

High-performance concurrent communication development in UNIX using the ACE library framework

Skill Level: Intermediate

[Arpan Sen \(arpan@syncad.com\)](mailto:arpan@syncad.com)

Lead Engineer

Synapti Computer Aided Design Pvt Ltd

30 Jun 2009

The ACE open source toolkit helps developers create robust, portable multithreading applications. Take a peek into some of the ways you can create applications that use ACE threads.

The Adaptive Communication Environment (ACE) is a high-performance, open source, object-oriented framework and C++ class library that simplifies the development of network applications. The ACE toolkit is a combination of an operating system layer and a collection of C++ facades that encapsulate the network API. This article shows how to design high-performance, concurrent, object-oriented networking applications using ACE threads. For a complete description of ACE, including how to download and install the toolkit, see [Resources](#).

ACE classes for creating and managing threads

The following classes are involved in spawning and managing multiple threads within a process:

- **ACE_Thread_Manager:** This is the main class responsible for creation, management, and synchronization of threads. Every operating system has its own subtleties in handling threads: This wrapper class makes these vagaries opaque to the application developer.
- **ACE_Sched_Params:** You use this class to manage the scheduling

priority of individual threads, which is defined as part of the `ace/Sched_Params.h` header in the ACE source distribution. Scheduling policies vary and could be round robin or first come, first served.

- **ACE_TSS:** Using global variables in a multithreaded application is a synchronization issue. The `ACE_TSS` class provides the thread-specific storage pattern, providing an abstraction of data that is logically global to the program but is in reality private to every thread. The `ACE_TSS` class provides the `operator()` method, which in turn provides the thread-specific data.

Understanding the thread manager class

Native operating system threading APIs are not portable: Both syntactic and semantic differences exist. For example, UNIX® `pthread_create()` and Windows® `CreateThread()` methods create a thread, but with syntactic differences. The `ACE_Thread_Manager` class comes with the following capabilities:

- It can spawn one or more threads, each running its own designated function.
- It can manage related threads as a cohesive collection, known as a *thread group*.
- It manages the scheduling priority of individual threads.
- It allows for synchronization between threads.
- It may alter thread attributes such as stack size.

The salient methods of the `ACE_Thread_Manager` class are described in [Table 1](#).

Table 1. ACE_Thread_Manager class methods

Method name	Description
<code>instance</code>	The <code>ACE_Thread_Manager</code> class being a singleton, this method is used to access the unique copy of the Thread Manager.
<code>spawn</code>	This method creates a new thread, an input argument to it being the C/C++ function pointer that does the application-designated work.
<code>exit</code>	This method exits a thread and releases all the thread's resources.
<code>spawn_n</code>	This method creates multiple threads belonging to the same thread group.
<code>close</code>	This method closes down and releases resources for all the created threads.

suspend	Given a thread ID, the thread manager suspends the thread.
resume	The thread manager resumes the thread suspended earlier.

Usage variants of the ACE_Thread_Manager class

You can use the `ACE_Thread_Manager` class as a singleton or make multiple instantiations of the class. For the singleton, you use a call to the `instance` method for access. In situations where you need to manage multiple thread groups, it is useful to have different thread-manager classes, each controlling its own set of threads.

Look at the example in [Listing 1](#), which creates a thread.

Listing 1. Creating a thread using the ACE_Thread_Manager class

```
#include "ace/Thread_Manager.h"
#include <iostream>

void thread_start(void* arg)
{
    std::cout << "Running thread..\n";
}

int ACE_TMAIN (int argc, ACE_TCHAR* argv[])
{
    ACE_Thread_Manager::instance()->spawn((ACE_THR_FUNC)thread_start);
    return 0;
}
```

[Listing 2](#) shows the prototype for the `spawn()` method, sourced from `ace/Thread_Manager.h`.

Listing 2. The ACE_Thread_Manager::spawn method prototype

```
int spawn (ACE_THR_FUNC func,
           void *arg = 0,
           long flags = THR_NEW_LWP | THR_JOINABLE |
THR_INHERIT_SCHED,
           ACE_thread_t *t_id = 0,
           ACE_hthread_t *t_handle = 0,
           long priority = ACE_DEFAULT_THREAD_PRIORITY,
           int grp_id = -1,
           void *stack = 0,
           size_t stack_size =
ACE_DEFAULT_THREAD_STACKSIZE,
           const char** thr_name = 0);
```

The sheer number of arguments needed to make just a basic thread might appear overwhelming to the first-time user, so let's take a closer look at the individual arguments and why they are needed:

- **ACE_THR_FUNC func:** This function is called upon when the thread is spawned.
- **void* arg:** An argument to the function called upon when the thread is spawned, `void*` implies that the user is free to pass in any application-specific data type or even coalesce multiple arguments to a single data using a structure.
- **long flags:** Several properties of the spawned thread are set using the `flags` variable. The individual attributes are `bitwise` or `-ed`. [Table 2](#) shows some of the attributes.
- **ACE_thread_t *t_id:** Use this function to access the ID of the created thread. Every thread has a unique ID.
- **long priority:** This is the priority at which the threads are spawned.
- **int grp_id:** If provided, this argument indicates whether the spawned thread is to be part of some existing thread group. Otherwise, if `-1` is passed as the argument, a new thread group is created, and the spawned thread is added to the same.
- **void* stack:** This is the pointer to the preallocated stack area. If `0` is provided as the argument, the operating system is requested to provide for the stack area of the spawned thread.
- **size_t stack_size:** This argument indicates the size of the thread stack in bytes. If `0` is provided as the argument for `stack_pointer` (item 7), the operating system is requested for a stack area with size `stack_size`.
- **const char** thr_name:** This argument is relevant only in platforms like VxWorks that have an ability to name threads. It is ignored for most purposes on UNIX systems.

Table 2. Thread properties and their description

Thread creation flag	Description
THR_CANCEL_DISABLE	Do not allow the thread to be cancelled.
THR_CANCEL_ENABLE	Allow the thread to be cancelled.
THR_DETACHED	Create an asynchronous thread. The exit status of the thread would not be available to any other threads. The thread resources are reclaimed by the operating system whenever the thread is terminated.
THR_JOINABLE	Allow the new thread's exit status to be available to other threads. This is also the default attribute in ACE-created threads. When this kind of thread dies, the resources are not reclaimed by the operating system until some other thread joins

	with it.
THR_NEW_LWP	Create an explicit kernel-level thread (as opposed to a user-level thread).
THR_SUSPENDED	Create a new thread in the suspended state.

[Listing 3](#) provides an example that creates multiple threads using the `spawn_n` method of the thread manager class.

Listing 3. Creating multiple threads using the `ACE_Thread_Manager` class

```
#include "ace/Thread_Manager.h"
#include <iostream>

void print (void* args)
{
    int id = ACE_Thread_Manager::instance()->thr_self();
    std::cout << "Thread Id: " << id << std::endl;
}

int ACE_TMAIN (int argc, ACE_TCHAR* argv[])
{
    ACE_Thread_Manager::instance()->spawn_n(
        4, (ACE_THR_FUNC) print, 0, THR_JOINABLE |
        THR_NEW_LWP);

    ACE_Thread_Manager::instance()->wait();
    return 0;
}
```

Alternative thread-creation mechanism in ACE

This section discusses an alternate thread-creation/management scheme from ACE, one that does not need the explicit level of fine-grained control of the thread manager. By default, every process is created with a single thread that starts and ends with the `main` function. Any extra threads need explicit creation. An alternative way of creating threads is to create a sub-class of the predefined `ACE_Task_Base` class, then override the `svc()` method. The new thread starts in the `svc()` method and ends when the `svc()` method returns. Before explaining further, take a look into the source shown in [Listing 4](#).

Listing 4. Creating a thread using `ACE_Task_Base::svc`

```
#include "ace/Task.h"

class Thread_1 : public ACE_Task_Base {
public:
    virtual int svc( ) {
        std::cout << "In child's thread\n";
        return 0;
    }
};
```

```
int main ( )
{
    Thread_1 th1;
    th1.activate(THR_NEW_LWP|THR_JOINABLE);
    th1.wait();
    return 0;
}
```

The application-specific behavior of the thread is coded inside the `svc()` method. To execute the thread, the `activate()` method (declared and defined as part of the `ACE_Task_Base` class) is called upon. When the thread is activated, the `main()` function waits for the child thread to complete execution. This is what the `wait()` method strives to achieve: The main thread is blocked until `Thread_1` is complete. This wait is a needed feature; otherwise, the main thread would have scheduled the child thread and made an exit. The C run time, on seeing the main thread exiting, would have destroyed all the child threads; as such, the child thread would probably never have been scheduled or executed.

Understanding ACE_Task_Base class in more detail

Here's a somewhat detailed look into a few of the methods in `ACE_Task_Base`.

[Listing 5](#) shows the `activate()` method prototype.

Listing 5. The ACE_Task_Base::activate method prototype

```
virtual int activate (long flags = THR_NEW_LWP |
    THR_JOINABLE | THR_INHERIT_SCHED,
                    int n_threads = 1,
                    int force_active = 0,
                    long priority =
    ACE_DEFAULT_THREAD_PRIORITY,
                    int grp_id = -1,
                    ACE_Task_Base *task = 0,
                    ACE_hthread_t thread_handles[ ] =
    0,
                    void *stack[ ] = 0,
                    size_t stack_size[ ] = 0,
                    ACE_thread_t thread_ids[ ] = 0,
                    const char* thr_name[ ] = 0);
```

You can use the `activate()` method to create one or more threads of control, each of which calls the same `svc()` method, and all running at the same priority with the same group ID. Here's a brief description of some of the input arguments:

- **long flags:** Refer to [Table 2](#).
- **int n_threads:** Create the number of threads specified by `n_threads`.
- **int force_active:** If this flag is True and there are existing threads

spawned by the task, then all the newly spawned threads would share the group ID of the previously spawned threads and ignore the value as passed to the `activate()` method.

- **long priority:** This argument assigns a priority to the thread or collection of threads. Scheduling priority values are operating system specific, and it's a safe bet to stick to the default value of `ACE_DEFAULT_THREAD_PRIORITY`.
- **ACE_hthread_t thread_handles:** If `thread_handles != 0`, then after the spawning of the n threads, this array will be assigned the individual thread handles.
- **void* stack:** If specified, this argument indicates an array of pointers to the base of the stacks for the individual threads.
- **size_t stack_size:** If specified, this argument indicates an array of integers denoting the size of the individual thread stacks.
- **ACE_thread_t thread_ids:** If `thread_ids != 0`, it is assumed to be an array that will hold the IDs of the n newly spawned threads.

[Listing 6](#) shows a couple of other useful routines in the `ACE_Task_Base` class.

Listing 6. Miscellaneous routines from `ACE_Task_Base`

```
// Block the main thread until all threads of this task
// are completed
virtual int wait (void);

// Suspend a task
virtual int suspend (void);

// Resume a suspended task.
virtual int resume (void);

// Gets the no. of active threads within the task
size_t thread_count (void) const;

// Returns the id of the last thread whose exit caused the
// thread count
// of this task to 0. A zero return status implies that
// the result is
// unknown. Maybe no threads are scheduled.
ACE_thread_t last_thread (void) const;
```

In order to create a thread in a suspended state (as opposed to a suspension explicitly using the `suspend()` method call), you need to pass the `THR_SUSPENDED` flag to the `activate()` method. You can resume the thread by calling the `resume()` method, as shown in [Listing 7](#).

Listing 7. Suspension and resumption of threads

```
Thread_1 th1;
th1.activate(THR_NEW_LWP|THR_JOINABLE|THR_SUSPENDED);
...// code in the main thread
th1.resume();
...// code continues in main thread
```

A second look into thread flags

Two kinds of threads are possible: kernel-level threads and user-level threads. If the `activate()` method is called without any argument, the default is to create a kernel-level thread. Kernel-level threads interact directly with the operating system and are scheduled by the kernel-level scheduler. In contrast, user-level threads operate within the process scope and are "assigned" kernel-level threads on an as-needed basis to complete some task. The flag `THR_NEW_LWP`, which is the default argument to the `activate()` method, always ensures that the new thread created is a kernel-level thread.

Thread hooks

ACE provides a global thread startup hook that allows the user to perform any kind of operation that is globally applicable to all threads. To create a startup hook, you need to create a subclass of the predefined class `ACE_Thread_Hook` and provide the definition for the `start()` method. The `start()` method accepts two arguments: a pointer to a user-defined function and a `void*`, which is passed to this user-defined function. In order to register the hook, you need to call the static method `ACE_Thread_Hook::thread_hook`, as shown in [Listing 8](#).

Listing 8. Using global thread hooks

```
#include "ace/Task.h"
#include "ace/Thread_Hook.h"
#include <iostream>

class globalHook : public ACE_Thread_Hook {
public:
    virtual ACE_THR_FUNC_RETURN start (ACE_THR_FUNC func,
void* arg) {
        std::cout << "In created thread\n";
        (*func)(arg);
    }
};

class Thread_1 : public ACE_Task_Base {
public:
    virtual int svc( ) {
        std::cout << "In child's thread\n";
        return 0;
    }
};

int ACE_TMAIN (int argc, ACE_TCHAR* argv[])
{
    globalHook g;
```

```
ACE_Thread_Hook::thread_hook(&g);  
Thread_1 th1;  
th1.activate();  
th1.wait();  
return 0;  
}
```

Note that the `ACE_THR_FUNC` pointer that is automatically passed to the startup hook is the same function that would have been called when the `activate()` method of the thread is executed. The outputs from the above code are the messages:

```
In created thread  
In child's thread
```

Conclusion

This article took a first look at the creation and management of threads using the ACE framework. The ACE framework has several other useful features, such as mutexes, guard blocks for synchronization, shared memory, and network services. See [Resources](#) for details.

Resources

Learn

- [Building and Install ACE](#): Learn how to build and install the ACE library.
- [ACE support](#): Get commercial support for the ACE library.
- [AIX and UNIX](#): Visit the developerWorks AIX and UNIX zone provides a wealth of information relating to all aspects of AIX systems administration and expanding your UNIX skills.
- [New to AIX and UNIX?](#) Visit the New to AIX and UNIX page to learn more.
- [Technology bookstore](#): Browse for books on this and other technical topics.

Get products and technologies

- [ACE framework](#): Download the ACE library framework.

Discuss

- [developerWorks blogs](#): Check out developerWorks blogs and get involved in the [developerWorks community](#).
- Participate in the AIX and UNIX forums:
 - [AIX Forum](#)
 - [AIX Forum for developers](#)
 - [Cluster Systems Management](#)
 - [IBM Support Assistant Forum](#)
 - [Performance Tools Forum](#)
 - [Virtualization Forum](#)
 - More [AIX and UNIX Forums](#)

About the author

Arpan Sen

Arpan Sen is a lead engineer working on the development of software in the electronic design automation industry. He has worked on several flavors of UNIX, including Solaris, SunOS, HP-UX, and IRIX as well as Linux and Microsoft Windows for several years. He takes a keen interest in software performance-optimization techniques, graph theory, and parallel computing. Arpan holds a post-graduate degree in software systems. You can reach him at arpan@syncad.com.

Trademarks

UNIX is a registered trademark of The Open Group in the United States and other countries.

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.