

Introduction to Chronolytix

A User-Space Deadline Processing Real-Time System for POSIX/Linux

Abstract

Traditionally, simple real-time systems have been designed as “simple” control loops perhaps with interrupts/background “tasks”. More complex systems may add multiple tasks but maintain a “master” task structure. Even more complex real-time systems may add a table driven build-time determined “time-based” scheduler. Further increasing complexity makes even this approach difficult/impossible to scale. With Linux and other non-real-time POSIX based OSES, developers desire to leverage the rich feature/application set are forced to use expensive “real-time” attached processor or a guest-OS real-time executive collocated on the same processor (ex., RTAI -- Real-Time Application Environment <http://www.rtai.org>). Neither of these approaches gives the programmer a unified application model. In addition, debug can be particularly painful when the POSIX OS does not control the resources of the system.

The Chronolytix system uses lightweight processes (lwp's) to allow a system with 1000s or 10000s of “tasks” running concurrently within a real-time constrained environment. It includes high resolution timing and timers that are designed to scale as well. Most importantly, Chronolytix places few application restrictions on the real-time system to be built – unlike an attached processor or collocated executive. And since it is purely Linux, it allows leveraging the Linux feature and application set. A schema for creating “building blocks” components for large and scalable real-time systems is presented.

Enabling Technologies

In order to make Linux into a real-time system several technologies are required/utilized. Since Linux 2.2, the system calls for POSIX real-time scheduling have been present. Chronolytix makes use of SCHED_FIFO first-in-first-out scheduling for all its real-time scheduled entities. In addition, since Linux is not a real-time OS, the real-time scheduling has been largely to control relative priority rather than the greater goal of meeting latency guarantees. Also in Linux 2.0, and subsequently improved every release, Linux added symmetric multiprocessing – the ability to load share across multiple processors. This is notable even on a single processor because it implies some degree of reentrancy to the Linux kernel. A great deal of

work has been done to decrease the non-reentrant portions of the Linux. Several attempts in Linux 2.4 were made to address preemption latency, however, the BKL (big kernel lock) and the inherent non-preemptability of the 2.4 kernel preclude sub-millisecond determinism. In Linux 2.6, Linux kernel preemption was made a compilation time option. Predictably, the next problem with short deterministic latencies was priority inversion of shared locks. There are a series of patches by Ingo Molnar (<http://people.redhat.com/mingo/realtime-preempt>) which add the ability to use a priority inheritance to all Linux wait-mechanisms. Included in this patch, but which must be enabled manually in kernel/rt.c, is the ability for timeshare scheduled tasks to also inherit real-

time priority. This is required for the Chronolytix system to make real-time deadlines if timeshare processes are present (and they will be.) The downside of timeshare priority-inheritance is if a timeshare process busy-waits it may lockup the system. Since the purpose of the Chronolytix system is to make processes react, busy-wait should be eschewed entirely.

From the user-space perspective, the next major enabler is support in the GNU C Library (GLIBC) for NPTL – the Next-gen POSIX Thread Library. The previous “linuxthread” POSIX library has severe problems with POSIX compliance and scalability. It is unsuitable for building real-time system based on “cheap” threads. Using GLIBC-2.3.5, the 1:1 pthread=>linux thread model is available with POSIX thread semantics. In addition, pthread_mutex, pthread_cond (condition variables), and pthread_barrier, etc. use the new (Linux 2.5) futex (fast user mutex) mechanism. This means that an uncontended pthread_mutex may be claimed/released for the cost of a memory reference rather than a system call. The NPTL in turn requires a GCC version (3.3.x or above) which allows hooks for thread local storage (ie., GCC is thread aware and cooperates with the GLIBC). This allows low cost user-space-only implementation of many pthread thread-specific calls such as pthread_self and the pthread_get/set_specific. The GCC 3.3.x require an assembler and linker that are also thread aware. Taken together, the NPTL threads and the fully preemptible Linux kernel, give a decent real-time response in the presence of a time sharing load.

Coding Standards

Chronolytix is, in its essence, an event-driven programming model. The events in this case are units of work that have real-time deadlines that should model the real-world phenomena they interpret or control.

Contested ownership of shared objects and their respective lifecycles – who owns what, when and for how long – is one of the most costly to debug problems with event-driven reactive programming. To address this, Chronolytix makes several “coding standard” directives.

First, for shared objects with multiple “owners” or race-y shared lifecycles, use of reference counted lifecycle control is recommended. Reference counting object allocation is the means that Java and other “garbage collected” languages use to know when an object may be reclaimed. It is commonly used in C++ where overriding rvalue operators can make reference counting invisible to the application user (<http://www.boost.org>). This means that simple use of a reference counted variable in scope will incur cost unbeknownst to the class user. More on Chronolytix reference counting automatic object lifecycle control is documented below.

The second coding standard is that the caller owns passed arguments on return from the callee. That is, if a pointer to an object is passed to a subroutine, that subroutine must copy-out the argument content (or acquire a reference in the reference counted case) prior to returning to the caller. This allows the caller to overwrite/free/etc. the passed arguments subsequent to the subroutine call. The “caller owns the lifecycle of passed arguments” contract eliminates nasty race conditions and possible leaks.

Another strongly recommended coding standard is that shared structures have as their first element a 4 byte RTMagic type. This element is used to hold an unique 4 byte packed ASCII identifier, for example an armed timer has a magic value “Timr”. (Although in reality there is no enforcement of unique). Having all shared data structures be “self identifying” allows for ease in debug of malloc/free, as well as postmortem heap usage (for leak detection).

But the real power of structure identification magic is to allow the code that uses “passed” data to validate its contents prior to using it. By quickly identifying use-after-free and heap overwrite (corruption) or incorrect object lifecycle (pointer reference to stack), the system can be aborted with the stack and data prior to “corruption”. Without runtime structure

validation subtle heap corruption may not be detected for a considerable amount of time and many incorrect memory writes later. To encourage low-cost use of validation, RTMagic.h includes macro expansion of per-type inline validation code. The net cost of validation is ~4 x86 instructions to validate a pointer against type.

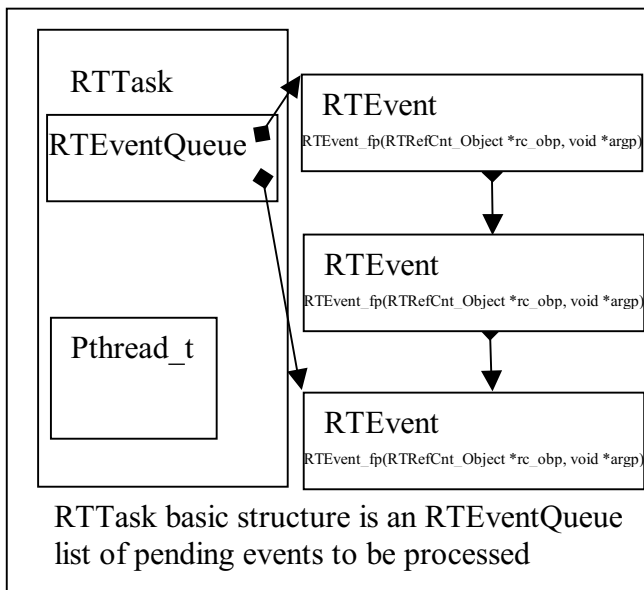
Task (RTTask and RTEventQueue)

```

typedef struct {
    RTMagic magic;
    pthread_mutex_t mutex;
    pthread_cond_t condition;
    RTEventPrivate *first, *last; /* header is first and last */
    int closed; /* if TRUE, return NULL on last event and queue no more */
    unsigned long event_count; /* number of events read */
    RTEventPrivate *current_event; /* non-NULL if not waiting */
} RTEventQueuePrivate;

typedef struct task {
    RTMagic magic;
    pthread_t thread;
    pid_t tid;
    enum task_state state;
    char *name;
    void *(*start)(void *); /* start routine */
    void *argp; /* optional argument to be passed */
    RTEventQueue eq;
} Task; /* our private task construct */

```



The RTTask is a fully functioning Linux schedulable pthread_t which is bound at construction time to an RTEventQueue. The RTEventQueue is the basis of FIFO ordered event processing in both real-time deadline RTTasks known as RTReactiveResponse and dedicated RTTasks. An example of a dedicated RTTask would be an RTTask used for logging or asynchronous I/O processing. Applications use an RTEventQueue to send an RTEvent work quanta to the RTTask to be processed.

Reactive Response (RTReactiveResponse)

A RTReactiveTask is a real-time method which is run on a RTTask but which has no stack context of its own. That is, a reactive task is bound to a physical thread and executes on that thread. Since it has no stack context it may not block (ie., sleep) on the shared thread – or other reactive tasks bound to the same thread would miss their respective real-time commitments. The advantage to this discipline is that the encapsulation of a reactive task is in 10s of bytes, and therefore 10,000s of them can be present with reasonable memory usage. A typical POSIX thread has a persistent minimum memory usage of 4KB user + 8KB kernel = 12KB total. The overhead of a reactive invocation is on the order of a subroutine call rather than a system call.

In order to satisfy real-time response guarantees, the reactive tasking system creates a certain number of RTTasks and maps these to “contracted” real-time reactive responses. These tasks, called RTReactiveResponse, are given priorities in the POSIX SCHED_FIFO domain with the fastest response, typically 1mS, being highest priority. Lower priorities for the lesser guaranteed responses are monotonically less down to 1024mS response at the lowest of the real-time priorities. This monotonic prioritization is borrowed from rate-monotonic scheduling (RMS). Linux thread preemption

with the O(1) Linus scheduler allows higher priority reactive tasks to preempt lower priority reactive tasks. It should be noted that there is no “guarantee” that all the work required to run within 1mS can be run on a specific system. Despite our best efforts, there is no magic trick. However, timing information, as well as fatal and nonfatal errors, are logged when the system fails to meet its real-time requirements. It is intended that detailed low-latency logs will be sufficient to analyze the failure, and with appropriate understanding and redesign, it will be possible to meet the real-time contracts.

ReactiveTask (RTReactiveTask)

```
typedef struct {
    RTMagic magic;
    RTEventQueue eq; /* my bound event
queue */
    RTEvent *evp; /* event
description */
} RTReactiveTask;
```

An RTReactiveTask is a bundling of an RTTask’s RTEventQueue and an RTEvent. The purpose of the RTReactiveTask is to create a persistent binding of the EQ and the event, such that event instances are evanescent but can be sourced repeatedly by the RTReactiveTask. RTEvent is an opaque type which contains a unit of work. RTEvents are described more fully below. Order of execution of enqueued events within a reactive task is strictly FIFO. Thus objects acted upon by ReactiveTasks known to be running on the same ReactiveResponse may be designed to be lock-less because there is no preemption within the non-blocking processing of a single RTEvent.

Reference Counted Objects (RTRefCnt_Object)

```
typedef struct {
    RTMagic magic; /* our magic */
    atomic_t rc; /* reference count
*/
```

```

    void (*destructor)(void *); /*
destructor to call */
} PrivRefCount;

typedef void RRefCount_Object; /*
return void * pointer to generic
object */

```

One of the more challenging aspects to multi-threaded event driven programming is storage management. If an object's user frees the memory, another user may have a difficult-to-debug "use-after-free" problem. Alternately, failure to correctly manage storage can result in painful to debug memory leaks. Coordination between caller and callee, object source and sink, is ad-hoc and creates an undesirable coupling between object sources and sinks that require shared knowledge of the object lifecycle at design time.

The Chronolytix answer to this dilemma is to use reference counted objects to "automatically" manage object lifecycles for shared transitory objects. Reference counted objects (RRefCount_Object) are a container to a generic fixed sized object. A reference is acquired at construction. Additional references are acquired by calling RRefCount_Acquire(). Object references are released with RRefCount_Release(). Upon release of the last reference an optional destructor function supplied at construction time is called, and then the storage is returned to the system.

The implementation of the reference counted object is as a container of the object. The container has an identifying type and a reference counter. Some mechanism for obtaining the address of the container from the object address is assumed. Upon construction and each acquire() operation, the reference counter of the object container is atomically incremented. (That is, using interlocked machine instructions, the content of the memory is read-modified-written such that no other memory operations may interleave its accesses.) Similarly at release, the reference counter is atomically

decremented and tested for zero. If the reference count is now zero, the destruction of the object occurs. The use of machine atomic operations for the implementation, as well as inline compilation of the acquire and release functions, make reference counting objects extremely lightweight (low 10s of instructions) and lock-free as opposed to a heavy-weight mutex-based object locking strategy or a time/space expensive deep copy scheme. For examples of using reference counted objects for managed storage see the Timers and Registers in the following sections.

Timers (RTTimer)

```

typedef struct {
    RTMagic magic; /* magic number */
    RTClockCount due; /* ticks when
event made ready */
    int timer_list_index; /* index to
my list */
    RTReactiveTask *rtp; /* task to
activate */
    struct timer *next; /* next in
list */
} TimerList;

```

Timers are central to the design of applications likely to be run on Chronolytix. This is due in part to the idiom of implementing state machines with timeouts for real-time systems in general, and the fact the reactive task's lack of stack-context requires a reactive task be invoked "later" – presumably by a timer. In order to make timers fast and nearly O(1) for the common cases, a decision was made not to use the POSIX high resolution timer patch(es) for Linux. Since many thousand timers may be active, it was decided that a user-space solution for timers would allow an efficient and "Chronolytix"-aware implementation. The time base used is the real-time clock interrupt and RTC driver. The RTC driver allows a process to be woken every RTC periodic interrupt interval, which the Chronolytix system sets at 1024Hz.

As is the convention within Chronolytix, the `RTReactiveTask` passed to arm the timer is owned by the caller. The `RTTimer` will clone it (using `RTReactiveTask_Clone()`) before returning to the caller. To be explicit, the Chronolytix convention maintains all objects passed to arm a timer are owned by the caller upon return from the arm call. Arming of a timer returns a `RTTimedId` which is used to cancel the timer. So that timers are multi-thread safe and to reduce the number of race-conditions inherent in timers (ie., use after free, etc.), `RTTimers` are implemented as reference counted objects. There are two primary references: the armed timer and the `RTTimedId` returned to the caller. If the timer expires, the invoked reactive task should ensure the release of any held reference (since it is unlikely that canceling an already expired timer would be useful). Similarly, if the timer arming caller has no interest in canceling the timer, a release of the `RTTimedId` immediately will allow the timer to be automatically freed when it expires.

Events (RTEvent)

```
typedef struct event {
    RTMagic magic; /* magic number */
    RTClockCount  enqueued; /* ticks
when event enqueued */
    RTClockCount  stamped; /* ticks
when event time stamped */
    char *name; /* Reference counted
name of this task (or method) */
    RTEvent_fp fp; /* function to run
with opaque args */
    void *this; /* reference counted
object */
    void *arg; /* argument */
    struct event *next; /* next in
list */
} RTEventPrivate;
```

,

As is the Chronolytix convention, `RTEvents` are owned by the caller. That is, when a call with

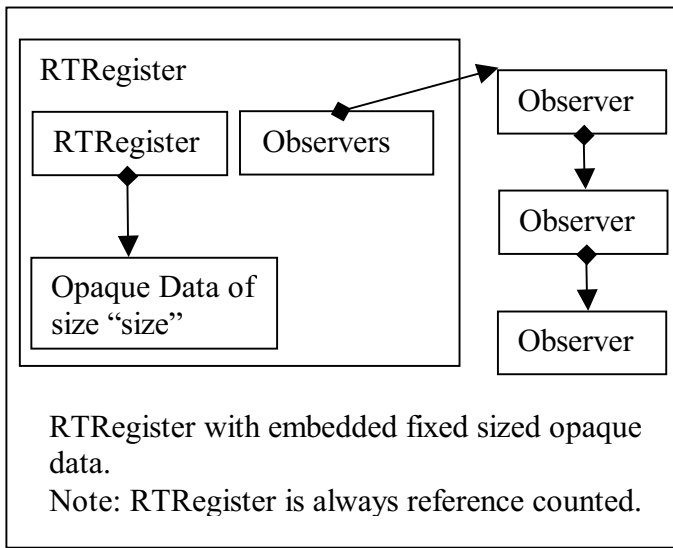
an `RTEvent` argument completes, the lifecycle of the `RTEvent` is the caller's. The callee will have made use of the `RTEvent` (or copied it with `RTEvent_Clone()`) prior to returning control to the caller. The work unit has a name which is copied into a reference counted string in the event at construction time. Subsequent event clones share this string and its lifecycle ends when there are no more references. It is necessary that this shared name be reference counted because events are unique and evanescent and thus not reference counted. An `RTEvent` also has a function to be called and two arguments. The event mechanism will handle storage management issues for the caller/callee if the pointer argument is in fact a reference counted type. If an event argument is a regular, non-reference counted pointer, storage management is the responsibility of the caller/callee. It is highly recommended to use the reference counted arguments unless the object is "permanent" – ie., non-racy. The first argument is a `void *` "object" pointer -- *this*. The type of the object should be "known" by the event function. In the case of the reference counted argument, the event function holds the reference for the *this* object for the duration of the event call. The reference count is decremented by the event mechanism upon return from the event function. Thus, storage management of the event object is "automatic", "simple" and "leak-free". The second argument to an event function is a `void *` argument. This argument can be any valid pointer or `NULL`. The "valid pointer" constraint is placed on this argument by the storage management scheme as this will cause the reference counting code to de-reference the "5" and crash the program. The requirement that a `void *` argument be a valid pointer is thought to be reasonable cost for the automatic storage management property of the `RTEvent` argument.

General Purpose Register (RTRegister)

```

typedef struct {
    RTMagic magic;
    const char *name;
    pthread_mutex_t mutex;
    size_t size; /* size of object */
    struct ob_listener *listeners; /* list of observers */
    RTRefCount_Destructor destructor; /* for the opaque value data */
    /*
    ** value is the last element in RTRegister so that value data
    ** can follow immediately after it. The RTRegister and the
    ** value data are wrapped in a RefCnt_Object.
    */
    RTRegister_Data *value; /* pointer to object */
} RTRegisterPrivate;

```



The RTRegister is a named abstraction for a generic hardware register. The RTRegister is a reference counted object. The size of the register data is determined at construction time. There is no constraint from Chronolytix on the content of the register. The semantics of the binary object contained in the RTRegister is entirely the responsibility code that uses it. The possibility exists that the “register” is, in fact, a data structure whose value may be updated and used by its clients. If the size of the data is less-than-or-equal to the sizeof(long), congruence and assignment use C language operators == and =, respectively. Otherwise, RTRegister uses memcmp(3) and memcpy(3) to test for congruence and to do the value data assignment, respectively.

However, the true function of the register is to allow clients to observe a register – ie., to “register” for notification of register updates. That is, whenever a register is updated (ie., written to with a change in value), a unique event will be invoked for each observing reactive task. Each event will have a reference counted “snapshot” called an RTRegisterUpdate object passed as its void *argument. It is often correct for the reactive event to resample the register value to make sure that action taken is correct at the current instant since the state of the register may have changed since this event was enqueued. Since the register update object is reference counted, the event mechanism itself manages the storage. The event function owns the reference for its duration and then the storage is automatically freed (upon the presumably “last” reference release).

Persistent Register (RTPersistentRegister)

Chronolytix uses the building block of the RTRegister to build a non-volatile storage class. For this class, using the Linux mmap and msync memory mapped files to contain a CRC checksummed named register. This non-volatile

register data is loaded by this class into the RTRegister name-space at initialization time. By being an RTRegister_Observe() client on the register, all data value updates are sync’ed to the backing store “automatically”. By having the the update observer run as a RTReactiveResponse, ordered writes are maintained “reactively”.

Conditioned Observer (RTConditionedObserver)

Chronolytix uses the building block of the RTRegister and RTTimer to construct a time conditioned observer. It is common in real-world phenomena that there is a natural frequency for physical events. For example, a door may be physically opened and closed only twice per second. However, the switch/digital I/O subsystem which detects the door-open and door-closed conditions may have a natural frequency in the microsecond range. It may detect spurious transitions caused by vibrations in the door and its sensor as the door is opening or closing. Software handling the door-open/door-closed condition would like to have a physically possible notification with the “bouncy” switch phenomenon removed. The RTConditionedObserver object observes a register and uses application supplied “make-response” and “break-response” to rate-limit assertion of the “make” and “break” reactive events. For example, the detection of a power supply failure is typically more important to the correct operation and safety of a system than is the eventual restoration of power – ie., the “break” event may have a low latency response, and the “make” response, being sufficiently large to bound the maximum frequency of the application model of the system (ie., approximate the natural frequency of the “real” system).

Although designed with “make”/”break” switch like events in mind, the rate limited bi-static with hysteresis model should work well with the conditioned assertion of other types of multi-static phenomena as well.

Conclusion

Chronolytix supplies the framework described above for building scalable deadline scheduled reactive real-time systems under an enhanced

LinuxOS/GlibC environment. Object oriented storage management, runtime type validation and a schema for extensible “building blocks” allow developers to quickly build application specific content using pre-tested components. The Chronolytix system should allow real-time application developers to build the app – not yet-another first-time system.

Author

David F. Carlson has been designing, writing and debugging real-time system for over twenty years. He has also been a founder of three real-time software companies, each one more challenging, rewarding and fun than the previous. In his spare time he hacks Linux, sails his Freedom21, and fixes things.