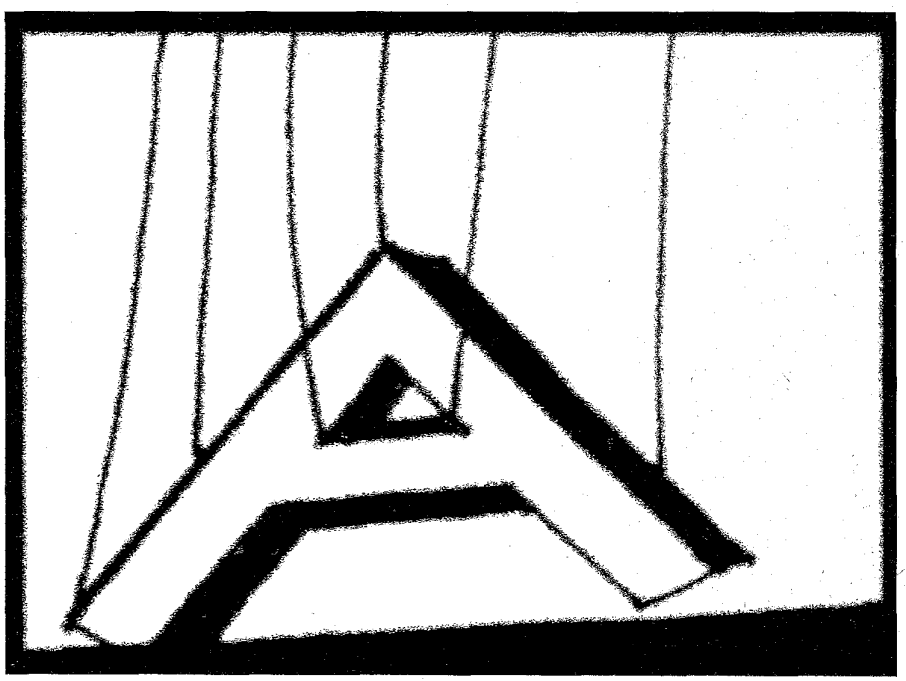
Multithreading Programs: Guidelines for DCE Applications

Multithreading provides a popular mechanism for achieving concurrency, but managing that concurrency can daunt even experienced programmers. The authors offer a tutorial on using threads safely and effectively in an RPC-supported, distributed environment.



DAVID E. RUDDOCK
and BALAKRISHNAN DASARATHY
Bellcore

A concurrency mechanism that is gaining a lot of attention and popularity lately is multithreading — the use of multiple threads or flows of control within a single program. The use of threads enables low system overhead because using multiple threads within a process reduces the number of context switches that the operating system performs on the process. Context switching between two processes is a lot more expensive, in terms of host resources, than context switching between multiple threads within a process.

The use of threads, however, requires an application programmer to manage concurrency or synchronization explicitly. Traditionally, these issues have been the concern of system-program developers. We believe that in most application scenarios a disciplined use of threads can be easily learned and employed, and we show how this can be accomplished.

We have successfully developed multithreaded applications using the Open Software Foundation's Distributed Computing Environment. DCE threads are based on the evolving Posix standards for threads.[1] This article provides an introduction to programming with threads for experienced software developers familiar with languages such as C. Although the focus of our work has been in the DCE arena, the guidelines we define here can be extended to other environments that support multithreading,

0740-7459/96/$05.00 © 1996 IEEE

such as Sun Microsystem's SunOS 5.3 (Solaris) and Windows 95.

DCE supports the remote-procedure-call[2-5] communication-synchronization paradigm across address spaces and multithreading within an address space. The RPC paradigm, which is similar to the procedural-call paradigm within a single address space, can be easily mastered and thus provides a migration path for reengineering centralized systems to distributed systems. Moreover, the RPC paradigm, being synchronous, can lead to better application recovery than would asynchronous paradigms in the case of remote computation failure.

One drawback of synchronous RPCs, however, is that an RPC call causes blocking when executed. Multithreading is introduced to address the blocking issues by allowing the programmer to create multiple threads within an address space that perform multiple operations in a concurrent paradigm. When one thread of a process is waiting on an RPC reply — or any input or output — other threads within the process can be doing useful tasks, which increases an application's throughput. Thus, a threads-and-RPC combination affords the best of both worlds: ease of use, failure recovery, and performance.

Not all applications will benefit from multithreading. For example, performance of CPU-bound applications tends to decrease when multithreaded on a uniprocessor: the model is less efficient on CPU-bound processes because of the threads' additional context switching. However, on a multiprocessor threads scheduler that is supported by the operating system, the increase in true parallelism can make the multithreaded model more efficient. Michael Powell and Steve Kleiman give a more detailed analysis of uniprocessor versus multiprocessor computing and user-level threads versus threads supported by the operating system.[6]

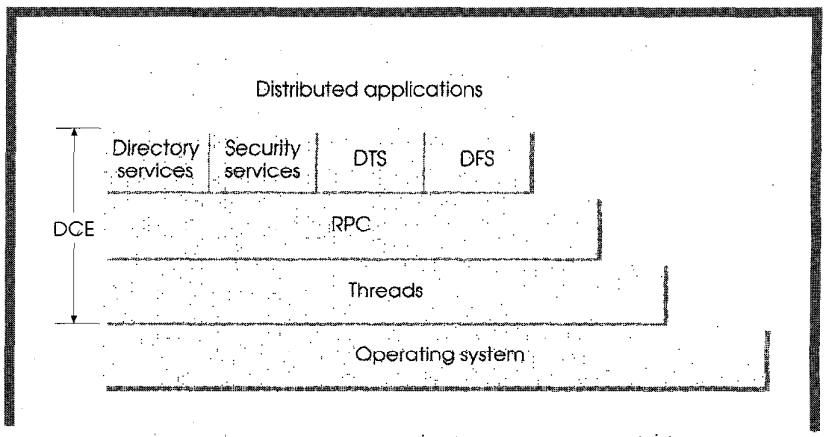


Figure 1. Block diagram of the Open Software Foundation's Distributed Computing Environment. DCE supports the RPC communication-synchronization paradigm across address spaces and multithreading within an address space.

DCE

DCE is a collection of services for the development, deployment, and use of transparent distributed systems using the client-server architecture. Enabling application-level interoperability and portability among heterogeneous platforms through common application programming interfaces is the heart of DCE. It supports the remote-procedure-call synchronous-communication paradigm across address spaces and over various network protocols, with multithreading within an address space for concurrency. DCE uses a directory service and name server to provide client-server location transparency. Directory services are provided within an administration domain, called a cell, and among cells using Domain Name Service and X.500.

DCE security services are based on Kerberos, a security protocol and system developed as part of Project Athena at MIT. It provides a trustworthy, shared-secret authentication system. Its DCE services include authentication of servers and clients, support for resource authorization by an application server in providing services to its clients, and various levels of message integrity and encryption — all at different cost levels. DCE contains two other distributed services: its Distributed File System provides access to files across machines and its Distributed Time Service assists in synchronizing clocks.

Architecturally, DCE lies between the applications and the operating systems and network services. DCE client applications issue a request for service using DCE functions. DCE, in turn, uses the operating system and network services to communicate that request to a server and to communicate the results of the remote computation back to the client. Figure 1 shows a block diagram of DCE. For a more detailed overview of DCE, see the Open Software Foundation's publications.[7,8]

THREADS

The DCE multithreading service allows multiple, simultaneous control flows within a single process or address space. The main advantage of threads is increased throughput by more efficient use of system resources.[9]

Figure 2 shows the difference

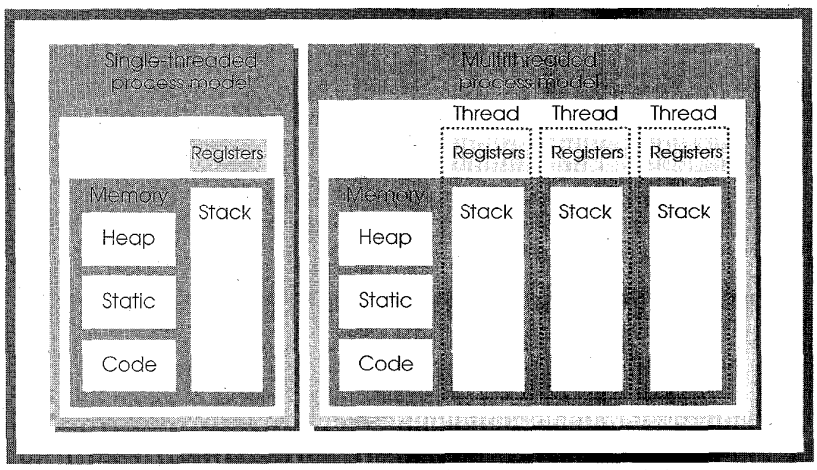


Figure 2. *Comparison of the single-threaded and multithreaded programming models. The multithreaded model has multiple stack-and-register pairs allocated to each thread.*

between the single-threaded and multithreaded programming models. Both have heap-, static-, code-, stack-, and register-memory allocations. The multithreaded model, however, has multiple stack-and-register pairs allocated for each thread. This lets multiple threads have both thread-specific, stack-and-register data and shared, heap-and-static data within a single address space. The data allocated from heap behaves like a stack in that the local data of a procedure is allocated from the top of the heap when the procedure is invoked and the data is released from the top of the heap when the procedure returns. The static data, on the other hand, persists until the return of a procedure. The code segment is shared by all threads within the address space.

By using threads in a client-server computation model, as supported by DCE, server applications can service multiple clients concurrently. DCE servers are multithreaded by default. A client can use threads to make multiple simultaneous requests to a server or to multiple servers. Each thread progresses independently using its own stack-space and register resources, periodically synchronizing with each other and sharing the heap- and static-data process resources as necessary. Some threads continue processing while other threads wait for services such as disk I/O or network-packet reception.

The applicability of threads is not restricted to distributed systems. Consider a communications application that reads from and writes data to an asynchronous communications port in addition to performing other application tasks. Because the reception of data from the port is asynchronous, you cannot know when data will be received. In a single-threaded environment, you would typically use interrupt processing — in which reception of data interrupts the application — or polling the port for data with timed-out read calls, to handle asynchronous input. However, both require multiple operating-system calls that are expensive in a shared-processing environment. You could implement the application using two threads, one to receive asynchronous data and the second to execute other application operations. The thread that reads from the port can continuously read data and pass it along to the other thread for processing. When no data is available to read, the thread will block until more data is received. In the meantime, the other thread continues to execute other application tasks. For a general overview of threads, see Andrew Birrell's *An Introduction to Programming with Threads.*[10]

Threads implementation. A thread implementation can either be in user or kernel (system) space. Currently, many DCE thread implementations are done in user space. As threads become a basic unit for scheduling and resource allocation by an operating system, this will change. For instance, Sun's Solaris operating system supports kernel-level threads. In a user-space threads implementation, threads management is done in user time and the operating system has no control of the threaded environment except to make resources available to the entire process. The management of threads within the process is analogous to the process management within an operating system: scheduling and resource allocation take place, but at the user level. Also, like processes within an operating system, threads have processing states and scheduling policies associated with them.

Threads API. The DCE threads provide a set of primitive function calls that serve as application programming interfaces to create, administer, and synchronize threads within a single address space. These primitive operations can be classified into the following groups:

- Administration includes functions for threads creation, cancellation, priority setting, stack size setting, and clean-up after thread termination. These functions let you tune attributes of a thread to meet specific requirements.
- Synchronization lets multiple threads communicate with each other. Thread synchronization prevents race, deadlock, and priority-inversion conditions.
- Signal-handling catches and sends operating-system signals. These interfaces can be used in the traditional sense to communicate to other processes in a system or to create event-driven applications.
- Thread-specific data store lets a thread have its own version of a global data structure.

To introduce the threads APIs used in this article's examples, in the box that starts on page 84 we summarize the various APIs typically required to realize a threaded application. You may want to read this text first if you have little or no Posix threads experience.

THREADED PROGRAMMING

The examples that follow show when and how to use DCE threads. We adopted the sample code frag-

```
db_access_thread() {
 .......
 pthread_lock_global_np();      /* get global lock */
  open_data_base (db_name);
  first_data_base_operation();
   .......
  last_data_base_operation();
  close_data_base(db_name);
 pthread_unlock_global_np();    /* release global lock */
 .......
}
```

Figure 3. *Example of coarse-grain locking. This code fragment contains a sequence of operations on a database. In this and subsequent figures, threads appear in italic.*

```
#define LOCK(X) if (pthread_mutex_lock (&X) == -1) {   \
 printf ("Error: Can't lock mutex\n");  \
 exit (-1); }
#define UNLOCK(X) if (pthread_mutex_unlock (&X) == -1) {    \
 printf ("Error: Can't unlock mutex\n");          \
 exit (-1); }
 .........
db_access_thread()      {
 pthread_mutex_t        db_mutex;
.........
 LOCK(db_mutex);        /* lock database access */

  open_data_base (db_name);
  first_data_base_operation();
   .......
  last_data_base_operation();
  close_data_base;
 UNLOCK(db_mutex);      /* unlock database access */
}
```

Figure 4. *Example of fine synchronization. The global lock-unlock is replaced with a local mutex lock-unlock known to all threads.*

ments from systems developed using DCE. With each example, we include the rationale for choosing one implementation technique over another. These examples are not optimized and, to enhance readability, may omit variable declarations, mutex and condition variable-initialization routines, and error checking.

Synchronization granularity. A key aspect of threads programming is synchronization among threads for their proper interaction. Synchronization is required when a thread is about to enter a "critical-processing region"; that is, when it needs to lock out other threads from changing a shared resource or wait for some predetermined event to happen.

DCE threads have two synchronization object types: a mutex — short for mutual exclusion, and commonly referred to as a lock — and a condition variable. Mutexes ensure that the integrity of a shared resource is maintained by serializing the thread-access and thread-update functions. Condition variables serve as a signaling mechanism between threads. You should use mutexes for short-term locking, such as serializing updates to a data item, and condition variables for long-term locking, such as waiting for an asynchronous event to occur. We make this generalization because a system's underlying mutex functionality could be of the spin-lock type. A spin-lock mutex would constantly execute, spinning until it obtains a lock. This wastes system resources during relatively long waits.

Threads synchronization, or locking, ranges from fine to coarse. We define fine locking as having a locking mechanism for each shared resource; coarse locking has a single locking mechanism for several resources and for all threads within the process.

Coarse synchronization. You need coarse synchronization during the execution of function and library calls that are not thread-safe. These calls require that all related thread activity be suspended while the controlling thread continues processing. The code in Figure 3 shows coarse-grain locking for a sequence of operations on a database.

The function calls to pthread_lock_global_np and pthread_unlock_global_np lock and unlock a single, process-wide mutex. In Figure 3 and the rest of the examples, calls to the threads' API are highlighted. By introducing this type of locking in a DCE-based server, you can serialize database accesses within the server. Remember, DCE servers are multithreaded by default and process requests as they are received. This implementation has the advantage of being easy to program: the global lock and unlock functions require the addition of only two lines of code around the critical region. The drawback of using the global lock is that all other threads, including nondatabase-related threads, that use the global lock-unlock are not allowed to run during database access, thus reducing concurrency in the process.

Fine synchronization. The use of the global lock in the previous example was convenient in development terms, but not efficient. A fine locking mechanism can be established to make this code more efficient. This involves replacing the global lock-unlock with a local mutex lock-unlock known to all threads that access the database. This lets all database-related threads serialize access while letting other threads run. In Figure 4, the variable db_mutex is associated with the database db_name. The pthread_mutex_lock function call returns immediately after locking the mutex, if the mutex is available, or blocks until a lock can be obtained. The pthread_mutex_lock and pthread_mutex_unlock functions return a -1 if a system or programming error occurs.

Intermediate synchronization. You can use intermediate levels of synchronization to let one mutex protect multiple resources. You would do this to simplify application code while maintaining correct semantic processing. For exam-

ple, a data structure or data object may consist of *n* variables. In intermediate synchronization, you associate just one mutex to provide access to the structure. All threads assigned to access any of the variables must first lock the mutex before making any reads or writes on the object. This preserves data integrity because only one thread can lock the mutex at a time; programming complexity is reduced as well because only one mutex is used instead of *n* mutexes.

Intermediate synchronization reduces the code's length and complex-

CORE APIS FOR DCE THREADS

The format of all Posix-thread APIs is pthread_<object>_<operation>, in which <object> can be an item such as a mutual exclusion — or mutex — and can possibly be null. Function calls that end with the _np suffix are called nonportable. DCE threads are based on Posix 1003.4a Draft 4 standards.[1] Because the Posix document is evolving, the DCE threads are not compatible with the current or final version of the Posix threads standard. To maximize application portability, we identify some incompatibility issues between the Posix and DCE threads.

New threads are created using the pthread_create function call. This routine reserves thread stack space from the heap, assigns attributes to the thread such as priority, and schedules the specified thread to be executed. This function may or may not return before the thread starts processing.

We recommend that thread_attr_default be passed as the thread attribute parameter until the threads package used conforms to the Posix standard. The only exception to this recommendation occurs when dealing with threads that require larger stack size than the default.

Thread synchronization. Mutexes must be initialized before they can be used by any thread. You do this with the pthread_mutex_init function call, with a pointer to a mutex object and a parameter specifying the attributes to give the mutex.

A mutex can be in either a locked or unlocked state and is locked or unlocked by calling the pthread_lock or pthread_unlock functions respectively. Each of these function calls requires a pointer to the mutex to be locked or unlocked. The semantics of the pthread_mutex_trylock function call is similar to the pthread_mutex_lock function call except that the call returns immediately when the mutex is locked. Locked mutexes can only be unlocked by the thread that issued the lock, which is also known as the owner.

Operations on mutexes, in either state, depend on the attributes assigned to each mutex at the time of creation. There are three kinds of mutexes: fast, which is the default; recursive; and nonrecursive.

Mutexes must be deleted from the runtime environment when they are no longer needed by the program. Removal of unneeded mutexes returns memory to the threads manager for other threads to use. Mutexes are deleted using pthread_mutex_destroy.

Condition variables. Sending and receiving notification to other threads that some predefined state has been achieved requires condition variables. The concepts of condition variables and mutexes are similar in nature but are different in practice. Mutexes are good for short-lived locking and unlocking such as variable incrementing and decrementing. Condition variables should be used for longer term conditional locking operations such as "suspend thread processing until variable counter is equal to 7." In the Posix nomenclature, the condition waited for is called the predicate.

Because condition variables are a shared resource, each one must have a mutex associated with it. Recursive mutexes must not be used with condition variables because the implicit unlock function of pthread_cond_wait and pthread_cond_timedwait may not put the mutex in an unlocked state for other threads to lock.

There are two thread functions that a thread can call to block until a condition is raised by another thread:

- pthread_cond_wait waits for a condition to be raised by another thread that shares the same condition variable and mutex pair.
- pthread_cond_timedwait is functionally identical to pthread_cond_wait except that the application can specify a time to stop waiting for the condition and return control to the calling process. The duration of the wait period is specified as an absolute date and time and is usually constructed using the pthread_get_expiration_np function.

These functions must be called from within a loop that tests any predicates before allowing the thread to continue. This is necessary because the Posix threads specification lets the conditional await return unpredictably. Without testing the predicate, a thread may falsely proceed and generate undesired results.

Both of these functions must be passed a locked mutex and an initialized condition variable. The respective call unlocks the mutex before waiting for the condition to be raised. When the condition is raised, the mutex is once again locked and the control is returned to the calling function.

A thread can notify other threads that any predicates are true in one of two ways:

- pthread_cond_signal wakes a randomly selected thread that is waiting on a

ity. Fine synchronization would require a mutex for each variable in the structure. The disadvantage of intermediate synchronization is that a thread assigned to update one variable in the structure must wait while another thread updates a different variable.

In general, the level of synchronization — coarse, intermediate, or fine — should match the expected level of parallelism in the application. You should use high degrees of synchronization when the probability of parallelism among threads is high. You should use decreasing levels of

condition variable. All other threads that are waiting on the condition variable remain blocked.

♦ pthread_cond_broadcast wakes all the threads waiting on a condition variable. The first thread to execute is determined by the threads scheduler.

Two other complementing function calls can be used to synchronize threads: pthread_join and pthread_exit. The pthread_join causes the calling thread to return the exit code of a specified thread in the terminated state. This function causes the calling thread to

♦ Return immediately if the thread specified is in the terminated state.

♦ Block the calling thread until the specified thread terminates.

♦ Fail when the specified thread has been detached. This is because detached threads have had their stack space returned to the process.

You should know that it is likely that the runtime semantics of this function will change in the final version of the Posix standard. Currently, a pthread_join call to a terminated thread returns the exit code of the specified thread and more than one thread is allowed to join to a terminated thread. Newer versions of the Posix standard specify that a pthread_join call will do an implicit pthread_detach on the specified thread in addition to returning the exit code. This means that only one thread can join to a terminated thread. Therefore, you should avoid multiple joins to the same thread.

The pthread_exit function call terminates processing of the calling thread. This function causes the thread to stop processing and stores the exit information for other threads to inspect. The allocated stack space remains in place until pthread_detach is called. This function also pops pending cleanup routines.

Thread cancellation. The DCE thread-cancellation function lets one thread mark itself or another thread for termination. The target thread is allowed to queue cancellation requests and execute predetermined cleanup routines before terminating.

Cancellation is controlled by the threads interrupt-control functions. Threads cancellation requests are processed according to the threads interruptibility state. When enabled, cancellations take place at either interruption points in the thread or when a thread is asynchronously terminated.

Each thread maintains state information on how to process cancellation requests. This information is classified into two categories:

♦ *Interrupt enable.* This information determines if the thread should process or ignore cancellation requests. Cancellation is enabled and disabled by calling the pthread_setcancel function with the CANCEL_ON/CANCEL_OFF parameter. CANCEL_ON is the default behavior for all threads.

♦ *Interrupt type.* This information determines how a thread processes cancellation requests when CANCEL_ON is enabled. When the interrupt enable mode is ENABLE_ON, threads can be canceled either synchronously, which is the default, or asynchronously. A thread can allow or disallow asynchronous cancels by calling the pthread_setasynccancel function with the CANCEL_ON or CANCEL_OFF parameter. A thread is in synchronous cancel mode whenever the asynchronous cancelability is turned off and the cancel mode is CANCEL_ON.

Thread cancellation requests are made by calling the pthread_cancel function with the thread identifier of the thread to cancel. When a cancellation request is issued to a thread in synchronous cancel mode, the threads-management functions mark the target thread for cancellation and terminate the thread at the next cancellation point. A cancellation point is reached when a thread calls either the pthread_setasynccancel, pthread_testcancel, pthread_delay_np, pthread_join, pthread_cond_wait, or pthread_cond_timedwait function calls.

Threads in the asynchronous cancel mode are terminated immediately upon receipt of a cancellation request. Because termination is immediate, the thread should not be performing any operation that would result in an undesired state when it is canceled. For example, the results are nondeterministic if a thread is canceled during a call to the malloc function call. Results of this function-call type are not known because the canceled thread has no way of indicating that the call either failed or succeeded.

Therefore, threads should not be asynchronously canceled when they are acquiring, holding, or updating shared-data structured objects or when they are releasing system resources.

REFERENCE

1. *Threads Extension for Portable Operating Systems, P1003.4a, Draft 4*, IEEE CS Technical Committee on Operating Systems, CS Press, Los Alamitos, Calif., 1990.

```
 1 typedef struct {/* struct is a general mechanism to pass */
 2     handle_t binding_handle; /* several parameters to a
       newly */
 3} thread_args; /* created thread*/
 4 int ctr; /* counter & Number of running threads */
 5 pthread_mutex_t       sync; /* mutex to update structure
       members */
 6 pthread_cond_t sync_cv; /* CV to signal boss thread */
 7 main() {
 8 thread_args t_arg [MAX_THREADS]; /* array of structures */
 9 pthread_t   t_id [MAX_THREADS];   /* array of thread ids */
10   .... /* get server binding info */
11   LOCK(&sync);/* first lock the mutex for the wait call */
12 for ( ctr = 0; ctr <  N; ctr++ ){/*ctr is shared by all
       threads*/
13     t_arg[ctr].binding_handle = servers[ctr];
14     t_arg[ctr].running = &ctr;
15     pthread_create(&t_id[ctr], pthread_attr_default,
16       RPC_FUNCTION, &t_arg[ctr]);
17 }
18 while( ctr != 0)       /* wait for all threads to stop */
19   pthread_cond_wait(&sync_cv, &sync);
20 UNLOCK(&sync);
     ....     /* remainder of application code */
21 RPC_FUNCTION ( arg )
22 thread_args *arg; {
23   ....    /* make RPC call and other operations  */
24 LOCK(&sync);  /* prevent other threads from decrementing */
25 ctr -= 1;   /* decrement running threads count */
26 if(ctr == 0)  /* is this the last thread */
27   pthread_cond_signal (&sync_cv); /* wake up main thread*/
28 UNLOCK(&sync);/* let other worker threads run */
29   pthread_exit(0); /* exit and terminate thread */
30 }
```

Figure 5. *Example of the synchronous boss/worker model. In this code fragment, a boss thread creates* n *worker threads, each of which will execute an RPC call to its respective server. The worker thread that completes the last RPC signals a condition variable to inform the boss thread that work is complete.*

```
typedef struct {
      handle_t binding_handle; /* binding handle to server */
} thread_args;
 main() {
   thread_args  t_arg [MAX_THREADS];     /* array of structures */
   pthread_t    t_id   [MAX_THREADS];    /* array of thread ids */
       .....
   for( ctr = 0 ; ctr < N ; ctr++ ) {   /* create N threads */
         t_arg[ctr].binding_handle = servers[ctr];
         pthread_create (&t_id[ctr], pthread_attr_default,
                     RPC_FUNCTION, &t_arg[ctr]);
   }
   for( ctr = 0 ; ctr < N; ctr++) {
       pthread_join (  t_id[ctr]  );    /* join with thread */
   }
         .....
RPC_FUNCTION(  arg  )
   thread_args *arg; {
       .....          /* make RPC call and other operations */
   pthread_exit(0);
}
```

Figure 6. *Alternative method of boss/worker thread synchronization. This approach simplifies the program by reducing the number of lines of code needed to synchronize the threads.*

synchronization as the probability of parallelism within the process decreases. Finally, you must evaluate design trade-offs between efficiency and code complexity on a case-by-case basis.

Simultaneous multiple threads and RPCs. Threads can be used by applications to execute RPCs in parallel even with one CPU: one thread can continue processing while other threads are blocked waiting for the synchronous RPC to complete. Multiple RPCs and I/Os can also be issued using synchronous I/O multiplexing functions such as the select function call. However, this solution requires that the application manage the synchronous reads and writes and the polling of file descriptors.

Synchronous boss/worker model. A typical computation model for the use of multithreading is the boss/worker model, in which a boss thread assigns tasks to *n* worker threads. The boss may wait for a reply from one, some, or all *n* workers before proceeding. We call the first scenario described below synchronous because the boss thread waits for all *n* workers to complete before continuing.

To comply with the synchronization rules outlined earlier, we use condition variables in this example — waiting for the completion of multiple RPCs could take a relatively long time.

In the code fragment shown in Figure 5, a boss thread creates *n* worker threads, each of which will execute an RPC call to its respective server. The worker thread that completes the last RPC signals a condition variable to inform the boss thread that work is complete. Figure 4 shows the definitions for the LOCK and UNLOCK macros. The list of binding handles for servers is contained in the array servers. The loop index variable ctr is known to all related threads because it is a global variable.

This code shows several key points. The function RPC_FUNCTION executes simultaneously as *n* separate threads. Figure 2 shows that each instance of local variables within each thread exists on a separate stack space. The code segment for the function is shared among threads. This allows multiple simultaneous execution of the same function in the same address space with different arguments. When one thread blocks on an RPC call, or

```
 1 struct ARGS {
 2        handle_t       binding_handle;
 3 }       t_arg[MAX_THREADS];
 4 LOCK(done);
 5 started = 0;
 6 for( ctr = 0; ctr < N; ctr++ ) {
 7       t_arg[ctr].binding_handle = servers[ctr];
 8         pthread_create(&t_id[ctr], pthread_attr_default,
 9         RPC_FUNCTION, &t_arg[ctr]);
10 }
11 started++; /* tell threads all threads started */
12 pthread_cond_broadcast( &start_cv );/* tell threads to start
*/
13 while ( first_server == NULL )
14 pthread_cond_wait( &done_cv, &done );
15 UNLOCK( done );
16 for (ctr = 0; ctr < N; ctr++ )  {    /* clean up threads
                                        space */
17        pthread_cancel( t_id[ctr] ); /* kill the threads */
18        pthread_detach( &t_id[ctr] ); /* clean up space */
19 }
20 .... /* remainder of application */
21 RPC_FUNCTION ( arg )
22 struct ARGS arg;
23 {
24 if( started != 1 ) {  /* wait until all threads created */
25 LOCK( start );
26 while( started != 1 )
27        pthread_cond_wait( &start_cv, &start );
28 UNLOCK( start );
29 }
30 pthread_setasynccancel( CANCEL_ON );
31 rpc_mgmt_is_server_listening( arg->binding_handle, &status);
32 LOCK(done); /* prevent other threads from
                changing first_server variable */
33 if ((status == error_status_ok) && (first_server == NULL)){
34 first_server = arg->binding_handle;
35 pthread_cond_signal( &done_cv );
36 }
37 UNLOCK(done);
38 pthread_exit(0);
39 }
```

Figure 7. *Example of the asynchronous boss/worker model. The code selects an active server from a list of potential servers in a DCE-based application by creating a thread for each potential server that sends a call to each respective server. Binding information from the first server that responds is passed back to the application.*

is preempted by the threads scheduler, another thread is assigned CPU time.

Each thread requires its own separate arguments structure as declared on line 8. Multiple structures are required because the arguments for each thread must persist after the pthread_create call. The use of multiple structures would not be necessary in a single-threaded model.

The RPC_FUNCTION protects both the running counter decrement in line 25 and the test for zero in line 26. This avoids a race condition between one thread setting the variable to zero and another thread reading a zero value, or two threads reading the same value and decrementing one from the same value, such as losing an update. The order of lock, decrement, test, and unlock guarantees that the thread that set the variable to zero reads the value as zero.

The program uses the pthread_cond_wait call on line 19 and the predicate test of ctr != 0 to determine when all threads have completed. Predicate tests are required when using the pthread_cond_wait and the pthread_cond_timedwait routines because the Posix specification does not specify rigid return criteria for these functions: Either of these calls can return at any time, not just when the application signals it to return. The pthread_cond_wait and pthread_cond_timedwait calls unlock the associated mutex before waiting for the respective condition to be raised. When the condition is raised, the mutex is once again locked and control is returned to the calling function.

This example uses a medium-level synchronization because the mutex sync used for the condition variable sync_cv is also used to serialize access to the variable *ctr*.

Figure 6 shows that you could also achieve synchronization by using the pthread_join routine to wait for each thread to complete. This approach simplifies the program by reducing the number of lines of code needed to synchronize the threads. We provided the previous example to show the interrelationships between the various threads' APIs.

Asynchronous boss/worker model. Some applications will have requirements similar to those shown in the previous section, except that the boss thread only waits until the first RPC completes before resuming execution. For example, a client can have a set of service providers, generally of the read-only type, that supply an identical service such as replicated directory. In such a case, a client can locate an active server by either polling each, one at a time, using the single-threaded model, or by polling all the servers at almost the same time using the multithreaded model. The search in the multithreaded model is called off as soon as a server responds.

Figure 7 shows a code fragment that selects an active server from a list of potential servers in a DCE-based application. It does this by creating a thread for each potential server that sends an rpc_mgmt_is_server_listening call to each respective server. Binding information from the server that responds first is passed back to the application using the first_server global variable. The remaining RPC threads are canceled. The servers to contact are contained in the array servers. This code illustrates several important points:

- On line 30, asynchronous cancellation of each worker thread is enabled so that each thread can be canceled by the boss thread on line 17 after the first RPC returns.
- On line 27, asynchronous cancellation is enabled in each worker thread

```
struct THREAD_ARGS { /* global definition */
    int mode;       /* synchronous/asynchronous flag */
     pthread_mutex_t sync; /* mutex to prevent race condition */
   pthread_cond_t    sync_cv; /* condition variable */
};
BossFunction() {
struct THREAD_ARGS *p;     /* pointer to the thread arguments*/
struct timespec add, limit;   /* structures to track time */
p = (struct THREAD_ARGS *) malloc(sizeof(THREAD_ARGS));
p->mode = 0;          /* initialize to synchronous mode */
LOCK(&(p->sync));           /* Lock the mutex */
pthread_create(&thread_id, pthread_attr_default,
          DoStudy,  p);   /* create new thread*/
add.tv_sec = 2; /* set number of seconds to wait */
add.tv_nsec = 0;/* and zero nanoseconds */
pthread_get_expiration_np(&add,&limit); /*expiration time */
  /* sync_cv is a globally declared condition variable */
if(pthread_cond_timedwait(&(p->sync_cv),&(p->sync),&limit)!= 0 )
          p->mode = 1;      /* RPC is now asynchronous */
UNLOCK(&(p->sync));
return 1;
}
....
void DoStudy( struct THREAD_ARGS *p) {
   .... /* make RPC call                  and other operations*/
  LOCK(&(p->sync ));
   if ( p->mode == 1 ) {   /* did RPC exceed time limit */
   .... /* do Asynchronous processing; e.g., report RPC results
           in a separate widget */
   }
   else {
     .... /* do Synchronous processing */
     pthread_cond_signal( &sync_cv );   /* wake boss thread*/
   }
  UNLOCK( &(p->sync) );
pthread_exit(0);
}
```

Figure 8. *Example of simulating asynchronous RPC calls. From the application's perspective, the RPC appears synchronous when it completes in 2 seconds and asynchronous when it exceeds 2 seconds.*

```
#define READ 0    /* Label for the read end of the pipe      */
#define WRITE 1   /* Label for the write end of the pipe     */
 int gui_pipe[2]; /* declare pipe variable                   */
   ........       /* process initialization                  */
pipe( gui_pipe ); /* create pipe                             */
XtAddInput(gui_pipe[READ],  XtInputReadMask,  CB,  NULL);
 .......          /* more main() thread processing           */
XtMainLoop();     /* start the event loop                    */
exit(0);          /* exit main loop of X                     */
}
/* This thread will be called by the application and will run */
/* with its own stack space. When the Get_remote_data returns */
/* a message will be passed to the event loop manager which   */
/* in turn calls the call back function to perform the        */
/* X related functions that are not thread-safe.              */
Send_RPC_Thread() {          /* thread that sends RPC         */
 .......                     /* init code \xll                */
 Get_remote_data();          /* send the RPC call             */
 write( gui_pipe[WRITE],"done", 4); /* write string to pipe   */
 pthread_exit(0);            /* exit the thread               */
}                            /* end of the Boss thread        */
/* the call back function (CB) will be called by the event    */
/* manager and executed as part of the main() thread          */
CB( W, c_data, cb_data ) { /* CB routine to process worker info*/
   ........       /* declare variables. */
 read( gui_pipe[READ], some_line, 4); /* remove data from pipe */
   .......     /* Perform X related functions as main() thread */
}
```

Figure 9. *Example of making a program thread-safe by ensuring that the main thread performs all the X-related processing and that the remaining threads, using the interprocess-communications pipes mechanism, communicate to the X event-loop manager as if they were separate processes.*

after the pthread_cond_wait call. This prevents the deadlock condition of canceling a thread that is holding a shared resource; in this case, the shared resource is the start mutex.

- The first_server variable updates once and only once because of the initial-condition test done on line 33. Without the initial-condition test, a race condition would exist between other worker threads that want to update the variable and the boss thread, which wants to terminate the search. Also, without the test another worker thread can update the first_server variable before the boss thread gets a chance to terminate the remaining active threads.
- The pthread_join function cannot be used because it is not known which thread will complete first.
- The code fragment in Figure 7 can easily be modified if the boss thread must wait for completion of the first K of *n* threads or the first K of *n* threads that return with some "success" replies. In Figure 7, K = 1.

Simulating asynchronous RPC calls. The main form of communication among processes in DCE is synchronous RPC: the calling party waits until it gets a reply from the called party or until the RPC runtime returns a communication failure. Asynchronous RPCs can be simulated using threads and timed condition waits. An asynchronous RPC facilitates a calling process to wait for a reply only for a specific amount of time. If no reply is received during that time, the calling process proceeds with its execution. It has the option of either receiving the reply in background mode or ignoring the reply.

For example, consider an application with a graphical user interface. We want the GUI to have a maximum blocking time of 2 seconds. The application's RPC-round-trip time, the time from the send to the receive, ranges from 500 milliseconds to 4 seconds. To meet the blocking requirement, a boss

thread creates a worker thread to make the RPC call. In this case, the boss thread is the main thread of the GUI. The boss thread can use the pthread_cond_timedwait either to wait for the RPC to complete or for the 2-second wait to expire. In either case, the GUI does not block for more than 2 seconds. Figure 8 shows that, from the application's perspective, the RPC appears synchronous when it completes within 2 seconds and asynchronous when it exceeds 2 seconds.

For readability, we omit the predicate test required around the pthread_cond_timedwait and the initialization of the mutex and condition variables.

The arguments to the worker thread must be allocated from the heap and not from the stack. This is necessary because automatic variables declared in the boss thread would be invalid in the worker thread once the boss times-out and returns to its calling function.

```
server_thread(client_input. server reply)
char *client_input, *server_reply; {
        int fd[2], n;
        socketpair(AF_UNIX,
        SOCK_STREAM, 0, fd);
        if(fork()) { /* parent */
                close( fd[1]);
                write( fd[0], client_input, strlen(client_input));
                n = read (fd[0], server_reply, MAXLINE);
                server_reply[n] = '\0'; /* null terminate string
*/
                return;
        }
        else {  /* child */
                close(fd[0]);
                if (fd[1] != STDIN_FILENO)
                        dup2(fd[1], STDIN_FILENO);
                if (fd[1] != STDOUT_FILENO)
                        dup2(fd[1], STDOUT_FILENO);
                execlp("PROGRAM_NAME", "PROGRAM_NAME", NULL);
        }
}
```

Figure 10. *Example of porting an existing application to DCE. The code is used by the server thread to create a separate address space that runs the nonreentrant server code. A biderictional pipe does the input and output of the new process.*

INTEGRATING THREADS

As of X11, Release 5, X-based programs are not thread-safe. However, you can take steps to let multiple control flows exist in a process that is not thread-safe. You can do this by making sure that only one thread, usually the main thread, performs all the X-related processing and that the remaining threads, using the interprocess-communications pipes mechanism, communicate to the X event-loop manager as if they were separate processes.

Figure 9 shows a code snip that accomplishes this. All X-related operations are performed as part of the main thread and the RPC call is issued from a separate thread. The RPC thread and the main thread are connected with a unidirectional IPC pipe. The read end of the pipe is connected to the event-loop manager using the XtAddInput function call and the write end of the pipe is recognized by the RPC thread because it is a global variable. When the RPC is issued, the thread sending the call will block and the main thread can continue to accept user input. When the RPC completes, a string of characters is written to the pipe. The writing of this data will cause the X-event-loop manager to call the call-back routine specified during the XtAddInput function call. This call-back routine performs the application operations on the main thread in a thread-safe fashion.

Porting existing applications to DCE. Often, we must convert existing applications to DCE. The introduction of parallelism increases the complexity of the port because most applications are written in the single-threaded paradigm. To be thread-safe, the server code must be rewritten to be reentrant. This rewrite converts global and static variables to a local scope and inserts locking and synchronization steps where necessary to protect shared data. Usually the client code will not change much because of the client's generally low degree of parallelism.

The costs of converting a server program to support concurrency increase with its size and complexity. Also, many third-party products, such as database-management and GUI products, do not support concurrent processing. Therefore, you must separate the processes that are not thread-safe from those that are. You do this by having the server thread execute a fork and exec to create a new address space to execute the non-reentrant program. Communications between the new process and the server thread can be done using any interprocess-communications facilities available to the application.

Figure 10 shows the code used by the server thread to create a separate address space that runs the nonreentrant server code. A bidirectional pipe does the input and output to the new process. The parent process writes the data to the pipe and then reads the response from the pipe. The child process reads and writes data using stdin and stdout.

This implementation suffers from the disadvantage that a separate

process is created for each client request. However, you must weigh the additional use of system resources against the costs of converting a nonreentrant program to be thread-safe.

Large data structures. A thread's stack space can quickly be used up by declaring large automatic data structures or arrays in a thread. Diminished stack space can cause the thread to exceed its allocated stack and cause undesirable results. To reduce stack-space problems, memory allocations for large objects such as data structures and arrays should be taken from the heap. This decreases the chances of a thread running out of stack space.

The heap can also run out of space when there is either too much demand for space or the memory is mismanaged. However, this situation can be easily detected and gracefully managed by the application. The same cannot be said for a thread running out of stack space because a segmentation violation signal is generated and the program either terminates or the signal manager has no indication of which thread ran out of stack space.

Currently, the absence of debuggers for threaded programs poses a major problem when writing distributed applications. In the absence of a commercially supported multithreaded debugger, we offer some practical advice on developing client-server-based distributed applications:

- First, develop the interface specification.
- Test and debug the server and client as a single-threaded, single-address-space program. The server is a subroutine and the client is a main program.
- Modify this stand-alone program as a single-threaded server program for all critical regions, using pthread_lock_global_np and pthread_unlock_global_np constructs and a single-threaded client program. Test them as client and server by linking with IDL compiler-generated stubs and header files.
- Multithread the server and then the client, preferably in that order.

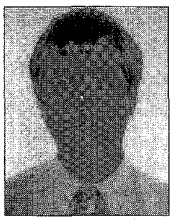
Breaking the development into these steps facilitates the principle of separation of concerns. Specifically, within an address space program-logic concerns are isolated from distribution concerns, which in turn are dealt with separately from concurrency issues. ◆

ACKNOWLEDGMENTS

We thank Maurice Lampell, Gary Levin, Bob Robillard, Diane Ruddock, and John Unger, all of Bellcore, for reviewing this document.

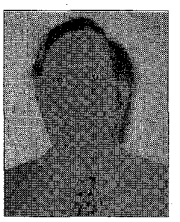
REFERENCES

1. "Threads Extension for Portable Operating Systems," P1003.4a, Draft 4, Technical Committee on Operating Systems of the IEEE CS Press, Los Alamitos, Calif., 1990.
2. A.D. Birrell and B.J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Computer Systems*, Jan. 1984, pp. 39-59.
3. H.E. Bal et al., "Programming Languages for Distributed Computing System," *ACM Computing Surveys*, Mar. 1989, pp. 261-322.
4. K. Ravindran and S.T. Chanson, "Failure Transparency in Remote Procedure Calls," *IEEE Trans. Computers*, Aug. 1989, pp. 1173-1187.
5. W. Richard Stevens, *Unix Network Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1990.
6. M.L. Powell et al., "SunOS Multi-Thread Architecture," *Proc. Usenix*, Usenix Assoc., Berkeley, Calif., 1991, pp. 65-79.
7. *Introduction to DCE, Rev. 1.0*, Prentice-Hall, Englewood Cliffs, N.J., 1993.
8. *OSF DCE Application Development Guide, Rev. 1.0*, Prentice-Hall, Englewood Cliffs, N.J., 1993.
9. B. Dasarathy, K. Khalil, and D.E. Ruddock, "Some DCE Performance Analysis Results," *Proc. DCE Workshop*, Springer-Verlag, Berlin, 1993, pp. 47-62.
10. A.D. Birrell, *An Introduction to Programming with Threads*, Systems Research Center, Digital Equipment Corp., Palo Alto, Calif., 1989.



David E. Ruddock has 11 years experience developing software solutions at Bellcore. For the last four years he has worked on the design and development of multithreaded, object-oriented, distributed systems. Before joining Bellcore, he worked at Bell Telephone Laboratories.

Ruddock received an MS in computer science from the Stevens Institute of Technology and a BS in electrical engineering from the New Jersey Institute of Technology.



Balakrishnan "Das" Dasarathy has been, for the past five years, a consulting engineer on several Bellcore Workstation Software Factory products and has been applying distributed-system technologies, including DCE and Encina, to a variety of Bellcore products and systems in several domains. Prior to joining Bellcore, Dasarathy worked for Concurrent Computer Corporation and GTE Labs. He recently accepted a position from J.P. Morgan as vice president and technical architect to direct some of that company's platform work for trading applications.

Dasarathy received a PhD in computer and information science from Ohio State University. He is a senior member of the IEEE and a member of ACM.

Address questions about this article to Ruddock at Bellcore, PY4-4N305, Piscataway, NJ 08854; der@cc.bellcore.com.