Reading 5

Question 1

According to the lockset algorithm, when does eraser signal a data race? Why is this condition chosen?

每一个共享变量都收一个锁保护

1. 一个变量只有一个程序访问的时候状态时virgin
2. 另一个访问但不写的进程叫共享状态，更新lockset，但不警报
3. 访问且写的进程叫共享-修改状态，更新lockset，发出警报警告race
   这个过程伴随着锁集细化，锁集细化到最后就可以让eraser知道一个共享变量对应的lock具体是哪一个

因为在第二个写且访问的进程进入时，才有警报的必要（其他不警报只更新可以节约资源提高速度）

Question 2

Under what conditions does Eraser report a false positive? What conditions does it produce false negatives?

False positive：没有竞争却报告竞争

1.内存重用:报告了错误警报，因为在不重置shadow memory的情况下重用了memory。

2.专用锁定:报告错误警报，因为在运行时未将信息传递给eraser而进行了锁定。 这通常是由多个读取器/单个写入器锁的私有实现引起的，这些实现不属于eraser提供的标准 pthreads 接口。

3.良性竞赛:发现了不影响程序正确性的真实数据竞赛。其中一些是故意的，而其他则是偶然的。

false negative：有竞争却没报
3.4中 唤醒等待 I / O 完成的线程。如果在设备驱动程序和线程之间共享数据，这可能会导致eraser在有竞争的情况下却没有报告。因为信号量不是"所有者"，所以橡皮擦很难推断出它们

正在 保护哪些数据，从而导致发出错误警报。

Quesiton 3

Typically, instrumenting a program changes the intra-thread timing (the paper calls it interleaving). This can cause bugs to disappear when you start trying to find them. What aspect of the Eraser design mitigates this problem?

1. 原有方法：在大多数系统中，将中断级 别提高到 n 可确保只有优先级大于 n 的中断才能得到服务，直到降低中断级别为止。 可以使用升高然后恢复的中断级别代替锁定 ：

   level := SetInterruptLevel(n);

   (* Manipulate data *)

   RestoreInterruptLevel(level);

   这种差异机制被纳入eraser的实现中：为每个不同的中断级别分配锁
   不同等级的锁的设置让不同thread 的调度有了一定的区别，通过中断级别来分配锁

2. 设置一些接口来避免虚假的报错
   不报告良性竞赛的接口：
   eraserignoreon（）
   eraserrigoof（）
   防止内存重用竞赛，添加了：
   eraserreuse（add，size）
   私有锁的实现：
   EraserReadLock(lockj)
   EraserReadUnlock(lock)
   EraserWriteLock(lock)
   EraserWriteUnlock(lock)

Question 4

Please raise at least one question of your own for the discussion.

问题1：根据三种不同服务器上的实验，当eraser调用了相应的接口来避免上报false positive

的错误时，eraser的实施效果就十分好，目前eraser这种lockset的算法还有哪些不足之处？还有哪里需要改进？

问题2：多线程程序中对于eraser仍未实现多重锁定保护可能会引发什么问题？假如没有实现对一个程序多个锁的逐一检查，就没有多重锁定的保护，可能会引起什么问题？或者说实现了多重锁定保护能解决什么问题？

问题3:对于僵局：拿锁较多的程序获得更高的优先级的这个策略应该如何添加或应用到eraser的策略中，添加后对eraser会有什么改进？

文章总结：
多线程编程困难且容易出错

introduction

调试多线程程序可能很困难。
因此可以动态监测多线程编程
锁：两个线程访问一个变量，需要加上锁

related works

1. 监视器（安全性与封闭性）
   缺点：难以高效实现
2. lockset算法
   Eraser监视程序执行的所有读写检查程序是否遵守规则
   每一个共享变量都收一个锁保护
   Eraser不知道锁和变量的对应关系，需要从历史中推断保护关系

每一个共享变量v， eraser维护v的候选锁集合C（v）
这个集合包含目前为止为计算保护了v的锁

锁集细化：初始化时，将其候选视为持所有锁，访问变量的时候，eraser用C（v）与当前线程持有的一组锁的交集来更新C（v）

可以保证任何时候C（v）中都有保护v的锁
若C（v）空，则没有锁保护v

2.1 改进
共享变量经常在不持有锁的情况下进行初始化
某些共享变量仅在初始化期间写入，而在其后为只读。无需锁即

可安全地访问它们。
读写锁允许多个读取器访问共享变量，但仅允许一个写入器进行访问。
2.2 初始化与读取共享
橡皮擦认为共享变量在第二
个线程首次访问时要初始化。
只要仅由单个线程访问变量，读和写就不会影响候选集。

独占状态： 第一个线程访问时的状态，C（v）不变
共享状态：第二个线程访问，C（v）被更新但是如果此时C（v）为空，也不会警报
共享修改状态：第二个线程访问， C（v）被更新 报告竞争
2.3读写锁
每个变量v 都有一些锁m保护v，对于v的每一次写入，m处于写模式
locks_held（t）为线程t在任何模式下持有锁的集合
write_locks_held（t）的写锁为线程t在写模式下保持的锁的集合
对于每个v，将C（v）初始化为所有锁的集合
在线程t每次读取v的时候，
设置C（v）=C（v）∩locks held(t)
如果C（v）是空，发出警告
在t对v的每一次写入中，
设置C（v）=C（v）∩write_locks held(t)
如果C（v）是空，发出警告

发生写操作时，将纯保留在读取模式下的锁从候选集中删除，因为由写
程序持有的此类锁无法防止写程序与某些其他读程序线程之间的数据竞争。

3. 实施橡皮擦
3.1 代表锁候选集
事实上 不同的锁组很少，用一个整数表示这些锁组
3.2 性能
速度降低10-30倍
橡皮擦通过将变量的地址添加到固定的影子内存偏移量来将锁集索引与每个变量关联。索引又从 锁集索引表中选择一个锁向量。在这种情况下，共享变量 v 与包含 mu1 和 mu2 的一组锁相关联。
3.2 程序注释
三类错误警报：
1.内存重用:报告了错误警报，因为在不重置影子内存的情况下重用了内存。
2.专用锁定:报告错误警报，因为在运行时未将信息传递给橡皮擦而进行了锁定。 这通常是由多个读取器/单个写入器锁的私有实现引起的，这些实现不属于橡皮 擦提供的标准 pthreads 接口。

3.良性竞赛:发现了不影响程序正确性的真实数据竞赛。其中一些是故意的，而其他则是偶然的。

改进：

不报告良性竞赛的接口：

eraserignoreon（）

eraserrigoof（）

防止内存重用竞赛，添加了：

eraserreuse（add，size）

私有锁的实现：

EraserReadLock(锁定)

EraserReadUnlock(锁定)

EraserWriteLock(锁定)

EraserWriteUnlock(锁定)

3.4内核中的race检测

在大多数系统中，将中断级 别提高到 n 可确保只有优先级大于 n 的中断才能得到服务，直到降低中断级别为止。 可以使用升高然后恢复的中断级别代替锁定

level := SetInterruptLevel(n);

(* Manipulate data *)

RestoreInterruptLevel(level);

唤醒等待 I / O 完成的线程。如果在设备驱动程序和线程之间共享数据，这可能会导 致橡皮擦出现问题。因为信号量不是“所有者”，所以橡皮擦很难推断出它们正在 保护哪些数据，从而导致发出错误警报。