# M-RSF：面向Unikernel的一种多级反馈队列任务调度机制

董博南[1,2]，杨秋松[1]，李明树[1]
（1.中国科学院大学，北京 100049；2.中国科学院软件研究所基础软件国家工程研究中心，北京 100190）

**摘　要：**Unikernel作为云计算领域的前沿技术，具有启动速度快和资源占用少的特点。但是，在云环境大规模任务调度场景下，目前Unikernel缺少根据其任务特点所定制的调度机制，这阻碍了Unikernel进一步发挥其性能优势。针对此问题，首先总结了Unikernel的任务特点。在此基础上，提出了一种新的面向Unikernel的多级反馈队列调度机制M-RSF及其数学模型，在不影响Unikernel结构特点的前提下，能够更有效地对云环境下Unikernel的任务进行调度。同时，基于对任务特性的深入分析，提出了一种新的负载模型，该负载模型不仅可以准确刻画Unikernel的任务特点，还可以为M-RSF中调度策略的参数设置较合理的取值。最后，对M-RSF调度机制在OSv Unikernel上进行了实现和验证，实验结果表明，M-RSF可以有效减少OSv Unikernel在大规模任务调度时的平均等待时间和平均周转时间，相比于未经改动的OSv Unikernel，平均等待时间和平均周转时间减少达15%以上。

**关键词：**云计算；Unikernel；调度机制；负载模型
**中图分类号：**TP302
**文献标志码：**A
**DOI：**10.11959/j.issn.1000–436x.2024061

# M-RSF: a multilevel feedback queue task scheduling mechanism for Unikernel

DONG Bonan[1,2], YANG Qiusong[1], LI Mingshu[1]
1. University of Chinese Academy of Sciences, Beijing 100049, China
2. National Engineering Research Center for Fundamental Software, Institute of Software Chinese Academy of Sciences, Beijing 100190, China

**Abstract:** Unikernel, as a cutting-edge technology in the field of cloud computing, is characterized by its fast start-up speed and minimal resource usage. However, in the context of large-scale task scheduling in cloud environments, Unikernel currently lacks a customized scheduling mechanism based on its task characteristics, which hinders the further exploitation of its performance advantages. To address this issue, the distinctive task characteristics of Unikernel were initially summarized. Subsequently, a novel multi-level feedback queue scheduling mechanism for Unikernel, referred to as M-RSF, along with its mathematical model, were proposed. Without affecting the structural features of Unikernel, this mechanism could more effectively schedule Unikernel tasks in cloud environments. Meanwhile, a new load model was put forward based on an in-depth analysis of task characteristics. The model not only accurately depicted the task characteristics of Unikernel but also provided a more reasonable value setting for the scheduling policy parameters in M-RSF. Lastly, the implementation and validation of the M-RSF scheduling mechanism on the OSv Unikernel were conducted. Experimental results indicate that M-RSF can effectively reduce the average waiting time and average turnaround time of the OSv Unikernel during large-scale task scheduling, achieving a reduction of more than 15% compared to the unmodified OSv Unikernel.

**Keywords:** cloud computing, Unikernel, scheduling mechanism, load model

收稿日期：2023–10–27；修回日期：2024–02–19
基金项目：中国科学院战略性先导科技专项基金资助项目（No.XDA-Y01-01, No.XDC02010600）
**Foundation Items:** The Strategic Priority Research Program of Chinese Academy of Sciences (No.XDA-Y01-01, No.XDC02010600)

## 0　引言

虚拟化技术是目前云服务的核心技术之一，传统的虚拟化技术致力于提供一个物理设备的抽象，包括虚拟的 CPU、I/O 设备和内存等。随着人们对虚拟化技术需求的提升，传统虚拟化技术在资源管理和性能不足方面的缺点逐渐显现[1]。虽然容器技术的出现在一定程度上解决了上述问题，但是，由于容器必须与其主机系统共享一个通用的操作系统内核，因此在隔离性和安全性上还存在着一定问题。面对上述情况，一种基于库操作系统构建的 Unikernel 应运而生[2]。与传统操作系统相比，Unikernel 中的应用程序和库操作系统位于同一地址空间，降低了上下文切换的成本，并且 Unikernel 只包含应用程序所必需的库，进而具有更小的资源占用比和更快的启动速度；与容器相比，每个 Unikernel 有自己的内核，比共享内核的容器有更强的隔离性[3]。因此，Unikernel 的上述特点可以和传统虚拟化技术以及容器技术互补，并在云环境中充分发挥性能和空间优势，以提供高效率和高质量的服务。

然而，Unikernel 在云环境下提供性能和空间优势的同时，也为 Unikernel 在云环境下的调度机制带来了诸多挑战。Unikernel 在结构上采用了单一地址空间的设计并且删除了冗余组件，导致其只能支持功能较为简单的应用程序；另一方面，Unikernel 的结构特性使其没有上下文切换的成本，具有了更快的启动速度和运行效率[4]。因此云环境中 Unikernel 的负载具有以下 3 个特点：首先，云计算的主要特点之一就是任务规模庞大，特别是在 Unikernel 所特有的性能优势的保障下，其能够支持规模较为庞大的负载，吞吐量可达 $10^4$~$10^5$ request/s[5]；其次，由于 Unikernel 仅支持功能较为单一的单进程应用，因此其负载多为功能较为简单的短作业任务，同时偶尔有长作业任务出现，例如在 Unikernel 上运行轻量级数据库 Redis 进行某次较大数据读写时，会出现执行时间增加的情况[6]；最后，和传统虚拟化技术在云环境中所面对的具有动态性和多样性的负载不同，Unikernel 的负载之间不存在抢占关系，并且类型几乎一致。

基于上述情况，现有 Unikernel 的研究着重于将特定应用场景与 Unikernel 进行适配整合[7]，其 CPU 调度器要么设计简单，要么使用现有操作系统复杂的调度机制，造成了组件的冗余，违背了 Unikernel 精简的设计思想。MirageOS Unikernel[8] 和 KylinX Unikernel[9] 依赖于 MiniOS 为其提供的包括调度机制在内的内核功能，采用了先来先服务（FCFS, first come first service）的调度策略，但是这种调度策略过于简单，并且不适用于 Unikernel 短作业居多的调度场景。OSv Unikernel[10] 采用了多队列多处理器调度（MQMS, multi-queue multi-processor scheduling）策略，并在多个队列中应用了不同的策略，但是任务在执行时不能在队列中移动，进而导致该机制存在负载不均的问题[11]。X-container[12] 和 UKL[13] 基于 Linux 内核构建，因此其调度机制为 Linux 内核中默认的完全公平调度（CFS, completely fair scheduling）策略，该策略对 Unikernel 来说又过于冗余和复杂。因此，这些调度机制并没有根据 Unikernel 特性而深入设计，进而降低了云环境下 Unikernel 的运行效率。在上述背景下，本文提出了一种新的调度机制 M-RSF 及其数学模型，能够针对云环境下 Unikernel 的任务特点进行有效调度，进而提升 Unikernel 的运行效率。

本文的主要贡献如下。

1) 提出一种面向 Unikernel 的多级反馈队列调度机制 M-RSF，并提炼了该机制对应的数学模型。在不改变 Unikernel 结构特性的前提下，能够对云环境下 Unikernel 的任务进行更为有效的调度。M-RSF 最显著的贡献是在分析云环境下 Unikernel 任务特点的基础上，结合 Unikernel 特性提出了一种新的多级反馈队列算法，设计了符合 Unikernel 调度场景和负载特点的调度机制。

2) 提出一种新的负载模型，该负载模型可以准确地模拟云环境下 Unikernel 的任务特点，并且为 M-RSF 中调度策略的参数设置准确的取值。

3) 在 OSv Unikernel 上实现了 M-RSF，并验证了 M-RSF 的有效性。

## 1　相关工作

目前，越来越多的研发团队加入 Unikernel 的研发中，微软、IBM、NEC 和 Docker 等知名研究机构和 IT 企业都已经为 Unikernel 技术做出了显著贡献，这些研究项目和研究成果广泛应用于云计算、人工智能、物联网等领域[8]。表 1 从运行环境、

特点以及主要调度策略等方面对主流Unikernel相关研发工作进行了总结[2,4,7,9-10]。

调度机制作为Unikernel系统中的核心组成，其作用是决定任务的执行次序以确保整个Unikernel系统能够高效地运行，由此实现资源利用率的最大化。调度机制可以将任务集合分类为工作队列、就绪队列和等待队列。工作队列包含系统中所有任务；就绪队列包含所有运行条件已经具备的任务；等待队列包含因等待或者其他情况而无法继续运行的任务[14]。然后调度机制根据所设计的调度算法从队列中选出待执行的任务。结合表1可以看出，常见的Unikernel调度策略有[15]FCFS、SJF、RR、CFS以及MQMS等，大部分现有Unikernel的调度机制均使用以上调度策略。

FCFS调度策略每次从就绪队列中选择最先进入队列的任务，然后一直运行，直到任务执行完成或被阻塞，才会继续从队列中选择下一个任务接着运行。MirageOS Unikernel[8]和KylinX Unikernel[9]基于MiniOS进行构建并且依赖于MiniOS为其提供的包括FCFS调度策略在内的内核功能。IncludeOS Unikernel[16]同样采用了FCFS的调度策略，并且采用了一种非抢占式的任务管理方法，通过使用主处理器将任务分配给每个应用处理器来实现同时执行多个任务，即每来一个任务分配一个虚拟处理器。这种方法虽然提升了任务的执行效率，但本质上依然是沿用了FCFS的调度策略。FCFS调度策略实现简单，主要缺点是排在长作业后面的短作业需要等待很长时间，对短作业较为不利，因此并不适用于Unikernel短作业居多的场景调度特点。

相比于FCFS侧重于长作业的调度策略，SJF调度策略[15]会优先选择运行时间最短的任务来执行。FlexOS Unikernel[17]采用了SJF调度策略，该策略对短作业较为有利，可以提升整个系统的吞吐率，主要缺点是如果当前队列有非常多的短作业，那么就会使得长作业不断后移，周转时间变长，致使长作业长期不被运行，进而产生饥饿问题。因此同样不适合Unikernel任务规模大并伴有长作业的场景调度特点，因为优先执行的大量短作业任务会进一步加剧长作业任务出现饥饿问题。

与FCFS和SJF侧重长作业和短作业不同，RR调度策略为待执行的任务预先分配了时间片，当前任务使用完时间片后才切换下一个任务运行。HermitCore Unikernel[18]采用了基于优先级的RR策略，该策略相比于FCFS和SJF来说，对长作业和短作业相对公平，主要问题一方面是时间片的长度设置，如果时间片设置太短会导致过多的上下文切换，降低了运行效率，如果设置太长又可能会引起对短作业任务的响应时间变长[19]；另一方面是HermitCore所沿用的基于优先级的调度策略并不适合Unikernel任务之间类型相同、不存在抢占关系的任务特点。

相比于上述较为简单的调度策略，CFS是传统操作系统所使用的公平调度策略。X-containers Unikernel[12]和UKL Unikernel[13]基于Linux内核进行构建，因此所使用的调度机制均为Linux内核中默认的CFS策略，虽然CFS策略更能够保证任务之间的公平性以及应对Unikernel任务规模大的特点，但对于Unikernel来说过于冗余，不符合Unikernel精简的设计要求[20]。另外，云环境下Unikernel的任务特点是功能单一的简单任务，CFS策略对于Unikernel来说过于复杂。

与以上较为简单或过于复杂的策略不同，MQMS是一种多队列调度机制，包含了多个队列，每种队列可以使用不同的调度策略或将多种策略进行结合[21]。OSv Unikernel[10]提出一种多队列调度机制，包含多个调度队列，在不同的队列中应用了

**表1 Unikernel相关研发工作**

| Unikernel | 运行环境 | 特点 | 主要调度策略 |
|---|---|---|---|
| MirageOS | Xen KVM | 采用Ocaml进行开发，性能优异，安全性高 | FCFS调度策略 |
| KylinX | Xen | 基于微型系统重构，隔离性强 | FCFS调度策略 |
| IncludeOS | VirtualBoxKVM | 极小的磁盘空间和内存占用，高效的I/O | FCFS调度策略 |
| FlexOS | KVM | 采用多种安全配置策略，具有较强的安全性 | 最短作业优先（SJF, shortest job first）调度策略 |
| HermitCore | KVM | 重构Linux系统，能直接运行Linux程序 | 轮询（RR, round-robin）调度策略 |
| X-containers | Xen | 重构Linux系统，支持Linux程序 | CFS策略 |
| UKL | KVMXen | 重构Linux系统，支持多种配置 | CFS策略 |
| OSv | XenKVM VirtualBox | 兼容性较强，与Linux程序和多个VMM兼容 | MQMS策略 |

FCFS 和 RR 等策略，并且每个调度相互独立。但每个队列所使用的调度策略不一致，导致任务执行时间存在差异，进而会出现负载不均的问题[11]。虽然 OSv 通过运行负载平衡器在一定程度上缓解了此问题，但在 Unikernel 任务规模大的云环境场景下，仍不能完全避免此问题。

除了上述调度机制外，还有一种多级反馈队列（MLFQ, multi-level feedback queue）调度策略[22-23]也适用于在云环境下进行任务调度，相比于 MQMS 中任务被永久分配给一个队列，MLFQ 中的任务在队列中可移动，更为灵活，并且 MLFQ 综合了诸多调度策略的优点，每个新到达的任务都可以很快得到响应，特别是短作业型任务只用较少的时间就可执行完成。但是，MLFQ 并未在 Unikernel 中应用，一方面由于 MLFQ 存在抢占机制，不符合 Unikernel 任务之间相对平等的特点，需进行改进和优化；另一方面，在 MLFQ 中，目前缺乏根据 Unikerne 任务规模大、短作业居多并伴有长作业的任务特点所设计的调度策略。

综上来看，云环境中 Unikernel 的负载具有以下 3 个特点：任务规模大；功能单一的短作业居多并伴有长作业；任务类型几乎一致，不存在抢占关系。但现有 Unikernel 研究着重于应用场景的拓展和适配，在调度机制方面，并没有根据 Unikernel 特性和上述任务特点进行深入设计。虽然 Unikernel 的应用场景在不断拓展，但在实际生产环境中，如果 Unikernel 缺少高效的调度机制，将会对 Unikernel 的运行效率造成影响，导致其无法发挥性能优势。

## 2　M-RSF

本文从 Unikernel 系统层面提出了一种新的调度机制 M-RSF 及其数学模型，其主要思路是在不影响 Unikernel 单地址空间结构特点的基础上，基于多级反馈队列算法进行深度改进，通过对 Unikernel 任务特点以及调度场景进行分析，并利用本文提出的负载模型对任务进行模拟和优化时间片设置，以使 Unikernel 能够更为有效合理地对云环境下的任务进行调度。

### 2.1　总体设计

M-RSF 的主要思想是基于 MLFQ 和 Unikernel 的架构特性以及任务特点，设计新的调度机制。M-RSF 总体设计如图 1 所示，主要由 2 个部分组成，分别是调度模块和负载模型。其中，调度模块主要包括嵌套队列 Q1、第二级队列 Q2 以及第三级队列 Q3。任务会优先进入嵌套队列 Q1，然后进入 Q2 以及 Q3，直至最终执行完成，退出队列。其中，每个队列根据当前 Unikernel 任务状态，采用不同的调度策略，以达到任务的高效调度。另外，输入模块使用可扩展负载模型，该模型可以利用规则生成器，对 Unkernel 应用场景中的任务进行模拟。

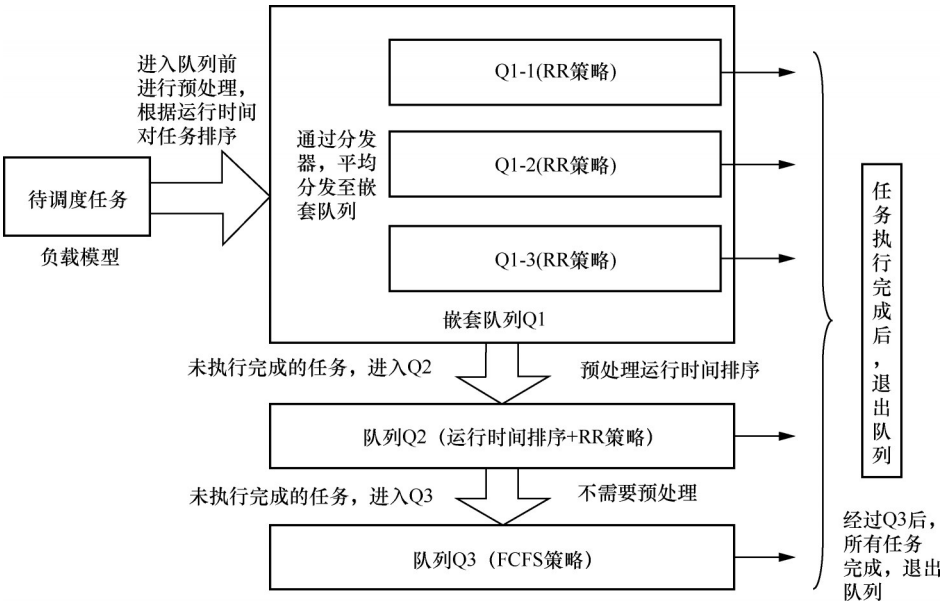


图 1　M-RSF 总体设计

### 2.1.1　M-RSF调度机制

M-RSF是一种针对Unikernel负载特点所设计的调度机制，如图1所示，包括三级队列Q1、Q2、Q3，其中Q1是嵌套队列包含3个子队列，Q1的主要工作流程如图2所示。

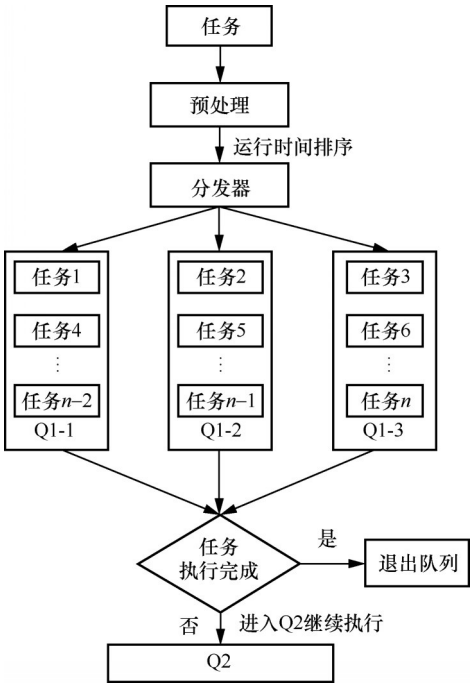


图2　Q1的主要工作流程

1) 首先，对待调度的任务进行预处理，并通过分发器，依次平均分配至Q1-1、Q1-2和Q1-3。与传统Unikernel调度队列不同，本文结合Unikernel任务规模大的特点，在第一级队列Q1中采用了嵌套队列的设计方式，将Q1分为3个子队列，使任务尽快进入执行队列。

2) 在预处理的过程中，根据任务运行时间对任务由小到大依次排序，本文通过使用负载模型，模拟Unikernel任务特点和数据分布的方式，对任务运行时间进行随机预设。在传统操作系统的任务调度中，大多通过任务到达时间进行排序[24-25]，即优先到达队列的任务优先进入队列，此时如果有长作业优先到达，进而优先进入队列，后面大量短作业任务不得不排队进行等待。因此，根据任务到达时间进行排序，对短作业居多的场景十分不友好，会导致总体调度时间和周转时间的增长。所以本文根据Unikernel短作业居多并伴有长作业的任务特点，设计了根据任务运行时间大小的方案进行排序，这样设计的好处是对短作业居多的场景较为有利，即任务运行时间越小的任务越优先进入执行队列。以Q1-1为例，任务根据运行时间排序，先后进入执行队列。与传统操作系统不同，在时间片设置方面，本文提出一种Unikernel负载模型，根据负载模型可以对RR策略中的时间片进行优化设置。在嵌套队列Q1中，利用负载模型设置合理的时间片后，可以使大规模任务尽快执行，避免排队时间过长出现饥饿问题，同时根据所设置的时间片，还可以保证部分短作业任务在队列Q1中得以执行完成。最终，在队列Q1中执行完成的任务，退出队列，未执行完成的任务，则等待进入队列Q2。

3) Q2的主要工作流程如图3所示。对Q1-1、Q1-2、Q1-3中未执行完成的任务再次进行预处理，此处的预处理操作和先前预处理类似，主要是对剩余任务的运行时间进行排序，并且依然采用由小到大的排序方式，即短作业优先进入队列执行，以有利于Unikernel中短作业居多的场景。进入队列Q2后，依然根据负载模型对RR策略中的时间片进行合理设置，进一步保证嵌套队列Q1中未执行完成的短作业任务能够在Q2中执行完成。通过Q1、Q2这两级队列，在RR策略所分配时间片以内的短作业任务都得以执行完成，并且Unikernel中长作业任务在Q1和Q2中也得到了执行，避免了饥饿问题。最终，未执行完成的少量长作业任务，进入队列Q3。

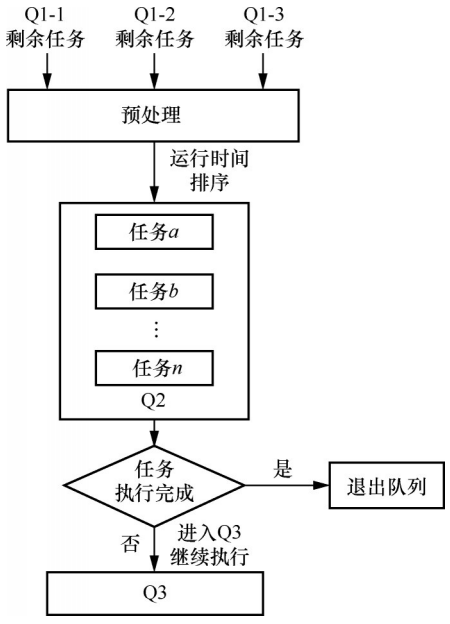


图3　Q2的主要工作流程

4) Q3的主要工作流程如图4所示。剩余长作业任务进入Q3前，因为大部分短作业任务都已执行完成，所以不再需要进行预处理。在Q3队列中，不再使用RR和最短时间优先的组合调度策略，而是使用FCFS的调度策略，FCFS适用于长作业场景，有利于Unikernel在短作业任务已经执行完成的情况下，执行余下的长作业任务。最终所有剩余的长作业任务在Q3队列中执行完成。另外，在M-RSF中并没有设计抢占机制，因为经过对Unikernel任务特点的分析，发现任务之间类型几乎一致，所以在调度机制设计时取消了抢占，以确保算法的简洁，进而保证了Unikernel的精简性原则。

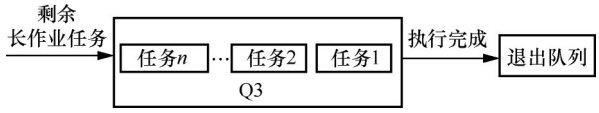


图4　Q3的主要工作流程

综上，相比于传统多级反馈队列算法，M-RSF调度机制根据Unikernel的任务特点进行了专有定制。首先，M-RSF调度机制在队列Q1中并未采用普通队列，而是根据Unikernel任务规模大的特点设计了嵌套队列，相比于传统多级反馈队列算法中使用的普通队列，嵌套队列可以使大量任务在到达队列后得以尽快执行，减少了任务的等待时间；其次，在M-RSF调度机制的队列Q1和队列Q2中，根据Unikernel任务短作业居多并伴有长作业的特点，设计了按任务运行时间排序的执行规则，相比于传统多级反馈队列中按照到达时间排序，根据任务运行时间排序更有利于短作业居多的场景。另外，根据负载模型，在队列Q1和队列Q2中，对RR策略中的时间片进行了优化设置，确保短作业任务都可以在前两级队列中完成，此处并未采用传统多级反馈队列中动态时间片的设计方法，因为动态时间片常用于传统操作系统以及云环境中动态和多样的复杂任务[26]，对Unikernel场景下功能和类型较为单一的任务并不适用；再次，在M-RSF调度机制中，根据Unikernel任务类型几乎一致的特点，取消了抢占设计，相比于传统多级反馈队列算法包含的抢占机制，进一步对算法进行了精简；最后，在M-RSF调度机制的队列Q3中，由于短作业任务已经在Q1、Q2中执行完成，不再需要对剩余的长作业任务进行预处理等操作，因此根据Unikernel伴有少数长作业的任务特点，在Q3中设计了FCFS策略，以保证Unikernel中的长作业任务在最后一级队列中顺利执行完成。

### 2.1.2　M-RSF算法复杂性分析

算法复杂性的度量与计算时间和存储空间有关，主要包括时间复杂度和空间复杂度这两类度量。由于空间复杂度计算相对简单，并且无法反映算法执行速度或响应时间[27]，因此本节主要从更能表征算法运行时间与输入数据规模的时间复杂度来分析M-RSF算法。在程序实际运行中，算法的执行包括加减乘除和排序等操作。其中，排序主要包括交换和比较的运算，而运算次数主要与使用何种排序算法以及数列有关。为便于评估算法，可通过运行时间$t(s)$来衡量算法的时间复杂度[28]，如式(1)所示。

$$t(s)=c_{\mathrm{a}}A(s)+c_{\mathrm{s}}S(s)+c_{\mathrm{m}}M(s)+c_{\mathrm{d}}D(s)+c_{\mathrm{c}}C(s)+c_{\mathrm{e}}E(s) \tag{1}$$

其中，$c_{\mathrm{a}}$、$c_{\mathrm{s}}$、$c_{\mathrm{m}}$、$c_{\mathrm{d}}$、$c_{\mathrm{c}}$和$c_{\mathrm{e}}$分别代表一次加法、减法、乘法、除法、比较和交换运算所需的时间；函数$A(\cdot)$、$S(\cdot)$、$M(\cdot)$、$D(\cdot)$、$C(\cdot)$和$E(\cdot)$分别为算法中加法、减法、乘法、除法、比较和交换操作所对应的次数。

根据式(1)，对M-RSF的时间复杂度进行计算。首先，在进入队列Q1之前，M-RSF根据任务的运行时间，对任务进行了排序的预处理操作，M-RSF使用了快速排序的方法，因此预处理操作的时间复杂度为$O(n\log n)$。预处理后，任务分别进入嵌套队列Q1-1、Q1-2、Q1-3中并行执行，在嵌套队列中设计了时间片轮询策略，因此Q1队列中的时间复杂度为$O\left(\frac{n}{3}\right)$。进而可以得出M-RSF中经过队列Q1后的时间复杂度为

$$t(\mathrm{Q1})=O\left(n\log n+\frac{n}{3}\right) \tag{2}$$

接下来，未完成的剩余任务进入队列Q2，在进入Q2前，同样根据任务的剩余完成时间进行了排序，此处与Q1的预处理采用了同样的排序算法。因此，进入队列Q2前的预处理操作的时间复杂度为$O(\phi n\log\phi n)$，其中，$\phi$为到达Q2时剩余任务的百分比。进入队列Q2后，同样设计了时间片轮询的策略，因此队列Q2中的时间复杂度为$O(\phi n)$，进而可以得出M-RSF中队列Q2的时间复杂度为

$$t(\mathrm{Q2})=O(\phi n\log\phi n+\phi n) \tag{3}$$

最后，未完成的任务不用经过预处理操作，直接进入队列Q3，在队列Q3中设计了先来先服务的策略，因此Q3处的时间复杂度为

$$t(\mathrm{Q3}) = O(\psi n) \tag{4}$$

其中，$\psi$表示到达Q3时剩余任务的百分比。因此根据式(1)，可以得到M-RSF的时间复杂度为

$$t(\mathrm{M\text{-}RSF}) = O\left(n\log n + \frac{n}{3} + \phi n\log\phi n + \phi n + \psi n\right) \tag{5}$$

如式(5)所示，云环境下Unikernel的任务多为短作业，因此大多数的任务均在队列Q1和Q2中完成执行，所以$O(\psi n)$可忽略。另外，$O\left(\frac{n}{3}\right)$和$O(\phi n)$相对于$O(n\log n)$影响较小，也可隐去。进而可以将式(5)化简为

$$t(\mathrm{M\text{-}RSF}) = O(n\log n + \phi n\log\phi n) \tag{6}$$

在此基础上，对OSv原始的调度策略进行时间复杂度的分析。由于OSv基于较为复杂的freeBSD内核进行重构[29]，因此，在每级队列中采用了较为复杂的抢占式优先级以及公平调度策略，时间复杂度为$O(n\log n)$和$O(n^2)$。因此，参照M-RSF的时间复杂度方法，可以得出OSv的时间复杂度为

$$t(\mathrm{OSv}) = O\left\{\varphi(n\log\varphi n + n^2) + (1-\varphi)\cdot \left[n\log(1-\varphi)n + n^2\right]\right\} \tag{7}$$

其中，$\varphi$代表各级队列中任务数的百分比。通过对比式(6)和式(7)可得，M-RSF的时间复杂度明显小于OSv中的原始调度策略的时间复杂度，这也进一步证明了M-RSF的有效性。

### 2.1.3 M-RSF数学模型

排队论是研究系统随机聚散现象和随机服务工作过程的数学理论和方法，又称随机服务系统理论[30]。本节基于排队模型，提出一种用于计算M-RSF调度机制中任务执行时延的数学模型。通过数学模型，可以对待执行任务的延迟起到一定的预防和抑制作用。

一个完整的排队模型[31]应主要由$X$、$Y$、$Z$、$A$、$B$表示，其中，$X$代表顾客到达间隔时间、$Y$代表服务时间、$Z$代表服务台数目、$A$和$B$分别代表系统容量和顾客源数量。根据已有研究，上述要素$X$、$Y$、$Z$、$A$、$B$已具有多种表达方式和表达类型[32]，为了便于通过排队论模型求得任务执行时延，需要根据M-RSF算法的特性确定与排队模型$X$、$Y$、$Z$、$A$、$B$的对应关系。

首先，对于时间间隔$X$和服务时间$Y$的表达类型，文献[33]指出在云环境调度系统中，各任务相继到达队列的时间间隔$X$和队列执行时间$Y$均服从指数分布，通常用$M$表示；其次，对于服务台数目$Z$的表达类型，考虑到M-RSF由多级队列组成，其中Q1是嵌套队列，包含3个子队列，而队列Q2和Q3是串联队列，因此可以看作一个整体，所以参考排队论模型，此处服务台数目，即接收队列数目应设置为3。对于系统容量$A$的表达类型，M-RSF可以用非零自然数$N$表示；最后，对于顾客源数目$B$，M-RSF算法在调度时对应大规模任务，因此任务数目设置为无穷多个。综上，排队论模型与M-RSF的对应关系如表2所示。

**表2 排队论模型与M-RSF的对应关系**

| 排队论通用数学模型 | M-RSF算法 |
|---|---|
| $X$ | $M$ |
| $Y$ | $M$ |
| $Z$ | 3 |
| $A$ | $N$ |
| $B$ | $\infty$ |

根据表2的对应关系，参考排队论模型，可得出M-RSF有$k$个任务的概率分布方程，如式(8)所示。

$$\pi_k = \frac{n^n\rho^k}{n!}\left[\sum_{k=0}^{n-1}\frac{(n\rho)^k}{k!} + \frac{(n\rho)^n}{n!(1-\rho)}\right]^{-1} \tag{8}$$

其中，$\pi_k$表示R-MSF算法中包含$k$个任务的概率；$\rho=\frac{\sigma}{\mu}$，$\sigma$代表时间间隔下任务的完成数，$\mu$代表同一时间间隔下任务的到达数；$n$是接收端数量。根据表2中的对应关系，此处$n$=3。在此基础上，将表2中对应关系的取值代入式(8)，可进一步将$k$个任务概率的分布方程化简为

$$\pi_k = \frac{3(1-\rho)(\rho)^k}{1-(\rho)^{N+1}} \tag{9}$$

在计算出M-RSF有$k$个任务的概率后，可利用统计学方法，计算M-RSF中任务等待队长$L$的期望E($L$)，如式(10)所示。

$$\mathrm{E}(L) = \sum_{n=1}^{N} k\pi_{k+1} = \frac{3\rho^2}{1-\rho} - \frac{(N+1)\rho^{N+1}}{1-\rho^{N+1}} \tag{10}$$

最后，根据排队论中排队等待时间等于队长的期望与系统通过率的比值的结论[34]，可得任务的等待时间，即任务时延$T$为

$$T=\frac{E(L)}{\rho}=\frac{\dfrac{3\rho^2}{1-\rho}-\dfrac{(N+1)\rho^{N+1}}{1-\rho^{N+1}}}{\rho}=\frac{3\rho}{1-\rho}-\frac{(N+1)\rho^{N}}{1-\rho^{N+1}} \tag{11}$$

综上，式(11)为M-RSF调度机制中对应的任务执行时延的数学模型，该数学模型可以对任务执行时延进行较为准确的预测，进而可以通过调整相关参数，对待执行任务的延迟起到一定预防和抑制作用。

### 2.1.4　负载模型

目前，关于任务调度模拟的研究主要集中在对任务运行时间进行赋值[35]领域，共有3种赋值方法。1)预测法。文献[36]提出一种对偶单纯形的数学方法对任务运行时间进行预测。基于类似的数学方法，文献[37]提出一种利用机器学习以及人工神经网络的方法对大量任务的运行时间进行规模较大的预测。预测法的主要问题是具有较大的不确定性和一定误差。2)固定赋值法。文献[38-39]均采用经验值，在实验场景中对任务的运行时间进行赋值。固定赋值法的主要问题是灵活性和适用性较差，对调度机制的有效性缺乏说服力。3)随机赋值法。文献[40]采用随机赋值的方法对任务进行赋值。随机赋值法的主要问题是缺乏对任务特点进行模拟，导致无法准确评估调度机制的有效性。结合上述3种方法的特点，本节提出一种用于模拟Unikernel任务特点的负载模型，该负载模型可以更准确地模拟Unikernel的任务特点，并且还可以为M-RSF中调度策略的参数设置较为合理的取值，一方面解决了随机赋值法缺乏对任务特点进行模拟的根本问题；另一方面也避免了预测法的不确定性问题以及固定赋值法的适用性问题。

负载模型主要是根据Unikernel的实际调度场景，对大规模的Unikernel任务进行模拟。在统计学中，逆变换法可用于随机数生成，也常用于随机变量的模拟。本文基于逆变换法提出一种用于模拟Unikernel任务特点的负载模型，并根据该负载模型，对多级队列中RR策略的时间片进行优化设置。

在负载模型中，借助特定的函数分布来随机分配任务运行时间的值以对Unikernel任务进行模拟，同时短作业居多并伴有长作业是Unikernel任务最主要的特点，则可以用任务运行时间表示作业长短，即当横坐标任务运行时间越小时，纵坐标随机数的生成频度应该越高。因此，假设要模拟一个随机变量运行时间$x$，可以设计为遵循均值$\lambda$的指数分布，该分布下的概率密度函数（PDF, probability density function）如式(12)所示。

$$f(x|\lambda)=\frac{1}{\lambda}\mathrm{e}^{-\frac{x}{\lambda}},x\geqslant 0,\lambda>0 \tag{12}$$

计算式(12)的期望，如式(13)所示。

$$\mathrm{E}(x)=\int_0^{\infty}xf(x)\mathrm{d}x=\int_0^{\infty}\frac{x}{\lambda}\mathrm{e}^{-\frac{x}{\lambda}}\mathrm{d}x=\lambda\int_0^{\infty}\frac{x}{\lambda}\mathrm{e}^{-\frac{x}{\lambda}}\mathrm{d}\left(\frac{x}{\lambda}\right)=\lambda\int_0^{\infty}t\mathrm{e}^{-t}\mathrm{d}t=\lambda \tag{13}$$

对式(12)取积分，可得出式(12)的累积分布函数（CDF, cumulative distribution function），如式(14)所示。

$$F(x)=1-\mathrm{e}^{-\frac{x}{\lambda}} \tag{14}$$

根据逆变换法对式(14)取逆，可写出逆CDF，如式(15)所示。

$$F^{-1}(u)=-\lambda\log(1-u) \tag{15}$$

式(15)是累积分布函数的逆函数，为负载模型，即当$u$在（0,1）分布且$\lambda$为正数时，对应的横坐标为任务运行时间。负载模型生成的任务运行时间随机数分布直方图如图5所示。其中，横坐标代表任务运行时间，纵坐标代表频度，且任务运行时间越大，频度越低。根据图5可以看出，任务运行时间大多集中在较小的数值内，与Unikernel的任务特点表现一致。

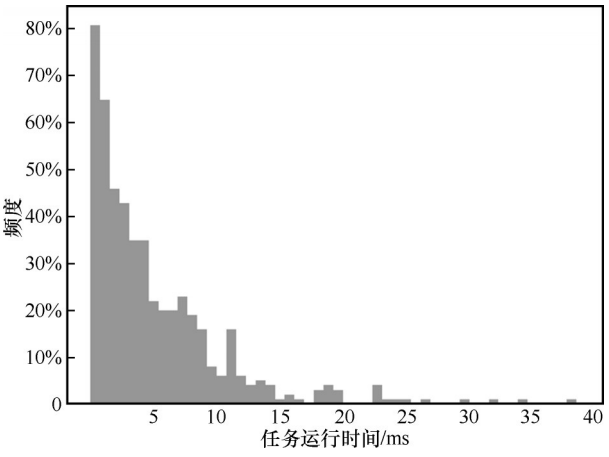


图5　负载模型生成的任务运行时间随机数分布直方图

另外，根据式(13)可以得出任务运行时间的期望为$\lambda$，由式(13)可以看出，该期望值与随机生成的任务数量无关，式(13)代表当前模型下所有可能

取值的平均结果和取值均衡点，即任务运行时间的加权平均值，因此在RR调度策略中，可将时间片长度设置为$\lambda$，则RR的取值为任务运行时间的加权平均值，进而避免时间片设置过长或过短所引起的运行效率降低的问题。因此在该取值均衡点下，令$X$=2$\lambda$，其中前两级队列中的执行时间共为2$\lambda$，由此根据式(14)可以计算出总任务的86.5%可以在前两级队列中完成，该完成率符合Unikernel短作业居多的任务特点。因此，将RR策略中的时间片长度设置为负载模型中的$\lambda$，相较于经验取值和随机取值，更有利于提升M-RSF调度机制的工作效率。

需要特别说明的是，对于云计算的其他场景，可以根据任务特点以及所对应负载模型的不同，对时间片的取值进行相应的计算。例如，在大数据分析场景下，数据具有完全随机性，即长作业和短作业随机出现，因此可以设计为遵循均匀分布的函数，进而可以基于本节的逆变换法利用均匀分布$X$~$U$($a$,$b$)生成负载模型，此分布下的负载模型如图6所示，其中，$m$和$n$表示频度。

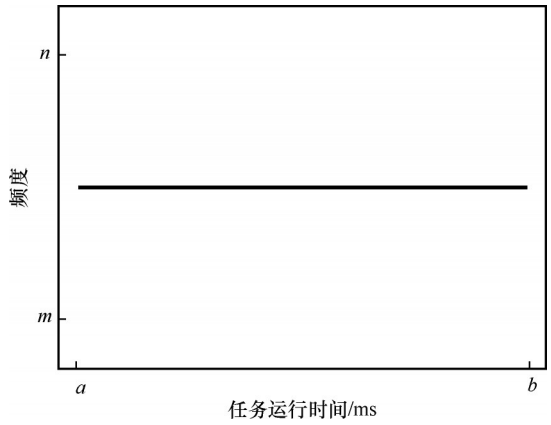


图6　均匀分布下负载模型

由图6可知，在($a$,$b$)区间内，生成任意作业的概率一致，同时为保证M-RSF机制仍能在此场景下适用，$a$和$b$的差值不宜过大。因此，根据本节提出的方法，均匀分布下随机变量$x$的概率密度函数为

$$f(x)=\frac{1}{b-a},a<x<b \tag{16}$$

根据式(16)，可以计算出此概率密度函数的期望为E($x$)=0.5($a$+$b$)，因此，针对云环境下的大数据分析场景，执行队列中的时间片应设置为$t$=0.5($a$+$b$)。综上，在设计负载模型时，应对场景下的任务特点进行准确把握，通过对任务特点的分析，构建负载模型，并依据负载模型，计算不同场景下的时间片取值。

### 2.2　实现

OSv Unikernel采用单地址空间设计，可以直接在虚拟机管理程序上运行，相比于其他Unikernel，OSv最显著的特点是能够支持现有的Linux程序。此外，OSv还可以通过提供可编译链接的语言运行时（如JVM），支持Java等语言编写的程序。OSv架构如图7所示。

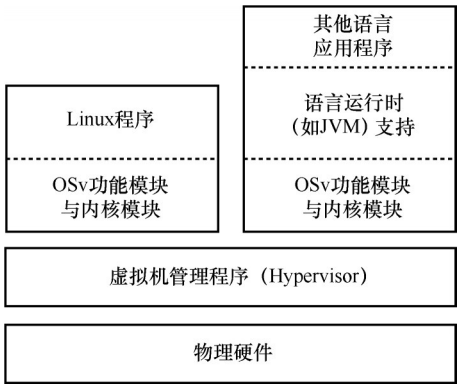


图7　OSv架构

以OSv为基础，对M-RSF进行设计和实现的好处主要在于[10]以下几个方面。

1) 使用C++语言开发，运行时支持多种语言，能够满足云计算场景下运行不同编程语言应用的需求。

2) OSv可以直接在标准的虚拟机管理程序上运行，方便在不修改外部依赖平台的情况下提供新的系统特性，通用性比较强。

3) OSv系统在Unikernel诸多系统设计中生态比较完善，研究者比较活跃，具有代表性。

4) OSv采用了多队列的调度机制，相比于其他Unikernel，调度效果更好。因此，基于OSv进行算法修改和实现，可以更好地检验M-RSF机制的有效性。

在OSv上实现M-RSF机制时，主要工作是将OSv的调度策略修改为按照M-RSF机制来选择下一个待执行的任务。经分析，这其中的关键函数由OSv中CPU类的reschedule函数触发，因此主要工作是对类成员函数reschedule进行修改，并将其实现为本文提出的多级反馈队列机制M-RSF，其中，OSv的reschedule函数的工作流程如图8所示。

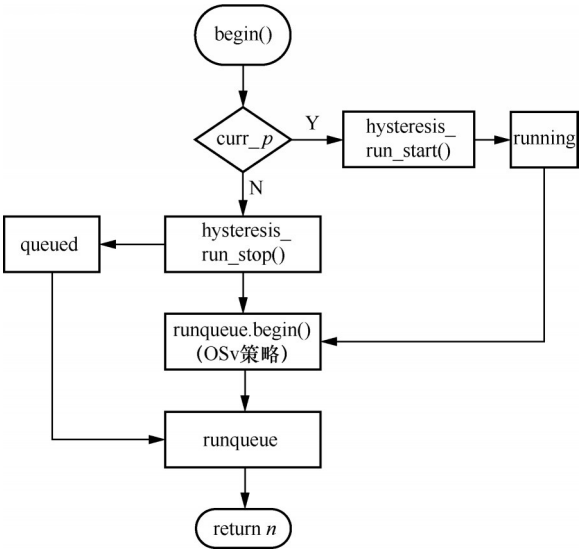


图8　OSv的reschedule函数的工作流程

由图8可知，OSv实际仅包含一个CPU类，在该CPU类的reschedule函数中，首先判断curr_$p$是否满足运行条件。如果满足，则调用hysteresis_run_start()方法，更新该线程的活跃度，并继续保持running状态，直至运行完成，最后调用runqueue.begin()方法，采用该函中的OSv策略，从runqueue即就绪队列中选择下一个运行的线程$n$，最后返回；如果不满足，则调用hystereis_run_stop()方法，转入queued状态，并进入就绪队列runqueue等待再次被调度。然后，当curr_$p$不再运行时，reschedule函数会调用runqueue.begin()方法，从runqueue即就绪队列中选择下个运行的线程$n$，最后返回。

根据上述对OSv中reshedule工作流程的分析，本文主要从reshedule函数中的runqueue.begin()方法入手，将其中的OSv策略修改为本文提出的M-RSF策略。由于OSv中只含有一种CPU类，而本文提出的M-RSF含有3个CPU类，这3个CPU类协同进行工作，因此修改后的工作流程如图9~图11所示。

如图9所示，在本文提出的M-RSF机制的CPU类1中，对runqueue.begin()进行修改。以实例Q1-1为例，首先线程通过pre_deal()函数进行由小到大的排序，并进入就绪队列runqueue(Q1-1)，然后利用sched_RR()函数为线程分配时间片，此处时间片设为$\lambda$。在此时间片下，若线程执行完成，则退出队列，并从runqueue(Q1-1)中返回下一个待执行的线程；若在此时间片下线程未执行完成，则进入队列Q2，同时，处于queued状态的线程也一并进入Q2等待处理。

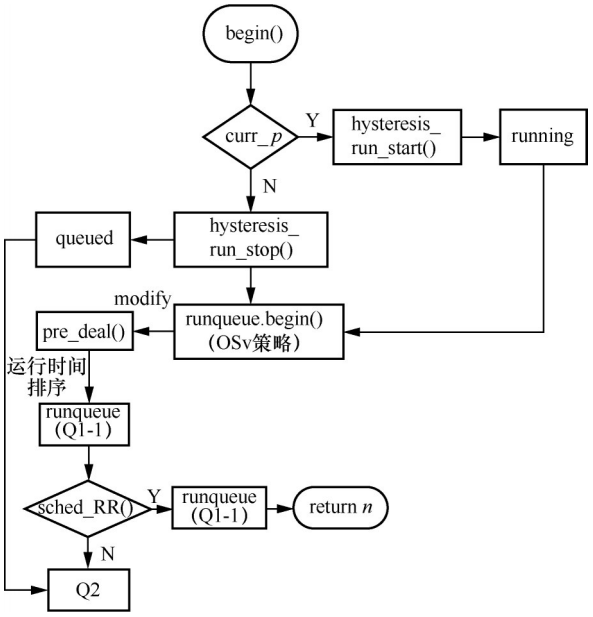


图9　M-RSF的CPU类1（Q1-1）

如图10所示，在M-RSF机制的CPU类2中，工作原理和CPU类1相似，主要区别一方面是CPU类1的实例是嵌套队列Q1-$X$，而CPU类2的实例是普通队列Q2；另一方面是就绪队列为runqueue(Q2)，当线程执行完成时，直接从就绪队列runqueue(Q2)中选出下一个待执行的线程并返回，如果线程未执行完成，则进入队列Q3。

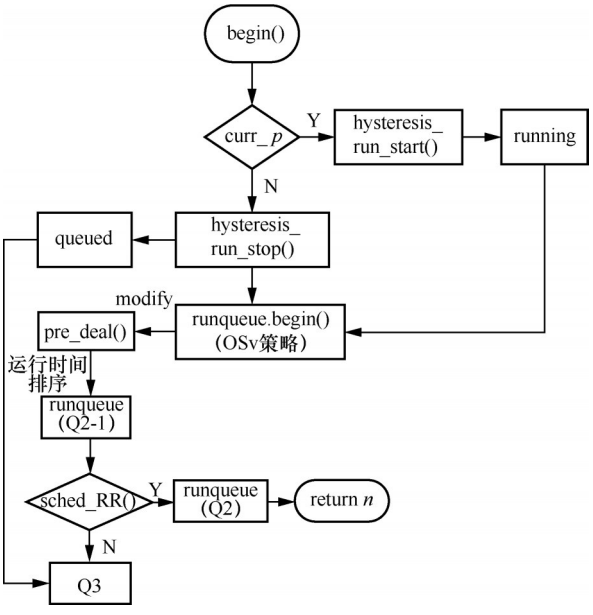


图10　M-RSF的CPU类2（Q2）

如图11所示，在CPU类3的实例Q3中，CPU类2的实例Q2下未完成执行的线程以及处于queue状态的线程进入队列Q3，采用先来先服务原则，保证所有线程顺利执行完成。至此，结束整个工作流程。

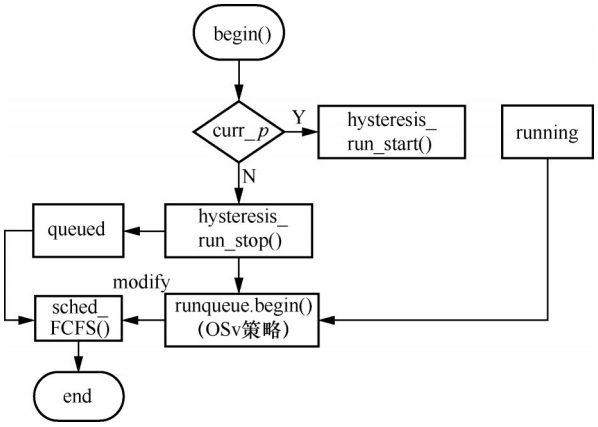


图11　M-RSF的CPU类3（Q3）

综上，对于OSv调度策略的扩展可通过修改runqueue.begin()方法，并将其替换为本文提出的M-RSF机制。在实际的实现过程中，M-RSF机制通过3个CPU类以及5个队列实例的相互协作，完成了对任务的高效调度，相互协作示意如图12所示，其中箭头表示任务流向。

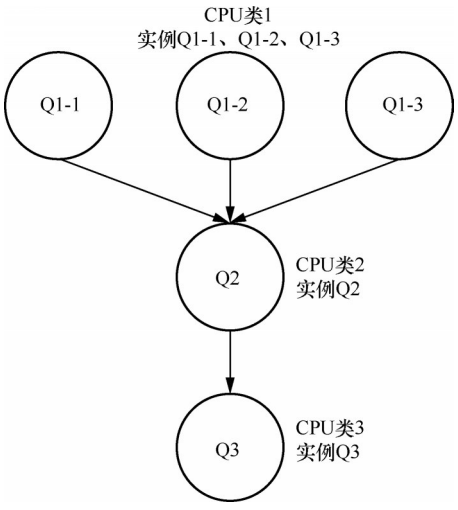


图12　相互协作示意

## 3　实验

本文在Intel平台上运行Linux内核版本为5.4.0的Ubuntu20.04.6操作系统上进行了实验，具体配置信息如表3所示。

表3　配置信息

| 系统资源 | 详细信息 |
|---|---|
| CPU | Intel Core™ i7-10700 |
| 操作系统 | Ubuntu20.04.6 |
| 内核版本 | 5.4.0 |
| 内存 | 64 GB |
| 硬盘 | 1 TB |

### 3.1　实验设计

首先对负载模型的有效性，即负载模型输入值$\lambda$和RR时间片取值的关系进行实验验证。在此基础上，利用负载模型，在同样的负载条件下，使用本文提出的M-RSF机制与其他常见的调度机制从执行时间的角度进行对比，以验证M-RSF算法的执行效率。然后，在云环境中随着任务数量的增多和计算复杂性的增加，最有可能出现的情况就是内存资源紧张。因此，采用内存开销作为指标，对比M-RSF算法和其他常见算法的内存开销，以验证算法对资源的使用情况。接下来，在Unikernel上对未经修改的MQMS OSv机制和使用M-RSF机制后的M-RSF OSv进行对比实验，以验证M-RSF在Unikernel上的有效性，其中，MQMS为多队列调度机制，是OSv目前采用的调度机制。最后，结合实际场景，对未经改动的OSv和M-RSF OSv以及MirageOS Unikernel的调度机制进行对比，以评估M-RSF OSv在实际场景下与未经改动的OSv和其他Unikernel（MirageOS）的实际调度效果。

**实验1**　负载模型的有效性。根据2.1.4节的Unikernel负载模型，分别令RR的时间片长度为$0.5\lambda$、$\lambda$、$1.5\lambda$、$2.0\lambda$，并进行3组实验，借助负载模型，设置输入的任务数量分别为500、1 000和2 000以及该任务数量下对应生成的任务运行时间随机值，输出结果为平均等待时间（WT, waiting time）和平均周转时间（TAT, turn-around time）。其中，等待时间是指任务等待运行所花费的时间，周转时间是指任务从等待到执行完成所花费的总时间。在系统调度实验中，通常使用平均等待时间和平均周转时间作为调度策略的衡量标准。因此，本文采用平均等待时间和平均周转时间作为输出进行对比。

**实验2**　M-RSF和其他调度机制的执行效率。在实验1中，得到合理的RR的时间片长度与$\lambda$的对应取值关系后，进一步对比M-RSF和其他常见调度机制对任务的调度能力，以验证M-RSF机制及

其相关算法本身的有效性。与实验1类似，借助负载模型作为输入，设置的任务输入数量分别为500、1 000和2 000以及该任务数量下对应生成的任务运行时间随机值，分别对比M-RSF和FCFS、SJF、RR、MLFQ以及CFS的平均等待时间和平均周转时间。其中，由于OSv Unikernel使用了MQMS策略，因此，本实验对M-RSF调度机制和前文提到的MQMS策略暂不做讨论，而是在实验4中结合OSv Unikernel进行统一对比。

**实验3**　M-RSF和其他算法的内存开销。在本实验中，借助负载模型作为输入，由于是计算内存开销，因此直接设置任务数量为2 000，连续重复执行50次取均值，分别对比M-RSF和FCFS、SJF、RR、MQMS以及CFS的内存开销。在具体实施时，借助Linux top工具查看当前算法对应进程下的内存占用率作为计量方式，对比M-RSF与其他常见算法的内存开销，以验证算法对资源的使用情况。

**实验4**　MQMS OSv和M-RSF OSv。在本实验中，主要是验证M-RSF在Unikernel上的有效性。同样使用负载模型作为输入，其中根据负载模型输入的任务数量分别为500、1 000和2 000以及对应生成的任务运行时间随机值，分别使用OSv当前的调度机制MQMS OSv和本文提出的M-RSF OSv对上述任务进行调度执行，输出为2种机制下任务调度执行时所对应的平均等待时间和平均周转时间。

**实验5**　MQMS OSv和M-RSF OSv以及MirageOS Unikernel。在本实验中，通过在不同的Unikernel上构建UDP来测试吞吐量，选取UDP主要有以下2个原因：一方面，UDP提供了一种简单快速的数据传输方式，进而可以通过建立UDP连接，进行大量的消息收发，便于对不同Unikernel的调度策略进行评估；另一方面，选取UDP是因为其大多数请求通常可以被视为短作业，并且在生命周期内可以发送和接收多个数据包，因此UDP非常适合运行在Unikernel中，这样UDP请求可以被快速的接收和处理并返回。实验主要步骤以及设置如下。①在MQMS OSv、M-RSF OSv以及MirageOS Unikernel上构建UDP。选取MirageOS Unikernel的主要原因一方面是MirageOS主要应用于云计算场景，适合与M-RSF OSv进行对比；另一方面，MirageOS开源且代码完善，并且和OSv一样，均提供了构建UDP服务的依赖库，方便在Unikernel上运行UDP。②给Unikernel添加网络支持，配置网络接口等，并使其监听指定的端口，接收并回复客户端的消息。③在另一台处于同一网络的机器上运行UDP客户端，将消息发送到Unikernel的UDP服务器，然后接收并显示来自服务器的回复。④利用Wireshark工具获取吞吐率。

### 3.2　实验结果与分析

**实验1**　负载模型的有效性

不同任务数量下M-RSF机制的平均等待时间和平均周转时间如图13~图15所示。

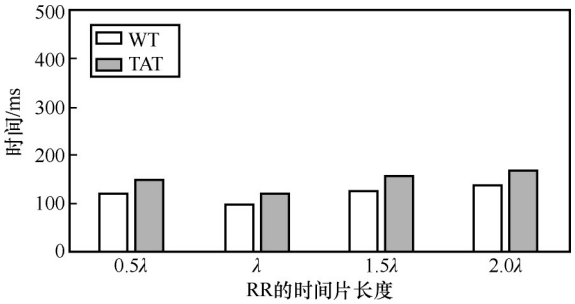


图13　任务数量为500时M-RSF机制的平均等待时间和平均周转时间

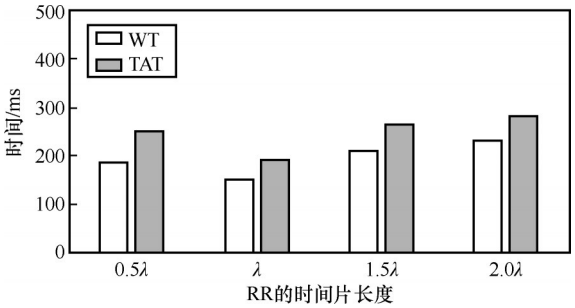


图14　任务数量为1 000时M-RSF机制的平均等待时间和平均周转时间

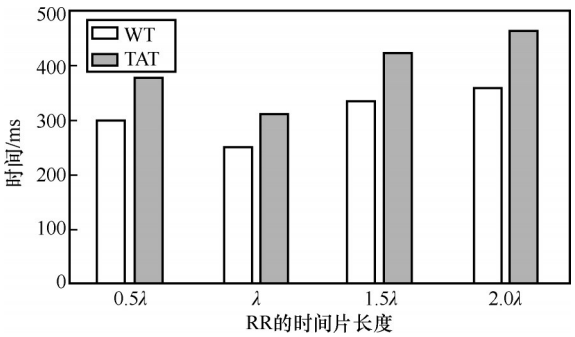


图15　任务数量为2 000时M-RSF机制的平均等待时间和平均周转时间

由图13可以看出，在任务数量为500时，当把M-RSF机制中的Q1和Q2中RR策略的时间片长度设置为取值均衡点$\lambda$时，平均等待时间WT和平均

周转时间TAT均为最小。由此可见，在$\lambda$时具有更好的调度效果。另外，当时间片长度设置为0.5$\lambda$时，平均等待时间和平均周转时间要小于1.5$\lambda$和2.0$\lambda$，这是因为Unikernel的任务多为短作业，当时间片长度设置为0.5$\lambda$时，在Q1和Q2中，也能保证大部分短作业任务可以完成，而当时间片长度设置为2$\lambda$时，时间片长度设置过长，导致任务的等待时间延长，从而在图13中表现最差。由图14和图15同样可以看出，当时间片长度设置为$\lambda$，即负载模型的取值均衡点时，M-RSF调度效果最好。另外，由图13~图15还可以看出，当时间片长度设置为负载模型的任务均衡点$\lambda$时，所表现出的调度优势随着任务数量的增加而增强。

综上，由图13~图15可以看出，当时间片长度设置为$\lambda$，即将Q1和Q2的RR策略中的时间片长度设置为Unikernel负载模型的取值均衡点时，平均等待时间和平均周转时间在任意数量的任务数下均为最短。因此，其调度表现在该取值下效果最好。

**实验2**　M-RSF和其他调度机制执行效率

不同任务数量下M-RSF和其他调度机制的平均等待时间和平均周转时间如图16~图18所示。

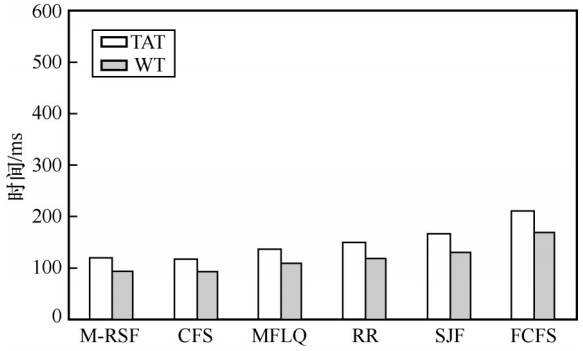


图16　任务数量为500时M-RSF和其他调度机制的平均等待时间和平均周转时间

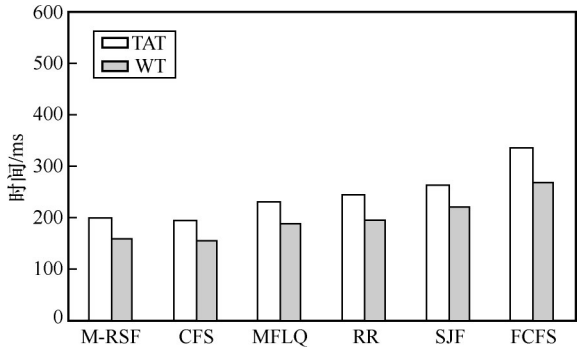


图17　任务数量为1 000时M-RSF和其他调度机制的平均等待时间和平均周转时间

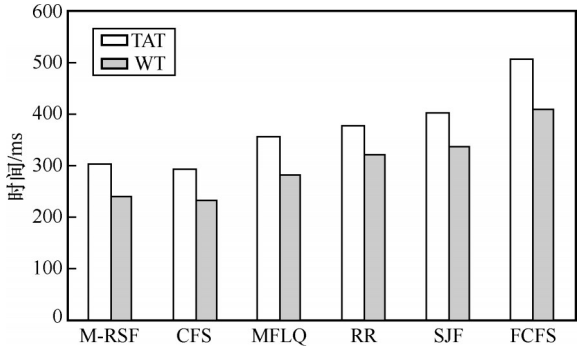


图18　任务数量为2 000时M-RSF和其他调度机制的平均等待时间和平均周转时间

从图18可以看出，平均等待时间和平均周转时间最小的是M-RSF机制和CFS机制。也就是说，调度效果最好的是M-RSF机制和CFS机制。其中，从数据上来看，CFS机制略微好于M-RSF机制，但差距极其细微，并不明显。这是因为CFS是传统操作系统所使用的公平调度策略，调度能力较强。但是CFS机制对于Unikernel来说过于复杂，不符合Unikernel的精简设计原则。同时，从图18还可以看出，除了复杂的CFS机制外，M-RSF机制在平均等待时间和平均周转时间上均小于其他机制。其中，与M-RSF调度机制表现差距最大的是FCFS调度机制，这是因为FCFS机制对短作业较为不利，这也是其在Unikernel短作业居多的任务特点下调度效果较差的主要原因。其次是SJF机制和RR机制，这2种机制相比于FCFS，虽然调度表现要好一些，但仍不能满足Unikernel任务特点的要求。与M-RSF机制表现相对较为接近的是MFLQ机制，在本实验中，M-RSF表现优于MFLQ，一方面是因为M-RSF设计了嵌套队列，以应对云环境下的大规模任务，并且在任务到达时，M-RSF使用任务运行时间对任务进行排序，而非使用任务到达时间，这样做的好处是可以使短作业尽快进入执行队列，避免了短作业任务因等待长作业任务执行而出现饥饿问题；另一方面是因为M-RSF利用负载模型，合理地设置了时间片大小，并且在最后一级队列中使用了FCFS策略，以确保长作业任务执行完成，并且M-RSF取消了抢占机制，在算法的精简性上也好于MFLQ机制。

综上，由图16~图18可以看出，在任务调度的执行效率表现上，M-RSF机制与复杂的CFS机制表现几乎相当，优于其他常见的FCFS等机制，并

且所表现出的调度优势随着任务数量的增加而增强，进而也证明了M-RSF机制及其算法本身的有效性。

**实验3**　M-RSF和其他调度机制的内存开销

M-RSF和其他调度机制的内存开销如图19所示。

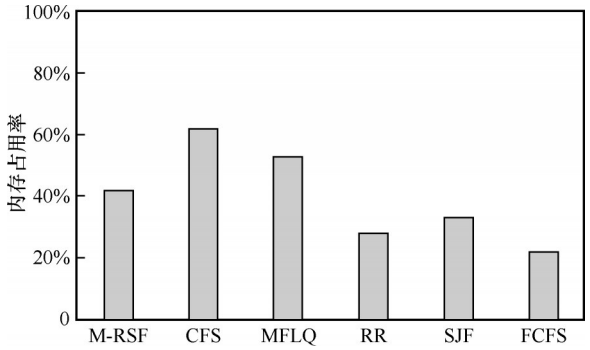


图19　M-RSF和其他调度机制的内存开销

如图19所示，在任务输入数量为2 000，且连续执行50次情况下，内存占用率最高的是CFS策略，因为该策略包含多次排序和抢占等复杂运算，在同等条件下，其内存占用率达62%。接下来是MQMS策略，由于该策略存在负载不均[21]的问题，这会导致内存增加，在同等条件下，MQMS的内存占用率约为53%。然后是本文提出的M-RSF策略，在同等条件下，内存占用率为42%，相比于CFS策略和MQMS策略，内存占用率分别减少约为20%和11%。内存占用率最小的是FCFS策略，因为该策略最为简单，不需要进行排序等运算操作。综上来看，虽然M-RSF策略的内存占用率高于较为简单的FCFS和SJF策略，但是参考实验2的结果可知，M-RSF获得了执行效率的优势，并且内存消耗也小于MQMS和CFS等复杂策略。

**实验4**　MQMS OSv和M-RSF OSv

不同任务数量下MQMS OSv和M-RSF OSv的平均等待时间和平均周转时间如图20~图22所示。

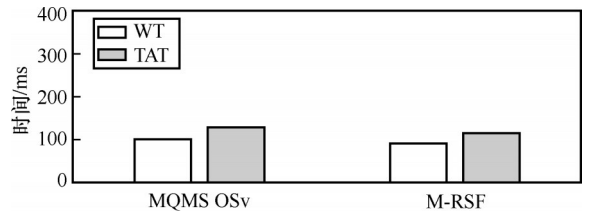


图20　任务数量为500时MQMS&M-RSF的平均等待时间和平均周转时间

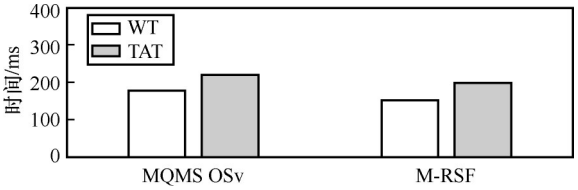


图21　任务数量为1 000时MQMS&M-RSF的平均等待时间和平均周转时间

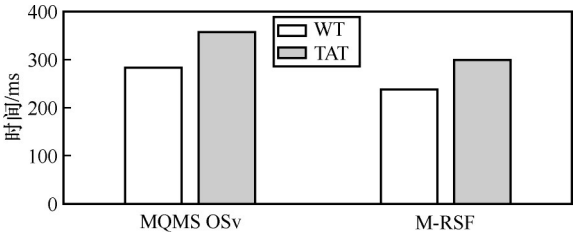


图22　任务数量为2 000时MQMS&M-RSF的平均等待时间和平均周转时间

由图20可以看出，当RR时间片长度为取值均衡点$\lambda$、任务数量为500时，M-RSF OSv的平均等待时间和平均周转时间均少于MQMS OSv。其中，M-RSF OSv相比于未做调度机制更改的MQMS OSv，平均等待时间减少了9.9%，平均周转时间减少了9.3%。如图21所示，当任务数量增加到1 000时，M-RSF OSv的表现同样优于MQMS OSv，平均等待时间减少了13.4%，平均周转时间减少了12.8%。最后，如图22所示，当任务数量增加至2 000时，M-RSF OSv相比于MQMS OSv，平均等待时间减少了15.7%，平均周转时间减少了16.1%。综上来看，OSv Unikernel使用本文提出的M-RSF调度机制后，任务执行效率得到了有效提升，并且随着任务数量的增加，任务的调度和执行效率也逐渐增强，这也进一步证明了本文提出的M-RSF机制在Unikernel大规模任务调度场景下的有效性。

**实验5**　MQMS OSv和M-RSF OSv以及MirageOS Unikernel

不同Unikernnel的吞吐率对比如图23所示（服务器运行40个并发线程，每个线程在测试机器上发出50 000个请求）。

本实验结合实际场景，利用吞吐率对不同Unikernel的调度机制进行评估，即吞吐率越高，调度机制越有效。如图23所示，在云环境应用场景中，M-RSF OSv的吞吐率最高，相比MQMS OSv和MirageOS分别提高约22%和34%，主要原因是M-RSF对云环境下Unikernel的任务特点更具

有针对性，因此表现出了吞吐率的优势，这也验证了M-RSF在实际云环境中的有效性。另外，从图23中还可以看出，MirageOS吞吐率表现最差，这是因为MirageOS使用了FCFS策略，该策略对短作业较为不利，所以MirageOS的吞吐率低于MQMS OSv和M-RSF OSv。

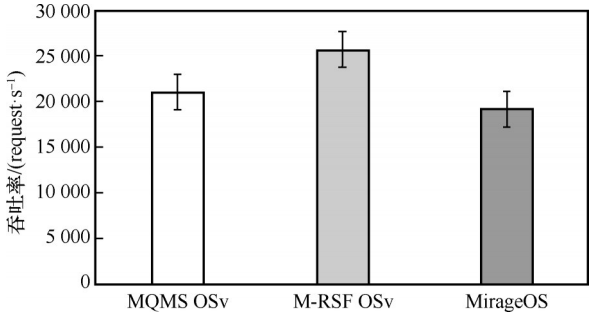


图23　不同Unikernnel的吞吐率对比

## 4　结束语

本文针对云环境下Unikernel的任务调度问题，提出了一种面向Unikernel的多级反馈队列调度机制M-RSF，在保持Unikernel结构不变的情况下，M-RSF能对任务进行更为有效的调度。通过在OSv Unikernel上实现和验证M-RSF，结果显示M-RSF在平均周转时间和平均等待时间上都优于原始OSv的调度机制，证明了M-RSF的有效性。未来，随着Unikernel在人工智能和物联网等领域的应用增多，需要重新设计和改进Unikernel的调度策略和负载模型，这也是本文下一步的研究方向。

## 参考文献：

[1] ALAM T. Cloud computing and its role in the information technology[J]. IAIC Transactions on Sustainable Digital Innovation (ITSDI), 2020, 1(2): 108-115.

[2] MADHAVAPEDDY A, MORTIER R, ROTSOS C, et al. Unikernels[J]. ACM SIGARCH Computer Architecture News, 2013, 41(1): 461-472.

[3] GOETHALS T, SEBRECHTS M, ATREY A, et al. Unikernels vs containers: an in-depth benchmarking study in the context of microservice applications[C]//Proceedings of the 2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2). Piscataway: IEEE Press, 2018: 1-8.

[4] AGACHE A, BROOKER M, FLORESCU A. Firecracker: lightweight virtualization for serverless applications[C]//Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation. Berkeley: USENIX Association, 2020: 419-434.

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

[6] XAVIER B, FERRETO T, JERSAK L. Time provisioning evaluation of KVM, docker and unikernels in a cloud platform[C]//Proceedings of the 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). Piscataway: IEEE Press, 2016: 277-280.

[7] LI Z, CHENG J, CHEN Q, et al. RunD: a lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing[C]//Proceedings of 2022 USENIX Annual Technical Conference. Berkeley: USENIX Association, 2022: 53-68.

[8] IMADA T. MirageOS unikernel with network acceleration for IoT cloud environments[C]//Proceedings of the 2018 2nd International Conference on Cloud and Big Data Computing. New York: ACM Press, 2018: 1-5.

[9] ZHANG Y, CROWCROFT J, LI D, et al. KylinX: a dynamic library operating system for simplified and efficient cloud virtualization[C]//Proceedings of the 2018 Annual Technical Conference. Piscataway: IEEE Press, 2018: 173-186.

[10] KIVITY A, LAOR D, COSTA G, et al. OSv-optimizing the operating system for virtual machines[C]//Proceedings of the 2014 Annual Technical Conference. Piscataway: IEEE Press, 2014: 61-72.

[11] EL-SHARAWY E E. A review on the CPU scheduling algorithms: comparative study[J]. International Journal of Network Security, 2021, 21(1): 19-26.

[12] SHEN Z M, SUN Z, SELA G E, et al. X-containers: breaking down barriers to improve performance and isolation of cloud-native containers[C]//Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. New York: ACM Press, 2019: 121-135.

[13] RAZA A, UNGER T, BOYD M, et al. Unikernel linux (UKL)[C]//Proceedings of the Eighteenth European Conference on Computer Systems. New York: ACM Press, 2023: 590-605.

[14] LI T, BAUMBERGER D, KOUFATY D A, et al. Efficient operating system scheduling for performance-asymmetric multi-core architectures[C]//Proceedings of the 2007 ACM/IEEE conference on Supercomputing. New York: ACM Press, 2007: 1-11.

[15] GHAFARI R, KABUTARKHANI F H, MANSOURI N. Task scheduling algorithms for energy optimization in cloud environment: a comprehensive review[J]. Cluster Computing, 2022, 25(2): 1035-1093.

[16] BRATTERUD A, WALLA A A, HAUGERUD H, et al. IncludeOS: a minimal, resource efficient unikernel for cloud services[C]//Proceedings of the 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom). Piscataway: IEEE Press, 2015: 250-257.

[17] LEFEUVRE H, BĂDOIU V A, JUNG A, et al. FlexOS: towards flexible OS isolation[C]//Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. New York: ACM Press, 2022: 467-482.

[18] LANKES S, PICKARTZ S, BREITBART J. HermitCore: a unikernel for extreme scale computing[C]//Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers. New York: ACM Press, 2016: 1-8.

[19] ALHAIDARI F, BALHARITH T Z. Enhanced round-robin algorithm in the cloud computing environment for optimal task scheduling[J]. Computers, 2021, 10(5): 63-75.

[20] RAZA A, SOHAL P, CADDEN J, et al. Unikernels: the next stage of

linux’s dominance[C]//Proceedings of the Workshop on Hot Topics in Operating Systems. New York: ACM Press, 2019: 7-13.

[21] ZHAO Y, SUO K, WU X F, et al. Preemptive multi-queue fair queuing[C]// Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing. New York: ACM Press, 2019: 147-158.

[22] PEMASINGHE S, RAJAPAKSHA S. Comparison of CPU scheduling algorithms: FCFS, SJF, SRTF, round robin, priority based, and multi-level queuing[C]//Proceedings of the 2022 IEEE 10th Region 10 Humanitarian Technology Conference (R10-HTC). Piscataway: IEEE Press, 2022: 318-323.

[23] CHEN K-H, GÜNZEL M, JABLKOWSKI B. Unikernel-based real-time virtualization under deferrable servers: analysis and realization[C]// Proceedings of 2022 34th Euromicro Conference on Real-Time Systems. Piscataway: IEEE Press, 2022:132-141.

[24] OMAR H K, JIHAD K H, HUSSEIN S F. Comparative analysis of the essential CPU scheduling algorithms[J]. Bulletin of Electrical Engineering and Informatics, 2021, 10(5): 2742-2750.

[25] MOSTAFA S M, AMANO H. Dynamic round robin CPU scheduling algorithm based on K-means clustering technique[J]. Applied Sciences, 2020, 10(15): 5134.

[26] BISWAS D, SAMSUDDOHA M, ASIF M R A, et al. Optimized round robin scheduling algorithm using dynamic time quantum approach in cloud computing environment[J]. International Journal of Intelligent Systems and Applications, 2023, 15(1): 22-34.

[27] KE J C, WU C H, ZHANG Z G. Recent developments in vacation queueing models: a short survey[J]. International Journal of Operations Research, Citeseer, 2010, 7(4): 3-8.

[28] ZENIL H. A review of methods for estimating algorithmic complexity: options, challenges, and new directions[J]. Entropy, 2020, 22(6): 612.

[29] IZURIETA C, BIEMAN J. The evolution of FreeBSD and linux[C]// Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering. New York: ACM Press, 2006: 204-211.

[30] ADHIKARI S, ABBAS M, CHAKRABORTY M, et al. Analysis of average waiting time and server utilization factor using queueing theory in cloud computing environment[J]. The International Journal of Nonlinear Analysis and Applications (IJNAA), 2021, 12(3): 1259-1267.

[31] SHEN G B, LI Q, AI S, et al. How powerful switches should be deployed: a precise estimation based on queuing theory[C]//Proceedings of the IEEE Conference on Computer Communications. Piscataway: IEEE Press, 2019: 811-819.

[32] PRADOS-GARZON J, AMEIGEIRAS P, RAMOS-MUNOZ J J, et al. Performance modeling of softwarized network services based on queuing theory with experimental validation[J]. IEEE Transactions on Mobile Computing, 2021, 20(4): 1558-1573.

[33] GHOMI E J, RAHMANI A M, QADER N N. Applying queue theory for modeling of cloud computing: a systematic review[J]. Concurrency and Computation: Practice and Experience, 2019, 31(17): e5186.

[34] MAS L, VILAPLANA J, MATEO J, et al. A queuing theory model for fog computing[J]. The Journal of Supercomputing, 2022, 78(8): 11138-11155.

[35] ALSULAMI A A, ABU AL-HAIJA Q, THANOON M I, et al. Performance evaluation of dynamic round robin algorithms for CPU scheduling[C]//Proceedings of the 2019 SoutheastCon. Piscataway: IEEE Press, 2019: 1-5.

[36] MAHESH KUMAR M R, RAJENDRA B R, NIRANJAN C K, et al. Prediction of length of the next CPU burst in SJF scheduling algorithm using dual simplex method[C]//Proceedings of the Second International Conference on Current Trends in Engineering and Technology. Piscataway: IEEE Press, 2014: 248-252.

[37] HELMY T, AL-AZANI S, BIN-OBAIDELLAH O. A machine learning-based approach to estimate the CPU-burst time for processes in the computational grids[C]//Proceedings of the 2015 3rd International Conference on Artificial Intelligence, Modelling and Simulation (AIMS). Piscataway: IEEE Press, 2015: 3-8.

[38] YADAV R, UPADHYAY A. A fresh loom for multilevel feedback queue scheduling algorithm[J]. International Journal of Advances in Engineering Sciences, 2012, 2: 21-23.

[39] SINGH J, DEEPALI G. An smarter multi queue job scheduling policy for cloud computing[J]. International Journal of Applied Engineering Research, 2017, 12(9): 1929-1934.

[40] CHAHAR V, RAHEJA S. Fuzzy based multilevel queue scheduling algorithm[C]//Proceedings of the 2013 International Conference on Advances in Computing, Communications and Informatics (ICACCI). Piscataway: IEEE Press, 2013: 115-120.

[作者简介]

**董博南**（1987– ），男，北京人，中国科学院大学博士生，主要研究方向为操作系统、计算机架构、系统安全。

**杨秋松**（1977– ），男，河北沧州人，博士，中国科学院大学教授、博士生导师，主要研究方向为操作系统、软件工程、系统安全。

**李明树**（1966– ），男，吉林长春人，博士，中国科学院大学教授、博士生导师，主要研究方向为操作系统、软件工程、分布式系统。