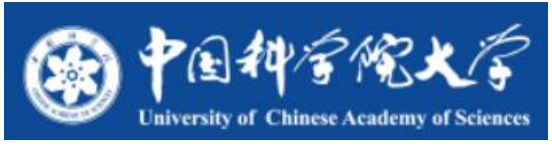
ISCAS
中国科学院软件研究所
Institute of Software,Chinese Academy of Sciences
中国科学院大学
University of Chinese Academy of Sciences

系统初始化

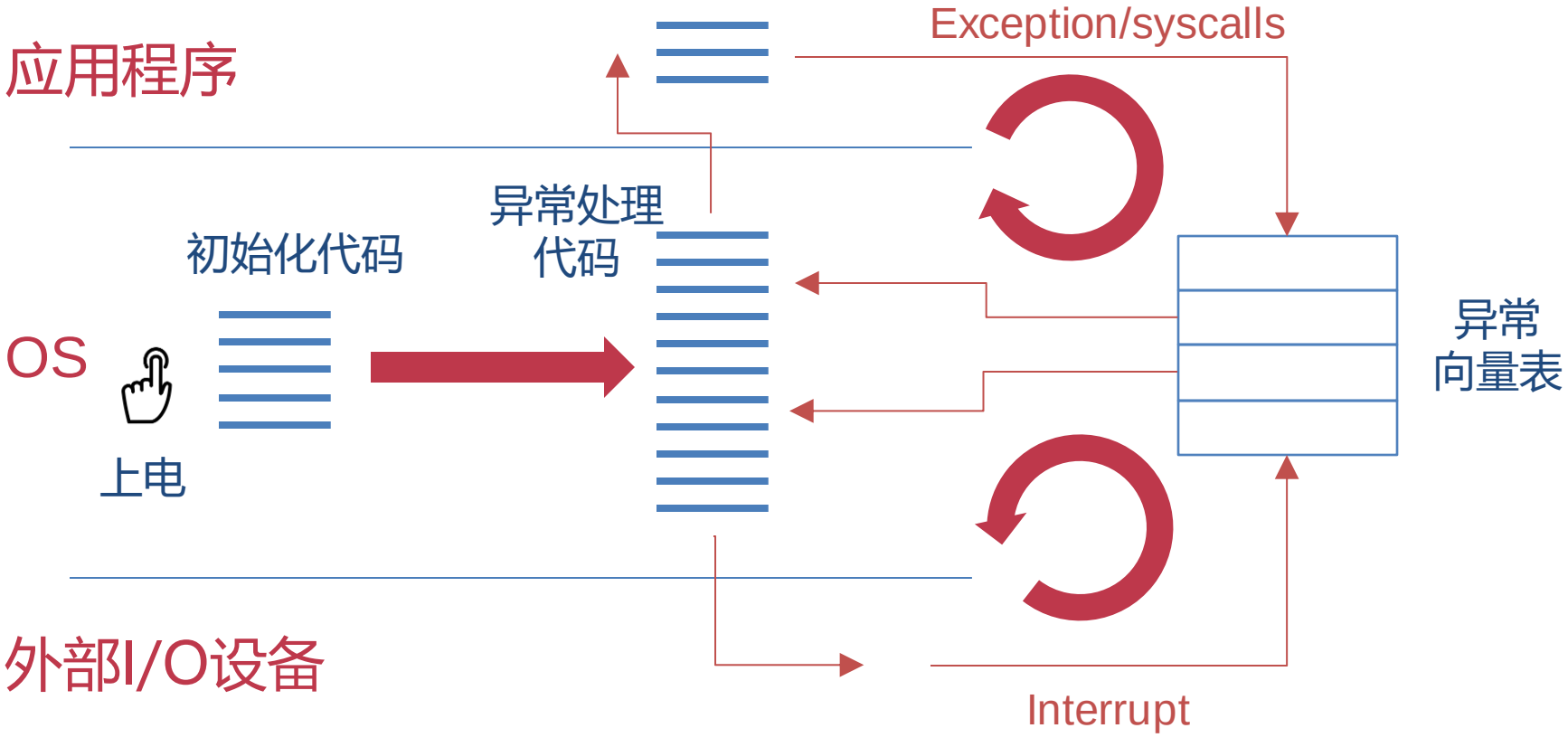
郑晨

改编声明

- 本课程教学及PPT内容基于**上海交通大学并行与分布式系统研究所**发布的操作系统课程修改，原课程官网：
  - https://ipads.se.sjtu.edu.cn/courses/os/index.shtml
- 本课程修改人为**中国科学院软件研究所**，用于国科大操作系统课程教学。



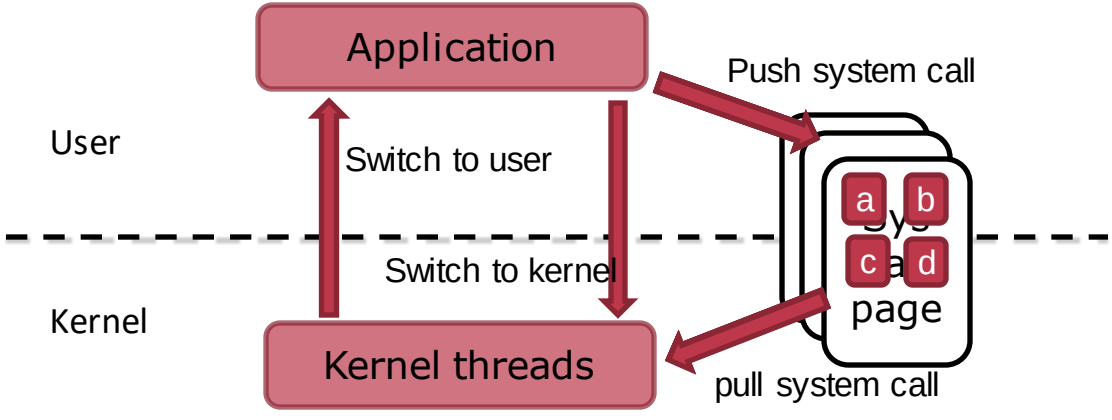
Review: 操作系统的执行流（"双循环"）



Review: 并非所有syscall都会下陷—vDSO

- **内核定义**
  - 在编译时作为内核的一部分
- **用户态运行**
  - 将gettimeofday的代码加载到一块与应用共享的内存页
  - 这个页称为：vDSO
    - Virtual Dynamic Shared Object
  - Time 的值同样映射到用户态空间（只读）
    - 只有在内核态才能更新这个值
- **Q：和以前的gettimeofday相比有什么区别?**

Review: 并非所有syscall都会下陷—FlexSC



计算机启动

启动流程：从上电到等待用户输入

按下开关

- 硬件环境（除CPU）初始化
- 加载内核代码到内存中

bootloader

内核的第一行代码

- 初始化MMU、外设（如网卡、硬盘）
- 若有多核，则初始化其他CPU核

kernel

第一个用户态应用的第一行代码

- 加载shell的二进制

init & loader

shell的第一行代码

- 等待键盘输入

shell

shell等待用户输入

启动：一个复杂的过程

- **为什么用“boot”表示启动?**
  - Boot源自bootstrap – 鞋带
  - "pull oneself up by one's bootstraps"
    - 通过拽鞋带，把自己拉起来

The computer term bootstrap began as a metaphor in the 1950s. In computers, pressing a bootstrap button caused a hardwired program to read a bootstrap program from an input unit. The computer would then execute the bootstrap program, which caused it to read more program instructions. It became a self-sustaining process that proceeded without external help from manually entered instructions. As a computing term, bootstrap has been used since at least 1953.

加电硬件初始化过程

- **加电自检**
  - 基本过程

    初始化BIOS

    检查CPU寄存器

    检查BIOS代码的完整性

    检查DMA、timer、interrupt controller

    检查系统内存

    检查系统总线和外部设备

    跳转到下一级BIOS(如VGA-BIOS)执行并返回

    识别可以启动的设备(CD-ROM?USB?HDD?)
  - 谁来执行这些检查?

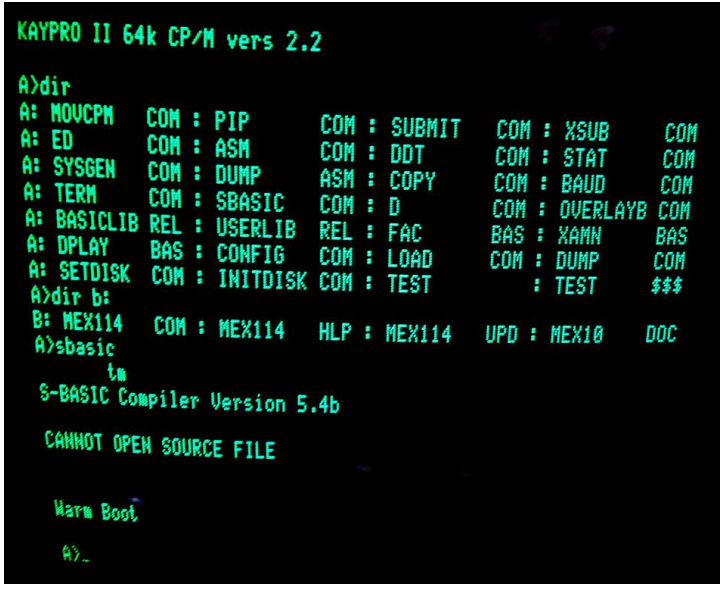
内核启动的2个主要任务

- **任务-1：配置页表并开启虚拟内存机制，允许使用虚拟地址**
  - 页表究竟该如何具体配置?
  - 难点：开启地址翻译的前一行指令使用物理地址，开启后立即使用虚拟地址，前后如何衔接?

- **任务-2：配置异常向量表并打开中断，允许"双循环"**
  - 异常向量表如何配置?
  - 打开后，异常处理的指令流如何流动?

BIOS的作用和结构

CP/M操作系统



搭载了BIOS的CP/M操作系统
诞生于 1975年

Gary Kildall

BIOS大爆发


BIOS SETUP UTILITY
Main Advanced Power Boot Tools Exit
System Time [16:29:59]
System Date [Tue 09/18/2007]
Legacy Diskette A [Disabled]
Language [English]
▸ SATA 1 : [ST3120026AS]
▸ SATA 2 : [Not Detected]
▸ SATA 3 : [Not Detected]
▸ SATA 4 : [Not Detected]
▸ SATA 5 : [Not Detected]
▸ SATA 6 : [ST3120026AS]
▸ SATA Configuration
▸ System Information
Use [ENTER], [TAB] or [SHIFT-TAB] to select a field.
Use [+] or [-] to configure system Time.
↔ Select Screen
↑↓ Select Item
+- Change Field
Tab Select Field
F1 General Help
F10 Save and Exit
ESC Exit
v02.58 (C)Copyright 1985-2007, American Megatrends, Inc.

BIOS的作用和结构

- **BIOS**
  - Basic Input/Output System，基本输入输出系统

BIOS的作用和结构

- **BIOS的作用**
  - 在计算机开机时对系统各组件进行检查
  - 加载引导程序或操作系统
  - 向操作系统提供系统配置信息
  - 向操作系统提供硬件访问接口，向操作系统隐藏硬件的变化
    - 现代操作系统会忽略BIOS提供的抽象层并直接访问硬件

BIOS的功能与位置

- **BIOS中主要存放以下程序段。**
- **（1）自诊断程序**

  通过读取CMOSRAM中的内容，识别硬件配置，并对其进行自检和初始化。
- **（2）CMOS设置程序**

  引导过程中，用特殊热键启动，进行设置后，存入CMOS RAM中。
- **（3）系统自检装载程序**

  在自检成功后，将磁盘0磁道0扇区上的引导程序装入内存，让其运行以装入系统。
- **（4）主要I/0设备的驱动程序和中断服务**

BIOS的作用和结构

- **BIOS的结构**
  - BIOS的物理结构

存储BIOS代码

  - BIOS代码的结构

存储BIOS数据，包括各种系统配置

项目名称	原始大小	压缩大小	原始文件名
0. System BIOS	20000h(128.00K)	13C31h(79.05K)	865IDC19.BIN
1. XGROUP CODE	0D960h(54.34K)	09806h(38.01K)	awardext.rom
2. CPU micro code	04000h(16.00K)	03FA2h(15.91K)	CPUCODE.BIN
3. ACPI table	045C1h(17.44K)	01A7Dh(6.62K)	ACPITBL.BIN
4. EPA LOGO	0168Ch(5.64K)	002AAh(0.67K)	AwardBmp.bmp
5. YGROUP ROM	05D00h(23.25K)	03E56h(15.58K)	awardeyt.rom
6. GROUP ROM [0]	05360h(20.84K)	024B5h(9.18K)	_EN_CODE.BIN
7. VGA ROM [1]	0C000h(48.00K)	06B05h(26.75K)	SDG_2831.DAT
8. GROUP ROM [5]	004F0h(1.23K)	002A4h(0.66K)	SDG_2831.VBT
9. Flash ROM	0A00Ch(40.01K)	05777h(21.87K)	AWDFLASH.EXE
10. PCI ROM [A]	0C000h(48.00K)	05DFCh(23.50K)	4212.BIN

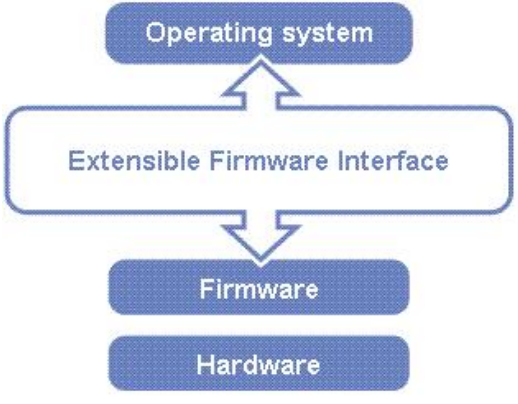
EFI/UEFI简介

BIOS → UEFI

70s ~ 90s

EFI / UEFI

- **Intel提出来EFI取代BIOS interface**
  - EFI（Extensible Firmware Interface）
- **2005年，Intel再次提出UEFI取代EFI**
  - UEFI（Unified Extensible Firmware Interface）

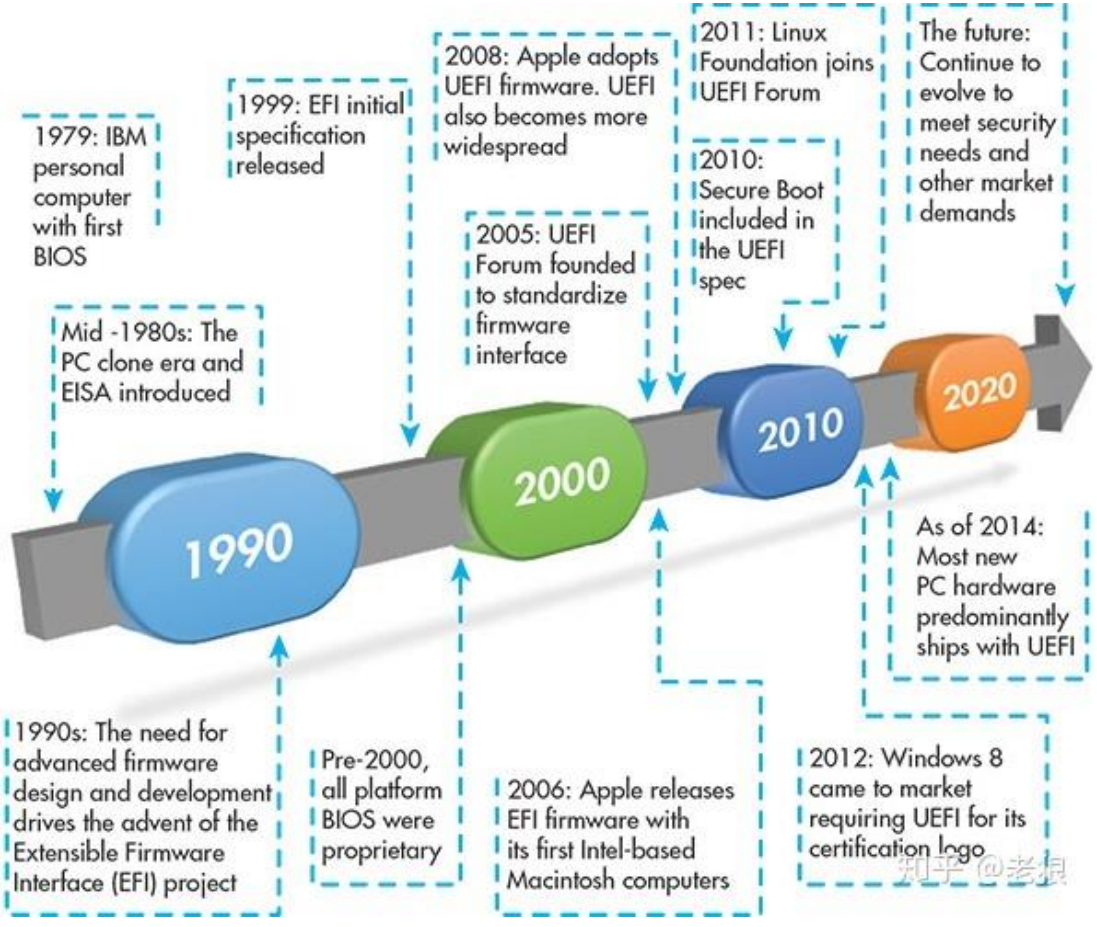


安腾处理器

经典BIOS：汇编

新BIOS：C语言

BIOS 发展史


1979: IBM personal computer with first BIOS
1999: EFI initial specification released
2008: Apple adopts UEFI firmware. UEFI also becomes more widespread
2011: Linux Foundation joins UEFI Forum
The future: Continue to evolve to meet security needs and other market demands
2010: Secure Boot included in the UEFI spec
2005: UEFI Forum founded to standardize firmware interface
Mid -1980s: The PC clone era and EISA introduced
1990
2000
2010
2020
As of 2014: Most new PC hardware predominantly ships with UEFI
1990s: The need for advanced firmware design and development drives the advent of the Extensible Firmware Interface (EFI) project
Pre-2000, all platform BIOS were proprietary
2006: Apple releases EFI firmware with its first Intel-based Macintosh computers
2012: Windows 8 came to market requiring UEFI for its certification logo

EFI/UEFI简介

- **为何需要EFI/UEFI**
  - EFI是用模块化，C语言，动态链接的形式构建的系统，较BIOS而言更易于实现，容错和纠错特性更强，缩短了研发时间

UEFI和传统BIOS的区别

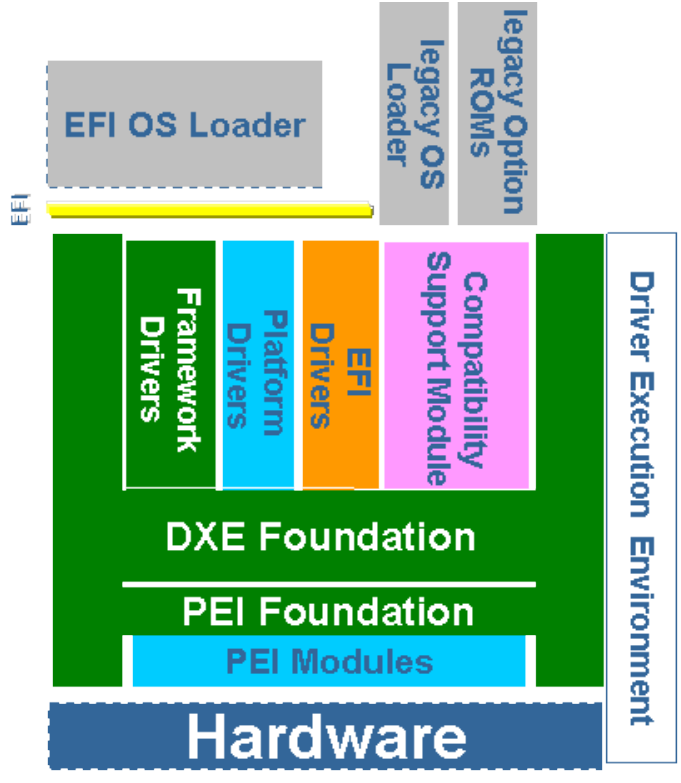
- BIOS三大任务：
  - 1.初始化硬件
  - 2.提供硬件的软件抽象
  - 3.启动操作系统
- UEFI三大优势
  - 1.标准接口
  - 2.开放统一
  - 3.开源

UEFI

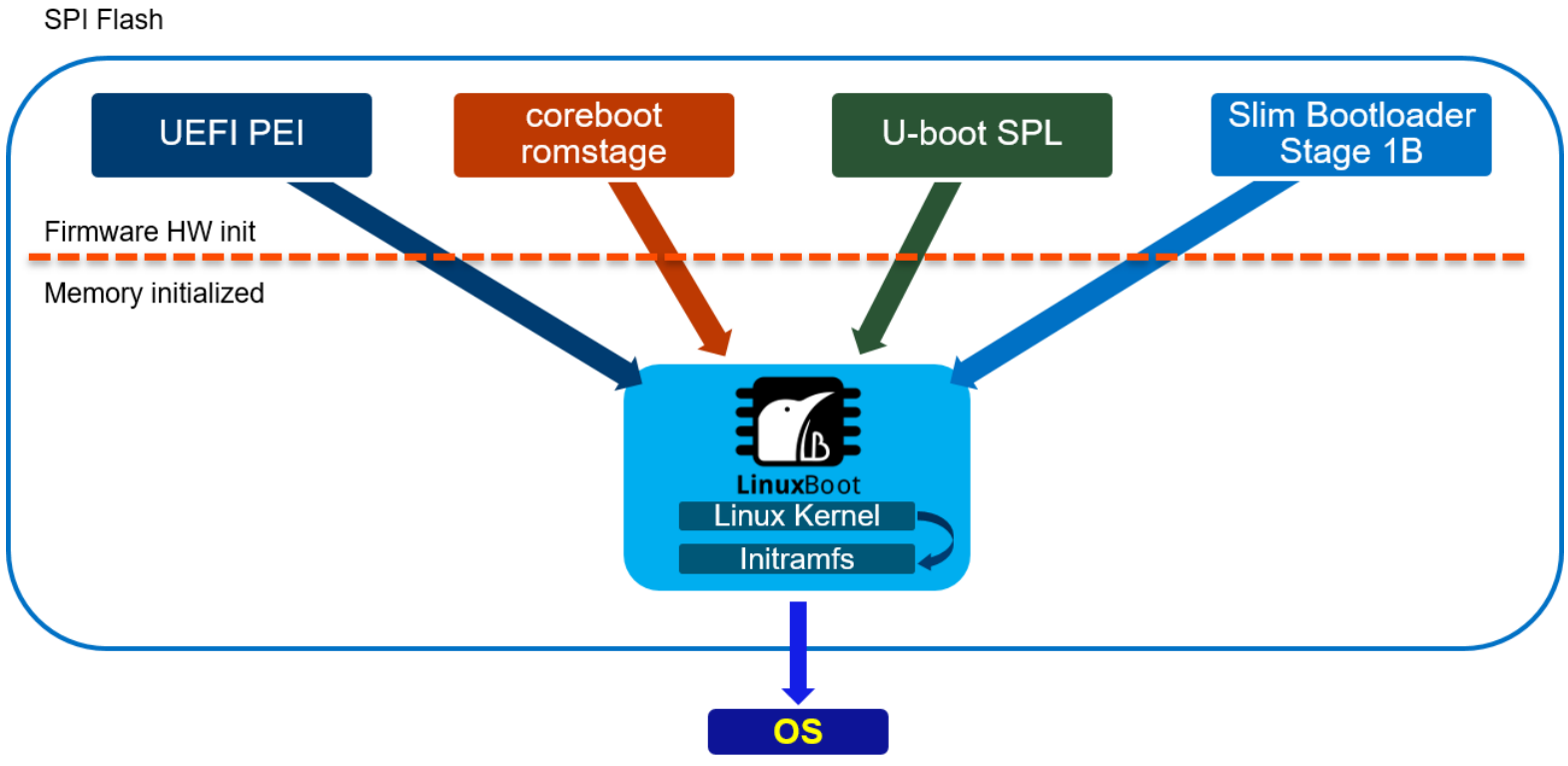
- **为何需要EFI/UEFI**
  - EFI运行于32位或64位模式，突破传统16位代码的寻址能力。而BIOS的硬件服务程序都以16位代码的形式存在，这就给运行于增强模式的操作系统访问其服务造成了困难
  - 它利用加载EFI驱动的形式，识别及操作硬件
  - EFI系统下的驱动并不是由可以直接运行在CPU上的代码组成的，而是用EFI Byte Code编写而成的。这是一组专用于EFI驱动的虚拟机器语言，必须在EFI驱动运行环境下被解释运行。这就保证了充分的向下兼容性

UEFI框架结构

- EFI/UEFI的结构



当前进展——LinuxBoot



GRUB引导程序

Grub引导程序简介

- **引导程序简介**
  - 什么是bootloader?
    - 开机时，引导操作系统启动的程序
    - BIOS在完成硬件检测和资源分配后，将硬盘MBR中的bootloader读到系统的RAM中，然后将控制权交给bootloader
    - bootloader的主要任务就是将操作系统内核从硬盘加载到RAM中，然后跳转到内核的入口点去执行，即启动操作系统
    - 常见的bootloader
      - Grub, isolinux, uboot, openSBI，ntldr(用于启动Windows系统)，Linuxboot
  - 为何需要bootloader?
    - 操作系统需要被加载到内存中正确的位置
    - 需要为操作系统提供启动参数，以实现定制化启动

Linux启动流程

MBR

- **MBR**

boot loader	Disk Partition Table				magic number
code	1	2	3	4	55AAH
MBR					

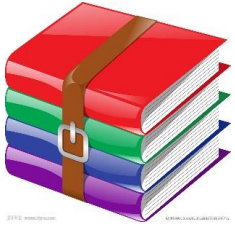
- **GPT**

PMBR	Partition Table						Partition					Table Backup	GPT Backup
MBR	GPT HDR	1	2	3	4	...	1	2	3	4	...	Partition Table Backup	GPT HDR Backup
LBA0	LBA1	LBA2				LBA3~ LBA34	LBA35~LBA-35					LBA-2~ LBA-34	LBA-1

Grub引导程序简介

- **Grub简介**
  - Linux上最常用的bootloader
    - GNU GRUB是一个来自GNU项目的启动引导程序。GRUB允许用户可以在计算机内同时拥有多个操作系统，并在计算机启动时选择希望运行的操作系统
    - GRUB可通过链式引导来引导Windows系统
    - 支持所有的Linux文件系统，也支持Windows的FAT和NTFS文件系统
    - 支持图形界面，可定制启动菜单和背景图片，支持鼠标
    - 拥有丰富的终端命令，用户可以查看硬盘分区的细节，修改分区设置，临时重新映射磁盘顺序，从任何用户定义的配置文件启动

Kernel



压缩的Kernel

解压后的Kernel

根文件系统rootfs

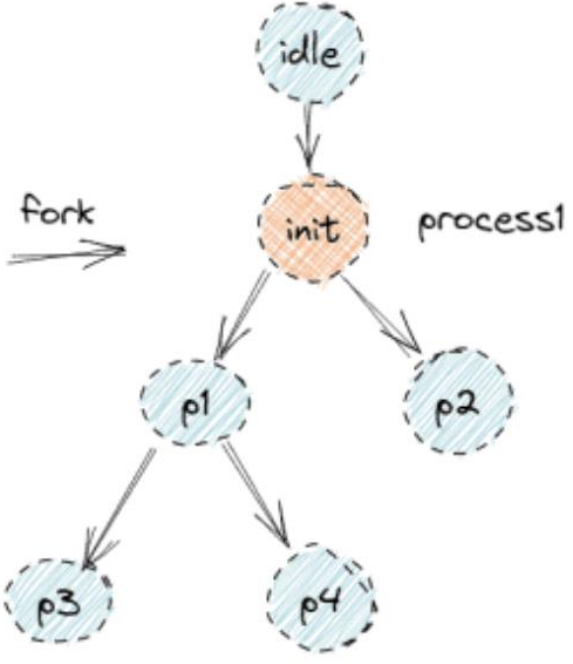
第一个程序init

Init

- init，初始化，顾名思义，该程序就是进行OS初始化操作，实际上是根据/etc/inittab(定义了系统默认运行级别)设定的动作进行脚本的执行，第一个被执行的脚本为/etc/rc.d/rc.sysinit，这个是真正的OS初始化脚本。
  - 1、激活udev和selinux；
  - 2、根据/etc/sysctl.conf文件，来设定内核参数；
  - 3、设定系统时钟；
  - 4、装载硬盘映射；
  - 5、启用交换分区；
  - 6、设置主机名；
  - 7、根文件系统检测，并以读写方式重新挂载根文件系统；
  - 8、激活RAID和LVM设备；
  - 9、启用磁盘配额；
  - 10、根据/etc/fstab，检查并挂载其他文件系统；
  - 11、清理过期的锁和PID文件
- 执行完后，根据配置的启动级别，执行对应目录底下的脚本，最后执行/etc/rc.d/rc.local这个脚本，至此，系统启动完成。

Init进程

- **进程1**
  - 通过fork系统调用创建
  - 第一个用户态进程
  - 所有用户态进程的父进程或先祖进程
- **主要责任:**
  - 系统startup
  - Zombie进程管理
- **重要性:**
  - 进程1的稳定性代表了系统的稳定性



Runlevel

runlevel，运行级别，不同的级别会启动的服务不一样，init会根据定义的级别去执行相应目录下的脚本，Linux的启动级别分为以下几种

0：关机模式
1：单一用户模式(直接以管理员身份进入)
2：多用户模式（无网络）
3：多用户模式（命令行）
4：保留
5：多用户模式（图形界面）
6：重启

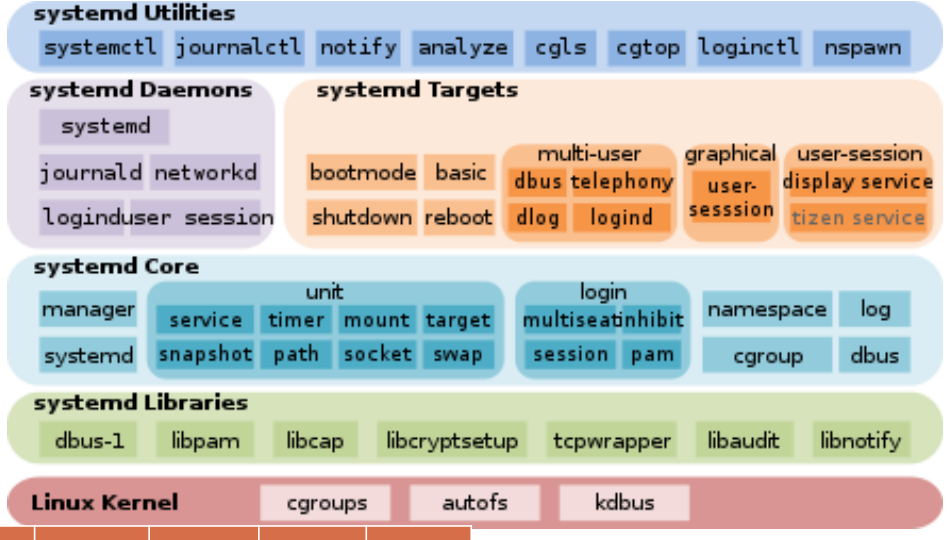
在不同的运行级别下，/etc/rc.d/rc这个脚本会分别执行不同目录下的脚本

- Run level 0 – /etc/rc.d/rc0.d/
- Run level 1 – /etc/rc.d/rc1.d/
- Run level 2 – /etc/rc.d/rc2.d/
- Run level 3 – /etc/rc.d/rc3.d/
- Run level 4 – /etc/rc.d/rc4.d/
- Run level 5 – /etc/rc.d/rc5.d/
- Run level 6 – /etc/rc.d/rc6.d/

这些目录下的脚本只有K*和S*开头的文件，K开头的文件为开机需要执行关闭的服务，S开头的文件为开机需要执行开启的服务。

当前进展——init->Systemd

- **Init缺点**
  - 启动时间长。init进程是串行启动，只有前一个进程启动完，才会启动下一个进程。
  - 启动脚本复杂。脚本需要自己处理各种情况，这往往使得脚本变得很长。



Init 软件	说明	启动管理	进程回收	服务管理	并行启动	设备管理	资源控制	日志管理
sysvinit	早期版本使用的初始化进程工具，逐渐淡出舞台	✓	✓	-	-	-	-	-
Upstart	Debian、Ubuntu 等系统使用的 initdaemon	✓	✓	✓	✓	-	-	-
systemd	提高系统的启动速度，相比传统的 System V 是一大革新，已被大多数 Linux 发行版所使用。	✓	✓	✓	✓	✓	✓	✓

当前进展——init->Systemd

Current State of Process1

- Different distributions have different process1. Up to now, there are 20+ different processe1 implementations.

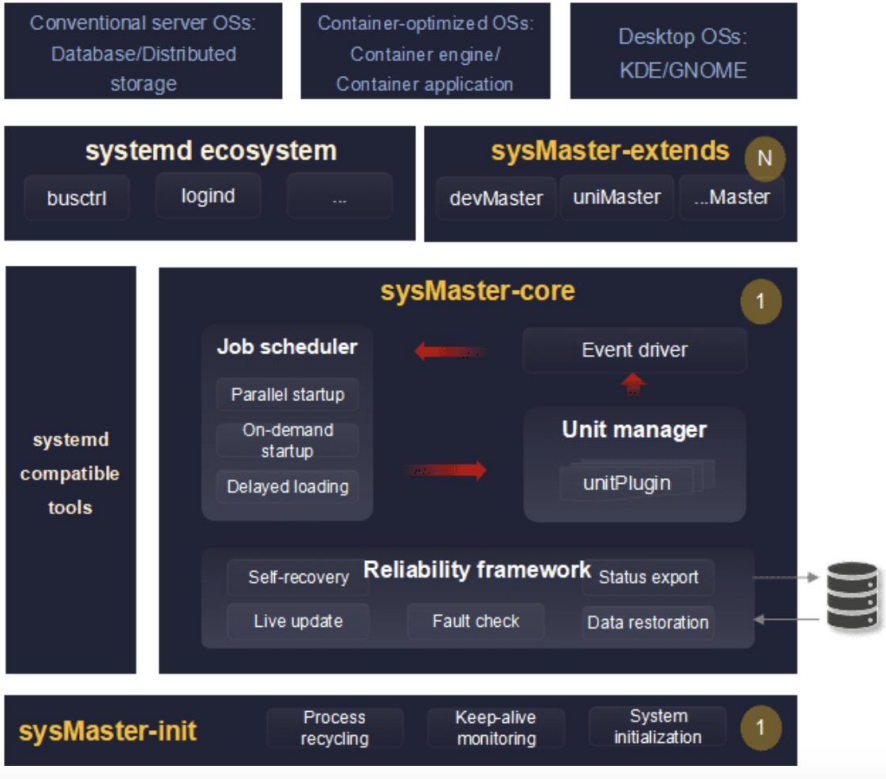
[● ◂] systemd	upstart	SysVinit	openrc	launchd automation scripts
2010' • Red Hat, openSUSE • Parallel startup • On-demand loading • De facto standard	90' • Debian, ChromeOS • Asynchronous work • Service monitoring • Extensible event-driven model	80' • PCLinuxOS, Porteus • /etc/initab • Serial startup • Startup script	• Alpine, Gentoo • start-stop-daemon • supervise-daemon	• Mac OS X 10.4 • Daemons and Agents

Process1	Description	Startup Mgmt.	Process Recycling	Service Mgmt.	Parallel Startup	Device Mgmt.	Resource Control	Log Mgmt.
SysVinit	The init process tool used in earlier versions. It gradually fades out of the stage.	✓	✓					
upstart	Formerly used by Debian 7 and Ubuntu 14. Project is in maintaince mode only	✓	✓	✓	✓			
systemd	Faster startup speed. Compared with traditional SysVinit, systemd is a major innovation and has been used by most Linux distributions.	✓	✓	✓	✓	✓	✓	✓

The demands on Process 1 varies in different OSs

当前进展——SysMaster全新的1号进程

- **SysMaster-Init**
  - Zombie reaping
  - 监控sysmaster-core
  - Less 1k LoC
- **SysMaster-core**
  - 依赖管理器
  - Hot restart，live update
- **SysMaster-extends**



内核代码加载与运行——RISC-V

第一行代码

- **常见 RISC-V 开发板Linux启动流程**
  - 板子上电后，CPU 从固定地址运行 ROM 中的代码
  - ROM 包含简单的设备驱动，从 flash 或者 SD 卡中加载 bootloader
  - 再由 bootloader 加载内核、initramfs 等到内存，跳转到 Linux 内核启动

全志 D1 的启动流程

- SPL：在 ROM 中运行，加载 U-Boot
- U-Boot: 从 0x4a00_0000 物理地址开始执行，加载 Linux
- Linux: 从 0x4200_0000 物理地址开始执行

参考：https://linux-sunxi.org/Allwinner_Nezha

Linux 内核的起始地址（链接脚本）

内核二进制从 .head.text 节开始

arch/riscv/kernel/vmlinux.lds.S

```
SECTIONS
{
        /* Beginning of code and text segment */
        . = LOAD_OFFSET;
        _start = .;
        HEAD_TEXT_SECTION
        . = ALIGN(PAGE_SIZE);
```

内核启动入口

include/asm-generic/vmlinux.lds.h

```
/* Section used for early init (in .S files) */
#define HEAD_TEXT  KEEP(*(.head.text))

#define HEAD_TEXT_SECTION
        .head.text : AT(ADDR(.head.text) - LOAD_OFFSET) {
                HEAD_TEXT
        }
```

内核运行的第一行代码

include/linux/init.h

内核的第一个字节开始就是这里的代码

```
/* For assembly routines */
#define __HEAD          .section        ".head.text","ax"
#define __INIT          .section        ".init.text","ax"
#define __FINIT         .previous
```

```
__HEAD
ENTRY(_start)
        /*
         * Image header expected by Linux boot-loaders. The image header data
         * structure is described in asm/image.h.
         * Do not modify it without modifying the structure and all bootloaders
         * that expects this header format!!
         */
#ifdef CONFIG_EFI
        /*
         * This instruction decodes to "MZ" ASCII required by UEFI.
         */
        c.li s4,-13
        j _start_kernel                 跳转到 _start_kernel
#else
        /* jump to start kernel */
        j _start_kernel
        /* reserved */
        .word 0
#endif
```

arch/riscv/kernel/head.S

```
ENTRY(_start_kernel)
        /* Mask all interrupts */
        csrw CSR_IE, zero
        csrw CSR_IP, zero


        /* Load the global pointer */
.option push
.option norelax
        la gp, __global_pointer$
.option pop

        /*
         * Disable FPU & VECTOR to detect illegal usage of
         * floating point or vector in kernel space
         */
        li t0, SR_FS_VS
        csrc CSR_STATUS, t0
```

禁用中断

加载 gp 寄存器

内核中禁用浮点和向量单元

```
#ifndef CONFIG_XIP_KERNEL
        /* Clear BSS for flat non-ELF images */
        la a3, __bss_start
        la a4, __bss_stop
        ble a4, a3, clear_bss_done
clear_bss:
        REG_S zero, (a3)
        add a3, a3, RISCV_SZPTR
        blt a3, a4, clear_bss
clear_bss_done:
#endif
        /* Save hart ID and DTB physical address */
        mv s0, a0
        mv s1, a1

        la a2, boot_cpu_hartid
        XIP_FIXUP_OFFSET a2
        REG_S a0, (a2)

        /* Initialize page tables and relocate to virtual addresses */
        la tp, init_task
        la sp, init_thread_union + THREAD_SIZE
        XIP_FIXUP_OFFSET sp
        addi sp, sp, -PT_SIZE_ON_STACK
```

将 .bss 清零

保存启动的 hartid

设置 sp 为启动时使用的栈，
tp 指向当前任务

```
        call setup_vm
#ifdef CONFIG_MMU
        la a0, early_pg_dir
        XIP_FIXUP_OFFSET a0
        call relocate_enable_mmu
#endif /* CONFIG_MMU */

        call setup_trap_vector
        /* Restore C environment */
        la tp, init_task
        la sp, init_thread_union + THREAD_SIZE
        addi sp, sp, -PT_SIZE_ON_STACK

#ifdef CONFIG_KASAN
        call kasan_early_init
#endif
        /* Start the kernel */
        call soc_early_init
        tail start_kernel
```

页表初始化

arch/riscv/mm/init.c

启动 MMU，转到高地址继续运行

重新设置正确的 sp 和 tp
（gp 在 relocate_enable_mmu 中设置）

进行一些其它组件的初始化，然后正式进入内核代码运行

页表和 MMU 初始化

内核地址映射

```
SECTIONS
{
        /* Beginning of code and text segment */
        . = LOAD_OFFSET;
        _start = .;
        HEAD_TEXT_SECTION
        . = ALIGN(PAGE_SIZE);
```

```
#include <asm/pgtable.h>
#define LOAD_OFFSET KERNEL_LINK_ADDR
```

```
#define ADDRESS_SPACE_END       (UL(-1))

#ifdef CONFIG_64BIT
/* Leave 2GB for kernel and BPF at the end of the address space */
#define KERNEL_LINK_ADDR        (ADDRESS_SPACE_END - SZ_2G + 1)
```

include/asm/pgtable.h

内核启动时从 0x4020_0000 开始，而内核代码需要在0xFFFF_FFFF_8000_0000 运行

如何配置 MMU 来完成这个切换?

页表和 MMU 初始化

- Linux 内核运行需要 MMU 启用
- 内核刚开始运行 MMU 未启用
- 需要写好一个页表将内核映射到高地址，然后启用 MMU

页表初始化

- **setup_vm 初始化启动用到的两个页表**
  - trampoline_pg_dir 启用 MMU 前后所用，映射启用 MMU 的代码到高地址
  - early_pg_dir 内核最初启动的时候所用，映射整个内核到高地址

页表初始化

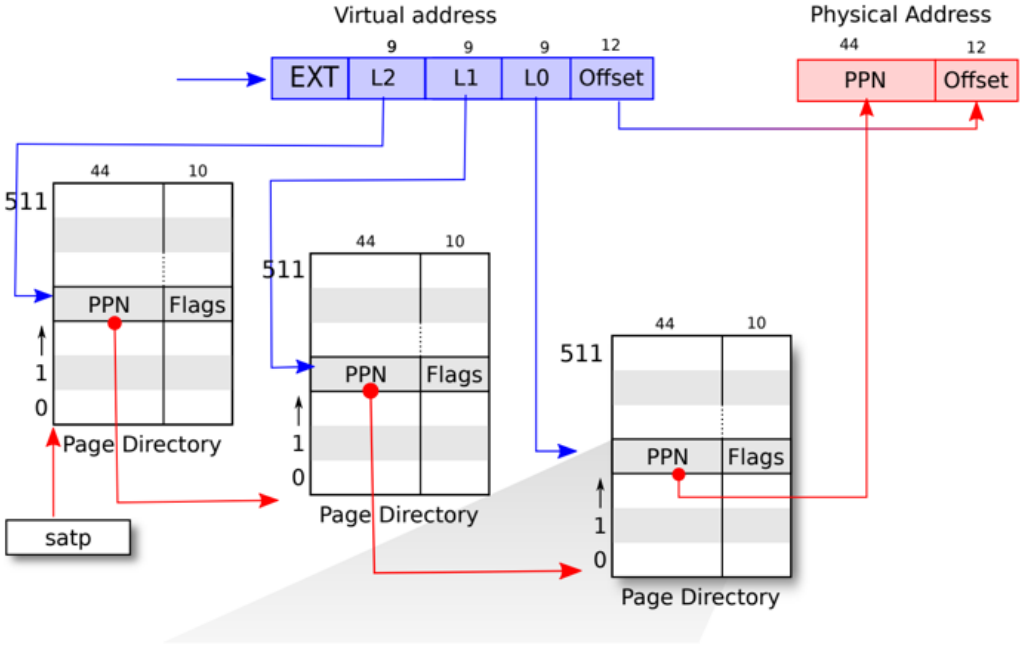
trampoine_pg_dir
用一个 2 MiB 大页映射内核开头，作为启用 MMU 用的跳板

```
asmlinkage void __init setup_vm(uintptr_t dtb_pa)
{

    /* Setup trampoline PGD and PMD */
    create_pgd_mapping(trampoline_pg_dir, kernel_map.virt_addr,
                       trampoline_pgd_next, PGDIR_SIZE, PAGE_TABLE);

    create_pmd_mapping(trampoline_pmd, kernel_map.virt_addr,
                       kernel_map.phys_addr, PMD_SIZE, PAGE_KERNEL_EXEC);
```

```
#define PMD_SHIFT       21
/* Size of region mapped by a page middle directory */
#define PMD_SIZE        (_AC(1, UL) << PMD_SHIFT)
```

2 MiB

2M大页



pgd 1GiB 大页　　pmd 2MiB 大页　　pte 4KiB 页

页表初始化

```c
/*
 * Setup early PGD covering entire kernel which will allow
 * us to reach paging_init(). We map all memory banks later
 * in setup_vm_final() below.
 */
create_kernel_page_table(early_pg_dir, true);

/* Setup early mapping for FDT early scan */
create_fdt_early_page_table(__fix_to_virt(FIX_FDT), dtb_pa);
```

early_pg_dir
映射整个内核到高地址

arch/riscv/mm/init.c

```c
static void __init create_kernel_page_table(pgd_t *pgdir, bool early)
{
        uintptr_t va, end_va;

        end_va = kernel_map.virt_addr + kernel_map.size;
        for (va = kernel_map.virt_addr; va < end_va; va += PMD_SIZE)
                create_pgd_mapping(pgdir, va,
                                   kernel_map.phys_addr + (va - kernel_map.virt_addr),
                                   PMD_SIZE,
                                   early ?
                                        PAGE_KERNEL_EXEC : pgprot_from_va(va));
}
```

启用 MMU 前后

- **relocate_enable_mmu 启用 MMU，并跳转到高地址继续执行**
  - 用 trampoline_pg_dir 衔接
  - 启用 MMU
  - 返回后 pc 在高地址

启用 MMU 前后

```
        .global relocate_enable_mmu
relocate_enable_mmu:
        /* Relocate return address */
        la a1, kernel_map
        XIP_FIXUP_OFFSET a1
        REG_L a1, KERNEL_MAP_VIRT_ADDR(a1)
        la a2, _start
        sub a1, a1, a2
        add ra, ra, a1
```

将返回地址加上偏移量修改为高地址，
保证返回后正常运行

启用 MMU 前后

```
/* Point stvec to virtual address of intruction after satp write */
la a2, 1f
add a2, a2, a1
csrw CSR_TVEC, a2

/* Compute satp for kernel page tables, but don't load it yet */
srl a2, a0, PAGE_SHIFT
la a1, satp_mode
REG_L a1, 0(a1)
or a2, a2, a1

/*
 * Load trampoline page directory, which will cause us to trap to
 * stvec if VA != PA, or simply fall through if VA == PA.  We need a
 * full fence here because setup_vm() just wrote these PTEs and we need
 * to ensure the new translations are in use.
 */
la a0, trampoline_pg_dir
XIP_FIXUP_OFFSET a0
srl a0, a0, PAGE_SHIFT
or a0, a0, a1
sfence.vma
csrw CSR_SATP, a0
.align 2
1:
```

这里将异常处理向量地址设置为
csrw satp, a0 指令之后的高地址

_start_kernel

```
la a0, early_pg_dir
```

切换前后使用跳板页

启用 MMU 之后下一条指令的 pc 还是低地址，没有映射，发生缺页异常。因为目前临时异常入口点的位置就是下一条指令的高地址位置，所以代码跳转到标签 1，在高地址继续运行

启用 MMU 前后

```
1:
        /* Set trap vector to spin forever to help debug */
        la a0, .Lsecondary_park
        csrw CSR_TVEC, a0

        /* Reload the global pointer */
.option push
.option norelax
        la gp, __global_pointer$
.option pop

        srl a2, a0, PAGE_SHIFT
        /*                          la a1, satp_mode
         * Switch to kernel p       REG_L a1, 0(a1)
         * avoid using the tr       or a2, a2, a1
         * the first superpag
         * because that first superpage is translated the same way.
         */
        csrw CSR_SATP, a2
        sfence.vma

        ret
```

(将 stvec 设置回死循环，以便调试)

重新加载 gp

_start_kernel

```
la a0, early_pg_dir
```

necessary in order to
are only correct for
aranteed to work

页表设置为 early_pg_dir

返回地址是之前计算的高地址

异常向量初始化

异常向量初始化

- **setup_trap_vector 初始化异常向量**
  - RISC-V Linux 处理异常和中断都只用一个入口点 handle_exception
- **RISC-V 的系统调用也是一个异常**

异常向量初始化

```
setup_trap_vector:
        /* Set trap vector to exception handler */
        la a0, handle_exception
        csrw CSR_TVEC, a0

        /*
         * Set sup0 scratch register to 0, indicating to exception vector that
         * we are presently executing in kernel.
         */
        csrw CSR_SCRATCH, zero
        ret
```

RISC-V Linux 只有一个异常入口点，在 handle_exception

sscratch = 0 说明异常从内核发生

sscratch ≠ 0 指向一个 task_struct 表示内核从用户态发生

上下文保存与恢复

发生异常时进入 handle_exception，第一步先保存上下文

```
SYM_CODE_START(handle_exception)
        /*
         * If coming from userspace, preserve the user thread pointer and load
         * the kernel thread pointer.  If we came from the kernel, the scratch
         * register will contain 0, and we should continue on the current TP.
         */
        csrrw tp, CSR_SCRATCH, tp
        bnez tp, _save_context

            ...

    _save_context:
            REG_S sp, TASK_TI_USER_SP(tp)
            REG_L sp, TASK_TI_KERNEL_SP(tp)
            addi sp, sp, -(PT_SIZE_ON_STACK)
            REG_S x1,  PT_RA(sp)
            REG_S x3,  PT_GP(sp)
            REG_S x5,  PT_T0(sp)
            save_from_x6_to_x31
```

```
/* save all GPs except x1 ~ x5 */
.macro save_from_x6_to_x31
REG_S x6,  PT_T1(sp)
REG_S x7,  PT_T2(sp)
REG_S x8,  PT_S0(sp)
REG_S x9,  PT_S1(sp)
REG_S x10, PT_A0(sp)
REG_S x11, PT_A1(sp)
REG_S x12, PT_A2(sp)
REG_S x13, PT_A3(sp)
REG_S x14, PT_A4(sp)
REG_S x15, PT_A5(sp)
REG_S x16, PT_A6(sp)
REG_S x17, PT_A7(sp)
REG_S x18, PT_S2(sp)
REG_S x19, PT_S3(sp)
REG_S x20, PT_S4(sp)
REG_S x21, PT_S5(sp)
REG_S x22, PT_S6(sp)
REG_S x23, PT_S7(sp)
REG_S x24, PT_S8(sp)
REG_S x25, PT_S9(sp)
REG_S x26, PT_S10(sp)
REG_S x27, PT_S11(sp)
REG_S x28, PT_T3(sp)
REG_S x29, PT_T4(sp)
REG_S x30, PT_T5(sp)
REG_S x31, PT_T6(sp)
.endm
```

arch/riscv/include/asm/asm.h

上下文保存与恢复

```
SYM_CODE_START_NOALIGN(ret_from_exception)
        REG_L s0, PT_STATUS(sp)

          ...
        csrw CSR_STATUS, a0
        csrw CSR_EPC, a2

        REG_L x1,  PT_RA(sp)
        REG_L x3,  PT_GP(sp)
        REG_L x4,  PT_TP(sp)
        REG_L x5,  PT_T0(sp)
        restore_from_x6_to_x31

        REG_L x2,  PT_SP(sp)


          ...
```

处理异常完毕恢复上下文，
操作类似，只是反过来

中断和异常处理

arch/riscv/kernel/entry.S

```
        /*
         * MSB of cause differentiates between
         * interrupts
         */
        bge s4, zero, 1f

        /* Handle interrupts */
        tail do_irq
1:

        /* Handle other exceptions */
        slli t0, s4, RISCV_LGPTR
        la t1, excp_vect_table
        la t2, excp_vect_table_end
        add t0, t1, t0
        /* Check if exception code lies within bo
        bgeu t0, t2, 1f
        REG_L t0, 0(t0)
        jr t0
1:

        tail do_trap_unknown
```

csrr s4, CSR_CAUSE

在 handle_exception 中，读取 CSR scause

按照符号位判断：

如果 scause >= 0 说明是异常

如果 scause < 0 说明是中断

```
        .section ".rodata"
        .align LGREG
        /* Exception vector table */
SYM_CODE_START(excp_vect_table)
        RISCV_PTR do_trap_insn_misaligned
        ALT_INSN_FAULT(RISCV_PTR do_trap_insn_fault)
        RISCV_PTR do_trap_insn_illegal
        RISCV_PTR do_trap_break
        RISCV_PTR do_trap_load_misaligned
        RISCV_PTR do_trap_load_fault
        RISCV_PTR do_trap_store_misaligned
        RISCV_PTR do_trap_store_fault
        RISCV_PTR do_trap_ecall_u /* system call */
        RISCV_PTR do_trap_ecall_s
        RISCV_PTR do_trap_unknown
        RISCV_PTR do_trap_ecall_m
        /* instruciton page fault */
        ALT_PAGE_FAULT(RISCV_PTR do_page_fault)
        RISCV_PTR do_page_fault   /* load page fault */
        RISCV_PTR do_trap_unknown
        RISCV_PTR do_page_fault   /* store page fault */
excp_vect_table_end:
SYM_CODE_END(excp_vect_table)
```

中断都经 do_irq 处理，各种异常按表分发

系统调用

```
        .section ".rodata"
        .align LGREG
        /* Exception vector table */
SYM_CODE_START(excp_vect_table)
        RISCV_PTR do_trap_insn_misalig
        ALT_INSN_FAULT(RISCV_PTR do_tr
        RISCV_PTR do_trap_insn_illegal
        RISCV_PTR do_trap_break
        RISCV_PTR do_trap_load_misalig
        RISCV_PTR do_trap_load_fault
        RISCV_PTR do_trap_store_misaligned
        RISCV_PTR do_trap_store_fault
        RISCV_PTR do_trap_ecall_u /* syst
        RISCV_PTR do_trap_ecall_s
        RISCV_PTR do_trap_unknown
        RISCV_PTR do_trap_ecall_m
        /* instruciton page fault */
        ALT_PAGE_FAULT(RISCV_PTR do_page_
        RISCV_PTR do_page_fault   /* load
        RISCV_PTR do_trap_unknown
        RISCV_PTR do_page_fault   /* stor
excp_vect_table_end:
SYM_CODE_END(excp_vect_table)
```

```
asmlinkage __visible __trap_section void do_trap_ecall_u(struct pt_regs *regs)
{
        if (user_mode(regs)) {
                long syscall = regs->a7;

                regs->epc
                regs->orig
                riscv_v_vs
                syscall =

                if (syscall >= 0 && syscall < NR_syscalls)
                        syscall_handler(regs, syscall);
                else if (syscall != -1)
                        regs->a0 = -ENOSYS;

                syscall_exit_to_user_mode(regs);
```

系统调用是异常编号8，处理一些特殊情况后，交给syscall_handler执行系统调用

```
static inline void syscall_handler(struct pt_regs *regs, ulong syscall)
{
        syscall_t fn;

#ifdef CONFIG_COMPAT
        if ((regs->status & SR_UXL) == SR_UXL_32)
                fn = compat_sys_call_table[syscall];
        else
#endif
                fn = sys_call_table[syscall];

        regs->a0 = fn(regs->orig_a0, regs->a1, regs->a2,
                      regs->a3, regs->a4, regs->a5, regs->a6);
}
```

arch/riscv/include/asm/syscall.h

系统调用

```
#undef __SYSCALL
#define __SYSCALL(nr, call)     [nr] = (call),

void * const sys_call_table[__NR_syscalls] = {
        [0 ... __NR_syscalls - 1] = sys_ni_syscall,
#include <asm/unistd.h>
};
```

syscall_handler根据 a7 寄存器中的系统调用的编号，查sys_call_table表，找到第n 项是系统调用编号 n 的处理函数，调用进行处理

```
static inline void syscall_handler(struct pt_regs *regs, ulong syscall)
{
        syscall_t fn;

#ifdef CONFIG_COMPAT
        if ((regs->status & SR_UXL) == SR_UXL_32)
                fn = compat_sys_call_table[syscall];
        else
#endif
                fn = sys_call_table[syscall];

        regs->a0 = fn(regs->orig_a0, regs->a1, regs->a2,
                      regs->a3, regs->a4, regs->a5, regs->a6);
}
```

系统调用

```c
#include <asm-generic/unistd.h>

/*
 * Allows the instruction cache to be flushed from userspace.  Despite RISC-V
 * having a direct 'fence.i' instruction available to userspace (which we
 * can't trap!), that's not actually viable when running on Linux because the
 * kernel might schedule a process on another hart.  There is no way for
 * userspace to handle this without invoking the kernel (as it doesn't know the
 * thread->hart mappings), so we've defined a RISC-V specific system call to
 * flush the instruction cache.
 *
 * __NR_riscv_flush_icache is defined to flush the instruction cache over an
 * address range, with the flush applying to either all threads or just the
 * caller.  We don't currently do anything with the address range, that's just
 * in there for forwards compatibility.
 */
#ifndef __NR_riscv_flush_icache
#define __NR_riscv_flush_icache (__NR_arch_specific_syscall + 15)
#endif
__SYSCALL(__NR_riscv_flush_icache, sys_riscv_flush_icache)

/*
 * Allows userspace to query the kernel for CPU architecture and
 * microarchitecture details across a given set of CPUs.
 */
#ifndef __NR_riscv_hwprobe
#define __NR_riscv_hwprobe (__NR_arch_specific_syscall + 14)
#endif
__SYSCALL(__NR_riscv_hwprobe, sys_riscv_hwprobe)
```

arch/riscv/include/uapi/asm/unistd.h
RISC-V 特有系统调用

```c
#define __NR_io_setup 0
__SC_COMP(__NR_io_setup, sys_io_setup, compat_sys_io_setup)
#define __NR_io_destroy 1
__SYSCALL(__NR_io_destroy, sys_io_destroy)
#define __NR_io_submit 2
__SC_COMP(__NR_io_submit, sys_io_submit, compat_sys_io_submit)
#define __NR_io_cancel 3
__SYSCALL(__NR_io_cancel, sys_io_cancel)

#if defined(__ARCH_WANT_TIME32_SYSCALLS) || __BITS_PER_LONG != 32
#define __NR_io_getevents 4
__SC_3264(__NR_io_getevents, sys_io_getevents_time32, sys_io_getevents)
#endif

#define __NR_setxattr 5
__SYSCALL(__NR_setxattr, sys_setxattr)
#define __NR_lsetxattr 6
__SYSCALL(__NR_lsetxattr, sys_lsetxattr)
#define __NR_fsetxattr 7
__SYSCALL(__NR_fsetxattr, sys_fsetxattr)
#define __NR_getxattr 8
__SYSCALL(__NR_getxattr, sys_getxattr)
#define __NR_lgetxattr 9
__SYSCALL(__NR_lgetxattr, sys_lgetxattr)
#define __NR_fgetxattr 10
__SYSCALL(__NR_fgetxattr, sys_fgetxattr)
#define __NR_listxattr 11
```

...

include/uapi/asm-generic/unistd.h
各个架构通用的系统调用

小结:

- **设置初始化时的简单页表 early_pg_dir，并开启虚拟内存机制**
- **设置异常向量 stvec = handle_exception**
  - 处理异常前保存进程上下文、返回进程前恢复其上下文

内核启动前的初始化

RISC-V 机器上电到内核开始运行

（以常用的基于 U-Boot 和 OpenSBI 的启动流程为例）

- **从 ROM 中开始执行代码**
  - SoC 芯片内部包含一块 ROM 保存了一些简单的驱动
  - 从 SD 卡或 flash 加载 U-Boot SPL 到一块小内存中（单独的 SRAM 或 L2 cache-as-RAM）
- **U-Boot SPL 进行最早的初始化**
  - DDR 内存、时钟等最重要的设备
  - 从 SD 卡加载 OpenSBI 和 U-Boot 本体到 DDR 内存，然后跳转到 OpenSBI 运行

RISC-V 机器上电到内核开始运行

- **OpenSBI**
  - 初始化 IPI 和时钟设备
  - 初始化自身准备好提供 SBI 服务
  - 切换到 Supervisor mode，跳转到 U-Boot
- **U-Boot**
  - 从预先配置的启动设备加载内核到内存
  - （来源可以是 SD 卡、NVMe、网络……）
  - 跳转到内核入口点
- **操作系统开始运行**

Devicetree

- **软件如何获知机器上有哪些功能和设备**
  - ROM 代码比较固定，可以在直接代码里对照硬件资源编写
  - 通用的操作系统，Linux 内核如何在各种机器上都可以运行？
- **嵌入式设备：Devicetree**
  - 文本格式 (Devicetree source, DTS) 和二进制格式 (Flattened devicetree, FDT 或 DTB)
  - 包含设备信息、MMIO 地址、中断连接方式、时钟复位电源连接方式等信息
  - 与内核同时加载到内存中，跳转到各软件入口点时传入地址
  - 惯例：a0 是当前核心的 hartid，a1 是内存中 FDT 的物理地址
  - （U-Boot 和 OpenSBI 为了方便及复用 devicetree 已经完成的工作，也用 FDT 获得硬件配置）

RISC-V UEFI

- **RISC-V 机器固件也可以提供 UEFI 接口**
  - 提供 PCIe 控制器的驱动，可以以通用方式访问 PCIe 设备
  - 提供 PCIe Option ROM 支持，在早期启动时就可以用上外部设备的功能（显卡的显示功能、网卡的网络启动功能……）
- **操作系统通过 ACPI 获知硬件信息**
  - 比 devicetree 提供更多的信息，还提供更多接口，如通用的 PCIe 设备访问，热插拔通知，电源管理接口
  - 当然也比 devicetree 复杂
  - （在比较简单的设备上 UEFI 也可以不提供 ACPI 而提供 devicetree）

两种启动的对比

- **定制化的主板（常见的RISC-V开发板，通常不再扩展其他设备）**
  - 需要初始化具体主板相关硬件如GPIO和内存等
  - 从 devicetree 获知有哪些设备
  - 操作系统中需要很多设备相关的驱动
- **通用的主板（常见如PC，通常需要再插入其他设备）**
  - 系统配置情况在开机时候是不知道的
  - 需要探测（Probe）、Training(内存和PCIe)和枚举（PCIe等等即插即用设备）
  - UEFI/ACPI提供了整个主板、包括主板上外插的设备的软件抽象
    - 通过定义的接口把这些信息传递给OS，使OS不改而能够适配到所有机型和硬件

问题

- **Q：如果一台计算机安装了2个操作系统，如何选择?**
  - Windows会在MBR安装NTLDR
  - Linux会在MBR安装grub，可以选择Windows或Linux
  - 如果先安装Linux，再安装Windows，会出什么问题?
  - 如何解决这个问题？用USB启动Linux重新安装grub
  - 如果安装了Mac OS，会怎么样?

RISC-V 嵌入式设备启动的特点

- **通常与设备强相关**
  - ROM 通常是闭源的
  - 早期启动流程通常是厂商提供的代码
    - 如 DDR 内存控制器的初始化代码，即使有 C 源码一般也是无法理解的很多魔法数字
- **缺点：对可插拔外设的兼容性**
  - 如 RISC-V 嵌入式设备支持 PCIe 外设，需要操作系统内核包含具体硬件上的 PCIe 控制器驱动

总结

- **x86平台常见组合**
  - BIOS（在ROM）：传统BIOS、EFI/UEFI、coreboot
  - Bootloader（在磁盘的MBR）：NTLDR、Grub
  - Linux kernel（在磁盘其他位置）
- **RISC-V嵌入式平台常见组合**
  - ROM code（在ROM）：主板厂商私有
  - Bootloader（在SD卡或SPI flash）：如u-boot（非必须）
  - Linux kernel（在SD卡）