libImpUtil & libROWS description of design & functionality

Andrew Hunter
<ahunter@impulse.org.uk>

September 6, 1999

Abstract

This document describes the details of the design & implementation of some of ROWS & ImpUtil's important features.

Contents

Chapter 1

Introduction

> A bug in the code is worth two in the documentation.
> – fortune

ROWS is a widget set. Its primary purpose is to provide a set of tools for building graphical applications. It is designed to be portable, multithreaded, reasonably small and easy to use. This doucment will describe the ways in which ROWS meets, or does not meet those aims.

ROWS itself is split into two libraries - libImpUtil, which provides architecture-independant utility functions, and libROWS, which provides both the architecture-independant and architecture-dependant graphical functions. At some future point, the architecture-dependant code (which is already in seperate source files) will be placed in a library of its own.

1.1 Language choice

ROWS is written primarily in (ANSI) C, for the following reasons: C is a 'lowest common denominator' among languages - interfaces to other languages (such as Perl) are easiest to write in C; C is fast; C is portable - almost all systems have an ANSI C compiler[1]. The disadvantage of using C is that it is not designed to operate in the object-orientated manner of a widget set such as ROWS - C++ would sometimes seem a better choice, although there are some confusing problems associated with using it: compilers obeying the standards are hard to find[2]; compiling thread-safe C++ programs can be confusing (for example, the default configuration of egcs is not thread-safe, and I don't think the common gcc 2.7.2 can be compiled as thread-safe); it is hard to verify a C++ configuration with a configure script (just look at the glibc 2.1 configure script for instance - the only way it finds a broken C++ compiler is to explicitly check for the version).

Recommended kit for compiling ROWS is glibc 2.0.7 or above, with POSIX thread support (failing that, a POSIX 1b compliant C library will do the trick), GNU gcc or egcs (egcs 1.1.2 for preference), GNU make 3.75 or 3.77 (your milage may vary with 3.76 or some other manufacturors make) and X11R6.3 or above. If your system does not support native threads, GNU pth will provide some useful thread functionality - currently native thread support is there for POSIX threads, Sun threads[3] and GNU pth. If you have another system with its own thread library, writing support for that system into impulse_thread would be much appreciated. Usually, POSIX threads is a fall-back over proper native threads, as they can be slower than native threads or GNU pth[4].

---

[1] or at least a K&R compiler, and probably GCC, too

[2] although egcs (now gcc...) is getting quite good

[3] well, not as I write this. Support will be there when this is turned into a postscript file, I hope

[4] GNU pth is post-operative in nature and can be (much) faster than pre-emptive threads in situations that don't parallelise well. Impulse's event system is one such example...

Chapter 2

libImpUtil

2.1 Design goals and function

libImpUtil became an offshoot from ROWS proper a few weeks after development began, as during that development, the widget set had obtained a lot of functions which could be considered to have uses other than in the development of graphical applications. Most of these functions are POSIX.1 compliant and should have a high degree of portability to other architechtures with help from the autoconf script.

2.2 Sparse arrays

Confusingly, provided by impulse_hash[1]. This provides an array with a notional size of 2^{32} entries, although it actually allocates much less space. The current design is most efficient with the allocated space if the values of the entries tend to be clustered into groups. A sparse array is thread-safe, although note that the data contained within may not be. The data itself is a `void *`, and may contain anything that can be represented by that type - note that it is your responsibility to free the data if it is deleted, and the value of an uninitialised position is NULL - you can reset a position to this value if it becomes unused.

2.2.1 Data structure

Impulse sparse arrays are designed with access speed and reasonable memory conservation in mind. Note that they use a lot more space than the equivalent flat array, as there is a little overhead associated with each item. The current design ensures that data access time is not a function of array size in any sufficiently populated arrays (there are population patterns that result in a low memory footprint and faster access time, although the mutex locking is probably the slowest part of any access, no matter what array size you have).

The structure of the array is defined by the constant `IMPULSE_HASH_GRANULARITY`. This must divide 32 with no remainder, and is the size of a subkey. When it comes to finding a value in the array, the key is divided into several subkeys ($k_1 \ldots k_n$), each of `IMPULSE_HASH_GRANULARITY` bits[2] - together, these must uniquely identify the original key. The data structure itself is an n-ary tree, where the key values define which branch to take at each stage. When a leaf is reached, it is checked to see if it is the leaf containing the given key - if not, the key's value is empty (NULL); if so, the key's value is stored in that node. When a key has a value set, nodes are created in the tree until that key has a unique node (so an array with only one item in it might only have the root node).

2.2.2 Implementation

Obviously, the choice of keys and their ordering can have a big effect on the storage efficiency of this design of sparse array. The current implementation uses a fairly simple method to split the key into subkeys: from

[1] when I actually write hash functions, this will change

[2] for a granularity of m, $\frac{m}{2}$ subkeys are required

a key consisting of bits $b_0 b_1 b_2 \ldots b_x$ and a value of `IMPULSE_HASH_GRANULARITY` of m, subkey k_y consists of bits $b_{m(y-1)} \ldots b_{my-1}$. The subkeys representing the least significant bits of the key are used at the top level of the tree - for low values of `IMPULSE_HASH_GRANULARITY` (around 1-4) this provides best storage efficiency. For higher values of granularity, the most significant bits should be used instead.

2.2.3 Usage

The data type for a sparse array is `impulse_sparse`. This should be treated as an opaque data type. A sparse array is initialised with `impulse_sparse_create`(). Items can be stored in the array using `impulse_sparse_add`(), and retrieved using `impulse_sparse_recover`(). When finished with, the memory used by the sparse array can be freed using `impulse_sparse_free`().

Functions

- impulse_sparse *impulse_sparse_create(void)

  Creates an empty sparse array. All values are initialised to NULL.

- void impulse_sparse_add(impulse_sparse_array *array, uint32 key, void *data)

  Sets an item in a sparse array. The item number is given by `key`, and it is set to the value of `data`.

- void *impulse_sparse_recover(impulse_sparse *array, uint32 key)

  Recovers the value stored at location `key`.

- void impulse_sparse_free(impulse_sparse *array)

  Frees the memory used to store the sparse array. Note that this does not free the data pointed to by the locations in the array.

2.3 Threads

Impulse provides a standardised interface to the operating system's threading system. Using impulse_thread to perform multithreading in your application helps ensure that your application is portable to all other systems that libImpUtil runs on[3], and also helps ensure that your application uses a threading scheme that is compatible with Xlib.

2.3.1 Facilities provided

impulse_thread provides facilities for creating and exiting from threads; getting the current thread ID; comparing the thread ID to that of another thread; yielding to other threads and sleeping for a certain time[4]. It also provides mutexes (simple semaphores for solving the mutual exclusion problem), condition variables (basic inter-thread communication), message queues (for sending inter-thread messages), recursive mutexes (which can be locked more than once by one thread), read/write locks (which solve the readers & writers problem) and a global mutex to help in program initialisation.

It does not provide initialisers for mutex, rwlocks, etc because not all thread libraries provide a sensible way of defining these. However, the global mutex can be used to ensure that these structures are initialised correctly. See the initialisation of event_mutex in impulse_event.c for an example of how this is achieved.

[3] for example, some versions of LinuxThreads do not support POSIX 1003.1b read/write locks. impulse_thread contains workarounds for this.

[4] note: sleep() is not thread-safe on all systems (not being required to be by POSIX), notably SunOS/Solaris. impulse_thread_nap should always be used in preference

2.3.2 Creating an interface to a new thread library

To create an interface to a new thread library, you need to add checks and options to configure.in for this library. You then need to edit impulse_setup.h.in appropriately so that there is a `THR_<FOO>` (where `<FOO>` is your thread library's identifier) definition. Next, you need to add suitable data types for `impulse_thread`, `impulse_mutex`, etc in `impulse_thread.h`. The typedefs themselves must be pointer types, as they can be set to NULL to indicate that no value is stored there. Usually, these types are pointers to structs such as _impulse_mutex, which contain the actual (architecture-specific) data. After this, you want to implement the functions (whose definitions are constant) in impulse_thread.c. Finally, you need to check the rest of libImpLib for other thread-specific references (easily done by grepping for pthread_). The most significant of these is impulse_event, which needs to know the best way to interrupt a select() call running in one thread from another thread[5]. If your thread library does not directly implement a function, you will need to emulate it using the library's other functions - particularily note the difference between mutexes and general semaphores.

2.4 Events

Impulse provides a general-purpose event handling routine, based on the assumption that there are 3 types of event that are of interest to your program: timeouts, filedescriptor events and user-defined ('system') events. User-defined events must be triggered by another piece of code, but filedescriptor events and timeouts are triggered by the event library itself. The library is designed to distribute events to multiple threads, as well as providing a transparent interface for single-threaded application, and is intelligent enough not to require a whole thread to itself.

2.4.1 Facilities provided

Two main facilities are provided: event collection & generation and event callback functions. Events are collected using `impulse_event_get_next_predicate()`, from any thread, and can be generated by listening on a filedescriptor, setting an alarm or directly triggering them with `impulse_event_trigger()`. Event callbacks can be set with `impulse_event_register()`, and are called by events collected using `impulse_event_handle()`.

2.4.2 Multithreaded semantics

When an event is generated, it must be delivered to a listening thread (or stored for later collection). In order to ensure that threads do not get events they cannot handle, `impulse_event_get_next_predicate()` is used to ensure that only interesting events are delivered to this thread. It has a priority value, so if several threads are interested in a given event, the thread with the lowest value is chosen (where several threads have the same priority, the choice is nondeterministic). When a file descriptor or alarm event is delivered, the listener is deregistered to prevent large amounts of unhandled events being queued. There is also an `impulse_event_get_next()` that returns all events as they are generated (it has a priority of 256) - this is a bad thing to call as, due to the nondeterministic nature of the event system, you could deprive other threads of the events they needed to deal with - it is there for use by programs which are single-threaded in nature. `impulse_event_handle()` will collect some of the events that it has handlers registered for, with a priority of 128[6] - note that `impulse_event_get_next_predicate()` might actually make the call to the handler routine instead of `impulse_event_handle()`.

Note that alarm and filedescriptor events must be deregistered immediately after occuring in order to ensure sane behaviour. The reason for this is that select() or poll() will return for a filedescriptor for as long as it has data waiting. This means that if there are several threads dealing with events and one of them picks up a filedescriptor event, passing control to another one while it deals with the situation, if the filedescriptor is not deregistered, the second thread may pick up on the same event. Therefore event registration functions are

[5] Usually, this is done by sending a signal, causing select() to EINTR, although GNU pth, for example, has an event system that provides a neater way of doing it...

[6] here is an illustration of the problem with `impulse_event_get_next()` - although `impulse_event_handle()` uses a priority of 128, it is not waiting while the function is executing. This means that `impulse_event_get_next()` could jump in and get an event that occured during the execution of the handler function. `impulse_event_handle()` then would not get this event to be handled.

'one-shot', to avoid the situation where several threads are all trying to deal with the same event. Convienience functions are provided to give the multi-shot behaviour found in single-threaded systems.

There are also two hook functions used, called just before and just after select() is invoked. These are intended for use by a widget set based on libImpUtil (that would be libROWS, then). An application based purely on libImpUtil may use them for its own purposes, however: they are designed for use in situations where buffering may occur, such as the X event queue - any X function may collect events for the queue, and under some circumstances (such as when synchronous behaviour is on), the file descriptor can have no data waiting, but there are events pending. This makes the functions useful where a buffering IO library is used (explicit support for buffering IO libraries may be forthcoming if there is a demand. Buffering by the operating system or C library is not a problem). A hook function may return an impulse_event structure to indicate a generated event, overriding the select() command.

2.4.3 How it works

`impulse_event_get_next_predicate`() can be called from any thread because it has two behaviours. When at least one thread has called `impulse_event_get_next_predicate`(), one (and only one) thread is actually generating and collecting events and the others are waiting for the collecting thread to signal them. There are two sorts of signal - one is 'passing the buck', when the collecting thread has matched its predicate and is about to return and wants another thread to take over its role, and the other is 'event occured', when the collecting thread has matched the recepiant's predicate and is passing the details to it. The upshot of this is that all predicate functions are executed by the collecting thread, which is not necessarily the thread that called `impulse_event_get_next_predicate`() and is not necessarily the same thread as called the predicate function previously - this can be important if thread-specific data is in use. An interesting note is that a thread might pass the buck to another thread before the thread that passed the buck to it has finished its job, or it might be that an `impulse_get_next_predicate()` is starting up at the time the buck is being passed. In an old implementation, I hadn't considered these two situations and occasionally a thread would jump in and take control of collecting events before the intended recipient of the buck had taken control. The variable `buck_passing` solves this race condition.

The actual event collection is done by waiting on `select`(). The obvious problem with this is what to do if another thread registers an interest in an event while waiting or a user event is triggered. The obvious solution is to interrupt the `select`() call, usually by sending a signal to the collecting thread. Signals are not supported by `impulse_thread`, so this is implemented in a way specific to the thread library being used[7].

GNU pth

It seems that GNU pth has a few bugs associated with select(), specifically that timeouts do not always work as expected. In order to work around this, libImpUtil uses pth_select_ev, and implements timeouts using GNU pth events. Note that the IO semantics of GNU pth mean that a single blocking thread will block the entire process - there are some work arounds for this, however (pth_read, for example). The best solution is to use impulse_event for IO selecting and doing a nonblocking read wherever required - this will ensure good performance with all threading systems that can be supported by libImpUtil[8].

[7] Signals are not supported by impulse_thread, as I view their function to be slightly low-level. If there is an interest in signals, they will be reported as events and not through the normal asynchronous channels. Also note that SIGIO has been chosen as the signal, as LinuxThreads uses both SIGUSR1 & SIGUSR2 for its own nefarious purposes.

[8] Windowing systems may require a different approach. X can be made to cooperate by judicious use of XPending() and XNextEvent()