On Designing Lightweight Threads for Substrate Software

Matthew Haines
Department of Computer Science
University of Wyoming

On Designing Lightweight Threads for Substrate Software

Matthew Haines*

Department of Computer Science
University of Wyoming
haines@cs.uwyo.edu

Abstract

Existing user-level thread packages employ a "black box" design approach, where the implementation of the threads is hidden from the user. While this approach is often sufficient for application-level programmers, it hides critical *design decisions* that system-level programmers must be able to change in order to provide efficient service for high-level systems. By applying the principles of Open Implementation Analysis and Design, we construct a new user-level threads package that supports common thread abstractions and a well-defined meta-interface for altering the behavior of these abstractions. As a result, system-level programmers will have the advantages of using high-level thread abstractions without having to sacrifice performance, flexibility, or portability.

1 Introduction

Lightweight threads are useful for a variety of purposes. An application-level programmer will typically use threads to facilitate asynchronous scheduling for a number of related tasks. For example, consider an event-driven application, such as Xlib [25], where lightweight threads are used to schedule tasks for execution based on an external event. In this context, threads free the programmer from the details of dynamic scheduling. Fine-grain control over the behavior of a thread is typically not needed. There is no shortage of lightweight thread packages for application-level programmers, and a short list of such systems would likely include pthreads [22] (the POSIX interface for lightweight threads [16]), Solaris threads [26], fastthreads [2], and cthreads [23].

Lightweight threads are also useful for supporting independent tasks generated by parallel or concurrent programming languages. In this context, system-level programmers use lightweight threads as a major building-block of a *multithreaded runtime system*. For example, each of the following languages is supported by a multithreaded runtime system: CC++ [12], Fortran M [12], Opus [15], Orca [7], PC++ [8], Sisal [13], Split-C [11], and SR [3]. However, *none* of these multithreaded runtime systems employs a single thread package for all platform implementations, and few use *any* of the lightweight thread packages listed in the preceding paragraph. While this may be surprising, there are several reasons why multithreaded runtime system designers shy away from standard lightweight thread packages, including:

1. *Lack of flexibility.* Existing thread packages are implemented as black boxes, so it is almost always impossible to change the detailed behavior of threads, mutexes, run lists, etc. However, most multithreaded runtime systems require explicit control over scheduling decisions and the interaction of threads with a communication substrate. For example, the Panda runtime system [7], which supports the Orca programming language [5], requires preemptive scheduling of threads with priorities, and the ability to turn preemption off and explicitly poll for incoming messages when there are no active threads. On the other hand, the runtime system that supports the SR programming language [3] assumes non-preemptive scheduling, in which the scheduler is free to select the next thread to the executed from a list of runnable threads. Both languages support communication between multiple processors, and this communication directly affects thread scheduling.

2. *Lack of performance.* Existing thread packages are geared towards supporting application-level programmers, who typically require threads to behave as normal Unix processes would. However supporting this behavior, including proper

*Supported in part by the National Aeronautics and Space Administration under NASA Contract No. NASA-19480, while the author was in residence at ICASE, NASA Langley Research Center, and by the University of Wyoming Faculty Grant-in-aid and International Travel Grant Programs.

signal handling, adds a good deal of overhead to the thread operations. In contrast, most multithreaded runtime systems want bare-bones threads that are very fast, and to which they will add the complexities they require. The key here is that the runtime system designer, rather than the thread package designer, is in control of the tradeoffs between functionality and performance.

3. *Lack of portability.* Multithreaded runtime systems must execute on a wide variety of hardware and operating system platforms, which is extremely problematic for most lightweight thread packages.

4. *Lack of information.* Multithreaded runtime system designers often require tracing information for debugging and statistics information for tuning. Existing thread packages provide little or no support for obtaining this sort of information about the execution of the threads.

The central problem with using existing thread packages to support multithreaded runtime systems is that most existing thread packages are designed as "black-boxes," providing only a very limited number of ways in which behavior can be modified. This is typically limited to altering the default size for a thread stack and choosing between a small, fixed-number of pre-implemented thread scheduling policies. This is almost always too restrictive for system-level programmers, and so most end up "rolling their own" thread packages.

Though "black box" abstractions are effective for constructing complex systems because they hide the details of an implementation, they don't always work because "there are times when the implementation strategy for a module cannot be determined before knowing how the module will be used in a particular system" [18]. As we've seen, this is particularly true for multithreaded runtime systems employing lightweight threads, where many of the implementation decisions are to be made by the runtime system designer, not the thread package designer. Recently, the object-oriented research community has been addressing the issue of improved design methodologies for substrate (system-level) software [9, 18, 19, 20, 27]. From this research, a new design methodology for substrate software has emerged, called Open Implementation [18, 19]. The basic idea behind this new design methodology is to open the proverbial black box using a well-defined interface, called a *meta-interface*, that describes how the abstractions provided in the user-interface are to behave.

In this paper we provide an Open Implementation analysis and design of lightweight, user-level threads. As a proof-of-concept, we have implemented this design to create a thread library called **OpenThreads**. The goal of this research is to produce a lightweight threads library that provides both a simple user-level interface and a robust meta-level interface for altering the behavior of the abstractions, so that a single threads package can be efficient, flexible, and portable. In addition, we extend the Open Implementation design methodology to address portability concerns by defining a system-level interface that clearly defines underlying dependencies.

The remainder of this paper is divided into five sections. Section 2 provides background on threads and Open Implementation analysis and design. Section 3 provides the Open Implementation analysis and design for OpenThreads, and discusses the three interfaces interfaces. Section 4 provides a discussion of our design relating to performance and open issues. Section 5 outlines related research projects, and we conclude in Section 6.

2 Background

In this section we provide background information for readers unfamiliar with either lightweight, user-level threads or Open Implementation Analysis and Design.

2.1 User-level Threads

User-level threads provide the ability for a programmer to create and control multiple, independent units of execution entirely outside of the operating system kernel (i.e., in user-space). The *state* of these threads is often minimal, consisting usually of an execution stack allocated in heap space and the set of CPU registers, and so these threads are often termed *lightweight*.

Since the OS kernel controls addressing and scheduling for the CPU, user-level threads must be multiplexed atop one or more kernel-level entities, such as Unix processes [4], Mach kernel threads [1], or a Sun Lightweight Processes (LWP) [24]. This "multiplexing" is commonly referred to as *scheduling* of the user-level threads. The kernel-entity (hereafter referred to as a "process") also provides a common address space that is shared by all threads multiplexed onto that process, and synchronization primitives are provided to keep the memory consistent. It is also possible for threads to have some amount of *thread-specific data* by storing pointers to this data on each thread stack.

Scheduling policies for lightweight threads can be broadly classified either as *non-preemptive*, in which a thread executes until completion or until it decides to willingly yield the processor, or as *preemptive*, in which a thread can be interrupted at an arbitrary point during its execution so that some other thread may execute. Orthogonal to the issue of preemption, thread scheduling can incorporate a wide range of capabilities, including priorities, hand-off scheduling, and arbitrary run list structures (such as trees). Clearly, there are many design choices to be made for thread scheduling, and many runtime system designers will want to switch between various policies depending on the state of the system. If a user-level thread package is not useful to a system-level programmer, lack of control over scheduling is commonly at the root of the cause.

2.2 Open Implementation Analysis and Design

In [18], Kiczales introduces a new approach for the design of substrate software called Open Implementation, and in this section we summarize these ideas. The reader is encouraged to examine [18] and [20] for more details on this design philosophy.

We have already stated that black-box abstractions do not always work because "there are times when the best implementation strategy for a module cannot be determined without knowing how the module will be used in a particular situation" [18]. So, what happens when a programmer using a black-box abstraction is confronted with conflict between how the abstraction is implemented and how the abstraction *should be* implemented? Since the current implementation is hidden within the internal portion of the black-box (as depicted in Figure 1), the programmer must "code around" the problem. This results in either *hematomas of duplication* or *coding between the lines*.

A hematoma of duplication occurs when a system-level programmer writes his own threads package, ensuring that his performance and flexibility demands are met. In addition to increasing the size and complexity of the resulting system, hematomas can result in convoluted code above the runtime system, where the black-box thread package may still be used.

Coding between the lines occurs when a programmer writes code in a particularly contorted way to get the desired performance or functionality. For example, consider a multithreaded runtime system designer who plans to create and destroy many threads. If the underlying thread package does not provide a way to cache the thread control blocks and thread stacks, the programmer might have to create server threads that never really dies so that resource management overheads are minimized.

These examples demonstrate that black-box designs often hide too much of the implementation for substrate software. While some of the implementation decisions are details that can be (and should be) hidden without problem, others are crucial to writing efficient software and should be exposed in a controlled manner. These crucial design decisions are called *dilemmas*.

The Open Implementation design depicted in Figure 1 provides a mechanism for exposing dilemmas to the programmer so that these crucial design decisions can be made on a per-application basis. This mechanism is represented as a new interface, called the *meta interface*, that is presented to the application programmer for altering the behavior of the abstractions presented in the user interface. The meta interface provides a clean and controlled mechanism for customizing the implementation of substrate software and is the key to the Open Implementation design philosophy. Section 3.2.2 details the meta interface for OpenThreads.

3 Design

In this section we outline the design of OpenThreads. In Section 3.1 we itemize the issues are faced when designing a lightweight thread package, and in Section 3.2 we explain how these issues are exposed to the user in terms of *interfaces*.

3.1 Dilemmas

We begin this section with an examination of the dilemmas that occur in the design of a thread package. As we stated in Section 2.2, the key difference between a black box design and an open implementation design is the level of control over these dilemmas.

3.1.1 Thread States

The lifetime of a thread is marked by a series of transformations between different *states*. A common set of thread states includes: being created, being placed onto an active run list, being selected (scheduled) for execution, being blocked on a mutex or condition variable, and being terminated. The transitions that take a thread from one state to the next can differ from one thread package to the next. For example, a blocked thread can either be resumed as an active thread or as a runnable thread. Figure 2 depicts these states transitions.

Besides the dilemma of where to place a thread that becomes unblocked, there are dilemmas associated with the transitions between the states. For example, what should be done when a thread is created or terminated? How about when a thread is in transition between the active and runnable states? In existing thread systems, these transitions are hidden from the user. This makes it impossible, for example, to trace the execution of a thread, since the user would need control over several transitions.

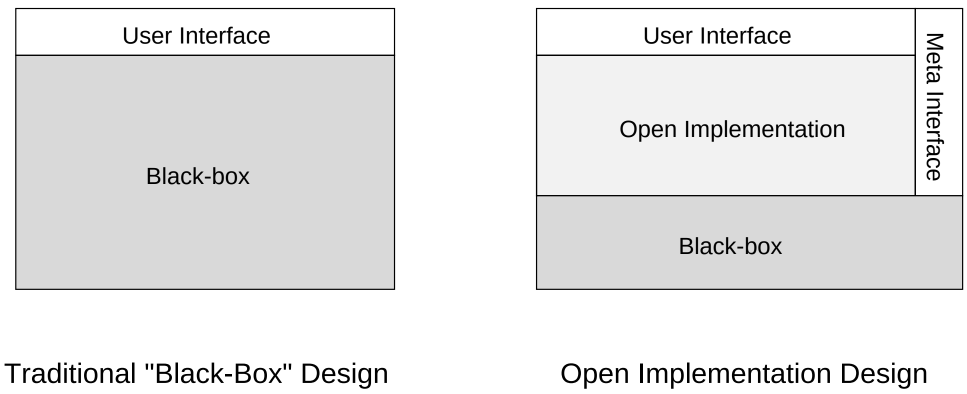

Figure 1: "Black-box" and "Open Implementation" designs

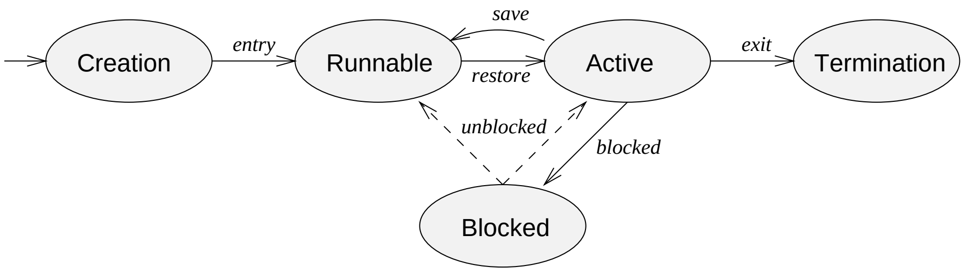

Figure 2: Thread states

Our approach to these dilemmas is to define a set of events that occur (perhaps repeatedly) during the life of a thread, and provide a mechanism for specifying user-defined actions to be performed whenever a thread encounters one of these events. As depicted in Figure 2, the following events are defined for a thread: **entry** - when a thread first begins execution; **exit** - when a thread is about to be terminated; **save** - when a thread moves from an active state to a runnable state; **restore**, when a thread moves from a runnable state to an active state; **blocked**, when a thread moves from an active state to a blocked state; and **unblocked**, when a thread moves from a blocked state to either an active or runnable state.

In addition to these thread-specific states, there are system states of either executing some runnable thread or being idle because all remaining threads are blocked. The idle system state is often achieved when all threads are waiting for some external event, such as a message or interrupt, to occur. The transition of the system from the active state to the idle state is also important, and so we define three more events to cover this situation: **idle begin** - when all threads in the system become blocked; **idle spin** - the duration of being idle; and **idle end** - when some thread becomes runnable again.

3.1.2 Thread Lists and Scheduling

A *thread list* is a data structure used to hold a collection of threads. The list may be used to hold threads that are ready to execute, called a *run list*, or may be used to hold threads in some other state, such as blocked on a mutex or condition variable. Since all threads must be on some thread list, state transitions typically involve moving a thread from one list to another. The scheduling policy for threads is therefore determined by the structure of the thread lists and the implementation of the operations that remove a thread from a list (`get`) and place a thread onto a list (`put`). For example, simple FIFO scheduling is achieved by using a FIFO queue for a run list, whereas priority scheduling might involve a tree of FIFO queues, where each leaf of the tree represents a queue of threads with the same priority.

Existing thread packages hide the concept of thread lists and provide abstract notions of scheduling for the system, such as FIFO or Round-Robin. However, this black-box approach prevents system-level programmers from gaining control over the most fundamental part of a thread package. For example, what if a tree structure would be most efficient for a run list, or what if multiple run lists are desired? What if mutex variables are to have thread lists which are scheduled differently from the run list, or from condition variable blocked lists? These dilemmas about the nature and behavior of thread lists need to be exposed to the system-level programmer.

Our approach to these dilemmas is to make thread lists explicit, first-order entities in the system, and allow the user to define the `get` and `put` primitives for each list. Thread lists (or queues) are then explicitly associated with mutexes, condition variables, and runnable threads. OpenThreads provides mechanisms for specifying which list a thread is to be placed onto when yielding (`ot_thread_yield_onto`) and when initializing a thread (as an argument to `ot_thread_init`). Note that although there are multiple thread lists, only one at a time may be designated as the official "run list", which is where ready threads are selected for execution. The run list is specified as an argument to the `ot_begin_mt` function.

3.1.3 Context Switching Modes

Every thread maintains *state information* that defines the thread. Minimally, this set includes the stack pointer, the instruction pointer, and the contents of the live registers. During a context switch between two threads, the state information of the old thread is saved and the state information of the new thread is restored.

Existing thread packages treat all context switches the same with respect to the state of a thread. However, this should not always be the case. Some threads, for example, may only use the integer registers, so saving and restoring the floating point registers at each context switch is a waste of precious cycles. But since the thread package designer doesn't know if threads will be using the floating point registers or not, a conservative decision is made and all registers are saved.

Our approach to these dilemmas is to allow for the context of a thread to be defined as either involving all registers, only the integer registers, or no registers. A more flexible approach might allow for a user-defined function that saves the exact thread state needed, but such a function would almost certainly be platform-dependent, thus violating portability.

3.1.4 Stack Management

Often, the most time-consuming portion of creating a new thread is allocating and aligning the thread stack. While most thread packages offer support to change the size of a thread's stack, few allow the programmer to specify the stack allocation policy. Lack of control over this dilemma is another common reason why multithreaded runtime systems abandon common thread packages. For example, the programmer may want to cache the stacks because threads are created and destroyed rapidly, but there are never more than a small number of threads alive at any one time. Another example is when the programmer wishes to enable checks on stack overflow, or to resize the stack at runtime to enable stack growth.

Our approach to this dilemma is to allow the user to specify stack allocation and release policies.

3.1.5 Timing and Profiling

Existing thread systems offer little or no support for thread timing and profiling. While most application-level programmers may not care about how many times a given thread is switched or what the total execution time for a thread is, most system-level programmers do care about these measures. However, because the state transitions that define these measures are hidden from the user, it is usually impossible to gather these statistics even if the user wants to.

Our solution to this dilemma is to allow the user to take advantage of our exposed thread events and install monitoring code that will be executed whenever a thread reaches one of these events. For example, to count the number of times that a thread is switched from runnable to active, we can simply install the following function to be invoked whenever a thread triggers the *restore* event:

```
void bump () {
    ctxswCounter++;
}
```

3.2 Interfaces

We now discuss a mechanism for making these dilemmas available to the user in a clean and well-defined manner. Recall that in Figure 1, there are two interfaces presented to the user rather than just one. The first is the *user-level* interface, which defines the abstractions that are supported. The second is the *meta-level* interface, which defines how to change the behavior of the abstractions supported in the user-level interface. Thus, the meta-level interface represents the realization of our design dilemmas. We call this a *meta*-interface because it's an interface that describes how another interface (the user-interface) should behave.

3.2.1 The User-level Interface

The user-interface for OpenThreads provides a simple and clean mechanism for creating threads, mutex variables, condition variables, and thread lists. Noticeably absent are the plethora of routines that define the API for packages like pthreads, which is possible because the user-interface for OpenThreads is not concerned with modifying behavior. As with any thread package, OpenThreads allows the programmer to create new threads, yield, exit, wait for a mutex, and

```
extern void ot_init (int *argc, char *argv[], char *pkg_prefix);
extern void ot_start (ot_queue_t *runq);
extern void ot_end (void);

typedef void (ot_userf_t)(void *p0);
extern void ot_thread_init (ot_thread_t *thread, ot_userf_t *start,
                void *args, unsigned tid, ot_queue_t *runq);
extern void ot_thread_yield (void);
extern void ot_thread_yield_onto (ot_queue_t *destq);
extern void ot_thread_exit (void);
extern unsigned ot_thread_id (void);
extern ot_thread_t *ot_current_thread (void);
extern void ot_thread_setspecific (ot_thread_t *thread, void *ptr,
                void (*cleanup)(void*));
extern void *ot_thread_getspecific (ot_thread_t *thread);

extern void ot_queue_init (ot_queue_t *q);

extern void ot_mutex_init (ot_mutex_t *m, ot_queue_t *blockq);
extern void ot_mutex_lock (ot_mutex_t *m);
extern int ot_mutex_trylock (ot_mutex_t *m);
extern void ot_mutex_unlock (ot_mutex_t *m);

extern void ot_cond_init (ot_cv_t *cv, ot_queue_t *blockq,
                ot_mutex_t *m);
extern void ot_cond_wait (ot_cv_t *cv);
extern void ot_cond_bcast (ot_cv_t *cv);
```

Figure 3: OpenThreads user interface

block on a condition variable. These functions are detailed in Figure 3.

There are a few calls in this interface that bear special attention. First, the thread initialization function used to create a thread, `ot_thread_init`, takes a thread list (queue) as an argument, and places the newly created thread on that list. This gives the programmer explicit control over managing run lists. Second, the `ot_thread_yield_onto` routine is used to specify a destination thread list upon which the existing thread will be put. Again, this gives the programmer explicit control over thread list management. Third, the `ot_thread_setspecific` and `ot_thread_getspecific` calls are used to assign and retrieve a single generic pointer contained within the thread control block. This single pointer is meant to satisfy the needs of multithreaded runtime system designers, who all employ the concept of a *task* within their systems. Each task contains an instance of an OpenThread, which will perform the actual context switching between the tasks. However, since the current thread can only be defined in terms of an OpenThread, finding the current task requires a pointer from the OpenThread control block back to the surrounding task. Any additional thread-specific data can be multiplexed atop this single pointer without all users having to pay the cost in time and space to maintain an arbitrarily-long list of thread-specific pointers.

One problem with many thread packages is the vagueness with which multithreaded execution begins and ends, and what happens to the original thread of control within the process. OpenThreads makes these points of control explicit by creating two functions that mark the beginning end ending of multithreaded execution: `ot_begin_mt` and `ot_end_mt`, respectively. In between these calls the original process thread of control, now called the *process thread*, is allowed to execute any other code it desires, and is treated just like any other thread in the system. It can, for example, be blocked on a condition variable or be re-scheduled for execution on the run list. When the `ot_end_mt` call returns, the system is single-threaded again and another round of multithreaded execution may be initiated if desired. There are also functions for initializing the OpenThreads package (`ot_init`) and cleaning up after the package (`ot_done`). Sample code for a process initiating multithreaded execution is given in Figure 4.

3.2.2 The Meta-level Interface

The OpenThreads meta interface (Figure 5) provides the hooks needed to customize the design dilemmas listed in Section 3.1. In most cases, these decisions are set up as events that trigger user-specified actions, or *callback functions*, to occur. For example, the `otm_push_callback` routine allows the user to specify a callback function to be invoked whenever the specified event is triggered. As mentioned in Section 3.1.1, events can be either *thread-specific*, which occurs whenever any thread enters a specific state, or *global*, which occurs when the system itself enters a new state. The valid thread-specific events are thread entry, thread exit, thread save, thread restore, thread blocking, and thread unblocking. The valid global events are system idle begin, system idle spin, and system idle end. Callback functions for each event are maintained in a stack, so that multiple functions can be associated with each event. For example, this would allow a set of tracing functions to be installed atop a set of timing functions. All function for an event are called in stack order, and the `otm_pop_callback` routine provides a way of clearing or rearranging the callback stack of a thread at any time.

The `otm_install_queue` function allows the user to provide implementations for the `get` and `put` functions of a thread list, as well as to define how large the link components of the thread list need to be. OpenThreads will invoke the `get` and `put` functions of a list whenever a scheduling decision needs to be made. This allows for multiple, user-defined thread lists to be used at the same time within OpenThreads, giving the user total control over scheduling. The interface allows different implementations to be associated with different thread lists at the same time.

The `otm_define_switch` routine allows the user to define the thread switching mode for a given thread (or for all threads). The valid switching modes are *all*, *integer*, or *none*, referring to the registers to be saved.

3.2.3 The System-level Interface

One of the key elements in the design of a multithreaded runtime system is *portability*. Since parallel and concurrent languages execute in a wide variety of environments, their runtime systems must support a wide-range of platforms.

To enable portability, we added a *system-level* interface (see Figure 6) to the traditional Open Implementation design, resulting in the overall design of OpenThreads with three interfaces, as depicted in Figure 7. The system interface provides a single place for mapping all dependencies that cannot be satisfied from within the OpenThreads implementation.

These routines are then mapped (usually with symbolic constants) onto platform-specific routines that provide the necessary functionality. Therefore, a successful port of OpenThreads requires modification of exactly one header file in a clean and well-defined way. Note that the routines at this level are decidedly low-level, so that additional overheads are not incurred. The thread initialization and context switch-

```
void main (int argc, char *argv[])
{
    ot_thread_t threads[NT];
    ot_queue_t runq;
    ...
    ot_init (&argc, argv, "ot_");
    ot_queue_init (&runq);
    for (int i = 0; i < NT; i++)
        ot_thread_init (&threads[i], entry_func, arg, &runq);
    ...
    ot_begin_mt (&runq);
    /* == begin multithreaded execution == */
    ...
    ot_end_mt ();
    /* == end multithreaded execution == */
    ...
    ot_done ();
}
```

Figure 4: Sample code for initiating multithreaded execution

```
extern void otm_init (void);
extern void otm_install_stackalloc (otm_sallocf_t *salloc,
                otm_sfreef_t *sfree);
extern void otm_define_switch (otm_switch_mode_t, ot_thread_t *t);
extern void otm_push_callback (int cbid, otm_callback_t *cbfunc);
extern otm_callback_t *otm_pop_callback (int cbid);
extern void otm_install_queue (ot_queue_t *q, unsigned qlink_size,
                unsigned qimp_size, otm_qinitf_t *init,
                otm_qgetf_t *get, otm_qputf_t *put);
```

Figure 5: OpenThreads meta interface

```
extern void *ots_stack_align (void *stack);
extern ots_stack_t *ots_stack_pointer (void *storage, int size);
extern ots_stack *ots_stack_init (ots_stack_t *sp, ots_userf_t *userf,
                void *userarg, ots_inif_t *initf, void *initarg);
extern void *ots_switch_all (ots_helperf_t *helper, void *arg1,
                void *arg2, ots_stack_t *new);
extern void *ots_lock_init (ots_lock_t lock);
extern int ots_lock_acquire (ots_lock_t lock);
extern void void ots_lock_release (ots_lock_t lock);
```

Figure 6: OpenThreads system interface

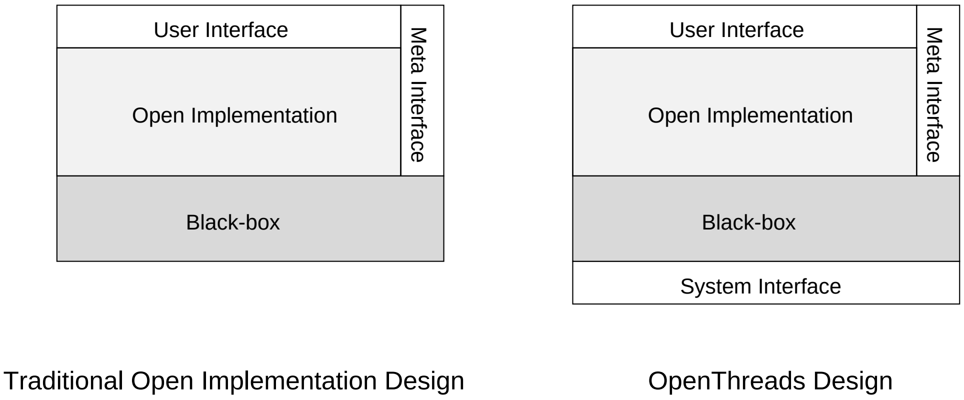

Figure 7: OpenThreads design

ing routines intentionally mimic the QuickThreads [17] macros, which are an excellent example of how very-low-level thread support can be provided in a machine-independent manner.

4 Discussion

In this section we discuss our design in terms of our original design goals and in terms of open issues that have yet to be addressed or resolved.

4.1 Design Goals

The design of OpenThreads is based on three essential goals: flexibility, efficiency, and portability. Our hypothesis is that substrate software requires success for each goal, and that existing lightweight thread packages fail in one or more of these goals. For example, one may argue that pthreads [16] is efficient and provides portability because its the "standard," but it falls short in providing the flexibility demanded by most system-level programmers. For example, using pthreads it is impossible to trace thread execution when the scheduling policy is round-robin (preemptive).

We now examine our Open Implementation of lightweight threads with respect to these design goals:

1. **Flexibility**.

   The flexibility of our design is manifest by the meta-level interface, and the ability of a programmer to provide her own solutions to the design dilemmas outlined in Section 3.1. As a concrete example of its flexibility, we have adapted the Panda multithreaded runtime system [7] to use OpenThreads for implementing its tasks.

2. **Efficiency**.

   With regards to performance, we can report on the performance of OpenThreads in comparison POSIX Pthreads and the QuickThreads package upon which OpenThreads is built. The comparison with Pthreads demonstrates the efficiency of OpenThreads for doing basic multithreading. The comparison with QuickThreads shows how much overhead we've added to the underlying low-level switching routines. As a demonstration of portability, we present these tests on three different processor architectures: the MIPS R4400, the Sun SPARC, and the DEC ALPHA. The measurements are given in Table1. All numbers were gathered using averages for multiple runs, with an initial untimed run used to reduce cache miss effects. For context switches, OpenThreads saved all registers.

   A second set of experiments evaluates the flexibility of OpenThreads to support the Panda runtime system. The hand-crafted Panda thread package was developed for the Amoeba operating distributed system and runs on 50 MHz SPARC processors with 32 Mbyte of memory and 4 Kbyte instruction and 2 Kbyte data caches (direct mapped). To achieve high performance the Panda scheduler is not built on top of Amoeba's kernel threads, but it was derived from a user-level thread package developed at MIT by Wallach and Kaashoek. These results are presented in Table 2.

   Given the extensive tuning performed on the Panda thread package and the overhead of OpenThreads, we expected the generic Panda threads implemented with OpenThreads (Panda-OT) to perform less than the hand-crafted Panda threads. The performance results presented in Table 2, however, show that Panda-OT has the fastest context switch time on Amoeba: 40.2 μs (Panda) versus 35.4 μs (Panda-OT). Examination of the low-level context switch code provided by QuickThreads (used in OpenThreads) revealed that it contains a SPARC specific opti-

	MIPS R4400		SPARC		ALPHA	
	ctxsw	create	ctxsw	create	ctxsw	create
QuickThreads STP	0.9	5.6	6.5	10.5	1.2	2.7
OpenThreads	1.3	7.0	7.2	13.5	2.3	3.1
Pthreads	23.8	17.5	28.5	160.3	—	—

Table 1: OpenThreads raw performance on various architectures; *ctxsw* is the time in microseconds for a context switch and *create* is the time in microseconds to create a single thread with default stacksize (usually 8Kb).

	Amoeba		Solaris	
	ctxsw	create	ctxsw	create
Panda	40.2	108.9	34.1	417.4
Panda-OT	35.4	262.7	7.4	62.0

Table 2: Performance of Panda thread packages on various platforms.

mization that saves one register-window underflow trap. Saving one trap to the Amoeba kernel accounts for approximately 7 μs. Since we did not want to change Panda's implementation for the sake of making this comparison, we have provided the empirical numbers instead. However, the same optimization could be applied to the Panda threads, which would negate the performance advantage of OpenThreads for this experiment. In the end, we see about a 6% overhead for OpenThreads, which is a small price to pay for the added flexibility and portability.

Unlike the context switch results, the results for thread creation show that Panda-OT performs poorly in comparison to native Panda; creating a thread with Panda-OT is more than twice as expensive as with native Panda. This large overhead, however, is not a consequence of the flexibility of OpenThreads, but of a slight difference in scheduling policy of both thread packages. Panda-OT implements true priority scheduling, and therefore takes a scheduling decision as soon as the new thread is created and placed on the run queue. This results in a context switch to the newly created thread, which initializes itself and then terminates immediately. Once the thread has exited, another context switch occurs to transfer control back to the main thread. The Panda scheduler, on the other hand, does not take a scheduling decision until the main thread voluntarily yields control. The difference in scheduling policy causes Panda-OT to incur two additional context switches, thus accounting for the performance difference.

Table 2 also contains performance numbers for Panda-OT on Solaris. In this case we compare Panda-OT to the original Panda implementation that uses native Solaris threads. The performance was measured on a 70 Mhz SPARCstation 4 with 64 Mb of memory running Solaris 2.4. Panda-OT performs much better than native Panda: Panda-OT switches between threads over four times as fast and creates new threads over six times as fast. Since we do not have the source code of Solaris threads available, it is difficult to determine the exact causes of this great difference. We believe, however, that part of the explanation is Solaris ability to support a mixture of kernel and user threads. This functionality adds considerably to the complexity of the threads system, and requires expensive precautions to guard against preemption at various levels.

3. **Portability**.

   Our system-level interface isolates all underlying dependencies in a single file that needs to be changed for each new platform. In some cases, there are no changes required at all. This is due to the fact that OpenThreads is currently mapping its low-level thread operations onto QuickThreads [17], which is already very portable. The Panda port runs the same code on both the Solaris and Amoeba platforms, and regular Orca programs were run to ensure the stability of the port.

   OpenThreads has been tested on the Sun SuperSPARC and UltraSPARC architectures under both SunOS 4.1 and Solaris 2.5, the MIPS R4400 architecture under IRIX 5.3, and the Alpha AXP architecture under DEC OSF-1. In addition, QuickThreads runs on the Intel 80x86, the Motorola 88000, the HP-PA, the KSR, and the VAX. As a result, OpenThreads can easily be ported to these architectures.

4.2 Open Issues

There are still a few open issues that remain in the design and implementation of OpenThreads.

1. Thread identification is the task of determining which thread, or more specifically, which thread control block is currently active at any given time. One way to do this is to reserve a global register to hold this pointer. This is the approach taken by Solaris threads [28], and has the advantage of being very fast and not requiring global memory. However, compiler and architecture support are required to reserve this register, and the loss of a register on RISC-based architectures is always cause for concern. Another approach is to keep the current thread pointer on a well-known offset in each thread stack [6]. This approach eliminates the need for both global memory and register space, but requires fancy stack alignments. Another approach, and the one currently employed by OpenThreads, is to use a global variable for storing the current thread pointer. This approach is the easiest to implement, does not require compiler support for extra registers, and does not require fancy stack manipulation. However, it does require per-processor global memory. A formal investigation regarding the best method for thread identification is still an open issue.

2. Kernel threads represent an opportunity to increase processor utilization in the face of blocking kernel calls, such as I/O. However, multiplexing user-level threads atop kernel threads requires a little finesse. We are in the process of revising OpenThreads to be safe in the presence of kernel threads. This requires protecting critical regions with kernel locks and identifying all global data as either shared among the kernel threads or private. Kernel thread private global data, such as the pointer to the current thread, needs to be stored as thread-specific data for each kernel thread. Powell et.al. [24] provide a discussion of this mapping for Solaris threads mapped onto LWPs. Another issue to be addressed in supporting kernel threads is the impact of kernel thread decisions on the meta-level interface. For example, some kernel threads might support features that allow for better optimization of the user threads, such as upcalls. To what degree should these decisions be supported in the meta-interface? Kernel threads also raise the specter of multiprocessor issues, which we have completely ignored to this point.

3. Multithreaded runtime systems require both thread and communication components, and both must work well together. We are in the process of examining the issues regarding the combination of threads with communication models, and plan to test the flexibility of our system for performing platform-independent optimizations using a combination of thread and communication modules. Other systems combining threads with communication primitives include Chant [14], Nexus [12], PVM-threads [21], and MPI-threads [10].

4. Signals. Most papers on lightweight threads include long and involved discussions about signal handling in the presence of threads. We will avoid this discussion for now, and simply state that OpenThreads currently exposes all threads to the same signal mask. We do, however, ensure that if a signal arrives while the system is in an atomic section, the signal handler is delayed until after the atomic section has been completed.

5. Debugging. Debugging multithreaded systems has always been a trying experience because there are almost no debuggers that recognize the threads. OpenThreads does provide critical support in this regard by allowing traces to be made for thread state transitions by installing print statements at the various thread-specific event points. However, more sophisticated debugging tools are clearly required.

5 Related Research

OpenThreads represents a novel approach to the design of user-level threads, in which the user is given the opportunity to change the behavior of high-level abstractions in a well-defined manner. Many thread packages, such as pthreads [16], support an extensive user-interface with some behavior-modifying commands intertwined (such as attribute specification for threads). However, these systems do not take a systematic approach to exposing the critical design dilemmas and, as a result, fall short in providing the flexibility required by most system-level programmers.

QuickThreads [17] is a thread-building toolkit that offers platform independent micro-instructions for managing thread stacks. QuickThreads is similar to assembly language programming in terms of flexibility, speed, and complexity. OpenThreads builds on the QuickThreads design philosophy of keeping things simple, and provides high-level abstractions whose behavior can be modified by the user in a well-defined manner.

The initial description of Open Implementation Analysis and Design [18] provided the motivation for much of this work. However, the initial description fails to talk about portability concerns. As a result, we extended the design to include a new system-level interface that unifies and defines all system dependencies.

6 Conclusions

It would seem that the last thing we need these days is another user-level thread package. From the standpoint of an application-level programmer I would have to agree. However, from the standpoint of a system-level programmer building multithreaded runtime systems, I would disagree. The evidence suggests that none of the current thread packages are being widely used by system-level programmers. In this paper we introduce the design of a user-level thread package for substrate software. The idea here is to identify all of the crucial design dilemmas that occur in building a thread package, and provide a clean and well-defined way for users to change these decisions. The result is a thread package with a simple user interface and a powerful meta interface for changing the behavior of the abstractions defined by the user interface. A system interface should also be used to isolate and define all underlying dependencies.

We have designed and built OpenThreads as a proof-of-concept for the ideas outlined in this paper, and are in the process of adapting several multithreaded runtime systems to use OpenThreads. We will report on the success of these attempts in the future.

7 Acknowledgments

Thanks to Koen Langendoen for helpful comments on a preliminary version of this paper, for his patience in describing the details of the Panda runtime system, and for helping me to debug OpenThreads in the process of porting Panda. It was a fun three days. Thanks also to Greg Benson for his description of the SR runtime system, and for his views on thread identification mechanisms. Finally, I thank Orran Krieger and the anonymous reviewers, whose detailed comments greatly helped to improve this paper.

References

[1] J. J. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young. Mach: A new kernel foundation for UNIX development, July 1986.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *ACM Symposium on Operating Systems Principles*, pages 95–109, 1991.

[3] Gregory R. Andrews and Ronald A Olsson. *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings, 1993. ISBN 0-8053-0088-0.

[4] Maurice J. Bach. *The Design of the UNIX Operating System*. Software Series. Prentice-Hall, 1986.

[5] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[6] Greg Benson. Information on thread identification using stacks. Personal communication, March 1996.

[7] Raoul Bhoedjang, Tim Rühl, Rutger Hofman, Koen Langendoen, Henri Bal, and Frans Kaashoek. Panda: A portable platform to support parallel programming languages. In *Symposium on Experiences with Distributed and Multiprocessor Systems IV*, pages 213–226, San Diego, CA, September 1993.

[8] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel systems. In *Supercomputing*, pages 588–597, Portland, OR, November 1993.

[9] Martin D. Carroll and Margaret A. Ellis. *Designing and Coding Reusable C++*. Addison Wesley, 1995.

[10] Aswini K. Chowdappa, Anthony Skjellum, and Nathan E. Doss. Thread-safe message passing with P4 and MPI. Technical Report TR-CS-941025, Computer Science Department and NSF Engineering Research Center, Mississippi State University, April 1994.

[11] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Supercomputing*, pages 262–273, Portland, OR, November 1993.

[12] Ian Foster, Carl Kesselman, Robert Olson, and Steven Tuecke. Nexus: An interoperability layer for parallel and distributed computer systems. Technical Report Version 1.3, Argonne National Labs, December 1993.

[13] Matthew Haines and Wim Bohm. Task management, virtual shared memory, and multithreading in a distributed memory implementation of Sisal. In *Parallel Architectures and Languages*, pages 12–23. Springer-Verlag Lecture Notes in Computer Science, Vol 694, 1993.

[14] Matthew Haines, David Cronk, and Piyush Mehrotra. On the design of Chant: A talking threads package. In *Proceedings of Supercomputing*, pages 350–359, Washington D.C., November 1994. ACM/IEEE.

[15] Matthew Haines, Bryan Hess, Piyush Mehrotra, John Van Rosendale, and Hans Zima. Runtime support for data parallel tasks. In *Proceedings of The Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 432–439, McLean VA, February 1995. IEEE.

[16] IEEE. *Standard for Threads Interface to POSIX*, 1996. P1003.1c.

[17] David Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington, 1993.

[18] Gregor Kiczales. Foil for the workshop on open implementation. In *Proceedings of the Workshop on Open Implementation*, 1994. Available at `http://www.xerox.com/PARC/spl/eca/oi.html`.

[19] Gregor Kiczales, Robert DeLine, Arthur Lee, and Chris Maeda. Open implementation analysis and design of substrate software. In *Tutorial Notes, OOPSLA 95*, Austin, TX, October 1995. ACM/SIGPLAN.

[20] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[21] Ravi Konuru, Jeremy Casas, Robert Prouty, Steve Otto, and Jonathan Walpole. A user-level process package for PVM. In *Proceedings of Scalable High Performance Computing Conference*, 1994.

[22] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Winter USENIX*, pages 29–41, San Diego, CA, January 1993.

[23] Bodhisattwa Mukherjee. A portable and reconfigurable threads package. Technical Report GIT-ICS-91/02, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, June 1991. Also appears in Proceedings of Sun User's Group Technical Conference, pages 101–112.

[24] M.L. Powell, S.R. Kleinman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS Multi-thread Architecture. In *Proceedings of USENIX Winter Technical Conference*, Dallas, TX, 1991.

[25] Carl Schmidtmann, Michael Tao, and Steven Watt. Design and implementation of a multithreaded Xlib. In *Winter USENIX*, pages 193–203, San Diego, CA, January 1993.

[26] SunSoft Manual Set. *Solaris Multithreaded Programming Guide.* SunSoft Press, 1996. ISBN 0-13-160896-7.

[27] W. Staringer. Constructing applications from reusable components. *IEEE Software*, 11(5):61–68, September 1994.

[28] D. Stein and D. Shah. Implementing lightweight threads. In *Proceedings of USENIX Summer Technical Conference*, San Antonio, TX, July 1992.