导航

博客园
首页
新随笔
联系
订阅 XML
管理

| < | | | 2020年4月 | | | > |
|---|---|---|---|---|---|---|
| 日 | 一 | 二 | 三 | 四 | 五 | 六 |
| 29 | 30 | 31 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| 26 | 27 | 28 | 29 | 30 | 1 | 2 |
| 3 | 4 | 5 | 6 | 7 | 8 | 9 |

公告

昵称： 钱小小
园龄： 5年5个月
粉丝： 0
关注： 1
+加关注

搜索

找找看
谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

我的标签

linux基础(3)
网络(3)
转载(3)
GRO(2)
linux内核(2)
nova(1)
open-falcon(1)
OpenStack(1)
iptables(1)
jsonrpc(1)
更多

# 【翻译】QEMU内部机制：宏观架构和线程模型

系列文章：

1. 【翻译】QEMU内部机制：宏观架构和线程模型(本文)
2. 【翻译】QEMU内部机制：vhost的架构
3. 【翻译】QEMU内部机制：顶层概览
4. 【翻译】QEMU内部机制：内存

原文地址：http://blog.vmsplice.net/2011/03/qemu-internals-overall-architecture-and.html

原文时间：2011年3月5日

作者介绍：Stefan Hajnoczi来自红帽公司的虚拟化团队，负责开发和维护QEMU项目的block layer, network subsystem和tracing subsystem。

目前工作是multi-core device emulation in QEMU和host/guest file sharing using vsock，过去从事过disk image formats, storage migration和I/O performance optimization

## QEMU内部机制：宏观架构和线程模型

本文是QEMU内部机制系列文章的第一篇，目的是分享QEMU的工作原理，并让新的代码贡献者更容易地熟悉QEMU的基线代码。

运行一台vm包括执行vm的代码、处理定时器、IO并且响应外部命令(用户向qemu发送的命令)。为了完成所有这些事情，需要一个能够以安全的方式调解资源，并且不会在一个需要花费长时间的磁盘IO或外部命令操作的场景下暂停vm的执行的架构。有两种常见的用于响应多个事件源的编程架构：

1.并行架构：将热舞分配到进程或线程中以便同时执行，我把它称为"线程架构"
2.事件驱动架构：通过运行一个主循环来响应事件，将事件分发到事件处理器中。该方式一般是通过在多个文件描述符上执行select或poll等类型的系统调用实现的

QEMU实际上使用了一种将事件驱动编程和线程混合起来的架构。它这么做是为了避免事件驱动编程模型的单线程架构无法利用多核cpu的优势。而且，在某些场景下，编写一个独立的线程来执行某项任务比集成至事件驱动中实现起来更简单。但是，QEMU的核心是事件驱动的，它的大部分代码运行在事件驱动的环境下。

## QEMU的事件驱动核心

事件驱动架构是围绕将事件分发至处理函数的事件循环进行的。QEMU的主事件循环是main_loop_wait()函数，它执行下列任务：

> 1.等待文件描述符可读或可写。文件描述符的角色很关键，因为包括文件、套接字、管道和多种其他资源在内，都是文件描述符。文件描述符可以通过qemu_set_fd_handler()函数添加。
> 2.运行到期的定时器。定时器使用qemu_mod_timer()函数添加。
> 3.运行bottom-halves(中断语境中的下半部)，它与立即到期的定时器类似。BH用于避免重入和调用堆栈溢出，它可以通过qemu_bh_schedule()函数添加。

当一个文件描述符状态就绪、一个定时器到期或一个BH被调度时，事件循环使用回调来响应该事件。在回调环境下，有两个简单的规则：

> 1.核心代码的其他部分不会被同时执行，所以无需进行同步。对于其他核心代码而言，回调的执行是序列化、原子化得。在任何时刻，仅有一个线程拥有执行核心代码的控制权。
> 2.不应该执行任何阻塞式的系统调用或长时间的计算任务。因为事件循环在继续响应其他事件之前必须等待回调结束，所以回调应避免执行时间过长。破坏该规则会导致vm暂停，而且无法响应外部命令。

第二个规则有时比较难以实现，从而在QEMU中有一些阻塞式的代码。实际上，在qemu_aio_wait()函数中甚至嵌套了一个子事件循环，它用于等待全局事件循环的文件描述符的一个子集。但愿在以后的重构代码中能将这些违背规则的实现移除掉。新增的代码基本上不会有理由去违反该规则，如果一定要执行阻塞式任务，一个解决方案是将它交给独立的工作线程去完成。

## 将特定任务交给工作线程

**随笔分类**

Apache全景分析(1)
QT(1)
安全
编程基础(27)
技术调研(19)
数据库(1)
虚拟化(13)

**随笔档案**

2020年2月(1)
2019年7月(14)
2019年6月(16)
2019年5月(1)
2019年4月(6)
2016年3月(2)
2015年12月(2)
2015年3月(12)
2014年12月(2)
2014年11月(9)

**wxWidgets相关**

wxWidgets简明教程
wxWidgets的linux安装配置
wxWidgets在线帮助

**最新评论**

1. Re:【转】extern "C"的含义和用法
nm test.so可以查看符号表

--钱小小
2. Re:“云端融合”思想的自我摸索（很不靠谱）
目前，云计算技术的发展日新月异，它可以促进信息技术和数据资源充分合理利用，是信息化发展的必然趋势。下面是我对于云计算与操作系统的简单想法，其中内容不乏失实及误导之陈述，请领导批评指正。 经过将近十年的...
--钱小小
3. Re:debian包的补丁管理工具：quilt
来自ubuntu的dh_quilt_patch命令帮助：NAME dh_quilt_patch - apply patches listed in debian/patches/seriesSYNOP...
--钱小小

4. Re:在Linux下开发多语言软件(gettext解决方案)
附图bug修复报告:
--钱小小

**阅读排行榜**

1. golang类型断言的使用（Type Assertion）(3770)
2. 进程占用过高cpu的排查(977)
3. 【转】QEMU Monitor机制实例分析(792)
4. 【转】理解qemu对设备的模拟机制(618)
5. 【翻译】QEMU内部机制：宏观架构和线程模型(402)

**评论排行榜**

1. 在Linux下开发多语言软件(gettext解决方案)(1)
2. debian包的补丁管理工具：quilt(1)
3. “云端融合”思想的自我摸索（很不靠谱）(1)
4. 【转】extern "C"的含义和用法(1)

虽然很多IO操作可以在非阻塞式的方式执行，但是某些系统调用并没有非阻塞版本的实现。此外，有的长时间运行的计算任务会持续占用CPU，并且难以将其分解为回调的形式。在这些场景下，可以使用独立的工作线程将这些任务从QEMU的核心代码流程中单拿出来处理。

一个使用工作线程的示例是posix-aio-compat.c文件，它实现了异步文件IO。当QEMU的核心代码发起一个aio请求后，它会被放置于一个队列中。工作线程从该队列取出请求，在QEMU的核心代码流程之外处理该请求。由于他们是独立的线程，在处理中就可以放心的执行阻塞式的操作了。这种实现方式同时会处理线程与线程间的、线程与QEMU核心代码间的同步和交互操作。

还有一个例子是ui/vnc-jobs-async.c文件，它使用工作线程执行镜像压缩和编码等cpu消耗型任务。

QEMU的核心代码大部分都是线程不安全的，所以工作线程无法直接调用QEMU的核心代码。有一个例外是qemu_malloc()这样的简单常用函数是线程安全的，但这仅仅是一个例外。这种线程不安全的实现规则导致了工作线程不能将事件交互回QEMU的核心代码。

当一个工作线程需要通知QEMU的核心代码时，一个管道或qemu_eventfd()文件描述符会被加入到事件循环中。工作线程可以向文件描述符执行写操作，同时，事件循环会在该文件描述符准备就绪时调用回调函数。而且，必须使用信号来保证事件循环能够在任何场景下都有机会运行。posix-aio-compat.c中使用了该方式，在下一节介绍完vm的代码运行机制之后，可以对该方式有更深刻的理解。

## 执行vm的代码

上述内容主要关注的是QEMU的事件循环，但是QEMU更重要的用处在于能够执行vm的代码。

有两种执行vm代码的方式：Tiny Code Generator(TCG)和KVM。TCG使用动态二进制翻译技术来模拟vm，也称为Just-in-Time(JIT)编译技术。KVM利用现代cpu提供的硬件虚拟化技术直接在宿主的cpu上安全的执行vm的代码。本文不会关注这些具体的执行方式，重要的是这些执行方式都允许QEMU跳到vm代码并执行它。

跳到vm代码之后，QEMU的控制权就会交给vm。运行vm代码的线程不能处于事件循环下，因为此时vm拥有cpu的控制权。通常，运行vm代码的时间是有限的，因为对模拟设备寄存器的读写以及其他异常会中断vm代码的执行并将控制权交还给

QEMU。在某些极端场景下，vm可能会占用过多的时间从而不会让出控制权，此时，QEMU就会处于无法响应的状态。

为了解决这种vm长期占用QEMU线程的控制权的问题，可以使用signals(信号)来跳出vm。UNIX信号会让当前执行流程交出控制权并且调用信号处理函数。这样，QEMU就可以中断vm代码的执行并且回到主事件循环中，并且开始处理pending状态的事件。

该机制会导致正在执行vm代码的QEMU无法立即响应并处理新事件。大部分情况下QEMU最终会处理这些事件，但这种额外的延迟会导致性能问题。因此，定时器、IO完成和工作线程对QEMU核心的通知等事件会使用信号来确保事件循环立即运行。

现在，您可能想知道事件循环、smp多核架构的vm等宏观架构是什么样的。既然我们已经知道了QEMU的线程模型和执行vm代码的机制，下一节将对宏观架构进行介绍。

## iothread和非iothread架构

传统的QEMU架构是在单线程中执行vm代码和事件循环代码。QEMU默认采用该架构，称为非-iothread架构，使用默认编译选项./configure && make即可激活。QEMU线程会在接收到信号或遇到异常时结束vm代码的执行，获取控制权。然后它使用非阻塞式的select运行一次事件循环，之后它会继续执行vm的代码，并且重复此过程。

若vm启动时使用了-smp 2参数，QEMU并不会创建额外的线程。QEMU依旧使用单线程，但它会多路复用两个vcpus用于执行vm代码和事件循环。因此，非-iothread的方式无法发挥多核宿主机的优势，并且在开启smp的vm场景下性能堪忧。

但请注意，QEMU虽然只有一个核心线程，但它会有0个或多个工作线程，这些线程可能是临时的也可能是永久的。这些线程仅执行特定的任务并且不会执行vm代码或处理事件。我这里强调这一点是由于很多人搞不清楚工作线程，并且把它们误认为是由于使用了多vcpus而启动的线程。请记住，非-iothread架构下仅会启动一个QEMU核心线程。

最新的架构是QEMU为每个vcpu启动一个线程，外加一个独立的事件循环线程，称为iothread架构，编译时使用./configure --enable-io-thread进行激活。每一个vcpu都可以并行的执行vm的代码，为SMP机制提供真正的支持。iothread负责执行事件循环。通过维护一个全局互斥锁来保证QEMU的核心代码依旧不会被vcpu线程和iothread同时执行。大部分时间中，vcpu线程将执行vm的代码并且不会hold该全局互斥锁。同样地，在

大部分时间中iothread会阻塞在select调用，并且也不会hold该全局互斥锁。

请注意：TCG模式不是线程安全的，所以即使在iothread架构下，QEMU依旧是使用单线程多路复用vcpu的方式。只有KVM模式可以利用每vcpu线程的优势。

## 总结和展望

希望这可以帮助大家理解QEMU的宏观架构(该架构也被KVM使用)。
之后，本文的细节可能会更新。并且我希望能够将默认架构从非-iothread更改为iothread，甚至将非-iothread架构移除。我将会尝试在qemu项目更新时，更新本文。

原文如下：

## QEMU Internals: Overall architecture and threading model

This is the first post in a series on **QEMU Internals** aimed at developers. It is designed to share knowledge of how QEMU works and make it easier for new contributors to learn about the QEMU codebase.

Running a guest involves executing guest code, handling timers, processing I/O, and responding to monitor commands. Doing all these things at once requires an architecture capable of mediating resources in a safe way without pausing guest execution if a disk I/O or monitor command takes a long time to complete. There are two popular architectures for programs that need to respond to events from multiple sources:

1. Parallel architecture splits work into processes or threads that can execute simultaneously. I will call this **threaded architecture**.
2. Event-driven architecture reacts to **events** by running a main loop that dispatches to event handlers. This is commonly implemented using the `select(2)` or `poll(2)` family of system calls to wait on multiple file descriptors.

QEMU actually uses a **hybrid architecture** that combines event-driven programming with threads. It makes sense to do this because an event loop cannot take advantage of multiple cores since it only has a single thread of execution. In addition, sometimes it is simpler to write a dedicated thread to offload one specific task rather than integrate it into an event-driven architecture. Nevertheless, the core of QEMU is event-driven and most code executes in that environment.

## The event-driven core of QEMU

An event-driven architecture is centered around the event loop which dispatches events to handler functions. QEMU's main event loop is `main_loop_wait()` and it performs the following tasks:

1. Waits for **file descriptors** to become readable or writable. File descriptors play a critical role because files, sockets, pipes, and various other resources are all file descriptors. File descriptors can be added using `qemu_set_fd_handler()`.
2. Runs expired **timers**. Timers can be added using `qemu_mod_timer()`.
3. Runs **bottom-halves** (BHs), which are like timers that expire immediately. BHs are used to avoid reentrancy and overflowing the call stack. BHs can be added using `qemu_bh_schedule()`.

When a file descriptor becomes ready, a timer expires, or a BH is scheduled, the event loop invokes a **callback**that responds to the event. Callbacks have two simple rules about their environment:

1. No other core code is executing at the same time so **synchronization is not necessary**. Callbacks execute sequentially and atomically with respect to other core code. There is only one thread of control executing core code at any given time.

2. No blocking system calls or long-running computations should be performed. Since the event loop waits for the callback to return before continuing with other events, it is important to avoid spending an unbounded amount of time in a callback. Breaking this rule causes the guest to pause and the monitor to become unresponsive.

This second rule is sometimes hard to honor and there is code in QEMU which blocks. In fact there is even a nested event loop in `qemu_aio_wait()` that waits on a subset of the events that the top-level event loop handles. Hopefully these violations will be removed in the future by restructuring the code. New code almost never has a legitimate reason to block and one solution is to use dedicated worker threads to offload long-running or blocking code.

## Offloading specific tasks to worker threads

Although many I/O operations can be performed in a non-blocking fashion, there are system calls which have no non-blocking equivalent. Furthermore, sometimes long-running computations simply hog the CPU and are difficult to break up into callbacks. In these cases dedicated **worker threads** can be used to carefully move these tasks out of core QEMU.

One example user of worker threads is `posix-aio-compat.c`, an asynchronous file I/O implementation. When core QEMU issues an aio request it is placed on a queue. Worker threads take requests off the queue and execute them outside of core QEMU. They may perform blocking operations since they execute in their own threads and do not block the rest of QEMU. The implementation takes care to perform necessary synchronization and communication between worker threads and core QEMU.

Another example is `ui/vnc-jobs-async.c` which performs compute-intensive image compression and encoding in worker threads.

Since the majority of core QEMU code is **not thread-safe**, worker threads cannot call into core QEMU code. Simple utilities like `qemu_malloc()` are thread-safe but that is the exception rather than the rule. This poses a problem for communicating worker thread events back to core QEMU.

When a worker thread needs to notify core QEMU, a pipe or a `qemu_eventfd()` file descriptor is added to the event loop. The worker thread can write to the file descriptor and the callback will be invoked by the event loop when the file descriptor becomes readable. In addition, a signal must be used to ensure that the event loop is able to run under all circumstances. This approach is used by `posix-aio-compat.c` and makes more sense (especially the use of signals) after understanding how guest code is executed.

## Executing guest code

So far we have mainly looked at the event loop and its central role in QEMU. Equally as important is the ability to **execute guest code**, without which QEMU could respond to events but would not be very useful.

There are two mechanism for executing guest code: **Tiny Code Generator** (TCG) and **KVM**. TCG emulates the guest using dynamic binary translation, also known as Just-in-Time (JIT) compilation. KVM takes advantage of hardware virtualization extensions present in modern Intel and AMD CPUs for safely executing guest code directly on the host CPU. For the purposes of this post the actual techniques do not matter but what matters is that both TCG and KVM allow us to jump into guest code and execute it.

Jumping into guest code takes away our control of execution and gives control to the guest. While a thread is running guest code it cannot simultaneously be in the event loop because the guest has (safe) control of the CPU. Typically the amount of time spent in guest code is limited because reads and writes to emulated device registers and other exceptions cause us to leave the guest and give control back to QEMU. In extreme cases a guest can spend an unbounded amount of time without giving up control and this would make QEMU unresponsive.

In order to solve the problem of guest code hogging QEMU's thread of control **signals** are used to break out of the guest. A UNIX signal yanks control away from the current flow of execution and invokes a signal handler function. This allows QEMU to take steps to leave guest code and return to its main loop where the event loop can get a chance to process pending events.

The upshot of this is that new events may not be detected immediately if QEMU is currently in guest code. Most of the time QEMU eventually gets around to processing events but this additional latency is a performance problem in itself. For this reason timers, I/O completion, and notifications from worker threads to core QEMU use signals to ensure that the event loop will be run immediately.

You might be wondering what the overall picture between the event loop and an SMP guest with multiple vcpus looks like. Now that the threading model and guest code has been covered we can discuss the overall architecture.

## iothread and non-iothread architecture

The traditional architecture is a single QEMU thread that executes guest code and the event loop. This model is also known as **non-iothread** or `!CONFIG_IOTHREAD` and is the default when QEMU is built with `./configure && make`. The QEMU thread executes guest code until an exception or signal yields back control. Then it runs one iteration of the event loop without blocking in `select(2)`. Afterwards it dives back into guest code and repeats until QEMU is shut down.

If the guest is started with multiple vcpus using `-smp`

2, for example, no additional QEMU threads will be created. Instead the single QEMU thread multiplexes between two vcpus executing guest code and the event loop. Therefore non-iothread fails to exploit multicore hosts and can result in poor performance for SMP guests.

Note that despite there being only one QEMU thread there may be zero or more worker threads. These threads may be temporarily or permanent. Remember that they perform specialized tasks and do not execute guest code or process events. I wanted to emphasise this because it is easy to be confused by worker threads when monitoring the host and interpret them as vcpu threads. Remember that non-iothread only ever has one QEMU thread.

The newer architecture is one QEMU thread per vcpu plus a dedicated event loop thread. This model is known as **iothread** or `CONFIG_IOTHREAD` and can be enabled with `./configure --enable-io-thread` at build time. Each vcpu thread can execute guest code in parallel, offering true SMP support, while the iothread runs the event loop. The rule that core QEMU code never runs simultaneously is maintained through a global mutex that synchronizes core QEMU code across the vcpus and iothread. Most of the time vcpus will be executing guest code and do not need to hold the global mutex. Most of the time the iothread is blocked in `select(2)` and does not need to hold the global mutex.

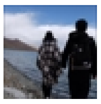
Note that TCG is not thread-safe so even under the iothread model it multiplexes vcpus across a single QEMU thread. Only KVM can take advantage of per-vcpu threads.

## Conclusion and words about the future

Hopefully this helps communicate the overall architecture of QEMU (which KVM inherits). Feel free to leave questions in the comments below.
In the future the details are likely to change and I

hope we will see a move to `CONFIG_IOTHREAD` by default and maybe even a removal of `!CONFIG_IOTHREAD`.
I will try to update this post as `qemu.git` changes.

分类: 虚拟化



钱小小
关注 - 1
粉丝 - 0
+加关注

0　0

« 上一篇： 【真的很先进】阿里云在2018-KVM Forum上分享的动态迁移实践
» 下一篇： 【翻译】QEMU内部机制：顶层概览

posted on 2019-06-19 11:07 钱小小 阅读(403) 评论(0) 编辑 收藏

**相关博文:**
· JavaI/O模型
· Java知多少（56）线程模型
· 高性能IO模型浅析
· reactor模型框架图和流程图libevent

· Netty高性能之Reactor线程模型
» 更多推荐...

**最新 IT 新闻**:
· 特斯拉经营范围新增电信业务等 并正式迁入上海自贸区
· 任正非最新讲话：艰苦奋斗的目的是过幸福生活
· 聚美优品宣布完成私有化，正式从纽交所退市
· 智能高空作业机器人公司史河科技完成3500万元Pre A+轮融资
· 亚马逊下调广告营销联盟佣金费率 或重创出版商营收
» 更多新闻...