x) {
  Global--;
  return NULL;
}

int main() {
  pthread_t t[2];
  pthread_create(&t[0], NULL, Thread1, NULL);
  pthread_create(&t[1], NULL, Thread2, NULL);
  pthread_join(t[0], NULL);
  pthread_join(t[1], NULL);
}
```



```
$ clang++ simple_race.cc -fsanitize=thread -fPIE -pie -g
$ ./a.out
==================
WARNING: ThreadSanitizer: data race (pid=26327)
  Write of size 4 at 0x7f89554701d0 by thread T1:
    #0 Thread1(void*) simple_race.cc:8 (exe+0x000000006e66)

  Previous write of size 4 at 0x7f89554701d0 by thread T2:
    #0 Thread2(void*) simple_race.cc:13 (exe+0x000000006ed6)

  Thread T1 (tid=26328, running) created at:
    #0 pthread_create tsan_interceptors.cc:683 (exe+0x00000001108b)
    #1 main simple_race.cc:19 (exe+0x000000006f39)

  Thread T2 (tid=26329, running) created at:
    #0 pthread_create tsan_interceptors.cc:683 (exe+0x00000001108b)
    #1 main simple_race.cc:20 (exe+0x000000006f63)
==================
ThreadSanitizer: reported 1 warnings
```

TSan - 重要定义

```
void* thread1(void* arg) {
    pthread_mutex_lock(&m);
    x = 1;                  // 写 W1
    ready = 1;
    pthread_cond_signal(&cond); // 同步事件: signal
    pthread_mutex_unlock(&m);
    return NULL;
}

void* thread2(void* arg) {
    pthread_mutex_lock(&m);
    while (!ready) {
        pthread_cond_wait(&cond, &m); // 同步事件: wait
    }
    int y = x;              // 读 R2
    pthread_mutex_unlock(&m);
    printf("y = %d\n", y);
    return NULL;
}
```

事件（E）：程序中发生的一系列操作

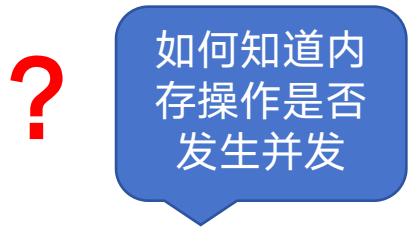
段（S）：线程里不包含同步事件的一段连续内存访问，是TSan的执行单元

锁集合（lockset）：在某个段中，线程持有的锁的集合

先行发生（happens-before）：一个线程PA发送了消息，而另一个线程PB需要等待接收消息才能继续执行，这种先后关系称为先行发生关系。(记作 PA ≺PB)

TSan - 并发识别

数据竞争：多个线程并发访问同一共享内存地址，且至少其中一个访问操作为写操作的情况。



三种识别并发的算法：

happens-before (HB)：只依赖同步（锁/信号/线程创建等）建立的因果关系来判断是否并发——如果能捕获所有同步，误报几乎没有，但会漏报许多未在当前执行中显现出来的竞态。

lockset（Eraser）：跟踪每次访问时持有的锁的交集；一旦交集变空就报告潜在竞态——召回高，但会产生很多假阳性（因为有些“同步”不是显式锁，或初始化阶段出现无锁访问）。

Hybrid（TSan）：把 HB 用来排除因果已同步的情况，用 lockset（或锁集合信息）增强对没有 HB 保护的并发访问的检测——兼顾低误报和高召回，并通过注解机制处理自定义同步。

Lockset

两个访问的锁集合的交集为空，那么这两个访问为并发情况

对每个变量维护一个“保护该变量的锁集合”，每次线程访问变量时，把当前线程持有的锁集合与变量的保护集合取交集。若交集最终变为 ∅，说明没有单一锁能保护所有访问，可能存在并发数据竞争，从而报告错误。

```
void* t1(void* arg) {
    pthread_mutex_lock(&m);
    x = 1;    // W1, lockset={m}
    pthread_mutex_unlock(&m);
}
void* t2(void* arg) {
    pthread_mutex_lock(&m);
    x = 2;    // W2, lockset={m}
    pthread_mutex_unlock(&m);
}
```

Lockset(W1) = {m}, Lockset(W2) = {m}

Lockset(W1) && Lockset(W2) = { m}

访问被同个锁保护，不会发生数据竞争

```
int x = 0;

void* t1(void* arg) { x = 1; }   // lockset=∅
void* t2(void* arg) { x = 2; }   // lockset=∅
```

Lockset(W1) = {∅}, Lockset(W2) = {∅}

Lockset(W1) && Lockset(W2) = { ∅}

访问没有被同个锁保护，会发生数据竞争，报错

优点：能发现很多潜在的竞态

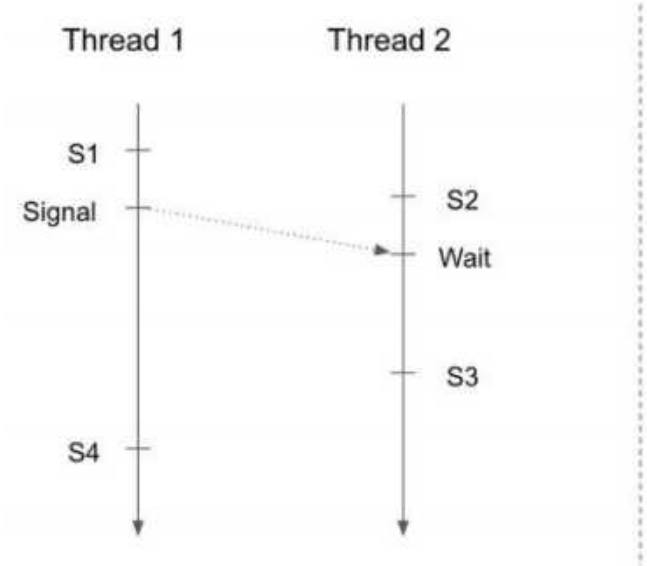
缺点：误报多，仅依靠锁集合交集无法容易把很多合法情况判定为竞态

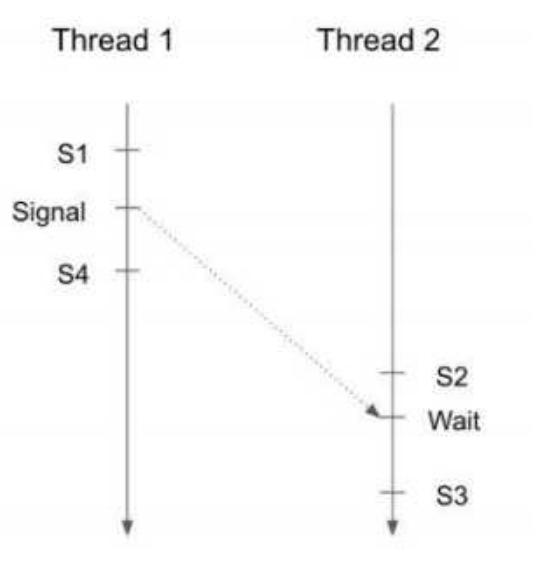
```
void* thread1(void* arg) {
    x = 42;  // 初始化时没有锁
    return NULL;
}

void* thread2(void* arg) {
    pthread_mutex_lock(&m);
    printf("%d\n", x);   // 读
    pthread_mutex_unlock(&m);
    return NULL;
}
```

Happens-before

两个访问互相不为 HB 关系，那么这两个访问为并发情况

记录同步事件并传播逻辑时钟（Lamport/向量时钟，用于表示执行顺序），当访问 A 的逻辑时钟 < 访问 B 的逻辑时钟时，就认为 A happens-before B；只有当两个访问不可比较（既不 A < B 也不 B < A）时才可能是并发。





S1 < S4：在同个线程内按顺序执行

S1 < S3：T1 和 T2 存在同步信号

S1、S2： 无法判断HB关系，为并发

优点：误报非常少，只在没有因果关系、且出现了真正并发访问时才报告

缺点：漏报多，依赖对所有同步的正确捕获，特定调度下触发的并发无法发现

Hybrid

TSan采用混合方法，结合了Lockset和HB，减少了误报和漏报

用 HB 来过滤那些因果上已同步的访问（这一类是真正安全的，不要报），对剩下的“仍可能并发”的访问，检查锁集合（lockset）进一步判断同步。

既避免了 lockset 的许多误报（通过 HB 排除已同步的情况），也补充了纯 HB 的漏报（通过 lockset 找到那些没有 HB 但确实潜在的并发访问）

```
HANDLE-READ-OR-WRITE-EVENT(IsWrite, Tid, ID)
 1  ▷ Handle event READ Tid(ID) or WRITE Tid(ID)
 2  (SSWr, SSRd) ← GET-PER-ID-STATE(ID)
 3  Seg ← GET-CURRENT-SEGMENT(Tid)
 4  if IsWrite
 5    then ▷ WRITE event: update SSWr and SSRd
 6         SSRd ← {s : s ∈ SSRd ∧ s ⊀ Seg}
 7         SSWr ← {s : s ∈ SSWr ∧ s ⊀ Seg} ⊔ {Seg}
 8    else ▷ READ event: update SSRd
 9         SSRd ← {s : s ∈ SSRd ∧ s ⊀ Seg} ⊔ {Seg}
10  SET-PER-ID-STATE(ID, SSWr, SSRd)
11  if IS-RACE(SSWr, SSRd)
12    then ▷ Report a data race on ID
13         REPORT-RACE(IsWrite, Tid, Seg, ID)
```

Happens-before 伪代码

```
IS-RACE(SSWr, SSRd)
 1  ▷ Check if we have a race.
 2  NW ← SEGMENT-SET-SIZE(SSWr)
 3  for i ← 1 to NW
 4    do W1 ← SSWr[i]
 5       LS1 ← GET-WRITER-LOCK-SET(W1)
 6       ▷ Check all write-write pairs.
 7       for j ← i + 1 to NW
 8         do W2 ← SSWr[j]
 9            LS2 ← GET-WRITER-LOCK-SET(W2)
10            ASSERT(W1 ⊀ W2 and W2 ⊀ W1)
11            if LS1 ∩ LS2 = ∅
12              then return true
13       ▷ Check all write-read pairs.
14       for R ∈ SSRd
15         do LSR ← GET-READER-LOCK-SET(R)
16            if W1 ⊀ R and LS1 ∩ LSR = ∅
17              then return true
18  return false
```

Lockset 伪代码

vector clock

向量时钟可以用于判断HB关系，进一步判断同步关系

在并发系统中，物理时间可能会有不同步、消息延迟等不可预测情况，因此不能靠物理时间判定先后

逻辑时钟给每个事件打“时间戳”，这些时间戳能反映因果关系（如果 A → B 则 timestamp(A) < timestamp(B)）

常见逻辑时钟：

- Lamport 时钟（标量）—— 能保证若 A → B 则 L(A) < L(B)，但无法判断并发
- 向量时钟（vector clock）—— 可以精确判断 A ≺ B、B ≺ A，还是并发

1、系统有 N 个进程。每个进程 p 保持一个长度 N 的向量 VC_p[1..N]（初始全 0）

2、本地事件：VC_p[p] += 1

3、发送消息：先做本地事件（或至少确保 VC_p[p] 增 1），把 VC_p 附到消息里

4、接收消息（附带 msg.VC）：对 i=1..N 做 VC_p[i] = max(VC_p[i], msg.VC[i])，然后 VC_p[p] += 1

5、若 对所有 i: VC(a)[i] ≤ VC(b)[i] 且至少一个严格小 → VC(a) < VC(b)，即 a happens-before b

6、若 VC(a) ⊀ VC(b) 且 VC(b) ⊀ VC(a) → a 与 b 并发

VC1 = [0,0], VC2 = [0,0]

VC1 = [1,0], VC2 = [0,0]

VC1 = [2,0], VC2 = [0,0]

VC1 = [2,0], VC2 = [2,1]

VC1[1] = 2 <= VC2[1]= 2

VC1[2] = 0 < VC2[2]= 1

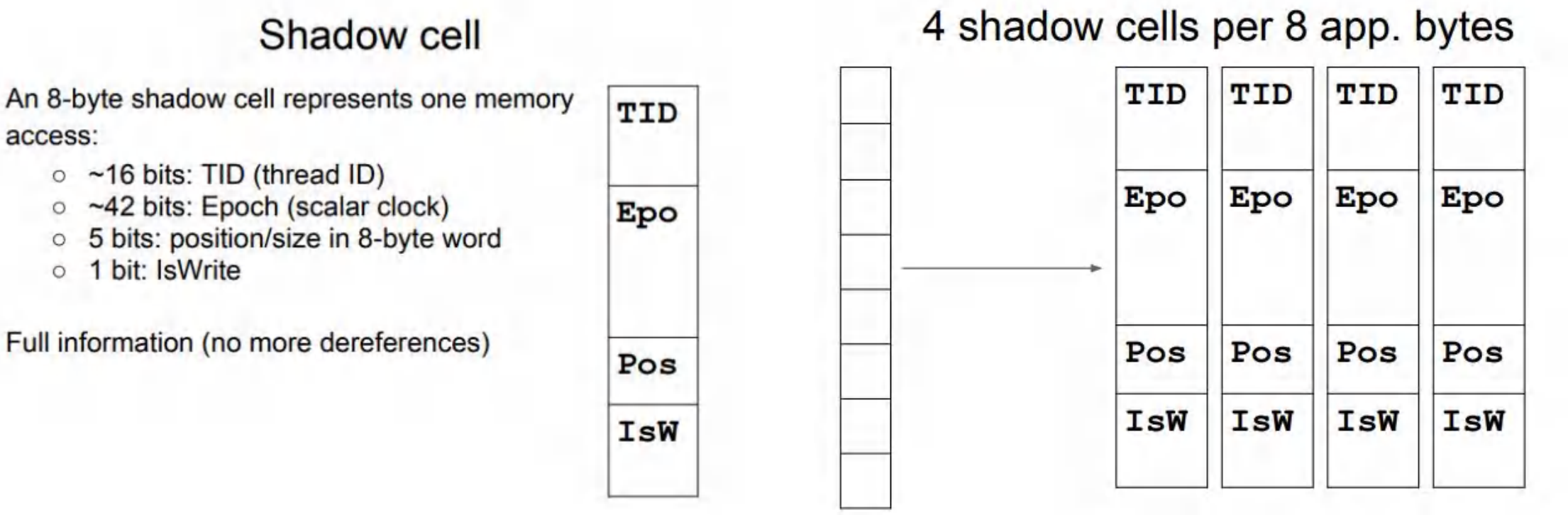
p1 happens-before p2

TSan - LLVM 实现

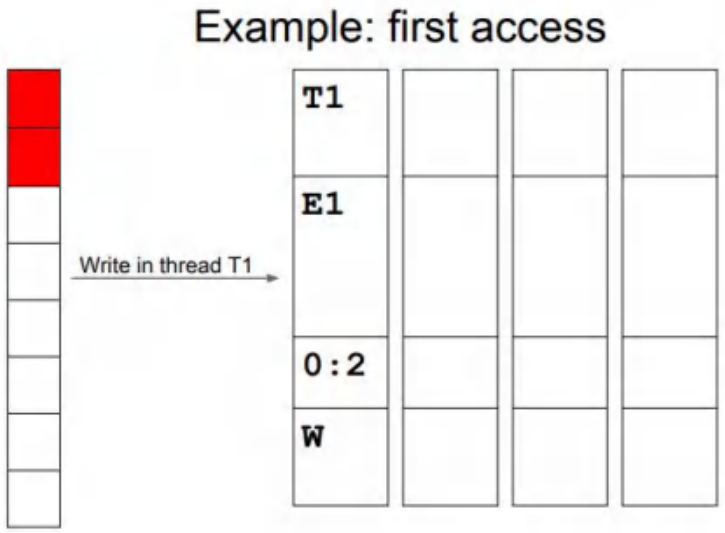
影子内存（Shadow Memory）：是一块与程序真实内存按比例映射的额外区域，用来记录每个字节的访问元信息

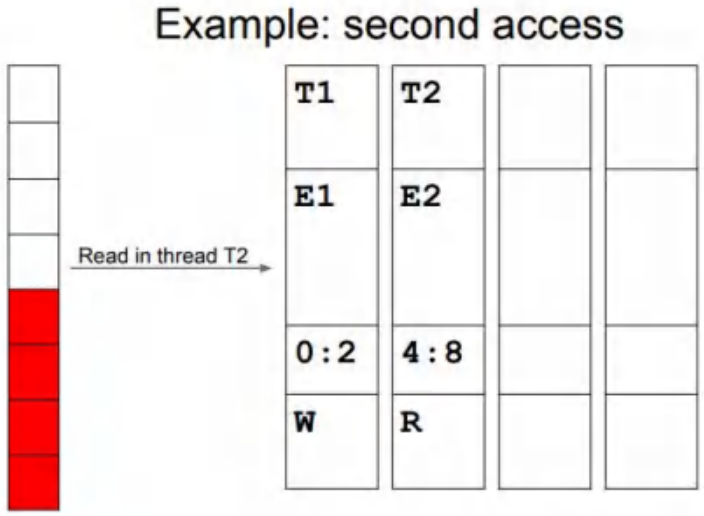
TSan 的一个影子内存单元，大小为8字节，包含一些线程的关键信息

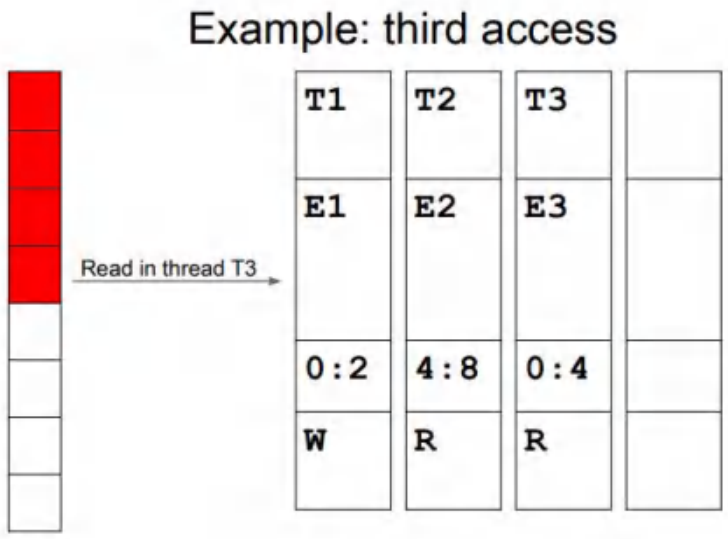
TSan 一个8字节的内存空间对应4个影子内存单元，也就是8个字节需要64字节的影子内存（1:4）

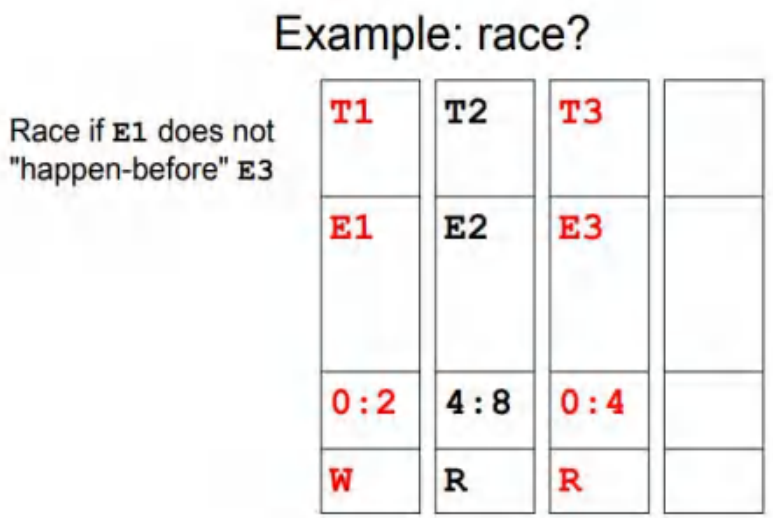


TSan - LLVM 实现


Example: first access
Write in thread T1
T1
E1
0:2
W


Example: second access
Read in thread T2
T1
T2
E1
E2
0:2
4:8
W
R


Example: third access
Read in thread T3
T1
T2
T3
E1
E2
E3
0:2
4:8
0:4
W
R
R


Example: race?
Race if E1 does not "happen-before" E3
T1
T2
T3
E1
E2
E3
0:2
4:8
0:4
W
R
R