Tera Operating System Runtime

Tera Computer Company
2815 Eastlake Ave E
Seattle, WA 98102

October 7, 1994

1 Introduction

This document presents the high level design of the Tera Operating System runtime. The runtime provides the support necessary for streams to execute within the operating system, such as stack management, trap handlers, and synchronization. The OS runtime design continues to evolve as needed.

2 OS Chore Types

The OS runtime implements the following chore types:

- supervisor-promoted chore (`SPChore`)
- kernel-promoted chore (`KPChore`)
- supervisor daemon (`SvDaemon`)
- kernel daemon (`KernelDaemon`)
- parallel chores (`ParChore`)

A *promoted chore* is created as a result of a user chore entering the OS via a privileged entry point. A supervisor-promoted chore is a user chore that has temporarily been granted supervisor privileges as the result of calling a supervisor library routine. A supervisor-promoted chore executes within the user's address space on behalf of the user to fulfill a particular request for system services. When a supervisor-promoted chore executes a kernel library call, it is transformed into a kernel-promoted chore. Promoted chores execute exclusively within the user's address space.

Both kernel and supervisor daemons execute within a privileged task that is constructed at boot time and is referred to as task0. This task has a team residing in protection domain 0 of every processor; this domain is referred to as the OS domain. Both kernel and supervisor daemons execute within the OS domain. Kernel daemons execute requests on behalf of the kernel and provide services that are independent of any particular user request, such as processor and memory scheduling.

The supervisor daemon create routine allows a function to be specified for the newly created daemon to execute. In particular, the function may be the virtual processor entry routine. A

virtual processor serves as a work horse for executing parallel chores. These parallel chores provide system-wide services, such as networking support or disk management. There is another document that discusses the supervisor runtime model in detail, so it will not be discussed further here.

3 Chore Control Blocks

Every stream executing in the OS has a chore control block (CCB) associated with it. The chore control block contains information about the chore, as well as fields used by the compiler (for stack management), and trap handler. By convention, a pointer to the chore control block is stored in a pre-designated general register, and is therefore always accessible to the chore.

Figure 1 describes the class hierarchy for chore control blocks. The base class, `RT_CCB`, is the generic chore control block exported by the generic runtime. It contains fields used by the compiler and the trap handler, including an initial stack segment and a default save area, for saving the register state of the chore when a trap occurs, or when it is suspended (refer to figure 2). Class `RT_Chore` is specialized for the user level runtime and is not discussed here.

Class `OS_CCB` is specialized for use in the operating system, and has additional fields which are common to all OS chores: a pointer to the current `OS_Team` the chore is executing in, the restart address (to resume a suspended chore), arguments which are passed to the restart code, and the CCB type. The `OS_Team` is a per-team data structure maintained by the OS runtime which records information about the OS chores currently executing in the team. The `OS_CCB` and `OS_Team` data structures are further specialized for each OS chore type.

OS chore ids consist of two parts: the task id and a chore index. The chore index is unique within the task. This results in a system-wide unique id, which facilitates debugging. In addition, given a chore id, it is straightforward to identify the task to which the chore belongs. The CCB can then be found by iterating across the team chore lists. (Note that the address of the CCB alone is not system-wide unique because `SPChores` are allocated from task private data segments (to enable

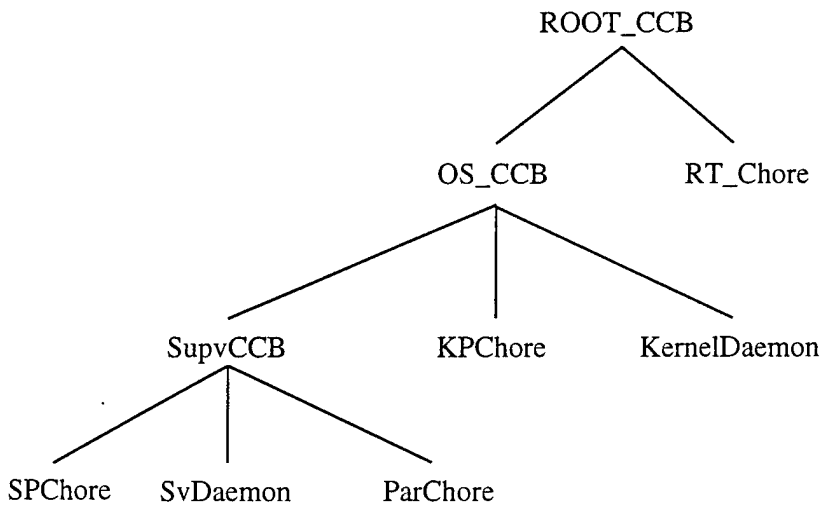

FIGURE 1: Chore Control Block Class Hierarchy

them to be memory-swapped with the task).)

[Note: currently the chore id is 64 bits. This will be modified to 128 bits: 64 bit TeamID and 64 bit chore index.] The following sections describe the execution model and runtime requirements for each OS chore type.

3.1 Supervisor-promoted Chores

System services are provided by a collection of supervisor libraries. The libraries are permanently resident in the program memories of all processors. They are mapped into the privileged common pages with supervisor level privileges; thus the libraries appear transparently in the address space of all executing tasks. A user chore enters the supervisor by branching to a supervisor entry point. A small assembler routine resides at the entry point, which is responsible for installing a supervisor environment prior to calling the actual supervisor routine. The first instruction in the entry stub is a `level_enter` instruction, which promotes the privilege level of the stream to supervisor level. For safety, lookahead is prohibited and all traps are disabled on entry. The entry stub is responsible for installing a privileged environment, consisting of a chore control block, trap handler, and stack.

Figure 2 illustrates class `SPChore`, which is used for supervisor-promoted chores. Note that this is for illustration purposes only, and does not accurately reflect all the fields in the structure. The control block contains fields for storing the user CCB pointer, stack pointer, and trap handler for possible use later during upcalls. It also has a pointer to the `SpChore` pool from which it was originally allocated (which may not be the same as the `OS_Team` it is currently executing in). To enable fast allocation of `SPChore` blocks, the system maintains a pool of pre-initialized blocks on a per-team basis. The size of the pool is fixed, which imposes a limit on the total number of concurrent system calls which can be in progress at any given time within a team. If the maximum is reached, the system call fails and an error code is returned.

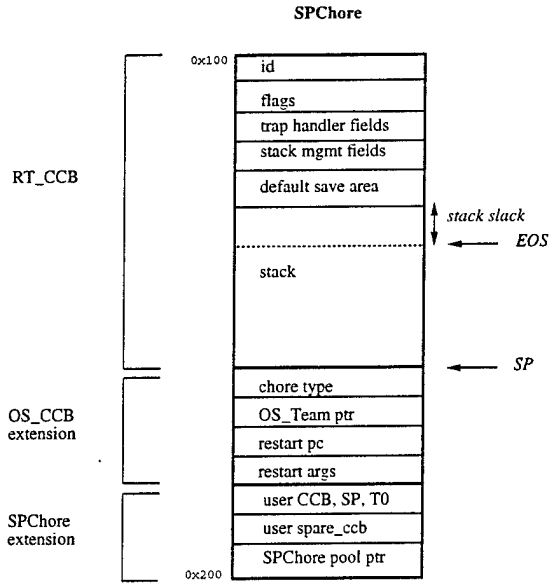

FIGURE 2: Supervisor-Promoted Chore Control Block

When the supervisor call completes, the supervisor environment is freed, the user environment (CCB, stack pointer, and trap handler) is restored, and a `level_return` instruction is executed to return to the user code. (Note: The supervisor entry stub does not touch caller-save registers; if the actual supervisor routine uses them, they are saved and restored according to the normal compiler conventions for "C" functions.)

There are the following restrictions on the activities of supervisor-promoted chores:

- They are not allowed to change protection domains. This would be very dangerous, since the stack is allocated from task private memory, and is not mapped in other domains.
- Suspension requires an upcall to the user runtime.

3.2 Kernel-Promoted Chores

Supervisor-promoted chores obtain kernel services by executing kernel library routines. Similar to supervisor library routines, kernel library routines have protected entry points and the `level_enter, level_rtn` instructions are used to enter and exit kernel mode. When a supervisor-promoted chore enters the kernel, it is referred to as a kernel-promoted chore (`KPChore`). To simplify the synchronization model, and to avoid having to deal with potential deadlock situations, Kernel-promoted chores are not allowed to suspend.

3.3 Kernel Daemons

At boot time, one protection domain on each processor is permanently reserved for exclusive use by the kernel. The streams that execute within these domains are referred to as `KernelDaemons`, and the task is known as `task0`. Daemons implement a simple model of multi-threaded execution. The possible execution states associated with a KernelDaemon are: *running*, *blocked*, *ready*, and *donated*. The daemon state diagram is pictured in figure 3.

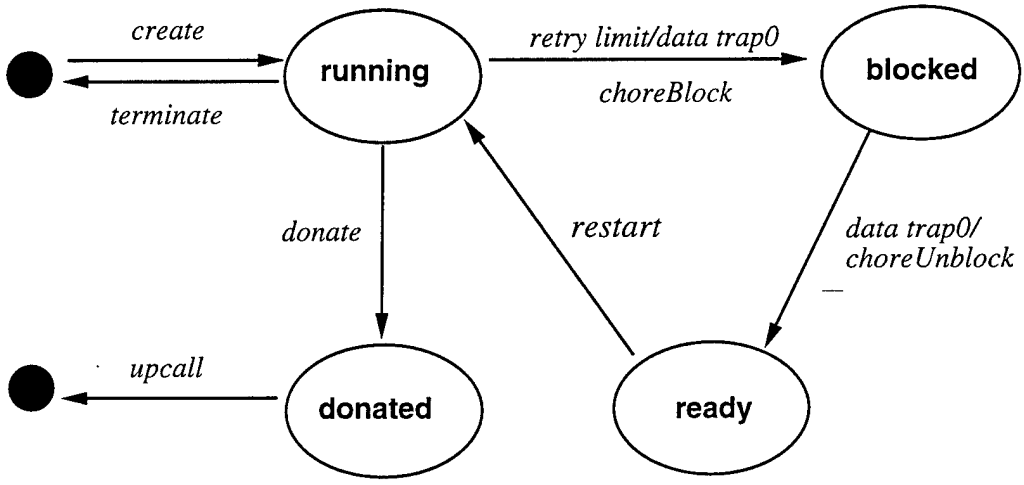

FIGURE 3: Kernel Daemon State Diagram

The following operations on daemons are supported:

- create: create a new daemon which begins execution in a specified function
- terminate: cause the caller to terminate execution.
- block: block the caller
- unblock: make a previously blocked daemon ready to restart
- restart: resume execution of an unblocked daemon
- donate: donate the caller to a user domain in order to execute an upcall (does not return)

A KernelDaemon can be *donated* to a user domain in order to restart a suspended supervisor-promoted chore. Currently, when a daemon executes an upcall, no assumptions are made about its return. (Note: the new upcall policy differs from this.)

Each `KernelDaemonTeam` has one special stream, called the `ListenerDaemon`. The `ListenerDaemon` monitors the unblocked lists for all the teams resident on its processor, and dispatches daemons to restart unblocked chores. It also monitors the per-processor alarm queue, and dispatches expired alarms. The `ListenerDaemon` is also responsible for monitoring the per-processor remote action queue. Whenever a remote action request arrives, it spawns a worker `KernelDaemon` to service the request. The `ListenerDaemon` is not allowed to suspend or to enter user domains.

The system also creates `KernelDaemons` that act as servers for various resources. For instance, each processor has an instance of a scheduler daemon that is responsible for scheduling single-team tasks onto the processor's protection domains. Each server daemon has a private message channel associated with it, that has been registered with the system channel name server. The code executed by a server daemon generally consists of a simple loop, procesing events as they arrive on the message channel. Servers are usually allowed to suspend, pending arrival of requests.

4 Stack Management

Stack management for OS chores preserves the convention adopted by the Tera compiler for user code, except for the following changes:

1. The activation frame size never exceeds the stack chunk size.
2. The stack chunks are allocated from privileged memory.
3. For now, there is only a single stack chunk.

For now, the OS stack is fixed size, and is statically allocated as part of the `OS_CCB`. Eventually, the plan is to move towards a more flexible stack allocation policy, similar to that used in the user runtime. The current approach avoids the complexity inherent in managing resource exhaustion (i.e. running out of free stack chunks).

The generic chore control block (`RT_CCB`) contains the following fields at fixed offsets: an initial stack chunk, a pointer to a pool of additional stack chunks, and pointers to the linkage section and text address of a stack chunk allocation function. There is an agreement between the stack

allocator and the compiler that there will always be at least n words at the end of the stack that can be freely used. Therefore, if the routine is a leaf routine and the frame size is less than or equal to n, the prologue code simply increments the stack pointer. Otherwise, if the frame size is greater than n, or it is not a leaf routine, a stack limit check is done (subtract the activation frame size from the stack pointer and compare it against the end of stack pointer), and the allocation routine is called if necessary, passing it the frame size needed. (Note: the compiler currently optimizes this by first comparing the activation frame size against a built-in constant that denotes the stack chunk size; if the frame size is greater, the stack limit check is not done, and the allocation routine is called. The built-in value can be overridden by specifying a size on the command line.) The user allocator either uses a cached chunk, or calls `malloc` to obtain a chunk of the appropriate size.

For OS code, the chunk size is guaranteed to be large enough to contain the largest possible activation frame, so the allocator never needs to call `malloc` to obtain a chunk. The compiler provides an option which causes the frame size to be printed for each routine. The build for the OS should collect these and make sure nothing is too big. The compiler will eventually also support an option which takes the chunk size as an argument and prints a warning message if the stack size exceeds that. Note that the default stack chunk contained in the `RT_CCB` need not be the same size as the chunks managed by the allocator, but it must be large enough to contain the largest possible stack frame.

Here is a possible solution to the resource exhaustion problem. To speed allocation of stack chunks, free chunks are kept in a pool. To minimize contention, each `OS_Team` has its own pool. The pool is allocated from the supervisor-common heap, and is replenished when it dips below a low-water mark. If the supervisor-common heap has no memory available, the allocation routine arranges to obtain more memory (by making a kernel call to expand the supervisor heap). The allocation routine keeps a private stack chunk just for this purpose, and arranges for additional requests to wait (at a barrier) until the current pool expansion has completed. Because it may be holding synchronization locks, the supervisor stream requesting the memory cannot be suspended or aborted. Thus, it is essential that the kernel fulfill the request to expand the supervisor stack segment even if it means swapping something else out in order to obtain the memory. Of course, this opens the door for potential deadlock situations (supervisor asks for more memory, kernel asks supervisor to swap out a task, ...). A possible solution is for the kernel to keep a small amount of memory just for such emergency situations, which it uses to fulfill the supervisor request (it could even be allocated from I/O memory if necessary), then immediately arranges to free up some memory by mi-swapping some tasks.

5 Exceptions

A trap exchanges the SSW with the contents of target register 0, which contains the address of the trap handler. For supervisor-promoted chores, the trap handler is installed on entry to the supervisor by the entry routine. For supervisor daemons, the trap handler is installed by the daemon creation routine. The trap handler consists of three parts: a primary trap handler, secondary trap handlers, and tertiary trap handlers. The primary trap handler is invoked when a trap occurs, and is responsible for saving the hardware state of the stream in a special trap save area. It then vectors to the appropriate secondary or tertiary handler, which does the actual work needed to handle the trap. The secondary handlers take care of synchronization traps, and the tertiary handlers take care of all other traps. The primary trap handler executes with traps and lookahead disabled. The secondary handlers execute with traps and lookahead enabled, but are

implemented as assembler routines which do not follow regular "C" calling conventions, whereas the tertiary trap handlers do follow "C" calling conventions. The primary trap handler is generic; i.e. the same primary trap handler code is used at user, supervisor, and kernel level. The secondary and tertiary trap handlers are specialized for the runtime model used at each level. The generic RT_CCB contains a pointer to a table of tertiary handlers, allowing the handlers to be customized on a per-chore basis.

The generic CCB contains a default save area, and a pointer to a pool of save areas. The save area is used to store the register state of the chore. The default save area is used if it is available; otherwise a nested trap has occurred, so a save area is allocated from the save area pool. For now, the size of the save area pool is fixed; the maximum trap nesting depth is limited by the size of the save area pool (currently set to 4). Eventually, a more flexible policy may be implemented; note that expansion of the save area pool faces the same deadlock potential as expansion of the stack chunk pool.

6 Synchronization Mechanisms

Two types of synchronization mechanisms are provided for use in the operating system: message passing, and locks. Both are described in detail in separate documents. This section elaborates on the synchronization protocols associated with these mechanisms.

6.1 Mutual Exclusion Primitives

The system supports three classes of mutex locks: *spin*, *spin-suspend*, and *explicit-suspend*. Spin locks are used for short waits, while explicit-suspension is used for synchronization which is expected to result in a long-term wait. Spin-suspend locks are two-phase locks, in which suspension occurs following a timeout; spin-suspend locks are used for situations where the wait is of unknown duration.

Spin locks are used for critical sections of code which regulate concurrent access to shared data structures. Critical sections protected by *spin* locks must execute as quickly as possible, and must not suspend within the critical section. The caller "busy-waits" on a synchronized variable until the synchronization condition is satisfied. If a retry limit trap occurs, the trap handler simply reissues the memory reference. Since concurrent memory references may complete in arbitrary order, it is possible for a chore to be granted access to a critical section even though another chore has been waiting longer. Thus, it is possible (although highly unlikely) for a chore to suffer indefinite postponement (i.e. starvation).

Explicit-suspend locks are used for synchronization which is expected to result in a long-term wait. If the sychronization condition is not satisfied, the caller is immediately suspended. The *spin-suspend* lock is a hybrid which spins until a retry limit trap occurs, at which time the caller is suspended. Like spin locks, spin-suspend locks suffer from unfair scheduling because of the random order in which memory references are retried. *A chore is not allowed to hold any spin locks while it is suspended.*

Every lock type has a specific suspension protocol associated with it, which is enforced by the lock access methods. This prevents multiple synchronization protocols from acting upon the same lock; for instance, it is not possible for one chore to attempt to acquire a lock with spin-only semantics while a different chore attempts to acquire the same lock with spin-suspend semantics.

Chore Type	spin-only	spin-suspend	explicit-suspend	upcall
SvDaemon	x	x	x	
KernelDaemon	x	x	x	
SvPromoted	x	x	x	x
KernelPromoted	x			
ListenerDaemon	x			
IOP Drivers	x			

TABLE 1: Suspension Protocols Based on Chore Type

The chore type determines the suspension protocols available to a chore, and thus specifies the lock types that can be used by a chore. Table 1 summarizes the suspension protocols for OS chores. All OS chore types may spin, but some are not allowed to be suspended. Supervisor daemons executing at supervisor level are suspendable. Supervisor-promoted chores are suspendable, but must execute an upcall to notify the user runtime when suspending. The upcall protocol for suspension of supervisor-promoted chores is described in a separate document. Kernel-promoted chores (which includes supervisor daemons executing kernel calls) are not suspendable (because of the necessity to notify the user runtime to release the virtual processor). In general, `KernelDaemons` are suspendable; the `ListenerDaemon` and the IOP drivers are notable exceptions.

For debugging purposes, it is desirable to store the id of the chore currently holding the lock in the lock itself. The access state of the memory location associated with the lock is initialized to be empty. To acquire the lock, a chore attempts to store a value (it's id) into the memory cell, which causes an empty-to-full transition. Any subsequent attempts to lock the cell will block waiting for the cell to become empty. To release the lock, the chore issues a load of the memory cell, which causes a full-to-empty transition.

6.2 Message Passing

The message passing system is implemented in shared memory, and is used for communication with supervisor and kernel daemons, and for event notification. The message passing system is described in detail in another document; this discussion is limited to synchronization issues involved with message passing.

To briefly review, the basic abstractions supported by the message passing system are messages and channels. A channel is a message queue, identified by a unique channel id. The owner of the channel is designated by its task id. Messages are fixed size, and contain a header followed by some data. Chores from any task are allowed to send messages to a channel, but only chores executing within the task which owns the channel are allowed to receive messages from the channel. The message passing system provides the following primitives: non-blocking send, non-blocking receive, and blocking receive. Send operations are always non-blocking; if the queue is full, the send returns an error code. A non-blocking receive on an empty queue returns with a null message pointer. The blocking receive takes a flag which specifies what the behavior should be if the queue is empty: `SPIN` indicates that the chore should use a spin lock to wait for the reply; `SUSPEND` indicates that the chore should be immediately suspended; `SPIN_SUSPEND` indicates that the chore should be suspended following a retry limit exception. Suspended chores are blocked waiting on a location

associated with the channel. When a message arrives on a previously empty channel, one of the chores waiting on the channel is resumed. The specific method for resuming suspended chores is described below.

A task cannot terminate until all supervisor level activity has terminated. Currently, there is no clean way of aborting suspended SPChores, so we must wait for them to complete. If all SPChores are blocked, the task will be memory-swapped, so it does not consume resources while blocked. Eventually, the synchronization condition the blocked SPChore is waiting on will be satisfied, at which time the task is swapped in and the unblocked SPChore is restarted.

There are a few complications to keep in mind regarding privilege levels. Since the message passing system is used to communicate between supervisor and kernel level, the data structures used to implement the system reside in a data segment which is shared between the supervisor and kernel, to allow supervisor chores to use message passing primitives without having to enter kernel mode. (Caution: this allows a buggy supervisor to step all over kernel messages.) A kernel library call will be provided which allows a supervisor chore to unblock a kernel chore which is suspended pending arrival of a message from the supervisor (kernel call is necessary since the kernel chore suspension record cannot be modified by the supervisor chore).

7 Chore Suspension Mechanisms

This section describes the mechanisms for suspending and restarting OS chores. The following discussion applies specifically to supervisor-promoted chores; the mechanisms for supervisor and kernel daemons are similar except that upcalls are not necessary. The state diagram for supervisor-promoted chores is pictured in figure 4. There are two mechanisms for suspension: implicit suspension/resumption is implemented by the retry limit and data trap0 handlers, whereas explicit suspension/resumption requires an explicit call to suspend and wakeup routines. (Note: this section needs to be updated to reflect the new upcall protocol.)

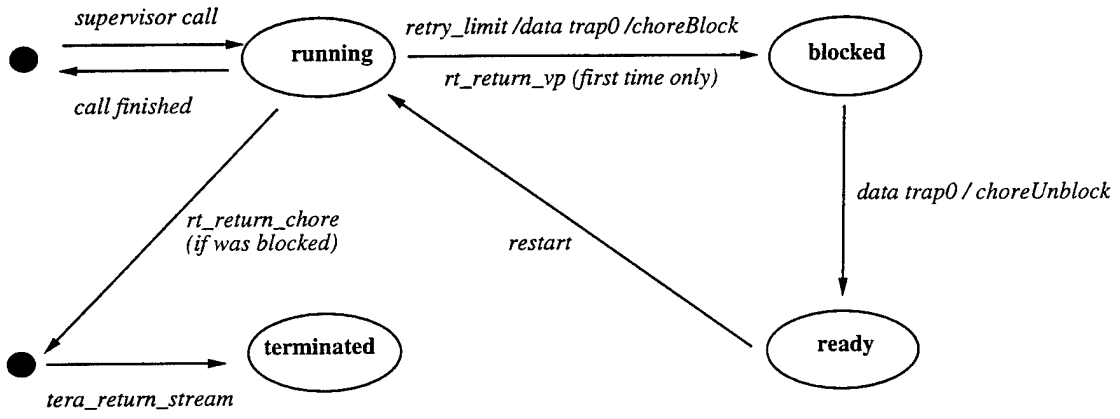

FIGURE 4: Supervisor-promoted Chore State Diagram

7.1 Implicit Block/Unblock

The implicit suspension mechanism is used for implementing the two-phase lock protocol (spin-suspend). The approach is similar to that used in the user level runtime. The details of suspension and restart are handled by the retry limit and data trap0 trap handlers.

A chore attempting to obtain a spin-suspend lock first sets the retry flag in its chore control block to indicate to the trap handler that if a retry limit trap occurs, it should be suspended. To obtain the lock, the chore attempts to store its id into the memory cell associated with the lock, with synchronized access. The hardware transparently retries the memory reference until it succeeds or a retry limit trap occurs. The retry limit trap handler then constructs a chore suspension record (CSR), which contains a pointer to the saved supervisor state and the CCB, saves the current value of the memory cell in the CSR, places a pointer to the CSR in the memory location, and sets the

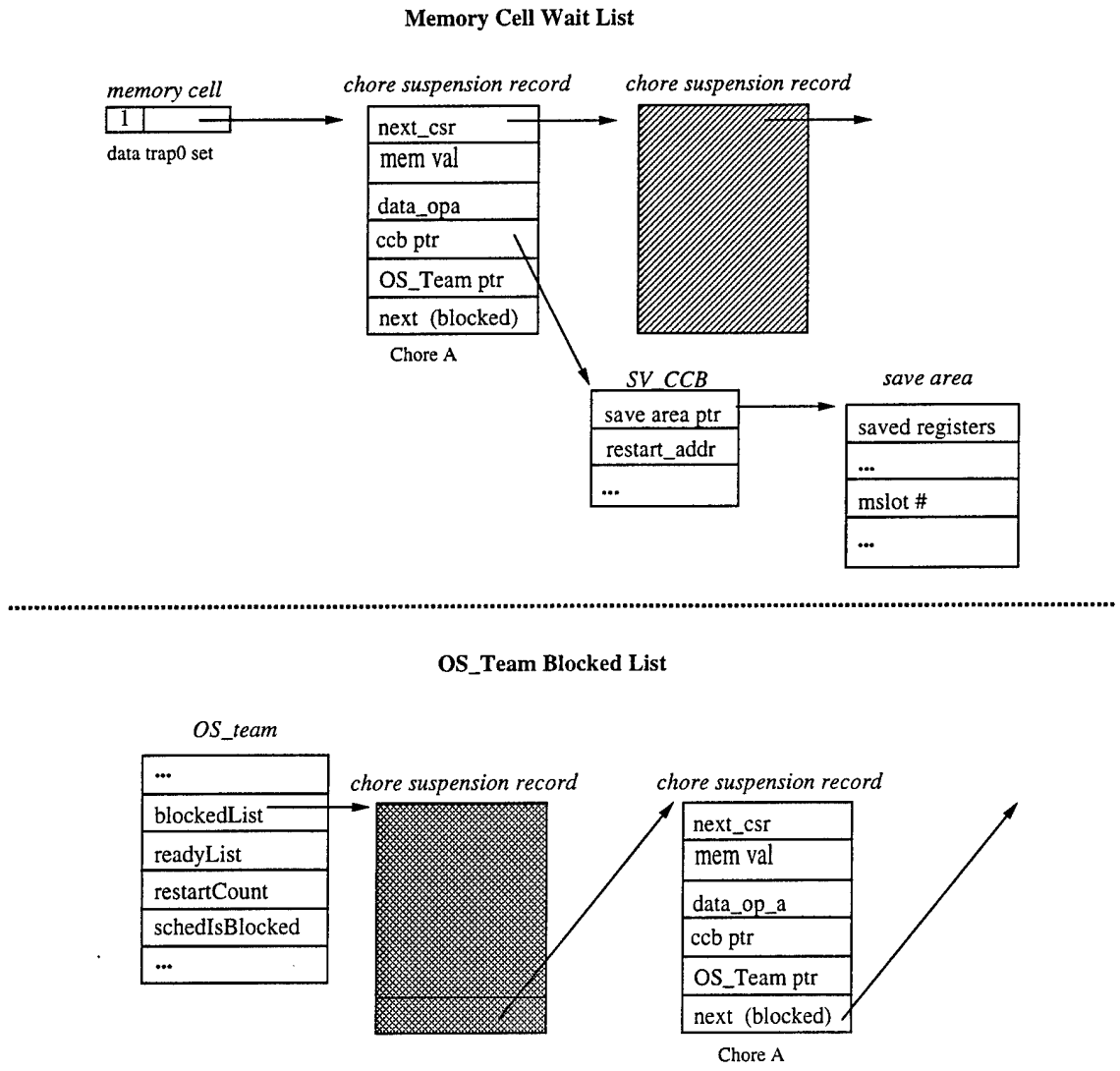

FIGURE 5: Blocked Supervisor-Promoted Chore

data trap0 bit in the access state of the memory cell (see figure 5). This causes other chores which subsequently attempt to acquire the lock to also become blocked. The memory cell wait list is a list of all the chores currently waiting on that location. For debugging purposes, the CSR is also added to a per-team list of blocked chores. The `OS_Team` `blockedList` contains all the supervisor chores in the team which are currently blocked waiting on the same or different memory locations. The chore then executes an `rt_return_vp` upcall to notify the user runtime.

When the spin-suspend lock is unlocked, the chore executing the unlock code attempts to load a value from the memory cell associated with the lock, with synchronized access. Since the data trap0 bit is set in the memory cell access state, the load causes a data trap to occur. The data trap0 handler locates the list of chore suspension records associated with the memory cell, and unblocks the first waiting chore (removes the CSR from the memory cell wait list and from its `OS_Team` `blockedList`, and adds it to its `OS_Team` `readyList`). A pointer to the new head of the memory cell wait list is placed in the original memory location. If this was the last waiter, the data trap0 bit is also cleared. Waking just the first waiter is a little tricky: the data trap0 handler relocates the memory reference by modifying the `data_opa` for both the producer and consumer to point to a new temporary location (such as a field in the chore suspension record); when the `data_op_redo` is executed, it operates on the new memory location. (Note: the producer may redo the operation directly, without using `data_op_redo`.)

At pm-swap time, the user runtime informs the OS (via a flag to the `swap_complete` supervisor call) if all its chores are blocked on external events. The synchronization system cooperates with the scheduler to ensure that tasks which have been blocked on synchronization events are resumed in a timely fashion. The data trap0 handler is responsible for notifying the scheduler when any of these synchronization events are satisfied. It does this by examining the `isBlocked` flag in the `SPTask`, which is raised by the scheduling system when a task is memory-swapped. If high, the scheduler is notified (by sending it an `TaskUnBlock` event), which causes the scheduler to resume the blocked task.

Once a CSR has been placed on the `readyList`, it is the responsibility of the kernel listener daemon to arrange to have the ready chore restarted. Each `OS_Team` has a `restartRequestCount` associated with it, which records the number of chores which are ready to be restarted by the listener. A restart request is posted whenever a chore is added to the `OS_Team` `readyList`. Each `ListenerDaemon` maintains an array of pointers to resident `OS_Teams`, indexed by domain id (registration occurs at team load). The listener sits in a loop, monitoring the `restartRequestCount` for each team. If the count is non-zero, the listener creates a new daemon which executes a `domain_enter` into the target domain and invokes an upcall to the user runtime requesting that the chore be restarted.

To restart a supervisor-promoted chore, the dispatched daemon moves the chore suspension record from the `OS_Team` `readyList` to the (per-task) `notifiedList` and executes an `rt_return_chore` upcall to notify the user runtime. This call results in a `stream_quit` at the user level, so the daemon is, in effect, donated to the user. Eventually, the user runtime will restart the suspended user chore, which will invoke a `callback_resume` supervisor call to restart the corresponding suspended supervisor chore; `callback_resume` removes the chore suspension record from the `notifiedList` and resumes execution of the supervisor chore at the restart address stored in the CCB, passing as an argument a pointer to the save area. Note that the supervisor chore may resume execution in a team other than the one it originally was executing in. The upcall protocol is described in detail in a separate document.

There are occasions when the listener should not dispatch a daemon to the user domain (e.g. the team is being pm-swapped). Class `OS_Team` contains a number of fields used for managing the

synchronization of chore restart requests (see osTeam.w for a detailed discussion).

The suspension/restart mechansims for supervisor and kernel daemons differ only in that user upcalls are not necessary. When a daemon is suspended, it simply saves its state and quits. When it is ready to restart, the kernel daemon dispatched by the listener to restart it assumes the identity of the ready daemon. It removes the chore suspension record from the `OS_Team` `readyList` and frees it, returns its current chore control block to the CCB pool from which it was allocated, installs the CCB of the ready chore, and executes a `level_rtn` to the restart address stored in the CCB.

7.1.1 Explicit Block/Unblock

In addition to the implicit suspension mechanisms described above, routines to explicitly block and unblock chores are provided.

- choreBlock: block the calling chore
- choreUnblockOne: unblock the first waiting chore
- choreUnblockAll: unblock all waiting chores

Note that a chore must voluntarily block itself; i.e. there is no mechanism for blocking an arbitrary chore. These routines take as a parameter a pointer to a wait queue. When the `choreBlock` routine is invoked, a chore suspension record is allocated and enqueued on the wait queue and the team blocked list, the calling chore's state is saved in a save area, and the chore executes a quit instruction.

8 SVC Notification

This section discusses synchronization issues related to supervisor calls. Recall that kernel-promoted chores are not allowed to block within the kernel. Thus, supervisor calls which would normally result in blocking within the kernel must instead block at supervisor level and arrange to be notified by the kernel when the desired condition is satisfied. The following (greatly simplified) pseudocode sketch of a hypothetical supervisor call `svc_xxx` illustrates the solution:

<supervisor call sketch>≡

```
svc_xxx(arg) {
    validate arg
    channel_id = myteam->channel_alloc()
    result = kc_xxx(arg, channel_id)
    while (result == KR_WOULDBLOCK) {
        message = recvb(channel_id, SUSPEND);
        result = kc_xxx(arg, channel_id)
    }
    return result
}
```

The following illustrates the kernel library call. In this example, the variable `foo` is a data structure which is shared by several kernel routines. The field `mutex$` is used to implement critical sections which provide exclusive access to other fields in the `Foo` structure.

<kernel call sketch>≡

```
struct Foo {
    SpinLock mutex$
    EventNotificationList event_list
    int status                                                    5
} foo;

kc_xxx(arg, channel_id) {
    foo.mutex$.lock()
    if (foo.status != XXX) {                                      10
        ev = new EventNotificationRequest(EVENT_XXX, channel_id)
        eventNotificationList.enqueue(ev)
        foo.mutex$.unlock()
        return KR_WOULDBLOCK
    }                                                             15
    else {
        result = do_operation()
        eventNotificationList.postEvent(EVENT_YYY)
        foo.mutex$.unlock()
        return result                                             20
    }
}
```

The supervisor-promoted chore enters kernel level and attempts to execute the operation. If the operation cannot be executed (e.g. due to other concurrent lengthy operations in progress), the kernel library routine constructs and enqueues an EventNotificationRequest. The chore then returns to supervisor level and waits for notification. When notification occurs, the operation is retried. The OS chore that posts the event is responsible for also posting the notifications. Event notification is implemented via message passing.

Note that this approach ensures that the promoted user stream bears the majority of the cost associated with the supervisor call. In addition, the implementation is simplified because the context of the call is preserved.