# RickLiu实习期学习工作总结

## 工作流程

### 1、Gerrit仓库

* Gerrit网页操作，查看、处理代码和发送reviewer等。
* 学习复杂的git命令：cherry-pick、git commit –amend、***git rebase***，已经能熟练的中处理和提交git项目。

### 2、BUG维修

* 熟悉Redmine和Gerrit的BUG上报和维修流程，根据不同结论添加报告。
* 简单的BUG、LOG搜索和定位，知道哪部分的代码和哪些问题是我们处理。

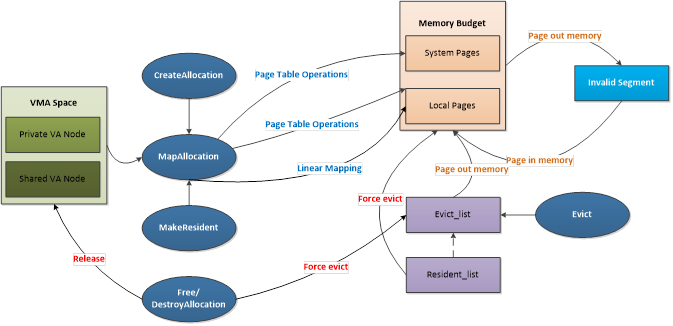
## Linux图形栈

* Xorg-xserver： > X的驱动包括软件部分DIX和硬件部分DDX，在X11/module/driver文件夹下有硬件相关driver，2D driver驱动模块为arise\_drv.so，3D driver是arise\_dri。  
  > Xorg在初始化会probe显示device驱动，如果没找到就会依次找：fbdev(默认framebuffer)、vesa、KMS(有glamor加速)，在xorg.conf中有X的初始配置，-10表示显卡相关。  
  加载显示驱动后，Xorg加载输入设备(xinput)。  
  > DDX代码在xorg源码的xfree86文件夹下，各厂商DDX驱动的实现格式相似。Xorg的log在var中。  
  > 窗口的专用屏幕外缓冲区（后缓冲区、假前缓冲区、深度缓冲区、模板缓冲区等）的分配由Xserver gbm本身完成，合成窗口管理器将其用作合成作为最终的屏幕后缓冲。  
  Xorg 2D driver只是xserver的一个模块，主要实现2D加速画点和线。我们的驱动里面实现了一套私有的rxa加速函数，以及通过glamor3D加速。
* drm：为与dri联合使用，上层显示服务提供接口（/dri/card0,/dri/renderD128），在用户态对应LibDRM库，底层就是显卡driver代码，通过一系列 ioctl(或者ioctl包装的辅助函数)实现用户态程序到内核驱动最后操作硬件，使用户态程序可以访问显存、DMA和OB。drm设备继承了pcie设备。drm中的gem模块实现buffer的管理，更底层的实现我们公司自己实现的而不是ttm。
* dri：dri驱动由硬件厂商实现，驱动程序可以直接访问硬件，Xorg可以通过OGL走dri实现硬件加速，利用3D驱动绘制2D图像（glx）。通过gdb可以追踪调用路径。
* LibGL库：是实现了OpenGL的编程接口，用户使用LibGL库提供的开发接口开发OpenGL应用程序，对用户屏蔽了底层的实现细节，其实就是向图形驱动下发OpenGL的调用。
* KMS：完成显卡配置，能直接操作底层硬件实现显示，不依赖Xserver。但是通过KMS写显示代码更复杂，需要定义所有组件的行为，找到所有可以用路径并选择一条， 也需要代码实现buffer flip。
* bit-blit：块传输图，将图像从内存中的一个区域快速地复制到另一个区域生成一个副本，gpu可通过它获得数据。
* fence：标记一个GPU pipeline事件，用于cpu与gpu的同步。当cpu写入一个fence，所有关联的ip核上的ringbuffer都会flush cache，防止数据被覆盖。 多个渲染命令作为一个packet提交到ringbuffer，提交的IT\_BODY部分既可以是渲染命令也可以是命令的地址。提交一次cpu渲染命令后，绑定fence回调函数发送到gpu，并 查询fence状态。
* DMAbuffer：dmabuf是buffer与file的结合，即dma-buf既是块物理buffer，又是个file。应用程序可以通过file属性直接写数据，GPU可以在地址空间直接读。 cpu访问dmabuffer通过kmap和vmap，外设访问也有特定的函数，通过mmap映射给应用程序。
* MMIO：将外设内存或者IO映射到cpu可见的物理地址，让cpu访问外部内存和io跟访问内存一样。
* PCImemory：snoop（不用管） unsnoop（手动flush）
* IOMMU：外设的虚拟地址转换，外设访问内存跟cpu一样访问虚拟地址。
* sysfs：对sysfs相关的实现和函数基本了解，能定义并使用文件节点，熟悉linux这部分文档的规范。
* power-gating/dvfs：amd上的通过level来设置，大范围给了performance、battery、balance调节clock的上限，可以通过手动调节电压和电源的level。 004上主要由pg\_cg-gating和dvfs两部分,pgcg auto模式通过硬件idle计数调节，manual模式通过driver的task密度调节。dvfs通过checksize和threshold 两个参数调节，在一个checksize中工作状态大于threshold的上限就加，小于下限就减。
* Mesa：开源图形库，利用软件实现了图形api，硬件的usermod驱动和mesa结合使用。Gallium3D： Gallium3D是Mesa的一部分，是一个开发3D图形驱动的框架。 glvnd是nvidia的开源项目，加glvnd层将openglapi分配给具体的实现。

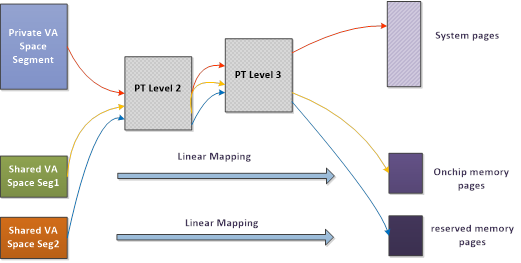
## GLENFLY

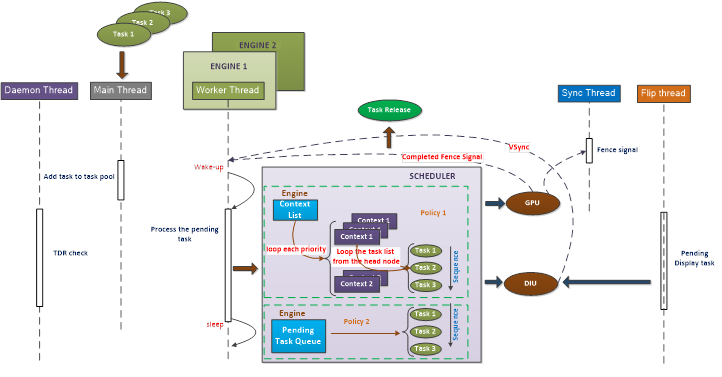
* 驱动代码架构：xf86-video-zx是2Ddriver，CIL2是中间层结构，E3K中是硬件层代码，还有linux相关代码等。
* Linux文件夹下的core层代码是核心代码，e3k是chip层代码，对内核层提供了函数接口。vidmm是内存分配和管理的代码，vidsch是任务调度和事件处理相关的代码， 还有CM上下文相关代码，中断处理和display相关代码。
* CM：上下文管理，其实就是一个大结构体绑定了显卡资源。
* **VidMM**：

1. 概述：负责管理allocation的生命周期、将物理地址映射到虚拟地址等操作。



1. 地址空间管理：将显存分为私有和共享空间，一个独立的部分叫做zone，zone有特定的分配格式。初始化zone后分配bigmap并填page table， 根页表PTE存在TTBR寄存器，进行MMIO和DMA操作时就会查表。



1. reserved\_vma：记录kmd资源信息，例如dma fence page等 。
2. segment内存用于kernel mode资源，包括一些全局资源，如fence buffer、CM buffer，并包括一些内部使用资源，比如页表缓冲区。
3. VM操作：在初始化阶段分配的zone上allocate一段内存(node list)->分配物理页，填3级页表。（文档中有虚拟内存空间计算表）->vma-space更新page directory让gpu能直接detect页表
4. GPU allocation lifeCycle：是否共享->是否evict（页置换）->引用计数为0 销毁->检查shared引用->销毁对应的物理内存和页表。
5. 内存数据结构：pa rbt - va rbt ，heap - buddy（合并） 

**·VidSch**：

1. 概述：决定哪个任务可以被GPU执行，何时执行，比如渲染、同步、显示。
2. TDR：探测硬件状态，别让硬件停了。
3. 线程：主线程（添加工作队列），工作线程（每个engine都有），守护线程，同步线程（fence，flip）
4. scheduler管理：一个adapter一个，各自engine都有（管理ringbuffer）。scheduler需要内存所以依赖vidmm部分。
5. vm id？:虚拟内存编号，各种scheduler entity分配内存时都有vmid。
6. task：一个任务包括多个命令，可由多个初始task组成一个package。常见的task包括render、paging（页置换和 fill）、sync（fence）、wait、signal、display等等，task结构体中专门指出dma的描述和类型。 生命周期由创建、等待、执行、释放，其中包括fence和fence回调。
7. fence：wait fence的task只要等到比它大的fence就会wakeup。server采用异步的方式，由kernel mode唤醒fence等待队列。fence也分cpu类型和gpu类型。
8. display fence：软件fence，在flip时kernel mode发送。
9. 调度优先级：优先级调度和先进先出，优先级只分两级，低和高。

* **中断**：中断分为schedule（fence vsync busy）、power（pg cg dvfs）、display（热拔插 vsync）三大类，中断程序包括读中断和中断处理两部分
* 功能子系统：调试系统，例如内存分配、任务生命周期、内存分页等

## OpenGL

* OpenGL概述：opengl是一组应用层图形api，只是接口规范并不是具体实现，具体实现可走软件的mesa或vendor硬件实现。渲染管线包括：顶点绘制、图元 装配、光栅化、片段着色等基本流程， 对应着***顶点着色器、细分着色器、几何着色器、片段着色器***。
* OpenGL编程步骤：利用EGL或GLFW等中间层窗口框架创建渲染窗口，OGL在其创建的window或者surface上绘制；定义并编译GLSL的pipeline渲染代码， 最少包括顶点着色器和片段着色器；然后在上下文中bind VBO、EBO等数据类型，VBO就是在显存中分配一个内存块，GPU只用从这里取数据不用一直改所以很快。 新版本的OGL可以将VBO bind到VAO上方便定义和切换，老版本需要在draw循环中bind VBO；然后获取纹理数据到纹理单元（1-16个），其中包括格式转换和 格式定义等；最后就是在循环中draw，***清屏->激活VAO、 着色器->加载纹理->绘制->响应回调事件***。
* OpenGL坐标系统：屏幕零点坐标在中间，3D坐标系为右手坐标，z轴指向屏幕外，可视范围都在-1，1内。3D可视空间变换包括：模型矩阵、观察矩阵、 投影矩阵，模型矩阵的作用就是把物体放到合适的接近真实的大小和位置，观察矩阵就是看的方向和距离，投影（视角）矩阵就是截取视线角度和范围。 对于看不到的物体，可以开启深度测试，OpenGL会基于深度缓冲把绘制点与当前深度缓冲对比，丢弃小于它的点。
* 纹理：纹理就是细节图像，在VBO中可指定纹理坐标，纹理坐标和OpenGL的坐标不同，零点在左下角。可以通过设置一些纹理相关的宏实现不一样的效果， 比如环绕复制方式，采样方式等。GLSL中通过uniform类型传递纹理。
* 模板和帧缓冲：模板缓冲就是当前时刻存储一个固定矩阵，绘制时与固定矩阵加设置条件，这里设置条件可以任意，通常是绘制的位置模板值为1就不绘制。 绘制边框轮廓常用模板缓冲。帧缓冲就是数据内存，GLFW创建了默认帧缓冲，包括颜色、模板、深度的。可以通过glGenFramebuffers创建自定义帧缓冲， 然后给帧缓冲attach一个具体缓冲，最后分配renderbuffer数据(深度、模板)和texture数据内存。主循环要使用帧缓冲要bind加上具体数据，并和默认帧缓冲切换。
* 光照：光源叠加就是两个RGB相乘，对于被照物体就是在片段着色器上输出乘个颜色向量。冯氏光照模型包括环境光、（漫）反射光、镜面光，环境光就是光源 和物体即使不在也有一些光亮，漫反射就是光源对物体的影响，镜面光就是物体表面的影响。计算光照时距离不重要，方向重要，通过法线夹角计算漫反射， 镜面光由物体材质决定。光源属性包括点光源、平行光、距离衰减，多个光源、属性叠加也是相乘。
* 贴图：贴图就是纹理，传入的方式和纹理一样。光照贴图将采样值乘上光照分量有更真实的效果。
* 细分着色器：就是切点和线，增加点和线，可以自己定义分割方法，新增的点可以输入到几何着色器。
* 几何着色器：对传入的顶点进行连线，决定几何形状。学习网站实现了法向量可视化的example，就是在绘制每个三角平面时加一个法向量线。
* **其他**： 3D model对象：通过一些组件网格组成可直接加载的大模型，每一个组件mesh都要实现draw，相当于一个一个组件画
* blending：混合颜色，实现不透明的效果，可通过宏开启。
* 面剔除：有一个宏控制，可剔除正向和反向面 。
* 立方体贴图：将6张图放在空间中，这一部分包括折射、反射的计算函数
* 高级数据：VBO中的数据改成自定义拼接格式 。
* GLSL内建变量：内建变量就是提供了一些变量实现特殊的功能，比如点放大、自定义深度值、面剔除等。**接口块**就是一个结构体，uniform也有块结构。
* 实例化：用于复制对象，通过模型矩阵在世界空间生成大量实体。
* gamma矫正：用非线性曲线表示色阶 。
* 抗锯齿：多重采样，一个光栅格子多个点
* pixmap：基于ASCII编码的图像格式，其实也就是一张图片
* 阴影贴图：光线向量计算出阴影范围，深度贴图：视线到物体的距离，法线贴图：根据颜色值改变法线 。
* PBR：基于物理定律的渲染，根据一些物理公式 。
* 调试错误查询glGetError，GLSL可以用颜色值调试输出。

## 显卡知识

* PCIE：是计算机地址总线的一种，pci的延伸，有更快的速率和带宽,pcie插槽通道有1、4、8、16。PCIE的传输协议跟网络包有点像，也有帧头帧尾、 插帧去帧的过程，PCIE设备开发针对交换层的信息包TLP。显卡属于pcie设备隶属于总线空间。
* PCIE地址空间：PCIE设备有四类内存：configure space、io space、memory space、message。在上点bios阶段会读取configure space的信息， 包括显卡厂商信号，硬件版本等信息，中间部分映射物理地址的叫bar需要填写，通过读取bar的位长确定空间大小（有一段值是写死的，判断写死的最高位然后确定空间大小）， PCIE在memory低地址段有一段映射空间，映射DRAM（MMIO）到这里,这个空间叫IO地址空间,对于X86来说，IO采用独立编址，arm中的IO使用统一编址。为了兼容老版本的pci， memory空间中有相同的configure space。这里实现了cpu放内存或者io能访问到pcie域。
* pci匹配：硬件信息在sys/bus/pci/devices下的uevent，总线驱动程序执行device\_match\_driver扫描 /lib/modules/uname -r/modules.alias，匹配MODAILAS变量中的pci信息。 pci维护了所有厂商的id表，系统需要更新这个表。
* 独显内存：MMIO的内存大小有时远小于独显内存，这是因为task中一次不需要这么大的dmabuffer，驱动不断的做新的dmabuffer并告诉gpu使用， 然后不断开新的buffer，只用转换显存其实地址就行。 core不直接访问显存，通过L2 cache和L1 cache访问。
* cpu访问gpu：准备一个命令字，通过I/O端口或者MMIO映射的寄存器发送；利用MMIO做DMA把内存里的东西送到显存里；如果有返回命令，这里可能要等待DMA中断完成之类的动作； cpu通过总线地址映射访问gpu内存。
* 显卡硬件：显卡不像CPU一样处理很多条件判断或冒险，流水线要保证高并发才能处理大量数据。显卡核心SIMD相当于处理单元，里面包含很多stream processer（core） 也就是线程资源，下面包含渲染管线（就是OGL中的管线）。显卡还包括电源模块、3D渲染模块、VCP（编解码），VPP（加速），内存，display等。
* display：framebuffer是硬件无关的抽象，plane（主平面、视频overlay、鼠标、背景等下层平面），crtc，显示器mode，vblank是留下一截空白部分 用于同步，防止画面撕裂。
* Video：video driver，vaapi，vdpau
* 半导体：IP核，Fabless模式，IDM，IC流程，SOC（设计 制造 验证）。
* 串口调试：硬件需要串口转换线和usb转换线，启动的boot更改参数
* 内核调试：在内核模块中测试内核函数，printk打印内核消息，kgdb调试内核源码，公司驱动zx\_info打印消息。
* 打log和看log：系统简易log看dmesg、查看驱动相关看kernel/debug、系统起来之前异常看串口log、Xorg看var/log
* tool工具：gdb、debug-server
* linux查看驱动信息的**命令**：
* link: ldd、cat /proc/\**/maps* 、lsof -ef PID 、ldconfig (system load) 、 nm -D 、 objdump/readelf
* Xorg: cat /var/log/Xorg.0.log 、 cat ./.local/share/xorg/Xorg.0.log 、 /usr/local/lib/xorg 、 cat /usr/share/X11/xorg.conf.d/
* Memory: cat /proc/iomem 、 dpkg -l 、 sudo apt show