Linux 驱动开发 / fbdev 双缓存 / 快速入门

嵌入式小傻瓜
于 2021-02-22 08:33:31 发布

文章标签： 编程语言 java linux 区块链 python

哈喽，我是老吴。

我回来啦！

过年回老家，特别充实，大部分时间都在带娃～

女儿快 2 岁了，走起路来像模像样，在屋子里转来转去，有的时候还会小跑，痴迷于外出逛街。

目前我稍微有点理解她了。

2岁的小孩特别倔强，她想做或者不愿意做的事，一般都会跟我死磕到底，很难强迫她。

我发现 2 个比较有效的方法：

1> 转移注意力

例如大半夜她吵着要出去玩，就得先用好吃的水果稳住她.;

2> 顺势而为

例如她一定要玩剪刀，家里就提前准备好一把玩具剪刀，给她玩;

哈哈哈，还有很多好玩的技巧，以后再分享了，下面开始学技术。

一、为何需要 double buffer?

single buffer 会导致：

屏幕撕裂(tearing)，即在屏幕上同时看到多帧数据拼接在一起。


Tear Point #1 --->
Tear Point #2 --->

点击查看大图

single buffer 为何会造成撕裂：

refresh rate 和 frame rate 不一致。

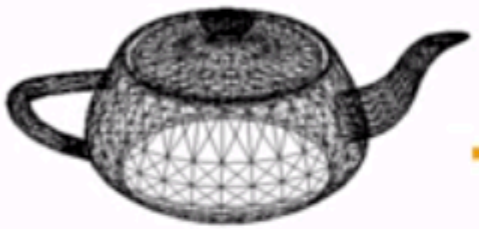
refresh rate 表示的是 屏幕每秒能更新多少次显示，例如 30hz / 60hz。

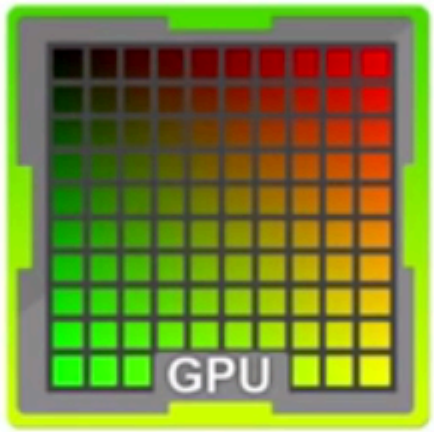
Refresh Rate

60Hz

点击查看大图
frame rate 表示的是 lcd controller / gpu 每秒能绘制多少帧数据，例如 30fps / 60fps。

Frame Rate

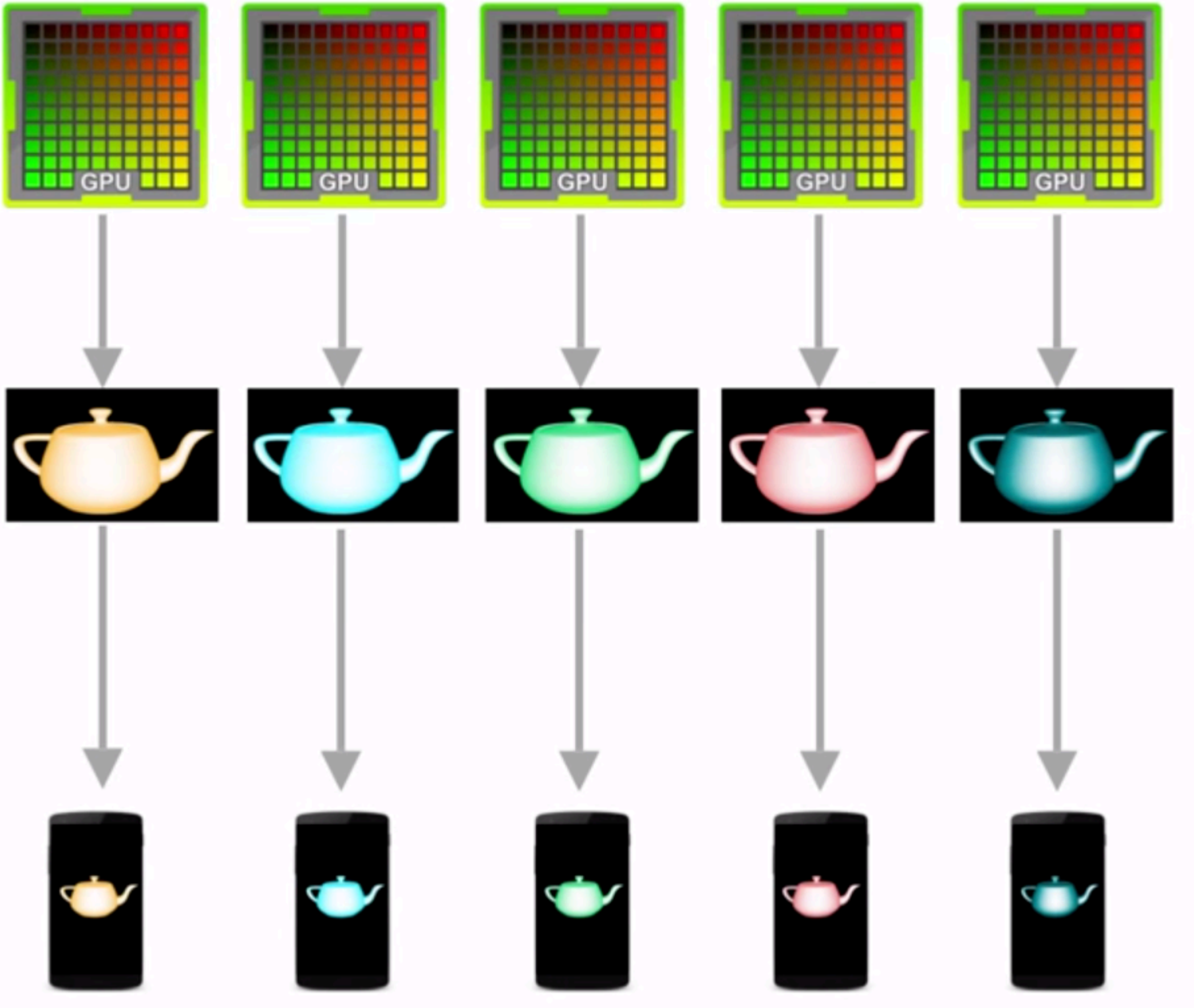




30fps

60fps

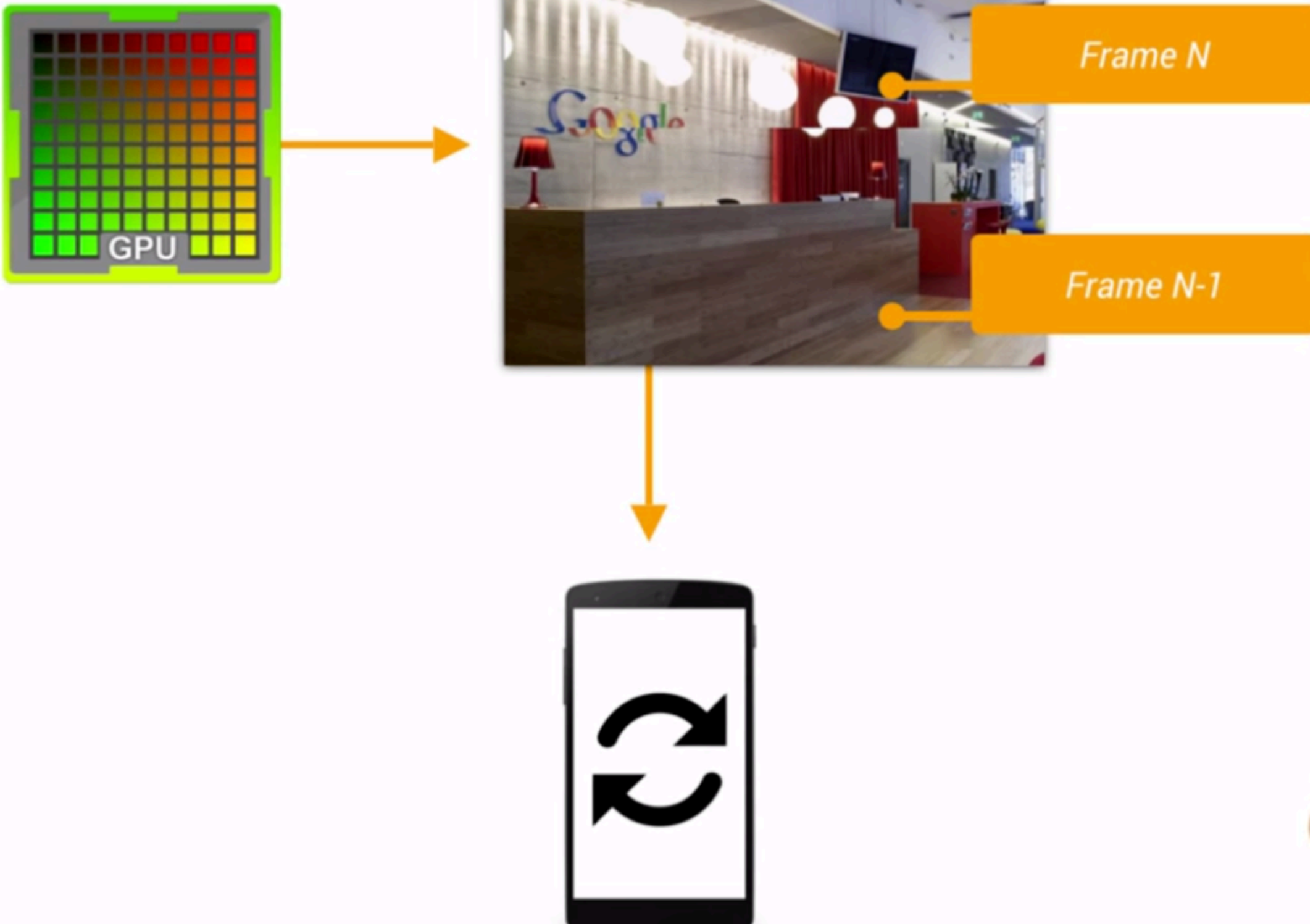
点击查看大图
LCD controller / gpu 和 屏幕协作完成一帧图像的显示:


GPU
GPU
GPU
GPU
GPU

点击查看大图
在 single buffer 的场景下，LCD user 和 LCD controller / gpu 总是在共用同一个 framebuffer，且没有同步机制。

LCD user 是写者，LCD controller / gpu 是读者。

由于存在竞争关系且读写没有同步机制，framebuffer 里必须会发生同时存在frame N 和 frame N-1 的数据，此时 LCD 将 framebuffer 的数据显示出来时，就会看到撕裂的效果：


GPU
Google
Frame N
Frame N-1

点击查看大图

可以通过 double buffer+vsync 解决撕裂的问题。

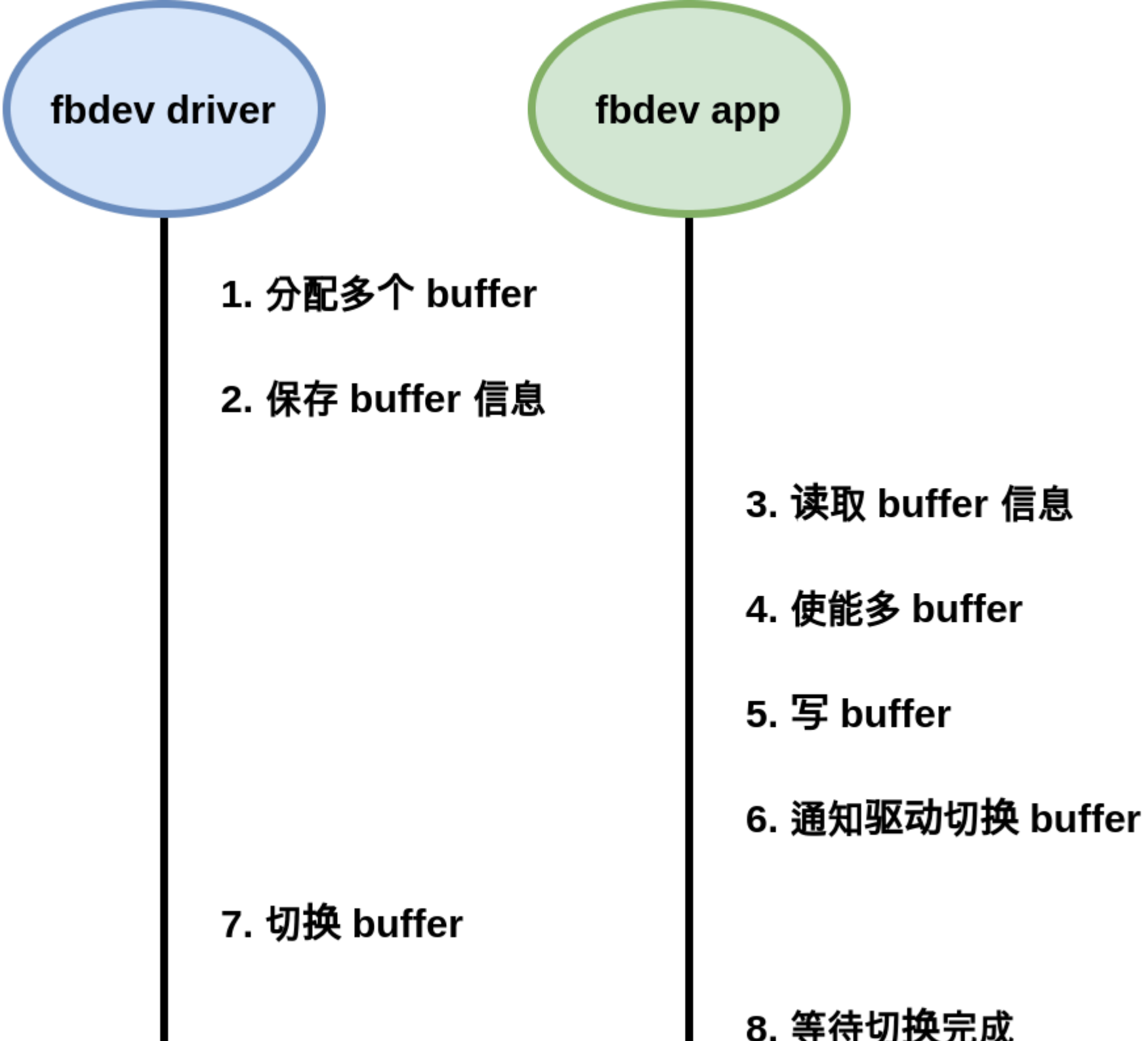
double buffer，顾名思义，就是有 2 个 framebuffer，其工作逻辑如下：

- LCD controller : draw fb0 to screen
- LCD user : write data to fb1
- LCD controller : draw fb1 to screen
- LCD user : write data to fb0
- 循环...

vsync 机制则用于确保一帧图像能不被打断地显示在屏幕。

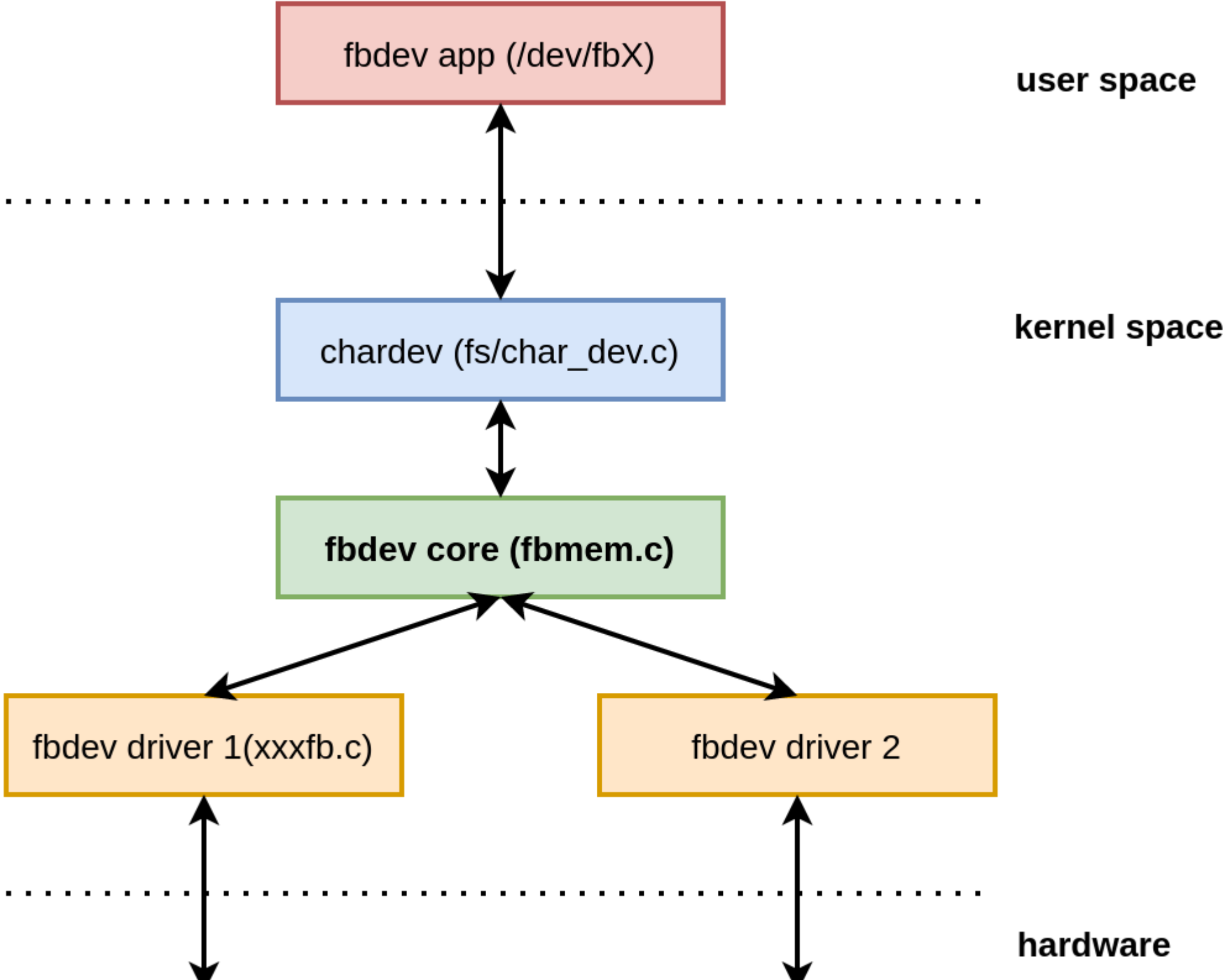
如何支持 double buffer?

需要驱动和应用互相配合：


fbdev driver
fbdev app
1. 分配多个 buffer
2. 保存 buffer 信息
3. 读取 buffer 信息
4. 使能多 buffer
5. 写 buffer
6. 通知驱动切换 buffer
7. 切换 buffer
8. 等待切换完成

二、编写支持 double buffer 的 fbdev 驱动

fbdev 框图：


fbdev app (/dev/fbX)
user space
chardev (fs/char_dev.c)
kernel space
fbdev core (fbmem.c)
fbdev driver 1(xxxfb.c)
fbdev driver 2
hardware

先梳理一下思路：

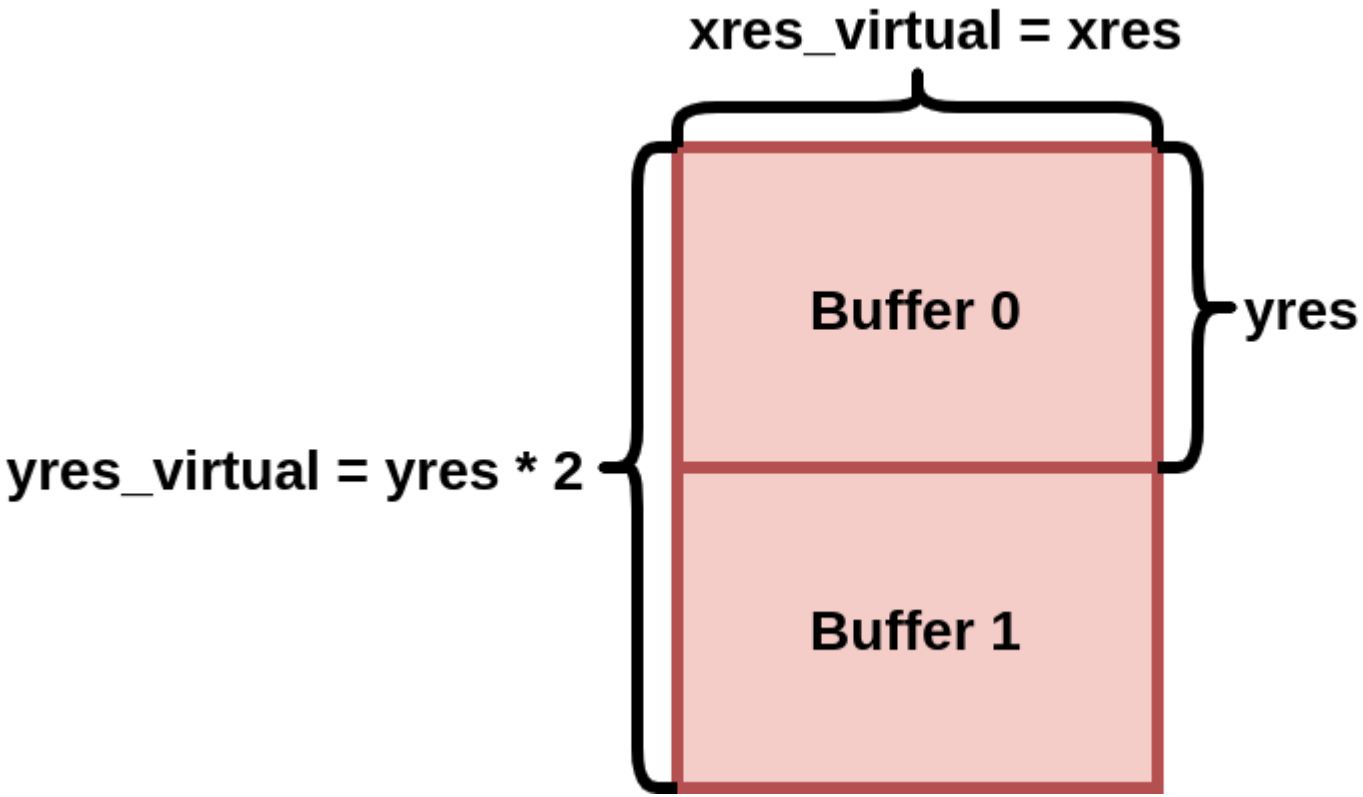
让驱动支持 double buffer 需要做 3 件事。

1. 申请2 x buffer：

```
size = (2 * width * height);
fbi->screen_base = dma_alloc_wc(sfb->dev, size, &map_dma, GFP_KERNEL);
```

2. 将 buffer 相关的信息保存 struct fb_info-> struct fb_var_screeninfo。

```
struct fb_var_screeninfo {
    __u32 xres;           /* visible resolution        */
    __u32 yres;
    __u32 xres_virtual;        /* virtual resolution        */
    __u32 yres_virtual;
    __u32 xoffset;           /* offset from virtual to visible */
    __u32 yoffset;           /* resolution            */
    ...
}
```



点击查看大图
xres 和 yres 是真实的 LCD 分辨率的宽和长;

xres_virtual 和 yres_virtual 是显存区域的宽和长;

xoffset 和 yoffset 用于指定当前使用哪一个 Buffer 进行绘制。使用 Buffer0 时 ， xoffset = 0， yoffset=0; 使用 Buffer1 时， xoffset = 0, yoffset = yres * 1;

3. 支持切换 buffer，具体的就是实现 ioctl：FBIOPAN_DISPLAY。

pan 的本意是平移，可以想象成显存上方有一个取景框，平移取景框可以看到不同的显示内容。

实例分析：goldfishfb.c

goldfishfb.c 是虚拟硬件 goldfish 的 fbdev 驱动，我们可以参考这个文件，学习如何实现 double buffer。

1. 分配 2 x buffer：

```
int goldfish_fb_probe()
{
    ...
    framesize = width * height * 2 * 2;

    fb->fb.screen_base = (char __force __iomem *)dma_alloc_coherent(&pdev->dev, framesize, &fbpaddr, GFP_KERNEL);
}
```

2. 设置 fb_var_screeninfo：

```
int goldfish_fb_probe()
{
    ...
    fb->fb.var.xres  = width;
    fb->fb.var.yres  = height;
    fb->fb.var.xres_virtual = width;
    fb->fb.var.yres_virtual = height * 2;
}
```

3. 实现 ioctl / FBIOPAN_DISPLAY：

```
static struct fb_ops goldfish_fb_ops = {
 ...
 .fb_pan_display = goldfish_fb_pan_display,
};
```

```
int goldfish_fb_pan_display()
{
    ...

    // 将新的显存地址告知 lcd controller
```

```
    writel(fb->fb.fix.smem_start + fb->fb.var.xres * 2 * var->yoffset,

     fb->reg_base + FB_SET_BASE);
    // 等待 LCD controller 的 vsync 信号
    wait_event_timeout(fb->wait,fb->base_update_count != base_update_count, HZ / 15);
}
```

当LCD controller 将一帧图像完整地显示在 LCD 上后，就会产生一个中断，在中断里就会执行唤醒睡眠在 fb_pan_display 里的进程。

如果你想多了解一些，可以阅读 DRM 框架里的 fbdev 兼容代码，此代码也是支持 double buffer的：

- linux/drivers/gpu/drm/*/*_drm_fbdev.c
- linux/drivers/gpu/drm/drm_fb_helper.c

三、编写支持 double buffer 的 fbdev 应用

驱动支持 double buffer 后，还得在应用程序里将其使用起来。

先梳理一下思路：

1. 检查是否支持 double buffer;
2. 使能 double buffer：FBIOPUT_VSCREENINFO;
3. 更新 buffer 里数据;
4. 通知驱动切换 buffer：FBIOPAN_DISPLAY;
5. 等待切换完成：FBIO_WAITFORVSYNC;

实例分析：show_color.c

```
static int fd_fb;
```

```
static struct fb_fix_screeninfo fix;    /* Current fix */
static struct fb_var_screeninfo var;    /* Current var */
static int screen_size;
static unsigned char *fb_base;
static unsigned int line_width;
static unsigned int pixel_width;

int main(int argc, char **argv)
{
    int i;
    int ret;
    int buffer_num;
    int buf_idx = 1;
    char *buf_next;
    unsigned int colors[] = {0x00FF0000, 0x0000FF00, 0x000000FF, 0, 0x00FFFFFF};  /* 0x00RRGGBB */
    struct timespec time;

    ...

    fd_fb = open("/dev/fb0", O_RDWR);
    ioctl(fd_fb, FBIOGET_FSCREENINFO, &fix);
    ioctl(fd_fb, FBIOGET_VSCREENINFO, &var);

    line_width  = var.xres * var.bits_per_pixel / 8;
    pixel_width = var.bits_per_pixel / 8;
    screen_size = var.xres * var.yres * var.bits_per_pixel / 8;

    // 1. 获得 buffer 个数
    buffer_num = fix.smem_len / screen_size;
    printf("buffer_num = %d\n", buffer_num);

    fb_base = (unsigned char *)mmap(NULL , fix.smem_len, PROT_READ | PROT_WRITE, MAP_SHARED, fd_fb, 0);
    if (fb_base == (unsigned char *)-1) {
        printf("can't mmap\n");
        return -1;
    }
```

```
    if ((argv[1][0] == 's') || (buffer_num == 1)) {
    printf("single buffer:\n");
    while (1) {
        for (i = 0; i < sizeof(colors)/sizeof(colors[0]); i++) {
            lcd_draw_screen(fb_base, colors[i]);
            nanosleep(&time, NULL);
        }
    }
} else {
    printf("double buffer:\n");

    // 2. 使能多 buffer
    var.yres_virtual = buffer_num * var.yres;
    ioctl(fd_fb, FBIOPUT_VSCREENINFO, &var);

    while (1) {
        for (i = 0; i < sizeof(colors)/sizeof(colors[0]); i++) {

            // 3. 更新 buffer 里的数据
            buf_next =  fb_base + buf_idx * screen_size;
            lcd_draw_screen(buf_next, colors[i]);

            // 4. 通知驱动切换 buffer
            var.yoffset = buf_idx * var.yres;
            ret = ioctl(fd_fb, FBIOPAN_DISPLAY, &var);
            if (ret < 0) {
                perror("ioctl() / FBIOPAN_DISPLAY");
            }

            // 5. 等待帧同步完成
            ret = 0;
            ioctl(fd_fb, FBIO_WAITFORVSYNC, &ret);
            if (ret < 0) {
                perror("ioctl() / FBIO_WAITFORVSYNC");
            }
```

```
                buf_idx = !buf_idx;
                nanosleep(&time, NULL);
            }
        }

    }

    munmap(fb_base , screen_size);
    close(fd_fb);

    return 0;
}
```

运行：

```
$ ./show_color single
buffer_num = 1
single buffer:

$ ./show_color double
buffer_num = 2
double buffer:
```

该程序会在屏幕上循环的显示不同的颜色。

当传入 "single" 参数时，使用单 buffer，可见撕裂。

当传入 "double" 参数时，使用双 buffer，不再撕裂。

代码不是很复杂，我就不再详细分析了。

如果你想多了解一些，可以阅读开源软件 SDL-1.2 里的 sdl_fbvideo.c，此代码也支持了 double buffer。

另外，现在越来越多的显示设备走的是 DRM 框架，该框架自然是支持多 buffer的。感兴趣的小伙伴，自行查看下面的代码：

https://github.com/dvdhrm/docs/blob/master/drm-howto/modeset-double-buffered.c

四、相关参考

百问网 / 韦东山驱动大全教学视频：https://www.100ask.net/detail/p_5ff2c46ce4b0c4f2bc4fa16d/8

维基百科：https://en.wikipedia.org/wiki/Screen_tearing

思考技术，也思考人生

要学习技术，更要学习如何生活。