漫谈 Wine 之一：

WINE 的系统结构

毛德操

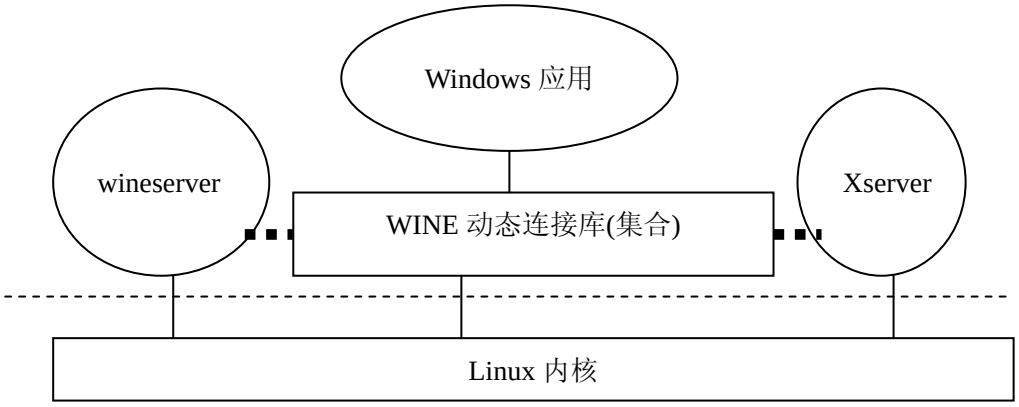
我们所要开发的 Linux 兼容内核主要的一个源泉就是 Wine。另一方面，我们之所以要开发兼容内核的的重要原因之一，就是因为 Wine 不能很好地解决在 Linux 内核上运行 Windows 应用的问题。为此，我们应该对 Wine 进行比较深入的研究，取得对 Wine 的深入理解，特别是要搞清两个问题：1）Wine 是什么样的？2）为什么 Wine 是那样的？如此才能明白我们为什么要开发兼容内核，以及我们将怎样来开发兼容内核。当然，对 Wine 的研究不应该孤立地进行，而必须把它放在 Linux 和 Windows 两个操作系统的背景下来开展。为此目的，我将撰写一些漫谈 Wine 的文章，围绕着上述的两个问题作一些介绍和分析，以期引起广泛的讨论和参与，然后再进入具体的开发阶段。这些文章将采用一事一议的方式，有的以介绍为主，有的以分析为主；有的围绕预定的题材，有的针对网友们提出的问题。

1．概述

Wine 是 Windows 应用软件与 Linux 内核之间的适配层，体现为一个 Wine 服务进程(wineserver)和一组动态连接库(相当于 Windows 的众多 DLL)。此外，Wine 的 GUI 用户界面仍依赖于 X11，由动态连接库 x11drv 和 X11 服务进程构成。其中 x11drv 是作为 Wine 与 X11 之间的界面，而 X11 服务进程本来就存在，因为 GNOME 也要通过 X11 服务进程操作图形界面。这样，在运行某个 Windows 应用时，系统中至少有 3 个进程与之有关：

- 应用进程本身。所有对 DLL 的调用均在该进程的上下文中运行。需要得到 Wine 的服务，或者通过 Wine 间接提供的其他(特别是内核)服务时，应用进程经由 Wine 所提供的各种动态连接库逐层往下调用。在 Wine 内部，这个进程往往需要通过 socket 和 pipe 与 Wine 服务进程通信，以接受服务进程的管理和协调；另一方面又可能经由另一个动态连接库 x11drv 通过别的 socket 与 X11 服务进程通信，向其发送图形操作请求并接收键盘和鼠标输入。
- Wine 服务进程。其主要的作用有：提供 Windows 进程间通信与同步的手段；Windows 进程和线程的管理，注册表服务，包括文件在内的各种 W32“对象”的管理；等等。
- X11 服务进程。图形的显示，以及键盘和鼠标输入。

这些进程以及动态连接库的关系如下图所示：



在这三个进程中，对桌面系统而言，Xserver 本来就存在，并非因为用了 Wine 才引入的，所以我们先关注其余两个进程。但是，从整个桌面系统的性能与效率来看，特别是对于图形/图像操作十分频繁的应用(如游戏)而言，Xserver 也是个瓶颈，所以我们将来还要回到这个问题上来，不过那是后话。

实际上，具体的 Windows 应用进程是从另一个进程，即 Wine 的作业装入程序 wine 转化过来的。用户一开始时启动的是 wine，而具体的 Windows 程序名是个参数，是由 wine 为具体目标程序的运行建立起与 Wine 服务进程的连接、装入目标程序，并转入目标程序运行的。

只要没有特别说明，下面讲到的“服务进程”都是指 Wine 服务进程。

2．Wine 服务进程

服务进程 wineserver 的存在独立于具体的 Windows 应用，所有的 Windows 应用在启动时都要建立起与服务进程的 socket 和 pipe 连接。所以，一般而言服务进程在启动具体 Windows 应用之前就应存在。不过，如果不存在，则作业装入程序 wine 会先启动服务进程运行，再与之建立连接。

在某种意义上，可以认为服务进程是在为 Windows 应用程序提供“远地过程调用(RPC)”、即跨进程的系统调用；其作用可以从它的过程调用表，即函数指针数组 req_handlers[ ] 看出个大概：

```
static const req_handler req_handlers[REQ_NB_REQUESTS] =
{
    (req_handler)req_new_process,
    (req_handler)req_new_thread,
    (req_handler)req_terminate_process,
    (req_handler)req_terminate_thread,
     ……
    (req_handler)req_load_dll,
    (req_handler)req_unload_dll,
     ……
    (req_handler)req_create_event,
    (req_handler)req_create_mutex,
    (req_handler)req_create_semaphore,
     ……
    (req_handler)req_create_file,
    (req_handler)req_alloc_file_handle,
    (req_handler)req_get_handle_fd,
    (req_handler)req_flush_file,
    (req_handler)req_lock_file,
    (req_handler)req_unlock_file,
    ……
    (req_handler)req_create_socket,
    (req_handler)req_accept_socket,
    (req_handler)req_set_socket_event,
    (req_handler)req_get_socket_event,
```

```
    (req_handler)req_enable_socket_event,
    (req_handler)req_set_socket_deferred,
      ……
    (req_handler)req_open_console,
    (req_handler)req_read_console_input,
    (req_handler)req_write_console_output,
      ……
    (req_handler)req_output_debug_string,
    (req_handler)req_continue_debug_event,
    (req_handler)req_debug_process,
    (req_handler)req_debug_break,
      ……
    (req_handler)req_set_key_value,
    (req_handler)req_get_key_value,
    (req_handler)req_load_registry,
      ……
    (req_handler)req_create_timer,
    (req_handler)req_open_timer,
    (req_handler)req_set_timer,
    (req_handler)req_cancel_timer,
      ……
    (req_handler)req_get_msg_queue,
    (req_handler)req_send_message,
    (req_handler)req_get_message,
    ……
    (req_handler)req_create_window,
    (req_handler)req_link_window,
    (req_handler)req_destroy_window,
    (req_handler)req_set_foreground_window,
    (req_handler)req_set_focus_window,
    (req_handler)req_set_global_windows,
      …… 共有将近 200 个函数
};
```

可见，服务进程参与了对进程/线程、进程间通信/同步(互斥)、文件、窗口、定时器、控制台、注册表等等资源的管理与服务。

服务进程对于进程、进程间通信等等的管理(与服务)与 Linux 内核所提供的管理并不冲突，而是对内核的补充。这是因为 Linux 内核所提供的是对于 Linux 进程/线程的管理，而 Windows 应用所期待的是对于 Windows 进程/线程的管理。服务进程之所以需要存在，其原因之一，就是为了要在(由 Linux 内核所所提供的)Linux 进程管理的基础上建立起对于 Windows 进程/线程的管理。所以，服务进程只提供其中附加的部分，例如进程/线程的调度就是由 Linux 内核提供的，而与服务进程无关。服务进程本身也是一个 Linux 进程，但不是 Windows 进程。同样，对别的资源的管理也有类似的问题。

如果用一句话来概括服务进程存在的理由，那就是“核内差异核外补”。所谓“核内差

异”，是指 Linux 和 Windows 两个内核之间的差异。而“核外补”，是指在核外、即用户空间设法加以补偿，以填平两个内核间的差异。“核外补”的手段可以有多种，例如用动态连接库也可以实现许多“核外补”；而服务进程是其中的手段之一，而且我们将看到这是效率很低的一种手段。显然，用服务进程这种效率很低的手段来实现“核外补”，只能是不得已而用之。后面(在另一篇漫谈中)我将分析 Wine 为什么不得已而采用了服务进程这种手段。

既然服务进程提供对进程间通信、注册表等等的管理，它自然就成为所有 Windows 进程的核心与枢纽。可以说，服务进程是整个 Wine 平台的核心。在某种意义上，服务进程还是 Linux 内核的外延，因为“核外补”的目的与结果正是在用户空间形成一个虚拟的(模拟的)“Windows 内核”。

事实上，在真正的 Windows 系统上也有个类似的进程叫 csrss，是所谓“Win32 子系统”的核心。这个进程之所以存在，是因为早期的 WinNT 3.51 内核更接近于微内核(微内核与宏内核之间的“核内差异”当然很大)。可是，从 WinNT 4.0 开始，其内核就再不能说是微内核、而应该说是带有微内核痕迹的宏内核了。所以，csrss 已经经历了很大的变化，现在的 csrss 只能说是早期 csrss 的残留了。而 Wine 的服务进程，则恰恰比较接近于早期的 csrss。所以，csrss 的这个变化过程，正好也为我们提供了一个参考和启示，使我们看到微软原来怎么想、后来怎么想、采取了一些什么措施、怎样发展/过渡到了现在的系统结构。对此我将专门写一篇漫谈、或者穿插在别的漫谈中加以论述。

在 Wine 的源代码中，服务进程的主体是 server/fd.c 中的一个无限循环 main_loop()。

[main() > main_loop()]

```
void main_loop(void)
{
    ……
    while (active_users)
    {
        timeout = get_next_timeout();

        if (!active_users) break;    /* last user removed by a timeout */

        ret = poll( pollfd, nb_users, timeout );
        if (ret > 0)
        {
            for (i = 0; i < nb_users; i++)
            {
                if (pollfd[i].revents)
                {
                    fd_poll_event( poll_users[i], pollfd[i].revents );
                    if (!--ret) break;
                }
            }
        }
    }
```

}

这个循环通过 poll()监视已经建立的 socket 与 pipe，每当从某个端口接收到信息时就通过 fd_poll_event()执行有关的处理程序。

服务进程启动之初只有一个 socket，即 master_socket，这是服务进程的大门，就好像是它的 IP 地址一样。在工具软件 wine 的帮助下，所有的 Windows 应用进程一开始就通过这个 socket 与服务进程联系。当服务进程从 master_socket 接收到请求时，就为这个客户进程建立起一个进程管理块，并为之建立起一对管道(pipe)，作为这个进程与服务进程之间的主要通信手段。与此有关的程序在 master_socket_poll_event()中，master_socket 的 fd_ops 数据结构 master_socket_fd_ops()保证了服务进程对此函数的调用(在 server/request.c 中)。

此后，当 Windows 应用进程需要调用服务进程时，就通过所建立的管道发出请求，而服务进程则通过 fd_poll_event()执行由管道的 fd_ops 数据结构 thread_fd_ops 中给定的函数 thread_poll_event()。在这个函数中，服务进程以调用请求中给出的调用号(例如 REQ_load_dll)为下标，在上述数组 req_handlers[ ]内找到相应的函数并加以执行，最后加以答复。至于 Windows 应用进程，则在发出请求以后就进入睡眠，等待来自服务进程的答复，然后又恢复运行。下面是 thread_poll_event()的伪代码：

```
[main() > main_loop() > fd_poll_event() > thread_poll_event()]
/* handle a client event */
static void thread_poll_event( struct fd *fd, int event )
{
    struct thread *thread = get_fd_user( fd );
    assert( thread->obj.ops == &thread_ops );

    if (event & (POLLERR | POLLHUP)) kill_thread( thread, 0 );
    else if (event & POLLIN) read_request( thread );
    else if (event & POLLOUT) write_reply( thread );
}
```

数据结构 struct fd 中含有具体管道属于(通往)哪一个进程/线程的信息，因此可以知道是谁发来的请求。在正常的情况下，这个函数会调用 read_request()，我们只看其主体部分：

```
[main() > main_loop() > fd_poll_event() > thread_poll_event() > read_request()]
/* read a request from a thread */
void read_request( struct thread *thread )
{
    ……
    if (!thread->req_toread)    /* no pending request */
    {
        if ((ret = read( get_unix_fd( thread->request_fd ), &thread->req,
                                sizeof(thread->req) )) != sizeof(thread->req)) goto error;
        if (!(thread->req_toread = thread->req.request_header.request_size))
        {
            /* no data, handle request at once */
            call_req_handler( thread );
```

```
            return;
        }
        if (!(thread->req_data = malloc( thread->req_toread )))
            fatal_protocol_error( thread,
                                  "no memory for %d bytes request\n", thread->req_toread );
    }
    ……
}
```

所谓 read_request()，实际上是“读取并执行”请求。具体的执行由 call_req_handler()启动，其主体是：

[main() > main_loop() > fd_poll_event() > thread_poll_event() > read_request()
 > call_req_handler()]

```
/* call a request handler */
static void call_req_handler( struct thread *thread )
{
    union generic_reply reply;
    enum request req = thread->req.request_header.req;

    current = thread;
    current->reply_size = 0;
    clear_error();
    memset( &reply, 0, sizeof(reply) );

    if (debug_level) trace_request();

    if (req < REQ_NB_REQUESTS)
    {
        req_handlers[req]( &current->req, &reply );
        if (current)
        {
            if (current->reply_fd)
            {
                reply.reply_header.error = current->error;
                reply.reply_header.reply_size = current->reply_size;
                if (debug_level) trace_reply( req, &reply );
                send_reply( &reply );
            }
            else fatal_protocol_error( current, "no reply fd for request %d\n", req );
        }
        current = NULL;
        return;
```

```
    }
    fatal_protocol_error( current, "bad request %d\n", req );
}
```

这里的指针数组 req_handlers[ ]，就是我们前面已经看到过的。

这种跨进程的“远地过程调用(RPC)”与对动态连接库的调用有个本质的区别，那就是：对动态连接库的调用不跨进程，目标过程的执行是在同一个(进程调度意义上的)上下文、同一个地址空间中完成的；而 RPC 则跨进程，是在不同的上下文和地址空间中完成的。这样，一方面起到了将服务进程中的许多数据结构跟应用进程相隔离的作用，另一方面也便于横向的(不同进程间的)协调和集中的管理。但是，在这种跨进程的调用中至少要涉及两次进程调度(与切换)，势必会引入一定程度的延迟和性能下降。

3．应用进程与服务进程的通信

应用进程最初时通过 socket 向服务进程“报到”，并建立起 pipe 连接，以后就通过 pipe 管道按一定的规程与服务进程通信，向服务进程请求 RPC 服务。为方便编程，Wine 的代码中定义并大量使用了几个宏操作，搞清这几个宏操作以及相关的过程对于代码的阅读和理解很有好处。

我们先看一段实际的代码：

```
NTSTATUS
WINAPI NtFlushBuffersFile(HANDLE hFile,    IO_STATUS_BLOCK* IoStatusBlock )
{
    NTSTATUS ret;
    HANDLE hEvent = NULL;

    SERVER_START_REQ( flush_file )
    {
        req->handle = hFile;
        ret = wine_server_call( req );
        hEvent = reply->event;
    }
    SERVER_END_REQ;
    ……
    return ret;
}
```

这是一个 Windows 应用进程(即客户端)向服务进程请求冲涮一个已打开文件时所调用的函数。参数 hFile 类似于 Linux 上的打开文件号(但有区别，将来我还要讲这个事)。

指针 req 和 reply 是在 SERVER_START_REQ 中定义的，分别指向用来发出请求和接收答复的数据结构，而函数 wine_server_call()则把请求发送给服务进程并等待其答复。

这里的 SERVER_START_REQ 和 SERVER_END_REQ 两个宏操作定义如下：

```
#define SERVER_START_REQ(type) \
    do { \
        struct __server_request_info __req; \
        struct type##_request * const req = &__req.u.req.type##_request; \
        const struct type##_reply * const reply = &__req.u.reply.type##_reply; \
        memset( &__req.u.req, 0, sizeof(__req.u.req) ); \
        __req.u.req.request_header.req = REQ_##type; \
        __req.data_count = 0; \
        (void)reply ; \
        do

#define SERVER_END_REQ \
        while(0); \
    } while(0)
```

结合上面的代码，此时首先把发送给服务进程的数据结构视为 struct flush_file_request，而把服务进程的答复则视为 struct flush_file_reply。至于调用号则为 REQ_flush_file，这个常数定义于 server_protocol.h 中的 enum request{}。两个宏操作以及夹在其中的代码构成嵌套的 do{ }while(0)语句。总的来说就是完成对 flush_file_request 的设置，将其发送给服务进程，等待其答复，并从答复中抽取所需的信息。

这样，经过 C 编译的预处理，前面的一段代码就成为这样：

```
    do {
        struct __server_request_info __req;
        struct type##_request * const req = &__req.u.req. flush_file_request;
        const struct type##_reply * const reply = &__req.u.reply. flush_file_reply;
        memset( &__req.u.req, 0, sizeof(__req.u.req) );
        __req.u.req.request_header.req = REQ_flush_file;
        __req.data_count = 0;
        (void)reply ;
        do
        {
            req->handle = hFile;
            ret = wine_server_call( req );
            hEvent = reply->event;
        }
        while(0);
    } while(0)
```

这里的函数 wine_server_call()就是把调用请求发送给服务进程，并等待服务进程的回答，其代码在 server.c 中。

```
unsigned int wine_server_call( void *req_ptr )
{
```

```
    struct __server_request_info * const req = req_ptr;
    sigset_t old_set;

    sigprocmask( SIG_BLOCK, &block_set, &old_set );
    send_request( req );
    wait_reply( req );
    sigprocmask( SIG_SETMASK, &old_set, NULL );
    return req->u.reply.reply_header.error;
}
```

这里的 send_request()将调用请求写入通向服务进程的管道，wait_reply()则睡眠等待(从另一个管道中)读取服务进程的回答。所以，不妨认为 wait_reply()是一条鸿沟，这里面隔着服务进程的相关服务，在本例中这就是实际冲涮具体已打开文件的操作。

如前所述，当服务进程接受到调用请求时会根据调用号从函数指针数组 req_handlers[ ]中找到相应的指针并加以调用。对于 flush_file，这个指针是 req_flush_file，其源代码为：

```
/* flush a file buffers */
DECL_HANDLER(flush_file)
{
    struct fd *fd = get_handle_fd_obj( current->process, req->handle, 0 );
    struct event * event = NULL;

    if (fd)
    {
        fd->fd_ops->flush( fd, &event );
        if ( event )
        {
            reply->event = alloc_handle( current->process, event, SYNCHRONIZE, 0 );
        }
        release_object( fd );
    }
}
```

显然，DECL_HANDLER 是个宏定义，其定义如下：

```
#define DECL_HANDLER(name) \
    void req_##name( const struct name##_request *req, struct name##_reply *reply )
```

这样，这个函数经过预处理以后就成为了：

```
void req_flush_file ( const struct flush_file_request *req, struct flush_file_reply *reply )
{
    ......
}
```

注意 req_flush_file()中的 fd->fd_ops->flush(fd, &event)，这里的函数指针 flush 实际上指向函数 file_flush()，这是由数据结构 file_fd_ops 给定的：

```
static const struct fd_ops file_fd_ops =
{
    file_get_poll_events,           /* get_poll_events */
    file_poll_event,                /* poll_event */
    file_flush,                     /* flush */
    file_get_info,                  /* get_file_info */
    file_queue_async,               /* queue_async */
    file_cancel_async               /* cancel_async */
};
```

我们接着看 file_flush()的代码：

```
[main_loop() > fd_poll_event() > thread_poll_event() > read_request()
  > call_req_handler() > req_flush_file() > file_flush()]

static int file_flush( struct fd *fd, struct event **event )
{
    int ret = (fsync( get_unix_fd(fd) ) != -1);
    if (!ret) file_set_error();
    return ret;
}
```

这里的 fsync()是 Linux 系统调用，而 get_unix_fd()将代表着 Windows 已打开文件的数据结构 fd 转换成代表着 Linux 已打开文件的“打开文件号”。

完成了所要求的服务以后，程序依次返回到 call_req_handler()中，并在那里调用 send_reply()：

```
[main_loop() > fd_poll_event() > thread_poll_event() > read_request()
  > call_req_handler() > send_reply()]

/* send a reply to the current thread */
static void send_reply( union generic_reply *reply )
{
    int ret;

    if (!current->reply_size)
    {
        if ((ret = write( get_unix_fd( current->reply_fd ),
                          reply, sizeof(*reply) )) != sizeof(*reply)) goto error;
    }
```

```
    ……
}
```

于是，Linux 内核将睡眠中的 Windows 应用进程唤醒，并调度其运行，控制又回到了 Windows 应用进程的手里。

显然，凡代码中出现 SERVER_START_REQ 的代码一定是在客户端，而出现 DECL_HANDLER 的代码一定是在服务端。

现在我们不妨以伪代码的形式粗粗地看一下整个操作的过程全貌：

```
NtFlushBuffersFile()
{
    ……
    SERVER_START_REQ( flush_file )
    {
        req->handle = hFile;
        wine_server_call( req )
        {
            切换到服务进程
            DECL_HANDLER(flush_file) ();
            切换回应用进程
        }
        hEvent = reply->event;
    }
    SERVER_END_REQ;
    ……
}
```

看了这个过程，读者恐怕自然会问：已打开文件是具体进程的“个人财产”，要冲涮尽可直接操作，即使“Windows 已打开文件”和“Linux 已打开文件”在语义上有些区别，也应该可以在动态连接库中加以转换(例如将 Handle 转换成打开文件号)补偿，为什么要由服务进程来完成冲涮呢？这不有点怪吗？其实，还有更怪的事情，不过那是我另一篇漫谈要讲的内容。另一方面，Wine 的设计/开发者当然清楚这个道理，之所以如此纯属无可奈何，那是我又另一篇漫谈的话题。