Red Hat Enterprise Linux for Real Time 9

为低延迟操作优化 RHEL 9

在 Red Hat Enterprise Linux 中配置 Linux 实时内核

Last Updated: 2024-04-05

Red Hat Enterprise Linux for Real Time 9 为低延迟操作优化 RHEL 9

在 Red Hat Enterprise Linux 中配置 Linux 实时内核

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

摘要

作为管理员，您可以在 RHEL for Real Time 内核中调优工作站。这些调整可提高性能，方便故障排除或者优化系统。

目录

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。

提交对特定段落的评论

1. 查看 **Multi-page HTML** 格式的文档，并确保在页面完全加载后看到右上角的 **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** 字段中输入您的改进建议。包括到文档相关部分的链接。
5. 点 **Submit Bug**。

第 1 章 RHEL 9 中的实时内核调整

延迟或响应时间定义为事件和系统响应之间的时间，通常以 microseconds(microseconds)衡量。

对于在 Linux 环境中运行的大多数应用程序，基本性能调优可以足够地提高延迟。对于延迟需要低、可负责且可预测的行业，红帽有一个可以调整的内核替换，以便延迟满足这些需求。RHEL for Real Time 9 提供与 RHEL 9 的无缝集成，并使客户端有机会在其机构中测量、配置和记录延迟时间。

RHEL for Real Time 9 旨在在精心调优的系统中用于具有极高确定性要求的应用程序。内核系统调优在确定性方面提供了大量改进。

开始之前，在使用 RHEL for Real Time 9 前，对标准 RHEL 9 系统进行常规系统调整。

警告

未能执行这些任务可能会阻止 RHEL Real Time 部署获得一致的性能。

1.1. 调整指南

- 实时调优是一个迭代过程；您几乎不会能够调整一些变量，知道这个变化是可以实现的最佳选择。准备花天或数周时间缩小最适合您的系统的调优配置集合。
  另外，请始终运行较长的测试。更改一些调优参数，然后执行五分钟的测试运行不是一组调优的良好验证。使测试的长度可以调整，并运行它们的时间已超过几分钟。尝试通过测试运行几个小时，缩小到几小时或几小时或天的几小时或天范围缩小，以便尝试和捕获最高延迟或资源耗尽的基点。

- 在应用程序中构建测量机制，以便您可以准确地衡量特定一组调整更改对应用程序性能的影响。Aecdotal 的证据（例如："鼠标更顺利。"）通常出错，与个人和个人不同。执行硬度并记录它们以便稍后进行分析。

- 在测试运行之间调整变量会非常方便，但这样做意味着您就无法有办法缩小影响您的测试结果。保持测试运行之间的调优更改尽可能小。

- 在调整时也试图进行大量更改，但最好进行增量更改。您会发现，按照从最低到最高优先级的值的工作方式将产生更好的结果。

- 使用可用的工具。借助 **tuna** 调优工具，可以轻松更改线程和中断的处理器关联、线程优先级和隔离处理器以使用应用程序。通过 **taskset** 和 **chrt** 命令行工具，您可以完成大多数 Tuna 的作用。如果遇到性能问题，**ftrace** 和 **perf** 工具可帮助查找延迟问题。

- 使用外部工具更改策略、优先级和关联性，而不是硬编码值。通过使用外部工具，您可以尝试许多不同的组合并简化您的逻辑。找到一些提供良好结果的设置后，您可以将它们添加到应用程序中，或者设置启动逻辑以在应用程序启动时实施设置。

1.2. 线程调度策略

Linux 使用三个主要线程调度策略。

- **SCHED_OTHER**（有时称为 **SCHED_NORMAL**）

这是默认的线程策略，并由内核控制动态优先级。优先级根据线程活动进行更改。具有此策略的线程被视为具有实时优先级 0（零）。

- **SCHED_FIFO** （最初为先出）
  优先级范围为 **1 - 99** 的实时策略，**1** 为最高，99%。**SCHED_FIFO** 线程始终具有高于 **SCHED_OTHER** 线程的优先级更高（例如，优先级为 **1** 的 **SCHED_FIFO** 线程将具有高于 *任何* **SCHED_OTHER** 线程的优先级）。作为 **SCHED_FIFO** 线程创建的任何线程都具有固定优先级，并将运行，直至被高优先级线程阻止或抢占。

- **SCHED_RR** (Round-Robin)
  SCHED_RR 是对 **SCHED_FIFO** 的修改。具有相同优先级的线程具有量量，并且是所有等同优先级 **SCHED_RR** 线程之间的轮循程序。此策略很少使用。

第 2 章 指定要运行的 RHEL 内核

您可以在启动过程中在 GRUB 菜单中手动选择所需内核，引导任何安装的内核、标准或 Real Time。您还可以将内核配置为默认引导。

安装 RHEL for Real Time 内核时，它会自动设置为默认内核，并在下次引导时使用。

2.1. 显示默认内核

您可以默认显示配置为引导的内核。

流程

- 查看默认内核：

```
# grubby --default-kernel

/boot/vmlinuz-kernel-rt-5.14.0-70.13.1.rt21.83.el9_0
```

命令的输出中的 **rt** 显示默认内核是实时内核。

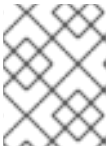
2.2. 显示运行的内核

您可以显示当前运行的内核

流程

- 显示系统当前正在运行的内核。

```
# uname -a
Linux rt-server.example.com kernel-rt-5.14.0-70.13.1.rt21.83.el9_0 …
```



注意

当系统收到次要更新（例如从 8.3 到 8.4）时，默认内核可能会自动从 Real Time 内核改回标准内核。

2.3. 配置默认内核

您可以配置默认引导内核。

流程

1. 列出已安装的 Real Time 内核。

```
# ls /boot/vmlinuz*rt*
/boot/vmlinuz-kernel-rt-5.14.0-70.13.1.rt21.83.el9_0
```

2. 将默认内核设置为列出的 Real Time 内核。

```
# grubby --set-default real-time-kernel
```

将 ***real-time-kernel*** 替换为 Real Time 内核版本。例如：

```
# grubby --set-default /boot/vmlinuz-kernel-rt-5.14.0-70.13.1.rt21.83.el9_0
```

验证步骤

- 显示默认内核：

```
# grubby --default-kernel
/boot/vmlinuz-kernel-rt-5.14.0-70.13.1.rt21.83.el9_0
```

第 3 章 运行并解释硬件和固件延迟测试

您可以通过在 RHEL Real Time 内核中运行 **hwlatdetect** 程序，测试并验证潜在的硬件平台是否适合实时操作。

先决条件

- 确保已安装 **RHEL-RT** （用于 Real Time）和 **rt-tests** 软件包。
- 检查供应商文档，了解低延迟操作所需的调整步骤。
  供应商文档提供了减少或删除将系统转换为系统管理模式(SMM)的任何系统管理中断(SMI)的说明。当系统位于 SMM 中时，它会运行固件而不是操作系统代码。这意味着，在 SMM 中过期的任何计时器等待系统转换到正常操作。这可能导致无法解释的延迟，因为 Linux 无法阻止 SMIs，我们实际占用 SMI 的唯一指示可在供应商相关的性能计数器寄存器中找到。

警告

红帽强烈建议您不要完全禁用 SMI，因为它可能会导致出现灾难性硬件故障。

3.1. 运行硬件和固件延迟测试

您不需要在运行 **hwlatdetect** 程序时在系统中运行任何负载，因为测试会查找硬件架构或 BIOS/EFI 固件带来的延迟。**hwlatdetect** 的默认值每秒轮询 0.5 秒，并在连续调用之间报告大于 10微秒的值来获取时间。**hwlatdetect** 返回系统中可能 **的最佳** 最大延迟。

因此，如果您有一个应用程序需要最大延迟值小于 10us，**hwlatdetect** 将报告其中一个差距为 20us，则系统只能保证延迟 20us。

注意

如果 **hwlatdetect** 显示系统无法满足应用程序的延迟要求，请尝试更改 BIOS 设置或与系统厂商合作来获取满足应用程序延迟要求的新固件。

先决条件

- 确保已安装 **RHEL-RT** 和 **rt-tests** 软件包。

流程

- 运行 **hwlatdetect**，指定测试持续时间（以秒为单位）。
  hwlat 通过轮询 clock-source 来查找硬件和固件导致的延迟，并寻找无法解释的差距。

```
# hwlatdetect --duration=60s
hwlatdetect:  test duration 60 seconds
 detector: tracer
 parameters:
  Latency threshold:    10us
  Sample window:        1000000us
```

```
  Sample width:         500000us
  Non-sampling period:  500000us
  Output File:          None

Starting test
test finished
Max Latency: Below threshold
Samples recorded: 0
Samples exceeding threshold: 0
```

其他资源

- **hwlatdetect** man page.
- 解释硬件和固件延迟测试

3.2. 解释硬件和固件延迟测试结果

这提供有关 **hwlatdetect** 程序的输出信息。

例子

- 以下结果代表了经过调优的系统，可最小化固件系统中断。在这种情况下，**hwlatdetect** 的输出如下所示：

```
# hwlatdetect --duration=60s
hwlatdetect:  test duration 60 seconds
 detector: tracer
 parameters:
  Latency threshold: 10us
  Sample window:     1000000us
  Sample width:      500000us
  Non-sampling period:  500000us
  Output File:       None

Starting test
test finished
Max Latency: Below threshold
Samples recorded: 0
Samples exceeding threshold: 0
```

- 以下结果代表了一个无法调整的系统，以便最大程度降低固件系统中断。在这种情况下，**hwlatdetect** 的输出如下所示：

```
# hwlatdetect --duration=10s
hwlatdetect:  test duration 10 seconds
 detector: tracer
 parameters:
  Latency threshold: 10us
  Sample window:     1000000us
  Sample width:      500000us
  Non-sampling period:  500000us
  Output File:       None
```

```
Starting test
test finished
Max Latency: 18us
Samples recorded: 10
Samples exceeding threshold: 10
SMIs during run: 0
ts: 1519674281.220664736, inner:17, outer:15
ts: 1519674282.721666674, inner:18, outer:17
ts: 1519674283.722667966, inner:16, outer:17
ts: 1519674284.723669259, inner:17, outer:18
ts: 1519674285.724670551, inner:16, outer:17
ts: 1519674286.725671843, inner:17, outer:17
ts: 1519674287.726673136, inner:17, outer:16
ts: 1519674288.727674428, inner:16, outer:18
ts: 1519674289.728675721, inner:17, outer:17
ts: 1519674290.729677013, inner:18, outer:17----
```

这个结果显示，在连续读取系统时钟源时，会在 15-18 us 范围内显示 10 延迟。

hwlatdetect 使用 tracer 机制来检测无法解释的延迟。

注意

之前的版本使用了内核模块而不是 **ftrace** tracer。

了解结果

输出显示测试方法、参数和结果。

表 3.1. 测试方法、参数和结果

参数	值	描述
测试持续时间	**10 秒**	测试的时间（以秒为单位）
Detector	**tracer**	运行 **检测** 器线程的工具
参数		
延迟阈值	**10us**	允许的最大延迟
示例窗口	**1000000us**	1 秒
width 示例	**500000us**	1/2 秒
非修改周期	**500000us**	1/2 秒
输出文件	**无**	输出保存到的文件。
结果		

参数	值	描述
最大延迟	**18us**	测试超过 **Latency threshold** 的最高延迟。如果没有超过 **Latency threshold** 的示例，则报告会显示 **Below threshold**。
记录的示例	**10**	测试记录的样本数。
超过阈值的样本	**10**	由测试记录的示例数量，其中延迟超过 **Latency threshold**。
运行期间的 smiss	**0**	测试运行期间发生的系统管理中断数(SMI)。

detector 线程运行可执行以下伪代码的循环：

```
t1 = timestamp()
 loop:
  t0 = timestamp()
  if (t0 - t1) > threshold
     outer = (t0 - t1)
  t1 = timestamp
  if (t1 - t0) > threshold
     inner = (t1 - t0)
  if inner or outer:
     print
  if t1 > duration:
     goto out
  goto loop
 out:
```

t0 是每个循环开始时的时间戳。**t1** 是每个循环末尾的时间戳。内循环比较检查 t0 - t1 不超过指定阈值(10 us default)。outer 循环比较循环底部和 top t1 - t0 之间的时间。最后一次读取时间戳寄存器之间的时间应该是十个纳秒（本质上是寄存器读取、比较和条件跳过），因此固件或系统组件连接方式导致连续读取之间的任何其他延迟。

注意

hwlatdetector 程序为内向和外发程序输出的值是最大延迟的最佳情况。延迟值是连续读取当前系统时钟源（通常是 Time Stamp Counter 或 TSC 注册）之间的增量，但也有可能是 HPET 或 ACPI 电源管理时钟，以及由硬件确认软件组合引入的延迟。

在找到合适的硬件确认组件组合后，下一步是测试系统的实时性能，同时存在负载。

第 4 章 运行并解释系统延迟测试

RHEL for Real Time 提供 Rt **eval** 工具来测试负载中的系统实时性能。

4.1. 先决条件

- 已安装 **RHEL for Real Time** 软件包组。
- 系统的 root 权限。

4.2. 运行系统延迟测试

您可以运行 **rteval** 工具来测试负载中的系统实时性能。

先决条件

- 已安装 **RHEL for Real Time** 软件包组。
- 系统的 root 权限。

流程

- 运行 **rteval** 工具。

```
# rteval
```

rteval 程序启动一个重量系统负载的 **SCHED_OTHER** 任务。然后，它会测量每个在线 CPU 的实时响应。负载是循环中 Linux 内核树和 **hackbench** 合成基准的并行创建。

目标是让系统进入一个状态，每个内核始终都有一个要调度的作业。作业执行各种任务，如内存分配/免费、磁盘 I/O、计算任务、内存复制等。

当负载启动后，rteval 会启动 cyclictest 测量程序。该程序在每个在线内核上启动 SCHED_FIFO 实时线程。然后，它会测量实时调度响应时间。

每个测量线程都需要一个时间戳，睡眠间隔，然后在启动后使用另一个时间戳。测量延迟是 t1 -(t0 + i)，这是实际生成时间 t1 之间的差值值，以及第一个时间戳 t0 以及 sleep 间隔 i 的理论唤醒时间。

rteval 运行的详情将写入到 XML 文件中，以及系统的引导日志。此报告显示在屏幕上，并保存到压缩文件中。

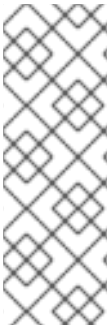
文件名采用 rteval- <*date*>-*N*-tar.bz2 格式，其中 < date> 是报告生成的日期，*N* 是 < *date* > 上运行的 Nth 计数器。

以下是一个 rteval 报告示例：

```
System:
Statistics:
 Samples:          1440463955
 Mean:             4.40624790712us
 Median:           0.0us
 Mode:             4us
 Range:            54us
 Min:              2us
 Max:              56us
 Mean Absolute Dev: 1.0776661507us
 Std.dev:          1.81821060672us

CPU core 0       Priority: 95
Statistics:
 Samples:          36011847
 Mean:             5.46434910711us
 Median:           4us
 Mode:             4us
 Range:            38us
 Min:              2us
 Max:              40us
 Mean Absolute Dev: 2.13785341159us
 Std.dev:          3.50155558554us
```

这个报告包括了系统硬件、已用运行长度以及时间结果（每个cpu 和系统范围的）的详情。



注意

要从其生成的文件中重新生成 rteval 报告，请运行

 # rteval --summarize rteval-*<date>*-*N*.tar.bz2

第 5 章 设置持久性内核调整参数

当您决定决定适用于您的系统的调优配置后，您可以在重启后保留更改。

默认情况下，编辑的内核调整参数仅在系统重启或参数被显式更改后才会生效。这可用于建立初始调优配置。它还提供了一种安全机制。如果编辑的参数导致机器被错误处理，重启机器会将参数返回到前面的配置。

5.1. 进行持久性内核调整参数更改

您可以通过在 /etc/sysctl.conf 文件中添加 参数，对内核调整参数进行持久性更改。

注意

这个过程 *不会改变* 当前会话中的任何内核调整参数。输入的 /etc/sysctl.conf 的更改只会影响将来的会话。

先决条件

- 根权限

流程

1. 在文本编辑器中打开 /etc/sysctl.conf。
2. 使用参数值将新条目插入到文件中。

   通过删除 /proc/sys/ 路径来修改参数名称，将剩余的斜杠(/)更改为句点(.)以及包含参数的值。

   例如，要使命令 echo 0 > /proc/sys/kernel/hung_task_panic 持久，请在 /etc/sysctl.conf 中输入以下内容：

```
# Enable gettimeofday(2)
kernel.hung_task_panic = 0
```

3. 保存并关闭该文件。

4. 重启系统以使更改生效。

验证

- 验证配置：

```
~]# cat /proc/sys/kernel/hung_task_panic
0
```

第 6 章 通过避免运行不必要的应用程序来提高性能

每个正在运行的应用程序都使用系统资源。确保系统上没有不必要的应用程序可以显著提高性能。

先决条件

- 系统的 root 权限。

流程

1. 不要运行 图形界面，而不绝对要求，特别是在服务器上。

   检查系统是否默认配置为引导到 GUI：

   ```
   # systemctl get-default
   ```

2. 如果命令的输出是 graphical.target，请将系统配置为引导进入文本模式：

   ```
   # systemctl set-default multi-user.target
   ```

3. 除非您要在系统上使用 邮件传输代理(MTA)，否则请禁用它。如果需要该 MTA，请确保它经过精心调整或考虑将其移动到专用计算机。

   如需更多信息，请参阅 MTA 的文档。

   重要

   使用 MTA 发送系统生成的信息，这些消息由 cron 等程序执行。这包括日志记录功能（如watch ）生成的 报告。如果您计算机上的 MTA 已禁用，您将无法接收这些邮件。

4. 外设设备，如 mice、keyboards、webcams 发送可能会对延迟造成负面影响的中断。如果您不使用图形界面，请删除所有未使用的外围设备并禁用它们。

   如需更多信息，请参阅设备文档。

5. 检查可能会影响性能的自动 cron 作业。

```
# crontab -l
```

禁用 crond 服务或任何不需要的 cron 作业。

如需更多信息，请参阅 CRON(8) man page。

6. 检查您的系统中的第三方应用程序和外部硬件供应商添加的任何组件，并删除所有不需要的任何组件。

第 7 章 由于日志而减少或避免系统变慢的问题

日志更改写入到磁盘的顺序可能与它们到达的顺序不同。内核 I/O 系统可重新排序日志更改，以优化使用可用的存储空间。日志活动可以通过重新排序日志更改并提交数据和元数据导致系统延迟。因此，日志文件系统可能会降低系统速度。

XFS 是 RHEL 8 使用的默认文件系统。这是一个日志文件系统。名为 ext2 的较旧文件系统不使用日志。除非您的机构特别需要日志，请考虑使用 ext2。许多红帽最佳基准结果都使用 ext2 文件系统。这是顶级初始调优建议之一。

XFS 等日志记录文件系统，记录文件最后一次访问的时间（时间）。如果您需要使用日志记录文件系统，请考虑禁用 时间。

7.1. 禁用一个时间

禁用 atime 通过限制写入文件系统日志数量来提高性能并降低功耗。

流程

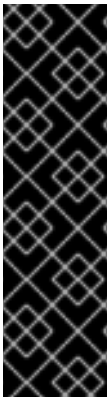
禁用 atime ：

1. 使用您所选的文本编辑器打开 /etc/fstab 文件，并找到 root 挂载点的条目。

```
/dev/mapper/rhel-root       /       xfs    defaults&hellip;
```

2. 编辑选项部分，使其包含术语 noatime 和 nodiratime。noatime 选项可防止在读取文件时更新时间戳，nodiratime 选项会停止更新目录内节点访问时间。

```
/dev/mapper/rhel-root       /       xfs    noatime,nodiratime&hellip;
```



重要

有些应用程序依赖于 更新时间。因此，这个选项只在没有使用此类应用程序的系统中才有意义。

或者，您可以使用 relatime 挂载选项，它将确保在之前访问时间早于当前修改时间时才更新访问时间。

7.2. 其他资源

- mkfs.ext2(8) man page
- mkfs.xfs(8) man page
- mount(8)man page

第 8 章 禁用延迟敏感负载的图形控制台输出

内核会在启动信息后立即向 printk 传递信息。内核向日志文件发送消息，即使没有附加到无头服务器的监控器，也会在图形控制台中显示消息。

在一些系统中，发送到图形控制台的输出可能会在管道中引入 stalls。这可能会在等待数据传输时导致任务执行的潜在延迟。例如，发送到 teletype0 (/dev/tty0) 的输出可能会在某些系统中造成潜在的停滞。

要防止意外的 stall，您可以限制或禁用向图形控制台发送的信息：

- 删除 tty0 定义。
- 更改控制台定义的顺序。
- 关闭大多数打印 功能 并确保将 ignore_loglevel 设置为 未配置。

本节包含防止图形控制台登录图形适配器并控制在图形控制台中打印的信息的步骤。

8.1. 禁用图形控制台日志记录到图形适配器

teletype (tty) （默认内核控制台）通过将输入数据传递至系统并显示在图形控制台上的输出信息来启用与系统的交互。

未配置图形控制台，防止其在图形适配器中登录。这使得 tty0 对系统不可用，并帮助在图形控制台中禁用打印信息。

注意

禁用图形控制台输出不会删除信息。这个信息会在系统日志中打印，您可以使用 journalctl 或 dmesg 工具访问这些信息。

流程

1. 打开 /etc/sysconfig/grub 文件。

2. 从 GRUB_CMDLINE_LINUX 密钥中删除 console=tty0 值。

3. 运行 grub2-mkconfig 命令，以重新生成 /boot/grub2/grub.cfg 文件：

```
# grub2-mkconfig -o /boot/sysconfig/grub2/grub.cfg
```

grub2-mkconfig 命令收集新配置更改，并重新生成 /boot/grub2/grub.cfg 文件。

8.2. 禁用在图形控制台上打印的消息

您可以通过在 /proc/sys/kernel/printk 文件中配置所需的日志级别，来控制发送到图形控制台的输出信息量。

流程

1. 查看当前的控制台日志级别：

```
$ cat /proc/sys/kernel/printk
 7    4    1    7
```

该命令打印系统日志级别的当前设置。该数字对应于系统日志记录器的当前、默认、最小值和引导默认值。

2. 在 /proc/sys/kernel/printk 文件中配置所需的日志级别。

```
$ echo “1” > /proc/sys/kernel/printk
```

命令更改当前的控制台日志级别。例如，设置日志级别 1 将只打印警报信息并防止在图形控制台中显示其他消息。

第 9 章 管理系统时钟以满足应用程序需求

NUMA 或 SMP 等多处理器系统具有多个硬件时钟实例。在引导期间，内核发现可用的时钟源并选择要使用的时钟源。要提高性能，您可以更改用于满足实时系统最低要求的时钟源。

9.1. 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。

9.2. 查看系统中可用的时钟源

系统中可用时钟源列表位于 /sys/devices/system/clocksource/clocksource0/available_clocksource 文件中。

流程

- 显示 available_clocksource 文件。

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
```

在这个示例中，系统中的可用时钟源为 TSC、HPET 和 ACPI_PM。

9.3. 查看当前正在使用的时钟源

系统中的当前使用的时钟源保存在 /sys/devices/system/clocksource/clocksource0/current_clocksource 文件中。

流程

- 显示 current_clocksource 文件。

```
# cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

在这个示例中，系统中的当前时钟源是 TSC。

9.4. 临时更改要使用的时钟源

有时因为时钟中已知问题，不使用系统主应用程序的最佳时钟。处理所有有问题的时钟后，系统就可以使用无法满足实时系统最低要求的硬件时钟。

对关键应用程序的要求因各系统而异。因此，每个应用程序的最佳时钟也会不同。有些应用程序依赖于时钟分辨率，提供可靠纳秒读的时钟更合适。最多读取时钟的应用程序往往可以从具有较小读成本的时钟中受益（读取请求和结果之间的时间）。

在这些情况下，可以覆盖内核所选时钟，只要您了解这个覆盖的副作用，并可创建一个环境，不会触发给定硬件时钟已知短语。

重要

内核自动选择最佳的时钟源。不建议覆盖所选时钟源，除非很理解了影响。

先决条件

- 系统上的 root 权限。

流程

1. 查看可用的时钟源。

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
```

在这个示例中，系统中的可用时钟源为 TSC、HPET 和 ACPI_PM。

2. 将您要使用的时钟源名称写入 /sys/devices/system/clocksource/clocksource0/current_clocksource 文件。

```
# echo hpet > /sys/devices/system/clocksource/clocksource0/current_clocksource
```

注意

这个过程更改了当前正在使用的时钟源。系统重启时，会使用默认时钟。要使更改具有持久性，请参阅 进行持久性内核性能优化参数更改。

验证步骤

- 显示 current_clocksource 文件，以确保当前时钟源是指定的时钟源。

```
# cat /sys/devices/system/clocksource/clocksource0/current_clocksource
hpet
```

在这个示例中，系统中的当前时钟源是 HPET。

9.5. 读取硬件时钟源的成本比较

您可以比较系统中时钟的速度。从 TSC 读取涉及从处理器读取寄存器。从 HPET 时钟读取涉及读取内存区域。从 TSC 读取速度较快，当每秒时间戳数千条消息时，它提供显著的性能优势。

先决条件

- 系统上的 root 权限。
- clock_timing 程序必须位于系统中。如需更多信息，请参阅 clock_timing 程序。

流程

1. 更改到保存了 clock_timing 程序的目录。

```
# cd clock_test
```

2. 查看系统中可用的时钟源。

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
```

在这个示例中，系统中的可用时钟源为 TSC、SpeT、Spe T 和 ACPI_PM。

3. 查看当前使用的时钟源。

```
# cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

在这个示例中，系统中的当前时钟源是 TSC。

4. 与 ./clock_timing 程序一起运行 时间 实用程序。输出中会显示读取时钟源 1,000万次所需的时间。

```
# time ./clock_timing

real 0m0.601s
user 0m0.592s
sys 0m0.002s
```

以下示例显示了以下参数：

- 实时 - 从程序调用开始的总时间，直到进程结束。real 包括用户和内核时间，通常大于后面的两个的总和。如果这个过程是由具有更高优先级的应用程序中断，或者由硬件中断（IRQ 等系统事件等系统事件中断，这的时间也被等待计算 在实际 下。

- user - 进程在用户空间内花费的时间执行不需要内核干预的任务。

- sys - 内核在执行用户进程所需的任务时使用的时间。这些任务包括打开文件、读取和写入文件或 I/O 端口、内存分配、线程创建和网络相关活动。

5. 将您要测试的下一个时钟源的名称写入 /sys/devices/system/clocksource/clocksource0/current_clocksource 文件。

```
# echo hpet > /sys/devices/system/clocksource/clocksource0/current_clocksource
```

在这个示例中，当前的时钟源被改为 HPET。

6. 对于所有可用时钟源，重复步骤 4 和 5。

7. 比较第 4 步获得所有可用时钟源的结果。

其他资源

- time(1) man page

9.6. CLOCK_TIMING 程序

clock_timing 程序读取当前的时钟源 1,000万次。与 时间 实用程序结合使用，它测量执行此操作所需的时间。

流程

创建 clock_timing 程序：

1. 为程序文件创建一个目录。

```
$ mkdir clock_test
```

2. 更改到创建的目录。

```
$ cd clock_test
```

3. 创建一个源文件，并在文本编辑器中打开。

```
$ vi clock_timing.c
```

4. 在文件中输入以下内容：

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

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

5. **保存文件并退出编辑器。**

6. **编译 文件。**

```
$ gcc clock_timing.c -o clock_timing -lrt
```

clock_timing 程序已就绪，可在保存它的目录中运行。

第 10 章 控制电源管理转换

您可以控制电源管理转换来改进延迟。

先决条件

- 系统的 root 权限。

10.1. 节能状态

现代处理器主动从低状态转变为更高的节能状态(C-states)。不幸的是，从高节能状态转换到正在运行的状态会消耗的时间比实时应用程序的最佳优化要多。为防止这些转换，应用程序可以使用电源管理服务质量(PM QoS)接口。

使用 PM QoS 接口，系统可以模拟 idle=poll 和 processor.max_cstate=1 参数的行为，但具有更精细的节能状态控制。idle=poll 可防止处理器进入 空闲状态。processor.max_cstate=1 可防止处理器进入更深入的 C-states（energy-save 模式）。

当应用程序包含 /dev/cpu_dma_latency 文件时，PM QoS 接口可防止处理器进入 deep sleep 状态，这会导致在退出时造成意外的延迟。当文件关闭时，系统会返回到节能状态。

10.2. 配置电源管理状态

您可以将值写入 /dev/cpu_dma_latency 文件，以更改进程的最大响应时间（以微秒为单位）。您还可以在应用程序或脚本中引用此文件。

先决条件

- 系统上的 root 权限。

流程

1. 打开 /dev/cpu_dma_latency 文件。在 low-latency 操作期间，使文件描述符保持打开。

2. 向文件写入 32 位数字。此数值代表了微秒内的最大响应时间。对于可能的响应时间，请使用 0。

示例

以下是使用此方法防止电源转换和维护低延迟的程序示例。

```
main()

static int pm_qos_fd = -1;

void start_low_latency(void)
{
 s32_t target = 0;

 if (pm_qos_fd >= 0)
  return;
 pm_qos_fd = open("/dev/cpu_dma_latency", O_RDWR);
 if (pm_qos_fd < 0) {
   fprintf(stderr, "Failed to open PM QOS file: %s",
          strerror(errno));
   exit(errno);
 }
 write(pm_qos_fd, &target, sizeof(target));
}

void stop_low_latency(void)
{
 if (pm_qos_fd >= 0)
   close(pm_qos_fd);
}
```

10.3. 其他资源

- 基于 Robert Love 的 Linux 系统编程.

第 11 章 为系统调整设置 BIOS 参数

这部分包含有关您可以配置的各种 BIOS 参数的信息，以改进系统性能。

注意

每个系统和 BIOS 供应商都使用不同的术语和导航方法。因此，本节只包含有关 BIOS 设置的一般信息。

如果您需要帮助查找特定设置，请检查 BIOS 文档或联系 BIOS 供应商。

11.1. 禁用电源管理以提高响应时间

BIOS 电源管理选项有助于通过更改系统时钟频率或者将 CPU 放入各种睡眠状态之一来节省电源。这些操作可能会影响系统对外部事件的响应速度。

要提高响应时间，禁用 BIOS 中的所有电源管理选项。

11.2. 通过禁用错误检测和修正单元来提高响应时间

错误检测和检测(EDAC)单元是用于检测和更正从错误更正代码(ECC)内存错误的设备。通常，EDAC 选项的范围是从没有 ECC 检查对所有内存节点进行定期扫描以找出错误。EDAC 级别越高，BIOS 使用的时间就越长。这可能导致缺少关键事件期限。

要提高响应时间，请关闭 EDAC。如果无法做到这一点，将 EDAC 配置为最低功能级别。

11.3. 通过配置系统管理中断提高响应时间

系统管理中断(SMI)是一个硬件厂商工具，用来确保系统正常工作。BIOS 代码通常服务 SMI 中断。smis 通常用于rmal 管理、远程控制台管理(IPMI)、EDAC 检查和各种其他日常任务。

如果 BIOS 包含 SMI 选项，请检查厂商以及任何相关文档，以确定其安全禁用它们的范围。

警告

虽然可以完全禁用 SMIs，但红帽强烈建议您不要这样做。删除您的系统生成和服务 SMIs 可能会导致灾难性硬件故障。

第 12 章 通过隔离中断和用户进程来最小化系统延迟

在响应各种事件时，需要尽量减少或消除延迟。为此，您可以将中断(IRQ)与不同专用 CPU 上的另一个进程隔离中断(IRQ)。

12.1. 中断和进程绑定

与不同 CPU 上的用户进程隔离中断(IRQ)可以最小化或消除实时环境中的延迟。

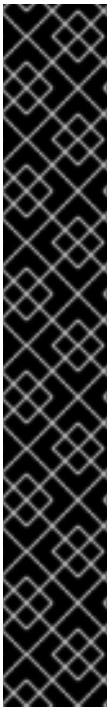
中断通常在 CPU 之间均匀共享。当 CPU 必须写入新数据和指令缓存时，这可能会延迟中断处理。这些中断延迟可能会导致与在同一 CPU 上执行的其他处理冲突。

可以将时间关键中断和进程分配给特定的 CPU（或一系列 CPU）。这样，处理此中断的代码和数据结构最有可能在处理器和指令缓存中。因此，专用进程可以尽快运行，而所有其他非关键进程在其他 CPU 上运行。这特别重要：涉及的速度接近或位于内存限制和可用的外围总线带宽。任何等待内存被放入处理器缓存时，对总处理时间和确定性的影响都会有明显影响。

在实践中，最佳性能完全特定于应用程序。例如，使用类似公司的功能调整应用程序需要完全不同的性能优化。

- 当其中一个公司隔离了 4 个用于操作系统功能的 CPU 和中断处理时，他们就会发现最佳结果。剩余的 2 个 CPU 仅适用于应用程序处理。

- 当另一公司将网络相关应用程序进程绑定到处理网络设备驱动程序中断的单一 CPU 中时，他们发现了最佳决定。



重要

要将进程绑定到 CPU，通常需要知道给定 CPU 或 CPU 范围的 CPU 掩码。CPU 掩码通常表示为 32 位位掩码、十进制数或十六进制数字，具体取决于您使用的命令。

表 12.1. 例子

CPU	Bitmask	十进制	十六进制
0	00000000000000000000000000000001	1	0x00000001
0, 1	00000000000000000000000000000011	3	0x00000011

12.2. 禁用 IRQBALANCE 守护进程

irqbalance 守护进程默认启用，并定期以同样的方式强制 CPU 处理中断。但是，在实时部署中不需要 irqbalance，因为应用程序通常会绑定到特定的 CPU。

流程

1. 检查 irqbalance 的状态。

```
# systemctl status irqbalance
irqbalance.service - irqbalance daemon
  Loaded: loaded (/usr/lib/systemd/system/irqbalance.service; enabled)
  Active: active (running) ...
```

2. 如果 irqbalance 正在运行，请禁用它并停止它。

```
# systemctl disable irqbalance
# systemctl stop irqbalance
```

验证

- 检查 irqbalance 的状态是否不活跃。

```
~]# systemctl status irqbalance
```

12.3. 将 CPU 从 IRQ 平衡中排除

您可以使用 IRQ 平衡服务指定要排除的 CPU，以便考虑中断(IRQ)平衡。/etc/sysconfig/irqbalance 配置文件中的 IRQBALANCE_BANNED_CPUS 参数控制这些设置。参数的值是一个 64 位十六进制数掩码，掩码的每个位代表 CPU 内核。

流程

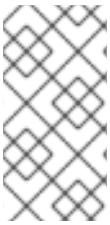
1. 在首选的文本编辑器中打开 /etc/sysconfig/irqbalance，并查找标题为 IRQBALANCE_BANNED_CPUS 的文件部分。

```
# IRQBALANCE_BANNED_CPUS
# 64 bit bitmask which allows you to indicate which cpu's should
# be skipped when reblancing irqs. Cpu numbers which have their
# corresponding bits set to one in this mask will not have any
# irq's assigned to them on rebalance
#
#IRQBALANCE_BANNED_CPUS=
```

2. 取消注释 IRQBALANCE_BANNED_CPUS 变量。

3. 输入适当的位掩码，以指定 IRQ 平衡机制要忽略的 CPU。

4. 保存并关闭该文件。



注意

如果您正在运行最多 64 个 CPU 内核的系统，请使用逗号分隔八个十六进制数。例如：IRQBALANCE_BANNED_CPUS=00000001,0000ff00

表 12.2. 例子

CPU	Bitmask
0	00000001
8 - 15	0000ff00
8 - 15, 33	00000001,0000ff00

注意

在 RHEL 7.2 及更高版本中，如果 /etc/sysconfig/ irqbalance 中未设置 IRQ，则 irqbalance 程序会在 /etc/sysconfig/irqbalance 中通过 isolcpus 内核参数隔离。

12.4. 手动将 CPU 关联性分配给单独的 IRQ

分配 CPU 关联性可让绑定和取消绑定进程和线程到指定的 CPU 或 CPU 范围。这可减少缓存问题。

流程

1. 通过查看 /proc/interrupts 文件，检查每个设备正在使用的 IRQ。

```
~]# cat /proc/interrupts
```

每行显示 IRQ 号、每个 CPU 中出现的中断数，后面是 IRQ 类型和描述。

```
           CPU0       CPU1
  0:   26575949         11         IO-APIC-edge  timer
  1:         14          7         IO-APIC-edge  i8042
```

2. 将 CPU 掩码写入特定 IRQ 的 smp_affinity 条目。CPU 掩码必须表示为十六进制数字。

例如，以下命令指示 IRQ 编号 142 仅在 CPU 0 上运行。

```
~]# echo 1 > /proc/irq/142/smp_affinity
```

只有在发生中断时，更改才会生效。

验证步骤

1. 执行将触发指定中断的活动。

2. 检查 /proc/interrupts 是否有变化。

配置 IRQ 的指定 CPU 上的中断数量增加，指定关联性外配置的 IRQ 中断数不会增加。

12.5. 使用 TASKSET 工具将进程绑定到 CPU

taskset 实用程序使用任务的进程 ID(PID)来查看或设置其 CPU 关联性。您可以使用 实用程序启动具有所选 CPU 关联性的命令。

要设置关联性，您需要获取 CPU 掩码为十进制或十六进制数字。mask 参数是一个位掩码，用于指定哪些 CPU 内核是被修改的命令或 PID 的法律。

重要

taskset 实用程序可用于 NUMA(Non-Uniform Memory Access)系统，但它不允许用户将线程绑定到 CPU 和最接近的 NUMA 内存节点。在这样的系统中，Tasket 不是首选工具，而 numactl 实用程序则应该用于其高级功能。

如需更多信息，请参阅 numactl(8) man page。

流程

- 使用必要的选项和参数运行 taskset。
  - 您可以使用 -c 参数而不是 CPU 掩码指定 CPU 列表。在本例中，my_embedded_process 会被指示在 CPU 0,4,7-11 上运行。

```
~]# taskset -c 0,4,7-11 /usr/local/bin/my_embedded_process
```

    在大多数情况下，这个调用更为方便。
  - 要设置当前运行的进程的关联性，请使用 taskset 并指定 CPU 掩码和进程。

    在本例中，指示 my_embedded_process 只使用 CPU 3（使用 CPU 掩码的十进制版本）。

```
~]# taskset 8 /usr/local/bin/my_embedded_process
```

  - 您可以在 bitmask 中指定多个 CPU。在本例中，my_embeed_process 会被指示在处理器 4、5、6 和 7 上执行（使用 CPU 掩码的十六进制版本）。

```
~]# taskset 0xF0 /usr/local/bin/my_embedded_process
```

- 您可以为已经运行的进程设置 CPU 关联性，以将 -p (--pid)选项用于 CPU 掩码，以及您要更改的进程的 PID。在本例中，指示 PID 为 7013 的进程仅在 CPU 0 上运行。

```
~]# taskset -p 1 7013
```

注意

您可以组合列出的选项。

其他资源

- taskset(1) man page
- numactl(8) man page

第 13 章 管理内存不足状态

内存(OOM)是指已分配所有可用内存的计算状态。通常，这会导致系统崩溃并正常停止工作。

以下提供了避免系统中 OOM 状态的说明。

13.1. 先决条件

- 系统上的 root 权限。

13.2. 更改内存不足值

/proc/sys/vm/panic_on_oom 文件包含一个控制内存不足(OOM)行为的切换的值。当文件包含 1 时，OOM 上的内核 panics 会正常停止工作。

默认值为 0， 它指示内核在系统处于 OOM 状态时调用 oom_killer 功能。通常，oom_killer 会终止不必要的进程，允许系统保留。

您可以更改 /proc/sys/vm/panic_on_oom 的值。

流程

1. 显示 /proc/sys/vm/panic_on_oom 的当前值。

```
# cat /proc/sys/vm/panic_on_oom
0
```

要更改 /proc/sys/vm/panic_on_oom 中的值：

2. 将新值回显到 /proc/sys/vm/panic_on_oom。

```
# echo 1 > /proc/sys/vm/panic_on_oom
```

注意

建议您在 OOM 上发出 Real-Time 内核 panic(1)。否则，当系统遇到 OOM 状态时，它不再是确定的。

验证步骤

1. 显示 /proc/sys/vm/panic_on_oom 的值。

```
# cat /proc/sys/vm/panic_on_oom
1
```

2. 验证显示的值是否与指定的值匹配。

13.3. 以内存不足状态时终止进程以终止

您可以对通过 oom_killer 功能终止的进程排列。这样可保证高优先级进程在 OOM 状态保持运行。每个进程都有一个目录 /proc/*PID*。每个目录都包括以下文件：

- oom_adj - oom_adj 有效的分数位于 -16 到 +15 范围。这个值用于计算进程的性能占用空间，使用一种算法来考虑进程运行的时间（其他因素除外）。
- oom_score - 包含使用 oom_adj 中的值计算的算法结果。

在内存不足状态中，oom_killer 功能以最高的 oom_score 终止进程。

您可以通过编辑进程的 oom_adj 文件来排列要终止的进程。

先决条件

- 知道您要优先排序的进程的进程 ID(PID)。

流程

1. 显示一个进程的当前 oom_score。

```
# cat /proc/12465/oom_score
79872
```

2. 显示进程的 oom_adj 的内容。

```
# cat /proc/12465/oom_adj
13
```

3. 编辑 oom_adj 的值。

```
# echo -5 > /proc/12465/oom_adj
```

验证步骤

1. 显示进程的当前 oom_score。

```
# cat /proc/12465/oom_score
78
```

2. 验证显示的值是否小于前面的值。

13.4. 为进程禁用内存不足终止程序

您可以通过将 oom_adj 设置为 -17 保留的值，为进程禁用 oom_killer 功能。这将保持进程处于活动状态，即使处于 OOM 状态。

流程

- 将 oom_adj 的值设置为 -17。

```
# echo -17 > /proc/12465/oom_adj
```

验证步骤

1. 显示进程的当前 oom_score。

```
# cat /proc/12465/oom_score
0
```

2. 验证显示的值为 **0**。

第 14 章 通过禁用 PC 卡守护进程降低 CPU 使用率

pcscd 守护进程管理与并行通信（PC 或 PCMCIA）和智能卡(SC)读取的连接。虽然 pcscd 通常是一个低优先级任务，但它通常可以使用比任何其他守护进程更多的 CPU。这种额外的背景措施可带来更高的抢占成本，导致实时任务和其他对确定性影响的其他不可影响。

先决条件

- 系统上的 root 权限。

流程

1. 检查 pcscd 守护进程的状态。

```
# systemctl status pcscd
● pcscd.service - PC/SC Smart Card Daemon
     Loaded: loaded (/usr/lib/systemd/system/pcscd.service; indirect; vendor preset:
disabled)
     Active: active (running) since Mon 2021-03-01 17:15:06 IST; 4s ago
TriggeredBy: ● pcscd.socket
       Docs: man:pcscd(8)
   Main PID: 2504609 (pcscd)
      Tasks: 3 (limit: 18732)
     Memory: 1.1M
        CPU: 24ms
     CGroup: /system.slice/pcscd.service
             └─2504609 /usr/sbin/pcscd --foreground --auto-exit
```

Active 参数显示 pcsd 守护进程的状态。

2. 如果 pcsd 守护进程正在运行，请将其停止。

```
# systemctl stop pcscd
Warning: Stopping pcscd.service, but it can still be activated by:
  pcscd.socket
```

3. 将系统配置为确保 pcsd 守护进程在系统引导时不会重启。

```
# systemctl disable pcscd
Removed /etc/systemd/system/sockets.target.wants/pcscd.socket.
```

验证步骤

1. 检查 pcscd 守护进程的状态。

```
# systemctl status pcscd
● pcscd.service - PC/SC Smart Card Daemon
     Loaded: loaded (/usr/lib/systemd/system/pcscd.service; indirect; vendor preset:
disabled)
     Active: inactive (dead) since Mon 2021-03-01 17:10:56 IST; 1min 22s ago
TriggeredBy: ● pcscd.socket
       Docs: man:pcscd(8)
   Main PID: 4494 (code=exited, status=0/SUCCESS)
        CPU: 37ms
```

2. 确保 Active 参数的值是 inactive(dead) 的。

第 15 章 平衡日志记录参数

syslog 服务器通过网络从程序转发日志消息。这种情况的发生频率越小，待处理事务越大可能是。如果事务非常大，则可能导致 I/O 激增。要防止这种情况，请让间隔不可小。

系统日志记录守护进程 syslogd 用于从不同程序收集信息。它还从内核日志记录守护进程 klogd 收集内核报告的信息。通常，syslogd 记录到本地文件，但也可以将其配置为通过网络记录到远程记录服务器。

流程

启用远程日志：

1. 配置要将日志发送到的计算机。如需更多信息，请参阅 Red Hat Enterprise Linux 上的 Remote Syslogging 使用 rsyslog。

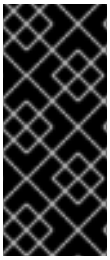
2. 配置将将日志发送到远程日志服务器的每个系统，使其 syslog 输出写入到服务器，而不是发送到本地文件系统。要做到这一点，编辑每个客户端系统中的 /etc/rsyslog.conf 文件。对于该文件中定义的每个日志记录规则，请将本地日志文件替换为远程日志记录服务器的地址。

```
# Log all kernel messages to remote logging host.
kern.*     @my.remote.logging.server
```

上面的示例将客户端系统配置为将所有内核信息记录到 @my.remote.logging.server 上的远程机器。

另外，您还可以将 syslogd 配置为记录本地生成的系统信息，方法是在 /etc/rsyslog.conf 文件中添加以下行：

```
# Log all messages to a remote logging server:
.    @my.remote.logging.server
```



重要

syslogd 守护进程不会包括其生成的网络流量的内置速率限制。因此，红帽建议在将 RHEL 用于 Real Time 系统时，只建议您的机构需要远程记录日志信息。例如：内核警告、身份验证请求以及如下内容：其他消息应在本地记录。

其他资源

- **syslog(3) man page**
- **rsyslog.conf(5) man page**
- **rsyslogd(8) man page**

第 16 章 使用 TUNA CLI 提高延迟

您可以使用 tuna CLI 改进系统中的延迟。与 tuna 命令一起使用的选项决定了调用的方法来提高延迟。

16.1. 先决条件

- 已安装 RHEL for Real Time 软件包组，以及 tuna 软件包。
- 系统的 root 权限。

16.2. TUNA CLI

tuna 命令行界面(CLI)是一个工具，可帮助您对系统进行调优。

注意

为 tuna 创建了一个新的图形界面，但还没有发布。

tuna CLI 可以用来调整调度程序可调项，调优线程优先级、IRQ 处理程序和隔离 CPU 内核和套接字。tuna 旨在降低执行调优任务的复杂性。该工具设计为在运行中的系统上使用，并且更改会立即发生。这允许任何特定于应用程序的测量工具在更改后马上查看和分析系统性能。

tuna CLI 具有操作选项和修饰符选项。在计划修改的操作之前，必须在命令行中指定修饰符选项。所有修饰符选项都适用于遵循的步骤，直到修饰符选项被覆盖。

16.3. 使用 TUNA CLI 隔离 CPU

您可以使用 tuna CLI 将中断(IRQ)与不同专用 CPU 上的用户进程隔离中断(IRQ)来最小化实时环境中的延迟。有关隔离 CPU 的更多信息，请参阅 中断和进程绑定。

先决条件

- 已安装 RHEL for Real Time 软件包组，以及 tuna 软件包。

- 系统的 root 权限。

流程

- 隔离一个或多个 CPU。

```
# tuna --cpus=cpu_list --isolate
```

其中 *cpu_list* 是要隔离的 CPU 的逗号分隔列表。

例如：

```
# tuna --cpus=0,1 --isolate
```

16.4. 使用 TUNA CLI 将中断移到指定的 CPU

您可以使用 tuna CLI 将中断(IRQ)移到专用 CPU，以最小化或消除实时环境中的延迟。有关移动 IRQ 的更多信息，请参阅 中断和进程绑定。

先决条件

- 已安装 RHEL for Real Time 软件包组，以及 tuna 软件包。
- 系统的 root 权限。

流程

1. 列出将 IRQ 列表附加到的 CPU。

```
# tuna --irqs=irq_list --show_irqs
```

其中 *irq_list* 是您要列出附加 CPU 的 IRQs 的逗号分隔列表。

例如：

```
# tuna --irqs=128 --show_irqs
  # users         affinity
 128 iwlwifi          0,1,2,3
```

2. 将 IRQ 列表附加到 CPU 列表。

```
# tuna --irqs=irq_list --cpus=cpu_list --move
```

其中 *irq_list* 是您要附加的 IRQs 的逗号分隔列表，*cpu_list* 是将附加到的 CPU 的逗号分隔列表。

例如：

```
# tuna --irqs=128 --cpus=3 --move
```

验证步骤

- 比较之前和将任何 IRQ 移动到指定的 CPU 后所选 IRQ 的状态。

```
# tuna --irqs=128 --show_irqs
  # users         affinity
 128 iwlwifi          3
```

16.5. 使用 TUNA CLI 更改进程调度策略和优先级

您可以使用 tuna CLI 更改进程调度策略和优先级。

先决条件

- 已安装 RHEL for Real Time 软件包组，以及 tuna 软件包。
- 系统的 root 权限。

注意

分配 OTHER 和 NATCH 调度策略不需要 root 权限。

流程

1. 查看线程的信息。

```
# tuna --threads=thread_list --show_threads
```

其中 *thread_list* 是您要显示的进程的逗号分隔列表。

例如：

```
# tuna --threads=rngd --show_threads
                  thread       ctxt_switches
  pid SCHED_ rtpri affinity voluntary nonvoluntary            cmd
 3571   OTHER     0  0,1,2,3    167697          134           rngd
```

2. 修改进程调度策略和线程的优先级。

```
# tuna --threads=thread_list --priority scheduling_policy:priority_number
```

其中：

- *thread_list* 是您要显示的调度策略和优先级的进程的逗号分隔列表。
- *scheduling_policy* 是以下之一：
  - 其他
  - BATCH
  - FIFO - First In First Out
  - RR - Round Robin
-

priority_number 是一个优先级号，从 0 到 99，其中 0 不是优先级，99 是最高优先级。

注意

OTHER 和 BATCH 调度策略不需要指定优先级。另外，唯一有效的优先级（如果指定）是 0。FIFO 和 RR 调度策略的优先级需要 1 或更高优先级。

例如：

```
# tuna --threads=rngd --priority FIFO:1
```

验证步骤

- 查看线程的信息，以确保信息更改。

```
# tuna --threads=rngd --show_threads
                  thread       ctxt_switches
  pid SCHED_ rtpri affinity voluntary nonvoluntary            cmd
 3571   FIFO     1  0,1,2,3    167697         134            rngd
```

第 17 章 安装 KDUMP

在新的 Red Hat Enterprise Linux 安装中默认安装并激活 kdump 服务。以下小节解释了 kdump 是什么以及如何在默认启用时安装 kdump。

17.1. KDUMP

kdump 是提供崩溃转储机制的服务。该服务可让您保存系统内存内容进行分析。kdump 使用 kexec 系统调用在没有重启的情况下引导到第二个内核（一个 *捕获内核*），然后捕获崩溃内核的内存（ *崩溃转储* 或 *vmcore*）并将其保存到文件中。这个第二个内核位于系统内存保留的一部分。

重要

内核崩溃转储可能会是系统失败时唯一可用的信息（关键错误）。因此，在关键任务环境中运行 kdump 非常重要。红帽建议系统管理员在正常内核更新周期内定期更新和测试 kexec-tools。这在部署了新内核功能时尤为重要。

您可以为机器上的所有安装内核启用 kdump，或只为指定内核启用 kdump。当计算机上有多个内核使用时，这非常有用，其中一些内核足够稳定，没有关注它们可以崩溃。

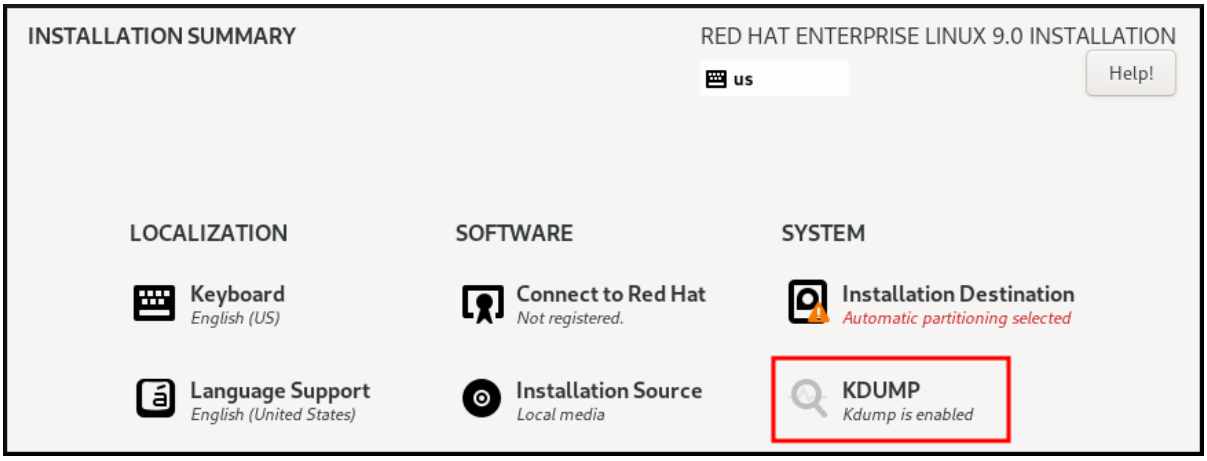
安装 kdump 时，会创建一个默认的 /etc/kdump.conf 文件。该文件包含默认最小 kdump 配置。您可以编辑此文件来自定义 kdump 配置，但这不是必需的。

17.2. 使用 ANACONDA 安装 KDUMP

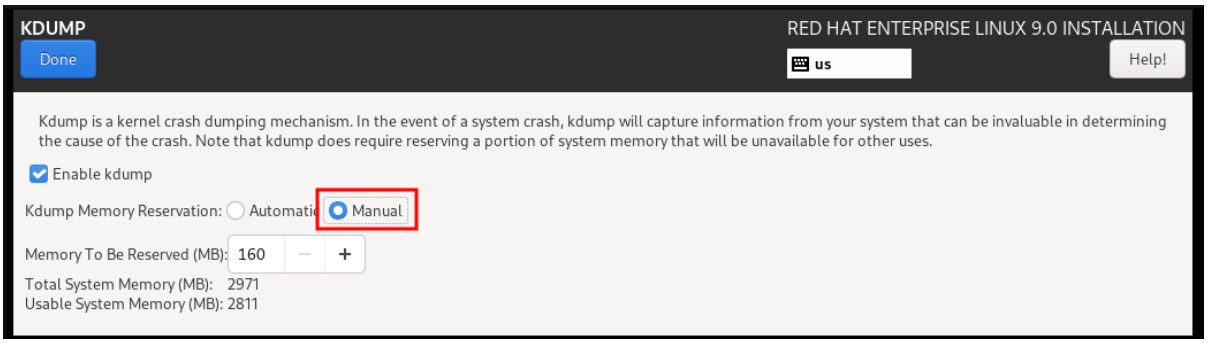
Anaconda 安装程序在互动安装过程中为 kdump 配置提供了一个图形界面页面。安装程序屏幕标题为 KDUMP，可在主 安装摘要屏幕中使用。您可以启用 kdump 并保留所需的内存量。

流程

1. 进入 Kdump 字段。
2. 如果尚未启用，请启用 kdump。



3. 为 kdump 定义应保留多少内存。



17.3. 在命令行中安装 KDUMP

在某些情况下，有些安装选项（如自定义 Kickstart 安装） 默认情况下不安装 或启用 kdump。如果是您的问题单，请按照以下步骤执行。

先决条件

- 活跃的 RHEL 订阅
- kexec-tools 软件包
- 满足 kdump 配置和目标的要求。详情请查看支持的 kdump 配置和目标。

流程

1. 检查您的系统中是否安装了 kdump ：

```
# rpm -q kexec-tools
```

- 如果安装了该软件包，输出：

```
# kexec-tools-2.0.22-13.el9.x86_64
```

如果没有安装该软件包，输出：

```
package kexec-tools is not installed
```

2. 通过以下方法安装 kdump 和其他必要的软件包：

```
# dnf install kexec-tools
```

第 18 章 在命令行中配置 KDUMP

以下小节解释了如何计划和构建 kdump 环境。

18.1. 估算 KDUMP 大小

在规划和构建 kdump 环境时，务必要了解崩溃转储文件所需的空间量。

makedumpfile --mem-usage 命令估计崩溃转储文件所需的空间量。它生成内存用量报告。这个报告可帮助您确定转储级别，哪些页面可以安全地排除。

流程

- 执行以下命令生成内存用量报告：

```
# makedumpfile --mem-usage /proc/kcore


TYPE       PAGES    EXCLUDABLE    DESCRIPTION
----------------------------------------------------------
ZERO        501635      yes        Pages filled with zero
CACHE        51657      yes        Cache pages
CACHE_PRIVATE 5442        yes        Cache pages + private
USER        16301       yes        User process pages
FREE        77738211    yes        Free pages
KERN_DATA     1333192     no         Dumpable kernel data
```

重要

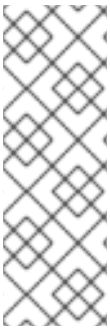
makedumpfile --mem-usage 命令在页面中报告所需的内存。这意味着您必须计算用于内核页面大小的内存大小。

18.2. 配置 KDUMP 内存用量

在系统引导过程中为 kdump 保留内存。内存大小是在系统 Grand Unified Bootloader(GRUB)2 配置文件中配置的。内存大小取决于配置文件中指定的 crashkernel= 选项的值以及系统物理内存的大小。

crashkernel= 选项可以通过多种方式定义。您可以指定 crashkernel= 值，或者配置 auto 选项。crashkernel=auto 参数根据系统中物理内存总量自动保留内存。配置后，内核将自动为捕获内核保

留一个适当数量所需的内存量。这有助于防止内存不足(OOM)错误。



注意

kdump 的自动内存分配因系统硬件架构和可用内存大小而异。

如果系统自动分配低于最小内存阈值，您可以手动配置保留内存量。

先决条件

- **root 权限。**
- **满足 kdump 配置和目标的要求。详情请查看支持的 kdump 配置和目标。**

流程

1. **编辑 /etc/default/grub 文件。**
2. **设置 crashkernel= 选项。**

   例如：要保留 128 MB 内存，请使用：

   ```
   crashkernel=128M
   ```

   或者，您可以根据安装的内存总量将保留内存量设置为变量。变量中的内存保留语法为 crashkernel=*<range1>*:*<size1>*,*<range2>*:*<size2>*。例如：

   ```
   crashkernel=512M-2G:64M,2G-:128M
   ```

   如果系统内存总量介于 512 MB 和 2 GB 之间，则上述示例保留 64 MB 的内存。如果内存量大于 2 GB，则会保留 128 MB。

- **保留内存的偏移。**

  有些系统需要使用特定的固定偏移保留内存，因为 crashkernel 保留非常早，并希望保

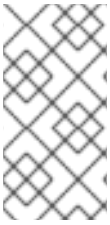
留一些区域以供特殊使用。如果设置了偏移，则保留内存从此偏移开始。要偏移保留的内存，请使用以下语法：

```
crashkernel=128M@16M
```

在上例中，kdump 从 16MB 开始保留 128 MB 内存（物理地址 0x01000000）。如果偏移参数设为 0 或完全省略，kdump 会自动偏移保留内存。在设置变量内存保留时还可使用此语法。在这种情况下，偏移总是最后指定（例如 crashkernel=512M-2G:64M,2G-:128M@16M）。

3. 使用以下命令更新 GRUB2 配置文件：

```
# grub2-mkconfig -o /boot/grub2/grub.cfg
```



注意

为 kdump 配置内存的替代方法是使用 grubby 实用程序更新一个引导条目、多个引导条目或所有引导条目。

其他资源

- kdump 的内存要求
- 配置内核命令行参数
- grub2-mkconfig 脚本静默忽略 GRUB_CMDLINE_LINUX 中的选项
- 如何在系统引导前手动修改 GRUB 中的引导参数
- 如何在 Red Hat Enterprise Linux 8 中安装并引导自定义内核
- grubby(8) 手册页

18.3. 配置 KDUMP 目标

崩溃转储通常作为本地文件系统中的文件存储，直接写入设备。另外，您还可以为崩溃转储设置，以便使用 NFS 或 SSH 协议通过网络发送。一次只能设置这些选项之一来保留崩溃转储文件。默认行为是将其保存在本地文件系统的 /var/crash/ 目录中。

先决条件

- 根 权限。
- 满足 kdump 配置和目标的要求。详情请查看支持的 kdump 配置和目标。

流程

- 要在本地文件系统的 /var/crash/ 目录中存储崩溃转储文件，请编辑 /etc/kdump.conf 文件并指定路径：

```
path /var/crash
```

选项 路径 /var/crash 代表到 kdump 保存崩溃转储文件的文件系统的路径。当您在 /etc/kdump.conf 文件中指定转储目标时，路径相对于指定的转储目标。

如果您没有在 /etc/kdump.conf 文件中指定转储目标，则该路径代表根目录的绝对路径。根据当前系统中挂载的内容，会自动执行转储目标和调整的转储路径。

kdump 将崩溃转储文件保存在 /var/crash/var/crash 目录中，当转储目标挂载到 /var/crash 时，选项路径 也设置为 /etc/kdump.conf 文件中的 /var/crash。例如，在以下实例中，ext4 文件系统已挂载到 /var/crash，其 path 被设置为 /var/crash:

```
grep -v ^# etc/kdump.conf | grep -v ^$
ext4 /dev/mapper/vg00-varcrashvol
path /var/crash
core_collector makedumpfile -c --message-level 1 -d 31
```

这会导致 /var/crash/var/crash 路径。要解决这个问题，请使用选项 path / 而不是 path /var/crash

- 要改变保存崩溃转储的本地目录，以 root 用户身份编辑 /etc/kdump.conf 配置文件，如下所述。

1. 从 #path /var/crash 行的开头删除 hash 符号("#")。

2. 使用预期的目录路径替换该值。例如：

```
path /usr/local/cores
```

重要

在 RHEL 8 中，当 kdump systemd 服务启动 - 否则服务失败，使用 path 指令定义的目录必须存在。这个行为与早期版本的 RHEL 不同，如果启动该服务时不存在该目录，则会自动创建该目录。

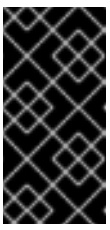
- 要将文件写入不同的分区，以 root 用户身份编辑 /etc/kdump.conf 配置文件，如下所述。

1. 根据您的选择，从 #ext4 行的开头删除 hash 符号("#")。

   - 设备名称（ #ext4 /dev/vg/lv_kdump 行）
   - 文件系统标签（ #ext4 LABEL=/boot 行）
   - UUID（ #ext4 UUID=03138356-5e61-4ab3-b58e-27507ac41937 行）

2. 将文件系统类型以及设备名称、标签或者 UUID 更改为所需值。例如：

```
ext4 UUID=03138356-5e61-4ab3-b58e-27507ac41937
```



重要

建议您使用 LABEL= 或 UUID= 指定存储设备。无法保证 /dev/sda3 等磁盘设备名称在重启后保持一致。

- 要将崩溃转储直接写入设备，请编辑 /etc/kdump.conf 配置文件：

1. 删除 #raw /dev/vg/lv_kdump 行开头的哈希符号 ("#")。

2. 使用预期的设备名称替换该值。例如：

```
raw /dev/sdb1
```

- 要使用 NFS 协议将崩溃转储保存到远程机器中，请编辑 /etc/kdump.conf 配置文件：

1. 删除 #nfs my.server.com:/export/tmp 行开头的哈希符号 ("#")。

2. 使用有效的主机名和目录路径替换该值。例如：

```
nfs penguin.example.com:/export/cores
```

- 要使用 SSH 协议将崩溃转储保存到远程机器中，请编辑 /etc/kdump.conf 配置文件：

1. 从 #ssh user@my.server.com 行的开头删除 hash 符号("#")。

2. 使用有效的用户名和密码替换该值。

3. 在配置中包含 SSH 密钥。

   - 从 #sshkey /root/.ssh/kdump_id_rsa 行的开头删除哈希符号。

   - 将该值改为您要转储的服务器中有效密钥的位置。例如：

```
ssh john@penguin.example.com
sshkey /root/.ssh/mykey
```

18.4. 配置 KDUMP 核心收集器

kdump 服务使用 core_collector 程序捕获崩溃转储镜像。在 RHEL 中，makedumpfile 实用程序是默

认的核心收集器。它通过以下方式帮助缩小转储文件：

- 压缩崩溃转储文件的大小，并只复制使用不同的转储级别所需的页面
- 排除不必要的崩溃转储页面
- 过滤崩溃转储中包含的页面类型。

语法

```
core_collector makedumpfile -l --message-level 1 -d 31
```

选项

- -c、-l 或 -p ：指定每个页的压缩 dump 文件的格式，使用 zlib 用于 -c 选项、使用 lzo 用于 -l 新选项，或 snappy 用于 -p 选项。
- -d (dump_level) ：排除页面，它们不会复制到转储文件中。
- --message-level ：指定消息类型。您可以通过使用这个选项指定 message_level 来限制打印的输出。例如，把 message_level 设置为 7 可打印常见消息和错误消息。message_level 的最大值为 31

先决条件

- Root 权限
- 满足 kdump 配置和目标的要求。详情请查看支持的 kdump 配置和目标。

步骤

1.

以 root 用户身份，编辑 /etc/kdump.conf 配置文件并从 #core_collector makedumpfile -l --message-level 1 -d 31 的开头删除 hash 符号("#")。

2. 要启用崩溃转储文件压缩，请执行：

```
core_collector makedumpfile -l --message-level 1 -d 31
```

l 选项指定 dump 压缩的文件格式。d 选项将转储级别指定为 31。--message-level 选项指定消息级别为 1。

另外，请考虑以下带有 -c 和 -p 选项的示例：

- 使用 -c 压缩崩溃转储文件：

```
core_collector makedumpfile -c -d 31 --message-level 1
```

- 使用 -p 压缩崩溃转储文件：

```
core_collector makedumpfile -p -d 31 --message-level 1
```

其他资源

- makedumpfile(8) 手册页
- kdump 配置文件

18.5. 配置 KDUMP 默认失败响应

默认情况下，当 kdump 在配置的目标位置创建崩溃转储文件时，系统会重启，并会在此过程中丢失转储。要更改此行为，请遵循以下步骤。

先决条件

- root 权限。

- 满足 kdump 配置和目标的要求。详情请查看支持的 kdump 配置和目标。

步骤

1. 以 root 用户身份，从 /etc/kdump.conf 配置文件的 #failure_action 行的开头删除 hash 符号("#")。

2. 将值替换为所需操作。

```
failure_action poweroff
```

其他资源

- 配置 kdump 目标

18.6. 测试 KDUMP 配置

您可以在机器进入生产环境前测试崩溃转储过程是否正常工作。

警告

以下命令会导致内核崩溃。在按照这些步骤操作时要小心，且不要在活跃的生产环境中使用它们。

步骤

1. 在启用了 kdump 的情况下重启系统。

2. 确保 kdump 正在运行：

```
# systemctl is-active kdump
active
```

3. 强制 Linux 内核崩溃：

```
echo 1 &gt; /proc/sys/kernel/sysrq`
echo c &gt; /proc/sysrq-trigger`
```

警告

上面的命令可崩溃内核，而且需要重新引导。

再次引导后，*address-YYYY-MM-DD-HH:MM:SS*/vmcore 文件会被创建，位于您在 /etc/kdump.conf 文件中指定的位置（默认为 /var/crash/）。

注意

此操作确认配置的有效性。另外，也可以使用此操作记录崩溃转储使用代表工作负载完成所需的时间。

其他资源

- 配置 kdump 目标

第 19 章 启用 KDUMP

这部分提供了为所有安装的内核或特定内核启用和启动 kdump 服务所需的信息和步骤。

19.1. 为所有安装的内核启用 KDUMP

您可以为机器上安装的所有内核启用并启动 kdump 服务。

先决条件

- 管理员特权

流程

1. 将 crashkernel=auto 命令行参数添加到所有安装的内核中：

```
# /grubby --update-kernel=ALL --args="crashkernel=auto"
```

2. 启用 kdump 服务。

```
# systemctl enable --now kdump.service
```

验证

- 检查 kdump 服务是否正在运行：

```
# systemctl status kdump.service

○ kdump.service - Crash recovery kernel arming
     Loaded: loaded (/usr/lib/systemd/system/kdump.service; enabled; vendor preset:
disabled)
     Active: active (live)
```

19.2. 为特定安装内核启用 KDUMP

您可以在机器上为特定内核启用 kdump 服务。

先决条件

- 管理员特权

流程

1. 列出在机器上安装的内核。

```
# ls -a /boot/vmlinuz-*
/boot/vmlinuz-0-rescue-2930657cd0dc43c2b75db480e5e5b4a9 /boot/vmlinuz-4.18.0-330.el8.x86_64 /boot/vmlinuz-4.18.0-330.rt7.111.el8.x86_64
```

2. 在系统的 Grand Unified Bootloader(GRUB)2 配置文件中添加特定的 kdump 内核。

   例如：

```
# grubby --update-kernel=vmlinuz-4.18.0-330.el8.x86_64 --args="crashkernel=auto"
```

3. 启用 kdump 服务。

```
# systemctl enable --now kdump.service
```

验证

- 检查 kdump 服务是否正在运行：

```
# systemctl status kdump.service

○ kdump.service - Crash recovery kernel arming
     Loaded: loaded (/usr/lib/systemd/system/kdump.service; enabled; vendor preset: disabled)
     Active: active (live)
```

19.3. 禁用 KDUMP 服务

要在引导时禁用 kdump 服务，请按照以下步骤操作。

先决条件

- 满足 kdump 配置和目标的要求。详情请查看支持的 kdump 配置和目标。

- 安装 kdump 的所有配置都是根据您的需要设置的。详情请参阅 安装 kdump。

流程

1. 在当前会话中停止 kdump 服务：

```
# systemctl stop kdump.service
```

2. 禁用 kdump 服务：

```
# systemctl disable kdump.service
```

警告

建议您设置 kptr_restrict=1。在这种情况下，kdumpctl 服务会加载崩溃内核，无论正在启用或不启用内核地址空间(KASLR)。

故障排除步骤

当 kptr_restrict 没有设置为(1)时，如果启用了 KASLR，则 /proc/kcore 文件的内容生成为所有零。因此，kdumpctl 服务无法访问 /proc/kcore 并载入崩溃内核。

要临时解决这个问题，/usr/share/doc/kexec-tools/kexec-kdump-howto.txt 文件会显示警告信息，建议 kptr_restrict=1 设置。

要确保 kdumpctl 服务加载崩溃内核，请验证 sysctl.conf 文件中列出了 kernel.kptr_restrict = 1。

其他资源

- 在 RHEL 中配置基本系统设置

第 20 章 确保已挂载 DEBUGFS

debugfs 文件系统专门设计用于调试并将信息提供给用户。它会在 RHEL 8 中自动挂载到 /sys/kernel/debug/ 目录中。

注意

debugfs 文件系统使用 ftrace 和 trace-cmd 命令挂载。

流程

验证 **debugfs 是否已挂载：**

- **运行以下命令：**

```
# mount | grep ^debugfs
debugfs on /sys/kernel/debug type debugfs (rw,nosuid,nodev,noexec,relatime,seclabel)
```

如果挂载 debugfs，命令会显示 debugfs 的挂载点和属性。

如果没有挂载 debugfs，命令会返回任何内容。

第 21 章 设置调度程序优先级

Red Hat Enterprise Linux for Real Time 内核允许对调度程序优先级进行精细的控制。它还允许以高于内核线程的优先级调度应用程序级别的程序。

警告

设置调度程序优先级可能会带来后果，并可能导致系统变得无响应，或者如果关键内核进程无法根据需要运行，则可能导致系统变得无响应。最终，正确的设置取决于工作负载。

21.1. 查看线程调度优先级

使用一系列级别设置线程优先级，范围从 0（最低优先级）到 99（最高优先级）。systemd 服务管理器可用于在内核启动后更改默认线程优先级。

流程

- 要查看正在运行的线程的调度优先级，请使用 tuna 程序：

```
# tuna --show_threads
                    thread       ctxt_switches
  pid SCHED_ rtpri affinity voluntary nonvoluntary            cmd
 2      OTHER     0    0xfff       451            3        kthreadd
 3       FIFO     1        0     46395            2     ksoftirqd/0
 5      OTHER     0        0        11            1    kworker/0:0H
 7       FIFO    99        0         9            1   posixcputmr/0
 ...[output truncated]...
```

21.2. 在引导过程中更改服务优先级

使用 systemd，您可以为在引导过程中启动的服务设置实时优先级。

单元配置指令用于在引导过程中更改服务优先级。引导过程优先级更改通过使用 /etc/systemd/system/ 服务 .system.d/priority.conf 的*service* 部分中的以下指令来实现：

CPUSchedulingPolicy=

设置已执行进程的 CPU 调度策略。获取 Linux 上可用的调度类之一：

- 其他
- batch
- idle
- fifo
- rr

CPUSchedulingPriority=

为已执行的进程设置 CPU 调度优先级。可用的优先级范围取决于所选的 CPU 调度策略。对于实时调度策略，可以使用 1（最低优先级）和 99 （最高优先级）之间的整数。

先决条件

- 管理员特权。
- 引导时运行的服务。

流程

对于现有服务：

1. 为该服务创建一个补充服务配置目录文件。

```
# cat <<-EOF > /etc/systemd/system/mcelog.system.d/priority.conf
```

2. 将调度策略和优先级添加到 *[SERVICE]* 部分中的 文件。

例如：

```
[SERVICE]
CPUSchedulingPolicy=fifo
CPUSchedulingPriority=20
EOF
```

3. 重新加载 systemd 脚本配置。

```
# systemctl daemon-reload
```

4. 重启该服务。

```
# systemctl restart mcelog
```

验证

- 显示服务优先级。

```
$ tuna -t mcelog -P
```

输出显示服务的已配置优先级。

例如：

```
                thread       ctxt_switches
  pid SCHED_ rtpri affinity voluntary nonvoluntary            cmd
826     FIFO    20  0,1,2,3        13            0          mcelog
```

其他资源

- 使用 systemd 单元文件。

21.3. 配置服务的 CPU 使用量

使用 systemd，您可以指定可在哪些服务上运行的 CPU。

先决条件

- **管理员特权。**

流程

1. **为该服务创建一个补充服务配置目录文件。**

```
# md sscd
```

2. **使用 *[SERVICE]* 部分中的 CPUAffinity 属性，将用于该服务的 CPUAffinity 属性添加到 文件中。**

例如：

```
[SERVICE]
CPUAffinity=0,1
EOF
```

3. **重新加载 systemd 脚本配置。**

```
systemctl daemon-reload
```

4. **重启该服务。**

```
systemctl restart _service_
```

验证

- **显示将指定服务限制为的 CPU。**

```
$ tuna -t mcelog -P
```

其中，*service* 是指定的服务。

以下输出显示 mcelog 服务限制为 CPU 0 和 1。

-

```
                  thread       ctxt_switches
  pid SCHED_ rtpri affinity voluntary nonvoluntary            cmd
12954   FIFO    20      0,1         2            1          mcelog
```

21.4. 优先级映射

调度程序优先级在组中定义，某些专用于特定内核功能的组。

表 21.1. 线程优先级表

优先级	线程	描述
1	低优先级内核线程	此优先级通常为需要超过 **SCHED_OTHER** 的任务保留。
2 - 49	可供使用	用于典型的应用程序优先级的范围。
50	默认 hard-IRQ 值	这个优先级是基于硬件的中断的默认值。
51 - 98	高优先级线程	对定期执行的线程使用此范围，且必须快速响应时间。**不要将** 这个范围用于 CPU 密集型线程，因为它将阻止响应较低级别的中断。
99	Watchdogs 和 migration	必须以最高优先级运行的系统线程。

21.5. 其他资源

- 使用 systemd 单元文件

第 22 章 非一致性内存访问

taskset 实用程序仅适用于 CPU 关联性，并且不知道其他 NUMA 资源，如内存节点。如果要与 NUMA 结合使用进程绑定，请使用 numactl 命令而不是 taskset。

有关 NUMA API 的更多信息，请参阅 Andi Kleen 的白皮书是 Linux 的 NUMA API。

其他资源

- Andi Kleen 的白皮书，Linux 的 NUMA API
- numactl(8) man page

第 23 章 RT 的 RHEL 中的 INFINIBAND

InfiniBand 是一个通信架构类型，通常用于提高带宽，提高服务质量(QOS)，并为故障转移提供。它还可用来使用 Remote Direct Memory Access(RDMA)机制来提高延迟。

在 RHEL for Real Time 下支持 Infiniband 与 RHEL 8 下提供的 Infiniband 相同。

其他资源

- Infiniband 入门

第 24 章 使用 ROCE 和高性能网络

RoCE （融合以太网）是一个协议，它通过以太网实现 Remote Direct Memory Access(RDMA)。它允许您在数据中心维护一致的高速环境，同时为关键事务提供确定的低延迟数据传输。

High Performance Networking (HPN)是一组共享库，向内核中提供 RoCE 接口。HPN 不使用标准以太网基础结构将数据直接放入远程系统内存，从而减少 CPU 开销并降低基础架构成本。

对 RHEL for Real Time 下的 RoCE 和 HPN 的支持与 RHEL 8 提供的支持不同。

注意

有关如何设置以太网网络的详情，请参考 配置 RoCE。

第 25 章 网络确定提示

TCP 可能会对延迟产生大量影响。TCP 增加了延迟以便获取效率、控制拥塞以及确保可靠交付。在调整时，请考虑以下点：

- 您需要订购的交付吗？
- 您是否需要保护数据包丢失？

  多次传输数据包可能会导致延迟。
- 您是否需要使用 TCP？

  考虑通过使用套接字上的 TCP_NODELAY 禁用 Nagle 缓冲算法。Nagle 算法收集一次性发送的小传出数据包，并对延迟产生不利影响。

有很多用于调优网络的工具。本节介绍一些更有用的工具。

25.1. 合并中断

在传输大量吞吐量的数据的系统中，使用默认值或增加 coalescence 可以增加吞吐量并降低出现 CPU 中断的数量。对于需要快速网络响应的系统，建议减少或禁用并发问题。

要减少中断数量，可以收集数据包，并为数据包集合生成的单一中断。

先决条件

- 管理员特权。

流程

- 要启用 coalescing 中断，请使用 --coalesce 选项运行 ethtool 命令。

```
# ethtool -C tun0
```

验证

验证是否启用了 coalescing 中断。

```
# ethtool -c tun0
Coalesce parameters for tun0:
Adaptive RX: n/a  TX: n/a
stats-block-usecs: n/a
sample-interval: n/a
pkt-rate-low: n/a
pkt-rate-high: n/a

rx-usecs: n/a
rx-frames: 0
rx-usecs-irq: n/a
rx-frames-irq: n/a

tx-usecs: n/a
tx-frames: n/a
tx-usecs-irq: n/a
tx-frames-irq: n/a

rx-usecs-low: n/a
rx-frame-low: n/a
tx-usecs-low: n/a
tx-frame-low: n/a

rx-usecs-high: n/a
rx-frame-high: n/a
tx-usecs-high: n/a
tx-frame-high: n/a

CQE mode RX: n/a  TX: n/a
```

25.2. 避免网络拥塞

I/O 交换机通常会受到 back-pressure，在出现完整的缓冲区后构建网络数据构建。您可以更改暂停参数并避免网络拥塞。

先决条件

- 管理员特权

流程

- 要更改暂停参数，请使用 -A 选项运行 ethtool 命令。

```
# ethtool -A enp0s31f6
```

验证

验证 pause 参数已更改。

```
# ethtool -a enp0s31f6
Pause parameters for enp0s31f6:
Autonegotiate: on
RX:  on
TX:  on
```

25.3. 监控网络协议统计

netstat 命令可用于监控网络流量。

流程

监控网络流量：

```
$ netstat -s
Ip:
    Forwarding: 1
    30817508 total packets received
    2927 forwarded
    0 incoming packets discarded
    30813320 incoming packets delivered
    19184491 requests sent out
    181 outgoing packets dropped
    2628 dropped because of missing route
Icmp
    29450 ICMP messages received
    213 input ICMP message failed
    ICMP input histogram:
        destination unreachable: 29431
        echo requests: 19
    10141 ICMP messages sent
    0 ICMP messages failed
    ICMP output histogram:
        destination unreachable: 10122
        echo replies: 19
IcmpMsg:
        InType3: 29431
        InType8: 19
        OutType0: 19
        OutType3: 10122
Tcp:
    162638 active connection openings
    89 passive connection openings
```

```
    38908 failed connection attempts
    17869 connection resets received
    48 connections established
    8456952 segments received
    9323882 segments sent out
    69885 segments retransmitted
    1143 bad segments received
    56209 resets sent
Udp:
    21929780 packets received
    1319 packets to unknown port received
    712919 packet receive errors
    10134989 packets sent
    712919 receive buffer errors
    180 send buffer errors
    IgnoredMulti: 39231
```

25.4. 其他资源

- ethtool(8)
- netstat(8)

第 26 章 使用 TRACE-CMD 追踪延迟

trace-cmd 程序是 ftrace 工具的一个前端。它可以启用 ftrace 操作，而无需写入 /sys/kernel/debug/tracing/ 目录。trace-cmd 在安装时不会添加任何开销。

先决条件

- 管理员特权。

26.1. 安装 TRACE-CMD

trace-cmd 工具为 ftrace 工具提供了一个前端。

先决条件

- 管理员特权

流程

- 安装 trace-cmd。

```
# dnf install trace-cmd
```

26.2. 运行 TRACE-CMD

您可以使用 trace-cmd 工具访问所有 ftrace 功能。

先决条件

- 管理员特权。

流程

- 输入 trace-cmd *命令*

其中 *command* 是一个 ftrace 选项。

注意

有关命令和选项的完整列表，请查看 trace-cmd(1) 手册页。大多数个别命令也具有自己的 man page，trace-cmd-*命令*。

26.3. TRACE-CMD 示例

这提供了很多 trace-cmd 示例。

例子

- 在 *myapp* 运行时，启用和开始在内核中执行的记录功能。

```
# trace-cmd record -p function myapp
```

此记录来自所有 CPU 和所有任务的功能，甚至与 *myapp* 无关。

- 显示结果。

```
# trace-cmd report
```

- 仅记录在 *myapp* 运行时 开始的功能。

```
# trace-cmd record -p function -l 'sched*' myapp
```

- 启用所有 IRQ 事件。

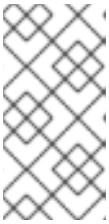
```
# trace-cmd start -e irq
```

- 启动 wakeup_rt tracer。

```
# trace-cmd start -p wakeup_rt
```

- 在禁用功能追踪时启动 preemptirqsoff tracer。

```
# trace-cmd start -p preemptirqsoff -d
```



注意

RHEL 8 中的 trace-cmd 版本关闭 ftrace_enabled 而不是使用 function-trace 选项。您可以使用 trace-cmd start -p 功能再次启用 ftrace。

- 在 trace-cmd 开始修改系统前，恢复系统所处的状态。

```
# trace-cmd start -p nop
```

如果您要在使用 trace-cmd 后使用 debugfs 文件系统，这很重要，无论是在平均时间中重启该系统。

- 跟踪单个 trace 点。

```
# trace-cmd record -e sched_wakeup ls /bin
```

- 停止追踪。

```
# trace-cmd record stop
```

26.4. 其他资源

- trace-cmd(1) man page

第 27 章 使用 TUNED-PROFILES-REALTIME 隔离 CPU

要为应用程序线程提供尽可能的执行时间，您可以隔离 CPU。因此，从 CPU 中删除尽可能多的额外任务。隔离 CPU 通常涉及：

- 删除所有用户空间线程。
- 删除任何未绑定内核线程（绑定内核线程与特定的 CPU 绑定，且无法移动）。
- 通过修改系统中每个 Interrupt Request(IRQ)的 /proc/irq/ N /smp_affinity 属性来移除中断。

本节演示了如何使用 tuned-profiles-realtime 软件包的 isolated_cores=cpulist 配置选项自动进行这些操作。

先决条件

- 管理员特权。

27.1. 选择要隔离的 CPU

选择 CPU 进行隔离需要仔细考虑系统的 CPU 拓扑。不同的用例可能需要不同的配置：

- 如果您的应用程序有一个多线程应用程序，其中线程需要通过共享缓存相互通信，则可能需要在同一 NUMA 节点或物理套接字中保存。
- 如果您运行多个不相关的实时应用程序，可将 CPU 除以 NUMA 节点或套接字进行隔离。

hwloc 软件包提供有助于获取 CPU 信息的实用程序，包括 lstopo-no-graphics 和 numactl。

先决条件

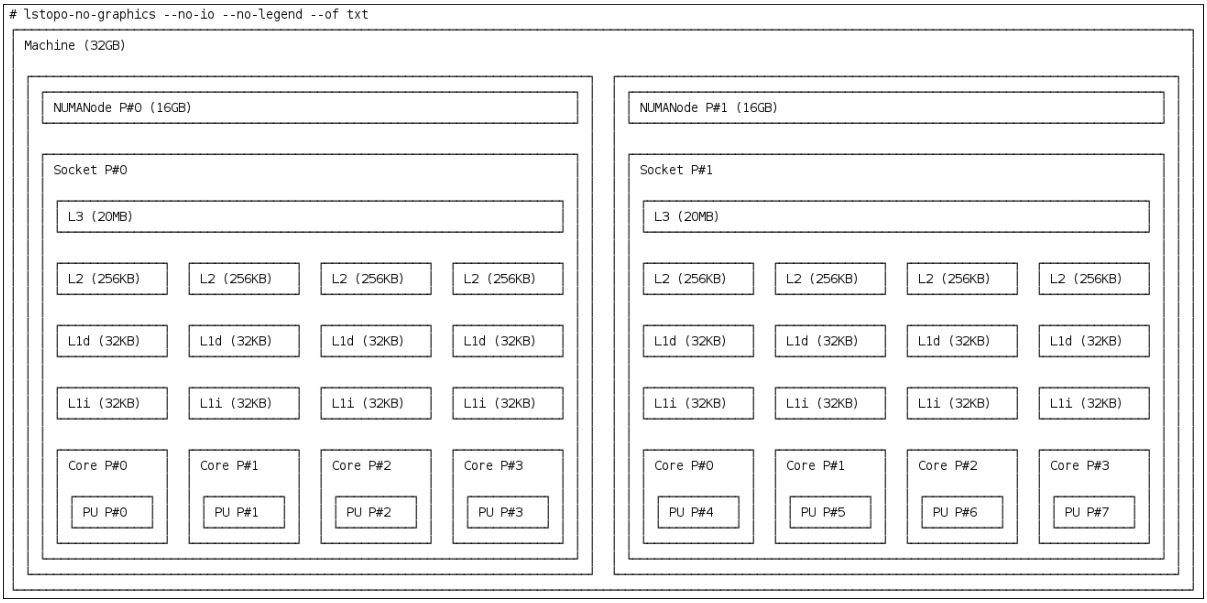
- 必须安装 hwloc 软件包。

流程

1. 查看物理软件包中可用 CPU 的布局：

```
# lstopo-no-graphics --no-io --no-legend --of txt
```

图 27.1. 使用 lstopo-no-graphics 显示 CPU 布局



此命令对多线程应用程序很有用，因为它显示可用的内核和套接字数量以及 NUMA 节点的逻辑距离。

此外，hwloc-gui 软件包提供 lstopo 实用程序，它会生成图形输出。

2. 查看有关 CPU 的更多信息，如节点之间的距离：

```
# numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3
node 0 size: 16159 MB
node 0 free: 6323 MB
node 1 cpus: 4 5 6 7
node 1 size: 16384 MB
node 1 free: 10289 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

其他资源

- hwloc(7) man page

27.2. 使用 TUNED 的 ISOLATED_CORES 选项隔离 CPU

隔离 CPU 的初始机制是在内核引导命令行中指定引导参数 isolcpus=cpulist。对于 RHEL for Real Time，推荐的方法是使用 TuneD 守护进程及其 tuned-profiles-realtime 软件包。

先决条件

- 已安装 TuneD 和 tuned-profiles-realtime 软件包。

流程

1. 作为 root 用户，在文本编辑器中打开 /etc/tuned/realtime-variables.conf。

2. 设置 isolated_cores=*cpulist* 以指定您要隔离的 CPU。您可以使用 CPU 编号和范围。

   示例：

   ```
   isolated_cores=0-3,5,7
   ```

   这会隔离核心 0、1、2、3、5 和 7。

   在带有 8 个内核的两个套接字系统中，其中 NUMA 节点 0-3，NUMA 节点 1 具有内核 4-8，用于为多线程应用程序分配两个内核，请指定：

   ```
   isolated_cores=4,5
   ```

   这可防止将任何用户空间线程分配给 CPU 4 和 5。

要为不相关的应用程序选择来自不同 NUMA 节点的 CPU，请指定：

```
isolated_cores=0,4
```

这可防止将任何用户空间线程分配给 CPU 0 和 4。

3. 使用 tuned-adm 程序激活 realtime TuneD 配置集。

```
# tuned-adm profile realtime
```

4. 重启机器。

验证

- 在内核命令行中搜索 isolcpus 参数：

```
$ cat /proc/cmdline | grep isolcpus
BOOT_IMAGE=/vmlinuz-4.18.0-305.rt7.72.el8.x86_64 root=/dev/mapper/rhel_foo-root ro
crashkernel=auto rd.lvm.lv=rhel_foo/root rd.lvm.lv=rhel_foo/swap
console=ttyS0,115200n81 isolcpus=0,4
```

第 28 章 使用 NOHZ 和 NOHZ_FULL 参数隔离 CPU

nohz 和 nohz_full 参数修改指定 CPU 上的活动。要启用这些内核引导参数，您需要使用以下 TuneD 配置集之一： realtime-virtual-host、realtime-virtual-guest 或 cpu-partitioning。

nohz=on

减少一组特定 CPU 的定时器活动。

nohz 参数主要用于减少空闲 CPU 上的计时器中断。这有助于通过允许空闲 CPU 在降低电源模式下运行，从而帮助监管生命周期。虽然不适用于实时响应时间，但 nohz 参数不会直接影响实时响应时间。但是，需要激活 nohz _full 参数，对实时性能有积极影响。

nohz_full=*cpulist*

nohz_full 参数会以不同的方式处理指定 CPU 列表的计时器循环。如果将 CPU 指定为 nohz_full CPU，且 CPU 上只有一个可运行的任务，那么内核将停止将计时器守护进程发送到该 CPU。因此，运行应用程序可能会花费更多时间，而服务中断和上下文切换所花费的时间更少。

其他资源

- 配置内核 Tick 时间

第 29 章 限制 SCHED_OTHER 任务迁移

您可以使用 sched_nr_migrate 变量限制 SCHED_OTHER 迁移到其他 CPU 的任务。

先决条件

- 管理员特权。

29.1. 任务迁移

如果 SCHED_OTHER 任务生成大量其他任务，则它们将在同一 CPU 上运行。迁移 任务或 softirq 将尝试平衡这些任务，以便它们可在空闲 CPU 上运行。

可以调整 sched_nr_migrate 选项，以指定每次将移动的任务数量。因为实时任务具有不同的迁移方法，所以它们不会直接受到此问题的影响。但是，当 softirq 移动任务时，它会锁定 run 队列 spinlock，从而禁用中断。

如果需要移动大量任务，它会在中断禁用时发生，因此不允许同时进行计时器事件或唤醒。当将 sched_nr_migrate 设置为较大的值时，这可能会导致实时任务具有严重的延迟。

29.2. 使用 SCHED_NR_MIGRATE 变量限制 SCHED_OTHER 任务迁移

增加 sched_nr_migrate 变量提供来自 SCHED_OTHER 线程的高性能，该线程生成许多任务，以实时延迟费用计算。

对于以 SCHED_OTHER 任务性能为低实时任务延迟，这个值必须降低。默认值为 8。

流程

- 要调整 sched_nr_migrate 变量的值，请直接向 /proc/sys/kernel/sched_nr_migrate 回显：

```
~]# echo 2 > /proc/sys/kernel/sched_nr_migrate
```

验证

- 查看 /proc/sys/kernel/sched_nr_migrate 的内容：

```
~]# cat > /proc/sys/kernel/sched_nr_migrate
2
```

第 30 章 减少 TCP 性能高峰

生成 TCP 时间戳可能会导致 TCP 性能高峰。sysctl 命令控制 TCP 相关条目的值，设置在 /proc/sys/net/ipv4/tcp_timestamps 中找到的时间戳内核参数。

先决条件

- 管理员特权。

30.1. 关闭 TCP 时间戳

关闭 TCP 时间戳可能会降低 TCP 性能高峰。

流程

- 关闭 TCP 时间戳：

```
# sysctl -w net.ipv4.tcp_timestamps=0
net.ipv4.tcp_timestamps = 0
```

输出显示 net.ip4.tcp_timestamps 选项的值是 0。也就是说，TCP 时间戳被禁用。

30.2. 打开 TCP 时间戳

生成时间戳可能会导致 TCP 性能高峰。您可以通过禁用 TCP 时间戳来降低 TCP 性能高峰。如果您发现生成 TCP 时间戳没有导致 TCP 性能高峰，您可以启用它们。

流程

- 启用 TCP 时间戳。

```
# sysctl -w net.ipv4.tcp_timestamps=1
net.ipv4.tcp_timestamps = 1
```

输出显示 net.ip4.tcp_timestamps 的值为 1。也就是说，启用 TCP 时间戳。

30.3. 显示 TCP 时间戳状态

您可以查看 TCP 时间戳生成的状态。

流程

- 显示 TCP 时间戳生成状态：

```
# sysctl net.ipv4.tcp_timestamps
net.ipv4.tcp_timestamps = 0
```

值 1 表示生成时间戳。值 0 表示不会生成时间戳。

第 31 章 减少 CPU 性能高峰

内核命令行 skew_tick 参数可以平稳地进入运行具有延迟敏感的应用程序的大型系统。在 Linux 系统中，实时延迟增长的常见源是在 Linux 内核计时器循环处理程序中通用锁定的多个 CPU 冲突。

先决条件

- 有管理员权限。

流程

- 将 skew_tick 引导参数设置为 1：

第 32 章 使用 RCU 回调提高 CPU 性能

Read-Copy-Update (RCU)系统为在内核中相互排除线程的无锁定机制。由于执行 RCU 操作，在删除内存是安全的时，调用恢复有时会在 CPU 上排队。

使用 RCU 回调提高 CPU 性能：

- 您可以删除 CPU 作为运行 CPU 回调的候选者。
- 您可以分配一个 CPU 来处理所有 RCU 回调。此 CPU 称为内务 CPU。
- 您可以从消耗 RCU 卸载线程的责任中缓解 CPU。

这种组合减少了专用于用户工作负载的 CPU 的干扰。

先决条件

- 管理员特权。
- 安装了 tuna 软件包

32.1. 卸载 RCU 回调

您可以使用 rcu_nocbs 和 rcu_nocb_poll 内核参数来卸载 RCU 回调。

流程

- 要从运行 RCU 回调的候选中移除一个或多个 CPU，在 rcu_nocbs 内核参数中指定 CPU 列表，例如：

```
rcu_nocbs=1,4-6
```

或者

```
rcu_nocbs=3
```

第二个示例指示 CPU 3 为 no-callback CPU 的内核。这意味着，RCU 回调不会在 rcuc/$CPU 线程固定到 CPU 3 中完成，但在 rcuo/$CPU 线程中。您可以将此 trhead 移到 housekeeping CPU，以缓解 CPU 3 不会被分配的 RCU 回调作业。

32.2. 移动 RCU 回调

您可以分配一个 housekeeping CPU 来处理所有 RCU 回调线程。要做到这一点，使用 tuna 命令并将所有 RCU 回调移到 housekeeping CPU。

流程

- 将 RCU 回调线程移到 housekeeping CPU 中：

```
# tuna --threads=rcu --cpus=x --move
```

其中 *x* 是内务 CPU 的 CPU 号。

此操作缓解 CPU *X* 以外的所有 CPU，以处理 RCU 回调线程。

32.3. 从中断 RCU 卸载线程中分离 CPU

虽然 RCU 卸载线程可以在另一个 CPU 上执行 RCU 回调，但每个 CPU 负责推断对应的 RCU 卸载线程。您可以缓解来自这一责任的 CPU，

流程

- 设置 rcu_nocb_poll 内核参数。

此命令会导致计时器定期提升 RCU 卸载线程，以检查是否存在回调是否在运行。

32.4. 其他资源

- 避免在实时内核中出现 RCU Stalls

第 33 章 实时调度问题和解决方案

本节介绍实时调度问题和可用解决方案。

实时调度策略

RHEL for Real Time 的两个实时调度策略共享一个主要特征：它们运行直至被高优先级线程抢占，或直到它们被休眠或执行 I/O。如果是 SCHED_RR，则可由操作系统抢占线程，以便可以运行其他相等 SCHED_RR 优先级的线程。在这两种情况下，POSIX 规格没有提供任何置备，它们定义了策略以便降低优先级线程来获取任何 CPU 时间。

这个实时线程特征意味着可轻松编写一个应用程序，以单调来单调给定 CPU 的 100%。但是，这会导致操作系统出现问题。例如，操作系统负责管理系统范围的和每个 CPU 资源，并且必须定期检查这些资源描述的数据结构，并与它们执行内务操作。但是，如果一个核心由 SCHED_FIFO 线程进行单调，它无法执行其内务处理任务。最终，整个系统变得不稳定，可能会崩溃。

在 RHEL for Real Time 内核上，中断处理程序作为带有 SCHED_FIFO 优先级的线程运行。（十六进制默认优先级为 50 ）。高于中断处理器线程的 SCHED_FIFO 或 SCHED_RR 策略的 cpu-hog 线程可以防止中断处理程序运行。这会导致程序等待这些中断发送的数据丢失并失败。

实时调度程序节流

Red Hat Enterprise Linux for Real Time 附带了一个保护机制，它允许系统管理员通过实时任务分配使用。这种保护机制被称为实时调度程序节流。实时调度程序节流由 /proc 文件系统的两个参数控制：

- /proc/sys/kernel/sched_rt_period_us

  定义 microsecondss（微秒）的周期以视为 100% 的 CPU 带宽。默认值为 1,000,000 abrts（1 秒）。对周期值的改变必须非常顺利，因为周期太长或太小是平等危险。

- /proc/sys/kernel/sched_rt_runtime_us

  可用于所有实时任务的总带宽。默认值为 950,000 50,000s (0.95 s)，或者以其他单词，以 95% 的 CPU 带宽。将值设为 -1 表示实时任务可能需要最多使用 100% 的 CPU 时间。这只有在设计了实时任务并且没有明显的注意事项（如未绑定的轮询循环）时，才足够了。

  实时节流机制的默认值定义实时任务可使用 CPU 时间的 95%。剩余的 5% 将投入非实时任务，例如在 SCHED_OTHER 和类似调度策略下运行的任务。请务必注意，如果单个实时任务占

据了 95% 的 CPU 时间插槽，那么该 CPU 上的剩余时间任务将不会运行。只有非实时任务使用剩余的 5% 的 CPU 时间。

默认值的影响包括：

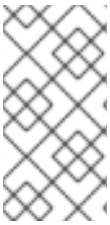
- 恶意实时任务不允许非实时任务运行，不会锁定系统。
- 实时任务具有最多的 CPU 时间可用，这可能会影响其性能。

其他资源

- real-Time 组调度

第 34 章 应用程序调整和部署

以下小节提供有关增强和开发 RHEL for Real Time 应用程序的提示。



注意

通常，尝试使用由 POSIX （可扩展操作系统接口）定义的 API。RHEL for Real Time 符合 POSIX 标准。RHEL for Real Time 内核的降低延迟也基于 POSIX。

34.1. 实时应用程序中的信号处理

传统的 UNIX 和 POSIX 信号具有其用途，特别是针对错误处理，但它们不适用于实时应用程序中的事件交付机制。这是因为当前 Linux 内核信号处理代码非常复杂，主要是因为旧的行为和需要支持的许多 API。这种复杂性意味着，在提供信号时所需的代码路径并非是最佳状态，应用程序可能会遇到长时间的延迟。

UNIX 信号背后的原始动机是不同"线程"不同的"线程"之间的多路控制线程（进程）。信号的行为方式类似于操作系统中断。也就是说，当向应用程序发送信号时，应用程序的上下文会被保存，并开始执行之前注册的信号处理程序。信号处理程序完成后，应用将返回到执行在信号发送时所处的位置。这种做法会变得复杂。

在实时应用程序中，无法信任信号。更好的选择是使用 POSIX Threads(pthreads)来分布工作负载并在不同组件间通信。您可以使用 mutexes、condition 变量和障碍的 pthreads 机制协调线程组。通过这些相对新结构的代码路径比信号的传统处理代码更为清晰。

其他资源

- POSIX Signal Model 的要求

34.2. 同步线程

sched_yield 命令是一个同步机制，它可以允许较低优先级线程运行。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。

较高的优先级线程可以调用 sched_yield 以允许其他线程运行。调用进程将移至正在该优先级运行的进程队列的尾部。如果发生这种情况时，没有以相同的优先级运行其他进程，则调用进程将继续运行。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。

当 SCHED_DEADLINE 任务调用 sched_yield（） 时，它会提供配置的 CPU，并且剩余的运行时会立即调整，直到下一期为止。sched_yield（） 行为允许任务在下一期的开始便开始。

调度程序可以更好地判断何时以及是否还有其他工作线程正在等待运行。避免在任何实时任务中使用 sched_yield。

流程

- 要调用 sched_yield（） 系统代码，请运行以下命令：

```
for(;;) {
        do_the_computation();
        /*
 * Notify the scheduler the end of the computation
         * This syscall will block until the next replenishment
         */
    sched_yield();
      }
```

SCHED_DEADLINE 任务使用基于冲突的搜索(CBS)算法限度，直到下一次执行循环开始为止。

其他资源

- pthread.h(P)
- sched_yield(2)
- sched_yield(3p)
- 当内核被替换 时，可以更改其行为 的技术。

34.3. 实时调度程序优先级

systemd 命令可用于为在引导过程中启动的服务设置实时优先级。这是在 *引导 期间更改服务优先级* 所述。

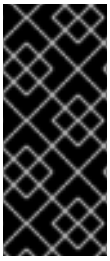
在该流程给出的示例中，可能会为一些内核线程获得非常高的优先级。这允许默认优先级与 Java(RTSJ)的实时时间规格的要求进行 良好的集成。RTSJ 需要一系列优先级从 10 到 89。

对于 RTSJ 没有被使用的部署，应用程序可使用大量调度优先级在 90 下。在调度优先级中的任何应用程序线程 49 时需要特别小心，因为它可以防止重要的系统服务运行，因为它可能会阻止重要系统服务运行。这可能导致无法预计的行为，包括阻止网络流量、阻止虚拟内存分页，以及因为文件系统日志被阻断而导致数据崩溃。

如果调度了任何应用程序线程的优先级 89，请确保线程只运行非常短的代码路径。如果不这样做，则 RHEL for Real Time 内核的低延迟功能会降低。

为非特权用户设置实时优先级

默认情况下，只有 root 用户可以更改优先级和调度信息。要授予非特权用户能够调整这些设置，最好的方法是将非特权用户添加到 realtime 组中。



重要

您还可以通过编辑 /etc/security/limits.conf 文件来更改用户特权。但是，这可能导致重复操作并导致常规用户的系统无法使用。如果您决定编辑此文件，请谨慎操作，并在进行更改前始终创建副本。

34.4. 其他资源

- HOWTO: 构建 RT-application

第 35 章 使用 TCP_NODELAY 提高网络延迟

默认情况下，TCP 使用 Nagle 的算法收集小的传出数据包以一次性发送所有。这可能导致延迟率更高。

先决条件

- 管理员特权。

35.1. 使用 TCP_NODELAY 的效果

在启用 TCP_NODELAY 的套接字上，每个发送数据包需要低延迟的应用程序都必须在启用 TCP_NODELAY 的套接字上运行。这会在事件发生后马上向内核发送缓冲区写入。

请注意，为了让 TCP_NODELAY 生效，应用程序必须避免执行小型、逻辑相关的缓冲区写入操作。否则，这些小的写入会导致 TCP 发送这些缓冲区作为单个数据包，从而降低整体性能。

如果应用程序有多个与逻辑相关的缓冲，且必须以一个数据包发送，请应用以下临时解决方案之一以避免性能不佳：

- 在内存中构建连续数据包，然后将逻辑数据包发送到使用 TCP_NODELAY 配置的套接字上的 TCP。
- 在配置了 TCP_NODELAY 的套接字上创建一个 I/O vector，并使用 writev 命令将其传递给内核。
- 使用 TCP_CORK。TCP_CORK 告知 TCP 在发送任何数据包前等待应用程序删除 cork。该命令会导致它接收的缓冲区附加到现有缓冲区中。这使得应用程序可以在内核空间中构建数据包，在使用为层提供提取的不同库时，可能需要此数据包。

当应用程序中的各个组件在内核中构建逻辑数据包时，该套接字应该被取消标记，允许 TCP 立即发送累积的逻辑数据包。

35.2. 启用 TCP_NODELAY

当发生事件时，TCP_NODELAY 将缓冲区写入内核，且无延迟。使用 setsockopt 命令启用 TCP_NODELAY。

流程

1. 使用 setsockopt 命令启用 TCP_NODELAY。

```
# int one = 1;

# setsockopt(descriptor, SOL_TCP, TCP_NODELAY, &one, sizeof(one));
```

2. 应用其中一个 follwing 临时解决方案以防止性能下降：

- 在内存中构建连续数据包，然后将逻辑数据包发送到使用 TCP_NODELAY 配置的套接字上的 TCP。
- 创建一个 I/O 向量，并使用 writev 在配置了 TCP_NODELAY 的套接字上的写入v 传递给内核。

35.3. 启用 TCP_CORK

TCP_CORK 可防止 TCP 发送任何数据包，直到套接字"不成"。

流程

1. 使用 setockopt 命令启用 TCP_CORK。

```
# int one = 1;

# setsockopt(descriptor, SOL_TCP, TCP_CORK, &one, sizeof(one));
```

2. 在由应用程序中的多个组件在内核中构建逻辑数据包后，禁用 TCP_CORK。

```
# int zero = 0;

# setsockopt(descriptor, SOL_TCP, TCP_CORK, &zero, sizeof(zero));
```

TCP 会立即发送累积逻辑数据包，而不等待应用中任何其他数据包。

35.4. 其他资源

- tcp(7)
- setsockopt(3p)
- setsockopt(2)

第 36 章 加载动态库

在开发实时应用程序时，请考虑在启动时解析符号，以避免程序执行期间出现非绝对延迟。请注意，在启动时解析符号可能会减慢程序初始化速度。

您可以使用 ld.so 设置 LD_BIND_NOW 变量（动态链路器/加载器/加载程序），指示动态链路器/应用程序启动时加载。

例如，以下 shell 脚本使用值 1 导出 LD_BIND_NOW 变量，然后使用 FIFO 的调度程序策略和优先级 1 运行程序。

```
#!/bin/sh

LD_BIND_NOW=1
export LD_BIND_NOW

chrt --fifo 1 /opt/myapp/myapp-server &
```

其他资源

- ld.so(8)

第 37 章 使用 MUTEX 防止资源过度使用

互斥(mutex)算法被用来防止过度使用常见资源。

37.1. MUTEX 选项

使用双向排除(mutex)算法来防止进程同时使用通用资源。快速用户空间 mutex(futex)是一个工具，它允许用户空间线程在不要求上下文切换到内核空间的情况下声明 mutex。

当您使用标准属性初始化 pthread_mutex_t 对象时，会创建一个私有、非递归、非忙碌和非优先级继承功能时。这个对象不提供 pthreads API 和 RHEL for Real Time 内核提供的任何 benfits。

要从 pthreads API 和 RHEL for Real Time 内核受益，请创建一个 pthread_mutexattr_t 对象。此对象存储为 futex 定义的属性。

注意

术语 futex 和 mutex 用于描述 POSIX 线程(pthread)mutex 结构。

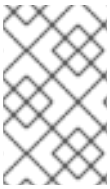
37.2. 创建 MUTEX 属性对象

要为 mutex 定义任何其他功能，请创建一个 pthread_mutexattr_t 对象。此对象存储为 futex 定义的属性。

流程

- 使用以下任一方式创建 mutex 属性对象：
  - pthread_mutex_t(*my_mutex*);
  - pthread_mutexattr_t(*&my_mutex_attr*);
  - pthread_mutexattr_init(*&my_mutex_attr*);

有关高级 mutex 属性的更多信息，请参阅高级 mutex 属性。



注意

本节不包括对函数的返回值的检查。这是必须始终执行的基本安全步骤。

37.3. 创建带有标准属性的 MUTEX

当您使用标准属性初始化 pthread_mutex_t 对象时，会创建一个私有、非递归、非忙碌和非优先级继承功能时。

流程

- 使用以下任一方法在 pthreads 中创建 mutex 对象：
  - pthread_mutex_t(*my_mutex*);
  - pthread_mutex_init(*&my_mutex*, *&my_mutex_attr*);

    其中 *&my_mutex_attr;* 是 mutex 属性对象。

37.4. 高级 MUTEX 属性

以下高级 mutex 属性可以存储在 mutex 属性对象中：

mutex 属性

共享和私有 mutexes

可以在进程间使用共享 mutexes，但可以创建更多的开销。

pthread_mutexattr_setpshared(&my_mutex_attr, PTHREAD_PROCESS_SHARED);

实时优先级继承

您可以使用优先级继承来避免优先级问题。

```
pthread_mutexattr_setprotocol(&my_mutex_attr, PTHREAD_PRIO_INHERIT);
```

强大的 mutexes

当短语时，pthread 中的强大的 mutexes 将发布。然而，这会显著降低成本。这个字符串中的 _NP 表示这个选项是非 POSIX 或不可移植。

```
pthread_mutexattr_setrobust_np(&my_mutex_attr, PTHREAD_MUTEX_ROBUST_NP);
```

mutex 初始化

可以在进程间使用共享 mutexes，但是它们会产生更多的开销。

```
pthread_mutex_init(&my_mutex_attr, &my_mutex);
```

37.5. 清理 MUTEX 属性对象

使用 mutex 属性对象创建 mutex 属性对象后，您可以保留属性对象来初始化相同类型的更多 mutexes，或者您可以进行清理。mutex 在这两种情况下都不受影响。

流程

- 使用 _destroy 命令清理属性对象。

```
pthread_mutexattr_destroy(&my_mutex_attr);
```

mutex 现在作为常规 pthread_mutex 运行，可以正常锁定、解锁和销毁。

37.6. 其他资源

- futex(7)

- **pthread_mutex_destroy(P)**
- **pthread_mutexattr_setprotocol(3p)**
- **pthread_mutexattr_setprioceiling(3p)**

第 38 章 分析应用程序性能

perf 是一个性能分析工具。它提供了一个简单的命令行界面，并提取 Linux 性能测量中的 CPU 硬件差异。perf 基于内核导出的 perf_events 接口。

perf 的一个优点是，它既是内核和架构中。可检查分析数据，而无需特定的系统配置。

先决条件

- 必须在系统中安装 perf 软件包。
- 管理员特权。

38.1. 收集系统范围统计信息

perf 记录命令用于收集系统范围统计数据。它可以在所有处理器中使用。

流程

- 收集系统范围性能统计数据。

```
# perf record -a
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.725 MB perf.data (~31655 samples) ]
```

在本例中，所有 CPU 都使用 -a 选项表示，进程在几秒钟后被终止。结果显示收集 0.725 MB 数据并将其保存到新创建的 perf.data 文件中。

验证

- 确保结果文件已创建。

```
# ls
perf.data
```

38.2. 归档性能分析结果

您可以使用 perf 归档命令分析其他系统上 perf 的结果。如果出现以下情况，可能不需要：

- 分析系统中已存在动态共享对象(DSO)，如二进制和库，如 ~/.debug/ 缓存。
- 两个系统都有相同的二进制文件集合。

流程

1. 从 perf 命令创建结果的存档。

```
# perf archive
```

2. 从归档创建 tarball。

```
# tar cvf perf.data.tar.bz2 -C ~/.debug
```

38.3. 分析性能分析结果

现在，可以使用 perf report 命令直接调查 perf 记录 功能中的数据。

流程

- 直接从 perf.data 文件或存档 tarball 中分析结果。

```
# perf report
```

报告的输出根据应用程序的最大 CPU 使用量排序。它显示示例是否已在进程的内核或用户空间中出现。

这个报告显示从中获取示例模块的信息：

  - 没有出现在内核模块中的内核示例标有符号 [kernel.kallsyms]。

- 发生在内核模块中的内核示例被标记为 [module] [ext4]。
- 对于用户空间的进程，结果可能会显示与进程关联的共享库。

  这个报告表示进程在内核或用户空间中是否发生。
- 结果 [.] 表示用户空间。
- 结果 [k] 表示内核空间。

精细细节可用于审核，包括适合经验丰富的 perf 开发人员的数据。

38.4. 列出预定义的事件

有一系列可用选项来获取硬件追踪点的活动。

流程

- 列出预定义的硬件和软件事件：

```
# perf list
List of pre-defined events (to be used in -e):
  cpu-cycles OR cycles                       [Hardware event]
  stalled-cycles-frontend OR idle-cycles-frontend    [Hardware event]
  stalled-cycles-backend OR idle-cycles-backend      [Hardware event]
  instructions                              [Hardware event]
  cache-references                          [Hardware event]
  cache-misses                              [Hardware event]
  branch-instructions OR branches                    [Hardware event]
  branch-misses                             [Hardware event]
  bus-cycles                                [Hardware event]

  cpu-clock                                 [Software event]
  task-clock                                [Software event]
  page-faults OR faults                          [Software event]
  minor-faults                              [Software event]
  major-faults                              [Software event]
  context-switches OR cs                         [Software event]
  cpu-migrations OR migrations                       [Software event]
```

```
alignment-faults                        [Software event]
emulation-faults                        [Software event]
...[output truncated]...
```

38.5. 获取有关指定事件的统计信息

您可以使用 perf stat 命令查看特定事件。

流程

1. 使用 perf stat 功能查看上下文切换数：

```
# perf stat -e context-switches -a sleep 5
^Performance counter stats for 'sleep 5':

         15,619 context-switches

      5.002060064 seconds time elapsed
```

结果显示发生 5 秒，15619 上下文切换。

2. 通过运行脚本查看文件系统活动。以下示例显示了脚本示例：

```
# for i in {1..100}; do touch /tmp/$i; sleep 1; done
```

3. 在另一个终端中运行 perf stat 命令：

```
# perf stat -e ext4:ext4_request_inode -a sleep 5
 Performance counter stats for 'sleep 5':

             5 ext4:ext4_request_inode

      5.002253620 seconds time elapsed
```

结果显示，在 5 秒之内要求创建 5 个文件，这表示有 5 个 索引节点 请求。

38.6. 其他资源

- perf 帮助 *COMMAND*

- **perf(1)man page**

第 39 章 使用压力测试实时系统

stress-ng 工具可衡量系统的能力，以便保持在不可或缺的状况下保持有效效率。stress-ng 工具是一个压力性工作负载生成器，用于加载并压力所有内核接口。它包括广泛的压力机制，称为 stressors。压力测试使机器用硬和出差硬件问题，如在系统过度运行和操作系统错误等。

270 不同的测试会超过 270。这包括用于练习浮动点、整数、位操作、控制流和虚拟内存测试的 CPU 特定的测试。

注意

使用 stress-ng 工具时要谨慎，因为某些测试可能会影响系统的rmal 区三点。这可能会影响系统性能，并导致过度的系统崩溃，这可能会难以停止。

39.1. 测试 CPU 浮动点单元和处理器数据缓存

floating-point 单元是处理器的功能部分，用于执行浮动点操作。浮点数单元处理数学操作，并将浮动数字或十进制计算更容易。

使用 --matrix-method 选项，您可以压力测试 CPU 浮动点操作和处理器数据缓存。

先决条件

- 系统的 root 权限

流程

- 要测试一个 CPU 上的浮动点 60 秒，请使用 --matrix 选项：

```
# stress-ng --matrix 1 -t 1m
```

- 要对多个 CPU 运行多个 stressors，请使用 --times 或 -t 选项：

```
# stress-ng --matrix 0 -t 1m

stress-ng --matrix 0 -t 1m --times
stress-ng: info:  [16783] dispatching hogs: 4 matrix
stress-ng: info:  [16783] successful run completed in 60.00s (1 min, 0.00 secs)
```

```
stress-ng: info:  [16783] for a 60.00s run time:
stress-ng: info:  [16783] 240.00s available CPU time
stress-ng: info:  [16783] 205.21s user time  ( 85.50%)
stress-ng: info:  [16783] 0.32s system time (  0.13%)
stress-ng: info:  [16783] 205.53s total time  ( 85.64%)
stress-ng: info:  [16783] load average: 3.20 1.25 1.40
```

带有 0 stressors 的特殊模式，查询要运行的可用 CPU 数量，从而无需指定 CPU 号。

需要的总 CPU 时间为 4 x 60 秒（240 秒），其中 0.13% 在内核中是 85.50%，所有 CPU 的 stress-ng 运行 85.64%。

- 要使用 POSIX 消息队列测试在进程间传递的消息，请使用 -mq 选项：

```
# stress-ng --mq 0 -t 30s --times --perf
```

mq 选项配置特定的进程，以使用 POSIX 消息队列强制上下文切换。此压力测试旨在降低数据缓存未命中。

39.2. 使用多个 STRESS 机制测试 CPU

stress-ng 工具运行多个 stress 测试。在默认模式中，它会并行运行指定的压力测试机制。

先决条件

- 系统的 root 权限

流程

- 运行多个 CPU 压力实例，如下所示：

```
# stress-ng --cpu 2 --matrix 1 --mq 3 -t 5m
```

在示例中，sess-ng 运行 CPU stressors 的两个实例，一个列表压力器和消息队列压力测试的三个实例来测试五分钟。

- 要并行运行所有 stress 测试，请使用 -all 选项：

```
# stress-ng --all 2
```

在本例中，s ess-ng 并行运行所有 stress 测试的两个实例。

- 要以特定顺序运行每个不同的压力，请使用 --seq 选项。

```
# stress-ng --seq 4 -t 20
```

在本例中，sess-ng 以 1 到 20 分钟后运行一次，每个压力的实例数与在线 CPU 的数量匹配。

- 要从测试运行中排除特定的 stressors，请使用 -x 选项：

```
# stress-ng --seq 1 -x numa,matrix,hdd
```

在本例中，stress-ng 运行所有的 stressors，每个实例都不包括 numa、hdd 和 key stressors 机制。

39.3. 测量 CPU HEAT 生成

为衡量 CPU 热生成，指定的压力器会在短时间内生成高时序，测试系统的无线可靠性和稳定性。使用 --matrix-size 选项，您可以在短时间内测量 CPU 温度。

先决条件

- 系统的 root 权限。

流程

1. 要在指定时间段高时测试 CPU 行为，请运行以下命令：

```
# stress-ng --matrix 0 --matrix-size 64 --tz -t 60

 stress-ng: info:  [18351] dispatching hogs: 4 matrix
 stress-ng: info:  [18351] successful run completed in 60.00s (1 min, 0.00 secs)
 stress-ng: info:  [18351] matrix:
 stress-ng: info:  [18351] x86_pkg_temp   88.00 °C
 stress-ng: info:  [18351] acpitz   87.00 °C
```

在这个示例中，stress-ng 配置处理器软件包 rmal 区域，使其在 60 秒期间达到 88 位 Celsius。

2. （可选）要在运行结束时输出报告，请使用 --tz 选项：

```
# stress-ng --cpu 0 --tz -t 60

 stress-ng: info:  [18065] dispatching hogs: 4 cpu
 stress-ng: info:  [18065] successful run completed in 60.07s (1 min, 0.07 secs)
 stress-ng: info:  [18065] cpu:
 stress-ng: info:  [18065] x86_pkg_temp   88.75 °C
 stress-ng: info:  [18065] acpitz   88.38 °C
```

39.4. 使用 BOGO 操作测量测试结果

stress-ng 工具可以通过测量每秒 bogo 操作来测量压力测试吞吐量。bogo 操作的大小取决于正在运行的压力。测试结果不精确，但提供性能的粗略估算。

您不能将此测量用作准确的基准测试指标。这些估计有助于了解不同内核版本或用于构建 压力 的不同编译器版本的系统性能变化。使用 --metrics-brief 选项显示可用 bogo 操作总数以及您机器上的矩阵压力。

先决条件

- 系统上的 root 权限

流程

- 要在 bogo 操作中测量测试结果，可将 与 --metrics-brief 选项搭配使用：

```
# stress-ng --matrix 0 -t 60s --metrics-brief

stress-ng: info: [17579] dispatching hogs: 4 matrix
stress-ng: info: [17579] successful run completed in 60.01s (1 min, 0.01 secs)
stress-ng: info: [17579] stressor bogo ops real time usr time sys time   bogo ops/s
bogo ops/s
stress-ng: info:  [17579]                (secs)   (secs)  (secs)  (real time) (usr+sys time)
stress-ng: info:  [17579] matrix  349322   60.00    203.23   0.19      5822.03      1717.25
```

--metrics-brief 选项显示测试结果以及 列表 压力运行的实时 bogo 操作总数，60 秒。

39.5. 生成虚拟内存压力

当内存压力下时，内核开始写出要交换的页面。您可以使用 --page-in 选项对虚拟内存进行压力测试，以强制非识别页面重新交换到虚拟内存。这会导致虚拟机进行大量练习。使用 --page-in 选项，您可以为 bigheap、mmap 和虚拟机(vm)stressors 启用此模式。--page-in 选项（ touch 分配的页面不在内核中）,强制它们进入页面。

先决条件

- 系统的 root 权限。

流程

- 要压力测试虚拟内存，请使用 --page-in 选项：

```
# stress-ng --vm 2 --vm-bytes 2G --mmap 2 --mmap-bytes 2G --page-in
```

在这个示例中，E ess-ng 测试内存在有 4GB 内存的系统上压力，小于分配的缓冲区大小，vm stressor 的 2 x 2GB 和启用了 --page-in 的 mmap stressor 的 2 x 2GB 压力。

39.6. 测试设备上的大型中断负载

以高频率运行计时器可产生大量中断负载。带有适当选择的计时器频率的 -timer 压力可以强制每秒许多中断。

先决条件

- 系统上的 root 权限。

流程

- 要生成中断负载，请使用 --timer 选项：

```
# stress-ng --timer 32 --timer-freq 1000000
```

在本例中，stress-ng 测试 32 个实例，位于 1MHz。

39.7. 在程序中生成主要页面错误

使用 stress-ng，您可以通过在未在内存中加载的页面生成主要页面错误来测试和分析页面错误率。在新的内核版本中，userfaultfd 机制会通知错误在进程虚拟内存布局中有关页面错误的故障发现线程。

先决条件

- 系统上的 root 权限。

流程

- 要在早期内核版本上生成主要页面错误，请使用：

```
# stress-ng --fault 0 --perf -t 1m
```

- 要在新内核版本上生成主要页面错误，请使用：

```
# stress-ng --userfaultfd 0 --perf -t 1m
```

39.8. 查看 CPU 压力测试机制

CPU 压力测试包含很多用于设置 CPU 的机制。您可以使用 哪个 功能输出输出来查看所有机制。

如果您没有指定测试方法，默认情况下，Eessor 会以大约循序检查所有压力者，从而利用每个压力测试 CPU。

先决条件

- 系统上的 root 权限。

流程

1. 打印所有可用的压力机制，使用 该选项 ：

```
# stress-ng --cpu-method which

cpu-method must be one of: all ackermann bitops callfunc cdouble cfloat clongdouble
```

```
correlate crc16 decimal32 decimal64 decimal128 dither djb2a double euler explog fft
fibonacci float fnv1a gamma gcd gray hamming hanoi hyperbolic idct int128 int64
int32
```

2. 使用 --cpu-method 选项指定特定的 CPU 压力方法：

```
# stress-ng --cpu 1 --cpu-method fft -t 1m
```

39.9. 使用验证模式

验证 模式会在测试处于活跃状态时验证结果。它完整性会检查测试运行的内存内容，并报告任何意外的故障。

所有 stressors 没有 验证模式，启用一个将减少 bogo 操作统计信息，因为在这个模式下运行的额外验证步骤。

先决条件

- 系统上的 root 权限。

流程

- 要验证压力测试结果，请使用 --verify 选项：

```
# stress-ng --vm 1 --vm-bytes 2G --verify -v
```

在本例中，s stress-ng 使用配置了 --verify 模式的 vm stressor 打印针对虚拟映射内存检查的详细内存检查。它能够检查内存中的读取和写入结果。

第 40 章 创建和运行容器

本节提供有关使用实时内核创建和运行容器的信息。

先决条件

- 安装 podman 和其他容器相关工具。
- 熟悉在 RHEL 8 中管理和管理 Linux 容器。
- 安装 kernel-rt 软件包和其他与时间相关的软件包。

40.1. 创建容器

您可以将所有以下选项用于实时内核和主 RHEL 内核。kernel-rt 软件包会带来潜在的确定性改进，并允许常见的故障排除。

先决条件

- 管理员特权.

流程

以下流程描述了如何配置与实时内核相关的 Linux 容器。

1. 创建您要用于容器的目录。例如：

   ```
   # mkdir cyclictest
   ```

2. 更改到该目录：

   ```
   # CD cyclictest
   ```

3. 登录到提供容器 registry 服务的主机：

```
# podman login registry.redhat.io
Username: my_customer_portal_login
Password: ***
Login Succeeded!
```

有关登录到 registry 主机的更多信息，请参阅 *构建、运行和管理容器*。

4. 创建以下 Dockerfile：

```
# vim Dockerfile
FROM rhel8
RUN subscription-manager repos --enable=rhel-8-for-x86_64-rt-rpm
RUN dnf -y install rt-tests
ENTRYPOINT cyclictest --smp -p95
```

5. 从包含 Dockerfile 的目录构建容器镜像：

```
# podman build -t cyclictest .
```

40.2. 运行容器

您可以使用 Dockerfile 运行容器。

流程

1. 使用 podman run 命令运行容器：

```
# podman run --device=/dev/cpu_dma_latency --cap-add ipc_lock --cap-add sys_nice --cap-add sys_rawio --rm -ti cyclictest

/dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.08 0.10 0.09 2/947 15

T: 0 ( 8) P:95 I:1000 C: 3209 Min: 1 Act: 1 Avg: 1 Max:  14

T: 1 ( 9) P:95 I:1500 C: 2137 Min: 1 Act: 2 Avg: 1 Max:  23

T: 2 (10) P:95 I:2000 C: 1601 Min: 1 Act: 2 Avg: 2 Max:   7

T: 3 (11) P:95 I:2500 C: 1280 Min: 1 Act: 2 Avg: 2 Max:  72
```

```
T: 4 (12) P:95 I:3000 C: 1066 Min: 1 Act: 1 Avg: 1 Max:   7

T: 5 (13) P:95 I:3500 C:  913 Min: 1 Act: 2 Avg: 2 Max:  87

T: 6 (14) P:95 I:4000 C:  798 Min: 1 Act: 1 Avg: 2 Max:   7

T: 7 (15) P:95 I:4500 C:  709 Min: 1 Act: 2 Avg: 2 Max:  29
```

本例演示了 podman run 命令以及所需的特定于时间的选项。例如：

- 第一位置(FIFO)调度程序策略可用于通过 --cap-add=sys_nice 选项在容器内运行的工作负载。这个选项还允许在调整实时工作负载时设置 CPU 关联性线程，另一个重要的配置维度。

- --device=/dev/cpu_dma_latency 选项使主机设备在容器内可用（由 cyclictest 工作负载用于配置 CPU 空闲时间管理）。如果指定的设备不可用，则会出现类似以下信息的错误：

  ```
  WARN: stat /dev/cpu_dma_latency failed: No such file or directory
  ```

  当预先使用类似这些错误消息的错误消息时，请参阅 podman-run(1)手册页。要获得在容器内运行的特定工作负载，其他 podman-run 选项可能会有所帮助。

  在某些情况下，您还需要添加 --device=/dev/cpu 选项来添加该目录层次结构，映射每个 CPU 设备文件，如 /dev/cpu/*/msr。

40.3. 其他资源

- 在 RHEL 8 中构建、运行和管理 Linux 容器

- 在 RHEL 8 中安装 Linux 实时内核

- 在 Red Hat Enterprise Linux 8 中配置 Linux 实时内核

第 41 章 显示进程的优先级

您可以使用 sched_getattr 属性显示进程优先级的相关信息，以及进程的调度策略信息。

先决条件

- 管理员特权

41.1. 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属性。

其他资源

- chrt(1) man page

41.2. 使用 CHRT 实用程序显示进程优先级

您可以显示指定进程的当前调度策略和调度优先级。

流程

- 使用 -p 选项运行 chrt 工具，指定正在运行的进程。

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

# chrt -p 476
pid 476's current scheduling policy: SCHED_OTHER
pid 476's current scheduling priority: 0
```

41.3. 使用 SCHED_GETSCHEDULER 库调用显示进程优先级

实时进程使用一组库调用来控制策略和优先级。您可以使用 sched_getscheduler（） 库调用来显示指定进程的调度程序策略。

先决条件

先决条件

- 包含 sched.h 文件。

流程

- 使用以下代码运行脚本：

```
#include <sched.h>

int policy;

policy = sched_getscheduler(pid_t pid);
```

policy 变量保存指定进程的调度程序策略。

其他资源

- sched_getscheduler(2) man page

41.4. 显示调度程序策略的有效范围

sched_get_priority_min（） 和 sched_get_priority_max（） 函数可用于检查给定调度程序策略的有效优先级范围。

流程

- 运行以下代码。

```
#include <stdio.h>
#include <unistd.h>
#include <sched.h>

main()
{

  printf("Valid priority range for SCHED_OTHER: %d - %d\n",
        sched_get_priority_min(SCHED_OTHER),
        sched_get_priority_max(SCHED_OTHER));

  printf("Valid priority range for SCHED_FIFO: %d - %d\n",
        sched_get_priority_min(SCHED_FIFO),
        sched_get_priority_max(SCHED_FIFO));
```

```
  printf("Valid priority range for SCHED_RR: %d - %d\n",
      sched_get_priority_min(SCHED_RR),
      sched_get_priority_max(SCHED_RR));
}
```

注意

如果系统不知道指定的调度程序策略，则该函数返回 -1 和 errno 被设置为 EINVAL。

注意

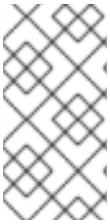
SCHED_FIFO 和 SCHED_RR 都可以是 1 到 99 范围中的任何数字。但是，POSIX 无法遵循这个范围，可移植程序应使用这些调用。

其他资源

- sched_get_priority_min(2) man page
- sched_get_priority_max(2) man page

41.5. 显示进程的时间片

SCHED_RR (round-robin)策略与 SCHED_FIFO （first-in、first-in、first-out）策略稍有不同。SCHED_RR 分配以轮循轮转相同的优先级的并发进程。这样，每个进程都会分配一个时间片。sched_rr_get_interval（） 函数报告分配给每个进程的时间片。



注意

虽然 POSIX 要求此功能 *必须* 仅处理配置为使用 SCHED_RR 调度程序策略运行的进程，sched_rr_get_interval（） 函数可检索 Linux 上任何进程的时间片长度。

timeslice 信息返回为 timespec。这是自 1970 年 1 月 1 日 00:00:00 GMT 的基础时间起的秒数和纳秒：

```
struct timespec {
  time_t tv_sec;  /* seconds / long tv_nsec; / nanoseconds */
}
```

流程

显示进程的时间片：

1. 使用以下代码创建一个测试脚本：

```
#include <stdio.h>
#include <sched.h>

main()
{
  struct timespec ts;
  int ret;

  /* real apps must check return values */
  ret = sched_rr_get_interval(0, &ts);

  printf("Timeslice: %lu.%lu\n", ts.tv_sec, ts.tv_nsec);
}
```

2. 在这些示例中，使用 sched_03 运行程序，具有不同的策略和优先级。

```
$ chrt -o 0 ./sched_03
Timeslice: 0.38994072
```

```
$ chrt -r 10 ./sched_03
Timeslice: 0.99984800
```

```
$ chrt -f 10 ./sched_03
Timeslice: 0.0
```

其他资源

- nice(2) man page
- getpriority(2) man page
- setpriority(2) man page

41.6. 显示进程的调度策略和相关属性

sched_getattr（） 函数查询当前应用于由 PID 识别的指定进程的调度策略。如果 PID 等于零，则检索调用进程的策略。

size 参数应反映对用户空间所知的 sched_attr 结构的大小。内核将 sched_attr::size 填充到其 sched_attr 结构的大小。

如果输入结构比较小，内核将返回所提供空间之外的值。因此，系统调用会失败并显示 E2BIG 错误。其他 sched_attr 字段已填写，如 sched_attr 结构 中所述。

流程

- 运行以下代码。

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <linux/unistd.h>
#include <linux/kernel.h>
#include <linux/types.h>
#include <sys/syscall.h>
#include <pthread.h>

#define gettid() syscall(NR_gettid) #define SCHED_DEADLINE 6 /* XXX use the proper
syscall numbers / #ifdef x86_64 #define NR_sched_setattr 314 #define
NR_sched_getattr 315 #endif struct sched_attr { u32 size; u32 sched_policy; u64
sched_flags; / SCHED_NORMAL, SCHED_BATCH / s32 sched_nice; / SCHED_FIFO,
SCHED_RR / u32 sched_priority; / SCHED_DEADLINE (nsec) */
    u64 sched_runtime; u64 sched_deadline;
    u64 sched_period; }; int sched_getattr(pid_t pid, struct sched_attr *attr, unsigned
int size, unsigned int flags) { return syscall(NR_sched_getattr, pid, attr, size, flags);
}



int main (int argc, char **argv)
{
    struct sched_attr attr;
    unsigned int flags = 0;
    int ret;

    ret = sched_getattr(0, &attr, sizeof(attr), flags);
    if (ret < 0) {
        perror("sched_getattr");
        exit(-1);
    }
```

```
    printf("main thread pid=%ld\n", gettid());
    printf("main thread policy=%ld\n", attr.sched_policy);
    printf("main thread nice=%ld\n", attr.sched_nice);
    printf("main thread priority=%ld\n", attr.sched_priority);
    printf("main thread runtime=%ld\n", attr.sched_runtime);
    printf("main thread deadline=%ld\n", attr.sched_deadline);
    printf("main thread period=%ld\n", attr.sched_period);

    return 0;
}
```

输出示例

```
main thread pid=321716
main thread policy=6
main thread nice=0
main thread priority=0
main thread runtime=1000000
main thread deadline=9000000
main thread period=10000000
```

41.7. SCHED_ATTR 结构

sched_attr 结构包含或定义指定线程的调度策略及其关联的属性。sched_attr 结构有以下形式：

```
struct sched_attr {
 u32 size;
 u32 sched_policy
 u64 sched_flags
 s32 sched_nice
 u32 sched_priority

 /* SCHED_DEADLINE fields */
        u64 sched_runtime
        u64 sched_deadline
        u64 sched_period
    };
```

sched_attr 数据结构

size

线程大小（以字节为单位）。如果结构的大小小于内核结构，则可将其他字段假定为 0。如果大小大于内核结构，内核会将所有附加字段验证为 0。

注意

当 sched_attr 结构大于 内核结构 时，sched_setattr（） 函数会失败，并包含内核结构和更新大小，以包含内核结构的大小。

sched_policy

调度策略

sched_flags

使用 fork（） 函数控制进程分叉时的调度行为。调用过程称为父进程，新进程称为子进程。有效值：

- 0 ：子进程从父进程继承调度策略。
- SCHED_FLAG_RESET_ON_FORK: fork（）：子进程不会继承父进程的调度策略。相反，它会被设置为默认的调度策略 (struct sched_attr){ .sched_policy = SCHED_OTHER }。

sched_nice

使用 SCHED_OTHER 或 SCHED_BATCH 调度策略时，指定要设置的 nice 值。nice 值是一个范围为 -20（ 高优先级）到 +19 （低优先级）的数值。

sched_priority

指定在调度 SCHED_FIFO 或 SCHED_RR 时要设置的静态优先级。对于其他策略，将 priority 指定为 0。

只能为截止时间调度指定 SCHED_DEADLINE 字段：

- sched_ runtime ：指定截止时间调度的运行时参数。该值以纳秒表示。
- SCHED_ DEADLINE：指定截止时间调度的 截止时间 参数。该值以纳秒表示。
- sched_period: 指定截止时间调度的 period 参数。该值以纳秒表示。

第 42 章 查看抢占状态

使用 CPU 的进程可以放弃它们正在使用的 CPU，可以是 voluntarily 或非自愿。

42.1. 抢占

进程可以自愿生成 CPU，因为它们已完成，或者因为正在等待事件，如磁盘中的数据、按键或网络数据包。

进程也可以非自愿地生成 CPU。这称为抢占功能，在优先级较高的进程希望使用 CPU 时发生。

抢占可以对系统性能造成负面影响，持续抢占可导致一个称为"垃圾箱(thrashing)的状态。当进程持续被抢占时，不会运行任何进程完成，就会发生这个问题。

更改任务的优先级有助于减少非自愿抢占功能。

42.2. 检查进程的抢占状态

您可以检查指定进程的自愿和非自愿抢占状态。状态存储在 /proc/*PID*/status 中。

先决条件

- 管理员特权。

流程

- 显示 /proc/*PID*/status 的内容，其中 *PID* 是进程的 ID。以下显示了 PID 为 1000 的进程的抢占状态。

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

第 43 章 使用 CHRT 工具为进程设置优先级

您可以使用 chrt 工具为进程设置优先级。

先决条件

- 管理员特权

43.1. 使用 CHRT 工具设置进程优先级

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属性。

流程

- 要设置进程的调度策略，请使用相应的命令选项和参数运行 chrt 命令。在以下示例中，受命令影响的进程 ID 为 1000，优先级(-p)为 50。

```
# chrt -f -p 50 1000
```

要使用指定的调度策略和优先级启动应用，请添加应用程序的名称以及相关的属性（如果需要）。

```
# chrt -r -p 50 /bin/my-app
```

有关 chrt 实用程序选项的更多信息，请参阅 chrt 实用程序选项。

43.2. CHRT 工具选项

chrt 实用程序选项包括命令选项和参数，指定命令的进程和优先级。

策略选项

-f

将调度程序策略设置为 SCHED_FIFO。

-o

将调度程序策略设置为 SCHED_OTHER。

-r

将调度程序策略设置为 SCHED_RR (round robin)。

-d

将调度程序策略设置为 SCHED_DEADLINE。

-p *n*

将进程的优先级设置为 ***n***。

将进程设置为 SCHED_DEADLINE 时，您必须指定 运行时、截止时间 和 句点 参数。

例如：

```
# chrt -d --sched-runtime 5000000 --sched-deadline 10000000 --sched-period 16666666 0
video_processing_tool
```

其中

- --sched-runtime 5000000 是以 nanoseconds 的运行时间。
- --sched-deadline 10000000 是 nanoseconds 的相对期限。
- --sched-period 16666666 是 nanoseconds。
- 0 是 chrt 命令所需的未使用优先级的占位符。

43.3. 其他资源

- chrt(1) man page

第 44 章 使用库调用设置进程的优先级

您可以使用 chrt 工具为进程设置优先级。

先决条件

- 管理员特权。

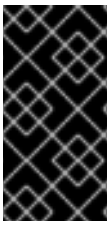
44.1. 设置优先级的库调用

以下库调用用于设置非实时进程的优先级。

- nice
- setpriority

这些功能调整非实时进程的 nice 值。nice 值充当调度程序如何订购要在处理器上运行的就绪非实时进程列表的建议。列表部的头条进程，然后直到列表进一步向下运行。

实时进程使用一组不同的库调用来控制策略和优先级，本节将详细介绍。



重要

以下是包含 sched.h 标头文件的所有功能。确保始终检查来自函数的返回代码。适当的 man page 概述了所使用的各种代码。

44.2. 使用库调用设置进程优先级

可以使用 sched_setscheduler（） 函数来设置调度程序策略和其他参数。目前，实时策略具有一个参数，sched_priority。这个参数用于调整进程的优先级。

sched_setscheduler（） 函数需要三个参数，格式为： sched_setscheduler(pid_t pid, int policy, const struct sched_param *sp);.

注意

sched_setscheduler(2) man page 列出了 sched_setscheduler 的所有可能返回值，包括错误代码。

如果进程 ID 为零，sched_setscheduler（） 函数将作用于调用的进程。

以下代码摘录将当前进程的调度程序策略设置为 SCHED_FIFO 调度程序策略，并将优先级设置为 50：

```
struct sched_param sp = { .sched_priority = 50 };
int ret;

ret = sched_setscheduler(0, SCHED_FIFO, &sp);
if (ret == -1) {
  perror("sched_setscheduler");
  return 1;
}
```

44.3. 使用库调用设置进程优先级参数

sched_setparam（） 函数用于设置特定进程的调度参数。然后可以使用 sched_getparam（） 函数进行验证。

与 sched_getscheduler（） 函数不同的是仅返回调度策略，sched_getparam（） 函数会返回给定进程的所有调度参数。

流程

使用以下代码摘录，它读取给定实时进程的优先级并增加两个代码：

```
struct sched_param sp;
int ret;

ret = sched_getparam(0, &sp);
sp.sched_priority += 2;
ret = sched_setparam(0, &sp);
```

如果这个代码用于真实应用程序，则需要从函数检查返回值并适当处理任何错误。

重要

请注意递增优先级。不断向调度程序优先级添加两个，如本例中所示，可能最终导致优先级无效。

44.4. 为进程设置调度策略和相关属性

sched_setattr（） 系统调用为 PID 中指定的实例 ID 设置调度策略及其关联的属性。当 pid=0 时，sched_setattr（） 作用于调用线程的进程和属性。

流程

- 调用 sched_setattr 指定调用调用操作的进程 ID，以及以下实时调度策略之一：

实时调度策略

SCHED_FIFO

调度 first-in 和 first-out 策略。

SCHED_RR

调度循环策略。

SCHED_DEADLINE

调度截止时间调度策略。

Linux 还支持以下非实时调度策略：

非实时调度策略

SCHED_OTHER

调度标准循环时间共享策略。

SCHED_BATCH

调度进程的"批处理"风格的执行。

SCHED_IDLE

调度非常低的优先级后台作业。SCHED_IDLE 只能用于静态优先级 0， 并且 nice 值对此策略没有影响。

此策略旨在以极低的优先级运行作业（比使用 SCHED_OTHER 或 SCHED_BATCH 策略的 +19 nice 值越低）。

44.5. 其他资源

- sched_attr-structure