ファイルシステムへのエントリーポイント

前回の課題のトレースで、

FS へ WRITE のパラメータで

アセンブラで定義されていた /usr/src/lib/i386/rts のファイル、_sendrec.s から

```
__sendrec:
        push    ebp
        mov     ebp, esp
        push    ebx
        mov     eax, SRCDEST(ebp)       ! eax = dest-src
        mov     ebx, MESSAGE(ebp)       ! ebx = message pointer
        mov     ecx, BOTH               ! _sendrec(srcdest, ptr)
       _int     SYSVEC                  ! trap to the kernel
        pop     ebx
        pop     ebp
        ret
 50
```

int SYSVEC

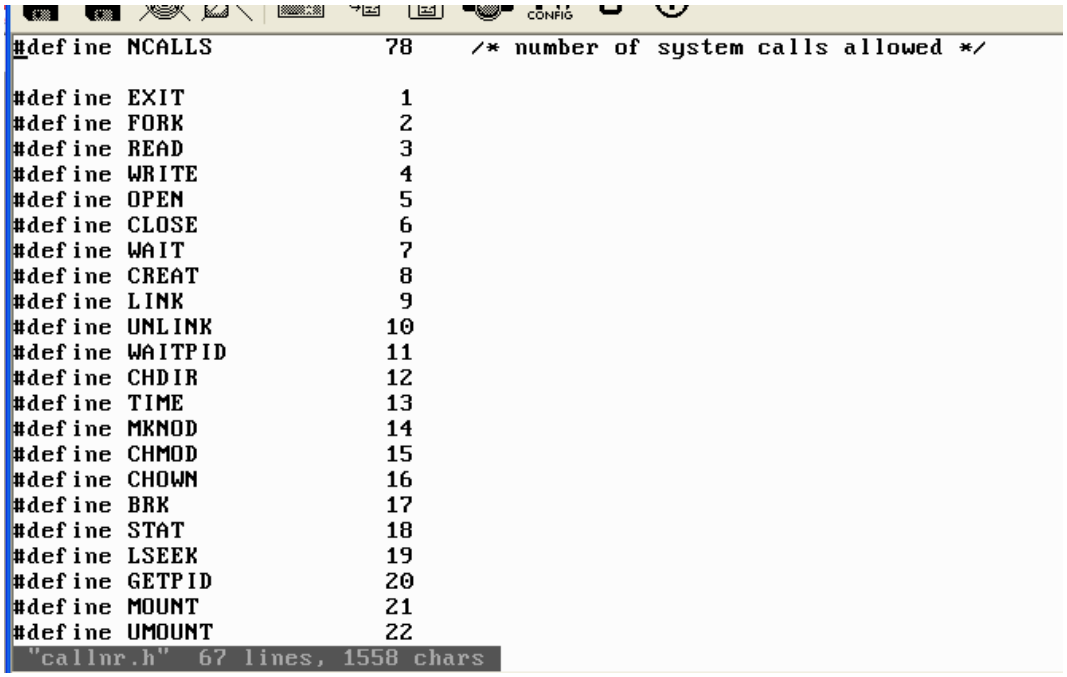
で「割込み」が発行されているところまでを辿りました。

この int SYSVEC は、割り込み番号「SYSVEC」で interrupt を発行するというアセンブラ命令で、SYSVEC の値は８行目で

```
.sect .text; .sect .rom; .sect .data; .sect .bss
.define __send, __receive, __sendrec

! See ../h/com.h for C definitions
SEND = 1
RECEIVE = 2
BOTH = 3
SYSVEC = 33

SRCDEST = 8
MESSAGE = 12

!*===========================================================================*
!                           _send and _receive                              *
!*===========================================================================*
! _send(), _receive(), _sendrec() all save ebp, but destroy eax and ecx.
.define __send, __receive, __sendrec
.sect .text
__send:
        push    ebp
        mov     ebp, esp
        push    ebx
        mov     eax, SRCDEST(ebp)       ! eax = dest-src
        mov     ebx, MESSAGE(ebp)       ! ebx = message pointer
 8
```

33 に定義されていました。

WRITE は、/usr/include/callnr.h というファイルを開くと



```
#define NCALLS              78    /* number of system calls allowed */

#define EXIT                 1
#define FORK                 2
#define READ                 3
#define WRITE                4
#define OPEN                 5
#define CLOSE                6
#define WAIT                 7
#define CREAT                8
#define LINK                 9
#define UNLINK              10
#define WAITPID             11
#define CHDIR               12
#define TIME                13
#define MKNOD               14
#define CHMOD               15
#define CHOWN               16
#define BRK                 17
#define STAT                18
#define LSEEK               19
#define GETPID              20
#define MOUNT               21
#define UMOUNT              22
 "callnr.h"  67 lines, 1558 chars
```

として、４として定義されています。

つまり、ファイルシステムに対して、WRITE という「OS の機能の番号」を渡して、SYSVEC の「割込み番号」（この SYSVEC が、カーネル呼び出し割込みの番号）として、割り込みを発行した、つまり、カーネルを呼び出したことになります。

カーネルでは、プログラムの起動時に/usr/src/kernel/mpx386.s というファイルが以下の割込みハンドラの定義を行っています。

```
! is loosely equivalent to a prototype in C code -- it makes it possible to
! link to an entity declared in the assembly code but does not create
! the entity.

.define _idle_task
.define _restart
.define save

.define _divide_error
.define _single_step_exception
.define _nmi
.define _breakpoint_exception
.define _overflow
.define _bounds_check
.define _inval_opcode
.define _copr_not_available
.define _double_fault
.define _copr_seg_overrun
.define _inval_tss
.define _segment_not_present
.define _stack_exception
.define _general_protection
.define _page_fault
.define _copr_error
```

ここでは、FS（ファイルシステム）も、MM（メモリ管理）も、全て「カーネル呼び出し」

として一括して扱います。

SYSVEC の割込み番号は、/usr/src/kernel/protect.c のファイルで辿ることができます。

８０行目から定義が始まる prot_init()で、以下のような gate_table の定義があります。

```
  ,
  gate_table[] = {
        divide_error, DIVIDE_VECTOR, INTR_PRIVILEGE,
        single_step_exception, DEBUG_VECTOR, INTR_PRIVILEGE,
        nmi, NMI_VECTOR, INTR_PRIVILEGE,
        breakpoint_exception, BREAKPOINT_VECTOR, USER_PRIVILEGE,
        overflow, OVERFLOW_VECTOR, USER_PRIVILEGE,
        bounds_check, BOUNDS_VECTOR, INTR_PRIVILEGE,
        inval_opcode, INVAL_OP_VECTOR, INTR_PRIVILEGE,
        copr_not_available, COPROC_NOT_VECTOR, INTR_PRIVILEGE,
        double_fault, DOUBLE_FAULT_VECTOR, INTR_PRIVILEGE,
        copr_seg_overrun, COPROC_SEG_VECTOR, INTR_PRIVILEGE,
        inval_tss, INVAL_TSS_VECTOR, INTR_PRIVILEGE,
        segment_not_present, SEG_NOT_VECTOR, INTR_PRIVILEGE,
        stack_exception, STACK_FAULT_VECTOR, INTR_PRIVILEGE,
        general_protection, PROTECTION_VECTOR, INTR_PRIVILEGE,
#if _WORD_SIZE == 4
        page_fault, PAGE_FAULT_VECTOR, INTR_PRIVILEGE,
        copr_error, COPROC_ERR_VECTOR, INTR_PRIVILEGE,
#endif
        { hwint00, VECTOR( 0), INTR_PRIVILEGE },
        { hwint01, VECTOR( 1), INTR_PRIVILEGE },
        { hwint02, VECTOR( 2), INTR_PRIVILEGE },
```

このファイルの 137 行目で、

```
        { hwint13, VECTOR(13), INTR_PRIVILEGE },
        { hwint14, VECTOR(14), INTR_PRIVILEGE },
        { hwint15, VECTOR(15), INTR_PRIVILEGE },
#if _WORD_SIZE == 2
        { p_s_call, SYS_VECTOR, USER_PRIVILEGE },       /* 286 system call */
#else
        { s_call, SYS386_VECTOR, USER_PRIVILEGE },      /* 386 system call */
#endif
        { level0_call, LEVEL0_VECTOR, TASK_PRIVILEGE },
  };

  /* Build gdt and idt pointers in GDT where the BIOS expects them. */
  dtp= (struct desctableptr_s *) &gdt[GDT_INDEX];
  * (u16_t *) dtp->limit = (sizeof gdt) - 1;
  * (u32_t *) dtp->base = vir2phys(gdt);

  dtp= (struct desctableptr_s *) &gdt[IDT_INDEX];
  * (u16_t *) dtp->limit = (sizeof idt) - 1;
```

と定義されていて、SYS386_VECTOR（３３）という割込み番号では、s_call をコールするように、割込み番号に対する割込みサービスプログラムを割り振っています。

ここで「割込みベクトル」と呼ばれる、ハードウェア割り込みなどの処理プログラムのアドレスが登録されています。
SYS386_VECTOR という「定義名」は、const.h で定義されています。つまり s_call()がカーネルへの入り口だ、ということになります。

33 番の場合には_s_call というアセンブラプログラムに処理が渡されます。_s_call は、/usr/src/kernel/mpx386.s の 362 行目から定義されていて、

```
!*                              _s_call                                 *
!*======================================================================*
        .align  16
_s_call:
_p_s_call:
        cld                     ! set direction flag to a known value
        sub     esp, 6*4        ! skip RETADR, eax, ecx, edx, ebx, est
        push    ebp             ! stack already points into proc table
        push    esi
        push    edi
    o16 push    ds
    o16 push    es
    o16 push    fs
    o16 push    gs
        mov     dx, ss
        mov     ds, dx
        mov     es, dx
        incb    (_k_reenter)
        mov     esi, esp        ! assumes P_STACKBASE == 0
        mov     esp, k_stktop
        xor     ebp, ebp        ! for stacktrace
                                ! end of inline save
        sti                     ! allow SWITCHER to be interrupted

        push    ebx             ! pointer to user message
        push    eax             ! src/dest
        push    ecx             ! SEND/RECEIVE/BOTH
        call    _sys_call       ! sys_call(function, src_dest, m_ptr)
                                ! caller is now explicitly in proc_ptr
        mov     AXREG(esi), eax ! sys_call MUST PRESERVE si
        cli                     ! disable interrupts

! Fall into code to restart proc/task running.

!*======================================================================*
!*                              restart                                 *
!*======================================================================*
_restart:

! Flush any held-up interrupts.
! This reenables interrupts, so the current interrupt handler may reenter.
! This does not matter, because the current handler is about to exit and no
```

と書かれています。ここで、sys_call を呼び出していることに注意してください。
_k_reenter は、このアセンブラプログラムで、カーネルの再帰的呼び出しをコントロールするために使われています。

ここから 386 で、sys_call という C 言語のプログラムに制御を渡します。この sys_call は /usr/src/kernel/proc.c の 116 行目で定義されています。

どのようにして、「割り込みベクトルが初期化されるか」は、順調に行けば、第 8 回の「ブートローディングと OS の初期化」のところで見てみます。

```
}

/*===========================================================================*
 *                                sys_call                                   *
 *===========================================================================*/
PUBLIC int sys_call(function, src_dest, m_ptr)
int function;                   /* SEND, RECEIVE, or BOTH */
int src_dest;                   /* source to receive from or dest to send to */
message *m_ptr;                 /* pointer to message */
{
/* The only system calls that exist in MINIX are sending and receiving
 * messages.  These are done by trapping to the kernel with an INT instruction.
 * The trap is caught and sys_call() is called to send or receive a message
 * (or both). The caller is always given by proc_ptr.
 */

  register struct proc *rp;
  int n;

  /* Check for bad system call parameters. */
  if (!isoksrc_dest(src_dest)) return(E_BAD_DEST);
  rp = proc_ptr;

  if (isuserp(rp) && function != BOTH) return(E_NO_PERM);
```

これが入り口だよ、というコメントが書かれています。ここから、mini_send という関数が呼び出されて、「プロセス間」の通信で、FS や MM にメッセージが渡されます。
実質的に、外部からの OS のシステム割り込みは、大半が FS や MM に渡されます。

FS（ファイルシステム）では、`/usr/src/fs/main.c` の関数 main()（39 行目に定義されている）では、５０行目の `while(TRUE)`の構造（無限ループ）で、ポーリングをかけていて、上記の mini_send などでスリープ状態から起こされると、仕事を取ってきて、call_vec というプログラムの「間接参照」アドレステーブルから実質的な処理プログラムを呼び出します。

```
  fs_init();

  /* This is the main loop that gets work, processes it, and sends replies. */
  while (TRUE) {
        get_work();             /* sets who and fs_call */

        fp = &fproc[who];       /* pointer to proc table struct */
        super_user = (fp->fp_effuid == SU_UID ? TRUE : FALSE);   /* su? */

        /* Call the internal function that does the work. */
        if (fs_call < 0 || fs_call >= NCALLS)
                error = ENOSYS;
        else
                error = (*call_vec[fs_call])();

        /* Copy the results back to the user and send reply. */
        if (error != SUSPEND) reply(who, error);
        if (rdahed_inode != NIL_INODE) read_ahead(); /* do block read ahead */
  }
```

ここで、「入り口」から、それぞれの必要な処理に分岐します。

この call_vec　は、「割込みベクトル」を処理した後の、さらにカーネルの機能番号ごとの処理プログラムを呼び出しています。/usr/src/kernel/table.c では、call_vec が定義されています。

```
PUBLIC _PROTOTYPE (int (*call_vec[]), (void) ) = {
        no_sys,         /*  0 = unused  */
        do_exit,        /*  1 = exit    */
        do_fork,        /*  2 = fork    */
        do_read,        /*  3 = read    */
       _do_write,       /*  4 = write   */
        do_open,        /*  5 = open    */
        do_close,       /*  6 = close   */
        no_sys,         /*  7 = wait    */
        do_creat,       /*  8 = creat   */
        do_link,        /*  9 = link    */
        do_unlink,      /* 10 = unlink  */
        no_sys,         /* 11 = waitpid */
        do_chdir,       /* 12 = chdir   */
        do_time,        /* 13 = time    */
        do_mknod,       /* 14 = mknod   */
        do_chmod,       /* 15 = chmod   */
        do_chown,       /* 16 = chown   */
        no_sys,         /* 17 = break   */
        do_stat,        /* 18 = stat    */
        do_lseek,       /* 19 = lseek   */
 23
```

ここで、２３行目を見ると、4 = write と書かれていますが、この call_vec の添え字と、前回トレースで調べた /usr/include/minix/vallnr.h の中身とを比較してみてください。

横に並べてみると、このようになります。つまり、callnr.h で調べた番号は、この call_vec の間接参照アドレスの添え字になっているのです。

```
#define NCALLS          78
                                PUBLIC _PROTOTYPE (int (*call_vec[]), (void) ) =
#define EXIT             1              no_sys,         /*  0 = unused  */
#define FORK             2              do_exit,        /*  1 = exit    */
#define READ             3              do_fork,        /*  2 = fork    */
#define WRITE            4              do_read,        /*  3 = read    */
#define OPEN             5             _do_write,       /*  4 = write   */
#define CLOSE            6              do_open,        /*  5 = open    */
#define WAIT             7              do_close,       /*  6 = close   */
#define CREAT            8              no_sys,         /*  7 = wait    */
#define LINK             9              do_creat,       /*  8 = creat   */
#define UNLINK          10              do_link,        /*  9 = link    */
#define WAITPID         11              do_unlink,      /* 10 = unlink  */
#define CHDIR           12              no_sys,         /* 11 = waitpid */
#define TIME            13              do_chdir,       /* 12 = chdir   */
#define MKNOD           14              do_time,        /* 13 = time    */
#define CHMOD           15              do_mknod,       /* 14 = mknod   */
#define CHOWN           16              do_chmod,       /* 15 = chmod   */
#define BRK             17              do_chown,       /* 16 = chown   */
#define STAT            18              no_sys,         /* 17 = break   */
#define LSEEK           19              do_stat,        /* 18 = stat    */
#define GETPID          20              do_lseek,       /* 19 = lseek   */
#define MOUNT           21       23
#define UMOUNT          22
 "/usr/include/minix/callnr.h"  67 lines, 1558 chars
```

MINIX のコマンドレファレンスを調べた際に、様々なコマンドがあることに気づいたでしょうか？例えば、chmod なども、ここで定義されています。
タイマ（TIME）割り込みは何番でしょうか？

これが、「割込み発行」から、カーネル（または、FS や MM）の中に顔を出すまでの、ワープトンネルの中の構造です。

割り込みを受け付けた際に、戻り先の情報やレジスタ情報を保存する、などの記述も、この通り道の中で行っています。いわゆる「割込み処理ベクトル」を登録したり、ポーリングしている複数のプログラムを起動したりする部分は、プログラムの初期化で行いますので、本日の説明は行いません。

つまり、「どこに処理を引き継いでいるか」を知るためには、出口にある、FS や MM の table.c で、「処理している関数の名前」を見ればいいことになります。

実は、この先、もう一つこうした「間接参照」が辿りにくい形で記述されている部分があります。
DEVICE　IO ファイルのトレースです。
ここでは、複数のデバイス（周辺機器）を抽象化（仮想化）しているために、OPEN や CLOSE、READ や WRITE などの関数も、テーブル参照で分岐するようになっています。

このため、本日の課題のトレースでは、トンネルの入り口と、出口となる「関数名」までは、プリント教材に記しておきました。