For: rees
Printed on: Tue, Oct 13, 1992 17:43:04

Document: vstreams
Last saved on: Thu, Jun 18, 1987 11:03:24

To be presented at Summer 1987 Usenix Conference, Phoenix, Arizona.

A Dynamically Extensible Streams Implementation

Jim Rees, Margaret Olson, J. Sasidhar

apollo computer inc
330 Billerica Road
Chelmsford, MA 01824

Introduction

System V Streams [1,2] provides a uniform execution environment for communications protocols in the Unix operating system [3]. Our implementation is based on the Domain system, and takes advantage of several features of that system to make it easier to write, debug, install and configure Streams modules and drivers.

Streams modules, buffers, and queues all reside in user global address space, and are accessible from any user process. They are not bound in with the operating system kernel. Modules and drivers are self−contained and self−describing. New protocols may be added to any configuration without rebuilding anything, and without any knowledge of the existing configuration.

Protocols are accessible from both the Streams defined interfaces and from existing Domain interfaces, including the Open System Toolkit [4]. This is made possible by a global Type Manager that manages all Streams protocol types.

Goals

We had several goals in mind for this project:

- Provide a consistent base for implementing communications protocols.
- Make it easy to port Streams modules and drivers from other systems.
- Make it easy to install and configure Streams modules and drivers.
- Provide performance at least as good as that of existing communications products implemented on the Domain system.

The Streams Environment

A *stream* consists of a *stream head*, a *driver,* and zero or more *modules*. When a stream is first opened, it connects a user process directly to the driver, which may be a real device driver, a pseudo−device driver,

Unix is a trademark of AT&T.
DOMAIN is a registered trademark of Apollo Computer Inc.

Copyright 1987 Apollo Computer Inc.

or a *multiplexor*. The stream head implements the familiar Unix I/O primitives, such as `read` and `write`. It transforms the procedural interface provided by the I/O system into the message passing interface understood by the modules. The modules each consist of two *queues*, providing for full−duplex processing in the *upstream* (toward the head) and *downstream* (toward the driver) directions. The modules communicate with the head, the driver, and each other by passing messages contained in *message blocks.* Modules are connected together in a stack, and may be pushed and popped on the stream by making an `ioctl` call.

A multiplexor is a type of device driver that may have several streams attached to it both on the upstream and downstream sides. Multiplexors are typically used to implement transport providers, such as TCP.

Although originally designed as a replacement for the `clist` structs used by the tty routines in the Unix kernel, Streams is currently used primarily as the foundation on which to build communications protocol packages. The protocols are implemented as multiplexors and modules, and are connected between client user processes and network devices. The complete stack of modules and multiplexors implementing a particular protocol is called a *protocol stack*.

In System V, the modules, drivers and message blocks are statically configured into the kernel. Adding a new module or driver, or changing the size of the message block pool, requires configuring a new kernel and rebooting. The modules and drivers run in a kernel context, so debugging requires the use of kernel debugging tools and skills. An errant module or driver may cause the system to crash. On those systems that do not support kernel paging, all of the code and data must be resident in physical memory, even if it is not used in a particular configuration.

Our implementation of Streams runs entirely in user space. Modules and drivers are loaded into user global address space at boot time, but are not part of the kernel. They are demand paged into virtual memory as they are needed. The standard user debugger can be used to operate on them. Kernel data structures are protected from faulty modules by the same kernel address space protection boundary that protects the kernel from all user space code.

The Unix System V Kernel Environment

Streams modules and drivers make certain assumptions about the environment in which they run that are not valid in a user space context. Validating these assumptions is one of the more difficult problems involved in moving the Streams system to user space.

Unix kernels are typically single threaded. System calls running on behalf of a user process run to completion without the possibility of another system call running at the same time, unless the first system call explicitly relinquishes use of the processor. This greatly reduces the need to put mutual exclusion locks on kernel data structures during critical sections of code. User processes, however, are time shared. A user process may be suspended at any time to allow another user process to run. We added a set of mutual exclusion locks to prevent multi−threading of critical sections of code.

The Unix kernel runs in a single address space. Our user space implementation runs in many different user address spaces, depending on which context happens to be active at the time a module is scheduled. Modules are intended to get pointers to their message blocks from a procedural interface, but there is no inherent reason why a module can't save a pointer to some piece of storage in a static data structure, and access it later from a different context. We constrain the Streams data structures to appear at the same place in every user address space context.

Several kernel data structures are available for inspection by Streams modules, including the u. area and the process table. These data structures would not normally appear in a user context. If the kernel provided procedural interfaces to inspect these data structures, better data abstraction could have been provided. As it is, our implementation must generate these data structures every time a stream is opened or a module is pushed, since the system has no way of knowing whether the module will access the data structure or not.

Module Loading

The Domain system boot loader is capable of loading object modules into user global address space. Any code or data appearing in this part of the address space is accessible at the same address from any user process. The procedure text (code) is simply mapped in, read−only, and demand paged from the file containing the object module. Data sections are loaded into per−process private address space, so that they appear at the same address in every process, but represent a different, private piece of storage for each process. This mechanism is used to implement global libraries.

Streams modules are not bound to any particular user process context. They expect that their static data sections will map to the same piece of real storage regardless of what context they are running in. We have modified the boot loader to load the static data sections of Streams modules into global read−write storage. The loader runs in user space and obtains its storage from a memory allocator that can allocate from either private or global space, so this was an easy modification to make.

Streams modules and drivers are added to a system by placing the object code that implements them into the `/sys/modules` directory. At the time the node is booted, the boot code loads any object modules it finds in this directory into user global address space. For convenience, sets of related modules and drivers are often bound together into a single object module.

Since the modules are loaded into user space and run in a user context, the standard debugger can be used to operate on the modules and their data structures. Although debugging in this way is not as easy as debugging a standard user program, it is not as difficult as debugging a kernel based Streams implementation would be.

Module Initialization

Each module and driver (or group of related modules and drivers) has an initialization routine that the boot program calls after loading the module. As part of its initialization, each module makes a call to declare itself to the rest of the system. One of the arguments to this call is a pointer to the `streamtab` data structure for that module. The `streamtab` contains pointers to information completely describing the module, including pointers to its `put, service, open` and `close` routines, and a pointer to the `module_info` structure. As each module is declared, the system builds a list of available modules in the `fmodsw` table, and keeps a list of pointers to the `streamtab` structures of those modules.

Each device driver also makes a call at initialization time to declare information about itself. This information is described in the section on device drivers.

The Domain I/O System

The Domain I/O system, or *IOS* [5], is based on an object oriented, UID named, typed file system. *Objects* (files and devices) are named by a Universal Identifier (*UID*), which can be thought of as a

device/inode pair that is unique across all machines for all time. Every object has a *type*, which is also named by a UID. When a file is opened for I/O, the `open` system call looks up the UID of the object and locates a *type manager* for that type of object. The type manager has entry points for each of the standard I/O *operations*, such as `read` and `write`. When a user program makes one of the standard I/O calls on an open stream, the system invokes the manager to perform the operation on that object. New types of objects can be defined and added to the system dynamically by writing a manager for that type and installing it in a well−known place (`/sys/mgrs`) without having to rebuild the kernel or reboot the machine.

Operations are grouped into logical sets, called *traits*, according to the generic kind of object they operate on. `Open` is part of the *OC* (open/create) trait. `Read, write, seek,` and the other generic I/O operations are part of the *IO* trait. All useful managers implement at least the OC and IO traits. Managers may also optionally implement any of several other pre−defined *auxiliary traits*. For example, all tty−like types implement the *TTY* trait, and all network protocol−like types implement the *socket* trait. The socket trait is modelled after the Berkeley socket call interface. As part of its initialization, each type manager declares to the system which traits it implements on the types that it supports.

The Streams Type Manager

Our Streams system provides a type manager that implements many of the pre−defined traits on behalf of all of the Streams device drivers. Each Streams device driver has an associated type UID which is stored in the file system. The file system implements a mapping between device names and the associated type UIDs. The Streams type manager currently supports the OC, IO, TTY, and socket traits for all the Streams device types.

As part of its initialization, the Streams type manager declares support for the OC trait for all Streams devices known to the system, as listed in the `cdevsw` table. The `cdevsw` table contains entries for all the streams devices configured for the node, and is built at boot time as the device drivers are loaded.

For each Streams device the type manager declares the auxiliary traits for that type of device as listed in the `cdevsw` table. Each module may also support one or more auxiliary traits. These traits are listed for each module in the `fmodsw` table, which is built dynamically as the modules are loaded. Support for these auxiliary traits is declared when a module is pushed on a stream. The list of traits supported by a particular device is kept on a per−process basis. If a device has had the TTY trait declared for it as a result of a TTY processing module having been pushed on top of it, for example, then only the process that pushed the TTY module will be able to use TTY operations on that device.

When an application opens a Streams device, the type manager's open operation does the work common to all the streams devices, including allocating a queue pair for the head of the stream, and then calls the device driver's open routine through the pointer in the `cdevsw` table.

The Streams type manager implements most trait calls by converting them to messages, sending them downstream, and awaiting a response. Some trait operations, such as creating a socket, require pushing a special module that interprets the messages peculiar to that trait. There is a duality between the procedure call oriented Domain trait mechanism and the passing of messages in the streams environment.

The operations of the transport library interface (TLI) have been recast as a trait to extend the functionality to the rest of the Domain system. It should be possible for those managers that now support the socket trait to also support the TLI trait by adding TLI operations to the manager.

All `ioctl` calls are mapped into the appropriate trait calls by the C library. If the stream is managed by the Streams type manager then the manager converts them to the proper IOCTL messages.

Device Drivers

Non−streams device drivers in the Domain system are implemented in user space, and are not necessarily connected to the I/O system. Their open, close, read and write entry points may be called by any user program directly, and typically present a device−dependent interface. To connect the device driver to the I/O system in a device−independent way, a type manager must be written. For non−streams devices, a separate type manager is usually provided for each type of device.

The interface between the type manager and the device driver is similar to the device driver interface in System V. The driver's initialization routine makes a device declaration call to export to the system the entry points for the `cdevsw` structure, which is built dynamically as drivers are loaded. The driver also exports an *event count* and a list of traits it is willing to support.

The event count is the basic process synchronization primitive in the Domain system [6]. System calls are provided to read, advance, and wait on event counts. Streams device drivers need not be real hardware drivers, but may be layered on top of some other system service, such as a low−level datagram service. These services typically export an event count to their clients. The driver itself has no process context associated with it, so it exports the event count to the Streams system, which will then call the driver's interrupt routine when the event count is advanced.

The Domain system currently has a limit of 32 event counts on which any process can wait at one time. This puts a global upper limit on the number of device drivers that may be open at one time, since all driver event counts are waited on by a single process.

Service Procedure Scheduling

The module scheduler is called by the type manager just before the return from any operation. At this time any module with work to do will get a chance to run. In addition, a server process provides a context in which timeouts and upstream events can run.

Each Streams driver exports an event count to the server. When an upstream event occurs, the event count is advanced, and the server wakes up and calls the driver's interrupt routine. Processing for downstreams events, generated as a result of some user program making a system call, run in the context of the process generating the event. This reduces context switch overhead, and can even eliminate the need to switch from user to kernel context in some cases.

Streams modules are subject to user process scheduling and suffer from synchronization problems. To provide the necessary synchronization, Streams queues and data structures are locked with a set of mutual exclusion routines provided by the underlying operating system [7]. The locks are transparent to the modules themselves. The current implementation locks all queues with a single lock.

Streams module procedures may execute in the context of the client process, the Streams server process or an unrelated user process. Downstream puts typically run in the context of the putting process. Upstream puts typically run in the context of the server process. A put called from a service procedure runs in the context in which the service procedure is being executed.

If a putting process enables a queue then the service procedures are usually run before returning from the system call. The execution of the service procedures may be delayed if another process had started

the execution of the service procedures and was interrupted. The service procedures will then execute when the interrupted process regains control. At any particular time only one process can be executing service procedures and these procedures are executed in a round robin fashion.

The server process is started at boot time, after the modules and drivers are loaded. It is an ordinary user process. After performing some initialization work, it sleeps on a set of event counts, including all the device driver event counts and a timer event count. When it wakes up, it calls all the driver interrupt routines whose event counts have been advanced, runs any timeout routines that are due, and runs the module scheduler. The scheduler then runs any service procedures that have been enabled.

Configuration Parameters

The Domain implementation of Streams takes advantage of the virtual memory facilities to reduce the configuration work required of the system administrator. There are three classes of tunable parameters related to Streams. The approach taken for each class is given below.

Parameters that control the sizes of static tables, such as the maximum numbers of streams and queues, are made relatively large since these tables are allocated in virtual address space rather than wired memory as is the case in a wired kernel implementation.

Parameters that control the number of message buffers are assigned reasonable initial values but are extendible at run time with minimum run time penalty. Depending on the virtual memory size supported by the machine, there are upper limits placed on the buffer usage to preserve the flow control of protocol stacks.

Some parameters, such as the limits on the percentage of data blocks available at each priority level, are common to different protocol stacks running on the same node. The approach taken was to make them constant and known to protocol implementors so that the burden is placed on the implementors to make their protocols work with optimum performance. This is easier than the Unix System V model of making the system administrator choose values for these parameters.

Buffer Space Allocation

The buffer space for messages is allocated from user global space during the initialization of the type manager. The pool of buffers is automatically extended at run time if needed. The message blocks are not necessarily page aligned, and are pageable (not wired into physical memory). Device drivers that need messages in wired memory must do a final copy from stream allocated data blocks to private wired memory.

Older Apollo nodes have a 16 Mbyte virtual address space, of which 3 Mbytes is allocated to user global space. Most of this space is already occupied by the global libraries. On these machines the upper limit on total buffer space is limited to approximately 200 kbytes. Newer nodes all have at least a 64 Mbyte address space with plenty of unused global space.

Kernel Services and Data Structures

One of our primary goals was to be able to port Streams modules from Unix System V to the Domain system with a minimum of changes. Streams modules are written to run in a kernel context, and expect

to be able to use kernel services and data structures. Our user space implementation must then provide at least some of the kernel services these modules expect.

Streams modules in a kernel implementation run in an interrupt context. Since they can not make any assumptions about the context in which they run, they are unable to make any use of most of the data structures in the kernel. Driver and multiplexor open routines, however, do expect to run in the context of the user process doing the open. These routines may want to access kernel data structures that are not normally accessible from user space. We provide user space copies of parts of the u. area and process table for use by driver open routines. These copies contain only those fields that driver open routines might reasonably want to access. The structures are filled in by the Streams open procedure before the driver's open procedure is called.

`Sleep()` and `wakeup()` are implemented in terms of event counts. `Sleep()` can not be called except from driver open and close context, and attempts to do so are flagged as an error. Event counts are a generalized form of `sleep` and `wakeup`. An event count has a monotonically increasing value that can be advanced, or incremented, by a specified amount. A user process can read the current value of the event count, and can arrange to sleep until the event count reaches a given value. With the `sleep` and `wakeup` mechanisms, there is a timing window between testing some condition and sleeping until the condition is satisfied. This window is usually closed by writing a loop to sleep until the condition is satisfied. This results in the condition being tested at least twice. With event counts, the loop can be eliminated by reading the event count, testing the condition, then sleeping until the event count reaches a value greater than the value read.

`Timeout()` is implemented by the module scheduler. Timeout routines are scheduled to run in the same way that module service procedures are scheduled. The Domain user space timer event count only has a resolution of about 1/4 second, as opposed to the 1/60 second clock ticks that are standard in kernel implementations.

`Malloc()` and `mfree()` are implemented in terms of the standard user space storage allocater, using the global storage pool.

Evaluation

We believe that well−written modules should be easy to port to our Streams implementation. Well−written modules would only use those interfaces documented for use by Streams module implementors. Past experience suggests that real life programmers may not strictly adhere to these interfaces. The temptation, when writing code destined for a known implementation such as the System V kernel, is to make use of undocumented features. Programmers should resist this temptation when possible.

Porting device drivers is considerably harder, because of the inherent hardware dependencies involved. We have tried to make the interface from the I/O switch to the driver as standard as possible. Drivers that do not actually talk to hardware, such as pseudo−drivers and multiplexors, should be fairly easy to port.

The module and driver initialization routines are extra pieces that programmers must provide when porting to our Streams implementation. We have traded convenience to the programmer for convenience to the system administrator. Although the programmer must do some extra work to make the modules and drivers self−describing, the administrator only has to copy the object modules into a directory and reboot. We feel that this is a fair tradeoff.

Adding new modules currently requires rebooting, as they must be loaded into global space by the boot loader. There is no fundamental reason why the run time loader can't be made to load these modules into global space after boot time. This would make it possible to add new modules to a running system.

The current method for debugging global routines, such as Streams modules, is awkward. The code is normally mapped read−only, which makes it impossible to set breakpoints. It can be loaded read−write by giving a special flag to the boot program, but this introduces new problems. The breakpoint may be encountered in any process context, and unless it happens to be the context that is being debugged, the node crashes. Unfortunately, modules do not run in any predictable context. One possible fix would be to provide a debugging mode, in which all module processing would occur in the context of the server process.

The user space timer granularity of 1/4 second may be too coarse for some purposes. This does not seem to be a problem for our TCP/IP implementation, which uses the same timer mechanism.

There is currently work underway to implement a full communications protocol using Domain Streams. When this work is done, we will have a better idea of how good the performance is. We think that performance will be at least as good as that of existing protocols on the Domain system. Transmits potentially require no process context switching, as the downstream processing can occur in the context of the process that initiates the write. Upstream processing in some cases can be done in the context of whatever user process happens to be active, since the modules make no assumptions about which context they run in. Measurements on our TCP/IP implementation suggest that the reduction in context switch overhead may produce significant performance improvements. Further work is required to tune the system for optimum performance.

Acknowledgements

In addition to the authors, Larry Allen, Robert Israel, Tom LeMaire, Nat Mishkin and Eric Shienbrood participated in the design phase of this project.

References

[1] Dennis Ritchie, "A Stream Input−Output System," *AT&T Bell Laboratories Technical Journal,* Vol. 63 no. 8 (October 1984).

[2] David Olander, Gilbert McGrath, Robert Israel, "A Framework for Networking in System V," *Usenix Conference Proceedings,* Atlanta, Ga. (June 1986).

[3] Andrew Rifkin, et. al., "RFS Architectural Overview," *Usenix Conference Proceedings,* Atlanta, Ga. (June 1986).

[4] Jim Rees, Paul H. Levine, Nathaniel Mishkin, Paul J. Leach, "An Extensible I/O System," *Usenix Conference Proceedings,* Atlanta, Ga. (June 1986).

[5] David P. Reed and Rajendra K. Kanodia, "Synchronization with Eventcounts and Sequencers," *Communications of the ACM* (February 1979).

[6] *Using the Open System Toolkit to Extend the Streams Facility,* Apollo Computer Inc. (April 1986).

[7] Edsger Dijkstra, "The Structure of the 'THE' Multiprogramming System," *Communications of the ACM* (May 1968).