漫谈 Wine 之三：

Wine 的文件读写

毛德操

我在上一篇漫谈中谈到了 Windows 文件操作的一些特点，并且说明了：如果我们死守“核内差异核外补”这个策略，那么采用文件服务进程是个比较好的方法，但是缺点(代价)是效率较低。事实上，Wine 采用的正是这个方法。

在典型的采用文件服务进程的方法中，所有客户(client)进程的已打开文件都归服务进程所有，受服务进程管理，由服务进程进行实际的操作；而客户进程对已打开文件的访问则都由服务进程作为代理来完成。这种典型方法的效率是很低的，原因主要在于文件的读写操作。

在所有的文件操作中，最为频繁的是读/写操作，而效率最低的恰恰又是读/写操作。

以写文件操作为例，客户进程先准备好一个缓冲区，然后调用 API 上的相应函数，就 Wine 而言是 WriteFile()。这是由 Windows 的 Win32 API 规定的，类似于 Linux 上的 fwrite()。WriteFile()这个函数在动态连接库 kernel32.dll 中，执行的结果是向下调用另一个动态连接库 ntdll.dll 中的 NtWriteFile()。在 Windows 系统中，ntdll.dll 中的 NtWriteFile()直接进行系统调用，调用内核中的 NtWriteFile()，就像 Linux 的 C 库程序 fwrite()直接进行系统调用、调用内核中的 sys_write()一样。可是，如果采用的是文件服务进程的方法，NtWriteFile()就需要通过进程间通信把操作请求连同缓冲区的内容发送给文件服务进程。这里缓冲区内容的传递有可能引起缓冲区复制，并且可能要复制两次，一次是从客户进程的用户空间复制到内核中，另一次是从内核复制到服务进程的用户空间。就 Linux 内核而言，这是通过 copy_from_user()和 copy_to_user()两段汇编语言程序实现的。缓冲区的复制势必引起效率的显著下降。所以，要改进效率，或者说要防止效率的显著下降，首先就要设法避免缓冲区复制，而 Wine 在这方面做得可算巧妙。

为防止进程间通信过程中的缓冲区复制，Wine 利用了通过 Socket 实现文件访问授权的办法。我们知道，Unix 域的 Socket 有个特殊的功能，就是在通过 sendmsg()和 recvmsg()传递报文的同时让发送端进程将对其已打开文件的访问权授予接收端进程(读者可参阅“情景分析下册第 73-113 页”)。服务进程将对于目标文件的访问权授予客户进程之后，客户进程就可以直接进行读/写，而不必再由服务进程代劳、更不必因传递缓冲区内容而进行缓冲区复制了。且看 NtWriteFile()的代码：

[WriteFile() > NtWriteFile()]

```
NTSTATUS WINAPI NtWriteFile(HANDLE hFile, HANDLE hEvent,
                            PIO_APC_ROUTINE apc, void* apc_user,
                            PIO_STATUS_BLOCK io_status,
                            const void* buffer, ULONG length,
                            PLARGE_INTEGER offset, PULONG key)
{
    int unix_handle, flags;

    ……
```

```
    io_status->Information = 0;
    io_status->u.Status = wine_server_handle_to_fd( hFile, GENERIC_WRITE, &unix_handle,
&flags );
    if (io_status->u.Status) return io_status->u.Status;

    ……

    if (flags & (FD_FLAG_OVERLAPPED|FD_FLAG_TIMEOUT))
    {
        ……
    }

    if (offset)
    {
        FILE_POSITION_INFORMATION      fpi;

        fpi.CurrentByteOffset = *offset;
        io_status->u.Status = NtSetInformationFile(hFile, io_status, &fpi, sizeof(fpi),
                                                   FilePositionInformation);
        ……
    }

    /* synchronous file write */
    while ((io_status->Information = write( unix_handle, buffer, length )) == -1)
    {
        if ((errno == EAGAIN) || (errno == EINTR)) continue;
        if (errno == EFAULT)
        {
            io_status->Information = 0;
            io_status->u.Status = STATUS_INVALID_USER_BUFFER;
        }
        else if (errno == ENOSPC) io_status->u.Status = STATUS_DISK_FULL;
        else io_status->u.Status = FILE_GetNtStatus();
        break;
    }
    wine_server_release_fd( hFile, unix_handle );
    return io_status->u.Status;
}
```

先简单作一介绍。客户进程先通过 wine_server_handle_to_fd()与服务进程交互，从服务进程取得对已打开目标文件的访问授权。然后根据几个标志位决定所要求的是异步写还是同步写。实际应用中多数都是同步写，所以我们跳过针对异步写操作的代码。如果是指定了位置的写，那就先要通过 NtSetInformationFile()实现类似于 lseek()的操作，我们也把它跳过去。接下来就是 write()。注意这就是 Linux 的系统调用 write()，而且这是在客户进程一边直接进

行、而不是通过服务进程代理的。最后，还有个对 wine_server_release_fd()的调用，其作用是释放对目标文件的访问权。

我们先分段看 wine_server_handle_to_fd()的代码。

[WriteFile() > NtWriteFile() > wine_server_handle_to_fd()]

```
int wine_server_handle_to_fd( obj_handle_t handle, unsigned int access,
                                                    int *unix_fd, int *flags )
{
    obj_handle_t fd_handle;
    int ret, fd = -1;

    *unix_fd = -1;
    for (;;)
    {
        SERVER_START_REQ( get_handle_fd )
        {
            req->handle = handle;
            req->access = access;
            if (!(ret = wine_server_call( req ))) fd = reply->fd;
            if (flags) *flags = reply->flags;
        }
        SERVER_END_REQ;
        if (ret) return ret;

        if (fd != -1) break;
```

首先就是向服务进程发出 get_handle_fd 请求(实际上是 req_get_handle_fd)，这个请求通过当前进程通向服务进程的管道发送到服务进程。

当服务进程被调度运行时，根据这个特定的请求，所执行的是源代码中的 DECL_HANDLER(get_handle_fd)，实际上经过编译预处理以后是 req_get_handle_fd()。

```
/* get a Unix fd to access a file */
void req_get_handle_fd ( const struct get_handle_fd_request *req,
                                    struct get_handle_fd_reply *reply )
{
    struct fd *fd;

    reply->fd = -1;

    if ((fd = get_handle_fd_obj( current->process, req->handle, req->access )))
    {
        int unix_fd = get_handle_unix_fd( current->process, req->handle, req->access );
        if (unix_fd != -1) reply->fd = unix_fd;
```

```
            else if (!get_error())
            {
                assert( fd->unix_fd != -1 );
                send_client_fd( current->process, fd->unix_fd, req->handle );
            }
            reply->flags = fd->fd_ops->get_file_info( fd );
            release_object( fd );
        }
}
```

这里的 get_handle_fd_obj()根据具体的客户进程，以及这个客户进程所看到的 Windows“打开文件号”、即 Handle、查看一个对照表，还有一个参数是将要对此文件执行的访问类型(如读、写)，用于访问权限的检查。如果这个进程的这个 Handle 是有效的，那么服务进程这一头已经建立起从<进程，Handle>到目标文件在服务进程中的 Unix 打开文件号的映射，所以返回指向表现着这个映射关系的 struct fd 数据结构的指针 fd。然后进一步通过 get_handle_unix_fd()取得从<进程，Handle>到目标文件在客户进程中的 Unix 已打开文件号的映射，并取得该文件在客户进程中的打开文件号 unix_fd。如果客户进程对此文件的读/写已经不是第一次，那么映射已经建立，unix_fd 就不是-1，于是就把 reply->fd 设置成 unix_fd，作为对客户进程的回答。

读者也许感到奇怪，既然这已经不是第一次读/写这个已打开文件，那么客户进程自然应该知道它的打开文件号，怎么倒要来问服务进程呢？这个问题等一下就会明白。

反之，如果这是对方(客户进程)第一次读/写这个已打开文件，那么从<进程，Handle>到目标文件在客户进程中的已打开文件号的映射尚未建立，所以由 get_handle_unix_fd()返回的 unix_fd 为-1。但是，只要服务进程已经打开目标文件，则 fd->unix_fd、即目标文件在服务进程中的已打开文件号就不会是-1。在这种情况下，就通过 send_client_fd()把对这个已打开文件的访问权授予客户进程。我提到过 Wine 对文件读/写的优化，关键就在这里。

我们接下去看 send_client_fd()的代码。

[req_get_handle_fd() > send_client_fd()]

```
/* send an fd to a client */
int send_client_fd( struct process *process, int fd, obj_handle_t handle )
{
    int ret;

    ……

#ifdef HAVE_STRUCT_MSGHDR_MSG_ACCRIGHTS
    msghdr.msg_accrightslen = sizeof(fd);
    msghdr.msg_accrights = (void *)&fd;
#else   /* HAVE_STRUCT_MSGHDR_MSG_ACCRIGHTS */
    msghdr.msg_control    = &cmsg;
    msghdr.msg_controllen = sizeof(cmsg);
    cmsg.fd = fd;
```

```
#endif    /* HAVE_STRUCT_MSGHDR_MSG_ACCRIGHTS */

    myiovec.iov_base = (void *)&handle;
    myiovec.iov_len  = sizeof(handle);

    ret = sendmsg( get_unix_fd( process->msg_fd ), &msghdr, 0 );
    if (ret == sizeof(handle)) return 0;
    ……
}
```

注意这里的参数 fd 不是数据结构指针，而是服务进程的一个已打开文件号，现在就是要通过 Unix 域 Socket 通信的一个特殊功能把对这个已打开文件的访问权授予对方。要实现此种功能(可参阅“情景分析下册”)，发送端必须调用 sendmsg()，而接收端必须调用 rcvmsg()。这就是这儿调用 sendmsg()的原因。

回到 req_get_handle_fd()、即 DECL_HANDLER(get_handle_fd)的代码，下面就没有什么操作了，所以服务进程把 get_handle_fd_reply 数据结构作为响应报文发回给客户进程。注意这个发送与 sendmsg()的区别。前者是通过管道发送，并且此时客户进程也正在这个管道的另一端等待接收。而后者，如果调用了的话，是通过一个 Unix 域的 Socket 发送，但是客户进程此刻还没有企图从 Socket 的另一端接收，发送过去的报文将进入其接收队列。

客户进程受内核调度继续运行之后，首先取得响应报文发回来的打开文件号 fd。如果 fd 不是-1，那么这就是目标文件在客户进程这一头的 Unix 打开文件号，这就行了。可是，如果 fd 是-1，那就说明这是本进程第一次企图读/写这个文件，此时应该从上述的 Socket 获取对此已打开文件的授权。

我们继续往下看 wine_server_handle_to_fd()的代码：

[WriteFile() > NtWriteFile() > wine_server_handle_to_fd()]

```
        /* it wasn't in the cache, get it from the server */
        fd = receive_fd( &fd_handle );
        /* and store it back into the cache */
        ret = store_cached_fd( &fd, fd_handle );
        if (ret) return ret;

        if (fd_handle == handle) break;
        /* if we received a different handle this means there was
          * a race with another thread; we restart everything from
          * scratch in this case.
          */
    }    /* end for */

    if ((fd != -1) && ((fd = dup(fd)) == -1)) return STATUS_TOO_MANY_OPENED_FILES;
    *unix_fd = fd;
    return STATUS_SUCCESS;
}
```

显然，这段代码中 receive_fd()的目的就是从 Socket 获取授权。

[WriteFile() > NtWriteFile() > wine_server_handle_to_fd() > receive_fd()]

```
static int receive_fd( obj_handle_t *handle )
{
    struct iovec vec;
    int ret, fd;

#ifdef HAVE_STRUCT_MSGHDR_MSG_ACCRIGHTS
    struct msghdr msghdr;

    fd = -1;
    msghdr.msg_accrights    = (void *)&fd;
    msghdr.msg_accrightslen = sizeof(fd);
#else  /* HAVE_STRUCT_MSGHDR_MSG_ACCRIGHTS */
    struct msghdr msghdr;
    struct cmsg_fd cmsg;

    cmsg.len   = sizeof(cmsg);
    cmsg.level = SOL_SOCKET;
    cmsg.type  = SCM_RIGHTS;
    cmsg.fd    = -1;
    msghdr.msg_control    = &cmsg;
    msghdr.msg_controllen = sizeof(cmsg);
    msghdr.msg_flags      = 0;
#endif  /* HAVE_STRUCT_MSGHDR_MSG_ACCRIGHTS */

    msghdr.msg_name    = NULL;
    msghdr.msg_namelen = 0;
    msghdr.msg_iov     = &vec;
    msghdr.msg_iovlen  = 1;
    vec.iov_base = (void *)handle;
    vec.iov_len  = sizeof(*handle);

    for (;;)
    {
        if ((ret = recvmsg( fd_socket, &msghdr, 0 )) > 0)
        {
#ifndef HAVE_STRUCT_MSGHDR_MSG_ACCRIGHTS
            fd = cmsg.fd;
#endif
            if (fd == -1) server_protocol_error( "no fd received for handle %d\n", *handle );
```

```
            fcntl( fd, F_SETFD, 1 ); /* set close on exec flag */
            return fd;
        }
        if (!ret) break;
        if (errno == EINTR) continue;
        if (errno == EPIPE) break;
        server_protocol_perror("recvmsg");
    }
    /* the server closed the connection; time to die... */
    server_abort_thread(0);
}
```

不言而喻，这里的关键是 recvmsg()。我们知道，每个进程的“打开文件表”的主体是个指针数组，每个表项的下标就是“打开文件号”。如果一个表项代表着一个已打开文件、即一个读/写文件的上下文，这个指针就指向一个 struct file 数据结构。所谓 sendmsg()把对某个已打开文件的访问授权发送过来，实际上就是把这个指针(放在报文的头部作为控制信息)发送了过来。而 rcvmsg()，在接收报文中别的信息(如果有的话)的同时，负责在本进程的“打开文件表”中找到一个空闲的表项，然后把传过来的指针“安装”在这个位置上，并将其下标通过 msghdr、更确切地说是 msghdr.msg_control、传给用户空间。这样，当客户进程以后使用这个打开文件号来访问目标文件时，内核根据这个指针找到的就是服务进程使用的那个 struct file 数据结构(一进入内核，就不存在进程间的边界了)。这也解释了为什么要通过 Unix 域的 Socket 传送这种授权，因为 Unix 域 Socket 是用于本机的。而若是网络 Socket，则两个进程不在同一机器上，当然就不能通过指针指到对方的机器上去。

于是，当从 receive_fd()回到 wine_server_handle_to_fd()的时候，所返回的就是目标文件在客户进程本地的打开文件号。这样，客户进程就可以直接读/写这个文件了。这当然是一种优化，因为这么一来就可以避免跨进程的缓冲区复制了。

下面的事情很有意思。接着调用的是 store_cached_fd()。顾名思义，这是要建立一个从 Handle 到打开文件号的便查表，所以说是“缓存的(cached)”。那么这个便查表在那里呢？初一想应该是在客户进程这一边、即客户进程的用户空间，要不怎么说是“缓存”呢？我们看看这个函数的代码。

[WriteFile() > NtWriteFile() > wine_server_handle_to_fd() > store_cached_fd()]

```
inline static int store_cached_fd( int *fd, obj_handle_t handle )
{
    int ret;

    SERVER_START_REQ( set_handle_info )
    {
        req->handle = handle;
        req->flags  = 0;
        req->mask   = 0;
        req->fd     = *fd;
        if (!(ret = wine_server_call( req )))
```

```
        {
            if (reply->cur_fd != *fd)
            {
                /* someone was here before us */
                close( *fd );
                *fd = reply->cur_fd;
            }
        }
        else
        {
            close( *fd );
            *fd = -1;
        }
    }
    SERVER_END_REQ;
    return ret;
}
```

毫无疑问，这是“缓存”在服务进程那一边。我们在这里不深入到这个函数中去了，有兴趣的读者可以自己找来阅读。事实上，这个“缓存的”对照表就是前面服务进程那一头通过 get_handle_unix_fd()查找过程中用到的一些数据结构。这也是为什么如果不是第一次访问，服务进程就能找到从<进程，Handle>到目标文件在客户进程中的已打开文件号的映射的原因。在客户进程调用 store_cached_fd()之前，服务进程虽然通过 Socket 向客户进程授权，却无从知道其到达客户进程一边以后被安装在哪一个打开文件号上。现在，通过向服务进程登记，服务进程一边就建立起了这种映射。以后每当客户进程要访问这个文件时，就再通过 wine_server_handle_to_fd()查询，就像上面看到的那样；只是那时已经不是第一次，因此就不用那么费事了(不过仍是跨进程的查询)。

读者也许会问，既然客户进程已经知道了这个打开文件号，为什么就不在本地建立一个 Handle 与打开文件号的对照表，却要把这信息存储到服务进程那一边去呢？问题就出在遗传。作为客户进程的一个已打开文件，是有可能需要遗传的。可是，如果把 Handle 与打开文件号的对照表存放在客户进程的用户空间，就无法把这个对照表也一起遗传给子进程。另一方面，既然打定主意不触及内核、即不可能保存在内核中，那就只好把这信息存放在服务进程一边，要用的时候再去取了。当然，代价是效率的下降。

读者可能还会问，Handle 本身也在用户空间，也不能遗传。皮之不存，毛将焉附；子进程既然丢失了 Handle，这打开文件号还有什么意义呢？其实还是有意义的，因为父进程还可以通过别的手段把 Handle 的数值传递给子进程，包括至少下列几种手段：

1） 通过环境变量。

2） 通过命令行参数。

3） 作为编程约定(例如对于 stdin、stdout、stderr 三个通道的使用就是一种约定)。

由此看来，Wine 这样的处理确实是有道理的。

再回到 wine_server_handle_to_fd()中往下看。下面有个对 dup()的调用，这个系统调用的作用是为一个已打开文件增添一个打开文件号，使两个打开文件号标志着同一个文件的上下文。后面读者就会看到，NtWriteFile()中调用 write()时用的就是这个新增添出来的打开文件

号，而且在完成了 write()以后就又调用 wine_server_release_fd()、实际上就是 close()把这个新增添的打开文件号关闭掉。换言之，这个新增添的打开文件号就是专为本次读/写操作而增添，仅供本次读/写操作使用的。

想必读者又会问，这是为什么呢？既然客户进程已经有了一个打开文件号，为什么不直接使用它，而要再复制一个、用了以后又把它关闭呢？我猜想，这是为了防止线程之间的冲突。我们知道，一个进程里面可以有多个线程，都在共享同一个用户空间以及所有的资源，而调度的单位其实是线程而不是进程。另一方面，对于服务进程而言，已打开文件属于进程而不是个别的线程。这样，就可能会发生这样的情景：

1. 客户进程的线程 A 已经要求服务进程将一个已打开文件的 Handle 转换成客户进程本地的打开文件号。
2. 在线程 A 执行 write()之前，在某两条机器指令之间发生了中断，并且该次中断导致内核调度同一客户进程的线程 B 先运行(线程 A 处于用户空间，因而是可以被剥夺的)。
3. 线程 B 关闭了这个已打开文件。
4. 然后内核又调度线程 A 继续运行。
5. 线程 A 用前面取得的打开文件号调用 write()，但是这个文件实际上已被关闭。

这当然不是个令人愉快的情景。而 dup()和 close()的结合使用就解决了这个问题，因为 dup()的结果一方面是在本地增添了一个打开文件号，另一方面也使目标文件的使用计数递增。这样，即使线程 B 要关闭这个文件，由于其使用计数不会降到 0，就可以保证线程 A 能正常完成其操作。

回到 NtWriteFile()的代码，现在已经有了标志着目标文件的打开文件号，如果不要求移动文件内的读/写位置，也不要求进行异步写，那就是对 write()的直接调用、然后关闭通过 dup()增添的打开文件号了。

与典型的由服务进程代理完成文件读/写的方案相比，Wine 所作的优化无疑可以提高效率，因为这样可以避免跨进程的缓冲区复制。避免跨进程缓冲区复制的另一种办法是把缓冲区放在共享内存区中，但那同样需要通过进程间通信给对方发送通知(实际上还需要保证互斥操作)。

然而，尽管如此，Wine 的文件操作的效率是否就已经够好、可以令人满意了呢？这要看参照物是什么。显然，参照物就是 Windows 本身。特别是在许多用户已经有了先入之见的条件下，只要 Wine 的性能有明显可以觉察的差距，就有可能成为用户拒绝使用 Linux 的理由。如果按这样的要求来考虑，那么 Wine 的文件操作效率还是不能令人满意。

以上述的写文件操作为例(实际上读操作也是一样)，从整个过程看，最低限度的操作有这么一些：

- 客户进程通过进程间通信向服务进程发送 get_handle_fd 请求。注意这里至少有两次系统调用，一次是通过 send_request()调用对于管道的 write()，然后又通过 wait_reply()调用对于管道的 read()。
- 内核调度服务进程运行。
- 服务进程根据 Handle 找到目标文件在客户进程中的打开文件号，并通过进程间通信将其发送给客户进程。这里至少有三次系统调用，一次是通过 send_reply()调用对于管道的 write()，然后是系统调用 poll()，接着又通过 read_request()调用对于具体管道的 read()。这里的系统调用 poll()类似于 select()。
- 内核调度客户进程运行。

- 客户进程通过系统调用 dup()复制一个新的、临时的已打开文件号。
- 然后用临时的已打开文件号调用 write()。
- 最后通过系统调用 close()关闭临时的已打开文件号。

与由服务进程代理的写文件操作相比，这个过程避免了两次缓冲区复制，但是多了一次 dup()系统调用和一次 close()系统调用。二者相抵，这样做还是划算的。但是如果与 Windows 上单个 NtWriteFile()系统调用相比，或是与 Linux 上单个 write()系统调用相比，则显然效率要低很多，整个过程中的系统调用竟有八次之多。如果我们对有关的代码作更深入的分析，就可以估算出平均要多执行多少条指令、需要耗费多少时间。

然而，如果从桌面应用的角度看，则用户所感受到的性能降低甚至可能更甚于按上列操作过程计算出来的程度。这里的问题在于两次进程调度。

当客户进程通过 IPC 向服务进程发送请求时，服务进程被唤醒，内核则进行进程调度和切换。显然，用户此时希望服务进程能马上被调度运行，这样才能保证快速的响应。可是能否如愿以偿呢？这就不一定了。如果此时有个优先级更高的进程因某种原因(例如中断)而进入了就绪状态，服务进程就只好往后靠了。所以，在上列的第一和第二两个动作不一定是连续的，中间可能会有别的进程插进来。而对于桌面用户，这个优先级更高的进程得倒运行的正面效果也许感觉不到，而负面的效果却感觉到了。同样，当服务进程通过 IPC 回答客户进程时，客户进程也完全可能一时不能被调度运行。读者可能会想，只要把服务进程的优先级别设置的足够高，就可以保证服务进程立即被调度运行了。而且这样做也有充分的理由，因为服务进程平时都在睡眠，只是在需要它服务时才“用兵一时”，服务一完就又睡眠了。好，就算是那样，可是客户进程呢？总不能把客户进程的优先级也设置得足够高吧？

其实，我们在这里需要一种机制，就是给予客户进程和服务进程之间的通信特殊的待遇，使得每当一方向另一方发送请求或响应时就临时把对方的优先级提得很高。显然，Linux 内核并不提供这样的机制。这是因为，作为宏内核，Linux 并不鼓励其应用软件、特别是对于性能有较高要求的应用软件、在本机上采用“服务进程/客户进程”的系统结构。宏内核的一个重要特点就是在内核中提供、而不是依靠服务进程来提供各种系统服务。相比之下，微内核则往往依靠服务进程来提供系统服务(所以内核才能由宏变微)。

在这个问题上，我们不妨回顾一下 WinNT 及其服务进程 csrss 的演变。早期的 WinNT 是以微内核系统 MACH 为蓝本的。到 WinNT 4.0，一方面是由于原 VMS 团队骨干的加入，才变得像 VMS 了，所以 WinNT 4.0 既是里程碑又是分水岭。总之，在 WinNT 4.0 之前，例如 WinNT 3.51，当时的 WinNT 内核更接近于微内核。再加上当时的设计者想在 WinNT 内核上既能支持其“本土”的应用，又能支持 POSIX 应用和 OS/2 应用，因此就决定在较大程度上采用服务进程来为应用提供系统服务。这样，对于不同的应用，例如对于 POSIX 应用，只要换上不同的服务进程和不同的 DLL 就可以了。这就是 WinNT 的所谓不同的“应用子系统”。这个思路，实际上跟 Wine 试图通过服务进程来让 Linux 内核兼容 Windows 应用是一致的(不过微软在这方面是说的多做的少，而 Wine 倒是认认真真在做)。WinNT 本土应用、即所谓“Win32 子系统”的服务进程叫 csrss，在早期 WinNT 系统中起着至关重要的作用。其作用主要倒不在文件操作(文件操作不经过服务进程)，而在于对图形界面和窗口的操作。显然，这里也有上面所说的性能问题。而在引起性能下降的诸多因素中，微软的研发人员认为客户进程和服务进程之间的调度是个重要因素。为解决这个问题，微软设计了一种称为“快速本地过程调用(Quick LPC)”的机制，其核心是所谓“事件对(Event Pair)”和对事件对实行操作的系统调用、以及基于事件对的特殊进程调度。一个事件对由“高”和“低”两个事件量构成，客户进程和服务进程各执一端，例如服务进程等待客户进程发来请求时就在“高”

事件量上睡眠，而客户进程在等待服务进程发回响应时就在“低”事件量上睡眠。向服务进程发出请求时，客户进程通过系统调用 NtSetHighWaitLowThread()唤醒服务进程并使其立即被调度运行，自身则进入睡眠。相应地服务进程在发回响应时则通过系统调用 NtSetLowWaitHighThread() 唤醒客户进程并使其立即被调度运行，同时自身进入睡眠。内核在处理这两个系统调用时实行特殊的进程调度，即不管优先级高低、总是调度给定事件对上的对端进程运行。这还不够，早期的 WinNT 还专为这两个系统调用采用了两条独立的自陷指令 0x2b 和 0x2c，而不与常规的系统调用合用 0x2e，以进一步提高效率。“Undocumented WindowsNT”一书中对此有详细的叙述，读者可以参阅。

快速 LPC 可以消除因进程调度(先调度别的进程运行)而引起的延迟，又减少了系统调用的次数，从而使总体的效率有所改善，但是却不能从根本上解决问题。一般而言，本来一次系统调用就可以完成的操作，通过服务进程进行却至少要 3 次系统调用和两次进程调度/切换才能完成。这里的 3 次系统调用是：客户进程向服务进程发请求，服务进程在处理以后向客户进程发响应，以及具体的操作本身。两次进程调度/切换则发生于服务进程和客户进程之间。这里说“至少”，是因为往往 3 次系统调用还不够。

所以，即便是有了快速 LPC，早期 WinNT 的性能还是不理想。分析下来，不理想的原因就是本该在内核中实现的功能放在用户空间实现，降低了效率；而对于桌面应用来说这是不能接受的。所以，到 WinNT 4.0 就动了大手术，把原来 csrss 的主要功能、即对图形界面和窗口的操作移到了内核里面。移到内核里面的那一块就是 win32k.sys。而仍留在 csrss 中的则是 csrss.exe 和 csrsrv.dll。所谓 win32k.sys，意思是“内核中的 Win32 子系统”，k 就是指内核。那么，移入内核的这一块有多大呢？下面这几个数字可以给读者一个印像：在我的 WindowsXP 机器上，win32k.sys 的大小是 1772KB，而 csrss.exe 是 4KB，csrsrv.dll 是 29KB。就是说，csrss.exe 和 csrsrv.dll 加在一起还不及 win32k.sys 的零头，只是它的几十分之一。事实上，win32k.sys 为 WinNT 的内核增加了六百多个系统调用(原有的常规系统调用却只有二百多个)。这一改造，对于桌面应用而言，WinNT 就比 Linux 还强、还彻底了，因为毕竟 Linux 的 X11 是作为服务进程在用户空间运行的。而 Wine 再来个服务进程，让文件操作也过服务进程的手，那可真是“雪上加霜”了。

当然，缓冲区复制也是性能下降的重要因素。WinNT 对此也采取了措施，可以通过把用户空间的存储页面临时映射到内核空间来避免复制。

难道我们不应该从 WinNT 的演变中吸取些什么、借鉴些什么吗？