1-1-2010

Extensible Scheduling in a Haskell-based Operating System

Kenneth William Graunke
Portland State University

Extensible Scheduling in a Haskell-based Operating System

by

Kenneth William Graunke

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

Thesis Committee:
Andrew Tolmach, Chair
Barton Massey
James Hook

Portland State University

Abstract

This thesis presents *Lighthouse*, an experimental branch of the Haskell-based House operating system which integrates Li et al.'s Lightweight Concurrency framework. First and foremost, it improves House's viability as a "real operating system" by providing a new extensible scheduler framework which makes it easy to experiment with different scheduling policies. In particular, Lighthouse extends Concurrent Haskell with thread priority and implements a priority-based scheduler which significantly improves system responsiveness when compared with GHC's normal round-robin scheduler. Even while doing this, it improves on House's claim of being "written in Haskell" by moving a whole subsystem out of the complex C-based runtime system and into Haskell itself.

In addition, Lighthouse also includes an alternate, simpler implementation of Lightweight Concurrency which takes advantage of House's unique setting (running directly on uniprocessor x86 hardware). This experience sheds light on areas that need further attention before the system can truly be viable—primarily interactions between blackholing and interrupt handling. In particular, this thesis uncovers a potential case of self-deadlock and suggests potential solutions.

Finally, this work offers further insight into the viability of using high-level languages such as Haskell for systems programming. Although laziness and blackholing present unique problems, many parts of the system are still much easier to express in Haskell than traditional languages such as C.

Dedication

This work is dedicated to Jesus Christ, in whom are hidden all the treasures of wisdom and knowledge (Colossians 2:3). He means everything to me.

Acknowledgments

This work could not have been accomplished without the help of others. Many thanks to Adam Wick for his great help with updating House to modern GHC so it would build on my machine. Thanks also to Peng Li for providing the foundation for such interesting research, as well as for providing his code. Both were instrumental in helping me get started. Thanks are also due to Simon Marlow, who helped nail some crashes that had me stumped in a matter of mere minutes—I am truly impressed—as well as for reviewing code, and offering advice on how to implement transactional memory. I am very grateful for my advisor, Andrew Tolmach, who offered solid direction and guidance, puzzled through difficult problems with me, and saw me through difficult times. John Chee, my friend, brother in Christ, and fellow student, kindly reviewed my thesis.

I owe much to my brother, Paul Graunke, who in many ways helped set me on my path in life. He introduced me to Linux while I was in grade school, helped me with C programming, introduced me to Scheme and functional programming, and much more. Like him, I chose to study computer science and programming languages. Paul graciously reviewed my thesis on many occasions, provided very insightful feedback, and helped me overcome many challenges. Without him, I suspect I may never have finished. Paul also has shown me much about God and life in the Spirit.

I am profoundly grateful for the Church, and my brothers and sisters who have stood by me through hard times. Chris Lyons has had a huge impact in my life, having helped

me come to know Christ in a time when I was deseperately searching for hope. More than a pastor, he is a great friend and a joy to share life with. Carol Wilkerson taught me much through her passion for prayer, and has been a wonderful mentor. Her husband Roger Wilkerson shared his passion for the scriptures, and helped me understand much. Robby Rittenhouse and Dylan Baker, great friends and dear brothers in Christ, have walked with me through much of life. Jean and Dave Nevills have been a constant encouragement to me through their passion and enthusiasm. My sister, Christy Jones, introduced me to the scriptures and been a living testament to me through her actions.

Many others also deserve my thanks—far more than I could possibly mention. I am particularly grateful to all those who have prayed steadfastly for me over the years, and to Jesus Christ, the one truly deserving of all honor, thanks, and praise.

Contents

List of Figures

Chapter 1

Introduction

House [1, 5, 29] is a Haskell-based operating system developed by the Programatica project [33], a joint effort of Portland State University and the Oregon Graduate Institute. While it is primarily a proof of concept, it does support several common OS features such as TCP/IP networking, userspace binaries, and a graphical interface. At the core, House is based on a "bare metal" port of the Glasgow Haskell Compiler (GHC) runtime system that runs directly on an uniprocessor x86-based machine instead of as a process on a host OS.

Although House's kernel and drivers are written in Haskell, the underlying GHC runtime system (RTS) is written in C and C-- [13], and represents a significant part of the operating system—about 50,000 lines of code.[1] Not only is it large—it is also highly complicated, containing many intricate details, tricks, and optimizations. From an assurance point of view, formally verifying such a magnitude of complex low-level code is daunting, if not downright impossible today. Verification aside, experience at Portland State has also shown that working on the RTS is no easy task, particularly for new students or developers.

One of the larger components of the RTS is the concurrency and scheduling support, weighing in at about 10,000 lines of code. Although GHC provides concurrent

[1] This is strictly a line count, including comments, whitespace, and so on.

threads, it offers no notion of thread priority and is hardcoded with a simplistic round-robin scheduling policy. For House, this means that running a few compute bound background tasks will seriously degrade system performance—so much so that it can be nearly impossible to use the graphical interface.

Shortly before this work began, Li et al. proposed a new concurrency interface for GHC called *Lightweight Concurrency* [16]. This new system proposed to implement only a small set of primitives (the *substrate*) in the RTS itself while moving most of the complexity—such as scheduling and concurrent data structures—into a concurrency library written in Haskell. Li et al. claimed this would simplify the RTS, separate mechanism from policy, and improve the safety and customizability of GHC's concurrency features. In particular, one of the claims was that Lightweight Concurrency could be used to implement an operating system scheduler. If true, all of these could greatly benefit House.

This thesis presents *Lighthouse*, an experimental variant of House based on the Lightweight Concurrency framework and GHC 6.8.2. First and foremost, it improves House's viability as a "real operating system" by providing a new *extensible scheduler* framework which allows for easy experimentation with a variety of scheduling policies. In particular, Lighthouse extends GHC's standard concurrency interface (Concurrent Haskell) [22] with thread priority and includes several simple priority-based schedulers that greatly improve system responsiveness. Even while doing this, it improves on House's claim of being "written in Haskell" by moving a great deal of code from the runtime system into Haskell itself.

Secondly, Lighthouse puts the Lightweight Concurrency framework into practice in a tangible way, demonstrating that it really can be used as the basis for a preemptive multitasking operating system. Although Peng Li was gracious enough to provide a

copy of his substrate implementation, Lighthouse reimplements almost all of it to take advantage of House's unique setting (running directly on uniprocessor x86 hardware), significantly simplifying certain parts.

While Lighthouse is able to corroborate many of Li et al.'s claims, it also sheds light on areas that need further attention before the system can truly be viable—first and foremost interactions between blackholing and interrupt handling, and to a much lesser degree, asynchronous exceptions.

Lighthouse does not attempt to address performance concerns regarding Lightweight Concurrency; it is more concerned with the expressivity and flexibility of the framework, as well as the simplicity of the resulting code. While many parts of the concurrency library are indeed straightforward to implement, other portions remain quite challenging. Even in a high-level language like Haskell, writing system-level code can still be far from simple.

Chapter 2 summarizes the Lightweight Concurrency framework and other relevant concepts. Chapter 3 covers Lighthouse's reimplementation of the substrate and the necessary RTS changes. Chapter 4 covers the concurrency library, including the new scheduling framework, a simple reimplementation of round-robin scheduling (now in Haskell), and much of GHC's `Control.Concurrent` interface. Chapter 5 then adds thread priority into the mix, presents a variety of schedulers built atop the new framework, and covers sources of priority inversion and various solutions. Finally, Chapter 6 discusses the benefits of Lighthouse compared to traditional House, comments on Haskell as a systems programming language, and discusses future and related work.

The Lighthouse source code is available at the author's website [30].

Chapter 2

The Lightweight Concurrency Framework

Li et al.'s *Lightweight Concurrency* framework [16] offers an alternative approach for supporting concurrency in GHC. Instead of having the language designers hardwire one particular concurrency system into the RTS, it offers a minimal *substrate* upon which others can build a variety of *concurrency libraries*. One such library could be an implementation of the traditional Concurrent Haskell interface [22] with conventional threads, MVars, QSems, Chans and the like—but others could implement something entirely different.

While the substrate is written in C and C-- [13] as part of the RTS, concurrency libraries are implemented in Haskell. As much as possible, the substrate attempts to implement mechanism, rather than policy—only the bare essentials needed to support concurrency. For example, the substrate implements a mechanism for switching threads, while a library may implement a particular scheduling algorithm.

This separation seems appealing—experimenting with new algorithms in Haskell is almost certainly easier than implementing them directly in the RTS, which is notoriously difficult to work on. Furthermore, although policy may change—or even vary depending on the application—the necessary mechanism likely will not. Thus, the remaining RTS code should not only be simpler, but less subject to change.

Figure 2.1 The Lightweight Concurrency substrate interface

Transactional Memory (PTM)

```
data PTM a
data PVar a
instance Monad PTM
newPVar    ::  a -> PTM (PVar a)
readPVar   ::  PVar a -> PTM a
writePVar  ::  PVar a -> a -> PTM ()
atomically ::  PTM a -> IO a
```

Thread Local State (TLS)

```
data TLSKey a
newTLSKey ::  a -> IO (TLSKey a)
getTLS    ::  TLSKey a -> PTM a
setTLS    ::  TLSKey a -> a -> IO ()
```

Continuations

```
data SCont
newSCont ::  IO () -> IO SCont
switch   ::  (SCont -> PTM SCont) -> IO ()
```

Upcall Interface

```
timerHandler     ::  IO ()
blackholeHandler ::  IO Bool -> IO ()
```

2.1 A look at the Substrate

The substrate is divided into three main components—a basic synchronization mechanism, continuations, and thread local state. Primarily, these take the form of RTS primitives; the substrate also specifies a few functions which a concurrency library must implement. These tie together the RTS and the concurrency library—the library can call the primitives, and the RTS may make upcalls to handle certain events, such as a timer. Parts of the published substrate [16] are unnecessary due to House's unique setting (running directly on uniprocessor x86 hardware); Figure 2.1 summarizes the subset used for this work.

2.2 Primitive Transactional Memory (PTM)

One of the key decisions in the design of a concurrency substrate is the choice of synchronization mechanism. Concurrent Haskell's MVars are not an option since they support blocking, which relies on scheduling—the business of the concurrency library. While MVars could be supported in a library, the substrate needs something more primitive.

Traditionally, locks would be the mechanism of choice, but their use can be particularly tricky in a lazy language like Haskell. As an illustration, the following code might be used to implement Concurrent Haskell's `threadDelay`, storing threads in a priority queue sorted by the amount of time they wish to wait:

```
do lock schedulerLock
   q <- readIORef sleepPQ
   writeIORef sleepPQ (sortedInsert (delay, thread) q)
   unlock schedulerLock
```

In order to perform the sorted insert, `delay` must be fully evaluated to a number—yet it could very well be a thunk representing a long, drawn-out computation. In other words, executing this seemingly small critical section could take arbitrarily and unpredictably long. Since the idea is to hold locks—especially spinlocks—for as little time as possible, this is very unappealing.

One could try to move unpredictably long evaluation outside of critical sections via `seq` or `$!` to force any potential thunks before taking the lock. While this is feasible for `delay`, the case where `q` itself is a thunk is clearly more challenging. In general, ensuring a piece of code does not evaluate any thunks is too heavy of a burden to place on Haskell programmers.

With this in mind, it seems necessary to look for an alternative, such as so-called

"optimistic" lock-free synchronization mechanisms. Optimistic systems assume that the common case is little to no contention and that the overhead of proactive mutual exclusion is *usually* a waste. In an optimistic system, threads proceed without locks, but must record assumptions and atomically check them as they attempt to write their results. A simple example is the atomic compare-and-swap (CAS) operation found on many systems—a code fragment might read a value, compute things based upon that, and use a CAS operation to simultaneously ensure the value it read is still valid while attempting to store the new value. If another thread changed the value, the operation will fail, and the computation will need to be started over. Although CAS is rather limited, research has shown that one can build operating systems based on variants such as double-compare-and-swap [20]—but it is not at all straightforward.

Another popular optimistic system is *transactional memory* [8], which borrows its main idea from database systems. Database transactions allow one to group several queries into a logical group—or transaction—whose effects are all-or-nothing. Within a transaction, one can modify various tables and rows, but these changes are not visible until the whole transaction finishes and atomically *commits* them. Should a conflict arise between two simultaneous transactions, one of them will be *rolled back* such that any changes it made are undone. Once rolled back, the transaction will automatically retry until it succeeds.

Similarly, transactional memory allows one to group reads and writes to various memory locations into a single unit whose effects are all-or-nothing, with automatic rollback and retry. One of the well known advantages of transactional memory is its compositionality—it is much easier to combine transactional programs than lock-based ones. This is particularly useful in higher-order functional languages which make extensive use of compositionality.

GHC offers an implementation of Software Transactional Memory (STM) [7]. However, the substrate cannot use the full-fledged STM directly since STM includes blocking capabilities which are the business of the concurrency library. Instead, Lightweight Concurrency offers a stripped down or *Primitive* Transactional Memory (PTM). For those familiar with STM, PTM is essentially identical but excludes `retry` and `orElse`, the two functions which can introduce blocking.

Figure 2.1 shows the PTM interface. It defines a new type of mutable variable—the PVar—with read and write operations. Both operations are in a new PTM monad, which represents transactional computations. Importantly, neither actually alters the variables directly, but both record assumptions and stage changes to be verified and done atomically during the final commit. A final operation, `atomically`, evaluates a transaction and atomically commits its changes.

PTM greatly benefits from Haskell's strong, polymorphic typing—since `readPVar` and `writePVar` operate in the PTM monad, PVars cannot be accessed outside of a transaction. Also, side-effecting computation—such as writing to an IORef—cannot be done inside a transaction. While this might seem crippling in many languages, it is not a problem in Haskell, where most computation is pure. Such typing may seem natural to Haskell programmers, who are accustomed to special kinds of mutable variables. However, many transactional memory systems allow mixed transactional and non-transactional operations on the same memory locations. PTM has simpler and clearer semantics.

Notably, PTM does not attempt to prevent starvation—longer running transactions may have trouble committing if they repeatedly conflict with quick ones. Thus, it is best to keep transactions short. Should a transaction hit an expensive thunk, it is at risk of becoming invalid—however, the work done to evaluate the thunk will not be lost,

hopefully allowing it to run more quickly in subsequent tries [16].

One might think that, being lock-free, other threads can proceed while one evaluates an expensive thunk—which is true with one exception. If two threads both try to force the same thunk, the second thread will hit a *blackhole* (defined in section 2.7) and be forced to wait until the first one finishes. While this may diminish some of the benefit of PTM, it is still a solid improvement—problems only arise if two threads evaluate exactly the same thunk, rather than simply run two sections of code protected by the same lock. This effectively provides a fine-grained locking scheme without any of the trouble of designing or managing one.

2.3 Thread Local State (TLS)

Lightweight Concurrency also includes implicitly passed *thread local state* (TLS). Local state can be useful for storing things that every thread has, but has a unique version of—for example, thread IDs, standard input/output handles, or random number seeds. While it is theoretically possible to pass this state explicitly to every function that needs it, in practice it is often awkward and difficult to do so.

TLS is dictionary based—each thread has its own dictionary, and there is also a global default dictionary. To store a new item, one creates a new (typically global) TLS key:

```
signalsBlocked :: TLSKey Bool
signalsBlocked = unsafePerformIO[1]$ newTLSKey False
```

This stores the default value, `False`, in the global dictionary. A thread could then override this value by calling:

[1] `unsafePerformIO` is currently the best-practice method for creating global variables in Haskell and in this case is safe.

```
setTLS signalsBlocked True
```

This stores the new value, `True`, in that thread's local dictionary. When looking up a key, `getTLS` first checks the thread's local dictionary, then falls back to the global one. Notably, this can never fail, since `newTLSKey` stored the default value before returning the opaque key—so `getTLS` does not need to use a Maybe type.

The intent is for TLS to be integrated in the RTS itself. One might think to use a global data structure in Haskell—such as a hash table indexed by thread ID—but this has two disadvantages. For one, it would probably be much slower than a normal memory reference, which could be bad if it is accessed frequently. More importantly, however, automated garbage collection would not work—the programmer would have to somehow deallocate a thread's local state when the thread died.[2] TLS is also simple enough that adding it to the RTS seems reasonable.

2.4 Haskell Execution Contexts (HECs)

Li et al. describe Haskell Execution Contexts (HECs), which can be thought of as virtual CPUs that run Haskell code. Normally, the RTS maps HECs to OS threads or physical CPUs (if running on bare metal), bringing the host system's true concurrency into the picture. Being uniprocessor, Lighthouse contains exactly one HEC.

2.5 Continuations

Following a long standing tradition [25], Lightweight Concurrency offers continuations as the basis of threading. In a stack based implementation like GHC, it is much more

[2] A partial solution would be to have the structure store weak pointers, using the thread ID as key and dictionary as value. This would automatically free the dictionary, but something would still have to clean out the dead weak pointers from the table. Many systems provide a weak hash table for this very purpose, but GHC does not.

efficient to use non-first class *stack continuations* (SConts) which can only be used once. The following data type and primitives expose continuations to Haskell:

```
data SCont
newSCont :: IO () -> IO SCont
switch   :: (SCont -> PTM SCont) -> IO ()
```

One can think of an SCont as an object representing a suspended IO computation that, when switched to, continues where it left off. Of course, such concurrent computations need to get started somehow, and the first primitive does just that: `newSCont io` creates a continuation which, when run, performs the monadic action `io` on an empty stack.

`switch` is really the heart of the system. First, it captures the currently running computation as an SCont so it can be resumed later. Then it atomically gives this to the supplied function, and transfers control to the SCont it receives in return. One can think of the supplied function as a scheduler of sorts—it might store the current computation's SCont on a ready queue, and return the next SCont on that queue. Alternatively, it might put the current SCont on a blocked queue, and return something else—many things are possible.

When transferring control, `switch` also marks the target SCont as "used;" a correct implementation must issue a run-time error if the target SCont has already been used. This guarantees that stack continuations are only used once.

From the point of view of a single computation, nothing extraordinary happens when switching—it is running, then it calls `switch`, and the next thing it notices (should it ever be switched back to) is its return from `switch`. It has no idea that it was ever suspended, shuffled around in the system, and resumed. Certainly, some side effects/changes in the world may have occurred, but that is natural for an IO computation.

2.6 Preemption

While `switch` provides all that is needed for cooperative threading, more is needed to support preemption in the new world. Previously, the RTS internal timer was used directly by the scheduler, but now timer events must somehow be exposed to the Haskell level concurrency library. This is done via an upcall from the substrate to the concurrency library's required `timerHandler` function. Implementing such upcalls is tricky—one must alter the current stack so it looks like the Haskell code being executed had explicitly called `timerHandler`. The nice thing about doing so is that when `timerHandler` returns, everything continues running as usual.

A simple implementation of `timerHandler` might simply be to call `yield`, which in turn would rely on `switch`.

2.6.1 Interrupt Handling

While the Lightweight Concurrency substrate does not deal with interrupt handling *per se*, one can see it as a logical extension of `timerHandler`—handling not only timer interrupts, but all kinds. Lighthouse alters the substrate interface to provide a more general upcall:

```
interruptHandler :: IO ()
```

In fact, in Lighthouse, `timerHandler` is part of the concurrency library, not the substrate—it is simply called by `interruptHandler`. The new call uses the same stack arrangement as `timerHandler`. Section 3.2 discusses interrupt handling in more detail.

2.7 Laziness and Blackholes

Blackholing [6, 11, 18, 23] is a technique for preventing space leaks in lazy languages such as Haskell. While it is mostly transparent in everyday programming, it has a significant impact when implementing a concurrency system.

To understand blackholing, one must delve into how GHC evaluates Haskell. As an example, consider finding the last element in a list:

```
last []     = error "no last element"
last [x]    = x
last (x:xs) = last xs

let z = last [1..5000]
```

Recall that Haskell is *lazy*, meaning that expressions are not evaluated until their value is demanded. Instead, delayed computations are stored as *thunks*, which can later be "entered"—or evaluated—to produce a value. In the example, `z` starts as a thunk to compute `last [1..5000]`. When demanded, the thunk will be evaluated as follows (the notation T_{foo} signifies a thunk which, when entered, would compute `foo`):

$T_{\text{last [1..5000]}}$	(`z` starts as a thunk)
`last $` $T_{[1..5000]}$	(expand application node)
`last $ 1:`$T_{[2..5000]}$	(pattern matching in `last` demands thunk evaluation)
`last $` $T_{[2..5000]}$	(substitution: `last (x:xs) = last xs`)
`last $ 2:`$T_{[3..5000]}$	(pattern matching in `last` demands thunk evaluation)
⋮	⋮
`last $ 5000:`$T_{[]}$	(pattern matching in `last` demands thunk evaluation)
`5000`	(substitution: `last [x] = x`)

Not only is Haskell lazy, it memoizes computed values by overwriting thunks with their value. The most obvious approach would be to update thunks at each step of

evaluation—however, this is very inefficient. Not only would each step of evaluation require a write to the shared heap, but many intermediate values would need to be generated [23]. In the above example, each step would require a heap write.

The STG machine [12], which is the basis for GHC, avoids this inefficiency by performing thunk evaluation on the stack and only updating the thunk once the result is fully evaluated.

Since `last` only needs to operate on one cons cell at a time and the list `[1..5000]` is created lazily, it should be able to run in constant space—and a very small amount at that. However, if thunk `z` is left in the heap until it is fully evaluated (to `5000`), then the heap still contains a live reference to the start of the list. Not a single cons cell can be freed by the garbage collector; just before the final update, the entire list will be in memory. In general, preserving the original thunk causes many common operations run in $O(n)$ space instead of $O(1)$ space. This "space leak" is particularly important since it can lead to abrupt termination due to a heap overflow.

To stop the leak, GHC first copies a thunk's contents onto the stack, then overwrites the heap cell with a special object called a *blackhole*.[3] Unlike thunks, the garbage collector does not examine the contents of blackholes, so the heap no longer holds a live reference to the computation. When GHC finishes evaluating the thunk, it overwrites the blackhole with the final value (or an indirection to it). This system allows programs to once again run in $O(1)$ space, but without the inefficiency of the naïve approach.

Blackholing can also detect certain ill-founded recursive definitions. For example, consider the definition $\mathtt{x} = 1 + \mathtt{x}$. Normally, evaluating the thunk `x` would eventually result in a stack overflow—each time the evaluator encountered `x`, it would find it to

[3] GHC actually implements *lazy blackholing*, which defers this overwriting of thunks with blackholes until garbage collection time. For small thunks that can be evaluated in a small amount of space, this avoids the overhead of blackholing.

be a thunk and evaluate it again, pushing a new activation record on the stack, and recursing indefinitely. In contrast, if the thunk is overwritten by a blackhole when it is first evaluated, the second reference to `x` would not be a thunk at all—instead, the evaluator would find a blackhole. In a sequential system, such a self-referential loop is the only explanation for entering a blackhole, so the RTS can almost immediately raise an error (GHC's `BlockedOnBlackHole` exception).

2.7.1 Concurrency

In a concurrent system, however, encountering a blackhole is not necessarily an error. If thread *A* tries to evaluate a thunk, then thread *B* also tries to evaluate it, thread *B* will see a blackhole. In current GHC, the RTS would block thread *B*, making it wait until thread *A* finishes evaluating the thunk.

To facilitate this blocking, the RTS maintains a global queue of all threads blocked on blackholes (regardless of which); each thread's status field points to the specific blackhole the thread is waiting for. The RTS's scheduler periodically traverses this "blackhole queue," checking if the object pointed to by the thread's status is still a blackhole. If not, it wakes the thread, appending it to the run queue [6].

Lightweight Concurrency must take a different approach, however, as both blocking and scheduling are concepts defined by the Haskell-level concurrency library, not the RTS substrate; it is impossible for the RTS to block a thread. Instead, the RTS makes an upcall to the concurrency library (as it does for `timerHandler`) whenever a thread attempts to evaluate a blackhole:

```
blackholeHandler :: IO Bool -> IO ()
```

This function can then call `switch`. It could also store the current thread's SCont on

a Haskell-level queue which the Haskell-level scheduler could check, effectively implementing the same approach as the traditional RTS. To facilitate this, the RTS also supplies an `IO Bool` function which allows the concurrency library to check whether the blackhole has been resolved yet. Alternatively, `blackholeHandler` could simply try to switch to the thread; if the thread still cannot proceed, it will simply call `blackholeHandler` again. The result is a valid (if inefficient) implementation of `blackholeHandler` [15, 16].

Unfortunately, Lightweight Concurrency's `blackholeHandler` approach cannot be implemented correctly; section 3.5 discusses why.

Chapter 3

Implementing the Substrate

Although Peng Li was gracious enough to provide a copy of his substrate implementation, Lighthouse reimplements almost all of it to take advantage of House's unique setting, significantly simplifying certain parts. In particular, it includes an implementation of PTM written in Haskell, removing one of the most complex parts of the substrate from the RTS. Being uniprocessor, Lighthouse also has no need of functions to manage multiple HECs, foreign call tracking, and support for "bound" threads (which always run on a particular CPU).

In other words, while using the original version of Lightweight Concurrency [16] would strengthen House's claim to be "written in Haskell," Lighthouse's custom implementation does even better.

3.1 Stack Continuations and Switching

3.1.1 Representation

Internally, GHC's runtime system represents threads as thread state objects (TSOs). Primarily, a TSO contains a thread's stack and space to save the state of the STG machine, such as the current register values and stack pointer, when a thread is switched out. Lighthouse eliminates most of the other fields [26], often by moving the correspond-

ing functionality into Haskell code, but sometimes simply because they are no longer necessary. TSOs live in the Haskell heap, and can be garbage collected like any other object.

TSOs already contain all the information necessary to resume a thread, and thus form the basis for Lighthouse's stack continuations. To expose them to Haskell, Lighthouse includes a new TSO# primitive type (representing pointers to TSO objects) and two primitive operations: `getTSO#` returns the currently running thread's TSO, and `newTSO#` is used in creating new continuations.

With this in place, an SCont is simply a TSO pointer and a mutable "used up" flag[1]:

```
data SContStatus = Used | Usable
data SCont = SCont TSO# (IORef SContStatus)
```

3.1.2 Creating Threads (newSCont)

GHC's RTS has a `createIOThread` function which does most of the work to create new threads. Given a closure representing the IO () computation the thread should run, it allocates and initializes a new TSO, and returns a pointer to it. The `newTSO#` primitive operation (written in C--) simply wraps this, returning a TSO# to Haskell. In turn, `newSCont` (written in Haskell) wraps that, packaging the TSO# together with `newIORef Usable` to create an actual SCont.

One thing to keep in mind is that, being in the substrate, `newSCont` only does the bare minimum and creates a shell of a thread. For example, it does not set up with a proper thread ID or prepare the thread to handle asynchronous exceptions. The library function `forkIO` (section 4.3.3) uses `newSCont`, but fully sets up a thread.

[1] One might think to define SConts in terms of a Maybe type—`SCont (Maybe TSO#)`—but unfortunately this fails with a Kind error since TSO# is an unboxed type.

3.1.3 Switching

At the RTS level, a new `switch#` primitive offers a way to switch to a given thread. It takes a single argument—a pointer to the target thread's TSO.

Since GHC's runtime is stack based, HECs constantly examine their current thread's stack, perform the indicated operation, and adjust the stack accordingly. Each HEC contains a register set for the GHC virtual machine, including a pointer to the currently running TSO (`CurrentTSO`), and the current stack pointer (`Sp`). Each TSO also contains a saved copy of `Sp` which is updated when the thread is switched out, and used to restore the HEC's `Sp` when switching back in.

The procedure is essentially[2]:

1. Save the HEC's `Sp` register to `CurrentTSO->Sp`
2. Change `CurrentTSO` to point to the target thread's TSO
3. Restore the HEC's `Sp` from `CurrentTSO->Sp`

Oblivious to this change, the HEC merrily continues executing as usual—but now processes the new stack, running the target thread.

The interface on the Haskell side is somewhat fancier than that of the RTS primitive. Recall the type signature for `switch`:

```
switch :: (SCont -> PTM SCont) -> IO ()
```

The given function takes a stack continuation representing the current thread, and returns (in PTM context) the continuation for a thread to switch to.

At the Haskell level, the procedure for `switch` is:

[2]Simon Marlow kindly reviewed Lighthouse's implementation of the `switch#` primitive; at his suggestion, the actual implementation uses load/save thread state functions provided by GHC to perform this process in a cleaner, more readable fashion.

Figure 3.1 Haskell code for `switch`

```
switch :: (SCont -> PTM SCont) -> IO ()
switch scheduler = do s1 <- getSCont
                      s2 <- atomically (scheduler s1)
                      switchTo s2
  where getSCont :: IO SCont
        getSCont = do ref <- newIORef Usable
                      ...call getTSO# and return SCont tso ref...
        switchTo :: SCont -> IO ()
        switchTo (SCont tso ref) =
          do status <- readIORef ref
             case status of
               Used   -> error "Attempted to switch to used SCont!"
               Usable -> do writeIORef ref Used
                            ...call switch# tso...
```

1. Create an SCont for the current thread.

2. Call the supplied "scheduler" to get the continuation for the target thread. (It may also stash the current thread's SCont on a queue of some sort.)

3. Assert that the target SCont is usable, then mark it as used up.

4. Call switch# on the target's TSO#.

Figure 3.1 shows the code for `switch`, aside from the more esoteric detail of how one calls primitive IO operations.

3.2 Interrupt Handling

While the Lightweight Concurrency substrate does not deal with interrupt handling *per se*, one can see it as a logical extension of `timerHandler`—handling not only timer interrupts, but all kinds. Interrupt handling can be tricky in any operating system, but offers a few extra challenges in a garbage-collected language such as Haskell.

Recall the conventional approach to interrupt handling: When a hardware interrupt occurs, the system immediately runs a C based handler, stealing time from whatever process happened to be running. When done, the interrupt handler returns, the trap/fault is successfully handled, and whatever was running continues. Handlers are expected to be fairly quick, often only fetching data off the hardware, storing it in some data structure, and scheduling the actual processing for later. Spinlocks and IRQ disabling safeguard these structures against concurrent access between interrupt handlers and the rest of the system.

Unfortunately, this "run now" approach does not work in GHC—nor in many other garbage-collected systems. In general, it is not safe to interrupt Haskell execution at any arbitrary point.

3.2.1 Safe Points

As Haskell code runs, it constantly modifies the heap, always storing new things. This is usually a two-step process:

1. Allocating space for a new object by increasing the heap frontier pointer.

2. Actually writing the object to the newly allocated heap space (potentially multiple words).

Between these two steps, there is a moment when the heap is an inconsistent state—the new space supposedly contains an object, but does not yet. In fact, the memory probably still contains remnants of whatever was previously at that address—so reading the heap and trying to interpret it could be very dangerous.

Haskell execution is full of these tiny critical sections. Since they occur so often, and are so quick, the overhead of conventional locking would dominate the time required to

do the actual work, killing performance.

As such, GHC and several other run time systems take a different approach. Rather than having to constantly check if the heap is locked, code can simply assume that it is, except at special *safe points* [24] where the heap is *not* locked. In effect, this melds together all the small critical sections into one big one, reducing overhead considerably.

3.2.2 Haskell Interrupts (HsIRQs)

Since Haskell cannot be interrupted immediately, interrupt handling must take a delayed approach. When a hardware interrupt occurs, the handler must acknowledge it and arrange for it to be handled later. When the RTS reaches a safe point, it can check for pending interrupts and finally handle them by scheduling or invoking Haskell code. Since it can quickly become confusing what one means by "interrupt"—either the actual hardware interrupt, or the point when Haskell finds out about it—this paper refers to the latter as "Haskell interrupts," or HsIRQs for short.

Clearly, this delayed approach will suffer increased interrupt latency, but if safe points are frequent enough, it should be acceptable. Exactly when safe points occur depends on the details of the RTS implementation, but they usually occur at garbage collection time since the collector must access the heap as a whole. Since virtually all Haskell code allocates memory—and thus will trigger garbage collection—safe points are usually frequent.

Unfortunately, it is possible to write loops that do not allocate, and hence have no safe points; such loops may prevent HsIRQs indefinitely, effectively locking the system. The GHC implementers have opted to overlook this issue, however, since most examples where it arises are contrived. One such place where this can become an issue is House's system idle thread; special care must be taken to ensure it allocates or otherwise includes

a safe point. Notably, a foreign function call also triggers a safe point.

The next sections describe two approaches for implementing HsIRQs—first, House's existing implementation and why it would not work with Lightweight Concurrency, followed by Lighthouse's new approach.

3.2.3 House's Traditional Method

Interrupts are quite similar to the UNIX notion of signals—both are asynchronous non-data-carrying events which a process (or OS) may handle. This is the key to House's traditional method—reusing GHC's existing signal handling mechanism to handle interrupts.

House's traditional interrupt handling centers on three key data structures:

1. A Haskell-level IRQ handler table, mapping IRQ numbers to IO () computations. (Handlers are registered by device drivers, also written in Haskell.)

2. A parallel table in C mapping IRQ numbers to StgClosure (Haskell object) pointers for each of the handlers.

3. An array of handlers for pending interrupts, limited to 16 (the number of IRQs available on a basic x86 machine).

When a hardware interrupt occurs, a small C routine looks up the pointer for the appropriate Haskell handler in the parallel table and pushes it to the array of pending handlers. Periodically, the RTS checks for any signals awaiting delivery; House's signal handler processes the array and forks a new Haskell thread for each of the pending handlers, placing it at the end of the RTS's single run queue. Importantly, this is all done at the RTS level, in C code.

While this mapping is conceptually simple, it is hardly ideal. Forking a new thread for each interrupt introduces a lot of overhead—not only the cost of creating the new thread, but also the latency of having to wait for the scheduler to get around to running it. (The fact that House's places it at the end of the run queue rather than the beginning is likely an oversight, but even with this fixed, House's approach still performs poorly.)

3.2.3.1 The RTS Cannot Schedule Threads

House's approach is also impossible with the Lightweight Concurrency framework because *the RTS cannot schedule threads*—all thread scheduling is now done in Haskell code. The RTS needed this ability for three things—interrupt handling, blackhole handling, and supporting finalizers for the weak pointers provided by the System.Mem.Weak module. Finalizers are supposed to run (or be scheduled) when a weak pointer's key becomes unreachable and are intricately tied to the RTS's garbage collection.

Lighthouse simply removes finalizer support from the RTS, as House does not need it. This is likely not a great loss—GHC's finalizers are already unreliable and come with no guarantees of eventually running [28].

Still, this does cause one to question whether it would be useful to retain the ability for the RTS to create and schedule threads. It would not be hard for the substrate to include a RTS-side queue of threads and offer a primitive such as:

```
checkForRTSWork :: IO SCont
```

The Haskell-based scheduler could then poll for incoming work from the RTS.

3.2.4 A New Method: Thread Hijacking

A final blow to House's old approach is that it cannot handle timer interrupts—running those in their own thread would be entirely useless! Rather than mixing two approaches, Andrew Tolmach, Simon Marlow, and I generalized Lightweight Concurrency's approach for timers (section 2.6) to handle all interrupts. In lieu of `timerHandler` (now part of the base scheduler—section 4.2.1) we defined a new function in the House kernel:

```
interruptHandler :: IO ()
```

When an actual hardware interrupt occurs, the C level handler simply sets the corresponding bit in a global `pending_irqs` bit field, marking it for later handling. When the RTS reaches a safe point, it checks `pending_irqs`. If any interrupts are waiting, it pushes an `interruptHandler` closure onto the current Haskell thread's stack before continuing, effectively "hijacking" the current thread for its own purposes.

When run, `interruptHandler` performs a FFI call to a small C routine which clears and returns *one* IRQ from `pending_irqs`. Then it looks up the appropriate handler in House's Haskell-side IRQ handler table, and calls it. Finally, it loops to see if any other interrupts were pending.

When no more interrupts are pending, `interruptHandler` returns and is popped from the stack—leaving the thread in the same state as before we hijacked it. Computation continues where it left off, without any special RTS intervention.

Notably, `interruptHandler` must be added as one of the garbage collector's roots so anything accessed by HsIRQ handlers (like scheduler queues) stays alive.

This approach has many advantages over House's old method. First, it has very low overhead—there is no need to create and schedule a whole new thread for small, quick

tasks. (For larger tasks, the HsIRQ handler is of course free to fork a new thread via `forkIO`.) It also has lower latency—the HsIRQ handler runs immediately (as soon as GHC reaches a safe point). Secondly, it is simpler—no IRQ handler lookup table is needed in C, and timer interrupts can be handled via the same mechanism as everything else. Finally, it is very similar to conventional interrupt handling—handlers steal time from whatever thread happened to be running at the time.

3.2.4.1 Issues

Despite these advantages, the approach does have problems. First, `interruptHandler` may reenter. While running one handler, the RTS may reach another safe point and hijack the thread again, pushing a new instance of `interruptHandler` to handle a second interrupt. While this is a bit odd, it is safe—PTM ensures that if both handlers try to write to the same structure, the earlier transaction will retry. Also, since `interruptHandler` atomically clears a pending IRQ's bit as it acquires it, there is no danger of this second instance trying to handle the same IRQ.

Secondly, HsIRQ handlers may switch threads. Consider reentrance, with the second interrupt being a timer. The scheduler is oblivious to the fact that it is in interrupt context and may decide to yield, switching to a new thread. The first interrupt is not lost, just delayed—it will still be handled when its host thread runs again. Of course, this could be a while, especially if the current thread happened to be low priority according to whatever scheduling policy is in place.

Also, a HsIRQ handler could simply block—say, on an MVar—which would result in the host thread getting blocked, as well. Unfortunately, there is no longer anything to prevent a poorly written device driver from hanging random threads. That said, blocking while in interrupt context is well known to be a terrible error in most systems.

A final concern is that is that HsIRQ handlers have full access to their host thread's local state. While this at first may seem to violate the notion of "thread local," there are some legitimate uses for this. For example, a timer handler may need to read a thread's priority from TLS in order to schedule it correctly. The scheduler may also wish to update certain per-thread accounting data. In both of these cases, the HsIRQ handler is really acting on behalf of the host thread. Naturally, HsIRQ handlers can only access TLS entries for which they have the key; keeping TLS keys local to a module is a good way to restrict access.

3.3 PTM

Lighthouse includes a unique implementation of PTM which is written in Haskell rather than being built into the RTS. Simon Marlow suggested the approach and Andrew Tolmach implemented it—I simply fixed a minor bug and integrated it with the rest of the system. One of the key reasons for this approach is to support *nested transactions*, which can arise as follows:

1. A thread is in the non-atomic "log building" phase of transaction *A*—running a computation of type PTM a.

2. It gets hijacked by an interrupt handler—of type IO ()—which starts a new transaction *B*.

3. When done, the handler returns and the thread is back in transaction *A*.

Supporting nested transactions is quite desirable—it means both normal code and interrupt handlers can use PTM and will not affect each other unless their transactions actually share variables. This is one of the key reasons we chose not to use GHC's ex-

isting STM code as the basis for PTM—STM does not support nested transactions, and adding such support would have been a significant amount of work.

With PTM in Haskell, the substrate must provide *some* mechanism to ensure the atomicity of the "verify and commit" stage. To this end, I implemented two primitives to temporarily disable Haskell Interrupts (3.2.2):

```
disableHsIRQs :: IO Bool
restoreHsIRQs :: Bool -> IO ()
```

These eliminate pseudo-concurrency—if no interrupts can be delivered, there is no chance of switching to another thread. This guarantees the current thread has exclusive access to all the PVars. Unfortunately, this is only viable in single processor settings.

One final concern is whether blackholing (section 2.7) jeopardizes the atomicity of the commit stage. In Li et al.'s formulation of PTM, "no Haskell operations are involved" [16] in the commit stage, which is a clear guarantee that blackholes cannot arise. Although Lighthouse's PTM does not have this property, it is also safe from blackholing. When verifying the log, it uses GHC's `reallyUnsafePtrEq#` to perform pointer comparisons, and thus never has to evaluate or force the the contents of PVars. Thus, the only possible place for thunks is in the log structure itself, which is local to the current thread and transaction.

This approach to PTM has worked well, offering a simple implementation and moving a significant amount of functionality into Haskell—directly supporting Lighthouse's goals. However, it is not without it is drawbacks—for one, there is no explicit support for saving computations across retries. Also, relying on `reallyUnsafePtrEq#` is non-portable and depends on the inner workings of the RTS.

3.4 TLS

Unlike the other components, Lighthouse uses Peng Li's TLS implementation almost verbatim. The only real difference is that his supports primitives to alter other threads' local state—ostensibly for initialization purposes—while Lighthouse does not. While it may be excusable for a parent to use such primitives when creating a child thread, sacrificing purity is dangerous. Nothing prevents such primitives from being used elsewhere, violating the whole notion of "thread local" state.

Both implementations provide a single global dictionary containing the default values for all keys, as well as partial copies in each TSO which contain locally modified values. The dictionaries are simply represented as arrays of `StgClosure` (Haskell object) pointers; TLSKeys are simply indexes into those arrays. Both use the same indexing—the TSO copies are simply prefixes of the global array. Notably, this works because once TLS keys are created, they live forever.

`setTLS` checks if the given key/index is within the TSO's prefix; if so, it simply updates it. If not, it expands the prefix by copying default values from the global array. `getTLS` does not expand the prefix—if the requested key is not in the local copy, it looks it up in the global copy. `newTLS` simply appends to the array of global defaults and returns the index—it does not have to alter TSO copies at all.

Finally, the global TLS dictionary must be added as one of the garbage collector's roots; the garbage collection code for TSOs must also traverse the local TLS. Both of these are straightforward extensions.

3.4.1 getTLS and PTM

A final point of interest is that `getTLS` is in the PTM monad rather than the IO monad. Yet, Lighthouse's implementation of PTM does not log/verify TLS reads. Since the state is thread local, and `setTLS` is in the IO monad, it should be impossible for such state to change during a transaction.

However, as noted in section 3.2.4.1, this is not entirely true—HsIRQ handling can hijack a thread in the middle of a transaction, and runs in IO context. Thus, it is fully capable of calling `setTLS` and altering thread local state in the midst of a transaction.

Since interrupt handlers run in arbitrary threads, it is not likely to be useful or even sensible for them to tamper with thread local state. As this is the only way for TLS to change within a transaction (`unsafePerformIO` aside), it did not seem worthwhile to add complexity and overhead to PTM logging.

3.5 Blackholes

Lighthouse does not yet implement blackhole handling; if the concurrency system must handle a blackhole, it simply kills the operating system. Two unresolved problems stand in the way of a correct implementation. First, the blackhole handler can reenter. Secondly, interactions between interrupt handling and blackholing can lead to self-deadlock. The root cause is that the RTS's blackholing affects thread scheduling—a concept implemented by the concurrency library.

A potential solution for the first problem is discussed below, and should be feasible to implement. However, the second problem remains unresolved; a solution may require substantial changes to GHC or even eliminate the need for `blackholeHandler` altogether. With this in mind, implementing a partially working `blackholeHandler` did

not seem worthwhile—especially considering that doing so would require a substantial amount of work on the RTS.

3.5.1 Reentrance

The first issue is that `blackholeHandler` can reenter—that is, `blackholeHandler` itself can encounter a blackhole. Prior work [15, 16] suggests using TLS to detect such reentrance and to busy wait when it occurs. Unfortunately, this does not work in a uniprocessor setting.

Busy waiting presumes that another thread is running simultaneously on a different processor—one which, hopefully, will resolve the blackhole. However, in a uniprocessor setting, busy waiting would at best stall the system until a timer interrupt occurred. The only sensible approach would be to switch to another thread, but finding another thread to run involves Haskell operations which may well blackhole.

For Lighthouse, likely the most sensible solution is to prevent reentrance altogether. This could be accomplished by disabling HsIRQs during `blackholeHandler`, and using Haskell's module system to ensure that any data structures touched by `blackholeHandler` cannot be accessed in any other context. With these restrictions, blackhole handling would run to completion, and no other threads could be in the process of evaluating necessary thunks. Thus, reentrance would simply be a bug. In fact, the RTS could assert that HsIRQs are on before calling `blackholeHandler`, detecting both reentrance and blackholes during the PTM commit stage—a bug either way.

3.5.2 Self-Deadlock

Consider a scenario involving nested PTM transactions (section 3.3) which both need to evaluate some of the same thunks. First, the outer transaction begins evaluating thunks

and generating blackholes. Before it finishes, the RTS hijacks the thread and runs a HsIRQ handler which starts a transaction. Then, this inner transaction hits one of the blackholes caused by the outer transaction. The inner transaction cannot proceed until the outer transaction finishes its thunk evaluation. But the outer transaction cannot proceed until the inner transaction finishes, by virtue of it being closer to the top of the stack. Neither can proceed, and the thread is deadlocked with itself!

Furthermore, this deadlock will almost certainly cause the whole system to fail. The deadlocked thread cannot call `switch`, so the only way anything else could run is if the RTS detects the situation and steps in. Since the scheduler is in Haskell, it is hard to imagine how this could work; finally, it may be that the shared thunk causing the issue was part of the scheduler's run queue. If so, whole system failure is inevitable.

Technically, this situation can arise without nested transactions: if a thread begins evaluating thunk x and is hijacked by a HsIRQ handler that also tries to evaluate thunk x, it will hit its own blackhole. However, sharing data between interrupt and non-interrupt contexts without synchronization will fail spectacularly on most systems—the real surprise is that the synchronization mechanism, PTM, offers no protection against this danger.

With this in mind, one might argue that Lighthouse's thread hijacking is at fault. Although Lighthouse exacerbates the problem by using thread hijacking for all kinds of interrupts, Li's formulation of Lightweight Concurrency uses the same approach to handle timers [15]. It is easy to see how this could lead to trouble: a thread might be in the middle of forking—and thus adding a thread to the scheduler's run queue—when a timer occurs and `timerHandler` needs to consult that very queue.

If thread hijacking is only used for timers, it may be possible to rewind the stack to just before the inserted `timerHandler` call, effectively discarding the timer, since

another will come anyway. Of course, this is rather dubious—`timerHandler` is in the IO monad and may have already performed other side effects that cannot be undone. Another question worth asking: Just how many timers are going to be discarded before the thread is finally switched out? While one time slice may be tolerable, a rogue thread may be able to trigger this behavior more frequently, and degrade system performance.

3.5.3 Conflicting Synchronization Mechanisms

Stepping back for a moment, perhaps the real problem is with blackholing itself. What started out as a transparent RTS detail designed to improve space behavior has become a synchronization mechanism—and a rather unfriendly one, at that! Creating a blackhole is analogous to successfully taking a lock, and encountering a blackhole is like failing to take a lock held by another thread. The unfortunate part is that programs are forced to unwittingly take such locks all the time. There is no forewarning and no recourse—a thread has no way of knowing it is about to fail to take a lock, and once it attempts, there is no alternative but to block.

Here is where the locking analogy breaks down: in a locking system, there are a set number of locks—usually a small number govern a particular piece of data—and a set number of lock acquisition sites. Though challenging, it is possible to reason about program behavior. In contrast, it is widely accepted that it is infeasible to reason about whether Haskell code will encounter a blackhole. The only safe assumption is that a thread may block at any time, when evaluating anything, just because some other thread (or possibly the same, in the presence of thread hijacking) started evaluating a thunk first.

This is terrible from a concurrent programming perspective—Concurrent Haskell affords programmers no control over the situation. Both locking and transactional memory

systems allow the programmer to decide when to block, when to busy wait or retry, and when to abort or take some other course of action. This is especially important when writing system level code.

In Lightweight Concurrency, the synchronization mechanism of choice is PTM, so programmers expect PTM to govern any shared access—but blackholing takes over. In a sense, they are two conflicting synchronization mechanisms. `blackholeHandler` is, at best, an awkward attempt to cope with the fact that the RTS forces threads to block even though "blocking" has become a concept defined by the concurrency library.

3.5.4 An Alternative to Blackhole Synchronization

An alternative to blackholing is to allow *reevaluation*—if two threads try to evaluate the same thunk simultaneously, simply let them do so. While this duplicates work, for small thunks, it may be less expensive than arranging to block. On a multiprocessor, synchronization is clearly still required when replacing the thunk with its memoized value—only one thread should perform the update, while the other's work should be discarded. However, this is straightforward and could be done completely in the RTS via spinlocks or atomic operations—it is completely invisible to the Concurrent Haskell programmer.

Reevaluation would cleanly solve all of Lighthouse's blackholing issues. In the nested transaction scenario, if the inner transaction could simply reevaluate, it would not need to wait for the outer transaction, so it could proceed—and ultimately, both would finish. Furthermore, there would be no `blackholeHandler`, so the reentrance problem would disappear, as would many scheduling issues, such as priority inversion

caused by blackholes.[3]

However, eliminating blackholes altogether would come at a high cost, both in terms of space and time. Simply leaving the thunk in place would restore the original space leak [11], making operations like `last [1..10000]` run in $O(n)$ instead of $O(1)$ space. To make matters worse, multiple threads may trigger this space leak at the same time.

This leads to the question of whether a hybrid solution would be possible—one which would keep blackholes, but eliminate the blocking aspect. GHC already has a method of "freezing" the state of a thunk's evaluation and writing it back to the heap—essentially, turning a blackhole back into a thunk that continues where evaluation left off [6]. This "resumable blackhole" support is already necessary if a thread is killed, since leaving unresolved blackholes is unsafe [23].

Rather than fully reevaluating a thunk, it may be possible to use this feature to migrate that thunk's evaluation from one thread to another, allowing the second thread to continue where the first left off. Naturally, this only makes sense if the thread originally evaluating the thunk is not currently running. If it is, then the second thread could busy wait until the evaluating thread finishes, or exhausts its time slice, at which point it could attempt to migrate the work. While this may seem wasteful, as long as one or the other is evaluating the thunk, both threads are effectively making progress.

The question is whether this could be done efficiently; deciding when to reevaluate and when to "pack up and continue elsewhere" would likely be key. For small thunks, reevaluation likely makes sense—some space loss is likely tolerable, and the amount of work required to coordinate between threads may eclipse the amount needed to simply compute the value. For very large thunks, it may make sense to try and migrate the

[3] If a low priority thread is evaluating a thunk when a second, high priority thread also demands the value, the high priority thread will block on the blackhole and be forced to wait until the low priority thread is finished—clear priority inversion.

evaluation. Some heuristic would need to be developed to determine which method to employ.

Further research in this direction would be beneficial, as it could both eliminate concurrency issues caused by blackhole synchronization, yet retain most of the space properties offered by blackholing. However, GHC modifications of this magnitude are definitely outside the scope of this project.

Chapter 4

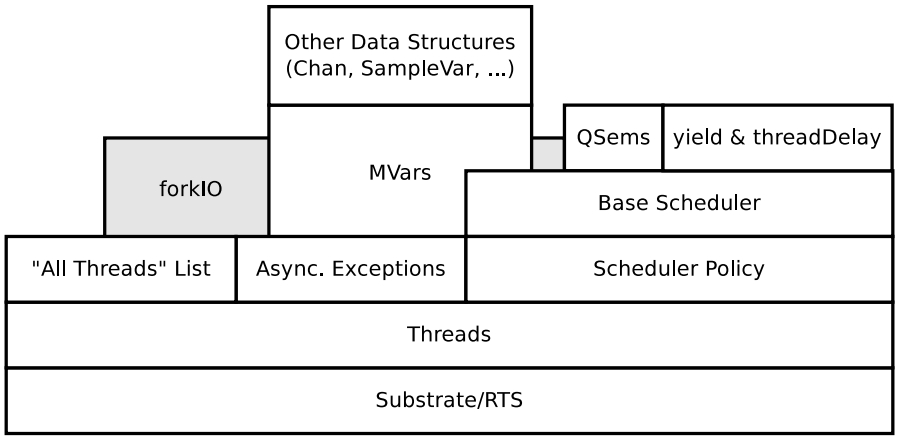
Implementing Concurrency Libraries

With the substrate in place, the next step is to build *concurrency libraries* on top of it. While the substrate may have been more challenging to implement, these libraries really are the meat of the system. For example, the substrate defines stack continuations, but a library expands on them to define full *threads*. It also offers an interface programmers can use to write concurrent programs without having to use the low-level substrate functions directly. Finally, the scheduler is now a library, as well; Lighthouse aims to implement a simple but flexible framework which allows one to easily experiment with different policies and scheduling algorithms.

As it stands today, House uses the standard Concurrent Haskell interface, so a sensible starting place is to implement Control.Concurrent and attempt to preserve its original semantics. While previous work [16] already explored this to an extent, doing so in the context of Lighthouse should provide a thorough testbed, offering greater insight into viability of Lightweight Concurrency. In particular, there are places where capturing GHC's original semantics is cumbersome—perhaps even impossible—without changes to the substrate or cripplingly coupled library code.

However, the ultimate goal is not to provide a 100% semantically equivalent implementation of Control.Concurrent—simply a concurrency library that is useful in the

Figure 4.1 Components of Lighthouse's Concurrency Library



Higher components are built on top of lower components.

context of House. Being a full fledged operating system, rather than an application, there are likely places where Lighthouse would be better served by a different interface.

Figure 4.1 shows the relationship between the components described in this chapter.

4.1 Threads

4.1.1 Thread Control Blocks (TCBs) and ThreadIds

While TLS can be used to store private information about a thread, there is a need for a place for "public" data that other threads can read or alter. For example, storing thread priority in TLS would prevent the scheduler (which runs in the context of other threads) from reading a thread's priority. It would also preclude other threads from boosting or altering a thread's priority, even given its ThreadId. Likewise, `killThread` and `throwTo` take a ThreadId and need to send an exception.

While one usually thinks of thread IDs as simple numbers, there is no particular reason this must be so—any unique identifier will work. In fact, standard GHC represents ThreadIds by a pointer to the TSO (section 3.1.1), as this is already a unique

identifier. However, doing so is somewhat suboptimal, as holding a thread's ID prevents the TSO—and hence whole stack—from being freed; it also seems unnecessarily tied to RTS internals.

Lighthouse defines ThreadIds as a Haskell-side *thread control block* (TCB) which implements this notion of "public" thread information:

```
data ThreadId = TCB Int (PVar (Seq Exception)) (PVar Priority)
```

The first component is simply a monotonically increasing number which allows for a nicer implementation of `show`. The second is a FIFO queue (based on Data.Sequence) of asynchronous exceptions sent to this thread, which are waiting to be raised (see section 4.4). Finally, the TCB also contains the thread's priority as defined in chapter 5. In the future, other subsystems may store additional public data in the control block.

Lighthouse also stores each thread's ID in TLS, allowing each thread to easily access its own control block. Since thread local state really is local, each thread must store the ID itself during its initialization code (section 4.3.3). However, a thread may get interrupted before it is fully initialized—say, due to a timer interrupt—and the scheduler will likely need to know when this happens. Hence, the TLS key uses a Maybe type with a default value of `Nothing`:

```
tidTLSKey :: TLSKey (Maybe ThreadId)
tidTLSKey = unsafePerformIO $ newTLSKey Nothing
```

4.1.2 Thread = TCB + SCont

Working with stack continuations directly is rarely convenient. Consider implementing a scheduler's run queue as a list of SConts. The scheduler needs each thread's continuation in order to switch to it—but the SCont itself says nothing about *what* thread it is,

making it impossible to tell which threads are on a particular queue.

Most of the time, it is far more convenient to use a TCB and SCont pair, which in Lighthouse is simply called a Thread:

```
data Thread = Thread ThreadId SCont
```

To facilitate their use, Lighthouse provides a wrapper around the substrate's `switch` primitive—called `switchT`—that operates at the Thread level:

```
switchT :: (Thread -> PTM Thread) -> IO ()
```

The tricky part is that `switch` only supplies the current thread's SCont, but the passed function also needs the TCB. However, this is not a showstopper—`switchT` can simply fetch the TCB from TLS and combine both pieces to get the full Thread.

This works well until `switchT` gets called from an uninitialized thread—one whose TLS still has `Nothing` for the ThreadId—in which case it's not possible to create the full Thread. The current implementation handles this by simply returning—refusing to switch—so that the thread will continue and finish initializing. This seems to be the most convenient; code that absolutely needs to switch can still use the unwrapped, original `switch`.

Figure 4.2 shows the code; the final `checkSignals` call handles asynchronous exceptions (section 4.4).

4.2 The OS Scheduler

Lighthouse's scheduler is divided into two parts—the base scheduler (used by the rest of the system), and a specific scheduler policy (only used by the base scheduler). This division allows one to easily experiment with various scheduling policies, while abstracting

Figure 4.2 Haskell code for the `switchT` wrapper

```
switchT f =
  do switch $ \currSC ->
       do m <- getTLS tidTLSKey
          case m of
            Nothing -> return currSC -- refuse to switch if uninitialized
            Just currTCB ->
              do (Thread nextTCB nextSC) <- f (Thread currTCB currSC)
                 return nextSC
     checkSignals -- handle asynchronous exceptions
```

Figure 4.3 The new OS scheduler interface

Base Scheduler Interface	**Policy Interface**
`timerHandler :: IO ()`	`timeUp :: IO Bool`
`getNextThread :: PTM Thread`	`getNextThread :: PTM (Maybe Thread)`
`schedule :: Thread -> PTM ()`	`schedule :: Thread -> PTM ()`
`scheduleIn :: Int -> Thread -> PTM ()`	

common functionality and keeping a consistent interface. Figure 4.3 shows the interface for both parts.

The scheduler is primarily *passive*—that is, instead of actively switching threads, it simply provides routines to fetch what should run next, or schedule threads appropriately. Other systems, such as MVars or Control.Concurrent's `yield`, will call `switchT`.

4.2.1 Base Scheduler

The base scheduler has a fairly small interface:

- `timerHandler :: IO ()`

  This is the handler for timer interrupts, called by `interruptHandler` on IRQ0. If the current thread is fully initialized, `timerHandler` consults the scheduler policy to see if it has exhausted its allotted time. If so, it simply calls `yield`. Otherwise, it returns.

- `schedule :: Thread -> PTM ()`

  This marks the given thread "ready" and schedules it to run sometime in the future.

- `scheduleIn :: Int -> Thread -> PTM ()`

  This is the backing for Control.Concurrent's `threadDelay` (section 4.3.4), providing a way to put a thread to sleep for a given amount of time. Once that time is up, the thread will be marked ready again.

- `getNextThread :: PTM Thread`

  This returns the next thread to run, as determined by the specific scheduler policy. Before consulting the scheduler policy, it also wakes up any sleeping threads who need to be marked ready.

  It may be that all real threads in the system are blocked, and there is nothing to run. In fact, this is not unusual—for example, the main text shell may be awaiting user input, which will come via some future keyboard interrupt. Even so, it does not return a `Maybe Thread`. Usually, this is called when switching threads; if the current one is about to block, there *must* be something else to switch to.

  To handle this, the base scheduler contains a single system idle thread which only runs when nothing else can. Since nothing could possibly be runnable until the next interrupt, the idle thread puts the CPU to sleep until the next hardware interrupt via the x86 `hlt` instruction (accessed by a foreign call to a one line C wrapper). Notably, the foreign call also functions as a safe point, allowing normal HsIRQ handling (section 3.2.2) to occur and possibly schedule some threads. The idle thread then simply calls `switchT` and `getNextThread` again, looking for any newly awakened threads.

4.2.1.1 Support for Sleeping Threads

A fairly natural way to implement sleeping threads is to store them in a single min-priority queue, sorted by wake time. Conveniently, the Data.Heap module (in package "heap") already implements such a data structure:

```
type SleepQueue = MinPrioHeap Int Thread
sleepQ :: PVar SleepQueue
sleepQ = unsafePerformIO $ newPVarIO Heap.empty
```

House's basic time unit is a "jiffy"—the length of time between consecutive timer interrupts. Furthermore, it keeps a count of how many jiffies have elapsed since system startup, providing a simple timestamp accessible via a tiny C function. Lighthouse includes a wrapper for this which returns the result in a PTM context; this is harmless since it is an idempotent operation—if the transaction retries, it will simply get a newer timestamp.

```
getJiffies :: Integral i => PTM i
```

Most code expects to use a real unit of time, though, so a helper function exists to convert microseconds to jiffies:

```
usecToJiffies :: Integral i => i -> i
usecToJiffies usec = usec `div` 20000
```

With these in place, writing `scheduleIn` is easy:

```
scheduleIn :: Int -> SCont -> PTM ()
scheduleIn usec thread = if jfs <= 0 then schedule thread else
  do now <- getJiffies -- current timestamp
     q <- readPVar sleepQ
     writePVar sleepQ (insert (now + jfs, thread) q)
  where jfs = usecToJiffies usec -- duration in jiffies
```

`wakeThreads` finds all threads who have slept long enough and schedules them.

```
wakeThreads :: PTM ()
wakeThreads =
  do q <- readPVar sleepQ
     now <- getJiffies
     let (elts, q') = Heap.span (\(when, _) -> when <= now) q
     writePVar sleepQ q'
     mapM_ (schedule . snd) elts
```

Making `getNextThread` call `wakeThreads` completes the implementation. Best of all, once this code type-checked and compiled, it worked on the first try. This was almost never the case when working on RTS-level C or C-- code. Although small, this is a great example of how high-level languages can be beneficial for systems programming—the obvious approach can be implemented straightforwardly. Haskell already provides key abstractions such as the priority queue and the ability to extract a list of elements matching a predicate. It also removes a whole class of potential errors due to memory mismanagement, and PTM conveniently takes care of any concurrency issues.

4.2.2 Scheduler Policy

While the base scheduler provides the interface to the rest of the world and common functionality, most of the actual scheduling work is done by the specific scheduling policy implementation. Currently, one cannot change schedulers at run time (though it is likely possible); instead, one chooses a scheduler via a line in the base scheduler:

```
import qualified LwConc.Scheduler.RoundRobin as SchedPolicy
```

Since one may implement many different scheduling policies, it is convenient to simply refer to them as "schedulers." The interface for a scheduler is rather minimal:

- `schedule :: Thread -> PTM ()`

Figure 4.4 Implementation of a round-robin scheduler atop Lighthouse's new scheduling framework.

```
readyQ :: PVar (Seq Thread)
readyQ = unsafePerformIO $ newPVarIO Seq.empty

schedule :: Thread -> PTM ()
schedule thread =
  do q <- readPVar readyQ
     writePVar readyQ (q |> thread)

getNextThread :: PTM (Maybe Thread)
getNextThread =
  do q <- readPVar readyQ
     case viewl q of
       (t :< ts) -> do writePVar readyQ ts
                       return (Just t)
       EmptyL    -> return Nothing

timeUp :: IO Bool
timeUp = return True
```

- `getNextThread :: PTM (Maybe Thread)`

  This is just like `getNextThread` in the base scheduler, except that it returns `Maybe Thread`—at the policy level, it is convenient to be able to say there is nothing to run and let the base scheduler take care of idling.

- `timeUp :: IO Bool`

  A scheduler may give certain threads more than one time slice before switching them out. On each timer interrupt, the base scheduler calls `timeUp` to let it know one time slice has expired, and to ask whether the current thread needs to be switched out.

4.2.3 A Round Robin Scheduler

Standard GHC simply uses round-robin scheduling, which is extremely easy to implement using Lighthouse's scheduling framework. Notably, the single ready queue is based on Data.Sequence (from the "containers" package), which offers a list-like structure with good performance when adding/removing items at either end. Figure 4.4 shows the code for the round-robin scheduler; chapter 5 presents more advanced schedulers.

4.3 Control.Concurrent Functions

Haskell's Control.Concurrent module provides a collection of basic but essential functions; these provide the ability to create new threads, yield, look up the current thread's ID, and wait for a certain number of milliseconds. The following sections give implementations for the subset of Control.Concurrent used in Lighthouse.

4.3.1 myThreadId

Since each thread's ID is stored in TLS (section 4.1.1), `myThreadId` is trivial to implement:

```
myThreadId :: IO ThreadId
myThreadId = do val <- getTLS tidTLSKey
                case val of
                  Nothing  -> error "uninitialized thread"
                  Just tid -> return tid
```

4.3.2 yield and die

`yield` is simple to implement atop `switchT`—it simply needs to pass it along to the scheduler, and request the next thread:

```
yield :: IO ()
yield = switchT $ \currThread -> do schedule currThread
                                    getNextThread
```

Lighthouse also includes a new function called `die` which switches to another thread, but does not schedule the current one again. This is critical, because when a thread wants to finish, it *must* switch to something else, or the system will grind to a halt. Once a thread is neither running nor scheduled, it should be garbage collected.

```
die :: IO ()
die = switch $ \_ -> do (Thread _ sc) <- getNextThread
                        return sc
```

Using the unwrapped version of `switch` allows even uninitialized threads to die.

4.3.3 forkIO

`forkIO` is based on the `newSCont` substrate primitive (section 2.5) but wraps the provided computation with special initialization and termination code. Notably, a thread must call `die` after performing its computation so that the scheduler can run something else. Otherwise, it will return to the RTS with a `ThreadFinished` status, resulting in the whole system shutting down. It also must call `die` in the case of an unhandled exception, so `forkIO` wraps the provided computation in a catch block. Finally, this initialization code also sets up the thread control block and ID (section 4.1.1).

```
forkIO :: IO () -> IO ThreadId
forkIO computation =
  do tid <- newThreadId
     sigsblocked <- getTLS blockSignalsKey
     newThread <- newSCont $ catchException
          (do -- Inherit exception-blocked status from parent.
              setTLS blockSignalsKey sigsblocked
```

```
            setTLS tidTLSKey (Just tid)
            checkSignals -- check if killed before first run
            computation
            die)
        (\e -> die)
```

`checkSignals` and `blockSignalsKey` are part of Lighthouse's asynchronous exception implementation, covered in section 4.4.

4.3.4 threadDelay

With the base scheduler's support for sleeping threads (section 4.2.1.1), implementing `threadDelay` is trivial:

```
threadDelay :: Int -> IO ()
threadDelay usec = switchT $ \currThread ->
  do scheduleIn usec currThread
     getNextThread
```

4.4 Asynchronous Exceptions

One of the key features programmers expect from a threading system is the ability to *kill* threads. In Concurrent Haskell, this is done via the following function:

```
killThread :: ThreadId -> IO ()
```

Beyond this, it is sometimes handy for threads to be able to interrupt or signal one another. Standard GHC offers the ability to raise an exception in a target thread via the `throwTo` function:

```
throwTo :: ThreadId -> Exception -> IO ()
```

In fact, by using an exception for thread kill, GHC is able to simply implement `killThread` in terms of `throwTo`:

```
killThread tid = throwTo tid (AsyncException ThreadKilled)
```

On the surface, this may merely seem to be convenient implementation, but it is actually very important—sending a catchable termination signal rather than outright killing gives the recipient a chance to release any resources it holds before dying. In House, one such example is the text console lock—if a thread dies while holding the lock, no one will ever be able to print again! By using an ordinary catch block, threads can safeguard against this. In fact, "exception-proof" wrappers such as `withMVar` do just this, returning the resource before re-throwing the caught exception.

While Lighthouse may not require asynchronous exceptions, it does need the ability to kill threads—and have them release their resources. In light of the above, supporting Concurrent Haskell's asynchronous exceptions seemed like a good idea.

4.4.1 Trouble with Semantics

While scant on details, both the Haskell Workshop paper [16] and Li's dissertation [15] propose a polling approach in which `throwTo` delivers an exception by writing it to a PVar in the target thread's local state. Each thread would check its PVar during context switching and raise the exception if one is present. The Haskell Workshop paper states that this is "not quite as responsive" as GHC's current implementation—the target thread may not receive the exception until it next runs or the end of its time slice.

"Not quite as responsive" is a significant understatement. GHC/Concurrent Haskell's semantics allow one to interrupt sleeping or blocked threads by sending them asynchronous exceptions. In doing so, the target thread is woken up and scheduled; it raises

the exception as soon as it runs [19]. This is extremely useful—if a thread is blocked indefinitely (or inordinately long), one would really like to be able to kill it. Having to wait until a thread's next run may mean that threads which, for example, are blocked waiting for user input are unkillable. This is undesirable, and is a marked departure from standard semantics.

Unfortunately, retaining this capability is tricky. `throwTo` would have to unblock the target thread—no matter whether it is blocked on an MVar or QSem, asleep due to `threadDelay`, or blocked on some other new data structure implemented by the concurrency library author. Not only must it be scheduled, but it *must* be removed from whatever blocked queue it was on—otherwise, another thread would likely try to wake it up via normal means and end up using the same SCont twice—a fatal error.

One potential approach could be to keep a thread status field in the TCB—either `Runnable SCont` or `Blocked (PTM Thread)`—not only signifying that a thread is blocked, but providing data structure specific code to extricate a thread from whatever queue it is blocked on. Then `throwTo` could check if a thread is blocked when throwing—atomically, via PTM—and extricate/reclaim the thread if so.

A second departure in semantics is that GHC's current implementation of `throwTo` is synchronous—it will not return until the target thread actually raises the exception. This provides a nice property where, if two threads both try to kill each other, only one will actually succeed [27]. This is likely less of an issue, however—the original paper on asynchronous exceptions in Haskell [19] proposed an asynchronous version of `throwTo`.

4.4.2 A Compromise

While not ideal, Lighthouse currently implements the basic polling approach outlined by the Lightweight Concurrency papers instead of trying to follow the standard semantics—it is simple and seems to be sufficient for the time being.

Each thread's control block (introduced in 4.1.1) contains the PVar for incoming asynchronous exceptions:

```
data ThreadId = TCB Int (PVar (Seq Exception)) (PVar Priority)
```

This makes it simple to implement `throwTo` since it already has the target ThreadId, and thus the PVar. It actually stores a queue of incoming exceptions (using the Data.Sequence module) since a thread could potentially receive asynchronous exceptions from multiple sources, handle them, and still wish to receive the rest. Lighthouse does not require this, however—it would be sufficient to use `PVar (Maybe Exception)` and overwrite the stored exception rather than queueing. In Lighthouse, `killThread` *does* override any other pending exceptions, since `AsyncException ThreadKilled` is essentially an ultimatum.

Each thread polls for pending asynchronous exceptions via a new function, `checkSignals :: IO ()`. However, this is done implicitly each time the thread runs, since both `forkIO` (section 4.3.3) and `switchT` (Figure 4.2) automatically call it.

Concurrent Haskell also provides two higher-order functions for controlling asynchronous exception delivery:

```
block   :: IO a -> IO a
unblock :: IO a -> IO a
```

The former turns off asynchronous exception delivery in the context of the supplied computation; the latter turns it back on temporarily when nesting the two. Prior work

on Lightweight Concurrency does not even mention these functions, but thankfully, they were fairly easy to implement using a simple boolean TLS key. They are also very necessary since `withMVar`, `modifyMVar`, and the like rely on them, and House makes extensive use of MVars. Figure 4.5 shows the code for `throwTo`, `killThread`, `checkSignals`, and `block`.

In the future, it would almost certainly be worth attempting to add the ability to kill blocked threads, using an approach like the one outlined above. In the long term, House will likely gain a notion of process—each process having several threads—at which point this should definitely be revisited. Likely it will be useful to have some kind of signaling mechanism at both the thread and the process levels, but it is unclear how Concurrent Haskell's asynchronous exceptions will fit into such a system.

4.5 MVars

MVars are Haskell's primary concurrent data structure, and the building block for almost all concurrent Haskell programs; House is no exception. Conceptually, an MVar is a box that holds at most one item. Threads can use this box to pass values to one another—one puts something in, while another takes it out. If a thread tries to take a value when the box is empty, it blocks, waiting until a value becomes available. Similarly, if a thread tries to store a value into a full box, it blocks until another thread empties the box. MVars ensure FIFO ordering for waiting threads.

Figure 4.6 shows the core MVar interface: creation (in either initial state), blocking take and put, as well as non-blocking variants. Notably, these are all RTS primitives in standard GHC—and thus written in C-- [13]—but in Lighthouse are implemented in Haskell atop PTM.

An MVar can be represented as a mutable PVar containing its state—either *full* with

Figure 4.5 Code for `throwTo`, `killThread`, `checkSignals`, and `block`

```
throwTo :: ThreadId -> Exception -> IO ()
throwTo tid@(TCB _ tbox _) exn =
  do atomically $ do exns <- readPVar tbox
                     writePVar tbox (exns |> exn)
     checkSignals -- in case we were the target

killThread :: ThreadId -> IO ()
killThread tid@(TCB _ tbox _) =
  do atomically $ writePVar tbox (singleton (AsyncException ThreadKilled))
     checkSignals -- in case we were the target

checkSignals :: IO ()
checkSignals =
  do mx <- atomically $
             do mtid <- mySafeThreadId
                sigsBlocked <- getTLS blockSignalsKey
                case (mtid, sigsBlocked) of
                  (Nothing, _) -> return Nothing
                  (_, True)    -> return Nothing
                  (Just tid@(TCB _ tbox _), _) ->
                    do exns <- readPVar tbox
                       case viewl exns of
                        (e :< es) -> writePVar tbox es >> return (Just e)
                        EmptyL    -> return Nothing
     -- We need to throw outside of the atomically so it does not retry.
     case mx of
       Nothing  -> return ()
       Just exn -> throw exn

block :: IO a -> IO a
block computation =
  do saved <- atomically $ getTLS blockSignalsKey
     setTLS blockSignalsKey True
     -- Catch synchronous (not asynchronous) exns so we unwind properly
     x <- catchException computation
                         (\e -> setTLS blockSignalsKey saved >> throw e)
     setTLS blockSignalsKey saved
     checkSignals
     return x
```

Figure 4.6 The core MVar interface (formerly primitives)

```
newEmptyMVar ::  IO (MVar a)
newMVar ::  a -> IO (MVar a)
takeMVar ::  MVar a -> IO a
putMVar  ::  MVar a -> IO ()
tryTakeMVar ::  MVar a -> IO (Maybe a)
tryPutMVar  ::  MVar a -> IO Bool
isEmptyMVar ::  MVar a -> IO Bool
addMVarFinalizer ::  MVar a -> IO () -> IO ()
```

a particular value, or *empty*. If full, it has a queue of blocked writers—both threads and the values they tried to publish. If empty, the queue is of blocked readers—threads waiting to receive a value—coupled with PVar receptacles for those values. This leads to the following data definitions:

```
newtype MVar a = MVar (PVar (MVState a)) deriving Eq
data MVState a =
     Full a (Seq (a, Thread))       -- queue of blocked writers
   | Empty  (Seq (PVar a, Thread)) -- queue of blocked readers
```

Figure 4.7 shows Lighthouse's implementation of `takeMVar` and `putMVar`, the two most essential functions. Since both of these functions may block the current thread, they have to use `switchT` rather than `atomically`. Recall its type signature from section 4.1.2:

```
switchT :: (Thread -> PTM Thread) -> IO ()
```

This seems problematic, since `takeMVar` needs to switch threads *and* return a value. One way to work around this is to have the PTM transaction store the value in a temporary PVar, then read its contents outside the `switchT`. In code, this might look like:

```
do hole <- newPVarIO undefined
   switchT $ do ...
                writePVar hole (value to return)
                return (thread to run next)
   value <- atomically $ readPVar hole
```

Figure 4.7 Implementation of `takeMVar` and `putMVar` (blocking MVar operations)

```
takeMVar :: MVar a -> IO a
takeMVar mv@(MVar p) =
  do hole <- newPVarIO undefined
     switchT $ \currThread ->
       do st <- readPVar p
          case st of
            Full x bq ->
              do writePVar hole x -- pass value back so it can be returned
                 case viewl bq of
                   EmptyL -> -- no waiting writers...now empty
                     do writePVar p (Empty empty)
                        return currThread
                   ((x',t) :< ts) ->              -- waiting writer to process:
                     do writePVar p (Full x' ts) -- publish their value
                        schedule t                -- wake them up
                        return currThread
            Empty bq -> -- block (queuing hole for answer)
              do writePVar p (Empty (bq |> (hole, currThread)))
                 getNextThread
     atomically $ readPVar hole

putMVar :: MVar a -> a -> IO ()
putMVar mv@(MVar p) x = switchT $ \currThread ->
  do st <- readPVar p
     case st of
       Empty bq ->
         case viewl bq of
           EmptyL ->
             do writePVar p (Full x empty) -- nobody waiting; just store the value
                return currThread
           ((hole,t) :< ts) ->
             do writePVar hole x -- pass value through hole directly to blocked reader
                writePVar p (Empty ts) -- take them off the blocked queue
                schedule t             -- put them back on the ready queue
                return currThread
       Full y bq -> -- block (queuing value to write)
         do writePVar p (Full y (bq |> (x, currThread)))
            getNextThread
```

Figure 4.8 Implementation of `tryTakeMVar` and `tryPutMVar` (non-blocking variants)

```
tryTakeMVar :: MVar a -> IO (Maybe a)
tryTakeMVar (MVar p) = atomically $
  do st <- readPVar p
     case st of
       Full x bq ->
         do case viewl bq of
              EmptyL -> writePVar p (Empty empty)
              ((x',t) :< ts) -> do writePVar p (Full x' ts)
                                   schedule t
            return (Just x)
       Empty bq -> return Nothing

tryPutMVar  :: MVar a -> a -> IO Bool
tryPutMVar (MVar p) x = atomically $
  do st <- readPVar p
     case st of
       Empty bq ->
         do case viewl bq of
              EmptyL -> writePVar p (Full x empty) -- no one waiting, just store
              ((hole,t) :< ts) ->
                do writePVar hole x -- pass value through hole to blocked reader
                   writePVar p (Empty ts) -- remove reader from blocked queue
                   schedule t
                   return True
       Full y bq -> return False
```

While it might seem beneficial to abstract this pattern, it is useful to work with the PVar directly. Consider calling `takeMVar`: if the box is full, the PTM transaction can simply return the value via the PVar `hole`, as in the code above. However, if the box is empty, the value must come from another thread! In this case, `takeMVar` stores `hole` on the queue of blocked readers in addition to the reader's Thread. When another thread calls `putMVar`, it stores the new value in `hole` before waking the reader, passing the value through to the other thread. The MVar will still by empty—putMVar passes the value directly to the first blocked reader via its `hole`.

Figure 4.9 "Exception proof" MVar wrapper functions

```
readMVar :: MVar a -> IO a
swapMVar :: MVar a -> a -> IO a
withMVar :: MVar a -> (a -> IO Bool)
modifyMVar_ :: MVar a -> (a -> IO a) -> IO ()
modifyMVar  :: MVar a -> (a -> IO (a, b)) -> IO b
```

Figure 4.8 shows Lighthouse's implementation of the non-blocking variants (`tryPutMVar` and `tryTakeMVar`). These are much simpler since they do not need to switch threads, and thus can use `atomically` instead of `switchT`. Lighthouse does not implement `addMVarFinalizer`, as it removes all finalizer support from the RTS for reasons discussed in section 3.2.3.1.

4.5.1 "Exception-proof" Wrappers

The Concurrent Haskell interface also includes several MVar convenience functions—shown in Figure 4.9—which allow one use MVars safely in the presence of exceptions. For example, `modifyMVar_` takes an MVar's value, operates on it, and stores the new value—but attempts to restore the original value if an exception arises.

This "exception-proof" property is very useful. In House, the text mode console buffer is stored in an MVar, and a thread may be writing to the console when it receives `AsyncException ThreadKilled`. These functions ensure it releases the console before dying—otherwise, nothing would be able to write to the console again!

Fortunately, standard GHC already implements these functions in Haskell, based on `takeMVar`/`putMVar` and `block`/`unblock` (section 4.4.2). Lighthouse simply reuses the existing GHC implementation.

4.6 Other Data Structures

Likewise, all the other Concurrent Haskell data structures—Chan, QSem, QSemN, and SampleVar—are written in Haskell using MVars; the existing implementations work "out of the box." However, Lighthouse does reimplement QSems (section 5.3.1) to solve priority inversion issues.

4.7 An "All Threads" List

A potentially useful operating system feature would be an "all processes" list, such as that offered by the UNIX `ps` command. In House, this would correspond to a Haskell-side "all threads" list. Implemented naïvely , such a list would prevent threads from ever being garbage collected, since it would hold references to every thread in the system. While one could devise workarounds for this, the simplest solution is to implement the list using GHC's weak pointers:

```
allThreads :: PVar [Weak ThreadId]
```

This way, the list does not hold a full reference to the threads, and they can be garbage collected as normal. When a thread is garbage collected, its weak pointer will automatically be reset to `Nothing`. Thus, over time, the list will accumulate many `Nothing` entries corresponding to dead threads. Periodically, these entries should be cleaned out via the following functions:

```
cleanWeakList   :: [Weak a] -> PTM [Weak a]
cleanAllThreads :: PTM ()
```

One troublesome point is that `deRefWeak` is in the IO monad, while `cleanWeakList` must be in PTM since access to the list must be protected against concurrent updates.

To work around this, Lighthouse provides a version of `deRefWeak` that is in the PTM monad. This is safe, however—if the transaction calling `deRefWeak` obtains `Just t`, it holds a true reference which prevents `t` from being spontaneously garbage collected. Should the transaction need to retry, it will call `deRefWeak` again and may obtain `Nothing`—but this is simply a newer state.

One might also consider implementing this list cleaning via GHC's finalizers, but Lighthouse does not support finalizers for reasons mentioned in section 3.2.3.1.

While the all threads list will likely be very useful in the future—say, for implementing a command similar to UNIX's `kill`—Lighthouse currently only uses it to list the system's live threads. The following is some simple code to do so:

```
showAllThreads :: IO String
showAllThreads = do ws <- atomically $ do cleanAllThreads
                                          readPVar allThreads
                    strs <- mapM (liftM show . deRefWeak) ws
                    return (show strs)
```

This information is very helpful when debugging the concurrency library.

Chapter 5

Priorities and Scheduling Policy

One of the key features House lacks is a notion of thread priority—after running a few compute-bound tasks (infinite loops), the system grinds to a standstill. This is particularly noticeable in graphical mode: one can see the screen progressively drawing, and the keyboard and mouse become virtually unusable. Primarily, this is due to GHC's simple round-robin scheduler—threads handling these critical system tasks run at the same priority as everything else, intermixed with supposed "background" tasks.

Lighthouse remedies this by adding basic extensions for thread priority, making it possible to lower the priority of these compute-bound tasks compared to the interactive portions of the system. In addition to making the system much more usable, such support seems desirable in its own right—most operating systems have a notion of priority. Extending House with priorities brings it closer to being a "real operating system."

With the Lightweight Concurrency framework in place, adding basic support for priority is a breeze—everything is written in Haskell, without any modifications to complicated RTS code.

5.1 Extensions for Priority

Defining priority is straightforward. A basic definition might look like:

```
data Priority = Low | Medium | High
  deriving (Show, Eq, Ord, Bounded, Enum, Ix)
```

One may want more levels of priority than this—for example, UNIX has forty. However, by using Haskell's type classes, most code does not actually have to care about the exact levels of priority—it can simply use `[minBound .. maxBound]` to get a list of all priorities.

Lighthouse offers the following interface for accessing and altering thread priority:

```
getPriority   :: ThreadId -> PTM Priority
setPriority   :: ThreadId -> Priority -> PTM ()
myPriority    :: PTM Priority
setMyPriority :: Priority -> PTM ()
```

In particular, any thread can set another thread's priority. This seemed useful for priority inheritance, but one could certainly remove or restrict `setPriority` if desired. In contrast, `getPriority` is essential for the system to function—if thread *A* unblocks and schedules thread *B*, it has to be able to access thread *B*'s priority in order to place it on the proper queue.

Lighthouse stores priority in the TCB—which conveniently happens to be the ThreadId (see section 4.1.1):

```
data ThreadId = TCB Int (PVar (Seq Exception)) (PVar Priority)
```

`getPriority` and `setPriority` simply write the new priority to that PVar; the other two obtain the current thread's TCB from TLS, and then do likewise. If the thread is not yet fully initialized—where the TCB is not yet registered in TLS—`myPriority` returns `maxBound` (so it will get initialized as soon as possible), and `setMyPriority` does nothing.

5.1.1 Timeliness

Since all these functions do is set a variable, setting a thread's priority may not boost it immediately. For example, assume the system uses a multilevel queue scheduler, and that thread *A* is low priority and ready to run—and thus in the scheduler's "Low" ready queue. If thread *B* then runs `setPriority` *A* `High`, thread *A* will still live in the "Low" queue until it actually runs—the system makes no attempt to move it from one to the other. However, after it runs, it will be properly placed in the "High" queue.

5.2 Scheduling Policies

One of the goals of Lighthouse is to create an *extensible* scheduler framework (described in section 4.2) in which it is easy to experiment with different policies. With the framework in place, it remains to create a scheduling policy that better serves House.

The following sections detail a number of such scheduling policies. While none are particularly novel or sophisticated, each embodies an idea that one could use in creating a good scheduler. As it turns out, almost anything is an improvement over House's traditional round-robin scheduler.

5.2.1 Extra Time Slices (Longslice)

The idea behind the first scheduler, *Longslice*, is to simply allow higher priority threads to run for several time slices before being switched out. Beyond that, it is simply a round-robin system. Figure 5.1 contains the new `timeUp` implementation; the rest of the code is the same as that given in section 4.2.3.

For each thread, Longslice must store a counter of how many time slices are left before the thread should be switched out. This is easy with TLS—all it needs to do is

Figure 5.1 Implementation of the Longslice scheduling policy

```
ticksKey :: TLSKey Int
ticksKey = unsafePerformIO $ newTLSKey 0

extraTicks :: Priority -> Int
extraTicks p = 10 * fromEnum p

timeUp :: IO Bool
timeUp =
  do slicesLeft <- getTLS ticksKey
     if slicesLeft < 1
        then do p <- atomically myPriority     -- out of time
                setTLS ticksKey (extraTicks p) -- reset count
                return True
        else do setTLS ticksKey (slicesLeft-1) -- used up a slice
                return False                   -- keep going
```

create a single key (`ticksKey`). Since the Longslice module does not export `ticksKey`, no other code can access this counter, preventing threads from tampering with it and giving themselves more time.

With the counter in place, `timeUp` simply checks if the current thread has used all its time slices. If so, it resets the count to the number of slices allowed for its priority, and returns True, indicating the thread should be switched out. Otherwise, it simply decrements the count and returns False, allowing the thread to continue.

Granting extra time slices should reduce scheduling overhead and increase throughput at the expense of higher latency, favoring compute-bound tasks over IO-bound ones. While this seems like the wrong thing to favor when trying to improve House's interactivity, it still may be helpful when combined with other techniques. Notably, the number of extra time slices given to each priority can easily be altered, and do not need to be distributed evenly. For example, one could grant low priority tasks only one time slice, but define a special "extremely urgent" priority that gets 1,000 extra slices in the hopes

that the task will run to completion.

5.2.2 Multilevel Queue Scheduler (Multilevel)

Lighthouse's second scheduler, *Multilevel*, provides a separate run queue for each priority level and draws from higher priority queues more often. It aims to offer lower latency at higher priorities, allowing critical tasks to run earlier and more often. This can be particularly beneficial for IO-bound tasks.

Each time `getNextThread` is called, it selects a priority, then returns the first thread in that priority's run queue. If there are none, it falls back to a lower priority queue. To select a priority, it cycles through a static list. For example, if priority $A > B > C > D > E$, Multilevel might select the following:

$$A, AB, ABC, ABCD, ABCDE, A, AB, ABC, ABCD, ABCDE, ...$$

Figure 5.2 shows the code. `readyQs` is a global array of ready queues, indexed by priority for easy access. `schedule` simply places the given thread on the appropriate queue. `priorityBox` contains an infinite, repeating list of which priorities to draw from next; `getNextPriority` selects one. Finally, `getNextThread` selects a priority and tries to pull the first thread from that priority's ready queue. If it is empty, it retries at a lower priority—pulling from a different queue—and ultimately returns Nothing if all queues are empty.

Using a list is convenient since it allows one to try various priority orderings. In particular, another potentially useful order is:

$$A, AAB, AABABC, AABABCD, AABABCDABCDE$$

Figure 5.2 Implementation of the Multilevel scheduling policy

```
type ReadyQ = PVar (Seq Thread)
readyQs :: Array Priority ReadyQ
readyQs = listArray (minBound, maxBound) (unsafePerformIO $ sequence
           [newPVarIO Seq.empty | p <- [minBound .. maxBound :: Priority]])

priorityBox :: PVar [Priority]
priorityBox = unsafePerformIO $ newPVarIO $ (concat . repeat) priorityOrder
  where priorityOrder = [A,A,B,A,B,C,A,B,C,D,A,B,C,D,E]

getNextPriority :: PTM Priority
getNextPriority =
  do (p:ps) <- readPVar priorityBox
     writePVar priorityBox ps
     return p

getNextThread :: PTM (Maybe Thread)
getNextThread = do priority <- getNextPriority
                   tryAt priority
  where tryAt priority = do let readyQ = readyQs ! priority
                            q <- readPVar readyQ
                            case viewl q of
                              (t :< ts) -> do writePVar readyQ ts
                                              return (Just t)
                              EmptyL -> if priority == minBound
                                           then return Nothing
                                           else tryAt (pred priority)

schedule :: Thread -> PTM ()
schedule thread@(Thread tcb _) =
  do priority <- getPriority tcb
     let readyQ = readyQs ! priority
     q <- readPVar readyQ
     writePVar readyQ (q |> thread)
```

This gives higher priorities even lower latency, yet still interleaves them.

Finally, note that this approach can easily be combined with Longslice to offer both more frequent execution *and* longer time slices.

5.2.3 Heuristic-based Scheduler (Dynamic)

While Multilevel can guarantee that one priority will run more often than another, it makes no guarantee that a particular *task* will run more often. To illustrate, let there be two priorities, A and B, and assume Multilevel's priority order is A, A, A, B. Then, let the run queues be the following:

A	a_1	a_2	a_3	a_4
B	b_1			

Assuming nothing blocks, the order in which the tasks would run is:

$$a_1, a_2, a_3, b_1 / a_4, a_1, a_2, b_1 / a_3, a_4, a_1, b_1 / a_2, a_3, a_4, b_1 / \cdots$$

In sixteen time slices, twelve would be spent running A-class work, while only four would be spent running B-class work—three times as many, as one would expect from the A, A, A, B ordering. Yet b_1 received *four* time slices, while each of the a_i only received three! Even at a lower priority, $b_1$was able to get more work done.

Lighthouse's *Dynamic* scheduler attempts to remedy this. The basic idea is to run the current priority's *whole queue* before moving on, rather than just a single task. In the same situation, Dynamic's run order would look like:

$$\underbrace{a_1, a_2, a_3, a_4}_{A}, \underbrace{a_1, a_2, a_3, a_4}_{A}, \underbrace{a_1, a_2, a_3, a_4}_{A}, \underbrace{b_1}_{B}, \cdots$$

In one cycle, each of the a_i received three time slices, while b_1 only received one. Overall, the system spent nine times as much time processing A-class work as B-class work.

In general, for any two priorities A and B, if A appears n times as often as B in the priority ordering, the two schedulers make the following claims:

- Multilevel: Overall, the system will be running A-class work n-times as often B-class work.
- Dynamic: Each task in A should run n-times as often as *any* task in B.

The latter is a much better claim—programmers boost a thread's priority because they want it to go relatively faster and likely have no idea how many tasks are at either priority level—nor should they have to care. Since it works so well, Dynamic is Lighthouse's default scheduler.

Note that Dynamic's property does not imply Multilevel's. Consider the following scenario:

A	a_1			
B	b_1	b_2	b_3	b_4

Multilevel: $a_1, a_1, a_1, b_1, a_1, a_1, a_1, b_2, a_1, a_1, a_1, b_2, a_1, a_1, a_1, b_2$

Dynamic: $\underbrace{a_1}_{A}, \underbrace{a_1}_{A}, \underbrace{a_1}_{A}, \underbrace{b_1, b_2, b_3, b_4}_{B}$

Here, Dynamic's property still holds—a_1 would run for three time slices, while each of the b_i would only run for one. Yet, Multilevel's property does not hold: overall, the system actually spent more time processing B-class work.

Implementing the notion of "running the entire queue" poses a challenge, since the queue is always changing. New tasks may be added at any time, and tasks which run

Figure 5.3 Implementation of the Dynamic scheduling policy

```
priorityBox :: PVar [(Int, Priority)]
priorityBox = unsafePerformIO $ newPVarIO $ []

getNextPriority :: PTM Priority
getNextPriority =
  do l <- readPVar priorityBox
     case l of
       [] -> do est <- mapM (\prio -> do q <- readPVar (readyQs ! prio)
                                         return (Seq.length q, prio))
                            order
               -- skip empty priorities:
               writePVar priorityBox (filter (\(x,_) -> x /= 0) est)
               return A
       ((1,p):ps) -> do writePVar priorityBox ps
                        return p
       ((n,p):ps) -> do writePVar priorityBox ((n-1,p):ps)
                        return p
  where order = [A,B,A,B,C,A,B,C,D,A,B,C,D,E]
```

will likely be enqueued again, making the queue appear cyclic. To solve this, Dynamic uses the length of the run queues as a heuristic. Every time it cycles through the priority order, it notes the length of the run queues, and decides to run that many tasks before moving to the next priority. If no new tasks are added, this heuristic gives precisely the desired behavior; otherwise, it should still be fairly close.

Figure 5.3 shows the new code for `getNextPriority` and `priorityBox`; everything else is the same as the Multilevel scheduler presented in Figure 5.2.

5.2.4 Stochastic Scheduler (Chance)

Lighthouse's *Chance* scheduler takes a non-deterministic approach. Starting with the highest priority, it randomly rolls a die to see if a lower priority will get a chance to run—currently, only a 10% chance, though this is easy to change. If nothing is available

at the selected priority, it will fall back to a lower priority.

The only change from Multilevel (Figure 5.2) is the `getNextPriority` function:

```
getNextPriority :: PTM Priority
getNextPriority = helper maxBound
  where helper x | x == minBound = return x
        helper x = do r <- d100 -- roll a 100 sided die
                      if r < 10 -- % chance of trying a lower priority.
                         then helper (pred x)
                         else return x
```

The `d100` function generates a random number between 0 and 99. Unfortunately, it does not use the standard System.Random module; Lighthouse's schedulers must be in GHC's `base` package, and the module simply had too many dependencies to feasibly include in `base`. Instead, it uses a simple linear congruential generator.

5.2.5 Priority Inversion Debugging (FixedHigh)

FixedHigh is a variant of Multilevel that always draws from the highest priority queue, only pulling from lower ones if there is nothing to run at the desired level. This leads to terrible starvation, but can be very useful for debugging priority inversions—it is entirely consistent, predictable, and offers a stark difference between high and low priority that should be easy to see.

5.2.6 Summary

The above sections present a variety of priority-based scheduling policies. Some aim to increase throughput for CPU-bound tasks, while others offer reduced latency for IO-bound workloads. Each take a different approach when deciding what to run next—granting extra time slices, multilevel queueing, heuristics, even non-determinism.

Regardless of whether or not these are great scheduling policies, they demonstrate that it is now very easy to experiment, and even extend the system with new notions such as priority. With Lightweight Concurrency and Lighthouse's scheduling framework, new policies can be written in straightforward readable Haskell code, as opposed to C. This greatly benefits House as it claims to be written in Haskell.

Furthermore, this offers a lot of flexibility. Lighthouse can now include multiple scheduling policies targeted at particular workloads; selecting one is as easy as changing one line and rebuilding. Should a particular workload have more unique needs—perhaps even a different priority system—it would not be terribly difficult to write a new scheduler which is custom tailored to the situation.

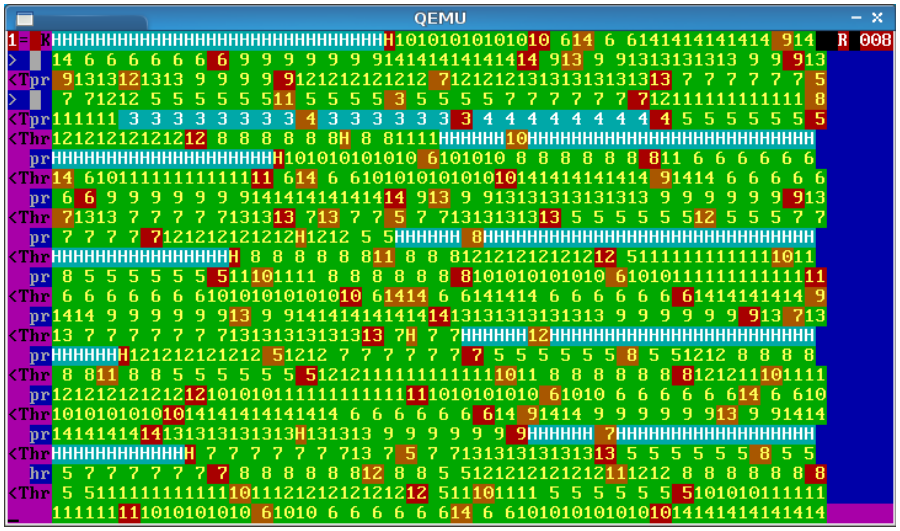
By default, Lighthouse uses the Dynamic scheduler. With it, the user interface remains responsive even in the presence of many computationally intensive but low priority background tasks. However, this was not originally the case due to several priority inversion issues. This is hardly surprising though, seeing as neither Concurrent Haskell nor House were designed with thread priority in mind. The next section discusses the problem and presents various techniques to solve it.

5.3 Priority Inversion

While Concurrent Haskell's MVar interface is widely used, systems built using it can easily suffer from priority inversion. Since MVars are inherently FIFO, threads trying to acquire an unavailable resource simply queue up with no regard to their priority. While this is sometimes the desired effect, it is definitely not in the case of semaphores/mutexes—one common use for MVars.

Priority inversion is no academic problem—in fact, this FIFO locking makes or breaks the usability of the system. One telling test is to run many low priority infinite

Figure 5.4 Visualizing priority inversion



Useful/high priority work (cyan) is repeatedly interrupted by low priority tasks (green).

loops in the background while running useful work at high priority. Even running the simple userspace "hello world" program was very slow, taking *seconds* to run, painstakingly printing one letter at a time. No matter how sharp of a distinction the scheduler drew between priority levels—even the FixedHigh scheduler—it had absolutely no impact.

Figure 5.4 shows a visual history of thread execution during the "hello world" test. Each time a thread ran, the scheduler printed its ID to the console (left to right, top to bottom). Threads 1–4 were system threads, such as the display and keyboard drivers; threads 5–14 were `minBound` priority infinite loops, and finally "H" was the `maxBound` priority "hello world" program. Cyan threads represent potentially "useful" work, and green threads represent useless looping. Red indicates when a thread blocked on an MVar, and brown shows when it became unblocked/runnable.

With this diagram, it became readily apparent that the tasks were queuing up on some shared MVar and suffering from priority inversion—or really, priority ignorance!

Figure 5.5 QSem implementation atop PTM; no longer single wake-up

```
newtype QSem = QSem (PVar (Int, [Thread]))

newQSem :: Int -> IO QSem
newQSem init = do sem <- newPVarIO (init,[])
                  return (QSem sem)

waitQSem :: QSem -> IO ()
waitQSem q@(QSem pv) =
  do bucket <- newPVarIO False
     switchT $ \currThread ->
                  do (avail,ts) <- readPVar pv
                     if avail > 0
                        then do writePVar pv (avail-1,[])
                                writePVar bucket True
                                return currThread
                        else do writePVar pv (avail, currThread:ts)
                                getNextThread
     gotIt <- atomically $ readPVar bucket
     unless gotIt $ waitQSem q

signalQSem :: QSem -> IO ()
signalQSem (QSem pv) = atomically $
  do (avail,blocked) <- readPVar pv
     writePVar pv (avail+1,[])
     mapM_ schedule blocked -- wake them all
```

The only question was which one. The culprit turned out to be something in House's userspace binary support, and was actually a QSem (quantity semaphore).

5.3.1 QSem Reimplementation

In standard GHC, QSems are single wake-up and implemented using MVars, and thus inherit the FIFO priority issue. However, there is really no reason for semaphores to be FIFO—most implementations make few guarantees (if any) on acquisition ordering.

To solve this, Lighthouse reimplements QSems directly atop PTM; Figure 5.5 con-

tains the entirety of the code. Rather than waking up one thread, Lighthouse's implementation wakes up all waiting threads when a semaphore becomes available, letting them compete to acquire it. This has greater overhead than single wake up, but it lets the scheduler decide which thread will run first (and thus acquire the semaphore), rather than enforcing a static scheduling decision. However, if one wished to keep the single wake up property, it would be easy to scan the list of blocked threads and select the one with the highest priority.

This reimplementation solved the priority inversion problem, and allowed the "hello world" program to print instantly.

5.3.2 Priority Inheritance

Of course, traditional priority inversion is still a problem: If a low priority thread L holds a resource and a high priority thread H wants to acquire it, H will have to wait. It effectively makes progress as if it were low priority. A common solution to this is "priority inheritance"—temporarily boosting a lock holder's priority to the maximum of any waiting threads.

Lighthouse includes an experimental Control.Concurrent.Lock module which implements mutexes with priority inheritance. Figure 5.6 shows the data types and exported interface.

When thread A attempts to `lock`, it first checks the PVar. If `Unlocked`, it writes `Locked` A `[]`, claiming the lock (assuming the transaction commits). It then returns a `LockKey`—essentially a piece of code which unlocks it. If `Locked` B `ts`, it compares its priority to that of B, boosting B's priority if necessary. Then it adds itself to `ts`, the list of threads waiting for the mutex, and switches to some other thread. To unlock, a thread calls unlock with the key, whose code resets the lock state to `Unlocked` and wakes up

Figure 5.6 Control.Concurrent.Lock interface for mutexes with priority inheritance

```
data LockState = Unlocked | Locked ThreadId [SCont] (not exported)
newtype Lock = Lock (PVar LockState) (abstract)
newtype LockKey = LockKey (IO ()) (abstract)
newLock  :: IO Lock
lock     :: Lock -> IO LockKey
tryLock  :: Lock -> IO (Maybe LockKey)
unlock   :: LockKey -> IO ()
withLock :: Lock -> IO a -> IO a
```

any waiting threads so they can try again and compete for the lock.

Although the `lock`/`unlock` interface supports boosting priority, it does not reset it once done—the thread must do that itself. Since this is rather cumbersome, Lighthouse provides a simpler interface which should suffice in most cases. `withLock` saves the thread's priority, takes the lock, performs the supplied computation, releases the lock, then resets the thread's priority back to what it was. This is very easy to use; the only caveat is that if the supplied computation changes the thread's priority, that change will be lost when priority is reset.

Chapter 6

Conclusions

Lighthouse, an experimental variant of House based on Lightweight Concurrency, offers many advantages over House.

First, priority-based scheduling noticeably improves system responsiveness. When running a few background tasks, House would slow to a crawl, rendering the GUI almost useless. In contrast, Lighthouse can run a great number of low-priority background tasks and still remain usable. Furthermore, priority scheduling is a common feature expected in most "real operating systems."

Secondly, Lighthouse's scheduling framework is extensible, making it possible to implement new policies in straightforward, readable Haskell code. This makes it feasible for developers to experiment and compare different scheduling algorithms, alter the notion of priority, or extend the system with new concepts. In particular, users deploying Lighthouse could write a scheduler custom-tailored to a particular workload.

Finally, Lighthouse contains approximately 10,000 lines less C and C-- code than House, offering a stronger case for being "written in Haskell."

This raises the question: should Lighthouse become the next version of House? Even with the above advantages, it is not clear that it should.

One concern is whether Lighthouse is maintainable. House is based on a set of

patches against standard GHC which allows it to run on "bare metal" x86 hardware. These patches must be periodically rebased to apply to new versions of GHC, and while Lighthouse's RTS is simpler than House's, it is a greater delta from upstream GHC. This is especially true since the GHC developers have not yet incorporated Lightweight Concurrency due to performance concerns. Furthermore, even if they did, they would not likely use Lighthouse's uniprocessor implementation of the substrate. This would shift more of the maintenance burden from the GHC developers to the House developers—a much smaller group. On the other hand, the existing House patches may already be unmaintainable—before work on Lighthouse began, House was still based on the highly outdated GHC 6.2, and had to be updated to 6.6 and later 6.8. At the time of writing, GHC 6.12 is the latest release.

Lighthouse's simplified RTS is also missing a few features compared to standard GHC. For one thing, it does not support full STM. While this could likely be implemented as a new layer atop PTM, initial work in that direction looked to be challenging—and tangential to the main work since House currently does not use STM. Secondly, Lighthouse does not support GHC's finalizers since they require the RTS (garbage collector) to be able to schedule threads, which is impossible without extending the Lightweight Concurrency substrate. Section 3.2.3.1 proposes such an extension.

Being primarily a change in implementation, one would expect Lightweight Concurrency to support Concurrent Haskell's asynchronous exception semantics; no prior work mentions otherwise. However, it turns out that writing library code to do so is not at all straightforward. The original system woke blocked threads in order to deliver asynchronous exceptions, but this would require tight integration with every kind of blocking data structure. Lighthouse instead implements the approach suggested by the original Lightweight Concurrency proposal [16]: threads check for pending asyn-

chronous exceptions during context switches—that is, when they run. Unfortunately, since asynchronous exceptions are used to kill threads, this means that blocked threads become unkillable—a serious regression.

Curiously, the original Lightweight Concurrency paper [16] goes so far as to claim that implementing `throwTo` on top of the substrate is "a breeze" compared to GHC's existing implementation, but then lists asynchronous exception support as missing from the reference implementation. This work has shown that implementing asynchronous exceptions atop Lightweight Concurrency is anything but a breeze. Section 4.4 offers some preliminary ideas on how to improve the situation.

Perhaps the weakest part of the Lightweight Concurrency framework, and hence Lighthouse, is the blackhole handling. Currently, Lighthouse crashes if two threads try to concurrently evaluate a thunk; it does not yet implement Lightweight Concurrency's `blackholeHandler`. This is primarily due to two issues that must be solved in order to have a working implementation: reentrance, and self-deadlock.

In Lightweight Concurrency, `blackholeHandler` can reenter—that is, `blackholeHandler` itself can encounter a blackhole. Prior work [15,16] suggests using TLS to detect such reentrance and to busy wait when it occurs—but this does not work in a uniprocessor setting. The only way to handle reentrance would be to switch to another thread, which now involves Haskell operations which may well blackhole. In light of this, section 3.5.1 proposes prohibiting reentrance altogether, and offers a way to accomplish this.

Section 3.5.2 presents a far more serious issue—self-deadlock. It is entirely possible that a thread may be evaluating a thunk when it is hijacked to run an interrupt handler which needs to evaluate that same thunk. The handler will hit a blackhole, and cannot proceed until the evaluation is finished—but it never will, since the interrupt handler is

higher up on the same stack performing the evaluation. Even worse, this self-deadlock will almost certainly cause the whole system to fail.

This issue is not limited to Lighthouse, but exists in Lightweight Concurrency itself. While Lighthouse uses thread hijacking for all kinds of interrupts, the original formulation of Lightweight Concurrency uses the same mechanism to handle timers. A thread might be in the middle of forking—and thus adding a thread to the scheduler's run queue—when a timer occurs and `timerHandler` needs to consult that very queue.

In reality, the issue is that blackholing has become a synchronization mechanism in and of itself—a thread may encounter a blackhole and be forced to block at any time. This is terrible from a systems programming perspective; the concurrency library author has no control over the situation. `blackholeHandler` is, at best, an awkward attempt to cope with the fact that the RTS still forces threads to block even though "blocking" has become a concept defined by the concurrency library.

Section 3.5.4 proposes altering GHC to remove the blocking aspect of blackholing, either via reevaluation (for small thunks) or by using the resumable blackhole mechanism [6, 23] to move evaluation from one location to another. This would eliminate the need for `blackholeHandler` altogether, solve the self-deadlock issue, and best of all relegate blackholing to its place as a transparent RTS detail. It is clear that this approach would be expensive; the question is whether it can be done efficiently enough to be acceptable.

Overall, Lightweight Concurrency works well; this work has shown that it truly can be used as the basis for a multitasking operating system, and that House would benefit greatly from using it. However, a solution for these issues—especially blackholing—must be developed before Lighthouse can truly replace traditional House.

6.1 Haskell and Systems Programming

Haskell offers many advantages over traditional languages used for systems programming. It is both type-safe and memory-safe, preventing whole classes of bugs. Being pure, it is also easier to reason about than traditional side-effecting code. As with most high-level languages, Haskell also provides many useful libraries—in particular data structures—and is very expressive, making it easy to prototype experimental features.

Section 4.2.1.1 presents the concurrency library code to support sleeping threads (i.e. `threadDelay`). Although small, it is a great example of how Haskell can be beneficial for systems programming. The obvious approach can be implemented in a mere twenty lines of code using the existing Haskell priority queue/heap module. Haskell is also expressive enough that abstract tasks such as extracting a list of elements from the heap which match a predicate can be done concisely; the idea is not lost in the details. Haskell prevents a whole class of potential errors due to memory mismanagement, and PTM conveniently takes care of any concurrency issues with minimal effort. Best of all, once the code type-checked and compiled, it worked on the first try. This was almost never the case when working on RTS-level C or C-- code. The rest of the scheduling code, including the stochastic and heuristic-based multilevel queue schedulers presented in chapter 5, are also straightforward.

Of course, Haskell is not a silver bullet; system programming is still challenging. Other portions of the system are more complex, and took more effort to implement correctly. For example, deciding what the scheduler should do when there are no threads to run—it is not immediately obvious why simply busy-waiting when trying to fetch the next thread does not work, and must be done in a dedicated idle thread.[1] Nor is it obvious

[1] Busy-waiting in `getNextThread` would prevent a thread from finishing blocking on an MVar or concurrent channel, which means it is not in a position to notice when an interrupt handler has actually supplied the data it was waiting for.

how to ensure that the idle thread contains a safe point so interrupts can be delivered. Tracking down priority inversions is still difficult.

The lack of debugging tools is definitely a drawback. Debugging an operating system is challenging, but tools exist for popular systems such as Linux. Debugging Haskell code has traditionally been tricky as well; combine the two, and it is really a challenge. Even the venerable print statement can fail to tell the whole story—coordinating debugging print statements in RTS code with those in concurrency library code and finally with House's console output can be a challenge. Also, certain information—such as a thread's ID or priority—now only has meaning in Haskell, so print statements from the RTS cannot access this information.

Laziness is another drawback of systems programming in Haskell. As mentioned in section 2.2, it is hard to judge how long it may take to evaluate a section of code—almost anything could be an expensive thunk. In particular, interrupt handlers usually run in the context of whatever thread was running, stealing a little bit of time, but must finish quickly. This is hard to guarantee in a lazy language. Furthermore, laziness essentially rules out locking as a viable synchronization mechanism, essentially forcing the decision to use an optimistic synchronization system such as transactional memory. Even then, many issues with blackholes arise—as noted above—none of which would occur in a strict language.

While not limited to Haskell, Lightweight Concurrency's approach of moving runtime system functionality into the language itself presents a few software engineering issues. Modularity issues come up frequently—it is very easy to get circular dependencies. For example, PTM needs to be able to catch exceptions which can arise if a transaction encounters an inconsistent state [8], and potentially re-throw them. Instinctively, one might try to import `Control.Exception` which defines `catch` and `throw`.

Unfortunately, it also defines `killThread` and `throwTo`, which are now based on PTM!

Careful layering can help avoid this. In Lighthouse, the Lightweight Concurrency modules (`LwConc.*`) import from `GHC.*`, but not `Control.*`. This allows `Control` modules to freely import from `LwConc`. In the above example, the `LwConc.PTM` module imports `GHC.Exception` while `Control.Exception` imports both `GHC.Exception` and `LwConc.PTM`.

Package dependencies can also be an issue. The concurrency library depends on many data structure modules, such as `Data.Sequence` (basic queues), `Data.Heap` (priority queues), `Data.List` (extra list functions), and `Data.IORef` (mutable state). Since the concurrency library is in the `base` package, these modules must be included in `base` as well, rather than in the `containers` and `heap` packages. This can be tricky when modules have many dependencies, such as `System.Random`.

6.2 Related Work

House is not alone; there are a variety of other operating systems based on functional languages. "Hello" [4] is an operating system based on Standard ML of New Jersey (SML/NJ) and the Linux kernel. ML/OS [3] is a port of SML/NJ to run on OSKit [3], a framework designed to aid in developing experimental operating systems; PLT Scheme [31] also runs atop OSKit. A more recent project, Kinetic [32], also aims to create a Haskell-based operating system. Shivers [24] has done much research on the interaction between hardware and modern languages, including key topics such as how to handle interrupts in the presence of automatic memory management.

STING [9, 10], a concurrent dialect of Scheme, is similar to Li et al.'s Lightweight Concurrency. It provides threads and virtual processors as first-class Scheme objects, and allows policy and scheduling to be implemented in Scheme. While a strict lan-

guage, STING includes the notion of *futures*, or threads which evaluate an expression and return the result. Threads can demand the value of a future, and will block if the value is not yet determined—somewhat similar to Haskell's lazy thunks and blackholing. As an optimization, STING implements *thread stealing*, a mechanism that allows the demanding thread to inline ("steal") the future's expression if work is not currently being evaluated on a virtual processor. Section 3.5.4 of this work proposes a similar mechanism for handling concurrent thunk evaluation in Haskell.

Morrisett and Tolmach [21] extended SML/NJ to support multiprocessors by combining SML's existing first-class continuations with new virtual processors ("procs") and mutexes ("locks") which can be manipulated from within SML. The Manticore Project [2] is a modern effort to provide lightweight concurrency in SML/NJ; Manticore offers reusable continuations and uses compare and swap as the basic synchronization mechanism. Unlike Haskell, SML/NJ is also a strict language.

Eager Haskell [17] is an implementation of Haskell that defaults to a strict evaluation strategy, but falls back to lazy evaluation after exceeding certain resource bounds. This strategy allows it to maintain Haskell's clean equational semantics yet bound the space leaks that come with purely lazy evaluation. However, it still uses blackholing as part of lazy evaluation, so Eager Haskell would not solve Lightweight Concurrency's blackholing issues. Unfortunately, support for Concurrent Haskell is not discussed.

6.3 Future Work

Lighthouse is just the beginning of what could be done with House and Lightweight Concurrency. Unlike most operating systems, House currently has no notion of *process*—only individual threads. One way to define a process is as a group of threads, potentially with certain resource limits. With processes in place, the hierarchical scheduler

model proposed by prior work on Lightweight Concurrency [16] becomes very intriguing; it could be a great way to not only schedule processes, but also allow processes great flexibility in how their threads are scheduled. Furthermore, support for processes calls the asynchronous exception mechanism into question. Users of the system would like to be able to interrupt or kill a whole application; they do not care about individual threads. Some form of process-level signaling/communication mechanism is necessary. At the same time, threads within a process will likely want to communicate as well. It is not yet clear whether either interface should look like Concurrent Haskell's asynchronous exceptions.

Currently, most of House is built atop concurrent channels, which in turn are built atop MVars. Unfortunately, this means much of the system is built atop a mechanism that inherently has no regard for thread priority. One idea is to convert more of House to use PTM (or STM) directly; in some cases, it may be more convenient, and may offer better performance and less priority inversion.

STM also remains unimplemented. While it could likely be implemented as a layer atop PTM, implementing the ability to block until one of the transactional variables changes is challenging. One obvious way to implement STM's TVar is as PTM PVar with a list of blocked threads to be awakened when the variable changes. The tricky part is that a thread might be waiting on any number of variables, yet its continuation can only be activated once. In effect, the thread must be atomically removed from all TVar queues when it is awakened; otherwise, future TVar changes may attempt to activate a dead continuation. This is not straightforward, especially if one tries to implement a solution that works in both uniprocessor and multiprocessor settings.

Another idea for future work is to build more exotic concurrency libraries atop the Lightweight Concurrency framework. Lighthouse, along with prior work, implement an

interface very similar to Concurrent Haskell. While this is a good thing to implement, it is very traditional; Lightweight Concurrency is intended to support very different concurrency libraries atop the substrate. One example of a unique concurrency library might be an embedding of the Orc programming language [14] in Haskell, built directly atop the substrate. This would further demonstrate the viability of the Lightweight Concurrency approach.

References

[1] Iavor S. Diatchki, Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. Writing systems software in a functional language: an experience report. In *PLOS '07: Proceedings of the 4th workshop on Programming languages and operating systems*, pages 1–5. ACM, 2007.

[2] Matthew Fluet, Mike Rainey, and John Reppy. A scheduling framework for general-purpose parallel languages. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 241–252. ACM, 2008.

[3] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: a substrate for kernel and language research. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 38–51. ACM, 1997.

[4] G. Fu. Design and implementation of an operating system in Standard ML. Master's thesis, University of Hawaii, 1999.

[5] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in haskell. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 116–128. ACM, 2005.

[6] Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a shared-memory multiprocessor. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, Haskell '05, pages 49–61. ACM, 2005.

[7] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 48–60. ACM, 2005.

[8] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, ISCA '93, pages 289–300. ACM, 1993.

[9] Suresh Jagannathan and Jim Philbin. A customizable substrate for concurrent languages. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 55–81. ACM, 1992.

[10] Suresh Jagannathan and Jim Philbin. A foundation for an efficient multi-threaded scheme system. In *Proceedings of the 1992 ACM conference on LISP and functional programming*, LFP '92, pages 345–357, New York, NY, USA, 1992. ACM.

[11] Richard Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–79, 1992.

[12] Simon Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine Version 2.5. *Journal of Functional Programming*, 2:127–202, 1992.

[13] S.P. Jones, N. Ramsey, and F. Reig. C--: a portable assembly language that supports garbage collection. *Lecture Notes in Computer Science*, 1702:1–28, 1999.

[14] David Kitchin, Adrian Quark, William Cook, and Jayadev Misra. The Orc programming language. *Formal Techniques for Distributed Systems*, 5522:1–25, 2009.

[15] Peng Li. *Programmable concurrency in a pure and lazy language*. PhD thesis, University of Pennsylvania, 2008.

[16] Peng Li, Simon Marlow, Simon Peyton Jones, and Andrew Tolmach. Lightweight concurrency primitives for GHC. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, Haskell '07, pages 107–118. ACM, 2007.

[17] Jan-Willem Maessen. Eager Haskell: resource-bounded execution yields efficient iteration. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 38–50. ACM, 2002.

[18] Simon Marlow and Simon Peyton Jones. The new GHC/Hugs runtime system. `http://research.microsoft.com/en-us/um/people/simonpj/papers/new-rts.ps.gz`, 1998.

[19] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous exceptions in Haskell. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 274–285. ACM, 2001.

[20] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Computer Science Department, Columbia University, June 1991.

[21] J. Gregory Morrisett and Andrew Tolmach. Procs and locks: a portable multiprocessing platform for standard ML of New Jersey. In *Proceedings of the fourth*

ACM SIGPLAN symposium on Principles and practice of parallel programming, PPOPP '93, pages 198–207. ACM, 1993.

[22] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 295–308. ACM, 1996.

[23] Alastair Reid. Putting the spine back in the Spineless Tagless G-Machine: An implementation of resumable black-holes. *Lecture Notes in Computer Science*, 1595:186–199, 1999.

[24] Olin Shivers, James W. Clark, and Roland McGrath. Atomic heap transactions and fine-grain interrupts. In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, ICFP '99, pages 48–59. ACM, 1999.

[25] Mitchell Wand. Continuation-based multiprocessing. In *Proceedings of the 1980 LISP Conference*, pages 19–28, August 1980.

[26] GHC Wiki: Capabilities and Scheduling. `http://hackage.haskell.org/trac/ghc/wiki/CapabilitiesAndScheduling`.

[27] Haskell Hierarchical Libraries: Documentation for the Control.Exception module. `http://haskell.org/ghc/docs/6.8.2/html/libraries/base/Control-Exception.html`.

[28] Haskell Hierarchical Libraries: Documentation for the System.Mem.Weak module. `http://haskell.org/ghc/docs/6.8.2/html/libraries/base/System-Mem-Weak.html`.

[29] House website. `http://programatica.cs.pdx.edu/House/`.

[30] Lighthouse source code. http://web.cecs.pdx.edu/~kennyg/house/.

[31] PLT Scheme. http://www.plt-scheme.org/.

[32] The Kinetic Operating System. http://www.ninj4.net/kinetic/.

[33] The Programatica Project. http://programatica.cs.pdx.edu/.