Operating System Development with ATS

Work in Progress

Matthew Danish
Computer Science Department
Boston University, MA, USA 02215
http://cs-people.bu.edu/md

Hongwei Xi
Computer Science Department
Boston University, MA, USA 02215
hwxi@cs.bu.edu

Abstract

Typical operating system design is marked by trade-offs between speed and reliability, features and security. Most systems are written in a low-level untyped programming language to achieve optimal hardware usage and for other practical reasons. But, this often results in CPU, memory, and I/O protection flaws due to mistakes in unverified code. On the other hand, fully verified systems are exceedingly hard to construct on any industrial scale. A high-level programming language, with an expressive type system suitable for systems programming, can help alleviate many of these problems without requiring the enormous effort of full verification.

Categories and Subject Descriptors D.1.1 [*PROGRAMMING TECHNIQUES*]: Applicative (Functional) Programming; D.4.5 [*OPERATING SYSTEMS*]: Reliability

General Terms Languages, Reliability

Keywords dependent types, linear types, operating systems

1. Introduction

Operating systems software is fundamental to the modern computer: all other applications are dependent upon the correct and timely provision of system services. Ideally, the operating system is written with maximum attention to detail and the use of optimal algorithms. In practice, many difficult decisions must be made during the design and implementation of a realistic system.

There are many aspects of operating system development which contribute to this situation: the low-level behavior of hardware can be finicky, the asynchronous combination of system processes may produce unforeseen results, many of the resource-management problems are intractable to solve optimally, the slightest mistake can have profound consequences, and there is little room for any wasteful overhead. To maximize performance and ease of hardware interaction, most operating systems software is written in type-unsafe, low-level memory model programming languages like C or C++. But, typically this leads to compromised reliability and security because of programmer error.

High-level programming languages with modern, strong type systems have been offering guarantees of memory safety to applications programmers for decades. There are a number of such languages which have found a useful combination of partial program verification and ease of use. But, with few exceptions, these languages have not found favor amongst systems programmers. An old objection is that automated compiler optimizations are not sufficient to obtain suitable performance—but this has become less of an issue in recent years. A more troubling objection is that the high-level type system and memory model of many programming languages does not permit the flexibility or ability to reference or describe structures that an operating systems programmer needs, in ways that would statically catch problems before they manifest at run-time.

There are many projects that aim to address the issue of system software reliability in various ways; see section 5.1 for a catalogue of past work. These projects range from full verification efforts, to operating system designs that employ software-based protection, to high-level programming languages that supply features for better low-level work. The human resources required for even a relatively small-scale verification are tremendous, as shown by seL4 [16], or impose special requirements, such as Verisoft [1] which runs only on an obscure architecture. The approach taken by SPIN [3] and subsequent projects restricts the software protection model to that of a relatively inexpressive type-system. Few high-level languages offer dependent types, linear types, and efficient C compilation and integration with the same level of practicality as ATS.

This work aims to show that a programming language with dependent and linear types, ATS, can be used to phase reliable components into an existing operating system written in an unsafe language. The new, type-checked, code can be fully integrated at any level into the kernel without any need for special run-time support. In addition, the programmer can decide just how strong or how weak a level of specification is desired, as a balance between practicality and verification completeness is reached.

A general-purpose language like ATS is far broader in design than may be necessary or desirable for systems programming. Dependently-typed languages are notoriously difficult to learn how to program with effectively. Finding a domain-specific subset of ATS which trades some expressivity for usability—a language that could give benefit to non-experts in programming language theory—is another goal.

2. Platform

The ATS programming language The ATS project [31] is a compiler for a programming language with dependent types, linear types, and easy C integration. It also supports the manipulation of unboxed types of arbitrary size (also called flat types) and has template meta-programming capabilities to go with it. The ATS dependent type system [27] is largely based on earlier work on Dependent ML (DML) [29] and is flexible enough to be able to encode

PLPV'10, January 19, 2010, Madrid, Spain.

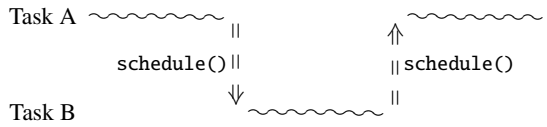

Figure 1. Linear time-line of processor execution

mathematical proofs [28][30], or to be used to write Linux™ device drivers [23]. This is achieved through syntactic separation of terms which may appear in proofs and those that may appear in programs. Certain properties, like guaranteed termination, are required of proofs, while programs may exploit general recursion.

The ATS compiler compiles itself and it is on the order of 90,000 lines of ATS code. There are a number of contributed libraries including, for example, some for AVL trees, hash-tables, and binary heaps that take advantage of linear and dependent types for efficiency and correctness. ATS programs can elide support for language run-time features (like garbage collection)—this is helpful for operating system development.

The Quest operating system Quest [26] is a small operating system developed in C for research into embedded systems, meeting service requirements of end-users, and general pedagogical purposes. It is designed to operate on an Intel™ 32-bit Pentium™ architecture machine [13], it supports symmetric multiprocessing, and has a few basic peripheral drivers. It provides a minimal Unix-like set of system calls and it is simple, having less than 8000 lines of code.

3. Case Study: Scheduler

The scheduler of the Quest operating system presented a good starting point because it is already written in a fairly high-level style. There are a number of properties, an important subset of which could be easily encoded in ATS, which provide for a quick demonstration of the feasibility of compiling and loading ATS code as part of a real operating system kernel.

3.1 The `schedule` function

The main kernel interface is a function named `schedule`. When invoked, a simple algorithm picks one task from a queue of tasks which are not running and switches the processor context of memory-management and register-state to that of the selected task. The old task's context is saved to memory, to be picked up again at some future point in time.

`schedule` is a peculiar function when viewed on a linear time-line of processor execution. One example is depicted in figure 1: a task *A* is interrupted and the kernel invokes `schedule()` and then the following return occurs in the context of task *B*. In the future, some other invocation will cause a return to task *A*. The function requires no arguments and returns no values. However, there are very important pre- and post-conditions associated with the function, which if not satisfied, will lead to dead-locks and system crashes.

The scheduler manipulates some data-structures, and these interactions must take place atomically without interruption—this is called a *critical section*. On a uniprocessor machine, it is sufficient to disable interrupts while working in this critical section. Interrupts are already disabled upon entry into privileged mode by way of interrupt-gates [13], a hardware feature which is used by Quest to gain access to the kernel. On a multiprocessor machine, this is no longer sufficient because it is possible that more than one processor enters an interrupt-gate and attempts to use the scheduler.

Therefore, for multiprocessor operation, there is a spin-lock which guards access to the kernel's data-structures. Using atomic machine instructions, a processor waits for a word in memory to become equal to `0` so it can write a `1` into that location. While it is waiting, the processor "spins" around a tight loop while doing no useful work. If every entry into the kernel follows this protocol, then only one processor at a time will be able to modify kernel data-structures while the others wait. Spin-locks do waste CPU resources, but they have the advantage that the latency between an unlock and the subsequent lock operation is very low, they require no support from a scheduler, and they are simple to implement and build upon. Using spin-locks achieves a basic level of protection for critical sections which, while not ideal, is suitable for our purposes.

```
absview KERNEL_LOCK_v

extern fun
lock_kernel ():pure  (KERNEL_LOCK_v | void)

extern fun
unlock_kernel (pf: KERNEL_LOCK_v | (* empty *) ):pure  void

extern fun
schedule (pf: !KERNEL_LOCK_v | (* empty *) ): void
```

Listing 1. Scheduler function prototypes

The spin-lock must be obtained prior to critical sections and released in a timely fashion but only after the critical section has completed. The lock cannot be grabbed inside of the `schedule` function because the critical section may be larger than just an invocation of `schedule`. For example, a task may want to add to the queue before scheduling, and these operations must be completed together without interruption. In addition, it is not possible to relinquish the lock before the hardware context switch, because then another processor might try to use certain resources which are not yet available until after the switch executes. Therefore, the most general interface requires a pre- and post-condition to the scheduler: that the kernel spin-lock be held before scheduling and that it is still held afterwards. In the C implementation, this is given as a comment, but in the ATS implementation it can be enforced as a type.

3.2 Specification

The relevant ATS function prototypes are given in listing 1. An **`absview`** declares an abstract view, also known as a linear proposition, to represent the kernel spin-lock. The lock is still implemented in C, therefore it is declared abstract to ATS. "View" is the name given to a linear type associated with a proof.

Since it is abstract, two functions are declared to manipulate the kernel lock: `lock_kernel` and `unlock_kernel`. These are implemented in C, and have no parameters or return values, but do have proof implications. The first function, `lock_kernel`, produces a linear proof term of the view for the kernel lock. The second function, `unlock_kernel`, is said to consume a linear proof term of that view. Finally, the third function, `schedule`, uses but does not consume the linear proof term—this is denoted by the `!` operator prefixed to the view.

The production and consumption of views has no run-time overhead or presence. Propositions and views which appear on the left-hand side of the `|` syntactic separator are erased after type-checking. To the outside world, these three functions all appear to be parameter-less and without return value. To the type-checker, these annotations enforce the pre-condition that the kernel be locked before scheduling, and that the kernel lock be explicitly released after scheduling.

This is a relatively simple but surprisingly common usage of views. It is also imperfect, because it does not prevent the kernel lock from being grabbed more than once (and thus dead-locking

```
1  implement schedule (pf_lock | (* empty *)) = let
2    (* Search a bitmap which gets set when a runqueue is non-empty *)
3    val prio = bitmap_find_first_set (!runq_bitmap, MAX_PRIO_QUEUES)
4      where {val (vbox pf | runq_bitmap) = get_runq_bitmap (pf_lock | (* empty *) ) }
5  in if prio < 0 then
6      let (* Nothing to do, go IDLE. *)
7        val phys_id = LAPIC_get_physical_ID ()
8        val idle_sel = idleTSS_selectors[phys_id]
9          where {val (vbox pf | idleTSS_selectors) = get_idle_TSS_selectors (pf_lock | (* empty *) ) }
10       val cur_task = str ()
11     in if task_id_eq (cur_task, idle_sel) then
12          () (* No task switch needed, was already IDLE. *)
13       else jmp_gate (pf_lock | idle_sel)
14     end
15   else let (* Got a task waiting at runqueue[prio]. *)
16       val (pf_runq | p_runq) = runqueue_get_view_ptr (pf_lock | (* empty *) )
17       val next = queue_remove_head (p_runq[prio])
18       (* Clear the bit if this was the last item on the queue: *)
19       val _ = if not (task_id_eq (task_id_zero, p_runq[prio])) then ()
20               else let val (vbox pf | runq_bitmap) = get_runq_bitmap (pf_lock | (* empty *) )
21                    in bitmap_clr (!runq_bitmap, prio) end
22       val _ = runqueue_put_view_ptr (pf_runq | p_runq)
23       val cur_task = str ()
24     in if task_id_eq (next, cur_task) then
25          () (* No task switch needed, resume current task. *)
26       else jmp_gate (pf_lock | next)
27     end
28 end
```

Listing 2. Scheduler implementation

immediately) by the same processor. This is an example of the trade-off between practicality and specification completeness. The types could be designed to prevent multiple locking. But the most common errors are due to holding the kernel lock and forgetting to release it, which this does address. Anecdotally, one bug in the C implementation arose because one branch of some conditional logic unlocked the kernel twice. That kind of bug can be very difficult to track down because it does not cause an immediate dead-lock, but instead allows a second processor to manipulate kernel data-structures while another one was already working. This problem would have been caught by the ATS type-checker using these annotations.

3.3 Implementation

The implementation of the `schedule` function in ATS is shown in listing 2 as a demonstration. The basic algorithm is to check a set of queues for any waiting tasks. There are `MAX_PRIO_QUEUES` total queues in decreasing priority order. If no task is waiting, then the processor-specific idle task is read from an array. In every case it is necessary to check if the current task is the same as the selected task, and if so, do nothing but return.

The code is about the same length as a C implementation of the same function but the ATS type-system is checking much more. For example, on line 3 an inline assembly routine is invoked which uses an Intel™ machine instruction to quickly find the first set bit in a word representing a bit-map. The linear proposition which proves safe access to the run-queue bit-map pointer is contained in a `vbox`, which is a construct that automatically revokes views upon leaving the static scope of its binding. This means simply that the proof that permits the use of the pointer is contained to a small portion of the overall code. In addition, on that line, the `runq_bitmap` is defined to be a certain size and that is statically checked against the second argument, `MAX_PRIO_QUEUES`, for bounds-safety.

There are other examples of the use of views for the safety of pointer access (see also [33]). The `task_id` is an indexed integer type, which permits reasoning that is used by some of the queue manipulation functions. The `jmp_gate` and `str` functions are actually inline wrappers around assembly which is optimized into a single instruction by the C compiler.

These two listings, and more, currently constitute the scheduler for a working version of the Quest operating system and it has been tested using emulators such as QEMU [2], Bochs [7], and VMWare [14] as well as on real machines.

4. Case Study: Simple Memory Manager

4.1 Background

An operating system kernel is supplied with some basic hardware tools to manage address spaces and protect memory, but it is largely responsible for the proper design and implementation of memory allocators. There are several levels to consider:

Physical memory represents the real, hardware resource. It is divided into *frames*: consecutive regions of RAM all of the same size and alignment, typically 4096 bytes.

Virtual memory is the layer at which most applications and kernel code typically works. It is divided into *pages* reminiscent of physical frames except that consecutive pages may be assigned to arbitrary frames (or not assigned at all).

Address spaces are sets of virtual memory page assignments. An address space provides a coherent view of memory for an application, but it could be composed of disparate physical frames.

The kernel heap is a subset of an address space designated for miscellaneous and dynamic kernel allocation of indefinite extent. It is typically managed by an allocator capable of handing out and keeping track of smaller chunks of memory.

Different methods are used to manage these resources. For example, in Quest, a bit-map is maintained to keep track of the first 128MB of physical memory. Virtual memory assignments must be

```
extern prfun
FREE_TABLE_v_of_LOCK_v (pf: LOCK_v):proof
  (FREE_TABLE_v, FREE_TABLE_v  -o  LOCK_v)
                               proof

extern prfun
USED_TABLE_v_of_LOCK_v (pf: LOCK_v):proof
  (USED_TABLE_v, USED_TABLE_v  -o  LOCK_v)
                               proof
```

Listing 3. Some allocator specifications

formulated into a set of tables in memory referred to as *page tables* so that the hardware memory-management unit can read it. A full address space is given as a *page table directory* and the current address space is stored in a designated machine register [13]. The kernel heap can be managed much like user-space allocators do such as those used for `libc malloc`, but it typically has a few requirements not found at user-level:

- Low-latency operation is important because it may be invoked in the midst of some critical section.
- Fragmentation can lead to poor performance.
- Oftentimes aligned or restricted pointers are desired by the caller.
- If memory runs out, then it must use the other subsystems of the kernel to gain more (or fail). This can lead to unbounded delay.

In general, the design of a kernel heap allocator is a difficult problem and there is not a single solution for all purposes. A real operating system provides several mechanisms which have different trade-offs.

4.2 A simple allocator

Quest has a solution that is well-known [17], simple, quick, avoids external fragmentation at the cost of internal fragmentation, and offers certain alignment guarantees. Only blocks with a size that is equal to a power of two are handed out. There is a minimum size (currently 2^5 bytes) and a maximum size (currently 2^{16} bytes). When a request comes in for a certain number of bytes n, the allocator looks for the smallest k such that $n \leq 2^k$ is satisfied. Then it either pulls a block out of the pool reserved for 2^k-sized blocks, or it requests a 2^{k+1}-sized block and splits it. The address and size of the block are recorded in a table and then a pointer is returned to the caller. In the future, when the block is freed, the table is consulted to find out how large it is and then it gets placed back in the appropriate free-block pool.

4.3 Properties of interest

The allocator implementation in ATS is too large to include here but there are a number of interesting fragments which highlight potentially useful techniques for similar work.

Preventing pointer aliasing Listing 3 demonstrates a way to use linear types to avoid aliasing pointers. There already exists a definition for `LOCK_v` that is very similar to listing 1, so there is no need to repeat it. The proof-function `FREE_TABLE_v_of_LOCK_v` is not a real function, but rather more like a theorem available for use by the programmer in proofs and checked by the compiler prior to type erasure. It consumes a lock view and produces two linear views: the first is of a table holding the free-block lists, the second is a linear proof implication that allows us to reconstruct the lock view if we give up the table view.

Since the only way to get a free-block table view is to invoke this function on the lock view, and the only way to get the lock view back is to give up the free-block table view, this prevents aliasing

```
absviewtype FREE_BLOCK_vt (int)

sortdef index = {i:int | MIN_POW ≤ i; i ≤ MAX_POW}

extern fun
split_free_block
  {i:index | MIN_POW < i}
  (b: !FREE_BLOCK_vt(i) >> FREE_BLOCK_vt(i-1),
   i:int i):pure
     FREE_BLOCK_vt(i-1)
```

Listing 4. Splitting a free block

of the table pointer. The used-block table is guarded in the same way. This idea can also be used to provide a more robust version of listing 1.

Specification simplifies implementation Listing 4 specifies the function which splits a block into two half-sized blocks. The notion of a "free block" is defined as an abstract linear type constructor, also known as an abstract viewtype constructor in ATS. It is indexed by an `int` which denotes the size of the block: `FREE_BLOCK_vt(i)` is the static term for a viewtype of a region of memory of size 2^i bytes that is available for use.

The types of ATS static terms are called *sorts*. The type constructor `FREE_BLOCK_vt` has the sort int → type where int, type are base sorts. The sort alias named `index` is an example of a subset sort—a proposition combined with a sort. The ATS concrete syntax is purposefully designed to mimic set-comprehension notation for clarity, and permits any linear constraint. In this case, the `index` sort is restricted to the valid values for the size of a free block, and is meant to be used for indexing the `FREE_BLOCK_vt` type constructor.

The definition of `split_free_block` adds another proposition which ensures that the index of the free block given as a parameter is greater than the minimum. The reason for this, ultimately, is that `split_free_block` must not operate on minimum-sized blocks. The definition goes on to specify that the function takes two dynamic arguments: the block b, and the index i. The type of the argument b expresses that it has a linear viewtype and it preserves (denoted by the prefix operator `!`) the linear view—but with a modification (denoted in shorthand by the infix operator `>>`), namely that the index is decremented. The type of the dynamic argument i is the type constructor `int`[1] indexed by the static term i. The return type of the function is another linear viewtype also with a decremented index.

The end result is that before invoking `split_free_block` there is one linear value of viewtype `FREE_BLOCK_vt(i)` and afterwards there are two linear values of viewtype `FREE_BLOCK_vt(i-1)`. The first output will remain, in some sense, the same as the input (in reality, it will be the same pointer value) but now it can only be treated as a region of size 2^{i-1} bytes. The actual implementation can be one line of simple inline-able C code because of such strong specifications.

Re-use with confidence In the original C implementation, a table of allocated blocks was maintained as a simple linear array. More efficient data-structures could have been used but the effort of debugging complex manipulations of trees in C outweighed the benefit. During the porting effort to ATS, it was possible to immediately take advantage of an existing AVL tree library written and provided with proofs. With templates and flat types, it integrated easily into

[1] The name `int` is overloaded as a base sort, a type, and a type constructor, depending on context.

```
extern fun
remove_used_table
  {i:index}
  (pf: !USED_TABLE_v | b: !FREE_BLOCK_vt i,
   i: &(int?) >> opt (int(i), err==0)):pure
    #[err:int | err ≤ 0] int err
```

Listing 5. Interface to library

the kernel and compiled to specialized instantiations for the table entries.

In listing 5 is one of the interface functions which shows a few features that are useful when writing systems code. First, though, it continues to take advantage of the subset sort and linear view techniques discussed previously. Then, the parameter *i* is specified as call-by-reference using the &-notation. Using the >> shorthand again, it specifies that the function is going to modify the reference parameter. It expects an uninitialized `int` (denoted `int?`) as input, which means that this specifies an *output* parameter. After the function returns, the type of *i* is `opt (int(i), err==0)` which is a constructor asserting that an `int` statically known as *i*, the index of the free-block, will be stored in the variable `i` if err = 0. The static variable `err` is bound by the syntax `#[err:int | err ≤ 0]` which is an existential quantifier and also asserts that `err` is non-positive. The actual return value of the function is an int corresponding to the static int `err`.

The net effect is that the index of the block is returned as an output parameter by reference, but only when the error code is zero. The specification of this function prevents it from being mis-used: in order to get the dynamic value of the index, ATS will force the programmer to prove that err = 0 (by using an **if** statement, for example). This is achieved efficiently, using common techniques like call-by-reference and error codes.

5. Conclusion

5.1 Related work

Operating Systems Previously, a simple Linux™ device driver was written using an early version of ATS. That work demonstrated the feasibility of using ATS for device driver development and illustrated a few possible benefits of doing so [23].

House [9] is an operating system project written primarily in the Haskell functional programming language. It takes advantage of a rewrite of the GHC [22] run-time environment that eliminates the need for OS support, and instead operates directly on top of PS/2-compatible hardware. Then a foreign function interface is used to create a kernel written in Haskell. There is glue code written in C that glosses over some of the trickiness. For example, interrupts are handled by C code which sets flags that the Haskell code can poll at safe points. This avoids potentially corrupting the Haskell heap due to interruptions of the Haskell garbage collector while it is an inconsistent state. The Hello Operating System [8] is an earlier than and similar project to House which features a kernel written and compiled using Standard ML of New Jersey [4], bootstrapped off of Linux™.

Singularity [12] is a microkernel operating system written in a high-level and type-safe programming language that uses language properties and software isolation to guarantee memory safety and eliminate the need for hardware protection domains in many cases. It incorporates linear types into the language to enable optimization of certain communications channels.

SPIN [3] is an operating system written in a type-safe language called Modula-3 [5] and it relies upon the language's safety and module mechanisms to build its protection model and extensibility features, though the expressiveness of the type system is relatively limited compared to modern languages.

Verification The Verisoft project [1] sought to create a fully-verified system from the hardware level up. It relies upon custom hardware architecture which has itself been formally verified, and a verified compiler to that instruction set. Interfaces with peripherals are modelled as finite state machines, though only to a limited extent.

The seL4 project is based on the family of micro-kernels known as L4 [18]. Recently, a refinement proof was completed [16] that demonstrated the adherence of a high-performance C implementation to a generated executable specification, created from a Haskell prototype, and checked in the Isabelle [21] proof assistant. The Haskell prototype is itself checked against a high-level design. One weakness is that changes made at a high-level can have rippling effects and may require the entire proof to be redone.

The VFiasco [11] project takes a different tactic towards the development of a verified operating system. It presumes that a high-level strongly typed language will need to have its rules broken in order to implement various kernel functionality. Therefore, they claim, it is better to write the kernel in an unsafe language such as C++ and then discharge proofs that are mechanically generated from the C++ source code, using an external theorem prover system.

Languages CCured [20] checks memory safety of existing C programs by applying a carefully designed strong type system which differentiates between different kinds of pointer use. It inserts run-time checks where memory safety cannot be proven statically, and tracks meta-data about certain pointers in order to implement those run-time safety tests—these are often called "fat" pointers.

SafeDrive [32] employs a type system called Deputy [6] that uses type annotations on C programs to eliminate the need for "fat" pointers in most cases. This means that in many cases, existing binary interfaces can be used without modification—for example, in a Linux™ kernel driver.

BitC [24] is a language intended to be used for systems programming with an ML-style type system with effects. It allows precise control of the representation of types in memory while enjoying the advantage of static type inference. Theorem-proving is syntactically supported but left to a (yet to be created) plugin or third-party application.

Cyclone [15] is a memory-safe dialect of C. It achieves its goal by restricting the behavior of C programs and then recovering some of that expressiveness by giving the programmer features that: insert NULL-pointer checks, use "fat" pointers when pointer arithmetic is desired, have growable regions as an alternative to classical manual memory management, and more.

Guru [25] is a programming language similar to ATS in that it is designed to support dependently-typed functional programming that is efficiently compiled to C. While ATS is *stratified* syntactically, Guru is based on OpTT which supports arbitrary program terms in types—but at the expense of permitting general recursion.

Ynot [19] is an extension to the Coq [10] proof assistant for reasoning about dependently-typed programs with side-effects, using monads and programmer-supplied specifications. Programs in languages such as Haskell can be automatically extracted from Coq proofs but the facility is not currently well-supported enough to suffice for Ynot. A compiler for Ynot is planned for future development.

5.2 Future work

The successful integration of two ATS modules into Quest shows promise that it is feasible to write significant portions of an operating system in a language with dependent and linear types. The

question then becomes: what advantage is there gained by doing this? This can be explored on a number of axes.

Performance Does the ATS compiler cause any performance degradation despite compiling to relatively straightforward C? Does having the confidence to use more complex and efficient data-structures in ATS than C make up that difference?

Debugging Can ATS be used to avoid tedious debugging sessions that are the hallmark of systems programming? Can lightweight theorem-proving help catch bugs before they manifest, or help track down existing bugs through increasing annotation?

Interface What would a dependent and linear type-based specification of the system call interface look like? Would it help prevent security flaws?

And finally, it remains to be seen what changes in the language design can ease the burden of specification and theorem-proving on the programmer, and make it more accessible to a wider audience.

Acknowledgments

Richard West, with Gary Wong, created the Quest OS project, and spent time answering many questions about it. Gabriel Parmer made helpful suggestions regarding the memory allocator and other details of systems programming. Likai Liu provided feedback on drafts of this paper. This work is partially supported by NSF grant no. CCF-0702665.

References

[1] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. The Verisoft Approach to Systems Verification. In *VSTTE '08: Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, pages 209–224, Berlin, Heidelberg, 2008. Springer-Verlag.

[2] Fabrice Bellard et al. QEMU: machine emulator. `http://www.qemu.org/`, 2009.

[3] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin GÃijn Sirer, Emin Gun Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, 1995.

[4] Matthias Blume et al. Standard ML of New Jersey. `http://www.smlnj.org/`, 2009.

[5] Luca Cardelli et al. Modula-3 report (revised). Technical report, Digital Equipment Corp. (now HP Inc.), Nov 1989. `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-52.html`.

[6] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. *Programming Languages and Systems*, chapter Dependent Types for Low-level Programming, pages 520–535. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007.

[7] Bryce Denny, Christophe Bothamy, Donald Becker, et al. BOCHS: x86 PC emulator. `http://bochs.sourceforge.net/`, 2009.

[8] Guangrui Fu. Design and Implementation of an Operating System in Standard ML. Master's thesis, University of Hawaii, Aug 1999. `http://www2.hawaii.edu/~esb/prof/proj/hello/`.

[9] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. *SIGPLAN Not.*, 40(9):116–128, 2005.

[10] Hugo Herbelin. Coq proof assistant. `http://coq.inria.fr/`, 2009.

[11] M. Hohmuth and H. Tews. The VFiasco approach for a verified operating system. In *Proceedings of the 2nd ECOOP Workshop on Programming Languages and Operating Systems*, 2005. `http://www.cs.ru.nl/H.Tews/Plos-2005/ecoop-plos-05-letter.pdf`.

[12] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. In *ACM SIGOPS Operating System Review*, volume 41, pages 37–49. Association for Computing Machinery, Apr 2007.

[13] Intel Inc. Intel™ 64 and IA-32 Architectures Software Developer's Manuals. `http://www.intel.com/products/processor/manuals/`.

[14] VMWare Inc. VMWare: Virtual Machine software. `http://www.vmware.com/`, 2009.

[15] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[16] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, Oct 2009.

[17] Kenneth C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, 1965.

[18] Jochen Liedtke. Toward Real Microkernels. CACM, 39(9):70–77, 1996.

[19] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 229–240, New York, NY, USA, 2008. ACM.

[20] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[21] Larry Paulson and Tobias Nipkow. Isabelle proof assistant. `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`, 2009.

[22] Simon Peyton-Jones, Simon Marlow, et al. The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`, 2009.

[23] Rui Shi. Implementing reliable Linux device drivers in ATS. In *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 41–46, New York, NY, USA, 2007. ACM.

[24] Swaroop Sridhar, Jonathan S. Shapiro, and Scott F. Smith. Sound and complete type inference for a systems programming language. In *APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pages 290–306, Berlin, Heidelberg, 2008. Springer-Verlag.

[25] Aaron Stump, Morgan Deters, Adam Petcher, Todd Schiller, and Timothy Simpson. Verified programming in Guru. In *PLPV '09: Proceedings of the 3rd workshop on Programming languages meets program verification*, pages 49–58, New York, NY, USA, 2008. ACM.

[26] Richard West. The Quest operating system. `http://www.cs.bu.edu/fac/richwest/quest`, 2009.

[27] Hongwei Xi. Applied Type System (extended abstract). In *Post-workshop Proceedings of TYPES 2003*, pages 394–408, 2004.

[28] Hongwei Xi. ATS/LF: a type system for constructing proofs as total functional programs. `http://www.ats-lang.org/PAPER/ATSLF-PAfestschrift.pdf`, 2004.

[29] Hongwei Xi. Dependent ML: An approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, 2007.

[30] Hongwei Xi. Examples encoding deduction systems. `http://www.ats-lang.org/EXAMPLE/LF/LF.html`, 2009.

[31] Hongwei Xi et al. The ATS language. `http://www.ats-lang.org/`, 2009.

[32] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: safe and recoverable extensions using language-based techniques. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 45–60, Berkeley, CA, USA, 2006. USENIX Association.

[33] Dengping Zhu and Hongwei Xi. Safe programming with pointers through stateful views. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, pages 83–97. Springer, 2005.