Adding native thread support to SBCL
- or -
n things every programmer should know about signal handling

Daniel Barlow
metacircles Ltd
dan@metacircles.com

July 14, 2003

Abstract

Threads (lightweight processes sharing a common address space) are now a near-standard technique used in many large applications, and supported - in varying ways and to different extents - in many languages and most operating systems. We added thread support to the Steel Bank Common Lisp environment on Linux platforms, using the clone() syscall : this presents interesting challenges

- Lisp's introspective abilities and runtime environment requires OS support for things like writing to our own instruction stream, setting breakpoints on ourself, and finding the register contents and faulting address in a SIGSEGV handler. These features are often provided, more seldom tested, and hardly ever documented.
- The primary interface to the Linux kernel is through glibc, and defined in terms of the glibc header files. Accessing this functionality from languages other than C is sometimes harder than it needs to be.
- The de-facto standard for CL multithreaded programming was based on special-purpose Lisp hardware (where, for example, user hooks could be installed directly into the scheduler) and is less suited to today's mainstream computing systems.

Topics include thread-local storage, garbage collection, mmap, breakpoints and ptrace, signal handling, floating point, terminal handling, dynamic linking, and atomic operations. We conclude with a look at some tools (cparse, SWIG, FFIGEN) to help in this area, and some recommendations for kernel and library authors about how they can make life easier for users of non-C languages.

1 The challenge

Traditionally Lisp and Unix were very different cultures. Unix vendors of the 80s produced workstations that, although expensive by PC standards, actually had a very attractive price/performance ratio. Unix is based around a philosophy of multiple independent processes which communicate by sending

streams of bytes to each other - typically using sockets, files or pipes.

Lisp companies, on the other hand, produced fiendishly expensive single-user workstations with Lisp-based operating systems and hardware support for fast type and bounds checking. Because these systems were single user, and the hardware protected against most kinds of accidental damage, Lisp systems used to allow each task to see all the others: no marshalling overhead, easy object sharing, no memory protection. They called their tasks 'processes', but in today's terms they're more like threads.

Unfortunately for Lisp machine vendors, "fiendishly expensive" and "custom hardware" could mean only one thing. The Lisp hardware customers were mostly the AI companies: the combined effect of Moore's Law making general-purpose hardware a lot faster, and the bottom falling out of the AI market[1], mostly killed them.

Thanks to the the ongoing march of technology, in this decade we can put a fully functional Lisp onto a standard hardware platform, and it's not even the biggest process - Mozilla or OpenOffice.org comfortably exceed the resource requirements for Unix-based Lisps. Stock hardware has the CPU power, and standard OSes such as Linux can – with a small amount of forcing – make good software platforms for a Lisp.

Steel Bank Common Lisp (SBCL) is a native code Common Lisp environment that runs on a number of Unix platforms. It started life as a fork from CMU Common Lisp, which was originally part of the Spice project at CMU - so its lineage (and some of its code) can be traced back for decades. Over the course of the last nine months we added support for Linux kernel threads to SBCL: in this paper we look generally at the places where we have to do unusual things for Unix, but with especial emphasis on features we need for threads.

[1] Comparison with the dot-com bust is left as an exercise for the reader

2 The language

Why Lisp, anyway? If you ask ten different Lisp programmers you'll get ten different answers, but some of the common traits are

- Interactive : like interpreted BASICs of old, Lisp implementations usually start up in a state that expressions can be typed into them to be executed immediately. This makes testing and prototyping much faster. Note that, unlike interpreted BASICs, any self-respecting Lisp also integrates a compiler into this environment.
- Extensible : the parentheses that plague so many Lisp newbies are actually a feature. Because the printed syntax corresponds so closely to the parse trees, macros make it easy to rewrite source forms to extend the syntax of the language in exciting ways. This eases the creation of domain-specific languages, so making bottom-up programming a lot more fun.
- Multiparadigm : flexible syntax has meant that Lisp has often been used by language researchers when experimenting with new language paradigms. Out of the box, Common Lisp has support for programming in imperative, functional, OO style, or some combination of these styles. Nor is it hard to write, for example, a Prolog interpreter in Lisp, if the problem is best expressed in declarative style, or to add support for lazy evaluation.
- Standard : Common Lisp was standardised by ANSI in 1994 (making it the first ANSI stan-

dard object-orientated language, incidentally). The standard was mostly a formalisation of what existing Lisps are doing, so tends to be pragmatic instead of idealistic.

2.1 Multiprocessing in Lisp

The Common Lisp standard does not include a threading interface. Although the Lisp hardware systems of old had Lisp functions to inspect and manipulate multiple tasks, other Lisp implementations of the day were running on Unix or on microcomputers, so didn't usually support threading and would not have voted for a change that required them to. Regardless, the interface has since become something of a de facto standard for Lisps that did support threads, usually using userland threads with a scheduler written in Lisp. However, it makes some assumptions about the thread implementation that make it a poor fit for kernel threads.

For example, **without-interrupts** is a macro that wraps around code that must be executed without interruption. On a Lisp machine, it really does turn interrupts off. On Unix, it masks signals. This means that the scheduler won't run for the duration, so Lisp code often called this to disable multitasking while doing some operation that must be atomic. Needless to say, this is slow and difficult to arrange when the kernel is doing the scheduling - and bad news for scalability anyway.

process-wait is another problematic function: it takes a predicate function as a parameter, and stops the current process until the predicate returns true. With a Lisp-based scheduler this is pretty simple to implement, but in an OS scheduler, we're going to have to call the predicate *every time* we switch to this process, then switch away again if it's false. Although we could reduce the system load by adding a delay, this is still not a sensible use of cycles.

The problem is not that this function exists - after all, sometimes you need to do this - the problem is that there are no other functions in the standard interface to make a thread wait for an event, so people end up writing their own queue implementations on top of this. If we provided - and encouraged people to use - a queue implementation that didn't do all these extra context switches to processes that we should already know aren't ready to run, things would go much more smoothly.

Given these problems, and with the intention of creating an interface that looks a bit more familiar to today's programmers, we mostly ignored the Lispm interface. We're calling our threads threads, not processes. Threads are identified by ids, and we have functions like **make-thread**, **current-thread-id** and **destroy-thread** to manipulate them.

It's likely that a Lispm-like compatibility layer will be added to ease porting from other Lisps, but even in that layer it's unlikely that we'll ever implement **without-interrupts**.

3 The platform

Linux: "because it's there". Our interest is in creating a free Lisp development environment, so a proprietary OS would be a non-starter. The alternative, to implement an OS from scratch, would be a seriously mammoth task, and to implement from scratch with the same attention to performance and scalability as has been lavished on Linux, would probably take as long as Linux did. Why reinvent the wheel when there's a perfectly good antigravity belt there for the taking?

In this section, we look at some of the differences between SBCL and "normal" Unix programs.

3.1 Calling convention

In C, functions accept a fixed number of arguments, each of a fixed type. A limited degree of support for variable numbers of arguments is available with `stdarg`. There is a single return value.

In Lisp, we can (in general) pass any type of object to any function, and additionally we have "keyword", "optional" and "rest" arguments[2], and multiple return values. It's somewhere between difficult and impossible to shoehorn all of this into a C-compatible calling convention, and SBCL doesn't even try. Calls to C functions therefore involve a glue layer which translates from one argument passing convention to the other, so are probably best avoided in time-critical inner loops.

3.2 Preprocessing

Even if we did use C calling convention, we'd still have a problem where parts of a C language API are specified as preprocessor macros. To inline C code into our Lisp functions would involve doing things to gcc that rms would really not be quite happy with. So, we have to write wrapper functions for these macros, usually in C, and suffer the additional function call overhead.

In the particular context of adding thread support, this presents a particular problem for thread-local data access. As we'll see later, Lisp needs a lot of thread-local data, and having to do a call into C on every access is a good deal slower than we'd really like it to be.

[2] An optional argument is declared with a name in the parameter list, which is bound to a default value if unsupplied. A rest argument is a name to which a list of all the unnamed arguments is bound.

3.3 Signal handling and debugging

It is unusual for a Unix program to continue running for very long after receiving SIGSEGV, SIGBUS, or SIGTRAP. It's not unexpected for it to stop after receiving SIGFPE (e.g. on a divide-by-zero error).

However, dropping back to the shell prompt is usually undesirable in an interactive system: post-mortem debugging is generally far less productive than in-process debugging where all the state (including open files, network connections, etc) is available, and besides we'd rather not kill off any of the other threads that are running unrelated jobs. So we really want to catch these signals and run error handlers that can fix the problem and continue.

(Indeed, sometimes we intentionally ask for these signals in normal operation. For example, the garbage collector tracks writes to old pages by mapping them read-only and catching SIGSEGV when they're written to)

To diagnose and recover effectively we need more than just the signal number. We declare all our signal handlers using `sigaction()` with the `SA_SIGACTION` flag. This causes our handler to be called with three arguments: the second argument is a `siginfo` structure, and the third is usually a `ucontext` structure [3] containing a wealth of information including the program counter at the time of signalling, the register contents, and other per-signal information - for example, in the case of a SIGSEGV, the faulting address. Much of this information is used internally in SBCL or passed onto the debugger.

Libraries that install signal handlers are, therefore, bad news. Even if their handlers still work in our

[3] This holds on Solaris, Tru64, FreeBSD, and all Linux architectures we have ports for, other than the SPARC. I don't know why it has to be different

runtime, and even if they then call the previously installed handler instead of bailing out, they rarely do it properly with correct siginfo and ucontext - and almost certainly don't think to check whether we were using `sigaltstack()` to get an alternate signal stack. So, at best our handlers will not know why they're running, and more likely we'll get a core dump with fairly meaningless backtrace, or an infinite loop of segmentation faults.

SBCL users have seen this problem in practice with SDL, for example. It installs a SIGSEGV handler, presumably to free up resources and return the console to text mode if necessary. This is bad, because we use SIGSEGV for our page write protection. When SDL quits it restores the old handler, but it doesn't restore it with the SA_SIGINFO flag, so it *still* loses[4]

3.3.1 Introspection

SBCL includes an interactive compiler - in fact, there is no interpreter; everything that is evaluated is compiled first. When the user asks to evaluate a function, we compile it directly into memory, and then jump to it. This presents cache consistency issues somewhat like those for self-modifying code, though happily far more tractable as we are very unlikely to be executing in the same page as we're modifying. Although I am aware of no standard function to 'flush the icache', it's usually possible somehow - or, on x86, happens automatically.

There are other circumstances in which we write to code pages, that need similar treatment: for example, when setting breakpoints, or during GC when moving functions from one place to another. It should be noted that hitting a breakpoint causes us to receive SIGTRAP in our own process - this works pretty well in practice, but tends to confuse gdb if you're also running under gdb. Ideally it would be able to tell when a SIGTRAP was due to a breakpoint it had inserted and when it should instead be delivered to the process it's attached to.

Libraries that call `exit()` are bad news in any circumstances.

[4]This was determined about a year ago by an SBCL user, and I believe was reported as a bug to SDL maintainers. It may well have been changed since

4 Threading

In this section we look at the specific issues we encountered adding thread support to SBCL.

4.1 Thread creation, destruction, accounting

In Linux 2.4, the POSIX threads interface is implemented by LinuxThreads. Writing to this interface would make us compatible with all platforms that offer POSIX threads, if it were possible. There's a problem, though: in kernel versions up to and including 2.4 (it gets a lot better in 2.6) there's a pretty poor match between the kernel's thread model and the requirements of the POSIX thread standard. This requires LinuxThreads to do some pretty offensive things.

The most important problem is signal handling. POSIX threading has some fairly strict requirements about which thread(s) a signal is delivered to, which in LinuxThreads are implemented by having one thread catch a signal and resignal it to another. As we've seen already, though, resignalling is a no-no when we're using three-argument signal handlers.

We decided instead to use the kernel threads directly. After all, POSIX specifies interfaces for C,

not for high-level languages. So, we call clone() itself, or at least the small C library wrapper for it that allocates a stack for the child.

We're not directly using NPTL. This is partly because we can't, as so much of it is defined as gcc or binutils extensions and we're not using gcc or binutils, and it's partly because POSIX semantics are not a requirement for us. We should emphasise, though, that the work done on the kernel for NPTL *is* important to us: by using kernel threads we benefit directly from the speed and scalability optimisation they do, and when 2.6 is available with new kernel facilities like futexes, we'll use them too.

4.1.1 Garbage Collection

Garbage collection works by periodically pausing the program, then, starting with the 'root set' - the variables that we know are in use, such as the CPU registers and the local variables on the stack - following all the references from those variables to other objects, and copying everything we find into some new area. When we run out of objects, anything left uncopied is evidently no longer needed and can be deallocated.[5]

This is going to be simpler if we know that nothing is changing the data behind our backs, so we have to stop the other threads while it happens. To make life slightly simpler, we dedicate the parent thread of SBCL exclusively to cleaning up after its children: usually it sits in a loop calling `waitpid()` and reaping children. When signalled by a child to perform GC, it uses `ptrace()` to attach and suspend to each of the children, and when they're all safely stopped (we're notified by `waitpid()` for each child) we can do GC. The `PTRACE_PEEKUSR` call is used to get the child's registers to add to the root set.

There is a wrinkle. Sometimes a thread may set its 'pseudo-atomic' flag: this says that it's in the middle of some operation that shouldn't be interrupted by signals or garbage collection - for example, it has allocated but not yet initialised memory for a new object. After we stop the world for GC, we have to check the flag for each thread and resume any that were in pseudo-atomic, letting them run until they reach a safe suspension point.

4.2 Special variables, thread-local storage

Common Lisp has a feature called "special variables". A special is declared using defvar or defparameter, and behaves almost but not quite like a global variable. If we write

```
(defvar *foo* 1)
```

then *foo* is set to 1 in all parts of the program – *except* that we can temporarily rebind it to some other value

```
(let ((*foo* 2))
  (some computation))
```

and during the body of the let form it will temporarily be set to 2. Even if the body calls some other function, *foo* will still be 2 in that other function. When the let form finishes executing it reverts to the value 1. Computer science graduates will recognise this as dynamic scoping; Perl programmers will recognise it as a 'local' variable (as opposed to the more normal 'my' variables).

But what do we do if two threads refer to *foo* at once, and one or both of them has rebound it? If it only has a single value at any one time this could

[5] This is an oversimplification

get really messy: are rebindings in A visible in B? We'd rather they weren't: a change to a bound special variable should affect only the thread it happens in.

So we need some form of thread-local storage, and we need it to be at least reasonably fast. All of memory is the same in both threads, and the only thing that differs between threads is the register contents. We have a few options:

We could dedicate a register to the thread storage base. On an x86 we don't have a lot of registers available, though. Tying a register up permanently for this purpose would make code run more slowly, and implementing this changes would also need quite a lot of work on the compiler (which already has uses for most of them wired in).

Each thread has its own stack: if we can identify the current thread from the stack pointer, we can use that as an index into a table of thread-local values. But we only have registers (`%esp`, `%ebp`) for the bottom of the stack - which keeps moving up and down - and for the current frame. Either we'd have to push the thread base address onto the stack on every function call, or we could make sure that each stack is aligned on, say, a 2Mb boundary, and then find the top of the stack just by masking the stack pointer appropriately. But 2Mb is a lot of stack if we want to run lots and lots of threads.

Or we could use a segment register. Linux uses the 32 bit 'flat' mode of the 386, so segment registers don't see as much use as they did back in the "good old" MSDOS days - usually they're all set to 0, in fact - but they're still available. We use the %fs register, having been warned that %gs would be used by NPTL.

Setting up segment registers in flat mode is a little more complicated than it used to be in real mode. Rather than being just an offset to the memory area we want to access, the %fs register is now a pointer into an entry in a descriptor table, which stores various attributes of the memory area: size, location, permissions etc. This can be a Local Descriptor Table or the Global Descriptor table. Manipulating the values in a descriptor table is a root-only operation, but the kernel people in their infinite foresight have created a `modify_ldt()` system call (mostly for the use of Wine, which needs to fake %fs and %gs convincingly for Windows apps to run) that we can use.

There is a downside to using the LDT: chiefly that it limits us to 8192 entries and hence 8192 threads. As part of the 2.6/NPTL work, Ingo Molnar introduced a user-specified slot in the GDT (again, primarily for Wine to use) which is reloaded per-process in the kernel context switch. When 2.6 is available more widely, we'll probably switch to using this, although it should be noted that we currently have fixed 2Mb stacks, so until we fix that to allow variable stack sizes we can't get nearly that many threads anyway.

The eventual upshot is that we can set up a block of memory for each thread, assign an offset for every variable that gets dynamically bound, then write e.g. %fs:20 to access the thread-local value for the variable at offset 20. Because %fs points to a different place in each thread, the value that comes back will be different in each thread.

4.3 Locks

Most of the complexity of thread programming is in correctly arbitrating access to shared resources or portions of code that aren't safe to be run by multiple threads at once. In fact the latter is the same thing as the former, given that the reason the code isn't safe to run more than once is that it accesses shared re-

sources. So, any threaded environment must offer some kind of locking constructs.

Spinlocks are built on top of `lock cmpxchg`. They're not directly available to end-users - though note that this is Lisp so it's intentionally not impossible to call internal functions if you know what you're doing. User-available locks have queues attached to them: a waiting thread puts itself on a queue, then sleeps by calling `sigtimedwait()` and can be woken later by the lock holder when it relinquishes the lock. When 2.6 is available we'll provide futex-based locks which should be a little more efficient than this, and allow fewer possibilities for missed signals.

We provide ordinary mutexes (one thread may hold a mutex, others have to wait). These are not recursive: if you own a lock and you try to get it again, you will hang potentially forever, because it's not free - if you're attempting to get a lock you already have, that usually indicates a problem. Recursive locks are also available for situations where they're necessary. If you attempt to get a recursive lock and you have it already, you just carry on holding it.

We also have condition variables, similar to those in the Posix thread model and in Java. These provide a race-free means for a thread that holds a lock to release it and go to sleep on a queue until woken by some other thread. When it's woken, it's guaranteed to reacquire the lock before being allowed to do anything else.

Timeouts are important: we usually don't want to lock indefinitely, and there are other situations where we want to abort an operation if it takes too long as well. For example in a web server, there's usually little point in serving a response if it's taken more than about a minute to compute, because the user will have gone elsewhere. Timeouts are trivial to implement in a native threads system: all we need do is call alarm(), and install a SIGALRM handler that signals a Lisp condition (similar to an exception in languages like Java). Then we can handle the timeout at whatever point is appropriate using standard Lisp facilities.

5 Future work

There is more work to be done on the threading implementation to make it really robust. This includes resizing the TLS area when we run out of room in it, adding parameters to control the stack sizes, and so forth.

Larger projects for the thread support include the ability to benefit from many of the features introduced in 2.6 by and for NPTL, such as futexes and the per-process GDT slots, and ports to non-x86 architectures (which can use an ordinary register for TLS instead of needing a segment selector) and non-Linux OSes. Andreas Fuchs has already reported some initial success on a FreeBSD port using their `rfork_thread` call.

Other SBCL projects in current development include Unicode support, MacOS X and Windows ports, and changes to take advantage of 64 bit address spaces.

6 Existing tools for FFI

Creating Lisp language bindings for a library is to some extent very easy. Given a function name and a shared library, we just use `dlopen()` and `dlsym()`, and call it.

The tedious part is in knowing what to call it *with*. Some tools exist that help with digging constants and

structure layouts out of header files, but it's not a completely automatic process.

- SB-GROVEL is part of SBCL. It takes a configuration file listing the constants and structure/union names/fields of interest, then writes, compiles and runs a short C program that uses sizeof and offsetof to determine the appropriate numbers, and writes them out as a Lisp source file. It has rough edges still, and obviously doesn't help with API calls or variables that are only implemented as preprocessor macros (e.g. `errno`)

- cparse [6] does a moderately good but not perfect job of parsing C headers. Unfortunately, anything less than gcc-style perfection is probably insufficient to parse GNU Libc headers.

- FFIGEN [7] is a patched gcc that can be used to build a dbm file of library calls in a format that makes it easy to create language bindings to them. OpenMCL (another free Lisp compiler, for PPC systems only) uses this and it's claimed to be pretty slick. As it's based on gcc it's not trivial to build, though, and someone will need to keep updating it against new versions of gcc as they're released. Presently only works with OpenMCL.

- SWIG, the Simplified Wrapper and Interface Generator, takes annotated C/C++ header files as its input specification, and can generate bindings for several high-level languages. It's reported that it now has s-expressions as an output format, and experimental support for transforming these to UFFI [8] declarations. If it became commonplace for library authors to annotate their header files appropriately, this could be a very powerful tool.

[6] http://bricoworks.com/ moore/cparse/

[7] http://openmcl.clozure.com/Doc/interface-translation.html

[8] UFFI is a portable CL package for foreign language interfaces.

7 Recommendations

Today's Unix-like systems are definitely easier to do interesting things with than those of a few years ago. It's often been possible for a long time to get the fault address from a SIGSEGV, but looking at old CMUCL code it's apparent that the exact method for doing this used to vary widely between different Unices, and often involved looking in kernel source to find the signal handler stack frame layout. These days - except, as we noted, on Linux/SPARC[9] - it's a pretty safe bet that `SA_SIGINFO` and a `ucontext` will do the trick.

However, the kernel/C library alone is no longer the platform for useful programs - today programmers depend on a host of layered libraries, whether toolkits such as GNOME or KDE, or smaller special purpose libraries like cdparanoia. In a spirit of continuous improvement, then, I'd like to close by offering the following suggestions to library authors:

- When the usual interface to something involves use of the preprocessor (e.g. `errno` or the `stat` family of calls), provide and document an alternative functional interface that can be called with `dlsym()`.

- Avoid mandating "convenience" features such as installing signal handlers for cleanup or calling exit() in error situations. You can't anticipate what the user will or won't find convenient, so allow some way to disable these things.

[9] and MacOS 10.1, but happily they've fixed it in 10.2

- If you can avoid monopolising the event loop, this is a good thing. Everyone wants to monopolise the event loop, and only one of the contenders can win.
- For extra points, provide some machine-readable file describing your exported functions and data structures. This could be as simple as running your header files through SWIG to make sure it can parse them and output appropriate interfaces.