ChCore GUI Lab

上海交通大学 并行与分布式系统研究所

2024 年 8 月

目录

1 简介

ChCore 的 GUI 框架以 Wayland 协议为基础，后端采用自主编码的 Compositor，前端采用 LVGL 嵌入式 GUI 库。完成本实验，你将学习到：

- 作为现代 GUI 前后端通信协议的 Wayland 的原理；
- 标准 Wayland 协议后端合成器的实现方法；
- 借助 LVGL 前端库（或者不借助），实现一个自己的 ChCore GUI App，并且完全清楚在这个 GUI App 中图像数据以及交互事件是如何借助 GUI 系统在硬件和用户态应用之间传递。

2 环境准备

2.1 实机运行

和之前一样，本实验依旧借助树莓派平台，树莓派 3b 和树莓派 4b 均可用。另外，需要借助 USB 鼠标、键盘以及一台能够支持树莓派 RGBA 数据的显示器（ChCore 中的 HDMI 驱动会对这一点进行检测）。在你完成某一步的代码编写时，按照下列步骤在实机上运行。如果你进行过之前的实验并观看了视频教程，或者是使用过树莓派，你应该已经有了一些基本的操作经验。

1. 首先，使用下列命令进行编译：

   ```
   $ make rpi3-board
   ```

   或是

   ```
   $ make rpi4-board
   ```

   此时，如果编译没有错误，你将会在/build 文件夹下找到一个名为 kernel8.img 的文件。

2. 首先需要准备一张合适的 MicroSD 卡，并适当分区（建议直接将树莓派官方提供的 Raspberry Pi OS 使用 dd 命令烧写到 MicroSD 卡，比较便捷），保证第一个分区是 FAT32 文件系统。然后将 kernel/arch/aarch64/boot/raspi3（或是 raspi4）/firmware 中的所有文件拷贝到 MicroSD 卡的 FAT32 分区，再将构建出的 ChCore 内核 build/kernel8.img 也拷贝到 FAT32 分区。

3. 将 MicroSD 卡插进树莓派，并使用 TTL 转 USB 线将树莓派与电脑连接。连接方式为：白色 RX (D-) 线连接到树莓派的 GPIO 上 GPIO 14 TXD 引脚，绿色 TX (D+) 线连接到树莓派的 GPIO 15 RXD 引脚，黑色地线连接到树莓派的任意 GND 引脚（如 6 号引脚或 14 号引脚）。具体连接方式如图1所示。

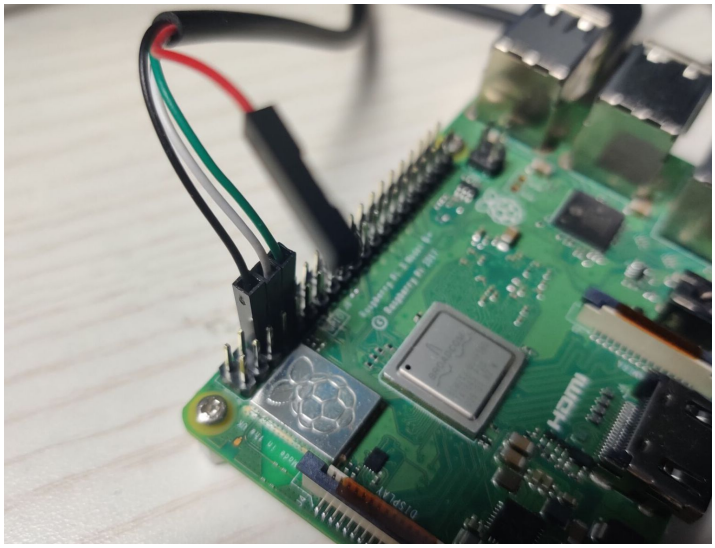

图 1: 树莓派的连接方式

4. 将 USB 线插到你的个人计算机上。如果你使用的是 Linux 系统，你需要借助 minicom 工具。依据你的发行版，使用 apt、yum、pacman 等包管理器安装 minicom，然后运行以下命令：

```
$ sudo minicom -D /dev/ttyUSB0
```

如果你使用的是树莓派 4b，则需要在 minicom 的 Serial port setup 选项中选择 230400 8N1 并保存。如果你使用的是 Windows 系统 +Linux 虚拟机的组合，你也可以直接在 Windows 系统上运行串口调试工具，可以在微软应用商店中搜索 “串口调试助手” 并且下载，或者是一些其它的串口调试工具；将端口选择为有 “USB-to-Serial” 字样的端口，同样波特率设置为 115200（树莓派 3b）或是 230400（树莓派 4b），数据位为 8，校验位为 None，停止位为 1，然后点击打开端口。

5. 启动树莓派的电源，即可在屏幕上看到输出，并且与其 shell 交互。如果使用的是 windows 系统的串口调试助手，则可能需要在向树莓派发送命令的时候手动在命令的最后添加一个回

车。

2.2 Qemu 运行

以往的实验能够很方便地在 Qemu 上进行，而不幸的是，Qemu 在树莓派上对于除了显示器以外的交互设备支持很差，很难模拟鼠标和键盘的输入，因此我们需要借助一些工具。你可以使用 tools/VirtualMouseKeyboard 中的工具，它会捕获宿主机的键盘和鼠标输入，并且通过 TCP 发送到 Qemu 的端口，在 ChCore 内部通过特殊的驱动程序来模拟 USB 键盘和鼠标的输入。

如果你使用 docker，你可以通过以下方式编译并且启动 ChCore：

```
# 进入到实验根目录
$ make
$ make qemu
```

然后，你可以使用 tools/VirtualMouseKeyboard 中的工具，输入运行 Qemu 的 Linux 系统的 IP 地址和端口号，点击连接即可。默认端口号是 10010 和 10011，你可以通过修改环境变量的方式修改 qemu 运行时监听的端口。

```
$ make qemu KEYBOARD_PORT=10086 MOUSE_PORT=10087
```

注意：由于虚拟机鼠标的特殊性，该工具不支持在虚拟机中运行，包括远程桌面、WSL 等。因此如果你使用的是 Windows+Linux 虚拟机的方式完成实验，请在宿主 Windows 系统中运行该工具，并使用 Windows 宿主系统可以访问的 Linux 客户系统的 IP 地址。在工具的使用过程中，你可以通过在命令行中添加两个启动参数来控制鼠标和滚轮的灵敏度，两个参数均为整数，分别表示鼠标和滚轮的灵敏度，默认是 2 和 10。基于不同系统，你可能需要灵活地调整这两个参数。虚拟键鼠工具使用 Unity 编写，其源码在 tools/VirtualMouseKeyboard/VirtualMouseKeyboard.unitypackage 中。

如果你在实验时无法使用 docker（我们的 docker 环境基于 x86_64 架构），例如在 Apple A 系列芯片上运行，你可以通过以下方式重新编译 qemu：

```
$ git clone -b stable-9.0 https://gitlab.com/qemu-project/qemu.git
$ cd qemu
$ git apply ${project_dir}/tools/qemu/qemu.patch # 其中 ${project_dir} 需
  ↪ 要替换为本实验的根目录
$ mkdir build
```

```
$ cd build
$ ../configure
$ make
$ echo 'export PATH="'$(pwd)':$PATH"' >> ~/.bashrc # 将当前路径添加到环境变
↪ 量
$ source ~/.bashrc
```

通过以下方式编译 ChCore：

```
$ make
# 此时因为 docker 没有启动，会报错
$ ./chbuild -l build
```

然后，通过以下指令启动 ChCore：

```
$ qemu-system-aarch64 -machine raspi4b -serial null -serial mon:stdio
↪ -serial tcp::10010,server,nowait -serial tcp::10011,server,nowait -m
↪ size=2G -kernel ./build/kernel.img
```

3 Demo

我们事先准备了一些 Demo，用于展示 ChCore GUI 的运行情况。在 demo_images 文件夹中，对于树莓派 3、树莓派 4 的实机以及模拟树莓派 4 的 Qemu 平台，我们都各自准备了一个镜像。对于两种实机，你需要将相应的镜像按照上文提到的位置，并且重命名为 kernel8.img；对于 Qemu 使用的镜像，你需要将其复制到 build 文件夹中（在第一次构建之前或者执行 make clean 之后 build 文件夹不存在，此时可以自己创建），然后按照上文提到的方式运行即可。

进入 ChCore 的 Shell 之后，你可以通过运行相应的命令来尝试相关的 Demo。我们一共准备了五个 Demo，分别是基于 lib100ask（https://github.com/100askTeam/lv_lib_100ask）的计算器、2048、文件管理器、Memory Game，以及基于开源 GBA 模拟器 VBANext 的 GBA 模拟器（https://github.com/FASTSHIFT/lv_gba_emu）。运行以下命令以启动相应 Demo：

```
$ lv_file.bin
$ lv_cal.bin
```

```
$ lv_2048.bin
$ lv_memgame.bin
$ gba_emu.bin -f A:gba_roms/{rom_name}
```

五条命令分别对应五个 Demo。其中在 GBA 模拟器的 Demo 中，我们准备了《古墓丽影》初代开源版本（https://github.com/XProger/OpenLara，rom 名为 OpenLara.gba）以及《宝可梦：绿宝石》（漫游 &TGB 联合汉化，rom 名为 Pokemon.gba）。

我们的 Demo 源码全部在 user/apps 目录下提供，在你完成本实验的 PART 1 和 PART 2 后，你也可以将 lib100ask 中其它 APP 添加到镜像中，或者是将一些其它的 GBA ROM 放置在 ramdisk/gba_roms 目录下以编译进 ChCore 的 ramdisk 中进行游玩体验。（目前存在一些未知原因导致的有些 ROM 在运行时会导致 Segmentation Fault 问题）

另外，对于下面的每个需要你填写代码的源码文件，我们都提供了一个参考实现的二进制文件，如果你在实验过程中遇到了困难，这些二进制文件或许可以帮助你暂时跳过这一部分，继续进行实验。你可以通过更改 CMakeLists.txt 中源码来将原本的 c 文件替换为这些二进制文件。以下是这些源码文件在 CMakeLists.txt 中的位置：

- PART 1：uconnection.c：user/chcore-libs/graphic/wayland/CMakeLists.txt:14
- PART 1：shm.c：user/chcore-libs/graphic/wayland/CMakeLists.txt:21
- PART 1：smm.c：user/chcore-libs/graphic/lvgl/lv_drivers/CMakeLists.txt:8
- PART 2：All 3 files：user/system-services/system-servers/gui/CMakeLists.txt:16

4 PART 0 Wayland 协议、ChCore GUI 架构简介

4.1 Wayland 简介

Wayland 是一个通信协议，规定了显示服务器与其客户机之间的通信方式，而使用这个协议的显示服务器称为 Wayland Compositor。它由 Kristian Høgsberg 于 2008 年发起，目标是用更简单的现代化视窗系统取代 X Window System。整个架构分为前后端。前端即 Wayland Client，是一个个使用 Wayland 来进行 UI 显示的用户程序；后端即 Wayland Compositor，负责将来自客户端的图像信息显示到屏幕上，以及将交互事件传递到客户端，在 ChCore 中，这个 Compositor 是 gui.srv，其源代码在 user/system-services/system-servers/gui/目录下。前后端端进行消息传递时，使用的是类似面向对象的 RPC 的接口，即每个连接中会维护若干的 Object，每个 Object 上

都有若干的方法，其中一些是前端调用后端的方法，被称为 Request；另一些是后端调用前端的方法，被称为 Event。每个对象在前端被称为一个 Proxy，而在后端被称为一个 Resource。

4.2 Wayland 编程模型

每个对象在被使用之前，需要在前后端进行注册。我们以表示鼠标箭头的 Pointer 对象为例，其中对于后端为前端提供的方法，使用如下方式进行注册：

```
static struct wl_pointer_interface wl_pointer_impl = {
    wl_pointer_set_cursor,
    wl_pointer_release,
}; //被注册的函数指针

r_pointer = wl_resource_create(c, &wl_pointer_interface, 7, id);
wl_resource_set_implementation(r_pointer, &wl_pointer_impl, p, NULL);
```

对于前端为后端提供的方法，通过以下方式进行注册：

```
const struct wl_pointer_listener pointer_listener = {
    pointer_enter_handler,
    pointer_leave_hanlder,
    pointer_motion_handler,
    pointer_button_handler,
    pointer_axis_handler,
};//被注册的函数指针

wl_pointer_add_listener(pointer, &pointer_listener, NULL);
```

这样，这些函数指针便被注册到了系统当中，当前端调用 **wl_pointer_set_cursor()** 函数时，就会调用到后端被注册的 wl_pointer_set_cursor() 函数；同样地，当后端调用 **wl_pointer_ send_enter()** 函数时，就会调用到前端所注册的 pointer_enter_handler() 函数。

具体地，前端需要先与后端服务器建立连接，获取后端全局的 Display 对象，通过这个 Display 对象即可进行通信。通过调用不同的接口，前后端之间进行 FrameBuffer 的绑定、提交，以及鼠标、键盘事件的传递等。具体协议可见https://wayland.app/protocols/wayland。另外，在 user/apps/wayland_demo.c 中提供了一个最简的 Wayland 前端程序的 Demo，你可以通过这个

Demo，结合官方的协议说明、或是网络上查找的资料，学习 Wayland 基本的编程模型。

4.3 ChCore GUI 架构

在 ChCore 中，Wayland Compositor 被编译为一个系统服务 gui.srv，默认地，我们已经通过配置文件指定这个进程开机启动。客户端通过 IPC 调用 gui.srv 提供的接口建立链接，随后通过 pipe 发送序列化数据的方式进行通信。

你可以直接使用 Wayland 协议进行编程，但是我们引入了一个被称为 LVGL 的嵌入式图形化界面的库，能够帮助你更加方便地开发出基于 Wayland 和后端通信的 GUI 程序。

后端能够通过我们预先编译的 circle.srv，即树莓派平台上的驱动库，来获取一个 Framebuffer 地址，以及对应鼠标事件和键盘事件的两个管道。在接收到来自前端的中的像素数据时，后端会根据前端提供的，每一帧中发生变化的区域信息，将该区域的像素点填充到硬件 Framebuffer 中；而与此同时，后端会不停地接收并且处理来自驱动的鼠标、键盘事件，解析之后发送给相应的客户端。

对于支持音频播放的机型，通过 libaudio_driver，前端直接和驱动进行交互，播放流式的或者固定长度的音频数据。

总体而言，ChCore 的 GUI 架构如图2所示。在本次实验中，我们重点关注其中的通过 Wayland Protocal 进行通信的部分，以及后端 Compositor 的实现。除此之外，你也将会学习如何借助（或者不借助，不建议）LVGL 编写一个在 ChCore 上运行的 GUI 程序。

5 PART 1 Wayland 协议实现

ChCore 上的 Wayland 协议的实现从标准的 Wayland 协议实现移植而来，这一标准实现是运行在 Unix 系的操作系统上的，鉴于 ChCore 对其特性支持不完整，我们基于 ChCore 的特性进行了一些修改。你可以参照标准实现进行这一部分实验。

你可以从 https://gitlab.freedesktop.org/wayland/wayland找到标准的 Wayland 协议实现。

5.1 前后端事件传递

在事件传递的结构中，前后端对于 in 和 out 各维护两个环形队列。通过两个 pipe（标准实现中为一个 domain Socket 连接）传递数据。

事件出口：一切事件函数的调用实际上是（前端：wl_proxy_marshal()，后端：wl_resource_post_ event()）函数的调用，这个函数会将所调用的函数序列化，然后写入环形 buffer 中。前后端都

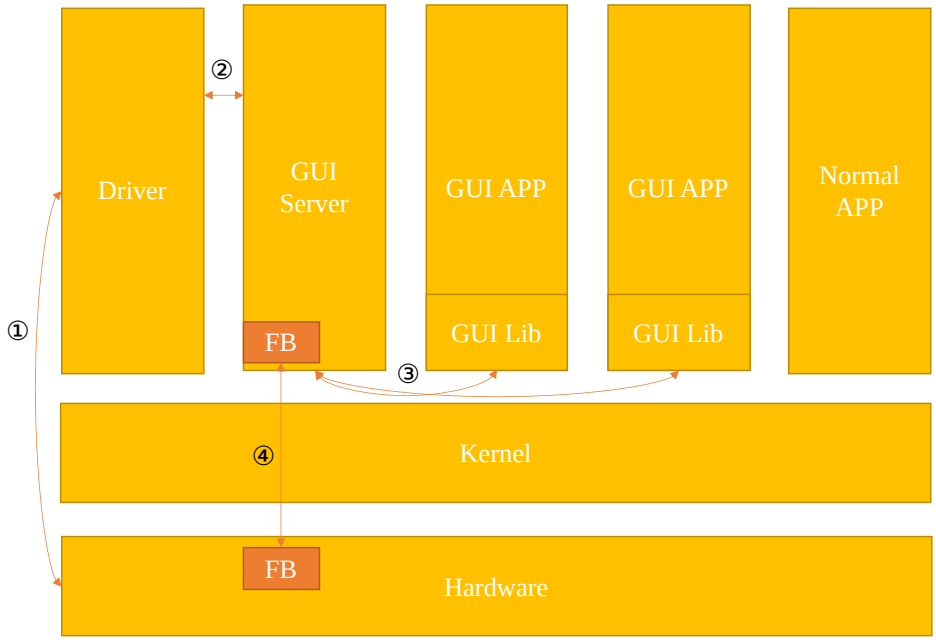

图 2: ChCore 总体 GUI 框架

①: MMIO/DMA

②: IPC Call/Pipe

③: Wayland Protocol/IPC Call

④: Memory Map

会定期调用（前端：display_flush()，后端：wl_display_flush_clients()），最终调用到 connection_flush()。

事件入口：客户端会维护一个队列 event_queue，里面是反序列化好的事件。通过 wl_display_read_events()，最终调用到 queue_events()，将环形队列中的数据反序列化加入 event 队列中；调用 wl_display_dispatch_queue_pending()，则会从队列中读取事件，最终通过 wl_closure_invoke() 进行函数调用。在后端，每个客户端都对应一个 wl_event_source 结构体（反之不成立，鼠标、键盘等事件源也是 event_source）。在注册成 event source 时，wl_event_loop_dispatch() 函数被注册为客户端类型 event source 的处理函数，因此后端会周期性地通过 wl_event_loop_dispatch() 函数遍历这些客户端，通过注册的处理函数函数从客户端对应的 pipe 中读取数据、进行反序列化并且调用对应的函数。

消息传递：发送事件时，将事件序列化写入环形 buffer；在 connection_flush() 时从环形 buffer 读取并使用 pipe（标准实现为 domain socket）传递；读取时从 pipe（domain socket）读入环形 buffer，然后从环形 buffer 读取，并进行反序列化。

问题 1

参考官方的标准实现，阅读 user/chcore-libs/graphic/wayland/src/connection.c 中的 wl_connection_read()（比较简单的实现是 ChCore 上当前的实现，一次把 pipe 读空或者把 buffer 读满；标准实现中是一直循环到 read 返回 0），wl_connection_write()，wl_connection_flush() 函数，并且将它们补全。(将 domain socket 的信息传递移植到 ChCore 的 Pipe)。

类似于传统 Unix 系统中的文件，ChCore 中的系统资源由 Capabilities 管理。因此当我们需要传递一块共享内存时，我们无法像 Wayland 标准实现中那样通过 Domain Socket 的 Control Message 传递的方式传递 fd，而是需要传递 Capability。因此在 ChCore 中，Wayland 事件参数的序列化中，字母'h' 用来表示这个参数是一个 Capability。在序列化和反序列化时，需要调用 usys_cap_transfer() 接口来完成 Capability 的传递。

在 ChCore 中的用户态中，每个进程所拥有的每个 Capability 由一个整数表示，而就像 fd 一样，这个整数无法直接传递给其它进程使用。使用 usys_cap_transfer() 接口，可以将一个 Capability 从一个进程传递到另一个进程。具体地，这个函数输入接收方进程的 Capability 以及要传递的 Capability 号，然后将这个 Capability 传递给接收方进程，函数返回的数字将会是对方进程中这个 Capabilitiy 所对应的 Cap 号。

问题 2

在 Unix 系系统中，当调用的函数某个参数是 fd 时，Domain Socket 具有跨进程传递 fd 的功能；而 ChCore 的 libPipie 不具有这个功能。查看 user/chcore-libs/graphic/wayland/src/connection.c 中的 wl_closure_marshal()、wl_connection_demarshal() 以及 chcore_copy_pmos_to_connection() 三个函数，并进行补全，使用 usys_cap_transfer() 接口完成序列化和反序列化时 cap 的传递。查看 Wayland 的标准实现，说说在标准实现中 fd 是如何在客户端和 Compositor 之间进行传递的。仔细分析较其而言，你在 ChCore 中使用的传递 cap 的方法是否有什么风险?

注释 1：ChCore 中的 libPipe

ChCore 中的 libPipe 是基于一段共享的物理内存（即 PMO，我们在下一部分会提到）的环形 buffer，在创建之后，你可以使用如下几个接口使用它：

```
u32 pipe_read(const struct pipe *pp, void *buf, u32 n);
u32 pipe_read_exact(const struct pipe *pp, void *buf, u32 n);
u32 pipe_write(const struct pipe *pp, const void *buf, u32 n);
u32 pipe_write_exact(const struct pipe *pp, const void *buf, u32 n);
bool pipe_is_empty(const struct pipe *pp);
bool pipe_is_full(const struct pipe *pp);
```

其中，pipe_read() 和 pipe_read_exact() 的区别在于，前者输入一个大小，从 pipe 中读取最大为这个大小的数据；而 pipe_read_exact() 则会保证读取的数据大小等于输入的参数 n，如果无法读取 n 位的数据则不进行任何读取并且返回 0。同样地，pipe_write_exact() 用于保证写入的数据大小等于输入的参数 n，如果无法写入 n 位的数据则不进行任何写入并且返回 0。这个特性可以用于传递大小为特定大小的结构，例如一个结构体，而不必担心写入或读出的数据不完整的问题。

5.2 共享内存

在 Wayland 中，屏幕上显示的像素信息在前后端之间是通过一块共享内存区域实现的。当前端调用 wl_shm_create_pool() 之前，需要创建一个物理内存对象（在标准实现中，这一过程通过 mmap() 创建一块匿名内存，并将这块内存的 fd 通过 Domain Socket 共享给后端；而在 ChCore 中，我们创建一个 PMO 类型的内核对象，然后通过 usys_cap_transfer() 传递给后端）。在后端接收到这个 PMO 之后，会调用 shm_create_pool()，并且将这个 PMO 映射到自己的地址空间中。在后端接收到前端传递的 PMO 之后，前后端之间的共享内存区域就建立了。后续从这个 pool 中创建 buffer 时，只需要在前后端同步 buffer 在这个共享内存区域中的 offset 和大小即可。

然而，在我们的 Compositor 的实现中，这个共享内存区域的基地址是由后端单独复制保存一份的，这也就意味着，当前端调用 wl_shm_pool_resize() 的时候，假如后端的虚拟地址空间中的这块内存区域的基地址发生了变化而 Compositor 并没有感知到，那么后端将无法正确地访问到这块内存区域。因此，我们需要保证在发生 resize 的时候，后端的内存区域的基地址保持不变。幸运的是，我们的 PMO 中有按需映射这一类型，也就是说，我们可以创建一个逻辑上任意大的

物理内存区域，而当真正访问这块区域中的每个页的时候，内核才会真正地分配物理内存，并且将其加入到页表中。也许这个特性可以被利用来解决我们上面提到的问题?

问题 3

依据上述描述，参照 Wayland 标准实现中两个函数的作用，借助按需分配的 PMO，在 user/chcore-libs/graphic/wayland/src/wayland-shm.c 中完成 ChCore 版本的 shm_create_pool()、shm_pool_unref() 和 shm_pool_finish_resize()。与此同时，在 user/chcore-libs/graphic/lvgl/lv_drivers/wayland/smm.c 的 alloc_pool() 函数中补全创建和映射 PMO 的部分，以便 LVGL 库能够正常使用。你可以参考我们给出的 user/apps/wayland_demo.c 中的代码，但请注意这其中的代码并非最终的正确实现。

注释 2：ChCore 中的 PMO 对象和用户态内存映射管理

ChCore 中的 PMO 是一类内核对象，用 Capabilitiy 表示。这一对象表示一段物理内存空间。你可以通过 usys_create_pmo() 的系统调用接口来创建一个 PMO 对象。PMO 对象的类型有几种，其中本实验中关心的有 PMO_DATA 和 PMO_SHM 类型。其中 PMO_DATA 类型会在创建时就分配物理内存，而 PMO_SHM 类型则是按需分配的，即只有在访问到这个 PMO 中的某个页面的时候，内核才会真正地分配物理内存并且写入页表，因此你可以创建一个任意大的 PMO，以此来表示一个在逻辑上任意大的物理地址空间。而对于 PMO_DATA 类型的 PMO，在创建 PMO 时，系统就会为这个 PMO 分配连续的大小为你传入 size 的物理内存；而当你进行虚拟地址的映射时，这一部分物理内存和对应虚拟内存的映射关系会直接被写入页表。如果你已经完成了 Lab 2，这一部分你应当非常熟悉。

在 ChCore 中，你可以通过以下的用户态接口手动进行虚拟内存空间的分配和映射：

```
1 void *chcore_auto_map_pmo(cap_t pmo, unsigned long size, unsigned
  ↪  long perm);
2 void chcore_auto_unmap_pmo(cap_t pmo, unsigned long vaddr, unsigned
  ↪  long size);
```

例如，加入你想要创建一块大小为 2MB，权限为读写的物理内存区域，并且将其映射到用户态空间，你可以使用如下代码：

```
cap_t pmo = usys_create_pmo(2 * 1024 * 1024, PMO_DATA);
void *vaddr = chcore_auto_map_pmo(pmo, 2 * 1024 * 1024, VM_READ |
↪  VM_WRITE);
// 此时你可以通过 vaddr 来访问这块物理内存
// ...
chcore_auto_unmap_pmo(pmo, (unsigned long)vaddr, 2 * 1024 * 1024);
```

注意，当你对一个 PMO 进行 unmap 然后再进行 map 时，你可能会得到不同的虚拟地址。

5.3 实验结果与测试-1

当你的实验进行到这里，恭喜你，你已经完成了实验的第一部分。你可以通过运行 user/apps/wayland_demo.c 来测试你的实现。在启动 ChCore 后，在 ChCore 中的 Shell 中运行第一个 Demo。

```
$ wayland_demo.bin
```

在这个 Demo 中，你可以看到一个简单的窗口，无法进行任何交互，并且这个简单的窗口在不停地闪烁。在下一部分中，我们将对这些问题进行解决。

这个 Demo 的源码在 user/apps/wayland_demo.c 中，你可以通过查看这个文件来了解如何使用 Wayland 协议来编写一个简单的 GUI 程序。

6 PART 2 后端 Compositor

6.1 窗口的像素信息

经过对于 Wayland 协议接口的基本了解，现在的你已经清楚 Wayland 协议是如何在前后端之间传递像素信息的：前端通过调用 wl_surface_attach_buffer() 将一个共享内存的 buffer 绑定到一个 surface 上，通过 wl_surface_damage_buffer() 标记发生更改的区域，然后通过 wl_surface_commit() 将这个 surface 提交给后端；后端在接收到这个 surface 之后，会将这个 buffer 的像素信息填充到硬件的 Framebuffer 中。

在 ChCore 的实现中，后端在每个窗口都维护一个额外的 buffer，当收到前端的 damage 信息时，会将这个信息记录下来；而当收到前端的 commit 信息的时候，才会根据 damage 的区域，将这个 buffer 的像素信息填充到硬件的 Framebuffer 中。

注释 3：ChCore 中的 libg2d

在 ChCore 中，我们实现了一个名为 g2d（graphics 2D）的库，其接口和实现参考了 Windows 中的 GDI 库，用于各个窗口、屏幕之间各个区域的计算和传递。如果你对于 GDI 库很熟悉，下面的部分你可以略过。g2d 中有几个关键数据结构如下：

- RECT 结构体用于表示一个方形区域，你可以用它来标记窗口中一个区域，例如，记录一个一个窗口中 damage 的信息。

```
typedef struct rect RECT;
```

- Region 结构体表示一个屏幕上的区域，这个区域可以是离散的，其实质是一个 RECT 的集合，你可以用下面这些接口来对其进行操作。

```
typedef struct region *PRGN;
int set_rc_region(PRGN rgn, const RECT *rc);
inline void clear_region(PRGN rgn);
inline bool region_null(const PRGN rgn);
/* Region Combination Mode */
enum RGN_COMB_MODE {
        RGN_ADD, /* dst = src1 + src2 */
        RGN_AND, /* dst = src1 & src2 */
        RGN_COPY, /* dst = src */
        RGN_DIFF, /* dst = src1 - src2 */
        RGN_OR, /* dst = src1 | src2 */
        RGN_XOR /* dst = src1 ^ src2 */
};
inline int combine_region(PRGN dst, const PRGN src1, const PRGN
    ↪  src2, enum RGN_COMB_MODE mode);
void move_region(PRGN rgn, int x, int y);
```

其中，set_rc_region() 函数能够使用一个方形区域对 Region 进行初始化；combine_region 能够对 Region 进行集合之间的操作，值得注意的是，这些操作并不是将 RECT 视为一个元素，而是将 RECT 中的每一个像素点视为一个元素，因此你可以通过这个接口来计算两个 Region 之间的交集、并集等。而 move_region 用于做坐标变换，你可以用它进行窗口坐标系和屏幕坐标系之间的转换。

- Bitmap 结构体表示一个位图，而 DC 结构体则表示一个设备上下文，包含位图和深度等信息。在本实验中，你可能用到以下接口进行绘制

```
inline void dc_select_clip_rgn(PDC dc, PRGN rgn);
inline int bitblt(const PDC dst, int x1, int y1, int width, int
  ↪  height, const PDC src, int x2, int y2, enum ROP rop);
```

第一个函数为特定的设备上下文选择一个剪裁区域，在这个设备上下文上的绘制将会被限制在这个区域中进行；而第二个函数则是将一个设备上下文的内容绘制到另一个设备上下文中，其参数中的 x，y 分别表示源 DC 和目标 DC 的起始位置，而 width，height 则表示绘制的区域的大小。实际被绘制的区域是由参数指定的区域、两个 DC 的有效区域和目标 DC 剪裁区域的交集。被绘制的区域中，每一个像素点都将根据 ROP 指定的操作处理到目标 DC 中，即 dst = src ROP dst。你可以使用 ROP_SRCCOPY 来使得 dst = src，即直接复制源 DC 中的像素数据到目标 DC 中。

在 ChCore 中，窗口的结构体中保存的信息大致如下：

```
enum WIN_STATE {
    WIN_UNINIT,
    WIN_INIT,
    WIN_MOVED,
    WIN_PAINTED,
    WIN_REMOVED,
    WIN_HIDDEN,
};
struct wm_window { /*Properties*/
    // ...
```

```
        PDC dc;
        // ...
        PDC buffer_attached_dc;
        // ...
        PRGN damage;
        PRGN rgn_upd;
        PRGN rgn_upd_tmp;
        RECT rc;
        // ...
        RECT new_rc;
        // ...
        enum WIN_STATE state;
        // ...
};
```

窗口的状态有如下含义：

- WIN_UNINIT：窗口未初始化，即从来没有被 Paint 过，此时窗口的 dc 里没有内容，因此合成器在绘制时应当忽略这个窗口。
- WIN_INIT：窗口已经初始化，可以被合成器绘制。但是由于窗口的内容并没有发生变化，窗口的 dc 和前端是同步的，因此不需要进行重新绘制。
- WIN_MOVED：窗口的位置发生了变化，整个窗口的像素信息、以及窗口的旧位置的信息都需要被更新。
- WIN_PAINTED：窗口的像素信息发生了变化，需要进行绘制。UNINIT 和 INIT 状态的窗口经过绘制后都会变为 PAINTED 状态。
- WIN_REMOVED：窗口被移除，此时应当将这个窗口原始的位置的像素信息进行更新，然后删除该窗口。
- WIN_HIDDEN：窗口被隐藏，此时应当将这个窗口原始的位置的像素信息进行更新，但是不删除该窗口。这种状态目前用于维护 subsurface_destroy 的语义。

后端通过如下方式依据前端的调用来更新窗口的像素信息：

- 当前端调用 wl_surface_attach_buffer() 时，窗口会将这个 buffer（实质上是共享内存的的引用，这也解释了上面我们提到的共享内存的基地址不能改变）保存在窗口的 buffer_-attached_dc，便于后续使用。
- 当前端调用 wl_surface_damage_buffer() 时，窗口会将这个 damage 的区域添加到窗口的 rgn_upd_tmp 中。
- 当前端调用 wl_surface_commit() 时，会调用到窗口会将 rgn_upd_tmp 中的区域合并到 rgn_upd 中，然后将 rgn_upd 中的区域填充到窗口的 dc 中。此时，如果发现 attach 的 buffer 大小发生了变化，说明前端将整个窗口的像素信息都更新了，那么窗口会将整个窗口的区域添加到 rgn_upd 中，并且窗口本身会被 resize。调整窗口的状态信息。

以上流程保证了前后端之间像素信息的传递，前端调用 attach、damage、commit 接口提交的信息能够在合适的时机被正确地填充到窗口的 DC 中，下一步，就是合成器如何将这些 DC 中的信息绘制到硬件的 Framebuffer 中。

问题 4

依据上述描述以及代码中的注释，完成 user/system-services/system-servers/gui/wm_window.c 中的 window_paint() 函数。

光标的显示和后端的其它窗口有所不同，尽管前端可以使用 wl_pointer_set_cursor() 接口将一个窗口提交为光标。在后端，光标的图案由一个 dc 维护，而其位置和 hotspot 等信息维护在 pointer 结构体中。当前端使用这个接口将一个 surface 提交为 cursor surface 时，其相关信息会被记录在后端的 surface 结构体中；而当这个 surface 被 commit 时，surface 结构体中的数据就会被应用。同时我们还会维护一个默认的光标数据，当当前没有 cursor surface 被应用时，这个默认的光标数据就会被应用。相关的代码在 user/system-services/system-servers/gui/wm_cursor.c 和 user/system-services/system-servers/gui/wayland_backend.c 中有所体现。

6.2 合成器

在 ChCore 上，当 GUI 系统初始化的时候，Framebuffer 已经被填充到 struct wingmr 的 winmgr 成员 dc 中，你只需要通过全局变量 wm.dc 访问这个设备上下文，向其中填入的像素便会被显示在显示器上。

注释 4：ChCore 在树莓派上的 Framebuffer

此处我们简要介绍一下 ChCore 在树莓派上是如何获取硬件 Framebuffer 的：树莓派的很多硬件由其 GPU（VideoCore）控制，并且提供了一种被称为 Mailbox 的机制用于 CPU 和 VideoCore 之间的通信，其中在 Mailbox 的 Tag Channel 中，有一个命令（Tag）用于获取 Framebuffer 的地址（Allocate buffer，0x00040001）。我们使用了开源的树莓派驱动库 Circle，它提供一个 IPC 服务，当其它进程通过这个服务向其索要 Framebuffer 地址时，它 Lazy 地通过 Mailbox 获取这个 Framebuffer 的地址，设置好相关参数，并且返回这个地址。在 gui.srv 运行的开始，我们通过 iom_probe_hdmi() 函数向驱动请求这个 Framebuffer 的地址，然后使用这个地址创建全局变量 wm.dc，这个变量便是合成器在绘制时的目标设备上下文。

如果只是希望完成这个实验中的题目，你并不需要了解这些驱动相关的知识。但如果你对于这一部分内容感兴趣，你可以查看https://github.com/raspberrypi/firmware/wiki/Mailboxes来了解更多关于树莓派 Mailbox 的信息。

后端的窗口以两种数据结构被组织起来：一个链表（zlist）和一个森林（plist）。链表维护了所有窗口的指针，而因为 subsurface 的存在，森林用于记录各个窗口之间的父子关系；在遍历绘制时，我们遍历这个森林的所有根节点，然后对每棵树进行后序遍历进行绘制。

到现在为止，对于每个窗口，我们拥有它自身的 Framebuffer，其位置信息、状态信息和其中需要更新的区域的信息。依据这些信息，合成器将这些窗口绘制到屏幕上。合成器的绘制过程主要分为两个阶段：第一个阶段是遍历所有窗口，确定整个屏幕上需要更新的区域；第二各阶段再次遍历所有窗口，并且更新这个窗口和整个屏幕更新区域的交集。

对于没有窗口存在的背景，我们将其作为一个特殊的窗口处理，这个窗口不连接前端，并且被添加在 plist 的最后。

问题 5

在第二个阶段遍历所有窗口时，我们在对某个窗口进行绘制时，会记录下屏幕上被绘制的区域的信息，并且在后续窗口被绘制时除掉这一部分区域，这样做会导致什么结果？结合你在上一部分实验中看到的实验现象，谈谈为什么要这样做？

另外，我们上面提到，绘制时更新的是某个窗口和整个屏幕更新区域的交集，为什么不是窗口得到更新区域和屏幕更新区域的交集？

第一阶段我们对于每个窗口调用 wm_process_update_window() 函数，其中我们会维护一

个 rgn_upd 区域和一个 rgn_clip 区域，以得到整个屏幕上需要更新的区域；其中前者表示屏幕上需要更新的区域，后者表示在后续的遍历中不会再被加入到 rgn_upd 中的区域，在大部分时候这两个集合都是相等的，但是在某些情况下，比如你确定一个区域在本轮显示中不会被更新时，你可以将这个区域加入到 rgn_clip 中，这样在后续的遍历中，这个区域就不会被加入到 rgn_upd 中。具体的添加规则我们在上一部分已经大概地介绍过，依据窗口不同的状态，“需要被更新” 的部分会被加入到 rgn_upd 中；因为已经在 rgn_upd 中被添加过一次，所以同时也会被添加到 rgn_clip 中；对于确定没有更新的 WIN_INIT 类型的窗口，其区域会被加入到 rgn_clip 中。

第二阶段我们会对每个窗口调用 wm_window_repaint() 函数，正如我们上面已经提到过的，这个过程中我们会使用上个阶段得到的 rgn_upd 区域，并重新维护一个 rgn_clip 区域。对于每个窗口，我们求取这个窗口的整体区域和 rgn_upd 的交集，然后减掉 rgn_clip 中的区域，将得到的这个区域作为这个窗口的更新区域绘制到屏幕上，同时将这个区域加入到 rgn_clip 中。这个过程保证了只有需要更新的区域被绘制到屏幕上，同时保证了每个像素点只被绘制一次。

问题 6

依据上述描述以及代码中的注释，完成 user/system-services/system-servers/gui/wmmgr.c 中的 wm_process_update_window() 和 wm_window_repaint() 函数。

对于光标，我们会在一个窗口被标记为 cursor surface 时就将其从窗口列表中删除；上一节我们已经介绍了光标数据（包括 hotspot 和像素数据）的生命周期，现在我们关注它是如何被绘制到屏幕上的。上一节我们提到，我们使用一个 dc 来维护光标的像素数据，而绘制模块负责将这个 dc 绘制到屏幕上：当列表中的所有窗口都被绘制完成之后，进行光标的绘制。在 update 阶段，我们需要把光标的旧位置添加到 rgn_upd 中，以保证光标不会在旧的位置有残留；而在 paint 阶段，只需要简单地将光标绘制到屏幕上即可，而此时如果光标的位置没有发生变化，那么光标区域中不属于 rgn_upd 区域的像素集合是不会发生变化的，因此我们在绘制的时候可以不考虑这部分。

6.3 实验结果与测试-2

当你的实验进行到这里，恭喜你，你已经完成了实验的第二部分的前半段。再来运行 wayland_demo.bin，此时你可以看到右上角的光标，以及可以正常显示的窗口。

6.4 交互

完成了显示之后，我们最后需要关注的是交互部分，即来自驱动的键盘鼠标数据是如何被 GUI 后端处理，并且转变为 Wayland 事件反馈给前端的。

注释 5：ChCore 中的键盘、鼠标驱动以及数据模型

ChCore 中的键盘、鼠标驱动也是移植自第三方开源驱动库 Circle，分别通过一个 Pipe 传递。GUI 后端通过 IPC 调用拿到这两个 Pipe。当硬件的键盘鼠标事件到来时，驱动会将这些事件处理为特定的格式写入 Pipe；而这些 Pipe 在后端会被注册为我们上面曾经提到的 EventLoop，周期性地从 Pipe 中读取数据，然后调用相应的函数进行处理。在后端处理键鼠数据的函数位于 user/system-services/system-servers/gui/wayland_backend.c 中，分别为 pointer_handle_raw_event() 和 keyboard_handle_raw_event()。

键盘和鼠标的事件分别由各自的结构体表示，其中鼠标的事件用以下结构体表示：

```
    typedef struct {
        u32 buttons;
        int x;
        int y;
        int scroll;
} raw_pointer_event_t;
```

其中 buttons 是一个 bitmap，表示鼠标按键的按下状态。在 ChCore 中，我们只关心所有鼠标都拥有的三个按键，依照 USB HID 标准，第 0 位（0x1）、第 1 位（0x2）、第 2 位（0x4）分别表示左键、右键和中键。而后面的 x、y、scroll 分别表示鼠标在 x 方向、y 方向和滚轮上的移动距离。注意这三个值并非绝对位置，而是位置的 Δ。

键盘的事件由以下结构体表示：

```
typedef struct {
    u32 mods_depressed;
    u32 mods_latched;
    u32 mods_locked;
    u32 keys[6];
} raw_keyboard_event_t;
```

其中前三个值表示修饰键的状态，分别表示按下、暂时锁定和锁定的修饰键。在这里，ChCore 的实现不符合 USB HID 规定，对于键位码，我们是这样规定的：

```
#define SHIFT_MASK    (1 << 0)
#define CAP_LOCK_MASK (1 << 1)
#define CTRL_MASK     (1 << 2)
#define ALT_MASK      (1 << 3)
#define NUM_LOCK_MASK (1 << 4)
#define SCROLL_LOCK_MASK (1 << 5)
```

与 USB HID 标准不同的是，我们不区分左右修饰键，并且将三组 LOCK 键也加入到了 depressed 修饰键中。而结构体中的 keys 表示当前被按下的键码，最多支持 6 个键，每个按键对应的键码遵循 USB HID 规定。

USB HID 标准参考：https://www.usb.org/sites/default/files/hut1_21.pdf

鼠标和键盘都会维护一个 focus 指针，用于标记当前聚焦的窗口。当鼠标从一个窗口进入到另一个窗口时，鼠标的聚焦窗口就会发生改变；而当鼠标在新的窗口中点击之后，键盘的聚焦窗口才会改变。当鼠标或键盘聚焦在背景窗口时，这一指针就会被设置为 NULL。同时，我们还会维护鼠标和键盘当前的按键状态，以便在新的事件到来时判断按键的状态是否发生了变化。

对于处理鼠标事件的函数 pointer_handle_raw_event()，基本的流程如下：

1. 鼠标聚焦窗口的引用计数处理。
2. 如果鼠标位置发生了变化，更新光标的位置。
3. 如果先前客户端调用了 wl_surface_move()，则将需要移动的窗口的位置更新。
4. 如果鼠标所在的窗口没有发生变化，则发送 motion 事件。
5. 如果鼠标所在的窗口发生了变化，则调用 pointer_change_focus() 函数。这个函数依据鼠标原先的聚焦窗口和现在的聚焦窗口，发送 leave 和 enter 事件。
6. 如果鼠标的按键状态发生了变化，发送 button 事件；如果鼠标的滚轮发生了变化，发送 axis 事件。
7. 如果鼠标的按键被按下，调用 keyboard_change_focus() 函数，这个函数会判断键盘的聚焦窗口是否发生了变化，如果发生了变化，则发送 leave 和 enter 事件。以及调用 wm_set_top_win() 函数，将鼠标所在的窗口置于最上层。

对于键盘事件的处理函数 keyboard_handle_raw_event()，基本的流程如下：

1. 键盘聚焦窗口的引用计数处理。
2. 如果键盘的按键状态发生了变化，发送 key 事件。
3. 如果键盘的修饰键状态发生了变化，发送 modifiers 事件。

关于 serial：在 Wayland 中，键鼠事件都需要一个 Serial 来标记事件发生的前后顺序，在 ChCore 中，我们使用全局的 operate_serial 来标记事件的序列号，每次向前端发送需要序列号的事件的时候，序列号就会递增；这样后端在收到前端的相应的时候，便可以知道事件的前后顺序，从而拒绝过期的事件。

问题 7

依据上述描述以及代码中的注释，完成 user/system-services/system-servers/gui/wayland_backend.c 中的 pointer_handle_raw_event() 和 keyboard_handle_raw_event() 函数。

6.5 实验结果与测试-3

恭喜你，当你进行到这里时，你已经实现了整个 GUI 系统的基本功能。现在，你可以运行 wayland_demo.bin，你可以使用鼠标左键拖动窗口，并且使用鼠标右键点击窗口以改变窗口的颜色。除此之外，其余的 Demo 也变得可用，你可以尝试运行本文第 3 节中的 Demo 来检查你的实现是否正确。除此之外，请同时运行多个 GUI 程序保证你的实现能够正确地处理多个窗口，以及它们的相互遮挡关系。

如果你使用的是树莓派 4b 实机，你可以在 user/apps/lv_gba_emu/port/gba_port_audio.c 中依照注释说明将相应的代码取消注释，将 3.5mm 耳机插入树莓派 4b 上的 3.5mm 耳机接口，这样你就可以运行一个具有声音的 GBA 模拟器了。

7 PART 3 编写你自己的 GUI 程序

到现在为止，你已经参与完成了 ChCore 上整个 GUI 系统的实现，并且已经了解了其运作原理了。接下来，让我们来点更酷的事情：编写你自己的 GUI 程序，让它在 C 和 Core 上运行吧！现在是———自由发挥时间！

或许你也发现了，直接使用 Wayland 接口进行 GUI 编程并非一件容易的事情，因为它是一个底层的协议，需要你自己实现很多细节。因此，在 ChCore 中，我们一般使用 LVGL 库来进行 GUI 编程，这是一个开源的 GUI 库，提供了很多现成的控件和样式，你可以很容易地使用它来编写你的 GUI 程序。在本节中，我们将介绍如何使用 LVGL 库来编写一个简单的 GUI 程序。在使用之前，确保你已经正确完成了问题 3 中关于 LVGL 的部分。

在正式开始之前，我们建议你阅读 LVGL 的官方文档（https://docs.lvgl.io/master/），这样你可以更好地了解 LVGL 的使用方法。而在一切开始之前，你需要为 LVGL 创建一个窗口并绑定到一个 LVGL 上下文，在此之后，你的绘制都将在这个上下文中进行。具体地，在 ChCore 中，使用 LVGL 库进行开发的基本编程模型如下：

```
#include <unistd.h>
#include <lvgl.h>
#include "lv_drivers/wayland/wayland.h"

#define H_RES (500)
#define V_RES (500)

int main()
{
    lv_init();
    lv_wayland_init();
    lv_disp_t * disp = lv_wayland_create_window(H_RES, V_RES, "Window
    ↪  Title", NULL);
    lv_disp_set_default(disp);
    lv_group_t * g = lv_group_create();
    lv_group_set_default(g);
    lv_indev_set_group(lv_wayland_get_keyboard(disp), g);

    // Your Code


    while(lv_wayland_window_is_open(disp)){
```

```
22          usleep(1000 * lv_wayland_timer_handler());                  //! run
            ↪  lv task at the max speed
23      }
24      lv_wayland_deinit();
25      return 0;
26  }
```

在这个模板中，我们首先初始化了 LVGL 库，然后初始化了 Wayland 后端，创建了一个窗口并绑定到一个 LVGL 上下文，然后设置了默认的显示器和输入设备组。在这之后，你可以在你的代码中使用 LVGL 库提供的 API 进行绘制。最后，我们进入一个循环，不断地调用 lv_wayland_timer_handler() 函数，这个函数会处理所有的事件，包括键盘、鼠标事件等。当窗口被关闭时，我们退出循环，然后释放资源。

例如，在你的代码部分，你可以使用如下代码来创建一个简单的按钮：

```
1 lv_obj_t * btn = lv_btn_create(lv_scr_act(), NULL);
2 lv_obj_set_pos(btn, 10, 10);
3 lv_obj_set_size(btn, 100, 50);
4 lv_obj_t * label = lv_label_create(btn, NULL);
5 lv_label_set_text(label, "Button");
```

编写好你的代码之后，你可以使用以下方法将其添加到 ChCore 项目中：

1. 将你的代码文件添加到 user/apps/目录下。
2. 在 user/apps/CMakeLists.txt 中添加你的代码文件，并链接到相应的库。例如：

```
1 add_executable(lvgl_demo.bin lvgl_demo.c)
2 target_link_libraries(lvgl_demo PRIVATE lvgl)
3 target_link_libraries(lvgl_demo PRIVATE lv_drivers)
```

3. 重新编译 ChCore，你的文件 lvgl_demo.bin 便会出现在 Ramdisk 中。

我们不推荐你不借助 LVGL，直接使用 Wayland 协议进行 GUI 编程，因为这样会使得你的工作变得复杂。但是，如果你对此感兴趣，你可以查看 user/apps/wayland_demo.c 中的代码，这个文件中展示了如何使用 Wayland 协议进行 GUI 编程。注意：这个文件并非一个正确的版本，基于你

上面做过的题目，这个文件需要进行一定程度的修改。同时，在 CMakeLists.txt 中，你需要将你的文件链接到 wayland_client 上。你需要阅读 user/system-services/system-servers/gui/wayland_-backend.c 中的代码，以了解我们实现了 Wayland 标准协议中的哪些接口，避免使用那些未实现的接口，或者添加这些接口的实现。

现在，你可以运行你的 GUI 程序了！

注释 6：GUI Guider

为了协助你更加方便地使用 LVGL 进行编程，这里我们推荐一个第三方的、不开源但是免费的工具：GUI Guider（图3）。GUI Guider 是一个图形化的工具，由恩智浦半导体开发，可以帮助你快速地生成 LVGL 代码，你可以通过拖拽控件、设置属性等方式来生成你的 GUI 程序。GUI Guider 生成的代码需要进行一定的调整以放置于我们上面提到的编程框架中，但是它可以帮助你快速地生成一个基本的 GUI 程序。你可以在https://www.nxp.com/design/design-center/software/development-software/gui-guider:GUI-GUIDER下载 GUI Guider。

值得注意的是，如果你需要导入一个图片资源，GUI Guider 是不支持我们所使用的 32bit ARGB 格式的，因此你需要在它生成图片的代码表示之后，手动地将其替换为我们的格式。你可以使用 LVGL 的图像转换工具来将你的图片转换为这个格式（https://lvgl.io/tools/imageconverter）。在页面中选择 v8 选项，然后选择你的图片，最后选择 CF_-TRUE_COLOR_ALPHA 格式，点击 Convert 按钮，你就可以得到一个 C 文件，将这个文件中的数组中 COLOR_DEPTH 为 32 的部分替换到你的代码中，然后在图片的代码文件中修改相关信息即可。

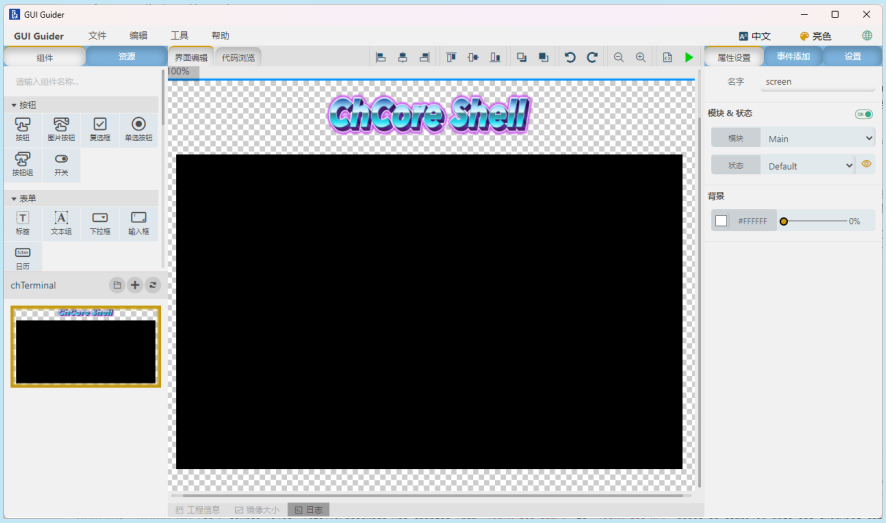

图 3: GUI Guider 用户界面

注释 7：ChCore 上的音频播放

如果你需要你的 GUI 程序播放音频，你可以使用 ChCore 提供的音频播放服务。在 ChCore 中，我们只支持 RAW 格式的音频数据的播放，你可以使用 ChCore 中的 libaudio_-driver 来连接到音频驱动，进行音频的播放。你需要在你的文件中包含 audio_driver.h，并且将你的代码链接到 audio_driver。具体使用方法如下：

```
1 void play_sound(const void* buffer, uint32_t nBitsPerSample,
  ↪ uint32_t nChannels, uint32_t nSamples);
2 typedef struct audio_buffer_handler audio_buffer_handler_t;
3 audio_buffer_handler_t* connect_audio_driver(uint32_t
  ↪ nBitsPerSample, uint32_t nChannels, int64_t frames_latency_max);
4 void disconnect_audio_driver(audio_buffer_handler_t* handler);
5 int write_to_audio_buffer(audio_buffer_handler_t* handler, const
  ↪ void* data, size_t nSamples);
```

如果你只需要播放一段固定长度的音频，那么你只需要使用 play_sound() 函数，其中的 buffer 指向你的音频数据，nBitsPerSample 表示每个采样的位数（8 （unsigned sound

data）或 16 （signed sound data））, nChannels 表示声道数（1 或 2）, nSamples 表示采样数。如果你需要播放一个动态的音频流，那么你需要使用 connect_audio_driver() 函数来连接到音频驱动，然后使用 write_to_audio_buffer() 函数来写入音频数据，最后使用 disconnect_audio_driver() 函数来断开连接。frames_latency_max 表示你允许后端播放相较于前端输入的最大延迟帧数，每帧为 512 个采样点。我们在 GBA Emulator 中将这个参数设置为 2。

注意，在只有你使用树莓派 4b 实机的情况下，你才能够使用这个音频播放服务。在 QEMU 上或树莓派 3b 上，这个服务是不可用的。另外，ChCore 的音频驱动的采样率固定为 44100Hz，因此你需要将你的音频数据转换为这个采样率。

问题 8

自由发挥时间！编写你自己的 GUI 程序，让它在 ChCore 上运行！你可以尝试编写一个小游戏，例如贪吃蛇、俄罗斯方块等，也可以尝试编写一个小工具，例如计算器、日历等。你可以参考 user/apps/lv_lib_100ask/中的代码。

8 实验总结

在本次实验中，我们学习了如何在 ChCore 上实现一个简单的 GUI 系统。我们首先了解了 GUI 系统的基本原理，包括前端和后端的交互方式，以及绘制和事件处理的流程。然后，我们深入研究了 ChCore 中 GUI 系统的实现细节，包括窗口管理、合成器、光标和交互处理等。我们通过完成一系列的练习，掌握了如何编写 GUI 系统的关键函数和数据结构。最后，我们使用 LVGL 库编写了自己的 GUI 程序，并在 ChCore 上成功运行。

通过本次实验，希望你已经掌握了以下知识和技能：

- 理解基于 Wayland 的 GUI 系统的基本原理和工作流程
- 熟悉在特定的操作系统上实现 GUI 系统的方法
- 掌握窗口管理、合成器、光标和交互处理等关键函数和数据结构的编写方法
- GUI 程序的编写和调试

再次恭喜你完成了本次实验，祝你在操作系统的学习道路上学习愉快！