Proposals for enhancement of UNIX* on the VAX

July 21, 1981
Revised August 31, 1981

William Joy and Robert Fabry

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

(415) 642-7780

ABSTRACT

This report describes several proposals for enhancements to the UNIX system on the VAX to meet the needs of the users in the ARPA research community.

The areas covered in this report include inter-process communication and networking facilities, segmentation and shared-file access, file system facilities and performance improvements, system support for large software projects and software distribution, standardization of system facilities, operational support, and ongoing software efforts.

An appendix provides a index to the document in a summary of proposed system facilities.

We welcome comments on these proposals, either by U.S. Mail to the address given above, or electronically. Our ARPANET addresses are wnj@berkeley and fabry@berkeley. Our *uucp* addresses are ucbvax!wnj and ucbvax!fabry. Electronic mail is preferred.

* UNIX is a trademark of Bell Laboratories.

TABLE OF CONTENTS

1. Introduction

This report presents our proposals for enhancements to UNIX on the VAX. Succeeding sections describe proposals for various parts of the system. The rest of this section outlines these proposals.

Section 2 describes a proposal for interprocess communication on UNIX and an interface using these IPC facilities to networks, both local and long haul. We expect that there will be many different networks interfaced to UNIX and that the facilities described here can be used to easily interface to these different networks.

Section 3 describes the proposed extensions to UNIX memory management. Current large scale AI and image processing programs are generally limited by architectural or system constraints to a few Megabytes of address space; by the end of the decade we expect that similar large programs may routinely use address spaces as large as a Gigabyte. VLSI design programs for large designs may likewise use enormous amounts of both space and time. The proposals in this section address the management of extremely large address spaces and propose a segment based view of virtual memory. Facilities to provide segment reference control and copy-on-write like facilities are also described. Special needs of programs that do involved stack manipulations are also addressed.

Section 4 describes proposed changes to the UNIX file system organization to provide greater throughput. The file system design focuses on information organization for maximum locality of access and high data throughput across a range of mass storage technologies.

Section 5 describes file system facilities that are needed for various applications but not provided by the current file system. Examples include locking of files to control concurrent access and symbolic links.

Section 6 describes system support for software projects and software distributions. It builds on the CMU project implementation, combining it with other facilities: source revision control, strong naming of projects, enhanced UNIX groups, standards for Makefiles, and automated distribution facilities. The proposed facilities provide for convenient distribution of large bodies of software.

Section 7 describes areas of the system where standardization on a single set of facilities will benefit the user community. New standards are suggested to cover the format of the system documentation, contents of system libraries, mail processing protocols and formats, the primitives for handling software signals, the interface of the terminal driver, the format of information used by debuggers, and the environment for screen management support.

Section 8 describes issues in operational support of the system. Several new facilities to be integrated or provided in the standard system are described: the notion of an operator, clean localization of the system (making more of the binaries cpu site independent), error logging, enhancements to dump and restore procedures, and provision of new archival and retrieval facilities.

Section 9 covers miscellaneous topics including the construction of a software availability database, hardware support, and the status of various system programs that are being worked on including debuggers and the FORTRAN 77 system.

We conclude in section 10 with a table of the proposed kernel facilities.

2. Interprocess communications and networking

This section describes our proposed inter-process communications facilities for UNIX. Our proposal constructs an IPC framework that can be used to build a number of different protocols for communication, and to support different distributed operating systems and applications.

Initially we intend to add the facilities described here to UNIX. We will then begin to implement portions of UNIX itself using the IPC as an implementation tool. This will involve layering structure on top of the IPC facilities. The eventual result will be a distributed UNIX kernel based on the IPC framework.

The IPC mechanism is based on an abstraction of a space of communicating entities communicating through one or more *sockets*. Each socket has a *type* and an *address*. Information is transmitted between sockets by *send* and *receive* operations. Sockets of specific *type* may provide other *control* operations related to the particular protocol of the *socket*.

In providing access to the communcations space, we will initially support only three socket types, but have specifically designed the facilities so that new socket types may be easily added. The initially proposed socket types provide virtual circuits and datagrams. Circuits are two-way reliable data streams, and datagrams are unreliable one-way messages that are sent without explicit acknowledgment and often with limitations on length. These facilities admit simple and efficient implementations both in the single machine case and when interfacing to network protocols, and this is why they were chosen initially.

The first version of the IPC facilities for UNIX will support an IPC address space that is an extension of the TCP/IP address space, a comparitively flat 32 bit address space with additional addressing available at each node. We expect to add generic addressing, broadcasting and multiplexing as needed and to experiment with the amount of late binding in the "addressing" scheme. The flexibility to allow this is explicitly provided by our basic model. We expect that in constructing a distributed UNIX system on top of the basic model we will provide services such as migration of processes, but we do not insist that the address space underlying the IPC have the ability to directly and transparently support migration; we will layer it on while implementing UNIX if necessary.

When we use the facilities described here to implement networked versions of the UNIX system we will build on the IPC address space to derive resource identifiers (larger objects that contain addresses, rights and authentication) and use encryption and other well-known techniques to create protection domains and do authentication. The reader is assumed to be familar with such techniques.

To support multiplexing of communications in UNIX both a synchronous facility based on the ADA *select* statement and an asynchronous software-interrupt (signal) based facility are provided. These facilities are not part of the basic IPC model, but of its embedding in the UNIX system.

The IPC facilities are integrated into the current UNIX name space by *portals*, entries in the file system that invoke server processes when accessed. These entries are designed to be used by naive processes that are unaware of the use of communication. The basic IPC communications facilities and *portals* may be used to provide services on a single machine and in a networked environment.

A more complete description of the motivation of the IPC architecture described here, measurements of a prototype implementation, comparisons with other work and a complete bibliography are given in CSRG TR/3: "An IPC Architecture for UNIX".

2.1. Goals

We see at least four distinct areas where UNIX IPC will be important:

* In supporting inter-process communication within a single machine.
* In supporting access to the facilities of the available local and long-haul networks.
* In constructing services on a tightly coupled set of machines to make the facilities of all machines available to users.
* In constructing servers for autonomous machines, which allow access to resources while retaining local administrative control.
* To provide uniform access to IPC objects and current UNIX objects.

In meeting these needs we wish to keep, as at present, the UNIX kernel largely as an i/o multiplexor. We wish to place facilities unrelated to the basic IPC mechanisms (such as name servers and authenticators) outside the kernel.

2.2. Assumptions

Our design is based on the layered models for distributed systems, such as the ISO Open Systems Architecture. We assume that the system facilities are built on services provided by network layers in that model and make assumptions in our design about the internetwork:

* The internetwork provides datagram services and perhaps virtual circuits.
* The internetwork provides origin and destination addresses in all messages.
* All entities with which we wish to communicate can be given internetwork addresses.

The facilities to be provided by the kernel to the users processes include:

+ Datagram and virtual circuit access to the network.
+ Buffering and multiplexing of communications.
+ Creation of servers when they are referred to, so that they need not pre-exist.
+ Translation of access to names in the UNIX name space into accesses to server processes.
+ Translations of system calls into protocol when communicating with servers that simulate UNIX objects such as file and directory hierarchies.

Facilities not to be provided by the kernel are:

– A network name server.
– Control of information access and protection in the network.
– Transmission of structured information and data representation conversion.

Such facilities are desirable, but will be implemented outside the kernel so that application-specific and site-specific facilities can be created.

2.3. Addresses and sockets

We assume that the transport layer of the system provides us with an internetwork wide address space. Each message to be sent includes source and destination addresses. The type *in_addr* will be used to refer to an internetwork address. We expect, but do not require, that such addresses be of fixed length.

For definiteness the reader may assume that an *in_addr* has the following form:

```
typedef struct in_addr {
        int     ipaddr;             /* internet address */
        int     moreprecise;        /* sub-addressing at destination */
} in_addr;
```

We expect that some internetwork addresses will be generic and some will be location independent. The resources available in this way will vary from network to network.

Our proposal uses a *socket* abstraction in both the circuit and datagram implementations. Sockets are the destination of all internetwork communication. If a socket is not active (no process is servicing it) when communication is attempted to the socket then the information may be discarded or a server may be created to service the socket.

The types of sockets available are represented by the type *in_proto*:

```
typedef enum in_proto { SOCK_DG, SOCK_CALL, SOCK_VC } in_proto;
```

Each socket has some buffering associated with it. SOCK_DG datagram sockets buffer incoming datagrams; SOCK_CALL call director sockets buffer incoming and outgoing calls; SOCK_VC virtual circuit sockets have a queue for incoming data on their circuit and logically reference a matching SOCK_VC socket where transmitted data is stored.

Active sockets are referenced by small integer "file descriptors". A set of file descriptors is represented by the type *fd_set* that is represented by a bit string and is used in the *select* primitive for synchronous i/o multiplexing.

2.4. Datagram facilities

A datagram is a short piece of data sent to a specific socket address. No guarantee of reliable delivery is made for datagrams, and they are typically limited in length to just over 512 characters per datagram.

A socket for receipt of datagrams may be created by using the *socket* system call:

```
in_addr addr;
in_addr pref;
int s;

s = socket(SOCK_DG, &addr, &pref);
```

The returned *s* is a descriptor for a socket, and the returned *addr* is the address of the created socket. If the third argument to the *socket* call is a 0, then the system chooses an address for the created socket. You can specify *pref* if you wish to set up a specific, well-known socket, e.g. for a server. If an error occurs then a −1 value is returned for *s* as is normal in UNIX.

To send a datagram from a socket the system provides a *send* primitive, which is invoked

```
in_addr dest;
char *msg; int len;

... initialize values of s, dest, msg, len...
send(s, &dest, msg, len);
```

to send *msg* of *len* bytes to *dest*. The value of *dest* must be initialized before this call from well known data (e.g. the network equivalent of "411" and "555-1212" or "15.000Mhz") or by obtaining it from another process.

A datagram can be received by a *receive* system call:

```
int d;
in_addr source;
char msg[MAXMSG]; int len;

... initialize socket d with addr dest as above...
len = receive(d, &source, msg, MAXMSG);
```

that returns, in the supplied message buffer *msg*, *len* bytes from the source address returned in *source*. If the datagram would not fit in the supplied buffer, then the remainer is discarded and the *len* gives the length of the datagram before truncation. Each *receive* call removes a single datagram from the buffer space associated with the socket.

The following example shows a time server program that creates an internetwork datagram socket to which a message can be sent causing a message with the time to be returned. It could be used by a small computer on a network to obtain the time of day from a central server.

```
#include <inet.h>       /* defines in_addr, SOCK_DG, etc. */
#include <types.h>
#include <wellknown.h>          /* defines WWV_ADDR and others */

/* tsaddr is the well-known-address of the time server */
in_addr  tsaddr = WWV_ADDR;

main()
{
    char buf[1]; int len;
    in_addr addr;
    int s;
    char *ctime(), timestr;
    time_t t;

    s = socket(SOCK_DG, 0, &tsaddr);
    if (s < 0) { printf("can't get socket\n"); exit(1); }
    for (;;) {
        /*
         * We receive a datagram and discard its contents,
         * to get the address of the sender.  A more sophisticated
         * time server might handle several requests based
         * on the contents of the received datagram.
         */
        receive(s, &addr, buf, sizeof (buf));
        time(&t);                       /* get binary time */
        timestr = ctime(&t);            /* convert to string form */
        send(s, &addr, timestr, strlen(timestr));
    }
}
```

Here the *socket* call associates this process with the time server socket whose address is specified, returning −1 if there is something wrong with *ts_addr* (i.e. not providable on this machine) or if the socket is already in use (e.g. by another instance of the time server). If the socket is openable the server loops reading a packet from the socket for the sole purpose of obtaining the address it came from and sending back the time without further ado.

2.5. Circuit facilities

To use a virtual circuit one first obtains a SOCK_CALL call director socket that is associated with a specific network address. Calls may be placed from and answered at this socket. Each call placed or answered yields a distinct new SOCK_VC virtual circuit socket that allows for the reliable, flow-controlled transmission of arbitrary amounts of data to and from the party at the other end of the circuit. Circuits allow specially marked *urgent* information to be sent, give out-of-band notification of the presence of urgent data, and allow record boundaries to be marked in the stream. These circuit options are described in section 2.8.

Processes can send and receive data on a circuit with the normal UNIX *read* and *write* calls. Conversations are flow controlled by the underlying mechanisms; if the sender writes data faster than the receiver can accept it, the sender will block. If the receiver reads data when none is available, it will block pending receipt of more data.

In the default stream mode, a read returns as soon as data is available and the system does not preserve any boundaries within the information stream. A record oriented mode for data transmission will be describe in section 2.8.

So that incoming and outgoing calls may be queued, a process must have access to a call director socket to place or receive a call. A SOCK_CALL socket is created with a *socket* call:

```
int s;
in_addr addr, pref;

s = socket(SOCK_CALL, &addr, &pref);
```

The returned *s* is a "file" descriptor for a socket for establishing virtual circuits, by calling and receiving calls. When calls are placed or answered additional descriptors are obtained for the SOCK_VC virtual circuit sockets corresponding to the calls.

A call is received by doing:

```
int t;
in_addr caller;

t = answer(s, &caller);
```

This returns a descriptor for the new SOCK_VC socket for the conversation with *caller*. Several *answer* calls may be done on a single call director socket; each yields a SOCK_CALL virtual circuit socket representing a single conversation.

To place a call establishing a circuit one must first have access to a SOCK_CALL call director socket at some address. Assuming the SOCK_CALL socket exists as *s* created as above, a call could be placed by:

```
int t;
in_addr callee;

... initialize callee ...
t = call(s, &callee);
```

After placing a call, a new descriptor is obtained corresponding to the new SOCK_VC virtual circuit socket. If the call fails then a value of −1 is returned. When the conversation with *callee* is complete, the virtual circuit socket *t* can be closed.

Both *call* and *answer* may be done at a single SOCK_CALL socket.

The following example uses the circuit facilities build a telnet server creating server processes (login commands) each time someone connects to the telnet socket:

```
#include <inet.h>
#include <signal.h>
#include <wellknown.h>

in_addr teladdr = TELNET_ADDR;

main()
{
    void reaper();
    int s = socket(SOCK_CALL, 0, &teladdr);

    if (s < 0) { printf("can't get socket\n"); exit(1); }
    sigset(SIGCHLD, reaper);
    for (;;) {
        int t = answer(s, 0);
        if (fork() == 0) {
            dup2(t, 0); dup2(0, 1); dup2(0, 2);
            close(s); close(t); close(p);
            execl("/etc/tellogin", 0);
            exit(1);
        }
        close(t);
    }
}

#include <wait.h>
/* reaper() allows all children which have died to exit. */
void reaper() { while (wait3(0, WNOHANG, 0) >= 0) continue; }
```

Here the basic server *answers* to the telnet socket it created. Each time a connection is made to the virtual circuit socket a new instance of a special login server /etc/tellogin is created. When a login is complete, the child exits and the *reaper* routine is called with a signal; it collects the terminated children.

2.6. Multiplexing facilities

In writing communications oriented programs it is often desirable to process information arriving from more than one source. The proposed IPC facilities provide three mechanisms for use in handling communication with more than one party: a synchronous facility based on the *select* statement, a facility for preventing i/o operations from blocking, and an asynchronous facility based on software interrupts. The latter two facilities will be described in section 2.9. We here describe multiplexing with *select*. Multiplexing facilities are generally useful for UNIX and we expect they will be gradually made available for more system services and devices. We expect to provide them for terminals with the first release of the IPC.

To support synchronous processing of information from more than one source we provide a *select* call, of the form:

```
int nfds, nready;
fd_set reads, writes;

nready = select(nfds, &reads, &writes, timeout);
```

The *select* call is provided with a structure describing file descriptors that are interesting; *reads* for descriptors where readability is interesting and *writes* for descriptors writability is interesting. The system examines each specified descriptor to see if there is an input or output operation possible on it, and returns in *reads* and *writes* sets of all such descriptors. *Nfds* gives the count of descriptors representable by type *fd_set* so that the size of the second and third arguments to *select* need not be fixed in the system, but may vary from program to program.

Either *reads* or *writes* may be specified as 0 to denote that no descriptors are interesting to read or write. If no descriptor comes ready within *timeout* milliseconds, the *select* returns, returning a value of 0. *Timeout* may be 0 for immediate return or −1 to not return prematurely.

The name *select* is chosen from the name of the statement in the ADA language whose semantics are similar. The *select* statement is also similar to the *await* mechanism provided in extensions to UNIX at BBN. The difference is the way that the interesting sockets are described and returned. With *await* the system keeps a list of interesting file descriptors internally, instead of having it specified at each call, and the return value is an array of integers instead of a bit mask. *Await* does not provide the timeout facility. Library routines to simulate *await* could easily be implemented using the facilities of *select*.

An important point in the semantics of *select* is that it imposes no bias. The mechanism for selecting among sockets that can be processed is left to the user.

The previous example program made use of an asynchronous facility for handling process termination. A reasonable extension to UNIX would be to provide a record on a special circuit when child processes terminate. This program could then be written using *select* to service the two circuits synchronously.

Assume that a call *waitsocket* yields a socket on which messages of type *child_status* are placed when child processes terminate. A revised version of the previous example is shown below.

Here we have used standard library routines *setfd* that adds an element to a bit-set of type *fd_set* and a routine *getfd* that destructively removes an element from one of these sets returning the value −1 when the set is empty.

2.7. Portals

The mechanism whereby services may be created in the UNIX file system name space involves creating a bridge between the file system name space and an IPC socket called a *portal*. Portals are client/server links and as such are asymmetric. The client accessing the portal may well be unaware that the object referenced is not a traditional UNIX object; in all but the most trivial cases, the server of the portal is interpreting a protocol and is cognizant of the existence of the portal.

A portal is created by the call

```
#include <inet.h>
#include <signal.h>
#include <wellknown.h>

#define  FOREVER     -1

in_addr teladdr = TELNET_ADDR;
fd_set sandp, choose;

main()
{
     int s = socket(SOCK_CALL, 0, &teladdr);
     int p = waitsocket();
     int t;

     if (s < 0 || p < 0) { printf("can't get socket\n"); exit(1); }
     setfd(&sandp, s); setfd(&sandp, p);
     for (;;) {
          choose = sandp;
          select(NOFILE, &choose, 0, FOREVER);
          while ((i = getfd(&choose)) >= 0) {
               if (i == p) {
                    child_status chstatus;
                    read(p, &chstatus, sizeof (chstatus));
                    continue;
               }
               t = answer(s, 0);
               if (fork() == 0) {
                    dup2(t, 0); dup2(0, 1); dup2(0, 2);
                    close(s); close(t);
                    execl("/etc/tellogin", 0);
                    exit(1);
               }
               close(t);
          }
     }
}
typedef enum portal_kind
  { PORTAL_CALL, PORTAL_FILE, PORTAL_DEV, PORTAL_DIR; }
portal_kind;

portal_kind kind;
char *name;
int mode;
char *server;
int s;

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

where ***name*** is the pathname for the portal, ***mode*** is the UNIX protection mode for ***name***, and ***server*** is a string specifying for the server to be invoked when the portal is accessed. The ***kind*** specifies the type of portal, and thereby specifies the protocol generated by the kernel for operations by client processes on it. The *s* returned is a descriptor for a SOCK_CALL call director socket to which the kernel will place calls when *opens* are done on ***name***.

UNIX protection modes are used to control access to the sockets associated with a portal. The call director socket for a portal is not accessible using internetwork addresses. It is therefore accessible only using a reference through the file system name space.

2.7.1. Portal protocols

The portal types are implemented by the kernel by translating system calls applied to the file descriptors returned from *opens* on a *portal* into protocol records on the SOCK_VC sockets the server receives when it answers *calls*. The exact specification of these protocols is beyond the scope of this paper, but we outline the basic nature of the protocols here.

A PORTAL_CALL portal acts like a virtual circuit socket, and simply passes calls onto the underlying SOCK_CALL socket.

A PORTAL_FILE translates reads and writes on the underlying SOCK_VC resulting from an *open* into a record-oriented request packet to the server. The kernel expects an appropriate reply to complete the operation for the client. Operations *fstat* and *lseek* are also possible on descriptors obtained by clients by opening a PORTAL_FILE.

A PORTAL_DEV is like a PORTAL_FILE, but also allows *control* operations, a generalization of *ioctl* to be described in section 7.6. A PORTAL_DEV thus can be used to simulate a general UNIX device, such as a terminal.

A PORTAL_DIR can be used to simulate a UNIX directory, as calls such as *open*, *unlink* and *creat* are translated into appropriate protocol. A result of such a call is often another connection to a service process to provide a file interface via the PORTAL_FILE or PORTAL_DEV protocol.

The system call *chdir* to remote directories can be supported by allowing the current directory to be a connection to a server implementing the PORTAL_DIR protocol.

2.7.2. Portal activation

The service process need not exist when a portal is first referenced. If it does not, a socket is created and associated with the in-core information about the file system entry for the portal. The *server* string is taken as a path name of the server program and that server is created in the environment of the process referencing it, receiving as descriptor 0 the socket associated with the portal, inheriting the current directory and user-id of the accessing process. The server process may be set-user-id to allow it to run in a different protection domain. The server process created has as parent the process that created it but is marked to not notify the parent when it finishes execution, since the accessing process is not aware of its presence.

The portal process may service more than one request on the descriptor or exit at any time. Processes accessing a portal may wait for the server to service them much as callers wait for an *answer* to occur on a virtual circuit.

When a portal is created the *portal* call returns a descriptor for the portal. Portals thus are created *live*. If the pointer to the *server* in a portal call is 0, this portal is accessible only while it is live; the portal will be closed if the server dies. A process may thus establish a portal that it will serve and bypass the server creation mechanism.

2.7.3. Portal examples

The example given below shows a mail server utility that looks up forwarding addresses:

```
main()
{
        int p;
        char *lookup();

        unlink("forwarding");
        p = portal(PORTAL_CALL, "forwarding", 0666, 0);
        for (;;) {
                int s, len;
                char name[128]; char *addr;

                s = answer(p, 0);
                recordmode(s, 1);
                len = read(s, name, sizeof (name));
                addr = lookup(name);
                write(s, addr, strlen(addr));
                close(s);
        }
}
```

The server creates a portal named *forwarding* of virtual circuit type. If you want to look up a forwarding address you can do:

```
FILE *f = fopen("forwarding", "rw");
recordmode(fileno(f), 1);
fprintf(f, "jones\n");
fgets(f, buf);
```

We could also write a server to be created automatically instead of manually. We would create the portal using a call:

```
portal(PORTAL_CALL, "/etc/forwarding", 0666, "/etc/forwarder");
```

Then when the file /etc/forwarding is first referenced, a /etc/forwarder will be created to service it. This portal would normally be created by a shell command:

```
$ portal call /etc/forwarding /etc/forwarder
```

The server /etc/forwarder would be created with descriptor 0 referring to the portal /etc/forwarding, and would be written:

```
main()
{
    char *lookup();

    for (;;) {
        int s, len;
        char name[128]; char *addr;

        s = answer(0, 0);
        recordmode(s, 1);
        len = read(s, name, sizeof (name));
        addr = lookup(name);
        write(s, addr, strlen(addr));
        close(s);
    }
}
```

A server could be created in internetwork space by using a *socket* instead of a portal, or automatically created on reference in internetwork address space using a *association*. These facilities are discussed in the next section.

2.8. Providing network accessible services

Recall that *portals* are not accessible using the internetwork addressing mechanisms, so that UNIX protection applies to them. It is thus necessary to provide a separate facility to allow servers to be dynamically created as a result of internetwork address space references.

The call

```
in_addr addr;
in_proto kind;
char *server;

associate(&addr, kind, server);
```

specifies that a server of type *kind* is to be provided for internetwork address *addr*; the address must be on the current machine. A reference to the address *addr* causes the specified *server* to be created and given access to the newly created socket of type *kind*, either SOCK_DG or SOCK_CALL. The created process will be run with user-id and group-id of the user who supplied the association, from the root directory of the file system, and with the system initialization process as parent. The power to create associations may be limited administratively on a particular machine. It is likely that certain internetwork addresses will be reserved to privileged user-id's, and that normal users would not be allowed to specify these addresses for associations.

An association may be removed by a

```
disassociate(&addr);
```

As an example of the use of associations, assume that an internetwork registry exists on the local network and we wish to create a service program that will be known to the registry. The program given below creates an association for the server and registers it with the registry. This program could be invoked as

```
$ register servicename program
```

to register *servicename* to access *program*. We assume that the registry

operates by accepting a *call* from the program followed by three records on the connection: the operation type as the first record, consisting of the word *register* for registration requests. For registrations the second record is the name to be registered, and the third record is the internetwork address.

Note: in this example we use *printf* to print error messages; in a production program we would use the C library routine *perror* that looks up an error message, and can yield more precise system characterizations of the error. We use *printf* here since the error messages in the source can help understand the program while calls to *perror* would all have the form

```
perror(x);
```

where *x* would be *s* or *t*. This is not enlightening to the code reader.

```
#include <inet.h>
#include <wellknown.h>

in_addr  registry = REGISTRY_ADDR;      /* well-known */
in_addr  serviceaddr;
char     response[128];

/*
 * register servicename program
 */
main(argc, argv)
     int argc;
     char *argv[];
{
     int s, t;
     char *servicename, *program;

     if (argc != 3) {
          printf("usage: register servicename program\n");
          exit(1);
     }
     servicename = argv[1];
     program = argv[2];
     /*
      * Get a socket to call the registry with.
      * Since both this and the socket to be registered
      * are assumed to be call director sockets we simplify
      * the program by just registering the socket we are talking on.
      */
     s = socket(SOCK_CALL, &serviceaddr, 0);
     if (s < 0) { printf("no sockets available\n"); exit(1); }
     t = call(s, &registry);
     if (t < 0) { printf("registry doesn't answer\n"); exit(1); }
     if (associate(&serviceaddr, SOCK_CALL, program) < 0) {
          printf("can't associate service\n");
          exit(1);
     }
     recordmode(t, 1);
     write(t, "register", 8);
     write(t, servicename, strlen(servicename));
     write(t, &serviceaddr, sizeof (serviceaddr));
     closesend(t);
```

```
        if (read(t, &response, sizeof (response)) < 0) {
            printf("no response from registry\n");
            exit(1);
        }
        if (strcmp(t, "ok") != 0) {
            printf("error registering: %s\n", response);
            disassociate(&serviceaddr);
            exit(1);
        }
    }
```

We note in passing that the placement of the service name in the registry and the placement of the association of the name in the local association table would ideally be done as a single distributed atomic operation.

2.9. More details about circuits

We now describe the rest of the facilities and attributes of virtual circuits that were not yet described. The calls described in the following sub-sections are written as library routines, and will use the *ioctl*-like system control interface (see also section 7.6).

2.9.1. Record mode

Circuits support a record mode, where each piece of data written on the circuit is considered a single record, and reads return complete records. This allows records to be read and written conveniently. The call

```
recordmode(s, 1);
```

sets a virtual circuit socket to be in record mode. A newly created virtual circuit socket is not in record mode. Record mode may be disabled by doing

```
recordmode(s, 0);
```

If you read only part of a record while in record mode because the buffer supplied to *read* or the read buffering of the socket is insufficiently large to contain the entire record, then the remainder of the record made available on successive reads. The call

```
recordbetween(s);
```

returns 1 if the specified stream is at a boundary between records, or 0 if it is not.

If only the writer is in record mode, then reads will never return data across record boundaries. If only the reader is in record mode then data will normally be aggregated to requested lengths before being presented to the reader.

A record may be created from data presented in multiple *write* calls by turning record mode off, writing data as required, and turning record mode on just before the last write in the record.

2.9.2. Urgent data

Circuits support a notion of urgent data. A circuit can be set into urgent mode by doing

```
urgentmode(s, 1);
```

or disabled by specifying a second argument 0. Data transmitted while in urgent

mode is marked, and causes the recipient of the data to process it specially. By default, urgent data arriving on a circuit causes generation of a signal SIGURG. This signal may be ignored if urgent data is to be processed synchronously.

The set of channels with urgent data may be determined by doing

```
fd_set whichareurg;

... initialize whichareurg to interesting sockets ...
urgentsockets(NOFILE, &whichareurg);
```

This selects out of the sockets in the bit-mask *whichareurg* those with pending urgent data; all other bits are cleared.

While a socket has pending urgent data the

```
urgentpending(s);
```

call will return true. When the next byte to be read is part of urgent data the predicate

```
urgentnext(s);
```

will return true.

The normal way of processing urgent data is to read out records from the input until the *urgentpending* flag drops. Then the last piece of urgent data will remain in the input buffer.

A single *read* call never returns both urgent and non-urgent data; it therefore suffices to check *urgentnext* before each call to *read* to determine the type of the data to be read.

2.9.3. Failure of circuits

If a permanent failure occurs in a circuit the circuit will be marked invalid. A process that attempts to read from or write to a failed circuit will be given an error indication and then sent a signal indicating a broken connection if further reads or writes are attempted. When processing circuits asynchronously a notification is sent immediately when a circuit fails; see section 3.5.3.

2.9.4. Circuits simulating pipes

A circuit can be used to simulate a pipe directly as the semantics are upward compatible; the reverse direction of the circuit will not be used, and can be severed to prevent accidental use. If the circuit fails, the signal sent on the next access to the circuit performs the same function as the SIGPIPE signal for pipes.

2.9.5. Closing

The call

```
closesend(t);
```

reports to the other party in a call that the call is no longer needed by sending an end-of-file on the connection. The call will continue while the other party is sending, and more data can be received on *t*, but no more data may be sent.

When all copies of the descriptor *t* created in *fork* or by *dup* have been destroyed, the circuit will be shut down after allowing the write buffers to drain.

Calls pending when a call director socket closes cause a new server to be created to service it if the socket has a server via a *portal* or a *association;*

2.10. Non-blocking and interrupt-driven i/o

To support servers and other processes that wish to not block in doing communications processing, a call to set a *socket* or other UNIX file descriptor into a non-blocking mode is provided:

```
nonblocking(s, 1);
```

After setting a socket non-blocking, operations that would block because of insufficient buffering on output or lack of available data on input will return a new error ENBLOCK. This is normally returned to a caller in C as a −1 return from a system call, with the global variable *errno* set to ENBLOCK.

The operation can be retried later, as *select* will report the socket ready when it becomes unconstipated.

A *call* placed on a non-blocking call director socket will immediately return a SOCK_VC virtual circuit socket descriptor, even though the call is not complete. The returned file descriptor will *select* as ready for writing when the call completes or fails to connect. At that point a *socketstatus* operation can be done on the circuit socket to determine the status of the *call*. A *timeout* may be used with the select to limit the length of time spent waiting for a call to complete.

Certain applications may require that they be notified immediately whenever input/output is possible. If such asynchronous operations are required, this can be enabled by doing:

```
asynchronous(s, 1);
```

Then when input is available or output becomes possible after a blockage the process that is doing *asynchronous* processing on the socket is notified with a SIGIO signal. A *select* with a *timeout* of 0 can be used to identify the subset of the asynchronous sockets that need service.

Asynchronous can also be used in addition to *nonblocking* when placing and receiving calls. The sequence:

```
in_addr addr, dest;
int s, c;

... initialize dest in some manner ...
s = socket(SOCK_CALL, &addr, 0);
nonblocking(s, 1); asynchronous(s, 1);
c = call(s, &dest);
```

places a call on the socket *s* and immediately returns a descriptor *c* because the socket *s* is marked non-blocking. Because *s* is marked asynchronous, a SIGIO is posted when the call to *dest* succeeds or fails and the call socket *c* will appear in a *select* as ready for writing. A *socketstatus* call, described below, can be used to determine whether the call succeeded or failed.

A similar technique can be used with *answer;* if a *call* were placed to socket *s* in the example above then a SIGIO would also be generated, and the socket *s* would show as being readable, the data being the incoming call. A *answer* could be used establish connection.

SOCK_VC virtual circuit sockets marked asynchronous cause SIGIO to be sent immediately when the circuit fails.

Because of the specialized nature of *asynchronous* i/o and to avoid difficult semantic and implementation difficulties only one process may mark a socket asynchronous at a time.

2.11. Status inquiries, watermarks, and options

A *socketstatus* operation can be used to get information about a socket:†

```
in_status state;

socketstatus(s, &state);
```

in the following structure:

```
typedef struct in_status {
    in_proto protocol;      /* SOCK_DG, SOCK_CALL or SOCK_VC */
    in_addr source;         /* socket address */
    in_addr dest;           /* destination address, for circuits */
    in_state state;         /* state of the connection */
    fd_waterm srcwm;        /* watermarks for sending */
    fd_waterm rcvwm;        /* watermarks for receiving */
} in_status;
```

The *protocol* field tells the protocol the socket supports; the currently defined protocols are SOCK_DG for datagram protocols, SOCK_CALL for call director sockets where *call* and *answer* are possible, and SOCK_VC for the virtual circuit sockets resulting from *call* and *answer*. The field *addr* is the address of this socket. The field *dest* is used only for SOCK_VC sockets, where sockets obtained by *call* or *answer* report peer addresses.

The field *state* shows the state of a call in a SOCK_VC, and has the values:

IN_CALLING	Call is pending
IN_CALLFAILED	Call failed
IN_OPEN	Call has succeeded and circuit is open
IN_CLOSING	Call is closing
IN_CLOSED	Call has closed
IN_BROKEN	Call broke due to some failure

The watermark fields specify the amount of transmit and receive buffering in this file descriptor. Each has the following structure:

```
typedef struct fd_waterm {
    int  lowat;
    int  hiwat;
    int  timeout;
} fd_waterm;
```

The *hiwat* watermark reflects the total amount of buffering available. The *lowat* and *timeout* are used in non-blocking input/output. On output, a non-blocking sender will receive an error when the high water mark is reached and the data is not transmissible within *timeout* milliseconds. The sender will be notified when the amount of output pending drops to the *lowat* watermark.

A receiver will be notified if *lowat* data accumulates, or if any data has accumulated and *timeout* time has elapsed.

The *lowat* and *hiwat* are in bytes, and the *timeout* is measured in milliseconds. Reasonable defaults for the various fields are set by the system. The watermarks may be set by the user by

† This call is implemented as a *ioctl*.

```
fd_waterm rdwm, wtwm;

watermarks(s, &rdwm, &wtwm);
```

where either the second or third argument may be specified as 0 to specify that the read or write watermarks are not to be changed.‡

The interpretation of options for data transmissions such as priority and security classifications varies from network to network and tends to be interpreted in ways that are hard to generalize to different networks. This is akin to device control, where different devices will allow different operations. Instead of specifying all possible options with each message to be sent, which would involve complicated processing for each message, we will use per-socket state to localize most of the option setting to the socket setup phase.

UNIX currently provides an *ioctl* operation to deal with device specific control operations, and we wish to use a similar mechanism for socket option specification. See section 7.7 for a discussion of some problems with *ioctl*, and a description of the *control* operation to be used here. We define *control* operations on sockets to set options. For example:

```
control(f, "precedence", "high", -1, 0, 0);
```

could set the precedence of the circuit *f* to be high and

```
char security[32]; int slen;

slen = control(f, "security", 0, 0, security, sizeof (security));
```

might return the current security of *f* as a character string to *security*.

The *watermarks* primitive of the previous section might be implemented by:

```
watermarks(s, rdwm, wtwm)
    int s;
    fd_waterm *rdwm, *wtwm;
{

    if (rdwm)
        control(s, "readwm", (char *)rdwm, sizeof(*rdwm), 0, 0);
    if (wtwm)
        control(s, "writewm", (char *)wtwm, sizeof(*wtwm), 0, 0);
}
```

We intend to study the appropriate standard set of *control* operations for sockets and provide suggestions for such a set at a later date.

2.12. Extensions being considered

The facilities described here provide basic access to the communications model described at the beginning of section 2. They can be used to provide higher-level facilities such as location-independent resources and resource access with different naming, protection and error-recovery strategies.

The facilities can also be extended in two ways: by extending the communications facilities (more sophisticated addressing; more protocols), or by extending the interface provided by the UNIX kernel to application processes (building higher level facilities than provided by the communications facilities).

‡ The *watermarks* call is implemented as an *ioctl*.

We expect that additional socket types corresponding to different communication models will be desirable. For example a reliably-delivered-message abstraction seems useful, independent of the connection implied by a virtual circuit. This abstraction could be provided by a SOCK_RDM socket type given a definition of the semantics of failure to deliver.

At the UNIX level we expect to provide additional facilities for controlling and debugging communcations. We expect that it will be desirable to be able to control all aspects of selected processes input/output behavior to debug them or simulate any desired environment. We expect to provide hooks for a controlling process to monitor the requests made by a process and to be able to interpose itself in communcations to take traces or redirect data.

The ability for processes to exchange access to existing sockets seems desirable to many systems builders. This can be provided by allowing processes to yield sockets to other processes wish to take them. We believe that this facility is properly part of UNIX, not part of the underlying communcations mechanism. We intend to provide such facilities in the network operating system version of UNIX. Similarly, we believe that the migration of processes can be provided without the aid of special mechanisms in the communications media.

2.13. Status of the implementation

We have implemented a prototype of the mechanism described here that supports single-machine pipes and datagrams, and have been using it on our development machine for a several months. It is significantly faster than the older IPC mechanisms of UNIX (mpx and pipes) and simple to implement.

We are working a full implementation of this IPC that we will interface to TCP/IP running on the ARPANET and also to our local area networking hardware (3M ETHERNET). We expect that this implementation will be in a form suitable for testing at other sites in the fall of 1981.

2.14. Alternatives and comparison

We are considering alternatives to the *urgent* data handling mechanism here. A reader of an earlier version of this proposal pointed out that a more convenient mechanism might be a non-blocking *readurgent* call.

Rashid at CMU has implemented a message-based IPC for UNIX that also serves as the basis for the SPICE machine operating system on the PERQ. The CMU IPC differs from our proposal in several ways:

- It provides reliably delivered messages rather than datagrams and circuits. The messages have attributes as being either reliable or unreliable and have headers that contain many of the fields found in the TCP protocol. With the mechanisms proposed here messages can be constructed by applications either based on datagrams or on top of circuits. A new socket type could be added to implement reliable messages in the primitives layer.
- The targets of message transmission are not fixed in location, but may be moved from machine to machine in a way transparent to user processes. In our proposal, such migrations are the responsibility of the application programs, that communicate about such movements using the internetwork address space for reference.
- The CMU IPC will do data representation conversions and scatter and gather data to and from the process address space when messages are sent or received. In our proposal such facilities are the function of application libraries, not of the UNIX kernel.

* Selection facilities are built into several IPC calls. In our proposal they are available as a separate *select* facility that can be used with other UNIX file descriptors.

We expect to compare the facilities, performance, and usage of the CMU and Berkeley IPC proposals more in the near future.

3. Memory management facilities

In this section we describe proposed enhancements to the memory management facilities of UNIX to allow UNIX applications programs to take advantage of the large address space available in the VAX architecture.

3.1. Standard UNIX facilities

The standard version 7 UNIX system has simple memory management facilities. Each process has four areas of memory: a pure code area known as the "text" segment, a private area filled with initialized data values known as the "data" segment, a private area filled with zero known as the "bss" segment, and a stack in its own "stack" segment. Most UNIX implementations provide these four areas using only two base-bounds memory management regions: the text segment is placed before the data and then the bss segment in one region, and the stack in the other.

The only use of shared memory in standard UNIX is the pure code "text" area shared by default among all users. Processes may grow by expanding their stack region when making calls and by allocating stack-local variables, or by allocating more memory beyond the end of the "bss" segment.

3.2. Previous VAX enhancements

The current VAX system pages the regions described in the previous section in a way transparent to application programs. It also demand-loads the initial contents of the pure code "text" and initialized "data" segments, making copies of the pages of the files from which these segments are initialized on first reference.

Facilities are provided in the current system for users to read from files in the copy-on-reference fashion used by the system to set up newly executing programs. This *vread* facility has not, however, proved useful or popular, and it and the *vwrite* and *vadvise* facility will be deleted in the new system and their function replaced by mechanisms described here.

3.3. Goals

A strong motivation for use of the VAX is the large address space available. Each process can have up to 2^30 bytes of data in each of two regions available to it, giving a maximum per-process address space of 2 Gigabytes. To use such a large address space it is necessary to avoid making copies of the data in the space. It is necessary that the system obtain the data from and share it with file data whenever appropriate. Good performance from the system algorithms is necessary if extremely large address space programs are to be run.

The major goals of our memory management facility design are:

- To support the extremely large address spaces possible with the VAX hardware. We would like to be able to run a 2 Gigabyte process address space on machines with as little as 2 Megabytes of physical memory.
- To support shared access to data and the special requirements of the large VAX applications such as image processing and LISP systems. Such programs often need special treatment from the paging algorithms in the system and want to gain control and recover after otherwise fatal errors such as stack overflows and protection violations.

* To have reasonable performance on huge virtual jobs. This will require support from the file system, which must provide high bandwidth access to file data, and support from the user, who can help by organizing his process to have as well-behaved virtual memory behavior as possible, and by giving the system advice about the behavior of his program.
* To develop facilities that are portable to different machines with possibly different memory management architectures. We expect that the demanding nature of research applications will cause them to be run a wide variety of processors, some of which can run this version of UNIX if its primitives are portable.

3.4. Motivations for segments

To achieve the goals described above and manage an extremely large address space, we are basing our memory management design on segment level primitives, not on page level primitives. Segment based facilities seem desirable for at least two major reasons:

+ Programs written using segments can be ported easily to machines that have only page level memory-management control. The VAX is an example; it does not have segmentation, so this will be simulated. Programs written using extensive page-level controls tend to be less portable. Our design thus attempts to encourage a portable programming style.
+ Segments provide a clean structuring of the address space with useful granularity, and offer useful places for placement of instrumentation to gather page-reference information. Memory usage is likely to break down naturally and somewhat independently into usage in different segments.

3.5. Allocating segments

Segments are represented by their base virtual addresses. On a machine with a uniform address space this will just be some number in the address-space-range of the machine. On a machine with segmentation hardware the address will be a (segment,offset) pair.

The basic segment allocation primitive takes as argument a file descriptor and a range of locations in that "file" and returns a virtual address that is the base of the mapped range. The primitive *segalloc* is invoked:

```
int fd; off_t offset; int len;
enum seg_share { SEG_PRIVATE, SEG_SHARED; } share;
caddr_t pref;
caddr_t va;

va = segalloc(fd, offset, len, share, &pref);
```

The argument *fd* specifies the file or special device to be mapped into the address space of the calling process. The arguments *offset* and *len* give the offset into *fd* and count of bytes to be mapped. If *fd* describes a file then its length is made to be at least *offset+len* bytes by extending it with 0 data if necessary.

If *share* is SEG_SHARED addresses to bytes starting at the returned address refer to the contents of the file or device represented by *fd* starting at offset. For shared segments, writing to these bytes is permitted if the file *fd* is available for writing, and is equivalent to writing on the associated file or device.

If *share* is SEG_PRIVATE the returned space refers to private data storage that is initialized from the corresponding file or device data. The virtual

memory returned from a *segalloc* of SEG_PRIVATE space is, by default, readable and writable.

The final argument *pref* may be used to give the address of a variable containing a preferred address for the segment. If the argument is 0 then the system chooses a location for the segment in a way not specified externally. The use of *pref* arguments is machine specific, and is regularly used only by system specific routines and special applications.

3.6. Segment sizes and rounding

Memory management hardware on most machines does not permit exact, bit-length control over how much address space is available to processes. Thus the system does not promise that exactly and only the range [va,va+len) will be accessible after a call to *segalloc* returns a value *va*. There may be some extra locations accessible outside this range, but accessing them should be considered an error. In our proposed VAX implementation, memory will be available to a 1024 byte boundary on both ends of the mapped region for SEG_PRIVATE data, and to a 65536 byte boundary for SEG_SHARED data.

To take advantage of the memory management hardware on a particular machine, the system may have to align the mapped data, e.g. on page boundaries. Because the VAX has no indirect page table entries, and to simplify the system, reduce the amount of work involved in running large programs, and to make sharing of page-table-pages possible, the VAX implementation will align all mapped regions on 65536 byte boundaries so that:

(va & 0xffff) == (offset & 0xffff)

That is, the low 16 bits of the returned address from *segalloc* will agree with the low 16 bits of the offset mapped. This allows the "second-level" page tables of the VAX to be used to achieve page-table-page sharing. As we will see below, page table size for large processes can be substantial, so making page-table-page sharing possible is a desirable goal.

3.7. Segment protections

The default protection mode for a shared segment is inherited from that of the file descriptor *fd*. On the VAX, this must either be *read* or *read-write* since the VAX does not support write-only memory, and users cannot be permitted to map files to be readable simply because they have write access to them.

The protection assigned to a segment may be changed with a *segchmod* call

segchmod(va, mode)

where *mode* is chosen from:

SEG_NA	no access
SEG_R	read access
SEG_W	write access
SEG_X	execute access

The last three accesses may be combined, as in SEG_R|SEG_W to give read-write access. All machines are expected to support SEG_NA, SEG_R, and SEG_R|SEG_W. On machines that do not support execute-only access, SEG_X will be folded to SEG_R access. The VAX has a restriction that SEG_W access is not permitted without SEG_R, since the hardware does not support write-only access.

3.8. Freeing segments

To free the address space occupied by a segment a program can issue the *segfree* call:

segfree(va)

passing the address returned by *segalloc*. The address space previously allocated to the segment is then returned and made available for allocation by future *segalloc* calls.

3.9. Giving the system advice

Large virtual memory programs often have repetitive or predictable behavior. Authors of such programs are often aware of this behavior. We provide a *segadvise* call, of the form:

segadvise(va, advice)

The advice to be given to the system about the segment at *va* is required to have no semantic effect on the result of the program.*

Typical calls to *segadvise* might instruct the system that pre-fetching of a set of pages seem desirable, that the program is finished using a particular section of virtual memory and that it can be reasonably swapped out, or that the program will be referencing many pages quickly with little rereferencing (e.g. LISP garbage collection.) A facility similar to *segadvise* called *vadvise* has been successfully used in the current system.

3.10. Special segments

Calls to allocate segments may access two special files. The first is normally available as **/dev/text**, which is a special device that indirects to yield a handle on the file containing the program that is running. This makes it possible to re-map pages of the running program conveniently.

The other special file is **/dev/zero** which is a special interface to swap space, and that will give a distinct piece of swap space to be initialized with zeros each time it is mapped in.

3.11. How exec can be written

Using the facilities above we can now give code showing how the *exec* system call creates a new process image. First we should explain that process images in the new system will have a 65536 byte hole between the end of each segment and the start of the next. The first 65536 bytes of process address space are not mapped, and serve to catch indirect references through uninitialized pointers.

After this 65536 byte gap comes the beginning of the process image file, starting with the process header and continuing through the process pure "text" space. There is then another 65536 byte gap before the "data" space, another 65536 byte rounding virtual hole, and then the "bss" uninitialized variables.†

The following C code could be used in the system to set up these segments, starting in an empty process virtual memory. The *exec* code here is very VAX

* This excludes timing-dependent programs, whose output may differ from run to run, and may notice the timing improvements obtained when good advice is interpreted properly.
† The virtual holes preserve alignment between the data file and the address space it is mapped to.

specific, and uses macros defined in the system header file <a.out.h>. The symbol SEG_TEXTFD stands for an instance of the file /dev/text, and SEG_ZEROFD stands for an instance of the file /dev/zero.

```
#define SEGRND  65536  /* rounding to segment boundary */

caddr_t pref;

/* allocate program data (text segment) starting at SEGRND */
pref = SEGRND;
segalloc(SEG_TEXTFD, 0, N_TXTOFF(e)+e.a_text, SEG_SHARED, &pref);

/* allocate initialized (data) segment, after text and SEGRND hole */
pref += SEGRND + N_TXTOFF(e) + e.a_text;
segalloc(SEG_TEXTFD, N_DATAOFF(e), N_SYMOFF(e)-N_DATAOFF(e),
   SEG_PRIVATE, &pref)

/* allocate uninitialized (bss) segment, after another hole */
pref += SEGRND + e.a_data;
segalloc(SEG_ZEROFD, 0, e.a_bss, SEG_PRIVATE, &pref);
```

The system would also have to set up the stack for the new process, but this operation is not shown here.

3.12. Simulating copy-on-write

A user program can build a "copy-on-write" like facility at the segment level if the hardware permits restartable instructions, or with more work if it does not. The facility can be implemented by establishing a handler for the "Memory fault" and "Bus error" signals. If a fault then occurs on a protection violation, the signal handling routine will get control. It can modify the accessibility of the referenced data by re-mapping the segment to be modified as SEG_PRIVATE data, and return to the code that was interrupted.

This style of copy-on-write support makes it possible to build copy-on-write like facilities even on machines where instructions are not restartable, provided the code that can fault is written in a way that the user-supplied signal handling routine can backup.

A user program may also monitor both references and modifications to segments by using access modes. For example, after a garbage collection, a LISP system may mark its segments read-only, and make them writable only after a writing is noted. Then, when the next garbage collection is to be done, the system can know that certain sections of address space have not been referenced or modified respectively and avoid garbage collection overhead.

3.13. Special requirements for stacks

Some VAX applications will need to maintain complex stacks. For instance, INTERLISP uses a spaghetti stack and wishes to regain control if the piece of stack being used is exhausted. This requires that the system deliver a signal to the process on a different stack when the first stack overflows.

A similar need arises in languages that support multiple tasks and that provide a fixed size stack per task. If the system were to deliver signals to such a process on the per-task stack, then the size of stack needed would depend on system parameters, an undesirable situation.

To support these applications, we are proposing to extend the system to allow specification of a stack for delivering signals. The call

```
caddr_t asp;
int onsigstack;

sigstack(asp, onsigstack)
```

provides the system with a stack pointer to be used in delivering signals *asp*. The call also informs the system whether the process wishes to consider itself "on" the signal stack, using the integer parameter *onsigstack*.

When a signal is to be dispatched, the system first checks to see if the process is on its signal stack. If not, then the current stack pointer value is saved and the system arranges for it to be restored on return from the signal handling routine. The stack pointer is set to the signal stack location and the kernel remembers that the user process is on the signal stack.

In normal usage, a process will take a signal on the signal stack, run a small amount of code, and then return to the pre-signal frame. The return from the signal handler resets the signal stack automatically. If the process wishes to take a non-local exit from the signal routine, then it must inform the system of the restoration of the signal stack to be performed using a *sigstack* call.

If the process wishes to invoke code from the signal stack that uses a different stack, then the process should provide the system with a new *sigstack* so that signals can be delivered there during the nested invocation; this is necessary because the system would otherwise have no way of finding the top of the signal stack.*

3.14. Huge processes and page table sizes

In running huge processes on the VAX an important concern is the amount of physical memory required for processes that use large amounts of virtual memory. Whether the virtual memory is used or not, it is required to have a certain amount of physical memory allocated to page tables for resident processes.

In the current UNIX system, the kernel keeps all the page tables for resident processes in non-paged memory. Large VAX systems currently see as much as 16 Megabytes of active virtual memory, and since 1 byte of page tables is needed for every 128 bytes of resident virtual memory, this means that as much as 512k bytes of memory is occupied by user page tables. While this is acceptable for running virtual loads of 16 Megabytes, it will certainly not be acceptable when processes as large as a Gigabyte are run, since a Gigabyte process will require 8 Megabytes of page tables.

The new UNIX system on the VAX will consider the address space to be composed of 65536 byte virtual pieces. A single process address space will have 32768 of these pieces, that can be allocated to its various segments. The system will control page table space at this granularity. Only the descriptive information required to locate and manage the page table pages describing the 65536 byte pieces of virtual memory need be resident with a process. It is conservatively estimated that each of these 65536 byte virtual pieces will require 16 bytes of physical memory when the associated process is resident. Thus a Gigabyte process will require roughly a quarter Megabyte of resident information describing these second level page table entries.

* Since, unlike the hardware interrupt stack pointer, the signal stack pointer is not kept in a register separate from the normal stack pointer.

3.15. Page replacement algorithms for VAX

The VAX lacks the reference gathering hardware needed to gather the information used by many page replacement algorithms. This forces the system to use software to gather reference information and makes such information gathering much more expensive. A variant of the clock global replacement algorithm is being used in the current system to do replacement with minimal amounts of reference information, and a good deal of experience with this algorithm has been obtained.

We are experimenting with a special low-level coding of the reference gathering code in the system, which may make the cost of reference gathering several times cheaper. If this works out, then it may be possible to experiment with some other page replacement algorithms.

We have taken traces of programs typical of image processing and other scientific work. Many of the programs that run on large data sets exhibit regular patterns in their virtual memory behavior. The *segadvise* call can be used to inform the system of the presence of such behavior. We hope to experiment with algorithms in the system to detect patterns of behavior and to adapt the page replacement and pre-fetching algorithms accordingly.

In particular, we have already experimented with giving the system advice that a program is sequential, and with advice that a program is likely to have little re-reference to its pages. The former is true of multi-dimensional FFT's running on large data sets, and the latter is true of a LISP system running a large, non-compacting garbage collection. In both cases we observe substantial improvements in running times and reduced overheads in the system because of the advice from the user programs. We expect to experiment with such advice for other large programs.

In the 4.1bsd release of the system we fixed a problem with the placement of pre-paged pages. In the new release, pre-paged pages are placed at the bottom of the "free list", not in the clock loop. This allows us to pre-page more pages, and to use the pre-paged pages more effectively. We have measured the 4.1bsd system on the benchmarks that Dave Kashtan ran of UNIX and VMS paging. The 4.1 system and the VMS measurements are nearly identical for all benchmarks, with the 4.1 system faster on benchmarks that are inherently sequential if the system is told to expect sequential behavior.

3.16. Status and related changes

Implementation of these proposals will proceed in parallel with the higher-performance file system effort (described in section 4), which is currently underway. We expect that we will have a prototype system with a higher-performance file system and the new memory management facilities sometime in late 1981.

There are some related changes that will have to be made to support the new memory management facilities:

+ A new load format will have to be created that allows for the segment placement implied by the new primitives.
+ The debuggers will have to be changed to understand the mappings and the new segmentation.
+ The core file images will have to be changed to include segment data.

+ The file system performance enhancements will need to be in place to take full advantage of the new memory management facilities.

We will use instrumentation facilities already in place in the 4.1bsd system to measure and analyze system performance using the new facilities. We have sample programs that are large VAX applications that will be measured under the new facilities to tune and debug them.

3.17. Alternatives and comparison

We considered using a TENEX "prnap" like facility for controlling virtual memory. Such a facility has been implemented for UNIX on VAX by John Reiser of Bell Laboratories. We decided that the needs of programs could be met without the additional internal complexity of *pmap*, that was felt to be a hindrance when such enormous address spaces are to be supported.

If individual mapping of 512 byte pages were permitted in a 2 Gigabyte address space, then the system would have 4 million pages to deal with for a single process. Thus we went to the 65536 byte granularity in memory management, as this will allow us to handle these gigantic programs even on small machines.

We have considered providing different page-replacement algorithms for the system, including a working-set dispatcher, but feel that the data consumptive nature of the most demanding applications will be satisfied only by algorithms that can be told of or adapt to trends in memory referencing. We feel that the current global replacement algorithm will work adequately in the large process environment and admits the hooks that are needed for exploitation of patterns of reference.

4. File system performance enhancements

This section describes the proposed changes to the file system organization and algorithms to increase performance. We defer discussion of changes to the user interface to the file system to the next section.

4.1. Standard UNIX file system

The traditional UNIX system, that runs on the PDP-11, has simple and elegant file system facilities. File system input/output is buffered by the kernel so that there are no alignment constraints on data transfers and all operations are made to appear synchronous. All transfers to the disk are in 512 byte blocks, which may be placed arbitrarily within the data area of the file system. No constraints other than available space are placed on file growth.

4.2. Previous VAX enhancements

The current VAX system has improved the standard UNIX file system in two notable ways:

* The file system has been made crash-recoverable by changing it so that all modifications of critical information are staged so that they can either be completed or abandoned cleanly by a repair program after a crash.
* The file system performance has been improved by nearly a factor of 2 by changing the basic block size from 512 to 1024 bytes.

4.3. Goals

We expect that large virtual memories will be constructed by mapping files from the file system, using the mechanisms described in the previous section. Paging of data in and out of the file system is likely to occur frequently. We therefore need a file system that provides higher bandwidth than the current one which provides only about 40k bytes per second per arm. The primary means for improving file system performance are to improve the locality of reference to minimize seek latency and to improve the layout of data to make larger data transfers possible.

4.4. Major problems

A typical 150 Megabyte UNIX file system consists of 4 Megabytes of file system indexing information and 146 Megabytes of file system data. A major problem with this organization is that the indexing "inode" information is segregated from the data by being at one end of the disk space allocated to the file system. Thus accessing a file almost certainly involves long seeks. Files in a single directory are not typically allocated slots in consecutive locations in the 4 Megabytes of indexing information, causing many non-consecutive blocks to be accessed in executing common hierarchical operations, such as gathering information about or data from a files in a single directory.

The allocation of data block to files is also a major problem. The current file system never transfers more than 1024 bytes per disk read or write, and often finds that the next sequential data is not on the same cylinder, causing seeks between these 1024 byte transfers. The combination of the small block size, limited read-ahead in the system, and many seeks severly limits file system throughput.

4.5. Description of approach

We propose to reorganize the file system by dividing the space for a file system into areas called *cylinder groups* each of which contains a few cylinders. Each cylinder group will have some inode slots for files and a bit map and other summary information describing the usage of data blocks within that group of cylinders.

Performance will be increased by laying out the hierarchical file system data so that related information is in the same cylinder group, minimizing seek distance. Data will be laid out so that larger blocks can be read in single reads, greatly increasing file system throughput.

As an example a file system of 300000 sectors (150 Megabytes) could be divided into 100 cylinder groups of 1.5 Megabytes each. Each cylinder group would have about 256 inode slots and a bit-map describing availability of its blocks and inodes. The file system data storage would be divided into 4096 byte data blocks. Small files will receive only a fraction of one of these blocks. In large files several 4096 byte blocks could be allocated consecutively so that large data transfers are possible.

4.6. Policies for new file system

The system will provide on-line layout policies that try to limit seeks. Directories, which can be allocated in any of the inode slots, will normally be allocated in the cylinder group that has the most free space, extrapolating a mean size for each of the directories currently in the cylinder groups. File indexing "inode" slots will normally be allocated in the cylinder group where their directories are located; if there is no room there, then they will be allocated using an overflow policy similar to that used in a hash table with internal rehash.

Blocks will be allocated in a device-dependent way. On most devices we prefer to place newly allocated blocks adjacent to the previous block in the same file. If this adjacent block is not available, then the new block will be located rotationally well-positioned on the same cylinder as the previous block. If no blocks are found on the same cylinder as the previous block, then the system will look somewhere else in the same cylinder group. If this also fails to find an allocatable block then the system will look in another cylinder group that has a reasonable amount of space to locate another free set of blocks.

4.7. Measurements of program speeds

To formulate performance goals for the file system it is important to understand the speed of various programs consuming data, and the limiting performance of the current file system organization using differing block sizes. Basic times for operations on the VAX 11/780 with a single memory controller and currently available disk hardware are given in the following table:

Procedure call	20 usec
Examine 512 bytes	110 usec
Trivial system call	140 usec
Copy 512 bytes	220 usec
Context switch	220 usec
Write system call	1 msec
Disk rotation time	16 msec
Seek time	10-50 msec

The limiting overhead in data intensive operations is often the memory bandwidth. When no input/output is taking place data can be fetched from

memory at 4.5 Mb/second, using the VAX string instructions. If any processing is to take place on the data, or if any input/output is taking place on the machine, then the available bandwidth is reduced. Measurements of basic operations and common programs are given in the following table:

Operation	Data rate
Fetch data	4.5 Mb/cpu sec
Fetch with mba active	3.5 Mb/cpu sec
Fetch with 2 mbas active	2.6 Mb/cpu sec
CRC	300 Kb/cpu sec
Loader *ld*	100 Kb/cpu sec
Cat program	42 Kb/cpu sec
egrep program	38 Kb/cpu sec
ed read/write	23 Kb/cpu sec
make of system	22 Kb/cpu sec
fgrep/grep programs	20 Kb/cpu sec
Assembler *as*	15 Kb/cpu sec
Compiler *cc*	10 Kb/cpu sec
Peephole optimizer *c2*	8 Kb/cpu sec
Lisp compiler *liszt*	8 Kb/cpu sec
Troff running *-me* macros	3 Kb/cpu sec

The measurements of fetching of data from memory in blocks show the effect of running high bandwidth devices during memory-intensive cpu operations, where each active i/o device reduces the available bandwidth by about 1 Mb/sec.

The CRC instruction timing shows the speed of a data intensive microcode loop that involves a fair amount of calculation. This program runs at 1/3 the speed of most currently available disks.

The fastest standard UNIX program we could find, aside from the file copying programs, was the UNIX loader. When loading large programs the loader does not process each byte of data individually. This leads to much higher bandwidth than the *cat* program, that is the simplest possible program that uses the character at a time primitives of the standard i/o library. The *cat* program is a loop:

```
int c;
while ((c = getchar()) != NULL)
    putchar(c);
```

The *egrep* program is the fastest example we could find of a program that non-trivially processes all its input data. It is a program for scanning a file for any of a set of patterns, written using a powerful algorithm.

More typical of UNIX utility speeds are the programs *ed*, *make* remaking a large program (the system), the more simple pattern searching programs *fgrep* and *grep*, and the assemblers and compilers *as*, *cc*, *c2*, and *liszt*. These programs range in speed from about 8 to 25 Kb/cpu second on a 11/780. Slowest of all are programs that do substantial processing on each input character, such as the typesetting program *troff*. *Troff* is further slowed by extensive macro interpretation.

4.8. Estimates of file system performance

The observed performance of the constant block size file systems is given in the next table, and extrapolated form the 2048 and 4096 byte block sizes:

Block size	Throughput
512 bytes	20 Kb/sec/arm
1024 bytes	40 Kb/sec/arm
2048 bytes	80 Kb/sec/arm
4096 bytes	160 Kb/sec/arm

We can estimate the performance of our new file system using a basic block size of 4096 bytes and with some pessimistic assumptions about data layout. We assume that the file system will be unable to allocate consecutive 4096 byte blocks, but will be able to place an average of 4 consecutive blocks in a cylinder before a seek is required. We assume that the seek to be required is a long seek. Under these assumptions and in the sequential access case we expect that the new file system will provide 35-40% disk utilization and about 300-350 Kb/sec/arm.

The degree to which this file system organization will improve on the 4096 byte block version of the current file system organization will depend on whether the patterns of file access allow the locality of layout under the new organization to be beneficial. Large applications are expected to benefit greatly if their data requirements have locality. There is little we can do for uncorrelated requests under any organization.

4.9. Buffering and page caching

The current version of UNIX transfers data from the disk into buffers in the kernel address space and then copies these buffers to user address space. If the buffers in both address spaces are properly aligned, then this transfer can be effected without copying using the memory management hardware. This is especially desirable when large amounts of data are to be transferred.

If the buffers in the process address space are properly aligned (on 1024 byte boundaries) we intend to transfer the data to the user programs without copying. Further, even in the absence of copy-on-write, we can remember that pages in user address space are copies of pages from a file and, if the pages are still in core and not modified when we need that file page again, can reuse the page. If the user issues another read request specifying the same buffer we can reclaim unmodified pages from the user and place them in a kernel file system cache.

4.10. Fragmentation in the new organization

In this section, for definiteness, we assume that the desired file system block size is 4096 bytes and that the disk sector size is 512 bytes; these are variables in the file system design, but it is easier to use the numbers for reference.

In UNIX, each file has an array of indices of file system blocks. For the purposes of this section, assume that the first 8 blocks of the file are described to by the basic file indexing (inode) structure.* The inode structure also contains other pointers to indirect blocks containing further block indices. In a file system with a 512 byte basic block size, a singly indirect block contains 128 further block addresses of four bytes each, a double indirect block contains 128

* The actual number may vary from system to system, but is usually in the range 5-13.

addresses of further single indirect blocks, etc.

The following table shows the effect of increasing the file system block size on the amount of wasted space in the file system. The machine measured to obtain these figures was our largest time sharing system, and had roughly 1 Gigabyte of on-line storage. The active user file systems containing roughly 500 Megabytes of formatted space were measured.

Space used	% waste	Organization
421.3 Mb	0.0	Raw data
439.0 Mb	4.2	512 byte rounding of data
450.4 Mb	6.9	512 byte block UNIX file system
470.9 Mb	11.8	1024 byte block UNIX file system
515.5 Mb	22.4	2048 byte block UNIX file system
813.2 Mb	45.6	4096 byte block UNIX file system

Here we measure the space wasted as the percentage of space on the disk not containing file system data, ignoring the fixed amount of space for the inodes. As the block size on the disk is increased, the fragmentation rises quickly, to an intolerable 45.6% waste with 4096 byte file system blocks, since there are so many small files.

To avoid the fragmentation in storing small files, we allow the file system space allocator to divide a single file system block into a few fragments. Our file system block size is 4096 bytes composed of 4 1024 byte fragments, the size of the blocks in the current file system. We allow the space allocator to break up a file system block and allocate these smaller pieces to files.

It suffices to allocate fragments only to file that are less than 8 file system block long (the files that require no indirect blocks). On the system measured above, fully 97% of all files were in this category, and they used about 1/2 of the space in the file systems. Such a small file is represented by up to 7 full file system blocks of data and then possibly some additional data. The full file system blocks are represented in the normal way. If there remains data that will fit in 3 or fewer 1024 byte pieces, we find a unallocated fragment of a file system block and store the data there. If we have to fragment a file system block to obtain the space for this small amount of data, another file may yet use the remaining fragments.

The fragmentation in this organization is less than that the current 1024 byte file system organization, and only slightly more than the 512 byte block UNIX file system: 8.2%. A 512/4096 byte hybrid file system keeps more indexing information, but uses even less space than the 512 byte block traditional UNIX file system: 5.4%. The new organization is efficient because it uses little space for small files and also uses little indexing information.

4.11. Status

We have done a good deal of measurement of the static characteristics of current file systems and examined the dynamic characteristics of applications programs. We have constructed utilities to build file systems in the new format and are working on a user-level implementation of the new file system format. After our development 11/750 arrives in late July 1981, we intend to convert it to the new file system format and to debug the new system algorithms on this machine. Integration of the new memory management facilities described in section 3 will then take place in a system supporting the new file system organization.

4.12. Alternatives and comparison

We considered converting UNIX to an extent based file system much like the DEMOS file system. This approach was rejected because it did not seem necessary to get the performance we needed, and because we expected that some sites might wish to experiment with file organizations that allowed data pages to be shared between files. This is much more easily handled under a block level organization than a extent based organization. Similarly if a copy-on-write facility were ever to be implemented for UNIX it would benefit greatly from a block at a time indexing scheme.

We are planning to compare the performance of this file system with the VMS file system and other file systems for similar machines. The current comparison shows that the UNIX file system is slower than the VMS file system, but we expect that the new version of the UNIX file system will be faster.

5. New file system facilities

This section describes new facilities to be provided by the file system in support of the other facilities proposed in this report and to solve other minor problems.

5.1. Symbolic links

The current UNIX system supports multiple "links" to files in the same file system. This link concept is fundamental; files do not live in directories, but exist separately and are referenced by links. When all the links are removed, the file is deallocated.

This style of links does not support references across physical file systems, nor does it support inter-machine linkage. We propose to include *symbolic links* to support such usage.

A special file type, the "symbolic link" file will contain a pathname. When the system encounters this file while interpreting a name, the contents of the symbolic link file will be prepended to the rest of the pathname, and this name will be interpreted to yield the resulting full pathname. If the symbolic link file contains an absolute pathname, then this absolute pathname will be used. The symbolic link will otherwise be taken starting at the location of the link in the file hierarchy.*

We are currently investigating the best way to implement symbolics in UNIX, looking especially at systems for other machines which implement links (notably MULTICS). Symbolic links have previously been implemented for UNIX by Jim Kulp at IIASA in Austria. To incorporate them he also provided a way for system utilities to refer to the links themselves as well as the object referenced by the links. Incorporating them also involves some changes to utilities such as *du*, *ls*, and *find*, so that they can treat such links in a desirable way. To gain access to the link itself, not the file object referenced by the link, a special quoting convention can be provided. We could say that a file name that ends with the character '#' refer to the symbolic link itself.

It also might be useful to provide a mode in which the system does not interpret symbolic links. Thus a program that wishes to transverse a hierarchy without taking indirections can disable symbolic links.

One set of possible calls for symbolic link routines would be:

```
symlink(name1, name2)
char *name1, *name2;
```

that creates a symbolic link *name2* whose contents are the string *name1*

```
symunlink(name2)
char *name2;
```

that removes the symbolic link *name2* and not the *name1* specified when the link was created. This can also be used with non-symbolic links in a program that wishes to remove the links themselves, not the linked to files.

```
symfollow(wanted)
int wanted;
```

that can be called with 0 or to disable following of symbolic links

* Naming directory references, described in the next section, are considered to be absolute pathnames.

5.2. Naming directories

To support the project notion (to be described in section 6), and as a base for communication between processes in a single session we propose to add a per-process "naming directory". This will be a normal UNIX directory with a very short name "@", a prefix-character to pathnames much like the character "/" which refers to the root directory. It represents a third point in the file file system from which names spring, augmenting the current "current directory" and "root directory" notions. The naming directory concept is derived from the similar one in the Apollo DOMAIN operating system and from the uses of logical name tables in VMS and device translations in various PDP-10 based operating systems.

The naming directory will support the project notion described in section 6. A project is a hierarchy of source and binary programs, library routines and documentation. The proposed normal way of accessing such a hierarchy is to place a symbolic link from your naming directory to the root of the project. Thus the project "vlsi" might have its root directory "/h2/vlsi" in which case one would place a symbolic link named "vlsi" in ones naming directory and henceforth reference the project files as "@vlsi/...".

The naming directory will support screen-oriented command interpreters and front ends through conventions on communication. For instance, the *write* command can be changed to look in the target users naming directory for a file named "writeportal" and to open that file to communicate if it exists. In this way a *write* command can communicate with a screen manager process (such as, say, the CMU *emacs* editor) to obtain window space. This is greatly preferable to the current state where such writes greatly disrupt the state of the screen.

The naming directory implementation is simple: If a path name begins with the character "@" the search begins not at the current directory but at the naming directory. A new system call *chnamdir* changes the current naming directory.

For backwards compatibility, the use of naming directories in support of projects can be simulated by a set of library routines that interpret the UNIX system calls that take file names. Other uses of naming directories to support screen-oriented programming environments are possible only on the newer version of UNIX supporting IPC facilities.

5.3. Locking primitives

Many sites have expressed the desire for some file locking primitives. It is desirable that it be possible to lock files so that other concurrent access can be prohibited to maintain consistency.

The new UNIX 3.0 system from Bell Labs implements a flag to the *open* call that causes a file creation to fail if the file already exists. This allows testing for locks by attempting to create them to work. In the current system, the lock setting has bugs when used by the super-user unless the *link* primitive is used.*

Mike Accetta at CMU has implemented a set of *ioctl* calls that provide file level locking. There are *ioctls* to return a structure giving the count of processes reading and writing the file, to set the file in exclusive write mode, prohibiting further attempts at access to write, to set the file in exclusive update mode, prohibiting further access to read or to write, and to clear the

* I.e. a file is created whose name is unique to the current process and the current process tries to link it to the lock file. The *link* operation is atomic.

exclusive locks.

John Bass at ONYX Systems has implemented granular file locking. This allows sequences of bytes within files to be locked, and detects deadlock conditions. The deadlock detection in Bass's scheme cannot work in a distributed system, and thus we feel that this aspect of the scheme should be avoided. This scheme could yet be implemented by timing out requests.

We are continuing to investigate the form of locking which should be integrated into the kernel of a distributed system. We have so far found no locking primitives which seem suitable.

5.4. Append access and no-delay opens

To atomically append to files, the append mode of access supported by most operating systems has been added to UNIX 3.0. A further *open* option to allow opening of communications lines without waiting for carrier has also been provided. We feel that these facilities are, indeed, useful, and propose to adopt the UNIX 3.0 open mode (as extended by the open locking options described above) into the standard VAX system.

5.5. Truncate

The current UNIX system lacks a primitive to truncate the logical length of a file. This makes implementation of certain FORTRAN 77 facilities expensive. Also, a convenient way of modifying files with mapping is to allocate a segment for them and then write data into the segment, and unmap and truncate the file. This is possible only if there is a system call

```
truncate(name, length);
char *name;
```

that removes portions of the file after the specified length. This can be simulated (albeit slowly) on older UNIX systems as it is currently in the FORTRAN 77 i/o library.

5.6. Rename

Programs that create new versions of data files typically create the new version in another file and then do

```
unlink("cur");
link("new", "cur");
unlink("new");
```

This sequence of operations leaves a window where there is no instance of the file *cur*, causing occasional mysterious anomalies. This can be solved by providing a system primitive:

```
rename(newname, oldname);
char *newname, *oldname;
```

that does what the preceding sequence does, but atomically, so that there is always an instance of *newname*. We propose to add this to the standard version of UNIX.

5.7. Per-file cache flushing

The current system makes no provisions for flushing the file system cache of blocks from a file. This makes it difficult to write application programs that attempt to be certain to leave data bases in a consistent state. We feel that an operation to flush all the buffers associated with a particular file would be

valuable. This will involve remembering, in the buffer cache, which file each buffered block belongs to and also identifying such blocks in the virtual memory of processes. This operation can either be an *ioctl* or a new system call of the form:

```
int fd;

syncfd(fd)
```

5.8. Status

Symbolic links have been implemented for UNIX before and are also implemented in a other operating systems. They require changes to a few programs that are concerned with traversing the file system hierarchy and other than that affect only one routine in the kernel: *nami*.

Naming directories are extremely simple to implement. They will affect a few user programs that use file names beginning with "@" (e.g. Rand's MH program that names a file just that "@"), and a few programs that do detailed manipulation of path names (e.g. "csh" which attempts to figure out what directory you are in after a "chdir" will have to understand the effect of "@").

The *truncate* system call implementation is tricky, since the operation has to be carefully staged so that no duplicate blocks appear if the system crashes during a truncate. The operation is a superset of a *creat* system call, and the code can be combined.

Per-file cache flushing can be added easily when the system is changed over to the new file system organization described in the preceding section.

6. Software projects and distribution support

This section describes a set of changes that extend the conventions for use of UNIX to simplify software interchange. The underlying structure for the proposal was proposed and implemented by Steve Shafer at CMU: the project notion. The proposal defined here integrates the ideas proposed by CMU with some changes based on experience with the project notion at Berkeley. It also include other facilities and standards useful in software support and distribution.

6.1. Current UNIX facilities

Developing large software projects on the current UNIX system requires establishment of conventions for locating parts of the project within the file system hierarchy. Special conventions are often developed per-project, and much "bailing-wire" is needed to hold the project files together. Cries of anguish are often heard if file system hierarchies are moved from disk to disk to alleviate space shortages, as users scurry to convert absolute, and now invalid, path names into new and no more robust names.

With the current system, large software modules to be distributed to other sites often require local customization. Header files have to be edited to reflect true path names where software is or will be stored. It is difficult to install software that is finicky about the locations of commands.

While it is possible for each software effort to develop their own set of conventions and procedures for dealing with this environment, it seems extremely desirable to develop system support and tools for a more robust and portable notion. We will call organized groups of related programs to be managed and ported "projects", following the work done at CMU by Steve Shafer.

6.2. Goals

The goals of this proposal are:

* To support the development of large packages of software by providing a framework for development, based on the framework used by the developers of UNIX.
* To support maintenance of software by adopting conventions for building executable versions of software and for storing the source code and documentation that make this accessible to standard utilities.
* To support distribution of software by making it easy to install software modules in different parts of the file system hierarchy while retaining short and significant names for the various files. Support for co-existence of several versions of a single package (old, current, new, experimental, etc.) for use by different users or at different times is important.

6.3. Components of the proposal

The basis for this proposal is a hierarchy of directories and files called a "project". Projects will be supported by conventional use of the naming directory and symbolic link facilities described in the previous section, which give them mnemonic names, and allow different versions to co-exist with different instances selected by different users. Conventions for makefile's and the use of source revision control facilities will allow reconstruction of the programs in a project to be done automatically and allow information to be obtained that describes the current state or history of any file in a project. Facilities for distribution of notification of changes to projects and automatic update of remote copies of software over networks can be developed based on standard

descriptions of project structure.

6.4. CMU project notion

Following Shafer, we create a UNIX hierarchy for each group of related programs or project. This hierarchy mimics the normal **/usr** file system subdirectories in function and includes directories

bin	containing binaries of project programs.
exp	containing directories for users in the project.
include	containing header file for use in the project.
lib	containing subroutines and shared data files.
man	containing manual entries for project components.
src	containing source code for project commands

Each project also has a normal UNIX group and a bulletin board associated with it. The addition of commands and system facilities to help maintain such hierarchies and the large efforts associated with them is the topic of the rest of this section.

6.5. Strong naming support for projects

There are several important naming requirements for projects:

* It should be easy for users to choose the projects to include in their working environments, and to name files in these hierarchies.
* References from files and libraries in a multi-project environment should clearly denote the projects they are referencing. Thus if a script needs a special version of a standard program, this should be clearly marked in the script.
* Projects should be located in a way that is independent of their absolute placement in the UNIX hierarchy, so that they can be easily transported from machine to machine.

The current CMU project implementation uses search paths, which are part of the UNIX "environment" and interpreted by special library routines, to located commands in projects. This has the problem that the components referenced in source code, scripts and makefiles are not explicit even when exactly one component is desired, and that there are no non-absolute names for project components.

We propose to use the naming directory facility and symbolic links, described in the previous section, to support strong naming for projects. Users would place symbolic links in their naming directories to projects that they wished to use. Thus a entry "vlsi" in my directory on the "ucbarpa" VAX might be a symbolic link to "/ra/vlsi", while someone who was developing a new version of this project might have "vlsi" linked to "/ra/vlsi.new". If each of us ran a program "mkpla" written by a individual that referenced "@vlsi/bin/plot", then we would get the versions of the plot routine that we desired: I would get the current version, while the developer could get the newest experimental version.

This facility is similar in usage to the name table translations on other systems, but since the naming directory is accessible in the UNIX file system it requires much less system mechanism. It is advantageous to put naming support for these directories into the operating system so that it will work in all programs. This provides much stronger support for the project notion.

6.6. Makefile standards

Maintenance and distribution is made substantially easier when all project programs and data bases can be reconstructed by standard *makefile* descriptions. The current system distribution *makefile* descriptions support:

make install	Build a new version of the components in this directory and install them.
make clean	Remove unnecessary binaries from this directory, to minimize space usage.
make	Just make the new components, don't install them.

We propose that all distributed commands should be controlled by makefiles that accept these standard entry points. These constitute a minimum acceptable set of controls for all components. We find the use of these standard makefile entry points preferable to manual operation of commands and manual installation.

6.7. Reviving the UNIX group facility

The UNIX group mechanism is designed to support work among groups of users. Thus all the developers in a project could belong to the same project group. Currently, however, a user may only be in one group at a time and must lose command context when changing groups.

Steve Zimmerman at CCA has implemented a version of the group mechanism that allows users to be in all their groups at the same time. Files created are then placed in the group of the containing directory, not the group of the current user (which is no longer uniquely defined!).

This change enhances the group facility and makes groups much more useful with projects. We propose that in the next version of the system users be allowed to be in multiple groups at a time.

6.8. Source revision control

It is important to have facilities to retain records of old versions of programs and changes made to them. The current CMU project implementation uses a *whist* command to annotate source code with commentary about changes. This is useful, but the inclusion of SCCS-like facilities for control of versions is also needed. Walter Tichy at Purdue is completing a new "Revision Control System" (RCS) which has facilities like SCCS.

We propose that both SCCS and RCS should be integrated into the project mechanism. It should be possible to distribute RCS to all users of UNIX on the VAX; SCCS is less widely available because of licensing constraints. Both SCCS and RCS should be modified to include facilities like the current *whist*.

6.9. Notification/update facilities

A standard method of providing notification of changes to project software is desirable. CMU uses a *post* command that puts messages on bulletin boards, and has software for distribution changes on a local network.

We propose that methods for automatic distribution in large and local nets be developed and be standardized. Methods of notification should be supported by databases associated with mailers and should allow different ways of storing news associated with projects to be used, including:

news	a derivative of the standard *msgs* program, developed at LBL
netnews	a program developed for the *USENET*, a phone network of UNIX systems
post	as used at CMU
mhnews	a news system based on the Rand MH program

It is important that projects be able to retain information about software that has been distributed, and be provided some support for taking bug reports and suggestions (e.g. standard mail boxes for projects at sites where they are installed that can be set up to forward suggestions.)

6.10. Role of unique-identifiers for files

A difficult problem in distributing large software systems is identifying files and making sure that the correct pieces are available for construction of a system. The system can aid this by providing unique identifiers for files that can be preserved when the files are copied from machine to machine. It is also useful for the source code to be stamped with revision numbers to be retrieved by programs such as the *what* program of the current system.

So that systems that maintain software versions can be constructed for a distributed environment we propose that all incarnations of UNIX files be assigned identifiers unique in space and time that can be retained when the files are copied and restored by the source code management utilities when older versions of files are reconstructed. This is not used by any current programs, but current research in automatic construction of distributed software by Eric Schmidt at PARC suggests that such identifiers are valuable. We also propose that a system call be provided to return such a unique identifier.

6.11. Towards site-independent programs

One difficulty with current programs is that they tend to build in site dependencies. A particularly bad example is mail programs that deal with multiple networks, which tend to have a good deal of local knowledge built into them, and hence must be modified and recompiled each time they are moved from cpu to cpu.

It is extremely valuable for programs to be site-independent, and to make system databases available for program inspection at each site to allow site-specific program actions. We propose (in the section on operations below) to make the standard programs in the system more machine independent by making information such as the current system name and network connections, information about users and information about locally available resources available in standard files accessed by library routines. We propose that projects should develop similar site-specific data bases so project binaries and libraries are as cpu-independent as possible.

6.12. Status

A version of projects is running at CMU and on the PDP-11 UNIX systems at Berkeley. We expect to consult with the staff at CMU about the proposal in this section, and to work with both the people at CMU, Walter Tichy at Purdue and the people at CCA to integrate and evaluate the new project proposal.

We propose to provide naming directories and symbolic links soon so that these can be tested with the new project implementation at CMU, CCA and Purdue. We propose to provide the unique-id facility for files with the first release of the new file system organization.

We also propose to work with CMU to develop a new document describing the enhanced notion of projects described here and develop notification and update standards and procedures based on those used at CMU.

7. Standards

This section describes areas of the system where there are nagging problems that will get worse if some attempt at standardization is not made. The problems are not unique to the VAX system – all versions of UNIX could benefit from standardizing on solutions to problems such as those discussed here.

The typical alternative here is to continue with the status quo. This has the advantage of backwards compatibility but will tend to create more problems than it solves in this way. We prefer to adopt clear improvements on the current approaches, getting a simpler and cleaner system in the long run in exchange for some short term revisions.

7.1. Manual format

There are several goals in proposing a new standard for the manuals. There is the obvious desire to keep the manual stable, as the costs of printing the manuals are prohibitively expensive for some. On the other hand, we desire to keep manuals up to date, and quickly include new facilities in the manual.

Our proposal is to define a base system that is represented in the manual and to set up facilities for the additions of sections of project documentation to the manual. The commands *key* and *toc* that CMU implemented as part of their project implementation provide some needed facilities.

CMU also printed abridged manuals by default, treating maintenance commands such as the games as projects. This seems reasonable. A useful form of an abridged manual would include a table of contents for all available documentation, so omitted pages could be run off on line and later obtained separately.

We propose that a new format be adopted with a release of the system in early 1982, with advance notification of the format change. This will allow documentation to be prepared for projects to be distributed with this version of the system. We expect that a preliminary version of the project system can be made available to sites in late 1981 to allow shared software projects and their documentation can be put in a suitable format.

7.2. Libraries

It is important that the contents of the standard libraries contain only a prescribed set of functions so that programs do not have hidden dependencies on locally modified routines. We propose to develop a list of what is in the standard C library and to put new facilities to be added to the ARPA standard system in an ARPA standard library so that the dependencies of newly developed programs on facilities of the ARPA standard system will be explicit.

We feel that it is important to support convenient naming of project specific libraries, and propose that the loader support the project general library notion by taking the form "–l@X" to be the library "@X/lib/libX.a", and the form "–l@X/Y" to be the library "@X/lib/libY.a".

7.3. Mail

UNIX mail is confusing because of the presence of many mailers, mail systems, and network interfaces. Several important new standards need to be handled, such as the new Internet Mail formats, the new Mail transfer protocol, interface of the mail system to UUCP, and to CSNET, etc.

Currently, there are 4 low-level mail handling systems in general use on UNIX:

MMDF Developed at Delaware and that is the basis for Phonenet. This system has a good architecture for mail services. We don't have any experience with using this program but intend to learn more about it soon. It currently does not handle *uucp* traffic.

delivermail Developed at Berkeley, this is a mail routing program that manages mail going to different networks. It can handle the ARPANET, uucp and local network mail simultaneously.

BBN MAIL The new mail system at BBN handles the new MTP protocol, as well as local net mail forwarding.

RAND MH The low level facilities underlying Rands MH system provide groups, aliases and mail transmission facilities.

Each of these programs currently provides facilities provided by none of the others. On the other hand, the programs all provide similar facilities and it is clearly disadvantageous for all four of these systems (and perhaps others) to be developed independently to meet the same needs.

We hope that the persons responsible for these systems will investigate the facilities of the other systems. It would be valuable to standardize on a single mail delivery system, a single format for storing incoming mail, and a single data base format for mail forwarding and mail groups. The many existing mail readers interfaces should be changed to work with the new standard delivery programs. Many of them inadequately process the header information. Fixes for many of these are available in the community (e.g. from CMU and CCA for the Mail program), and should be incorporated as part of the changeover to a new standard mail system.

We intend to pursue the selection of a single standard low-level mail system for the VAX.

7.4. Signals

The signal handling mechanisms of UNIX version 7 are inadequate for safe processing of asynchronous events, having race conditions in them that make them unsafe. Newer mechanisms were provided in the 4bsd release of the VAX system that give clean and safe semantics to signals, treating them as software interrupts that are blocked while they are being processed.

We propose that the newer implementation of the signal handling mechanism be incorporated as the standard one in the VAX system. There are some minor incompatibilities in the way in which interrupted system calls are restarted, but these incompatibilities are felt to be less bothersome than continuing to use a standard implementation of signals that is neither safe to use nor clean.

7.5. Terminal driver interface

The current system supports two different terminal drivers, one that is standard from version 7 UNIX and one a more fully functional terminal driver typical of PDP-10 systems. The new UNIX to be released by Bell Laboratories, UNIX 3.0, has yet another terminal driver interface.

The UNIX 3.0 terminal driver interface is clean, and could be adopted as a standard interface. Programs that wish to use the older version 7 terminal driver interface can use a compatibility interface package.

We propose to provide the facilities of the current new terminal driver and the needs of the INTERLISP implementors for special hooks in the terminal driver with extensions to the UNIX 3.0 driver.

7.6. Control; cleaned up ioctls

The current UNIX *ioctl* system call suffers from a lack of specification of the lengths of the control information being exchanged. We propose to define a new operation that has *ioctl's* semantics but with full parameter specification. This *control* operation will have the form

```
int f;
char *request;
char *idata; int ilen;
char *odata; int olen;
int reslen;

reslen = control(f, request, idata, ilen, odata, olen);
```

Here *f* is a UNIX file descriptor, *request* is a null-terminated string specifying the request, *idata* is a string containing input for the request of length *ilen*, and *odata* provides a place for storing the corresponding result value of maximum length *olen*. The returned *reslen* is the length of the result, which may be shorter than *olen*.

To allow for the easy use of null-terminated strings in *idata*, a *ilen* of −1 will be interpreted by the C library as indicating that *idata* is a null-terminated string.

We believe that this *control* primitive, with its much cleaner interface, will provide a much more stable base for definition of device-specific controls than *ioctl*.

7.7. Debugging information format

The information present in the current symbol table in the UNIX executable files is inadequate for construction of symbolic debuggers. It does not contain enough information about variable types. A new debugger is being written by a student at Berkeley, and is suffering from the lack of this information. We feel it is desirable to have a symbol table format for UNIX that includes adequate information.

We propose to work with other interested parties to define a new symbol table format that permits the representation of all information about the standard languages C, Pascal and FORTRAN 77. It is expected that the ADA implementations for the VAX will require significantly greater complexity in the symbol table information, and we do not propose to handle ADA, although input from ADA implementors would be valuable in defining the new format. The new format should be portable to machines other than the VAX, and should work, for example, also on PDP-11, C/70 and 68000 based UNIX systems. The new debugger will not be constrained by VAX licensing and should be easy to port to work on these machines as well.

7.8. Screen environment support

Programs that wish to build screen oriented command environments are rudely interrupted by current UNIX commands for inter-user communication such as *write*, *wall* and the mail arrival notification daemon. Programs that are to run in windows also need a communications path to a screen manager.

The naming directory can be used by programs such as *write* and *wall* to locate a hook for sending information through a screen manager to the terminal. Conventional hooks could be placed there for processes that wish to communicate to the user "@writeportal", "@mailportal", etc.

We propose to investigate an appropriate set of conventions for these programs to use and to develop these conventions in cooperation with other sites that are working on screen oriented programs. We also propose to investigate providing a facility whereby the messages that the kernel sends to user processes are sent to a place other than the current "/dev/tty". Such messages include messages that tape devices are offline and that file systems are full, and also corrupt the screen of screen managers.

7.9. Other areas

There are undoubtedly other areas where development of new standards and interfaces can benefit the users of UNIX, and we welcome input about and proposals for such standards.

8. Operational support

This sections discusses needs for operation support of the system, including file system backup and retrieval procedures and error logging.

8.1. Standard UNIX facilities

The standard UNIX/32V system provides dump and restore procedures for file system backup and accounting gathering for login time and process resource usage. The system must be manually rebooted after a crash and manual procedures instituted to reconstructed any file systems that are damaged. The standard system does not handle bad media and does not record error messages that are printed on the console.

8.2. Current VAX facilities

The VAX system has been enhanced substantially from the standard version 7 UNIX system. A new installation and setup guide exists for the VAX system that clearly explains the operational procedures. The dump program has access to a table describing how often file systems should be backed up, and it is thus much easier to tell when the file systems need to be backed up.

The system automatically restarts after a crash, and runs an automatic repair program. The system performs critical disk operations in a careful way, doing some disk operations synchronously so that the post-crash repair program can either finish or back out each incomplete operation.

A simple description file describes each VAX CPU and can be used to load a system containing exactly those drivers required. The system supports multiple instances of all standard devices and placement of devices on multiple MASSBUS or UNIBUS adapters. Full ECC recovery and DEC standard bad block handling on disks are supported. System sizing is simplified by automatic extrapolation of needed table size from a constant "maximum active users" in the description file.

Error messages printed on the system console are in a more readable format than those printed by standard version 7. They are saved in a buffer in memory that is retrieved after a system crash and stored in a disk file for later examination. Device error bits are decoded symbolically in the error messages.

8.3. Overview of needs

Most operational needs are addressed in the current version of the system. Remaining needs include support for an "operator", who can execute maintenance functions but with less privilege than the "super-user", clean localization of site-specific information to make the system binaries more portable, standard error logging for quicker repair, improvements to the dump/restore program and provisions for user archival and retrieval of files to relieve pressure on disk space.

8.4. Operator notion

In the current standard system, a person who is to do such maintenance operations as file system dumps and restores is required to have super-user privileges, allowing unrestricted access to all system facilities. This is undesirable on many systems, and several sites have implemented a notion of an "operator" with maintenance privileges but not all privileges.

We feel that this notion is a useful one. We propose to integrate the changes made at CMU for the support of an operator into the standard system.

8.5. Clean localization of system

The standard system has commands that have to be recompiled per-site because they contain site-dependent information. There should be a standard provision and use of the information needed by these programs so that they could be site-independent. The programs are typically part of the mail system or have compiled machine names into them.

The information about users on the system is also not cleanly parameterized. Some systems put information about users into the GECOS field of the password file, but this seems less than desirable. We propose to develop a standard form for a user information file. Any such data base should be extensible, and contain, at minimum, the information accessed by the current *finger* command.

Other system information such as the terminal type databases currently exists in several files because of the evolutionary path by which these files were developed. We propose to compress this information into single data bases where appropriate to make maintenance of this information simpler.

Finally, we propose to add a new standard directory **/local**, which on each system will contain all the local files and databases. Databases that currently exist in other directories with long-term associations, such as **/etc/passwd** will be replaced by symbolic links to their counterparts in **/local**.

8.6. Error logging

The current UNIX system does not produce error log information in a format that DEC field service is used to. More seriously, the system does not log recovered soft errors, so that impending problems can go undiscovered when evidence of their onset would otherwise be available. At least one site (UCLA) had many problems with their VAX that might have been avoided or alleviated if full error logging were available.

Don Markuson at CMU is working on an implementation of error logging in UNIX. He is cooperating with the UNIX group at DEC, which previously produced a system written by Fred Cantor called "v6m", which was a modified version 6 UNIX that supported error logging.

8.7. Dump/restore needs

The *dump* and *restore* programs have been modified by CMU and Wisconsin to do multiple dumps per tape and to restore hierarchies respectively. We feel that these modifications should be combined and incorporated back into the standard system.

8.8. Archive/retrieve design

UNIX sorely needs a system whereby users can request portions of the file system hierarchy be safely archived on tape, so that they can later request them be restored. Our group at Berkeley is working on two programs, *archive* and *retrieve* that will meet this need.

The *archive* command will take a list of file names and queue them for archival. When the files are archived, an entry will be made in a data base associated with the user noting information that can later be used to retrieve the file, and the user will receive mail notification that the archival has taken place.

A *retrieve* command will queue a request for file retrieval from an archive tape. The file will later be retrieved when a extraction program is run by an operator.

So that users may have confidence in the archive/retrieve procedure, we intend that an option be available to make multiple copies (normally 2) of each archive tape, and that this procedure result in tapes which are stored in separate locations. Provision of manpower to make the turnaround time on *archive* and *retrieve* requests sufficiently low to encourage use should be paid back in lowered disk space usage.

We intend that *archive* and *retrieve* will store files on magnetic tape in *tar* format and maintain an on-line database of the files that have been stored.

9. Miscellaneous topics

This concluding section contains discussion of several topics of general interest that didn't fit naturally in any of the other sections.

9.1. Software census and contribution to standard system

We are currently preparing to mail questionnaires to all users of the VAX system asking them to tell us the software they have brought up on the VAX that they are willing to share with the general VAX community. We hope to take the information gathered by this "VAX software census" and place it in an on-line data base. We hope that this information will eventually be available through CSNET for general examination and update by authorized users.

We are also interested in finding out what software efforts are going on. Our questionnaire will ask both what kinds of software are being developed and what software the different sites are interested in porting to UNIX. We hope that this procedure will make us aware of the software that is available, and help us to tell what software should be made available in a standard system.

9.2. Electronic forum for system users

We are interested in creating an electronic forum for users of the VAX UNIX system. The forum "unix-wizards@sri-unix" has proven a useful information exchange for a limitied set of VAX users. We plan to establish a forum for ARPA users of the VAX UNIX system as soon as our NCP C/70 is firmly on the ARPANET.

An electronic mailbox "csvax.4bsd-bugs@berkeley", available via uucp as "ucbvax!4bsd-bugs", has been available for about 6 months, although only a few sites have been submitting trouble reports.* We hope to advertise this more widely, mentioning it in the questionnaires. Another mail box "csvax.4bsd-ideas@berkeley" collects ideas for improvements to the system. Some of the proposals discussed in this report benefited from suggestions mailed to "4bsd-ideas."

9.3. Hardware support; new and dual processors

The VAX UNIX system supports all released DEC hardware for the VAX except the TU78 tape transport. We are working with the UNIX group within DEC to provide support for new DEC devices and VAX processors as they are released.

Bob Kridle, of the Systems Support Group at U.C. Berkeley and Bill Joy have prepared a document giving hints to UNIX users on Configuration of VAX systems. This document has helped many sites bring up economical VAX installations.

Recently, George Goble and Mike Marsh of the Electrical Engineering department at Purdue University have created a dual processor 11/780 UNIX system, by cabling an additional VAX processor to an 11/780 SBI. While some minor problems remain with running compatibility mode on the slave processor, the system is functional.

Since current VAX 11/780 systems are limited in growth largely by the available CPU power, this appears to be an attractive way to get nearly twice the CPU horsepower of a single processor system for much less additional cost. Addition of a CPU and a second memory controller to a single CPU VAX 11/780 system, and provision of 4 Megabytes for the second CPU should be possible for

* Official requests from ARPA contractors related to the VAX UNIX system should be mailed to "csrg@berkeley".

under $100,000. With some help from DEC it would be possible to run this configuration in shops where more processor power is needed at a lower cost than replication of entire systems.

We have been working with George Goble and Mike Marsh to develop reasonable processor scheduling algorithms for the dual 11/780. We intend to encourage DEC to provide support for this option and assist us in fixing minor problems with this configuration.

9.4. Debuggers

The VAX UNIX system currently comes with two debuggers: *adb* and *sdb*. The *adb* debugger is oriented towards examination of memory and object code, and currently has no knowledge of source text. The *sdb* debugger knows about source code, but suffers from several minor bugs and lack of information in the symbol table needed to do proper handling of displayed variable values. *Sdb* also does not contain an expression parser powerful enough to accept source language expressions.

Robert Elz of the University of Melbourne has extended *adb* to provide some programmability. We have worked with Rob Gurwitz at BBN to provide *adb* with knowledge to interpret the VAX page tables and to make it more useful for debugging the UNIX kernel. We have recently made some minor modifications to *adb* so that it records the source line information used by *sdb* when present in the object file, and hence can show source as well as object code. We intend to fix the display of local variables in *adb* and make this improved debugger available to other sites in the next release.

A source language symbolic debugger was written for the Pascal interpreter *px* on the PDP-11 by Mark Linton at Berkeley. This debugger is currently being moved to the VAX and made to work for C, Pascal and FORTRAN 77 code. We hope that this debugger will be part of a future release of the system.

9.5. Fortran 77

There are many sites that would like to use UNIX on their VAX systems but have need of a fast FORTRAN implementation. While the *f77* compiler is a complete implementation of the language, the speed of compiled code produced by the compiler is noticeably less than that produced by the VMS FORTRAN compiler. This is not surprising. The *f77* compiler is not an optimizing compiler, while the VMS FORTRAN compiler is.

Stuart Feldman, one of the authors of *f77*, visited Berkeley last academic year and formed a group to work on optimizations in *f77*. This group is now in the process of implementing the designed optimization pass of the compiler, and hopes to have a prototype of the new compiler running by the end of the year. We have funding to hire a programmer to work on *f77* next year to finish this project.

We also hope to incorporate the improvements made to FORTRAN by Jim Kulp at IIASA in Austria. Kulp's group produced documentation designed to help users of FORTRAN on other machines learn to use *f77*. We hope to work with the Computer Center at Berkeley to make the documentation produced in Austria more widely available.

A group of students at Berkeley under the direction of Prof. Kahan are producing basic math library routines such as *sin* and *sqrt* to conform to the new IEEE standards. These routines will be integrated into the standard VAX UNIX math library as they become available.

9.6. Detaching jobs

We realize the desirability of detaching jobs from one terminal and reattaching them to another terminal. This facility was considered for inclusion when the job control facilities were added to UNIX but rejected because of the difficulty of communicating the change of environment to the newly attached jobs. If conventions for use of the naming directory as a communications area are adequate, this problem can be solved. Jobs that are reattached could look in their naming directory to see the terminal type they are now attached to and discover other aspects of their environment. We plan to investigate the provision of *attach* and *detach* facilities in future releases of the system.

9.7. UNIX and VMS: performance and facilities

There has been a good deal of discussion of the relative performance of UNIX and VMS. Much of the available information is now out of date, and more will be outdated as the facilities described here are incorporated into UNIX and new versions of VMS become available.

Our recent measurements show that the differences in paging performance of UNIX and VMS reported by Kashtan at SRI are no longer significant. We measured the behavior of the 4.1bsd system running his benchmarks and got times not significantly different from the times he reported for VMS. When we used *vadvise* to tell the system that the sequential access jobs were, indeed, sequential, then the system outperformed VMS substantially.

In our experience the largest reason for research sites to use VMS is the quality of the FORTRAN on VMS. We hope that a future release of a better *f77* compiler will make the FORTRAN issue moot, so that the choice need not be made for FORTRAN alone.

We expect to run a new set of UNIX and VMS benchmarks after the facilities described here are in place in the system, probably sometime early in 1982. The results of these benchmarks should prove valuable for further refinements to the systems.

I. Index and summary of proposed system facilities

The following table summarizes the new system facilities proposed in this paper. The entries in the table are system calls (whose names are all in lower case), constants related to system calls (whose names are all upper case), and new types associated with the new facilities (which are given in italics). Each item is classified as relating to memory management facilities *mman*, IPC and networking *ipc*, the file system *filsys*, or general needs *general*. Other categories include *changed* for system calls whose interface is changed, or *deleted* for system calls to be deleted.

Name	Kind	See	Description
answer	ipc	2.5	Receive a call establishing virtual circuit
associate	ipc	2.8	Provides a server for a network address
asynchronous	general	2.10	Request interrupt notification about i/o
call	ipc	2.5	Place a call establishing virtual circuit
chnamdir	general	5.2	Change naming directory
closesend	ipc	2.7.5	Close transmit half of a circuit
control	general	7.6	Replacement for *ioctl* with cleaner interface
disassociate	ipc	2.8	Remove an association from *associate*
ENBLOCK	general	2.10	Error returned instead of blocking with *nonblocking*
fd_set	general	2.3	Type representing set of file descriptors
fd_waterm	general	2.11	Type representing watermarks for i/o
in_addr	ipc	2.3	Internetwork address type
in_proto	ipc	2.3	Socket type, from SOCK_DG, SOCK_VC, SOCK_CALL
ioctl	deleted	7.6	To be replaced by *control* with cleaner interface
nonblocking	general	2.10	I/o requests return ENBLOCK instead of blocking
open	changed	5.3	New flags from UNIX 3.0 and for locking
portal	ipc	2.7	Create a server gateway in UNIX file system
portal_kind	ipc	2.7	Portal types defining protocols
PORTAL_CALL	ipc	2.7	Portal type for simple circuit connections
PORTAL_FILE	ipc	2.7	Portal type for file emulation
PORTAL_DEV	ipc	2.7	Portal type for device emulation
PORTAL_DIR	ipc	2.7	Portal type for directory emulation
receive	ipc	2.4	Receive a datagram
recordbetween	ipc	2.9.1	Is a circuit between records?
recordmode	ipc	2.9.1	Place circuit in record mode
rename	general	5.6	Atomic rename primitive for file system
segadvise	mman	3.9	Give system advice about a segment
segalloc	mman	3.5	Allocate a segment in virtual memory
segchmod	mman	3.7	Change access protection of a segment
segfree	mman	3.8	Free a segment in virtual memory
SEG_SHARED	mman	3.5	Segment is to be shared (in *segalloc*)
SEG_PRIVATE	mman	3.5	Segment is private (in *segalloc*)
SEG_NA	mman	3.5	No access allowed in segment (in *segchmod*)
SEG_R	mman	3.7	Read access allowed in segment (in *segchmod*)
SEG_W	mman	3.7	Write access allowed in segment (in *segchmod*)
SEG_X	mman	3.7	Execute access allowed in segment (in *segchmod*)
select	general	2.6	Provides a synchronous i/o multiplexing facility
send	ipc	2.4	Send a datagram
signal	changed	7.4	New signal facility to become standard
sigstack	general	3.13	Provide special stack for signal processing
SIGIO	general	2.10	Input/output possible signal (with *asynchronous*)
SIGURG	ipc	2.9.2	Urgent data arrival signal
socket	ipc	2.4	Create a socket for IPC communications

Name	Kind	See	Description
socketstatus	ipc	2.11	Return internal state of a socket
SOCK_CALL	ipc	2.3	Call director socket for establishing circuits
SOCK_DG	ipc	2.3	Datagram socket type
SOCK_VC	ipc	2.3	Virtual circuit socket type
symlink	filsys	5.1	Create a symbolic link
symunlink	filsys	5.1	Remove a symbolic link
symfollow	filsys	5.1	Enable/disable symbolic links
syncfd	general	5.7	Flush buffering associated with file or device
truncate	filsys	5.5	Shorten the length of a file
urgentmode	ipc	2.9.2	Place circuit in urgent data mode
urgentnext	ipc	2.9.2	Is next data in circuit urgent?
urgentpending	ipc	2.9.2	Is there any upcoming urgent data?
urgentsockets	ipc	2.9.2	Return set of sockets with urgent data pending
vadvise	deleted	3.1	Replaced by ***segadvise*** facilities
vread	deleted	3.1	Replaced by ***segalloc*** facilities
vwrite	deleted	3.1	Replaced by ***segalloc*** facilities
watermarks	general	2.12	Set buffering watermarks for stream descriptor
@	general	5.2	Naming directory filename prefix character