libvirt sanlock工作机制与代码分析

基于3.9.5版本sanlock和8.0.0版本libvirt

工作机制

基本原理

sanlock 是一款构建于共享存储之上的锁管理器。有权访问该存储的主机可以执行锁定操作。在这些主机上运行的应用程序会在共享块设备或文件上被分配一小部分空间，并使用 sanlock 来进行其特定于自身应用的同步操作。从内部来看，sanlock 守护进程使用两种基于磁盘的租约算法来管理锁：Delta租约算法和 Paxos 租约算法。

Delta 租约的获取速度较慢，且需要定期对共享存储进行输入/输出操作。Sanlock 仅在内部使用它们来持有其“host_id”（一个从 1 到 2000 的整数主机标识符）的租约。这些租约用于防止两台主机使用相同的主机标识符。Delta 租约的续订还可以表明主机是否处于活动状态。

Paxos 租约的获取速度很快，Sanlock 将它们作为通用资源租约提供给应用程序使用。磁盘 Paxos 算法在内部使用 host_id 来表示不同的主机以及 Paxos 租约的持有者。Delta 租约提供唯一的 host_id 以实现 Paxos 租约，而 Delta 租约的续订则作为 Paxos 租约续订的代理。

在外部，sanlock 守护进程通过 libsanlock 提供了一个锁定接口，该接口使用“锁空间（lockspaces）”和“资源（resources）”的概念。锁空间是应用程序在共享存储上为自己创建的一个锁定上下文。当每台主机上的应用程序启动时，它会“加入”该锁空间。然后，它可以在共享存储上创建“资源”。每个资源代表一个应用程序特定的实体。应用程序可以获取和释放资源上的租约。

要从应用程序中使用 sanlock，执行以下步骤：

- 为应用程序分配共享存储，例如来自 SAN 的共享 LUN 或 LV，或者来自 NFS 的文件。
- 将存储提供给应用程序。
- 应用程序使用此存储和 libsanlock 为自己创建锁空间和资源。
- 应用程序在启动时加入锁空间。
- 应用程序获取和释放资源上的租约。

在 sanlock 中，锁空间和资源如何转换为 delta 租约和 paxos 租约：

锁空间

- 锁空间基于每个使用锁空间的主机所持有的 delta 租约。
- 锁空间是磁盘上的一系列 2000 个 delta 租约，需要 1MB 的存储空间。（有关大小变化，请参见下面的“存储”部分。）
- 锁空间可以支持最多 2000 个并发主机同时使用，每个主机使用不同的 delta 租约。
- 应用程序可以 i) 创建、ii) 加入和 iii) 离开锁空间，这分别对应于 i) 在磁盘上初始化 delta 租约集、ii) 获取其中一个 delta 租约和 iii) 释放 delta 租约。
- 创建锁空间时，应用程序会提供唯一的锁空间名称和磁盘位置。
- 创建/初始化锁空间时，sanlock 会在文件或磁盘上格式化 2000 个磁盘上 delta 租约结构的序列，例如 /mnt/leasefile（NFS）或 /dev/vg/lv（SAN）。
- 锁空间中的 2000 个单独 delta 租约通过编号进行标识：1,2,3,...,2000。
- 每个 delta 租约是 1MB 锁空间中的一个 512 字节扇区，根据其编号进行偏移，例如，delta 租约 1 的偏移量为 0，delta 租约 2 的偏移量为 512，delta 租约 2000 的偏移量为 1023488。（有关大小变化，请参见下面的“存储”部分。）
- 当应用程序加入锁空间时，必须指定锁空间名称、共享磁盘/文件上的锁空间位置以及本地主机的 host_id。然后，sanlock 会获取与 host_id 对应的 delta 租约，例如，以 host_id 1 加入锁空间会获取 delta 租约 1。
- delta 租约、锁空间租约和 host_id 租约这几个术语可互换使用。
- sanlock 通过将主机的唯一名称写入 delta 租约磁盘扇区，延迟后读回并验证其相同性来获取 delta 租约。
- 如果未指定唯一主机名，sanlock 会使用 product_uuid（如果可用），否则会生成一个 uuid 作为主机的名称。delta 租约算法依赖于主机使用唯一名称。

- 每台主机上的应用程序应配置有唯一的 host_id，其中 host_id 是 1-2000 之间的整数。
- 如果主机配置错误且具有相同的 host_id，delta 租约算法旨在检测此冲突，并且只有一个主机能够获取该 host_id 的 delta 租约。
- delta 租约确保锁空间 host_id 正由在 delta 租约中指定的具有唯一名称的单个主机使用。
- 解决 delta 租约冲突的速度较慢，因为该算法基于等待和观察一段时间以查看其他主机是否写入相同的 delta 租约扇区。如果多个主机尝试使用相同的 delta 租约，延迟会大幅增加。因此，最好配置应用程序使用不会冲突的唯一 host_id。
- sanlock 获取 delta 租约后，必须续订租约，直到应用程序离开锁空间（这对应于释放 host_id 上的 delta 租约）。
- sanlock 默认每 20 秒续订一次 delta 租约，方法是在 delta 租约扇区中写入新的时间戳。
- 当主机在锁空间中获取 delta 租约时，可以称之为“加入”锁空间。一旦加入锁空间，它就可以使用与锁空间关联的资源。

资源

- 锁空间（lockspace）是应用程序可以锁定和解锁资源的环境。
- sanlock 使用 Paxos 租约来实现对资源的租约。Paxos 租约和资源租约这两个术语可互换使用。
- Paxos 租约存在于共享存储上，需要 1MB 的空间。它包含一个唯一的资源名称和锁空间的名称。
- 应用程序为其 sanlock 资源和其上的租约赋予自己的含义。sanlock 资源可以表示一些共享对象（如文件），或者主机之间的某个唯一角色。
- 资源租约与特定的锁空间相关联，并且只能由已加入该锁空间的主机使用（即这些主机在该锁空间中持有 host_id 的Delta租约）。
- 应用程序必须跟踪其锁空间和资源的磁盘位置。sanlock 不维护应用程序创建的锁空间或资源的任何持久索引或目录，因此应用程序需要记住自己放置租约的位置（即哪些文件或磁盘以及偏移量）。
- sanlock 不直接续订 Paxos 租约（尽管它可以这样做）。相反，主机Delta租约的续订代表了对该主机在相关锁空间中所有 Paxos 租约的续订。实际上，许多 Paxos 租约续订被合并为一个Delta租约续订，从而在使用多个 Paxos 租约时减少了 I/O。
- 磁盘 Paxos 算法允许多个主机同时尝试获取同一个 Paxos 租约，并将产生一个资源租约的唯一获胜者/所有者。（除了默认的独占租约外，还可以实现共享资源租约。）
- 磁盘 Paxos 算法涉及对 Paxos 租约磁盘区域的扇区进行特定的读写序列。每个主机在 Paxos 租约磁盘区域中有一个专用的 512 字节扇区，用于写入自己的“选票”，并且每个主机读取整个磁盘区域以查看其他主机的选票。磁盘区域的第一个扇区是“领导者记录”，它保存最后一次 Paxos 投票的结果。Paxos 投票的获胜者将投票结果写入领导者记录（投票的获胜者可能选择另一个竞争主机作为 Paxos 租约的所有者）。
- 获取 Paxos 租约后，不再在 Paxos 租约磁盘区域中进行 I/O 操作。
- 释放 Paxos 租约涉及写入一个扇区以清除领导者记录中的当前所有者。
- 如果持有 Paxos 租约的主机发生故障，Paxos 租约的磁盘区域仍指示该租约由故障主机拥有。如果另一个主机尝试获取该 Paxos 租约，并发现租约由另一个 host_id 持有，它将检查该 host_id 的Delta租约。如果该 host_id 的Delta租约正在续订，则 Paxos 租约已被拥有且无法获取。如果该所有者 host_id 的Delta租约已过期，则 Paxos 租约也已过期，可以被获取（通过执行 Paxos 租约算法）。
- 主机之间的“交互”或“感知”仅限于它们尝试获取同一个 Paxos 租约的情况，并且需要检查所引用的Delta租约是否已过期。
- 当主机不尝试同时锁定相同资源时，它们之间没有交互或感知。一个主机的状态或操作对其他主机没有影响。
- 为了加快在 Paxos 租约冲突时检查Delta租约过期的速度，sanlock 会跟踪锁空间中其他Delta租约的过去续订情况。

资源索引

资源索引（rindex）是sanlock的一个可选功能，应用程序可以利用它来跟踪资源租约的偏移量。如果没有资源索引，应用程序必须自行跟踪其资源租约在磁盘上的位置，并在创建新租约时找到可用位置。

sanlock的资源索引在锁空间之后在磁盘上使用两个对齐大小区域。第一个区域存放资源索引条目，每个条目记录一个资源租约的名称和位置。第二个区域存放一个私有的Paxos租约，由sanlock内部使用以保护资源索引的更新。

应用程序通过“format（格式化）”函数在磁盘上创建资源索引。格式化是一个仅涉及磁盘的操作，不与活动的锁空间交互，因此可以在不先调用add_lockspace的情况下调用它。应用程序需要遵循将锁空间写入设备开头（偏移量为0）的约定，并紧接着锁空间区域格式化资源索引。在格式化时，应用程序必须设置与锁空间相匹配的扇区大小和对齐大小的标志。

要使用资源索引，应用程序需要：

- 使用“create（创建）”函数在磁盘上创建一个新的资源租约。这取代了write_resource函数。create函数需要资源索引的位置和新资源租约的名称。sanlock会找到一个空闲的租约区域，在该位置写入新的资源租约，更新资源索引中的名称:偏移量，并将偏移量返回给调用者。调用者在获取资源租约时使用此偏移量。

- 使用“delete（删除）”函数删除磁盘上的资源租约（这也对应于write_resource函数）。sanlock会清除资源租约以及对应的资源索引条目。后续的create调用可能会使用此相同的磁盘位置为不同的资源租约。
- 使用“lookup（查找）”函数根据资源租约的名称发现资源租约的偏移量。调用者通常会在获取资源租约之前调用此函数。
- 使用“rebuild（重建）”函数在资源索引损坏或变得不一致时重新创建它。此函数会扫描磁盘以查找资源租约，并创建新的资源索引条目以匹配它找到的租约。
- “update（更新）”函数直接操作资源索引条目，通常不应由应用程序使用。在正常使用中，create和delete函数会操作资源索引条目。update主要用于测试或修复。

租约所有权

对于主机ID为N+1的扇区N上的delta_lease，其leader_record.owner_id为N+1。每次获取delta_lease时，leader_record.owner_generation都会递增。当获取delta_lease时，leader_record.timestamp字段被设置为主机的当前时间，而leader_record.resource_name被设置为主机的唯一名称。当主机续订delta_lease时，它会写入新的leader_record.timestamp。当主机释放delta_lease时，它会将leader_record.timestamp写为零。

当主机获取paxos_lease时，它使用其在锁空间中持有的delta_lease中的host_id/generation值。在运行paxos算法时，它使用这个host_id/generation在paxos_dblock中标识自己。算法的结果是获胜的host_id/generation，即paxos_lease的新所有者。获胜的host_id/generation被写入paxos_lease的leader_record.owner_id和leader_record.owner_generation字段，并设置leader_record.timestamp。当主机释放paxos_lease时，它将leader_record.timestamp设置为0。

当paxos_lease处于空闲状态（leader_record.timestamp为0）时，多个主机可能会尝试获取它。使用paxos_dblock结构的paxos算法将仅选择其中一个主机作为新所有者，并将该所有者写入leader_record。此时，paxos_lease将不再空闲（timestamp非零）。其他主机将看到这一点，并且在paxos_lease再次空闲之前不会尝试获取它。

如果paxos_lease已被拥有（timestamp非零），但所有者未在特定时间内续订其delta_lease，则paxos_lease中的所有者值将过期，其他主机将使用paxos算法获取paxos_lease，并设置新的所有者。

租约过期

- 如果主机未能续订其Delta租约（例如，失去了对存储的访问权限），其Delta租约最终将过期，另一台主机将能够接管该主机持有的任何资源租约。sanlock必须确保两个不同主机上的应用程序不会同时持有和使用相同的租约。
- 当sanlock在一段时间内未能续订Delta租约时，它将开始采取措施阻止本地进程（应用程序）使用与即将过期的锁空间Delta租约相关联的任何资源租约。sanlock会在另一台主机可能接管本地持有的租约之前很久就进入这种“恢复模式”。sanlock必须有足够的时间来停止所有正在使用即将过期租约的本地进程。
- sanlock使用三种方法来停止正在使用即将过期租约的本地进程：

1. 正常关闭。sanlock将执行应用程序之前为此情况指定的“正常关闭”程序。关闭程序会告知应用程序因其租约即将到期而关闭。应用程序必须通过停止其活动并释放其租约（或退出）来响应。如果应用程序未指定正常关闭程序，sanlock将向进程发送SIGTERM信号。进程必须在规定的时间内（见-g选项）释放其租约或退出，否则sanlock将采用下一种停止方法。
2. 强制关闭。sanlock将向使用即将过期租约的进程发送SIGKILL信号。这些进程在收到SIGKILL信号后有固定的时间来退出。如果有任何进程未在此时间内退出，sanlock将采用下一种方法。
3. 主机重置。sanlock将触发主机的看门狗设备以强制重置它。sanlock会仔细管理看门狗设备的定时，以确保它在任何其他主机可能接管本地进程持有的资源租约之前不久触发。

租约失败

如果一个持有资源租约的进程失败或退出而没有释放其租约，sanlock 将自动为其释放租约（除非使用了持久性资源租约）。

如果 sanlock 守护进程无法在特定时间段内续订锁空间的Delta租约（参见“租约过期”部分），sanlock 将进入“恢复模式”，在此模式下，它会尝试停止和/或终止持有即将过期锁空间中资源租约的任何进程。如果这些进程未能及时退出，sanlock 将使用本地看门狗设备强制重置主机。

如果 sanlock 守护进程崩溃或挂起，它将无法续订其与 wdmd 守护进程之间每个锁空间的连接过期时间。这将导致本地看门狗设备过期，主机将被重置。

看门狗

Sanlock 使用 `wdmd(8)` 守护进程来访问 `/dev/watchdog` 设备。`wdmd` 将多个超时复用到单个看门狗定时器上。这是必要的，因为每个锁空间的 delta 租约是独立续订和过期的。Sanlock 为每个正在续订的锁空间 delta 租约维护一个 `wdmd` 连接。每个连接都有一个在未来几秒内的

过期时间。每次成功续订 delta 租约后，相关联的 `wdmd` 连接的过期时间也会被更新。如果 `wdmd` 发现任何连接已过期，它将不会续订 `/dev/watchdog` 定时器。如果连续续订失败次数足够多，看门狗设备将触发并重置主机。（鉴于 `wdmd` 的复用特性，来自多个锁空间的短暂重叠续订失败可能会导致看门狗误触发。）

Delta 租约续订和看门狗续订之间的直接联系提供了一种可预测的看门狗触发时间，这是基于其他主机可见的 delta 租约续订时间戳。Sanlock 能够根据 delta 租约时间知道另一台主机上的看门狗何时已经触发。此外，如果另一台主机上的看门狗设备未能按预期触发，该主机继续进行的 delta 租约续订将使这一点变得明显，并防止从故障主机获取租约。

如果 Sanlock 能够停止/终止所有使用即将过期的锁空间的进程，那么与该锁空间相关联的 `wdmd` 连接将被移除。已过期的 `wdmd` 连接将不再阻止 `/dev/watchdog` 的续订，主机应避免被重置。

wdmd

这个守护进程打开/dev/watchdog设备，并允许多个独立源来决定每个KEEPALIVE是否完成。在每个测试间隔（默认10秒）内，守护进程会测试每个源。如果任何测试失败，则不执行KEEPALIVE。在默认配置中，如果60秒内没有执行KEEPALIVE，看门狗定时器将重置系统（“触发超时”）。这意味着，如果单个测试连续失败5-6次，看门狗将触发并重置系统。在存在多个测试源的情况下，连续较少的单独失败也可能导致重置，例如：

T秒，P通过，F失败
T00: 测试1 P，测试2 P，测试3 P：执行KEEPALIVE
T10: 测试1 F，测试2 F，测试3 P：跳过KEEPALIVE
T20: 测试1 F，测试2 P，测试3 P：跳过KEEPALIVE
T30: 测试1 P，测试2 F，测试3 P：跳过KEEPALIVE
T40: 测试1 P，测试2 P，测试3 F：跳过KEEPALIVE
T50: 测试1 F，测试2 F，测试3 P：跳过KEEPALIVE
T60: 测试1 P，测试2 F，测试3 P：跳过KEEPALIVE
T60: 看门狗触发，系统重置

（根据时间安排，系统可能会在T60之前的某个时间点重置，因此T60时的测试不会运行。）

测试源：客户端
使用libwdmd库，程序通过Unix套接字连接到wdmd，并向wdmd发送定期消息以更新其连接的过期时间。在每个测试间隔，wdmd会检查连接的过期时间是否已到。如果是，则该客户端的测试失败。

测试源：脚本
wdmd会在每个测试间隔从指定目录运行脚本。如果脚本以0退出，则测试被视为成功，否则为失败。如果脚本在测试间隔结束时仍未退出，则被视为失败。

wdmd设计和功能的一个关键方面是，如果任何一个源在触发超时的时长内未能通过测试，看门狗保证会触发，无论系统上的其他源是通过还是失败。如上所示，由于多个测试失败的综合影响而导致的意外重置是一个可接受的副作用。

如果未加载其他看门狗模块，wdmd的init脚本将加载softdog模块。

wdmd不能与系统上需要打开/dev/watchdog的其他任何程序（例如watchdog(8)）一起使用。

可配置的看门狗超时

看门狗设备通常具有60秒的超时时间，但某些设备的超时时间是可配置的。要使用不同的看门狗超时时间，请在sanlock.conf中将watchdog_fire_timeout（以秒为单位）设置为设备支持的值。所有主机上必须配置相同的watchdog_fire_timeout（因此，所有主机的看门狗设备都必须支持相同的超时时间）。不匹配的值将使看门狗提供的租约保护失效。

通常应一起配置watchdog_fire_timeout和io_timeout。默认情况下，sanlock使用watchdog_fire_timeout=60和io_timeout=10。其他可考虑的组合包括：
watchdog_fire_timeout=30与io_timeout=5
watchdog_fire_timeout=10与io_timeout=2

较小的值会增加主机在等待缓慢的I/O操作完成或临时I/O故障解决时被看门狗重置的可能性。同时，由于独立的、重叠的锁空间故障（每个故障本身都无关紧要），看门狗意外重置的可能性也会增加。

存储

在创建锁空间、资源（和资源索引）时，应指定扇区大小和对齐大小。“对齐大小”是锁空间或资源在磁盘上的大小，即其占用的磁盘空间量。锁空间和资源应使用匹配的扇区和对齐大小，并且偏移量必须是对齐大小的倍数。能够使用某个锁空间或资源的最大主机数取决于扇区大小和对齐大小的组合，如下所示。使用锁空间的主机的主机ID不能大于该锁空间的最大主机数（max_hosts）值。

接受的扇区大小和对齐大小组合，以及相应的最大主机数（和最大主机ID）为：

扇区大小512，对齐大小1M，最大主机数2000
扇区大小4096，对齐大小1M，最大主机数250
扇区大小4096，对齐大小2M，最大主机数500
扇区大小4096，对齐大小4M，最大主机数1000
扇区大小4096，对齐大小8M，最大主机数2000

当未指定扇区大小和对齐大小时，行为将与这些大小可配置之前的行为相匹配：在报告扇区大小为512的设备上，使用512/1M/2000；在报告扇区大小为4096的设备上，使用4096/8M/2000；在文件上，始终使用512/1M/2000。（其他组合与sanlock 3.6或更早版本不兼容。）

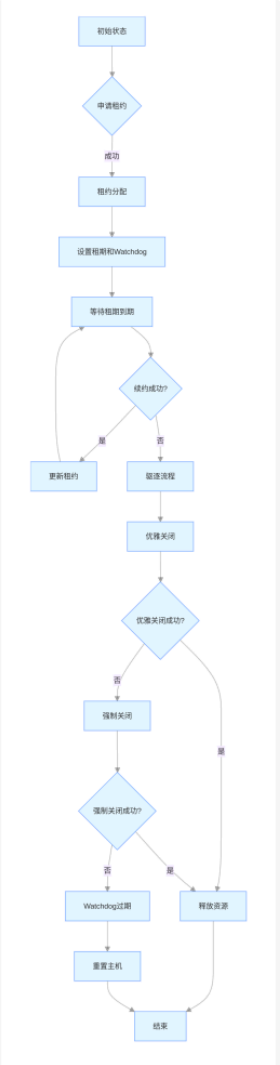
在基于主机进行镜像或复制的共享块设备上使用sanlock可能无法正常工作。在共享文件上使用sanlock时，所有sanlock I/O应定向到一个文件服务器。

机制理解

租房理论

sanlock是房东，锁空间是一栋楼，资源是楼里每个出租的房子，租约是每个房子的出租合同，应用进程是租户。租户向房东租房子（应用程序向sanlock获取租约），房东保证房子最终租给一个租户（sanlock保证最终一个应用程序获取租约，sanlock还支持共享租约，即“合租”）并确定租期（租约TTL）和租期到期前多少天续约（续约周期），同时给了租户房子钥匙并确定有效期（启动watchdog并设置过期时间）。到了续约时间，如果续约成功，房东更新租期（租约TTL）并延长房子钥匙有效期（更新watchdog过期时间）；如果没有续约（续约失败），房东通知租户在几天内搬走（sanlock给应用进程发送优雅关闭信号，并等待优雅关闭超时时间）。如果租户按期搬走并解除合同（应用进程优雅关闭并释放租约成功），房东收回房子钥匙（关闭watchdog），可以将房子租给其他租户（其他应用进程获取租约）；如果租户没在规定时间内搬走（应用进程优雅关闭并释放租约超时），房东上门强制要求租户必须在房子钥匙有效期内搬走（sanlock给应用进程发送kill信号，强制关闭应用进程，进程需要在watchdog过期前退出）。如果租户在房子钥匙有效期内搬走（应用进程在watchdog过期前成功退出），房东收回房子钥匙（sanlock释放租约并关闭watchdog），可以将房子租给其他租户（其他应用程序获取租约）；如果租户没在房子钥匙有效期内搬走（应用程序在watchdog过期前未成功退出），直接更换房子门锁（watchdog重置主机），可以将房子租给其他租户（其他应用程序获取租约）。

工作流程图



工作时序

sanlock的工作机制中涉及很多时间元素，如续约周期，watchdog超时时间等，这些时间元素之间有严格的时序关系。下面结合工作场景，对工作时序进行理解。

工作场景：续约失败，sanlock杀掉应用进程失败

---

wdmd测试间隔默认10s（每10s检查wdmd连接是否过期）

watchdog超时时间默认60s（当前时间距上次喂狗时间达到60s）

io超时时间默认10s

续约周期默认20s

续约失败容忍时间默认80s（当前时间-上次续约成功时的时间超过80s则关闭应用进程）

续约告警容忍时间默认60s（当前时间-上次续约成功时的时间超过60s则告警）

主机dead容忍时间默认140s（续约失败容忍时间+watchdog过期时间）

优雅关闭超时时间默认40s

0s：sanlock续约成功，sanlock调用wdmd检查wdmd连接是否过期（过期时间是当前时间0s+续约失败容忍时间80s），wdmd检查0<80，没过期，喂狗

10s：wdmd检查10<80，没过期，喂狗

20s：sanlock续约成功，sanlock调用wdmd检查wdmd连接是否过期（过期时间是当前时间20s+续约失败容忍时间80s），wdmd检查20<100 or 80（如果wdmd检查在sanlock调用之前是80，在sanlock调用之后是100，下同），没过期，喂狗

30s：wdmd检查30<100，没过期，喂狗

40s：sanlock续约成功，sanlock调用wdmd检查wdmd连接是否过期（过期时间是当前时间40s+续约失败容忍时间80s），wdmd检查40<120 or 100，没过期，喂狗

50s：wdmd检查50<120，没过期，喂狗

60s：sanlock续约失败，wdmd检查60<120，没过期，喂狗

70s：wdmd检查70<120，没过期，喂狗

80s：wdmd检查80<120，没过期，喂狗

90s：wdmd检查90<120，没过期，喂狗

100s：wdmd检查100<120，没过期，喂狗；sanlock输出续约告警信息（距上次续约成功的40s时间点过了60s）

110s：wdmd检查110<120，没过期，喂狗；wdmd输出watchdog closed unclean和检查告警信息

120s：达到续约失败容忍时间（距上次续约成功的40s时间点过了80s），sanlock开始优雅关闭应用进程；wdmd检查120>=120，wdmd连接过期，停止喂狗；sanlock输出续约失败信息；wdmd输出检查失败信息。

- watchdog将在上次喂狗的时间110s+watchdog超时时间60s=170s时超时
- 租约将在上次续约成功时间40s+续约失败容忍时间80s+watchdog超时时间60s（或上次续约成功时间40s+主机dead容忍时间140s）=180s时过期
- wdmd可能在119s时检查没过期并喂狗，所以watchdog超时时间可以到179s；从而watchdog可能在170s，也可能在179s超时，但租约还是到180s过期

130s：wdmd检查130>120，过期，不喂狗

140s：wdmd检查140>120，过期，不喂狗

150s：wdmd检查150>120，过期，不喂狗

160s：wdmd检查160>120，过期，不喂狗；优雅关闭应用进程超时（距优雅关闭开始时间120s过了40s），sanlock发送SIGKILL信号杀进程

170s：wdmd检查170>120，过期，不喂狗；应用进程未杀掉，watchdog最早此时超时，重置主机

179s：应用进程未杀掉，watchdog最晚此时超时，重置主机

180s：租约过期，其他应用进程可以获取租约

代码分析

sanlock守护进程初始化

1、守护进程主函数执行初始化。
src/main.c

```
int main(int argc, char *argv[])
{
   int rv;

   BUILD_BUG_ON(sizeof(struct sanlk_disk) != sizeof(struct sync_disk));
   BUILD_BUG_ON(sizeof(struct leader_record) > LEADER_RECORD_MAX);
   BUILD_BUG_ON(sizeof(struct helper_msg) != SANLK_HELPER_MSG_LEN);
```

```
8
9   /* initialize global EXTERN variables */
10
11  set_sanlock_version();
12
13  kill_count_max = 100;
14  helper_ci = -1;
15  helper_pid = -1;
16  helper_kill_fd = -1;
17  helper_status_fd = -1;
18
19  pthread_mutex_init(&spaces_mutex, NULL);
20  INIT_LIST_HEAD(&spaces);
21  INIT_LIST_HEAD(&spaces_rem);
22  INIT_LIST_HEAD(&spaces_add);
23
24  memset(&com, 0, sizeof(com));
25  com.use_watchdog = DEFAULT_USE_WATCHDOG;
26  com.watchdog_fire_timeout = DEFAULT_WATCHDOG_FIRE_TIMEOUT;
27  com.kill_grace_seconds = DEFAULT_GRACE_SEC;
28  com.high_priority = DEFAULT_HIGH_PRIORITY;
29  com.mlock_level = DEFAULT_MLOCK_LEVEL;
30  com.names_log_priority = LOG_INFO;
31  com.max_worker_threads = DEFAULT_MAX_WORKER_THREADS;
32  com.io_timeout = DEFAULT_IO_TIMEOUT;
33  com.write_init_io_timeout = DEFAULT_WRITE_INIT_IO_TIMEOUT;
34  com.aio_arg = DEFAULT_USE_AIO;
35  com.pid = -1;
36  com.cid = -1;
37  com.sh_retries = DEFAULT_SH_RETRIES;
38  com.quiet_fail = DEFAULT_QUIET_FAIL;
39  com.renewal_read_extend_sec_set = 0;
40  com.renewal_read_extend_sec = 0;
41  com.renewal_history_size = DEFAULT_RENEWAL_HISTORY_SIZE;
42  com.paxos_debug_all = 0;
43  com.max_sectors_kb_ignore = DEFAULT_MAX_SECTORS_KB_IGNORE;
44  com.max_sectors_kb_align = DEFAULT_MAX_SECTORS_KB_ALIGN;
45  com.max_sectors_kb_num = DEFAULT_MAX_SECTORS_KB_NUM;
46  com.use_hugepages = 1;
47  com.debug_cmds = ~0LL;
48  com.debug_hosts = 1;
49
50  /* By default disable cmds that often cause too much logging. */
51  clear_cmd_debug(SM_CMD_INQ_LOCKSPACE);
52  clear_cmd_debug(SM_CMD_GET_LOCKSPACES);
53  clear_cmd_debug(SM_CMD_GET_HOSTS);
54  clear_cmd_debug(SM_CMD_READ_LOCKSPACE);
55  clear_cmd_debug(SM_CMD_READ_RESOURCE);
56  clear_cmd_debug(SM_CMD_READ_RESOURCE_OWNERS);
57  clear_cmd_debug(SM_CMD_WRITE_RESOURCE);
58
59  if (getgrnam("sanlock") && getpwnam("sanlock")) {
60      com.uname = (char *)"sanlock";
61      com.gname = (char *)"sanlock";
62      com.uid = user_to_uid(com.uname);
63      com.gid = group_to_gid(com.uname);
64  } else {
65      com.uname = NULL;
```

```
        com.gname = NULL;
        com.uid = DEFAULT_SOCKET_UID;
        com.gid = DEFAULT_SOCKET_GID;
    }

    memset(&main_task, 0, sizeof(main_task));

    /*
     * read_config_file() overrides com default settings,
     * read_command_line() overrides com default settings and
     * config file settings.
     */
    read_config_file();

    rv = read_command_line(argc, argv);
    if (rv < 0)
        goto out;

    switch (com.type) {
    case COM_DAEMON:
        rv = do_daemon();
        break;

    case COM_CLIENT:
        rv = do_client();
        break;

    case COM_DIRECT:
        rv = do_direct();
        break;
    };
 out:
    return rv == 0 ? EXIT_SUCCESS : EXIT_FAILURE;
}

static int do_daemon(void)
{
    struct utsname nodename;
    int fd, rv;

    run_dir = env_get(SANLOCK_RUN_DIR, DEFAULT_RUN_DIR);
    privileged = env_get_bool(SANLOCK_PRIVILEGED, 1);

    /* This can take a while so do it before forking. */
    setup_groups();

    if (!com.debug) {
        /* TODO: copy comprehensive daemonization method from libvirtd */
        if (daemon(0, 0) < 0) {
            log_tool("cannot fork daemon\n");
            exit(EXIT_FAILURE);
        }
    }

    setup_limits();
    setup_timeouts();
    setup_helper();

```

```
124     /* main task never does disk io, so we don't really need to set
125      * it up, but other tasks get their use_aio value by copying
126      * the main_task settings */
127
128     sprintf(main_task.name, "%s", "main");
129     setup_task_aio(&main_task, com.aio_arg, 0);
130
131     rv = client_alloc();
132     if (rv < 0)
133         return rv;
134
135     helper_ci = client_add(helper_status_fd, process_helper, helper_dead);
136     if (helper_ci < 0)
137         return rv;
138     strcpy(client[helper_ci].owner_name, "helper");
139
140     setup_signals();
141     setup_logging();
142
143     if (strcmp(run_dir, DEFAULT_RUN_DIR))
144         log_warn("Using non-standard run directory '%s'", run_dir);
145
146     if (!privileged)
147         log_warn("Running in unprivileged mode");
148
149     /* If we run as root, make run_dir owned by root, so we can create the
150      * lockfile when selinux disables DAC_OVERRIDE.
151      * See https://danwalsh.livejournal.com/79643.html */
152
153     fd = lockfile(run_dir, SANLK_LOCKFILE_NAME, com.uid,
154             privileged ? 0 : com.gid);
155     if (fd < 0) {
156         close_logging();
157         return fd;
158     }
159
160     setup_host_name();
161
162     setup_uid_gid();
163
164     uname(&nodename);
165
166     if (com.io_timeout != DEFAULT_IO_TIMEOUT || com.watchdog_fire_timeout != DEFAULT_WATCHDOG_FIRE_TIMEOUT) {
167         log_warn("sanlock daemon started %s host %s (%s) io_timeout %u watchdog_fire_timeout %u",
168             VERSION, our_host_name_global, nodename.nodename, com.io_timeout, com.watchdog_fire_timeout);
169         syslog(LOG_INFO, "sanlock daemon started %s host %s (%s) io_timeout %u watchdog_fire_timeout %u",
170             VERSION, our_host_name_global, nodename.nodename, com.io_timeout, com.watchdog_fire_timeout);
171     } else {
172         log_warn("sanlock daemon started %s host %s (%s)",
173             VERSION, our_host_name_global, nodename.nodename);
174         syslog(LOG_INFO, "sanlock daemon started %s host %s (%s)",
175             VERSION, our_host_name_global, nodename.nodename);
176     }
177
178     setup_priority();
179
180     rv = thread_pool_create(DEFAULT_MIN_WORKER_THREADS, com.max_worker_threads);
181     if (rv < 0)
```

```
        goto out;

    rv = setup_listener();
    if (rv < 0)
        goto out_threads;

    setup_token_manager();
    if (rv < 0)
        goto out_threads;

    /* initialize global eventfd for client_resume notification */
    if ((efd = eventfd(0, EFD_CLOEXEC | EFD_NONBLOCK)) == -1) {
        log_error("couldn't create eventfd");
        goto out_threads;
    }

    main_loop();

    close_token_manager();

 out_threads:
    thread_pool_free();
 out:
    /* order reversed from setup so lockfile is last */
    close_logging();
    close(fd);
    return rv;
}
```

2、设置优雅关闭超时时间。

src/timeouts.c

```
void setup_timeouts(void)
{
    /*
     * graceful shutdown is client pids stopping their activity and
     * releasing their sanlock leases in response to a killpath program
     * they configured, or in response to sigterm from sanlock if they
     * did not set a killpath program.  It's an opportunity for the client
     * pid to exit more gracefully than getting sigkill.  If the client
     * pid does not release leases in response to the killpath/sigterm,
     * then eventually sanlock will escalate and send a sigkill.
     *
     * It's hard to know what portion of recovery time should be allocated
     * to graceful shutdown before escalating to sigkill.  The smaller the
     * watchdog timeout, the less time between entering recovery mode and
     * the watchdog potentially firing.  10 seconds before the watchdog
     * will fire, the idea is to give up on graceful shutdown and resort
     * to sending sigkill to any client pids that have not released their
     * leases.  This gives 10 sec for the pids to exit from sigkill,
     * sanlock to get the exit statuses, clear the expiring wdmd connection,
     * and hopefully have wdmd ping the watchdog again before it fires.
     * A graceful shutdown period of less than 10/15 sec seems pointless,
     * so if there is anything less than 10/15 sec available for a graceful
     * shutdown we don't bother and go directly to sigkill (this could
     * of course be changed if programs are indeed able to respond
     * quickly during graceful shutdown.)
     */
```

```
    if (!com.kill_grace_set && (com.watchdog_fire_timeout < DEFAULT_WATCHDOG_FIRE_TIMEOUT)) {
        if (com.watchdog_fire_timeout < 60 && com.watchdog_fire_timeout >= 30)
            com.kill_grace_seconds = 15;
        else if (com.watchdog_fire_timeout < 30)
            com.kill_grace_seconds = 0;
    }
}
```

3、创建helper子进程负责执行kill信号或者执行killpath指定的程序。

src/main.c

```
/*
 * first pipe for daemon to send requests to helper; they are not acknowledged
 * and the daemon does not get any result back for the requests.
 *
 * second pipe for helper to send general status/heartbeat back to the daemon
 * every so often to confirm it's not dead/hung.  If the helper gets stuck or
 * killed, the daemon will not get the status and won't bother sending requests
 * to the helper, and use SIGTERM instead
 */

static int setup_helper(void)
{
    int pid;
    int pw_fd = -1; /* parent write */
    int cr_fd = -1; /* child read */
    int pr_fd = -1; /* parent read */
    int cw_fd = -1; /* child write */
    int pfd[2];

    /* we can't allow the main daemon thread to block */
    if (pipe2(pfd, O_NONBLOCK | O_CLOEXEC))
        return -errno;

    /* uncomment for rhel7 where this should be available */
    /* fcntl(pfd[1], F_SETPIPE_SZ, 1024*1024); */

    cr_fd = pfd[0];
    pw_fd = pfd[1];

    if (pipe2(pfd, O_NONBLOCK | O_CLOEXEC)) {
        close(cr_fd);
        close(pw_fd);
        return -errno;
    }

    pr_fd = pfd[0];
    cw_fd = pfd[1];

    pid = fork();
    if (pid < 0) {
        close(cr_fd);
        close(pw_fd);
        close(pr_fd);
        close(cw_fd);
        return -errno;
    }

```

```
    if (pid) {
        close(cr_fd);
        close(cw_fd);
        helper_kill_fd = pw_fd;
        helper_status_fd = pr_fd;
        helper_pid = pid;
        return 0;
    } else {
        close(pr_fd);
        close(pw_fd);
        is_helper = 1;
        run_helper(cr_fd, cw_fd, (log_stderr_priority == LOG_DEBUG));
        exit(0);
    }
}
```

src/helper.c

通过和主进程之间的管道读取命令并执行：

```
int run_helper(int in_fd, int out_fd, int log_stderr)
{
    unsigned int sysfs_val;
    char name[16];
    struct pollfd pollfd;
    struct helper_msg hm;
    unsigned int fork_count = 0;
    unsigned int wait_count = 0;
    time_t now, last_send, last_good = 0;
    int timeout = STANDARD_TIMEOUT_MS;
    int rv, pid, status;

    memset(name, 0, sizeof(name));
    sprintf(name, "%s", "sanlock-helper");
    prctl(PR_SET_NAME, (unsigned long)name, 0, 0, 0);

    rv = setgroups(0, NULL);
    if (rv < 0)
        log_debug("error clearing helper groups errno %i", errno);

    memset(&pollfd, 0, sizeof(pollfd));
    pollfd.fd = in_fd;
    pollfd.events = POLLIN;

    now = monotime();
    last_send = now;
    rv = send_status(out_fd);
    if (!rv)
        last_good = now;

    while (1) {
        rv = poll(&pollfd, 1, timeout);
        if (rv == -1 && errno == EINTR)
            continue;

        if (rv < 0)
            exit(0);

        now = monotime();
```

```
40
41      if (now - last_good >= HELPER_STATUS_INTERVAL &&
42          now - last_send >= 2) {
43          last_send = now;
44          rv = send_status(out_fd);
45          if (!rv)
46              last_good = now;
47      }
48
49      memset(&hm, 0, sizeof(hm));
50
51      if (pollfd.revents & POLLIN) {
52          rv = read_hm(in_fd, &hm);
53          if (rv)
54              continue;
55
56          /* terminated at sender, but confirm for checker */
57          hm.path[SANLK_HELPER_PATH_LEN-1] = '\0';
58          hm.args[SANLK_HELPER_ARGS_LEN-1] = '\0';
59
60          if (hm.type == HELPER_MSG_RUNPATH) {
61              pid = fork();
62              if (!pid) {
63                  run_path(&hm);
64                  exit(-1);
65              }
66
67              fork_count++;
68
69              /*
70              log_debug("helper fork %d count %d %d %s %s",
71                        pid, fork_count, wait_count,
72                        hm.path, hm.args);
73              */
74          } else if (hm.type == HELPER_MSG_KILLPID) {
75              kill(hm.pid, hm.sig);
76
77          } else if (hm.type == HELPER_MSG_WRITE_SYSFS) {
78              sysfs_val = atoi(hm.args);
79              rv = write_sysfs_uint(hm.path, sysfs_val);
80              log_debug("write_sysfs %s %u rv %d", hm.path, sysfs_val, rv);
81          }
82      }
83
84      if (pollfd.revents & (POLLERR | POLLHUP | POLLNVAL))
85          exit(0);
86
87      /* collect child exits until no more children exist (ECHILD)
88         or none are ready (WNOHANG) */
89
90      while (1) {
91          rv = waitpid(-1, &status, WNOHANG);
92          if (rv > 0) {
93              wait_count++;
94
95              /*
96              log_debug("helper wait %d count %d %d",
97                        rv, fork_count, wait_count);
```

```
            */
            continue;
        }

        /* no more children to wait for or no children
           have exited */

        if (rv < 0 && errno == ECHILD) {
            if (timeout == RECOVERY_TIMEOUT_MS) {
                log_debug("helper no children count %d %d",
                    fork_count, wait_count);
            }
            timeout = STANDARD_TIMEOUT_MS;
        } else {
            timeout = RECOVERY_TIMEOUT_MS;
        }
        break;
    }
  }

  return 0;
}

static void run_path(struct helper_msg *hm)
{
  char arg[SANLK_HELPER_ARGS_LEN];
  char *args = hm->args;
  char *av[MAX_AV_COUNT + 1]; /* +1 for NULL */
  int av_count = 0;
  int i, arg_len, args_len;

  for (i = 0; i < MAX_AV_COUNT + 1; i++)
    av[i] = NULL;

  av[av_count++] = strdup(hm->path);

  if (!args[0])
    goto pid_arg;

  /* this should already be done, but make sure */
  args[SANLK_HELPER_ARGS_LEN - 1] = '\0';

  memset(&arg, 0, sizeof(arg));
  arg_len = 0;
  args_len = strlen(args);

  for (i = 0; i < args_len; i++) {
    if (!args[i])
      break;

    if (av_count == MAX_AV_COUNT)
      break;

    if (args[i] == '\\') {
      if (i == (args_len - 1))
        break;
      i++;

```

```
            if (args[i] == '\\') {
                arg[arg_len++] = args[i];
                continue;
            }
            if (isspace(args[i])) {
                arg[arg_len++] = args[i];
                continue;
            } else {
                break;
            }
        }

        if (isalnum(args[i]) || ispunct(args[i])) {
            arg[arg_len++] = args[i];
        } else if (isspace(args[i])) {
            if (arg_len)
                av[av_count++] = strdup(arg);

            memset(arg, 0, sizeof(arg));
            arg_len = 0;
        } else {
            break;
        }
    }

    if ((av_count < MAX_AV_COUNT) && arg_len) {
        av[av_count++] = strdup(arg);
    }

  pid_arg:
    if ((av_count < MAX_AV_COUNT) && hm->pid) {
        memset(arg, 0, sizeof(arg));
        snprintf(arg, sizeof(arg)-1, "%d", hm->pid);
        av[av_count++] = strdup(arg);
    }

    execvp(av[0], av);
  }
```

4、初始化并分配客户端管理数据结构。

src/main.c

```
/* FIXME: add a mutex for client array so we don't try to expand it
   while a cmd thread is using it.  Or, with a thread pool we know
   when cmd threads are running and can expand when none are. */

static int client_alloc(void)
{
    int i;

    /* pollfd is one element longer as we use an additional element for the
     * eventfd notification mechanism */
    client = malloc(CLIENT_NALLOC * sizeof(struct client));
    pollfd = malloc((CLIENT_NALLOC+1) * sizeof(struct pollfd));

    if (!client || !pollfd) {
        log_error("can't alloc for client or pollfd array");
        return -ENOMEM;
```

```
    }

    for (i = 0; i < CLIENT_NALLOC; i++) {
        memset(&client[i], 0, sizeof(struct client));
        memset(&pollfd[i], 0, sizeof(struct pollfd));

        pthread_mutex_init(&client[i].mutex, NULL);
        client[i].fd = -1;
        client[i].pid = -1;

        pollfd[i].fd = -1;
        pollfd[i].events = 0;
    }
    client_size = CLIENT_NALLOC;
    return 0;
}
```

5、创建sanlock worker线程池

src/main.c

```
static int thread_pool_create(int min_workers, int max_workers)
{
    pthread_t th;
    int i, rv = 0;

    memset(&pool, 0, sizeof(pool));
    INIT_LIST_HEAD(&pool.work_data);
    pthread_mutex_init(&pool.mutex, NULL);
    pthread_cond_init(&pool.cond, NULL);
    pthread_cond_init(&pool.quit_wait, NULL);
    pool.max_workers = max_workers;
    pool.info = malloc(max_workers * sizeof(struct worker_info));

    if (!pool.info)
        return -ENOMEM;

    memset(pool.info, 0, max_workers * sizeof(struct worker_info));

    for (i = 0; i < min_workers; i++) {
        rv = pthread_create(&th, NULL, thread_pool_worker,
                (void *)(long)i);
        if (rv) {
            log_error("thread_pool_create failed %d", rv);
            rv = -1;
            break;
        }
        pool.num_workers++;
        daemon_status_num_workers++; /* for status reporting */
    }

    if (rv < 0)
        thread_pool_free();

    return rv;
}

static void *thread_pool_worker(void *data)
{
```

```
    struct task task;
    struct cmd_args *ca;
    long workerid = (long)data;

    memset(&task, 0, sizeof(struct task));
    setup_task_aio(&task, main_task.use_aio, WORKER_AIO_CB_SIZE);
    snprintf(task.name, NAME_ID_SIZE, "worker%ld", workerid);

    pthread_mutex_lock(&pool.mutex);

    while (1) {
        while (!pool.quit && list_empty(&pool.work_data)) {
            pool.free_workers++;
            pthread_cond_wait(&pool.cond, &pool.mutex);
            pool.free_workers--;
        }

        while (!list_empty(&pool.work_data)) {
            ca = list_first_entry(&pool.work_data, struct cmd_args, list);
            list_del(&ca->list);
            pool.info[workerid].busy = 1;
            pool.info[workerid].cmd_last = ca->header.cmd;
            pthread_mutex_unlock(&pool.mutex);

            call_cmd_thread(&task, ca);
            free(ca);

            pthread_mutex_lock(&pool.mutex);
            pool.info[workerid].busy = 0;
            pool.info[workerid].work_count++;
            pool.info[workerid].io_count = task.io_count;
            pool.info[workerid].to_count = task.to_count;
            pool.info[workerid].paxos_hugebuf_created = task.paxos_hugebuf_created;
            pool.info[workerid].paxos_hugebuf_used = task.paxos_hugebuf_used;
        }

        if (pool.quit)
            break;
    }

    pool.num_workers--;
    daemon_status_num_workers--; /* for status reporting */
    if (!pool.num_workers)
        pthread_cond_signal(&pool.quit_wait);
    pthread_mutex_unlock(&pool.mutex);

    close_task_aio(&task);
    return NULL;
}
```

6、设置本地套接字监听端口及处理函数。

src/main.c

初始化套接字，并绑定端口，设置处理函数：

```
static int setup_listener(void)
{
    struct sockaddr_un addr;
```

```
    int rv, fd, ci;

    rv = sanlock_socket_address(run_dir, &addr);
    if (rv < 0)
        return rv;

    fd = socket(AF_LOCAL, SOCK_STREAM, 0);
    if (fd < 0)
        return fd;

    unlink(addr.sun_path);
    rv = bind(fd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un));
    if (rv < 0)
        goto exit_fail;

    rv = chmod(addr.sun_path, DEFAULT_SOCKET_MODE);
    if (rv < 0)
        goto exit_fail;

    rv = chown(addr.sun_path, com.uid, com.gid);
    if (rv < 0) {
        log_error("could not set socket %s permissions: %s",
            addr.sun_path, strerror(errno));
        goto exit_fail;
    }

    rv = listen(fd, 5);
    if (rv < 0)
        goto exit_fail;

    fcntl(fd, F_SETFL, fcntl(fd, F_GETFL, 0) | O_NONBLOCK);

    ci = client_add(fd, process_listener, NULL);
    if (ci < 0)
        goto exit_fail;

    strcpy(client[ci].owner_name, "listener");
    return 0;

 exit_fail:
    close(fd);
    return -1;
}
```

src/sanlock_sock.c

初始化监听地址：

```
int sanlock_socket_address(const char *dir, struct sockaddr_un *addr)
{
    memset(addr, 0, sizeof(struct sockaddr_un));
    addr->sun_family = AF_LOCAL;
    snprintf(addr->sun_path, sizeof(addr->sun_path) - 1, "%s/%s",
        dir, SANLK_SOCKET_NAME);
    return 0;
}
```

src/sanlock_sock.h

默认的本地监听套接字：

```
#define SANLK_SOCKET_NAME "sanlock.sock"
```

src/main.c

服务端处理函数，对于监听套接字，每接收一个新的连接则分配一个client结构与之对应，例如在SM_CMD_KILLPATH命令中处理设置killpath的功能：

```
static void process_listener(int ci GNUC_UNUSED)
{
    int fd;
    int on = 1;

    fd = accept(client[ci].fd, NULL, NULL);
    if (fd < 0)
        return;

    setsockopt(fd, SOL_SOCKET, SO_PASSCRED, &on, sizeof(on));

    client_add(fd, process_connection, NULL);
}

static void process_connection(int ci)
{
    struct sm_header h;
    void (*deadfn)(int ci);
    int rv;

    memset(&h, 0, sizeof(h));

 retry:
    rv = recv(client[ci].fd, &h, sizeof(h), MSG_WAITALL);
    if (rv == -1 && errno == EINTR)
        goto retry;
    if (!rv)
        goto dead;

    log_client(ci, client[ci].fd, "recv %d %u", rv, h.cmd);

    if (rv < 0) {
        log_error("client connection %d %d %d recv msg header rv %d errno %d",
                  ci, client[ci].fd, client[ci].pid, rv, errno);
        goto bad;
    }
    if (rv != sizeof(h)) {
        log_error("client connection %d %d %d recv msg header rv %d cmd %u len %u",
                  ci, client[ci].fd, client[ci].pid, rv, h.cmd, h.length);
        goto bad;
    }
    if (h.magic != SM_MAGIC) {
        log_error("client connection %d %d %d recv msg header rv %d cmd %u len %u magic %x vs %x",
                  ci, client[ci].fd, client[ci].pid, rv, h.cmd, h.length, h.magic, SM_MAGIC);
        goto bad;
    }
    if (client[ci].restricted & SANLK_RESTRICT_ALL) {
        log_error("client connection %d %d %d recv msg header rv %d cmd %u len %u restrict all",
                  ci, client[ci].fd, client[ci].pid, rv, h.cmd, h.length);
        goto bad;
```

```
51      }
52      if (h.version && (h.cmd != SM_CMD_VERSION) &&
53          (h.version & 0xFFFF0000) > (SM_PROTO & 0xFFFF0000)) {
54          log_error("client connection %d %d %d recv msg header rv %d cmd %u len %u version %x",
55                    ci, client[ci].fd, client[ci].pid, rv, h.cmd, h.length, h.version);
56          goto bad;
57      }
58
59      client[ci].cmd_last = h.cmd;
60
61      switch (h.cmd) {
62      case SM_CMD_REGISTER:
63      case SM_CMD_RESTRICT:
64      case SM_CMD_VERSION:
65      case SM_CMD_SHUTDOWN:
66      case SM_CMD_STATUS:
67      case SM_CMD_HOST_STATUS:
68      case SM_CMD_RENEWAL:
69      case SM_CMD_LOG_DUMP:
70      case SM_CMD_GET_LOCKSPACES:
71      case SM_CMD_GET_HOSTS:
72      case SM_CMD_REG_EVENT:
73      case SM_CMD_END_EVENT:
74      case SM_CMD_SET_CONFIG:
75          call_cmd_daemon(ci, &h, client_maxi);
76          break;
77      case SM_CMD_ADD_LOCKSPACE:
78      case SM_CMD_INQ_LOCKSPACE:
79      case SM_CMD_REM_LOCKSPACE:
80      case SM_CMD_REQUEST:
81      case SM_CMD_EXAMINE_RESOURCE:
82      case SM_CMD_EXAMINE_LOCKSPACE:
83      case SM_CMD_ALIGN:
84      case SM_CMD_WRITE_LOCKSPACE:
85      case SM_CMD_WRITE_RESOURCE:
86      case SM_CMD_READ_LOCKSPACE:
87      case SM_CMD_READ_RESOURCE:
88      case SM_CMD_READ_RESOURCE_OWNERS:
89      case SM_CMD_SET_LVB:
90      case SM_CMD_GET_LVB:
91      case SM_CMD_SHUTDOWN_WAIT:
92      case SM_CMD_SET_EVENT:
93      case SM_CMD_FORMAT_RINDEX:
94      case SM_CMD_REBUILD_RINDEX:
95      case SM_CMD_UPDATE_RINDEX:
96      case SM_CMD_LOOKUP_RINDEX:
97      case SM_CMD_CREATE_RESOURCE:
98      case SM_CMD_DELETE_RESOURCE:
99          rv = client_suspend(ci);
100         if (rv < 0)
101             goto bad;
102         process_cmd_thread_unregistered(ci, &h);
103         break;
104     case SM_CMD_ACQUIRE:
105     case SM_CMD_RELEASE:
106     case SM_CMD_INQUIRE:
107     case SM_CMD_CONVERT:
108     case SM_CMD_KILLPATH:
```

```
        /* the main_loop needs to ignore this connection
           while the thread is working on it */
        rv = client_suspend(ci);
        if (rv < 0)
            goto bad;
        process_cmd_thread_registered(ci, &h);
        break;
    default:
        log_error("client connection ci %d fd %d pid %d cmd %d unknown",
            ci, client[ci].fd, client[ci].pid, h.cmd);
        goto bad;
    };

    return;

 bad:
    return;

 dead:
    log_client(ci, client[ci].fd, "recv dead");
    deadfn = client[ci].deadfn;
    if (deadfn)
        deadfn(ci);
}
```

处理设置killpath命令的函数

- src/cmd.c

命令处理函数：

```
void call_cmd_thread(struct task *task, struct cmd_args *ca)
{
    uint32_t cmd = ca->header.cmd;

    switch (cmd) {
    case SM_CMD_ACQUIRE:
        cmd_acquire(task, ca, cmd);
        break;
    case SM_CMD_RELEASE:
        cmd_release(task, ca, cmd);
        break;
    case SM_CMD_INQUIRE:
        cmd_inquire(task, ca, cmd);
        break;
    case SM_CMD_CONVERT:
        cmd_convert(task, ca, cmd);
        break;
    case SM_CMD_REQUEST:
        cmd_request(task, ca, cmd);
        break;
    case SM_CMD_ADD_LOCKSPACE:
        strcpy(client[ca->ci_in].owner_name, "add_lockspace");
        cmd_add_lockspace(ca, cmd);
        break;
    case SM_CMD_INQ_LOCKSPACE:
        strcpy(client[ca->ci_in].owner_name, "inq_lockspace");
        cmd_inq_lockspace(ca, cmd);
        break;
```

```
    case SM_CMD_REM_LOCKSPACE:
        strcpy(client[ca->ci_in].owner_name, "rem_lockspace");
        cmd_rem_lockspace(ca, cmd);
        break;
    case SM_CMD_ALIGN:
        cmd_align(task, ca, cmd);
        break;
    case SM_CMD_WRITE_LOCKSPACE:
        cmd_write_lockspace(task, ca, cmd);
        break;
    case SM_CMD_WRITE_RESOURCE:
        cmd_write_resource(task, ca, cmd);
        break;
    case SM_CMD_READ_LOCKSPACE:
        cmd_read_lockspace(task, ca, cmd);
        break;
    case SM_CMD_READ_RESOURCE:
        cmd_read_resource(task, ca, cmd);
        break;
    case SM_CMD_READ_RESOURCE_OWNERS:
        cmd_read_resource_owners(task, ca, cmd);
        break;
    case SM_CMD_EXAMINE_LOCKSPACE:
    case SM_CMD_EXAMINE_RESOURCE:
        cmd_examine(task, ca, cmd);
        break;
    case SM_CMD_KILLPATH:
        cmd_killpath(task, ca, cmd);
        break;
    case SM_CMD_SET_LVB:
        cmd_set_lvb(task, ca, cmd);
        break;
    case SM_CMD_GET_LVB:
        cmd_get_lvb(task, ca, cmd);
        break;
    case SM_CMD_SHUTDOWN_WAIT:
        cmd_shutdown_wait(task, ca, cmd);
        break;
    case SM_CMD_SET_EVENT:
        cmd_set_event(task, ca, cmd);
        break;
    case SM_CMD_FORMAT_RINDEX:
        cmd_format_rindex(task, ca, cmd);
        break;
    case SM_CMD_REBUILD_RINDEX:
        cmd_rebuild_rindex(task, ca, cmd);
        break;
    case SM_CMD_UPDATE_RINDEX:
        rindex_op(task, ca, "cmd_update_rindex", RX_OP_UPDATE, cmd);
        break;
    case SM_CMD_LOOKUP_RINDEX:
        rindex_op(task, ca, "cmd_lookup_rindex", RX_OP_LOOKUP, cmd);
        break;
    case SM_CMD_CREATE_RESOURCE:
        rindex_op(task, ca, "cmd_create_resource", RX_OP_CREATE, cmd);
        break;
    case SM_CMD_DELETE_RESOURCE:
        rindex_op(task, ca, "cmd_delete_resource", RX_OP_DELETE, cmd);
```

```
        break;
    };
}
```

一个进程只能设置自己的killpath，而不能设置其他进程的：

```
/* N.B. the api doesn't support one client setting killpath for another
   pid/client */

static void cmd_killpath(struct task *task, struct cmd_args *ca, uint32_t cmd)
{
    struct client *cl;
    int cl_ci = ca->ci_target;
    int cl_fd = ca->cl_fd;
    int cl_pid = ca->cl_pid;
    int rv, result, pid_dead;

    cl = &client[cl_ci];

    log_cmd(cmd, "cmd_killpath %d,%d,%d flags %x",
            cl_ci, cl_fd, cl_pid, ca->header.cmd_flags);

    rv = recv_loop(cl_fd, cl->killpath, SANLK_HELPER_PATH_LEN, MSG_WAITALL);
    if (rv != SANLK_HELPER_PATH_LEN) {
        log_error("cmd_killpath %d,%d,%d recv path %d %d",
                  cl_ci, cl_fd, cl_pid, rv, errno);
        memset(cl->killpath, 0, SANLK_HELPER_PATH_LEN);
        memset(cl->killargs, 0, SANLK_HELPER_ARGS_LEN);
        result = -ENOTCONN;
        goto done;
    }

    rv = recv_loop(cl_fd, cl->killargs, SANLK_HELPER_ARGS_LEN, MSG_WAITALL);
    if (rv != SANLK_HELPER_ARGS_LEN) {
        log_error("cmd_killpath %d,%d,%d recv args %d %d",
                  cl_ci, cl_fd, cl_pid, rv, errno);
        memset(cl->killpath, 0, SANLK_HELPER_PATH_LEN);
        memset(cl->killargs, 0, SANLK_HELPER_ARGS_LEN);
        result = -ENOTCONN;
        goto done;
    }

    cl->killpath[SANLK_HELPER_PATH_LEN - 1] = '\0';
    cl->killargs[SANLK_HELPER_ARGS_LEN - 1] = '\0';

    if (ca->header.cmd_flags & SANLK_KILLPATH_PID)
        cl->flags |= CL_KILLPATH_PID;

    result = 0;
 done:
    pthread_mutex_lock(&cl->mutex);
    pid_dead = cl->pid_dead;
    cl->cmd_active = 0;
    pthread_mutex_unlock(&cl->mutex);

    if (pid_dead) {
        /* release tokens in case a client sets/changes its killpath
           after it has acquired leases */
        release_cl_tokens(task, cl);
```

```
        client_free(cl_ci);
        return;
    }

    send_result(ca->ci_in, cl_fd, &ca->header, result);
    client_resume(ca->ci_in);
}
```

发送修改killpath的消息给sanlock守护进程；

- src/client.c

```
int sanlock_killpath(int sock, uint32_t flags, const char *path, char *args)
{
    char path_max[SANLK_HELPER_PATH_LEN];
    char args_max[SANLK_HELPER_ARGS_LEN];
    int rv, datalen;

    datalen = SANLK_HELPER_PATH_LEN + SANLK_HELPER_ARGS_LEN;

    memset(path_max, 0, sizeof(path_max));
    memset(args_max, 0, sizeof(args_max));

    snprintf(path_max, SANLK_HELPER_PATH_LEN-1, "%s", path);
    snprintf(args_max, SANLK_HELPER_ARGS_LEN-1, "%s", args);

    rv = send_header(sock, SM_CMD_KILLPATH, flags, datalen, 0, -1);
    if (rv < 0)
        return rv;

    rv = send_data(sock, path_max, SANLK_HELPER_PATH_LEN, 0);
    if (rv < 0) {
        rv = -errno;
        goto out;
    }

    rv = send_data(sock, args_max, SANLK_HELPER_ARGS_LEN, 0);
    if (rv < 0) {
        rv = -errno;
        goto out;
    }

    rv = recv_result(sock);
 out:
    return rv;
}
```

处理添加lockspace命令的函数

src/cmd.c

```
static void cmd_add_lockspace(struct cmd_args *ca, uint32_t cmd)
{
    struct sanlk_lockspace lockspace;
    struct space *sp;
    uint32_t io_timeout;
    int async = ca->header.cmd_flags & SANLK_ADD_ASYNC;
    int fd, rv, result;

```

```
    fd = client[ca->ci_in].fd;

    rv = recv_loop(fd, &lockspace, sizeof(struct sanlk_lockspace), MSG_WAITALL);
    if (rv != sizeof(struct sanlk_lockspace)) {
        log_error("cmd_add_lockspace %d,%d recv %d %d",
                  ca->ci_in, fd, rv, errno);
        result = -ENOTCONN;
        goto reply;
    }

    log_cmd(cmd, "cmd_add_lockspace %d,%d %.48s:%llu:%s:%llu flags %x timeout %u",
            ca->ci_in, fd, lockspace.name,
            (unsigned long long)lockspace.host_id,
            lockspace.host_id_disk.path,
            (unsigned long long)lockspace.host_id_disk.offset,
            ca->header.cmd_flags, ca->header.data);

    io_timeout = ca->header.data;
    if (!io_timeout)
        io_timeout = com.io_timeout;

    rv = add_lockspace_start(&lockspace, io_timeout, &sp);
    if (rv < 0) {
        result = rv;
        goto reply;
    }

    if (async) {
        result = rv;
        log_cmd(cmd, "cmd_add_lockspace %d,%d async done %d", ca->ci_in, fd, result);
        send_result(ca->ci_in, fd, &ca->header, result);
        client_resume(ca->ci_in);
        add_lockspace_wait(sp);
        return;
    }

    result = add_lockspace_wait(sp);
 reply:
    log_cmd(cmd, "cmd_add_lockspace %d,%d done %d", ca->ci_in, fd, result);
    send_result(ca->ci_in, fd, &ca->header, result);
    client_resume(ca->ci_in);
}
```

src/lockspace.c

创建lockspace线程

```
int add_lockspace_start(struct sanlk_lockspace *ls, uint32_t io_timeout, struct space **sp_out)
{
    struct space *sp, *sp2;
    int listnum = 0;
    int rv;
    int i;

    if (!ls->name[0] || !ls->host_id || !ls->host_id_disk.path[0]) {
        log_error("add_lockspace bad args host_id %llu name %s path %s",
                  (unsigned long long)ls->host_id,
                  ls->name[0] ? "set" : "empty",
                  ls->host_id_disk.path[0] ? "set" : "empty");
```

```
    return -EINVAL;
  }

  sp = malloc(sizeof(struct space));
  if (!sp)
    return -ENOMEM;
  memset(sp, 0, sizeof(struct space));

  memcpy(sp->space_name, ls->name, NAME_ID_SIZE);
  memcpy(&sp->host_id_disk, &ls->host_id_disk, sizeof(struct sanlk_disk));
  sp->host_id_disk.sector_size = 0;
  sp->host_id_disk.fd = -1;
  sp->host_id = ls->host_id;
  sp->io_timeout = io_timeout;
  sp->set_bitmap_seconds = calc_set_bitmap_seconds(io_timeout);
  pthread_mutex_init(&sp->mutex, NULL);

  if (com.renewal_read_extend_sec_set)
    sp->renewal_read_extend_sec = com.renewal_read_extend_sec;
  else
    sp->renewal_read_extend_sec = io_timeout;

  for (i = 0; i < MAX_EVENT_FDS; i++)
    sp->event_fds[i] = -1;

  if (com.renewal_history_size) {
    sp->renewal_history = malloc(sizeof(struct renewal_history) * com.renewal_history_size);
    if (sp->renewal_history) {
      sp->renewal_history_size = com.renewal_history_size;
      memset(sp->renewal_history, 0, sizeof(struct renewal_history) * com.renewal_history_size);
    }
  }

  pthread_mutex_lock(&spaces_mutex);

  /* search all lists for an identical lockspace */

  sp2 = _search_space(sp->space_name, &sp->host_id_disk, sp->host_id,
          &spaces, NULL, NULL, NULL);
  if (sp2) {
    pthread_mutex_unlock(&spaces_mutex);
    rv = -EEXIST;
    goto fail_free;
  }

  sp2 = _search_space(sp->space_name, &sp->host_id_disk, sp->host_id,
          &spaces_add, NULL, NULL, NULL);
  if (sp2) {
    pthread_mutex_unlock(&spaces_mutex);
    rv = -EINPROGRESS;
    goto fail_free;
  }

  sp2 = _search_space(sp->space_name, &sp->host_id_disk, sp->host_id,
          &spaces_rem, NULL, NULL, NULL);
  if (sp2) {
    pthread_mutex_unlock(&spaces_mutex);
    rv = -EAGAIN;
```

```
    goto fail_free;
  }

  /* search all lists for a lockspace with the same name */

  sp2 = _search_space(sp->space_name, NULL, 0,
            &spaces, &spaces_add, &spaces_rem, &listnum);
  if (sp2) {
    log_error("add_lockspace %.48s:%llu:%.256s:%llu conflicts with name of list%d s%d %.48s:%llu:%.256s:%llu",
        sp->space_name,
        (unsigned long long)sp->host_id,
        sp->host_id_disk.path,
        (unsigned long long)sp->host_id_disk.offset,
        listnum,
        sp2->space_id,
        sp2->space_name,
        (unsigned long long)sp2->host_id,
        sp2->host_id_disk.path,
        (unsigned long long)sp2->host_id_disk.offset);
    pthread_mutex_unlock(&spaces_mutex);
    rv = -EINVAL;
    goto fail_free;
  }

  /* search all lists for a lockspace with the same host_id_disk */

  sp2 = _search_space(NULL, &sp->host_id_disk, 0,
            &spaces, &spaces_add, &spaces_rem, &listnum);
  if (sp2) {
    log_error("add_lockspace %.48s:%llu:%.256s:%llu conflicts with path of list%d s%d %.48s:%llu:%.256s:%llu",
        sp->space_name,
        (unsigned long long)sp->host_id,
        sp->host_id_disk.path,
        (unsigned long long)sp->host_id_disk.offset,
        listnum,
        sp2->space_id,
        sp2->space_name,
        (unsigned long long)sp2->host_id,
        sp2->host_id_disk.path,
        (unsigned long long)sp2->host_id_disk.offset);
    pthread_mutex_unlock(&spaces_mutex);
    rv = -EINVAL;
    goto fail_free;
  }

  sp->space_id = space_id_counter++;
  list_add(&sp->list, &spaces_add);
  pthread_mutex_unlock(&spaces_mutex);

  /* save a record of what this space_id is for later debugging */
  log_warns(sp, "lockspace %.48s:%llu:%.256s:%llu",
      sp->space_name,
      (unsigned long long)sp->host_id,
      sp->host_id_disk.path,
      (unsigned long long)sp->host_id_disk.offset);

  rv = pthread_create(&sp->thread, NULL, lockspace_thread, sp);
  if (rv) {
```

```
        log_erros(sp, "add_lockspace create thread failed %d", rv);
        rv = -1;
        goto fail_del;
    }

    *sp_out = sp;
    return 0;

 fail_del:
    pthread_mutex_lock(&spaces_mutex);
    list_del(&sp->list);
    pthread_mutex_unlock(&spaces_mutex);
 fail_free:
    free_sp(sp);
    return rv;
}

static void *lockspace_thread(void *arg_in)
{
    char bitmap[HOSTID_BITMAP_SIZE];
    struct delta_extra extra;
    struct task task;
    struct space *sp;
    struct leader_record leader;
    uint64_t delta_begin, last_success = 0;
    int sector_size = 0;
    int align_size = 0;
    int max_hosts = 0;
    int log_renewal_level = -1;
    int rv, delta_length, renewal_interval = 0;
    int id_renewal_seconds, id_renewal_fail_seconds;
    int acquire_result, delta_result, read_result;
    int rd_ms, wr_ms;
    int opened = 0;
    int stop = 0;
    int wd_con;

    if (com.debug_renew)
        log_renewal_level = LOG_DEBUG;

    sp = (struct space *)arg_in;

    memset(&task, 0, sizeof(struct task));
    setup_task_aio(&task, main_task.use_aio, HOSTID_AIO_CB_SIZE);
    memcpy(task.name, sp->space_name, NAME_ID_SIZE);

    id_renewal_seconds = calc_id_renewal_seconds(sp->io_timeout);
    id_renewal_fail_seconds = calc_id_renewal_fail_seconds(sp->io_timeout);

    delta_begin = monotime();

    rv = open_disk(&sp->host_id_disk);
    if (rv < 0) {
        log_erros(sp, "open_disk %s error %d", sp->host_id_disk.path, rv);
        acquire_result = -ENODEV;
        delta_result = -1;
        goto set_status;
    }
```

```
187     opened = 1;
188
189     rv = delta_read_lockspace_sizes(&task, &sp->host_id_disk, sp->io_timeout, &sector_size, &align_size);
190     if (rv < 0) {
191         log_erros(sp, "failed to read device to find sector size error %d %s", rv, sp->host_id_disk.path);
192         acquire_result = rv;
193         delta_result = -1;
194         goto set_status;
195     }
196
197     if ((sector_size != 512) && (sector_size != 4096)) {
198         log_erros(sp, "failed to get valid sector size %d %s", sector_size, sp->host_id_disk.path);
199         acquire_result = SANLK_LEADER_SECTORSIZE;
200         delta_result = -1;
201         goto set_status;
202     }
203
204     max_hosts = size_to_max_hosts(sector_size, align_size);
205     if (!max_hosts) {
206         log_erros(sp, "invalid combination of sector size %d and align_size %d", sector_size, align_size);
207         acquire_result = SANLK_ADDLS_SIZES;
208         delta_result = -1;
209         goto set_status;
210     }
211
212     if (sp->host_id > max_hosts) {
213         log_erros(sp, "host_id %llu too large for max_hosts %d", (unsigned long long)sp->host_id, max_hosts);
214         acquire_result = SANLK_ADDLS_INVALID_HOSTID;
215         delta_result = -1;
216         goto set_status;
217     }
218
219     sp->sector_size = sector_size;
220     sp->align_size = align_size;
221     sp->max_hosts = max_hosts;
222
223     set_lockspace_max_sectors_kb(sp, sector_size, align_size);
224
225     sp->lease_status.renewal_read_buf = malloc(sp->align_size);
226     if (!sp->lease_status.renewal_read_buf) {
227         acquire_result = -ENOMEM;
228         delta_result = -1;
229         goto set_status;
230     }
231
232     /* Connect first so we can fail quickly if wdmd is not running. */
233     wd_con = connect_watchdog(sp);
234     if (wd_con < 0) {
235         log_erros(sp, "connect_watchdog failed %d", wd_con);
236         acquire_result = SANLK_WD_ERROR;
237         delta_result = -1;
238         goto set_status;
239     }
240
241     /*
242      * Tell wdmd to open the watchdog device, set the fire timeout and
243      * begin the keepalive loop that regularly pets the watchdog.  This
244      * only happens for the first client/lockspace.  This fails if the
```

```
245     * watchdog device cannot be opened by wdmd or does not support the
246     * requested fire timeout.
247     *
248     * For later clients/lockspaces, when wdmd already has the watchdog
249     * open, this does nothing (just verifies that fire timeout matches
250     * what's in use.)
251     */
252    rv = open_watchdog(wd_con, com.watchdog_fire_timeout);
253    if (rv < 0) {
254       log_erros(sp, "open_watchdog with fire_timeout %d failed %d",
255             com.watchdog_fire_timeout, wd_con);
256       acquire_result = SANLK_WD_ERROR;
257       delta_result = -1;
258       disconnect_watchdog(sp);
259       goto set_status;
260    }
261
262    /*
263     * acquire the delta lease
264     */
265
266    delta_begin = monotime();
267
268    delta_result = delta_lease_acquire(&task, sp, &sp->host_id_disk,
269                       sp->space_name, our_host_name_global,
270                       sp->host_id, &leader);
271    delta_length = monotime() - delta_begin;
272
273    if (delta_result == SANLK_OK)
274       last_success = leader.timestamp;
275
276    acquire_result = delta_result;
277
278    /* we need to start the watchdog after we acquire the host_id but
279       before we allow any pid's to begin running */
280
281    if (delta_result == SANLK_OK) {
282       rv = activate_watchdog(sp, last_success, id_renewal_fail_seconds, wd_con);
283       if (rv < 0) {
284          log_erros(sp, "activate_watchdog failed %d", rv);
285          acquire_result = SANLK_WD_ERROR;
286       }
287    } else {
288       if (com.use_watchdog)
289          close(wd_con);
290    }
291
292 set_status:
293    pthread_mutex_lock(&sp->mutex);
294    sp->lease_status.acquire_last_result = acquire_result;
295    sp->lease_status.acquire_last_attempt = delta_begin;
296    if (delta_result == SANLK_OK)
297       sp->lease_status.acquire_last_success = last_success;
298    sp->lease_status.renewal_last_result = acquire_result;
299    sp->lease_status.renewal_last_attempt = delta_begin;
300    if (delta_result == SANLK_OK)
301       sp->lease_status.renewal_last_success = last_success;
302    /* First renewal entry shows the acquire time with 0 latencies. */
```

```
303     save_renewal_history(sp, delta_result, last_success, 0, 0);
304     pthread_mutex_unlock(&sp->mutex);
305
306     if (acquire_result < 0)
307         goto out;
308
309     sp->host_generation = leader.owner_generation;
310
311     while (1) {
312         pthread_mutex_lock(&sp->mutex);
313         stop = sp->thread_stop;
314         pthread_mutex_unlock(&sp->mutex);
315         if (stop)
316             break;
317
318         /*
319          * wait between each renewal
320          */
321
322         if (monotime() - last_success < id_renewal_seconds) {
323             sleep(1);
324             continue;
325         } else {
326             /* don't spin too quickly if renew is failing
327                immediately and repeatedly */
328             usleep(500000);
329         }
330
331
332         /*
333          * do a renewal, measuring length of time spent in renewal,
334          * and the length of time between successful renewals
335          */
336
337         memset(bitmap, 0, sizeof(bitmap));
338         memset(&extra, 0, sizeof(extra));
339         create_bitmap_and_extra(sp, bitmap, &extra);
340
341         delta_begin = monotime();
342
343         delta_result = delta_lease_renew(&task, sp, &sp->host_id_disk,
344                         sp->space_name, bitmap, &extra,
345                         delta_result, &read_result,
346                         log_renewal_level,
347                         &leader, &leader,
348                         &rd_ms, &wr_ms);
349         delta_length = monotime() - delta_begin;
350
351         if (delta_result == SANLK_OK) {
352             renewal_interval = leader.timestamp - last_success;
353             last_success = leader.timestamp;
354         }
355
356
357         /*
358          * publish the results
359          */
360
```

```
361         pthread_mutex_lock(&sp->mutex);
362         sp->lease_status.renewal_last_result = delta_result;
363         sp->lease_status.renewal_last_attempt = delta_begin;
364
365         if (delta_result == SANLK_OK)
366             sp->lease_status.renewal_last_success = last_success;
367
368         if (delta_result != SANLK_OK && !sp->lease_status.corrupt_result)
369             sp->lease_status.corrupt_result = corrupt_result(delta_result);
370
371         if (read_result == SANLK_OK && task.iobuf) {
372             /* NB. be careful with how this iobuf escapes */
373             memcpy(sp->lease_status.renewal_read_buf, task.iobuf, sp->align_size);
374             sp->lease_status.renewal_read_count++;
375         }
376
377         /*
378          * pet the watchdog
379          * (don't update on thread_stop because it's probably unlinked)
380          */
381
382         if (delta_result == SANLK_OK && !sp->thread_stop)
383             update_watchdog(sp, last_success, id_renewal_fail_seconds);
384
385         save_renewal_history(sp, delta_result, last_success, rd_ms, wr_ms);
386         pthread_mutex_unlock(&sp->mutex);
387
388
389         /*
390          * log the results
391          */
392
393         if (delta_result != SANLK_OK) {
394             log_erros(sp, "renewal error %d delta_length %d last_success %llu",
395                       delta_result, delta_length, (unsigned long long)last_success);
396         } else if (delta_length > id_renewal_seconds) {
397             log_erros(sp, "renewed %llu delta_length %d too long",
398                       (unsigned long long)last_success, delta_length);
399         } else {
400             if (com.debug_renew) {
401                 log_space(sp, "renewed %llu delta_length %d interval %d",
402                           (unsigned long long)last_success, delta_length, renewal_interval);
403             }
404         }
405     }
406
407     /* watchdog unlink was done in main_loop when thread_stop was set, to
408        get it done as quickly as possible in case the wd is about to fire. */
409
410     disconnect_watchdog(sp);
411 out:
412     if (delta_result == SANLK_OK)
413         delta_lease_release(&task, sp, &sp->host_id_disk,
414                             sp->space_name, &leader, &leader);
415
416     if (opened)
417         close(sp->host_id_disk.fd);
418
```

```
    /*
     * TODO: are there cases where struct resources for this lockspace
     * still exist on resource_held/resource_add/resource_rem?  Is that ok?
     * Should we purge all of them here?  When a lockspace is removed and
     * pids are killed, their resources go through release_token_async,
     * which will see token->space_dead, and those resources are freed
     * directly.  resources that may have already been on resources_rem and
     * the resource_thread may be in the middle of releasing one of them.
     * For any further async releases, resource_thread will see that the
     * lockspace is going away and will just free the resource.
     */

    purge_resource_orphans(sp->space_name);
    purge_resource_free(sp->space_name);

    close_event_fds(sp);

    close_task_aio(&task);
    return NULL;
}
```

lockspace线程做了如下事情：

- 通过wdmd与watchdog建立连接（connect_watchdog）
- 使用wdmd打开watchdog设备，设置过期时间，并开始续订喂狗（open_watchdog）
- 获取delta租约（delta_lease_acquire）
- 启动watchdog（activate_watchdog）
- 持续续约（delta_lease_renew）
- 续约成功，继续喂狗（update_watchdog）
- lockspace线程停止时，断开watchdog连接（disconnect_watchdog）

7、进入主循环-main_loop函数。

sanlock守护进程主循环

- src/main.c

1、循环poll每个套接字，如果有数据则执行相应的work函数，也就是之前设置好的process_connection函数。

```
static int main_loop(void)
{
    void (*workfn) (int ci);
    void (*deadfn) (int ci);
    struct space *sp, *safe;
    struct timeval now, last_check;
    int poll_timeout, check_interval;
    unsigned int ms;
    int i, rv, empty, check_all;
    char *check_buf = NULL;
    int check_buf_len = 0;
    uint64_t ebuf;

    gettimeofday(&last_check, NULL);
    poll_timeout = STANDARD_CHECK_INTERVAL;
    check_interval = STANDARD_CHECK_INTERVAL;

    while (1) {
        /* as well as the clients, check the eventfd */
        pollfd[client_maxi+1].fd = efd;
```

```
	pollfd[client_maxi+1].events = POLLIN;

	rv = poll(pollfd, client_maxi + 2, poll_timeout);
	if (rv == -1 && errno == EINTR)
	    continue;
	if (rv < 0) {
	    /* not sure */
	    log_client(0, 0, "poll err %d", rv);
	}
	for (i = 0; i <= client_maxi + 1; i++) {
	    /*
	     * This index for efd has no client array entry.  Its
	     * only purpose is to wake up this poll loop in which
	     * case we just clear any data and continue looking
	     * for other client entries that need processing.
	     */
	    if (pollfd[i].fd == efd) {
	        if (pollfd[i].revents & POLLIN) {
	            log_client(i, efd, "efd wake"); /* N.B. i is not a ci */
	            eventfd_read(efd, &ebuf);
	        }
	        continue;
	    }

	    /*
	     * FIXME? client_maxi is never reduced so over time we
	     * end up checking and skipping some number of unused
	     * client entries here which seems inefficient.
	     */
	    if (client[i].fd < 0)
	        continue;

	    if (pollfd[i].revents & POLLIN) {
	        workfn = client[i].workfn;
	        if (workfn)
	            workfn(i);
	    }
	    if (pollfd[i].revents & (POLLERR | POLLHUP | POLLNVAL)) {
	        log_client(i, client[i].fd, "poll dead");
	        deadfn = client[i].deadfn;
	        if (deadfn)
	            deadfn(i);
	    }
	}


	gettimeofday(&now, NULL);
	ms = time_diff(&last_check, &now);
	if (ms < check_interval) {
	    poll_timeout = check_interval - ms;
	    continue;
	}
	last_check = now;
	check_interval = STANDARD_CHECK_INTERVAL;

	/*
	 * check the condition of each lockspace,
	 * if pids are being killed, have pids all exited?
```

```
 * is its host_id being renewed?, if not kill pids
 */

pthread_mutex_lock(&spaces_mutex);
list_for_each_entry_safe(sp, safe, &spaces, list) {

    if (sp->killing_pids && all_pids_dead(sp)) {
        /*
         * move sp to spaces_rem so main_loop
         * will no longer see it.
         */
        log_space(sp, "set thread_stop");
        pthread_mutex_lock(&sp->mutex);
        sp->thread_stop = 1;
        deactivate_watchdog(sp);
        pthread_mutex_unlock(&sp->mutex);
        list_move(&sp->list, &spaces_rem);
        continue;
    }

    if (sp->killing_pids) {
        /*
         * continue to kill the pids with increasing
         * levels of severity until they all exit
         */
        kill_pids(sp);
        check_interval = RECOVERY_CHECK_INTERVAL;
        continue;
    }

    /*
     * check host_id lease renewal
     */

    if (sp->align_size > check_buf_len) {
        if (check_buf)
            free(check_buf);
        check_buf_len = sp->align_size;
        check_buf = malloc(check_buf_len);
    }
    if (check_buf)
        memset(check_buf, 0, check_buf_len);

    /*
     * check_our_lease will return check_all=1 if
     * all ls leases have been successfully read
     * since it was last called.  They are copied
     * into check_buf for check_other_leases.
     * So, check_other_leases will be called once
     * after each successful read of other leases.
     */
    check_all = 0;

    rv = check_our_lease(sp, &check_all, check_buf);
    if (rv)
        sp->renew_fail = 1;

    if (rv || sp->external_remove || (external_shutdown > 1)) {
```

```
137                 log_space(sp, "set killing_pids check %d remove %d",
138                     rv, sp->external_remove);
139                 sp->space_dead = 1;
140                 sp->killing_pids = 1;
141                 kill_pids(sp);
142                 check_interval = RECOVERY_CHECK_INTERVAL;
143
144             } else if (check_all) {
145                 check_other_leases(sp, check_buf);
146             }
147         }
148         empty = list_empty(&spaces);
149         pthread_mutex_unlock(&spaces_mutex);
150
151         if (external_shutdown && empty)
152             break;
153
154         if (external_shutdown == 1) {
155             log_debug("ignore shutdown, lockspace exists");
156             external_shutdown = 0;
157         }
158
159         free_lockspaces(0);
160         rem_resources();
161
162         gettimeofday(&now, NULL);
163         ms = time_diff(&last_check, &now);
164         if (ms < check_interval)
165             poll_timeout = check_interval - ms;
166         else
167             poll_timeout = 1;
168     }
169
170     free_lockspaces(1);
171
172     daemon_shutdown_reply();
173
174     return 0;
175 }
```

2. 检查每个lockspace中需要关闭的应用进程是否已关闭，如果关闭，关闭watchdog（deactivate_watchdog）；如果没有，继续升级手段进行关闭（kill_pids）。

src/watchdog.c

```
1  void deactivate_watchdog(struct space *sp)
2  {
3      int rv;
4
5      if (!com.use_watchdog)
6          return;
7
8      log_space(sp, "wdmd_test_live 0 0 to disable");
9
10     rv = wdmd_test_live(sp->wd_fd, 0, 0);
11     if (rv < 0) {
12         log_erros(sp, "wdmd_test_live in deactivate failed %d", rv);
13
14         /* We really want this to succeed to avoid a reset, so retry
```

```
        after a short delay in case the problem was transient... */

     usleep(500000);

     rv = wdmd_test_live(sp->wd_fd, 0, 0);
     if (rv < 0)
         log_erros(sp, "wdmd_test_live in deactivate 2 failed %d", rv);
  }

  wdmd_refcount_clear(sp->wd_fd);
}
```

3. 同时定时检查每个Lockspace的租约是否正常更新。

- src/lockspace.c

```
/*
 * check if our_host_id_thread has renewed within timeout
 */

int check_our_lease(struct space *sp, int *check_all, char *check_buf)
{
  int id_renewal_fail_seconds, id_renewal_warn_seconds;
  uint64_t last_success;
  int corrupt_result;
  int gap;

  pthread_mutex_lock(&sp->mutex);
  last_success = sp->lease_status.renewal_last_success;
  corrupt_result = sp->lease_status.corrupt_result;

  if (sp->lease_status.renewal_read_count > sp->lease_status.renewal_read_check) {
     /*
      * NB. it's unfortunate how subtle this is.
      * main loop will pass this buf to check_other_leases next
      */
     sp->lease_status.renewal_read_check = sp->lease_status.renewal_read_count;
     *check_all = 1;
     if (check_buf)
         memcpy(check_buf, sp->lease_status.renewal_read_buf, sp->align_size);
  }
  pthread_mutex_unlock(&sp->mutex);

  if (corrupt_result) {
     log_erros(sp, "check_our_lease corrupt %d", corrupt_result);
     return -1;
  }

  gap = monotime() - last_success;

  id_renewal_fail_seconds = calc_id_renewal_fail_seconds(sp->io_timeout);
  id_renewal_warn_seconds = calc_id_renewal_warn_seconds(sp->io_timeout);

  if (gap >= id_renewal_fail_seconds) {
     log_erros(sp, "check_our_lease failed %d", gap);
     return -1;
  }

  if (gap >= id_renewal_warn_seconds) {
```

```
        log_erros(sp, "check_our_lease warning %d last_success %llu",
                  gap, (unsigned long long)last_success);
    }

    if (com.debug_renew > 1) {
        log_space(sp, "check_our_lease good %d %llu",
                  gap, (unsigned long long)last_success);
    }

    return 0;
}
```

3. 如果更新租约失败，并超过一段时间则调用kill_pids函数终止持有该Lockspace的租约的进程。

- src/main.c

遍历这个Lockspace的所有client（每个client代表一个获取Resources的客户），向helper进程发送kill掉他的命令。

```
static void kill_pids(struct space *sp)
{
    struct client *cl;
    uint64_t now, last_success;
    int id_renewal_fail_seconds;
    int ci, sig;
    int do_kill, in_grace;

    /*
     * all remaining pids using sp are stuck, we've made max attempts to
     * kill all, don't bother cycling through them
     */
    if (sp->killing_pids > 1)
        return;

    id_renewal_fail_seconds = calc_id_renewal_fail_seconds(sp->io_timeout);

    /*
     * If we happen to renew our lease after we've started killing pids,
     * the period we allow for graceful shutdown will be extended. This
     * is an incidental effect, although it may be nice. The previous
     * behavior would still be ok, where we only ever allow up to
     * kill_grace_seconds for graceful shutdown before moving to sigkill.
     */
    pthread_mutex_lock(&sp->mutex);
    last_success = sp->lease_status.renewal_last_success;
    pthread_mutex_unlock(&sp->mutex);

    now = monotime();

    for (ci = 0; ci <= client_maxi; ci++) {
        do_kill = 0;

        cl = &client[ci];
        pthread_mutex_lock(&cl->mutex);

        if (!cl->used)
            goto unlock;

        if (cl->pid <= 0)
            goto unlock;

```

```
        /* NB this cl may not be using sp, but trying to
           avoid the expensive client_using_space check */

        if (cl->kill_count >= kill_count_max)
            goto unlock;

        if (cl->kill_count && (now - cl->kill_last < 1))
            goto unlock;

        if (!client_using_space(cl, sp))
            goto unlock;

        cl->kill_last = now;
        cl->kill_count++;

        /*
         * the transition from using killpath/sigterm to sigkill
         * is when now >=
         * last successful lease renewal +
         * id_renewal_fail_seconds +
         * kill_grace_seconds
         */

        in_grace = now < (last_success + id_renewal_fail_seconds + com.kill_grace_seconds);

        if (sp->external_remove || (external_shutdown > 1)) {
            sig = SIGKILL;
        } else if ((com.kill_grace_seconds > 0) && in_grace && cl->killpath[0]) {
            sig = SIGRUNPATH;
        } else if (in_grace) {
            sig = SIGTERM;
        } else {
            sig = SIGKILL;
        }

        /*
         * sigterm will be used in place of sigkill if restricted
         * sigkill will be used in place of sigterm if restricted
         */

        if ((sig == SIGKILL) && (cl->restricted & SANLK_RESTRICT_SIGKILL))
            sig = SIGTERM;

        if ((sig == SIGTERM) && (cl->restricted & SANLK_RESTRICT_SIGTERM))
            sig = SIGKILL;

        do_kill = 1;
    unlock:
        pthread_mutex_unlock(&cl->mutex);

        if (!do_kill)
            continue;

        send_helper_kill(sp, cl, sig);
      }
    }
```

4. 向helper进程发送kill命令，helper进程进行kill，但是并不向主进程返回执行结果。

- src/main.c

填充相应的数据结构，并向管道写入命令：

```
/*
 * We cannot block the main thread on this write, so the pipe is
 * NONBLOCK, and write fails with EAGAIN when the pipe is full.
 * With 512 msg size and 64k default pipe size, the pipe will be full
 * if we quickly send kill messages for 128 pids.  We retry
 * the kill once a second, so we'll retry the write again in
 * a second.
 *
 * By setting the pipe size to 1MB in setup_helper, we could quickly send 2048
 * msgs before getting EAGAIN.
 */

static void send_helper_kill(struct space *sp, struct client *cl, int sig)
{
    struct helper_msg hm;
    int rv;

    /*
     * We come through here once a second while the pid still has
     * leases.  We only send a single RUNPATH message, so after
     * the first RUNPATH goes through we set CL_RUNPATH_SENT to
     * avoid futher RUNPATH's.
     */

    if ((cl->flags & CL_RUNPATH_SENT) && (sig == SIGRUNPATH))
        return;

    if (helper_kill_fd == -1) {
        log_error("send_helper_kill pid %d no fd", cl->pid);
        return;
    }

    memset(&hm, 0, sizeof(hm));

    if (sig == SIGRUNPATH) {
        hm.type = HELPER_MSG_RUNPATH;
        memcpy(hm.path, cl->killpath, SANLK_HELPER_PATH_LEN-1);
        memcpy(hm.args, cl->killargs, SANLK_HELPER_ARGS_LEN-1);

        /* only include pid if it's requested as a killpath arg */
        if (cl->flags & CL_KILLPATH_PID)
            hm.pid = cl->pid;
    } else {
        hm.type = HELPER_MSG_KILLPID;
        hm.sig = sig;
        hm.pid = cl->pid;
    }

    log_erros(sp, "kill %d sig %d count %d", cl->pid, sig, cl->kill_count);

 retry:
    rv = write(helper_kill_fd, &hm, sizeof(hm));
    if (rv == -1 && errno == EINTR)
        goto retry;

```

```
    /* pipe is full, we'll try again in a second */
    if (rv == -1 && errno == EAGAIN) {
        helper_full_count++;
        log_space(sp, "send_helper_kill pid %d sig %d full_count %u",
                cl->pid, sig, helper_full_count);
        return;
    }

    /* helper exited or closed fd, quit using helper */
    if (rv == -1 && errno == EPIPE) {
        log_erros(sp, "send_helper_kill EPIPE");
        close_helper();
        return;
    }

    if (rv != sizeof(hm)) {
        /* this shouldn't happen */
        log_erros(sp, "send_helper_kill pid %d error %d %d",
                cl->pid, rv, errno);
        close_helper();
        return;
    }

    if (sig == SIGRUNPATH)
        cl->flags |= CL_RUNPATH_SENT;
}
```

5. 当进程被杀死后，主进程main_loop函数在与该进程连接poll套接字时，会收到POLLIN和POLLHUP事件，从而执行如下函数，释放client相关资源：

- src/main.c

```
static int main_loop(void)
{
...
            if (pollfd[i].revents & (POLLERR | POLLHUP | POLLNVAL)) {
                deadfn = client[i].deadfn;
                if (deadfn)
                    deadfn(i);
            }
...
}

void client_pid_dead(int ci)
{
    struct client *cl = &client[ci];
    int cmd_active;
    int i, pid;

    /* cmd_acquire_thread may still be waiting for the tokens
       to be acquired.  if it is, cl->pid_dead tells it to release them
       when finished.  Similarly, cmd_release_thread, cmd_inquire_thread
       are accessing cl->tokens */

    pthread_mutex_lock(&cl->mutex);
    if (!cl->used || cl->fd == -1 || cl->pid == -1) {
        /* should never happen */
        pthread_mutex_unlock(&cl->mutex);
        log_error("client_pid_dead %d,%d,%d u %d a %d s %d bad state",
```

```
            ci, cl->fd, cl->pid, cl->used,
            cl->cmd_active, cl->suspend);
        return;
    }

    log_debug("client_pid_dead %d,%d,%d cmd_active %d suspend %d",
          ci, cl->fd, cl->pid, cl->cmd_active, cl->suspend);

    if (cl->kill_count)
        log_error("dead %d ci %d count %d", cl->pid, ci, cl->kill_count);

    cmd_active = cl->cmd_active;
    pid = cl->pid;
    cl->pid = -1;
    cl->pid_dead = 1;

    /* when cmd_active is set and cmd_a,r,i_thread is done and takes
       cl->mutex to set cl->cmd_active to 0, it will see cl->pid_dead is 1
       and know they need to release cl->tokens and call client_free */

    /* make poll() ignore this connection */
    pollfd[ci].fd = -1;
    pollfd[ci].events = 0;

    pthread_mutex_unlock(&cl->mutex);

    /* it would be nice to do this SIGKILL as a confirmation that the pid
       is really gone (i.e. didn't just close the fd) if we always had root
       permission to do it */

    /* kill(pid, SIGKILL); */

    if (cmd_active) {
        log_debug("client_pid_dead %d,%d,%d defer to cmd %d",
              ci, cl->fd, pid, cmd_active);
        return;
    }

    /* use async release here because this is the main thread that we don't
       want to block doing disk lease i/o */

    pthread_mutex_lock(&cl->mutex);
    for (i = 0; i < cl->tokens_slots; i++) {
        if (cl->tokens[i]) {
            release_token_async(cl->tokens[i]);
            free(cl->tokens[i]);
        }
    }

    _client_free(ci);
    pthread_mutex_unlock(&cl->mutex);
}
```

wdmd守护进程初始化

wdmd/main.c

```
/* If wdmd exits abnormally, /dev/watchdog will eventually fire, and clients
   can detect wdmd is gone and begin to shut down cleanly ahead of the reset.
```

```
   But what if wdmd is restarted before the wd fires?  It will begin petting
   /dev/watchdog again, leaving the previous clients unprotected.  I don't
   know if this situation is important enough to try to prevent.  One way
   would be for wdmd to fail starting if it found a pid file left over from
   its previous run. */

int main(int argc, char *argv[])
{
    int do_probe = 0;
    int rv;

    while (1) {
        int c;
        int option_index = 0;

        static struct option long_options[] = {
            {"help",       no_argument,       0,  'h' },
            {"probe",      no_argument,       0,  'p' },
            {"dump",       no_argument,       0,  'd' },
            {"trytimeout", required_argument, 0,  't' },
            {"forcefire",  no_argument,       0,  'F' },
            {"version",    no_argument,       0,  'V' },
            {0,            0,                 0,  0 }
        };

        c = getopt_long(argc, argv, "hpdVDH:G:S:s:k:w:t:F",
                        long_options, &option_index);
        if (c == -1)
            break;

        switch (c) {
            case 'h':
                print_usage_and_exit(0);
                break;
        case 'p':
            do_probe = 1;
            break;
        case 't':
            do_probe = 1;
            try_timeout = atoi(optarg);
            break;
        case 'F':
            forcefire = 1;
            break;
        case 'd':
            print_debug_and_exit();
            break;
            case 'V':
                print_version_and_exit();
                break;
            case 'D':
                daemon_debug = 1;
                break;
            case 'G':
                socket_gname = strdup(optarg);
                break;
            case 'H':
                high_priority = atoi(optarg);
```

```
            break;
        case 'S':
            allow_scripts = atoi(optarg);
            break;
        case 's':
            scripts_dir = strdup(optarg);
            break;
        case 'k':
            kill_script_sec = atoi(optarg);
            break;
        case 'w':
            snprintf(option_path, WDPATH_SIZE, "%s", optarg);
            option_path[WDPATH_SIZE - 1] = '\0';
            break;
        }
    }

    if (forcefire && !do_probe) {
        fprintf(stderr, "Use force fire (-F) with a timeout (-t).\n");
        exit(EXIT_FAILURE);
    }

    if (do_probe) {
        rv = setup_shm();
        if (rv < 0) {
            fprintf(stderr, "cannot probe watchdog devices while wdmd is in use.\n");
            openlog("wdmd-probe", LOG_CONS | LOG_PID, LOG_DAEMON);
            syslog(LOG_ERR, "cannot probe watchdog devices while wdmd is in use.\n");
            exit(EXIT_FAILURE);
        }

        rv = probe_watchdog();

        close_shm();

        if (rv < 0)
            exit(EXIT_FAILURE);
        else
            exit(EXIT_SUCCESS);
    }

    socket_gid = group_to_gid(socket_gname);

    if (!daemon_debug) {
        if (daemon(0, 0) < 0) {
            fprintf(stderr, "cannot fork daemon\n");
            exit(EXIT_FAILURE);
        }
    }

    openlog("wdmd", LOG_CONS | LOG_PID, LOG_DAEMON);

    setup_priority();

    rv = lockfile();
    if (rv < 0)
        goto out;

```

```
    rv = setup_shm();
    if (rv < 0)
        goto out_lockfile;

    rv = setup_signals();
    if (rv < 0)
        goto out_shm;

    rv = setup_scripts();
    if (rv < 0)
        goto out_lockfile;

    rv = setup_files();
    if (rv < 0)
        goto out_scripts;

    rv = setup_clients();
    if (rv < 0)
        goto out_files;

    /* Sets watchdog_path */
    rv = setup_watchdog();
    if (rv < 0)
        goto out_clients;

    /* Sets watchdog_identity and itco */
    setup_identity(watchdog_path);

    log_info("wdmd started S%d H%d G%d using %s \"%s\"", allow_scripts, high_priority,
         socket_gid, watchdog_path, watchdog_identity[0] ? watchdog_identity : "unknown");

    daemon_setup_done = 1;

    rv = test_loop();

    close_watchdog();
  out_clients:
    close_clients();
  out_files:
    close_files();
  out_scripts:
    close_scripts();
  out_shm:
    close_shm();
  out_lockfile:
    unlink(lockfile_path);
  out:
    return rv;
}
```

libvirt中lockfailure相关的源码分析

libvirt中使用on_lockfailure标签来设置虚拟机的锁失败后的处理，其处理方法如下（src/conf/domain_conf.h）：

```
typedef enum {
    VIR_DOMAIN_LOCK_FAILURE_DEFAULT,
    VIR_DOMAIN_LOCK_FAILURE_POWEROFF,
    VIR_DOMAIN_LOCK_FAILURE_RESTART,
    VIR_DOMAIN_LOCK_FAILURE_PAUSE,
```

```
    VIR_DOMAIN_LOCK_FAILURE_IGNORE,

    VIR_DOMAIN_LOCK_FAILURE_LAST
} virDomainLockFailureAction;
```

目前只支持“默认”、“关机”和“暂停”的处理方式 ，如果是默认则不做任何处理，相当于让sanlock自己处理（src/locking/lock_driver_sanlock.c）。

```
static int
virLockManagerSanlockRegisterKillscript(int sock,
                                        const char *vmuri,
                                        const char *uuidstr,
                                        virDomainLockFailureAction action)
{
    g_auto(virBuffer) buf = VIR_BUFFER_INITIALIZER;
    char *path;
    char *args = NULL;
    int ret = -1;
    int rv;

    switch (action) {
    case VIR_DOMAIN_LOCK_FAILURE_DEFAULT:
        return 0;

    case VIR_DOMAIN_LOCK_FAILURE_POWEROFF:
    case VIR_DOMAIN_LOCK_FAILURE_PAUSE:
        break;

    case VIR_DOMAIN_LOCK_FAILURE_RESTART:
    case VIR_DOMAIN_LOCK_FAILURE_IGNORE:
    case VIR_DOMAIN_LOCK_FAILURE_LAST:
        virReportError(VIR_ERR_CONFIG_UNSUPPORTED,
                       _("Failure action %s is not supported by sanlock"),
                       virDomainLockFailureTypeToString(action));
        goto cleanup;
    }

    virBufferEscape(&buf, '\\', "\\ ", "%s", vmuri);
    virBufferAddLit(&buf, " ");
    virBufferEscape(&buf, '\\', "\\ ", "%s", uuidstr);
    virBufferAddLit(&buf, " ");
    virBufferEscape(&buf, '\\', "\\ ", "%s",
                    virDomainLockFailureTypeToString(action));

    /* Unfortunately, sanlock_killpath() does not use const for either
     * path or args even though it will just copy them into its own
     * buffers.
     */
    path = (char *) VIR_LOCK_MANAGER_SANLOCK_KILLPATH;
    args = virBufferContentAndReset(&buf);

    VIR_DEBUG("Register sanlock killpath: %s %s", path, args);

    /* sanlock_killpath() would just crop the strings */
    if (strlen(path) >= SANLK_HELPER_PATH_LEN) {
        virReportError(VIR_ERR_INTERNAL_ERROR,
                       _("Sanlock helper path is longer than %d: '%s'"),
                       SANLK_HELPER_PATH_LEN - 1, path);
        goto cleanup;
```

```
    }
    if (strlen(args) >= SANLK_HELPER_ARGS_LEN) {
        virReportError(VIR_ERR_INTERNAL_ERROR,
                       _("Sanlock helper arguments are longer than %d:"
                         " '%s'"),
                       SANLK_HELPER_ARGS_LEN - 1, args);
        goto cleanup;
    }

    if ((rv = sanlock_killpath(sock, 0, path, args)) < 0) {
        char *err = NULL;
        if (virLockManagerSanlockError(rv, &err)) {
            virReportError(VIR_ERR_INTERNAL_ERROR,
                           _("Failed to register lock failure action: %s"),
                           NULLSTR(err));
            VIR_FREE(err);
        } else {
            virReportSystemError(-rv, "%s",
                                 _("Failed to register lock failure"
                                   " action"));
        }
        goto cleanup;
    }

    ret = 0;

 cleanup:
    VIR_FREE(args);
    return ret;
}
```

如果是“关机”或“暂停”的处理方式则执行如下程序，且不可更改（src/locking/lock_driver_sanlock.c）：

```
#define VIR_LOCK_MANAGER_SANLOCK_KILLPATH LIBEXECDIR "/libvirt_sanlock_helper"
```

libvirt_sanlock_helper程序目前只支持“关机”和“暂停”（src/locking/sanlock_helper.c）。

```
int
main(int argc, char **argv)
{
    const char *uri;
    const char *uuid;
    virDomainLockFailureAction action;
    virConnectPtr conn = NULL;
    virDomainPtr dom = NULL;
    int ret = EXIT_FAILURE;

    int authTypes[] = {
        VIR_CRED_AUTHNAME,
        VIR_CRED_ECHOPROMPT,
        VIR_CRED_PASSPHRASE,
        VIR_CRED_NOECHOPROMPT,
    };
    virConnectAuth auth = {
        .credtype = authTypes,
        .ncredtype = G_N_ELEMENTS(authTypes),
        .cb = authCallback,
    };

```

```
    if (virGettextInitialize() < 0)
        exit(EXIT_FAILURE);

    if (getArgs(argc, argv, &uri, &uuid, &action) < 0)
        goto cleanup;

    if (!(conn = virConnectOpenAuth(uri, &auth, 0)) ||
        !(dom = virDomainLookupByUUIDString(conn, uuid)))
        goto cleanup;

    switch (action) {
    case VIR_DOMAIN_LOCK_FAILURE_POWEROFF:
        if (virDomainDestroy(dom) == 0 ||
            virDomainIsActive(dom) == 0)
            ret = EXIT_SUCCESS;
        break;

    case VIR_DOMAIN_LOCK_FAILURE_PAUSE:
        if (virDomainSuspend(dom) == 0)
            ret = EXIT_SUCCESS;
        break;

    case VIR_DOMAIN_LOCK_FAILURE_DEFAULT:
    case VIR_DOMAIN_LOCK_FAILURE_RESTART:
    case VIR_DOMAIN_LOCK_FAILURE_IGNORE:
    case VIR_DOMAIN_LOCK_FAILURE_LAST:
        fprintf(stderr, _("unsupported failure action: '%s'\n"),
                virDomainLockFailureTypeToString(action));
        break;
    }

 cleanup:
    virObjectUnref(dom);
    if (conn)
        virConnectClose(conn);

    return ret;
}
```

主要流程总结

1、sanlock主函数启动，等待客户端请求命令
2、wdmd主函数启动，等待客户端请求
3、libvirt发送命令给sanlock设置killpath
4、libvirt发送命令给sanlock添加lockspace，sanlock创建lockspace线程
5、lockspace线程做如下事情：
（1）通过wdmd与watchdog建立连接
（2）通过wdmd打开watchdog设备，设置过期时间
（3）获取租约
（4）通过wdmd启动watchdog
（5）周期性续约
（6）续约成功，通过wdmd检查wdmd连接是否过期，没过期则喂狗
（7）续约失败，返回结果，继续下次续约
6、sanlock主循环检查租约，如果检查失败，开始优雅关闭应用进程
7、wdmd检查连接过期，停止喂狗

8、sanlock主循环检查应用进程是否关闭，如果没关闭，继续kill应用进程；如果关闭，通过wdmd关闭watchdog
9、如果watchdog没关闭且过期，重启节点