Linux 操作系统

作业报告

班级： 2104102

学号： 2021211041

姓名： 蒲海博

哈尔滨工业大学（威海）计算机学院

报告撰写要求

1） 正文使用五号字，单倍行距，中文使用宋体，英文使用 **times new roma** 字体；
2） 注意行文的规范，段落空行等；
3） 报告采用双面打印，使用左侧装订方式；
4） 报告电子版请提供 **pdf** 格式。

（此页需要打印）

成绩评定标准表

评审项目	评定内容	成绩标准	成绩
报告内容完成情况	作业内容所有项目是否完成，结果是否准确，相关原理和算法介绍是否完整等	90 分	
报告格式规范性	文档格式是否一致、规范；文字是否一致、规范；图表是否规范；术语是否准确等	10 分	
总体评价		总成绩	

1. 分析 ext4 文件系统原理。 20 分

（1）ext4 文件系统概述

ext4 文件系统，即第四代扩展文件系统（Fourth Extended filesystem)，是 linux 系统下的日志文件系统， 是 ext3 文件系统的后继版本，其文件系统容量达到 1EB,理论上支持无限数量的子目录。ext4 可以提供更佳的性能和可靠性，还有更为丰富的功能。

（2）容量概念

A. 扇区（sector）

扇区是磁盘最小的储存单位，可以通过命令行 fdisk-1 得知单位每扇区的大小（一般是 512byte）,机械硬盘 HDD 的可用空间大小计算公式是 heads（磁头数量）cylinders（柱面数量）sectors（扇区数量）每个 sector 大小（512byte）。固态硬盘 SSD 没有磁头，主面的概念，所以固态可用空间的总大小是 sectors（扇区数量）每个 sector 大小（512byte）。这几个属性是固定不能修改，但可以通过命令读取得到。

B. 块（block）

块是文件系统 EXT4,FAT32,XFS 等最小的储存单位，使用命令 blkid 可查看文件系统类型。每一个块只能存储一个文件的数据。默认操作系统每个块的大小是 4k（4096bytes），一个 block 由连续的 sector 组成。一个文件在文件系统中的块不一定是连续的。

C. 索引节点（inode）

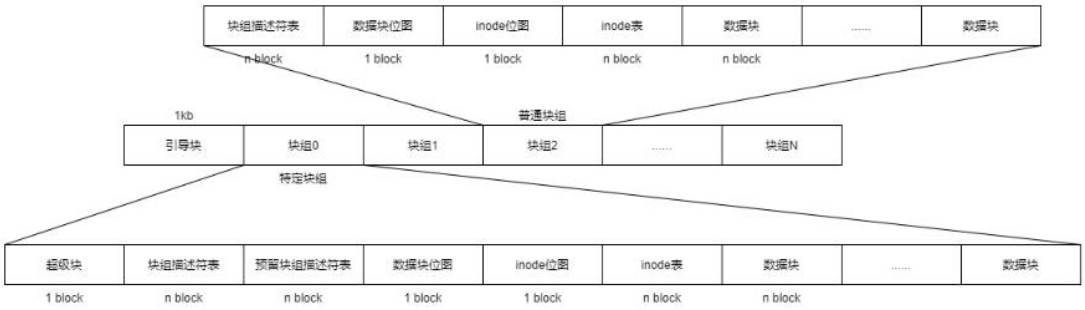
记录文件的权限、属性和数据所在块 block 的号码，每个文件都有且仅有一个的 inode，每 个 inode 都有自己的编号，可以把 inode 简单地理解为文档索引。

D. 数据区块（data block）

存储的文件内容，也叫数据区块 （datablock）,每个 block 都有自己的编号。目录的 data block 保存该目录下所有文件以及子目录的名字，inode 编号等信息。

（3） Ext4 文件系统的结构

一个 Ext4 文件系统被分成一系列块组。为减少磁盘碎片产生的性能瓶颈，块分配器尽量保持每个文件的数据块都在同一个块组中，从而减少寻道时间。以 4KB 的数据块为例，一个块组可以包含 32768 个数据块，也就是 128MB。每个块组一般包括超级块、块组描述符表、预留块组描述符表、数据位图、inode 位图、inode 表、数据块。Ext4 文件系统主要使用块组 0 中的超级块和块组描述符表，在特定的块组(譬如说 0,3,5,7)才有超级块和块组描述符表的冗余备份。普通块组中不含冗余备份，那么块组就以数据块位图开始。如下图所示：



A.引导块

为磁盘分区的第一个块，记录文件系统分区的一些信息，引导加载当前分区的程序和数据被保存在这个块中。系统初始时根据 MBR 的信息来识别硬盘，其中包括了一些执行文件就来载入系统，这些执行文件就是 MBR 里前面 446 字节里的 boot loader 程式，而后面的 16 字节 X4 的空间就是存储分区表信息的位置，最后以 0x55AA 这两个字节

结束，分区表主要储存一下三种信息：分区号、分区起始位置、分区大小。

如下图：

B.超级块

超级块用于存储文件系统全局的配置参数(如：块大小，总的块数和 inode 数)和动态信息(如：当前空闲块数和 inode 数)，其处于文件系统开始位置的 1k 处，所占大小为 1k。为了系统的健壮性，最初每个块组都有超级块和组描述符表(以下将用 GDT)的一个拷贝，但是当文件系统很大时，这样浪费了很多块(尤其是 GDT 占用的块多)，后来采用了一种稀疏的方式来存储这些拷贝，只有块组号是 3, 5 ,7 的幂的块组(如 0,3,5,7)才备份这个拷贝。通常情况下，只有主拷贝(第 0 块块组)的超级块信息被文件系统使用，其它拷贝只有在主拷贝被破坏的情况下才使用.

C.块组描述符

GDT 用于存储块组描述符，其占用一个或者多个数据块，具体取决于文件系统的大小。它 主要包含块位图，inode 位图和 inode 表位置，当前空闲块数，inode 数以及使用的目录数。 每个块组都对应这样一个描述符，目前该结构占用 32 个字节，因此对于块大小为 4k 的文 件系统来说，每个块可以存储 128 个块组描述符。由于 GDT 对于定位文件系统的元数据非 常重要，因此和超级块一样，也对其进行了备份。GDT 在每个块组（如果有备份）中内容都 是一样的，其所占块数也是相同的。从上面的介绍可以看出块组中的元数据如块位图，inode 位图,inode 表其位置不是固定的。

D.块组

每个块组包含一个块位图块,一个 inode 位图块，一个或多个块用于描述 inode 表和用于存 储文件数据的数据块，除此之外，还有可能包含超级块和所有块组描述符表(取决于块组号 和文件系统创建时使用的参数)。

E.数据块位图

块位图用于描述该块组所管理的块的分配状态。如果某个块对应的位未置位，那么代表该块未分配，可以用于存储数据；否则，代表该块已经用于存储数据或者该块不能够使用(譬如该块物理上不存在)。由于块位图仅占一个块，因此这也就决定了块组的大小。如果一个数据块大小是 4KB 的话，那一个位图块可以表示 410248 个数据块的使用情况，这也是单个块组具有的最大数据块个数。这样可以算出一个块组大小是 128MB。

F.Inode 位图

Inode 位图用于描述该块组所管理的 inode 的分配状态。我们知道 inode 是用于描述文件的元数据，每个 inode 对应文件系统中唯一的一个号，如果 inode 位图中相应位置位，那么代表该 inode 已经分配出去；否则可以使用。由于其仅占用一个块，因此这也限制了一个块组中所能够使用的最大 inode 数量。

G.Inode 表

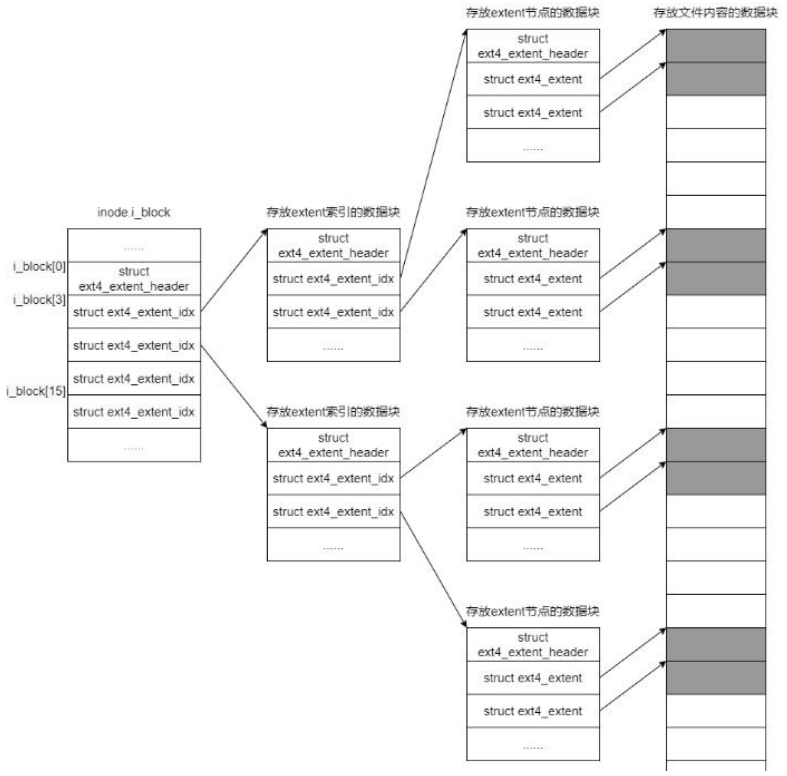
Inode 表用于存储 inode 信息。它占用一个或多个块(为了有效的利用空间，多个 inode 存储在一个块中)，其大小取决于文件系统创建时的参数，由于 inode 位图的限制，决定了其最大所占用的空间。

以上这几个构成元素所处的磁盘块成为文件系统的元数据块，剩余的部分则用来存储

真正的文件内容，称为数据块，而数据块其实也包含数据和目录。

功能	占用	备注
GroupO padding	1024bytes	仅 groupO 含有
ext4 super block	1 block	仅某些 group 含有
Group Descriptors	many blocks	仅某些 group 含有
Reserved GDT Blocks	many blocks	仅某些 group 含有
Data Block Bitmap	1 block,	所有 group 均含有
inode Bitmap	1 block	所有 group 均含有
inode Table	many blocks	所有 group 均含有
Data Blocks	many blocks	所有 group 均含有

文件的大小决定着他的存放方式。如果一个文件的大小小于 60 字节，文件的内容是直接放在 inode 中，没有对应的数据块。如果一个文件的大小大于 60 字节，小于 60KB(15 个数组，每个数组存放一个数据块编号,每个数据块为 4K,4KB*15=60KB)。这时候文件内容存放在数据块。如果一个文件的大小大于 60KB，就需要使用到 Extent 树结构体了。



(4)数据块和 Inode 分配策略

在机械磁盘上，保持相关的数据块相互接近可以总的磁头移动时间，因而可以加速磁盘 10。 在 SSD 上虽然没有磁头转动，数据局部性可以增加每次 10 请求的传输的数据大小，因而 减少响应 IO 请求的传输次数。数据的局部性对单个擦除块的写入产生影响，可以加速文件重写的速度。因而尽可能减少碎片是必要的。inode 和数据块的分配策略可以保证数据的局部集中。以下为 inode 和数据块的分配策略：

A.多块分配可以减少磁盘碎片。

当文件初次创建的时候，块分配器预测性地分配 8KB 的 磁盘空间给文件。当文

件关闭的时候，未使用的空间当然也就释放了。但是如果推测是 正确的，那么文件数据将写到一个多个块的 extent 中。

B.延迟分配。

当一个文件需要更多的数据块引起写操作时，文件系统推迟决定新数据在磁 盘上的存放位置，直到脏的 buffer 写到磁盘为止。

C.尽量保持文件的数据块与其 inode 在同一个块组中。可以减少磁盘寻找时间.

D.尽量保持同一个目录中的所有 inodes 与目录位于同一个块组中。前提是一个 目录中的文件是相关的。

E.磁盘卷被分成 128MB 的块组。

当在根目录中创建目录时，inode 分配器扫描块组并将新 目录放到它找到的使用负荷最小的块组中。这可以保证目录在磁盘上的分散性。

(5)读取文件的过程

A.根据文件所在目录的 inode 信息，找到目录文件对应数据块

B.根据文件名从数据块中找到对应的 inode 节点信息

C.从文件 inode 节点信息中找到文件内容所在数据块块号

D.读取数据块内容

(6)文件的拷贝、剪切的底层过程

A.拷贝文件：创建一个新的 inode 节点，并且拷贝数据块内容

B.剪切文件：同个分区里边 mv, inode 节点不变，只是更新目录文件对应数据块里边的文 件名和 inode 对应关系；跨分区 mv,则跟拷贝一个道理，需要创建新的 inode,因为 inode 节点不同分区是不能共享的。

(7)软连接和硬连接实现过程

A.软连接：创建软连接会创建一个新的 inode 节点，其对应数据块内容存储所链接的文件名信息，这样原文件即便删除了，重新建立一个同名的文件，软连接依然能够生效。

B.硬链接：创建硬链接，并不会新建 inode 节点，只是 links 加 1,还有再目录文件对应数据块上增加一条文件名和 inode 对应关系记录；只有将硬链接和原文件都删除之后，文件才会真正删除，即 links 为 0 才真正删除。

(8)ext4 相较于 ext3 的改进和联系

A.与 Ext3 兼容

ext4 特地设计为尽可能地向后兼容 ext3。这不仅允许 ext3 文件系统原地升级到 ext4；也允许 ext4 驱动程序以 ext3 模式自动挂载 ext3 文件系统，因此使它无需单独维护两个代码库。

B.更大的文件系统和更大的文件。

ext3 文件系统使用 32 位寻址，这限制它仅支持 2 TiB 文件大小和 16 TiB 文件系统系统大小（这是假设在块大小为 4 KiB 的情况下，一些 ext3 文件系统使用更小的块大小，因此对其进一步被限制）。ext4 使用 48 位的内部寻址，理论上可以在文件系统上分配高达 16 TiB 大小的文件，其中文件系统大小最高可达 1000000 TiB（1 EiB）。在早期 ext4 的实现中有些用户空间的程序仍然将其限制为最大大小为 16 TiB 的文件系统，但截至 2011 年，e2fsprogs 已经直接支持大于 16 TiB 大小的 ext4 文件系统。例如，红帽企业 Linux 在其合同上仅支持最高 50 TiB 的 ext4 文件系统，并建议 ext4 卷不超过 100 TiB。

C.无限数量的子目录

Ext3 目前只支持 32,000 个子目录，而 Ext4 支持无限数量的子目录。

D.Extents

Ext3 采用间接块映射，当操作大文件时，效率极其低下。一个 100MB 大小的文件，在 Ext3 中要建立 25,600 个数据块(每个数据块大小为 4KB)的映射表。而 Ext4 引入了现代文件系统中 extents 概念，每个 extent 为一组连续的数据块，上述文件则 表示为“该文件数据保存在接下来的 25,600 个数据块中”，提高了不少效率。

E.多块分配

当写入数据到 Ext3 文件系统中时，Ext3 的数据块分配器每次只能分配一个 4KB 的 块，写一个 100MB 文件就要调用 25,600 次数据块分配器，而 Ext4 的多块分配器“multiblock allocator”(mballoc)支持一次调用分配多个数据块。

F.延迟分配

Ext3 的数据块分配策略是尽快分配，而 Ext4 和其它现代文件操作系统的策略是尽可能地延迟分配，直到文件在 cache 中写完才开始分配数据块并写入磁盘，这样就能优 化整个文件的数据块分配，与前两种特性搭配起来可以显著提升性能。

etx4 的不足

ext4 只是一个纯 文件系统，而不是存储卷管理器。这意味着，即使你有多个磁盘 —— 也就是奇偶校验或冗余，理论上你可以从 ext4 中恢复损坏的数据，但无法知道使用它是否对你有利。虽然理论上可以在不同的层中分离文件系统和存储卷管理系统而不会丢失自动损坏检测和修复功能，但这不是当前存储系统的设计方式，并且它将给新设计带来重大挑战。

2.分析 linux(4.X 版本以上)进程调度源码，尝试解释进程调度 CFS 调度算法原理，给出主要数据结构和函数的相关解释（给出代码出处——路径和文件名）30 分

⑴概述

进程调度是操作系统中的一个重要机制，它负责按照一定的策略将系统资源分配给各个进程，以实现多个进程的公平竞争和合理利用。 在 Linux 内核中，进程调度的核心是完全公平调度（Completely Fair Scheduler，CFS）算法。CFS 算法基于红黑树结构实现，通过虚拟运行时间来衡量进程的优先级。进程的优先级反映了进程与 CPU 资源的竞争程度，优先级较高的进程能够更早地获得 CPU 资源。

Completely Fair Scheduler，完全公平调度器，用于 Linux 系统中普通进程的调度。

CFS 采用了红黑树算法来管理所有的调度实体 sched_entity，算法效率为 O(log(n))。CFS 跟踪调度实体 sched_entity 的虚拟运行时间 vruntime，平等对待运行队列中的调度实体 sched_entity，将执行时间少的调度实体 sched_entity 排列到红黑树的左边。

调度实体 sched_entity 通过 enqueue_entity()和 dequeue_entity()来进行红黑树的出队入队。

(2)主要原理

CFS 算法使用了虚拟运行时间（vruntime）作为度量进程的优先级的标准。虚拟运行时间可以理解为进程在没有被限制使用 CPU 时实际消耗的时间。CFS 调度器通过不断更新进程的 vruntime 来保证公平性。每个进程的 vruntime 是一个动态值，VRuntime = 基准时间 + 节点的权重 * 使用时间。进程被调度时，CFS 调度器会选择具有最小 vruntime 值的进程运行，即选择虚拟运行时间最小的进程获得 CPU 资源。

(3)数据结构

A.调度类

Linux 内核抽象了一个调度类 struct sched class,这是一种典型的面向对象的设计思想，将共性的特征抽象出来封装成类，在实两化各个调度器的时候，可以根据具体的调 度算法来实现。这种方式做到了高内聚低耦合，同时又很容易扩展新的调度器。

在调度核心代码"linux-5.10.0\linux-5.10.0\kernel\sched\core.c"中，使用的方式是 task->sched_class->xxx_func，其中 task 表示的是描述任务的结构体 struct task_stmct,

在该结构局包含了往务所使用的调度器，进而能找到对应的函数指针来完成蔼用执行，有点类似于 C++中的多态机制。

C:\linux-5.10.0\linux-5.10.0\kernel\sched\sched.h

```
struct sched_class {

#ifdef CONFIG_UCLAMP_TASK
    int uclamp_enabled;
#endif

    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*yield_task)   (struct rq *rq);
    bool (*yield_to_task)(struct rq *rq, struct task_struct *p);

    void (*check_preempt_curr)(struct rq *rq, struct task_struct *p, int flags);

    struct task_struct *(*pick_next_task)(struct rq *rq);

    void (*put_prev_task)(struct rq *rq, struct task_struct *p);
    void (*set_next_task)(struct rq *rq, struct task_struct *p, bool first);

#ifdef CONFIG_SMP
    int (*balance)(struct rq *rq, struct task_struct *prev, struct rq_flags *rf);
    int  (*select_task_rq)(struct task_struct *p, int task_cpu, int flags);

    struct task_struct * (*pick_task)(struct rq *rq);

    void (*migrate_task_rq)(struct task_struct *p, int new_cpu);

    void (*task_woken)(struct rq *this_rq, struct task_struct *task);

    void (*set_cpus_allowed)(struct task_struct *p,
                 const struct cpumask *newmask,
                 u32 flags);

    void (*rq_online)(struct rq *rq);
    void (*rq_offline)(struct rq *rq);

    struct rq *(*find_lock_rq)(struct task_struct *p, struct rq *rq);
#endif

    void (*task_tick)(struct rq *rq, struct task_struct *p, int queued);
    void (*task_fork)(struct task_struct *p);
    void (*task_dead)(struct task_struct *p);

    /*
     * The switched_from() call is allowed to drop rq->lock, therefore we
     * cannot assume the switched_from/switched_to pair is serialized by
     * rq->lock. They are however serialized by p->pi_lock.
     */
    void (*switched_from)(struct rq *this_rq, struct task_struct *task);
    void (*switched_to)  (struct rq *this_rq, struct task_struct *task);
    void (*prio_changed) (struct rq *this_rq, struct task_struct *task,
                  int oldprio);

    unsigned int (*get_rr_interval)(struct rq *rq,
                    struct task_struct *task);

    void (*update_curr)(struct rq *rq);

#define TASK_SET_GROUP      0
#define TASK_MOVE_GROUP     1

#ifdef CONFIG_FAIR_GROUP_SCHED
    void (*task_change_group)(struct task_struct *p, int type);
#endif
};
```

B.rq/cfs_rq/task_struct/task_group/sched_entity

struct rq：每个 CPU 都右一个对应的运行队列；

structcfs_rq： CFS 运行队列，该结构中包含了 structrb_root_cached 红黑树，用于链接调度实体 struct sched_entityo rq 运行队列中对应了一个 CFS 运行队列，此外，在 task_group 结构中也会为每个 CPU 再维护一个 CFS 运行队列；

struct task_struct：任务的描述符，包含了进程的所有信息，该结构中的 struct

sched_entity,用于参与 CFS 的调度；

struct task_group：组调度，Linux 支持将任务分组来对 CPU 资源进行 分配管理，该结构中为系统中的每个CPU都分配了 struct sched entity调度实体和 struct cfs_rq运行队列，其中 struct sched entity 用于参与 CFS 的调度；

struct sched_entity：调度实体，也是 CFS 调度管理的对象；

```
struct cfs_rq {
    struct load_weight load;              //CFS运行队列的负载权重值
    unsigned int nr_running, h_nr_running; //nr_running: 运行的调度实体数（参与时间片计算）

    u64 exec_clock;    //运行时间
    u64 min_vruntime;  //最少的虚拟运行时间，调度实体入队出队时需要进行增减处理
#ifndef CONFIG_64BIT
    u64 min_vruntime_copy;
#endif

    struct rb_root_cached tasks_timeline;  //红黑树，用于存放调度实体

    /*
     * 'curr' points to currently running entity on this cfs_rq.
     * It is set to NULL otherwise (i.e when none are currently running).
     */
    struct sched_entity *curr, *next, *last, *skip; //分别指向当前运行的调度实体、下一个调度的调度实体、CFS运行队列中排最后

#ifdef    CONFIG_SCHED_DEBUG
    unsigned int nr_spread_over;
#endif

#ifdef CONFIG_SMP
    /*
     * CFS load tracking
     */
    struct sched_avg avg;              //计算负载相关
    u64 runnable_load_sum;
    unsigned long runnable_load_avg;   //基于PELT的可运行平均负载
#ifdef CONFIG_FAIR_GROUP_SCHED
    unsigned long tg_load_avg_contrib; //任务组的负载贡献
    unsigned long propagate_avg;
#endif
    atomic_long_t removed_load_avg, removed_util_avg;
#ifndef CONFIG_64BIT
    u64 load_last_update_time_copy;
#endif

#ifdef CONFIG_FAIR_GROUP_SCHED
    /*
     *   h_load = weight * f(tg)
     *
     * Where f(tg) is the recursive weight fraction assigned to this group.
     */
    unsigned long h_load;
    u64 last_h_load_update;
    struct sched_entity *h_load_next;
#endif /* CONFIG_FAIR_GROUP_SCHED */
#endif /* CONFIG_SMP */

#ifdef CONFIG_FAIR_GROUP_SCHED
    struct rq *rq;    /* cpu runqueue to which this cfs_rq is attached */ //指向CFS运行队列所属的CPU RQ运行队列

    /*
     * leaf cfs_rqs are those that hold tasks (lowest schedulable entity in
     * a hierarchy). Non-leaf lrqs hold other higher schedulable entities
     * (like users, containers etc.)
     *
     * leaf_cfs_rq_list ties together list of leaf cfs_rq's in a cpu. This
     * list is used during load balance.
     */
    int on_list;
    struct list_head leaf_cfs_rq_list;
    struct task_group *tg;    /* group that "owns" this runqueue */ //CFS运行队列所属的任务组

#ifdef CONFIG_CFS_BANDWIDTH
    int runtime_enabled;   //CFS运行队列中使用CFS带宽控制
    u64 runtime_expires;   //到期的运行时间
    s64 runtime_remaining; //剩余的运行时间

    u64 throttled_clock, throttled_clock_task; //限流时间相关
    u64 throttled_clock_task_time;
    int throttled, throttle_count;   //throttled: 限流, throttle_count: CFS运行队列限流次数
    struct list_head throttled_list; //运行队列限流链表节点，用于添加到cfs_bandwidth结构中的cfttle_cfs_rq链表中
#endif /* CONFIG_CFS_BANDWIDTH */
#endif /* CONFIG_FAIR_GROUP_SCHED */
};
```

2分析

程分析，围绕着CFS调度类实体：fair_sched_class 中的关键函数来展开。

看fair_sched_class都包含了哪些函数：

```
/* All the scheduling class methods: */
const struct sched_class fair_sched_class = {
    .next          = &idle_sched_class,
    .enqueue_task  = enqueue_task_fair,
    .dequeue_task  = dequeue_task_fair,
    .yield_task    = yield_task_fair,
    .yield_to_task = yield_to_task_fair,
```

(3) 相关函数解释

A. 进程选择模块　　代码出处"C:\linux-5.10.0\linux-5.10.0\kernel\sched\fair.c”

通过选择红黑树最左的节点，即所含 vruntime 最小的节点作为下一节点

```
static struct sched_entity *__pick_next_entity(struct sched_entity *se)
{
    struct rb_node *next = rb_next(&se->run_node);

    if (!next)
        return NULL;

    return __node_2_se(next);
}
```

B. 向红黑树中插入事件 代码出处"C:\linux-5.10.0\linux-5.10.0\kernel\sched\fair.c”

这一步骤发生在进程变成可运行态，或者通过 fork 系统调用第一次创建进程时。

```
static void
enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    bool renorm = !(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_MIGRATED);
    bool curr = cfs_rq->curr == se;

    /*
     * If we're the current task, we must renormalise before calling
     * update_curr().
     */
    if (renorm && curr)
        se->vruntime += cfs_rq->min_vruntime;

    update_curr(cfs_rq);

    /*
     * Otherwise, renormalise after, such that we're placed at the current
     * moment in time, instead of some random moment in the past. Being
     * placed in the past could significantly boost this task to the
     * fairness detriment of existing tasks.
     */
    if (renorm && !curr)
        se->vruntime += cfs_rq->min_vruntime;

    /*
     * When enqueuing a sched_entity, we must:
     *   - Update loads to have both entity and cfs_rq synced with now.
     *   - Add its load to cfs_rq->runnable_avg
     *   - For group_entity, update its weight to reflect the new share of
     *     its group cfs_rq
     *   - Add its new weight to cfs_rq->load.weight
     */
    update_load_avg(cfs_rq, se, UPDATE_TG | DO_ATTACH);
    se_update_runnable(se);
    update_cfs_group(se);
    account_entity_enqueue(cfs_rq, se);

    if (flags & ENQUEUE_WAKEUP)
        place_entity(cfs_rq, se, 0);

    check_schedstat_required();
    update_stats_enqueue_fair(cfs_rq, se, flags);
    check_spread(cfs_rq, se);
    if (!curr)
        __enqueue_entity(cfs_rq, se);
    se->on_rq = 1;

    /*
     * When bandwidth control is enabled, cfs might have been removed
     * because of a parent been throttled but cfs->nr_running > 1. Try to
     * add it unconditionally.
     */
    if (cfs_rq->nr_running == 1 || cfs_bandwidth_used())
        list_add_leaf_cfs_rq(cfs_rq);

    if (cfs_rq->nr_running == 1)
        check_enqueue_throttle(cfs_rq);
}
```

C. 从红黑树中删除进程 代码出处"C:\linux-5.10.0\linux-5.10.0\kernel\sched\fair.c”

这一步操作发生在进程阻塞，即进程变成不可运行状态或者当进程终止时。

```c
static void
dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    /*
     * Update run-time statistics of the 'current'.
     */
    update_curr(cfs_rq);

    /*
     * When dequeuing a sched_entity, we must:
     *   - Update loads to have both entity and cfs_rq synced with now.
     *   - Subtract its load from the cfs_rq->runnable_avg.
     *   - Subtract its previous weight from cfs_rq->load.weight.
     *   - For group entity, update its weight to reflect the new share
     *     of its group cfs_rq.
     */
    update_load_avg(cfs_rq, se, UPDATE_TG);
    se_update_runnable(se);

    update_stats_dequeue_fair(cfs_rq, se, flags);

    clear_buddies(cfs_rq, se);

    if (se != cfs_rq->curr)
        __dequeue_entity(cfs_rq, se);
    se->on_rq = 0;
    account_entity_dequeue(cfs_rq, se);

    /*
     * Normalize after update_curr(); which will also have moved
     * min_vruntime if @se is the one holding it back. But before doing
     * update_min_vruntime() again, which will discount @se's position and
     * can move min_vruntime forward still more.
     */
    if (!(flags & DEQUEUE_SLEEP))
        se->vruntime -= cfs_rq->min_vruntime;

    /* return excess runtime on last dequeue */
    return_cfs_rq_runtime(cfs_rq);

    update_cfs_group(se);

    /*
     * Now advance min_vruntime if @se was the entity holding it back,
     * except when: DEQUEUE_SAVE && !DEQUEUE_MOVE, in this case we'll be
     * put back on, and if we advance min_vruntime, we'll be placed back
     * further than we started -- ie. we'll be penalized.
     */
    if ((flags & (DEQUEUE_SAVE | DEQUEUE_MOVE)) != DEQUEUE_SAVE)
        update_min_vruntime(cfs_rq);
}
```

D. 调度器入口　　代码出处"C:\linux-5.10.0\linux-5.10.0\kernel\sched\sched.c”

进程调度器的入口函数为 scheduleQ,总体流程即为选择合适的调度策略选出下一个需要被调度的进程任务，然后进行一次上下文切换，将进程置为运行态。

```c
asmlinkage __visible void __sched schedule(void)
{
    struct task_struct *tsk = current;

    sched_submit_work(tsk);
    do {
        preempt_disable();
        __schedule(SM_NONE);
        sched_preempt_enable_no_resched();
    } while (need_resched());
    sched_update_worker(tsk);
}
EXPORT_SYMBOL(schedule);
```

3.linux 内核（2.6 或以上版本或者是 openEuler 的内核代码安装）的下载、安装、编译的具体步骤和截图；(虚拟机安装)20 分

（1）内核源码下载

在 **https://gitee.com/openeuler/kernel/repository/archive/**中选择 **5.10.0-13.0.0.zip** 下载

```
[hit20212110410@hecs-5357 ~]$ wget https://gitee.com/openeuler/kernel/repository/archive/5.10.0-13.0.0.zip
--2023-07-13 11:40:56--  https://gitee.com/openeuler/kernel/repository/archive/5.10.0-13.0.0.zip
Resolving gitee.com (gitee.com)... 212.64.63.190, 212.64.63.215
Connecting to gitee.com (gitee.com)|212.64.63.190|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://gitee.com/openeuler/kernel/repository/blazearchive/5.10.0-13.0.0.zip?Expires=1689220895&Signature=u3UaQvfid668
djc3gZTqjIFnEOKQBg5xECn6MxLq0x0%3D [following]
--2023-07-13 11:41:35--  https://gitee.com/openeuler/kernel/repository/blazearchive/5.10.0-13.0.0.zip?Expires=1689220895&Signatu
re=u3UaQvfid668djc3gZTqjIFnEOKQBg5xECn6MxLq0x0%3D
Reusing existing connection to gitee.com:443.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [application/zip]
Saving to: '5.10.0-13.0.0.zip'

5.10.0-13.0.0.zip                   [      <=>                                      ] 219.29M  1.16MB/s    in 2m 59s

2023-07-13 11:44:35 (1.22 MB/s) - '5.10.0-13.0.0.zip' saved [229945849]

[hit20212110410@hecs-5357 ~]$ _
```

（2）解压

```
[hit20212110410@hecs-5357 ~]$ unzip 5.10.0-13.0.0.zip
```

```
[hit20212110410@hecs-5357 ~]$ ls
5.10.0-13.0.0.zip  bash  bin  kernel-5.10.0-13.0.0  test  test.sh
[hit20212110410@hecs-5357 ~]$
```

（3）编译内核并安装

A. 系统备份

```
[hit20212110410@hecs-5357 ~]$  cd ~
[hit20212110410@hecs-5357 ~]$  yum install lrzsz
 tar czvf boot_origin.tgz /boot/
 sz boot_origin.tgz    Error: This command has to be run with superuser privileges (under the root user on most systems).
[hit20212110410@hecs-5357 ~]$  tar czvf boot_origin.tgz /boot/
tar: Removing leading `/' from member names
/boot/
tar: /boot/initramfs-5.10.0-60.18.0.50.r865_35.hce2.x86_64.img: Cannot open: Permission denied
tar: /boot/initramfs-5.10.0-60.18.0.50.r865_35.hce2.x86_64kdump.img: Cannot open: Permission denied
tar: /boot/initramfs-0-rescue.img: Cannot open: Permission denied
/boot/vmlinuz-5.10.0-60.18.0.50.r865_35.hce2.x86_64
/boot/dracut/
tar: /boot/grub2: Cannot open: Permission denied
/boot/symvers-5.10.0-60.18.0.50.r865_35.hce2.x86_64.gz
tar: /boot/System.map-5.10.0-60.18.0.50.r865_35.hce2.x86_64: Cannot open: Permission denied
/boot/vmlinuz-0-rescue
/boot/efi/
/boot/efi/EFI/
tar: /boot/efi/EFI/hce: Cannot open: Permission denied
/boot/.vmlinuz-5.10.0-60.18.0.50.r865_35.hce2.x86_64.hmac
tar: /boot/config-5.10.0-60.18.0.50.r865_35.hce2.x86_64: Cannot open: Permission denied
/boot/loader/
tar: /boot/loader/entries: Cannot open: Permission denied
tar: Exiting with failure status due to previous errors
[hit20212110410@hecs-5357 ~]$  sz boot_origin.tgz
```

B. 进入源码根目录

```
[hit20212110410@hecs-5357 ~]$ cd kernel-5.10.0-13.0.0
[hit20212110410@hecs-5357 kernel-5.10.0-13.0.0]$
```

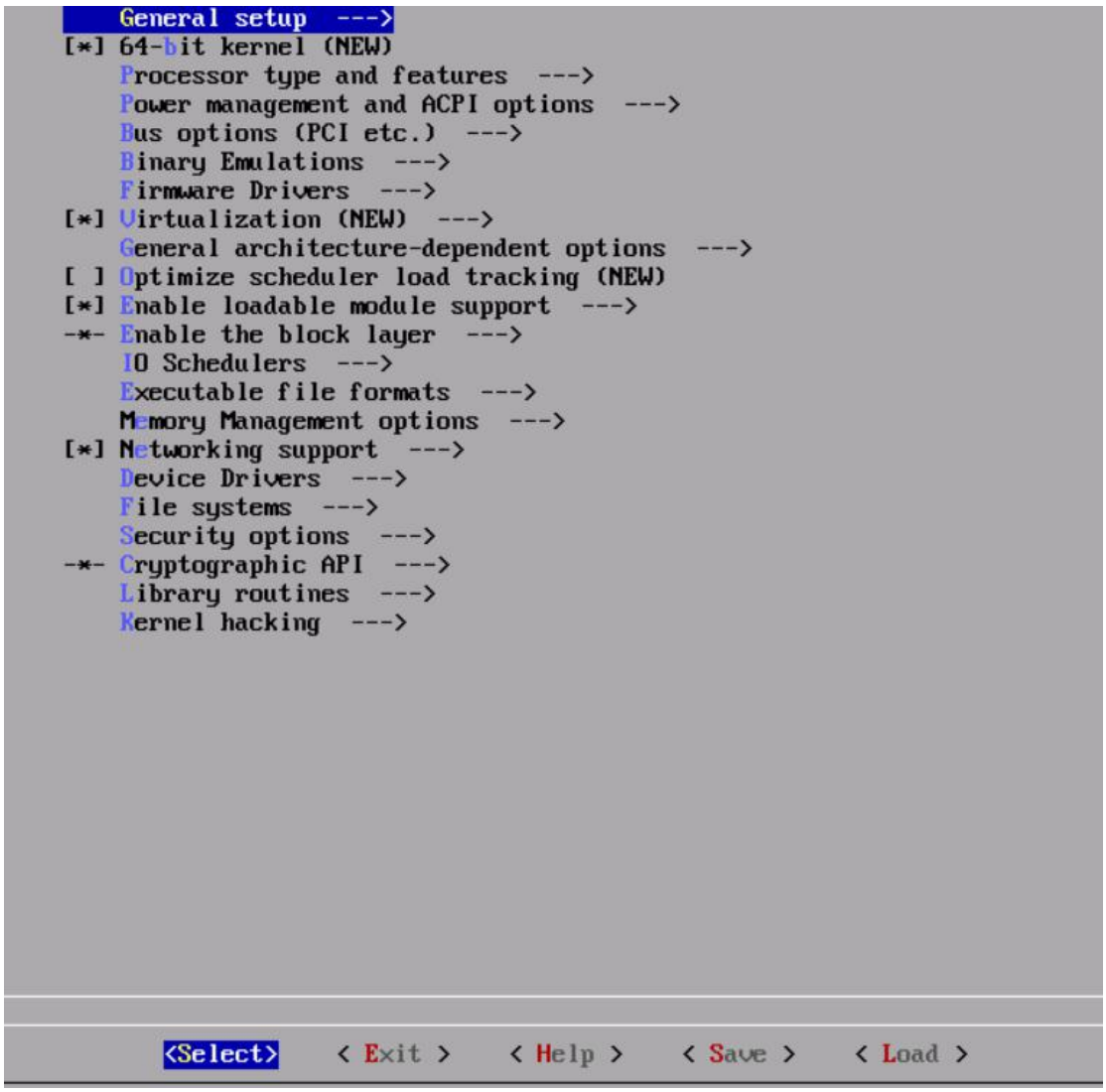
C. 生成内核配置文件.config

```
[hit20212110410@hecs-5357 kernel-5.10.0-13.0.0]$ sudo cp -v /boot/config-$(uname -r) .config
[sudo] password for hit2021211041:
'/boot/config-5.10.0-60.18.0.50.r865_35.hce2.x86_64' -> '.config'
[hit20212110410@hecs-5357 kernel-5.10.0-13.0.0]$
```

D. 配置命令

```
[hit2021211041@hecs-5357 kernel-5.10.0-13.0.0]$ make menuconfig
  HOSTCC  scripts/basic/fixdep
  UPD     scripts/kconfig/mconf-cfg
  HOSTCC  scripts/kconfig/mconf.o
  HOSTCC  scripts/kconfig/lxdialog/checklist.o
  HOSTCC  scripts/kconfig/lxdialog/inputbox.o
```

E. 编译成功



F. 安装

make -j4 安装

```
  CC      net/core/net_namespace.o
  CC      fs/lockd/procfs.o
  CC      sound/core/pcm_lib.o
  AR      fs/lockd/built-in.a
  CC      drivers/video/fbdev/core/softcursor.o
```

```
[hit2021211041@hecs-5357 kernel-5.10.0-13.0.0]$ make install
sh ./arch/x86/boot/install.sh 5.10.0 arch/x86/boot/bzImage \
        System.map "/boot"
```

（4）遇到问题

make menuconfig 时报错

```
[hit20212110410@hecs-5357 kernel-5.10.0-13.0.0]$ sudo make menuconfig
  LEX     scripts/kconfig/lexer.lex.c
/bin/sh: line 1: flex: command not found
make[1]: *** [scripts/Makefile.host:9: scripts/kconfig/lexer.lex.c] Error 127
make: *** [Makefile:603: menuconfig] Error 2
[hit20212110410@hecs-5357 kernel-5.10.0-13.0.0]$ _
```

通过下载 flex 包和 bison 包解决

```
[hit20212110410@hecs-5357 kernel-5.10.0-13.0.0]$ sudo yum install flex
Last metadata expiration check: 0:09:57 ago on Thu 13 Jul 2023 01:21:18 PM CST.
Dependencies resolved.
================================================================================
 Package              Architecture          Version                  Repository      Size
================================================================================
Installing:
 flex                 x86_64                2.6.4-3.r1.hce2          updates        311 k
Installing dependencies:
 m4                   x86_64                1.4.19-2.hce2            base           190 k

Transaction Summary
================================================================================
Install  2 Packages

Total download size: 501 k
Installed size: 1.4 M
```

```
[hit20212110410@hecs-5357 kernel-5.10.0-13.0.0]$ make menuconfig
  LEX     scripts/kconfig/lexer.lex.c
  YACC    scripts/kconfig/parser.tab.[ch]
/bin/sh: line 1: bison: command not found
make[1]: *** [scripts/Makefile.host:17: scripts/kconfig/parser.tab.h] Error 127
make: *** [Makefile:603: menuconfig] Error 2
[hit20212110410@hecs-5357 kernel-5.10.0-13.0.0]$ sudo yum install bison
Last metadata expiration check: 0:11:05 ago on Thu 13 Jul 2023 01:21:18 PM CST.
Dependencies resolved.
================================================================================
 Package              Architecture          Version                  Repository      Size
================================================================================
Installing:
 bison                x86_64                3.8.2-1.hce2             base           398 k

Transaction Summary
================================================================================
Install  1 Package

Total download size: 398 k
Installed size: 1.4 M
```

4. linux 堆内存管理原理是什么？与 windows 系统的内存管理有何区别？30 分

(1) 堆内存管理概述

不同的平台有不同的堆内存管理机制，最初 linux 默认的是 dlmalloc，但是由于 其不支持多线程管理，后来被支持多线程的 ptmalloc2 代替了。ptmalloc 是基于 glibc 实现的内存分配器，它是一个标准实现，所以兼容性较好。pt 表示 per thread 的意 思。ptmalloc 实现了 malloc(),free()以及一组其他函数，以提供动态内存管理，同时 支持多线程。分配器处于用户空间和内核空间之间，响应用户的分配请求，向操作 系统申请内存。

ptmalloc 进行堆分配时有两种方式：brk 和 mmap。brk 分配的堆区由 libc 管理， 在程序结束之前不会返还给操作系统，mmap 分配的堆区在使用后会直接 munmap 返回给系统。

（2）内存管理与数据结构

linux 的堆内存管理分为三个层次，分别为分配区 arena、堆 heap 和内存块 chunk。

A.Arena：堆内存最上层即为分配区 arena。

分配区 arena 分为主分配区（main arena）和线程分配区（thread arena)。分配区 arena 对应的数据结构为 malloc_state，即 Arena Header，每个 thread 只含有一个 Arena Header。Arena Header 包含 bins 的信息、top chunk 以及最后一个 remainder chunk。

程序的 Arena 数量是受到限制的，在线程数小于等于 Arena 限制数量时，glibc malloc 会分别为每个用户线程创建一个新的 thread arena。此时，各个线程与 arena 是 一一对应的。但是，当线程总数已达 Arena 上限而又有新的用户线程调用 malloc 的 时候，就无法再为该线程分配新的 arena 了，那么就需要重复使用已经分配好的若干 个 arena 中的一个。选择线程进行哪个 arena 重复利用：

glibc malloc 循环遍历所有可用的 arenas,在遍历的过程中，它会尝试上锁该 arena。如果成功 lock(该 arena 当前对应的线程并未使用堆内存则表示可以上锁)，比 如将 main arena 成功上锁，那么就将上锁线程的 arena 返回给用户，即表示该上锁 arena 被待分配的新线程共享使用。

如果没能找到可用的 arena,那么就将待分配的新线程的 malloc 操作阻塞，直到 有可用的 arena 为止。

如果新线程再次调用 malloc 的话，glibc malloc 就会先尝试使用最近访问的 arena,即上锁的 arena□如果此时上锁的 arena 可用的话，就直接使用，否则就将新 线程阻塞，直到上锁的 arena 再次可用为止。

B.Heap：每个分配区下最少有一个堆。

堆区对应的数据结构 heap_info,即 Heap Header，因为一个 thread arena（注意：不包含 main thread）可以包含多个 heaps，所以为了便于管理，就给每个 heap 分配一个 heap header。在当前 heap 不够用的时候，malloc 会通过系统调用 mmap 申请新的堆空间，新的堆空间会被添加到当前 thread arena 中，便于管理。

C.Chunk:每次 glibc 分配的内存块称为 chunk。

对应的数据结构 malloc_chunk: 即 Chunk Header，一个 heap 被分为多个 chunk。可以将堆分成三类：allocated chunk (已分配的)， free chunk (空闲的)，top chunk (顶部 chunk) □

从本质上来说，所有类型的 chunk 都是内存中一块连续的区域，只是通过该区域中特定位置的某些标识符加以区分。为了简便，我们先将这 3 类 chunk 简化为 2 类：allocated chunk 以及 free chunk,前者表示已经分配给用户使用的 chunk,后者表示未使用的 chunko 无论是何种堆内存管理器，其完成的核心目的都是能够高效地分配和回收内存块(chunk)。

（3）各阶段堆内存分布

① 在主线程调用 malloc 之前：

程序进程中是没有堆数据段的，并且在创建在创建线程前，也是没有线程堆栈的。

② 在主线程调用 malloc 之后：

系统给程序分配了堆栈，堆空间称为 arena,因为是主线程分配的，所以叫做 main arena(每个 arena 中含有多个 chunk,这些 chunk 以链表的形式加以组织)。当 main arena 中有过多空闲内 存的时候,也会通过减小 program break location 的方式来缩小 main arena 的大小。

③ 在主线程调用 free 之后：

从内存布局可以看出程序的堆空间并没有被释放掉，原来调用 free 函数释放已经分配了的空间并非直接“返还”给系统，而是由 glibc 的 malloc 库函数加以管理。 它会将释放的 chunk 添加到 main arenas 的 bin（这是一种用于存储同类型 free chunk 的双链表数据结构）中。在这里，记录空闲空间的 freelist 数据结构称之为 bins。 之后当用户再次调用 malloc 申请堆空间的时候，glibc malloc 会先尝试从 bins 中 找到一个满足要求的 chunk,如果没有才会向操作系统申请新的堆空间。

④ 在 thread 1 调用 malloc 之前：

thread 1 中并没有堆数据段，但是此时 thread 1 自己的栈空间已经分配完毕。

⑤ 在 thread 1 调用 malloc 之后：

threadl 的堆数据段已经分配完毕了，只有可读写的空间才是 threadl 的堆空间，即 threadl arenao

⑥ 在 threadl 调用 free 之后：

与 main thread 的原理相似，程序的堆空间并没有被释放掉，原来调用 free 函数释放已经分配了的空间并非直接“返还”给系统，而是由 malloc 库函数加以管理。 之后当用户再次调用 malloc 申请堆空间的时候，glibc malloc 会先尝试从 bins 中 找到一个满足要求的 chunk,如果没有才会向操作系统申请新的堆空间。

（4）Linux 和 Windows 内存管理的区别

A.Linux 系统：

Linux 内存管理包括 5 个内存管理器：物理内存管理器，内核内存管理器，虚拟 内存管理器，内核虚拟内存管理器，用户空间内存管理器。物理内存管理的第一个层次就是介质的管理。内核内存管理除了对内存整页的使用，有些时候，内核也需要像用户程序使用 malloc 一样，分配一块任意大小的空间。这个功能是由 slab 系统来实现的。

虚拟内存管理器主要使用了请求分页机制和交换机制。当进程运行时，不必把整个进程的映像都放在内存中，而只需要在内存中保留当前用到的那一部分页面。当进程访问到某些尚未存在内存的页面时，就由核心把这些页面装入内存。

用户内存管理中，malloc 是 libc 的库函数，用户程序一般通过它（或类似函数）来分配内存空间。libc 对内存的分配有两种途径，一是调整堆的大小，二是 mmap 一个新的虚拟内存区域（堆也是一个 vma）。

内存交换机制。当系统中存在内存不足时，Linux 内存管理子系统就需要释放一些内存页，从而增加空闲内存页的数量。此任务由内核的交换守护进程 kswapd 完成。Kswapd 有自己的进程控制块 task_struct 结构，它与其他进程一样受内核调度，但是没有自己独立的地址空间，只使用系统空间，所以把它称为线程。它的任务就是保证系统有足够的空闲内存页。

Linux 优先使用物理内存，当物理内存还有空闲时，linux 是不会释放内存的，即时占用内存的程序已经被关闭了（这部分内存就用来做缓存了）。也就是说，即时你 有很大的内存，用过一段时间后，也会被占满。这样做的好处是，启动那些刚开启过 的程序、或是读取刚存取过得数据会比较快，对于服务器很有好处。

B.Windows 系统：

windows 内存管理方式主要分为：页式管理，段式管理，段页式管理。

页式管理的基本原理是将各进程的虚拟空间划分为若干个长度相等的页；页式管

理把内存空间按照页的大小划分成片或者页面，然后把页式虚拟地址与内存地址建立一一对应的页表；并用相应的硬件地址变换机构来解决离散地址变换问题。页式管理采用请求调页或预调页技术来实现内外存存储器的统一管理。其优点是没有外碎片，每个内碎片不超过页的大小。缺点是，程序全部装入内存，要求有相应的硬件 支持。例如地址变换机构缺页中断的产生和选择淘汰页面等都要求有相应的硬件支 持。这增加了机器成本，增加了系统开销。

段式管理的基本思想是把程序按照内容或过程函数关系分段，每段都有自己的名字。一个用户作业或进程所包括的段对应一个二维线形虚拟空间，也就是一个二维虚拟存储器。段式管理程序以段为单位分配内存，然后通过地址映射机构把段式虚拟地址转换为实际内存物理地址。其优点是可以分别编写和编译，可以针对不同类型的段采用不同的保护，可以按段为单位来进行共享，包括通过动态链接进行代码共享。缺点是会产生碎片。

段页式管理：为了实现段页式管理，系统必须为每个作业或进程建立一张段表以管理内存分配与释放、缺段处理等。另外由于一个段又被划分成了若干个页。每个 段必须建立一张页表以把段中的虚页变换成内存中的实际页面。显然与页式管理时 相同，页表中也要有相应的实现缺页中断处理和页面保护等功能的表项。段页式管 理的段式管理与页式管理方案结合而成的所以具有他们两者的优点。但反过来说， 由于管理软件的增加，复杂性和开销也就随之增加了。另外需要的硬件以及占用的 内存也有所增加。使得速度降下来。

windows 则总是给内存留下一定的空闲空间，即时内存有空闲也会让程序使用一些虚拟内存，这样做的好处是，启动新的程序比较快，直接分给它些空闲内存就可 以了，而 **linux** 下呢？由于内存经常处于全部被使用的状态，则要先清理出一块内存，再分配给新的程序使用，因此，新程序的启动会慢一些.

总之，**Linux** 和 **Windows** 的堆内存管理主要有以下区别：

1. 管理实体：**Linux** 的堆内存管理是由 **C** 库的函数来实现的，而 **Windows** 的堆内存管理是由 **Heap Manager** 来实现的。

2. 分配算法：**Linux** 的堆内存管理采用基于分配器（如 **ptmalloc**）的方式进行内存分配和释放，而 **Windows** 的堆内存管理采用了多种分配算法。

3. 接口函数：**Linux** 提供的堆管理函数主要有 **malloc**、**free** 等，而 **Windows** 提供了一些专门用于堆内存管理的 **API** 函数，如 **HeapAlloc**、**HeapFree** 等。

4. 功能差异：**Windows** 的 **Heap Manager** 提供了更丰富的功能，如堆的大小调整堆的安全性设置、堆的共享等。而 **Linux** 的堆内存管理相对简单，功能比较有限。