清 华 大 学

综 合 论 文 训 练

题目：monoRCore 模块化操作系统的设计与完善

系　　别：计算机科学与技术系

专　　业：计算机科学与技术

姓　　名：贾悦婷

指导教师：陈　渝　副教授

2023 年 6 月 6 日

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存该论文。

(涉密的学位论文在解密后应遵守此规定)

签　名：_____ 导师签名：_____ 日　期_____

中文摘要

模块化操作系统通过精心设计每个模块的抽象接口实现了操作系统内核功能的拆分，使得对单一内核功能的修改与完善更加方便和容易，这一特点使得模块化操作系统受到操作系统研究者的青睐。monoRCore 是基于清华大学计算机系操作系统课程教学内核 rCore-Tutorial 改进得到的模块化操作系统。本文完善和改进了 monoRCore 操作系统，向其中添加了调度模块和页面置换模块，并在调度模块和页面置换模块中，分别实现了多种调度算法和页面置换算法。

任务调度和页面置换是操作系统的重要功能。操作系统通过特定的调度算法决定某一时刻应当执行哪个处于等待状态的任务，而任务的执行次序对操作系统的性能有着至关重要的影响。操作系统通过页面置换为任务提供足够大的虚拟存储空间，并通过特定的页面置换算法决定某一时刻虚拟存储空间中的哪些部分应当被放置在内存中。页面置换算法对内存中放置的内容的选择影响任务运行过程中的缺页率，从而影响操作系统的性能表现。因此，为 monoRCore 添加调度模块和页面置换模块是对 monoRCore 系统的重要改进。

本文还对添加的调度模块和页面置换模块中的所有调度算法和页面置换算法进行了实验，这些实验展现了算法实现的正确性，以及不同算法在不同任务上的表现和性能。

关键词：模块化操作系统；调度算法；页面置换算法；虚拟存储

ABSTRACT

Modular operating systems(OS) separate different kernel functionalities by carefully designed abstract interfaces for each module, making it convenient and easy to revise and refine a single kernel function. This makes modular OS favorable to OS researchers. monoRCore is a modular OS developed from rCore-Tutorial, an OS kernel used for educational purposes by Tsinghua University. This paper refines monoRCore by adding scheduling module and page replacement module to monoRCore and implements multiple scheduling and page replacement algorithms in the corresponding module.

Scheduling and page replacement are vital functionalities in an OS. An OS uses a certain scheduling algorithm to decide which waiting task to run at a certain moment. Task execution order has a crucial impact on OS performance. Page replacement is responsible for providing enough virtual memory space for tasks and the page replacement algorithm decides which part of the virtual space should be placed in RAM. These decisions affect the page fault frequency during task execution and thereby affect OS performance. Therefore, adding scheduling module and page replacement module is an important improvement to monoRCore.

This paper also presents the experiments for all the scheduling algorithms and page replacement algorithms implemented in the corresponding module. The experiments show the correctness of the implementation and how these algorithms react to different workloads.

Keywords: modular operating system; scheduling algorithm; page replacement algorithm; virtual memory

目　　录

主要符号对照表

OS	操作系统
RTOS	实时操作系统
FSS	公平共享调度
FCFS	先来先服务
RR	时间片轮转
SJF	最短作业优先
STCF	最短完成时间优先
HRRN	最高响应比优先
MQ	多级队列
MLFQ	多级反馈队列
EDF	最早截止时间优先
RMS	单调速率调度
DDL	截止时间
FIFO	先进先出
PFF	缺页率

第 1 章　引言

1.1　课题背景

1.1.1　课题目标与内容

本课题的目标是完善 monoRCore 的任务调度模块，并添加多种可选的调度策略。同时为 monoRCore 添加页面置换模块，并添加多种可选的页面置换策略。本课题中同时还分别对添加的调度策略和页面置换策略进行了测试、比较和分析。

1.1.2　rCore-Tutorial 与 monoRCore

rCore-Tutorial[1]是清华大学计算机系操作系统课程使用的教学操作系统，目前更新到了第三版，即 rCore-Tutorial-v3（后简称为 rCore)。rCore 使用 Rust 语言编写，支持 RISC-V 指令集，可以运行在 QEMU 模拟器中和 K210 开发板上。

rCore 包含 9 个简易操作系统，它们分别位于 9 个分支上，对应于对应于清华大学操作系统课程的 9 个课程实验（ch1 - ch9)。这 9 个操作系统的复杂度递增，每个操作系统都在其前一个操作系统的基础上进行了功能的完善与丰富。最终的 ch9 对应的操作系统实现了进程通讯、多线程并发、文件系统、输入/输出设备管理等功能，是基础功能完备的操作系统。

由于课程实验的内容是逐步递进的，同学们往往需要将上一次实验的内容转移到下一次实验的操作系统上，在上一次实验的基础上进行修改完善。而 rCore 的 9 个课程实验位于 9 个分支上，使得这种转移往往十分繁琐。为解决这一问题，课程组对 rCore 进行了模块化封装，得到了 rCore-Tutorial-in-single-workspace 操作系统 (后简称为 monoRCore)[2]。

由于 monoRCore 进行了模块化封装，对于 monoRCore 中模块的扩展可以直接被多个实验共享。同时，模块化封装也使得完成实验任务所需要修改的代码更加集中，便于同学们完成实验，也有利于操作系统课程的教学。

1.2　模块化操作系统

传统操作系统的各个功能的代码之间有着复杂的依赖关系，使得对操作系统的部分功能进行修改变得繁琐和困难。图 1.1是 Linux 中各个组成部分之间的依

赖关系图，其中箭头上的数字表示出现依赖的次数。

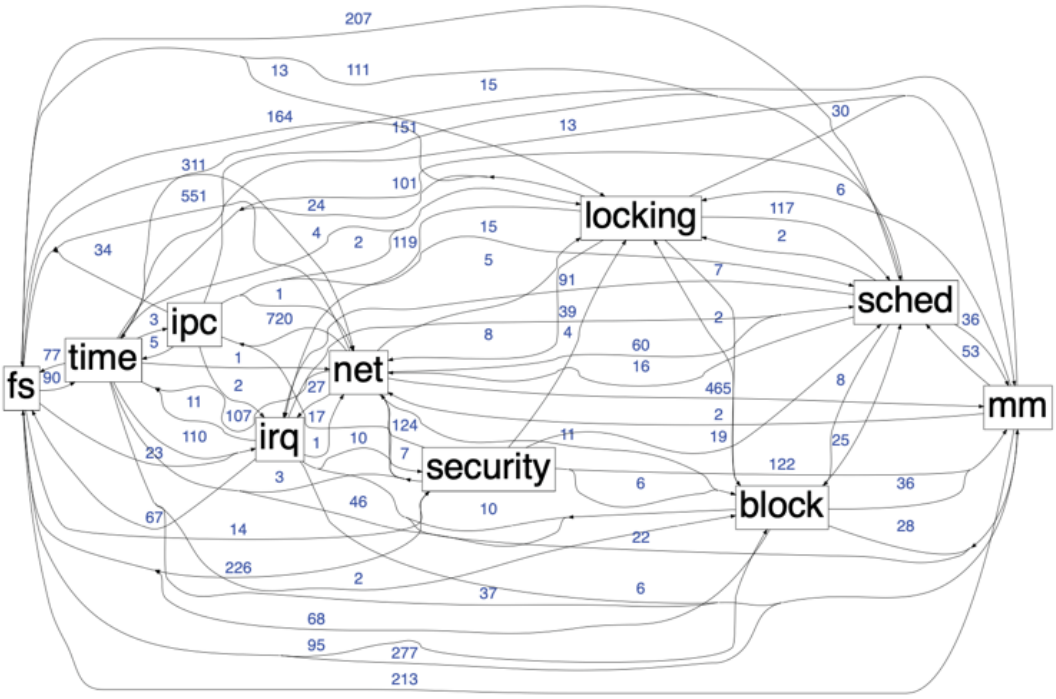


图 1.1　Linux 内核组成部分之间的依赖关系图[3]

为了能够更方便的进行操作系统部分功能的修改，从而为操作系统的研究提供便利，模块化操作系统的概念被提出。模块化操作系统通过将各个功能进行模块化抽象，精心设计每个模块对外的抽象接口，从而隐藏了每个模块内部的实现，使得功能模块的添加、删除和修改更为便捷。犹他大学开发的 OSKit[4] 就是一个典型的为方便操作系统研究设计的模块化操作系统。OSKit 支持使用者通过设置编译选项和编写简单的内核启动文件，来选择需要的功能模块。同时，OSKit 通过精心设计模块间的“胶水代码”(glue code)，减少了模块间的互相依赖，使得使用者能方便的修改其中某个模块的内容，而不需要修改其他模块和其他代码。

模块化操作系统除了能方便操作系统的开发和相关研究以外，还可以用于生成轻量级的虚拟化容器。模块化操作系统容易进行内核功能的删除，因此可以根据需要运行的具体任务，确定所需内核功能的子集，并使用模块化操作系统编译出仅包含这些功能的操作系统，用以生成运行该任务的轻量级容器。由于这样生成的操作系统的功能大幅度简化，其生成的虚拟容器在性能和占用的空间上都较使用传统操作系统的容器更具优势。Unikraft[3] 是支持轻量级容器生成的模块化操作系统的一个范例。使用 Unikraft 生成的 NGINX 容器运行 `wrk` 的吞吐量相比

Linux 容器提升了 4 倍[3] 。ArceOS[5] 是使用 Rust 语言编写的、支持轻量级容器生成的操作系统，它受到了 Unikraft 的启发，目前正在开发中。

1.3 操作系统中的任务调度

1.3.1 进程与线程

操作系统为用户程序提供了执行环境，同时也管理着正在运行的每个用户程序。进程 (Process) 和线程 (Thread) 是操作系统中的两个抽象概念。其中，进程是较早提出的。在其提出之初，进程指的是在操作系统管理下，程序的一次执行过程。线程是为了提高进程内的并发性而出现的。一个进程往往可以被分成多个可并行执行的任务，这些任务需要共用一些资源（如内存等）。这些细分的任务就被称为线程。不同于进程，线程之间共享地址空间和内存等资源，因此并发执行更为方便。在线程提出后，线程成为了程序执行过程的载体，而进程则成为了线程的容器，管理其下所有线程的共享资源。

1.3.2 任务调度

任务调度是指在有多个任务时，由操作系统确定每一时刻应当运行哪个任务，并进行任务切换。在线程出现前，进程是操作系统进行调度和资源分配的基本单位。在线程出现后，线程就成了操作系统进行调度的单位。

模块化操作系统要求各个功能模块能自由组合，是否包含线程的功能也是在编译时指定的。因此对于模块化操作系统，调度的基本单位可能是线程也可能是进程，所以在本文中，使用“任务”来代指被调度的对象。

1.4 操作系统中的页面置换

页面置换指的是操作系统将内存中的物理页帧替换到磁盘等外部存储设备中，并从外部存储设备中取出物理页帧数据，将其加载到内存中的操作。

之所以需要进行这样的操作，是因为现代操作系统希望给每个用户任务提供尽量大的空间，从而使用户任务能顺利完成。操作系统可以使用内存或者外部设备为各个用户任务提供所需要的运行空间。由于硬件层面上，内存中的数据访问快，外部存储设备中的数据访问慢，操作系统会优先使用内存为各个用户任务提供运行空间。但提供给每个用户任务的空间加起来往往会超过物理内存的大小，因此操作系统需要不定期的将一部分存储在内存中的物理页帧替换到外部存储

设备中，并从外部存储中取出用户当前需要使用的页帧内容，才能在给各个用户任务提供足够大的空间的同时，保证用户任务能快速访问运行空间中的数据。

由于操作系统往往给每个用户任务分配了虚拟地址空间，操作系统的页面置换操作对于用户任务来说是隐形的，即用户任务无法察觉其访问的内容是在内存上还是外部存储设备中，也不知晓访问过程中是否发生页面置换。分配虚拟地址空间具体来说，是操作系统给每个用户任务建立了一个虚拟地址 (Virtual Address) 到物理地址 (Physical Address) 的“映射表”，当用户任务以虚拟地址访问某一位置时，硬件会查找操作系统提供的“映射表”，将虚拟地址转化为对应的物理地址，然后完成访问内存中对应的物理位置，进行相关操作。操作系统提供的“映射表”被称为页表 (Page Table)，这是由于操作系统是以页帧为单位进行空间分配、置换和地址映射的。如果用户任务访问的位置所对应的页帧不在内存中，会触发缺页异常 (Page Fault)，操作系统收到异常后，会进行上述的页面置换操作，将对应的页帧从外部存储设备换入内存中。

1.5 论文结构

本文分为 6 个章节。其中，第一章节主要介绍课题背景、相关概念和与课题相关的工作，第二章节将介绍 monoRCore 的架构，第三章节介绍调度模块的设计和实现，第四章节将介绍页面置换模块的设计和实现，第五、六章节将分别介绍调度算法和页面置换算法的评测。

第 2 章　monoRCore 架构设计

2.1　monoRCore 的总体架构

monoRCore 包含 8 个部分：syscall 系统调用模块，task-manage 任务管理模块，kernel-context 任务上下文模块，kernel-alloc 内核存储分配器模块，kernel-vm 虚存管理模块，信号 (signal) 处理模块，easy-fs 文件系统，sync 同步模块。monoRCore 支持通过编写少量的内核启动和胶水代码，并设置编译选项，实现对模块的自由组合。下表2.1 展示了每个模块需要额外添加的胶水代码的内容。其中，进行 kernel-alloc 的堆分配器初始化之前，需要进行内核堆空间的初始化。

表 2.1　各模块需要额外添加的内容表

模块名	需要额外添加的内容
syscall	各个系统调用的具体实现
task-manage	进程（和线程）结构体的具体实现；任务管理器初始化；启动程序加载
kernel-context	需要进行内核异界传送门的初始化
kernel-alloc	堆分配器初始化
kernel-vm	页表和页表管理器的具体实现
signal	无
easy-fs	块设备读写的具体实现
sync	无

2.2　monoRCore 中的任务管理

monoRCore 与任务管理和调度相关的代码位于 task-manage 模块中。在这个模块中定义了 2 种任务管理器，分别用于编译时选择不启用线程时的任务管理和启用线程的任务管理。monoRCore 的任务管理器接受使用者额外添加的任务控制块，实现了任务添加、删除和任务调度功能。

2.2.1 任务控制块和任务管理器

任务控制块 (Task Control Block, 简称为 TCB) 是指操作系统为进行任务状态和资源管理，为每个任务记录相关信息的结构体。操作系统会为进程和线程都创建相应的结构体，分别称作进程控制块 (Process Control Block，简称为 PCB) 和线程控制块 (Thread Control Block，简称为 TCB)。

通常来说，PCB 包含此进程序号、其父进程的序号、进程持有的文件描述符、进程的地址空间信息。如果操作系统启用了线程，则还应当包含该进程的所有线程，以及用于线程同步的锁、信号量等资源。如果不启用线程，则应当包含用于任务切换和异常捕捉的上下文信息。启用线程时，用于任务切换和异常捕捉的上下文信息将移至 TCB，并且 TCB 中还应当包含线程序号和其所属进程的序号。

任务管理器需要实现任务池和调度器两部分的功能。任务管理器需要实现任务池的相关功能，即需要存储所有正在运行的进程和线程的控制块，支持通过进程或线程序号查找对应控制块，以及在进程或线程运行结束时释放相关资源，和添加新启动的进程或线程的控制块的功能。任务管理器还要实现一个调度队列，调度队列是一个包含所有未被阻塞的正在运行的任务的序列，任务管理器根据调度队列来判断当前应当运行哪个任务。

2.2.2 monoRCore 中的任务管理器

monoRCore 中的 TCB 需要使用者自行编写，任务管理器会接受使用者编写的 TCB 结构体，完成通用的任务管理操作。不同于通常的操作系统，使用者编写的 TCB 不用包含进程之间父子关系的信息，和进程与线程间的隶属关系。在 monoRCore 中，这两部分信息以关系表的形式存储在了通用任务管理器中。monoRCore 的 task-manage 模块中包含 2 种通用任务管理器。其中，用于不启用线程使用的通用管理器中仅包含进程池以及进程父子关系表。而启用线程使用的任务管理器包含线程池和进程池，以及进程父子关系表和进程线程间的隶属关系表。

monoRCore 的任务管理器对外提供了 3 个任务池的接口：`insert`，`delete` 和 `get_mut`，分别用于添加新任务、删除任务和获取任务的 TCB。这 3 个接口定义在了代码1 的 `Manage` 抽象接口类中。同时，monoRCore 的任务管理器还对外提供了 2 个更新调度队列的接口：`add`, `fetch`，用于将任务添加至调度队列，和取出调度队列中的下一个任务。这两个接口定义在了代码2的 `Schedule` 抽象接口类中。

代码 1 Manage 抽象接口类

```rust
/// Manager trait
pub trait Manage<T, I: Copy + Ord> {
    /// 插入 item
    fn insert(&mut self, id: I, item: T);
    /// 删除 item
    fn delete(&mut self, id: I);
    /// 获取 mut item
    fn get_mut(&mut self, id: I) -> Option<&mut T>;
}
```

代码 2 Schedule 抽象接口类

```rust
/// Scheduler trait
pub trait Schedule<I: Copy + Ord> {
    /// 入队
    fn add(&mut self, id: I);
    /// 出队
    fn fetch(&mut self) -> Option<I>;
}
```

在原有的 monoRCore 中，使用的是简单的时间片轮转 (RR) 算法，即操作系统循环执行调度队列中的任务，每个任务执行在一次循环中均执行一个周期。该调度算法容易实现，但在性能上有不足。因此像 monoRCore 中补充经典调度算法，为使用者提供更多的调度算法选择，是该模块化操作系统的一个重要的完善方向。

2.3 monoRCore 中的存储管理

monoRCore 中的存储管理主要涉及 kernel-alloc 模块和 kernel-vm 模块。其中，kernel-alloc 模块主要涉及内核堆的空间分配和回收，kernel-vm 模块主要涉及任务的虚拟地址空间管理。kernel-vm 模块涉及的功能与页面置换关系更加紧密，将在本节中重点介绍。

kernel-vm 中定义了地址空间结构体`AddressSpace`，用于管理每个任务的虚拟地址空间。`AddressSpace` 中包含该地址空间对应的页表，和任务申请的所有虚存。虚存 (Virtual Memory) 指的是以虚拟地址形式表示的一块存储空间。当运行中的任务需要申请某一块虚存时，需要通过 `AddressSpace` 在其页表中添加该虚存地址的映射，并将这块虚存的虚拟地址区间记录在 `AddressSpace` 中。当运行中的任务需要释放其拥有的某块虚存时，则需要在页表中去除该虚存

虚拟地址的映射，以及 AddressSpace 中这块虚存虚拟地址区间的记录。运行中的任务申请和释放虚存是通过系统调用 (syscall) 实现的，申请内存在 UNIX 中对应 mmap 系统调用，而释放则对应 munmap 系统调用。AddressSpace 同时还提供地址翻译的功能，这样操作系统能根据任务提供的虚拟地址，找到其对应的物理地址，从而实现任务到操作系统的信息传递。另外，进程的创建、执行程序的加载和进程复制的过程中，也伴随着 AddressSpace 的初始化、虚存申请和复制。

由于 monoRCore 是面向操作系统课程实验的教学 OS，monoRCore 中并未实现任何页面置换算法，因此当内核堆空间不足以满足任务的存储需求时会崩溃退出。因此，为 monoRCore 添加页面置换模块，以保障操作系统的运行稳定，是 monoRCore 的一个重要的完善方向。

第 3 章　调度模块的设计与实现

3.1　概述

本文对 monoRCore 中的任务管理模块进行了扩展，使之能支持多种调度算法。为了让调度模块能支持多种调度算法，本文设计实现了 scheduler 模块。本文向 monoRCore 额外添加了 8 种调度算法：最短作业优先算法 (SJF), 最短完成时间优先算法 (STCF)，最高响应比优先算法 (HRRN)，多级反馈队列算法 (MLFQ)，彩票调度算法 (Lottery)，步长调度算法 (Stride)，单调速率算法 (RMS) 和最早截止时间优先算法 (EDF)。这些调度算法对外暴露 scheduler 模块中定义的通用接口，因此使用者可以通过编译选项来选择和更改内核使用的调度算法，而不需要修改其他部分的代码内容。

本章将首先介绍 scheduler 模块的设计，接着将介绍各个调度算法和其在 scheduler 模块中的实现。

3.2　调度模块的设计

3.2.1　与调度相关的系统调用

不同于简单的 RR 算法，复杂的调度算法往往会在任务运行状态发生变化时，对调度队列进行更新，并根据任务之前执行的历史信息，来选择当前应当优先执行的算法。在本小节中，将介绍用于任务主动触发自身运行状态变化的系统调用。

首先，是与进程相关的 `fork` 和 `exec` 系统调用。`fork` 调用用于对当前进程进行复制，复制时除了对当前进程的资源进行复制外，复制得到的新进程的当前执行位置也与被复制进程当前的执行位置保持一致。因此，在进程复制时，调度器需要对任务历史信息进行复制。`exec` 调用用于让当前进程执行给定的程序。由于 `exec` 调用完成后，该进程将从头执行某个新程序，所以调度器需要对其记载的任务历史信息进行清零。

接着，是与任务状态变化相关的 `sleep`、`yield` 和 `exit` 调用。`sleep` 和 `yield` 都是任务当下主动放弃继续执行。其中，`sleep` 调用中指定了任务“休眠”的时间，在“休眠”时间内，任务不会被加入调度队列。而 `yield` 调用只是放弃了当下继续执行，任务会被立即重新加入调度队列中。`exit` 用于任务的退出，这时调度器需要释放用于存储其相关历史信息的内存。

3.2.2 对外接口设计

在 scheduler 模块中，与调度相关的任务历史信息是存在各个调度器内部的。这样做使得使用者能在不修改其补充的 TCB 结构体代码的前提下，随意选择内核使用的调度算法。因此，scheduler 模块为 `Schedule` 抽象接口类添加了在任务执行与调度相关的系统调用时，进行相关历史信息更新的接口。这些接口被称为系统调用钩子 (syscall hook)，它们在相关系统调用执行成功后被调用，由 scheduler 模块提供具体实现。

在系统调用钩子之外，scheduler 模块还为 `Schedule` 抽象接口类补充了内核钩子（kernel hook）。内核钩子在内核开始执行某个任务和暂停某个任务的执行时被调用，调用时向调度器提供了开始执行（或停止执行）时的时间戳 (timestamp)，以便于调度器计算该任务的执行时间等信息。

代码块 3 是改进后的 `Schedule` 抽象接口类。

代码 3 改进后的 Schedule 抽象接口类

```rust
pub trait Schedule<I: Copy + Ord> {
    /// 将任务加入调度队列
    fn add(&mut self, id: I);
    /// 查找当前应当运行的任务，并移出调度队列
    fn fetch(&mut self) -> Option<I>;

    /// exec 系统调用钩子
    fn update_exec(&mut self, id: I, args: &ExecArgs);
    /// fork 系统调用钩子
    fn update_fork(&mut self, parent_id: I, child_id: I);
    /// sleep 系统调用钩子
    fn update_sleep(&mut self, id: I);
    /// 内核钩子，在任务即将开始执行时调用
    fn update_sched_to(&mut self, id: I, time: usize);
    /// 内核钩子，在任务停止执行时调用
    fn update_suspend(&mut self, id: I, time: usize);
}
```

其中，`update_exec` 函数接受了 `ExecArgs` 类型的参数。部分调度算法需要提供如任务的预期执行时间等参数，这些参数由应用程序在进行 `exec` 系统调用时告知调度器。由于各个调度算法用到的参数不一样，因此每个调度算法需要定义自己的 `ExecArgs` 结构体。另外，`Schedule` 抽象接口类中没有上文提到的 `exit` 和 `yield` 两个系统调用的相关钩子。对于 `exit` 系统调用来说，这是由于其执行完毕后会调用 `Manage` 抽象接口类的 `delete` 接口，完成其历史

信息和 TCB 占用的空间释放。对于 `yield` 系统调用来说，其实现是暂停当前任务的执行并将该任务重新加入调度队列，在暂停当前任务的执行的过程中会触发 `update_suspend` 钩子，调度器的相关更新会在该钩子中完成。

scheduler 模块中实现的每个调度算法，都各自实现了 `Schedule` 抽象接口类和 `Manage` 抽象接口类。下图 3.1 以包含线程的通用任务管理器为例，展示了完善后的 task-manage 模块的架构，其中线程管理器的部分由 scheduler 模块实现。

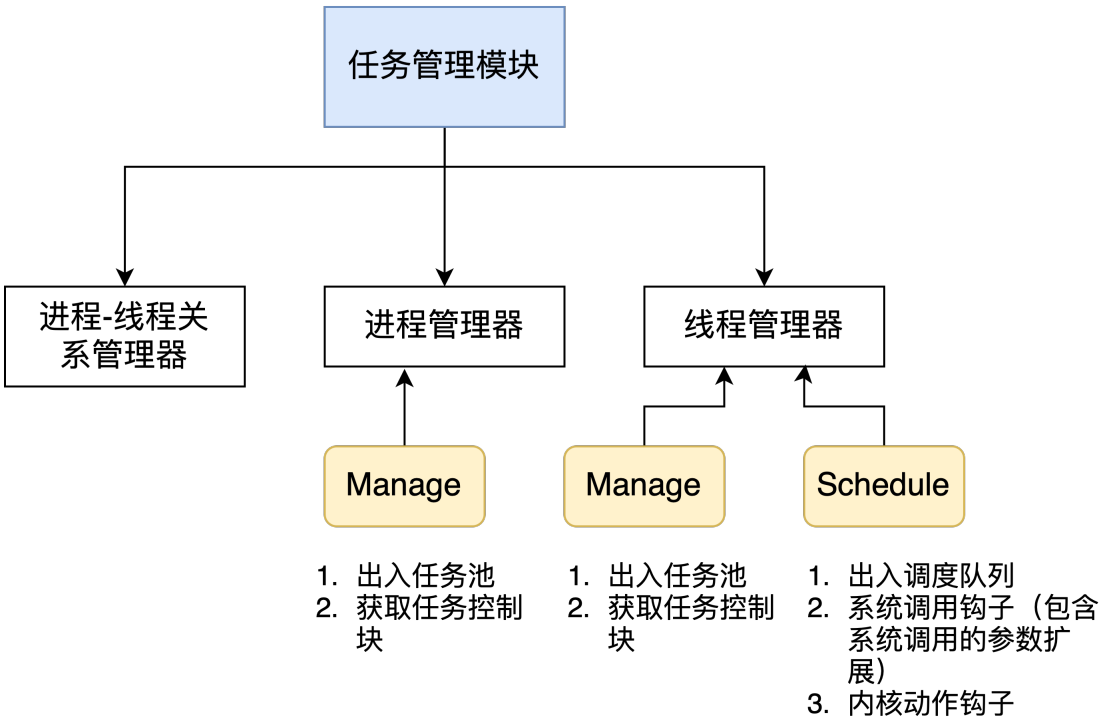


图 3.1　task-manage 模块的架构

3.2.3　内部成员

前文中提到，与调度相关的历史信息是存在调度器内部的。因此，scheduler 模块中一个典型的调度器大致包含如下几个内部成员：任务池，任务历史信息表，调度队列以及当前正在执行任务的相关信息。其中，任务池中包含了当前所有未完成任务的 TCB，任务历史信息表则记录了当前所有未完成任务的补充信息。补充信息包括任务总运行时间，任务优先级等。正在执行任务的相关信息包括正在运行任务的序号，该任务何时开始运行等。

3.3 各调度算法的实现

3.3.1 批处理系统的调度

在批处理操作系统中，用户将一批任务提交给操作系统，之后不再干预任务的执行（即不进行交互）。批处理系统没有交互性，常用于以科学计算为主的大型计算机上。先来先服务算法（First Come First Serve) 是最简单的批处理系统调度算法，它按照用户提交任务的顺序依次执行各个任务。在 scheduler 模块中实现的最短作业优先 (SJF)，最短剩余时间优先 (STCF) 和最高响应比优先 (HRRN) 算法也是用于批处理系统的调度算法。

SJF 算法需要用户给定每个任务的预期执行时间，操作系统按照预期执行时间最短到最长的顺序，依次执行各个任务。该调度算法不需要记录任务运行的其他信息。实现时使用二叉堆数据结构来存储调度队列，使得调度器 `Schedule` 抽象接口类的 `fetch` 操作的复杂度是 $O(1)$，`add` 操作的复杂度是 $O(\log n)$，n 为当前未完成的任务数（下同）。

STCF 算法也需要用户给定每个任务的预期执行时间 $\hat{T}$，同时需要调度器记录每个任务的累计运行时间 t_r。根据公式 $\hat{t}_l = \hat{T} - t_r$ 得到每个任务预期的剩余执行时间 $\hat{t}_l$。STCF 算法每次调度时，选择执行预期剩余执行时间 $\hat{t}_l$ 最小的任务。在 scheduler 模块中，调度队列也是使用二叉堆来完成的。通过 `Schedule` 抽象接口类的两个内核钩子，调度器可以计算出任务在单次调度中的运行时长，将单次运行时长累加可以得到累计运行时长。

HRRN 算法每次调度时选择响应比 R 最高的任务。其中，响应比的计算公式是：

$$R = \frac{t_{wait} + t_{pred}}{t_{pred}} \tag{3.1}$$

其中, t_{wait} 是任务的总等待时间，t_{pred} 是任务的预期运行时间。因此，HRRN 算法需要用户给定任务的预期运行时间。调度器会通过任务上次停止的时间，计算出这一次等待的时间，并和之前记录的等待时间相加，得到总等待时间。调度器每次进行 `fetch` 操作时，都需要对每个任务计算其总等待时间，并根据响应比选择应当执行的任务。因此 `fetch` 操作是 $O(n)$ 的，`add` 操作则是 $O(1)$ 的。

3.3.2 交互式系统的调度

交互式操作系统允许用户通过输入来与运行中的任务进行交互，也支持网络设备。由于交互式操作系统中大量交互式任务需要留在任务池中，等待用户或者

网络设备的输入，而输入到来的时间很难确定，预测交互式任务的总执行时间是不可行的。同时，交互式任务和后台任务具有不同的运行特点。交互式任务往往会主动放弃继续执行，通过休眠等待用户输入，而当用户输入到来时，则期望能尽快响应用户操作，避免用户感觉卡顿。

多级反馈队列 (MLFQ) 算法是基于交互式任务的运行特点设计的调度算法。MLFQ 算法将任务分为交互式任务和后台任务两种，并将交互式任务置于更高的优先级，每次调度时优先执行交互式任务，只有当所有交互式任务都休眠时才执行后台任务。其调度器通过观察任务在执行时是否频繁出现放弃执行的操作，来判断该任务是否属于交互式任务，不需要用户提供任务类型信息。

MLFQ 调度器中包含多个队列，每个队列对应一个优先级，其中放置属于该优先级的所有任务。每次调度时，选择最高优先级对应的队列，然后循环执行其中的任务。在 scheduler 模块中，MLFQ 调度器会通过 `Schedule` 抽象接口类中的 `sleep` 系统调用钩子，记录当前任务在执行过程中是否出现主动休眠，如果该任务是通过主动休眠结束本次执行，则该任务的优先级保持不变，否则该任务在本次执行后的优先级下降 1 级。一个任务在创建时会被置于最高优先级，对于后台任务，其优先级会逐渐下降至最低优先级，而交互式任务能保持较高的优先级。MLFQ 调度器的 `fetch` 和 `add` 操作都是 $O(1)$。

3.3.3 公平共享调度

由于多用户操作系统的出现，平衡不同用户和不同进程的资源分配，保障资源分配公平成为了一个重要需求。由此诞生了公平共享调度 (Fair Share Scheduling，简称为 FSS) 策略。FSS 算法中任务被分配到的执行时间与任务的优先级成正比。scheduler 模块中实现的步长 (Stride) 调度算法和彩票 (Lottery) 调度算法就是两种经典的 FSS 算法。

Stride 调度算法定义每个任务的步长 s 为一个与任务优先级 p 成反比的值，每次调度时任务的总步长都增加 s，调度器总选择步长最小的任务来执行。在 scheduler 模块中，定义了一个数值较大的无符号整数`BIG_STRIDE`，每个任务的步长为 $s = \frac{BIG\_STRIDE}{p}$。Stride 调度器的调度队列同样使用二叉堆实现，并且需要用户指定任务的优先级。

Lottery 调度算法给每个用户分配一定数量的彩票，用户可以将这些彩票分配给其创建的每个任务。每次调度时，Lottery 调度器进行一次抽奖，并执行拥有中奖号码的彩票的任务。在 scheduler 模块中，Lottery 调度器记录了每个任务拥有

的彩票数，以及调度队列中所有任务拥有的彩票总数。每次调度时，根据生成不大于彩票总数的正整数随机值，再遍历调度队列，找到拥有此彩票的任务。因此，调度器的 `fetch` 操作是 $O(n)$ 的，而 `add` 操作是 $O(1)$ 的。实际上，如果使用平衡二叉树来查找拥有对应彩票的任务，则 `fetch` 操作的复杂度可降为 $O(\log n)$，而 `add` 操作的复杂度提高至 $O(\log n)$。在 scheduler 模块中，每个任务拥有的彩票数是用户 `exec` 系统调用指定的。

3.3.4 实时操作系统的调度

实时操作系统 (Real Time Operating System，简称 RTOS) 是指当外界事件或数据产生时，能以最快速度处理，并在规定时间内完成响应的操作系统。RTOS 中运行的任务，在外界事件或数据到来时，如果不能在规定的时间段内完成响应，则会导致严重错误或者不良影响。由于外界事件或数据的到来往往具有周期性，RTOS 将其上运行的任务简化为周期性任务来进行调度选择。RTOS 上同时有多个任务等待执行，RTOS 的调度算法需要根据任务的周期和规定的响应时间，协调各个任务的运行时间，减少任务响应超时的次数。scheduler 模块中实现的最早截止时间优先 (EDF) 和单调速率 (RMS) 调度算法，就是 RTOS 的经典调度算法。

EDF 算法要求用户指定任务运行的周期，以及截止时间 (deadline)，即需要完成响应的时间段长度。在每次调度时，EDF 调度器选择截止时间最近的任务进行调度。在 scheduler 模块中，调度队列是用二叉堆实现的，`fetch` 操作的复杂度是 $O(1)$，`add` 操作的复杂度是 $O(\log n)$。

RMS 算法只要求用户指定任务运行的周期，每次调度时总选择周期最短的任务进行执行。这样调度的好处是可以避免周期较短的任务多次超时，出现任务堆积。在 scheduler 模块中，调度队列也是用二叉堆实现的。

第 4 章　页面置换模块的设计与实现

4.1　概述

本文向 monoRCore 中添加了页面置换模块 frame-manage ，并在页面置换模块中实现了 6 种经典页面置换算法：先进先出 (FIFO) 算法，时钟 (Clock) 算法，改进的时钟算法，缺页率 (PFF) 算法和工作集 (Work Set) 算法。这些页面置换算法对外暴露相同的接口，使用者可以通过更改编译选项选择内核使用的页面置换算法，也可以选择不启用任何页面置换算法。

本章将首先介绍页面置换模块的设计，包括 frame-manage 模块对外暴露的接口，和 frame-manage 模块与其他模块之间的依赖关系等内容。本章接着将介绍 frame-manage 模块中各个页面置换算法及其实现。

4.2　页面置换模块的设计

4.2.1　概述

页面置换的相关定义和有关背景知识已经在前文中介绍，本小节中不在赘述，只针对前文中的页面置换流程进行总结。下图 4.1 是触发内核页面置换操作的流程示意图。

页面置换模块主要需要实现的是硬件抛出缺页 (Page Fault) 异常时的处理过程。当出现缺页时，页面置换模块需要检查用于分配页面的内存是否充足，从当前内存上的所有页面中选择一个或多个页面，将其数据置换到外部设备上，并从外部设备上取出对应页面数据，将并其置换到内存中。

为对内存中的页面的查找、内存中页面的空间分配和回收，frame-manage 模块中实现了用于任务页面分配的 `FrameAllocator`。monoRCore 原有的用于内核内存分配的 kernel-alloc 模块仍被保留，但 kernel-alloc 中的分配器将只用于内核堆空间的分配，仅处理内核自身运行时的空间需求。

为方便实现数据在外部设备和内存之间的置换，frame-manage 模块中实现了管理置换操作的 `IdeManager`。`IdeManager` 内部管理着页面数据在外部设备上的存储位置，支持通过任务序号和对应的虚拟地址查找对应页面数据的存储位置。

frame-manage 模块中为页面置换这一操作设计了通用的接口，这一部分将在

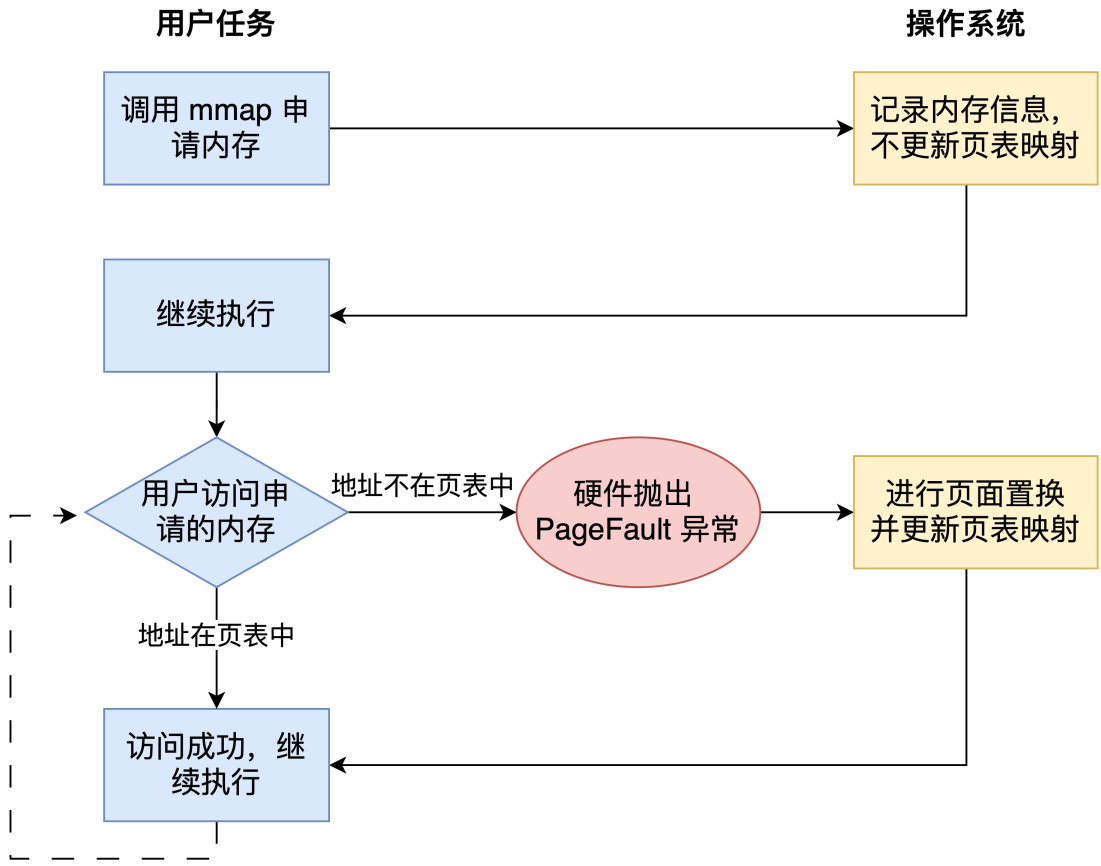


图 4.1　页面置换流程图

下一小节具体介绍。

4.2.2　接口设计

用于任务分配的 FrameAllocator 实现了frame_alloc，frame_dealloc 和 frame_check 三个功能接口，分别用于为页面申请内存，释放页面占用的内存，和检查是否仍有内存用于页面分配。这三个功能接口都只在 frame-manage 模块内使用，不对外暴露。FrameAllocator 需要在内核启动时进行初始化，其初始化接口 init 对外暴露。内核启动时，需要调用 init 以设定页面分配的空间的起始和结束位置的物理地址。

IdeManager 实现了swap_out，swap_in，check和clear_disk_frames 四个功能接口，分别用于：从外部设备中换出页面数据，将页面数据换入外部设备中，检查对应页面的数据是否在外部设备中和释放某任务在外部设备上的所有页面空间。这四个功能接口都只在 frame-manage 模块的内部调用。

frame-manage 模块中为每个页面置换算法实现了页面置换处理器，这些处理器实现了相同的接口，因此使用者可以根据编译选项任意选择内核的页面这换算法。页面置换处理器中记录了所有内存中页面的相关信息，包括每个页面的物理地址，其所属任务的序号以及对应的虚拟地址。在进行页面置换处理器的通用接口设计时，frame-manage 模块中将页面置换操作细分为两个阶段。在第一阶段中，页面置换处理器根据其记录的所有页面信息，选择需要被替换出内存的所有页面，并完成数据换出操作；在第二阶段中，通过 FrameAllocator 为换入的页面分配内存，并完成对页面置换处理器内部信息的更新。

代码4展示了页面置换处理器的通用接口设计。

代码 4 页面置换接口设计

```
pub trait Manage<Meta:VmMeta, M:PageManager<Meta>+'static> {
  fn new() -> Self;

  fn handle_pagefault<F>(&mut self,
                get_memory_set: &F, vpn: VPN<Meta>,
                task_id: usize)
  where F: Fn(usize) -> &'static mut AddressSpace<Meta, M>;

  fn work<F>(&mut self, get_memory_set: &F, task_id: usize)
    -> Vec<(PPN<Meta>, VPN<Meta>, usize)>
  where F: Fn(usize) -> &'static mut AddressSpace<Meta, M>;

  fn insert_frame(&mut self, vpn: VPN<Meta>,
                ppn: PPN<Meta>, task_id: usize,
                frame: FrameTracker);

  // clear all frames related to certain task
  // called when the task exits
  fn clear_frames(&mut self, task_id: usize);

  fn handle_time_interrupt<F>(&mut self, get_memory_set: &F)
  where F: Fn(usize) -> &'static mut AddressSpace<Meta, M>;
}
```

其中，work 接口对应第一阶段的选择被替换页面的操作，insert_frame 接口对应更新页面处理器内部信息的更新。各处理器的 handle_pagefault 接口使用相同代码实现，在该接口的代码中会调用 work 和 insert_frame 接口。clear_frames 接口用于在任务结束时释放任务在内存中的页面空间，handle_time_interrupt接口用于在硬件时钟中断 (Time Interrupt) 出现时

进行处理器的相关信息更新。

frame-manage 模块对外只提供处理缺页异常的 handle_pagefault 接口，在任务结束时释放任务在内存和外部设备上所有页面的 del_memory_set 接口，和在时钟中断出现时进行信息更新的时钟中断钩子 time_interrupt_hook。其中，del_memory_set 接口中会调用页面置换处理器的 clear_frames 和 IdeManager 的 clear_disk_frames 接口。handle_pagefault 接口和 time_interrupt_hook 接口会调用页面置换处理器的对应接口。

4.2.3 与其他模块之间的依赖关系

在进行页面置换时，需要在任务页表中加入换入内存的页面的地址映射，并删除被换出内存的页面的地址映射，以防止任务访问到错误数据。而任务虚拟存储和页表的管理是在 kernel-vm 模块中实现的，因此 frame-manage 模块依赖于 kernel-vm 模块的实现。

同时，由于每个任务的虚存管理器 AddressSpace 位于其 PCB 中，修改对应任务的页表需要首先获取其 PCB，而 PCB 是由 task-manage 模块持有并管理的，因此 frame-manage 模块也要依赖于 task-manage 模块的实现。

4.3 各页面置换算法的实现

4.3.1 局部页面置换算法

局部页面置换 (Local Page Replacement) 算法为每个任务分配相同的页面数目，当发生页面置换时，只选择正在执行的任务的页面中的一个进行替换。frame-manage 模块中实现的先进先出 (FIFO）算法，时钟 (Clock) 算法和改进时钟算法都属于局部页面置换算法。

FIFO 算法进行页面置换时，总选择该任务最早被映射的页面进行替换。frame-manage 模块的 FIFO 页面置换处理器为每个任务都创建了一个队列，每次在内存中创建新页面时，新页面的相关信息会被加入队列末尾。每次需要进行页面置换时，都取出队列第一项的页面进行替换。

时钟算法用一个如图 4.2 所示的环形缓冲来存储每个任务位于内存中页面的信息，并用一个指针来记录队头的位置。每次需要进行页面置换时，都从队头指针开始遍历每一个页面的访问情况，选择第一个未被访问的页面进行替换，并将这之前遇到的所有页面的访问位清零，如果环形缓冲的每个页面都被访问过，则

选择原队头指针指向的页面。之后将队头指针指向被替换页面位置的下一位置。

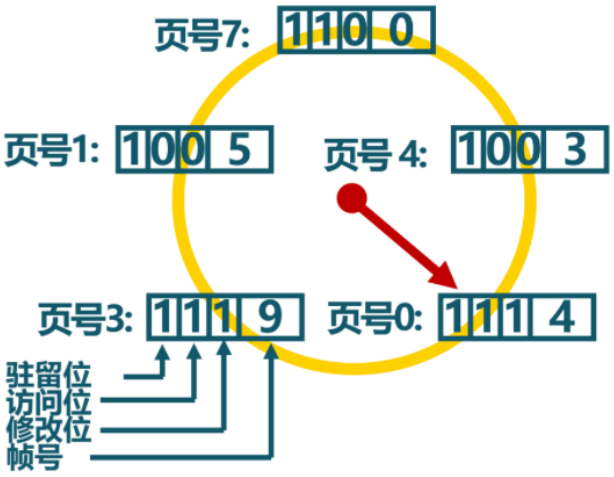


图 4.2　环形缓冲示意图

改进的时钟算法与时钟算法类似，不同在于改进的时钟算法考虑了页面被修改带来的额外开销。由于被修改过的页面进行页面置换时需要重新写入，而未被修改的页面无需进行重新写入，所以选择被修改过的页面进行页面置换的成本更高，应当优先选择未被修改的页面。改进的时钟算法每次置换首先通过遍历查找是否有未被访问且未被修改的页面，如有则选择第一个此类型的页面。接着查找是否有未被访问但被修改过的页面，如有则选择第一个此类型的页面，如无则继续查找是否有被访问但未被修改的页面，最后才选择队头指针指向的页面。查找完成后，清零访问位而不清零修改位。

monoRCore 使用 SV39 分页方案，在 SV39 规则下的页表中，每一个页表项 (页表中一个映射) 包含了记录页表项是否被访问过的访问位 (Bit)，以及是否被修改过的修改位。当任务访问某一虚拟地址时，硬件会将其页表中对应页表项的访问位置为 1，修改某一虚拟地址上的数据时同理。

4.3.2　全局页面置换算法

全局页面置换 (Global Page Replacement) 算法在发生缺页时，可以选择内存中所有的页面中的任意个页面，将其置换到外部设备中，并将目标页面从外部设备中取至内存。在全局页面置换算法中，每个任务在内存中的页面数无上限。frame-manage 模块中实现的缺页率 (PFF) 算法和工作集 (Work Set) 算法就属于全局页面置换算法。

工作集算法记录在最近固定个时钟周期内被访问的页面，这些页面的集合被

称为工作集。图 4.3 展示了工作集的计算方式，图中 Δ 是工作集记录的时间间隔长度，在此图中，t 时刻的工作集为 $\{7,5,1,6,2,3,4\}$。每次缺页时，无论是否仍有内存用于分配新页面，总将在最近固定个时钟周期内未被访问的页面换出内存。在 frame-manage 模块中，工作集页面置换处理器内部记录了最近几个时钟周期内被访问的页面，并通过实现 `handle_time_interrupt` 接口，实现记录当前时钟周期内被访存的页面的功能。处理器中使用了环形数组来记录最近多个周期的被访问的所有页面，环形数组的每一项是一个向量，记录某一个时钟周期内被访问的所有页面。

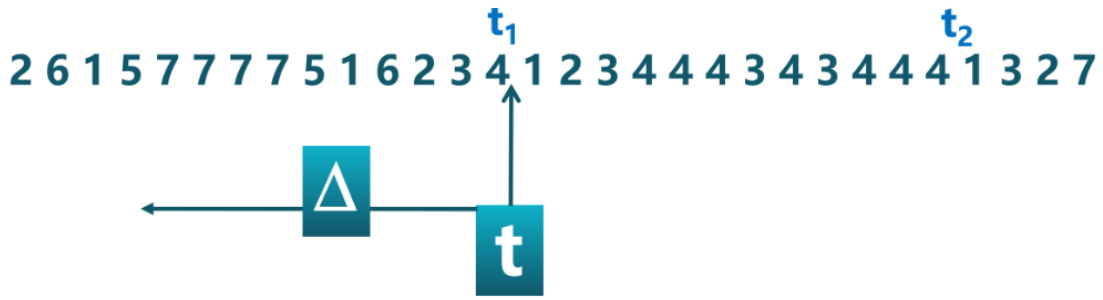


图 4.3　工作集计算示例

PFF 算法是对工作集算法的一种拓展。工作集算法静态的将最近某时间段内被访问的页面留在内存中，并将其他页面换出内存，而 PFF 算法会记录操作系统的缺页率，当缺页率较小时将最近未访问的页面换出，而当缺页率过大时则尽可能只将页面换入而不将任何页面换出内存。具体实现上，给定某个时间差 Δt，每次缺页时，计算当前时间 t_{cur} 与上次缺页时间 t_{last} 的差值，如果 $t_{cur} - t_{last}$ 大于等于 Δt，则将最近未访问的页面换出。否则，如果操作系统仍有多余内存用于页面分配，则不换出任何页面。

无论是工作集算法还是 PFF 算法，当最近某时间段内所有页面都被访问过，则不会选择任何页面进行置换。但可能同时出现所有页面都被访问，且操作系统内存不足的情况。对于这种情况，frame-manage 模块中会选择使用 FIFO 策略进行处理。

第 5 章　调度算法的评测

5.1　测试环境与测例

本章节介绍对 scheduler 模块中实现的所有调度算法的评测。实现的调度算法包括：批处理系统调度算法 SJF、STCF 和 HRRN，交互式系统调度算法 MLFQ，FSS 调度算法 Stride 和 Lottery，以及 RTOS 调度算法 RMS 和 EDF。本文对每一类调度算法分别进行了评测和分析。每一类调度算法的测例将在下一节中具体介绍。评测结果证明了各算法实现的正确性，并体现了各个算法的特点。

本章节中所有测试均在单核 QEMU 模拟器上运行。测试时，内核均开启了线程功能，并不启用页面置换功能。

5.2　测试结果和分析

5.2.1　SJF, STCF 与 HRRN 调度算法

对批处理系统的测试包含五个测例，它们的预期运行时间如下表 5.1。在操作系统内核启动完成后，会创建并运行初始任务 `INIT_PROC`，`INIT_PROC` 中会使用`exec` 命令运行测试脚本。测试脚本中会一次为每个测例创建进程，每个测例的创建时间也如下表 5.1 所示。其中，创建时间是以 `INIT_PROC` 执行至创建第一个测例的进程的时刻为零点得出的。下文中的测试启动方式也如前所述。

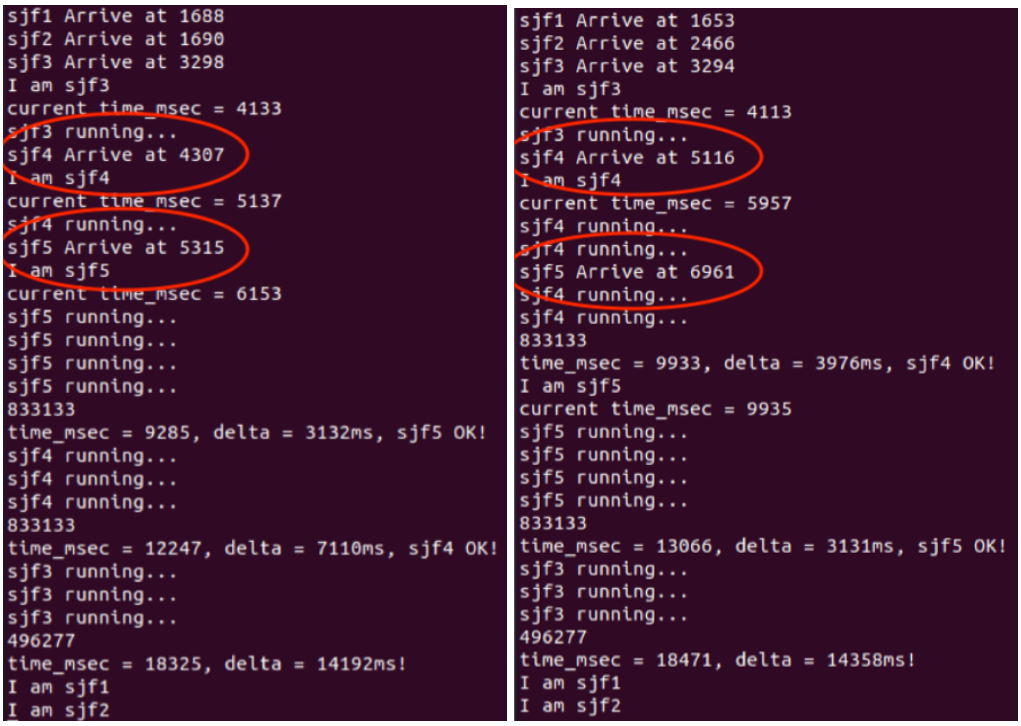
表 5.1　批处理系统测例

测例名	预期运行时间 (ms)	创建时间 (ms)
sjf1	10000	0
sjf2	100000	0
sjf3	3000	0
sjf4	1200	1000
sjf5	1100	2000

从测例内容上来看，sjf1 和 sjf2 是平凡的，它们在执行一次输出后，进行对应时长的休眠。sjf3，sjf4 和 sjf5 中都进行了通过递推求斐波那契数列第 n 项除以 1000007 的余数的运算，并在完成运算后输出了对应结果和运算耗时。n 为较大

的常数，在每个测例代码中具体指定。

下图 5.1 中展示了测试在内核分别使用 SJF 和 STCF 调度算法时的执行结果。



(a) SJF 算法测试结果　　(b) STCF 算法测试结果

图 5.1　SJF 和 STCF 算法测试结果

对于 SJF 算法，在 sjf1，sjf2，sjf3 同时到达时，由于 sjf3 的预期执行时间最短，所以执行 sjf3。当 sjf4 在随后到达时，由于其预期执行时间更短，因此发生抢占现象，即 sjf3 的执行中断，转而执行 sjf4。当 sjf5 最后到达时，同样发生抢占。图 5.1(a) 中用红圈标出了两次抢占的位置。

不同于 SJF 算法，STCF 算法需要计算预期剩余执行时间 $t_{left} = t_{pred} - t_{run}$，并总选择预期剩余执行时间最短的任务进行执行。当 sjf1，sjf2，sjf3 同时到达时，它们的总运行时间均为 0 ms，预期剩余执行时间分别为 10000 ms，100000 ms 和 3000 ms，因此执行 sjf3。当 sjf4 在 sjf3 运行了 1000 ms 后到达时，sjf3 的预期剩余时间为 2000 ms，而 sjf4 的预期剩余时间为 1200 ms，因此发生抢占，改为执行 sjf4。当 sjf5 在 sjf4 运行了 1000 ms 后到达时，sjf4 的预期剩余时间为 200 ms，而 sjf5 为 1100 ms，因此选择预期剩余时间更短的 sjf4，不发生抢占。图 5.1(b) 中用红圈标出了 sjf4 和 sjf5 到达时的情况，不同于 SJF 算法，STCF 算法中只发生一次抢占。

HRRN 算法总选择响应比最高的任务运行。下图 5.2 展示了测试在内核使用

HRRN 算法时的执行结果。任务的响应比按照前文中的公式3.1计算：$R = \frac{t_{wait}+t_{pred}}{t_{pred}}$，其中，$t_{wait}$ 是任务的总等待时间，t_{pred} 是任务的预期运行时间因此，随着任务的总等待时间的变化，任务的响应比 R 会逐渐增大，使其在调度队列中的位置逐渐靠前。因此在图 5.2 中，会出现 sjf4，sjf5 交替执行的情况。由于 sjf3 的预期执行时间更长，所以需要等待更长的时间才能和 sjf4、sjf5 的响应比相近，因此 sjf3 是最后执行的。

```
sjf1 Arrive at 1679
sjf2 Arrive at 2566
sjf3 Arrive at 3376
I am sjf1
I am sjf2
I am sjf3
current time_msec = 4209
sjf3 running...
sjf4 Arrive at 5213
I am sjf4
current time_msec = 6376
sjf4 running...
sjf5 Arrive at 7041
sjf4 running...
sjf4 running...
I am sjf5
current time_msec = 9128
sjf5 running...
sjf4 running...
sjf5 running...
sjf5 running...
833133
time_msec = 12161, delta = 5785ms, sjf4 OK!
sjf5 running...
sjf3 running...
833133
time_msec = 13926, delta = 4798ms, sjf5 OK!
sjf3 running...
sjf3 running...
496277
time_msec = 18479, delta = 14270ms!
```

图 5.2　HRRN 算法测试结果

5.2.2　MQ 与 MLFQ 调度算法

多级队列 (MQ) 中给每个任务拥有静态的优先级，该优先级在任务创建时由用户指定。而 MLFQ 算法中，任务具有动态的优先级。MLFQ 算法根据任务是否出现频繁的休眠来改变任务的优先级，使得不休眠的任务的优先级随着其执行时间的增长逐渐下降，从而满足前台任务响应低延迟的需求。

对 MLFQ 算法的测试中包含 3 个测例：mlfq3，mlfq4 和 mlfq5。这 3 个测例的计算内容一致，均为递推计算斐波那契数列的第 600000 项除以 1000007 的余数，并将该结果输出。不同之处在于，mlfq4 测例中，模拟前台交互式任务，在运算完固定项之后通过 `sleep` 系统调用放弃继续执行，进入休眠状态。而 mlfq3 和 mlfq5 类似于后台任务，不进行此操作。

图 5.3 中展示了测试在内核使用 MLFQ 调度算法时的测试结果，红线处标出了各任务执行结束的位置。可以看出由于 mlfq4 测例模拟了前台交互式任务，因此保持了较高的优先级，并对 mlfq3 形成抢占，最终运行结束时间先于模拟后台任务的 mlfq3 和 mlfq5。

图 5.3　MLFQ 调度算法测试结果

该测试在反应了 MLFQ 算法实现的正确性之余，还展现了 MLFQ 算法的一个漏洞：任务可以通过刻意的主动休眠，使得自身保持高优先级，抢占处理器资源，使得其他任务长时间等待。

5.2.3　Stride 与 Lottery 调度算法

对公平共享调度 (FSS) 算法的测例包含 6 个测例，6 个测例的优先级依次为 5 到 10。每个测例运行相同的时长，测例中粗略的测量了在这段时间内被调度次

数，并在运行完成后，输出测例的优先级、被调度次数以及被调度次数与优先级之比。

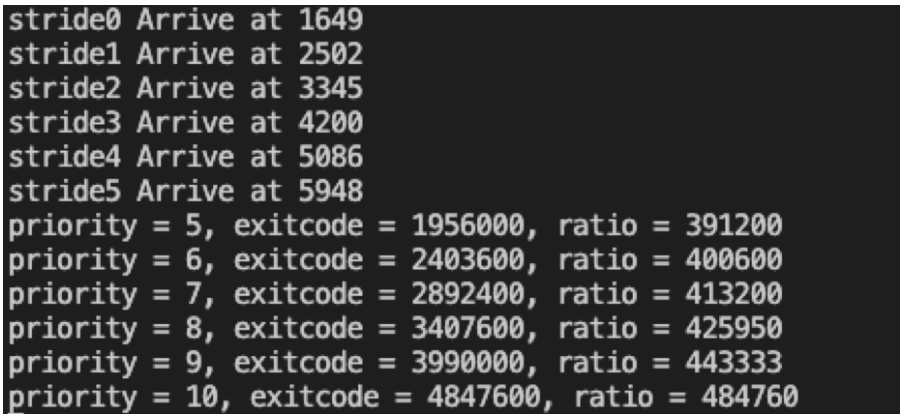
测例中调度次数的测量的具体实现如下代码块 5。操作系统进行任务调度时是以时间片 (Time Slot）为单位进行的，即每隔一个固定时间，操作系统进行一次任务调度。每个时间片中，`count_during` 中的循环能执行的次数基本相同。因此，可以通过记录循环的执行次数，间接计算出每个任务被运行了多少个时间片，从而得到在 `MAX_TIME` 时间段中该任务被调度的总次数。

代码 5 测例中调度次数测量的具体实现

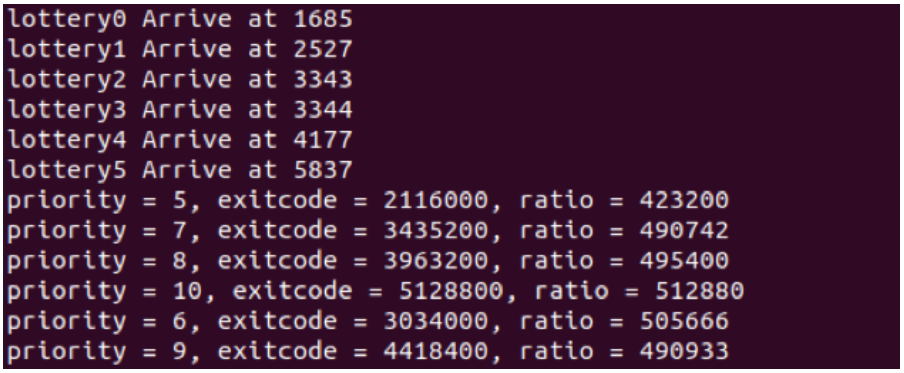
```
fn spin_delay() {
    let mut j = true;
    for _ in 0..10 {
        j = !j;
    }
}

// total run time(ms, including waiting)
const MAX_TIME: isize = 40000;
pub fn count_during(prio: isize) -> isize {
    let start_time = get_time();
    let mut acc = 0;
    // set_priority(prio);
    loop {
        spin_delay();
        acc += 1;
        if acc % 400 == 0 {
            let time = get_time() - start_time;
            if time > MAX_TIME {
                return acc;
            }
        }
    }
}
```

前文中提到公平共享调度的特性是任务被调度时间总长与任务优先级成正比，而被调度的总时长与 `count_during` 返回的计数值也成正比，因此预期的实验结果是每个测例输出的计数值与其优先级之比基本一致。

下图 5.4 中展示了测试在内核分别使用 Stride 和 Lottery 调度算法时的执行结果。从执行结果中可看出，各任务计数值与优先级之比（即输出的 `ratio` 部分）基本一致，满足 FSS 算法的特性。



(a) Stride 算法测试结果



(b) Lottery 算法测试结果

图 5.4　FSS 调度算法测试结果

5.2.4　RMS 与 EDF 调度算法

对实时操作系统 (RTOS) 调度算法的测试包含 3 个测例，每个测例内包含一个周期性任务，它们的周期、每次处理的预期时间以及第一个周期的截止期限如下表 5.2 所示。在测试时，执行这些测例的进程被同时创建。

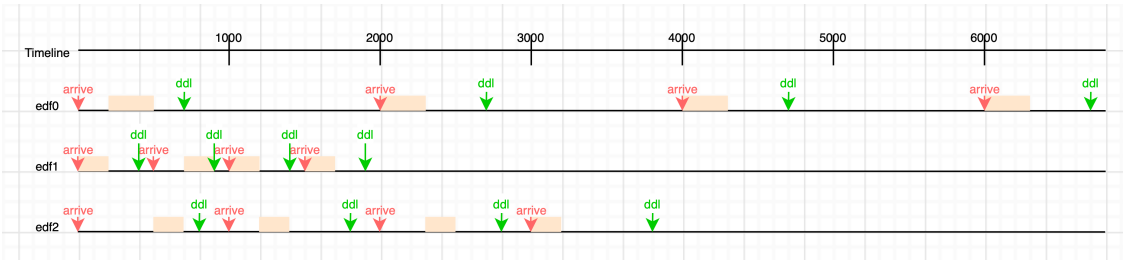
表 5.2　RTOS 系统测例

测例编号	周期 (ms)	单次执行用时 (ms)	首次截止期限 (ms)
0	2000	300	700
1	500	200	400
2	1000	200	800

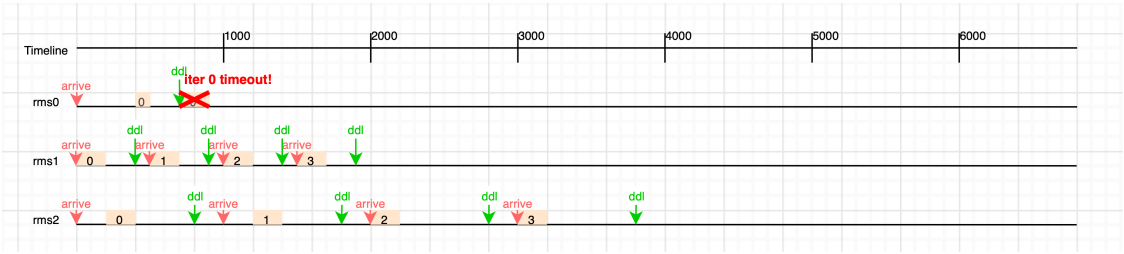
虽然上表 5.2 中包含了单次执行用时参数，但 RMS 和 EDF 调度算法在执行调度时，均不需要此参数。RMS 算法仅需知道任务的周期，而 EDF 算法仅需要知道任务的周期和首次截止期限。单次执行用时这一参数只用于测例判断当前执

行是否超时。

RTOS 系统的每个测例中的任务重复执行 4 个周期，测例中会根据每个周期中任务执行完毕的时间，判断是否出现处理超时。如果出现了超时现象，则该测例将异常退出，不再执行剩余周期。下图 5.5 展示了此测试在分别使用 EDF 和 RMS 算法时的预期执行结果。



(a) EDF 算法预期执行结果



(b) RMS 算法预期执行结果

图 5.5　RTOS 调度算法测试预期执行结果

图 5.5 中时间轴原点为测例进程被创建的时间点，红色箭头为每个周期任务的到达时间（对应 RTOS 中外来事件或数据到达的时间点），绿色为每个周期的截止时间，长条块表示任务被调度运行的时间段。第 i 个周期的截止时间 DDL_i 的计算公式为 $DDL_i = i \times Period + DDL_0$，其中 DDL_0 是第一个周期的截止时间。由图 5.5 的预期执行结果图可以看出，使用 RMS 调度算法时，0 号测例会在第一个周期内出现超时现象，而使用 EDF 调度算法在运行过程中则不会出现超时。

下图 5.6 中展示了测试在内核分别使用 EDF 和 RMS 调度算法时的测试结果。测试结果中展示了每个任务在各个周期的开始执行时间以及执行结束时间。在图 5.6(b) 的 RMS 算法测试结果中的红色报错部分，反应了 0 号任务在首个周期就出现了超时并异常退出，这与预期执行结果相符合。

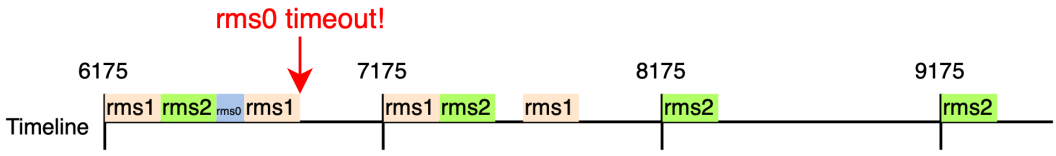
对图 5.6 中的输出结果中的各个时间节点进行可视化，得到图 5.7。从可视化结果中可以进一步看出，EDF 和 RMS 算法的测试结果中，各个任务每个周期执

(a) EDF 算法测试结果　　(b) RMS 算法测试结果

图 5.6　RTOS 调度算法测试结果

行的时间区间与理论分析一致，反应了 EDF 算法和 RMS 算法实现的正确性。

(a) EDF 算法结果可视化



(b) RMS 算法结果可视化

图 5.7　RTOS 调度算法测试结果可视化

第 6 章　页面置换算法的评测

6.1　测试环境与测例

本章介绍对 frame-manage 模块中实现的页面置换算法的测试。frame-manage 模块中共实现了 5 种页面置换算法：先进先出 (FIFO) 算法，时钟算法，改进的时钟算法，缺页率 (PFF) 算法和工作集算法。其中，FIFO 算法、时钟算法和改进的时钟算法属于局部页面置换 (Local Page Replacement) 算法，缺页率算法和工作集算法属于全局页面置换 (Global Page Replacement) 算法。

本文共设计了 7 个测例用于这 5 个算法的测试，其中对 PFF 算法和工作集算法，分别测试了在不同缺页率参数和工作集大小参数下的情况。这 7 个测例的具体内容将在本小节之后的部分介绍。本章所有测试均在单核 QEMU 模拟器中运行，运行时启用了线程功能，并选择了时间片轮转 (RR) 调度算法。本章所有测试中，均设置内核中用于页面分配的内存空间大小为 399×4096 比特，即内存中最多存放 399 个页面，对于局部页面置换算法，限制每个任务在内存中也至多存放 399 个页面。

6.1.1　测试的运行

本小节将介绍测试的启动和运行方式。如前文中对调度算法的测试，当内核完成启动后，会创建并执行 `INIT_PROC` 程序，在本章的测试中，`INIT_PROC` 程序会通过 `exec` 系统调用执行命令行程序 `user_shell`。使用者可以通过 `user_shell` 程序交互式的运行文件系统中预装载的所有测例。

在本节的所有测试中，7 个测例可以被分别执行，也可以按照任意次序依次执行。分别执行是指，每次执行测例时，都需要重新编译内核，并保证该测例是通过 `user_shell` 运行的第一个任务。这样做保证了每个测例在开始运行时，内存中都无任务页面。由于 frame-manage 模块实现了内存和外部设备上的页面回收，任务执行完毕退出后，其位于内存和外部设备上的页面空间都被释放，因此也可以按照任意次序执行。

6.1.2　测例内容

对 frame-manage 模块进行测试的 7 个测例的主要内容均为访存 (包含读和写) 操作。这 7 个测例中按照不同规律，对大小不同的数据空间进行了读写操作。测

试时，记录不同调度算法在完成这些读写操作的过程中，出现的缺页次数，并将其作为评价和比较不同调度算法的指标。记录缺页次数的具体实现将在下一小节中介绍。

在这 7 个测例中，测例 1 到测例 5 都是单进程任务，而测例 6 和测例 7 中都创建了 3 个进程。并且测例测例 1 和测例 2 中的读写操作，都是按照从低地址到高地址的顺序进行的，而测例 3 到测例 7 中都进行了随机访问。

测例 1 中申请了 100 个页面的空间，并在此空间中从低到高依次进行了写入，全部写入后依次进行了读出操作。测例 2 与测例 1 中基本一致，也是从低地址到高地址依次进行了两次访存操作，不同在于测例 2 中申请了 400 个页面的空间。

测例 3，测例 4 和测例 5 也像前两个测例一样，先在其申请的空间中进行了随机写入，之后按照写入的次序读出了被写入位置的数据。这 3 个测例的不同之处在于它们进行选择写入位置时的随机方式不同。

测例 3 中申请了 100 个页面的空间，并在此空间中进行了随机访问。测例 3 中的随机方式是在顺序访问地址序列的基础上，加上在 $[0, 4096)$ 区间上均匀分布的偏移。具体来说，第 i 次写入的地址 $addr_i$ 满足：

$$addr_i = start + i + rand(4096, seed_i),\ \forall i \in [0, len) \tag{6.1}$$

其中，$start$ 是申请的空间的起始地址，len 为申请的空间的大小，$rand(4096, seed_i)$ 表示使用 $seed_i$ 作为种子生成的 $[0, 4096)$ 区间上的随机整数，$rand$ 生成的随机整数符合均匀分布。选择 4096 作为随机整数的上限，是由于 monoRCore 使用的 Sv39 页表规则中规定的页面大小为 4096 比特 (byte)。按照此方式生成的随机地址序列中，相邻两项的地址总属于同一页面或是相邻的页面。该测例中通过使用完全相同的种子序列 $\{seed_i\}$，实现了按照相同随机地址序列将写入内容读出的操作。

测例 4 中申请了 400 个页面的空间，其访存的随机方式是完全随机的，即每次写入的地址 $addr_i = start + rand(len, seed_i)$。其中，$rand$ 的定义与前文一致，len 为申请的空间的大小，在本测例中 len 即为 400×4096。`page_test4` 中的随机写入和对应的读出操作各进行 4096 次。

测例 5 中也申请了 400 个页面的空间。在进行访存操作时，测例 5 将申请的空间均分成 64 个子空间，按照子空间起始地址的高低顺序，依次对每个子空间进行完全随机访问。该测例中对每个子空间进行了 64 次完全随机的写入操作，以及 64 次对应的读出操作，因此共进行了 4096 次随机写入。

测例 6 中创建了 3 个子进程，每个子进程中都申请了 400 个页面的空间，并进行了与测例 5 一致的访存操作。测例 7 中也创建了 3 个子进程，3 个子进程各自申请了 400 个页面的空间，其中 2 个子进程分别按照测例 4 和测例 5 的随机方式进行访存操作，余下的 1 个子进程按照顺序访问的方式进行访存操作。

6.1.3 缺页次数的记录

本章中的所有测试中，均记录执行每个测例时的总缺页次数，作为评测和分析页面置换算法的指标。缺页次数的记载是在内核中进行的。本文在 monoRCore 中添加了静态计数器，用于记载内核启动后出现的缺页的总次数。计数器的初始值为 0。当因硬件抛出的缺页异常时 (包括读缺页异常、写缺页异常和指令缺页异常)，计时器的值加一。当应用退出，即内核接收到 `exit` 系统调用时，内核以调试信息的方式输出计数器的值。通过计算应用启动前后计数器值的差，可以得到运行该测例时出现的缺页次数。

6.2 测试结果和分析

本章中对 frame-manage 模块中实现的 5 种页面置换算法进行了测试，其中对于 PFF 算法，测试了其在设置的缺页时间间隔`PFF_T`分别为 10^5 和 7.2×10^5 时的表现，这两个参数值对应的毫秒数为 8 和 57.6 。对于工作集算法，测试了其在设置的工作集大小`WORKSET_NUM`分别为 5 和 20 时的表现。表 6.1 中展示了各算法设置条件下的测例缺页次数。其中，PFF 算法括号内的参数为缺页时间间隔参数`PFF_T`的值，工作集算法括号内的参数为工作集大小参数`WORKSET_NUM`的值。

从实验结果中可以看出，相比于全局页面调度算法，局部页面调度算法在测试上的整体表现更好。导致这一现象的一个原因是测例中创建的任务进程数相对较少，而全局页面调度算法在任务数量相对较多时效果更为明显。对于本章测试中的测例，局部页面置换算法会将用于页面分配的内存空间完全占满，而全局页面置换算法并没有将这部分空间占满，全局页面置换算法的缺页率也因此升高。另一个原因是 PFF 算法和工作集算法对相关参数的选取都较为敏感，并且 PFF 算法还受到任务执行速度波动的影响。

测例 1 和测例 2 均为顺序访问，而测例 3 中相邻两次访问位置对应的页面总相同或者相邻，也基本上近似于顺序访问，因此各个算法的缺页次数的理论值是容易求得的。对于测例 1，其申请的总页面数量低于内核规定的内存中的页面上

表 6.1　批处理系统测例

算法	测例 1	测例 2	测例 3	测例 4	测例 5	测例 6	测例 7
FIFO	100	800	100	803	800	2400	3700
时钟算法	100	800	100	501	799	2400	3407
改进的时钟算法	100	800	100	803	800	2400	2986
PFF(10^5)	200	800	200	8146	5348	16860	14781
PFF(7.2×10^5)	200	800	200	5274	3148	14415	13679
工作集 (5)	200	800	200	7118	800	2705	9145
工作集 (20)	100	800	100	6326	800	3073	8836

限，因若此使用局部页面置换算法，在进行写入操作时，内核将其申请的空间完整的加载至内存中，之后进行读出操作时不出现缺页，总的缺页次数为 100 次。而对于测例 2，其申请的总页面数超过了规定页面上限，因此无法将申请的空间完整的加载至内存，从而导致读出操作时也出现缺页。对测例 3 的理论分析同测例 1。根据实现的 3 种局部页面置换算法的特性容易得出，测例 2 中每次访存操作均会出现缺页，因此总缺页次数为 800。对于工作集算法，如果`WORKSET_NUM`足够大，使得`WORKSET_NUM`个时钟周期内，能完成所有的写入操作，则测例申请的空间会被尽可能完全加载至内存中。如果`WORKSET_NUM`不够大，则先被写入的页面会被置换出内存。因此在`WORKSET_NUM`为 20 时，测例 1 和测例 3 只出现 100 次缺页，而`WORKSET_NUM`为 5 时，测例 1 和测例 3 都出现了 200 次缺页。对于 PFF 算法的理论分析与工作集算法类似，如果 `PFF_T` 大于测例完成一个整页的写入的时间，则每次缺页时均会将页面加入内存，而不进行置换。但这种情况下 frame-manage 模块中实现的 PFF 算法极易退化为 FIFO 算法，因此测试时 `PFF_T` 并未取得足够大，所以测例 1 和测例 3 都出现了 200 次缺页。由于测例 2 申请的空间总是无法完全加载至内存的，因此对于 PFF 算法和工作集算法的各种设置，其缺页次数总为 800。

从表 6.1 中可以看出，局部页面置换算法在完全随机访问的测例 4 上的表现总体好于局部随机访问的测例 5，而全局页面置换算法在局部随机访问的测例 5 上表现更好。这体现出全局页面置换算法更依赖程序访存规律性的特点。局部页面置换算法中，FIFO 和改进的时钟算法在测例 4 和测例 5 上的表现相近，而时钟算法在测例 4 上的表现好于测例 5。

对于测例 6 和测例 7，其子进程申请的页面空间之和远超内核用于页面分配

的内存上限，使得大量页面需要被存放在外部设备上。使用局部页面置换算法时，由于测例 6 的 3 个子进程按测例 5 的局部随机访问方式进行内存访问，在对每个子空间进行多次写入时不会出现重复缺页，因此测例 6 的总缺页次数为 2400 次。而由于测例 7 中包含进行完全随机访问的子进程，该子进程中同一页面的多次写入 (或多次读入) 可能间隔时间较长，会导致对该页面的多次写入 (或多次读入) 中出现该页面的重复缺失的情况，导致测例 7 的总缺页次数远超 2400 次。使用工作集算法时，同样由于完全随机访问会造成重复缺页，测例 7 的缺页次数远超测例 6。

对于工作集算法，WORKSET_NUM越大，能驻留内存的页面数通常越多，缺页次数也越少，但工作集过大会导致完成页面置换的速度显著下降。同时在WORKSET_NUM达到一定大小时，工作集中的页面数达到内核规定的内存中页面数的上限，此时再增大WORKSET_NUM不会影响缺页次数。

对于 PFF 算法，PFF_T越大，同样使得驻留内存的页面数增多。但如前文所言，当PFF_T过大时，frame-manage 模块中的 PFF 算法会退化为 FIFO 算法，这可能导致缺页次数增多。

插图和附表索引

参考文献

[1] CHEN Y. rcore-tutorial-book-v3 3.6.0-alpha.1[EB/OL]. http://rcore-os.cn/rCore-Tutorial-Book-v3/.

[2] YANG D. rcore-tutorial-in-single-workspace[EB/OL]. https://github.com/YdrMaster/rCore-Tutorial-in-single-workspace.

[3] KUENZER S, BĂDOIU V A, LEFEUVRE H, et al. Unikraft: fast, specialized unikernels the easy way[C]//Proceedings of the Sixteenth European Conference on Computer Systems. 2021: 376-394.

[4] FORD B, BACK G, BENSON G, et al. The flux oskit: A substrate for kernel and language research[C]//Proceedings of the sixteenth ACM symposium on Operating systems principles. 1997: 38-51.

[5] JIA Y. Arceos[EB/OL]. https://github.com/rcore-os/arceos.

附录 A　外文资料的书面翻译

调度器间的战争：FreeBSD 的 ULE 算法 vs Linux 的 CFS 算法

目录

A.1 摘要

这篇文章分析了两个被广泛使用的开源调度器的设计和实现方式对应用性能的影响。这两个开源调度器分别是：FreeBSD 系统的默认调度器 ULE，以及 Linux 系统的默认调度器 CFS。我们在其他方面相同的场景下比较了 ULE 和 CFS 算法。我们将 ULE 移植至 Linux 系统中，并用它来调度原本由 CFS 进行调度的所有线程。我们比较了改成使用 ULE 算法的内核和运行 CFS 的标准 Linux 内核在多种应用上的表现。我们所观察到的差异完全是由调度选择的差异导致的，而和 FreeBSD 与 Linux 之间的差异无关。

在比较实验中没有完全的胜者。在很多任务上两个调度算法的表现相似，但在一些任务上两个算法有着十分显著和令人吃惊的差异。ULE 可能导致饥饿，即使操作系统只运行了一个包含多个完全一样的线程的应用。但 ULE 导致的饥饿在一些任务上可能提高应用的性能表现。CFS 中更复杂的复杂平衡机制能更快的对任务负载变化做出反应，但在较长时间尺度下，ULE 的负载更加平衡。

A.2 引言

操作系统内核调度程序负责保持硬件资源（CPU 核、内存、I/O 设备）的高利用率，同时为对延迟敏感的应用程序提供快速响应时间。操作系统内核必须对工作负载变化做出反应，并以最小的开销处理大量的 CPU 核和线程[15]。本文比较了两种部署最广泛的开源操作系统的默认调度程序：Linux 中使用的完全公平调度程序 (CFS) 和 FreeBSD 中使用的 ULE 调度程序。我们的目标不是宣布总冠军。事实上，我们发现对于某些工作负载，ULE 更好，而对于其他工作负载，CFS 更好。相反，我们的目标是说明两个调度器的设计和实现差异如何反映在不同工作负载下的应用程序性能中。

ULE 和 CFS 都是为在大型多核机器上调度大量线程而设计的。出于可扩展性方面的考虑，两个调度程序都为每个核建立了运行队列。在上下文切换时，核心仅访问自己的运行队列以查找要运行的下一个线程。定期地和在选定的时间，例如，当一个线程被唤醒时，ULE 和 CFS 都会执行负载平衡，即，它们试图平衡在不同核的运行队列中处于等待状态的任务的工作量。

然而，ULE 和 CFS 在设计和实现选择上有很大不同。FreeBSD ULE 是一个简单的调度程序（在 FreeBSD 11.1 中有 2,950 行代码），而 Linux CFS 则复杂得多（在最新的 LTS Linux 内核 Linux 4.9 中有 17,900 行代码）。FreeBSD 运行队列是先

进先出 (FIFO) 的。对于负载平衡问题，FreeBSD 努力使每个核的线程数均匀。在 Linux 中，CPU 核根据其运行队列中线程的先前执行时间、优先级和可感知的缓存行为来决定下一个要运行的线程。Linux 不是平均分配内核之间的线程数，而是努力平均分配等待状态的工作的工作量。

比较 ULE 和 CFS 的主要挑战是应用程序性能不仅取决于调度程序，还取决于其他操作系统的子系统，例如网络、文件系统和内存管理，这些子系统在 FreeBSD 和 Linux 之间也有所不同。为了隔离 CFS 和 ULE 之间差异的影响，我们将 ULE 移植到 Linux，并使用它作为默认调度器来运行机器上的所有线程（包括通常由 CFS 调度的内核线程）。然后，我们比较了带有 ULE 的修改后的 Linux 和使用 CFS 的默认 Linux 内核之间在应用程序上的性能表现。

我们首先通过在单个内核上运行应用程序和应用程序组合来检查 ULE 和 CFS 做出的单核调度决策的影响，然后我们在机器的 CPU 核上运行应用程序，并研究负载均衡器的影响. 我们使用 37 个应用程序，从高性能科学计算 (scientific HPC) 应用程序到数据库。虽然在许多情况下两种算法有着相似的性能，但 CFS 和 ULE 偶尔会有非常不同的行为，即使在在单个应用程序且应用程序每个核心运行一个线程的情况下也是如此。

本文做出以下贡献：

- 我们在 Linux 中提供了 FreeBSD 的 ULE 调度程序在 Linux 上的完整移植版本，并将其作为开源[7]发布。我们的实现 FreeBSD 11.1 版本中包含的所有功能和启发式方法。
- 我们在其他方面相同的环境中比较了 ULE 和 CFS 调度器下的应用程序性能。
- 与 CFS 不同，ULE 可能会无限期地使它认为非交互的线程处于饥饿状态。令人惊讶的是，当系统只执行由相同线程组成的单个应用程序时，也可能发生饥饿。更令人惊讶的是，这种行为实际上对某些任务（例如，数据库）有益。
- CFS 更快地收敛到平衡负载，但从较长时间尺度来看，ULE 实现了更好的负载平衡。
- CFS 用于避免迁移线程的启发式方法可能会损害每个核仅使用一个线程的 HPC 工作负载的性能，因为 CFS 有时会在同一核上放置两个线程，而 ULE 总是在每个内核上放置一个线程。

本文其余部分的概要如下。第 2 节介绍 CFS 和 ULE 调度程序。第 3 节描述

了我们将 ULE 移植到 Linux 以及 ULE 的原本实现和我们的移植版本之间的主要区别。第 4 节介绍了我们实验中使用的机器和任务。第 5 节分析了 CFS 和 ULE 中单核调度的影响。第 6 节分析了 CFS 和 ULE 的负载均衡器。第 7 节介绍相关工作，第 8 节总结。

A.3 CFS 和 ULE 算法概述

A.3.1 Linux 的 CFS 算法

A.3.1.1 单核调度

Linux 的 CFS 实现了一种加权公平排队算法：它在线程之间平均分配 CPU 周期，这些线程由它们的优先级加权（由它们的 niceness 表示，高 niceness 意味着低优先级，反之亦然）[6]。为此，CFS 通过被称为 vruntime 的多因素值对线程进行排序。vruntime 等于线程已使用的 CPU 时间量除以线程的优先级。具有相同优先级和相同 vruntime 的线程执行相同的时间量，这意味着核心资源已在它们之间公平共享。为了保证所有线程的 vruntime 公平增长，当当前运行的线程被抢占时，CFS 会选择运行 vruntime 最低的线程。

自 Linux 2.6.38 以来，CFS 中的公平性概念已经从线程间的公平性发展为应用程序间的公平性。在 Linux 2.6.38 之前，每个线程都被视为一个独立的实体，并与系统中的其他线程共享相同的资源。这意味着使用多线程的应用程序比单线程应用程序获得更大的资源份额。在较新的内核版本中，同一应用程序的线程被分组到一个称为 cgroup 的结构中。一个 cgroup 有一个 vruntime，它对应于其所有线程的 vruntime 的总和。CFS 然后将其算法应用于 cgroups，确保线程组之间的公平性。当选择一个 cgroup 进行调度时，cgroup 内具有最低 vruntime 的线程将被执行，以确保 cgroup 内的公平性。cgroup 也可以嵌套。例如，systemd 自动配置 cgroups 以确保不同用户之间的公平性，并保证给定用户的应用程序之间的公平性。

CFS 通过在给定时间段内调度所有线程来避免线程饥饿。对于执行少于 8 个线程的 CPU 核，默认时间段为 48 毫秒。当一个核执行 8 个以上线程时，时间周期随线程数增长，其毫秒数等于 6 乘以线程数；选择 6 毫秒是为了避免过于频繁地抢占线程。具有较高优先级的线程获得较高的时间段份额。由于 CFS 调度具有最低 vruntime 的线程，因此 CFS 需要防止线程的 vruntime 远低于等待调度的其他线程的 vruntime。如果发生这种情况，具有低 vruntime 的线程可能会运行很长

时间，导致其他线程出现饥饿。实际上，CFS 确保任意两个线程之间的 vruntime 差异小于抢占周期（6ms）。它在两个关键点这样做：（i）当一个线程被创建时，线程开始时的 vruntime 等于在运行队列中等待的线程的最大 vruntime，以及（ii）当一个线程在休眠之后被唤醒时，它的 vruntime 被更新为至少等于等待被调度的线程的最小 vruntime。使用最小的 vruntime 还可以确保首先安排休眠时间长的线程。这在桌面系统上是一种理想的策略，因为它最大限度地减少了交互式应用程序的延迟。大多数交互式应用程序大部分时间都在休眠，等待用户输入，并在用户与它们交互时立即被调度。

CFS 还使用启发式方法来提高缓存使用率。例如，当一个线程被唤醒时，它会检查它的 vruntime 和当前正在运行的线程的 vruntime 之间的差异。如果差异不大（小于 1 毫秒），则当前运行的线程不会被抢占——CFS 牺牲延迟以避免频繁的线程抢占，因为这可能会对缓存产生负面影响。

A.3.1.2 负载平衡

在多核设置中，Linux 的 CFS 平均分配机器所有 CPU 核的工作量。这不同于平衡线程数。例如，如果用户运行 1 个 CPU 密集型线程和 10 个大部分处于休眠状态的线程，CFS 可能会在单个核上调度这 10 个大部分处于休眠状态的线程。

为了平衡工作量，CFS 使用线程和核心的负载指标。线程的负载对应于线程的平均 CPU 使用率：从不休眠的线程比经常休眠的线程具有更高的负载。与 vruntime 一样，线程的负载由线程的优先级加权。核心的负载是可在该核心上运行的线程的负载总和。CFS 试图平衡各 CPU 核的负载。

CFS 在创建或唤醒线程时会考虑内核的负载。调度器首先决定哪些核适合托管线程。这个决定涉及许多启发式算法，例如发起唤醒的线程唤醒其他线程的频率。例如，如果 CFS 检测到一对多生产者-消费者模式，那么它会在机器上尽可能多地分散消费者线程，并且机器的大多数核都被认为适合托管被唤醒的线程。在一对一的通信模式中，CFS 将合适核心的列表限制为与发起唤醒的线程共享缓存的核心。然后，在所有合适的核中，CFS 选择负载最低的核来唤醒或创建线程。

负载平衡也会定期发生。每 4 毫秒，每个核心都会尝试从其他核心窃取工作。这种负载平衡考虑了机器的拓扑结构：相比从“远程”的核心（例如，在一个远程 NUMA 节点），核心更频繁地尝试从“靠近”它们的核心（例如，共享缓存）窃取工作。当一个核决定从另一个核窃取工作时，它会尝试通过从另一个内核窃取多达 32 个线程来平衡两个内核之间的负载。核心在空闲时也会立即调用周期

性负载均衡器。

在大型 NUMA 机器上，CFS 不会相互比较所有核的负载，而是以分层方式平衡负载。例如，在一台有 2 个 NUMA 节点的机器上，CFS 会均衡 NUMA 节点内部的核心负载，然后比较 NUMA 节点的负载（定义为节点中核心的平均负载）来决定是否平衡节点之间的负载。如果节点之间的负载差异很小（实际小于 25%），则不执行负载平衡。两个核心（或核心组）之间的距离越大，CFS 进行平衡负载所需的不平衡度就越高。

A.3.2 FreeBSD 的 ULE 算法

A.3.2.1 单核调度

ULE 使用两个运行队列来调度线程：一个运行队列包含交互式线程，另一个运行队列包含批处理线程。当 CPU 核空闲时，使用称为空闲的第三个运行队列。这个运行队列只包含空闲任务。

拥有两个运行队列的目的是优先考虑交互式线程。批处理通常在没有用户交互的情况下执行，因此调度延迟不太重要。ULE 使用 0 到 100 之间的值来度量交互性惩罚，并通过该值记录线程的交互性。该值被定义为线程运行时间 r 和线程自愿休眠时间 s （不包括等待 CPU 的时间）的函数，计算如下：

$$scaling\ factor = m = 50 \tag{A.1}$$

$$penalty(r,s) = \begin{cases} \frac{m}{\frac{s}{r}} & s > r \\ \frac{m}{\frac{s}{r}} + m & otherwise \end{cases} \tag{A.2}$$

位于范围下半部分 (≤ 50) 的惩罚值意味着线程自愿休眠的时间多于运行时间，而高于该范围的惩罚值则相反。

ULE（默认情况下）只记录线程生命周期的最后 5 秒的休眠和运行时间。一方面，拥有大量历史记录会延长检测批处理线程所需的时间。另一方面，历史太少会造成检测中的噪音[18]。

为了对线程进行分类，ULE 首先计算一个定义为交互性惩罚值加上友好度的分数。如果一个线程的分数低于某个阈值，则该线程被认为是交互式的，这个阈值在 FreeBSD11.1 中默认为 30。当该线程的 niceness 值为 0 时，这大致对应于该线程 60% 以上的时间都在休眠。否则，它被归类为批处理。负的 niceness 值（高优先级）使线程更容易被认为是交互的。

创建线程时，它会继承其父线程的运行时和休眠时间（以及交互性）。当一

个线程死亡时，它在最后 5 秒内的运行时间被返回给它的父线程。这会惩罚在交互时生成批处理子线程的父线程。

在交互式和批处理运行队列中，线程按优先级进一步排序。交互式线程的优先级是其分数的线性插值（即，惩罚值为 0 的线程具有最高交互优先级，而惩罚为 30 的线程具有最低交互优先级）。在交互式运行队列中，每个优先级都有一个 FIFO 队列。为了将线程添加到运行队列，调度程序将线程插入到由线程优先级索引的 FIFO 队列的末尾。从该运行队列中选择一个线程运行只需通过获取非空的优先级最高的 FIFO 队列中的第一个线程即可完成。

批处理线程的优先级取决于它们的运行时间：一个线程运行得越多，它的优先级就越低。线程的 niceness 也对优先级有线性影响。在批处理运行队列中，每个优先级也有一个 FIFO 队列。插入和移除工作与交互式队列基本一样，但略有不同。为了避免批处理线程之间的饥饿，ULE 通过最小化线程之间的运行时间差异来尝试在批处理线程之间保持公平，类似于 CFS 通过计算 vruntime 所做的那样。

选择下一个要运行的线程时，ULE 首先在交互式运行队列中进行搜索。如果一个交互式线程准备好被调度，它返回那个线程。如果交互式运行队列为空，ULE 将在批处理运行队列中搜索。如果两个运行队列都为空，则意味着核心处于空闲状态，没有线程会被调度。

ULE 搜索队列的顺序有效地赋予交互式线程相对于批处理线程的绝对优先级。如果机器执行太多交互式线程，批处理线程可能会处于饥饿。然而，人们认为，由于根据定义交互式线程休眠时间多于执行时间，因此不会出现饥饿现象。

线程运行一段定义为时间片的有限时间。与 CFS 相反，线程的时间片过期的速率是相同的，无论其优先级如何。但是，时间片的长度会随着当前在核上运行的线程数而变化。一个核执行 1 个线程时，时间片为 10 个 CPU 周期（78 毫秒）。当多个线程正在运行时，时间片的长度为此值除以线程数，同时最低为 1 个 CPU 周期（1/127 秒）。在 ULE 中，完全抢占被禁用，这意味着只有内核线程可以抢占其他线程。

A.3.3 负载平衡

ULE 仅均衡每个内核的线程数。在 ULE 中，CPU 核的负载被简单地定义为当前可在该内核上运行的线程数。与 CFS 不同，ULE 不会将线程分组到 cgroups 中，而是将每个线程视为一个独立的实体。

在为新创建或唤醒的线程选择核时，ULE 使用亲和启发式算法。如果线程被认为在它运行的最后一个核上缓存亲和，那么它会被放置在这个核上。否则，ULE 会在拓扑中找到被认为是缓存亲和的最高级别，如果没有缓存亲和的核，则找到整个机器。ULE 首先尝试从其中找到一个最低优先级高于该线程的核。如果失败，ULE 再试一次，但这次是在机器的所有核心上。如果这也失败了，ULE 只会选择机器上运行线程数最少的核。

ULE 也会每 500-1500 毫秒（周期的持续时间是随机选择的）定期平衡线程。与 CFS 不同，周期性负载平衡仅由核心 0 执行。核心 0 只是简单地尝试平衡核心之间的线程数量，如下所示：来自负载最重的核心 (donor) 的线程将被迁移到较少负载的核心（receiver）。一个核只能是线程的提供方或接收者一次，并且负载平衡器会重复执行此操作，直到找不到提供方或接收者为止，这意味着一个核可以在每次负载平衡器调用时放弃或接收最多一个线程。

当核的交互式和批处理运行队列为空时，ULE 也会进行线程平衡。ULE 尝试从与空闲核共享高速缓存的负载最大的核中窃取。如果 ULE 窃取线程失败，它会在更高级别的拓扑中尝试，依此类推，直到它最终成功窃取线程。与周期性负载平衡器一样，空闲窃取机制最多窃取一个线程。

ULE 中的周期性负载平衡发生的频率低于 CFS，但在 ULE 中的线程放置期间选择 CPU 核时涉及更多计算。理由是拥有更好的初始线程放置避免了频繁运行全局负载平衡器的需要。①

A.4 将 ULE 算法移植到 Linux 内核中

在本节中，我们描述了将 ULE 移植到 Linux 时遇到的问题，以及我们的移植版本与原始 FreeBSD 代码之间的主要区别。

Linux 内核提供了一个 API 来向内核添加新的调度器。调度器必须实现表 A.1 中显示的一组函数。这些函数负责在运行队列中添加和删除线程、选择要调度的线程以及将线程放置在核心上。

FreeBSD 没有为调度器提供这样的 API。相反，它声明必须定义的函数的原型，这意味着一次只能使用一个调度器，这与 Linux 不同，在 Linux 中，多个调度类可以共存。幸运的是，ULE 调度程序中的函数可以很容易地映射到 Linux 中的对应部分（参见表 A.1）。在少数接口不匹配的情况下，也可以找到解决方法。

① 在最新版本的 FreeBSD 中，由于一个错误，周期性负载平衡器永远不会执行[2]. 在我们的 ULE 代码中，我们修复了错误，使得负载平衡器会定期执行。

例如，Linux 使用单个函数将新创建的线程和已唤醒的线程入队，而 FreeBSD 使用两个函数。Linux 在其函数中使用标志参数来区分这两种情况。使用这个标志来选择相应的 FreeBSD 功能就足够了。

表 A.1　Linux 调度器 API 和 FreeBSD 中的等价函数

Linux	FreeBSD 中等价的函数	用途
`enqueue_task`	`sched_add` 用于新线程, `sched_wakeup` 用于被唤醒的线程	在调度队列中添加一个线程
`dequeue_task`	`sched_rem`	从调度队列中删除一个线程
`yield_task`	`sched_relinquish`	放弃 CPU 并交给调度器接管
`pick_next_task`	`sched_choose`	选择被调度的下一个任务
`put_prev_task`	`sched_switch`	更新刚执行完毕的线程的数据
`select_task_rq`	`sched_pickcpu`	为新线程（或被唤醒的线程）选择 CPU

除了接口之外，CFS 和 ULE 在一些底层假设上也有所不同。最显着的区别与运行队列数据结构中当前线程的存在与否有关。Linux 调度类机制依赖于当前线程在核上运行时停留在运行队列数据结构中的假设。在 ULE 中，在核心上运行的线程从运行队列数据结构中删除，并在其时间片耗尽时添加回来，因此 FIFO 属性成立。当尝试在 Linux 中实现此行为时，我们遇到了几个问题，例如线程试图更改其调度类从而导致内核崩溃。我们决定改用 Linux 的做事方式，将当前正在运行的线程留在运行队列中。因此，我们不得不稍微更改 ULE 负载平衡以避免迁移当前正在运行的线程。

此外，在 ULE 中，当将线程从一个 CPU 迁移到另一个 CPU 时，调度器会获取两个运行队列上的锁。在 Linux 中，这种锁定机制会导致死锁。因此，我们修改了 ULE 负载平衡代码以使用与 CFS 相同的机制。

最后，在 FreeBSD 中，ULE 负责调度所有线程，而 Linux 对不同的优先级范围使用不同的调度策略（例如，高优先级线程的实时调度程序）。在这项工作中，我们主要感兴趣的是比较优先级在 CFS 范围 (100-139) 内的工作负载。因此，我们缩小了 ULE 惩罚值以适应 CFS 范围。

A.5 实验环境

A.5.1 机器

我们在具有 32GB RAM 的 32 核机器 (AMD Opteron 6172) 上评估 ULE。所有实验均在最新的 LTS Linux 内核 (4.9) 上进行。我们还在较小的台式机（8 核 Intel i7-3770）上进行了实验，并得出了类似的结论。由于篇幅限制，我们从论文中省略了这些结果。

A.5.2 实验任务

为了评估 CFS 和 ULE 的性能，我们使用了合成的基准测试程序和实际应用程序。Fibo 是一个计算斐波那契数列的合成应用程序。Hackbench[1] 是 Linux 社区设计的用于对调度程序进行压力测试的基准测试。它创建大量运行时间很短的线程并使用管道 (Pipe) 交换数据。我们还根据完成时间从 Phoronix 测试集[3] 中选择了 16 个应用程序。我们排除了在单核上需要超过 10 分钟才能完成的 Phoronix 应用程序，或者那些太短以至于无法进行可靠时间测量的应用程序。被选择的 16 个 Phoronix 应用程序是：编译基准测试（build-apache、build-php）、压缩（7zip、gzip）、图像处理（c-ray、dcraw）、科学计算（himeno、hmmer、scimark）、密码学应用程序 (john-the-ripper) 和网络应用程序 (apache)。我们使用 NAS 基准集[10] 对 HPC 应用程序进行基准测试，使用 Parsec 基准套件[11] 对并行应用程序进行基准测试，使用 Sysbench[5] 和 MySQL 和 RocksDB[4] 作为数据库基准。我们使用一个 sysbench 和 RocksDB 的读写工作来测试具有不同行为的线程的调度。

A.6 单核调度的评估

在本节中，我们在单核上运行应用程序以避免负载平衡器的可能导致的副作用。CFS 和 ULE 在单核调度方面的主要区别在于批处理线程的处理：CFS 试图对所有线程公平，而 ULE 优先考虑交互式线程。我们首先通过在同一个核上共同调度批处理和交互式应用程序来分析此设计决策的影响，我们表明在 ULE 下，批处理应用程序可能会出现无限时间的饥饿。然后我们证明即使系统只运行一个应用程序，ULE 导致的饥饿也可能发生。我们通过比较 37 个应用程序的性能来结束本节，并展示有关线程抢占的不同选择如何影响性能。

A.6.1 同时调度多个应用时的公平性和饥饿现象

在本节中，我们分析了运行多应用程序工作负载的 CFS 和 ULE 的行为，该工作负载包括一个从不休眠的计算密集型线程（fibo，计算数字）和一个其线程大部分休眠的应用程序（sysbench，一个数据库，使用 80 线程）。每个 CPU 核拥有超过 80 个线程在数据中心中并不少见[15]。这些线程永远不会同时处于活动状态；他们主要等待传入的请求，或者等待存储在磁盘上的数据。

Fibo 单独运行 7 秒，然后启动 sysbench。然后两个应用程序都运行完成。图A.1(a) 显示了 fibo 和 sysbench 在 CFS 上的累计运行时间，图A.1(b) 显示了 ULE 上的相同指标。

在 CFS 上，sysbench 在 235 秒内完成，然后 fibo 单独运行。fibo 线程和 sysbench 线程共享机器。当 sysbench 执行时，fibo 运行时间的增长大约是单独运行时的一半，这意味着 fibo 获得 50% 的核心。这是意料之中的：CFS 尝试在两个应用程序之间公平地共享核心。实际上，由于舍入误差，fibo 获得的 CPU 使用率略低于一半。

在 ULE 上，sysbench 能够在 143 秒内完成相同的工作负载，然后 fibo 自行运行。sysbench 运行时 fibo 饥饿。sysbench 线程主要处于休眠状态，因此它们被归类为交互线程，并且绝对优先于 fibo。由于 sysbench 使用 80 个线程，这些线程能够使核心饱和，并阻止 fibo 运行。图A.2 显示了 fibo 和 sysbench 的交互性惩罚随时间的演变。两个应用程序开始时都是交互式的（惩罚为 0）。fibo 的惩罚迅速上升到最大值，然后 fibo 不再被认为是交互的。相反，Sysbench 线程在其整个执行过程中保持交互（惩罚低于 30 的限制）。因此，sysbench 线程绝对优先于 fibo 线程。只要 sysbench 正在运行，这种情况就会持续存在（即饥饿时间不受 ULE 的限制）。

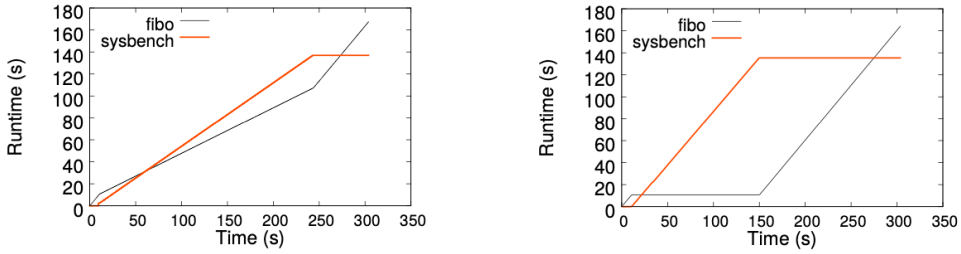
表A.2显示了 fibo 和 sysbench 在 CFS 和 ULE 上的总执行时间，以及请求 sysbench 的延迟。Sysbench 在 CFS 上运行速度慢 50%，因为它与 fibo 共享 CPU，而不是像在 ULE 上那样独立运行（CFS 为 290 个事务/秒，ULE 为 532 个）。Fibo 在 ULE 上执行 sysbench 期间“停止”，但随后开始自行执行，因此可以比在 CFS 上与 sysbench 同时运行时更有效地使用缓存。因此，fibo 在 ULE 上的运行速度略快于在 CFS 上。

我们发现 ULE 调度程序使用的策略可以很好地处理对延迟敏感的应用程序。这些应用程序通常被正确地归类为交互式应用程序，并且优先于后台线程。为了在 Linux 中获得相同的结果，延迟敏感的应用程序必须由实时调度程序执行，它

表 A.2　fibo 和 sysbench 在使用 CFS 和 ULE 时的执行时间，sysbench 在使用 CFS 和 ULE 时的请求的平均延迟。

	CFS	ULE
Fibo - 执行时间	160 s	158s
Sysbench - 事件/秒	290	532
Sysbench - 平均延迟	441 ms	125 ms

比 CFS 具有绝对优先级。



(a)　　(b)

图 A.1　fibo 和 sysbench 在 (a) CFS 和 (b) ULE 上的累积运行时间。(a) 在 CFS 上，当 sysbench 执行时，fibo 继续累积运行时间，尽管速度更慢，这意味着 fibo 不会出现饥饿。(b) 在 ULE 上，当 sysbench 执行时，fibo 停止累积运行时间，这意味着它处于饥饿状态。

A.6.2　单个应用下的公平性和饥饿现象

ULE 在上面的多应用程序场景中表现出的饥饿也发生在单应用程序工作负载中。我们现在使用 sysbench 来举例说明这种行为。

在 ULE 中，新创建的线程在 fork 时继承其父线程的交互性惩罚。在 sysbench 中，创建主线程时会继承 bash 进程的交互性损失。由于 bash 大部分时间都在休眠，因此 sysbench 被创建为一个交互式进程。sysbench 主线程初始化数据并创建线程。在这样做的同时，它从不睡觉，所以它的交互性惩罚增加了。第一个线程创建时交互性惩罚低于交互阈值，而其余线程创建时交互性惩罚高于它。因此，第一个线程获得绝对优先于其余线程。由于这些线程大部分时间都在等待新请求，因此它们的交互性损失保持较低（减少到 0），并且它们的优先级仍然高于在初始化过程后期创建的线程。如果交互式线程使 CPU 始终处于忙碌状态，则后期创建的线程可能永远处于饥饿状态。

图A.3 显示了 sysbench 线程的累积运行时间，图A.4 显示了它们的交互性损

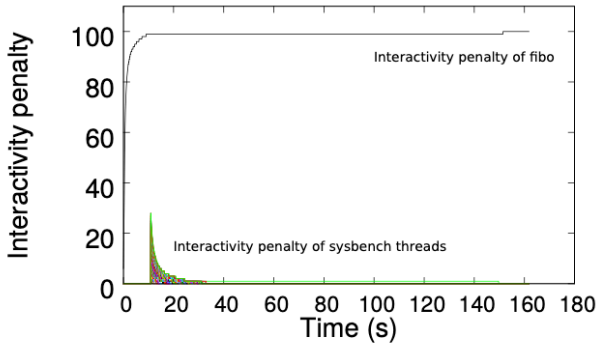


图 A.2　线程的交互性损失随时间的变化。Fibo 的惩罚迅速上升到最大值，而 sysbench 线程的惩罚下降到 0。

失。Sysbench 被配置为使用 128 个线程。较早创建的线程执行，它们的交互性惩罚降为 0。较晚创建的线程永远不会执行。

与直觉相反，在此基准测试中，ULE 实际上比 CFS 表现更好，因为它避免了过度订阅：机器运行尽可能多的线程。因此，ULE 的平均延迟低于 CFS。总的来说，我们发现这种在纸面上看似有问题的饥饿机制在所有线程竞争执行相同工作的应用程序中表现非常好。

与 sysbench 相比，我们测试的科学计算应用程序不受饥饿影响，因为它们的线程从不休眠。经过短暂的初始化后，所有线程都被视为后台线程，并以公平的方式进行调度。

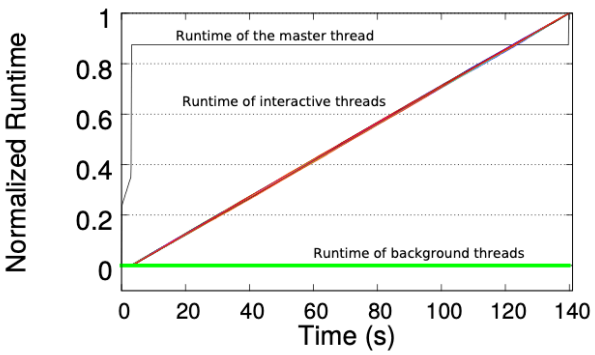


图 A.3　ULE 上 sysbench 线程的累计运行时间。主线程首先产生 128 个线程。80 个线程被归类为交互并执行，48 个线程被归类为后台和并处于饥饿状态。

A.6.3　性能分析

我们现在分析单核调度对 37 个应用程序性能的影响。我们将“性能”定义如下：对于数据库工作负载和 NAS 应用程序，我们比较每秒的操作数，对于其

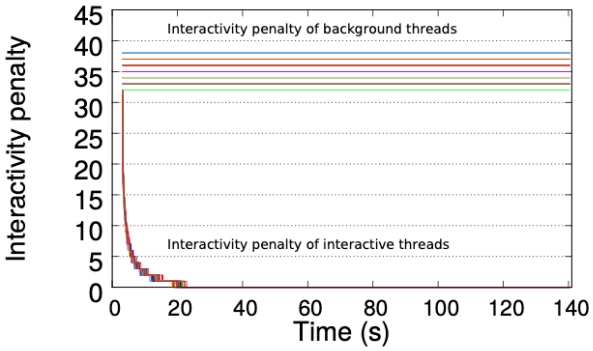


图 A.4　图A.3 中显示的线程的交互性损失。线程在创建时继承其父线程的交互性损失。有些线程是在低惩罚下创建的，并且它们的惩罚随着它们执行而降低（图的底部），而其他线程是在高惩罚下创建的并且从不执行（图的顶部）。

他应用程序，我们比较“1/执行时间”。“性能”越高，调度程序的性能就越好。图A.5 显示了 CFS 和 ULE 在单核上的性能差异，其中百分比高于 0 意味着使用 ULE 的应用程序执行速度比 CFS 快。

总的来说，调度器对大多数工作负载影响不大。事实上，大多数应用程序都使用执行相同工作的线程，因此 CFS 和 ULE 最终都以时间片轮转方式调度所有线程。平均性能差异为 ULE 高出 1.5%。尽管如此，scimark 在 ULE 上比 CFS 慢 36%，而 apache 在 ULE 上比 CFS 快 40%。

Scimark 是一个单线程 Java 应用程序。它启动一个计算线程，Java 运行时在后台执行其他 Java 系统线程（用于垃圾收集器、I/O 等）。当使用 ULE 执行应用程序时，计算线程可能会延迟，因为 Java 系统线程被认为是交互式的，并且优先于计算线程。

apache 工作负载由两个应用程序组成：运行 100 个线程的主服务器 (httpd) 和单线程负载注入器 ab。ULE 和 CFS 之间的性能差异可以通过有关线程抢占的不同选择来解释。

在 ULE 中，完全抢占被禁用，而 CFS 在刚被唤醒的线程的 vruntime 比当前正在执行的线程的 vruntime 小得多时抢占正在运行的线程（实际相差 1ms）。在 CFS 中，ab 在基准测试期间被抢占了 200 万次，而它从未被 ULE 抢占过。这种行为解释如下：ab 首先向 httpd 服务器发送 100 个请求，然后等待服务器应答。当 ab 被唤醒时，它会检查哪些请求已被处理并向服务器发送新的请求。由于 ab 是单线程的，所以发送到服务器的所有请求都是按顺序发送的。在 ULE 中，ab 能够发送与收到响应一样多的新请求。在 CFS 中，ab 发送的每个请求都会唤醒一个 httpd 线程，该 httpd 线程会抢占 ab。

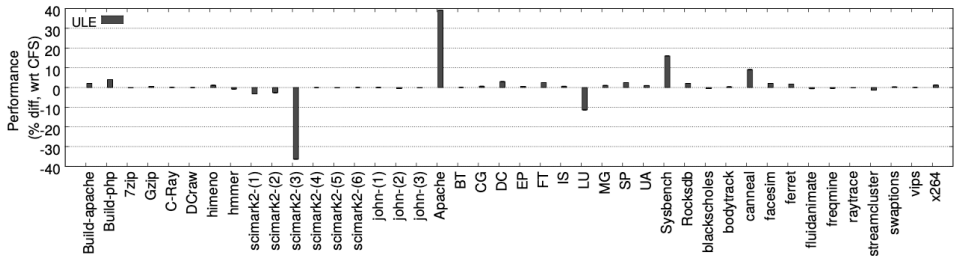


图 A.5　在单核上，ULE 相对于 CFS 的性能。大于 0 的数字表示应用程序在 ULE 上比在 CFS 上运行得更快。

A.7　负载平衡器的评估

在本节中，我们分析了负载平衡和线程放置策略对性能的影响。在 CFS 和 ULE 中，负载平衡会周期性发生，线程放置会在线程创建或唤醒时发生。我们首先分析周期性负载平衡器在机器的所有核上平衡一个静态工作负载所花费的时间。然后我们分析 CFS 和 ULE 在放置线程时做出的选择。接下来，我们比较在 CFS 和 ULE 上运行的 37 个应用程序的性能。最后，我们分析了多应用程序工作负载的性能。

A.7.1　周期性负载平衡

CFS 依赖于一个相当复杂的负载指标。它每 4 毫秒运行一次分层负载平衡策略。ULE 仅尝试平衡内核上的线程数。ULE 中负载平衡发生的频率较低（周期在 0.5 秒到 1.5 秒之间变化）并且忽略了硬件拓扑。我们现在评估这些策略如何影响平衡机器负载所需的时间。

为此，我们将 512 个旋转线程固定在核心 0 上，我们启动一个任务集命令来取消固定线程，然后让负载均衡器平衡核心之间的负载。所有线程都执行相同的工作（无限空循环），因此我们期望负载均衡器在 32 个核中的每个核上放置 16 个线程。图A.6 显示了每个核上的线程数随时间的演变。在图中，32 行中的每一行都代表给定核上的线程数。取消固定线程的任务集命令在 14.5 秒时启动。

在 ULE 上，一旦线程被取消固定，空闲核心就会从核心 0 中窃取线程（每个核心最多一个），因此在取消固定之后，核心 0 有 $512 - 31 = 481$ 个线程，而其他每个核心都有 1 个线程。随着时间的推移，周期性负载平衡器被触发并尝试平衡线程数。但是，由于负载均衡器一次仅从核心 0 迁移一个线程，因此需要调用 450 次以上的负载均衡器或大约 240 秒才能达到平衡状态。

CFS 可以更快地平衡负载。取消固定后 0.2 秒，CFS 从核心 0 迁移了超过 380

个线程。令人惊讶的是，CFS 从未实现完美的负载平衡。CFS 仅在两个节点之间的不平衡“足够大”（实际负载差异为 25%）时才平衡 NUMA 节点之间的负载。因此一个节点中的核心可以有 18 个线程，而另一个节点中的核心只有 15 个。

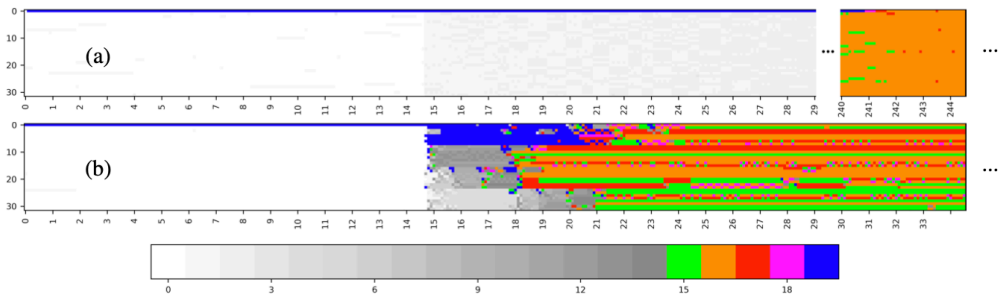


图 A.6　使用 (a) ULE 和 (b) CFS 时，每个核上的线程数随时间的变化。每条线代表一个核（总共 32 个），x 轴是时间轴（以秒为单位），颜色代表核上的线程数。低于 15 的线程数以灰色阴影表示。在执行的前 14.5 秒内，线程固定在核 0 上。

虽然 CFS 使用的负载平衡策略非常适合解决系统中的大量不平衡负载，但当完美的负载平衡对性能很重要时，它不太适合。

A.7.2　线程放置策略

我们使用 c-ray 研究线程的放置，c-ray 是 phoronix 基准测试集中的图像处理应用程序。C-ray 首先创建 512 个线程。线程在创建时不会选择固定的 CPU 核，因此调度程序会为每个线程选择一个核。然后所有线程在执行计算之前都在屏障上等待。由于所有线程都以相同的方式运行，我们希望 ULE 在该配置中比 CFS 表现更好：ULE 总是在线程数量最少的核上复制线程，因此从一开始就应该完美地平衡负载。

图A.7 显示了每个核上可运行线程数随时间的变化。ULE 中的负载始终是平衡的，但令人惊讶的是，ULE 需要超过 11 秒才能让所有线程运行，而 CFS 只需 2 秒。这种延迟可以用饥饿来解释。C-ray 使用级联屏障，其中线程 0 唤醒线程 1，线程 1 唤醒线程 2，等等。线程最初创建时具有不同的交互性惩罚，一些线程最初是交互的，而其他线程最初是批处理的（相同与 sys-bench 中的原因相同，请参阅第 A.6.2 节）。当批处理线程被唤醒时，它总处于饥饿状态，直到所有交互式线程都完成前，或者直到它们的惩罚增加到足以将它们降级为批处理运行队列。实际上，在 c-ray 中，线程在屏障之后永远不会休眠，因此最终所有线程都变成了批处理，但是，在它们降级为批处理线程之前，最初被归类为批处理的线程无法

唤醒其他线程。因此，所有线程在屏障之后被唤醒需要 11 秒。

CFS 反而是公平的，所有的线程都很快被唤醒。然后，CFS 遇到了我们在第 A.7.1 节中解释的不完善的负载平衡问题。尽管存在这些负载平衡差异，c-ray 在 CFS 和 ULE 上仍同时完成，因为 c-ray 创建的线程多于 CPU 核数，且两个调度器始终保持所有核都是忙碌的。用 CFS 时确实会发生更多抢占，但不影响性能。

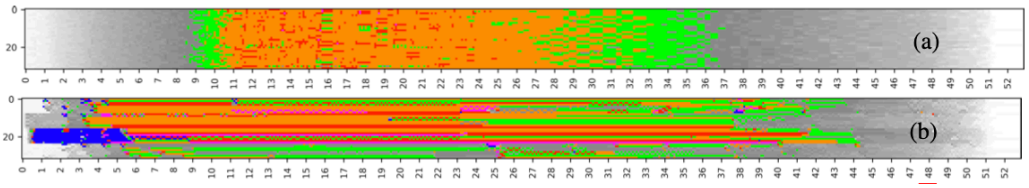


图 A.7　使用 (a) ULE 和 (b) CFS 运行 c-ray 时，每个核上的线程数随时间的变化。与图A.6 相反，线程不会开始固定在核 0 上。

A.7.3　性能分析

图A.8显示了多核环境中 CFS 和 ULE 之间的性能差异。CFS 和 ULE 之间的平均性能差异很小：ULE 性能高出 2.75% 。

来自 NAS 基准测试套件的 MG 从 ULE 的负载平衡策略中获益最多：它在 ULE 上比在 CFS 上快 73%。MG 产生与机器中的 CPU 核一样多的线程，并且所有线程执行相同的计算。当一个线程完成计算后，它会在自旋屏障上等待 100 毫秒，然后如果某些线程仍在计算则休眠。ULE 正确地为每个核放置一个线程，然后再也不会迁移它们。线程在屏障中花费很少的时间相互等待，并且从不休眠。相反，CFS 对核心负载的微小变化做出反应（例如，由于内核线程唤醒），有时会错误地将两个 MG 线程放在同一个核心上。由于 MG 使用屏障，调度在同一个核心上的两个线程最终会延迟整个应用程序。这种情况导致的延迟超过 50%，因为单独调度在一个核上的线程进入睡眠状态，然后必须被唤醒，从而增加了障碍的延迟。这种次优的线程放置也解释了 CFS 和 ULE 在 FT 和 UA 上的性能差异。ULE 平衡使用的线程数量的简单方法在类似 HPC 的应用程序上效果更好，因为它最终为每个核放置一个线程，然后再也不会迁移它们。

由于 ULE 负载平衡器的开销，Sysbench 在 ULE 上速度较慢。当一个线程被唤醒时，ULE 扫描机器的核心来为线程找到合适的核心，最坏的情况下，可能会扫描所有核 3 次。这种最坏的情况发生在 sysbench 中的大多数唤醒操作时，导致所有 CPU 周期的 13% 用于扫描机器的核。为了验证这个假设，我们用一个简单的函数替换了 ULE 唤醒函数，该函数返回线程先前运行的 CPU 核，然后观察到

ULE 和 CFS 之间没有区别。

在我们测试的所有基准测试中，我们在 ULE 中观察到的调度程序花费的最高时间是 13%，我们在 CFS 中观察到的调度程序花费的最高时间是 2.6%。请注意，ULE 遇到 sysbench 的极端情况，但在其他基准测试中的开销很低，即使它们产生大量线程也是如此：例如在 hackbench（32 000 个线程）中，ULE 的开销是 1%（相较于 CFS 的 0.3%）。

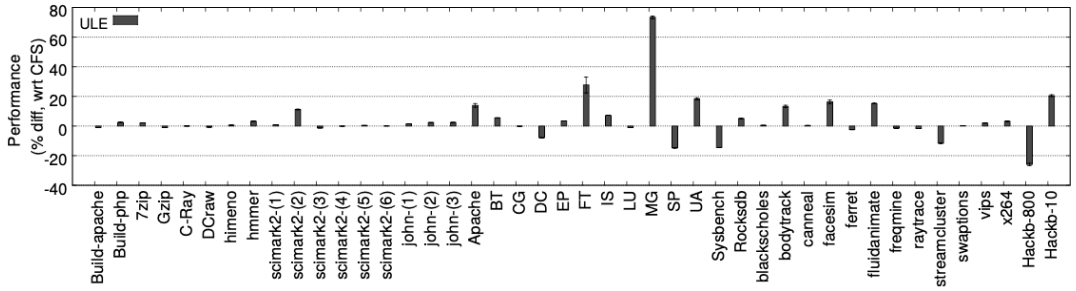


图 A.8　在多核机器上，ULE 相对于 CFS 的性能

A.7.4　多应用工作负载

最后，我们使用一组不同的应用程序评估交互式和后台工作负载的组合：c-ray + EP（来自 NAS）是一个工作负载，其中两个应用程序都被 ULE 视为后台，fibo + sysbench 和 blackscholes + ferret 是只有一个应用程序是交互式的工作负载，而 apache + sys-bench 是一个完全交互式的工作负载。图A.9 显示了 CFS 和 ULE 相对于单独在机器上运行各应用程序的性能表现（越高越好）。总体而言，大多数应用程序在与另一个应用程序共同调度时运行速度较慢。

当两个应用程序都是非交互式的（c-ray + EP）时，CFS 和 ULE 的性能相似。这是意料之中的，因为它们以类似的方式安排后台线程。EP 单独执行时在 ULE 上运行速度稍快，当它与 c-ray 共同调度时，这种性能差异仍然存在。当两个应用程序都处于交互状态时，CFS 和 ULE 的性能也相似。

对于 blackscholes + ferret，ULE 优先考虑交互式线程，ferret 不受与 blackscholes 共同调度的影响。然而，Blackscholes 的运行速度要慢 80% 以上。在这种情况下，blacksc-holes 并没有一直处于饥饿，因为 ferret 并没有 100% 地使用所有核。相反，CFS 公平地共享 CPU，两个应用程序都受到同样的影响（共调度对这些应用程序的影响小于 50%，因为 ferret 和 blackscholes 都没有扩展到 32 核）。

令人惊讶的是，当与 fibo 共同安排时，sysbench 在 ULE 上的性能比在 CFS 上更差，即使它被正确归类为交互式并且优先于 fibo 线程。ULE 中缺乏抢占解

释了性能差异。MySQL 没有实现完美的扩展，当在 32 个内核上执行时，锁争用迫使线程在等待锁被释放时进入休眠状态。因此，fibo 不会处于饥饿。当 MySQL 的一个锁被释放时，ULE 不会抢占当前正在运行的线程（通常是 fibo）来调度一个新的 MySQL 线程进入临界区。这会增加 sysbench 执行的延迟（最多 fibo 时间片的长度，在 7.8 毫秒和 78 毫秒之间）。

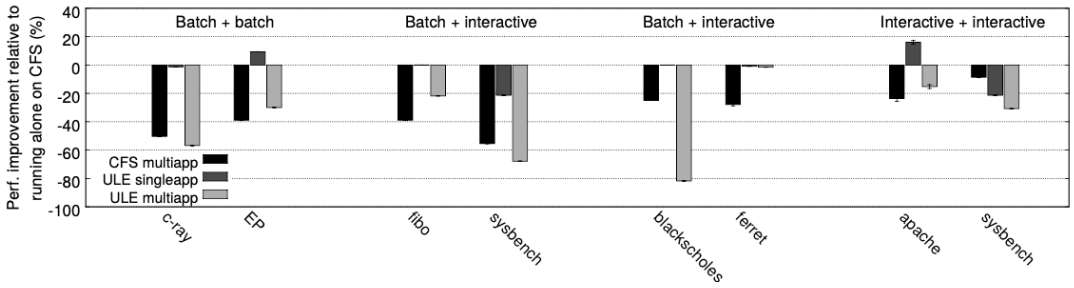


图 A.9　CFS 和 ULE 在多应用程序工作负载上各应用程序相对于单独在 CFS 上运行时的性能。

A.8　相关工作

以前的工作比较了 FreeBSD 和 Linux 的设计选择。阿巴菲等人[8-9] 比较了调度器运行队列中线程的平均等待时间。施耐德等人[19] 比较了两个操作系统的网络堆栈性能。FreeBSD 做出的设计选择也经常在 Linux 内核邮件列表[21] 上进行讨论。这项研究的方法不同：我们没有比较两个完整的操作系统，而是将 FreeBSD 中的 ULE 调度器移植到 Linux。据我们所知，这是 ULE 和 CFS 设计的首次同类比较。

Linux 调度器设计在以前的工作中也已经讨论过。托里等人[20] 将 Linux 调度器的延迟与多级反馈队列的自定义实现进行比较。黄等人将 CFS 的公平性与 Linux 2.6.23 版本之前的默认调度器 O(1) 调度器[23] 以及 RSDL 调度器（旋转楼梯截止调度器）[22] 进行比较。格罗夫斯等人[13] 将 CFS 的开销与 BFS（Brain Fuck Scheduler）进行比较，BFS 是一种简单的调度器，旨在提高 CPU 核较少的机器的响应能力。其他工作研究了调度器的开销。卡内夫等人[15] 报告仅运行 CFS 调度器的时间就占所有数据中心 CPU 周期的 5% 以上。

操作系统的性能经常通过测量内核版本之间的性能演变来评估。Linux 内核性能项目[12] 始于 2005 年，旨在测量 Linux 内核中的性能退化。莫里森等人[17] 提出 Litmus 测试以发现调度程序中的性能回归。系统社区也经常报告操作系统

中的性能问题。洛齐等人[16] 报告了 Linux 调度程序中的错误，这些错误可能会导致仍有工作等待被安排到其他核上时一些核永久闲置。哈吉等人[14] 报告了早期内核版本中类似的性能错误。在撰写本文期间，我们还向 FreeBSD 社区报告了调度程序中的错误[2]。在这项研究中，我们选择通过修复明显的错误来比较 ULE 和 CFS 的“无故障”版本，这些错误不是调度器有意保留的特点。

A.9 总结

在多核机器上调度线程很难。在本文中，我们对两种广泛使用的调度器的设计选择进行了公正的比较：FreeBSD 的 ULE 调度器和 Linux 的 CFS。我们发现，即使在简单的工作负载上，它们的行为也不同，并且没有任何调度器在所有工作负载上都比另一个更好。

参考文献

[1] Hackbench.

[2] [PATCH] fix bug in which the long term ule load balancer is executed only once.

[3] Phoronix test suite.

[4] ROCKSDB - PERFORMANCE BENCHMARKS.

[5] Sysbench.

[6] THE DESIGN OF CFS.

[7] ULE port in linux.

[8] J Abaffy and T Krajčovič. Pi-ping-benchmark tool for testing latencies and throughput in operating systems. In *Innovations in Computing Sciences and Software Engineering*, pages 557–560. Springer, 2010.

[9] Jaroslav Abaffy and Tibor KRAJČOVIČ. Latencies in linux and freebsd kernels with different schedulers–O(1), CFS, 4BSD, ULE. In *Proceedings of the 2nd International Multi-Conference on Engineering and Technological Innovation*, pages 110–115, 2009.

[10] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The NAS parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 158–165, 1991.

[11] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[12] Tim Chen, Leonid I Ananiev, and Alexander V Tikhonov. Keeping kernel performance from regressions. In *Linux Symposium*, volume 1, pages 93–102, 2007.

[13] Taylor Groves, Jeff Knockel, and Eric Schulte. BFS vs. CFS scheduler comparison. *The University of New Mexico*, 11, 2009.

[14] Ashif S Harji, Peter A Buhr, and Tim Brecht. Our troubles with linux and why you should care. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, pages 1–5, 2011.

[15] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 158–169, 2015.

[16] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The Linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[17] Mac Mollison, Björn Brandenburg, and James H Anderson. Towards unit testing real-time schedulers in LITMUSRT. In *Proceedings of the 5th Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 33–39, 2009.

[18] Jeff Roberson. ULE: a modern scheduler for FreeBSD. 2003.

[19] Fabian Schneider and Jörg Wallerich. Performance evaluation of packet capturing systems for high-speed networks. In *Proceedings of the 2005 ACM conference on Emerging network experiment and technology*, pages 284–285, 2005.

[20] Lisa A Torrey, Joyce Coleman, and Barton P Miller. A comparison of interactivity in the Linux 2.6 scheduler and an MLFQ scheduler. *Software: Practice and Experience*, 37(4):347–364, 2007.

[21] L Torvalds. Is sendfile all that sexy?

[22] Shen Wang, Yu Chen, Wei Jiang, Peng Li, Ting Dai, and Yan Cui. Fairness and interactivity of three CPU schedulers in Linux. In *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 172–177. IEEE, 2009.

[23] CS Wong, IKT Tan, RD Kumari, JW Lam, and W Fun. Fairness and interactive performance of O (1) and CFS Linux kernel schedulers. In *2008 International Symposium on Information Technology*, volume 4, pages 1–8. IEEE, 2008.

书面翻译对应的原文索引

[1] Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. The battle of the schedulers: FreeBSD {ULE} vs. Linux {CFS}. In *2018 {USENIX} Annual Technical Conference ({USENIX} {ATC} 18)*, pages 85–96, 2018.

致　　谢

衷心感谢陈渝副教授在毕业设计期间对我的悉心指导。

感谢弋力老师，刘玉身老师，陈渝老师在留学申请和科研过程中给予我的帮助、鼓励与支持。感谢为我提供过指导和帮助的其他计算机系的老师。

感谢在大学四年中为我提供过无私帮助的各位辅导员们，感谢他们的认真工作与无私奉献。感谢与我共度了四年大学时光的同学们，因为有你们的陪伴，我的大学生活才能如此丰富多彩。

声　　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

综合论文训练记录表

学生姓名		学号		班级	
论文题目					
主要内容以及进度安排	指导教师签字： 考核组组长签字： 年　　月　　日				
中期考核意见	考核组组长签字： 年　　月　　日				

指导教师评语	指导教师签字：_____ 年　　月　　日
评阅教师评语	评阅教师签字：_____ 年　　月　　日
答辩小组评语	答辩小组组长签字：_____ 年　　月　　日

总成绩：_____

教学负责人签字：_____

年　　月　　日