Proceedings of the IASTED International Conference
Parallel and Distributed Computing and Systems
November 3-6, 1999 in Cambridge Massachusetts, USA

A C++ Thread Package for Concurrent and Parallel Programming

Jie Chen*and William Watson III
High Performance Computing Group
Thomas Jefferson National Accelerator Facility
12000, Jefferson Avenue
Newport News, VA 23606
USA
{*chen,watson*}*@jlab.org*

Abstract

Recently thread libraries have become a common entity on various operating systems such as Unix, Windows NT and VxWorks. Those thread libraries offer significant performance enhancement by allowing applications to use multiple threads running either concurrently or in parallel on multiprocessors. However, the incompatibilities between native libraries introduces challenges for those who wish to develop portable applications. This paper presents a C++ library that is implemented efficiently on various operating systems, offers uniform and simple interfaces to underlying thread libraries, and allows applications to be portable and easy to develop.

Keywords: Thread, concurrent and parallel programming, object-oriented system.

1 Introduction

Disciplined concurrent programming can improve the structure and performance of applications on both uniprocessor and multiprocessor machines. As a result, support of *threads*, or lightweight processes, has become a common entity of operating systems and programming languages.

A thread is an independent flow of control within a process, composed of a context and a sequence of instructions to execute. A thread differs from a traditional notion of a heavyweight process in that it separates the notion of execution from the other state needed to run a program. A single thread executes a portion of a program, while cooperating with other threads that are concurrently executing the same program. A multi-threaded program usually reduces complexity of a program by managing information that is normally kept on a per-heavyweight-process basis in common. On a multiprocessor machine, a multi-threaded application may have better performance if multiple executions can be carried out in parallel on more than one processors. Even on a uniprocessor machine, better response time and increased throughput can be achieved by overlapping computation and communication [1].

Although threads have been around for decades, the use of threads only started to become popular in the 1990s. During this time, multiprocessor system, client/server applications, real-time systems and parallel processing were making their way into daily computing. Almost all operating systems for shared memory multiprocessor and uniprocessor support concurrent programming with thread libraries such as POSIX Pthreads [2], SunOS LWP and threads [3] and Windows Threads [4]. However, thread libraries differ in various aspects such as user threads or kernel threads, bounded threads or unbounded threads, 1 to 1 or M to N mapping from user threads to kernel threads and so on. The syntax of application program interfaces (API) of those libraries also differ significantly. There are even subtle differences between different implementations of the same POSIX specification. Developing a multi-threaded application using a particular thread library will lead to poor portability and higher maintenance difficulty in addition to steep learning curve of a set of APIs.

This paper presents a C++ library (*cdevThread*) that hides implementation details on a particular operating system, provides a uniform set of interfaces to an underlying thread package, and facilitates easy development of multi-threaded applications by providing implementations of several commonly used concurrent and parallel programming paradigms such as barrier synchronization [5] and workpile model [6].

2 Design and Implementation

Developing multi-threaded software is difficult since it requires detailed knowledge of many concepts such as (1) creation and termination of threads, (2) synchronization of multiple threads, and (3) data racing and dead lock detection. Moreover, applications are often more difficult to port to different operating systems, and are harder to maintain. Developing multi-threaded applications based upon C++ interfaces that encapsulate existing C interfaces helps improve software quality factors such as correctness, ease of learning and ease of use, portability and extensibility. This section illustrates the design and implementation in brief of the *cdevThread* library.

* Author for correspondence.

2.1 Synchronization

In a multi-threaded application thread operations must occur in a correct order, access to shared objects must be coordinated, all threads must work together to achieve the desired result. Several mechanisms are available to coordinate, or to synchronize, threads within a process like mutex and semaphore. The *cdevThread* library encapsulates these mechanisms in several classes categorized in the following sections.

2.1.1 Mutex

An instance of class *cdevMutex* is an object that allows multiple threads to synchronize access to shared resources. The *cdevMutex* has two states: locked and unlocked. Once a mutex has been locked by a thread, other threads attempting to lock the mutex will block until they can lock the mutex. When the locking thread unlocks the mutex, one of the threads blocked on the mutex will acquire it. Another class is *cdevCond* which is a useful synchronization mechanism when a thread needs to wait for an event to happen. The above two classes are implemented using only inline functions to minimize function call overhead. Several classes like *cdevRecursiveMutex*, *cdevRWMutex* and *cdevSpinLock* are provided in the library for more elaborate synchronizations schemes. Furthermore default values for mutex constructions are provided to prevent programming errors like priority inversion [8].

2.1.2 Block Synchronization

Programs using the above synchronization mechanisms have to lock a mutex at the beginning of a critical block and unlock the mutex at all exit points within this block. To reduce the effort of locking/unlocking of a critical block, a C++ class *cdevGuard* or *cdevSynchronized* is implemented to take advantage of C++ features to acquire the mutex when an object of this class is constructed and to release the mutex when the object is destroyed. The following code segment illustrate this feature:

```
class A
{
private: cdevMutex lock_;
public:
  void foo (void)
  {
    cdevGuard guard (lock_);
    ......................
  }
};
```

Similarly a new base class called *cdevMonitor* is provided to allow a derived class to be locked/unlocked in the same way as the above example. In addition it offers *wait*, *notify* and *notifyAll* methods to inter-thread communication within an object of the same class. Figure 1 shows the relationship among some of the classes.

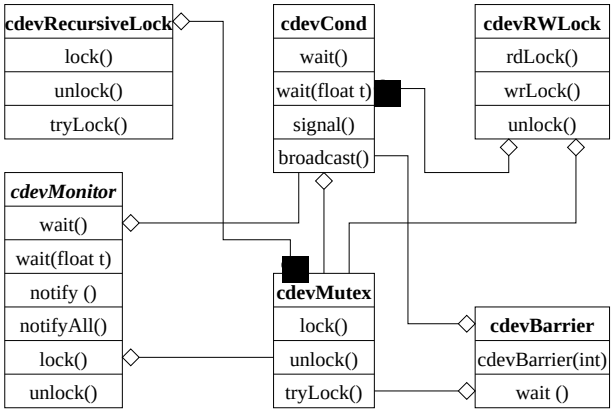

Figure 1: Class hierarchy for some of the synchronization classes using OMT notation.

2.2 Thread Management

A thread undergoes a series of state transitions once it has been created. All threads have to be managed by either a library or by applications. Unfortunately not all libraries provided by operating systems have a simple and easy thread management API . To allow easy creation, clean termination and better information retrieval of a thread, *cdevThreadPtr* coupled with *cdevRunnable* can be used to describe the behavior of a thread. A thread can also be registered into a group (*cdevThreadGroupPtr*) which manages all threads that are registered. This group can wait for all threads to finish, to apply certain actions to one or to all threads and so on. A thread can be stopped or killed by other threads. A cleanup method can be inserted into an object of *cdevRunnable* class to allow clean exit of a thread. Furthermore, a simple technique of counted pointer is used in *cdevThreadPtr* and *cdevThreadGroupPtr* to handle memory management of threads and groups. Figure 2 presents a hierarchical view of the classes in this section.

2.3 Commonly used paradigms

In order to reduce development effort for concurrent and parallel applications, implementations of several commonly used paradigms are provided. Two of these paradigms are worth mentioning here. One is *synchronization barrier* (*cdevBarrier*) which offers the capability to halt several threads at a point until all threads have reached this point. Another is *workpile model* (*cdevWorkPile*) which spawns a predefined number of threads as worker threads to wait for work assigned by a master thread.

3 Applications and Examples

This section presents an application using the *cdevThread* library to increase its server network responsiveness and to reduce porting effort to different operating systems, and illustrates a simple example using the *workpile* paradiagm.

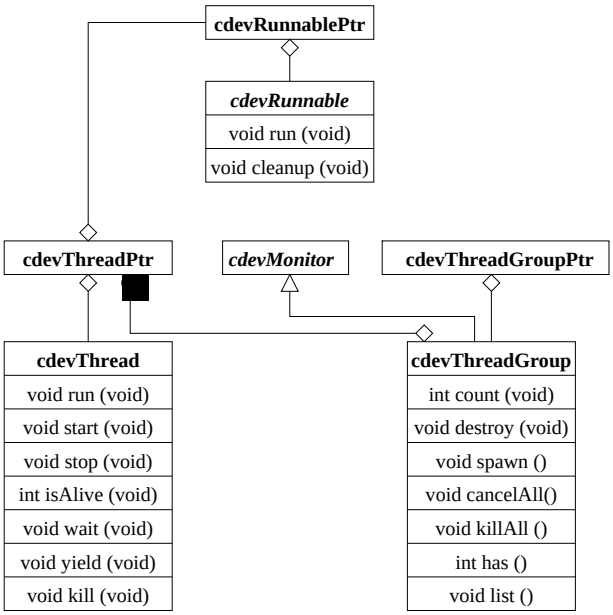

Figure 2: Class hierarchy for thread management using OMT notation.

3.1 Common Message Logging System

The Common Message Logging System [7] (CMLOG) system is an object-oriented and distributed system that not only allows applications and systems to log data (messages) of any type into a centralized database but also lets applications view incoming messages in real-time or retrieve stored data from the database according to selection rules. It consists of a concurrent network server that handles incoming logging or searching messages, a client daemon that buffers and sends logging messages to the server, a Motif browser that can view incoming message in real-time or display stored data in the database, and APIs that can be used by applications to send data to or retrieve data from the database via the server. The requests to the servers of the CMLOG system are either to log messages or to query the database. The service duration for these requests range from very short for logging a single message to very long for retrieving a large number of messages in the database. It is therefore necessary to implement the server using multi-threaded techniques to improve concurrency. In particular, implementation of the *workpile* paradigm provided in the *cdevThread* library is used in the CMLOG server to allow a master thread to add network requests to a queue. Worker threads are signaled to perform the work according to the requests in the queue. In addition the CMLOG system has to be portable among various operating systems such as SunOS, HP-UX and VxWorks. Using the *cdevThread* library makes porting much easier. Figure 3 illustrates the runtime architecture of the CMLOG system and its server.

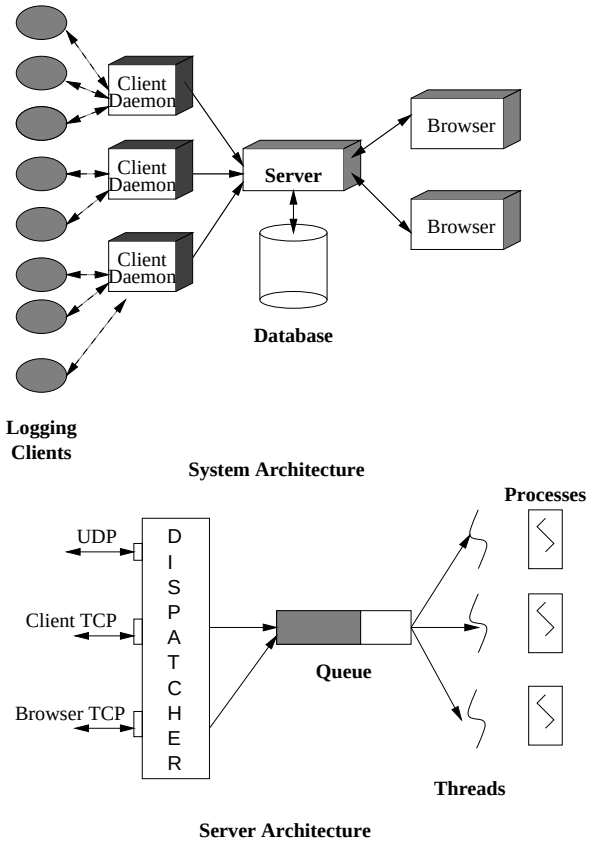

Figure 3: CMLOG system and server architecture.

3.2 Parallel Quicksort

Quicksort is a good example of a divide-and-conquer algorithm that can be implemented using the *workpile* paradigm. The algorithm is easily adapted to this paradigm by making the unit of work the sorting of an array. The following is a code segment of this implementation:

```
class paraQSorter: public cdevWorkPileWorker
{
public:
    void doit (void);
private:
    float* data;
    int n;
};
main ()
{
    paraQSorter *worker[NTHREADS];
    cdevWorkPile work (NTHREADS, worker[0]);
```

```
    work.run ();
    work.wait ();
}
```

4 Conclusions

The *cdevThread* library helps simplify the development of correct, concise, portable, and efficient concurrent and parallel applications. It uses classes to encourage modular APIs, provides frameworks to enable rapid development, offers default values to simplify the interfaces and to reduce programming errors, and inlines functions to eliminate overhead. Currently the library is ported to SunOS, HP-UX and VxWorks. Porting to Windows NT is underway. Finally, source code is available at http://www.jlab.org/cdev

References

[1] Bil Lewis and Daniel J. Berg. "*Threads Primer*". Upper Saddle River NJ, Prentice-Hall. 1996.

[2] Institute of Electrical and Electronics Engineers. "*Portable Operating System Interface (POSIX) - Part 1: Application Program Interface (API) [C Language] - Amendment2: Threads Extensions*". Institute of Electrical and Electronics Engineers. 1-55937-573-6.

[3] M. Powell, S. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. "Sunos Multi-threaded Architecture", *Proceedings of USENIX Winter Conference 1991*, p1.

[4] Huster, H "Inside Windows NT". Microsoft Press

[5] H. S. Stone. "*High-Performance Computer Architecture"*. Reading, Mass., Addison-Wesley. 1987.

[6] S. Ahuja, N. Carriero, and D. Gelernter. "Linda and Friends", *Computer* Vol. 19, p26.

[7] Jie Chen, Walt Akers, Matt Bickley, Danjin Wu and William Watson III, "CMLOG, A Common Message Logging System", *Proceedings of ICALEPCS 1997*, p358.

[8] Phillip A. Laplante, "*Real-time Systems Design and Analysis, An Engineer's Handbook*". IEEE Press, 1993.