《openEuler内核编程》

课程讲稿

第四章 第3讲

内核进程隔离

软件所制

第四章 第3讲 内核进程隔离

**学时：**1学时

**教学目的：**系统学习内核进程隔离

**PPT讲稿：**



今天的内容是第三讲，内核进程隔离。



这一节分三部分，我们将主要了解一下进程的命名空间、Proc文件系统，Cgroup机制和LXC的内容。



进程隔离是为保护操作系统中进程互不干扰而设计的一组不同硬件和软件的技术。避免进程A写入进程B的情况发生。 进程的隔离实现，使用了虚拟地址空间。进程A的虚拟地址和进程B的虚拟地址不同，这样就防止进程A将数据信息写入进程B。

进程隔离的安全性通过禁止进程间内存的访问可以方便实现

相比之下，一些不安全的操作系统（例如DOS）能够允许任何进程对其他进程的内存进行写操作。

主机的虚拟化框架能在同一物理机器上实现提供不同的用户操作系统视图，使用ＶＭＷａｒｅ虽然也可以提供一种解决方案，但是资源分配不是很好，而且是在以一种ＯＳ为基础,在其上模拟实现另一种ＯＳ，命名空间为系统的虚拟化提供了不同的实现思路，一台主机可以同时运行多个内核，即同时运行多个操作系统（同种操作系 统），全局系统资源通过命名空间隔离开，各个容器相当于一个独立的ＯＳ，这样的实现更加灵活，命名空间也可以组成层次结构。

每一个进程其所包含的命名空间都被抽象成一个nsproxy指针，共享同一个命名空间的进程指向同一个指针。指针的结构通过引用计数（count）来确定使用者数目。

当一个进程其所处的用户空间发生变化的时候就发生分裂。通过复制一份老的命名空间数据结构然后做一些简单的修改，接着赋值给相应的进程。

Uts,ipc,mnt,pid,net都是子系统

看了上面的数据结构，我们就会基本明白，命名空间本身只是一个框架，需要其他实行虚拟化的子系统实现自己的命名空间。这些子系统的对象就不再是全局维护的一份结构了，而是和进程的用户空间数目一致，每一个命名空间都会有对象的一个具体实例。目前Linux系统实现的命名空间子系统主要有UTS、IPC、MNT、PID以及NET网络子模块。实际上，Linux 内核实现 namespace 的主要目的就是为了实现轻量级虚拟化（容器）服务。在同一个 namespace 下的进程可以感知彼此的变化，而对外界的进程一无所知。这样就可以让容器中的进程产生错觉，仿佛自己置身于一个独立的系统环境中，以此达到独立和隔离的目的。

namespace 的 API 包括 clone()、setns() 以及 unshare()，还有 /proc 下的部分文件。为了确定隔离的到底是哪种 namespace，在使用这些 API 时，通常需要指定以下六个常数的一个或多个，通过|（位或）操作来实现。你可能已经在上面的表格中注意到，这六个参数分别是 CLONE\_NEWIPC、CLONE\_NEWNS、CLONE\_NEWNET、CLONE\_NEWPID、CLONE\_NEWUSER 和 CLONE\_NEWUTS。

clone系统调用创建新进程的同时会创建namespace，传递相关参数

从 3.8 版本的内核开始，用户就可以在 /proc/[pid]/ns 文件下看到指向不同 namespace 号的文件，效果如下所示，形如 [4026531839] 者即为 namespace 号。

如果两个进程指向的 namespace 编号相同，就说明他们在同一个 namespace 下，否则则在不同 namespace 里面。/proc/[pid]/ns 的另外一个作用是，一旦文件被打开，只要打开的文件描述符（fd）存在，那么就算 PID 所属的所有进程都已经结束，创建的 namespace 就会一直存在。那如何打开文件描述符呢？把 /proc/[pid]/ns 目录挂载起来就可以达到这个效果，命令如下。

刚提到，在进程都结束的情况下，也可以通过挂载的形式把 namespace 保留下来，保留 namespace 的目的自然是为以后有进程加入做准备。通过 setns() 系统调用，你的进程从原先的 namespace 加入我们准备好的新 namespace，使用方法如下。

最后要提的系统调用是 unshare()，它跟 clone() 很像，不同的是，unshare() 运行在原先的进程上，不需要启动一个新进程，使用方法如下。

调用 unshare() 的主要作用就是不启动一个新进程就可以起到隔离的效果，相当于跳出原先的 namespace 进行操作。这样，你就可以在原进程进行一些需要隔离的操作。Linux 中自带的 unshare 命令，就是通过 unshare() 系统调用实现的，有兴趣的可以在网上搜索一下这个命令的作用。

UTS namespace 提供了主机名和域名的隔离，这样每个容器就拥有独立的主机名和域名了，在网络上就可以被视为一个独立的节点，在容器中对 hostname 的命名不会对宿主机造成任何影响。

在创建新的uts namespace时，需要设置单独的hostname，从而不影响到原进程

容器中进程间通信采用的方法包括常见的信号量、消息队列和共享内存。然而与虚拟机不同的是，容器内部进程间通信对宿主机来说，实际上是具有相同 PID namespace 中的进程间通信，因此需要一个唯一的标识符来进行区别。申请 IPC 资源就申请了这样一个全局唯一的 32 位 ID，所以 IPC namespace 中实际上包含了系统 IPC 标识符以及实现 POSIX 消息队列的文件系统。在同一个 IPC namespace 下的进程彼此可见，而与其他的 IPC namespace 下的进程则互相不可见。



ID namespace 隔离非常实用，它对进程 PID 重新标号，即两个不同 namespace 下的进程可以有同一个 PID。每个 PID namespace 都有自己的计数程序。内核为所有的 PID namespace 维护了一个树状结构，最顶层的是系统初始时创建的，我们称之为 root namespace。他创建的新 PID namespace 就称之为 child namespace（树的子节点），而原先的 PID namespace 就是新创建的 PID namespace 的 parent namespace（树的父节点）。通过这种方式，不同的 PID namespaces 会形成一个等级体系。所属的父节点可以看到子节点中的进程，并可以通过信号等方式对子节点中的进程产生影响。反过来，子节点不能看到父节点 PID namespace 中的任何内容。

每当在内核中新建一个进程时，都会在全局内创建一个唯一的描述符来描述该进程。在没有命名空间的情况下，用一个单项递增的数字作为进程号即可。为了实现进程隔离，引入层次 化的命名空间。由于子命名空间中的所有进程在父命名空间中都是可见的，导致在子命名空间中新建一个进程会出现在往上的每层命名空间都有一个唯一的进程号 描述。这样，一个进程在整个系统中可能有多个进程描述符，在一般的使用时，显示的进程号往往是该进程对应到顶层命名空间的进程号。每个容器包括一定独立的资源，隔离开来进行管理。



子命名空间会复制一些父命名空间的数据，且进程必须是父命名空间的子集。



当我们新建一个 PID namespace 时，默认启动的进程 PID 为 1。我们知道，在传统的 UNIX 系统中，PID 为 1 的进程是 init，地位非常特殊。他作为所有进程的父进程，维护一张进程表，不断检查进程的状态，一旦有某个子进程因为程序错误成为了“孤儿”进程，init 就会负责回收资源并结束这个子进程。所以在你要实现的容器中，启动的第一个进程也需要实现类似 init 的功能，维护所有后续启动进程的运行状态。

明白了内核设计的良苦用心。PID namespace 维护这样一个树状结构，非常有利于系统的资源监控与回收。Docker 启动时，第一个进程也是这样，实现了进程监控和资源回收，它就是 dockerinit。

PID namespace 中的 init 进程如此特殊，自然内核也为他赋予了特权——信号屏蔽。如果 init 中没有写处理某个信号的代码逻辑，那么与 init 在同一个 PID namespace 下的进程（即使有超级权限）发送给它的该信号都会被屏蔽。这个功能的主要作用是防止 init 进程被误杀。

那么其父节点 PID namespace 中的进程发送同样的信号会被忽略吗？父节点中的进程发送的信号，如果不是 SIGKILL（销毁进程）或 SIGSTOP（暂停进程）也会被忽略。但如果发送 SIGKILL 或 SIGSTOP，子节点的 init 会强制执行（无法通过代码捕捉进行特殊处理），也就是说父节点中的进程有权终止子节点中的进程。

一旦 init 进程被销毁，同一 PID namespace 中的其他进程也会随之接收到 SIGKILL 信号而被销毁。理论上，该 PID namespace 自然也就不复存在了。但是如果 /proc/[pid]/ns/pid 处于被挂载或者打开状态，namespace 就会被保留下来。然而，保留下来的 namespace 无法通过 setns() 或者 fork() 创建进程，所以实际上并没有什么作用。

我们常说，Docker 一旦启动就有进程在运行，不存在不包含任何进程的 Docker，也就是这个道理

为什么创建其他 namespace 时 unshare() 和 setns() 会直接进入新的 namespace 而唯独 PID namespace 不是如此呢？因为调用 getpid() 函数得到的 PID 是根据调用者所在的 PID namespace 而决定返回哪个 PID，进入新的 PID namespace 会导致 PID 产生变化。而对用户态的程序和库函数来说，他们都认为进程的 PID 是一个常量，PID 的变化会引起这些进程奔溃。

换句话说，一旦程序进程创建以后，那么它的 PID namespace 的关系就确定下来了，进程不会变更他们对应的 PID namespace。

19:

Mount namespace 通过隔离文件系统挂载点对隔离文件系统提供支持，它是历史上第一个 Linux namespace，所以它的标识位比较特殊，就是 CLONE\_NEWNS。

20:

一个挂载状态可能为如下的其中一种：

共享挂载（shared）、从属挂载（slave）、共享 / 从属挂载（shared and slave）、私有挂载（private）和不可绑定挂载（unbindable）

21:

通过上节，我们了解了 PID namespace，当我们兴致勃勃地在新建的 namespace 中启动一个“Apache”进程时，却出现了“80 端口已被占用”的错误，原来主机上已经运行了一个“Apache”进程。怎么办？这就需要用到 network namespace 技术进行网络隔离啦。

Network namespace 主要提供了关于网络资源的隔离，包括网络设备、IPv4 和 IPv6 协议栈、IP 路由表、防火墙、/proc/net 目录、/sys/class/net 目录、端口（socket）等等。一个物理的网络设备最多存在在一个 network namespace 中，你可以通过创建 veth pair（虚拟网络设备对：有两端，类似管道，如果数据从一端传入另一端也能接收到，反之亦然）在不同的 network namespace 间创建通道，以此达到通信的目的。

一般情况下，物理网络设备都分配在最初的 root namespace（表示系统默认的 namespace，在 PID namespace 中已经提及）中。但是如果你有多块物理网卡，也可以把其中一块或多块分配给新创建的 network namespace。需要注意的是，当新创建的 network namespace 被释放时（所有内部的进程都终止并且 namespace 文件没有被挂载或打开），在这个 namespace 中的物理网卡会返回到 root namespace 而非创建该进程的父进程所在的 network namespace。

当我们说到 network namespace 时，其实我们指的未必是真正的网络隔离，而是把网络独立出来，给外部用户一种透明的感觉，仿佛跟另外一个网络实体在进行通信。为了达到这个目的，容器的经典做法就是创建一个 veth pair，一端放置在新的 namespace 中，通常命名为 eth0，一端放在原先的 namespace 中连接物理网络设备，再通过网桥把别的设备连接进来或者进行路由转发，以此网络实现通信的目的。

也许有读者会好奇，在建立起 veth pair 之前，新旧 namespace 该如何通信呢？答案是 pipe（管道）。我们以 Docker Daemon 在启动容器 dockerinit 的过程为例。Docker Daemon 在宿主机上负责创建这个 veth pair，通过 netlink 调用，把一端绑定到 docker0 网桥上，一端连进新建的 network namespace 进程中。建立的过程中，Docker Daemon 和 dockerinit 就通过 pipe 进行通信，当 Docker Daemon 完成 veth-pair 的创建之前，dockerinit 在管道的另一端循环等待，直到管道另一端传来 Docker Daemon 关于 veth 设备的信息，并关闭管道。dockerinit 才结束等待的过程，并把它的“eth0”启动起来。整个效果类似下图所示。

22:

User namespace 主要隔离了安全相关的标识符（identifiers）和属性（attributes），包括用户 ID、用户组 ID、root 目录、[key](http://man7.org/linux/man-pages/man2/keyctl.2.html)（指密钥）以及[特殊权限](http://man7.org/linux/man-pages/man7/capabilities.7.html)。说得通俗一点，一个普通用户的进程通过clone() 创建的新进程在新user namespace 中可以拥有不同的用户和用户组。这意味着一个进程在容器外属于一个没有特权的普通用户，但是他创建的容器进程却属于拥有所有权限的超级用户，这个技术为容器提供了极大的自由。

23:

Proc文件系统是在内存中，而不是在硬盘上面的，所以是伪文件。

24:

/proc 的文件可以用于访问有关内核的状态、计算机的属性、正在运行的进程的 状态等信息。大部分 /proc 中的文件和目录提供系统物理环境最新的信息。尽管 /proc 中的文件是虚拟的，但它们仍可以使用任何文件编辑器或像'more', 'less'或 'cat'这样的程序来查看。当编辑程序试图打开一个虚拟文件时，这个文件就通过内核中的信息被凭空地创建了。

proc 文件系统可以被用于收集有用的关于系统和运行中的内核的信息。

25:

由于系统的信息，如进程，是动态改变的，所以用户或应用程序读取proc文件时，proc文件系统是动态从系统内核读出所需信息并提交的。

上面讨论的大部分 proc 的文件是只读的。而实际上 proc 文件系统通过 proc 中可读写的文件提供了对内核的交互机制。写这些文件可以改变内核 的状态，因而要慎重改动这些文件。

在/proc下还有三个很重要的目录：net，scsi和sys。 Sys目录是可写的，可以通过它来访问或修改内核的参数，改变内核行为.而net和scsi则依赖于内核配置。例如，如果系统不支持scsi，则scsi 目录不存在。

26:

接下来我们介绍一下cgroup机制

Cgroups是control groups的缩写，是Linux内核提供的一种可以限制、记录、隔离进程组所使用的物理资源（如：cpu,memory,IO等等）的机制。最初由google的工程师提出，后来被整合进Linux内核。Cgroups也是LXC为实现虚拟化所使用的资源管理手段，可以说没有cgroups就没有LXC。

Cgroups以分组的形式对进程使用系统资源的行为进行管理和控制。也就是说，用户通过cgroup对所有进程进行分组，再对该分组整体进行资源的分配和控制。

27:

Cgroups最初的目标是为资源管理提供的一个统一的框架，既整合现有的cpuset等子系统，也为未来开发新的子系统提供接口。现在的cgroups适用于多种应用场景，从单个进程的资源控制，到实现操作系统层次的虚拟化

28:

在cgroups中，任务就是系统的一个进程。

控制族群就是一组按照某种标准划分的进程。Cgroups中的资源控制都是以控制族群为单位实现。一个进程可以加入到某个控制族群，也从一个进程组迁移到另一个控制族群。一个进程组的进程可以使用cgroups以控制族群为单位分配的资源，同时受到cgroups以控制族群为单位设定的限制。

控制族群可以组织成hierarchical的形式，既一颗控制族群树。控制族群树上的子节点控制族群是父节点控制族群的孩子，继承父控制族群的特定的属性。

一个子系统就是一个资源控制器，比如cpu子系统就是控制cpu时间分配的一个控制器。子系统必须附加（attach）到一个层级上才能起作用，一个子系统附加到某个层级以后，这个层级上的所有控制族群都受到这个子系统的控制。

29:

每次在系统中创建新层级时，该系统中的所有任务都是那个层级的默认 cgroup（我们称之为 root cgroup ，此cgroup在创建层级时自动创建，后面在该层级中创建的cgroup都是此cgroup的后代）的初始成员。

一个子系统最多只能附加到一个层级。一个层级可以附加多个子系统。一个任务可以是多个cgroup的成员，但是这些cgroup必须在不同的层级。系统中的进程（任务）创建子进程（任务）时，该子任务自动成为其父进程所在 cgroup 的成员。然后可根据需要将该子任务移动到不同的 cgroup 中，但开始时它总是继承其父任务的cgroup。

30:

系统内的所有进程形成一个根进程组，根据系统对资源的需求，这个根进程组将被进一步细分为子进程组，子进程组内的进程是根进程组内进程的子集。经过细分，形成具有层次的进程组树。

31:

由图可以看出，cgroup在用户态提供统一的用户接口，而每个子系统对资源的控制功能则通过其钩子函数实现。这样使得cgroup在上层是一个统一的框架，而下层则可以实现多种资源的控制。

cgroup在Linux内核中是以文件系统的形式存在的，不过cgroup对应的这种文件系统与proc文件系统类似，都是只存在于内存中的“虚拟”文件系

32:

Cgroup的子系统包括blkio、cpu、cpuacct、cpuset、devices、freezer、memory、net\_cls和ns

33:

LXC可以在操作系统层次上为进程提供的虚拟的执行环境，一个虚拟的执行环境就是一个容器。可以为容器绑定特定的cpu和memory节点，分配特定比例的cpu时间、IO时间，限制可以使用的内存大小（包括内存和swap空间），提供device访问控制，提供独立的namespace（网络、pid、ipc、mnt、uts

34:

LXC是所谓的操作系统层次的虚拟化技术，与传统的HAL（硬件抽象层）层次的虚拟化技术相比有以下优势：是更小的虚拟化开销（LXC的诸多特性基本由内核特供，而内核实现这些特性只有极少的花费，具体分析有时间再说）。且便于快速部署。利用LXC来隔离特定应用，只需要安装LXC，即可使用LXC相关命令来创建并启动容器来为应用提供虚拟执行环境。传统的虚拟化技术则需要先创建虚拟机，然后安装系统，再部署应用。

LXC跟其他操作系统层次的虚拟化技术相比，最大的优势在于LXC被整合进内核，不用单独为内核打补丁。

35:

完