An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit

Douglas C. Schmidt
schmidt@cs.wustl.edu
http://www.cs.wustl.edu/~schmidt/
Department of Computer Science
Washington University
St. Louis, MO 63130, (314) 935-7538

This document is available as technical report number WUCS-95-31, from techrep@cs.wustl.edu at Washington University, St. Louis.

Abstract

This paper describes the design of the ACE object-oriented thread encapsulation C++ class library. This library shields programmers from differences between Solaris threads, POSIX pthreads, and Win32 threads. The architecture of this class library is presented from an end-user and internal design perspective and key design and implementation issues are discussed. Readers will gain an understanding of the overall design approach, as well as the tradeoffs between various software quality factors such as performance, portability, and extensibility.

1 Introduction

Certain types of distributed applications benefit from using a concurrent model of execution to perform their tasks. Concurrency is particularly useful to improve performance and simplifying programming for network servers on multiprocessor platforms. For server applications, using threads to handle multiple client requests concurrently is often more convenient and less error-prone than the following design alternatives:

- Artificially serializing requests at a transport layer interface;
- Queueing requests internally and handling them iteratively;
- Forking a heavy-weight process for each client request.

This paper describes a C++ class library contained in the ADAPTIVE Communications Environment (ACE) [1]. ACE encapsulates and enhances the lightweight concurrency mechanisms provided both by Solaris 2.x threads [2], POSIX Pthreads [3], and Win32 threads [4].

The material presented in this paper is intended for a technical audience interested in understanding the strategies and tactics of object-oriented (OO) concurrent programming using threads. It is assumed that the reader is familiar with general OO design and programming techniques (such as design patterns [5], application frameworks [6], modularity, information hiding, and object modeling [7]), OO notations (such as OMT [8]), fundamental C++ programming language features (such as classes, inheritance, dynamic binding, and parameterized types [9]), basic UNIX systems programming concepts (such as process management, virtual memory, and interprocess communication [10]), and networking terminology (such as client/server architectures [11], RPC [12], CORBA [13], and TCP/IP [14, 15]).

This paper does not assume in-depth knowledge of concurrency, in general, or Solaris/POSIX/Win32 multi-threading and synchronization mechanisms, in particular. An overview of concurrent programming and multi-threading is presented in Section 3. The overview defines key terminology and outlines the various alternative mechanisms available for concurrent programming on Solaris 2.x, POSIX pthreads, and Win32 threads.

This paper is organized as follows: Section 2 gives an overview of the goals of the ACE OS thread encapsulation library and outlines the OO architecture of the library components. Section 3 presents relevant background material on concurrent programming, in general, and the Solaris multi-threading model, in particular. Section 4 presents an end-user perspective that motivates the design of the ACE thread encapsulation library, focusing on a "use case" example culled from a concurrent client/server application. Section 5 describes the public interfaces and internal design of the ACE threads encapsulation library in detail. Section 6 presents several examples that illustrate the OO components defined in Section 5. Finally, Section 7 presents concluding remarks.

2 Overview of the ACE OO Concurrency Mechanisms

2.1 Overall Goals

A distinct feature of modern operating systems (such as Solaris, OSF/1, Windows NT, and OS/2) compared to previous generations of SunOS is its integrated support for kernel-level and user-level multi-threading and synchronization.[1] However, the existing multi-threading and synchronization mechanisms shipped with these operating systems are relatively low-level APIs written in C. Developing applications using a mixture of C++ classes and low-level C APIs places an unacceptable burden on developers. Mixing these two styles within a single application leads to an "impedance" mismatch between object-oriented and procedural programming. Such a hybrid programming style is distracting and a chronic maintenance problem.

To avoid having each developer re-implement their own *ad hoc* C++ wrappers for OS threading mechanisms, the ACE toolkit provides a set of object-oriented concurrency components described in this paper. These ACE components provide a portable and extensible interface for concurrent programming. This interface simplifies thread management and synchronization mechanisms used to develop clients and servers. This interface has been ported to many drafts of the POSIX pthreads standard [3], Solaris threads [2], Microsoft WIN32 threads [4], and VxWorks tasks.

2.1.1 Overall Requirements

In conjunction with the goal of encapsulating and simplifying the concurrency substrate of OS threading mechanisms, the ACE OO thread encapsulation class library is being developed in response to the following common application requirements.

- *Simplify program design* – by allowing multiple application tasks to proceed independently using conventional synchronous programming abstractions (such as CORBA remote method invocations);
- *Transparently improve performance* – by using the parallel processing capabilities of hardware platforms such as the SPARCcenter 1000 and 2000 shared memory symmetric multi-processors;
- *Explicitly improve performance* – by reducing data copying and by overlapping computation with communication;
- *Improve perceived response time* – for interactive applications (such as user interfaces or network management applications) by associating separate threads with different tasks or services in an application.

2.1.2 Design Goals

The ACE OO thread class library was developed to achieve the following design goals:

- Improve consistency of programming style by enabling developers to use C++ and OO consistently throughout their concurrent applications.
- Improve the portability and reusability of the underlying concurrency mechanisms.
- Reduce the amount of obtrusive changes to make applications thread-safe.
- Eliminate or minimize the potential for subtle synchronization errors.
- Enhance abstraction and modularity *without* compromising performance.

2.2 Architectural Overview of ACE OO Thread Encapsulation Components

Figure 1 is a Booch object model that illustrates the components in the ACE threads encapsulation class library. These components include the C++ classes and class categories described below.[2]

2.2.1 The ACE Locks Class Category

• **Mutex, Thread_Mutex, and Process_Mutex:** These classes provide a simple and efficient mechanism that serializes access to a shared resource (such as a file or object in shared memory). They encapsulate Solaris, POSIX, and Win32 synchronization variables (`mutex_t`, `pthread_mutex_t`, and `HANDLE`, respectively) and are described in Section 5.1.1.

• **RW_Mutex, RW_Thread_Mutex, RW_Process_Mutex:** These classes serialize access to shared resources whose contents are searched more than they are changed. They encapsulate the Solaris `rwlock_t` synchronization variable (the POSIX pthreads and Win32 threads implementation uses other mechanisms) and is described in Section 5.1.3.

• **Semaphore, Thread_Semaphore, Process_Semaphore, :** These classes implement Dijkstra's "counting semaphore" abstraction, which is a general mechanism for serializing multiple threads of control. They encapsulate the Solaris `sema_t` synchronization variable (the POSIX pthreads and Win32 threads implementation use other mechanisms) and is described in Section 5.1.2.

• **Null_Mutex:** The `Null_Mutex` class provides a zero-overhead implementation of the locking interface used by the other C++ wrappers for synchronization. This class is described in Section 5.1.5.

• **Token:** The `Token` class provides a more general-purpose synchronization mechanism than a `Mutex`. For example, it implements "recursive mutex" semantics, where a thread that owns the token can reacquire it without deadlocking. In addition, threads that are blocked awaiting a `Token` are serviced in strict FIFO order as other threads release the token (in contrast, `Mutex` don't strictly enforce an acquisition order). This class is described in Section 5.1.6.

[1] This section focuses on Solaris 2.x threading and synchronization mechanisms for concreteness. However, most of the mechanisms, design principles, and interfaces are equivalent for POSIX Pthreads and Win32 threads, as well.

[2] All ACE classes are prefixed with `ACE_` to avoid polluting the global name space of programs. For brevity, this prefix has been omitted in all the following code.

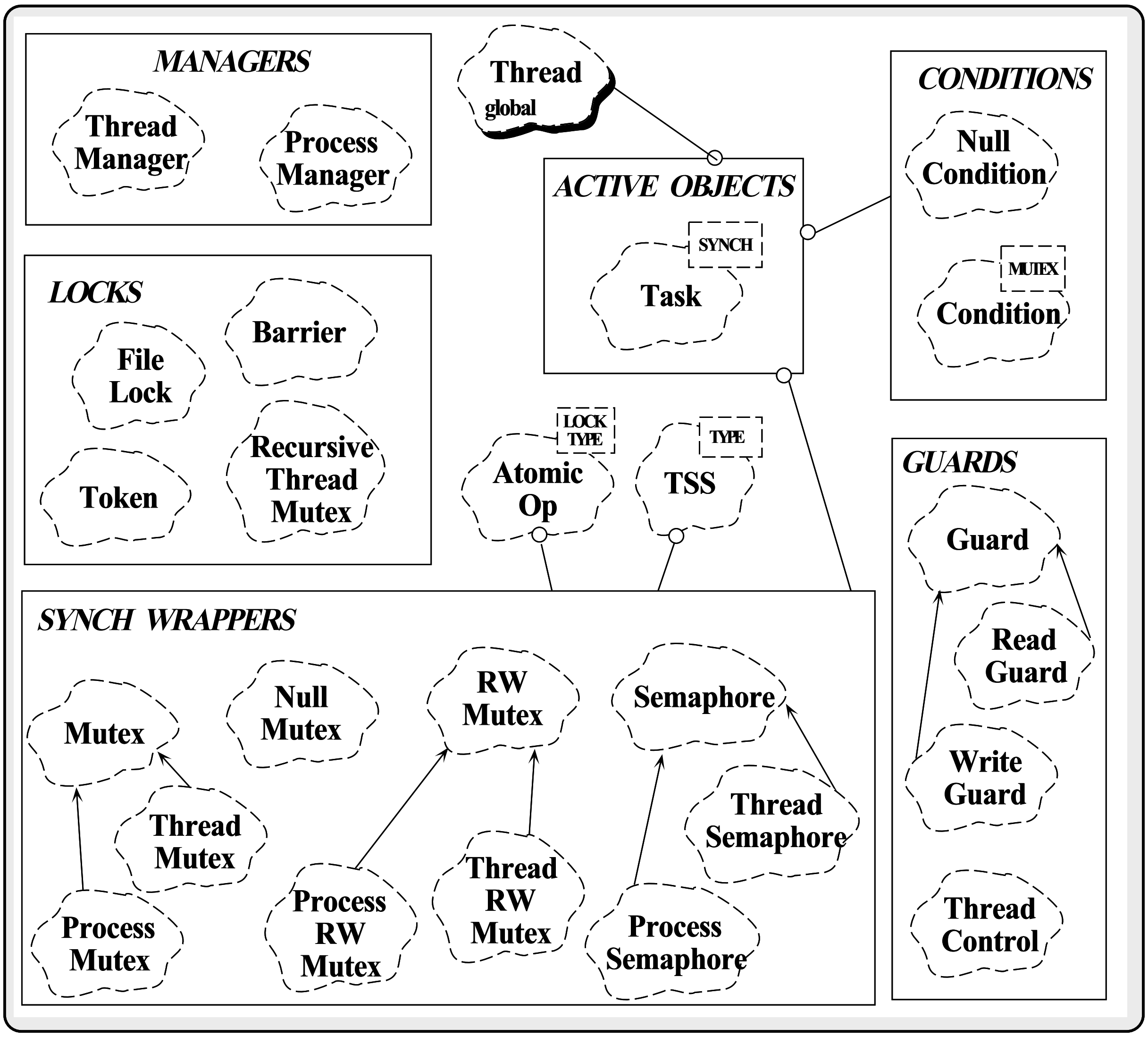

Figure 1: Object Model for ACE OO Thread Encapsulation Components

• **Recursive_Thread_Mutex:** A `Recursive_Thread_Mutex` extends the default Solaris thread mutex semantics by allowing calls to `acquire` methods to nest as long as the thread that owns the lock is the one that re-acquires it. It works with the `Thread_Mutex` class outlined above and is described in Section 5.1.4.

2.2.2 The ACE Guards Class Category

• **Guard, Write_Guard, and Read_Guard:** these classes ensure that a lock is automatically acquired and released upon entry and exit to a block of C++ code, respectively. They are described in Section 5.2.1.

• **Thread_Control:** The `Thread_Control` class is used in conjunction with the `Thread_Manager` class to automate the graceful termination and cleanup of a thread's activities within its originating function. This class is described in Section 5.2.2.

2.2.3 The ACE Conditions Class Category

• **Condition:** The `Condition` class is used to block on a change in the state of a condition expression involving shared data. It encapsulates the Solaris and POSIX pthreads `cond_t` synchronization variable (Win32 threads are implemented using other mechanisms) and is described in Section 5.3.1.

• **Null_Condition:** The `Null_Condition` class provides a zero-overhead implementation of the `Condition` interface used for single-threaded applications. It is described in Section 5.3.2.

2.2.4 The ACE Managers Class Category

• **Thread_Manager:** The `Thread_Manager` class contains a set of mechanisms to manage groups of threads that collaborate to implement collective actions. This class is described in Section 5.4.1.

• **Thread_Spawn:** The `Thread_Spawn` class provides a standard utility that manages the creation of threads to handle requests from clients concurrently. This class is described in Section 5.4.2.

2.2.5 The ACE Active Objects Class Category

• **Task:** The `Task` class is the central mechanism in ACE for defining *active objects* [16, 17]. These active objects queue messages for input and output and perform user-defined message processing services in separate threads of control. This class is described in Section 5.5.1.

2.2.6 Miscellaneous ACE Concurrency Classes

• **Thread:** The `Thread` class encapsulates the Solaris threads, POSIX Pthreads, and Win32 threads family of thread creation, termination, and management routines. This class is described in Section 5.6.1.

• **Atomic_Op:** The `Atomic_Op` class transparently parameterizes synchronization into basic arithmetic operations. This class is described in Section 5.6.2.

• **Barrier:** The `Barrier` class implements "barrier synchronization," which is particularly useful for many types of parallel scientific applications. This class is described in Section 5.6.3.

• **TSS:** The `TSS` class allows objects that are "physically" thread-specific (*i.e.,* private to a thread) to be accessed as though they were "logically" global to a program. This class is described in Section 5.6.4.

3 Background on Concurrent Programming and Multi-threading

Most UNIX systems programmers are familiar with traditional process management system calls (such as `fork`, `exec`, `wait`, and `exit`). There is less experience, however, with emerging multi-threading and synchronization mechanisms for UNIX (such as Solaris threads [2], POSIX pthreads [3], or Win32 threads [4]). This section presents an overview of background material relevant to concurrent programming and Solaris threads. More detailed discussions of concurrent programming, and Solaris/POSIX/Win32 threads appear in [2, 18, 19, 3, 4].

3.1 Processes and Threads

A *process* is a collection of resources that enable the execution of program instructions. These resources include virtual memory, I/O descriptors, a run-time stack, signal handlers, user and group ids, and access control tokens. On earlier-generation UNIX systems (such as SunOS 4.x), processes were "single-threaded." In UNIX, operations in single-threaded programs are generally synchronous since control is always in either the program (*i.e.,* the user code) or in the operating system (via system calls). To some extent, the single-threaded nature of traditional UNIX processes simplifies programming since processes do not interfere with one another without explicit intervention by programmers.

However, many applications (particularly networking servers) are difficult to develop using single-threaded processes. For example, a single-threaded network file server must not block for extended periods of time handling one client request since the responsiveness for other clients would suffer. There are several common workarounds to avoid blocking in single-threaded servers:

- *Event demultiplexers/dispatcher* – one approach is to develop an event demultiplexer/dispatcher (such as the object-oriented Reactor framework [20]). This technique is widely used to manage multiple input devices in single-threaded user-interface frameworks. The main event demultiplexer/dispatcher detects an incoming event, demultiplexes the event to the appropriate event handler, and then dispatches an application-specific callback method associated with the event handler.

  The primary drawback with this approach is that long duration conversations must be developed as finite state machines. This approach becomes unwieldy as the number of states increase. In addition, since only non-blocking operations may be used, it is difficult to improve performance via techniques such as "I/O streaming" or schemes that benefit from locality of reference in data and instruction caches.

- *User-level co-routines* – another approach is to develop a non-preemptive, user-level co-routine package that explicitly saves and restores context information. This enables tasks to suspend their execution until another co-routine resumes them at a later point. The multi-tasking mechanisms on Windows 3.1 and the Mac System 7 OS are widely available systems that use this approach.

  In general, co-routines are complicated to use correctly since developers must manually perform task preemption by explicitly yielding the thread of control periodically. Moreover, each task must execute for a relatively short duration. Otherwise, clients may detect that requests are being handled sequentially rather than concurrently. Another limitation with co-routines is that application performance may be reduced if the OS blocks all services in a process whenever one task incurs a page fault. Moreover, the failure of a single task (*e.g.,* spinning in an infinite loop) may hang the entire process.

- *Multi-processing* – another approach for alleviating the complexity of single-threaded UNIX processes is to use the coarse-grained multi-processing capabilities provided by the `fork` and `exec` system calls. `Fork` spawns a separate child process that executes a task concurrently with its parent. It is possible for separate processes to collaborate directly by using mechanisms such as shared memory and memory-mapped files. On a local host, shared memory is a faster means of IPC than message passing since it avoids explicit data copying.

However, the overhead and inflexibility of `fork` and `exec` makes dynamic process invocation prohibitively expensive and overly complicated for many applications. For example, the process management overhead for short-duration services (such as resolving the Ethernet number of an IP address, retrieving a disk block from a network file server, or setting an attribute in an SNMP MIB) is excessive. Moreover, it is difficult to exert fine-grain control over scheduling and process priority using `fork` and `exec`. In addition, processes that share C++ objects in shared memory segments must make non-portable assumptions about the placement of virtual table pointers.

Multi-threading mechanisms provide a more elegant, and sometimes more efficient, way to overcome the limitations with the traditional concurrent processing techniques described above. A thread is a single sequence of execution steps performed in the context of a process. In addition to an instruction pointer, a thread consists of other resources such as a run-time stack of function activation records, a set of general-purpose registers, and thread-specific data.

Conventional workstation operating systems (such as variants of UNIX [2, 21, 22] and Windows NT [4]) support the concurrent execution of multiple processes, each of which may contain 1 or more threads. A process serves as the unit of protection and resource allocation within a separate hardware protected address space. A thread serves as the unit of execution that runs within a process address space that is shared with 0 or more threads.

3.2 Benefits of Threads-based Concurrent Programming

It is often advantageous to implement concurrent applications that perform multiple tasks in separate threads rather than in separate processes for the following reasons:

- *Thread creation* – unlike forking a new processes, spawning a new thread does not require (1) duplicating the parent's address space memory, (2) setting up new kernel data structures, and (3) consuming an extra process slot in order to perform a subtask within a larger application.
- *Context switching* – Threads maintain minimal state information. Therefore, context switching overhead is reduced since less state information must be stored and retrieved. In particular, context switching between threads is less time consuming than context switching between UNIX heavyweight processes. This is due to the fact that TLB virtual address mappings not need be changed when switching between threads in the same process. Moreover, threads that run strictly in user-level do not incur any context switching overhead.
- *Synchronization* – It may not be necessary to switch between kernel-mode and user-mode when scheduling and executing an application thread. Thread-synchronization is less expensive than process-synchronization. For example, the entities being synchronized are often not global entities, but are localized. Global synchronization always involves the kernel, whereas the local (or "intra-process") synchronization used by application threads may require no kernel intervention.
- *Data Copying* – communicating between separate threads via shared memory is often much faster than using IPC message passing between separate processes since it avoids the overhead of explicit data copying. For example, cooperating database services that frequently reference common memory-resident data structures may be simpler and more efficient to implement via threads. In general, using the shared address space of a process to communicate between threads is easier and more efficient than using shared memory mechanisms (such as System V shared memory or memory-mapped files) to communication between processes.

3.3 Overview of Multi-Processing and Multi-threading on Solaris

This section summarizes relevant background material on the multi-processing (MP) and multi-threading (MT) mechanisms provided by Solaris 2.x. Details of other threading models and implementations (such as SGI, Sequent, OSF/1, and Windows NT) are somewhat different, though the basic concepts are very similar.

A traditional UNIX process is a relatively "heavyweight" entity that contains a single-thread of control. In contrast, the thread-based concurrency mechanisms available on Solaris 2.x are more sophisticated, flexible, and efficient (when used properly). As shown in Figure 2, the Solaris MP/MT architecture operates at 2 levels (kernel-space and user-space) and contains the following 4 components:

- *Processing elements* – which are the CPUs that execute user-level and kernel-level instructions. The semantics of the Sun MP/MT model are intended to work for both uni-processors and symmetrical multi-processors on shared memory hardware.
- *Kernel threads* – which are the fundamental entities that are scheduled and executed by the processing elements (PEs) in kernel space. The OS kernel maintains a small data structure and a stack for each kernel thread. Context switching between kernel threads is relatively fast since it does not require changing virtual memory information.

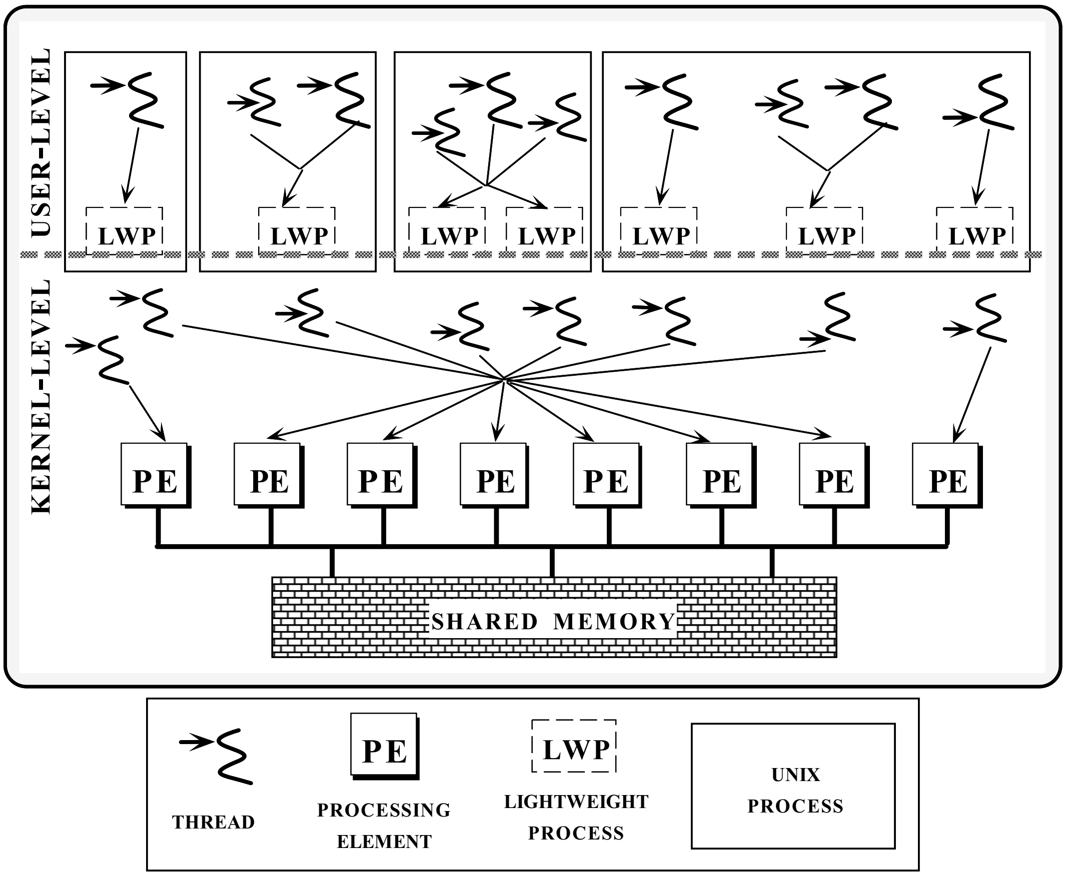

Figure 2: Solaris 2.x Multi-processing and Multi-threading Architecture

- *Lightweight processes (LWPs)* – which are associated with kernel threads. In Solaris 2.x, a UNIX process is no longer a thread of control. Instead, each process contains one or more LWPs. There is a 1-to-1 mapping between its LWPs and its kernel threads.[3] The kernel-level scheduler in Solaris uses LWPs (and thereby kernel threads) to schedule application tasks. An LWP contains a relatively large amount of state (such as register data, accounting and profiling information, virtual memory address ranges, and timers). Therefore, context switching between LWPs is relatively slow.

  For the time-sharing scheduler class (the default), the scheduler divides the available PE(s) among multiple active LWPs via *preemption*. With this technique, each LWP runs for a finite period of time (typically 10 milliseconds). After the time-slice of the current LWP has elapsed, the OS scheduler selects another available LWP, performs a context switch, and places the preempted LWP onto a queue. The kernel schedules LWPs using several criteria (such as priority, availability of resources, scheduling class, etc.). There is no fixed order of execution for LWPs in the time-sharing scheduler class.

- *Application threads* – Each LWP may be thought of as a "virtual PE," upon which application threads are scheduled and multiplexed by a user-level thread library. Each application thread shares its process address space with other threads, though it has a unique stack and register set. An application thread may spawn other application threads. Within a process, each of these application threads execute independently (though not necessarily in parallel depending on the hardware).

  Solaris 2.x provides a multi-level concurrency model that permits application threads to be spawned and scheduled using one of the following two modes:

  1. *Bound threads* – which map 1-to-1 onto LWPs and kernel threads. Bound threads permit independent tasks to execute in parallel on multiple PEs. Thus, if two application threads are running on separate LWPs (and thus separate kernel threads), they may execute in parallel (assuming they are running on a multiprocessor or using asynchronous I/O). Moreover, application threads may perform blocking system calls and handle page faults without impeding each other's progress.

     A kernel-level context switch is required to reschedule bound threads. Likewise, synchroniza-

[3]On the other hand, not every kernel thread has an LWP. For example, there are system threads (like the pagedaemon, NFS daemon, and the callout thread) that have a kernel thread and operate entirely in kernel space.

tion operations on bound threads require OS kernel intervention. Bound threads are most useful when an application is designed to take advantage of parallelism available on the hardware platform. Since each bound thread requires allocation of kernel resources, it may be inefficient to allocate a large number of bound threads.

2. *Unbound threads* – which are multiplexed in an n-to-m manner atop one or more LWPs and kernel threads by a thread run-time library. This user-level library implements a non-preemptive, cooperative multi-tasking concurrency model. It schedules, dispatches, and suspends unbound threads, while minimizing kernel involvement. Compared with using application threads bound to LWPs, unbound application threads require less overhead to spawn, context switch, and synchronize.

   Depending upon the number of kernel threads that an application and/or library associates with a process, one or more unbound threads may execute on multiple PEs in parallel. Since each unbound thread does not allocate kernel resources, it is possible to allocate a very large number of unbound threads without significantly degrading performance.

3.4 Challenges of Concurrent Programming

On a multi-processor, more than one LWP may run in parallel on separate PEs. On a uni-processor, only one LWP will be active at any point in time. Regardless of the hardware platform, programmers must ensure that access to shared resources (such as files, databases records, network devices, terminals, or shared memory) is serialized to prevent *race conditions*. A race condition occurs when the order of execution of two or more concurrent LWPs leads to unpredictable and erroneous results (such as a database record being left in an inconsistent state). Race conditions may be eliminated by using the Solaris 2.x synchronization mechanisms described in Section 3.5. These mechanisms serialize access to critical sections of code that access shared resources.

In addition to the challenges of concurrency control, the following limitations arise when using multi-threading (rather than multi-processing or single-threaded, reactive event loops) to implement concurrent applications:

- *Robustness* – Executing all tasks via threads within a single process address space may reduce application robustness. This problem occurs since separate threads within the same process address space are not protected from one another. In order to reduce context switching and synchronization overhead, threads receive little or no protection from the hardware memory management unit (MMU).[4]

  Since threads are not protected, one faulty service in a process may corrupt global data shared by services running on other threads in the process. This, in turn, may produce incorrect results, crash an entire process, cause a network server to hang indefinitely, etc. A related problem is that certain UNIX system calls invoked in one thread may have undesirable side-effects on an entire process. For example, the `exit` system call has the side-effect of destroying all the threads within a process (`thr_exit` should be used to terminate only the current thread).

- *Access Privileges* – Another limitation with multi-threading is that all threads within a process share the same userid and access privileges to files and other protected resources. Therefore, to prevent accidental or intentional access to unauthorized resources, network services that base their security mechanisms on process ownership (such as the Internet `ftp` and `telnet` services) are typically implemented in separate processes.

- *Performance* – A common misconception is that multi-threading an application will automatically improve performance. In many circumstances, however, multi-threading does not improve performance. For example, compute-bound applications on a uni-processor [19] will not benefit from multi-threading since computation will not overlap communication. In addition, fine-grained locking causes high levels of synchronization overhead [23, 24]. This prevents applications from fully exploiting the benefits of parallel processing.

  There are some circumstances where multi-threading may improve performance significantly. For example, a multi-threading connection-oriented application gateway may benefit by being run on a multi-processor platform. Likewise, on a uni-processor, I/O-bound applications may benefit from multi-threading since computation is overlapped with communication and disk operations.

3.5 Overview of Solaris 2.x Synchronization and Threading Mechanisms

This section outlines and illustrates the synchronization and threading mechanisms available on Solaris 2.x, POSIX pthreads, and Win32 threads. In these systems, threads share various resources (such as open files, signal handlers, and global memory) within a single process address space. Therefore, they must utilize *synchronization mechanisms* to coordinate access to shared data, to avoid the race conditions discussed in Section 3.4. To illustrate the need for synchronization mechanisms, consider the following C++ code fragment:

```
typedef u_long COUNTER;
COUNTER request_count; // At file scope

void *run_svc (Queue<Message> *q)
{
  Message *mb; // Message buffer
```

[4] An MMU protects separate process address spaces from accidental or malicious corruption by other active processes in the system.

```
  while (q->dequeue (mb)) > 0)
  {
    // Keep track of number of requests
    ++request_count;

    // Identify request and
    // perform service processing here...
  }
  return 0;
}
```

This code forms part of the main event-loop of a network daemon (such as an distributed database for medical images or a distributed file server). In the code, the main event-loop waits for messages to arrive from clients. When a message arrives, the main thread removes it from the message queue via its `dequeue` method. Depending on the type of message that is received, the thread then performs some type of processing (*e.g.,* image database query, file update, etc.). The `request_count` variable keeps track of the number of incoming client requests. This information might be used to update an attribute in an SNMP MIB.

The code shown above works fine as long as `run_svc` executes in a single thread of control. Incorrect results will occur on many multi-processor platforms, however, when `run_svc` is executed simultaneously by multiple threads of control running on different PEs. The problem here is that the code is not "thread-safe." Since auto-increment operations on the global variable `request_count` contain a race condition. Thus, different threads may increment obsolete versions of the `request_count` variable stored in their per-PE data caches.

This phenomenon may be demonstrated by executing the C++ code in Example 1 below on a shared memory multi-processor running the Solaris 2.x operating system. Solaris 2.x allows multiple threads of control to execute in parallel on a shared memory multi-processor. The example shown below is a simplified version of the network daemon illustrated above:

Example 1

```
typedef u_long COUNTER;
static COUNTER request_count; // At file scope

void *run_svc (int iterations)
{
  for (int i = 0; i < iterations; i++)
    ++request_count; // Count # of requests

  return (void *) iterations;
}

typedef void *(*THR_FUNC)(void *);

// Main driver function for the
// multi-threaded server.

int main (int argc, char *argv[])
{
  int n_threads  =
    argc > 1 ? atoi (argv[1]) : 4;
  int n_iterations =
    argc > 2 ? atoi (argv[2]) : 1000000;

  thread_t t_id;

  // Divide iterations evenly among threads.
  int iterations = n_iterations / n_threads;

  // Spawn off N threads to run in parallel.
  for (int i = 0; i < n_threads; i++)
    thr_create (0, 0, THR_FUNC (&run_svc),
                (void *) iterations,
                THR_BOUND | THR_SUSPENDED,
                &t_id);

  // Resume all suspended threads
  // (threads id's are contiguous...)
  for (i = 0; i < n_threads; i++)
    thr_continue (t_id--);

  // Wait for all threads to exit.
  int status;
  while (thr_join (0, &t_id,
                   (void **) &status) == 0)
    cout << "thread id = " << t_id
         << ", status = " << status << endl;

  cout << n_iterations  << " = iterations\n"
       << request_count << " = request count"
       << endl;
  return 0;
}
```

The Solaris `thr_create` thread library routine is called n_thread times to spawn n new threads of control. In this example, each newly created thread executes the `run_svc` function, which is passed the value of `iterations` as its only argument. This value causes the `run_svc` routine to iterate $\frac{n\_iterations}{n\_threads}$ times.

Each thread is spawned using the `THR_BOUND` and `THR_SUSPENDED` flags. `THR_BOUND` informs the Solaris thread run-time library to bind the thread to a dedicated LWP. Each LWP may run in parallel on a separate PE in a multi-processor system. The `THR_SUSPENDED` flag creates each thread in the "suspended" state. This ensures that all threads are completely initialized before resuming the tests by calling `thr_continue`. The `thr_continue` function is a Solaris thread library routine that resumes the execution of suspended threads. Note that this example takes advantage of the fact that thread ids are allocated contiguously by Solaris in ascending order.

Once all threads have been resumed, the `thr_join` routine blocks the execution of the main thread. `thr_join` is similar to the UNIX `wait` system call – it reaps the status of exiting threads. `thr_join` will reap threads and return 0 until all the threads running `run_svc` have exited. When all other threads have exited, the main thread prints out the total number of `iterations` and the final value of `request_count`, and then exits the program.

Compiling this code into an executable `a.out` file and running it on 1 thread for 10,000,000 iterations produces the following results:

```
% a.out 1 10000000
thread id = 4, status = 1000000
10000000 = iterations
10000000 = request count
```

This result appears as expected. However, when executed on 4 threads for 10,000,000 iterations on a 4 PE machine, the program prints the following:

```
% a.out 4 10000000
thread id = 5, status = 1000000
```

```
thread id = 7, status = 1000000
thread id = 6, status = 1000000
thread id = 4, status = 1000000
10000000 = iterations
5000000  = request count
```

Clearly, something is wrong since the value of the global variable request_count is only one-half the total number of iterations. The problem here is that auto-increments on variable request_count are not being serialized properly.

In general, run_svc will produce incorrect results when executed in parallel on shared memory multi-processor platforms that do not provide *strong sequential order* cache consistency models. To enhance performance, many shared memory multi-processors employ *weakly-ordered* cache consistency semantics. For example, the V.8 and V.9 family of SPARC multi-processors provides both *total store order* and *partial store order* memory cache consistency semantics. With total store order semantics, reading a variable that is being accessed by threads on different PEs may not be serialized with simultaneous writes to the same variable by threads on other PEs. Likewise, with partial store order semantics, writes may also not be serialized with other writes. In either case, expressions that require more than a single load and store of a memory location (such as foo++ or i = i - 10) may produce inconsistent results due to cache latencies across multiple PEs. To ensure that reads and writes of variables shared between threads are updated correctly, programmers must manually enforce the order that changes to these variables become globally visible.

A common technique for enforcing a strong sequential order on a *total store order* or *partial store order* shared memory multi-processor is to protect the increment of the request_count variable using a synchronization mechanism. Solaris 2.x provides several synchronization mechanisms. This paper describes C++ wrappers for the four primary synchronization mechanisms provided by Solaris 2.x: *mutexes*, *readers/writer locks*, *counting semaphores*, and *condition variables* [19]. ACE contains C++ wrappers (Mutex, RW_Lock, Semaphore, and Condition) that encapsulate these four Solaris 2.x synchronization mechanisms (mutex_t, rwlock_t, sema_t, and cond_t, respectively). In the remainder of Section 3 we outline the behavior of the Solaris synchronization mechanisms. Section 4 illustrates the use of C++ wrappers to simplify common synchronization variable usage and to improve program reliability.

3.5.1 Mutual Exclusion Locks

Mutex exclusion locks (commonly called "mutexes" or "binary semaphores") are used to protect the integrity of a shared resource that is accessed concurrently by multiple threads of control. A mutex serializes the execution of multiple threads by defining a critical section where only one thread executes its code at a time. Mutexes are simple (*e.g.,* only the thread owning a mutex may release it) and efficient (in terms of time and space).

Operations on mutex variables in operating systems like Solaris 2.x are implemented via adaptive spin-locks. Spin-locks ensure mutual exclusion by using an atomic hardware instruction. A spin-lock is a simple and efficient synchronization mechanism for certain types of short-lived resource contention, like auto-incrementing the global request_count variable illustrated in Example 1 above. An adaptive spin-lock polls a designated memory location using the atomic hardware instruction until one of the following conditions occur [2]:

Mutexes provide an efficient form of mutual exclusion. They define a critical section where only a single thread may execute at a time.

- The value at this location is changed by the thread that currently owns the lock. This signifies that the lock has been released and may now be acquired by the spinning thread.
- The thread that is holding the lock goes to sleep. At this point, the spinning thread also puts itself to sleep to avoid unnecessary polling.

On a multi-processor, the overhead incurred by a spin-lock is relatively minor. Hardware-based polling does not cause contention on the system bus since it only affects the local PE caches of threads that are spinning on a mutex.

A simple and efficient type of mutex is a "non-recursive" mutex. A non-recursive does not allow the thread currently owning a mutex to reacquire the mutex without releasing it first. Otherwise, deadlock will occur immediately. Solaris 2.x and POSIX pthreads implement non-recursive mutexes via the mutex_t data type and the associated mutex_lock and mutex_unlock functions. Win32 does not provide a non-recursive mutex.

Both POSIX pthreads and Win32 threads implement both recursive and non-recursive mutexes (other types of mutexes are discussed in Section 4.4). As described in Section 5.1.4, the ACE OO thread encapsulation library provides the Mutex C++ wrapper to portable implement non-recursive mutex semantics. Non-recursive mutexes are portably implemented in the ACE Recursive_Thread_Mutex class.

3.5.2 Readers/Writer Locks

Readers/writer locks are similar to mutexes. For example, the thread that acquires a readers/writer lock must also release it. Multiple threads may acquire a reader/writer lock simultaneously for reading, but only one thread may acquire the lock for writing. Readers/writer mutexes help to improve concurrent execution when resources protected by the mutex are read far more often than they are written.

Solaris 2.x supports readers/writer mutexes via its rwlock_t type. Neither POSIX pthreads nor Win32 threads support readers/writer locks natively. As described in 5.1.3, the ACE thread library provides a class called RW_Mutex that portably implements the semantics of readers/writer locks within a C++ wrapper class. Both the Solaris and ACE implementations of readers/writer locks gives preference to writers. Thus, if there are multiple readers and a single writer waiting on the lock the writer will acquire it first.

3.5.3 Counting Semaphores

Counting semaphores are conceptually non-negative integers that may be incremented and decremented atomically. If a thread tries to decrement a semaphore whose value equals zero the thread is suspended until another thread increments the semaphore.

Counting semaphores are useful for keeping track of changes in shared program state. They record the occurrence of a particular event Since semaphores maintain state they allow threads make decisions based upon this state, even if the event has occurred in the past.

Semaphores are less efficient than mutexes since they retain additional state and use sleep-locks, rather than spin-locks. However, they more general since they need not be acquired and released by the same thread that acquired them initially. This enables them to be used in asynchronous execution contexts (such as signal handlers).

Solaris 2.x supports for semaphores via its `sema_t` type. Win32 supports semaphores as `HANDLE`s. POSIX pthreads does not support semaphores natively. As described in Section 5.1.2, the ACE thread library provides a class called `Semaphore` that portable implements the semantics of semaphores within a C++ wrapper class.

3.5.4 Condition Variables

Condition variables provide a different flavor of locking than mutexes, readers/writer locks, and counting semaphores. These three other mechanisms make collaborating threads wait while the thread holding the lock executes code in a critical section. In contrast, a condition variable is typically used by a thread to make itself wait until a condition expression involving shared data attains a particular state. When another cooperating thread indicates that the state of the shared data has changed the scheduler wakes up a thread that is suspended on that condition variable. The newly awakened thread then re-evaluate its condition expression and potentially resumes processing if the shared data has attained an appropriate state.

The condition expression waited for by a condition variable may be arbitrarily complex. In general, condition variables permit more complex scheduling decisions, compared with the other synchronization mechanisms. Condition variable synchronization is implemented using sleep-locks, which trigger a context switch and allow another thread to execute until the lock is acquired. As described in Section 3.5.1, mutexes are implemented using adaptive spin-locks. Spin-locks consume excessive resources if a thread must wait a long amount of time for a particular condition to become signaled.

Condition variables are more useful than semaphores or mutexes for situations involving condition expressions semantics. In this case, a waiting thread must block until a certain condition expression involving shared state becomes true (*e.g.,* a list is no longer empty and network flow control abates). In this case, there is no need to maintain event history. Thus, condition variables do not record when they are "signalled." If not used correctly, this can lead to "lost wakeup" problems [19].

Solaris 2.x and POSIX pthreads support condition variables via the `cond_t` type. The native Win32 API does not support condition variables. As described in Section 5.3.1, the ACE thread library provides a class called `Condition` that portably implements the semantics of condition variables within a C++ wrapper class.

3.6 Process vs. Thread Synchronization Semantics

To increase flexibility and improve performance, Solaris 2.x provides two flavors of synchronization semantics that are optimized for either (1) threads that execute within the same process (*i.e.,* intra-process serialization) and (2) threads that execute in separate processes (*i.e.,* inter-process serialization). In Solaris 2.x, the `USYNC_THREAD` flag to the `*_init` functions of the synchronization mechanisms creates variables that are optimized for threads within a single processes. Likewise, the `USYNC_PROCESS` flag creates a synchronization variable that is valid across multiple processes. The latter type of synchronization mechanism is more general, though somewhat less efficient if all threads run within a single process.

3.7 Mutex Example

The following code illustrates how Solaris mutex variables may be used to solve the auto-increment serialization problem we observed earlier with `request_count`:

Example 2

```
typedef u_long COUNTER;
// At file scope
static COUNTER request_count;
// mutex protecting request_count (initialized
// to zero).
static mutex_t m;

void *run_svc (void *)
{

  for (int i = 0; i < iterations; i++)
  {
    mutex_lock (&m); // Acquire lock
    ++request_count; // Count # of requests
    mutex_unlock (&m); // Release lock
  }

  return (void *) iterations;
}
```

In the code above, `m` is a global variable of type `mutex_t`. In Solaris, any synchronization variable that is zero'd out is initialized using its default semantics. For example, the `mutex_t` variable `m` is always initialized to start out in the *unlocked* state. Therefore, the first time that `mutex_lock` is called it will acquire ownership of the lock. Any other thread that attempts to acquire the lock will be forced to wait (*e.g.,* by spinning) until the owner of the lock releases m.

Example 2 shown above solves the original synchronization problem, it suffers from the following drawbacks:

- *Inelegant and inconsistent* – the code mixes C functions with C++ objects, as well as different identifier naming conventions. Using a hybrid programming style is distracting and can become a maintenance problem.

- *Obtrusive* – the solution requires changing the source code. When developing a large software system, manually performing these types of changes leads to maintenance problems if all these changes are not made consistently.

- *Non-portable* – this code will only work with Solaris 2.x synchronization mechanisms. In particular, porting the code to use POSIX pthreads and Windows NT threads will require changing the locking code.

- *Error-prone* – It is easy for programmers to forget to call `mutex_unlock`. This will starve other threads that are trying to acquire the lock. Furthermore, deadlock will occur if the owner of the lock tries to reacquire a mutex it owns already.

  It is also possible that a programmer will forget to initialize the mutex variable. As mentioned above, in Solaris 2.x, a zero'd `mutex_t` variable is implicitly initialized. However, no such guarantees are made for `mutex_t` variables that are allocated as fields in dynamically allocated structures or classes. Moreover, other thread mechanisms (such as POSIX pthreads and Windows NT threads) do not make these guarantees and all synchronization objects must be initialized explicitly.

In Section 4 we will examine how the use of C++ wrappers helps to overcome these problem, by improving the functionality, portability, and robustness of Solaris synchronization mechanisms.

4 Simplifying Concurrent Programming with OO and C++

This section examines a use case to motivate and demonstrate the benefits of encapsulating Solaris concurrency mechanisms within C++ wrappers. This use case depicts a representative scenario that is based upon a production system [25]. Additional examples of the ACE OO thread encapsulation class library appear in Section 6, following the presentation the library interfaces in Section 5.

Many useful C++ classes have evolved incrementally by generalizing from solutions to practical problems that arise during system development. After the interface and implementation of a class have stabilized, however, this iterative process of generalizing classes over time is often deemphasized. That is unfortunate since a major barrier to entry for object-oriented design and C++ is (1) learning and internalizing the process of *how* to identify and describe classes and objects and (2) understanding when and how to apply (or not apply) C++ features such as templates, inheritance, dynamic binding, and overloading to simplify and generalize their programs.

In an effort to capture the dynamics of C++ class design evolution, the following section illustrates the process by which object-oriented techniques and C++ idioms were incrementally applied to solve a surprisingly subtle problem. This problem arose during the development of a family of concurrent distributed applications that execute efficiently on both uni-processor and multi-processor platforms. This section focuses on the steps involved in generalizing from existing code by using templates and overloading to transparently parameterize synchronization mechanisms into a concurrent application. The infrastructure code is based on components in the ADAPTIVE Communication Environment (ACE) framework described in [1, 26, 20].

This example examines several C++ language features that solve the serialization problem presented in Section 3.5.1 more elegantly. As described in that section, the original solution was inelegant, non-portable, error-prone, and required obtrusive changes to the source code. This section illustrates a progression of C++ solutions that build upon insights from prior iterations in the design evolution.

4.1 An Initial C++ Solution

A somewhat more elegant solution to the original problem is to encapsulate the existing Solaris `mutex_t` operations with a C++ `Thread_Mutex` wrapper, as follows:[5]

```
class Thread_Mutex
{
public:
  Thread_Mutex (void) {
    mutex_init (&lock_, USYNC_THREAD, 0);
  }
  ~Thread_Mutex (void) {
    mutex_destroy (&lock_);
  }
  int acquire (void) {
    return mutex_lock (&lock_);
  }
  int release (void) {
    return mutex_unlock (&lock_);
  }

private:
  // Solaris 2.x serialization mechanism.
  mutex_t lock_;
};
```

One advantage of defining a C++ wrapper interface to mutual exclusion mechanisms is that application code now becomes

[5] In this paper, many examples of C++ classes are shown with the methods implemented within the class definition. This style is for exposition purposes only and should not be used when developing applications.

more portable across OS platforms. For example, the following is an implementation of the Thread_Mutex class interface based on mechanisms in the Windows NT WIN32 API [4]:[6]

```
class Thread_Mutex
{
public:
  Thread_Mutex (void) {
    InitializeCriticalSection (&lock_);
  }
  ~Thread_Mutex (void) {
    DeleteCriticalSection (&lock_);
  }
  int acquire (void) {
    EnterCriticalSection (&lock_); return 0;
  }
  int release (void) {
    LeaveCriticalSection (&lock_); return 0;
  }

private:
  // Win32 serialization mechanism.
  CRITICAL_SECTION lock_;
};
```

The use of the Thread_Mutex C++ wrapper class cleans up the original code somewhat, improves portability, and ensures that initialization occurs automatically when a Thread_Mutex object is defined, as shown in the code fragment below:

Example 3

```
typedef u_long COUNTER;
// At file scope.
static COUNTER request_count;
// Thread_Mutex protecting request_count.
static Thread_Mutex m;

void *run_svc (void *)
{
  for (int i = 0; i < iterations; i++)
  {
    m.acquire ();
    // Count # of requests.
    ++request_count;
    m.release ();
  }

  return (void *) iterations;
}
```

However, the C++ wrapper approach does not solve all the problems identified in Section 3.5.1. In particular, it does not solve the problem of forgetting to release the mutex (which still requires manual intervention by programmers). In addition, using class Thread_Mutex still requires obtrusive changes to the original non-thread-safe source code.

4.2 Another C++ Solution

A straightforward way to ensure locks will be released automatically is to use the semantics of C++ class constructors and destructors. The following utility class uses these language constructs to automate the acquisition and release of a mutex:

[6]Note that this implementation only supports mutexes within a single process. ACE also implements process-wide mutexes using other Win32 mechanisms.

```
class Guard
{
public:
  Guard (const Thread_Mutex &m): lock_ (m) {
    lock_.acquire ();
  }
  ~Guard (void) {
    lock_.release ();
  }

private:
  const Thread_Mutex &lock_;
}
```

A Guard defines a "block" of code over which a Thread_Mutex is acquired and then released automatically when the block is exited. It employs a C++ idiom idiom commonly known as "constructor as resource acquisition – destructor as resource release" [9, 27, 7].

As shown in the code above, the constructor of a Guard class acquires the lock on the Thread_Mutex object automatically when an object of the class is created. Likewise, the destructor of a Guard class automatically unlocks the Thread_Mutex object when the object goes out of scope.

Note that the lock_ data member of class Guard is a reference to a Thread_Mutex object. This avoids the overhead of creating and destroying an underlying Solaris mutex_t variable every time the constructor and destructor of a Guard are executed.

By making a slight change to the code, we now guarantee that a Thread_Mutex is acquired and released automatically:

Example 4

```
typedef u_long COUNTER;
// At file scope.
static COUNTER request_count;
// Thread_Mutex protecting request_count.
static Thread_Mutex m;

void *run_svc (void *)
{
  for (int i = 0; i < iterations; i++)
  {
    {
      // Automatically acquire the mutex.
      Guard monitor (m);

      ++request_count;
      // Automatically release the mutex.
    }
    // Remainder of service processing omitted.
  }
}
```

However, this solution still has not fixed the problem with obtrusive changes to the code. Moreover, adding the extra '{' and '}' curly brace delimiter block around the Guard is inelegant and error-prone. A maintenance programmer might misunderstand the importance of the curly braces and remove them, yielding the following code:

```
  for (int i = 0; i < iterations; i++)
  {
    Guard monitor (m);

    ++request_count;
    // Remainder of service processing omitted.
  }
```

Unfortunately, this "curly-brace elision" has the side-effect of eliminating all concurrent execution within the application by serializing the main event-loop. Therefore, all computations that would have executed in parallel within that section of code will be serialized unnecessarily.

4.3 An Improved C++ Solution

To solve the remaining problems in a transparent, unobtrusive, and efficient manner requires the use of two additional C++ features: parameterized types and operator overloading. We use these features to provide a template class called `Atomic_Op`, a portion of which is shown below (the complete interface appears in Section 5.6.2):

```
template <class TYPE>
class Atomic_Op
{
public:
  Atomic_Op (void) { count_ = 0; }
  Atomic_Op (TYPE c) { count_ = c; }
  TYPE operator++ (void) {
    Guard monitor (lock_);
    return ++count_;
  }
  operator TYPE () {
    Guard monitor_ (lock_);
    return count_;
  }
  // Other arithmetic operations omitted...

private:
  Thread_Mutex lock_;
  TYPE  count_;
};
```

The `Atomic_Op` class redefines the normal arithmetic operations (such as ++, −−, +=, etc.) on built-in data types to make these operations work atomically. In general, any class that defines the basic arithmetic operators will work with the `Atomic_Op` class due to the "deferred instantiation" semantics of C++ templates.

Since the `Atomic_Op` class uses the mutual exclusion features of the `Thread_Mutex` class, arithmetic operations on objects of instantiated `Atomic_Op` classes now work *correctly* on a multi-processor. Moreover, C++ features (such as templates and operator overloading) allow this technique to work *transparently* on a multi-processor, as well. In addition, all the method operations in `Atomic_Op` are defined as inline functions. Therefore, an optimizing C++ compiler will generate code that ensures the run-time performance of `Atomic_Op` is no greater than calling the `mutex_lock` and `mutex_unlock` function directly.

Using the `Atomic_Op` class, we can now write the following code, which is almost identical to the original non-thread safe code (in fact, only the typedef of `COUNTER` has changed):

Example 5

```
typedef Atomic_Op<u_long> COUNTER;
// At file scope
static COUNTER request_count;

void *run_svc (void *)
{
  for (int i = 0; i < iterations; i++)
  {
    // Actually calls Atomic_Op::operator++()
    ++request_count;
  }
}
```

By combining the C++ constructor/destructor idiom for acquiring and releasing the `Thread_Mutex` automatically, together with the use of templates and overloading, we have produced a simple, yet remarkably expressive parameterized class abstraction. This abstraction operates correctly and atomically on an infinite family of types that require atomic operations. For example, to provide the same thread-safe functionality for other arithmetic types, we simply instantiate new objects of the `Atomic_Op` template class as follows:

```
Atomic_Op<double> atomic_double;
Atomic_Op<Complex> atomic_complex;
```

4.4 Extending Atomic_Op by Parameterizing the Type of Mutual Exclusion Mechanism

Although the design of the `Atomic_Op` and `Guard` classes described above yield correct and transparently thread-safe programs, there is still room for improvement. In particular, note that the type of the `Thread_Mutex` data member is hard-coded into the `Atomic_Op` class. Since templates are available in C++, this design decision represents an unnecessary restriction that is easily overcome. The solution is to parameterize `Guard` and add another type parameter to the template class `Atomic_Op`, as follows:

```
template <class LOCK>
class Guard
{
// Basically the same as before...

private:
  // new data member change.
  const LOCK &lock_;
};

template <class LOCK, class TYPE>
class Atomic_Op
{
  TYPE operator++ (void)
  {
    Guard<LOCK> monitor (lock_);
    return ++count_;
  }
  // ...

private:
  LOCK lock_; // new data member
  TYPE count_;
};
```

Using this new class, we can make the following simple change at the beginning of the source file:

```
typedef Atomic_Op <Thread_Mutex, u_long> COUNTER;
// At file scope.
COUNTER request_count;

// ... same as before
```

4.5 Design Rationale and Performance Issues

Before making the changes described above, it is worthwhile to analyze the motivation for using templates to parameterize the type of mutual exclusion mechanism used by a program is beneficial. After all, just because C++ supports templates does not make them useful in all circumstances. In fact, parameterizing and generalizing the problem space via templates without clear and sufficient reasons may increase the difficulty of understanding and reusing C++ classes.

The use of templates in the `Atomic_Op` class raises several issues. The first is "what is the run-time performance penalty for all the added abstraction?" The second question is "instead of templates, why not use inheritance and dynamic binding to emphasize uniform mutex interface and to share common code?" The third is "aren't the synchronization properties of the program being obscured by using templates and overloading?" Several of these questions are related and this section discusses tradeoffs involving different design alternatives.

4.5.1 Performance

The primary reason why templates are used for the `Atomic_Op` class involve run-time efficiency. Once expanded by an optimizing C++ compiler during template instantiation, the additional amount of run-time overhead is minimal or non-existent. In contrast, inheritance and dynamic binding incur overhead at run-time in order to dispatch virtual method calls.

Figure 3 illustrates the performance exhibited by the mutual exclusion techniques used in Examples 2 through 5 above.[7] This figure depicts the number of seconds required to process 10 million iterations, divided into 2.5 million iterations per-thread. The test examples were compiled using the `-O4` optimization level of the Sun C++ 3.0.1 compiler. Each test was executed 10 times on an otherwise idle 4 PE Sun SPARCserver 690MP. The results were averaged to reduce the amount of spurious variation (which proved to be insignificant).

Example 2 uses the Solaris `mutex_t` functions directly. Example 3 uses the C++ `Thread_Mutex` class wrapper interface. Surprisingly, this implementation consistently performed better than Example 1, which used direct calls to the underlying Solaris mutex functions. Example 4 uses the `Guard` helper class inside of a nested curly brace block to ensure that the `Thread_Mutex` is released automatically. This version required the most time to execute. Finally, Example 5 uses the `Atomic_Op` template class, which is only slightly less efficient than using the Solaris mutex functions directly. More aggressively optimizing C++ compilers would reduce the amount of variation in the results.

[7] Example 1 is the original erroneous implementation that did not use any mutual exclusion operations. Although it operates extremely efficiently (approximately 0.09 seconds to process 10,000,000 iterations), it produces results that are totally incorrect!

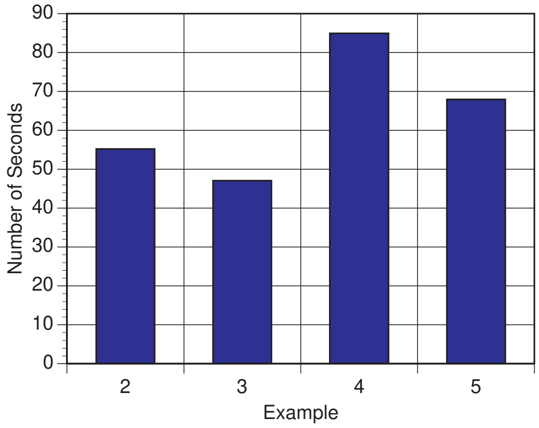

Figure 3: Number of Seconds Required to Process 10,000,000 Iterations

Example	*u*secs per operation	Ratio
Example 2	2.76	1
Example 3	2.35	0.85
Example 4	4.24	1.54
Example 5	3.39	1.29

Table 1: Serialization Time for Different Examples

Table 1 indicates the number of micro-seconds (*u*secs) incurred by each mutual exclusion operation for Examples 2 through 5. Recall that each loop iteration requires 2 mutex operations (one to acquire the lock and one to release the lock). Example 2 is used as the baseline value since it uses the underlying Solaris primitives directly. The third column of Examples 3 through 5 are normalized by dividing their values by Example 2.

4.5.2 Portability

One motivation for parameterizing the type of mutual exclusion mechanism is to increase portability across OS platforms. Templates decouple the formal parameter class name "`Thread_Mutex`" from the actual name of the class used to provide mutual exclusion. This is useful for platforms that already use the symbol `Thread_Mutex` to denote an existing type or function. By using templates, the `Atomic_Op` class source code would not require any changes when porting to such platforms.

However, a more interesting motivation arises from the observation that there are actually several different flavors of mutex semantics one might want to use (either in the same program or across a family of related programs). Each of these mutual exclusion flavors share the same basic acquire and release protocol, but they possess different serialization and performance properties. Section 5.1.1 presents a number of mutual exclusion mechanisms that have proven to be useful in practice.

4.5.3 Transparency

One argument *against* using templates to parameterize synchronization is that the level of transparency hides the mutual exclusion semantics of the program. Whether this is considered a "bug" or a "feature" depends on how one believes that concurrency and synchronization should be integrated into a program. For class libraries that contain basic building-block components (such as the `Map_Manager` described in Section 5.1.1), allowing synchronization semantics to be parameterized is often desirable since this enables developers to *precisely* control and specify the concurrency semantics that they want. The alternatives to this strategy are (1) don't use class libraries if multi-threading is used (which obviously limits functionality), (2) do all the locking outside the library (which may be inefficient or unsafe), or (3) hard-code the locking strategy into the library implementation (which is also inflexible and potentially inefficient). All these alternatives are antithetical to principles of reuse in object-oriented software systems.

4.5.4 Evaluating the Tradeoffs

Selecting an appropriate design strategy for developing a class library that supports concurrency depends on several factors. For example, certain library users may welcome simple interfaces that hide concurrency control mechanisms from view. In contrast, other library users may be willing to accept more complicated interfaces in return for additional control and increased efficiency. A layered approach to class library design may satisfy both groups of library users. Using this design approach, the lowest layers of the class library export most or all of the parameterized types as template arguments. The higher layers provide reasonable default type values and provide an easier-to-use application developer's programming interface.

The new "default template argument" feature recently adopted by the ANSI C++ committee facilitates the development of class libraries that satisfy both types of library users. This feature allows library developers to specify common default types as arguments to template class and function definitions. For example, the following modification to template class `Atomic_Op` provides it with typical default template arguments:

```
template <class LOCK = Thread_Mutex,
          class TYPE = u_long>
class Atomic_Op
{
// Same as before
};

// ...

#if defined (MT_SAFE)
// Default is Thread_Mutex and u_long.
typedef Atomic_Op<> COUNTER;
#else
// Don't serialize.
typedef Atomic_Op<Null_Mutex> COUNTER;
#endif /* MT_SAFE */
COUNTER request_count;
```

Due to the complexity that arises from incorporating concurrency into applications, C++ templates are quite useful for reducing redundant development effort. However, as with any other language feature, it is possible to misuse templates and needlessly complicate a system's design and implementation (not to mention increasing the compile and link times). One heuristic to use when deciding whether to use parameterized types is to keep track of when existing code is about to be duplicated in a way that only modifies the data types. If there is another reasonable scenario that would require yet a third version that only differs by types, this indicates that generalizing the original code to use templates may be worthwhile.

5 Public Interfaces and Internal Design

This section describes the public interfaces and relevant internal design aspects of components in the ACE OO thread encapsulation library. The ACE components are divided into the following groups:

- *Low-level C++ threading APIs* – which provides a low-level C++ wrapper for the underlying OS threading and synchronization APIs. The low-level C++ threading APIs are contained in a class called `OS`. This This class encapsulates all the differences between various versions of UNIX and WIN32. The other components in ACE are programmed to use only the methods in this class, which makes it easier to port ACE to new platforms.
- *High-level C++ threading class* – which allow threaded applications to be programmed using higher-level C++ features like constructors/destructors and templates. The high-level C++ threading APIs are written in terms of the low-level ACE `OS` class. They are divided into three groups:
  - C++ wrappers for locking mechanisms (described in Section 5.1).
  - C++ wrappers for native threading functions (described in Section 5.6.1).
  - Higher-level thread management classes (described in Section 5.4.1).

A use case example of the ACE OO thread components was presented in Section 4. The remainder of this section presents a more comprehensive discussion of the public interfaces, functionality, and internal design of ACE. Where appropriate, this section depicts private portions of the C++ wrapper classes in order to illustrate how the wrappers are mapped onto Solaris 2.x threading and synchronization mechanisms. The implementations of the POSIX pthreads and Win32 wrappers are similar.

5.1 The ACE Locks Class Category

The ACE C++ wrappers provide a portable, type-safe, object-oriented interface to Solaris, POSIX, and Win32 OS synchronization mechanisms described in Section 3.5. The following bullets outline the primary benefits of these ACE C++ wrappers:

• **Improve correctness:** by automating the initialization of synchronization objects that appear as fields in C++ classes and structs, as well as by guaranteeing that locks are acquired and released automatically.

• **Uniform synchronization interface:** all the C++ wrappers for threading and synchronization provide a uniform interface for acquiring and releasing various types of locks. In particular, all components in the ACE Locks class category support four common methods: `acquire`, `try_acquire`, `release`, and `remove`. This uniformity makes it possible to use the lock classes as type parameters in conjunction with other ACE synchronization components (such as those defined in Section 5.2.1, 5.6.2, and 5.1.4).

• **More intuitive error reporting:** the Solaris 2.x and POSIX pthreads synchronization functions use a somewhat non-standard mechanism for returning errors to callers. In contrast, the ACE wrappers use a more standard approach that returns -1 if a failure occurs, along with setting `errno` to indicate the cause of the failure.

• **Simplify common usage pattern:** the wrappers simplify common usage patterns for low-level threading and synchronization mechanisms. The code shown illustrates this point by using the ACE C++ wrappers for `mutex_t` and `cond_t` to implement a simple version of Dijkstra's counting semaphores (*i.e.,* P and V are equivalent to `acquire` and `release`, respectively).

```
class Semaphore
{
public:
  Semaphore (int initial_value)
    : count_nonzero_ (lock_) {
    // Automatically acquire lock.
    Guard<Thread_Mutex> monitor (lock_);

    count_ = initial_value;
   // Automatically release the lock
  }

  // Block the thread until the semaphore
  // count becomes greater than 0,
  // then decrement it.

  void acquire (void) {
    // Automatically acquire lock
    Guard<Thread_Mutex> monitor (lock_);

    // Wait until semaphore is available.
    while (count_ == 0)
      count_nonzero_.wait ();

    count_ = count_ - 1;
    // Automatically release the lock
  }

  // Increment the semaphore, potentially
  // unblocking a waiting thread.

  void release (void) {
    // Automatically acquire lock
    Guard<Thread_Mutex> monitor (lock_);

    // Allow waiter to continue.
    if (count_ == 0)
      count_nonzero_.signal ();

    count_ = count_ + 1;
    // Automatically release the lock
  }

private:
  Thread_Mutex lock_;
  Condition<Thread_Mutex> count_nonzero_;
  u_int count_;
};
```

Note how the constructor for the `Condition` object `count_nonzero_` binds the `Thread_Mutex` object `lock_` together with the `Condition` object. This simplifies the `Condition::wait` calling interface. In contrast, the native Solaris and pthreads `cond_t cond_wait` interface requires a mutex to be passed as a parameter on every call to `wait`.

Solaris 2.x and Win32 provide a built-in implementation of counting semaphores (see the discussion in Section 3.5.3). However, the POSIX Pthreads [3] threads library does *not* include a semaphore. Therefore, the class shown above both illustrates the use of ACE C++ wrappers and documents a portable `Semaphore` implementation for POSIX Pthreads in the ACE thread encapsulation library.

5.1.1 The Mutex Classes

The ACE mutex wrappers provide a simple and efficient mechanism that serializes access to a shared resource. They encapsulate the Solaris and POSIX pthreads `mutex_t` synchronization variable, as well as the Win32 `HANDLE`-based mutex implementation. The class definition for `Mutex` is shown below:

```
class Mutex
{
public:
  // Initialize the mutex.
  Mutex (int type = USYNC_THREAD);

  // Implicitly destroy the mutex.
  ~Mutex (void);

  // Explicitly destroy the mutex.
  int remove (void);

  // Acquire lock ownership (wait
  // for lock to be released).
  int acquire (void) const;

  // Conditionally acquire lock
  // (i.e., don't wait for lock
  // to be released).
  int try_acquire (void) const;

  // Release lock and unblock
  // the next waiting thread.
  int release (void) const;

private:
  mutex_t lock_;
  // Type of synchronization lock.
};
```

In ACE, a thread may enter a critical section by invoking the `acquire` method on a `Mutex` object. Any calls to this method will block until the thread that currently owns the lock has left its critical section. To leave a critical section, a thread invokes the `release` method on the `Mutex` object it currently owns. Calling `release` enables another thread that is blocked on the mutex to enter its critical section.

The `Thread_Mutex` and `Process_Mutex` classes inherit from `Mutex` and use its constructor to create the appropriate type of mutex, as follows:

```
class Thread_Mutex : public Mutex
{
public:
  Thread_Mutex (void): Mutex (USYNC_THREAD);
};

class Process_Mutex : public Mutex
{
public:
  Thread_Mutex (void): Mutex (USYNC_PROCESS);
};
```

These calls are mapped onto the appropriate underlying API for creating thread- and process-specific mutexes, respectively. In particular, the Win32 implementation of `Thread_Mutex` uses the more efficient, but less powerful `CRITICAL_SECTION` implementation, whereas the `Process_Mutex` implementation uses the less efficient, but more powerful Win32 mutex `HANDLE`.

5.1.2 The Semaphore Classes

The ACE semaphore wrappers class implement Dijkstra's "counting semaphore" abstraction, which is a general mechanism for serializing multiple threads of control. They encapsulate the Solaris `sema_t` synchronization variable. The `Semaphore` class interface is shown below:

```
class Semaphore
{
public:
  // Initialize the semaphore,
  // with default value of "count".
  Semaphore (u_int count,
             int type = USYNC_THREAD,
             void * = 0);

  // Implicitly destroy the semaphore.
  ~Semaphore (void);

  // Explicitly destroy the semaphore.
  int remove (void);

  // Block the thread until the semaphore count
  // becomes greater than 0, then decrement it.
  int acquire (void) const;

  // Conditionally decrement the semaphore if
  // count greater than 0 (i.e., won't block).
  int try_acquire (void) const;

  // Increment the semaphore, potentially
  // unblocking a waiting thread.
  int release (void) const;

private:
  sema_t semaphore_;
};
```

The `Thread_Semaphore` and `Process_Semaphore` classes inherit from `Semaphore` and use its constructor to create the appropriate type of semaphore, as follows:

```
class Thread_Semaphore : public Semaphore
{
public:
  Thread_Semaphore (void): Semaphore (USYNC_THREAD);
};

class Process_Semaphore : public Semaphore
{
public:
  Thread_Semaphore (void): Semaphore (USYNC_PROCESS);
};
```

5.1.3 The RW_Mutex Classes

The ACE readers/writer wrappers serialize access to resources whose contents are searched more than they are changed. They encapsulate the `rwlock_t` synchronization variable, which is implemented natively on Solaris and emulated by ACE on Win32 and Pthreads. The `RW_Mutex` interface is shown below:

```
class RW_Mutex
{
public:
  // Initialize a readers/writer lock.
  RW_Mutex (int type = USYNC_THREAD,
            void *arg = 0);

  // Implicitly destroy a readers/writer lock.
  ~RW_Mutex (void);

  // Explicitly destroy a readers/writer lock.
  int remove (void);

  // Acquire a read lock, but
  // block if a writer hold the lock.
  int acquire_read (void) const;

  // Acquire a write lock, but
  // block if any readers or a
  // writer hold the lock.
  int acquire_write (void) const;

  // Conditionally acquire a read lock
  // (i.e., won't block).
  int try_acquire_read (void) const;

  // Conditionally acquire a write lock
  // (i.e., won't block).
  int try_acquire_write (void) const;

  // Unlock a readers/writer lock.
  int release (void) const;

private:
  rwlock_t lock_;
};
```

Note that POSIX Pthreads and Win32 threads do not provide a `rwlock_t` type. To ensure code portability, ACE provides an `RW_Mutex` implementation based on existing low-level synchronization mechanisms such as mutexes and condition variables. In addition, ACE provides `RW_Thread_Mutex` and `RW_Process_Mutex` implementations, as well.

5.1.4 The Recursive_Thread_Mutex Class

A `Recursive_Thread_Mutex` extends the default Solaris non-recursive locking semantics. It allows calls to acquire methods to be nested as long as the thread that owns the lock is the one that re-acquires it. It works with the Thread_Mutex class.

By default, Solaris provides non-recursive mutexes. These semantics are too restrictive in certain circumstances. Therefore, ACE provides support for recursive locks via the Recursive_Thread_Mutex class. Recursive locks are particularly useful for callback-driven C++ frameworks [28, 20], where the framework event-loop performs a callback to user-defined code. Since the user-defined code may subsequently re-enter framework code via a method entry point, recursive locks are useful to prevent deadlock from occurring on locks held within the framework during the callback.

The following C++ class implements recursive lock semantics for the Solaris 2.x synchronization mechanisms (note that POSIX Pthreads and Win32 provide recursive mutexes in their native thread libraries):

```
class Recursive_Thread_Mutex
{
public:
  // Initialize a recursive mutex.
  Recursive_Thread_Mutex (const char *name = 0
                          void *arg = 0);

  // Implicitly release a recursive mutex.
  ~Recursive_Thread_Mutex (void);

  // Explicitly release a recursive mutex.
  int remove (void);

  // Acquire a recursive mutex (will increment
  // the nesting level and not deadmutex if
  // owner of the mutex calls this method more
  // than once).
  int acquire (void) const;

  // Conditionally acquire a recursive mutex
  // (i.e., won't block).
  int try_acquire (void) const;

  // Releases a recursive mutex (will not
  // release mutex until nesting level == 0).
  int release (void) const;

  thread_t get_thread_id (void);
  // Return the id of the thread that currently
  // owns the mutex.

  int get_nesting_level (void);
  // Return the nesting level of the recursion.
  // When a thread has acquired the mutex for the
  // first time, the nesting level == 1.  The nesting
  // level is incremented every time the thread
  // acquires the mutex recursively.

private:
  void set_nesting_level (int d);
  void set_thread_id (thread_t t);

  Thread_Mutex nesting_mutex_;
  // Guards the state of the nesting level
  // and thread id.

  Condition<Thread_Mutex> lock_available_;
  // This is the condition variable that actually
  // suspends other waiting threads until the
  // mutex is available.

  int nesting_level_;
  // Current nesting level of the recursion.

  thread_t owner_id_;
  // Current owner of the lock.
};
```

The following code illustrates the implementation of the methods in the Recursive_Thread_Mutex class:

```
Recursive_Thread_Mutex::Recursive_Thread_Mutex
  (const char *name, void *arg)
  : nesting_level_ (0),
    owner_id_ (0),
    nesting_mutex (name, arg),
    lock_available_ (nesting_mutex_, name, arg)
{
}

// Acquire a recursive lock (will increment
// the nesting level and not deadlock if
// owner of lock calls method more than once).

int
Recursive_Thread_Mutex::acquire (void) const
{
  thread_t t_id = Thread::self ();

  Thread_Mutex_Guard mon (nesting_mutex_);

  // If there's no contention, just
  // grab the lock immediately.
  if (nesting_level_ == 0)
    {
      set_thread_id (t_id);
      nesting_level_ = 1;
    }
  // If we already own the lock,
  // then increment the nesting level
  // and proceed.
  else if (t_id == owner_id_)
    nesting_level_++;
  else
    {
      // Wait until the nesting level has dropped to
      // zero, at which point we can acquire the lock.
      while (nesting_level_ > 0)
        lock_available_.wait ();

      set_thread_id (t_id);
      nesting_level_ = 1;
    }

  return 0;
}

// Releases a recursive lock.

int
Recursive_Thread_Mutex::release (void) const
{
  thread_t t_id = Thread::self ();

  // Automatically acquire mutex.
  Thread_Mutex_Guard mon (nesting_mutex_);

  nesting_level_--;
  if (nesting_level_ == 0)
    // Inform waiters that the lock is free.
    lock_available_.signal ();

  return 0;
}
```

The following is an example of Recursive_Thread_Mutex based on a variation of the Atomic_Op COUNTER presented in Section 4. In the example, Atomic_Op is called by multiple recursive function calls within a single thread:

```
// Counter is a recursive lock.
typedef Atomic_Op<Recursive_Thread_Mutex>
        COUNTER;
// Keep track of the recursion depth.
static COUNTER recursion_depth;

int factorial (int n)
```

```
{
  if (n <= 1) {
    cout << "recursion depth = "
         << recursion_depth << endl;
    return n;
  }
  else {
    // First call acquires lock, subsequent
    // calls increment nesting level.
    recursion_depth++;
    return factorial (n - 1) * n;
  }
}
```

The use of a Recursive_Thread_Mutex prevents deadlock from occurring when the recursion_depth counter is incremented. Although this illustrates recursive lock behavior, it is not a very convincing example. A program executing factorial in multiple threads would produce unpredictable results since recursion_depth is a global various that would be modified serially by multiple threads of control! A more appropriate (and less expensive) locking strategy in this case would use the Thread-Specific Storage pattern [29] described in Section 5.6.4.

5.1.5 The Null_Mutex Class

The Null_Mutex class provides a zero-overhead implementation of the general locking interface shared by the other C++ wrappers for threading and synchronization. The interface and trivial implementation for Null_Mutex is shown below:

```
class Null_Mutex
{
public:
  Null_Mutex (void) {}
  ~Null_Mutex (void) {}
  int remove (void) { return 0; }

  int acquire (void) const { return 0; }
  int try_acquire (void) const { return 0; }
  int release (void) const { return 0; }
};
```

As shown in the code above, the Null_Mutex class implements the acquire and release methods as "no-op" inline functions that are removed completely by a compiler optimizer. Section 6 illustrates the use of the Null_Mutex.

5.1.6 The Token Class

This class provides a more general-purpose synchronization mechanism than Mutexes. For example, it implements "recursive mutex" semantics, where a thread that owns the token can reacquire it without deadlocking. In addition, threads that are blocked awaiting a Token are serviced in strict FIFO order as other threads release the token. In contrast, Mutexes don't strictly enforce an acquisition order.

The interface for the Token class is shown below:

```
class Token
{
public:
  // Initialization and termination.
  Token (const char *name = 0, void * = 0);
  ~Token (void);

  // Acquire the token, sleeping until it is
  // obtained or until <timeout> expires.
  // If some other thread currently holds
  // the token then <sleep_hook> is called
  // before our thread goes to sleep.

  int acquire (void (*sleep_hook)(void *),
               void *arg = 0,
               Time_Value *timeout = 0);

  // This behaves just like the previous
  // <acquire> method, except that it
  // invokes the virtual function called
  // <sleep_hook> that can be overridden
  // by a subclass of Token.
  int acquire (Time_Value *timeout = 0);

  // This should be overridden by a subclass
  // to define the appropriate behavior before
  // <acquire> goes to sleep.  By default,
  // this is a no-op...
  virtual void sleep_hook (void);

  // An optimized method that efficiently
  // reacquires the token if no other threads
  // are waiting.  This is useful for if you
  // don't want to degrad the quality of
  // service if there are other threads
  //  waiting to get the token.
  int renew (int requeue_position = 0,
             Time_Value *timeout = 0);

  // Become interface-compliant with other
  // lock mechanisms (implements a
  // non-blocking <acquire>).
  int tryacquire (void);

  // Shuts down the Token instance.
  int remove (void);

  // Relinquish the token.  If there are any
  // waiters then the next one in line gets it.
  int release (void);

  // Return the number of threads that are
  // currently waiting to get the token.
  int waiters (void);

  // Return the id of the current thread that
  // owns the token.
  thread_t current_owner (void);
};
```

5.2 The ACE Guards Class Category

5.2.1 The Guard Classes

Compared with the C-level mutex APIs, the Mutex wrapper described in Section 5.1.1 provides an elegant interface for synchronizing multiple threads of control. However, Mutex is potentially error-prone since it is possible to forget to call the release method (shown in Section 3.7). This may occur either due to programmer negligence or due to the occurrence of C++ exceptions.

Therefore, to improve application robustness, the ACE synchronization facilities leverage off the semantics of C++ class constructors and destructors to ensure that Mutex locks will be automatically acquired and released. ACE provides a family of classes called Guard, Write_Guard, and Read_Guard that ensure a lock is automatically acquired and released upon entry and exit to a block of C++ code, respectively.

The Guard class is the most basic guard mechanism and is defined as follows:

```
template <class LOCK>
class Guard
{
public:
  // Implicitly and automatically acquire (or try
  // to acquire) the lock.
  Guard (LOCK &l, int block = 1): lock_ (&l)
    {
      result_ = block
        ? acquire () : tryacquire ();
    }

  // Implicitly release the lock.
  ~Guard (void) {
    if (result_ != -1)
      lock_.release ();
  }

  // 1 if locked, 0 if can't acquire lock
  // (errno will contain the reason for this).
  int locked (void) {
    return result_ != -1;
  }

  // Explicitly release the lock.
  int remove (void) {
    return lock_->remove ();
  }

  // Explicitly acquire the lock.
  int acquire (void) {
    return lock_->acquire ();
  }

  // Conditionally acquire the lock (i.e., won't block).
  int tryacquire (void) {
    return lock_->tryacquire ();
  }

  // Explicitly release the lock.
  int release (void) {
    return lock_->release ();
  }

private:
  // Pointer to the LOCK we're guarding.
  LOCK *lock_;

  // Tracks if acquired the lock or failed.
  int result_;
};
```

An object of the Guard class defines a "block" of code over which a lock is acquired and then released automatically when the block is exited. Note that this mechanism will work for the Mutex, RW_Mutex, and Semaphore synchronization wrappers. This illustrates another benefit of using C++ wrappers: they promote interface conformance by adapting gratuitously incompatible interfaces (such as Solaris 2.x semaphores and mutexes).

By default, the Guard class constructor shown above will block until the lock is acquired. There are cases where non-blocking acquire calls are necessary (*e.g.,* to prevent deadlock). Therefore, the ACE Guard constructor can be given a second parameter that instructs it to use the lock's try_acquire method rather than acquire. Callers may then use Guard's locked method to test atomically whether the lock was actually acquired or not.

The Read_Guard and Write_Guard classes have the same interface as the Guard class. However, their acquire methods read locks and write locks, respectively.

5.2.2 The Thread_Control Class

The Thread_Control class is used in conjunction with the Thread_Manager class to automate the graceful termination and cleanup of a thread's activities within its originating function. For example, Thread_Control's constructor stores state information. This information automatically removes the thread from an associated Thread_Manager when the function used to invoke the thread originally terminates. This technique works correctly regardless of (1) which path through the function is executed and (2) whether exceptions are thrown. In this respect, the Thread_Control class behaves like the Guard utility class presented in Section 5.2.1.

The interface for the Thread_Control class is presented below:

```
class Thread_Control
{
public:
  // Initialize the thread control object.
  // If INSERT != 0, then register the thread
  // with the Thread_Manager.
  Thread_Control (Thread_Manager *, int add = 0);

  // Implicitly kill the thread on exit and
  // remove it from its associated Thread_Manager.
  ~Thread_Control (void);

  // Explicitly kill the thread on exit and
  // remove it from its associated Thread_Manager.
  void *exit (void *status);

  // Set the exit status (and return status).
  void *set_status (void *status);

  // Get the current exit status.
  void *get_status (void);
};
```

5.3 The ACE Conditions Class Category

5.3.1 The Condition Class

The Condition class is used to block on a change in the state of a condition expression involving shared data. It encapsulates the Solaris threads and POSIX pthreads cond_t synchronization variable. The Condition class interface is presented below:

```
template <class MUTEX>
class Condition
{
public:
  // Initialize the condition variable.
  Condition (const MUTEX &m,
             int type = USYNC_THREAD,
             void *arg = 0);

  // Implicitly destroy the condition variable.
  ~Condition (void);

  // Explicitly destroy the condition variable.
  int remove (void);

  // Block on condition, or until absolute
  // time-of-day has elapsed.  If abstime
  // == 0 use blocking wait().
  int wait (Time_Value *abstime = 0) const;

  // Signal one waiting thread.
  int signal (void) const;
```

```
  // Signal *all* waiting threads.
  int broadcast (void) const;

private:
  cond_t cond_;
  // Reference to mutex lock.
  const MUTEX &mutex_;
};
```

Note that Win32 does not provide a condition variable abstraction. Therefore, the ACE threading library implements this using other ACE components such as semaphores and mutexes.

5.3.2 The Null_Condition Class

The `Null_Condition` class is a zero-cost implementation of the `Condition` interface described above. It's methods are all implemented as no-ops. This is useful for cases where mutual exclusion is simply not needed (*e.g.,* a particular program or service will *always* run in a single thread of control and/or will not contend with other threads for access to shared resources). The reason for having `Null*` classes is to allow applications to parameterize the type of synchronization they require *without* requiring changes to the application code. The `Null_Condition` class interface is presented below:

```
template <class MUTEX>
class Null_Condition
{
public:
  Null_Condition (const MUTEX &m,
                  int type = 0,
                  void *arg = 0) {}
  ~Null_Condition (void) {}
  int remove (void) { return 0; }
  int wait (Time_Value *abstime = 0) const {
    errno = ETIME; return -1;
  }
  int signal (void) const {
    errno = ETIME; return -1;
  }
  int broadcast (void) const {
    errno = ETIME; return -1;
  }
};
```

The `Null_Condition` class is identical in spirit to the `Null_Mutex` class described in Section 5.1.5.

5.4 The ACE Thread Managers Class Category

5.4.1 The Thread_Manager Class

The `Thread_Manager` class contains a set of mechanisms to manage groups of threads that collaborate to implement collective actions. For example, the `Thread_Manager` class provides mechanisms (such as `suspend_all` and `resume_all`) that allow any number of participating threads to be suspended and resumed atomically. The `Thread_Manager` class also shields applications from many incompatibilities between different flavors of multi-threading mechanisms (such as Solaris, POSIX, and Win32 threads).

The interface of the `Thread_Manager` class is illustrated below:

```
class Thread_Manager
{
public:
  // Initialize the thread manager.
  Thread_Manager (int size);

  // Implicitly destroy thread manager.
  ~Thread_Manager (void);

  // Initialize the manager with room
  // for SIZE threads.
  int open (int size = DEFAULT_SIZE);

  // Release all resources.
  int close (void);

  // Create a new thread.
  int spawn (THR_FUNC,
             long, thread_t * = 0,
             void *stack = 0,
             size_t stack_size = 0);

  // Create N new threads.
  int spawn_n (int n, THR_FUNC,
               void *args, long flags);

  // Clean up when a thread exits.
  void *exit (void *status);

  // Blocks until there are no
  // more threads running.
  void wait (void);

  // Resume all stopped threads.
  int resume_all (void);

  // Suspend all threads.
  int suspend_all (void);

  // Send signum to all stopped threads.
  int kill_all (int signal);

private:
  // ...
};
```

5.4.2 The Thread_Spawn Class

The `Thread_Spawn` class provides a standard utility that manages the creation of threads to handle requests from clients concurrently. This class behaves as a "thread factory", accepting connections from clients and spawning threads "on-demand" to run the service specified by a user-supplied service handler (`SVC_HANDLER`).

The interface for the `Thread_Spawn` class is presented below:

```
template <class SVC_HANDLER,
          class PEER_ACCEPTOR,
          class PEER_ADDR>
class Thread_Spawn
  : public Acceptor <SVC_HANDLER,
                     PEER_ACCEPTOR,
                     PEER_ADDR>
{
public:
  // = Initialization methods.
  Thread_Spawn (Thread_Manager *tm,
                Reactor *);
  virtual int open (const PEER_ADDR &sia,
                    Reactor *);
protected:

  virtual int handle_input (int fd);
```

```
  // Template method that accepts connection
  // and spawns a thread.

  virtual int handle_close (int fd, Reactor_Mask);
  // Called when this factory is closed down.

  virtual SVC_HANDLER *make_svc_handler (void);
  // Factory method that creates an appropriate
  // SVC_HANDLER *.

  virtual int thr_flags (void);
  // Returns the flags used to spawn a thread.
};
```

Note how this classes inherits from the ACE `Acceptor` class, which is a generic factory for passively connecting clients and creating service handlers [30].

5.5 The ACE Active Objects Class Category

5.5.1 The Task Class

The `Task` class is the central mechanism in ACE for creating user-defined *active objects* [16] and *passive objects* that process application messages. An ACE `Task` can perform the following activities:

- Can be dynamically linked;
- Can serve as an endpoint for I/O operations;
- Can be associated with multiple threads of control (i.e., become a so-called "active object");
- Can store messages in a queue for subsequent processing;
- Can execute user-defined services.

The `Task` abstract class defines an interface that is inherited and implemented by derived classes in order to provide application-specific functionality. It is an abstract class since its interface defines the pure virtual methods (`open`, `close`, `put`, and `svc`) described below. Defining `Task` as an abstract class enhances reuse by decoupling the application-independent components provided by the `Stream` class category from the application-specific subclasses that inherit from and use these components. Likewise, the use of pure virtual methods allows the C++ compiler to ensure that a subclass of `Task` honors its obligation to provide the following functionality:

- *Initialization and Termination Methods* – Subclasses derived from `Task` must implement `open` and `close` methods that perform application-specific `Task` initialization and termination activities. These activities typically allocate and free resources such as connection control blocks, I/O handles, and synchronization locks.

  `Tasks` can be defined and used either together with `Modules` or separately. When used with `Modules` they are stored in pairs: one `Task` subclass handles read-side processing for messages sent upstream to its `Module` layer and the other handles write-side processing messages send downstream to its `Module` layer. The `open` and `close` methods of a `Module`'s write-side and read-side `Task` subclasses are invoked automatically by the `ASX` framework when the `Module` is inserted or removed from a Stream, respectively.

- *Application-Specific Processing Methods* – In addition to `open` and `close`, subclasses of `Task` must also define the `put` and `svc` methods. These methods perform application-specific processing functionality on messages. For example, when messages arrive at the head or the tail of a Stream, they are escorted through a series of inter-connected `Tasks` as a result of invoking the `put` and/or `svc` method of each `Task` in the Stream.

  A `put` method is invoked when a `Task` at one layer in a Stream passes a message to an adjacent `Task` in another layer. The `put` method runs *synchronously* with respect to its caller, *i.e.,* it borrows the thread of control from the `Task` that originally invoked its `put` method. This thread of control typically originate either "upstream" from an application process, "downstream" from a pool of threads that handle I/O device interrupts [31], or internal to the Stream from an event dispatching mechanism (such as a timer-driven callout queue used to trigger retransmissions in a connection-oriented transport protocol `Module`).

  If an ACE `Task` executes as a *passive object* (*i.e.,* it always borrows the thread of control from the caller), then the `Task::put` method is the entry point into the `Task` and serves as the context in which `Task` executes its behavior. In contrast, if an ACE `Task` executes as an *active object* the `Task::svc` method is used to perform application-specific processing *asynchronously* with respect to other `Tasks`. Unlike `put`, the `svc` method is not directly invoked from an adjacent `Task`. Instead, it is invoked by a separate thread associated with its `Task`. This thread provides an execution context and thread of control for the `Task`'s `svc` method. This method runs an event loop that continuously waits for messages to arrive on the `Task`'s `Message Queue` (see next bullet).

  Within the implementation of a `put` or `svc` method, a message may be forwarded to an adjacent `Task` in the Stream via the `put_next Task` utility method. `Put_next` calls the `put` method of the next `Task` residing in an adjacent layer. This invocation of `put` may borrow the thread of control from the caller and handle the message immediately (*i.e.,* the synchronous processing approach illustrated in Figure 4 (1)). Conversely, the `put` method may enqueue the message and defer handling to its `svc` method that is executing in a separate thread of control (*i.e.,* the asynchronous processing approach illustrated in Figure 4 (2)). As discussed in [1], the particular processing approach that is selected has a significant impact on performance and ease of programming.

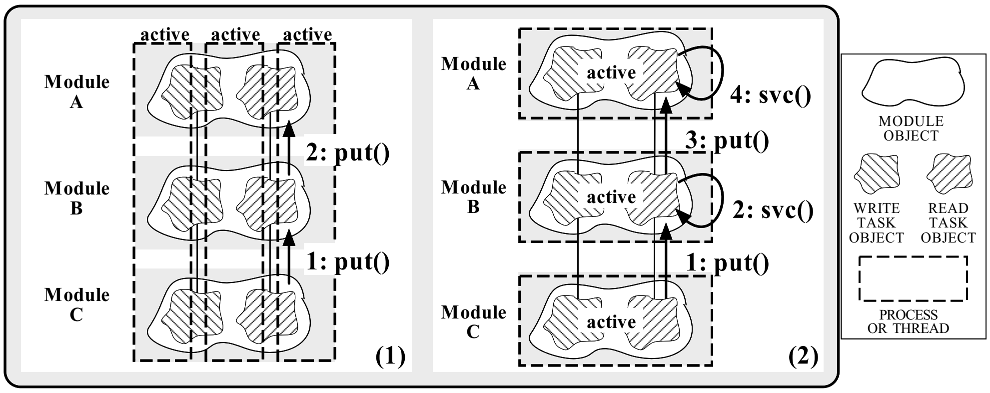

Figure 4: Alternative Methods for Invoking `put` and `svc` Methods

- *Message Queueing Mechanisms* – In addition to the `open`, `close`, `put`, and `svc` pure virtual method interfaces, each `Task` also contains a `Message Queue`. A `Message Queue` is a standard component in ACE that is used pass information between `Tasks`. Moreover, when a `Task` executes as an active object, its `Message Queue` is used to buffer a sequence of data messages and control messages for subsequent processing in the `svc` method. As messages arrive, the `svc` method dequeues the messages and performs the `Task` subclass's application-specific processing tasks.

  Two types of messages may appear on a `Message Queue`: simple and composite. A simple message contains a single `Message Block` and a composite message contains multiple `Message Blocks` linked together. Composite messages generally consist of a *control* block followed by one or more *data* blocks. A control block contains bookkeeping information (such as destination addresses and length fields), whereas data blocks contain the actual contents of a message. The overhead of passing `Message Blocks` between `Tasks` is minimized by passing pointers to messages rather than copying data.

  `Message Queues` contain a pair of high and low water mark variables that are used to implement layer-to-layer flow control between adjacent `Modules` in a Stream. The high water mark indicates the amount of bytes of messages the `Message Queue` is willing to buffer before it becomes flow controlled. The low water mark indicates the level at which a previously flow controlled `Task` is no longer considered to be full.

The interface of the `Task` class is provided below:

```
template <class SYNCH>
class Task : public Service_Object
{
public:
  // Initialization/termination methods.
  Task (Thread_Manager *thr_mgr = 0,
        Message_Queue<SYNCH> *mp = 0);
  virtual int open (void *flags = 0) = 0;
  virtual int close (u_long = 0) = 0;

  // Transfer msg into the queue to handle
  // immediate processing.
  virtual int put (Message_Block *,
                   Time_Value *tv = 0) = 0;

  // Run by a daemon thread to handle
  // deferred processing.
  virtual int svc (void) = 0;

protected:
  // Turn the task into an active object..
  int activate (long flags);

  // Routine that runs the service routine
  // as a daemon thread.
  static void *svc_run (Task<SYNCH> *);

  // Tests whether a message can be enqueue
  // without blocking.
  int can_put (Message_Block *);

  // Insert message into the message list.
  int putq (Message_Block *, Time_Value * = 0);

  // Extract the first message from the list.
  int getq (Message_Block *&, Time_Value * = 0);

  // Return a message to the queue.
  int ungetq (Message_Block *,
              Time_Value * = 0);

  // Transfer message to the adjacent Task
  // in a Stream.
  int put_next (Message_Block *,
                Time_Value * = 0);

  // Turn the message back around.
  int reply (Message_Block *,
             Time_Value * = 0);

  // Task utility routines to identify names.
  const char *name (void) const;
  Task<SYNCH> *sibling (void);
  Module<SYNCH> *module (void) const;

  // Check if queue is a reader.
  int is_reader (void);
  // Check if queue is a writer.
  int is_writer (void);

  // Special routines corresponding to
  // certain message types.
  int flush (u_long flag);

  // Manipulate watermarks.
  void water_marks (IO_Cntl_Msg::IO_Cntl_Cmds,
                    size_t);
```

5.6 Miscellaneous ACE Concurrency Classes

5.6.1 The Thread Class Utility

The `Thread` class utility encapsulates the Solaris, POSIX, and Win32 family of thread creation, termination, and management routines within C++ wrappers. This class provides a common interface that is mapped onto Solaris threads, POSIX Pthreads, Win32 threads.

The interface of the `Thread` class is provided below:

```
typedef void *(*THR_FUNC)(void *);

class Thread
{
public:
  // Spawn N new threads, which execute
  // "func" with argument "arg".
  static int spawn_n (size_t n, THR_FUNC func,
                      void *arg, long flags,
                      void *stack = 0,
                      size_t stack_size = 0);

  // Spawn a new thread, which executes
  // "func" with argument "arg".
  static int spawn (THR_FUNC,
                    void *arg, long,
                    thread_t * = 0,
                    void *stack = 0,
                    size_t stack_size = 0,
                    hthread_t *t_handle = 0);

  // Wait for one or more threads to exit.
  static int join (hthread_t, hthread_t *,
                   void **);

  // Suspend the execution of a thread.
  static int suspend (hthread_t);

  // Continue the execution of a
  // previously suspended thread.
  static int resume (hthread_t);

  // Send signal signum to the thread.
  static int kill (thread_t, int signum);

  // Return the unique ID of the thread.
  static thread_t self (void);

  // Yield the thread to another.
  static void yield (void);

  // Exit current thread, returning "status".
  static void exit (void *status);

  // Set LWP concurrency level of the process.
  static int setconcurrency (int new_level);

  // Get LWP concurrency level of the process.
  static int getconcurrency (void);

  static int sigsetmask (int how,
                         const sigset_t *set,
                         sigset_t *oset = 0);
  // Change and/or examine calling thread's
  // signal mask.

  static int keycreate (thread_key_t *keyp,
                        void (*)(void *value));
  // Allocates a <keyp> that is used to
  // identify data that is specific to each
  // thread in the process. The key is global
  // to all threads in the process.

  static int setspecific (thread_key_t key,
                          void *value);
  // Bind value to the thread-specific data
  // key, <key>, for the calling thread.

  static int getspecific (thread_key_t key,
                          void **valuep);
  // Stores the current value bound to <key>
  // for the calling thread into the location
  // pointed to by <valuep>.
};
```

5.6.2 The Atomic_Op Class

The `Atomic_Op` class transparently parameterizes synchronization into basic arithmetic operations.

```
template <class LOCK, class TYPE>
class Atomic_Op
{
public:
  // Initialize count_ to 0.
  Atomic_Op (void);

  // Initialize count_ to c.
  Atomic_Op (TYPE c);

  // Atomically increment count_.
  TYPE operator++ (void);

  // Atomically increment count_ by inc.
  TYPE operator+= (const TYPE inc);

  // Atomically decrement count_.
  TYPE operator-- (void);

  // Atomically decrement count_ by dec.
  TYPE operator-= (const TYPE dec);

  // Atomically compare count_ with rhs.
  TYPE operator== (const TYPE rhs);

  // Atomically check if count_ >= rhs.
  TYPE operator>= (const TYPE rhs);

  // Atomically check if count_ > rhs.
  TYPE operator> (const TYPE rhs);

  // Atomically check if count_ <= rhs.
  TYPE operator<= (const TYPE rhs);

  // Atomically check if count_ < rhs.
  TYPE operator< (const TYPE rhs);

  // Atomically assign rhs to count_.
  void operator= (const TYPE rhs);

  // Atomically return count_.
  operator TYPE ();

private:
  LOCK lock_;
  TYPE  count_;
};
```

5.6.3 The Barrier Class

The `Barrier` class implements "barrier synchronization," which is particularly useful for many types of parallel scientific applications. This class allows `count` number of threads to synchronize their completion (so-called "barrier synchronization"). The implementation uses a "sub-barrier generation numbering" scheme to avoid overhead and to ensure that all threads exit the barrier correct.

```
class Barrier
{
public:
  // Initialize the barrier to
  // synchronize count threads.
  Barrier (u_int count,
           int type = USYNC_THREAD,
```

```
            void *arg = 0);

  // Block the caller until all count threads
  // have called wait() and then allow all
  // the caller threads to continue in parallel.
  int wait (void);
};
```

5.6.4 The TSS Class

The TSS class allows objects that are "physically" thread-specific (*i.e.,* private to a thread) to be accessed as though they were "logically" global to a program. The underlying "thread specific storage" pattern this class is based upon is described in [29].

The following is the public interface of the ACE TSS class:

```
template <class TYPE>
class TSS
{
public:
  // If caller passes a non-NULL ts_obj *
  // this is used to initialize the
  // thread-specific value.  Thus, calls
  // to operator->() will return this value.
  TSS (TYPE *ts_obj = 0);

  // Get the thread-specific object for the key
  // associated with this object.  Returns 0
  // if the data has never been initialized,
  // otherwise returns a pointer to the data.
  TYPE *ts_object (void);

  // Use a "smart pointer" to obtain the
  // thread-specific object associated with
  // the key.
  TYPE *operator-> ();
};
```

6 Using the ACE OO Thread Encapsulation Library

This section presents several examples that illustrate the use of the key features in the ACE threading library. Refer back to the interfaces described in Section 5 to determine the behavior of the ACE concurrency components.

6.1 A Map Manager for Message Demultiplexing

Often, selecting a mutual exclusion mechanism with the appropriate semantics depends on the context in which a class is being used. The following example illustrates the interface and implementation of a "map manager" component in the general ACE toolkit [32]. This component is typically used in a network server to map external identifiers (such as port numbers or connection ids) onto internal identifiers (such as pointers to queues where messages are stored when outgoing links to a satellite become flow controlled). A portion of the Map_Manager interface and implementation are shown below:

```
template <class EXT_ID,
          class INT_ID,
          class LOCK>
class Map_Manager
{
public:
  // Associate EXT_ID with the INT_ID.
  int bind (EXT_ID ext_id,
            const INT_ID *int_id)
  {
    Write_Guard<LOCK> monitor (lock_);
    // ...
  }

  // Break any association of EXT_ID.
  int unbind (EXT_ID ext_id)
  {
    Write_Guard<LOCK> monitor (lock_);
    // ...
  }

  // Locate INT_ID associated with EXT_ID and
  // pass out parameter via INT_ID.
  // If found return 0, else -1.
  int find (EXT_ID ext_id, INT_ID &int_id)
  {
    Read_Guard<LOCK> monitor (lock_);

    if (locate_entry (ext_id, int_id)
      // ext_id is successfully located.
      return 0;
    else
      return -1;
  }

private:
  LOCK lock_;
  // ...
};
```

One advantage of this approach is that the lock_ will be released regardless of which execution path exits a method. For example, lock_ is released properly if either arm of the if/else statement returns from the find method. In addition, this "constructor/destructor as resource acquisition/release" idiom also properly releases the lock_ if an exception is raised during processing in the definition of the locateEntry helper method. This is useful since the C++ exception handling mechanism is designed to call all necessary destructors upon exit from a block in which an exception is thrown. Note that had we written the definition of find using explicit calls to acquire and release the lock_, *i.e.*:

```
  int find (EXT_ID ext_id, INT_ID &int_id)
  {
    lock_.acquire ();

    if (locateEntry (ext_id, int_id)
    {
      // ext_id is successfully located.
      lock_.release ();
      return 0;
    }
    else
    {
      lock_.release ();
      return -1;
    }
  }
```

the find method logic have been more contorted and less space efficient. In addition, there is no guarantee that lock_ would be released if an exception was thrown in the locateEntry method.

The type of LOCK that the Map_Manager template class is instantiated with depends upon the particular structure of parallelism in the program code when it is used. For example, in some situations it is useful to declare:

```
typedef Map_Manager <Addr, TCB, Mutex>
        MAP_MANAGER;
```

and have all calls to `find`, `bind`, and `unbind` automatically serialized. In other situations, it is useful to turn off synchronization without touching any existing library code by using the `Null_Mutex` class:

```
typedef Map_Manager <Addr, TCB, Null_Mutex>
        MAP_MANAGER;
```

In yet another situation, it may be the case that calls to `find` are *far* more frequent than `bind` or `unbind`. In this case, it may make sense to use the `RW_Mutex` readers/writer lock:

```
typedef Map_Manager <Addr, TCB, RW_Mutex>
        MAP_MANAGER;
```

Through the use of C++ wrappers and templates, we can create a highly-portable, platform-independent mutual exclusion class interface that does not impose arbitrary syntactic constraints on our use of different synchronization mechanisms. By using templates to parameterize the type of locking, little or no application code must change to accommodate new synchronization semantics. As always, however, the selection of an appropriate synchronization mechanism should be guided by thorough profiling and empirical measurements.

6.2 A Thread-safe Message Queueing Mechanism

This example illustrates the use of the ACE `Condition` wrapper 5.1.1 and the ACE `Mutex` wrapper 5.3.1. The code is extracted from the `Message_Queue` class, which is contained in the `Task` class described in Section 5.5.1. A `Message_Queue` can parameterized by the type of synchronization policy needed to achieve the desired level of concurrency control. By default, the level of concurrency control is "thread-safe," as defined by the `MT_Synch` class in the ACE `Synch.h` file:

```
class MT_Synch
{
public:
  typedef Condition<Mutex> CONDITION;
  typedef Mutex MUTEX;
};
```

If `MT_Synch` is used to instantiate `Message_Queue`, all public methods will be thread-safe, with the corresponding overhead that implies. In contrast, if the `Null_Synch` class is used to instantiate `Message_Queue`, all public methods will not be thread-safe, and there will be no additional overhead. `Null_Synch` is also defined in `Synch.h`, as follows:

```
class Null_Synch
{
public:
  typedef Null_Condition<Null_Mutex> CONDITION;
  typedef Null_Mutex MUTEX;
};
```

An example use of `Message_Queue` appeared in the `run_svc` function at the beginning of Section 3.5. `Message_Queue` is modeled after the message queueing and buffer management facilities provided by System V STREAMS [33] and BSD UNIX [34].

An ACE `Message_Queue` is composed of one or more `Message_Blocks` that are linked together by `prev_` and `next_` pointers. In addition, a `Message_Block` may also be linked to a chain of other `Message_Blocks`. This structure enables efficient manipulation of arbitrarily-large messages *without* incurring a great deal of memory copying overhead.

```
// The contents of a message are represented
// internally by a Message_Block.

class Message_Block
{
public:
  Message_Block (size_t size,
                 Message_Type type = MB_DATA,
                 Message_Block *cont = 0,
                 char *data = 0);
  // ...
};
```

A `Message_Queue` is a thread-safe queueing facility for messages. Note the use of the C++ "traits" idiom to combine both the `Condition` and `Mutex` types into a single template parameter.

```
template <class SYNCH = MT_Synch>
class Message_Queue
{
public:
  // Default high and low water marks.
  enum
  {
    // 0 is the low water mark.
    DEFAULT_LWM = 0,
    // 1 K is the high water mark.
    DEFAULT_HWM = 4096,
    // Message queue was active
    // before activate() or deactivate().
    WAS_ACTIVE   = 1,
    // Message queue was inactive
    // before activate() or deactivate().
    WAS_INACTIVE = 2
  };

  // Initialize a Message_Queue.
  Message_Queue (size_t hwm = DEFAULT_HWM,
                 size_t lwm = DEFAULT_LWM);

  // Destroy a Message_Queue.
  ~Message_Queue (void);

  /* Checks if queue is full/empty. */
  int is_full (void) const;
  int is_empty (void) const;

  // Enqueue and dequeue a Message_Block *.
  int enqueue_tail (Message_Block *new_item,
                    Time_Value *tv = 0);
  int enqueue_head (Message_Block *new_item,
                    Time_Value *tv = 0);
  int dequeue_head (Message_Block *&first_item,
                    Time_Value *tv = 0);

  // Deactivate the queue and wakeup all threads
  // waiting on the queue so they can continue.
  int deactivate (void);

  // Reactivate the queue so that threads can
  // enqueue and dequeue messages again.
```

```
  int activate (void);

private:
  // Routines that actually do the enqueueing
  // and dequeueing (assumes locks are held).
  int enqueue_tail_i (Message_Block *);
  int enqueue_head_i (Message_Block *);
  int enqueue_head_i (Message_Block *&);

  // Check the boundary conditions.
  int is_empty_i (void) const;
  int is_full_i (void) const;

  // Implement activate() and deactivate()
  // methods (assumes locks are held).
  int deactivate_i (void);
  int activate_i (void);

  // Pointer to head of Message_Block list.
  Message_Block *head_;
  // Pointer to tail of Message_Block list.
  Message_Block *tail_;
  // Lowest number before unblocking occurs.
  int low_water_mark_;
  // Greatest number of bytes before blocking.
  int high_water_mark_;
  // Current number of bytes in the queue.
  int cur_bytes_;
  // Current number of messages in the queue.
  int cur_count_;
  // Indicates that the queue is inactive.
  int deactivated_;

  // C++ wrapper synchronization primitives
  // for controlling concurrent access.
  SYNCH::MUTEX     lock_;
  SYNCH::CONDITION notempty_;
  SYNCH::CONDITION notfull_;

};
```

The implementation of the Message_Queue class is shown below. The constructor of Message_Queue create an empty message list and initializes the Condition objects. Note that the Mutex lock_ is automatically created by its default constructor.

```
template <class SYNCH>
Message_Queue::Message_Queue (size_t hwm,
                              size_t lwm)
  : notfull_ (lock_),
    notempty_ (lock_)
{
  // ...
}
```

The following methods check if queue is "empty" (*i.e.,* contains no messages) or "full" (*i.e.,* contains more than high_water_mark_ bytes in it). Note how these methods, like the others below, utilize a pattern whereby public methods acquire locks and private methods assume that locks are held.

```
template <class SYNCH> int
Message_Queue<SYNCH>::is_empty_i (void) const
{
  return cur_bytes_ <= 0 && cur_count_ <= 0;
}

template <class SYNCH> int
Message_Queue<SYNCH>::is_full_i (void) const
{
  return cur_bytes_ > high_water_mark_;
}

template <class SYNCH> int
Message_Queue<SYNCH>::is_empty (void) const
{
  Guard<SYNCH::MUTEX> monitor (lock_);
  return is_empty_i ();
}

template <class SYNCH> int
Message_Queue<SYNCH>::is_full (void) const
{
  Guard<SYNCH::MUTEX> monitor (lock_);
  return full ();
}
```

The following methods are used to activate and deactivate a Message_Queue. The deactivate method deactivates the queue and awakens all threads waiting on the queue so they can continue. No messages are removed from the queue. Any other operations called until the queue is activated again will immediately return -1 with errno == ESHUTDOWN. It returns WAS_INACTIVE if queue was inactive before the call and WAS_ACTIVE if queue was active before the call. This information allows the caller to detect changes in state.

```
template <class SYNCH> int
Message_Queue<SYNCH>::deactivate (void)
{
  Guard<SYNCH::MUTEX> m (lock_);
  return deactivate_i ();
}

template <class SYNCH> int
Message_Queue<SYNCH>::deactivate_i (void)
{
  int current_status = deactivated_
    ? WAS_INACTIVE : WAS_ACTIVE;

  // Wake up all the waiters.
  notempty_.broadcast ();
  notfull_.broadcast ();

  deactivated_ = 1;
  return current_status;
}
```

The activate method reactivates the queue so that threads can enqueue and dequeue messages again. It returns WAS_INACTIVE if queue was inactive before the call and WAS_ACTIVE if queue was active before the call.

```
template <class SYNCH> int
Message_Queue<SYNCH>::activate (void)
{
  Guard<SYNCH::MUTEX> m (lock_);
  return activate_i ();
}

template <class SYNCH> int
Message_Queue<SYNCH>::activate_i (void)
{
  int current_status =
    deactivated_ ? WAS_INACTIVE : WAS_ACTIVE;
  deactivated_ = 0;
  return current_status;
}
```

The enqueue_head method inserts a new item at the front of the queue. As with the other enqueue and dequeue methods, if the tv parameter is NULL, the caller will block until action is possible. Otherwise, the caller else will block waiting for amount of time in *tv. A blocked call returns, however, when the queue is closed, when a signal occurs, or if the time specified in tv elapses and errno == EWOULDBLOCK.

```
template <class SYNCH> int
Message_Queue<SYNCH>::enqueue_head
  (Message_Block *new_item, Time_Value *tv)
{
  Guard<SYNCH::MUTEX> monitor (lock_);

  if (deactivated_) {
    errno = ESHUTDOWN;
    return -1;
  }

  // Wait while the queue is full.

  while (is_full_i ()) {
    // Release the lock_ and wait for
    // timeout, signal, or space becoming
    // available in the list.
    if (notfull_.wait (tv) == -1) {
      if (errno == ETIME)
        errno = EWOULDBLOCK;
      return -1;
    }
    if (deactivated_) {
      errno = ESHUTDOWN;
      return -1;
    }
  }

  // Actually enqueue the message at the
  // head of the list.
  enqueue_head_i (new_item);

  // Tell any blocked threads that the
  // queue has a new item!
  notempty_.signal ();
  return 0;
}
```

Note how this method only signals the `notempty_` condition object when the queue was previously empty. This optimization improves performance by reducing the amount of context switching caused by unnecessary signaling. The two other enqueue and dequeue methods perform similar optimizations.

The `enqueue_tail` method inserts new item at the end of the queue. It returns the number of items on the queue.

```
template <class SYNCH> int
Message_Queue<SYNCH>::enqueue_tail
  (Message_Block *new_item, Time_Value *tv)
{
  Guard<SYNCH::MUTEX> monitor (lock_);

  if (deactivated_) {
    errno = ESHUTDOWN;
    return -1;
  }
  // Wait while the queue is full.

  while (is_full_i ()) {
    // Release the lock_ and wait for
    // timeout, signal, or space becoming
    // available in the list.
    if (notfull_.wait (tv) == -1) {
      if (errno == ETIME)
        errno = EWOULDBLOCK;
      return -1;
    }
    if (deactivated_) {
      errno = ESHUTDOWN;
      return -1;
    }
  }

  // Actually enqueue the message at
  // the end of the list.
  enqueue_tail_i (new_item);

  // Tell any blocked threads that
  // the queue has a new item!
  notempty_.signal ();
  return 0;
}
```

The `dequeue_head` method removes the front item on the queue and passes it back to the caller. This method returns a count of the number of items still on the queue.

```
template <class SYNCH> int
Message_Queue<SYNCH>::dequeue_head
  (Message_Block *&first_item, Time_Value *tv)
{
  Guard<SYNCH::MUTEX> monitor (lock);

  // Wait while the queue is empty.

  while (is_empty_i ()) {
    // Release the lock_ and wait for
    // timeout, signal, or a new message
    // being placed in the list.

    if (notempty_.wait (tv) == -1) {
      if (errno == ETIME)
        errno = EWOULDBLOCK;
      return -1;
    }
    if (deactivated_) {
      errno = ESHUTDOWN;
      return -1;
    }
  }

  // Actually dequeue the first message.
  dequeue_head_i (first_item);

  // Tell any blocked threads that
  // the queue is no longer full.
  notfull_.signal ();
  return 0;
}
```

The following code illustrates an ACE implementation of the classic "bounded buffer" program using a `Message_Queue`. The program uses two threads to concurrently copy stdin to stdout. Figure 5 illustrates the relations between the ACE components that run concurrently. The producer thread reads data from the stdin stream, creates a message, and then queues the message in the `Message_Queue`. The consumer thread dequeues the message and writes it to stdout. To save space, most of the error checking has been omitted.

```
#include "Message_Queue.h"
#include "Thread_Manager.h"

typedef Message_Queue<MT_Synch> MT_Message_Queue;

// Global thread manager.
static Thread_Manager thr_mgr;
```

The `producer` reads data from `stdin` into a message and queue the message for the consumer. A `NULL` pointer is enqueued when there is no more data to read. This pointer is used to inform the consumer when to exit.

```
static void *
producer (MT_Message_Queue *msg_queue)
{
  // Insert thread into thr_mgr.
  Thread_Control tc (&thr_mgr);
  char buf[BUFSIZ];

  for (int n; ; ) {
```

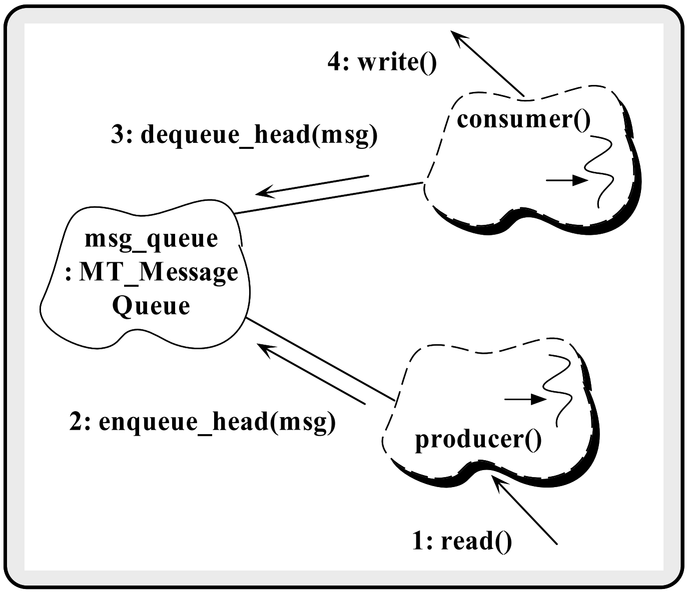

Figure 5: Concurrent Producer and Consumer

```
    // Allocate a new message.
    Message_Block *mb
      = new Message_Block (BUFSIZ);

    n = read (0, mb->rd_ptr (), mb->size ());
    if (n <= 0) {
      // Shutdown message to the other
      // thread and exit.
      mb->length (0);
      msg_queue->enqueue_tail (mb);
    }

    // Send the message to the other thread.
    else {
      mb->wr_ptr (n);
      msg_queue->enqueue_tail (mb);
    }
  }

  // The destructor of Thread_Control removes
  // the exiting thread from the
  // Thread_Manager automatically.
  return 0;
}
```

The consumer dequeues a message from the Message_Queue, writes the message to the stderr stream, and deletes the message. The producer sends a NULL pointer to informs the consumer to stop reading and exit.

```
static void *consumer
  (MT_Message_Queue *msg_queue)
{
  Message_Block *mb = 0;
  // Insert thread into thr_mgr.
  Thread_Control tc (&thr_mgr);
  int result = 0;

  // Keep looping, reading a message out
  // of the queue, until we timeout or get a
  // message with a length == 0, which signals
  // us to quit.

  for (;;)
    {
      result = msg_queue->dequeue_head (mb);

      if (result == -1)
        return -1;

      int length = mb->length ();

      if (length > 0)
        ::write (1, mb->rd_ptr (), length);

      delete mb;

      if (length == 0)
        break;
    }

  // The destructor of Thread_Control removes
  // the exiting thread from the
  // Thread_Manager automatically.
  return 0;
}
```

The main function spawns off two threads that run the producer and consumer functions to copy stdin to stdout in parallel.

```
int main (int argc, char *argv[])
{
  // Use the thread-safe instantiation
  // of Message_Queue.
  Message_Queue msg_queue;

  thr_mgr.spawn (THR_FUNC (producer),
                 (void *) &msg_queue,
                 THR_NEW_LWP | THR_DETACHED);
  thr_mgr.spawn (THR_FUNC (consumer),
                 (void *) &msg_queue,
                 THR_NEW_LWP | THR_DETACHED);

  // Wait for producer/consumer threads to exit.
  thr_mgr.wait ();
  return 0;
}
```

6.3 A Concurrent Network Database Server

The following example illustrates a concurrent network database server developed using the ACE thread management components. Client requests trigger the server to lookup "employees" by their unique numerical ID. If there is a match, the name is returned to the client.

Each client request to the server is run in parallel. This example illustrates the use of the Thread_Manager and the Thread_Control classes. In addition, it also illustrates the use of the ACE C++ wrapper classes for sockets [35].

The code shown below is intentionally simplified for this example and does not represent how a highly robust and efficient implementation would be developed. For example, a production implementation would place an upper-bound on the number of spawned bound threads to avoid consuming large amounts of kernel resources. In addition, a production implementation would clearly use a more sophisticated database scheme.

```
#include "SOCK_Acceptor.h"
#include "Thread_Manager.h"

// Per-process thread manager.
Thread_Manager thr_mgr;

// Function called when a new thread is created.
// This function is passed a connected client
// SOCK_Stream, which it uses to receive a
// database lookup request from a client.
```

```
static void *
lookup_name (ACE_HANDLE handle)
{
  // Local thread control object.
  Thread_Control tc (&thr_mgr);

  enum {
    // Maximum line we'll read from a client.
    MAXLINE = 255,
    // Maximum size of employee name.
    EMPNAMELEN = 512
  };

  // Simple read-only database.
  static struct {
    int emp_id;
    const char emp_name[EMPNAMELEN];
  } emp_db[] =
    {
      123, "John Wayne Bobbit",
      124, "Cindy Crawford",
      125, "O. J. Simpson",
      126, "Bill Clinton",
      127, "Rush Limbaugh",
      128, "Michael Jackson",
      129, "George Burns",
      0, ""
    };

  SOCK_Stream new_stream;
  char recvline[MAXLINE];
  char sendline[MAXLINE];

  new_stream.set_handle (handle);

  ssize_t n = new_stream.recv (recvline, MAXLINE);
  int emp_id = atoi (recvline);
  int found = 0;

  for (int index = 0;
       found == 0 && emp_db[index].emp_id;
       index++)
    if (emp_id == emp_db[index].emp_id) {
      found = 1;
      n = sprintf (sendline, "%s",
                   emp_db[index].emp_name);
    }

  if (found == 0)
    n = sprintf (sendline, "%s", "ERROR");

  new_stream.send_n (sendline, n + 1);
  new_stream.close ();

  // The destructor of Thread_Control removes the
  // exiting thread from the Thread_Manager
  // automatically.
  return 0;
}

// Default port number.
static const int default_port = 5000;

int
main (int argc, char *argv[])
{
  // Port number of server.
  u_short port = argc > 1
    ? atoi (argv[1]) : default_port;

  // Internet address of server.
  INET_Addr addr (port);

  // Passive-mode listener object.
  SOCK_Acceptor server (addr);

  SOCK_Stream new_stream;

  // Wait for a connection from a client
  // (this illustrates a concurrent server).
  for (;;) {
    // Accept a connection from a client.
    server.accept (new_stream);

    // Spawn off a thread-per client request.
    thr_mgr.spawn (THR_FUNC (lookup_name),
                   (void *) new_stream.get_handle (),
                   THR_BOUND | THR_DETACHED);
  }
  // NOTREACHED
  return 0;
}
```

7 Concluding Remarks

This paper motivates and describes the object-oriented thread encapsulation class library provided in ACE. The ACE thread class library provides several benefits to developers:

- Improve the consistency of programming style by enabling developers to use C++ and OO throughout their concurrent applications.
- Reduce the amount of obtrusive changes to make applications thread-safe. For example, utility classes in the library (such as `Atomic_Op`, `Mutex`, `RW_Mutex`, `Semaphore`, and `Condition`) improve the portability and reusability of the underlying OS-specific concurrency mechanisms.
- Eliminate or minimize the potential for subtle synchronization errors. Several of the ACE thread library classes (such as `Guard` and `Thread_Control`) ensure that resources (such as locks and library data structures) are allocated and released properly, even if exceptions occur.
- Enhance abstraction and modularity *without* compromising performance. Using C++ language features (such as inline functions and templates) ensures that the additional functionality provided by the ACE OO thread library does not reduce efficiency significantly.

The ACE OO thread encapsulation library has been used on a number of commercial projects. These products include the Ericsson EOS family of telecommunication switch monitoring applications, Bellcore ATM switch management software, the network management subsystem and core infrastructure subsystem of the Motorola Iridium global personal communications system, and enterprise-wide electronic medical imaging systems at Kodak and Siemens.

References

[1] 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.

[2] 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.

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

[4] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.

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

[6] R. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, vol. 1, pp. 22–35, June/July 1988.

[7] G. Booch, *Object Oriented Analysis and Design with Applications (2^{nd} Edition)*. Redwood City, California: Benjamin/Cummings, 1993.

[8] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.

[9] Bjarne Stroustrup, *The C++ Programming Language, 2^{nd} Edition*. Addison-Wesley, 1991.

[10] W. R. Stevens, *Advanced Programming in the UNIX Environment*. Reading, Massachusetts: Addison Wesley, 1992.

[11] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP Vol III: Client – Server Programming and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1992.

[12] Sun Microsystems, *Open Network Computing: Transport Independent RPC*, June 1995.

[13] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 1.2 ed., 1993.

[14] W. R. Stevens, *UNIX Network Programming, First Edition*. Englewood Cliffs, NJ: Prentice Hall, 1990.

[15] W. R. Stevens, *TCP/IP Illustrated, Volume 1*. Reading, Massachusetts: Addison Wesley, 1993.

[16] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Proceedings of the 2^{nd} Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–7, September 1995.

[17] D. C. Schmidt and C. D. Cranor, "Half-Sync/Half-Async: an Architectural Pattern for Efficient and Well-structured Concurrent I/O," in *Proceedings of the 2^{nd} Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–10, September 1995.

[18] A. D. Birrell, "An Introduction to Programming with Threads," Tech. Rep. SRC-035, Digital Equipment Corporation, January 1989.

[19] Sun Microsystems, Inc., Mountain View, CA, *SunOS 5.3 Guide to Multi-Thread Programming*, Part number: 801-3176-10 ed., May 1993.

[20] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[21] A. Garg, "Parallel STREAMS: a Multi-Process Implementation," in *Proceedings of the Winter USENIX Conference*, (Washington, D.C.), Jan. 1990.

[22] S. Saxena, J. K. Peacock, F. Yang, V. Verma, and M. Krishnan, "Pitfalls in Multithreading SVR4 STREAMS and other Weightless Processes," in *Proceedings of the Winter USENIX Conference*, (San Diego, CA), pp. 85–106, Jan. 1993.

[23] D. C. Schmidt and T. Suda, "Measuring the Performance of Parallel Message-based Process Architectures," in *Proceedings of the Conference on Computer Communications (INFOCOM)*, (Boston, MA), pp. 624–633, IEEE, April 1995.

[24] Mats Bjorkman and Per Gunningberg, "Locking Strategies in Multiprocessor Implementations of Protocols," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (San Francisco, California), ACM, 1993.

[25] D. C. Schmidt, "A Family of Design Patterns for Application-level Gateways," *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)*, vol. 2, no. 1, 1996.

[26] D. C. Schmidt and P. Stephenson, "An Object-Oriented Framework for Developing Network Server Daemons," in *Proceedings of the 2^{nd} C++ World Conference*, (Dallas, Texas), SIGS, Oct. 1993.

[27] G. Booch and M. Vilot, "Simplifying the Booch Components," *C++ Report*, vol. 5, June 1993.

[28] M. A. Linton and P. R. Calder, "The Design and Implementation of InterViews," in *Proceedings of the USENIX C++ Workshop*, November 1987.

[29] T. Harrison and D. C. Schmidt, "Thread-Specific Storage: A Pattern for Reducing Locking Overhead in Concurrent Programs," in *OOPSLA Workshop on Design Patterns for Concurrent, Parallel, and Distributed Systems*, ACM, October 1995.

[30] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Actively and Passively Initializing Network Services," in *Workshop on Pattern Languages of Object-Oriented Programs at ECOOP '95*, (Aarhus, Denmark), August 1995.

[31] N. C. Hutchinson and L. L. Peterson, "The *x*-kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, vol. 17, pp. 64–76, January 1991.

[32] D. C. Schmidt, "The ADAPTIVE Communication Environment: Object-Oriented Network Programming Components for Developing Client/Server Applications," in *Proceedings of the 12^{th} Annual Sun Users Group Conference*, (San Francisco, CA), pp. 214–225, SUG, June 1994.

[33] UNIX Software Operations, *UNIX System V Release 4 Programmer's Guide: STREAMS*. Prentice Hall, 1990.

[34] W. R. Stevens, *TCP/IP Illustrated, Volume 2*. Reading, Massachusetts: Addison Wesley, 1993.

[35] D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, "Object-Oriented Components for High-speed Network Programming," in *Proceedings of the 1^{st} Conference on Object-Oriented Technologies and Systems*, (Monterey, CA), USENIX, June 1995.