ITC Suggestions for 4.X BSD

David S. H. Rosenthal
Michael Leon Kazar
David Nichols
Mahadev Satyanarayanan
Bob Sidebotham

Information Technology Center
Carnegie-Mellon University

Introduction

The mission of the Information Technology Center, which is funded by IBM, is to develop the software infrastructure for a campus-wide deployment of powerful personal workstations at C-MU. To this end, we have been working with a large number of 4.2 workstations of various kinds to:

1. Support remote file systems.

2. Support advanced user interfaces.

3. Support the SNA address family.

4. Support dynamic linking and shared libraries.

5. Support authentication in a hostile environment.

This experience leads us to suggest some directions in which we would like 4.X BSD to evolve, none of which are particularly original. Some, indeed, may be addressed in 4.3 BSD. We provide a brief survey of our experience, and then cover each of the directions in some detail.

Experience

Remote File Systems

The ITC has developed a remote file system that uses a workstation's local disk as a cache of recently used files and provides a large collection of untrusted workstations with the illusion of uniform access to a single vast Unix file system[1]. The files are actually stored on many cooperating trusted servers, which collaborate to appear as a single file system to clients. Two such file systems have been in production use since the fall of 1984, combined they support 115 clients using 6 servers.

The clients of this service run a modified kernel; `open()`, `close()` and some other calls are intercepted and turned into messages that appear in a special file. These messages are read by the user-level cache manager pro-

cess, which fetches files from the servers, stores them in a cache directory, and replies via the same special file. The replies contain the inode number of the cached copy, which is substituted in the remote inode and used by `read()` and `write()` calls.

The cache manager communicates with the file servers using an RPC mechanism on top of TCP/IP stream sockets. Each client has an individual server process on each server it uses.

The current revision of the file system is changing:

1. to UDP-based RPC, to avoid running out of file decriptors in the server process.

2. to a single file server process, to avoid the cost of communicating between server processes.

3. To have the server use special `openi()`, `readi()`, `writei()` system calls, to avoid `namei()` wherever possible.

Advanced User Interfaces

The ITC has developed a user-level window server[2], which is highly portable between displays and workstations (the most recent port took 4 hours 55 minutes; it runs on 7 different displays on 3 different workstations). It has been in production use since the end of 1983, and supports a user interface toolkit including a reformat-on-the-fly multi-font editor[3]. The toolkit has been used to develop a large collection of utilities and educational applications, including browsers for the file system, mail, and news, diagram and equation editors, and an implementation of the micro-Tutor CAI language.

The server needs to have the pixels on the display (and, preferably, the mouse position registers) in its address space. Depending on the particular workstation, some form of `mmap()` may be available to control this access, or it may have to be provided for all processes (as on the microVAX).

Clients communicate with the server using a special-purpose RPC protocol over TCP/IP stream sockets. The normal `stdio` buffering is used to batch calls until one needs a reply; this provides adequate performance only because care is taken to avoid replies wherever possible.

Conventional Unix applications require a tty of some kind. This is provided using either a 24*80 emulator or a reformat-on-the-fly typescript manager. Both use `ptys`, but the typescript manager has problems with this approach. It wants to generate echoes, manage rub-out and kill pro-

cessing, and so on, so it uses TIOCREMOTE mode. But it cannot find out about `ioctl()`s the application does, so it cannot disable echo correctly.

Surprisingly, this approach gives adequate performance. But the combination of paging and the 4.2 scheduler means that response is not very predictable. The window manager has much information about the processes likely to be interactive, but cannot transmit it to the scheduler and pager.

SNA

The ITC has developed an implementation of the LU6.2 version of SNA for 4.2. It has been in production use since the spring of 1985, driving IBM3820 laser printers at 19.2Kbaud via the Sun UARTS.

Dynamic Linking

The ITC is developing its user interface toolkit to support display, editing, storage and transmission of documents, files containing objects such as text, diagrams, equations, and others defined by particular applications. To permit applications to define new objects and, for example, to include them in mail, mail readers and writers must be able to locate and use newly defined editing and display code. The ITC has experimented with several dynamic linking schemes, including a compiler that generates pure position-independent code[4], and is currently using one that involves running an **ld**-equivalent over normal **.o** files at run-time. Other planned uses for dynamic linking at the ITC include user extensions to the window manager, for example special-purpose mouse tracking and curve drawing.

Authentication

The ITC has developed an authentication mechanism that can operate in a hostile environment, complementing the security features of the remote file system. Login, su, and other programs have been modified to consult a network authentication server, using a three-phase handshake, to obtain authentication tokens. Among these is a session key that encrypts the RPC headers between the local file system daemon and the file servers.

Suggestions

Virtual Memory Enhancements

The most important development of 4.X from the ITC's viewpoint would be an implementation of `mmap()`, substantially as specified. Mapping devices is essential to support bitmap displays, and is very useful for mice and up-down keyboards.

Shared libraries are becoming essential, and their implementation requires either:

1. A compiler that generates position-independent code.

2. The ability to map libraries at fixed positions into sparse pieces of a large address space.

In either case, copy-on-write semantics for the private variables of library routines is almost essential. Shared public writable pages, or a mechanism for sending pages to another process, would be useful as part of a high-speed RPC mechanism. They would avoid copying the RPC data twice across the user/kernel boundary.

Multiple File System Type Support

Many groups are developing remote file systems for Unix; what is needed is a common interface behind which they can all compete, analogous to SUN's **vnode** interface. Almost any interface that expressed file system calls in terms of operations on objects free of details about how they were represented on a medium would be acceptable. We believe, in particular, that the ITC's file system could be implemented behind the **vnode** interface. Doing so would save us the considerable effort involved in re-installing our file system hooks in multiple new versions of Unix.

It seems inevitable that `stat()`-ing a file in a remote file system that really implements Unix semantics is expensive. A major cause of `stat()` calls is `getwd()`, and as file systems become larger and people work further and further from the root `getwd()` will become steadily more expensive. It is fairly simple for the kernel to maintain the path used to get to the current working directory, and implement an efficient:

```
n = getcwd(buf, size, offset);
```

Intercepting System Calls

There are a number of cases in which it is desirable for user-level processes to intercept and process system calls issued by some other process. For example:

1. Remote file systems (for example VICE).

2. Emulating obsolete function (for example the TTY driver in a workstation environment).

Limited facilities of this kind were implemented as the *Stream I/O* system in the 8th Edition[5], and provided a more efficient and flexible replacement

for the TTY driver.

A full implementation would provide:

a) a bi-directional channel for transport of data and control blocks.

b) end modules controlled by a mask specifying which system calls are to be converted into protocol blocks (and *vice versa*). The modules should specify protocol for *all* descriptor-oriented system calls.

c) an interface for adding new processing modules into the channel.

d) Both block and character streams. Block streams could be mounted to support remote file systems.

Fast RPC Mechanism

Most of the new services we are developing use an RPC package to communicate with their servers. We use several different RPC packages:

a) a special TCP-based buffered implementation for the window manager.

b) a general TCP-based synchronous implementation for the initial version of the file and authentication systems.

c) a general UDP-based synchronous implementation for the re-implementaion of the file and authentication systems.

Performance of RPC is critical to the overall performance of the system, and is in general inadequate. There are two cases that need improvement:

1. On-machine RPC, which should be implemented using shared writable pages and an efficient semaphore mechanism, instead of file-based communication. This would avoid copying the arguments and return values twice across the kernel/user boundary.

2. Off-machine RPC, which should use a faster UDP `send()`.

Timing one-byte writes on a typical 4.2 implementation, we find:

```
 760 ms. for 1000 /dev/null writes (0.76 ms/call)
1800 ms. for 1000 file writes (1.80 ms/call)
4360 ms. for 1000 sendtos (4.36 ms/call)
3720 ms. for 1000 sends (3.72 ms/call)
```

It seems unreasonable that `send()`, which does not need any handshak-

ing, should be more than twice as expensive as a file write.

The 15% margin between send() and sendto() suggests that `sendto()` should cache the connection information and re-use it (i.e. postpone the `in_pcbdisconnect()` until the socket is used with a different address). We estimate that perhaps 80% of `sendto()`s refer to the same address as their predecessors. The route information should be cached in a similar fashion,

UDP is checksumming all output datagrams, even though these checksums are almost never checked.

Authentication Services

The 16-bit uid is too short. A scheme that enabled uids to act as public keys would be interesting. The ITC's experience in trying to develop a system in which the workstations trust only the file servers, and not each other, indicates that some further support for authentication is needed, but no definite suggestions have been agreed. One that has been implemented is the Process Authentication Group, an un-changeable (except by root) token stored in the proc structure that is inherited by the children of a process except if they are SUID. This allows the processes inheriting rights as a result of a single authentication handshake to be identified.

Lightweight Process Support

For most ITC programs, the only mechanism that is used for waiting is `select()`, and the 4.2 `select()` is far too expensive. In particular, applications such as the new file system servers and the SNA daemons which use the ITC's lightweight process support spend unreasonably large amounts of time in `select()`.

An `mmap()` that supported sparse address spaces would help with the management of multiple stacks.

Scheduling & Process Groups

Processes, process group leaders, and super-user processes should be able to raise and lower their priority and that of the group within the limits set by the priority inherited from their parents. This would enable the window manager to raise the priority of processes with the input focus.

References

1. M. Satyanarayanan *et al. The ITC Distributed File System: Principles & Design*, to be presented at the ACM Symp. on Operating Systems Principles, East Orcas, WA, December 1985.

2. J. A. Gosling & D. S. H. Rosenthal, *A Window Manager for Bit-mapped Displays and Unix*, to appear in *Methodology of Window Managers*, North-Holland.

3. J. A. Gosling, *An Editor-Based User Interface Toolkit*, Proceedings of PROTEXT 1984, Dublin October 1984.

4. M. L. Kazar, *Camphor: A Programming Environment for Extensible Systems*, Proceedings of USENIX, Portland OR, June 1985.

5. D. M. Ritchie, *A Stream Input-Output System*, Bell Labs Tech. J., **63**(8), October 1984 pp. 1897-1910.