THE ARACHNE KERNEL

Version 1.2

by

Raphael Finkel
Marvin Solomon

Computer Science Technical Report #380

April 1980

THE ARACHNE KERNEL [*]

Version 1.2
April 1980

Raphael Finkel
Marvin Solomon

Technical Report 380

Abstract

Arachne is a multi-computer operating system running on a network of LSI-11 computers at the University of Wisconsin. This document describes the implementation of the Arachne kernel at the level of detail necessary for a programmer who intends to add a module or modify the existing code. Companion reports describe the purposes and concepts underlying the Arachne project, present the implementation details of the utility processes, and display Arachne from the point of view of the user program.

---

[*] 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

THE ARACHNE KERNEL

1. INTRODUCTION

The Arachne project at the University of Wisconsin is implementing a distributed operating system for several co-operating LSI-11 microcomputers [1,2,3]. Documentation for programmers writing code to be run under Arachne can be found in a companion report [4,5,6]. Arachne is in a state of flux; the details of the kernel are likely to have changed since the time this report was written.

The operating system is divided into the kernel and the utility processes. The kernel manages storage, process creation, deletion, and scheduling, and messages. Utility processes provide terminal handling, file management, interactive command line interpretation, and resource allocation.

This report describes the Arachne kernel at the level of detail necessary for systems programmers who might be modifying or inserting code. Utility programs are documented in a companion report [7] and further described elsewhere [8].

All code for Arachne is written in the C language [9] or in the Unix [10] PDP-11 assembler language. The C compiler has been

modified to provide stack-limit checking at the entry to each procedure, since the LSI-11 does not have any hardware stack limit check or memory management. The Arachne kernel is compiled and linked on Unix on a PDP-11/40 and is then sent to the various LSI-11 machines through DR-11C (sixteen-bit parallel) lines. Details of the linking and sending procedures are given below.

User programs that run under Arachne are written either in C or in Elmer, a new language developed by the Arachne project. Elmer has the advantage that its code is completely relocatable and the language is strongly typed. Various parts of the kernel, particularly those dealing with core images (Section 4.6) treat C and Elmer programs differently.

1.1 Revisions

Version 1.1 of Arachne differs in several significant ways from Version 1.0 [4,11]. First, Elmer has been introduced. Interfaces for this language have caused changes in the scheduler and service call dispatcher. Second, the driver for the DRV-11 lines that connect the machines has been completely revised, with a large resulting gain in speed. Third, link tables for processes can now expand so that the occasional process that needs a large table can get one. Fourth, certain locations in low core are used to store certain state variables to assist in debugging and tuning. Fifth, process identifiers are now unique even across restarts of Arachne. Sixth, messages now include length information, so the various copying steps can be sped up in the case of short messages.

Version 1.2 differs from Version 1.1 [5] in the following ways: First, the checksum is periodically checked on the kernel code. Second, the message receipt structure "usmesg" has been abolished, and "urmesg" no longer has a field to hold the message that is being received. Instead, the sending and receiving procedures now take an explicit argument pointing to the message string. This modification enables Elmer programs to use standard send and receive. Third, the service call convention for Elmer has been modified and the special cases for Elmer in the scheduler have been largely removed. Fourth, if a bit in the global debugging location is set, checksums are verified for each process before it is executed. Fifth, a new link restriction, MAYERROR, allows the link holder to send an error message, receipt of which raises an exception. Sixth, the "userdie" service call now takes an argument that becomes the body of any resulting DESTROYED messages. Kernel-induced termination of processes also provides information in the body of the DESTROYED messages. Seventh, exceptions may be raised during the execution of service calls. All USRERRORs raise exceptions. Uncaught exceptions terminate the process that caused them. Processes may catch exceptions. Eighth, a new service call "usercatch" allows a process to establish a routine that will catch messages asynchronously. Ninth, the clock routines have been completely rewritten to fix an 8% time loss and to simplify the algorithm. Wakeups are now accurate only to the nearest second. Tenth, facilities have been added for gathering statistics on the CPU usage of processes.

Finally, 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. PREPARING THE KERNEL

2.1 Source Files

All files for Arachne reside in the directory /usr/network/roscoe. Each source file has a name and an extension, for example, "clock.h" has the name "clock" and the extension "h". The extensions follow this pattern:

extension	purpose
u	source file in C, should be compiled with the stack-limit-checking compiler.
c	source file in C, should be compiled with the normal compiler.
s	source file in assembler.
h	header file for C, defines global data structures meant to be included in several modules.
o	object file produced from corresponding u, c, or s file.
lda	object file in LSI-11 absolute loader format.

The source files for utility processes are in the subdirectory user/util. These files are all of the "u" and "h" variety. The Arachne loader, although part of the kernel, is also in this directory, since it follows user, not kernel, naming conventions. The object files for all user processes, including utility processes, are stored in the subdirectory "user". Many library routines are available in the library "libr.a" in that subdirectory. The source for these routines is in the subdirectory "library". Some Elmer library routines are in the subdirectory "elmer".

The kernel source code is mostly found in thirteen "u" files. In addition, there are three "c" files, 2 "s" files, the loader (a "u" file in the "user" subdirectory), and two empty files that define entry points for the loader. Each file begins with an extensive comment naming the procedures, data, and structures that are imported into, exported from, and local to that module. Many modules use "h" header files to communicate exported data and structures. The cross-reference file "cref" contains a complete listing of all procedures that are defined in one module and used in other modules. This file lists each procedure name, the defining file, and the names of all referencing files. It is also easy to find all uses of a particular function, say "foo", by using the "grep" program:

```
     grep foo *.u *.c *.s
```

This program searches for all mentions of "foo" in all source files.

2.2 Compilation and linking

The kernel can be compiled and linked by invoking "lprep". This command file calls "compile" on each source file for the kernel. The program "compile" compares the date on the source file (whether its extension is "u", "c", or "s") with the date on its object file (extension "o"). If the former has been changed since the latter was prepared, the appropriate compiler/assembler is invoked to create a new object file. The linker is then invoked to combine the various object files. Finally, "lprep" invokes "mklda" to convert the Unix object file format into absolute loader format, producing "arachnenew.lda". After this version has been checked out, it may be moved into "arachne.lda", which is the current working version of Arachne.

2.3 Loading and Starting

To load a new version of Arachne into a cleared LSI-11, use the Unix program "coml". This program acts as the console terminal to the LSI-11 and can send whole files either on the console line or through the fast word-parallel (DR-11C) line. By default, "coml" uses the fast line to machine Ø. To set it to, say, machine 1, type "<control-T>1".

In the cleared state, the LSI-11 executes microcode that implements ODT (octal debugging technique) [12]. To enter ODT at any time, send a <break> to the console. The program "coml" sends a break when the user types "<control-T>B". ODT prompts for input with "@". To invoke the hardware absolute loader, follow this procedure:

who	what	meaning
user	<ctr-T>B	Interrupt LSI-11
LSI	@	prompt
user	173000G	start a diagnostic program at 173000.
LSI	$	prompt
user	AL<cr>	starts the absolute loader
user	<ctr-T>S	send along slow line
coml	file =	prompt for file name
user	drload.lda<cr>	name of software loader
coml	sending	
LSI	Loading	software loading starting

If the PDP-11/40 is not connected to the LSI-11 console, but a terminal is, then the following program should be entered through the terminal:

location	contents
1000	005737
	167730
	002375
	013721
	167734
	077006
	000000
R0	000273
R1	157000

Once these values are loaded, start the LSI-11 at location 1000 (command 1000G), and send the file "drload.abs" across the fast line to the LSI-11. (See below for two ways to send files across the fast line.)

Once "drload" is present, it can be started by typing "157000G" to ODT. The message "Loading" should appear immediately. After this message appears, there are two ways to load the kernel:

who	what	meaning
user	<ctr-T>F	send along fast line
coml	file =	prompt for file name
user	arachne.lda<cr>	name of kernel file
coml	sending	
drload	Starting	

The other way is to give the Unix command

```
cp foo /dev/dr7
```

where 7 should be replaced by the line number for that machine.

Several core locations are intended not to be disturbed between loads; however, the hardware loader can damage these numbers. They are as follows:

location	meaning	suggested value
157700	machine no.	0 to 4
157702	print delay	0 for fast terminal (crt)
		2000 for coml at 1200 baud
		3000 for coml at 300 baud
157704	nextid	anything unusual

The variable "nextid" is used to prevent processes started under the new version of Arachne from having the same id as those under the old version. Only the lower eight bits of "nextid" need to be set.

It is possible to restart Arachne at any time by halting execution and starting at location 700 (ODT terminal command 700G).

3. FUNDAMENTAL MODULES

3.1 Initialization

When Arachne is started, a small amount of code in module "crtl.s" causes a Unibus reset to clear all interrupt enablings, sets the kernel stack to location 700, and jumps to routine "main" in module "lsi.c". This routine sets the kernel stack limit to 400 (in global location 1000), inhibits interrupts, and invokes "initall", which calls the initialization code for each module.

After all modules are initialized, "initall" prints the name of the operating system, the machine id, and the amount of memory used.

The "main" routine then establishes the periodic checksum calculation by calling "dochksum", which reschedules itself on the clock (Section 3.8) every 10 seconds. Finally, "main" initiates the kernel job "kernjob" in "kernjob.u". (For details of scheduling, see Section 4.) Then "startscheduler" in module "schedule.u" is called.

When the kernel job starts, it acts as a normal process controlled by the scheduler. It first prompts the terminal for one of three methods of proceeding. The first is to load no programs, but to wait in a kernel-job loop for a message to arrive from a foreign site. The second is to load in a program. If this option is chosen, the kernel job submits a normal user load

request that will prompt the terminal for the file and will read it across the fast line. Methods like those given in Section 2.3 can be used to send the desired program. After the program is loaded, it is allowed to execute, and the kernel job remains in a loop waiting for either a message from a foreign site or for the termination of the loaded program. If the latter occurs, the kernel job again presents the three options. The third option is to load a new resource manager that should be linked into the web of existing resource managers. The kernel will prompt for the configuration desired for this machine (that is, what other utility processes should be resident) and for the machine number of a working Arachne site. The details of starting resource managers are outside the scope of this document; see [7]. The kernel job interface to the resource manager is one of the two instances in which the kernel needs to know details of utility processes. (The other instance is the loader, which needs to know file system protocols. The loader is discussed in Section 4.6.)

<u>Files</u>
    crtl.s, lsi.c, kernjob.u

<u>Procedures</u>
main()
    Calls initall, then starts up kernjob.
initall()
    Calls the intialization code for each module.
kernjob()
    Loads the first user program, then waits for messages from remote kernels to load up new programs.
dochksum()
    Computes the checksum, complains if an error is found. Sets a clock event that will run it again in 10 seconds.

3.2 Low level interrupt handling

A small module, "lowestirp.s", is in charge of dispatching the various interrupts that may occur. These routines generally save machine registers r0 and r1 and call a C routine in the kernel. When that routine is done, the registers are restored and the interrupt is dismissed. The interrupts dealt with are the clock interrupt, which calls "timesup" in "clock.u" (Section 3.8) and the interrupts from the various dr lines to other machines, which are dispatched to "inhandle" or "outhandle" in "line.u" (Section 5.2). These latter interrupt routines also place an argument on the stack to tell which dr line caused the interrupt. User interrupts (Section 6.1) are caught by "lowu0" and "lowu1" (and others, as more are allowed) and dispatched to "uinterrupt" in "interrupt.u". Trace traps are caught by "tracetrap", which prints the location of the trapping instruction and allows the user to continue either with trace trap on or off. This routine is seldom used, because it seems to be unreliable.

In addition, "lowestirp.s" provides the routines "pause", used to display a number and halt the machine (for debugging; see Section 3.3) and "psset", to set the processor priority (for locks; see Section 3.6).

Files
    lowestirp.s

Data
irprout
    A table for routing dr line interrupts.
irprsize
    The length of "irprout"; used in initialization of the line handlers.
lowuint
    A table of user interrupt handlers.

Procedures
clkint
    Calls "timesup" when the clock ticks.
setflag
    During parts of initialization, non-existent memory traps are dispatched here to set a global flag that can be examined.
int psset(newps)
    Sets priority to newps (actually, priority left-shifted by 5 bits). Returns old priority.
pause(code)
    Displays the code and halts. Can be continued by the ODT command "P".
tracetrap
    Prints the current program counter, accepts X to leave trace trap on, anything else to turn it off.

3.3 Debugging Aids

The module "error.c" contains various debugging aids. This module is compiled with the standard C compiler, since stack overflow calls "syserror" in this module.

The routine "pause" in module "lowestirp.s" prints its decimal argument and then halts. The console ODT command "P" will proceed from the halt.

When the Arachne kernel discovers a situation from which it cannot recover, the macro SYSERROR (defined in "util.h") calls the routine "syserror" in module "error.c". This routine prints a coded message to the console terminal and then executes a "pause(-1)". The messages are listed in the file "syserrors"; their text is not stored in the kernel proper in order to save space. It is usually not wise to continue from such an error.

Non-existent memory traps and illegal instruction traps cause a message to be printed from routine "nxmerror". The proper interrupt vector is established during "initall" in "lsi.c". If the error is in the kernel, syserror is invoked. Otherwise "userdie("bad trap")" in "schedule.u" is called to terminate the offending process.

Every ten seconds the kernel runs "dochksum" in "lsi.c" to insure that its code space has not been damaged. The new checksum is computed and stored in the variable "checksum" if that variable holds a zero; otherwise, the new checksum is compared with the current one. If there is a discrepancy, a syserror is signalled.

Service routines (Section 3.7) return failure to the calling process by invoking the macro USRERROR (defined in "util.h"), which calls "usererror" in "error.c" and then returns a negative error code. This procedure raises an exception by setting "curerror" in "schedule.u" to 1 and then prints a debugging message. A list of errors can be found in the file "usererrors".

Exceptions can also be raised by receipt of an error message. If the user has invoked "uerrhandle" in "schedule.u", then when the service call during which the exception was raised returns through the wormhole, the user-supplied routine is invoked with the same arguments as the service call. In addition, two extra arguments are placed in front of the old argument list: the

service code and its returned value. The user routine will return to the calling point within the user program. If "uerrhandle" has not been invoked, raised exceptions cause a call to "die("exception not caught")".

If bit 020 is set in "debug", then the scheduler will verify the checksum of the text segment of each process right before it is scheduled. Processes whose text segments have been garbaged are terminated.

The routine "snap" in module "error.c" checks its argument against the global variable "debug", which is always stored in location 776. If "debug" and the argument to "snap" have any bits in common, then a walkback of routine return addresses is printed to the terminal and "pause" is invoked.

The global location "debug" can also be used as a guard on various diagnostic output while new modules or routines are being tested.

In addition, several important variables are kept in low core to facilitate debugging.

location	name	meaning
776	debug	flags for debugging
760	curusr	process number of current process
756	numkmes	number of remaining message buffers
762	curerror	whether an exception has been raised

Several debugging routines that print various structures can be found in the file "debug.u". These routines are not normally included in the kernel but can be edited into the kernel source during debugging. They include "printprocs" to print the status of each process, "printkm" to print a kmesg buffer, "println" to print a link, "prntclkq" to display the clock queue, "chkwt" to check the consistency of the message arrival queues, and "free-print" to print a map of free space.

Files
```
     lowestirp.s, error.c, debug.u
```

Procedures
```
pause(code)
     Print the code on the terminal and halt.
snap(tag)
     If location "debug" and "tag" have any bits in common, print
     a traceback of procedure frames and pause(tag).
nxmerror(dummy)
     Print message, then either call schedcall or syserror.
     Print a trace of the stack and pause.
syserror(code)
     Print the code (which can be found in the file "syserrors",
     then call "pause(-1)".
```

3.4 I/O

The lowest-level input-output routines that deal with the console terminal are in module "io.c". Since these routines can be called after a stack limit exception, they are compiled by the normal C compiler that does not check for stack overflow. The routine "outchar" sends a character to the console terminal after sitting for a while in a busy loop. The number of iterations of this loop is stored in the global location "delaylen" at 157702 (octal), which can be set in ODT. The delay is to slow down the natural rate of transfer so that if the program "com1" is receiving the console stream, Unix can keep up with the line.

The "printf" routine used by the kernel is in module "lsi.c". It acts much like "printf" in Unix, except it does not have widths in its format specifications. "Printf" is only used for debugging output; terminal I/O is ordinarily handled by the terminal driver utility process [6,7]. The module "lsi.c" de-

fines "putchar", which calls "outchar" in "io.c".

Files
```
     io.c, lsi.c
```

Procedures
```
outchar(dev,c) char c;
     Print one character to the terminal at address dev.
char inchar(dev)
     Read one character from the terminal at the address dev.
ttyflush()
     Remove any waiting character from the teletype input data
     register.
printf(pmesg,arg) char *p_mesg;
     Emergency and debugging output using putchar().
putchar(c) char c;
     Same as outchar(TTY,c)
char getchar()
     Same as inchar(TTY)
```

Data
```
delaylen
     Location where tty delay is stored, defined in crtl.s
TTY
     address of console terminal registers (0177560)
```

3.5 Free storage management

Free storage management routines are found in the module "free.u". During Arachne initialization, the routine "freeinit" grabs a large section of memory from the end of the kernel using the "sbrk" routine in "lsi.c". The boundary tag method [13] is used to allocate chunks of storage through the routine "freeget" and to return them through the routine "freerel". The regions that are returned are intialized to contain -2 in every word to catch subsequent initialization errors in the users of the regions. Free storage is used to find room for certain kernel tables that are made once during Arachne initialization and to provide room for core images. This latter space is reclaimed by a reference count technique. (See Section 4.6.)

Files

    free.u, lsi.c

Data Structures

```
int freesize
     Length of the free space in words
int *freespace
     Start of the freespace.  Set by sbrk(freesize) at initiali-
     zation.
int freeptr
     Head of a doubly linked chain of blocks of free storage.
struct { int frsize, *frnext, *frprev; }
     Format of the header of a free block.
char *highwater
     Highest address allocated by sbrk.
```

Procedures

```
freeinit()
     Free storage initialization
int *freeget(size)
     Returns a block of "size" words of free storage.  Returns 0
     if the required space is unavailable.
freerel(ptr) int *ptr
     Release the block of storage pointed to by "ptr".
char *sbrk(count)
     Allocates "count" bytes of storage by extending highwater.
int outrange(addr) char *addr;
     Returns TRUE if addr is not in any reasonable user range.
```

3.6 Locks

Several sensitive queues must not be simultaneously modified by an interrupt-level routine and a standard routine. Access to these queues is controlled by interlocks found in the module "lock.u", which provides routines "wait" and "signal". The "wait" routine makes sure that no lock is active, then places the CPU at high priority by calling "psset" in "lowestirp.s". (The LSI-11 has only two priority levels.) The "signal" routine restores the original priority (which may have been high). No interrupts are serviced between these calls. Four queues are locked in this fashion and will be discussed later: the ready

process queue, free message buffer queue, received message queue, and outgoing message queues. In addition, several routines in "clock.u" and "timing.u" use "wait" and "signal" instead of clearing and resetting the clock interrupt enable bit, since clearing it sometimes causes a bus error during acquisition of the clock interrupt vector.

Files

    lock.u and lock.h

Data Structures

```
int NUMLOCKS
     Number of lock types.  These locks are defined:
      RQLOCK:  ready queue lock; used in schedule.u
      KMBUFLOCK:  available list of message buffers; used in
     kmess.u
      IOQLOCK:  list of outgoing messages; used in line.u
      KMWTLOCK:  list of received messages; used in kmess.u
      CLKLOCK:  event queue lock; used in clock.u in lieu of di-
     sabling the clock
int curlock
     -1 if no lock active, else name of lock.
int lastps
     processor status before the current lock was entered.
```

Procedures

```
lockinit()
     Initialization routine.
wait(lock)
     Make sure no lock is in force, then move to high priority.
signal(lock)
     Make sure the argument is the lock currently in force, then
     return to the old priority.
```

3.7 Service calls

Processes running under Arachne request kernel assistance by invoking kernel service routines. C programs call these service routines by placing arguments on the stack, placing a service code in register r1, and then subroutine-jumping to location

1ØØ2.

Elmer programs place the arguments at the start of data space instead. The user's registers rØ and r1 are saved on the user's stack, and the arguments to the service routine are taken from the start of the user's data area and placed on the stack. The service routine proper is called by a subroutine call, so when it returns, "sys" can restore the registers and compute where in the Elmer program to return. The return address is found by adding rØ and r4. (Register r4 points to the start of code space, and rØ is an offset in that space.)

The routine "sys" in the module "crtl.s" resides at 1ØØ2. It has access to the scheduler variable "curtype", which indicates whether the currently running process is of the C or Elmer type. It decodes these calls and invokes the appropriate kernel routine. Each service call is represented by two words in a table, indexed by the service code: The first is the address of the service routine, and the second is a set of flags indicating whether the routine may be called at interrupt level and at normal level, and whether it is a privileged instruction. No use is currently made of the privileged flag. (Interrupt handling is discussed in Section 6.1.) This same routine calls the routine "maydie" in "schedule.u" so that the calling process may be terminated if termination is pending. (See the discussion on the scheduler, Section 4.)

When the service routine is finished, control is returned to the wormhole routine, which checks whether an exception was raised during its execution. If not, execution continues at the

user's point of call. If an exception has been raised and the user has not established an exception handler (via "uerrhandle" in "error.c"), then the user is terminated. If an exception handler has been established, it is called with the stack arranged to appear that the user had called it directly instead of the service call. Its arguments are the returned value from the service call, the service code, and then all the arguments to the service call in order.

During execution of the service routine, the stack belonging to the calling process is used. For this reason, most kernel routines are compiled with the stack-checking version of the C compiler. The service routine must return to the wormhole and it must access its arguments on the stack, but the wormhole itself needs to save two local variables (the service code and the return address to the user), so a copy of the arguments is made before the service call is actually invoked. At the point that the service call is invoked, the stack looks like this:

```
wormhole return address
arg 1 copy
...
arg NMARGS copy
service code
user return address
arg 1
...
arg NMARGS
```

If the user-supplied exception handler is invoked, the stack is set this way:

```
user return address
service routine return value
service code
arg 1
...
arg NMARGS
```

Service routines usually have two names: the one that applies in the kernel, and the alias employed by user processes. The aliases are defined in a companion report [6].

Files
    crtl.s

Data Structures
table
    Branch table for dispatching service calls.

Procedures
sys
    At location 1002. Dispatches service calls.

3.8 The clock

The module "clock.u" provides a routine "setalarm", which allows an arbitrary routine with up to four arguments to be called some number of seconds in the future. If the caller of "setalarm" specifies a "delay" of n, then the given routine will be called when the clock has interrupted n+1 more times. The hardware clock interrupts once per second, so the given routine will called between n and n+1 seconds from the time setalarm is called. The scheduled routine will run at high priority. The routine "setalarm" returns a code that can be used to turn the alarm off before it expires by invoking "turnoff" with that code as an argument.

The clock routines are implemented with a programmable clock, which is set to interrupt at one-second intervals. The

routines "setalarm" and "time" use locks (Section 3.6) to prevent clock interrupts from occurring at critical times. (Clearing the interrupt-enable bit can make the processor halt.)

A queue of events stores the alarms that have been set; each node includes the time, in seconds from the epoch (beginning of 1973, Madison standard time), when the event is to occur.

When the clock interrupts, it calls "timesup" at interrupt level to increment "datereg" and "timeofday", "nexttime" in "timing.u" to increment "timeslot" for statistics gathering (Section 4.7), and to execute all events on the queue whose time has expired. The clock stays in repeat interrupt mode, so Arachne's notion of time is maintained with the accuracy of the clock hardware.

The date, given in seconds since Jan 1, 1973, may be obtained with the service routine "date", which returns a long integer. The service routine "setdate" may be used to change this date. Another service routine, "time", returns a long integer that counts in increments of .0001 seconds. This latter timer cannot be modified. "Time" returns the sum of "timeofday" and the elapsed time since the last clock interrupt. This elapsed time is determined from the clock's count register, which decrements by one every ten-thousandth of a second. These services are used both by user programs and outgoing message sending and message reception, which are discussed in Sections 5.2 and 5.4.

Files
```
     clock.h and clock.u
```

Data Structures
```
int NBRARGS
     number of arguments to alarm routine, currently four
struct cqvector {
      int (*cqfunc)();   /* function to call when  alarm  expires
     */
      int cqargs[NBRARGS];  /* arguments to that function */
      };
char *timeofday
     interval timer in seconds
long datereg
     seconds since the epoch (beginning of 1973, Central standard
     time)
int uniquecode
     used to distinguish alarm events
struct clkqnode {
      struct clkqnode *cqnext; long expire;   /* time  when  this
     event is to occur; in seconds from the epoch */
      int cqcode;   /* distinguishing code */
      struct cqvector cqaction;
      };
struct clkqnode clkqueue[CLKQSIZE],*clkqtop,*clkqfree
     Queue of alarms, start of queue, head of free list of  queue
     nodes.
```

Procedures
```
long time()
     Returns the current interval timer value (in .0001 seconds),
     as determined from "timeofday" and the clock count register.
long date()
     Returns the current value of "datereg".
setdate(n) long n;
     Sets datereg equal to n.
int setalarm(delay,action) struct cqvector *action
     Places an alarm on the queue.  When it expires,  the  action
     will be taken.  Returns a code to be used for "turnoff".
turnoff(code)
     Remove the alarm that has the given code.
timesup()
     Called by a clock interrupt every second to run the  pending
     events.
clockinit()
     Initialization  of  the  clock  data  structures  and  clock
     hardware.   The clock registers are not modified again after
     this routine finishes.
```

4. PROCESS MANAGEMENT

Processes are managed by the scheduler and the core image routines. There are two kinds of core images, C and Elmer. C images contain code, initialized data, and space for the uninitialized data. Elmer images contain only code. They are treated slightly differently by the modules that handle processes.

4.1 Scheduler data

The scheduler resides in "schedule.u". This module keeps an array of per-process information, which includes a stack frame pointer, the current status of the process (non-existent, sleeping, ready, running, halted, to be halted, to be terminated), a pointer to the next process on the ready queue (if this process is itself ready), the lowest address allowed for the stack, an index into a reference count table for purposes of storage reclamation, the type of the core image (Elmer or C), the address of the data area (for Elmer processes only), the exception handler address (Section 6.2), asynchronous message handler address (Section 5.5), arguments and active channels for asynchronous message receipt, and timing information (Section 4.7).

Each process is identified by two numbers: an index into the process table (process number) and a network-wide unique identifier (process id). The module "schedule.u" provides routines "getid" and "getno" for conversion between the two. The high byte of the process id is the machine id; the low byte is a sequence number. The variable "nextid" is saved across instantiations of Arachne so that process identifiers are not reused soon after a Arachne machine goes down and then comes up and rejoins the network. When a new process is to be assigned its process id, the low byte of nextid is incremented so long as it is equal to any extant process id. After nextid is assigned to the new process, the low byte of nextid is incremented. Process ids are used as message destinations, since it is wrong to direct a message to a new process that happens to have the same process number as the desired but terminated process.

4.2 Initiating processes

A process is started by a call to "initiate", which takes a core image, an argument, an owner id, and, for Elmer, a file descriptor. A new stack is created and initialized so when the process starts, it will appear as if it has been called with the given argument. The stack is also prepared with a return address that points to "fallthrough", so if the process returns, "fallthrough" can perform a "schedcall(DIE)" and properly terminate it. For an Elmer image, a smaller stack is allocated, since Elmer allocates all local variables in its data area rather than on the stack. In addition, room for the Elmer data area is created and initialized by calling "uload" in module "uloader".

In either case, the reference count on the core image is incremented to indicate that another process is running in it. A check is made to insure that the owner of the core image is performing the initiate. Finally, the process is awakened by a call

to "awaken".

The routine "awaken" causes a process whose status is "sleeping" to be placed on the ready queue with status "ready". It does nothing if the process was not sleeping. (For this reason, "initiate" first sets the status to "sleeping" and then calls "awaken".) The usual reason that processes enter the sleeping state is to await a message. It never hurts to spuriously awaken a process, since it will check to see if the right message has arrived and will return to the sleeping state if it hasn't. (See Section 5.1 on messages.)

Scheduling is round-robin non-pre-emptive. The scheduler itself remains in a loop in routine "scheduler". Once a process has been chosen for execution, the scheduler loop verifies the checksum in its text area (if the 020 bit is on in global location "debug"), then it calls "strtusr" (in "resume.s"), which switches to the process stack and transfers control to that process. The stack limit for the new process is placed in location 1000 (octal) so that procedure entry code can check for stack limit violations. (Elmer programs do not use the stack, but the kernel routines they invoke use the small stack in the Elmer program.) Interrupts are prevented during all stack switching, when the stack limit implied by location 1000 is not consistent with the stack pointer in hardware register r6. (Since interrupt-level routines also use stack-limit checking, they would likely signal a stack overflow.) Processes are run at low priority.

4.3 Process switching

The details of process switching depend on the standard C subroutine linkage conventions. Normally, each activation of a procedure has a corresponding stack frame. More recent activations correspond to stack frames at lower addresses. A frame is identified by an address (called the frame pointer) of one of its fields. The word addressed by the frame pointer contains the frame pointer for the next older (higher addressed) frame. The following word (next higher address) contains the return address, and subsequent words contain the actual parameters, the first actual parameter being at the lowest address. The three words preceding the frame pointer word are used to save registers r2, r3, and r4. Subsequent locations are used for local variables and temporary storage. Register r5 always contains the current frame pointer:

```
     r6 -->  temporary storage
             local variables
             saved registers
     r5 -->  previous frame pointer
             return address
             first actual parameter
             second actual parameter
```

(Lower addresses are towards the top of the diagram.)

To call procedure "foo", say, the calling program pushes the actual parameters onto the stack and then executes a "jsr pc,foo" or a "jsr pc,*$foo" instruction, thus pushing the return address onto the stack. If the program "foo" was compiled without stack limit checking, it first executes "jsr r5,csv", which pushes the previous frame pointer. (The routines "csv", "ncsv" and "cret" are in the C library.) The routine "csv" then saves r2, r3, and r4, sets r5, and returns to "foo". The code of "foo" then decrements r6 to make room for local variables. The stack pointer r6 ends up pointing one word beyond (below) the last local variable.

The stack-limit-checking version of "foo" is similar, but instead of calling "csv" and then making room for local variables, it places the negative of the number of bytes of local variables into r1 and calls "ncsv", which checks the limit (in location 1000) to see that the space for local variables can be granted with room to spare and then decrements the stack pointer. Return from a function is accomplished in either version by a jump to "cret", which restores the registers r2, r3, r4, and r5 to their values prior to the call, sets r6 to point to the return address on the stack, and executes an "rts pc" instruction. Actual parameters are cleared from the stack by the calling program. (All parameters are passed by value.) Just before returning from a procedure, "cret" sets r6 to point to the word following the word addressed by r5 during the procedure, that is, the return address. Hence, the entire state of a process can be saved by storing r5, provided the process is just about to return from a procedure.

A process that wishes to relinquish control calls "schedcall". The code in "schedcall" stores the current r5 in the stack frame pointer field for the current process, replaces it by the saved r5 from the scheduler "process", and returns. Similarly, "strtusr" resumes a process by restoring the saved r5 value and doing an ordinary return. To the process, it looks as if the call to "schedcall" returns immediately.

4.4 Process switching in Elmer

Elmer process expect that r5 points to the data area (which includes both initialized and uninitialized data), and the first word of the data area points to the start of the code area. When an Elmer process makes a service call, it places an argument count and the arguments themselves starting at the second word of its data space. The return address is in r3, represented as an offset from the beginning of the code space. The service call is handled by "sys" in "crtl.s", which places the arguments on the stack, so that the scheduler can treat Elmer and C programs alike.

4.5 Treatment of schedcall

The argument to "schedcall", one of DIE, CONTINUE, and SLEEP, is passed back to the scheduler as a result of the invocation of "strtusr". In the case of DIE, "killoff" is invoked and another process is chosen at the start of the scheduler loop. In the case of CONTINUE, the process remains ready but is placed at the end of the ready queue. In the case of SLEEP, the process is placed in state "sleeping". This alternative is used by service routines that need to wait an unspecified amount of time and are willing to allow other processes to execute in the meantime. Forms of schedcall(DIE) and schedcall(CONTINUE) are provided for processes in the service routines "userdie" and "usernice", aliases for "die" and "nice". "Userdie" takes an argument that

is placed in "diesmesg" so that "killoff" can provide "lnclear" (Section 5.3) with it.

The procedure "killoff" first removes the target process from the ready queue, if it is there. The status is set to "halted". If "killoff" was invoked to halt the process, it then returns. (This feature is not currently used.) If "killoff" is to terminate the process, it then invokes "intclear" to remove any interrupt handlers associated with the target (Section 6.1), "freerel" to return the target's stack (and the data area for Elmer processes), "lnclear" in "lnmaint.u" to clear the target's link table (Section 5.3), "kmclear" in "kmess.u" to remove any messages awaiting the target (Section 5.4), and then it reduces the reference count that records how many processes are active in the target's core image. If the process is an Elmer process, its data area is returned to free storage. The routine "lnclear" can itself indirectly cause the termination of other processes, so "killoff" is indirectly recursive. A running process cannot be halted or terminated, so "killoff" changes the state of a running process to "to be killed" or "to be halted", as appropriate. Each service call invokes the procedure "maydie", which checks to see if the current process is in one of these states. If so, "schedcall" is invoked to return control to the scheduler, which can then invoke "killoff" to finish the action. In case the process was slated for termination, "killoff" sets "diemesg" to "killed".

Both "awaken" and the scheduler loop use locks (Section 3.6) to prevent corruption of the ready queue.

The scheduler is itself started by the routine "startscheduler". First, the routine "scheduler" is initiated as if it were a normal process. This action provides the scheduler with a reasonable stack. Then "strtusr" is invoked to transfer control to the scheduler. Its first action is to invoke "killoff" on itself, removing all vestiges of itself from the process table. The "killoff" routine knows not to remove the scheduler's stack.

Files

    schedule.h, schedule.u, resume.s

Data Structures

```
STKLEN
     Length of a user stack, in words.
int NUMPROCS
     Size of the process table.
int idtable[NUMPROCS]
     Table of process identifiers for each process number.
int curusr
     Process number of currently running process.
int curerror
     If not zero, the current service call has raised  an  excep-
     tion.
int curtype
     Indicates if current process is C or Elmer, so that the ser-
     vice call dispatcher can handle service calls correctly.
int nextid
     Non-reusable id for next process to be initiated, stored  at
     absolute location 157704.
char *diemesg
     Either 0 or points to a message provided by the current user
     in a userdie call or by the kernel in terminating the user.
int kernno
     process number of kernel job; set by lsi.c
struct ppnode {
       int ppfptr;  /* stack frame pointer: stored r5 of user pro-
     gram */
       int ppstatus;  /* One of the following:
        PPNQNEX 01
        PPSLEEPING 02
        PPREADY 04
        PPRUNNING 010
        PPHALTED 020
```

```
        PPTOHALT 040
        PPTOKILL 0100
        */
       int ppnext;  /* for linking into queues: in [0..NUMPROCS-1]
      */
       int *ppstklim;  /* stacktop (lowest address allowed).  Also
      address of stack for allocation */
       int ppcodeno;      /* index into codetab, -1  if  resident.
      */
       int pptype;    /* 0 if C program, 1 for Elmer */
       int ppdata;   /* location of data area of Elmer program */
       char *pperror;  /* location of user error handler, or 0 */
       int *ppcatch;     /*  address of message catcher */
       int ppcdata;      /*  catcher data area */
       int ppurmess;     /* catcher urmess area */
       int ppallcatch;   /*  bitstring  of  all  channels  caught.
      Used to determine if a channel is caught, NOT ppcatchl */
       int ppcpend;       /* true if a catch is pending */
       int pptime[TIMESLOTS];  /* number of 10,000ths of secs used
      in this interval */
       };
struct ppnode proctab[NUMPROCS]
      per-process data; indexed by procno.
int schedfp
      stack-frame pointer for the scheduler
int schdstklim
      stack limit for the scheduler
int schedno
      process number of the scheduler until it wipes it out
int rqfirst, rqlast
      head and tail of the ready queue
int curusr
      Process number of currently running process.
int NUMMODULES
      Length of core image table.
struct codetentry {
       int cdrefcount; /* reference count (-1 if not in use) */
       char *cdmemory;  /* start address of this segment */
       int cdlength;  /* number of words in the text part  of  the
      segment */
       int cdchksum;  /* checksum of the text part of the  segment
      */
       int cddata; /* 0 for C, otherwise length of data  area  for
      Elmer */
       int cdowner;  /* process id of owner */
       } codetab[NUMMODULES]; The core image table.
```

<u>Procedures</u>

maydie()
: If current process is in state "to kill" or "tohalt", calls "schedcall".

startscheduler()
: Called during initialization. Starts up the scheduler as a process, then transfers to it.

killoff(userno,how)
: The second argument is either PPTOKILL or PPTOHALT. Halts or terminates the target process. If the process is running, sets the status to PPTOKILL or PPTOHALT. If the process gets terminated, removes its stack, link table, data area (Elmer only), waiting messages, and interrupt handlers.

int newcseg(start,owner,data,length) int *start
: Finds a free slot in "codetab", initializes it so that its "cdmemory" field holds "start", and returns the slot index. Computes the checksum of the text part of the code segment and stores in in "codetab". Data is the number of words in the data area for Elmer, Ø for C. Length is the size of the text region, in words.

int getcodeseg(userno)
: Returns the core image stored in the proctab.

int getfreeno()
: Finds an available user number.

fallthrough()
: Procedure called if a process returns. Calls "schedcall(DIE)".

int initiate(codeseg,arg,owner,fd)
: Creates a new process in codeseg, creating the appropriate stack space to this process. Codeseg names the entry in codetab that governs this code segment. If the codeseg is for Elmer, then fd must be a file descriptor for the source file, so the data setment can be read. Makes a new process entry in proctab. Its entry point is the start of the memory in the given core image, and the stack is initialized to hold the given argument. The process is left in state "ready".

scheduler()
: First removes itself from the process table, retaining its own stack. Then enters a loop in which a ready process is scheduled and run through "strtusr". Upon return (through "schedcall"), process may be rescheduled, terminated, or neither, for the case CONTINUE, DIE, and SLEEP.

awaken(procno)
: Cause the given process to be placed on the runnable queue if it was sleeping.

userdie(mesg) char *mesg
: This service call is invoked by the kernel call "die(mesg)" and effects "schedcall(DIE)". It also puts mesg in "diemesg" for communication with lnclear.

usernice()
: This service call is invoked by the kernel call "nice" and effects "schedcall(CONTINUE)".

char *uerrhandle(addr) char *addr
: This service call establishes the "pperror" field in "proctab" for this user, and it returns the previous value.

char *gerrhandler()
: Returns the "pperror" field in "proctab".

schedinit()
: Initializes local tables.

int getno(procid)
: Finds the process number for the given process id.

int getid(procno)
: Finds the process id for the given process number.

strtusr(fptr,result,chkflag) int *fptr, *result, chkflag
: In "resume.s". Starts the process whose frame pointer is given. The scheduler stack pointer is saved in fptr. When this routine returns (the user process has called "schedcall(arg)"), "arg" is returned through "result". If "chkflag" is TRUE, then "maycatch" in "kmess.u" is called to treat any asynchronous message receipts that are pending (Section 5.5).

schedcall(kind)
: In "resume.s". Switches back to scheduler stack frame, places "kind" as a result, and returns from what looks like the call to "strtusr" made earlier.

checkmem(cdptr) struct codetentry *cdptr
: Returns the checksum of the given text space. In a userdie call or by the kernel in terminating the user.

4.6 <u>Core images</u>

The module "lifeline.u" provides service calls that allow one process to control another. A process can cause a program to be loaded into memory by invoking "userload" (an alias for "load"), which takes a file name and a file descriptor. The file descriptor is a link to an open file. (Links are described in Section 5.3, and the file handler is described elsewhere [6,7].)

The program "userload" calls routine "uload" in module "uloader.u" to bring in the core image of the file. This module is written as a user program, using standard service calls to accomplish the necessary communication with the file manager to read the file. If the file descriptor is -1, then "uloader" tries to read the file directly over the fast line to Unix. In this case, the file name is used as a console prompt; otherwise, the file handler is employed. The routine "uload" uses a privileged service call to acquire free space for the new load image. By inspecting the first bytes of the file, it can determine whether the file is Elmer or C. In the former case, only the code part is read in. In the latter case, the code part is read, relocation is performed, initialized data are loaded, and room is reserved and cleared for uninitialized data. The loader returns the starting address of the loaded program. The service routine "userload" finishes by establishing a code segment for the new image by calling "newcseg" in "schedule.u" and returning the index of the code segment that "newcseg" provides.

The caller of "userload" can then use this image number to start a new process in the loaded image. Since loading and startup are independent, the calling process (usually the resource manager) may start several processes in the same image and may save the images of terminated processes in which new processes may be started later. Starting a process is accomplished by the service call "userstart" (an alias for "startup"), which takes a code segment, an argument, a link to the parent, and a disposition code for that link. This routine calls "initiate" to start up a new process in that code segment (if the owner is right). If the core image to be started is an Elmer program, then "userstart" requires an extra argument: a link to the file system for the data area of the program. "Userstart"

calls "uload" in "uloader" to read in Elmer data areas.

The new process is started with a fresh link table that contains one link, the parent link supplied to "userstart". (See the discussion of links in Section 5.3.) This link is either duplicated for the child or given away outright, depending on the disposition. This link has the restriction "NODESTROY", so the child cannot destroy it except by terminating. (The parent can thus be furnished an unforgeable notification of the child's termination.) A <u>lifeline</u> is then created for the parent. It appears in many ways like a link in the parent's link table, except that it is not possible to send a message along it, and it has the restriction bit LIFELINE. The destination of the lifeline is the new child.

A kludge to allow remote loading of programs checks the file name in the "userload" call if the file descriptor is good. If the file name is a small integer between Ø and NUMMACH, the number of machines, then the call is taken as a request for remote loading. In this case, two extra arguments are used, one to be the parent link of the new process, and one to be the argument to that process. A special message is sent to the kernel job on the destination to request loading. Since the process id of the kernel job on a different machine is not universally known, a special case is made. A process id having the machine number in the upper byte and a zero in the lower byte can always be used to refer to the kernel job on that machine. Protocols for the kernel-to-kernel message are in "kernkern.h". The kernel job on the remote machine not only loads the program, it also starts it with the supplied parent link and argument. The remote kernel job returns the lifeline to the started process by sending its response along a link that it builds to the requesting process with channel 15. The routine "userload" waits for this response, then returns the lifeline to the calling process.

A process holding a lifeline may use it to control the owner of the lifeline (that is, the process to which it points). The service routine "userkill", alias for "kill", takes a lifeline as an argument. This routine sends a KILL notification to the target process. (Eventually, "userhalt" and "userresume" may be added.)

In order to remove a core image, a process may invoke the service routine "userremove" (an alias for "remove") in "schedule.u". This routine reclaims the memory through "freerel" and frees the slot in the segment table. The caller must be the owner of that segment, and the reference count must be Ø.

<u>Files</u>

lifeline.u, lnmaint.h, uloader.u, kernkern.h, kerjob.u, schedule.u.

<u>Procedures</u>

int userload(prog,fd) char *prog
    Load in a new program from the given file. Either use the file descriptor, or the file name. If prog=1, load remotely and return lifeline. Else return the image number of the new core image.
int userstart(codeseg,arg,plink,disp)
    Start a process in the given image with the given argument. Initialize it to have the given parent link. Return a lifeline to the new process.
int userkill(lifeline)
    Send a KILL note to the process pointed to by the lifeline.
userremove(codeseg)
    If the caller owns the code segment, and the reference count is Ø, reclaim the storage for the segment. This routine is in "schedule.u".
uload(fname,startloc,length,fd)

    char *fname; char **startloc;" int *length;
    Loads the named file (which must be in Unix a.out format). If startloc = Ø, it is a fresh file, and startloc will be set to a new area to report where the file was put. Length will return the size in words of the text portion of the file. If startloc is anything else, then it is assumed that only the data area is to be read, starting where ever startloc requires. Fd is a link to a file system process. This link is closed, even on failure.

4.7 <u>Timing</u>

Each process has an array of TIMESLOTS (currently 5) entries that record recent CPU usage. This array stored in the process table (Section 4.1) as "pptime". Each entry refers to an interval of one second, and the number stored there is the number of 1Ø,ØØØths of a second that the given process ran during that interval. Information is only retained for the previous TIMESLOTS seconds. These arrays are indexed by the variable "timeslot". Just before the scheduler starts a process, it records the current time in "starttime" to the nearest 1Ø,ØØØth of a second. When the process returns to the scheduler, that start time is subtracted from the current time, and the result is added to the appropriate entry in the timing array.

The module "timing.u" contains routines for gathering timing statistics for processes. The routine "nexttime" is called from the clock (Section 3.8) every second. This routine increments "timeslot" and clears the array entry associated with the new second for each non-running process. For the currently running process, "nexttime" adds the increment that has been used in the current second into the old timeslot before advancing the index. It also puts the current time in "starttime" so that if the process should return to the scheduler during the current interval, the scheduler will add the correct increment to the timing array.

The routine "userdsp", an alias for "display", may be invoked by a process to get timing statistics about another process. The target process is named by a link (Section 5.3). If it is on a different machine, "userdsp" returns an error. If not, "userdsp" totals the time used in the last TIMESLOTS-1 complete seconds and returns the percentage of the time (from 0 to 100) that the target process used the CPU during that interval.

<u>Files</u>
    timing.h, timing.u

<u>Data Structures</u>
TIMESLOTS
    The number of seconds of information stored for each process, currently 5.
int timeslot
    An index into the pptime array in each ppnode.
long int starttime
    when the current job started

<u>Procedures</u>
nexttime()
    A second has elapsed. Increments timeslot mod NUMSECS and clears the pptime word in each process. Special action for the current process: Add remnant of current second into its timing before incrementation, and change starttime.
int userdsp(linknumber)
    Called by a user process. Returns the fraction of the last TIMESLOTS slots that was used by the process at the destination of the linknumber. This answer will be an integer between 0 and 100. Returns -2 if the destination does not exist or is on a different machine. Causes usererror -1 if the given link is bad.

5. MESSAGES

All communication among processes is carried out through the medium of messages. Three major modules in Arachne deal with message handling. The module "line.u" (and the small module "route.u") deal with messages sent to foreign sites or arriving from foreign sites. The module "message.u" contains the service calls "receive" and "send". The central message-handling module is "kmess.u".

5.1 <u>Central message handling</u>

The module "kmess.u" maintains a pool of unused message buffers each of which has slots for the message text, the message length (from 0 to MSLEN), routing information (destination, channel, code), a note field that indicates the purpose of the message (for example, DATA or DESTROYED), an enclosed link (Section 5.3), and a pointer field used for linking unused buffers into a queue. All access to the pool of message buffers is protected by locks (Section 3.6).

Message buffers can be acquired by invoking the routine "getkmesg" and released by "rlkmesg". The former routine takes a priority argument. If the priority is 1, then any available buffer is returned. If the priority is 2, then a buffer is only given if there are at least 1/4 of the original buffers free. If the priority is 3, then a buffer is only given if there are at least 1/2 of the original buffers free. These distinctions are used to implement flow control. The callers to "getkmesg" are in the other two message modules.

Each process has a queue of messages waiting for it, arranged in the order they arrive. This queue is protected by locks (Section 3.6). Messages are placed on the queue by the routine "sendit", which is invoked by both the other message modules. This routine places the message on the appropriate queue if its destination is local to this machine. (The destination is a process id, which includes the machine id. A process id that has lower byte Ø is interpreted to refer to the kernjob, whatever its proper process id may be.) If the destination is no longer alive, the message is discarded. If the destination is on a foreign machine, then "sendit" finds the appropriate line on which to send the message by calling "getline" in the module "route.u", then calling "sendblock" in the module "line.u" to ship it off (Section 5.2). The routine "sendit" recognizes one special case: If the "note" field of the message to a local process is the code "KILL" or "HALT", then instead of delivering the message, sendit invokes "killoff" in module "schedule" with argument TOKILL or TOHALT. (Section 4.5. HALT is not currently used.) In the case of KILL aimed at a non-running process, the "diemesg" mechanism that provides owners of its links with meaningful destruction notices does not work; since "diemesg" is global and "sendit" may run at interrupt level, "diemesg" is not used.

The dual to "sendit" is "waitmess", which is invoked by module "message.u" to get a message from the queue for a process.

The caller specifies a set of channels. The routine "waitmess" will return the first message on the appropriate queue whose channel is one of those specified. The caller also specifies a timeout period. If the timeout is Ø and no appropriate message is waiting, then "waitmess" returns failure. If the timeout is negative and no message is waiting, the routine "waitmess" calls "schedcall(SLEEP)" to allow other processes to carry on. Every time "sendit" deposits a message in a process queue, it invokes "awaken" to inform that process. If the destination was not asleep, then "awaken" has no effect. If the message was not the one expected, then the process returns to sleep. Eventually, the process that is sleeping for a message will be awakened and will be able to continue. ("Schedcall" and "awaken" are described in Section 4.5.) If the delay is positive and no appropriate message has yet arrived, then "waitmess" uses the clock routine "setalarm" (Section 3.8) to awaken the sleeping process after that amount of time. If an appropriate message arrives first, the alarm is turned off. When the alarm rings, "waitmess" is awakened, discovers that no message has arrived, and returns failure to the caller. The routine "waitmess" uses "time" in the module "clock.u" to distinguish between an alarm and the awakening that accompanies the arrival of a message.

The length field on the message dictates how many characters are copied from it into the process' buffer.

In some cases of user error, the module "message.u" must give back a message it has taken. In this case, "giveback" is invoked to put it at the head of the queue.

When a process is killed, "killoff" in "schedule.u" calls "kmclear" in "kmess.u" to remove any message that might be queued up for the process that is dying. The buffers occupied by the messages are reclaimed.

Files

kmess.h, kmess.u

Data Structures

```
struct kmesg{
      int kmnext;  /* links together free list */
      int kmlength;  /* number of bytes of message in kmbody */
      int kmdest;   /* destination process id */
      int kmcode, kmnote, kmchan;
      struct link kmlnenc;  /* enclosed link */
      char kmbody[MSLEN]; /* MSLEN defined in lnmaint.h */
      }
     One kernel message buffer.
int kmesgsize
     Size of a kmesg in words.  Set at Arachne initialization.
struct kmesg *kmesgbuf
     Initialized to kmesgbuf[NUMKMES].
struct kmesg *kmbufavail
     Head of available list.
int numkmes, warn1, warn2
     Number of buffers left, 1/2 the original number of  buffers,
     1/4 the original number of buffers.
NUMKMES
     Original number of kernel message buffers available
NUMWTMES
     Number of waiting messages that can  be  held  before  users
     take them.
struct kmesg kmwthed[NUMPROCS]
     Headers for each process into a list of kmesgs  waiting  for
     receipt, each linked through the kmnext field.
```

Procedures

```
struct kmesg *getkmesg(priority)
     Returns a pointer to a free kmessage buffer.  The meaning of
     priority  is  described  above.  Returns 0 on failure (if no
     buffer is available at this priority).
rlkmesg(kmess) struct kmesg *kmess
     Returns the kmessage buffer to the free buffer pool.
kminit()
     Initializes all tables for kmess.u.
sendit(kmess) struct kmesg *kmess
     Sends a message to the destination indicated in the message.
     If the message looks foreign, try to route it through an ap-
     propriate neighbor.
int waitmess(who,chans,delay,kmgot) struct kmesg **kmgot
     Wait for a message for user number "who" on any of the chan-
     nels  indicated in the mask "chans".  Delay is a maximum de-
     lay (in seconds), after which a return of -1 is given if  no
     message  was received.  A delay of -1 indicates no time lim-
     it.  Kmgot is a result parameter, set to point  to  the  re-
     ceived kmessage buffer.
giveback(kmess) struct kmesg *kmess
     Place the indicated message back at  the  head  of  incoming
     messages for the current user (curusr).
kmclear(who)
     Remove any incoming messages on the queue for process number
     "who".
```

5.2 Inter-machine Messages

The modules "line.u" and "route.u" provide low-level communication with other machines. "Route.u" associates physical lines with processor id's by the routine "getline". The actual handling of physical lines is in "line.u".

"Line.u" keeps tables for each physical line indicating the current state of the communication. Input and output lines are completely independent. For input lines, the information stored includes the address of the device registers, the state of the transmission (idle, waiting for a free buffer, receiving the message size, and receiving the message body), a pointer to the message buffer holding the incoming message, and a count of how many bytes are left to transmit. Output lines carry much the same information in addition to a queue of message buffers awaiting transmission. No acknowledgement or checksums are used, since experience has shown that the lines are very reliable. All access to the outgoing message queues is protected by locks (Section 3.6). All the queues are locked simultaneously for simplicity.

During initialization, this module determines which physical lines to neighbors exist by attempting to read the status register for each line. If the register does not exist, the processor traps, and this trap is caught by "setflag" in "lowestirp.s". This routine sets a flag that is examined by line initialization.

The routine "sendblock" places a given message buffer on the appropriate output queue and calls "outfrob". "Outfrob" signals "linehandle" by clearing and setting the output interrupt enable bit on the appropriate line. This toggling is performed at high priority. If the line is currently not engaged in output, this action will cause an output interrupt. If it is, then an output interrupt will occur in any case as soon as output is finished.

Input interrupts on DRV-11 lines are dispatched to "inhandle", and output interrupts to "outhandle". (See Section 3.2 on interrupt handling.) These routines take an argument that indicates which line caused the interrupt. Interrupts are serviced based on the current state of communication on the particular line that caused an interrupt.

An output interrupt on an idle line causes "linehandle" to examine the queue of outgoing messages on that line. If it is not empty, then the first is picked and readied for sending. An input interrupt on an idle line causes "linehandle" to prepare to receive a foreign message. It calls "getkmesg" in "kmess.u" with priority 3 to find a message buffer in which to place the incoming message. If this request fails, then "linehandle" does not accept the first word of the incoming message. In this case, the

machine trying to send the message will not be able to use the line until the recipient machine is able to find a buffer and reads the header word from the line. The procedure "frobin" is used by "rlkmesg" in module "kmess.u" to cause input interrupts on those lines waiting for a message buffer whenever a buffer becomes available. Once the header word has been accepted, the sending side of the line sends the entire message buffer. During the bulk of the transmission, both "inhandle" and "outhandle" try to stay locked in the interrupt routine in order to use the physical line at peak efficiency. If the receiver fails to receive at a reasonable rate, then the sender dismisses the interrupt, but will come back when the next transmitted word has been read.

<u>Files</u>

    line.u, route.u, and line.h

<u>Data Structures</u>

```
int linetab[NUMMACH] (in route.u)
     Linetab[n] is the number of the line to machine n.
struct drvblock {
     int drcsr; /* common status register */
     int droutbuf; /* output data buffer */
     int drinbuf; /* input data buffer */
     int drfiller; /* not used */
     }
     The structure of a block of registers pertaining to one phy-
     sical DRV-11 (parallel-word) interface to another LSI-11.
int numnbrs
     Number of physical links to neighboring LSI-11's.  Set dur-
     ing initialization.
struct inblock {
      int *indr;  /* the drv block address */
      int instate;  /* place in protocol for this DRV-11 line */
      struct kmesg *incurmess;  /* where o put incoming stuff */
      int *inptr;  /* points within incurmess */
      int inwdcnt;  /* number of words left to transfer */
      } ininfo [NUMNBRS]
struct outblock {
      int *outdr;  /* the drv block address */
      int outstate;  /* place in protocol for this DRV-11 line */
      struct kmesg *outcurmess;  /* kmess currently in transit */
      int *outptr;  /* points within outcurmess */
     int outwdcnt;  /* number of words left to transfer */
     struct kmesg *iooutqhead, *iooutqtail;  /*  head,  tail  of
    output queue, linked through kmnext field.
     } outinfo [NUMNBRS] Data indicating the state of one DRV-11
    line.
struct intervect {
     int *outnpc, outnps, *innpc, innps; /* new pc  and  ps  for
    output and input interrupts */
     } Interrupt vector block for one DRV-11 line.
```

<u>Procedures</u>

```
routeinit()
    Initializes linetab.
int getline(machno)
    Returns the line corresponding to machine "machno".   Aborts
    on an invalid machine number.
char *irpvect(mach)
    Returns the address of  the  interrupt  vector  for  machine
    "mach".
char *drv(mach)
    Returns the address of the device register block for machine
    "mach".
irpinit()
    Initializes the states (ioblocks) of all lines.
sendblock(mach, block)  struct kmesg *block
    Attempts to send the message pointed to by  "block"  to  the
    neighbor whose machine id is "mach".  If there is no room on
    the output queue, sets an alarm to try again later, but  re-
    turns for now.
outfrob(dr)
    Causes an output interrupt on the DRV-11 line whose  address
    is "dr".
infrob()
    Causes an input interrupt on those DRV-11 lines in "waiting"
    state.
inhandle(machine)
    Services an input interrupt that occurred on the  line  from
    machine "machine".
outhandle(machine)
    Services an output interrupt that occurred on the line  from
    machine "machine".
```

5.3 <u>Links</u>

Each process has a table of <u>links</u> that are used to send messages. The table of links, "lntab", and all manipulations of links are handled in the module "lnmaint.u". A destination field

of 0 indicates that an entry is not in use.

A new link can be created by the service routine "lnmake", an alias for the service call "link". This routine returns a small number that the process can use to refer to this link; processes never have direct access to the link table. The new link is initialized with a destination pointing to the calling process, code, channel and restriction according to information provided by the calling process. The code and channel are provided to the process when it receives any message that comes along that link to help it identify the purpose of the message. The channel can also be used for selective receipt of messages. The restrictions are a set of permissions and actions that will govern the use of this link. The <u>owner</u> (the process that created the link) can either allow or prohibit the <u>holder</u> (a process that is given the link in order to send messages to the owner) giving the link away or duplicating it. The owner can request notification when the link is duplicated, given away, or destroyed. None of these restrictions applies as long as the link is PRISTINE, that is, was not received in a message or duplicated from another link. Notifications are special messages along the channel and code of the link with the unforgeable note field indicating what notification type it is. The restriction bits also indicate whether the link is a use-once resource (a REPLY link) or a permanent resource (a REQUEST link). New processes are given a link (always numbered 0) that their parent presented to the kernel. This link is restricted by NODESTROY, so the child may not destroy the link and fool the parent into believing that the child has terminated. Finally, the MAYERROR restriction bit allows error messages (Section 5.4) to be sent on the link.

Lifelines (Section 4.6) are like links, except that LIFELINE is set as a restriction. The owner cannot create a link with LIFELINE on or PRISTINE off. It is not possible to send a message across a lifeline.

After a link has been created, it can be given to another process as an enclosure in a message. Making a link can overflow the space in lntab for the process. Overflow can also result from receiving a link in a message. The routine "lnget", called to find a free space in this table, will expand the table if necessary. The initial allocation is added on each time an overflow occurs, up to a fixed limit. The table space is reserved from free space (Section 3.5).

The routine "lnclear" is called from "killoff" in "schedule.u". It sets the destination field of all links in the current process' link table to Ø, clearing them. If any link has the TELLDEST (tell upon destruction) bit set, then "tell(DESTROYED,diemesg)" is invoked. If any link has the LIFELINE bit set, then "tell(KILL)" is invoked to kill the destination process. If any link has MAYERROR but not TELLDEST set, then "tell(ERROR,"holder died")" is invoked to warn the destination process.

The service routine "lndestr", an alias for "destroy", is used to destroy links. It clears the destination field after sending any necessary notifications by using "tell". It refuses to destroy a link with NODESTROY set. ("lnclear" is the only way

to destroy such a link.)

The service routine "lnok", an alias for "linkok", returns Ø if the given link number is currently valid; otherwise a negative error code is returned.

<u>Files</u>
      lnmaint.u and lnmaint.h

<u>Data Structures</u>
```
struct link {
      int lncode;
      int lndest;  /* destination.  Ø if link not in use */
      int lnrestr;  /* restriction bits */
      int lnchan;  /* channel */
      }
struct lntentry {
      int lntsize;
      struct link *lntaddr;
      } lntab[NUMPROCS];
     A table of all links for all processes.
int NUMLINKS
     Number of initial links per process.
int MAXNUMLINKS
     Maximum number of links per process.
```

<u>Procedures</u>
```
int lnmake(code, chan, restr)
     Make a new link, with destination pointing to the calling
     process, with the given channel, code, and restrictions.
lnclear(userno,mesg) char *mesg;
     Remove all links for this user.  If any have the TELLDEST
     restriction, send a DESTROYED notification to the destina-
     tion, with "mesg" as the body.  If any has the KILLABLE res-
     triction, kill the destination with a KILL notification.  If
     any has MAYERROR but not TELLDEST, warn the destination with
     an ERROR notification.
telldup(dup,pln) struct link *pln;
     If the given link has TELLDUP or TELLGIVE restrictions, and
     dup is DUP or NODUP, respectively, then a notification is
     sent along this link.
lncopy(to,from)
     Copies the link, setting PRISTINE off.
int lndup(ulink)
     Duplicate the given link, and return the index of the new
     one.  If the link has the TELLDUP restriction, send the
     DUPPED notification to the owner.
int lndestr(ulink)
     Destroy the given link, but return an error code if not pos-
     sible.  If the link has the TELLDEST restriction, send the
     DESTROYED notification to the owner.
```

```
int lnfree(userno,ln) struct link **ln
     Return the index and address of a free  link  in  the  given
     user's link table, if possible.
int lndecode(alink,ln) struct link **ln
     Check to see if "alink" is  a  valid  link  number  for  the
     current user.  If so, return a pointer to the link table en-
     try via the result parameter "ln".  Otherwise, return -1.
klnmake(ln) struct link *ln
     Makes an entry in the caller's link table that contains this
     link.  This procedure is used by kernjob to forge links.
lninit()
     Initialize all tables for lnmaint.u.
int lnget(who,where) struct link **where;
     Returns the link number of an unused link for process "who".
     Sets  "where"  to  point  to the link itself.  Increases the
     size of the process' link table if necessary, but not beyond
     MAXNUMLINKS.  Returns -1 for failure.
int lnstart(who)
     Sets up an initial link table for process  "who"  with  NUM-
     LINKS  links,  all  with  null  destinations.  Returns -1 on
     failure.
```

5.4 User messages

The user interface to message passing is contained in the module "message.u". The service routines in this module extensively check consistency of arguments given by calling processes. For example, pointers to memory are checked against "outrange" in module "lsi.c". If any errors are found, the service routine first undoes any work it may already have performed (which may involve returning a message buffer via "giveback" in "kmess.u") and returns to the caller by using the macro "USRERROR" (Section 3.3).

The service routine "sendumes", an alias for the user service call "send", checks that all the actions desired by the caller are in consonance with the permissions of the link across which the message is to be sent. If all is well, "sendumes" acquires a message buffer from "getkmesg" in "kmess.u". The prior-

ity argument is 1 if the message is traveling on a reply link and 2 otherwise. (Replies are more likely to be actively awaited by their destinations than are requests, which may build up.) If "getkmesg" refuses to allocate a message buffer, then the calling process waits for one in a "schedcall(CONTINUE)" loop inside "trygetkmesg" in "message.u". Once a message buffer is available, the contents of the caller's message (if any) are copied into the message buffer, along with the destination field extracted from the link along which it is being sent and any link to be enclosed. Only as much of the message as specified in the length field is copied into the kernel message buffer. If the disposition has ERROR set and the carrier link has MAYERROR, then the note field of the message buffer is set to ERROR; otherwise, it is set to DATA. The message buffer is then given to "sendit" in "kmess.u" to direct the message toward its recipient. The link on which the message was sent and the enclosed link are then removed from the sender's link table if this action is called for by the restrictions on the carrier link and the disposition arguments, respectively. If the restrictions on these links dictate, notifications are sent to their destinations.

All notifications are sent by the routine "tell" in "message.u", which immediately calls "telling", which takes as arguments the destination, channel, code, and notification type. If "telling" cannot get a message buffer at priority 2, it sets an alarm on the clock to try again in one second. By the time the alarm rings, the original link along which the notification is to be sent may have disappeared, but the necessary information from it is stored as the arguments to "telling".

The dual service routine "recumesg", an alias for "receive", is used to receive messages. The caller indicates which collection of channels to use and where to put the contents of the message when it arrives. The message body is placed in a location specified by the caller; other information is placed in a "urmesg" structure provided by the caller. This structure contains fields for the length, the note, and the channel and code describing the link used. The calling process can request a timeout after which the call should fail if no message has arrived. This argument is passed directly on to "waitmess" in module "kmess.u". After the contents of the message have been copied into the process data area, the message buffer is reclaimed with "rlkmesg" in "kmess.u". If the note on the message was ERROR, then an exception is raised (Section 6.2), but otherwise the receipt is successful.

<u>Files</u>

    message.u and message.h

<u>Data Structures</u>

```
struct urmesg
      /* what a user receives */
      int urlength, urcode, urnote, urchan;
       /* length of body, user-defined  code,  system-given  note
     (DUPPED, DESTROYED, GIVEN, ERROR, DATA) */
       int urlnenc;  /* enclosed link index into usr linktab */
       };  /* end of struct urmesg */
```

<u>Procedures</u>

```
tell(who,what,mesg) struct link *who; char *mesg
     Send a message over link "who"  with  note  "what",  one  of
     DUPPED,  DESTROYED,  ERROR,  GIVEN,  and KILL.  Uses routine
     "telling".  In case of DESTROYED and ERROR, "mesg" is passed
     to "telling" as well.
telling(dest,chan,code,what,mesg) char *mesg;
     If can't send a note immediately, set an alarm to try  again
     in a second.
mscopy(to,from) char *to, *from
     Copy body of message.  If from = 0, fill "to" with nulls.
int sendumes(ulink,elink,data,length,disp) char *data
     Send message "data" with length "length" over the link  num-
     bered  "ulink" in the current process' link table.  If elink
     is the index of a valid link in the  current  process'  link
     table,  create a link in the destination process' link table
     equivalent to it.  Remove elink from  the  sending  process'
     link  table  if disp does not have DUP set.  The message has
     note DATA unless disp has ERROR set.  Return 0 for  success,
     a negative error code for failure.
int recumesg(chans,data,urmess,delay) struct urmesg *urmess
     Receive a user message on the channel combination  specified
     by  "chans".  Place it into the user-provided buffer "data".
     Extra information (length of message, channel,  code,  note)
     is  placed in "urmess".  If delay >= 0 and no message is re-
     ceived in "delay" seconds, fail.  Return 0  for  success,  a
     negative failure code for fail.  The timeout failure code is
     -3.  If the note of the received message is ERROR, terminate
     the caller.
struct kmesg *trygetkmesg(priority)
     Call getkmesg(priority) until a message buffer is  obtained.
     Perform "schedcall(CONTINUE)" while waiting.
telldup(disp,pln) struct link *pln
     If disp has the DUP bit set, tell owner of link pln  DUPPED;
     otherwise,  tell  owner  GIVEN.   In latter case, remove pln
     from table by setting its destination to 0.
```

5.5 <u>Asynchronous</u> <u>Message</u> <u>Receipt</u>

A process may arrange for messages arriving on certain channels to be asynchronously received as soon as they arrive. This arrangement is established by a call to "usercatch", which is an alias for "catcher". The arguments indicate the channels that are to be treated specially, the address of the procedure that is to be called asynchronously, and addresses of data structures into which the message is to be placed when that procedure is called. If the address of the catcher procedure is Ø, then instead of adding the given channels to the active set, those channels are removed from it.

"Usercatch" establishes asynchronous receipt by initializing several fields in the "ppnode" structure in the process table (Section 4.1). The address of the catcher is placed in the "ppcatch" field, the message contents address in "ppcdata", and the urmess address in "ppurmess". The field "ppallcatch" is set to hold a bitstring naming all the channels that are active for asynchronous receipt. Only one catch procedure may be specified at any time; a new specification overrides the old one.

Whenever "sendit" in "kmess.u" delivers a message, it checks to see if it should be caught asynchronously instead of delivered. If so, the process target process is either currently running or not. If it is running, then "docatch" in "message.u" is called to dispatch the message to the catcher routine. "Docatch" sets "curlevel" to 1 to imply that an interrupt-level routine is running. This value prevents the catcher procedure supplied by the process from using most of the service calls. The "awaken" service call is the only exception; it makes a special check to insure that it is not used during a catch. The catcher procedure itself is called without arguments. If it returns TRUE, then the channel remains active; otherwise, the channel on which the message arrived is deactivated for asynchronous receipt.

If the target process is not currently running, then the

"ppcpend" field in the process table is set to indicate that the next time the process is run, "docatch" should be called. The last argument to "strtuser" in the scheduler is the value of the "ppcpend" field. If it is TRUE, then "strtuser" will call "maycatch" in "message.u" before continuing the process from its previous place. This procedure calls "docatch" for each message on the input message queue that should be received asynchronously.

```
Procedures
int usercatch(chans,data,urmess,caproc)
      char *data; struct urmesg *urmess; int *caproc;
     If "caproc" is Ø, removes the given channels from the set
     that are active for asynchronous receipt. If "caproc" is
     not Ø, then the channels named will be activated for catch-
     ing messages. In addition, "data", "urmess", and "caproc"
     become the new standard way to treat messages that are
     caught.
docatch(kmess) struct kmesg *kmess;
     Called if this message is one that the current user wished
     to catch. Calls the user's catch routine, and either dis-
     ables or re-enables the channel for catching depending on
     how that routine returns. During the catch routine, set
     "curlevel" to 1 to prevent most service calls from being
     used. The awaken service call is a special case, treated
     there.
maycatch()
     Sees if any message on the input message queue for the
     current process fits any of the catchers. If so, invokes
     the appropriate catcher for each one.
```

6. INTERRUPTS AND EXCEPTIONS

Processes can inform the kernel that they wish to handle their own interrupts. Likewise, exceptions that are raised may be handled by the offending process.

6.1 Interrupts

The module "interrupt.u" contains the routines that accomplish interrupt dispatching. A process can call the service routine "userhandler", an alias for "handler", naming an interrupt vector, the address of the routine that should service interrupts arriving through that vector, and a channel number. The routine "userhandler" sets the interrupt vector to jump at high priority to a routine in "lowestirp.s", one of "uint0", "uint1", up to the number of allowed interrupt handlers. Each of these routines places an argument (one of 0, 1, ...) on the stack and calls the routine "uinterrupt" in "interrupt.u". This routine therefore receives an argument telling which of the possible interrupts has happened. It then calls the appropriate user-supplied interrupt routine. When that routine finishes, "uinterrupt" returns, and the interrupt is dismissed from "uint0" or whichever one it came through.

While the process is handling an interrupt, it may invoke the service routine "userawaken", an alias for "awaken", which takes no arguments. This routine, in "interrupt.u", causes a message to be directed to the process that created the handler. The correct process is found through the variable "curhandler", which is set by "uinterrupt" when the interrupt is dispatched. The message is on the channel specified by the process when it invoked "userhandler", with note INTERRUPT. The code is 0, and the body of the message is empty.

No other service calls may be called by a process interrupt

handler. Such calls will be aborted by "sys" in "crtl.s".

```
Files
     interrupt.u and lowestirp.s

Data Structures
int NUMHDLS
     Number of handlers that may exist.
int numhdls
     current number of handlers in existence
struct hdlnode {
      int (*hdentry)();  /* entry point of user handler */
      int hdchannel;     /* channel for awaken calls */
      int *hdvector;     /* vector for the interrupt */
      int hddest;   /* destination for  awakens.   0  means  this
     handler not in use */
      };
struct hdlnode hdlset[NUMHDLS], *curhandler
     Set of handlers, current handler in force.
int *lowuint[]
     In lowestirp.s:  pointers to individual uint routines

Procedures
intclear(procid)
     Clears all interrupt vectors owned by  this  process.   This
     operation is part of killing a process.
userhandler(vector,entry,channel) int *vector, *entry
     Service call to set up a handler at given vector, with given
     entry   point.    The  channel  is  used  in  an  associated
     "userawaken" call.
uinterrupt(hindex)
     Called from "uint?" in "lowestirp.s" when an  interrupt  oc-
     curs.  Dispatches to appropriate handler.
int userawaken()
     Service call to be used  only  from  an  interrupt  handler.
     Causes a notification to be sent to the process that created
     the handler.
uintinit()
     Initialization of the interrupt table.
```

6.2 Exceptions

Exceptions are raised when a process makes a service call that cannot be completed because the arguments are bad. Service call routines use the macro USRERROR when the caller has made a mistake. This macro calls "usererror" in "error.c", which prints a message and sets "curerror" to 1.

Also, messages may be sent that contain error notifications. If the "dup" argument to "sendumes" has ERROR set, and the link along which the message is to be sent has MAYERROR in the restriction bits, then an error message is sent. Receipt of an error message (by "recumes" in "message.u", Section 5.4, but not by asynchronous receipt, Section 5.5) will also set "curerror" to 1.

After any service routine completes, control is returned to the caller through the wormhole at "sys" in "crtl.s". This code checks "curerror" to see if the service call completed normally. If not, then the exception can have one of two effects: If the caller has taken no special precautions against exceptions, then the wormhole calls "userdie("exception not caught")", terminating the process.

If the caller has arranged for exceptions to be caught, then the exception handler provided by the process earlier is called. The arguments to this procedure are the returned value of the service call, the service code, and then the arguments to the service call during whose execution the exception was raised. The exception handler may take any action it wishes; when it returns, control passes to the point to which the service call would have returned. Any value that the exception handler returns becomes the visible result of the service call.

A user process may request that all exceptions be caught in this fashion by invoking "uerrhandler" in "schedule.u", an alias for "errhandler". This procedure takes one argument: the address of the error handler procedure. If it is 0, then "uerrhandler" removes any existing error handler. The address of the current error handler is stored in the "pperror" field of the process table entry for each process (Section 4.1). The wormhole discovers this address by calling "gerrhandler" in "schedule.u".

Files
```
     schedule.u, error.c, crtl.s
```

Procedures
```
char *uerrhandler(addr) char *addr;
     Establishes a new error handler.  Returns  the  old  handler
     address.
char *gerrhandler()
     Called from "crtl.s".  Returns the current error handler.
```

7. ACKNOWLEDGEMENTS

The authors are pleased to acknowledge the assistance of Professor Sun Zhong Xiu, visiting scholar from Nanking University (Peoples Republic of China), as well as students Jonathan Dreyer, Jack Fishburn, Will Leland, Paul Pierce, and Ronald Tischler. Their hard work has helped Arachne to reach its current level of development.

8. REFERENCES

[1] M. H. Solomon and R. A. Finkel, "ROSCOE -- a multiminicomputer operating system," Technical Report #321, University of Wisconsin--Madison Computer Sciences (September 1978).

[2] M. H. Solomon and R. A. Finkel, "ROSCOE: a multimicrocomputer operating system," Proceedings of the Second Rocky Mountain Symposium on Microcomputers, pp. 291-310 (Au-

gust 1978).

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

[4] R. L. Tischler, R. A. Finkel, and M. H. Solomon, "Roscoe User Guide, Version 1.0," Technical Report #336, University of Wisconsin--Madison Computer Sciences (September 1978).

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

[6] R. A. Finkel, M. H. Solomon, and R. Tischler, "Arachne User Guide, Version 1.2," Technical Report #379, University of Wisconsin--Madison Computer Sciences (February 1980).

[7] R. A. Finkel, M. H. Solomon, and R. L. Tischler, "Roscoe Utility Processes," Technical Report #338, University of Wisconsin--Madison Computer Sciences (February 1979).

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

[9] B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice-Hall (1978).

[10] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," Communications of the ACM 17, 7, pp. 365-375 (July 1974).

[11] R. A. Finkel and M. H. Solomon, "The Roscoe Kernel, Version 1.0," Technical Report #337, University of Wisconsin--Madison Computer Sciences (September 1978).

[12] DEC (Digital Equipment Corporation), Microcomputer Handbook (second edition). 1976.

[13] D. E. Knuth, The Art of Computer Programming Volume 1--Fundamental Algorithms (second edition), Addison Wesley (1973).