A Preliminary Look at Spring and POSIX.4

Marie Sie-Een Teo

July 1995

1 Introduction and Motivation

The suite of POSIX standards aims to provide source code portability through standardization. POSIX.4 [10] is an extension of POSIX for dealing with real-time. This is an investigation into the current status of Spring, and comparion of that with POSIX.4 standards. Our goal is to provide recommendations/proposed designs for extensions to the current Spring kernel, which, as much as is beneficial, heads in the direction of POSIX.4 compatibility. We begin by looking at the area of clocks and timers in section 2. The sections after that cover signals, shared memory, memory locking, scheduling, process management and threads, in that order.

2 Clocks and Timers

2.1 Current Spring Implementation

Currently, there is only one system call available to the user, `get_sys_info` which can be passed a flag `SW_CLOCK_INFO` to obtain the software clock value. Time resolution is 0.5 ms on the current hardware.
Clock and timer definitions in Spring are found in relevant header <timing.h>. Some macros are defined for reading the hardware clock, together with the following primitives:

- timer_init:
- timer_add:
- timer_finish:
- ap_start_timer:
- ap_stop_timer:
- ap_clock_isr:
- sp_clock_isr:

2.2 POSIX.4 Clocks and Timers

2.2.1 Clocks

POSIX.4 extends POSIX timers to include implementation defined clocks, supports finer resolution and more flexible signal delivery. If we are supporting POSIX.4 clocks fully, then define `_POSIX_TIMERS` in <unistd.h>, and the following must be in <time.h>:

```
#define CLOCK_REALTIME xxx
```

This is the minimum single clock that must be supported, `CLOCK_REALTIME` (which should correspond to the hardware clock). The type

```
typedef xxx clockid_t;
```

is the defined type for clocks. E.g. `CLOCK_REALTIME` is of type `clockid_t`. With clocks, time is specified by the following data structure

```
struct timespec {
  ...
  time_t   tv_sec;   /* Number of seconds */
  long     tv_nsec;  /* Number of nanoseconds ($<10^{9}$) */
  ...
};
```

This data structure supports nanosecond resolution. The clock `CLOCK_REALTIME` must support a resolution of at least 50Hz or $2x10^7$ ns.

2.2.2 Timers

POSIX.4 allows the dynamic creation of timers. Again in <time.h>:

```
typedef xxx timer_t;
```

defines the type for timers. The data structure used for specifying timer intervals is

```
struct itimerspec {
  ...
  struct timespec it_value; /* Time till first expiry */
  struct timespec it_interval; /* Interval for interrupt thereafter */
  ...
 };
```

The `it_value` is the amount of time until the interval timer expires for the first time; `it_interval` indicates the interval at which the timer will interrupt thereafter.

2.2.3 System Calls

The system calls for clocks are:

- clock_getres: gets the resolution of a clock
- clock_gettime: gets the time of a particular clock
- clock_settime: sets the time of a paricular clock

The system calls for timers are:

- timer_create: creates a timer based on a particular clock
- timer_delete: deletes a timer
- timer_getoverrun: get overrun count for a timer
- timer_gettime: get time remaining on a timer before expiration

2.3 Recommendations for Spring

2.3.1 What extensions are reasonably possible?

Implementing the full POSIX.4 functionality is possible. The underlying hardware clock, ability to create software clocks and timers is already there in Spring. There seem to be no obvious problems of compatibility with the current kernel except for one issue – the use of signals for timers. We may choose to use a different mechanism for signalling timer expirations.

2.3.2 What extensions are good?

The benefits of having extra clock and timer functionality (not just in view of POSIX.4) are many. Spring applications could use a well-defined uniform interface for creating and using clocks and timers, which is exactly what POSIX.4 tries to provide. Hence we think that there should be as compliant an implementation as possible. Appendix A provides the detailed description of the system calls we should provide for full compliance.

3 Signals

Signals are a way to indicate events such as software interrupts and exceptions. The mechanism is slow, and is generally not meant for interprocess communiation (signals usually carry no actual data). What signals provide is a mechanism appropriate for indicating events that occur asynchronously such as interrupts or events that require immediate responses such as exceptions.

3.1 Current Spring Implementation

Currently, there is no signalling mechanism in Spring. Any events are indicated using global flags and polling. There is no signalling mechanism for user-to-user events. [THIS HAS TO BE CHECKED WITH GARY]

3.2 POSIX.4 Signals

POSIX.4 extended POSIX signals from the basic UNIX-like signals, to include much more functionality. One of the main concerns of POSIX.4 was the limited number of signals POSIX supported for application use. Hence, POSIX.4 supports an implementation-defined set of at least 8 signals for use by applications. POSIX.4 signals also can carry data and can be queued to the receiving process and delivered in order (lowest numbered first).

3.3 Recommendations for Spring

3.3.1 What extensions are reasonably possible?

The signal mechanism required by POSIX.4 is quite extensive. Implementing just regular POSIX signals would already require substantial changes to the code. Therefore, adding any kind of signals to Spring would likely involve heavy costs relative to what added functionality Spring might gain since:

- Within Spring signals would have a much more limited role because of the Spring model of hard real-time process execution. Spring is intended for hard real-time processes, execution of which is well-known and predictable. Asynchronous and non-deterministic events are given no guarantees and hard real-time tasks must not be dependent on them.
- Signals are also a costly mechanism, involving multiple traps to the kernel during, posting, delivering and handling of signals [15]. Implementing this heavy cost mechanism does not make sense if we can provide the same functionality in a method more suitable for a hard real-time system.

3.3.2 What extensions are good?

POSIX.4 extensions to signals, are quite complex, giving the signals more of an IPC bent, quite different from the original intention of signals, so for our discussion we use just traditional POSIX/UNIX signals. We divide them into two categories and consider the implications of using them in Spring:

1. *User-to-user* signals are signals arising from a process in user mode.

    - Application defined signals
      These are meant to provide asynchronous, immediate, low bandwidth communication. With Spring, we run into problems because of the asynchrony required. Asynchronous and non-deterministic events are given no guarantees and hard real-time tasks must not be dependent on them.
    - Timer expirations
      Timer expirations are usually reported as user-to-user signals. If we implement the recommendations of the previous section on timers, then we are going to need to provide for this functionality.

2. *Kernel-to-user* signals are signals on exceptions and software interrupts to user processes.

    - Process management/job control signals
      Spring is non-preemptive; processes are broken into tasks with pre-scheduled start times and deadlines. A number of the traditional requirements for process management signals do not arise with Spring process. For instance, a guaranteed process cannot be asynchronously suspended and continued; a process which has some of its tasks already executed, cannot be aborted, although future instances of the process may be.

We conclude that the only funtionality we might really need is some form of user-to-user signals. We do not need to provide kernel-to-user functionality because the asynchrony of signals means many of the kernel-to-user signals would be limited to soft or non-realtime processes. User-to-user signals however, are set up by the programmer, and the process should expect and know how to handle them. Hence, given the high cost of implementing and using signals, we recommend providing a different and simpler mechanism for user-to-user signal functionality. We would especially find this mechanism useful for timers. In recommending a simpler mechanism, we are willing to give up the main advantage of signals, pure asynchrony. Instead, our recommended event handling mechanism relies on polling. Please see Appendix B for the proposed design.

4 Shared Memory

Two or more processes can share a memory object when the object is mapped into more than one address space. This is the fastest form of IPC, requiring only the cost of a memory read. However, there must be some form of synchronization to control access to shared memory to avoid inconsistency. The kernel can provide this automatically or the user may be required to explicitly do this.

4.1 Current Spring Implementation

In Spring a shared memory object is called a shared segment. Shared segments exist as independent objects that are defined at compile time using the SDL, and created during system initialization time. Hence during process initialization they already exist, and processes can attach such segments. The shared segments actually consist of one or more resources, which are generally shared data structures. These resources are each described in SDL, and are grouped together to form a shared segment. Processes attaching the shared segment can then access the resources by name.
Resources have R, W or RW access protections. They can be accessed in exclusive or shared mode or both, and can be attached to a user or system process, or both. Resources can be exported directly or indirectly; if a resource data structure is referenced only by name or local pointers then it can be directly exported, otherwise it must be indirectly exported to keep pointer values valid.

4.2 POSIX.4

POSIX.4 shared memory objects are named regions of storage that can be mapped onto the address space of one or more processes to allow them to share the associated memory. The call to create a shared memory object returns a file descriptor. This file descriptor is used in the calls to set the size of the shared memory object and actually map/unmap in the memory object.

4.3 Recommendations for Spring

Here are the main differences between the Spring approach and the POSIX.4 requirements:

- Spring shared segments are predefined in SDL.
- Spring refers to shared segments and the resources they consist of by name, POSIX.4 uses file descriptors initially , then pathname-like names. POSIX.4 does this because shared memory is closed related to file mapping.
- POSIX.4 (optionally) uses user-level counting semaphores for synchronization. The user must explicitly create, open, wait, post, close and unlink sempahores. However, in Spring, access to shared segments is guarded by a kernel semaphore mechanism. The user only needs to surround the code using a resource with the `with` statement, indicating that the shared segment is a resource that must be available to that block of code.

- In POSIX.4, the user is require to set the size of the shared memory objects. Initially a shared memory object is of size zero and user must grow it dynamically to the required size. A non-zero object size can also be shrunk. In Spring, shared segments are predefined in SDL to be at least the size of one page. The size cannot be changed dynamically.

4.3.1 What extensions are reasonably possible?

The differences listed above are mainly at the API since POSIX.4 does not specify the actual implementation of shared memory objects. For instance, although the system calls in POSIX.4 imply a dynamic shared memory creation and management, it is not required. Predefined shared memory objects in Spring, can have the same calls as the POSIX.4 create calls, but be actually allocated at compile-time. Naming requirements are just conventions, since shared memory objects are not actual files. POSIX.4 user-level semaphores are optional and we can ignore those requirements, especially since we already provide synchronized access. Lastly, any calls to dynamically change the size of shared memory object can be ignored. We ensure that we allocate the maximum required space (to the closest page size).
Finally, both POSIX.4 and Spring provide the same functionality for direct or indirect pointer access to shared memory, allowing the user to specifiy a logical address space for the shared memory object. They also both address the same protection and access privileges.
Conceptually, Spring fulfills all the requirements of POSIX.4 shared memory. All the above differences are not stumbling blocks. Hence, it is possible to change the interface of Spring shared memory calls to fit POSIX.4.

4.3.2 What extensions are good?

There are really no extensions to make, since we have a complete implementation of shared memory, which we can make POSIX.4 compliant if we wish. Whether we wish to will depend on future direction and goals of Spring.

5 Memory Locking

5.1 POSIX.4

POSIX.4 provides memory locking for real-time processes that cannot tolerate the unpredictable and expensive page faults that occur with demand paging and swapping. POSIX.4 provides two levels for locking memory; by entire processes under the option _POSIX_MEMLOCK, or by a range of addresses under the option _POSIX_MEMLOCK_RANGE. When locking entire processes, POSIX.4 allows specifying current and/or future mappings. For using an address range the user has to know exactly where in memory the section of the process to be locked is. This is highly hardware dependent and can be tricky. Any memory that is locked can be unlocked, and locks do not stack, so only one unlock is required to unlock any memory.

5.2 Current Spring Implementation

Memory locking is an essential underlying design choice of Spring []. All processes have their pages preallocated, loaded in at creation time, and kept memory resident. Hence, no demand paging occurs during execution. Since all memory is kept resident by design any memory locking operations are redundant in Spring.

5.3 Recommendations for Spring

There are no extensions required to Spring in this area. When porting POSIX.4 code, we can modify the compiler to just ignore all memory lock/unlock calls.

6 Scheduling

6.1 POSIX.4

POSIX.4 real-time scheduling is intended to be extensible and portable. Three policies are defined

- SCHED_FIFO: preemptive, priority-based
- SCHED_RR: preemptive, priority-based with time quantum
- SCHED_OTHER: implementation defined

POSIX.4 chose FIFO and RR as being most in use by real-time systems. Although there are many arguments as to why such policies are not real-time scheduling policies [13] As well as providing for the commonly used policies, POSIX.4 provides a means for defining a non-portable implementation specific scheduling policy with SCHED_OTHER.
To be extensible, POSIX.4 provides a *sched_param* struct which can be used to pass any number and type of parameter. There are calls to set and get a scheduling policy and its parameters for a process.

6.2 Current Spring Implementation

There is no priority-based scheduling mechanism in Spring. The Spring scheduling algorithm integrates CPU and resource scheduling to provide guarantees. The focus is on resource requirements, not on priority. The idea is that by understanding the resource and timing requirements of a process, it can be broken down into periods of non-blocking execution, called tasks. Such tasks can then be scheduled such that the process is guaranteed to meet its deadline.

6.3 Recommendations for Spring

6.3.1 What extensions are reasonably possible?

To be able to provide RR and FIFO, we have to introduce priority values for processes within the system. This priority can be a user supplied value, within the an implementation defined range. Under FIFO and RR, processes will run based on priority at run-time. The system will not be able to provide the same level of predictability provided using Spring scheduling. We foresee that FIFO and RR priority scheduling would be limited to soft-realtime and non-realtime processes. In the case of hard real-time applications, gaining the predictability is probably going to be worth the cost of redesigning the application to use the Spring scheduling algorithm.
(We can also provide priority values for tasks, and do FIFO and RR for tasks?)

6.3.2 What extensions are good?

Providing the POSIX.4 set of scheduling policies, might be an interesting exercize since it is the first step towards coming up with flexible modular scheduling component. The Spring

algorithm can be provided as the SCHED_OTHER policy, and shall still be the central algorithm of the system.

7 Process Creation and Management

7.1 POSIX.4

Process creation and management depend on the model of process execution that the system provides. The general POSIX/UNIX time-sharing model is based on dynamic forking, waiting and exiting of processes. POSIX.4 inherits this same model.
Forking is used to spawn a new process from within a process. The new process has its own address space, its own user and system stack and is unique and independent from the parent process. Generally, this new process is then *exec'd* (using of the family of exec calls) to replace the process code with the executable code of a different program. A process can also *wait* on another process.

7.2 Current Spring Implementation

In Spring, the process execution model is very closely tied to Spring scheduling approach. All processes are known and well-described using SDL. Hard real-time processes especially are analyzed for timing properties and resource use, so their behavior is understood (and hence predictable) at compile-time. All processess are then created during process initialization during the initialization of the whole system.
Spring processes are well described in SDL so that required resources can be statically allocated and the process created at initialization time. Currently processes are activated at process initialization time as well, although there is no reason why we cannot provide dynamic process activation in Spring. The `create_process` primitive is the Spring process creation primitive. It creates and loads a process, activating one instance. [CHECK WITH GARY]
We know this is not equivalent to traditional forking which usually involves dynamic creation and activation and also the copying of the entire parent process first, before new code is overlaid over the copy.
The *wait* primitive is also a difficult one, in the context of current Spring. The Spring proc_exit() does in spirit what exit() requires, although there are some aspects of cleanup and termination that will be different. For instance, with proc_exit() there cannot be any memory unlocking.

7.3 Recommendations for Spring

7.3.1 What extensions are reasonably possible?

The difference in POSIX.4 and Spring models of process execution makes it uncertain that POSIX.4 primitives will be suitable for Spring. This is closely tied to scheduling issues, e.g. if we were wished to provide priority-based scheduling then, the current Spring execution model does not have to always hold.
Traditional forking and exec-ing are not possible in the current Spring model, however, we can provide dynamic process activation. We can have the compiler replace fork+exec call with the `create_process` primitive. The use of wait is a little more problematic. This is equivalent to a precedence constraint on the exit task of the waited upon task and the task

doing the wait. Same semantics can be achieved with a synchronous communiation from the waited upon task to the waiting task.

7.3.2 What extensions are good?

Again, for real-time systems, process management must be done in vew of scheduling philosophy. If we wish to providing a more non-realtime/soft-realtime bent to Spring, or if we wish to provide priority-based scheduling, then we may consider extending the current process management primitives.

8 POSIX Threads

In this section, we provide an overview of POSIX Threads [11], a much more detailed discussion of real-time threads and a design and implementation of Spring kernel threads can be found in [5].

8.1 Approach

The basic PThreads thread model is priority driven and preemptive, with signal handling, mutual exclusion and synchronized waiting. It must be stressed that PThreads is purely a standard and any compliant implementation of PThreads is not constrained in any way beyond providing the appropriate interfaces. Hence, issues that are related to implementation are not part of the standard. For instance, some issues the standard does not address are: dynamic versus static thread creation, user-level versus kernel level threads, the interface between scheduling at thread level and at kernel/processor level. Any comparison to the two previous packages must be done with this in mind, since for the most part we are evaluating only PThread interfaces and models.

8.2 Description of Threads

PThreads have attributes which can be configured at creation time. The following attributes are currently defined by the standard: scheduling policy, scheduling priority, scheduling scope, scheduling parameters, inherit scheduling, and stacksize. These attributes do not fulfill many real-time application needs. However, an implementation may define further attributes, hence using PThreads for real-time, will involve adding any number of further attributes such as deadline, starttime, worst case execution time, periodic/aperiodic, critical/essential/non-real-time. (And of course, along with these attributes, there must be set of defined semantics for how these attributes affect thread behavior.)
In order to support implementation defined attributes, PThreads specifies the use of opaque attributes objects which are set to the desired configuration values and passed to the *pthread_create()* call. Hence, thread attributes are configurable only at creation time, there is no interface for dynamically changing a thread's attributes.

8.3 Scheduling

PThreads defines only a scheduling interface and no policies. PThreads considers two scheduling models; a system model, where all threads are scheduled against all other threads in the system and a process model where threads are scheduled against other threads within the same process. The latter model requires defining an interface between scheduling at the threads level and scheduling at the kernel/processor level. A thread is then said to have either a *system contention scope* or a *process contention scope*; which is an attribute and may be configured at thread creation time. Other configurable attributes are if the created thread inherits the scheduling attributes of the creating thread, and the scheduling policy of the new thread.
POSIX.4 specifies a priority scheduling mechanism with **FIFO**, **RR** and one **OTHER** (implementation-defined) policies. [10] The PThreads interface is consistant with this and

has an option to indicate that these scheduling policies are implemented. In that case, *priority* is then defined as a thread attribute and there are system calls for setting and getting a thread priority attribute (at creation time). If more than one scheduling policy is defined, then scheduling policies can be specified on a per-thread basis.

8.4 Synchronization

PThreads provides *mutexes* and *condiation variables* as the primary means for synchronization. The PThreads interface does not explicitly provide a way to bound blocking on a mutex. Only one call, *pthread_mutex_trylock()*, which does not block if mutex is already locked, provides deterministic behavior for mutex locking. For condition variables, *pthread_cond_timedwait()* provides bounded waiting. However, all condition variables are associated with a mutex which must be locked by the waiting thread before performing a wait on a condition variable. Hence, to provide bounded waiting, PThreads anticipates (but does not require) the implementation of the *priority inheritance* protocol and *priority ceiling* protocol. Both can optionally be implemented, and there are system calls defined to set inheritance and priority ceilings.

8.5 Other Issues

PThreads defines extensive interfaces for signal handling and thread cancellation. The PThread model provides per-thread signal masks, per-process signal vectors and single delivery of each signal either to a specific thread or to a process (within which the choice of receiving thread is implementation defined). The effect of a signal delivery to a thread should be identical to effect of delivery to a process. Thread cancellation is well-defined with specified cancellation points (when the thread checks to see if it has received a cancellation signal) and the ability to specify cancellation cleanup handlers. Finally, PThread also provides for thread-specific data, to allow threads to have thread-specific global variables.

8.6 Critique

8.6.1 In general:

- PThreads is a real-time standard only in that it is consistant with POSIX.4 (Real-Time Extensions). Remove the considerations for co-existance with POSIX.4 and we basically have non-real-time threads.

- Since PThreads is only a standard, it does not consider the (real-time) implementation challenges which other (real-time) threads packages have attempted to solve and which we would face if we chose to implement PThreads. The only obvious plus for using PThreads is POSIX compliance.

8.6.2 More specifically:

- One limitation of attribute objects is that the creation of a thread requires several system calls since separate calls are required for creating an attribute object, setting

the required attributes then actually creating the thread. However, one advantage of this is that it is easy to set up classes of threads, by reusing attribute objects.

- Another limitation is that there is no link between an attribute object and a thread, except what is maintained by the programmer. So, any thread attributes are not directly visible at the interface level.
- PThreads does not consider schedulability and there is no notion of guarantee.

References

[1] Bach, M. *The Design of the Unix Operating System*, Prentice-Hall, 1991

[2] Corwin, W., Locke, D., and Gordon, K., "Overview of the IEEE P1003.4 RealTime Extension to POSIX," *Real-Time Systems Newsletter 6*, 1, Winter 1990, 9-18

[3] Gallmeister, B., *Programming for the Real World*, O'Reilly and Assoc., January 1995

[4] Gallmeister, B., "POSIX and its real-time extensions", *Real-Time Engineering*, Vol.1, No.1, 9-27

[5] Humphrey, M., Wallace, G. and Stankovic, J., "Kernel-Space Threads for Dynamic Hard Real-Time Environments," Spring Project Documentation, University of Massachusetts, Amherst, MA, 1995

[6] Joseph, M., "Realtime POSIX: Boon or bunk?," *Computer Design*, September 1994, 155-160

[7] Nahum, E., Ramamritham, K. and Stankovic, J. "Real-Time Interprocess Communication in the Spring Kernel," Spring Project Documentation, University of Massachusetts, Amherst, MA, May 1992

[8] Niehaus, D., "Spring System Calls," Spring Project Documentation, University of Massachusetts, Amherst, MA, June 1992

[9] Niehaus, D., "Program Representation and Translation for Predictable Real-Time Systems," Ph.D. Dissertation, University of Massachusetts, Amherst, MA, Feb. 1994

[10] IEEE, "Realtime Extension for Portable Operating Systems (P1003.4/Draft6)," February, 1989

[11] IEEE, " Threads Extension for Portable Operating Systems (P1003.4a/Draft7)," February, 1989

[12] Furht, B, et al., *Real-Time Unix Systems Design and Application Guide*, Kluwer Academic Publishers, 1991

[13] Stankovic, J. and Ramamritham, K. "The Spring Kernel: A New Paradigm for Real-Time Systems," *IEEE Software*, 8(3):62-72, May 1991

[14] Tanenbaum, A. *Modern Operating Systems*, Prentice-Hall, 1992

[15] Thekkath, C., and Levy, H., "Hardware and Software Support for Efficient Exception Handling," Technical Report, University of Washington, July 1994

[16] *VRTX32/68020*, Hunter Ready, Palo Alto, CA, 1987

A Appendix A: Spring-POSIX.4 Clocks and Timers

The following are the POSIX.4 calls for Clocks and Timers, which the Spring extension should adhere to (as much as is possible).

A.1 Clock System Calls

```
int clock_getres(clockid_t which_clock, struct timespec *resolution)
```

Retrieves the resolution of the clock identified by `which_clock`. Returns O on success and sets resolution with the resolution of indicated system clock. Returns -1 on failure with errno set to:

EINVAL The specified clock does not exist

```
int clock_gettime(clockid_t which_clock const struct timespec *current_time)
```

Retrieves the time as indicated by `which_clock`. Returns 0 on success and sets current_time with the clock time. Returns -1 on error and sets errno to the following:

EINVAL The specified clock does not exist

```
int clock_settime(clockid_t which_clock, struct timespec *setting)
```

Sets the identified clock to the indicated time. If resolution is coarses than a nanosecond, value is rounded down. Returns 0 on success and sets indicated clock to the time in setting. Returns -1 on error, does nothing to the clock, and sets errno:

EINVAL The clock specified does not exist or the time setting specified negative or $> 10^9$ nanoseconds

EPERM Calling process does not have privilege to set the indicted clock

A.2 Timer System Calls

```
int timer_create(clockid_t which_clock,
                 struct sigevent*timer_event_spec,
                 timer_t *created_timer_id)
```

Creates an interval timer for the calling process for periodic or one-shot timer interrupts. Timer expirations result in signals being delivered to the process as specified in the sigevent struct `timer_event_spec`. The default, when `timer_event_spec` is NULL is the SIGALRM signal. See signals for sigevent struct description. These signals are not queued (costly to queue an indeterminate number of signals) but `timer_getoverrun()` will return the number of times a timer has expired. Returns 0 on success and the ID of the timer in created_timer_id. Returns -1 on error, and sets errno:

EINVAL The specified clock does not exist, signal number does not exist

EAGAIN Cannot create any more timers for this process (limits are implementation defined)

```
int timer_delete(timer_t timer_id)
```

Frees up timer_id created by create_timer.
Returns 0 on success.
Returns -1 on error and sets errno to:

EINVAL The specified clock does not exist

```
int timer_settime(timer_t timer_id, int flags,
                  const struct timespec *new_setting,
                  struct itimerspec *old_setting)
```

Sets up an interval timer to expired once or periodically. The timer `timer_id` must have been created by `create_timer`. Note : timers are not inherited across forks. The value `new_setting->it_value` is the timer interval after which the first expiry occurs, if zero, the timer is disarmed. `new_setting->it_interval` specifies the following periodic intervals if non-zero, else the timer will expire only once. If flags is 0 then `new_setting->_value` is relative time to current time. If flags has `TIMER_ABSTIME` set, then it is absolute time. The old setting is returned in `old_setting` if `old_setting` is not NULL. Returns 0 on success. On error, returns -1 and sets errno:

EINVAL The specified clock does not exist

```
int timer_gettime(timer_t timer_id, struct itimerspec *current)
```

Returns current with the current setting of the interval timer of the timer timer_id. Current will contain the amount of time remaining until the timer expires next, and the interval between expirations. Returns timer interval relative to current time even if setting is using absolute time.

```
int timer_getoverrun(timer_t timer_id)
```

Returns the overrun count for the timer timer_id. This is the number of timer expirations that have occured between the time the timer expiration signal was queued to the process and the time at which the signal was delivered to the process. Timer expiration signals are not queued (because there can be a potentially large number). The max value for the count is DELAYTIMER_MAX. When it overflows DELAYTIMER_MAX is returned. If signal is never delivered then the count is undefined, hence the last expiry signal should be handled before retrieving the overrun count. Returns overrun count on success.
Note: minimum allowed value for DELAYTIMER_MAX is 32 (too small) A good value would be $2^{sizeofoverruncount} - 1$.

B Appendix B: Spring Event Handling Mechanism

We recommend a very simple event mechanism to cover user-to-user events. This mechanism is not equivalent to signals, since involves flags and polling. For now, we have only one event:

- timer expiration:
  Similar to POSIX.4 SIGALRM, indicates that a timer owned by the process has expired. Currently we use flags (global and local) to indicate that a timer has expired.

B.1 Design

We know that we can provide event notification mechanism on top of IPC. Briefly, every process will have an event port, consisting of 1 message word (a bit map of event flags) created automatically at process initialization time by the system. BUT, a better design would be NOT to go through the IPC and implement a separate mechanism. We use event flag groups similar to the VRTX32 implementation [16].
An event flag group is just a word (16 or 32 bit) with each bit being a flag indicating either a set (1) or clear (0) state.
Primitives are needed to :

- `create_event_flag_group`
  We can create an event flag group per process at init time.
- `clear_event`
  Clears one or more events in an event flag group. This should be done after an event has been handled.
- `delete_event_flag_group`
  Deletes the event flag group
- `post_event`
  Post an event(s) to the specified process (event flag group)
- `query_event`
  Returns the current event flag group Called on every task entry to check for events

B.1.1 SIGALRM

The event SIGALRM should be defined.
Algorithm for timer expiration:

1. the process using a timer must create its own event flag group
2. the process can then use timer system calls to set a timer
3. then process polls event flag group
4. on timer expiration, the kernel must post the event to the flag group

Note: pending on timers implies that timer use is restricted to soft and non-realtime processes.

B.1.2 Issues

- Events are asynchronous hence any hard real-time processes should not be dependent on an event.
- A process using events, should check for events on its own and always be prepared to handle all possible cases involving event occurence/non-occurence.
- Where would we put the event flag group for the process? Should be a global resource like ports, assigned from kernel pool, and belonging to the process.