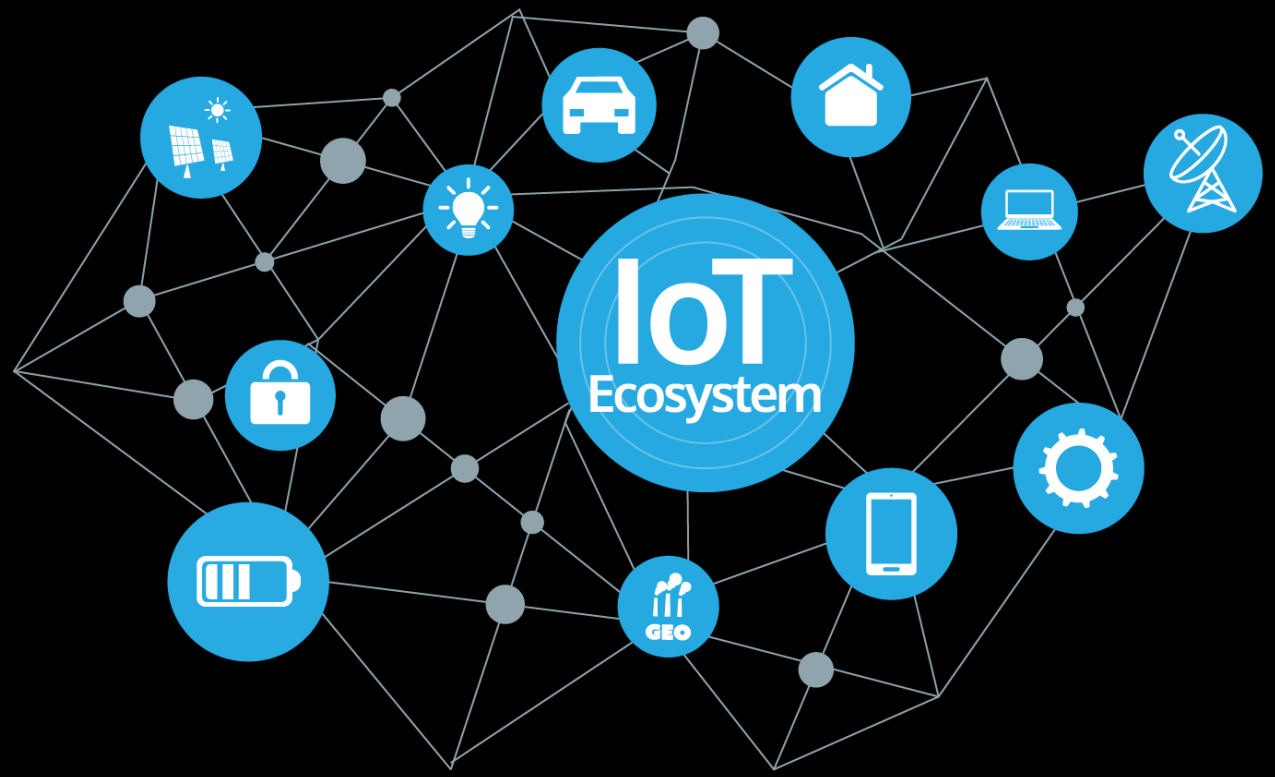
Ridiculous-includeOS

小组成员：刘紫檀 张博文 虞佳焕 汤兖霖

目录

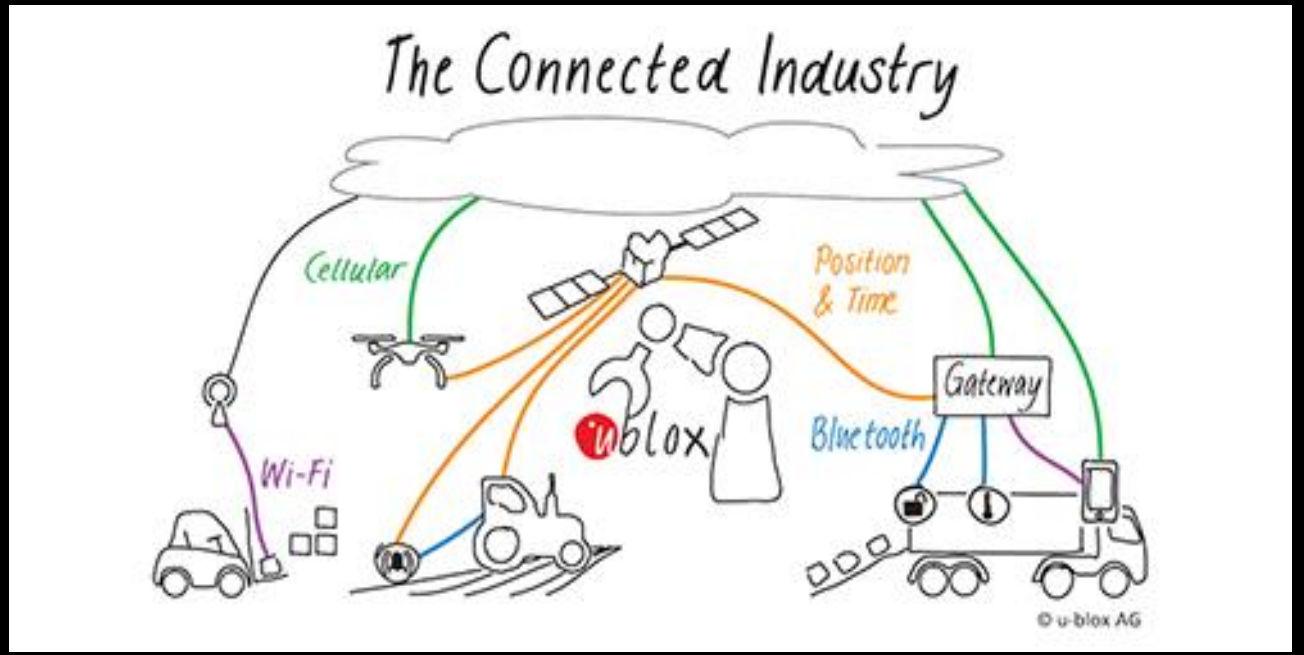
IoT Introduction

- IoT(Internet of Things)即物联网,它的定义为把所有物品通过射频识别等信息传感设备与互联网连接起来，实现智能化识别和管理。
- 要求：可靠传递,通过各种电信网络与互联网的融合,将物体的信息实时准确地传递出去。



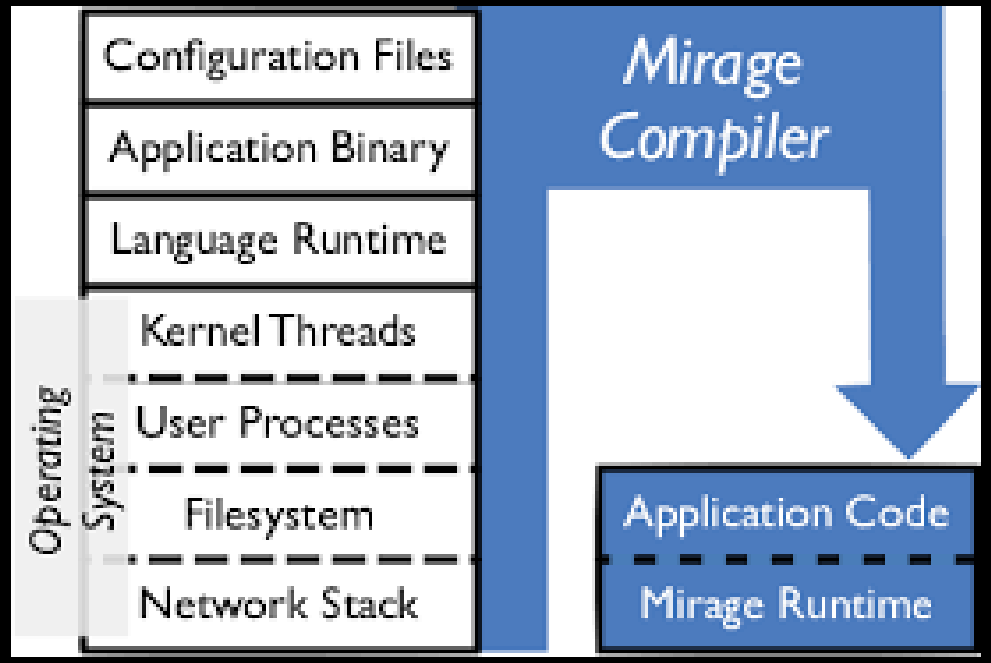
IoT 目前存在的问题及解决方案

- 延迟高、通信不可靠：物联网有很多领域对延迟和通信的可靠性具有很高的要求，如果出现问题会造成比较严重的后果。
- 安全性有待提升：传统操作系统组件庞杂，攻击面广，敏感信息容易泄露，造成安全问题 。
- 解决方案之一：使用 Unikernel。



什么是 Unikernel ？

- 它是专用的，单地址空间的，使用 Library OS 构建出来的镜像。
- Unikernel 不仅可以运行在 Bare-Metal 上，也可以运行在虚拟机上。



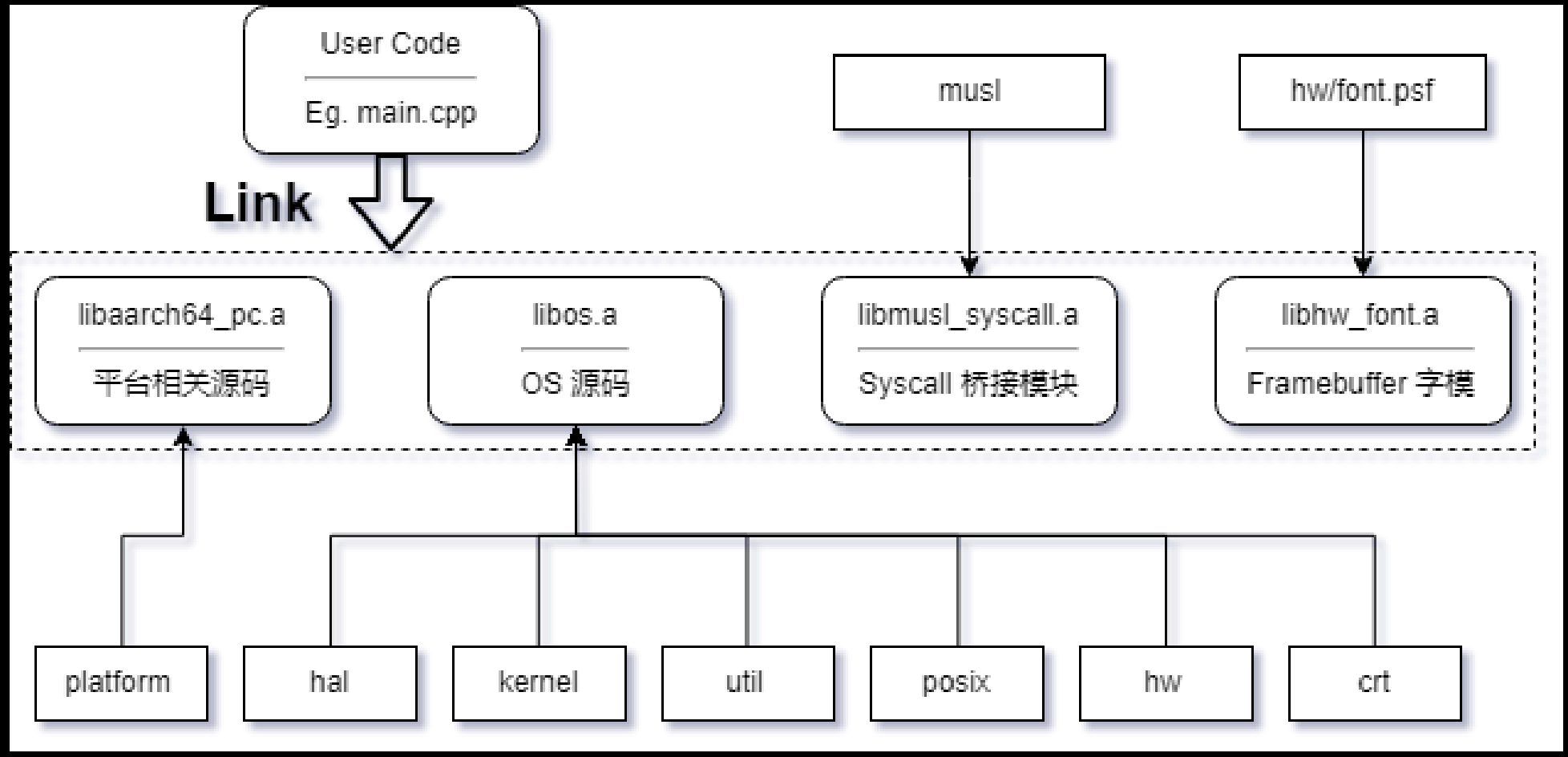
IncludeOS简介

- 一个 Unikernel 的开源实现
- 性能优良，启动迅速，最快能在几十毫秒之内启动
- 体积小，通常只需很小的磁盘和内存
- 系统组件少，通常更安全
- 支持在部分裸机上运行
- 延迟很低
- 使用 C++ 构建，应用程序开发较其它 Unikernel 平易近人

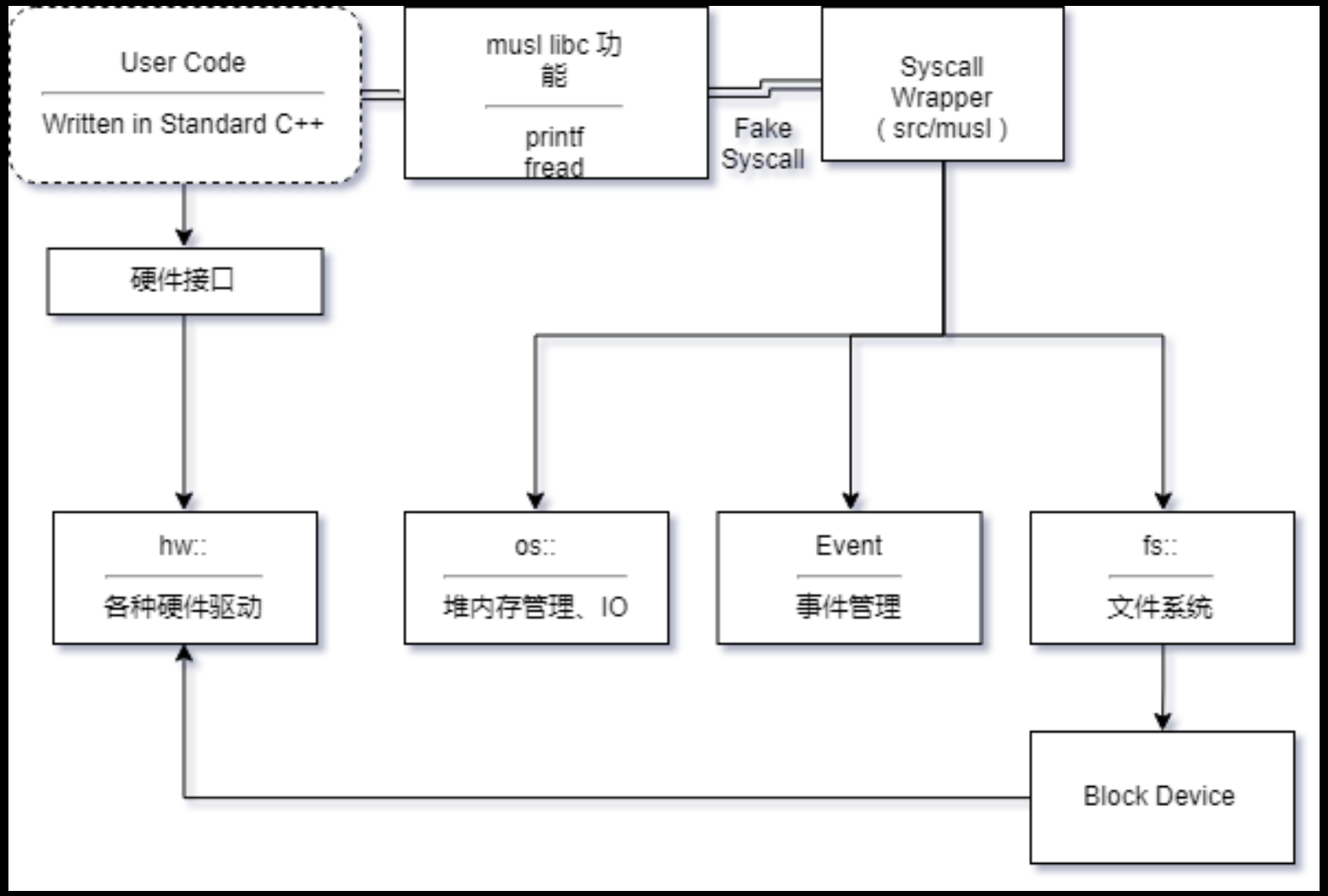
项目进展简介

- 基本完成预定目标
- 有一些问题还未解决

IncludeOS Source Architecture



IncludeOS Runtime Architecture



Conan 简介

- Conan 在新版本（IncludeOS v0.15+）中被正式采用
- Conan 是一个为 C / C++ 程序设计的包管理器

- 通过把需要的库、依赖打成包
  - 管理预编译的二进制 / 编译选项集合 – CFlag & CXXFlag
  - 管理 IncludePath 和要链接的库 – IncludeDirs & Target Link Libraries
  - 按依赖顺序获取并按包内脚本编译包 - Dependencies
  - 管理包在不同平台下不同选项的编译 - Profile & Settings

Workflow with Conan

- For Executable Project
  1. conan install
  2. cmake
  3. make
- For Library Project in Editable Mode
  1. conan install
  2. conan build
  3. conan create / export
  4. Publish, etc.

Conan Profile

```
[settings]
os=Linux
os_build=Linux
arch=armv8
arch_build=x86_64
compiler=gcc
compiler.version=8
compiler.libcxx=libstdc++11
build_type=Release
[options]
[build_requires]
*: binutils/2.31@includeos/toolchain
[env]
CC=aarch64-linux-gnu-gcc-8
CXX=aarch64-linux-gnu-g++-8
CFLAGS=-mcpu=cortex-a53 -O2 -g
CXXFLAGS=-mcpu=cortex-a53 -O2 -g
```

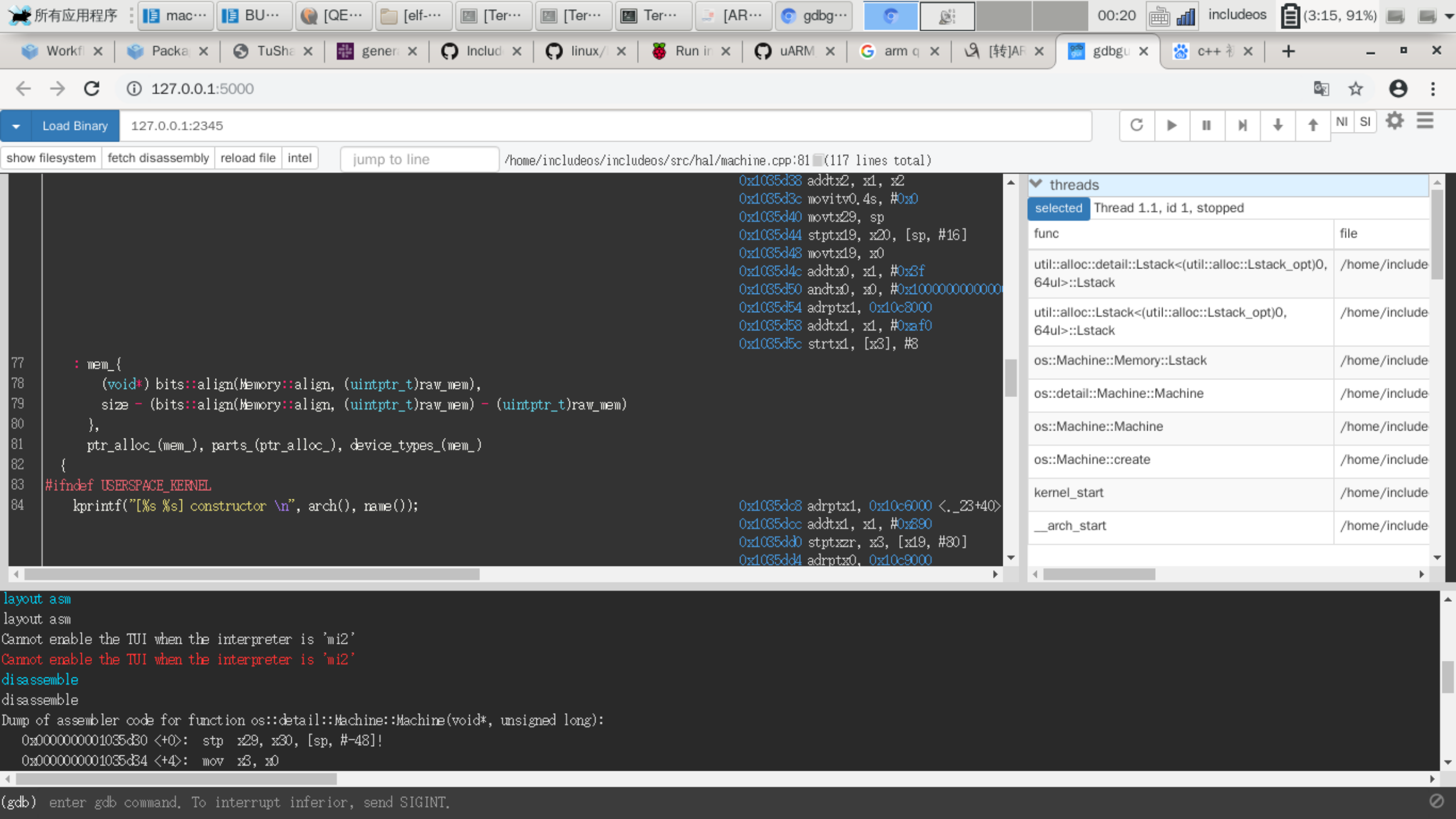
Build 过程

1. 准备 Profile 文件
   - 交叉编译一定要选择好正确的 CC
2. 在 IncludeOS repo 下运行 conan install，安装所有依赖包
3. Conan editable add && conan build
4. 到用户程序目录进行 conan install && cmake && make

- 更具体的细节请参见繁复的 reports/build.md

如何调试裸机 Kernel / Bootloader

- qemu-system-aarch64 –M raspi3 –kernel kernel8.img –S –gdb tcp::2345 -drive file=sd.img,format=raw,if=sd
  - 但请注意，QEMU 不检查内存对齐（包括对 SIMD 寄存器的访问），而 ARMv8a 检查；ARMv8a 的 SCTLR 只能关掉对常规寄存器的检查
- gdbgui –g aarch64-linux-gnu-gdb
  - target remote localhost:2345
  - file hello.elf.bin 或 add-symbol-file kernel8.elf 0x80000　# ELF .text.addr
- ObjGUI, Graphical Frontend to objdump
- 编译的时候开启 –g && DWARF 节在链接和 ELF 装入内存的时候没有丢失 && 源代码在编译时的位置 -> 有源码 ⇔ 汇编的信息
  - 用 ld –verbose 查看默认链接脚本，有对 DWARF 的处理行为（原样保留）


127.0.0.1:5000
Load Binary
127.0.0.1:2345
show filesystem
fetch disassembly
reload file
intel
jump to line
/home/includeos/includeos/src/hal/machine.cpp:81 (117 lines total)
0x1035d38 addtx2, x1, x2
0x1035d3c movitv0.4s, #0x0
0x1035d40 movtx29, sp
0x1035d44 stptx19, x20, [sp, #16]
0x1035d48 movtx19, x0
0x1035d4c addtx0, x1, #0x3f
0x1035d50 andtx0, x0, #0x1000000000000
0x1035d54 adrptx1, 0x10c8000
0x1035d58 addtx1, x1, #0xaf0
0x1035d5c strtx1, [x3], #8
77 : mem_{
78 (void*) bits::align(Memory::align, (uintptr_t)raw_mem),
79 size - (bits::align(Memory::align, (uintptr_t)raw_mem) - (uintptr_t)raw_mem)
80 },
81 ptr_alloc_(mem_), parts_(ptr_alloc_), device_types_(mem_)
82 {
83 #ifndef USERSPACE_KERNEL
84 kprintf("[%s %s] constructor \n", arch(), name());
0x1035dc8 adrptx1, 0x10c6000 <._23+40>
0x1035dcc addtx1, x1, #0x890
0x1035dd0 stptxzr, x3, [x19, #80]
0x1035dd4 adrptx0, 0x10c9000
threads
selected
Thread 1.1, id 1, stopped
func
file
util::alloc::detail::Lstack<(util::alloc::Lstack_opt)0, 64ul>::Lstack
/home/include
util::alloc::Lstack<(util::alloc::Lstack_opt)0, 64ul>::Lstack
/home/include
os::Machine::Memory::Lstack
/home/include
os::detail::Machine::Machine
/home/include
os::Machine::Machine
/home/include
os::Machine::create
/home/include
kernel_start
/home/include
__arch_start
/home/include
layout asm
layout asm
Cannot enable the TUI when the interpreter is 'mi2'
Cannot enable the TUI when the interpreter is 'mi2'
disassemble
disassemble
Dump of assembler code for function os::detail::Machine::Machine(void*, unsigned long):
0x0000000001035d30 <+0>: stp x29, x30, [sp, #-48]!
0x0000000001035d34 <+4>: mov x3, x0
(gdb) enter gdb command. To interrupt inferior, send SIGINT.

moserial

文件(F) 编辑(E) 帮助(H)

连接 记录 发送文件 接收文件 端口设置 首选项 帮助

接收的 ASCII 接收的 HEX

```
mem_offset :

RAM BASE :

RAM SIZE :

Synchronous: Data abort, same EL, Address size fault at level 1:
  ESR_EL1 0000000096000061 ELR_EL1 0000000001035D68
 SPSR_EL1 00000000200003C4 FAR_EL1 0000000001102018
```

发送的 ASCII 发送的 HEX

上传

发送 ASCII CR+LF 结束

/dev/ttyUSB0 断开 115200,8N1 发送：0，接收：0

Executable and Linkable Format

- Executable: 将正确的代码段装入内存，跳转到起始地址执行
- Linkable: 正确的将不同目标文件链接在一起

- Section != Segment
- ELF Phdr 中存储着所有 Segment 的权限和 VMA
- 按照 VMA 的指示将 ELF 的各个 Segment 装入内存
- 跳转到 VMA 处执行

AArch64 Bootloader for IncludeOS

- IncludeOS 需要 ELF 信息初始化 Musl libc 库
  - ELF Auxiliary Vector 中的 AT_PHDR, AT_PHENT 和 AT_PHNUM
    - 可以参见 man getauxval
  - 用来实现 TLS 相关逻辑（musl 的 __init_tls()）
- 树莓派的 Bootloader 只能加载 Flat Binary / Linux Image
- ARM 的硬件配置种类多样，需要统一的板级信息接口
  - Linux 使用 Flattened Device Tree 实现解耦合
  - Bootloader 需要加载 FDT 到内存，供操作系统完成初始化
- **那么，如何实现？**

Multiboot Specification

- Introduced by GNU GRUB
- 将 Header 放在前 8K 的区域
- Bootloader 负责根据 Header 地址计算出下面的内容的真实偏移，然后加载入内存的对应位置
- 简化 Bootloader 实现，需要目标可执行文件配合

```
typedef struct {
    uint32_t magic;
    uint32_t flags;
    uint32_t checksum;
    uint32_t header_addr;
    uint32_t load_addr;
    uint32_t load_end_addr;
    uint32_t bss_end_addr;
    uint32_t entry_addr;
} multiboot_hdr;
```

Place Multiboot Information

```
.text
.align 4
_MULTIBOOT_START_:
// Multiboot header
// Must be aligned, or we'll have a hard time finding the header
.word 0x1BADB002
.word 0x00010003
.word 0 - 0x1BADB002 - 0x00010003
.word _MULTIBOOT_START_  // .extern _MULTIBOOT_START_
.word _LOAD_START_       // .extern _LOAD_START_
.word _LOAD_END_         // .extern _LOAD_END_
.word _end               // .extern _end
.word _start             // .extern _start
```

Linker Script

```
SECTIONS
{
    PROVIDE ( _ELF_START_ = 0x1000000);
    PROVIDE ( _LOAD_START_ = _ELF_START_);
    __stack_top = _ELF_START_ - 16;

    . = _ELF_START_ + SIZEOF_HEADERS;

    // To link: ld -T linker.ld
}
```

AArch64 Bootloader for IncludeOS

- 将 IncludeOS ELF Binary 和 DTB 文件用 objcopy 处理成 .data 段的目标文件，和前面的 Bootloader 代码链接在一起
- 如何在运行时让 Bootloader 知道 .data 段中两个文件的起始地址和结束地址？
  - objcopy 在导出时会同时导出相对 .data 的两个符号
  - 所以只需要在 C Source 中 extern char _binary_hello_elf_bin_start;
  - 注意，**其符号为其值**，所以 &_binary_hello_elf_bin_start 为其地址
  - （类型无关紧要）

Bugs

- 裸机在 util::Lstack、libc 等各种奇怪场合抛异常，但是模拟器运行的好好的
  - 根据在 QEMU 中的调试结果，是编译器 emit 的对 SIMD 寄存器的 ld/st 指令未对齐，引发 Alignment Fault
  - 可能的解决方案：完善异常处理，模拟 Unaligned 指令执行；定位并且修改某些非对齐数据结构的访问
- 在高等级的优化选项下，在奇怪的地方异常
  - 原因大体可能与第一条相似，还没有仔细分析此问题

吐槽 #1

- 我以为的 C++

```
class Box {
   public:
      double length;
      double breadth;
};
……
   Box Box1;
   Box1.height = 5.0;
   Box1.length = 6.0;
```

- IncludeOS 的 C++

```
template<
  typename T,
  typename C = typename detail::closure_decay<T, R, Args...>::type
> explicit inplace_triv(T&& closure) :
  invoke_ptr_{ static_cast<invoke_ptr_t>(
    [](storage_t& storage, Args&&... args) -> R
    { return reinterpret_cast<C&>(storage)(std::forward<Args>(args)...); }
  )}
```

吐槽 #2

- 我以为的 Cmake

```
PROJECT(main)
CMAKE_MINIMUM_REQUIRED(VERSION 2.6)
ADD_SUBDIRECTORY( src )
AUX_SOURCE_DIRECTORY(. DIR_SRCS)
ADD_EXECUTABLE(main ${DIR_SRCS}  )
TARGET_LINK_LIBRARIES( main Test )
```

- IncludeOS 的 CMake

```
os.cmake ( 398 Lines )
Includeos.cmake ( 75 Lines )
……
```

GPIO 支持

- 本质上是对特定内存读写的封装
- ~~（可以简单地重写Lab1）~~

```c
#include "GPIO.h"

int gpio_test_main()
{
    volatile __uint8_t led = 29;
    gpio_func_select(led, GPIO_FSEL_OUTP);
    int i;
    // blink 200 times.
    for (i = 0; i < 200; i++)
    {
        gpio_set(led);
        gpio_delay(1000000);
        gpio_clr(led);
        gpio_delay(1000000);
    }
    return 0;
}
```

Mailbox简介

- VideoCore 是 Broadcom 使用的非传统意义的GPU。
- 树莓派的 boot 是由 VideoCore 负责的。
- VideoCore 还负责外设的管理。
- Mailbox 是 CPU 与 VideoCore 通信的工具。

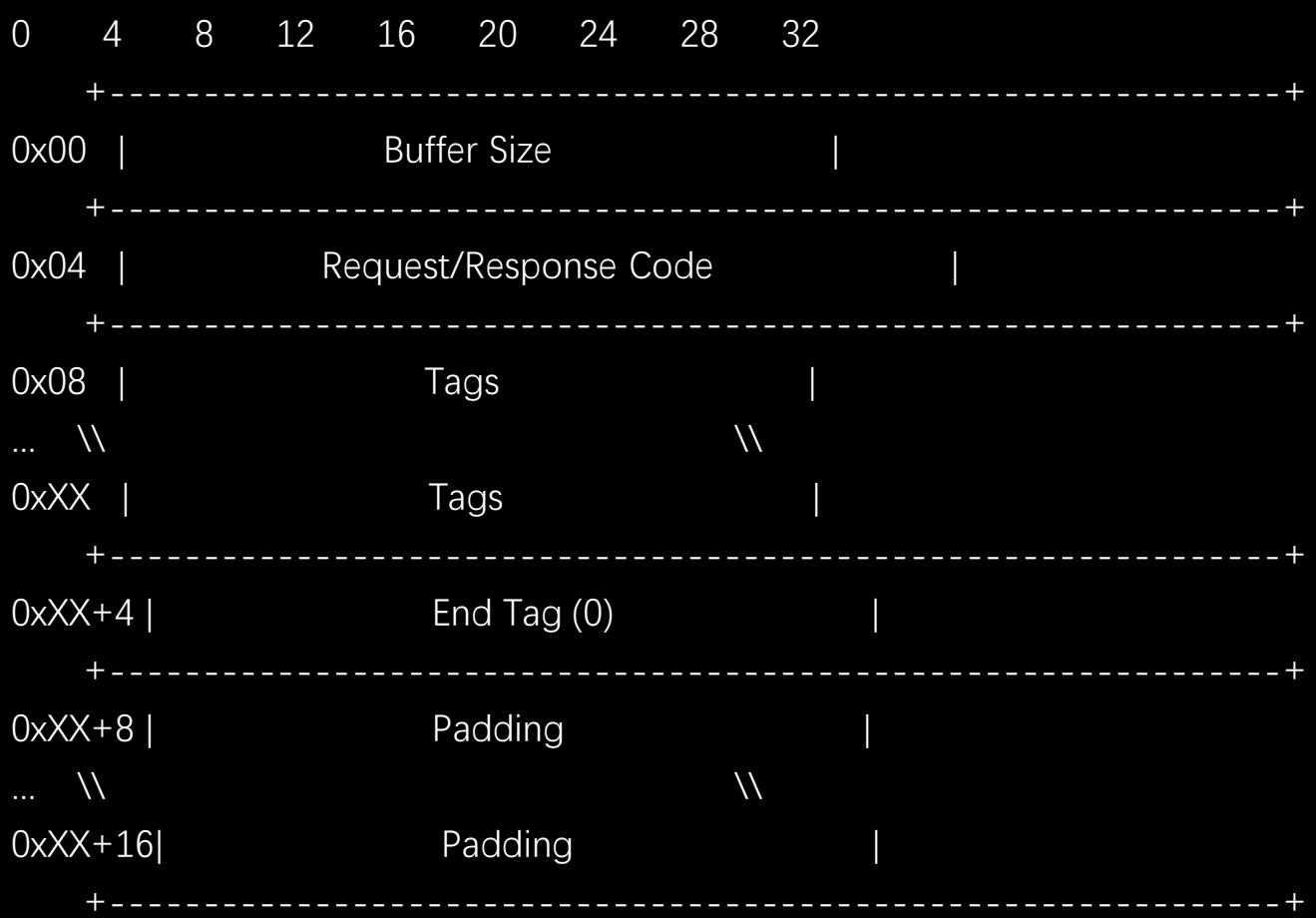
Mailbox简介

- 可用的channels，不同的channel，消息格式不同
  0: Power management
  1: Framebuffer
  2: Virtual UART
  3: VCHIQ
  4: LEDs
  5: Buttons
  6: Touch screen
  7:
  8: Property tags (ARM -> VC)
  9: Property tags (VC -> ARM)
- 本次主要使用 channel 8

Mailbox简介

- “Messages have a fairly complex and poorly documented structure.”
- 消息格式

Mailbox简介


0 4 8 12 16 20 24 28 32
0x00 | Buffer Size |
0x04 | Request/Response Code |
0x08 | Tags |
... \\ \\
0xXX | Tags |
0xXX+4 | End Tag (0) |
0xXX+8 | Padding |
... \\ \\
0xXX+16| Padding |

Mailbox简介

- Message Tag 格式:
  - 0x000XYZZZ
    - X为硬件设备标识
    - Y为命令种类:
      - 0 = get, 4 = test, 8 = set
    - ZZZ 为特定的命令标识
  - Buffer size（参数长度）
  - Request/response code（0 if request， 0x80000000 + length of the result if response）
  - 参数或结果

Screen 支持

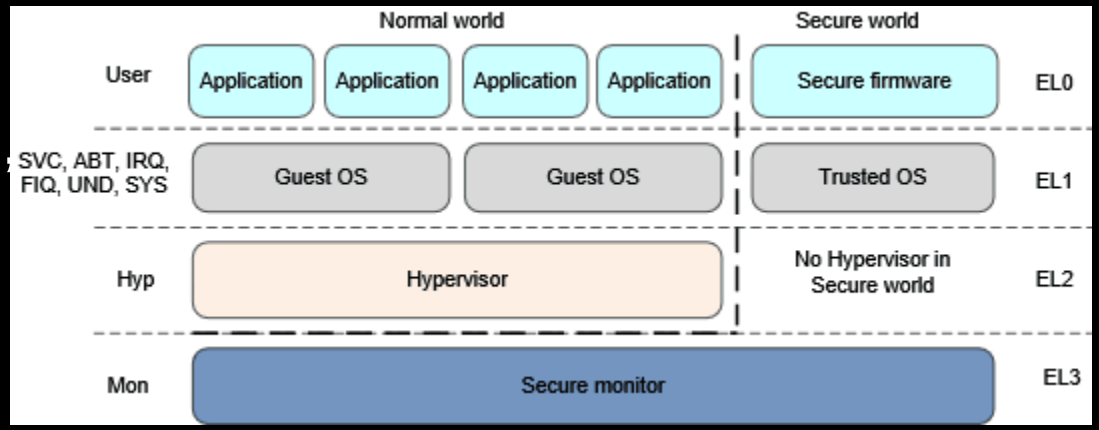
- Framebuffer是Linux抽象出来提供给用户进程读写屏幕的设备
- 可以直接通过写framebuffer来向屏幕输出
- 期间需要使用mailbox和VideoCore通信

UART 支持

- 默认情况下，UART发送和接收引脚分别位于GPIO 14和GPIO 15上
- 首先使用 Mailbox 设置时钟频率
- 设置 GPIO 14 和 15 的 function 为 alto
- 设置 pull-up/down resistor
- 设置 UART 传输参数

AArch64

- AArch64 处理器有 4 个运行状态，
  - EL0 : Applications
  - EL1 : Kernel
  - EL2 : Hypervisor
  - EL3 : Secure monitor
- Raspberry Pi 3b+ 的 SoC 支持 EL0~2
- 在本项目中，为了减少 Exception Level 切换的开销，降低响应时间，所有的代码都运行在 Exception Level 1

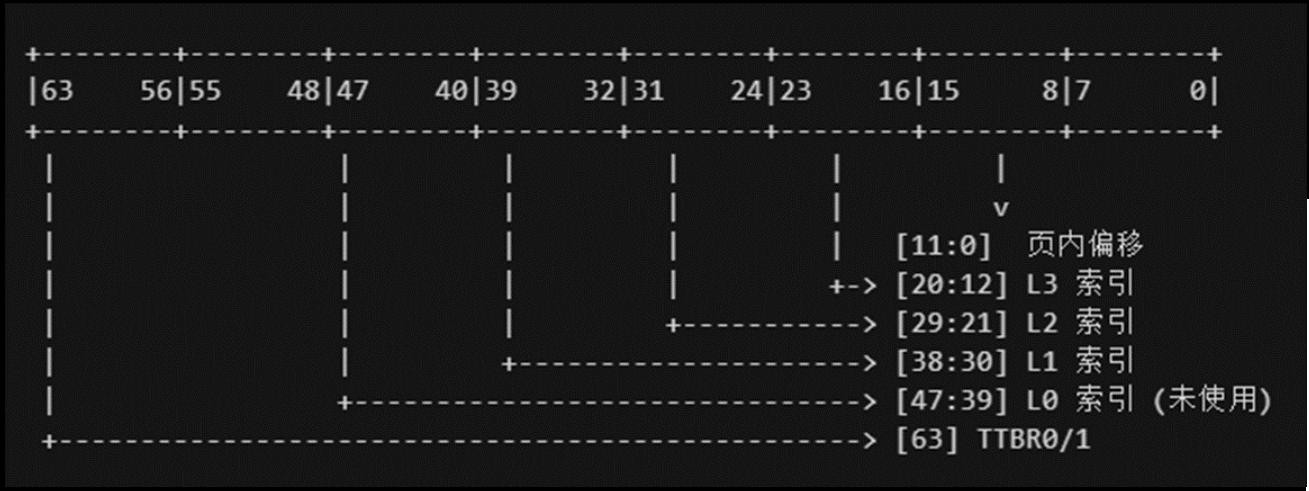


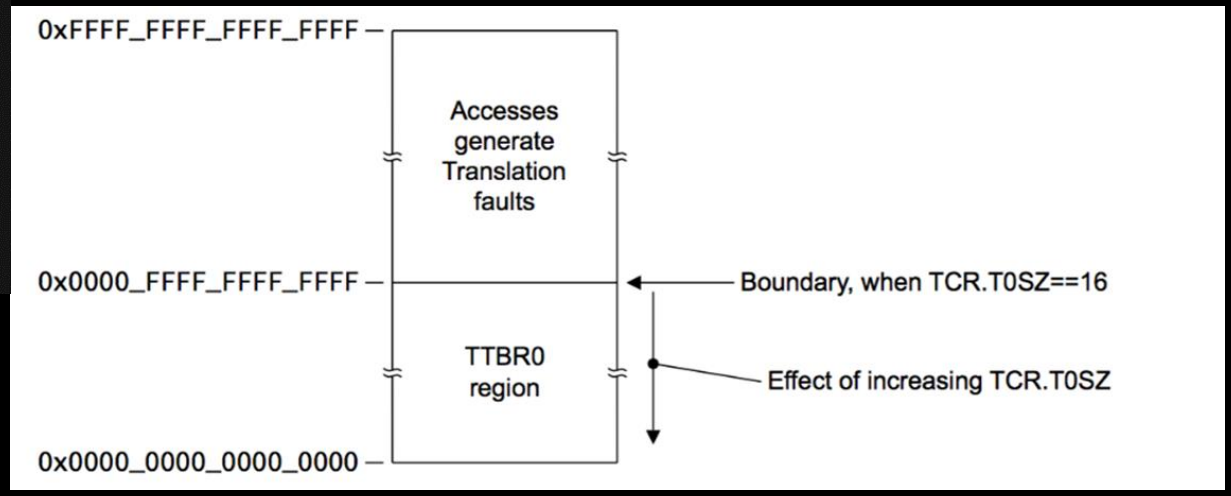
MMU in Arch64

- AArch64 System Registers About MMU
  - TCR (Translation Control Register) : Controls translation table walks required for stage 1 translation of memory accesses and holds cacheability and shareability information for the accesses.
  - MAIR (Memory Attribute Indirection Registers) : To provide the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations.
  - TTBR0/1 (Translation Table Base Register) : Holds the address of the translation table (0 for user space address, 1 for kernel sapce)
    - IncludeOS 只使用 user space address, 因此只需配置 TTBR0
  - SCTLR (System Control Register) : Hold control bit of MMU, cache, aligment check, etc.

MMU in Arch64

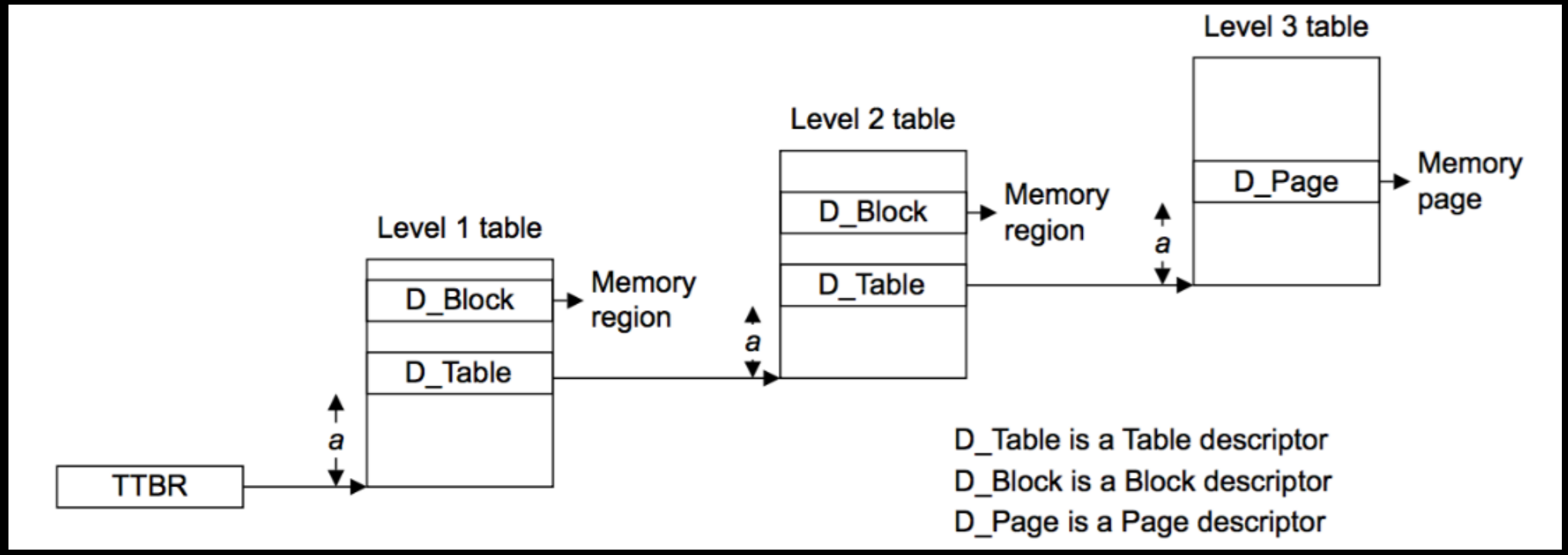
- 本项目目前实现了一个最大地址空间 1GB、页大小为 4KB 的 3 级用户空间页表，目前只把逻辑地址转化成值一样的物理地址。





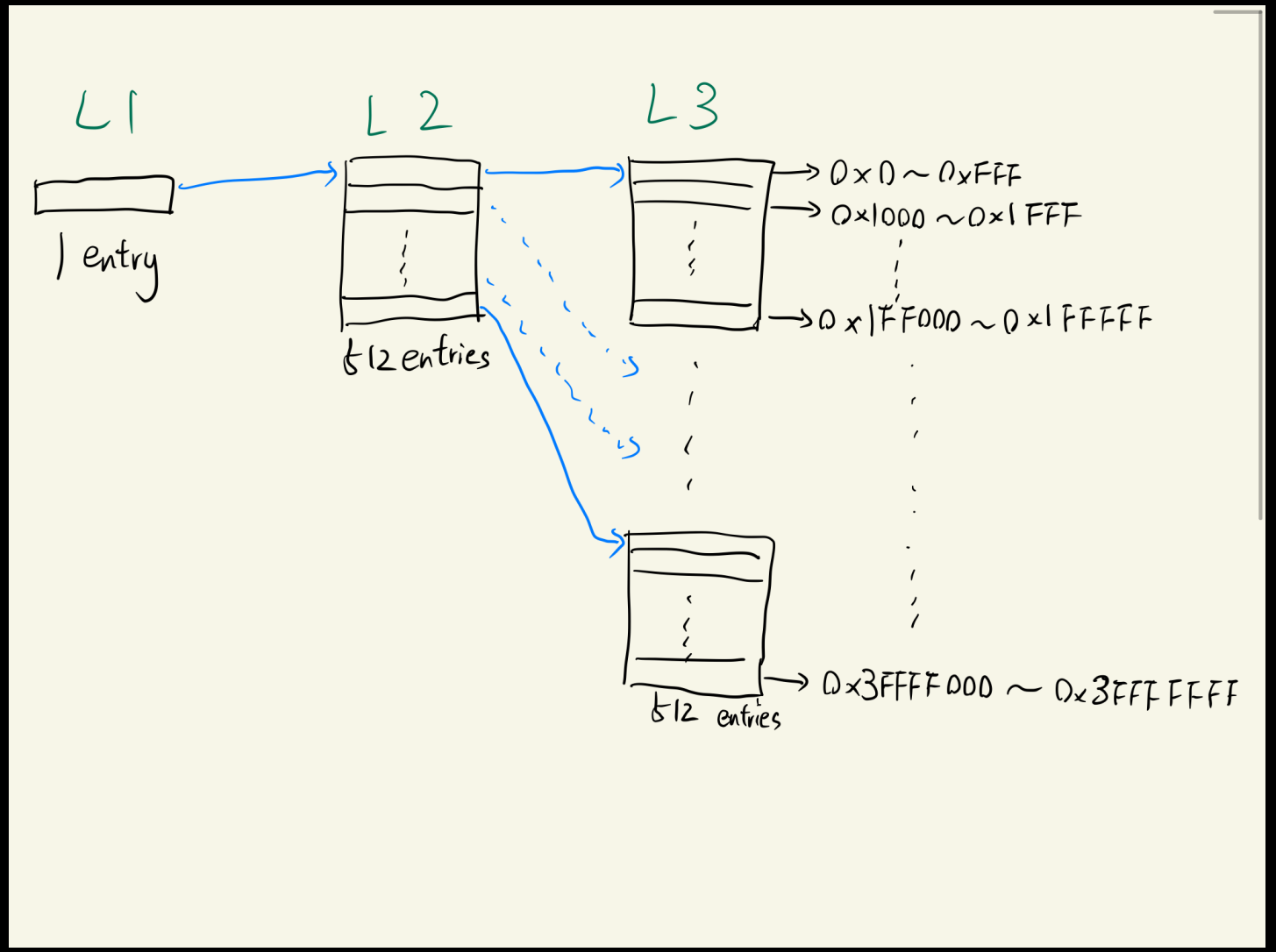
MMU in Arch64

- 地址翻译过程



MMU in Arch64

- 页表结构



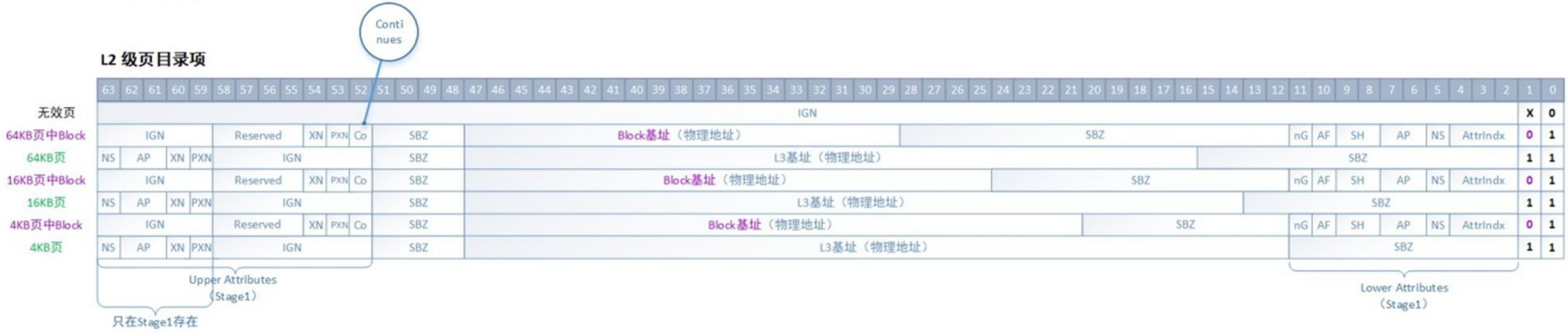
MMU in Arch64

- 有关页表属性的一些 Flags

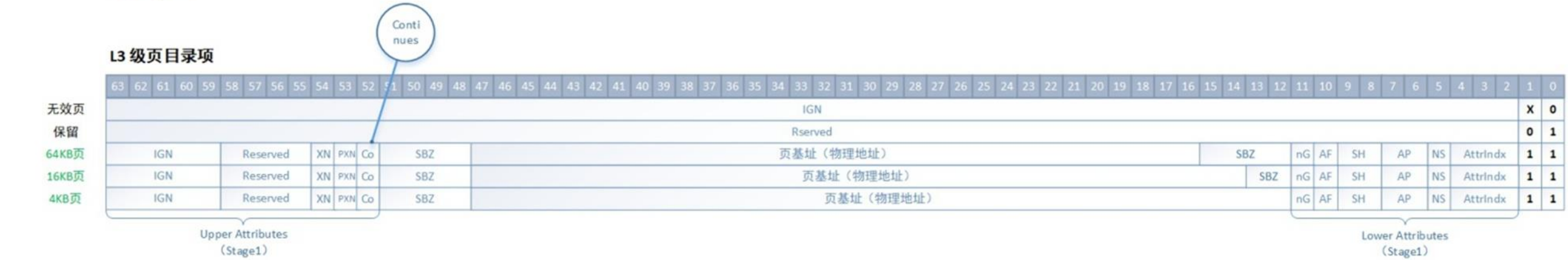
```
#define PAGESIZE 4096

// granularity
#define PT_PAGE 0b11  // 4k granule
#define PT_BLOCK 0b01 // 2M granule
// accessibility
#define PT_KERNEL (0 << 6) // privileged, supervisor EL1 access only
#define PT_USER (1 << 6)   // unprivileged, EL0 access allowed
#define PT_RW (0 << 7)     // read-write
#define PT_RO (1 << 7)     // read-only
#define PT_AF (1 << 10)    // accessed flag
#define PT_NX (1UL << 54)  // no execute
// shareability
#define PT_OSH (2 << 8) // outter shareable
#define PT_ISH (3 << 8) // inner shareable
// defined in MAIR register
#define PT_MEM (0 << 2) // normal memory
#define PT_DEV (1 << 2) // device MMIO
#define PT_NC (2 << 2)  // non-cachable

#define TTBR_CNP 1
```

L1 级页目录项

L2 级页目录项



L3 级页目录项



MMU in Arch64

- L1

```
page_base[0 * page_entries + 0] = (page_addr + 1 * PAGESIZE) | // L2 physical address
                                  PT_PAGE |                   // page
                                  PT_AF |                     // accessed flag
                                  PT_USER |                   // non-privileged
                                  PT_ISH |                    // inner shareable
                                  PT_MEM;                     // normal memory
```

- L2

```
page_base[1 * page_entries + i] = (page_addr + (2 + i) * PAGESIZE) | // L3 physical address
                                  PT_PAGE |                          // page
                                  PT_AF |                            // accessed flag
                                  PT_USER |                          // non-privileged
                                  PT_ISH |                           // inner shareable
                                  PT_MEM;                            // normal memory
```

MMU in Arch64

- L3

```
uint64_t page_index = i * page_entries + j;
uint64_t entry = (page_index * PAGESIZE) | // L3 physical address
                 PT_PAGE |                 // page
                 PT_AF |                   // accessed flag
                 PT_USER;                  // non-privileged
// executable RO
if ((page_index >= text_start_page) && (page_index < exec_end_page))      You, a fe
{
    entry |= (PT_RO | PT_ISH | PT_MEM);  // RO, inner shared, memory
}
// read only data
else if ((page_index >= ro_start_page) && (page_index < ro_end_page))
{
    entry |= (PT_RO | PT_NX | PT_ISH | PT_MEM);  // ro, nx inner shared, memory
}
// mmio Outter shared
else if (page_index >= mmio_start_page)
{
    entry |= (PT_NX | PT_RW | PT_OSH | PT_DEV);  // rw, nx, outter shared, memory
}
// other RW
else
{
    entry |= (PT_NX | PT_RW | PT_ISH | PT_MEM);  // rw, nx, inner shared, memory
}
```

MMU in Arch64

- 配置好的页表把包含程序可执行部分的页设置为只读，其他页设置为不可执行 (NX)，对于静态编译的 IncludeOS，能减少大量程序被恶意破坏的风险
- 受限于中断系统的工作进度，目前的页表配置完成后是静态的，暂不能处理缺页中断，也不支持虚拟内存
- 由于树莓派相较于普通的物联网设备有足够大的内存，一般任务并不需要用到虚拟内存

Exception

- Interrupt
  - IRQ（普通中断）
  - FIQ （快速中断，更高的处理优先级）
  - Serror (System Error)
- Aborts
  - synchronous： Instruction/Data Abort， Page Fault
  - asynchronous：外部硬件故障
- Reset
- System Call
  - Supervisor Call (SVC)： User Program 向 Kernel 申请服务
  - Hypervisor Call (HVC)： Guest OS 向 Hypervisor 申请服务
  - Secure monitor Call (SMC)： 切换进入 Secure Mode

Exception

- AArch64 的异常处理用到以下几个 System Registers
  - VBAR (Vector Base Address Register)： 异常向量表基地址
  - ESR (Exception Syndrome Register)： 记录异常类型、原因
  - ELR (Exception Linker Register)： 发生异常的 PC
  - SPSR (Saved Program Status Register)： 发生异常时的 Program Status Register
  - FAR (Fault Address Register)： 记录产生 Data Abort 异常的访存地址

Exception

- 异常向量表的组织结构如右
- 需要在对应的的偏移量中放入合适的程序

Address	Exception type	Description
VBAR_ELn + 0x000	Synchronous	Current EL with SP0
+ 0x080	IRQ/vIRQ	
+ 0x100	FIQ/vFIQ	
+ 0x180	SError/vSError	
+ 0x200	Synchronous	Current EL with SPx
+ 0x280	IRQ/vIRQ	
+ 0x300	FIQ/vFIQ	
+ 0x380	SError/vSError	
+ 0x400	Synchronous	Lower EL using AArch64
+ 0x480	IRQ/vIRQ	
+ 0x500	FIQ/vFIQ	
+ 0x580	SError/vSError	
+ 0x600	Synchronous	Lower EL using AArch32
+ 0x680	IRQ/vIRQ	
+ 0x700	FIQ/vFIQ	
+ 0x780	SError/vSError	

Exception

- 为了便于调试，小组针对部分布异常做了异常处理

```
.align 11
_vectors:
// synchronous
.align 7
mov x0, #0
mrs x1, esr_el1
mrs x2, elr_el1
mrs x3, spsr_el1
mrs x4, far_el1
b exc_handler

// IRQ
.align 7
mov x0, #1
mrs x1, esr_el1
mrs x2, elr_el1
mrs x3, spsr_el1
mrs x4, far_el1
b exc_handler

// FIQ
.align 7
mov x0, #2
mrs x1, esr_el1
mrs x2, elr_el1
mrs x3, spsr_el1
mrs x4, far_el1
b exc_handler

// SError
.align 7
mov x0, #3
mrs x1, esr_el1
mrs x2, elr_el1
mrs x3, spsr_el1
mrs x4, far_el1
b exc_handler
```

Exception

```
void exc_handler(unsigned long type, unsigned long esr, unsigned long elr, unsigned long spsr, unsigned long far)
{
    // print out interruption type
    switch(type) {
        case 0: uart_puts("Synchronous"); break;
        case 1: uart_puts("IRQ"); break;
        case 2: uart_puts("FIQ"); break;
        case 3: uart_puts("SError"); break;
    }
    uart_puts(": ");
    // decode exception type
    switch(esr>>26) {
        case 0b000000: uart_puts("Unknown"); b
        case 0b000001: uart_puts("Trapped WFI/
        case 0b001110: uart_puts("Illegal exec
        case 0b010101: uart_puts("System call"
        case 0b100000: uart_puts("Instruction
        case 0b100001: uart_puts("Instruction
        case 0b100010: uart_puts("Instruction
        case 0b100100: uart_puts("Data abort,
        case 0b100101: uart_puts("Data abort,
        case 0b100110: uart_puts("Stack alignm
        case 0b101100: uart_puts("Floating poi
        default: uart_puts("Unknown"); break;
    }
    // decode data abort cause
    if(esr>>26==0b100100 || esr>>26==0b100101) {
        uart_puts(", ");
        switch((esr>>2)&0x3) {
            case 0: uart_puts("Address size fault"); break;
            case 1: uart_puts("Translation fault"); break;
            case 2: uart_puts("Access flag fault"); break;
            case 3: uart_puts("Permission fault"); break;
        }
        switch(esr&0x3) {
            case 0: uart_puts(" at level 0"); break;
            case 1: uart_puts(" at level 1"); break;
            case 2: uart_puts(" at level 2"); break;
            case 3: uart_puts(" at level 3"); break;
        }
    }
    // dump registers
    uart_puts(":\n  ESR_EL1 ");
    uart_hex(esr>>32);
    uart_hex(esr);
    uart_puts(" ELR_EL1 ");
    uart_hex(elr>>32);
    uart_hex(elr);
    uart_puts("\n SPSR_EL1 ");
    uart_hex(spsr>>32);
    uart_hex(spsr);
    uart_puts(" FAR_EL1 ");
    uart_hex(far>>32);
    uart_hex(far);
    uart_puts("\n");
}
```

USB

- 树莓派的网卡作为 USB 设备挂载，没有 USB 支持就无法联网
  - 树莓派的千兆网卡被挂载在 USB 2.0 总线上
- USB 系统的复杂度比较高
  - USB 2.0 Specification 有 650 页
  - 目前找到的最简单的树莓派 USB 系统的实现也有 3000+ 行代码
- 小组成员找到了一个开源的树莓派 USB 系统的实现，但涉及中断系统以及 DMA，所以 USB 系统暂时难以整合
- 目前关于 USB 的工作主要是熟悉 USB 设备树的结构和枚举过程等
- 吐槽：IncludeOS 的项目结构比较杂乱，并且使用的 Modern C++ 有太多的 magic

eMMC

- 树莓派通过操作特定的 MMIO 实现对 eMMC 设备(SD Card) 的读写
- 一个 eMMC 基本操作大致分为如下几个步骤
  - 数据准备
  - 通过 MMIO 发送命令
  - 通过 MMIO 传送数据

eMMC

```
#define EMMC_ARG2
#define EMMC_BLKSIZECNT
#define EMMC_ARG1
#define EMMC_CMDTM
#define EMMC_RESP0
#define EMMC_RESP1
#define EMMC_RESP2
#define EMMC_RESP3
#define EMMC_DATA
#define EMMC_STATUS
#define EMMC_CONTROL0
#define EMMC_CONTROL1
#define EMMC_INTERRUPT
#define EMMC_INT_MASK
#define EMMC_INT_EN
#define EMMC_CONTROL2
#define EMMC_SLOTISR_VER
```

```
/**...
int sd_status(unsigned int mask)
{...
}

/**...
int sd_int(unsigned int mask)
{...
}

/**...
int sd_cmd(unsigned int code, unsigned int arg)
{...
}

/**...
int sd_readblock(unsigned int lba, unsigned char *buffer, unsigned int num)
{...
}

/**...
int sd_writeblock(unsigned char *buffer, unsigned int lba, unsigned int num)
{...
}

/**...
int sd_clk(unsigned int f)
{...
}

/**...
int sd_init()
{...
}
```

```
00000
00004
00002

00000
00000
00000
f0000
e0000
00020
00004
00002
00001
```

```
00
00
00
00
0x08020000
0x0C030000
0x11220010
0x12220032
0x17020000
0x18220010
0x19220032
0x37000000
(0x06020000|CMD_NEED_APP)
(0x29020000|CMD_NEED_APP)
(0x33220010|CMD_NEED_APP)
```

eMMC

- IncludeOS 项目里已经有平台无关的 FS, VFS 等软件部分的部分实现，需要软硬件层面的对接

```
int sd_init();
int sd_readblock(unsigned int lba, unsigned char *buffer, unsigned int num);
int sd_writeblock(unsigned char *buffer, unsigned int lba, unsigned int num);
```

File system 支持

- 硬件设备的接口
- 文件系统(FAT32 魔改版)的初始化
- 数据结构之间的转换
- 文件项的维护
- 异步操作

VFS 实现

- 一个单例
- 两个 std::map 分别维护设备和文件节点的挂载
- VFS_entry： Node in virtual file system tree
- 可以比如任何东西到 VFS 上，比如挂载一个函数
- file_fd 只有雏形，尚未对接

未来工作展望

- 裸机 boot
- USB 驱动支持
- 网络支持

谢谢