2022 虎符 pwn hfdev (一)

原创

yongbaoii 于 2022-03-29 23:11:17 发布 76 收藏
分类专栏： CTF 文章标签： 网络安全

本文链接：https://blog.csdn.net/yongbaoii/article/details/123789460

前言

虎符的这道qemu本以为做不出来挺可惜，复现之后发现活该我做不出来
真是道好题。

我们首先需要学习一大堆这道题涉及到的前导知识。

QEMU内部机制：宏观架构和线程模型

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。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。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

qemu的事件驱动核心是glib事件循环。

> Glib事件循环机制提供了一套事件分发接口，使用这套接口注册事件源（source）和对应的回调，可以开发基于事件触发的应用。Glib的核心是poll机制，通过poll检查用户注册的事件源，并执行对应的回调，用户不需要关注其具体实现，只需要按照要求注册对应的事件源和回调
> Glib事件循环机制管理所有注册的事件源，主要类型有：fd，pipe，socket，和 timer。不同事件源可以在一个线程中处理，也可以在不同线程中处理，这取决于事件源所在的上下文（ GMainContext）。一个上下文只能运行在一个线程中，所以如果想要事件源在不同线程中并发被处理，可以将其放在不同的上下文

事件循环状态机

Glib对一个事件源的处理分为4个阶段：初始化，准备，poll，和调度。

Glib状态机的每个阶段都提供了接口供用户注册自己的处理函数。分别如下：

```
prepare:  gboolean (*prepare) (GSource *source, gint *timeout_);
Glib初始化完成后会调用此接口，此接口返回TRUE表示事件源都已准备好，告诉Glib跳过poll直接检查判断是否执行对应回调。返回FALSE表示需要poll机制监听事件源是否准备好，如果有事件源没准备好，通过参数timeout指定poll最长的阻塞时间，超时后直接返回，超时接口可以防止一个fd或者其它事件源阻塞整个应用


query: gint (g_main_context_query) (GMainContext *context, gint max_priority, gint *timeout_, GPollFD *fds, gint n_fds);
Glib在prepare完成之后，可以通过query查询一个上下文将要poll的所有事件，这个接口需要用户主动调用


check: gboolean (*check) (GSource *source);
Glib在poll返回后会调用此接口，用户通过注册此接口判断哪些事件源需要被处理，此接口返回TRUE表示对应事件源的回调函数需要被执行，返回FALSE表示不需要被执行


dispatch: gboolean (*dispatch) (GSource *source, GSourceFunc callback, gpointer user_data);
Glib根据check的结果调用此接口，参数callback和user_data是用户通过g_source_set_callback注册的事件源回调和对应的参数，用户可以在dispatch中选择直接执行callback，也可以不执行
```

prepare对应初始化到准备阶段

query对应准备到poll阶段

check对应poll到调度阶段

dispatch对应调度到初始化阶段

glib机制

https://www.cnblogs.com/silvermagic/p/9087881.html

https://blog.csdn.net/woai110120130/article/details/99701442

demo来自https://blog.csdn.net/huang987246510/article/details/90738137

```c
#include <glib.h>
/*
函数打印标准输入中读到内容的长度
*/
gboolean io_watch(GIOChannel *channel,
                  GIOCondition condition,
                  gpointer data)
{
    gsize len = 0;
    gchar *buffer = NULL;

    g_io_channel_read_line(channel, &buffer, &len, NULL, NULL);

    if(len > 0)
        g_print("%d\n", len);

    g_free(buffer);

    return TRUE;
}

int main(int argc, char* argv[])
{
    GMainLoop *loop = g_main_loop_new(NULL, FALSE); // 获取一个上下文的事件循环实例，context为NULL则获取默认的上下文循环
    GIOChannel* channel = g_io_channel_unix_new(1); // 将标准输入描述符转化成GIOChannel，方便操作

    if(channel) {
        g_io_add_watch(channel, G_IO_IN, io_watch, NULL);
        // 将针对channel事件源的回调注册到默认上下文，告诉Glib自己对channel的输入（G_IO_IN）感兴趣
        // 当输入准备好之后，调用自己注册的回调io_watch，并传入参数NULL。
        g_io_channel_unref(channel);
    }

    g_main_loop_run(loop); // 执行默认上下文的事件循环
    g_main_context_unref(g_main_loop_get_context(loop));
    g_main_loop_unref(loop);

    return 0;
}
```

```c
#include <glib.h>


typedef struct _MySource MySource;
/* 自定义事件源，继承自Glib的GSource类型*/
struct _MySource
{
    GSource _source; // 基类
    GIOChannel *channel;
    GPollFD fd;
};
/*事件源回调函数，读出iochannel中的内容，打印其长度*/
static gboolean watch(GIOChannel *channel)
{
    gsize len = 0;
    gchar *buffer = NULL;

    g_io_channel_read_line(channel, &buffer, &len, NULL, NULL);
    if(len > 0)
```

```
    if(len > 0)
     g_print("%d\n", len);
    g_free(buffer);

    return TRUE;
}
/*
状态机prepare回调函数，timeout等于-1告诉poll如果IO没有准备好，一直等待，即阻塞IO
返回FALSE指示需要poll来检查事件源是否准备好，如果是TRUE表示跳过poll
*/
static gboolean prepare(GSource *source, gint *timeout)
{
    *timeout = -1;
    return FALSE;
}
/*
状态机check回调函数，检查自己感兴趣的fd状态（events）是否准备好
用户通过设置events标志设置感兴趣的fd状态（包括文件可读，可写，异常等）
revents是poll的返回值，由内核设置，表明fd哪些状态是准备好的
函数功能：
当感兴趣的状态和poll返回的状态不相同，表示fd没有准备好，返回FALSE，Glib不发起调度
反之返回TRUE，Glib发起调度
*/
static gboolean check(GSource *source)
{
    MySource *mysource = (MySource *)source;

    if(mysource->fd.revents != mysource->fd.events)
     return FALSE;

    return TRUE;
}
/*
状态机dispatch回调函数，prepare和check其中只要有一个返回TRUE，Glib就会直接调用此接口
函数逻辑是执行用户注册的回调函数
*/
static gboolean dispatch(GSource *source, GSourceFunc callback, gpointer user_data)
{
    MySource *mysource = (MySource *)source;

    if(callback)
      callback(mysource->channel);

    return TRUE;
}
/*
当事件源不再被引用时，这个接口被回调
*/
static void finalize(GSource *source)
{
    MySource *mysource = (MySource *)source;

    if(mysource->channel)
      g_io_channel_unref(mysource->channel);
}

int main(int argc, char* argv[])
{
    GError *error = NULL;
    GMainLoop *loop = g_main_loop_new(NULL, FALSE); // 从默认上下文获取事件循环实例
```

```
    GSourceFuncs funcs = {prepare, check, dispatch, finalize}; // 声明用户定义的状态机回调
    /*
Glib允许用户自己定义事件源，但需要把Glib的事件源作为"基类"，具体实现是把GSource
作为自定义事件源的第一个成员，在创建事件源时传入状态机回调函数和自定义事件源的结构体大小
*/
    GSource *source = g_source_new(&funcs, sizeof(MySource));
    MySource *mysource = (MySource *)source;
/*
创建一个文件类型的GIOChannel，GIOChannel就是Glib对文件描述符的封装，实现其平台可移植性
GIOChannel在所有Unix平台上都可移植，在Windows平台上部分可移植
GIOChannel的fd类型可以是文件，pipe和socket
*/
    if (!(mysource->channel = g_io_channel_new_file("test", "r", &error))) {
        if (error != NULL)
            g_print("Unable to get test file channel: %s\n", error->message);

        return -1;
    }
/*获取GIOChannel的fd，放到GPollFD的fd域中*/
    mysource->fd.fd = g_io_channel_unix_get_fd(mysource->channel);
    /*设置感兴趣的文件状态，这里时文件可读状态*/
    mysource->fd.events = G_IO_IN;
    /*
    传给poll的文件描述符结构体
struct GPollFD {
    gint  fd;   // 文件描述符
    gushort  events;  // 感兴趣的文件状态
    gushort  revents; // 返回值，由内核设置
};
*/
    g_source_add_poll(source, &mysource->fd); // 将文件描述符添加到事件源中
    g_source_set_callback(source, (GSourceFunc)watch, NULL, NULL); // 设置事件源的回调函数
    /*
    设置事件源优先级，如果多个事件源在同一个上下文，这个事件源都准备好了，优先级高的事件源会被Glib优先调度
    */
    g_source_set_priority(source, G_PRIORITY_DEFAULT_IDLE);
    g_source_attach(source, NULL); //将事件源添加到Glib的上下文，此处上下文为NULL，表示默认的上下文
    g_source_unref(source);

    g_main_loop_run(loop); // Glib开始执行默认上下文的事件循环

    g_main_context_unref(g_main_loop_get_context(loop));
    g_main_loop_unref(loop);

    return 0;
}
```

事件循环初始化

qemu事件循环初始化遵循Glib接口，和普通应用初始化流程类似，在`qemu_init_main_loop`中实现，简化版代码如下：

```
static void
qemu_init_main_loop()
{
    GSource *src;

    qemu_aio_context = aio_context_new();  // 创建qemu定制的事件源qemu_aio_context
    gpollfds = g_array_new(FALSE, FALSE, sizeof(GPollFD));



    src = aio_get_g_source(qemu_aio_context); // 从定制事件源中获取Glib原始的事件源
    g_source_set_name(src, "aio-context");  // 设置事件源名称
    g_source_attach(src, NULL);     // 将事件源添加到Glib默认事件循环上下文
    g_source_unref(src);

    src = iohandler_get_g_source(); // 获取另一个定制的事件源 iohandler_ctx
    g_source_set_name(src, "io-handler");
    g_source_attach(src, NULL);     // 将事件源添加到Glib默认事件循环上下文
    g_source_unref(src);
}
```

状态机回调函数的定制

qemu丰富了Glib的事件源，状态机回调的实现逻辑变的复杂，但基本框架还是遵循Glib的接口。`aio_context_new`函数中的`aio_source_funcs`就是状态机回调函数的声明

```
static GSourceFuncs aio_source_funcs = {
    aio_ctx_prepare,
    aio_ctx_check,
    aio_ctx_dispatch,
    aio_ctx_finalize
};
```

事件源的定制

qemu定制的事件源，不仅实现了对描述符的监听，还实现了事件通知，时钟事件源监听和下半部。这些实现都体现在了描述事件源的结构体AioContext中。

QEMU主线程在qemu_init_main_loop函数里创建了运行在默认上下文的事件源qemu_aio_context，因此这个事件源会被QEMU主线程监听，作为主事件循环。这个主事件循环的上下文包含QEMU中绝大部分服务的fd，比如VNC server和QMP monitor服务端的socket等，QEMU对外暴露的服务，通过这个机制进行处理。具体的事件源创建函数如下：

```
struct AioContext {
    GSource source;
    QemuRecMutex lock;
    QLIST_HEAD(, AioHandler) aio_handlers;
    uint32_t notify_me;
    QemuLockCnt list_lock;
    struct QEMUBH *first_bh;
    bool notified;
    EventNotifier notifier;
    QSLIST_HEAD(, Coroutine) scheduled_coroutines;
    QEMUBH *co_schedule_bh;
    struct ThreadPool *thread_pool;
    QEMUTimerListGroup tlg;
    int external_disable_cnt;
    int poll_disable_cnt;
    int64_t poll_ns;
    int64_t poll_max_ns;
    int64_t poll_grow;
    int64_t poll_shrink;
        bool poll_started;
    int epollfd;
    bool epoll_enabled;
    bool epoll_available;
  };
```

1、source: glib主事件循环的事件源结构体，一个glib主事件循环可以挂接多个事件源，每个事件源都有其对应的处理函数。
2、aio_handlers：IO处理事件链表，链表的每个成员代代表了一个IO事件，里面集成了要探测的文件描述符，读写处理函数等。这个QEMU执行IO任务的主要的事件类型。
3、notify_me：QEMU对glib事件源进行了封装，最终加入gsource的事件源只有一个，其他所有的事件都是通过这个事件来分发的，这就意味着，QEMU的其他所有事件发生的，都必须发送这个加入gsource的事件，这个事件就是主事件循环的通知事件，通知主循环处理QEMU事件。notify_me字段是个用来优化事件发送的字段，当这个字段被置位时，代表主循环已准备好轮休事件，这时可以向glib循环发送事件，否则，就没有必要发送事件。
4、first_bh:QEMU支持的底半部机制，它运用于一些敏感场合不适宜执行大量代码时，这样可以把一些关键代码在敏感场合孩子小，而其他一些不关键的大量代码延后放在底半部里面执行。firt_bh字段存放的是低半部链表中的第一个底半部。
5、notified: 代表已经发出通知事件，通知主循环处理。
6、notifier:这个就是封装主循环通知事件的结构体，它其实是基于Linux的eventfd实现的。eventfd包含两个文件描述符，一个用于写，一个用于读，向写描述符写入，在读描述符可以读到写入的内容，eventfd机制可用于进程/线程间通信，也可用于内核和用户空间的通信。QEMU把读描述符加入主事件循环的事件源，写描述符用于发出通知，通知主事件循环处理事件。
7、scheduled_coroutines和co_schedule_bh两个字段是用来处理协程的，协程也是一种异步执行机制，QEMU的协程是基于底半部实现的。
8、tlg:定时器组链表，QEMU支持定时器机制，QEMU的定时器也是用来执行一些定时执行的任务。QEMU定时器也是主事件循环需要处理的一种任务。
9、剩下的字段poll、epoll等都是为了高效的监控通知时件而设计的，利用操作系统的poll或epoll等技术实现。

从QEMU定制的事件源来看，QEMU支持 4 种不同类型的任务即QEMU把其要处理的任务分为了 4 种不同的类型：iohander是其中最主要的用来处理io任务、低半部用来延迟执行一些不太紧急的任务、协程、定时器任务用来处理一些定时执行的任务。QEMU的任务不是定死的，都是可以根据需要动态的添加到这四中任务类型中。