Address Space Manipulation Support in Spring

Douglas Niehaus

June 1992

1 Introduction

The *process* is a basic building block within the Spring programming interface for representing computations. Each process provides a single thread of control within an independent address space. This model is simple and familiar, and sufficient for representing many classes of computations. However, it is also useful to permit a programmer to use more than one process to represent a computaiton. In these cases, it is often necessary to specify how the processes will interact. This interaction can be based on message passing, or it can be based on creating areas of *overlap* between the address spaces of the processes which are interacting.

The control of address space overlap among processes requires an interface with which to specify the degree an character of the overlap. System support for such an interface is generally based on manipulations of the *memory maps* for the processes involved. However, these kinds of address space manipulations are useful in ways which go beyond simple sharing of data structures among cooperating processes. They are also useful for making memory mapped control registers of hardware devices visible to processes devoted to controlling them. They can be used to provide processes direct access to system status information, or to supply support for IPC. While all of these ideas are familiar, systems do not always make the interfaces they provide for accomplishing these goals particularly easy to use.

The purpose of this document is to describe a simple and effective interface for controlling address space manipulation in Spring, without either imposing unnecessary limitations on the possible shared memory configurations, or requiring excessive programming effort. Section 2 describes the compile time and run time interfaces, and then discusses how these are used in the context of the Spring process semantics. Section 3 then gives some examples of how

basic shared memory would be set up, and then how slightly less common uses of memory mapping would be done. Section 4 then gives a summary and draws a few conclusions.

2 Programming Interface

At the simplest level, the program for a process can be specified without reference to how the memory of the process representing the computation at run time will be arranged. However, as we try to specify more sophisticated or subtle computations, exceptions to this very simple rule arise. The developer thus requires a way to specify some constraints on how the memory of the process should be structured. these constraints could take many forms, but our goal is to provide the simplest interface that fulfills all the roles that shared memory will have in Spring.

There are three basic ways in which shared memory can be used in the Spring system. The first is the most obvious, in that it is a section of memory that is shared among a number of processes, but where the exact physical location of the memory is not important, and where the shared memory should only exist as long as the processes using it exist. The second way of using it is to support a section of shared memory which exists independently of the processes using it, and which msut be assigned to a particular board within the system, but whose exact physical address is not important. This could be useful for defining Spring scheduling resources which represent data structures, since their location must be under configuration control, but the set of processes using the resource may change with time. Finally, the interface should be capable of defining a section of memory assigned to a specific physical memory address. While this segment may or may not be shared, the interface is useful for supporting memory mapped machine control sensor and control registers. At the program level, it is convenient to assign a name to each shared memory area. Processes then use references to these names when defining or attaching to a shared memory area.

2.1 Run Time

The system has a set of three system calls to permit a process to attach and release a shared memory segment. These calls are: **shm_attach**, **shm_attach_phys**, and **shm_release**. The first two give the process the ability to attach to a named segment. In the case of **shm_attach**, the named segment may or may not already exist. If it is already defined,

then the calling process's memory map is modified to map the segment's logical address to the specified physical address. If the named segment is does not already exist, then a physical page is allocated for it on the processor of the calling process, and a logical address is assigned the page. These logical addresses are allocated from a pool common to all the shared segments, ensuring a unique logical address for each shared segment. This is more restrictive than required for all shared segments, but is used to simplify the first implementation. This is discussed further in section 3.

The **shm_attach_phys** routine is almost the same, but has an additional argument specifying the physical address that the section should occupy. This is intended *only* for supporting memory mapped hardware control registers, at this time. As with **shm_attach**, the segment is created if the system does not already know about it. however, if another reference exists, then the physical addresses of the exisitng and new segments must match, or an error occurs. Not also, that the physical address must be *global*, that is, outside the range considered local by each processor board.

The **shm_release** call is simple enough, releasing the calling process's use of the shared section. The complications of this operation arise when the system must consider if other processes are still using it, and if it is meant to survive even when no processes are currently using it.

2.2 Compile Time

At compile time, the designer of a system may wish to define a named segment of memory, and have this definition established as the system is booting, before the processes are activated. This wopuld thus permit the designer to create named memory sections that were not specifically the property of a single process, but of the system as a whole. This would be useful for defining machine control areas, and for specifying the location of shared data structures represented as resources for scheduling purposes.

This ability will be implemented as part of the configuration language, and will permit the creation of two kinds of memory segments. The first will be a segment assigned to a particular processor, but where the specific physcial address is not important. The second will specify a specific physical address, which should be outside the range of the physical page pool of any processor. This is used for machine control areas.

These configuration directives will be used to construct a series of shared segment defi-

nitions which are given to the Spring system at boot time, prior to process activation.

2.3 Process Semantics

Consideration of how the shared memory interfaces might reasonably be used raise a point about the semantics of a Spring process. The problem is that a process may need to engage in an *initialization* phase of arbitrary duration before it is prepared to satisfy real-time performance demands. An example of this would be a process's need to attach to a shared memory segment, and perhaps initialize its contents, before beginning real-time periodic execution.

In response to this, I am implementing the idea of initialization and real-time entry points for a process. Statements inthe configuration language will enable the designer to specify the entry point names, and their addresses will be given in the process executable. When a process is activated, its initialization entry point will be executed. Note that this will be a non-real-time activity. The process context will then be prepared for real-time execution using the real-time entry point.

3 Examples

A few examples of how the definition of memory segments can be used to accomplish several interesting system design and configuration objectives may be helpful in understanding the implementation.

3.1 Shared Memory

The simplest use of a shared memory area is to permit address space overlap between processes for the purpose of sharing data that does not contain pointers. In this case, the name given to the section is sufficient, and the logical image of the physical memory supporting the shared segment could be at different addresses in the two process spaces. However, if the shared segment contains pointers that must be valid within the shared segment, then the segment *must* appear at the same logical address in each of the process spaces. Further, if the segment contains pointers that must be valid in sections of the process spaces that are *not* shared, then the process spaces must be exact images of one another. This is conceivable if we imagine that two or more instances of a given computation coordinate their actions

through a shared section. It is also possible to consider implementing such a situation as multiple threads of control through a single address space. However, we will leave that to a later stage of system development.

The current iplementation makes the simplifying assumption that every shared segment will appear at a unique logical address. This is enforced by associating a map with the shared segment list, which represents the portion of the process logical space where shared segments can be defined. Whena segment is defined, a logical page is allocated for it. Subsequent attempts to attach to the segment will find the existing definition, and will use the assigned logical address. Since no segment can be defined without using this interface, no collisions in logical space will occur, and all uses of a given segment will use the same logical address.

Physical memory is assigned when the first definition of the segment is encountered. If this is in the context of a system call made by a process within its initialization entry point, then the physical page is allocated on the board where the process is executing. If the segment is being defined during system boot, before processes are activated, then the segment definition includes a processor assignment. A message is sent to the processor, telling it to allocate a phsycial page, and return its address. This is useful for assigning Sping resources which represent data structures to specific processors.

3.2 Other Memory Mapped Objects

The interface for defining shared segments can also be put to other uses. The **shm_attach_phys** call permits the process to specify a physical address to which the segment will be mapped. This segment is assigned a logical address from the common pool, as described for more generic shared segments, but is given the specified physical address. Subsequent attempts to attach this segment are checked for agreement on the physical address involved. This would be useful for initializing a robotics application where it required access to the robot's sensors and control registers through a memory mapped image.

Other kinds of information could also be made available to a process through memory mapped images. A standard set of system status information could be defined, and presented using a named segment. Processes which fulfilled a planning role could then attach to this segment to gain access to the system status information. Similarly, A more AI-like application might wish to define a blackboard area which existed independent of any single

process. This could be easily done by defining a named shared segment of the required size. each process using the blackboard would attach to it, and use the application-specific blackboard structure to use it.

3.3 Predefining Shared Segments

The use of predifinition for some shared segments is important for several reasons. First, it enables the developers to define memory based objects that exist apart from any given process. This is important for supporting Spring resources, and for calculating and controlling process execution times. The execution times are involved because they will be different if the memory access required to touch a shared segment is local to the processor where a process is executing, or requires using the VME bus.

Another reason predefinition is important, is that it can be used to ensure support for pointers within shared segments, without conflicts. If we predefine all the segments using pointers, each will be assigned a separate logical address. If we let this be done during process activation, then conflicts may arise, depending on process activation order.

Predefinition also permits a more general interface for machine control, from the process's point of view. If the memory mapped control page for a robot is predefined as a shared segment, then all the robot control processes need only reference the name of the segment, without knowing its physical location. this approach is obviously more flexible with respect to using the robotics application software on machines with different physical configurations.

4 Conclusion

The interface for defining and using shared memory segments is simple, but effective. It provides a way to accomplish the semantics we can reasonably expect for Spring, with a reasonable implementation investment. Further, this implementation can be extended over time to support more subtle semantics, with a minimum of effort.