ARACHNE USER GUIDE

Version 1.2

by

Raphael Finkel
Marvin Solomon
Ron Tischler

Computer Sciences Technical Report 379

February 1980

ARACHNE USER GUIDE*

Version 1.2
February 1980

Raphael Finkel
Marvin Solomon
Ron Tischler

Abstract

Arachne is a multi-computer operating system running on a network of LSI-11 computers at the University of Wisconsin. This document describes Arachne from the viewpoint of a user or a writer of user-level programs. All system service calls and library routines are described in detail. In addition, the command-line interpreter and terminal input conventions are discussed. Companion reports describe the purposes and concepts underlying the Arachne project and give detailed accounts of the Arachne utility kernel and utility processes.

----
*This research was supported in part by the United States Army under contract #DAAG29-75-C-0024. We have been forced to change the name of the Roscoe distributed operating system, since Roscoe is a registered trademark of Applied Data Research, Incorporated. The new name we have chosen is Arachne; the operating system and research continue unchanged.

TABLE OF CONTENTS

ARACHNE USER GUIDE

1. INTRODUCTION

Arachne is an experimental operating system for controlling a network of microcomputers. It is currently implemented on a network of five Digital Equipment Corporation LSI-11 computers connected by medium-speed lines.* The essential features of Roscoe are:

1. All processors are identical. Similarly, all processors run the same operating system kernel. However, they may differ in the peripheral units connected to them.

2. No memory is shared between processors. All communication involves messages explicitly passed between physically connected processors.

3. No assumptions are made about the topology of interconnection except that the network is connected (that is, there is a path between each pair of processors). The connecting hardware is assumed to be sufficiently fast that concurrent processes can cooperate in performing tasks.

----

*This equipment was purchased with funds from National Science Foundation Research Grant #MCS77-08968.

4. The network appears to the user to be a single powerful machine. A process runs on one machine, but communicating processes have no need to know if they are on the same processor and no way of finding out. (Migration of processes to improve performance is transparent to the processes involved.)

5. The network is constructed entirely from hardware components commercially available at the time of construction (January, 1978).

6. The software is all functional. Although Roscoe has undergone much revision, it has been working for over a year.

1.1 Purpose of this Document

This document describes Arachne from the point of view of a user or user-programmer. It is both a tutorial and a reference guide to the facilities provided to the user. All information necessary to the programmer of applications programs should be found here.

Further discussion of the concepts and goals of Arachne are discussed in [Solomon 78, 79]. That document also lists some research problems that the Arachne project intends to investigate. The operating system kernel that provides the facilities listed below is described in considerable detail in [Finkel 78, 80b]. Similar detailed documentation about utility processes (such as the File System Process, the Teletype Driver, the Command Interpreter, and the Resource Manager) is contained in [Finkel 79a, 79b].

Arachne has been developed with extensive use of the UNIX operating system [Ritchie 74]. All code (with the exception of a small amount of assembly language) is written in the C programming language [Kernighan 78]. The reader of this document is assumed to be familiar with both UNIX and C.

A new programming language called Elmer, is being designed for applications programs under Arachne; it will be described in a future report. Arachne programs may be written in either Elmer or C. Currently, the library is available only in C.

1.2 Caveat

Arachne is in a state of rapid flux. Therefore, many of the details described in this Guide are likely to change. The reader who intends to write Arachne programs should check with one of the authors of this report for updates.

1.3 Format of this Guide

Section 2 provides an overview of the concepts and facilities of Arachne. Section 3 describes the facilities by name, arranged according to general subject areas. Section 4 is a programmer's reference manual. Each function is listed alphabetically, its syntax and purpose are described, and it is classified as a service call (an invocation of an operating system kernel routine) or a library routine (a procedure linked into the user program). Section 5 describes the command line interpreter and lists the commands that may be entered from the terminal. Section 6 describes the conventions governing terminal input/output. Section 7 presents protocols for communicating with the various utility processes.

1.4 Revisions

The following changes have been made to Arachne since version 1.0 of this document:

There is a new service call, "linkok", to determine if a link number is currently valid. The library routine "call" uses this service call to avoid sending a message across a bad link.

Messages now include length information. The library routines "call" and "recall" have been modified to reflect this change. The file and terminal protocols have also been simplified.

The following changes have been made since version 1.1 of this document:

A new utility process, the *pipe*, is now available. Pipes allow the output of one user process to be attached to the input of another.

The structure "usmesg" has been abolished, and "urmesg" no longer contains the body of the message. Instead, both "send" and "receive" have a new argument that specifies the message body.

A new link restriction, MAYERROR, is orthogonal to all other restrictions. The last argument to send may have the ERROR bit on, in which case the message is considered an error report if the link across which it is sent has MAYERROR specified. Receipt of an error report raises an exception.

The "die" service call now takes a character-string argument. This argument becomes the body of any DESTROYED message that is generated due to the termination of the calling process.

When a process dies, error reports are sent along any links that it holds with restriction MAYERROR but not TELLDEST.

Many errors caused by service calls raise exceptions. An exception can only occur during a service call. If it is not caught, the guilty process terminates. Exceptions may be caught with the "errhandler" service call.

A new facility for asynchronous message receipt, called "catch", allows a procedure to be specified that will be invoked as soon as a message arrives on the specified channels.

The "display" kernel call returns timing information about the owner of any link.

We have been forced to change the name of the Roscoe distributed operating system, since Roscoe is a registered trademark of Applied Data Research, Incorporated. The new name we have chosen is Arachne; the operating system and research continue unchanged.

2. ROSCOE CONCEPTS AND FACILITIES

The fundamental entities in Arachne are: _files_, _programs_, _core images_, _processes_, _links_, and _messages_. The first four of these are roughly equivalent to similar concepts in other operating systems; the concepts of links and messages are idiomatic to Arachne. A file is a sequence of characters on disk. Each file has directory information giving the time of last modification and restrictions on reading, writing, and execution. The contents of a file may contain header information that further identifies it as an executable program. Version 1 of Arachne uses the UNIX file system; therefore, the reader familiar with UNIX should have no problem understanding Arachne files.

Program files contain _text_ (machine instructions), _initialized data_, and a specification of the size of the uninitialized global data space (_bss_) required by the program. Program files also contain relocation information and an optional symbol table.

2.1 _Processes_

A _process_ is a locus of activity executing a program. Each process is associated with a local data area called its _stack_. A program that never modifies its global initialized or bss data but only its local (stack) data is _re-entrant_, and may be shared by several processes without conflict. A main-storage area containing the text of a program, its initialized data, and a bss data area, but not including a stack, is called a _core image_. C core images may not share text areas unless they are reentrant; the text and data areas of Elmer programs are loaded separately, so Elmer programs may share text even if they are not reentrant. The initiation of a process entails locating or creating (by _loading_) a core image, allocating a stack, and initializing the necessary tables to record its state of execution. Similarly, when a process _dies_, its tables are finalized and its stack space is reclaimed. If no other processes are executing in its core image, then the space occupied by the core image is available for re-use.

2.2 Links

All communication is performed by message passing across links. A link combines the concepts of a communications path and a "capability." A link represents a logical one-way connection between two processes, and should not be confused with a line, which is a physical connection between two processors. The link concept is central to Arachne. It is inspired and heavily influenced by the concept of the same name in the Demos operating system for the Cray-1 computer [Baskett 77]. Each link connects two processes: the holder, which may send messages over the link, and the owner, which receives them. The holder may duplicate the link or give it to another process, subject to restrictions associated with the link itself. (See "Link restrictions" below.) The owner of a link, on the other hand, never changes.

Links are created by their owners. When a link is created, the creator specifies a code and a channel. The kernel automatically tags each incoming message with the code and channel of the link over which it was sent. Channels are used by a process to partition the links it owns into subsets: When a process wants to receive a message, it specifies a set of channels. Only a message coming over a link corresponding to one of the specified channels is eligible for reception. A link is named by its holder by a small positive integer called a link number, which is an index into a table of currently-held links maintained by the kernel for the holder. All information about a link is stored in this table. (No information about a link is stored in the tables of the owner.)

2.3 Messages

A message may be sent by the holder to the owner of a link.

A message may contain, in addition to MSLEN (currently 40) characters of text, an enclosed link. The sender of the message specifies the link number of a link it currently holds. The kernel adds an entry to the link table of the destination process and gives its link number to the recipient of the message. In this way, the recipient becomes the holder of the enclosed link. If the original link is not destroyed, the sender and the recipient hold identical copies of the link.

2.4 Link restrictions

Links may be created with various restrictions. These can be characterized as modes, permissions, and notifications. The orthogonal modes are REQUEST and REPLY. A reply link is distinguished by the fact that it can only be used once; it is destroyed when a message is sent over it. A reply link may not be the enclosed link in a message sent over another reply link. Similarly, a request link cannot be sent over a request link. These restrictions enforce a communication protocol in which all communications between two processes connected by a REQUEST link are initiated by the holder of that link.

Two permissions are GIVEALL and DUPALL, controlling distribution of the affected link to other parties. A third permission is MAYERROR, which allows the holder to send an error message, whose receipt will raise an exception.

The notifications are TELLGIVE, TELLDUP, and TELLDEST. When these restrictions are in force, unforgeable messages are sent to the owner of the link when it is given away, duplicated, or destroyed. (The last of these messages contains a body provided by the holder if it dies holding the link.)

2.5 Service calls

The Arachne kernel is a module that resides identically on all the machines of the network and provides various services for user programs. The services are requested by means of *service calls*, which appear to the caller to be procedure invocations.

The chief function of the kernel is to support link maintenance and message passing by providing service calls to create and destroy links and send, receive and catch messages. Additional service calls create and destroy processes, read and set "wall-clock" and high-resolution interval timers, specify a handler to catch exceptions, and establish interrupt handlers for processes that control peripheral devices.

2.6 Utility processes

Arachne has been designed so that as many as possible of the traditional operating system functions are provided not by the kernel, but by ordinary processes. These *utility processes* may invoke service calls not intended to be used by the casual user, but otherwise they behave exactly like user processes. The *terminal driver* is an example. One terminal driver resides on each processor that has a terminal. All terminal input/output by other processes is requested by messages to this process. It understands and responds to most commands accepted by a file (see below), as well as a few extra ones, such as "set modes" (e.g., echo/no echo, hard copy/soft copy).

A _file manager_ process has access to the Arachne file system, currently implemented on the supporting PDP-11/40. A request to open a file sent to any file manager process causes a link to be created representing the open file. To the user of a file, the open file behaves like a process that understands and responds to messages requesting read and write operations. The file is closed by destroying the link. A version of the file manager that uses a floppy disk instead of the PDP-11/40 file system is also available; it follows the same protocols as the other file manager.

The most complex utility process is the _resource manager_ (RM). Resource managers reside on all processors and are connected by a network of links. A process can request an RM to create a new process. The RM may create the process on its own machine or relay the request to another RM based on such considerations as location of the process that requested the creation, availability of free memory, proximity of resources such as devices and files, and the possibility that the required program is already in memory.

The new process is started with a link to its RM, over which it can request links to the process that requested its creation, to a file manager process, to a terminal driver, or to other resources. The RM can kill the process, or it can give a special link to another process (usually a terminal driver) that may be used to kill it.

2.7 Library routines

Functions provided by service calls are rather primitive, and communication with utility processes can involve complicated protocols. An extensive library of routines has been provided to simplify writing of programs that use service calls and utility processes. These routines serve to hide the communication necessary to accomplish various tasks, and make it especially easy to introduce software not originally designed for the Arachne environment. These routines can only be used with C programs; the Elmer library is under construction.

3. SUBJECT-AREA GUIDE

This section lists service calls and library routines by subject area.

3.1 Links and Messages

A new link is created by a process through the "link" service call. Initially, the creator is both holder and owner of the link. The creator specifies what channel and code to associate with the link, so that future messages arriving along it can be selectively received and identified. In addition, the creator may place restrictions on the use of the link, controlling whether or not it may be given to third parties, duplicated, or used repeatedly, and requiring notifications to be sent along it in the event of link duplication, transferral, or destruction. Finally, links may specify that they can carry error messages. Receipt of an error message terminates the recipient.

Messages are sent with the "send" service call, which specifies a link over which the message is to be sent, the message text and an optional enclosed link. It also indicates if the message is an error message.

Messages are accepted by "receive," which specifies a set of channels, a place to put the message, and a maximum time the recipient is willing to wait. "Receive" can also be used to sleep a specified period of time by waiting for a message that will never arrive. Asynchronous message receipt is accomplished by "catch", which has the same arguments as receive, except it has no wait time, and it specifies a procedure to call when an appropriate message arrives. This catcher procedure is very limited in the kernel calls it can perform.

A simple send-receive protocol is embodied in the library functions "call" and "recall," which are simpler to use than send and receive, and should be adequate for most routine communication. The "call" library routine sends a message along a given link, enclosing a reply link. It then waits five seconds for a response, which it returns to the caller. If no answer has arrived in five seconds, it returns failure, and the "recall" routine can be invoked to continue waiting for the tardy response.

3.2 Processes

A process may spawn others by communicating with the resource manager; typical cases are handled by the library routine "fork". The requestor indicates whether the child should be run as a foreground, background, or detached job. Foreground processes are attached to a terminal and can be terminated by entry of a control-C. Background processes may only be terminated by requesting the resource manager to remove them, which is accomplished by the library routine "killoff". Detached processes cannot be terminated except at their own request. The caller also indicates whether the child process may share its core image with other processes, whether an old and inactive core image may be used, or whether a fresh core image is required.

Every user process is started holding link number 0, whose destination is the resource manager on that process's machine. When calling "fork", the parent may indicate a link that it wishes to give to the child; the child obtains this link with the library routine "parline", which communicates with the resource manager along link 0. A process can terminate itself by calling "die"; it can yield the CPU to another process by the service call "nice". (Scheduling is not pre-emptive.)

Four low-level process-control service calls are provided for the use of the resource manager; they are not intended for the typical user. The service call "load" arranges for bringing new core images into the processor on which the caller resides. If there is no room, the call returns failure, and the resource manager can try to find a neighboring resource manager that might have better luck. Once a core image is loaded, processes can be started in it with the service call "startup", which provides the new process with an initial link 0 of the caller's choosing. The "kill" service call removes a process, and "remove" reclaims its core image. The separation of images and processes allows one core image to be used simultaneously be several processes, and a core image may be saved after the last process is gone to speed up the next invocation of a process that would use it.

3.3 Timing

Arachne has two notions of time. One is the wall clock, which keeps track of seconds in real time. Messages sent between resource managers are routinely used to keep the various machines synchronized. There is also an interval timer, which may be used to monitor elapsed time in increments of ten-thousandths of seconds. No process may change the interval timer.

The wall clock is referenced, changed, enciphered, and deciphered by "date", "setdate", "datetol", and "ltodate", respectively. The interval timer is referenced by "time". The percentage of time used by any process may be discovered with "display".

3.4 Interrupts and Exceptions

User programs may handle their own interrupts. This feature is currently used by the terminal driver. A process may establish an interrupt-level routine with the "handler" service call. This call names not only the interrupt handling routine and which interrupt it is intended to service, but also a channel along which to receive messages sent by that interrupt routine. The interrupt-level routine should, of course, be thoroughly debugged and fast. Interrupt-level routines may notify the process that established them by the service call "awaken". This call causes a special message to be sent to the master routine along the channel it specified in its "handler" call. Since the master and interrupt-level routines share code and data, all details of the communication are embedded in shared variables; the awaken message itself is empty.

If a processes arranges for asynchronous receipt of messages by using a "catch" service call (see "Links and Messages" above), then arrival of such a caught message will not preempt any other process. However if the catching process is currently executing, control will immediately switch to the catcher routine within the process.

Exceptions are raised by many service all errors (usually poorly formed service calls) and by receipt of error messages. Usually, exceptions cause the termination of the offending process. Exceptions may be caught by establishing a handler with the "errhandler" service call. When an exception arises, the handler will be invoked with arguments indicating the value returned from the failed service call, the service call number, and all the arguments to the service call. Return from the handler acts like return from the service call.

3.5 Input/Output

To use files, a process first obtains a link to the file manager process by calling the library routine "fsline", which communicates with the local resource manager. This link is used in subsequent library routine calls: "open" and "create" make new files or ready old ones for reading or writing, and return links to be used for manipulations of those files. The library routines "read", "write", and "seek" act much like the Unix file primitives of the same name to provide random access into the open file. A file is closed by the library routine "close", which is identical to the service call "destroy", which destroys a link. Finally, the library routine "stat" returns various information about the open file. Each of these library routines packages a request in a message that is sent across the file access link to the file manager process.

To use the terminal, a process obtains input and output links by calling the library routines "inline" and "outline", respectively, which communicate with the local resource manager. An input link can be used to discover or change terminal modes (only the command interpreter uses this feature) and to perform terminal input. An output link can be used for terminal output. These links may also be "closed"; they are closed automatically when a process dies. The terminal driver allows at most one input link to be open at a time.

Reading is performed by the library routines "read" and "readline". Writing is performed by "write" and, if formatting is desired, by "print". Each of these routines works equally well in dealing with a file instead of the terminal. The service call "printf" is identical to "print" except that it always uses the terminal; it is a debugging tool not intended for the typical user.

The user familiar with UNIX is cautioned against assuming that any particular buffer size is particularly efficient for reads or writes, because Arachne splits up I/O into packets of size MSLEN bytes anyway.

3.6 Miscellaneous Routines

The following routines from the C library also exist in the Arachne library: atoi, long arithmetic routines, reset, setexit, strcopy, streq, strge, strgt, strle, strlen, strlt, strne, and substr.

An additional routine supplied by Arachne is "copy".

3.7 Preparing User Programs

User programs for Arachne are written in the C programming language. They are compiled under UNIX on the PDP-11/40 and should include the files "user.h" and "util.h" in directory /usr/network/roscoe/user. Source programs should have filenames ending with ".u". To prepare a file named "foo.u", execute "makeuser foo", which creates an executable file for Arachne named "foo". The executable files are always stored in /usr/network/roscoe/user.

4. ROSCOE PROGRAMMER'S MANUAL

The following is an alphabetized list of all the Arachne service calls and library routines. For each service call error result, the notation "(*)" indicates that the error causes an exception to be raised.

4.1 Alias (Library Routine)

```
int alias(fslink,fname1,fname2) char *fname1, *fname2;
```

The new name "fname2" is associated with file "fname1". The argument "fslink" is the caller's link to the file manager. The old name is still valid. Possible errors: The combined length of "fname1" and "fname2" must not exceed MSLEN-6. The name "fname2" must not already be in use. File "fname1" must exist. All errors return -1.

4.2 Awaken (Service Call)

```
awaken()
```

Only an interrupt level routine may use this call. It sends a message to the process that performed the corresponding "handler" call along the channel specified by that "handler" call.

Returned values: Success returns a value of 0. -2 is returned if the message cannot be sent because no buffers are available; an "awaken" may succeed later.

4.3 Call (Library Routine)

```
int call(ulink,outmess,inmess,outlen,inlen)
      char *outmess,*inmess; int *inlen;
```

This routine sends a message to another process and receives a reply. The link over which the message is sent is "ulink", which should be a REQUEST link. The argument "outmess" points to the message body to be sent, of size "outlen". (Caution: "outlen" should include the terminating null, if the message is a string.) Similarly, "inmess" points to where the reply body will be put. The length of the reply will be placed in the integer pointed to by "inlen"; if the user doesn't need this feature, "inlen" may be set to 0. If "inmess" is 0, any reply will be discarded. An error is reported if the reply does not arrive in five seconds (see "recall"). In normal cases, the return value is the link enclosed in the return message; it is -1 if there isn't any enclosure. Ignoring errors, the user may consider this routine an abbreviation for:

```
      struct urmesg urmess;
      send(ulink,link(0,CHAN16,REPLY),outmess,outlen,NODUP);
      receive(CHAN16,inmess,&urmess,5);
      if (inlen) *inlen = urmess.urlength;
      return(urmess.urlnenc);
```

Returned values: Under normal circumstances, the return value is either -1 or a link number. -2 means an error occurred while sending, -3 means the waiting time expired, -4 means that the return link was destroyed, -5 means that something was received with the wrong code, meaning that the user program is also using CHAN16 for some other purpose, -6 means that a return link couldn't be created in the first place, -7 means that the ulink was bad.

NOTE: CHAN16 is implicitly used; for this reason, the user is advised to avoid this channel entirely. Several other library routines also invoke "call", and thus use CHAN16.

4.4 Catch (Service Call)

```
int catch(chans,data,urmess,catcher) char *data, int catcher();

struct urmesg {            /* for receiving messages */
      int urcode;   /* chosen by user, see "link" */
      int urnote;   /* filled in by Arachne, see "receive" */
      int urchan;   /* chosen by user, see "link" */
      int urlnenc;  /* index of enclosed link */
      int urlength; /* length of incoming message */
} *urmess;
```

The arguments are the same as for the receive service call, except for the last one. The procedure specified by "catcher" is activated as an asynchronous message recipient for messages that appear on the channels indicated. If a catcher is active on some channel, then any message that arrives on that channel will cause the asynchronous invocation of the catcher, which takes no arguments. The message itself is placed in "data" and "urmess" in the same way as for "receive".

The catcher procedure may inspect the message and modify global variables; it may not invoke any service calls except "printf". If the catcher returns FALSE, it will be deactivated from the channel across which the message came; if it returns

TRUE, it remains active.

If a catcher has already been activated for some channels, and a new "catch" call names other channels, then the union of all the channels active before and now indicated will be activated for catchers. There is only one catcher procedure, one "data", and one "urmess" at any time; subsequent "catch" calls can replace these values with new ones.

If "catcher" is 0, then instead of activating the given channels, they are deactivated with respect to catching messages. All channels not mentioned in "chans" are unaffected. The "data" and "urmess" arguments are ignored in this case.

If the destination of a message is both waiting to receive it and has a catcher activated to catch it, the message is given to the catcher, not the receive call. Catching a message prevents it from also being received.

Messages are caught in the order in which they arrive at the destination.

Returned values: 0 is returned on success. -1 (*) means the argument "catcher" was bad, -2 (*) means "urmess" or "data" was bad. In this case, the other specified channels may or may not get catchers.

4.5 <u>Close</u> (<u>Library Routine</u>)

```
int close(file)
```

The argument "file" is either a link to an open file, or a terminal input or output link. The returned value is 0 on success, negative on failure (actually, "close" is synonymous with "destroy"). These links are automatically closed when a process dies; however, execution of this command gives the caller more room in its link table. Also, closing the terminal input makes it possible for another process to open it.

4.6 Copy (Service Call)

int copy(link);

This service call returns a copy of the given link. If the link is restricted, copy may fail or cause a notification to be sent. Returned values: 0 for success, -1 (*), -2 (*) if the original link number is out of range or not in use, -3 (*) if the link is protected against duplication, -4 (*) if there is no room in the user's link table for a new link. (See "link".)

4.7 Create (Library Routine)

int create(fslink,fname,mode) char *fname;

If the file named "fname" exists, it is opened for writing and truncated to zero length. If it doesn't exist, it is created and opened for writing. The argument "fslink" is the caller's link to the file manager. The protection bits for the new file are specified by "mode"; these bits have the same meaning as for UNIX files, but all files on Arachne have the same owner. The returned values are as in "open".

4.8 Date (Service Call)

long date();

This service call returns the value of the wall clock, which is a long integer representing the number of seconds since midnight, Jan 1, 1973, CDT.

4.9 Datetol (Library Routine)

long datetol(s) char s[12];

This library routine converts a character array with format "yymmddhhmmss" into a long integer, representing the number of seconds since midnight (00:00:00) Jan 1, 1973. It accepts dates up to 991231235959 (end of 1999); -1 is returned on error.

4.10 Destroy (Service Call)

int destroy(ulink)

Link number "ulink" is removed from the caller's link table.

Returned values: 0 is returned on success. -1 (*) means that the link number is out of range, -2 (*) means that it is an invalid link, and -3 (*) means the link may not be destroyed (link 0 has this property).

4.11 Die (Service Call)

die(mesg) char *mesg;

This call terminates the caller. All links held by the caller are destroyed. As these links are destroyed, DESTROYED messages are sent along all links that have the TELLDEST restriction; these messages contain "mesg" as the body (always MSLEN bytes), unless "mesg" is 0. Error messages are sent along all links that have the MAYERROR restriction but not TELLDEST.

Various errors can cause a "die" to be automatically generated. Here are the possible contents of "mesg":

```
bad die message
bad trap
exception not caught
killed
fell through
core image damaged
```

4.12 Display (Service Call)

```
int display(link);
```

This call returns a number in the range 0 to 100 that represents the percentage of CPU time used by the owner of the given link averaged over the last 4 seconds. 0 means that the process has not run at all; 100 means that the process has been active the entire time.

Returned values: -1 (*) if the link points to a different machine, -2 (*) if the link number is invalid.

4.13 Errhandler (Service Call)

```
char *errhandler(addr) char *addr;
```

A new exception handler is established to catch exceptions raised during service calls and receipt of error messages. The handler is a routine at location "addr". The old handler address is returned. A 0 value for "addr" disables exception catching.

If not caught, exceptions cause the termination of the offending process. When an exception arises, the handler will be invoked with these arguments: the value returned from the failed service call, the service call number, and all the arguments to the service call. Return from the handler acts like return from the service call. To ignore exceptions, use a handler that only returns its first argument.

Returned values: -1 (*) if "addr" is unreasonable, the address of the old handler (possibly 0) otherwise.

4.14 Fork (Library Routine)

```
int fork(fname,arg,mode) char *fname;
```

The resource manager starts a new process running the program found in the file named "fname", which must be in executable load format. The function named "main" is called with the integer argument "arg". "Mode" is a combination (logical "or") of the following flags, defined in "user.h":

one of these:	FOREGROUND, BACKGROUND, or DETACHED
and one of these:	SHARE, REUSE, EXCLUSIVE, or VIRGIN

If FOREGROUND is specified, then the new process can be killed by entering a control-C on the console. FOREGROUND is mainly used by the command interpreter. If BACKGROUND is specified, then a "process identifier" is returned that may be used to subsequently "killoff" the child. DETACHED (i.e., neither FOREGROUND nor BACKGROUND) is the default. If SHARE is specified, then the resource manager will be willing to start this new process in the same code space as another process executing the same file, if that process was also spawned in SHARE mode. If REUSE is specified, the code space of an earlier process can be reused. If EXCLUSIVE is specified, then this process may not be started on a machine which already has a process using the same executable file. VIRGIN means that a new copy must be loaded, and is the default. If the call succeeds, a link of type REQUEST and TELLDEST is given to the resource manager; the child may obtain this link by invoking "parline". The caller may receive messages from the child over this link, which has code 0 and channel CHAN14.

A returned value of -1 indicates an error. Success is indicated by a return value of 0, except in the case of BACKGROUND mode, when the return value is a "process identifier".

4.15 Fsline (Library Routine)

```
int fsline();
```

This routine returns the number of a REQUEST link to be used for communication with the file manager Process. An error gives a returned value of -1.

4.16 Handler (Service Call)

```
handler(vector,func,chan) (*func)();
```

The address of a device vector in low core is specified by "vector". The interrupt vector is initialized so that when an interrupt occurs, the specified routine "func" is called at interrupt level. If the interrupt level routine performs an "awaken" call, a message will arrive on channel "chan" with urcode 0 and urnote "INTERRUPT" (see "receive").

Returned values: Success returns a value of 0. -1 (*)

means that there have been too many handler calls on that machine (the limit is currently 2). -2 (*) means that the channel is invalid. -3 (*) means that the vector address is unreasonable. -4 (*) means that the vector is already in use.

4.17 Inline (Library Routine)

int inline();

This routine returns the number of a REQUEST link to be used for subsequent terminal input. The terminal driver only allows one input link to be open at any time. An error returns a value of -1.

4.18 Kill (Service Call)

kill(lifeline);

The process indicated by "lifeline" (the return value of a successful "startup" call) is terminated as if it had performed "die("killed")". The lifeline is not destroyed.

Returned values: Success returns a value of 0. -1 (*) indicates that the link is invalid or not a "lifeline".

Only the resource manager and terminal driver should use this call.

4.19 Killoff (Library Routine)

int killoff(procid);

This routine asks the resource manager to kill a process that the calling process previously created as a BACKGROUND process with a "fork" request. The value returned from that "fork" is "procid". The effect on the dead process is as if it had called "die".

0 is returned for success, -1 for failure.

4.20 Link (Service Call)

int link(code,chan,restr)

A new link is created. The caller becomes the new link's owner (forever) and holder (usually not for very long). The caller specifies an integer, "code", which is later useful to the caller to associate incoming messages with that link. The caller also specifies "chan" as one of sixteen possibilities, CHAN1, ..., CHAN16, which are integers containing exactly one non-zero bit. Channels are used to receive messages selectively. CHAN16 should be avoided, for reasons explained in "call". CHAN15 should also be avoided, since the kernel uses it for remote loading. The returned value is the link number that the caller should use to refer to the link. The argument "restr" is the sum of various restriction bits that tell what kind of link it is. The possibilities are:

```
GIVEALL
DUPALL
TELLGIVE
TELLDUP
TELLDEST
REQUEST
REPLY
MAYERROR
```

"GIVEALL" means that any holder may give the link to someone else. "DUPALL" means that any holder may duplicate it (i.e., give it to someone with "dup" = DUP; see "send"). "TELLGIVE", "TELLDUP", and/or "TELLDEST" cause the owner to be notified whenever a holder gives away, duplicates, and/or destroys the link, respectively (see "receive"). A process may duplicate, give away, or destroy a newly created link without restriction and without generating notifications; restrictions and notifications only apply to links received in messages. A link must be either of type "REQUEST" or "REPLY". A REPLY link cannot be duplicated and disappears after one use; a REQUEST link can be used repeatedly unless it is destroyed by its holder. An enclosed link must always be of the opposite type from the link over which it is being sent. If "MAYERROR" is specified, then error messages may be sent along this link. (See "send" and "receive".)

Returned values: The normal return value is a non-negative link number. -1 (*) means that the link was specified as either both or neither of REPLY and REQUEST; -2 (*) means that the channel is invalid, -3 (*) means there is no room for a new link (currently 20 links are allowed to each process).

4.21 Linkok (Service Call)

int linkok(link)

The returned value is 0 if the link number is currently valid, -1 if it is out of range, and -2 if it is in range but does not denote a valid link.

4.22 Load (Service Call)

int load(prog,fd,plink,arg) char *prog;

This call loads a program. If "fd" is -1, the console operator is requested to load "prog" manually. If "fd" is a valid link number (it should be a link to an open file) and "prog" is -1, the file is loaded on the same machine. In either of these cases, the return value is an "image", to be used for subsequent "startup" or "remove" calls.

If "fd" is a link and "prog" is a machine number, the file is loaded remotely on the corresponding machine and started. The arguments "plink" and "arg" have the same meaning as in the "startup" call. The "plink" is automatically given (not duplicated). The return value is a "lifeline", as for a "startup" call.

Returned values: A nonnegative image number or lifeline number is returned on success. -2 (*) and -3 (*) mean that the link "fd" was out of range or was invalid, respectively. -5 means that there wasn't room for the new image. -6 means that there are too many images. -10 (*) means that the caller had no room for the lifeline. -11 (*) means that the "plink" was out of range or had an invalid destination.

Only the resource manager should use this call.

4.23 Ltodate (Library Routine)

```
ltodate(n,s) long n; char s[30];
```

This library routine converts a long integer, representing the number of seconds since Jan 1, 1973, into a readable character string telling the time, day of the week, and date. Dates later than 1999 are not converted correctly.

4.24 Nice (Service Call)

```
nice()
```

This call allows the Arachne scheduler to run any other runnable process. (Arachne has a round-robin non-pre-emptive scheduling discipline; "nice" puts the currently running process at the bottom.) It is used to avoid busy waits.

4.25 Open (Library Routine)

```
int open(fslink,fname,mode) char *fname;
```

The file named "fname" is opened for reading if "mode" is 0, for writing if "mode" is 1, and for both if "mode" is 2. The argument "fslink" is the caller's link to the file manager. The returned value is a link number, used for subsequent "read", "write", and "close" operations. This link may be given to other processes, but not duplicated. -1 is returned on error.

4.26 Outline (Library Routine)

int outline();

This routine returns the number of a link to be used for subsequent terminal output. An error returns a value of -1.

4.27 Parline (Library Routine)

parline();

This routine asks the resource manager for a link to the parent of the caller. It assumes that the parent gave the resource manager a REQUEST link when it spawned the child. An error returns a value of -1.

This call is typically used by a program being run by the command interpreter; the parent link (to the command interpreter) is used to get the command line arguments.

4.28 Print (Library Routine)

int print(file,format,args...) char *format;

This routine implements a simplified version of UNIX's "printf". The argument "file" is either a link to an open file or a terminal output link. The input is formatted and then "write" is called. The "format" is a character string to be written, except that two-byte sequences beginning with "%" are treated specially. "%d", "%o", "%c", "%w", and "%s" stand for decimal, octal, character, long integer, and string format, respectively. As these codes are encountered in the format, successive "args" are written in the indicated manner. (Unlike "printf", there are no field widths.) A "%" followed by any character other than the above possibilities disappears, so "%%" is written out as "%". Only 6 arguments are allowed.

4.29 Read (Library Routine)

int read(file,buf,size) char *buf;

The argument "file" is either a link to an open file or a terminal input link. At most "size" bytes are read into the buffer "buf"; fewer are read if end-of-file occurs. For the terminal, control-D is interpreted as end-of-file. The returned value is the number of bytes actually read.

4.3Ø Readline (Library Routine)

int readline(file,buf,size) char *buf;

This routine is the same as "read", except that it also stops at the end of a line. For a file a "newline" character is interpreted as end-of-line; however, "readline" is very inefficient for files. For the terminal, a "line-feed" or "carriage return" terminates a line; the last character placed in the buffer will be "newline" (octal 12). Control-D or control-W will also terminate a line, but they will not be included in the bytes read. The returned value is the number of bytes read.

4.31 Recall (Library Routine)

```
int recall(inmess,inlen) char *inmess; int *inlen;
```

If a previous "call" (or "recall") returned a value of -3, meaning that the message did not arrive in 5 seconds, a process can invoke the library routine "recall" to continue waiting. Only the return message buffer and place to store the length are specified (cf. "call").

Returned values: These are the same as for "call", except that -2 and -6 don't apply.

4.32 Receive (Service Call)

```
int receive(chans,data,urmess,delay) char *data;

struct urmesg {            /* for receiving messages */
      int urcode;   /* chosen by user, see "link" */
      int urnote;   /* filled in by Arachne, see "receive" */
      int urchan;   /* chosen by user, see "link" */
      int urlnenc;  /* index of enclosed link */
      int urlength; /* length of incoming message */
} *urmess;
```

The caller waits until a message arrives on one of several channels, the sum of which is specified by "chans". All other messages remain queued for later receipt. The code and channel of the link for the incoming message are returned in "urcode" and "urchan", respectively.

The value of "urnote" is one of six possibilities: DUPPED, DESTROYED, GIVEN, INTERRUPT, DATA, or ERROR. The first three of these mean that the link's holder has either duplicated, destroyed, or given away the link (see "send" and "link"). In the case of "DESTROYED", the body of the message may contain data placed there during termination of the sender (see "die" and "kill"). "INTERRUPT" is discussed under "handler". "DATA" means that the message was sent by "send".

"ERROR" means either that the message was sent by "send", but the link had "MAYERROR" and the sender specified "ERROR", or the link had "MAYERROR" but not "TELLDEST" and the holder terminated (see "link"). Receipt of an error message raises an exception (see "errhandler").

The newly assigned link number for the link enclosed with the message is reported in "urlnenc"; the caller now holds this link). If no link was enclosed, "urlnenc" is -1. The length of the incoming message is reported in "urlength". The argument "data" must point to a buffer of size MSLEN into which the incoming message, if any, will be put. The caller may discard the message by setting "data" to zero. The argument "delay" gives the time in seconds that the caller is willing to wait for a message on the given channels; a "delay" of 0 means that the call will return immediately if no message is already there, and a "delay" of -1 means that there is no limit on how long the caller will wait. A process can sleep for a certain amount of time by waiting for a message that it knows won't come (e.g., on an unused channel).

Returned values: 0 is returned on success. -1 (*) means the caller has no room for the enclosed link (see link; the message can be successfully received later), -2 (*) means that the argument "urmess" was bad, -3 means that the waiting time expired.

4.33 Remove (Service Call)

remove(image)

The code segment indicated by "image", the return value of a successful "load" call, is removed. Only the process that performed a "load" is allowed to subsequently "remove" that image.

Returned values: Success returns a value of 0. -1 (*) means that the image either doesn't exist or is in use, or that the caller didn't originally load the image.

The resource manager uses this call to create space for new images; no other program should use this call.

4.34 Seek (Library Routine)

int seek(file,offset,mode)

The argument "file" is a link to an open file. The current position in the file is changed as specified by the "offset" and "mode". A value for "mode" of 0, 1, or 2 refers to the beginning, the current position, or the end of the file, respectively. The "offset" is measured from the position indicated by "mode"; it is unsigned if "mode" = 0, otherwise signed. A returned value of 0 indicates success, -1 indicates failure.

4.35 Send (Service call)

int send(ulink,elink,data,length,dup) char *data;

This call sends a message along link number "ulink". The message body is "data" and its length is "length". If no message is to be sent, either "data" or "length" should be zero. If the caller wishes to pass another link that it holds with the message, it specifies that link's number in "elink" (the "enclosed link"). If there is no enclosure, "elink" should be -1. The use of elinks is restricted in various ways; see "link".

The argument "dup" specifies either "DUP" or "NODUP"; in the first case, the enclosed link is duplicated so that both the sender and receiver will hold links to the same owner; in the second case, the enclosed link is given away so that only the receiver of the message will hold it.

The "dup" argument also may specify "ERROR" (this bit should be ored into "DUP" or "NODUP"). If "ulink" has the "MAYERROR" restriction, then an "ERROR" message will be sent to the recipient. If "MAYERROR" is not set, then "ERROR" has no effect.

Returned values: 0 is returned on success. -1 (*) means that the ulink number is bad and -2 (*) means that the ulink is invalid. -3 (*) and -4 (*) have corresponding meanings for the elink. -5 (*) means that the message was bad, -6 (*) means that the elink can't be duplicated, -7 (*) means that the elink can't be given away, and -8 (*) means the message is too long.

No error is reported if the destination process has terminated; in this case, the message is discarded.

4.36 <u>Setdate</u> (<u>Service Call</u>)

setdate(n) long n;

This service call sets the wall clock to "n", which is a long integer representing the number of seconds since midnight, Jan 1, 1973.

Only the command interpreter and resource manager should use this call.

4.37 Startup (Service Call)

int startup(image,arg,plink,dup,fd)

This call starts a process whose code segment is indicated by "image", the return value of a successful "load" call. The child is given "arg" as its argument to "main". The child's link number 0 is "plink", a link owned by the caller; this link is either given to the child or duplicated depending on whether "dup" is NODUP or DUP, respectively. The child cannot destroy link 0. For C programs, the data area is part of the image; for Elmer programs, "startup" causes a new data area to be created. The "fd" argument should be a link number for an open file that holds the Elmer program; it is used to load the data segment.

Returned values: Success returns a non-negative lifeline number, which can be used for a subsequent "kill". -1 (*) means that the caller had no room for the lifeline (see "link"). -2 (*) or -3 (*) means that the "plink" was out of range or had an invalid destination, respectively. -4 means that there was no room for the new process' stack (or data area: Elmer only). -5 (*) means that the "image" was invalid, -6 (*) means that "image" is an Elmer program, and "fd" is bad.

Only the resource manager should use this call.

4.38 Stat (Library Routine)

int stat(fslink,fname,statbuf) char statbuf[36];

This library routine gives information about the file named "fname". The argument "fslink" is the caller's link to the file manager. An error returns a value of -1. After a successful call, the contents of the 36-byte buffer "statbuf" have the following meaning:

```
struct{
      char     minor;          minor device of i-node
      char     major;          major device
      int inumber;
      int      flags;
      char     nlinks;         number of links to file
      char     uid;            user ID of owner
      char     gid;            group ID of owner
      char     size0;          high byte of 24-bit size
      int size1;          low word of 24-bit size
      int addr[8];        block numbers or device number
      long     actime;         time of last access
      long     modtime;   time of last modification
      } *buf;
```

NOTE:
Some of these fields are irrelevant, since all Arachne files have the same owner.

4.39 Time (Service Call)

long time();

This service call returns a long integer that may be used for timing studies. The integer is a measure of time in intervals of ten-thousandths of seconds. NOTE: The time wraps around after a full double word (32 bits).

4.40 Unlink (Library Routine)

```
int unlink(fslink,fname) char *fname;
```

This library routine removes the file named "fname"; it cleans up after "create" and "alias". The argument "fslink" is the caller's link to the file manager. Error returns a value of -1.

4.41 Write (Library Routine)

```
write(file,buf,size) char *buf;
```

The argument "file" is either a link to an open file or a terminal output link. Using this link, "size" bytes are written from the buffer "buf". There are no return values.

5. CONSOLE COMMANDS

The Command Interpreter is a utility process that reads the teletype. When the Command Interpreter is awaiting a command, it types the prompt ".". A command consists of a sequence of "arguments" separated by spaces. Otherwise, spaces and tabs are ignored except when included in quotation marks ("). Within quotes, two consecutive quotes denote one quote; otherwise, quotation marks are deleted. The first "argument" is interpreted as a "command" (see below). Command names may be truncated, provided the result is unambiguous. It is intended that all commands will differ in their first three characters.

The "run" command may be followed by from one to MAXCOMS (4) commands separated by the symbol "^". The terminal output of the command to the left of a "^" is buffered by a special "pipe" process, and fed as though it were terminal input to the command to the right of the "^". The output from the last process in a pipe may be redirected to a file by following it with " ^ to outfilename". The input to the first process in a pipe may be obtained from a file by preceding it with "from infilename ^ ". Although "to" and "from" appear to be the names of processes in the pipe, they do not count towards the MAXCOMS maximum. Furthermore, "to" and "from" are reserved words to the command interpreter, and hence neither may be the name of a user program.

The following is an alphabetized list of console commands.

5.1 alias <filename1> <filename2>

The second indicated file becomes another name for the first indicated file. If either of these is "deleted", the other (logical) copy still exists; however, changes to either affect both.

5.2 background <filename> <arg>

The indicated file must be executable. It is started as a BACKGROUND process, with the integer argument "arg". The Command Interpreter prints out the new process's process identifier, which may be used for subsequent "killing" and then gives the next prompt.

5.3 copy <filename1> <filename2>

The second indicated file is created with a copy of the contents of the first indicated file.

5.4 delete <filename>

The indicated file is deleted.

5.5 dump <address>

Prints a screenful of memory locations in octal for debugging.

5.6 help

A list of available commands is displayed.

5.7 kill <arg>

The indicated argument should be the process identifier returned from a previous "background" command. The process referred to by the process identifier is killed.

5.8 make <filename>

The named file is created. Subsequent input is inserted into the file; the input is terminated by a control-D.

5.9 rename <oldname> <newname>

The name of file "oldname" is changed to "newname".

5.10 run <filename> { <arg> } { ^ <filename> { <arg> } }

The indicated files should be executable files. The right-most one is run as a FOREGROUND process. The others are run as BACKGROUND processes. The Resource Manager is given a REQUEST link, which the new process may use to ask for the command line arguments. When the loaded program starts up, the argument to "main" tells the number of command line arguments. To get the individual arguments, the loaded program sends a message to the Command Interpreter (its parent). The first word of the message is ARGREQ, and the second is an integer specifying which argument is desired. The name of the program is argument number 0. The returned message body is the argument, which is a null-terminated string of length at most MSLEN.

5.11 set <modelist> or SET <modelist>

This command changes the console input modes. The mode list is a sequence of keywords "x" or "-x", where "x" can be any of the following:

upper	(the terminal is upper case)
echo	(the terminal echoes input)
hard	(the terminal is hard-copy)
tabs	(the terminal has hardware tabs)

Keywords may be abbreviated according to the same rules as commands. The format "x" turns on the corresponding mode, "-x" turns it off. (UPPER is recognized for upper; "lower" means "-upper".) For more information, see the section "CONSOLE INPUT PROTOCOLS".

5.12 <u>time</u> <<u>format</u>>

If a format is given (as "yymmddhhmm"), the wall clock is set to that time and printed. With no argument, "time" prints the wall clock time.

5.13 <u>type</u> <<u>filename</u>>

The indicated file is typed.

6. TERMINAL INPUT PROTOCOLS

The terminal driver performs interrupt-driven I/O, which allows for typing ahead. Also, the following characters have special meanings:

Control-C	kill the running program (but don't kill the command interpreter itself)
Control-D	end of file (terminates a "read" or "readline")
Control-W	end of line (but no character sent)
line-feed	end of line
carriage return	end of line
rubout	erase last character (unless line empty)
Control-X	erase current line
Control-S	enter scroll mode; pause every 18 lines of

output; if paused, allow the next 18 lines to be printed.
    Control-Q   leave scroll mode; if paused, allow output to continue.
    escape   next character should be sent as is

In "echo" mode, input is echoed, otherwise not. In "hard" mode, output is designed to be legible on hardcopy devices; otherwise the terminal driver assumes that the cursor can move backward, as on a CRT. In "tabs" mode, advantage is taken of hardware tabs on the terminal. In "upper" mode, the terminal is assumed to only have upper case. Input is converted to lower case, unless escaped. Upper case characters are printed and echoed with a preceding "!". Escaped [,], @, ^, and \ are converted to {, }, `, ~, and |, respectively, and the latter are similarly indicated by preceding "!"s.

7. UTILITY PROCESS PROTOCOLS

This section describes the protocols that user programs must follow to communicate with the utility processes when the library routines described earlier are inadequate. Four utility processes are the resource manager, the file manager, the terminal driver, and the command interpreter. The resource manager keeps track of which programs are loaded and/or running on the local machine. The kernel and the resource manager reside on each machine. The terminal driver governs I/O on the console; the command interpreter interprets console input. The file manager implements a file system by communicating with the PDP-

11/40. It need not exist on every machine.

During Arachne initialization, one resource manager is started. It loads a full complement of utility processes (the terminal driver, command interpreter, and file manager) on its machine and various utility processes on the other machines. When a particular resource manager is not given a local terminal driver or file manager, it shares the one on the initial machine.

7.1 Input/Output Protocols

This section describes the message formats used for communicating with the file manager and terminal driver processes. A program that explicitly communicates with the file manager or terminal driver must include the header files "filesys.h" and "ttdriver.h", which define the necessary structures.

To open an input or output line to the terminal, to change the modes on the terminal, or to inform the teletype of whom it should kill when encountering a control-C, a message is sent over the terminal link of the following form:

```
     struct ttinline{
          char tticom;
          char ttisubcom;
          char ttimodes;
     }
```

"tticom" is either OPEN, STTY, MODES, or TOKILL In the case of OPEN, "ttisubcom" is either READ or WRITE, and the return message has the new link enclosed In the case of STTY, "ttimodes" tells what the new modes should be (a bit-wise sum of ECHO, TABS, HARD, and UPPER). In the case of MODES (to find out the current modes), the return message has the modes in "ttimodes". In the case of TOKILL (to inform the terminal driver which process to kill on receipt of control-C), the message encloses a lifeline.

To open, create, unlink, alias, or get status information on a file, a message is sent over the file manager link in the following form:

```
struct ocmesg{
     int ocaction;
     int ocmode;
     char fsname[MSLEN-4];
}
```

"ocaction" is either OPEN, CREATE, UNLINK, ALIAS, or STAT. "ocmode" is the mode for OPEN or CREATE; in the case of ALIAS, it holds the length of the first file name. "ocname" contains the file name (or, in the case of ALIAS, the concatenation of two file names), null-terminated. In the cases of OPEN or CREATE, a successful return contains a valid enclosed link; for UNLINK, STAT, or ALIAS, there is no enclosed link. In the case of STAT, the return message has the structure of a "rdmesg" as in the case of READ below; the response has length 36 or 0, corresponding to success or failure, respectively. In all other cases, the response is one word: 0 on success, -1 on failure.

For either the terminal or the file manager, reading or writing is done by sending a message of the following form:

```
struct fsmesg{
     int fsaction;
     int fslength;
}
```

"fsaction" should be either READ, READLINE, or WRITE. "fslength" tells how many bytes are intended to be read, or are being sent to be written. In the case of WRITE, the text is sent in subsequent messages, and nothing is returned. In the cases of READ or READLINEy s the response is of the following form:

```
     struct rdmesg{
          char rdtext[MSLEN];
     }
```

The maximum allowable read is size MSLEN. The actual size of the returned message is contained in the "urlength" field.

To perform a seek on an open file, send a message to the file manager of the following form:

```
     struct skmesg{
          int skaction;  /* should be SEEK */
          int skoffset;
          int skmode;
     }
```

The return message is one word: 0 for success, -1 for failure.

7.2 Resource Manager Protocols

Processes that communicate explicitly with the resource manager must include the header file "resource.h". The following structure is declared there:

```
struct rmmesg {     /* messages to resource managers */
     int rmreq;      /* type of request */
     int rmarg;      /* various miscellaneous arguments */
     int rmmode;     /* the mode for STARTs or KILLs */
}
```

The resource manager keeps track of which images (code segments) and processes exist. A separate resource manager runs on each machine in the network; these programs communicate with each other, but are relatively independent.

Each resource manager holds a terminal link and file manager link, which are either for local utility processes or else links received from the first resource manager initialized. Whenever a resource manager has a local terminal it also has a local command interpreter.

There are three kinds of processes: FOREGROUND, BACK-GROUND, and DETACHED. When a process is started, its link 0 is owned by the local resource manager, to whom all of this process's requests are directed.

The first FOREGROUND process for any terminal is always the command interpreter, which initially "has the ball". Each terminal always has one FOREGROUND process that "has the ball". The process "with the ball" may create another FOREGROUND process, which means that the child now "has the ball". The meaning of "having the ball" is that a control-C entered on the corresponding terminal will terminate the process. When the process "with the ball" terminates, its parent then "recovers the ball", and will be terminated by the next control-C. If one of the processes in this FOREGROUND chain terminates, the chain is re-linked appropriately. The command interpreter is an exception in that control-C's have no effect on it.

A process may also create another process as a BACKGROUND process. In this case, the child's process identifier is returned to the parent, and later the parent can use this identifier to terminate the child. These identifiers are assigned by the resource manager, and are distinct from the process identifiers used in the kernel.

A DETACHED process cannot be terminated by either method.

A user may make five kinds of requests on its resource manager:

1. RMTTREQ Request

The resource manager is requested to give the requestor a link to the requestor's terminal. This link will be sent over the enclosed link in the request, which should therefore be a REPLY link.

2. RMFSREQ Request

The resource manager duplicates its file manager link and sends it back over the enclosed link in the request, which should therefore be a REPLY link.

3. RMSTART Request

The resource manager will start a process, using the link enclosed with this request for two purposes: 1) to respond to the request (see conditions for response below), or 2) to save it and give to the child if the child asks for it (see RMPLINK below). The caller must be careful, of course, not to give a REPLY link if both uses are intended. Also, the caller must make the enclosed link GIVEALL if the resource manager should try to load the process on another machine, rather than giving up if it doesn't fit on the local one. The RMSTART request also specifies the file name and an integer argument to be given to the child when it starts.

The caller also specifies a "mode" for starting the child, which is a combination of bits with various meanings. The user should specify either BACKGROUND, FOREGROUND, or DETACHED (the default is DETACHED). FOREGROUND is only allowed if the requestor currently "has the ball" for its terminal. The user may specify EXCLUSIVE, which causes the resource manager to load it on a machine only if there is no like-named core image with its EXCLUSIVE bit set, on that machine. The user should specify either SHARE, REUSE, EXCLUSIVE, or VIRGIN (the default is VIRGIN). These alternatives are described above (see "fork"). The user should also specify either GENTLY or ROUGHLY (the default is GENTLY). If GENTLY, the resource manager will first try to load it locally without throwing out any other unused images and then will try to do the same on other machines. When this fails, or if ROUGHLY was specified, it tries to make room locally for the new process, and then tries to do so on other machines. The user should also specify either ANSWER or NOANSWER (the default is NOANSWER). If ANSWER is specified, or if BACKGROUND was specified, then the resource manager sends a reply over the enclosed link. The first word of the reply is the return code; -1 always means failure; 0 means success except in the case of BACKGROUND, when the value returned is the process identifier of the child.

An existing code segment is reusable if the filename still refers to an existing publicly executable load format file that has not been modified since the copy in question was loaded. Any number of processes may share a code segment. The terminal associated with a child process is always the same as the one associated with its parent; the command interpreter is loaded with a terminal during initialization.

4. RMKILL Request

The resource manager kills the process whose process identifier is given as part of the request. The request may enclose a link that is used to give a one-word acknowledgement of success or failure if the request specifies ANSWER (as in RMSTART, described above). The process being killed must of course be BACKGROUND, and only the process that started it is allowed to kill it.

5. RMPLINK Request

The resource manager returns the link that was originally enclosed with the request that started this process. It is returned over the link enclosed with the RMPLINK request, which must therefore be of the proper type, whichever that may be.

8. ACKNOWLEDGEMENTS

The authors would like to acknowledge the assistance of Professor Sun Zhong Xiu of Nanking University, Peoples Republic of China, and the following graduate students who have been involved in the Arachne project: Jonathan Dreyer, Jack Fishburn, Michael Horowitz, Will Leland, and Paul Pierce. Their hard work has helped Arachne to reach its current level of development.

9. REFERENCES

Baskett, F., Howard, J. H., Montague J. T., "Task Communication in Demos", Proceedings of the Sixth Symposium on Operating Systems Principles, pp. 23-31, November 1977.

Finkel, R. A., Solomon, M. H., The Roscoe Kernel, Version 1.0, University of Wisconsin--Madison Computer Sciences Technical Report #337, September 1978.

Finkel, R. A., Solomon, M. H., Tischler, R. L., Roscoe Utility Processes, University of Wisconsin--Madison Computer Sciences Technical Report #338, February 1979.

Finkel, R. A., Solomon, M., and Tischler, R., "The Roscoe Resource Manager", Proceedings of Compcon Spring 1979, pp. 88-91, February, 1979.

Finkel, R. A., Solomon, M. H., Tischler, R., Roscoe User Guide, Version 1.1, University of Wisconsin--Madison Mathematics Research Center Technical Report #1930, March 1979.

Finkel, R. A., Solomon, M. H., Tischler, R., Arachne User Guide, Version 1.2, University of Wisconsin--Madison Computer Sciences Technical Report #379, February 1980.

Finkel, R. A., Solomon, M. H., The Arachne Kernel, Version 1.2, University of Wisconsin--Madison Computer Sciences Technical Report #380, February 1980.

Kernighan, B. W., Ritchie, D. M., The C Programming Language, Prentice-Hall, 1978.

Ritchie, D. M., Thompson, K., "The UNIX Time-Sharing System", Communications of the ACM, Vol. 17, No 7, pp. 365-375, July 1974.

Solomon, M. H., Finkel, R. A., ROSCOE -- a multiminicomputer operating system, University of Wisconsin--Madison Computer Sciences Technical Report #321, September 1978.

Solomon, M., and Finkel, R., "The Roscoe Distributed Operating System", Proceedings of the Seventh Symposium on Operating Systems Principles, pp. 108-114, 10-12 December, 1979.

Tischler, R. L., Finkel, R. A., Solomon, M. H., Roscoe User

Guide, Version 1.0, University of Wisconsin--Madison Computer Sciences Technical Report #336, September 1978.