Design and Implementation of a Multi-Threaded Xlib

Written By

Carl Schmidtmann, Consultant to Digital Equipment Corporation
Michael Tao, Xerox Corporation
Steven Watt, Consultant to Xerox Corporation

Copyright © 1992 by Digital Equipment Corporation and Xerox Corporation

Abstract

In the MIT X Window System's library Version 11 Release 5 (Xlib) there is minimal support for multi-threaded applications. Programmers writing multi-threaded programs using Xlib are required to provide locking or designate a single thread to handle many of the calls to X functions in their programs.

In this paper we will describe the design and implementation of an upgraded version of Xlib that provides more support for multi-threaded applications. Our goals were to make as few changes to the Application Programming Interface as possible, make the locking invisible to the programmer using the library, and maintain the current portability and performance of the library. This library was implemented on Digital Equipment Corporation's version of OSF/1 using the Pthreads library and Xerox Corporation's Cedar environment.

Trademarks

OSF/1	Open Software Foundation
AIX	International Business Machines Corporation
SUN, Solaris	SUN Microsystems
VMS	Digital Equipment Corporation
Xerox	Xerox Corporation
UNIX	UNIX International
X Window System	X Consortium

Introduction

Today, most of the graphics workstations have a single, high performance CPU. However there are several new systems available now, and more on the way, that contain several CPUs to provide higher levels of performance. To take better advantage of these multi-processor systems, most of the major workstation operating systems are beginning to include support for multi-threading. Multi-threading allows the programmer to divide an application into several parts such that the parts can run concurrently. This allows one process to use several CPUs at one time.

While multi-processor systems are the driving force behind vendor support of multi-threading, even single CPU systems can benefit from multi-threaded programming. Multi-threading provides better handling of asynchronous events and background processing within a single program. Multi-threading can also simplify writing a program that needs a responsive user interface, while at the same time performing long computational tasks or waiting for responses from slow interfaces. In general, a well written multi-threaded program will be more efficient at utilizing the available resources whether running on a single or multiple processor system.

The X Window System has become the de facto standard for high performance workstations. All major and most minor vendors of these workstations supply an X server and X libraries with their systems. Most of these implementations of X are based directly on the code supplied by the X Consortium with only minor changes incorporated for the target platform.

Currently the support for multiple threaded applications is very limited in the X Window System as it is supplied by the X Consortium. With the trend toward multi-processor systems and multi-threaded operating systems, full support for multi-threading is critical if X is to remain the standard for high performance graphics workstations.

Our goal of creating a multi-threaded version of Xlib started as part of another project to port an application from a multi-threaded, proprietary windowing system onto X. Our first pass was a "quick and dirty" patch of the X11 Release 4 (X11R4) Xlib code. After that was done, we moved on to Release 5 (X11R5) Xlib code with the intention of making our work available through the X Consortium as part of the standard releases. This paper describes our work so far and points out some of the work that should still be done.

Terminology

Process Context - A standard system process including the data address space, the program text and at least one execution thread. The process context also contains a single set of other operating system resources such as file descriptors, socket connections, etc.

Thread - A single sequential flow of control that executes within a process context. Each thread has its own local stack frame, but shares its data area and heap with all other

threads within the same process context. Threads are scheduled in a fashion similar to standard process scheduling. Once a thread is scheduled to run, it starts executing and continues until it blocks (waiting for a resource to become available), or it uses up its time slice. Then, another thread is scheduled to execute. Threads are also referred to as lightweight processes.

Single-Threaded - A program with only a single execution thread or a library not designed to be used in a multi-threaded environment.

Multi-Threaded - A program that uses multiple execution threads, or a library that allows concurrent access from multiple threads.

Thread Safe - A library that can be used in a multi-threaded program without the use of special precautions or locks by the application programmer. This does not mean that the library is multi-threaded. An example of a thread safe, but not multi-threaded, library is one that uses a locking mechanism to serialize *all* access to the library. In that way, no more than one thread can run within the library at a time.

Multi-Processor - A system that contains multiple CPUs. This paper only considers multi-processing systems that have shared memory.

Inter-Thread Communication - Any one of several methods by which two threads can exchange information. Usually, this involves writing a message to an area accessible to both threads, and then notifying the receiving thread that the message is present. Whatever the method, both threads must know in advance what the semantics and mechanics of the exchange will be.

Mutex (MUTual EXclusion) - a synchronization object that controls access to a shared resource. The use of a mutex has the effect of serializing access by multiple threads to the protected resource.

Crowd Monitor - a synchronization object that permits multiple read and exclusive write access to a resource. It is usually used for a resource that is read frequently by several threads and rarely updated.

Condition Variable - a synchronization object that allows one thread to suspend its execution until another thread notifies the first thread that it can resume execution. Condition variables are usually used for producer/consumer synchronization where the consumer waits on the condition variable when it needs a resource that is not currently available. The producer then "notifies" the condition variable to awaken the consumer when the resource is available.

Deadlock - a condition in which two or more threads are waiting for a condition that will never occur. For example, each of two threads is holding a resource that the other one is waiting on. Thread 1 gets a lock for resource A and then is preempted by thread 2. Thread 2 gets a lock for resource B and then attempts to get the lock for resource A. Thread 2 is then blocked because thread 1 has the lock. Thread 1 then resumes, and

attempts to get the lock for resource B. Now, both thread 1 and thread 2 are blocked. This deadlock cannot be resolved unless some third party (another thread) kills either thread 1 or 2.

Background

The next two sections, **Multi-threaded Issues** and **Using a Single Threaded Library**, are intended for readers that are not familiar with multi-threaded programming. They describe multi-threading concepts, and the issues of programming in a multi-threaded environment. Readers familiar with multi-threaded programming may skip to the section titled **Multi-threading in R5 Xlib**.

Multi-threading Issues

Multi-threading is a paradigm that allows a single process to act as if it were several processes that all share a common environment. Multiple threads running in a single process may be scheduled independently and can be preempted by each other.[1] The reason for multiple threads in a single process context, instead of multiple processes using shared memory, is that the overhead of thread switching is significantly lower than that of process switching. In addition to sharing data, threads also share file handles, socket connections, and other per-process resources.

Because all the threads in a process context share resources, there are some particular issues that must be considered when designing a multi-threaded program or library. The most fundamental issue is that whenever a shared resource is being accessed, the design must ensure that there is a mechanism for controlling this access. For read access, the validity of the resource must be ensured (i.e., does it exist?, not being modified?, etc.). For write access, exclusive access needs to be guaranteed. While the design must provide a mechanism for controlling access to shared resources, it is *ultimately the responsibility of the programmer to use it*.

In a multi-processor system, threads run at *exactly* the same time. In a uniprocessor system, they are time-sliced to *appear* that way. In either case it is possible, when writable data are shared by both threads, that one thread can interfere with another. Therefore, access to shared resources must be tightly controlled.

The simplest example of the problem with concurrent access is a read-modify-write operation. Two threads running concurrently can read a shared variable, change the value, and then write it to the variable. The write operations, while appearing simultaneous, will actually happen serially, and whichever value was written last will

---

[1]Some multi-threaded systems do not do preemptive scheduling among threads. However, software that is designed for preemption will operate properly on a non-preemptive system.

overwrite the first value. This is true for a simple increment (e.g., i++) as well as for reading a record, performing several calculations on it, and then writing it.

To control access to shared resources, a multi-threaded system provides mutexes and condition variables.[2] Proper use of these mechanisms requires some design considerations and some engineering discipline. Without proper design, a multi-threaded program is subject to random behavior, poor performance and possible incorrect results. These are the same problems faced when creating a single-threaded program but they manifest themselves in different ways. A tutorial on proper multi-threaded design is beyond the scope of this paper, but for those not familiar with multi-threaded concepts, the following is a brief discussion of the mechanisms and design principles we employed to make Xlib multi-threaded. For those interested in more a more detailed introduction to programming with threads, refer to [Bir89] and [Ora91].

A mutex is used to serialize access to a shared resource. When a thread needs access to a shared resource, it locks the mutex, does what it needs to do and then unlocks the mutex. If thread A tries to lock a mutex and thread B has already locked it, thread A is blocked until the mutex is unlocked by thread B. After thread B unlocks the mutex thread A will be granted the lock and may continue. If several threads are blocked trying to lock a mutex, only one thread will be awakened at a time.

A condition variable is used to synchronize two or more threads. If thread A needs some data from thread B it will lock the mutex associated with the data and check to see if the data is available. If the data is not yet available it will wait on the condition variable for that data. The mutex is unlocked while thread A is waiting on the condition variable. When thread B has the data, it locks the mutex and writes the data to the shared area. It then notifies the condition variable and unlocks the mutex. When the condition variable is notified, the mutex is relocked and thread A continues. It can then retrieve the data and unlock the mutex. A condition variable is always associated with a mutex.

Using mutexes and condition variables, it is possible to construct a crowd monitor[Smi92]. The crowd monitor allows multiple threads to obtain a read lock and perform reads concurrently. When a thread requests write access, it will be blocked until all of the threads doing reads have given up their read locks. The writing thread then obtains a write lock, which blocks all new readers or writers. When the write thread is finished, it gives up the write lock and any waiting threads are allowed to continue.

When designing multi-threaded code, all shared resources must be identified and some control mechanism defined for access to each shared resource. The design must then ensure that all accesses to shared resources use the mechanism correctly. A well-structured design will make this job easier. By localizing access to a shared resource to a few routines, it is easier to verify the behavior of the locking mechanism to avoid

---

[2]For this paper we are using the mechanisms defined by the IEEE POSIX 1003.4a draft proposal. Other multi-threaded systems will provide similar functionality but may use different terms.

deadlocks and race conditions. Also, structuring the code and the mutexes in a hierarchical manner will help to ensure that mutexes are locked in the same order by all routines. Of course, all the work done in design will not guarantee that the code will work correctly; the programmer needs discipline to ensure that the proper mechanisms are always used *correctly*.

Using a Single Threaded Library

Designated Thread

There are several possible methods of making a library thread safe. One method to make a library thread safe is to use the library in a single-threaded manner. To use a library that is not thread safe in a multi-threaded program, the application programmer designates one thread in the program as the interface to the library. All other threads needing access to the library must use inter-thread communication with the designated thread, which will make the call and return any results to the originating thread by way of inter-thread communication. This method is useful if the source code for the library is not available, but it puts the burden on the application programmer. This extra effort can easily amount to more than the effort saved by using the library.

Encapsulation

Another way to make a library thread safe is to put a wrapper around the library so all calls into the library go through the wrapper. The wrapper functions lock a mutex and make the library call. This method makes the library thread safe without making any internal modifications to the library. The drawback to this approach is that the library is still single threaded, and the multi-threaded program that uses it is constrained to have only a single thread executing in the library at any one time. This may cause a bottleneck if the library is called frequently from multiple threads.

Multi-threading

The next method of making a library thread safe is to redesign and recode the library so it supports concurrent access. This will usually require a significant amount of redesign work, since most of the library functions have probably been designed and optimized to take advantage of the single threaded environment. It also requires the person modifying the library to perform a lot of reverse engineering to determine what needs to be locked, where deadlocks might occur, and which changes might impact the functionality. During redesign, the library interface may also need alterations. The changes may be required to maintain functionality, add functionality, or limit functionality.

The advantage of using a designated thread or encapsulating the library is that it does not require any modifications to the library. These are the only alternatives if the source code for the library is not available. The disadvantage of these two methods is that the library becomes a single shared resource, and could become a bottleneck if the library is used heavily by more than a single thread. The designated thread method also places a burden on the application programmer to deal with multi-threaded issues.

We chose to modify Xlib to make it multi-threaded because the library would be heavily used by multiple threads. Also, since Xlib is usually a core library that comes with the operating system, users would expect it to be thread safe and not require any special locking or access restrictions.

The best method for making a library thread safe is to design it to be multi-threaded from the start. This obviously isn't feasible for existing libraries, but it should be considered for new work if there is any chance the new library will be used in a multi-threaded environment. It is our belief that this applies to *any* library now being written.

Multi-threading in R5 Xlib

Xlib has a few multi-threaded features; the display lock, the static lock in XOpenDisplay, and the event queue used in conjunction with the Display lock.

All server requests and all access to the event queue are locked with the Display lock. This protects the request buffer, the event queue, and the server connection. Presumably, this lock was meant to protect all access to the Display structure, but the current implementation allows many accesses without regard to the lock. A new patch to the header file is now hiding the internals of the Display structure from application programs, but several of the toolkit libraries violate this intent and directly access the Display structure and its resources.

The static lock in XOpenDisplay protects the XOpenDisplay function so that no more than one thread can be executing the function at a time. This keeps the static list of Display structures from being accessed concurrently by multiple threads.

The event queue allows events to be read from the server connection while waiting for a reply to a server request. It also allows the connection to be read more efficiently, by reading all events present before processing them. Access to this queue is protected by the lock in the Display structure.

Limitations in R5 Xlib

In our environment, the multi-threaded features in the R5 Xlib fell short of providing a truly thread safe library. The areas in which we found problems were the handling of the display connection, unprotected static data areas and the error handler mechanism. The concept of a single lock for all Display-related activity does provide thread safe access, but it does not allow for any parallelism. It also requires that the application programmer has knowledge of the internals of Xlib and makes allowances for it.

Server Connection

In the current implementation, if a thread is waiting for events from the server, no requests can be sent to the server until at least one event is received and queued. Then, the thread leaves the library, allowing the requests to be sent. A multi-threaded application will typically have one thread that does event dispatching and other threads

that handle background tasks. If one of the background threads needs to send a request, it will almost always be blocked until an event arrives. If the user is waiting for a background process to complete and notify him by updating the display, there is a deadlock with the user.

On the other hand, if a thread sends the server a request that requires a reply, all access to the request buffer or the event queue is blocked until the reply is received and the thread returns.

Static Data

Another problem with Xlib in a multi-threaded environment is that several sections of the code depend on static, and sometimes global, data. Currently the only static areas that are protected by locks are the list of Displays and the event structure free list. Some examples of unprotected static data are the Quark table, the Xrm databases, the Context table, the Error Text table, the Xcms color map tables and the error handlers.

Error Handlers

The error handlers in Xlib are global function pointers that can be changed at any time by an application program or an internal function. This means that when a thread registers an error handler, there is no guarantee that the handler will still be installed when the error it was supposed to handle arrives. There are also internal routines that install their own error handlers, make server requests, and then restore the original error handler. This can cause unpredictable behavior. For example, one thread saves the current error handler, installs a new error handler, and then waits for a response from the server. While the first thread is waiting, a second thread installs a new error handler. The consequences of this case are that the first thread's error handler is not active when it is needed, and when the first thread restores its saved error handler, the second thread's handler is lost.

Goals of This Implementation

Our work to create a thread safe Xlib was part of a project to port a large multi-threaded application to X, from a proprietary windowing system. Because of the project deadline it was vital that the first pass at the library be completed quickly. Therefore, all the first pass accomplished was to make event handling and server requests thread safe. This work was done on X11R4.

During this stage of the project, X11R5 was released, so it was decided that working on the latest version of the code would be more productive than having to do everything twice. It was also decided at this time that our work would be contributed to the X Consortium. The goals of this second effort included all of Xlib and were better defined than the first pass. The goals were:

1. Robust Multi-Threading

The goal here was to make the library truly multi-threaded and not just thread safe. The library needed to allow for parallelism where possible (e.g., read events and send requests simultaneously) without the application needing to know what calls were safe to execute in parallel. It also meant that thread support would be built into the code without kludges.

2. Same Code For Multiple and Single Threads

All the changes that were made had to support both multi-threaded and single threaded applications. This goes beyond putting #ifdef's around everything. The library should be compiled with multi-threaded support if it is available on the platform for use in multi-threaded applications. A single threaded program should be able to use the same library file (shared or statically linked) without any knowledge or special code for multi-threading.

3. No Interface or Functionality Changes

While this was a goal, we knew from the start that there might be some things that couldn't be made thread safe. We also knew there would probably be some extra functionality that would be useful in a multi-threaded environment. The strongest incentive to keep to this goal was that the only user documentation we had to write was to describe any differences in the interface.

4. Minimal Structural Changes

While the internals of a library can be changed without affecting application programs, we tried not to modify the internal workings unnecessarily. The reasons for this were a need for our work to be accepted by the Consortium and because the less we changed the less we might break.

Library Modifications

The modifications made to Xlib for the project fell into three categories: general modifications to ensure appropriate resource locking, enhancements to existing functionality, and introduction of new functionality.

General Modifications

The problems involved with the Xcms, Quark and Xrm routines were the same: The need to lock static data against multiple initialization and multiple writers. The code was analyzed to determine the best places a single lock could be added simply. In most cases,

these functions are of a fast-in-fast-out nature, so there is not much probability of long waits on the lock. The exceptions to this generalization are the various routines that do operations on entire databases.

A lock has been added to the Xrm database structure to serialize manipulations of the database. However, there are still some potential sequencing problems; if one thread stores one value for a resource, and another thread stores another value at (almost) the same time, it is simply a race to see which thread's data actually gets stored, and which one's data falls on the floor. Also, with the current scheme, only one thread may be searching the database at a time.

Fortunately, writes to these databases are generally done only at initialization time, and most accesses are read-only after that. There are specific enhancements that could be applied to both Xcms and Xrm, but resources were not available to investigate them fully. They are described in the **Future Work** section.

Enhanced and New Functionality

We have added some new features and modified some of the existing functionality. These changes required some additions to the Xlib interface but we have tried in all cases to leave the original interface and functionality intact. We have modified the internal access to the server connection so that a thread waiting to read from the connection does not prevent a thread from writing to the connection. We have also allowed for round trip requests to be handled while there is a thread waiting for events. For multi-threaded programs that divide the handling of events among several different threads, we have implemented support for multiple event queues for a single server connection. Along with the multiple event queues, we have added a timeout option when requesting events so a thread doesn't block forever waiting for an event. An area in which we both enhanced and added functionality was the error handling mechanisms.

Server Communications

A mechanism was added to control reading the server connection without blocking access to any other part of Xlib. This allows an event handling thread to spend most its time waiting for events without preventing other threads from sending requests to the server.

To implement this, it was necessary to modify the round trip request handling by creating a queue for replies. Now, access to both events and replies is done by way of queues. A thread sending a round trip request will check the reply queue, and a thread waiting for events will check the event queue. Each queue has a mutex that controls access to the queue. After locking the mutex, the thread checks to see if what it is looking for is already available. If a reply or an event is available, it is taken off the queue, the mutex is unlocked, and the reply/event is returned. If the queue doesn't contain what the thread is looking for, the thread will attempt to read from the server connection. Refer to Figures 1 and 2.

All read access to the server connection has been consolidated into a single routine, access to which is controlled by a semaphore-like mechanism. Whenever a thread needs to read from the server connection, it locks the read mutex and checks the read flag. If the flag is set, the thread locks the appropriate queue mutex, unlocks the read mutex, and waits on the queue condition variable. If the read flag is not set, the thread sets it, and unlocks the read mutex. The thread then reads from the connection. Note that the read flag controls access to the connection, and the read mutex controls access to the read flag.

When an event or reply is received, the appropriate queue's mutex is locked and the event/reply is placed on the queue. The queue's condition variable is then notified to wake up any thread waiting for that queue. The queue mutex is then unlocked. If the thread doing the reads is not interested in what was received, it continues to read from the connection.

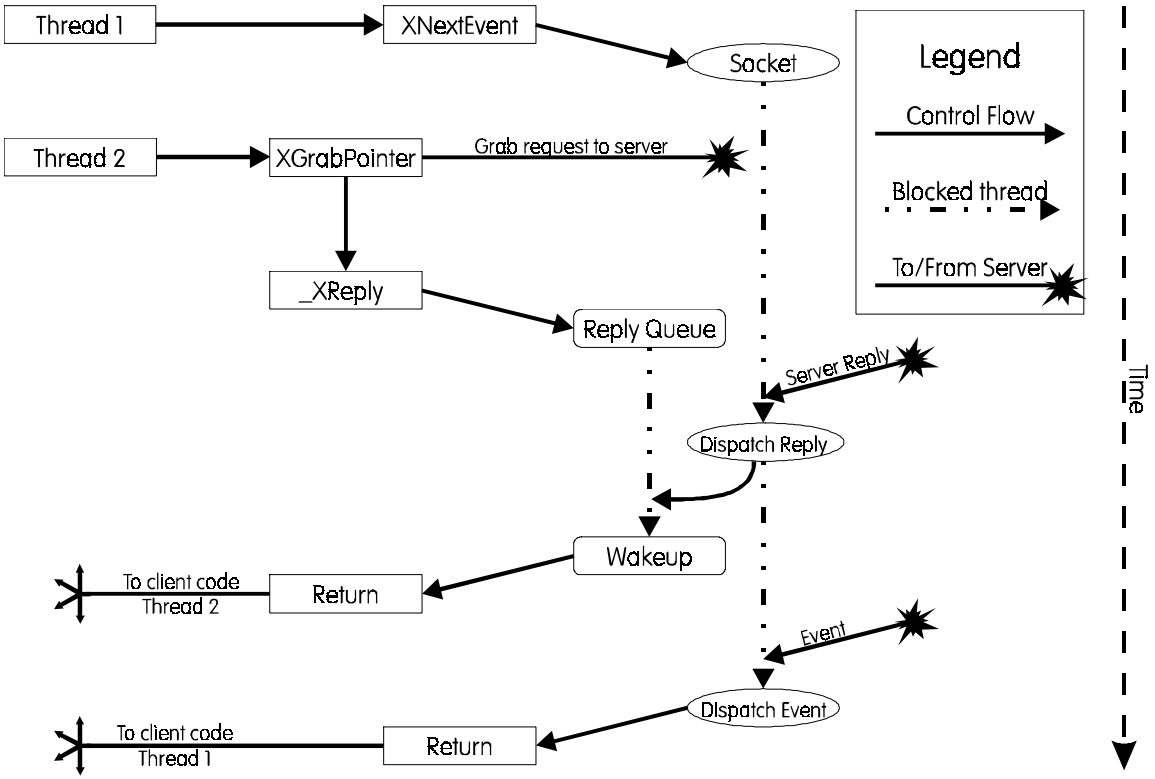

Figure 1

If the thread was interested in the reply/event received, it will stop reading from the connection. The reading thread then locks the read mutex, clears the read flag, and wakes up a thread waiting on a queue condition variable, so the waiting thread can start reading. The thread that was reading from the server connection can then get the event/reply from the queue and return it. A thread waiting on a queue condition variable must check the queue when it wakes up to see if what it was waiting for has arrived. If it hasn't arrived, the thread tries again to gain read access to the connection.

A consequence of having a reply queue is that it is possible to have more than one outstanding round trip request. The server is still going to process the requests serially, so this is not a large performance enhancement, but it allows an overlap between transmission time and request processing.

Hooks have been added to the code to allow implementation of separate read and write threads. This would allow a thread to be dedicated to reading from the server connection and putting whatever arrives onto a queue. Another thread could be designated to perform all writing to the connection. These extra threads would save some checking and locking overhead but cause a little more thread switching. The mechanism has not been implemented yet because it requires changes and/or additions to the Xlib interface to allow the application to indicate the desire to start these extra threads.

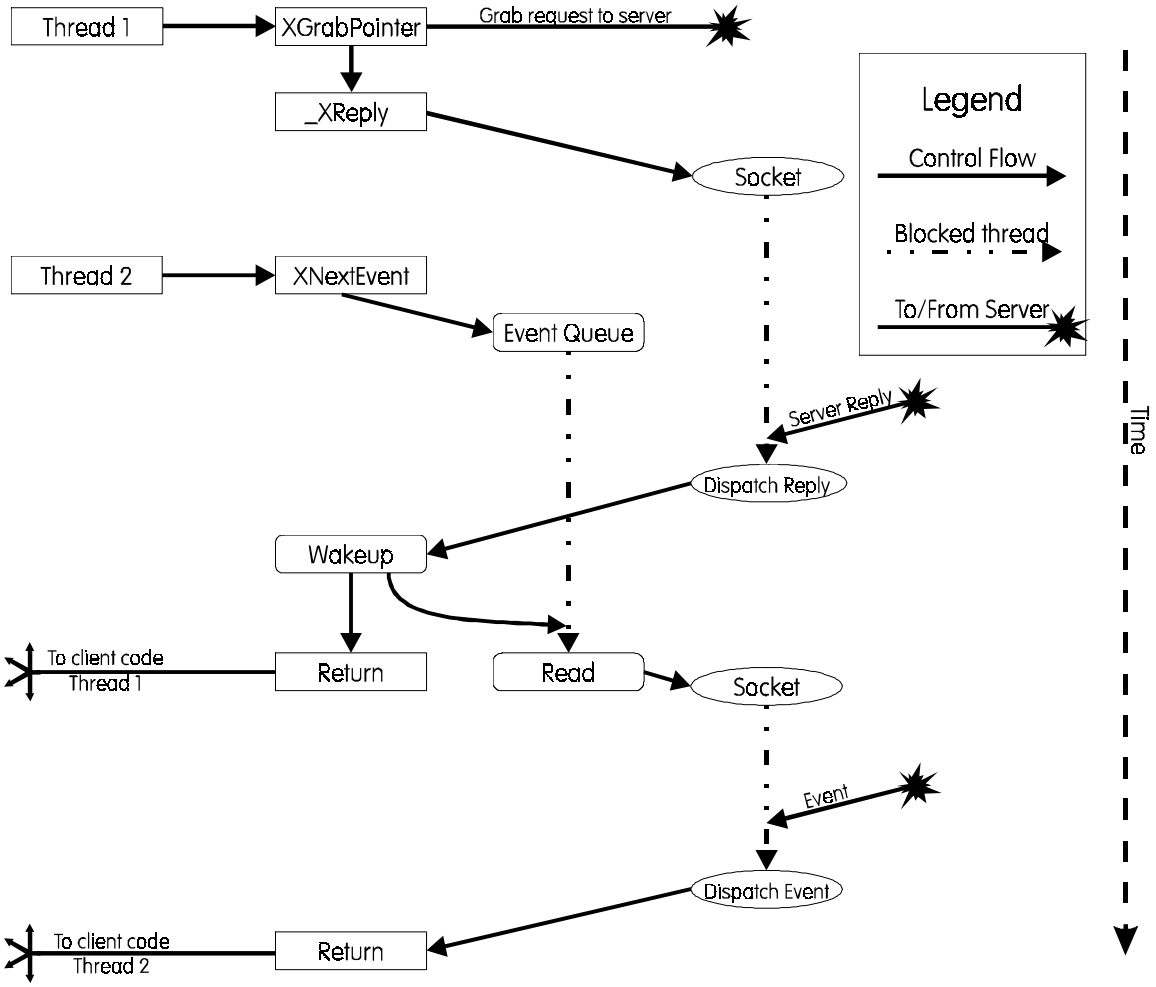

Figure 2

Multiple Event Queues

The new multiple event queuing design is closely fitted into the original Xlib architecture. All existing Xlib event retrieving functions operate on the default event queue. A new set of multiple event queuing functions operate on any created event

queues as well as the default event queues. Multi-threaded clients can create multiple event queues and select any events to be dispatched onto these event queues. The same event can be placed on multiple queues. A mask and a predicate function are used together to filter the events placed on a queue. Each event received from the server will be checked against the mask for the queue. If there is a match and the predicate function is non-NULL, it will be called. If the predicate function returns true or is NULL, the event is added to the queue. If the mask doesn't match or the predicate returns FALSE, the event will not be put on the queue. Each event is checked against all queues so that a single event can be placed on more than one queue. For performance reasons, the number of queues should be kept to a minimum. Also, masks should be used to limit the number of times a predicate function is called, and predicate functions should be as fast and short as possible.

When the client opens a display connection, a default event queue is created with default values of all 1's for the mask, and null for the predicate. The original Xlib event functions will operate on this event queue.

Each event queue is designed to be accessed by a single thread only. If a client chooses to access an event queue in multiple threads, the client must guarantee the mutual exclusion of accesses to the event queue.[3]

Timeout Support for Event Retrieving Functions

We implemented a new version of each of the existing event-retrieving functions that block. These functions have a timeout parameter, and return TRUE if an expected event is received within the timeout interval. Otherwise, they will wait until the timeout interval has expired, and then return FALSE.

Error Handling

The global error handlers, _XErrorFunction and _XIOErrorFunction, were replaced by three per Display handler lists. There is a non-fatal handler list for both internal and external (to Xlib) error handler functions. There is also a list for fatal error handlers. When a non-fatal error occurs, usually an unexpected error message from the server, the thread reading from the server connection calls each handler on the internal non-fatal error handler list. If one of them handles the error, it returns TRUE and no more handlers are called. If none of the internal handlers returns TRUE, then each of the handlers on the external list is called. If one of these handles the error, it returns TRUE and no more handlers are called. If no handler returns TRUE, the default non-fatal error handler is called.

---

[3]This is arguably a bug, and could be fixed with another layer of locking. However, accessing one queue from multiple threads will probably produce unpredictable results.

Fatal errors are handled in a similar fashion. However, the return value of the handlers is ignored and all the fatal error handlers are called. After the handlers are called, the default fatal error handler is called.

New functions were added to the Xlib interface to allow application programs to add and remove handlers from the handler lists. Error handlers are not guaranteed to be called within the thread that registered them. They are also not guaranteed to be called in any particular order.

This new way of dealing with error handlers allows applications to have an error handler watching for its errors, and to have cleanup routines registered if there is a fatal error. By making the error handlers a per display resource it is now possible to have a display connection crash without the entire application crashing with it.

To support existing applications, the functions XSetErrorHandler and XSetIOErrorHandler each register a handler as the default non-fatal and fatal error handlers respectively.

Future Work

Many tasks need to be completed to make X truly multi-threaded, most of which are design and structural changes. Our approach was to do as much of the job as was needed by our clients, and whatever else we had time for. This forced some tasks to be put on hold.

Xlib Enhancements

Extensions

A major area that still requires work is the support code for the extension library. This code provides support for most of the X extensions (e.g., MIT-SHM, PEX, SHAPE). This code cannot be made thread safe without doing major redesign work. Most of the extension skeleton code does not take multi-threaded issues into account, and none of the standard extensions that we examined addressed multi-threaded issues. The redesign of Xlib internals also forces a change in the way that extensions are written. Each standard extension should be examined and redesigned with multi-threaded issues in mind. These changes must reflect the different way that reply and error handling work, and locking must be carefully considered. We did implement the MIT-SHM (shared memory) extension by writing a new interface that handles locking and calls the MIT routines, but this implementation is far from optimal.

Possible Restructure of Xrm Functions

Xrm has one major obstruction to multi-threading: The XrmGetDatabase and XrmSetDatabase routines. These routines manipulate the per-display database pointer. Once a thread has that pointer, it can do whatever it likes to the database, and then reset

the per-display pointer. If two threads do this, there is no predicting what will happen. For the current implementation, it was decided that this is a client problem; don't use XrmGetDatabase or XrmSetDatabase once multiple threads are running. Unfortunately, that violates one of our primary goals, which was to hide all locking from the application programmer.

Xrm also needs more work with locking, primarily to take advantage of crowd monitors. This would allow multiple threads to read one database at the same time, while enforcing correct behavior during writing. The current implementation allows only a single thread to be reading from a given database at one time.

Possible Restructure of Xcms Functions

The Xcms system is very heavily self-referential code, and was very difficult to make thread-safe. It makes calls to other modules of the library, which can call back into the Xcms module. This breaks the lock hierarchy necessary to prevent deadlocks. This code could also benefit from the use of crowd monitors to allow multiple readers.

Redesign Xcms Colormap Search

The function XcmsCCCOfColormap and the function it calls, CmapRecForColormap, need to be looked at very carefully. XcmsCCCOfColormap currently runs through a loop for every visual of every screen on the display. It creates a window with the passed-in Colormap on each visual, and waits for the BadMatch to come back from the server. This process is fine in a single-threaded system, but since it calls XCreateWindow, it will not work in a multi-threaded system.[4] The quick fix would be to move the XCreateWindow request formatting into CmapRecForColormap, but it would be preferable to come up with a better implementation. Two possibilities are: adding a new protocol request, or using a general resource identification extension.

Reading & Writing

A performance improvement could be gained by double-buffering server requests. This way, requests can be prepared in one buffer while the other buffer is being sent to the server. The value of this approach is not known; it will probably have a small impact in high-bandwidth environments. It will probably have a much greater impact in low-bandwidth environments, where the time spent writing out a buffer could be significant. However, running a program that produces enough output to fill both buffers would saturate the transport line.

Double buffering, when used in conjunction with a separate writer thread, encourages more "fairness" in the distribution of scheduling time. No particular application thread

---

[4]This method for finding a visual for a Colormap causes a deadlock because it calls XCreateWindow with the Display lock held. XCreateWindow also locks the Display lock, resulting in a deadlock on some systems. This method must be used synchronously, causing potentially long delays.

pays the price for a buffer write, since it is handled by the writer thread. This fairness improvement also applies for a separate reader thread, because no application thread spends time reading replies and events that it will not process.

There is some support for separate reader and writer threads in this version of the library. However, we have not provided a method to allow either of these threads to be created. Implementing a creation mechanism for these threads requires modification to the Xlib interface. The primary advantage of this approach is better turn-around on data from the server. It also makes the implementation of double-buffered output much simpler.

Internationalization

The Xsi internationalization support in X11R5 gives some indication that it was designed with multi-threaded issues in mind. The Ximp support does not. The resources to fully investigate the internationalization support in X11R5 were not available, and our client had an existing, proprietary method for handling multi-national text.

Related Work

Connection Status Call

A call should be added to Xlib to allow toolkits to avoid networking-system dependencies. This call would allow a toolkit or client program to ask if input is available on a list of display connections, possibly giving a timeout argument. This call is not strictly required in a multi-threaded system, but could be a useful alternative to using XConnectionNumber on several display connections and having to call the poll or select routines directly. This would improve portability of multi-display applications, since they would no longer have to know the correct system interface for multiplexed input. This call would be better than the retrieve event with timeout functions that we added, if non-blocking event reading methods were used.

Sun is working on a thread safe version of Xt and XView [Sma92]. However, these versions treat the entire Xt and XView subsystems as locked resources, such that only one thread can be running in any part of the libraries at a time. This greatly reduces the concurrency possibilities in a heavily graphics-oriented application, or in one that has a large number of threads that use the toolkit.

The need for higher-level libraries to be multi-threaded is becoming critical. Currently, OSF/1 has threads, as do Solaris 2.0, VMS, AIX 3.2, System 5 Release 4, and SunOS 4 with the "lwp" library. Each of the higher-level toolkits and widget sets for these platforms should be written to allow multiple threads to do useful work at the same time. A toolkit and widget set that are multi-threaded should be made available as part of the core X distribution.

Multi-threaded X Applications

Applications will need to be written to take advantage of the multi-threaded model of computing; multi-threaded libraries are not useful if there are no multi-threaded clients to use them. It is unlikely that this will be a problem: there is already one major software package using this code, and more are in development.

In a multi-threaded environment, it will be possible for a program to have one thread doing data acquisition in the background, displaying its data as it receives it. Another thread could be doing computations on the received data, and displaying the results of the computations. A third thread would handle user requests and events. This resulting application would have better responsiveness and cleaner code structure than one using the callbacks, timeouts, and work procedures employed with current toolkits.

Conclusions

1. The Consortium must create a policy for multi-threading.

In the authors' opinions, it is imperative that the X Consortium require new additions to the core X distribution to be thread-safe. If this commitment is not made, then the state of the library will oscillate from fully multi-threaded to its current inconsistent state, where some things work, some things might work, and some things cannot work.

2. Multi-threaded Xlib is possible.

We believe we have demonstrated that the current Xlib code from the X Consortium can be made multi-threaded without breaking the interface. The library we have created works with current single-threaded applications without modification. There will undoubtedly be some applications that break because they were cheating and calling internal Xlib routines that were not part of the external interface. Toolkits are the biggest violators of the Xlib interface but as we pointed out, they will need work to make them multi-threaded also.

3. Multi-threaded Xlib is needed.

The need for a multi-threaded Xlib (as well as Xt, Motif, OLIT, etc.) is clear from the fact that most major workstation suppliers are now shipping a multi-threaded operating system with their workstations. If they are not shipping thread safe libraries with those systems, the application developers are going to be in for some unpleasant surprises.

4. There is still much work to be done.

The work to be done to supply fully multi-threaded libraries for X is significant. Xlib still needs some work, most of the toolkits have not even been examined with multi-threading in mind, and from what we can tell nobody has done anything with the widget sets. Since some of the widget sets are built and maintained by suppliers of multi-threaded operating systems, we assume they are working on it. Finishing this work will

require some cooperation from the groups that have built the different portions of the libraries, the major workstation vendors, and the X Consortium.

5. New X features must be designed with multi-threading in mind.

Lastly, we strongly urge anyone who is working on updates or additions to the libraries make their code multi-threaded. We feel it would be a mistake for the Consortium to take on the task of supporting two versions (single threaded and multi-threaded) of each of the libraries. This would make maintenance a nightmare. However, it would be as big a mistake to leave the code as it is and let the suppliers of the multi-threaded systems each make their own updates to create a multi-threaded library: that invalidates any attempt at standardization.

Acknowledgments, and Thanks to:

Xerox Corporation, and Digital Equipment Corporation, co-sponsors of this effort to produce a multi-threaded Xlib.

Ann Ting (Xerox), Brian Hoshiko (Xerox), and Alex Phillips (Digital),without whose support the multi-threaded Xlib project would not have been possible.

Rita Laughter, for editing and driving our writing effort (we needed it).

Catherine Watt, for final grammatical and structural editing magic (we **really** needed it).

References

[Atk89] Atkinson, Demers, Hauser, Jacobi, Kessler, Weiser. "Experiences Creating a Portable Cedar" SIGPLAN '89 Conference on Programming Language Design and Implementation. June 19-23, 1989

[Bir89] Andrew D. Birrell, "An Introduction to Programming with Threads." Research Report #35, January 1989. Digital Equipment Corp. Systems Research Center

[Ora91] O'Reilly & Associates, Inc. "Guide to OSF/1: A Technical Synopsis" ch. 2, 11

[Sma92] Bart Smaalders, Brian Warkentine and Kevin Clarke, "Prototyping MT-Safe Xt and XView Libraries." The X Resource, Issue 1, pp. 91-107 Winter 1992

[Smi92] John Allen Smith, "The Multi-Threaded X Server." The X Resource, Issue 1, pp. 73-89, Winter 1992

[Wei89] Weiser, Demers, Hauser. "The Portable Common Runtime Approach to Interoperability" ACM Symposium on Operating Systems Principles. December 1989