VirtualBox

Shared Folders

sf_write_end -> sf_reg_write_aux -> vboxCallWrite を追ってみる

TODO: ioctl から HostServices に繋げるコードがよくわからない

- vboxCallWrite
  - Additions/common/VBoxGuestLib/VBoxGuestR0LibSharedFolders.c
    - ***VBoxGuestR0Lib*** = ゲストOSの Ring 0 (特権モード) で呼び出すライブラリ
  - ***SHFL_FN_WRITE*** は src/VBox/HostServices/SharedFolders/service.cpp の ***svcCall*** で case 分で分岐に参照される
    - SHFL_FN_WRITE の場合 svcCall は vbfsWrite を呼び出す
      - vbfsWrite はホストOSのシステムコール呼び出しに繋がる。説明は後述
      - ということでホスト側の動作は HostServices/SharedFolders を追えばおk
    - `static DECLCALLBACK(void) svcCall (void *, VBOXHGCMCALLHANDLE callHandle, uint32_t u32ClientID, void *pvClient, uint32_t u32Function, uint32_t cParms, VBOXHGCMSVCPARM paParms[])`
      - なげーよ

```
DECLVBGL(int) vboxCallWrite(PVBSFCLIENT pClient, PVBSFMAP pMap, SHFLHANDLE hFile,
                            uint64_t offset, uint32_t *pcbBuffer, uint8_t *pBuffer, bool fLocked)
{
    int rc = VINF_SUCCESS;

    VBoxSFWrite data;

    // SHFL_FN_ を作る。大事
    //#define VBOX_INIT_CALL(a, b, c)          \
    //    LogFunc(("%s, u32ClientID=%d\n", "SHFL_FN_" # b, \
    //             (c)->ulClientID)); \
    //    (a)->result      = VINF_SUCCESS;     \
    //    (a)->u32ClientID = (c)->ulClientID;  \
    //    (a)->u32Function = SHFL_FN_##b;      \
    //    (a)->cParms      = SHFL_CPARMS_##b
    //
    // SHFL_FN_WRITE (SharedFolder_Function_Write) を呼ぶ
    VBOX_INIT_CALL(&data.callInfo, WRITE, pClient);

    data.root.type                    = VMMDevHGCMParmType_32bit;
    data.root.u.value32               = pMap->root;

    data.handle.type                  = VMMDevHGCMParmType_64bit;
    data.handle.u.value64             = hFile;
    data.offset.type                  = VMMDevHGCMParmType_64bit;
    data.offset.u.value64             = offset;
    data.cb.type                      = VMMDevHGCMParmType_32bit;
    data.cb.u.value32                 = *pcbBuffer;
    data.buffer.type                  = (fLocked) ? VMMDevHGCMParmType_LinAddr_Locked_In : VMMDevHGCMParm
    data.buffer.u.Pointer.size        = *pcbBuffer;
```

```
    data.buffer.u.Pointer.u.linearAddr  = (uintptr_t)pBuffer;

    rc = VbglHGCMCall (pClient->handle, &data.callInfo, sizeof (data));

/*    Log(("VBOXSF: VBoxSF::vboxCallWrite: "
         "VbglHGCMCall rc = %#x, result = %#x\n", rc, data.callInfo.result));
*/
    if (RT_SUCCESS (rc))
    {
        rc = data.callInfo.result;
        *pcbBuffer = data.cb.u.value32;
    }
    return rc;
}
```

- VbglHGCMCall
  - src/VBox/Additions/common/VBoxGuestLib/HGCM.cpp
    - ***HGCM*** = ***Host-Guest Communication Manager***
    - ***Vbgl*** = ***Virtual Box Guest Lib***

```
DECLVBGL(int) VbglHGCMCall (VBGLHGCMHANDLE handle, VBoxGuestHGCMCallInfo *pData, uint32_t cbData)
{
    int rc = VINF_SUCCESS;

    VBGL_HGCM_ASSERTMsg(cbData >= sizeof (VBoxGuestHGCMCallInfo) + pData->cParms * sizeof (HGCMFunctionPara
                        ("cbData = %d, cParms = %d (calculated size %d)\n", cbData, pData->cParms, sizeof (

    // VBOXGUEST_IOCTL_HGCM_CALL は VBoxGuestCommonIOCtl で参照される
    rc = vbglDriverIOCtl (&handle->driver, VBOXGUEST_IOCTL_HGCM_CALL(cbData), pData, cbData);

    return rc;
}
```

- vbglDriverIOCtl
  - src/VBox/Additions/common/VBoxGuestLib/SysHlp.cpp

```
int vbglDriverIOCtl (VBGLDRIVER *pDriver, uint32_t u32Function, void *pvData, uint32_t cbData)
{
    Log(("vbglDriverIOCtl: pDriver: %p, Func: %x, pvData: %p, cbData: %d\n", pDriver, u32Function, pvData,

# ifdef RT_OS_WINDOWS
    // Windows だ !!! 省略しよう
    // ...
# elif defined (RT_OS_OS2)
    // Windows だ !!! 省略しよう
    // ...
# else
    // windows 以外はここ
    return VBoxGuestIDCCall(pDriver->pvOpaque, u32Function, pvData, cbData, NULL);
# endif
}
```

- VBoxGuestIDCCall
  - src/VBox/Additions/common/VBoxGuest/VBoxGuestID-unix.c.h
    - VBoxGuest-linux.c が #include してる
  - ***IDC*** = ***Inter Driver Communication***

```
/**
 * Perform an IDC call.
 *
 * @returns VBox error code.
 * @param   pvSession           Opaque pointer to the session.
 * @param   iCmd                Requested function.
 * @param   pvData              IO data buffer.
 * @param   cbData              Size of the data buffer.
 * @param   pcbDataReturned     Where to store the amount of returned data.
 */
DECLEXPORT(int) VBOXCALL VBoxGuestIDCCall(void *pvSession, unsigned iCmd, void *pvData, size_t cbData, size
{
    PVBOXGUESTSESSION pSession = (PVBOXGUESTSESSION)pvSession;
```

```
    LogFlow(("VBoxGuestIDCCall: %pvSession=%p Cmd=%u pvData=%p cbData=%d\n", pvSession, iCmd, pvData, cbDat
    AssertPtrReturn(pSession, VERR_INVALID_POINTER);
    AssertMsgReturn(pSession->pDevExt == &g_DevExt,
                    ("SC: %p != %p\n", pSession->pDevExt, &g_DevExt), VERR_INVALID_HANDLE);

    return VBoxGuestCommonIOCtl(iCmd, &g_DevExt, pSession, pvData, cbData, pcbDataReturned);
}
```

- VBoxGuestCommonIOCtl
  - src/VBox/Additions/common/VBoxGuest/VBoxGuest.cpp
  - ioctl で ゲストOSからホストOS との通信をする関数
  - iFunction ででかい分岐が連なる
    - VBoxGuestCommonIOCtl_***** の呼び出しに続く
    - VBOXGUEST_IOCTL_HGCM_CALL

```
/**
 * Common IOCtl for user to kernel and kernel to kernel communication.
 * ゲストOSのユーザランド -> ゲストOSのカーネル -> ホストOSのカーネル
 *
 * This function only does the basic validation and then invokes
 * worker functions that takes care of each specific function.
 *
 * @returns VBox status code.
 *
 * @param   iFunction           The requested function.
 * @param   pDevExt             The device extension.
 * @param   pSession            The client session.
 * @param   pvData              The input/output data buffer. Can be NULL depending on the function.
 * @param   cbData              The max size of the data buffer.
 * @param   pcbDataReturned     Where to store the amount of returned data. Can be NULL.
 */
int VBoxGuestCommonIOCtl(unsigned iFunction, PVBOXGUESTDEVEXT pDevExt, PVBOXGUESTSESSION pSession,
                         void *pvData, size_t cbData, size_t *pcbDataReturned)
{

// ...

    else if (VBOXGUEST_IOCTL_STRIP_SIZE(iFunction) == VBOXGUEST_IOCTL_STRIP_SIZE(VBOXGUEST_IOCTL_HGCM_CALL(
    {
        bool fInterruptible = pSession->R0Process != NIL_RTR0PROCESS;
        CHECKRET_MIN_SIZE("HGCM_CALL", sizeof(VBoxGuestHGCMCallInfo));
        rc = VBoxGuestCommonIOCtl_HGCMCall(pDevExt, pSession, (VBoxGuestHGCMCallInfo *)pvData, RT_INDEFINIT
                                           fInterruptible, false /*f32bit*/, false /* fUserData */,
                                           0, cbData, pcbDataReturned);
    }
```

- VBoxGuestCommonIOCtl_HGCMCall
  - src/VBox/Additions/common/VBoxGuest/VBoxGuest.cpp

```
static int VBoxGuestCommonIOCtl_HGCMCall(PVBOXGUESTDEVEXT pDevExt,
                                         PVBOXGUESTSESSION pSession,
                                         VBoxGuestHGCMCallInfo *pInfo,
                                         uint32_t cMillies, bool fInterruptible, bool f32bit, bool fUserDat
                                         size_t cbExtra, size_t cbData, size_t *pcbDataReturned)
{
    const uint32_t  u32ClientId = pInfo->u32ClientID;
    uint32_t        fFlags;
    size_t          cbActual;
    unsigned        i;
    int             rc;

    /*
     * Some more validations.
     */
    if (pInfo->cParms > 4096) /* (Just make sure it doesn't overflow the next check.) */
    {
        LogRel(("VBoxGuestCommonIOCtl: HGCM_CALL: cParm=%RX32 is not sane\n", pInfo->cParms));
        return VERR_INVALID_PARAMETER;
    }

    cbActual = cbExtra + sizeof(*pInfo);
#ifdef RT_ARCH_AMD64
```

```
    if (f32bit)
        cbActual += pInfo->cParms * sizeof(HGCMFunctionParameter32);
    else
#endif
        cbActual += pInfo->cParms * sizeof(HGCMFunctionParameter);
    if (cbData < cbActual)
    {
        LogRel(("VBoxGuestCommonIOCtl: HGCM_CALL: cbData=%#zx (%zu) required size is %#zx (%zu)\n",
               cbData, cbData, cbActual, cbActual));
        return VERR_INVALID_PARAMETER;
    }

    /*
     * Validate the client id.
     */
    RTSpinlockAcquire(pDevExt->SessionSpinlock);
    for (i = 0; i < RT_ELEMENTS(pSession->aHGCMClientIds); i++)
        if (pSession->aHGCMClientIds[i] == u32ClientId)
            break;
    RTSpinlockReleaseNoInts(pDevExt->SessionSpinlock);
    if (RT_UNLIKELY(i >= RT_ELEMENTS(pSession->aHGCMClientIds)))
    {
        static unsigned s_cErrors = 0;
        if (s_cErrors++ > 32)
            LogRel(("VBoxGuestCommonIOCtl: HGCM_CALL: Invalid handle. u32Client=%RX32\n", u32ClientId));
        return VERR_INVALID_HANDLE;
    }

    /*
     * The VbglHGCMCall call will invoke the callback if the HGCM
     * call is performed in an ASYNC fashion. This function can
     * deal with cancelled requests, so we let user more requests
     * be interruptible (should add a flag for this later I guess).
     */
    Log(("VBoxGuestCommonIOCtl: HGCM_CALL: u32Client=%RX32\n", pInfo->u32ClientID));
    fFlags = !fUserData && pSession->R0Process == NIL_RTR0PROCESS ? VBGLR0_HGCMCALL_F_KERNEL : VBGLR0_HGCMC
#ifdef RT_ARCH_AMD64
    if (f32bit)
    {
        if (fInterruptible)
            rc = VbglR0HGCMInternalCall32(pInfo, cbData - cbExtra, fFlags, VBoxGuestHGCMAsyncWaitCallbackIr
        else
            rc = VbglR0HGCMInternalCall32(pInfo, cbData - cbExtra, fFlags, VBoxGuestHGCMAsyncWaitCallback,
    }
    else
#endif
    {
        if (fInterruptible)
            rc = VbglR0HGCMInternalCall(pInfo, cbData - cbExtra, fFlags, VBoxGuestHGCMAsyncWaitCallbackInte
        else
            rc = VbglR0HGCMInternalCall(pInfo, cbData - cbExtra, fFlags, VBoxGuestHGCMAsyncWaitCallback, pI
    }
    if (RT_SUCCESS(rc))
    {
        Log(("VBoxGuestCommonIOCtl: HGCM_CALL: result=%Rrc\n", pInfo->result));
        if (pcbDataReturned)
            *pcbDataReturned = cbActual;
    }
    else
    {
        if (   rc != VERR_INTERRUPTED
            && rc != VERR_TIMEOUT)
        {
            static unsigned s_cErrors = 0;
            if (s_cErrors++ < 32)
                LogRel(("VBoxGuestCommonIOCtl: HGCM_CALL: %s Failed. rc=%Rrc.\n", f32bit ? "32" : "64", rc)
        }
        else
            Log(("VBoxGuestCommonIOCtl: HGCM_CALL: %s Failed. rc=%Rrc.\n", f32bit ? "32" : "64", rc));
    }
    return rc;
}
```

- VbglR0HGCMInternalCall
- src/VBox/Additions/common/VBoxGuestLib/HGCMInternal.cpp

```
DECLR0VBGL(int) VbglR0HGCMInternalCall(VBoxGuestHGCMCallInfo *pCallInfo, uint32_t cbCallInfo, uint32_t fFla
                                       PFNVBGLHGCMCALLBACK pfnAsyncCallback, void *pvAsyncData, uint32_t u3
{
    bool                    fIsUser = (fFlags & VBGLR0_HGCMCALL_F_MODE_MASK) == VBGLR0_HGCMCALL_F_USER;
    struct VbglR0ParmInfo   ParmInfo;
    size_t                  cbExtra;
    int                     rc;

    /*
     * Basic validation.
     */
    AssertMsgReturn(   !pCallInfo
                    || !pfnAsyncCallback
                    || pCallInfo->cParms > VBOX_HGCM_MAX_PARMS
                    || !(fFlags & ~VBGLR0_HGCMCALL_F_MODE_MASK),
                    ("pCallInfo=%p pfnAsyncCallback=%p fFlags=%#x\n", pCallInfo, pfnAsyncCallback, fFlags),
                    VERR_INVALID_PARAMETER);
    AssertReturn(   cbCallInfo >= sizeof(VBoxGuestHGCMCallInfo)
                 || cbCallInfo >= pCallInfo->cParms * sizeof(HGCMFunctionParameter),
                 VERR_INVALID_PARAMETER);

    Log(("GstHGCMCall: u32ClientID=%#x u32Function=%u cParms=%u cbCallInfo=%#x fFlags=%#x\n",
         pCallInfo->u32ClientID, pCallInfo->u32ClientID, pCallInfo->u32Function, pCallInfo->cParms, cbCall

    /*
     * Validate, lock and buffer the parameters for the call.
     * This will calculate the amount of extra space for physical page list.
     */
    rc = vbglR0HGCMInternalPreprocessCall(pCallInfo, cbCallInfo, fIsUser, &ParmInfo, &cbExtra);
    if (RT_SUCCESS(rc))
    {
        /*
         * Allocate the request buffer and recreate the call request.
         */
        VMMDevHGCMCall *pHGCMCall;
        rc = VbglGRAlloc((VMMDevRequestHeader **)&pHGCMCall,
                         sizeof(VMMDevHGCMCall) + pCallInfo->cParms * sizeof(HGCMFunctionParameter) + cbExt
                         VMMDevReq_HGCMCall);
        if (RT_SUCCESS(rc))
        {
            bool fLeakIt;
            vbglR0HGCMInternalInitCall(pHGCMCall, pCallInfo, cbCallInfo, fIsUser, &ParmInfo);

            /*
             * Perform the call.
             */
            rc = vbglR0HGCMInternalDoCall(pHGCMCall, pfnAsyncCallback, pvAsyncData, u32AsyncData, &fLeakIt)
            if (RT_SUCCESS(rc))
            {
                /*
                 * Copy back the result (parameters and buffers that changed).
                 */
                rc = vbglR0HGCMInternalCopyBackResult(pCallInfo, pHGCMCall, &ParmInfo, fIsUser, rc);
            }
            else
            {
                if (   rc != VERR_INTERRUPTED
                    && rc != VERR_TIMEOUT)
                {
                    static unsigned s_cErrors = 0;
                    if (s_cErrors++ < 32)
                        LogRel(("VbglR0HGCMInternalCall: vbglR0HGCMInternalDoCall failed. rc=%Rrc\n", rc));
                }
            }

            if (!fLeakIt)
                VbglGRFree(&pHGCMCall->header.header);
        }
    }
    else
        LogRel(("VbglR0HGCMInternalCall: vbglR0HGCMInternalPreprocessCall failed. rc=%Rrc\n", rc));

    /*
     * Release locks and free bounce buffers.
     */
    if (ParmInfo.cLockBufs)
        while (ParmInfo.cLockBufs-- > 0)
```

```
        {
            RTR0MemObjFree(ParmInfo.aLockBufs[ParmInfo.cLockBufs].hObj, false /*fFreeMappings*/);
#ifdef USE_BOUNCE_BUFFERS
            RTMemTmpFree(ParmInfo.aLockBufs[ParmInfo.cLockBufs].pvSmallBuf);
#endif
        }

    return rc;
}
```

- vbglR0HGCMInternalDoCall
  - src/VBox/Additions/common/VBoxGuestLib/HGCMInternal.cpp

```
/**
 * Performs the call and completion wait.
 *
 * @returns VBox status code of this operation, not necessarily the call.
 *
 * @param   pHGCMCall           The HGCM call info.
 * @param   pfnAsyncCallback    The async callback that will wait for the call
 *                              to complete.
 * @param   pvAsyncData         Argument for the callback.
 * @param   u32AsyncData        Argument for the callback.
 * @param   pfLeakIt            Where to return the leak it / free it,
 *                              indicator. Cancellation fun.
 */
static int vbglR0HGCMInternalDoCall(VMMDevHGCMCall *pHGCMCall, PFNVBGLHGCMCALLBACK pfnAsyncCallback,
                                    void *pvAsyncData, uint32_t u32AsyncData, bool *pfLeakIt)
{
    int rc;

    Log(("calling VbglGRPerform\n"));
    rc = VbglGRPerform(&pHGCMCall->header.header);
    Log(("VbglGRPerform rc = %Rrc (header rc=%d)\n", rc, pHGCMCall->header.result));

    /*
     * If the call failed, but as a result of the request itself, then pretend
     * success. Upper layers will interpret the result code in the packet.
     */
    if (    RT_FAILURE(rc)
        &&  rc == pHGCMCall->header.result)
    {
        Assert(pHGCMCall->header.fu32Flags & VBOX_HGCM_REQ_DONE);
        rc = VINF_SUCCESS;
    }

    /*
     * Check if host decides to process the request asynchronously,
     * if so, we wait for it to complete using the caller supplied callback.
     */
    *pfLeakIt = false;
    if (rc == VINF_HGCM_ASYNC_EXECUTE)
    {
        Log(("Processing HGCM call asynchronously\n"));
        rc = pfnAsyncCallback(&pHGCMCall->header, pvAsyncData, u32AsyncData);
        if (pHGCMCall->header.fu32Flags & VBOX_HGCM_REQ_DONE)
        {
            Assert(!(pHGCMCall->header.fu32Flags & VBOX_HGCM_REQ_CANCELLED));
            rc = VINF_SUCCESS;
        }
        else
        {
            /*
             * The request didn't complete in time or the call was interrupted,
             * the RC from the callback indicates which. Try cancel the request.
             *
             * This is a bit messy because we're racing request completion. Sorry.
             */
            /** @todo It would be nice if we could use the waiter callback to do further
             *  waiting in case of a completion race. If it wasn't for WINNT having its own
             *  version of all that stuff, I would've done it already. */
            VMMDevHGCMCancel2 *pCancelReq;
            int rc2 = VbglGRAlloc((VMMDevRequestHeader **)&pCancelReq, sizeof(*pCancelReq), VMMDevReq_HGCM(
            if (RT_SUCCESS(rc2))
            {
                pCancelReq->physReqToCancel = VbglPhysHeapGetPhysAddr(pHGCMCall);
```

```
                rc2 = VbglGRPerform(&pCancelReq->header);
                VbglGRFree(&pCancelReq->header);
            }
#if 1 /** @todo ADDVER: Remove this on next minor version change. */
            if (rc2 == VERR_NOT_IMPLEMENTED)
            {
                /* host is too old, or we're out of heap. */
                pHGCMCall->header.fu32Flags |= VBOX_HGCM_REQ_CANCELLED;
                pHGCMCall->header.header.requestType = VMMDevReq_HGCMCancel;
                rc2 = VbglGRPerform(&pHGCMCall->header.header);
                if (rc2 == VERR_INVALID_PARAMETER)
                    rc2 = VERR_NOT_FOUND;
                else if (RT_SUCCESS(rc))
                    RTThreadSleep(1);
            }
#endif
            if (RT_SUCCESS(rc)) rc = VERR_INTERRUPTED; /** @todo weed this out from the WINNT VBoxGuest coc
            if (RT_SUCCESS(rc2))
            {
                Log(("vbglR0HGCMInternalDoCall: successfully cancelled\n"));
                pHGCMCall->header.fu32Flags |= VBOX_HGCM_REQ_CANCELLED;
            }
            else
            {
                /*
                 * Wait for a bit while the host (hopefully) completes it.
                 */
                uint64_t u64Start       = RTTimeSystemMilliTS();
                uint32_t cMilliesToWait = rc2 == VERR_NOT_FOUND || rc2 == VERR_SEM_DESTROYED ? 500 : 2000;
                uint64_t cElapsed       = 0;
                if (rc2 != VERR_NOT_FOUND)
                {
                    static unsigned s_cErrors = 0;
                    if (s_cErrors++ < 32)
                        LogRel(("vbglR0HGCMInternalDoCall: Failed to cancel the HGCM call on %Rrc: rc2=%Rrc
                }
                else
                    Log(("vbglR0HGCMInternalDoCall: Cancel race rc=%Rrc rc2=%Rrc\n", rc, rc2));

                do
                {
                    ASMCompilerBarrier();       /* paranoia */
                    if (pHGCMCall->header.fu32Flags & VBOX_HGCM_REQ_DONE)
                        break;
                    RTThreadSleep(1);
                    cElapsed = RTTimeSystemMilliTS() - u64Start;
                } while (cElapsed < cMilliesToWait);

                ASMCompilerBarrier();           /* paranoia^2 */
                if (pHGCMCall->header.fu32Flags & VBOX_HGCM_REQ_DONE)
                    rc = VINF_SUCCESS;
                else
                {
                    LogRel(("vbglR0HGCMInternalDoCall: Leaking %u bytes. Pending call to %u with %u parms.
                            pHGCMCall->header.header.size, pHGCMCall->u32Function, pHGCMCall->cParms, rc2))
                    *pfLeakIt = true;
                }
                Log(("vbglR0HGCMInternalDoCall: Cancel race ended with rc=%Rrc (rc2=%Rrc) after %llu ms\n",
            }
        }
    }

    Log(("GstHGCMCall: rc=%Rrc result=%Rrc fu32Flags=%#x fLeakIt=%d\n",
         rc, pHGCMCall->header.result, pHGCMCall->header.fu32Flags, *pfLeakIt));
    return rc;
}
```

- VbglGRPerform
- src/VBox/Additions/common/VBoxGuestLib/GenericRequest.cpp

```
DECLVBGL(int) VbglGRPerform (VMMDevRequestHeader *pReq)
{
    RTCCPHYS physaddr;

    // これなんだ
    int rc = vbglR0Enter ();
```

```
    if (RT_FAILURE(rc))
        return rc;

    if (!pReq)
        return VERR_INVALID_PARAMETER;

    physaddr = VbglPhysHeapGetPhysAddr (pReq);
    if (  !physaddr
       || (physaddr >> 32) != 0) /* Port IO is 32 bit. */
    {
        rc = VERR_VBGL_INVALID_ADDR;
    }
    else
    {
        ASMOutU32(g_vbgldata.portVMMDev + VMMDEV_PORT_OFF_REQUEST, (uint32_t)physaddr);
        /* Make the compiler aware that the host has changed memory. */
        ASMCompilerBarrier();
        rc = pReq->rc;
    }
    return rc;
}
```

- ASMOutU32
  - include/iprt/asm-amd64-x86.h
  - ***ASM** = **AMD64 and x86 Specific Assembly Functions***
  - outl で I/Oポートにデータを書く
  - VMMDevRequestHeader の開始アドレスを書いて、ホストOSが読む?

```
/**
 * Writes a 32-bit unsigned integer to an I/O port, ordered.
 *
 * @param   Port    I/O port to write to.
 * @param   u32     32-bit integer to write.
 */
#if RT_INLINE_ASM_EXTERNAL && !RT_INLINE_ASM_USES_INTRIN
DECLASM(void) ASMOutU32(RTIOPORT Port, uint32_t u32);
#else
DECLINLINE(void) ASMOutU32(RTIOPORT Port, uint32_t u32)
{
# if RT_INLINE_ASM_GNU_STYLE
    __asm__ __volatile__("outl %1, %w0\n\t"
                         :: "Nd" (Port),
                            "a" (u32));

# elif RT_INLINE_ASM_USES_INTRIN
    __outdword(Port, u32);

# else
    __asm
    {
        mov     dx, [Port]
        mov     eax, [u32]
        out     dx, eax
    }
# endif
}
#endif
```

- VbglPhysHeapGetPhysAddr
  - src/VBox/Additions/common/VBoxGuestLib/PhysHeap.cpp
  - 物理ヒープ?の物理アドレスを出す?

```
DECLVBGL(uint32_t) VbglPhysHeapGetPhysAddr (void *p)
{
    uint32_t physAddr = 0;
    VBGLPHYSHEAPBLOCK *pBlock = vbglPhysHeapData2Block (p);

    if (pBlock)
    {
        VBGL_PH_ASSERTMsg((pBlock->fu32Flags & VBGL_PH_BF_ALLOCATED) != 0,
                          ("pBlock = %p, pBlock->fu32Flags = %08X\n", pBlock, pBlock->fu32Flags));
```

```
        if (pBlock->fu32Flags & VBGL_PH_BF_ALLOCATED)
            physAddr = pBlock->pChunk->physAddr + (uint32_t)((uintptr_t)p - (uintptr_t)pBlock->pChunk);
    }

    return physAddr;
}
```

HostServices

ホストOS側のVirtualBoxVM プロセスで実行されるユーザランドのコード。 Not Kernel

- src/VBox/HostServices/SharedFolders/vbsf.h に ホストOSが svcCall で呼び出すAPIが定義されている
- ホストOSの VirtualBoxVMプロセスが呼び出すシステムコールの抽象化ラッパー
  - ホストOSのプラットフォームによって呼び出すべきシステムコールが違うので抽象化をかます必要がある

```
// ...

int vbsfCreate (SHFLCLIENTDATA *pClient, SHFLROOT root, SHFLSTRING *pPath, uint32_t cbPath, SHFLCREATEPARMS

int vbsfClose (SHFLCLIENTDATA *pClient, SHFLROOT root, SHFLHANDLE Handle);

int vbsfRead(SHFLCLIENTDATA *pClient, SHFLROOT root, SHFLHANDLE Handle, uint64_t offset, uint32_t *pcbBuffe
int vbsfWrite(SHFLCLIENTDATA *pClient, SHFLROOT root, SHFLHANDLE Handle, uint64_t offset, uint32_t *pcbBuf
int vbsfLock(SHFLCLIENTDATA *pClient, SHFLROOT root, SHFLHANDLE Handle, uint64_t offset, uint64_t length, u
int vbsfUnlock(SHFLCLIENTDATA *pClient, SHFLROOT root, SHFLHANDLE Handle, uint64_t offset, uint64_t length,
int vbsfRemove(SHFLCLIENTDATA *pClient, SHFLROOT root, SHFLSTRING *pPath, uint32_t cbPath, uint32_t flags);
int vbsfRename(SHFLCLIENTDATA *pClient, SHFLROOT root, SHFLSTRING *pSrc, SHFLSTRING *pDest, uint32_t flags)
int vbsfDirList(SHFLCLIENTDATA *pClient, SHFLROOT root, SHFLHANDLE Handle, SHFLSTRING *pPath, uint32_t flag
int vbsfFileInfo(SHFLCLIENTDATA *pClient, SHFLROOT root, SHFLHANDLE Handle, uint32_t flags, uint32_t *pcbBu
int vbsfQueryFSInfo(SHFLCLIENTDATA *pClient, SHFLROOT root, SHFLHANDLE Handle, uint32_t flags, uint32_t *pc
int vbsfSetFSInfo(SHFLCLIENTDATA *pClient, SHFLROOT root, SHFLHANDLE Handle, uint32_t flags, uint32_t *pcbE
int vbsfFlush(SHFLCLIENTDATA *pClient, SHFLROOT root, SHFLHANDLE Handle);
int vbsfDisconnect(SHFLCLIENTDATA *pClient);
int vbsfQueryFileInfo(SHFLCLIENTDATA *pClient, SHFLROOT root, SHFLHANDLE Handle, uint32_t flags, uint32_t *
int vbsfReadLink(SHFLCLIENTDATA *pClient, SHFLROOT root, SHFLSTRING *pPath, uint32_t cbPath, uint8_t *pBuf
int vbsfSymlink(SHFLCLIENTDATA *pClient, SHFLROOT root, SHFLSTRING *pNewPath, SHFLSTRING *pOldPath, SHFLFSO

#endif /* __VBSF__H */
```

vbsfCreate

ホストOSがUNIX系OSなら、最終的に open(2) を呼び出すはず

- vbfsOpenFile か vbsfOpenDir に続く
  - vbsfOpenFile から RTFileOpen を呼び出す
    - ***RT*** = RunTime ?

```
// include/iprt/mangling.h
# define RTFileOpen                                     RT_MANGLER(RTFileOpen)

// include/VBox/VBoxGuestMangling.h
#define RT_MANGLER(symbol)   VBoxGuest_##symbol

// include/VBox/SUPDrvMangling.h
#define RT_MANGLER(symbol)   VBoxHost_##symbol

// 最終的に VBoxGuest_RTFileOpen, VBoxHOST_RTFileOpen のシンボルに変換される?
```

- src/VBox/Runtime 以下にプラットフォームごとのディレクトリが見つかる
  - src/VBox/Runtime/r3/posix/fileio-posix.cpp に POSIX な RTFileOpen実装がある
  - (ホストOSのプロセスとして) open(2) を呼び出している

```
RTR3DECL(int) RTFileOpen(PRTFILE pFile, const char *pszFilename, uint64_t fOpen)
{
    // ....

    /*
     * Open/create the file.
```

```
         */
        char const *pszNativeFilename;
        rc = rtPathToNative(&pszNativeFilename, pszFilename, NULL);
        if (RT_FAILURE(rc))
            return (rc);

        int fh = open(pszNativeFilename, fOpenMode, fMode);
        int iErr = errno;
```

ということで RT_MANGLER を見つけたら src/VBox/Runtime/ 以下の実装を追えばおk

IPRT って何?

```
IPRT, a portable runtime library which abstracts file access, threading, string manipulation, etc.
Whenever VirtualBox accesses host operating features, it does so through this library for cross-platform pc

http://www.virtualbox.org/manual/ch10.html
```