Presented at the
9th Annual X Technical Conference,
Boston, Mass., January 31, 1995

D11: a high-performance, protocol-optional, transport-optional window system with X11 compatibility and semantics

Mark J. Kilgard *
Silicon Graphics, Inc.

February 5, 1995

Abstract

Consider the dual pressures toward a more tightly integrated workstation window system: 1) the need to efficiently handle high bandwidth services such as video, audio, and three-dimensional graphics; and 2) the desire to achieve the under-realized potential for local window system performance in X11.

This paper proposes a new window system architecture called D11 that seeks higher performance while preserving compatibility with the industry-standard X11 window system. D11 reinvents the X11 client/server architecture using a new operating system facility similar in concept to the Unix kernel's traditional implementation but designed for user-level execution. This new architecture allows local D11 programs to execute within the D11 window system kernel without compromising the window system's integrity. This scheme minimizes context switching, eliminates protocol packing and unpacking, and greatly reduces data copying. D11 programs fall back to the X11 protocol when running remote or connecting to an X11 server. A special D11 program acts as an X11 protocol translator to allow X11 programs to utilize a D11 window system.

1 Introduction

High-bandwidth services like audio, video, image processing, and three-dimensional graphics are at or coming soon to a desktop computer near you. Processors and graphics hardware continue to get faster. The increasing relative cost of cache misses, context-switching, and transport overhead give an advantage to greater system locality. Users desire slicker application appearances, better responsiveness, and seamless inter-application communication.

What should window systems do to stay current with these trends?

This paper proposes a restructuring of the industry standard X Window System [21] to answer these trends. This new window system architecture, called D11,* provides higher performance local window system

*Mark J. Kilgard is a Member of the Technical Staff at Silicon Graphics, Inc. Mark believes software should get faster for more reasons than just that hardware got faster. Address electronic mail to mjk@sgi.com

*The D11 window system described in this paper is a proposal; an implementation of D11 does not exist nor do plans exist to implement D11 at this time.

operations while preserving complete compatibility with X11 clients and servers. The system utilizes a new operating system facility similar in concept to the Unix kernel's traditional implementation but designed for user-level execution. Local D11 clients can execute within the D11 window system kernel without compromising the window system's integrity. The transport of protocol is eliminated for local window system interactions. Context switching and data copying costs are minimized.

The name D11 refers not to a new revision of the X11 protocol; indeed D11 aims to obsolete the local use of the X11 protocol. D11 is a new window system architecture but fully X11 compatible. X11 programs can be trivially recompiled to utilize D11 since the full Xlib application programming interface (API) is retained by D11. Existing X11 clients will continue to work with the D11 window system by generating X11 protocol, and D11 clients will work with existing X11 servers by falling back to X11 protocol. D11 is not an extension or minor addition to X11. It is a reinvention of X11's fundamental architecture.

Section 2 motivates the new architecture for D11 by considering window system trends. Section 3 presents the means for better performance in D11. Section 4 contrasts D11 to other approaches. Section 5 describes the *active context* operating system facility that D11 utilizes. Section 6 considers the implementation of D11 and how to provide complete X11 compatibility in D11.

2 Window System Trends

The two dominant, standard window systems today are X11 and Microsoft Windows. X11 is the *de facto* window system for Unix workstations; Microsoft Windows is the Microsoft-imposed standard for PCs.

2.1 Network Extensibility Considered

X11 uses a network extensible protocol that allows cross-machine and even cross-vendor and cross-operating system interoperability of X clients and servers. X11's network extensibility is in high contrast to the lack of network extensibility in Microsoft Windows. Most X clients run reasonably well locally *and* remotely. In most cases, the user is unaware when a client is running remotely. The success of X's transparent network extensibility has even created an entirely new network device: the X terminal [8].

The X community often lauds network extensibility over PC users of Microsoft Windows, but Microsoft Windows has gained widespread usage despite the lack of network extensibility.

For most PC users, contemporary PC technology provides plenty of power for tasks such as word processing, spreadsheets, and electronic mail. The idea of remote execution of windowed applications is considered interesting but frivolous. And in the homogeneous world of PCs, support for heterogeneous machines and operating systems is largely unnecessary.

For Microsoft Windows users, the network can be an important source of services (file systems, databases, and electronic mail) but the graphical front ends to such services can easily be implemented as a local Microsoft Windows program. Even when the storage or computational burden of an application like a database is beyond the capability of a desktop machine, the user's graphical application need be only an interface between the user and another computer managing the database. If services are sufficiently network extensible, a network extensible window system may not be necessary.

One should be careful not to draw the wrong conclusion. The point is not that network extensible window systems are unimportant, but that in a homogeneous environment using desktop computers with more than adequate power for common applications, network extensible window systems are not necessary. In a heterogeneous environment with users sharing computer resources beyond the capability of the machine on their desk, network extensible window systems have proven to be vitally important.

The important point is X11 is competing with window systems that are not network extensible but that derive important performance benefits from the optimization opportunities available to a purely local window system implementation.

2.2 Future Trends for High Bandwidth Applications

Future trends in computing suggest that local windowed applications may become more important. The bandwidth required for current and future interactive services such as audio, video, and three-dimensional graphics are difficult to meet without specialized protocols or workstation support. When an application requires the manipulation of huge quantities of video, audio, and graphical data at interactive and constant rates, distributing the application at the semantic level of window system operations may not be workable.

Future trends in the use of video, audio, image processing and three-dimensional graphics imply that the graphical and interactive components of applications using these capabilities will execute on the local desktop computer. Already most workstations and PCs have some capacity to handle high-bandwidth services; future machines will be even more capable. All these capabilities for video, audio, image processing, and 3D will still need to be networked, but specialized network protocols can be developed to meet the demands of these services.

If the local desktop computer is the most efficient location for the entire window system component of future applications, then a new window system must optimize the case of local window system applications. The effective division between client and server for future applications is less likely to be at the level of window system operations.

2.3 Protection and Reliability

An important benefit of client/server based window systems is the added protection and reliability that comes from isolating the window system server from the window system clients. A faulty client should not compromise the window system. Microsoft's Windows is moving in this direction. X has always had this type of protection.

Protection in X11 is achieved through isolating the window system server from its clients by putting the clients and server in separate address spaces. This isolation incurs a significant cost in cache misses, context switching, protocol packing and unpacking, and transport overhead.

A client/server architecture using separate processes is not the only means to achieve protection. D11 provides the same measure of protection as client/server based window systems but does so using the *active context* facility described in Section 3 and detailed in Section 5. D11 does not isolate the window system kernel from clients using isolated address spaces; instead D11 uses virtual memory techniques to allow the client to in effect *become* the window system kernel through controlled interfaces without a heavyweight process switch and change of address space.

2.4 X11 Compatibility

Considering future trends without regard to the past and present is dangerous. While X is not perfect [9], one cannot deny the enormous success of the X Window System. The X Window System now claims over 3.5 million X capable seats [2]. X has well-established, portable APIs well known to workstation programmers. X is supported by nearly all workstation vendors worldwide. The number of X tools and applications grows daily.

A new window system standard in the X tradition should not abandon the existing base of applications, toolkits, programmers, and users. For this reason, D11 retains complete X11 window system compatibility.

3 Reinventing X11

D11 is about reinventing X11 for higher performance. This does *not* mean abandoning compatibility with the current X11 protocol or X11 semantics or network extensibility. Instead, D11 concentrates on optimizing the local window system interface. Programs compiled to support the D11 interface *will* operate with current X11 servers, *and* all X11 clients will operate *unchanged* with a D11 window system. The substantive difference is

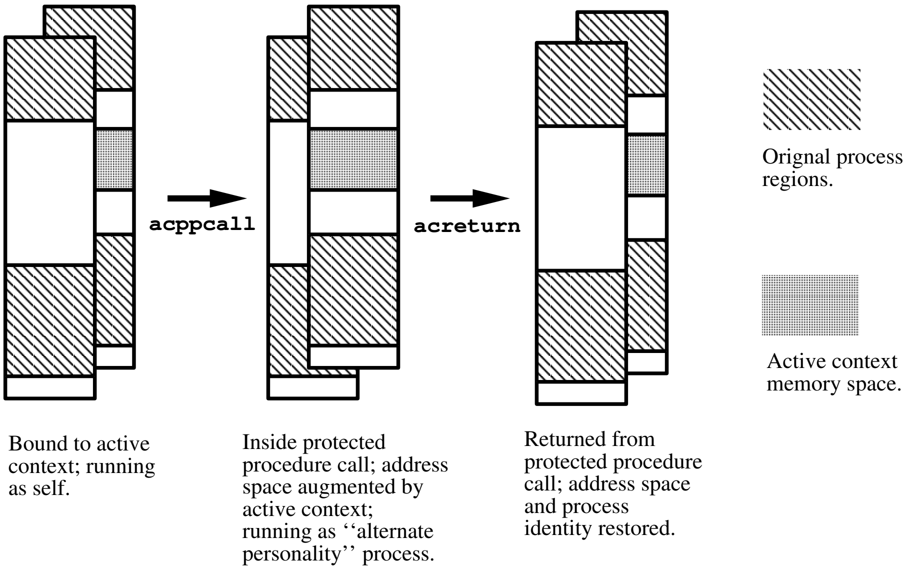

Figure 1: Calling and returning from a protected procedure call. The address space mapping on top indicates the current active address space mapping.

that D11 programs can achieve substantially better window system performance than equivalent X11 clients running locally.

The API for D11 is a superset of the X11 Xlib API. The D11 API is augmented with higher performance local interfaces for some existing Xlib operations. When a D11 program runs locally within the D11 framework, these interfaces allow window system operations to be performed in a manner optimized for the local case. When a D11 program runs remotely or connects to a local X11 server, the new D11 interfaces generate the X11 protocol to achieve the same effect at lower performance.

The D11 implementation does not structure the window system service as a separate process that clients communicate with via X11 protocol byte streams. Instead, the X server is reimplemented via a user-level kernel facility called active contexts. Instead of writing and reading X protocol through bytes streams, D11 programs call the D11 window system directly through a *protected procedure call* mechanism. A protected procedure call is a change of execution state into an active context where the program's address space is augmented by the memory regions for the active context and the process's identity (including all standard Unix resources like file descriptors and signal handlers) is transformed into that of the D11 window system kernel. On return from a protected procedure call, the program identity and address space are restored. Figure 1 shows the effect of a protected procedure call on a process's address space and identity.

The active context facility is very much analogous to the Unix kernel's traditional implementation. In Unix, when a process traps into the kernel via a system call, the processor vectors to a well-defined kernel system call handler and then continues running at a higher privilege level with an address space augmented by the kernel's memory regions. An important distinction between a Unix kernel and the D11 active context is that threads of control within an active context execute in user mode like other Unix processes, not in kernel mode like the Unix kernel. Because of D11's kernel style architecture, what one would call the X server in X11 is called the window system kernel in D11.

Logically, a D11 program is subscribing to a *service* for window system functionality. Likewise, one could consider a Unix process to be subscribing to the Unix kernel for operating system services. While D11 does implement a service, we avoid the term "client/server" when describing D11 because it usually suggests an implementation style with the client and server implemented as distinct processes.

It is worth noting that many micro-kernel architectures have been implemented that take this notion of Unix as a service to the logical conclusion of implementing Unix as a server process and emulation

Operation	User CPU usage	Kernel bcopy CPU usage	Rate (ops/sec)	Protocol round trip involved
x11perf -getimage500	9.8%	25.3%	29	√
x11perf -prop	19.4%	1.1%	1,870	√
x11perf -getimage10	33.3%	1.5%	1,450	√
x11perf -seg1	38.0%	22.6%	1,070,000	
x11perf -dot	38.2%	15.7%	1,840,000	
x11perf -putimage500	40.2%	22.2%	26	
x11perf -getimage100	40.7%	10.3%	421	√
x11perf -rect1	51.9%	15.2%	899,000	
x11perf -putimage100	55.2%	16.5%	529	
x11perf -seg10	57.4%	15.0%	771,000	
x11perf -putimage10	69.4%	6.5%	24,600	
x11perf -rect10	76.7%	8.0%	40,4000	
x11perf -ucirculate (4 children)	84.3%	3.2%	245,000	
x11perf -noop	86.3%	3.0%	431,000	
x11perf -seg100	† 86.7%	3.8%	182,000	
x11perf -ucirculate (50 children)	91.6%	1.9%	125,000	
x11perf -rect100	‡ 95.9%	0.0%	25,400	

Table 1: Indications of processor utilization bottlenecks for assorted X operations. These results are generated by kernel profiling of an R4400 150 Mhz SGI Indy running IRIX 5.3. User CPU usage includes both client and X server CPU usage jointly. During the measurements, the CPU was never idle. Notes: † includes 42.3% of time actually spent in kernel's stalling graphics FIFO full interrupt handler. ‡ likewise includes 84.0% of time spent in FIFO full interrupt handler.

libraries extracted from the kernel proper [11, 14]. The advantage of a client/server structuring in separate address spaces is modularity and distributability. The advantage of kernel style structuring is less data transfer and lower context switching overhead for potentially better performance in tightly integrated systems (like local window systems) than a client/server implementation. The duality of client/server "message-oriented" structuring and kernel style "procedure-oriented" structuring has been recognized for some time [19]. Reimplementing X as D11 should provide further evidence of this duality.

3.1 Protocol Optional, Transport Optional

D11's most important optimization is making the packing and unpacking of X11 protocol and its transport optional in the local case. Standard implementations of Xlib and the X server work by writing and reading X11 protocol over a reliable byte-stream connection (usually TCP or Unix domain sockets). Excepting local optimizations to these underlying transport mechanisms [10], the local and remote case use essentially the same mechanism.

Consider the work that goes into this packing, transport, and unpacking process. The individual steps underlying a typical Xlib and X server interaction look like:

1. The client program makes an Xlib call.

2. Xlib packages up the call arguments into X11 protocol in an internal buffer.

   Steps 1 and 2 repeat until Xlib determines the protocol buffers should be flushed. This happens because a reply needs to be received, an explicit `XFlush` call has been made, the buffer is full, or Xlib is asked to detect incoming events or errors. Buffering X11 protocol is an important Xlib optimization since it increases the size of protocol transfers for better transport throughput.

3. When a flush is required, Xlib writes the protocol buffer to the X connection socket, transferring the data through the operating system's transport code.

4. The X server has an event dispatch loop that blocks checking for input on client sockets using the `select` system call.

5. `select` unblocks and reports pending input from a client that has written to its X connection socket.

6. The protocol request is read by the X server, the type of the request is decoded, and the corresponding protocol dispatch routine is called.

7. The dispatch routine unpacks the protocol request and performs the request.

   If the request returns a reply, the sequence continues.

8. Xlib normally blocks waiting for every reply to be returned.

9. The X server encodes a protocol reply.

10. The X server writes the reply to the receiving client's X connection socket.

11. The client unblocks to read the reply.

12. The reply is decoded.

There are a number of inefficiencies in the local case of the protocol execution sequence above. Table 1 shows that a number of important X operations can spend from 25% to 90% of their time within the operating system kernel. Clearly, operating system overhead can have a substantial impact on X performance.

There are three types of operating system overhead that are reduced by D11:

Protocol packing and unpacking. The protocol packing and unpacking in Steps 2, 7, 9, and 12 are done strictly according to the X11 protocol encoding. Unfortunately, the protocol encoding is designed for reasonable compactness so 16-bit and 8-bit quantities must be handled and packed at varying alignments that are often handled relatively inefficiently by RISC processor designs.

Using protected procedure calls, a process directly passes D11 API routine parameters to the window system, skipping the inefficient packing and unpacking of protocol.

Transport. Moving X protocol from one process to another in Steps 3, 6, 10, and 11 requires reading and writing protocol by the X server and its clients. Protected procedure calls and an active context augmented address space allow data to be passed to and from the D11 window system kernel without any reading and writing of protocol buffers.

The kernel `bcopy` CPU usage percentages in Table 1 provide a lower bound to the transport overhead of X protocol transport. The `bcopy` overhead is over 20% for some important X operations. The `bcopy` overhead counts only the raw data copying overhead and not the socket implementation and `select` overhead.

Context switching. Because the client process generating X requests is not the same process as the one executing X requests, there is context switching overhead for the completion of every X request. Fortunately, protocol buffering allows this cost to be amortized across multiple requests, but the overhead of context switching still exists. For X requests that generate a reply forcing a "round trip," the kernel overhead due largely to context switching is quite high as Table 1 shows (up to 80% for the `x11perf -prop` test that is mostly context switching overhead).

Beyond the cost of the actual context switch, there is a cost due to cache competition between the client and server processes [6].

> Execution of a protected procedure call is in effect a context switch, but is hopefully a lighter weight context switch than the typical Unix process switch between contexts. Among the advantages of a protected procedure call over a Unix context switch is that no trip through the Unix scheduler is necessary and that most processor registers do not need to be saved.

As an example of how the transport-less, protocol-less D11 window system improves performance, consider an `XGetImage` call. In X11, such a call is expensive because it requires a client/server round trip representing two heavyweight context switches, three copies of the image (an image copy from the screen to an X server reply buffer; a copy from the reply buffer to the kernel; and a copy from the kernel to the client), and the overhead of protocol packing and unpacking. In a local D11 program, the same `XGetImage` call is implemented with a single protected procedure call (two lightweight context switches), no protocol packing or unpacking, and a single copy directly from the screen to the D11 protocol buffer (because the caller memory space is directly available within the active context).

3.2 Private X Resources

All X resources (windows, colormaps, cursors, graphics contexts, etc.) exist in a global name space. This is natural when the entire window system exists within X server. A second D11 performance advantage accrues from supporting "private" graphics contexts and pixmaps.

Rendering to pixmaps within the X server is inefficient because X requests are being generated, transported, and executed to manipulate what are essentially arrays of pixels within the X server's address space. Efficiency of rendering to pixmaps would be greatly improved if the pixmap resided in the client's address space. But the major advantage of pixmaps is that they can be used efficiently to transfer pixels to and from windows and other pixmaps via `CopyArea` requests.

D11 supports a private pixmap resource created by `XCreatePrivatePixmap` that is accessible only to the local D11 program that created it. The memory for the pixmap resides in the D11 program's address space, not within the D11 active context. Private pixmap rendering can be done without ever entering the D11 active context, greatly increasing the efficiency of pixmap rendering. The ease and efficiency of transferring pixels to and from other windows and pixmaps still exists because a D11 `CopyArea` request executing within the D11 active context can access the pixmap memory residing in the protected procedure caller's memory space. Use of private pixmaps by local D11 programs achieves both maximum rendering *and* `CopyArea` performance.

To support private pixmaps, private graphics contexts (GCs) are created by calling `XCreatePrivateGC`. Using public GCs with private pixmaps can work by reading the public GC state from the active context and using the public GC state as necessary.[1]

Remote D11 programs can always fall back to conventional "public" pixmaps and GCs with lower pixmap rendering performance, so support of private pixmaps and GCs does not compromise network extensibility.

Since most X11 clients would work fine if all their GCs and pixmaps are treated as private resources, a mode set by the D11 routine `XPrivateResourceMode` forces the standard `XCreatePixmap` and `XCreateGC` routines to create private resources. With a single new routine an X program recompiled for D11 could benefit from private resources.

3.3 Benefits

D11 optimizes local window system performance by reducing contexting switching overhead, eliminating transport overhead, eliminating packing and unpacking overhead, and improving pixmap rendering performance. Because D11 programs can fall back to X11 protocol for remote connections, network extensibility is preserved.

[1] The core X11 protocol does not allow the state of a GC to be returned to a client so special D11 support will be needed for this.

4 Other Approaches

The D11 architecture is not the only considered means of improving or reinventing the X Window System. We consider two other approaches and critically compare them to D11.

4.1 Shared Memory Transport

As previously discussed, the transport of X protocol between the client and server requires considerable CPU resources. A shared memory transport for X provides a faster means of transferring X11 protocol between local X clients and the X server by avoiding copying X protocol buffers. Many workstation vendors provide some sort of shared memory transport and typically report X benchmark results based on shared memory transport results.

Details on shared memory transport implementations are not well published because these transports often rely on proprietary operating system features and are treated as proprietary performance advantages.

In practice, the implementation of a shared memory transport is more involved than one might imagine. A simple reader/writer shared memory queue with semaphores is unacceptable because the X server must not be allowed to hang indefinitely on a semaphore or spinlock used to arbitrate access to the queue. The X server must fairly service requests and timeouts from multiple input sources. It must also robustly handle asynchronous client death and corruption of the shared memory queue.

Other logistical problems exist. The reasonable assumption of the MIT sample X server [1] is that each request is dispatched from a complete and contiguous request buffer. The X server could construct a complete and contiguous request by coalescing partial buffers from the shared memory transport into a contiguous buffer, but this obviates much of the benefit of the shared memory transport for large requests (where a shared memory transport is most beneficial) since the request data must be copied. For this reason, a shared memory transport implementation is likely to allocate a shared memory queue as large as the largest X request (most servers support the X protocol limit of a quarter megabyte).[2] To obtain better performance, some vendors allocate even larger shared memory queues. The result is a transport that performs well for benchmarks but has poor locality of reference within the shared memory queue. In a benchmark situation where a single shared memory connection is in use, this is not a problem. But if every local X client on the system is cycling through their individual shared memory queues, the result is poor memory system behavior hindering overall system performance.

Excepting a few important requests like `GetImage` and `GetProperty`, the server-to-client direction is not as critical to X performance as the client-to-server direction. This makes it likely that a shared memory transport will use shared memory only for the client-to-server direction. The Xlib necessity for a `select`-able file descriptor and the need to detect transport failure makes the use of a standard TCP or Unix domain transport likely for the server-to-client direction. Such an implementation decision restricts the performance benefit of a shared memory transport to the (admittedly more important) client-to-server direction.

Aside from the implementation issues with a shared memory transport, the performance potential from a shared memory transport is limited. While a shared memory transport reduces the transport costs, it does not reduce the two other system costs of window system interactions: context switching and protocol packing and unpacking. A shared memory transport may also introduce new memory system strains.

Table 1 indicates a number of important performance cases that a shared memory transport does not address. Notice that operations requiring a reply have high operating system overhead (80% for `GetProperty`; over 90% for large `GetImage`s). Such cases demonstrate the cost of context switching and sending data from server to client because each Xlib call waits for a reply, forcing a round trip between the client and server. Also notice that in no case is the kernel `bcopy` overhead larger than 25%. This places a fairly low upper bound on the performance potential of a shared memory transport because kernel data copies are the fundamental savings from a shared memory transport.

[2]This approach is complicated by the Big Requests Extension introduced in X11R6 [22].

4.2 Direct Graphics Hardware Access

Support for local clients to directly send graphics rendering requests to the graphics hardware without interacting with the window system proper is a technique known as *direct graphics hardware access* (DGHA). DGHA mitigates both context switching and protocol packing and unpacking overhead for the execution of rendering requests to windows; such rendering requests can be executed directly by DGHA clients. While DGHA helps the performance of window rendering requests, non-rendering requests such as `CreateWindow` and `AllocColor` continue to be executed by the X server. Whether pixmap rendering requests use DGHA techniques depends on the DGHA implementation, though most implementations probably do not support direct pixmap rendering.

Many workstation vendors implement some form of DGHA, particularly to support 3D rendering when graphics hardware might otherwise be under utilized. The techniques used vary from software locking of graphics for multiple rendering processes [5] to utilizing asynchronously context-switchable graphics hardware [26, 15, 17] that virtualizes graphics rendering [27].

Two major impediments to implementing DGHA for X are the global nature of X resources and the semantic restrictions of atomicity and sequentiality demanded by the X protocol.

The global nature of X resources greatly complicates DGHA for X rendering because X resources such as graphics contexts and pixmaps are allocated from a global name space accessible to all connections to the X server. Potentially, an X client can access the resources of any other client (in practice this is rare). This means DGHA implementations are violating the strict sense of the X protocol if they handle the state of X resources such as graphics contexts and pixmaps as local to their usage.

OpenGL's GLX extension for X [13] explicitly supports a form a DGHA known as *direct rendering* (as opposed to indirect OpenGL rendering that is executed by the X server). GLX provides this functionality by explicitly distinguishing direct resources that can be maintained within the client from indirect resources that are maintained globally within the X server. This solution is unfeasible for X DGHA implementations because it is impossible to redefine the existing global nature of X resources.

X atomicity demands that "the overall effect must be as if individual requests are executed to completion in some serial order." X sequentiality demands that "requests from a given connection must be executed in delivery order."[3]

Strictly interpreted, the atomicity requirement would make virtual graphics techniques employing asynchronous context switches illegal because atomicity could not be guaranteed. The expense of the atomicity requirement for rendering operations is unfortunate since few tangible benefits derive from it.

It is entirely expected for X protocol requests within a connection to execute in their generated order. But this sequentiality requirement forces extra synchronization when rendering requests are executed separately from the non-rendering requests. Effectively, DGHA means rendering and non-rendering requests are executed in two distinct streams

For example, a `MapWindow` non-rendering request could be generated to map an override-redirect pop-up menu, followed immediately by rendering requests to draw the menu. The X client must guarantee completion of the `MapWindow` request before using DGHA to execute the rendering requests or else the rendering will likely be done to an unmapped window. This implies an X11 protocol round trip must be performed after the `MapWindow`. The reverse guarantee (that non-rendering requests must not execute before rendering requests) must also be made, but it is easier for a client to know a DGHA rendering request has completed.

OpenGL's GLX explicitly distinguishes the OpenGL rendering stream from the X protocol stream, in effect relaxing X protocol semantics for OpenGL rendering. The `glXWaitX` and `glXWaitGL` routines allow a user to explicitly synchronize the streams.

For OpenGL, this explicit solution for supporting DGHA is reasonable, but for core X rendering the solution is unworkable because X programs implicitly rely on non-rendering and rendering requests executing

[3] Found in the "Flow Control and Concurrency" section of X11 protocol specification [21].

Client trace	Requests	Reply %	Render %	Description
mwm startup	2,389	25.5%	7.2%	Generic Motif window manager starting with 10 pre-existing toplevel windows.
4Dwm startup	3,362	26.8%	7.9%	SGI window manager starting with with 10 pre-existing toplevel windows.
xterm startup	74	24.3%	21.6%	Athena widget based terminal emulator
editres startup	143	16.8%	33.6%	Athena widget resource editor
apanel startup	799	20.9%	48.9%	Motif 1.2 SGI audio control panel startup
chost startup	1,047	21.39%	50.4%	Motif 1.2 SGI host management tool using enhanced Motif
glp startup	819	15.7%	55.2%	Motif 1.2 SGI printer graphical front-end
xrn-motif startup	608	3.9%	56.3%	Motif 1.1 news reader
MediaMail startup	2,414	7.1%	56.9%	Motif 1.2 SGI version of zmail GUI
xwsh startup	265	18.1%	59.2%	SGI terminal emulator
cdman startup	670	18.4%	66.3%	Music CD player using internal SGI toolkit
xfig startup	1,718	4.9%	64.5%	xfig 2.0 public domain drawing program
dekstopManager startup	2,675	13.4%	70.2%	Motif 1.2 SGI Indigo Magic file manager viewing home directory with many files
mwm startup, then usage	3,482	23.6%	14.9%	Generic Motif window manager starting with 10 pre-existing toplevel windows, then user interaction
4Dwm startup, then usage	6,391	15.9%	27.1%	SGI window manager starting with 10 pre-existing toplevel windows, then user interaction
xrn-motif startup, then usage	16,219	0.4%	77.4%	Motif 1.1 news reader news reading session

Table 2: X protocol request ratios for various X client protocol traces. A render request is a core protocol request potentially appropriate for DGHA use; that is, it manipulates graphics contexts or reads or writes pixels excepting `ClearArea`.

sequentially as expected.

Contrasting D11 to DGHA approaches, D11 requires no semantic changes to the X11 model (though relaxing the atomicity requirement for rendering is probably useful) and, unlike DGHA, D11 does not segregate the execution of rendering requests from non-rendering requests so no extra synchronization is needed between the two requests types. DGHA helps only DGHA-capable rendering requests while D11 helps all classes of requests. Table 2 shows that X clients combine rendering and non-rendering requests, particularly during program startup. The results show that for many clients, more than 40% of the requests are non-rendering requests that DGHA will not improve. It also indicates the likely cost associated with synchronizing DGHA requests with non-DGHA requests. The ratio of replies to requests indicates that up to 25% of requests require replies, generally each necessitating a client/server round trip, an expensive operation that gets no benefit from DGHA. The percentage of rendering requests is *overstated* because it also includes pixmap rendering and `CopyArea` that many DGHA implementations will likely not improve.

5 The Active Context

The active context is the operating system facility that underlies the implementation of D11. D11 requests from local D11 programs are translated into protected procedure calls that enter the D11 window system kernel. Because protected procedure calls can be implemented so that they are inexpensive relative to a standard Unix context switch and no protocol needs to be packed, sent through a transport, and unpacked, D11 protected procedure calls should be a net win over local use of the X11 protocol using conventional transports. This section details the working of active contexts and protected procedure calls for a Unix style operating system.

5.1 Initiating an Active Context

To initiate the D11 window system, a standard Unix process "bootstraps" the active context for the D11 window system kernel. The process requests that the operating system create the D11 active context. The active context is a reserved range of user-space memory. The call to create an active context looks like:

```
fd = acattach("/tmp/.D11-ctx:0", AC_CREATE, 0x50000000, 0x10000000);
```

This call creates an active context, "attaches" the active context to the calling process, and reserves a 256-megabyte range of virtual memory in a normally unallocated region of user address space. File system path names provides a convenient name space for active contexts, but active contexts are not otherwise reliant on the file system. Only privileged processes may create an active context.

The creator of an active context or a process running inside the active context can use the `mmap` family of system calls [12] to allocate, deallocate, and control memory within the active context. The effect of these calls is seen by all users of the active context. After an active context is created, the creator is expected to "populate" the context with text, read-only data, and writable data before enabling protected procedure calls to enter the active context.

The D11 bootstrap program `mmap`s the text and static data of the actual D11 window system kernel. The bootstrap program also establishes file descriptors, synchronization primitives (such as semaphores and spinlocks), and memory mappings for the graphics hardware and input devices.

The *entry point* for an active context is the address of initial execution when a process enters an active context via a protected procedure call. Initially, there is no entry point. The entry point to the active context is set by calling:

```
accontrol(fd, AC_SET_ENTRY, entry_func);
```

`entry_func` is the address of the routine within the active context to be used as the active context entry point.

To make it possible for other processes on the system to attach to and make protected procedure calls into a newly created active context:

```
accontrol(fd, AC_ALLOW_ENTRY, /* true */ 1);
```

must be called and an entry point must be established. The `accontrol` can be called only by the creator of an active context.

5.2 Protected Procedure Calls

Once an active context is set up and entry by other processes is allowed, other processes can call `acattach` (without the `AC_CREATE` flag) to attach to the active context. For example:

```
fd = acattach("/tmp/.D11-ctx:0", 0, /*ignored*/ 0, /*ignored*/ 0);
```

Attaching to an active context creates an "alternate personality" process (with a slot in the system process table). This *alternate personality process* (APP) exists as a standard Unix process and is a child of the active context creator (*not* the process attaching to the active context). An APP inherits file descriptors, signal handlers, the effective uid, and other Unix process attributes from the active context creator. In many ways, the APPs associated with an active context are similar to IRIX *process share groups* [3, 24] to the extent that they share Unix system resources with one another in a way consistent with expected Unix operating system semantics.

A process attached to an active context can initiate a protected procedure call to "switch personalities" to the process's APP by calling:

```
status = acppcall(fd, op, ...);
```

The result of `acppcall` is that the calling process is suspended and execution is handed off to the process's APP which begins execution at the active context entry point. The entry point is called with the following parameters:

```
entry_func(app_id, op, ...);
```

The `app_id` is a small unsigned integer indicating what APP is running. The `entry_func` uses this parameter to determine the identity of the APP within the active context. The `op` parameter is an integer indicating the operation to perform.

An APP returns from a protected procedure call by calling:

```
acreturn(status);
```

Returning from a protected procedure call suspends the APP and resumes the process initiating the protected procedure call. `status` is an unsigned integer returned as the successful return value of `acppcall`.

Multiple APPs can enter the active context simultaneously, so shared synchronization mechanisms within the active context should be used to coordinate execution within the active context.

5.3 Augmenting the Address Space

While most Unix process resources of an APP are inherited from or shared with the active context creator, the address space of an APP is the caller's address space augmented by the active context memory range. The benefit of this augmentation is that data can be read and written directly into the caller's address space during a protected procedure call.

The active context memory range is mapped in a fixed address space range for all APPs using the active context. This means the code within the active context can be position dependent and pointers to data within the active context need no translation. The active context address range can be thought of as a *sub-address space* shared by all APPs.

For D11, protected procedure calls into the D11 active context mean that no protocol packing, transport, or unpacking is necessary for local D11 programs entering the D11 window system.

The augmented address space does mandate a number of rules for writing code running within an active context to ensure robustness. These rules are analogous to the rules Unix kernel programmers follow to ensure robustness in the kernel. Active context code should *not* call routines in the non-active address space range. Reads and writes through pointers supplied by the protected procedure caller *must* be verified as outside the active context address range. Signal handlers must be established to trap memory access violations due to accessing memory outside the active context address range and remedy the situation.[4]

5.4 Active Context Notification

The signals delivered to an APP are distinct from signals delivered to the calling process. While running in a protected procedure call, Unix signals posted for the calling process that would not result in immediate process death are not delivered until `acreturn` is called. If a signal resulting in immediate process death

[4] Windows NT style *structured exception handling* language support [7] might be useful to make explicit in the code handling of memory access violations when accessing memory outside the active context.

such as SIGKILL is posted for a calling process, the calling process dies immediately, but the APP continues running until acreturn is called at which point the APP terminates. One ramification of this is that protected procedure calls should not block indefinitely within the active context. For this reason, some means must exist for active context users to block on events occurring within the active context.

The file descriptor for the active context is not usable for I/O operations but is select-able and can be used to receive notification of state changes within the active context. Any process running within an active context can call:

```
acnotify(app_id);
```

to cause select to indicate that the active context file descriptor of the specified APP is "readable." Calling acnotify is usually preceded by some indication in a shared data structure within the active context of why the notification was delivered. In response to an acnotify, a process should perform a protected procedure call into the active context to determine why the signal was delivered. The call:

```
acack();
```

will acknowledge a posted active context notification for the calling APP.

5.5 Implementation Issues for Active Contexts

The acattach family of routines can be implemented as new Unix system calls. The most critical aspect of implementing active contexts is ensuring protected procedure calls are as fast as possible. Techniques for fast, local cross-address space communication such as *lightweight procedure calls* [4] are readily applicable to optimizing active context protected procedure calls. The most important difference between a protected procedure call and other cross-address space procedure call mechanisms is the augmented address space.

Because APPs exist as distinct process table entries within the Unix kernel, current Unix semantics can be cleanly mapped to APPs. Most areas of the Unix kernel can be completely unaffected by the introduction of active contexts.

Current RISC processors supporting a *virtual cache with keys* [23] should make possible efficient address space augmentation for protected procedure calls. The protected procedure caller and its APP can each maintain distinct *address space ids* (ASIDs). The acppcall and acreturn operations need to change the ASID appropriately.

This scheme treats the augmented address space as a completely distinct address space which is somewhat inefficient in translation lookaside buffer (TLB) entry usage. Processor support for distinct user and kernel modes is very similar to the address space augmentation needs of protected procedure calls. Unfortunately, kernel mode is quite inappropriate for the user-level execution of active contexts. Some processors like the MIPS R4000 [20] do provide a *supervisor* mode with more limitations than kernel mode. But MIPS supervisor mode enables access to memory outside the typical user range and is probably still too permissive for use in implementing active contexts.

The reservation of a large chunk of user address space for the active context memory range may be an issue for 32-bit processors that are limited to two gigabytes of user address space. Support for 64-bit address spaces obviates this problem. Also in such systems, support for *protection domains* [18] may also reduce the cost of address space augmentation by treating protected procedure call exit and entry as a lightweight protection domain switch.

6 Implementing D11

We begin by discussing D11's implementation for local D11 clients; support for remote D11 programs and X11 clients is discussed afterwards.

6.1 Within the D11 Active Context

The D11 window system kernel is implemented as an active context and bootstrapped as described in the previous section. The executable code for the D11 kernel can be largely reused from the X11 server's implementation. Code that needs to be reimplemented for D11 includes initialization, the main event loop, and client I/O code. The vast bulk of the X server code such as machine-independent and device-dependent rendering code, font handling code, etc. can be identical to the X11 server's implementation.

Once the active context creator "populates" and activates the active context, it takes the role of a housekeeper for the active context. It manages window system kernel tasks that are not initiated by D11 program requests. For example, reading input events, handling the screen saver, managing window system reset, and reaping terminated APPs are all done by the D11 kernel housekeeping thread.

D11 programs attach to the D11 active context during `XOpenDisplay`. The first "establish connection" protected procedure call enters the D11 kernel and establishes the data structures for the new D11 connection. Within the D11 active context is a global locking semaphore that is acquired whenever a process running within the active context manipulates global data structures. Acquiring this semaphore while executing requests satisfies the X atomicity requirement. While a global lock may seem too coarse, the request dispatch loop of current single-threaded X servers is logically equivalent to a global lock.

Once a connection is established, the D11 program can make protected procedure calls. Based on the `op` parameter to the `entry_func`, the D11 kernel executes the correct code to handle the D11 request. When executing a request, the APP may directly manipulate the caller's address space to read arguments and return data (such accesses must be correctly protected against memory violations).

Because multiple processes may run in the active context at one time, code within the D11 active context must correctly lock accesses to data structures. Because of the use of a single global lock, most locking concerns are straightforward. Existing X server code for executing requests can generally assume the global lock is already acquired and so existing code does not need any special locking. Inexpensive locking around the global memory allocation pool is necessary. Multi-threaded and multi-rendering X servers [16, 25] already abide by this requirement.

So a typical D11 request is executed by generating a protected procedure call that enters the D11 active context, acquires the global lock, calls the appropriate request execution code, releases the lock, and returns from the protected procedure call. In the process of executing a request, data may be transferred from the caller's memory space; and if the request returns a reply, data may be written into the caller's memory space.

As mentioned earlier, the D11 active context creator is watching for asynchronous events such as mouse or keyboard input. When such input arrives, the housekeeping process will acquire the global lock, determine how to handle the input, generate events to be sent to D11 connections if necessary, and release the global lock. Generating events for a D11 connection is done by placing events in the connection's pending event queue. Each queue has a lock that arbitrates access to the queue. Any D11 active context process can send events by acquiring the lock for a D11 connection's event queue, appending the events to the queue, and calling `acnotify` for the APP owning the queue, then releasing the queue lock. Errors are also placed in the queue. But replies are returned synchronously.

6.2 The D11 Interface Library

In the local case, the D11 interface library (essentially Xlib for D11) translates D11 requests into D11 active context protected procedure calls. In the case of a remote D11 kernel or X11 server, the D11 interface library uses the X11 protocol. A per-connection local or remote jump table is used to vector D11 request generating routines to the correct local or remote request generator.

A D11 program can use its active context file descriptor to determine when events or errors have been placed in its D11 connection event queue. By using the `select` system call on the D11 program's active context file descriptor, the program can determine when events are pending because `acnotify` is called whenever events are appended to a D11 program's event queue. `XNextEvent` and `XPending` can block and poll for events respectively using this mechanism. A "return events and errors" protected procedure call copies all the events and errors from a D11 connection's event queue into the caller's memory space. `acack` acknowledges active context notification. This is actually more efficient than Xlib's event reading strategy that requires a 32-byte `read` system call per event, reply, or error received to ensure the length field is decoded before any more data is read so variable length replies can be read into contiguous buffers. D11 makes the returning of events and errors more efficient than in X11.

6.3 Handling Client and Server Termination

If a D11 program terminates, its associated APP is terminated cleanly. If the APP is currently executing when its associated D11 program terminates, the protected procedure call executes to completion before the APP terminates. If the APP terminates unexpectedly (because of a memory fault or termination by a signal), the `acppcall` returns with an error indicating the active context has become unattached (the window system has crashed). When the APP terminates (because of D11 program death or a software fault), the active context creator is notified it should reap the APP. If the APP died cleanly (because its associated process terminated), the housekeeping thread will clean up the data structures associated with the APP's D11 connection. If the APP died of a software fault, the reaper will generate a fatal window system error.

6.4 X11 Protocol to D11 Conversion

A critical compatibility concern is how to make old X clients (and remote D11 programs) that generate X11 protocol work with the D11 window system. The solution is an X11-to-D11 gateway process that converts X11 protocol requests to the X server's standard ports (such as TCP port 6000) into D11 protected procedure calls. This gateway process is a standard Unix process that attaches to the D11 active context. The process makes "raw protocol" protected procedure calls that pass raw protocol buffers to the D11 kernel to be unpacked and executed. A single gateway process could broker multiple X11 protocol connections simultaneously, with each connection having its own associated APP. The gateway process is also responsible for correctly returning events, replies, and errors to the appropriate X11 protocol connections.

One might think the X11-to-D11 gateway would be inefficient relative to a traditional X server. This does not have to be the case. Because D11 protected procedure calls require no copying in passing protocol to the D11 kernel, there is no more protocol copying than when using a traditional X server.

6.5 Implementation Concerns

Several concerns about D11 usage must be addressed by a real implementation. D11 requests are executed synchronously. A multiprocessor workstation could *potentially* achieve a performance benefit from the current client/server implementation of X11 because an X client and the X server could run in parallel on different processors. Potentially, D11's lower operating system overhead could make up for the advantage of X11's client/server parallelism.

A given implementation of protected procedure calls may not be fast enough. Xlib's buffering of protocol allows the operating system overhead to be amortized across several Xlib calls. If protected procedure calls are too expensive to be done per D11 request, a similar buffering scheme may be necessary for D11 requests. This reintroduces the cost of protocol packing and unpacking but retains the other performance benefits of D11.

An APP per window system connection may be rather expensive in terms of the APP's associated kernel data structures. The cost of APPs should be minimized.

7 Conclusions

D11's kernel style window system architecture promises significant performance benefits over X11's current client/server architecture involving clients and servers running in separate address spaces. The performance benefits of D11 come from reduced context switching overhead, the elimination of local protocol packing and unpacking, the elimination of local transport overhead, and more efficient support for pixmap rendering.

D11 does require significant operating system support for active contexts, alternate personality processes, and protected procedure calls. The performance of protected procedure calls is vital to fulfilling D11's promise. Implementing the window system component of D11 is eased because most of the existing X11 client and server code can be reused in a D11 implementation.

Comparing D11 with other methods for improving window system performance, D11 reduces system overhead for local window system usage in all significant areas. D11 can be considered a logical extension of DGHA techniques to encompass the full range of X requests, not just rendering requests. D11 can also be considered the next logical step beyond a shared memory transport, where not only is the copy of data from the transport buffer eliminated, but so is the copy into the transport buffer and all associated protocol packing and unpacking.

No compromises in network transparency, X11 compatibility, or protection and reliability are made by D11. D11 favors local window system usage at a time when trends for high-bandwidth applications will likely result in greater reliance on and need for local window system performance. D11 optimizes local window system performance currently unharnessed by X11.

References

[1] Susan Angebranndt, Raymond Drewry, Phil Karlton, Todd Newman, Bob Scheifler, Keith Packard, Dave Wiggins, "Definition of the Porting Layer for the X v11 Sample Server," *X11R6 documentation*, April 1994.

[2] Stephen Auditore, "The Business of X," *The X Journal*, March 1994.

[3] Jim Barton, Chris Wagner, "Enhanced Resource Sharing in Unix," *Computing Systems*, Spring 1988.

[4] Brian Bershad, Thomas Anderson, Edward Lazowska, Henry Levy, "Lightweight Remote Procedure Call," *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, December 1989.

[5] Jeff Boyton, et.al., "Sharing Access to Display Resources in the Starbase/X11 Merge System," *Hewlett-Packard Journal*, December 1989.

[6] Bradley Chen, "Memory Behavior for an X11 Window System," *Usenix Conference Proceedings*, January 1994.

[7] Helen Custer, *Inside Windows NT*, Microsoft Press, 1993.

[8] Björn Engberg, Thomas Porcher, "X Window Terminals," *Digital Technical Journal*, Fall 1991.

[9] Hania Gajewska, Mark Manasee, Joel McCormack, "Why X Is Not Our Ideal Window System," *Software Practice and Experience*, October 1990.

[10] Michael Ginsberg, Robert Baron, Brian Bershad, "Using the Mach Communication Primitives in X11," *Usenix Mach III Symposium Proceedings*, April 1993.

[11] David Golub, Randall Dean, Alessandro Forin, Richard Rashid, "Unix as an Application Program," *Usenix Conference Proceedings*, June 1990.

[12] William Joy, et.al., "Berkeley Software Architecture Manual 4.3BSD Edition," *Unix Programmer's Supplementary Documents*, Volume 1, 1987.

[13] Phil Karlton, *OpenGL Graphics with the X Window System*, Ver. 1.0, Silicon Graphics, April 30, 1993.

[14] Yousef Khalidi, Michael Nelson, "An Implementation of Unix on an Object-oriented Operating System," *Usenix Conference Proceedings*, January 1993.

[15] Mark Kilgard, "Going Beyond the MIT Sample Server: The Silicon Graphics X11 Server," *The X Journal*, SIGS Publications, January 1993.

[16] Mark Kilgard, Simon Hui, Allen Leinwand, Dave Spalding, "X Server Multi-rendering for OpenGL and PEX," *Proceedings of the 8th Annual X Technical Conference* appearing in *The X Resource*, January 1994.

[17] Mark Kilgard, David Blythe, Deanna Hohn, "System Support for OpenGL Direct Rendering," submitted to *Graphics Interface '95*, May 1995.

[18] Eric Koldinger, Jeffrey Chase, Susan Eggers, "Architectural Support for Single Address Space Operating Systems," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[19] Hugh Lauer, Roger Needham, "On the Duality of Operating System Structures," *Proceedings of the 2nd International Symposium on Operating Systems*, October 1978, reprinted in *Operating Systems Review*, April 2, 1979.

[20] MIPS Computer Systems, *MIPS R4000 Microprocessor User's Manual*, 1991.

[21] Robert Scheifler, James Gettys, *X Window System*, 3rd edition, Digital Press, 1992.

[22] Robert Scheifler, "Big Requests Extension," Vesion 2.0, *X11R6 documentation*, 1994.

[23] Curt Schimmel, *Unix Systems for Modern Architectures*, Addison-Wesley, 1994.

[24] Silicon Graphics, *Parallel Programming on Silicon Graphics Computer Systems*, Version 1.0, Document Number 007-0770-010, December 1990.

[25] John Allen Smith, "The Multi-Threaded X Server," *The X Resource: Proceeding of the 6th Annual X Technical Conference*, O'Reilly & Associates, Winter 1992.

[26] C. H. Tucker, C. J. Nelson, "Extending X for High Performance 3D Graphics," *Xhibition 91 Proceedings*, 1991.

[27] Doug Voorhies, David Kirk, Olin Lathrop, "Virtual Graphics," *Computer Graphics*, August 1988.