The SR Run-Time System Interface

Amy Morgenstern
Vicraj Thomas
Gregg Townsend

Department of Computer Science
The University of Arizona

October 12, 1994

The SR book (*The SR Programming Language: Concurrency in Practice*, by Gregory Andrews and Ronald Olsson) presents an overview of the SR run-time system (RTS) in its Appendix E. This document describes details of the interface to the run-time system and is aimed at the reader who wishes to modify the system or the generated code that calls it.

The RTS provides *primitives* that are used by the code generated by the SR compiler to access services provided by the RTS. The C files that implement these primitives may be found in the **`rts`** subdirectory of the SR distribution.

Multiprocessor SR (MultiSR) uses locks to avoid interference among processes. These locks are mentioned briefly in some of the function descriptions and are described more completely in the appendix.

General Conventions

Identifiers beginning with an initial capital, such as **`Array`**, **`Ocap`**, and **`Func`**, are the names of types defined in **`sr.h`**.

Several functions have an initial **`locn`** parameter encoding the source file and line number of the SR statement that generated the function call. This information is printed in trace messages or when an error is detected. The decoding of a **`locn`** value is done by **`sr_fmt_locn`**.

Several source files contain initialization functions having names that begin with **`sr_init_`**. Each is called once, before any of the other related functions. Because these initialization functions are trivial, they are not listed in this document.

Memory Management Primitives

The memory management primitives are used to allocate and free storage at run time.

`char *sr_new (char *locn, int len)`

> Allocates memory for a **`new(`***type***`)`** statement. This function is independent of the other allocation functions because failure is returned to the programmer and because the region can only be freed on explicit request.

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.

void sr_dispose (char *locn, Ptr addr)

Deallocates a block of memory that was allocated by **sr_new**.

Ptr sr_alloc (int size)

Allocates a region of memory, by calling **malloc()**, and checks for success. The program is aborted if memory cannot be obtained. This function is used by all other allocation functions except **sr_new**.

Resource-owned memory

The RTS records the ownership of certain allocated regions. A region may be owned by one particular resource or by none. Memory owned by a resource is freed when that resource is destroyed.

Ptr sr_alloc_rv (int size)

Allocates memory for a new resource instance. This is called by resource initialization code immediately after entry.

Ptr sr_alc (int size, int need_rmutex)

Allocates a region of contiguous memory and remembers which resource, if any, is to be considered its owner. If **size** is positive, the current resource becomes the owner; if it is negative, no resource is associated with the region. The parameter **need_res_mutex** is true unless the resource lock is already held.

String *sr_alc_string (int maxlength)

Allocates a string and initializes its header. Ownership is set to the current resource.

void sr_free (Ptr addr)

Frees a region previously allocated by **sr_alc** or **sr_alc_string**. Used when not holding the resource lock.

void sr_locked_free (Ptr addr)

Frees a region previously allocated by **sr_alc** or **sr_alc_string**. Used when the caller already holds the resource lock.

void sr_res_free (Rinst r)

Frees all memory owned by the resource **r**.

Memory Pool Primitives

Some data structures are grouped into *pools*. A pool is a set of memory regions of a particular type. Regions are allocated and freed using the functions below; freed memory remains in the pool for reuse. Pool memory is not owned by any resource.

The pool routines themselves use no externally visible locks; however, they may call user routines that use such locks.

Pool sr_makepool (char *name, int size, int max, Func init, Func re_init)

Initializes a pool of descriptors. Parameter **name** is used in error messages. Each descriptor is of size **size**; a maximum of **max** descriptors are allowed. Function **init** is called when a pool element needs to be initialized for its first use; function **re_init** is called upon destruction to prepare the element for later reuse. Either function parameter may be null if no function is needed.

Ptr sr_addpool (Pool p)

Allocates a new descriptor from pool **p** and returns a pointer to it.

void sr_delpool (Pool p, Ptr el, int have_mutex)

Returns element **el** to pool **p**. Parameter **have_mutex** is true if the lock **pl->pmutex** is already held.

void sr_eachpool (Pool p, Func f)

Calls the function **f** once for each currently allocated member of pool **p**. The single argument passed to function **f** is a pointer to a descriptor. Function **f** must not allocate or deallocate items from pool **p**.

Resource Management Primitives

The resource management primitives are used to create and destroy resources and to inform the RTS when the initial or final code has been executed. There are three major data structures associated with these primitives: the operation capability (**Ocap**), the resource capability (**Rcap**) and the creation request block (**CRB**). The **CRB** includes a packet header used for remote creation; the only field of the packet header that is used by the generated code is the **size** field. These data structures are defined in **sr.h**.

Ptr sr_create_resource (char *locn, int rpat, Vcap vm, CRB c, int rsize)

Called by the generated code to create a new resource instance. **rp** is the resource pattern number, **v** is the target virtual machine, **c** is the address of the creation request block, and **rsize** is the size of the resource capability. The parameters are placed in the CRB and **sr_create_res** is called.

void sr_create_res (CRB *crbp)

Actually creates a new resource instance. Various data structures are created or updated and a process is spawned to execute the initialization code. The caller blocks until the initialization code completes or replies. A pointer to the new resource capability is returned.

void sr_create_global (char *locn, int rpatid)

Creates a new instance of a global in a manner similar to resource creation. If the global is already running, the call is ignored.

void sr_destroy (char *locn, Rcap rcp, int have_rid_mutex)

Destroys the resource **rcp**. A process is created to execute the final code of the resource. The current process blocks until the final code is executed. All memory owned by the resource is freed and all processes executing in the resource are killed. The primitive does not return if called from the resource being destroyed; in this case, the calling process is killed. The argument **have_rid_mutex** is true if the caller holds the lock for **res_pool->pmutex**.

void sr_dest_all ()

Destroys all the resource instances, but not the globals, on the current machine.

void sr_destroy_globals ()

Destroys all the globals on the current virtual machine in a hierarchical order based on what other globals they import.

void sr_finished_init ()

Signals that the resource initialization code has completed. The resource creator is awakened if it was not awakened earlier by a reply. This primitive does not return; instead, the calling process is killed.

void sr_finished_final ()

Signals that the resource final code has completed. The destroying process is awakened. This primitive does not return; instead, the calling process is killed.

Process Management and Scheduling Primitives

This section describes the primitives that create, destroy and schedule processes.

`void sr_init_proc (Func start_code)`

Initializes the process management system. The free list of process descriptors is set up. The job servers are created. The calling process is killed and execution continues by calling **`start_code`** in a new SR process context.

`Proc sr_spawn (Func pc, int pri, Rinst r, int have_lock, a1, a2, a3, a4)`

Creates a new process that will begin by calling **`pc(a1,a2,a3,a4)`**. A new process descriptor is allocated and initialized. Stack space is allocated and set up for process initiation. The process is added to the list for the specified resource, but not placed on the ready list. Parameter **`have_lock`** is true if the caller holds the resource lock.

`void sr_activate (Proc pr)`

Places a process **`pr`** created by **`sr_spawn`** on the ready list.

`void sr_kill (Proc pr, Rinst res_mutex_held)`

Kills an SR process. The parameter **`res_mutex_held`** points to **`pr->res`** if the caller holds the lock **`pr->res->mutex`** and is null otherwise.

`void sr_scheduler ()`

Requests a context switch to the next eligible process. The caller must hold the lock **`sr_queue_mutex`**. Before calling **`sr_scheduler`**, the caller must requeue its own process on the appropriate queue.

`void sr_loop_resched (char *locn)`

Reschedules the current process to let another process execute. It checks to see if any napping or I/O-blocked jobs can be awakened. It is called by the generated code every time **`sr_max_loops`** loop iterations have occurred.

`void sr_reschedule (Proc pr)`

Reschedules process **`pr`** on the appropriate ready or idle list. The caller must hold the lock **`sr_queue_mutex`**.

`void sr_setpri (int newpri)`

Sets the priority of the current process to **`newpri`**. If this is a decrease in priority, a context switch to another process may result.

Remote Request Processing Primitives

The following primitives service requests for a remote machine.

`Pach sr_remote (Vcap dest, enum ms_type type, Pach ph, int size)`

Passes a message of type **`type`** for execution on virtual machine **`dest`**, waits for the reply, and returns a pointer to that reply. Parameters **`dest`**, **`type`**, and **`size`**, plus the current VM and priority, are stored in the message's packet header **`ph`** before sending the message.

```
void sr_rmt_callme (Pach ph)
void sr_rmt_create (Pach ph)
void sr_rmt_destroy (Pach ph)
void sr_rmt_destop (Pach ph)
void sr_rmt_destvm (Pach ph)
void sr_rmt_invk (Pach ph)
void sr_rmt_query (Pach ph)
void sr_rmt_receive (Pach ph)
```

Each of these processes a particular type of request from a remote VM and then sends back an acknowledgement.

`void sr_rcv_call (Pach ph)`

Processes a **`call`** invocation that was returned in response to a remote receive request. The packet is in the form of an invocation block, which is then passed to the original receiver. When a reply is sent to the invocation, it is caught here and passed back to the invoking machine.

```
void sr_net_start (char abuf[])
void sr_net_connect (int n, char *address)
Bool sr_net_known (int n)
void sr_net_more (Pach ph)
enum ms_type sr_net_recv (Pach ph)
void sr_net_send (int dest, enum ms_type type, Pach ph, int size)
```

These primitives service network requests.

RTS Support for Synchronization Primitives

This section describes the RTS primitives required by SR's synchronization statements.

Invocations

All invocations are made using the **`sr_invoke`** primitive. The generated code provides an invocation block that describes the invocation.

`Ptr sr_invoke (char *locn, invb ibp)`

Invokes an operation. The caller must initialize the **`size`**, **`opc`**, and **`type`** fields of the invocation block **`ibp`**. The arguments for the invocation are located immediately beyond the fixed portion of **`ibp`**, and the return values will be returned in this same area. A pointer to the invocation block is returned; the block may have been moved if it called an operation on a remote machine.

`invb sr_reply (char *locn, invb ibp)`

Sends an early reply to the invoker of an operation. A copy of the invocation block is retained by the current process and the original is returned to the invoker. A reply in the initialization code of a resource is treated very much like an **`sr_finished_init`**: the creator process is awakened but the current process is not killed. (The compiler ignores reply statements in a final block.) The address of the new copy of the invocation block is returned.

`Ptr sr_forward (char *locn, Invb obp, Invb ibp)`

Forwards an argument list and responsibility for a reply to another operation. The packet header of the old invocation block, **`obp`**, is copied into the packet header of a new invocation block, **`ibp`**. A new process is spawned to execute the operation specified in **`ibp`**.

`void sr_make_proc (Ocap *ocap, enum op_type type, Func ept)`

Adds a new resource proc operation. Called during resource initialization.

`void sr_kill_resops (Rinst res)`

Kills all operations of resource **`res`**. Any pending input invocations are purged. This is called by **`sr_destroy`**.

`void sr_finished_proc (invb ibp)`

Called by the generated code when a **`proc`** operation has finished. If the operation was called, the invoker is notified. This primitive does not return; instead, the current process is killed.

The Concurrent Invocation Statement

This section describes the commands that are used to execute the **co** statement.

void sr_co_start (char *locn)

Creates a **co** block for the start of the **co** statement. The new block is linked to the current process.

Ptr sr_co_wait (char *locn)

Blocks until an arm of a **co** terminates and returns a pointer to the invocation block used in the original invocation. The generated code can use the invocation block to find out which arm of the **co** terminated. To be able to do this, the generated code sets the **co.arm_num** field in each invocation block before making the invocations inside the **co**. A pointer to the original invocation block is returned so that the generated code can copy result parameters and find out which arm terminated. **NULL** is returned after all arms of the **co** have terminated.

void sr_co_call (Invb ibp)

Handles a call within a **co** statement.

void sr_co_call_done (Invb ibp)

Signals that a call invocation from a **co** statement has returned. If the invoker is still interested in the event, it is notified.

void sr_co_send (Invb ibp)

Handles a send within a **co** statement. The invocation block is copied and returned to the invoker.

void sr_co_end (char *locn)

Indicates the end of a **co** statement. The **co** block is freed.

The Input Statement

An important concept while dealing with the input statement is that of a *class*. The compiler groups operations in an input statement into a class. A class is the transitive closure of the relation ‘‘serviced by the same input statement.’’ The generated code, however, does not have to concern itself with the structure of a class, and as far as it is concerned, a pointer to a class is just a character pointer.

Ptr sr_make_class ()

Makes a new operation class and returns a pointer to it.

Ptr sr_make_semop (char *locn)

Creates an operation implemented by a semaphore and returns a pointer to it.

void sr_init_semop (char *locn, Ptr sems, Ptr initvals, int ndim)

Initializes an unoptimized semaphore or array of semaphores by issuing a series of send operations.

Ocap sr_new_op (char *locn, Class clap)

Creates and returns a single new dynamic operation for **new(op...)**.

void sr_make_inops (Ptr addr, Class clap, int ndim, int type)

Creates one or more input operations and stores their capabilities at **addr**. The parameter **ndim** gives the number of dimensions for an array of ops or is zero for a simple op; **type** is either

INPUT_OP or **DYNAMIC_OP**.

void sr_dest_op (char *locn, Ocap opc)
void sr_dest_array (char *locn, Ptr addr)

Destroys a single dynamic operation or an array of them.

void sr_kill_inops (Ptr addr, int ndim)

Removes local operations from the operation table and purges any pending invocations. As with the preceding function, **ndim** gives the dimensionality of an operation array.

void sr_iaccess (Class clap, Bool else_present)

Reserves access to an input operation class for a sequence of calls to **sr_get_anyinv**, **sr_get_myinv**, or **sr_chk_myinv**. Exclusive access is lost when the any of those calls block the process or when the process calls **sr_rm_iop**. The calling process blocks if another process already has access to the class. Parameter **else_present** is true if the input statement has an **else** clause and is false otherwise.

When an input statement has a synchronization statement, the generated code reserves the class and then iterates through the queue until it finds an acceptable invocation, which it then accepts by calling **sr_rm_iop**. Parameter values in the invocation blocks can be used in the evaluation of the synchronization expression.

When the input statement has a scheduling expression, the generated code inspects all the pending invocations and evaluates the scheduling expression for each. Again, the parameter values can be used.

invb sr_get_anyinv (char *locn, Class clap)

Returns a pointer to the next uninspected invocation block of operation class **clap**. The process must have previously reserved the class by calling **sr_iaccess**. If the class is empty and the **sr_iaccess** call specified no **else** clause, the process blocks until an invocation appears. If the class is empty and there is an **else** clause, a null is returned.

invb sr_chk_myinv (Ocap opc)

Returns a pointer to the next uninspected invocation of operation **opc**, leaving the invocation in the queue. The process must have previously reserved the operation's class by calling **sr_iaccess**. If no invocation is found, a null pointer is returned.

Bool sr_cap_ck (char *locn, Ptr oentry, Ocap opc)

Checks whether the operation entry **oentry** matches the capability **opc** given in an input statement, and returns **TRUE** if so.

void sr_rm_iop (char *locn, char *cp, Invb ibp)

Removes an invocation block from a class queue and releases access to the class. If **ibp** is null, access is released but no operation is dequeued.

void sr_finished_input (char *locn, Invb ibp)

Signals the exit from the body of an input statement. The invocation block is freed if the block if it is no longer needed.

Ptr sr_receive (char *locn, Ocap opc, Bool else_present)

Unconditionally dequeues the next invocation from an operation class and returns a pointer to its invocation block. If no invocation is available and an **else** clause is present, a null pointer is returned; otherwise, the process blocks. **sr_receive** combines the functions of **sr_iaccess**, **sr_get_anyinv**, **sr_cap_ck**, and **sr_rm_iop**, and is additionally the only entry point providing for input from a remote machine.

`int sr_query_iop (char *locn, Ocap *opc)`

Returns the pending invocation count for an input operation. This implements the **`?`** operator.

Semaphore Primitives

Semaphores are really pointers to structures, but as far as the generated code is concerned they are just pointers to characters.

`Ptr sr_make_sem (int init_val)`

Creates a semaphore with a specified initial value and returns a pointer to it.

`void sr_kill_sem (Sem sp)`

Destroys a semaphore.

`void sr_P (char *locn, Sem sp)`

Does a P on semaphore **`sp`**.

`void sr_V (char *locn, Sem sp)`

Does a V on semaphore **`sp`**.

Input/Output Primitives

This section describes primitives that do input and output.

`File sr_open (char *fname, int mode)`

Opens the file named by **`fname`**. Files can be opened in one of three modes: **`READ`** (0), **`WRITE`** (1), or **`READWRITE`** (2). A file pointer is returned on a successful open; **`NULL`** is returned if the open fails.

`void sr_flush (char *locn, File fp)`

Flushes the buffers of the file **`fp`**. The file remains open.

`void sr_close (char *locn, File fp)`

Closes the file **`fp`**.

`int sr_read (char *locn, File fp, char *at, arg1, ...)`

Reads zero or more variables from file **`fp`**, returning the number of items read. Argument **`at`** gives the type and number of the arguments that follow. If no items are read and EOF is encountered, **`EOF`** is returned.

`void sr_printf (char *locn, File fp, String sp, String fmt, char *at, ...)`

Generates output for any of the predefined functions **`printf`**, **`sprintf`**, **`write`**, or **`put`**. Either **`fp`**, a file pointer, or **`sp`**, a string pointer, is null to select printf or sprintf behavior. The argument **`at`** gives the type and number of the arguments that follow.

`int sr_scanf (char *locn, File fp, String sp, String fmt, char *at, ...)`

Reads formatted input from the file **`fp`** or the string **`sp`**, whichever is not null. Argument **`at`** gives the type and number of the arguments that follow. The function **`sr_scanf`** returns the number of items read. If no items are read and EOF is encountered, **`EOF`** is returned.

`int sr_get_string (char *locn, File fp, String *s)`

Reads a string value for **`get(s)`**. The number of characters read is returned.

`int sr_get_carray (char *locn, File fp, String *s)`

Reads a character array for **`get(a)`**. The number of characters read is returned.

`int sr_seek (char *locn, File fp, int seektype, int offset)`

Moves the file pointer to be **`offset`** bytes from the beginning, current position, or end of the file **`fp`**, depending on whether **`seektype`** is **`ABSOLUTE`** (0), **`RELATIVE`** (1), or **`EXTEND`** (2), respectively. The new offset relative to the beginning of the file is returned.

`int sr_where (char *locn, File fp)`

Returns the current position in the file **`fp`** relative to the beginning of the file.

`Bool sr_remove (char *fname)`

Removes the file given by **`fname`**. Returns true if successful.

Primitives dealing with Virtual Machines

This section describes primitives that manipulate virtual machines. The first call to any of these on the main virtual machine initiates execution of **srx**, the central VM coordinator, in a separate Unix process. All of these routines function by sending their parameters to **srx** for processing.

`void sr_locate (char *locn, int n, String *host, String *exe)`

Implements the **`locate`** statement.

`Vcap sr_crevm (char *locn, int physm)`

Creates a new virtual machine on physical machine **`physm`**, returning a VM capability.

`Vcap sr_crevm_name (char *locn, String *host)`

Creates a new virtual machine on the computer named **`host`**, returning a VM capability.

`void sr_destvm (char *locn, int vm)`

Destroys the virtual machine **`vm`**.

Clock Functions

These functions deal with the system clock.

`int sr_age ()`

Returns the age in milliseconds (elapsed time during execution) of the current virtual machine.

`void sr_nap (char *locn, int msec)`

Delays the calling process for **`msec`** milliseconds. If **`msec`** is nonpositive, the caller is rescheduled with no delay behind any other ready processes of the same priority.

Termination Processes

These functions are called to end a program in various ways.

`void sr_stop (int exitcode, int report_blocked)`

Terminates the execution of the SR program. Output streams of all VMs are flushed and the VMs are terminated. The VMs exit with the specified exit code.

`void sr_abort (char *msg)`

Prints an fatal error message and aborts the program.

void sr_net_abort (char *s)

Identical to **sr_abort** but called by the network routines.

void sr_loc_abort (char *locn, char *msg)

Aborts the program giving source line information.

void sr_malf (char *msg)

Aborts the program indicating a run-time malfunction. Only called in ‘‘cannot happen’’ situations.

void sr_message (char *label, char *msg)

Prints a run-time message preceded by a label such as **"warning"**.

int sr_runerr (locn, errno, args ...)

Aborts the program giving source line information and a message selected by the index **errno** from the list in **runerr.h**. Additional arguments may be inserted in the message *a la* printf.

void sr_stk_corrupted ()
void sr_stk_overflow ()
void sr_stk_underflow ()

Issues an error message and aborts the program due to a problem detected by the context switch routines.

Conversion Functions

The following functions convert an SR string to the requested type, returning the converted value:

```
int     sr_boolval (int locn, String *s)
int     sr_charval (String *s)
int     sr_intval (int locn, String *S)
Ptr     sr_ptrval (int locn, String *S)
Real    sr_realval (int locn, String *s)
Array*  sr_chars (String *s)
```

The following functions convert a C string to boolean or integer, returning success or failure:

```
int     sr_cvbool (char *sp, Bool *bp)
int     sr_cvint (char *sp, int *ip)
```

The following functions each convert a particular type to an SR string, returning a pointer to a newly allocated string:

```
Ptr     sr_fmt_arr (Array *a)
Ptr     sr_fmt_bool (Bool b)
Ptr     sr_fmt_char (Char c)
Ptr     sr_fmt_int (int n)
Ptr     sr_fmt_ptr (Ptr p)
Ptr     sr_fmt_real (Real r)
```

The following functions implement the **getarg** predefined function. Each returns 1 if successful, 0 if the conversion fails, or **EOF** if **n** is out of range.

```
int     sr_arg_bool (int n, Bool *p)
int     sr_arg_carray (int n, Array *a)
int     sr_arg_char (int n, Char *p)
int     sr_arg_int (int n, Int *p)
int     sr_arg_ptr (int n, Ptr *p)
int     sr_arg_real (int n, Real *p)
```

```
int     sr_arg_string (int n, String *s)
```

Miscellaneous Utilities

Many functions are so small and straightforward that no detailed description is needed.

Math Functions

These functions implement some arithmetic operations and predefined functions. (Most arithmetic operations require no run-time functions; most predefined math functions are implemented by direct calls to the C library.)

`int`	`sr_imin (int nargs, int v, ...)`	*integer minimum*
`int`	`sr_imax (int nargs, int v, ...)`	*integer maximum*
`int`	`sr_rmin (int nargs, Real v, ...)`	*real minimum*
`int`	`sr_rmax (int nargs, Real v, ...)`	*real maximum*
`Real`	`sr_rtor (char *locn, Real x, Real y)`	*real* `**` *real*
`Real`	`sr_rtoi (char *locn, Real x, int y)`	*real* `**` *int*
`int`	`sr_itoi (char *locn, int x, int y)`	*int* `**` *int*
`Real`	`sr_round (char *locn, Real x)`	`round(`*real*`)`
`Real`	`sr_rmod (char *locn, Real x, Real y)`	*real* `mod` *real*
`Real`	`sr_imod (char *locn, int x, int y)`	*int* `mod` *int*
`void`	`sr_seed (Real x)`	`seed(`*x*`)`
`Real`	`sr_random (Real x, Real y)`	`random(`*x,y*`)`

String Functions

These functions implement string comparison, concatenation, slicing, slice assignment, and swapping.

```
int     sr_strcmp (String *l, String *r)
Ptr     sr_cat (char *addr1, int len1, ... , NULL, 0)
Ptr     sr_sslice (char *locn, String s, int i, int j)
String* sr_chgstr (char *locn, String *s, int i, int j, String *v)
String* sr_sswap (char *locn, String *lside, String *rside)
```

Array Functions

These functions implement several operations involving arrays.

```
Array*  sr_acopy (char *locn, Array *dest, Array *src)
int     sr_acount (Array *a)
Ptr     sr_astring (Array *a)
Array*  sr_aswap (char *locn, Array *lside, Array *rside)
Ptr     sr_clone (char *locn, Ptr addr, int len, int n)
Array*  sr_init_array (locn, addr, esize, initv, ndim, lb1, ub1, ...)
Ptr     sr_slice (locn, a1, a2, esize, nbounds, lb1, ub1, ...)
Array*  sr_strarr (char *locn, Array *dest, int lb, int ub, Array *src)
```

Tracing and Debugging Functions

These functions trace significant events and assist in debugging the run-time system.

```
int   sr_trace (char *action, char *locn, Ptr addr)
int   sr_bugout (char *f, int v1, int v2, int v3, int v4, int v5)
void  sr_debug (char *fmt, inv v1, int v2, int v3)
int   sr_get_debug ()
```

```
void  sr_set_debug (int n)
```

Context Switch Functions

The following system-dependent functions implement SR's underlying lightweight threads package. They are described in detail in *Porting the SR Programming Language*. The actual code is located in the **`csw`** directory.

```
void  sr_build_context ()
void  sr_chg_context ()
void  sr_chk_stack ()
```

MultiSR Functions

The following system-dependent functions support MultiSR. They are also described in *Porting the SR Programming Language*. The actual code is located in the **`multi`** directory.

```
void  sr_init_multiSR ()
void  sr_jobserver_first ()
void  sr_create_jobservers (Func code, int n )
```

References

Gregory R. Andrews and Ronald A. Olsson, *The SR Programming Language: Concurrency in Practice* Benjamin/Cummings, Redwood City, CA, 1993.

G.R. Andrews, R.A. Olsson, M.A. Coffin, M. Elshoff, I. Nilsen, T. Purdin, and G. Townsend, *An Overview of the SR Language and Implementation.* ACM Transactions on Programming Languages and Systems, vol. 10, no. 1, Jan. 1988, pp. 51-86.

Gregg M. Townsend and Dave Bakken, *Porting the SR Programming Language.* Department of Computer Science, The University of Arizona, 1993. Distributed with the SR system.

Appendix: Lock Usage

Dave Bakken

Multiprocessor SR (MultiSR) requires several locks for proper synchronization. This appendix describes the purpose of each lock and the relationships of the locks to each other.

The defined type **`Mutex`** is used to declare a lock; **`Mutex`** variables are accessed only through the macros **`INIT_LOCK`**, **`LOCK`**, and **`UNLOCK`**. In uniprocessor versions of SR these macros have no effect.

Code that concurrently reserves multiple locks must follow nesting rules to avoid deadlock. Interdependencies with other locks are identified for each lock. A linear ordering reflecting these interdependencies appears at the end of the appendix. Terms such as ''first'' and ''last'' refer to the order of acquisition.

SR assumes machine-word atomicity. Locks are not used to protect variables that are a single word, such as the global **`msclock`** or the individual elements of the array **`fp_table`**.

Individual Locks

Locks indicated with **`->`** as part of the name are components of structures. Locks whose names begin with **`sr_`** are global. Other locks are local to a single source file. The defining file name is indicated for each lock.

`cob_st->cobmutex [oper.h]`

This lock protects all the elements of a concurrent statement block structure **`cob_st`**. The lock is nested once, in **`sr_co_end`**, and is acquired after **`cob_pool->pmutex`**. In **`sr_co_wait`**, if the **`co`** block has not terminated, the lock is released until the **`co`** statement has terminated. **`cobmutex`** is then re-acquired.

`mem_mutex [alloc.c]`

This lock protects **`all_mem`**, the linked list of owned memory regions. This lock is taken after the appropriate **`res->rmutex`** has been acquired.

`sr_exec_up_mutex [globals.h]`

This lock protects **`sr_exec_up`**, the variable that tells if **srx** has been started. It is not nested.

`alloc_mutex [misc.c]`

This lock protects the variables **`low_alloc`** and **`high_alloc`**, which are used by RTS Primitives **`sr_new`** and **`sr_dispose`**. It is not nested.

`class_pool->pmutex [oper.c]`

This lock protects the memory allocation pools. It is not nested.

`class_st->clmutex [oper.h]`

This lock protects a class structure. Except for **`rmutex`**, **`res_pool->pmutex`**, and **`sr_main_res_mutex`**, it is obtained before all other locks.

`cob_pool->pmutex [co.c]`

This lock protects the memory allocation pool for the **`co`** statement. The only nesting for this lock is in **`sr_co_end()`**, where it is acquired before **`cobp->cobmutex`**.

`cre_mutex [vm.c]`

This protects **`num_crevm_name`**, which assigns a unique serial number for each VM created using a text string (instead of a machine number). It is never held while another lock is acquired.

currfd_mutex [socket.c]

This protects **currfd**, the current file that **sr_net_recv** is checking. It is never held while acquiring another lock.

send_mutex [socket.c]

Large messages sent to other virtual machines may be broken into parts. This lock ensures that the parts are sent consecutively with no other messages interspersed. **send_mutex** also protects the outgoing message counters. It is not nested.

debug_mutex [debug.c]

This synchronizes debug outputs and is acquired and freed (indirectly) inside the **DEBUG** macro. This lock is always acquired last.

final_mutex [main.c]

This protects **finals_started** and **shutdown_started** to ensure that finalization and shutdown are done no more than once. It is always acquired last.

maxfd_mutex [socket.c]

This protects the variable **maxfd**, the highest file descriptor checked by **sr_net_recv**. It is acquired after **wait_ready_set_mutex** and is never held while another lock is acquired.

mfd_fdm_mutex [socket.c]

This lock protects the variables that maintain mappings between machines and file descriptors, **mfd[n]** and **fdm[n]**. It is nested inside **wait_ready_set_mutex** and is never held when another lock is acquired.

oper_st->omutex [oper.h]

This protects an operation table entry. The lock is acquired before **sr_queue_mutex** but after **clmutex** and **rmutex**. The most complex nesting occurs in **sr_kill_resops**. It acquires **rmutex**, **clmutex**, and **omutex**. Then it calls **sr_kill_sem**, which acquires **sr_queue_mutex** and **sem_pool->pmutex**.

oper_pool->pmutex [oper.c]

This lock protects the run-time operation pool. It is not nested.

proc_pool->pmutex [pool.c]

This lock is used to protect the process memory allocation pool. It must be acquired after **sr_queue_mutex**. A given proc will either be on a protected queue or be served by only one job server, so simultaneous access is not possible.

random_mutex [math.c]

This protects the variables used by **sr_random**. It is not nested.

remote_mutex [remote.c]

This lock protects the arrays **started[MAX_VM]** and **waiting[MAX_VM]** used in function **contact**. It is acquired before the queue mutex.

res_pool->pmutex [res.c]

This lock protects the resource memory allocation pool. The only nesting is with **rmutex** in the function **sr_dest_all**, and here **res_pool->pmutex** is acquired *first*. Function **sr_dest_all** is called during VM destruction, and it must hold **res_pool->pmutex** the entire time so no other resources can be created.

rin_st->rmutex [res.h]

This protects a resource instance descriptor. Only the locks **res_pool->pmutex** and **sr_main_res_mutex** are acquired earlier.

sem_pool->pmutex [semaphore.c]

This lock protects the semaphore memory allocation pool. It is never held when another lock is acquired.

sr_main_res_mutex [globals.h]

This protects **sr_main_res**, which is the capability for the main resource. It is interlocked only with **rmutex**, and is acquired first.

sr_queue_mutex [globals.h]

This protects the I/O list, the nap list, and all queues. It interacts with a number of locks but is acquired before all others except **clmutex**, **rmutex**, and **omutex**. Some functions that need the lock do not know whether it is already held. To handle this situation, the macros **LOCK_QUEUE** and **UNLOCK_QUEUE** are used in nested pairs to access it. The first call to **LOCK_QUEUE** acquires the lock and subsequent calls do nothing but increment a private counter. Calls to **UNLOCK_QUEUE** just decrement the counter; when it reaches zero, the lock is released. Except for **sr_scheduler**, which requires that its caller lock the queue, the pairs are contained in the same function.

sr_fd_lock[n] [globals.h]

This array of locks protects files. It is indexed by file descriptor. Only one file lock is held at any time. Except for **debug_mutex**, file locks acquired last.

proc_st->stack_mutex [sr.h]

This protects **pr->stack**. Routine **sr_scheduler** finds the next **pr** on some queue, then sets **old_cur_proc** to itself and the **sr_cur_proc** to **pr**. It then does a context switch to the job server's private context, which is just an infinite loop in **switch_proc**. Routine **switch_proc** does four things:

1) releases **old_cur_proc->stack_mutex**
2) acquires **sr_cur_proc->stack_mutex**
3) calls **sr_chg_context(sr_cur_proc->stack)**
4) if (**old_cur_proc->status == FREE)**
   free_proc(old_cur_proc)

Since the job server holds no other locks while acquiring **stack_mutex**, it is acquired before all other locks.

wait_ready_set_mutex [socket.c]

This protects **waitset**, the set of file descriptors checked for input, and incoming message counters. It is acquired before **mfd_fdm_mutex**, but other than that it is never held when another lock is acquired.

Linear Ordering

A linear ordering of lock classes is presented below. When two locks are needed concurrently, the one in the lower numbered class is acquired first. There are no nestings between locks in the same class. In other words, while a lock from class *n* is held, no other lock from a class *m* may be obtained if $m \leq n$.

Class 0	stack_mutex *(always held by the current proc)*
Class 1	res_pool->pmutex
Class 2	sr_main_res_mutex
Class 3	rmutex
Class 4	clmutex
Class 5	omutex
	remote_mutex
Class 6	sr_queue_mutex
	wait_ready_set_mutex
Class 7	alloc_mutex
	class_pool->pmutex
	cob_pool->pmutex
	cre_mutex
	currfd_mutex
	send_mutex
	final_mutex
	mem_mutex
	oper_pool->pmutex
	proc_pool->pmutex
	random_mutex
	sr_exec_up_mutex
Class 8	cobmutex
Class 9	maxfd_mutex
	mfd_fdm_mutex
	sem_pool->pmutex
	started_mutex
Class 10	sr_fd_lock[]
Class 11	debug_mutex