Thread Verification vs. Interrupt Verification

John Regehr
School of Computing
University of Utah

ABSTRACT

Interrupts are superficially similar to threads, but there are subtle semantic differences between the two abstractions. This paper compares and contrasts threads and interrupts from the point of view of verifying the absence of race conditions. We identify a small set of extensions that permit thread verification tools to also verify interrupt-driven software, and we present examples of source-to-source transformations that turn interrupt-driven code into semantically equivalent thread-based code that can be checked by a thread verifier.

1. INTRODUCTION

For programs running on general-purpose operating systems on PC-class hardware, threads are probably the most important abstraction supporting concurrent programming. On the other hand, interrupts are the dominant method for expressing concurrency in embedded systems, particularly those based on small microcontroller units (MCUs). All major MCU architectures support interrupts, and a large number of these chips are deployed in embedded systems: according to a Gartner report, 3.5 billion 8-bit MCUs and a billion 16-bit MCUs were shipped in 2003. The correctness of interrupt-driven software is important: a substantial number of these 4.5 billion MCUs were deployed in safety-critical applications such as vehicle control and medical automation.

Most work on verifying concurrent software has focused on thread-based and process-based concurrency; interrupts have received relatively little attention. The thesis of this position paper is that:

> Verifying the absence of race conditions in interrupt-driven systems is important, but the technology for this is primitive. We must understand interrupts and their semantics in order to understand where thread verifiers can, and cannot, be applied to interrupt-driven systems. In particular, we want to identify a minimal set of extensions to verifiers for thread-based programs that permit them to also check interrupt-based programs. A secondary goal is to exploit the semantics of interrupts to make checking faster and more precise.

Of course, threads come in many flavors. In this paper we'll assume POSIX-style threads [5]: preemptively scheduled blocking threads, scheduled either in the kernel or at user level.

2. THREADS AND INTERRUPTS

The following are some significant ways in which threads differ from interrupts.

Blocking. Interrupt cannot block: they run to completion unless preempted by other interrupts. The inability to block is very inconvenient, and it is one of the main reasons that complex logic should not be implemented in interrupt code. However, non-blocking execution has a few compelling advantages. First, all interrupts, in addition to the non-interrupt execution context, can share a single call stack. Threads, of course, require their own stacks, making them an unsuitable abstraction for low-end MCUs that typically have at most a few KB of RAM. Second, since interrupts never block, their internal states are invisible to non-interrupt code. In other words, interrupts execute atomically with respect to code running in the non-interrupt context. Third, non-blocking execution means that interrupts are not subject to most forms of deadlock.

Preemption and scheduling. Threads typically have symmetrical preemption relations: for any given pair of threads, either one can preempt the other. In contrast, asymmetrical preemption relations are the norm for interrupts. First, all interrupts can preempt non-interrupt code, whereas non-interrupt code can never preempt any interrupt. Second, interrupts are often scheduled using fixed priorities, resulting in asymmetrical preemption relations among interrupt handlers.

Some hardware platforms, such as the programmable interrupt controller on a PC, enforce priority scheduling of interrupts. On other systems, prioritized interrupt scheduling must be implemented in software. For example, the Atmel ATmega128 [1], a popular MCU that is the basis for Mica2 sensor network nodes [7], performs priority scheduling only among interrupts that are concurrently pending. Once an interrupt begins to execute on an ATmega128, it

can be preempted by an interrupt of any priority, if interrupts are enabled. To implement priority-based preemptive scheduling on this platform, software must manipulate the individual enable bits associated with various interrupts.

Concurrency control. A thread lock uses blocking to prevent a thread from passing a given program point until the lock resource becomes available. Since interrupts cannot be blocked once they begin to execute, concurrency control consists of preventing an interrupt from starting to execute in the first place. This is accomplished either by disabling all interrupts, or by selectively disabling only interrupts that might interfere with a particular computation. The former is cheaper while the latter avoids incidentally delaying unrelated code.

Since preemption in interrupt-based systems is asymmetrical, so must be locking. In other words, while the non-interrupt context must disable interrupts in order to execute atomically with respect to interrupts, code running in interrupt mode does not need to take any special action to run atomically with respect to non-interrupt code.

Reentrance. An interrupt is *reentrant* if there may be multiple concurrent invocations of the same handler. Reentrant interrupts come in two varieties: accidental and deliberate. Accidental reentrance occurs when a developer misunderstands the consequences of enabling interrupts inside an interrupt handler. We mention this type of reentrance because it usually leads to buggy systems, and because in our experience it is common in real embedded systems.

Deliberate reentrance is a sophisticated technique that can be used to reduce interrupt latency. It is useful when code close the the beginning of a long-running interrupt handler is subject to time constraints. In this case, by permitting subsequent invocations of the handler to fire before previous ones have finished executing, the average latency of the early part of the handler can be decreased. For example, the timer interrupt handler in the AvrX system [3] is reentrant in order to avoid missing ticks.

Reentrance has two unavoidable costs. First, later parts of the handler will have their average latency increased, rather than decreased. Second, in practice it is difficult to create correct reentrant interrupt code.

Invocation style. Some interrupts are *spontaneous*: they may fire at any time. For example, interrupts generated by a network interface are spontaneous. Other interrupts are *requested*: they only occur in response to an action taken by the running program. For example, timer interrupts are requested, as are analog to digital converter (ADC) completion interrupts. The causal relation between an interrupt request and a subsequent interrupt can only be seen by understanding the semantics of the underlying hardware.

Deferring interrupts. Interrupts are often subject to time constraints: they need to execute within a bounded time after they become pending. The best way to ensure that these constraints are met is to make interrupt handlers short. A common idiom is for an interrupt to perform minimal computation associated with the interrupt request, such as acknowledging a hardware condition and possibly moving data associated with the interrupt into a memory buffer. Then, the interrupt handler initiates a longer-running computation in a deferred context, such as a bottom-half handler or thread.

3. VERIFYING INTERRUPTS

Given the existence of good thread verification tools, it is tempting to simply shoehorn interrupts into the thread model. In other words, if we can transform an interrupt-driven program into a semantically equivalent thread-based program, then an existing thread verifier can be used to find bugs in the original system. This section uses a running example to show where this is easy and where it is difficult.

Consider a C program that contains two interrupt handler functions: `irq5()` and `irq7()`. We further assume that interrupts are scheduled using priorities: interrupt 5 can preempt interrupt 7, but not the other way around. A non-portable language extension is used to tell the compiler to generate interrupt prologue and epilogue code for the interrupt functions instead of using the standard calling convention. Any functions called from an interrupt handler can use the standard calling convention.

The skeleton of this system's code is:

```
__INTERRUPT__ void irq5(void) {
  ...code...
}
__INTERRUPT__ void irq7(void) {
  ...code...
}
int main(void) {
  ...code...
}
```

Of course this code cannot be checked as-is by a verification tool that understands threads; it needs to be rewritten by a source-to-source translation tool. To treat an interrupt as a thread, the verification tool must consider the interrupt as being invoked in an infinite loop:

```
void irq5_thread(void) {
  while (1) irq5();
}
```

Interrupt 7 looks the same. In addition, the interrupt threads must be spawned at boot time:

```
int main(void) {
  create_thread(irq5_thread);
  create_thread(irq7_thread);
  ...code...
}
```

Unfortunately we cannot yet run the transformed code through a thread checking tool: a number of the semantic differences that we discussed in Section 2 still need to be addressed.

Blocking, preemption, and scheduling. In the expected case, two threads may each preempt each other at any instruction boundary where preemption is not suppressed by locks. Even if one thread has a higher priority, the scheduler in a general-purpose OS makes dynamic priority adjustments that can temporarily boost the priority of a low-priority thread. Furthermore, a thread running on a general-purpose OS can encounter a page fault on any memory reference, causing it to block involuntarily.

Interrupts, on the other hand, are not subject to dynamic priority adjustments and must never block on a page fault or for any other reason. Consequently, the highest-priority runnable interrupt always executes: priorities are strictly enforced. Support for "strict priorities" must be added to a model checker in order to support analysis of interrupt-based code. The first benefit is that this will eliminate false positives: race conditions that cannot actually occur. The second benefit is that the size of the state space can be greatly reduced by considering fewer preemption points. This insight was exploited by Hatcliff et al. [11] in the context of fixed-priority threading, but it is not clear how they dealt with the problem of priority inversion through page faults (the issue is not addressed in their paper).

The threads that are used to model interrupts must be spawned with priorities that the verification tool interprets as strict:

```
int main(void) {
  create_thread(irq5_thread,
                HIGHEST_STRICT_PRIORITY);
  create_thread(irq7_thread,
                SECOND_HIGHEST_STRICT_PRIORITY);
  ...code...
}
```

It is important that priority range assigned to interrupt handlers does not overlap with the priority range that is available to threads.

Concurrency control. To emulate interrupt-style concurrency control, thread locks must be created to model the processor's interrupt-enable bits. Functions for executing atomically in the original interrupt-based program look like these:

```
void begin_critical_section (void)
{
  // non-portable code for disabling interrupts
}
void end_critical_section (void)
{
  // non-portable code for enabling interrupts
}
```

In practice these functions are written in such a way that they operate properly when nested. We ignore the problem of recursive locks here; they are supported by many threading systems. To create versions of these functions that a thread-checking tool can make use of, we translate them as follows:

```
void begin_critical_section (void)
{
  acquire_mutex (irq5_lock);
  acquire_mutex (irq7_lock);
}
void end_critical_section (void)
{
  release_mutex (irq5_lock);
  release_mutex (irq7_lock);
}
```

To complete the transformation supporting concurrency control, we need to turn asymmetrical interrupt-style synchronization into symmetrical thread-style synchronization by requiring that each "interrupt thread" always runs with the appropriate lock held:

```
void irq5_thread(void) {
  while (1) {
    acquire_mutex (irq5_lock);
    irq5();
    release_mutex (irq5_lock);
  }
}
void irq7_thread(void) {
  while (1) {
    acquire_mutex (irq7_lock);
    irq7();
    release_mutex (irq7_lock);
  }
}
```

It is necessary to have a lock for each interrupt handler, rather than a single global lock, to allow interrupts to preempt each other. If an interrupt-driven program uses the processor's individual interrupt enable bits in addition to the global interrupt bit, then these must be modeled with additional locks.

Reentrance. A reentrant interrupt is one that can have multiple invocations on the stack at the same time. Since the number of concurrent invocations cannot usually be bounded, only verifiers that can model an unbounded number of threads, such as Henzinger et al.'s CIRC extensions to Blast [12], are suitable for reasoning about reentrant interrupts.

Invocation style. The code presented so far models spontaneous interrupts that may fire at any moment; it can also be used to model requested interrupts but this loses information about the causal relation between the request and the subsequent interrupt. We know of no tool for analyzing interrupts that exploits causality between requests and interrupts, but we believe that this would be useful. To model interrupt requests, explicit interrupt request calls must be inserted either manually or automatically. An interrupt request can be exposed to a thread checking tool using a condition variable:

```
void request_irq5(void) {
  cond_signal (irq5_request);
}
```

Then, the associated interrupt is only permitted to fire following a request:

```
void irq5_thread(void) {
  while (1) {
    cond_wait (irq5_request);
    acquire_mutex (irq5_lock);
    irq5();
    release_mutex (irq5_lock);
  }
}
```

Deferring interrupts. Like interrupt requests cause interrupts to fire, interrupts cause deferred code to run. Deferred thread-mode code requires no special support since the interrupt will signal a waiting thread using standard thread

primitives. Bottom-half handlers in the Linux kernel, or DPCs (deferred procedure calls) in the Windows kernel [20], must be modeled as run-to-completion code running at a priority strictly higher than any real thread and strictly lower than any interrupt.

Summary. We believe that thread verification tools can be used to check interrupt-driven software under three conditions:

1. A source-to-source transformation tool is used to convert interrupt code into semantically equivalent thread code.
2. The thread checking tool supports a separate class of strict priorities that (1) are higher-priority than any real thread and (2) model the fact that interrupts run atomically with respect to lower-priority interrupts and non-interrupt code.
3. If the interrupt-driven code contains any reentrant interrupts, the thread-checking tool must be able to model multiple outstanding invocations of a single interrupt handler.

4. OTHER INTERRUPT VERIFICATION PROBLEMS

Although this paper has focused on race conditions, interrupt-driven code is subject to a number of other problems that are not present, or are less serious, in thread-based systems.

Many systems support both threads and interrupts, resulting in error-prone mixed concurrency models where different kinds of locks (e.g., thread mutexes and interrupt spinlocks) are required to defend against preemption by different kinds of concurrent activities. Heterogeneous concurrency is common in embedded systems and also in general-purpose operating systems such as Linux and Windows. Our previous work includes TSL [19], a step towards a principled way to reason about asymmetrical preemption relations in these kinds of systems.

In constrained embedded systems stack memory overflow due to nested interrupts is a real possibility, and is a source of difficult memory corruption errors. Previous work by us [18], by Brylow et al. [4], and by Chatterjee et al. [6] addresses the problem of detecting stack overflows in interrupt-driven programs.

Finally, problems can be caused by the fact that interrupts run at the highest priority: if interrupts arrive too frequently, non-interrupt work may be starved. Purely offline verification of the absence of overload problems is often not possible since the rate of arrival of interrupt requests may depend on, for example, another node on the network. Rather, online scheduling solutions [17] are required.

5. RELATED WORK

This section presents a brief survey of the literature on the semantics of interrupts, and on verification of interrupt-driven programs.

Interrupt semantics. Most attempts to fit interrupts into a system's semantics focus on making interrupts look like threads. For example, Hills [13] provides perhaps the clearest existing description of the problems associated with interrupts, and proposes as a solution a model where the interrupt handler is simply an atomic stub that awakens some threads. These threads perform the processing that would normally run in interrupt context. Similarly, Leyva-del-Foyo and Mejia-Alvarez's work [8], the Nemesis OS [15], TimeSys Linux [21], and Solaris [14] all take the interrupts-as-threads approach.

Race checking for interrupt-driven systems. nesC [10], the programming language for TinyOS applications, checks for race conditions in interrupt-driven applications by looking for variables that are accessed by interrupt handlers and that are not protected by atomic execution. Henzinger et al. [12] refine nesC's race checking using an analysis that can show that some non-atomic variable accesses are safe. Mercer and Jones [16] exploit GDB's state saving and restoring capabilities to model check interrupt-driven embedded code. The SLAM [2] model checker and the RacerX [9] race condition detector are used to find bugs in kernel code, including interrupt handlers.

6. CONCLUSION

There has been much recent research on tools to find race conditions in multithreaded code. A natural question to ask is: Can these tools be leveraged to check interrupt-driven operating system and embedded system code and, if so, how? We have described a few extensions to thread checking tools that will enable them to check interrupt-driven systems, and also presented examples of source-to-source translations that turn interrupt-driven code into semantically equivalent thread-driven code.

7. REFERENCES

[1] Atmel, Inc. ATmega128 datasheet, 2002. `http://www.atmel.com/atmel/acrobat/doc2467.pdf`.

[2] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Proc. of the 1st EuroSys Conf.*, Leuven, Belgium, April 2006.

[3] Larry Barello. The AvrX real time kernel, 2004. `http://barello.net/avrx`.

[4] Dennis Brylow, Niels Damgaard, and Jens Palsberg. Static checking of interrupt-driven software. In *Proc. of the 23rd Intl. Conf. on Software Engineering (ICSE)*, pages 47–56, Toronto, Canada, May 2001.

[5] D. Butenhof. *Programming with POSIX Threads.* Addison Wesley, 1997.

[6] Krishendu Chatterjee, Di Ma, Rupak Majumdar, Tian Zhao, Thomas A. Henzinger, and Jens Palsberg. Stack size analysis for interrupt-driven programs. In *Proc. of the 10th Static Analysis Symp.*, pages 109–126, San Diego, CA, June 2003.

[7] Crossbow Technology, Inc. `http://xbow.com`.

[8] Luis E. Leyva del Foyo and Pedro Mejia-Alvarez. Custom interrupt management for real-time and embedded system kernels. In *Proc. of the Workshop on Embedded and Real-Time Systems Implementation (ERTSI)*, Lisbon, Portugal, December 2004.

[9] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In

Proc. of the 19th ACM Symp. on Operating Systems Principles (SOSP), Bolton Landing, NY, October 2003.

[10] David Gay, Phil Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 1–11, San Diego, CA, June 2003.

[11] John Hatcliff, Xinghua Deng, Matthew B. Dwyer, Georg Jung, and Venkatesh Prasad Ranganath. Cadena: an integrated development, analysis, and verification environment for component-based systems. In *Proc. of the 25th Intl. Conf. on Software Engineering (ICSE)*, pages 160–173, Portland, OR, May 2003.

[12] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with Blast. In *Proc. of the 10th Intl. Workshop on Model Checking of Software (SPIN)*, pages 235–239, Portland, OR, May 2003.

[13] Ted Hills. Structured interrupts. *ACM SIGOPS Operating Systems Review*, 27(1):51–68, January 1993.

[14] Steve Kleiman and Joe Eykholt. Interrupts as threads. *ACM SIGOPS Operating Systems Review*, 29(2):21–26, April 1995.

[15] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[16] Eric G. Mercer and Michael D. Jones. Model checking machine code with the GNU debugger. In *Proc. of the SPIN Workshop on Model Checking of Software*, San Francisco, CA, August 2005.

[17] John Regehr and Usit Duongsaa. Preventing interrupt overload. In *Proc. of the 2005 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Chicago, IL, June 2005.

[18] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. In *Proc. of the 3rd Intl. Conf. on Embedded Software (EMSOFT)*, pages 306–322, Philadelphia, PA, October 2003.

[19] John Regehr, Alastair Reid, Kirk Webb, Michael Parker, and Jay Lepreau. Evolving real-time systems using hierarchical scheduling and concurrency analysis. In *Proc. of the 24th IEEE Real-Time Systems Symp. (RTSS)*, Cancun, Mexico, December 2003.

[20] David A. Solomon and Mark E. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, third edition, 2000.

[21] TimeSys Corporation. TimeSys Linux. `http://timesys.com/`.