Porting the SR Programming Language

Gregg Townsend
Dave Bakken

Department of Computer Science
The University of Arizona

October 5, 1994

This document outlines the steps necessary to port the SR system to a new system architecture. The reader should first review the companion document *Installing the SR Programming Language,* which gives an overview of the system and its organization.

The first stage of any port of SR is the construction and verification of a single-threaded system. That is described first. With additional effort, SR can be configured to utilize true multiprocessing on architectures that provide this. This optional second stage is described later.

If you decide to attempt a port, please let us know by electronic mail and keep us informed of your progress. We'll try to assist by answering questions and offering suggestions. If you succeed with the port, please send us a copy of your changes for possible inclusion in future versions of SR. Several of the existing configurations are based on such contributed code.

Part 1 — Porting Single Threaded SR

System Requirements

Most modern Unix systems provide a good base for implementing SR. SR assumes the following:

> Memory is addressable at the byte level; characters are 8 bits, ASCII coded.
> The C type *int* is at least 32 bits wide.
> All pointer types are the same size as a C *long*.
> User stacks can be switched by assembly code.
> A Berkeley-style *socket*(2) interface is available.

The first two assumptions pervade the system. The third is used by runtime routines that use *long* as a universal argument type. Context switching is localized to **rts/process.c** and the assembly code described below. Dependencies on Berkeley networking are concentrated mostly in **rts/socket.c** and **rts/srx.c**.

Porting the C Code

The present SR system has been built in several different environments. It is intended to compile without warning messages under compilers designed for either traditional or ANSI C.

To configure a new architecture, edit the file **arch.h** in the main directory and add a section similar to the others. Use conditional code triggered by a symbol that is predefined by the C compiler.

The SR Programming Language comes from the Department of Computer Science, The University of Arizona, Tucson, Arizona 85721 USA. The implementation is available by anonymous FTP from ftp.cs.arizona.edu, and the SR Project may be reached by sending electronic mail to sr-project@cs.arizona.edu.

Define **ARCH** to be a string describing the target architecture. Define **SFILE** as an appropriately named assembly language file (to be described later). If **<float.h>** is not available as a standard **#include** file, define **LOW_REAL** and **HIGH_REAL** to be the smallest and largest positive **double** values representable on the architecture. (Be sure to use values that **printf** can handle; do not specify something that rounds up and prints as **Infinity**.) If you later find that a special compiler option is needed to accommodate the complex code generated by SR, come back and define **BIGCC** using the existing examples as a model.

To check out the C code, first create an empty file in the **csw** directory matching the name given for **SFILE**. Configure the system according to the installation guide, and run **make –k** to compile.

If there are errors, you will need to modify the source code. Our preference is to try first for a portable solution; or, failing that, to isolate the code in a machine-specific **#ifdef**. Pervasive problems can sometimes be handled by modifying **gen.h** in the main directory.

Writing the Assembly Language Code

Each SR virtual machine is implemented as a single Unix process, with the machines communicating via sockets. Within a virtual machine, SR implements a lightweight process facility. Assembly code to switch contexts is required for each new machine; scheduling decisions are handled in the existing C code.

Three entry points must be supplied for creating, switching, and checking process contexts. A ''context'' is just a block of memory containing stack space, saved registers, and whatever else is necessary. The C code never looks inside a context array and doesn't care how it's laid out. The existing **.s** files give some examples for different architectures.

The needed entry points are:

sr_build_context (entry, context, size, arg1, arg2, arg3, arg4)
void (*entry)(); /* *entry point of function to be called* */
char *context; /* *context array* */
int size; /* *size of context array, in bytes* */
long arg1,... arg4; /* *arguments to be passed to the entry point* */

> Initialize a context array so that, when activated by **sr_chg_context**, it will call the function specified by *entry* with the four supplied arguments. (We assume that no problems arise from possibly calling a C function with too many arguments.) The called function is never expected to return; if it does, a stack underflow abort should occur (see below).
>
> The context need not be set up so that **sr_chg_context** invokes *entry* directly; for example, in the **mips.s** code, it proved easiest to plant the address of some additional assembly code to invoke the desired function when triggered.
>
> The context array will be aligned on an address that is a multiple of 8.

sr_chg_context (newctx, oldctx)
char *newctx, *oldctx; /* *context arrays* */

> Suspend execution of the current lightweight process *oldctx* in favor of the one identified by *newctx*. Generally, this means saving one set of registers (including stack and frame pointers) and restoring another. It is only necessary to save those registers that the C compiler expects to be saved across function calls. **sr_chg_context** is always called explicitly by the SR runtime system, never by random events such as interrupts.
>
> The first time **sr_chg_context** is called, the program is using its original C stack; *oldctx* is zero, and no registers need be saved. On subsequent calls, *oldctx* is a context created earlier by **sr_build_context**.
>
> Some older ports of SR ignore the *oldctx* parameter and instead use a saved value. These versions continue to work for single-threaded SR, but will need modification if they are to be used to implement MultiSR.

sr_check_stk (context)

> Check that the current stack in *context* has not overflowed its bounds. This routine is called by the

runtime system as a sanity check.

sr_check_stk is called only when executing in an **sr_build_context** context.

For all three functions, error conditions should be handled by calling one of the C functions **sr_stk_overflow**, **sr_stk_underflow**, or **sr_stk_corrupted** as appropriate. It is a good idea to check the integrity of a new context, if possible, before switching to it.

Integration and Testing

Go to the **csw** subdirectory and place the new assembly language code in the file named by **SFILE** in **../arch.h**. Add that file name to the **SRC** definition in the Makefile. The Makefile uses *cc*(1) to select and copy the **.s** file to **asm.s**, which is then assembled by *as*(1).

The **cstest** program is provided for testing assembly code in a simpler environment than that of SR. Type **make cstest** to build this test program. When run, **cstest** should produce output that is identical with the file **cstest.stdout**.

When the **cstest** output is correct, incorporate it into the library by moving back to the top directory and again running **make**. This now builds a complete SR system that includes the new code.

Initial system testing can be performed using the **quick** and **examples** subdirectories of the verification suite (**vsuite**) provided with the SR source code. To run these tests, enter ‘‘**srv/srv quick examples**’’. For a port to a new architecture, further testing is in order. Be sure to get and run the full verification suite as described in the installation guide. When the full suite runs successfully you can be reasonably certain of having a solid implementation of SR.

Part 2 — Porting MultiSR

Introduction

MultiSR is a configuration of the SR programming language that utilizes true multiprocessing on systems having more than one processor. In MultiSR, SR’s lightweight threads package multiplexes SR threads on top of concurrent processes provided by the system.

Requirements

To be able to host MultiSR, a multiprocessor system must provide:

- concurrent processes with access to shared variables
- facilities for dynamically allocating additional shared memory
- a fast locking facility, such as spin locks
- a way for each process to determine its index (0..n)

MultiSR is ported by defining macros and functions that interface the SR runtime system to the multiprocessing system. Some editing of Makefiles and other configuration files is also likely to be required.

For MultiSR to work, it is important that the assembly language code be reentrant. Most older implementations of **sr_chg_context** and **sr_check_stk** ignore the *oldctx* parameter and use a variable saved in static memory, which only works correctly on a uniprocessor. (However, the **i386.s** code works correctly on a Sequent because the value it saves is private to a particular process.)

Configuring MultiSR

Select a name for the new configuration of MultiSR that suggests the platforms to which it applies. For example, the Sequent port is named **dynix** because it is applicable to Sequents running the Dynix operating system. Choose a name and substitute it where this document uses **xxxxx.**

Begin by editing files in the **multi** directory. Copy **irix.h** and **irix.c** to make new files **xxxxx.h** and **xxxxx.c** respectively. Edit these files as described below, using the Dynix and Irix files as models.

File multi/xxxxx.h

This file casts the system's multiprocessing facilities in terms expected by SR. It is included by many components of the SR system and by the generated C code.

1. Define **MULTI_SR** with no value. This enables conditional compilation of MultiSR code throughout the SR system.
2. Add any **#include** directives and/or function declarations needed to use the multiprocessing facilities.
3. Define **MALLOC(n)** and **UNMALLOC(a)** macros that function like **malloc(n)** and **free(a)** but allocate shared memory. These will probably map directly to system functions. **MALLOC** must return a **char *** or **void *** pointer just like **malloc.**
4. If the C code generated by SR needs to be compiled with a special **cc** option, such as the ''−Y'' needed for shared variables on the Sequent, define the option as a text string with the name **MULTI_CC_OPT**.
5. Several macros are required for declaring and manipulating locks. Define **multi_mutex_t** as the datatype to be used for lock variables. If the locks themselves must be allocated dynamically, use a pointer type, and define **multi_alloc_lock(a)** to allocate a lock and **multi_free_lock(a)** to free one. If locks are declared statically, define these as **0**. Define **multi_reset_lock(a)** to initialize or reinitialize a lock, **multi_lock(a)** to reserve a lock, and **multi_unlock(a)** to release a lock.
6. Define **SHARED_FILE_OBJS** if the system allows a file opened by one process to be used by another, and **UNSHARED_FILE_OBJS** if not. If you are uncertain, **SHARED_FILE_OBJS** is a good guess and will be validated later by the test package.
7. Define **FIRST_SHARED_FD** and **LAST_SHARED_FD** to give a range of file descriptors that are always shared regardless of the general rule. For example, standard output and standard error are always safely shared because they are created by an ancestor of all the processes in the program and because SR flushes output buffers after every write. If these are the only files, **FIRST_SHARED_FD** and **LAST_SHARED_FD** should be **1** and **2** respectively. If you defined **SHARED_FILE_OBJS**, define **FIRST_SHARED_FD** and **LAST_SHARED_FD** to be **0** and **255**; there is little penalty if **LAST_SHARED_FD** is too large.
8. Define **MAX_JOBSERVERS** to specify an upper limit on the number of concurrent processes. This limit will include the implicit I/O process that is created if **UNSHARED_FILE_OBJS** is defined. For maximum benefit, the limit should allow at least as many user processes as the number of available processors.
9. Define **MY_JS_ID** as a macro that returns a process's index (ranging from 0 to the number of concurrent processes). If this is not easily available from the system, make it a function call and add a routine for obtaining it to the **.c** file below.

File multi/xxxxx.c

This file is included by **../rts/process.c**, causing it to be incorporated in the SR runtime system. It contains three C functions needed for porting MultiSR. All functions must be provided.

1. **sr_init_multiSR()** is called before any locks are allocated or initialized and before any other function in this file is called. It need not do anything, but may be used for any necessary global initialization.
2. **sr_create_jobservers(code,n)** creates **n** concurrent processes, each executing the function **code(arg)**, which never returns. **sr_create_jobservers** is itself assumed to disappear (not return), leaving only the spawned **code** functions running. The **void *** parameter **arg** can be any value; it is uninterpreted by **code** but can be used to communicate a value with **sr_jobserver_first** (below).
3. **sr_jobserver_first(arg)** need not do anything, but may be used for error checking or per-process initialization. It is called immediately by each instance of **code** spawned by **sr_create_jobservers**. **arg** is the argument passed to that code.

Any other variables or functions needed by the port may be added to this file. Initialization, if needed, can be performed in **sr_init_multiSR**. New external and function names should begin with **sr_**. The runtime system function **sr_missing_children** may be used if the requested number of processes cannot be obtained; the Dynix code provides an example of its use.

Testing the MultiSR Porting Primitives

The **multi** directory includes a program for testing the machine-dependent primitives needed by MultiSR. The **Makefile** is currently set up to build and run the test on the existing platforms; it will need editing to work elsewhere. The test program checks several things:

- It verifies that **SHARED_FILE_OBJS** work, if configured
- It exercises the locking primitives
- It allocates shared memory to pass data among processes
- It ensures that process indices can be obtained

The correct output of the test program is dependent on the number of processes. The program **mexpect.c** synthesizes a copy of what is expected, given a process count. This synthesized output can be compared with the test program's actual output to verify its correctness.

Building and Testing MultiSR

Return to the main SR directory and note that **srmulti.h** and **srmulti.c** are symbolic links. Enable MultiSR by redirecting these to point to the two new files **multi/xxxxx.h** and **multi/xxxxx.c**. Also make any necessary **Makefile** or **Configuration** changes, such as to CFLAGS or LIBR definitions.

Type **make sclean** and then **make** to rebuild the system with MultiSR enabled. Try building and running a few SR programs with the environment variable SR_PARALLEL set to enable multiprocessing. Finally, use **srv** to run the SR verification suite as described in the SR porting guide, and be sure to run it at least twice (with SR_PARALLEL both set and unset).

The issue of debugging MultiSR is beyond the scope of this document, but a couple of hints are worth mentioning. The environment variable SR_DEBUG can be set to enable runtime tracing of an SR program; details are contained in the file **rts/debug.h**. An annotated list of the locks used by MultiSR appears in the appendix of *The SR Run-Time System Interface*.