

Task/Scheduler Logic: Reasoning about Concurrency in Component-Based Systems Software

Alastair Reid John Regehr
School of Computing, University of Utah
Salt Lake City, UT 84112–9205, USA
{reid, regehr}@flux.utah.edu
<http://flux.utah.edu/>

Abstract

Although component-based software development promises increased reuse and faster development time, it has proven difficult to build component-based systems software. One obstacle is that the concurrency structure in systems software tends to be complex. First, instead of a single scheduler, there is a hierarchy of schedulers: the processor schedules interrupts, the OS schedules software interrupts and threads, and threads run event loops. This gives rise to many different execution environments, each with its own restrictions on actions that can be taken by code running in it. Second, the preemption relationships between these execution environments are often asymmetric: an interrupt handler can preempt a thread but not vice versa. This results in an asymmetric pattern of locking where low priority code must protect against high priority code but not vice versa. This situation is rare in other application domains but common in systems software.

We have developed Task/Scheduler Logic (TSL) for reasoning about component-based systems software. We show that TSL can be used to reason about race conditions, illegal lock usage, and redundant or unnecessary synchronization points in component-based systems software. Further, we show that TSL can realistically be applied to large, complex systems.

1. Introduction

Component-based systems software, as exemplified by systems such as Koala [20], Click [12], TinyOS [10], and the OSKit [5], is motivated by a number of requirements: short time to market, evolvability to meet changing needs, long-term software reuse, and small memory footprint in delivered systems. Another requirement of systems software, high dependability, is potentially undermined both by the diversity of interconnections that components permit and by the highly concurrent nature of systems software.

Concurrency and complex interconnections are particularly difficult to reason about in systems software because typically there are many different *execution environments* such as interrupt handlers, software interrupts, kernel threads, and user-level threads. An execution environment is a set of rules that particular pieces of code must obey. Some rules restrict actions that code may take and other rules describe actions that must be taken to safely interact with other environments. For example, an interrupt handler must never block and a thread must disable interrupts before interacting with a data structure that the thread shares with an interrupt handler.

Existing ways to reason about concurrency in programs (for example, [1, 4]) typically assume a homogeneous execution environment, e.g., preemptive multi-threading with blocking locks. Additional machinery is required to permit reasoning about concurrency in the presence of multiple execution environments. One possible approach would be to construct a complete set of rules for code executing in a particular set of execution environments, such as the environments that make up the Linux kernel. We have taken a different approach based on the insight that the global problem of reasoning about execution environments in a large system can be broken down into many small local problems by exploiting the *hierarchical scheduling* relations between different parts of the system. That is, we exploit the fact that if, for example, a scheduler providing an event-processing loop runs in the context of an interrupt handler, it *inherits* a number of properties from the interrupt handler: a very high priority, an inability to block, etc. Similarly, the children of the event scheduler (i.e., the event handlers) inherit not only interrupt properties, but also event handler properties: for example, they are scheduled non-preemptively with respect to other event handlers run by the same event loop.

This paper describes our *Task/Scheduler Logic* (TSL): a logic for reasoning about concurrency in systems containing multiple execution environments. TSL permits:

- the detection of components with incorrect or insufficient synchronization code (race conditions) even across

execution environments,

- the elimination of redundant synchronization, e.g., because a component that implements correct locking happens to be instantiated in a scenario where concurrent access to a protected resource is impossible,
- inference of the set of synchronization primitives that can be used to satisfy the atomicity requirements of a particular critical section, and
- detection of violations of some types of execution environment requirements, e.g., that interrupt handlers avoid blocking.

In our experience the vast majority of concurrency issues in systems software are simple race conditions that are fixed by inserting or moving the placement of a simple lock. We see relatively few uses of more sophisticated approaches such as lock-free synchronization [9], locks that allow multiple readers but only one writer, etc. — these tend to be employed only in a few performance hot spots. We also find that most locks in systems software are “static”: the number allocated and the use made of them is determined at build time rather than at run time. Consequently, we have chosen to keep TSL easy to use by excluding features needed to model more advanced concepts. Rather than striving to catch all concurrency errors no matter what burden this places on the programmer, TSL is designed to find the majority of concurrency errors with relatively little effort.

Our long term vision is that, given a convenient way to reason about concurrency in the presence of diverse execution environments, it will be possible to develop systems software components that are as agnostic as possible with respect to their execution environment. This is in sharp contrast to today’s software where almost all software components target a particular environment or set of environments. For example, Java classes assume multithreading with blocking locks, and loadable kernel modules for Linux hard-code uses of interrupt handlers, spinlocks, kernel threads, etc. making it difficult to reuse this code. Such specificity can increase maintenance costs; for instance, developers currently waste valuable time reimplementing Linux device drivers for use in RTLinux [21].

This paper makes the following contributions.

- We identify and describe a problem in applying existing concurrency models to component-based systems software and outline a solution (Section 2).
- We present TSL, a logic for concurrency in component-based systems software that includes explicit support for hierarchical scheduling and asymmetric preemption relations (Section 3).
- We provide TSL specifications for four representative classes of schedulers (Section 4).
- We describe how reasoning in TSL may be automated for use in large, complex systems (Section 5).

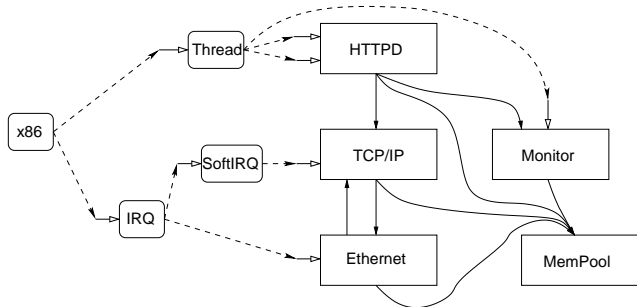


Figure 1. A simple component-based monitoring system with Web interface (right) and its scheduling hierarchy (left)

- We demonstrate the applicability of TSL to a real system (Section 6).

We consider related work in Section 7 and conclude in Section 8.

2. The problem

This section motivates TSL using an example that illustrates some of the problems typically encountered when building systems software out of components. The example also shows how the execution environment structure of a system is independent of its traditional modular decomposition. In other words, a component will often contain code that runs in multiple environments, and an execution environment will usually run code from multiple components.

The execution environments for a particular system are determined by examining each component’s relation to a hierarchy of schedulers such as that shown in Figure 1. This figure is in two parts: a hierarchy of schedulers on the left and a collection of application components on the right. The two parts are joined by dashed lines where a scheduler invokes a *task*. In this paper a *task* is anything run directly by a scheduler such as a thread, a software interrupt, a hardware interrupt, a signal handler, etc.

The schedulers are arranged in a hierarchy indicated by dashed lines. At the top of the scheduler hierarchy (furthest to the left) is a priority scheduler (implemented by hardware in the CPU) that prioritizes interrupts over other code. Rather than directly running tasks this scheduler invokes two other schedulers: the interrupt scheduler and the thread scheduler. The interrupt scheduler, implemented mostly in hardware, is a priority scheduler that supports interrupt request levels. Underneath the interrupt scheduler, a software interrupt scheduler (implemented in software) executes *bottom-half* handlers non-preemptively. Broadly speaking an execution environment is either top-half, meaning that it runs in a thread context, or bottom-half, meaning that it runs in an interrupt context. Software interrupt tasks run at lower priority than hardware interrupts but have

higher priority than threads. The thread scheduler, implemented in software, is a conventional preemptive thread scheduler.

In Figure 1 the application component interconnection graph is depicted to the right of the scheduling hierarchy. The application is an embedded system that is designed to (1) monitor a device such as a greenhouse ventilation system and (2) make information about the system available to HTTP clients. Upon request the HTTPD component retrieves data from the *Monitor* component and sends it out on the network using the *TCP/IP* and *Ethernet* components. All components make use of a memory allocator. For simplicity we have omitted many infrastructure components: building a system like this with an operating system kit such as the OSKit [5] requires an order of magnitude more components. The component interconnections are of two types: explicit and implicit. *Explicit connections* are due to a component importing an interface provided by another component; the connection from *TCP/IP* to *Ethernet* is of this form. *Implicit connections* are due to a component passing a function pointer to another component; the connection from *Ethernet* back to *TCP/IP* is caused by *TCP/IP* providing a packet-processing callback function to *Ethernet* when it opens an Ethernet device.

This figure omits an important detail: most components contain *resources* that must be protected from being used by two tasks simultaneously using *locks*. For the purposes of this paper, we use “lock” to refer to a range of synchronization primitives: mutexes, disabling interrupts, or using a spinlock. We exclude from consideration synchronization primitives that do not rely on mutual exclusion, such as non-blocking synchronization and reader-writer locks.

Systems like this, that are made up of many components and in which multiple execution environments co-exist, raise a number of questions that TSL provides the means to answer:

- Are there any “classic systems errors” such as using a thread lock in an interrupt handler?
- Is the system free of race conditions? The problem is not just the usual problem of making sure that all resources are protected by locks, but of making sure that all resources are protected by the right locks. For example, thread locks provide no protection against interrupt handlers.
- Are there lock acquisitions that are unnecessary, that can be removed as a performance optimization?
- Given a resource to be protected and a proposed place in the callgraph to acquire a lock, which locks suffice to eliminate races?

To answer these questions we need:

- a method to calculate the properties of a hierarchy of schedulers from properties of the individual schedulers;
- a method to calculate the concurrency properties of a

graph of components from properties of the individual components;

- specifications of the schedulers in question; and
- specifications of the flow of control, use of locks, and use of resources by the components.

The rest of this paper describes these methods and specifications in detail. Of course there are many other interesting questions to ask about a system. For example, when there are several acceptable choices for a lock, which one is best? Is the resulting system free of deadlock and livelock? Will the real-time tasks in the system meet their deadlines? The answers to these questions are beyond the scope of TSL.

We have built many systems like the one shown in Figure 1 using the OSKit, a toolkit of operating system components, and Knit [19], a component definition and linking language for systems software written in C. Our experience revealed one other important consideration: our approach must be simple enough that the people who currently build such systems can use it. We believe it is reasonable to expect some sophistication in subtle concurrency issues from those writing a scheduler, but we can expect at most a basic understanding of the issues from those writing components and, especially, from those configuring systems from those components to build real systems. This led us to sacrifice some of the potential power of TSL in order to keep it simple.

3. Task/Scheduler Logic

Our *Task/Scheduler Logic* (TSL) is designed to allow modular description of the concurrency scenarios that commonly occur in systems software. Since some of these issues are rather subtle and may be unfamiliar, we shall build up to TSL in a series of steps starting with tasks and resources and then adding details due to locks, callgraphs, and scheduler hierarchies.

3.1. Tasks and resources

We start with a simple model of concurrent systems that consist of a uniprocessor scheduler, a set of tasks, and a set of resources accessed by those tasks. *Tasks* are flows of control through components; in general a task is explicitly invoked by a *scheduler* that is implemented either in hardware (e.g., an interrupt controller) or software (e.g., a traditional thread scheduler). All tasks have a well-defined entry point and most tasks also *finish*. Some tasks encapsulate an infinite loop and these never finish.

Some notation that we shall use throughout this paper: The variables t, t_1 , etc. range over tasks and the variables r, r_1 , etc. range over resources. We write $t \rightarrow r$ if a task t uses a resource r . We write $t_1 \not\prec t_2$ if a task t_2 can *preempt* a

task t_1 . That is, $t_1 \not\prec t_2$ if t_2 can start to run after t_1 starts to execute but before t_1 finishes.

For example, an interrupt handler t_{irq} can preempt the main task t_{main} but not vice versa: $t_{main} \not\prec t_{irq}$. The definition of $\not\prec$ is asymmetric: $t_1 \not\prec t_2$ does not imply $t_2 \not\prec t_1$. This asymmetry lets us express a number of common relations between tasks:

- That t_1 is strictly higher priority than t_2 on a uniprocessor priority scheduler is modeled by $\neg(t_1 \not\prec t_2) \wedge (t_2 \not\prec t_1)$.
- That t_1 and t_2 are mutually preemptible is modeled by $t_1 \not\prec t_2 \wedge t_2 \not\prec t_1$.
- That t_1 and t_2 cannot run simultaneously is modeled by $\neg(t_1 \not\prec t_2) \wedge \neg(t_2 \not\prec t_1)$.

It might seem more obvious to take the notion of two tasks running simultaneously as a primitive concept in the logic but, since this notion is symmetric, it cannot be used to describe asymmetric relationships.

We provisionally define a race condition to occur if two tasks access the same resource and one task can preempt the other.

$$\begin{aligned} \text{race}(t_1, t_2, r) &\stackrel{\text{def}}{=} t_1 \rightarrow r \\ &\wedge t_2 \rightarrow r \\ &\wedge t_1 \neq t_2 \\ &\wedge t_1 \not\prec t_2 \end{aligned}$$

This definition will be refined in Section 3.2.

3.2. Locks

Locks are used to eliminate race conditions by eliminating some preemption relations. We shall use the variables l , l_1 , etc. to range over locks and the variables L , L_1 , etc. to range over sets of locks. Generalizing the definition of $\not\prec$ to take locks into account, we write $t_1 \not\prec_L t_2$ if parts of a task t_2 which need a set of locks L can start to run while a task t_1 holds L . For example, if holding a thread lock lk blocks a task t_2 from entering critical sections protected by lk , then $t_1 \not\prec_{lk} t_2$ or, if disabling interrupts with the cpu lock prevents any part of a task t_2 from running, then $t_1 \not\prec_{cpu} t_2$.

We write $t \rightarrow_L r$ if t holds a set of locks L while using a resource r . For example, if a task eth accesses a memory resource r_{mem} with interrupts disabled (i.e., while holding a lock cpu) we would write $eth \rightarrow_{cpu} r_{mem}$. If the same task also accesses the same resource without disabling interrupts we would write $eth \rightarrow_{cpu} r_{mem} \wedge eth \rightarrow_{\emptyset} r_{mem}$.

Locks satisfy three important properties. First, if t_1 can be preempted while holding a set of locks, then t_1 can be preempted while holding fewer locks:

$$t_1 \not\prec_{L_1} t_2 \wedge L_1 \supseteq L_2 \Rightarrow t_1 \not\prec_{L_2} t_2$$

Second, if t_1 can be preempted by t_2 while holding either a set of locks L_1 or a set of locks L_2 , then t_1 can be preempted by t_2 while holding both sets of locks.

$$t_1 \not\prec_{L_1} t_2 \wedge t_1 \not\prec_{L_2} t_2 \Rightarrow t_1 \not\prec_{L_1 \cup L_2} t_2$$

Finally, preemption is a transitive relation: if t_1 can be preempted by t_2 and t_2 can be preempted by t_3 , then t_1 can be preempted by t_3 .

$$t_1 \not\prec_{L_1} t_2 \wedge t_2 \not\prec_{L_2} t_3 \Rightarrow t_1 \not\prec_{L_1 \cap L_2} t_3$$

We generalize the definition of a race condition to take locks into account as follows:

$$\begin{aligned} \text{race}(t_1, t_2, r) &\stackrel{\text{def}}{=} t_1 \rightarrow_{L_1} r \\ &\wedge t_2 \rightarrow_{L_2} r \\ &\wedge t_1 \neq t_2 \\ &\wedge t_1 \not\prec_{L_1 \cap L_2} t_2 \end{aligned}$$

That is, a race can occur if two tasks t_1 and t_2 use a common resource r with some common set of locks $L_1 \cap L_2$, and if t_2 can preempt t_1 even when t_1 holds those locks. For example, if some task t_1 uses a resource r with locks $\{l_1, l_2, l_3\}$ and another task t_2 uses r with locks $\{l_2, l_3, l_4\}$ then they hold locks $\{l_2, l_3\}$ in common and a race occurs iff $t_1 \not\prec_{\{l_2, l_3\}} t_2$.

Tasks may implicitly hold certain locks; for example, in some systems interrupt handlers run with interrupts disabled. Taking these implicit locks into account is a matter of specification and they need no specific support in TSL.

We do not distinguish between those locks that *block* a task (e.g., thread locks) and those that merely prevent preemption by another task (e.g., disabling interrupts). While they are different from the point of view of liveness, they behave identically in terms of race conditions: they prevent a task from entering a critical section.

3.3. Callgraphs

Since the implementation of a task is usually split across a number of components we must describe enough of the function callgraph to enable the set of resources accessed by a task to be constructed from a set of components and their interconnections. We shall use the variables f , f_1 , etc. to range over functions. We write $t \downarrow f$ if f is the *entry point* of a task t ; $f_1 \downarrow_L f_2$ if f_1 acquires locks L before calling a function f_2 ; and, $f \rightarrow_L r$ if a function f holds a set of locks L while using a resource r .

For example, in Figure 1 we showed a connection from the *TCP/IP* component to the *Ethernet* component meaning that any function exported by the *TCP/IP* component could potentially call any function exported by the *Ethernet* component. We also showed a connection from

the *Ethernet* component to the *TCP/IP* component because the *TCP/IP* component registers a callback function with the *Ethernet* component, which is called when an Ethernet packet is received. In both cases we are using a (crude) approximation of the actual dynamic callgraph: we are omitting all the internal callgraph structure of the components and we are including calls that could not possibly happen in practice. That is, we assume that a function call is made unless it is provably not made. By overestimating the callgraph we ensure that we will not miss any potential race conditions, but we run the risk of seeing false positives. We discuss the problem of false positives further in Section 6 when we consider the application of TSL to large examples. Similarly, it is safe to treat a set of independent resources as a single resource and to assume that a function uses a resource without a given lock unless we can prove that it does not do so.

We extend this notation to cover chains of calls as follows. Note that locks accumulate down the callgraph.

$$f_1 \downarrow_{L_1} f_2 \wedge f_2 \downarrow_{L_2} f_3 \Rightarrow f_1 \downarrow_{L_1 \cup L_2} f_3$$

With this notation, we can define what it means for a task to use a resource. A task t uses a resource r if the task's endpoint f_1 calls a function f_2 that uses r :

$$t \rightarrow_{L_1 \cup L_2} r \stackrel{\text{def}}{=} \exists f_1, f_2. \quad \begin{array}{l} t \downarrow f_1 \\ \wedge f_1 \downarrow_{L_1} f_2 \\ \wedge f_2 \rightarrow_{L_2} r \end{array}$$

3.4. Hierarchical scheduling

The model developed thus far is suitable for describing simple, monolithic preemptive and non-preemptive schedulers. To concisely describe the structure of real systems software, however, we need to acknowledge the hierarchical relationship of the hardware and software schedulers contained in the system.

When we build scheduler hierarchies we find that a *scheduler* at one level can be regarded as a *task* scheduled by the level above it. For example, a software interrupt scheduler runs a number of tasks below it in the hierarchy but, to the layer above, it is just another task to be scheduled by the hardware interrupt scheduler.

We treat schedulers as a special kind of task and use the variables t, t_1 , etc. to range over both tasks and schedulers. We write $t_1 \triangleleft t_2$ if a scheduler t_1 is above a scheduler or task t_2 in the hierarchy; \triangleleft is the *parent* relation. The ability to preempt or to be preempted is inherited down the hierarchy: if two tasks can preempt each other then so can their children:

$$\begin{array}{l} t_1 \triangleleft t_2 \wedge t_1 \not\downarrow_L t_3 \wedge t_1 \neq t_3 \Rightarrow t_2 \not\downarrow_L t_3 \\ t_1 \not\downarrow_L t_2 \wedge t_1 \neq t_2 \wedge t_2 \triangleleft t_3 \Rightarrow t_1 \not\downarrow_L t_3 \end{array}$$

A consequence of these axioms is that if the *nearest common scheduler* to two tasks is a non-preemptive scheduler, then neither task can preempt the other. Results such as this are useful, for example, when using TSL to prove that a lock is unnecessary in a particular composition of components: mutually non-preemptible tasks require no locks to protect resources that (only) they share.

Each lock is associated with a particular scheduler. We write $t \dashv l$ if a scheduler t provides a lock l and require that each lock is provided by exactly one scheduler.

One of the classic errors in systems programming is for a routine that runs in a bottom-half execution environment (i.e., an interrupt handler or bottom-half handler) to attempt to acquire a blocking lock. This is generally a fatal error for the simple reason that blocking is implemented in a thread scheduler and therefore applies only to threads. Or, in other words, the thread scheduler does not “know” enough about interrupts to be able to manipulate their state in such a way that they are put to sleep.

Using TSL we can check for a generalized version of the “blocking in interrupt” problem by ensuring that tasks only acquire locks provided by their (possibly transitive) parents in the scheduling hierarchy. To see why being a descendent is necessary, consider what happens when a scheduler's grandchild blocks while attempting to acquire a lock. The scheduler puts its grandchild to sleep not by acting on the grandchild directly but, rather, by putting its immediate child (the task's parent) to sleep. If the task to be put to sleep is not a descendent, however, there is no task which the scheduler is able to act on.

We formalize this generalization as follows:

$$\begin{array}{l} \text{illegal}(t, l) \stackrel{\text{def}}{=} \exists t_1. \quad \begin{array}{l} t_1 \dashv l \\ \wedge \neg(t_1 \triangleleft^+ t) \\ \wedge t \rightarrow_L r \\ \wedge l \in L \end{array} \end{array}$$

The *ancestor* relation \triangleleft^+ is the transitive closure of \triangleleft .

A consequence of the definitions of *race* and *illegal* is that to eliminate a race condition one must use a lock provided by a scheduler that is a common ancestor to all tasks that access the resource.

4. Specifying schedulers with TSL

This section shows how to model a number of common schedulers in TSL.

4.1. An event scheduler

Many systems include one or more simple event schedulers that execute a chain of handlers to respond to an event, where each handler runs to completion before another handler is invoked. These are the easiest systems to specify in

TSL since there are no locks and no preemption. All we have to specify is the scheduler hierarchy:

$$Event(t, t_1, \dots, t_m) \stackrel{\text{def}}{=} t \triangleleft t_1 \wedge \dots \wedge t \triangleleft t_m$$

4.2. A generic preemptive scheduler

Common preemptive schedulers potentially permit any task they schedule to preempt any other task they schedule. These schedulers include round-robin schedulers, time-sharing schedulers such as those found in UNIX and Windows, and earliest-deadline-first schedulers for real-time systems. For these schedulers, high priority tasks might be *more likely* to preempt low priority tasks but low priority tasks can preempt high priority tasks:

$$\begin{aligned} Pre(t, t_1, \dots, t_m, L) \stackrel{\text{def}}{=} & t \triangleleft t_1 \wedge \dots \wedge t \triangleleft t_m \\ & \wedge \forall l \in L. t \dashv\!\circ l \\ & \wedge \forall i, j. t_i \not\triangleleft t_j \end{aligned}$$

That is, a task can be preempted by those parts of sibling tasks which have no locks in common.

4.3. A strict priority scheduler

One of the most common schedulers encountered in systems software is a priority scheduler where high priority tasks can preempt low priority tasks but not vice versa. We call this a *strict* priority scheduler because it always permits high priority code to run to completion before running lower priority code. The most common example of a strict priority scheduler is the interrupt mechanism provided in microprocessors where one or more interrupt handlers are given priority over all user-mode code.

We specify a priority scheduler t with child tasks t_1, \dots, t_m where t_1 is lowest priority and t_m is highest priority as follows:

$$\begin{aligned} SPri(t, t_1, \dots, t_m, L) \stackrel{\text{def}}{=} & t \triangleleft t_1 \wedge \dots \wedge t \triangleleft t_m \\ & \wedge \forall l \in L. t \dashv\!\circ l \\ & \wedge \forall i, j. i < j \Rightarrow t_i \not\triangleleft t_j \end{aligned}$$

That is, a task can be preempted by those parts of higher-priority sibling tasks which have no locks in common.

4.4. A non-strict priority scheduler

Non-strict priority schedulers are those that do not always run high-priority tasks to completion before running low-priority tasks. For example, a priority-based scheduler in a real-time OS will run a lower-priority task while a high-priority task is blocked on a disk access or other resource. These schedulers don't satisfy the specification in the previous section because executing a low priority task after a

high priority task starts and before it completes counts as preemption and so breaks the last part of the specification.

The simplest solution is to treat such schedulers as generic preemptive schedulers (Section 4.2). More accurate results can be had by recognizing that only those tasks that access blocking resources (thread locks, disk, network, etc.) can be preempted:

$$\begin{aligned} NSPri(t, t_1, \dots, t_m, L) \stackrel{\text{def}}{=} & SPri(t, t_1, \dots, t_m, L) \\ & \wedge \forall i, j, r. t_i \rightarrow_l r \Rightarrow t_i \not\triangleleft_l t_j \end{aligned}$$

5. Reasoning about TSL specifications

TSL is designed to be easy to reason about both by humans and machines. Our original implementation was a fairly direct translation of the TSL axioms into Prolog Horn clauses. This let us verify the design but suffered from performance problems and, more seriously, could not handle cyclic callgraphs. Our current implementation is a simple forward-chaining evaluator that takes a specification for a system and derives all possible consequences of the TSL axioms. This is faster than the original implementation because forward-chaining has the effect of caching previously derived results whereas the (backward-chaining) Prolog implementation would repeatedly recalculate the callgraph. Unlike the original implementation, termination is not a problem because there are only a finite number of facts that can be stated about a finite set of schedulers, tasks, locks, functions, and resources. There is potential for a state space blowup especially when building the callgraph. In practice, however, there are typically five or fewer schedulers, five or fewer kinds of locks, and our component models ignore internal structure so we only consider functions exported from components. For example, in the full version of our example monitoring system consisting of 116 components, we consider 1059 functions, 5 tasks, and 2 kinds of locks. From this we derive 3400 calls relations from a task t to a function f (i.e., of the form $t \downarrow_L f$).

5.1. Revisiting the web server example

Consider again the example from Figure 1. Before we can analyze it with TSL we must label all the tasks (we name them $h1$, $h2$, t , e , and m), label all the schedulers (we name them $x86$, $thread$, irq , and $softirq$), and categorize each of the schedulers. We must also add resources (named rh , rb , rt , re , $rmon$, and $rmem$) and add their uses into the callgraph, add locks (named cpu and lk), attach locks to the scheduler that provides them, and label edges in the callgraph with locks acquired before the calls are made. Figure 2 shows these labels and relations. The example includes some errors in the use of locks that we shall discover using TSL. The schedulers are, of course,

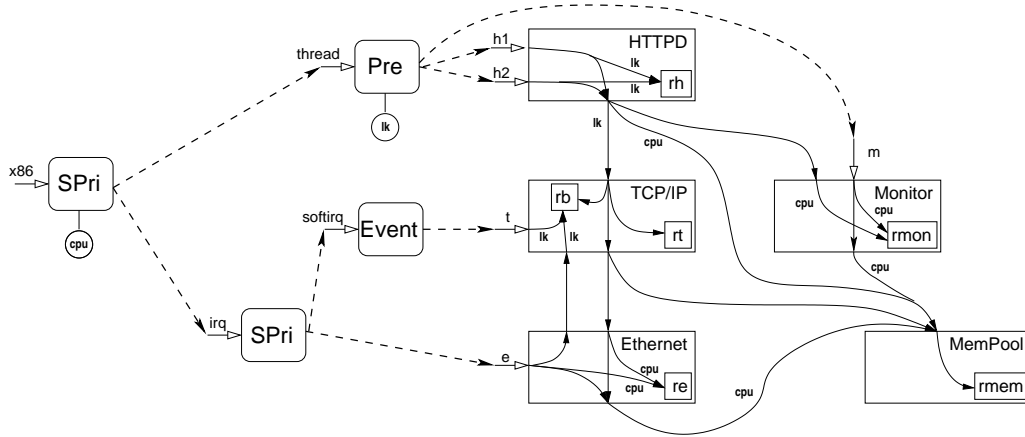


Figure 2. The monitoring system example revisited

also components, but to keep the example to a reasonable size we do not show their resources, locks used to protect those resources, calls to the functions they export, etc.

Using these labels we can describe the scheduler hierarchy concisely as follows:

$$\begin{aligned}
 SPri(x86, irq, thread, \{cpu\}) &\wedge \\
 SPri(irq, e, softirq, \emptyset) &\wedge \\
 Event(softirq, t) &\wedge \\
 Pre(thread, h1, h2, m, \{lk\}) &
 \end{aligned}$$

5.2. Checking for illegal locking

To detect cases of illegal locking our implementation computes a list of all the resources accessed by each task with a given set of locks. For example, from the callgraph and locks shown in the figure we generate the following table:

$h1, h2$	\rightarrow_{lk}	$\{rh, rt, rb, rmem\}$
$h1, h2$	$\rightarrow_{lk,cpu}$	$\{re, rmem\}$
$h1, h2$	\rightarrow_{cpu}	$\{rmon, rmem\}$
m	\rightarrow_{cpu}	$\{rmon, rmem\}$
t	\rightarrow_{lk}	$\{rb\}$
e	\rightarrow_{lk}	$\{rb\}$
e	\rightarrow_{cpu}	$\{re, rmem\}$

Given this table, it is straightforward to apply the definition of *illegal* to generate a list of all the illegal lock uses:

$$\begin{aligned}
 &illegal(t, lk) \\
 &illegal(e, lk)
 \end{aligned}$$

Both problems are caused by using the lock lk to protect the resource rb which is accessed by hardware and software interrupts and can be resolved by changing the lock to cpu .

Although these errors can be easily found by inspecting Figure 2, a real system would have an order of magnitude more components and interconnections. Furthermore, in our experience the large number of indirect connections makes errors of this type all too common.

5.3. Checking for races

Our primary concern is with the detection of race conditions: two tasks accessing a resource simultaneously. TSL provides a list of potential race conditions and can be used to examine the scheduler hierarchy and call chain to diagnose the cause of problems.

For example, from the scheduler hierarchy we can deduce that the following preemption relations hold:

$$\begin{aligned}
 h1, h2, m &\not\prec \emptyset & h1, h2, m \\
 h1, h2, m &\not\prec \emptyset & t \\
 t &\not\prec \emptyset & e
 \end{aligned}$$

Combining this with resource use and the definition of *race*, we obtain the following race conditions.

$$\begin{aligned}
 race(h1, h2, rmem) & \quad race(h2, h1, rmem) \\
 race(h1, m, rmem) & \quad race(h2, m, rmem) \\
 race(h1, e, rmem) & \quad race(h2, e, rmem)
 \end{aligned}$$

These can be fixed by acquiring the cpu lock when calling from *TCP/IP* to *MemPool*.

5.4. Eliminating redundant locks

In large, complex systems and when reusing parts of other systems, it is not uncommon to use a lock to “protect” a resource when, in fact, only one task uses the resource. Such cases can be efficiently detected in TSL by looking for resources protected by a lock where the removal of the lock does not cause any race conditions.

The system in Figure 2 does not contain any redundant locks. However, consider what would happen if, due to memory constraints, the developer was forced to only instantiate a single thread for the *HTTPD* component. In this case the locks protecting rh could be eliminated as could the thread lock providing atomic access to the top-half of the *TCP/IP* component.

5.5. Synchronization inference

For developers and users of components it is often easier to determine that a lock is required than to decide which lock to use. TSL can be used to list locks that could safely be used in a given context.

Instead of using an actual lock provided by a particular scheduler, the specification uses a *virtual lock*. To find an appropriate *physical lock*, our implementation first determines what the lock protects against by replacing the virtual lock with a “null lock” that provides no protection at all. Our implementation then returns a list of all locks that would prevent the preemptions between the tasks involved in the races.

For example, if the *cpu* locks in the *Monitor* component in Figure 2 were declared as virtual locks then TSL would inform us that acceptable lock implementations are *cpu* and *lk*.

TSL returns all valid locks to the developer rather than trying to choose the “best” lock. There are several interrelated reasons for this. First, it is often impossible to statically infer the performance overhead of a particular lock since the number of calls to the lock implementation is a dynamic property of the system. Second, it is difficult or impossible to statically determine how long the lock will be held. Hold time is important because, for example, in most systems it is perfectly acceptable to disable interrupts for 15 μ s but completely unacceptable to disable them for 150 ms. Finally, different systems have different high-level goals and choices of locking often have a global effect on system performance. One choice of lock might maximize system throughput while another maximizes the chances of a real-time task meeting its deadlines.

6. Experience using TSL

Our primary use of TSL has been in reasoning about systems built from the OSKit [5] using Knit [19]. The OSKit is a collection of components for building embedded systems and operating systems. Many of the components are derived from the Linux, FreeBSD, and Mach kernels and range in size from large, such as a TCP/IP stack derived from FreeBSD (over 18,000 lines of non-blank, non-comment code), to small, such as serial console support (less than 200 lines of code). Knit is a language and toolset for defining components for use in systems software. Knit allows us to work with C and assembly code, lets us clearly identify the imports and exports of a component, eliminates naming clashes between unrelated components, allows multiple instantiation of components, and automatically determines the order in which to initialize and finalize components.

6.1. Annotating components

We extended Knit syntax to include per-component TSL annotations and modified the Knit compiler to collect together all the annotations on the components making up a system. Knit also substitutes the names of component instances within the annotations to reflect component instantiation and interconnection. We then annotated each component with TSL specifications. We shall consider the form of these annotations in detail because the effectiveness of TSL depends on the effort required to annotate the components initially and to maintain those annotations over time.

Since most components are not schedulers, the hard work lies in annotating components with their callgraphs, lock usage, and resource usage. Since we are annotating components by hand we sought the simplest model that would yield useful results. We achieve this by making up to three passes over each component.

In the first pass, we model the mutable state of each component as a single resource. We use a very crude callgraph where every function exported from the component (including initializers and finalizers) calls every other function (including imports) and also uses the resource. In addition, we assume that any function pointer passed into a component is called by every function in the component and that any function pointer passed out of the component calls every function in the component and access the resource. A typical set of annotations is the following which describes the callgraph, declares a resource, and describes uses of the resource:

```
note "calls(${exports+inits+finis},
           ${imports}).";
note "resource(${self}).";
note "uses(${exports+inits+finis},${self}).";
```

Generating these annotations is a simple, repetitive task requiring only the types of the component interfaces.

In the second pass, we refine the models of the small number of components that directly interact with their execution environments. Typically this involves declaring any tasks they define, adding lock usage, or splitting the resource into separate resources. As far as possible, we do this based on the well-known and/or documented behavior of the component. For example, the FreeBSD kernel allows at most one process to execute in the kernel at a time, so, in the absence of evidence to the contrary, we assume that locks inside components derived from FreeBSD are only used to protect against interrupt handlers and software interrupts. Relying on documentation is, perhaps, not as reliable as studying the components ourselves but our purpose is not to verify or fix the correctness of the components but, rather, to use the components correctly.

In the third pass, we use Knit to produce complete TSL specifications for some sample systems and use our TSL implementation to generate lists of alleged errors. Initially,

many problems are reported; these can be divided into three categories:

1. Races in “pure” components that define no variables and do not access any hardware devices. We remove the resource from their specification since they have no mutable state.

2. Components designed to be accessed in distinct ways by different execution environments. Again, we study the documented design rules for the environment the component was originally designed for and then split the resources, the callgraph, and the resource use graph to reflect these rules.

3. Components that contain resources and are used by multiple tasks or execution environments, but were not designed to do so. In almost all such cases there is a genuine race condition and we either enclose the component in a wrapper component which acquires a lock before calling the wrappee or we modify the component to acquire a lock directly.

Our example application consists of 190,000 lines of code (including comments and blank lines) distributed over 116 components. We began by applying the “easy” annotations to all 300 components in our repository. That is, we added a coarse-grained annotation to all 300 components and then refined the annotations on roughly 50 components that directly interacted with execution environments. We then ran TSL on our example to generate a list of potential race conditions. This resulted in around 130 reported race conditions. Inspection revealed that most of these involved the resources of a core set of about 20 infrastructure components (like *MemPool* and error reporting and logging) that were accessed by all the tasks in our system. Of these, 13 were pure, 5 involved single-word resources that are modified atomically on most 32-bit processors, 2 used non-blocking synchronization, and the remainder required more careful study and refinement.

6.2. Lessons from using TSL on real code

Besides demonstrating the feasibility of using TSL with large, complex systems, our experience revealed problems in using TSL. We already knew that schedulers are made from components and therefore we usually cannot draw a clear dividing line between the components which contain resources and use locks to protect them and the scheduler hierarchy which provides locks. We had not fully appreciated all the consequences of this though.

A minor consequence is that some schedulers were made up of multiple components. Dealing with this requires a minor extension of TSL to allow the scheduler hierarchy to include function calls as well as just tasks and parent-child relationships.

A more serious consequence is that the scheduler hier-

archy in Figure 2 omits a task: a periodic interrupt that is used to trigger preemption in the thread scheduler. This interrupt accesses the data structures that the thread scheduler uses to keep track of the state of each thread and so these data structures (i.e., resources) must be protected by a lock; the choice of lock is dictated by the location of the interrupt scheduler and the thread scheduler in the hierarchy — just as it is for “ordinary” components. When we drew that task into Figures 1 and 2 there were two arrows entering the thread scheduler and it was initially not clear whether there was a just one parent task for the threads and if so, which it was. This issue was resolved when we recalled that although the periodic interrupt calls into the thread scheduler, the scheduler never runs any of its tasks in interrupt context.

7. Related work

Our work lies at the intersection of four areas: concurrency theory, systems software, programming languages, and component-based software development. There is a great deal of related work in these areas; in this section we compare TSL to what we feel are representative samples.

Concurrency theory: Concurrency languages/logics such as temporal logic [15] and the pi-calculus [16] provide the fundamental concepts and reasoning tools for discussing concurrency. Lamport [13], Owicki and Gries [17], and others have developed specific techniques for proving the correctness of algorithms. In general, approaches based on these works seem to be too detailed and too complex for use with complete systems designed by average programmers. Model checkers such as SPIN [11] and Bandera [2] represent a more promising approach to bringing the benefits of concurrency theory to developers. Model checkers are more powerful than TSL in that they can reason about liveness as well as safety. However, since model checkers lack axioms about hierarchical scheduling relationships, we do not believe that they could reason effectively about concurrency across multiple execution environments.

Systems software: Existing work on hierarchical scheduling [7, 6, 18] is concerned with the distribution of CPU time. In these systems all schedulers are assumed to be generic preemptive schedulers and therefore no useful information about preemption relations can be discovered by analyzing the hierarchy. As far as we know TSL is the first system to recognize and systematically exploit hierarchical scheduling relations that occur in systems code.

Programming languages: The trend towards inclusion of concurrency in mainstream language definitions such as Java and towards strong static checking for errors is leading programming language research in the direction of providing annotations [3, 8] or extending type systems to model locking protocols [4]. These efforts are complementary to the primary contribution of our work on reasoning about

multiple execution environments. They provide significantly finer-grained models of resource access and lock acquisition than do our callgraphs but do not model asymmetric preemption relations, scheduler hierarchies, or multiple execution environments. We believe that TSL and extended type systems would be a very powerful combination.

Component-based software development: Early versions of our Knit toolchain [19] had a primitive mechanism for tracking top/bottom-half execution environments. It modeled a special case of asymmetry in such environments but it did not model locks and locking. So, though we could move components from one environment to another and check the resulting systems, we could not add new execution environments or even model all of the environments in systems that we built.

Many research systems in component-based software and explicit software architectures, such as Click [12], Koala [20], and Ptolemy [14] can also analyze properties of component compositions. TSL differs from the analyses implemented in these systems by providing explicit support for reasoning about concurrency in the presence of diverse execution environments.

8. Conclusion

This paper presents our Task/Scheduler Logic (TSL), a novel way to check safety properties of component-based systems software containing multiple execution environments. TSL breaks down the global problem of reasoning about a composition of execution environments into many small local problems dealing with individual schedulers and the environments they create. The TSL axioms model the hierarchical relationships between environments, facilitating modular, general-purpose descriptions of how systems code behaves. TSL can check for race conditions, improper use of synchronization primitives (e.g. blocking in an interrupt handler), and unnecessary synchronization operations. It can also infer the set of locks that satisfy the atomicity requirements of a particular critical section.

We have shown that TSL can be applied to real software by annotating over 300 systems software components, implementing a TSL inference engine, and using the results of the engine to check properties of composed systems.

References

- [1] J. Aldrich, C. Chambers, E. G. Sirer, and S. Eggers. Eliminating unnecessary synchronization from Java programs. In *Proc. of the Static Analysis Symp.*, Venezia, Italy, Sept. 1999.
- [2] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. of the 22nd Intl. Conf. on Software Engineering*, June 2000.
- [3] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, Palo Alto, CA, Dec. 1998.
- [4] C. Flanagan and M. Abadi. Types for safe locking. In S. Swierstra, editor, *ESOP'99 Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108. Springer-Verlag, Mar. 1999.
- [5] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for OS and language research. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, Oct. 1997.
- [6] B. Ford and S. Susarla. CPU inheritance scheduling. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation*, pages 91–105, Seattle, WA, Oct. 1996.
- [7] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation*, pages 107–121, Seattle, WA, Oct. 1996.
- [8] A. Greenhouse and W. L. Scherlis. Assuring and evolving concurrent programs: Annotations and policy. In *Proc. of the 24th Intl. Conf. on Software Engineering*, pages 453–463, Orlando, FL, May 2002.
- [9] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, Jan. 1991.
- [10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Cambridge, MA, Nov. 2000.
- [11] G. J. Holzmann. The Spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.
- [12] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.
- [13] L. Lamport. The ‘Hoare logic’ of concurrent programs. *Acta Informatica*, 14(1):32–37, 1980.
- [14] E. A. Lee. Overview of the Ptolemy project. Technical Memorandum UCB/ERL M01/11, University of California at Berkeley, Mar. 2001.
- [15] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, 1992.
- [16] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100:1–77, 1992.
- [17] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.
- [18] J. Regehr and J. A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proc. of the 22nd IEEE Real-Time Systems Symp. (RTSS 2001)*, Dec. 2001.
- [19] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, pages 347–360. USENIX, Oct. 2000.
- [20] R. van Ommering. Building product populations with software components. In *Proc. of the 24th Intl. Conf. on Software Engineering*, Orlando, FL, May 2002.
- [21] V. Yodaiken. The RTLinux manifesto. In *Proc. of The 5th Linux Expo*, Raleigh, NC, Mar. 1999.