Spring System Calls

Douglas Niehaus

June 8, 1992

1 Introduction

As a predictable real-time system, Spring must provide system calls whose execution behavior are predictable. When an application process makes a system call, a specific set of steps are required to prepare the information required by the system, transfer control to the system, perform the service, and transfer control back to the application process. Each of these steps, with the exception of preparing the information and actually servicing the call, are the same for all system calls, and are implemented using methods close to those used in conventional systems.

This document describes each step in servicing a system call, and explains how to implement a new call. It also describes the relatively minor restrictions, on the portions of the system common to all calls required to ensure predictable behavior. Section 2 describes how system calls work, and the restrictions required to make their behavior predictable. Section 3 concludes the paper by summarizing the steps required to implement a new system call.

2 System Call Execution

System calls exist to provide a way for application programs to request services from the system, and their operation can be divided into several distinct phases. The first phase begins at the interface routine provided by the system service, which is responsible for preparing the information required by the system to service the request, and ends with the transition to system mode. The second phase begins with the execution of the handler common to all system calls, and ends when control is transferred to the routine which provides the requested service. Execution of the service specific handler is the third phase, while the fourth is the transition back to user mode. The rest of this section will discuss each phase in some detail, pointing out several issues which impact system call implementation and performance.

2.1 System Call Interface

The interface between application code and the system compromises among several issues. On the system side, it will generally be easier and more efficient to provide simple low level abilities, which generally means providing an interface that is less useful than the application programmer would like. From the application side, the programmer would like a powerful

```
addr_t
shm_attach(char *name, int seg_size, int *first)
{
        addr_t ret_val;
        int str_len;

        str_len = strlen(name);
        ret_val = sys_shm_attach(name, str_len, seg_size, first);
        return(ret_val);
}
```

Figure 1: C System Call Routine

interface providing a wide range of sophisticated services, but this often implies a less efficient, and more difficult, implementation on the system side. The situation is further complicated by the fact that the user interface requirements usually change far faster, and over a longer period of time, than that during which the operating system is modified.

The most common solution, which generally satisfies both sides, is to create a layer of relatively sophisticated library routines implementing an interface close to that desired by the application developer, which use the relatively primitive system calls actually provided by the operating system. These routines are included in the system's C library, and provide the interface to the system actually *used* by application programmers. This approach is so successful in the case of the UNIX system, that many experienced users name the higher level interface routines when asked to name some system calls. The same approach has been taken when implementing the system call interface for Spring.

In the interest of predictability and reasonable efficiency, we limit the arguments of Spring system calls to integers and pointers. This means that, on the current target architecture, all system call parameters are the same size. We also require that the pointers passed as parameters refer to objects of known size. In the case of pointers to structures, this is information is available at compile time as an attribute of the structure definition. However, in the case of strings or other variable sized buffers buffers, the size can only be determined at run-time, and is passed as an additional parameter to the call. The reason for this is that knowing the size of the object to which a pointer refers permits copying it into system space much more efficiently and predictably.

Higher level interface routines in Spring can thus be divided into two broad classes: those doing substantive processing to provide a more powerful interface or translate from the terms of one interface to those of another, and those doing only the simple operations required to determine the size of objects referenced through pointers. In either case, these interface routines are written in C, and during execution will eventually reach a point where they are ready to make the transition to system mode, which is done by calling a very simple routine written in assembly language. This simple routine is the actual system call.

As an example, consider the system call which is used by a process to attach to a shared segment. Figure 1 shows the interface routine written in C, while Figure 2 gives the actual

```
| Call format:
| addr_t
| sys_shm_attach(char *name, int name_size, int seg_size, int *first)
|
.globl _sys_shm_attach
_sys_shm_attach:
        linkw a6, #0x0
        movel =SYSCALL_SHM_ATTACH_NUM, sp@-
        trap =SYSTRAP
|
| d0 has the return value
|
        unlk a6
        rts
```

Figure 2: Assembler System Call Routine

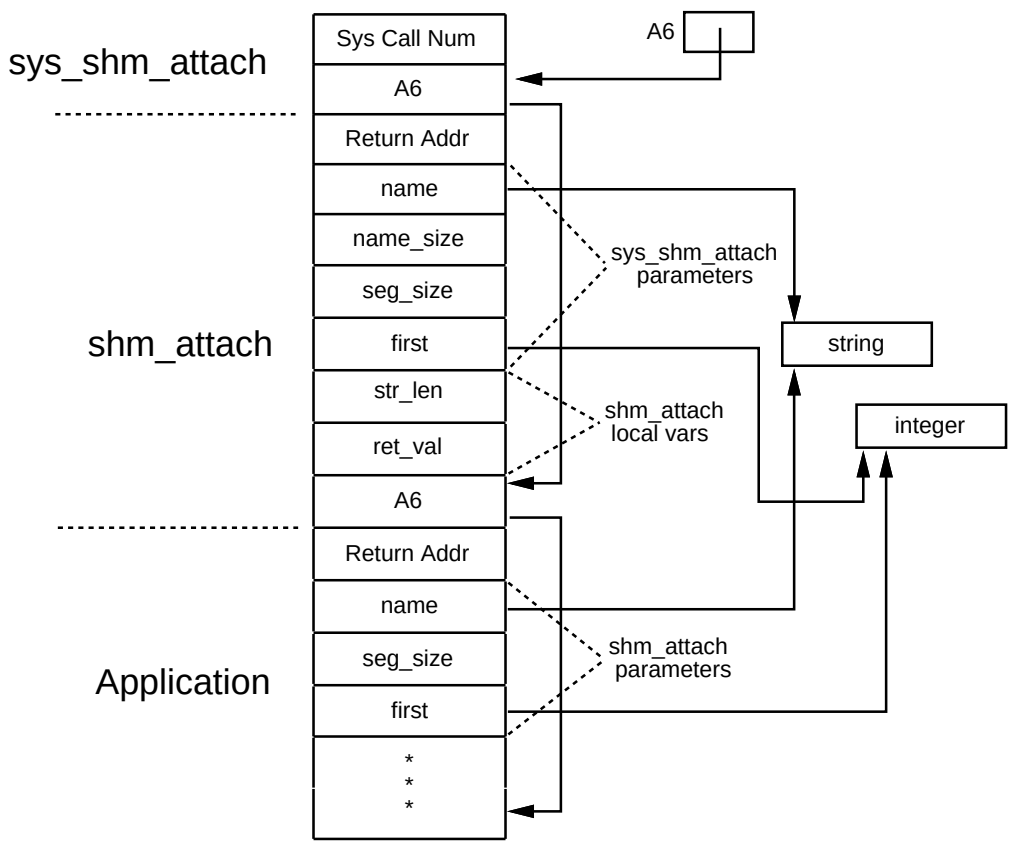

Figure 3: User Stack in System Call Interface Routine

```
.text
        .even
.globl _sys_call_trap
_sys_call_trap:
        jsr _syscall_trap_handler
        rte
```

Figure 4: System Call Trap Handler

system call written in assembler. The system call returns the logical base address of the segment if it succeeds, and returns zero on failure. If the call succeeds, then the integer pointed to by the parameter *first* is set to one if the calling process was the first to attach to the segment, and zero otherwise. This is useful when the first process to attach to a segment is responsible for initializing it. If the call fails, the integer pointed to by *first* gives an error number. The C routine falls into the simpler interface class, since it serves only to relieve the calling context of the burden of determining the input string's length, and then calls the assembler system call routine. Figure 3 illustrates how the user mode stack grows during a system call.

The bottom section of the stack in Figure 3 contains the context of the application process at the point where it makes the call to the **shm_attach** routine. The parameters of the call are pushed on the stack, the return address is added after that, and control is transferred to **shm_attach**. The first thing the routine does on entry, as is true of every C routine compiled by the Spring compiler, is to link the frame pointer **a6**, and to allocate enough space on the stack for the routine's local variables. In this case there are two. When **shm_attach** calls **sys_shm_attach**, the parameters of the call, now including the size of the string containing the segment name, are pushed on the stack along with the return address, and control is transferred to **sys_shm_attach**. This routine also links the frame pointer, but has no local variables. The number identifying this system call is pushed on the stack, and the **trap** instruction causes the transition to system mode. The numeric argument supplied to **trap** is important, since it identifies the event vector table (EVT) entry containing the address of the trap handler. In Spring this is the address of the routine **sys_call_trap**, which is discussed in the next section.

2.2 Initial System Processing

The system call interface routines, which are linked with the application program, are responsible for preparing the information required by the system call to service a given request. When that is complete, they execute the **trap** instruction, and transfer control to the system according a very simple rule. The 68020 uses the event vector table (EVT) to specify what routines should be run in response to a wide range of events. A section of the table is devoted to entries for the **trap** instruction, which takes a numeric argument, and thus selects which of **trap**'s entries to use. Traps are also sometimes called software interrupts.

In the Spring system we have filled of **trap**'s EVT entries with the address of the

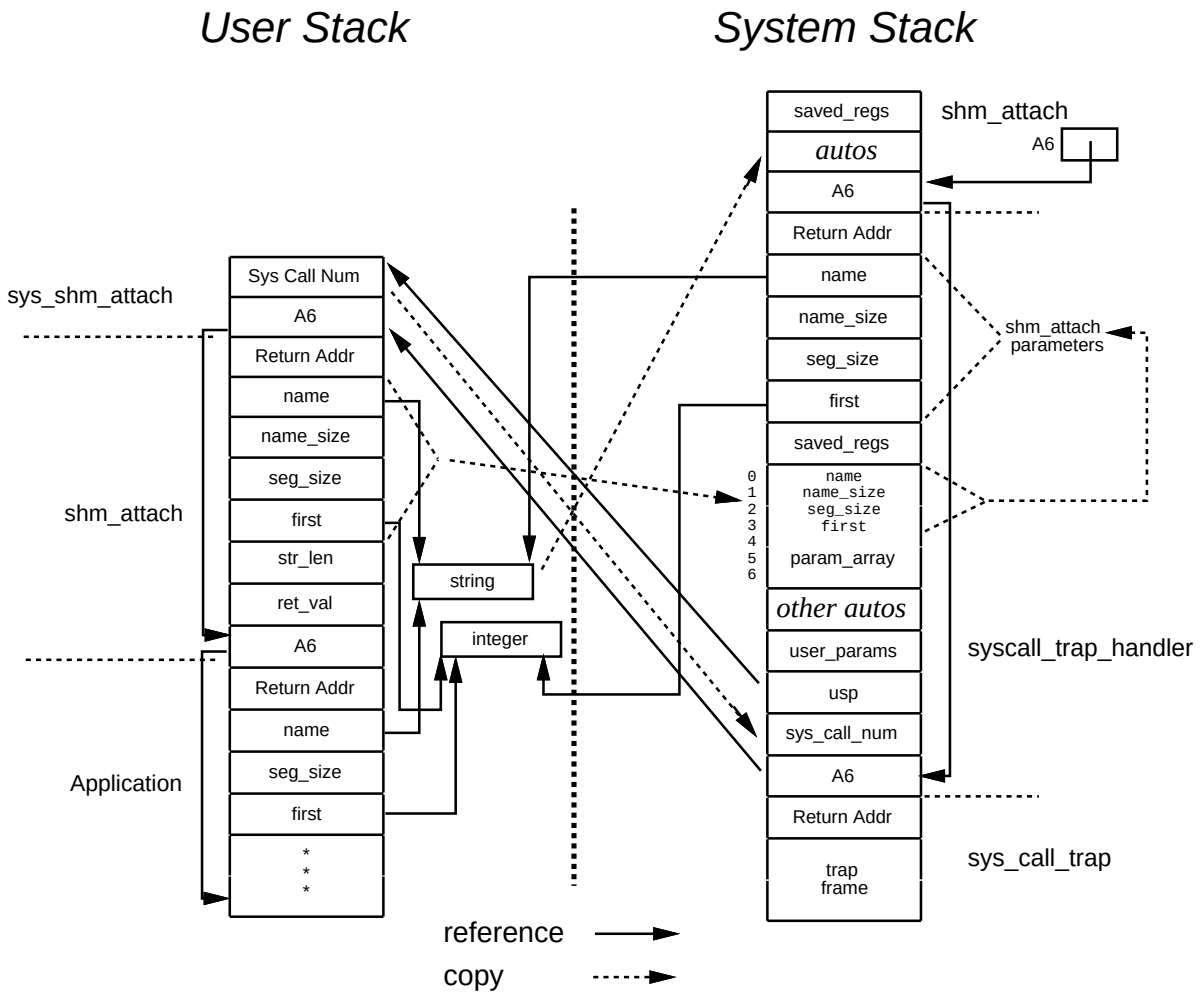

Figure 5: Stack State During System Call Service

sys_call_trap routine, illustrated in Figure 4, through which control is transferred for all system calls. On entry it immediately calls the system call trap handler written in C. However, it is important to note that the **trap** instruction stores information on the system stack specifying the address at which the trap was taken in the application program, and all the status information required to resume execution at the instruction following the **trap**. The **rte** instruction accomplishes this, and is the reason this routine is written in assembler. Routines written in C and compiled by the Spring compiler always use the return from subroutine instruction **rts**, which would not work.

Figure 5 shows the contents of the user and system stacks while the system call is being serviced. The **trap** instruction places the trap frame on the system stack, at the bottom of the system stack in the figure, and transfers control to **sys_call_trap**. This routine immediately calls the C routine **syscall_trap_handler**, which places the return address on the stack, and transfers control.

```
int
syscall_trap_handler()
{
    int     sys_call_num;
    addr_t  usp;
    addr_t  user_params;
    void    (*handler)();
    int     param_count;
```

```
    int     ret_val;
    int     param_array[SYS_CALL_MAX_PARAMS];
```

The definition of **syscall_trap_handler** shows that it takes no arguments, and declares several local variables. On entry to the routine, the frame pointer is linked, and the space for the local variables is allocated on the stack. An interesting detail is that the frame pointer chain actually points back into user space, which is exactly what we want, and ensures that it will have the the proper value when control is returned to the application.

The next set of operations executed by **syscall_trap_handler** bring the system call parameters from user space into system space. The user mode stack pointer (USP) still points to the top item on the stack, which is the system call number. The routine first obtains the USP value, and then copies the integer sized value from that address in user space into the local variable **sys_call_num**, using the **mm_moves_read_proc_mem** routine which permits the system to copy data from the address space of the process which is currently executing. This copy is illustrated in the figure by the dotted arrow going from the top of the user stack to the local variable **sys_call_num** on the system stack.

```
    usp = get_usp();
    mm_moves_read_proc_mem((addr_t)&sys_call_num, (addr_t)usp, sizeof(int));
```

The system call number is used as an index into the system call table, which contains an entry for every system call supported by the system. each entry in the table is of the **sys_call_t** type, and contains an entry specifying the number of parameters the call takes and the name of the routine inside the kernel which services such requests. Empty slots in the table specify **no_handler**, which currently causes a system panic, but could take less extreme measures.

```
typedef struct sys_call {
    int   param_count;
    void  (*handler)();
} sys_call_t;

sys_call_t sys_call_table[SYS_CALL_TABLE_SIZE] = {
              *
              *
              *
    0,  (void (*)())no_handler,   /* 4 */
    4, (void (*)())shm_attach,   /* 5 */
              *
              *
              *
};
```

Note that the fifth slot in the table states that the handler **shm_attach** should be called, and that it takes four arguments. This means that the number identifying the **shm_attach** system call is 5, that 5 is pushed onto the stack just before the **trap** instruction is issued, and is thus the number copied from the application address space into **sys_call_num**.

```
if( (param_count = sys_call_table[sys_call_num].param_count) > 0 ) {
    user_params = usp + 3 * SYS_CALL_PARAM_SIZE;
    mm_moves_read_proc_mem((addr_t)&param_array,
                           (addr_t)user_params,
                           (param_count*SYS_CALL_PARAM_SIZE));
}
```

The system call number is used as an index into the system call table, and if the number of parameters used by the system call is non-zero, then the parameters are copied from the application space into the local array **param_array**. Note that this step calculates the address of the parameters on the *user* stack, and that this calculation depends on the fact that the **sys_shm_attach** routine does nothing except link the frame pointer, and push the system call number on the stack. However, this is true of all Spring system calls, and is thus safe. The advantage of doing it this way is that it avoids having to push the parameters onto the stack inside **sys_shm_attach**, and thus saves having to copy the parameters an extra time. The copy required to transfer the parameters from the user stack to the system stack is symbolized by the horizontal dotted arrow in the figure.

```
handler = sys_call_table[sys_call_num].handler;
switch(sys_call_table[sys_call_num].param_count) {
    case 0:
        (*handler)();
        break;

    case 1:
        (*handler)(param_array[0]);
        break;
        *
        *
        *
    case 4:
       (*handler)(param_array[0], param_array[1],
                  param_array[2], param_array[3]);
       break;
        *
        *
        *
```

The only step left is to actually make the call to the system's handler for this specific call. The pointer to the function is in the *handler* field of the system call's entry in the system call table. However, while the parameters are in **param_array**, they are *not* where **shm_attach** will expect them on the stack, because of the registers that **syscall_trap_handler** may save on the stack. The Spring compiler produces code for all procedures which first links the frame pointer and allocates space for local variables, and then saves registers. If we were *sure* that the routine saved no registers, then we could call the handler as if it had no parameters,

because the entries in the *param_array* would already be in the correct position. This would save a copy step, and thus decrease the system call service time, but is not robust.

Instead, we switch on the number of parameters, and execute a call through the function pointer which explicitly mentions the appropriate number of parameters. This causes the parameters to be copied from **param_array** onto the stack, which is symbolized by the dotted arrow going from one part of the system stack to a higher section. The return address is placed on the stack, and control is transferred to the proper handler. In this case it is the system routine **shm_attach**. As with all C routines, it links the frame pointer, allocates space for local variables, and saves registers as required, as is illustrated by the system stack in Figure 5. This is the context in which the handler executes, as discussed in the next section.

2.3 System Call Specific Processing

Each system call in Spring is eventually served by a routine specific to the purpose. The previous section described how the information prepared by the system call interface routines in the application space was copied into the system space and made available to the proper routine in the context of a normal subroutine call. In this section we discuss how the handler makes use of that information, and the conventions used for return values.

The system routine handling requests to attach to shared segments is **shm_attach**, and illustrates a number of important points. First, its parameters *must* match those of the application interface routine written in assembler. This is obviously required to ensure that the input information specified by the application is properly delivered to the system routine.

```
addr_t
shm_attach(addr_t name,
           int    name_size,
           int    seg_size,
           addr_t first)
{
    int              i;
    char             name_buff[SHM_SEG_NAME_SIZE];
    char            *np;
    int              rv;
    shm_seg_info_t  *shminfop;

    /*
     * copy the segment name into kernel space
     */
    mm_moves_read_proc_mem((addr_t)name_buff, name, name_size);
    name_buff[name_size] = '\0';
    np = &name_buff[0];

    shminfop = k_shm_attach(Current_pcb, np, Board, seg_size, (addr_t)0,
                            (addr_t)0, SHM_ATTACH);
```

The other important point about the parameters is that only they are copied into system space. When the parameter is an integer, that is sufficient, but when it is a pointer the value copied is still relevant only as an address in *application space*. That is why the first thing **shm_attach** does is to use the character pointer and string size parameters to copy the shared segment's name into a buffer in system space. Note also that neither parameter is used after that, which is obviously the correct approach since the pointer is not valid in system space. Once the character pointer **np** is established as a valid reference to the segment name in system space, the routine calls k_shm_attach to actually attach to the segment. This concludes the discussion of how information is prepared for and delivered to the proper routine for handling each system call. We next address the required information is transferred back to the application, and how the system returns from the system call.

Just as there were some basic points about how information is passed into a Spring system call, there are some issues and conventions addressing how information is returned to the application. The second half of the **shm_attach** routine illustrates several important points. Recall that the application interface routine to the system call returns zero on error, the base address of the segment in the process's address space on success, and that the integer pointed to by the *first* parameter is set to indicate that a process was the first to attach to the segment, or an error number on failure.

```
    if ( shminfop == (shm_seg_info_t  *)0 ) {
        if ( first ) {
            mm_write_curr_proc_mem((addr_t)&Errno, first, sizeof(int));
        }
        return((addr_t)0);
    }

    if ( first ) {
        rv = ((shminfop->shmsegp->ref_count == 1) ? 1 : 0);
        mm_write_curr_proc_mem((addr_t)&rv, first, sizeof(int));
    }

    return(shminfop->log_addr);
}
```

The internal service routine **k_shm_attach** returns a pointer to a shared memory information descriptor holding the information about the shared segment, when it succeeds, and returns a null pointer when it fails. If it fails, it also sets the kernel's global error variable *Errno* to the appropriate value. This is copied into the integer pointed to by *first*, and a value of zero returned, as required by the application interface. Note that the address specified by *first* is only valid in the application's address space, and so we use the **mm_write_curr_proc_mem** routine to do this. On the other hand, if the **k_shm_attach** call returns a valid structure pointer, we set the integer pointed to by *first* according to whether the calling process is the first to attach to the segment, as indicated by the reference count. The return value is placed in the *d0* register, by compiler convention, and the specific system call handler returns control to **syscall_trap_handler**.

2.4 System Call Return

When the handler specific to the system call returns control to **syscall_trap_handler**, all operations on the calling process's address space are complete, and the return value is safely in the proper machine registers, as determined by the compiler's conventions. How and why the return values are properly handled is an important but reasonably subtle result of two factors. First, the Spring-C compiler is used to process both the system and application code, so the same conventions apply to the system call interface routines, and to the system code. The compiler uses register *d0* for integer values and *a0* for pointers. Registers *d1* and *a1* are used when 64-bit values are being returned. It is also important to note that these registers are always considered available as scratch registers, and are thus never saved and restored by subroutine calls. The system call specific handlers inside the kernel return whatever values are appropriate.

The second important factor is how **syscall_trap_handler** calls the specific handler. Recall that each entry in the system call table contains a pointer to the specific handler. These were cast as pointers to functions returning *void*, since the structure defining each entry had to specify *some* type for it. However, while **syscall_trap_handler** is declared as returning type *int*, all it does when the specific handler returns is return to the **syscall_trap** routine, which immediately executes the **rte** instruction. None of these operations affect any of the registers in which a value is returned, and so the system call return values are still in the proper registers when control is returned to the application.

The system call interface routine **sys_shm_attach** simply unlinks the frame pointer and executes an **rts** instruction, leaving any return values undisturbed. The higher level interface routine **shm_attach** uses an external definition for **sys_shm_attach** which declares it as returning **addr_t**, so the compiler will look in the proper register for the return value.

3 Creating a System Call

The previous sections have discussed how system calls work. This section discusses how to create a new system call, and specifically addresses what files must be created and modified. The problem is obviously divided into what is required for the application interface, and what is required to support the handler inside the operating system.

Inside the operating system, an unused entry in the system call table must be selected, and filled in with the proper number of parameters and the name of the system call specific handler. Assuming that the root of the kernel's source tree is *kernel*, the system call table is in the file *kernel/sycall/syscall.c*. The system call header file *kernel/inc/sys/syscall.h* should be updated with a defined constant giving the system call number, and an external declaration giving the function template of the system call specific handler. Finally, the system call specific handler should be written, and the makefile in *kernel/syscall* modified to include it in the set of files it compiles.

On the application side, the modifications are made to Spring's C library. The root of the library's source tree is in *sgs/lib/libc* and several subordinate directories hold the routines comprising the library. The *sys* directory holds the actual system calls, which are written in assembler. In the *sys* directory, the file *syscall.m4* contains statements defining the system call numbers. These *must* match those in the header file *kernel/inc/sys/syscall.h*. In fact,

there is some support for automatically deriving *syscall.m4* from *syscall.h*, but has not been fully implemented.

When implementing a new call, the low level routine is written in assembler, and is the same as all other routines, with the exception of the system call number it pushes on the stack. A more subtle point is the function template established for the routine in the appropriate header file. The template specifies the number and type of the parameters, as well as the return type of the function. These must match the declaration of the corresponding routine in the kernel. In particular, the type of return value the template specifies will significantly affect how the routines using the system call are compiled.

Any higher level interface routine required should be put in the directory most appropriate to the system call's function. The *stdio* directory holds routines associated with I/O. The *gen* directory holds routines of general interest. Other directories can be created as necessary.