溪江朔

vilalge

Village Kernel开发指南

从零开始写内核

目录

[第一章: 序言 4](#_Toc2133346780)

[一、 Village内核特点： 4](#_Toc772361148)

[二、 Village内核目标： 4](#_Toc1682651968)

[三、 进展说明 4](#_Toc119478833)

[四、 展望未来 5](#_Toc183536286)

[第二章: 搭建开发环境 7](#_Toc907841710)

[一、 系统要求 7](#_Toc224308035)

[二、 搭建开发环境 7](#_Toc1111343760)

[三、 克隆village-kernel项目 7](#_Toc1689296361)

[四、 使用vscode打开village-kernel项目 8](#_Toc122642340)

[五、 创建rootfs文件系统镜像 8](#_Toc1812990907)

[六、 运行与调试代码 9](#_Toc292706666)

[第三章: 内核结构 11](#_Toc1783383832)

[一、 vk.application: 应用层代码 11](#_Toc902803245)

[二、 vk.bootloader: 加载器代码 11](#_Toc1442172660)

[三、 vk.hardware: 硬件层代码 11](#_Toc2095456578)

[四、 vk.kernel: 微内核代码 11](#_Toc1754379293)

[五、 vk.library: C运行库代码 13](#_Toc902304141)

[六、 vk.network: 网络层代码 13](#_Toc1643666320)

[七、 vk.scripts: 工具脚本 13](#_Toc2017688879)

[八、 vk.build：编译输出文件夹 13](#_Toc382719576)

[第四章: 执行流程 14](#_Toc654391067)

[一、 启动加载内核并跳转执行 14](#_Toc1086906782)

[二、 初始化段数据和构造函数 18](#_Toc1146383692)

[三、 跳转到main函数 20](#_Toc47430560)

[四、 初始化芯片时钟 20](#_Toc448988883)

[五、 初始化内核模块 21](#_Toc2046104670)

[六、 挂载文件系统 21](#_Toc1225549279)

[七、 加载共享库及动态模块 22](#_Toc1291073776)

[八、 注册线程任务 23](#_Toc902183944)

[九、 开始调度任务 23](#_Toc1770998988)

[十、 加载第一个应用程序 26](#_Toc1056643896)

[十一、 应用层运行控制台 27](#_Toc1471683029)

[第五章: 适配新平台 29](#_Toc2007505904)

[第六章: 编写驱动模块 29](#_Toc1036150511)

[第七章: 编写应用程序 29](#_Toc636744854)

# 序言

## Village内核特点：

### 上层功能代码与底层驱动代码分离，可移植。

### 支持模块化，可裁剪，代码模块可分离

### 支持多线程，多任务

### 可动态加载模块，类似linux的insmod，rmmod

### 可运行应用程序，命令行run appname.exec

### 运行app时会根据编译时链接的动态库，进行加载so文件

## Village内核目标：

### 可在低端的嵌入式设备运行，也可以在高端的PC运行

### 让嵌入式开发者花费更少精力在底层，有更多精力搞好应用

### 适配更多通用设备，让开发者更快实现业务

### 不为项目更换MCU，需要重新适配底层而烦恼

## 进展说明

### 目前还处于开发阶段，各功能还不完善，代码还有点垃圾。

### 适配平台不多，目前只适配了cortex-m和i686平台。

### 已完成部分：

### 内存管理

### 中断管理

### 系统调度

### 任务管理

### 工作队列

### 线程同步（互斥锁，自旋锁，信号量）

### 文件系统（FAT）

### 动态加载（加载共享库，注册模块，运行程序）

### 正进行部分：

### 完善文件系统框架

### 完善注册模块逻辑

### 编写专用Bootloader

### 待完成部分：

### 编写GUI图形界面

### 适配POSIX接口

### 适配更多平台

### 其他文件系统

### 输入子系统

### 显示子系统

### 网络功能

## 展望未来

### 短时间内单靠一个人，搞不定太多，进度实在太慢。

### 编写适配平台底层代码，重写C库，重写文件系统，太耗时间。

### 寻找更多志同道合的码友，一起参与项目开发，人多力量大。

### 目前还很简陋，但星星之火，也可以燎原，一起把火烧起来。

# 搭建开发环境

## 系统要求

mac os / linux / windows（使用wsl子系统）

## 搭建开发环境

以mac os为例 (Linux一样可以ubuntu22.04测试过)

安装vscode, git

|  |
| --- |
| 安装简单，跳过。安装完成之后打开vscode，安装C/C++拓展插件，调试代码需要。 |

安装homebrew

|  |
| --- |
| /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)" |

安装交叉编译工具

|  |
| --- |
| brew install make gcc i686-elf-binutils i686-elf-gcc i386-elf-gdb |

安装qemu模拟器

|  |
| --- |
| brew install qemu |

如果出现too many open file错误时输入：

|  |
| --- |
| ulimit -n 4096 |

## 克隆village-kernel项目

ssh方式：

|  |
| --- |
| git clone git@github.com:village-kernel/village-kernel.git |

https方式：

|  |
| --- |
| git clone https://github.com/village-kernel/village-kernel.git |

## 使用vscode打开village-kernel项目

把项目目录village-kernel拉到vscode界面

接着打开vscode终端，拷贝配置文件

|  |
| --- |
| cp vk.scripts/configs/i686.config .config |

修改配置，进入Compiler选项

|  |
| --- |
| make menuconfig |

配置宿主机编译器：

|  |
| --- |
| () host compile prefix  (-13) host compile suffix |

配置交叉编译器：

|  |
| --- |
| (i686-elf-) cross compile prefix  () cross compile suffix |

编译项目

|  |
| --- |
| make |

## 创建rootfs文件系统镜像

**Mac OS**

切换到vscode终端，拷贝文件系统镜像

|  |
| --- |
| cp vk.scripts/rootfs.img rootfs.img |

右键选中rootfs.img，在Finder中打开，双击rootfs.img文件完成挂载

修改rootfs文件系统挂载路径

|  |
| --- |
| make menuconfig |

进入Compiler选项

|  |
| --- |
| (/Volumes/VILLAGE OS) rootfs path |

拷贝相关文件到文件系统

|  |
| --- |
| make rootfs |

**Linux**

切换到vscode终端，拷贝文件系统镜像

|  |
| --- |
| cp vk.scripts/rootfs.img rootfs.img |

终端挂载rootfs.img

|  |
| --- |
| sudo mount -o offset=512 rootfs.img /mnt |

修改rootfs文件系统挂载路径

|  |
| --- |
| make menuconfig |

进入Compiler选项

|  |
| --- |
| (/mnt) rootfs path |

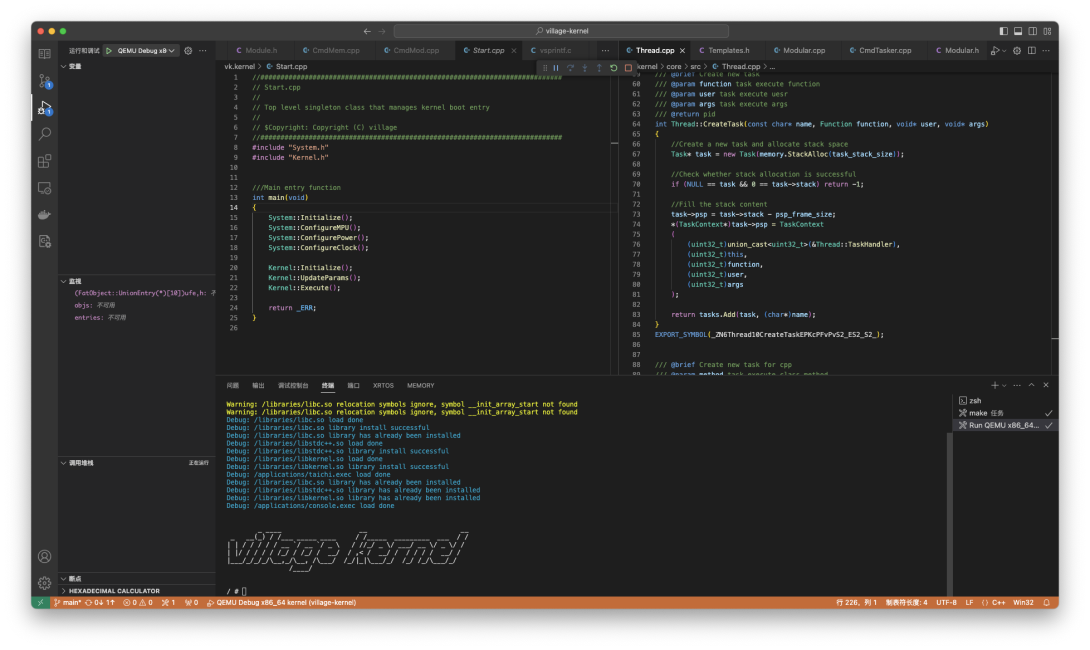
拷贝相关文件到文件系统

|  |
| --- |
| sudo make rootfs |

## 运行与调试代码

切换到vscode debug界面

选择QEMU Debug x86\_64 kernel



# 内核结构

## vk.application: 应用层代码

## vk.bootloader: 加载器代码

## vk.hardware: 硬件层代码

arch：架构名称

mcu:

|  |
| --- |
| 存放mcu厂家提供的底层代码文件，包括启动代码，链接文件。  定义编译时的相关参数以及链接时的相关参数。 |

hal:

|  |
| --- |
| 存放与处理器相关的底层hal库代码，这里抽象化硬件资源以供driver调用。 |

driver:

|  |
| --- |
| 存放专用驱动代码 |

说明：

|  |
| --- |
| 适配新平台时，主要工作都在此目录 |

## vk.kernel: 微内核代码

**arch：**

|  |
| --- |
| 主要存放一些与CPU架构相关的代码，时钟、中断与调度。  启动代码之类的属于更加底层，存放在vk.hardware子模块。  这样做的目的是为了更好剥离底层代码，底层有底层的实现方式。  为了逻辑通用，切记不要在该目录编写太多不通用的底层的代码。 |
| arm：arm平台相关目录  x86：i686平台相关目录 |

**binutils：**

|  |
| --- |
| 主要存放与ELF文件相关的代码，例如Elf加载器、执行器，共享库工具，内核模块工具。 |

**bridge：**

|  |
| --- |
| 为了POSIX接口专门预留的目录，负责桥接内核层与用户层，目前还没有做适配。 |

**core：**

|  |
| --- |
| 与内核相关的核心代码，包括线程管理，内存管理，中断管理，模块管理，驱动管理等。  由于这里的代码不加入libkernel.so库，故增加了Environment类，用来导出相关接口。 |

**filesys：**

|  |
| --- |
| 文件系统相关目录，目前已规划好接入更多文件系统的框架，但还需继续打磨。  文件系统目前只适配了FAT，还不完善，后面会继续更改。 |

**sync：**

|  |
| --- |
| 这里是用来存放线程安全的相关目录，目前只简单实现了互斥锁，自旋锁，信号量。  poll，epoll，wait之类的需要安排时间来实现，但优先级会低一点。 |

**utilities：**

|  |
| --- |
| 这里是存放一些工具类的目录，比如解析器之类的，主要用来辅助处理文本文件。 |

## vk.library: C运行库代码

## vk.network: 网络层代码

## vk.scripts: 工具脚本

## vk.build：编译输出文件夹

# 执行流程

## 启动加载内核并跳转执行

启动代码也没那么复杂，只需要把存储在扇区1及之后的2879个扇区内的内核代码，读取到0x10000之后的sram空间里，然后跳转到该地址执行就行了。后续会由专用bootloader接管。

以x86为例：

bios启动模式下，以硬盘第一个扇区为启动扇区，结尾标志为0xaa55，将从这开始读取代码执行。目前只需要在这个扇区里面完成内核读取和跳转即可。

启动进入16位实模式->读取内核代码到指定sram位置->切换到32位保护模式->重新设置数据段和栈->跳转到内核。

这部分代码使用AT&T汇编代码编写，可以使用gcc编译。

汇编代码在village内核的占比非常少，能用C/C++写绝不使用汇编。

启动代码文件：vk.hardware/x86/bios/cpu/core/boot.s

|  |
| --- |
| .org 0  .code16  .section ".text", "ax"  .set estack16, 0x9000  .set estack32, 0x2000000  .set appBaseAddr, 0x100000  .set appBaseSector, 1  .set appSectors, 2879  .global \_start  \_start:  movw %cs, %ax  movw %ax, %ds  movw %ax, %es  movw %ax, %ss  movw $estack16, %bp  movw %bp, %sp  call DisplayMsg  call ReadApplication  call ChangeVideoMode  call SwitchToProtectedMode  jmp .  # Display boot message  DisplayMsg:  pusha  movw $0x0600, %ax # Clear screen  movw $0x0700, %bx # Page 0, white on black  movw $0x00, %cx # left: (0, 0)  movw $0x184f, %dx # right: (80, 50)  int $0x10 # Display interrupt  movw $0x0, %ax # Reset es  movw %ax, %es  movw $diskBootMsg, %ax # Set the display msg address  movw %ax, %bp  movw $0x1301, %ax # Display string  movw $0x0007, %bx # Page 0, Red on black  movw $26, %cx # String length  movw $0, %dx # Show in where, dh: row dl: col  int $0x10 # Display interrupt  popa  ret  diskBootMsg: .asciz "Booting from Hard Disk..."  # Loading application from disk  ReadApplication:  movw $appSectors, %cx  movl $appBaseAddr, %ebx  movl $appBaseSector, %esi  \_ReadAppData:  call ReadFromDisk  addl $1, %esi  addl $512, %ebx  loop \_ReadAppData  ret  # Read data from disk  ReadFromDisk:  pushl %ebx  pusha  movw $0x1f2, %dx # 0x1f2  movb $1, %al # read one sector  out %al, %dx  inc %dx # 0x1f3  movl %esi, %eax  out %al, %dx  inc %dx # 0x1f4  movb %ah, %al  out %al, %dx  inc %dx # 0x1f5  shrl $16, %eax  out %al, %dx  inc %dx # 0x1f6  movb $0xe0, %al # LBA28 mode  orb %ah, %al # LBA address 27 ~ 24  out %al, %dx  inc %dx # 0x1f7  movb $0x20, %al # read cmd  out %al, %dx  \_Wait1:  in %dx, %al  test $0x80, %al  jne \_Wait1  \_Wait2:  in %dx, %al  test $0x08, %al  je \_Wait2  movw $256, %cx  movw $0x1f0, %dx  \_Readw:  in %dx, %ax  movw %ax, (%ebx)  addl $2, %ebx  loop \_Readw  popa  popl %ebx  ret  # Change video mode: 320x200 256color  ChangeVideoMode:  movw $0x13, %ax  int $0x10  ret  # GDT start label  gdtStart:  # the GDT starts with a null 8-byte  .long 0x0 # 4 byte  .long 0x0 # 4 byte  # GDT for code segment. base = 0x00000000, length = 0xfffff for flags  gdtCode:  .word 0xffff # segment length, bits 0-15  .word 0x0 # segment base, bits 0-15  .byte 0x0 # segment base, bits 16-23  .byte 0x9a # 10011010b # flags (8 bits)  .byte 0xcf # 11001111b # flags (4 bits) + segment length, bits 16-19  .byte 0x0 # segment base, bits 24-31  # GDT for data segment. base and length identical to code segment some flags changed again  gdtData:  .word 0xffff  .word 0x0  .byte 0x0  .byte 0x92 # 10010010b  .byte 0xcf # 11001111b  .byte 0x0  # GDT end label  gdtEnd:  # GDT descriptor  gdtDescriptor:  .word gdtEnd - gdtStart - 1 # size (16 bit), always one less of its true size  .long gdtStart # address (32 bit)  # define some constants for later use  codeSeg = gdtCode - gdtStart  dataSeg = gdtData - gdtStart  # Switch to protected mode  SwitchToProtectedMode:  cli # disable interrupts  lgdt gdtDescriptor # load the GDT descriptor  mov %cr0, %eax  or $0x1, %eax # set 32-bit mode bit in cr0  mov %eax, %cr0  ljmp $codeSeg, $Setup # far jump by using a different segment  # Setup segment, stack and goto bootloader  .code32  Setup:  movw $dataSeg, %ax # update segment  movw %ax, %ds  movw %ax, %ss  movw %ax, %es  movw %ax, %fs  movw %ax, %gs    movl $estack32, %ebp # update stack  movl %ebp, %esp    jmp \*(appBaseAddr) # jmp to application  jmp .  # boot section end  bootSectionEnd:  .org 510  .word 0xaa55 # Magic word |

## 初始化段数据和构造函数

跳转到内核之后：对数据段进行初始化->对构造函数进行初始化->跳转到main()。

其中对构造函数进行初始化时，也会把编译进内核的模块注册到对应的存储列表里。

数据初始化代码：vk.hardware/x86/bios/cpu/core/crt0.s

|  |
| --- |
| //###########################################################################  // crt0.s  // Low level file that manages system boot entry  //  // $Copyright: Copyright (C) village  //###########################################################################  .code32  .global \_start  /\* start address for the initialization values of the .data section.  defined in linker script \*/  .long \_sidata  /\* start address for the .data section. defined in linker script \*/  .long \_sdata  /\* end address for the .data section. defined in linker script \*/  .long \_edata  /\* start address for the .bss section. defined in linker script \*/  .long \_sbss  /\* end address for the .bss section. defined in linker script \*/  .long \_ebss  /\* stack used for SystemInit\_ExtMemCtl; always internal RAM used \*/  .section .text.\_start  .weak \_start  .type \_start, %function  \_start:  /\* Copy the data segment initializers from flash to RAM \*/  movl $\_sidata, %esi  movl $\_sdata, %ebx  jmp LoopCopyDataInit  CopyDataInit:  movl (%esi), %eax  movl %eax, (%ebx)  addl $4, %esi  addl $4, %ebx  LoopCopyDataInit:  cmpl $\_edata, %ebx  jne CopyDataInit  /\* Zero fill the bss segment. \*/  movl $\_sbss, %ebx  jmp LoopFillZeroboss  FillZerobss:  movl $0, (%ebx)  addl $4, %ebx  LoopFillZeroboss:  cmpl $\_ebss, %ebx  jne FillZerobss  /\* Call static constructors \*/  call \_\_libc\_init\_array  /\* Call the application's entry point.\*/  call main  jmp .  .size \_start, .-\_start |

## 跳转到main函数

执行完数据初始化之后跳转到main函数。

主函数文件vk.kernel/Start.cpp

|  |
| --- |
| ///Main entry function  int main(void)  {  System::Initialize();  System::ConfigureMPU();  System::ConfigurePower();  System::ConfigureClock();  Kernel::Initialize();  Kernel::UpdateParams();  Kernel::Execute();  return \_ERR;  } |

## 初始化芯片时钟

System负责初始化硬件相关资源（在vk.kernel/arch目录）。

System会初始化时钟，供电等硬件模块。

vk.kernel/arch/x86/bios/src/System.cpp

|  |
| --- |
| ///Configure the SysTick timer  void System::Initialize(void)  {  //Get the PIT value: hardware clock at 1193182 Hz  uint32\_t freq = 1000; //1000hz, 1ms  uint32\_t divider = 1193182 / freq;  uint8\_t low = low\_8(divider);  uint8\_t high = high\_8(divider);  //Send the command  PortByteOut(TIMER\_CMD, 0x36); //Command port  PortByteOut(TIMER\_CH0, low);  PortByteOut(TIMER\_CH0, high);  //Set interrupt handler  interrupt.SetISR(IRQ\_Systick, union\_cast<Function>(&System::SysTickCounter));  } |

## 初始化内核模块

Kernel负责初始化软件相关资源（在vk.kernel/core目录）。

Kernel则初始化内存，中断，异常，线程，调度，设备，模块。

vk.kernel/core/src/Kernel.cpp

|  |
| --- |
| /// @brief Kernel Execute  void Kernel::Initialize()  {  isReady = false;  //Initialize memory  memory.Initialize();  //Initialize interrupt  interrupt.Initialize();  //Initialize exception  exception.Initialize();  //Initialize thread  thread.Initialize();  //Initialize scheduler  scheduler.Initialize();  //Initialize device  device.Initialize();  //Initialize modular  modular.Initialize();  isReady = true;  } |

## 挂载文件系统

初始化modular时，如果启用了文件系统，则会挂载文件系统。

vk.kernel/filesys/core/src/FileSystem.cpp

|  |
| --- |
| /// @brief File system initialize  void FileSystem::Initialize()  {  for (FileSys\* fs = fileSys.Begin(); !fileSys.IsEnd(); fs = fileSys.Next())  {  fs->Setup();  }  //Mount root node "/"  mounts.Add(new MountNode((char\*)"/", (char\*)"/media/VILLAGE OS", 0755));  } |

## 加载共享库及动态模块

初始化modular时，如果启用动态加载模块功能，则会加载共享库以及动态模块。

vk.kernel/core/src/Loader.cpp

|  |
| --- |
| /// @brief Loader initialize  void Loader::Initialize()  {  Loading(\_Load\_Lib, "/libraries/\_load\_.rc");  Loading(\_Load\_Mod, "/modules/\_load\_.rc");  }  /// @brief Loader load  /// @param filename rc file path  void Loader::Loading(int type, const char\* filename)  {  RcParser\* parser = new RcParser(filename);  List<char\*>& runcmds = parser->GetRunCmds();  for (runcmds.End(); !runcmds.IsBegin(); runcmds.Prev())  {  if (\_Load\_Lib == type)  {  if (!libraryTool.Install(runcmds.Item())) break;  }  else if (\_Load\_Mod == type)  {  if (!moduleTool.Install(runcmds.Item())) break;  }  }  parser->Release();  delete parser;  } |

## 注册线程任务

各项模块初始化完成之后，则开始把模块注册进线程任务列表里面。

vk.kernel/core/src/Modular.cpp

|  |
| --- |
| void Modular::Execute()  {  status = \_StartExecute;  for (Module\* module = modules.Begin(); !modules.IsEnd(); module = modules.Next())  {  module->SetPid(thread.CreateTask(module->GetName(), (Method)&Modular::ModuleHandler, this, (void\*)module));  }  status = \_EndedExecute;  } |

## 开始调度任务

模块任务注册完成之后则开始进行任务调度了，滴答定时器触发任务调度。 SysTickHandler->PendSVHandler，保存当前任务栈及sp现场，切换下个任务，还原下个任务栈及sp。

vk.kernel/arch/x86/bios/src/Scheduler.cpp

|  |
| --- |
| //###########################################################################  // Scheduler.cpp  // Definitions of the functions that manage thread scheduler  //  // $Copyright: Copyright (C) village  //###########################################################################  #include "Scheduler.h"  #include "Interrupt.h"  #include "Thread.h"  #include "System.h"  /// @brief Constructor  Scheduler::Scheduler()  :isStartSchedule(false)  {  }  /// @brief Fini constructor  Scheduler::~Scheduler()  {  }  /// @brief Singleton Instance  /// @return Scheduler instance  Scheduler& Scheduler::Instance()  {  static Scheduler instance;  return instance;  }  /// @brief Definitions scheduler  Scheduler& scheduler = Scheduler::Instance();  /// @brief Scheduler initialize  void Scheduler::Initialize()  {  //Set the PendSV interrupt handler  interrupt.SetISR(IRQ\_PendSV, union\_cast<Function>(&Scheduler::PendSVHandler), (char\*)this);  //Append the systick interrupt handler  interrupt.AppendISR(IRQ\_Systick, union\_cast<Function>(&Scheduler::SysTickHandler), (char\*)this);  }  /// @brief Start scheduler  void Scheduler::Execute()  {  //Clear start schedule flag  isStartSchedule = false;  //Get frist task psp  uint32\_t psp = thread.GetTaskPSP();  //Set frist task esp  \_\_asm volatile("movl %0, %%esp" : "=r"(psp));  //Set start schedule flag  isStartSchedule = true;  //Set interrupt flag  \_\_asm volatile("sti");  //Execute thread  thread.Execute();  }  /// @brief Rescheduler task  /// @param access scheduler access  void Scheduler::Rescheduler(Scheduler::Access access)  {  if (false == isStartSchedule) return;  // trigger PendSV directly  \_\_asm volatile("int $31");  }  /// @brief PendSV handler  void \_\_attribute\_\_((naked)) Scheduler::PendSVHandler()  {  uint32\_t psp = 0;  //Push old task registers  \_\_asm volatile("pushl %ebp");  \_\_asm volatile("pushl %ebx");  \_\_asm volatile("pushl %esi");  \_\_asm volatile("pushl %edi");  \_\_asm volatile("movl %%esp, %0" : "=r"(psp));  //Save old task psp  thread.SaveTaskPSP(psp);  //Select next task  thread.SelectNextTask();  //Get new task psp  psp = thread.GetTaskPSP();  //Set new task esp  \_\_asm volatile("movl %0, %%esp" : "=r"(psp));  //Pop new task registers  \_\_asm volatile("popl %edi");  \_\_asm volatile("popl %esi");  \_\_asm volatile("popl %ebx");  \_\_asm volatile("popl %ebp");  \_\_asm volatile("sti");  \_\_asm volatile("ret");  }  /// @brief SysTick handler  void Scheduler::SysTickHandler()  {  scheduler.Rescheduler(Scheduler::Privileged);  } |

## 加载第一个应用程序

如果启用了文件系统及允许动态加载模块功能，调度到Loader时则会使用Executor加载并执行第一个应用程序“taichi.exec”

vk.kernel/core/src/Loader.cpp

|  |
| --- |
| /// @brief Loader execute  void Loader::Execute()  {  //Execute the first application of the village  executor.Run("/applications/taichi.exec");  } |

vk.kernel/binutils/src/Executor.cpp

|  |
| --- |
| /// @brief Executor Initialize  /// @param path elf file path  /// @param argv running argv  /// @return result  int Executor::Run(const char\* path, int argc, char\* argv[])  {  //Set argc and argv  this->argc = argc;  this->argv = argv;  //Load, parser and execute elf file  if (elf.Load(path) != Result::\_OK) return \_ERR;    //Create a sandboxed thread to run the app  int pid = thread.CreateTask(path, (Method)&Executor::Sandbox, this);  //Wait for task done  return thread.WaitForTask(pid);  }  /// @brief Executor execute app  void Executor::Sandbox()  {  elf.FillBssZero();  elf.InitArray();  elf.Execute(NULL, argc, argv);  elf.FiniArray();  elf.Exit();  } |

关于解析并加载elf文件，其实也不是很复杂，把elf通过文件系统读取出来，再按照数据读取内容到sram，并重定位相关entry，并执行初始化则可以。难点在加载内容到sram和重定位entry，重点重定位。相关代码在vk.kernel/binutils/src/ElfLoader.cpp，代码太多这里不粘贴了。

还有elf文件类型也得是DYN (Position-Independent Executable file)，如何生成该类型文件与ld flags相关，以下ld flags可以参考。

vk.hardware/x86/bios/cpu/Makefile

|  |
| --- |
| # application ld flags  ifeq ($(CONFIG\_VKLIBC), y)  APPLDFLAGS += $(MCU) -ffreestanding -nostdlib  else  APPLDFLAGS += $(MCU) -specs=nosys.specs -lc -lm -lnosys  endif  APPLDFLAGS += -Wl,-Map=$(@:.exec=.map),--cref  APPLDFLAGS += -Wl,--gc-sections  APPLDFLAGS += -Wl,--no-warn-rwx-segment  APPLDFLAGS += -Wl,--unresolved-symbols=ignore-in-shared-libs  APPLDFLAGS += -Wl,-m,elf\_i386  APPLDFLAGS += -pie -e main |

## 应用层运行控制台

内核调用并执行taichi.exec之后，则来到了应用层，目前taichi.exec只是简单的调用了console.exec。

后面会随着开发进度进行完善。

vk.application/taichi/src/Taichi.cpp

|  |
| --- |
| //###########################################################################  // Taichi.cpp  // The overall framework of the taichi  //  // $Copyright: Copyright (C) village  //###########################################################################  #include "Taichi.h"  /// @brief Constructor  Taichi::Taichi()  {  }  /// @brief Deconstructor  Taichi::~Taichi()  {  }  /// @brief Initialize  void Taichi::Initialize()  {  }  /// @brief Execute  void Taichi::Execute()  {  executor.Run("/applications/console.exec");  }  /// @brief main  extern "C" int main(void)  {  Taichi taichi;  taichi.Initialize();  taichi.Execute();  return 0;  } |

# 适配新平台

# 编写驱动模块

# 编写应用程序