OS Case Study
Arch Linux (Linux 5.18)

Liao Junxuan

2022 年 6 月 5 日

目录

1 历史

1.1 Linux 内核的历史

Linux 内核最初是 1991 年，当时的赫尔辛基大学本科生 Linus Torvalds 出于兴趣为自己的 386 计算机编写的，开发使用的操作系统是教学用的 Minix、编译器为 GNU 的 GCC. 当时的背景是这样的：自由软件运动的 GNU 项目运转了近十年，且取得了很大的进展，GCC 作为自由的编译器已经相当成功，其他用户态的工具都比较完善，但是内核项目仍处于早期阶段. 在此之前，Unix 于 1970 年发布，它的成功使得学界和商业界都乐于借鉴其模式和思想. 因此作为一本操作系统教材的配套实现，Minix 部分使用了 Unix 的理念，而 GNU 也把开发出与 Unix 兼容的自由操作系统为目标. Linux 0.1 版本发布后，世界上许多开发者对 Linus 的操作系统很感兴趣，他们加入了 Linux 的内核的开发，并将一些 GNU 系统的组件移植到 Linux 内核，最终使之成为 GNU 系统事实上的内核.

1992 年 Linus 将 Linux 0.99 在 GNU General Public License 下发布，Linux 正式成为自由软件.

2000 年左右，在以著名的论文 The Cathedral and the Bazaar 为代表的开源运动的影响下，Linux 作为开源软件的开发模式开始得到许多公司的重视. 许多硬件和软件公司在这时都开始为 Linux 提供支持，并加入 Linux 内核的开发.

如今（2020s）绝大多数互联网服务器运行 Linux 发行版 [18]，Linux 内核也广泛用于嵌入式设备和智能手机. 作为成熟的可移植的自由类 Unix 内核，Linux 既成为了开源运动的成功范例又为自由软件运动作出了突出的贡献.

1.2 发行版 Arch Linux

Arch Linux 是程序员兼吉他手 Judd Vinet 在 2002 年创建的一个滚动发行的 Linux 发行版. 其主要特点在于拥有可以自动管理依赖的包管理器 pacman 和受 FreeBSD 启发的包构建系统 [14]. 由于 Arch Linux 上构建和分享软件包非常简单，Arch Linux 用户可以使用的软件包非常丰富. 其简易

性和以用户为中心 [5] 的优点使其成为滚动发行模式的代表和最受桌面用户欢迎的 Linux 发行版之一.

Arch Linux 用户可以选择和配置自己的 Linux 内核，一般使用的是最新的 stable 分支上的普通的 Linux 内核.

Note

由于发行版的内核可以自己配置，发行版的区别对于本案例分析的影响不大，后文的分析将不太涉及发行版之间的区别.

2 Linux 内核的架构

Linux 是宏内核，即整个内核是一个共用地址空间的二进制程序. 宏内核架构是相对于微内核而言的. 微内核架构基于消息传递，作为内核的二进制程序只有最基本的功能，其他的不同组件处于不同的地址空间，它们之间通过消息传递的方式来进行交互. Linux 内核采用宏内核的架构主要是出于性能方面的考虑.[24] 同一地址空间的各个部分可以直接相互调用，没有消息传递的开销.

宏内核的架构不代表 Linux 内核不能采用模块化的设计，这得益于动态链接技术. Linux 内核由常驻内存的内核镜像 `vmlinux` 和动态加载的各种模块构成. 使用运行时加载的模块的好处有很多. 首先，这些模块在加载后就处于核心态，可以调用内核中的任何代码，访问硬件也没有限制，这可以使内核的设计者适度地划分各个功能之间的界限——避免功能之间耦合度过高的同时也不必在不同部分之间使用复杂的交互方法. 其次，动态加载模块可以在操作系统运行时更改内核，这对内核的开发有好处，因为不用编译整个内核并重启系统. 这也对节约资源有好处，因为可以只按需要加载内核模块，减少内存占用.

小问题 2.1：为什么叫“镜像”？

`vmlinux` 被称为内核“镜像”（kernel image），是因为这个文件在加载操作系统时被用来在内存中创建内核的副本. 详见 Image 一词的含义 [13].

要了解 Linux 如何实现模块化的宏内核架构，我们在案例分析中重点介绍两方面的内容：内核模块是如何构建的、内核模块是如何加载的.

2.1 内核构建系统

读源码 2.1：Kernel Build System

内核的构建系统有关的文档在 Documentation/kbuild 目录.

可以参考顶层目录的Makefile，各个目录下的Makefile、Kbuild. 另外，script目录下有用来链接 vmlinux和有关构建模块的脚本.

小问题 2.2：如何找到某个子系统的有关文件和信息？

Maintainer 文件中列出了 Linux 内核各个子系统的维护者，同时它还包括有关项目的网页和涉及的文件等有用的信息.

Linux 内核的构建系统是一套基于 GNU Make 的递归式的构建系统，包括各个目录下的 Makefile 和为这些 Makefile 提供基础设施支持的其他文件，通常称为 kbuild. [17]

整个内核构建系统的目标主要就是内核镜像 vmlinux 和可加载的模块. 各个 Kbuild Makefile 以向 obj-m 和 obj-y 变量增加内容的方式列出本目录应该构建的目标，其中 obj-m 变量中定义的目标会被构建成内核模块——后缀名为 .ko 的一种 ELF 文件，而 obj-y 变量中定义的目标会被 $(AR)[1] 合并到一个名为 built-in.a 的库中，最后被链接进 vmlinux. 顶层目录的 Makefile 递归地构建子目录.

有趣的是，内核的各个部分通常既可以编译到内核镜像中，也可以编译成可加载的内核模块，这是由目标是加入到 obj-m 还是 obj-y 来决定的. 例如 Listing 1所示，当 CONFIG_BTRFS_FS的值为 y（Yes）时，BTRFS 文件系统驱动——目标 btrfs.o将作为内核内置的一部分编译；而CONFIG_BTRFS_FS 的值为 m（Module）时，它将编译成可加载的内核模块.

```
1 # fs/btrfs/Makefile
2 obj-$(CONFIG_BTRFS_FS) := btrfs.o
```

Listing 1: 可配置的编译目标

[1]一般就是 GNU Binutils 里面的 ar，用于把多个 .o 目标文件归档到一个文件中.

类似 CONFIG_BTRFS_FS 的变量的定义来源于与内核构建系统一起工作的配置系统——Kconfig. 开发者往往会通过 make *config 来调用某种目录界面对将要构建的内核进行配置，选择要启用的功能并配置其中的参数. 这一步骤生成 Kconfig 语言的内核配置文件，为 Kbuild 提供各种变量，对构建出来的内核造成影响.

以 Arch Linux 为例，其官方的内核配置文件中有关上述 BTRFS 模块的一行为：CONFIG_BTRFS_FS=m. 这意味着 Arch Linux 的 Linux 包所使用的内核中，BTRFS 的支持是作为可加载的模块编译的.

内核构建系统的设计还允许在内核的代码树外构建内核模块，只要安装了对应版本的头文件和构建脚本，Linux 的使用者甚至可以在自己计算机的任何目录编写、构建和安装自己的内核模块. 同样以 Arch Linux 为例，要安装的是 linux-headers. 编译自己的模块时，首先编写 Kbuild 文件，声明要编译的目标. 然后利用 GNU Make 构建前更改工作目录的功能，使用安装在系统中的内核构建系统来编译自己的模块，如 Listing 2 所示. 其中 /lib/modules/`uname -r`/build 会被展开成当前运行内核的版本的构建目录，这个目录正是 linux-headers 包安装的.

```
1 make -C /lib/modules/`uname -r`/build M=$PWD
```

Listing 2: 为本机的内核构建模块

2.2 内核模块的加载

读源码 2.2：加载模块

模块在内核中的表示见 include/linux/module.h.

加载模块的函数为 kernel/module.c 中的 load_module.

init_module 是加载模块的系统调用，它把模块的内容从用户空间拷贝到内核，然后调用load_module.

在2.1中，我们了解到内核模块是 ELF 格式的二进制文件，加载模块就是要将这个文件加载到内核的地址空间中，并做一系列准备工作，使该模块进入正常的工作状态、与内核的其他部分可以互相调用.

加载模块的步骤主要是：

1. 读 ELF header，获取各个段的信息，如长度.
2. 确定内存布局，为模块分配内存.
3. 把各个段的内容转移到刚才分配的内存中.
4. 解析符号，把引用的符号重新变为直接指向变量的指针.
5. 重定位，调整代码中的地址使其与改变后的地址对应.
6. 传递参数，调用模块初始化的代码.

动态的符号解析是模块之间互操作性的基础. 内核中不是可加载模块的部分遵循一般的 C 语言链接的规则，而如果要暴露出可供模块使用的符号，就要使用 `export_symbol*` 这一类的宏来把该符号会被加入到几张特殊的符号表中. [26] 加载模块时，`simplify_symbols` 函数会在这些表中找到被导出的符号. 如果要使用的符号来源于另一个模块，则这两个模块之间存在依赖关系，内核会在每个模块的内存表示中记录其依赖的所有模块和依赖它的所有模块，模块之间的依赖关系可以帮助确定卸载模块的顺序等.

内核是如何知道要加载哪些模块的呢? 模块的加载一般是由用户态的程序发起的. Arch Linux 的启动过程 [4] 中，首先是 bootloader 加载 vmlinux 镜像，这时内核的根文件系统是内存中的逻辑上的文件系统，称为 initramfs（initial RAM file system）. 内核随后把磁盘上的 initramfs 镜像解压到该文件系统中，这时就进入了“早期用户空间”阶段. Arch Linux 的 init 系统采用的是 systemd，systemd 此时会启动一个名为 `systemd-modules-load` 的服务 [2]，该服务启动一个同名的程序，来根据用户的配置文件加载内核启动早期所需要的模块. 当真正的根文件系统被挂载后，内核模块的自动加载就由 systemd 的 udev 系统负责，udev 根据设备的变化、事件的发生来自动加载和卸载所需要的内核模块，以此为设备提供驱动程序和增改内核的功能.

[2] 位于`/usr/lib/systemd/system/systemd-modules-load.service`

Note

内核模块也可以手动管理. kmod 是 Linux 用户态的模块管理程序和库. 常用的命令有:

- 加载模块: `modprobe 模块名`
- 卸载模块: `rmmod 模块名`
- 显示依赖: `modprobe --show-depends 模块名`

3 进程管理

在正式介绍 Linux 内核中的进程和线程前，我们先利用课程中了解到的进程和线程的概念尝试直接分析内核中进程的表示方式——进程控制块（PCB），建立整体的认识. 最后，我们试着弄懂 Linux 的进程调度策略，并把重点放在最常用的 Completely Fair Scheduler（CFS）上.

3.1 进程控制块

`struct task_struct` 是 Linux 内核的进程控制块（PCB）. 它存储进程的标识符、进程的状态、指向存储进程上下文的数据结构的指针、调度所需的信息以及与进程相关的资源和指向其他 `task_struct` 的指针等. 所有与进程有关的操作都会直接或间接地修改进程控制块来达到目的. 下面，我们介绍管理进程所需要的多种信息，并介绍它们是如何在 `task_struct` 中表示的.

读源码 3.1：进程控制块

`struct task_struct` 是在include/linux/sched.h中定义的.

3.1.1 进程的状态

首先是进程的状态，一个进程可能处于这些状态：

- `TASK_NEW`. 当一个进程刚创建时，它的状态就被设置为`TASK_NEW`，表示这个进程已经被创建，但是还没有开始运行.
- `TASK_DEAD`. 进程已经结束.
- `TASK_RUNNING`. 进程正在运行，即该进程在正在运行的进程的队列里.
- `TASK_INTERRUPTIBLE` 或者 `TASK_UNINTERRUPTIBLE`. 处于这两个状态的进程正在等待，但是只有 `TASK_INTERRUPTIBLE` 状态的进程才会被信号唤醒.

- TASK_NOLOAD. 类似 TASK_UNINTERUPTIBLE 但是在统计数据中不被计入负载. [3]
- TASK_WAKING. 进程已经被要求唤醒，但是还没有进入正在运行的进程队列. [4]
- TASK_WAKEKILL. 该进程收到 SIGKILL 信号时会被唤醒.
- TASK_TRACED. 调试器暂停该进程来追踪它的运行状态.

这些状态被编码成掩码，读写进程的状态时，需要用这些掩码操作 task_struct 的 __state 域. 比如识别一个进程是否处于某状态，就要用它的 __state 和这个状态的掩码做与操作，若结果为 0 则不处于这个状态. 这些状态中有的状态可以组合成新的掩码，方便使用.

3.1.2 进程的 CPU 上下文

进程控制块还需要存储有关进程上下文的信息，例如有关栈和堆的信息和 CPU 内部寄存器的值. 这一部分信息既和内核的内存管理有关又依赖于具体的硬件架构，是实现进程调度中挂起和重启进程所需要的数据结构.

读源码 3.2：架构相关代码

为了提高可移植性，Linux 内核的代码区分不依赖于具体硬件架构的代码和针对特定架构的代码. 架构相关的代码全部在 arch 目录下. 例如 x86 和 x86_64 的代码位于 arch/x86.

显然，所有的汇编代码都应该放在该目录. 内存管理和进程管理所需的某些功能也依赖于 CPU 的特定功能，需要根据硬件功能定义数据结构和执行具体的指令，这些定义和实现也位于 arch 下具体架构的目录下. 不同的架构的代码尽量暴露出相同的接口，供架构无关代码使用.

[3] 见 https://lore.kernel.org/lkml/alpine.LFD.2.11.1505112154420.1749@ja.home.ssi.bg/T/

[4] 见 https://lore.kernel.org/lkml/tip-e9c8431185d6c406887190519f6dbdd112641686@git.kernel.org/

某一架构上的具体实现的文档可以在 Document/arch.rst 中的列表找到. 例如Document/x86/kernel-stacks.rst介绍了 x86_64 CPU 上内核为每一个进程维护的若干个栈.

task_struct的定义中，有一个 void *stack; 域. 这不是该进程的用户态的栈的起始地址，而是该进程的**内核栈**的起始地址，每次该进程通过系统调用从用户态进入内核态时，内核中的代码都在这个内核栈上执行.

tast_struct的最后一个成员是 struct thread_struct thread;，进程在 CPU 上的运行状态存放在这个架构相关的结构体中. x86_64 上该结构体的大小可以变化，因此它必须作为tast_struct的最后一个成员. 这个 struct thread_struct 的成员直接或间接地保存该进程的上下文.

我们在微机原理中学过 x86_32 架构对于上下文切换有专门的硬件支持. Global Descriptor Table（GDT）中有专门的描述符指向 Task State Segment（TSS）. TSS 可以用来存储所有的 x86 寄存器，在需要时自动进行硬件上下文切换，把寄存器的值存储在旧的 TSS 中，然后把新 TSS 的内容加载到寄存器中. x86_64 取消了硬件上下文切换的功能，但即使是在 32 位 x86 机器上，Linux 内核也不是用 TSS 来进行硬件切换，因为考虑到其它架构几乎都不支持这一机制，使用软件实现上下文切换可移植性更好.

Linux 内核在上下文切换时，寄存器的值实际上是存在进程的内核栈上的. 进程在切换之前，一定正在该进程的内核栈上执行内核态的代码，这时用户程序的寄存器已经在进入内核态后保存下来了，要保存运行在内核态时的寄存器，只需要 pushq 所有的*被调用者保存*寄存器即可. 这是因为在进入当前栈帧前后，调用者一定已经保存了其所需的*调用者保存*寄存器. 返回地址也在栈帧中有记录. 既然这个栈帧就记录着当前进程的上下文，PCB 只需保存这个栈帧的地址即可. 因此 thread_struct 有一个 unsigned long sp; 域，存放的就是切换前的寄存器 sp 的值，即这个栈帧的地址.

除了整数运算的寄存器，浮点和向量运算的寄存器也需要保存，但是由于浮点和向量运算不像整数运算那样常见，且保存这些状态有一定开销，之前的 Linux 内核只在这些运算真正执行过时才保存对应的寄存器. 这种“懒惰”的保存方式是通过 CPU 的硬件支持完成的：每次上下文切换时，CPU

的 TS 标志位就被置为 1；每次浮点或向量运算指令执行时，若 TS 标志位为 1，则会报 “Device Not Available” 异常. 内核在处理这个异常时会记录下来：该进程使用了浮点或向量运算. 再次切换进程时，若发现使用了浮点或向量运算，则保存相应的状态.[7] 而现代的 CPU 引入了特定的指令来保存 FPU 状态，减小了保存 FPU 状态的开销，而且程序越来越依赖向量指令来进行拷贝数据等操作，所以 Linux 在 2016 年取消了 “懒惰” 的保存方式，[5] 每次进程切换都保存 FPU 的状态. 除了进程切换，另外一个需要保存和恢复 FPU 状态的时机是内核代码在用户上下文中使用浮点或向量运算的前后，这也是按需进行的，即返回用户态时，若改变了 FPU 状态才加载用户的 FPU 状态.

FPU 的状态保存在 `thread_struct` 的 `struct fpu fpu;`域. 这个数据结构是变长的,这也是 `struct thread_struct thread;` 要放在 `task_struct` 最后的原因.

读源码 3.3：`thread_struct`

`thread_struct` 是架构相关的. x86 的 `thread_struct` 的定义在 arch/x86/include/asm/processor.h 中.

进程调度总是通过调用一个名为 `schedule()` （kernel/sched/core.c）的函数来实现. 真正完成调度的函数为 `__schedule()`（在同一个文件），其上方的注释说明了所有会调用它的情况.

而最终完成进程切换的是一个名为 `switch_to(prev, next, last)` 的宏 [a]. 其关键的操作就是切换进程的内核栈（见图 1）：

1. 将被调用者保存寄存器压入栈；
2. 把 rsp 寄存器存入原来进程的 `thread->sp`；
3. 把新进程的 `thread->sp` 放入 rsp 寄存器；
4. 最后从新的栈中弹出所有调用者保存寄存器.

[5] 见 https://lore.kernel.org/lkml/e3b2baadcd19bf8abcd3bcd60d19e8e50e75f63a.1453510332.git.luto@kernel.org/

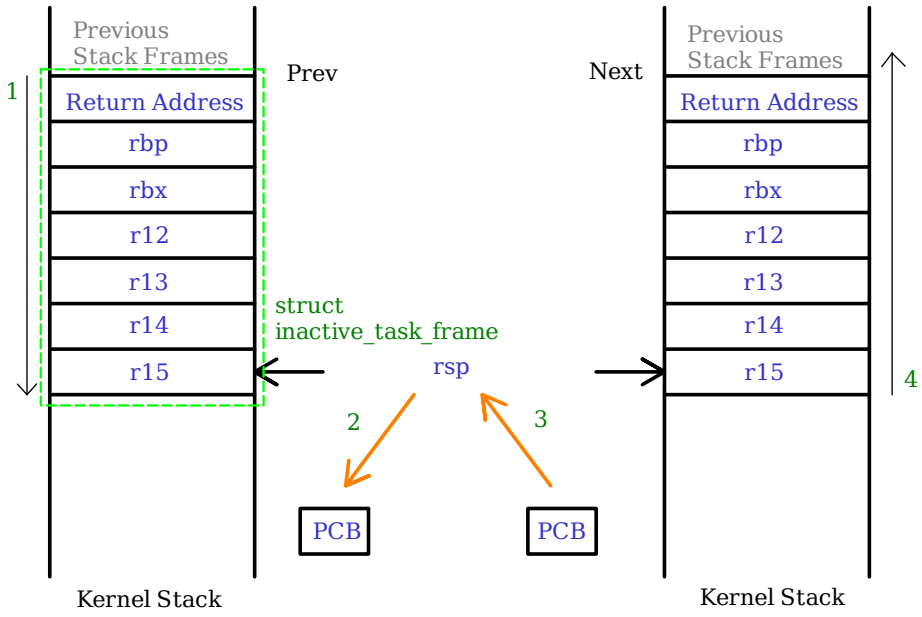

图 1: 进程切换中的内核栈切换

[a]x86 上的定义在 `arch/x86/include/asm/switch_to.h`. 64 位机器的实现，汇编部分在 `arch/x86/entry/entry_64.S`；C 语言部分在 `arch/x86/kernel/process_64.c`

小问题 3.1：`struct thread_info` 与 PCB 是什么关系呢？

`struct task_struct`是进程控制块，而 `struct thread_info` 存放的也是进程信息，只是这些信息更接近底层硬件，且需要能在汇编代码中直接使用. `struct thread_info` 的大小被控制在一个 cache 行内，这是由于这些底层信息需要被频繁查看，cache miss 会造成较大影响. 为了尽量利用空间，所有信息都按位编码在其中. 例如，上文提到的“内核使用了浮点或向量运算而需要加载原来的 FPU 状态”，就存储在 `thread_info` 的第 `TIF_NEED_FPU_LOAD` [a] 位.

`struct thread_info` 在更早的 Linux 版本中位于进程的内核栈的起始位置 [7]，可能造成一些疑惑. 现在的 x86 Linux 内核默认把 `struct thread_info` 放入 `struct task_struct`，与 PCB 的其他成员处于类似的地位. [b] [c]

[a] `arch/x86/include/asm/thread_info.h` 中定义的宏.

[b] 见 https://lore.kernel.org/all/cover.1473801993.git.luto@kernel.org/

[c] 见 https://lore.kernel.org/all/a50eab40abeaec9cb9a9e3cbdeafd32190206654.1473801993.git.luto@kernel.org

3.2 标识符、进程控制块的链表

识别进程的方式有很多，首先，指向`task_struct` 的指针就可以唯一地确定一个进程. 每一个进程还有编号，记录在 `pid_t pid;` 域中. 每一个进程按照创建顺序编号，每一个进程的这个 `pid` 域都不相同. 由于 3.2.2 中会介绍的原因，操作系统的使用者会期望某些调度单元的 PID 相同. 这是通过 `task_struct` 的 `pid_t tgid` （thread group id）域来实现的. 不同的 `task_struct` 的 `pid` 不能相同，`tgid` 却可能相同. 用户所得到的 PID 实际上是 `task_struct` 中的 `tgid` 而不是 `pid`.

进程的产生和销毁是动态的，这意味着进程控制块也需要动态分配，所有分配出来的进程控制块的地址都要通过某种方式留存起来，以供访问. Linux 内核记录多个进程控制块的方式之一是建立包含进程控制块的双向链表. 链表有多种实现，节点互相链接，每个节点指向一个表内的实体的方式为“非侵入式（non-intrusive）”的链表，实体中包含所需的链表指针的链表为“侵入式”链表. 如果比较访问下一个节点数据的性能，非侵入式链表比侵入式链表性能更好，因为非侵入式的链表少一次解引用的访存操作. 另外，寻找链表首尾元素和双向遍历的功能也比较重要，因此内核使用的链表是侵入式的双向链表.

如图 2所示，`struct list_head` 作为链表的节点，有 `prev` 和 `next` 两个指针，分别指向前一个节点和后一个节点. 链表中的元素通过成员中的

`struct list_head` 就可以互相链接.

除了构成全系统的进程表的 `struct list_head tasks`，`task_struct` 中还包括其他的 `list_head`，用于记录与该进程相关的其他进程. 比如，记录树形的父子进程的关系（见3.2.2），就是通过 `struct list_head children` 和 `struct list_head sibling`这两个链表实现的. 显然，类似于树的 Left-most child, right sibling 表示法 [23]，`children` 指向的就应该是某个子进程的 `sibling` 成员，而不是 `children` 成员. 以创建进程为例，需要把当前进程的 `sibling` 成员，插入到父进程的 `children` 所指的链表尾部：

```
/*
 * kernel/fork.c
 * copy_process()
 */
list_add_tail(&p->sibling, &p->real_parent->children);
```

进程控制块还需要通过 PID 快速获取，这是通过散列表实现的，这里不详细介绍.

3.2.1 其他信息

进程控制块中还包括这些方面的相关信息：内存管理、打开的文件、文件系统、调度器、统计数据等.

- 内存管理：进程的虚拟地址空间信息，**页表**（将在 4 中介绍）
- 文件：内核需要维护进程**打开的文件的表**，以记录该进程读写文件的状态.
- 文件系统：记录进程的**工作目录**和，操作文件所用的**权限**等.
- 调度：进程所属的**调度类型**，进程的**优先级**等. （将在 3.3 中介绍）
- 异常处理/进程通信：进程等待接收的**信号**，信号的屏蔽情况，**信号处理程序**的地址.

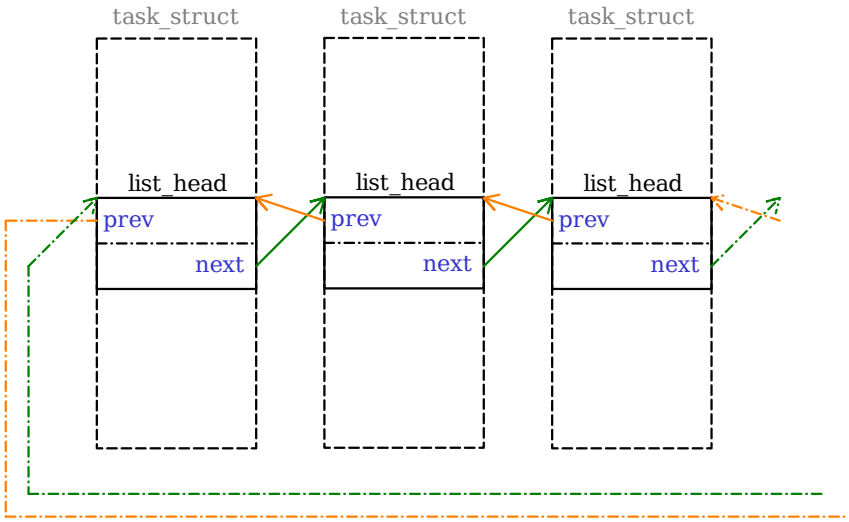
task_struct
task_struct
task_struct
list_head
prev
next
list_head
prev
next
list_head
prev
next

图 2: 侵入式双向链表

3.2.2 进程和线程的创建

小问题 3.2：内核中线程和进程的区别是什么？

前文我们一直以“进程”来描述 Linux 内核调度的单位. 即使代码中出现了 `thread` 的字样，我们仍用“进程”指代. 实际上，Linux 一直使用 task 来称呼调度的单位，只是在和 CPU 线程有关的地方使用了 thread 的说法. 下文将涉及 Linux 内核中线程和进程的区别.

内核中第一个进程是在启动时创建的，PID 为 0，名为空闲进程（the idle thread），内核中有一个 `struct task_struct init_task` 作为静态的全局变量，在启动时各个成员被赋上相对固定的初始值. 在内核的启动过程中，空闲进程进行一定的初始化，然后就创建 PID 为 1 的进程（the `init` process），进一步初始化，最后载入 init 程序 [7]（Arch Linux 中即为 systemd）.

除了第一个进程是静态分配，从零开始初始化的，其余进程的创建都是通过创建已有进程的拷贝来完成的. 这种模式可以追溯到 Unix 之前的系统，但是 Unix 系统采用这种 `fork` 的进程创建方式主要是因为实现上较为简单.[20] Linux 跟随 Unix 的设计思路，提供类似的接口，但又有所不同.

用于复制进程从而创建进程的多种系统调用最终都会使用内核中的同一段代码，只是调用时的参数不同，父进程和子进程共享的内容就不同. 可以共享的内容例如：

- 父进程（是否使用与被复制的进程相同的父进程）；
- 虚拟内存空间；
- 打开的文件；
- 文件系统信息（工作目录、根目录等）；
- 进程组信息（是否在一个组，tgid）；
- 信号处理程序，信号屏蔽状态；
- 其他资源如 I/O.

有些资源如栈是不能被共享的，每一个进程需要在自己的栈上运行. Linux 内核有在进程间共享资源的能力，是因为进程控制块中的某些信息并不直接存储在进程控制块中，而是单独存在，进程控制块中只记录指针.[24]

按照这种方式，只要使用不冲突的参数组合，Linux 内核就可以创建多种进程，有些可以被看作“进程”，有些可以可以被看作“线程”，还有介于“进程”和“线程”之间的，Linux 全部把他们当作调度的单位. 具体的线程模型（不同线程直接共享哪些数据，如何进行控制等）一般是由 POSIX 标准 [12] 定义的，实现 POSIX Thread （pthread）的程序库提供创建线程的函数，用合适的参数通过系统调用创建符合标准的线程.

读源码 3.4：进程创建

`struct task_struct init_task` 的定义在 `init/init_task.c` 中.

创建进程的函数主要在 `kernel/fork.c`，其调用关系为：

- `kernel_clone`
  - `copy_process`
    - `copy_thread`
      （架构相关，在`arch/x86/kernel/process.c`）

3.3 进程调度

3.3.1 调度的时机

简单地说，调度就是在合适的时间选择一个合适的处于“运行”状态（见3.1.1）的进程在 CPU 上运行.

在3.1.2中，我们已经提到，调度的入口为 `schedule()` 函数，只有这样一些情况 [6] 下，`schedule()` 会被执行，一个新的运行状态的进程会被选中：

- 进程主动进入等待的状态，如使用锁或信号量的时候.
- 从内核态返回用户态（完成中断处理、系统调用）的某些路径上，会检查是否需要重新调度，如果需要则调用 `schedule()`.

[6] 见 `kernel/sched/core.c` 函数 `schedule()` 上方的注释.

后者是一种异步地调用调度算法的方式，主要用于实现抢占. 当某个进程被唤醒时，它会被加入到运行队列中，如果该进程的优先级比当前 CPU 上的进程的优先级高，则需要抢占 CPU. 内核并不立刻进行抢占（因为当前进程可能正在执行用户代码，并不处于可以切换的状态），而是在进程控制块中的线程信息里设置一个标志（`TIF_NEED_RESCHED`），当前进程下一次处于内核态并检查该标志时，即调用 `schedule()`，顺利完成进程切换，实现抢占. 另外一个会设置 `TIF_NEED_RESCHED` 时机是时间片用尽的时候，这时当前进程应该尽快让出 CPU，重新执行调度算法.

3.3.2 进程调度策略

Linux 内核的调度采用模块化的设计，从 2.6.23 版本引入新的调度算法 Completely Fair Scheduler（CFS）后 [16]，不同的调度算法被封装成了不同的 `struct schedule_class` [7]. 每个 scheduling class 中用函数指针的方式存储了对应算法的实现，并且进程控制块中存放了一个指向 `struct sched_class` 的指针. 调度时，与具体算法无关的程序就可以用同样的方式调用不同的算法实现.[8].

小问题 3.3：不同的 scheduling class 是如何共存的？

不同的 scheduling class 优先级不同，每一个 scheduling class 都维护自己的运行队列，内核调度时会参考该优先级，详见下文调度的过程.

从总体上看，调度的过程主要有三个部分.

1. 当需要寻找下一个进程时，按照优先级顺序依次尝试不同的 scheduling class，若较高优先级没有要运行的进程，则尝试更低优先级的.

2. 使用硬件时钟来给当前进程的调度器提供“时间概念”，以供调度器判断该进程是否已经用完了它能使用的时间. 时钟的来源是硬件的中断，而通知调度器的方法，是调用 `sched_class` 中的 `task_tick` 函数，告知调度器已经过去了一段固定的时间.

[7] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=20b8a59f2461e1be911dce2cfafefab9d22e4eee

3. 根据进程的状态的转换来改变运行队列. 若进程从等待状态被唤醒，需要把它加入到对应的 scheduling class 的队列中，并且检查该进程是否应该抢占 CPU，若进程从运行状态进入等待状态，需要把它从队列中移除.

读源码 3.5：进程调度过程、sched_class

Linux 调度器的文档在 Document/scheduler 目录.

与算法无关的调度过程在 kernel/sched/core.c. 处理时间流逝的是 scheduler_tick 函数,选择下一进程并切换上下文的是 schedule 函数，唤醒进程的是 try_to_wake_up 函数.

struct sched_class 的定义在 include/linux/sched.h. 与上文提到的调度过程三个部分有关的函数有：

- pick_next_task 选取下一个进程.
- task_tick 通知时间流逝.
- enqueue_task，dequeue_task，yield_task 调整运行队列.
  - check_preempt_curr 检查是否要抢占.

不同的 struct sched_class 优先级是在包含链接脚本宏的 include/asm-generic/vmlinux.lds.h 中定义的，地址上的顺序就是优先级的顺序.

Linux 内核的调度算法分为实时和普通两类. 进程可以被设置成不同的调度策略，存储在 task_struct 的 policy 字段中. POSIX 规定了应该实现的几种策略 [22]，Linux 中的实现是：

- 实时
  - SCHED_FIFO 先来先服务.
  - SCHED_RR 轮转.
  - SCHED_DEADLINE 最早 deadline 优先.

- 普通
  - SCHED_NORMAL 一般进程.
  - SCHED_BATCH 批处理任务，最好能在切换之前运行更长时间.
  - SCHED_IDLE 优先级特别低的进程.

调度策略和 scheduling class 并不是一一对应的，scheduling class 有 5 种，:

- **fair** 普通的分时调度
  （SCHED_NORMAL、SCHED_BATCH、SCHED_IDLE）
- **rt** 调度实时进程.
  （SCHED_FIFO、SCHED_RR）
- **dl** Earliest Deadline First（EDF）调度器.
  （SCHED_DEADLINE）
- 特殊的
  - **idle** 用来调出 idle 进程，总是可以找到下一个进程：idle 进程.
  - **stop** 在 SMP 机器上停止 CPU 运行，用来做负载均衡等.

它们的优先级从高到低一次为：stop，dl，rt，fair，idle. 这样的顺序可以满足各个调度算法的目的首先优先保证进程的 deadline 能够满足，其次实时任务比普通进程优先级更高，如果所有算法都找不到需要运行的进程，才应该运行 idle 进程.

Note

有关多 CPU 的调度机制在此不会详细说明. 简单地说，每一个 CPU 都有自己的运行队列. 如今越来越普遍的非统一内存访问架构（NUMA）的机器需要实现更复杂的机制.

3.3.3 Completely Fair Scheduler

Completely Fair Scheduler 是 Linux 的普通进程的调度器，文档里面是这样概括它的设计思想的：[8]

> CFS basically models an "ideal, precise multi-tasking CPU" on real hardware.

理想的多任务处理器上，所有任务都“公平”地运行，也就是一段时间内，每一个进程都运行相同多的时间. CFS 则想要模拟这种理想的处理器，要做到这一点，需要记录所有进程在 CPU 上运行的时间，并且利用这个时间来选择下一个调度的进程和决定切换的时机. **忽略**一些实现和功能上的细节来看，每一次选择下一个要运行的进程时，CFS 都会选择可运行的进程中“已经运行的时间”最短的进程，如果当前进程在运行一段时间后，已经不是运行时间最短的进程，新的运行时间最短的进程将会抢占 CPU. 这样做的结果是，所有可运行的进程在任意一段不算太小的时间内，运行的时间都是近似相等的.

小问题 3.4：CFS 如何比较运行时间？运行了一天的进程和刚刚启动的进程有区别吗？

如果 CFS 利用比较运行时间来选择进程，那么为什么新创建的进程和已经运行了很久的进程都能够有机会运行呢？这是因为比较的不是进程总共运行了多少时间，而是比较一小段时间内运行的时间长短. 详见下文.

CFS 为每一个进程维护一个名为 `vruntime` 的变量，这个变量记录进程在 CPU 上运行的时间，但是不完全是运行时间，而是调整后用于比较进程“公平”程度的虚拟时间. 内核把调整 `vruntime` 的操作称为 normalization，调整的目的是反映一段时间内进程运行的时间长短，忽略创建时间的差异.

刚刚创建的进程显然应该具有最小的虚拟时间，因为它从未运行，应该被选中作为下一个运行的进程. 而已有的可运行进程虽然已经运行了一段时间，但是仍应该和刚刚创建的进程处于“同一起跑线”上. 因此，新进程进入队列时，应该把它的 `vruntime` 设为当前队列中最小的 `vruntime`，这样

从 `vruntime` 的角度来看，所有进程就好像是同一时间开始运行的. 再考虑进程进入等待状态和恢复运行状态的过程. 如果不做修改，重新进入队列的时候，它的 `vruntime` 就会与其他进程相差一个较长时间 t，为了弥补这个差距，它将一直运行. 而这是错误的，因为在 t 时间内，它本来（在理想的多进程 CPU 上）就不应该运行，而应该等待，让它多运行 t 的时间是“不公平”的.

综上所述，normalization 的规则为：

- 出运行队列时，`vruntime -= min_vruntime`，
- 进入运行队列时，`vruntime += min_vruntime`.

这个规则还覆盖了上文没有提到的一个过程，即进程在不同运行队列之间的迁移. 只要遵循 normalization 的规则，`vruntime` 在运行队列中就是一个可以和其他进程比较的绝对量，在运行队列外就是记录比较结果的相对量.

Note

上述规则可以用一个跑步的例子说明. 假设一群朋友在操场上慢跑，约定好大家应该尽量用同样的配速跑步，达到相同的锻炼效果. 要同步配速，最简单的方法是所有人一起跑，如果有人稍微落后，就加快速度赶上. [a] 这样所有人大约都在跑道的同一区域.

如果新来一人想要加入跑步的队伍，那么他应该从队伍所在的地方开始跑，而不是从跑道的起点起跑. 如果队伍中有人想休息，离开跑道后应该回到队伍中原来的位置，而不是从起点继续跑.

这个例子中，每个人所跑的实际距离相当于进程的实际运行时间；整个队伍所跑的距离相当于 `min_vruntime`，这是一个单调递增的值；整个队伍跑的距离加上个人距离队伍中最后一个人的距离相当于 `vruntime`.

[a] CFS 中稍有不同的是，同一时间只能有一个进程在一个 CPU 上执行.

小问题 3.5：vruntime是单调递增的，会溢出吗？

vruntime的数据类型是 u64，单位是纳秒. 64 位无符号整数（即使是在 32 位机器上，也是 64 位）大约能表示的时间为：

$$2^{64}\text{ns} = 584.94\text{yr}$$

所以应该不用考虑 vruntime 溢出的问题.

读源码 3.6：CFS

CFS 的文档在Documentation/scheduler/sched-design-CFS.rst.

CFS 的 scheduling class 为 sched_fair_class，实现在 kernel/sched/fair.c. 关于 normalize 的说明可见函数 enqueue_entity 上方的注释.

既然进程等待的时间不会被补偿，那么作为“桌面”系统 [8] 的调度器的 CFS 有利于提高系统的交互性吗？CFS 的调度下，系统的交互性其实比较好. 因为需要交互的 I/O 密集型的程序，往往还没有执行很久，甚至还没有失去最小 vruntime 地位时就已经进入等待状态了. 再到下次可运行时，因为它们的 vruntime 较小，所以会被优先调度.[24] 所以 CFS 下交互式的程序延迟较低，交互性较好.

现在我们再来关注之前忽略掉的细节.

首先是调度的粒度. 如果是理想的 CPU，所有的进程同时执行，彼此之间的 vruntime 应该没有任何区别. 这样的调度粒度就是 0. 这意味着进程切换在不断进行，每秒钟都进行无穷次进程切换. 而我们在 3.1.2 中已经认识到，上下文切换是一个开销很大的操作. 过于频繁的切换显然不利于有效利用 CPU. 为了避免过度调度、降低调度的频率，应该设置一个合适的调度粒度.[8] CFS 中调度粒度可由 /proc/sys/kernel/sched_min_granularity_ns 配置. 它的含义是，当新的进程抢占当前进程时满足：

$$vruntime_{\text{当前进程}} - min\_vruntime = sched\_min\_granularity\_ns$$

因此这也是进程持续运行而不被抢占的最小时间.

如果用户或系统管理员不希望不同进程“公平”地分享 CPU 时间怎么办？这就需要在增加 `vruntime` 时做一些调整. CFS 调度的所有进程的优先级都是 0，而用来区分紧急程度的是 nice 值. nice 值越高，进程就对其他进程越“友好”，占用的 CPU 时间应该越少. CFS 利用增加 `vruntime` 的机会来区别对待不同 nice 值的进程. 每次加上的值实际上是真正的运行时间乘上与 nice 值相关的权重，nice 值高于 0 的进程增加的时间比实际时间多，nice 值低于 0 的进程增加的时间比实际时间少，这样就可以通过用 `nice`、`renice` 等方法改变进程的调度优先级.

CFS 的运行队列在功能上是根据 `vruntime` 排序的优先队列，实现上，需要频繁取最小元素，删除元素和插入元素，若要取得较低的复杂度，需要引入更复杂的数据结构. CFS 使用的是红黑树（Linux 里叫 rbtree），一种常用的自平衡的排序树. 并且它还在平衡树的时候存储最左边的节点，以达到可以直接访问队列中最小元素的效果.

读源码 3.7：`vruntime`的更新，红黑树

更新 `vruntime` 的代码在 `kernel/sched/fair.c` 的 `calc_delta_fair` 函数中.

Linux 的红黑树的实现在 `lib/rbtree.c` 和 `include/linux/rbtree.h`. 值得注意的是，搜索和插入节点的代码需要自己实现，因为 C 语言缺乏对泛型的支持，而使用回调函数又有很大开销[a].

[a]例如 C 语言的 `qsort` 就远不如支持泛型的 C++ 的 STL 中的 `sort`，因为无法内联特定实现的函数，还需要解引用函数指针.

4 内存管理

Linux 的内存管理可以分为内存分配和虚拟内存两个部分.[24] 因为 Linux 采用分页的方式管理内存，内存分配部分的主要任务就是分配物理页、释放物理页. 而虚拟内存部分利用分配出来的物理页来提供内存的抽象，实现缓存、共享和保护等功能.

4.1 物理页面的管理

4.1.1 物理内存模型

Linux 的内存管理是基于分页技术的，因此物理内存均被看作是页面的数组. 但是物理内存的组织形式又是架构相关的，而且一种架构可以有多种组织方式. Linux 针对不同的物理内存形式，用不同的物理内存模型来管理. 大部分连续的内存对应 FLATMEM模型，更复杂的内存组织形式对应 SPARSEMEM 模型.[19] x86-64 支持这两种形式，但是 FLATMEM 不支持非统一内存访问架构（NUMA）机器. Arch Linux 和绝大多数 x86-64 发行版都配置的是 SPARSEMEM 模型.

SPARSEMEM模型下，物理内存的配置可以很灵活. 物理内存被分段表示，每一个段（section）指向连续存放的一系列物理页面. 内核中管理物理页面的程序可以存储在动态分配的数组中，因此甚至可以支持运行时添加内存设备. 配置的灵活性却会给物理内存的访问增加复杂性，在物理内存不连续的条件下，物理页号（PFN）并不能用于直接访问真正的物理页. 物理页号到物理页的映射方式有两种，一种方式是把 section 的信息编码进 PFN 中，并且在 `struct page` 中也记录 section 的值. 而默认配置下，例如 Arch Linux 使用的是 VMEMMAP 方式. 这是指在虚拟内存中专门分配一个连续的称作 virtual memory map 的空间来存放页面信息. Virtual memory map 是页面信息 `struct page` 的一维数组，以 PFN 为元素下标，所以要想从 PFN 找到页面信息（包含页面的物理地址），只需要计算偏移访问数组即可. 逻辑上，这个数组内有所有的页面的信息，而且是连续的，其大小可与整个物理地址空间的大小相比. 但是，正是因为它是虚拟地址空间的连续数组，而虚拟的页面又可以按需分配，所以它并不会真正占据很多空间. 这种在物

理页面管理中也使用虚拟内存的想法非常能体现虚拟内存的灵活性.

读源码 4.1：struct page，内存模型

struct page 是存储物理页面的信息的结构体. 其中包括物理页面的地址，分配和回收页面需要的信息等，定义在 include/linux/mm_types.h.

虚拟地址到物理地址的映射需要用 PFN 来找到物理页面，这就需要用 PFN 算出对应的 struct page，完成 PFN 和 page 之间的转换的是宏 pfn_to_page 和 page_to_pfn，不同的物理内存模型的实现不同，内存模型相关定义可见 include/asm-generic/memory_model.h. 其中 VMEMMAP 由于有虚拟地址上连续的vmemmap，其转换过程就是涉及数组元素偏移量的简单计算：

```
1 /* include/asm-generic/memory_model.h */
2 #define __pfn_to_page(pfn)  (vmemmap + (pfn))
3 #define __page_to_pfn(page) (unsigned long)((page)
      - vmemmap)
```

SPARSEMEM 的 VMEMMAP 模型下，struct page *vmemmap 由架构相关的代码定义. x86-64 的位于 arch/x86/include/asm/pgtable_64.h. 架构无关的用于填充 vmemmap 的代码位于 mm/sparse-vmemmap.c.

分 section 的 SPARSEMEM 内存模型解决了物理内存的差异性，那么分配页面时如何区分具有不同功能的内存呢？答案是内存的不同区域还有 ZONE 的区分. 有些设备不能访问整个地址空间，导致只有一部分地址可以用于直接内存访问 DMA. 这一部分内存被设置为 ZONE_DMA 或 ZONE_DMA32. 另外，一些物理地址只是用于访问设备，而不是真正的内存地址，这些空间被设置为 ZONE_DEVICE，其中的页面永远不会成为空闲页面. 其他的页面都可以用来当作普通的内存来分配，称为 ZONE_NORMAL. Linux 内核中的物理页面分配代码被称作 zoned page allocator，是因为每一个 zone 是被单独管理的，各个 zone 之间的分配互不干扰.

4.1.2 页面分配器

在每一个 zone 内部，分配器主要实现两个功能：分配页面和释放页面，即根据调用者的需要找到一定数量的空闲的页面，向调用者返回分配的页面的地址，并把这些页面记录为正在使用，在调用者使用完这些页面后，再向分配器归还这些页面，使他们重新变为可以分配的空闲页面.

在分配问题 [24]中，要避免的是碎片化问题. 页面分配器只关心外碎片化问题，因为其并不管理页面内部的内存组织方式. 外碎片化问题——无法找到需要的大小的空间即使总的空闲空间能够满足要求 [24] ——出现的原因是较小的已分配区域散落在不连续的区域中，导致没有大块的空间来满足分配要求. 既然问题出在小区域“割裂”大区域上，那么尽量减少分割的次数、并且尽可能多地合并就可以降低外碎片化程度. Linux 的页面分配器采用的“伙伴系统”（buddy system）就是这样的一个分配算法.

Note

实际上 Linux 内核的页面分配器还可以分配比页面更小的单元——fragments.

伙伴系统的主要思想是为不同大小的空闲内存块分别维护一个列表，方便直接找到所需大小的空闲块，并且多个小的块在被释放后可以合并为大的块，而大的块如果需要也可以很方便地分割成小的块. 具体的要求是这样的：块的大小必须为页面大小的 2^k 倍（k 为自然数），其中 k 被称为块的次[8]；如果要求的页面数量 n' 不为 2 的幂，则也要分配最小的满足要求的 2 的整数幂的数量的页面，即分配的页面的数量 n 满足：

$$n = 2^k = 2^{\lceil log_2 n' \rceil}$$

假设内存的容量为 2^m，所有页面都是空闲的时候，整个内存为一个 m 次的块. 每次分割空闲块的时候都是对半分割[9]，一个 k 次的块可以分割成两个 $k-1$ 次的较小的块. 为了减少碎片，每当两个原本是从同一个 k 次块分割

[8] order. 类似于多项式的次，如 x^3 为 3 次多项式，故我翻译成“次”，k 次的块有 2^k 个页面.

[9] split.

出来的两个 $k-1$ 次的块都空闲时，它们就会被重新合并[10]成原来的 k 次的较大块. 这样的两个相邻的块就是一对伙伴. 互为伙伴的两个块的次数一定相等，位置一定相邻. 但是次数相等、位置相邻的块不一定互为伙伴. 只有伙伴可以合并的规则保证了合并产生的块一定是按该块的大小对齐的，再结合对半分割的规则，可以推出：所有的块都是按其大小对齐的. 块的对齐对减少碎片化程度也有帮助.

伙伴系统的规定对操作二进制地址非常友好.[15] 可以观察到，若认为块中第一个页面的 PFN 为块的地址，则有：

- k 次块的地址的低 k 位为 0；
- 相邻的 k 次块地址的第 $k+1$ 位（由低到高从 1 开始数）相反；
- k 次块分割出的两个 $k-1$ 次块地址的低 k 位分别为 $0\underbrace{00\ldots0}_{k-1\text{个}}$ 和 $1\underbrace{00\ldots0}_{k-1\text{个}}$；

因此，伙伴关系可以直接通过对地址做简单的位运算得到，只需对块地址的第 k 位取反就可以算出它的伙伴的地址，这可以通过异或做到：

```
/* mm/internal.h */
static inline unsigned long
__find_buddy_pfn(unsigned long page_pfn, unsigned int
   order)
{
  return page_pfn ^ (1 << order);
}
```

Listing 3: 计算伙伴的 PFN

现在我们再来分析具体的分配和释放页面的算法，了解分割和合并是如何进行的.

若调用者要求分配 k 次的块，分配器首先找到最小的满足要求次数的空闲块列表，从列表中找到一个空闲块，并把它从空闲列表中移除. 若找到

[10]Knuth 称为 coalesce，Linux 内核称为 merge.

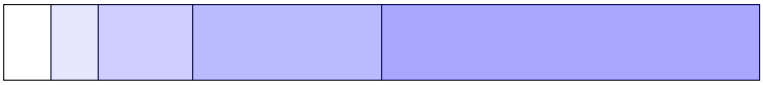

图 3: 伙伴系统分割示意图

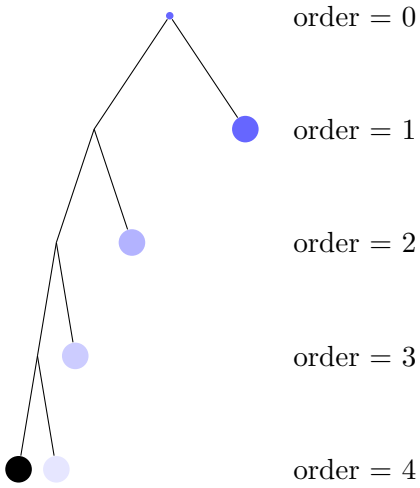

图 4: 伙伴系统分割树型示意图

的块的次数 $k' > k$，则分割直到获得 k 次的块. 分割 q 次块时，前一个 $q-1$ 次块作为分割出 $q-2$ 次块的对象，后一个块则被标记为 $q-1$ 次伙伴，并被加入到 $q-1$ 次的空闲列表中. 如图 3所示，有颜色的为每次分割的后一个块，它们被加入各自的空闲列表中，并且被记录成伙伴. 这种分割关系还可以用二叉树表示，如图 4所示，蓝色的节点为空闲块，且为伙伴，黑色的节点为已分配的块，只有兄弟节点才能合并，合并以后就变成了他们原来的父节点. 每层的块的次相等，且随层数递增.

若调用者需要释放之前分配的页面，分配器会找到它的伙伴（见 Listing 3），如果伙伴空闲，则把要释放的块和它的伙伴合并，并且把该伙伴从空闲列表中删除. 然后继续寻找新合成的块的伙伴，尝试合并，直到伙伴不空闲或没有伙伴，无法合并为止. 最后把合并而来的内存块加入到对应次的空闲列表中. 合并是分割的逆过程，图 3和图 4 在合并伙伴的过程中仍然适用.

伙伴系统可以达到很高的内存利用率，在模拟中可以在无法分配之前达到 95% 的利用率. 而且，伙伴系统不仅无需定期的压缩和移动来减少碎片，出现分割和合并操作的频率也很低，总体上使用得最频繁的大小的块也是最多的.[15]

读源码 4.2：Buddy system

伙伴系统分配器的实现主要在 mm/page_alloc.c 中.

各种分配内存的函数最终都会调用 `__get_free_pages()` 函数，通过向其传递不同的选项（GFP flags）来控制分配器的行为. 最终进行分配的函数是 `rmqueue()`. 它寻找最小的空闲块，并调用 `expand()` 来对较大的空闲块进行分割.

处理释放工作的是函数 `__free_one_page()`，它负责释放块，合并伙伴，更改空闲队列.

4.2 slab 分配器

使用伙伴系统的页面分配器是 Linux 内核的最底层的内存分配系统，在页面分配器的层次之上，还有更精细的内存分配机制. Linux 内核采用 slab 分配器来为内核中常用的多种需要动态分配的对象管理内存. 在 slab 分配器中，同种对象连续地存储在一起，批量地分配内存，有状态的对象在释放和分配之间，状态仍缓存在原来的空间中. 批量存储同样的对象有助于提高空间利用率、减小了内碎片化程度，而状态的缓存则避免了频繁初始化的开销. [6]

小问题 4.1：什么是对象？

slab 分配器管理的对象可以是存储状态的某种控制块，如进程控制块、inode 等. 也可以是无状态的缓冲区等.

在 slab 分配器中，对象存储在 cache 中，每种对象都有自己的 cache，每一个 cache 中又有若干 slab. slab 是一小段连续的空间，通常为就为一个页面，要分配的对象就连续地存放在 slab 中. slab 的分配过程中，内部的所有对象就都完成了初始化，所以从 slab 中分配得到的对象永远都是已经初

始化的，调用者在分配和释放的时候既不用重新初始化也不用做全部的清理工作，清理工作会在整个 slab 被销毁的时候进行. 这样的设计是考虑到有些对象的初始化开销甚至大于分配内存的开销，保留共用的初始状态可以显著减少由于反复初始化而带来的性能损失.[6]

借助 slab 还可以减少内碎片化程度. slab 根据存储对象的状态被分为三种：

- 存满对象的；
- 存有对象但没有装满的；
- 没有存储对象的，全部空闲的；

在分配新的对象时，若有没装满的 slab，一定使用没装满的 slab，而不引入新的碎片；只有在没有部分装满的 slab 时，才会使用空闲的 slab 或者分配新的 slab.

读源码 4.3：Slab allocator

slab 分配器的实现在 `mm/slab.c`.

5 设备管理

5.1 设备驱动模型

计算机的 CPU 通过总线连接到设备，设备驱动程序通过总线与设备沟通，并给其他部分暴露出友好的接口. 为总线、设备、驱动的相互操作而建立的程序框架在 Linux 内核中被称作设备驱动模型. [25][7]

设备模型是由一系列相互联系的对象表示的. 表示总线、设备、驱动等的对象之间相互连接，构成层次关系，以便相互操作. 把它们连接起来的是 `struct kobject`，一个嵌入其他结构体中提供功能的结构（嵌入的机制与 3.2 中的 `list_head` 一样），这其实是为了在 C 语言的简单结构下实现几个较复杂功能的一系列 hack.

`kobject` 接口提供的主要功能有 [10]：

- 引用计数. 因为设备驱动模型对象之间相互引用的关系较复杂，需要引用计数来确定对象的生命周期.
- 在需要释放对象时，根据对象的类型调用自定义的释放函数.
- 连接其他 `kobject`. 设备和设备等之间需要构成父子关系. 不同对象之间还可以利用 `kset` 或 `klist` 构成 collection. collection 和父子关系构成了 `kobject` 的层次结构.
- 向用户空间提供接口. `kobject` 在内核登记后就对应 sysfs 虚拟文件系统（Arch Linux 下挂载在 `/sys`）下的一个目录. 用户态程序可以通过该虚拟文件系统的接口访问内核内部的对象.
- 事件的发送. 对象状态更改（如设备插拔）时可以向用户空间发送事件.

利用这些功能，内核就能构建一个灵活的设备驱动模型.

5.1.1 总线

总线由 `struct bus_type` 表示. 所有设备在逻辑上都连在某一个总线上，所有驱动都要依靠总线访问设备. 因此总线的对象中的 `struct`

`subsys_private *p` 有两个装着 `kobject` 的 `kset`，分别引用该总线上的设备和驱动.

总线对象定义了总线普遍支持的操作，初始化具体的总线对象时要注册这些操作的函数指针，其中一些操作是做一些总线相关的工作再调用设备的回调函数. 主要的操作有：

- 在加入新的设备或驱动的时候调用的操作.
  - 检查总线上的某个指定设备是否与指定驱动相匹配；
  - probe，调用驱动的 probe 来把设备加入到驱动的管理中；
  - ……
- 设备从总线上移除后的操作.
- 为设备准备 DMA 的操作.
- 管理设备电源、功耗的操作.

总线下的层次结构可以在 sysfs 中看到. 每一种总线都在 `/sys/bus/` 下有自己的目录. 目录下的 `devices` 子目录是总线设备，为指向全局的 `devices` 子系统中的设备对象的符号链接；`drivers` 子目录下是总线的各个驱动. 例如 Listing 4 所示.

```
/sys/bus/pci-express
|-- devices
|   |-- 0000:00:07.0:pcie001 -> ../../../../devices/
   pci...
...
|   `-- 0000:00:1c.0:pcie010 -> ...
|-- drivers
|   |-- aer
|   |-- dpc
|   |-- pciehp
|   `-- pcie_pme
```

```
|-- drivers_autoprobe
|-- drivers_probe
`-- uevent
```

Listing 4: PCIE 总线在 sysfs 中的结构

5.1.2 设备

设备在 Linux 设备驱动模型中的对象为 `struct device`. 其中存储的一些信息有:

- 设备的名字;
- 设备的 “父设备”, 例如总线控制器、hub 控制器等;
- 设备所在的总线类型;
- 设备使用的驱动;

小问题 5.1: 总线也是设备吗?

值得注意的是虽然已经有表示总线类型的 `bus_type`, 总线也还是作为设备在设备驱动模型中表示, 一种总线可能是另一种总线的子设备, 例如挂在 PCI-E 总线下的 USB 总线.

5.1.3 驱动

设备驱动的对象是 `struct device_driver`. 一个驱动对象只能存在一个实例, 也就是说驱动只有一份, 一个驱动要管理与其关联的所有设备. 所以 `struct device_driver` 的 `struct driver_private *p` 要存储当前驱动所管理的所有设备.

向驱动添加设备的机制是存储在 `struct device_driver` 中的函数指针 `probe()`. 这个函数以 `struct device` 为参数, 应该做这样几件事 [9]:

- 确定设备确实存在;

- 确定设备的型号和版本能被驱动支持；
- 为设备需要的数据结构分配内存空间；
- 初始化设备的硬件；
- 最后把设备驱动和设备绑定在一起，填上各自的引用.

读源码 5.1：设备驱动模型

总线的表示 `struct bus_type` 的定义在 include/linux/device/bus.h. 实际的总线定义自己的 `struct bus_type`，如 PCI 的 `struct pci_bus_type` 在 drivers/pci/pci-driver.c.

设备的 `struct device` 定义在 include/linux/driver.h. 往往 `struct device` 不单独使用，特定总线的设备会把 `struct device` 嵌入在一个有更多信息的结构体中. 比如 PCI 的 `struct pci_dev`（include/linux/pci.h）、USB 的 `struct usb_dev`（include/linux/usb.h）.

驱动的 `struct device_driver` 在 include/linux/device/driver.h. 同样特定总线的驱动会把 `struct device_driver` 嵌入到自己定义的结构体中.

5.2 设备驱动过程

在 2.2 中，我们提到内核启动时，bootloader 加载 vmlinux 镜像，内核代码开始运行. 在开始启动 init 进程和运行 init 程序之间，内核首先做最核心的初始化，这其中就包括设备驱动模型的初始化. 系统的总线等重要的设备的内核模块是静态连接到 vmlinux 内的，在设备驱动模型初始化完成后，这些模块的初始化程序会运行，注册对应的设备和总线类型，对总线进行初始化并设置中断. 这样基本的驱动框架就建立好了，总线的设备驱动就可以对总线上的设备进行搜索. 每当找到一个新的设备，首先初始化相关的对象，然后通过总线的 `match()` 和驱动的 `probe()` 来遍历该总线下的所有驱动，如果发现匹配的驱动，则停止遍历，把设备和驱动绑定起来，这样设备模型的总线、设备、驱动就都联系起来了.

现在再把用户空间的操作考虑进去. 嵌有 kobject 的设备、驱动对象在创建、删除的时候都会向用户空间传递 uevent. 用户空间的程序可以通过监听事件，并且通过 sysfs 对设备驱动模型进行操作，从而支持更灵活的设备使用方式，如热插拔. 在用户态受限的环境下管理设备的好处还有稳定性和安全性的提高. 同样在 2.2 中提过，包括 Arch Linux 在内的大多数 x86 发行版都使用 systemd 的 udev 系统来在用户态管理设备. udev 负责监听 uevent 事件，根据 udev 自带的和用户自定义的规则来做出对应的动作，管理 /dev 下的 device node 文件，使其他程序可以通过 /dev 下的文件与设备驱动程序交互. udev 规则一般可以实现这样的一些功能 [27]：

- 加载、卸载驱动所在的模块（见 2.2）.
- 在 /dev 下为设备设置友好的名字.
- 通知其他程序，传递设备的事件.
- 进行系统管理，如挂载刚刚插入的硬盘.

使用 udev 的命令行工具 udevadm 可以直观地看到设备状态变化产生的操作. 比如运行 udevadm monitor --property 命令，插入一个 USB 键盘，就可以看到 udev 根据规则和事件创建的 device node.

```
1 UDEV  [401515.997363] add      /devices/pci0 ... /usb3
     /.../event12 (input)
2 ACTION=add
3 DEVPATH=/devices/ ... /usb3/ ... /event12
4 SUBSYSTEM=input
5 DEVNAME=/dev/input/event12
```

Note

这里描述的过程进行了大量的简化，许多重要的细节都忽略了. 例如驱动可能在 probe() 时还为准备好，这时它可以要求推迟 probe() 的操作.

读源码 5.2: driver_init()

初始化设备模型的函数入口为 driver_init()，位于 drivers/base/init.c，由 init/main.c 中的 do_basic_setup() 调用. drivers/base 目录为内核驱动的“core”部分，负责硬件驱动模型内部的初始化和实现模型内部的功能等.

drivers/base/core.c 中的 device_add() 处理添加设备时的一系列操作,调用 bus_probe_device(dev); 来匹配驱动. bus_probe_device() 位于 drivers/base/dd.c，dd 是“device/driver”的缩写，该文件中的函数处理设备和驱动之间的关系，例如绑定、通信等.

6 ext4 文件系统

ext4是近年许多 GNU/Linux 发行版的默认文件系统，于 2008 年在内核中稳定 [11]，是较为传统的日志文件系统，而不是 BTRFS、ZFS 那样的集成了卷管理器的文件系统，但功能也较为完善.

6.1 文件系统的系统调用

Linux 支持的文件系统种类非常丰富. 在处理多种文件系统共存和代码复用的问题上 Linux 内核采用了实现一个叫做 “虚拟文件系统” 的软件层的方法.

虚拟文件系统把文件系统中共有的概念抽象出来，规定了统一的对象和接口，特定的文件实现自己的算法和数据结构来提供这些对象，例如：

- Superblock. 整个文件系统的控制信息.
- Inode. 文件的控制信息.
- File. 已打开的文件的控制信息.
- Dcache. 目录条目的缓存.

有了这些通用的对象，文件系统就可以把自己注册到 Linux 的虚拟文件系统中，虚拟文件系统对内通过调用这些对象的方法来完成特定文件系统中的操作，对外提供系统调用，使用户态程序能够用同样的方式操作不同的文件系统. 文件系统的挂载和卸载是系统调用.

常用的读写文件的系统调用包括但不限于：

- `open` 打开已有的文件或者在文件系统中创建一个新的文件并打开，返回文件描述符；
- `read` 从已打开的文件读出内容放入缓冲区；
- `write` 向已打开的文件中写入内容；

[11] https://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=03010a3350301baac2154fa66de925ae2981b7e3

- `poll` 检查文件是否有活动，进程可以在没有活动时等待；
- `ioctl` 控制读写文件的方式；
- `mmap` 把文件映射到用户的虚拟内存空间中，从而通过读写内存的方式读写文件；

常用的管理文件的系统调用包括但不限于：

- `creat` 创建文件；
- `link` 创建硬链接，实现上一般是使不同的目录条目指向同一个 inode；
- `unlink` 删除 inode 的某“名字”，当没有其他引用时（如文件描述符、其他硬链接等），inode 会被删除；
- `mkdir` 创建目录；
- `rmdir` 删除目录；
- `rename` 对 inode “重命名”，可以把文件移动到别的目录；
- `stat` 从 inode 中读取信息；
- `chmod` 更改文件的权限.

小问题 6.1：系统调用的文档在哪里查？

Linux 的 man 手册的第二节是系统调用，可以用命令 `man 2 read` 查看 `read(2)` 的相关内容. 在 POSIX 的标准中也可以查到系统调用的规定.

6.2 磁盘空间和文件的组织

6.2.1 存储单元

ext4 中，磁盘的空间被分为几种单位：sector、block 和 block group.

- sector 是由硬盘决定的扇区，一般为 512 B. 较新（2010 后）硬盘一般扇区大小为 4096 B，可以工作在 4 KB 的原生扇区大小模式下，也可以模拟 512 B 的扇区大小.
- block 是 2 的整数次幂个连续扇区组成的单元，一次传输较多数据可以减小磁盘操作开销. 一般是 4 KB，为了匹配常见的内存页面的大小. 默认开启 64 位特性时，一个文件系统可以有 2^{64} 个 block.
- block group 是连续的 block，默认为 32768 个 block. ext4 会利用 block group 来提高局部性.

值得注意的是，在引用内存中的数据时，指针为按字节寻址的地址，也就是引用对象的第一个字节的字节数；而在外存中，存储单元为 block，所以指针均为 block number. 还有一点，因为文件系统不应依赖 CPU 的类型，所以应该定义好所有的数据的字节序. ext4 中除日志外的数据均为 little-endian.

6.2.2 文件的组织形式

文件的数据部分存储在数据 block 中，而要想把数据 block 组织成一个文件，还需要一些辅助用的 block. ext4 的空间分配方式为索引式，也就是组成文件的块可以不连续，用 inode 来索引分配给文件的块. inode 其实就表示了文件的连续的逻辑块号到不连续的物理块号的一个映射，和页表在虚拟内存中的作用类似.

inode 中存有文件的元数据和一些能够索引到数据 block 的数据结构. 普通的索引方式是直接或间接地存储每一个 block 的指针. 直接索引的 block 有 12 个（`EXT4_NDIR_BLOCK`），这 12 个指针的后面是 3 个指向间接索引块的指针，分别为一级、二级和三级间接索引. 15 个指针共有 60 个字节，存放在 `i_block` 字段中. 小的文件不需要间接索引块，而如果文件的数据块超过 12 个，就需要分配间接索引块，若一级间接索引仍不够再依次使用二级索引、三级索引. 若 block 大小为 4 KB，则每一个间接块可以存放 32 位指针（这里只能用 32 位的 block nunber）的数量为 $4 \times 2^{10}/4 = 2^{10}$. 故全部的二级索引块可以索引 $2^{10} \times 2^{10} \times 4\,\mathrm{kB} = 4\,\mathrm{GB}$. 可以看出，只有数 GB 的文件才要用到三级间接索引.

另外一种方式是 extent. 这是一种介于连续分配和索引之间的组织方式，可以弥补两者的不足. 分配时，尽量分配连续的 block，但是确实无法分配连续的空间时，也可以用分开的若干个连续区域来存储，每一个连续的空间称为一个 extent. 为了在连续的逻辑块号和这样的部分连续的物理块号之间建立索引，Linux 采用的是 extent tree，一种类似于 B+ 树的索引结构. 我们在数据结构课上学过，B+ 树是一种多分支的自平衡树，树中的元素都是叶子节点，非叶子节点只用于索引，extent tree 也是这样. 每一个 extent 记录起始块号和长度，树的叶子节点就是 extent. 索引方式下存储指针的位置，extent 方式则存放 extent 树. extent 树的每一个节点都有一个 header 描述这个节点可索引多少个 extent，已经有多少 extent 等信息. 如果 inode 的那 60 个字节可以放下一个 header 和所有 extent，则不需更多块来存放节点. 如果放不下，则要使用索引节点来间接指向 extent.[11]

`i_block` 的 60 个字节除了存放块索引和 extent tree，还可以存放符号链接的目标路径，若存不下，还需分配更多空间.

目录也是文件的一种，同样由 inode 索引. 与其他文件不同的是，目录文件的数据块存放的是目录中每一个文件或目录的条目，ext4 中，每一个文件的目录条目包含：

- 文件名字符数组、文件名长度（不超过 255）；
- 文件的 inode 号；
- 文件类型；

其中文件类型也在 inode 中存在，在目录条目复制一份的好处是，查看目录下所有文件类型时无需访问每一个文件的 inode，只需要查看所有条目即可.[3] 目录条目除了顺序存储，还可以组织成更复杂但高效的 htree（hashed btree）. 在这种组织形式下，用名字访问文件的时间复杂度更低，只需用文件名的散列值作为索引在 B 树中搜索.

小问题 6.2：ext4 文件系统与 ext3、ext2 有什么改进？

ext4、ext3、ext2 都是同一个系列的文件系统，是对 Minix 的文件系统的改进和扩展（**E**xtented file system). ext3 文件系统增加了日志功

能. ext4 文件系统则新增了一些功能，提升了一点性能，减少了一些限制. 上文就提到了几处 ext4 的改进之处：新增了 extent 方式的文件组织形式、在目录条目里面存储文件类型.

读源码 6.1：`struct ext4_inode`

ext4 的主要代码在 `fs/ext4` 目录中. `fs/ext4/ex4.h` 定义了 ext4 的 inode 在磁盘上的布局 `struct ext4_inode`，和在内存中的表示形式 `struct ext4_inode_info`.

`struc ext4_inode` 中的 `__le32 i_block[EXT4_N_BLOCKS]` 就是存放上述有关索引数据的 60 个字节.

关于多层间接索引的 inode 的使用，可参考的逻辑块号到索引块内偏移的代码：`fs/ext4/indirect.c` 中的 `ext4_block_to_path()`.

Extent tree 的实现见 `fs/ext4/ext4_extents.h` 等相关文件. `struct ext4_extent` 是记录 extent 位置的描述符，也是叶子节点. `struct ext4_extent_idx` 是非叶子节点.

6.2.3 磁盘上的物理布局

6.2.1 中提到了几种存储单位，其中 block 是存储的基本单位，sector 是与磁盘交流用的单位，而 block group 是文件系统整体组织的单位. 整个磁盘被分为若干 block group，动态分配空间时，总的原则是提高局部性，尽量把相关的结构分配在同一个 block group 中，因此每一个 block group “麻雀虽小，五脏俱全”，都是相同的布局 [1]：

1. 一个整个文件系统的 superblock. 每一个 block group 都存一个一样的文件系统控制块，减小文件系统关键信息丢失，完全不可恢复的概率.
2. Group 描述符表，所有 block group 的描述符的数组. 也是重复地记录在每个 group 中. 其后有一段保留的空间，用于在扩大文件系统时写更多的 group 描述符.
3. 数据 block 的 bitmap. 每一位对应一个 data block 是否空闲.

4. inode 的 bitmap. 每一位记录 inode 是否空闲.

5. inode 表. 固定数量的 inode 数组，所有 inode（在使用的和空闲的）都在这里.

6. 数据 block，间接索引 block，extent tree block 等等.

其中，group 描述符存储在 group 表中，每一个描述符对应一个 group，存放就是其他几个控制用的数据结构的位置：inode 和数据 block 的 bitmap、inode 表. 因此，只要找到了 group 描述符表，就可以找到 group 描述符，进而确定其他所有控制信息的位置，访问整个文件系统的信息就找全了. 这样也给了布局很大的灵活性，只有 group 描述符表是固定的，其他的信息都可以不固定，这基础上 group 的组织可以更加灵活.[2]

了解了磁盘上存放的控制信息，我们就知道了如何从逻辑上的号码访问物理上的位置，以及如何从物理地址推出逻辑上的结构. 所要做的就是一层层查找表和描述符，计算一些偏移量，跟随若干次指针.

6.3 文件系统的安全

为了保证文件系统的安全，除了要管理好文件的权限，还要做好系统完整性的检查，最好还能从故障中恢复.

ext4 的 inode 存有文件拥有者的标识符，并且在扩展的属性中记录了访问控制列表（ACL），作为 Unix 文件权限的补充，提供更精细的权限控制.

ext4 的所有数据结构如 superblock、inode、group 描述符等都支持校验，所有的元数据都可以算出校验和与存储的校验和进行比较，从而发现文件系统中是否有完整性错误.[1]

另外，ext4 还支持写日志的功能. 我们已经看到，与 LFS [21] 这种基于日志的文件系统不同，ext4 的文件并不以日志的形式保存. ext4 的日志功能 jbd2 完全是为了在系统故障后进行恢复而设计的，因而日志功能与文件系统的主体可以分离，用户也可以选择启用或者不启用日志功能. 日志在 ext4 中也是以普通的文件的方式存储，也有 inode，只不过对文件系统是隐藏的. 默认设置下，日志只写元数据. 每次文件系统的状态发生的改变需要保证原

子性时，改变都会先被写到日志里，然后才会被真正执行. 若由于系统故障，操作未能完成，文件系统可以重放日志里的记录来尝试恢复完整性.

7 附录

7.1 “小问题”列表

7.2 “读源码”列表

References

[1] *2. High Level Design – Ext4 — The Linux Kernel documentation*. https://docs.kernel.org/filesystems/ext4/overview.html. (Accessed on 05/31/2022).

[2] *3. Global Structures – Ext4 — The Linux Kernel documentation*. https://docs.kernel.org/filesystems/ext4/globals.html. (Accessed on 05/31/2022).

[3] *4. Dynamic Structures – Ext4 — The Linux Kernel documentation*. https://www.kernel.org/doc/html/latest/filesystems/ext4/dynamic.html#ftype. (Accessed on 05/31/2022).

[4] *Arch Boot Process - Arch Wiki*. https://wiki.archlinux.org/title/Arch_boot_process. (Accessed on 05/15/2022).

[5] *Arch Linux - ArchWiki*. https://wiki.archlinux.org/title/Arch_Linux#History. (Accessed on 05/09/2022).

[6] Jeff Bonwick et al. "The slab allocator: An object-caching kernel memory allocator". In: *USENIX summer*. Vol. 16. Boston, MA, USA. 1994.

[7] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel: From I/O Ports to Process Management*. O'Reilly Media, 2005. ISBN: 9780596554910. URL: https://books.google.com.hk/books?id=h0lltXyJ8aIC.

[8] *CFS Scheduler — The Linux Kernel documentation*. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html. (Accessed on 05/21/2022).

[9] *Device Drivers — The Linux Kernel documentation*. https://docs.kernel.org/driver-api/driver-model/driver.html. (Accessed on 05/30/2022).

[10] *Everything you never wanted to know about kobjects, ksets, and ktypes —The Linux Kernel documentation*. https://docs.kernel.org/core-api/kobject.html. (Accessed on 05/30/2022).

[11] *How Ext4 Extents Work?* https://ext2read.blogspot.com/2010/03/how-ext4-extents-work-earlier-ext2-and.html. (Accessed on 05/31/2022).

[12] IEEE and The Open Group. *pthread.h - The Open Group Base Specifications Issue 7, 2018 edition*. https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html. (Accessed on 05/21/2022).

[13] *image - Wiktionary*. https://en.wiktionary.org/wiki/image. (Accessed on 05/09/2022).

[14] *Judd Interview*. https://distrowatch.com/dwres.php?resource=interview-arch. (Accessed on 05/09/2022).

[15] Donald E. Knuth. *Fundamental Algorithms*. 3ed. Vol. 1. The Art of Computer Programming. Addison-Wesley Professional, 1997. ISBN: 978-0201896831.

[16] *Linux 2.6.23 - Linux Kernel Newbies*. https://kernelnewbies.org/Linux_2_6_23#The_CFS_process_scheduler. (Accessed on 05/21/2022).

[17] *Linux Kernel Makefiles —The Linux Kernel documentation*. https://www.kernel.org/doc/html/latest/kbuild/makefiles.html. (Accessed on 05/10/2022).

[18] *OS Usage Trends and Market Share*. https://web.archive.org/web/20150806093859/http://www.w3cook.com/os/summary//. (Accessed on 05/09/2022).

[19] *Physical Memory Model —The Linux Kernel documentation*. https://docs.kernel.org/vm/memory-model.html. (Accessed on 05/25/2022).

[20] Dennis M Ritchie. "The evolution of the Unix time-sharing system". In: *Symposium on Language Design and Programming Methodology*. Springer. 1979, pp. 25–35. URL: https://www.bell-labs.com/usr/dmr/www/hist.html.

[21] Mendel Rosenblum and John K Ousterhout. "The design and implementation of a log-structured file system". In: *ACM Transactions on Computer Systems (TOCS)* 10.1 (1992), pp. 26–52.

[22] *sched.h - The Open Group Base Specifications Issue 7, 2018 edition*. https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/sched.h.html. (Accessed on 05/23/2022).

[23] R. Sedgewick. *Algorithms in C, Parts 1-5 (Bundle): Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms*. Algorithms in C. Addison Wesley Professional, 2001. ISBN: 9780201756081. URL: https://books.google.com.hk/books?id=4IP%5C_ugAACAAJ.

[24] A. Silberschatz, G. Gagne, and P.B. Galvin. *Operating System Concepts*. 10th. Wiley, 2018. ISBN: 9781119800361.

[25] *The Linux Kernel Device Model — The Linux Kernel documentation*. https://docs.kernel.org/driver-api/driver-model/overview.html. (Accessed on 05/30/2022).

[26] *Unreliable Guide To Hacking The Linux Kernel — The Linux Kernel documentation*. https://docs.kernel.org/kernel-hacking/hacking.html. (Accessed on 03/06/2022).

[27] *Writing udev rules*. http://www.reactivated.net/writing_udev_rules.html#terminology. (Accessed on 05/31/2022).

索引