计算机科学
COMPUTER SCIENCE

基于混合关键系统的容器调度架构设计

邓广宏, 张棋恒

引用本文

邓广宏, 张棋恒. 基于混合关键系统的容器调度架构设计[J]. 计算机科学, 2023, 50(6A): 220800215-5.

DENG Guanghong, ZHANG Qiheng. Container-based Scheduling Architecture for Mixed-Criticality Systems [J]. Computer Science, 2023, 50(6A): 220800215-5.

相似文章推荐（请使用火狐或 IE 浏览器查看文章）

Similar articles recommended (Please use Firefox or IE to view the article)

深度学习容器云平台下的GPU共享调度系统

GPU Shared Scheduling System Under Deep Learning Container Cloud Platform

计算机科学, 2023, 50(6): 86-91. https://doi.org/10.11896/jsjkx.220900110

面向Docker容器的动态负载算法

Dynamic Loading Algorithm for Docker Container

计算机科学, 2021, 48(6): 276-281. https://doi.org/10.11896/jsjkx.200500152

基于Kubernetes的分布式TensorFlow平台的设计与实现

Design and Implementation of Distributed TensorFlow Platform Based onKubernetes

计算机科学, 2018, 45(11A): 527-531.

轨道交通实时以太网交换机启动性能的分析与优化

Analysis and Optimization of Boot-up Performance for Railway Real-time Ethernet Switch

计算机科学, 2017, 44(Z11): 276-280. https://doi.org/10.11896/j.issn.1002-137X.2017.11A.059

基于Linux的网络信息内容分析与监管技术

计算机科学, 2003, 30(4): 67-69.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.220800215

基于混合关键系统的容器调度架构设计

邓广宏　张棋恒
武汉数字工程研究所　武汉 430205
(stephenden@163.com)

摘　要　针对实时容器和非实时容器混合的关键任务系统中的容器调度的实时性保障问题以及 CPU 资源的分配问题,提出了一种实时/非实时容器混合关键系统的容器调度架构,基于分层调度模型调度容器控制组的实时运行队列,并限制了非实时容器分配的系统资源,来确保每个实时容器的实时性;同时增加负载监控和均衡策略,在保证实时性的条件下对非实时容器占用的 CPU 资源进行合理分配。实验结果表明,该方法解决了实时容器和非实时容器混用情况下调度机制对实时容器造成的实时性降低问题。

关键词:混合关键系统;实时容器;CPU 调度;Docker;Linux
中图法分类号　TP393

Container-based Scheduling Architecture for Mixed-Criticality Systems

DENG Guanghong and ZHANG Qiheng
Wuhan Digital Engineering Institute,Wuhan 430205,China

Abstract　The mixed-criticality systems composed of real-time containers and non-real-time containers have difficulties in ensuring the real time of scheduling and the allocation of cpu resources. In this paper,we present an architecture of scheduling RT containers and NRT containers for mixed-criticality systems,which is based on the hierachical scheduler to schedule the run-queues of the container control groups. By this means,our architecture ensures the real-time of RT containers by limiting system resources to NRT containers. We also add monitor and load balancer for workloads to ensure equitable allocation of CPU resources occupied by NRT containers. Experimental results show that the proposed architecture can improve the degradation of real-time in RT containers when RT containers coexist with NRT containers in mixed-criticality systems.

Keywords　Mixed-criticality system,Real-time container,CPU scheduling,Docker,Linux

1　引言

容器具有性能接近原生操作系统、能够简化集成测试过程、高资源利用率、低开销等优点,被广泛应用于舰船、航空航天、工业自动化等领域,特别是在军用领域,由于其存在较强的实时性、安全性和可靠性要求,容器往往需要针对业务系统进行特别设计,增强其实时性和安全性[1]。同时,业务系统的复杂性也造成多类容器共存共管和调度的问题,实时容器与非实时容器的调度保障设计和安全各类设计[2]就是该类系统极为重要且急需解决的问题。

目前,国内针对实时容器对外公开的研究较少,比较有代表性的有翼辉基于 SylixOS 开发的实时容器 ECS[3],但受限于开发时间以及软件生态支持,该实时容器尚不成熟,难以应对构建大型容器云的需求。国外对于实时容器的研究相对丰富,大多基于实时 Linux 补丁与 Docker 的组合实现实时容器。实现实时容器的方法根据实现原理不同可分为基于抢占式补丁、基于实时辅助微内核和基于分层调度方法 3 种[4],这些方法各有优劣,对应的应用场景不尽相同。

混合关键系统(Mixed-Criticality Systems)在汽车、舰船、航空航天等领域有着广泛的应用,它可以定义为在同一硬件平台上集成具有不同关键性的软件组件的实时系统[5]。混合关键系统要求任务执行时间和故障隔离得到保证。换句话说,低关键性任务的崩溃或延迟不应损害独立的高关键性任务的正确执行。因此,人们想要对关键任务和非关键任务进行隔离,于是基于容器的轻量级虚拟化技术在这一领域开始得到应用。在基于容器的混合关键性系统中[6],容器既可以执行实时任务也可以执行非实时任务,但同一容器中执行任务的类型必须相同。因此,如何合理地调度实时容器和非实时容器成为了一个重要的问题。在混合关键系统中实现实时容器不仅仅是简单地将 Docker 容器应用于实时操作系统,还要考虑众多兼容性问题、实时容器与非实时容器的 CPU 分配问题,以及容器化后如何保证任务响应延迟的最小化[7]。本文采用基于分层调度补丁的实时 Linux 与 Docker 提出了一种混合关键系统下的容器运行架构,主要针对该系统中的实时容器和非实时容器实现了一种合理的调度方案。

2　混合关键系统的容器架构

2.1　系统架构

目前,许多服务器应用程序使用 Docker 虚拟化为容器。这样的虚拟化系统可能存在多个容器,它们具有不同的关键性

通信作者:张棋恒(335428101@qq.com)

和时间限制，我们将这样的系统称为 Docker 容器化混合关键实时系统，它由一组实时容器和非实时容器组成，其中实时容器和非实时容器分别运行实时任务和非实时任务。如果想满足实时容器的时序要求，就必须对 CPU 进行合理的分配。

分层调度模型将 SCHED_DEADLINE 调度策略[8]（以 CBS 算法[9]为基础）扩展为调度 Linux 控制组，允许在 SCHED_DEADLINE 调度的控制组内以固定优先级调度用户线程。它能够确保实时容器被正确调度，在只有实时容器运行的系统中能取得很好的效果，但是在混合关键系统中，该调度策略不能保证非实时容器总是被分配到足够的 CPU 时间，特别是非实时容器对 CPU 的需求存在动态变化的可能性。

图 1 描述了基于实时 Linux 内核实现的混合关键实时系统的容器架构，其核心思想是由不同的模块分别对实时容器和非实时容器进行调度，因为实时任务有时间限制，所以系统会首先满足实时容器的 CPU 需求。实时容器的调度由基于分层调度补丁的 Linux 内核来解决，分层调度模型能够使用 SCHED_DEADLINE 策略来调度每个控制组的实时运行队列，保证每个实时容器的实时性。对于非实时容器，系统会在运行时通过动态调整模块动态调整每个非实时容器分配的 CPU 容量；同时，为了适应非实时容器可变的工作负载，采用性能监控模块每 1 秒收集一次非实时容器的 CPU 使用率，动态调整模块根据这些信息来确定如何调整 CPU 分配。

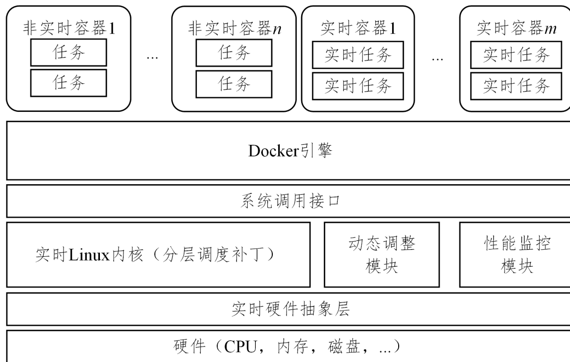


图 1 混合关键系统容器架构图

Fig. 1 Containerized mixed-criticality system architecture

2.2 实时容器改造

实时容器一般指在实时系统中运行的具有实时能力的容器，由于容器本身可以分为隔离的进程，因此对容器的实时性改造本质上可以视为两个步骤，即对 Linux 内核进程调度机制的改进以及对容器的一些特殊启动参数的设置。目前实现实时容器的研究可大致分为 3 个方向：基于抢占式补丁、基于实时辅助微内核（RT-Linux 或 Xenomai[10]）、基于分层调度方法[11]。由于本文采用基于分层调度的思想实现实时 Linux 内核，因此在本节对前两种方法不多赘述。

在 Linux 系统中，控制组（Cgroups）可以用于对进程进行分组并在其基础上对进程进行监控和资源管理。对于 Docker 而言，控制组为每个容器限制其能支配的资源，而实现容器的实时性要求对每个进程的资源使用进行精确的把控，因此对控制组的改进是实现容器实时性的关键问题。控制组技术将系统中的所有进程组织成一颗或多颗树结构，树结构的每个节点可视为一个控制组（可理解为一个进程组）。在分层调度的思想中，所有控制组都与明确指定的运行时间和周期相关联，并且 SCHED_DEADLINE 调度策略限制控制组中实时任务占用的 CPU 不能超出控制组 CPU 利用率（即指定的运行时间和周期的比值）的一个阈值。具体来说，一个父控制组下的所有子控制组的 CPU 利用率之和不大于父控制组的 CPU 利用率。每次创建子控制组时，系统从父控制组分配一部分运行时间给子控制组，同时将子控制组关联的截止期限实体（Deadline Entities）与父控制组关联的实体插入到同一运行队列中（即 dl_sched 运行队列，2.3 节会详细说明），进而通过 SCHED_DEADLINE 策略进行调度。这样，每个控制组的时间隔离属性和预留保证都能够得以保留，从而进一步保障容器的实时性。

2.3 实时容器/非实时容器调度设计

Linux 中基于容器的虚拟化是通过结合控制组（Cgroups）和命名空间（Namespace）这两种不同的机制来实现的。命名空间用于隔离和虚拟化系统资源，影响资源的可见性和可访问性；控制组用于将系统进程分组组织起来，并限制、控制或监视这些进程组使用的资源量，影响资源的调度和控制。

如图 2 所示，Linux 每个 CPU 具备一个运行队列（Runqueues），每个运行队列维护多个子调度器，每个子调度器维护自己的就绪队列，运行队列作为这些子调度器类的抽象对它们进行管理。当调度工作进行时，运行队列将根据情况选择子队列处理调度请求。调度器类包括 dl_sched_class，rt_sched_class，fair_sched_class 等，调度器类之间具有优先级的区分，高优先级调度器中的就绪任务总是优先执行，并且实时任务调度器类的优先级总是高于非实时任务。其中 rt_sched_class 是实时调度器类，实时进程一般通过 SCHED_FIFO 和 SCHED_RR 两种固定优先级调度策略调度；fair_sched_class 调度器类就是完全公平调度器（Completely Fair Scheduler，CFS）的抽象，普通的非实时进程由该调度器进行管理；dl_sched_class 调度器类对应另一种实时调度器，使用 SCHED_DEADLINE 策略对实时进程进行调度，该策略根据每个任务自己的期限（Deadline）来进行调度，最短期限的任务将被最先服务。

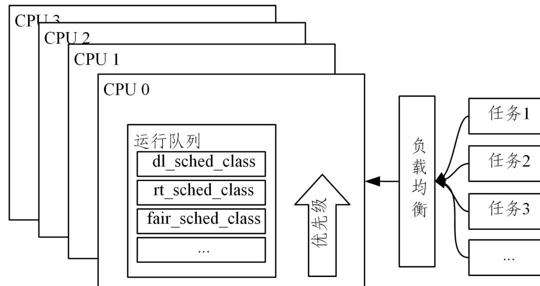


图 2 CPU 运行队列和子调度器的关系示意图

Fig. 2 Relationship between CPU runqueues and scheduler classes

分层调度机制使用基于 EDF 算法的 SCHED_DEADLINE 策略对容器进行调度，通过联系 dl_sched 子队列中的任务与容器 Cgroups 来实现。当调度容器内的任务时，则

使用固定优先级调度策略(FIFO/ RR)来选择最高优先级的任务。也就是说，分层调度机制是一个两级的调度层级结构，一个是基于 SCHED_DEADLINE 策略的根调度器，另一个是基于固定优先级的本地调度器，前者负责在容器级别进行调度，后者负责在任务级别进行调度。从技术上来说，每个控制组与每个运行队列的截止日期实体相关联，只有当组的截止日期实体已由 SCHED_DEADLINE 调度时，才会调度在控制组内运行的固定优先级任务。分层调度机制容易配置且取得了良好的效果，并能够降低系统内实时计算带宽的占用，为容器提供了良好的实时性。

分层调度补丁提供了对混合关键系统中实时容器的调度保证，使得实时容器总是能够获得所需 CPU 时间。然而，如何将剩余的 CPU 时间合理分配给非实时容器却没有一个合理的机制，如果根据默认的完全公平调度策略 CFS 对非实时容器进行 CPU 分配，非实时容器可能会因为工作负载在运行时发生变化，也就是说，它是不可预测的，而 CFS 只能根据已知的静态优先级进行公平调度[12]，不能根据实时工作负载为非实时容器提供足够的 CPU 时间。

本文针对非实时容器的 CPU 分配提出了一种新的策略，通过实现动态调整模块和性能监控模块的协同工作来应对非实时容器不确定的工作负载，将剩余的 CPU 时间合理分配给每个非实时容器。其中，性能监控模块主要负责监测非实时容器的 CPU 利用率，每间隔 1 秒记录一次信息；动态调整模块从性能监控模块收集需要的信息，并通过动态调整算法计算每个非实时容器应该被分配的 CPU 时间。

3 系统实现

3.1 实时容器的启动

启动实时容器首先需要实现 Linux 系统的实时化，这就需要在修改内核代码并应用补丁后重新编译 Linux 内核。以 Debian 系统为例，首先需要下载与系统内核版本近似的 Linux 内核，利用 patch 命令将补丁内容应用至内核源码，实现分层调度模型；然后使用 menuconfig 对 Linux 系统进行配置，将 Linux 内核的抢占模式改为完全可抢占的内核，使得所有内核代码可通过抢占来减少内核的延迟；最后重新编译内核代码并重启系统就能得到应用了分层调度补丁的实时 Linux 内核。

在基于分层调度的实时 Linux 内核下，启动实时容器需要对 Docker daemon 进行设置，增加 cpu-rt-runtime 参数为容器设置可运行在实时模式下的最大时间，增加 ulimit rtprio 参数为容器设置最大的实时优先级限制，同时建议在 Docker 启动参数中将容器设置为 host 网络模式，降低网络传输带来的延迟，以获得最好的实时性。

3.2 混合容器动态调整算法

实时容器总是优先于非实时容器得到响应。本文提出的系统中，包含一组非实时容器$NRTC_i$($i=1,2,\cdots,n$)和一组实时容器RTC_j($j=1,2,\cdots,m$)。对于每个实时容器RTC_j，假设一个容器在每个周期P_j可以占用的 CPU 时间为Q_j，那么比值$\frac{Q_j}{P_j}$就可被视为实时容器RTC_j的 CPU 利用率。如果将可用的 CPU 时间表示为$T^{\text{available}}$，则剩余的 CPU 时间T^{remain}可表示为：

$$T^{\text{remain}}=T^{\text{available}}\times\left(1-\sum_{j=1}^{m}\frac{Q_j}{P_j}\right),j=1,2,\cdots,m \tag{1}$$

一开始，系统将剩余的 CPU 时间T^{remain}等分给系统内所有非实时容器，即每个非实时容器分配到的运行时间为$\frac{T^{\text{remain}}}{n}$。为了便于解释，假设非实时容器$NRTC_i$被分配到的 CPU 时间用$T_i^{\text{allocated}}$来表示。动态调整算法能够在运行时通过动态调整模块为非实时容器分配 CPU 时间。为了适应非实时容器内多变的工作负载，动态调整模块使用与非实时容器近期工作负载相关的信息来决定如何调整 CPU 时间的分配，这些信息是由工作负载模块每隔 1 秒进行收集的，主要包括所有非实时容器最近第 x 次的 CPU 使用时间T_i^x和最近 5 次的 CPU 使用时间平均值T_i^{avg}。每个非实时容器动态调整算法的伪代码如算法 1 所示。

算法 1 CPU 分配动态调整算法

输入：(n, T_i^1, $T^{avg\,i}$)

输出：无

1. 令 $\sigma\leftarrow 0$
2. for i=1→n do
3.   if (($T_i^{\text{allocated}}>T_i^{\text{avg}}\times 1.05$) ∧ ($T_i^1\leqslant T_i^{\text{avg}}\times 1.05$)) then
4. $\sigma\leftarrow\sigma+T_i^{\text{allocated}}-T_i^{\text{avg}}\times 1.05$
5. $T_i^{\text{allocated}}\leftarrow T_i^{\text{avg}}\times 1.05$
6.   end if
7. end for
8. 令 $\Delta\leftarrow\sigma/n$
9. for i=1→n do
10. $T_i^{\text{allocated}}\leftarrow T_i^{\text{allocated}}+\Delta$
11. end for

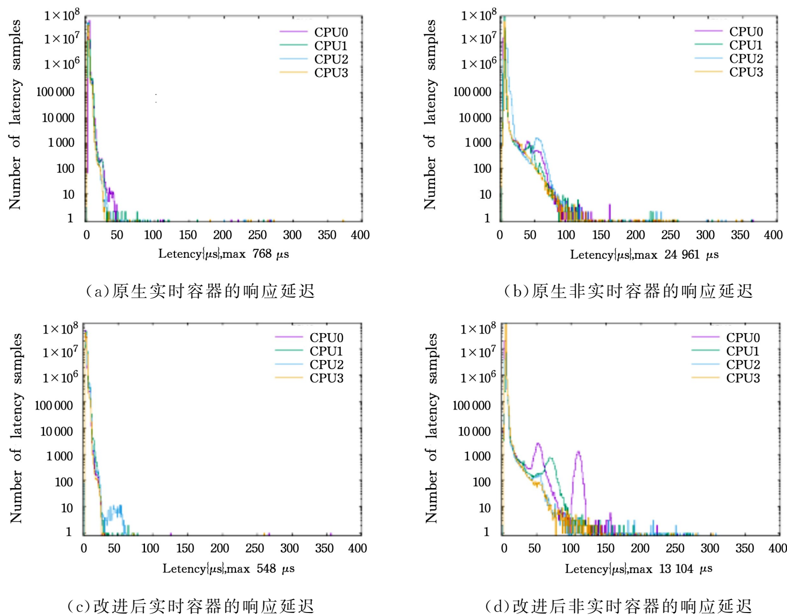
该算法首先对当前系统中每个非实时容器进行判断，若当前非实时容器被分配到的时间大于最近 5 个被记录容器(包括自身)CPU 使用时间的平均值，则说明该非实时容器被分配了过多的 CPU 时间，算法会将该容器分配到的 CPU 时间动态调整为平均值$T_i^{\text{avg}}\times 1.05$(多出的 5% 是为了避免抖动效应带来的影响)。同时，需要保证其自身 CPU 的使用时间不能超出新分配的 CPU 时间(算法第 3 行条件 2)。算法使用 σ 来记录每个容器动态调整后释放的 CPU 时间，并将这些冗余时间重新等分给系统中每个非实时容器。因此，如果一个非实时容器近期出现了较大的工作量，则可以通过动态调整算法获得更多的 CPU 时间，从而进一步提升性能。

4 实验

本文实验在 4 核 vCPU 的服务器以及 8 GB 内存的虚拟机上进行，处理器型号为 Intel Xeon Platinum 8374C CPU @ 2.70 GHz。为了验证本文提出的调度机制的效果，实验主要对比系统中同时运行实时和非实时容器的情况下，在采用

CPU 分配动态调整算法前后实时容器和非实时容器的响应延迟。为了达到这一实验目标，本文在 Debian 11.1 系统中同时运行两个实时 Docker 容器和两个非实时 Docker 容器，同时使用 stress-ng 工具为系统模拟工作负载，在 CPU 加压至 100%的情况下再通过 Cyclictest 工具分别测试两种容器内的最坏响应延迟。为了更普遍地反应容器的实时性，设置 Cyclictest 的参数线程间隔为 200 μs，循环次数为 100 000 000(共执行约 6 小时)，最终绘制直方图展示实验结果，如图 3 所示。

图 3 中的直方图纵轴表示采集的延迟样本数量，横轴表示响应延迟，其中图 3(a)和图 3(b)分别表示在没有采用 CPU 分配动态调整算法前实时容器和非实时容器的延迟情况，作为实验的对照组，图 3(c)和图 3(d)分别表示采用 CPU 分配动态调整算法后实时容器和非实时容器的延迟情况。在实时性的评估中，相较于延迟的平均值，最大响应延迟的大小是更应该被关注的指标。对比图 3(a)和图 3(c)可以看出，实时容器在两种情况下都具有较小的响应延迟，多数样本集中于 50 μs 以内，最大响应延迟分别达到 548 μs 和 768 μs，这主要得益于实时容器及内核的改造。对比图 3(b)与图 3(d)可知，由于系统总是优先响应实时容器，因此非实时容器在满载的情况下具有相对较高的延迟，经过本文算法的改进，系统中每个非实时容器都能在近似的时间内得到响应，且在一定程度上缩短了每个非实时容器的响应延迟。



(a)原生实时容器的响应延迟

(b)原生非实时容器的响应延迟

(c)改进后实时容器的响应延迟

(d)改进后非实时容器的响应延迟

图 3 采用 CPU 分配动态调整算法前后实时容器和非实时容器的延迟折线图

Fig. 3 Latency line chart of RT and NRT containers before and after the application of dynamic CPU allocation algorithm

结束语 本文针对军用领域容器应用的实时性问题，对在实时容器和非实时容器共存的混合关键系统下实时容器的实时性和安全性难以保障的问题，提出了一种混合关键系统的容器调度管理机制，基于分层调度模型调度容器控制组的实时运行队列，保证每个实时容器的实时性，并限制非实时容器分配的容器资源，增加负载监控和均衡策略，根据动态调整算法将其合理分配给非实时容器，这样能在保证实时容器的实时性的同时，使得非实时容器也能够在有限的时间内得到响应，不会出现不断被抢占而导致“饿死”的情况。实验结果表明，在实时容器和非实时容器混用的情况下，所提方法在保证实时容器实时性的同时，还能够保证每个非实时容器在有限且更短的时间内得到响应。

本文只考虑了单宿主机 CPU 的分配问题，在未来，我们计划针对多宿主机多容器的架构下改进 CPU 调度算法，并在多宿主机的环境中考虑更加复杂的容器编排问题，其重点在于容器的准入、分配以及在运行时性能下降的情况下对已部署的容器进行动态资源调整。

参 考 文 献

[1] HOFER F, SEHR M A, IANNOPOLLO A, et al. Industrial control via application containers: Migrating from bare-metal to IAAS[J]. arXiv:1908.04465, 2019.

[2] CINQUE M, DELLA CORTE R, ELISO A, et al. Rt-cases: Container-based virtualization for temporally separated mixed-criticality task sets[C]//31st Euromicro Conference on Real-Time Systems(ECRTS 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[3] LI X C. Overview of SylixOS Safty Container[J]. Microcontrollers & Embedded Systems, 2021, 21(6): 4.

[4] STRUHÁR V, BEHNAM M, ASHJAEI M, et al. Real-time containers: A survey[C]//2nd Workshop on Fog Computing and the IoT(Fog-IoT 2020). Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[5] BURNS A, DAVIS R I. A survey of research into mixed criticality systems[J]. ACM Computing Surveys (CSUR), 2017, 50(6): 1-37.

[6] CINQUE M, DE TOMMASI G. Work-in-Progress: Real-Time Containers for Large-Scale Mixed-Criticality Systems[C]//2017 IEEE Real-Time Systems Symposium(RTSS). IEEE, 2017: 369-371.

[7] BARLETTA M, CINQUE M, DE SIMONE L, et al. Achieving isolation in mixed-criticality industrial edge systems with real-

time containers[C]// 34th Euromicro Conference on Real-Time Systems (ECRTS 2022). Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[8] LELLI J, SCORDINO C, ABENI L, et al. Deadline scheduling in the Linux kernel[J]. Software: Practice and Experience, 2016, 46(6): 821-839.

[9] ABENI L, BUTTAZZO G. Integrating multimedia applications in hard real-time systems[C]// Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279). IEEE, 1998: 4-13.

[10] GERUM P. Xenomai-Implementing a RTOS emulation framework on GNU/Linux[J/OL]. http://xenomai.org/documentaion/xenomai-2.0/pdf/xenomai.pdf.

[11] ABENI L, BALSINI A, CUCINOTTA T. Container-based real-time scheduling in the linux kernel[J]. ACM SIGBED Review, 2019, 16(3): 33-38.

[12] WONG C S, TAN I, KUMARI R D, et al. Towards achieving fairness in the Linux scheduler[J]. ACM SIGOPS Operating Systems Review, 2008, 42(5): 34-43.

DENG Guanghong, born in 1981, Ph.D, professor. His main research interests include cloud computing, virtualization, AI and ship electronic information system.

ZHANG Qiheng, born in 1998, postgraduate. His main research interests include cloud computing and container technology.