tty 和 x 的切换流程

作者：郭隆基

时间：*[2023-09-25 一 10:31]*

UNIONTECH

简单介绍一下概念和学习路线

目录

这篇文档可以在：https://guolongji.xyz/post/tty-x-switch/ 这里看到更新。

第 1 章 概念

- Console

Console 意即控制台。Console 出现在操作系统诞生之前，那个时候的老式计算机还只能通过按钮开关等方式控制计算机的运转，而 Console 就是将控制计算机器件集中起来的面板。由于现代计算机不再使用控制台来控制电脑，所以现代操作系统中的 Console 一般指控制台终端。

- Terminal

Terminal 意即终端。顾名思义，是连接到计算机上的终端设备。当操作系统出现之后，不再使用控制台与计算机交互，而是使用命令，为了能够输入这些命令，出现了电传打字机（Teletype），程序输入输出可以通过电传打字机打印到屏幕上。所以 Terminal 其实是指一种监视计算机输入输出的硬件。但是现代操作系统中使用更多的是软件仿真终端。

- TTY

TTY 即 TeleTYpe，意即电传打字机。TTY 与 Terminal 初期是同一个概念，电传打字机连接到计算机上便是一台终端设备。当时的 TTY 需要使用 UART 驱动来传输数据，而现代计算机中的 TTY 直接将输出数据渲染成视频信号，输出到显示器中，这些操作都运行在内核态。目前只能在连接键盘与显示器的计算机上使用 TTY。

- PTY

PTY 即 Pseudo TeleTYpe，意即伪 TTY。PTY 是现代计算机中使用终端模拟软件或者 SSH 连接所使用的伪终端，运行在用户态。现代操作系统使用的终端都是伪终端，通过终端软件对终端进行模拟。PTY 为了保证能够与 TTY 兼容，采用了主从结构，分为 PTY slave side 与 PTY master side。

- PTY slave side

PTY slave side 的通过 TTY 驱动实现。PTY slave side 与 PTY master side 进行连接，将程序输入输出发送给 PTY master side 并显示出来。

- PTY master side

PTY master side 会将程序的输入输出显示在终端模拟软件中。例如当时用 SSH 远程连接时，PTY master side 与 SSH 连接，交换数据。

- PTMX

ptmx 是 Linux 操作系统中的一个特殊设备文件，它代表了伪终端（pseudo-terminal）主设备。伪终端是一种虚拟设备，用于提供交互式终端功能，让程序能够与终端进行通信。

ptmx 的全称是"Pseudo Terminal Master for X"，它是伪终端机制的主控端，通过打开 ptmx 设备可以获得一个伪终端从设备文件（例如 /dev/pts/0）。

在 Linux 中，当一个程序打开 ptmx 设备并请求一个新的伪终端时，内核会创建一对相互连接的伪终端从设备（slave），然后把主设备（master）的文件描述符和从设备的路径返回给程序。程序可以使用这对从设备和主设备进行通信，就像在与真实终端进行交互一样。

伪终端在很多应用中非常有用，比如远程登录、终端模拟器、串口通信等。它提供了一个可编程的、终端兼容的接口，使得程序可以以交互的方式与其他终端应用或设备进行通信，而无需直接依赖物理终端。

总结来说，ptmx 是 Linux 中用于创建和管理伪终端的主设备文件，它为程序提供了与终端交互的接口，为终端模拟、远程登录等操作提供了基础支持。

说人话就是，每打开一个 terminal 的程序 ptmx 就会创建一个新的"/dev/pts" 的下的设备。sysctl kernel.pty.max，能查到这个值是 4096。伪终端的数量也是有限制的。

- dev 下的 ttyX

可以 chvt 进入，即终端，而 pts 目录下的是不能那么进去的。

VT 不需要显卡驱动的支持。framebuffer 相关的代码在内核的 drivers/video/fbdev/fbcon.c 当中。

VT 驱动的代码在 drivers/tty/vt/vt.c 当中。其中的 visual_init 函数中调用 con_init 函数。

流程是内核启动初始化。

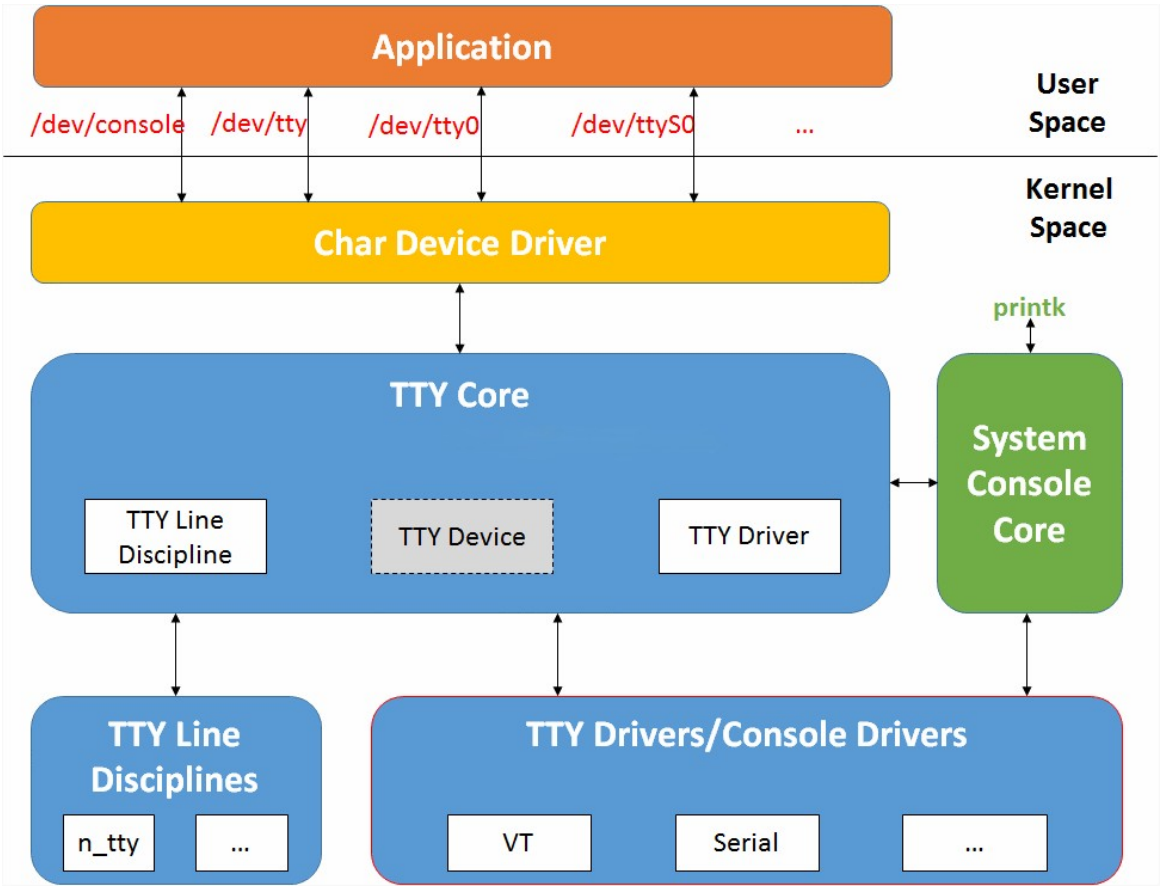


图 1.1: *tty* 子系统架构图

第 2 章 如何将一个 tty 和一个 xorg 关连起来？

分配新用户的 TTY 涉及到内核中几个关键的函数。以下是其中的一些函数：

1.get_unused_tty_index()：用于获取一个未被使用的 TTY 索引。每个 TTY 都有一个唯一的索引号，内核使用这个索引号来标识不同的 TTY 设备。

2.tty_alloc_driver()：用于为新的 TTY 实例分配一个 TTY 驱动结构体。TTY 驱动结构体保存了与 TTY 设备相关的信息和操作。

3.tty_init_dev()：用于初始化新的 TTY 设备。这个函数会设置 TTY 的状态、配置串口传输参数以及注册 TTY 设备到系统中。

4.tty_open()：用于打开 TTY 设备。当一个用户切换到一个新的 TTY 时，内核会调用这个函数来打开对应的 TTY 设备。

5.vt_do_activate()：用于激活虚拟终端（Virtual Terminal，VT）。在 Linux 中，每个 TTY 都对应一个 VT，用于提供文本终端的功能。这个函数会激活新的 VT，并将其与对应的 TTY 关联起来。

这些函数位于内核的 TTY 子系统中，负责管理和处理 TTY 设备。通过调用这些函数，内核能够分配和管理多个 TTY 设备，并在用户切换到不同的 TTY 时分配适当的资源和处理相应的操作。

xorg 通过命令行参数 -vt 1 来绑定。

2.1 vt 的切换流程

Ctrl-Alt-F2 在只有一个用户时，切换是切到 tty 的界面。但是如果有多个登陆的用户时就会切换到其它用户的图形界面。

当用户切换到新的 TTY 时，涉及到内核的一些操作来实现切换。下面是内核切换 TTY 的详细流程：

用户请求切换 TTY：用户可以通过按下 Ctrl+Alt+F1 到 F6（一般情况下，Linux 系统提供了 6 个 TTY 终端）的组合键来请求切换到相应的 TTY。

用户空间程序处理请求：当用户按下组合键后，一个特殊的信号（如 SIGINT）将发送给前台进程组中的所有进程。在这种情况下，TTY 驱动程序会接收到信号并执行相应的操作。

TTY 驱动程序检测信号并通知内核：TTY 驱动程序会检测到信号，并将其传递给内核。内核会响应该信号，并开始进行 TTY 切换的操作。

内核切换控制台：内核会根据信号中指定的 TTY 编号，将当前活动的 TTY 切换到新的 TTY。它会关闭当前 TTY 的输入和输出，并重新配置设备以与新的 TTY 关联。

内核执行 TTY 切换操作：内核会执行以下操作来完成 TTY 切换：

关闭当前 TTY 设备：内核将关闭当前 TTY 设备的输入和输出，在切换期间停止与 TTY 设备的通信。加载新 TTY 设备：内核会初始化和配置新的 TTY 设备，包括打开新 TTY 设备的输入和输出通道。切换虚拟终端结构：内核会更新虚拟终端结构，以反映当前活动的 TTY 设备。用户空间程序刷新屏幕：在 TTY 切换完成后，用户空间程序将被激活，并负责更新新 TTY 的屏幕内容。

总体而言，内核切换 TTY 的过程涉及到信号处理、设备配置和数据传输等操作。内核负责管理和控制 TTY 设备的状态，以确保正确的 TTY 切换和用户体验。这样，用户可以在不同的 TTY 中进行并发的文本模式会话。

第 3 章 dbus

- lightdm(org.freedesktop.DisplayManager)

登录管理：LightDM 提供用户界面，使用户能够输入用户名和密码登录系统。它支持不同的登录方式，如图形界面登录、远程登录和自动登录。

会话管理：LightDM 管理用户登录后的会话过程。它允许用户选择默认的窗口管理器或桌面环境，并为每个用户保存其首选设置。

多用户支持：LightDM 支持多个用户账号，并为每个用户提供独立的登录环境。这意味着多个用户可以同时登录到同一台计算机上，而不会相互干扰。

主题和样式定制：LightDM 允许用户自定义登录界面的外观和样式。用户可以选择不同的主题、背景图像、字体和颜色方案，以创建自己喜欢的登录界面。

扩展性和集成：LightDM 支持插件和扩展，可以与各种其他软件和服务集成。它可以与不同的窗口管理器、桌面环境和身份验证系统一起使用。

总的来说，LightDM 是一个灵活、可定制和易于使用的显示管理器，它提供了一个简单而强大的界面来管理用户登录会话。

- greeter(lightdm-deepin-greeter)
- deepin-authenticate(com.deepin.daemon.Authenticate)
- systemd-logind(org.freedesktop.login1)
- dde-lockservice(org.deepin.dde.LockService1)
- dde-system-daemon(org.deepin.dde.Accounts1)
- accounts-daemon(org.freedesktop.Accounts)

以上服务，基本都是 system bus，可以通过 d-feet 去看对应的接口。可以用 dbus-send，gdbus 和 qdus 来发送 dbus 信号，信号的

3.1 dbus 接口的使用

我们有一个服务 deepin-authenticate 是用 golang 来实现的。golang 的 dbus 库的文档在这里：
https://pkg.go.dev/github.com/godbus/dbus
Kwin 中用的是 qdbus 的库。
dbus 的核心概念就是总线消息。

第 4 章 linux 图形学中的一些概念

4.1 DM，WM 与 X Display Manager Control Protocol（XDMCP）

- 显示管理器 DM 又称为“登陆管理器”。有 lightdm，xdm（x display manager），sddm（Simple Desktop Display Manager），gdm（gnome Display manager）。

可以用登陆脚本代替 DM，比如 xinit + startx 的方式。xinit 程序允许用户手动启动 Xorg 显示服务器。startx 脚本是 xinit 的一个前端。

xorg-xinit 这个包切换用户最终调用的是：

- 窗口管理器

kwin（dde，支持 X11 和 wayland）、sway（支持 wayland）、wayfire（支持 wayland）、hyperland（现在最火的 wayland wm）、awesome、i3wm、mutter（gnome）……

在 uos 中可以安装 dde-dconfig-editor 可以选择用 kwin 来启动窗口管理器。

https://www.x.org/releases/current/doc/

man X，本地可以看到。关于 X11 的文档都在这个里面了，是英文，内容很多。

lightdm 的配置文件在：/etc/lightdm/ 里；

xorg 的配置文件在这里：/etc/X11/ 里；

通过 pstree 能看到图形的启动流程：

```
uos@guolongji:~$ pstree --ascii
systemd-+ ......
        |-lightdm-+-Xorg---3*[{Xorg}]
        |         |-lightdm-+-startdde-+-DeepinAIAssista---9*[{DeepinAIAssista}]
        |         |         |         |-DeepinVoiceWake---{DeepinVoiceWake}
        |         |         |         |-agent---2*[{agent}]
        |         |         |         |-bd-qimpanel.wat---sleep
        |         |         |         |-chrome-+-2*[cat]
        |         |         |         |        |-2*[chrome---7*[{chrome}]]
        |         |         |         |        |-chrome---chrome---24*[{chrome}]
        |         |         |         |        |-chrome-sandbox---chrome-+-chrome-+-19*[chrome---12*[{chrome}]]
        |         |         |         |        |                         |       |-3*[chrome---13*[{chrome}]]
        |         |         |         |        |                         |       |-chrome---14*[{chrome}]
        |         |         |         |        |                         |       |-chrome---4*[{chrome}]
        |         |         |         |        |                         |       |-2*[chrome---7*[{chrome}]]
        |         |         |         |        |                         |       `-chrome---15*[{chrome}]
        |         |         |         |        |                         `-chrome-sandbox---nacl_helper
        |         |         |         |        `-25*[{chrome}]
        |         |         |         |-dde-calendar-se---2*[{dde-calendar-se}]
        |         |         |         |-dde-clipboard---5*[{dde-clipboard}]
        |         |         |         |-dde-desktop---27*[{dde-desktop}]
        |         |         |         |-dde-dock---25*[{dde-dock}]
        |         |         |         |-dde-file-manage---28*[{dde-file-manage}]
        |         |         |         |-dde-launcher---24*[{dde-launcher}]
        |         |         |         |-dde-lock---24*[{dde-lock}]
```

```
|         |         |         |-dde-osd---7*[{dde-osd}]
|         |         |         |-dde-polkit-agen---5*[{dde-polkit-agen}]
|         |         |         |-dde-printer-hel---9*[{dde-printer-hel}]
|         |         |         |-dde-session-dae---50*[{dde-session-dae}]
|         |         |         |-deepin-deepinid---30*[{deepin-deepinid}]
|         |         |         |-deepin-defender---5*[{deepin-defender}]
|         |         |         |-emacs-+-codeium_languag-+-codeium_languag---15*[{codeium_languag}]
|         |         |         |       |                 `-15*[{codeium_languag}]
|         |         |         |       |-epdfinfo
|         |         |         |       |-python3---8*[{python3}]
|         |         |         |       `-10*[{emacs}]
|         |         |         |-emacs-+-codeium_languag-+-codeium_languag---17*[{codeium_languag}]
|         |         |         |       |                 `-19*[{codeium_languag}]
|         |         |         |       |-epdfinfo
|         |         |         |       |-mu
|         |         |         |       |-python3---8*[{python3}]
|         |         |         |       |-python3
|         |         |         |       `-5*[{emacs}]
|         |         |         |-evince---4*[{evince}]
|         |         |         |-kitty-+-bash---pstree
|         |         |         |       `-3*[{kitty}]
|         |         |         |-kwin_no_scale---kwin_x11---10*[{kwin_x11}]
|         |         |         |-wps---wps---9*[{wps}]
|         |         |         `-62*[{startdde}]
|         |         `-2*[{lightdm}]
|         `-2*[{lightdm}]
|- ......
```

第 5 章 AMD 显卡的架构演进

https://medium.com/high-tech-accessible/an-overview-of-amds-gpu-architectures-884432a717a6

sudo apt install mesa-utils

uos@guolongji:~$ glxinfo -B

name of display: :0
display: :0 screen: 0
direct rendering: Yes
Extended renderer info (GLX_MESA_query_renderer):
    Vendor: X.Org (0x1002)
    Device: AMD CAICOS (DRM 2.50.0 / 4.19.0-amd64-desktop, LLVM 7.0.1) (0x6779)
    Version: 19.2.6
    Accelerated: yes
    Video memory: 2048MB
    Unified memory: no
    Preferred profile: core (0x1)
    Max core profile version: 3.3
    Max compat profile version: 3.1
    Max GLES1 profile version: 1.1
    Max GLES[23] profile version: 3.1
Memory info (GL_ATI_meminfo):
    VBO free memory - total: 2047 MB, largest block: 2047 MB
    VBO free aux. memory - total: 1021 MB, largest block: 1021 MB
    Texture free memory - total: 2047 MB, largest block: 2047 MB
    Texture free aux. memory - total: 1021 MB, largest block: 1021 MB
    Renderbuffer free memory - total: 2047 MB, largest block: 2047 MB
    Renderbuffer free aux. memory - total: 1021 MB, largest block: 1021 MB
Memory info (GL_NVX_gpu_memory_info):
    Dedicated video memory: 2048 MB
    Total available memory: 3069 MB
    Currently available dedicated video memory: 2047 MB
OpenGL vendor string: X.Org
OpenGL renderer string: AMD CAICOS (DRM 2.50.0 / 4.19.0-amd64-desktop, LLVM 7.0.1)
OpenGL core profile version string: 3.3 (Core Profile) Mesa 19.2.6
OpenGL core profile shading language version string: 3.30
OpenGL core profile context flags: (none)
OpenGL core profile profile mask: core profile

OpenGL version string: 3.1 Mesa 19.2.6
OpenGL shading language version string: 1.40
OpenGL context flags: (none)

OpenGL ES profile version string: OpenGL ES 3.1 Mesa 19.2.6
OpenGL ES profile shading language version string: OpenGL ES GLSL ES 3.10

- Caicos 和 TeraScale 是什么关系？

  Caicos 和 TeraScale 是 AMD 显卡架构中的两个不同概念，它们之间存在一定的关系。

  TeraScale 是 AMD 显卡架构中的一个历史阶段，它首次于 2007 年推出，被广泛应用于 Radeon HD 3000、4000、5000 和 6000 系列显卡中。TeraScale 架构采用了传统的 VLIW（Very Long Instruction Word）设计，通过组合多个简单指令来实现复杂的计算任务。TeraScale 架构在当时是非常先进的，但随着计算单元的增多，复杂

度也逐渐增加，导致处理器设计难度和功耗成倍提升。

而 Caicos 则是 TeraScale 2 架构中的一个代表，它是 AMD Radeon HD 6400 和 6500 系列显卡所采用的架构。TeraScale 2 架构在 TeraScale 架构的基础上进行了一些优化，如引入了更高效的纹理单元和几何单元，并支持 DirectX 11 和 OpenGL 4.1 等新的标准。同时，TeraScale 2 架构还采用了更加先进的 40nm 制程工艺，使得功耗和热量控制得到了更好的平衡。

因此，可以说 Caicos 是 TeraScale 2 架构中的一个代表，是 AMD 在当时对 TeraScale 架构的改进和升级。虽然 Caicos 和 TeraScale 有所区别，但它们都是 AMD 显卡架构中的重要部分，并共同推动了 AMD 显卡技术的发展。

第 6 章 AMD 驱动代码的模块划分和代码介绍

- DCE

AMD 的 DCE (Display Core Engine) 模块是 AMD 显卡中的一个重要组件，它主要负责图形处理单元 (GPU) 和显示输出之间的通信和协调工作。DCE 模块具有以下功能：

1. 显示控制：DCE 模块负责显卡的显示控制功能，包括显示输出的配置、分辨率管理、刷新率控制等。它能够接收来自 GPU 的图像数据，并将其转换为适当的格式并输出到连接的显示设备上。

2. 多显示器支持：DCE 模块支持多显示器配置，可以同时驱动多个显示设备，例如多个显示器、投影仪等。它能够管理和控制多个显示输出，并确保它们按照用户的设置正确运行。

3. 显示连接管理：DCE 模块负责检测和管理显示设备的连接状态。它能够自动检测连接的显示设备，并在需要时进行重新配置和重新协商，以确保正常的显示输出。

4. 显示特性增强：DCE 模块还提供了一些显示特性的增强功能，例如色彩管理、色彩空间转换、多显示器融合等。这些功能可以提供更好的显示效果和用户体验。

总的来说，DCE 模块在 AMD 显卡中扮演着重要的角色，它是 GPU 与显示输出之间的桥梁，负责管理、控制和增强显示功能，使用户能够获得良好的图形和显示效果。

- DCE 模块和 display 模块是什么关系？

display 模块包含了 DCE 模块的所有功能，同时还集成了其他相关的硬件和软件组件，例如视频解码器、视频编码器、色彩管理等。它提供了一种精简和高效的架构，以便更好地支持多显示器配置和多种显示输出配置。

除了 DCE 模块的主要功能外，display 模块还提供了其他的增强功能，例如：

1. 包括 HDMI、DisplayPort 和 DVI 等多种显示输出标准的支持。

2. 支持硬件加速的视频解码和编码功能，能够加速高清视频的播放和转码。

3. 能够支持高分辨率和广色域的显示输出，以提供更好的画质和色彩表现效果。

总之，DCE 模块是 AMD 显卡中的一个核心组件，而 display 模块则是一个集成了 DCE 模块功能的子系统，并提供了其他相关的硬件和软件组件，以实现更好的画质和用户体验。

- UVD

UVD（Unified Video Decoder）是一个硬件模块，用于在 AMD 显卡上进行视频解码的加速处理。UVD 模块主要用于解码高清视频和其他常见的视频格式，以减轻 CPU 的负担并提供更流畅的视频播放体验。

UVD 模块有以下主要功能：

1. 视频解码加速：UVD 模块使用专门的硬件电路和算法来加速视频解码过程。它支持多种视频编码标准，包括 H.264、MPEG-2、VC-1 等，并能够快速解码高分辨率的视频内容。

2. 解码负载分担：通过使用 UVD 模块进行视频解码，可以将解码任务从 CPU 转移至显卡，减轻 CPU 的负担。这样可以释放 CPU 资源用于其他计算任务，提高系统整体性能和响应速度。

3. 节能和热量控制：UVD 模块在进行视频解码时，能够有效地管理功耗和热量产生。由于视频解码是一项相对固定的任务，UVD 模块可以根据需要动态调整功耗和频率，以提供最佳的性能和能效平衡。

总之，UVD 模块是 AMD 显卡中的一个硬件加速模块，用于视频解码任务。它能够减轻 CPU 负担，提供更流畅的视频播放体验，并具有节能和热量控制的优化功能。

- GMC

GMC（Graphics Memory Controller）是一个重要的硬件模块，它负责显卡中的内存管理和数据传输。GMC 模块可以有效地管理 GPU 内部的显存，从而提高图形渲染性能和效率。

GMC 模块主要有以下功能：

1. 内存分配和释放：GMC 模块可以管理 GPU 内部的显存，自动分配和释放内存，确保图形处理单元 (GPU) 能够快速访问并使用可用的显存。

2. 数据传输：GMC 模块负责数据在 GPU 内部和显存之间的传输。它可以在 GPU 内部的不同模块之间进行高速数据传递，以实现更快的图形渲染和数据处理。

3. 内存优化：GMC 模块能够对显存进行优化，以提高数据读写效率，减少延迟和提高响应速度。这些优化包括内存预取、内存压缩、数据压缩和解压等。

4. 带宽管理：GMC 模块可以管理 GPU 和显存之间的数据带宽，确保数据能够快速传输，并根据需要动态调整带宽分配，以提供更好的性能和能效平衡。

总之，GMC 模块是 AMD 显卡中的一个核心组件，它负责显存管理和数据传输，并能够提供内存优化和带宽管理等功能，以提高图形渲染性能和效率。

切换用户的时候一定是内核重新获取了 edid 的，内核当中获取 edid 的时间比较长。amdgpu 获取 edid 主要涉及以下几个文件：

x86-kernel/drivers/gpu/drm/amd/amdgpu/amdgpu_atombios.c

x86-kernel/drivers/gpu/drm/amd/amdgpu/amdgpu_i2c.c

x86-kernel/drivers/gpu/drm/amd/display/amdgpu_dm/amdgpu_dm.c

因为 oland 显卡不用 display 的代码，所以要从前两个文件来分析代码的调用。

具体的调用过程我下次讲。

通过 dmesg|grep drm 可以查到：

```
[    0.920573] [drm] radeon kernel modesetting enabled.
[    0.965343] [drm] amdgpu kernel modesetting enabled.
[    0.965343] [drm] amdgpu version: 5.11.32.40512
[    0.965344] [drm] OS DRM version: 4.19.0
[    0.967268] [drm] initializing kernel modesetting (OLAND 0x1002:0x6611 0x1462:0x3740 0x87).
[    0.967277] [drm] register mmio base: 0xA0300000
[    0.967278] [drm] register mmio size: 262144
[    0.967283] [drm] add ip block number 0 <si_common>
[    0.967284] [drm] add ip block number 1 <gmc_v6_0>
[    0.967284] [drm] add ip block number 2 <si_ih>
[    0.967284] [drm] add ip block number 3 <gfx_v6_0>
[    0.967285] [drm] add ip block number 4 <si_dma>
[    0.967285] [drm] add ip block number 5 <si_dpm>
[    0.967286] [drm] add ip block number 6 <dce_v6_0>
[    0.967286] [drm] add ip block number 7 <uvd_v3_1>
[    0.975071] [drm] BIOS signature incorrect 5b 7
[    0.975303] [drm] vm size is 64 GB, 2 levels, block size is 10-bit, fragment size is 9-bit
[    0.975335] [drm] Detected VRAM RAM=2048M, BAR=256M
[    0.975335] [drm] RAM width 64bits GDDR5
[    0.975358] [drm] amdgpu: 2048M of VRAM memory ready
[    0.975359] [drm] amdgpu: 7898M of GTT memory ready.
[    0.975361] [drm] GART: num cpu pages 262144, num gpu pages 262144
[    0.975901] [drm] Supports vblank timestamp caching Rev 2 (21.10.2013).
[    0.975902] [drm] Driver supports precise vblank timestamp query.
[    0.976076] [drm] Internal thermal controller with fan control
[    0.976082] [drm] amdgpu: dpm initialized
[    0.976096] [drm] AMDGPU Display Connectors
[    0.976096] [drm] Connector 0:
[    0.976097] [drm]   HDMI-A-1
[    0.976097] [drm]   HPD2
```

```
[    0.976098] [drm]    DDC: 0x1950 0x1950 0x1951 0x1951 0x1952 0x1952 0x1953 0x1953
[    0.976098] [drm]    Encoders:
[    0.976098] [drm]      DFP1: INTERNAL_UNIPHY
[    0.976099] [drm] Connector 1:
[    0.976099] [drm]    VGA-1
[    0.976099] [drm]    DDC: 0x194c 0x194c 0x194d 0x194d 0x194e 0x194e 0x194f 0x194f
[    0.976100] [drm]    Encoders:
[    0.976100] [drm]      CRT1: INTERNAL_KLDSCP_DAC1
[    0.976141] [drm] Found UVD firmware Version: 64.0 Family ID: 13
[    0.976809] [drm] PCIE gen 3 link speeds already enabled
[    1.530160] [drm] UVD initialized successfully.
```

oland 显卡用的是 dce_v6_0 这个模块：

6.1 激活 tty 的流程

分配新用户的 TTY 涉及到内核中几个关键的函数。以下是其中的一些函数：

1.get_unused_tty_index()：用于获取一个未被使用的 TTY 索引。每个 TTY 都有一个唯一的索引号，内核使用这个索引号来标识不同的 TTY 设备。

2.tty_alloc_driver()：用于为新的 TTY 实例分配一个 TTY 驱动结构体。TTY 驱动结构体保存了与 TTY 设备相关的信息和操作。

3.tty_init_dev()：用于初始化新的 TTY 设备。这个函数会设置 TTY 的状态、配置串口传输参数以及注册 TTY 设备到系统中。

4.tty_open()：用于打开 TTY 设备。当一个用户切换到一个新的 TTY 时，内核会调用这个函数来打开对应的 TTY 设备。

5.vt_do_activate()：用于激活虚拟终端（Virtual Terminal，VT）。在 Linux 中，每个 TTY 都对应一个 VT，用于提供文本终端的功能。这个函数会激活新的 VT，并将其与对应的 TTY 关联起来。

这些函数位于内核的 TTY 子系统中，负责管理和处理 TTY 设备。通过调用这些函数，内核能够分配和管理多个 TTY 设备，并在用户切换到不同的 TTY 时分配适当的资源和处理相应的操作。

ioctl(fd, VT_ACTIVATE, vt)，通过这个调用内核。

对应的是内核的 drivers/tty/vt/vt_ioctl.c 里面的两个处理函数：

vt_ioctl 和 vt_compat_ioctl 两个处理函数：

```
/*
 * ioctl(fd, VT_ACTIVATE, num) will cause us to switch to vt # num,
 * with num >= 1 (switches to vt 0, our console, are not allowed, just
 * to preserve sanity).
 */
case VT_ACTIVATE:
        if (!perm)
                return -EPERM;
        if (arg == 0 || arg > MAX_NR_CONSOLES)
                ret =  -ENXIO;
        else {
                arg--;
                console_lock();
```

```
            ret = vc_allocate(arg);
            console_unlock();
            if (ret)
                    break;
            set_console(arg);
    }
    break;
```

或者是：

```
    {
    // ......

    case VT_ACTIVATE:
    case VT_WAITACTIVE:
    case VT_RELDISP:
    case VT_DISALLOCATE:
    case VT_RESIZE:
    case VT_RESIZEX:
            goto fallback;

    /*
     * the rest has a compatible data structure behind arg,
     * but we have to convert it to a proper 64 bit pointer.
     */
    default:
            arg = (unsigned long)compat_ptr(arg);
            goto fallback;
    }

    return ret;

fallback:
    return vt_ioctl(tty, cmd, arg);
```

黑屏并不是说这个系统调用导致的，而是切换至新的 xorg，新的 xorg 会从读 edid 导致的黑屏。这个我是花了一段时间才想明白的。上面的代码就不展开了。

通过 perf 找到的调用堆栈如下：

drm_ioctl
drm_ioctl_kernel
drm_mode_getconnector
drm_helper_probe_single_connector_modes
amdgpu_connector_dvi_detect
amdgpu_connector_get_edid
drm_get_edid
drm_do_get_edid

drm_do_probe_ddc_edid
bit_xfer
drivers/gpu/drm/amd/amdgpu/amdgpu_connectors.c

```
static void amdgpu_connector_get_edid(struct drm_connector *connector)
{
        struct drm_device *dev = connector->dev;
        struct amdgpu_device *adev = dev->dev_private;
        struct amdgpu_connector *amdgpu_connector = to_amdgpu_connector(connector);

        if (amdgpu_connector->edid)
                return;

        /* on hw with routers, select right port */
        if (amdgpu_connector->router.ddc_valid)
                amdgpu_i2c_router_select_ddc_port(amdgpu_connector);

        if ((amdgpu_connector_encoder_get_dp_bridge_encoder_id(connector) !=
             ENCODER_OBJECT_ID_NONE) &&
            amdgpu_connector->ddc_bus->has_aux) {
                amdgpu_connector->edid = drm_get_edid(connector,
                                                      &amdgpu_connector->ddc_bus->aux.ddc);
        } else if ((connector->connector_type == DRM_MODE_CONNECTOR_DisplayPort) ||
                   (connector->connector_type == DRM_MODE_CONNECTOR_eDP)) {
                struct amdgpu_connector_atom_dig *dig = amdgpu_connector->con_priv;

                if ((dig->dp_sink_type == CONNECTOR_OBJECT_ID_DISPLAYPORT ||
                     dig->dp_sink_type == CONNECTOR_OBJECT_ID_eDP) &&
                    amdgpu_connector->ddc_bus->has_aux)
                        amdgpu_connector->edid = drm_get_edid(connector,
                                                              &amdgpu_connector->ddc_bus->aux.ddc);
                else if (amdgpu_connector->ddc_bus)
                        amdgpu_connector->edid = drm_get_edid(connector,
                                                              &amdgpu_connector->ddc_bus->adapter);
        } else if (amdgpu_connector->ddc_bus) {
                amdgpu_connector->edid = drm_get_edid(connector,
                                                      &amdgpu_connector->ddc_bus->adapter);
        }

        if (!amdgpu_connector->edid) {
                /* some laptops provide a hardcoded edid in rom for LCDs */
                if (((connector->connector_type == DRM_MODE_CONNECTOR_LVDS) ||
                     (connector->connector_type == DRM_MODE_CONNECTOR_eDP)))
                        amdgpu_connector->edid = amdgpu_connector_get_hardcoded_edid(adev);
        }
```

```
}
```

drivers/gpu/drm/drm_edid.c

```
/**
 * drm_get_edid - get EDID data, if available
 * @connector: connector we're probing
 * @adapter: I2C adapter to use for DDC
 *
 * Poke the given I2C channel to grab EDID data if possible.  If found,
 * attach it to the connector.
 *
 * Return: Pointer to valid EDID or NULL if we couldn't find any.
 */
struct edid *drm_get_edid(struct drm_connector *connector,
                          struct i2c_adapter *adapter)
{
        struct edid *edid;

        if (connector->force == DRM_FORCE_OFF)
                return NULL;

        if (connector->force == DRM_FORCE_UNSPECIFIED && !drm_probe_ddc(adapter))
                return NULL;

        edid = drm_do_get_edid(connector, drm_do_probe_ddc_edid, adapter);
        if (edid)
                drm_get_displayid(connector, edid);
        return edid;
}
EXPORT_SYMBOL(drm_get_edid);
```

drivers/gpu/drm/drm_edid.c

```
/**
 * drm_do_get_edid - get EDID data using a custom EDID block read function
 * @connector: connector we're probing
 * @get_edid_block: EDID block read function
 * @data: private data passed to the block read function
 *
 * When the I2C adapter connected to the DDC bus is hidden behind a device that
 * exposes a different interface to read EDID blocks this function can be used
 * to get EDID data using a custom block read function.
 *
 * As in the general case the DDC bus is accessible by the kernel at the I2C
 * level, drivers must make all reasonable efforts to expose it as an I2C
 * adapter and use drm_get_edid() instead of abusing this function.
```

```
 *
 * The EDID may be overridden using debugfs override_edid or firmare EDID
 * (drm_load_edid_firmware() and drm.edid_firmware parameter), in this priority
 * order. Having either of them bypasses actual EDID reads.
 *
 * Return: Pointer to valid EDID or NULL if we couldn't find any.
 */
struct edid *drm_do_get_edid(struct drm_connector *connector,
        int (*get_edid_block)(void *data, u8 *buf, unsigned int block,
                              size_t len),
        void *data)
{
        int i, j = 0, valid_extensions = 0;
        u8 *edid, *new;
        struct edid *override;

        override = drm_get_override_edid(connector);
        if (override)
                return override;

        if ((edid = kmalloc(EDID_LENGTH, GFP_KERNEL)) == NULL)
                return NULL;

        /* base block fetch */
        for (i = 0; i < 4; i++) {
                if (get_edid_block(data, edid, 0, EDID_LENGTH))
                        goto out;
                if (drm_edid_block_valid(edid, 0, false,
                                         &connector->edid_corrupt))
                        break;
                if (i == 0 && drm_edid_is_zero(edid, EDID_LENGTH)) {
                        connector->null_edid_counter++;
                        goto carp;
                }
        }
        if (i == 4)
                goto carp;

        /* if there's no extensions, we're done */
        valid_extensions = edid[0x7e];
        if (valid_extensions == 0)
                return (struct edid *)edid;

        new = krealloc(edid, (valid_extensions + 1) * EDID_LENGTH, GFP_KERNEL);
        if (!new)
```

```
        goto out;
edid = new;

for (j = 1; j <= edid[0x7e]; j++) {
        u8 *block = edid + j * EDID_LENGTH;

        for (i = 0; i < 4; i++) {
                if (get_edid_block(data, block, j, EDID_LENGTH))
                        goto out;
                if (drm_edid_block_valid(block, j, false, NULL))
                        break;
        }

        if (i == 4)
                valid_extensions--;
}

if (valid_extensions != edid[0x7e]) {
        u8 *base;

        connector_bad_edid(connector, edid, edid[0x7e] + 1);

        edid[EDID_LENGTH-1] += edid[0x7e] - valid_extensions;
        edid[0x7e] = valid_extensions;

        new = kmalloc_array(valid_extensions + 1, EDID_LENGTH,
                            GFP_KERNEL);
        if (!new)
                goto out;

        base = new;
        for (i = 0; i <= edid[0x7e]; i++) {
                u8 *block = edid + i * EDID_LENGTH;

                if (!drm_edid_block_valid(block, i, false, NULL))
                        continue;

                memcpy(base, block, EDID_LENGTH);
                base += EDID_LENGTH;
        }

        kfree(edid);
        edid = new;
}
```

```
        return (struct edid *)edid;

carp:
        connector_bad_edid(connector, edid, 1);
out:
        kfree(edid);
        return NULL;
}
EXPORT_SYMBOL_GPL(drm_do_get_edid);
```

drivers/gpu/drm/drm_edid.c

```
/**
 * drm_do_probe_ddc_edid() - get EDID information via I2C
 * @data: I2C device adapter
 * @buf: EDID data buffer to be filled
 * @block: 128 byte EDID block to start fetching from
 * @len: EDID data buffer length to fetch
 *
 * Try to fetch EDID information by calling I2C driver functions.
 *
 * Return: 0 on success or -1 on failure.
 */
static int
drm_do_probe_ddc_edid(void *data, u8 *buf, unsigned int block, size_t len)
{
        struct i2c_adapter *adapter = data;
        unsigned char start = block * EDID_LENGTH;
        unsigned char segment = block >> 1;
        unsigned char xfers = segment ? 3 : 2;
        int ret, retries = 5;

        /*
         * The core I2C driver will automatically retry the transfer if the
         * adapter reports EAGAIN. However, we find that bit-banging transfers
         * are susceptible to errors under a heavily loaded machine and
         * generate spurious NAKs and timeouts. Retrying the transfer
         * of the individual block a few times seems to overcome this.
         */
        do {
                struct i2c_msg msgs[] = {
                        {
                                .addr   = DDC_SEGMENT_ADDR,
                                .flags  = 0,
                                .len    = 1,
                                .buf    = &segment,
```

```
                }, {
                        .addr   = DDC_ADDR,
                        .flags  = 0,
                        .len    = 1,
                        .buf    = &start,
                }, {
                        .addr   = DDC_ADDR,
                        .flags  = I2C_M_RD,
                        .len    = len,
                        .buf    = buf,
                }
        };

        /*
         * Avoid sending the segment addr to not upset non-compliant
         * DDC monitors.
         */
        ret = i2c_transfer(adapter, &msgs[3 - xfers], xfers);

        if (ret == -ENXIO) {
                DRM_DEBUG_KMS("drm: skipping non-existent adapter %s\n",
                              adapter->name);
                break;
        }
    } while (ret != xfers && --retries);

    return ret == xfers ? 0 : -1;
}
```

drm/amd/amdgpu/amggpu_i2c.c

```
static void amdgpu_i2c_post_xfer(struct i2c_adapter *i2c_adap)
{
    struct amdgpu_i2c_chan *i2c = i2c_get_adapdata(i2c_adap);
    struct amdgpu_device *adev = i2c->dev->dev_private;
    struct amdgpu_i2c_bus_rec *rec = &i2c->rec;
    uint32_t temp;

    /* unmask the gpio pins for software use */
    temp = RREG32(rec->mask_clk_reg) & ~rec->mask_clk_mask;
    WREG32(rec->mask_clk_reg, temp);
    temp = RREG32(rec->mask_clk_reg);

    temp = RREG32(rec->mask_data_reg) & ~rec->mask_data_mask;
    WREG32(rec->mask_data_reg, temp);
    temp = RREG32(rec->mask_data_reg);
```

```
        mutex_unlock(&i2c->mutex);
}
```

之前高英杰的分享中 VGA 会隔 10s 循环读一下 edid。

gpu/drm/drm_probe_helper.c

```
static void output_poll_execute(struct work_struct *work)
{

        drm_connector_list_iter_begin(dev, &conn_iter);
        drm_for_each_connector_iter(connector, &conn_iter) {

                repoll = true;

                connector->status = drm_helper_probe_detect(connector, NULL, false);
                if (old_status != connector->status) {
                        const char *old, *new;

                        if (connector->status == connector_status_unknown) {
                                connector->status = old_status;
                                continue;
                        }
                       changed = true;
                }
        }
        drm_connector_list_iter_end(&conn_iter);

        mutex_unlock(&dev->mode_config.mutex);

out:
        if (changed)
                drm_kms_helper_hotplug_event(dev);

        if (repoll)
                schedule_delayed_work(delayed_work, DRM_OUTPUT_POLL_PERIOD);
}
```

我对 edid 不太了解，现在怀疑是 VGA 的循环读 edid 和切换用户时读 edid 同时进行的情况下在 i2c 的层面有了干扰

第 7 章 用户与系统 xorg 进程对应方法

通过 pstree 可以查看系统的服务启动流程。系统的启动流程大体描述一下：
创建多个用户，有的登陆，有的不登陆。

```
uos@uos-PC:~$ loginctl list-sessions
SESSION  UID USER    SEAT  TTY
      1 1000 uos     seat0
     10 1004 test4   seat0
     12 1007 test7   seat0
     16 1006 test6   seat0
     19 1008 test8   seat0
     32 1000 uos           pts/2
      4 1002 test2   seat0
      7 1001 test1   seat0
     c7  116 lightdm seat0

9 sessions listed.

uos@uos-PC:~$ loginctl show-session 1
Id=1
User=1000
Name=uos
Timestamp=Tue 2023-10-24 10:44:15 CST
TimestampMonotonic=4856519
VTNr=1
Seat=seat0
Display=:0
Remote=no
Service=lightdm-autologin
Desktop=deepin
Scope=session-1.scope
Leader=1137
Audit=1
Type=x11
Class=user
Active=no
State=online
IdleHint=no
IdleSinceHint=0
IdleSinceHintMonotonic=0
LockedHint=no

uos@uos-PC:~$ ps -ef|grep xorg|grep -v grep
```

```
root      987   839  0 10月24 tty1    00:00:17 /usr/lib/xorg/Xorg -background none :0
root     3108   839  0 10月24 tty2    00:00:10 /usr/lib/xorg/Xorg -background none :1
root     5265   839  0 10月24 tty3    00:00:04 /usr/lib/xorg/Xorg -background none :2
root     6921   839  0 10月24 tty4    00:00:05 /usr/lib/xorg/Xorg -background none :3
root     8327   839  0 10月24 tty5    00:00:04 /usr/lib/xorg/Xorg -background none :4
root     9802   839  0 10月24 tty6    00:00:07 /usr/lib/xorg/Xorg -background none :5
root    25787   839  0 10月24 tty7    00:00:05 /usr/lib/xorg/Xorg -background none :6
root    27291   839  0 10月24 tty8    00:00:05 /usr/lib/xorg/Xorg -background none :7
```

上面的“:0”是 X Display Server 的名字。

第 8 章 使用 perf 来调试

关于 perf 的使用可以看另一篇博客：https://guolongji.xyz/post/use-perf-and-flame-graph/

目的是找两个已经登陆的用户的两个不同的 Xorg 进程，对这两个不同的进程用 perf 工具进行统计。这两个进程都是在运行当中的，如果 perf 能够同时对两个 xorg 的进程进行分析，那么就可以打到耗时的地方。

```
uos@uos-PC:~$ ps -ef|grep kwin|grep -v grep
uos       1331  1172  0 10月24 ?        00:00:00 /bin/sh /usr/bin/kwin_no_scale
uos       1387  1331  0 10月24 ?        00:00:18 kwin_x11 -platform dde-kwin-xcb:appFilePath=/usr/bin/kwin_no_scale
test2     4171  3977  0 10月24 ?        00:00:00 /bin/sh /usr/bin/kwin_no_scale
test2     4261  4171  0 10月24 ?        00:00:09 kwin_x11 -platform dde-kwin-xcb:appFilePath=/usr/bin/kwin_no_scale
test1     5701  5541  0 10月24 ?        00:00:00 /bin/sh /usr/bin/kwin_no_scale
test1     5809  5701  0 10月24 ?        00:00:01 kwin_x11 -platform dde-kwin-xcb:appFilePath=/usr/bin/kwin_no_scale
test4     7353  7196  0 10月24 ?        00:00:00 /bin/sh /usr/bin/kwin_no_scale
test4     7463  7353  0 10月24 ?        00:00:01 kwin_x11 -platform dde-kwin-xcb:appFilePath=/usr/bin/kwin_no_scale
test7     8726  8572  0 10月24 ?        00:00:00 /bin/sh /usr/bin/kwin_no_scale
test7     8822  8726  0 10月24 ?        00:00:01 kwin_x11 -platform dde-kwin-xcb:appFilePath=/usr/bin/kwin_no_scale
test6    24815 24625  0 10月24 ?        00:00:00 /bin/sh /usr/bin/kwin_no_scale
test6    24891 24815  0 10月24 ?        00:00:00 kwin_x11 -platform dde-kwin-xcb:appFilePath=/usr/bin/kwin_no_scale
test8    26225 26070  0 10月24 ?        00:00:00 /bin/sh /usr/bin/kwin_no_scale
test8    26316 26225  0 10月24 ?        00:00:01 kwin_x11 -platform dde-kwin-xcb:appFilePath=/usr/bin/kwin_no_scale
```

```
uos@uos-PC:~$ cat /proc/1387/environ |grep -a --color DISPLAY
```

观察两个变量分别是:0 和:6

那么对应的两个进程号就是 987 和 25787。

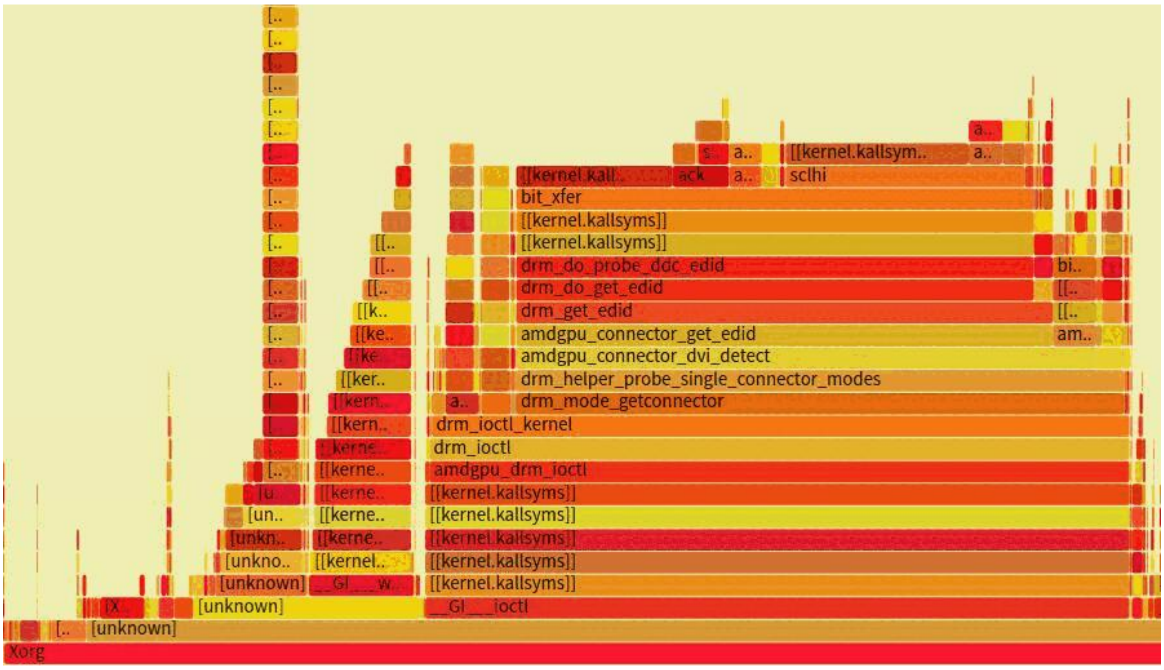
用 perf 来监视的话，分别在两个目录下运行：

perf record -p 987 -g – sleep 30

perf record -p 25787 -g – sleep 30

测试部步是先两个 ssh 在两个不同的目录运行两个进程的采样任务。手动在被测试的机器上切换 uos 和 test8 两个用户，现象是经过 6 到 7 次，没有黑屏，最后一次黑屏。经过 10s 之后采样结束。

生成的图如下：



xorg 的时钟占比为 94%，而 drm_get_edid 的时钟占比高达 32%。

显然被测试的 oland 显卡 drm_do_probe_ddc_edid 这个过程是有时候比较慢的。一直切换会由“经常不黑屏”的状态变为经常黑屏的状态。所以猜测是存在什么循环。

第 9 章 为什么 xorg 的数量比登陆的用户数多一个？

因为 lightdm-deepin-greeter 也会创建一个 xorg 用于过渡。

```
uos@uos-PC:~$ ps -ef|grep greeter|grep -v grep
lightdm   27361 27344   0 10月24 ?       00:00:00 /bin/bash /usr/bin/deepin-greeter
lightdm  27376 27361  0 10月24 ?     00:00:00 /bin/bash /usr/share/dde-session-shell/greeters.d/x/lightdm-deepin-greeter
lightdm   27377 27376   0 10月24 ?       00:00:02 /usr/lib/deepin-daemon/greeter-display-daemon
lightdm   27379 27376   0 10月24 ?       00:00:00 /bin/bash /usr/share/dde-session-shell/greeters.d/launch-binary
lightdm   27407 27379   0 10月24 ?       00:00:33 /usr/bin/lightdm-deepin-greeter
```

第 10 章 能不能写一个切换用户的脚本呢？

```
while true; do qdbus --system org.freedesktop.login1 /org/freedesktop/login1/seat/seat0 \
org.freedesktop.login1.Seat.SwitchToNext; sleep 2; qdbus --system org.freedesktop.login1 \
/org/freedesktop/login1/seat/seat0 org.freedesktop.login1.Seat.SwitchToPrevious; sleep 2; done
```

用 dbus 信号去切 tty 是没有问题的。与窗管的同事沟通了解到。dde-dock 可能是会刷新 edid 的状态，而用上面的命令不会刷新 edid 的状态，那么也就不会出现黑屏的问题了。那现在这个问题的的原因就基本上清晰了。

之前廖元用的脚本是这个：

```
#!/bin/bash
n=0
while (($n<100))
do
        sh -c "gdbus call --system --dest org.freedesktop.login1 --object-path /org/freedesktop/login1 \
--method org.freedesktop.login1.Manager.ActivateSession 2"
        echo $n
        n=$((n+1))
        sleep 2
        sh -c "gdbus call --system --dest org.freedesktop.login1 --object-path /org/freedesktop/login1 \
--method org.freedesktop.login1.Manager.ActivateSession c2"
        sleep 2
done
```

第 11 章 其它的相关 bug

机缘巧合之下，在 4.19 内核里发现了 radeon 驱动一个很神奇的问题，插拔 hdmi 线时候，先拔出一半等 10s 左右再全部拔出。这时候，在 sys 下读到的 hdmi 连接状态还是 connected。这个感觉还是很神奇的。切到 amdgpu 之后，也有这个问题。

https://blog.51cto.com/u_15155099/2767298

HDMI 拔出正常逻辑应该是：HPD 探测到电压变化触发中断，接下来 DDC 读取显示器 EDID 返回失败，最终到 dvi_detect 函数中，通过 DDC 返回的失败，设置显示器连接状态为 disconnected。

在这个问题中，HDMI 拔出，HPD 探测到电压变化中断触发，DDC 读取显示器 EDID 返回成功，detect 函数设置显示器连接状态是 connected。那真正出错的位置是 DDC 不应该读取到 EDID。

oland 显卡读取 edid 的逻辑有大问题。

第 12 章 什么是 EDID？

edid 核心的内容都在这里了：

https://www.graniteriverlabs.com.cn/technical-blog/edid-overview/

https://en.wikipedia.org/wiki/Extended_Display_Identification_Data

https://www.wpgdadatong.com.cn/blog/detail/72670

EDID revision history	
EDID v1.0, August 1994	Defined original 128-byte data format (deprecated)
EDID v1.1, April 1996	Added definition to existing data fields (deprecated)
EDID v1.2, November 1997	Added definition to existing data fields (deprecated)
EDID v1.3, September 1999	New baseline for EDID data structure
E-EDID v1.3, February 2000	Allows addition data stored as EDID extensions, and is used in HDMI
E-EDID v1.4, September 2006	Added support for consumer electronic product
DisplayID v2.0, November 2017	Introduced variable-length structures of up to 256 bytes

而目前我们所采用的是 EDID1.3 版本，EDID1.0、EDID1.1、EDID1.2 均已在 2001 年 1 月 1 日停止使用。

```
cat /sys/class/drm/card0-VGA-1/edid |hexdump -C
# 或者：cat /sys/class/drm/card0-HDMI-A-1/edid | hexdump -C
```

view sonic 的显示器举例：

- 下面是 VGA 的 edid：

```
00000000  00 ff ff ff ff ff ff 00  5a 63 22 3a 01 01 01 01  |........Zc":....|
00000010  29 1e 01 03 08 35 1d 78  2e e0 f5 a5 55 52 a0 27  |)....5.x....UR.'|
00000020  0c 50 54 bf ef 80 b3 00  a9 40 a9 c0 95 00 90 40  |.PT......@.....@|
00000030  81 80 81 40 81 c0 02 3a  80 18 71 38 2d 40 58 2c  |...@...:..q8-@X,|
00000040  45 00 0f 29 21 00 00 1e  00 00 00 ff 00 57 36 38  |E..)!........W68|
00000050  32 30 34 31 32 30 30 39  33 0a 00 00 00 fd 00 32  |204120093......2|
00000060  4b 18 52 11 00 0a 20 20  20 20 20 20 00 00 00 fc  |K.R...      ....|
00000070  00 56 41 32 34 33 31 2d  48 2d 32 0a 20 20 00 74  |.VA2431-H-2.  .t|
00000080
```

- 下面是 HDMI 的 edid：

```
00000000  00 ff ff ff ff ff ff 00  5a 63 3a c6 01 01 01 01  |........Zc:.....|
00000010  1e 1e 01 03 80 35 1d 78  2e b4 85 a3 56 50 a0 26  |.....5.x....VP.&|
00000020  0f 50 54 bf ef 80 b3 00  a9 40 a9 c0 95 00 90 40  |.PT......@.....@|
00000030  81 80 81 40 81 c0 02 3a  80 18 71 38 2d 40 58 2c  |...@...:..q8-@X,|
00000040  45 00 0f 28 21 00 00 1e  00 00 00 fd 00 32 4b 18  |E..(!........2K.|
00000050  52 11 00 0a 20 20 20 20  20 20 00 00 00 fc 00 56  |R...      .....V|
00000060  41 32 34 36 32 2d 48 0a  20 20 20 20 00 00 00 ff  |A2462-H.    ....|
00000070  00 57 42 31 32 30 33 30  30 35 33 34 34 0a 00 17  |.WB1203005344...|
00000080
```

重点看：

1.12h 和 13h，分别是 01 和 03。这就表明是 edid 1.3 版本，且不包含括展块。

2.23h 至 25h 为 Established timing，早期的分辨率信息。上面的是 bf ef 80，即：

1011 1111 1110 1111 1000 0000

Address	3 Bytes	Bit #	Description	Source
23h	1		Established Timing I	
		7	720 x 400 @ 70Hz	IBM, VGA
		6	720 x 400 @ 88Hz	IBM, XGA2
		5	640 x 480 @ 60Hz	IBM, VGA
		4	640 x 480 @ 67Hz	Apple, Mac II
		3	640 x 480 @ 72Hz	VESA
		2	640 x 480 @ 75Hz	VESA
		1	800 x 600 @ 56Hz	VESA
		0	800 x 600 @ 60Hz	VESA
24h	1		Established Timing II	
		7	800 x 600 @ 72Hz	VESA
		6	800 x 600 @ 75Hz	VESA
		5	832 x 624 @ 75Hz	Apple, Mac II
		4	1024 x 768 @ 87Hz(I)	IBM - Interlaced
		3	1024 x 768 @ 60Hz	VESA
		2	1024 x 768 @ 70Hz	VESA
		1	1024 x 768 @ 75Hz	VESA
		0	1280 x 1024 @ 75Hz	VESA
25h	1		Manufacturer's Timings	
		7	1152 x 870 @ 75Hz	Apple, Mac II
		6-0	Reserved for Manufacturer Specified Timings	

通过 xrandr 看能对应上显示器的信息：

```
uos@guolongji:~/Desktop$ xrandr
Screen 0: minimum 320 x 200, current 1920 x 1080, maximum 16384 x 16384
HDMI-0 disconnected (normal left inverted right x axis y axis)
DVI-0 disconnected (normal left inverted right x axis y axis)
VGA-0 connected primary 1920x1080+0+0 (normal left inverted right x axis y axis) 527mm x 297mm
   1920x1080     60.00*+
   1600x1200     60.00     1680x1050     59.95
   1400x1050     59.98     1600x900      60.00
   1280x1024     75.02     60.02     1440x900      59.89
   1280x960      60.00     1152x864      75.00
   1280x720      60.00     1024x768      75.03     70.07     60.00
   832x624       74.55
   800x600       72.19     75.00     60.32     56.25
   640x480       75.00     72.81     66.67     59.94
   720x400       70.08
```

3. 从 26h 到 35h，16 个 bit，每两个 bit 表示一个标准分辨率，即：b3 00 a9 40 a9 c0 95 00 90 40 81 80 81 40 81 c0

10110011 00000000

10100110 01000000

10100110 11010000

01100101 00000000

01100000 01000000

10000001 10000000

10000001 01000000

10000001 11010000

拿 10110011 00000000 来举例：10110011 即 b3，179，加上 31 即 210，乘以 8，即 1680，这个是 1680 的那个分辨率的信息。

4. 那么 1920 那个分辨率的信息是怎么拿到的呢？

02 3a 80 18 71 38 2d 40 58 2c 45 00 0f 28 21 00 00 1e

从 established timing 到 standard timing 到 Detailed timing descriptor，分辨率的解析就很麻烦。

这个的解析比较麻烦，可以参考：

https://winddoing.github.io/post/47714.html

有一个站可以解析 edid：http://www.edidreader.com/

开源的解析工具很多，https://github.com/dgallegos/edidreader ，供参考。

- 是不是 e-edid

edid 可能是 128 个 bit，也可能是 256 个 bit。分为 block0 和 block1 两部分。block1 叫 Extension block。

block1 可以没有。Block0 已对显示器的功能进行基本的描述，像是产品信息和支持的分辨率等等，但如果显示器要额外支持一些 HDMI 进阶功能：如色深、音讯格式、3D，此时就需要使用 Extension block 来描述。Extension block 的格式总共有下列几种，但在 HDMI 规范中必须至少使用一个 CEA-EXT。

倒数第二个字节是 Extension Block(s)：0，就不是 e-edid，非 0 就是。

第 13 章 FQA

- 为什么系统下和芯片中都需要包含固件代码？(显卡 vbios 和系统下的 bin)

在很多硬件设备中，特别是芯片、固件和驱动程序之间的关系非常紧密。固件代码通常被设计用于与硬件交互，并为操作系统或其他软件提供硬件相关的功能和操作。因此，在某些情况下，固件代码需要同时存在于硬件设备中和操作系统文件系统中。

具体来说，设备固件通常存储在硬件设备的非易失性存储器中，例如闪存、EEPROM 或 ROM 芯片中。这些固件代码包含了设备的基本操作和配置信息，以及一些初始化程序和驱动程序。

然而，在某些情况下，固件代码需要在操作系统中使用。例如，在 Linux 系统中，设备驱动程序通常需要使用固件代码才能正确地初始化和配置设备，从而使设备在操作系统中正常工作。因此，固件代码必须以某种方式从设备中传输到操作系统中。

为此，Linux 等操作系统提供了一种机制，允许将固件代码复制到操作系统文件系统中的某个位置，以便设备驱动程序在需要时可以加载并使用它。这就是为什么在系统中也需要包含固件代码。操作系统中的这些固件文件通常放置在/lib/firmware 目录中。

总之，固件代码需要同时存在于硬件设备中和操作系统文件系统中，因为它们在共同协作以使设备得以正常工作。