

暑 期 实 习 报 告

**课程名称： 暑期实习报告**

**专业班级： CS1601**

**学 号： U201614533**

**姓 名： 王洪磊**

**指导教师：**

**报告日期： 2019年8月20日**

计算机科学与技术学院

# 1、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：

首先通过tk\_core.timekeeper获得timekeeper，并获取当前时间；

修改timekeeper的wall\_to\_monotonic成员为原值减去设定的时间与当前时间的时间差；

修改timekeeper的xtime\_sec成员为设定时间的秒，tkr\_mono. xtime\_ nsec为相应的转换后的纳秒；

根据修改的值更新timekeeper；

通过访问每个cpu的hrtimer\_cpu\_base结构通知高精度定时器时钟的变化;

**隔离性：**

该系统调用涉及到时钟源tk\_core.timekeeper以及每个cpu的hrtimer\_cpu\_base结构。对timekeeper，容器与宿主的时钟是一致的，各个容器之间的时钟没有隔离，所以对该结构的修改存在运行时环境隔离与锁隔离问题。对hrtimer\_cpu\_base，每个cpu维护一个该结构，所以容器之间也是没有隔离的，访问时需要加锁，存在锁隔离。

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | 涉及资源 | 类型 | 隔离分类 |
| clock\_settime | tk\_core.timekeeper | I | LI、REI |
| clock\_settime | hrtimer\_cpu\_base | I | LI |

# 2、405 ：[clock\_adjtime](https://elixir.bootlin.com/linux/v5.1.16/source/kernel/time/posix-timers.c#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：

首先通过tk\_core.timekeeper获得timekeeper，并获取当前时间；

修改timekeeper的xtime\_sec成员，tkr\_mono. xtime\_ nsec加上调整的时间差；

修改timekeeper的wall\_to\_monotonic成员为原值减去调整的时间差；

通过更改的值更新timekeeper，并通知高精度定时器；

调整时钟每秒的增量，从而逐渐改变时间；

3）将ktx拷贝到用户空间utx。

**隔离性：**

该系统调用涉及到系统时钟的修改，隔离性问题与clock\_settime基本一致。

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | 涉及资源 | 类型 | 隔离分类 |
| clock\_adjtime | tk\_core.timekeeper | I | LI、REI |
| clock\_adjtime | hrtimer\_cpu\_base | I | LI |

# 3、406 ：[clock\_getres](https://elixir.bootlin.com/linux/v5.1.16/source/kernel/time/posix-timers.c#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，该宏最终就是TICK\_NSEC；

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、407 ：[clock\_nanosleep](https://elixir.bootlin.com/linux/v5.1.16/source/kernel/time/posix-timers.c#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，设置它的到期时间范围；

然后通过do\_nanosleep，设置当前进程状态为TASK\_INTERRUPTIBLE，调用hrtimer\_start\_expires将定时器插入到内核中维护的hrtimer\_cpu\_base，调用freezable\_schedule对进程调度，当前进程被挂起等待被唤醒；

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：I

该系统调用主要使用高精度计时器完成对进程的延时，高精度定时器需要插入到对应cpu的hrtimer\_cpu\_base结构中。此外，进行调度时需要将该进程从cpu的运行队列runqueue移除，当进程被唤醒时，再次将该进程加入runqueue。

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | 涉及资源 | 类型 | 隔离分类 |
| clock\_nanosleep | hrtimer\_cpu\_base | I | LI |
| clock\_nanosleep | runqueue | I | LI |

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

功能：

获得一个POSIX timer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。

使用当前进程的signal描述符与timerID在全局哈希表posix\_timers\_hashtable映射到对应链表头；

遍历链表找到该timer，获取该对应的k\_itimer结构，并为该timer加锁；

获得该timer的kclock结构，调用timer\_get成员函数，根据k\_itimer结构成员计算出剩余时间；

对该timer解锁；

2）将获取的时间放入setting所指的地址中。

隔离性：NI

该系统调用涉及全局哈希表posix\_timers\_hashtable，但是只涉及到读取，只为单个timer加锁，容器之间不会存在冲突；

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

功能：

设置一个POSIX timer的时间；

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 设置时间

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；

使用当前进程的signal描述符与timerID在全局哈希表posix\_timers\_hashtable映射到对应链表头；

遍历链表找到该timer，获取该对应的k\_itimer结构，并为该timer加锁；

获取定时器kclock结构，调用成员函数timer\_set，设置k\_itimer结构相关成员；

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timer到期时间存放到old\_setting。

隔离性：NI

该系统调用涉及全局哈希表posix\_timers\_hashtable，修改timer时间时只为单个timer的k\_itimer结构加锁，容器之间不会存在冲突。

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

功能：

获得文件描述符ufd引用的timer的当前时间设置。

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从当前进程的task\_struct的file成员获取该文件的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（ctx结构expired、tintv均非0），调用hrtimer\_forward\_now推后定时器的到期时间，并调用hrtimer\_restart，从内核中维护的hrtimer\_cpu\_base中删除旧定时器，插入新定时器；

根据timer的timerfd\_ctx结构，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：

该系统调用涉及的资源包括进程的task\_struct、内核中高精度定时器的hrtimer\_cpu\_base。对于task\_struct，用于获得打开的文件描述符的file结构，然后从private\_data中获取timerfd\_ctx结构，容器间不会冲突；对于hrtimer\_cpu\_base，用于在满足条件时维护定时器时间，修改时会对该结构加锁，容器间存在锁隔离。

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | 涉及资源 | 类型 | 隔离分类 |
| timerfd\_gettime | hrtimer\_cpu\_base | I | LI |

# 8、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；

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从当前进程的task\_struct的file成员获取该文件的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；

调用hrtimer\_try\_to\_cancel，从内核中维护的hrtimer\_cpu\_base中删除定时器；

如果定时器失效并且是周期的（ctx结构expired、tintv均非0），调用hrtimer\_forward\_now推后定时器的到期时间；

根据timer的timerfd\_ctx结构，填充it\_value、it\_interval字段；

设置定时器到期时间，初始化并启动一个高精度定时器加入到hrtimer\_cpu\_base；

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：

该系统调用涉及的资源包括进程的task\_struct、内核中高精度定时器的hrtimer\_cpu\_base。对于task\_struct，用于获得打开的文件描述符的file结构，然后从private\_data中获取timerfd\_ctx结构，容器间不会冲突；对于hrtimer\_cpu\_base，用于删除定时器，修改到期时间后重新加入，容器间存在锁隔离。

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | 涉及资源 | 类型 | 隔离分类 |
| timerfd\_gettime | hrtimer\_cpu\_base | I | LI |

# 9、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;

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，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：

该系统调用需要修改文件的inode，修改过程也需要加锁，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。

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | 涉及资源 | 类型 | 隔离分类 |
| utimensat | inode | IE(namespace) | REI、LI |

# 10、413：[pselect6](https://elixir.bootlin.com/linux/v5.1.16/source/fs/select.c#L774)

功能：

设置进程纳秒级的阻塞时间并等待文件描述符上的指定信号，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，并将指针指向的信号拷贝到size\_t结构和sigset\_t结构；

2）调用do\_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；

遍历当前进程的files，对每个文件，如果该文件满足条件（可读/可写/异常），记录加一，否则将该进程加入到该文件的等待队列；

在遍历完所有文件之后，并检查是否有满足条件的文件（retval值是否为0），或者是否超时，或者是否有未决信号，如果有那么直接跳出循环，否则调用poll\_schedule\_timeout，设置进程状态为TASK\_INTERRUPTIBLE，初始化一个高精度定时器，插入到hrtimer\_cpu\_base并启动，然后进行进程调度，该进程睡眠，时间到期再次处理一遍files；

等待被唤醒，然后获得剩余时间；

将原信号保存在current->saved\_sigmask中。

隔离性：

该系统调用涉及的数据结构只与进程相关，包括进程的task\_struct的files成员，以及poll\_wqueues等结构。此外，进程睡眠时间也涉及到定时器，会插入到内核维护的hrtimer\_cpu\_base；进程调度会将该进程从runqueue删除，被唤醒该进程会插入runqueue。

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | 涉及资源 | 类型 | 隔离分类 |
| pselect6 | hrtimer\_cpu\_base | I | LI |
| pselect6 | runqueue | I | LI |
| pselect6 | poll\_wqueues | IE(namespace) | LI、REI |

# 11、414 ：[ppoll](https://elixir.bootlin.com/linux/v5.1.16/source/fs/select.c#L1086)

功能：

等待一组文件描述符中的一个准备好执行I/O。

参数：

struct pollfd \_\_user \*, ufds, 文件描述符集合

unsigned int, nfds, 文件描述符个数

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

const sigset\_t \_\_user \*, sigmask, 阻塞信号

size\_t, sigsetsize 参数sigmask大小；

实现流程：

1）如果时间非NULL，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；

2）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；

3）将参数的ufds拷贝到poll\_list结构的entries成员中；

4）初始化一个poll\_wqueues结构；

5）遍历poll\_list中的每个pollfd，如果存在一个pollfd描述的事件准备好，则记录个数，然后返回个数；否则将当前进程加入对应文件的等待队列；

6）调用poll\_schedule\_timeout，设置进程状态为TASK\_INTERRUPTIBLE，初始化一个高精度定时器，插入到hrtimer\_cpu\_base并启动，然后进行进程调度，该进程睡眠；

7）释放poll\_wqueues、poll\_list结构的内存；

隔离性分析：

该系统调用涉及的数据结构只与进程有关，包括打开文件的等待队列、分配的poll\_wqueues结构等，不同容器的进程之前不会相互影响；此外，进程睡眠时间也涉及到定时器，会插入到内核维护的hrtimer\_cpu\_base；进程调度会将该进程从runqueue删除，被唤醒该进程会插入runqueue。

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | 涉及资源 | 类型 | 隔离分类 |
| ppoll | hrtimer\_cpu\_base | I | LI |
| ppoll | runqueue | I | LI |
| ppoll | poll\_wqueues | IE(namespace) | LI、REI |

# 12、415 ：无对应函数

# 13、416 ：[io\_pgetevents](https://elixir.bootlin.com/linux/v5.1.16/source/fs/aio.c#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wait\_event\_interruptible\_hrtimeout，先执行aio\_read\_events然后初始化一个高精度定时器并插入hrtimer\_cpu\_base，进行进程调度使该进程睡眠；aio\_read\_events函数执行为：

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：

本系统调用主要从当前进程的task\_struct结构的mm成员ring\_pages中读取io处理结果，容器间不存在隔离性问题；此外，系统调用的阻塞时间以及进程调度涉及内核维护的hrtimer\_cpu\_base以及runqueue结构，容器间存在锁隔离。

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | 涉及资源 | 类型 | 隔离分类 |
| io\_pgetevents | hrtimer\_cpu\_base | I | LI |
| io\_pgetevents | runqueue | I | LI |

# 14、417 ：[recvmmsg](https://elixir.bootlin.com/linux/v5.1.16/source/net/socket.c#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在进程task\_struct结构files中找到对应的file结构，然后获取其private\_data成员即得到socket结构；

循环调用sock->ops->recvmsg函数接收一个消息，然后获得当前时间，判断是否超时或者消息数到达vlen；

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：NI

该系统调用是在创建socket后调用的，不同进程创建的socket不会一样，所以容器之间不会有隔离性问题；此外，该系统调用涉及到进程的task\_struct结构的files成员获取对应的socket结构，同样不会出现隔离性问题。

# 15、418 ：[mq\_timedsend](https://elixir.bootlin.com/linux/v5.1.16/source/ipc/mqueue.c#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（file结构的f\_inode成员），然后根据该inode使用container\_of函数获得消息队列的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；最后修改inode的访问修改时间；

隔离性：

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，不同容器可以通过mq\_open打开同一个消息队列对之操作，这样肯定会相互影响，但是namespace机制已经实现IPC的隔离，不同IPC namespace间互不可见。

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | 涉及资源 | 类型 | 隔离分类 |
| mq\_timedsend | mequeue\_inode\_info | IE(namespace) | REI、LI |

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

参数：

mqd\_t, mqdes, 消息队列描述符

char \_\_user \*, u\_msg\_ptr, 存放消息的指针

size\_t, msg\_len, msg\_ptr大小

unsigned int \_\_user \*, u\_msg\_prio, NULL或者取得的消息优先级

const struct \_\_kernel\_timespec \_\_user \*, u\_abs\_timeout 阻塞时间

功能：

从mqdes所表示的消息队列中取出一个优先级最高且最老的消息，放入u\_msg\_ptr所指的内存中，u\_msg\_prio为取得的消息的优先级；如果消息队列为空，则等待到时间为u\_abs\_timeout（绝对时间）。

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）如果时间非空，转换为ktime类型（纳秒表示形式）；

3）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；

4）根据消息队列描述符获取对应的文件描述符；

5）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；

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

7）检查缓冲区空间大小是否足够（msg\_len）；

8）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；

9）如果消息队列中消息数为0：如果为非阻塞模式，返回-EAGIN；否则等待对应时间；

否则：从消息队对应的mequeue\_inode\_info结构中取出一条消息；

隔离性：

该系统调用隔离性与上一个系统调用一致，如果无隔离下，不同容器对一个消息队列操作，肯定会相互影响。但是namespace已经解决该隔离性问题。

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | 涉及资源 | 类型 | 隔离分类 |
| mq\_timedreceive | mequeue\_inode\_info | IE(namespace) | REI、LI |

# 17、420： [semtimedop](https://elixir.bootlin.com/linux/v5.1.16/source/ipc/sem.c#L2230)

参数：

int, semid, 信号量集标识符

struct sembuf \_\_user \*, tsops, 指向sembuf类型的数组，其中包括信号量位置，以及操作类型；

unsigned int, nsops, 数组sembuf元素个数；

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

功能：

对semid表示的信号集进行tsops所指的操作，共nsops个；sembuf结构为：

struct sembuf {

unsigned short sem\_num; /\* semaphore index in array \*/

short sem\_op; /\* semaphore operation \*/

short sem\_flg; /\* operation flags \*/

};

其中sem\_num指定信号量在集合中的位置（从0开始），sem\_op指定操作类型，sem\_flg为操作标记。sem\_flag可取IPC\_NOWAIT或SEM\_UNDO，如果为SEM\_UNDO，进程结束信号量恢复原值，IPC\_NOWAIT表示操作为非阻塞模式。如果sem\_op小于零，则进程必须具有alter权限，如果semval大于或等于该参数的绝对值，则操作可以立即进行，sem\_op的绝对值从semval中减去；如果sem\_op的绝对值大于semval，并且IPC\_NOWAIT在sem\_flg中指定，semop()失败，否则等待timeout指定的时间直到超时或者资源可用。如果sem\_op为零，这是一个“等待为零”的操作：如果semval为零，操作可以立即进行。否则，如果IPC\_NOWAIT在sem\_flg中指定，semop()失败，errno被设置为EAGAIN，否则semzcnt(等待该信号量的值变为0的线程数)加一，并且等待到超时或者信号量变为0。如果sem\_op是一个正整数，则加到信号量的值(semval)，该操作不会阻塞。如果timeout为NULL，则阻塞时间为无限长。

实现流程：

1）如果timeout非NULL，则将timespec类型转换为timespec64类型；

2）通过task\_struct结构的成员nsproxy获得进程的ipc名称空间；

3）如果信号集描述符小于0或者操作数组长度nsops小于1，返回-EINVAL;

4）将数组tsops拷贝到内核空间；

5）如果timeout非NULL，将转换后的类型转换为jiffies表示；

6）遍历拷贝后的nsops数组，查找是否需要alter权限、是否有SEM\_UNDO标记；

7）如果存在SEM\_UNDO标记，对该名称空间的信号集，获得undo结构，如果不存在则创建一个；

8）根据ipc名称空间与信号集描述符获得系统中对应的sem\_array结构；

9）检查进程的ipc权限；

10）为信号量加锁，如果操作值>0，进程不会阻塞；如果=0，信号值非0时会阻塞；如果<0，将信号值减到小于0的值时会阻塞。对每个操作进行处理，最后对信号量解锁。

隔离性：

该系统调用涉及的资源包括信号集的sem\_array结构，如果在无隔离的情况下，不同容器对同一信号操作肯定会相互影响，但是namespace已经隔离IPC，不同容器之间不可见；此外，该系统调用阻塞的时间使用了timer，该timer存放在全局的timer\_bases表中，插入时存在锁隔离。

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | 涉及资源 | 类型 | 隔离分类 |
| semtimedop | sem\_array | IE(namespace) | REI、LI |
| semtimedop | timer\_bases | I | LI |

# 18、421： [rt\_sigtimedwait](https://elixir.bootlin.com/linux/v5.1.16/source/kernel/signal.c#L3365)

参数：

const sigset\_t \_\_user \*, uthese, 等待的信号量集合

siginfo\_t \_\_user \*, uinfo, 如果非空，返回信号的siginfo

const struct \_\_kernel\_timespec \_\_user \*, uts, 等待的时间上限

size\_t, sigsetsize 参数uthese大小；

功能：

等待uthese指定的信号，sigsetsize指定大小，uts指定等待时间的上限，如果uinfo非NULL，用于返回信号的siginfo；

实现流程：

1）判断sigsetsize指定的大小是否正确，将uthese拷贝到内核空间，如果uts非空，将uts转换为timespec64类型；

2）如果时间参数非空，对转换后的timespec64判断有效性并转换为ktime类型（纳秒表示）；

3）从等待的信号集中删除SIGKILL、SIGSTOP信号；

4）检查是否有挂起的要等待的信号，如果有则立即返回该信号；

5）如果没有，并且转换后的时间非0：

首先设置进程的临时掩码real\_blocked为进程的阻塞信号掩码blocked；

然后使用要等待的信号设置进程的blocked成员；

设置进程状态为等待状态；

并使用高精度计时器使进程睡眠设定的时间；

然后使用real\_blocked恢复进程的阻塞信号blocked；

从信号队列取出等待的信号并返回。

隔离性：

该系统调用从进程的task\_struct结构中取出对应信号，该操作不涉及容器间的隔离性问题；此外，阻塞时间的时间涉及高精度定时器，进程的调度设计运行队列，这两个存在容器间的锁隔离问题。

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | 涉及资源 | 类型 | 隔离分类 |
| rt\_sigtimedwait | hrtimer\_cpu\_base | I | LI |
| rt\_sigtimedwait | runqueue | I | LI |

# 19、422： [futex](https://elixir.bootlin.com/linux/v5.1.16/source/kernel/futex.c#L3679)

参数：

u32 \_\_user \*, uaddr, 指向锁

int, op, 操作类型

u32, val, 与操作类型有关的值

struct \_\_kernel\_timespec \_\_user \*, utime, 操作超时时间

u32 \_\_user \*, uaddr2, 如果需要，该参数指向第二个锁

u32, val3 该参数含义取决于操作

功能：

futex()系统调用提供了一种方法用于等待某个条件变为真。它通常用作共享内存同步上下文中的阻塞结构。当使用futexe锁时，大多数同步操作都在用户空间中执行。用户空间程序只有在程序可能必须阻塞较长时间直到条件变为真时才使用futex()系统调用。其中uaddr指向要操作的锁，op为操作类型，val是与操作有关的值。后面三个参数只是在特定情况下使用，utime为阻塞的超时时间，uaddr2指向第二个锁，val3是与操作相关的值。

操作类型为FUTEX\_WAIT时，测试指定的锁值是否为val，如果是则休眠等待一个FUTEX\_WAKE操作，如果不是立即返回；如果utime非NULL，该值指向等待的时间，为NULL时时间无限长。其余两个参数忽略。

操作类型为FUTEX\_WAKE时，唤醒最多val个等待者，val=1时唤醒一个，等于INT\_MAX时唤醒所有等待者；最后三个参数忽略。

其他操作类型不再列出，详细见：<http://man7.org/linux/man-pages/man2/futex.2.html>

实现流程：

1）首先屏蔽操作op的FUTEX\_PRIVATE\_FLAG、FUTEX\_CLOCK\_ REALTIME位，从而得到操作类型；

2）如果时间utime非NULL，并且操作类型为：FUTRX\_WAIT\_x一类，将时间转换为ktime类型，并加上当前时间；如果操作类型为FUTEX\_x\_REQUEUE类，直接将utime转换为u32类型；

3）检查操作的各位是否冲突（主要为操作类型与标记）；

4）判断操作类型，分别调用不同功能函数实现，由于类型过多，以FUTEX\_WAIT为例：

首先设置val3为FUTEX\_BITSET\_MATCH\_ANY表示可由任何进程唤醒；

如果时间非NULL，定义一个高精度定时器，并初始化；

初始化该等待进程的futex\_q结构，获得对应hash\_bucket，将futex\_q结构插入到等待队列；

启动定时器，设置进程为TASK\_INTERRUPTIBLE，进行调度，当前进程挂起；直到超时、被唤醒或者一个信号；

隔离性：

每个futex锁都存放于全局结构\_\_futex\_data的futex\_hash\_bucket类型的成员queues数组中，该系统调用涉及对锁的等待队列的修改；此外，超时时间由高精度定时器实现，涉及到hrtimer\_cpu\_base，进程调度涉及到运行队列。

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | 涉及资源 | 类型 | 隔离分类 |
| futex | \_\_futex\_data.queues | I | LI、REI |
| futex | runqueue | I | LI |
| futex | hrtimer\_cpu\_base | I | LI |

# 20、423： [sched\_rr\_get\_interval](https://elixir.bootlin.com/linux/v5.1.16/source/kernel/sched/core.c#L5288)

参数：

pid\_t, pid, 进程号

struct \_\_kernel\_timespec \_\_user \*, interval 指向时间片写入地址

功能：

将pid标识的进程的时间片写入interval所指向的timespec结构中。指定的进程应该在SCHED\_RR（实时调度策略，时间片轮转）调度策略下运行。如果pid为0，则将调用进程的时间片写入interval。

实现流程：

1）使用pid找到对应进程task\_struct结构：如果pid为0，则即为当期进程current宏；否则首先根据当前进程的pid\_namespace结构的成员idr，以及进程号获得对应的pid结构，然后根据pid结构中的tasks成员获得对应的task\_struct；

2）对获得的进程所在的执行队列加锁，并调用task\_struct结构成员sched\_class的函数get\_rr\_interval获得时间片；最后解锁；

3）将获得的时间片转换为timespec64类型；

4）将时间信息拷贝到用户空间；

隔离性：

获取pid结构时需要读取pid\_hash表，不存在隔离性问题；

获得时间片时需要对相应进程所在运行队列加锁，所以，容器之间在runqueue上存在锁隔离。

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | 涉及资源 | 类型 | 隔离分类 |
| sched\_rr\_get\_interval | runqueue | I | LI |

# 21、424： [pidfd\_send\_signal](https://elixir.bootlin.com/linux/v5.1.16/source/kernel/signal.c#L3574)

参数：

int, pidfd, 文件描述符；

int, sig, 发送的信号；

siginfo\_t \_\_user \*, info, 信号信息；

unsigned int, flags 目前只能为0，将来可以用来扩展该系统调用；

功能：

通过文件描述符向一个进程发送信号；该系统调用目前仅通过PIDTYPE\_ PID发送信号，它不发送线程或进程组的信号。为了将该系统调用扩展到线程和进程组，应该使用@flags 参数（目前为实现）；

实现流程：

1）判断flags，如果不为0，返回-EINVAL；

2）通过当前进程的task\_struct结构（current宏）的成员files以及文件描述符获得对应的fd结构；

3）通过获得的文件的inode获得对应的proc\_inode结构，并取得其成员进程pid结构；

4）通过pid结构获得pid\_namespace，并验证信号发送者接受者是否在同一namespace，或者发送者是接收者的祖父namespace；

5）如果info参数非NULL，将该参数拷贝到内核空间，并验证信号是否符合要求（只允许进程向自身发送任意信号）；如果为NULL，根据sig初始化一个siginfo\_t结构；

6）将信号发送到目标pid对应进程的task\_struct结构中；

7）如果发送的信号是目标进程正在等待的信号，需要将该进程唤醒；

隔离性：

该系统调用主要涉及发送进程的task\_struct结构以及目标进程的task\_struct结构；如果无隔离的情况下，不同容器肯定会相互影响，但是namespace机制使不同容器间进程不可见，所以已经解决该隔离性问题。

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | 涉及资源 | 类型 | 隔离分类 |
| pidfd\_send\_signal | task\_struct | IE(namespace) | LI |
| pidfd\_send\_signal | task\_struct | IE(namespace) | REI |
| pidfd\_send\_signal | runqueue | I | LI |

# 22、425： [io\_uring\_setup](https://elixir.bootlin.com/linux/v5.1.16/source/fs/io_uring.c#L2995)

参数：

u32, entries, 提交队列、完成队列的深度；

struct io\_uring\_params \_\_user \*, params 设置的标记，并用于返回队列偏移量；

功能：

初始化提交队列SQ、完成队列CQ，此外，内核还初始化一个 Submission Queue Entries（SQEs）数组。SQ 和 CQ 中每个节点保存的都是 SQEs 数组的偏移量，而不是实际的请求，实际的请求只保存在 SQEs 数组中。这样在提交请求时，就可以批量提交一组 SQEs 上不连续的请求。返回值是一个 fd，进程使用这个 fd 进行 mmap，和 kernel 共享一块内存。

实现流程：

1）将params拷贝到内核空间，并判断flags正确性；

2）判断entries是否正确，如果等于0或者大于最大深度返回-EINVAL;

3）赋值拷贝后的io\_uring\_params结构成员sq\_entries为entries最接近的最大2的指数次幂；cq\_entries为sq\_entries的2倍；

4）分配io\_ring\_ctx结构、提交队列、完成队列内存；

5）如果参数的flags包含IORING\_SETUP\_SQPOLL标记，则新建一个线程用于处理SQ，线程插入到kthread\_create\_list中；

6）为io\_ring\_ctx结构分配一个文件描述符fd；

7）将分配的结果保存到params中；

隔离性：

该系统调用涉及的数据结构不同进程之间互不影响，所以不同容器之间也不会存在隔离性问题；但是创建线程插入到kthread\_create\_list会存在锁隔离；

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | 涉及资源 | 类型 | 隔离分类 |
| io\_uring\_setup | kthread\_create\_list | I | LI |

# 23、426： [io\_uring\_enter](https://elixir.bootlin.com/linux/v5.1.16/source/fs/io_uring.c#L2732)

参数：

unsigned int, fd, setup返回的文件描述符

u32, to\_submit, 提交的请求数；

u32, min\_complete, 等待完成的最小请求数；

u32, flags, 标记；

const sigset\_t \_\_user \*, sig, 信号

size\_t, sigsz 大小

功能：

io\_uring\_enter 被调用后会陷入到内核，内核将 SQ 中的请求提交给 Block 层。to\_submit 表示一次提交多少个 IO。

如果 flags 设置了 IORING\_ENTER\_GETEVENTS，并且 min\_complete > 0，那么这个系统调用会同时处理 IO 收割。这个系统调用会一直 block，直到 min\_complete 个 IO 已经完成。

如果在调用 io\_uring\_setup 时设置了 IORING\_SETUP\_SQPOLL 的 flag，提交过程完全不需要进行系统调用，新建的内核线程自动完成。

实现流程：

1）使用文件描述符获得file结构并得到private\_data指向的io\_ring\_ctx结构；

2）如果io\_ring\_ctx结构中的flags有IORING\_SETUP\_SQPOLL标记，并且参数flags有IORING\_ENTER\_SQ\_WAKEUP标记，则唤醒setup创建的线程；返回to\_submit；

3）如果to\_submit不为0，依次从io\_ring\_ctx结构的sq\_seqs中取出请求，并根据请求类型调用对应io函数（io\_read等等）；

4）如果flags有IORING\_ENTER\_GETEVENTS标记，需要返回IO处理结果：

如果setup时flags（保存在io\_ring\_ctx中）有IORING\_SETUP\_IOPOLL标记，则内核采用 Polling 的模式收割 Block 层的请求结果，实现就是从io\_ring\_ctx结构的poll\_list中检查处理结果；

否则，首先检查cq\_ring中完成的请求个数是否大于min\_events，如果是直接返回；否则，先使用参数sig设置进程的blocked信号，然后该进程挂起等待被唤醒，被唤醒后恢复blocked信号，返回；

隔离性：

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | 涉及资源 | 类型 | 隔离分类 |
| io\_uring\_enter | runqueue | I | LI |

# 24、427： [io\_uring\_register](https://elixir.bootlin.com/linux/v5.1.16/source/fs/io_uring.c#L3059)

参数：

unsigned int, fd, setup返回的io\_uring对应的文件描述符；

unsigned int, opcode, 标记；

void \_\_user \*, arg, 缓冲区地址或者文件地址；

unsigned int, nr\_args 参数arg元素个数；

功能：

io\_uring\_register()系统调用为fd引用的io\_uring实例注册用户缓冲区或文件。注册文件或用户缓冲区允许内核长期引用内部数据结构或创建应用程序内存的长期映射，从而大大减少I/O开销。

opcode可取如下值：

IORING\_REGISTER\_FILES/IORING\_UNREGISTER\_FILES 通过该系统调用提前注册一组file，缓解每次IO操作的fget()/fput()带来的开销。

IORING\_REGISTER\_BUFFERS/IORING\_UNREGISTER\_BUFFERS 通过该系统调用注册一组固定的IO buffers，当应用重用这些IO buffers时，只需要map/unmap一次即可，而不是每次IO都要去做。

实现流程：

1）根据文件描述符fd获得文件的file结构，并从private\_data中获得对应的io\_ring\_ctx结构；

2）根据opcode类型不同，分别实现不同功能，当为IORING\_REGISTER\_BUFFERS时：

判断是否已经注册过虚拟地址（io\_ring\_ctx结构成员user\_bufs是否为非NULL），如果已经注册过返回-EBUSY；

判断参数地址个数nr\_args是否在正常范围，不在则返回-EINVAL；在则为user\_bufs成员分配内存；

nr\_args个地址，对每个分别处理：首先将iovec结构的地址信息拷贝到内核空间，判断地址信息的有效性，然后为user\_bufs[i]分配对应的空间，并将转换后的地址信息存储到bvec成员中，最后在user\_bufs[i]中保存对应的虚拟地址信息；

当opcode为其他三种类型时实现流程类似。

隔离性：NI

该系统调用只是将虚拟地址或者file在io\_ring\_ctx结构中注册，只涉及修改对应的io\_uring实例的io\_ring\_ctx结构。