Experience Report: Building a Real Time OS Kernel in Smalltalk

Tim Rowledge, Interval Research Corporation. (Alas, Interval Research is no more)
tim@sumeru.stanford.edu

The team at Interval Research was given the task of making an entire OS for a media handling server/network controller where it would be 'Smalltalk all the way down'. Aside from a tiny library of context switching routines and machine startup procedures, the aim was to make it possible to write everything in Smalltalk; all the way up from device drivers to user applications.

As so often happens, the project was cancelled before we could truly claim to have completed it, but we did manage to build a good solid alpha system incorporating most of our technical aims.

The Aims

- There was to be no OS but Smalltalk. We allowed some lattitude in this target, at least to start with, since it seemed foolish to start off by writing a TCP/IP stack or ANSI C library. Our target hardware had a clib/monitor that included all the machine bootup code and it would have been very wasteful to duplicate this.
- Worst case interrupt response time of 100 microSeconds from raise to start of response. Interrupt code had to be either a 'native' function for items like the TCP/IP stack, or Smalltalk code in the general case.
- Live in an always-on, no downtime expected, end-user world with no tech- support or sysadmin staff to fix things.
- Survive with fairly limited resources:- a 200mips ARM cpu and 32Mbyte ram might not seem terribly limited, but it is quite different to a 2GHz Pc with 512Mb and virtual memory and regularly rebooted. This requirement was primarily a cost related limit; the device is a consumer item, not an office PC or similar.

Porting to custom hardware

Building a system with only a minimal OS is nothing particularly new for Smalltalk (c.f. the Alto and related systems, the Active Book, the Mitsubishi port of Squeak etc) and we used the normal techniques in this project. The target system had a perfectly acceptable ANSI C library and along with the Advanced Risc Machines Inc. supplied loader and monitor tools we had merely to do a port of Squeak. There were the usual sort of bugs and frustrations concommitant with custom hardware but this basic phase went very well. The chief problem was the time spent loadinga multi-megabyte image file via a slow serial port, which was solved by making it possible to laod the image into a simple flash memory filing system.

Making an RTOS with Smalltalk

The really interesting work was making a low latency RTOS out of Smalltalk. We were required to make it possible for interrupt handlers and device drivers to be written in Smalltalk or 'native code' (by which we understood anything written in any language that compiled to machine object code) or both. The latency had to be low enough to allow handling of media streams on the MediaWire network (see <http://www.aviodigital.com>) and yet overall performance had to remain snappy enough to run user programs handling email. web browsing, TV control, household security, phone systems and so on.

There are a number of activities within a normal Smalltalk system that stand in the way of such a fast interrupt response

- garbage collection
- large BitBLTs
- initializing large array objects
- complex returns
- bytecode dispatching
- any other long running primitives added for application use.

The general problem is that these long running actions have to be atomic in order for the VM to be in a safe state when an interupt is dealt with or to put it another way, you cannot switch processes except when a bytecode has just completed.

There are four forms of interrupt situation to consider:-

- ST interrupt handler, ST process currently executing
- ST interrupt handler, native code process executing
- native interrupt handler, ST process executing
- native interrupt handler, native process executing.

The difficult case is the first one; we must reach a safe state before process swapping in order to keep running. A native handler can interrupt at any time since it is not allowed to use any Smalltalk objects in Object space. The ProcessorScheduler class and Process class were replaced with ones that could schedule a new form of process which could involve either Smalltalk or native code.

Slang

One of the interesting new components of Squeak is the use of a simple Smalltalk subset to define the VM. We came to call this subset 'slang' both because it is a degenerate language variant and because it seemed to fit 'System Language'. Slang is executable Smalltalk despite the stilted style and being able to completely simluate a new VM design was a major win in many parts of our work.

We felt sure that the overall performance could be improved if we did not need to work via a C compiler to produce our VMs. Along with the wish to be able to dynamically extend the VM at runtime, C is not a particularly elegant way to express the control flow of a threaded VM. This lead us to produce a direct Slang to ARM object code compiler that allowed us to make some improvements in our expression of the VM code; in particular the support for the dispatch mechanism changes mentioned later.

With the Slang compiler included in the running system, it was expected that we would be able to add new primitive facilities or replace bugged ones on the fly without any need to restart the system. New devices added to the MediaWire network would be one example of what this could support.

Garbage Collection

Most gc systems involve the potential for quite long periods where interrupting the gc would be dangerous. The standard gc in Squeak is a non-interruptable stop, copy, and resume mode and it can involve moving a large number of objects and scanning many more. Attempting to handle an interrupt during this process would be catastrophic since the object headers are not in canonical form and the object data may be in two different spaces!. After looking into a number of possible gc techniques we concluded that Baker's treadmill had the best chance of satisfying our needs.

We based our work on papers by *Henry G. Baker, Jr "The Treadmill: Real-time Garbage Collection Without Motion Sickness" OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems* (which also appears in SIGPLAN Notices 27(3):66-70 March 1992) and the University of Texas, Austin, disseration proposal of Mark S. Johnstone *"Non-Compacting Memory Allocation and Real-Time Garbage Collection"].* See also the book *"Garbage Collection: algorithms for automatic dynamic memory management", John Wiley & Sons 1996.*

Most garbage collectors used in Smalltalk systems are stop-and copy collectors. These collectors suspend all Smalltalk execution, find the live objects, and copy those live objects to another space. Any objects which were not copied are garbage and their space is freed. Both the marking and copying of objects must be completed before execution resumes. Baker realized that all the copying was a costly way (at least in real time systems) to manage two sets: the live objects and the garbage objects. He proposed using a doubly linked list to obviate the need for copying. Live objects were snapped onto one list, objects being marked were snapped onto a second list, newly allocated objects were snapped onto a third, and free objects onto a fourth. Since the time to remove or add an object to a doubly-linked list is both bounded and almost certainly shorter than a copy, one can specify the maximum time during which the system cannot be interrupted.

Independent of the gc algorithm itself, one must provide a mechanism for allocating and deallocating space for each object. There are a number of engineering tradeoffs: e.g., overhead per object, the ability to merge freed space with adjacent free space, and whether the allocation of large objects causes difficuties. Initially, we used a scheme with "pages" of memory reserved for objects of the same size. However, the code became quite complex and we later switch to using Knuth's "buddy-system". Although the buddy-system wastes some space, the simplicity and upper bounds on execution time are quite attractive.

Our final design was a variant of Baker's treadmill (using Knuth's buddy-system for allocation/deallocation) which demonstrated the ability to run for many hours without losing objects, sandbarring excessively or crashing.

BitBLT

Although many BitBLTs are small operations, there are plenty of cases where a ~600*500 area of a screen might need updating. Given that it will take about a dozen cycles to handle the fetch, merge, mask, shift, write for each pixel, we quickly discover that 0.018sec is unnacceptable; it is after all 1800 minimum-latency periods. Even a 40pixel square icon will take up the 100microSecond allowed response time and allow no time to do the context switching.
There are two ways we found to reduce the problem:

- up to the point where pixels are written, it is possible to simply abandon the BLT and back out of the primitive in such a way as to allow a retry later.
- at the end of each line, it is possible to exit the BLT so long as the BitBLT parameters are written back with the corected start Y value. The ST code can then continue later. Obviously it is possible for this to result in incorrect displaying in extreme cases, but during our experiments we noticed no actual problems.

Large array init

Object allocation is normally very quick, but there are occasions where large arrays are instantiated; Bitmaps are one important case. The bulk of the time is spent nilling out the slots either with nil or 0. We concluded that it is feasible to split the object creation into three phases

- allocate memory, set headers to satisfy gc and other users that it is a legitimate object that needs no scanning
- fill the array with nil/0
- alter headers to suit the true class of the object

The first and last operations are quick and the middle can be interrupted without problems, with only a progress value being stored in the beginning of the object.

Returns

Method returns are quite simple and quick, but out of scope block returns can involve scanning, marking and otherwise touching many contexts. To help reduce this problem we developed an extended Context format which allowed the machine address of various important objects to be cached. Since these values are derived at each send it seems a pity to re-derive them for each return, especially when many returns are immediately followed by another return. Of course, normally these cached addresses quickly become invalid if any garbage collection that moves objects is invoked. Initially, we designed a simple timestamp mechanism so that a return would check the returned to context's timestamp against a global value set by any object moving gc. Once the non-moving gc was chosen, this became redundant. In order to allow the addresses to be kept in 'raw' form, the class Context was changed to a non-pointer class. Since much system code still expects to be able to access the original instance variables, a small number of accessor primitives were added. Obviously, any and all methods that refered to the contexts had to be checked and often rewritten to handle the changes.

Bytecode Dispatch

As mentioned previously, he only 'safe' time to swap processes is between bytecodes or at logically equivalent times. The above changes to BitBLT and array initialisation added a number of safepoints but there is still a problem in finding out when a swap has been requested. The BlueBook VM design incorporated an event check at every bytecode dispatch, something that added a high cost for relatively little value. Most later VMs changed to other mechanisms, such as the VisualWorks approach of checking during sends, primitive returns and backward branches. Squeak limits checking to backward long jumps and actual method activations.
We wanted to make it possible to swap at any of the safe points with minimal overhead and designed two mechanisms to support this:

- change the dispatch loop so that the base address of the bytecode routine address table is held in a register. When an interrupt is raised, that register is changed to point to another table with all addresses pointing to a check routine. This means we can avoid the cost of testing a flag at every bytecode fetch without losing the respone time benefit such a test would offer.

- add an extra pseudo-bytecode that simply returns false in the normal case but true when the aforementioned register has been altered. This routine is called by the BitBLT and array filler code described above in order to find out if an abort is called for.

Conclusion

The BitBLT, garbage collector, mixed format process scheduling and slang compiler were all developed to a stage where they demonstrated solid acceptable performance and reliability. In particular, we were able to generate a running VM with the slang compiler that demonstrated about 20% better performance than the C compiler version. The dispatcher changes and context redesign were simlulated successfully but never integrated into a full VM. The array init changes remain just a design description.

Credits

The above work was done by a team that worked together extremely well, consisting of
Don Charnley
Paul McCullough
Alan Purdy
Tim Rowledge
Frank Zdybel jr.
We all worked on most parts of the system at various times, but the memory manager was predominently by Paul, the ProcessScheduler by Frank, the Slang Compiler by Alan and the BitBLT changes by Don.
At various time we were ably assisted by Craig Latta, Mike Penk, Kerry Lynn and Phil McBride.