DragonOS

发行版本 *dev*

fslongjin

2022 年 11 月 06 日

入门

CHAPTER 1

DragonOS 简介

DragonOS 龙操作系统（以下简称“DragonOS”）是一个基于 x86-64 体系结构开发的，基于 GPLv2 协议开放源代码的 64 位操作系统

你可能对 DragonOS 中已经实现了哪些功能感兴趣，您可以转到这里：功能特性

1.1 DragonOS 的功能

1.1.1 规范

- 启动引导：Multiboot2
- 接口：posix 2008

1.1.2 内核层

内存管理

- 页分配器
- slab 分配器
- VMA
- MMIO 地址空间自动分配

多核

- 多核引导
- ipi 框架

进程管理

- 进程创建
- 进程回收
- 内核线程
- fork
- exec
- 进程睡眠（支持高精度睡眠）
- kthread 机制

同步原语

- mutex 互斥量
- semaphore 信号量
- atomic 原子变量
- spinlock 自旋锁
- wait_queue 等待队列

调度

- CFS 调度器
- 单核调度

IPC

- 匿名 pipe 管道

文件系统

- VFS
- fat32
- devfs
- rootfs

异常及中断处理

- APIC
- softirq 软中断
- 内核栈 traceback

内核数据结构

- 普通二叉树
- kfifo 缓冲区
- 循环链表
- IDR

内核实用库

- LZ4 压缩库（1.9.3）
- 字符串操作库
- ELF 可执行文件支持
- printk
- 基础数学库
- 屏幕管理器
- textui 框架
- CRC 函数库

系统调用

请见系统调用文档

测试框架

- ktest

驱动程序

- ACPI 高级电源配置模块
- IDE 硬盘
- AHCI 硬盘
- PCI
- XHCI（usb3.0）
- ps/2 键盘
- ps/2 鼠标
- HPET 高精度定时器
- RTC 时钟
- local apic 定时器
- UART 串口
- VBE 显示

1.1.3 用户层

LibC

- 基础系统调用
- 基础标准库函数
- 部分数学函数

shell 命令行程序

- 基于字符串匹配的解析
- 基本的几个命令

驱动程序

- ps/2 键盘用户态驱动

CHAPTER 2

构建 DragonOS

2.1 从 Docker 构建（推荐）

为减轻配置环境的负担，DragonOS 发布了一个 Docker 编译环境，便于开发者运行 DragonOS。我们强烈建议您采用这种方式来运行 DragonOS。

本节假设以下操作均在 Linux 下进行。

2.1.1 安装 Docker

您可以在 docker 官网下载安装 docker-ce.

详细信息请转到：https://docs.docker.com/engine/install/

2.1.2 获取 DragonOS 编译镜像

当您成功安装了 docker 之后，您可以通过以下命令，下载 DragonOS 的编译镜像：

```
docker pull dragonos/dragonos-dev:v1.0
```

2.1.3 安装 qemu 虚拟机

在本节中，我们建议您采用命令行安装 qemu：

```
sudo apt install -y qemu qemu-system qemu-kvm
```

请留意，若您的 Linux 系统是在虚拟机中运行的，还请您在您的 VMware/Virtual Box 虚拟机的处理器设置选项卡中，开启 Intel VT-x 或 AMD-V 选项，否则，DragonOS 将无法运行。

在某些 Linux 发行版的软件仓库中构建的 Qemu 可能存在不识别命令参数的问题，如果遇到这种问题，请卸载 Qemu，并采用编译安装的方式重新安装 Qemu

在该地址下载 Qemu 源代码：https://download.qemu.org/

解压后进入源代码目录，然后执行下列命令：

```
./configure --enable-kvm
make -j 8
sudo make install
```

2.1.4 下载 DragonOS 的源代码

假设您的计算机上已经安装了 git，您可以通过以下命令，获得 DragonOS 的最新的源代码：

```
git clone https://github.com/fslongjin/DragonOS
cd DragonOS
```

2.1.5 创建磁盘镜像

首先，您需要使用 tools 文件夹下的 create_hdd_image.sh，创建一块虚拟磁盘镜像。您需要在 tools 文件夹下运行此命令。

```
bash create_hdd_image.sh
```

2.1.6 运行 DragonOS

如果不出意外的话，这将是运行 DragonOS 的最后一步。您只需要在 DragonOS 的根目录下方，执行以下命令，即可运行 DragonOS。

```
bash run.sh --docker
```

若输入密码后仍提示权限不足，您可以使用以下命令运行：

```
sudo bash run.sh --docker
```

稍等片刻，DragonOS 将会被运行。

在 qemu 虚拟机被启动后，我们需要在控制台输入字母 `c`，然后回车。这样，虚拟机就会开始执行。

2.2 手动搭建开发环境

若您追求快速的编译速度，以及完整的开发调试支持，且愿意花费半个小时到两个小时的时间来配置开发环境的话，该小节的内容能帮助到您。

2.2.1 软件依赖

您需要编译安装以下软件依赖。他们的源代码可以在对应项目的官方网站上获得。

- grub 2.06 (不必使用 sudo 权限进行 install)
- qemu 6.2.0 (启用所有选项)

  需要注意的是，编译安装 qemu 将会是一件费时费力的工作，它可能需要花费你 40 分钟以上的时间。

  对于以下软件依赖，建议您使用系统自带的包管理器进行安装。

- gcc >= 8.3.0
- xorriso
- fdisk
- make
- VNC Viewer
- gdb

2.2.2 编译 DragonOS

1. 安装编译及运行环境
2. 进入 DragonOS 文件夹
3. 输入命令：`make -j 16` 即可编译

2.2.3 创建磁盘镜像

首先，您需要使用 `sudo` 权限运行 `tools/create_hdd_image.sh`，为 DragonOS 创建一块磁盘镜像文件。该脚本会自动完成创建磁盘镜像的工作，并将其移动到 `bin/`目录下。

2.2.4 运行 DragonOS

至此，准备工作已经完成，您可以在 DragonOS 项目的根目录下，输入

```
bash run.sh
```

然后，DragonOS 将会被启动，您可以通过 VNC Viewer 连接至虚拟机。在 qemu 虚拟机被启动后，我们需要在控制台输入字母 `c`，然后回车。这样，虚拟机就会开始执行。

CHAPTER 3

引导加载

DragonOS 采用 GRUB2 作为其引导加载程序，支持 Multiboot2 协议引导。目前仅支持 GRUB2.06 版本。

3.1 引导加载程序

3.1.1 原理

目前，DragonOS 仅支持 Legacy BIOS 进行启动引导。

在 `head.S` 的头部包含了 Multiboot2 引导头，里面标志了一些 Multiboot2 相关的特定信息，以及一些配置命令。

在 DragonOS 的启动初期，会存储由 GRUB2 传来的 magic number 以及 multiboot2_boot_info_addr。当系统进入 `Start_Kernel` 函数之后，将会把这两个信息保存到 multiboot2 驱动程序之中。信息的具体含义请参照 Multiboot2 Specification 进行理解，该部分难度不大，相信读者经过思考能理解其中的原理。

3.1.2 未来发展方向

- 增加对 UEFI 启动的支持

3.1.3 参考资料

- Multiboot2 Specification
- GNU GRUB Manual 2.06

3.2 Multiboot2 支持模块

Multiboot2 支持模块提供对 Multiboot2 协议的支持。位于 `kernel/driver/multiboot2` 文件夹中。

根据 Multiboot2 协议，操作系统能够从 BootLoader 处获得一些信息，比如基本的硬件信息以及 ACPI 表的起始地址等。

3.2.1 数据结构

`kernel/driver/multiboot2/multiboot2.h` 中按照 Multiboot2 协议的规定，定义了大部分的数据结构，具体细节可查看该文件: DragonOS/multiboot2.h at master · fslongjin/DragonOS · GitHub

---

3.2.2 迭代器

由于 Multiboot2 的信息存储在自 `multiboot2_boot_info_addr` 开始的一段连续的内存空间之中，且不同类型的 header 的长度不同，因此设计了一迭代器 `multiboot2_iter`。

函数原型

```
void multiboot2_iter(bool (*_fun)(const struct iter_data_t *, void *, unsigned int *),
                     void *data, unsigned int *count)
```

_fun

指定的 handler。当某个 header 的 tag 与该 handler 所处理的 tag 相同时，handler 将处理该 header，并返回 true。

其第一个参数为 tag 类型，第二个参数为返回的数据的指针，第三个值为计数（某些没有用到该值的地方，该值可以为空）

data

传递给 `_fun` 的第二个参数，`_fun` 函数将填充该指针所指向的内存区域，从而返回数据。

count

当返回的 **data** 为一个列表时，通过该值来指示列表中有多少项。

3.2.3 迭代工作函数

在模块中，按照我们需要获取不同类型的 tag 的需要，定义了一些迭代器工作函数。

CHAPTER 4

核心 API 文档

这里是 DragonOS 的核心 api 文档。

4.1 DragonOS 内核核心 API

4.1.1 循环链表管理函数

循环链表是内核的重要的数据结构之一。包含在 `kernel/common/list.h` 中。

`void list_init(struct List *list)`

描述

初始化一个 List 结构体，使其 prev 和 next 指针指向自身

参数

list

要被初始化的 List 结构体

```
void list_add(struct List *entry, struct List *node)
```

描述

将 node 插入到 entry 的后方

参数

entry

已存在于循环链表中的一个结点

node

待插入的结点

```
void list_append(struct List *entry, struct List *node)
```

描述

将 node 插入到 entry 的前方

参数

entry

已存在于循环链表中的一个结点

node

待插入的结点

```
void list_del(struct List *entry)
```

描述

从链表中删除结点 entry

参数

entry

待删除的结点

```
list_del_init(struct List *entry)
```

描述

从链表中删除结点 entry，并将这个 entry 使用 list_init() 进行重新初始化。

参数

entry

待删除的结点

```
bool list_empty(struct List *entry)
```

描述

判断链表是否为空

参数

entry

链表中的一个结点

```
struct List *list_prev(struct List *entry)
```

描述

获取 entry 的前一个结点

参数

entry

链表中的一个结点

```
struct List *list_next(struct List *entry)
```

描述

获取 entry 的后一个结点

参数

entry

链表中的一个结点

```
void list_replace(struct List *old, struct List *new)
```

描述

将链表中的 old 结点替换成 new 结点

参数

old

要被换下来的结点

new

要被换入链表的新的结点

`list_entry(ptr, type, member)`

描述

该宏能通过 ptr 指向的 List 获取到 List 所处的结构体的地址

参数

ptr

指向 List 结构体的指针

type

要被换入链表的新的结点

member

List 结构体在上述的“包裹 list 结构体的结构体”中的变量名

`list_first_entry(ptr, type, member)`

描述

获取链表中的第一个元素。请注意，该宏要求链表非空，否则会出错。

参数

与*list_entry()*相同

`list_first_entry_or_null(ptr, type, member)`

描述

获取链表中的第一个元素。若链表为空，则返回 NULL。

参数

与*list_entry()*相同

list_last_entry(ptr, type, member)

描述

获取链表中的最后一个元素。请注意，该宏要求链表非空，否则会出错。

参数

与*list_entry()*相同

list_last_entry_or_full(ptr, type, member)

描述

获取链表中的最后一个元素。若链表为空，则返回 NULL。

参数

与*list_entry()*相同

list_next_entry(pos, member)

描述

获取链表中的下一个元素

参数

pos

指向当前的外层结构体的指针

member

链表结构体在外层结构体内的变量名

list_prev_entry(pos, member)

描述

获取链表中的上一个元素

参数

与*list_next_entry()*相同

list_for_each(ptr, head)

描述

遍历整个链表（从前往后）

参数

ptr

指向 List 结构体的指针

head

指向链表头结点的指针 (struct List*)

list_for_each_prev(ptr, head)

描述

遍历整个链表（从后往前）

参数

与*list_for_each()*相同

list_for_each_safe(ptr, n, head)

描述

从前往后遍历整个链表（支持删除当前链表结点）

该宏通过暂存中间变量，防止在迭代链表的过程中，由于删除了当前 ptr 所指向的链表结点从而造成错误.

参数

ptr

指向 List 结构体的指针

n

用于存储临时值的 List 类型的指针

head

指向链表头结点的指针 (struct List*)

list_for_each_prev_safe(ptr, n, head)

描述

从后往前遍历整个链表.（支持删除当前链表结点）

该宏通过暂存中间变量，防止在迭代链表的过程中，由于删除了当前 ptr 所指向的链表结点从而造成错误.

参数

与*list_for_each_safe()*相同

list_for_each_entry(pos, head, member)

描述

从头开始迭代给定类型的链表

参数

pos

指向特定类型的结构体的指针

head

指向链表头结点的指针 (struct List*)

member

struct List 在 pos 的结构体中的成员变量名

`list_for_each_entry_reverse(pos, head, member)`

描述

逆序迭代给定类型的链表

参数

与*list_for_each_entry()*相同

`list_for_each_entry_safe(pos, n, head, member)`

描述

从头开始迭代给定类型的链表（支持删除当前链表结点）

参数

pos

指向特定类型的结构体的指针

n

用于存储临时值的，和 pos 相同类型的指针

head

指向链表头结点的指针 (struct List*)

member

struct List 在 pos 的结构体中的成员变量名

list_prepare_entry(pos, head, member)

描述

为*list_for_each_entry_continue()*准备一个’pos’结构体

参数

pos

指向特定类型的结构体的，用作迭代起点的指针

head

指向要开始迭代的 struct List 结构体的指针

member

struct List 在 pos 的结构体中的成员变量名

list_for_each_entry_continue(pos, head, member)

描述

从指定的位置的【下一个元素开始】，继续迭代给定的链表

参数

pos

指向特定类型的结构体的指针。该指针用作迭代的指针。

head

指向要开始迭代的 struct List 结构体的指针

member

struct List 在 pos 的结构体中的成员变量名

list_for_each_entry_continue_reverse(pos, head, member)

描述

从指定的位置的【上一个元素开始】,【逆序】迭代给定的链表

参数

与*list_for_each_entry_continue()*的相同

list_for_each_entry_from(pos, head, member)

描述

从指定的位置开始, 继续迭代给定的链表

参数

与*list_for_each_entry_continue()*的相同

list_for_each_entry_safe_continue(pos, n, head, member)

描述

从指定的位置的【下一个元素开始】, 继续迭代给定的链表.（支持删除当前链表结点）

参数

pos

指向特定类型的结构体的指针。该指针用作迭代的指针。

n

用于存储临时值的，和 pos 相同类型的指针

head

指向要开始迭代的 struct List 结构体的指针

member

struct List 在 pos 的结构体中的成员变量名

`list_for_each_entry_safe_continue_reverse(pos, n, head, member)`

描述

从指定的位置的【上一个元素开始】,【逆序】迭代给定的链表。(支持删除当前链表结点)

参数

与*list_for_each_entry_safe_continue()*的相同

`list_for_each_entry_safe_from(pos, n, head, member)`

描述

从指定的位置开始, 继续迭代给定的链表. (支持删除当前链表结点)

参数

与*list_for_each_entry_safe_continue()*的相同

4.1.2 基础 C 函数库

内核编程与应用层编程不同，你将无法使用 LibC 中的函数来进行编程。为此，内核实现了一些常用的 C 语言函数，并尽量使其与标准 C 库中的函数行为相近。值得注意的是，这些函数的行为可能与标准 C 库函数不同，请在使用时仔细阅读以下文档，这将会为你带来帮助。

字符串操作

`int strlen(const char *s)`

描述

测量并返回字符串长度。

参数

src

源字符串

```
long strnlen(const char *src, unsigned long maxlen)
```

描述

测量并返回字符串长度。当字符串长度大于 maxlen 时，返回 maxlen

参数

src

源字符串

maxlen

最大长度

```
long strnlen_user(const char *src, unsigned long maxlen)
```

描述

测量并返回字符串长度。当字符串长度大于 maxlen 时，返回 maxlen。

该函数会进行地址空间校验，要求 src 字符串必须来自用户空间。当源字符串来自内核空间时，将返回 0.

参数

src

源字符串，地址位于用户空间

maxlen

最大长度

```
char *strncpy(char *dst, const char *src, long count)
```

描述

拷贝长度为 count 个字节的字符串，返回 dst 字符串

参数

src

源字符串

dst

目标字符串

count

要拷贝的源字符串的长度

```
char *strcpy(char *dst, const char *src)
```

描述

拷贝源字符串，返回 dst 字符串

参数

src

源字符串

dst

目标字符串

```
long strncpy_from_user(char *dst, const char *src, unsigned long size)
```

描述

从用户空间拷贝长度为 count 个字节的字符串到内核空间，返回拷贝的字符串的大小

该函数会对字符串的地址空间进行校验，防止出现地址空间越界的问题。

参数

src

源字符串

dst

目标字符串

size

要拷贝的源字符串的长度

```
int strcmp(char *FirstPart, char *SecondPart)
```

描述

比较两个字符串的大小。

返回值

情况	返回值
FirstPart == SecondPart	0
FirstPart > SecondPart	1
FirstPart < SecondPart	-1

参数

FirstPart

第一个字符串

SecondPart

第二个字符串

```
printk(const char* fmt, ...)
```

描述

该宏能够在控制台上以黑底白字格式化输出字符串.

参数

fmt

源格式字符串

...

可变参数

```
printk_color(unsigned int FRcolor, unsigned int BKcolor, const char* fmt, ...)
```

描述

在控制台上以指定前景色和背景色格式化输出字符串.

参数

FRcolor

前景色

BKcolor

背景色

fmt

源格式字符串

...

可变参数

```
int vsprintf(char *buf, const char *fmt, va_list args)
```

描述

按照 fmt 格式化字符串，并将结果输出到 buf 中，返回写入 buf 的字符数量。

参数

buf

输出缓冲区

fmt

源格式字符串

args

可变参数列表

```
int sprintk(char *buf, const char *fmt, ...)
```

描述

按照 fmt 格式化字符串，并将结果输出到 buf 中，返回写入 buf 的字符数量。

参数

buf

输出缓冲区

fmt

源格式字符串

...

可变参数

内存操作

```
void *memcpy(void *dst, const void *src, uint64_t size)
```

描述

将内存从 src 处拷贝到 dst 处。

参数

dst

指向目标地址的指针

src

指向源地址的指针

size

待拷贝的数据大小

```
void *memmove(void *dst, const void *src, uint64_t size)
```

描述

与 memcpy() 类似，但是在源数据区域与目标数据区域之间存在重合时，该函数能防止数据被错误的覆盖。

参数

dst

指向目标地址的指针

src

指向源地址的指针

size

待拷贝的数据大小

4.1.3 CRC 函数

函数列表

```
uint8_t crc7(uint8_t crc, const uint8_t *buffer, size_t len)

uint8_t crc8(uint8_t crc, const uint8_t *buffer, size_t len)

uint16_t crc16(uint16_t crc, uint8_t const *buffer, size_t len)

uint32_t crc32(uint32_t crc, uint8_t const *buffer, size_t len)

uint64_t crc64(uint64_t crc, uint8_t const *buffer, size_t len)
```

描述

用于计算循环冗余校验码

参数说明

crc

传入的 CRC 初始值

buffer

待处理的数据缓冲区

len

缓冲区大小（字节）

4.2 原子变量

4.2.1 简介

DragonOS 实现了原子变量，类型为 atomic_t. 原子变量是基于具体体系结构的原子操作指令实现的。具体实现在 `kernel/common/atomic.h` 中。

4.2.2 API

请注意，以下 API 均为原子操作。

```
inline void atomic_add(atomic_t *ato, long val)
```

描述

原子变量增加指定值

参数

ato

原子变量对象

val

变量要增加的值

```
inline void atomic_sub(atomic_t *ato, long val)
```

描述

原子变量减去指定值

参数

ato

原子变量对象

val

变量要被减去的值

```
void atomic_inc(atomic_t *ato)
```

描述

原子变量自增 1

参数

ato

原子变量对象

```
void atomic_dec(atomic_t *ato)
```

描述

原子变量自减 1

参数

ato

原子变量对象

```
inline void atomic_set_mask(atomic_t *ato, long mask)
```

描述

将原子变量的值与 mask 变量进行 or 运算

参数

ato

原子变量对象

mask

与原子变量进行 or 运算的变量

```
inline void atomic_clear_mask(atomic_t *ato, long mask)
```

描述

将原子变量的值与 mask 变量进行 and 运算

参数

ato

原子变量对象

mask

与原子变量进行 and 运算的变量

4.3 内核数据结构

内核中实现了常用的几种数据结构，这里是他们的 api 文档。

4.3.1 kfifo 先进先出缓冲区

kfifo 先进先出缓冲区定义于 `common/kfifo.h` 中。您可以使用它，创建指定大小的 fifo 缓冲区（最大大小为 4GB）

kfifo_alloc

```
int kfifo_alloc(struct kfifo_t *fifo, uint32_t size, uint64_t reserved)
```

描述

通过动态方式初始化 kfifo 缓冲队列。fifo 缓冲区的 buffer 将由该函数进行申请。

参数

fifo

kfifo 队列结构体的指针

size

缓冲区大小（单位：bytes）

reserved

当前字段保留，请将其置为 0

返回值

当返回值为 0 时，表示正常初始化成功，否则返回对应的 errno

kfifo_init

```
void kfifo_init(struct kfifo_t *fifo, void *buffer, uint32_t size)
```

描述

使用指定的缓冲区来初始化 kfifo 缓冲队列

参数

fifo

kfifo 队列结构体的指针

buffer

缓冲区基地址指针

size

缓冲区大小（单位：bytes）

kfifo_free_alloc

```
void kfifo_free_alloc(struct kfifo_t* fifo)
```

描述

释放通过 kfifo_alloc 创建的 fifo 缓冲区. 请勿通过该函数释放其他方式创建的 kfifo 缓冲区。

参数

fifo

kfifo 队列结构体的指针

kfifo_in

```
uint32_t kfifo_in(struct kfifo_t *fifo, const void *from, uint32_t size)
```

描述

向 kfifo 缓冲区推入指定大小的数据。当队列中空间不足时，则不推入数据。

参数

fifo

kfifo 队列结构体的指针

from

源数据基地址指针

size

数据大小（单位：bytes）

返回值

返回成功被推入的数据的大小。

kfifo_out

```
uint32_t kfifo_out(struct kfifo_t *fifo, void *to, uint32_t size)
```

描述

从 kfifo 缓冲区取出数据，并从队列中删除数据。当队列中数据量不足时，则不取出。

参数

fifo

kfifo 队列结构体的指针

to

目标缓冲区基地址指针

size

数据大小（单位：bytes）

返回值

返回成功被取出的数据的大小。

kfifo_out_peek

```
uint32_t kfifo_out_peek(struct kfifo_t *fifo, void *to, uint32_t size)
```

描述

从 kfifo 缓冲区取出数据，但是不从队列中删除数据。当队列中数据量不足时，则不取出。

参数

fifo

kfifo 队列结构体的指针

to

目标缓冲区基地址指针

size

数据大小（单位：bytes）

返回值

返回成功被取出的数据的大小。

kfifo_reset

```
kfifo_reset(fifo)
```

描述

忽略 kfifo 队列中的所有内容，并把输入和输出偏移量都归零

参数

fifo

kfifo 队列结构体的指针

kfifo_reset_out

```
kfifo_reset_out(fifo)
```

描述

忽略 kfifo 队列中的所有内容，并将输入偏移量赋值给输出偏移量

参数

fifo

kfifo 队列结构体的指针

kfifo_total_size

```
kfifo_total_size(fifo)
```

描述

获取 kfifo 缓冲区的最大大小

参数

fifo

kfifo 队列结构体的指针

返回值

缓冲区最大大小

kfifo_size

```
kfifo_size(fifo)
```

描述

获取 kfifo 缓冲区当前已使用的大小

参数

fifo

kfifo 队列结构体的指针

返回值

缓冲区当前已使用的大小

kfifo_empty

```
kfifo_empty(fifo)
```

描述

判断 kfifo 缓冲区当前是否为空

参数

fifo

kfifo 队列结构体的指针

返回值

情况	返回值
空	1
非空	0

kfifo_full

```
kfifo_full(fifo)
```

描述

判断 kfifo 缓冲区当前是否为满

参数

fifo

kfifo 队列结构体的指针

返回值

情况	返回值
满	1
不满	0

4.3.2 ID Allocation

ida 的主要作用是分配 + 管理 id. 它能分配一个最小的, 未被分配出去的 id. 当您需要管理某个数据结构时, 可能需要使用 id 来区分不同的目标. 这个时候, ida 将会是很好的选择. 因为 ida 的十分高效, 运行常数相对数组更小, 而且提供了基本管理 id 需要用到的功能, 值得您试一试.

IDA 定义于 `idr.h` 文件中. 您通过 `DECLARE_IDA(my_ida)` 来创建一个 ida 对象, 或者 `struct ida my_ida; ida_init(&my_ida);` 来初始化一个 ida.

ida_init

```
void ida_init(struct ida *ida_p)
```

描述

通初始化 IDA, 你需要保证调用函数之前, ida 的 free_list 为空, 否则会导致内存泄漏.

参数

ida_p

指向 ida 的指针

返回值

无返回值

ida_preload

```
int ida_preload(struct ida *ida_p, gfp_t gfp_mask)
```

描述

为 ida 预分配空间. 您可以不自行调用, 因为当 ida 需要空间的时候, 内部会自行使用 `kmalloc` 函数获取空间. 当然, 设计这个函数的目的是为了让您有更多的选择. 当您提前调用这个函数, 可以避免之后在开辟空间上的时间开销.

参数

ida_p

指向 ida 的指针

gfp_mask

保留参数, 目前尚未使用.

返回值

如果分配成功, 将返回 0; 否则返回负数错误码, 有可能是内存空间不够.

ida_alloc

```
int ida_alloc(struct ida *ida_p, int *p_id)
```

描述

获取一个空闲 ID. 您需要注意, 返回值是成功/错误码.

参数

ida_p

指向 ida 的指针

p_id

您需要传入一个 int 变量的指针, 如果成功分配 ID, ID 将会存储在该指针所指向的地址.

返回值

如果分配成功, 将返回 0; 否则返回负数错误码, 有可能是内存空间不够.

ida_count

```
bool ida_count(struct ida *ida_p, int id)
```

描述

查询一个 ID 是否被分配.

参数

ida_p

指向 ida 的指针

id

您查询该 ID 是否被分配.

返回值

如果分配, 将返回 true; 否则返回 false.

ida_remove

```
void ida_remove(struct ida *ida_p, int id)
```

描述

删除一个已经分配的 ID. 如果该 ID 不存在, 该函数不会产生异常错误, 因为在检测到该 ID 不存在的时候, 函数将会自动退出.

参数

ida_p

指向 ida 的指针

id

您要删除的 id.

返回值

无返回值.

ida_destroy

```
void ida_destroy(struct ida *ida_p)
```

描述

释放一个 IDA 所有的空间, 同时删除 ida 的所有已经分配的 id.(所以您不用担心删除 id 之后, ida 还会占用大量空间.)

参数

ida_p

指向 ida 的指针

返回值

无返回值

ida_empty

```
void ida_empty(struct ida *ida_p)
```

描述

查询一个 ida 是否为空

参数

ida_p

指向 ida 的指针

返回值

ida 为空则返回 true，否则返回 false。

4.3.3 IDR

idr 是一个基于 radix-tree 的 ID-pointer 的数据结构. 该数据结构提供了建 id 与数据指针绑定的功能, 它的主要功能有以下 4 个:

1. 获取一个 ID, 并且将该 ID 与一个指针绑定
2. 删除一个已分配的 ID
3. 根据 ID 查找对应的指针
4. 根据 ID 使用新的 ptr 替换旧的 ptr

   您可以使用 `DECLARE_idr(my_idr)` 来创建一个 idr。或者您也可以使用 `struct idr my_idr; idr_init(my_idr);` 这两句话创建一个 idr。 至于什么是 radix-tree，您可以把他简单理解为一个向上生长的多叉树，在实现中，我们选取了 64 叉树。

idr_init

```
void idr_init(struct idr *idp)
```

描述

通初始化 IDR, 你需要保证调用函数之前, idr 的 free_list 为空, 否则会导致内存泄漏.

参数

idp

指向 idr 的指针

返回值

无返回值

idr_preload

```
int idr_preload(struct idr *idp, gfp_t gfp_mask)
```

描述

为 idr 预分配空间. 您可以不自行调用, 因为当 idr 需要空间的时候, 内部会自行使用 `kmalloc` 函数获取空间. 当然, 设计这个函数的目的是为了让您有更多的选择. 当您提前调用这个函数, 可以避免之后在开辟空间上的时间开销.

参数

idp

指向 idr 的指针

gfp_mask

保留参数, 目前尚未使用.

返回值

如果分配成功, 将返回 0; 否则返回负数错误码, 有可能是内存空间不够.

idr_alloc

```
int idr_alloc(struct idr *idp, void *ptr, int *id)
```

描述

获取一个空闲 ID. 您需要注意, 返回值是成功/错误码. 调用这个函数，需要您保证 ptr 是非空的，即: `ptr != NULL`, 否则将会影响 `idr_find/idr_find_next/idr_find_next_getid/...` 等函数的使用。(具体请看这三个函数的说明，当然，只会影响到您的使用体验，并不会影响到 idr 内部函数的决策和逻辑)

参数

idp

指向 ida 的指针

ptr

指向数据的指针

id

您需要传入一个 int 变量的指针, 如果成功分配 ID, ID 将会存储在该指针所指向的地址.

返回值

如果分配成功, 将返回 0; 否则返回负数错误码, 有可能是内存空间不够.

idr_remove

```
void* idr_remove(struct idr *idp, int id)
```

描述

删除一个 id, 但是不释放对应的 ptr 指向的空间, 同时返回这个被删除 id 所对应的 ptr。 如果该 ID 不存在, 该函数不会产生异常错误, 因为在检测到该 ID 不存在的时候, 函数将会自动退出，并返回 NULL。

参数

idp

指向 idr 的指针

id

您要删除的 id.

返回值

如果删除成功，就返回被删除 id 所对应的 ptr；否则返回 NULL。注意：如果这个 id 本来就和 NULL 绑定，那么也会返回 NULL

idr_remove_all

```
void idr_remove_all(struct idr *idp)
```

描述

删除 idr 的所有已经分配的 id.(所以您不用担心删除 id 之后, idr 还会占用大量空间。)

但是你需要注意的是，调用这个函数是不会释放数据指针指向的空间的。所以您调用该函数之前，确保 IDR 内部的数据指针被保存。否则当 IDR 删除所有 ID 之后，将会造成内存泄漏。

参数

idp

指向 idr 的指针

返回值

无返回值

idr_destroy

```
void idr_destroy(struct idr *idp)
```

描述

释放一个 IDR 所有的空间, 同时删除 idr 的所有已经分配的 id.(所以您不用担心删除 id 之后, ida 还会占用大量空间.) - 和 `idr_remove_all` 的区别是，释放掉所有的空间 (包括 free_list 的预分配空间)。

参数

idp

指向 idr 的指针

返回值

无返回值

idr_find

```
void *idr_find(struct idr *idp, int id)
```

描述

查询一个 ID 所绑定的数据指针

参数

idp

指向 idr 的指针

id

您查询该 ID 的数据指针

返回值

如果分配, 将返回该 ID 对应的数据指针; 否则返回 NULL.(注意，返回 NULL 不一定代表这 ID 不存在，有可能该 ID 就是与空指针绑定。)　　当然，我们也提供了 `idr_count` 函数来判断 id 是否被分配，具体请查看 idr_count 介绍。

idr_find_next

```
void *idr_find_next(struct idr *idp, int start_id)
```

描述

传进一个 start_id，返回满足“id 大于 start_id 的最小 id”所对应的数据指针。

参数

idp

指向 idr 的指针

start_id

您提供的 ID 限制

返回值

如果分配, 将返回该 ID 对应的数据指针; 否则返回 NULL.(注意，返回 NULL 不一定代表这 ID 不存在，有可能该 ID 就是与空指针绑定。)　　当然，我们也提供了 `idr_count` 函数来判断 id 是否被分配，具体请查看 idr_count 介绍。

idr_find_next_getid

```
void *idr_find_next_getid(struct idr *idp, int start_id, int *nextid)
```

描述

传进一个 start_id，返回满足“id 大于 start_id 的最小 id”所对应的数据指针。同时，你获取到这个满足条件的最小 id，即参数中的 *nextid。

参数

idp

指向 idr 的指针

start_id

您提供的 ID 限制

返回值

如果分配, 将返回该 ID 对应的数据指针; 否则返回 NULL.(注意，返回 NULL 不一定代表这 ID 不存在，有可能该 ID 就是与空指针绑定。)　　当然，我们也提供了 `idr_count` 函数来判断 id 是否被分配，具体请查看 idr_count 介绍。

idr_replace

```
int idr_replace(struct idr *idp, void *ptr, int id)
```

描述

传进一个 ptr，使用该 ptr 替换掉 id 所对应的 Old_ptr。

参数

idp

指向 idr 的指针

ptr

您要替换原来的 old_ptr 的新指针

id

您要替换的指针所对应的 id

返回值

0 代表成功，否则就是错误码 - 代表错误。

idr_replace_get_old

```
int idr_replace_get_old(struct idr *idp, void *ptr, int id, void **oldptr)
```

描述

传进一个 ptr，使用该 ptr 替换掉 id 所对应的 Old_ptr，同时你可以获取到 old_ptr。

参数

idp

指向 idr 的指针

ptr

您要替换原来的 old_ptr 的新指针

id

您要替换的指针所对应的 id

old_ptr

您需要传进该 (void**) 指针，old_ptr 将会存放在该指针所指向的地址。

返回值

0 代表成功，否则就是错误码 - 代表错误。

idr_empty

```
void idr_empty(struct idr *idp)
```

描述

查询一个 idr 是否为空

参数

idp

指向 idr 的指针

返回值

idr 为空则返回 true，否则返回 false。

idr_count

```
bool idr_count(struct idr *idp, int id)
```

描述

查询一个 ID 是否被分配.

参数

ida_p

指向 idr 的指针

id

您查询该 ID 是否被分配.

返回值

如果分配, 将返回 true; 否则返回 false.

4.4 内存管理

这里快速讲解了如何在 DragonOS 中分配、使用内存。以便您能快速的了解这个模块。

详细的内存管理模块的文档请参见：内存管理文档

4.4.1 内存分配指南

DragonOS 提供了一些用于内存分配的 api。您可以使用 *kmalloc* 来分配小的内存块，也可以使用 *alloc_pages* 分配连续的 2MB 大小的内存页面。

选择合适的内存分配器

在内核中，最直接、最简单的分配内存的方式就是，使用 `kmalloc()` 函数进行分配。并且，出于安全起见，除非内存在分配后一定会被覆盖，且您能确保内存中的脏数据一定不会对程序造成影响，在其余情况下，我们建议使用 `kzalloc()` 进行内存分配，它将会在 `kmalloc()` 的基础上，把申请到的内存进行清零。

您可以通过 `kmalloc()` 函数分配得到 32bytes 到 1MBytes 之间的内存对象。并且，这些内存对象具有以下的性质：

- 内存起始地址及大小按照 2 次幂对齐。（比如，申请的是 80bytes 的内存空间，那么获得的内存对象大小为 128bytes 且内存地址按照 128bytes 对齐）

对于需要大量连续内存的分配，可以使用 `alloc_pages()` 向页面分配器申请连续的内存页。

当内存空间不再被使用时，那么必须释放他们。若您使用的是 `kmalloc()` 分配的内存，那么您需要使用 `kfree()` 释放它。若是使用 `alloc_pages()` 分配的内存，则需要使用 `free_pages()` 来释放它们。

4.4.2 内存管理 API

SLAB 内存池

SLAB 内存池提供小内存对象的分配功能。

`void *kmalloc(unsigned long size, gfp_t gfp)`

获取小块的内存。

描述

kmalloc 用于获取那些小于 2M 内存页大小的内存对象。可分配的内存对象大小为 32bytes~1MBytes. 且分配的内存块大小、起始地址按照 2 的 n 次幂进行对齐。（比如，申请的是 80bytes 的内存空间，那么获得的内存对象大小为 128bytes 且内存地址按照 128bytes 对齐）

参数

size

内存对象的大小

gfp

标志位

```
void *kzalloc(unsigned long size, gfp_t gfp)
```

描述

获取小块的内存，并将其清零。其余功能与 kmalloc 相同。

参数

size

内存对象的大小

gfp

标志位

```
unsigned long kfree(void *address)
```

释放从 slab 分配的内存。

描述

该函数用于释放通过 kmalloc 申请的内存。如果 `address` 为 NULL，则函数被调用后，无事发生。

请不要通过这个函数释放那些不是从 `kmalloc()` 或 `kzalloc()` 申请的内存，否则将会导致系统崩溃。

参数

address

指向内存对象的起始地址的指针

物理页管理

DragonOS 支持对物理页的直接操作

```
struct Page *alloc_pages(unsigned int zone_select, int num, ul flags)
```

描述

从物理页管理单元中申请一段连续的物理页

参数

zone_select

要申请的物理页所位于的内存区域

可选值:

- ZONE_DMA DMA 映射专用区域
- ZONE_NORMAL 正常的物理内存区域，已在页表高地址处映射
- ZONE_UNMAPPED_IN_PGT 尚未在页表中映射的区域

num

要申请的连续物理页的数目，该值应当小于 64

flags

分配的页面要被设置成的属性

可选值:

- PAGE_PGT_MAPPED 页面在页表中已被映射
- PAGE_KERNEL_INIT 内核初始化所占用的页
- PAGE_DEVICE 设备 MMIO 映射的内存
- PAGE_KERNEL 内核层页
- PAGE_SHARED 共享页

返回值

成功

成功申请则返回指向起始页面的 Page 结构体的指针

失败

当 ZONE 错误或内存不足时，返回 `NULL`

```
void free_pages(struct Page *page, int number)
```

描述

从物理页管理单元中释放一段连续的物理页。

参数

page

要释放的第一个物理页的 Page 结构体

number

要释放的连续内存页的数量。该值应小于 64

页表管理

```
int mm_map_phys_addr(ul virt_addr_start, ul phys_addr_start, ul length, ul
flags, bool use4k)
```

描述

将一段物理地址映射到当前页表的指定虚拟地址处

参数

virt_addr_start

虚拟地址的起始地址

phys_addr_start

物理地址的起始地址

length

要映射的地址空间的长度

flags

页表项的属性

use4k

使用 4 级页表，将地址区域映射为若干 4K 页

```
int mm_map_proc_page_table(ul proc_page_table_addr, bool is_phys, ul
virt_addr_start, ul phys_addr_start, ul length, ul flags, bool user, bool
flush, bool use4k)
```

描述

将一段物理地址映射到指定页表的指定虚拟地址处

参数

proc_page_table_addr

指定的顶层页表的起始地址

is_phys

该顶层页表地址是否为物理地址

virt_addr_start

虚拟地址的起始地址

phys_addr_start

物理地址的起始地址

length

要映射的地址空间的长度

flags

页表项的属性

user

页面是否为用户态可访问

flush

完成映射后，是否刷新 TLB

use4k

使用 4 级页表，将地址区域映射为若干 4K 页

返回值

- 映射成功：0
- 映射失败：-EFAULT

```
void mm_unmap_proc_table(ul proc_page_table_addr, bool is_phys, ul
virt_addr_start, ul length)
```

描述

取消给定页表中的指定地址空间的页表项映射。

参数

proc_page_table_addr

指定的顶层页表的基地址

is_phys

该顶层页表地址是否为物理地址

virt_addr_start

虚拟地址的起始地址

length

要取消映射的地址空间的长度

`mm_unmap_addr(virt_addr, length)`

描述

该宏定义用于取消当前进程的页表中的指定地址空间的页表项映射。

参数

virt_addr

虚拟地址的起始地址

length

要取消映射的地址空间的长度

内存信息获取

`struct mm_stat_t mm_stat()`

描述

获取计算机目前的内存空间使用情况

参数

无

返回值

返回值是一个 `mm_mstat_t` 结构体，该结构体定义于 `mm/mm.h` 中。其中包含了以下信息（单位均为字节）：

参数名	解释
total	计算机的总内存数量大小
used	已使用的内存大小
free	空闲物理页所占的内存大小
shared	共享的内存大小
cache_used	位于 slab 缓冲区中的已使用的内存大小
cache_free	位于 slab 缓冲区中的空闲的内存大小
available	系统总空闲内存大小（包括 kmalloc 缓冲区）

CHAPTER 5

锁

这里是 DragonOS 的锁变量的说明文档。

5.1 锁的类型及其规则

5.1.1 简介

DragonOS 内核实现了一些锁，大致可以分为两类：

- 休眠锁
- 自旋锁

5.1.2 锁的类型

休眠锁

休眠锁只能在可抢占的上下文之中被获取。

在 DragonOS 之中，实现了以下的休眠锁：

- semaphore
- mutex_t

自旋锁

- spinlock_t

进程在获取自旋锁后，将改变 pcb 中的锁变量持有计数，从而隐式地禁止了抢占。为了获得更多灵活的操作，spinlock 还提供了以下的方法：

后缀	说明
_irq()	在加锁时关闭中断/在放锁时开启中断
_irqsave()/_irqrestore()	在加锁时保存中断状态，并关中断/在放锁时恢复中断状态

当您同时需要使用自旋锁以及引用计数时，一个好的方法是：使用 `lockref`. 这是一种额外的加速技术，能额外提供“无锁修改引用计数”的功能。详情请见：*lockref*

5.1.3 详细介绍

semaphore 信号量

semaphore 信号量是基于计数实现的。

当可用资源不足时，尝试对 semaphore 执行 down 操作的进程将会被休眠，直到资源可用。

mutex 互斥量

mutex 是一种轻量级的同步原语，只有 0 和 1 两种状态。

当 mutex 被占用时，尝试对 mutex 进行加锁操作的进程将会被休眠，直到资源可用。

特性

- 同一时间只有 1 个任务可以持有 mutex
- 不允许递归地加锁、解锁
- 只允许通过 mutex 的 api 来操作 mutex
- 在硬中断、软中断中不能使用 mutex

数据结构

mutex 定义在 `common/mutex.h` 中。其数据类型如下所示：

```
typedef struct
{

    atomic_t count; // 锁计数。1->已解锁。 0->已上锁,且有可能存在等待者
    spinlock_t wait_lock;   // mutex操作锁，用于对mutex的list的操作进行加锁
    struct List wait_list;  // Mutex的等待队列
} mutex_t;
```

API

mutex_init

void mutex_init(mutex_t *lock)

初始化一个 mutex 对象。

mutex_lock

void mutex_lock(mutex_t *lock)

对一个 mutex 对象加锁。若 mutex 当前被其他进程持有，则当前进程进入休眠状态。

mutex_unlock

void mutex_unlock(mutex_t *lock)

对一个 mutex 对象解锁。若 mutex 的等待队列中有其他的进程，则唤醒下一个进程。

mutex_trylock

void mutex_trylock(mutex_t *lock)

尝试对一个 mutex 对象加锁。若 mutex 当前被其他进程持有，则返回 0. 否则，加锁成功，返回 1.

mutex_is_locked

void mutex_is_locked(mutex_t *lock)

判断 mutex 是否已被加锁。若给定的 mutex 已处于上锁状态，则返回 1，否则返回 0。

5.2 lockref

lockref 是将自旋锁与引用计数变量融合在连续、对齐的 8 字节内的一种技术。

5.2.1 lockref 结构

```
struct lockref
{
    union
    {
#ifdef __LOCKREF_ENABLE_CMPXCHG__
        aligned_u64 lock_count; //␣
↪通过该变量的声明，使得整个lockref的地址按照8字节对齐
#endif
        struct
        {
            spinlock_t lock;
            int count;
        };
    };
};
```

5.2.2 特性描述

由于在高负载的情况下，系统会频繁的执行“锁定-改变引用变量-解锁”的操作，这期间很可能出现 spinlock 和引用计数跨缓存行的情况，这将会大大降低性能。lockref 通过强制对齐，尽可能的降低缓存行的占用数量，使得性能得到提升。

并且，在 x64 体系结构下，还通过 cmpxchg() 指令，实现了无锁快速路径。不需要对自旋锁加锁即可更改引用计数的值，进一步提升性能。当快速路径不存在（对于未支持的体系结构）或者尝试超时后，将会退化成“锁定-改变引用变量-解锁”的操作。此时由于 lockref 强制对齐，只涉及到 1 个缓存行，因此性能比原先的 spinlock+ref_count 的模式要高。

5.2.3 关于 cmpxchg_loop

在改变引用计数时，cmpxchg 先确保没有别的线程持有锁，然后改变引用计数，同时通过 `lock cmpxchg` 指令验证在更改发生时，没有其他线程持有锁，并且当前的目标 lockref 的值与 old 变量中存储的一致，从而将新值存储到目标 lockref。这种无锁操作能极大的提升性能。如果不符合上述条件，在多次尝试后，将退化成传统的加锁方式来更改引用计数。

5.2.4 参考资料

Introducing lockrefs - LWN.net, Jonathan Corbet

CHAPTER 6

进程管理模块

6.1 kthread 内核线程

内核线程模块定义在 common/kthread.h 中，提供对内核线程的及支持功能。内核线程作为内核的“分身”，能够提升系统的并行化程度以及故障容错能力。

6.1.1 原理

每个内核线程都运行在内核态，执行其特定的任务。

内核线程的创建是通过调用 kthread_create() 或者 kthread_run() 宏，向 kthreadd 守护线程发送创建任务来实现的。也就是说，内核线程的创建，最终是由 kthreadd 来完成。

当内核线程被创建后，虽然会加入调度队列，但是当其被第一次调度，执行引导程序 kthread() 后，将进入休眠状态。直到其他模块使用 process_wakeup()，它才会真正开始运行。

当内核其他模块想要停止一个内核线程的时候，可以调用 kthread_stop() 函数。该函数将会置位内核线程的 worker_private 中的 KTHREAD_SHOULD_STOP 标志位，并等待内核线程的退出，然后获得返回值并清理内核线程的 pcb。

内核线程应当经常检查 KTHREAD_SHOULD_STOP 标志位，以确定其是否要退出。当检测到该标志位被置位时，内核线程应当完成数据清理工作，并调用 kthread_exit() 或直接返回一个返回码，以退出内核线程。

6.1.2 创建内核线程

kthread_create()

原型

```
kthread_create(thread_fn, data, name_fmt, arg...)
```

简介

在当前 NUMA 结点上创建一个内核线程（DragonOS 目前暂不支持 NUMA，因此 node 可忽略。）

请注意，该宏会创建一个内核线程，并将其设置为停止状态.

参数

thread_fn

该内核线程要执行的函数

data

传递给 *thread_fn* 的参数数据

name_fmt

printf-style format string for the thread name

arg

name_fmt 的参数

返回值

创建好的内核线程的 pcb

kthread_run()

原型

```
kthread_run(thread_fn, data, name_fmt, ...)
```

简介

创建内核线程并加入调度队列。

该宏定义是 `kthread_create()` 的简单封装，提供创建了内核线程后，立即运行的功能。

6.1.3 停止内核线程

kthread_stop()

原型

```
int kthread_stop(struct process_control_block * pcb)
```

简介

当外部模块希望停止一个内核线程时，调用该函数，向 kthread 发送停止消息，请求其结束。并等待其退出，返回内核线程的退出返回值。

参数

pcb

内核线程的 pcb

返回值

内核线程的退出返回码。

kthread_should_stop()

原型

```
bool kthread_should_stop(void)
```

简介

内核线程可以调用该函数得知是否有其他进程请求结束当前内核线程。

返回值

一个 bool 变量

值	解释
true	有其他进程请求结束该内核线程
false	该内核线程没有收到停止消息

kthread_exit()

原型

```
void kthread_exit(long result)
```

简介

让当前内核线程退出，并返回 result 参数给 kthread_stop() 函数。

参数

result

内核线程的退出返回码

CHAPTER 7

DragonOS 调度

这里是 DragonOS 中，与进程调度相关的说明文档。

7.1 与“等待”相关的 api

如果几个进程需要等待某个事件发生，才能被运行，那么就需要一种“等待”的机制，以实现进程同步。

7.1.1 一. wait_queue 等待队列

wait_queue 是一种进程同步机制，中文名为“等待队列”。它可以将当前进程挂起，并在时机成熟时，由另一个进程唤醒他们。

当您需要等待一个事件完成时，使用 wait_queue 机制能减少进程同步的开销。相比于滥用自旋锁以及信号量，或者是循环使用 usleep(1000) 这样的函数来完成同步，wait_queue 是一个高效的解决方案。

警告： `wait_queue.h` 中的等待队列的实现并没有把队列头独立出来，同时没有考虑为等待队列加锁。所以在后来的开发中加入了 `wait_queue_head.h` 的队列头实现，实质上就是链表 + 自旋锁。它与 `wait_queue.h` 中的队列是兼容的，当你使用 `struct wait_queue_head` 作为队列头时，你同样可以使用等待队列添加节点的函数。

但是在之后的版本中可能会把两者合并，目前仍然没有进行，且存在头文件相互引用的问题:

- “spin_lock.h”引用了“wait_queue.h”

- “wait_queue_head.h”引用了“spin_lock.h”；

所以在合并之前必须解决这个问题。

简单用法

等待队列的使用方法主要分为以下几部分：

- 创建并初始化一个等待队列
- 使用 `wait_queue_sleep_on_` 系列的函数，将当前进程挂起。晚挂起的进程将排在队列的尾部。
- 通过 `wait_queue_wakeup()` 函数，依次唤醒在等待队列上的进程，将其加入调度队列

要使用 wait_queue，您需要 `#include<common/wait_queue.h>`，并创建一个 `wait_queue_node_t` 类型的变量，作为等待队列的头部。这个结构体只包含两个成员变量：

```
typedef struct
{
    struct List wait_list;
    struct process_control_block *pcb;
} wait_queue_node_t;
```

对于等待队列，这里有一个好的命名方法：

```
wait_queue_node_t wq_keyboard_interrupt_received;
```

这样的命名方式能增加代码的可读性，更容易让人明白这里到底在等待什么。

初始化等待队列

函数 `wait_queue_init(wait_queue_node_t *wait_queue, struct process_control_block *pcb)` 提供了初始化 wait_queue 结点的功能。

当您初始化队列头部时，您仅需要将 wait_queue 首部的结点指针传入，第二个参数请设置为 NULL

将结点插入等待队列

您可以使用以下函数，将当前进程挂起，并插入到指定的等待队列。这些函数大体功能相同，只是在一些细节上有所不同。

函数名	解释
wait_queue_sleep_on()	将当前进程挂起，并设置挂起状态为 PROC_UNINTERRUPTIBLE
wait_queue_sleep_on_unlock()	将当前进程挂起，并设置挂起状态为 PROC_UNINTERRUPTIBLE。待当前进程被插入等待队列后，解锁给定的自旋锁
wait_queue_sleep_on_interruptible()	将当前进程挂起，并设置挂起状态为 PROC_INTERRUPTIBLE

从等待队列唤醒一个进程

您可以使用 `void wait_queue_wakeup(wait_queue_node_t * wait_queue_head, int64_t state);` 函数，从指定的等待队列中，唤醒第一个挂起时的状态与指定的 `state` 相同的进程。

当没有符合条件的进程时，将不会唤醒任何进程，仿佛无事发生。

7.1.2 二. wait_queue_head 等待队列头

数据结构定义如下：

```
typedef struct
{
    struct List wait_list;
    spinlock_t lock;  // 队列需要有一个自旋锁,虽然目前内部并没有使用,但是以后可能会用.
} wait_queue_head_t;
```

等待队列头的使用逻辑与等待队列实际是一样的，因为他同样也是等待队列的节点 (仅仅多了一把锁)。且 wait_queue_head 的函数基本上与 wait_queue 一致，只不过多了 ***_with_node_*** 的字符串。

同时，wait_queue_head.h 文件中提供了很多的宏，可以方便您的工作。

提供的宏

宏	解释
DECLARE_WAIT_ON_STACK(name, pcb)	在栈上声明一个 wait_queue 节点，同时把 pcb 所代表的进程与该节点绑定
DECLARE_WAIT_ON_STACK_SELF(name)	传在栈上声明一个 wait_queue 节点，同时当前进程 (即自身进程) 与该节点绑定
DECLARE_WAIT_ALLOC(name, pcb)	使用 `kzalloc` 声明一个 wait_queue 节点，同时把 pcb 所代表的进程与该节点绑定，请记得使用 kfree 释放空间
DECLARE_WAIT_ALLOC_SELF(name)	使用 `kzalloc` 声明一个 wait_queue 节点，同时当前进程 (即自身进程) 与该节点绑定，请记得使用 kfree 释放空间

创建等待队列头

您可以直接调用宏

```
DECLARE_WAIT_QUEUE_HEAD(m_wait_queue_head);  // 在栈上声明一个队列头变量
```

也可以手动声明

```
struct wait_queue_head_t m_wait_queue_head = {0};
wait_queue_head_init(&m_wait_queue_head);
```

将结点插入等待队列

函数名	解释
wait_queue_sleep_with_node(wait_queue_head_t *head, wait_queue_node_t *wait_node)	传入一个等待队列节点，并设置该节点的挂起状态为 PROC_UNINTERRUPTIBLE
wait_queue_sleep_with_node_unlock(wait_queue_head_t *q, wait_queue_node_t *wait, void *lock)	传入一个等待队列节点，将该节点的 pcb 指向的进程挂起，并设置挂起状态为 PROC_UNINTERRUPTIBLE。待当前进程被插入等待队列后，解锁给定的自旋锁
wait_queue_sleep_with_node_interriptible(wait_queue_head_t *q, wait_queue_node_t *wait)	传入一个等待队列节点，将该节点的 pcb 指向的进程挂起，并设置挂起状态为 PROC_INTERRUPTIBLE

从等待队列唤醒一个进程

在 wait_queue.h 中的 wait_queue_wakeup 函数直接 kfree 掉了 wait_node 节点。对于在栈上的 wait_node, 您可以选择 wait_queue_wakeup_on_stack(wait_queue_head_t *q, int64_t state) 来唤醒队列里面的队列头节点。

7.1.3 三. completion 完成量

简单用法

完成量的使用方法主要分为以下几部分：

- 声明一个完成量 (可以在栈中/使用 kmalloc/使用数组)
- 使用 wait_for_completion 等待事件完成
- 使用 complete 唤醒等待的进程

等待操作

```
void wait_fun() {
    DECLARE_COMPLETION_ON_STACK(comp);  // 声明一个completion

    // .... do somethind here
    // 大部分情况是你使用kthread_run()创建了另一个线程
    // 你需要把comp变量传给这个线程，然后当前线程就会等待他的完成

    if (!try_wait_for_completion(&comp))  // 进入等待
        wait_for_completion(&comp);
}
```

完成操作

```
void kthread_fun(struct completion *comp) {
    // ..... 做一些事  .......
    // 这里你确定你完成了目标事件

    complete(&comp);
    // 或者你使用complete_all
    complete_all(&comp);
}
```

更多用法

kernel/sched/completion.c 文件夹中, 你可以看到 __test 开头的几个函数, 他们是 completion 模块的测试代码, 基本覆盖了 completion 的大部分函数. 你可以在这里查询函数使用方法.

初始化完成量

函数 `completion_init(struct completion *x)` 提供了初始化 completion 的功能。当你使用 `DECLARE_COMPLETION_ON_STACK` 来创建 (在栈上创建) 的时候, 会自动初始化.

关于完成量的 wait 系列函数

函数名	解释
wait_for_completion(struct completion *x)	将当前进程挂起，并设置挂起状态为 PROC_UNINTERRUPTIBLE。
wait_for_completion_timeout(struct completion *x, long timeout)	将当前进程挂起，并设置挂起状态为 PROC_UNINTERRUPTIBLE。当等待 timeout 时间 (jiffies 时间片) 之后, 自动唤醒进程。
wait_for_completion_interruptible(struct completion *x)	将当前进程挂起，并设置挂起状态为 PROC_INTERRUPTIBLE。
wait_for_completion_interruptible_timeout(struct completion *x, long timeout)	将当前进程挂起，并设置挂起状态为 PROC_INTERRUPTIBLE。当等待 timeout 时间 (jiffies 时间片) 之后, 自动唤醒进程。
wait_for_multicompletion(struct completion x[], int n)	将当前进程挂起，并设置挂起状态为 PROC_UNINTERRUPTIBLE。(等待数组里面的 completion 的完成)

关于完成量的 complete 系列函数

函数名	解释
complete(struct completion *x)	表明一个事件被完成, 从等待队列中唤醒一个进程
complete_all(struct completion *x)	表明与该 completion 有关的事件被标记为永久完成, 并唤醒等待队列中的所有进程

其他用于查询信息的函数

函数名	解释
completion_done(struct completion *x)	查询 completion 的 done 变量是不是大于 0，如果大于 0，返回 true；否则返回 false。在等待前加上这个函数有可能加速？(暂未经过实验测试，有待证明)
try_wait_for_completion(struct completion *x)	查询 completion 的 done 变量是不是大于 0，如果大于 0，返回 true(同时令 done-=1)；否则返回 false。在等待前加上这个函数有可能加速？（该函数和 `completion_done` 代码逻辑基本一致，但是会主动令 completion 的 done 变量减 1）

CHAPTER 8

内存管理文档

这里讲解了内存管理模块的一些设计及实现原理。

如果你正在寻找使用内存管理模块的方法，请转到：内存管理 *API* 文档

8.1 MMIO

MMIO 是“内存映射 IO”的缩写，它被广泛应用于与硬件设备的交互之中。

8.1.1 地址空间管理

DragonOS 中实现了 MMIO 地址空间的管理机制，本节将介绍它们。

为什么需要 MMIO 地址空间自动分配？

由于计算机上的很多设备都需要 MMIO 的地址空间，而每台计算机上所连接的各种设备的对 MMIO 地址空间的需求是不一样的。如果我们为每个类型的设备都手动指定一个 MMIO 地址，会使得虚拟地址空间被大大浪费，也会增加系统的复杂性。并且，我们在将来还需要为不同的虚拟内存区域做异常处理函数。因此，我们需要一套能够自动分配 MMIO 地址空间的机制。

这套机制提供了什么功能？

- 为驱动程序分配 4K 到 1GB 的 MMIO 虚拟地址空间
- 对于这些虚拟地址空间，添加到 VMA 中进行统一管理
- 可以批量释放这些地址空间

这套机制是如何实现的？

这套机制本质上是使用了伙伴系统来对 MMIO 虚拟地址空间进行维护。在 `mm/mm.h` 中指定了 MMIO 的虚拟地址空间范围，这个范围是 `0xffffa10000000000` 开始的 1TB 的空间。也就是说，这个伙伴系统为 MMIO 维护了这 1TB 的虚拟地址空间。

地址空间分配过程

1. 初始化 MMIO-mapping 模块，在 mmio 的伙伴系统中创建 512 个 1GB 的 `__mmio_buddy_addr_region`
2. 驱动程序使用 `mmio_create` 请求分配地址空间。
3. `mmio_create` 对申请的地址空间大小按照 2 的 n 次幂进行对齐，然后从 buddy 中申请内存地址空间
4. 创建 VMA，并将 VMA 标记为 `VM_IO|VM_DONTCOPY`。MMIO 的 vma 只绑定在 `initial_mm` 下，且不会被拷贝。
5. 分配完成

一旦 MMIO 地址空间分配完成，它就像普通的 vma 一样，可以使用 mmap 系列函数进行操作。

MMIO 的映射过程

在得到了虚拟地址空间之后，当我们尝试往这块地址空间内映射内存时，我们可以调用 `mm_map` 函数，对这块区域进行映射。

该函数会对 MMIO 的 VMA 的映射做出特殊处理。即：创建 `Page` 结构体以及对应的 `anon_vma`. 然后会将对应的物理地址，填写到页表之中。

MMIO 虚拟地址空间的释放

当设备被卸载时，驱动程序可以调用 `mmio_release` 函数对指定的 mmio 地址空间进行释放。

释放的过程中，`mmio_release` 将执行以下流程：

1. 取消 mmio 区域在页表中的映射。
2. 将释放 MMIO 区域的 VMA
3. 将地址空间归还给 mmio 的伙伴系统。

CHAPTER 9

文件系统

DragonOS 的文件系统模块由 VFS（虚拟文件系统）及具体的文件系统组成。

9.1 VFS 虚拟文件系统

在 DragonOS 中，VFS 作为适配器，遮住了具体文件系统之间的差异，对外提供统一的文件操作接口抽象。

9.1.1 DragonOS 虚拟文件系统概述

简介

DragonOS 的虚拟文件系统是内核中的一层适配器，为用户程序（或者是系统程序）提供了通用的文件系统接口。同时对内核中的不同文件系统提供了统一的抽象。各种具体的文件系统可以挂载到 VFS 的框架之中。

与 VFS 相关的系统调用有 open(), read(), write(), create() 等。

dentry 对象

dentry 的全称为 directory entry，是 VFS 中对于目录项的一种抽象数据结构。当读取具体文件系统时，将会由创建 dentry 对象。dentry 对象中包含了指向 inode 的指针。

dentry 对象为真实文件系统上的目录结构建立了缓存，一旦内存中存在对应路径的 dentry 对象，我们就能直接获取其中的信息，而不需要进行费时的磁盘操作。请注意，dentry 只是为提高文件系统性能而创建的一个缓存，它并不会被写入到磁盘之中。

inode 对象

inode 的全称叫做 index node，即索引节点。一般来说，每个 dentry 都应当包含指向其 inode 的指针。inode 是 VFS 提供的对文件对象的抽象。inode 中的信息是从具体文件系统中读取而来，也可以被刷回具体的文件系统之中。并且，一个 inode 也可以被多个 dentry 所引用。

要查找某个路径下的 inode，我们需要调用父目录的 inode 的 lookup() 方法。请注意，该方法与具体文件系统有关，需要在具体文件系统之中实现。

文件描述符对象

当一个进程试图通过 VFS 打开某个文件时，我们需要为这个进程创建文件描述符对象。每个文件对象都会绑定文件的 dentry 和文件操作方法结构体，还有文件对象的私有信息。

文件描述符对象中还包含了诸如权限控制、当前访问位置信息等内容，以便 VFS 对文件进行操作。

我们对文件进行操作都会使用到文件描述符，具体来说，就是要调用文件描述符之中的 file_ops 所包含的各种方法。

注册文件系统到 VFS

如果需要注册或取消注册某个具体文件系统到 VFS 之中，则需要以下两个接口：

```
#include<filesystem/VFS/VFS.h>

uint64_t vfs_register_filesystem(struct vfs_filesystem_type_t *fs);
uint64_t vfs_unregister_filesystem(struct vfs_filesystem_type_t *fs);
```

这里需要通过 `struct vfs_filesystem_type_t` 来描述具体的文件系统。

struct vfs_filesystem_type_t

这个数据结构描述了具体文件系统的一些信息。当我们挂载具体文件系统的时候，将会调用它的read_superblock 方法，以确定要被挂载的文件系统的具体信息。

该数据结构的定义在 kernel/filesystem/VFS/VFS.h 中，结构如下：

```
struct vfs_filesystem_type_t
{
    char *name;
    int fs_flags;
    //␣
↪解析文件系统引导扇区的函数，为文件系统创建超级块结构。其中DPTE为磁盘分区表entry（MBR、GPT不同）
    struct vfs_superblock_t *(*read_superblock)(void *DPTE, uint8_t DPT_type, void␣
↪*buf, int8_t ahci_ctrl_num, int8_t ahci_port_num, int8_t part_num);
    struct vfs_filesystem_type_t *next;
};
```

name

文件系统名称字符串

fs_flags

文件系统的一些标志位。目前，DragonOS 尚未实现相关功能。

read_superblock

当新的文件系统实例将要被挂载时，将会调用此方法，以读取具体的实例的信息。

next

指向链表中下一个 struct vfs_filesystem_type_t 的指针。

超级块 (superblock) 对象

一个超级块对象代表了一个被挂载到 VFS 中的具体文件系统。

struct vfs_superblock_t

该数据结构为超级块结构体。

该数据结构定义在 kernel/filesystem/VFS/VFS.h 中，结构如下：

```
struct vfs_superblock_t
{
    struct vfs_dir_entry_t *root;
    struct vfs_super_block_operations_t *sb_ops;
    void *private_sb_info;
};
```

root

该具体文件系统的根目录的 dentry

sb_ops

该超级块对象的操作方法。

private_sb_info

超级块的私有信息。包含了具体文件系统的私有的、全局性的信息。

struct vfs_super_block_operations_t

该数据结构为超级块的操作接口。VFS 通过这些接口来操作具体的文件系统的超级块。

该数据结构定义在 `kernel/filesystem/VFS/VFS.h` 中，结构如下：

```
struct vfs_super_block_operations_t
{
    void (*write_superblock)(struct vfs_superblock_t *sb);
    void (*put_superblock)(struct vfs_superblock_t *sb);
    void (*write_inode)(struct vfs_index_node_t *inode); // 将inode信息写入磁盘
};
```

write_superblock

将 superblock 中的信息写入磁盘

put_superblock

释放超级块

write_inode

将 inode 的信息写入磁盘

索引结点 (inode) 对象

每个 inode 对象代表了具体的文件系统之中的一个对象（目录项）。

struct vfs_index_node_t

该数据结构为 inode 对象的数据结构，与文件系统中的具体的文件结点对象具有一对一映射的关系。

该数据结构定义在 `kernel/filesystem/VFS/VFS.h` 中，结构如下：

```
struct vfs_index_node_t
{
    uint64_t file_size; // 文件大小
    uint64_t blocks;    // 占用的扇区数
    uint64_t attribute;

    struct vfs_superblock_t *sb;
    struct vfs_file_operations_t *file_ops;
    struct vfs_inode_operations_t *inode_ops;

    void *private_inode_info;
};
```

file_size

文件的大小。若为文件夹，则该值为文件夹内所有文件的大小总和（估计值）。

blocks

文件占用的磁盘块数（扇区数）

attribute

inode 的属性。可选值如下：

- VFS_IF_FILE
- VFS_IF_DIR
- VFS_IF_DEVICE

sb

指向文件系统超级块的指针

file_ops

当前文件的操作接口

inode_ops

当前 inode 的操作接口

private_inode_info

与具体文件系统相关的 inode 信息。该部分由具体文件系统实现，包含该 inode 在具体文件系统之中的特定格式信息。

struct vfs_inode_operations_t

该接口为 inode 的操作方法接口，由具体文件系统实现。并与具体文件系统之中的 inode 相互绑定。

该接口定义于 `kernel/filesystem/VFS/VFS.h` 中，结构如下：

```
struct vfs_inode_operations_t
{
    long (*create)(struct vfs_index_node_t *parent_inode, struct vfs_dir_entry_t␣
↪*dest_dEntry, int mode);
    struct vfs_dir_entry_t *(*lookup)(struct vfs_index_node_t *parent_inode, struct␣
↪vfs_dir_entry_t *dest_dEntry);
    long (*mkdir)(struct vfs_index_node_t *inode, struct vfs_dir_entry_t *dEntry, int␣
↪mode);
    long (*rmdir)(struct vfs_index_node_t *inode, struct vfs_dir_entry_t *dEntry);
    long (*rename)(struct vfs_index_node_t *old_inode, struct vfs_dir_entry_t *old_
↪dEntry, struct vfs_index_node_t *new_inode, struct vfs_dir_entry_t *new_dEntry);
    long (*getAttr)(struct vfs_dir_entry_t *dEntry, uint64_t *attr);
    long (*setAttr)(struct vfs_dir_entry_t *dEntry, uint64_t *attr);
};
```

create

在父节点下，创建一个新的 inode，并绑定到 dest_dEntry 上。

该函数的应当被 `sys_open()` 系统调用在使用了 `O_CREAT` 选项打开文件时调用，从而创建一个新的文件。请注意，传递给 create() 函数的 `dest_dEntry` 参数不应包含一个 inode，也就是说，inode 对象应当被具体文件系统所创建。

lookup

当 VFS 需要在父目录中查找一个 inode 的时候，将会调用 lookup 方法。被查找的目录项的名称将会通过 dest_dEntry 传给 lookup 方法。

若 lookup 方法找到对应的目录项，将填充完善 dest_dEntry 对象。否则，返回 NULL。

mkdir

该函数被 mkdir() 系统调用所调用，用于在 inode 下创建子目录，并将子目录的 inode 绑定到 dEntry 对象之中。

rmdir

该函数被 rmdir() 系统调用所调用，用于删除给定 inode 下的子目录项。

rename

该函数被 rename 系统调用（尚未实现）所调用，用于将给定的目录项重命名。

getAttr

用来获取目录项的属性。

setAttr

用来设置目录项的属性

9.1.2 VFS API 文档

9.2 FAT32 文件系统

9.2.1 简介

FAT32 文件系统是一种相对简单的文件系统。

FAT32 文件系统实现在 `kernel/filesystem/fat32/`中。

9.2.2 相关数据结构

struct fat32_BootSector_t

fat32 启动扇区结构体

```
struct fat32_BootSector_t
{
    uint8_t BS_jmpBoot[3];     // 跳转指令
    uint8_t BS_OEMName[8];     // 生产厂商名
    uint16_t BPB_BytesPerSec; // 每扇区字节数
    uint8_t BPB_SecPerClus;    // 每簇扇区数
    uint16_t BPB_RsvdSecCnt;  // 保留扇区数
    uint8_t BPB_NumFATs;       // FAT表数量
    uint16_t BPB_RootEntCnt;  // 根目录文件数最大值
    uint16_t BPB_TotSec16;     // 16位扇区总数
    uint8_t BPB_Media;         // 介质描述符
    uint16_t BPB_FATSz16;      // FAT12/16每FAT扇区数
    uint16_t BPB_SecPerTrk;    // 每磁道扇区数
    uint16_t BPB_NumHeads;     // 磁头数
    uint32_t BPB_HiddSec;      // 隐藏扇区数
    uint32_t BPB_TotSec32;     // 32位扇区总数
```

(续下页)

(接上页)

```
    uint32_t BPB_FATSz32;    // FAT32每FAT扇区数
    uint16_t BPB_ExtFlags;   // 扩展标志
    uint16_t BPB_FSVer;      // 文件系统版本号
    uint32_t BPB_RootClus;   // 根目录起始簇号
    uint16_t BPB_FSInfo;     // FS info结构体的扇区号
    uint16_t BPB_BkBootSec; // 引导扇区的备份扇区号
    uint8_t BPB_Reserved0[12];

    uint8_t BS_DrvNum; // int0x13的驱动器号
    uint8_t BS_Reserved1;
    uint8_t BS_BootSig;        // 扩展引导标记
    uint32_t BS_VolID;         // 卷序列号
    uint8_t BS_VolLab[11];     // 卷标
    uint8_t BS_FilSysType[8]; // 文件系统类型

    uint8_t BootCode[420]; // 引导代码、数据

    uint16_t BS_TrailSig; // 结束标志0xAA55
} __attribute__((packed));
```

struct fat32_FSInfo_t

该扇区存储了 FAT32 文件系统的一些参考信息。

```
struct fat32_FSInfo_t
{
    uint32_t FSI_LeadSig;
    uint8_t FSI_Reserved1[480];
    uint32_t FSI_StrucSig;
    uint32_t FSI_Free_Count;
    uint32_t FSI_Nxt_Free;
    uint8_t FSI_Reserved2[12];
    uint32_t FSI_TrailSig;
} __attribute__((packed));
```

FSI_LeadSig

FS info 扇区标志符数值为 0x41615252

FSI_Reserved1

保留使用，全部置为 0

FSI_StrucSig

FS_Info 扇区的另一个标志符，数值为 0x61417272

FSI_Free_Count

上一次记录的空闲簇数量，这是一个参考值

FSI_Nxt_Free

空闲簇的起始搜索位置，这是为驱动程序提供的参考值.

FSI_Reserved2 保留使用，全部置为 0

FSI_TrailSig

FS_Info 扇区结束标志，数值为 0xaa550000

struct fat32_Directory_t

短目录项结构体。

```
struct fat32_Directory_t
{
    unsigned char DIR_Name[11];
    unsigned char DIR_Attr;
    unsigned char DIR_NTRes;
    unsigned char DIR_CrtTimeTenth;
    unsigned short DIR_CrtTime;
    unsigned short DIR_CrtDate;
    unsigned short DIR_LastAccDate;
    unsigned short DIR_FstClusHI;
    unsigned short DIR_WrtTime;
    unsigned short DIR_WrtDate;
    unsigned short DIR_FstClusLO;
    unsigned int DIR_FileSize;
} __attribute__((packed));
```

DIR_Name

目录项名称。前 8bytes 为基础名，后 3bytes 为扩展名

DIRAttr

目录项属性。可选值有如下：

- ATTR_READ_ONLY
- ATTR_HIDDEN
- ATTR_SYSTEM
- ATTR_VOLUME_ID

- ATTR_DIRECTORY
- ATTR_ARCHIVE
- ATTR_LONG_NAME

DIR_NTRes

该项为 Windows 下特有的表示区域，通过该项的值，表示基础名和扩展名的大小写情况。该项的值为 EXT|BASE 组合而成，其中，具有以下定义：

BASE:LowerCase(8),UpperCase(0) EXT:LowerCase(16),UpperCase(0)

DIR_CrtTimeTenth

文件创建的毫秒级时间戳

DIR_CrtTime

文件创建时间

DIR_CrtDate

文件创建日期

DIR_LastAccDate

文件的最后访问日期

DIR_FstClusHI

文件起始簇号（高 16bit）

DIR_WrtTime

最后写入时间

DIR_WrtDate

最后写入日期

DIR_FstClusLO

文件起始簇号（低 16bit）

DIR_FileSize 文件大小

struct fat32_partition_info_t

该数据结构为 FAT32 分区的信息结构体，并不实际存在于物理磁盘上。这个结构体在挂载文件系统时被创建，作为文件系统的超级块的私有信息的一部分。

struct fat32_inode_info_t

该结构体是 VFS 的 inode 结构体的私有信息部分的具体实现。

9.2.3 已知问题

1. 对目录项名称的检查没有按照标准严格实现
2. 当磁盘可用簇数量发生改变时，未更新 FS_Info 扇区
3. 未填写目录项的时间字段

9.2.4 TODO

- 完全实现 VFS 定义的文件接口
- 性能优化

9.2.5 参考资料

FAT32 File System Specification - from Microsoft

9.3 rootFS 根文件系统

rootFS 是 DragonOS 开启后挂载的第一个文件系统, 它是一个基于内存的伪文件系统。rootfs 的功能主要是在具体的磁盘文件系统被挂载之前，为其他的伪文件系统提供挂载点，使得系统能被正确的初始化。

rootfs 的初始化将与 VFS 一同初始化。rootfs 将为系统的各项文件系统的挂载创建 dentry，使得其他的文件系统如 `devfs` 等，能在磁盘文件系统被挂载之前被正确的初始化。

当磁盘根文件系统被挂载后，将调用 `rootfs_umount()` 函数。该函数将会把原本挂载在 rootfs 上的各种伪文件系统迁移到磁盘根文件系统上。当迁移完成后，将会释放 rootfs 所占用的资源。

9.4 devFS 设备文件系统

devfs 是一种基于内存的伪文件系统，设备可注册到 devfs 中。对上层模块及应用程序而言，每个设备都是可操作的文件。

9.4.1 原理

由于每个设备被抽象为文件，因此对于驱动程序而言，只需实现文件的操作接口，上层的应用程序以及其他系统组件即可操作文件操作接口来控制硬件设备。

9.4.2 目录结构

按照设备的主类型的不同，将多种设备放置在 devfs 的不同文件夹下. 请注意，同一设备可以出现在不同的文件夹下。

- `char` 字符设备
- `block` 块设备
- `usb` usb 设备
- stdio 等设备放置在 devfs 的根目录下

9.4.3 设备注册

驱动程序可使用 `devfs_register_device()` 函数将设备注册到 devfs 之中。

9.4.4 设备卸载

【尚未实现】

CHAPTER 10

内核调试模块

这里是 DragonOS 的内核调试模块文档。

10.1 内核栈 traceback

10.1.1 简介

内核栈 traceback 的功能位于 `kernel/debug/traceback/`文件夹中。为内核态提供 traceback 的功能，打印调用栈到屏幕上。

10.1.2 API

`void traceback(struct pt_regs * regs)`

作用

该接口定义于 `kernel/debug/traceback/traceback.h` 中，将会对给定内核栈进行 traceback，并打印跟踪结果到屏幕上。

参数

regs

要开始追踪的第一层内核栈栈帧（也就是栈的底端）

10.1.3 实现原理

当内核第一次链接之后，将会通过 Makefile 中的命令，运行 `kernel/debug/kallsyms` 程序，提取内核文件的符号表，然后生成 `kernel/debug/kallsyms.S`。该文件的 rodata 段中存储了 text 段的函数的符号表。接着，该文件将被编译为 `kallsyms.o`。最后，Makefile 中再次调用 `ld` 命令进行链接，将 kallsyms.o 链接至内核文件。

当调用 `traceback` 函数时，其将遍历该符号表，找到对应的符号并输出。

10.1.4 未来发展方向

- 增加写入到日志文件的功能

CHAPTER 11

内核测试

本章节将介绍如何测试内核，包括手动测试以及自动测试。

我们需要尽可能的对内核进行完善的测试，以便我们能够更好的保证内核的稳定性，且减少其他模块的 debug 难度。

设置完善的测试用例能帮助我们尽可能的检测到问题，防止我们在写新的模块的时候，被已有的模块的一些藏得很深的 bug “背刺一刀”。

由于您难以借助 GDB 等工具进行调试，因此在内核中进行手动测试比应用程序测试要困难一些。

对于一些模块，我们可以使用编写代码进行单元测试，并输出异常信息。遗憾的是，并非所有模块都可以进行单元测试。比如我们常见的内存管理、进程管理等模块都不能进行单元测试。

11.1 内核测试框架

DragonOS 提供了一个测试框架，旨在对内核的一些模块进行自动化测试。内核测试框架位于 `ktest/` 下。

我们可以使用这个测试框架，按照规范编写测试代码，然后在合适的地方使用 `ktest_start()` 创建一个全新的内核线程并发起测试。

11.1.1 使用方法

创建自动测试程序

假如您要对 kfifo 模块进行自动测试，您可以在 ktest/下，创建一个名为 test-kfifo.c 的测试文件，并编写 Makefile。

在 test-kfifo.c 中，包含 ktest_utils.h 和 ktest.h 这两个头文件。

您需要像下面这样，在 test-kfifo.c 中，创建一个测试用例函数表，并把测试用例函数填写到其中：

```
static ktest_case_table kt_kfifo_func_table[] = {
    ktest_kfifo_case0_1,
};
```

然后创建一个函数，作为 kfifo 测试的主函数。请注意，您需要将它的声明添加到 ktest.h 中。

```
uint64_t ktest_test_kfifo(uint64_t arg)
{
    kTEST("Testing kfifo...");
    for (int i = 0; i < sizeof(kt_kfifo_func_table) / sizeof(ktest_case_table); ++i)
    {
        kTEST("Testing case %d", i);
        kt_kfifo_func_table[i](i, 0);
    }
    kTEST("kfifo Test done.");
    return 0;
}
```

编写测试用例

您可以创建一个或多个测试用例，命名为：ktest_kfifo_case_xxxxx. 在这个例子中，我创建了一个测试用例，命名为：ktest_kfifo_case0_1. 如下所示：

```
static long ktest_kfifo_case0_1(uint64_t arg0, uint64_t arg1)
```

这里最多允许我们传递两个参数到测试函数里面。

那么，我们该如何编写测试用例呢?

我们主要是需要设置一些情节，以便能测试到目标组件的每个情况。为了检验模块的行为是否符合预期，我们需要使用 assert(condition) 宏函数，对目标 condition 进行校验。若 condition 为 1，则表明测试通过。否则，将会输出一行 assert failed 信息到屏幕上。

发起测试

我们可以在 pid≥1 的内核线程中发起测试。由于 DragonOS 目前尚不完善，您可以在 `process/process.c` 中的 `initial_kernel_thread()` 函数内，发起内核自动测试。具体的代码如下：

```
ktest_start(ktest_test_kfifo, 0);
```

这样就发起了一个内核测试，它会创建一个新的内核线程进行自动测试，您不必担心第一个内核线程会被阻塞。

11.1.2 API 文档

ktest_start

```
pid_t ktest_start(uint64_t (*func)(uint64_t arg), uint64_t arg)
```

描述

开启一个新的内核线程以进行测试

参数

func

测试函数. 新的测试线程将会执行该函数，以进行测试。

arg

传递给测试函数的参数

返回值

测试线程的 pid

assert

```
#define assert(condition)
```

描述

判定 condition 是否为 1，若不为 1，则输出一行错误日志信息：

```
[ kTEST FAILED ] Ktest Assertion Failed, file:%s, Line:%d
```

kTEST

```
#define kTEST(...)
```

描述

格式化输出一行以 `[ kTEST ] file:%s, Line:%d` 开头的日志信息。

ktest_case_table

```
typedef long (*ktest_case_table)(uint64_t arg0, uint64_t arg1)
```

描述

ktest 用例函数的类型定义。

CHAPTER 12

处理器架构

该部分文档提供了和处理器架构相关的一些编程实现细节的描述。

12.1 x86-64 相关文档

12.1.1 USB Legacy 支持

简介

usb legacy support 指的是，由 BIOS 提供的，对 USB 鼠标、USB 键盘的支持。在支持并启用 USB Legacy Support 的计算机上，USB 鼠标、键盘由 BIOS 提供模拟，在操作系统看来，就像接入了 PS/2 鼠标、键盘一样。

相关

- 在初始化 USB 控制器时，需要关闭它的 USB Legacy Support

CHAPTER 13

LibC 文档

13.1 简介

LibC 是连接用户程序和操作系统的纽带，LibC 为应用程序提供了一系列标准库函数。应用程序可以通过 DragonOS 的 LibC，快速地与操作系统进行交互。DragonOS 的 LibC 主要依照 POSIX 2008 规范实现，与 Linux 下的 glibC 具有相似之处。

13.2 API 文档

13.2.1 ctype.h

函数列表（这里只列出已实现的函数）：

```
``int isprint(int c)`` : 传入一个字符，判断是否可以被输出

``int islower(int c)`` : 传入一个字符，判断是否是小写字母

``int isupper(int c)`` : 传入一个字符，判断是否是大写字母

``int isalpha(int c)`` : 传入一个字符，判断是否是字母

``int isdigit(int c)`` : 传入一个字符，判断是否是数字
```

(续下页)

(接上页)

```
``int toupper(int c)`` : 传入一个小写字母字符，返回这个字母的大写形式

``int tolower(int c)`` : 传入一个大写字母字符，返回这个字母的小写形式

``int isspace(int c)``  : 传入一个字符，判断是否是空白字符
```

宏定义:

```
### 暂无用处

``#define _U 01``

``#define _L 02``

``#define _N 04``

``#define _S 010``

``#define _P 020``

``#define _C 040``

``#define _X 0100``

``#define _B 0200``
```

13.2.2 dirent.h

简介

```
与文件夹有关的头文件。
```

结构体列表:

```
``struct DIR`` :

    变量列表:

    ``int fd`` : 文件夹id（不推荐修改）

    ``int buf_pos`` : 文件夹缓冲区指针的位置

    ``int buf_len`` : 文件夹缓冲区的大小（默认为256）

``struct dirent`` :

    变量列表:

    ``ino_t(see libc/sys/types.h) ino`` : 文件序列号（不推荐修改）

    ``off_t d_off`` : dir偏移量（不推荐修改）

    ``unsigned short d_reclen`` : 文件夹中的记录数

    ``unsigned char d_type`` : 目标的类型(有可能是文件，文件夹，磁盘)

    ``char d_name[]`` : 目标的名字
```

函数列表（这里只列出已实现的函数）:

```
``DIR opendir(const char *path)``

    传入文件夹的路径，返回文件夹结构体

``int closedir(DIR *dirp)``

    传入文件夹结构体，关闭文件夹，释放内存

    若失败，返回-1

``dirent readdir(DIR *dir)``

    传入文件夹结构体，读入文件夹里的内容，并打包为dirent结构体返回
```

宏定义:

文件夹类型:

``#define VFS_IF_FILE (1UL << 0)``

``#define VFS_IF_DIR (1UL << 1)``

``#define VFS_IF_DEVICE (1UL << 2)``

缓冲区长度的默认值

``#define DIR_BUF_SIZE 256``

13.2.3 errno.h

简介:

共享错误号码

属性:

``extern int errno`` : 通用错误代码

宏定义（复制自代码，了解即可）:

```
#define E2BIG 1            /* 参数列表过长，或者在输出buffer中缺少空间␣
↪或者参数比系统内建的最大值要大 Argument list too long.*/

#define EACCES 2           /* 访问被拒绝 Permission denied*/

#define EADDRINUSE 3       /* 地址正在被使用 Address in use.*/

#define EADDRNOTAVAIL 4    /* 地址不可用 Address  not available.*/

#define EAFNOSUPPORT 5     /* 地址family不支持 Address family not supported.*/

#define EAGAIN 6           /* 资源不可用，请重试。 Resource unavailable, try again␣
↪(may be the same value as [EWOULDBLOCK]).*/
```

(续下页)

(接上页)

```
#define EALREADY 7          /* 连接已经在处理 Connection already in progress.*/

#define EBADF 8             /* 错误的文件描述符 Bad file descriptor.*/

#define EBADMSG 9           /* 错误的消息 Bad message.*/


#define EBUSY 10            /* 设备或资源忙 Device or resource busy.*/

#define ECANCELED 11        /* 操作被取消 Operation canceled.*/

#define ECHILD 12           /* 没有子进程 No child processes.*/

#define ECONNABORTED 13     /* 连接已断开 Connection aborted.*/

#define ECONNREFUSED 14     /* 连接被拒绝 Connection refused.*/

#define ECONNRESET 15       /* 连接被重置 Connection reset.*/

#define EDEADLK 16          /* 资源死锁将要发生 Resource deadlock would occur.*/

#define EDESTADDRREQ 17     /* 需要目标地址 Destination address required.*/

#define EDOM 18             /* 数学参数超出作用域 Mathematics argument out of domain␣
↪of function.*/

#define EDQUOT 19           /* 保留使用 Reserved*/



#define EEXIST 20           /* 文件已存在 File exists.*/

#define EFAULT 21           /* 错误的地址 Bad address*/

#define EFBIG 22            /* 文件太大 File too large.*/

#define EHOSTUNREACH 23     /* 主机不可达 Host is unreachable.*/

#define EIDRM 24            /* 标志符被移除 Identifier removed.*/

#define EILSEQ 25           /* 不合法的字符序列 Illegal byte sequence.*/

#define EINPROGRESS 26      /* 操作正在处理 Operation in progress.*/
```

(续下页)

(接上页)

```
#define EINTR 27            /* 被中断的函数 Interrupted function.*/

#define EINVAL 28           /* 不可用的参数 Invalid argument.*/

#define EIO 29              /* I/O错误 I/O error.*/



#define EISCONN 30          /* 套接字已连接 Socket is connected.*/

#define EISDIR 31           /* 是一个目录 Is a directory*/

#define ELOOP 32            /* 符号链接级别过多 Too many levels of symbolic links.*/

#define EMFILE 33           /* 文件描述符的值过大 File descriptor value too large.*/

#define EMLINK 34           /* 链接数过多 Too many links.*/

#define EMSGSIZE 35         /* 消息过大 Message too large.*/

#define EMULTIHOP 36        /* 保留使用 Reserved.*/

#define ENAMETOOLONG 37     /* 文件名过长 Filename too long.*/

#define ENETDOWN 38         /* 网络已关闭 Network is down.*/

#define ENETRESET 39        /* 网络连接已断开 Connection aborted by network.*/

#define ENETUNREACH 40      /* 网络不可达 Network unreachable.*/

#define ENFILE 41           /* 系统中打开的文件过多 Too many files open in system.*/

#define ENOBUFS 42          /* 缓冲区空间不足 No buffer space available.*/

#define ENODATA 43          /* 队列头没有可读取的消息 No message is available on the␣
↪STREAM head read queue.*/

#define ENODEV 44           /* 没有指定的设备 No such device.*/

#define ENOENT 45           /* 没有指定的文件或目录 No such file or directory.*/

#define ENOEXEC 46          /* 可执行文件格式错误 Executable file format error.*/
```

(续下页)

(接上页)

```
#define ENOLCK 47           /* 没有可用的锁 No locks available.*/

#define ENOLINK 48          /* 保留 Reserved.*/


#define ENOMEM 49           /* 没有足够的空间 Not enough space.*/


#define ENOMSG 50           /* 没有期待类型的消息 No message of the desired type.*/

#define ENOPROTOOPT 51      /* 协议不可用 Protocol not available.*/

#define ENOSPC 52           /* 设备上没有空间 No space left on device.*/

#define ENOSR 53            /* 没有STREAM资源  No STREAM resources.*/

#define ENOSTR 54           /* 不是STREAM Not a STREAM*/

#define ENOSYS 55           /* 功能不支持 Function not supported.*/

#define ENOTCONN 56         /* 套接字未连接 The socket is not connected.*/

#define ENOTDIR 57          /* 不是目录 Not a directory.*/

#define ENOTEMPTY 58        /* 目录非空 Directory not empty.*/

#define ENOTRECOVERABLE 59 /* 状态不可覆盖 State not recoverable.*/



#define ENOTSOCK 60         /* 不是一个套接字 Not a socket.*/

#define ENOTSUP 61          /* 不被支持 Not supported (may be the same value as␣
↪[EOPNOTSUPP]).*/

#define ENOTTY 62           /* 不正确的I/O控制操作 Inappropriate I/O control operation.
↪*/

#define ENXIO 63            /* 没有这样的设备或地址 No such device or address.*/

#define EOPNOTSUPP 64       /* 套接字不支持该操作 Operation not supported on socket␣
↪(may be the same value as [ENOTSUP]).*/
```

(续下页)

(接上页)

```
#define EOVERFLOW 65        /* 数值过大，产生溢出 Value too large to be stored in data␣
↪type.*/

#define EOWNERDEAD 66       /* 之前的拥有者挂了 Previous owner died.*/

#define EPERM 67            /* 操作不被允许 Operation not permitted.*/

#define EPIPE 68            /* 断开的管道 Broken pipe.*/

#define EPROTO 69           /* 协议错误 Protocol error.*/



#define EPROTONOSUPPORT 70 /* 协议不被支持 Protocol not supported.*/

#define EPROTOTYPE 71       /* 对于套接字而言，错误的协议 Protocol wrong type for␣
↪socket.*/

#define ERANGE 72           /* 结果过大 Result too large.*/

#define EROFS 73            /* 只读的文件系统 Read-only file system.*/

#define ESPIPE 74           /* 错误的寻道 Invalid seek.*/

#define ESRCH 75            /* 没有这样的进程 No such process.*/

#define ESTALE 76           /* 保留 Reserved.*/

#define ETIME 77            /* 流式ioctl()超时 Stream ioctl() timeout*/

#define ETIMEDOUT 78        /* 连接超时 Connection timed out.*/

#define ETXTBSY 79          /* 文本文件忙 Text file busy.*/



#define EWOULDBLOCK 80      /* 操作将被禁止 Operation would block (may be the same␣
↪value as [EAGAIN]).*/

#define EXDEV 81            /* 跨设备连接 Cross-device link.*/
```

13.2.4 fcntl.h

简介

文件操作

函数列表：

``int open(const char * path,int options, ...)``

传入文件路径，和文件类型（详细请看下面的宏定义），将文件打开并返回文件id。

宏定义（粘贴自代码，了解即可）：

```
#define O_RDONLY 00000000 // Open Read-only

#define O_WRONLY 00000001 // Open Write-only

#define O_RDWR 00000002   // Open read/write

#define O_ACCMODE 00000003 // Mask for file access modes


#define O_CREAT 00000100 // Create file if it does not exist

#define O_EXCL 00000200 // Fail if file already exists

#define O_NOCTTY 00000400 // Do not assign controlling terminal

#define O_TRUNC 00001000 // 文件存在且是普通文件，并以O_RDWR或O_
↪WRONLY打开，则它会被清空

#define O_APPEND 00002000   // 文件指针会被移动到文件末尾

#define O_NONBLOCK 00004000 // 非阻塞式IO模式


#define O_EXEC 00010000 // 以仅执行的方式打开（非目录文件）

#define O_SEARCH 00020000   // Open the directory for search only

```

(续下页)

(接上页)

```
#define O_DIRECTORY 00040000 // 打开的必须是一个目录

#define O_NOFOLLOW 00100000 // Do not follow symbolic links
```

13.2.5 math.h

简介:

```
数学库
```

函数列表:

```
``double fabs(double x)`` : 返回 x 的绝对值

``float fabsf(float x)`` : 返回 x 的绝对值

``long double fabsl(long double x)``: 返回 x 的绝对值

``double round(double x)`` 四舍五入 x

``float roundf(float x)`` 四舍五入 x

``long double roundl(long double x)`` 四舍五入 x

``int64_t pow(int64_t x,int y)`` 返回 x 的 y 次方
```

13.2.6 stdio.h

简介:

```
向标准输入输出里操作
```

函数列表:

```
``int64_t put_string(char *str, uint64_t front_color, uint64_t bg_color)``

    输出字符串（带有前景色，背景色）

``int printf(const char *fmt, ...)``

    就是正常的 ``printf``

``int sprintf(char *buf,const char *fmt,...)```

    就是正常的 ``sprintf``

``int vsprintf(char *buf,const char *fmt,va_list args)``

    格式化，不建议调用，请用 printf 或 sprintf 替代。
```

宏定义

```
### 字体颜色的宏定义

    ``#define COLOR_WHITE 0x00ffffff  //白``

    ``#define COLOR_BLACK 0x00000000  //黑``

    ``#define COLOR_RED 0x00ff0000    //红``

    ``#define COLOR_ORANGE 0x00ff8000 //橙``

    ``#define COLOR_YELLOW 0x00ffff00 //黄``

    ``#define COLOR_GREEN 0x0000ff00  //绿``

    ``#define COLOR_BLUE 0x000000ff   //蓝``

    ``#define COLOR_INDIGO 0x0000ffff //靛``

    ``#define COLOR_PURPLE 0x008000ff //紫``
### 无需使用

    ``#define SEEK_SET 0 /* Seek relative to start-of-file */``
```

(续下页)

(接上页)

```
    ``#define SEEK_CUR 1 /* Seek relative to current position */``

    ``#define SEEK_END 2 /* Seek relative to end-of-file */``

    ``#define SEEK_MAX 3``
```

13.2.7 printf.h

不建议引用，需要 printf 函数请引用 stdio.h

13.2.8 stddef.h

简介:

```
定义了关于指针的常用类型
```

定义:

```
``typedef __PTDIFF_TYPE__ ptrdiff_t``  : 两个指针相减的结果类型

``NULL ((void *) 0)`` : 空指针
```

13.2.9 stdlib.h

简介:

```
一些常用函数
```

函数列表:

```
``void *malloc(ssize_t size)`` :  普通的 ``malloc``

``void free(void *ptr)`` : 释放内存

``int abs(int x)`` : x 的绝对值

``long labs(long x)`` : x 的绝对值
```

(续下页)

(接上页)

```
``long long llabs(long long x)`` : x 的绝对值

``int atoi(const char *str)`` 字符串转数字

``void exit(int status)`` : 普通的 ``exit``
```

13.2.10 string.h

简介:

```
字符串操作
```

函数列表:

```
``size_t strlen(const char *s)`` : 返回字符串长度

``int strcmp(const char *a,const char *b)`` 比较字符串的字典序

``char* strncpy(char *dst,const char *src,size_t count)``

    拷贝制定字节数的字符串

    dst: 目标地址

    src:  原字符串

    count: 字节数

``char* strcpy(char *dst,const char *src)`` : 复制整个字符串

``char* strcat(char *dest,const char* src)`` : 拼接两个字符串
```

13.2.11 time.h

简介:

```
时间相关
```

时刻以纳秒为单位

结构体:

```
``struct timespec`` : 时间戳

    ### 变量列表:

        ``long int tv_sec`` : 秒

        ``long int tv_nsec`` : 纳秒
```

宏定义:

```
``#define CLOCKS_PER_SEC 1000000`` 每一秒有1000000个时刻（纳秒）
```

函数列表:

```
``int nanosleep(const struct timespec *rdtp,struct timespec *rmtp)``

    休眠指定时间

    rdtp ： 指定休眠的时间

    rmtp ： 返回剩余时间

``clock_t clock()`` ： 获得当前系统时间
```

13.2.12 unistd.h

简介:

```
也是一些常用函数
```

函数列表:

```
``int close(int fd)`` : 关闭文件

``ssize_t read(int fd,void *buf,size_t count)`` : 从文件读取

    传入文件id, 缓冲区, 以及字节数

    返回成功读取的字节数

``ssize_t write(int fd,void const *buf,size_t count)`` : 写入文件

    传入文件id, 缓冲区, 字节数

    返回成功写入的字节数

``off_t lseek(int fd,off_t offset,int whence)`` : 调整文件访问位置

    传入文件id, 偏移量, 调整模式

    返回结束后的文件访问位置

``pid_t fork(void)`` : fork 当前进程

``pid_t vfork(void)`` : fork 当前进程, 与父进程共享 VM,flags,fd

``uint64_t brk(uint64_t end_brk)`` : 将堆内存调整为end_brk

    若end_brk 为 -1, 返回堆区域的起始地址

    若end_brk 为 -2, 返回堆区域的结束地址

    否则调整堆区的结束地址域, 并返回错误码

``void *sbrk(int64_t increment)`` :

    将堆内存空间加上offset (注意, 该系统调用只应在普通进程中调用, 而不能是内核线程)
```

(续下页)

(接上页)

```
    increment ： 偏移量

``int64_t chdir(char *dest_path)``

    切换工作目录（传入目录路径）

``int execv(const char* path,char * const argv[])`` ：执行文件
    path ： 路径
    argv ： 执行参数列表

``extern int usleep(useconds_t usec)`` ： 睡眠usec微秒
```

这里是所有 libc 头文件的集合, 在代码里可以这样引用: `#include<libc/xxx.h>`

13.3 设计文档

[内容待完善]

CHAPTER 14

系统调用 API

14.1 简介

CHAPTER 15

参与开发

DragonOS 社区欢迎您的加入！学习技术本身固然重要，但是以下这些文档将会帮助您了解 DragonOS 社区需要什么。

阅读这些文档将会帮助您参与到开发之中，并且能让您的代码能更快合并到主线。

15.1 代码风格

这份文档将会简要的介绍 DragonOS 的代码风格。每个人的代码风格都各不相同，这是一件非常正常的事情。但是，对于一个开源项目的可维护性而言，我们希望制定一些代码规范，以便包括您在内的每个开发者都能在看代码的时候更加舒服。一个充斥着各种不同代码风格的项目，是难以维护的。

我们在这里提出一些建议，希望您能够尽量遵循这些建议。这些建议与 Linux 的代码规范相似，但又略有不同。

15.1.1 0. 代码格式化工具

在提出下面的建议之前，我们建议您在开发的时候使用 Visual Studio Code 的 `C/C++ Extension Pack` 插件作为代码格式化工具。这些插件能为您提供较好自动格式化功能，使得您的代码的基本格式符合 DragonOS 的要求。

当您在编码时，经常性的按下 `Ctrl+shift+I` 或您设置的代码格式化快捷键，能帮助您始终保持良好的代码格式。

15.1.2 1. 缩进

一个制表符的宽度等于 4 个空格。代码的缩进是按照制表符宽度 (在多数编辑器上为 4 个字符) 进行缩进的。

这样能够使得您的代码变得更加容易阅读，也能更好的看出代码的控制结构。这样能避免很多不必要的麻烦！

举个例子：在 switch 语句中，将 switch 和 case 放置在同一缩进级别。并且将每个 case 的代码往右推进一个 tab。这样能让代码可读性变得更好。

```
switch (cmd)
{
case AHCI_CMD_READ_DMA_EXT:
    pack->blk_pak.end_handler = NULL;
    pack->blk_pak.cmd = AHCI_CMD_READ_DMA_EXT;
    break;
case AHCI_CMD_WRITE_DMA_EXT:
    pack->blk_pak.end_handler = NULL;
    pack->blk_pak.cmd = AHCI_CMD_WRITE_DMA_EXT;
    break;
default:
    pack->blk_pak.end_handler = NULL;
    pack->blk_pak.cmd = cmd;
    break;
}
```

15.1.3 2. 分行

我们建议，每行不要超过 120 个字符。如果超过了，除非有必要的理由，否则应当将其分为两行。

在分行时，我们需要从被分出来的第二行开始，比第一行的起始部分向右进行一个缩进，以表明这是一个子行。使用代码格式化的快捷键能让你快速完成这件事。

对于一些日志字符串而言，为了能方便的检索到他们，我们不建议对其进行分行。

对于代码的分行，请不要试图通过以下的方式将几个语句放置在同一行中，这样对于代码可读性没有任何好处：

```
// 错误示范(1)
if(a) return 1;

// 错误示范(2)
if(b)
    do_a(),do_b();
```

15.1.4 3. 大括号和空格

3.1 大括号

大括号的放置位置的选择是因人而异的，主要是习惯原因，而不是技术原因。我们推荐将开始括号和结束括号都放置在一个新的行首。如下所示：

```
while(i<10)
{
    ++i;
}
```

这种规定适用于所有的代码块。

这么选择的原因是，在一些编辑器上，这样放置括号，**编辑器上将会出现辅助的半透明竖线，且竖线两端均为括号**。这样能帮助开发者更好的定位代码块的层次关系。

下面通过一些例子来演示：

在下面这个代码块中，我们需要注意的是，`else if` 语句需要另起一行，而不是跟在上一个`}`后方。这是因为我们规定 `{`必须在每行的起始位置，并且还要保持缩进级别的缘故。

```
if (*fmt == '*')
{
    ++fmt;
}
else if (is_digit(*fmt))
{
    field_width = skip_and_atoi(&fmt);
}
```

当循环中有多个简单的语句的时候，需要使用大括号。

```
while (condition)
{
    if (test)
        do_something();
}
```

当语句只有 1 个简单的子句时，我们不必使用大括号。

```
if(a)
    return 1;
```

3.2 空格

对于大部分关键字，我们需要在其后添加空格，以提高代码的可读性。

请您在所有这些关键字后面输入一个空格：

```
if, switch, case, for, do, while
```

关键字 sizeof、typeof、alignof、__atrribute__ 的后面则不需要添加空格，因为使用他们的时候，就像是使用函数一样。

对于指针类型的变量，* 号要贴近变量名而不是贴近类型名。如下所示：

```
char *a;
void *func(char* s, int **p);
```

在大多数二元和三元运算符周围（在每一侧）使用一个空格，如下所示：

```
=  +  -  <  >  *  /  %  |  &  ^  <=  >=  ==  !=  ?  :
```

这些一元运算符后方没有空格

```
&  *  +  -  ~  !  sizeof  typeof  alignof  __attribute__  defined
```

特殊的例子，以下运算符的前后都不需要空格：

```
++  --  .  ->
```

15.1.5 4. 命名

DragonOS 中的命名规范不使用诸如 `TempValue` 这样的驼峰命名法的函数名，而是使用 `tmp` 这样言简意赅的命名。

注意，这里指的是我们在整个项目内都不希望使用驼峰命名法。并不意味着程序员可以随便的使用一些难以理解的缩写来作为变量名。

对于全局变量或者全局可见的函数、结构体而言，我们需要遵循以下的命名规定：

- 名称需要易于理解，且不具有歧义。如：对于一个计算文件夹大小的函数而言，我们建议使用 `count_folder_size()` 来命名，而不是 `cntfs()` 这样令其他人头大的命名。
- 全局的，非 static 的名称，除非有特别的必要，命名时需要遵循以下格式：*模块名缩写前缀* `_` *函数/变量名*。这样的命名能便于别人区分这个名称位于哪个模块内，也减少了由于命名冲突所导致的麻烦。
- 不需要让其他代码文件可见的全局名称，必须添加 `static` 修饰符。

对于函数内的局部变量而言，命名规范则是需要言简意赅。过长的名称在局部变量中没有太大的意义。

【文档未完成，待继续完善】

CHAPTER 16

与社区建立联系

社区公共邮箱：contact@DragonOS.org

DragonOS 负责人: longjin

工作邮箱: longjin@RinGoTek.cn

开发交流 QQ 群：115763565

DragonOS 官网：https://DragonOS.org

CHAPTER 17

发行日志

这里是 DragonOS 的发行日志，会记录 DragonOS 的每一个版本的更新内容。

17.1 V0.1.0

备注： 本文作者：龙进 longjin@RinGoTek.cn

2022 年 11 月 6 日

17.1.1 前言

DragonOS 从 2022 年 1 月 15 日开始开发，到如今已经经历了将近 300 天。在这么多个日夜里，已经数不清到底花了多少时间在 DragonOS 的开发之中，我基本上把所有的空闲时间都给了 DragonOS，保守估计总工时已经在 1000 小时以上。能够发布第一个版本，我感到十分有成就感。

在 2022 年 7 月以来，陆陆续续的，有来自 6 所高校或企业的小伙伴/大佬加入了 DragonOS 的开发。我当时非常的欣喜，我想，也许在大家的一同努力下，我们能创造出一个真正具有实用性的操作系统呢！我们累计召开了 14 次交流讨论会。我相信，在大家的共同努力下，将来，我们一定能创造出真正独立自主的、开放的、面向服务器领域应用的开源操作系统，并在生产环境中得到应用。

尽管 DragonOS 目前只是一个玩具水平的操作系统，只是“比本科生毕业设计难度略高的”操作系统。但是，请不要小看它，它的内在的架构设计，瞄准了 Linux5.18 及以后的发行版，虽尚未能达到 Linux 的水

平，但我们正努力追赶。得益于 Linux 的相关资料，DragonOS 在架构设计之时，学习了 Linux 的很多设计思想，相关组件都尽量考虑了可扩展性与可移植性。

千里之行，始于足下。DragonOS V0.1.0 版本的发布，是一个全新的开始。**希望在未来的十年里，我们能与众多伙伴们一同努力，在 2032 年，将 DragonOS 建设成为具有实用意义的，能够在服务器领域取得广泛应用的开源操作系统！**

百舸争流，奋楫者先；中流击水，勇进者胜。我相信，在接下来的时间里，在社区开发者们的不断努力下，我们的目标，终将成为现实！

17.1.2 特别鸣谢

在 DragonOS V0.1.0 版本的发布之际，我想对我的老师、前辈以及学校表示衷心的感谢！

- **佛山市南海区大沥镇中心小学姚志城老师**：您是带领我接触计算机，学会编程的领路人。十年前，与您交谈时，您说过："我们国家目前还没有自主的、成熟的操作系统"。这句话，为我的梦想埋下了种子。您培养了我对计算机的热爱，因此我选择了软件工程这个专业。感谢当年您的教导，师恩难忘！
- **佛山市南海区石门实验学校**：在石实就读的三年里，非常感谢石实的"扬长教育"理念，在老师们的培养下，让我充分发挥了自己的个性和特长，也取得了不错的成绩。在石实的三年里，我学会了 C++、Java 以及简单的算法，也自己开发了几个安卓 app，积累了将近 6 千行的代码量。
- **佛山市南海区石门中学**："任重道远，毋忘奋斗"是石中的校训，我想，这句校训，也应当成为我们每个新时代青年人的座右铭。在石门中学的三年，家国情怀教育对我产生了很大的影响。我想，我们作为新时代的青年，更应当肩负起时代的重任，奋勇拼搏，为祖国的发展，为民族的自强，为人类的未来，努力奋斗！
- **华南理工大学**："博学慎思，明辨笃行"，在华工，我得到了进一步的学习与发展。开拓了自己的视野，学会了跟很多人打交道。并且，在软件学院，我遇到了一群认真负责的老师。非常感谢学院对我的支持，支持我们成立项目组。我相信，在学院的支持下，能让 DragonOS 取得更好的发展，走的更远！
- **华南理工大学软件学院王国华老师**：王老师是我《操作系统》课程的老师，在她的指导下，我对操作系统的原理有了更深的理解，并参加了"泛珠三角 + 大学生计算机作品赛"，在 2022 年 6 月的广东省选拔赛中，DragonOS 取得了一等奖、最佳创新奖的好成绩。
- **华南理工大学软件学院汤峰老师**：汤老师是我们在校内的项目组的指导老师。在她的悉心指导下，我们将不断前行，保持我们前进的方向，持续构建开源社区。我由衷地感谢汤老师的指导！
- **Yaotian Feng**：在 Bilibili 上认识了这位非常厉害的老哥，解答了我的很多问题，很多次在我毫无头绪的 debug 了几天之后，几句话点醒了我，让我找到解决问题的路径。并且，他也跟我分享了容易踩坑的地方，让我在将要踩坑的时候能够有个心理预期，不至于那么难受哈哈哈哈。

17.1.3 贡献者名单

DragonOS V0.1.0 版本的发布，离不开以下小伙伴们的共同努力：

- 龙进 longjin@RinGoTek.cn
- zzy666-hw zzy666@mail.ustc.edu.cn
- 关锦权 guanjinquan@DragonOS.org
- 周于🄵 zhouyuzhe@DragonOS.org
- kkkkkong kongweichao@DragonOS.org
- houmkh jiaying.hou@qq.com
- wang904 1234366@qq.com
- Liric Mechan i@liric.cn
- Mustang handsomepd@qq.com
- Eugene caima12138@foxmail.com
- kun 1582068144@qq.com
- zhujikuan 1335289286@qq.com
- Alloc Alice 1548742234@qq.com

17.1.4 赞助者名单

感谢以下同学的赞赏，我们将不断努力！

- TerryLeeSCUT
- 悟
- slientbard

17.1.5 内核

遵循的一些标准规范

- 启动引导：Multiboot2
- 系统接口：posix 2008

硬件架构

- 目前支持在 x86-64 架构的处理器上运行

Bootloader

- 使用 Grub 2.06 作为 bootloader

内存管理

- 实现了基于 bitmap 的页分配器
- 实现了 slab 分配器，用来分配小块的、具有对齐要求的内存
- 抽象出 VMA（虚拟内存区域）
- 实现 VMA 反向映射机制
- 实现 MMIO 地址空间自动映射机制

多核

- 支持多核引导。也就是说，在 DragonOS 启动后，将会启动 AP 处理器。但是，为了简化其他内核模块的实现，目前 AP 处理器上，暂时没有任务在运行。
- 粗略实现了 IPI（处理器核间通信）框架

进程管理

- 支持进程的创建、回收
- 内核线程
- kthread 机制
- 用户态、内核态进程/线程的 fork/vfork（注意，用户态的 fork 和内核态的有一定的区别，内核态的 fork 更复杂）
- exec 让进程去执行一个新的可执行文件
- 进程的定时睡眠（sleep)（支持 spin/rdtsc 高精度睡眠、支持上下文切换方式的睡眠）

同步原语

- spinlock 自旋锁
- mutex 互斥量
- atomic 原子变量
- wait_queue 等待队列
- semaphore 信号量

调度相关

- CFS 调度器
- 单核调度（暂时不支持多核负载均衡）
- completion “完成”机制，让一个进程能等待某个任务的完成。

IPC 进程间通信

- 匿名管道

文件系统

- VFS 虚拟文件系统的基本功能
- FAT32 文件系统（尚不支持删除文件夹）
- devfs 设备文件系统。目前只将键盘文件注册到其中。
- rootfs 根文件系统，在真实的磁盘文件系统被挂载前，为其他的伪文件系统提供支持。
- 挂载点抽象。目前实现了文件系统的挂载，使用类似于栈的方式管理所有的挂载点。（将来需要优化这部分）

异常及中断处理

- 处理器异常的捕获
- 对 APIC 的支持
- softirq 软中断机制
- 能够对内核栈进行 traceback

内核数据结构

- 普通的二叉树
- kfifo 先进先出缓冲区
- 循环链表
- IDR 映射数据结构
- IDA ID 分配数据组件

屏幕显示

- VESA VBE 显示芯片驱动
- 实现了屏幕管理器，支持多个显示框架注册到屏幕管理器中。
- 实现了 TextUI 文本界面框架，能够渲染文本到屏幕上。并且预留了上下滚动翻页、多显示窗口的支持。
- printk

内核实用库

- 字符串操作库
- ELF 可执行文件支持组件
- 基础数学库
- CRC 函数库

软件移植

- 移植了 LZ4 压缩库（V1.9.3），为将来实现页面压缩机制打下基础。

内核测试

- ktest 单元测试框架
- 支持使用串口（COM1）输出屏幕内容到文件之中。

驱动程序支持

- IDE 硬盘
- AHCI 硬盘（SATA Native)
- ACPI 高级电源配置模块
- PCI 总线驱动
- XHCI 主机控制器驱动（usb3.0)
- ps/2 键盘
- ps/2 鼠标
- HPET 高精度定时器
- RTC 时钟
- local APIC 定时器
- UART 串口（支持 RS-232)
- VBE 显示
- 虚拟 tty 设备

系统调用

DragonOS 目前一共有 22 个有效的系统调用。

- SYS_PUT_STRING 往屏幕上打印字符
- SYS_OPEN 打开文件
- SYS_CLOSE 关闭文件
- SYS_READ 读取文件
- SYS_WRITE 写入文件
- SYS_LSEEK 调整文件指针
- SYS_FORK fork 系统调用
- SYS_VFORK vfork 系统调用
- SYS_BRK 调整堆大小为指定值
- SYS_SBRK 调整堆大小为相对值
- SYS_REBOOT 重启（将来 sysfs 完善后，将删除这个系统调用，请勿过度依赖这个系统调用）
- SYS_CHDIR 切换进程的工作目录
- SYS_GET_DENTS 获取目录中的目录项的元数据

- SYS_EXECVE 让当前进程执行新的程序文件
- SYS_WAIT4 等待进程退出
- SYS_EXIT 退出当前进程
- SYS_MKDIR 创建文件夹
- SYS_NANOSLEEP 纳秒级睡眠（最长 1 秒）在小于 500ns 时，能够进行高精度睡眠
- SYS_CLOCK 获取当前 cpu 时间
- SYS_PIPE 创建管道
- SYS_MSTAT 获取系统当前的内存状态信息
- SYS_UNLINK_AT 删除文件夹或删除文件链接

Rust 支持

- 实现了一个简单的 rust 语言的 hello world，计划在接下来的版本中，逐步转向使用 rust 进行开发。

17.1.6 用户环境

LibC

LibC 是应用程序与操作系统交互的纽带。DragonOS 的 LibC 实现了一些简单的功能。

- malloc 堆内存分配器
- 基础数学库
- 简单的几个与文件相关的函数
- pipe
- fork/vfork
- clock
- sleep
- printf

Shell 命令行程序

- 基于简单的字符串匹配的解析（不是通过编译课程学的的那一套东西做的，因此比较简单，粗暴）
- 支持的命令：ls,cd,mkdir,exec,about,rmdir,rm,cat,touch,reboot

用户态驱动程序

- 用户态键盘驱动程序

17.1.7 源码、发布版镜像下载

您可以通过以下方式获得源代码:

通过 Git 获取

- 您可以访问https://github.com/fslongjin/DragonOS/releases下载发布版的代码，以及编译好的，可运行的磁盘镜像。
- 我们在 gitee 上也有镜像仓库可供下载：https://gitee.com/DragonOS/DragonOS

通过 DragonOS 软件镜像站获取

为解决国内访问 GitHub 慢、不稳定的问题，同时为了方便开发者们下载 DragonOS 的每个版本的代码，我们特意搭建了镜像站，您可以通过以下地址访问镜像站：

您可以通过镜像站获取到 DragonOS 的代码压缩包，以及编译好的可运行的磁盘镜像。

- https://mirrors.DragonOS.org
- https://mirrors.DragonOS.org.cn

17.1.8 开放源代码声明

备注： 为促进 DragonOS 项目的健康发展，DragonOS 以 GPLv2 开源协议进行发布。所有能获得到 DragonOS 源代码以及相应的软件制品（包括但不限于二进制副本、文档）的人，都能享有我们通过 GPLv2 协议授予您的权利，同时您也需要遵守协议中规定的义务。

这是一个相当严格的，保护开源软件健康发展，不被侵占的协议。

对于大部分的善意的人们而言，您不会违反我们的开源协议。

我们鼓励 DragonOS 的自由传播、推广，但是请确保所有行为没有侵犯他人的合法权益，也没有违反 GPLv2 协议。

请特别注意，对于违反开源协议的，尤其是**商业闭源使用以及任何剽窃、学术不端行为将会受到严肃的追责**。（这是最容易违反我们的开源协议的场景）。

并且，请注意，按照 GPLv2 协议的要求，基于 DragonOS 修改或二次开发的软件，必须同样采用 GPLv2 协议开源，并标明其基于 DragonOS 进行了修改。亦需保证这些修改版本的用户能方便的获取到 DragonOS 的原始版本。

您必须使得 DragonOS 的开发者们，能够以同样的方式，从公开渠道获取到您二次开发的版本的源代码，否则您将违反 GPLv2 协议。

关于协议详细内容，还敬请您请阅读项目根目录下的 **LICENSE** 文件。请注意，按照 GPLv2 协议的要求，**只有英文原版才具有法律效力**。任何翻译版本都仅供参考。

开源软件使用情况

DragonOS 在开发的过程中，参考了一些开源项目的设计，或者引入了他们的部分代码，亦或是受到了他们的启发。现将他们列在下面。我们对这些开源项目的贡献者们致以最衷心的感谢！

格式：< 项目名 > - < 链接 > - < 开源协议 >

- Linux - https://git.kernel.org/ - GPLv2
- skiftOS - https://github.com/skiftOS/skift - MIT
- FYSOS - https://github.com/fysnet/FYSOS - FYSOS’ License
- LemonOS - https://github.com/LemonOSProject/LemonOS.git - BSD 2-Clause License
- LZ4 - https://github.com/lz4/lz4 - BSD 2-Clause license
- SerenityOS - https://github.com/SerenityOS/serenity.git - BSD 2-Clause license
- MINE - 《一个 64 位操作系统的设计与实现》田宇; 人民邮电出版社
- chcore - 《现代操作系统：设计与实现》陈海波，夏虞斌; 机械工业出版社
- SimpleKernel - https://github.com/Simple-XX/SimpleKernel - MIT

CHAPTER 18

Indices and tables

- genindex
- modindex
- search