High security, high performance,
low code size kernel

Lynix Microkernel White Paper

Prepared by
LynixCommunity

[WU PENG]
[wup95453@gmail.com]

[29 Dec 2024]

Introduction

Lynix 是 Layering Unix 的简称。我将对这两个词汇做如下解释。Layering 是 Lynix 的风格或是说基调，这与以往内核的 Modularity 基调不同。也就是说，Lynix 的成长路线不会是模块或是组件的堆叠，造就类 Linux 这种庞然大物。与之相反，Lynix 更倾向于以最基础的可用的模块出发，这些基础模块可以保证该内核的最小运行，开发者继承和派生这些基础模块来达到丰富该内核功能的目的。更进一步说，基础模块是第一层内核，派生模块是第二层，甚至是更多层，这有点类似于洋葱的结构。每一层内核都具备完整的可用的功能，区别在于代码规模和功能差异，这种结构对基础模块的接口定义要求很高。Unix 是 Lynix 的发展趋势，直白的说，就是产品生态的建设。我认为“内核+应用”才是一个可用的产品。按照以往的经验，在无厂商或是社区支持的前提下，去满足以往用户的使用习惯是这个产品能存活下的关键。不过，一味的支持用户接口定义的功能可能会造成内核的“结构变形”。总结下来，可以套用下面的公式来描述 Lynix：

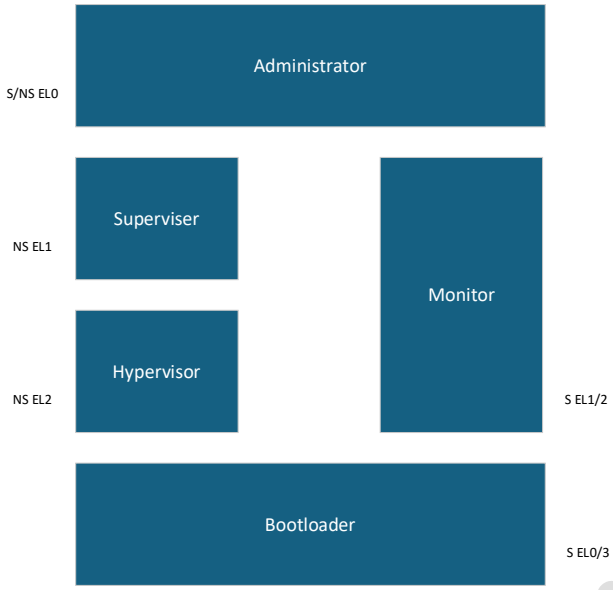
$$Lynix^{1} = {}_{base++}I^{2} + {}_{subset++}I^{'}$$

Understanding the Lynix

目前市面上的内核种类和数目太多了，但真正广泛使用的却很少，单是在嵌入式领域，可以列举的有：FreeRTOS/uCos(CP)，Zephyr/RT-thread(IOT)，Linux+RT(Algorithm)，TEE(Security)等。这些构型只是为了满足某个领域的特定需求且内核代码差异性过大，开发维护成本很高。以往开发者希望以微内核取代宏内核的方式来减少内核级的 SLOC，增加内核的健壮性；但是从系统的角度看，并未减少系统级 SLOC。Lynix 就是要接管上述这些内核所承载的特定需求，进而从整个系统级上看共用同一份代码，而不是同一个“名字”。由于 Lynix 是分层结构，对派生组件的增添和删除，即是对内核按照特定需求的扩展和裁剪，这将大大减少系统内不同构型造成的冗余代码的规模。举一个例子，同样是调度组件，不同内核的实现机制大同小异，只是不同贡献者的重复提交而已，但 Lynix 只需使用一份调度组件再搭配不同调度算法。
根据上述的描述，以 ARM 平台为例子，Lynix 期望满足/承担以下场景/角色：

- Monitor: Security World
- Hypervisor: Virtual Machine
- Superviser: Application
- Administrator: Resource Management
- Bootloader: Boot Process

[1] base 是基础内核，I^{2} 是 interface 和 implementation。$I^{'}$是某种标准生态接口，例如 POSIX 标准。base++是从基础内核到扩展内核的过程， subset++是从子集到全集适配生态接口的过程。这两个过程是同时进行的。

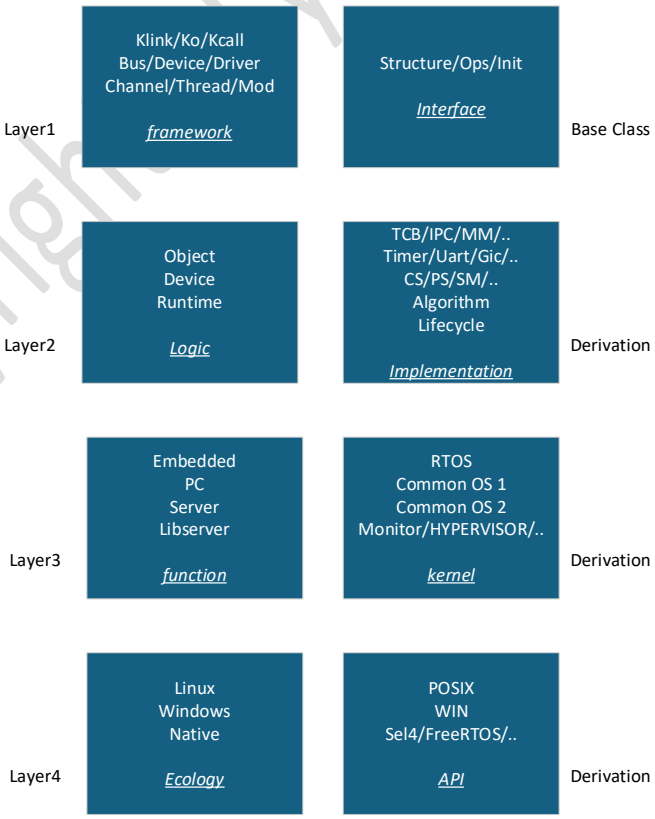


The Strategy of Lynix

Lynix 在设计上应遵循如下策略：

- Layer1: 通用框架作为根基，按照 OOP 编程的思路，Layer1 被视为（虚）基类；
- Layer2: 通用逻辑作为单元组件，也可以说是 Layer1 的派生；
- Layer3: 特定场景功能作为内核，也可以说是 Layer2 的派生；
- Layer4: 适用生态作为内核产品，也可以说是 Layer3 的派生。

如下图所示，



在 Layer1，定义三种基类框架作为接口，包括内核静态对象相关基类(klink/ko/kcall)，物理设备相关基类(bus/device/driver)和运行时对象相关基类(channel/thread/mod)。基类包含数据结构的定义，操作以及初始化相关内容。

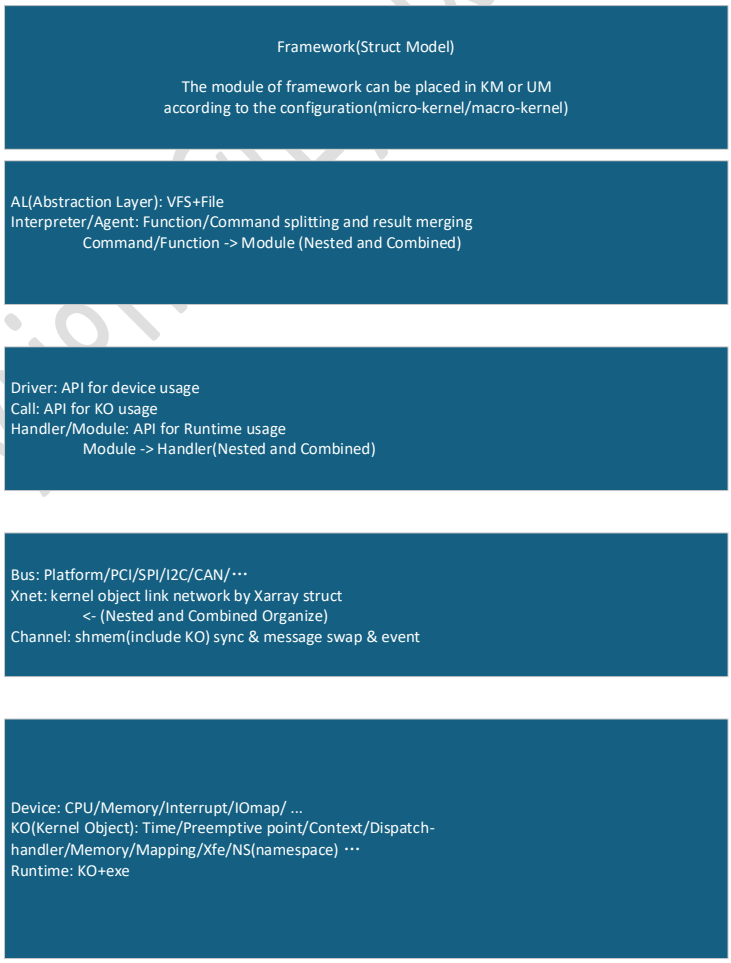
在 Layer2，定义基类的实现，即最小内核可用的逻辑。例如，对于内核静态对象，可以有 TCB/IPC/MM 等；对于物理设备，可以有 Timer/Uart/Gic 等；对于运行时对象，可以有 CS/PS/SM 等。不过，需要澄清的是，派生类的实现以实际情况而定：

- 三种派生类的算法或是说操作是不同的；
- 同一种派生类的不同对象的算法是不同的；
- 同一种派生类的同种对象的算法也是不同的。

在 Layer3，根据实际的业务场景，定义特定的功能。这些功能驱动 Layer2 的内核以不同的方向进行扩展。例如，在嵌入式场景中，需要将 Layer2 扩展成 RTOS，满足高实时性和确定性的要求；在 PC 或是 Server 场景中，需要将 Layer2 扩展成 Common OS，满足高吞吐性和低延迟的要求；在服务库场景中，需要将 Layer2 扩展成一个 Monitor 去处理高机密性任务的服务或是一个 Hypervisor 去管理高隔离性 VM 的服务。

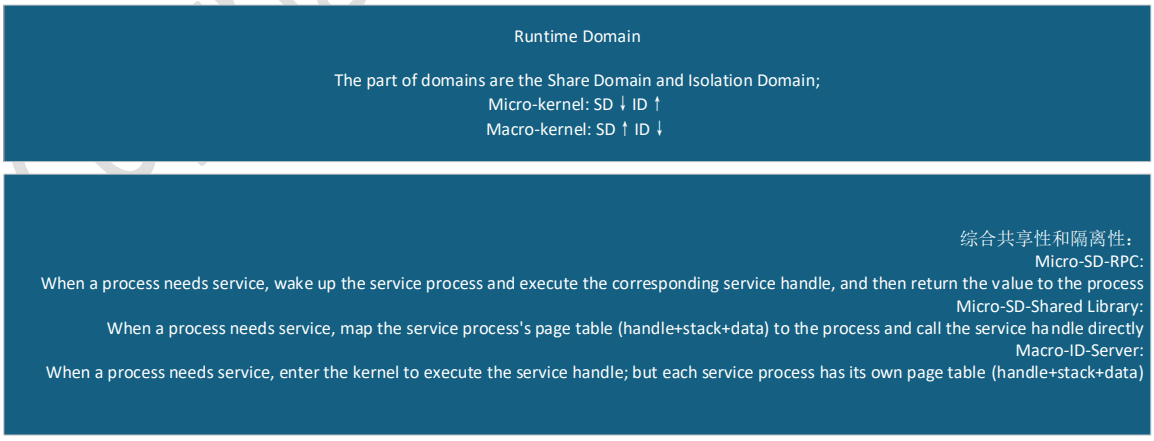
在 Layer4，根据现有成熟方案的不同生态，需要对 Layer3 更进一步的适配；例如，在 Linux 生态内，需要 Layer3 去适配 POSIX 标准的接口来支持原有的 Linux 应用。

更详细的分层结构如下图所示：

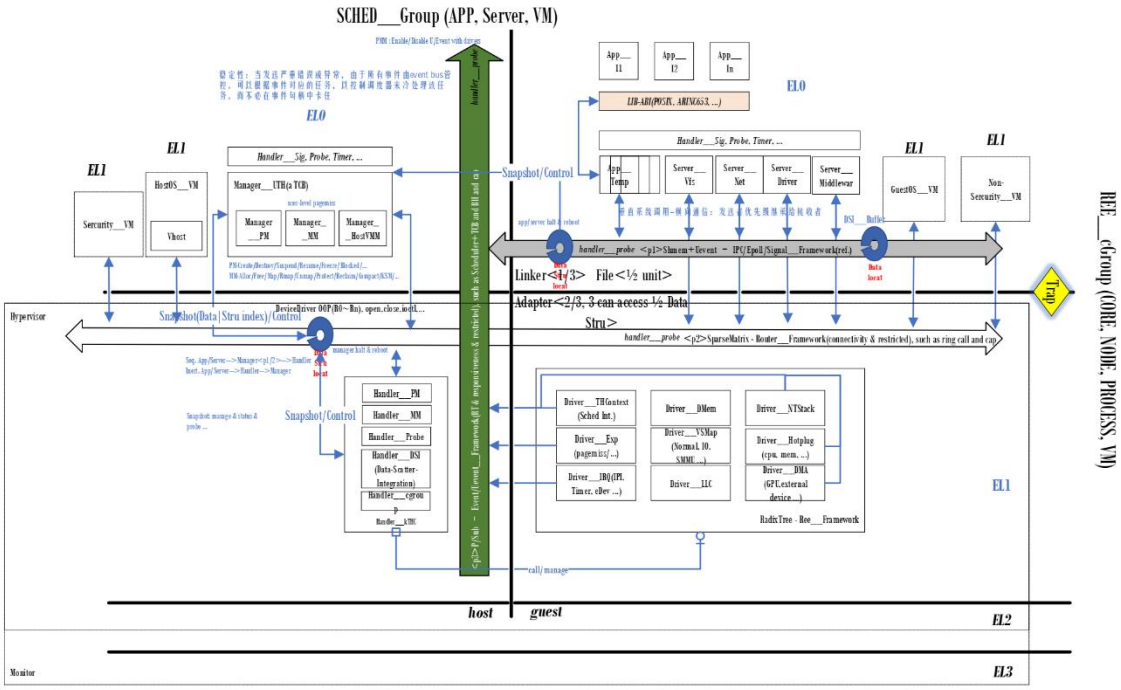


Layer1: framework

Layer2: logic



Layer3: function, including micro-kernel OR macro-kernel



Layer3/4: function&eco

Preparing the Lynix

Lynix 设计原则：
以框架设计为主，提高良好的接口和扩展性，框架数目在确定后不可增添，只可扩展。

Lynix 优先事项：

- 定义一个全态的驱动框架，支持某些板型的物理设备，驱动框架尽可能复用 Linux；
- 定义一个基本的数据结构库和算法库，支持驱动框架的使用；
- 定义一个基本的内核静态对象和运行时对象的框架，满足基本的内存管理和锁实现，支持驱动框架的使用。

Lynix 指导方针：
依照 TDD 的原则，确保每一次发布都是一个可用产品，以支持后续的增量开发。代码审查工作由不同的 Maintainer 负责，并设计充足的 CI 来保证代码质量；同时，用产品作为代码的真实评价，对厂商或是贡献者充分开放。

Conclusion

Appendix