THE ROSCOE KERNEL

by

Raphael Finkel
Marvin Solomon

Computer Sciences Technical Report #337

October 1978

THE ROSCOE KERNEL

Version 1.0
October 1978

Raphael Finkel
Marvin Solomon

Technical Report 337

Abstract

Roscoe 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 Roscoe 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 Roscoe project, present the implementation details of the utility processes, and display Roscoe from the point of view of the user program.

TABLE OF CONTENTS

THE ROSCOE KERNEL

Version 1.0 -- October, 1978

1. INTRODUCTION

The Roscoe project at the University of Wisconsin is implementing a distributed operating system for several co-operating LSI-11 microcomputers [Solomon 78a, 78b]. Documentation for programmers writing code to be run under Roscoe can be found in a companion report [Tischler 78a]. Roscoe 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 tasks of the kernel are storage management, process management, and message management. The tasks of the utility processes are terminal handling, file management, interactive command line interpretation, and resource allocation.

This report describes the Roscoe operating system at the level of detail necessary for systems programmers who might be modifying the kernel. Utility programs are documented in a compainion report [Tischler 78b].

All code for Roscoe is written in the C language [Kernighan 78] and in the Unix [Ritchie 74] 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 Roscoe 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.

1.1 Source Files

The source files for the Roscoe kernel 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 extension "u" indicates that the file is written in C and should be compiled with the stack-limit-checking compiler. The extension "c" indicates that the file is written in C and should be compiled with the normal C compiler. The extension "s" marks files written in assembler language. The extension "h" indicates header files that are meant to define data and structures and are included (by the file switch "include") in several C files. Each module is compiled separately; the result has extension "o". Files ready to be loaded on the LSI machines have extension "lda"; the linked collection of all Roscoe files is placed in "roscoe.lda".

The source files for utility processes are in the directory /usr/network/roscoe/user. These files are all of the "u" and "h" variety.

Many library routines are available in the library "libr.a"

in the user directory. The source for these routines is in /usr/network/roscoe/library.

1.2 Compilation and linking

The kernel can be compiled and linked by invoking "lprep". This shell 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 shell file "lprep" then calls the linker to combine the various object files. Finally, "lprep" invokes "mklda" to convert the Unix link file format into absolute loader (lda) format, producing "roscoe.lda".

2. FUNDAMENTAL MODULES

The kernel source code is mostly found in eleven "u" files. In addition, there are a few "c" and "s" files. Each file begins with an extensive comment indicating 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. If there is some doubt about the accuracy of cref, it is easy to find all uses of a particular function, say "foo", by us-

ing the "grep" program:

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

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

2.1 Initialization

To load a new version of Roscoe into a cleared LSI-11, one needs to employ 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 0. 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) [DEC 76]. 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 built-in absolute loader, give ODT the command "173000G", which causes the machine to run a diagnostic program in read-only memory starting at location 173000. This program prompts with "$". The command "AL<carriage return>" to the diagnostic program starts the absolute loader. To cause "coml" to send the software loader, type "<control-T>S". The program will prompt with "file = ", to which one responds "drload.lda<carriage return>". When this file has been loaded, it begins execution with the message "Loading".

Alternatively, the following bootstrap procedure may be used: When the LSI-11 is being reloaded, unless the power has been

turned off or the last program destroyed memory, "drload" will be present. To invoke it, type "157000G" to ODT. The message "Loading" should appear immediately. One way to send the kernel to the LSI-11 is to type "<control-T>F". When "coml" prompts for the file, type "roscoe.lda<carriage return>". After the file has been loaded, Roscoe should start. Another way to send a file to the LSI-11 avoids "coml" altogether. A file, say "foo", may be sent to a machine, say 7, across the fast line by "cp foo /dev/dr7".

The machine id must be set by hand in location 157700. If the machine id is wrong (it is printed along with the startup message), halt Roscoe, set it, and then restart. The machine id is placed at a location that is not damaged when a new version of Roscoe is read in by "drload", but it is destroyed if drload itself is reloaded.

It is possible to restart Roscoe at any time by halting execution ("coml" command <control-T>B) and starting at location 700 (ODT terminal command 700G). When Roscoe is started, a small amount of code in module "crtl.s" causes a Unibus reset (which clears 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) 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 initiates and awakens the kernel job "kernjob" in "lsi.c". Then "startscheduler" in module "schedule.u" is called. The kernel job is soon started; it executes a manual load request (see "Core Images", section 3.2) to load the first process (usually, the Resource Manager). After the kernel job starts the first process, it remains in a loop waiting for kernel-to-kernel load requests, described in section 3.2.

Files

lsi.c and kernkern.h

Procedures

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.

2.2 Debugging Aids

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

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 Roscoe kernel discovers a situation from which it cannot recover, the routine "syserror" in module "error.c" is invoked. 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 contin-

ue 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 "schedcall(DIE)" in "schedule.u" is called to terminate the offending process.

Service routines return failure to the calling process by invoking the macro USRERROR, which is defined in "util.h". This macro causes the error number to be printed and a negative error code to be returned. The procedure "usererror" in "error.c" prints the message. A list of errors can be found in the file "usererrors".

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.

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 "procprint" to print the status of each process, "kmprint" to print a kmesg buffer, "lnprint" to

print a link, "clkqprnt" to display the clock queue, and "free-print" to print a map of free space.

Files

lowestirp.s, error.c, debug.u

Data structures

int debug
: At location 776. Can be set through ODT to control debugging.

int numkmes
: At location 756. Stores the number of available kernel message buffers.

int curusr
: At location 760.

int wormno
: At location 774. Stores the value of register 1 the last time a service call went through the "wormhole".

Procedures

pause(code)
: Print the code on the terminal and halt.

snap(code)

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)".

2.3 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 157702 (octal), which can be set in ODT. The delay is to slow down the natural rate of transfer so that if the program "coml" 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 [Tischler 78a, 78b]. The module "lsi.c" defines "putchar", which calls "outchar" in "io.c".

Files

io.c

Procedures

outchar(dev,c) char c;

Print one character to the teletype at address dev.

ttyflush()

Remove any waiting character from the teletype input buffer.

2.4 Free storage management

Free storage management routines are found in the module "free.u". During Roscoe 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 [Knuth 68] is used to allocate chunks of storage through the routine "freeget" and to return them through the routine "freerel". Free storage is used to find room for certain kernel tables that are made once during Roscoe initialization and to provide room for processes. This latter space is reclaimed by a reference count technique described under Core Images, section 3.2.

Files

free.u

Data Structures

int freesize

Length of the free space in words

int *freespace

Start of the freespace. Set by sbrk(freesize) at initialization.

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.

Procedures

freeinit()
Free storage initialization

int *freeget(size)
Returns a block of "size" words of free storage. Returns -1 for error.

freerel(ptr) int *ptr
Release the block of storage pointed to by "ptr".

2.5 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 moves to high priority. (The LSI-11 has only two priority levels.) The "signal" routine restores the original priority (which may have been high). No interrupts are serviced betwen these calls. Four queues are locked in this fashion and will be discussed below: the ready process queue, free message buffer queue, received message queue, and outgoing message queues.

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

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.

2.6 Service calls

Processes running under Roscoe request kernel assistance by invoking kernel service routines. These invocations are performed by loading a routine code in register R1 and transferring control to location 1002 by a subroutine call instruction. The routine "sys" in the module "crtl.s" resides at this location. It decodes these calls and invokes the appropriate kernel routine. This "wormhole" technique allows user programs to communicate with the kernel without being linked to it. Each service call is represented by two words in a table: 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 5.) 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 3.1.) When the service routine is finished, control is returned directly to the calling process. 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.

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 [Tischler 78a].

Files

crtl.s

Data Structures

table

Branch table for dispatching service calls.

Procedures

sys

At location 1002. Dispatches service calls.

2.7 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; This routine will run at interrupt level. 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 when the next scheduled event has been reached. A queue of events stores the alarms that have been set; each node includes the time to the next event in the queue. The queue is not allowed to become empty; at least a "tick" event is always present.

The tick event occurs once a second to update the various time indicators. It also calls "setalarm" with a special argument of delay = -1, placing another tick on the queue one second after the first event on the queue, rather than one second from

the current time. (The first event on the queue is the tick currently being serviced.)

When the clock interrupts, "timesup" is called at interrupt level to act on one or more events on the queue. Each such event is carried out before updating the queue, so that the queue is never empty. (In particular, there's always a "tick" event present.) The algorithm tries not to lose any time on the clock while the action is being carried out: the clock stays in repeat interrupt mode, and the time for which it was last set is saved in "lasttime", so the time elapsed while servicing an interrupt can be calculated. "Timesup" may carry out several such actions before returning, if their times arrive while previous ones are happening.

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" works by returning "timeofday" plus the fractional part obtained by summing the "expire" fields in the clock queue up to the "tick" event. These services are used by outgoing message sending and message reception, which are discussed in sections 4.2 and 4.4.

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 */
         };
int timeofday
        interval timer in seconds
long datereg
        seconds since the epoch (beginning of 1973, Madison stan-
        dard time)
int uniquecode
        used to distinguish alarm events
struct clkqnode {
         struct clkqnode *cqnext;
         long expire; /* time in .0001 seconds between this event
                     and the following one */
         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
        queue.
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 ac-
        tion will be taken. Returns a code to be used for "tur-
        noff".
turnoff(code)
        Remove the alarm that has the given code.
tick()
        Called every second to update datereg and timeofday.
timesup()
        Called by clock interrupt to run the pending events.
clockinit()
        Initialization of the clock routines.
```

3. PROCESS MANAGEMENT

Processes are managed by the scheduler and the core image routines.

3.1 The scheduler

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, and an index into a reference count table for purposes of storage reclamation.

Each process is identified by two numbers: an index into the process table (process number) and a network-wide unique identifier (process id). The process id is a combination of the machine id and a sequence number. The module "schedule.u" provides routines "getid" and "getno" for conversion between the two. Process id's 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.

A process is started by a call to "initiate", which takes a core image, an argument, and an owner id. 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. 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".)

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 calls the routine "strtusr" (again in module "resume.s"), which switches to the process stack and transfers control to that process. The stack limit for this process is placed in location 1000 (octal) so that procedure entry code can check for stack limit violations. 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.) The routine "psset" in module "lowestirp.s" is used to switch the processor priority. Processes are run at low priority.

The details of process switching depend on the standard subroutine linkage conventions of C. 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 pointer. 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 preceeding 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
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 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.)

The contents of r6 on entry to "cret" are irrelevant; on return from a procedure, r6 always points to the word following the word addressed by r5 during the procedure. 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 r5 in a location associated with the current process (the field "ppfptr" in the per-process data structure "proctab[curusr]"), replaces it by the saved r5 from the scheduler "process", and returns. Similarly, "strtusr" resumes the 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.

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 first case, "killoff" is invoked and another process is chosen at the start of the scheduler loop. In the second case, the process remains ready but is awakened so

that it sits at the end of the ready queue. 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. In the third case, the process is placed in state "sleeping". Forms of schedcall(DIE) and schedcall(CONTINUE) are provided for processes in the service routines "userdie" and "usernice", aliases for "die" and "nice".

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. If "killoff" is to terminate the process, it then invokes "intclear" to remove any interrupt handlers associated with the target, "freerel" to return the target's stack, "lnclear" in "lnmaint.u" to clear the target's link table, "kmclear" in "kmess.u" to remove any messages awaiting the target, and then it reduces the reference count that records how many processes are active in the target's core image. 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" will change 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.

Both "awaken" and the scheduler loop use locks as described above to prevent misuse 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 insures that the scheduler will have 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.u, schedule.h, 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 nextid
Non-reusable id for next process to be initiated

int kernno
process number of kernel job; set by lsi.c

```
struct ppnode {
         int ppfptr;  /* stack frame pointer */
         int ppstatus;  /* One of the following:
          PPNONEX 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. */
         };
```

struct ppnode proctab[NUMPROCS]
per-process data; indexed by procno's

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.
struct codentry{
         int cdrefcount; /* reference count (-1 if not in use) */
         char *cdmemory;  /* start address of this segment */
         int cdowner;  /* process id of owner */
         }
int NUMMODULES
        Length of core image table.
struct codentry codetab[NUMMODULES]
        The core image table.
```

Procedures

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. Halt or terminate the target process. If the process is running, set the status to PPTOKILL or PPTOHALT. If the process gets terminated, remove its stack, link table, waiting messages, and interrupt handlers.

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

int getfreeno()
: Finds an available user number.

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

int initiate(codeseg,arg,owner)
: Makes a new process entry in pptab. 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 core image is placed in the pptab for future reference. The process is left in state "ready".

fallthrough(p)
: Called if process returns. Effects a schedcall(DIE).

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()
: This service call is invoked by the kernel call "die()" and effects "schedcall(DIE)".

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

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)
: In "resume.s". Starts the process whose frame pointer is given. The scheduler stack pointer is saved in fptr.

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.

3.2 Core images

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 4.3, and the file system is described elsewhere [Tischler 78b].) 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. The routine "uload" uses a privileged service call to acquire free space for the new load image. It returns the starting address of

the loaded program.

Core images are maintained in a table that associates an area in memory with an owner and a reference count of processes active in that area. The routine "newcseg" in "schedule.u" takes an address and an owner, finds a free slot in the segment table, sets the reference count to 0, establishes the owner, and stores the address as the memory for that core image. The service routine "userload" establishes a code segment for the new image by calling "newcseg" and returns the index of the code segment that "newcseg" provides.

The calling process 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 dead processes to start up again later. Starting a process is accomplished by the service call "userstart" (an alias for "startup"), which takes a code segment, and 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). 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 4.3.) 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 dying. (The parent can thus be furnished an unforgeable notification of the child's termination.) A lifeline

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 LIFE-LINE. 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 0 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. The process identifier of the kernel job is always a number with the machine id in the upper byte and a 0 in the lower byte. 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 whom 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.

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 0.

Files

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

Procedures

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,dup)
: 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 0, reclaim the storage for the segment. This routine is in "schedule.u".

4. MESSAGES

All communication among processes is carried out through the medium of messages. Three major modules in Roscoe 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 interface between message routines and the service calls "receive" and "send". The central message-handling module is "kmess.u".

4.1 Central message handling

The module "kmess.u" maintains a pool of unused message buffers each of which contains the message text, other user-accessible fields, the process id of the destination, 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 2.5).

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 left. If the priority is 3, then a buffer is only given if there are at least 1/2 of the original buffers left. 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 2.5). These 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.) 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. (See section 4.2, "Inter-machine messages".) 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 3.1, "The scheduler").

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 0 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. 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 3.1, "The scheduler".) If the delay is positive and no appropriate message is waiting, then "waitmess" uses the clock routine "setalarm" (section 2.7) 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.

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.u and kmess.h

Data Structures

```
struct kmesg{
        int kmnext;  /* links together free list */
        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 system initialization.

```
struct kmesg *kmesgbuf
```

Initialized to kmesgbuf[NUMKMES].

```
struct kmesg *kmbufavail
```

Head of available list.

```
int numkmes, warn1, warn2
```

Number of buffers left, half 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 kmwtlst{
        /* entry in a linked list of kmesg'es waiting to be re-
        ceived */
        struct kmesg *kmwtptr;  /* the waiting kmess */
        struct kmwtlst *kmwtnext;  /* next for this user */
        }
struct kmwtlst kmwttab[NUMWTMES]
```

Table of waiting messages.

```
struct kmwtlst kmwthed[NUMPROCS], *kmwtavail
```

Headers for each process into kmwttab, head of available

list of kmwtlst's.

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 appropriate neighbor. (See section 4.2, "Inter-machine Messages".)

int waitmess(who,chans,delay,kmgot) struct kmesg **kmgot
: Wait for a message for user number "who" on any of the channels indicated in the mask "chans". Delay is a maximum delay (in seconds), after which a return of -1 is given if no message was received. A delay of -1 indicates no time limit. Kmgot is a result parameter, set to point to the received kmessage buffer.

giveback(kmess) struct kmesg *kmess
: Place the indicated message back at the head of incomming messages for the current user (curusr).

printkm(mess) struct kmesg *mess
: For debugging purposes.

kmclear(who)
: Remove any incomming messages on the queue for process number "who".

4.2 Inter-machine Messages

The modules "line.u" and "route.u" deal with communication to 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 (idle, awaiting acknowledgement, sending data, receiving data, awaiting checksum, sending checksum), a pointer to the message buffer currently being sent or received, a pointer within that buffer to the active word, how much information is left to transfer, the current

checksum, and a queue of messages waiting to be sent on that line. All access to the outgoing message queues is protected by locks (section 2.5). All the queues are locked simultaneously for simplicity.

During kernel 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". This routine sets a flag that is examined by line initialization.

The routine "sendblock" places a given message buffer on the appropriate queue and signals the interrupt-level routine "linehandle" by using the routine "frob". If the queue is too full, then "sendblock" places an alarm on the clock queue to try again after one second and returns. "Frob" signals "linehandle" by clearing and setting the output interrupt enable bit on the appropriate line. 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 it is finished.

The routine "linehandle" is called whenever an input or an output interrupt occurs in any one of the lines. The interrupt itself is caught by a routine in "lowestirp.s" (see section 5, "Interrupt Handling") and dispatched to "linehandle" with arguments indicating which line was involved and whether the interrupt was on input or output. 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" responds to the foreign site with a negative acknowledgment. Otherwise a positive acknowledgement is sent. If "linehandle" is trying to send a message and receives a negative acknowledgement, then the outgoing message is not removed from the queue. Instead, an alarm is set to call "frob" again in a second. Once the acknowledgment has been sent and received, the sending side of the line sends the entire message buffer and follows it with a checksum. During the bulk of the transmission, "linehandle" attempts to stay at interrupt level in order to use the physical line at peak efficiency. If the receiver fails to receive at a reasonable rate, then the sender leaves interrupt level, but will come back when the next transmitted word has been read.

Files

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

Data Structures

```
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
        physical DRV-11 (parallel-word) interface to another
```

LSI-11.

int numnbrs
Number of phyical links to neighboring LSI-11's. Set during initialization.

```
struct ioblock {
 int iostate; /* eg IOWANTCKS */
 struct kmesg *iocurmess; /* kmess currently in transit */
 int *ioptr; /* information to transfer (points within a kmesg) */
 int iowdcnt; /* length of information left to transfer */
 int iochksum;
 struct kmesg *iooutqhead, *iooutqtail; /* head, tail of output queue, linked through kmnext field.
 } ioinfo [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.
```

Procedures

routeinit()
Initialize linetab.

int getline(machno)
Return the line corresponding to machine "machno". Abort on an invalid machine number.

int irpvect(mach)
Return the address of the interrupt vector for machine "mach".

int drv(mach)
Return the address of the device register block for machine "mach".

irpinit()
Initialize the states (ioblocks) of all lines.

sendblock(mach, block) struct kmesg *block
Attempt 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, set an alarm to try again later, but return for now.

frob(dr)
Cause an output interrupt on the DRV-11 line whose address is "dr".

linehandle(machine, irpkind)
Service an interrupt that occurred on the line from machine "machine". Irpkind is OUTIRP or INIRP.

4.3 Links

Each process has a table of links that are used to send messages. The table of links and all manipulations of links are handled in the module "lnmaint.u". This table, called "lntab", uses a destination field of 0 to indicate that an entry is not in use.

A new link can be created by the service routine "lnmake", which is an alias for the service call "link". This routine returns a small number that the process program 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 recipient of any message that comes along that link. The channel can also be used for selective reception of messages. The restrictions are a set of permissions and actions that will govern the use of this link. The owner (the process that created the link) can demand that no holder (a process that is given the link in order to send messages to the owner) be able to give the link away or duplicate it. On the other hand, these actions may be permitted. The owner can request that it be notified when the link is duplicated, given away, or destroyed. Notifications come as 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 RE-

PLY link) or a permanent resource (a REQUEST link). A special restriction LIFELINE is used to give the holder the capability to terminate the owner, that is, it makes the link into a lifeline (section 3.2).

After a link has been created, it can be given to another process as an enclosure in a message.

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 0, clearing them. If any link has the TELLDEST (tell upon destruction) bit set, then "tell(DESTROYED)" is invoked. If any link has the LIFELINE bit set, then "tell(KILL)" is invoked to kill 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".

Files

lnmaint.u and lnmaint.h

Data Structures

```
struct link {
        int lncode;
        int lndest;  /* destination.  0 if link not in use */
        int lnrestr;  /* restriction bits */
        int lnchan;  /* channel */
        }
int NUMLINKS
        Number of links per process.
struct link *lntab
        Initialized to hold NUMPROCS*NUMLINKS entries.
```

Procedures

```
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)
        Remove all links for this user.  If any have the TELLDEST
        restriction,  send a DESTROYED notification to the desti-
```

nation. If any has the KILLABLE restriction, kill the destination with a KILL notification.

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 possible. 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 entry via the result parameter "ln". Otherwise, return -1.

prntln(ln) struct link *ln
For debugging purposes.

lninit()
Initialize all tables for lnmaint.u.

4.4 User messages

The user interface to message-passing is contained in the module "message.u". The service routines in this module do extensive consistency checking 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 2.2).

The service routine "sendumes" is an alias for the user service call "send". This routine 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 priority 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 it by executing a "schedcall(CONTINUE)" 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. 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 situation. 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" is used to receive messages; it is an alias for "receive". The caller indicates which collection of channels to use and where to put the contents

of the message when it arrives. The repository for the message is a "urmesg", which contains fields for the message text, 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".

Files

message.u and message.h

Data Structures

struct urmesg
: What a user receives.

struct usmesg
: What a user sends. These structures are described in full in the Roscoe Users Manual.

Procedures

tell(who,what) struct link *who
: Send a message over link "who" with note "what", one of DUPPED, DESTROYED, GIVEN, and KILL. Uses routine "telling".

telling(dest,chan,code,what)
: 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,usmess,dup) struct usmesg *usmess
: Send message "usmess" over the link numbered "ulink" in the current process' link tablee 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 dup = FALSE. Return 0 for success, a negative error code for failure.

int ksend(dest,elink,usmess,dup) struct usmesg *usmess
: Send message "usmess" to process id "dest". This function is only callable by the kernel process. Otherwise, it is similar to sendumes.

int recumesg(chans,urmess,delay) struct urmesg *urmess
: Receive a user message on the channel combination specified by "chans". Place it into the user-provided buffer "urmess". If delay >= 0 and no message is received in "delay" seconds, fail. Return 0 for success, a negative failure code for fail. The timeout failure code is -3.

```
struct kmesg *trygetkmesg(priority)
        Call getkmesg(priority) until a message buffer is ob-
        tained.  Perform "schedcall(CONTINUE)" while waiting.
telldup(dup,pln) struct link *pln
        If dup = TRUE, tell owner of link pln DUPPED;  otherwise,
        tell  owner GIVEN.  In latter case, remove pln from table
        by setting its destination to 0.
```

5. INTERRUPT HANDLING

Processes can inform the kernel that they wish to handle their own interrupts. The module "interrupt.u" contains the routines that accomplish the 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, etc.) 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 interrupt routine in the process data space. 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 occurs. 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. ACKNOWLEDGEMENTS

The authors are pleased to acknowledge the assistance of Jonathan Dreyer, Jack Fishburn, James Gish, Frank Horn, Michael Horowitz, Will Leland, Paul Pierce, Ronald Tischler, and Milo Velimirovic. Their hard work has helped Roscoe to reach its current level of development and will be essential in completing its design and implementation.

REFERENCES

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

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

Knuth, D. E., The Art of Computer Programming, Vol. 1, 1973.

Solomon, M. H., Finkel, R. A., Roscoe: A Multi-Microcomputer Operating System, University of Wisconsin -- Madison Computer Sciences Department Technical report #321, April, 1978.

Solomon, M. H., Finkel, R. A., "ROSCOE: a multi-microcomputer operating system", Proceedings of the Second Rocky Mountain Symposium on Microcomputers, August 1978, pp. 291-310.

Tischler, R., Solomon, M., Finkel, R., Roscoe Users Guide, University of Wisconsin -- Madison Computer Sciences Department Technical report #336, 1978.

Tischler, R. L., Finkel, R. A., Solomon, M. H., Roscoe Utility Processes, University of Wisconsin -- Madison Computer Sciences Technical Report #338, September 1978.

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