stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 203/82 AUGUSTUS

D.S.H. ROSENTHAL

A SURVEY OF ASYNCHRONOUS I/O TECHNIQUES FOR UNIX

Preprint

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

CR Categories and Subject Descriptors: D.4.4 [Operating Systems]: Communications Management — Network communications; Terminal management

General Terms: Design

A Survey of Asynchronous I/O Techniques for UNIX

by

David S. H. Rosenthal

ABSTRACT

The UNIX operating system's model of I/O is fundamentally *synchronous*, that is, conceptually each process' computation and I/O are not overlapped. Although for the great bulk of applications this model is entirely adequate, certain applications find it irksome. The various techniques developed to overcome or avoid this restriction are described and contrasted, using example applications to illustrate the advantages and problems of each.

This paper has been submitted for publication.

1. Introduction

"The parson came to school and told us *not* to do it. The doctor came to school and told us *how* not to do it. Then the headmaster told us *where* not to do it."
A schoolboy on sex education.

In the original paper on UNIX,* Ritchie and Thomson set out the way in which a process does I/O:

"To the user, both reading and writing of files appear to be synchronous and unbuffered. That is, immediately after return from a **read** call the data are available; conversely, after a **write** the user's workspace may be reused."[10]

This means that, at least conceptually, computation and I/O are not overlapped; a process is either computing or it is doing I/O. To enforce this, if input is not available from a file when a process requests it, the process will sleep until input is available.

An attractive feature of UNIX is that it treats *all* I/O as file I/O; this allows simple programs to be very versatile, since devices, terminals and interprocess communication channels are all accessed as files. However, if a file represents a terminal or a pipe† a program asleep waiting for input on it may sleep indefinitely. Further, a process must address its I/O request to a single file, and during the sleep will ignore activity on other files.

For certain applications, generally those driving terminals or networks, indefinite sleeps and single-file I/O are unacceptable. Over the years a number of efforts have been made by the UNIX community to subvert the fundamentally synchronous I/O model underlying their system. These efforts have been applied to different versions of the system; this survey attempts to place them in a uniform context, but this does not represent the facilities of any particular version.

2. File I/O Model

The UNIX system calls implementing the file I/O model are relatively simple. They depend upon the concept of a *file descriptor*, which is a small integer used by a process as its local name for the global file object. A process obtains a descriptor for a file as a result of a successful

```
fd = open(name,flag);
```

invocation. The **flag** parameter specifies whether the file is to be read, written, or both. The **name** must have been the subject of a successful

```
fd = creat(name,mode);
```

or

```
success = mknod(name,mode,dev);
```

call, according as it is a normal file, or a directory or special file.† The **mode** parameter specifies the protection bits, which determine the operations the owner, other members of the group, and the general public can perform.

Once a file descriptor is available, it may be used to perform I/O on the file, using either

```
n = read(fd,buffer,count);
```

or

* UNIX is a Trademark of Bell Laboratories.
† An interprocess communication channel, which behaves as an anonymous FIFO file.
† A file name representing a physical I/O device.

```
n = write(fd,buffer,count);
```

In each case, an attempt will be made to transfer **count** bytes to/from the process' address **buffer**. The returned value **n** is the number of bytes actually transferred. On a **write**, it will be **count** except if an I/O error or end-of-medium has occurred. On a **read**, it will be less than **count** if the file pointer was too close to the end of the file, and it will be zero if the end of the file had been reached.

A file pointer is maintained for each open file. It is affected by **read** and **write** calls, so that by default I/O is sequential, but may also be altered by a

```
where = lseek(fd,offset,base);
```

invocation. The **offset** parameter specifies a place, relative to the beginning of the file, the file pointer, or the end of the file, depending on **base**. The value returned is the final position of the pointer. This call has no effect on certain kinds of files, such as terminals and pipes.

The kernel also maintains certain other information about files. Some is maintained by the "device-independent" part of the kernel and exists for all files, for example an "exclusive use" bit, forbidding futher **opens**. Other information is maintained by individual device drivers, and applies only to special files serviced by the driver, for example the echo switch, special character assignments, etc. for terminals. This information can be accessed by

```
success = ioctl(fd,funct,argp);
```

calls. The **funct** parameter specifies the particular function to be performed, and the **argp** parameter points to a structure containing input or output data for the function. For example, the following sequence sets echo on for the terminal whose descriptor is **fd**.

```
{       struct sgttyb       tty;
        ioctl(fd,TIOCGETP,&tty);
        tty.sg_flags |= ECHO;
        ioctl(fd,TIOCSETP,&tty); }
```

An interprocess communication channel is obtained by a process invoking

```
success = pipe(fdp);
```

where the **fdp** parameter points to a two-element array of file descriptors, one of which becomes a descriptor for the write end of the pipe, and the other becomes a descriptor for the read end. The data flow is thus *uni-directional*, from the write end to the read end. However, because the same process may have descriptors for both ends, this is not quite the absolute restriction it seems. A single pipe may be used to transfer in both directions between a pair of processes if both have descriptors for both ends.

Although this pair of file descriptors is obtained by a *single* process, it can be used for interprocess communication because processes inherit descriptors for their parent's open files. Note that, although a pipe behaves as a file, there is no name for it in the file system's directory structure, so it cannot be the subject of an **open** call. It can only be used by the process creating it, and its descendants.

3. Polling

The simplest approach to asynchronous I/O is for the program to loop, *polling* the files to see if they have input available, or will accept output. On the simple file I/O model, this is impossible, since if a file has no input it will cause the process to block indefinitely. Three different techniques have been developed to overcome this problem.

3.1. Abort the Blocked Calls

Consider a single-player game, such as Michael Toy's *worm*, in which a worm crawls continually around the VDU screen, eating randomly arranged titbits and growing correspondingly longer. The player may change direction by hitting the arrow keys. The program must run itself at regular intervals to update the worm's image, but must also read the terminal on each cycle to discover if the player has changed direction. The conventional **read** call, which sleeps until input is available, would constrain the worm to move only when the player hit a key.

Worm, and similar games, are examples of Seventh Edition UNIX's standard form of asynchronous I/O. They invoke

alarm(sec)

before issuing the **read** call. This causes the *signal* SIGALRM to be received after **sec** seconds. If the **read** has not returned by that time, it will be aborted with zero bytes read, and an error indication will be returned.

The concept of *signals* permits processes to respond to conditions arising asynchronously. A process may invoke

oldfp = signal(sig,newfp);

to announce that, if signal **sig** is received, the function pointed to by **newfp** should be invoked. The value returned is the previous function pointer for this signal. An asynchronous event, such as a floating point exception, or another process sending a signal, causes the system to interrupt the catching process, create an appropriate stack frame, and re-start the catching process at **newfp**. The effect is that the function is called asynchronously when the corresponding signal occurs.

Thus *worm* loops, calling first **alarm(1)** and then **read(tty,&c,1)**. If the read returns some bytes, then a command from the player must be processed. Even if no bytes are returned (because the signal occurred), the worm must move on. This approach is adequate for single-file applications not requiring fast response. If fast response is essential, the one-second resolution of the **alarm** call is inadequate. If more than one file must be polled, spending one second on each becomes excessive.

3.2. Avoid Issuing Calls Which Block

An example of an application needing to poll multiple files would be a multi-player version of the worm game. In this version, a player joining would be given a worm of his own to control, and would see all the worms chasing each other about the screen to get the food. The process implementing the game would need on each cycle to read each of the terminals in case any player had issued a command. The **alarm** technique would be far too slow.

The solution developed to support this type of program is an **ioctl** that informs the calling process whether a **read** to the file descriptor would block or not. Thus, the multi-worm game could poll each file with **ioctl**, and only issue the **reads** which would not block. In this way, it could cycle as fast as it liked, though a considerate game would sleep once per cycle to accommodate other uses of the computer.

3.3. Ensure That The Calls Don't Block

An alternative to the "will-I-block" enquiry has also been developed. This provides a file mode, settable by **ioctl**, instructing the system that I/O to this file is never to block. A **read** to an empty file in non-blocking mode returns immediately with zero bytes read.* A **write** to a congested pipe would return with fewer bytes written than requested.

An example of the use of this mode is a network protocol handler, enabling a UNIX system to appear as a host on a network. The connection to the network appears as a single file, carrying several conversations multiplexed together. Figure 1 shows a protocol process reading this file, demultiplexing the conversations, and writing each line to a pipe leading to an appropriate shell.† The output from each shell is received along another pipe, multiplexed together with the outputs of the other shells, and written to the network file.

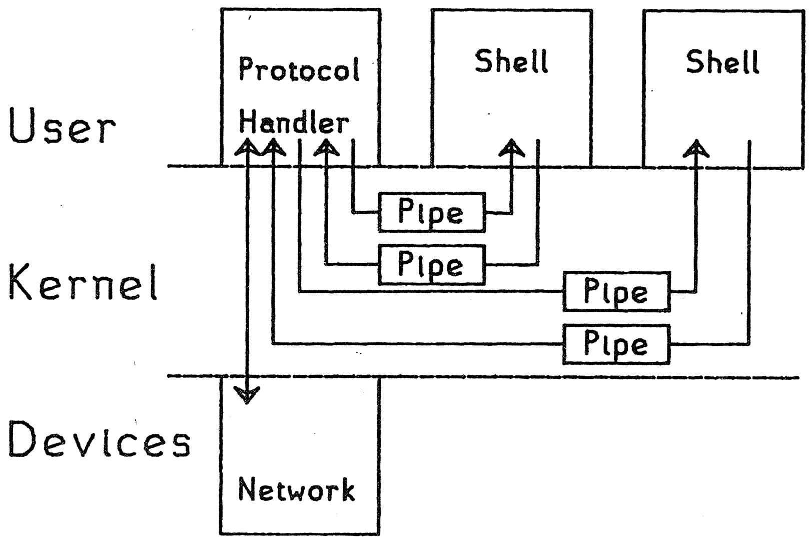

Figure 1 — Network Protocol Handler

All the pipes, and the network file, would have non-blocking mode set. The process would cycle, invoking **write** on each output file, and **read** on each input file. For each output file, the process would maintain a queue of bytes to be sent, shortening it by the number of bytes actually written. If a **read** returned some bytes, they would be (de)multiplexed and added to the appropriate queue.

Non-blocking mode has been most useful when combined with another development. *Named pipes*, or *fifos*, are files which behave as pipes, but which *do* have names in the directory hierarchy, and thus may communicate between un-related processes. If, in addition to creating anonymous pipes to communicate with the shells, the protocol handler created some named pipes, then other processes could use them to talk out to the network. (see Figure 2).

There are two problems with this approach. First, terminals normally wish to be able to transmit "end-of-file" to their shell and its descendants. Because of the double meaning of zero bytes read, this is no longer possible. Secondly, the shell and its descendants can no longer use **ioctl** to enquire or set the parameters of the terminal; there is no way to transmit the information through a pipe.

* Unfortunately, this sets up a double meaning for the zero bytes read, both "end-of-file" and "no input yet".

† Command interpreter.

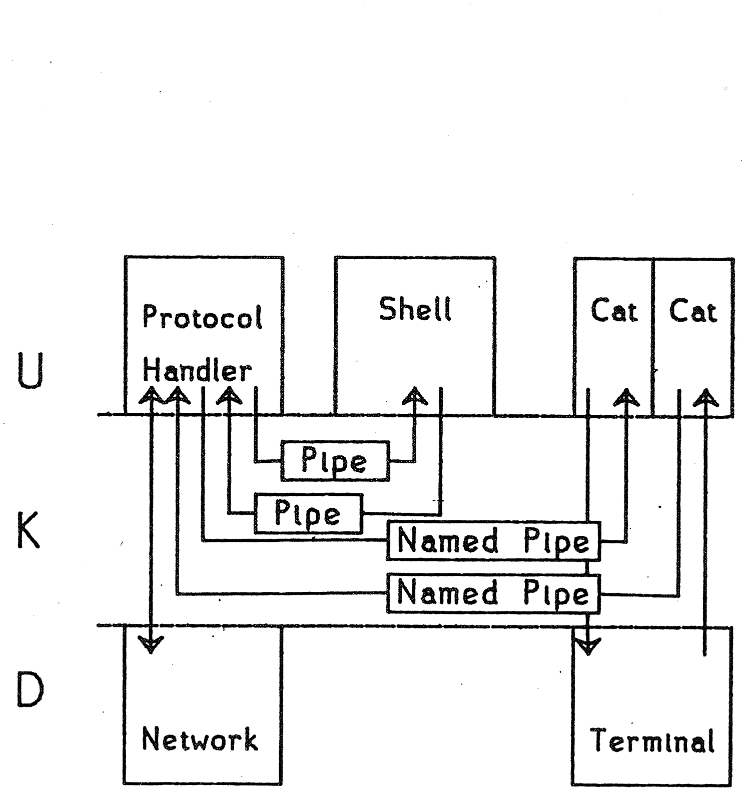

Figure 2 — Bi-directional Network Access

4. Multiplex Files

Like a pipe, a multiplexer is an interprocess communication facility. Like a named pipe, it may appear in the directory hierarchy, and be used to communicate between unrelated processes. All pipes, named or un-named, have only two ends, the read end and the write end. By contrast, a multiplexer has many ends, one of which is special. The special end is called the "multiplex file", and the other ends are called "channel files". Unlike a pipe, all these ends support bi-directional data flow.

A channel file appears to any process having it open to be a terminal, in that **ioctl** can be applied to it. Data is transferred into and out of a channel file by the normal read and write calls; it appears in every respect to be a normal file.

A multiplex file is *not* a normal file, in that the read and write calls are interpreted differently. Unlike a normal call, in which all **count** bytes pointed to by **buffer** are transferred, multiplex files interpret the bytes at **buffer** as a *header* structure and, optionally, some *data* to be transmitted. The header specifies the channel from/to which the data is being transferred. Thus, if the header specifies channel X, a **read** call is interpreted as "someone wrote these bytes into channel X", and a **write** call as "send these bytes to channel X".

Various *messages* may also be read from a multiplex file, describing the activities on the channels. They say things like "someone invoked **ioctl** on channel X" or "no-one has channel X open any longer". The details of these messages, the control functions used to reply to them, and the structures interpreted by the I/O calls, depend upon the particular implementation.

Figure 3 shows a typical rôle for a multiplexer, supporting a network protocol handler of the type described above. Note that in this case, the protocol handler need only poll two files; all its communication with other processes occurs via a single file descriptor.

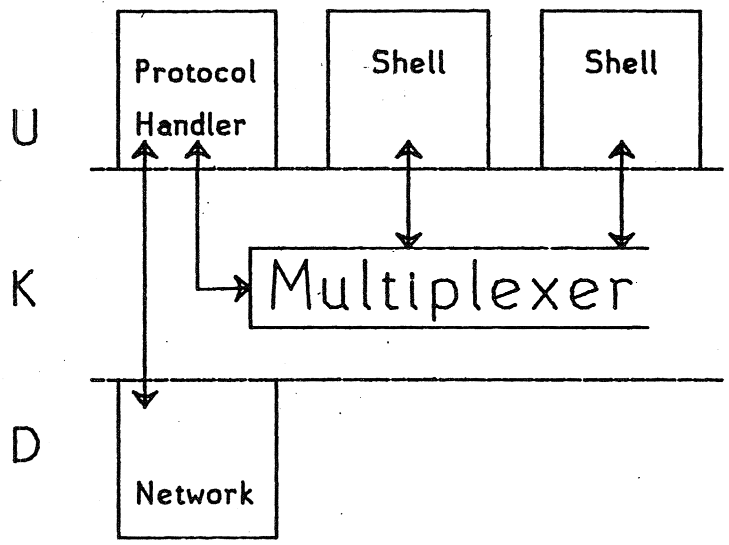

Figure 3 — Multiplexer for Network Access

4.1. Bell Labs' Multiplexer

A multiplexer is a standard part of Seventh Edition UNIX, albeit accompanied by a warning that it is "an experimental part of the operating system more subject to change and prone to bugs than other parts." It provides a full range of facilities, including dynamic creation of new channels.

The **read** call returns data formed into a structure defined by[*]

```
typedef struct {
        short    index;    /* Channel No. */
        short    count;    /* Data size, 0 = control */
        short    ccount;   /* Control size */
        char     data[];   /* Data, if any */
        } input_rec;
```

and the **write** call interprets an array of structures defined by

```
typedef struct {
        short    index;
        short    count;
        short    ccount;
        char     *data;    /* Pointer to data */
        } output_rec;
```

Note that in both cases, a single I/O call may result in the transfer of several messages. The control messages are shown in Table 1.

Among the special control functions, one with particular importance for networking is

* C does not permit variable size fields; these definitions are not to be taken too literally.

Table 1 - Bell Labs' Mpx Messages

Name	Meaning
M_WATCH	A process wants to open channel X
M_EOT	End of File on channel X
M_CLOSE	Channel X no longer open
M_BLK	Channel X congested
M_UBLK	Channel X no longer congested
M_IOCTL	A process invoked ioctl on channel X

connect(fd,cd,end)

which arranges for the terminal file described by **fd** and the channel described by **cd** to be spliced together in such a way that data flows between them without the intervention of any user-level process.

This facility permits a further simplification of the network protocol handler example. If the network connection file descriptor is **connected** to a channel of the multiplexer, *all* the protocol handler's I/O takes place via a single file descriptor (for the multiplex file). This finally eliminates all polling, the handler can sleep on **reads** on the multiplex file's descriptor. Terminal files can be **connected** to multiplexer channels to permit them to talk out to the network. (see Figure 4).

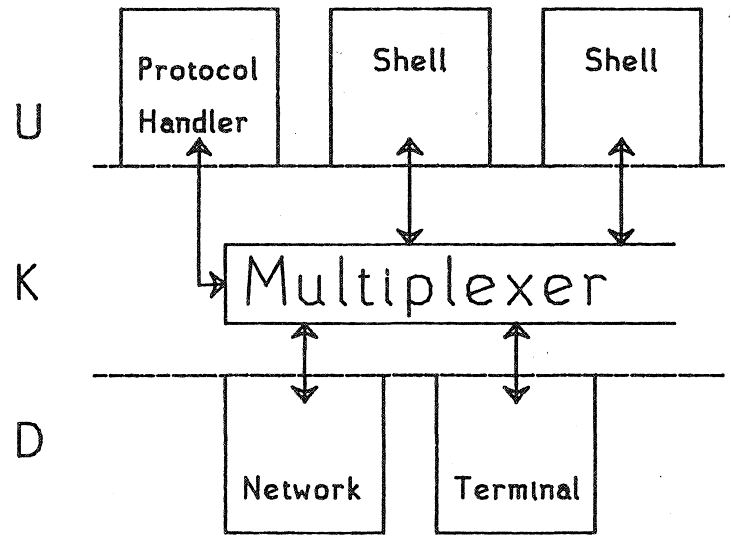

Figure 4 — Using **connect** for Network Access

4.2. Simpler Multiplexers

Bell Labs' multiplexer provides the facilities required to support network access, but it is too large to fit comfortably on the smaller PDP11s making up a large proportion of the UNIX community. As a result, several simpler multiplexers have been implemented.

4.2.1. Sydney's Multiplexer

One such simplified multiplexer was developed as part of the rapidly expanding UNIX network in Australia, which is implemented using asynchronous links and terminal-like I/O[8, 6]. As compared with the Bell Labs multiplexer, the following restrictions are imposed:

- The number of channels is static, determined at system generation time.
- The multiplex file is *not* visible to user processes, it is permanently connected to a character special file linking the system to another UNIX.
- Although each of the channels appears to be a terminal, **ioctl** calls are not transmitted, but are handled locally by the channel.

A **connect** facility is provided by a separate system call, applicable to all terminals, not just channel files. Except for this facility, this multiplexer is conceptually similar to that implemented at the University of Illinois to support the ARPANET NIP[2, 1]. Although the implementations are very different, the restrictions are the same. The way in which this multiplexer is used is shown in Figure 5.

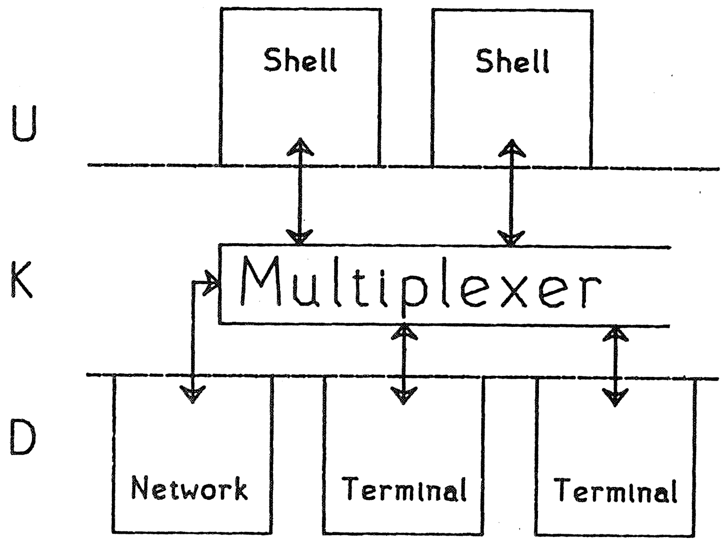

Figure 5 — The Sydney Multiplexer in Use

4.2.2. Edinburgh's Multiplexer

An alternative set of restrictions is imposed upon the multiplexer developed at Edinburgh to support RCONET access:

- The number of channels is static, determined at system generation time.
- The ability to **connect** files is completely eliminated.
- At the multiplex file end, only one message is transferred per I/O call.
- Instead of distinguishing between control and data messages using the header, all control messages are transferred on channel 0.

Synchronisation of messages and data is achieved by draining the channel before sending the message, and blocking the channel until the message has been received. These restrictions permitted the following very simple format:

```
typedef struct {
        char    chan;    /* Channel No. */
        char    data[];  /* Data, if any */
        } io_rec;
```

The length of the data need not be specified in the header; it is known from the number of bytes transferred. The set of messages is shown in Table 2.

Table 2 - Edinburgh Mpx Messages

Name	Meaning
MX_OPEN	A process wants to open channel X
MX_CLOSE	Channel X no longer open
MX_MORE	Channel X no longer congested
MX_IOCTL	A process invoked ioctl on channel X
MX_READ	A process is sleeping on channel X

As compared with the Bell Labs messages, the effect of M_EOT is obtained by zero-length transfers, the effect of M_BLK is obtained by setting non-blocking mode, when the system will truncate writes to congested channels, and MX_READ is needed by our line-oriented virtual terminal protocols to ensure prompts are flushed.

Unlike other multiplexers, particularly those supporting some form of **connect**, this multiplexer is a normal UNIX device driver, requiring no hooks in code elsewhere in the kernel. It provides the functions necessary to support network access, but the lack of a **connect** facility makes such access inefficient, since all data transferred to or from the network must be processed by at least one user-level process. The use of this multiplexer was shown in Figure 3.

5. Message Passing

The UNIX kernel may be viewed as providing a set of services to processes, requested via the system calls. In a conventional implementation, system calls are handled by the process itself; they cause it to change to a privileged state in which it runs kernel code and accesses global data, eventually reverting to the normal state as the call returns.

An alternative view, more adapted to asynchronous I/O, is that the services are provided by separate *server* processes, running privileged code on behalf of *all* processes. A system call causes an interprocess message to be sent to the appropriate server. The invoking process may then suspend itself awaiting a reply message.

5.1. The Carnegie-Mellon System

At Carnegie-Mellon University, Rashid's goal was to implement asynchronous I/O for UNIX, and other systems[9]. The result was a highly sophisticated message-based interprocess communication facility, applicable to many systems, but fully integrated with the conventional UNIX environment.

5.1.1. General IPC Facility

The intention was to specify a general IPC facility independent of a particular language, a particular operating system, or a particular machine, so that it could be used to provide uniform access to a network supporting different machines running different operating systems.

The basic concept of the IPC system is that of a *port*, a FIFO queue of limited length containing *messages*. A port is a global object, like a file, for which processes are given local names by the kernel. A process may create a port and obtain a "port descriptor" for it by

pd = AllocatePort(Qlength);

Processes may have three different types of access rights to a port, *ownership* of a port, *receive* access, and *send* access.

Initially, the creator has ownership and receive access to the new port, but it may pass (via messages) these rights to other processes. Only one process may own a port, and only one process may have receive access to a port, but they need not be the same process. The owner may attach a file name to the port, by invoking

success = AssertName(name,port,mode);

Any number of processes may have send access to a port, obtained by inheriting a descriptor, or by invoking

pd = Locate(name);

with an asserted name. These restrictions mean that a port implements *uni-directional* data flow.

The facility provides *flow control* for messages flowing through ports. The owner of a port may set its maximum queue length. If a process attempts to write to a congested port, it may elect to:

- Wait until the port is no longer congested.
- Receive an immediate error indication.
- Have the message accepted into a "pending" state, from which the kernel will eventually move it onto the queue. When this happens the sending process will receive a message. At most one message per sender per port may be pending.

The last option is important for server processes, who do not wish to sleep until their clients process messages, nor to have to poll their output ports to see if they are still congested.

The messages passed by the IPC facility are collections of *typed* data objects. This is essential both because the system is intended to be machine and language independent, and thus independent of internal representations of types, and because it must be possible to transmit messages containing port access rights. Because processes only ever have local names for ports, the kernel must be able to identify port names in messages and translate them from the local name space of the sender to that of the receiver. Ports are global only to a single machine; processes communicate with others elsewhere on a network by using ports owned by intermediate network servers on each machine. The servers deal with network addressing, the clients cannot know where the other parties to a conversation are located.

To cope with malfunctions, the IPC facility provides for *emergency* messages to be sent to a port by the kernel. These messages bypass the queue and are received before any normal messages. In this way, no special error handling functions are required, the receiving process merely distinguishes emergency messages by their type.

5.1.2. UNIX IPC Facility

This general IPC facility is mapped onto UNIX by specifying the relationship between its objects (ports, messages, etc.) and the normal UNIX objects. The most important of these relationships is that between ports and file descriptors, because this enables the IPC facility to provide file-like communication.

A file descriptor may stand for a *pair* of ports, or rather port descriptors. One of these is a remote port, for which some other process has receive rights, and this process has send access. The other is a local port, for which this process has receive access, and (presumably) the other process has send access. An I/O system call to such a file descriptor is converted into a **send** of a suitable message to the remote port, followed by a **receive** of a reply from the local port. Although the pair of ports in a descriptor provide a bi-directional data path, this does not imply both read and write file access via the descriptor; the data flowing in one direction may merely be arguments.

Another important relationship is between ports as global objects and file names. The owner of a port may attach it to a name in the directory hierarchy in two ways. The normal way is to assert the name, whereupon **opens** will return a file descriptor corresponding to a <local,remote> port descriptor pair. The alternative is to assert the name as a directory, whereupon **opens** which encounter this directory in their file name search will send a message containing the unparsed part of the filename to the asserting process and wait for a reply, before returning a similar file descriptor. This may be used, for example, to provide access to a remote filestore as if it were part of the local directory hierarchy. The server process would interpret the unparsed part of the file name as a name in the remote filestore.

The previous identifications permit the IPC to support file-like I/O, with certain new properties. The main one is that it is no longer possible for a client process to distinguish between services provided by the kernel, and services provided by a server process. The mode of access to either is via file descriptors.

Asynchronous I/O is supported by the ability to send a message without awaiting a reply,

success = Send(&msghead,timeout);

the ability to await messages from a *set* of ports,

success = Receive(portset,&msghead,timeout);

rather than from a single file descriptor. In both cases, **timeout** is the maximum time before return, and **msghead** is a structure containing descriptors for the port through which it was transmitted and for a reply port, and a description of the types and locations of the data, an extension of the **output_rec** used by Bell's multiplexer.

Using these facilities, and the "pending" message capability, server processes need never poll input or output ports; they may sleep on **receives** to their set of active ports without worrying about errors or output channels un-blocking, since both will cause messages. Alternatively, they may ask the kernel to generate a particular **signal** when a particular type of message arrives, or when a port un-blocks, and compute while not handling messages. The use of the IPC facility to support the protocol handler example is shown in Figure 6.

5.2. The Berkeley System

The team at Berkeley that maintains and develops UNIX for ARPA also requires networking facilities[5]. They are developing an IPC facility based on the layered model of networking used for the ISO Open System Interconnection architecture[11]. Their fundamental concept is of a global space of network *addresses*, each referring to an actual or potential *socket*. Processes invoke certain operations on sockets to achieve data flow between themselves. The set of operations applicable to an individual socket is determined by its *type*; several types of socket may be involved in supporting a single *protocol*. Initially, the system will support *datagram* and *virtual circuit* protocols.

Processes acquire the addresses of sockets with which they wish to converse by means outside the control of the IPC facility. Essentially, this means that some network addresses, those of certain server processes, must be *well-known*, that is, established by convention and wired into the programs using those services. Of course, the IPC facility may be used to build servers from which processes may enquire the addresses of servers providing different services, so that only a few names need be well-known.

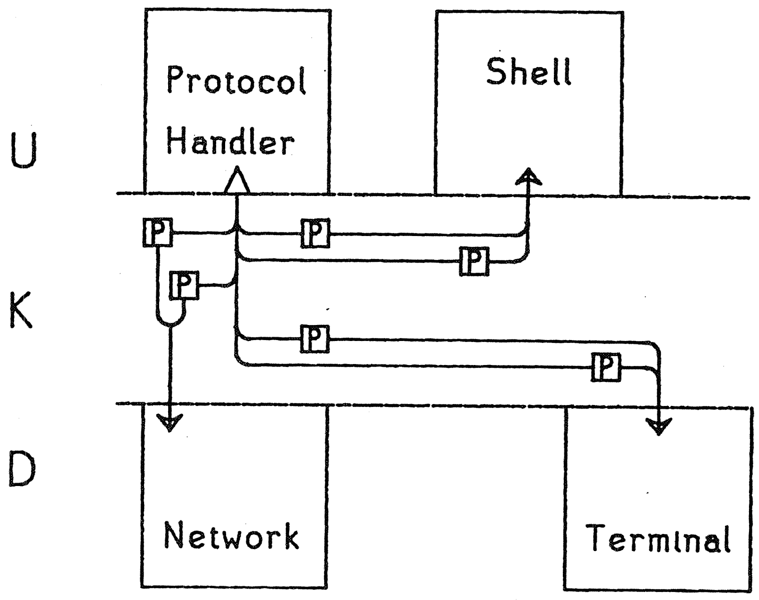

Figure 6 — CMU's IPC Facility for Network Access

A process creates a socket and obtains a "file descriptor" for it via

s = socket(type,&addr,&pref);

where **type** specifies the protocol to be used, **addr** will be set to the address chosen by the system for the socket. This will be **pref** if the process wants to create a well-known socket.

Using a descriptor for a datagram socket, a process may send a block of data to the socket whose address is **dest**, by

send(s,&dest,msg,len);

and receive blocks of data from other sockets, by

len = receive(s,&from,msg,MAXMSG);

when **len** (≤ **MAXMSG**) bytes will be stored at **msg** from the next datagram, and **from** will be set to the address of the **sending** socket.

The virtual circuit protocol involves two types of socket. A *call director* socket provides two operations. A process wishing to be called by others, typically a server of some kind, obtains a descriptor **s1** for a call director socket, and then invokes

t1 = answer(s1,&caller);

which waits for another process to invoke

t2 = call(s2,&callee);

on a call director socket for which it has descriptor **s2**, and with **callee** = **netaddr(s1)**. In the first process, **caller** will be set to **netaddr(s2)**, and a descriptor for a virtual circuit socket will be returned. In the second process, a similar descriptor will be returned. The two processes can use normal **read** and **write** calls to transfer data using these descriptors. A descriptor pair of this type can simulate a pipe.

Three facilities are provided to support communication with several parties at once. Non-blocking mode can be set on all descriptors. A facility called *watermarks* provides for **signals** to be generated whenever I/O becomes possible on a socket. The receiver will be signalled if more than **lowat** bytes accumulate, or if a timeout elapses and any data is available. A sender will be signalled if less than **hiwat** bytes are pending. The **select** facility enquires from a set of descriptors those available for immediate reading or writing.

File-like services may be provided by servers running on the *same* machine as their clients via *portals*, which provide file system names for special IPC sockets, without network addresses. A portal is created by

s = portal(kind,name,mode,server);

This creates the directory entry **name**, with the specified **mode**, and returns a descriptor for a call director socket. When a process **opens** the name, the kernel will place a **call** to this socket. If no process has a descriptor for the portal's socket when an **open** is attempted, the kernel will create a server process executing **server**; thus server processes need only exist while they are needed.

The **kind** of a portal specifies the file system calls the kernel will accept for it. Kinds are defined to emulate normal files, special files, and directories. When file system calls are applied to portals, the kernel packs their arguments into a standard record and sends it to the server. The server's replies, in a similar record, complete the file system call. Because portals are accessible only via file system names, the normal protection mechanisms apply. A facility is also provided to **associate** servers with network addresses, but this poses some protection problems, because there is no network-wide concept of "user".

5.3. Comparison of CMU and Berkeley Systems

The major difference between the two systems relates to addressing. The CMU system's ports have no addresses; the correspondence between a port on one machine and a port on another is maintained by communication server processes on each machine. The protocol to be used for a particular link is the concern of the communication server, thus the CMU system can span several incompatible networks. Because their clients are unaware of their location, servers may migrate about the network(s) in search of machines with the resources they need[4]. The Berkeley system's sockets have addresses, the correspondence is maintained by the kernel. This corresponds closely to real network protocols.

The difference between the CMU model of communication as messages, and the Berkeley models of datagrams and virtual circuits seem more apparent than significant. Servers written using either model look very similar.

Berkeley's I/O functions are always directed to a single socket, whereas some of CMU's functions are directed to a *set* of ports. Berkeley provide a separate **select** facility to discover the sockets to which an I/O operation may safely be addressed, and a **watermark** facility to enable sockets to signal their availability for I/O asynchronously. The race conditions separate facilities make possible do not appear serious, in other respects the separation seems advantageous.

The typed objects transmitted by the CMU system contrast with the byte streams transmitted by Berkeley's. The higher-level protocols needed to ensure that diverse machines interpreted information appropriately became onerous in the RIG system, on which CMU's is based. A system without global addresses must be able to detect port names in messages. On the other hand, the simplicity of byte streams has been a major advantage of UNIX.

6. Conclusion

Reviewing this varied collection of facilities, the requirements for UNIX to support asynchronous I/O, and the applications requiring it, may be summarised as:

- The capability to generate **signals** when data is available, or when it may be sent, to eliminate polling.
- The integration of inter-process and terminal I/O, so that **ioctl** may be processed either by a server or by the kernel, and that a client cannot know which is providing services. Ideally, facilities such as the terminal handler should be detached from particular files, and be capable of being added to *any* character special file or pipe. Bell Labs are working on this concept[7], which has affinities with the TOPS-20 terminal handler[3].
- The ability to create names in the directory structure referring to interprocess communication channels, so that the communicating processes need not be related.
- The ability to distinguish the originator of a message in an interprocess communication channel, so that a server may receive requests from many clients via a single channel.

The only widely available implementation is the multiplex file. In the short term multiplexers are capable of supporting networking, but in the longer term UNIXes will be operating in environments where networks will be linking many incompatible machines, and being used more ambitiously, and the extra capabilities of the CMU and Berkeley systems will become essential.

7. Acknowledgements

This work was supported by Science and Engineering Research Council grant GR/A80341. My grateful thanks are tendered to my colleagues Jim McKie, whose encyclopaedic knowledge of the UNIX kernel illuminated many dark corners of the code and documentation, and Fernando Pereira, who was a pleasantly demanding user of some of our experiments.

REFERENCES

[1] R. Ballocca, "Networking and the Process Structure of UNIX: A Case Study," *Proc. of COMPCON (Fall '78) Computer Communications Networks*, pp.306-311 (September 1978).

[2] G. L. Chesson, "The Network UNIX System," *Operating Systems Review* **9**(5), pp.60-66 (1975).

[3] DEC, "TOPS-20 Monitor Calls User's Guide," DEC-20-OMUGA-A-D, Digital Equipment Corp., Maynard, Massachusetts (May 1976).

[4] R. B. Dannenberg, "The Spice Butler," S110, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, Pennsylvania (October 1981).

[5] W. N. Joy and R. Fabry, "Proposals for Enhancement of UNIX on the VAX," Computer Systems Research Group, Dept. EECS, University of California, Berkeley, California (August 1981).

[6] R. J. Kummerfeld and P. R. Lauder, "The Sydney UNIX Network," *The Australian Computer Journal* **13**(2), pp.52-57 (May 1981).

[7] D. M. Ritchie, Bell Labs, EUUG Meeting, Amsterdam (April 1981).

[8] P. Lauder, "MX: An Indirect Driver for Multiplexing Virtual 'tty' Lines on to 'real' Lines," Basser Dept. of Computer Science, Sydney University, Sydney, Australia (October 1980).

[9] R. F. Rashid, "An Interprocess Communication Facility for UNIX," CMU-CS-80-124, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, Pennsylvania (February 1980).

[10] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Comm. Assoc. Comput. Mach.* **17**(7), pp.365-375 (July 1974).

[11] A. S. Tanenbaum, "Network Protocols," *Computing Surveys* **13**(4), pp.453-489 (December 1981).

Most of the information for this survey was obtained from the on-line documentation provided with the versions of UNIX distributed by Western Electric, the University of California at Berkeley, and the Australian and European UNIX User Groups.

MC NR

35225