Tools and Techniques for Building Fast Portable Threads Packages

David Keppel

University of Washington
Technical Report UWCSE 93-05-06

ABSTRACT

Threads are units of concurrent execution that can be viewed as abstract data types (ADTs) with operations to initialize and run them. It is common to improve performance by hard-coding allocation and scheduling policies, but that has led to the development of many threads packages, each with policies tuned for a particular set of applications. Further, the machine-dependence of threads packages restricts the availability of applications built on top of them. This paper examines techniques for building threads packages and discusses tradeoffs between performance, flexibility, and portability. It then introduces QuickThreads, a simple threads toolkit with a portable interface. This paper shows how QuickThreads can be used to implement a basic uniprocessor threads package and discusses the implementation of portable threads packages tuned to the needs of particular applications. For example, QuickThreads can be used to build barrier synchronization that runs in $O(\lg_2 \textit{processors})$ time units instead of the traditional $O(\lg_2 \textit{threads})$. This paper also reports on the performance and implementation of QuickThreads and describes some experiences using it to reimplement and port existing multiprocessor threads packages.

1. Introduction

Programming constructs usually trade between flexibility and overhead. For example, parallel constructs such as vector instructions have low scheduling overhead but are inflexible; processes are more flexible but have higher scheduling costs. Likewise, constructs often trade between portability and overhead. Thus, programs are decomposed so that each division uses the construct that provides the best match between flexibility and performance; and where portable constructs are inadequate, nonportable constructs are used. Conversely, system designers strive to reduce the overhead of constructs and remove machine dependencies from their interfaces so that the constructs can be used by the widest variety of clients.

This paper discusses two techniques for building fast portable threads packages. The first idea is to build threads packages around a thread abstract data type (ADT) that has a machine-independent interface and which encapsulates only the most machine-dependent operations: thread initialization and execution. The second idea is to expose portable details of the threads package implementation so that higher-level thread operations can be optimized to the thread implementation. For example, a barrier can be built using a dedicated run queue, so that one test checks both for queue empty and whether all threads have reached the barrier.

These techniques are discussed in the context of *QuickThreads*, a threads package core. QuickThreads is not a standalone threads package. Rather, it is used to build nonpreemptive user-space threads packages. QuickThreads provides a portable interface to machine-dependent code that performs thread initialization and context-switching. The higher-level threads package, referred to as the *client*, provides portable code to implement scheduling and

---

The author may be reach as `pardo@cs.washington.edu` or at the Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, Washington 98195. As of 1993, the code described in this document is available via anonymous ftp from ‘`ftp.cs.washington.edu`’ in ‘`pub/qt-001.tar.Z`’.

synchronization.

QuickThreads has two goals. One is to make it easy to write and port threads packages. QuickThreads provides machine-dependent code for creating, running and stopping threads. If the QuickThreads client avoids other machine dependencies, porting a threads package built with QuickThreads is as easy as reconfiguring QuickThreads and recompiling.

A second goal is to make threads packages with replaceable components, such as PRESTO [BLL88], but with performance closer to that of hand-coded packages with fixed policies, such as FastThreads [And90]. To achieve this, QuickThreads separates the notion of execution (starting and stopping threads) from thread allocation and scheduling. For example, changing scheduling policies can be as simple as using a different function pointer, and can be done efficiently at runtime. Thus, portable details of the threads package are not fixed, but can instead be tuned to the needs of the application.

QuickThreads is flexible, which can make it slower than hand-coded packages with fixed policies. However, QuickThreads is minimalist, so operations in QuickThreads are often close to the cost for the equivalent hand-crafted operations. QuickThreads is even closer to hand-crafted alternatives on machines with many registers, where the cost of saving and restoring registers dominates. QuickThreads has been used to reimplement two threads packages, and in both cases the performance was unchanged.

The rest of this document is organized as follows. First, Section 2 discusses general tradeoffs between thread context switching models and the idea of reducing the threads core to just the most machine-dependent operations. Then, Section 3 describes the QuickThreads programming model and programmer's interface. Section 4 elaborates by showing how QuickThreads can be used to implement a simple threads package. Next, Section 5 describes various QuickThreads considerations and idioms, including how it is used on multiprocessors and how various mechanisms such as critical sections and barriers are implemented efficiently on top of QuickThreads. Section 6 reports on the performance and implementation of QuickThreads. Section 7 describes and compares related work. Section 8 describes experiences using QuickThreads to reimplement other threads packages. Section 9 describes limitations of QuickThreads and considers future work. Section 10 concludes.

2. Design Decisions

This section describes three key design ideas: pushing synchronization into the client, so that the threads core implements only context switching operations; giving up some performance in order to achieve flexibility, but cutting corners carefully so that core operations are close to the cost of hand-crafted alternatives; and supporting variant argument lists (varargs).

2.1. Synchronization

Threads can avoid explicit dependencies on synchronization idioms by pushing all synchronization operations up to the threads client, which then provides synchronization that is tailored to the client's specific needs. The key is to ensure that the threads interface does not require or prohibit locking across thread context switches.

In many threads packages, a thread that blocks is put on a queue (dead, runable, blocked), and a new thread is removed from the runable queue and restarted. Races can occur on multiprocessors if a blocking thread is put on a queue and some other processor resumes the thread while the blocking thread's stack is still being used to select and start a new thread. Races can even happen in uniprocessors if an exiting thread attempts to free its own stack. There are various ways to avoid the race conditions:

Scheduler Threads: Each blocking operation returns control to a central *scheduler* thread, as in PRESTO [BLL88]. Usually there is one scheduler thread per processor. The scheduler thread enqueues the blocked thread. Races are avoided because a blocking thread is enqueued only once the thread is completely blocked and it's stack is frozen. The scheduler thread is never queued with other threads, so races never occur during switches to and from scheduler threads. A disadvantage of scheduler threads is that each blocking operation requires a context switch to the scheduler then a second context switch to the next thread. Another disadvantage is that on machines without per-processor private memory it is difficult to locate per-processor schedulers cheaply [TSS88, CLBL92] and shared schedulers must be protected with locks.

Locking: The extra context switch to the scheduler can be avoided if the blocking thread locks the queue until the old thread's stack is no longer being used. Some implementations hand-code an unlocking operation in the middle of the context switch. Other implementations provide portability and flexibility by keeping the lock operation out of line, but at the expense of longer lock holding times. When the context switch operation relies on separate lock and unlock operations, it becomes hard to implement nonblocking synchronization [Her88], which uses a single atomic operation.

State in Registers: Stack sharing problems can also be avoided if all transitional state fits in machine registers. Thus, some other processor can start running the old thread even though the processor that blocked it is still starting a new runable thread. A disadvantage of this approach is that the space is limited to machine registers, so only limited kinds of operations can be performed. For example, a stackless thread cannot perform procedure calls. Some architectures [SPA92] and operating systems [BVML92] do not allow a thread to be temporarily stackless.

Preswitch: A fourth approach is to block the old thread, switch to the new thread's stack, then run some code on behalf of the old thread, but using the new thread's stack. Races are avoided just as they are with scheduler threads, but the extra context switch is avoided. There are two innocent-seeming disadvantages that have important implications. First, it must be possible to perform thread operations transparently on the stack of the new thread. That means the new thread must *have* a stack, which may make lazy stack allocation difficult. Second, a thread cannot context switch to itself; thus, a second runable thread must always be available.

Stateless Schedulers: A fifth approach is to create "lightweight" scheduler threads that consist of stack space but no initialized state. A context switch saves the old thread state then switches to the scheduler stack, but no scheduler state is restored. The scheduler stack is simply used as a place to call a function on behalf of the thread that just blocked. Likewise, when the new thread is started, no scheduler state is saved. Stateless schedulers are faster than using heavyweight scheduler threads, because no scheduler state is saved or restored. However, it may still be hard to locate schedulers cheaply on machines without per-processor private memory. Stateless schedulers are similar to storing all transitional state in registers, insofar as scheduler state exists only during the context switch; lightweight schedulers are slightly slower, but are more general because they can perform function calls.

QuickThreads uses preswitch. The interface is designed so that QuickThreads can emulate all the other models except storing transitional state in machine registers.

Since locking is not a part of the QuickThreads context switch primitives, threads packages built with QuickThreads must perform locking at the "ends" of the context switch. Typically this takes teh form of a lock/unlock operation before the context switch to get the next thread and another afterwards to enque the blocked thread.

Hard-coding synchronization might improve performance but is avoided for the reasons discussed above: First, avoiding embedded synchronization improves portability. Second, it keeps the programming model simple. Third, synchronization isn't always needed (e.g., for a new thread that isn't in any locked queue), so the client is free to use just the minimum. Fourth, it may be the *end user* (the client of the threads package) that knows the most about the basic synchronization operations and scheduling, so sometimes even the threads package should avoid supplying them; indeed, this is a central part of the motivation for PRESTO [BLL88] and dynamic processor allocation policies [MVZ90]. Fifth, it would be harder to perform synchronization inline in QuickThreads than in hand-coded systems, and an out-of-line call would make context switches slower. Thus, QuickThreads performs context switching without any locking.

2.2. Flexability and Simplicity

Flexibility can be achieved several ways: by using powerful operations, by using customizable operations, or by stripping away operations that limit flexibility. Each, however, has disadvantages [Kep93]. Powerful operations are often slower. Customizable operations are sometimes slower and existing languages often do not support fine-grained customization. Stripping removes operations that are inflexible, but which also serve a useful purpose; that in turn forces the client to reimplement the functionality of the stripped operations. QuickThreads achieves flexibility by stripping all but the most essential operations, leaving just thread initialization and context switch.

QuickThreads makes several key decisions: First, the QuickThreads operations (initialize, start, stop) are simple enough that they are close to the cost of fixed alternatives. Second, QuickThreads does not depend on any other routines. The client need not worry about QuickThreads introducing spurious races or deadlock by calling e.g., memory allocation routines that are protected by locks. Third, although the client is made responsible for storage allocation, the client is also free to use whatever best suits the client and end user. Fourth, QuickThreads implements no scheduling policies or mechanisms; clients that need fancy policies (e.g., priority) can provide code tailored to the job, while those that need less (FIFO, LIFO) or none (coroutines) can provide whatever is fastest. Finally, QuickThreads likewise lacks semaphores, monitors, nonblocking I/O, and so on. Clients that need these operations can implement them, while those that don't need the operations do not pay for them.

QuickThreads provides only very basic machine-dependent operations to save and restore thread state. Thus, it rarely stands in the way of the rest of a thread package's implementation. The chief bottleneck of QuickThreads, compared to other threads packages, is that scheduling and locking operations are supplied by the client and executed via an indirect procedure call. QuickThreads is designed to minimize the number of excess procedure calls, using two procedure calls on each context switch. Hand-coded threads packages are faster because locking and scheduling policies are fixed, typically simple, and are inlined to minimize holding times and avoid procedure call overhead.

2.3. Variant Argument Lists

Thread creation primitives often take as arguments a function pointer and zero or more arguments to the function. Although this variant argument list or *varargs* interface is convenient for the end-user, it is unreliable and slows thread creation and startup. The problem is that when the thread creation function is called, it must have some way of discovering and setting aside the parameters. Doing so is hard. For example, `printf` discovers the correct number of arguments by examining the first argument; the threads package has no equivalent way of determining the number of arguments. Appendix A discusses the varargs problems in more detail.

From the view of the threads package, a better interface passes only one argument, a pointer to a structure that contains the actual arguments. (It may be necessary to copy the structure to thread-specific storage.) This is the approach used by, e.g., NewThreads [FM92]. However, many packages provide a varargs interface, so building a threads toolkit that supports only single-argument thread functions is unacceptable.

Although varargs can implement a single-argument interface, doing so typically makes thread initialization and invocation two times slower. Therefore, two interfaces are provided: one for varargs, one for the fast case. (The difference is only visible in thread initialization; single-argument and varargs threads are otherwise indistinguishable and may be mixed freely on run queues, etc.) However, using two interfaces makes QuickThreads harder to retarget and understand. In this case, the cost is deemed worthwhile because varargs is typically so much slower.

3. QuickThreads Programming

3.1. QuickThreads Programming Model

QuickThreads does not perform any allocation. It relies on the client threads package to allocate stacks, thread queues, and other auxiliary data structures. Likewise, QuickThreads does not manipulate any run queues or implement semaphores, monitors, and so on. Instead, it provides a simple mechanism that performs a context switch, then invokes a client function on behalf of the halted thread. The client function can then "clean up" the halted thread, putting it on an appropriate queue, deallocating it, etc.

A thread may be in various states: *uninitialized*; *initialized* and ready to run, but not yet started; *running* on a processor; *blocked*, and waiting to be reawakened; or *aborted* (or *dead*), in which case the thread can never be restarted. Initialized threads are started the same way that blocked threads are restarted; when the distinction is unimportant, they are both considered *runable*.

QuickThreads performs all thread manipulation using the thread's stack pointer. A client creates a thread by allocating a stack region. The machine architecture determines whether the stack grows up or down, so the client calls a QuickThreads routine, passing in the address and size of the stack region and getting back the stack pointer of the uninitialized thread. The client initializes a thread by calling a QuickThreads initialization primitive. The primitive initializes the stack with functions and arguments that will be used once the thread is started.

An initialized thread is indistinguishable from a suspended thread. Thus, the initialized thread may be started by passing the initialized thread's stack pointer to the QuickThreads context switching primitive.

The QuickThreads context switching primitive is called with the stack pointer of the next thread to run, and a *helper* function and arguments used to clean up the old thread once the context switch has completed. The helper function is a parameter to the QuickThreads context switch primitive (called `qt_cswap` in Figure 1), so it can be changed dynamically and can be different for each context switch site and for each thread.

```
client_cswap()
{
  new = schedule (runq)
  qt_cswap (new, runq, helper)
  runs_when_restarted()
}

helper (old, runq)
{
  deschedule (old, runq)
}
```

Figure 1. A context switch implementation.

The context switch primitive works as follows. First it saves the register values of the old thread on to the stack of the old thread, adjusting the stack pointer as needed. It then switches to the stack of the new thread; at this point the old thread has been suspended. Before the new thread is restarted, the context switch routine calls the client-supplied helper function. The helper function is passed the stack pointer of the old thread and certain arguments that were passed by the old thread as arguments to the context switch primitive. Often, one of the arguments is a queue, and the helper function puts the blocked thread in that queue. When the helper function returns, the context switch routine restores the registers of the new thread and restarts it.

Figure 2 shows a context switch in progress. The old thread has been suspended and control is now running on the new thread's stack. However, the new thread has not yet been resumed; a helper is executing on the stack of the new thread (stacks grow up in this figure). When the helper returns, the new thread will be resumed. Later, when `old` is resumed, a helper will run on it's stack.

```
                    |       |
                    |   :   |
                    |   :   |
|       |           +-------+ <- sp
|   :   |           | helper|
|   :   |           |  call |
+-------+ <- old    | frame |
| saved |           |       |
| regs. |           +-------+ <- new
+-------+           | saved |
|       |           | regs. |
|       |           +-------+
|       |           |       |
|       |           |       |
|       |           |       |
+-------+           +-------+
  old's               new's
  stack               stack
```

Figure 2. Stacks during context switch.

Threads are generally said to be in some kind of queue except when they are running. However, QuickThreads allows use of arbitrary data structures (arrays, graphs, fixed memory locations) and threads can be run while they are still embedded in a data structure.

3.2. Programming Interface

QuickThreads is written in C and assembler and is designed to work with threads packages written in C. Modules that use QuickThreads primitives must include the QuickThreads header file:

```
#include <qt.h>
```

Typically the *include path* option is used to tell the C compiler where to locate the header. The executable must be linked with the QuickThreads runtime library. The command below is typical, but varies from system to system.

```
cc -I/usr/local/include -L/usr/local/lib ... -lqt
```

The basic interface consists of routines (either macros or functions) to create, initialize, run, and stop threads.

3.2.1. Thread Creation

```
int QT_STKALIGN
qt_t *QT_SP (void *sto, int size)
```

The QuickThreads client allocates stacks. QuickThreads then manipulates threads using the thread's stack pointer. Stacks grow up on some machines and down on others, so the thread's stack pointer is discovered using `QT_SP`, which takes a pointer to the client-allocated stack and the size of the stack, and returns the thread's initial stack pointer. Different machines have different alignment requirements, so the storage `sto` must be aligned on at least a `QT_STKALIGN`-byte boundary, where `QT_STKALIGN` is a power of 2. Likewise, `size` must be a multiple of `QT_STKALIGN`.

This interface fails to be machine-independent in an important way: there is no easy way to get QuickThreads, the client, and the end user of the threads package to cooperate to estimate the stack size.[1] The usual solution is to overestimate the stack size.

3.2.2. Single-Argument Thread Initialization

```
typedef void *(qt_userf_t)(void *u);
typedef void (qt_only_t)(void *u, void *t, qt_userf_t *userf);

qt_t *QT_ARGS (qt_t *sp, void *u, void *t,
               qt_userf_t *userf, qt_only_t *only);
```

A thread is initialized by saving state on the thread's stack. The state is saved in a way that makes it appear that the thread has been suspended just as it is about to call a client-supplied function. The initialization routines take stack pointers returned by `QT_SP`; the initialization routines return initialized stack pointers.

The `QT_ARGS` routine is used for implementing threads packages that use a fast calling sequence with only a single fixed argument. `QT_VARGS` is used when varargs is used. Once a thread has been initialized the client does not need to distinguish between threads that are initialized but not yet run and those that have run.

`QT_ARGS` initializes a thread so that when the thread runs, the function `only` is called with three arguments: `u`, `t`, and `userf`. It is expected that `only` will perform thread-specific initialization, call `userf`, and then perform thread-specific cleanup. The cleanup must halt the thread; it is an error for `only` to return. The argument `u` is passed to the user function, while `t` is thread information for initialization and cleanup. Figure 3 shows typical code for `thread_create` and `only`.

1. Choosing the stack size is a portability problem with most threads packages. The principal exception is threads for languages that allow non-contiguious allocation of stack frames.

```
  thread_t *
thread_create (userf, arg)
{
  t = allocate_thread (...);
  t->sp = QT_ARGS (t->sp, arg, t, userf, only);
  return (t);
}

only (u, t, userf)
{
  global_current_thread = (thread_t *)t;
  (*userf)(u);
  next = dequeue (runable_queue);
  QT_ABORT (..., t, next);             /* Halt the thread. */
}
```

Figure 3. Example thread creation and startup code.

3.3. Varargs Thread Initialization

```
typedef void *(qt_vuserf_t)(...);
typedef void (qt_startup_t)(void *t);
typedef void (qt_cleanup_t)(void *t, void *vuserf_return);

qt_t *QT_VARGS (qt_t *sp, int nbytes, void *vargs, void *t,
                qt_startup_t *startup, qt_vuserf_t *userf,
                qt_cleanup_t *cleanup);
```

When varargs is needed, `QT_VARGS` is used to initialize the thread. It pushes the `nbytes` of arguments, with arguments described by `vargs`, a value returned by the C *stdargs* facility. The initialization routine also arranges that the function `startup` will be called with parameter `t`, then `userf` will be called with the arguments described by `varargs`, (but *not* the `nbytes` parameter), then `cleanup` will be called with parameter `t` and `vuserf_return`, the return value from `vuserf`. Figure 4 shows typical code for `thread_create`, `startup`, and `cleanup`.[2]

The `QA_VARGS` interface has `userf` called directly, because that is what is most efficient and portable. However, the functions `startup` and `cleanup` must be called separately, instead of being inlined in `only`, as they were with `QT_ARGS`. `QT_VARGS` also requires more copying, which slows thread initialization and startup.

Except for initialization, QuickThreads does not distinguish between threads initialized with `QT_ARGS` and those initialized with `QT_VARGS`. Thus, single-argument and varargs threads may be mixed freely.

3.3.1. Context Switching

2. The code is typical, but not strictly portable. Unfortunately, there is no truly portable way to do varargs in C. See Appendix A for details.

```
#include <stdarg.h>

  thread_t *
thread_create (func, nargs, ...)
{
  t = allocate_thread (...);
  va_start (ap, nargs);
  t->sp = QT_VARGS (t->sp, nargs * sizeof(word), ap, t,
                    startup, func, cleanup);
  va_end (ap);
  return (t);
}

startup (t)
{
  global_current_thread = (thread_t *)t;
}

cleanup (t, vuserf_return)
{
  ((thread_t *)t)->retval = vuserf_return;
  next = dequeue (runable_queue);
  QT_ABORT (..., t, next);             /* Halt the thread. */
}
```

Figure 4. Typical code for thread creation, startup and cleanup.

```
typedef void *(qt_helper_t)(qt_t *old, void *a0, void *a1);
typedef void *(qt_block_t)(qt_helper_t *helper, void *a0, void *a1,
                           qt_t *new);

qt_block_t QT_ABORT, QT_BLOCK, QT_BLOCKI;
```

A thread blocks by calling a QuickThreads *blocking routine*, `QT_ABORT`, `QT_BLOCK`, or `QT_BLOCKI`. The blocking routines halt the old thread, move to the stack of the thread `new`, and call the helper function, `helper`, with the stack pointer of the old (blocked) thread and the arguments `a0` and `a1`. When `helper` returns, the new thread is unblocked and resumed.

Note: a thread cannot context switch to itself.

The blocking routines vary in how much state they save on a context switch. `QT_BLOCK` saves all processor registers. `QT_BLOCKI` saves just the integer registers, reducing the context switch overhead. `QT_ABORT` aborts the old thread. `QT_ABORT` saves no registers and passes a garbage `old` value to the helper function; it is thus fastest, but the thread cannot later be resumed. (Some calling conventions require a stack walk when a thread exits. In these cases `QT_ABORT` may be *slowest*, but it still must be used to terminate a thread.)

The return value from the blocking routines is a pointer used to support communication between threads. In some systems, a blocking thread's helper function is called on a scheduler's stack. QuickThreads arranges that the return value from the helper function is used as the return value from the scheduler's call to a blocking routine. For example, the scheduler calls `QT_BLOCKI` to run a thread, the thread runs for a while then calls `QT_ABORT` to halt, call the helper, and then resume the scheduler by letting the scheduler's `QT_BLOCKI` return. The return value from the helper is used as the return value from the scheduler's call to `QT_BLOCKI`. This return mechanism is usually avoided because the blocking thread generally knows best what cleanup code should be run and because it is hard to pass multiple arguments through a single return value.

3.3.2. Allocating Space On The Stack

```
void *QT_SALLOC (qt_t *sp, unsigned size);
qt_t *QT_SADJ (qt_t *sp, unsigned size);
qt_t *QT_SUNADJ (qt_t *sp, unsigned size);
```

Sometimes it is useful to allocate temporary space on the stack of a blocked thread. Stacks can grow up or down, so the routines `QT_SALLOC`, `QT_SADJ`, and `QT_SUNADJ` hide stack direction and ensure that valid data remains unclobbered.

`QT_SALLOC` is similar to the standard library's `malloc`, but the `size` parameter must be a multiple of `QT_STKALIGN` and it returns storage that is aligned on `QT_STKALIGN`-byte boundaries. Note that on essentially all machines, the stack is at least maximally aligned. That is, the stack must be at least as aligned as any other data type. However, the stack may have much *stricter* alignment than other data types, so allocating a small region may consume substantially more memory.

`QT_SALLOC` does not adjust the stack pointer, so if it is used several times in succession, `QT_SADJ` must be called before each call except the first. It is usd each time with the same `size` parameter as the preceding call to `QT_SALLOC`; a new stack pointer is returned. When the thread is restarted, the unadjusted stack pointer must be used. The stack pointer can be "unadjusted" either by saving the unadjusted value or by calling `QT_SUNADJ`.

> *Note:* For each call to `QT_SADJ` there must be a distinct call to `QT_SUNADJ`, because `QT_SUNADJ (x+y)` is not the same as `QT_SUNADJ (x); QT_SUNADJ (y)`.

3.3.3. Starting QuickThreads

Most threads packages explicitly move from running as a single-threaded process to running threads. In QuickThreads, a thread is represented only by its stack pointer. The main process has a stack pointer, and, thus, is already a thread. Since it is already running, it does not need to be initialized. The main thread is blocked and restarted using the normal QuickThreads primitives.

3.3.4. Debugging

If the macro `QT_DEBUG` is defined, some of the operations will do extra work to help with debugging. The goal is to get errors reported more quickly. This functionality is less than great, but better than nothing.

> *Note:* A given implementation may provide anything from lots of help no help at all.

4. Examples

This section walks through a simple uniprocessor package, SimpleThreads, built using QuickThreads.[3] In the following, names starting with `qt_` and `QT_` are all parts of the QuickThreads package described above. Names starting with `stp_` and `STP_` are parts of SimpleThreads. The meaning of other names should be clear from the context.

4.1. Data Structures

The basic SimpleThreads thread is:

```
typedef struct stp_t {
  qt_t *sp;              /* QuickThreads handle. */
  void *sto;             /* 'malloc'-allocated stack. */
  struct stp_t *next;    /* Next thread in the queue. */
} stp_t;
```

---

3. See `stp.c` and `stp.h` in the QuickThreads distribution.

Threads are enqueued on a singly-linked list that uses the `next` field in each thread. Queues are represented with an `stp_q_t`, and are accessed with these routines:

```
void stp_qinit (stp_q_t *q)
void stp_qput (stp_q_t *q, stp_t *thread)
stp_t *stp_qget (stp_q_t *q)
```

The routine `stp_qget` returns a null pointer if the queue is empty.

4.2. SimpleThreads Package Initialization

SimpleThreads must be initialized before any other operations are performed. The runable queue is initialized to be empty.

```
static stp_q_t stp_global_runq;

  void
stp_init()
{
  stp_qinit (&stp_global_runq);
}
```

4.3. SimpleThreads Startup

One or more runable threads are created (discussed below) and put on the runable queue. Then `stp_start` is called to start the SimpleThreads package. It removes a thread from the runable queue, blocks the main thread, calls a helper to enqueue the blocked main thread, and runs the next thread.

```
static stp_t *stp_global_curr;

  void
stp_start()
{
  static stp_t stpmain;      /* Thread for the process. */
  stp_t *next;

  while ((next = stp_qget (&stp_global_runq)) != NULL) {
    stp_global_curr = next;
    QT_BLOCK (stp_yieldhelp, &stpmain, &stp_global_runq, next->sp);
  }
}
```

The main thread sets the notion of the current thread (`stp_global_curr`) to be `next`, then blocks and calls `stp_yieldhelp` on the next thread's stack. After the context switch, `stp_yieldhelp` saves the stack pointer of the blocked main thread, `sp`, by storing it in `stpmain` (pointed to by `old`), and then puts the main thread in the runable queue `stp_global_runq` (pointed to by `blockq`). Note that `QT_BLOCK` calls it's first parameter, `stp_yieldhelp`; and that the second and third arguments to `QT_BLOCK` are used as the second and third arguments to `stp_yieldhelp`.

```
  static void *
stp_yieldhelp (qt_t *sp, void *old, void *blockq)
{
  ((stp_t *)old)->sp = sp;
  stp_qput ((stp_q_t *)blockq, (stp_t *)old);
}
```

The main thread runs periodically because it is stored in the runable queue. When the main thread is restarted, control returns to `stp_start`, which checks for other runable threads, If there are no runable threads, `stp_start` returns, exiting the SimpleThreads package.

When the main thread is in the run queue, stp_qget can never return NULL. Thus, the threads package termination check appears only in stp_start instead of appearing in each context switch. That saves a check at every context switch, but if the run queue is typically short, checking in stp_start causes extra context switch overhead because the main thread is invoked frequently to check for termination.

4.4. Thread Creation

```
typedef void (stp_userf_t)(void *u)

void stp_create (stp_userf_t *f, void *u)
```

Threads are created with stp_create, where f is an end-user function that takes one void * parameter, u. The newly-created thread is put in the run queue.

Creation proceeds as follows: First, a SimpleThreads thread body (stp_t) and stack region are allocated. The stack region is rounded. Since the stack may shrink during rounding, QT_SP is passed a size that is slightly smaller than the actual allocation. Then, the thread is initialized with QT_ARGS so that when the thread starts running, it will call the routine stp_only with f, a pointer to the user function, the user function argument, u, and and a pointer to the SimpleThreads thread, t. The routine stp_only is discussed in more detail below.

The stack size, STP_STKSIZE, is a multiple of QT_STKALIGN and is chosen arbitrarily. xmalloc is like the C library malloc, but it never fails.

```
  void
stp_create (stp_userf_t *f, void *u)
{
  stp_t *t;
  void *sto;

  t = xmalloc (sizeof(stp_t));
  t->sto = xmalloc (STP_STKSIZE);
  sto = STP_STKALIGN (t->sto, QT_STKALIGN);
  t->sp = QT_SP (sto, STP_STKSIZE - QT_STKALIGN);
  t->sp = QT_ARGS (t->sp, u, t, (qt_userf_t *)f, stp_only);
  stp_qput (&stp_global_runq, t);
}
```

The end user can call stp_create any time after calling stp_init. The application must create at least one thread before calling stp_start, or stp_start will find the run queue empty and will simply return.

When the thread first runs, it calls stp_only. The running thread is initialized by saving t to a global variable. t is a pointer to the stp_t of the new thread, and it is used later by blocking operations to put the thread in blocked and runable queues. Next, stp_only calls the user function, f. If the user function returns, the thread is aborted, since it is an error for stp_only to return.

```
static stp_t *stp_global_curr;

  static void
stp_only (void *u, void *t, qt_userf_t *f)
{
  stp_global_curr = (stp_t *)t;
  (*(stp_userf_t *)f)(u);
  stp_abort();
  /* NOTREACHED */
}
```

4.5. Context Switching

```
void stp_abort (void)
void stp_yield (void)
```

A context switch halts the current (old) thread and runs a next (new) thread. There are two forms: one aborts and destroys the current thread. The other merely yields the current thread so that other threads can run.

The `stp_abort` routine selects a new thread to run, changes the notion of the currently-running thread, then aborts the old thread, calling `stp_aborthelp` to clean up. Since the main thread is in the run queue, there is always a next thread and never a need to check if `stp_qget` returned `NULL`.

```
  void
stp_abort (void)
{
  stp_t *old, *new;

  new = stp_qget (&stp_global_runq);
  old = stp_global_curr;
  stp_global_curr = new;
  QT_ABORT (stp_aborthelp, old, NULL, new->sp);
}
```

The helper `stp_aborthelp` runs after the old thread has been halted completely. It simply deletes the storage associated with the old thread. The helper `stp_aborthelp` runs on the stack of the new thread, so it is safe to delete the stack of the old thread. When `stp_aborthelp` returns, the new thread is resumed.

```
  static void *
stp_aborthelp (qt_t *sp, void *old, void *unused)
{
  free (((stp_t *)old)->stk);
  free (old);
}
```

Yielding is similar, but the helper function is called with a pointer to the queue on which to block the old thread. The helper function is `stp_yieldhelp`, described above. For yielding, the queue is just the run queue. Semaphores and monitors (if implemented) would use different queues.

```
  void
stp_yield()
{
  stp_t *old, *new;

  new = stp_qget (&stp_global_runq);
  old = stp_global_curr;
  stp_global_curr = new;
  QT_BLOCK (stp_yieldhelp, old, &stp_global_runq, new->sp);
}
```

QuickThreads does not allow a thread to context switch to itself. Thus, whenever a thread yields there must be another runable thread. In SimpleThreads, the main thread is always in the runable queue, so there is always a new thread to run. A different implementation might store the main thread elsewhere, in which case `stp_qget` could return `NULL`. All routines that call `stp_qget` would then need to check the return value. `stp_abort` would interpret `NULL` to mean that SimpleThreads should halt. `stp_yield` would use `NULL` to mean that the current thread is the only runable thread, so `stp_yield` should return, continuing the thread, rather than blocking and letting another thread run.

Note that in the current implementation, the main thread cannot call `stp_abort` or `stp_yield`. If it did, there might not be another runable thread (also, `stp_global_curr` might not be set correctly).

5. Considerations and Idioms

This section discusses some issues to be considered when using QuickThreads, some common idioms, and some uses of flexible scheduling.

5.1. Context Switching Models

QuickThreads can be used to emulate several other context switching models: locking across context switches, using scheduler threads, etc.

One model holds the lock across a context switch by performing the *lock* operation before calling the context switch primitive and performing *unlock* in the helper function. However, lock holding times will be longer than with hand-coded packages.

Another model returns control to a central scheduler thread. Since QuickThreads returns only a single pointer value, multiple values are passed in memory, pointed to by the single return pointer.

A third model uses “lightweight” scheduler threads that are simply stacks for helper functions to run on. The stacks have no underlying thread, so the helper function must never return. The helper function instead selects a new thread, then runs the new thread by aborting itself.

5.2. Multiprocessors

QuickThreads uses no global state. However, the client threads package must be concerned about synchronization and sharing for the queues that maintain threads between context switches, and the client must also be concerned with maintaining the notion of the current thread. Thus, the client must ensure threads are properly synchronized. The following describes some synchronization issues that must be managed by the client threads package.

The queue access functions must be coded for multiprocessor synchronization.

If the machine has per-processor private memory, the notion of the current thread can be private to each processor. If not, the current thread must be derived from the stack pointer, as in Amber [Cha90] and threads under scheduler activations [BMVL92], or a machine register must be set aside to point to the thread.

Multithreaded applications often use one process per processor and multiplex threads on top of each process. The stacks associated with the initial processes are called *main threads*. With several processors there are several main threads. Typically, main threads cannot be stored in the runable queue. On machines with per-processor private memory, some virtual addresses – notably, those used for main thread stacks – access different memory depending on which processor issues the reference. Thus, main threads must be per-processor and stored in per-processor memory. On machines without private memory, care is needed storing main threads in the runable queue, since when one processor is context switching, neither the old thread nor the new thread is in the runable queue. Thus, if a processor switches between two main threads, there are briefly fewer available main threads than processors.

On a uniprocessor, an empty runable queue indicates that there is no more available work. On a multiprocessor, the runable queue can be empty, but running threads on other processors can create new work. Thus, main threads must terminate only when all processors are idle.

5.3. Multiple Run Queues

Since QuickThreads does not define run queues, applications can use multiple queues and scheduling strategies. Multiple queues can be used various ways. For example, libraries can be implemented using threads and coroutines, without regard for whether the rest of the application uses threads. Also, threads used to implement simple control transfer (e.g., coroutines) can be implemented cheaply, even though other threads in the same application may use more complicated and expensive control transfer rules (e.g., priority scheduling). Further, simple queue status, such as “queue empty”, can be used to mark particular conditions, e.g., “all threads have reached the barrier”; in a conventional threads package, the queue may be inaccessible (buried in the implementation) or there may be other (non-barrier) threads in the run queue. Finally, When the number of threads is known, it is possible to implement run queues using faster fixed-size data structures such as arrays instead of linked lists.

5.4. Blocking Critical Sections

Blocking critical sections test a condition and, if true, block the current thread on a queue. If testing and blocking are not a single atomic operation, a race can arise when the condition goes false between the test and the context switch. QuickThreads does not provide atomic testing and enqueing; clients can supply several implementations, with tradeoffs.

On non-preemptive uniprocessors, no other threads run between the test and the block, so no race can arise. On preemptive uniprocessors, preemption can be disabled, deferred, or squashed so that no threads run between the test and enqueue; alternatively, the enqueue can check if there was a preemption and retry, as with optimistic synchronization [Her88].

On multiprocessors, the current thread can acquire a lock while testing the condition, then hold the lock until the thread has been blocked and put on the blocked queue. This technique is used by many threads packages. One disadvantage is that locks are held across context save and restore, increasing lock holding times and contention.

Another choice is to have the current thread test the condition and block *tentatively*. The helper function then atomically retests the condition and enqueues the thread if the condition is still true. If the condition has meanwhile gone false, the blocking thread is put in the runable queue. When the thread restarts, it retests the condition. One disadvantage is the cost of excess context switches. Another is that a thread which was ready to enter the critical section is now delayed, which may cause latency problems [ELZ86] or starvation.

Some of the latency and starvation problems can be solved by blocking tentatively, then restarting the thread if the condition has gone false during the context switch. In order to perform the tentative context switch, a thread was removed from the runable queue. When the blocking thread is restarted, the new thread is put back in the runable queue. Since the new thread was never actually run, the stack pointer remains unchanged, and the thread can simply be reinserted in the runable queue.

5.5. Barriers

A *barrier* is a control construct, where no thread can pass the barrier until all of a set of threads have reached the barrier. A typical implementation initializes the barrier with a count of the number of threads, and decrements the count as each thread reaches the barrier. Each decrement requires synchronization, so the barrier cost scales as *O(Nthreads).*

An alternative is to keep a per-processor count of the number of threads that have reached the barrier and to ensure that only barrier threads are in the (current) run queue. When the run queue goes empty, the processor can decrement the shared count by the number of threads that have been run by that processor. Thus, the barrier synchronization cost scales as *O(Nprocessors)* instead of *O(Nthreads)*.

Another alternative is to use a tree that synchronizes pairs of threads, then pairs of pair-synchronized threads, and so on. Although the total number of synchronization operations increases to $O(N\lg_2 N)$, the parallelism increases and the worst-case synchronization time is $O(\lg_2 N)$. Traditional threads packages synchronize pairs of threads so the barrier cost scales as $O(\lg_2 Nthreads)$. The same technique can be used with per-barrier run queues: instead of synchronizing threads, the tree synchronizes pairs of processors. Thus, barrier synchronization can be performed in $O(\lg_2 Nprocessors)$ synchronization times.

With a straightforward run queue implementation, there are *O(Nthreads)* synchronization operations that remove threads from the run queue. Distributed run queues can be used to increase parallelism and reduce contention [ALL89], but care is needed because straightforward implementations require *O(Nprocessors)* synchronization times to check for an empty run queue.

Standard barriers can have arbitrary threads in the run queue. Barrier implementations using QuickThreads can use per-barrier run queues. Threads are inserted in the run queue only when the barrier is being (re)started. Thus, a processor queue that becomes empty never needs to be rechecked during a barrier operation. It is thus possible to build a run queue that can check for queue empty without any synchronization.[4] However, thread migration (moving threads from busy processors to idle ones) still requires synchronization operations.

5.6. Marker Threads

A *marker thread* or just *marker* is one placed strategically in the run queue so that simply running the thread indicates a special queue condition that the marker then handles.[5] For example, a marker can be inserted as the last element in a LIFO (stack) run queue. The marker is run only when there are no other runable threads. The marker can then allocate stacks to startable threads, check for deadlock, start the next phase of the program, etc.

Marker threads are useful because they put control in a single place rather than e.g., requiring a check for empty queue on each access. However, an extra context switch is required each time the marker is invoked. The “main” thread in SimpleThreads is a marker: when the “main” thread runs, it checks for queue empty and, if necessary, halts the threads package. Because the “main” thread is always in the run queue, no other queue access needs to check if the queue is empty.

It is possible to have multiple marker threads in a queue.

5.7. Stackless Threads

A stackless thread is also called a *thread template* [BLL88]. Since threads are typically small except for their stack, deferring stack allocation can reduce memory consumption because only running threads have stacks. A QuickThreads thread is represented by a stack pointer, so it must implicitly have a stack. The client can build stackless threads several ways:

- The client thread is allocated without a stack. All threads (both complete and stackless) are on the same run queue and the context switch code checks each thread and assigns a stack if necessary.
- Stackless threads are put on a *startable* queue. When the run queue goes empty, thread templates are given stacks and made runable. Since stack allocation takes place as a part of handling an empty run queue, normal context switches run at full speed.
- Each thread is initialized with a small stack. When the thread starts running, the thread’s *startup* function reinitializes the thread with a large stack and restarts it. If stack allocation is nontrivial, it must be performed by a thread with a normal-sized stack.

With all of these schemes, the non-varargs case is fairly straightforward, but varargs is harder.[6]

5.8. Parameter Limits for Blocking Routines

Blocking routines pass only a few parameters to the helper function. More parameters can be “faked” by having the caller push some values on the old thread’s stack, then pass a pointer to that storage to the helper function. The rationalle for passing only a few parameters is that the case of a few parameters is common, passing more parameters is expensive, and that having the flexibility to pass more parameters shouldn’t penalize the common case.

5.9. Minimizing Thread State

In the SimpleThreads example, the notion of the current thread was encoded in a structure `stp_t`. It would be best if client threads packages could manipulate a thread by simply manipulating its stack pointer and without adding any state (such as an `stp_t`) to that required by QuickThreads. Usually, however, when a thread exits it must be possible to discover the base of the stack, given just the stack pointer.[7] One common way to find the stack base is to

4. It is also possible to build tree-structured run queues so that one data structure implements both the run queue and the barrier.
5. Marker threads are similar to control constructs in threaded interpreters and to the final states in finite automata [Tho68].
6. It is tempting to try to set aside `nbytes` of storage pointed to by `vargs`. However, the argument list described by `vargs` may actually represent more than `nbytes` of storage, may not point to the beginning or end of that storage, might not even be a pointer, and the argument list may have constrained alignment. See Appendix A.
7. It would be useful to have the capacity to unwind the stack and call the cleanup function, but no such facility is provided.

keep a global pointer (e.g., SimpleThreads's `stp_global_curr`). Another is to allocate all stacks on aligned boundaries; the stack base is found by rounding the stack pointer. The aligned/rounding technique is used by Amber [Cha90] and threads under scheduler activations [BMVL92].

Some queues, such as the SimpleThreads queues, expect each thread to provide storage for the `next` field. Such storage can be allocated on top of the thread stack using `QT_SALLOC`.

5.10. Auto-Grow Stacks

In some circumstances it is useful to grow the stack on demand, saving backing store allocation for threads that actually use large stacks. To implement this, each stack allocation is followed by a large unallocated region that is mapped to cause a fault on access. When the stack overflows, stack references go to the unallocated region. These references invoke a fault handler that maps additional storage in to the region, thus growing the stack [Gru91]. Stacks space can also be "scavenged", unmapping storage between the thread's current top-of-stack and the end of the thread's stack region.

5.11. Shared Stacks

The preceding discussion of thread stacks assumed that distinct storage is used for each thread's stack. An alternative is to have a shared stack. When a thread is restarted, its stack data is copied from the heap to the shared stack. When a thread is halted, the thread's stack data is copied back to the heap. Although copying takes a lot of time, a shared stack reduces the space used by halted threads. The intuition is that private stacks are large because they give each thread as much space as it might ever need. The shared stack is large so it can run threads, but the space used by each blocked thread needs to be only as big as the thread's state at the time it blocks. For some applications, allocating a few hundred bytes per thread is acceptable, but allocating e.g., a 4Kb stack for each runable thread will consume all available memory [Ros92, EAL93, Pin93].

Shared stacks can be implemented using QuickThreads. The thread's *current stack* is the region between the uninitialized stack pointer and the thread's current (blocked) stack pointer. Here, "between" includes the bytes at the lowest address but not those at the highest address. Since the stack can grow up or down, the current stack pointer may be larger or smaller than the uninitialized stack pointer. Thus, the region is either *[initialized..uninitialized)* or *[uninitialized..initialized).*

Each time a thread blocks it switches to a scheduler thread. The scheduler thread must use a different stack, but since there is only one scheduler per processor, the space overhead is small. The scheduler saves the old thread's stack data, finds a new thread, restores its stack data to the shared stack, then resumes the new thread.

Another alternative is to have a small number of shared stacks. Each thread can be in one of two states: *cached* or *noncached*. A cached thread has its stack data on a shared stack. An uncached thread must be copied to a shared stack before it can run. When a thread is first started it is assigned to one of the stacks, called its *designated* stack; a thread always runs on its designated stack. A context switch can go directly from an old thread to a cached new thread. An uncached new thread needs a context switch to the scheduler, where the thread on the designated stack is copied back to heap storage, and the new thread is copied in and run. The context switch to an uncached thread must go via the scheduler because the old and new threads might use the same designated stack. Context switches are fastest between cached threads, so the scheduling policies should try to run threads that are cached.

5.12. Preemption

QuickThreads is designed for nonpreemptive threads packages. Preemptive threads can be implemented by having timer signal handlers pushed on the thread's stack rather than going on a special signal stack. The signal handler can block the current thread using nonpreemptive primitives. Timer signals are reenabled when the signal handler returns. The signal handler doesn't return until the thread resumes, so timer signals must be reenabled explicitly. Prototypical code is shown in Figure 4.

Often, involuntary context switches should be disabled during e.g., critical sections. Context switches can be disabled easily by setting a global. The `sig_timer` routine checks the global, and, if set, the signal handler sim-

```
sig_timer()
{
  QT_BLOCK (..., sig_timer_helper, ...);
}

sig_timer_helper (...)
{
  ... clean up old thread ...
  ... reenable timer signals ...
  ... set the timer ...
}
```

Figure 4. Prototypical code for preemptive threads.

ply resets the timer and returns. With this technique, threads in locked sections can get excess quanta. However, the threads are doing critical section work, so the effective priority increase is usually acceptable.

> *Note:* A preemption during a context switch effectively preempts two threads: when the old thread is running it has a pointer to the next thread; when the helper runs on the new thread's stack it has a pointer to the old (blocking) thread.

This preemption mechanism leaves signal handler frames on the stack and may lead to stack growth if signals arrive while `sig_timer` or `sig_timer_helper` is executing. Workarounds exist, but lead to complicated code. Note also that the timer must be reenabled late to avoid race conditions.

5.13. Persistent Threads

A *persistent thread* is one that can be "frozen" and e.g., put in a disk file. The thread is later read in to memory and resumed. A persistent thread may be frozen and resumed many times; likely, it will be resumed each time in a different program, or at least a different run of the program. There are a variety of issue that must be resolved, such as ensuring the thread occupies the same portion of the address space from run to run, how the thread is to treat pointers outside of the thread's stack region, and how to synchronize with other users of the persistent thread.

QuickThreads does not resolve those issues but does help in one way: it separates the scheduling decisions of the application from the operation of the thread. When an application restarts a persistent thread it can provide the code and data that the thread uses to block itself. Each application can use the same thread with completely different scheduling protocols.[8]

6. Measurements and Implementation

This section reports on the performance of QuickThreads primitives on a number of machines and describes some practical details of the implementations.

6.1. Performance

QuickThreads is designed to give modest performance. QuickThreads alone is minimalist and thus fairly fast. This section gives performance of the primitives on several platforms: an AXP [Sit93] processor, with reported timing figures estimated; one 16MHz Intel i386 processor on a Sequent Symmetry [Seq88]; a 20MHz KSR1 processor [KSR91]; two SPARC-based [SPA92] Suns with kernel support for register windows [Kep91]; a DECstation 5000/200 using a MIPS R3000 processor [Kan87]; and a VAX-based [DEC81] VAXstation 3500. QuickThreads also runs on a platform using Motorola 88100 processors [Mot89], but no timing figures are available.

---

8. Applications will sometimes assume that thread stacks contain certain fields, e.g., a `next` field that is used to link together blocked threads. In this case, an application can create a *surrogate* thread that is scheduled normally. When the surrogate is invoked it invokes the persistent thread. When the persistent thread blocks it returns control to the surrogate, which then blocks in an application-defined way. Scheduling the surrogate thus implicitly schedules the persistent thread.

All measurements in this section are for microbenchmarks. Experiments were performed by running the program once to get it in to memory, then timing each of the microbenchmark runs when run on a lightly-loaded machine. Each microbenchmark executes the given operation(s) a "large" number of times. Time per operation is computed by dividing the elapsed (wallclock) time by the number of times the operation was performed. Since each operation was performed a large number of times and since each microbenchmark performs no other work, cache hit rates are probably close to 100% and register allocation is good because all registers are available for loops and context switches. Thus, the reported times are probably better than will be seen in practice.

As a rough benchmark, Figure 5 shows the raw times for many repetitions of a few simple operations. Note, however, that benchmark times are approximate since the benchmark code may have been optimized by e.g., the assembler. No microbenchmarks were run on the KSR1.

Machine	μSec Null Call	μSec Reg. Add	μSec Load Reg.
AXP	0.90	0.090	0.090
i386	1.89	0.14	–
KSR1	–	–	–
M88100	–	–	–
MIPS R3000	0.19	0.011	0.091
SPARC 4-65	1.36	0.049	0.089
SPARC 4-490	0.99	0.036	0.067
VAX 3500	5.4	0.23	3.5

Figure 5. Basic timing estimates.

Figure 6 shows the times required to perform thread initialization for both the single-argument version and the version that supports varargs, the latter measured passing several different numbers of arguments. Note that the KSR1 version does not currently support varargs.

Machine	Single Arg	Varargs 0	Varargs 2	Varargs 4	Varargs 8
AXP	<0.1	1.4	1.5	1.6	1.7
i386	2.5	15.7	18.1	22.0	–
KSR1	0.6	–	–	–	–
M88100	–	–	–	–	–
MIPS R3000	1.7	3.4	4.5	5.3	7.0
SPARC 4-65	1.8	5.1	6.2	7.4	10.7
SPARC 4-490	0.7	2.5	3.1	4.0	5.4
VAX 3500	4.1	29.0	31.0	36.0	42.0

Figure 6. Microseconds to initialize.

Figure 7 shows the times for integer-only and generic context switches. On some machines, all floating-point registers are caller-save, so integer-only and generic context switches are identical.

Figure 8 shows the time required to initialize, start and stop a thread. The reported time is that required for a "generator" thread to initialize an allocated thread plus two context switches: one to perform an integer context switch to block the generator and a second to abort the new thread and restart the generator. Varargs routines did not use any of their arguments.

Machine	Integer Cswap	Int+FP Cswap
AXP	1.0	2.0
i386	10.4	10.4
KSR1	6.2	14.6
M88100	–	–
MIPS R3000	1.9	4.6
SPARC 4-65	32.3	32.7
SPARC 4-490	16.7	16.5
VAX 3500	21.0	22.0

Figure 7. Microseconds to context switch.

Machine	Single Arg	Varargs 0	Varargs 2	Varargs 4	Varargs 8
AXP	1.1	2.7	2.7	2.8	2.9
i386	22.5	39.9	42.4	46.2	–
KSR1	12.0	–	–	–	–
M88100	–	–	–	–	–
MIPS R3000	4.6	7.2	8.3	9.2	10.7
SPARC 4-65	75.6	80.8	82.2	82.8	85.2
SPARC 4-490	38.6	39.7	41.5	41.2	43.9
VAX 3500	45.0	81.0	87.0	90.0	97.0

Figure 8. Microseconds to start and abort.

6.2. Implementation

The QuickThreads implementation for all platforms except for the KSR is about 400 lines of assembler and 400 lines of C and includes support for user functions that take variant argument lists. KSR support is an additional 400 lines and lacks variant argument list support. In Figure 9, the column “Assembler” is the number of assembly instructions, the column “C” is approximately the number of useful lines of C, and the column “Comments” is the total lines minus the other lines reported and consists of lines that are either blank or that contain only comments.

Machine	Assembler	C	Comments
Common	0	120	80
AXP	100	70	230
i386	35	30	120
KSR1	380	30	190
M88100	60	75	230
MIPS	75	30	150
SPARC	55	30	165
VAX	35	25	110
Total	740	410	1285

Figure 9. Implementation size, in lines of code.

The core is small enough that it should be straightforward to implement for a new machine. In the current implementations there has been no attempt at conciseness, nor have the implementations been tuned. There is no code to help with debugging.

The design of QuickThreads is optimized around several assumptions. It is expected that function calls are cheap and that passing a few unused arguments to the helper routines is cheap. Both of these assumptions are true on most RISC architectures, but may be a source of inefficiency on machines that pass arguments on the stack, etc. In particular, the i386 and VAX ports suffer relative to the RISC processors because the RISC processors can typically call the helper functions by simply changing the stack pointer and first argument register, while the CISC ports have calling conventions that require arguments to be copied from the old thread's stack to the new thread's stack.

Many implementations use essentially the same code for `QT_BLOCKI` and `QT_ABORT`, and `QT_BLOCK` is sometimes implemented by saving floating-point registers, then calling `QT_BLOCKI`.

Most implementations use the same routine both to restore state on context switches and to restore state on thread startup. Startup generally needs to restore only a little bit of state, but the context switch routine must restore many registers. Thus, thread startup time might be substantially better if the saved state of a thread indicated which routine to use to restore state. However, the implementation would be more complicated and context switches might be somewhat slower.

More than ½ of the AXP and 88100 implementations are for handling varargs. Other platforms are typically one-third.

The SPARC port is implemented using techniques described elsewhere [Kep91].

7. Related Work

Coroutines are functions that can both return a value and save their current state for later invocations [Knu73, Pra86]. Coroutines are typically invoked by name (e.g., the predecessor names the successor explicitly), scheduling is typically limited to fixed orderings, and coroutines cannot overlap execution.

Threads, *tasks*, and *lightweight processes* are generalized coroutines where control transfers are implicit (e.g., a thread suspends itself without naming its successor); thread scheduling (the order of execution of threads) can be general, with successors implied instead of being named explicitly; and threads can run concurrently, where coroutines use strict interleaving. A *preemptive* scheduling policy can suspend threads at arbitrary times; threads need not not explicilty invoke blocking constructs. Most threads packages provide various additional constructs for synchronization [Bir89].

Continuations are closures that implement general-purpose control constructs. Although continuations can implement threads,[9] threads do not necessarily rely on language and compiler support. Also, threads are typically optimized to the special case of context switch, where continuations are more general.

PRESTO [BLL88] is an extensible user-space threads package. It allows multiple preemptable threads to run concurrently on a multiprocessor. PRESTO was designed with the philosophy that users should be able to replace components. For example, the default scheduling is round-robin, but stack and other disciplines can be used [BLLW88]. Originally, basic thread creation and invocation cost several hundred microseconds. More recent versions have been sped up by hard-coding various allocation policies.

FastThreads [And90] is a non-preemptive user-space multiprocessor threads package that improves on PRESTO's performance by cleaning up the code, reimplementing in C and assembler, distributing the run queues to reduce contention, by using a cheap scheduling discipline (stack) and by hard-coding all policies. On a Sequent Symmetry, the basic thread context swap operation takes between 25 and 30 microseconds and includes the cost of scheduling and also grabbing and relinquishing a (potentially) shared run queue. FastThreads is thus an order of magnitude faster than PRESTO, but is also less flexible. Where PRESTO and QuickThreads both provide mechanisms for extending their basic features, FastThreads is designed to be extended by rewriting the threads package [And90].

---

9. Thread context switching is essentially control transfer via continuation passing.

Mach Cthreads [CD88] is widely-available and has been ported to many machines. Like FastThreads, many allocation and scheduling decisions are “locked in”. Cthreads is implemented in layers and one of the undocumented inner layers bears resemblance to QuickThreads [Bar93].

POSIX threads, Pthreads, have a set of behavior defaults and a set of attributes that may be set by user code to configure Pthreads to the application. However, since there are many configuration attributes, each thread operation potentially needs to make many runtime decisions in order to evaluate the attributes.

RapidThreads is a non-preempting uniprocessor user-space threads package core that sacrifices everything to get raw performance. Like QuickThreads, it omits nearly all functionality. The thread state is a block of dynamically-compiled self-modifying code. The code contains embedded constants describing how to save and restore the stack pointer and how to jump to the next thread [MP89]. On one processor of a Sequent Symmetry, a thread context switch takes about 5 microseconds.

NewThreads is a non-preempting user-space threads package that runs one thread on each processor of a message-passing multiprocessor. Since run queues are not shared, no locking is needed. NewThreads implements basic synchronization and message passing on top of a threads library.

Psyche [MSLM91], Scheduler Activations [ABLL91], and Synthesis [MP89] are all kernel approaches to providing first-class threads with costs and flexibility close to those of user-level threads. Psyche and Scheduler Activations use two-level scheduling so that most context switches occur without kernel intervention. Synthesis allows user processes to customize kernel threads in safe ways and uses very fast communication between kernel and user spaces [MP88, MP89]. These systems allow each application to choose (or build) a thread library that implements just the minimum needed functionality and, thus, runs with minimum overhead. Psyche emphasizes the use of applications with more than one thread model within a single application, though all of the above systems support multi-model programming to some degree. All of these systems focus on an effective kernel interface. QuickThreads instead focuses on the construction and tuning of the user-level threads packages within an application and could be used for that purpose under Psyche and Scheduler Activations.

Filaments [EAL93] is a nonpreemptive user-space threads package that restricts thread operations to those with particularly cheap implementations: threads do not have individual stacks; thread state is as small as 4 words; full thread creation costs about 30 instructions and thread start/exit can be less than 10. Threads that run in a special order (e.g., LIFO) can be run using a single stack. Threads that block in certain ways can be rewritten as several ordered threads with little state passed between them. Threads that require full stack generality cannot be run directly with Filaments.

QuickThreads attempts to get most of the flexibility of PRESTO and Pthreads, while retaining most of performance of FastThreads, Filaments and RapidThreads. It does so by discarding everything but the core context switch operations. It is up to the client to provide efficient operations with the desired semantics (e.g., FIFO or LIFO scheduling). A previous version of QuickThreads used special compiler support to get FastThreads’ performance without hard-coding any policies. The current version is slightly slower but does not rely on compiler support.

8. Experiences Using QuickThreads

QuickThreads has been used to reimplement two threads packages, PRESTO and NewThreads. The following sections describe some of the experiences and lessons learned.

8.1. Porting PRESTO

PRESTO was originally written for the Sequent and although it has been ported to other machines, the (tricky) core context switch code has not changed since the early prototype. The basic PRESTO context switch design is at odds with QuickThreads and showed some of QuickThreads’ strengths and weaknesses.

For this rewrite, PRESTO was compiled with uniprocessor command flags since initial targets were uniprocessors and spinlocks are a source of machine dependencies. Preemption was disabled because QuickThreads does not support it (although the PRESTO preemption mechanism is essentially the same as the QuickThreads preemption mechanism described above).

PRESTO is written in C++. Function calls typically use a hidden *self* argument. This makes it hard to discover the `vargs` parameter to `QT_VARGS` during thread creation. Furthermore, the original interface used the routine `nargs` to determine the number of arguments, and `nargs` is even less portable than the basic varargs mechanism (See Appendix A). PRESTO also stored the C++ *self* argument in to the thread structure at a different time than the argument list, splitting the argument list. Fortunately, the QuickThreads rewrite can take advantage of a previous rewrite of PRESTO that eliminated the use of varargs for some platforms, and also the corresponding dependence on `nargs`. The QuickThreads port still uses the code that splits the self argument away from the other argument, but it then reassembles the argument list using `QT_VARGS`. Thus, although the current PRESTO always uses a fixed number of arguments (two), it must pay the penalty of using varargs generality.

PRESTO initializes a new thread by switching to it, then having it overwrite its own stack while the thread is running on the stack. In the QuickThreads rewrite, this crock was removed and the stack is instead initialized just before the thread runs. The PRESTO context switch code already contained a test to see if the new thread has been run before; stack initialization was just added to an existing list of initializations.

PRESTO uses scheduler threads, but the scheduler and the thread it is running communicate via globals. Thus, the QuickThreads blocking function return value feature is not needed for communication between threads and scheduler. In the current implementation, however, the return value is used when threads join. When a thread terminates, the thread function can return a value that is passed to a joining thread. The ideal place to save the value is in the thread's `cleanup` function. For obscure reasons, it was hard to call C++ code from C code, so the current implementation returns a `void *` to the scheduler. The scheduler tests the value, and if it is not a designated pointer, calls the appropriate cleanup routine. The current code is expedient but strictly wrong since several bit patterns (pointer, integral, etc.) can be returned and the designated pointer does not have a unique bit pattern. Using inter-language calls would eliminate this problem.

Porting PRESTO took substantial time for understanding the context switch code. The actual rewrite was a few hours. Once a QuickThreads version was available on the DECstation, PRESTO was ported to a SPARCstation with about half an hour of matching system and PRESTO header files to get library calls to work correctly. No thread initialization or context switch code was changed.

8.2. Porting NewThreads

NewThreads initializes threads with a function pointer and single argument. Threads always have stacks, so `QT_ARGS` was used directly. The `startup` function simply calls the user's function and calls a NewThreads cleanup routine if the user's function returns.

Although threads can block in various parts of the NewThreads library, NewThreads calls a single routine to perform all context switches. The central routine calls `QT_BLOCK` or `QT_BLOCKI` depending on whether a thread flag indicates floating-point is in use. Each node of the machine behaves like a uniprocessor, so no queue locks are needed, and threads can e.g., be put in the runable queue when they are still being used. Therefore, the helper function simply updates the thread state.

The original context switch code restores new threads differently than threads that are being restarted. Similarly, it tests explicitly whether floating-point registers need to be restored. QuickThreads threads are treated uniformly, removing several tests from the central routine. However, extra procedure calls are needed to call the QuickThreads primitives, so overall performance is the same.

The port took about 2 hours of relaxed coding.

9. Future Work

9.1. Compiler Support

Simple compiler support could improve raw QuickThreads performance substantially.

On most machines, much of the context switch time is spent saving and restoring registers. The compiler could be directed that calls to blocking routines should use a different calling convention with more caller-save registers. At a context switch, the caller would save only live registers and the register saves could also take advantage of instruction scheduling. Without compiler support, the core of the context switch routine is a burst of memory writes and reads.

Calls to `QT_ABORT` need not save registers. With compiler support, these calls could use a convention with more callee-save registers.

As discussed above, most of the context switch code is loads and stores; the remainder is typically 5-10 instructions and could also be inlined in the caller, saving one procedure call.

Helper functions are invoked from within the context switch code at a time when all general registers may be clobbered freely. However, helper functions are compiled to obey normal function calling conventions and will thus needlessly save callee-save registers before using them. With compiler support, they could instead use a calling convention with more caller-save registers.

Although blocking routines are called with many different helper functions, most individual call sites always pass the same helper. In these cases, even the helper could be inlined.

Thread startup can sometimes omit parts of the normal thread startup protocol. For example, a client interface may avoid startup and cleanup code and guarantee that user functions never return. In theses cases, the user function could be called directly, saving the overhead of at least one procedure call.

Although the above optimizations could improve raw context switch times, they all tend to make the code larger and reduce locality. Thus, they should only be used with threads that are extremely fine-grained. Coarser-grained threads will already be spending most of their time doing other work. Further, for blocking sites where most callee-save registers are in use, inlining the blocking operation will improve scheduling of loads and stores but will be unable to remove any of them. Finally, helper functions are often small enough that they fit in the available caller-save registers of existing protocols, so in practice they save and restore few callee-save registers.

9.2. Tuning

Although the QuickThreads interface is simple, it still has many possible implementations, each with tradeoffs. For example:

- `QT_BLOCK` can save floating-point registers then call `QT_BLOCKI`, or totally separate routines can be used. The first is compact, but the second saves a procedure call and, potentially, argument copying.
- `QT_ARGS` can ready a new thread to be resumed from the “middle” of `QT_BLOCKI`, or each (blocked) thread can store a pointer to the code that will resume it. In the first case, starting a new thread will restore all callee-save registers, even though only a few values may be needed to start a thread. In the second case, thread startup is fast because only the key values need to be read. However, context switches may be slower because each context switch requires an indirect branch.

When performance is unacceptable, a different implementation may solve the problem. However, the programmer should be aware that optimizations for one case may be pessimizations for another. Ideally, the client should be able to specify where performance is most critical and get the proper implementation [Kic92, Kep93]. Note, however, that some conventions are incompatible and a thread cannot, in general, be started with a different convention than the one that stopped it.

9.3. Preemption

It would be good to extend the QuickThreads interface to handle preemption. However, it is tricky to do so without hard-coding allocation policies. The basic problem is that preemption must always have a place to store the context of the preempted thread. There are various alternatives, none perfect:

- When preemption occurs a *preemption function* is called on the stack of the preempted thread. The preemption function starts another thread or resumes the preempted one. However, nested preemptions can lead to stack overflow.

- Scheduler activations [ABLL90] maintain a pool of empty stacks. A new thread (a “scheduler activation”) is created on each preemption. Nested preemptions can be organized in a way that cannot cause stack overflow, but nested preemptions can exhaust the free stack pool.
- The preemption function, and all code that it calls, can be recognized by e.g., the interrupt handler. The observation is that preempting a preemption handler is redundant, so one of the preemptions can be discarded. Nested preemptions either restart, throwing away previously-saved preemption state, or continue, throwing away state saved by the nested preemption. Since redundant preemption state is discarded, the space needs are bounded by the number of threads rather than the number of simultaneous preemptions.
- A final option is that the preemption mechanism is disabled until it is reenabled by the user-level code. To avoid stack growth problems, preemption must be reenabled only after the preemption handler has finished. This mechanism is thus unable to handle certain preemtions, such as preemptions arising from page faults.

Another problem with supporting preemption is that preemption interfaces typically depend on both the machine architecture and the operating system. Thus, a QuickThreads mechanism to support preemption must either be defined in terms of the OS preemption mechanism, or QuickThreads must define a machine-dependent interface, with the client is forced to perform the mapping.

A good implementation should avoid excess copying. Note that saved state (`struct sigcontext`) is often pushed on the thread stack during preemption.

A typical preemption interface would reconstruct the thread from its state at the time of preemption. The thread must be halted during reconstruction. If preemptions typically arrive on the stack of the preempted process, the preemption handler must transfer control to another thread before it reconstructs the preempted thread from the helper.

An interface to a preemption routine would take the context of a preempted thread and use that to prepare the thread for later restart. For example:

```
#include <signal.h>

qt_t *QT_PREEMT (struct sigcontext *preempted)
```

A final issue is that preemption during a context switch effectively preempts two threads: both the running thread and the thread that is about to be started or which was just halted. A variety of subtle race, deadlock, and lock ownership conditions can arise in the face of preemption [BMVL92].

10. Conclusions

A minimalist thread ADT gives a threads package core with: good performance, client-built functionality, and portability as good as the client that calls it. Using an ADT improves over full threads packages in two ways: First, threads packages have been notoriously machine-dependent. and an ADT can be used to encapsulate machine dependencies and make it possible to build threads packages that are both portable and efficient. Second, the ADT separates the notion of execution from scheduling and allocation; it is therefore possible to implement improved idioms (marker threads, faster barriers, etc.) that rely on particular behavior that is hidden by traditional threads packages.

11. Acknowledgements

Geoff Voelker implemented and debugged much of the KSR1 port, and Paul Barton-Davis helped solve problems with anomalous timing. Robert Bedichek helped with the 88000 port. Dylan McNamee rewrote NewThreads to use QuickThreads. Thanks to Tom Anderson for discussions on thread and lock implementations, and to Paul Barton-Davis and Curtis Brown for comments on earlier drafts of this paper.

This work supported by NSF CCR-8619663, CCR-8702915A01, CCR-8801806, CCR-8904190, PYI MIP-9058-439, by Boeing W280638, and by Sun Microsystems.

References

[And90]

Thomas E. Anderson, *FastThreads User's Manual*. In the *Quartz* distribution, available via anonymous ftp from '`ftp.cs.washington.edu`' in '`pub/Quartz1.0.tar.Z`'.

[ABLL91]

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*. Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles, October 1991.

[ALL89]

Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. *The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors*. IEEE Transactions on Computers 38(12), December 1989, pp. 1631-1644.

[Bar93]

Paul Barton-Davis. Personal communication, 1993.

[Ber88]

Brian N. Bershad. *The PRESTO User's Manual*. UWCSE TR 88-01-04, January 1988. University of Washington.

[BLL88]

Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. *PRESTO: A System for Object-Oriented Parallel Programming*, Software−Practice and Experience, 18(8), pp. 713-732.

[BLLW88]

Brian N. Bershad, Edward D. Lazowska Henry M. Levy, and David B. Wagner. *An Open Environment for Building Parallel Programming Systems*. UWCSE TR 88-01-03, January 1988. University of Washington.

[BMVL92]

Paul Barton-Davis, Dylan McNamee, Raj Vaswani and Edward D. Lazowska. *Adding Scheduler Activations to Mach 3.0*. UWCSE TR 92-08-03 October 1992. University of Washington.

[Bir89]Andrew D. Birrell. *An Introduction to Programming with Threads*. Digital Equipment Corporation, 1989.

[Cha90]

Jeff Chase. Personal Communication, 1990

[CLBL92]

Jeffrey S. Chase, Hemry M. Levy, Miche Baker-Harvy and Edward D. Lazowska. *Opal: A Single Address Space System for 64-Bit Architectures*, Third IEEE Workshop on Workstation Operating Systems (WWOS-III). April 1992.

[CD88]

Eric C. Cooper and Richard P. Draves. *C Threads*, Carnegie Mellon University Technical Report CMU-CS-88-154, June 1988.

[DEC81]

VAX Architecture. Digital Equipment Corporation, 1981.

[EAL93]

Dawson R. Engler, Gregory R. Andrews, and David K. Lowenthal. *Efficient Support for Fine-Grain Parallelism*. TR 93-13, University of Arizona Department of Computer Science.

[ELZ86]

Derek Eager, Edward Lazowska, and John Zahorjan. *Adaptive Load Sharing in Homogeneous Distributed Systems*. IEEE Transactions on Software Engineering, 12(5), May 1986, pp. 662-675.

[FM92]

Edward W. Felten and Dylan McNamee. *NewThreads 2.0 User's Guide*. August 1992, unpublished.

[Gru91]
Dirk Grunwald. Personal communication, 1991.

[Her88]
Maurice Herlihy. *Impossibility and Universality Results for Wait-Free Synchronization*. Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, 1988, pp. 276-291.

[Kan87]
Gerry Kane. *MIPS R2000 RISC Architecture*. Prentice Hall, 1987.

[Kep91]
David Keppel. *Register Windows and User-Space Threads on the SPARC*. UWCSE TR 91-07-01, August 1991. University of Washington.

[Kep93]
David Keppel. *Managing Abstraction-Induced Complexity*. Technical Report UWCSE 93-06-02, University of Washington, June 1993.

[Kic92]
Gregor Kiczales. *Towards a New Model of Abstraction in Software Engineering*. Proceedings of the International Workshop on Reflection and Meta-Level Architecture, November 1992, pp 1-11.

[Knu73]
Donald Knuth. *The Art of Computer Programming*, Volume 1: *Fundamental Algorithms*. 1973. Second edition. Addison Wesley, Reading, Mass.

[KSR91]
Principles of Operations, Kendall Square Research, Incorporated. 1991.

[MSLM91]
Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc and Evangelos P. Markatos. *First-Class User-Level Threads*. Proceedings of the 13th ACM Symposium on Operating Systems Principles. October 13-16 1991. Pacific Grove, California. pp. 110-121.

[MP88]
Henry Massalin and Calton Pu. *Fine-Grain Scheduling*. Technical Report CUCS-381-88, Columbia University, November 1988.

[MP89]
Henry Massalin and Calton Pu. *Threads and Input/Output in the Synthesis Kernel*. Proceedings of the 12th ACM Symposium on Operating Systems Principles, December 1989, pp. 191-201.

[Mot89]
MC88100 RISC Microprocessor User's Manual, Motorola Corporation, 1989.

[MVZ90]
Cathy McCann, Raj Vaswani, and John Zahorjan. *A Dynamic Processor Allocation Policy for Multiprogrammed, Shared Memory Multiprocessors*. UWCSE TR 90-03-02, February 1991. University of Washington.

[Pin93]
Brian Pinkerton. Personal communication, February 1993.

[PKB+91]
Mike L. Powell, S. R. Kleiman, S. Barton, D. Shan, D. Sten, M. Weeks, *SunOS Multi-Thread Architecture*. Appears in *The SPARC Technical Papers*, Ben J. Catanzaro, Editor. Springer-Verlag, 1991. pp. 229-372.

[Pra86]
Terence W. Pratt. *Programming Language Design and Implementation*. May 1986. Second edition. Prentice-Hall, Englewood Cliffs, New Jersey.

[Ros92]
O. Rose. Personal Communication, April 1992.

[Seq88]
Symmetry Technical Summary, Sequent Computer Systems, Inc., 1988.

[Sit93]
Richard Sites. *Alpha AXP Architecture*. Communications of the ACM (CACM) 36(2), 1993.

[SPA92]
SPARC International. *The SPARC Architecture Manual*. Prentice-Hall, Englewood Cliffs, New Jersey 07632. ISBN 0-13-825001-4.

[Sta92]
Richard M. Stallman. *Using and Porting GNU CC*. The Free Software Foundation, 1992.

[Sun88]
SunOS Reference Manual, Sun Microsystems. Part Number 800-1751-10, May 1988.

[TSS88]
Charles M. Thacker, Lawrance Stewart, and Edward Satterthwaite, Jr. *Firefly: A Multiprocessor Workstation*. IEEE Transactions on Computers, 37(8):909-920, August 1988.

[Tho68]
Ken Thompson. *Regular Expression Search Algorithm*. Communications of the Association for Computing Machinery (CACM), 11(6):419-422, June 1968.

Appendix A Variant Argument Lists

Thread creation functions that accept variant argument lists are slower and less portable than those that take fixed argument lists. They are slower because the argument list must be copied and the size is not known at compile time; if the same arguments are passed in a structure, the compiler can produce optimized code for the job. The portability problems come from a number of places and, unfortunately, cause reliability problems, not just worse performance.

Some C libraries provide a function, `nargs`, that returns the number of words of arguments. In general, however, it is impossible to implement `nargs` without special support. On the VAX, arguments are popped by the callee, so the number of words must be passed to the callee, and, is available for use by `nargs`. The Sequent Symmetry also provides `nargs`, but relies on the caller to pop arguments from the stack immediately after the call and in certain stylized ways. Today, it is more common for the caller to allocate and free argument space at procedure entry, though some optimizing compilers have flags to force argument popping immediately after procedure return. For example, programs compiled with GNU CC [Sta92] can use the `-fno-defer-pop` compiler flag. The KSR implements `nargs` with a callback from the callee to a code fragment in the caller and the caller embeds the `nargs` value in an instruction immediate [KSR91]. In a threads package, the caller is the thread startup code, so that code must contain the needed code fragment. Thus, a threads package implementation of `nargs` requires one startup routine for each possible `nargs` value or dynamic compilation to create the needed startup routines on demand.

Some thread libraries have thread creation routines that require the user to pass an argument that is the number of words of arguments that should be passed to the thread. The first problem is solved, because `nargs` is not needed. However, the argument list may be larger than the sum of the sizes of the arguments, so the user of the thread library must know the details of the calling convention, which varies from machine to machine and sometimes from compiler to compiler.

The SunOS `lwp` package simplifies matters by allowing only integer and pointer arguments and requiring the end user to pass in the number of such arguments [Sun88, PKB+91]. However, problems can still arise with architectures where pointers and integers are different sizes (e.g., DEC AXP [Sit93] and KSR1 [KSR91]).

Varargs functions and fixed-argument functions may use different calling conventions. Thus, a function that gets passed parameters with varargs is only supposed to access them with varargs. Thus, calling `printf` as a thread routine is safe because `printf` is a varargs function. Using `puts` as a thread routine, however, may break since `puts` does not access its arguments with varargs. Although the differences are rarely important on current systems, future compilers that perform better optimization may be incompatible with mixed use.

Alignment restrictions may make it impossible to use varargs reliably. The Motorola 88000 family calling convention double-aligns the argument area and all double-sized parameters. For varargs under QuickThreads, the argument list is copied from the caller's stack to the callee's stack. In general, the client routine that calls `QT_VARGS` has both varargs parameters to pass to the user's routine, and non-varargs parameters, such as a pointer the user routine to call on thread startup. The non-varargs parameters are stripped (do not get passed to the user's routines) which shifts the varargs alignment. The number of words of shifting depends on the number of non-varargs arguments that are passed to the routine that calls `va_start`. If there are an odd number of words of non-varargs parameters, the varargs argument list is shifted by an odd number of words, leaving it unaligned in memory. The threads package lacks sufficient type information to realign the parameters when they are shifted. The threads package likewise lacks control over the user routine's use of varargs in the callee, so it is not possible to realign the parameters when they are used.

Appendix B Restarting A Thread

In some circumstances it is useful to block a thread tentatively, then restart it if conditions have changed. The basic idea is to test a condition; if the condition is true, select a next thread and block the old thread. In the old thread's helper, retest the condition: if it is still true, finish blocking the thread. But if the condition has gone false, the old thread is restarted and what was to have been the next thread is returned to the runable queue.

In Figure 10a, the routine `tentative_block` suspends the current thread and runs the helper routine `tentative_help1` on the stack of a new thread. Three arguments are passed to the helper: the thread to block, the queue to block on, and the next thread (which might be returned to the runable queue). Since QuickThreads passes only two arguments, three arguments are passed by passing the helper a pointer to a structure with two arguments.

```
tentative_block()
{
  arg_t self;

  next = qget (&runableq);
  self.thread = global_curr_running;
  self.blockon = &runableq;
  QT_BLOCK (tentative_help1, &self, next, next->sp);
  /* Woke up again. */
}
```

Figure 10a. Blocking tentatively.

Figure 10b shows a helper function that tests a condition, and, if true, finishes blocking the old thread. (The test and block must be atomic; for clarity, synchronization operations are omitted.) If the condition is false, it restarts the old thread by calling another helper function on the old thread's stack.

Figure 10c shows a second helper function that puts the next thread back on the runable queue and returns. Although the next thread's stack was used, the next thread was never actually run. Thus, it can be re-queued on the runable queue and it will be ready to run, just as it was when it was pulled from the queue by `tentative_block`. Note that the next thread's original stack pointer (that used by `tentative_block`) is used, and not the (garbage) value that results from calling a blocking routine in the helper function.[10]

---

10. Note to QuickThreads implementors: `QT_ABORT` may walk the call chain. The implementation must arrange that walks started in helpers or their descendants will walk only the helper function's call chain, and not that of the whole stack.

```
tentative_help1 (oldsp, oldself, next)
{
  if (condition) {
    oldself->thread->sp = oldsp;
    qput (oldself->blockon, oldself->thread);
  } else {
    QT_ABORT (tentative_help2, NULL, next, oldsp);
  }
}
```

Figure 10b. A helper to either finish blocking or restart.

```
tentative_help2 (unused1, unused2, next)
{
  qput (&runableq, next);
}
```

Figure 10c. A helper called from a helper to restart a thread.

The second helper returns the new thread to the runable queue. When the second helper returns, it restarts `tentative_block` and allows the old thread to continue runing.