## namespace实现的隔离：

参考链接：<https://blog.csdn.net/futurewu/article/details/78174130>



1、UTS namespace提供了主机名和域名的隔离，这样每个容器就可以拥有了独立的主机名和域名，在网络上可以被视作一个独立的节点而非宿主机上的一个进程。

2、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。

3、PID 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。

4、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。

5、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。隔离后，不同mount namespace中的文件结构发生变化也互不影响。你可以通过/proc/[pid]/mounts查看到所有挂载在当前namespace中的文件系统，还可以通过/proc/[pid]/mountstats看到mount namespace中文件设备的统计信息，包括挂载文件的名字、文件系统类型、挂载位置等等。

6、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（指密钥）以及特殊权限。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。

## cgroups实现的隔离

参考链接：<https://www.cnblogs.com/sammyliu/p/5886833.html>

Linux Cgroup 可​​​让​​​您​​​为​​​系​​​统​​​中​​​所​​​运​​​行​​​任​​​务​​​（进​​​程​​​）的​​​用​​​户​​​定​​​义​​​组​​​群​​​分​​​配​​​资​​​源​​​ — 比​​​如​​​ CPU 时​​​间​​​、​​​系​​​统​​​内​​​存​​​、​​​网​​​络​​​带​​​宽​​​或​​​者​​​这​​​些​​​资​​​源​​​的​​​组​​​合​​​。​​​您​​​可​​​以​​​监​​​控​​​您​​​配​​​置​​​的​​​ cgroup，拒​​​绝​​​ cgroup 访​​​问​​​某​​​些​​​资​​​源​​​，甚​​​至​​​在​​​运​​​行​​​的​​​系​​​统​​​中​​​动​​​态​​​配​​​置​​​您​​​的​​​ cgroup。所以，可以将 controll groups 理解为 controller （system resource） （for） （process）groups，也就是是说它以一组进程为目标进行系统资源分配和控制。

它主要提供了如下功能：

Resource limitation: 限制资源使用，比如内存使用上限以及文件系统的缓存限制。

Prioritization: 优先级控制，比如：CPU利用和磁盘IO吞吐。

Accounting: 一些审计或一些统计，主要目的是为了计费。

Control: 挂起进程，恢复执行进程。

我们看到 /sys/fs/cgroup 目录中有若干个子目录，我们可以认为这些都是受 cgroups 控制的资源以及这些资源的信息。

blkio — 这​​​个​​​子​​​系​​​统​​​为​​​块​​​设​​​备​​​设​​​定​​​输​​​入​​​/输​​​出​​​限​​​制​​​，比​​​如​​​物​​​理​​​设​​​备​​​（磁​​​盘​​​，固​​​态​​​硬​​​盘​​​，USB 等​​​等​​​）。

cpu — 这​​​个​​​子​​​系​​​统​​​使​​​用​​​调​​​度​​​程​​​序​​​提​​​供​​​对​​​ CPU 的​​​ cgroup 任​​​务​​​访​​​问​​​。​​​

cpuacct — 这​​​个​​​子​​​系​​​统​​​自​​​动​​​生​​​成​​​ cgroup 中​​​任​​​务​​​所​​​使​​​用​​​的​​​ CPU 报​​​告​​​。​​​

cpuset — 这​​​个​​​子​​​系​​​统​​​为​​​ cgroup 中​​​的​​​任​​​务​​​分​​​配​​​独​​​立​​​ CPU（在​​​多​​​核​​​系​​​统​​​）和​​​内​​​存​​​节​​​点​​​。​​​

devices — 这​​​个​​​子​​​系​​​统​​​可​​​允​​​许​​​或​​​者​​​拒​​​绝​​​ cgroup 中​​​的​​​任​​​务​​​访​​​问​​​设​​​备​​​。​​​

freezer — 这​​​个​​​子​​​系​​​统​​​挂​​​起​​​或​​​者​​​恢​​​复​​​ cgroup 中​​​的​​​任​​​务​​​。​​​

memory — 这​​​个​​​子​​​系​​​统​​​设​​​定​​​ cgroup 中​​​任​​​务​​​使​​​用​​​的​​​内​​​存​​​限​​​制​​​，并​​​自​​​动​​​生​​​成​​​​​内​​​存​​​资​​​源使用​​​报​​​告​​​。​​​

net\_cls — 这​​​个​​​子​​​系​​​统​​​使​​​用​​​等​​​级​​​识​​​别​​​符​​​（classid）标​​​记​​​网​​​络​​​数​​​据​​​包​​​，可​​​允​​​许​​​ Linux 流​​​量​​​控​​​制​​​程​​​序​​​（tc）识​​​别​​​从​​​具​​​体​​​ cgroup 中​​​生​​​成​​​的​​​数​​​据​​​包​​​。​​​

net\_prio — 这个子系统用来设计网络流量的优先级

hugetlb — 这个子系统主要针对于HugeTLB系统进行限制，这是一个大页文件系统。

## 数据结构定义：

**SYSCALL\_DEFINE2**

SYSCALL\_DEFINEx里面的x代表的是系统调用参数个数。

参数依次为：系统调用名、参数类型、参数名称、类型、名称。

**clockid\_t**

clockid\_t用于指定计时时钟的类型，有以下4种：

CLOCK\_REALTIME:系统实时时间,随系统实时时间改变而改变,即从UTC1970-1-1 0:0:0开始计时,中间时刻如果系统时间被用户该成其他,则对应的时间相应改变

CLOCK\_MONOTONIC:从系统启动这一刻起开始计时,不受系统时间被用户改变的影响

CLOCK\_PROCESS\_CPUTIME\_ID:本进程到当前代码系统CPU花费的时间

CLOCK\_THREAD\_CPUTIME\_ID:本线程到当前代码系统CPU花费的时间

**timespec**

struct timespec 用来存储当前的时间，其结构如下：

struct timespec

{

time\_t tv\_sec; /\* seconds \*/

long tv\_nsec; /\* nanoseconds \*/

};

#define [**CLOCKFD\_MASK**](https://elixir.bootlin.com/linux/v5.1.16/ident/CLOCKFD_MASK) ([**CPUCLOCK\_PERTHREAD\_MASK**](https://elixir.bootlin.com/linux/v5.1.16/ident/CPUCLOCK_PERTHREAD_MASK)|[**CPUCLOCK\_CLOCK\_MASK**](https://elixir.bootlin.com/linux/v5.1.16/ident/CPUCLOCK_CLOCK_MASK))

#define [**CPUCLOCK\_PERTHREAD\_MASK**](https://elixir.bootlin.com/linux/v5.1.16/ident/CPUCLOCK_PERTHREAD_MASK) 4

#define [**CPUCLOCK\_CLOCK\_MASK**](https://elixir.bootlin.com/linux/v5.1.16/ident/CPUCLOCK_CLOCK_MASK) 3

#define [**CLOCKFD**](https://elixir.bootlin.com/linux/v5.1.16/ident/CLOCKFD) [**CPUCLOCK\_MAX**](https://elixir.bootlin.com/linux/v5.1.16/ident/CPUCLOCK_MAX)

const struct [**k\_clock**](https://elixir.bootlin.com/linux/v5.1.16/ident/k_clock) [**clock\_posix\_dynamic**](https://elixir.bootlin.com/linux/v5.1.16/ident/clock_posix_dynamic) = {

.clock\_getres = [**pc\_clock\_getres**](https://elixir.bootlin.com/linux/v5.1.16/ident/pc_clock_getres),

.**[clock\_set](https://elixir.bootlin.com/linux/v5.1.16/ident/clock_set)** = [**pc\_clock\_settime**](https://elixir.bootlin.com/linux/v5.1.16/ident/pc_clock_settime),

.clock\_get = [**pc\_clock\_gettime**](https://elixir.bootlin.com/linux/v5.1.16/ident/pc_clock_gettime),

.clock\_adj = [**pc\_clock\_adjtime**](https://elixir.bootlin.com/linux/v5.1.16/ident/pc_clock_adjtime),

};

static const struct [**k\_clock**](https://elixir.bootlin.com/linux/v5.1.16/ident/k_clock) \* const [**posix\_clocks**](https://elixir.bootlin.com/linux/v5.1.16/ident/posix_clocks)[] = {

[[**CLOCK\_REALTIME**](https://elixir.bootlin.com/linux/v5.1.16/ident/CLOCK_REALTIME)] = &**[clock\_realtime](https://elixir.bootlin.com/linux/v5.1.16/ident/clock_realtime)**,

[[**CLOCK\_MONOTONIC**](https://elixir.bootlin.com/linux/v5.1.16/ident/CLOCK_MONOTONIC)] = &**[clock\_monotonic](https://elixir.bootlin.com/linux/v5.1.16/ident/clock_monotonic)**,

[[**CLOCK\_PROCESS\_CPUTIME\_ID**](https://elixir.bootlin.com/linux/v5.1.16/ident/CLOCK_PROCESS_CPUTIME_ID)] = &**[clock\_process](https://elixir.bootlin.com/linux/v5.1.16/ident/clock_process)**,

[[**CLOCK\_THREAD\_CPUTIME\_ID**](https://elixir.bootlin.com/linux/v5.1.16/ident/CLOCK_THREAD_CPUTIME_ID)] = &**[clock\_thread](https://elixir.bootlin.com/linux/v5.1.16/ident/clock_thread)**,

[[**CLOCK\_MONOTONIC\_RAW**](https://elixir.bootlin.com/linux/v5.1.16/ident/CLOCK_MONOTONIC_RAW)] = &**[clock\_monotonic\_raw](https://elixir.bootlin.com/linux/v5.1.16/ident/clock_monotonic_raw)**,

[[**CLOCK\_REALTIME\_COARSE**](https://elixir.bootlin.com/linux/v5.1.16/ident/CLOCK_REALTIME_COARSE)] = &**[clock\_realtime\_coarse](https://elixir.bootlin.com/linux/v5.1.16/ident/clock_realtime_coarse)**,

[[**CLOCK\_MONOTONIC\_COARSE**](https://elixir.bootlin.com/linux/v5.1.16/ident/CLOCK_MONOTONIC_COARSE)] = &**[clock\_monotonic\_coarse](https://elixir.bootlin.com/linux/v5.1.16/ident/clock_monotonic_coarse)**,

[[**CLOCK\_BOOTTIME**](https://elixir.bootlin.com/linux/v5.1.16/ident/CLOCK_BOOTTIME)] = &**[clock\_boottime](https://elixir.bootlin.com/linux/v5.1.16/ident/clock_boottime)**,

[[**CLOCK\_REALTIME\_ALARM**](https://elixir.bootlin.com/linux/v5.1.16/ident/CLOCK_REALTIME_ALARM)] = &**[alarm\_clock](https://elixir.bootlin.com/linux/v5.1.16/ident/alarm_clock)**,

[[**CLOCK\_BOOTTIME\_ALARM**](https://elixir.bootlin.com/linux/v5.1.16/ident/CLOCK_BOOTTIME_ALARM)] = &**[alarm\_clock](https://elixir.bootlin.com/linux/v5.1.16/ident/alarm_clock)**,

[[**CLOCK\_TAI**](https://elixir.bootlin.com/linux/v5.1.16/ident/CLOCK_TAI)] = &**[clock\_tai](https://elixir.bootlin.com/linux/v5.1.16/ident/clock_tai)**,

};

## 系统调用分析

系统调用表：<https://elixir.bootlin.com/linux/v5.1.16/source/include/uapi/asm-generic/unistd.h#L787>

### 1、系统调用号：404 ：[clock\_settime](https://elixir.bootlin.com/linux/v5.1.16/source/kernel/time/posix-timers.c#L1017)

功能：clock\_settime系统调用可以设置系统时间秒数与纳秒数。

参数：const clockid\_t which\_clock 时钟类型

const struct \_\_kernel\_timespec \_\_user \* tp 要修改的时间，结构中包括秒、纳秒信息。

实现流程：

1）clockid\_t转换为kclock类型。clockid\_t为参数类型，kclock结构中包括一系列函数指针。

2）如果转换结果为NULL或者成员函数clock\_set为NULL，返回 –EINVAL；

3）timespec转换为timespec64类型，如果返回值非0，返回–EINVAL；

4）调用k\_clock成员函数clock\_set函数设置时间，并返回该函数返回值。该函数又通过调用do\_sys\_settimeofday64实现。

隔离性：I （LI、VI）

对于 Docker 容器而言，容器与宿主的时钟是一致的，所以各个容器之间的时钟也没有隔离。同时在do\_settimeofday64()函数中，存在对锁的竞争。

### 2、系统调用号：405 ：[clock\_adjtime](https://elixir.bootlin.com/linux/v5.1.16/source/kernel/time/posix-timers.c" \l "L1062)

功能：该函数可以根据参数缓慢的修正系统时钟（CLOCK\_REALTIME那个）。

参数：

const clockid\_t which\_clock 时钟类型，只有CLOCK\_REALTIME有效。

struct \_\_kernel\_timex \_\_user \* utx 包含各种标志位，以及调整的时间。

实现流程：

1）将utx从用户空间拷贝到内核空间ktx；

2）调用do\_clock\_adjtime函数修正时间，其参数为时钟类型which\_clock、内核空间的ktx。其流程为：

根据whick\_clock类型生成kclock类型结构，然后调用kclock函数成员clock\_adj，对CLOCK\_REALTIME类型的时钟，该函数指针指向posix \_clock\_realtime\_adj函数，而该函数又通过调用do\_adjtimex实现。

3）将ktx拷贝到用户空间utx。

隔离性：I （LI、VI）

该系统调用涉及到系统时钟的修改，在容器中时钟没有隔离，并且在修改时涉及对timekeeper的锁竞争。

### 3、系统调用号：406 ：[clock\_getres](https://elixir.bootlin.com/linux/v5.1.16/source/kernel/time/posix-timers.c" \l "L1079)

功能：用来获取对应时钟类型能够提供的时间精确度。

参数：

const clockid\_t which\_clock 时间类型

struct \_\_kernel\_timespec \_\_user \* tp 用于保存精度信息

实现流程：

1）根据whick\_clock类型生成kclock类型结构；

2）调用kclock中成员函数clock\_getres获取精度信息。其中CLOCK \_REALTIME类型的时间精度通过宏 hrtimer\_resolution获取；

3）将精度信息存放在tp中。

隔离性：NI

该系统调用只读取时间精度信息，不存在隔离性问题。

### 4、系统调用号：407 ：[clock\_nanosleep](https://elixir.bootlin.com/linux/v5.1.16/source/kernel/time/posix-timers.c" \l "L1180)

功能：函数能够提供纳秒级的休眠时间。如果成功休眠了指定的时间，将返回0。

参数：

const clockid\_t which\_clock 检测的时钟类型，有效的类型有三种：CLOCK\_REALTIME、 CLOCK\_MONOTONIC 、CLOCK\_PROCESS\_CPUTIME\_ ID

int flags 指明用的是不是绝对时间。如果该值为0，则使用的是一个时间段(即休眠的时长)。如果该参数为设置为TIMER\_ABSTIME，那么就是使用绝对时间

const struct \_\_kernel\_timespec \_\_user \* rqtp 指明休眠时间。由参数flags指明这是一个时间段还是绝对时间。

struct \_\_kernel\_timespec \_\_user \* rmtp 当flags指明rqtp是一个时间段，并且本函数被信号打断时，该参数将得到余下的休眠时间。这个参数可以为NULL。

实现流程：

1）根据whick\_clock类型生成kclock类型结构kc；

2）如果kc为空，返回-EINVAL；

3）如果成员函数指针nsleep为空，返回-EOPNOTSUPP；

4）将休眠时间转换为timespec64类型；

5）检测转换后的休眠时间的有效性，如果无效返回-EINVAL；

6）根据flags以及rmtp设置宏current中的restart\_block结构值。

7）调用kclock中的nsleep函数完成睡眠时间的设置。

其中REALTIME、MONOTONIC两种时钟的nsleep函数通过hrtimer\_ nanosleep函数实现。该函数流程大致为：首先在堆栈中创建一个高精度定时器，设置它的到期时间，然后通过do\_nanosleep完成最终的延时工作，当前进程在挂起相应的延时时间后，退出do\_nanosleep函数，销毁堆栈中的定时器并返回0值表示执行成功。不过do\_nanosleep可能在没有达到所需延时数量时由于其它原因退出，如果出现这种情况，hrtimer\_nanosleep的最后部分把剩余的延时时间记入进程的restart\_block中，并返回ERESTART\_RESTARTBLOCK错误代码。CLOCK \_PROCESS\_CPUTIME\_ID类型的时钟，主要通过调用do\_cpu\_nanosleep函数实现。

隔离性：NI

该系统调用主要使用高精度计时器完成对进程的延时，涉及到的结构主要为进程描述符task\_struct和定时器。不同容器之间不会发生冲突。

### 5、系统调用号：408 ：[timer\_gettime](https://elixir.bootlin.com/linux/v5.1.16/source/kernel/time/posix-timers.c#L718)

功能：

获得一个活动定时器的剩余时间。

参数：

timer\_t timer\_id 定时器ID

struct \_\_kernel\_itimerspec \_\_user \* setting 存放获得的时间

实现流程：

1）调用do\_timer\_gettime函数获取剩余时间。

为该定时器加锁，同时获取该定时器的k\_itimer结构；

获取该定时器的kclock结构，如果该结构为空，或者timer\_get成员函数指针为空，返回-EINVAL，否则调用该函数获取剩余时间。

解锁。

2）将获取的时间放入setting所指的地址中。

隔离性：NI

只涉及读取存放定时器的哈希表的某个表目，不存在隔离性问题。

### 6、系统调用号：409 ：[timer\_settime](https://elixir.bootlin.com/linux/v5.1.16/source/kernel/time/posix-timers.c#L883)

功能：

设置一个定时器。

参数：

timer\_t, timer\_id 定时器ID

int, flags 标志时间类型，如果为TIMER\_ABSTIME，则new\_setting所指定的时间值会被解读成相对时间，否则为绝对时间。

const struct \_\_kernel\_itimerspec \_\_user \*, new\_setting 定时器到期时间

struct \_\_kernel\_itimerspec \_\_user \*, old\_setting 如果该值不是NULL，则之前的定时器到期时间会被存入其所提供的itimerspec。如果定时器之前处在未启动状态，则此结构的成员全都会被设定成0。

实现流程：

1）判断new\_setting是否为空，如果为空返回-EINVAL；

2）将new\_setting转换为itimerspec64类型；

3）调用do\_timer\_settime设置定时器；该函数大致流程为：

判断设置的时间是否有效，如果无效返回-EINVAL；

如果第四个参数不为空，设置为全0；

为定时器加锁，同时获取该定时器的k\_itimer结构；

获取定时器kclock结构，如果为空或者timer\_set函数指针为空，返回-EINVAL；否则调用该函数设置定时器到期时间。

解锁。

如果设置定时器时间返回TIMER\_RETRY，重新设置。

4）如果设置成功，并且old\_setting不为空，将之前定时器到期时间存放到old\_setting。

隔离性：NI

定时器在哈希表中的索引由当前进程的signal与定时器id一起求得。所以，即使docker容器没有隔离存放定时器的哈希表，在该系统调用也不会发生冲突。

### 7、系统调用号：410 ：[timerfd\_gettime](https://elixir.bootlin.com/linux/v5.1.16/source/fs/timerfd.c#L553)

功能：

获取定时器下次超时剩余的时间。

参数

int, ufd 函数timerfd\_create返回的文件句柄；

struct \_\_kernel\_itimerspec \_\_user \*, otmr 存放获得的时间，it\_value 字段表示距离下次超时的时间，如果该值为0，表示计时器已经解除，it\_interval字段表示定时器间隔时间；

实现流程：

1）调用do\_timerfd\_gettime函数获取剩余时间；该函数流程为：

根据整形文件句柄ufd获取该文件的fd结构；

获取fd结构file中的private\_data成员，该成员为定时器的timerfd\_ctx结构；

为该timerfd\_ctx结构加锁；

如果定时器失效并且是周期的，根据类别分别更新；

获取定时器剩余时间，填充it\_value、it\_interval字段；

解锁；

2）将获取的时间放入参数otmr所指向的位置；

隔离性：NI

该系统调用主要涉及文件的fd结构、以及定时器的timerfd\_ctx结构，不需要从系统中分配获得资源，容器环境下不存在隔离性问题。

### 8、系统调用号：411 ：[timerfd\_settime](https://elixir.bootlin.com/linux/v5.1.16/source/fs/timerfd.c#L535)

功能：

用于设置新的超时时间，并开始计时，能够启动和停止定时器。

参数：

int, ufd 函数timerfd\_create返回的文件句柄；

int, flags 为1代表设置的是绝对时间（TFD\_TIMER\_ABSTIME 表示绝对定时器）；为0代表相对时间；

const struct \_\_kernel\_itimerspec \_\_user \*, utmr 指定定时器的超时时间以及超时间隔时间；

struct \_\_kernel\_itimerspec \_\_user \*, otmr 如果不为NULL, 返回之前定时器设置的超时时间；

实现流程：

1）将utmr转换为itimerspec64类型；

2）调用do\_timerfd\_settime函数设置定时器时间；该函数流程为：

根据整形文件句柄ufd获取该文件的fd结构；

获取fd结构file中的private\_data成员，该成员为定时器的timerfd\_ctx结构；

停止定时器；

如果定时器过期并且是周期的，将它提前；

获得定时器之前的时间；

设置定时器时间；

3）如果otmr不为NULL，将之前定时器时间赋值给该指针指向的值。

隔离性：NI

该系统调用只涉及文件的fd结构以及定时器，不会从系统分配资源，容器下不存在隔离性问题。

### 9、系统调用号：412 ：[utimensat](https://elixir.bootlin.com/linux/v5.1.16/source/fs/utimes.c#L139)

功能：

以纳秒级的精度改变文件的时间戳；

参数：

int, dfd, 文件描述符，打开的文件描述符、-1或AT\_FDCWD

const char \_\_user \*, filename, 要修改的文件路径，如果文件描述符指向打开的文件，使用dfd，否则使用该路径打开文件，AT\_FDCWD表示使用当前工作目录；

struct \_\_kernel\_timespec \_\_user \*, utimes, 新修改的时间（包含两个元素的数组）或者NULL；

int, flags 0或AT\_SYMLINK\_NOFOLLOW

实现流程：

1）如果utimes非空，将数组中的两个元素转换为timespec64类型；

2）调用do\_utimes函数修改时间戳；该函数流程为：

如果时间参数非空，检测时间的有效性，如果无效返回-EINVAL;

检测flags值；

如果文件路径为空，并且文件描述符不为AT\_FDCWD，根据文件描述符获取文件fd结构，调用utimes\_common函数修改时间。否则使用文件路径打开文件并获取文件的path结构，然后调用utimes\_common函数修改时间。utimes\_common流程大致为：根据输入的参数设置iattr类型的结构成员，该结构包括需要修改的变量，需要设置ia\_valid成员，表示哪些变量需要修改，ia\_atime表示文件读取时间，ia\_mtime表示文件修改时间。然后调用notify\_change函数修改文件的inode。

隔离性：IE

该系统调用需要修改文件的inode，所以不同容器之间有隔离性问题，但是namespace已经实现文件系统的隔离，从而消除了该问题。

### 10、系统调用号：413：[pselect6\_time64](https://elixir.bootlin.com/linux/v5.1.16/source/fs/select.c" \l "L1360) //error

功能：

设置进程纳秒级的阻塞时间并等待指定信号，如果超时或者信号到达继续执行。

参数：

int, n, 文件描述符最大值+1

compat\_ulong\_t \_\_user \*, inp, 需要监视读变化的文件描述符

compat\_ulong\_t \_\_user \*, outp, 需要监视写变化的文件描述符

compat\_ulong\_t \_\_user \*, exp, 需要监视文件错误异常的文件描述符

struct \_\_kernel\_timespec \_\_user \*, tsp, 超时时间

void \_\_user \*, sig 阻塞信号

实现流程：

1）检测参数sig指针地址空间是否在该线程有效空间内；

2）调用do\_compat\_pselect函数，该函数流程为：

根据时间类型将输入时间转换为timespec64类型；将转换后的时间加上当前时间得到结束时间，如果输入的时间为0则结束时间也为0；

将进程阻塞信号current->blocked修改为输入的信号，并保存原来的信号；

设置超时时间，并等待信号；如果超时或者信号到达继续执行。

获得剩余时间；

将原信号保存在current->saved\_sigmask中。

隔离性：NI

该系统调用涉及的数据结构只与进程相关，容器之间不存在隔离性问题。

### 11、系统调用号：414 ：[ppoll\_time64](https://elixir.bootlin.com/linux/v5.1.16/source/fs/select.c#L1441) //error

功能：

参数：

struct pollfd \_\_user \*, ufds 存放被检测的文件描述符、被检测的事件

unsigned int, nfds, 文件描述符的个数

struct \_\_kernel\_timespec \_\_user \*, tsp, 超时时间

const compat\_sigset\_t \_\_user \*, sigmask, 信号

compat\_size\_t, sigsetsize

实现流程：

1）

隔离性分析：

### 12、系统调用号：415 ：无对应函数

### 13、系统调用号：416 ：[io\_pgetevents](https://elixir.bootlin.com/linux/v5.1.16/source/fs/aio.c" \l "L2087)

参数：

aio\_context\_t, ctx\_id, AIO句柄

long, min\_nr, 最少事件数

long, nr, 最大事件数

struct io\_event \_\_user \*, events, 存放事件结果

struct \_\_kernel\_timespec \_\_user \*, timeout, 超时时间

const struct \_\_aio\_sigset \_\_user \*, usig 阻塞的信号

功能：

首先设置进程的阻塞信号，然后尝试从由ctx\_id指定的aio上下文的完成队列中读取至少min\_nr个events的结果，至多nr个events的结果。如果ctx\_id无效，或min\_nr超出范围，或nr超出范围，或超时超出范围，则可能失败，返回-einval。如果指定的任何内存无效，则可能会以-efault失败。如果超时指定的超时时间在足够的事件可用之前已过，则可以返回0或<min\_nr，其中timeout==null指定无限超时。注意超时指向的超时是相对的。

实现流程：

1）如果timeout非空则转化为timespec64类型，转换失败返回-EFAULT；

2）如果输入信号usig非空，拷贝到内核空间，拷贝失败返回-EFAULT；

3）调用set\_user\_sigmask函数完成进程阻塞信号currnet->blocked的修改，并保存原信号；

4）调用do\_io\_getevents函数完成事件的读取，最终通过调用aio\_read\_events函数实现；

根据aio句柄获取对应的kioctx结构，涉及到对current->mm-> ioctx\_table的读取，采用的是rcu锁（rcu读取锁不会阻塞）；

判断超时时间，如果超时时间为0，调用一次aio\_read\_events函数，否则调用该函数直到超时或者该函数返回值为真；

使用结构kioctx成员ring\_lock加锁，并获取成员ring\_pages（用于存放异步处理的结果），从而获得异步处理结果的头尾指针；

当读取的events数小于nr时，一直循环读取，直到头指针移动到尾指针处（所有处理结果都已经读取）；

解锁；

5）保存原信号值；

隔离性：NI

本系统调用只涉及到aio相关数据结构以及进程结构task\_struct，不会从系统获取更多资源，不同进程的地址空间不一样，aio句柄也不会一样，不存在容器间的隔离性问题。

### 14、系统调用号：417 ：[recvmmsg](https://elixir.bootlin.com/linux/v5.1.16/source/net/socket.c" \l "L2630)

参数：

int, fd, 接收信息的socket；

struct mmsghdr \_\_user \*, mmsg, 指向mmsghdr结构数组的指针；

unsigned int, vlen, 上一参数数组的长度；

unsigned int, flags, 标记，可由以下一个或多个组成：MSG\_CMSG\_CLOEXEC、MSG\_DONTWAIT、MSG\_ERRQUEUE、MSG\_OOB、MSG\_PEEK、MSG\_TRUNC、MSG\_WAITALL；

struct \_\_kernel\_timespec \_\_user \*, timeout 指向超时时间，如果为NULL则超时时间无限长；

功能：

从套接字fd接收多条信息，直到时间超时或者接收到了vlen个消息；如果是非阻塞，则接收尽可能多的消息（最多vlen）然后立即返回；

实现流程：

1）如果timeout非空则转化为timespec64类型，转换失败返回-EFAULT；

2）如果timeout为NULL，以无限阻塞的方式调用do\_recvmmsg，实现读取vlen个消息；否则正常调用；

如果超时时间非NULL，将超时时间加上当前的单调时间得到结束时间，如果超时时间为0则结束时间也为0；

使用整形的fd获取对应的socket；

循环调用\_\_\_sys\_recvmsg函数接收一个消息，直到收到vlen个，或者时间超时；

返回收到的消息数；

3）如果返回值小于等于0，直接返回；

4）如果timeout不为NULL，将剩余时间存入该指针；

隔离性：IE

如果不同容器从相同端口号接收数据，一个收到数据，另一个就收不到，会有隔离性问题；但是namespace已经实现了网络资源的隔离。

### 15、系统调用号：418 ：[mq\_timedsend](https://elixir.bootlin.com/linux/v5.1.16/source/ipc/mqueue.c" \l "L1180)

参数：

mqd\_t, mqdes, 消息队列描述符

const char \_\_user \*, u\_msg\_ptr, 指向要插入的消息的指针

size\_t, msg\_len, 消息的长度

unsigned int, msg\_prio, 消息优先级

const struct \_\_kernel\_timespec \_\_user \*, u\_abs\_timeout 超时时间；

功能：

将msg\_ptr指向的消息添加到mqdes描述的消息队列中；msg\_len参数指定msg\_ptr指向的消息的长度，这个长度必须小于或等于队列的mq\_msgsize属性。允许零长度的消息。参数msg\_prio是一个非负整数，它指定此消息的优先级。消息被放置在队列中优先级递减，具有相同优先级的新消息放在具有相同优先级的旧消息之后。如果队列空间不足，则阻塞u\_abs\_timeout的时间，直到超时或者全部插入队列。

实现流程：

1）如果超时时间指针非空，将时间转换为timespec64类型，并检测合法性；

2）如果优先级超过最大优先级，返回-EINVAL；

3）如果时间非空，转换为ktime类型（纳秒表示形式）；

4）如果进程结构task\_struct成员audit\_contex非空，在该成员中记录传入的参数信息；

5）根据消息队列描述符获取对应的文件描述符；

6）获得文件inode，然后根据该inode获得消息队列的mequeue\_inode\_info结构首地址；

7）保存file的inode到进程的audit\_context结构的preallocated\_names成员中；

8）将u\_msg\_ptr指向的消息加载到msg\_msg结构中，并且保存长度和优先级；

9）如果消息队列对应的mequeue\_inode\_info结构中成员node\_cache为NULL（用于存放消息），需要分配该成员的内存；

10）如果队列中消息数等于最大消息数：如果为非阻塞模式直接返回，否则等待设定的超时时间；

否则：首先查找是否存在消息接收者，如果存在直接发送给接收者；不存在则将消息放入队列；

隔离性：IE

在无隔离时，每一个消息队列都对应一个全局mequeue\_inode\_info结构，该系统调用会修改该结构的信息，不同容器如果对同一个消息队列操作肯定会相互影响，但是namespace机制已经实现IPC的隔离，不同IPC namespace间互不可见。

### 16、系统调用号：419 ：[mq\_timedreceive](https://elixir.bootlin.com/linux/v5.1.16/source/ipc/mqueue.c" \l "L1194)

参数：

功能：

实现流程：

隔离性：

### 尚需解决

系统调用全过程

407号 休眠时间如何实现？是否影响？

408-411 定时器干啥的？ 是否影响？

413号 阻塞时间如何实现的？是否需要消耗系统资源，或影响其他容器？

貌似会涉及到cpu调度，以及如果不同容器等待相同信号？

414号 尚为完成

416号 如果两个容器对同一个aio句柄读取，会不会有影响？

分析涉及的数据结构是否为系统全局，如果是并且修改了信息，一般存在隔离性问题。