Red Hat Enterprise Linux for Real Time 9

了解 RHEL for Real Time

RHEL for Real Time 的基本概念介绍

Last Updated: 2023-12-30

Red Hat Enterprise Linux for Real Time 9 了解 RHEL for Real Time

RHEL for Real Time 的基本概念介绍

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

法律通告

Copyright © 2022 | You need to change the HOLDER entity in the en-US/Understanding_RHEL_for_Real_Time.ent file |.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

摘要

本指南帮助用户了解为 Real Time 内核优化 RHEL 的基本概念和关联的参考，以维护低延迟、一致的响应时间和确定性。

目录

使开源包含更多 4

对红帽文档提供反馈 5

第 1 章 用于 RHEL 的硬件平台(REAL TIME) 6
- 1.1. 处理器内核 6
- 1.2. 其他资源 7

第 2 章 用于 REAL TIME 的 RHEL 上的内存管理 8
- 2.1. 需求分页 8
- 2.2. 主要和次要页面错误 8
- 2.3. MLOCK 系统调用 9
- 2.4. 共享库 10
- 2.5. 共享内存 11

第 3 章 RHEL FOR REAL TIME 的硬件中断 12
- 3.1. LEVEL-SIGNALED 中断 12
- 3.2. MESSAGE-SIGNALED 中断 12
- 3.3. 不可屏蔽中断 12
- 3.4. 系统管理中断 12
- 3.5. 高级可编程中断控制器 13

第 4 章 RHEL 用于 REAL TIME 流程和线程 14
- 4.1. PROCESS 14
- 4.2. 线程 14
- 4.3. 其他资源 15

第 5 章 RHEL FOR REAL TIME 的应用程序时间戳 16
- 5.1. 硬件时钟 16
- 5.2. POSIX 时钟 16
- 5.3. CLOCK_GETTIME() FUNCTION 16
- 5.4. 其他资源 17

第 6 章 RHEL FOR REAL TIME 的调度策略 18
- 6.1. 调度程序策略 18
- 6.2. SCHED_DEADLINE 策略的参数 18

第 7 章 RHEL FOR REAL TIME 中的关联性 20
- 7.1. 处理器关联性 20
- 7.2. SCHED_DEADLINE 和 CPUSETS 20

第 8 章 RHEL FOR REAL TIME 中的线程同步机制 22
- 8.1. MUTEXES 22
- 8.2. 障碍 22
- 8.3. 条件变量 23
- 8.4. MUTEX 类 23
- 8.5. 线程同步功能 24

第 9 章 用于 REAL TIME 的 RHEL 中的套接字选项 25
- 9.1. TCP_NODELAY SOCKET 选项 25
- 9.2. TCP_CORK 套接字选项 26
- 9.3. 使用套接字选项的示例程序 26

第 10 章 RHEL FOR REAL TIME 调度程序 29

10.1. 用来设置调度程序的 CHRT 工具 29
10.2. 抢占调度 29
10.3. 调度程序优先级的库函数 30

第 11 章 RHEL FOR REAL TIME 中的系统调用 31
11.1. SCHED_YIELD() FUNCTION 31
11.2. GETRUSAGE（）函数 31

第 12 章 在 RHEL FOR REAL TIME 上使用 MLOCK （）系统调用 33
12.1. MLOCK （）和 MUNLOCK （）系统调用 33
12.2. 使用 MLOCK （）系统调用锁定页面 34
12.3. 使用 MLOCKALL （）系统调用锁定所有映射的页面 35
12.4. 使用 MMAP （）系统调用将文件或设备映射到内存 36
12.5. MLOCK （）系统调用的参数 37

第 13 章 在 RHEL FOR REAL TIME 中设置 CPU 关联性 39
13.1. 使用 TASKSET 命令调整处理器关联性 39
13.2. 使用 SCHED_SETAFFINITY（）系统调用设置处理器关联性 40
13.3. 隔离单个 CPU 来运行高利用率任务 41

使开源包含更多

红帽致力于替换我们的代码、文档和 Web 属性中存在问题的语言。我们从这四个术语开始：master、slave、黑名单和白名单。由于此项工作十分艰巨，这些更改将在即将推出的几个发行版本中逐步实施。详情请查看 CTO Chris Wright 的信息。

对红帽文档提供反馈

我们感谢您对我们文档的反馈。让我们了解如何改进它。

对特定段落提交评论

1. 查看 **Multi-page HTML** 格式的文档，并确保在页面完全加载后可看到右上角的 **Feedback** 按钮。
2. 使用光标突出显示您要评论的文本部分。
3. 点击在高亮文本旁的 **Add Feedback** 按钮。
4. 添加您的反馈，并点击 **Submit**。

通过 Bugzilla（需要帐户）提交反馈

1. 登录到 Bugzilla 网站。
2. 从 **Version** 菜单中选择正确的版本。
3. 在 **Summary** 字段中输入描述性标题。
4. 在 **Description** 字段中输入您对改进的建议。包括到文档相关部分的链接。
5. 点 **Submit Bug**。

第 1 章 用于 RHEL 的硬件平台(REAL TIME)

在设置实时环境时，正确配置硬件会扮演重要角色，因为硬件会影响您的系统运行方式。并非所有硬件平台都能够实时启用微调。在进行调优之前，您必须确保潜在的硬件平台能够实时。

硬件平台因供应商而异。您可以使用硬件延迟检测器(**hwlatdetect**)程序测试并验证硬件是否适合性。程序控制延迟 detector 内核模块，并有助于检测底层硬件或固件行为造成的延迟。

完成低延迟操作所需的任何调整步骤。具体步骤请参阅厂商文档。

先决条件

- 已安装 RHEL-RT 软件包。
- 完成低延迟操作所需的任何调整步骤。有关减少或删除将系统过渡到系统管理模式(SMM)的任何系统管理中断(SMI)的说明，请参阅厂商文档。

警告

红帽建议不要完全禁用系统管理中断(SMI)，因为它可能会导致灾难性硬件故障。

1.1. 处理器内核

实时处理器内核是物理中央处理单元(CPU)，它执行机器代码。套接字是处理器和计算机的主板之间的连接。套接字是处理器所放入的主板上的位置。有两组处理器：

- 使用一个可用内核占用一个插槽的单一核心处理器。
- 占用 4 个可用内核的插槽的四核处理器。

在设计实时环境时，请注意可用内核数、内核之间的缓存布局以及核心如何进行物理连接。

当有多个内核可用时，请使用线程或进程。在不使用这些构造的情况下编写程序，一次在单个处理器上运行。多核平台通过将不同的内核用于不同类型的操作来提供优势。

缓存

缓存对总体处理时间和确定性造成显著影响。通常，应用的线程需要同步对共享资源的访问权限，如数据结构。

使用 **tuna** 命令行工具(CLI)，您可以确定缓存布局并将交互的线程绑定到核心，以便它们共享缓存。缓存共享通过确保相互排除原语（mutex、condition 变量或类似数据）来减少内存故障，数据结构使用相同的缓存。

互连

增加系统上的内核数量可能会导致相互冲突的要求。这使得有必要确定互连拓扑，以帮助检测实时系统内核间的冲突。

现在，许多硬件厂商在内核和内存（称为非统一内存访问(NUMA)架构）之间提供透明的互连网络。

NUMA 是一个在多处理中使用的系统内存设计，内存访问时间取决于与处理器相关的内存位置。使用 NUMA 时，处理器可以比非本地内存快访问自己的本地内存，如处理器或处理器共享内存或内存。在 NUMA 系统中，了解互连拓扑有助于设置在相邻内核上经常通信的线程。

taskset 和 **numactl** 实用程序决定 CPU 拓扑。**taskset** 定义没有 NUMA 资源（如内存节点和 **numactl**）的 CPU 关联性，控制进程和共享内存的 NUMA 策略。

1.2. 其他资源

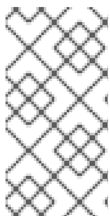
- 为 Real Time 安装 RHEL 9

第 2 章 用于 REAL TIME 的 RHEL 上的内存管理

实时系统使用虚拟内存系统，其中由用户空间应用程序引用的地址转换为物理地址。翻译是通过底层计算系统中页表和地址转换硬件的组合来实现的。在程序和实际内存之间转换机制的一个优点是，操作系统在需要或 CPU 请求时可以交换页面。

实时从二级存储到主内存交换页面，之前使用的页表条目被标记为无效。因此，即使在普通内存压力下，操作系统也可以从一个应用程序中检索页面并将其提供给另一个应用程序。这可能导致无法预计的系统行为。

内存分配实现包括要求分页机制和内存锁定(**mlock**（)）系统调用。



注意

在不同缓存和 NUMA 域中共享 CPU 的数据信息可能会导致流量问题和瓶颈。

在编写多线程应用程序时，在设计数据定位前考虑机器拓扑。拓扑是内存层次结构，包括 CPU 缓存和 Non-Uniform Memory Access(NUMA)节点。

2.1. 需求分页

需求分页与页面交换的分页系统类似。系统在需要或 CPU 需求时，载入二级内存中存储的页面。程序生成的所有内存地址都通过处理器中的地址转换机制。然后，地址将从进程特定虚拟地址转换为物理内存地址。这称为虚拟内存。转换机制中的两个主要组件是页表和翻译缓冲区(TLBs)

页表

页面表是物理内存中的多级表，其中包含虚拟到物理内存的映射。这些映射可由处理器中的虚拟内存转换硬件读取。

带有分配的物理地址的页表条目被称为常驻工作集。当操作系统需要释放其他进程的内存时，它可以从常驻工作集交换页面。在交换页面时，对此页面内的虚拟地址的任何引用都会产生页面错误，并导致页面重新分配。

当系统在物理内存上极低时，交换进程会启动到thrash，这会持续从进程中传输页面，并且永远不会允许进程完成。您可以通过在 **/proc/vmstat** 文件中查找 **pgfault** 值来监控虚拟内存统计信息。

翻译的缓冲

转换 Lookaside Buffers(TLBs)是虚拟内存转换的硬件缓存。任何具有 TLB 的处理器内核都会通过启动内存读取页表条目并行检查 TLB。如果虚拟地址的 TLB 条目有效，则内存读取将中止，并且 TLB 中的值用于地址转换。

TLB 在参考的本地原则上运行。这意味着，如果代码在很长一段时间内一直保留在一个内存区域（如循环或调用相关功能），则 TLB 引用避免了地址转换的主内存。这可显著加快处理时间。

在编写确定性和快速代码时，请使用可保持参考位置的功能。这意味着使用循环而不是递归。如果 recursion 不可避免，请将递归调用放在函数的末尾。这称为 tail-recursion，这使得代码在相对较小的内存区域工作，并避免从主内存调用表转换。

2.2. 主要和次要页面错误

RHEL for Real Time 通过将物理内存分成多个块，然后将内存映射到虚拟内存。当进程需要未映射或在内存中不再可用的特定页面时，会实时发生故障。因此，当 CPU 需要时，故障这意味着页面不可用。当进程遇到页面错误时，所有线程都会冻结，直到内核处理这个错误。解决此问题的方法有多种，但最好的解决方案可能是调整源代码以避免页面错误。

次要页面错误

当进程在初始化部分内存前试图访问一部分内存时，会实时进行次要页面错误。在这种情况下，系统会执行操作来填充内存映射或其他管理结构。次要页面错误的严重性取决于系统负载和其他因素，但它们通常短，并有微不足的影响。

主页面错误

当系统必须将内存缓冲区与属于其他进程的磁盘、内存不足页面或者将任何其他输入输出(I/O)活动设置为可用内存时，会发生实时主要故障。当处理器引用没有为其分配物理页面的虚拟内存地址时，会出现这种情况。对空页面的引用会导致处理器执行错误，并指示内核代码分配可显著增加延迟的页面。

实时，当应用程序显示性能时，检查 **/proc/** 目录中与页面错误相关的进程信息是很有帮助的。对于特定的进程标识符(PID)，您可以使用 **cat** 命令查看 **/proc/PID/stat** 文件以了解以下相关条目：

- 字段 2：可执行的文件名。
- 字段 10：次要页面错误的数量。
- 字段 12：主页面错误的数量。

以下示例演示了使用 **cat** 命令和 **管道** 功能查看页面错误，以只返回 **/proc/PID/stat** 文件的第二个、10 和 twelfth 行：

```
# cat /proc/3366/stat | cut -d\ -f2,10,12
  (bash) 5389 0
```

在示例输出中，PID 3366 的进程是 bash，它有 5389 次要页面错误，也没有主要页面错误。

其他资源

- 由 Robert Love 的 Linux 系统编程

2.3. MLOCK 系统调用

RHEL for Real Time memory lock (**mlock （)**）系统调用可启用调用进程锁定或解锁指定的地址空间。这可防止 Linux 分页锁定的内存以 swap 空间。将物理页面分配给页表条目后，对该页面的引用相对较快。**mlock（）** 系统调用有两个组：**mlock（）** 和 **munlock（）** 调用。

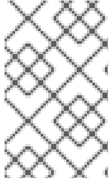
mlock（） 和 **munlock（）** 系统调用锁定并解锁特定进程地址页面。成功时，指定范围内的页面会保持在内存中常驻，直到 **munlock（）** 解锁页面。

mlock （） 和 **munlock （）** 调用以下参数：

- **addr**: 指定地址范围的开头。
- **len** ：指定地址空间长度（以字节为单位）。

成功时，**mlock（）** 和 **munlock（）** 返回 0。如果出现错误，则返回 -1 并设置 **errno** 来指示错误。

mlockall（） 和 **munlockall（）** 调用锁定或解锁整个程序空间。



注意

使用 **mlock（）** 不保证程序不会遇到页面 I/O。它确保数据保留在内存中，但无法确保它在同一页面中。虽然使用 **mlock（）**，但其他功能（如 **move_pages** 和 memory compactors）可以移动数据。

内存锁定页面会基于页进行，而不是堆栈。这意味着，如果两个动态分配的内存片段共享相同的页被调用 **mlock（）** 或 **mlockall（）** 锁定两次，则由一个调用将其解锁为 **munlock（）** 或 **munlockall（）**。因此，务必要了解应用程序解锁的页面，以防止双锁定或单一锁定问题。

减少双锁定或单一锁定问题的两个最常用的方案是：

- 跟踪已分配和锁定的内存区域，创建打包程序功能（在解锁页面之前），验证页面拥有的分配数量。这是设备驱动程序中使用的资源数原则。
- 执行分配考虑页面大小和协调，以防止在同一页面中出现双锁定。

其他资源

- **capabilities(7)** man page
- **mlock(2)** man page
- **mlock(3)** man page
- **mlockall(2)** man page
- **mmap(2)** man page
- **move_pages(2)** man page
- **posix_memalign(3)** man page
- **posix_memalign(3p)** man page

2.4. 共享库

用于 Real Time 的 RHEL 称为动态共享对象(DSO)，是预编译代码块的集合，称为功能。这些功能可在多个程序中重复使用，并在运行时加载或编译时间。

Linux 支持以下两个库类：

- 动态或共享库：作为可执行文件之外的独立文件存在。这些文件加载到内存中，并在运行时映射。
- 静态库：是在编译时静态链接到程序的文件。

ld.so 动态链路器加载程序所需的共享库，然后执行代码。然后，DSO 功能会通过映射到进程的地址空间来加载内存中的库，多个进程可以引用对象。您可以使用 **LD_BIND_NOW** 变量，将动态库配置为编译时加载。

在程序初始化前评估符号可以提高性能，因为如果内存页面位于外部磁盘上，则在运行时评估可能会导致延迟。

其他资源

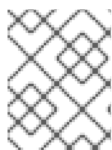
- **ld.so(8)** man page

2.5. 共享内存

在 RHEL for Real Time 中，共享内存是在多个进程间共享的内存空间。使用程序线程时，在一个进程上下文中创建的所有线程都可以共享相同的地址空间。这使得所有数据结构都可以被线程访问。通过 POSIX 共享内存调用，您可以将进程配置为共享地址空间的一部分。

您可以使用以下支持的 POSIX 共享内存调用：

- **shm_open（）**: 创建并打开新或打开现有的 POSIX 共享内存对象。
- **shm_unlink（）**: 取消链接 POSIX 共享内存对象。
- **mmap（）** ：在调用进程的虚拟地址空间中创建一个新的映射。



注意

使用 System V IPC **shmem（）** 集合的两个进程间共享内存区域的机制已弃用，在 RHEL for Real Time 上不再被支持。

其他资源

- **shm_open(3)** man page
- **shm_overview(7)** man page
- **mmap(2)** man page

第 3 章 RHEL FOR REAL TIME 的硬件中断

实时系统在其操作过程中收到许多中断，包括定期执行维护和系统调度决策的半常规"timer"中断。系统也可能收到特殊中断，如不可屏蔽中断(NMI)和系统管理中断(SMI)。设备使用硬件中断来指示需要注意的系统物理状态的变化。例如，其已读取一系列数据块的硬盘信号，或者当网络设备处理包含网络数据包的缓冲区时。

当实时发生中断时，系统会停止活动程序，并执行中断处理程序。

在实时中，硬件中断由中断数引用。这些数字被映射到创建中断的硬件片段。这可让系统监控创建中断的设备以及发生的时间。当实时发生中断时，系统会停止活动程序并执行中断处理程序。该处理程序会抢占其他正在运行的程序和系统活动。这会减慢整个系统并创建延迟。

RHEL for Real Time 修改处理中断的方式，以便提高性能并降低延迟。使用 **cat /proc/interrupts** 命令，您可以打印输出来查看发生的硬件中断类型、收到的中断数、中断的目标 CPU 以及生成中断的设备。

3.1. LEVEL-SIGNALED 中断

在实时中，level-signaled 中断使用提供自愿转换的专用中断行。设备控制器通过断言中断请求行上的信号来引发中断。中断行发送两个标记中的一个来代表二进制 1 或二进制 0。

当中断信号被行发送时，它会一直处于该状态，直到 CPU 重置为止。CPU 执行状态保存、捕获中断并分配中断处理程序。中断处理程序决定了中断的原因，通过执行必要的服务清除中断，同时恢复设备的状态。级别签名的中断更为可靠，并且支持多个设备，尽管它们比较复杂。

3.2. MESSAGE-SIGNALED 中断

在实时中，许多系统都使用消息签名中断(MSI)，它将信号发送为数据包或基于消息的电网上的专用消息。这种总线的常见示例是 Peripheral Component Interconnect Express (PCI Express 或 PCIe)。这些设备传输消息类型，PCIe 主机控制器将解析为中断消息。然后，主机控制器将 上的消息发送到 CPU。

在实时（根据硬件）中，PCIe 系统执行以下操作之一：

- 使用 PCIe 主机控制器和 CPU 之间的专用中断行发送信号。
- 通过 CPU HyperTransport 总线发送消息。

在实时中，PCIe 系统还可在传统模式下操作，在实施传统中断行以便支持旧操作系统或引导 Linux 内核时，在内核命令行中使用 **pci=nomsi** 选项 pci=nomsi。

3.3. 不可屏蔽中断

在实时中，不可屏蔽中断是系统中标准中断的硬件中断无法忽略。NMI 的优先级高于掩码中断。NMI 会出现信号，注意不可恢复的硬件错误。

在实时中，某些系统也使用 NMI 作为硬件监控器。当处理器收到 NMI 时，它会立即调用向中断向量指向的 NMI 处理程序来处理 NMI。如果满足某些条件，如在指定长度后不会触发中断，NMI 处理程序会向警告发出一个警告，并提供有关问题的调试信息。这有助于识别和防止系统锁定。

实时中断是硬件中断，通过在中断掩码注册位设置位来忽略这些中断。CPU 可以在关键处理期间暂时忽略可屏蔽中断。

3.4. 系统管理中断

实时，系统管理中断(SMI)提供扩展功能，如传统的硬件设备仿真，也可用于系统管理任务。smis 与不可屏蔽中断(NMI)相似，它们使用特殊的用电信号处理行，通常不可屏蔽。当发生 SMI 时，CPU 会进入系统管理模式(SMM)。在这个模式中，执行特殊的低级处理程序来处理 SMI。SMM 通常直接从系统管理固件提供，通常是 BIOS 或 EFI。

实时 SMI 最常用于提供传统硬件模拟。常见的例子是减少磁盘驱动器。如果没有附加 diskette 驱动器，操作系统会尝试访问磁盘，并触发 SMI。在这种情况下，处理器通过模拟设备提供操作系统。然后，操作系统将仿真视为旧设备。

在实时中，SMI 可能会对系统造成负面影响，因为它们无需直接参与操作系统。编写较差的 SMI 处理例程可能会消耗大量 CPU 时间，操作系统可能无法抢占处理程序。这可以在其他精心调整后创建定期高延迟，并会响应高度响应的系统。由于供应商可以使用 SMI 处理程序来管理 CPU 温度和 fan 控制，因此可能无法禁用它们。在这种情况下，您必须使用这些中断通知问题的供应商。

在实时中，您可以使用 **hwlatdetect** 程序隔离 SMI。它包括在 **rt-tests** 软件包中。这个实用程序测量 SMI 处理期间 CPU 使用的时间周期。

3.5. 高级可编程中断控制器

Intel 公司开发的高级可编程中断控制器(APIC)提供以下功能：

- 处理大量中断，将每个中断路由到一组特定的 CPU。
- 支持 CPU 间通信并移除多个设备共享单个中断行的需求。

实时 APIC 代表一系列设备和技术，它们以可扩展、可管理的方式生成、路由和处理大量硬件中断。它使用内置于每个系统 CPU 中的本地 APIC 的组合，以及多个直接连接到硬件设备的输入/输出 APIC。

在实时时，当硬件设备生成中断时，连接的 I/O APIC 会检测到并路由系统 APIC 总线到特定 CPU 的中断。操作系统知道 IO-APIC 连接到设备，并在该设备中中断行。高级配置和电源接口相关系统描述表(ACPI DSDT)包含有关主机系统的主板和外围程序组件和设备特定连接的信息，以及设备提供有关可用中断源的信息。两组数据一起提供整体中断层次结构的信息。

RHEL for Real Time 支持使用层次结构中连接的系统 APIC 基于 APIC 的中断管理策略，并通过负载均衡的方式向 CPU 提供中断，而不是针对特定 CPU 或一组 CPU。

第 4 章 RHEL 用于 REAL TIME 流程和线程

操作系统中的 RHEL for Real Time key factors 是最小中断延迟和最小线程切换延迟。虽然所有程序都使用线程和进程，但与标准 Red Hat Enterprise Linux 相比，RHEL for Real Time 会以不同的方式处理它们。

实时使用并行有助于提高任务执行和延迟的效率。并行是，当多个任务或多个子任务使用 CPU 的多核基础架构同时运行时。

4.1. PROCESS

实时进程在最简单的方面是执行中的程序。术语进程指的是一个独立地址空间，可能包含多个线程。当开发在一个地址空间中运行的多个进程的概念时，Linux 转向了一个与另一个进程共享地址空间的进程结构。这也可以正常工作，只要进程数据结构比较小。

UNIX® 风格的进程构造包含：

- 虚拟内存的地址映射。
- 执行上下文（PC、stack、注册）。
- 州和核算信息.

实时中，每个进程通过一个线程启动，通常称为父线程。您可以使用 **fork（）** 系统调用从父线程创建额外的线程。**fork （）** 创建新的子进程，与父进程相同，但新进程标识符除外。子进程独立于创建进程运行。父进程和子进程可以同时执行。**fork （）** 和 **exec （）** 系统调用之间的区别在于，**fork （）** 启动一个新进程，它是父进程的副本，**exec （）** 将当前进程替换为新进程镜像。

在实时中，**fork （）** 系统调用在成功时返回子进程的进程标识符，父进程返回非零值。在错误时，它会返回错误编号。

4.2. 线程

实时，一个进程中可以存在多个线程。进程的所有线程共享其虚拟地址空间和系统资源。线程是一个可调度的实体，其中包含：

- 程序计数器(PC)。
- 注册上下文。
- 堆栈指针。

在实时中，以下是创建并行性的潜在机制：

- 使用 **fork（）** 和 **exec（）** 函数调用来创建新进程。**fork （）** 调用会为进程创建一个名为 且具有唯一标识符的进程的精确重复。
- 使用 Posix 线程(**pthreads**)API 在已经运行的进程中创建新线程。

在对实时线程进行分叉前，您必须评估组件交互级别。当组件独立于另一个或交互时，创建新地址空间并将其作为新进程运行是有益的。当组件需要共享数据或经常通信时，在一个地址空间中运行线程会更高效。

实时中，**fork （）** 系统调用在成功时返回零值。在错误时，它会返回错误编号。

4.3. 其他资源

- **fork(2)** man page
- **exec(2)** man page

第 5 章 RHEL FOR REAL TIME 的应用程序时间戳

执行频繁 **时间戳的应用程序** 会受到读取时钟的 CPU 成本的影响。用于读取时钟的高成本和时间对应用程序性能造成负面影响。

您可以通过选择具有读取机制的硬件时钟（比默认时钟快）来降低读取时钟的成本。

在 RHEL for Real Time 中，使用带有 **clock_gettime（）** 函数的 POSIX 时钟可以获得更高的性能，从而生成时钟读取最低 CPU 成本。

对于使用硬件时钟以高读取成本使用硬件时钟的系统，这些优点更为明显。

5.1. 硬件时钟

在多处理器系统中发现的时钟源的多个实例，如非统一内存访问(NUMA)和 Symmetric 多进程(SMP)，在其自行交互，以及响应系统事件的方式，如 CPU 频率扩展或进入能源数，决定它们是否适合实时内核的时钟源。

首选时钟源是时间戳计数器(TSC)。如果 TSC 不可用，则高精度事件计时器(HPET)是第二个最佳选择。但是，并非所有系统都有 HPET 时钟，而有些 HPET 时钟可能不可靠。

如果没有 TSC 和 HPET，其他选项包括 ACPI Power Management Timer(ACPI_PM)、Programmable Interval Timer(PIT)和 Real Time Clock(RTC)。最后两个选项对于读取或具有低分辨率（时间粒度）很昂贵，因此与实时内核一起使用是更理想的选择。

5.2. POSIX 时钟

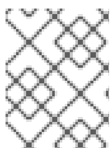
POSIX 是用于实施和表示时间源的标准。您可以将 POSIX 时钟分配给应用程序，而不影响系统中的其他应用程序。这与内核选择并在系统中实施的硬件时钟不同。

用于读取给定 POSIX 时钟的功能是 **clock_gettime（）**，它在 < **time.h&** gt; 中定义。kernel counterpart to **clock_gettime（）** 是一个系统调用。当用户进程调用 **clock_gettime（）** 时：

1. 对应的 C 库(**glibc**)调用 **sys_clock_gettime（）** 系统调用。
2. **sys_clock_gettime（）** 执行请求的操作。
3. **sys_clock_gettime（）** 将结果返回到用户程序。

但是，上下文从用户应用程序切换到内核具有 CPU 成本。尽管这种成本很低，但如果操作重复了数以千倍，但成本数量会给应用程序的整体性能造成影响。为了避免上下文切换到内核，因此可以更快地读取时钟，对 **CLOCK_MONOTONIC_COARSE** 和 **CLOCK_REALTIME_COARSE** POSIX 时钟的支持。

使用 **_COARSE** 时钟变体执行的 **clock_gettime（）** 读取的时间无需内核干预，且完全在用户空间中执行。这会产生显著的性能。对 **_COARSE** 时钟读取的时间有毫秒(ms)解析，这意味着不会记录小于 1ms 的时间间隔。POSIX 时钟的 **_COARSE** 变体适合任何可容纳几秒时钟解析的应用程序。



注意

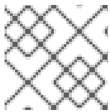
要将读取 POSIX 时钟的成本和解决方案与没有 **_COARSE** 前缀进行比较，请参阅 RHEL for Real Time 参考指南。

5.3. CLOCK_GETTIME() FUNCTION

以下代码演示了使用带有 **CLOCK_MONOTONIC_COARSE** POSIX 时钟的 **clock_gettime（）** 函数的代码示例：

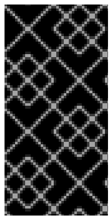
```
#include <time.h>
main()
{
 int rc;
 long i;
 struct timespec ts;

 for(i=0; i<10000000; i++) {
  rc = clock_gettime(CLOCK_MONOTONIC_COARSE, &ts);
 }
}
```

您可以通过添加检查来验证 **clock_gettime（）** 的值，验证 **rc** 变量的值，或者确保 **ts** 结构的内容是可信的。



注意

clock_gettime（） man page 提供了更多有关编写更可靠的应用程序的信息。



重要

使用 **clock_gettime（）** 函数的程序必须通过将 **-l rt** 添加到 **gcc** 命令行来与 rt 库相关联。

$ gcc clock_timing.c -o clock_timing -lrt

5.4. 其他资源

- **clock_gettime（）** man page
- 由 Robert Love 的 Linux 系统编程
- 了解 Daniel P. Bovet 和 Marco Cesati 的 Linux 内核

第 6 章 RHEL FOR REAL TIME 的调度策略

在实时中，调度程序是决定要执行的可运行线程的内核组件。每个线程都有关联的调度策略和静态调度优先级(**sched_priority**)。当具有较高静态优先级的线程就绪时，当前正在运行的线程会停止执行。然后，当前线程返回到其静态优先级的 **waitlist**。

所有 Linux 线程都有以下调度策略之一：

- **SCHED_OTHER** 或 'SCHED_NORMAL: 默认策略
- **SCHED_BATCH** ：与 **SCHED_OTHER** 类似，但会考虑吞吐量。
- **SCHED_IDLE**: 比 **SCHED_OTHER** 更低的优先级策略。
- **SCHED_FIFO** ：第一个位于 和第一个实时策略。
- **SCHED_RR** ：轮循实时策略。
- **SCHED_DEADLINE** ：一种调度程序策略，根据任务期限排列任务，最先运行最早的绝对期限。

6.1. 调度程序策略

在实时中，线程的优先级高于正常的线程。策略具有调度优先级值，范围从最小值为 1 到最大 99 值。

以下策略对实时至关重要：

- **SCHED_OTHER** 或 **SCHED_NORMAL**:
  这是 Linux 线程的默认策略。它有一个动态优先级，它根据线程的特性对系统进行更改。**SCHED_OTHER** 线程在 20（最高优先级）和 19（最低优先级）之间有 nice 值。默认情况下，SC **HED_OTHER** 线程的 nice 值为 0。
- **SCHED_FIFO** 策略
  带有 **SCHED_FIFO** 的线程，在 **SCHED_OTHER** 任务前运行。**SCHED_FIFO** 不使用 nice 值，而是使用 1（最低）和 99（最高）之间的固定优先级。优先级为 1 的 **SCHED_FIFO** 线程始终调度任何 **SCHED_OTHER** 线程。
- **SCHED_RR** 策略：
  SCHED_RR 策略与 **SCHED_FIFO** 策略类似。优先级相等的线程以轮循方式调度。**SCHED_FIFO** 和 **SCHED_RR** 线程运行，直到发生以下事件之一：
  - 线程进入 sleep 状态或等待事件。
  - 高优先级实时线程准备好运行。
    除非发生上述事件，线程在指定处理器上无限期运行，而较低优先级线程则停留在等待运行的队列中。这可能导致系统服务线程处于常驻，并防止交换出并失败文件系统数据冲刷。
- **SCHED_DEADLINE**:**SCHED_DEADLINE** 策略指定时间要求。它会根据任务的截止时间调度每个任务，即首先运行最早的截止时间的任务。
  内核要求 **runtimelatexdeadlineperiod** 为 true。所需选项之间的关系是 **runtime1028deadlineperiod**。

6.2. SCHED_DEADLINE 策略的参数

每个 **SCHED_DEADLINE** 任务都使用 **句点、运行时和 期限** 参数进行特征。这些参数的值是纳秒的整数。下表列出了这些参数。

表 6.1. SCHED_DEADLINE 参数

参数	描述
周期	**period** 是实时任务的激活模式。 例如，如果视频处理任务每秒有 60 个帧来处理，则每 16 毫秒的服务将排队新的帧，因此周期为 16 毫秒。
runtime	**运行时** 是分配给生成输出任务的 CPU 执行时间。实时执行上限，也称为"Worst Case Execution Time"(WCET)是 **运行时**。 例如，视频处理工具可能采用五毫秒来处理图像。因此，其运行时间为五毫秒。
deadline	**deadline** 是要生成输出的最长时间。 例如，如果任务需要在十毫秒内交付已处理的帧，则 **截止时间** 为十毫秒。

第 7 章 RHEL FOR REAL TIME 中的关联性

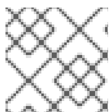
实时，系统中的每个线程和中断源都具有处理器关联属性。操作系统调度程序使用此信息来决定在哪些 CPU 上运行哪些线程和中断。

实时中的 Affinity 代表位掩码，掩码中的每个位代表一个 CPU 内核。如果位设置为 1，则线程或中断可以在该内核上运行；如果 0，则线程或中断将排除在核心上运行。关联性位掩码的默认值为 all，这意味着线程或中断可在系统中的任何内核上运行。

默认情况下，进程可以在任何 CPU 上运行。但是，可以通过更改进程的关联性来指示在预先确定的 CPU 上运行进程。子进程继承其父进程的 CPU 相关性。

一些比较典型的关联性设置包括：

- 为所有系统进程保留一个 CPU 核心，并允许应用程序在内核的其余部分中运行。
- 在同一 CPU 上允许线程应用程序和给定的内核线程（如网络 **softirq** 或驱动程序线程）。
- 每个 CPU 上的对制作者和消费者线程的对。



注意

关联性设置必须与 程序一起设计，才能实现良好的行为。

7.1. 处理器关联性

默认情况下，进程可以实时运行任意 CPU。但是，您可以通过更改进程的关联性，将进程配置为在预先确定的 CPU 上运行。子进程继承其父进程的 CPU 相关性。

在系统中调优操作的实际情况是确定运行应用程序所需的内核数量，然后隔离这些内核。这可以通过 Tuna 工具或 shell 脚本修改位掩码值来实现。

可以使用 **taskset** 命令来更改进程的关联性，修改 **/proc/** 文件系统条目会更改中断的关联性。使用 **taskset** 命令和 **-p** 或 **--pid** 选项，以及进程的进程标识符(PID)，检查进程的关联性。

-c 或 **--cpu-list** 选项显示内核的数字列表，而不是以位掩码形式显示。可以通过指定要绑定特定进程的 CPU 数量来设置关联性。例如，对于之前使用 CPU 0 或 CPU 1 的进程，您可以更改关联性，以便它只能在 CPU 1 上运行。除了 **taskset** 命令外，您还可以使用 **sched_setaffinity （）** 系统调用来设置处理器关联性。

其他资源

- **taskset(1)** man page
- **sched_setaffinity(2)** man page

7.2. SCHED_DEADLINE 和 CPUSETS

内核的截止时间调度类(**SCHED_DEADLINE**)在截止时间前实施第一次调度程序(EDF)，用于有有限期限的任务。它根据作业截止时间排列任务优先级：最早的绝对截止时间。除了 EDF 调度程序外，截止时间调度程序还实施恒定带宽服务器(CBS)。CBS 算法是一个资源保留协议。

CBS 保证每个任务在每个句点 **(T)** 期间接收其运行时 **(Q)**。每次激活任务时，CBS 会重新填充任务运行时间。在作业运行过程中，它会消耗 **其运行时**，如果任务没有 **运行时**，则任务会被节流和调度。节流机制可防止单个任务运行超过其运行时，有助于避免其他作业的性能问题。

实时，为了避免在 **截止时间** 任务的情况下过载系统，**截止时间调度程序** 会实施验收测试，每次任务被配置为 **使用截止时间调度程序** 运行时运行。接受的测试会保证 **SCHED_DEADLINE** 任务不使用超过 **kernel.sched_rt_runtime_us/kernel.sched_rt_period_us** 文件（默认为 950 ms）的 CPU 时间。

第 8 章 RHEL FOR REAL TIME 中的线程同步机制

同时，当两个或多个线程需要同时访问共享资源时，线程会协调使用线程同步机制。线程同步可确保一次只有一个线程使用共享资源。Linux 上使用的三个线程同步机制： Mutexes、Barriers 和 Condition 变量 (**condvars**)。

8.1. MUTEXES

mutex 从术语相互排除而出。相互排除对象同步对资源的访问。它是保证一个线程一次只能获取 mutex 的机制。

mutex 算法创建对每个代码的每个部分的串行访问，因此在任何时间点上只能有一个线程执行代码。Mutexes 使用称为 **mutex** 属性对象的属性对象来创建。它是一个抽象对象，其中包含几个取决于您选择的 POSIX 选项的属性。属性对象通过 **pthread_mutex_t** 变量定义。对象存储为 mutex 定义的属性。**pthread_mutex_init (&my_mutex, &my_mutex_attr)**, **pthread_mutexattr_setrobust （）** 和 **pthread_mutexattr_getrobust （）** 功能返回 0。在错误时，它们会返回错误号。

在实时中，您可以保留属性对象来初始化同一类型的更多 mutexes，也可以清理(destroy)属性对象。mutex 在这两种情况下都不受影响。Mutexes 包括标准和高级 mutexes 类型。

标准 mutexes

实时标准 mutexes 是私有、非递归、非攻击和非优先级继承能力 mutexes。使用 pthread_mutex_ **init(&my_mutex, &my_mutex_attr)初始化 pthread_mutex_ t** 会创建一个标准 mutex。使用标准 mutex 类型时，您的应用程序可能不从 **pthreads** API 和 RHEL for Real Time 内核提供的优势中获益。

高级 mutexes

使用额外功能定义的 Mutexe 称为高级 mutexes。高级功能包括优先级继承、mutex 的强大行为，以及共享和私有 mutexes。例如，对于强大的 mutexes，请初始化 **pthread_mutexattr_setrobust（）** 函数，可设置 robust 属性。同样，使用 **PTHREAD_PROCESS_SHARED** 属性允许任何线程在 mutex 上运行，只要线程可以访问其分配的内存。属性 **PTHREAD_PROCESS_PRIVATE** 设置私有 mutex。

非忙碌 mutext mutex 不会在手动发布前自动发布并保持锁定。

其他资源

- **futex(7)** man page
- **pthread_mutex_destroy(P)** man page

8.2. 障碍

与其他线程同步方法相比，障碍以非常不同的方式运行。障碍定义代码中所有活动线程停止的点，直到所有线程和进程达到这一障碍为止。当运行的应用程序需要确保所有线程在执行前完成特定任务时，可以使用障碍。

barrier mutex 实时使用以下两个变量：

- 第一个变量记录 barrier 的 **停止和** 传递 状态。
- 第二个变量记录了进入 barrier 的线程总数。

只有指定数量的线程数量达到定义的 barrier 时，barrier 才会设置状态。**当 barrier 状态设置为 通过 时，线程和进程将继续进行。pthread_barrier_init （） 函数分配所需的资源以使用定义的 barrier 并使用 attr 属性对象引用的属性进行初始化。**

当成功时，pthread_barrier_init （） 和 pthread_barrier_destroy （） 函数返回零值。在错误时，它们会返回错误号。

8.3. 条件变量

在实时中，条件变量(condvar)是一个 POSIX 线程构造，在继续前等待执行特定条件。通常，信号化条件与线程共享的数据状态相关。例如，condvar 可用于向处理队列发送数据条目，以及等待队列中处理该数据的线程。使用 pthread_cond_init （） 函数，您可以初始化条件变量。

当成功时，pthread_cond_init （）、pthread_cond_wait （） 和 pthread_cond_signal （） 函数返回零值。在错误时，它会返回错误编号。

8.4. MUTEX 类

下表列出了在编写或移植应用程序时需要考虑的 mutex 类。

表 8.1. mutex 选项

高级 mutexes	描述
shared mutexes	定义多个线程的共享访问，以在给定时间获得 mutex。共享 mutexes 可以创建高开销。此属性为 **PTHREAD_PROCESS_SHARED**。
私有 mutexes	确保仅在同一进程中创建的线程可以访问 mutex。属性是 **PTHREAD_PROCESS_PRIVATE**。
实时优先级继承	设置高于当前优先级优先级任务的低优先级任务的优先级级别。当任务完成后，它会释放资源，并且任务丢弃回其原始优先级，从而允许执行优先级更高的任务。属性是 **PTHREAD_PRIO_INHERIT**。
强大的 mutexes	当拥有线程终止时，将强大的 mutexes 设置为自动发布。字符串 **PTHREAD_MUTEX_ROBUST_ NP 中的值 substring NP** 表示强大的 mutexes 是非 POSIX 或不可移植

其他资源

- **futex(7) man page**

8.5. 线程同步功能

下表列出了用于线程同步机制的功能

表 8.2. Functions

功能	描述
pthread_mutexattr_init(&my_mutex_attr)	使用 **attr** 指定的属性启动 mutex。如果 **attr** 是 NULL，它会应用默认的 mutex 属性。
pthread_mutexattr_destroy(&my_mutex_attr)	销毁指定的 mutex 对象。您可以使用 **pthread_mutex_init（）** 重新初始化。
pthread_mutexattr_setrobust()	指定 mutex 的 robust 属性。
pthread_mutexattr_getrobust()	查询 mutex 的 robust 属性。
PTHREAD_MUTEX_ROBUST	定义一个线程，以在不解锁 mutex 的情况下终止。给您自己的该 mutex 的未来调用会自动成功，并返回值 **EOWNERDEAD** 以指示之前的 mutex 所有者不再存在。
pthread_barrier_init()	分配所需的资源以使用，并使用属性对象 **attr** 初始化 barrier。如果 **attr** 是 NULL，它会应用默认值。
pthread_cond_init()	初始化 condition 变量。**cond** 参数定义要使用 condition 变量属性对象 **attr** 中的属性启动的对象。如果 **attr** 是 NULL，它会应用默认值。
pthread_cond_wait()	阻止线程执行，直到它收到另一线程的信号。另外，对这个功能的调用也会在阻止前在 mutex 上释放相关的锁定。参数 **cond** 定义了要阻断的线程的 **pthread_cond_t** 对象。**mutex** 参数将 mutex 指定为 unblock。
pthread_cond_signal()	Unblocks 至少是在指定 condition 变量上阻止的线程之一。参数 **cond** 指定使用 **pthread_cond_t** 对象来取消阻塞线程。

第 9 章 用于 REAL TIME 的 RHEL 中的套接字选项

实时套接字是同一系统中两个进程（如 UNIX 域和回送设备或网络套接字等不同系统上）之间的数据传输机制。

传输控制协议(TCP)是最常用的传输协议，通常用于为需要持续持续通信的服务实现一致的低延迟，或者在低优先级限制的环境中对齐套接字。

借助新的应用程序、硬件功能和内核架构优化，TCP 必须引进新的方法来有效地处理更改。新方法可能会导致程序不稳定。因为程序的行为随着底层操作系统组件的变化而变化，所以必须小心处理。

TCP 中此类行为的一个示例是发送小缓冲区的延迟。这允许将它们作为网络数据包发送。缓冲区小的写入 TCP 并在所有时候都进行正常发送，但也可以创建延迟。对于实时应用程序，TCP_NODELAY socket 选项禁用延迟，并在它们就绪时立即发送小写入。

数据传输的相关套接字选项是 TCP_NODELAY 和 TCP_CORK。

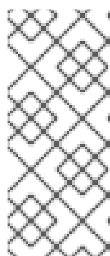
9.1. TCP_NODELAY SOCKET 选项

TCP_NODELAY socket 选项禁用 Nagle 的算法。使用 setsockopt socket API 功能配置 TCP_NODELAY，在各个数据包就绪时立即发送多个小的缓冲区写入。

通过在发送前先构建连续数据包，实现多个逻辑相关缓冲区作为单一数据包发送，从而获得更好的延迟和性能。或者，如果内存缓冲区在逻辑上关联但不连续，您可以创建 I/O 向量，并使用启用了 TCP_NODELAY 的套接字上的 writev 传递给内核。

以下示例演示了通过 setsockopt 套接字 API 启用 TCP_NODELAY。

```
int one = 1;
setsockopt(descriptor, SOL_TCP, TCP_NODELAY, &one, sizeof(one));
```



注意

要有效地使用 TCP_NODELAY，避免小、逻辑相关的缓冲区写入操作。使用 TCP_NODELAY 时，小型写入使 TCP 发送多个缓冲区作为单个数据包，这可能会降低整体性能。

其他资源

- **sendfile(2) man page**

9.2. TCP_CORK 套接字选项

TCP_CORK 选项收集套接字中的所有数据数据包，并防止传输它们，直到缓冲区填充到指定的限值为止。这可让应用程序在内核中构建数据包，并在禁用 TCP_CORK 时发送数据。TCP_CORK 使用 setsocketopt（） 函数在套接字文件描述符上设置。在开发程序时，如果必须从文件中发送批量数据，请考虑使用带有 sendfile（） 函数的 TCP_CORK。

当各种组件在内核中构建逻辑数据包时，使用 setsockopt socket API 将其配置为 1 来启用 TCP_CORK。这称为"展开套接字"。如果相应时间没有删除 cork，则 TCP_CORK 可能会导致错误。

以下示例演示了通过 setockopt 套接字 API 启用 TCP_CORK。

```
int one = 1;
setsockopt(descriptor, SOL_TCP, TCP_CORK, &one, sizeof(one));
```

在一些环境中，如果内核无法识别何时删除 cork，您可以按如下方式手动删除它：

```
int zero = 0;
setsockopt(descriptor, SOL_TCP, TCP_CORK, &zero, sizeof(zero));
```

其他资源

- **sendfile(2) man page**

9.3. 使用套接字选项的示例程序

TCP_NODELAY 和 TCP_CORK 套接字选项可显著影响网络连接的行为。TCP_NODELAY 禁用在应用程序就绪后立即发送数据数据包来实现的 Nagle 的算法。使用 TCP_CORK，您可以一起传输多个数据数据包，两者之间没有延迟。

示例程序提供影响这些套接字选项的性能信息。

对客户端的性能影响

您可以在不使用 TCP_NODELAY 和 TCP_CORK 套接字选项的情况下在客户端上发送小缓冲区写入。当不带参数运行 时，客户端将使用默认套接字选项。

- 要启动数据传输，请定义服务器 TCP 端口以及必须处理的数据包数量。例如，本测试中的 10,000 数据包。

```
$ ./tcp_nodelay_server 5001 10000
```

在所有情况下，它发送 15 个数据包，每个字符为 2 字节，并等待来自服务器的响应。

对回环接口的性能影响

以下示例使用回环接口来展示三种变化：

- 要立即发送缓冲区写入，请在使用 TCP_NODELAY 配置的套接字上设置 no_delay 选项。

```
$ ./tcp_nodelay_client localhost 5001 10000 no_delay

 10000 packets of 30 bytes sent in 1649.771240 ms: 181.843399 bytes/ms using
TCP_NODELAY
```

TCP 会立即发送缓冲区，从而禁用组合小数据包的算法。这提高了性能，但可能会导致为每个逻辑数据包发送小数据包。

- 要收集多个数据包并使用一个系统调用发送它们，请配置 TCP_CORK 套接字选项。

```
$ ./tcp_nodelay_client localhost 5001 10000 cork

 10000 packets of 30 bytes sent in 850.796448 ms: 352.610779 bytes/ms using
TCP_CORK
```

使用 cork 技术可显著降低发送数据包所需时间，因为它在其缓冲区中组合完整的逻辑数据包，并发送较少的总体网络数据包。您必须确保在适当的时间删除 cork。

在开发程序时，如果必须从文件中发送批量数据，请考虑使用带有 sendfile（） 选项的 TCP_CORK。

- 在不使用套接字选项的情况下测量性能。

```
$ ./tcp_nodelay_client localhost 5001 10000

10000 packets of 30 bytes sent in 400129.781250 ms: 0.749757 bytes/ms
```

当 TCP 组合缓冲区写入并等待检查比在网络数据包中最佳适合的数据外，这是基准测量。

其他资源

- sendfile(2) man page

第 10 章 RHEL FOR REAL TIME 调度程序

RHEL for Real Time 使用命令行实用程序可帮助您配置和监控进程配置。

10.1. 用来设置调度程序的 CHRT 工具

chrt 实用程序检查并调整调度程序策略和优先级。它可以启动具有所需属性的新进程，或更改正在运行的进程的当前属性。

chrt 实用程序采用 --pid 或 -p 选项指定进程 ID(PID)。

chrt 工具采用以下策略选项：

- -f 或 --fifo ：将计划设置为 SCHED_FIFO。
- -O 或 --other ：将计划设置为 SCHED_OTHER。
- -R 或 --rr ：将计划设置为 SCHED_RR。
- -d 或 --deadline ：计划设置为 SCHED_DEADLINE。

以下示例显示了指定进程的属性。

```
# chrt -p 468
pid 468's current scheduling policy: SCHED_FIFO
pid 468's current scheduling priority: 85
```

10.2. 抢占调度

实时抢占是一种临时中断执行任务的机制，希望稍后恢复它。当优先级较高的进程中断 CPU 用量时，会发生它。抢占可以对性能产生负面影响，并且持续抢占可能会导致状态化，称为"垃圾箱"。当进程不断抢占且任何进程需要完全运行时，会出现这个问题。更改任务的优先级有助于减少非自愿抢占功能。

您可以通过查看 /proc/PID/status 文件的内容，其中 PID 是进程标识符，以检查在单个进程上发生的自愿和非自愿抢占。

以下示例显示了 PID 为 1000 的进程的抢占状态。

```
# grep voluntary /proc/1000/status
voluntary_ctxt_switches: 194529
nonvoluntary_ctxt_switches: 195338
```

10.3. 调度程序优先级的库函数

实时进程使用一组不同的库调用来控制策略和优先级。该功能需要包含 sched.h 标头文件。必须在 sched.h 标头文件中定义 SCHED_OTHER、SCH_RR 和 SCHED_FIFO 的符号。

下表列出了为实时调度程序设置策略和优先级的功能。

表 10.1. 用于实时调度程序的库功能

Functions	描述
sched_getscheduler()	检索特定进程标识符(PID)的调度程序策略
sched_setscheduler()	设置调度程序策略和其他参数。此函数需要三个参数： **sched_setscheduler(pid_t pid** pid ,**int policy**,**const struct sched_param *sp);**
sched_getparam()	检索调度策略的调度参数。
sched_setparam()	设置与已设置的调度策略关联的参数，并可使用 **sched_getparam（）** 函数来验证。
sched_get_priority_max()	返回与调度策略关联的最大有效优先级。
sched_get_priority_min()	返回与调度策略关联的最低有效优先级。
sched_rr_get_interval()	显示每个进程的分配时间片。

第 11 章 RHEL FOR REAL TIME 中的系统调用

实时系统调用是应用程序程序用来与内核通信的功能。它是从内核订购资源的程序的机制。

11.1. SCHED_YIELD() FUNCTION

sched_yield（） 函数专为处理器设计，以选择除运行中的进程以外的进程。当从编写较差的应用程序中发布时，这种类型的请求容易出错。

当在具有实时优先级的进程中使用 sched_yield（） 函数时，它会显示意外行为。调用 sched_yield（） 的进程移动到以相同优先级运行的进程的尾部。如果没有以相同的优先级运行其他进程，名为 sched_yield（） 的进程将继续运行。如果该进程的优先级很高，则可能会创建忙碌循环，从而使机器无法使用。

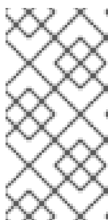
通常，不要在实时进程中使用 sched_yield（）。

11.2. GETRUSAGE（）函数

getrusage（） 函数从指定的进程或其线程中检索重要信息。报告信息，例如：

- 自愿和非自愿上下文切换的数量。
- 主要和次要页面错误。
- 使用的内存量。

getrusage（） 可让您查询应用程序，以提供与性能调优和调试活动相关的信息。getrusage（） 检索信息，否则需要从 /proc/ 目录中的几个不同文件进行目录，并且难以与应用上的特定操作或事件同步。



注意

并非所有填充 getrusage（） 结果的结构中包含的字段都由内核进行设置。其中一些仅出于兼容性的原因而保留。

其他资源

- **getrusage(2) man page**

第 12 章 在 RHEL FOR REAL TIME 上使用 MLOCK （）系统调用

用于 Real-Time 内存锁定(mlock（))函数的 RHEL 支持实时调用进程锁定或解锁指定的地址空间。这个范围可防止 Linux 在交换内存空间时分页锁定的内存。将物理页面分配给页表条目后，对该页面的引用会变得快速。mlock（） 系统调用包括两个功能： mlock（） 和 mlockall（）。同样，munlock（） 系统调用包含 munlock（） 和 munlockall（） 函数。

12.1. MLOCK （）和 MUNLOCK （）系统调用

mlock （） 和 mlockall （） 实时系统调用锁定指定的端口范围，不允许该内存被分页。有两个 mlock （） 系统调用组：

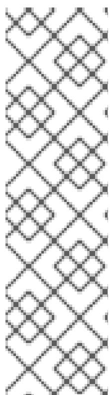
- mlock （） 调用：锁定指定的地址范围。
- munlock （） 调用：解锁指定的地址范围。

mlock （） 系统调用在地址范围中调用锁定页面，从 addr 开始并继续获取 len 字节。当调用成功时，包含指定地址范围一部分的所有页面都会保留在内存中，直到稍后解锁。

使用 mlockall（），您可以将所有映射的页面锁定到指定的地址范围中。这些都是代码的页、共享内存、内存映射文件等。内存锁定不堆栈。这意味着，被多个调用锁定的任何页面将解锁指定的地址范围或具有单个 munlock（） 调用的整个区域。通过 munlockall（） 调用，您可以解锁整个程序空间。

特定范围中包含的页面的状态取决于 标志参数中的值。标志 参数可以是 0 或 MLOCK_ONFAULT。

内存锁定不会被通过 fork 继承，并在进程终止时自动删除。



注意

请谨慎使用 mlock（）。过度使用可能会导致内存不足(OOM)错误。当某个应用程序较大，或者它有大的数据域时，当系统无法为其他任务分配内存时，mlock（） 调用可能会导致大量内存。

当对实时进程使用 mlockall（） 调用时，请确保保留了足够的堆栈页面。

12.2. 使用 MLOCK （）系统调用锁定页面

实时 mlock （） 调用使用 addr 参数指定地址范围的开头，并使用 len 定义地址空间长度（以字节为单位）。alloc_workbuf （） 函数动态分配内存缓冲区并锁定它。内存分配由 posix_memalig （） 函数执行，以将内存区域与页面匹配。功能 free_workbuf （） 解锁内存区域。

先决条件：

- 您有 root 权限或 CAP_IPC_LOCK 功能在大型缓冲区上使用 mlockall （） 或 mlock （）

流程

- 要使用 mlock（） 锁定页面，请运行以下命令：

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

void *alloc_workbuf(size_t size)
{
  void ptr;
  int retval;

  // alloc memory aligned to a page, to prevent two mlock() in the same page.
  retval = posix_memalign(&ptr, (size_t) sysconf(_SC_PAGESIZE), size);

  // return NULL on failure
  if (retval)
    return NULL;

  // lock this buffer into RAM
  if (mlock(ptr, size)) {
    free(ptr);
    return NULL;
  }
    return ptr;
 }

void free_workbuf(void *ptr, size_t size) {
  // unlock the address range
  munlock(ptr, size);

  // free the memory
  free(ptr);
}
```

验证

实时 mlock （） 和 munlock （） 调用成功时返回 0。如果出现错误，则返回 -1 并设置 errno 来指示错误。

12.3. 使用 MLOCKALL （）系统调用锁定所有映射的页面

要使用 mlockall （）和 munlockall （） 锁定和解锁实时内存，请将 标志 参数设置为 0 或恒定s: MCL_CURRENT 或 MCL_FUTURE。使用 MCL_FUTURE 时，将来的系统调用（如 mmap2 （）、sbrk2 （） 或 malloc3 （） ）可能会失败，因为它导致锁定的字节数超过允许的上限。

先决条件

- 有 root 权限。

流程

- 使用 mlockall （） 和 munlockall （） 实时系统调用：
  - 使用 mlockall （） 锁定所有映射的页面：

    ```
    #include <sys/mman.h>
    int mlockall (int flags)
    ```

  - 使用 munlockall （） 解除所有映射的页面：

    ```
    #include <sys/mman.h>
    int munlockall (void)
    ```

其他资源

- capabilities(7) man page
- mlock(2) man page
- mlock(3) man page

- **move_pages(2) man page**

- **posix_memalign(3) man page**

- **posix_memalign(3p) man page**

12.4. 使用 MMAP （）系统调用将文件或设备映射到内存

对于实时系统上的大型内存分配，内存分配(malloc)方法使用 mmap （） 系统调用来查找可寻址的内存空间。您可以通过在 标志 参数中设置 MAP_LOCKED 来分配和锁定内存区域。

作为 mmap（） 根据页分配内存，同一页面中没有两个锁定，这可以防止双锁定或单一锁定问题。

先决条件

- 有 root 权限。

流程

- 映射特定进程地址空间：

```
#include <sys/mman.h>
#include <stdlib.h>

void *alloc_workbuf(size_t size)
{
 void *ptr;

 ptr = mmap(NULL, size, PROT_READ | PROT_WRITE,
           MAP_PRIVATE | MAP_ANONYMOUS | MAP_LOCKED, -1, 0);

 if (ptr == MAP_FAILED)
  return NULL;

 return ptr;
}

void
free_workbuf(void *ptr, size_t size)
```

```
{
 munmap(ptr, size);
}
```

验证

- **在成功时，实时 mmap （） 调用会返回一个指向映射区域的指针。0n 错误，它会返回 MAP_FAILED 值，并设置 errno 来指示错误。**
- **当成功时，实时 munmap （） 调用会返回 0。0n 错误，它返回 -1 并设置 errno 来指明错误。**

其他资源

- **mmap(2) man page**
- **mlockall(2) man page**

12.5. MLOCK （）系统调用的参数

下表列出了 mlock（） 参数。

表 12.1. mlock 参数

参数	描述
addr	指定要锁定或解锁的进程地址空间。当 NULL 时，内核选择内存中数据的与页面对应的安排。如果 **addr** 不是 NULL，则内核会选择 nearby 页面边界，该边界始终高于 **/proc/sys/vm/mmap_min_addr** 文件中指定的值。
len	指定映射的长度，该映射必须大于 0。
fd	指定文件描述符。
prot	**mmap** 和 **munmap** 调用使用这个参数定义所需的内存保护。**Prot** 采用 **PROT_EXEC**、**PROT_READ**、**PROT_WRITE** 或 **PROT_NONE** 值的组合。

参数	描述
标记	控制 map相同文件的其他进程的映射可见性。它取一个值：**MAP_ANONYMOUS**、**MAP_LOCKED**、**MAP_PRIVATE** 或 **MAP_SHARED** 值。
MCL_CURRENT	锁定当前映射到进程中的所有页面。
MCL_FUTURE	设置模式来锁定后续内存分配。这些可能是增长的堆和堆栈、新内存映射文件或共享内存区域所需的新页面。

第 13 章 在 RHEL FOR REAL TIME 中设置 CPU 关联性

系统中的所有线程和中断源都具有处理器关联属性。操作系统调度程序使用此信息来确定要在 CPU 上运行的线程和中断。设置处理器关联以及有效策略和优先级设置可达到最大可能的性能。应用程序总是竞争资源，特别是 CPU 时间，与其他进程。根据应用程序，相关的线程通常会在同一内核上运行。另外，也可将一个应用程序线程分配给一个内核。

执行多任务处理的系统通常更容易地造成破坏。即使高优先级应用程序也可能延迟执行，较低优先级应用程序位于代码的关键部分。在低优先级应用程序退出关键部分后，内核会安全地抢占低优先级应用程序，并在处理器上调度高优先级应用程序。另外，由于缓存无效，将进程从一个 CPU 迁移到另一个 CPU 可能很昂贵。RHEL for Real Time 包括解决其中一些问题的工具，并允许更好地控制延迟。

关联性以位掩码表示，掩码中的每个位代表 CPU 内核。如果位设置为 1，则线程或中断可以在该内核上运行；如果 0，则线程或中断将排除在核心上运行。关联性位掩码的默认值为 all，这意味着线程或中断可在系统中的任何内核上运行。

默认情况下，进程可以在任何 CPU 上运行。但是，通过更改进程的关联性，您可以定义一个在一组预先确定的 CPU 上运行的进程。子进程继承其父进程的 CPU 相关性。

设置以下典型的关联性设置可实现最大可能的性能：

- 为所有系统进程使用单个 CPU 核心，并将应用程序设置为在内核的其余部分中运行。
- 在同一 CPU 中配置线程应用程序和特定内核线程（网络 softirq 或驱动程序线程）。
- 对每个 CPU 上的 producer-consumer 线程进行对。制作者和消费者是两类线程，生产者将数据插入到缓冲区中，并且消费者将其从缓冲区中删除。

在实时系统中调整相关合并的常见做法是确定运行应用程序所需的内核数，然后隔离这些内核。您可以使用 Tuna 工具或使用 shell 脚本修改位掩码值来达到此目的，如 taskset 命令。taskset 命令更改进程的关联性，修改 /proc/ 文件系统条目会更改中断的关联性。

13.1. 使用 TASKSET 命令调整处理器关联性

实时，taskset 命令有助于设置或检索正在运行的进程的 CPU 关联性。taskset 命令采用 -p 和 -c 选项。-p 或 --pid 选项会处理现有的进程，且不会启动新任务。-c 或 --cpu-list 指定了一个处理器数列表，

而不是一个位掩码。这个列表可以包含多个项目，用逗号分开，以及一系列处理器。例如： 0,5,7,9-11。

先决条件

- 有 root 权限。

流程

- 检查特定进程的进程关联性：

```
# taskset -p -c 1000
  pid 1000's current affinity list: 0,1
```

  该命令打印 PID 为 1000 的进程的关联性。进程配置为使用 CPU 0 或 CPU 1。

  - （可选）要配置特定 CPU 来绑定进程：

```
# taskset -p -c 1 1000
pid 1000's current affinity list: 0,1
pid 1000's new affinity list: 1
```

  - （可选）要定义多个 CPU 关联性：

```
# taskset -p -c 0,1 1000
pid 1000's current affinity list: 1
pid 1000's new affinity list: 0,1
```

  - （可选）在特定 CPU 上配置优先级级别和策略：

```
# taskset -c 5 chrt -f 78 /bin/my-app
```

    如需进一步粒度，您还可以指定优先级和策略。在示例中，命令使用 SCHED_FIFO 策略在 CPU 5 上运行 /bin/my-app 应用，优先级值为 78。

13.2. 使用 SCHED_SETAFFINITY（）系统调用设置处理器关联性

您还可以使用实时 sched_setaffinity （） 系统调用来设置处理器关联性。

前提条件

- 有 root 权限。

流程

- 使用 sched_setaffinity （） 设置处理器关联性：

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sched.h>

int main(int argc, char **argv)
{
  int i, online=0;
  ulong ncores = sysconf(_SC_NPROCESSORS_CONF);
  cpu_set_t *setp = CPU_ALLOC(ncores);
  ulong setsz = CPU_ALLOC_SIZE(ncores);

  CPU_ZERO_S(setsz, setp);

  if (sched_getaffinity(0, setsz, setp) == -1) {
    perror("sched_getaffinity(2) failed");
    exit(errno);
  }

  for (i=0; i < CPU_COUNT_S(setsz, setp); i) {
    if (CPU_ISSET_S(i, setsz, setp))
      online;
  }

  printf("%d cores configured, %d cpus allowed in affinity mask\n", ncores, online);
  CPU_FREE(setp);
}
```

13.3. 隔离单个 CPU 来运行高利用率任务

通过使用实时 cpusets 机制，您可以为 SCHED_DEADLINE 任务分配一组 CPU 和内存节点。在任务集中，包括使用任务的高和低 CPU，通过隔离 CPU 来运行高利用率任务，并在不同 CPU 集中调度较小的利用率任务，让所有任务能够满足分配的 运行时。

先决条件

- root 权限

流程

1. 创建两个目录，命名为 cpuset ：

```
# cd /sys/fs/cgroup/cpuset/
# mkdir cluster
# mkdir partition
```

2. 禁用 root Sharing 的负载平衡，以在 cpuset 目录中创建两个新的根域：

```
# echo 0 > cpuset.sched_load_balance
```

3. 在 cluster message 中，调度 CPU 1 上运行的低利用率任务到 7，验证内存大小，并将 CPU 命名为 exclusive:

```
# cd cluster/
# echo 1-7 > cpuset.cpus
# echo 0 > cpuset.mems
# echo 1 > cpuset.cpu_exclusive
```

4. 将所有低利用率任务移到 CREDENTIALS 目录中：

```
# ps -eLo lwp | while read thread; do echo $thread > tasks ; done
```

5. 创建命名为 cpuset 的分区，并分配高利用率任务：

```
# cd ../partition/
# echo 1 > cpuset.cpu_exclusive
# echo 0 > cpuset.mems
# echo 0 > cpuset.cpus
```

6. 将 shell 设置为 cpuset 并启动 deadline 工作负载：

```
# echo $$ > tasks
# /root/d &
```

通过这个设置，分区的 cpuset 目录中的任务隔离不会影响 clustertraffic 目录中的任务。这可让所有实时任务满足调度程序截止时间。