1.alsa 数据收发流程

playback 在用户态的流程：

aplay.c->pcm_write->writei_func->snd_pcm_writei->writei->snd_pcm_hw_writei->ioctl(fd, SNDRV_PCM_IOCTL_WRITEI_FRAMES, &xferi)

playback 在内核态的流程是：

snd_pcm_ioctl->snd_pcm_common_ioct->case SNDRV_PCM_IOCTL_WRITEI_FRAMES->snd_pcm_xferi_frames_ioctl->snd_pcm_lib_write->_snd_pcm_lib_xfer->writer->interleaved_copy->default_write_copy->memcpy(get_dma_ptr(substream->runtime, channel, hwoff), buf, bytes)

playback 内核态出发 dmaengine_pcm_prepare_and_submit 的调用栈是：

snd_dmaengine_pcm_trigger

dmaengine_pcm_trigger

snd_soc_pcm_component_trigger

soc_pcm_trigger

snd_pcm_do_start

snd_pcm_action_single

snd_pcm_action

snd_pcm_start

__snd_pcm_lib_xfer

snd_pcm_ioctl

sys_ioctl

ret_from_syscall

playback 时，硬件 dma 触发的流程是：

axi_chan_block_xfer_start 触发候，硬件就一直自动按照 linked-list LLI 链表来搬运。搬运本身是自动触发下一条的，搬运方向是 MEM_TO_DEV，速度会收到 i2s FIFO 的反压，也就是 FIFO 给 i2s 收发硬件往外吐的速度是比较低的（相对于 MEM 访问速度），硬件上 FIFO 空的时候，才会触发一次 dma 握手信号，就形成了类似流控的手段，让 dma“慢慢”搬。所以搬的速度就是播放的速度。

playback 时，硬件 dma 完成中断的流程是：

snd_pcm_update_hw_ptr0

snd_pcm_period_elapsed

dmaengine_pcm_dma_complete

vchan_complete

tasklet_action

__do_softirq

irq_exit

__handle_domain_irq

riscv_intc_irq

ret_from_exception

在 snd_pcm_update_hw_ptr0 中就是更新 hw_ptr 往前走了。

capture 在用户态的流程：

main->capture->pcm_read->readi_func->snd_pcm_readi->_snd_pcm_readi->snd_pcm_hw_readi->ioctl(fd, SNDRV_PCM_IOCTL_READI_FRAMES, &xferi)阻塞调用

capture 在内核态的流程是：

SNDRV_PCM_IOCTL_READI_FRAMES->snd_pcm_xferi_frames_ioctl->snd_pcm_lib_read->__snd_pcm_lib_xfer->snd_pcm_start->wait_fo

r_avail(等数据上来)->writer(后续将拷贝到用户空间)->interleaved_copy->default_read_copy->copy_to_user((void __user *)buf, get_dma_ptr(substream->runtime, channel, hwoff), bytes)

那么这里最耗时的是 wait_for_avail：

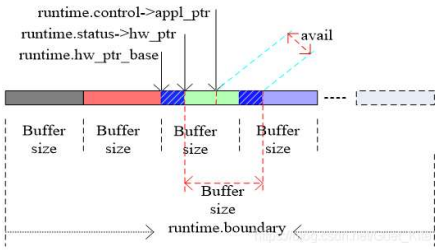
```
static int wait_for_avail(struct snd_pcm_substream *substream,
                                                snd_pcm_uframes_t *availp)
{
                struct snd_pcm_runtime *runtime = substream->runtime;
                int is_playback = substream->stream == SNDRV_PCM_STREAM_PLAYBACK;
                wait_queue_entry_t wait;
                int err = 0;
                snd_pcm_uframes_t avail = 0;
                long wait_time, tout;

                init_waitqueue_entry(&wait, current);
                set_current_state(TASK_INTERRUPTIBLE);
                add_wait_queue(&runtime->tsleep, &wait);

                if (runtime->no_period_wakeup)
                                wait_time = MAX_SCHEDULE_TIMEOUT;
                else {
                                /* use wait time from substream if available */
                                if (substream->wait_time) {
                                                wait_time = substream->wait_time;
                                } else {
                                                wait_time = 10;

                                                if (runtime->rate) {
                                                                long t = runtime->period_size * 2 /
                                                                                                runtime->rate;
                                                                wait_time = max(t, wait_time);
                                                }
                                                wait_time = msecs_to_jiffies(wait_time * 1000);
                                }
                }

                for (;;) {
                                if (signal_pending(current)) {
                                                err = -ERESTARTSYS;
                                                break;
                                }

                                /*
                                 * We need to check if space became available already
                                 * (and thus the wakeup happened already) first to close
                                 * the race of space already having become available.
                                 * This check must happen after been added to the waitqueue
                                 * and having current state be INTERRUPTIBLE.
                                 */
                                avail = snd_pcm_avail(substream);
                                if (avail >= runtime->twake)
                                                break;
                                snd_pcm_stream_unlock_irq(substream);

                                tout = schedule_timeout(wait_time);

                                snd_pcm_stream_lock_irq(substream);
                                set_current_state(TASK_INTERRUPTIBLE);
                                switch (runtime->status->state) {
                                case SNDRV_PCM_STATE_SUSPENDED:
                                                err = -ESTRPIPE;
                                                goto _endloop;
                                case SNDRV_PCM_STATE_XRUN:
                                                err = -EPIPE;
                                                goto _endloop;
                                case SNDRV_PCM_STATE_DRAINING:
                                                if (is_playback)
                                                                err = -EPIPE;
                                                else
                                                                avail = 0; /* indicate draining */
                                                goto _endloop;
                                case SNDRV_PCM_STATE_OPEN:
                                case SNDRV_PCM_STATE_SETUP:
                                case SNDRV_PCM_STATE_DISCONNECTED:
                                                err = -EBADFD;
                                                goto _endloop;
                                case SNDRV_PCM_STATE_PAUSED:
                                                continue;
                                }
                                if (!tout) {
                                                pcm_dbg(substream->pcm,
                                                                "%s write error (DMA or IRQ trouble?)\n",
                                                                is_playback ? "playback" : "capture");
                                                err = -EIO;
                                                break;
                                }
                }
 _endloop:
                set_current_state(TASK_RUNNING);
                remove_wait_queue(&runtime->tsleep, &wait);
                *availp = avail;
                return err;
}
```

黄色底色的行，可以看到是类似 wait_event_interruptible 的实现，`runtime->tsleep`

2.alsa buffer 管理

参考资料：

《ALSA 子系统（十二）------ALSA Buffer 的更新.pdf》



总体说就是，alsa 的整个 buffer 是个环形缓冲区。

appl_ptr 就是 application pointer，也就是用户态程序使用的指针，在播放的时候是写指针，在录音的时候是读指针。

hw_ptr 就是 hardware pointer，也就是内核态 alsa 子系统使用的指针，在播放的时候是读指针，在录音的时候是写指针。

播放的时候是 hw_ptr 追赶 appl_ptr

录音的时候是 appl_ptr 追赶 hw_ptr。

后面都以播放为例。

因为是环形缓冲区，hw_ptr/appl_ptr 到了缓冲区尾部就要折回。alsa 设计上使用了较大的缓冲区，避免经常发生折返的情况。

上图的 buffer_size=period_size*period_count。

period_size 是指每两个中断之间，能传输的帧数。注意是帧数。假设 period_size 是 1024 帧，每帧是 4 字节，双声道，一个声道是 2 字节也就是 16bit，则实际 dma 传输量是 period_size * 4 字节=4096 字节。

period_count 是 period 的个数，假设是 22（系统实测值），则 buffer_size 为 1024*22=22528 帧

整个缓冲区由多个 buffer_size 组成，boundary 表示整体缓冲区大小。

那么 buffer 使用的规则是：

hw_ptr_base 表示的是 hw_ptr 使用的是哪个 buffer_size 的 buffer。

appl_ptr 走在前面，hw_ptr 走在后面追赶。

当 appl_ptr 超过当前 buffer_size 的时候，会切换到下一个 buffer_size 继续写。

实际运行的时候，appl_ptr 会先走一个 buffer_size 大小，然后 hw_ptr 开始追赶，这期间 appl_ptr 和 hw_ptr 的前进速度一直，并一直相差一个 buffer_size。

图中的 avail，是 hw_ptr+buffer_size 后，减去 appl_ptr。

那么对于播放而言，avail=0 是正常的事情，假设 appl_ptr 走的整整比 hw_ptr 快了 1 一个 buffer_size，这是实际正常的运行情况。

只有当 avail 越来越大，到达一个 buffer_size 大小(stop_threshold 等于 buffer_size)的时候，表示 appl_ptr 走的太慢，到达了 underrun 的判定条件。

具体代码如图：

```
int snd_pcm_update_state(struct snd_pcm_substream *substream,
             struct snd_pcm_runtime *runtime)
{
    snd_pcm_uframes_t avail;

    avail = snd_pcm_avail(substream);
    if (avail > runtime->avail_max)
        runtime->avail_max = avail;
    if (runtime->status->state == SNDRV_PCM_STATE_DRAINING) {
        if (avail >= runtime->buffer_size) {
            snd_pcm_drain_done(substream);
            return -EPIPE;
        }
    } else {
        if (avail >= runtime->stop_threshold) {
            __snd_pcm_xrun(substream);
            return -EPIPE;
```