Multiprocessor Streams for Plan 9

David Leo Presotto

AT&T Bell Laboratories
Murray Hill, New Jersey 07974
research!presotto
presotto@research.att.com

ABSTRACT

This paper describes an implementation of Streams for the Plan 9 kernel, a multi-threaded, multiprocessor kernel with a system call interface reminiscent of UNIX. Rather than port Dennis Ritchie's Streams to Plan 9, we changed the abstraction to fit more naturally into the new environment. The result is a mechanism that has similar performance and is internally easier to program.

1. Introduction

Plan 9 is a new computing environment being built and used by the Computing Science Research Center at AT&T Bell Laboratories. Plan 9 consists of terminals, CPU servers and file servers connected by various networks. These components run specialized operating systems based on a common multi-threaded kernel. The kernel runs on both uniprocessors and shared memory multiprocessors.

Plan 9 communicates via a number of different networks. Therefore we decided to base all our network code on a single structure. This allowed us to solve at once a number of problems such as flow control, memory allocation, and user interface. Given our past experience with it, we chose Dennis Ritchie's Stream I/O System [Rit84] to provide the structure for Plan 9. This coroutine-based design, introduced in the Eighth Edition, provides a clean, flexible mechanism for handling asynchronous I/O. Although Plan 9's kernel is unrelated to that of the Eighth Edition [McI85], the concept of Streams remained directly applicable. We have, however, made two major alterations.

Plan 9 runs on multiprocessor systems so we wanted to exploit their concurrency. In the Plan 9 kernel, the basic unit of concurrency is the process. We therefore converted Ritchie's coroutine-based design to a process-based one. As we shall see later, this change has both advantages and disadvantages.

Associated with the change to a process-based structure, we also had to reduce the number of threads. If we had made the most obvious change to convert each of Ritchie's coroutines into a process, we would have incurred very high CPU penalties. No matter how cheap we make our kernel processes they would never be as cheap as coroutines. Instead, we chose a structure that performs in one process what Streams does in many coroutines.

The result is a structure very similar to Streams but, we believe, easier to program. The interfaces, flow control, and memory allocation are the same. However, the freedom to allow processing modules to block and to use any resources available to a user process has made many pieces much easier to program. A process is a familiar programming construct.

In the rest of this paper we will refer to Ritchie's Streams simply as Streams and to Plan 9 Streams as Plan 9.

2. Data Structures

Plan 9, aside from minor changes, uses the same data structures as Streams. Our description here is very brief and is intended to highlight the differences. We refer the reader to Dennis's excellent BLTJ paper [Rit84] for a more comprehensive treatment.

The appendix contains the C definitions of our data structures.

2.1. Stream

A `Stream` is a full duplex channel connecting a device or pseudo-device to a user process. User processes insert and remove data at one end of the stream. Kernel processes acting on behalf of a device insert data at the other.

A stream is made up of a linear list of *processing modules*. Each module has both an upstream (toward the user) and a downstream (toward the device) *put procedure*. Data is inserted into a stream by calling the put procedure of the module at either end of the stream. Each module calls the succeeding one to send data up or down the stream.

2.2. Queue

An instance of a processing module is described by a pair of `Queues`, one for each direction. Each queue contains:

- a pointer to the put procedure
- a pointer to the next queue
- a linked list of queued data blocks
- the number of bytes queued
- the number of blocks queued
- a spin lock to control access to the data structure

Unlike a Stream queue ours has no *service procedure*. Also, since on a multiprocessor setting priority levels is not a valid synchronization mechanism, we require a spin lock from the operating system to control access to the queue.

2.3. Block

The objects passed through the stream are described by a data structure called a `Block`. Our blocks are identical to Streams and contain a *base pointer*, a *limit pointer*, a *read pointer*, and a *write pointer*. Each pointer refers to memory mapped into kernel space. The base and limit are never changed and are used to describe the data allocated to the block. The read and write pointers point to the start and end of usable data within the block.

There are two block types, *data* and *control*. Data blocks are used to pass information from process to process. Control blocks are used to control the action of the modules. They both have the same format. Data blocks are often queued in a processing module until some condition is met for passing them along or freeing them. Control blocks are never queued but are passed from module to module until one accepts and frees them.

Streams also have data and control blocks. However, their control blocks come in multiple flavors, all queuable; some with the same priority as data and some higher. Higher priority blocks are moved to the front of the queue. At a result, the routines used to manipulate these control blocks tend to be complex.

When a module's *put* is called it is passed a pointer to a block. If one desires to pass many blocks atomically, the blocks may be chained together and a pointer to the first is passed to the procedure. This is similar to the way *mbufs* are passed in the BSD kernel [Lef89]. Streams need no such concept since no two threads run simultaneously in a Streams procedure. UNIX System V STREAMS [Bac86] have a much more complicated construct to pass a multi-block message along with a single put. The System V construct is used both for atomicity (they originally had no other block delimiters) and for performance.

3. Algorithms

3.1. Memory Allocation

Stream memory is allocated at system start time. A list is kept for each of several fixed block sizes. A process that requests a size receives the smallest block that can hold the request. Synchronizing access of the free lists is performed using a spin lock per free list.

Since all stream code runs in the context of processes, whenever an allocation cannot be immediately processed the caller blocks until a block of the right size is freed. The result is that momentary surges in used blocks do not panic the kernel as they sometimes did in the Eight Edition.

The number of lists, the specific block sizes and the number allocated of each size depends on the kernel. Terminals tend to use more small blocks, the servers more large ones. The allocated blocks reflect this.

3.2. User Interface

A stream is represented at user level as a directory containing at least two files, `ctl` and `data`. The first process to open either file creates the stream automatically. The last process to close destroys the stream. Writing to the `data` file causes a switch to kernel mode. The process then copies the data into kernel data blocks and calls the put procedure of the first downstream processing module for each block. The last block of a write is flagged with a delimiter in the event that the downstream module cares about write boundaries. In most cases the first put procedure calls the second, the second calls the third, and so on until the data is output. Thus, data may often be sent without taking a context switch. A write lock at the top of the stream assures that no two processes can simultaneously insert data into the top of the same stream, which insures that all writes are atomic.

Our system has no `ioctl` system call. The syntax and semantics of `ioctl` in UNIX are so uncontrolled that we left it out. Writing to the `ctl` file takes the place of `ioctl`. Writing to the control file is the same as writing to a data file except that the blocks created are of type control. A processing module parses each control block put to it. The commands in the control blocks are simple ASCII strings. Therefore, there is no problem with byte ordering when one system is controlling streams in a name space implemented on another processor. The time to parse the control blocks is not important since the control operation is a rare one, usually used only when starting operation on a stream.

The stream system intercepts the control blocks that control configuration of the stream. These control blocks are:

push *name* — to add an instance of the processing module *name* to the top of the stream.

pop — to remove the top module of the stream.

hangup — to send a control message containing the string ‘‘hangup’’ up the stream from the device end.

Other control blocks are read by each module they pass through.

Reading from the `data` file returns data queued at the top of the stream. The read terminates either when the read count is reached or when the end of a delimited block is read. There is a per stream read lock that ensures that only one process can read from the stream at a time. This ensures that the bytes read were contiguous bytes from the stream. Reading the `ctl` file is described in the section on multiplexing.

3.3. Device Input

When input exists at a device, the driver’s interrupt routine wakes up a kernel process to carry the data upstream. A kernel process is an ordinary process with no user level segments allocated to it and is scheduled just like any other process. Message-based devices like Ethernet [Met80] may have many processes ready to carry the next message upstream so that many messages can be processed simultaneously.

The kernel process carries the message upstream through protocol modules. Eventually, the message is delivered to the most upstream queue. The kernel process leaves it there and wakes any user process blocked in a read on that stream. Thus, the only difference between input and output is that the user process must perform the copy of the data from kernel blocks to user space. This can be a benefit in our multiprocessors in which each processor has a separate cache. If the kernel process were to copy the data into the user process it would most likely do it on the wrong processor and hence into the wrong cache. This would force the user process to fault it into its own cache, increasing the load on the shared memory bus.

3.4. Multiplexing

Most protocols need to multiplex several conversations onto a single physical device. This is added to our scheme using pseudo-devices, one for each multiplexed conversation. This is very similar to the way Eighth Edition handles TCP/IP. A group of pseudo-devices are coupled with a multiplexing processing module that is pushed onto a physical device stream. The device end modules on the pseudo-devices add the necessary header onto downstream messages and then put them to the module downstream of the multiplexor. The multiplexing module looks at each message moving up its stream and puts it to the correct pseudo-device stream after stripping whatever header it used to do the demultiplexing.

The user interface to a multiplexed protocol is a directory. The directory contains a `clone` file and a stream directory for each conversation. The stream directories are numbered *1* to *n* (see Figure 1). Opening the clone file is a macro for finding a free stream directory and opening its `ctl` file. Reading the control file returns the ASCII number of the conversation chosen. This allows the user process to find the corresponding `data` file.

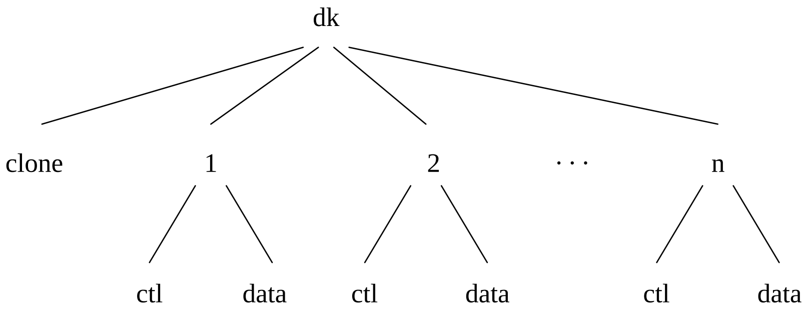

Figure 1

3.5. Pipes

Pipes, as in Eighth Edition, are just two streams joined at the device end. The `pipe` system call returns file descriptors for the data files of the two streams. The control files are inaccessible.

3.6. Helper Processes

Transport protocols need to retransmit lost data. However, to achieve true pipelining, the user process will want to queue data at the protocol module and return. Another process has to retransmit the data when needed since all put procedures must be called in the context of a process. For this purpose, processing modules can create kernel processes to perform such actions when needed. The processes are awakened on need by the processing module’s put procedure or whenever a timer expires.

3.7. Flow Control

In any system that queues data one needs a mechanism to keep a queue with a slow reader from absorbing all of memory. We use a flow control mechanism similar to Streams. Each queue keeps a count of bytes and a count of blocks queued there. Whenever either exceeds a predetermined limit, the high water flag is set for that queue. Each caller of a processing module checks the high water flag for the next queue before calling its put procedure. If the next queue is past its high water mark, the would-be caller goes to sleep, leaving a pointer to itself in its queue (the rendezvous structure in `Queue`). When a process empties a queue past half its high water mark it wakes up any process waiting at the previous queue.

Modules implementing transport protocols with window schemes implement flow control a little differently. Rather than go to sleep, they always pass data upstream. However, when the upstream queue is full, the transport module stops sending acknowledgements back to the remote system. Hence, the remote side will eventually stop sending. To open the window again, a helper process sleeps in the queue instead of the device process. When the next queue empties, the helper is awakened and it does whatever is needed to open the remote system's transmit window.

4. Performance

A different flavor of Streams cannot be evaluated without comparing it against earlier ones. Our results give a general idea of where the advantages and disadvantages of Plan 9 Streams lie. However, the systems compared are in many ways incomparable. The compilers, operating systems, and Streams code all have a considerable effect on the results.

For Plan 9 numbers we use 4 different configurations. Three are SGI Power Series machines with 1, 2, and 4 processors. We compare these against another 4 processor SGI Power Series running System V Release 3 and against a single processor MIPS M2000 system also running SVR3. The M2000 has the same CPU running at the same speed as the SGI machines. However, it has a considerably faster memory system. Outside of tight loops, this has a major impact on processing speed.

The other Plan 9 system is the Gnot terminal [Pik90]. This is a system developed in our center and now manufactured for us by AT&T. It uses a 25 MHz Motorola 68020. We compare it against a DEC MicroVAX 3. The machines on average are about the same speed. The Gnot CPU is about 4/3 the speed of the microVAX with a memory system that is about 2/3 the speed of the microVAX.

All tests measure both throughput and latency. The throughput is tested using the following program:

```
int i;
char buf[64*1024];
int p[2];

makeconnection(p);
switch(fork()){
case 0:
        close(p[1]);
        while(read(p[0], buf, sizeof buf) > 0)
                ;
        break;
default:
        close(p[0]);
        for(i = 0; i<ITER; i++){
                if(write(p[1], buf, sizeof buf) != sizeof buf){
                        perror("write");
                        exit(1);
                }
        }
        break;
}
```

The block size is chosen to be large to minimize the difference in system call speeds. The latency is tested using:

```
int p[2];
int i;
char c;

makeconnection(p);
switch(fork()){
case 0:
        close(p[1]);
        while(read(p[0], &c, 1) == 1)
                write(p[0], &c, 1);
        break;
default:
        close(p[0]);
        for(i = 0; i<ITER; i++){
                if(write(p[1], &c, 1) != 1){
                        perror("write");
                        exit(1);
                }
                if(read(p[1], &c, 1) != 1){
                        perror("read");
                        exit(1);
                }
        }
        break;
}
```

In both cases, we perform each operation for a large number of iterations to get an average time.

The first test (Table 1) compares Plan 9 pipes against SVR3 normal pipes, SVR3 stream pipes, a BSD sockets implementation running under SVR3, and Tenth Edition Streams. Since only two processes are involved in all configurations and no processing is being performed by processing modules, we are comparing the speed of the basic plumbing in the systems.

Table 1 - Pipes		
system	throughput MBytes/sec	latency ms
SGI/1 CPU Plan 9	6.0	.29
SGI/2 CPUs Plan 9	8.4	.21
SGI/4 CPUs Plan 9	8.4	.28
SGI/4 CPUs sVr3 old pipes	4.5	.51
M2000 sVr3 stream	8.0	.51
M2000 sVr3 sockets	8.0	.36
68020 Gnot Plan 9	1.79	1.67
uVAX 3 10th Edition spipe	1.04	1.69

From Table 1 we can see that for the large machines, Plan 9 has lower latency. This was the expected result since the straight call structure requires many fewer instructions than traditional Streams which must schedule each service procedure in addition to calling its put procedure. The dip of .08 ms when going from 1 processor to 2 is the result of concurrency. The second process is starting up before the first has returned from queuing its block.

The unexpected result is the rise from .21 ms to .28 when going from 2 processors to 4. We believe that this is contention over the process queue. We hope to verify this assumption before this paper is presented.

The low single processor SGI throughput for Plan 9 compared to the M2000 reflects the slower memory on the SGI box. When we use multiple processors, we take advantage of the concurrency and our throughput passes all the others.

Plan 9 on the Gnot compared to the Tenth Edition on the MicroVAX is less impressive. We still have a definite advantage in throughput. However, given the ratio of machine speeds, we should be much better in latency. Profiling the kernel showed that the disappointing latency time was due entirely to the MMU. The MMU on the Gnot only retains one process state. Whenever we switch context we do a lot of faulting to refill the MMU. The reason the throughput doesn't suffer from this problem is that the pipelining causes a lot of data to be moved per context switch.

To compare the performance of kernel driver processors to performing puts at interrupt level, we used our most prevalent network, Datakit [Che80]. It is both a local and wide area network spanning all of AT&T. The MicroVAX, SGI, and M2000 each have 8 megabit/sec links to Datakit. However, due to constraints in the Datakit the highest throughput is 2.6 megabits. The Gnots have a slower 2 megabit link [Pre88]. Table 2 presents performance of various systems through the Datakit. In all cases the remote system is a Plan 9 SGI processor. Once again, Plan 9 throughput matches or exceeds the throughput of the other systems. However, latency is worse. This is the price paid for using kernel processes to send device data upstream rather than doing it in the interrupt routine. The degradation is especially evident in the Gnot since it is the worst at process switching.

Table 2 - URP/Datakit

system	throughput KBytes/sec	latency ms
SGI/2 CPUs Plan 9	235	1.4
SGI/4 CPUs Plan 9	235	1.4
M2000 sVr3	235	1.2
68020 Gnot Plan 9	100	5.8
uVAX 3 10th Edition	85	3.2

Finally, we present some Ethernet performance results. We don't compare these against other systems since the protocol we use, Nonet, currently runs only on Plan 9. It was designed as a low weight transport protocol. It should be noted that the throughput figures are higher than any we've seen published to date for an Ethernet. This is as much a function the protocol as it is of Plan 9.

Table 3 - Nonet/Ethernet		
system	throughput KBytes/sec	latency ms
SGI/2 CPUs Plan 9	950	1.4
SGI/4 CPUs Plan 9	950	1.4

Related Work

We must mention Larry Peterson's *x*-Kernel work [Pet89]. The *x*-Kernel is very similar to Plan 9 Streams. It is used primarily to study the decomposition of network protocols. The process structure, multiplexing, and data structures are virtually identical to Plan 9 Steams.

The greatest differences between our systems are:

1) The *x*-Kernel uses very light weight kernel processes. They are just a PC and a stack. Our kernel processes carry all the baggage of user processes.

2) Rather than queue data at the user process and wait for the user process to read it, *x*-Kernel kernel processes call a put procedure in the user's address space which moves the data directly into user memory.

3) The message in the *x*-Kernel is in the form of a tree of blocks. A pointer to the top of the tree is passed through the processing modules. We use a linear structure.

Published performance of the *x*-Kernel is similar to ours with lower latency times. We hope to borrow some of the ideas of the *x*-Kernel to improve our own performance.

Conclusions

We have presented another variation of streams. The main advantage to Plan 9 Streams is making use of concurrency in multiprocessors. We have a very subjective belief that the process based model is easier to program than the coroutine based one.

The performance results show that the Plan 9 model has high throughput. However, the contexts switches caused by the kernel processes increase latency. Further research and tuning is required to reduce these costs.

5. References

Bac86. M. Bach, *The Design of the UNIX Operating System,* Prentice-Hall (1986).

Che80. G. L. Chesson and A. G. Fraser, ''Datakit Network Architecture,'' in *IEEE Compcon '80* (January, 1980).

Lef89. S. Leffler, M. K. McKusick, M. Karels, and J. Quarterman, *4.3BSD UNIX Operating System,* Addison Wesley (1989).

McI85. M. D. McIlroy, *Unix Programmer's Manual, Eighth Edition,* Bell Laboratories, Murray Hill, NJ, USA (February, 1985).

Met80. R. Metcalfe, D. Boggs, C. Crane, E. Taft, J. Shoch, and J. Hupp, ''The Ethernet Local Network: Three Reports,'' CSL-80-2, XEROX Palo Alto Research Centers (February, 1980).

Pet89. L. Peterson, ''RPC in the X-Kernel: Evaluating New Design Techniques,'' in *Proceedings Twelfth Symposium on Operating Systems Principles*, Litchfield Park, AZ (December, 1989).

Pik90. R. Pike, D. Presotto, K. Thompson, and H. Trickey, ''Plan 9 from Bell Labs,'' in *UKUUG Proceedings of the Summer 1990 Conference,* London, England (July, 1990).

Pre88. D. Presotto, ''Plan 9 from Bell Labs - The Network,'' in *EUUG Proceedings of the Spring 1988*

Conference, London, England (April, 1988).

Rit84. D. M. Ritchie, ‘‘A Stream Input-Output System,’’ *AT&T Bell Laboratories Technical Journal* **63**(8) (October, 1984).

UNIX is a registered trademark of AT&T Bell Laboratories

Datakit is a registered trademark of AT&T

Appendix

```
/*
 *  operations available to a queue
 */
typedef struct Qinfo    Qinfo;
struct Qinfo
{
        void (*iput)(Queue*, Block*);   /* input routine */
        void (*oput)(Queue*, Block*);   /* output routine */
        void (*open)(Queue*, Stream*);
        void (*close)(Queue*);
        char *name;
};

/*
 *  We reference kernel memory via descriptors kept in host memory
 */
typedef struct Block    Block;
struct Block
{
        Block   *next;
        uchar   *rptr;          /* first not consumed byte */
        uchar   *wptr;          /* first empty byte */
        uchar   *lim;           /* 1 past the end of the buffer */
        uchar   *base;          /* start of the buffer */
        uchar   flags;
        uchar   type;
};

/* flag bits */
#define S_DELIM 0x80    /* this block is the end of a higher level message */
#define S_CLASS 0x07

/* type values */
#define M_DATA 0
#define M_CTL 1

/*
 *  a list of blocks
 */
typedef struct Blist    Blist;
struct Blist {
        Lock;
        Block   *first;         /* first data block */
        Block   *last;          /* last data block */
        long    len;            /* length of list in bytes */
};
```

```
/*
 *  a queue of blocks
 */
typedef struct Queue    Queue;
struct Queue {
        Blist;
        int     nb;             /* number of blocks in queue */
        int     flag;
        Qinfo   *info;          /* line discipline definition */
        Queue   *other;         /* opposite direction, same line discipline */
        Queue   *next;          /* next queue in the stream */
        void    (*put)(Queue*, Block*);
        Rendez  r;              /* flow control rendezvous point */
        void    *ptr;           /* private info for the queue */
};
#define QHUNGUP 0x1     /* flag bit meaning the stream has been hung up */
#define QINUSE  0x2
#define QHIWAT  0x4     /* queue has gone past the high water mark */

/*
 *  a stream head
 */
struct Stream {
        Lock;                   /* structure lock */
        int     inuse;          /* use count */
        int     hread;          /* number of reads after hangup */
        int     type;           /* correlation with Chan */
        int     dev;            /* ... */
        int     id;             /* ... */
        QLock   rdlock;         /* read lock */
        QLock   wrlock;         /* write lock */
        Queue   *procq;         /* write queue at process end */
        Queue   *devq;          /* read queue at device end */
};
#define RD(q)           ((q)->other < (q) ? (q->other) : q)
#define WR(q)           ((q)->other > (q) ? (q->other) : q)
#define PUTNEXT(q,b)    (*(q)->next->put)((q)->next, b)
#define BLEN(b)         ((b)->wptr - (b)->rptr)
#define QFULL(q)        ((q)->flag & QHIWAT)
#define FLOWCTL(q)      { if(QFULL(q)) flowctl(q); }
```