Thread-Specific Storage for C/C++

An Object Behavioral Pattern for Accessing per-Thread State Efficiently

Douglas C. Schmidt
Timothy H. Harrison
{schmidt, harrison}@cs.wustl.edu
Dept. of Computer Science
Washington University, St. Louis[1]
St. Louis, MO, USA

Nat Pryce
np2@scorch.doc.ic.ac.uk
Department of Computing
Imperial College[2]
London, UK

This paper will appear at the 4^{th} annual Pattern Languages of Programming conference held in Allerton Park, Illinois, September, 1997. Permission is granted to make copies of this paper for the PLoP '97 conference proceedings.

Abstract

In theory, multi-threading an application can improve performance (by executing multiple instruction streams simultaneously) and simplify program structure (by allowing each thread to execute synchronously rather than reactively or asynchronously). In practice, multi-threaded applications often perform no better, or even worse, than single-threaded applications due to the overhead of acquiring and releasing locks. In addition, multi-threaded applications are hard to program due to the complex concurrency control protocols required to avoid race conditions and deadlocks.

This paper describes the Thread-Specific Storage pattern, which alleviates several problems with multi-threading performance and programming complexity. The Thread-Specific Storage pattern improves performance and simplifies multi-threaded applications by allowing multiple threads to use one logically global access point to retrieve thread-specific data without incurring locking overhead for each access.

1 Intent

Allows multiple threads to use one logically global access point to retrieve thread-specific data without incurring locking overhead for each access.

2 Motivation

2.1 Context and Forces

The Thread-Specific Storage pattern should be applied to multi-threaded applications that frequently access objects that are *logically* global but *physically* specific to each thread. For instance, operating systems like UNIX and Win32 report error information to applications using `errno`. When an error occurs in a system call, the OS sets `errno` to report the problem and returns a documented failure status. When the application detects the failure status it checks `errno` to determine what type of error occurred.

For instance, consider the following typical C code fragment that receives buffers from a non-blocking TCP socket:

```
// One global errno per-process.
extern int errno;

void *worker (SOCKET socket)
{
  // Read from the network connection
  // and process the data until the connection
  // is closed.

  for (;;) {
    char buffer[BUFSIZ];
    int result = recv (socket, buffer, BUFSIZ, 0);

    // Check to see if the recv() call failed.
    if (result == -1) {
      if (errno != EWOULDBLOCK)
        // Record error result in thread-specific data.
        printf ("recv failed, errno = %d", errno);
    } else
      // Perform the work on success.
      process_buffer (buffer);
  }
}
```

If `recv` returns -1 the code checks that `errno != EWOULDBLOCK` and prints an error message if this is not the case (*e.g.*, if `errno == EINTR`), otherwise it processes the buffer it receives.

2.2 Common Traps and Pitfalls

Although the "global error variable" approach shown above works reasonably[3] well for single-threaded applications, subtle problems occur in multi-threaded applications. In particular, race conditions in preemptively multi-threaded systems can cause an `errno` value set by a method in one thread to be interpreted erroneously by applications in other

[1]This research is supported in part by a grant from Siemens AG.
[2]This research is funded by British Telecom, plc.

[3]The Appendix discusses the tradeoffs of reporting errors using alternative techniques (such as exceptions and passing an explicit error parameter to each call).

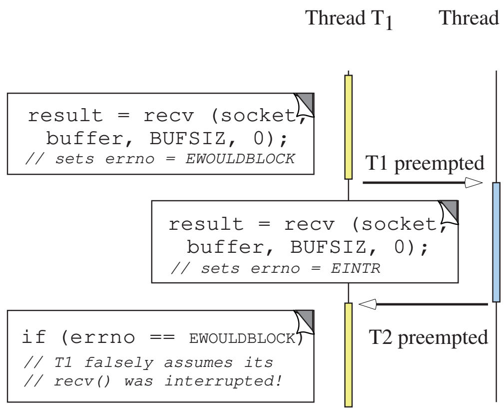

Figure 1: Race Conditions in Multi-threaded Programs

threads. Thus, if multiple threads execute the `worker` function simultaneously it is possible that the global version of `errno` will be set incorrectly due to race conditions.

For example, two threads (T_1 and T_2) can perform `recv` calls on the `socket` in Figure 1. In this example, T_1's `recv` returns -1 and sets `errno` to `EWOULDBLOCK`, which indicates no data is queued on the socket at the moment. Before it can check for this case, however, the T_1 thread is preempted and T_2 runs. Assuming T_2 gets interrupted, it sets `errno` to `EINTR`. If T_2 is then preempted immediately, T_1 will falsely assume its `recv` call was interrupted and perform the wrong action. Thus, this program is erroneous and non-portable since its behavior depends on the order in which the threads execute.

The underlying problem here is that setting and testing the global `errno` value occurs in two steps: (1) the `recv` call sets the value and (2) the application tests the value. Therefore, the "obvious" solution of wrapping a mutex around `errno` will not solve the race condition because the set/test involves multiple operations (*i.e.*, it is not atomic).

One way to solve this problem is to create a more sophisticated locking protocol. For instance, the `recv` call could acquire an `errno_mutex` internally, which must be released by the application once the value of `errno` is tested after `recv` returns. However, this solution is undesirable since applications can forget to release the lock, thereby causing starvation and deadlock. Moreover, if applications must check the error status after every library call the additional locking overhead will degrade performance significantly, even when multiple threads are not used.

2.3 Solution: Thread-Specific Storage

A common solution to the traps and pitfalls described above is to use the *Thread-Specific Storage* pattern. This pattern resolves the following forces:

• **Efficiency:** Thread-specific storage allows sequential methods within a thread to access thread-specific objects atomically without incurring locking overhead for each access.

• **Simplify application programming:** Thread-specific storage is simple for application programmers to use because system developers can make the use of thread-specific storage completely transparent at the source-code level via data abstraction or macros.

• **Highly portable:** Thread-specific storage is available on most multi-threaded OS platforms and can be implemented conveniently on platforms (such as VxWorks) that lack it.

Therefore, regardless of whether an application runs in a single thread or multiple threads, there should be no additional overhead incurred and no changes to the code required to use the Thread-Specific Storage pattern. For example, the following code illustrates how `errno` is defined on Solaris 2.x:

```
// A thread-specific errno definition (typically
// defined in <sys/errno.h>).
#if defined (_REENTRANT)
// The _errno() function returns the
// thread-specific value of errno.

#define errno (*_errno())
#else
// Non-MT behavior is unchanged.
extern int errno;
#endif /* REENTRANT */

void *worker (SOCKET socket)
{
  // Exactly the same implementation shown above.
}
```

When the `_REENTRANT` flag is enabled the `errno` symbol is defined as a macro that invokes a helper function called `_errno`, which returns a pointer to the thread-specific value of `errno`. This pointer is dereferenced by the macro so that it can appear on either the left or right side of an assignment operator.

3 Applicability

Use the Thread-Specific Storage pattern when an application has the following characteristics:

- It was originally written assuming a single thread of control and is being ported to a multi-threaded environment *without* changing existing APIs; or
- It contains multiple preemptive threads of control that can execute concurrently in an arbitrary scheduling order, and
- Each thread of control invokes sequences of methods that share data common only to that thread, and
- The data shared by objects within each thread must be accessed through a globally visible access point that is "logically" shared with other threads, but "physically" unique for each thread; and

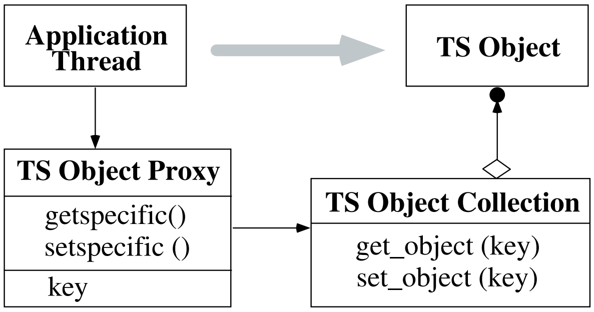

Figure 2: Structure of Participants in the Thread-Specific Storage Pattern

- The data is passed implicitly between methods rather than being passed explicitly via parameters.[4]

Understanding the characteristics described above is crucial to using (or not using) the Thread-Specific Storage pattern. For example, the UNIX `errno` variable is an example of data that is (1) logically global, but physically thread-specific and (2) passed implicitly between methods.

Do *not* use the Thread-Specific Storage pattern when an application has the following characteristics:

- Multiple threads are collaborating on a single task that requires concurrent access to shared data. For instance, a multi-threaded application may perform reads and writes concurrently on an in-memory database. In this case, threads must share records and tables that are not thread-specific. If thread-specific storage was used to store the database, the threads could not share the data. Thus, access to the database records must be controlled with synchronization primitives (*e.g.*, mutexes) so that the threads can collaborate on the shared data.
- It is more intuitive and efficient to maintain both a physical *and* logical separation of data. For instance, it may be possible to have threads access data visible only within each thread by passing the data explicitly as parameters to all methods. In this case, the Thread-Specific Storage pattern may be unnecessary.

4 Structure and Participants

Figure 2 illustrates the structure of the following participants in the Thread-Specific Storage pattern:

Application Threads

- Application threads use `TS Object Proxies` to access `TS Objects` residing in thread-specific storage. As shown in Section 9, an implementation of the Thread-Specific Storage pattern can use *smart pointers* to hide the `TS Object Proxy` so that applications appear to access the `TS Object` directly.

[4] This situation is common when porting single-threaded APIs to multi-threaded systems.

Thread-Specific (TS) Object Proxy (`errno` macro)

- The `TS Object Proxy` defines the interface of a `TS Object`. It is responsible for providing access to a unique object for each application thread via the `getspecific` and `setspecific` methods. For instance, in the error handling example from Section 2, the `errno` `TS Object` is an `int`.

  A `TS Object Proxy` instance is responsible for a type of object, *i.e.*, it mediates access to a thread-specific `TS Object` for every thread that accesses the proxy. For example, multiple threads may use the same `TS Object Proxy` to access thread-specific `errno` values. The `key` value stored by the proxy is assigned by the `TS Object Collection` when the proxy is created and is passed to the collection by the `getspecific` and `setspecific` methods.

  The purpose of `TS Object Proxies` is to hide `keys` and `TS Object Collections`. Without the proxies, the `Application Threads` would have to obtain the collections and use `keys` explicitly. As shown in Section 9, most of the details of thread-specific storage can be completely hidden via smart pointers for the `TS Object Proxy`.

Thread-Specific (TS) Object (`*_errno()` value)

- A `TS Object` is a particular thread's instance of a thread-specific object. For instance, a thread-specific `errno` is an object of type `int`. It is managed by the `TS Object Collection` and accessed only through a `TS Object Proxy`.

Thread-Specific (TS) Object Collection

- In complex multi-threaded applications, a thread's `errno` value may be one of many types of data residing in thread-specific storage. Thus, for a thread to retrieve its thread-specific error data it must use a `key`. This `key` must be associated with `errno` to allow a thread to access the correct entry in the `TS Object Collection`.

  The `TS Object Collection` contains a set of all thread-specific objects associated with a particular thread, *i.e.*, every thread has a unique `TS Object Collection`. The `TS Object Collection` maps `keys` to thread-specific `TS Objects`. A `TS Object Proxy` uses the `key` to retrieve a specific `TS Object` from the `TS Object Collection` via the `get_object(key)` and `set_object(key)` methods.

5 Collaborations

The interaction diagram in Figure 3 illustrates the following collaborations between participants in the Thread-Specific Storage pattern:

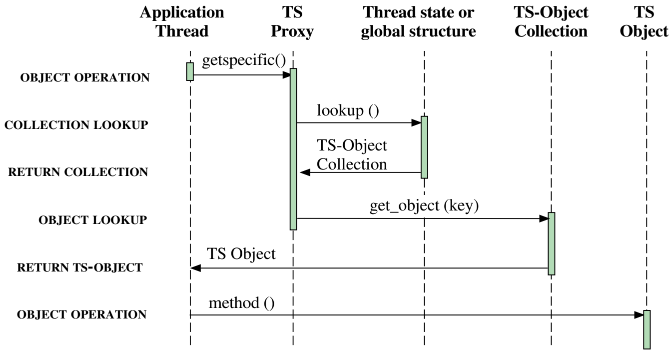

Figure 3: Interactions Among Participants in the Thread-Specific Storage Pattern

• **Locate the TS Object Collection:** Methods in each `Application Thread` invoke the `getspecific` and `setspecific` methods on the `TS Object Proxy` to access the `TS Object Collection`, which is stored inside the thread or in a global structure indexed by the thread ID.[5]

• **Acquire the TS Object from thread-specific storage:** Once the `TS Object Collection` has been located, the `TS Object Proxy` uses its `key` to retrieve the correct `TS Object` from the collection.

• **Set/get TS Object state:** At this point, the application thread operates on the `TS Object` using ordinary C++ method calls. No locking is necessary since the object is referenced by a pointer that is accessed only within the calling thread.

6 Consequences

6.1 Benefits

There are several benefits of using the Thread-Specific Storage pattern, including:

Efficiency: The Thread-Specific Storage pattern can be implemented so that no locking is needed to thread-specific data. For instance, by placing `errno` into thread-specific storage, each thread can reliably set and test the completion status of methods within that thread without using complex synchronization protocols. This eliminates locking overhead for data shared within a thread, which is faster than acquiring and releasing a mutex [1].

Ease of use: Thread-specific storage is simple for application programmers to use because system developers can make the use of thread-specific storage completely transparent at the source-code level via data abstraction or macros.

[5]Every thread in a process contains a unique identifying value called a "thread ID," which is similar to the notion of a process ID.

6.2 Liabilities

There are also the following liabilities to using the Thread-Specific Storage pattern:

It encourages the use of (thread-safe) global variables: Many applications do not require multiple threads to access thread-specific data via a common access point. When this is the case, the data should be stored so that only the thread owning the data can access it. For example, consider a network server that uses a pool of worker threads to handle incoming requests from clients. These threads may log the number and type of services performed. This logging mechanism could be accessed as a global `Logger` object utilizing Thread-Specific Storage. A simpler approach, however, would represent each worker thread as an Active Object [2] with an instance of the `Logger` stored internally. In this case, no overhead is required to access the `Logger`, as long as it is passed as a parameter to all functions in the Active Object.

It hides the structure of the system: The use of thread-specific storage hides the relationships between objects in an application, potentially making the application harder to understand. Explicitly representing relationships between objects can eliminate the need for thread-specific storage in some cases, as described in Appendix A.2.

7 Implementation

The Thread-Specific Storage pattern can be implemented in various ways. This section explains each step required to implement the pattern. The steps are summarized as follows:

1. Implement the TS Object Collections: If the OS does not provide an implementation of thread-specific storage, it can be implemented using whatever mechanisms are available to maintain the consistency of the data structures in the `TS Object Collections`.

2. Encapsulate details of thread-specific storage: As shown in Section 8, interfaces to thread-specific storage are typically weakly-typed and error-prone. Thus, once an implementation of thread-specific storage is available, use C++ programming language features (such as templates and overloading) to hide the low-level details of thread-specific storage behind OO APIs.

The remainder of this section describes how to implement the low-level thread-specific storage APIs. Section 8 provides complete sample code and Section 9 examines several ways to encapsulate low-level thread-specific storage APIs with C++ wrappers.

7.1 Implement the TS Object Collections

The `TS Object Collection` shown in Figure 2 contains all `TS Objects` belonging to a particular thread. This collection can be implemented using a table of pointers to

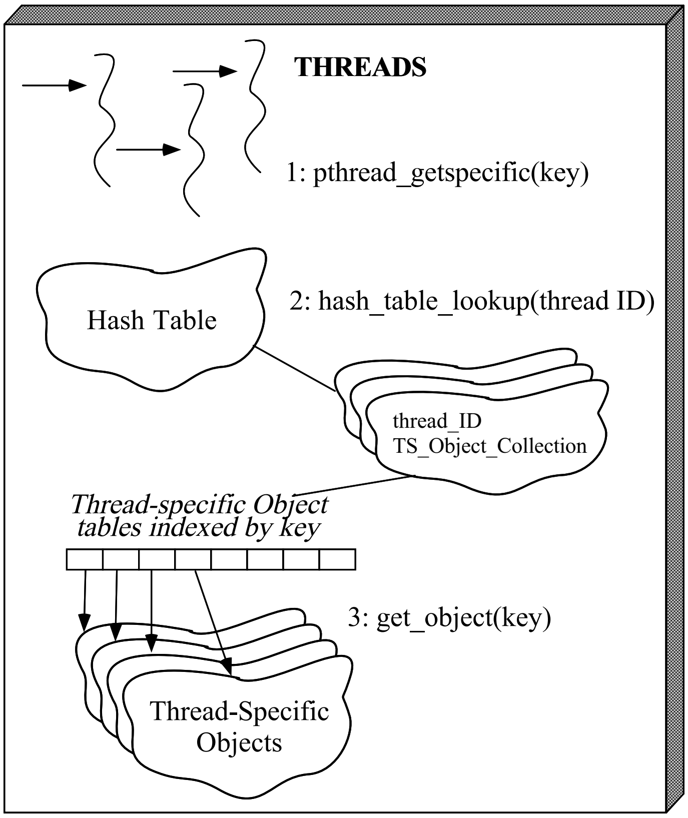

Figure 4: External Implementation of Thread-Specific Storage

`TS Objects` indexed by `keys`. A thread must locate its `TS Object Collection` before accessing thread-specific objects by their `keys`. Therefore, the first design challenge is determining how to locate and store `TS Object Collections`.

`TS Object Collections` can be stored either (1) externally to all threads or (2) internally to each thread. Each approach is described and evaluated below:

1. External to all threads: This approach defines a global mapping of each thread's ID to its `TS Object Collection` table (shown in Figure 4). Locating the right collection may require the use of a readers/writer lock to prevent race conditions. Once the collection is located, however, no additional locking is required since only one thread can be active within a `TS Object Collection`.

2. Internal to each thread: This approach requires each thread in a process to store a `TS Object Collection` with its other internal state (such as a run-time thread stack, program counter, general-purpose registers, and thread ID). When a thread accesses a thread-specific object, the object is retrieved by using the corresponding `key` as an index into the thread's internal `TS Object Collection` (shown in Figure 5). This approach requires no additional locking.

Choosing between the external and internal implementa-

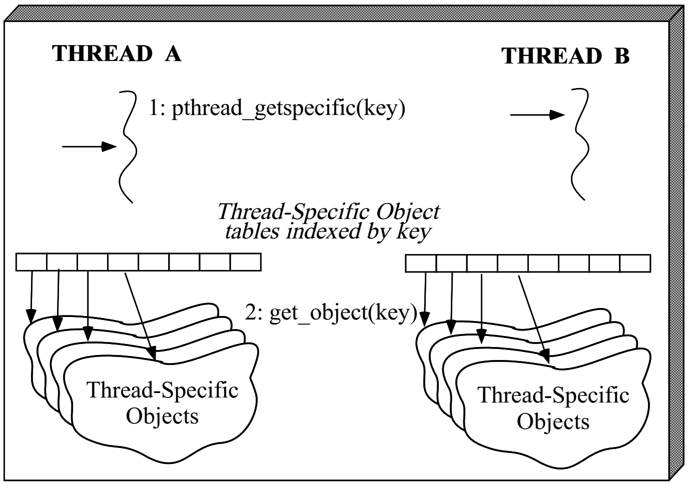

Figure 5: Internal Implementation of Thread-Specific Storage

tion schemes requires developers to resolve the following tradeoffs:

Fixed- vs. variable-sized TS Object Collections: For both the external and internal implementations, the `TS Object Collection` can be stored as a fixed-size array if the range of thread-specific keys is relatively small. For instance, the POSIX Pthread standard defines a minimum number of keys, `_POSIX_THREAD_KEYS_MAX`, that must be supported by conforming implementations. If the size is fixed (*e.g.*, to 128 keys, which is the POSIX default), the lookup time can be $O(1)$ by simply indexing into the `TS Object Collection` array using the object's key, as shown in Figure 5.

The range of thread-specific keys can be large, however. For instance, Solaris threads have no predefined limit on the number of keys. Therefore, Solaris uses a variable-sized data structure, which can increase the time required to manage the `TS Object Collection`.

Fixed- vs. variable-sized mapping of thread IDs to TS Object Collections: Thread IDs can range from very small to very large values. This presents no problem for internal implementations since the thread ID is implicitly associated with the corresponding `TS Object Collection` contained in the thread's state.

For external implementations, however, it may be impractical to have a fixed-size array with an entry for every possible thread ID value. Instead, it is more space efficient to have threads use a dynamic data structure to map thread IDs to `TS Object Collections`. For instance, one approach is to use a hash function on the thread ID to obtain an offset into a hash table bucket containing a chain of tuples that map thread IDs to their corresponding `TS Object Collection` (as shown in Figure 4).

Global vs. local TS Object Collections: The internal approach stores the `TS Object Collections` *locally* with the thread, whereas the external approach stores them *globally*. Depending on the implementation of the external table, the global location can allow threads to access other threads' `TS Object Collections`. Although this seems to defeat the whole point of *thread-specific* storage, it is useful if the thread-specific storage implementation provides automatic garbage collection by recycling unused keys. This feature is particularly important on implementations that limit the number of keys to a small value (*e.g.*, Windows NT has a limit of 64 keys per process).

However, using an external table increases the access time for every thread-specific object since synchronization mechanisms (such as readers/writer locks) are required to avoid race conditions if globally accessible table is modified (*e.g.*, when creating new keys). On the other hand, keeping the `TS Object Collection` locally in the state of each thread requires more storage per-thread, though no less *total* memory consumption.

8 Sample Code

8.1 Implementing the POSIX Pthreads Thread-Specific Storage API

The following code shows how thread-specific storage can be implemented when `TS Objects` are stored "internally" to each thread using a fixed-sized array of keys. This example is adapted from a publically available implementation [3] of POSIX Pthreads [4].

The `thread_state` structure shown below contains the state of a thread:

```
struct thread_state
{
  // The thread-specific error number.
  int errno_;

  // Thread-specific data values.
  void *key_[_POSIX_THREAD_KEYS_MAX];

  // ... Other thread state.
};
```

In addition to `errno` and the array of thread-specific storage pointers, this structure also includes a pointer to the thread's stack and space to store data (*e.g.*, the program counter) that is saved/restored during a context switch.

For a particular thread-specific object, the same key value is used to set and get thread-specific values for all threads. For instance, if `Logger` objects are being registered to keep track of thread-specific logging attributes, the thread-specific `Logger` proxy will be assigned some key value N. All threads will use this value N to access their thread-specific logging object. A count of the total number of keys currently in use can be stored globally to all threads, as follows:

```
typedef int pthread_key_t;

// All threads share the same key counter.
static pthread_key_t total_keys_ = 0;
```

The `total_keys_` count is automatically incremented every time a new thread-specific key is required, as shown in the `pthread_key_create` function below:

```
// Create a new global key and specify
// a "destructor" function callback.
int
pthread_key_create (pthread_key_t *key,
                    void (*thread_exit_hook) (void *))
{
  if (total_keys_ >= _POSIX_THREAD_KEYS_MAX) {
    // pthread_self() refers to the context of the
    // currently active thread.
    pthread_self ()->errno_ = ENOMEM;
    return -1;
  }

  thread_exit_hook_[total_keys_] = thread_exit_hook;
  *key = total_keys_++;
  return 0;
}
```

The `pthread_key_create` function allocates a new key value that uniquely identifies a thread-specific data object. In addition, it allows an application to associate a `thread_exit_hook` with a key. This hook is a pointer to a function that is called automatically when (1) a thread exits and (2) there is a thread-specific object registered for a key. An array of function pointers to "thread exit hooks" can be stored globally, as follows:

```
// Exit hooks to cleanup thread-specific keys.
static void
(*thread_exit_hook_[_POSIX_THREAD_KEYS_MAX]) (void);
```

The `pthread_exit` function below shows how thread exit hook functions are called in the implementation of `pthread_exit`:

```
// Terminate the thread and call thread exit hooks.
void pthread_exit (void *status)
{
  // ...
  for (i = 0; i < total_keys; i++)
    if (pthread_self ()->key_[i]
        && thread_exit_hook_[i])
      // Indirect pointer to function call.
      (*thread_exit_hook_[i])
        (pthread_self ()->key_[i]);
  // ...
}
```

Applications can register different functions for each thread-specific data object, but for each object the same function is called for each thread. Registering dynamically allocated thread-specific objects is a common use-case. Therefore, thread exit hooks typically look like the following:

```
static void
cleanup_tss_Logger (void *ptr)
{
  // This cast is necessary to invoke
  // the destructor (if it exists).
  delete (Logger *) ptr;
}
```

This function deallocates a dynamically allocated `Logger` object.

The `pthread_setspecific` function binds a `value` to the given `key` for the calling thread:

```
// Associate a value with a data key
// for the calling thread.
int pthread_setspecific (int key,
                         void *value)
{
  if (key < 0 || key >= total_keys) {
    pthread_self ()->errno_ = EINVAL;
    return -1;
  }

  pthread_self ()->key_[key] = value;
  return 0;
}
```

Likewise, pthread_getspecific stores into value the data bound to the given key for the calling thread:

```
// Retrieve a value from a data key
// for the calling thread.
int pthread_getspecific (int key,
                         void **value)
{
  if (key < 0 || key >= total_keys) {
    pthread_self ()->errno_ = EINVAL;
    return -1;
  }

  *value = pthread_self ()->key_[key];
  return 0;
}
```

Because data are stored internally in the state of each thread, neither of these functions requires any additional locks to access thread-specific data.

8.2 Using Thread-Specific Storage in Applications

The example below illustrates how to use the thread-specific storage APIs from the POSIX Pthread specification in a C function that can be called from more than one thread *without* having to call an initialization function explicitly:

```
// Local to the implementation.
static pthread_mutex_t keylock =
  PTHREAD_MUTEX_INITIALIZER;
static pthread_key_t key;
static int once = 0;

void *func (void)
{
  void *ptr = 0;

  // Use the Double-Checked Locking pattern
  // (described further below) to serialize
  // key creation without forcing each access
  // to be locked.

  if (once == 0) {
    pthread_mutex_lock (&keylock);
    if (once == 0) {
      // Register the free(3C) function
      // to deallocation TSS memory when
      // the thread goes out of scope.
      pthread_key_create (&key, free);
      once = 1;
    }
    pthread_mutex_unlock (&keylock);
  }

  pthread_getspecific (key, (void **) &ptr);

  if (ptr == 0) {
     ptr = malloc (SIZE);
     pthread_setspecific (key, ptr);
  }

  return ptr;
}
```

8.3 Evaluation

The solution above directly invokes the thread-specific library functions (such as pthread_getspecific and pthread_setspecific) in application code. However, these APIs, which are written in C, have the following limitations:

• **Non-portable:** The interfaces of POSIX Pthreads, Solaris threads, and Win32 threads are very similar. However, the semantics of Win32 threads are subtly different since they do not provide a reliable means of cleaning up objects allocated in thread-specific storage when a thread exits. Moreover, there is no API to delete a key in Solaris threads. This makes it hard to write portable code among UNIX and Win32 platforms.

• **Hard to use:** Even with error checking omitted, the locking operations shown by the func example in Section 8.2 are complex and non-intuitive. This code is a C implementation of the Double-Checked Locking pattern [5]. It's instructive to compare this C implementation to the C++ version in Section 9.2.1 to observe the greater simplicity, clarity, and type-safety resulting from the use of C++ wrappers.

• **Non-type-safe:** The POSIX Pthreads, Solaris, and Win32 thread-specific storage interfaces store pointers to thread-specific objects as void *'s. Although this approach is flexible, it's easy to make mistakes since void *'s eliminate type-safety.

9 Variations

Section 8 demonstrated how to implement and use the Thread-Specific Storage pattern via POSIX pthread interfaces. However, the resulting solution was non-portable, hard to use, and not type-safe. To overcome these limitations, additional classes and C++ wrappers can be developed to program thread-specific storage robustly in a type-safe manner.

This section illustrates how to encapsulate low-level thread-specific storage mechanisms provided by Solaris threads, POSIX Pthreads, or Win32 threads using C++ wrappers. Section 9.1 describes how to encapsulate the POSIX Pthread library interfaces with hard-coded C++ wrappers and Section 9.2 describes a more general solution using C++ template wrappers. The example used for each alternative approach is a variant of the Logger abstraction described in Section 6.2.

9.1 Hard-coded C++ Wrapper

One way to make all instances of a class be thread-specific is to use thread-specific library routines directly. The steps required to implement this approach are described below. Error checking has been minimized to save space.

9.1.1 Define the Thread-Specific State Information

The first step is to determine the object's state information that must be stored or retrieved in thread-specific storage. For instance, a `Logger` might have the following state:

```
class Logger_State
{
public:
  int errno_;
  // Error number.

  int line_num_;
  // Line where the error occurred.

  // ...
};
```

Each thread will have its own copy of this state information.

9.1.2 Define an External Class Interface

The next step is to define an external class interface that is used by all application threads. The external class interface of the `Logger` below looks just like an ordinary non-thread-specific C++ class:

```
class Logger
{
public:
  // Set/get the error number.
  int errno (void);
  void errno (int);

  // Set/get the line number.
  int line_num (void);
  void line_num (int);

  // ...
};
```

9.1.3 Define a Thread-Specific Helper Method

This step uses the thread-specific storage functions provided by the thread library to define a helper method that returns a pointer to the appropriate thread-specific storage. Typically, this helper method performs the following steps:

1. Key initialization: Initialize a key for each thread-specific object and use this key to get/set a thread-specific pointer to dynamically allocated memory containing an instance of the internal structure. The code could be implemented as follows:

```
class Logger
{
public:
  // ... Same as above ...

protected:
  Logger_State *get_tss_state (void);

  // Key for the thread-specific error data.
  pthread_key_t key_;

  // "First time in" flag.
  int once_;
};

Logger_State *Logger::get_tss_state (void)
{
  // Check to see if this is the first time in
  // and if so, allocate the key (this code
  // doesn't protect against multi-threaded
  // race conditions...).
  if (once_ == 0) {
    pthread_key_create (this->key_, free);
    once_ = 1;
  }

  Logger_State *state_ptr;

  // Get the state data from thread-specific
  // storage.  Note that no locks are required...
  pthread_getspecific (this->key_,
                       (void **) &state_ptr);

  if (state_ptr == 0) {
    state_ptr = new Logger_State;
    pthread_setspecific (this->key_,
                         (void *) state_ptr);
  }

  // Return the pointer to thread-specific storage.
  return state_ptr;
};
```

2. Obtain a pointer to the thread-specific object: Every method in the external interface will call the `get_tss_state` helper method to obtain a pointer to the `Logger_State` object that resides in thread-specific storage, as follows:

```
int Logger::errno (void)
{
  return this->get_tss_state ()->errno_;
}
```

3. Perform normal operations: Once the external interface method has the pointer, the application can perform operations on the thread-specific object as if it were an ordinary (*i.e.*, non-thread-specific) C++ object:

```
Logger logger;

int
recv_msg (HANDLE socket, char *buffer,
          size_t bufsiz)
{
  if (recv (socket, buffer, bufsiz, 0) == -1) {
    logger->errno () = errno;
    return -1;
  }
  // ...
}
```

```
int main (void)
{
  // ...
  if (recv_msg (socket, buffer, BUFSIZ) == -1
      && logger->errno () == EWOULDBLOCK)
    // ...
};
```

9.1.4 Evaluation of the Hard-coded Wrapper

The advantage of using a hard-coded wrapper is that it shields applications from the knowledge of the thread-specific library functions. The disadvantage of this approach is that it does not promote reusability, portability, or flexibility. In particular, for every thread-specific class, the developer needs to reimplement the thread-specific helper method within the class.

Moreover, if the application is ported to a platform with a different thread-specific storage API, the code internal to each thread-specific class must be altered to use the new thread library. In addition, making changes directly to the thread-specific class makes it hard to change the threading policies. For instance, changing a thread-specific class to a global class would require intrusive changes to the code, which reduces flexibility and reusability. In particular, each access to state internal to the object would require changes to the helper method that retrieves the state from thread-specific storage.

9.2 C++ Template Wrapper

A more reusable, portable, and flexible approach is to implement a `TS Object Proxy` template that is responsible for all thread-specific methods. This approach allows classes to be decoupled from the knowledge of how thread-specific storage is implemented. This solution improves the reusability, portability, and flexibility of the code by defining a proxy class called `TSS`. As shown below, this class is a template that is parameterized by the class whose objects reside in thread-specific storage:

```
// TS Proxy template
template <class TYPE>
class TSS
{
public:
  // Constructor.
  TSS (void);

  // Destructor
  ~TSS (void);

  // Use the C++ "smart pointer" operator to
  // access the thread-specific TYPE object.
  TYPE *operator-> ();

private:
  // Key for the thread-specific error data.
  pthread_key_t key_;

  // "First time in" flag.
  int once_;

  // Avoid race conditions during initialization.
  Thread_Mutex keylock_;

  // Cleanup hook that deletes dynamically
  // allocated memory.
  static void cleanup_hook (void *ptr);
};
```

The methods in this class are described below. As before, error checking has been minimized to save space.

9.2.1 The C++ Delegation Operator

Applications can invoke methods on a `TSS` proxy as if they were calling the target class by overloading the C++ delegation operator (`operator->`). The C++ delegation operator used in this implementation controls all access to the thread-specific object of class `TYPE`. The `operator->` method receives special treatment from the C++ compiler. As described in Section 9.2.3, it first obtains a pointer to the appropriate `TYPE` from thread-specific storage and then redelegates the original method invoked on it.

Most of the work in the `TSS` class is performed in the `operator->` method shown below:

```
template <class TYPE> TYPE *
TSS<TYPE>::operator-> ()
{
  TYPE *tss_data = 0;

  // Use the Double-Checked Locking pattern to
  // avoid locking except during initialization.

  // First check.
  if (this->once_ == 0) {
    // Ensure that we are serialized (constructor
    // of Guard acquires the lock).

    Guard <Thread_Mutex> guard (this->keylock_);

    // Double check
    if (this->once_ == 0) {
      pthread_key_create (&this->key_,
                          &this->cleanup_hook);

      // *Must* come last so that other threads
      // don't  use the key until it's created.
      this->once_ = 1;
    }
    // Guard destructor releases the lock.
  }

  // Get the data from thread-specific storage.
  // Note that no locks are required here...
  pthread_getspecific (this->key_,
                       (void **) &tss_data);

  // Check to see if this is the first time in
  // for this thread.
  if (tss_data == 0) {
    // Allocate memory off the heap and store
    // it in a pointer in thread-specific
    // storage (on the stack...).
    tss_data = new TYPE;

    // Store the dynamically allocated pointer in
    // thread-specific storage.
    pthread_setspecific (this->key_,
                         (void *) tss_data);
  }
```

```
  return tss_data;
}
```

The TSS template is a proxy that transparently transforms ordinary C++ classes into type-safe, thread-specific classes. It combines the operator-> method with other C++ features like templates, inlining, and overloading. It also utilizes patterns like Double-Checked Locking Optimization [5] and Proxy [6, 7]).

The Double-Checked Locking Optimization pattern is used in operator-> to test the once_ flag twice in the code. Although multiple threads could access the same instance of TSS simultaneously, only one thread can validly create a key (*i.e.*, via pthread_key_create). All threads will then use this key to access a thread-specific object of the parameterized class TYPE. Therefore, operator-> uses a Thread_Mutex keylock_ to ensure that only one thread executes pthread_key_create.

The first thread that acquires keylock_ sets once_ to 1 and all subsequent threads that call operator-> will find once_ != 0 and therefore skip the initialization step. The second test of once_ handles the case where multiple threads executing in parallel queue up at keylock_ before the first thread has set once_ to 1. In this case, when the other queued threads finally obtain the mutex keylock_, they will find once_ equal to 1 and will not execute pthread_key_create.

Once the key_ is created, no further locking is necessary to access the thread-specific data. This is because the pthread_{getspecific, setspecific} functions retrieve the TS Object of class TYPE from the state of the calling thread. No additional locks are needed since this thread state is independent from other threads.

In addition to reducing locking overhead, the implementation of class TSS shown above shields application code from knowledge of the fact that objects are specific to the calling thread. To accomplish this, the implementation uses C++ features such as templates, operator overloading, and the delegation operator (*i.e.*, operator->).

9.2.2 The Constructor and Destructor

The constructor for the TSS class is minimal, it simply initializes the local instance variables:

```
template <class TYPE>
TSS<TYPE>::TSS (void): once_ (0), key_ (0) {}
```

Note that we do not allocate the TSS key or a new TYPE instance in the constructor. There are several reasons for this design:

- **Thread-specific storage semantics:** The thread that initially creates the TSS object (*e.g.*, the main thread) is often not the same thread(s) that use this object (*e.g.*, the worker threads). Therefore, there is no benefit from pre-initializing a new TYPE in the constructor since this instance will only be accessible by the main thread.

- **Deferred initialization:** On some OS platforms, TSS keys are a limited resource. For instance, Windows NT only allows a total of 64 TSS keys per-process. Therefore, keys should not be allocated until absolutely necessary. Instead, the initialization is deferred until the first time the operator-> method is called.

The destructor for TSS presents us with several tricky design issues. The obvious solution is to release the TSS key allocated in operator->. However, there are several problems with this approach:

- **Lack of features:** Win32 and POSIX pthreads define a function that releases a TSS key. However, Solaris threads do not. Therefore, writing a portable wrapper is hard.

- **Race conditions:** The primary reason that Solaris threads do not provide a function to release the TSS key is that it is costly to implement. The problem is that each thread separately maintains the objects referenced by that key. Only when all these threads have exited and the memory reclaimed is it safe to release the key.

As a result of the problems mentioned above, our destructor is a no-op:

```
template <class TYPE>
TSS<TYPE>::~TSS (void)
{
}
```

The cleanup_hook is a static method that casts its ptr argument to the appropriate TYPE * before deleting it:

```
template <class TYPE> void
TSS<TYPE>::cleanup_hook (void *ptr)
{
  // This cast is necessary to invoke
  // the destructor (if it exists).
  delete (TYPE *) ptr;
}
```

This ensures that the destructor of each thread-specific object is called when a thread exits.

9.2.3 Use-case

The following is a C++ template wrapper-based solution for our continuing example of a thread-specific Logger accessed by multiple worker threads:

```
// This is the "logically" global, but
// "physically" thread-specific logger object,
// using the TSS template wrapper.
static TSS<Logger> logger;

// A typical worker function.
static void *worker (void *arg)
{
  // Network connection stream.
  SOCK_Stream *stream =
    static_cast <SOCK_Stream *> arg;

  // Read from the network connection
  // and process the data until the connection
  // is closed.
```

```
  for (;;) {
    char buffer[BUFSIZ];
    int result = stream->recv (buffer, BUFSIZ);

    // Check to see if the recv() call failed.
    if (result == -1) {
      if (logger->errno () != EWOULDBLOCK)
        // Record error result.
        logger->log ("recv failed, errno = %d",
                     logger->errno ());
    } else
      // Perform the work on success.
      process_buffer (buffer);
  }
}
```

Consider the call to `logger->errno` above. The C++ compiler replaces this call with two method calls. The first is a call to `TSS::operator->`, which returns a `Logger` instance residing in thread-specific storage. The compiler then generates a second method call to the `errno` method of the logger object returned by the previous call. In this case, `TSS` behaves as a proxy that allows an application to access and manipulate the thread-specific error value as if it were an ordinary C++ object.[6]

The `Logger` example above is a good example where using a logically global access point is advantageous. Since the `worker` function is global, it is not straightforward for threads to manage both a physical *and* logical separation of `Logger` objects. Instead, a thread-specific `Logger` allows multiple thread to use a single logical access point to manipulate physically separate TSS objects.

9.2.4 Evaluation

The `TSS` proxy design based on the C++ `operator->` has the following benefits:

• **Maximizes code reuse** by decoupling thread-specific methods from application-specific classes (*i.e.*, the formal parameter class TYPE) it is not necessary to rewrite the subtle thread-specific key creation and allocation logic.

• **Increases portability:** Porting an application to another thread library (such as the `TLS` interfaces in Win32) only require changing the `TSS` class, not any applications using the class.

• **Greater flexibility and transparency:** Changing a class to/from a thread-specific class simply requires changing how an object of the class is defined. This can be decided at compile-time, as follows:

```
#if defined (_REENTRANT)
static TSS<Logger> logger;
#else
// Non-MT behavior is unchanged.
Logger logger;
#endif /* REENTRANT */
```

[6]Note that C++ `operator->` does not work for built-in types like `int` since there are no methods that can be delegated to, which is why we cannot use `int` in place of the `Logger` class used above.

Note that the use-case for logger remains unchanged regardless of whether the thread-specific or non-thread-specific form of `Logger` is used.

10 Known Uses

The following are known uses of the Thread-Specific Storage pattern:

- The `errno` mechanism implemented on OS platforms that support the POSIX and Solaris threading APIs are widely-used examples of the Thread-Specific Storage pattern [1]. In addition, the C runtime library provided with Win32 supports thread-specific `errno`. The Win32 `GetLastError/SetLastError` functions also implement the Thread-Specific Storage pattern.
- In the Win32 operating system, windows are owned by threads [8]. Each thread that owns a window has a private message queue where the OS enqueues user-interface events. API calls that retrieve the next message waiting to be processed dequeue the next message on the calling thread's message queue, which resides in thread-specific storage.
- OpenGL [9] is a C API for rendering three-dimensional graphics. The program renders graphics in terms of polygons that are described by making repeated calls to the `glVertex` function to pass each vertex of the polygon to the library. State variables set before the vertices are passed to the library determine precisely what OpenGL draws as it receives the vertices. This state is stored as encapsulated global variables within the OpenGL library or on the graphics card itself. On the Win32 platform, the OpenGL library maintains a unique set of state variables in thread-specific storage for each thread using the library.
- Thread-specific storage is used within the ACE network programming toolkit [10] to implement its error handling scheme, which is similar to the `Logger` approach described in Section 9.2.3. In addition, ACE implements the type-safe thread-specific storage template wrappers described in Section 9.2.

11 Related Patterns

Objects implemented with thread-specific storage are often used as per-thread Singletons [7], *e.g.*, `errno` is a per-thread Singleton. Not all uses of thread-specific storage are Singletons, however, since a thread can have multiple instances of a type allocated from thread-specific storage. For instance, each `Task` object implement in ACE [10] stores a cleanup hook in thread-specific storage.

The `TSS` template class shown in Section 8 serves as a Proxy [7, 6] that shields the libraries, frameworks, and applications from the implementation of thread-specific storage provided by OS thread libraries.

The Double-Checked Locking Optimization pattern [5] is commonly used by applications that utilize the Thread-Specific Storage pattern to avoid constraining the order of initialization for thread-specific storage keys.

12 Concluding Remarks

Multi-threading an existing application often adds significant complexity to the software due to the additional concurrency control protocols needed to prevent race conditions and deadlocks [11]. The Thread-Specific Storage pattern alleviates some of synchronization overhead and programming complexity by allowing multiple threads to use one logically global access point to retrieve thread-specific data without incurring locking costs for each access.

Application threads use `TS Object Proxies` to access `TS Objects`. The proxies delegate to `TS Object Collections` to retrieve the objects corresponding to each application thread. This ensures that different application threads do not share the same `TS Object`.

Section 9.2 showed how the `TS Object Proxy` participant of the Thread-Specific Storage pattern can be implemented to ensure threads only access their own data through strongly-typed C++ class interfaces. When combined with other patterns (such as Proxy, Singleton, and Double-Checked Locking) and C++ language features (such as templates and operator overloading), the `TS Proxy` can be implemented so that objects using the Thread-Specific Storage pattern can be treated just like conventional objects.

Acknowledgements

Thanks to Peter Sommerlad and Hans Rohnert for their insightful comments on earlier versions of this paper.

References

[1] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Kernel," in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

[2] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[3] F. Mueller, "A Library Implementation of POSIX Threads Under UNIX," in *Proceedings of the Winter USENIX Conference*, (San Diego, CA), pp. 29–42, Jan. 1993.

[4] IEEE, *Threads Extension for Portable Operating Systems (Draft 10)*, February 1996.

[5] D. C. Schmidt and T. Harrison, "Double-Checked Locking – An Object Behavioral Pattern for Initializing and Accessing Thread-safe Objects Efficiently," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons, 1996.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[8] C. Petzold, *Programming Windows 95*. Microsoft Press, 1995.

[9] J. Neider, T. Davis, and M. Woo, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1*. Addison-Wesley, Reading, MA, 1993.

[10] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the* 6^{th} *USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[11] J. Ousterhout, "Why Threads Are A Bad Idea (for most purposes)," in *USENIX Winter Technical Conference*, (San Diego, CA), USENIX, Jan. 1996.

[12] H. Mueller, "Patterns for Handling Exception Handling Successfully," *C++ Report*, vol. 8, Jan. 1996.

[13] N. Pryce, "Type-Safe Session: An Object-Structural Pattern," in *Submitted to the* 2^{nd} *European Pattern Languages of Programming Conference*, July 1997.

[14] J. Gosling and F. Yellin, *The Java Application Programming Interface Volume 2: Window Toolkit and Applets*. Addison-Wesley, Reading, MA, 1996.

A Alternative Solutions

In practice, thread-specific storage is typically used to resolve the following two use-cases for object-oriented software:

1. To implicitly communicate information (*e.g.*, error information) between modules.
2. To adapt legacy single-threaded software written in a procedural style to modern multi-threaded operating systems and programming languages.

It is often a good idea, however, to avoid thread-specific storage for use-case #1 because it can increase coupling between modules and reduce reusability. In the case of error handling, for instance, thread-specific storage can often be avoided by using exceptions, as described in Section A.1.

The use of thread-specific storage for use-case #2 can not be avoided except through redesign. When designing new software, however, thread-specific storage can often be avoided by using exception handling, explicit intercomponent communication contexts, or reified threads, as described below.

A.1 Exception Handling

An elegant way of reporting errors between modules is to use exception handling. Many modern languages, such as C++ and Java, use exception handling as an error reporting mechanism. It is also used in some operating systems, such as Win32. For example, the following code illustrates a hypothetical OS whose system calls throw exceptions:

```
void *worker (SOCKET socket)
{
  // Read from the network connection
  // and process the data until the connection
  // is closed.
  for (;;) {
    char buffer[BUFSIZ];

    try {
      // Assume that recv() throws exceptions.
      recv (socket, buffer, BUFSIZ, 0);
      // Perform the work on success.
      process_buffer (buffer);
    } catch (EWOULDBLOCK) {
      continue;
    } catch (OS_Exception error) {
       // Record error result in thread-specific data.
       printf ("recv failed, error = %s",
               error.reason);
    }
  }
```

There are several benefits to using exception handling:

• **It is extensible:** Modern OO languages facilitate the extension of exception handling policies and mechanisms via features (such as using inheritance to define a hierarchy of exception classes) that have minimal intrusion on existing interfaces and usage.

• **It cleanly decouples error handling from normal processing:** For example, error handling information is not passed explicitly to an operation. Moreover, an application cannot accidentally "ignore" an exception by failing to check function return values.

• **It can be type-safe:** In a strongly typed languages, such as C++ and Java, exceptions are thrown and caught in a strongly-typed manner to enhance the organization and correctness of error handling code. In contrast to checking a thread-specific error value explicitly, the compiler ensures that the correct handler is executed for each type of exception.

However, there are several drawbacks to the use of exception handling:

• **It is not universally available:** Not all languages provide exception handling and many C++ compilers do not implement exceptions. Likewise, when an OS provides exception handling services, they must be supported by language extensions, thereby reducing the portability of the code.

• **It complicates the use of multiple languages:** Since languages implement exceptions in different ways, or do not implement exceptions at all, it can be hard to integrate components written in different languages when they throw exceptions. In contrast, reporting error information using integer values or structures provides a universal solution.

• **It complicates resource management:** *e.g.,* by increasing the number of exit paths from a block of C++ code [12]. If garbage collection is not supported by the language or programming environment, care must be taken to ensure that dynamically allocated objects are deleted when an exception is thrown.

• **It is potentially time and/or space inefficient:** Poor implements of exception handling incur time and/or space overhead even when exceptions are not thrown [12]. This overhead can be particularly problematic for embedded systems that must be small and efficient.

The drawbacks of exception handling are particularly problematic for system-level frameworks (such as kernel-level device drivers or low-level communication subsystems) that must run portably on many platforms. For these types of systems, a more portable, efficient, and thread-safe way to handle errors is to define an error handler abstraction that maintains information about the success or failure of operations explicitly.

A.2 Explicit Contexts for Intercomponent Communication

Thread-specific storage is usually used to store per-thread state to allow software components in libraries and frameworks to communicate efficiently. For example, `errno` is used to pass error values from a called component to the caller. Likewise, OpenGL API functions are called to pass information to the OpenGL library, which are stored in thread-specific state. The use of thread-specific storage can be avoided by explicitly representing the information passed between components as an object.

If the type of information that must be stored by the component for its users is known in advance, the object can be created by the calling thread and passed to the component as an extra argument to its operations. Otherwise, the component must create an object to hold context information in response to a request from the calling thread and return an identifier for the object to the thread before the thread can make use of the component. These types of objects are often called *context objects*; context objects that are created on demand by a software component are often called *sessions*.

A simple example of how a context object can be created by a calling thread is illustrated by the following error handling scheme, which passes an explicit parameter to every operation:

```
void *worker (SOCKET socket)
{
  // Read from the network connection and
  // process the data until the connection
  // is closed.

  for (;;) {
    char buffer[BUFSIZ];
    int result;
    int errno;

    // Pass the errno context object explicitly.
    result = recv (socket, buffer, BUFSIZ,
                   0, &errno);

    // Check to see if the recv() call failed.
    if (result == -1) {
      if (errno != EWOULDBLOCK)
        printf ("recv failed, errno = %d", errno);
```

```
  } else
    // Perform the work on success.
    process_buffer (buffer);
}
```

Context objects created by components can be implemented by using the Type-Safe Session pattern [13]. In this pattern, the context object stores the state required by the component and provides an abstract interface that can be invoked polymorphically. The component returns a pointer to the abstract interface to the calling thread that subsequently invokes operations of the interface to use the component.

An example of how Type-Safe Sessions are used is illustrated by the difference between OpenGL and the interface provided by the Java AWT library [14] for rendering graphics onto devices such as windows, printers or bitmaps. In the AWT, a program draws onto a device by requesting a `GraphicsContext` from the device. The `GraphicsContext` encapsulates the state required to render onto a device and provides an interface through which the program can set state variables and invoke drawing operations. Multiple `GraphicsContext` objects can be created dynamically, thereby removing any need to hold thread-specific state.

The benefits of using context objects compared with thread-local storage and exception handling are the following:

- **It is more portable:** It does not require language features that may not be supported universally;

- **It is more efficient:** The thread can store and access the context object directly without having to perform a look-up in the thread-specific storage table. It does not require the compiler to build additional data structures to handle exceptions.

- **It is thread-safe:** The context object or session handle can be stored on the thread's stack, which is trivially thread-safe.

There are several drawbacks with using context objects created by the calling thread, however:

- **It is obtrusive:** The context object must be passed to every operation and must be explicitly checked after each operation. This clutters the program logic and may require changes to existing component interfaces to add an error handler parameter.

- **Increased overhead per invocation:** Additional overhead will occur for each invocation since an additional parameter must be added to every method call, regardless of whether the object is required. Although this is acceptable in some cases, the overhead may be noticeable for methods that are executed very frequently. In contrast, an error handling scheme based on thread-specific storage need not be used unless an error occurs.

Compared to creating context objects in the calling thread, using sessions created by the component has the following benefits:

- **It is less obtrusive:** A thread does not have to explicitly pass the context object to the component as an argument to its operations. The compiler arranges for a pointer to context object to be passed to its operations as the hidden `this` pointer.

- **It automates initialization and shutdown:** A thread cannot start using a session until it has acquired one from a component. Components can therefore ensure that operations are never called when they are in inconsistent states. In contrast, if a component uses hidden state, a caller must explicitly initialize the library before invoking operations and shutdown the component when it has finished. Forgetting to do so can cause obscure errors or waste resources.

- **Structure is explicit:** The relationships between different modules of code is explicitly represented as objects, which makes it easier to understand the behavior of the system.

Creating context objects within the component has the following drawback compared to creating them upon the caller's stack:

- **Allocation overhead:** The component must allocate the session object on the heap or from some encapsulated cache. This will be usually be less efficient than allocating the object on the stack.

A.3 Objectified Threads

In an object-oriented language, an application can explicitly represent threads as objects. Thread classes can be defined by deriving from an abstract base class that encapsulates any state required to run as a concurrent thread and invokes an instance method as the entry point into the thread. The thread entry method would be defined as a pure virtual function in the base class and defined in derived classes. Any required thread-specific state (such as session contexts) can be defined as object instance variables, making it available to any method of the thread class. Concurrent access to these variables can be prevented through the use of language-level access control mechanisms rather than explicit synchronization objects.

The following illustrates this approach using a variant of the ACE `Task` [10], which can be used to associate a thread of control with an object.

```
class Task
{
public:
  // Create a thread that calls the svc() hook.
  int activate (void);

  // The thread entry point.
  virtual void svc (void) = 0;

private:
  // ...
};

class Animation_Thread : public Task
```

```
{
public:
  Animation_Thread (Graphics_Context *gc)
    : device_ (gc) {}

  virtual void svc (void)
  {
    device_->clear ();
    // ... perform animation loop...
  }

private:
  Graphics_Context *device_;
};
```

The use of objectified threads has the following advantages:

• **It is more efficient:** A thread does not need to perform a look-up in a hidden data structure to access thread-specific state.

• **It is not obtrusive:** When using an objectified thread, a pointer to the current object is passed as an extra argument to each function call. Unlike the explicit session context, the argument is hidden in the source code and managed automatically by the compiler, keeping the source code uncluttered.

The use of objectified threads has the following disadvantages:

• **Thread-specific storage is not easily accessible:** Instance variables cannot be accessed except by class methods. This makes it non-intuitive to use instance variables to communicate between reusable libraries and threads. However, using thread-specific storage in this way increases coupling between components. In general, exceptions provide a more decoupled way of reporting errors between modules, though they have their own traps and pitfalls in languages like C++ [12].

• **Overhead:** The extra, hidden parameter passed to every operation will cause some overhead. This may be noticeable in functions that are executed very frequently.