



中山大學

SUN YAT-SEN UNIVERSITY

Yat-Scheduler 设计开发文档

Proj #291

给 Linux 内核调度机制新增一个调度器

——考虑互斥同步的硬实时调度实现

队伍编号: T202410558992534

队伍名称: 上士闻道勤而习之

所属赛题: Proj291

项目成员: 黄政菘、薛沐恩、刘洋

校内导师: 赵帅、黎卫兵

项目导师: 刘冬华 (国科环宇)

所属高校: 中山大学

2024 年 05 月

摘要

在操作系统功能挑战赛赛题发布之后，我们队伍在指导老师的支持和帮助下，完成了从组建到磨合的过程，选定题目并完成了报名。在报名后到中期考核的这段时间里，队伍与项目导师取得了联系，进行了积极良好的沟通。与此同时，队伍积极学习比赛相关的专业知识，进行了赛题和基础开源项目的调研，对于项目整体有了相对比较全面和完整的认识，对于比赛开发的全流程也进行了比较科学完整的规划。

本文是关于队伍在初赛阶段取得的一些进展和对未来预期的一篇阶段性的技术报告，简要概括了队伍基本情况和在前期取得的项目进展、所遇的各种问题和困难、项目整体的预期目标和实现预期目标的规划与时间线。

文档目录

1. 项目介绍	5
1.1. 项目背景和意义	5
1.2. 项目目标和分析	6
1.3. 项目要求实现内容	7
1.4. 项目预期实现成果	8
1.5. 项目目前完成情况	8
1.6. 初赛项目开发历程	8
1.7. 队伍简介	9
1.7.1. 队伍成员简介	9
1.7.2. 项目导师简介	9
1.7.3. 指导老师简介	9
1.8. 初赛项目分工	10
2. 实时系统知识	11
2.1. 实时系统一些基本概念	11
2.2. 实时系统中任务模型	11
2.3. 实时系统下的调度算法	14
2.3.1. 按照优先级对调度算法进行分类	14
2.3.2. 按照抢占发生的原因对调度算法进行分类	14
2.3.3. 多核调度算法	15
2.4. Linux 中实时调度算法的实现	16
2.5. 现有研究调研概况	18
2.5.1. RTLinux, RTAI	18
2.5.2. OCERA	18
2.5.3. FRESCOR	18
2.5.4. Litmus-RT	19
2.5.5. Preempt_rt	19
2.6. 本项目的实时调度系统的设计	20
3. 调度器整体结构设计	21
3.1. 设计理念	21
3.2. 项目架构介绍	21
4. Yat-Scheduler 设计和实现	23
4.1. Linux 内核开发环境选择和搭建	23
4.1.1. 项目环境	23
4.1.2. 项目环境说明	23
4.1.3. 项目环境搭建概述	23
4.2. Yat-Scheduler 初始化	27
4.2.1. 调度初始化	27
4.3. Yat-Scheduler 基础模块设计和实现	32

4.3.1. 底层数据结构模块	32
4.3.2. 调度算法模块	33
4.4. 多核互斥资源共享协议设计和实现	36
4.4.1. 多核互斥资源共享协议介绍	36
4.4.2. 多核互斥资源共享协议设计	37
4.4.3. 多核互斥资源共享协议实现	37
4.5. 面向资源的优先级排序算法设计和实现	37
4.5.1. 面向资源的优先级排序算法介绍	37
4.5.2. 面向资源的优先级排序算法设计	40
4.5.3. 面向资源的优先级排序算法实现	49
4.6. 考虑阻塞的任务分配机制设计和实现	58
4.6.1. 考虑阻塞的任务分配机制介绍	58
4.6.2. 考虑阻塞的任务分配机制设计	58
4.6.3. 考虑阻塞的任务分配机制实现	62
5. 项目测试	68
5.1. Yat-Scheduler 功能测试	68
5.1.1. 测试概述	68
5.1.2. 测试方法	68
5.1.3. 测试结果	69
5.2. 多核互斥资源共享协议测试	69
5.3. 面向资源的优先级排序算法测试	69
5.4. 考虑阻塞的任务分配机制测试	69
5.5. Yat-Scheduler 性能测试	69
6. 遇到的困难和解决方法	70
6.1. 未来展望	70
7 参考资料	71

1 项目介绍

本小节将从项目背景开始，介绍赛题初衷，接着再结合赛题目标，对项目所要进行的主要工作进行分析，包括初赛阶段已经完成的，以及决赛阶段有待完成的。最后介绍项目最终实现的成果形式。

1.1 项目背景和意义

(1) 特定领域的 Linux 内核调度机制定制与优化

Linux 是开源操作系统，允许开发者自由修改和定制内核以满足特定需求。其以稳定性和性能著称，广泛应用于服务器、嵌入式系统和各种高性能计算环境。支持多种硬件平台，具有很强的可移植性。

然而随着计算需求的多样化和计算环境的复杂化，现有的 Linux 内核调度策略虽然能够满足大多数标准应用的需求，但在特定场景下（如实时系统、大数据处理、高性能计算等）仍有改进和优化的空间。深入 Linux 内核调度机制，设计并实现新的调度策略或优化现有调度器，以提升系统性能，增强系统的适应性和效率，这在当下有很大的必要性。

(2) 实时系统在多核处理器下的调度仍有改进空间

如今实时系统已广泛应用于航空航天、工业控制、通信、医疗等领域，其对任务的时间约束和响应时间有严格要求，需要在严格的时间约束内处理任务，确保及时和确定性的响应。随着技术的发展和需求的增加，实时系统还会在越来越多的应用场景中发挥着关键作用。并且随着多核处理器的普及，实时系统需要在多核环境下高效运行，充分利用多核资源来提高系统性能和响应能力。

然而在多核环境下，任务之间的资源争用、同步和通信变得更加复杂。需要有效的任务调度策略来避免优先级反转、任务阻塞和负载不均衡问题，确保系统的实时性和稳定性。并且共享资源（如 I/O 设备、内存）在多核环境下的访问控制需要有效的互斥机制。任务调度不仅要考虑任务的优先级，还需考虑任务对共享资源的访问需求和阻塞情况。

基于以上背景需求，我们希望通过设计和实现多核互斥资源共享协议，有效解决资源争用和优先级反转问题，提高系统的响应速度和稳定性。同时，设计和实现面向资源的优先级排序算法和考虑阻塞的任务分配机制，优化任务调度，减少任务间的阻塞和等待时间。

(3) Linux 系统的实时调度支持落后

标准的 Linux 内核主要设计用于通用用途，对实时性的支持较弱。需要对内核进行定制和优化，以满足实时系统的需求。并且在多核环境下，任务调度、资源共享和同步变得更加复杂，需要专门的机制来管理。

为了在 Linux 系统上实现实时性保障，开发一个高效的实时调度器至关重要。据我们的前期调研，当前 Linux 系统的实时调度支持，除了并不能真正算实时调度（不针对多核系统、仅仅根据优先级进行调度）的内置的 `SCHED_FIFO` 和 `SCHED_RR` 之外，主要有实时调度补丁 `Preempt_RT` [1] 和实时调度测试平台 `LitmusRT` [2]，`Preempt_RT` 无法很

好地处理多核环境下的任务调度、资源共享、优先级管理等问题；而 *Litmus^{RT}* 虽然功能相较全面且支持调度算法自定义，但由于其定位是一套为研究人员测试调度算法而开发的框架，并不面向实际的生产环境，还需要考虑兼容不同的由用户自己编写的调度算法，因此无法做到针对性地深入内核调度机制去优化特定的调度算法，且其早已于 2017 年停止维护，最新支持的内核版本永远停留在了 4.9.30 版本，无法支持当前一些新的内核特性。

因此基于以上三点，开发一款基于最前沿的实时调度理论技术，面向全新 Linux 内核，同时具备多核互斥资源共享协议、面向资源的优先级排序算法、考虑阻塞的任务分配机制的具有最高效性能的 Linux 内核实时调度器，为实时系统调度器开发提供一个高效、可靠的开源解决方案，在当下的意义无疑十分重大。

1.2 项目目标和分析

我们选择的题目为 *proj291-给 Linux 内核调度机制新增一个调度器*，题目链接为 <https://github.com/oscomp/proj291-Linux-kernel-scheduler>

本项目面向上述题目所提的要求，旨在完成以下三个目标：

- **目标 1: 设计并实现至少一个针对特定应用场景优化的 Linux 内核调度器**

目标解读：需要从头开始设计一个新的调度器，并将其实现为 Linux 内核的一部分。这意味着我们需要修改 Linux 内核代码，通常是在调度子系统中。并且调度器必须针对某个特定的应用场景进行优化。这可以是高性能计算、实时系统、低功耗设备等。我们需要明确选择一个具体的应用场景，并确定该场景下调度器需要优化的性能指标，如响应时间、吞吐量、延迟、能耗等。

- **目标 2: 对比分析新调度器与现有调度策略（如 CFS、Real-time Scheduler）的性能差异**

目标解读：这需要我们设计并执行一系列实验，将所设计的调度器与现有的调度策略进行对比。这包括 Linux 内核中常见的完全公平调度器（CFS）和实时调度器。还需要收集并分析各个调度器的性能数据，比较它们在特定应用场景下的表现。性能差异的指标可能包括但不限于：任务完成时间、系统吞吐量、CPU 利用率、上下文切换次数等。其中，CFS 和实时调度器是两个主要的比较对象，但也可以根据需求选择其他现有的调度策略进行对比。

- **目标 3: 提供详细的设计文档、实现指南和性能评估报告**

目标解读：需要撰写一份详细的设计文档，说明我们所开发的新调度器的设计思想、结构、算法以及其优化目标。这个文档应该详细描述设计过程和关键决策。还需要提供一份实现指南，详细描述如何将该调度器集成到 Linux 内核中。这包括必要的代码修改、编译步骤和配置选项。最后，我们需要编写一份性能评估报告，详细描述实验方法、性能数据的收集与分析过程，以及最终的对比结果。这个报告应该说明调度器的优点和缺点，并给出基于数据的结论。

这三个目标涵盖了调度器的设计、实现、性能评估以及文档撰写，要求我们在整个过程中都需要深入理解 Linux 内核的调度机制，并能够进行系统级的性能分析和优化。

1.3 项目要求实现内容

针对上述目标，项目指定的实现内容如下：

• 行动项 1: 开发环境编译

基于最新的内核版本，也即在 Linux 6.8 内核版本，Debian 12.5 发行版上编译内核，对内核做特定的定制，搭建 Yat-Scheduler 项目开发环境。

• 行动项 2: Yat-Scheduler 调度器整体框架的设计与实现

题目要求我们专注于 Linux 内核调度机制的创新与优化，支持多种硬件平台。因此我们需要从更宏观的视角去建立新的整体调度框架，而不只是局限在往 Linux 内部已经成型的框架里加功能。此外，为了支持多种硬件平台，我们需要在设计整体框架时充分考虑调度器在不同平台上的兼容性。

• 行动项 3: Yat-Scheduler 调度器各模块的设计与实现

在完成了调度器整体框架的设计后，我们紧接着需要设计调度器的各个模块，这预计会包含底层数据模块、调度跟踪模块、调度算法模块、同步处理模块、系统调用模块等五大模块，可能还会有其余的一些小模块需要实现。

• 行动项 4: 多核互斥资源共享协议的设计与实现

在多核环境中，任务可能会访问共享资源（如 I/O 设备、共享内存等）。为确保数据一致性和避免竞态条件，必须实现有效的互斥机制。互斥协议需要最小化锁争用和上下文切换带来的开销，同时确保高优先级任务不会被低优先级任务长时间阻塞（优先级反转问题）。

• 行动项 5: 面向资源的优先级排序算法的设计与实现

在实时系统中，任务的优先级排序不仅依赖于任务本身的重要性，还需考虑任务对共享资源的访问需求。因此我们所开发的调度器的目标是确保高优先级任务能够及时获得所需资源，同时减少任务间的优先级反转和阻塞时间。

• 行动项 6: 考虑阻塞的任务分配机制的设计与实现

在多核环境中，任务分配机制不仅要考虑负载均衡，还需考虑任务之间的阻塞关系，避免长时间的资源争用和阻塞。需要在任务分配时，充分利用各核的计算资源，尽量减少任务间的相互阻塞和等待时间。

• 行动项 7: 进行全面的测试与性能测试

为了验证调度器的正确性，我们需要对其进行全方位的功能测试。更进一步，为了验证我们开发的调度器在特定领域具有优秀性能，我们需要将其同 Linux 内置的 CFS 调度器以及其他一些 Linux 实时调度器项目进行性能上的对比。

• 行动项 8: 撰写详细的设计文档、实现指南和性能评估报告

软件工程课程教会我们，详细的设计文档和实现指南也是一个软件的重要组成部分，最能体现软件开发工程师的职业素养。为了让项目评审老师们能够通过阅读我们的项目设计文档，便可以理解我们所做的工作，我们将会提供详细的项目前置知识、严谨的算法建模以及形象的框架图或流程图；为了使我们的项目可以被复现，我们将会提供详细的项目实现指南；为了体现我们的调度器相较于其他调度器更优秀，我们将会提供项目的性能评估报告。

• **行动项 9 (可选) :** 将项目打包成补丁 (*patch*) 的形式向 **Linux** 社区提交开源贡献

若我们开发的调度器具有极其优秀的性能表现以及较好的鲁棒性，我们会考虑将其打包成补丁 (*patch*) 提交给开源社区，作为操作系统功能挑战赛成果产出的一部分。

1.4 项目预期实现成果

Yat-Scheduler, 全称 *Yet Another Toy Scheduler*, 一款基于最前沿的实时调度理论技术，面向全新 **Linux** 内核的具有高效性能的实时调度器。

整个调度器基于 *FPFPS (Fully Partitioned Fixed-Priority Scheduling)* 实现，由多核互斥资源共享协议、面向资源的优先级排序算法、考虑阻塞的任务分配机制三个主要技术部分组成。

1.5 项目目前完成情况

1. 完成了项目的调研和整体规划，这也是我们在初赛的预期任务目标

队伍成员在选定题目后，在指导老师的指导下，对于项目进行全面调研，同时学习项目所需的各种背景知识，形成一个项目的调研结果，对于项目需求有了一个基本的认识 and 了解。在有了对应的了解和认识之后，队伍成员进行了详细的分析和沟通，结合课内学习任务实际安排，提出了一个具有可行性的项目整体规划。

2. 初步实现了 **Yat-Scheduler** 的框架部分

由于课内课业实验压力较大，我们队伍三人能抽出用来开发时间不多，并且前期需要做大量的调研，设计调度器的整体实现框架、分析可行性，因此当前所实现的部分仅是我们整个项目的一小部分。在全国赛的第一阶段，我们将进行全面的项目完善。

1.6 初赛项目开发历程

预期内容	预期时间
前期准备工作：组队、联系项目导师、往届参赛学长介绍参赛经验	4.19-4.25
项目初步调研：可行性、实现方向、项目框架、技术栈，确定选题	4.26-5.2
项目深度调研：复现已有项目代码、论文，初步看懂实现逻辑，思考改进方式，深入调研 Linux 现有实时调度框架	5.3-5.9
进行了第一次项目知识研讨，首次添加项目文件，标准化项目文档	5.10-5.16
初步完成代码框架设计和试编译	5.17-5.23
仓库建设，初赛资料准备，撰写初赛文档	5.24-5.30

1.7 队伍简介

我们的队伍名为“上士闻道勤而习之”，三位成员和两位指导老师均来自于中山大学相关学院，有比赛相关的专业经验和经历，现将队伍成员和指导老师简要介绍如下。

1.7.1 队伍成员简介

- 队长**黄政菘**，中山大学计算机学院计算机科学与技术专业（系统结构方向）2021级本科生，个人主页 <https://hwyl.run/>，在系统领域有一定的探索经历。
- 队员**薛沐恩**，中山大学计算机学院计算机科学与技术专业（系统结构方向）2021级本科生，教育部“基础学科拔尖学生培养计划 2.0”成员，获得过美国大学生数学建模竞赛 *Meritorious Winner*，即优异奖(一等)以及 2022 年“移动云杯”算力网络应用创新大赛赛区优秀奖等专业相关竞赛奖励。
- 队员**刘洋**，中山大学软件工程学院软件工程专业 2021 级本科生。

1.7.2 项目导师简介

- **刘冬华**，国科环宇实时 Linux 系统领域专家，有多年 Linux 内核开发工作经验，参与开发实时系统望获 OS。

1.7.3 指导老师简介

- 指导老师**赵帅**，硕士生导师，“百人计划”引进副教授。本科（2008.09-2012.07）毕业于西安工业大学，硕士（2013.10-2014.10）、博士（2014.10-2018.08）毕业于英国约克大学。2018 年 10 月至 2022 年 9 月为英国约克大学计算机学院博士后。主要从事实时系统、操作系统领域的理论与应用研究，旨在围绕操作系统，为上层应用提供高性能、硬实时的计算与通讯保障，专注于复杂实时系统设计与分析、系统资源共享与管理、硬软件协同设计与寻优等具体方向。相关工作发表在 *DAC*、*RTSS* 与 *IEEE Trans. TDPS*、*TC*、*TCAD* 等国际顶级会议与重要期刊，获得约克大学计算机学院海外研究生奖与三次最佳论文提名。
- 指导老师**黎卫兵**，中山大学“百人计划”引进人才，2018 年获英国利兹大学机械工程博士学位，师从 *Robert Richardson* 教授（*Chair of the EPSRC UK-RAS network*, *Director of the EPSRC National Facility for Innovative Robotic Systems*），参与 EPSRC 资助的多个机器人相关科研项目。2018 年至 2020 年于香港中文大学周毓浩创新医学技术中心从事博士后研究工作，参与香港研究资助局（RGC）和香港创新科技署（ITC）资助的多个医疗机器人相关科研项目，旨在促进医学、工程学和计算机科学的多学科交叉融合，以期转化科技成果至临床应用。善于机器人运动学和动力学分析以及控制理论推导，具有熟练的 C/C++、Python 和 MATLAB 编程技能，擅长运用 ROS、Gazebo、V-REP、Webots、Solidworks 等专业软件进行机器人的设计、建模和控制，同时拥有丰富的硬件实现及调试经验。相关研究成果已发表在机器人及人工智能领域国际顶级期刊如 *IEEE/ASME TMECH*、*IEEE TIE*、*IEEE TMRB*、*IEEE RA-L*、*IEEE TNNLS*、*IEEE TCYB*、*IEEE TSMC*、*IEEE TII* 和机器人国际顶级会议 *ICRA* 及 *IROS* 中。

1.8 初赛项目分工

小组成员	分工内容
黄政崧	<ul style="list-style-type: none">• 确定开发环境和初赛目标• 确定项目整体架构并输出文档• 调研 <i>PREEMPT_RT</i> 和 <i>Litmus-RT</i> 框架• 整理问题问项目导师，和项目导师讨论实现细节• 建立新的调度器，并实现自定义锁机制• 设计多核互斥资源共享协议，定义接口算优先级和核心分配
薛沐恩	<ul style="list-style-type: none">• 实现面向资源的优先级排序算法• 调研实时系统下的调度算法以及 <i>Linux</i> 中实时调度算法的实现，并输出文档• 调研 <i>DMPO</i>、<i>OPA</i>、<i>RPA</i> 等目前已有的优先级排序算法，并输出文档• 设计实时系统下的基于 <i>Slack</i> 的优先级排序算法（<i>SPO</i>）• 复现前沿论文已有的实时系统下任务优先级排序算法，并输出文档
刘洋	<ul style="list-style-type: none">• 实现考虑阻塞的任务分配机制• 调研实时系统基本概念以及任务模型，并输出文档• 设计实时系统下考虑资源访问顺序的任务分配机制（<i>RAF</i>）• 复现前沿论文已有的实时系统下任务分配机制，并输出文档

2 实时系统知识

本小节主要介绍该项目的一些前置知识，同时也相当于展示我们初期调研的学习成果。主要概述实时系统知识和 Linux 下实时调度算法的实现，首先介绍实时系统的分类和任务模型，接着介绍实时系统下的调度算法的集中分类并介绍多核调度算法，然后介绍 Linux 中实时调度算法的实现，最后列举一些目前已有的实时调度工具/补丁。

2.1 实时系统一些基本概念

按照时间约束的程度，实时系统可以分为强实时系统和弱实时系统两种。

1) 强实时系统

强实时系统 (*Hard Real-Time System*, 也称硬实时系统): 在军事等关键领域中, 任务执行时间的约束性必须要得到完全满足, 否则就造成重大地安全事故。因此, 在这类系统的设计和实现过程中, 应采用各种方法, 保证在各种情况下时间约束性和功能需求都得到满足。

2) 弱实时系统

弱实时系统 (*Soft Real-Time System*, 也称软实时系统): 任务执行提出了时间约束性, 但是可以偶尔违反这种约束性, 并且对系统不会造成严重影响, 如视频系统就是弱实时系统。在视频系统中, 系统只需要保证绝大多数情况下视频数据能够及时传输给用户, 偶尔的数据传输延迟对使用不会造成大的影响。

本项目涉及到了 Linux 操作系统。Linux 操作系统是以一种分时操作系统, 有较好的平均响应时间和较高的吞吐量, 为了支持实时调度, 添加了两种调度算法: *SCHED_RR*, *SCHED_FIFO*。这两种并不是针对多核条件下的实时调度算法, 而是为任务赋予了实时任务的高优先级, 然后根据优先级不同进行调度的算法。而实时系统主要考虑任务按时完成, 尽量减少进程运行的不可预测性等。所以 Linux 不是一种专门的实时系统, 但与商业嵌入式操作系统相比, Linux 遵循 GPL, 具有源代码开放、定制方便、支持广泛的计算机硬件等优点, 所以通过修改内核调度部分, Linux 是可以很好地支持实时调度的。

2.2 实时系统中任务模型

在实时系统中, 有多个处理器, 用 m ($m \geq 1$) 表示。

在系统中, 用任务 (*task*) 表示进程或程序, 任务分为两种: 随机任务 (*sporadic task*) 和周期任务 (*periodic task*), 用 T_i 表示第 i 个任务, $1 \leq i \leq N$, N 是指任务总数目。

任务 (*task*) 包含一个或多个作业 (*job*), 用 jth 表示任务中第 j 个作业, 用 T_i^j 表示第 i 个任务的第 j 个作业。

对于不同的任务, 其在系统中调度的顺序, 是根据任务优先级 (*priority*) 的大小或者其他规则, 如先到先服务算法, 进行排序; 而对于一个任务中的不同作业, 其在该任务中的顺序是按照作业释放顺序排序的, 即先释放的作业先调度。

对于作业 T_i^j , 在系统中, 有以下几个基本时间概念:

1) **释放时间 (release time)**，即在释放时间之后的某个时间（可长可短），作业可以执行，用 $r(T_i^j)$ 表示。

2) **周期 (period)**，对于随机任务，周期是指任务中两个相邻作业的释放时间之间最短的时间间隔；对于周期任务，周期是固定的。对于一个任务的作业，周期是一个固定值，用 $p(T_i)$ 表示。

3) **执行开销 (execution cost)**，是指一个任务的所有作业的执行时间的最大值，即最坏情况下作业的执行时间。对于一个任务的作业，执行开销是一个固定值，用 $WCET$ (worst-case execution time) 表示。

4) **绝对死限 (absolute deadline)**，是指作业应该在绝对死限之前执行完毕，否则称为丢失死限，也称为延迟 (trady)。作业 T_i^j 的绝对死限为 $r(T_i^j) + p(T_i)$ ，那么作业 T_i^{j+1} 的释放时间 $r(T_i^{j+1}) \geq r(T_i^j) + p(T_i)$ 。如果对于一个任务的任意相邻两个任务总满足 $r(T_i^{j+1}) = r(T_i^j) + p(T_i)$ ，那么这个任务成为周期任务。

在这四个基本时间概念中，释放时间和绝对死限是每个作业所特有的，而周期和执行开销是一个任务的所有作业所共有的。图 2.1 表示了作业 T_i^j 的四个基本时间概念。

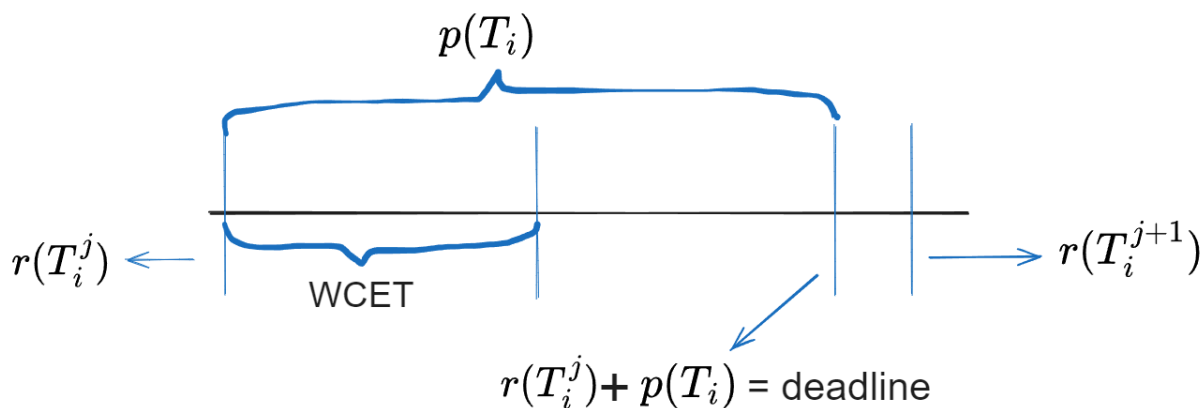


图 2.1 实时作业四个基本时间概念

如果在一个系统中，所有作业的死限都不允许丢失，那么这个系统称为硬实时系统 (hard real-time system)；如果一个系统允许作业的死限丢失，并且在死限之后的一段时间内保证该任务完成，那么这个系统称为软实时系统 (soft real-time system)。

以上是任务的理想调度模型，作业释放后，立刻执行等。而在实际环境中，由于物理硬件或者程序等原因，任务的调度过程中会经常出现延迟 (latency)，而延迟会出现在任务的释放与开始执行之间以及由于优先级不同导致任务抢占的过程中。所以，在实际跟踪实时任务的调度信息时，这些延迟必须考虑进去，否则将会影响一些数据的分析。在讲解延迟情况之前，先说明两个概念：1) 转入执行 (switch to)，就是作业被 CPU 调度，开始执行；2) 转出停止 (switch away)，就是作业不在被 CPU 调度，停止执行。下面介绍三种延迟：

1) 在空闲的 CPU 上，一个作业的释放与开始执行之间的延迟，完成与不再被 CPU 调度之间的延迟。图 2.2 说明了第一种延迟。



图 2.2 在空闲 CPU 上，任务调度的延迟

2) 在工作的 CPU 上，当前执行的任务的优先级高于已经释放的任务的优先级，那么新的任务必须等待旧的任务（当前执行的任务）执行完毕，才能执行。图 2.3 说明了第二种延迟。

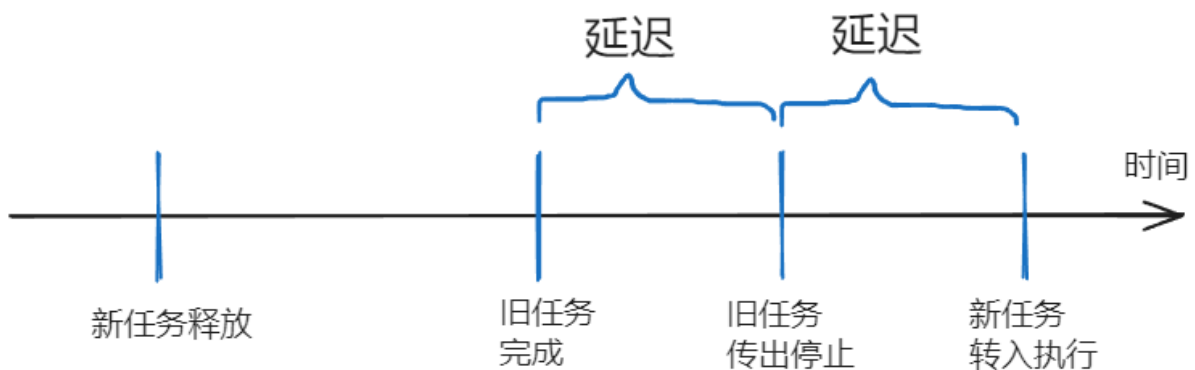


图 2.3 当前任务的优先级高于即将调度任务的优先级

3) 在工作的 CPU 上，当前执行的任务的优先级低于已经释放的任务的优先级，那么旧的任务必须停止，然后新的任务执行。图 2.4 说明了第三种延迟。

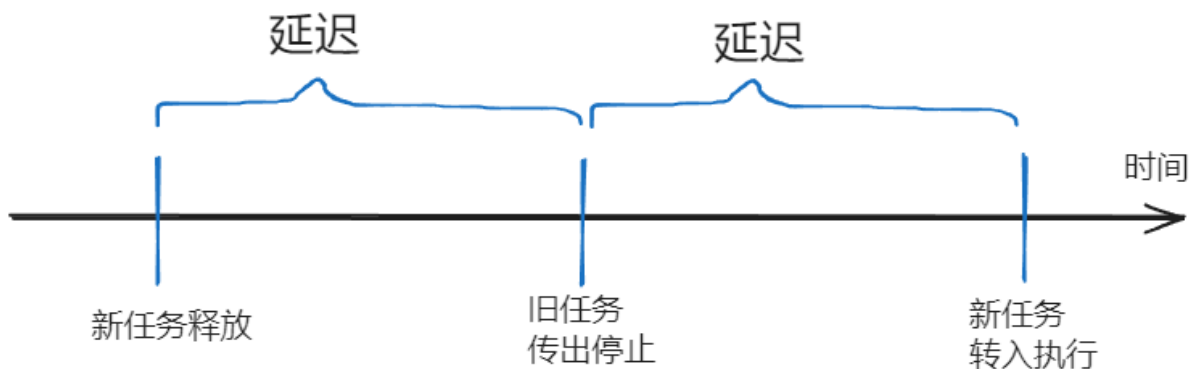


图 2.4 当前任务的优先级低于即将调度任务的优先级

任务在生成之后，会被添加到释放队列（*release queue*）中，如果条件允许，在释放队列中的任务会被添加到就绪队列（*ready queue*）中，供 CPU 调度。这在运行的任务位于运行队列（*run queue*）中。

2.3 实时系统下的调度算法

实时系统下的调度算法有很多，可以按照算法的某一特征分类。本部分将讨论按照优先级对调度算法进行分类，以及按照抢占发生的原因对调度算法分类。

2.3.1 按照优先级对调度算法进行分类

优先级的对象是在系统中调度的任务或者作业，而任务或作业的优先级是由调度算法参考任务或者作业的情况以及系统当时的环境根据相关的规则确定的。根据划分的优先级的特点，可以把调度算法分为 3 种：

(1) 静态优先级调度算法(*static priority scheduling*)

在这种调度算法下，任务一生成，其优先级就确定下来，并且不会改变，直至任务消亡，而且任务和作业具有相同的优先级。

RM (rate monotonic) 调度算法是这类调度算法的代表，该算法表示任务的释放频率（单位时间内释放几次任务）越大，优先级越高，使用的任务多是周期任务。由于是周期任务，任务的释放时间和释放频率是已知的，因此任务的优先级可以确定。

(2) 受限动态优先级调度算法(*restricted-dynamic-priority algorithms*)

这种调度算法又称为任务级别动态优先级、作业级别静态优先级调度算法 (*task-level dynamic-priority and job-level fixed-priority algorithms*)。任务中作业的优先级是在作业运行时确定，因此由于运行时情况不同，一个任务中不同作业的优先级可能不同，也可能相同，但是一旦作业的优先级确定下来，不会改变，直至作业消亡。

EDF (early deadline first) 调度算法是这类调度算法的代表，表示作业的死限时间越早，作业的优先级越大。所以在作业释放的时候，根据当时的调度情况，优先级可以确定，不再改变，直至作业消亡，并且由于每个作业释放的时候，调度情况是不相同的，所以每个作业的优先级也不是相同的。

(3) 不受限动态优先级调度算法(*unrestricted-dynamic-priority algorithms*)

这种调度算法又称为作业级别动态优先级调度算法 (*job-level dynamic-priority algorithms*)。任务中作业的优先级在作业运行过程中是不断改变的。

LLF (least-laxity-first) 调度算法是这类调度算法的代表。**LLF** 调度算法是指在不丢失死限的前提下，作业的执行可以被推迟的最大时间（也被称为作业的空闲时间）越大，优先级越低，因此根据在不同时刻，空闲时间的不同，作业的优先级不同。

2.3.2 按照抢占发生的原因对调度算法进行分类

在实时系统中，抢占的发生的原因有两种，第一种是由于一些外部事件的发生，如当前执行的任务的优先级低于已释放的任务的优先级，造成后者抢占前者；第二种是由于系统内部主动发出的抢占，如在规定的的时间点，系统发出抢占。因此，根据抢占发生的原因把调度算法分为两种。

(1) 事件驱动的调度算法(*event-driven scheduling*)

在这种调度算法下，抢占可能会发生在任意一个改变任务状态的事件中。如前边所提到的 **EDF**，就是事件驱动的调度算法。

这些事件种类很多，比如硬件发出的中断，任务的挂起和恢复等。

(2) 时间驱动的调度算法(*time-driven scheduling*)

在这种调度算法下，抢占只发生在预期定义的时间点，而这些时间点与运行时的动作无关。由于这些时间点是预期定义的，如果这些时间点是周期出现的，那么这些时间点是很容易确定的，抢占的发生也是周期事件。

最常见的例子是以量子为基础的调度算法 (*quantum-based scheduling*)，量子 (*quantum*) 又可称为时间片，同时在时间片边界处发生抢占。

2.3.3 多核调度算法

根据任务队列与 CPU 之间的关系，多核调度算法分为 3 种：

(1) 局部性多核调度算法 (*partitioning multi-core schedule*)

这种调度算法的思想是把运行任务集合中的任务分割成与处理器数目相同个数的运行队列，每个运行队列一直运行在某个处理器上，每个任务（包括所有作业）一直运行在某个处理器上，即一个处理器只对应一个运行队列。这样的运行队列相对于处理器，被称为本地运行队列。处理器在其本地运行队列上使用的调度算法可以是相同的，也可以是不同的。*P-EDF* (*partitioning early deadline first*) 是这种调度算法的代表。图 2.5 说明了这种调度算法的思想。

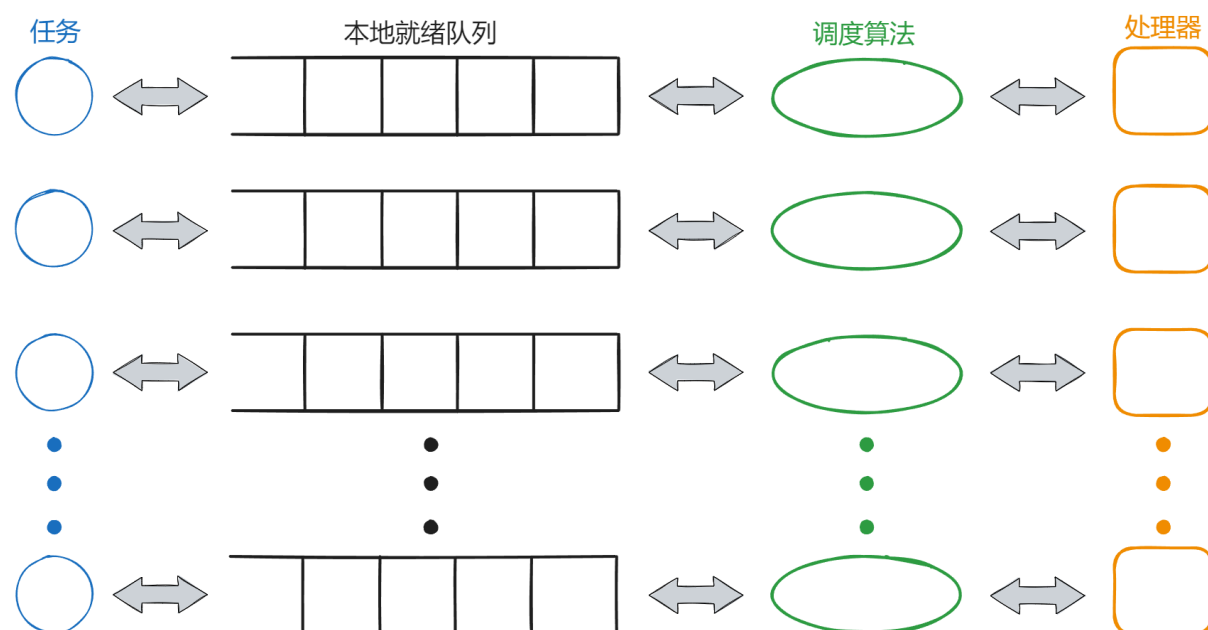


图 2.5 局部性多核调度算法

(2) 全局性多核调度算法 (*global multi-core schedule*)

这种调度算法的思想是这个系统只有一个运行队列，即全局运行队列 (*global run queue*)，符合运行条件的任务会被添加到全局运行队列中，然后由唯一的调度算法调度，任务的某一个作业被安排到某一个处理器上执行，在一段时间后，这个作业有可能迁移 (*migration*) 到另外一个处理器上执行，也就是说，一个作业在不同时刻有可能运行在不同的处理器上，一个任务的两个或多个作业也有可能运行在不同的处理器上。G-

EDF (*global early deadline first*)、PFair (*proportionate fair*) 是这种调度算法的代表。图 2.6 说明这种调度算法的思想。

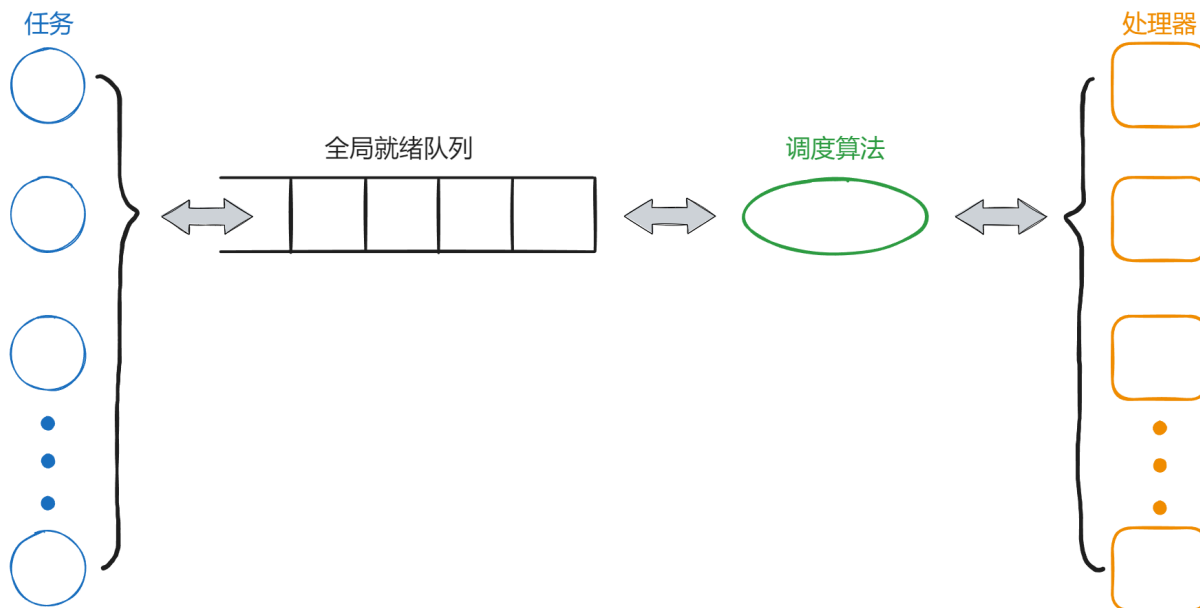


图 2.6 全局性多核调度算法

(3) 两级别混合调度算法 (*two-level hybrid multi-core schedule*)

顾名思义, 这种调度算法是以上两种算法的结合。这种调度算法的思想是: 一、第一级别调度, 使用全局性调度算法, 即运行任务都被安排在全局运行队列中, 把不同的作业安排到不同的处理器上; 二、第二级别, 使用局部性调度算法, 即把由第一级别调度安排的全局运行队列安排到不同本地运行队列中, 不再迁移到别的处理器上或别的运行队列中, 然后使用局部性调度算法把作业安排到相对应的处理器上执行。由于复杂, 实现起来比较麻烦, 这种调度算法目前没有代表性的调度算法。图 2.7 说明了这种调度算法的思想。

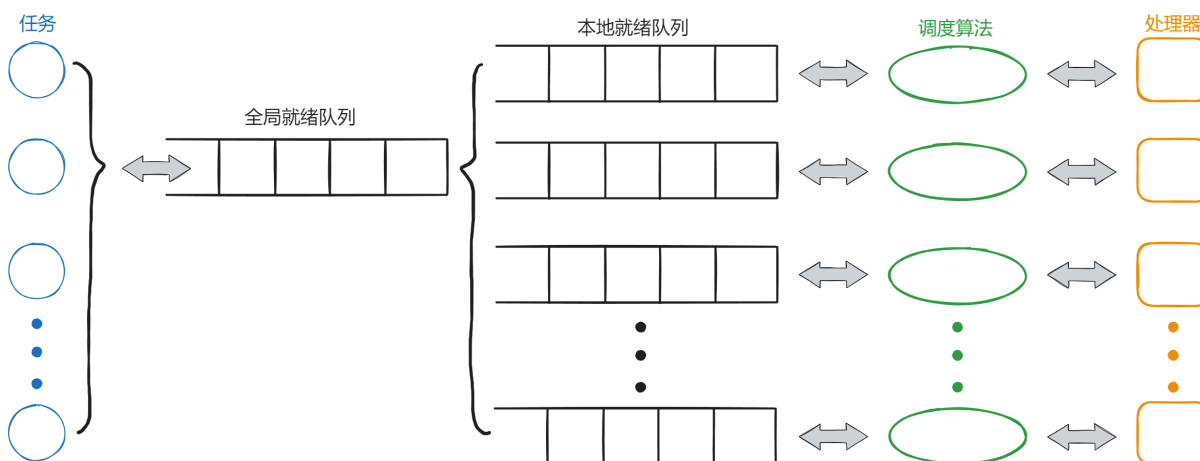


图 2.7 两级别混合调度算法

2.4 Linux 中实时调度算法的实现

Linux 内核中有两种实时调度算法：*SCHED_RR*，*SCHED_FIFO*。这两种算法并不是针对多核条件下的实时调度算法，而是根据优先级不同进行调度的算法。

1. ***SCHED_FIFO***: 先到先服务算法。一旦一个任务使用处理器，则会一直执行，直到更高优先级的任务到达或者该任务自动放弃处理器。对于相同级别的任务，系统不会发出抢占，而是等当前任务执行完毕或者调度更高优先级的任务，再去执行其他任务。
2. ***SCHED_RR***: 时间片轮转算法。当一个进程首次执行之前，系统会为该进程分配一定量的时间片，并且在该进程执行时减少相应时间片的数量；当进程的时间片使用完毕后，系统会重新为该进程分配一定量的时间片，并且把该进程放在就绪队列的队尾，这样就能保证系统中每个进程都能有一定量的时间运行，在一定程度上保证了调度的公平性。*SCHED_RR* 支持由于优先级不同而发生的抢占，即如果当前执行的任务是低优先级的，那么高优先级的任务可以抢占处理器，对于相同级别的任务，任务的执行依赖于系统分配的时间片，所以 *SCHED_RR* 可以保证相同优先级任务的调度的公平性。

在 Linux 系统中，每个进程通过系统调用 *sched_setscheduler()* 选择执行的调度算法。

1. 当进程选取 *SCHED_FIFO* 调度算法时，其调用情况为：

- (1) 创建任务时通过 *sched_setscheduler()* 选择 *SCHED_FIFO* 调度算法，并设置实时优先级 *rt_priority*(1-99)
- (2) 如果没有等待资源，则将该任务加入到就绪队列中
- (3) 调度程序遍历就绪队列，根据实时优先级计算调度权值($1000 + rt_priority$)，选择权值最高的任务使用 CPU，该 *FIFO* 任务将一直占有 CPU 直到有优先级更高的任务就绪或者主动放弃(如等待资源)
- (4) 调度程序发现有优先级更高的任务到达(高优先级任务可能被中断或定时器任务唤醒，再或被当前运行的任务唤醒，等等)，则调度程序立即在当前任务堆栈中保存当前 CPU 寄存器的所有数据，重新从高优先级任务的堆栈中加载寄存器数据到 CPU，此时高优先级任务开始运行，然后重复第 3 步
- (5) 如果当前任务因等待资源而主动放弃 CPU 使用权，则该任务将从就绪队列中删除，加入等待队列，然后重复第 3 步

2. 当进程选取 *SCHED_RR* 调度算法时，其调用情况为：

- (1) 创建任务时通过 *sched_setscheduler()* 选择 *SCHED_RR* 调度算法，并设置任务的实时优先级和 *nice* 值(*nice* 值将会转换为该任务的时间片的长度)
- (2) 如果没有等待资源，则将该任务加入到就绪队列中
- (3) 调度程序遍历就绪队列，根据实时优先级计算调度权值($1000 + rt_priority$)，选择权值最高的任务使用 CPU
- (4) 如果就绪队列中的任务时间片为 0，则会根据 *nice* 值重新设置该任务的时间片，同时将该任务放入就绪队列的末尾，然后重复步骤 3

(5) 当前任务由于等待资源而主动退出 CPU，则其加入等待队列中，重复步骤 3

2.5 现有研究调研概况

以下是我们在项目初期调研时所了解到的一些 Linux 上现有 *State of the art* (SOTA) 的 *Real-Time scheduling*

2.5.1 **RTLinux, RTAI**

RTLinux 是一个针对标准 Linux 内核的扩展，旨在为实时应用提供支持。它的开发目的是使 Linux 内核能够处理实时任务，而不仅仅是传统的非实时任务。*RTLinux* 采用了“中断抽象”的方法。这个方法的核心思想是在标准 Linux 内核和计算机硬件之间创建一个虚拟硬件层，称为“实时硬件抽象层” (*Real-Time Hardware Abstraction Layer*, *RTHAL*)。这个层次实际上只虚拟化了中断。每个来自真实硬件的中断源都被标记为实时或非实时。实时中断由实时子系统处理，而非实时中断由 Linux 内核管理。这意味着 *RTLinux* 能够识别并处理需要实时响应的中断，而将不需要实时响应的中断交给 Linux 内核处理。

最初，*RTAI* 是 *RTLinux* 的一个变种，旨在为 Linux 内核添加实时特性。尽管 *RTAI* 最初是基于 *RTLinux* 代码启动的，但两个项目的 API（应用程序编程接口）在不同的方向上进行了演化。主要开发者（教授 *Paolo Mantegazza*）对 *RTAI* 代码进行了重写，添加了新功能，并创建了一个更完整和强大的系统。因此，*RTAI* 和 *RTLinux* 之间存在差异。*RTAI* 社区还开发了 *Adaptive Domain Environment for Operating Systems* (*ADEOS*) 纳米内核，作为 *RTAI* 核心的替代方案。*ADEOS* 纳米内核采用了管道方案，其中每个域（操作系统）都有一个预定义的优先级入口。*RTAI* 是最高优先级的域，始终在 Linux 域之前处理中断，因此能够为任何硬实时活动提供服务，无论是在之前还是在完全抢占非硬实时任务之前。

2.5.2 **OCERA**

OCERA 是一个欧洲项目，专注于嵌入式实时应用程序。它采用开源方法，旨在为嵌入式系统提供一种综合的执行环境。在 *OCERA* 项目中，开发了一个针对 Linux 2.4 内核的实时调度器。这个调度器旨在允许嵌入式系统实时执行任务，以满足实时性要求。为了尽量减少对内核代码的修改，实时调度器被开发为一个小型补丁 (*patch*) 和一个外部可加载的内核模块。补丁的作用是通过一些钩子 (*hooks*) 将相关的调度事件导出给外部模块。这种方法的好处是简单灵活，但需要确保在内核中放置这些钩子的位置，这可能会成为挑战。由于钩子的位置可能会随内核版本的变化而发生变化，因此将实时调度器的代码移植到内核的新版本可能会很困难。这是因为内核的不断演进可能会导致钩子的位置和接口发生变化。

2.5.3 **FRESCOR**

FRESCOR 是一个研究项目，由欧洲联盟的第六框架计划部分资助。项目旨在开发技术和基础设施，以支持实时应用程序的设计和嵌入式系统中的应用，这些应用具有灵活的调度需求。个框架基于 *AQuoSA* (*Adaptive Quality of Service-driven Scheduling Architecture*) 并进一步添加了基于合同的 API 和一个复杂的中间件，用于指定和管理系

统的性能, 尽管 *FRESCOR* 项目提出了一种复杂的实时框架, 但它也面临着上述提到的一些问题和挑战, 这包括与内核版本变化相关的问题以及其他可能的挑战。

2.5.4 *Litmus-RT*

多核处理器的发展促进了操作系统对多核支持的研究, 其中重要的一点是调度算法的研究。由于各个调度算法的测试和性能比较没有一个统一的、合理的实际系统环境, 比如是否使用相同体系结构的 CPU, 是以应用程序的形式还是以系统内核的形式实现调度算法的测试, 这些都造成了结果的可信度不高。因此一种多核实时调度平台-*Litmus-RT* 用于在统一的系统环境下以系统内核的形式实现调度算法的测试和性能的比较。

Litmus-RT 平台由美国北卡罗来纳大学教堂山分校计算机科学学院 James H. Anderson 教授和他的多核实时调度小组研究和开发, 其全称是 *Linux Testbed for Multiprocessor Scheduling in Real-Time Systems*, 是一种基于 Linux 内核、针对多核实时系统背景、测试多核调度算法性能的一种测试平台。*Litmus-RT* 的设计初衷是在统一的实时系统条件下, 对不同类型的多核调度算法的性能、约束性、适用范围等方面进行横向比较, 具体为分别使用不同的调度算法, 对相同量负载的任务进行调度测试, 然后在一些指标数据上比较、分析。*Litmus-RT* 的实现包括三个模块: 一、底层数据模块, 如任务节点的链接、排序等等使用红黑树表示; 二、调度信息跟踪模块, 这是基于 Linux 本身的跟踪功能和时钟功能实现的模块, 已经实现了文字和图形两种表现形式, 不过对不同的多核调度算法的支持度不足; 三、调度算法插件模块, 实现了不同类型的、共 5 种调度算法。*Litmus-RT* 平台还包括其他工具, 用于分析调度算法的性能。总而言之, *LitmusRT* 是一个基于 Linux 的实时调度平台, 其提供了一些实现了不同实时调度算法的调度器插件以供我们进行实验, 也暴露了一些接口以让我们实现自己的调度插件。

Litmus-RT 虽然功能相较全面且支持调度算法自定义, 但由于其定位是一套为研究人员测试调度算法而开发的框架, 并不面向实际的生产环境, 还需要考虑兼容不同的由用户自己编写的调度算法, 因此无法做到针对性地深入内核调度机制去优化特定的调度算法, 且其早已于 2017 年停止维护, 最新支持的内核版本永远停留在了 4.9.30 版本, 无法支持当前一些新的内核特性。

2.5.5 *Preempt_rt*

“*Preempt-RT*”通常指的是 Linux 内核的 *Preemptive Real-Time* (抢占式实时) 补丁。这个补丁是针对 Linux 操作系统的一种改进, 它提供了更细粒度的抢占式多任务处理能力, 从而改善了实时性能。

在标准的 Linux 内核中, 调度器通常是基于分时的, 这意味着 CPU 时间被分配给系统中的各个任务, 每个任务运行一定的时间后会被调度器挂起, 让其他任务运行。虽然这对于大多数应用来说已经足够, 但对于需要严格时间保证的实时任务来说, 这种调度方式可能不够。

标准的 Linux 内核中不可中断的系统调用、中断屏蔽等因素, 都会导致系统在时间上的不可预测性, 对硬实时限制没有保证。目前, 针对 *real-time Linux* 的修改有两种成功的方案。一是直接修改 Linux 内核, 使其直接具有 *real-time* 能力; 另一是先运行一个

real-time 核心, 然后将 Linux 内核作为该 *real-time* 核心的 *idle task* 来运行。前者则称为 *PREEMPT-RT kernel*

PREEMPT-RT Patch 的核心思想是最小化内核中不可抢占部分的代码, 同时将为支持抢占性必须要修改的代码量最小化。对临界区、中断处理函数、关中断等代码序列进行抢占改进。

PREEMPT-RT 特性:

- 临界区可抢占
- 中断处理函数可抢占
- “关中断”代码序列可抢占
- 内核中的 *spinlock* 和 *semaphore* 支持优先级继承
- 延迟操作
- 降低延迟的措施

PREEMPT-RT 把 Linux 变成一个完全可抢占的内核, 改变有以下几点:

1. 通过 *rt_mutex* 的重新实现使内核里的锁源语可被抢占。以前被如 *spinlock_t* 和 *rwlock_t* 保护的临界区现在变得可以被抢占了
2. 为内核里的自旋锁和信号量实现优先级继承(*PI-Priority Inheritance*)
3. 把中断处理器变为可被抢占的内核线程: *PREEMPT-RT patch* 在内核线程上下文中处理软中断处理器

2.6 本项目的实时调度系统的设计

本项目是基于 Linux 开发的多核实时调度算法。由于我们队伍三人在这方面的开发经验尚缺, 在开发的过程中, 更希望参考一些主流的、成熟的、面向多核系统的 *Real-Time Scheduling* 的实现。*Preempt-rt* 和 *Litmus-rt* 和我们项目做的方向比较符合。但是 *Preempt-rt* 和 *Litmus-rt* 是互相冲突的, 我们需要参考其中一个框架来实现我们自己的多核实时调度系统。

由于 *Litmus* 是面向多核的实时调度工具, 经过考虑, 我们实现的也是多核调度算法, 所以策略更接近于 *Litmus-rt*, 实现的时候更多参考 *Litmus-rt* 实时调度系统的框架。

在多核调度算法的选择中, 我们希望设计一个简洁高效且灵活的算法, 所以采用局部性多核调度算法的模式。在局部性多核调度算法框架下改进(优化优先级排序算法, 实现考虑阻塞的任务分配机制等), 以优化其性能并得到我们自己的调度器。

其他的具体细节会在下文补充。

3 调度器整体结构设计

本节从设计的角度来看 *Yat-Scheduler* 整个项目，包括 *Yat-Scheduler* 项目最初的设计理念，以及最顶层的系统架构上的设计。

3.1 设计理念

我们将 *Yat-Scheduler* 初步定位为一款支持 X86 和 ARM 等体系结构、基于 Linux 内核的硬实时调度器，将会应用**最前沿**的实时调度理论技术，面向**最新**的 Linux 6.8 内核，实现目前 Linux 内核实时调度下的**最高**性能。基于 Linux 内核是指 *Yat-Scheduler* 将改造 Linux 内核的调度机制，支持随机任务模型，提供新的同步算法。

3.2 项目架构介绍

Yat-Scheduler 预计会由五部分组成：底层数据结构部分、调度算法部分、同步处理机制部分、调度跟踪部分、系统调用部分。图 3.1 表示了各个部分及其联系：

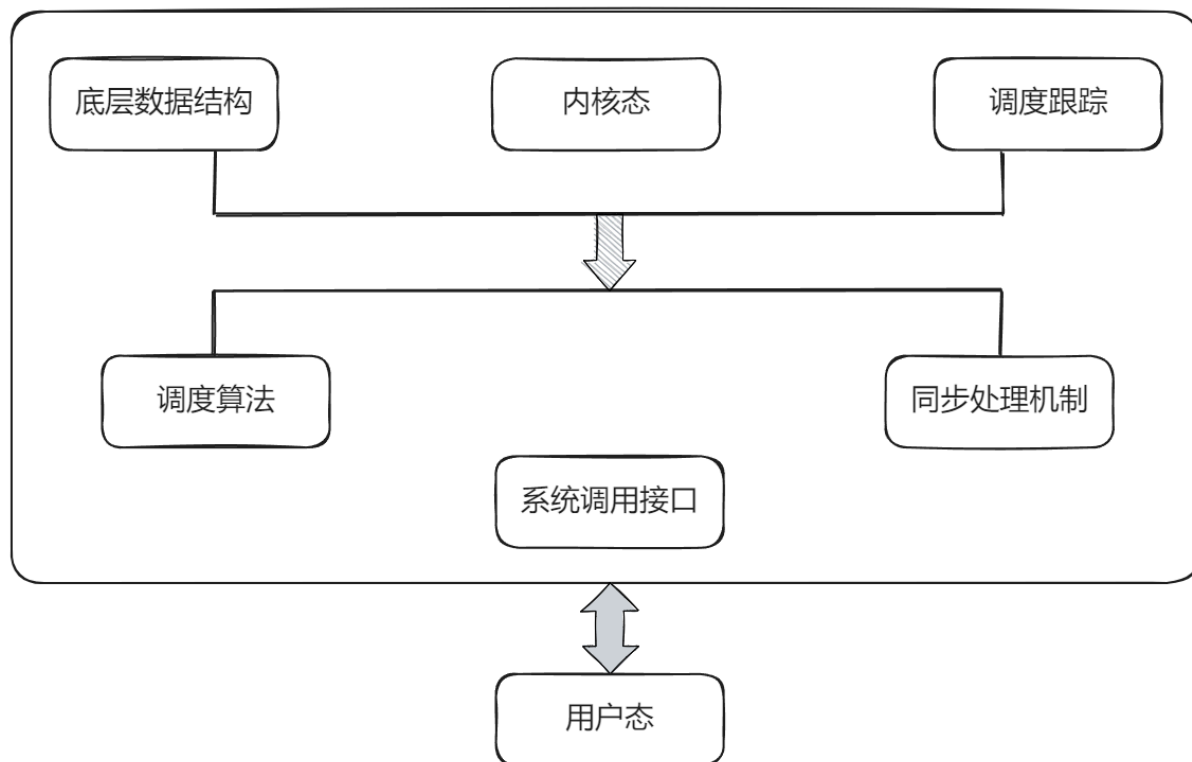


图 3.1 *Yat-Scheduler* 项目架构图

底层数据结构部分和调度跟踪部分是 *Yat-Scheduler* 支持部分。底层数据结构负责调度算法和同步处理机制数据结构的实现，调度跟踪部分使用 Linux 计数器负责调度信息记录的实现，为后续的性能测试提供数据信息支持。

调度算法部分和同步处理机制部分是 *Yat-Scheduler* 处理部分。调度算法部分包括调度算法（含面向资源的优先级排序算法）的实现，并整合为单独的模块存在于 *Yat-Scheduler* 中；同步处理机制部分将实现基于多核互斥资源共享协议的同步机制。

系统调用接口将提供多个系统调用供调度算法进行调用，用来编写实时任务程序。

在用户态, 我们将会设计实现最友好的用户交互体验, 支持全面的任务参数设定, 并支持以可视化窗口提交实时作业

4 Yat-Scheduler 设计和实现

本节具体介绍 Yat-Scheduler 项目的设计和实现，主要内容包括支持 Yat-Scheduler 的内核编译和开发环境的搭建以及 Yat-Scheduler 各个子模块的具体设计和实现。

4.1 Linux 内核开发环境选择和搭建

4.1.1 项目环境

Linux 发行版	Linux 内核版本	环境目的
Debian12.5	6.1.0-21-amd64	开发
RaspBerry Pi OS	6.6.31	性能测试

4.1.2 项目环境说明

在内核版本的选择上，因为根据前面的调研我们了解到，当前没有一款强有力的实时调度框架能够支持高版本的 Linux 内核，为了使我们开发的调度器能够作为补丁合入开源社区，同时也为了该调度器能够适配当前主流的内核技术，我们需要选择最新的内核版本，也即我们项目开始执行时最新的 Linux Kernel 6.8 版本。

在开发环境的选择上，Debian 相较于 Ubuntu 具有更加轻量、稳定、简洁等优点，因此我们选择当前最新的 Debian 发行版 12.5 作为我们的开发环境。

在测试环境的选择上，因为我们当前可用的资源比较有限，所以只好使用校内导师能够提供的树莓派作为调度器的测试环境。

4.1.3 项目环境搭建概述

本小节介绍我们初赛环境配置的相关内容

(1) 下载安装 debian 12.5

在 Debian 官网 <https://www.debian.org/> 点击 Download 下载最新版的 Debian 虚拟机镜像，在 VMware Workstation 新建 100G 磁盘，4G 内存的虚拟机，使用下载好的 debian-12.5.0-amd64-netinst.iso 镜像文件

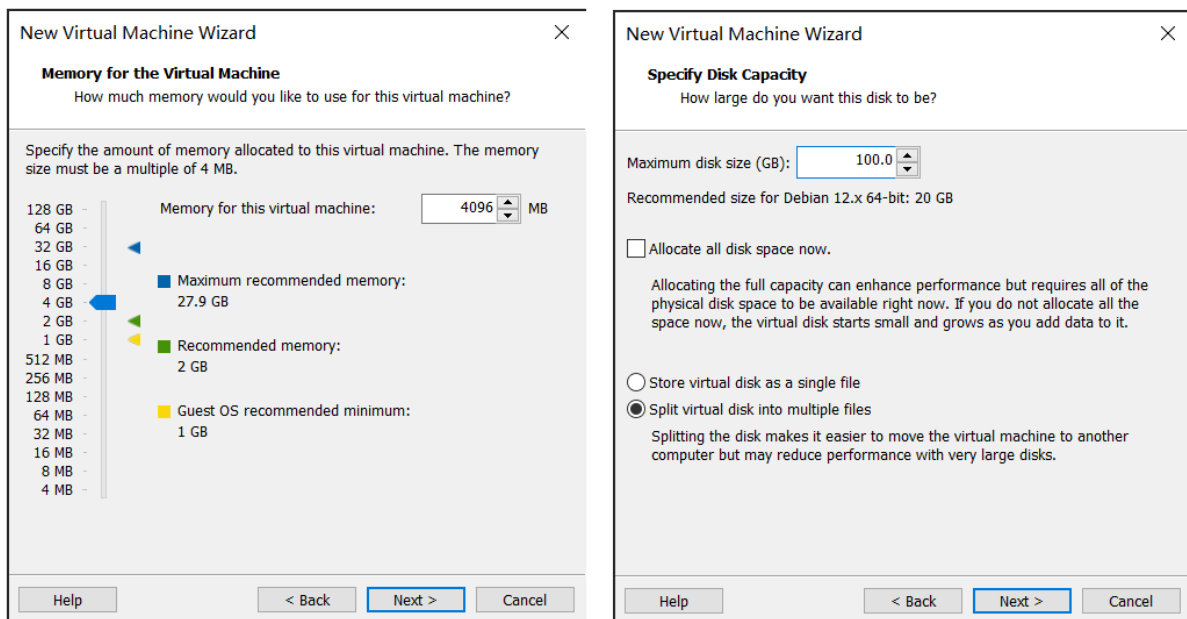


图 4.1 虚拟机容量与内存配置

启动虚拟机后选择 *Debian installer*，然后根据它的指引安装，设置本机名称为 *debian*，*root* 密码为 *root*，*user* 名为 *yat*，密码为 *yat*，其他均为默认即可。其中需要注意的是，在下载源的选择上需要选择国内的镜像源，否则会导致后续包的下载速度较慢。

将操作系统安装完，进入 *root* 用户登录，安装配置 *sudo*：

```
su root
ll /etc/sudoers
chmod u+w /etc/sudoers
vim /etc/sudoers ...
chmod 440 /etc/sudoers
```

其中 *vim* 编辑 *sudoers* 文件的步骤中，需要在 *root ALL=(ALL:ALL) ALL* 的行下面添加 *yat ALL=(ALL:ALL) ALL*，然后保存退出，此时再进入 *yat* 用户便可以使用 *sudo* 了。

(2) 安装必需的第三方软件包

Debian 默认自带的开发环境比较单一，无法支持我们本项目的开发，因此我们需要额外安装一些开发工具链，只需通过以下命令一键安装：

```
sudo apt update && apt upgrade
sudo apt install -y ssh vim git gcc curl zsh neofetch build-essential libncurses-
dev flex bison libssl-dev libelf-dev dwarves qemu-system-x86 gdb
```

其中 *ssh*，*vim*，*git*，*curl*，*zsh*，*neofetch* 等工具有助于更加高效便捷地进行开发，*gcc* 和 *gdb* 是 *Linux* 环境下进行编译和调试必不可少的，剩下的 *build-essential*，*libncurses-dev*，*flex*，*bison*，*libssl-dev*，*libelf-dev*，*dwarves*，*qemu-system-x86* 则是开发、编译、运行和调试 *Linux* 内核所必需的。

(3) 下载对应 *Linux* 内核源码

在阿里云开源镜像站找到 Linux 6.8 内核的下载地址 <https://mirrors.aliyun.com/linux-kernel/v6.x/linux-6.8.tar.xz>, 即内核 6.8 的源码, 回到虚拟机中使用 `wget` 命令下载, 下载完毕后使用 `tar -xvf` 指令进行解压

(4) 更改配置选项, 编译 Linux 内核

进入解压后的 Linux 内核文件夹, 执行 `make menuconfig` 打开配置选项菜单

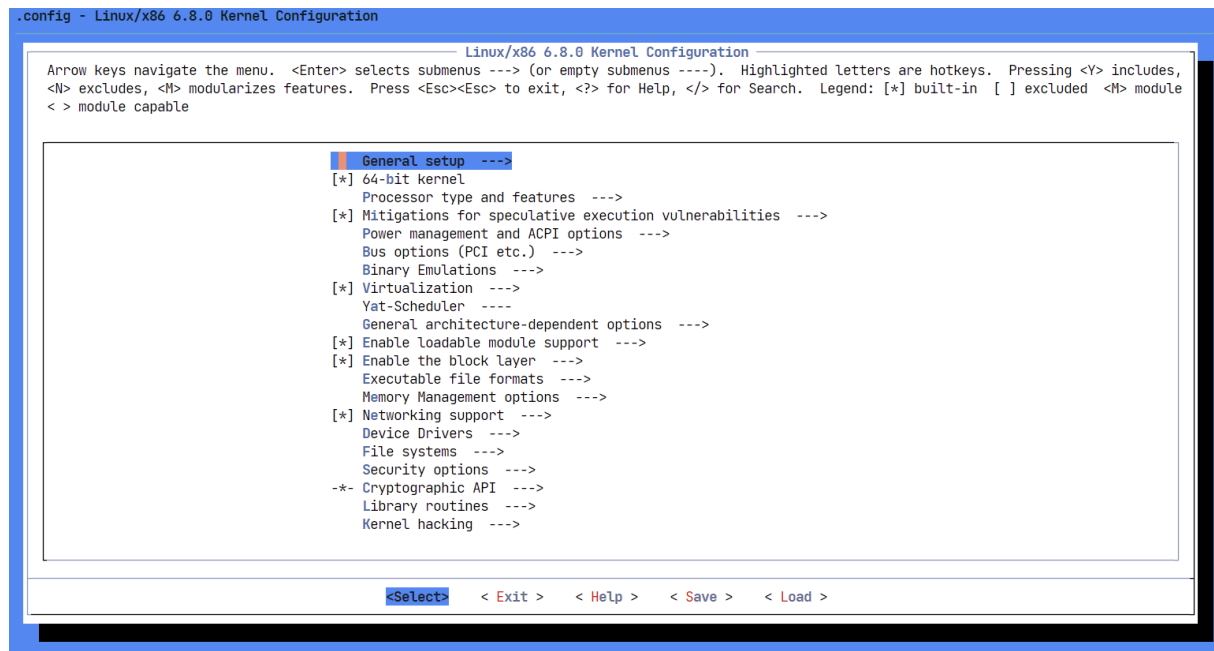


图 4.2 内核配置选项菜单

为了使得 Linux 内核能够最大限度地支持实时调度, 我们需要更改一些默认的配置选项:

- 启用内核抢占
 - 进入 "General setup -> Preemption model" 目录, 选择 "Preemptible Kernel (Low-Latency Desktop)"
- 禁用群组调度
 - 首先, 进入 "General setup", 禁用 "Automatic process group scheduling", 然后在 "General setup -> Control group support -> CPU controller" 目录下, 禁用 "Group scheduling for SCHED_OTHER"
- 禁用影响计时器频率的 CPU 频率缩放和一些电源管理选项
 - 在 General setup -> Timers subsystem -> Timer tick handling 目录下, 更改选项为 "constant rate, no dynticks"
 - 在 "Power management and ACPI options" 目录下, 禁用 "Suspend to RAM and standby", "Hibernation" 和 "Opportunistic sleep" 这三个选项
 - 在 "Processor type and features" 目录下禁用 "CPU Core priorities scheduler support" 选项, 然后在 "Power management and ACPI options -> CPU Frequency scaling" 目录下, 禁用 "CPU Frequency scaling"

保存退出, 然后执行 `make bzImage -j 4 > log` 指令编译内核, 整个过程会持续约 10 分钟。待编译完成后, 进入 `arch/x86_64/boot` 目录, 执行指令 `mkinitramfs -o initrd.img` 制作 `initrd` 镜像文件。至此, 内核运行前的准备工作已完成。

(5) 搭建基于 VSCode 的内核调试环境

本项目我们选择 VSCode 作为我们的代码编辑器。VSCode 全称 *Visual Studio Code*, 是一款跨平台的、免费且开源的现代轻量级代码编辑器, 支持几乎主流开发语言的语法高亮、智能代码补全、自定义快捷键、括号匹配和颜色区分、代码片段提示、代码对比等特性, 也拥有对 `git` 的开箱即用的支持。同时, 它还支持插件扩展, 通过丰富的插件, 用户能获得更多高效的功能。VSCode 是当今主流的代码编辑平台, 配置好 VSCode 能使我们内核开发的效率大幅提升。

首先打开 PC 端的 VSCode, 点击左下角“打开远程窗口”, 输入 `yat@192.168.88.128` 后按回车键, 即可通过 VSCode 访问虚拟机上的文件及目录。

在 `vscode` 工作目录下创建 `.vscode/launch.json`, 写入以下 `gdb` 配置内容并保存:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Kernel-Debug",
      "type": "cppdbg",
      "request": "launch",
      "miDebuggerServerAddress": "127.0.0.1:1234",
      "program": "${workspaceFolder}/Yat_kernel/vmlinux",
      "args": [],
      "stopAtEntry": false,
      "cwd": "${workspaceFolder}",
      "environment": [],
      "externalConsole": false,
      "logging": {
        "engineLogging": false
      },
      "MIMode": "gdb",
    }
  ]
}
```

回到虚拟机命令行界面 (也可以通过 VSCode 窗口下方的命令行窗口操作虚拟机系统) 进入 `arch/x86_64/boot` 目录并执行 `qemu-system-x86_64 -kernel bzImage -initrd initrd.img -append nokaslr -m 1G -S -s`, 启动 `qemu` 调试内核。

然后在 VSCode 左侧栏找到“Run and Debug”, 在最后的“BREAKPOINTS”处手动添加断点“`schedule`”, 然后按 `F5` 运行, 即可发现下图所示内核的启动成功在 `schedule` 函数下断住了, 之后我们便可以通过 VSCode 自带的单步跳过 (`F10`) 和单步步入 (`F11`) 等调试按钮进行方便的调试了, 并且左侧栏也可以显示当前断点的相关变量值以及堆栈信息。

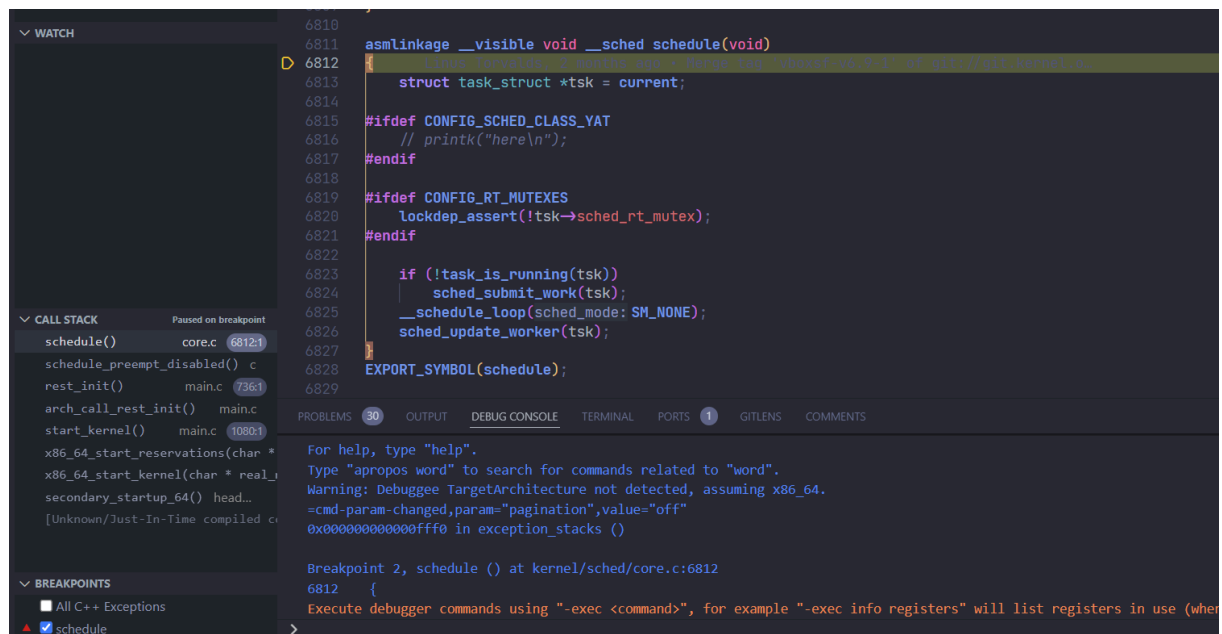


图 4.3 使用 VSCode 打断点并调试

至此，基于 VSCode 的 Linux 内核开发环境已搭建完毕。

(6) (可选) 配置 VSCode 的 `clangd` 插件，方便阅读代码

Microsoft 官方的 C++ 插件虽说能满足平常的代码高亮和定义跳转需求，但面对 Linux 内核这样的大型项目还是会显得有些卡顿，将鼠标移到函数上方常常需要等待几秒钟才显示函数的定义，更别说跳转到定义了。这极大影响了我们开发的效率。

因此我们通过配置基于 LLVM 的 `clangd` 插件，极大地改善了代码阅读效率。

因为该部分不是我们环境配置的重点，这里就不过多阐述了，具体的配置步骤可以参考网上相关的教程（如 <https://blog.csdn.net/ludaoyi88/article/details/135051470>）

4.2 Yat-Scheduler 初始化

Yat-Scheduler 是作为一种设备驱动添加到 Linux 内核中的，通过用户选择切换调度器，唯一地运行一种调度算法，调度跟踪部分只跟踪任务调度算法被设置为 `SCHED_YAT` 的实时任务。所以 Yat-Scheduler 的初始化是设备驱动程序的初始化，初始化函数设置为 `_init`，然后由 `module_init` 加载。

初始化部分将由多个函数共同完成，分为调度、跟踪初始化、跟踪工具、同步机制四类。其中调度初始化包括调度算法及支持的数据结构的初始化以及调度算法预算功能的初始化，跟踪初始化包括设备节点的初始化，Yat-Scheduler 调度信息文件的初始化，同步机制初始化包括 `srp` (*stack-based resource allocation protocol*, 以栈为基础的资源分配协议) 的初始化。

4.2.1 调度初始化

首先添加配置选项 `CONFIG_SCHED_CLASS_YAT` 和方法 `set_sched_class_yat_on()` 在 `init/Kconfig` 添加：

```

config SCHED_CLASS_YAT
    bool "YAT(Yet Another Toy) sched class"
    default n
    help
        Allow user to create "fully partitioned fixed-priority scheduling" task.

```

定义了一个名为 `SCHED_CLASS_YAT` 的内核配置选项。用户可以通过 `make menuconfig` 或者直接编辑 `.config` 文件来启用或禁用此调度类。如果启用（设为 `y`），这会允许用户创建一种“*fully partitioned fixed-priority scheduling*”（完全分区的固定优先级调度）任务。

在 `init/main.c` 添加：

```

unsigned int sched_class_yat_on;
static int __init set_sched_class_yat_on(char *str)
{
    sched_class_yat_on = 1;
    return 1;
}
early_param("fully-partitioned-fixed-priority-scheduling_class",
set_sched_class_yat_on);

```

`unsigned int sched_class_yat_on`；定义了一个全局变量 `sched_class_yat_on`，用于指示“*fully partitioned fixed-priority scheduling*”类是否启用。`static int __init set_sched_class_yat_on(char *str)` 定义了一个初始化函数，用于设置 `sched_class_yat_on` 变量。当被调用时，该函数会将 `sched_class_yat_on` 设置为 1，并返回 1。`early_param("fully-partitioned-fixed-priority-scheduling_class", set_sched_class_yat_on)`；使用 `early_param` 宏将内核参数“*fully-partitioned-fixed-priority-scheduling_class*”关联到初始化函数 `set_sched_class_yat_on`。这意味着如果内核启动参数中包含“*fully-partitioned-fixed-priority-scheduling_class*”，那么在内核初始化早期阶段会调用 `set_sched_class_yat_on` 函数，设置 `sched_class_yat_on` 为 1。

上述代码实现了一个可配置的调度类（YAT），并通过内核启动参数控制其启用状态。在内核启动时，如果检测到指定的启动参数，则会启用此调度类。这样设计便能根据需要启用或禁用特定的调度行为，而无需重新编译内核。

接着注册 `yat` 调度类，这里可以直接模仿其他调度类的注册方式，具体步骤如下：

在 `include/linux/sched` 目录下创建 `yat.h` 文件，写入如下内容：

```

#ifndef _LINUX_SCHED_YAT_H
#define _LINUX_SCHED_YAT_H
#ifdef CONFIG_SCHED_CLASS_YAT
#include <linux/llist.h>
struct yat_dispatch_q {
};
struct sched_yat_entity {
    u64    slice;
};
#else /* !CONFIG_SCHED_CLASS_YAT */

```

```
#endif /* CONFIG_SCHED_CLASS_YAT */
#endif /* _LINUX_SCHED_YAT_H */
```

在 *include/asm-generic/vmlinux.lds.h* 的 *SCHED_DATA* 宏添加 *(__yat_sched_class)*

```
#define SCHED_DATA \
    STRUCT_ALIGN(); \
    __sched_class_highest = .; \
    *(__stop_sched_class) \
    *(__dl_sched_class) \
    *(__rt_sched_class) \
    *(__fair_sched_class) \
    *(__yat_sched_class) \
    *(__idle_sched_class) \
    __sched_class_lowest = .;
```

在 *include/uapi/linux/sched.h* 的 *Scheduling policies* 处添加 *#define SCHED_YAT 8*

```
#define SCHED_NORMAL    0
#define SCHED_FIFO      1
#define SCHED_RR        2
#define SCHED_BATCH     3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE      5
#define SCHED_DEADLINE  6

#ifdef CONFIG_SCHED_CLASS_YAT
#define SCHED_YAT        8
#endif
```

在 *include/linux/sched.h* 的调度实体结构体那添加 *struct sched_yat_entity yat;*

```
struct sched_entity se;
struct sched_rt_entity rt;
struct sched_dl_entity dl;
#ifdef CONFIG_SCHED_CLASS_YAT
struct sched_yat_entity yat;
#endif
struct sched_dl_entity *dl_server;
const struct sched_class *sched_class;
```

在 *init/init_task.c* 添加

```
#ifdef CONFIG_SCHED_CLASS_YAT
.yat = {
    .slice = 0,
},
#endif
```

修改 *kernel/sched/core.c* 里的 *__setscheduler_prio* 函数, 并在 *for_each_possible_cpu* 函数里初始化 *yat* 的任务队列:

```
static void __setscheduler_prio(struct task_struct *p, int prio)
{
    if (dl_prio(prio)){
```

```

    p->sched_class = &dl_sched_class;
}
#ifdef CONFIG_SCHED_CLASS_YAT
    else if (yat_prio(p)){
        p->sched_class = &yat_sched_class;
        printk("\n\n=====select yat yat yat yat=====\n\n");
    }
#endif
    else if (rt_prio(prio)){
        p->sched_class = &rt_sched_class;
    }
    else
        p->sched_class = &fair_sched_class;

    p->prio = prio;
}
...
    init_rt_rq(&rq->rt);
    init_dl_rq(&rq->dl);
#ifdef CONFIG_SCHED_CLASS_YAT
    printk("=====init yat rq=====\n");
    init_yat_rq(&rq->yat);
#endif

```

在 `kernel/sched/sched.h` 里写入 `yat` 调度策略, 并使其成为有效的调度策略:

```

#ifdef CONFIG_SCHED_CLASS_YAT
static inline int yat_policy(int policy)
{
    return policy == SCHED_YAT;
}
#endif

static inline bool valid_policy(int policy)
{
#ifdef CONFIG_SCHED_CLASS_YAT
    return idle_policy(policy) || fair_policy(policy) ||
        rt_policy(policy) || dl_policy(policy) || yat_policy(policy);
#else
    return idle_policy(policy) || fair_policy(policy) ||
        rt_policy(policy) || dl_policy(policy);
#endif
}

```

并定义 `yat` 任务队列 `yat_rq`:

```

#ifdef CONFIG_SCHED_CLASS_YAT
struct yat_rq {
    struct task_struct *agent; /* protected by e->lock and rq->lock */

```

```
};
#endif /* CONFIG_SCHED_CLASS_YAT */
```

创建 *kernel/sched/yat.h* 和 *yat.c*，建立一系列简单的接口：

```
extern const struct sched_class yat_sched_class;
bool yat_prio(struct task_struct *p);
void enqueue_task_yat(struct rq *rq, struct task_struct *p, int flags);
void dequeue_task_yat(struct rq *rq, struct task_struct *p, int flags);
void yield_task_yat(struct rq *rq);
bool yield_to_task_yat(struct rq *rq, struct task_struct *p);
void wakeup_preempt_yat(struct rq *rq, struct task_struct *p, int flags);
struct task_struct *pick_next_task_yat(struct rq *rq);
void put_prev_task_yat(struct rq *rq, struct task_struct *p);
void set_next_task_yat(struct rq *rq, struct task_struct *p, bool first);
int balance_yat(struct rq *rq, struct task_struct *prev, struct rq_flags *rf);
int select_task_rq_yat(struct task_struct *p, int task_cpu, int flags);
void set_cpus_allowed_yat(struct task_struct *p, struct affinity_context *ctx);
void rq_online_yat(struct rq *rq);
void rq_offline_yat(struct rq *rq);
struct task_struct *pick_task_yat(struct rq *rq);
void task_tick_yat(struct rq *rq, struct task_struct *p, int queued);
void switched_to_yat(struct rq *this_rq, struct task_struct *task);
void prio_changed_yat(struct rq *this_rq, struct task_struct *task, int oldprio);
void update_curr_yat(struct rq *rq);
```

其中 *kernel/sched/yat.c* 的末尾需要定义 *yat* 调度类的一些继承属性：

```
DEFINE_SCHED_CLASS(yat) = {
    .enqueue_task    = enqueue_task_yat,
    .dequeue_task    = dequeue_task_yat,
    .yield_task      = yield_task_yat,
    .yield_to_task   = yield_to_task_yat,
    .wakeup_preempt  = wakeup_preempt_yat,
    .pick_next_task  = pick_next_task_yat,
    .put_prev_task   = put_prev_task_yat,
    .set_next_task   = set_next_task_yat,
#ifdef CONFIG_SMP
    .balance         = balance_yat,
    .select_task_rq  = select_task_rq_yat,
    .set_cpus_allowed = set_cpus_allowed_yat,
    .rq_online       = rq_online_yat,
    .rq_offline      = rq_offline_yat,
#endif
#ifdef CONFIG_SCHED_CORE
    .pick_task       = pick_task_yat,
#endif
    .task_tick       = task_tick_yat,
    .switched_to     = switched_to_yat,
    .prio_changed    = prio_changed_yat,
```



```
.update_curr    = update_curr_yat,  
#ifdef CONFIG_UCLAMP_TASK  
.uclamp_enabled = 0,  
#endif  
};
```

至此我们实现了一个空调度器，也即实现了调度的初始化

4.3 Yat-Scheduler 基础模块设计和实现

如前面图 3.1 所示，Yat-Scheduler 将分为五个模块：底层数据模块、调度跟踪模块、调度算法模块、同步处理模块、系统调用模块，每个模块处理任务调度过程的不同部分。

4.3.1 底层数据结构模块

根据数据结构作用的对象不同，底层数据结构模块可以分为 3 部分：

(1) 作用于就绪队列、释放队列及队列节点的数据结构——结构体 *bheap_node* 和 *bheap*

结构体 *bheap* 是运行队列和释放队列的主要数据结构，保存了两者的头结点指针以及队列中优先级最高的节点指针。结构体 *bheap* 的操作和实现使用 *Binomial Heap* 数据结构实现的。结构体 *bheap_node* 是队列中节点的数据结构，包含有父母节点指针、兄弟节点指针、孩子节点指针、高度值、值、自身的引用指针。

在结构体 *bheap* 中节点优先级的判断是根据调度算法模块中优先级比较函数，优先级最大的节点一般会放在队列的头结点中，并用结构体 *bheap* 中优先级最高的节点指针表示。

(2) 作用于调度算法的数据结构——结构体 *rt_domain_t*

结构体 *rt_domain_t* 作用是保存调度的就绪队列，各个任务的释放队列，任务就绪锁，任务释放锁，以及保存着当前调度算法下优先级比较函数指针、作业释放函数指针、检查任务是否重新调度函数指针。在调度算法模块工作过程中，通过结构体 *rt_domain_t*，调度算法模块可以获得和操作底层数据结构。

(3) 作用于任务 *task* 的数据结构——结构体 *rt_param*

在 Linux 的任务机制中，每个任务都会有一个 *task_struct* 数据结构，用以保存任务存在期间的所有信息，包括分配的内存、优先级、在哪个 *cpu* 上调度等。Yat-Scheduler 向该数据结构中添加 *rt_param* 成员，用以记录在 Yat-Scheduler 下运行的实时任务的各种信息。通过成员 *rt_param*，任务 *task* 可以操作任务和其作业的基本信息及调度信息，在哪个 *cpu* 上调度及在哪个 *cpu* 上链接，作用在哪个 *rt_domain_t* 上，以及任务的 *bheap_node* 和释放队列等等。

图 4.4 说明了底层数据模块的结构。

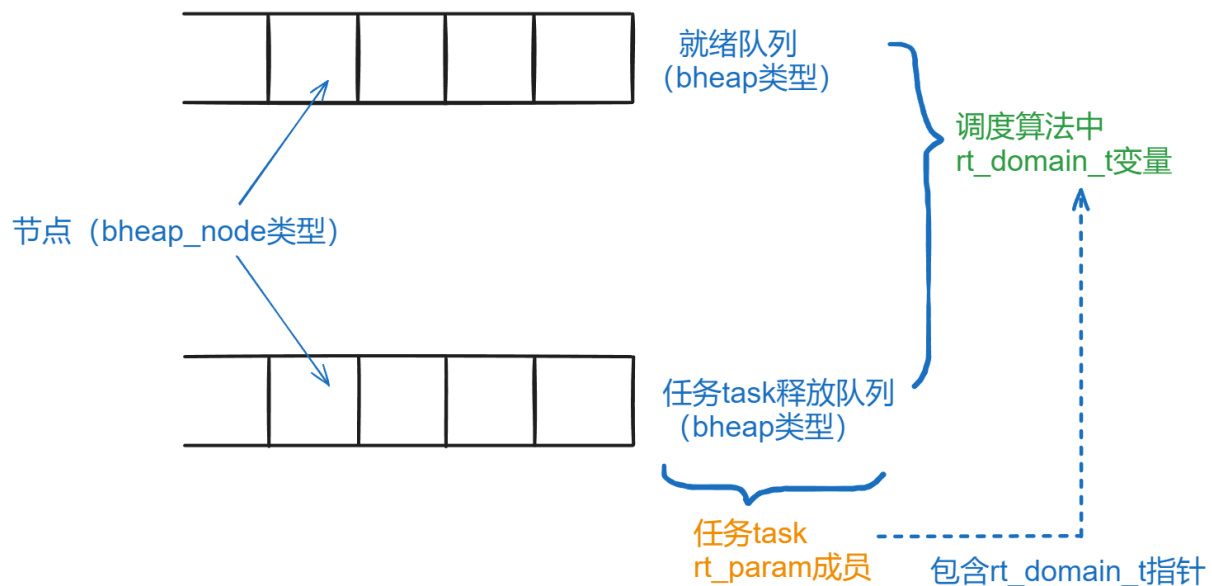


图 4.4 底层数据模块结构

4.3.2 调度算法模块

调度算法模块的框架图如 图 4.5 表示，分为四个部分：模块初始化、调度、调度触发、任务处理。

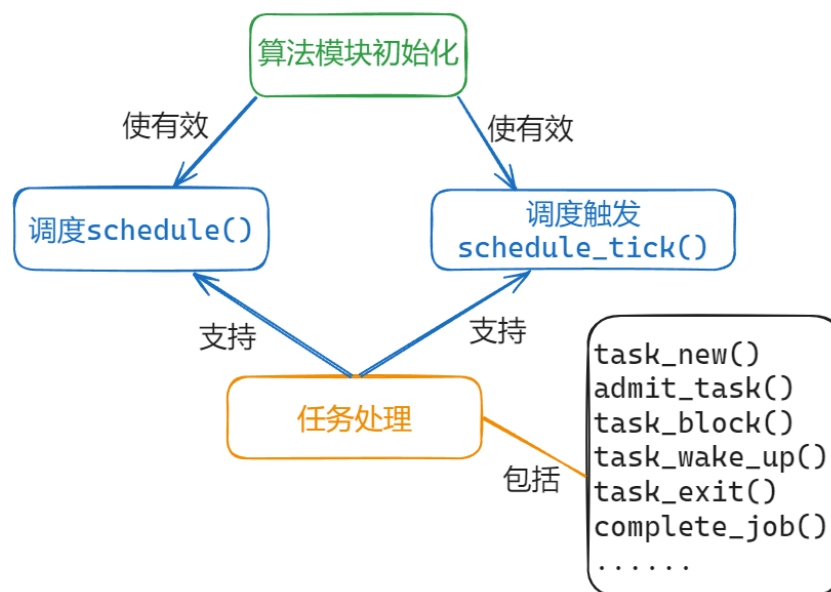


图 4.5 调度算法模块结构

在介绍调度模块之前，先介绍几个有关调度算法的知识，都与调度算法模块相关：

(1) 调度算法如何获取最高优先级任务

在不考虑抢占情况，根据调度算法，CPU 调度的任务一般是就绪队列中最高优先级任务。在 Yat-Scheduler 各个调度算法中，就绪队列均为 *bheap* 类型，由于 *Binomial Heap* 的结构特点及其操作，其头结点一直是整个堆的最值。在节点排序过程中，通过使用调度算法的优先级比较函数，就绪队列的头结点就是优先级最大的任务节点，具体实现是

在新的任务节点加入到就绪队列时，需要从头结点向下遍历每个节点与新的任务节点比较优先级大小，直到找到合适的位置。

(2) Linux 下的调度模型

在 Linux 系统中，调度模型主要是由 `schedule()` 和 `schedule_tick()` 两个函数控制的。任务被初始化并放到运行队列后，在某个时刻，任务应该获得对 CPU 的访问。负责把 CPU 的控制权传递到不同任务的两个函数是 `schedule()` 和 `schedule_tick()`。
`scheduler_tick()` 是一个由内核周期性调用的系统定时器，它把某一个任务标记为需重新调度 (`WILL_SCHEDULE`)。定时事件发生时，当前的任务就被保存起来，Linux 内核接管对 CPU 的控制。定时事件完成后，Linux 内核通常把控制权传回被保存的进程。然而，当所保存的任务被标记为需重新调度时，内核调用 `schedule()` 来选择需要激活哪个任务，并不一定选择内核接管控制前正在执行的那个任务。在内核接管控制前处于执行状态 (`RUNNING`) 的任务称为当前进程 (*current process*)。为了使情况不太复杂，在某些情况下，内核可以从内核获得控制权，称为内核抢占 (*kernel preemption*)。

(3) Yat-Scheduler 如何实现从 Linux 内核中的 `schedule()` 转换到我们实现的调度算法中的 `schedule()`

在 `schedule_tick()` 函数调用过程中，Yat-Scheduler 添加了 `yat_tick()` 函数，用来执行 Yat-Scheduler 相关调度算法的 `tick` 函数，然后把某一个进程设置为 `WILL_SCHEDULE`，同时判断是否重新调度。

在 Linux 系统调用 `schedule()` 函数过程中，会调用函数 `pick_next_task()`，该函数功能是根据进程的不同调度算法标志，调用不同的调度算法的 `pick_next_task()`，挑选出符合调度算法的下一个调度任务。而在 Yat-Scheduler 中，函数 `pick_next_task()` 直接调用 Yat-Scheduler 的调度算法的函数，而不考虑其他的调度算法。因此，在 Yat-Scheduler 中任务的调用只使用 Yat-Scheduler 的调度算法。然后，调用不同算法下的 `schedule()` 函数，选择下一个调度任务。

在 Yat-Scheduler 系统中，调度算法标志位在 表 4.1 中说明。

调度算法标志位	含义
<code>SCHED_NORMAL</code>	正常调度算法
<code>SCHED_FIFO</code>	先来先服务算法（实时任务）
<code>SCHED_RR</code>	时间片轮转换算法（实时任务）
<code>SCHED_BATCH</code>	批处理算法
<code>SCHED_IDLE</code>	处理空任务算法
<code>SCHED_YAT</code>	跳转到 Yat-Scheduler 去执行我们实现的调度算法

表 4.1 调度算法标志位

Yat-Scheduler 中从 Linux 内核中的 `schedule()` 转换到调度算法模块中 `schedule()` 的流程图见 图 4.6。

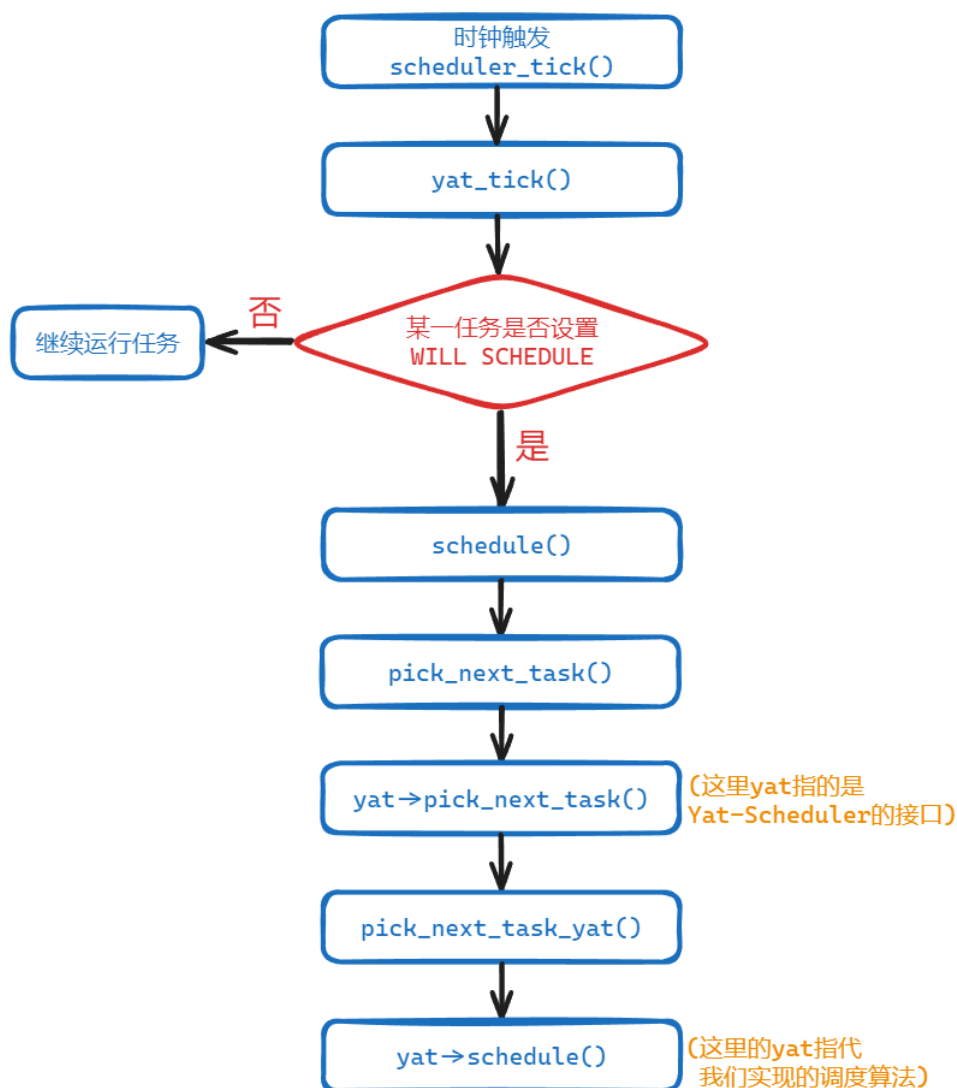


图 4.6 让内核跳转到调用我们实现的调度算法的流程

(4) 关键部分 (*critical section*) 的处理及优先级反转的处理

在调度算法处理调度的过程中，还需要解决关于关键部分获取的问题。关键部分又称为共享资源，其特点是在 t 时刻只允许一个任务读写其内容，如果多个任务同时读或写其内容，容易发生死锁，产生错误。为了控制对关键部分获取，Yat-Scheduler 内的调度算法将使用 FMLP [3] (a Flexible real-time Locking Protocol for Multiprocessors，一种在多核条件下灵活的实时锁协议，FMLP 介绍见 小节 4.4.1 多核互斥资源共享协议的介绍)，把调用关键部分的任务设置为非抢占 (*non-preemptable*)，即在任务占用这类资源时，不允许别的高优先级任务抢占该任务，直到该任务运行完毕或主动放弃这个关键部分。但是容易出现一个问题—优先级翻转：高优先级任务等待低优先级任务执行，这可能会导致调度结果分析出错。为了解决这个问题，FMLP 使用了优先级继承 (*priority inheritance*) 的概念，即当出现优先级翻转情况时，占用关键部分的任务会继承当前因为关键部分被抢占而等待的任务中最高的优先级的任务，用继承的任务来进行优先级的比较，间接地非抢占任务的低优先级变为高优先级，从而避免出现优先级翻转。在 Yat-Scheduler 中，任务有两种非抢占状态，一种是用户态非抢占状态 (*user_np*)，一种是内核态非抢占状态 (*kernel_np*)。

(5) 全局性调度算法中任务的两个状态

由于需要支持任务迁移和抢占, 对于 CPU, 当前调用的任务与通过 *Linux* 内核下默认的 `schedule()` 选取的任务可能不一样, 对于任务, 当前可能被安排到某个 CPU 上调度, 但是下一时刻, 可能迁移到另一个 CPU 上调度。为了区分理论调度任务和实际调度任务的不同, 任务和 CPU 分别设置两个状态: 链接状态 (*linked*) 和调度状态 (*scheduled*)。这两种状态只适用于全局性调度算法。

对于任务来说, 两个状态都是 *int* 类型, 分别表示链接和调度安排的 CPU 号。链接状态 (*linked_on*) 是指该任务应该在哪个 CPU 执行, 任务是否链接与任务的优先级有关; 调度状态 (*schedule_on*) 是指该任务实际上在哪个 CPU 执行, 与实际调度情况相关。两个状态是独立的, 互不影响。因此两个状态构成了任务的四种情况:

- 1) **未链接, 未调度**: 说明任务没有执行
- 2) **链接, 未调度**: 说明任务为当前就绪队列中优先级最高的任务, 但是发生了优先级继承, 当前执行任务控制着共享资源
- 3) **未链接, 调度**: 说明当调度算法选择下一个调度任务时, 当前执行的任务的优先级不是当前情况最高的, 但是由于该任务控制着共享资源, 所以发生了优先级继承, 该任务的优先级称为当前优先级最高的任务
- 4) **链接, 调度**: 说明该任务是当前就绪队列中优先级最高的任务, 同时没有发生抢占

对于 CPU (*cpu_entry_t* 类型) 来说, 两个状态都是任务 *task_struct* 类型, 链接状态是指通过 `schedule()` 函数选取的理论调度的任务, 调度状态是指实际上调度的任务。

4.4 多核互斥资源共享协议设计和实现

4.4.1 多核互斥资源共享协议介绍

多核互斥资源共享协议主要实现在了同步处理模块中。同步处理模块实现了两种同步处理机制: *SRP* [4] (*Stack Resource Policy*, 以栈为基础的资源分配协议) 和 *FMLP* [3] (*Flexible Multiprocessor Locking Protocol*, 灵活多核锁协议)。

SRP 协议是以抢占上限 (*preemptive ceiling*) 为基础的协议。在 *SRP* 中, 每个资源都有一个整型值的上限, 该上限是动态变化的, 与当前调度的任务的抢占级别 (*preemptive level*) 和优先级有关。在所有资源的上限中, 确定上限的最值。如果一个任务的抢占级别不大于上限的最值, 那么任务挂起, 否则任务执行。

在 *FMLP* 协议中, 资源被分为两种: 长型 (*long*) 和短型 (*short*)。资源长、短型的确定是由用户确定的。资源组 (*Resource groups*) 是 *FMLP* 协议中的基本单元。每个资源组要么只包含长型资源, 要么只包含短型资源, 并且由一个 *group lock* 保护。对于长型资源组, 组锁为信号量机制; 对于短型资源组, 组锁为非抢占的队列锁机制。

一个任务发出对短资源 *SR* 的请求, 如果 *SR* 空闲, 任务获得了包含 *SR* 的短资源组的组锁—队列锁。如果有其他任务请求 *SR*, 它们处于挂起状态, 以 *FIFO* 的顺序处于忙等待。在试图获得短资源组的组锁时, 任务必须首先是非抢占的, 直到任务释放组锁。

一个任务发出对长资源 *LR* 的请求，如果 *LR* 空闲，任务获得包含 *LR* 的长资源组的组锁—信号锁。如果有其他任务请求 *LR*，它们处于挂起状态，并被添加到一个 *FIFO* 队列中。当任务获得 *LR* 的组锁时，任务将继承因该长资源组而挂起的任务的优先级的最大值，而且可以抢占式的调度。

4.4.2 多核互斥资源共享协议设计

在 Yat-Scheduler 中，抢占级别为 *srp_priority* 类型变量，将实现一个 *srp_semaphore* 类型的变量，维护上限的最值。抢占级别与上限最值的比较函数是 *int srp_higher_prio(struct srp_priority* first, struct srp_priority* second)*，比较两者相对应任务的周期的长短。在抢占级别与上限最值比较之前，获取当前调度的任务，然后把任务的周期赋给任务的抢占级别变量。如果任务的抢占级别的周期比上限最值的周期长，说明任务可以运行，同时更新上限最值的周期。

为了实现 *FMLP*, Yat-Scheduler 将实现一个 *pi_semaphore* 类型的变量，包含一个表示最高级优先级的变量 *hp*，而 *hp* 包含一个 *task_struct* 类型的指针和一个 *task_struct* 指针类型的数组，分别存放当前调度过程中最高优先级的任务指针和每个 *CPU* 上优先级最高的任务指针。在等待队列中，判断 *hp* 存放的指针是否是最高优先级的指针，不是，则更新指针。

4.4.3 多核互斥资源共享协议实现

（该部分并未完全完成，待全国赛决赛一阶段，我们将全面完善）

先给出 Yat-Scheduler 中锁的定义：

```
struct yat_lock_ops;

struct yat_lock {
    struct yat_lock_ops *ops;
    int type;
};

struct yat_lock_ops {
    int (*open)(struct yat_lock*, void* __user);
    int (*close)(struct yat_lock*);

    int (*lock)(struct yat_lock*);
    int (*unlock)(struct yat_lock*);

    void (*deallocate)(struct yat_lock*);
};
```

其中 *yat_lock* 是用户空间信号量的通用基础结构，这种结构需要嵌入到协议特定的信号量中。*open* 和 *close* 表示当前任务尝试获取/删除对锁的引用。*lock* 和 *unlock* 表示尝试对任务的锁进行加锁或解锁。*deallocate* 表示锁不再被引用。

4.5 面向资源的优先级排序算法设计和实现

4.5.1 面向资源的优先级排序算法介绍

目前已有的优先级排序算法的简单介绍：首先介绍 3 个主流的任务优先级排序算法——DMPO, OPA 和 RPA

1. DMPO 算法

DMPO 算法为 *deadline* 越短的任务分配更高的优先级。即当 $Deadline_x < Deadline_y$ 则 $Priority_x > Priority_y$ ，通过这个规则实现对任务的优先级排序。

2. OPA 算法

给定一组未分配优先级的任务，该算法从最低优先级开始，检查是否存在可在该优先级上调度的未分配优先级的任务(此时假设所有其他未分配的任务具有更高的优先级)。如果找到这样的任务，则为其分配此优先级。然后算法进入下一个优先级并检查其余未分配的任务。如果为每个任务分配了优先级，则该算法将返回可调度的解决方案。OPA 是一种最佳优先级分配，因为它保证可以找到可调度的优先级排序(如果有)，最坏情况为 $n(n+1)/2$ 次迭代。此外，OPA 被证明比 DMPO 对于更广泛的应用程序语义来说是最佳的，例如具有偏移发布时间的系统和不确定 *deadline* 等情况。

OPA 算法伪代码：

```

Procedure OptimalPriorityAssignment(tasks)
    Initialize priority to 1
    Initialize an empty list assigned_tasks

    While tasks are not empty
        current_priority = priority
        CanSchedule = false

        For each task in tasks
            If task can be scheduled at current_priority
                // Assume all other unassigned tasks have higher priority
                assigned_tasks.append(task)
                Remove task from tasks
                CanSchedule = true
                Break // Exit the loop if a task is scheduled at this priority

        if CanSchedule == false
            Break // No task can be scheduled, exit the loop

        priority = priority + 1 // Move to the next priority level

    If assigned_tasks contains all tasks
        Return assigned_tasks // All tasks are assigned and schedulable
    Else
        Return "No schedulable solution found" // Not all tasks could be scheduled
EndProcedure

```

3. RPA 算法 [5]

在 OPA 算法执行的时候, 如果给定优先级存在多个可调度任务, 则任务会被分配任意优先级。这可能会导致系统只是可调度的, 但容易受到任务参数微小变化、执行预算超支或低估干扰的影响。为了解决这个问题, RPA 被开发为 OPA 的扩展, 并采用一种方法来指定应在每个优先级分配的确切任务。在 RPA 中, 引入了干扰函数来对任务在每个优先级上可能产生的潜在干扰量进行建模, 同时保持可调度性。通过此功能, RPA 旨在产生可以容忍最大量额外干扰的优先级排序。与 OPA 类似, RPA 从最低优先级开始, 需要 $n(n+1)/2$ 次二分搜索来找到所有优先级的最大附加干扰。在给定的优先级上, 可以容忍最大量额外干扰的任务将被分配该优先级。然后该算法迭代到下一个优先级, 直到所有任务都被分配了优先级。如果能够找到可行的优先级排序, 则保证该系统是可调度的, 并且能够应对所有优先级中可容忍的最小附加干扰

RPA 算法的伪代码:

```

Procedure OptimalPriorityAssignment(tasks)
    Initialize priority to 1
    Initialize an empty list assigned_tasks

    While tasks are not empty

        For each task in tasks
            binary search for the largest value of  $\alpha$  for which task  $\tau$  is
            schedulable at priority  $i$ 

            if no tasks are schedulable at priority  $i$ 
                return unschedulable
            else
                assign the schedulable task that tolerates the max  $\alpha$  at priority  $i$ 
                to priority  $i$ 

            priority = priority + 1 // Move to the next priority level

    If assigned_tasks contains all tasks
        Return assigned_tasks // All tasks are assigned and schedulable
    Else
        Return "No schedulable solution found" // Not all tasks could be scheduled
EndProcedure

```

本项目最终实现的优先级排序算法

上述三个算法在 *holistic schedule* 的情况下会出现问题。

在“面向资源的优先级排序算法设计”这一部分中我们会用具体的例子论证: 运行 *holistic schedulability tests* 时, 在 *fully partitioned* 的多处理器约束截止系统中, 共享资源由 MSRP 或 MrsP 管理时 DMPO 不是 *optimal* 的。

同时我们发现在执行 *holistic schedule* 的时候, 如果不知道本地高优先级任务和共享相同资源的远程任务的响应时间, 就不可能获得给定任务的准确响应时间。这会导致 *OPA* 和 *RPA* 算法出现问题。

原因: *OPA* 和 *RPA* 算法在确定某个 *task* 是否可调度的时候, 需要计算出这个任务的确切响应时间 (R), 但是这在 *holistic schedule* 下做不到。任何一个任务的响应时间的计算都需要知道其他相关任务的响应时间。*OPA* 类的算法试图获得固定的响应时间的时候假设所有未检查的任务 (即尚未分配优先级的任务) 具有更高的优先级, 但不假设这些任务有任何确切的优先级顺序。然而, 对于 *holistic tests*, 系统中任务的响应时间必须迭代计算, 直到所有任务的响应时间都固定下来, 并具有明确的任务优先级 (假设系统是可调度的)。而对于最初的 *OPA* 和 *RPA* 算法, 在 *holistic schedulability tests* 下, 因为每个任务无法知道本地高优先级任务和共享相同资源的远程任务的响应时间, 所以它们不可能获得给定任务的确切响应时间。这会导致优先级排序算法在执行过程中出现问题。

一种解决方法是直接令响应时间=Deadline 时间, 即 $R_x = D_x$, 下文中我们将这种方法称为 *OPA-D* 和 *RPA-D* 算法。然而, 这种方法的问题是, 通过用 D_x 替换 R_x , 在可调度系统中, 由于 $R_x \leq D_x$, 对可调度性结果会产生相当大的 *pessimism*, 这可能会导致实际上可行的系统无法调度。此外, 由于必须对可调度性测试做出妥协, 这些优先级排序算法的最优性也可能受到损害。

基于以上考虑本项目实现了一种新的优先级排序算法, 即基于 **Slack** 的优先级排序 (**SPO**), *SPO* 与 *OPA* 类算法有着类似的理念, 后者检查每个优先级 (从最低优先级开始), 并从所有未检查的任务 (即未分配优先级的任务) 中为任务分配优先级。然而, 与 *OPA* 类算法不同, *SPO* 引入了响应时间近似方法, 以尽量减少由于在整体可调度性测试中使用而产生的 *pessimism* 影响。本质上, *SPO* 考虑了响应时间依赖性, 旨在尽量减少具有这种依赖性的响应时间计算中的 *pessimism* 影响。

与 *OPA-D* 和 *RPA-D* 在计算给定任务 τ_i 的 R_i 时简单地假设其他所有任务的 $R=D$ 相比, *SPO* 在迭代过程中逐渐取代了对达到响应时间固定点的任务的悲观上限, 这将在下文中详细解释。因此, *SPO* 通常比 *OPA-D* 和 *RPA-D* 得到更准确的响应时间。此外, 在计算 R_i 时, *SPO* 以整体方式计算 τ_i 处理器上所有任务的响应时间 (将会在设计和实现部分详细介绍)。这种方法完全避免了 τ_i 与其局部高优先级任务之间的响应时间依赖性导致的 *pessimism* 影响。相比之下, 在 *OPA-D* 和 *RPA-D* 中, 这种 *pessimism* 存在于整个优先级分配过程中。

4.5.2 面向资源的优先级排序算法设计

1. 响应时间建模

A. 在 *Traditional Schedulability Tests* 下的模型

在 *MSRP* 和 *MrsP* 协议下, 给定任务 τ_i 的响应时间 R_i 由 [公式 1](#) 计算。 \bar{C}_i 是 τ_i 的总计算时间 (包括 τ_i (旋转) 等待和执行每个请求资源的时间)。 B_i 表示 τ_i 到达时可能产生的阻塞量。函数 $lhp(i)$ 返回一组优先级高于 τ_i 优先级的本地任务。此外, T_h 是 τ_h 的周期, C_h 是 τ_h 的总计算时间 (与 C_i 的原理相同)。

$$R_i = \overline{C}_i + B_i + \sum_{\tau_h \in \text{lhP}(i)} \left\lceil \frac{R_i}{T_h} \right\rceil \overline{C}_h \quad (1)$$

符号 \overline{C}_i 由 公式 2 确定。函数 $\sum_{r^k \in F(\tau_i)} N_i^k e^k$ 表示 τ_i 在一次发布中等待和执行每个请求资源（通过 $F(\tau_i)$ 算出）所花费的总时间，其中 N_i^k 是 τ_i 在一次发布中请求资源 r_k 的次数， e^k 是 r_k 的最坏情况访问时间。

$$\overline{C}_i = C_i + \sum_{r^k \in F(\tau_i)} N_i^k e^k \quad (2)$$

由于 *MSRP* 下对资源的请求是按照非抢占式 *FIFO* 顺序提供的，因此 e_k 实际上受包含对 r_k 的请求的处理器数量的限制，如 公式 3 所示，其中 $G(r_k)$ 表示需要 r_k 的任务集， c_k 表示使用 r_k 执行的最坏情况成本，函数 $\text{map}()$ 返回分配了给定任务的处理器集合， $|\cdot|$ 返回给定集合的大小。对于 *MrsP*，虽然它采用了抢占式资源访问方式，但在最坏情况下，一项任务会先代表 *FIFO* 队列中的所有其他任务执行，然后才能使用该资源（有关 *MrsP* 的帮助机制，请参阅 [6]）。因此，*MrsP* 任务可能产生的最坏情况阻塞时间也由 公式 3 计算，如 [6] 中所述。

$$e^k = |\text{map}(G(r^k))| * c^k \quad (3)$$

τ_i 的到达阻塞 (B_i) 通过 公式 4 和 公式 5 计算，其中 \hat{e}_i 是 τ_i 可能因远程延迟而产生的到达阻塞， \hat{b} 是底层操作系统中的最大非抢占部分， $F^A(\tau_i)$ 返回可能导致 τ_i 在到达时产生阻塞的资源集。

$$B_i = \max\{\hat{e}_i, \hat{b}\} \quad (4)$$

$$\hat{e}_i = \max\{e^k \mid r^k \in F^A(\tau_i)\} \quad (5)$$

然而，由于 *MSRP* 和 *MrsP* 使用不同的优先级来访问全局共享资源（即非抢占和优先级上限），因此在这些协议下计算此类资源的方法是不同的。

对于 *MSRP*，如果全局资源 r_k 被 τ_i 的本地低优先级任务（记为 τ_u ）请求，则 r_k 可能导致 τ_i 发生到达阻塞。对于本地资源，*PCP* [7] 被强制执行，因此如果资源的优先级上限高于 $\text{pri}(\tau_i)$ 且被 τ_u 请求，则该资源可能导致此类阻塞。公式 6 给出了在 *MSRP* 下可能导致到达阻塞的资源集（记为 $F_{\dagger}^A(\tau_i)$ ），其中 N_u^k 给出一次发布中从 τ_u 到 r_k 的请求数。

$$F_{\dagger}^A(\tau_i) = \{r^k \mid N_u^k > 0 \wedge (r^k \text{ is global} \vee \text{pri}(r^k) \geq \text{pri}(\tau_i))\} \quad (6)$$

对于 *MrsP* 来说，由于本地和全局资源都以上限优先级进行访问，因此 $F_{\dagger}^A(\tau_i)$ 的限制很简单，只需找到在 $P(\tau_i)$ （即 τ_i 的 *host processor*）上具有比 $\text{pri}(\tau_i)$ 更高的上限且由 τ_u 请求的资源即可，如 公式 7 所示。

$$F_{\dagger}^A(\tau_i) = \{r^k \mid N_u^k > 0 \wedge \text{pri}(r^k, P(\tau_i)) \geq \text{pri}(\tau_i)\} \quad (7)$$

B. 在 *Holistic Schedulability Tests* 下的模型

尽管上述 *Traditional Schedulability Tests* 有诸多优点，但是这些测试依赖于这样的假设：每次一个任务（用 τ_i 表示）请求资源时， $\text{map}(G(r_k))$ 中每个远程处理器上总会有一个

远程请求可以阻塞 τ_i , 而不管在时间段 R_i 内可以发出的实际可能请求数量。此外, 如 [8] 所述, 这些测试依赖于阻塞时间相对于任务执行时间的膨胀, 这引入了额外的 *pessimism* 影响。随后, [9] 提出了一种基于整数线性规划的针对几种自旋锁协议 (包括 *MSRP*) 的改进可调度性测试。后来, [10] 重新格式化了基于 *ILP* 的分析以消除任何优化的需要, 并在 *MrsP* 的背景下提出了一种整体可调度性测试, 它分析了给定时间段内可以发出的资源请求总数。这种整体分析可以直接应用于 *MSRP* 和 *MrsP*, 并采用相应的 $F^A(\tau_i)$ 函数 (即 公式 6 和 公式 7)

在 [10] 的整体分析中, τ_i 的响应时间由 公式 8 限定, 其中 C_i 是 τ_i 的纯最坏情况计算时间 (即不访问任何共享资源), E_i 是 τ_i 的总资源访问时间, 其中考虑了潜在的自旋延迟和来自每个本地高优先级任务 τ_h (它抢占了 τ_i , 但因请求锁定的资源而被阻止) 导致的 τ_i 产生的间接自旋延迟 (即传递阻塞), B_i 是 τ_i 的到达阻塞。

$$R_i = C_i + E_i + B_i + \sum_{\tau_h \in \text{lph}(i)} \left\lceil \frac{R_i}{T_h} \right\rceil \cdot C_h \quad (8)$$

E_i 通过 公式 9 得出, 其中 ζ_i^k 表示 τ_i 的本地高优先级任务向 r_k 发出的总请求数, $\xi_{i,m}^k$ 表示远程处理器 P_m 向 r_k 发出的请求数。请注意, 此分析使用整体方法限制阻塞时间, 其中在 τ_i 发布期间由于 r_k 而导致 τ_i 从远程处理器 P_m 可能产生的最大阻塞由 $N_i^k + \zeta_i^k$ (R_i 持续时间内 τ_i 及其本地高优先级任务向 r_k 发出的总请求数) 和 $\xi_{i,m}^k$ 之间的最小值限制。通过这样做, 此分析不像传统测试那样 *pessimism*, 因为:

1) 它计算出了可能导致阻塞的**确切**远程请求数量 (即避免了 *Traditional Schedulability Tests* 中使用的假设)

2) 每个关键部分 (即资源请求) 只考虑一次 (避免增加任务的计算时间)

$$E_i = \sum_{r^k \in \mathbb{R}} (N_i^k + \zeta_i^k + \sum_{P_m \neq P(\tau_i)} \min\{N_i^k + \zeta_i^k, \xi_{i,m}^k\}) \times c^k \quad (9)$$

ζ_i^k 和 $\xi_{i,m}^k$ 分别由 公式 10 和 公式 11 得出。如方程所示, 它们包含一个抖动间隔 (即分别为 R_h 和 R_j), 以延长 τ_i 的释放持续时间, 从而提供安全的上限。如 [9] 中所述和证明的 (参见 [9] 中的引理 5.1), 在多处理器系统中, 在给定的持续时间 t 内, 最多可以释放 $\left\lceil \frac{t+R_x}{T_x} \right\rceil$ 作业。该引理构成了计算给定持续时间内发出的请求数的基本方法, 并被 [8], [10], [11] 中改进的可调度性测试所应用。

$$\zeta_i^k = \sum_{\tau_h \in \text{lph}(i)} \left\lceil \frac{R_i + R_h}{T_h} \right\rceil N_h^k \quad (10)$$

$$\xi_{i,m}^k = \sum_{\tau_j \in \Gamma_{P_m}} \left\lceil \frac{R_i + R_j}{T_j} \right\rceil N_j^k \quad (11)$$

τ_i 的到达阻塞 (B_i) (在公式 8 中) 通过 公式 4 计算, 但采用不同的边界方法, 如 公式 12 和 公式 13 中所示。

$$\hat{e}_i = \max\{|\alpha_i^k| \cdot c^k \mid r^k \in F^A(\tau_i)\} \quad (12)$$

其中 α_i^k 表示包含未计入 r_k 的请求（即未在 E_i 中考虑的请求）的处理器集，其计算方式如下：

$$\alpha_i^k \triangleq \{P_m \mid \xi_{i,m}^k - \zeta_i^k - N_i^k > 0 \wedge P_m \neq P(\tau_i)\} \cup P(\tau_i) \quad (13)$$

对于属于 $F^A(\tau_i)$ 的每个 r_k ，该方程确定包含未计入请求的远程处理器（即 $\xi_{i,m}^k - \zeta_i^k - N_i^k > 0$ ），其中每个处理器中对 r_k 的请求在到达时都会阻塞 τ_i 。因此，通过限制此类处理器的数量，[公式 12](#) 得出 τ_i 在 $F^A(\tau_i)$ 中的所有资源中可能产生的最大阻塞

2. 最优性(optimal)分析

协议	传统测试	<i>Holistic</i> 测试
<i>MSRP</i> (†)	S_{\dagger}	S_{\dagger}°
<i>MrsP</i> (‡)	S_{\ddagger}	S_{\ddagger}°

表 4.2 *MSRP* 和 *MrsP* 的可调度性测试

定义 1：优先级分配算法 Λ 在任务模型、调度算法 G 和可调度性测试 S 下是最优的，当且仅当符合任务模型的每一组任务在其他优先级分配下被 S 视为可使用 G 调度，也可使用 Λ 调度。

可以用数学归纳法证明 *DMPO* 在 *Traditional Schedulability Tests* 下是 *optimal* 的，具体可见 [\[12\]](#)

归纳基：假设优先级分配算法 Λ_x 在可调度性测试 S_{\dagger} 或 S_{\ddagger} 下可使用 M 个处理器对给定任务集 Γ 进行调度，其中 Λ_x^x 表示任务集 Γ 的可调度优先级顺序。

归纳步骤：在 Λ_{Γ}^x 下，选择一对不按 *DMPO* 顺序排列的相邻任务，交换它们的优先级以形成新的优先级排序，记为 Λ_{Γ}^{x-1} 。然后，证明没有任务因为这种优先级交换而错过截止时间。对于具有 n 个任务的任务集，最多需要 $x = n(n+1)/2$ 次优先级交换才能将一个处理器中的优先级排序从 Λ_{Γ}^x 转移到 *DMPO*（即 $\Lambda_{\Gamma}^1 = \text{DMPO}_{\Gamma}$ ）。如果在所有处理器的整个优先级重新排序过程中，没有任务分别在 S_{\dagger} 和 S_{\ddagger} 下错过截止时间，则将不存在可用 Λ_{Γ}^x 调度但不能用 *DMPO* 调度的任务集，从而证明了 *DMPO* 的最优性

但是在 *holistic schedulability test* 下：在 S_{\dagger}° （*MSRP*）或 S_{\ddagger}° （*MrsP*）下由 *MSRP* 或 *MrsP* 管理共享资源的 *fully partitioned* 多处理器约束截止系统中，*DMPO* 不是最优的。

证明：本证明通过反例进行。图 4.4 显示了具有两个共享资源 r^1 和 r^2 的三处理器系统。表 4.3 和 4.4 给出了处理器 P_1 中的任务属性和资源使用情况。在 τ_2 和 τ_3 之间执行优先级交换。在优先级排序 W^x 下， $\text{pri}(\tau_3) > \text{pri}(\tau_2) > \text{pri}(\tau_1)$ ，且 $D_3 > D_2$ ，即截止时间较长的任务被分配更高的优先级。在 *DMPO* 下， $\text{pri}(\tau_2) > \text{pri}(\tau_3) > \text{pri}(\tau_1)$ ，因此优先级按截止时间的相反顺序分配。此外，我们假设 P_0 和 P_2 都对 r^1 和 r^2 有足够的请求，因此从 P_1 访问 r^1 （或 r^2 ）的成本始终为 $3c^1$ （或 $3c^2$ ）

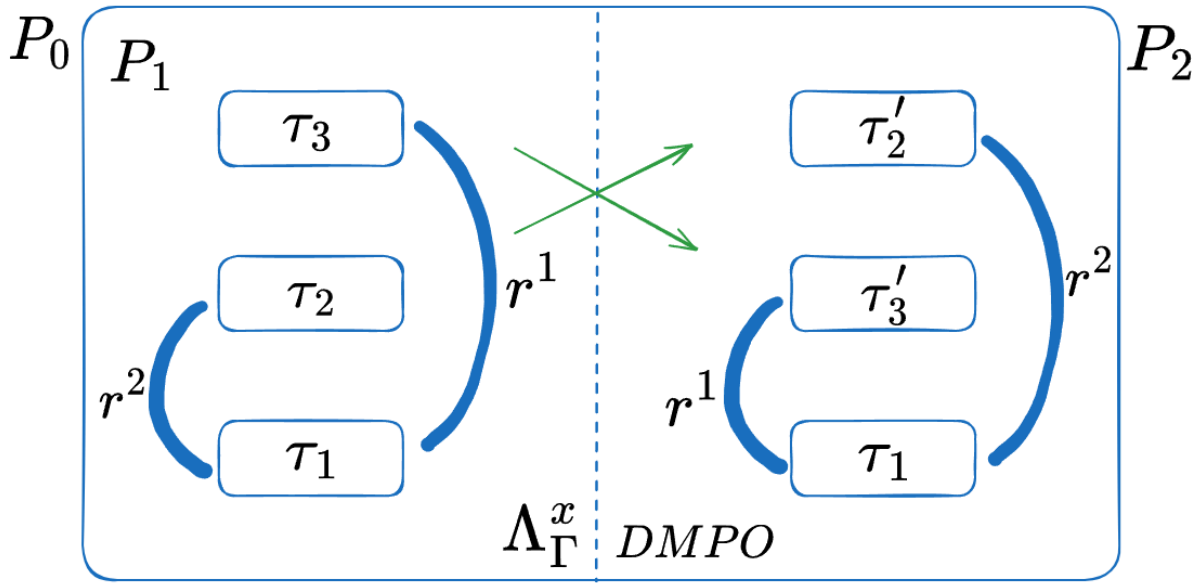


图 4.7 优先级交换的例子 1

$\mathbf{Task}(\tau_x)$	C_x	T_x	D_x
τ_2	1	17	17
τ_3	1	27	27

表 4.3 上图系统的 $P1$ 中任务属性

$\mathbf{Resource}(r^k)$	c^k	$G(r^k)$	N_x^k
r^1	1	$\{\tau_1, \tau_3\}$	$N_1^1 = 1, N_3^1 = 1$
r^2	2	$\{\tau_1, \tau_2\}$	$N_1^2 = 1, N_2^2 = 1$

表 4.4 上图系统的 $P1$ 中资源使用情况

在 $MSRP$ 下, 由于采用非抢占式资源访问方法, r^1 和 r^2 都可能导致 τ_2 和 τ_3 在两种优先级排序中出现到达阻塞, 因此在 Λ_Γ^x 和 $DMPO$ 中 $F^A(\tau_2) = F^A(\tau_3) = \{r^1, r^2\}$ 。优先级排序 Λ_Γ^x 下 τ_3 和 τ_2 的响应时间计算如下所示。

$$\begin{aligned}
R_3 &= C_3 + E_3 + B_3 \\
&= C_3 + 3c^1 + \max\{3c^1, 3c^2\} \\
&= 1 + 3 \times 1 + 3 \times 2 \\
&= 10 \\
R_2 &= C_2 + E_2 + B_2 + \left\lceil \frac{R_2}{T_3} \right\rceil C_3 \\
&= C_2 + 3c^2 + \max\{3c^1, 3c^2\} + \left\lceil \frac{R_2}{T_3} \right\rceil C_3 + \left\lceil \frac{R_2 + R_3}{T_3} \right\rceil 3c^1 \\
&= 1 + 3 \times 2 + 3 \times 2 + \left\lceil \frac{R_2}{27} \right\rceil \times 1 + \left\lceil \frac{R_2 + 10}{27} \right\rceil \times 3 \\
&= 17
\end{aligned}$$

从以上计算中，我们可以看出，在优先级交换之前，两个任务都能够满足截止期限。然而，在优先级交换之后， R'_3 却错过了截止期限，如下所示。

$$\begin{aligned}
 R'_2 &= C_2 + E_2 + B_2 \\
 &= C_2 + 3c^2 + \max\{3c^1, 3c^2\} \\
 &= 1 + 3 \times 2 + 3 \times 2 \\
 &= 13 \\
 R'_3 &= C_3 + E_3 + B_3 + \left\lceil \frac{R'_3}{T_2} \right\rceil C_2 \\
 &= C_3 + 3c^1 + \max\{3c^1, 3c^2\} + \left\lceil \frac{R'_3}{T_2} \right\rceil C_2 + \left\lceil \frac{R'_3 + 13}{17} \right\rceil 3c^2 \\
 &= 1 + 3 \times 1 + 3 \times 2 + \left\lceil \frac{R'_3}{17} \right\rceil \times 1 + \left\lceil \frac{R'_3 + 13}{17} \right\rceil \times (3 \times 2) \\
 &= 30
 \end{aligned}$$

使用 $MrsP$ 的整体测试（即 S_{\ddagger}° ），情况类似，但响应时间值不同。根据此协议（使用优先级上限），在 Λ_{Γ}^x 下 $F_{\ddagger}^A(\tau_3) = \{r_1\}$ 优先级交换后， $F_{\ddagger}^A(\tau_2) = \{r_2\}$ 。然而，这并没有改变交换前两个任务都是可调度的事实原来（ $R_3 = 7, R_2 = 17$ ），但 τ_3 在优先级降低后错过了截止时间（ $R'_2 = 13$ 和 $R'_3 = 30$ ）

基于上述反例，我们证明 $DMPO$ 在整体测试下并不是 *optimal* 的，因为方程 (10) 中存在响应时间依赖性，这是为了减少对膨胀任务执行时间的 *pessimism* 而引入的。

但是，上述计算都是假设每次访问 r_k 的成本为 $|map(G(r_k))|$ 。下面我们提供另一个例子，说明 $DMPO$ 的最优性也会由于远程任务的响应时间依赖性而受到破坏（在公式 (11) 中），该例子用于最小化在计算访问共享资源的成本时使用常数上限所带来的悲观情绪。图 4.5 展示了一个双处理器系统，其中有三个任务和一个由两个处理器共享的资源 (r^1)。表 4.5 和 4.6 给出了该系统的任务参数和资源使用情况。在 P_0 上对 τ_1 和 τ_2 进行优先级交换，如图 4.5 所示。在此示例中，优先级交换之前 $pri(\tau_1) > pri(\tau_2)$ ，而交换之后 $pri(\tau_2) > pri(\tau_1)$ 。

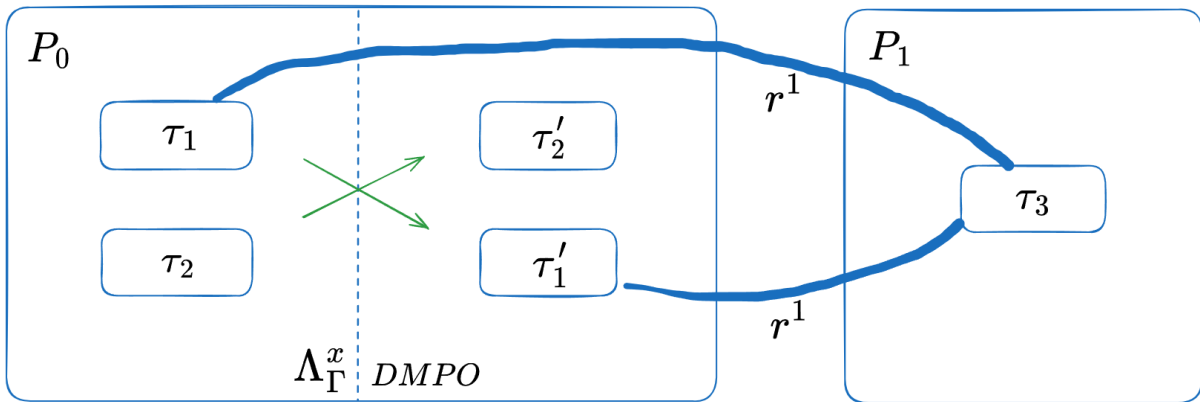


图 4.8 优先级交换的例子 2

$Task(\tau_x)$	P_x	C_x	T_x	D_x
----------------	-------	-------	-------	-------

τ_1	0	2	28	28
τ_2	0	5	20	20
τ_3	1	3	35	20

表 4.5 上图系统的任务属性

$\text{Resource}(r^k)$	c^k	$G(r^k)$	N_x^k
r^1	4	$\{\tau_1, \tau_3\}$	$N_1^1 = 1, N_3^1 = 3$

表 4.6 上图系统的资源使用情况

在 Λ_F^x 下, 无论应用 $MSRP$ 还是 $MrsP$, 任何任务都不会发生到达阻塞 (即, 所有任务的 $\hat{e} = 0$), 因为只有 τ_1 和 τ_3 请求 r^1 。因此, 在 S_{\dagger}° 和 S_{\ddagger}° 下, 所有三个任务的响应时间都相同, 并且所有任务都能够满足其截止期限, 如下所示

$$\begin{aligned}
R_1 &= C_1 + E_1 \\
&= C_1 + N_1^1 c^1 + \min\{N_1^1 c^1, \left\lceil \frac{R_1 + R_3}{T_3} \right\rceil N_3^1 c^1\} \\
&= 2 + 4 + \min\{4, \left\lceil \frac{R_1 + 18}{35} \right\rceil \times 12\} \\
&= 10 \\
R_2 &= C_2 + E_2 + \left\lceil \frac{R_2}{T_1} \right\rceil C_1 \\
&= C_2 + \min\left\{\left\lceil \frac{R_2 + R_1}{T_1} \right\rceil N_1^1 c^1, \left\lceil \frac{R_2 + R_3}{T_3} \right\rceil N_3^1 c^1\right\} \\
&\quad + \left\lceil \frac{R_2 + R_1}{T_1} \right\rceil N_1^1 c^1 + \left\lceil \frac{R_2}{T_1} \right\rceil C_1 \\
&= 5 + \min\left\{\left\lceil \frac{R_2 + 10}{28} \right\rceil 4, \left\lceil \frac{R_2 + 18}{35} \right\rceil 12\right\} + \left\lceil \frac{R_2 + 10}{28} \right\rceil 4 \\
&\quad + \left\lceil \frac{R_2}{28} \right\rceil \times 2 \\
&= 15 \quad ; \\
R_3 &= C_3 + E_3 \\
&= C_3 + N_3^1 c^1 + \min\{N_3^1 c^1, \left\lceil \frac{R_3 + R_1}{T_1} \right\rceil N_1^1 c^1\} \\
&= 2 + 12 + \min\{3 \times 4, \left\lceil \frac{R_3 + 10}{28} \right\rceil \times 1 \times 4\} \\
&= 18
\end{aligned}$$

然而, 在优先级交换 (即应用 $DMPO$) 之后, τ_3 会因为 τ_1 响应时间增加而错过其截止时间, 如下图 S_{\dagger}° 所示。

$$\begin{aligned}
R'_2 &= C_2 + B_1 \\
&= 5 + |\{P_0, P_1\}| \times 4 \\
&= 13 \\
R'_1 &= C_1 + E_1 + \left\lceil \frac{R'_1}{T_2} \right\rceil C_2 \\
&= 2 + 1 \times 4 + \min\{1 \times 4, \left\lceil \frac{R'_1 + 22}{35} \right\rceil \times 12\} + \left\lceil \frac{R'_1}{28} \right\rceil \times 5 \\
&= 15 \\
R'_3 &= C_3 + E_3 \\
&= 2 + 12 + \min\{12, \left\lceil \frac{R'_3 + 15}{28} \right\rceil \times 4\} \\
&= 22
\end{aligned}$$

在 S_+° 下, 唯一的区别是 τ_2 现在不会因资源上限设施而导致到达阻塞, 因此 $R'_2 = 5$ 。但是, 这不会影响 R'_1 和 R'_3 的值, 其中 τ_3 仍然错过截止时间。请注意, 在此示例中, $\left\lceil \frac{R_2 + R_1}{T_1} \right\rceil = \left\lceil \frac{R_2}{T_1} \right\rceil$, 因此, 在 R_2 期间不会因公式 (10) 中引入的抖动执行 τ_1 的额外作业。因此, 此示例中 DMPO 的非最优性仅由远程处理器的响应时间依赖性引起。

对于 OPA 和 RPA 算法

在 [5] 中, 对 OPA 和 RPA (包括它们的扩展) 形式化了三个应用条件:

1. 根据测试 S , 任务 τ_x 的可调度性可能取决于优先级高于 $pri(\tau_x)$ 的任务的任何独立属性, 但不取决于依赖于其相对优先级顺序的任务的任何属性。
2. 根据测试 S , 任务 τ_x 的可调度性可能取决于优先级低于 $pri(\tau_x)$ 的任务的任何独立属性, 但不取决于依赖于其相对优先级顺序的任务的任何属性。
3. 当任何两个相邻优先级的任务的优先级交换时, 如果先前可以以较低优先级进行调度, 则根据测试 S , 被分配了较高优先级的任务不能变为不可调度。

holistic 的可调度性测试违反了条件 1 和 2, 其中任务的响应时间取决于系统中所有其它任务的潜在响应时间, 并且这种依赖关系可能形成循环链。例如, τ_1 和 τ_2 被分配到不同的处理器上, 共享相同的资源, 因此计算任一任务的响应时间都需要计算另一任务的响应时间。在上文“面向资源的优先级排序算法介绍”中我们就分析了这种依赖会导致计算响应时间会出现的问题。所以, OPA 和 RPA 算法, 在 *holistic* 可调度性测试下, 也不是最优的, 需要进一步优化。

于是我们实现 SPO 算法来优化。SPO 引入了响应时间近似方法, 以尽量减少由于在整体可调度性测试中使用而产生的 *pessimism* 影响。本质上, SPO 考虑了响应时间依赖性, 旨在尽量减少具有这种依赖性的响应时间计算中的 *pessimism* 影响。SPO 在迭代过程中逐渐取代了对达到响应时间固定点的任务的悲观上限, 从而得到比 OPA-D 和 RPA-D 得到更准确的响应时间。

算法伪代码如下

Algorithm 1: The SPO Algorithm

```

1 for  $m=0, \dots, M$  do
2   for each priority level  $Pri$ , lowest first do
3     for  $\tau_x \in \text{unexamined tasks on } P_m$  do
4       Assuming that all unassigned tasks in  $P_m$ 
        have higher priorities, calculate  $\lambda_x$  for each
        unexamined task  $\tau_x$  by Algorithm 2;
       end
5     Assign priority  $Pri$  to the task with the largest
         $\lambda_x$ ;
     end
6   Get response times of all tasks in  $P_m$  via test  $S$ , and
        set  $R = D$  for all unexamined tasks and tasks with
         $R > D$  on  $P_m$ ;
   end
7 Get response time of all tasks in the system via  $S$ ;
8 if system is schedulable then
9   return true;
  else
10  return false;
  end

```

Algorithm 2: Computing λ_x for τ_x

```

1 Set  $R = D$  for each unexamined remote task;
2 Calculate response times of all tasks in  $P(\tau_x)$  iteratively
  and alternately by test  $S$ . The calculation ends when  $R$ 
  is fixed for each task in  $P_m$ , or  $R \geq \eta \cdot D$  for all other
  tasks that missed their deadlines;5
3 return  $D_x - R_x$ ;

```

图 4.9 SPO 算法及相关变量的计算

算法 1 给出了 SPO 算法的伪代码，其中 S 表示底层可调度测试。如算法所示，SPO 假设在应用优先级排序之前，任务已预先分配给每个处理器。对于具有 M 个处理器的系统，该算法从第一个处理器 P_0 开始，并为每个处理器中的任务生成优先级排序。对于给定的处理器 P_m ，该算法通过算法 2 计算每个优先级 Pri 上所有未检查任务的可用容量 λ_x （其中对于每个 τ_x ， $\lambda_x = D_x - R_x$ ）。然后，为具有最大可用容量 λ_x 的任务分配优先级（第 4 行）。在检查 P_m 中的每个任务后，在第 6 行计算该处理器中任务的响应时间。然后，该算法对每个处理器中的任务重复上述过程。最后，为每个任务分配一个优先级，SPO 通过可调度性测试 S 来验证给定的优先级排序是否能够形成可调度系统。

与 OPA-D 和 RPA-D 不同，SPO 算法将因假设所有其他任务的响应时间都等于它们的截止时间导致的 *pessimism* 降至最低。首先，为了估计给定任务的响应时间，SPO 在其托管处理器的上下文中计算其响应时间。也就是说，给定任务的响应时间是通过迭代和交替计算该处理器中所有任务的响应时间来计算的。具体而言，在每次迭代中，从优先级最高的任务开始，每个任务的响应时间都计算一次。以三个任务 τ_1 、 τ_2 、 τ_3 的划分为

例, 其中 $pri(\tau_1) > pri(\tau_2) > pri(\tau_3)$, 例如, R_2 的计算是通过迭代所有三个任务的响应时间计算来执行的, 以应对所讨论的响应时间依赖性。在每次迭代中, τ_1 、 τ_2 和 τ_3 的响应时间按此顺序计算一次以更新 R_2 。当所有任务都达到固定点或 $R \geq \eta \cdot D$ 时, 迭代过程终止。由此考虑了来自局部高优先级任务的响应时间依赖关系 (见算法 1 中的第 6 行和算法 2 中的第 2 行), 因而消除了忽略这种依赖关系 (例如 OPA-D 方法) 所导致的 *pessimism*

至于远程任务的响应时间依赖性, 该算法仅对未经检查的远程任务 (即未分配优先级的任务) 保持 $R = D$ 的假设。例如, 在为 P_5 中的任务分配优先级时, 使用 P_0 至 P_4 中任务的响应时间 (由前几轮的第 6 行计算) 来计算 P_5 中任务的响应时间, 而不是它们的截止时间。通过这样做, 通常可以获得 P_5 中任务更准确 (不太悲观) 的响应时间, 因为对于所有已检查的任务, $R \leq D$ 。

为此, 可以确定 SPO 的时间复杂度。尽管该算法包含三级嵌套循环, 但总共最多需要对第 4 行进行 $n(n+1)/2$ 次调用 (与 OPA 和 RPA 类似) 才能为具有 n 个任务的任务集分配优先级。对于第 4 行, 不同于 OPA 和 RPA 的 S 只进行一次调用, SPO 对测试 S 发出最多 n 次调用, 以获取每个任务的响应时间近似值。因此, SPO 算法的时间复杂度为 $O(n^3)$ 。但请注意实际可调度性测试 S 的时间复杂度为非多项式。

此外, 与 OPA 和 RPA 不同, SPO 允许所有任务都错过给定优先级的截止时间的情况。由于未检查的远程任务的响应时间被假定为其截止时间, 因此可能会出现任务的响应时间高于给定优先级下的截止时间, 但其中一些任务实际上是可调度的情况。在这种情况下, SPO 算法旨在将优先级分配给这些错过截止时间的任务中最接近可调度的任务。为实现这一点, 引入了一个扩展参数 η 来扩展迭代响应时间计算。其中, 对于除 τ_x (当前研究的任务) 之外的所有错过截止期限的任务的响应时间已达到 $\eta \cdot D$ 时, 计算才结束, 如算法 2 所示。

对于给定的优先级, 可能有几个任务的响应时间略高于其截止时间。然而, 随着进一步的迭代计算, 这些任务的 λ 值之间的差异可能会显现或放大, 其中与其他任务相比, 具有最高 λ 值的任务通常最接近在此优先级别下可调度。因此, 通过使用这种方法, 任务更有可能被分配适当的优先级, 从而增加了获得可行优先级排序的可能性。这种方法也在 [13] 中应用于不同的环境并证明了其有效性, 其中任务的响应时间计算扩展到错过截止时间的任务, 以确定最接近可调度的系统配置。

4.5.3 面向资源的优先级排序算法实现

算法的最重要的部分就是计算出每个任务的响应时间 R , C 语言实现响应时间计算的函数如下:

```
int  getResponseTimeSP0(SporadicTask  tasks[MAX_TASKS][MAX_TASKS],  Resource
resources[MAX_RESOURCES], int num_partitions, int isMSRP) {
    if (tasks == NULL)
        return 0;

    // Assign priorities by Deadline Monotonic
    assignPrioritiesByDM(tasks, num_partitions);
```

```

    long dummy_response_time[MAX_TASKS][MAX_TASKS]; // dummy_response_time 全部
    赋值为 deadline
    for (int i = 0; i < num_partitions; i++) {
        for (int j = 0; j < num_partitions; j++) {
            dummy_response_time[i][j] = tasks[i][j].deadline;
        }
    }

    // Now we check each task in each processor
    for (int i = 0; i < num_partitions; i++) {
        int partition = i;
        SporadicTask unassignedTasks[MAX_TASKS];
        for (int j = 0; j < num_partitions; j++) {
            unassignedTasks[j] = tasks[partition][j];
        }
        int sratingP = 500 - num_partitions * 2;
        int prioLevels = num_partitions;

        // For each priority level
        for (int currentLevel = 0; currentLevel < prioLevels; currentLevel++) {
            int startingIndex = num_partitions - 1;

            for (int j = startingIndex; j >= 0; j--) {
                SporadicTask task = unassignedTasks[j];
                int originalP = task.priority;
                task.priority = sratingP;

                // Sorting tasks based on priority
                qsort(tasks[partition], num_partitions, sizeof(SporadicTask),
                    comparePriority);

                long timeBTB = getResponseTimeForSBP0(partition, tasks, resources,
                    dummy_response_time, 1, 1, isMSRP, task, extendCal);

                task.priority = originalP;

                // Update addition_slack_BTBT
                task.addition_slack_BTBT = task.deadline - timeBTB;
            }

            // Sorting tasks based on addition_slack_BTBT
            qsort(unassignedTasks, num_partitions, sizeof(SporadicTask),
                compareSlack);

            // Update priority and remove task
            unassignedTasks[0].priority = sratingP;

            qsort(tasks[partition], num_partitions, sizeof(SporadicTask),

```

```

comparePriority);

        remove(unassignedTasks,0);
        sratingP += 2;
    }
qsort(tasks[partition], num_partitions, sizeof(SporadicTask),comparePriority);
    // Update dummy_response_time
    dummy_response_time[partition]=getResponseTimeForOnePartition(partition,
tasks, resources, dummy_response_time, 1, 1, isMSRP, 1);
}

int isEqual = 0, missdeadline = 0;
long response_time[MAX_TASKS][MAX_TASKS];
initResponseTime(tasks, response_time);

/* a huge busy window to get a fixed Ri */
while (!isEqual) {
    isEqual = 1;
    long response_time_plus[MAX_TASKS][MAX_TASKS];
    busyWindow(tasks, resources, response_time, 1, 1, isMSRP);

    for (int i = 0; i < num_partitions; i++) {
        for (int j = 0; j < num_partitions; j++) {
            if (response_time[i][j] != response_time_plus[i][j])
                isEqual = 0;
            if (response_time_plus[i][j] > tasks[i][j].deadline)
                missdeadline = 1;
        }
    }

    cloneList(response_time_plus, response_time);

    if (missdeadline)
        break;
}

if (isSystemSchedulable(tasks, response_time))
    return 1;
else
    return 0;
}

```

由于时间有限, 目前我们正在写 *SPO* 算法的调度接口。以上的实现暂未接入接口中, 所以不一定是最终版本, 最终用 *SPO* 算法计算任务响应时间的代码会由接口的具体实现而定。但是总体框架如此。

编写 *LitmusRT* 调度器插件

由于我们项目自己的调度插件接口暂时没有完成，我先在 *litmus-RT* 上编写我的调度插件，然后当本项目的接口编写完成后移植改插件到自己的系统中。以下是编写 *LitmusRT* 调度器插件的代码。位于 *litmus/litmus-rt/litmus/* 文件下，名字为 *sched_demo.c*

```
#include <linux/module.h>
#include <linux/percpu.h>
#include <linux/sched.h>
#include <litmus/litmus.h>
#include <litmus/budget.h>
#include <litmus/edf_common.h>
#include <litmus/jobs.h>
#include <litmus/litmus_proc.h>
#include <litmus/debug_trace.h>
#include <litmus/preempt.h>
#include <litmus/rt_domain.h>
#include <litmus/sched_plugin.h>

struct demo_cpu_state {
    rt_domain_t    local_queues;
    int            cpu;

    struct task_struct* scheduled;
};

static DEFINE_PER_CPU(struct demo_cpu_state, demo_cpu_state);

#define cpu_state_for(cpu_id)    (&per_cpu(demo_cpu_state, cpu_id))
#define local_cpu_state()        (this_cpu_ptr(&demo_cpu_state))

static struct domain_proc_info demo_domain_proc_info;

static long demo_get_domain_proc_info(struct domain_proc_info **ret) {
    *ret = &demo_domain_proc_info;
    return 0;
}

static void demo_setup_domain_proc(void) {
    int i, cpu;
    int num_rt_cpus = num_online_cpus();

    struct cd_mapping *cpu_map, *domain_map;

    memset(&demo_domain_proc_info, 0, sizeof(demo_domain_proc_info));
    init_domain_proc_info(&demo_domain_proc_info, num_rt_cpus, num_rt_cpus);
    demo_domain_proc_info.num_cpus = num_rt_cpus;
    demo_domain_proc_info.num_domains = num_rt_cpus;

    i = 0;
```

```

    for_each_online_cpu(cpu) {
        cpu_map = &demo_domain_proc_info.cpu_to_domains[i];
        domain_map = &demo_domain_proc_info.domain_to_cpus[i];

        cpu_map->id = cpu;
        domain_map->id = i;
        cpumask_set_cpu(i, cpu_map->mask);
        cpumask_set_cpu(cpu, domain_map->mask);
        ++i;
    }
}

static void demo_job_completion(struct task_struct *prev, int budget_exhausted)
{
    prepare_for_next_period(prev);
}

static void demo_requeue(struct task_struct *tsk, struct demo_cpu_state
*cpu_state) {
    if (is_released(tsk, litmus_clock())) {
        __add_ready(&cpu_state->local_queues, tsk);
    } else {
        add_release(&cpu_state->local_queues, tsk);
    }
}

static int demo_check_for_preemption_on_release(rt_domain_t *local_queues) {
    struct demo_cpu_state *state = container_of(local_queues, struct
demo_cpu_state, local_queues);

    if (edf_preemption_needed(local_queues, state->scheduled)) {
        preempt_if_preemptable(state->scheduled, state->cpu);
        return 1;
    }
    return 0;
}

static long demo_activate_plugin(void) {
    int cpu;
    struct demo_cpu_state *state;

    for_each_online_cpu(cpu) {
        TRACE("Initializing CPU%d...\n", cpu);
        state = cpu_state_for(cpu);
        state->cpu = cpu;
        state->scheduled = NULL;
        edf_domain_init(&state->local_queues,
                        demo_check_for_preemption_on_release,

```

```

        NULL);
    }
    demo_setup_domain_proc(); // 设置域信息
    return 0;
}

static long demo_deactivate_plugin(void) {
    destroy_domain_proc_info(&demo_domain_proc_info);
    return 0;
}

static struct task_struct* demo_schedule(struct task_struct * prev) {
    struct demo_cpu_state *local_state = local_cpu_state(); // 当前 CPU 状态, 主要是得到里面的前驱任务(scheduled 字段)

    struct task_struct *next = NULL; // next 指针为 null 表示调度后台任务
    int exists, out_of_time, job_completed, self_suspends, preempt, resched; // 前驱任务状态

    raw_spin_lock(&local_state->local_queues.ready_lock); // 持有当前 CPU 就绪队列的自旋锁

    BUG_ON(local_state->scheduled && local_state->scheduled != prev); // 确保 local_state 中的 scheduled 字段指向了 prev 参数
    BUG_ON(local_state->scheduled && !is_realtime(prev)); // 确保 prev 是实时任务

    exists = local_state->scheduled != NULL;
    self_suspends = exists && !is_current_running();
    out_of_time = exists && budget_enforced(prev) && budget_exhausted(prev);
    job_completed = exists && is_completed(prev);

    preempt = edf_preemption_needed(&local_state->local_queues, prev); // 如果就绪队列中存在比前驱任务优先级更高的任务, 就进行抢占

    resched = preempt; // 检查重调度的所有条件
    if (self_suspends) { // 如果前驱挂起, 不会对其重调度
        resched = 1;
    }

    if (out_of_time || job_completed) { // 如果前驱超时或者完成了, 也不会重调度
        demo_job_completion(prev, out_of_time);
        resched = 1;
    }

    if (resched) {
        // 如果此时不打算对前驱重调度, 并且前驱不是挂起(而是完成或者超时), 就将其再次入队
        if (exists && !self_suspends) {
            demo_requeue(prev, local_state);
        }
    }
}

```

```

    }
    next = __take_ready(&local_state->local_queues); // 取出就绪队列中的就绪任
务, 做为后继
    } else {
// 如果打算重调度前驱, 后继就是前驱
    next = local_state->scheduled;
    }

    local_state->scheduled = next; // 指定要调度的任务为后继
    if (exists && prev != next) { // 如果后继不是前驱, 那就是要调度新的任务
        TRACE_TASK(prev, "descheduled.\n");
    }
    if (next) {
        TRACE_TASK(next, "scheduled.\n");
    }

    sched_state_task_picked(); // 告诉 litmus 内核, 我们已经做了一个调度决策

    raw_spin_unlock(&local_state->local_queues.ready_lock); // 释放锁
    return next;
}

static long demo_admit_task(struct task_struct *tsk) {
    if (task_cpu(tsk) == get_partition(tsk)) {
        TRACE_TASK(tsk, "accepted by demo plugin.\n");
        return 0;
    }
    return -EINVAL;
}

static void demo_task_new(struct task_struct *tsk, int on_runqueue, int
is_running) {
    unsigned long flags;
    struct demo_cpu_state *state = cpu_state_for(get_partition(tsk));
    lt_t now;
    TRACE_TASK(tsk, "is a new RT task %llu (on runqueue:%d, running:%d)\n",
        litmus_clock(), on_runqueue, is_running);

    raw_spin_lock_irqsave(&state->local_queues.ready_lock, flags);
    now = litmus_clock();

    release_at(tsk, now);
    if (is_running) {
        BUG_ON(state->scheduled != NULL);
        state->scheduled = tsk;
    } else if (on_runqueue) {
        demo_requeue(tsk, state);
    }
}

```

```

    if (edf_preemption_needed(&state->local_queues, state->scheduled)) { // 抢占检查
        preempt_if_preemptable(state->scheduled, state->cpu);
    }

    raw_spin_unlock_irqrestore(&state->local_queues.ready_lock, flags);
}

static void demo_task_exit(struct task_struct *tsk) {
    unsigned long flags;
    struct demo_cpu_state *state = cpu_state_for(get_partition(tsk));
    raw_spin_lock_irqsave(&state->local_queues.ready_lock, flags);

    // 为了方便，我们假设任务不会再次入队了，这对应任务自己退出的情况；如果任务是被其他任务强行退出实时模式的，就要进行额外的队列管理
    if (state->scheduled == tsk) {
        state->scheduled = NULL;
    }

    raw_spin_unlock_irqrestore(&state->local_queues.ready_lock, flags);
}

static void demo_task_resume(struct task_struct *tsk) {
    unsigned long flags;
    struct demo_cpu_state *state = cpu_state_for(get_partition(tsk));
    lt_t now;
    TRACE_TASK(tsk, "woke up at %llu\n", litmus_clock());
    raw_spin_lock_irqsave(&state->local_queues.ready_lock, flags);

    now = litmus_clock();

    if (is_sporadic(tsk) && is_tardy(tsk, now)) {
        release_at(tsk, now);
    }
    if (state->scheduled != tsk) {
        demo_requeue(tsk, state);
        if (edf_preemption_needed(&state->local_queues, state->scheduled)) { // 抢占检查
            preempt_if_preemptable(state->scheduled, state->cpu);
        }
    }

    raw_spin_unlock_irqrestore(&state->local_queues.ready_lock, flags);
}

static struct sched_plugin demo_plugin = {
    .plugin_name      = "DEMO",

```



```

        .schedule                = demo_schedule,
        .task_wake_up            = demo_task_resume,
        .admit_task              = demo_admit_task,
        .task_new                = demo_task_new,
        .task_exit               = demo_task_exit,
        .get_domain_proc_info    = demo_get_domain_proc_info,
        .activate_plugin         = demo_activate_plugin,
        .deactivate_plugin       = demo_deactivate_plugin, // 注册销毁插件回调
        .complete_job            = complete_job,
};

static int __init init_demo(void) {
    return register_sched_plugin(&demo_plugin);
}

```

```
module_init(init_demo);
```

编译安装

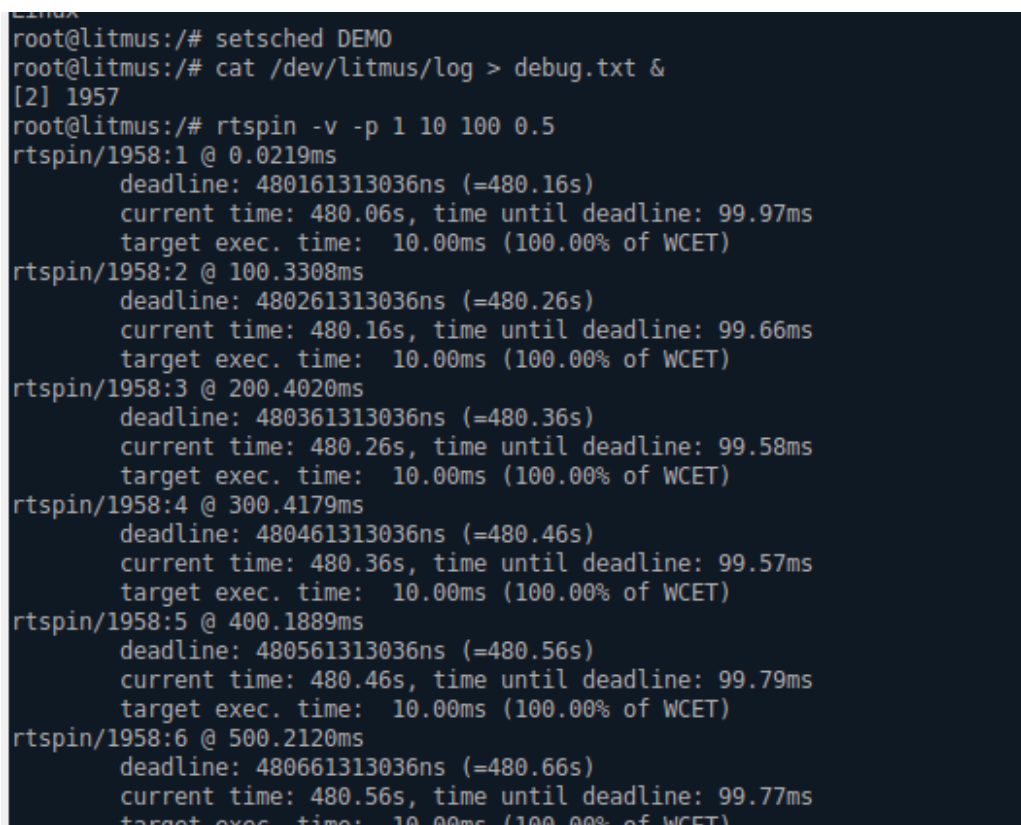
写完 `sched_demo.c` 文件还不算，我们要把要链接的文件名添加到 `litmus` 目录中的 `MakeFile` 中，在 `obj-y=...` 的后面一长串里，加上我们的 `sche_demo.o` 即可

```

.....
sched_pfp.o \
sched_demo.o

```

测试调度插件的运行：



```

Linux
root@litmus:/# setsched DEMO
root@litmus:/# cat /dev/litmus/log > debug.txt &
[2] 1957
root@litmus:/# rtspin -v -p 1 10 100 0.5
rtspin/1958:1 @ 0.0219ms
    deadline: 480161313036ns (=480.16s)
    current time: 480.06s, time until deadline: 99.97ms
    target exec. time: 10.00ms (100.00% of WCET)
rtspin/1958:2 @ 100.3308ms
    deadline: 480261313036ns (=480.26s)
    current time: 480.16s, time until deadline: 99.66ms
    target exec. time: 10.00ms (100.00% of WCET)
rtspin/1958:3 @ 200.4020ms
    deadline: 480361313036ns (=480.36s)
    current time: 480.26s, time until deadline: 99.58ms
    target exec. time: 10.00ms (100.00% of WCET)
rtspin/1958:4 @ 300.4179ms
    deadline: 480461313036ns (=480.46s)
    current time: 480.36s, time until deadline: 99.57ms
    target exec. time: 10.00ms (100.00% of WCET)
rtspin/1958:5 @ 400.1889ms
    deadline: 480561313036ns (=480.56s)
    current time: 480.46s, time until deadline: 99.79ms
    target exec. time: 10.00ms (100.00% of WCET)
rtspin/1958:6 @ 500.2120ms
    deadline: 480661313036ns (=480.66s)
    current time: 480.56s, time until deadline: 99.77ms
    target exec. time: 10.00ms (100.00% of WCET)

```

图 4.10 让内核跳转到调用我们实现的调度算法的流程

4.6 考虑阻塞的任务分配机制设计和实现

4.6.1 考虑阻塞的任务分配机制介绍

随着实时系统中实现的功能变得越来越复杂，系统中的任务通常需要对同一对象进行操作，而这些对象必须以互斥的方式进行访问，包括代码段、内存块、I/O 端口等。AUTOSAR 标准 [14] 强制要求的自旋锁被广泛采用，用于保护共享资源上的计算免受竞争条件的影响，并应用资源共享协议来管理对共享资源的访问 [15]。然而，在强制执行自旋锁的情况下，系统中的任务可能会由于共享资源的竞争而引发资源争用。特别是当任务从不同的处理器请求相同的资源时，它们会在其主机处理器上忙等待，直到请求的资源被授予。这会导致这些处理器中所有任务的潜在高阻塞，并可能显著破坏系统的可调度性 [8]。

为了减轻处理器之间的资源争用，我们提出了一种具有自旋锁管理共享资源的 *FPPPS (Fully-Partitioned Fixed-Priority Scheduling)* 多处理器系统的任务分配方法 *RAF*，其中共享资源由来自其主机处理器的任务访问。*RAF* 不依赖时间限制作为指导，而是利用资源竞争模型 (RCM) 来估计任务之间的资源争用程度。基于 RCM 的指导，*RAF* 始终将资源争用最高的任务分配给同一处理器，以减少资源争用，从而减少任务的阻塞。

4.6.2 考虑阻塞的任务分配机制设计

1. 设计目标：

- 减少资源争用
- 降低任务阻塞时间，提高系统可调度性
- 通用性和高效性

所提出的任务分配方法 *RAF* 利用了资源争用模型 (RCM)，该模型可以近似估计任务之间的资源争用程度，即争用系数 (CF)。利用近似 CF 作为指导，该方法可根据以下基本原则做出分配决策：始终 (i) 将 CF 近似值最高的任务分组；(ii) 将 CF 近似值最高的任务组分配给一个处理器。

2. 系统模型：

设系统包含一组相同处理器 Λ 和一组在 *FP-FPS* 方案下调度的随机任务 Γ (*sporadic task*)。 Λ 中的第 x 个处理器表示为 λ_x 。任务 τ_i (Γ 中的第 i 个任务) 由 $\tau_i = \{C_i, T_i, D_i, P_i, A_i\}$ 定义，其中 C_i 是不访问共享资源的纯最坏情况执行时间， T_i 是周期， D_i 是 $D_i \leq T_i$ 的约束期限， P_i 是优先级， A_i 是分配。每个任务都有唯一的优先级， P_i 值越高表示优先级越高。函数 $\Gamma(\lambda_x)$ 返回分配给 λ_x 的所有任务的集合。为简单起见，我们使用 U 来表示给定任务或任务组的利用率，例如分别为 U_{τ_i} 和 $U_{\Gamma(\lambda_x)}$ 。函数 $|\cdot|$ 表示给定集合的大小，例如 $|\Lambda|$ 给出系统中处理器的数量。该系统还包含一组受自旋锁保护的共享资源 R 。对于资源 r^k (R 中的第 k 个资源)， c^k 表示 r^k 上的最坏情况计算时间， N_i^k 给出由 τ_i 在一次发布中向 r^k 发出的请求的数量。函数 $F(\cdot)$ 表示给定任务所请求的资源。多个处理器中的任务之间共享的资源称为全局资源，而在处理器内访问的资源称为本地资源。

3. 整体机制：

对于一组给定的任务，RAF 会形成一组任务组 G ，其中每个组都包含资源争用程度较高的任务。此外，还对所有任务组应用了一个利用率约束 \bar{U} ，以约束一个组允许的最大利用率，即 $\bar{U} \geq U_{\mathcal{G}_a}, \forall \mathcal{G}_a \in G$ 。在这项工作中， \bar{U} 被设置为系统在每个处理器中的平均利用率，即 $\bar{U} = U_{\Gamma} / |\Lambda|$ 。构建好 G 后，就可以根据任务组与处理器上现有任务之间的 CF 近似值进行分配，将任务组分配给处理器。

如上所述，RAF 中的组形成和分配决策都是基于 RCM 提供的 CF 近似做出的。为了指导分配决策，RCM 需要为任意两个给定任务组生成 CF 近似值，表示为 $\Delta(\mathcal{G}_a, \mathcal{G}_b)$ 。 $\Delta(\mathcal{G}_a, \mathcal{G}_b)$ 值越高表示任务组之间的资源争用程度越高。对于不共享任何资源的 \mathcal{G}_a 和 \mathcal{G}_b ，RCM 将通知分配器 $\Delta(\mathcal{G}_a, \mathcal{G}_b) = 0$ 。此外，具有单个任务 τ_i 的输入被视为具有一个任务的一组，即 $\{\tau_i\}$ 。

RAF 由基于资源争用的任务分组、处理器任务组的分配、资源争用模型 (RCM) 三个子部分组成：

(1) 基于资源争用的任务分组

任务分组技术遵循以下原则：始终合并具有最高 CF 近似值且强制执行 \bar{U} 的两个组。这样只有具有高资源争用的任务才会被合并到一组中，即使它们与系统中的其他任务共享某些资源。此外，使用 \bar{U} 有效地避免了创建无法安装到任何处理器中的重型组。

Algorithm 1: Task group formation

```

1  $G = \emptyset$ ;
2 for each  $\tau_i \in \Gamma$  do
3    $\mathcal{G}_a = \{\tau_i\}$ ;  $G = G \cup \mathcal{G}_a$ ;
4 end
5 compute  $\Delta(\mathcal{G}_a, \mathcal{G}_b), \forall \mathcal{G}_a, \mathcal{G}_b \in G \wedge a \neq b$ ;
6 while true do
7    $(\mathcal{G}_a, \mathcal{G}_b) = \underset{a,b}{\operatorname{argmax}} \{ \Delta(\mathcal{G}_a, \mathcal{G}_b) \mid U_{\mathcal{G}_a} + U_{\mathcal{G}_b} \leq \bar{U} \}$ ;
8   if  $\Delta(\mathcal{G}_a, \mathcal{G}_b) = 0 \vee (\mathcal{G}_a, \mathcal{G}_b) = \emptyset$  then
9     return  $G$ ;
10  end
11   $\mathcal{G}_a = \mathcal{G}_a \cup \mathcal{G}_b$ ;  $G = G \setminus \mathcal{G}_b$ ;
12  Compute  $\Delta(\mathcal{G}_a, \mathcal{G}_s), \forall \mathcal{G}_s \in G \wedge a \neq s$ ;
13 end
14 return  $G$ ;

```

图 4.11 Algorithm 1: Task group formation

算法 1（图 4.11）给出了基于 RCM 的完整任务分组过程。该算法以系统中的所有任务 Γ 作为输入，初始化 $|\Gamma|$ 任务组，其中每个组包含一个任务（第 1-4 行）。对于任何两个组 \mathcal{G}_a 和 \mathcal{G}_b ， CF 近似值是使用 RCM 计算的，即第 5 行中的 $\Delta(\mathcal{G}_a, \mathcal{G}_b)$ 。然后，在满足 $U_{\mathcal{G}_a} + U_{\mathcal{G}_b} \leq \bar{U}$ 的任意两个组中，找出 CF 近似值最高的一对组，开始组形成（第 7 行）。如果这样的组对是通过正 CF 近似获得的，则通过合并两个组来创建一个新组（第 11 行）。此外，还会计算新创建的组与 G 中现有组之间的 CF 近似值，以便在后续迭代中进一步形成组（第 12 行）。这个过程一直持续到每个组独立于其他组（即 $\Delta(\mathcal{G}_a, \mathcal{G}_b) = 0$ ）

或没有组可以在强制执行 \bar{U} 的情况下合并为止（第 8-10 行）。由于前一种情况，具有单一任务且不共享任何资源的组在整个过程中不会被合并。

特点 1: 所提出的方法形成一组任务组，其中每个组中的任务具有最高的竞争性。此外， \bar{U} 还用于避免形成无法在单个处理器中分配的大型任务组。

(2) 处理器任务组的分配

Algorithm 2: Resource-aware task allocation

```

1 for each  $\mathcal{G}_a \in G$  do
2    $\Omega_a = \sum_{\tau_i \in \mathcal{G}_a} \Delta(\{\tau_i\}, \mathcal{G}_a \setminus \tau_i)$ ;
3 end
4 for each  $x \in [1 \dots \min\{|G|, |\Lambda|\}]$  do
5    $\mathcal{G}_a = \underset{a}{\operatorname{argmax}} \{\Omega_a \mid \forall \mathcal{G}_a \in G\}$ ;
6    $A_i = \lambda_x, \forall \tau_i \in \mathcal{G}_a; \quad G = G \setminus \mathcal{G}_a$ ;
7 end
8 while  $G \neq \emptyset$  do
9    $\lambda_x = \underset{x}{\operatorname{argmin}} \{U_{\Gamma(\lambda_x)} \mid \forall \lambda_x \in \Lambda\}$ ;
10   $\mathcal{G}_a = \underset{a}{\operatorname{argmax}} \{\Delta(\mathcal{G}_a, \Gamma(\lambda_x)) \mid \forall \mathcal{G}_a \in G\}$ ;
11  if  $U_{\lambda_x} + U_{\mathcal{G}_a} \leq 1$  then
12     $A_i = \lambda_x, \forall \tau_i \in \mathcal{G}_a; \quad G = G \setminus \mathcal{G}_a$ ;
13  else
14    for each  $\tau_i \in \mathcal{G}_a$ , highest  $\Delta(\{\tau_i\}, \Gamma(\lambda_x))$  first do
15      if  $U_{\lambda_x} + U_{\tau_i} \leq 1$  then
16         $A_i = \lambda_x; \quad \mathcal{G}_a = \mathcal{G}_a \setminus \tau_i$ ;
17      end
18    end
19    if no task in  $\mathcal{G}_a$  is allocated to  $\lambda_x$  then
20      return unfeasible;
21    end
22  end
23 end
24 return  $\{A_i, \forall \tau_i \in \Gamma\}$ ;

```

图 4.12 Algorithm 2: Resource-aware task allocation

构建 G 后，执行算法 2（图 4.12），将 G 映射到一组处理器 Λ 。该算法首先根据 CF 计算每个 \mathcal{G}_a 的权重 Ω_a （第 1-3 行）。这个权重反映了 \mathcal{G}_a 中任务之间内部资源竞争的程度。然后分两步进行分配。第一步基于 Ω_a 对 G 中最多 $|\Lambda|$ 组产生初始分配（第 4-7 行）。然后，第二步根据 RCM 的指导决定其余任务组的分配（第 8-23 行）。

在第一步中，该方法识别最多 $\min\{|G|, |\Lambda|\}$ 个具有最高 Ω_a 的任务组，并将这些组中的每一个分配给一个处理器（第 5-6 行）。对于具有相同 Ω_a 的组，采用利用率较高的组。因为所有组都是在 \bar{U} 强制且 $\bar{U} \leq 1$ 的情况下形成的，所以可行。通过在这些组中本地化任务，我们减轻了系统中具有最高 CF 近似值的任务之间的争用。

初始分配之后，第二步按照以下规则映射未分配的组：具有最小 $U_{\Gamma(\lambda_x)}$ 的 λ_x 优先，具有最高 $\Delta(\mathcal{G}_a, \Gamma(\lambda_x))$ 的 \mathcal{G}_a 优先（第 9-10 行）。这增加了分配完整任务组的机会，并且可以减少一般情况下的资源争用。如果 \mathcal{G}_a 适合 λ_x ，则组中的所有任务都分配给该处理器（第 11-12 行）。否则，为了提高分配方法的成功率并减少竞争， \mathcal{G}_a 中的任务被单独分配，其中具有最高 $\Delta(\{\tau_i\}, \Gamma(\lambda_x))$ 的任务总是分配给 λ_x 。然而，在这种情况下，如果 \mathcal{G}_a

中没有任务可以分配给 λ_x (即利用率最低的处理器), 则算法立即返回, 找不到可行的分配 (第 19-21 行)。整个过程直到每项任务都得到分配, 即 $A_i, \forall \tau_i \in \Gamma$, 才算结束。

特点 2: 所提出的方法基于 RCM 将任务分配给处理器, 而不需要特定的可调度性测试。

特点 3: 所提出的方法下的任务不需要额外的运行时设施, 例如 [16] 中的迁移, 并且运行时开销可以忽略不计, 相当于传统的 FPFPS 系统。

这总结了所提出的任务分配方法的描述。该方法的时间复杂度为 $O(n^2)$, 需要最多为 $|\Gamma| \times |\Gamma|$ 次迭代来 (i) 计算 G 中所有组对的 $\Delta(\mathcal{G}_a, \mathcal{G}_b)$, (ii) 形成任务组, 以及 (iii) 将任务组分配给处理器。正如分配过程所示, RCM 提供的 CF 近似值对于所提出的分配方法至关重要。

(3) 资源争用模型 (RCM)

RCM 既没有为具有共享资源的任务提供安全的计时界限, 也没有为任务之间的资源争用提供准确的量化。RCM 的目标是考虑自旋锁的主要特征, 在任务之间生成具有足够准确度的 CF 近似值, 以便为任务分配提供有效的指导, 从而减少争用。同时, 这样做也使得 RCM 可以快速地产生产 CF 近似值, 并与优先级分配算法、资源共享协议和可调度性测试分离。

如 [8] 中所述, 根据资源请求的服务顺序, 自旋锁可大致分为无序锁、先进先出锁和基于优先级的锁。其中, FIFO 和基于优先级的自旋锁都强制规定特定的资源访问顺序, RCM 可以捕获该顺序以提供准确的 CF 近似值。我们以资源访问顺序作为先验知识构建了细粒度的 RCM。

在 FIFO 自旋锁下, 每个全局资源都与一个 FIFO 队列关联, 该队列包含来自所有处理器的请求 (基于其到达顺序), 其中队列头部的任务始终被授予资源 [11]。这引出了引理 1。

引理 1: 使用 FIFO 自旋锁, 对 r_k 的 τ_i 请求最多可以被远程处理器对 r_k 的请求阻塞一次。

证明: 对于非抢占式自旋锁, 资源访问严格遵循“鸽巢原理”, 即处理器在任何时刻最多只能发出一个请求。这在 [8]、[10] 中得到证明。对于抢占式任务, 较高优先级的任务可以抢占自旋任务 (spinning task) τ_i 以获得 r_k , 但是这些任务在抢占 τ_i 后无法再请求 r_k 。否则, 可能会发生死锁, 其中 τ_i 无法被服务, 因为它被 FIFO 队列中在 τ_i 之后等待 (自旋) 的任务抢占。因此, 处理器在任何时刻最多包含一个带有 FIFO 自旋锁的全局资源请求。

根据引理 1, 在 FIFO 自旋锁下, τ_i 可能从 \mathcal{G}_a 中引起的争用在 公式 14 中估计, 表示为 $\phi^{fifo}(\tau_i, \mathcal{G}_a)$ 。符号 $N_i^k(t)$ 给出了在时间段 t 内从 τ_i 到 r^k 的请求数量, 即 $N_i^k(t) = \lceil \frac{t}{T_i} \rceil \times N_i^k$ 。假设 τ_i 和 \mathcal{G}_a 分配在两个不同的处理器上, 则 τ_i 会因 \mathcal{G}_a 中的任务访问 r^k 而导致最多 $N_{i(T_i)}^k$ 个阻塞。

$$\phi^{fifo}(\tau_i, \mathcal{G}_a) = \sum_{r^k \in \mathbb{R}} \min\{N_i^k(T_i), \sum_{\tau_j \in \mathcal{G}_a} N_j^k(T_i)\} \times c^k \quad (14)$$

对于基于优先级的自旋锁，我们假设从 τ_i 到 r^k 的请求优先级是已知的，记为 P_i^k 。使用基于优先级的自旋锁，请求全局资源的任务按其访问优先级排序，并且始终向具有最高优先级的任务授予资源。在这种情况下， τ_i 可能会因所有对 r^k 的高优先级请求和最多一个来自远程处理器的低优先级请求访问 r^k 而受到阻塞。因此，假定 τ_i 和 \mathcal{G}_a 被分配给两个不同的处理器， τ_i 可能因 \mathcal{G}_a 产生的争用可通过 [公式 15](#) 近似得出。注意在 [公式 15](#) 中添加了一个额外的请求来反映低优先级资源请求的潜在阻塞。

$$\phi^{prio}(\tau_i, \mathcal{G}_a) = \sum_{r^k \in F(\tau_i) \cap F(\mathcal{G}_a)} \left\{ \left(\sum_{\tau_j \in \mathcal{G}_a \wedge P_i^k < P_j^k} N_j^k(T_i) \times c^k \right) + c^k \right\} \quad (15)$$

通过 [公式 14](#) 和 [公式 15](#)，可以得到两种类型的自旋锁下 \mathcal{G}_a 和 \mathcal{G}_b 之间的 CF 近似值，如 [公式 16](#) 所示。该公式计算了两组任务分配给不同处理器时可能产生的争用总和，数值越大，意味着将两组任务分配给同一处理器（如果可能的话）的收益越高。

$$\Delta(\mathcal{G}_a, \mathcal{G}_b) = \sum_{\tau_i \in \mathcal{G}_a} \phi(\tau_i, \mathcal{G}_b) + \sum_{\tau_j \in \mathcal{G}_b} \phi(\tau_j, \mathcal{G}_a) \quad (16)$$

RCM 中没有明确考虑本地高优先级任务的传递阻塞。然而，通过引导分配，将具有较高 $\Delta(\mathcal{G}_a, \mathcal{G}_b)$ 的任务组映射到同一处理器，由于争用减少，它隐式地减轻了一般情况下的此类阻塞。此外，这也避免了过于复杂的计算，因为计算可能需要有关底层系统的额外信息。

4.6.3 考虑阻塞的任务分配机制实现

RAF 为任务分配机制的核心函数，将任务分配给不同的分区。

其主要步骤如下：

1. 初始化任务的分区和优先级。
2. 根据任务的截止时间对任务进行排序，并分配优先级。
3. 将访问资源的任务和不访问资源的任务分成两个列表。
4. 对访问资源的任务进行分组。
5. 将所有任务合理地放置在处理器上。
6. 先将分组（访问资源的任务）放在核心上。
7. 采用 *Worst-Fit-Decreasing* 将不访问资源的任务放在核心上

C 语言实现 RAF 函数如下：

```
/**
 * RAF 方法将任务分配给不同的分区。
 * 接收一个需要分配的任务列表，一个资源列表，总的分区数，以及每个核心的最大利用率作为输入。
 * 返回一个 SporadicTask 对象的列表的列表，其中每个列表代表一个分区，并包含分配给该分区的任务。
 *
 * @param tasksToAllocate 需要分配到不同分区的任务列表。
 * @param resources 任务可能需要的资源列表。
 * @param total_partitions 可用于任务分配的总分区数（或核心数）。
 * @param maxUtilPerCore 一个组允许的最大利用率。
```

* @return ArrayList<ArrayList<SporadicTask>> 这返回一个 SporadicTask 对象的列表的列表，其中每个列表代表一个分区，并包含分配给该分区的任务。

```

*/
GPtrArray** RAF(GPtrArray* tasksToAllocate, GPtrArray* resources,
                int total_partitions, double maxUtilPerCore)
{
    // 将所有属于 tasksToAllocate 的待分配任务的分区初始化置-1;
    for (int i = 0; i < tasksToAllocate->len; i++) {
        SporadicTask* sporadicTask = g_ptr_array_index(tasksToAllocate, i);
        sporadicTask->partition = -1; // 分区
        sporadicTask->priority = -1; // 优先级
        sporadicTask->Ri = 0; // 响应时间
    }

    // (初始化) 建立最终返回分组
    GPtrArray** final_tasks = malloc(total_partitions * sizeof(GPtrArray*));
    for(int i = 0; i < total_partitions; i++) {
        final_tasks[i] = g_ptr_array_new();
    }

    // sort the tasks by the ddls 升序排序
    GPtrArray* tasks = g_ptr_array_sized_new(tasksToAllocate->len);
    g_ptr_array_add_range(tasks, tasksToAllocate->pdata, tasksToAllocate->len);
    g_ptr_array_sort(tasks, (GCompareFunc)compareSporadicTaskByDDL);

    // grant priority
    for(int i = 0; i < tasks->len; i++){
        SporadicTask* task = g_ptr_array_index(tasks, i);
        task->priority = tasks->len - i;
    }

    // 将访问资源的任务和不访问资源的任务分成两个 list
    GPtrArray* tasksToAllocate_WithResource = g_ptr_array_new();
    GPtrArray* tasksToAllocate_NoResource = g_ptr_array_new();

    for(int i = 0; i < tasks->len; i++){
        SporadicTask* task = g_ptr_array_index(tasks, i);
        if(task->resource_required_index->len != 0){
            g_ptr_array_add(tasksToAllocate_WithResource, task);
        }
        else{
            g_ptr_array_add(tasksToAllocate_NoResource, task);
        }
    }

    /* Step1. 对于访问资源的任务进行分组 */
    // 初始化每个 task 一个 group
    GPtrArray* task_group_list =

```

```

g_ptr_array_new_with_free_func((GDestroyNotify)g_ptr_array_unref);
    for(int i = 0; i < tasksToAllocate_WithResource->len; i++){
        SporadicTask* task = g_ptr_array_index(tasksToAllocate_WithResource, i);
        GPtrArray* task_group = g_ptr_array_new();
        g_ptr_array_add(task_group, task);
        g_ptr_array_add(task_group_list, task_group);
    }

// 递归进行分组
GPtrArray* group_list = Grouping(task_group_list, resources, maxUtilPerCore);

// CF 大到小
g_ptr_array_sort_with_data(group_list, (GCompareDataFunc)compareCF,
resources);

/* Step2. 将所有任务合理地放置在 processor 上 */
// 每个核心的利用率初始化为 0
double* utilPerPartition = (double*)malloc(total_partitions * sizeof(double));
for (int i = 0; i < total_partitions; i++) {
    utilPerPartition[i] = 0.0;
}

/* Step2-1. 先将分组(访问资源的任务)放在核心上 */
// 分组数小于等于核心数
if(group_list->len <= total_partitions){
    for(int i = 0 ; i < group_list->len; i++){
        // 第 i 个 group 放在第 i 个核心上,并更新每个核心的利用率
        GPtrArray* group = g_ptr_array_index(group_list, i);
        for(int j = 0; j < group->len; j++){
            SporadicTask* task_in_group = g_ptr_array_index(group, j);
            task_in_group->partition = i;
            g_ptr_array_add(final_tasks[i], task_in_group);
            utilPerPartition[i] += task_in_group->util;
        }
    }
}

// 分组数大于核心数
else {
    for(int i = 0 ; i < total_partitions; i++){
        GPtrArray* group = g_ptr_array_index(group_list, i);
        for(int j = 0; j < group->len; j++){
            SporadicTask* task_in_group = g_ptr_array_index(group, j);
            // 第 i 个 group 放在第 i 个核心上 (前 total_partition)
            task_in_group->partition = i;
            g_ptr_array_add(final_tasks[i], task_in_group);
            utilPerPartition[i] += task_in_group->util;
        }
    }
}

```



```

    // 分好的全移除
    for(int i = 0 ; i < total_partitions; i++){
        g_ptr_array_remove_index(group_list, 0);
    }
    // 递归分配
    if(!RAFAAllocating(group_list, resources, utilPerPartition,
total_partitions, final_tasks)){
        return NULL;
    }
}

/* Step2-2.采用 Worst-Fit-Decreasing 将不访问资源的任务放在核心上 */
// 按利用率从大到小排序（降序）
g_ptr_array_sort(tasksToAllocate_NoResource,
(GCompareFunc)compareSporadicTaskByUtil);

for(int i = 0; i < tasksToAllocate_NoResource->len; i++) {
    SporadicTask* task = g_ptr_array_index(tasksToAllocate_NoResource, i);
    int target = -1;
    double minUtil = 2.0;

    // 找到利用率最小的核心作为本次分配的目标
    for(int j = 0; j < total_partitions; j++) {
        if(minUtil > utilPerPartition[j]) {
            minUtil = utilPerPartition[j];
            target = j;
        }
    }

    if(target == -1) {
        fprintf(stderr, "RAF error!\n");
        return NULL;
    }

    // 当前任务的利用率 (task->util) 小于或等于目标处理器的剩余利用率 (1 - minUtil)
    则分配
    if((double)1 - minUtil >= task->util) {
        task->partition = target;
        g_ptr_array_add(final_tasks[target], task);
        utilPerPartition[target] += task->util;
    } else {
        fprintf(stderr, "RAF error!\n");
        return NULL;
    }
}

// 在不再需要 GPtrArray 时释放内存
g_ptr_array_unref(tasks);

```

```

    g_ptr_array_unref(tasksToAllocate_WithResource);
    g_ptr_array_unref(tasksToAllocate_NoResource);
    g_ptr_array_unref(task_group_list);
    g_ptr_array_unref(group_list);

    // 返回最终的任务分组
    return final_tasks;
}

```

Grouping 函数根据任务的竞争因子 (CF) 对任务进行分组。该函数是算法 1 (图 4.11) 的具体实现, 将具有最高 CF 的组合为一个, 只要新组的总利用率不超过每个核心的最大利用率。

- *param task_group_list* 一个 *SporadicTask* 对象的列表的列表。每个列表代表一组任务。
- *param resources* 一个 *Resource* 对象的列表。这些是任务可能需要访问的资源。
- *param maxUtilPerCore* 每个处理器核心允许的最大利用率。
- *return* 分组后的任务对象列表的列表。每个列表代表一组任务。

```

/**
 * @param task_group_list 访问资源的任务对象列表的列表。每个列表代表一组任务。
 * @param resources 任务可能需要的资源列表。
 * @param maxUtilPerCore 每个处理器核心允许的最大利用率。
 *
 * @return 返回一个新的任务组列表, 其中每个任务组的利用率不超过 maxUtilPerCore。
 */

```

```

GPtrArray* Grouping(GPtrArray* task_group_list, GPtrArray* resources, double
maxUtilPerCore);`

```

Grouping 函数的主要步骤如下:

1. 计算 *task_group_list* 中每对组的 CF 并记录下来。
2. 按照 CF 的降序对任务组对进行排序。
3. 试图合并 CF 最高的一对任务组。如果合并后的利用率不超过 *maxUtilPerCore* ,则进行合并。
4. 合并后, 从 *task_group_list* 中删除原始的两个组并添加新的组。
5. 重复该过程, 直到不能再合并更多的组。
6. 算法结束, 返回当前的任务组列表。

RAFAAllocating 函数用于将任务组分配到各个处理器上。

```

/**
 * @param group_list 待分配的访问资源的任务组列表。
 * @param resources 任务可能需要的资源列表。
 * @param utilPerPartition 每个处理器核心 (分区) 的利用率列表。
 * @param total_partitions 处理器核心的总数。
 * @param final_tasks 任务对象列表的列表, 表示最终的任务分配结果。
 *
 * @return 如果所有任务组都成功分配到处理器核心上, 返回正数; 否则, 返回负数。
 */

```

```
int RAFAllocating(GPtrArray* group_list, GPtrArray* resources, double*  
utilPerPartition, int total_partitions, GPtrArray* final_tasks);
```

RAFAllocating 函数的主要步骤如下:

1. 如果所有任务组都已分配, 返回 *true*。
2. 找出剩余利用率最高的核心。
3. 根据该核心上的任务和 *group_list* 中所有组的 *CF* 进行排序。
4. 如果 *CF* 最大的组可以放入剩余利用率最高的核心, 则将该组分配到该核心, 并更新利用率数组。
5. 如果不能将整个组放入核心, 则将任务组内的任务按 *contention factor* 从大到小排序, 并尝试将组内的单个任务分配到核心, 直到不能再放入更多任务。
6. 如果一个任务都不能分配, 则返回 *false*。
7. 递归调用自身, 继续分配剩余的任务组。

5 项目测试

本小节预计会介绍 *Yat-Scheduler* 项目的测试结果，将会包含各部分的测试概述、测试方法以及相应的测试结果。*Yat-Scheduler* 项目预期需要做的测试包括以下几个部分：

- *Yat-Scheduler* 各模块功能测试
- 多核互斥资源共享协议测试
- 面向资源的优先级排序算法测试
- 考虑阻塞的任务分配机制测试
- *Yat-Scheduler* 性能测试

以下将分别进行介绍

5.1 *Yat-Scheduler* 功能测试

5.1.1 测试概述

功能测试预计会包含很多部分，包含但不限于各模块（底层数据结构部分、调度算法部分、同步处理机制部分、调度跟踪部分、系统调用）以及调度接口的测试，目前我们已经实现了部分接口的测试。

5.1.2 测试方法

接口测试的主要方法是在关键的函数（需要测试的接口）内部使用内核调试 *printk* 语句输出对应的信息，可以顺带将参数一同输出，这样测试能够更加全面，例如在空调度器的优先级判断接口和任务出入队列接口处可以这样使用 *printk* 语句：

```
bool yat_prio(struct task_struct *p)
{
    printk("\n\n=====enter judge: yat_prio=====, policy:%d \n\n", p->policy);
    // return p->policy == SCHED_YAT;
    return 1;
}

void enqueue_task_yat(struct rq *rq, struct task_struct *p, int flags) {
    rq->yat.agent = p;
    printk("=====enqueue_task_yat  %d\n", p->pid);
}

void dequeue_task_yat(struct rq *rq, struct task_struct *p, int flags) {
    rq->yat.agent = NULL;
    printk("=====dequeue_task_yat  %d\n", p->pid);
}
```

然后使用 *qemu* 和 *gdb* 进行调试，即进入 *arch/x86_64/boot* 目录并执行 *qemu-system-x86_64 -kernel bzImage -initrd initrd.img -append nokaslr -m 2G -S -s*，启动 *qemu* 窗口，接着在另一个命令行界面输入 *gdb vmlinux* 启动 *gdb* 并加载符号表，然后输入 *tar remote:1234* 连接本地 *qemu*，输入 *b schedule* 在调度的入口处打断点，或者可以直接在

想要测试的接口处打断点，然后输入 *c* 开始运行，在 *schedule* 函数处断住之后可以通过不断输入 *n* 往下执行，直到看到 *qemu* 窗口输出 *printk* 信息，即说明接口功能测试成功。

5.1.3 测试结果

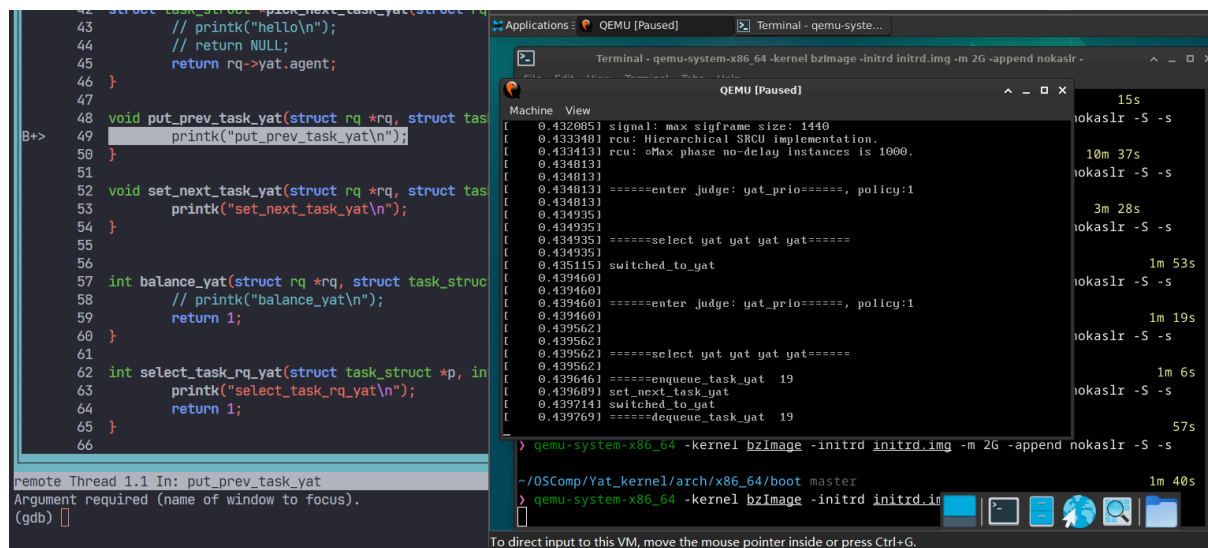


图 5.1 部分接口功能测试

从上图可以看出，在内核启动的调度过程中调度器能进入到 *yat* 调度类中，并且输出了一系列接口名称信息，说明我们的调度接口是有效的。

5.2 多核互斥资源共享协议测试

TODO

5.3 面向资源的优先级排序算法测试

TODO

5.4 考虑阻塞的任务分配机制测试

TODO

5.5 Yat-Scheduler 性能测试

TODO

6 遇到的困难和解决方法

在初期的调研和开发过程中，我们队伍遇到了一些亟待解决的问题和困难：

- 队伍成员涉及保研夏令营等工作的准备，加之**课内课业实验压力较大**，能抽出的开发时间有限。该困难在进入决赛后，等期末考试完放暑假了或许会有所缓解。
- 由于队伍的队员来自不同学院，分别处在中山大学广州校区东校园和珠海校区，队伍内部的**沟通大多数情况下只能通过线上进行**，沟通效率势必会受到一定的影响，如果选择在重要的时间点队伍会进行线下的集中讨论和工作的话就不可避免地会带来差旅和住宿的开销。这对于队员来说也存在着一定的压力和负担。
- 网上**缺少**开发 *Linux* 调度器相关的零基础**教程**，前期无从下手，只能参考一些开源的调度器项目慢慢摸索。

6.1 未来展望

进入全国决赛阶段后，待七八月份暑假空闲时间较多，我们将全面完善调度器的各个模块，并实现多核互斥资源共享协议、面向资源的优先级排序算法、考虑阻塞的任务分配机制这三个主要的核心技术部分，最后进行全面的功能和性能测试。

7 参考资料

- [1] linuxfoundation, 《PREEMPT_RT Official Documentation》. [在线]. 载于: <https://wiki.linuxfoundation.org/realtime/documentation/start>
- [2] B. Brandenburg, 《Linux Testbed for Multiprocessor Scheduling in Real-Time Systems》. [在线]. 载于: <https://www.litmus-rt.org/index.html>
- [3] A. Block, H. Leontyev, B. B. Brandenburg, 和 J. H. Anderson, 《A flexible real-time locking protocol for multiprocessors》, 收入 13th IEEE international conference on embedded and real-time computing systems and applications (RTCSA 2007), 2007, 页 47–56.
- [4] T. P. Baker, 《Stack-based scheduling of realtime processes》, Real-Time Systems, 卷 3, 期 1, 页 67–99, 1991.
- [5] R. I. Davis 和 A. Burns, 《Robust priority assignment for fixed priority real-time systems》, 收入 28th IEEE International Real-Time Systems Symposium (RTSS 2007), 2007, 页 3–14.
- [6] A. Burns 和 A. J. Wellings, 《A schedulability compatible multiprocessor resource sharing protocol–MrsP》, 收入 2013 25th euromicro conference on real-time systems, 2013, 页 282–291.
- [7] L. Sha, R. Rajkumar, 和 J. P. Lehoczky, 《Priority inheritance protocols: An approach to real-time synchronization》, IEEE Transactions on computers, 卷 39, 期 9, 页 1175–1185, 1990.
- [8] A. Wieder 和 B. B. Brandenburg, 《On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks》, 收入 2013 IEEE 34th Real-Time Systems Symposium, 2013, 页 45–56.
- [9] B. B. Brandenburg, 《Scheduling and locking in multiprocessor real-time operating systems》, 2011.
- [10] S. Zhao, J. Garrido, A. Burns, 和 A. Wellings, 《New schedulability analysis for MrsP》, 收入 2017 IEEE 23rd international conference on embedded and real-time computing systems and applications (RTCSA), 2017, 页 1–10.
- [11] S. Zhao, 《A FIFO spin-based resource control framework for symmetric multiprocessing》, 2018.
- [12] S. Zhao 等, 《Priority Assignment on Partitioned Multiprocessor Systems With Shared Resources》, IEEE Transactions on Computers, 卷 70, 期 7, 页 1006–1018, 2021, doi: 10.1109/TC.2020.3000051.

-
- [13] A. Racu 和 L. S. Indrusiak, 《Using genetic algorithms to map hard real-time on noc-based systems》, 收入 *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, 2012, 页 1–8.
- [14] S. Furst 等, 《AUTOSAR–A Worldwide Standard is on the Road》, 收入 *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, 2009.
- [15] B. B. Brandenburg, 《Multiprocessor real-time locking protocols》, *Handbook of Real-Time Computing*. Springer, 页 347–446, 2022 年.
- [16] M. Yang, W.-H. Huang, 和 J.-J. Chen, 《Resource-oriented partitioning for multiprocessor systems with shared resources》, *IEEE Transactions on Computers*, 卷 68, 期 6, 页 882–898, 2018.