

项目设计报告

李志锐

目录

目录	i
1 综述	1
2 qemu-rv64 平台适配	3
2.1 编译运行方式	3
2.2 移植需求	3
2.3 OpenSBI 简介	3
2.4 移植说明	4
2.4.1 qemu 的配置	4
2.4.2 地址空间布局	4
2.5 移植详细内容	4
2.5.1 简述	4
2.5.2 libcpu 适配	5
2.5.2.1 编译配置文件适配	5
2.5.2.2 启动代码适配	6
2.5.2.3 一些通用宏（位于 riscv.h 中，在之后会很常用）	7
2.5.2.4 上下文切换接口的对接	7
2.5.2.5 内核分页功能的实现（page.c/h）	12
2.5.2.6 MMU 支持的实现（riscv_mmu.c/h mmu.c./h）	14
2.5.2.7 Cache 支持的实现（cache.c）	16
2.5.2.8 Syscall 对接	16
2.5.2.9 系统 tick 接口对接（tick.c/h）	18
2.5.2.10 加入符号表解析支持及 Stacktrace 支持（symbol_analysis.c/h）	19
2.5.3 lwp 适配	23
2.5.3.1 编译配置文件适配	23
2.5.3.2 arch_user_space_init 适配（arch_user_space_init.c）	24
2.5.3.3 arch_user_stack 适配（arch_user_stack.c）	24
2.5.3.4 lwp_gcc 适配（lwp_gcc.S）	24
2.5.4 bsp 适配（bsp/qemu-virt-rv64）	25
2.5.4.1 编译配置文件适配	25
2.5.4.2 中断控制器驱动适配（driver/plic.c/h）	25
2.5.4.3 UART 驱动适配（driver/drv_uart.c/h）	26
2.5.4.4 板级初始化接口实现（board.c/h）	26
2.5.4.5 中断框架实现、trap 接口适配及加入 stacktrace 支持（trap.c/h）	27
3 Kendryte K210 平台适配	29
3.1 编译运行方式	29

3.2 移植综述	29
3.3 OpenSBI 适配	30
3.3.1 platform.c 修改	30
3.3.2 K210 中断转发机制的实现	30
3.3.3 config.mk 修改	33
3.3.4 实现编译脚本	33
3.3.5 Musl libc 适配	35
3.3.6 MMU 适配	35
3.3.7 中断控制器驱动适配	35
3.3.8 PLL 驱动适配 (driver/drv_pll.c/h)	36
3.3.9 UART 驱动适配 (drivers/drv_uart.c/h)	36
3.3.10 Tick 驱动适配 (libcpu/risc-v/k210_mmu/tick.c)	36
3.3.11 板级初始化接口实现	36
3.3.12 中断框架实现、trap 接口适配及加入 stacktrace 支持	37
3.3.13 SD 卡驱动适配及 FAT32 文件系统适配	37
4 全志 D1 平台适配	39
4.1 编译运行方式	39
4.2 移植综述	39
4.3 MMU 适配	39
4.4 UART 驱动适配 (drivers/drv_uart.c/h)	40
4.5 Tick 驱动适配 (libcpu/risc-v/t-head/c906/tick.c)	41
5 DreamOS 内核的相关说明 (我自己的内核)	43
5.1 设计综述	43
5.2 内核设计	43
5.2.1 os_annotation.c	45
5.2.2 os_bitmap.c	47
5.2.3 os_debug.c	47
5.2.4 os_device.c	48
5.2.5 os_elf.c	49
5.2.6 os_file.c	49
5.2.7 os_hashmap.c	51
5.2.8 os_init.c	52
5.2.9 os_interrupt.c	52
5.2.10 os_io.c	53
5.2.11 os_memory.c	53
5.2.11.1 memory/os_memory_page.c	53
5.2.11.2 memory/os_memory_slub.c	54
5.2.12 os_mmu.c	54
5.2.13 os_mutex.c	55
5.2.14 os_string.c	55
5.2.15 os_syscall.c	55
5.2.16 os_task.c	56
5.2.17 os_terminal_color.c	58
5.2.18 os_tick.c	58
5.2.19 os_vfs.c	59

5.2.20	os_waitqueue.c	60
5.3	qemu-rv64 平台的移植	61
5.4	编译运行方式	61
5.4.1	移植综述	61
5.4.2	firmware (firmware/sbi)	61
5.4.3	arch (arch/riscv64)	61
5.4.3.1	arch.c	62
5.4.3.2	arch.h	62
5.4.3.3	arch_asm.S	62
5.4.3.4	arch_err.h	62
5.4.3.5	arch_mmu.c	63
5.4.3.6	arch_mmu.h	63
5.4.3.7	arch_trap.c	64
5.4.3.8	arch_trap.h	64
5.4.3.9	entry_gcc.S	65
5.4.3.10	function.S	65
5.4.3.11	interrupt_gcc.S	65
5.4.3.12	syscall_gcc.S	65
5.4.4	bsp (bsp/qemu-virt-rv64)	65
5.4.4.1	bsp_init.c	66
5.4.4.2	bsp_interface_asm.S	66
5.4.4.3	bsp_interface.c	66
5.4.4.4	entry_main.c	66
5.4.4.5	task_main_entry.c	66
5.4.4.6	tick.c/h	67
5.4.4.7	trap_handler.c	67
5.4.4.8	osconfig.h	67
5.4.4.9	osconfig.py	67
6	未来的工作展望	69

第 1 章

综述

注意，本文档拥有标签，可以使用 **SumatraPDF/Foxit Reader** 等阅读器利用标签阅读本文档

这次项目我主要实现了如下几部分内容：

- 为 qemu-rv64 实现了 RT-Thread Smart 适配
- 为 Kendryte K210 实现了 RT-Thread Smart 适配
- 为全志 D1 实现了 RT-Thread Smart 适配

其中，由于上述后两部分均派生于第一部分的代码，因此，本文档将从第一部分开始说明。

同时，在移植之余，我同时还在进行自己的内核的编写，内核位于仓库的 **dreamos** 目录中，详情在之后的章节描述。

本次移植全程需要使用 **riscv64-unknown-elf** 工具链，目前使用的是 **riscv64-unknown-elf-gcc-8.3.0-2020.04.1-x86_64-linux-ubuntu14** (<https://github.com/sifive/freedom-tools/releases/tag/v2020.04.0-Toolchain.Only>，我所使用的平台为 **ubuntu 16.04**)，注意将工具链的 **bin** 目录加入 **PATH** 环境变量。

同时本次移植依赖 **scons** 工具，并要求默认的 **Python** 版本必须为 **2.7**，不能是 **3.x**，否则可能会报各种各样的错误。

若提示 “**scons: *** [SConscript] ValueError : unsupported pickle protocol: 5**” 提示，这通常是由于本次和上次编译使用的 **scons** 版本不同，请删除编译目录下的 **.sconsign.dblite** 文件再行编译，若提示某些头文件和工具链冲突，注意删除 **cconfig.h** 文件，这样在编译时编译脚本就会智能识别工具链所包含的头文件并重新生成 **cconfig.h**。

以下系统编译时可能会提示 **Permission Denied**，这是因为其依赖 **osdebugsupport64** 工具，可以分别给 **rt-thread/tools/osdebugsupport** 下的两个可执行文件以及 **dreamos/tools/osdebugsupport** 下的两个可执行文件为当前用户加上可执行权限。

这里我同时实现了 **Musl libc** 和 **RT-Thread** 库的适配，由于商业的原因，这两个库的源代码不能开源，但是我将编译好的库文件和相关头文件分别放置在了仓库的如下位置：

- **Musl libc** 的内核部分：**rt-thread/components/libc/compilers/musl/libc**
- **Musl libc** 的用户态部分：**userapps/sdk/libc**
- **RT-Thread** 库：**userapps/sdk/rt-thread**

同时为用户程序实现了与 Musl libc 库适配的系统调用接口程序，该程序位于 `userapps/sdk/syscall.S`。

`userapps` 目录中包含四个用户程序：`hello`、`ping`、`pong`、`vi`，其中 `hello` 是一个简单的 `printf` 测试程序，同时对 `malloc` 的内存的数据进行自增测试，`ping/pong` 则用于进程间通信测试，测试方法是先执行 “`pong &`”，启动 `pong` 进程并让其运行在后台，然后执行 “`ping`” 完成通信测试，`vi` 则是移植好的 `vi` 文本编辑器。

`userapps/root` 目录即为根文件系统，以下是正确的用户程序编译和根文件系统生成方法：

1. 进入 `userapps` 目录
2. 在 `userapps/root` 目录中建立 `mnt` 目录，`mnt` 目录中不要放置任何文件
3. 修改 `.config` 文件中的工具链路径
4. 执行 `scons`，然后可以看到结果文件生成到 `userapps/root/bin` 目录中
5. 在 `userapps/root/bin` 目录中为每个 `elf` 执行 “`riscv64-unknown-elf-strip -strip-all elf 文件名`” 去除每个用户程序中的调试节区从而缩小用户程序的体积
6. 修改 `userapps/write_rom.sh` 文件，将其中的 `bsp` 目录指向你需要为其生成根文件系统的 `bsp`
7. 执行 `write_rom.sh`（这一步可能会出现权限问题，注意要给 `stripall.sh` 以及 `write_rom.sh` 都加上可执行权限）

要先生成根文件系统再执行内核 **BSP** 的编译！！

第 2 章

qemu-rv64 平台适配

2.1 编译运行方式

首先在 `bsp` 目录下的 `rtconfig.py` 中配置正确的工具链路径，然后在 `bsp` 目录中执行 “`scons -jN`” (`N` 表示并行数)，即可得到 `rtthread.bin`，运行 `qemu-nographic.sh` 脚本即可启动。

2.2 移植需求

- 内核和用户程序依赖 `musl libc` 库
- 用户程序依赖 `RT-Thread` 库
- 依赖 `qemu` 的 `OpenSBI`

2.3 OpenSBI 简介

`OpenSBI` 的仓库地址为<https://github.com/riscv/opensbi>，其目的是提供一个 `RISC-V SBI` (`Supervisor Binary Interface`) 的实现，该实现面向在特定平台上运行于 `M` 态的固件 (`firmware`)，利用 `OpenSBI`，可以让运行于 `S` 态下的 `OS` 按照 `RISC-V SBI` 规范访问相关硬件，如控制台的输入输出等。

本次移植将会使用如下几个接口：

```
static inline void sbi_console_putchar(int ch);
static inline int sbi_console_getchar(void);
static inline void sbi_set_timer(uint64_t stime_value);
```

`SBI` 操作头文件及相关源代码来自 `FreeBSD`，遵循 `BSD` 许可证，该许可证兼容 `RT-Thread Smart` 所使用的 `Apache` 许可证。

2.4 移植说明

2.4.1 qemu 的配置

以下是我所采用的 qemu 的基本启动参数：

```
qemu-system-riscv64 -nographic -machine virt -m 256M -kernel rtthread.bin
```

上述启动参数将 RAM 设置为 256MiB，无图形化环境，单核，未开启 Cache。

注意，这里采用的 Qemu 版本号为 5.2.0，请注意检查

2.4.2 地址空间布局

以下是我设计的地址空间结构（由于地址过长，使用下划线对地址进行分隔以便于阅读）：

表 2.1：地址空间布局

起始地址	空间大小	功能
0x00000000_00000000	0x00000000_80200000	硬件访存空间及其它用途
0x00000000_80200000	0x00000000_7FE00000	内核程序和数据空间（包括栈和堆）
0x00000001_00000000	0x00000000_50000000	用户程序启动参数
0x00000001_50000000	0x00000000_B0000000	保留空间
0x00000002_00000000	0x00000000_70000000	用户程序和静态数据空间
0x00000002_70000000	0x00000000_90000000	用户程序栈空间
0x00000003_00000000	0xFFFFFFF0_00000000	用户程序堆空间

2.5 移植详细内容

2.5.1 简述

本次移植共分为三大步骤：

- 适配 libcpu
- 适配 lwp
- 适配 bsp

RT-Thread Smart 的 libcpu 中保存和 CPU 架构相关的代码，lwp 下保存轻量级进程相关的架构支持代码，bsp 则为板级支持代码。

2.5.2 libcpu 适配

这部分的移植共分为以下几大步骤：

- 编译配置文件适配
- 启动代码适配
- 上下文切换接口适配
- Trap 对接与中断接口对接
- 内核分页功能对接
- MMU 支持对接
- Cache 支持对接
- Syscall 接口对接
- 系统 Tick 接口对接
- 加入符号表解析支持及 Stacktrace 支持

2.5.2.1 编译配置文件适配

1. 需要创建 libcpu/risc-v/virt64 目录
2. 修改 libcpu/risc-v/SConscript 文件为如下内容：

```
# RT-Thread building script for bridge

import os
from building import *

Import('rtconfig')

cwd = GetCurrentDir()
group = []
list = os.listdir(cwd)

# add common code files
if rtconfig.CPU == "e906" :
    group = group
elif rtconfig.CPU == "nuclei" :
    group = group
elif rtconfig.CPU == "virt64" :
    group = group
else :
    group = group + SConscript(os.path.join(cwd, 'common', 'SConscript'))

# cpu porting code files
if rtconfig.CPU == "e906" :
    group = group + SConscript(os.path.join(cwd, rtconfig.VENDOR, rtconfig.CPU, 'SConscript'))
else :
```

```
group = group + SConscript(os.path.join(cwd, rtconfig.CPU, 'SConscript'))

Return('group')
```

也就是为 virt64 加入相关配置项。

3. 在 libcpu/Kconfig 文件中加入 “config ARCH_RISCV”、“config ARCH_RISCV64” 等配置项，并让 ARCH_RISCV64 配置项自动选择 ARCH_RISCV 与 ARCH_CPU_64BIT 配置项，即如下代码：

```
config ARCH_RISCV
    bool

config ARCH_RISCV64
    select ARCH_RISCV
    select ARCH_CPU_64BIT
    bool
```

2.5.2.2 启动代码适配

以下是启动代码的初始化流程（位于 startup_gcc.S 中）：

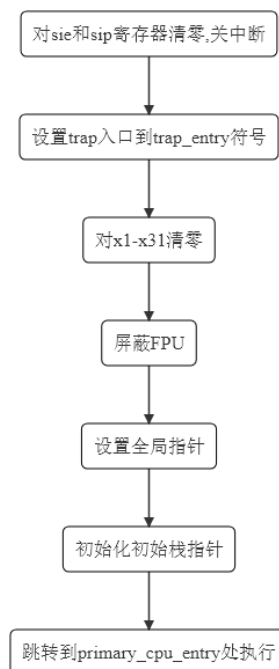


图 2.1: 启动代码流程图

可以看到，上述完成了基本的初始化步骤后就转入了 primary_cpu_entry 处执行，而 primary_cpu_entry 位于 bsp/qemu-virt-rv64/driver/board.c 中，其实质是初始化 bss 节区并从 entry 接口启动内核。

2.5.2.3 一些通用宏（位于 `riscv.h` 中，在之后会很常用）

表 2.2: `riscv.h` 中的一些通用宏

宏定义	宏功能
<code>__SIZE(bit)</code>	返回 2^{bit}
<code>__MASK(bit)</code>	返回连续 <code>bit</code> 个 1，用于生成右对齐的连续屏蔽保留信号
<code>__UMASK(bit)</code>	对 <code>__MASK(bit)</code> 取反，用于生成右对齐的连续屏蔽信号
<code>__MASKVALUE(value,maskvalue)</code>	等效 <code>value & maskvalue</code> ，用于保留某些位
<code>__UMASKVALUE(value,maskvalue)</code>	等效 <code>value & ~maskvalue</code> ，用于屏蔽某些位
<code>__CHECKUPBOUND(value,bit_count)</code>	若只有最低 <code>bit_count</code> 个位包含 1，则返回 1，否则返回 0，用以检测一个值是否超出范围
<code>__CHECKALIGN(value,start_bit)</code>	若最低 <code>start_bit</code> 个位只包含 0，则返回 1，否则返回 0，用以检测一个地址是否对齐
<code>__PARTBIT(value,start_bit,length)</code>	取出一个值从 <code>start_bit</code> 位（底为 0）开始的 <code>length</code> 个位
<code>__ALIGNUP(value,bit)</code>	按 <code>bit</code> 向上对齐，保证最低 <code>bit</code> 个位为 0
<code>__ALIGNDOWN(value,bit)</code>	按 <code>bit</code> 向下对齐，保证最低 <code>bit</code> 个位为 0

2.5.2.4 上下文切换接口的对接

上下文切换接口包含两部分，一部分是位于 `context_gcc.S` 中的 `rt_hw_context_switch_to` 与 `rt_hw_context_switch`，另一部分则是位于 `interrupt_gcc.S` 中中断处理的部分。

`rt_hw_context_switch_to` 用于启动系统的第一个线程，而 `rt_hw_context_switch` 用于非中断中发起的上下文切换（即通过在线程上下文中主动调用 `rt_schedule` 发起）

以下是我所采用的上下文栈结构：

表 2.3: 上下文栈结构

偏移地址	字段
0x00	epc
0x08	ra
0x10	sstatus
0x18	gp
0x20	tp
0x28	t0
0x30	t1

偏移地址	字段
0x38	t2
0x40	s0_fp
0x48	s1
0x50	a0
0x58	a1
0x60	a2
0x68	a3
0x70	a4
0x78	a5
0x80	a6
0x88	a7
0x90	s2
0x98	s3
0xA0	s4
0xA8	s5
0xB0	s6
0xB8	s7
0xC0	s8
0xC8	s9
0xD0	s10
0xD8	s11
0xE0	t3
0xE8	t4
0xF0	t5
0xF8	t6
0x100	user_sp_exc_stack

也就是说上下文栈的大小为 264 字节。

上下文栈的初始化接口对接（cpuport.c）：

这一部分需要实现接口 `rt_hw_stack_init`，该接口的功能是初始化线程栈，并将包含有上下文的栈的栈底地址返回，本次实现主要做了如下事情：

1. 将上下文中的所有寄存器初始化为 `0xdeadbeef`
2. 将 `ra` 寄存器设置为线程退出函数的地址，通常用于线程退出时的一些环境清理工作

3. 初始化 `gp` 寄存器为内核全局指针
4. 初始化 `a0` 寄存器为线程参数
5. 初始化 `epc` 字段为线程入口函数地址
6. 初始化 `user_sp_exc_stack` 字段为线程内核栈栈顶，用于用户态程序的 `syscall` 和中断结束前的上下文切换

`rt_hw_context_switch_to` 接口（位于 `context_gcc.S` 中）：

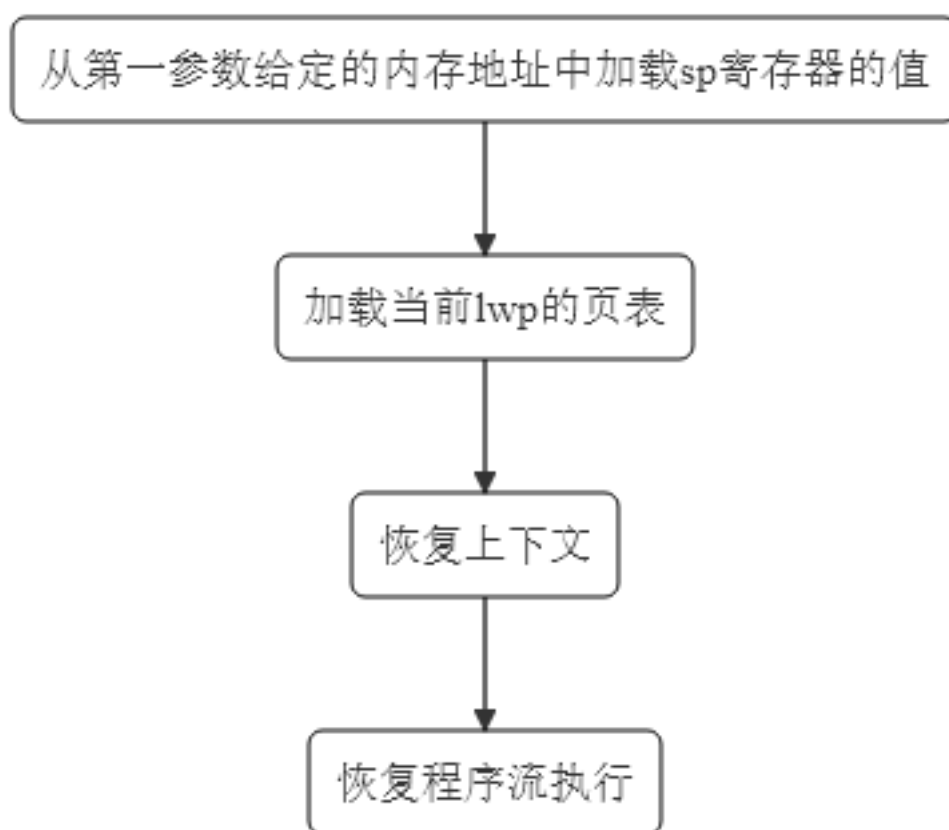
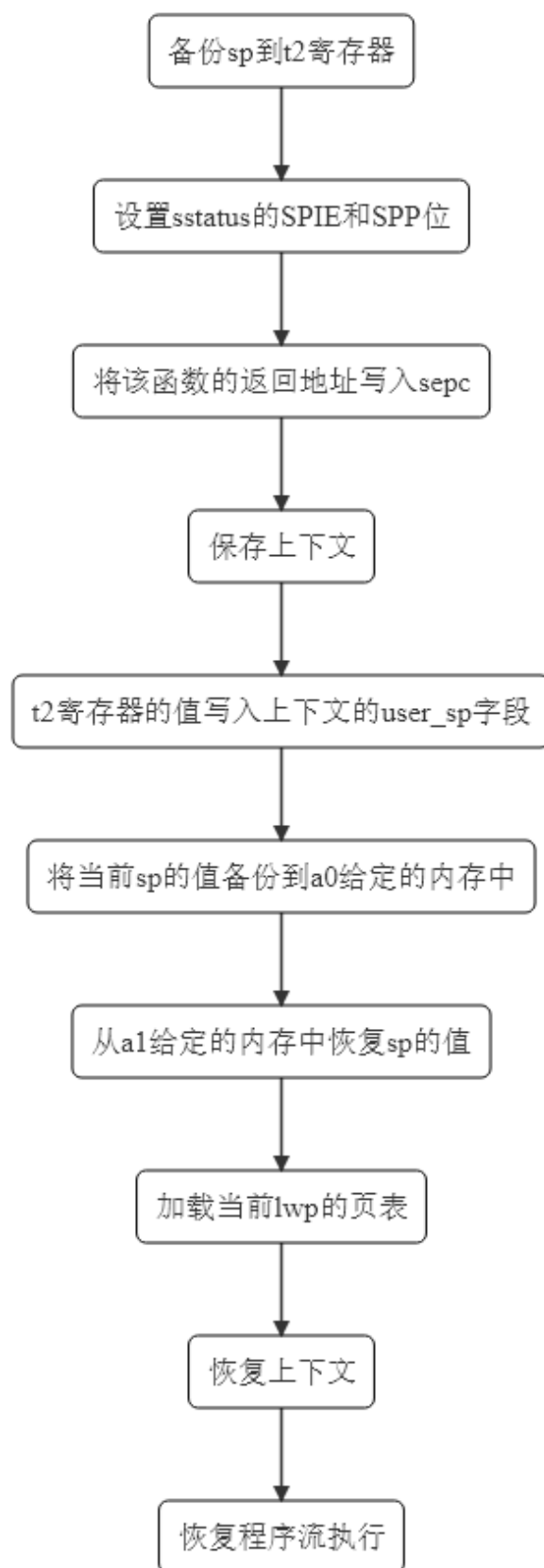


图 2.2: `rt_hw_context_switch_to` 接口流程图

`rt_hw_context_switch` 接口（位于 `context_gcc.S` 中）：

图 2.3: `rt_hw_context_switch` 接口流程图

`trap_entry` 入口及中断内的上下文切换（位于 `interrupt_gcc.S`）:

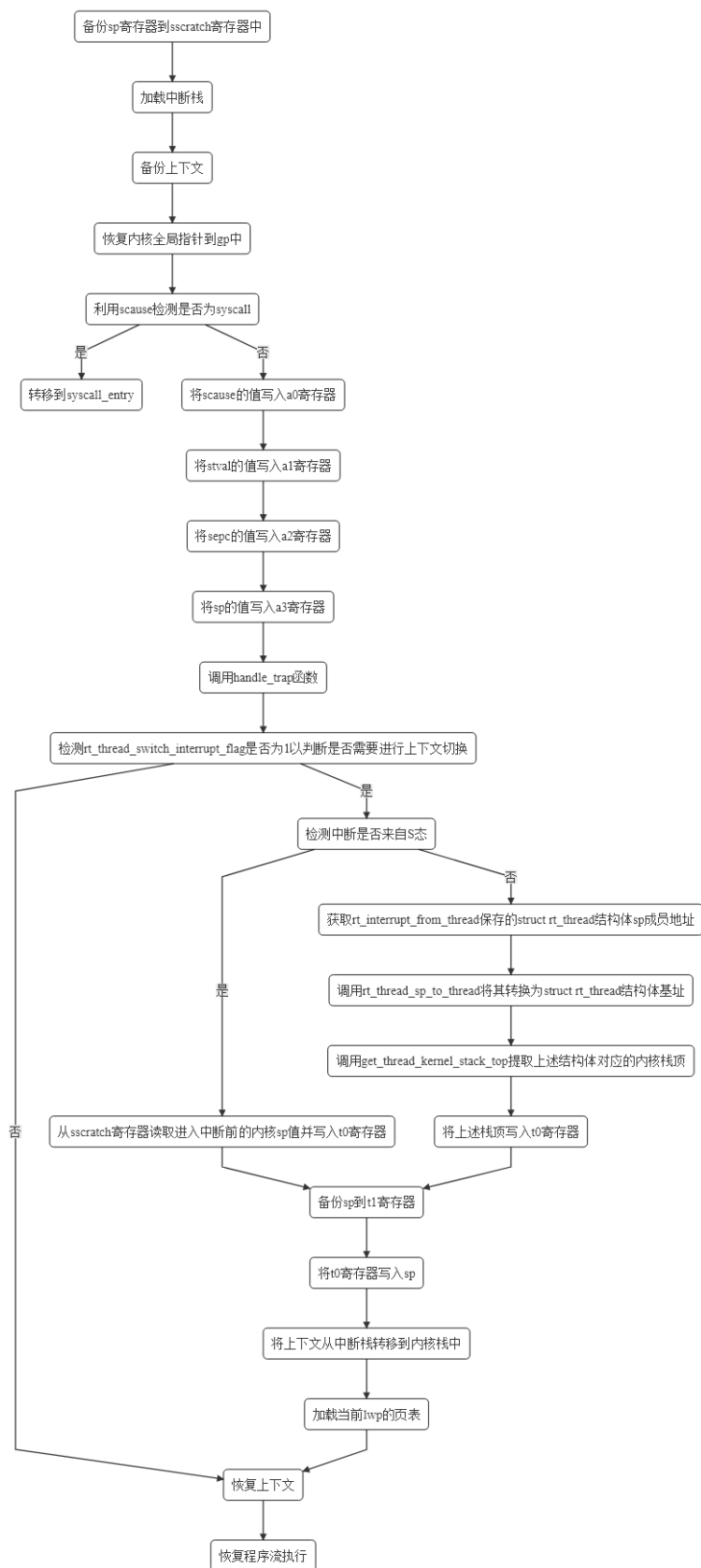


图 2.4: trap_entry 入口及中断内的上下文切换流程图

2.5.2.5 内核分页功能的实现 (page.c/h)

内核分页功能基于伙伴分配算法实现了如下接口：

```
rt_size_t rt_page_bits(rt_size_t size); //将实际的大小转换为最接近的页面大小对2的对数
void *rt_pages_alloc(rt_size_t size_bits); //分配2^size_bits大小的页面
void rt_pages_free(void *addr, rt_size_t size_bits); //在指定地址释放大小为2^size_bits大小的页面
void rt_pageinfo_dump(); //打印页面信息
void rt_page_get_info(size_t *total_nr, size_t *free_nr); //获取总页面数和空闲页面数
void rt_page_init(rt_region_t reg); //初始化页面
```

这部分具体实现参考了 RT-Thread Smart 的 ARM 版实现，同时，我在https://github.com/lizhirui/dreamos_cversion/blob/master/src/phypage.c给出了同样有效且更简洁的实现。

这里我要对这个 **buddy system** 实现中的初始化时的内存空间分配公式特别说明：该 **buddy system** 的 **metadata** 虽然是在内存中按顺序存放的，但是在划分空间的时候，假定 **metadata** 不跨越页边界，因此其计算较为复杂，首先定义几个变量：

- **nr** - 每页可以存放的 **metadata** 项数
- **total** - 总页数
- **mnr** - **metadata** 需要占用的页数

根据这些定义我们可以得到如下方程：

$$\lceil \frac{total - mnr}{nr} \rceil = mnr$$

上式中 **total - mnr** 可以理解为数据页数，也可以理解为所需的 **metadata** 的数量，这里上取整是因为，如果 **metadata** 的数量是小数，那么必然要多占一页用于保存 **metasize**。

我们令：

$$total - mnr \equiv k * nr + p (k \in \mathbb{Z}, k \geq 0, 0 \leq p < nr)$$

那么方程变为：

$$k + \lceil \frac{p}{nr} \rceil = mnr$$

当 $p = 0$ 时，方程变为：

$$k = mnr$$

将 **k** 代入后可得：

$$\frac{total - mnr}{nr} = mnr$$

变换后可解得：

$$mnr = \frac{total}{nr + 1}$$

当 $p > 0$ 时， k 变为：

$$k = \frac{total - mnr - p}{nr}$$

同时：

$$\lceil \frac{p}{nr} \rceil = 1$$

因此方程变为：

$$\frac{total - mnr - p}{nr} + 1 = mnr$$

可解得：

$$mnr = \frac{total - p + nr}{nr + 1}$$

为了尽可能增加可用数据页面的数量，并简化计算，因此这里让 mnr 尽可能大，这里令 $p = 0$ ，因此上式变为：

$$mnr = \frac{total + nr}{nr + 1}$$

综合上述两种情况，为了保证 `metasize` 足够描述所有的剩余数据页面，并简化计算这里对上述两种情况取最大值，可得：

$$mnr = \frac{total + nr}{nr + 1}$$

这也就是下面这行代码的来历：

```
rt_size_t mnr = (total + nr) / (nr + 1);
```

而我内核中的 `buddy system` 的空间划分公式要比这个简单的多，这个放到后面说。

2.5.2.6 MMU 支持的实现 (riscv mmu.c/h mmu.c./h)

一些背景知识:

RISC-V 包含 3 种分页模式，分别为 Sv32、Sv39、Sv48，这三种分页模式顾名思义，分别对应 32 位、39 位和 48 位虚拟地址。

由于 RT-Thread Smart 本次移植的目标架构为 RV64，因此此处选择 Sv39 虚拟地址映射模式，该模式的虚拟地址及物理地址结构如下：

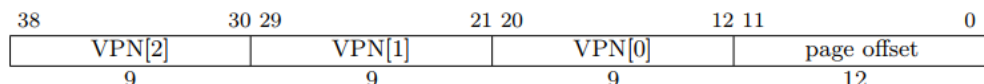


Figure 4.16: Sv39 virtual address.

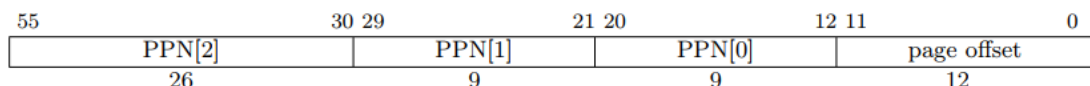


Figure 4.17: Sv39 physical address.

图 2.5: Sv39 模式的虚拟地址及物理地址的结构

单个页表项的定义如下:

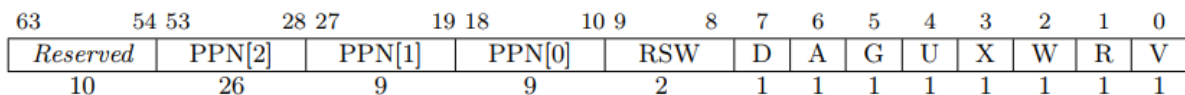


图 2.6: 单个页表项的定义

其中各位含义如下：

- **RSW**: 保留给 **S** 态软件使用
- **D: Dirty**, 当对应的页发生写入操作时, 该位被硬件置位, 可被软件清零
- **A: Accessed**, 当对应的页发生读取、写入或取指操作时, 该位被硬件置位, 可被软件清零
- **G: global**, 指明一个页表项是否是一个全局项 (和 **ASID** 无关), 若该页表项非叶子结点且被设置 **G** 位, 则其所有下级页表项均为全局
- **U: user**, 指明一个页面是否在用户态可访问, 该属性在非叶子结点会被忽略, 当 **sstatus.SUM** 为 **0** 时, 在 **S** 态访问此类页面会失败
- **XWR**: 分别对应 **execute**、**write**、**read**, 若都为 **0**, 则表示该项有下级结点, 即 **PPN** 给定的是下级页表项的物理地址
- **V: valid**, 指明一个页表项是否有效

Sv39 具有 512GiB 的虚拟地址空间，最多支持三级页表，同时该模式支持三种页面大小，对应于三级，分别为 4KiB(2^{12} B)、2MiB(2^{21} B)、1GiB(2^{30} B)。4 前述 PPN 实际上含义为 Physical Page Number，即物理页编号，该值乘以页面大小才为真正的页面基地址，而每个存储页表的页面大小固定为 4KiB。

RISC-V 采用 `sfence.vma` 指令刷新 TLB，该指令有两个操作数，都为寄存器，其含义分别为 `asid` 和 `vaddr`，若都为 `x0`，则对整个 TLB 刷新。

RISC-V 采用 `satp` 寄存器存储虚拟地址模式、当前 `ASID` 与页表物理地址，其定义如下：

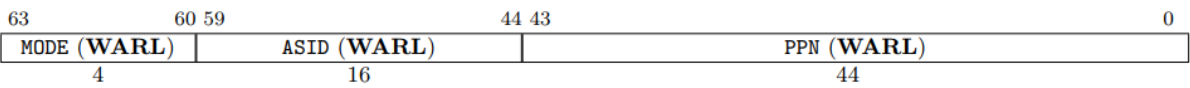


Figure 4.12: RV64 Supervisor address translation and protection register `satp`, for MODE values Bare, Sv39, and Sv48.

图 2.7: `satp` 寄存器定义

其中 `MODE` 字段的取值如下：

RV32		
Value	Name	Description
0	Bare	No translation or protection.
1	Sv32	Page-based 32-bit virtual addressing (see Section 4.3).

RV64		
Value	Name	Description
0	Bare	No translation or protection.
1–7	—	<i>Reserved</i>
8	Sv39	Page-based 39-bit virtual addressing (see Section 4.4).
9	Sv48	Page-based 48-bit virtual addressing (see Section 4.5).
10	<i>Sv57</i>	<i>Reserved for page-based 57-bit virtual addressing.</i>
11	<i>Sv64</i>	<i>Reserved for page-based 64-bit virtual addressing.</i>
12–15	—	<i>Reserved</i>

图 2.8: `satp` 寄存器的 `mode` 字段取值

本次移植共实现了如下几个接口：

- `mmu_set_pagetable` 设置当前页表
- `mmu_enable_user_page_access` 使能 S 态用户页面访问
- `mmu_disable_user_page_access` 失能 S 态用户页面访问
- `mmu_table_get` 获取当前页表
- `switch_mmu` 设置当前页表
- `rt_hw_mmu_map_init` 初始化 MMU 映射
- `rt_hw_mmu_kernel_map_init` 初始化 MMU 内核映射（按 1GiB 页大小为单位和物理地址一对一映射）
- `rt_hw_mmu_map` 映射页面
- `rt_hw_mmu_map_auto` 自动分配物理页面并映射页面
- `rt_hw_mmu_unmap` 取消映射页面
- `rt_hw_mmu_v2p` 虚拟地址转物理地址

`mmu_set_pagetable` 通过将页表地址和虚拟地址转换模式写入 `satp` 寄存器并执行 `sfence.vma x0` 指令来实现。

`mmu_enable_user_page_access` 与 `mmu_disable_user_page_access` 通过设置和取消 `sstatus` 寄存器的 `PUM` 位来实现用户页面的访问使能和失能。

`mmu_table_get` 通过读取 `current_mmu_table` 全局变量来获得当前的页表。

`switch_mmu` 将 `current_mmu_table` 全局变量的值置为当前页表的地址，同时调用 `mmu_set_pagetable` 实现页表的设置，并通过 `cache` 相关接口完成 `cache` 的刷新和无效化。

`rt_hw_mmu_map_init` 根据给定的地址空间范围以及分配好的虚拟页表初始化给定的 `rt_mmu_info` 结构体。

`rt_hw_mmu_kernel_map_init` 通过初始化给定的 `rt_mmu_info` 结构体中的虚拟页表的前四项来完成低 4GiB 内存在内核中的一一映射。

`rt_hw_mmu_map` 寻找页表中的空闲虚拟地址并与物理地址关联。

`rt_hw_mmu_map_auto` 调用 `buddy system` 完成物理页分配并调用 `rt_hw_mmu_map` 的一个内部实现完成与虚拟地址的关联。

`rt_hw_mmu_unmap` 通过在页表删除对应关联来实现。

上述三个函数，在必要时会调用 `buddy system` 申请或释放第二、三级页表。

`rt_hw_mmu_v2p` 通过查询页表完成虚拟地址到物理地址的转换。

上述四个接口的实现同样参考了 RT-Thread Smart ARM 版的实现思路。

2.5.2.7 Cache 支持的实现 (cache.c)

要实现 Cache 支持，必须实现如下若干接口：

- `rt_cpu_icache_line_size` 返回 I\$ 缓存行大小
- `rt_cpu_dcache_line_size` 返回 D\$ 缓存行大小
- `rt_hw_cpu_icache_invalidate` 将 I\$ 指定的地址范围对应的缓存行全部无效化
- `rt_hw_cpu_dcache_invalidate` 将 D\$ 指定的内存范围对应的缓存行全部无效化
- `rt_hw_cpu_dcache_clean` 将 D\$ 指定的内存范围对应的缓存行全部写入 RAM 从而保持 Cache 与 RAM 的数据一致性
- `rt_hw_cpu_icache_ops` 将 I\$ 相关操作封装的统一调用接口
- `rt_hw_cpu_dcache_ops` 将 D\$ 相关操作封装的统一调用接口
- `rt_hw_cpu_dcache_flush_all` 刷新整个 D\$
- `rt_hw_cpu_icache_invalidate_all` 无效化整个 I\$
- `rt_hw_cpu_icache_status` 获取 I\$ 状态
- `rt_hw_cpu_dcache_status` 获取 D\$ 状态

由于本次 `qemu-virt64` 未使用 `cache`，因此此处的相关函数实际都为空，无实际作用，仅为与内核对接所需的必要接口。

2.5.2.8 Syscall 对接

`syscall` 对接分为两部分，一部分为位于 `interrupt_gcc.S` 中的汇编部分，另一部分位于 `syscall_c.c` 中的 c 部分，在此约定，`a0~a6` 寄存器保存 `syscall` 的 7 个参数，`a7` 保存 `syscall` 的编号。

以下是 `syscall1` 的处理流程（`syscall_entry` 入口，位于 `interrupt_gcc.S`）：

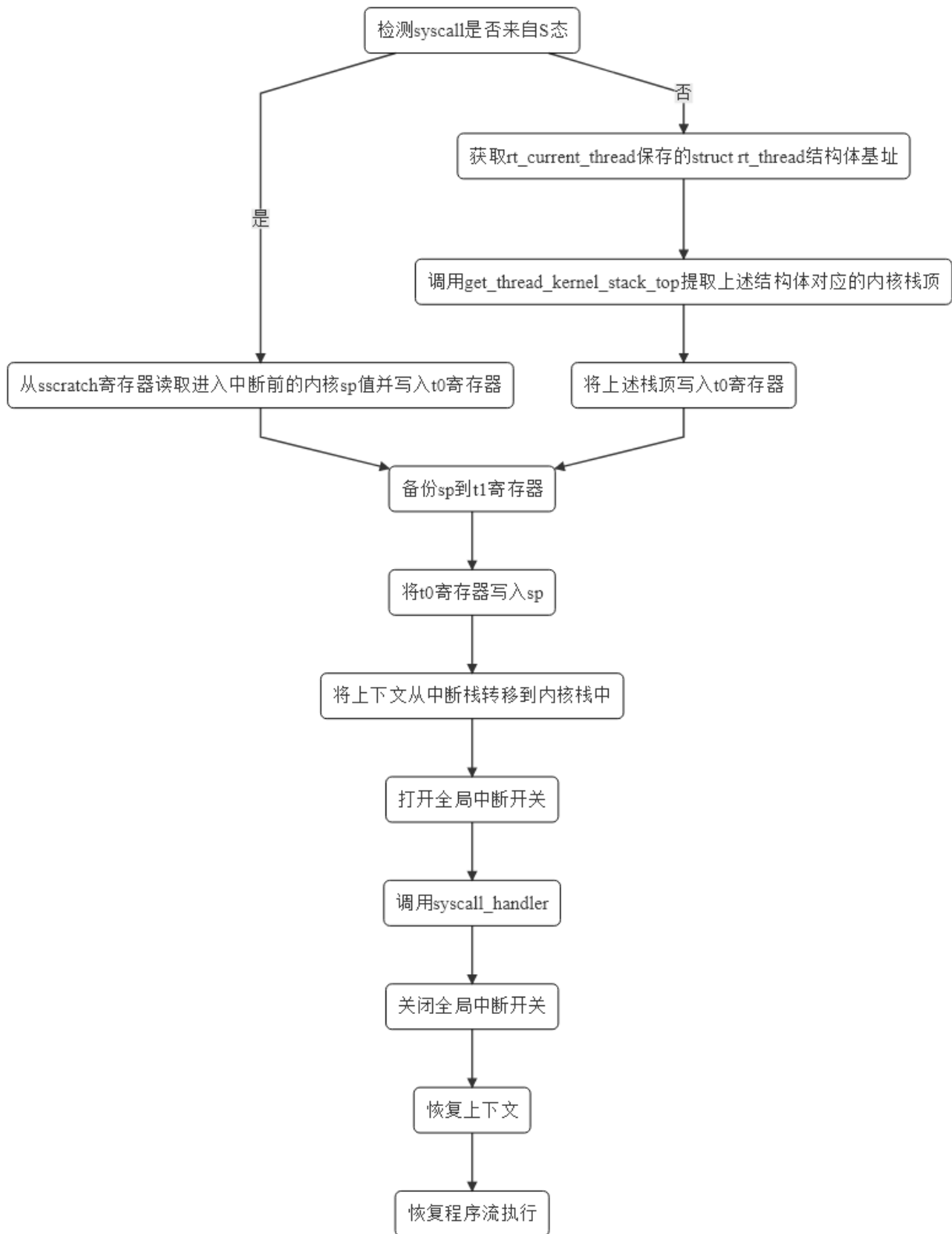


图 2.9: `syscall1` 的处理流程

以及第二入口（`syscall_handler` 入口，位于 `syscall_c.c`）：

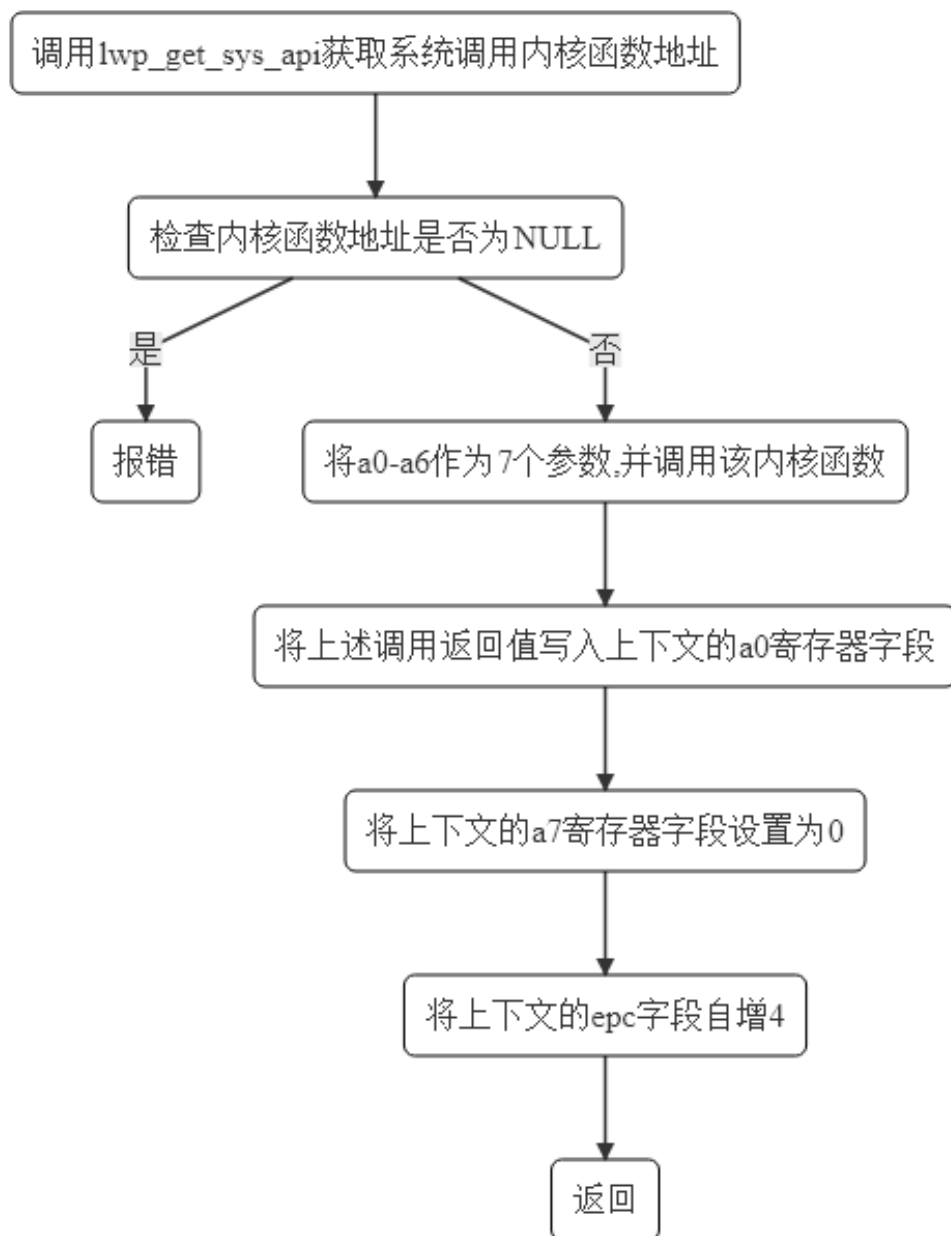


图 2.10: syscallL 第二入口的处理流程

2.5.2.9 系统 tick 接口对接 (tick.c/h)

实现 tick 驱动有以下几个要点:

1. 实现 tick 中断, 并在中断中调用 rt_tick_increase
2. 实现 rt_hw_tick_init, 用来对 tick 进行初始化

本次共实现了三个接口:

- get_ticks 通过直接调用 rdttime 指令获取并返回定时器的 tick 值
- tick_isr tick 中断处理程序
- rt_hw_tick_init 实现 tick 的初始化

我们可以很容易从<https://github.com/qemu/qemu/blob/master/hw/riscv/virt.c>第 593 行得知, `tick` 的时钟频率为 `SIFIVE_CLINT_TIMEBASE_FREQ`, 而该常量定义在https://github.com/Comsecuris/qemu-hexagon/blob/master/include/hw/riscv/sifive_clint.h中, 值为 `100000000`。

`RT-Thread Smart` 定义常量 `RT_TICK_PER_SECOND` 表示内核每秒的 `tick` 数, 将其转换为周期 (ms) 的公式为 $\text{interval}(\text{ms}) = 1000 / \text{RT_TICK_PER_SECOND}$ 。

那么定时器的增量值 $\text{tick_cycle} = \text{interval}(\text{ms}) / 1000 * \text{freq} - 1$, 其中 $\text{freq} = 100000000\text{Hz}$, 这里 freq 的单位为 Hz, 因为从 `time` 与 `timecmp` 相等到产生中断需要一个时钟周期, 因此这里-1。

`rt_hw_tick_init` 执行步骤如下:

1. 清除 `SIE` 寄存器中的 `STIP` 位, 关掉 `S` 态定时器中断
2. 调用 `get_ticks` 并在其返回值的基础上加上到下次触发中断前的 `tick` 值作为参数传递给 `sbi_set_timer`
3. 置位 `SIE.STIP`, 重新打开 `S` 态定时器中断

`tick_isr` 执行步骤如下:

1. 调用 `rt_tick_increase` 接入内核
2. 调用 `get_ticks` 并在其返回值的基础上加上到下次触发中断前的 `tick` 值作为参数传递给 `sbi_set_timer`

2.5.2.10 加入符号表解析支持及 `Stacktrace` 支持 (`symbol_analysis.c/h`)

为了方便内核的调试, 我为内核提供了异常发生时的变量符号显示及 `Stacktrace` 功能, 为了实现这些功能, 需要实现符号表的解析功能。

我原本的方案是直接由内核分析 `.symtab` 节来实现, 但实测存在以下两个问题:

1. `.symtab` 节区只存在于 `elf` 文件中, 无法导出到 `binary` 文件中, 并且该节区不是一个具有 `alloc` 属性的节区
2. `.symtab` 节区体积比较大, 且包含一些无用的符号, 这里的分析工作只需要属性为 `OBJECT` (对应于变量) 与 `FUNC` (对应于函数) 以及类型虽为 `NONE` 但带有 `GLOBAL` 的符号 (对应于汇编文件中的标签及链接脚本中的导出符号)

基于如上两个原因, 我编写了一个独立的程序, 该程序在编译脚本中生成 `binary` 文件后被调用, 负责从 `elf` 文件中提取 `.symtab` 节区, 进行结构重构后写入 `binary` 文件的 `.osdebug` 节区中, 该工具的源代码和可执行文件都位于 `tools/osdebugsupport` 中, `Makefile` 会同时生成面向 `elf32` 和 `elf64` 的两个可执行程序 (分别为 `osdebugsupport` 与 `osdebugsupport64`)。

为了实现上述功能, 首先我定义了如下的两个结构体:

```
typedef struct os_symtab_header
{
    size_t function_table_offset; // 函数表的偏移地址 (相对于节区起始地址, 下同)
    size_t function_table_num; // 函数表中的符号数量
    size_t object_table_offset; // 对象表的偏移地址
}
```

```

size_t object_table_num; // 对象表中的符号数量
size_t general_symbol_table_offset; // 一般符号（指代类型虽为NONE但带有GLOBAL的符号）表的偏移地址
size_t general_symbol_table_num; // 一般符号表中的符号数量
size_t string_table_offset; // 字符串表的偏移地址
size_t string_table_size; // 字符串表的大小（字节为单位）
}os_syntab_header;

typedef struct os_syntab_item
{
    size_t name_offset; // 符号名称在字符串表中的偏移地址
    size_t address; // 该符号所代表的地址
    size_t size; // 该符号所代表的大小
}os_syntab_item;

```

.osdebug 节的头部为 os_syntab_header 结构体，这个结构体作为这个节区的头部信息，描述了整个节区的布局（从低地址向高地址方向）：

1. .osdebug 头部（os_syntab_header 结构体）
2. 函数表
3. 对象表
4. 一般符号表
5. 字符串表

这里要求上述五个部分的起始地址都要对齐到机器字长边界（即 size_t 指代的对齐边界），因此.osdebug 节区必须对齐到机器字长边界，否则内核将无法找到节区头部。

函数表、对象表、一般符号表中每一项都是连续排列的，并且是按照符号地址非递减排列的，以便在内核中可以对表进行二分查找，其内存布局由结构体 os_syntab_item 来描述。

osdebugsupport 程序分析 elf 文件中的.syntab 节区，将所有满足上述三种属性的符号分别放置到对应的三个表中，并进行字符串表的构造。

这里为了找到 binary 中的.osdebug 节区相对于 binary 文件的偏移地址，遍历了 elf 中所有具有 alloc 属性的节区，并获得了其中分配地址最小的节区地址（即 sh_addr 属性），然后由.osdebug 节区地址减去上述地址后便可得到所需的偏移地址。

为了实现上述功能，内核的链接文件中必须包含如下节区，实现类似于下面这样：

```

. = ALIGN(8);

.osdebug :
{
    _osdebug_start = .;
    . += 87K;
    _osdebug_end = .;
} > SRAM

```

其中的 87K 即为预分配空间大小，如果节区太小，osdebugsupport 程序会报错，就像下面这样：

```
file_start_offset: 0x80200000
align_bound: 8
function symbol num: 1333
object symbol num: 545
general symbol num: 38
string table size: 31121 Byte(s)
need space: 77176 Byte(s)
.osdebug section space: 8192 Byte(s)
.osdebug section is too small
scons: *** [rtthread.elf] Error 10
```

可以看到，scons 编译时提示 osdebugsupport 程序抛出了错误码 10，也表示了这个问题。
而正确的输出则像下面这样：

```
file_start_offset: 0x80200000
align_bound: 8
function symbol num: 1333
object symbol num: 545
general symbol num: 38
string table size: 31121 Byte(s)
need space: 77176 Byte(s)
.osdebug section space: 89088 Byte(s)
.osdebug in file is from 0x93000
generate finished
```

该程序预定义了一些错误码：

表 2.4： 程序错误码定义表

错误码	含义
0	正常
1	参数数量不正确
2	elf 文件无法被打开
3	binary 文件无法被打开
4	elf 文件读取失败
5	elf 文件头部表示该 elf 文件不是一个 elf 文件
6	elf 文件与该处理程序位数不对应
7	无法在 elf 文件中找到.symtab 节区
8	无法在 elf 文件中找到.osdebug
9	无法在 elf 文件中找到任何可用于定位文件偏移地址的（即具有 alloc 属性的）节区（这个错误不应该发生）

错误码	含义
10	.osdebug 节区太小

其调用格式如下所示：

```
osdebugsupport[64] elf文件路径 binary文件路径
```

例如：

```
osdebugsupport64 rtthread.elf rtthread.bin
```

内核中的 `symbol_analysis.c` 中包含了如下实现接口：

```
os_syntab_item *find_symbol_table(rt_size_t symbol_table_addr,rt_size_t symbol_num,
    rt_size_t address);
const char *get_symbol_name(os_syntab_item *symbol);
void print_symbol(os_syntab_item *symbol,rt_size_t address);
void print_symbol_info(rt_size_t address,rt_bool_t function);
void print_stacktrace(rt_size_t epc,rt_size_t fp);
```

由于函数表、对象表、一般符号表的内部结构完全相同，因此这里提供了 `find_symbol_table` 用于在指定的表中查找某个地址对应的符号的描述结构体指针，其返回值的符号遵循如下原则：

- 该符号的地址必须 \leq 给定的地址
- 该符号的地址必须是在所有符号中最接近该地址的
- 若存在若干个符号地址相同，那么该符号必然是序列编号最小的那一项（以便内核能根据这个符号遍历所有相同地址的符号并全部显示出来）

若找不到满足上述条件的符号，则函数返回空指针（`RT_NULL`）

为了加速查找速度，这里采用二分查找的方式来实现上述功能。

`get_symbol_name` 接口可以根据给定的符号指针从字符串表中找到对应的符号名指针并返回。

`print_symbol` 可以根据给定的符号和地址向终端打印出如下格式的信息（当地址相对于符号的偏移不为 0 时）：

< 符号名: 符号大小 > + 地址相对于符号的偏移
或（当地址相对于符号的偏移为 0 时）
< 符号名: 符号大小 >

`print_symbol_info` 用于打印出一个地址关联的全部符号信息，输入有两个参数，分别表示要检索的地址和检索类型（`function` 为 `RT_TRUE` 表示该地址对应一个指令，`RT_FALSE` 表示该地址对应一个数据）

当地址表示指令时，采用如下的顺序输出符号信息：

1. 检索函数符号表，并打印出所有的关联符号信息（一般来说应当只有一条），由于函数符号具有尺寸信息，这里还要判断地址是否在某个函数符号的地址范围内，如果在，则才认为是命中符号
2. 若第 1 步未找到合法符号，则检索一般符号表，并打印出所有的关联符号信息
3. 若第 1 步未找到合法符号，则检索对象符号表，并打印出所有的关联符号信息（一般来说应当只有一条），由于对象符号具有尺寸信息，这里还要判断地址是否在某个对象符号的地址范围内，如果在，则才认为是命中符号

采用上述的策略是因为指令地址应当先和函数符号表进行匹配，如果匹配失败，说明这是一个汇编函数或者不是函数，因为汇编函数名实际上是个无大小的标号，因此需要在一般符号表中检索，但因为同样这有可能不是一个合法的指令地址，因此同时也检索对象符号表，以方便排查程序。

当地址表示数据时，采用如下的顺序输出符号信息：

1. 检索对象符号表，并打印出所有的关联符号信息（一般来说应当只有一条），由于对象符号具有尺寸信息，这里还要判断地址是否在某个对象符号的地址范围内，如果在，则才认为是命中符号
2. 若第 1 步未找到合法符号，则检索一般符号表，并打印出所有的关联符号信息
3. 若第 1 步未找到合法符号，则检索函数符号表，并打印出所有的关联符号信息（一般来说应当只有一条），由于函数符号具有尺寸信息，这里还要判断地址是否在某个函数符号的地址范围内，如果在，则才认为是命中符号

采用上述的策略是因为数据地址应当先和对象符号表进行匹配，如果匹配失败，说明这是一个汇编符号、链接文件中的导出符号或者是栈、堆中的某个符号，因此需要在一般符号表中检索，但因为同样这有可能指向一条待修改的指令，因此同时也检索函数符号表，以方便排查程序。

如果未找到任何匹配的符号，则该函数会输出 `<Unknown Symbol>` 消息。

`print_stacktrace` 用于在出错时打印出栈跟踪信息，该函数含有两个参数，分别表示发生 `trap` 时的 `epc` 和 `fp` 寄存器值，想要使用这个功能，要求程序必须以 `-O0` 选项编译，否则由于无法通过 `fp` 来追踪栈帧，故而功能无法使用。

这里首先输出 `epc` 对应的符号，然后通过 `fp` 进行栈反向遍历，从每个栈帧中提取上一层的 `fp` 以及当前的返回地址，并逐项输出符号信息，直到当前栈帧不在内存地址范围内或者返回地址为 `0` 时截止。

2.5.3 lwp 适配

这部分的移植共分为以下几大步骤：

- 编译配置文件适配
- `arch_user_space_init` 相关接口适配
- `arch_user_stack` 相关接口适配
- `lwp_gcc` 相关接口适配

2.5.3.1 编译配置文件适配

1. 创建目录 `components/lwp/arch/risc-v/virt64`

2. 在 lwp/Kconfig 文件中为 RT_USING_LWP 增加新的依赖项 ARCH_RISCV64

```
config RT_USING_LWP
    bool "Using light-weight process"
    select RT_USING_DFS
    select RT_USING_LIBC
    select RT_USING_POSIX_CLOCKTIME
    depends on ARCH_ARM_CORTEX_M || ARCH_ARM_ARM9 || ARCH_ARM_CORTEX_A ||
        ARCH_RISCV64
    default n
    help
        The lwp is a light weight process running in user mode.
```

3. 在 lwp/SConscript 中加入 virt64 架构

```
support_arch = {"arm": ["cortex-m3", "cortex-m4", "cortex-m7", "arm926", "cortex-a"],
    "risc-v": ["virt64"]}
```

2.5.3.2 arch_user_space_init 适配 (arch_user_space_init.c)

这里需要适配 arch_user_space_init 与 arch_kernel_mmu_table_get 两个接口：

- arch_user_space_init 负责初始化 lwp 的页表和堆，将堆的截止地址设置为 USER_HEAP_VADDR，这样就能支持自动堆扩展，同时将内核页表的低 4GiB 的部分复制到 lwp 的页表中，这样 syscall 和中断才能被正常处理
- arch_kernel_mmu_table_get 负责返回内核页表指针

2.5.3.3 arch_user_stack 适配 (arch_user_stack.c)

这部分主要需要适配 arch_expand_user_stack，以便支持用户程序栈的自动向下增长：

- 判断给定的 addr 参数是否在用户栈范围内（即 USER_STACK_VSTART 与 USER_STACK_VEND 之间），若在，则调用 lwp_map_user 在给定地址处分配一个页并自动映射，成功返回 1，失败返回 0

2.5.3.4 lwp_gcc 适配 (lwp_gcc.S)

这里主要适配 lwp_user_entry、set_user_context 与 lwp_thread_return 接口：

- lwp_user_entry 接口负责从内核线程转入用户态执行程序，为用户态程序配置正确的栈和返回地址，并为用户态的 libc 初始化提供参数
- set_user_context 接口负责将 lwp_thread_return 接口中保存的线程退出代码拷贝到线程栈上，用于提供用户态程序创建的线程自然退出的支持
- lwp_thread_return 接口负责发起 syscall id 为 1，参数为 0 的系统调用（即 sys_exit），来帮助用户线程的退出

2.5.4 bsp 适配 (bsp/qemu-virt-rv64)

这部分适配共分为以下几大步骤:

- 编译配置文件适配
- 中断控制器驱动适配
- UART 驱动适配
- 板级初始化接口实现
- 中断框架实现、trap 接口适配及加入 stacktrace 支持

2.5.4.1 编译配置文件适配

- 首先需要创建 bsp/qemu-virt-rv64 目录, 在其中创建相关的 SConscript 与 SConstruct, 并正确配置 SConstruct 中的 CPU 与 ARCH 选项
- 配置链接脚本, 这里直接借用 ARM 的链接脚本, 并修改其中的入口点为 0x80200000, 这是 qemu 的 opensbi 引导内核的入口点, 为了支持内核 stacktrace, 加入了如下 section:

```
.osdebug :
{
    _osdebug_start = .;
    . += 87K;
    _osdebug_end = .;
} > SRAM
```

- 配置 Kconfig, 在其中创建 BOARD_virt 项, 让其选择相关依赖项
- 创建 drivers 与 applications 目录, 在 drivers 目录中放置 encoding.h 与 sbi 相关的支持文件, 在 applications 目录中创建 main.c, 并在其中实现 main 函数, 该函数作为 main 内核线程的入口函数

2.5.4.2 中断控制器驱动适配 (driver/plic.c/h):

qemu 所采用的 PLIC 的基地址为 0x0C000000, 这里我借用了 bigmagic 的驱动实现, 我将其移植到了 RT-Thread Smart 上, 实现了如下几个接口:

```
void plic_set_priority(int irq,int priority);//设置中断优先级
void plic_irq_enable(int irq);//启用中断
void plic_irq_disable(int irq);//禁用中断
void plic_set_threshold(int mthreshold);//设置中断触发阈值
int plic_claim(void);//获取中断号
void plic_complete(int irq);//通知PLIC中断处理完成
```


2.5.4.3 UART 驱动适配 (driver/drv_uart.c/h):

这里按照 RT-Thread 驱动模型实现串口设备，并注册为 `uart` 设备，主要需要实现模型所规定的 `drv_uart_putc` 与 `drv_uart_getc` 接口，这里先前的实现方案如下：

- 这两个接口直接调用 `sbi` 提供的接口，同时，由于 `sbi` 未提供中断相关的操作接口，因此此处串口数据接收采取线程轮询接收的方式来实现
- 线程中包含一个无穷循环，该循环每 `10ms` 调用一次 `rt_hw_serial_isr`，让 `rt-thread` 系统内核调用 `drv_uart_getc` 接口，根据返回值判断是否有有效数据，并将有效数据加入数据队列中
- 由于 `UART` 初始化时，系统的线程机制还未完成初始化，因此，该轮询线程延迟到 `main` 线程中被启动

最近借鉴了 `bigmagic` 的最新驱动实现，对驱动进行了改进，了解到了 `qemu` 的 `16550` 控制器的正确用法 (<http://byterunner.com/16550.html>)，最终完成串口中断的支持。

2.5.4.4 板级初始化接口实现 (board.c/h)

板级初始化部分负责初始化基础启动环境，引导内核，并完成内核启动时的初始化工作，这里主要是实现了如下几个接口：

```
void init_bss(); // 初始化内核BSS节区
void primary_cpu_entry(); // 初始化程序入口，由startup_gcc.S调用，负责初始化bss、关中断、调用entry启动RT-Thread Smart内核
void rt_hw_board_init(); // 板级初始化接口，该接口会在RT-Thread Smart系统内核初始化时被调用，负责初始化中断、UART、Tick、堆、控制台设备、板级其它组件，并在最后初始化内核页表
```

以下是 `rt_hw_board_init` 的详细步骤：

1. 调用 `rt_hw_interrupt_init()` 初始化中断框架
2. 调用 `rt_hw_uart_init()` 初始化 `UART` 控制器
3. 调用 `rt_hw_tick_init()` 初始化系统调度定时器
4. 调用 `rt_system_heap_init()` 初始化内核堆，内核堆的范围是由位于 `board.h` 中的 `RT_HW_HEAP_BEGIN` 与 `RT_HW_HEAP_END` 两个宏所决定（这里分配了 `100MiB`）
5. 调用 `rt_console_set_device()` 设置默认控制台设备
6. 调用 `rt_components_board_init()` 完成其它板级组件初始化
7. 调用 `rt_hw_mmu_map_init()` 初始化内核页表的 `4GB` 空间之上的全部区域，即从 `0x1_00000000` 到 `0xFFFFFFFF_FFFFFFFF` 的范围
8. 调用 `rt_page_init()` 初始化 `buddy system` 的分页机制范围，这里依赖 `board.h` 中的两个宏 `RT_HW_PAGE_START` 与 `RT_HW_PAGE_END` 用以确定地址范围（这里分配了 `100MiB`）
9. 调用 `rt_hw_mmu_kernel_map_init()` 完成内核低 `4GB` 地址范围内的虚拟->物理地址等价一对一映射
10. 执行 `switch_mmu` 将 `MMU` 切换到 `Sv39` 地址翻译模式并将上述内核页表挂入 `MMU`

2.5.4.5 中断框架实现、trap 接口适配及加入 stacktrace 支持 (trap.c/h)

这里实现了如下接口：

```
static void rt_default_interrupt_handler(int vector,void *param);//一个默认的中断处理程序
void rt_hw_interrupt_init();//中断框架初始化
void rt_hw_interrupt_mask(int vector);//中断屏蔽
void rt_hw_interrupt_umask(int vector);//中断开放
rt_isr_handler_t rt_hw_interrupt_install(int vector,rt_isr_handler_t handler,void *param,const char *name);//中断处理程序安装
void dump_regs(struct rt_hw_stack_frame *regs);//打印寄存器信息和Stacktrace信息
void handle_trap(rt_size_t scause,rt_size_t stval,rt_size_t sepc,struct rt_hw_stack_frame *sp);//Trap处理入口
```

handle_trap 处理步骤如下：

1. 若 scause 指示这是一个 S 态时钟中断，则调用 tick_isr 进入时钟中断处理程序
2. 若第 1 步不成立且 scause 指示这是一个 S 态外部中断，则从 plic 获取外部中断号并通知 plic 中断处理完成，然后进入对应的外部中断处理程序
3. 若第 2 步不成立，则判断当前是否为中断，若是，则提示 “Unknown Interrupt”
4. 若第 3 步不成立，则判断当前是否为缺页异常，若是则调用 arch_expand_user_stack 尝试进行用户栈扩展
5. 若第 4 步不成立，则打印异常名称、异常寄存器的值，并调用 dump_regs 打印出通用寄存器的值（带符号标注）和 MMU 状态，并打印出 Stacktrace 信息

第 3 章

Kendryte K210 平台适配

3.1 编译运行方式

首先在 `bsp` 目录下的 `rtconfig.py` 中配置正确的工具链路径，然后在 `bsp` 目录中执行 “`compile.sh`”，即可得到 `image.sh`，修改 `run.sh` 中的串口设备文件，然后运行 `run.sh` 脚本即可烧录，然后使用任意串口调试工具即可进行测试。（注意要让 `copmile.sh` 以及 `compile_opensbi.sh` 拥有可执行权限）

3.2 移植综述

该适配代码派生自 `qemu-rv64` 平台，因此这里主要讨论我的移植修改部分。

目前经过测试，**K210** 基本符合 **1.9.1** 版特权指令集规范（实际上是来自于某个已丢失的 **1.9** 版草案标准），该规范和现行规范有一些差异，同时也检查出一些问题，以下是我所检测出的问题：

1. `tlb` 无效化使用 `sfence.vm` 而不是 `sfence.vma` 指令
2. 开启页表机制需要在 `mstatus` 而不是 `sstatus` 中实现，且 `Sv39` 模式的编号为 `9` 而不是现在的 `8`
3. 新标准的 `SUM` 位值为 `1` 时表示允许 `S` 态访问用户页面，而在 **K210** 使用的早期版本中，该位被称作 `PUM`，且值为 `0` 时表示允许 `S` 态访问用户页面
4. 在 **K210** 中，写到未映射页面会引起 `7` 号而不是 `15` 号异常，因为在早期版本中，`15` 号异常未定义
5. **K210** 的 `lr/sc` 指令组合永远失败，怀疑实现有 `bug`，其他 `A` 扩展原子操作指令行为均正常。
6. 在 **K210** 中，存储页表地址的寄存器是 `sptbr` 而不是 `satp`
7. **K210** 的 `Sv39` 物理地址长度为 `50` 位而不是 `56` 位
8. **K210** `M` 态中断不支持向 `S` 态的中断代理机制，需要借助 `S` 态的软中断机制实现
9. 尽管不知道为什么，但似乎在切换页表时只使用 `sfence.vm` 指令，可能会导致进程崩溃（有些指令没有在 `MMU` 切换完成前就访问了内存？不确定），在该指令前后分别依次都加上 `fence` 和 `fence.i` 指令后，通过压力测试，具体出错原因和出错机理不明，可能是 `sfence.vm` 指令实际没有保证前后的访存指令顺序问题？（当年丢失的草案文件已无法找到，不确定当年的具体定义，仅根据 **1.9.1** 版本规范似乎找不到原因），目前来看这种修改方式是有效的

以下的修改主要针对上面几个部分分别进行，大致分为如下几个步骤：

- OpenSBI 适配
- Musl libc 适配
- MMU 适配
- 中断控制器驱动适配
- PLL 驱动适配
- UART 驱动适配
- Tick 驱动适配
- 板级初始化接口实现
- 中断框架实现、trap 接口适配及加入 stacktrace 支持
- SD 卡驱动适配及 FAT32 文件系统适配

3.3 OpenSBI 适配

由于在 1.9.1 标准中，页表机制必须在 M 态开启，因此需要在 OpenSBI 中提前初始化页表机制，同时 K210 不支持向 S 态的代理机制，因此需要借助软中断机制来实现该机制，实现思路如下：

先从 github 上下载 opensbi 的源代码并放置到 bsp 的 opensbi 目录中，然后进入 platform/k210 目录，这里首先要对 platform.c 文件修改，加入上面所说的功能支持，然后要修改 config.mk，来支持内核启动，最后还需要在 bsp 下实现编译脚本。

3.3.1 platform.c 修改

对于 MMU 的问题，我首先分配了一个静态的 MMUTable 数组用来保存四个页表项，然后在 k210_final_init 函数中使用如下代码初始化 MMU 机制：

```
u64 ms = csr_read(mstatus);
ms &= ~(0xf << 24);
ms |= 9 << 24;
csr_write(mstatus, ms);
MMUTable[0] = 0x000002FULL;
MMUTable[1] = 0x100002FULL;
MMUTable[2] = 0x200002FULL;
MMUTable[3] = 0x300002FULL;
csr_write(sptbr, ((u64)MMUTable) >> 12);
```

上述代码首先在 mstatus 寄存器中开启 S 态的 Sv39 地址翻译模式，然后将 MMUTable 的四项（每项对应 1GB 的内存）都设置为叶子节点，且具有 RWX 权限，然后将页表地址写入 sptbr 寄存器，这样就完成了 0x00000000~0xFFFFFFFF 地址范围物理地址到虚拟地址的等价一对一映射，方便内核引导。

3.3.2 K210 中断转发机制的实现

对于 K210 的中断转发机制，我的设计是这样的：

1. 当中断发生时，必然触发 OpenSBI 中的 M 态中断入口，在此先在 mie 寄存器中关掉外部中断（防止进入 S 态之后由于中断未处理导致 PLIC 不释放 pending 标志，引起再度进入 M 态处理中断），然后设置 mip 寄存器中的 S 态软件中断触发位，从而准备触发 S 态软件中断
2. 在 M 态外部中断处理结束后，S 态软件中断紧接着会被触发，在内核中，从 PLIC 获取外部中断号，然后通知 PLIC 中断处理完成，然后在外部中断处理彻底完成之后，向 S 态发起一个系统调用开启 mie 寄存器中的外部中断使能位从而继续接收新的外部中断

那么实际上在 OpenSBI 中有两个工作需要做：

1. 在发生 M 态外部中断的时候，设置 mie 寄存器关掉外部中断，然后设置 mip 寄存器中的 S 态软件中断触发位
2. 为操作 mie 寄存器的外部中断使能位提供相应系统调用

为了实现第一步，我们找到 OpenSBI 源码目录中的 lib/sbi/sbi_trap.c 文件，定位到 sbi_trap_handler 函数，这个函数是 OpenSBI 用来处理 Trap 的入口，在 switch(mcause) 中加入一个入口项 IRQ_M_EXT，并通过以下的代码实现上述的需求：

```
switch (mcause) {
case IRQ_M_TIMER:
    sbi_timer_process();
    break;
case IRQ_M_SOFT:
    sbi_ipi_process();
    break;
case IRQ_M_EXT:
    csr_clear(mie, MIP_MEIP);
    csr_set(sip, SIP_SSIP);
    break;
default:
    msg = "unhandled external interrupt";
    goto trap_error;
};
```

为了实现第二步，我们回到 platform.c，SBI 的系统调用分为两部分：r7 寄存器保存扩展编号，r6 寄存器保存对应扩展中的功能号，由于这个系统调用属于厂家自定义系统调用，这部分系统调用的扩展范围由 SBI_EXT_VENDOR_START 和 SBI_EXT_VENDOR_END 常量决定，即范围为 0x09000000~0x09FFFFFF，我们这里采用 SBI_EXT_VENDOR_START 作为扩展编号。

然后我们要定义两个系统调用：开启外部中断和关闭外部中断，其功能号定义如下：

```
#define SBI_EXT_VENDOR_ENABLE_EXTERNAL_INTERRUPT 0
#define SBI_EXT_VENDOR_DISABLE_EXTERNAL_INTERRUPT 1
```

然后编写对应的处理函数：

```

static int sbi_ecall_vendor_handler(unsigned long extid, unsigned long funcid,
                                     const struct sbi_trap_regs *regs,
                                     unsigned long *out_val,
                                     struct sbi_trap_info *out_trap)
{
    int ret = SBI_OK;

    switch (funcid)
    {
        case SBI_EXT_VENDOR_ENABLE_EXTERNAL_INTERRUPT:
            csr_set(mie, MIP_MEIP);
            csr_set(mstatus, MSTATUS_MIE);
            break;

        case SBI_EXT_VENDOR_DISABLE_EXTERNAL_INTERRUPT:
            csr_clear(mie, MIP_MEIP);
            break;

        default:
            ret = SBI_ENOTSUPP;
            break;
    }

    return ret;
}

```

为了将这两个系统调用注册到 OpenSBI，我们需要定义一个结构体：

```

struct sbi_ecall_extension ecall_vendor = {
    .extid_start = SBI_EXT_VENDOR_START,
    .extid_end = SBI_EXT_VENDOR_START,
    .handle = sbi_ecall_vendor_handler,
};

```

该结构体定义了我们的系统调用扩展起始编号和结束编号，以及处理程序函数指针。

然后在 `k210_final_init` 中调用 `sbi_call_register_extension` 对系统调用进行注册。

最终对于 `k210_final_init` 函数的完整修改如下：

```

static int k210_final_init(bool cold_boot)
{
    void *fdt;

    u64 ms = csr_read(mstatus);
    ms &= ~(0xf << 24);
    ms |= 9 << 24;
}

```

```
    csr_write(mstatus,ms);
    MMUTable[0] = 0x0000002fFULL;
    MMUTable[1] = 0x1000002fFULL;
    MMUTable[2] = 0x2000002fFULL;
    MMUTable[3] = 0x3000002fFULL;
    csr_write(sptbr,((u64)MMUTable) >> 12);

    sbi_ecall_register_extension(&ecall_vendor);

    if (!cold_boot)
        return 0;

    fdt = sbi_scratch_thishart_arg1_ptr();
    fdt_cpu_fixup(fdt);
    fdt_fixups(fdt);

    return 0;
}
```

3.3.3 config.mk 修改

OpenSBI 中含有很多种 firmware, 这些 firmware 支持以不同的方式引导内核, 这里我们选用 FW_JUMP 固件, 并从 0x80030000 处引导内核 (为 OpenSBI 预留 192KB 的空间):

```
FW_JUMP=y
FW_JUMP_ADDR=0x80030000
```

3.3.4 实现编译脚本

为了实现 opensbi 的编译, 我在 bsp 目录下编写了一个 compile_opensbi.sh 脚本, 该脚本内容如下:

```
#!/bin/bash

pushd opensbi/opensbi
export CROSS_COMPILE=riscv64-unknown-elf-
export PLATFORM_RISCV_XLEN=64
make PLATFORM=kendryte/k210 O=../
popd
```

该脚本首先进入 OpenSBI 源码目录, 然后设置编译工具链前缀和 RISC-V 架构位数, 然后指定为 K210 架构并指定编译结果文件生成目录, 实际为 bsp 目录下的 opensbi 目录。

在编译后, 会在 opensbi 目录生成 lib 和 platform 两个目录, 目前对我们有用的是 platform/kendryte/k210/fw_jump.bin 文件, 这个在后期要和内核拼接在一起。

为了实现最终的烧录镜像编译，我们编写了 `compile.sh` 编译脚本，该脚本内容如下：

```
#!/bin/bash

./compile_opensbi.sh && scons && touch image.bin && dd if=/dev/null of=image.bin &&
dd if=opensbi/platform/kendryte/k210/firmware/fw_jump.bin of=image.bin && dd if=
rtthread.bin of=image.bin bs=1 seek=196608
```

该脚本首先调用 `compile_opensbi.sh` 编译 OpenSBI，然后建立一个空白的 `image.bin` 用来保存烧录镜像内容，紧接着将 `fw_jump.bin` 写入到镜像头部，并将内核文件写入到镜像 192KB 的偏移地址处。

为了方便烧录，我同时编写了烧录脚本，即 `bsp` 目录下的 `run.sh`，内容如下：

```
#kflash -p /dev/ttyUSB0 -b 1500000 -t image.bin
kflash -p /dev/ttyS17 -b 1500000 image.bin
```

可以注意到上面有一个注释行，实际上如果在 Linux 虚拟机中编译，可以用 `-t` 参数直接启动终端进行测试，但是在 `ws1` 中，该烧录脚本指定的终端启动后，键盘输入无响应，因此为了在 `ws1` 下正常使用，暂时去掉 `-t` 标志，并通过其他串口调试工具进行调试。

`-p` 参数后面就是烧录串口设备文件路径，`-b` 参数后面则是烧录波特率，最后指定烧录镜像文件即可。

尽管目前没有使用到 `sbi` 提供的 `lib` 库，但是考虑到未来可能使用，因此这里在 `rtconfig.py` 中修改 `LFLAGS` 参数，加入 `sbi` 相关的链接库：

```
LFLAGS = DEVICE + ' -nostartfiles -Wl,--gc-sections,-Map=rtthread.map,-cref,-u,
_start -T link.lds -lc -lm -lsbi -lsbiutils -lplatsbi -Lopensbi/lib -Lopensbi/
platform/kendryte/k210/lib '
```

同时为了方便内核调用新增的 `sbi` 系统调用，需要修改 `bsp` 目录下的 `driver/sbi.h` 文件，加入如下两条定义：

```
static __inline void sbi_vendor_enable_external_interrupt()
{
    SBI_CALL0(SBI_EXT_ID_VENDOR, SBI_EXT_VENDOR_ENABLE_EXTERNAL_INTERRUPT);
}

static __inline void sbi_vendor_disable_external_interrupt()
{
    SBI_CALL0(SBI_EXT_ID_VENDOR, SBI_EXT_VENDOR_ENABLE_EXTERNAL_INTERRUPT);
}
```


3.3.5 Musl libc 适配

这里主要是针对 K210 的原子操作指令问题进行修改，将库中所有使用到 `lr/sc` 指令对的地方全部换成了 `amo` 开头的其它原子操作指令，从而修复诸如 `malloc` 调用时卡死的问题。

3.3.6 MMU 适配

这里需要修改 `libcpu/risc-v/k210_mmu/riscv_mmu.c` 文件，将 `mmu_set_pagetable` 改为将页表地址写入 `sptbr`，同时将 `mmu_enable_user_page_access` 于 `mmu_disable_user_page_access` 的代码实现调换即可。

然后修改 `libcpu/risc-v/k210_mmu/riscv_mmu.h` 文件，修改物理地址长度宏 `PHYSICAL_ADDRESS_WIDTH_BITS` 为 50，然后将 `mmu_flush_tlb()` 宏修改为如下定义：

```
#define mmu_flush_tlb() do{asm volatile("fence");asm volatile("fence.i");asm volatile("sfence.vu x0");asm volatile("fence");asm volatile("fence.i");}while(0)
```

3.3.7 中断控制器驱动适配

为了实现中断控制器驱动，首先我们要下载 K210 的 SDK：在 `bsp/k210-smart` 目录中执行 `scons --menuconfig`，然后依次进入 `RT-Thread online packages -> peripheral libraries and drivers`，选中 `the kendryte-sdk package for rt-thread`，并进入，将版本设置为 `v0.5.6`，然后保存退出，紧接着执行 `pkgs --update` 进行包的更新。

完成后进入本次移植 `bsp` 目录下的 `packages` 目录中，为了保证源代码的正常上传，需要删除该目录及所有子目录中的 `.git` 目录，同时修改 `rt-thread` 目录下的 `.gitignore`，去掉对 `packages` 目录的过滤。

然后我们要修改 `lib` 中的部分头文件，去掉对 K210 原来 SDK 的依赖，并转向对 RT-Thread Smart 内核的依赖。

为了适配 PLIC，我们需要打开 SDK 中 `lib/drivers` 目录中的 `plic.c` 文件，加入 `rtthread.h`、`sbi.h` 与 `trap.h` 头文件，因为本次移植不支持多核，直接定义一个 `current_coreid()` 宏，值为 0。

接着修改 `plic_init` 函数，在函数最后去掉对 `mie` 寄存器的中断开关写入操作，转而调用 `sbi_vendor_enable_external_interrupt` 打开中断。

为了将 PLIC 接入 RT-Thread Smart 中断框架，需要增加一个 `plic_interrupt_agent` 函数作为 PLIC 和中断框架之间的中断处理程序代理程序，该代理程序负责调用注册在 PLIC 中真正的处理程序（因为 K210 SDK 中的众多驱动程序依赖 PLIC 的中断处理程序注册接口，因此才做此兼容设计），然后在 `plic_irq_register` 最后调用 `rt_hw_interrupt_install` 函数在 RT-Thread Smart 中断框架中为给定的中断号绑定调用代理接口。

为了解决 `sysctl` 对 `usleep` 函数的依赖，由于执行这里的代码时 RT-Thread Smart 调度器还未启动，因此暂时不能使用 RT-Thread Smart 的延时函数，这里实现一个简单的 `usleep` 用以满足 `sysctl` 的延时需求：

```
void usleep(int x)
{
    volatile int i = 0;

    for(i = 0; i < 10000; i++);
}
```

然后修改 SDK 下的 SConscript，配置所需的编译源代码文件。

3.3.8 PLL 驱动适配 (driver/drv_pll.c/h)

这里实现了两个接口，分别为 drv_pll_get_cpu_freq 和 rt_hw_pll_init，分别用于获取 CPU 的频率和初始化 PLL。

这里将 PLL0 设置为 320MHz、PLL1 设置为 160MHz、PLL2 设置为 45.1584MHz，这部分初始化主要参考的 K210 官方的驱动程序。

PLL0 作为芯片的主要时钟，提供给 CPU 和除 I2S 外的各个外设，输入是外部的 26MHz 时钟。

PLL1 输出时钟作为 AI 的工作时钟，输入是外部的 26M 时钟。

PLL2 输出时钟作为 I2S 的工作时钟，输入有三个选择：外部的 26M 时钟或者 PLL0/1 的输出时钟。

3.3.9 UART 驱动适配 (drivers/drv_uart.c/h)

这里按照标准 RT-Thread 驱动框架的方式实现，改动分别如下：

- rt_hw_uart_init 中在注册串口设备之前调用 uarths_init 对串口初始化以适应 PLL 频率改动，并在注册后注册串口接收中断
- 修改了 drv_uart_putc 和 drv_uart_getc，令其分别指向 uarths_putchar 和 uarths_getchar 函数
- 修改了调试用的 drv_uart_puts 函数，其中改为调用 uarths_putchar

3.3.10 Tick 驱动适配 (libcpu/risc-v/k210_mmu/tick.c)

这里修改了 rt_hw_tick_init 中的 tick_cycles 计算方式，改为如下计算方式：

```
tick_cycles = interval * sysctl_clock_get_freq(SYSCTL_CLOCK_CPU) / CLINT_CLOCK_DIV /
1000ULL - 1;
```

3.3.11 板级初始化接口实现

这里在 rt_hw_board_init 函数中，rt_hw_interrupt_init 调用之前加入了 rt_hw_pll_init，来保证在其他硬件初始化前首先初始化 PLL，保证之后依赖 PLL 频率的硬件都能正确初始化。

3.3.12 中断框架实现、trap 接口适配及加入 stacktrace 支持

这里有如下改动：

- 在 `rt_hw_interrupt_init` 的最后加入 `plic_init` 函数调用，从而正确初始化 `plic`，同时调用 `set_csr(sie, SIP_SSIP)` 打开 S 态的软件中断使能以便接收 OpenSBI 转发的外部中断
- 修改了 `dump_regs` 中 MMU 信息的输出部分，将输出寄存器从 `satp` 改为 `sptbr`，同时针对 K210 的 MMU 结构调整了输出字段
- 在 `handle_trap` 中将处理外部中断的分支的 `scause` 指向 S 态软件中断，并将实现改为如下形式（即先清除软件中断挂起信号，然后从 PLIC 获取外部中断号并通知 PLIC 完成，接下来处理中断，最后调用 `sbi_vendor_enable_external_interrupt` 重新开放外部中断接收）：

```
else if(scause == (uint64_t)(0x8000000000000001))
{
    clear_csr(sip, SIP_SSIP);
    int plic_irq = (int)plic_irq_claim();
    plic_irq_complete(plic_irq);
    rt_interrupt_enter();
    irq_desc[plic_irq].handler(plic_irq, irq_desc[plic_irq].param);
    rt_interrupt_leave();
    sbi_vendor_enable_external_interrupt();
}
```

- 由于 K210 无缺页异常，发生缺页时同样是触发 7 号异常（Store/AMO Access Fault），因此栈扩展相关分支改为如下形式（将缺页异常和 Store 异常合并处理，根据扩展成功与否判断如何处理异常）：

```
#ifdef RT_USING_USERSPACE
    if(id == 7)
    {
        if(arch_expand_user_stack((void *)stval))
        {
            return;
        }
    }
#endif
```

3.3.13 SD 卡驱动适配及 FAT32 文件系统适配

为了适配 SD 卡驱动，首先在 `bsp` 目录下执行 `scons --menuconfig`，依次进入 RT-Thread Components -> Device virtual file system，在其中将 The maximal number of mounted file system 和 The maximal number of file system type 都修改为 4，然后启用 Enable elem-chan fatfs 选项，进入该选项，将 OEM code page 设置为 936 以支持中文，然后启用 Using

RT_DFS_ELM_WORD_ACCESS 选项并将 Support long file name 设置为 3: LFN with dynamic LFN working buffer on the heap 以便启用长文件名支持，并保证 Enable the reentrancy 以保证线程安全。

然后进入 RT-Thread Components -> Device Drivers, 启用 Using SPI Bus/Device device drivers 选项，并启用 Using SD/TF card driver with spi 选项，然后保存退出。

然后在 driver/drv_spi.c/h 中调用 K210 SDK 中的 spi 接口函数实现 SPI 驱动适配，相关实现可以参见源码，这里不再赘述。

紧接着实现 drv_sdcard.c, 里面实现了一个 rt_hw_sdcard_init, 并用 INIT_COMPONENT_EXPORT (rt_hw_sdcard_init) 进行了注解，表示这是一个板级组件初始化程序，将会由 RT-Thread Smart 内核在合适的初始化时机自行调用。

我所使用的是 M5Stack 的开发板，SCLK、SPI0_D0、SPI0_D1、CS 分别对应 30、33、31、32 号 IO，由于 SD 卡协议的特殊性，CS 需要手工控制，因此将 CS 绑定到 GPIOHS7（即高速 GPIO 的 7 号 IO），并调用 GPIOHS 的接口将该 IO 初始化为输出模式，且输出高电平。

然后和配置好初始化项，在系统中初始化一个合适的 SPI 设备，并初始化一个 SD 卡设备。

接着在 bsp 目录下的 applications/mnt.c 中 romfs_mount 函数中加入一个挂载项：

```
dfs_mount("sdcard", "/mnt", "elm", 0, RT_NULL);
```

为了保证挂载顺利进行，这里需要进入镜像根目录中的 userapps 目录中，在 root 目录中建立一个空的 mnt 目录，并修改 write_rom.sh 脚本让该脚本指向 k210-smart bsp 目录并执行该脚本，做这一步是为了让根文件系统下拥有 mnt 挂载点。

由于 RT-Thread Smart 自带的 SD 卡驱动与 K210 控制器有一定的不兼容问题，因此这里对系统内的 SD 卡驱动也进行了修改，修改的文件存在于 components/drivers/spi/spi_msdc.c 文件中。

不兼容的原因是该驱动假定 SPI 控制器可以同时进行读写操作，因此这里针对这个问题修改了若干处，其中一部分代码甚至借助在发送缓冲区中填入 FF 来延时，但是在 K210 中，这会造成对应字节时间本应接收的数据丢失。

第 4 章

全志 D1 平台适配

4.1 编译运行方式

首先在 `bsp` 目录下的 `rtconfig.py` 中配置正确的工具链路径，然后在 `bsp` 目录中执行 “`scons -jN`” (`N` 表示并行数)，即可得到 `rtthread.bin`，运行 `generateimg.sh` 即可生成可用于 `uboot` 启动的 `rtthread.img`，使用支持 Y-Modem 协议的串口调试工具（如 `XShell`）连接开发板，启动时长按 `1` 进入 `uboot`，然后执行 “`loady 45000000;bootm 45000000`”，然后通过 Y-Modem 协议传输镜像后系统即可启动。

4.2 移植综述

该适配代码派生自 `qemu-rv64` 平台，因此这里主要讨论我的移植修改部分。

以下的修改大致分为如下几个步骤：

- MMU 适配
- UART 驱动适配
- Tick 驱动适配

4.3 MMU 适配

这里需要修改 `libcpu/risc-v/k210_mmu/risv_mmu.h` 文件，全志 D1 采用的 C906 核对页表项定义进行了扩展，加入了如下几个位：

- 59 - SEC 位
- 60 - Shareable 位
- 61 - Bufferable 位
- 62 - Cacheable 位
- 63 - Strong Order 位

由于目前没有相关设计文档，因此上面字段的若干位含义不明，目前猜测 `Cacheable` 指示对应的页是否可被 `Cache`，`Strong Order` 则是用于设备映射页，用来保证访问满足 `Strong Order` 访存顺序要求。目前只能参照 C906 测试版本的 RT-Thread 实现，修改 `PAGE_DEFAULT_ATTR_LEAF` 与 `PAGE_DEFAULT_ATTR_NEXT` 宏，加入如下选项：

```
/* C-SKY extend */
#define PTE_SEC    (1UL << 59) /* Security */
#define PTE_SHARE  (1UL << 60) /* Shareable */
#define PTE_BUF    (1UL << 61) /* Bufferable */
#define PTE_CACHE  (1UL << 62) /* Cacheable */
#define PTE_SO     (1UL << 63) /* Strong Order */

#define PAGE_DEFAULT_ATTR_LEAF (PAGE_ATTR_RWX | PAGE_ATTR_USER | PTE_V | PTE_G |
    PTE_SHARE | PTE_BUF | PTE_CACHE | PTE_A | PTE_D)
#define PAGE_DEFAULT_ATTR_NEXT (PAGE_ATTR_NEXT_LEVEL | PTE_V | PTE_G | PTE_SHARE |
    PTE_BUF | PTE_CACHE | PTE_A | PTE_D)
```

目前尚不清楚为什么要加上 `PTE_A` 和 `PTE_D` 选项，但实测不加就会导致 MMU 映射无效。

然后要修改 `mmu.c` 中的 `rt_hw_mmu_kernel_map_init` 函数，修改为如下内容：

```
rt_size_t paddr_start = __UMASKVALUE(VPN_TO_PPN(vaddr_start, mmu_info -> pv_off),
    PAGE_OFFSET_MASK);
rt_size_t va_s = GET_L1(vaddr_start);
rt_size_t va_e = GET_L1(vaddr_start + size - 1);
rt_size_t i;

for(i = va_s; i <= va_e; i++)
{
    mmu_info -> vtable[i] = COMBINEPTE(paddr_start, PAGE_ATTR_RWX | PTE_G | PTE_V |
        PTE_CACHE | PTE_SHARE | PTE_BUF | PTE_A | PTE_D);
    paddr_start += L1_PAGE_SIZE;
}

rt_hw_cpu_tlb_invalidate();
```

即为每个页表项预先加上额外的 `PTE_CACHE | PTE_SHARE | PTE_BUF | PTE_A | PTE_D` 属性。

然后要在 `bsp` 目录下的 `board.c` 中的 `rt_hw_board_init` 函数中，在执行完 `rt_hw_mmu_kernel_map_init` 函数之后，要为低 1GB 设备 MMIO 映射区去掉 `PTE_CACHE` 与 `PTE_SHARE` 属性，同时加上 `PTE_SO` 属性，以保证设备寄存器的正常访问。

4.4 UART 驱动适配 (drivers/drv_uart.c/h)

由于目前对于全志 D1 的 UART 控制器及 PLIC 控制器相关的控制细节不清楚，因此这里采用线程轮询的方式来实现串口接收功能，实现方案如下：

- `drv_uart_putc` 直接调用 `sbi` 提供的接口
- 由于 `sbi` 中的串口接收驱动存在 BUG，无法正确检测 UART 控制器是否真的收到了有效数据，导致将无效数据重复返回，因此这里借鉴 Linux 中全志 D1 的驱动，单独实现 `drv_uart_getc` 接口
- 线程中包含一个无穷循环，该循环每 10ms 调用一次 `rt_hw_serial_isr`，让 `rt-thread` 系统内核调用 `drv_uart_getc` 接口，根据返回值判断是否有有效数据，并将有效数据加入数据队列中
- 由于 UART 初始化时，系统的线程机制还未完成初始化，因此，该轮询线程延迟到 `main` 线程中被启动

4.5 Tick 驱动适配 (libcpu/risc-v/t-head/c906/tick.c)

由于目前对于全志 D1 的各种外设细节不清楚，因此这里暂定 `tick_cycles` 的值为 40000。

第 5 章

DreamOS 内核的相关说明（我自己的内核）

5.1 设计综述

在移植 RT-Thread 的整个过程中，我学到了很多设计思想和设计理念，为了整理之，开始尝试编写自己的内核。

该内核采用 C 语言编写，目前会尽量同步在 github 仓库更新：https://github.com/lizhirui/dreamos_cversion。

该内核的名字叫做 DreamOS，采用 arch/bsp/firmware 分离的方式组织代码，并采用了 scons 作为编译系统，借用了 RT-Thread 所采用的 scons 编译脚本。

以下对于接口实现的文档仅为简要文档，实际的接口参数说明和接口实现参见源代码，内核相关函数都已使用 Doxygen 文档格式进行注释标记

内核相关的文件名和导出函数名均以 `os_` 开头，并依照所属模块进行命名

目录结构如下：

表 5.1: DreamOS 源代码目录结构

目录名	用途
arch	CPU 指令集架构相关代码
bsp	特定 CPU 板级平台代码
firmware	固件操作代码
include	内核公共头文件引入区
src	内核源代码文件
tools	存放编译脚本和各种工具

5.2 内核设计

src 目录下包含如下几个目录及文件：

表 5.2: `src` 目录文件用途

文件名	用途
<code>memory</code>	内存分配算法目录
<code>vfs</code>	虚拟文件系统目录
<code>os_annotation.c</code>	内核注解实现
<code>os_bitmap.c</code>	位图实现
<code>os_debug.c</code>	内核调试功能实现
<code>os_device.c</code>	设备管理实现
<code>os_elf.c</code>	ELF 加载实现
<code>os_file.c</code>	文件操作实现
<code>os_hashmap.c</code>	hashmap 实现
<code>os_init.c</code>	内核初始化实现
<code>os_interrupt.c</code>	中断控制实现
<code>os_io.c</code>	控制台操作实现
<code>os_memory.c</code>	内存管理实现
<code>os_mmu.c</code>	MMU 子系统实现
<code>os_mutex.c</code>	互斥锁实现
<code>os_string.c</code>	一些常用内存和字符串操作函数
<code>os_syscall.c</code>	系统调用相关和系统调用实现
<code>os_task.c</code>	任务调度实现
<code>os_terminal_color</code>	控制台颜色控制实现
<code>os_tick.c</code>	内核 Tick 相关功能实现
<code>os_vfs.c</code>	VFS 实现
<code>os_waitqueue.c</code>	等待队列实现

其中，`src/memory` 目录中包含如下几个文件：

表 5.3: `src/memory` 目录文件用途

文件名	用途
<code>os_memory_page</code>	Buddy System 页面分配算法实现
<code>os_memory_slub</code>	Slub 内存分配算法实现

而 `src/vfs` 目录中则包含如下几个文件：

文件名	用途
os_vfs_devfs.c	设备文件系统实现
os_vfs_romfs	只读文件系统实现

内核同时导出了如下几个接口：

```
void arch_task_switch(task_t *old_task,task_t *new_task);//用于arch实现任务切换
void arch_task_stack_frame_init(task_t *task);//用于arch实现初始化任务栈帧
void bsp_early_init();//用于进入内核时的初始化
void bsp_after_heap_init();//用于完成堆分配后的初始化
void bsp_after_task_scheduler_init();//用于调度器完成初始化之后的初始化
void bsp_puts(const char *str);//用于向控制台打印字符串
bool_t bsp_interrupt_disable();//用于中断屏蔽
void bsp_interrupt_enable(bool_t enabled);//用于中断开启
bool_t bsp_interrupt_handler(enum interrupt_type interrupt_type);//用于中断处理
```

下面开始逐文件进行介绍。

5.2.1 os_annotation.c

该文件用于内核的注解处理，所谓注解功能是指，在函数中放置一些标记用于环境检查，当环境检查不通过时抛出异常（如某些函数要求 VFS 完成初始化，某些函数要求运行在线程上下文下）。

该文件中仅实现了一个注解违例处理程序，接口如下：

```
void os_annotation_handler(const char *ex_string,const char *func,os_size_t line,
    const char *error_msg);//注解违例处理程序
```

同时在 os_annotation.h 中声明了几个现有的注解：

```
//功能未实现
#define OS_ANNOTATION_NOT_IMPLEMENT() OS_ANNOTATION(OS_FALSE,"This function isn't implemented!")
//需要任务上下文
#define OS_ANNOTATION_NEED_TASK_CONTEXT() OS_ANNOTATION((!os_is_in_interrupt()) && (os_task_scheduler_is_initialized()) && (os_task_get_current_task() != OS_NULL),"This function must be called in thread context!")
//需要非中断上下文（不一定是任务上下文，还可能是启动第一个任务之前的初始上下文）
#define OS_ANNOTATION_NEED_NON_INTERRUPT_CONTEXT() OS_ANNOTATION(!os_is_in_interrupt(),"This function can't be called in interrupt context!")
//需要中断上下文
#define OS_ANNOTATION_NEED_INTERRUPT_CONTEXT() OS_ANNOTATION(os_is_in_interrupt(),"This function must be called in interrupt context!")
```

```
//需要内存子系统完成初始化
#define OS_ANNOTATION_NEED_DYNAMIC_MEMORY() OS_ANNOTATION(os_memory_is_initialized()
    ,"This function can't be called until dynamic memory system is initialized
    correctly!")
//需要任务调度器完成初始化
#define OS_ANNOTATION_NEED_TASK_SCHEDULER() OS_ANNOTATION(
    os_task_scheduler_is_initialized(),"This function can't be called until task
    scheduler is initialized correctly!")
//需要MMU子系统完成初始化
#define OS_ANNOTATION_NEED_MMU() OS_ANNOTATION(os_mmu_is_initialized(),"This
    function can't be called until mmu is initialized correctly!")
//需要MMU完成预初始化
#define OS_ANNOTATION_NEED_MMU_PREINIT() OS_ANNOTATION(os_mmu_is_preinitialized(),"
    This function can't be called until mmu is preinitialized correctly!")
//需要VFS完成初始化
#define OS_ANNOTATION_NEED_VFS() OS_ANNOTATION(os_vfs_is_initialized(),"This
    function can't be called until vfs is initialized correctly!")
//需要设备管理器完成初始化
#define OS_ANNOTATION_NEED_DEVICE() OS_ANNOTATION(os_device_is_initialized(),"This
    function can't be called until device is initialized correctly!")
```

其中，OS_ANNOTATION 宏的定义如下：

```
#define OS_ANNOTATION(condition,error_msg)
\
do
\
{
\
    if (!(condition))
\
    {
\
        os_annotation_handler(#condition,__FUNCTION__,__LINE__,error_msg);
\
    }
\
}while(0)
```

上述宏用于提示注解违例的代码行以及相应的错误信息。

5.2.2 os_bitmap.c

该文件实现了完整的位图支持，主要用于各类资源分配的场所，其实现了如下接口：

```
os_err_t os_bitmap_create(os_bitmap_p bitmap,os_size_t size,void *memory,os_size_t
    default_value);//创建一个位图
void os_bitmap_remove(os_bitmap_p bitmap);//销毁一个位图，该位图的内存若为自动分配，
    则会自动销毁，否则需要由用户自行处理
os_size_t os_bitmap_get_bit(os_bitmap_p bitmap,os_size_t id);//获取位图的指定位
void os_bitmap_set_bit(os_bitmap_p bitmap,os_size_t id,os_size_t value);//设置位图的
    指定位
os_size_t os_bitmap_find_some_ones(os_bitmap_p bitmap,os_size_t start_id,os_size_t
    count);//找到位图中连续的若干个1，并返回起始位号（从0开始）
os_size_t os_bitmap_find_some_zeros(os_bitmap_p bitmap,os_size_t start_id,os_size_t
    count);//找到位图中连续的若干个0，并返回起始位号（从0开始）
void os_bitmap_set_bits(os_bitmap_p bitmap,os_size_t start_id,os_size_t count,
    os_size_t value);//批量设置位图的某些位为指定值
```

当位图创建时，如果 `memory` 参数为 `OS_NULL`，则会自动分配内存，并在 `remove` 时自动销毁该内存。

位图的大小恒为 2 的幂。

5.2.3 os_debug.c

该文件实现了符号查找的支持和 `Stacktrace` 支持，详情可见 RT-Thread Smart Qemu-rv64 平台移植中关于 `Stacktrace` 的相关实现说明。

该文件导出了如下接口：

```
os_syntab_item *find_symbol_table(size_t symbol_table_addr,size_t symbol_num,size_t
    address);
const char *get_symbol_name(os_syntab_item *symbol);
void print_symbol(os_syntab_item *symbol,size_t address);
void print_symbol_info(size_t address,bool_t function);
void print_stacktrace(size_t epc,size_t fp);
```

该文件重点实现了一个用于编译器的断言宏，该宏实现如下：

```
#define OS_BUILD_ASSERT(condition) (((void)sizeof(char[1 - 2 * (!(condition)])))
```

实现该宏的原因在于，有时需要执行一些检查，当检查未通过时提醒用户并且不允许代码编译通过。通常的做法是采用 `#ifdef` 与 `#error` 等宏处理子句，但是预处理器不支持一些特殊的常量表达式，如含有 `sizeof` 的常量表达式，而这些表达式虽然在预处理阶段无法计算，但是在编译期却可以被计算出来，因此我编写的宏将检查下放到了编译期，其原理是：

这实际是一个 `sizeof` 语句, `sizeof` 的对象是一个 `char` 数组, 而 `char` 数组的大小则由表达式 “`1 - 2 * (!(condition))`” 决定, 当断言条件成立时, `!(condition)` 的值为 `0`, 则上述表达式的值就为 `1`, 此时无任何语法错误, 即可编译通过, 但是, 若断言条件不成立时, `!(condition)` 的值为 `1`, 则上述表达式的值就为 `-1`, 而数组大小不允许为负数, 此时编译器就会抛出错误, 引导用户去检查。

5.2.4 os_device.c

该文件实现了设备模型和设备管理相关的接口, 目前系统的所有设备均由下面的结构体描述:

```
struct os_device
{
    char *name; //设备名
    os_bool_t initialized; //指示设备是否已初始化
    os_size_t flag; //设备标志
    os_size_t open_flag; //设备打开标志, 参见OS_FILE_FLAG_相关标志
    os_size_t refcnt; //引用数
    os_mutex_t lock; //锁
    void *priv_data; //私有数据
    os_list_node_t node; //列表节点
    os_device_ops_p ops; //设备操作函数集结构体指针
};
```

其中, 设备操作函数集结构体定义如下:

```
//设备操作函数集结构体
typedef struct os_device_ops
{
    os_err_t (*init)(os_device_p dev); //设备初始化
    os_err_t (*open)(os_device_p dev, os_size_t open_flag); //设备打开
    os_err_t (*close)(os_device_p dev); //设备关闭
    os_err_t (*read)(os_device_p dev, void *buf, os_size_t pos, os_size_t size); //设备
        读取
    os_err_t (*write)(os_device_p dev, const void *buf, os_size_t pos, os_size_t size);
        //设备写入
    os_err_t (*ioctl)(os_device_p dev, os_size_t cmd, os_size_t arg); //设备ioctl
}os_device_ops_t, *os_device_ops_p;
```

同时提供了如下接口:

```
os_list_node_p os_device_get_list(); //获取设备列表
os_device_p os_device_find(const char *name); //从设备列表中寻找一个设备
os_err_t os_device_register(os_device_p dev); //设备注册
os_err_t os_device_op_init(os_device_p dev); //调用设备的初始化函数
os_err_t os_device_op_open(os_device_p dev, os_size_t open_flag); //调用设备的打开函数
os_err_t os_device_op_close(os_device_p dev); //调用设备的关闭函数
```

```

os_err_t os_device_op_read(os_device_p dev,void *buf,os_size_t pos,os_size_t size);
    //调用设备的读取函数
os_err_t os_device_op_write(os_device_p dev,const void *buf,os_size_t pos,os_size_t
    size);//调用设备的写入函数
os_err_t os_device_op_ioctl(os_device_p dev,os_size_t cmd,os_size_t arg);调用设备的
    ioctl函数
os_bool_t os_device_is_initialized();//判断设备管理器是否已经初始化完成
void os_device_init();//初始化设备管理器

```

5.2.5 os_elf.c

该文件实现了 ELF 的识别和加载，用于 `sys_execve` 系统调用加载用户态应用程序时用，实现了如下接口：

```

os_err_t os_elf_load(const char *path,os_size_t *entry);//加载ELF，执行该函数会同时
    导致为任务创建新的页表和文件符号表

```

5.2.6 os_file.c

该文件是 VFS 的一部分，实现了文件操作支持，定义了如下结构体：

```

//文件状态结构体
typedef struct os_file_state
{
    os_size_t type;//类型
    os_size_t mode;//模式
    os_size_t size;//大小
    os_time_t access_time;//最后访问时间
    os_time_t modification_time;//最后修改时间
    os_time_t creation_time;//最后创建时间
}os_file_state_t,*os_file_state_p;

typedef struct os_file_fd os_file_fd_t,*os_file_fd_p;

//目录项结构体
typedef struct os_dirent
{
    char name[OS_VFS_PATH_MAX + 1];//名称
    os_size_t type;//类型
}os_dirent_t,*os_dirent_p;

//文件操作函数集结构体
typedef struct os_file_ops
{

```

```

os_err_t (*open)(os_file_fd_p fd); // 文件打开
os_err_t (*close)(os_file_fd_p fd); // 文件关闭
os_err_t (*ioctl)(os_file_fd_p fd, os_size_t cmd, os_size_t arg); // 文件ioctl
os_err_t (*read)(os_file_fd_p fd, void *buf, os_size_t size); // 文件读取
os_err_t (*write)(os_file_fd_p fd, const void *buf, os_size_t size); // 文件写入
os_err_t (*flush)(os_file_fd_p fd); // 文件flush
os_err_t (*lseek)(os_file_fd_p fd, os_size_t offset); // 文件读写指针调整
os_err_t (*readdir)(os_file_fd_p fd, os_dirent_p entry, os_size_t count); // 读取目
    录项
}os_file_ops_t, *os_file_ops_p;

// 文件节点结构体
typedef struct os_file_node
{
    char path[OS_VFS_PATH_MAX + 1]; // 文件路径
    os_size_t type; // 类型
    os_size_t size; // 大小
    os_vfs_mp_p mp; // 关联的挂载点结构体指针
    os_file_ops_p ops; // 文件操作函数集结构体指针
    os_size_t refcnt; // 引用数
    void *priv_data; // 私有数据
    os_list_node_t node; // 列表节点
    os_mutex_t lock; // 锁
}os_file_node_t, *os_file_node_p;

// 文件描述符结构体
struct os_file_fd
{
    os_file_node_p fnode; // 关联的文件节点结构体指针
    os_size_t open_flag; // 打开标志
    os_size_t pos; // 当前读写指针位置
    os_list_node_t node; // 列表节点
    os_mutex_t lock; // 锁
};

// 文件描述符表结构体
typedef struct os_file_fd_table
{
    os_bitmap_t fd_bitmap; // 文件描述符位图
    os_hashmap_t fd_hashmap; // 文件描述符哈希表
    os_list_node_t fd_list; // 文件描述符列表
    os_size_t refcnt; // 引用数
    os_mutex_t lock; // 锁
}os_file_fd_table_t, *os_file_fd_table_p;

```

其中，每打开一个文件，就会创建一个新的文件描述符，但是对于同一个文件，至多只会创建一个文件节点，也就是说，多个文件描述符可能关联同一个文件节点，因此，文件的一些私有数据（包括路径、所属文件系统）等信息都隶属于文件节点，而读写指针、文件打开标志等则隶属于文件描述符。

为了实现用户态程序支持，需要为每一个文件描述符分配唯一 `id`，因此，采用位图进行 `id` 分配，并采用 `hashmap` 将 `id` 映射到文件描述符结构体指针，以提供接近于 $O(1)$ 的快速映射查询功能，同时，隶属于同一个文件描述符表的所有文件描述符都被串入文件描述符列表，以供文件描述符表销毁时可以将描述符所占内存全部正确回收。

5.2.7 `os_hashmap.c`

该文件实现了一个 `hashmap`，即通过哈希表加速键值对的访问，`hashmap` 结构体定义如下：

```
struct os_hashmap
{
    os_list_node_t *list; // 哈希节点列表集合
    os_size_t count; // 哈希节点列表数
    os_size_t count_bit; // 哈希节点列表数的2的对数
    os_hashmap_hash_function_t hash_function; // 哈希函数
};
```

为了加速哈希值的计算，避免取余计算，因此哈希表的大小固定为 2 的幂，同时，为了解决冲突问题，对于哈希表的每一个哈希值，都关联了一个列表，从而在哈希值冲突时进行键的匹配。

当哈希函数设计不好时，`hashmap` 的性能可能从 $O(1)$ 退化到 $O(n)$ ，因此，哈希函数的设计至关重要，未来也可以改为采用红黑树实现 `hashmap` 的每一个冲突列表，这样，最坏情况就降到了 $O(\log n)$ ，性能大大改善。

系统预先实现了一个默认的哈希函数，其实现就是简单的取余操作，具体实现如下：

```
static os_size_t os_hashmap_default_hash_function(os_hashmap_p hashmap, os_size_t key)
{
    return key & MASK(hashmap -> count_bit);
}
```

`hashmap` 的每一项称为一个哈希节点，其定义如下：

```
// 哈希节点
typedef struct os_hashmap_item
{
    os_size_t key; // 键
    void *value; // 值
    os_list_node_t node; // 列表节点
}os_hashmap_item_t,*os_hashmap_item_p;
```

可以注意到，该 `hashmap` 仅支持键为整数的情况，目前尚没有字符串或其它支持，之所以没有实现，是因为内核目前没有相关需求。

`hashmap` 实现了如下接口：

```

os_err_t os_hashmap_create(os_hashmap_p hashmap,os_size_t count,
    os_hashmap_hash_function_t hash_function);// 创建hashmap
void os_hashmap_remove(os_hashmap_p hashmap);// 销毁hashmap
os_err_t os_hashmap_set(os_hashmap_p hashmap,os_size_t key,void *value);// 设置某个键
    对应的值, 如果对应的键不存在, 将会自动创建
os_bool_t os_hashmap_get(os_hashmap_p hashmap,os_size_t key,void **value);// 获取某个
    键对应的值
void os_hashmap_remove_item(os_hashmap_p hashmap,os_size_t key);删除某个键

```

5.2.8 os_init.c

这里面实现了四个函数, 其中 `print_logo` 用于打印内核 Logo, `print_system_info` 用于调用 `print_logo` 打印内核 Logo, 同时打印内核基本信息, `os_build_check` 用于系统编译期检查, 而 `os_init` 则用于内核初始化, 这也是内核的入口。

`os_init` 函数执行步骤如下:

1. 执行系统编译器检查
2. 调用 `bsp_early_init` 完成最基础的 bsp 初始化工作
3. 调用 `print_system_info` 打印内核 Logo 和信息
4. 调用 `os_memory_init` 初始化内存子系统
5. 调用 `os_mmu_init` 初始化 MMU 子系统
6. 调用 `bsp_after_heap_init` 完成堆分配后的 bsp 初始化工作
7. 调用 `os_device_init` 初始化设备管理器
8. 调用 `os_vfs_init` 初始化 VFS
9. 调用 `os_task_scheduler_init` 完成任务调度器初始化
10. 调用 `bsp_after_task_scheduler_init` 完成任务调度器初始化后的 bsp 初始化工作
11. 调用 `os_task_scheduler_start` 启动调度器

一部分初始化工作由于依赖任务上下文, 因此无法在此完成初始化工作, 只能等待调度器启动加载 `idle` 任务, `idle` 任务加载 `main` 任务, 然后在 `main` 任务中完成剩余的初始化工作 (包括设备文件系统、只读文件系统、互斥锁等)。

5.2.9 os_interrupt.c

该文件实现了一个简单的中断框架支持, 具体实现了如下几个接口:

```

void os_enter_interrupt();// 进入中断时通知内核更新当前的中断层次
void os_leave_interrupt();// 离开中断时通知内核更新当前的中断层次
bool_t os_is_in_interrupt();// 判断当前程序是否运行在中断中
bool_t os_interrupt_disable();// 关中断, 并返回关中断前的中断状态
void os_interrupt_enable(bool_t enabled);// 开中断, 或者说是还原中断状态

```

5.2.10 os_io.c

该文件主要实现了 `os_printf` 和 `os_puts`，通过调用 `bsp_puts` 接口来实现，代码借用了 RT-Thread 的 `rt_kprintf` 函数实现。

接口如下：

```
void os_printf(const char *fmt,...);
void os_puts(const char *str);
```

5.2.11 os_memory.c

该文件实现了内存子系统，为内核提供了内存分配和释放功能，即动态内存管理功能，其包含了 Buddy System 支持及 Slub 支持，前者用于页面级别的内存分配，后者用于小于一个页面大小的内存块申请，具体采用哪个算法，则由内存子系统智能识别。内存子系统实现了如下接口：

```
void os_memory_init(); //内存子系统初始化函数
os_bool_t os_memory_is_initialized(); //指示内存子系统是否已初始化完成
void *os_memory_alloc(os_size_t size); //分配指定大小的内存
void os_memory_free(void *mem); //释放内存
os_size_t os_get_allocated_memory(); //获取已分配的内存大小，获取的是buddy system已经分配的页面数，不考虑slub中的缓存空闲页面
os_size_t os_get_total_memory(); //获取总内存大小
os_size_t os_get_free_memory(); //获取空闲内存大小
```

下面是内存子系统内的关联算法实现：

5.2.11.1 memory/os_memory_page.c

该文件实现了 Buddy System，这里首先说一下该实现所采用的内存划分方法：

空闲内存被分为两个部分，前半部分是 `metadata`，后半部分是数据页面，和 RT-Thread Smart 中实现不同的是，这里的页面划分初始化算法更简单：

这里首先定义几个变量：

- `meta_size` - `metadata` 结构体的大小（实际上是向最小 2 的幂上界进行了对齐）
- `page_size` - 页面大小
- `mem_size` - 可用内存大小
- `page_num` - 数据页面数

根据上面的定义，很容易得到如下的式子：

$$page_num = \lfloor \frac{mem_size}{meta_size + page_size} \rfloor$$

通过上面的式子，很容易就将内存划分成了两部分。

这里导出了如下函数：

```
void *os_memory_page_alloc(size_t size); // 页面分配（其大小为2的幂，且>=size）
void os_memory_page_free(void *addr); // 页面释放
size_t os_memory_page_get_allocated_page_count(); // 获取已分配的页面数量
size_t os_mmeory_page_get_total_page_count(); // 获取总页面数量
size_t os_memory_page_get_free_page_count(); // 获取空闲页面数量
void os_memory_page_init(); // buddy system 初始化
```

5.2.11.2 memory/os_memory_slub.c

该文件实现了 Slub 内存分配算法，相比于 Linux 而言，这里采用的算法实现要更为简单一些，为了初期实现降低复杂度，暂时不考虑 NUMA 系统和 Cache 性能优化问题。Slub 内存分配算法依赖 Buddy System 内存分配方法提供内存页，Slub 负责将内存页进行进一步细分，提供小于页大小的各类内存块的分配支持，其内存块的支持范围由 OS_MEMORY_SLUB_MIN_ORDER 与 OS_MEMORY_SLUB_MAX_ORDER 共同确定：OS_MEMORY_SLUB_MIN_ORDER 目前的设定值为 3，也就是表示最小的内存块大小为 8 字节，OS_MEMORY_SLUB_MAX_ORDER 目前的设定值为 11，也就是表示最大的内存块大小为 2K；在目前的规则中，最大的内存块大小必须等于页面大小的一半，否则会引起内存子系统的请求智能分发出错，导致分配出现不可预料的问题，同时，OS_MEMORY_SLUB_MIN_ORDER 也不可过小，否则会导致元信息过多，造成页面空间的巨大浪费。

这里导出了如下函数：

```
void os_memory_slub_init(); // Slub 初始化函数
void *os_memory_slub_alloc(os_size_t size); // 分配指定大小的 Slub
void os_memory_slub_free(void *object); // 释放一个内存块
```

5.2.12 os_mmu.c

该文件实现了 MMU 子系统，提供了 IO Mapping 及映射操作支持，目前至多支持四级页表，提供了如下操作接口：

```
void *os_mmu_create_io_mapping(os_mmu_vtable_p vtable, os_size_t pa, os_size_t size);
// 创建 IO Mapping
void os_mmu_remove_io_mapping(os_mmu_vtable_p vtable, void *va, os_size_t size); // 移除 IO Mapping
os_err_t os_mmu_create_mapping_auto(os_mmu_vtable_p vtable, os_size_t va, os_size_t size, os_mmu_pt_prot_t prot); // 在指定虚拟地址处映射并自动分配物理内存空间
os_bool_t os_mmu_is_initialized(); // 指示 MMU 子系统是否已初始化完成
os_err_t os_mmu_vtable_create(os_mmu_vtable_p vtable, os_mmu_pt_l1_p l1_vtable, os_size_t va_start, os_size_t va_size); // 创建页表
```

```

void os_mmu_vtable_remove(os_mmu_vtable_p vtable,os_bool_t remove_mapping);//销毁页
    表
void os_mmu_switch(os_mmu_vtable_p vtable);//切换页表
os_mmu_vtable_p os_mmu_get_current_vtable();//获取当前的页表结构体指针
os_bool_t os_mmu_page_fault_handler(os_size_t addr,os_bool_t write);//页面fault处理
    程序
void os_mmu_init();//MMU子系统初始化

```

目前内核仅支持线性映射，但是同时也支持 IO Mapping。为了保证各个任务页表的 IO Mapping 一致性，这里需要借助 Page Fault，当内核访问试图访问一个无效地址时，若检测到属于内核页表的 IO Mapping 区域，则将 IO Mapping 从内核页表中复制一份到当前的任务页表中。（目前暂未完全实现）

5.2.13 os_mutex.c

该文件实现了互斥锁的支持，同时支持递归调用，该锁的实现依赖等待队列，目前实现了如下接口：

```

void os_mutex_init(os_mutex_p mutex);//互斥锁初始化
void os_mutex_lock(os_mutex_p mutex);//互斥锁锁定，支持递归调用
void os_mutex_unlock(os_mutex_p mutex);//互斥锁解锁，支持递归调用

```

5.2.14 os_string.c

由于目前内核不依赖于任何 libc 库，因此需要实现一些库函数，如下所示（因功能和对应类似名字的 C 函数一致，这里不再对功能进行赘述）：

```

size_t os_strlen(const char *str);
void os_memset(uint8_t *ptr,uint8_t value,size_t size);
void os_memcpy(uint8_t *dst,uint8_t *src,size_t size);
void os_strcpy(char *dststr,const char *srcstr);

```

5.2.15 os_syscall.c

该文件负责实现 syscall 相关的函数，并进行具体的 syscall 实现，目前实现了如下 syscall 相关的接口：

```

static os_err_t os_copy_from_user(void *kernel_mem,os_size_t user_mem,os_size_t
    mem_size);//用于实现将内存从用户态拷贝到内核态，通常用于系统调用入口从用户内存读
    取数据时
static os_err_t os_copy_to_user(os_size_t user_mem,void *kernel_mem,os_size_t
    mem_size);//用于实现将内存从内核态拷贝到用户态，通常用于系统调用出口将数据写回用
    户内存时

```

```
static os_err_t os_clone_copy_memory(os_task_p task); //用于Clone时拷贝页面映射并拷贝所有的页面数据，暂时不支持写时复制机制
```

同时，这里实现了 `openat`、`close`、`getdents64`、`read`、`write`、`fstat`、`clone`、`execve`、`exit`、`getpid`、`getppid`、`yield`，其中 `execve` 与 `write` 系统调用已可以正常使用，其它系统调用暂未测试。

5.2.16 os_task.c

这是一个优先级时间片轮转算法调度器的实现，`task` 结构体的定义如下：

```
//任务入口函数类型
typedef os_ssize_t (*task_func_t)(os_size_t arg);

//任务退出函数类型
typedef void (*task_exit_func_t)(os_ssize_t exit_code);

//任务状态枚举
typedef enum os_task_state
{
    OS_TASK_STATE_RUNNING, //运行态
    OS_TASK_STATE_READY,  //就绪态
    OS_TASK_STATE_BLOCKING, //阻塞态
    OS_TASK_STATE_SLEEPING, //睡眠态
    OS_TASK_STATE_STOPPED, //终止态（任务已经彻底消亡，等待环境清理）
}os_task_state_t;

//文件描述符表的前置类型声明
typedef struct os_file_fd_table os_file_fd_table_t,*os_file_fd_table_p;

//任务结构体
typedef struct os_task
{
    os_size_t sp; //栈顶指针，这个必须在结构体的第一项，以方便上下文切换汇编程序访问
    os_size_t stack_addr; //栈起始地址
    os_size_t stack_size; //栈大小
    struct os_task *parent; //父任务
    char name[OS_VFS_PATH_MAX + 1]; //任务名
    char path[OS_VFS_PATH_MAX + 1]; //任务路径
    os_size_t pid; //Process ID
    os_size_t tid; //Thread ID
    os_size_t sid; //Session ID
    os_size_t priority; //任务优先级
    os_size_t tick_init; //拥有的时间片
    os_size_t tick_remaining; //剩余时间片
    os_task_state_t task_state; //任务状态
    task_func_t entry; //任务入口
    os_size_t arg; //任务入口参数
```

```

    task_exit_func_t exit_func; //任务退出函数（用于任务结束后的环境清理）
    os_ssize_t exit_code; //任务退出码
    os_size_t brk; //堆上界
    os_size_t init_brk; //堆下界
    os_file_fd_table_p fd_table; //文件描述符表
    os_mmu_vtable_p vtable; //页表
    os_list_node_t task_node; //任务列表中的节点
    os_list_node_t schedule_node; //调度列表中的节点
    os_list_node_t child_list; //子任务列表
    os_list_node_t child_node; //子任务列表中的节点
}os_task_t,*os_task_p;

```

并导出了如下接口：

```

os_task_t *os_task_get_current_task(); //获取当前的任务
os_err_t os_task_init(os_task_p task,os_size_t stack_size,os_size_t priority,
    os_size_t tick_init,task_func_t entry,os_size_t arg,const char *name); //初始化一个任务结构体
void os_task_remove(os_task_p task); //销毁任务，该函数用于将任务结构体从内核中剥离，一般仅用于任务自身的环境清理，请不要在任务运行时调用该函数，否则可能导致系统崩溃
void os_task_startup(os_task_p task); //启动任务，该任务必须首先被初始化，否则行为不可预测
void os_task_yield(); //立即放弃所有的剩余时间片
void os_task_sleep(); //让当前任务进入睡眠态
void os_task_wakeup(os_task_t *task); //唤醒任务
os_task_p os_task_get_task_by_pid(os_size_t pid); //通过pid获取任务结构体指针
void os_task_schedule(); //执行任务调度，该函数将会选出下一个应当执行的任务，若下一任务与当前任务不同，将会触发任务切换行为
os_bool_t os_task_scheduler_is_initialized(); //判断调度器是否已经完成初始化
void os_task_scheduler_init(); //任务调度器初始化
void os_task_scheduler_start(); //任务调度器启动，从这时候开始，就进入了第一个任务上下文，该函数不会返回
void os_task_switch_vtable(os_task_p task); //切换到指定任务的页表
void os_task_print_tree(os_task_p task); //打印任务树
os_task_p os_task_get_root_task(); //获取根任务
os_task_p os_task_get_main_task(); //获取main任务

```

调度器在进行任务切换时，如果检测到任务处于中断内，则设置如下几个标志推迟到中断处理结束时由 arch 负责任务切换，否则立即调用 arch 的接口进行任务切换：

```

if(os_is_in_interrupt())
{
    need_lazy_task_switch = OS_TRUE;
    lazy_old_task = old_task;
    lazy_next_task = next_task;
}
else

```

```
{  
    arch_task_switch(old_task,next_task);  
}
```

5.2.17 os_terminal_color.c

该文件实现了终端前景色和背景色的控制，支持如下若干种颜色：

```
//终端颜色枚举  
typedef enum  
{  
    TERMINAL_COLOR_BLACK = 30,  
    TERMINAL_COLOR_RED,  
    TERMINAL_COLOR_GREEN,  
    TERMINAL_COLOR_YELLOW,  
    TERMINAL_COLOR_CYAN,  
    TERMINAL_COLOR_PURPLE,  
    TERMINAL_COLOR_BLUE,  
    TERMINAL_COLOR_WHITE  
}terminal_color_t;
```

并实现了如下接口：

```
void terminal_color_set_fg(terminal_color_t color);//设置终端前景色  
void terminal_color_set_bg(terminal_color_t color);//设置终端背景色  
void terminal_color_set(terminal_color_t fg,terminal_color_t bg);//设置终端背景色和  
    前景色
```

5.2.18 os_tick.c

该文件实现了内核的调度定时器中断处理程序 `os_tick_handler`，该程序首先获取当前的任务，然后检查当前任务的剩余时间片，假设未用完，则减去一个时间片，否则初始化时间片，并且执行 `task_schedule` 进入任务调度。

同时该文件也实现了内核 Tick 管理，为内核提供时间支持，包括延时等相关功能支持，具体接口实现如下：

```
void os_tick_handler();//内核调度定时器中断处理程序，应在定时器中断中被调用  
os_size_t os_tick_get();//获取当前的系统全局Tick值
```


5.2.19 os_vfs.c

该文件实现了 VFS，其中 VFS 主要包含两个对象，即文件系统与挂载点，同时还包含上面所提到的文件管理相关的对象。

其中文件系统信息结构体和挂载点信息结构体的实现如下（同时还有一个描述文件系统状态的结构体）：

```
//VFS文件系统状态结构体
typedef struct os_vfs_state
{
    os_size_t block_size;//块大小
    os_size_t block_count;//总块数量
    os_size_t block_free;//剩余块数量
}os_vfs_state_t,*os_vfs_state_p;

//VFS文件系统操作函数集结构体
typedef struct os_vfs_ops
{
    os_err_t (*mount)(os_vfs_mp_p mp,os_size_t mount_flag,void *priv_data);//文件系
        统挂载
    os_err_t (*unmount)(os_vfs_mp_p mp);//文件系统卸载
    os_err_t (*mkfs)(os_vfs_p fs,os_device_p dev);//文件系统格式化
    os_err_t (*statfs)(os_vfs_mp_p mp,os_vfs_state_p state);//获取文件系统状态
    os_err_t (*unlink)(os_vfs_mp_p mp,const char *path);//删除文件
    os_err_t (*stat)(os_vfs_mp_p mp,const char *path,os_file_state_p state);//获取文
        件状态
    os_err_t (*rename)(os_vfs_mp_p mp,const char *old_path,const char *new_path);//
        重命名文件
    os_file_ops_p file_ops;
}os_vfs_ops_t,*os_vfs_ops_p;

//VFS文件系统信息结构体
struct os_vfs
{
    char *name;//文件系统名称
    os_size_t flag;//文件系统标志
    os_vfs_ops_p ops;//操作函数集
    os_list_node_t node;//链表节点
    os_mutex_t lock;//锁
    os_size_t mount_refcnt;//挂载引用数，指示有多少挂载点依赖该结构体
};

//VFS挂载点结构体
struct os_vfs_mp
{
    os_vfs_p fs;//关联的文件系统结构体指针
    char path[OS_VFS_PATH_MAX + 1];//挂载点路径（已经过正规化）
```

```

os_size_t subpath_offset;//子路径偏移（即完整路径从该偏移开始属于该挂载点内的路径）
os_device_p dev;//关联设备结构体指针
const char *dev_path;//设备路径
os_size_t mount_flag;//挂载标志，参考OS_FILE_FLAG开头的标志
os_list_node_t node;//链表节点
os_mutex_t lock;//锁
os_size_t mp_cnt;//指示有多少子挂载点
os_size_t open_file_cnt;//指示该挂载点有多少打开的文件
void *priv_data;//私有数据
};

```

可以看到，上述两个结构体中，都含有独立的锁和引用计数，其中细粒度锁是为了效率优化，提高并行度，引用数支持是防止依赖关系未解除时依赖项被删除或非法修改（如文件系统对应的挂载点还挂在系统中时，移除对应的文件系统，或者某个挂载点上有打开的文件时，直接移除挂载点，这都会造成严重问题）。

为了防止死锁，这里规定细粒度的加锁顺序：文件系统-> 挂载点-> 文件节点-> 文件描述符。

VFS 实现了如下接口：

```

void vfs_lock();//VFS加锁
void vfs_unlock();//VFS解锁
os_err_t os_vfs_normalize_path(const char *path,char *buf);//正规化路径，该过程会去除路径中所有的.和..，并保证不出现连续的两个/，同时保证结尾不为/除非为根目录
os_vfs_mp_p os_vfs_find_mp_by_path(const char *path);//根据路径查找挂载点，并返回挂载点结构体指针，原理是进行前缀路径匹配，最长的前缀匹配的挂载点就是目标挂载点
os_err_t os_vfs_register(const os_vfs_p fs);//注册新的文件系统
os_err_t os_vfs_mount(const char *mount_path,const char *fs_name,const char *dev,os_size_t mount_flag,void *priv_data);//挂载文件系统
os_err_t os_vfs_unmount(const char *mount_path);//卸载文件系统
os_err_t os_vfs_mkfs(const char *fs_name,const char *dev);//格式化文件系统
os_err_t os_vfs_statfs(const char *mount_path,os_vfs_state_p state);//获取文件系统状态信息
os_err_t os_vfs_unlink(const char *path);//删除文件
os_err_t os_vfs_stat(const char *path,os_file_state_p state);//获取文件状态信息
os_err_t os_vfs_rename(const char *old_path,const char *new_path);//重命名文件，不支持跨文件系统
os_bool_t os_vfs_is_initialized();//指示VFS是否完成初始化
void os_vfs_init();//VFS初始化函数

```

5.2.20 os_waitqueue.c

该文件提供了等待队列的实现，目前等待队列被应用于互斥锁的实现中，目前实现了如下几个接口：

```

void os_waitqueue_add(os_waitqueue_p waitqueue,os_waitqueue_node_p node);//向等待队列中加入节点，通常由os_waitqueue_wait调用，而不是由外部直接调用

```

```
void os_waitqueue_remove(os_waitqueue_node_p node); //从等待队列中移除节点，通常由
os_waitqueue_wait调用，而不是由外部直接调用
void os_waitqueue_wait(os_waitqueue_p waitqueue); //让当前任务在指定的等待队列中等
待，执行该函数后，当前任务会进入睡眠态
void os_waitqueue_wakeup(os_waitqueue_p waitqueue); //唤醒等待队列中的一个任务，若队
列为空，则不动作
os_bool_t os_waitqueue_empty(os_waitqueue_p waitqueue); //检测等待队列是否为空
os_task_p os_waitqueue_get_head(os_waitqueue_p waitqueue); //获取等待队列的头部任务
void os_waitqueue_init(os_waitqueue_p waitqueue); //初始化等待队列
```

5.3 qemu-rv64 平台的移植

5.4 编译运行方式

首先在 bsp 目录下的 osconfig.py 中配置正确的工具链路径，然后在 bsp 目录中执行 “scons -jN”（N 表示并行数），即可得到 dreamos.bin，运行 make run 即可启动。

5.4.1 移植综述

为了完成该平台的移植，需要分别实现 firmware、arch 以及 bsp 三部分。

5.4.2 firmware (firmware/sbi)

这里直接采用和 RT-Thread Smart 中同样的 sbi 头文件来实现 sbi 接口支持。

5.4.3 arch (arch/riscv64)

包含如下几个文件，用途分别如下：

表 5.5: arch 目录文件用途

文件名	用途
arch.c	实现了任务栈初始化接口
arch.h	导出了几个架构相关的同步宏
arch_asm.S	实现了任务切换接口
arch_err.h	定义了架构相关的错误码
arch_mmu.c	实现了 MMU 接口
arch_mmu.h	完成了架构 MMU 信息的定义和各级页表及页表属性的定义，同时也对接了一些 MMU 接口
arch_syscall.c	实现了 syscall 接口
arch_syscall.h	定义了 syscall 常量

文件名	用途
arch_trap.c	实现了 Trap 接口
arch_trap.h	定义了 Trap 的上下文栈结构，该结构同时作为任务上下文栈结构
encoding.h	riscv 架构的相关头文件
entry_gcc.S	启动代码
function.S	包含了一些同步功能实现
interrupt_gcc.S	包含 trap 入口
stackframe.h	包含一些上下文操作相关的汇编宏
syscall_gcc.S	包含系统调用接口

下面逐文件介绍。

5.4.3.1 arch.c

这里定义了 `arch_task_stack_framt_init` 接口的实现，该实现首先将上下文栈每一项初始化为 `0xdeadbeef`，然后将 `ra` 设置为任务的退出函数，将 `gp` 设置为内核的全局指针 `$_global_pointer$`，将 `a0` 设置为任务入口参数，将 `sepc` 设置为任务入口，将 `user_sp` 设置为任务栈顶，将 `sp` 指向任务栈中的上下文栈结构起始位置，将 `sstatus` 初始化为 `0x000401120`，以便保证任务初始运行在 `S` 态并开启中断。

5.4.3.2 arch.h

这里定义了几个关键的同步宏：

```
#define SYNC() do{asm volatile("fence;fence.i");}while(0)
#define SYNC_DATA() do{asm volatile("fence");}while(0)
#define SYNC_INSTRUCTION() do{asm volatile("fence.i");}while(0)
```

5.4.3.3 arch_asm.S

这个文件实现了 `arch_task_switch` 接口，如果该接口的第一个参数为 `NULL`，则会直接切入新任务而丢弃旧任务的上下文，这个设计是为调度器启动第一个任务时使用。

5.4.3.4 arch_err.h

这个文件定义了架构相关的错误码。

5.4.3.5 arch_mmu.c

这个文件对接了如下几个内核接口：

```
os_err_t os_mmu_create_mapping(os_mmu_vtable_p vtable,os_size_t va,os_size_t pa,
    os_size_t size,os_mmu_pt_prot_t prot);//创建映射
os_err_t os_mmu_remove_mapping(os_mmu_vtable_p vtable,os_size_t va,os_size_t size);
    //移除映射
os_size_t os_mmu_find_vaddr(os_mmu_vtable_p vtable,os_size_t va_start,os_size_t size
    );//寻找空闲虚拟地址
void arch_mmu_switch(os_mmu_vtable_p vtable);//切换页表
os_bool_t os_mmu_io_mapping_copy(os_mmu_vtable_p vtable);//从内核页表拷贝IO Mapping
    到指定页表
os_err_t os_mmu_kernel_mapping_copy(os_mmu_vtable_p vtable);//从内核页表拷贝Kernel
    Mapping到指定页表
os_err_t os_mmu_user_mapping_copy(os_mmu_vtable_p dst_vtable,os_mmu_vtable_p
    src_vtable);//将一个页表中的用户映射拷贝到另一个页表中
void *os_mmu_user_va_to_kernel_va(os_mmu_vtable_p vtable,os_size_t user_va);//利用指
    定页表将虚拟地址转换为物理地址
void os_mmu_preinit();//MMU预初始化
void os_mmu_preinit_secondary();//MMU二级预初始化
os_bool_t os_mmu_is_preinitialized();//指示MMU初始化是否已完成
os_mmu_vtable_p os_mmu_get_kernel_pagetable();//获取当前的内核页表
```

其中 MMU 预初始化实现了内核页表和内核跳转表的初始化，而二级预初始化则负责切换到内核页表，为内核页表一级页表完成物理地址到虚拟地址的变换，并销毁内核跳转表。

内核启动时还未开启 MMU，因此采用的是物理地址，这里构造了一个内核跳转表，使得内核的物理地址在跳转表中有一一对应的相等的虚拟地址，同时还有内核正式运行时的虚拟地址映射。内核需要借助跳转表跳到正式的虚拟地址处运行，在完成跳转后，切换到正式的内核页表并销毁跳转表。

5.4.3.6 arch_mmu.h

这个文件中按照 SV39 标准定义了三级页表的结构，同时按照如下结构分配虚拟地址空间：IO Mapping 0xFFFF FFC0 0000 0000 ~ 0xFFFF FFFF FFFF FFFF 256GB Kernel Mapping 0x0000 0020 0000 0000 ~ 0x0000 003F FFFF FFFF 128GB User Mapping 0x0000 0000 0000 0000 ~ 0x0000 001F FFFF FFFF 128GB

其中用户映射地址虽然从 0x0000 0000 0000 0000 开始，但由于 0 地址通常表示 OS_NULL，为了避免歧义，实际用户映射地址是从 1 号页面开始分配（从 0 开始计数）。

这里同时也对接了 tlb 冲刷接口，考虑到 RV 架构的虚拟地址存在不连续性（实际上连续，只是由于虚拟地址中未被使用到的一部分位要和最高有效位保持一致），因此专门提供了 OS_MMU_GET_EFFECTIVE_VA 与 OS_MMU_GET_REAL_VA 用于在实际的虚拟地址和去除未被使用位的虚拟地址（这样可以保证虚拟空间的连续性）之间转换。

5.4.3.7 arch_trap.c

这个文件实现了重要的 `trap_handler` 接口，这也是 C 的 Trap 处理入口，进入该函数后，首先会检测是否为中断，如果是则交给 `bsp_interrupt_handler` 处理，如果处理成功直接返回，否则进行报错，报错会显示各种通用寄存器（包括符号）以及 CSR 和 MMU 的状态显示，并会打印出 `Stacktrace`。

5.4.3.8 arch_trap.h

这里定义了任务上下文栈结构：

表 5.6: 任务上下文栈结构

偏移地址	字段
0x00	sepc
0x08	ra
0x10	sstatus
0x18	gp
0x20	tp
0x28	t0
0x30	t1
0x38	t2
0x40	s0_fp
0x48	s1
0x50	a0
0x58	a1
0x60	a2
0x68	a3
0x70	a4
0x78	a5
0x80	a6
0x88	a7
0x90	s2
0x98	s3
0xA0	s4
0xA8	s5
0xB0	s6
0xB8	s7
0xC0	s8

偏移地址	字段
0xC8	s9
0xD0	s10
0xD8	s11
0xE0	t3
0xE8	t4
0xF0	t5
0xF8	t6
0x100	user_sp

5.4.3.9 entry_gcc.S

这里实现了程序入口，首先清零 `sie` 和 `sip` 寄存器，然后配置 `stvec` 指向 `Trap` 入口 `trap_entry`，然后关闭 `FPU` 并将内核默认栈地址写入 `sscratch` 寄存器，紧接着清零其它的通用寄存器，并清零 `bss`，紧接着完成 `MMU` 的预初始化，启用 `mmu` 跳入内核虚拟地址空间，完成 `MMU` 二级预初始化，并最后跳入 `main` 函数开始执行。

5.4.3.10 function.S

这个文件中实现了几个同步接口，目前处于废弃状态。

5.4.3.11 interrupt_gcc.S

这里实现了 `trap_entry` 入口，该函数首先切换到 `Trap` 栈，然后保存上下文后切入 `trap_handler` 处理，在处理完后根据 `need_lazy_task_switch` 标志的状态进行延迟的任务切换处理，最后返回正常程序流执行。

5.4.3.12 syscall_gcc.S

这里实现了 `syscall_entry`，是 `syscall` 处理的入口。该实现首先加载内核栈，然后从中断栈复制上下文，并调用 `arch_syscall_handler` 进行实际的 `syscall` 处理，在 `syscall` 处理期间，中断开关处于打开状态，以允许 `syscall` 被抢占。

5.4.4 bsp (bsp/qemu-virt-rv64)

包含如下几个文件，用途分别如下：

表 5.7: bsp 目录文件用途

文件名	用途
bsp_init.c	BSP 初始化函数实现

文件名	用途
bsp_interface_asm.S	实现了 BSP 的中断开关函数
bsp_interface.c	实现了 BSP 的其他接口
bsp.h	bsp 公共头文件，用于引入 bsp 的其它头文件到内核
entry_main.c	包含 main 函数
task_main_entry.c	包含 main 任务的入口
tick.c/h	Tick 驱动的实现
trap_handler	bsp_interrupt_handler 中断处理接口的实现
osconfig.h	包含内核配置参数
osconfig.py	包含内核编译参数

下面逐文件介绍。

5.4.4.1 bsp_init.c

其中仅实现了函数 `bsp_init`，目前实现为空。

5.4.4.2 bsp_interface_asm.S

这里实现了内核导出的 `bsp_interrupt_disable` 与 `bsp_interrupt_enable` 接口。

5.4.4.3 bsp_interface.c

这里实现了内核导出的 `bsp_early_init`、`bsp_after_heap_init`、`bsp_after_task_scheduler_init` 以及 `bsp_puts`，其中 `bsp_early_init` 中调用 `tick_init` 初始化了 Tick，`bsp_puts` 直接利用 `firmware` 的 `sbi_console_putchar` 来实现。

5.4.4.4 entry_main.c

这里实现了 `main` 函数，其中首先调用 `bsp_init` 完成 bsp 最早的初始化，然后调用 `kernel_init` 启动内核。

5.4.4.5 task_main_entry.c

这里实现了 `os_task_main_entry` 这个 `main` 任务入口，这也是内核启动时运行的第二个任务，负责完成剩余的依赖任务上下文的初始化工作；同时也实现了 `os_task_user_entry` 这个 `user` 任务入口，这也是内核启动时运行的第三个任务，负责启动第一个用户程序。

`main` 任务按照以下步骤完成剩余的内核初始化工作：

1. 初始化只读文件系统。

2. 初始化设备文件系统。
3. 初始化控制台设备。
4. 初始化互斥锁。
5. 初始化并启动用户任务（该任务负责）

而 `user` 任务则将自身转入用户态并通过调用 `execve` 系统调用从文件系统加载第一个 `ELF` 文件。

5.4.4.6 tick.c/h

这个 `Tick` 驱动的实现和上述 `RT-Thread Smart` 驱动移植到 `qemu-rv64` 平台时的实现类似。

5.4.4.7 trap_handler.c

这里实现了 `bsp_interrupt_handler` 用于处理 `bsp` 中断，这里处理了 `S` 态的 `Timer` 中断。

5.4.4.8 osconfig.h

这里定义了几个内核配置参数，分别如下所示：

```
#define OS_PRINTF_BUFFER_SIZE (256)
#define PAGE_BITS (12)
#define MEMORY_BASE (0x80000000UL)
#define MEMORY_SIZE (128 * 0x100000)
#define TASK_PRIORITY_MAX (31)
#define TICK_PER_SECOND (100)
#define IDLE_TASK_STACK_SIZE (8192)
#define IDLE_TASK_TICK_INIT (10)
#define MAIN_TASK_STACK_SIZE (16384)
#define MAIN_TASK_PRIORITY (10)
#define MAIN_TASK_TICK_INIT (1)
#define OS_VFS_PATH_MAX (255)
#define OS_TASK_MAX_NUM (65536)

#define SLUB_MIN_PARTIAL (2)

#define OS_MAX_OPEN_FILES (128)

#define OS_ARCH64

#define OS_CONSOLE_DEVICE "/dev/console"
```

5.4.4.9 osconfig.py

这里类似于 `RT-Thread Smart` 中的 `rtconfig.py`，唯一不同的地方在于，这里多了一个 `FIRMWARE` 参数用于指定 `firmware` 类型。

第 6 章

未来的工作展望

内核方面，我会继续实现 **DreamOS**，并以 **task** 的形式完成 **Process**、**Thread** 的概念构建，同时加入 **Session** 的支持以及权限管理系统等，同时要加入多终端支持，并进一步调整代码结构，优化性能等。