Solo32:
A Concurrent Pascal Operating System
with UNIX Interfaces

by

Martin Wilde

B.S., Kansas State University, Manhattan, Kansas, 1983

-------------------

A Master's Report

submitted in partial fulfillment of the
requirements for the Degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1984

Approved by:

Richard A. McBride
Major Professor
I wish to thank Dr. Richard A. McBride for all the help and guidance that he has given me throughout my project and report. Additional thanks go to my committee members for their useful knowledge of the operating system area. Special thanks to Mr. Robert Young for his expert knowledge of the Solo system and Concurrent Pascal.
TABLE OF CONTENTS

Chapter 1. Introduction ........................................... 1
  1.0 Overview .................................................... 1
  1.1 Motivation ................................................ 1
  1.2 Background ............................................... 3
  1.3 Report Organization ..................................... 11

Chapter 2. Implementation Background ............................ 12
  2.0 Overview of Solo ......................................... 12
  2.1 History of Solo at KSU .................................. 19
  2.2 Reestablishment of Solo .................................. 21
  2.3 Implementation Goals .................................... 25

Chapter 3. System Implementation ................................ 27
  3.0 Introduction ............................................. 27
  3.1 Sequential Programs ..................................... 28
      3.1.0 The Unix32 Program ................................ 28
      3.1.1 The CallUnix32 Program ........................... 33
      3.1.2 The Diskspace Program ............................. 34
  3.2 Kernel Modifications ................................... 34
      3.2.0 CallUnix32 Interface ............................... 34
      3.2.1 Solo Reinitialization .............................. 35

Chapter 4. Conclusion ............................................ 38

References ....................................................... 41

Appendices
  User Guides .................................................. 44
  Pas32 and Cpasca1 Shells ..................................... 50
  Solo32 Program Source ....................................... 55
  Sequential Pascal Programs Source ........................... 89
  Cpasca1 Kernel Source ....................................... 125
  Cpasca1 Runtime Library Source .............................. 160
  Solo System Files ............................................ 196
LIST OF FIGURES

1.0 Solo Implementation Under the UNIX operating System . .5
2.0 The Data Flow of the Solo Operating System ..........16
2.1 The Solo File System ....................................18
3.0 The Process of Solo Reinitialization ....................37
CHAPTER 1
INTRODUCTION

1.0 OVERVIEW

Solo is an operating system that is written entirely in Concurrent Pascal. Application programs are written in Sequential Pascal and are utilized by Solo to support various functions of an operating system. With a few additional functions Solo can be expanded to a useful operating system. The following report contains a description of an implementation of Solo which allows the user to have access to the UNIX file system and environment. The UNIX Ver.7 operating system with Berkley 4.1 enhancements is currently running on a 32-bit Interdata 8/32 at the KSU Department of Computer Science.

1.1 MOTIVATION

Many people may ask the question isn't Solo a dying system? The answer is no because Solo is a powerful enough operating system to provide most features that a single user might need. It is small enough so that a person studying operating systems can understand the whole system in a short amount of time. As a measure of complexity one can compare Solo with UNIX or CMS in terms of the number of lines of source code. The UNIX Kernel is about 7000 - 10,000 lines of C code and CMS is over 100,000 lines of PL/1. The original version of Solo required approximately 1300 lines of Concurrent Pascal and small
Sequential Pascal programs to handle various functions. For
the person studying operating systems or the system
programmer, Solo has many other features as cited by Brinch
Hansen[1]:

(1) Less than 4 percent of it (Solo) is written in
machine language. The rest is written in Sequential and
Concurrent Pascal.

(2) In contrast to machine-oriented languages, Pascal
does not contain low-level programming features, such as
registers, addresses and interrupts. These are all
handled by the virtual machine which executes compiled
programs.

(3) System protection is achieved largely by means of
compile-time checking of access rights. Run-time
checking is minimal and is not supported by hardware
mechanisms.

(4) Solo was the first major example of a
hierarchical concurrent program implemented by means of
abstract data types (classes, monitors and processes).

This small size and added modularity of the Solo system allows
for ease of expansion and modifications. The majority of
expansions can be made using Sequential Pascal programs. New
functions can be added to the Concurrent Pascal section
because of Solo's structure.
1.2 BACKGROUND

Solo is implemented as an operating system that runs on top of the UNIX operating system. In this way, the UNIX system views Solo as just another user's program executing. This is conceptually the same as the virtual machine facility (VM) of IBM which enables each user to view his computing as one independent of other users. This has the effect of allowing multiple virtual computers to be created from one real computer[IBM]. Figure 1 illustrates the virtual machine environment in which Solo executes. The UNIX operating system creates a virtual machine (VM1 of Figure 1.0) environment for running systems and programs. UNIX can be described as a virtual machine monitor (VMM1) which provides the virtual machine with three essential characteristics[PG]:

(1) The virtual machine monitor (VMM) provides an environment for programs which is essentially identical to the original machine. (2) Programs run in this environment show at worst only minor decreases in speed. (3) The VMM is in complete control of system resources.

An operating system can be thought of as a VMM because it is a control program. Goldberg[PG] cites that a control program consists of three modules: the dispatcher, the allocator, and the interpreter. The dispatcher is a top level control module whose task is to decide which underlying module is dispatched to perform an operation. The second control module, the allocator, decides what system resources are to be
provided to the VMs which the VMM controls. In the case of a single VM, this module's task is to keep the VM and VMM separate. With multiple VMs, its task is to control the use of system resources. The purpose of the last module, the interpreter, is to simulate any privileged instructions for the VM.

The Solo operating system - VM1, which runs on top of UNIX, can be decomposed further in terms of Goldberg's notation into VMM2 and VM2. The Concurrent Pascal portion and the Sequential Pascal programs form VM2. The Kernel and Run-Time Library form a VMM2 because they handle system requests from the programs and map their virtual devices into real devices.

The Concurrent Pascal portion and the Sequential Pascal programs can be further broken down into VMM3 and VM3. The Job and I/O processes form the dispatcher module of VMM3 which handles requests from Sequential Pascal programs. These requests are then passed onto the other routines to be executed. Since there are several processes executing concurrently in Solo, the concurrent portion allocates resources to prevent them from being accessed by processes at the same time. The interpreting of calls from Sequential programs is performed to handle privileged commands such as the execution of another Sequential program or I/O requests. Thus, the Sequential programs which run on the Solo operating system, which in turn utilizes the Kernel and the Run-Time Library represents a virtual machine which runs on top of the
Figure 1.0 Solo Implementation Under the UNIX Operating System.
virtual machine monitor (UNIX).

The design of the virtual machine is crucial in determining the portability of programs written for it. Most early operating systems provided very rudimentary virtual machines for user jobs. These virtual machines performed little more than a partitioning of the real machine resources. The interface presented to the user closely resembled the actual hardware interface. Indeed, for various reasons many operating systems still present such interfaces. Some of these interface drawbacks are[GP]:

(1) Software written for these rudimentary virtual machines is not easily transported to a machine with different interfaces.

(2) Such facilities, though possibly adequate for user jobs, are not directly suited to the implementation of a hierarchy of one or more operating system modules, which require in addition the ability to control and communicate with other modules or programs.

Solo overcomes the drawback of low-level commands because it is written in Concurrent Pascal with its job control language being Sequential Pascal. When the transportation is made to another machine only the lowest level features of the Concurrent Pascal language must be interfaced. The facilities for Sequential program communication and control are readily available in Solo because the purpose of it and its Kernel is
to implement the execution of Sequential programs, communication between programs, and the interfaces for input and output mechanisms to the underlying system.

The use of an operating system that utilizes modules to provide features to the user has developed an important concept of hierarchical structuring. If a system is developed which utilizes hierarchical ordering of system components, the design and testing stages are simplified. Simplification in the designing of new modules exists because the designer does not have to know how the lower level structures work, only how to use them[RP]. As mentioned, the simplification of testing is a feature of systems in which hierarchical ordering is employed. Programs can be tested component by component from the bottom up. In this manner, underlying components can be assumed to be functionally correct when testing the components at the next level. At the bottom of the structure is components which do not use any other components, while at the top is those components which no other components use. Thus, programming errors within new components can be pinpointed to the new additions since the interfaces are well-defined procedures that have already been tested.

One of the first operating systems to use the concept of hierarchical structuring was the RC 4000[87]. The main goal of this system was to structure the system nucleus so that extensions could be added at later times to accommodate the addition of new modules to the system. The RC 4000 system
made use of the concept of concurrent processes to control the flow of data through the system. Processes could be in one of three states: (1) ready, in which the process is on a ready list or queue awaiting its time slice for execution, (2) waiting or blocked, a state that a process is placed in when the process is waiting for an external event or condition, and (3) running, a state in which the process has the CPU and is executing.

Process synchronization was accomplished by the use of monitors. The monitor coordinated events between processes and granted access rights to system variables and devices. Message passing monitors allowed processes to communicate with each other. The message passing mechanism also served as a basis for process synchronization. A process could be queued or delayed until it received a message, acted upon it and sent a reply to the original sending process releasing it. Dynamic process creation and deletion was provided so processes could be created at run time. Utilizing this, they could be constituted to perform a function then destroyed once the task was performed. With this feature the requirement of extendability of the system was met. Whenever a new module needed to be added to the system a new process was created to perform that particular task.

The use of processes to provide a hierarchical structure was also used in the T.E.E. multiprogramming system[RM]. E.W. Dijkstra developed the system to demonstrate the use of
concurrent processes. Processes communicated with other processes by means of system variables that were shared in a mutually exclusive fashion by the system's processes. Before a process could perform an operation on a shared variable the process had to first test a semaphore to determine the state of the variable. This concept of implementing mutual exclusion on system variables is the same underlying concepts that is used to implement the monitor facility in Brinch Hansen's RC 4000 operating system mentioned previously. The T.H.E. system employed a process hierarchy to control the use of system devices. At top levels, the hierarchy processes were only able to perform basic operations on a device. The actual interface or operations on the system device was performed at the lower levels of the system. The use of a process hierarchy enhanced the system's ability to protect the integrity of devices. The integrity was implemented by only allowing processes at the top levels to perform a limited number of operations on a device and processes at the lower levels, which interacted with the actual device, were not limited in the operations they could perform. Thus, access rights could be passed from one layer of the hierarchy to another. The use of virtual devices or protection provided the illusion of protection.

Brinch Hansen developed the Concurrent Pascal language to provide a facility and an environment for the development of concurrent programs, namely operating systems. It allowed for features such as process creation and destruction, mutual
exclusion of system variables, and the ability of programs to be structured in a hierarchical manner. The resultant product that he produced has been used as a language for the implementation of concurrent systems. The underlying structure of systems written in his language was provided by a Kernel. Since the operating systems produced by the language were aimed at being a virtual machines, the Kernel concept was superior to that of using a Monolithic monitor. The reasoning for this is that the Kernel only handles the interprocess communication between a user's program and the underlying system. In contrast the Monolithic monitor performs both the operation of handling and processing a program's requests[HDJ]. Thus, the concept of an abstract machine is preserved in the use of the Kernel concept for Concurrent Pascal.

Although Concurrent Pascal has been around for many years, its use is still not widespread for system development. Its primary use has been that of a research tool and a language for small system development. Some of the reasons for this are that the language only allows static allocation of resources and access rights; the language is procedure oriented so the facilities for message passing are not implemented; and the limited synchronization primitives of the language has also been a drawback. These reasons, coupled with the size limiting restrictions placed on it by using the Kernel concept, have prevented its widespread usage.
1.3 REPORT ORGANIZATION

The intent of this project is to reestablish the Solo operating system and implement extensions on the original system to allow interfaces to the UNIX environment. The general background of operating systems and various implementations has been surveyed in Chapter 1. Chapter 2 provides an insight to the background of the Solo operating system and its structure. Necessary modifications that were performed in order to establish a working model of the Solo system are also described in Chapter 2. In addition to this are the goals and ideas used in the design and implementation of the expanded system. Chapter 3 presents the actual implementation details. Finally, Chapter 4 summarizes the results of this project and discusses possible enhancements.
CHAPTER 2

IMPLEMENTATION BACKGROUND

2.0 OVERVIEW OF SOLO

Before going into the implementation details of the Solo system, a brief overview of Solo and Concurrent Pascal will aid the reader. Concurrent Pascal extends Sequential Pascal with the additional features of concurrent processes, monitors, and classes. A Concurrent Pascal program consists of a collection of Sequential Pascal programs called processes. The processes consist of a data structure and statements that operate on it. The program statements are executed strictly one at a time. These processes may be executed concurrently. Processes communicate and synchronize with one another by means of sharing variables. These shared variables are known to the entire system and have an associated set of operations that can be performed on them. Message passing is usually done through buffers which are shared variables.

A Concurrent Pascal monitor can define a collection of variables or resources, and the operations that can manipulate it. The operations that can be performed on the shared variables are defined as entry points into the monitor. Each entry point is in itself a procedure which performs a
particular operation on the data structure. Implicit synchronization occurs because only one process at a time is allowed to be executed inside a monitor. Primitives such as Continue and Delay are used to explicitly control synchronization operations. If a process finds the monitor's variables are not in a desired state, then the process can block or delay itself until another process later changes the state of the variables and continues the delayed process.

The last Concurrent Pascal feature to be reviewed is the class. The class is a normal procedure which performs one operation. A class can only be initialized once. After initialization, its parameters and private variables exist forever. The class can only access its own temporary and permanent variables. These variables cannot be accessed by other processes or monitors. Since a class cannot be passed as a parameter to a process or monitor, it cannot be called simultaneously. Thus, the class is declared as a permanent variable within a system component.

All processes, monitors, and classes are represented as type declarations. Access rights to instances of types can be distributed. The type checking of the compiler is responsible for protecting the private variables of one process from that of another process. In that, the compiler checks that the process only accesses variables that are declared local to it and those that are passed to it. Thus, in Concurrent Pascal the access rights are static, not dynamic.
The Solo system uses these Concurrent Pascal primitives as its basis. Monitors are used to limit access rights to the system disk, operator's console, and other utilities. Processes make up the three major parts of Solo. These processes consist of the Job process, the Input process, and the Output process. The Job process performs the central processing and uses the Input and Output Processes to perform I/O which the job requests. A sequential program interacts with the process that executes it by means of entry points or calls to procedures defined within the process. These interface procedures and their parameter types are declared in the prefix section of the sequential program. The prefix enables the compiler to perform type checking of calls to the operating system. These procedure invocations or requests may range from reading a page from the disk, writing a line to the console, or invoking another sequential program. Through the use of the Job process and the other processes the system allows multiple events to be happening at the same instant.

Solo facilitates interprocess communication through monitors that provide message buffers. The arguments passed between processes may either be a device name or a filename. Figure 2.0 shows the mechanisms for data flow inside the system. Once a process receives an argument through the ARG buffers from another process, it sends back an argument indicating whether any errors occurred during the transaction. These buffers are implemented as monitors to provide a method of synchronization between processes. Another reason for
implementing them as monitors is that the information does not have to be stored on the disk and thus increases efficiency.

Various Sequential Pascal programs in addition to the Concurrent Pascal part, make up the Solo system. These include the Do program, the Io program, the File program, the Console program, and various other server programs. The Do program is first loaded by the Job process and executed to interact with the user's console. The Do program provides the necessary command parsing and validation of user's input. Once a command has been validated, the Do program passes the program's name and arguments to the Job process where it is retrieved and executed. The arguments to a program can be a filename, devices for input or output, or any other conceivable parameter. If the parameter is a device name, for example, the program can pass it through the ARG buffers and the program name is executed by one of the Io programs to perform I/O for its invoker. Programs may also be invoked by another program for example, to store a newly created file onto the disk. The Io program is initially loaded and executed by the I/O process. This program receives commands via the ARG buffers and executes requested I/O programs for the program that is currently running under the Do program.

The File program provides maintenance for the system's virtual disk. The File program may either be invoked from the terminal or by a program. The user can update or modify files on the system's disk through arguments such as: Create,
Figure 2.0 The Data Flow of the Solo Operating System
Replace, Protect, Delete, and Rename. In Solo, files are stored as contiguous pages, where each of the pages is 512 bytes long. The maximum length of a file is limited to 256 pages. Files are described by their name, type, protection, and disk address. All of this information is contained in the disk's catalog which is a special file that resides on the disk. The file's name may be up to 12 characters in length. The type field of the file describes the format. The file may be either scratch, ascii, seqcode, or concode. This typing feature allows the system to check that executed programs always have a type of seqcode, and the editing and displaying functions are restricted to ascii files. Once a filename has been looked up in the catalog (by using a hashing method), the file may be obtained by using the disk address as a pointer to the file's page map. The page map contains the length of the file and at what disk addresses the individual pages are stored. Figure 2.1 illustrates the file system and its page maps.

A program creates a new file by outputting data to a file called Next. This file is a scratch file with a length of 255 pages. Once it has been created, the File program renames the file to its desired name and a new instance of the Next file is created on the disk. The file system also has a file which contains unused disk pages and indicates the number of free disk pages. This file is used to place the file on the disk and to determine if sufficient room exists for the new file.
Figure 2.1 The Solo File System
One of the most commonly used Sequential programs is the Console program. It can be invoked to provide either input from the terminal or output to the terminal. The program is usually passed as an argument to a program that requires I/O from the terminal. As before, the program name (Console) is passed to the I/O process and executed by the Io program. Some of the other commonly used Sequential programs are the Printer program, which is invoked to provide output to a print device for a program; the Edit program, which when supplied with an input file (or device driver program) and an output file performs editing functions. The List program, which depending upon its invocation can either list the files contained in the catalog or display the entire contents of the cataloged files, also the Copy program can copy a file from one file to another or to a device. Together with these Sequential utility programs the Solo operating system is formed.

2.1 HISTORY OF SOLO AT KSU

The following paragraphs describe the evolution and uses of the Solo operating system at KSU. Entailed are how it was first used and for what purposes. Also included are the various problems that the single user system presented when operating on a multiuser system.

Solo was first implemented at KSU on an Interdata 8/32 mini computer. The original Brinch Hansen system which consisted of the Concurrent Pascal compiler, Kernel, Solo operating
system, and a Sequential Pascal compiler was first used to provide an environment for program development. Since the 8/32 offered a crude and low level operating system (OS/32-MT), the Solo system provided a substantial improvement for program development. Both Concurrent and Sequential Pascal were available in this environment, while they were previously unavailable. Because of the difference in the PDP-11/45 (the original machine on which Solo was implemented) being a single user system and the 8/32 being a multiple user system, Solo was implemented as a virtual machine. The Solo system disk was implemented as a virtual disk to recreate the characteristics of the disk which the PDP implementation assumed. The virtual disk was seen as a normal file to the 8/32 operating system (OS/32-MT). This use of the virtual disk concept required the Concurrent Pascal Kernel to be rewritten to provide the necessary OS/32-MT system calls to handle disk I/O requests from the Solo system. In addition to the changes for disk I/O, all supervisory functions were changed to use OS/32-MT system calls.

The use of Solo in this manner continued until a Sequential Pascal Compiler was developed which allowed programs to be compiled and used outside of the Solo environment. The compiler was developed by taking the first seven passes of the Solo Sequential Compiler and adding a machine code interpreter for the compiler's output. This was later modified to directly produce machine code to improve execution. The use of the prefix was still used and the routines to implement the
prefix calls were provided in a Sequential Run-Time Library. Once the Sequential compiler was implemented, a Concurrent Pascal compiler was developed. To provide a more user oriented and modifiable Kernel the Concurrent Pascal Kernel was rewritten in a high level language. Many of the machine dependent routines, such as set manipulations and external procedure linkage, were moved to a Run-Time Library for Concurrent programs. The Solo system was discontinued because reliable Sequential and Concurrent compilers were directly accessible under the operating system.

2.2 REESTABLISHMENT OF SOLO

Before the Solo system could be modified to provide UNIX interfaces, a number of modifications had to be performed to allow the porting of the system to the UNIX environment. Many of the modifications were reimplementations of Concurrent Pascal features that were no longer used when the current compilers and Kernel were developed. The necessary modifications that were made in the Run-Time Library and Kernel are summarized as follows:

(1) The attribute and the setheap commands were restored to provide their respective features for the Progstack monitor in Solo.

(2) The wait procedure was partially rewritten to only requeue a process that was calling it. This differs from Brinch Hansen's implementation in which a process
was blocked for 1 second. Since the Kernel is no longer time multiplexed, this feature could not be totally reimplemented.

(3) The provision for a file to be assigned at run time which had both read and write access was added to the Kernel. This was required for the use of the Virtual disk. Prior to this, only either read or write access could be assigned to a file assigned at run-time.

(4) The control operations on the console and disk were reimplemented to provide the ability to reinitialize the Solo system. The method in which this feature was restored is detailed in Chapter 4.

(5) To provide timing information the realtime procedure and attribute were implemented utilizing the UNIX time procedure in the Kernel. Although this modification does now provide time values, the actual values returned will vary from one run to another because of the load on the system.

The current Solo system differs from the original Brinch Hansen system[BH1] in various aspects. The methods in which the differences lie is highlighted in the subsequent facets:

(1) Since the Interdata 8/32 treated integers as 32 bit values, the Concurrent Pascal compiler reserved four bytes of storage for each integer. This required all integers (which were 16 bit values on the PDP-11/45) used
in I/O operations to be typed as shortintegers. In this manner the compiler only reserved two bytes of storage for them instead of four.

(2) Packed record declarations were used instead of regular record declarations to allow for full word alignment of record types. This was required on all record types that were passed during I/O operations.

(3) Because the UNIX operating system is line oriented for terminal operations during Solo operation, the cancelline and cancelchar features of the Typewriter class were removed. These were replaced by the normal character delete functions of the terminal.

(4) The original OS/32-MT implementation of Solo used two virtual disks, one for user programs and the other for system programs. This led to each of the original Solo disk routines being modified to check the iodevice parameter of the ioparam to determine which disk to access.

(5) Since allocation of data space for Sequential programs is not required, the memory reservation declarations of the Job and I/O processes were commented out.

(6) Both the Job and I/O processes include the added UNIX interface functions. These functions will be described later in Chapter 4.
Once these modifications were made, the Solo system was successfully compiled and executed. Before new Sequential programs could be added to the Solo system, they had to be compiled and relocated using the original Solo Relocator. The programs had to be compiled with the "-r" option so that the object code would be correctly located and that the linking phase of the compilation was not performed. The Relocator program performed the task of building a memory image file for the Sequential programs so that their prefix routines would be mapped to their respective Job process entry points. Before the Relocator program could be used, it had to be modified so that it would correctly compile and execute. The modifications done were a subset of those made to Solo, such as the packing of records and declaring integer variables used in I/O operations to be shortintegers.

The available Virtual disk did not contain a File program for Solo file maintainance. This required the File program to be copied from one of the original Solo distribution tapes. Many of the same modifications that had to be performed on the File program were like those done for the Relocator program. Once the File program was compiled and relocated, it had to be placed on the Virtual disk. This required a moderate amount of work because a method of moving files from UNIX to the Solo file system had not yet been established. Since an executable file on the Virtual disk was in the form of object code, an editor could not be used to move the code. The UNIX file copying command was used to copy the File program over an
existing Seqcode file on the Virtual disk. Once it was installed on the disk, it was renamed from the original name of the Seqcode file to the File32 program. After the File32 program had been established, the Solo system was able to function for program development.

2.3 IMPLEMENTATION GOALS

The use of hierarchical structuring and portability were the two main design considerations that were kept in mind when implementing the additional functions to the Solo system. Because of Solo's structure, the Solo nucleus provides only basic operations through the prefix to application programs. The addition of new prefix routines, their accompanying Job, and I/O processes entry points should leave this feature intact. In this manner the Sequential Pascal programs perform the bulk of the work and invoke the prefix routines only to perform a specific task.

If proper structuring is used in program development, the result is system portability. Portability in the simplest terms can be defined as the ability to move or transfer an object from one place to another with a minimum amount of work[RP]. Structured programs promote portability because application programs written at the higher layers of the hierarchy have the property of being virtually machine independent. This concept also applies to programs written for the lower layers. If the routines are written to perform one specific function, then their porting to another system
requires only understanding and recoding a small piece of the
system. This idea was used in the design of the operating
system Kernel for the THOTH system[DC]. The Kernel routines
were kept small to facilitate porting of the system to other
machines. An extension of this was used in the Solo Kernel,
for if the Kernel is moved to another machine only the UNIX
interfaces have to be rewritten. Thus, the device-specific
Kernel routines are very small and simple in function.

Through the use of structuring and the porting strategies
mentioned previously, the new additions to the Solo system
were designed. The next chapter details the actual programs
and modification details that were used in the project.
CHAPTER 3

SYSTEM IMPLEMENTATION

3.0 INTRODUCTION

This chapter discusses the implementation and general operation of the UNIX interface routines. Included with the functional description of the programs is a discussion of the Kernel and Run-Time Library modifications. Program procedures will not be discussed in detail whenever a procedure is trivial or internal program documentation is sufficiently detailed. The User's Manual (Appendix A) overviews the specific operating procedures for Solo and the Unix interface routines. Appendix B contains the modified Pas32 and Cpascal shells for the correct compilation of Sequential and Concurrent Pascal programs. The actual source code for the modified Solo operating system, Sequential Pascal routines, Kernel, and the Run-Time Library is listed in Appendices C through F. Appendix G contains a brief description of the program's UNIX filenames and how they are to be compiled if further modifications are to be performed.
3.1 SEQUENTIAL PROGRAMS

The Sequential Pascal programs Unix32 and CallUnix32 enable the Solo user to transfer files between systems and execute a command in the UNIX shell, respectively. Another Sequential program, DiskSpace, was developed to give the user an indication of the number of free pages on the Virtual disk. The following sections describe the Sequential programs in more detail.

3.1.0 THE UNIX32 PROGRAM

The purpose of the Unix32 program is to transfer files between the UNIX file system and the Solo file system. Before diving into the program's internals, a brief description of the UNIX file system will help to enhance the reader's understanding of the program's code.

The UNIX file system is accessed with the same commands as the Solo system, such as open, read, write and close, but the differences lie in the manner in which the files are actually accessed and how they are stored. Files in the UNIX system are stream-oriented rather than record-oriented as in most other systems[KR]. With stream I/O, data can be dealt with in any length the user desires. The file is normally terminated with an end-of-stream character. This marker is, however, invisible to the user. All file operations in UNIX are performed as function calls. These calls return a value which is used to determine whether or not the operation was
successful. Files are opened by passing a pointer to the filename and the access mode desired. The mode is a value designating read, write, or their combination. The value returned is called the file descriptor and is used for all subsequent operations on that file. If the value returned was less than zero, the desired file was not found.

The read and write functions are basically the same with the parameters being the file descriptor, a pointer to a buffer, and the number of bytes to be transferred. The number of bytes to be transferred can be virtually any value, but the most efficient occur with 512 byte transfers. A major difference in the two routines is found in the results returned. The read routine returns the actual number of bytes read from the file or a zero if the end-of-file (EOF) was reached. If a transmission error occurred during the read, a negative result is returned. The write routine also returns the number of bytes transferred, except that a negative value returned indicates that a transmission error has occurred.

The close function closes a file when given a valid file descriptor. The value returned indicates whether the file closure was successful. Besides the primitive file operations mentioned previously, two other file operations are provided. They are the create and access functions.

The create function creates a new file, given a pointer to the filename and the mode of access. Error values are returned if the file can not be created. The other command
provided is the access function. The access function allows the user to check a given file for the proper mode of usage (read, write, or execute). Although there are many other available file operations in the UNIX system, only the aforementioned ones are used in the Unix32 routine.

The input parameters to the Unix32 program are the operation type (INPUT or OUTPUT), file type (if the operation is INPUT), and the filename. The actual syntax of the commands are listed in the user's guide (Appendix A). The operation field is used as an indicator for the direction of file transfers with respect to the UNIX file system. An INPUT direction indicates that the file is to be input from UNIX to the Solo file system. The other direction, OUTPUT, indicates that the file is to be moved from Solo to the UNIX file system.

Once the program has been invoked, the command line is parsed to determine its legality and the type of operation to be performed. The input operation has a general format of: UNIX32(INPUT,Filename,FILEKIND). The second field of the input command line must contain an identifier type which indicates a valid array of 12 characters. The last field of the line must be a legal file type under Solo. This use of a file type field is so that the new file can be correctly cataloged. The output command line has the format of: UNIX32(OUTPUT,Filename). The output operation requires only one additional parameter - the filename. This field is
checked to validate that it is of an identifier type. While the command is being parsed an illegal field will cause the parsing to cease and an error is displayed on the console. The error message displayed informs the user of the correct invocation and the possible parameter types that the different fields can have.

Processing of the command begins after a correct command has been entered. Inputting a file is initiated by obtaining the free page set stored on the Virtual disk. The page set obtained is actually the page map of the Next file where the newly inputted file is stored until the FILE32 program actually creates the file and catalogs it. For the filename to be correctly recognized by the UNIX system, it must have a null character appended to the end of it. The access of the UNIX file is checked before the file is opened. The program is terminated and an access denial message is printed if the file does not have read access or is not found. Once the file is opened, it is read in blocks of 512 bytes until its EOF is reached. Each time a page is read, it is transferred to the Virtual disk. If at any time while the file is being read in and the length of the file exceeds 256 pages, the program terminates and displays an error message indicating that the maximum length of a Solo file has been reached.

After being successfully transmitted from UNIX to Solo, the file is automatically created and named by running the FILE32 program with the files parameters. A message is printed
identifying the run time error if the FILE32 program terminates abnormally. The UNIX file is closed after the FILE32 program establishes the newly created file. Finally, the input operation and the program are terminated.

The outputting of a file is basically the same as inputting a file except in reverse. The file is first looked up in the Solo catalog to obtain its page map. If the file is not located in the catalog, an error message is printed and the program terminated. Once the file has been located and its pagemap obtained, the filename is then converted to a UNIX filename as described before. If the attempt to open the file in the UNIX system fails, a new file is created and given read and write access. A check is also made while the file is being opened to determine if the user has write access. If they do not, an error message is printed indicating the denial of write access and the program is terminated.

Data is transferred page by page from the Solo file to the UNIX file once it has been successfully opened or created. This continues until the Solo EOF is reached. At this point the last file block is transferred to UNIX and the file is closed. Because the UNIX file system does not employ an EOF character directly, the procedures for the input and output of Solo files contain code that attaches an EOF marker to a file imported to Solo and the deletion of the Solo EOF marker on files that are exported to the UNIX file system.
The output of the Unix32 program can consist of various error messages. These messages are used to inform a user of illegal operations or errors resulting from the requested file transfer operation. Once an error is detected, the user is prompted and the program is terminated. The User's Guide (Appendix A) contains a complete listing of the error messages that can arise from the use of the Unix32 program.

3.1.1 THE CALLUNIX32 PROGRAM

The CallUnix32 program provides the Solo user with the ability to execute a routine in the UNIX shell. Since most of the interface to the UNIX environment is provided in the Kernel, only the general operation of the routine will be described in this section. Details of the Kernel are described later in this chapter. The program prompts the user for the shell command so that no parameters are passed to the program at invocation time. Once the program commences execution, the user is prompted for input by the displaying of a dollar sign ('$'). The prompt is to indicate to the user that his/her next command will be executed in the UNIX shell. After the command is executed, the user is returned to the Solo environment and the program is terminated.

The other output of the program is in the form of messages indicating in which environment the user is able to execute a command. A listing of the messages as well as command invocation details is provided in the User's guide (Appendix A).
3.1.2 THE DISKSPACE PROGRAM

The Solo file system includes a file that provides information on the number of free disk pages that exist on the Virtual disk. This file is used by the Diskspace program to inform the user of the number of unused pages. The program expects no input arguments. Once execution begins, the program consults the aforementioned file and obtains the number of free pages on the disk. The output of the program is the number of free pages and a header to inform the user of this information. A complete syntax description is provided in the User's guide (Appendix A).

3.2 KERNEL MODIFICATIONS

The following sections list the major modifications and components that were added to the Concurrent Pascal Kernel. The descriptions provided are general in nature. More specific details of the code to implement the functions can be found in the Appendix containing the Kernel listing.

3.2.0 CALLUNIX32 INTERFACE

The Kernel code for the UNIX interface employs the "exec1" function to execute a file or, in this case, the shell. The use of this command is similar to that described in the UNIX programming guide[RR]. The parameters to the exec1 function are the file's name to be executed and the arguments. Upon execution of the exec1 call, the existing program is overlayed and the new program is run. No method is provided for
returning to the calling program because of the program overlay. This drawback was overcome by creating another process (using the fork primitive) and assigning the exec function to it. In this way the calling program or parent forked a new process and then waited for that child process to terminate.

3.2.1 SOLO REINITIALIZATION

The reinitialization feature, as described before, was reimplemented to allow the user to reinitialize and execute Solo. The new implementation differs from the original code[BMH] in which the Solo program was reloaded from disk and then executed. Instead, the Kernel is reinitialized and the program is restarted. This modification was required because the present Kernel does not have the capabilities of directly loading a file from disk and then executing it.

The Loader Process of Solo performs the sequence of operations to reboot the system. The process uses control operations on the console and the disk to initiate the rebooting. The control operation on the console delays the calling process until the operator pushes the BFL key. A control operation on the disk then performs the reinitialization of the program. These control operations had to be reestablished for the console and disk devices. Figure 3.0 illustrates the sequence of events that occur when the system is rebooted. The first event is a control operation being performed on the console. As mentioned, this delays the
calling process. It remains blocked until the Kernel I/O routine intercepts the BRL character. At this point the LoaderProcess is removed from the blocked queue and placed on the ready list. When the process is finally given its turn to execute, a control operation is performed on the disk. The control command is received by the Kernel which allows it to reinitialize itself and the Solo program.
Figure 3.0 Reinitialization of the Solo System
CHAPTER 4

CONCLUSION

In this report we have explored the past development of Solo and have discussed recent modifications made to this operating system. We have also touched upon why it provides a valuable tool for program development and the study of concurrent processes. The main goal of the project was to port as many features of the original KSU version of the Solo system (which ran under OS/32-MT) to the UNIX environment. To take advantage of the UNIX system various features such as the ability to transfer files between systems and the executing of a command in the UNIX shell were added to the Solo system. These enhancements were done to improve the usefulness of the system. The additional features added to the usefulness of Solo.

Although the porting did restore many features of the original system, there remained a few features such as a time multiplexed Kernel and the removing of Solo's virtual devices, that were not reimplemented in the latest version of Solo. The reasons for this were that the features were either undesirable or the task of porting them was beyond the scope of the project. The enhancements described in the following paragraphs are, in this writer's opinion, the most likely
candidates for future work.

The Concurrent Pascal Kernel does provide for process switching; however, it does not provide time multiplexing of processes. With the current implementation, the Kernel performs a context change only when a running process is delayed or no longer able to execute. This situation presently allows a process to execute to completion without ever relinquishing the processor. At present programs must be designed with this fact in mind.

There are two possible solutions for rewriting of the present Kernel to accommodate time multiplexing. One solution is to have another program executing in parallel with the Concurrent Pascal program. This program can be created in the same way that the unixcall routine performs the shell execution. The new program would periodically signal the concurrent program to perform a process or context switch. This solution has the drawback of the switch having to be performed in the Kernel. The concurrent program would have to determine whether it was executing in the Pascal code, the Kernel code, or the Fun-Time Library when the signal or interrupt occurred. If the program was not in the Kernel, it would then have to transfer execution to the Kernel to perform the process switch.

An alternative solution is to use the OS/32 timer SVC. This approach used to be implemented to provide the process switch when the Kernel was written in Pascal and also when it
was written in assembly language. As with the first solution, this alternative also presents the same problems to the implementor.

The concept of using the Virtual disk to simulate a disk for the Solo system could be alleviated by rewriting the disk routines to obtain their I/O from files in the UNIX system instead of from the Virtual disk. This feature was not implemented for two reasons. First, the modification would remove many of the routines from the program code and change the code which is currently used as a teaching guide in Brinch Hansen's textbook[8H1]. The second reason is that such a change would make the Solo implementation UNIX dependent and detriment the system's portability. Thus, the modification of the disk routines does not appear to be feasible.


APPENDIX A

USER GUIDES
Contained in this Appendix is the format of commands and responses produced by the Solo operating system. The intent of this is to provide a quick reference for the user unfamiliar with the software.

Execution of the Solo system is initiated with the command line of:

```
solo32 -k120k -f1x solo.vd -f3 printout -f4r cardinput
```

For convenience, this command line can be placed in a shell file. The first item of the command line is the object code file called `solo32`. This is followed by the amount of memory that is required to execute Solo. The system’s virtual disk is assigned as logical unit 1 and has read and write access. Logical unit 3 is the printdevice and has write access by default. The filename of printout is arbitrary and can have any user name assigned to it. It should be noted that this file is created each time Solo is executed and erases any contents that the file has in it prior to execution. The last logical unit (4) is the carddevice. As with the printdevice the filename of cardinput is arbitrary. The file can be any file that is desired to be input using the carddevice. If an optional name is desired the file should contain at least an Solo end-of-medium symbol (control Y) or the Card Process of Solo will terminate upon its initialization due to the file being empty.

Once the command line is entered or a shell containing it
invoked, the system identifies itself with the following prompt:

DO32:

After this is displayed, Solo is ready for commands to be entered.

The reinitialization of Solo, as mentioned before, consists of typing a control G followed by a carriage return. This causes Solo to reinitialize itself and display again the system prompt (DO32:). Solo may be terminated at any time by either striking the break key or typing the command:

LOGOFF

The logoff command allows the program to terminate normally and print the following messages:

SOLO TERMINATING

The main program has terminated

The last message produced is printed by the Kernel as a result of a concurrent program being terminated. The other method of termination (striking the break key) causes an unnecessary core dump and should only be used in cases of the failure of the Solo system to respond to the user.

The Sequential Pascal programs are invoked by typing the name and their optional arguments followed by a carriage return. The commands are entered after the system prompt of
DO32 is displayed. The following paragraphs contain a quick reference for the added programs invokations and the messages produced by them. User guides for programs not listed here can be found in the Solo Tutorials[NN].

The CallUNIX32 program is invoked with the command:

CALLUNIX32

This is followed by the routine identifying itself with the line:

CALLUNIX32:

The next message produced is to inform the user that they are entering the UNIX environment. This is immediately followed by the shell prompt.

ENTERING THE UNIX ENVIRONMENT...
$
$

A desired command may then be typed after the prompt. The user is returned to Solo once the UNIX command has been executed. The user is made aware of this by the message:

BACK AT THE SOLO ENVIRONMENT

After this message is displayed on the console the program is terminated.

The Unix32 program invocation requires, in addition to the program name, the mode of operation (INPUT or OUTPUT), and if
the operation is INPUT the filetype. If this format is not followed the following error message is produced:

TRY AGAIN

UNIX32(INPUT,FILENAME,KIND)
UNIX32(OUTPUT,FILENAME)
WHERE FILENAME:IDENTIFIER
KIND: (SCRATCH,ASCII,SEQCODE,CONCODE)

The error messages produced by the program result from improper usage of the file systems. The list which follows contains these messages and the probable cause of them.

READ ACCESS DENIED - The file to be input does not have read access for the user.

FILE UNKNOWN TO UNIX - The input filename is not located in the user's UNIX directory.

SOLO FILE LIMIT EXCEEDED - The length of the new file is longer than the maximum Solo filelength of 255 pages.

SOLO FILE NOT FOUND - The file to be output was not found in the Solo directory(CATALOG).

WRITE ACCESS DENIED - This message produced during the output operation warns the user that the file does not have write access for the user.
The other messages produced by the program are from the File32 program which is executed to save a file in the Solo catalog. These error messages can be found in the aforementioned tutorials.

The Diskspace program is invoked in the same way as the CallUnix32 program is with the single command:

```
DISKSPACE
```

The information produced by the program is the number of free pages that exist on the Virtual disk. The format of the output is:

```
VIRTUAL DISK SPACE(PAGES):
```

This line is then followed by the number of pages.
APPENDIX B

PAS32 AND CPASCAL SHELLS
This appendix contains the modified shells for the correct compilation of Sequential Pascal and Concurrent Pascal programs to be used for the Solo system. The Pas32 shell was modified so that the relocate feature could be used. As mentioned before, this is required for correct execution of Sequential Pascal programs running under the Solo system. The modification consisted of adding the shell command lines of:

```bash
if test -n "$ropt"
  then reloc -k100k -f1r /tmp/p.out$$ > $prog.rel
     rm /tmp/p.out$$
     exit
fi
```

The first line tests whether the "-r" option was specified. If the result is true, then the relocator program is executed with its input being the compiled code produced by the compiler. The relocated code of the sequential program is then placed in a file with the sequential program's original name and an extension of ".rel". Once the relocator program is done, the shell then removes a temporary work file and exits without performing the linking phase which is normally performed when compiling non-Solo programs. The code for the relocator program and the modified Pas32 shell must reside within the user's directory. This can be changed by moving the relocator's code (reloc) to another directory and appending the path to that directory in front of the it.
prog="$1"
shift
oopt=
pasopt="ns"
symasn=
copt=
ropt=
ldopt=
versn="/usr/pas32"
lib="/usr/pas32/lib"
while test $# -ne 0
do
case $1 in
  -o)
oopt="$1 $2"
shift;;
  -c)
copt="$1";;
  -r)
ropt="$1";;
  -i -n)
ldopt="$ldopt $1";;
  -t)
lib=".";;
  w sym \(SYM\)
symasn="-f3 $prog.sym"
pasopt="$pasopt $1";;
  *)
pasopt="$pasopt $1"
esac
shift
done
if test -n "$copt"
then objout="$prog.o"
else objout="/tmp/a.out$"
fi
if
$versn/prep < $prog.p > /tmp/prep$
then :
else rm /tmp/prep$; exit;
fi
if
$versn/pascal32 -k128k -f1t -f2t -f7
/tmp/p.out$ $symasn < /tmp/prep$ $pasopt
then :
else rm /tmp/p.out$; exit
fi
rm /tmp/prep$
if test -n "$ropt"
then reloc -k100k -f1r /tmp/p.out$ > $prog.rel
rm /tmp/p.out$
exit
fi
cvobj /tmp/p.out$ $objout
The Cpascal shell for Concurrent Pascal program compilation (ie. the Solo32 program) was rewritten so that the modified Kernel's and Run-Time Library's object files are linked to the program instead of the normal ones. As was the case with the relocator program before, the object files must at the present time reside in the user's directory. This can be changed, as before, by placing the code in another directory and including the path to that directory with its name.

```
prog="$1"
shift
oopt=
pasopt="ns"
symasgn=
copt=
ropt=
ldopt=
versn="/usr/pas32"
lib="/usr/pas32/lib"
driver="cpascal"
while test "$#" -ne 0 do
case $1 in
  -o)
    oopt="$1 $2"
    shift;;
  -c)
    copt="$1";;
  -r)
    ropt="$1";;
  -i -n)
    ldopt="$1$opt $1";;
  -t)
    lib=".";;
  -v)
```
set -x;:

w sym W SYM
    symasgn="-f3 $proq.sym"
pasopt="$pasopt "$1;:
*)
pasopt="$pasopt "$1
esac
shift
done
if test -n "$copt"
then objout="$proq.o"
else objout="/tmp/a.out$$"
fi
if
    nice -5 $versn/prep < $proq.p > /tmp/prep$$
then :
else rm /tmp/prep$$; exit;
fi
if
    nice -5 $versn/$driver -k128k -f1t -f2t -f7
        /tmp/p.out$$ $symasgn < /tmp/prep$$ $pasopt
then :
else rm /tmp/p.out$$; exit
fi
    rm /tmp/prep$$
if test -n "$ropt"
then echo reloc not supported
    rm /tmp/p.out$$
exit
fi
    nice -5 cvobj /tmp/p.out$$ $objout
    rm /tmp/p.out$$
if test -n "$copt"
then exit
fi
    nice -5 ld -X $o0pt $ldopt conlibrary.o $objout
    kernel.o $lib/condummy.o -lc
    rm /tmp/a.out$$
APPENDIX C

THE SOLO32 PROGRAM
"PER BRINCH HANSEN

INFORMATION SCIENCE
CALIFORNIA INSTITUTE OF TECHNOLOGY

THE SOLO SYSTEM
8 JUNE 1975"

"DAVID NEAL

DEPARTMENT OF COMPUTER SCIENCE
KANSAS STATE UNIVERSITY

MODIFICATIONS OF THE SOLO SYSTEM FOR USE
UNDER INTERDATA'S OS/32-MT AND FOR THE USE
OF TWO VIRTUAL DISKS

OCTOBER 15, 1977"

"MARTIN WILDE

DEPARTMENT OF COMPUTER SCIENCE
KANSAS STATE UNIVERSITY

MODIFICATIONS OF THE SOLO SYSTEM FOR USE
WITH UNIX. INCLUDES ENTRY POINTS FOR
UNIX FILE MANIPULATION

1984"

KERNEL

TYPE FULLWORD = INTEGER;
CONST PAGELength = 512;
TYPE PAGE = ARRAY [1..PAGELength] OF CHAR;
TYPE INTEGER = SHORTINTEGER;
CONST IDLength = 12;
TYPE IDENTIFIER = ARRAY[1..IDLength] OF CHAR;
CONST UNIXLENGTH = 13;
TYPE UNIX_NAME = ARRAY[1..UNIXLENGTH] OF CHAR;
TYPE ATTRINDEX = (ATTRCALL, ATTRHEAP, ATTRLINE, ATTRSIT,

ATTRRTM);
FUNCTION ATTRIBUTE(A:ATTRINDEX):FULLWORD;
PROCEDURE SETHEAP(A:FULLWORD);
PROCEDURE START(A:INTEGER);
PROCEDURE STOP(A,B:INTEGER);
PROCEDURE WAIT;
FUNCTION REALTIME:INTEGER;
PROCEDURE SVC1;
PROCEDURE SVC2;
PROCEDURE SVC7;
PROCEDURE GETMEM;
PROCEDURE BREAKPNT (LN: FULLWORD);
PROCEDURE SVC15;
FUNCTION UNIXREADS (FILEDES: INTEGER; VAR BUF: PAGE;
LENGTH: INTEGER): INTEGER;
FUNCTION UNIXWRTIES (FILEDES: INTEGER; BUF: PAGE;
LENGTH: INTEGER): INTEGER;
FUNCTION UNIXCLOSES (FILEDES: INTEGER): INTEGER;
FUNCTION UNIXCREATES (UNIXNAME: UNIX_NAME;
MODE: INTEGER): INTEGER;
FUNCTION UNIXACCESSSES (UNIXNAME: UNIX_NAME;
MODE: INTEGER): INTEGER;

PROCEDURE CALLUNIX;
PROCEDURE QUIT;
END;

"###################################################
# IO TYPES #
###################################################"

TYPE IODEVICE =
  (TYPEDEVICE, PRIVATEDISK, TAPEDEVICE, PRINTDEVICE,
   CARDDVICE, TAPE2DEVICE, ITAMDEVICE, VARDEVICE,
   SYSTEMDISK);  "KSu"

TYPE IOOPERATION =
  (INPUT, OUTPUT, MOVE, CONTROL);

TYPE IOARG =
  (WRITEEOP, REWIND, UPSPACE, BACKSPACE);

TYPE IORESULT =
  (COMPLETE, INTERVENTION, TRANSMISSION, FAILURE,
   ENDFILE, ENDMEDIUM, STARTMEDIUM);

TYPE IOPARAM = PACKED RECORD
  OPERATION: IOOPERATION;
  STATUS: IORESULT;
  ARG: 0..32768
END;

CONST
NL = '(:10:)'; FF = '(:12:)'; CR = '(:13:)'; EM = '(:25:)';

CONST LINELENGTH = 132;
TYPE LINE = ARRAY [1..LINELENGTH] OF CHAR;
CONST PAGELENGTH = 512;
CONST PROCESSCOUNT = 7;
TYPE PROCESSQUEUE = ARRAY[1..PROCESSCOUNT] OF QUEUE;

TYPE FIFO =
CLASS(LIMIT: INTEGER);

VAR HEAD, TAIL, LENGTH: INTEGER;

"PROCEDURE DUMMY; BEGIN END;"

FUNCTION ENTRY ARRIVAL: INTEGER;
BEGIN
  ARRIVAL := TAIL;
  TAIL := TAIL MOD LIMIT + 1;
  LENGTH := LENGTH + 1;
END;

FUNCTION ENTRY DEPARTURE: INTEGER;
BEGIN
  DEPARTURE := HEAD;
  HEAD := HEAD MOD LIMIT + 1;
  LENGTH := LENGTH - 1;
END;

FUNCTION ENTRY EMPTY: BOOLEAN;
BEGIN EMPTY := (LENGTH = 0) END;

FUNCTION ENTRY FULL: BOOLEAN;
BEGIN FULL := (LENGTH = LIMIT) END;

BEGIN HEAD := 1; TAIL := 1; LENGTH := 0 END;
TYPE RESOURCE =
  MONITOR

VAR FREE: BOOLEAN; Q: PROCESSQUEUE; NEXT: FIFO;

PROCEDURE ENTRY REQUEST;
BEGIN
  IF FREE THEN FREE:= FALSE
  ELSE DELAY(QF NEXT.ARRIVAL));
END;

PROCEDURE ENTRY RELEASE;
BEGIN
  IF NEXT.EMPTY THEN FREE:= TRUE
  ELSE CONTINUE(QF NEXT.DEPARTURE));
END;

BEGIN FREE:= TRUE; INIT NEXT(PROCCOUNT) END;

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""`
TYPE TYPEWRITER =
CLASS(DEVICE: IODEVICE);

CONST LINELIMIT = 81;

PROCEDURE WRITECHAR(X: CHAR);
VAR PARAM: IOPARAM; C: CHAR;
BEGIN
  PARAM.OPERATION:= OUTPUT;
  C:= X;
  IO(C, PARAM, DEVICE);
END;

PROCEDURE ENTRY WRITE(TEXT: LINE);
VAR PARAM: IOPARAM;
  I: INTEGER; C: CHAR;
BEGIN
  PARAM.OPERATION:= OUTPUT;
  I:= 0;
  REPEAT
    I:= I + 1; C:= TEXT[I];
    IO(C, PARAM, DEVICE);
    UNTIL (C = NL) OR (I = LINELIMIT);
    IF C <> NL THEN WRITECHAR(NL);
  END;
END;

PROCEDURE ENTRY READ(VAR TEXT: LINE);
CONST BEL = ' (·7·)';
VAR PARAM: IOPARAM;
  I: INTEGER; C: CHAR;
BEGIN
  WRITECHAR(BEL);
  PARAM.OPERATION:= INPUT;
  I:= 0;
  REPEAT
    IO(C, PARAM, DEVICE);
    "CODE TO CANCEL INPUT LINES AND CHARACTERS DELETED - KSU"
    I:= I + 1; TEXT[I]:= C;
    UNTIL (C = NL) OR (I = LINELIMIT);
    IF C <> NL THEN
      BEGIN
        WRITECHAR(NL);
        TEXT[LINELIMIT + 1]:= NL;
      END;
  END;
END;
BEGIN END;
TYPE TERMINAL =
CLASS(ACCESS: TYPERESOURCE);

VAR UNIT: TYPEWRITER;

PROCEDURE ENTRY READ(HEADER: LINE; VAR TEXT: LINE);
VAR CHANGED: BOOLEAN;
BEGIN
  ACCESS.REQUEST(HEADER, CHANGED);
  IF CHANGED THEN UNIT.WRITE(HEADER);
  UNIT.READ(TEXT);
  ACCESS.RELEASE;
END;

PROCEDURE ENTRY WRITE(HEADER, TEXT: LINE);
VAR CHANGED: BOOLEAN;
BEGIN
  ACCESS.REQUEST(HEADER, CHANGED);
  IF CHANGED THEN UNIT.WRITE(HEADER);
  UNIT.WRITE(TEXT);
  ACCESS.RELEASE;
END;

BEGIN INIT UNIT(TYPEDEVICE) END;
TYPE TERMINALSTREAM =
CLASS(OPTERATOR: TERMINAL);

CONST LINELIMIT = 80;

VAR HEADER: LINE; ENDINPUT: BOOLEAN;
    INP, OUT: RECORD COUNT: INTEGER; TEXT: LINE END;

PROCEDURE INITIALIZE(TEXT: LINE);
BEGIN
    HEADER := TEXT;
    ENDINPUT := TRUE;
    OUT.COUNT := 0;
END;

PROCEDURE ENTRY READ(VAR C: CHAR);
BEGIN
    WITH INP DO
    BEGIN
        IF ENDINPUT THEN
        BEGIN
            OPTERATOR.READ(HEADER, TEXT);
            COUNT := 0;
        END;
        COUNT := COUNT + 1;
        C := TEXT[COUNT];
        ENDINPUT := (C = NL);
    END;
END;

PROCEDURE ENTRY WRITE(C: CHAR);
BEGIN
    WITH OUT DO
    BEGIN
        COUNT := COUNT + 1;
        TEXT[COUNT] := C;
        IF (C = NL) OR (COUNT = LINELIMIT) THEN
        BEGIN
            OPTERATOR.WRITE(HEADER, TEXT);
            COUNT := 0;
        END;
    END;
END;

PROCEDURE ENTRY RESET(TEXT: LINE);
BEGIN INITIALIZE(TEXT) END;

BEGIN INITIALIZE("UNIDENTIFIED:(::10:)") END;
TYPE DISK =
CLASS(TYPEUSE: TYPERESOURCE);
"DISKDEVICE ADDED TO ALL PROCEDURES TO ALLOW THE USE OF MULTIPLE VIRTUAL DISKS"

VAR OPERATOR: TERMINAL;

FUNCTION XLTESTAT (STATUS: UNIV INTEGER): INTEGER;
BEGIN
  XLTESTAT := STATUS;
END;

PROCEDURE TRANSFER(COMMAND: IOOPERATION;
  PAGEADDR: UNIV IOARG; VAR BLOCK: PAGE;
  DISKDEVICE: IODEVICE);

  "STATUS CHANGES MADE FOR OS/32-MT FOIBLES"

VAR PARAM: IOPARAM; RESPONSE: LINE;
BEGIN
  WITH PARAM, OPERATOR DO
  BEGIN
    OPERATION := COMMAND;
    ARG := ORD(PAGEADDR);
    IO(BLOCK, PARAM, DISKDEVICE);
    IF STATUS = ENDFILE THEN "ERROR"
      STATUS := COMPLETE;
    WHILE STATUS <> COMPLETE DO
    BEGIN
      RESPONSE[1] := CHR(XLTESTAT (STATUS) + ORD('0'));
      WRITE('DISK: ERROR (:10:) ', RESPONSE);
      READ('PUSH RETURN (:10:) ', RESPONSE);
      IO(BLOCK, PARAM, DISKDEVICE);
    END;
  END;
END;

PROCEDURE ENTRY READ(PAGEADDR: INTEGER;
  VAR BLOCK: UNIV PAGE; DISKDEVICE: IODEVICE);
BEGIN TRANSFER(INPUT, PAGEADDR, BLOCK, DISKDEVICE) END;

PROCEDURE ENTRY WRITE(PAGEADDR: INTEGER;
  VAR BLOCK: UNIV PAGE;
  DISKDEVICE: IODEVICE);
BEGIN TRANSFER(OUTPUT, PAGEADDR, BLOCK, DISKDEVICE) END;

BEGIN INIT OPERATOR(TYPEUSE) END;
CONST MAPLENTH = 255;
TYPE FILEMAP = PACKED RECORD
  FILELENGTH: SHORTINTEGER;
  PAGESET: ARRAY "1..MAPLENTH" OF SHORTINTEGER
END;

TYPE DISKFILE =
CLASS(TYPEUSE: TYPERESOURCE);

"IODEVICE ADDED TO PROCEDURES TO ALLOW THE USE OF MULTIPLE VIRTUAL DISKS" - KSU

VAR UNIT: DISK; MAP: FILEMAP; OPENED: BOOLEAN;

ENTRY LENGTH: INTEGER;

FUNCTION INCLUDES(PAGENO: INTEGER): BOOLEAN;
BEGIN
  INCLUDES := OPENED &
  (1 <= PAGENO) & (PAGENO <= LENGTH);
END;

PROCEDURE ENTRY OPEN(MAPADDR: INTEGER; DISK: IODEVICE);
BEGIN
  UNIT.READ(MAPADDR, MAP, DISK);
  LENGTH := MAP.FILELENGTH;
  OPENED := TRUE;
END;

PROCEDURE ENTRY CLOSE;
BEGIN
  LENGTH := 0;
  OPENED := FALSE;
END;

PROCEDURE ENTRY READ(PAGENO: INTEGER; VAR BLOCK: UNIV PAGE;
  DISK: IODEVICE);
BEGIN
  IF INCLUDES(PAGENO) THEN
    UNIT.READ(MAP.PAGESET[PAGENO], BLOCK, DISK);
  END;

PROCEDURE ENTRY WRITE(PAGENO: INTEGER; VAR BLOCK: UNIV PAGE;
  DISK: IODEVICE);
BEGIN
  IF INCLUDES(PAGENO) THEN
    UNIT.WRITE(MAP.PAGESET[PAGENO], BLOCK, DISK);
  END;
BEGIN
  INIT UNIT (TYPEUSE);
  LENGTH := 0;
  OPENED := FALSE;
END;

"================================
# CATALOG STRUCTURE #
================================" 

TYPE FILEKIND =
  (EMPTY, SCRATCH, ASCII, SEQCODE, CONCODE);

TYPE FILEATTR = PACKED RECORD
  KIND: FILEKIND;
  ADDR: SHORTINTEGER;
  PROTECTED: BOOLEAN;
  NOTUSED: ARRAY [1..5] OF SHORTINTEGER
END;

TYPE CATENTRY = PACKED RECORD
  ID: IDENTIFIER;
  ATTR: FILEATTR;
  KEY, SEARCHLENGTH: SHOPTINTEGER
END;

CONST CATPAGELENGTH = 16;
TYPE CATPAGE = ARRAY [1..CATPAGELENGTH] OF CATENTRY;

CONST CATADDR = 154;
TYPE DISKTABLE =
CLASS(TYPEUSE: TYPERESOURCE;
    CATADDR: INTEGER);

"DISKDEVICE PERMANENT VARIABLE AND DISK PARAMETER ADDED TO
ALLOW FOR USE OF MULTIPLE VIRTUAL DISKS          - KSU"

VAR FILE: DISKFILE;
    PAGENO: INTEGER; DISKDEVICE: IODEVICE; BLOCK: CATPAGE;

ENTRY FLENGTH: INTEGER;

PROCEDURE ENTRY READ(I: INTEGER; VAR ELEM: CATENTRY;
                          DISK: IODEVICE);

VAR INDEX: INTEGER;
BEGIN
    IF DISK <> DISKDEVICE THEN
        WITH FILE DO
            BEGIN
                CLOSE;
                FILE.OPEN(CATADDR, DISK);
                FLENGTH := FILE.LENGTH * CATPAGELength;
                PAGENO := 0;
                DISKDEVICE := DISK;
            END;
    INDEX := (I - 1) DIV CATPAGELength + 1;
    IF PAGENO <> INDEX THEN
        BEGIN
            PAGENO := INDEX;
            FILE.READ(PAGENO, BLOCK, DISK);
        END;
    ELEM := BLOCK[(I - 1) MOD CATPAGELength + 1];
END;

BEGIN
    INIT FILE(TYPEUSE);
    DISKDEVICE := PRIVATEDISK;
    FILE.OPEN(CATADDR, DISKDEVICE);
    FLENGTH := FILE.LENGTH * CATPAGELength;
    PAGENO := 0;
END;
TYPE DISKCATALOG =
MONITOR (TYPEUSE: TYPERESOURCE;
       DISKUSE: RESOURCE; CATADDR: INTEGER);
"THE DISK PARAMETER HAS BEEN ADDED TO ALLOW FOR THE USE OF MULTIPLE VIRTUAL DISKS" - KSU
VAR TABLE: DISKTABLE;

FUNCTION HASH(ID: IDENTIFIER): INTEGER;
"THE VALIDITY OF THE HASH FUNCTION DEPENDS ON THE FACT THAT ALL VIRTUAL DISKS HAVE THE SAME CATALOG SIZE" - KSU
VAR KEY, I: INTEGER; C: CHAR;
BEGIN
  KEY := 1; I := 0;
  REPEAT
    I := I + 1; C := ID[I];
    IF C <> ' ' THEN
      KEY := KEY * ORD(C) MOD TABLE.FLENGTH + 1;
    UNTIL (C = ' ') OR (I = IDLENGTH);
  HASH := KEY;
END;

PROCEDURE ENTRY_LOOKUP
(ID: IDENTIFIER; VAR ATTR: FILEATTR; VAR FOUND: BOOLEAN;
DISK: IODEVICE);
VAR KEY, MORE, INDEX: INTEGER; ELEM: CATENTRY;
BEGIN
  DISKUSE.REQUEST;
  KEY := HASH(ID);
  TABLE.READ(KEY, ELEM, DISK);
  MORE := ELEM.SEARCHLENGTH;
  INDEX := KEY; FOUND := FALSE;
  WHILE NOT FOUND & (MORE > 0) DO BEGIN
    TABLE.READ(INDEX, ELEM, DISK);
    IF ELEM.ID = ID THEN
      BEGIN ATTR := ELEM.ATTR; FOUND := TRUE END
    ELSE
      BEGIN
        IF ELEM.KEY = KEY THEN MORE := MORE - 1;
        INDEX := INDEX MOD TABLE.FLENGTH + 1;
      END;
  END;
  DISKUSE.RELEASE;
END;

BEGIN INIT TABLE(TYPEUSE, CATADDR) END;
TYPE DATAPFILE =
CLASS(TYPEUSE: TYPERESOURCE; DISKUSE: RESOURCE;
       CATALOG: DISKCATALOG);

VAR FILE: DISKFILE; OPENED: BOOLEAN;

ENTRY LENGTH: INTEGER;

PROCEDURE ENTRY OPEN(ID: IDENTIFIER; VAR FOUND: BOOLEAN);
VAR ATTR: FILEATTR;
BEGIN
   CATALOG.LOOKUP(ID, ATTR, FOUND, PRIVATE_DISK);
   IF FOUND THEN
      BEGIN
         DISKUSE.REQUEST;
         FILE.OPEN(ATTR.ADDR, PRIVATE_DISK);
         LENGTH := FILE.LENGTH;
         DISKUSE.RELEASE;
      END;
      OPENED := FOUND;
   END;
END;

PROCEDURE ENTRY CLOSE;
BEGIN
   FILE.CLOSE;
   LENGTH := 0;
   OPENED := FALSE;
END;

PROCEDURE ENTRY READ(PAGENO: INTEGER; VAR BLOCK: UNIV PAGE);
BEGIN
   IF OPENED THEN
      BEGIN
         DISKUSE.REQUEST;
         FILE.READ(PAGENO, BLOCK, PRIVATE_DISK);
         DISKUSE.RELEASE;
      END;
   END;

PROCEDURE ENTRY WRITE(PAGENO: INTEGER; VAR BLOCK: UNIV PAGE);
BEGIN
   IF OPENED THEN
      BEGIN
         DISKUSE.REQUEST;
         FILE.WRITE(PAGENO, BLOCK, PRIVATE_DISK);
         DISKUSE.RELEASE;
      END;
   END;
BEGIN
    INIT FILE (TYPEUSE);
    LENGTH := 0;
    OPENED := FALSE;
END:
TYPE PROGSTATE = (READY, NOTFOUND, NOTSEQ, TOOBIG);

CONST STORELENGTH1 = 40;
TYPE PROGSTORE1 = 
ARRAY [1..STORELENGTH1] OF PAGE;

TYPE PROGFILE1 = 
CLASS(TYPEUSE: TYPERESOURCE; DISKUSE: RESOURCE;
    CATALOG: DISKCATALOG);

"PROGRAM FILES ARE FOUND BY FIRST SEARCHING THE PRIVATE DISK
IF NOT FOUND THERE, THE SYSTEM DISK IS THEN SEARCHED -KSU"

VAR FILE: DISKFILE;
ENTRY STORE: PROGSTORE1;

PROCEDURE ENTRY OPEN(ID: IDENTIFIER; VAR STATE: PROGSTATE);
VAR ATTR: FILEATTR; FOUND: BOOLEAN; PAGENO: INTEGER;
    DISK: IODEVICE;
BEGIN
    CATALOG.LOOKUP(ID, ATTR, FOUND, PRIVATEDISK);
    IF FOUND THEN DISK := PRIVATEDISK
    ELSE BEGIN
        CATALOG.LOOKUP(ID, ATTR, FOUND, SYSTEMDISK);
        DISK := SYSTEMDISK;
    END;
    WITH DISKUSE, FILE, ATTR DO
    IF NOT FOUND THEN
        STATE := NOTFOUND ELSE
    IF KIND <> SEQCODE THEN
        STATE := NOTSEQ ELSE
    BEGIN
        REQUEST;
        OPEN(ADDR, DISK);
        IF LENGTH <= STORELENGTH1 THEN
            BEGIN
                FOR PAGENO := 1 TO LENGTH DO
                    READ(PAGENO, STORE[PAGENO], DISK);
                STATF := READY;
            END ELSE
                STATE := TOOBIG;
            CLOSE;
            RELEASE;
    END;
END;
BEGIN
  INIT FILE (TYPEUSE);
END;

CONST STORELENGTH2 = 8;
TYPE PROGSTORE2 =
  ARRAY [1..STORELENGTH2] OF PAGE;

TYPE PROGPFILE2 =
CLASS(TYPEUSE: TYPERESOURCE; DISKUSE: RESOURCE;
  CATALOG: DISKCATALOG);

VAR FILE: DISKFILE;

ENTRY STORE: PROGSTORE2;

PROCEDURE ENTRY OPEN (ID: IDENTIFIER; VAR STATE: PROGSTATE);
VAR ATTR: FILEATTR; FOUND: BOOLEAN; PAGENO: INTEGER;
  DISK: IODEVICE;
BEGIN
  CATALOG. LOOKUP (ID, ATTR, FOUND, PRIVATEDISK);
  IF FOUND THEN DISK := PRIVATEDISK
  ELSE BEGIN
    CATALOG. LOOKUP (ID, ATTR, FOUND, SYSTEMDISK);
    DISK := SYSTEMDISK;
    END;
  WITH DISKUSE, FILE, ATTR DO
  IF NOT FOUND THEN
    STATE := NOTFOUND ELSE
  IF KIND <> SEQCODE THEN
    STATE := NOTSEQ ELSE
  BEGIN
    REQUEST;
    OPEN (ADDR, DISK);
    IF LENGTH <= STORELENGTH2 THEN
    BEGIN
      FOR PAGENO := 1 TO LENGTH DO
        READ (PAGENO, STORE[PAGENO], DISK);
      STATE := READY;
    END ELSE
    STATE := TOOBIG;
    CLOSE;
    RELEASE;
  END;
END;

BEGIN
  INIT FILE (TYPEUSE);
END;
TYPE RESULTTYPE =
    (TERMINATED, OVERFLOW, POINTERERROR, RANGEERROR,
     VARIANTERROR, HEAPLIMIT, STACKLIMIT, CODELIMIT, TIMELIMIT,
     CALERROR);

"TYPE ATTRINDEX =
    (CALLER, HEAPTOP, PROGLINE, PROGREULT, RUNTIME):"

TYPE PROGSTACK =
    MONITOR

CONST STACKLENGTH = 5;

VAR STACK: ARRAY [1..STACKLENGTH] OF
    RECORD
        PROGID: IDENTIFIER;
        HEAPADDR: FULLWORD
    END;
    TOP: 0..STACKLENGTH;

FUNCTION ENTRY SPACE: BOOLEAN;
BEGIN SPACE: = (TOP < STACKLENGTH) END;

FUNCTION ENTRY ANY: BOOLEAN;
BEGIN ANY: = (TOP > 0) END;

PROCEDURE ENTRY PUSH (ID: IDENTIFIER);
BEGIN
    IF TOP < STACKLENGTH THEN
        BEGIN
            TOP := TOP + 1;
            WITH STACK[TOP] DO
                BEGIN
                    PROGID := ID;
                    HEAPADDR := ATTRIBUTE (ATTRHEAP);
                END;
        END;
    END;

PROCEDURE ENTRY POP
    (VAR LINE, RESULT: UNIV INTEGER);
CONST TERMINATED = 0;
BEGIN
    LINE := ATTRIBUTE (ATTRLINE);
    RESULT := ATTRIBUTE (ATTRRESULT);
    CASE RESULT OF
        0: RESULT := TERMINATED;
        16, 17, 18: RESULT := ORD (OVERFLOW);
7: RESULT := ORD (POINTERERROR);
1, 2, 3, 4, 5: RESULT := ORD (RANGEERROR);
6: RESULT := ORD (VARIANTERROR);
9: RESULT := ORD (HEAPLIMIT);
8: RESULT := ORD (STACKLIMIT);
END;
    IF RESULT <> TERMINATED THEN
      SETHEAP (STACK[TOP].HEAPADDR);
      TOP := TOP - 1;
    END;

PROCEDEPE ENTRY GET (VAR ID: IDENTIFIER);
BEGIN
    IF TOP > 0 THEN ID := STACK[TOP].PROGID;
    END;
BEGIN TOP := 0 END;

"########################################################
# TASKKIND AND ARGTYPE #
########################################################"

TYPE TASKKIND = (INPUTTASK, JOBTASK, OUTPUTTASK);

TYPE ARGTAG =
               (NILTYPE, BOOLTYPE, INTTYPE, IDTYPE, PTRTYPE);

ARGTYPE = RECORD
            TAG: ARGTAG;
            ARG: IDENTIFIER
       END;

CONST MAXARG = 10;
TYPE ARGLIST = ARRAY [1..MAXARG] OF ARGTYPES;

TYPE ARGSEQ = (INP, OUT);
TYPE ARGBUFFER =
MONITOR
VAR BUFFER: ARGTYPE; FULL: BOOLEAN;
SEND, RECEIVER: QUEUE;

PROCEDURE ENTRY READ(VAR ARG: ARGTYPE);
BEGIN
  IF NOT FULL THEN DELAY(RECEIVER);
  ARG := BUFFER; FULL := FALSE;
  CONTINUE(SENDER);
END;

PROCEDURE ENTRY WRITE(ARG: ARGTYPE);
BEGIN
  IF FULL THEN DELAY(SENDER);
  BUFFER := ARG; FULL := TRUE;
  CONTINUE(RECEIVER);
END;
BEGIN FULL := FALSE END;

TYPE LINEBUFFER =
MONITOR
VAR BUFFER: LINE; FULL: BOOLEAN; SENDER, RECEIVER: QUEUE;

PROCEDURE ENTRY READ(VAR TEXT: LINE);
BEGIN
  IF NOT FULL THEN DELAY(RECEIVER);
  TEXT := BUFFER; FULL := FALSE;
  CONTINUE(SENDER);
END;

PROCEDURE ENTRY WRITE(TEXT: LINE);
BEGIN
  IF FULL THEN DELAY(SENDER);
  BUFFER := TEXT; FULL := TRUE;
  CONTINUE(RECEIVER);
END;
BEGIN FULL := FALSE END;
TYPE PAGEBUFFER =
MONITOR
VAR BUFFER: PAGE; LAST, FULL: BOOLEAN;
    SENDER, RECEIVER: QUEUE;

PROCEDURE ENTRY READ(VAR TEXT: PAGE; VAR EOF: BOOLEAN);
BEGIN
  IF NOT FULL THEN DELAY(RECEIVER);
  TEXT := BUFFER; EOF := LAST; FULL := FALSE;
  CONTINUE(SENDER);
END;

PROCEDURE ENTRY WRITE(TEXT: PAGE; EOF: BOOLEAN);
BEGIN
  IF FULL THEN DELAY(SENDER);
  BUFFER := TEXT; LAST := EOF; FULL := TRUE;
  CONTINUE(RECEIVER);
END;

BEGIN FULL := FALSE END;
TYPE CHARSTREAM =
CLASS(BUFFER: PAGEBUFFER);

VAR TEXT: PAGE; COUNT: INTEGER; EOF: BOOLEAN;

PROCEDURE ENTRY READ(VAR C: CHAR);
BEGIN
  IF COUNT = PAGELENGTH THEN
    BEGIN
      BUFFER.READ(TEXT, EOF);
      COUNT := 0;
    END;
    COUNT := COUNT + 1;
    C := TEXT[ COUNT ];
  IF C = EM THEN
    BEGIN
      WHILE NOT EOF DO BUFFER.READ(TEXT, EOF);
      COUNT := PAGELENGTH;
    END;
  END;
END;

PROCEDURE ENTRY INITREAD;
BEGIN COUNT := PAGELENGTH END;

PROCEDURE ENTRY WRITE(C: CHAR);
BEGIN
  COUNT := COUNT + 1;
  TEXT[ COUNT ] := C;
  IF (COUNT = PAGELENGTH) OR (C = EM) THEN
    BEGIN
      BUFFER.WRITE(TEXT, FALSE); COUNT := 0;
      IF C = EM THEN BUFFER.WRITE(TEXT, TRUE);
    END;
  END;

PROCEDURE ENTRY INITWRITE;
BEGIN COUNT := 0 END;

BEGIN END;
TYPE JOBPROCESS =
PROCESS
(TYPEUSE: TIPERESOURCE; DISKUSE: RESOURCE;
CATALOG: DISKCATALOG; INBUFFER, OUTBUFFER: PAGEBUFFER;
INREQUEST, INRESPONSE, OUTREQUEST, OUTRESPONSE: ARGBUFFER;
STACK: PROGSTACK);

"PROGRAM DATA SPACE = " "+24000"
"KSU - ALLOWS MAX DATA ADDRESSABILITY"

CONST MAXFILE = 2;
TYPE FILE = 1..MAXFILE;

CONST DATASIZE = 24000;
TYPE DATASPACE = ARRAY [1..DATASIZE] OF BYTE;
VAR
OPERATOR: TERMINAL; OậmSTREAM: TERMINALSTREAM;

INSTREAD, OUTREAD: CHARSTREAM;
FILES: ARRAY [FILE] OF DATAPkE;
CODE: PROGFILE;
UNIX:UNIXIO;

DATA: DATASPACE;

PROGRAM JOB (VAR PARAM: ARGLIST; STORE: PROGSTORE;
    DATALEN:INTEGER; VAR DATA:DATASPACE);
ENTRY READ, WRITE, OPEN, CLOSE, GET, PUT, LENGTH,
    MARK, RELEASE, IDENTIFY, ACCEPT, DISPLAY, READPAGE,
    WRITEPAGE, READLINE, WRITELINE, READARG, WRITEARG,
    LOOKUP, IOTRANSFER, IOMOVE, TASK, RUN,
    UNIXOPEN, UNIXREAD, UNIXWRITE, UNIXCLOSE, UNIXCREATE,
    UNIXACCESS, UNIXCALL, EXIT;

PROCEDURE CALL (ID: IDENTIFIER; VAR PARAM: ARGLIST;
    VAR LINE: INTEGER; VAR RESULT: RESULTTYPE);
VAR STATE: PROGSTATE; LASTID: IDENTIFIER;
BEGIN
WITH CODE, STACK DO
BEGIN
    LINE:= 0;
    OPEN (ID, STATE);
    IF (STATE = READY) & SPACE THEN
    BEGIN
        PUSH (ID);
JOB(PARM, STORE, DATASIZE, DATA);
POP(LINE, RESULT);
END ELSE
IF STATE = TOO BIG THEN RESULT := CODELIMIT
ELSE RESULT := CALLERROR;
IF ANY THEN
BEGIN GET(LASTID); OPEN(LASTID, STATE) END;
END;
END;

PROCEDURE ENTRY READ(VAR C: CHAR);
BEGIN INSTREAM.READ(C) END;

PROCEDURE ENTRY WRITE(C: CHAR);
BEGIN OUTSTREAM.WRITE(C) END;

PROCEDURE ENTRY OPEN
(F: FILE; ID: IDENTIFIER; VAR FOUND: BOOLEAN);
BEGIN FILES[F].OPEN(ID, FOUND) END;

PROCEDURE ENTRY CLOSE(F: FILE);
BEGIN FILES[F].CLOSE END;

PROCEDURE ENTRY GET(F: FILE; P: INTEGER; VAR BLOCK: PAGE);
VAR NEWTIME: INTEGER;
BEGIN
FILES[F].READ(P, BLOCK);
END;

PROCEDURE ENTRY PUT(F: FILE; P: INTEGER; VAR BLOCK: PAGE);
BEGIN FILES[F].WRITE(P, BLOCK) END;

FUNCTION ENTRY LENGTH(F: FILE): INTEGER;
BEGIN LENGTH := FILES[F].LENGTH END;

PROCEDURE ENTRY MARK(VAR TOP: FULLWORD);
BEGIN TOP := ATTRIBUTE(ATTPHEAP) END;

PROCEDURE ENTRY RELEASE(TOP: FULLWORD);
BEGIN SETHEAP(TOP) END;

PROCEDURE ENTRY IDENTIFY(HEADER: LINE);
BEGIN OPSTREAM.RESET(HEADER) END;

PROCEDURE ENTRY ACCEPT(VAR C: CHAR);
BEGIN OPSTREAM.READ(C) END;

PROCEDURE ENTRY DISPLAY(C: CHAR);
BEGIN OPSTREAM.WRITE(C) END;

PROCEDURE ENTRY READPAGE(VAR BLOCK: PAGE; VAR EOF: BOOLEAN);
BEGIN INBUFFER.READ(BLOCK, EOF) END;
PROCEDURE ENTRY WRITEPAGE(BLOCK: PAGE; EOF: BOOLEAN);
BEGIN OUTBUFFER.WRITE(BLOCK, EOF) END;

PROCEDURE ENTRY READLINE(VAR TEXT: LINE);
BEGIN END;

PROCEDURE ENTRY WRITELINE(TEXT: LINE);
BEGIN END;

PROCEDURE ENTRY READARG(S: ARGSEQ; VAR ARG: ARGTYPE);
BEGIN
  IF S = INP THEN INRESPONSE.READ(ARG)
  ELSE OUTRESPONSE.READ(ARG); END;

PROCEDURE ENTRY WRITEARG(S: ARGSEQ; ARG: ARGTYPE);
BEGIN
  IF S = INP THEN INREQUEST.WRITE(ARG)
  ELSE OUTREQUEST.WRITE(ARG); END;

PROCEDURE ENTRY LOOKUP
(ID: IDENTIFIER; VAR ATTR: FILEATTR; VAR FOUND: BOOLEAN);
"CATALOG IS SEARCHED FOR PRIVATE DISK FOLLOWED BY SYSTEM DISK
  ONLY SEQCODE FILES ARE FOUND ON THE SYSTEM DISK - FSU"
BEGIN
  CATALOG.LOOKUP(ID, ATTR, FOUND, PRIVATEDISK);
  IF NOT FOUND THEN BEGIN
    CATALOG.LOOKUP(ID, ATTR, FOUND, SYSTEMDISK);
    FOUND := FOUND AND (ATTR.KIND = SEQCODE);
  END;
END;

PROCEDURE ENTRY IOTRANSFER
(DEVICE: IODEVICE; VAR PARAM: IOPARAM; VAR BLOCK: PAGE);
BEGIN
  IF (DEVICE = PRIVATEDISK) OR (DEVICE = SYSTEMDISK) THEN BEGIN
    DISKUSE.REQUEST;
    IO (BLOCK, PARAM, DEVICE);
    WITH PARAM DO
      IF STATUS = ENDFILE THEN
        STATUS := COMPLETE;
    DISKUSE.RELEASE;
    END ELSE
    IO (BLOCK, PARAM, DEVICE);
END;

PROCEDURE ENTRY IOMOVE(DEVICE: IODEVICE; VAR PARAM: IOPARAM);
BEGIN IO (PARAM, PARAM, DEVICE) END;

FUNCTION ENTRY TASK: TASKKIND;
BEGIN TASK:= JOB TASK END;

PROCEDURE ENTRY RUN
(ID: IDENTIFIER; VAR PARAM: ARGLIST;
 VAR LINE: INTEGER; VAR RESULT: RESULTTYPE);
BEGIN CALL (ID, PARAM, LINE, RESULT) END;

FUNCTION ENTRY UNIXOPEN(VAR UNIXNAME:UNIX_NAME;
 MODE:INTEGER):INTEGER;
BEGIN UNIXOPEN:=UNIXOPENS(UNIXNAME,MODE); END;

FUNCTION ENTRY UNIXREAD(FILEDES:INTEGER;VAR BUF:PAGE;
 LENGTH:INTEGER):INTEGER;
BEGIN UNIXREAD:=UNIXREADS(FILEDES,BUF.LENGTH); END;

FUNCTION ENTRY UNIXWRITE(FILEDES:INTEGER;BUF:PAGE;
 LENGTH:INTEGER):INTEGER;
BEGIN UNIXWRITE:=UNIXWRITES(FILEDES,BUF.LENGTH); END;

FUNCTION ENTRY UNIXCLOSE(FILEDES:INTEGER):INTEGER;
BEGIN UNIXCLOSE:=UNIXCLOSES(FILEDES); END;

FUNCTION ENTRY UNIXCREATE(VAR UNIXNAME:UNIX_NAME;
 MODE:INTEGER):INTEGER;
BEGIN UNIXCREATE:=UNIXCREATES(UNIXNAME,MODE); END;

FUNCTION ENTRY UNIXACCESS(VAR UNIXNAME:UNIX_NAME;
 MODE:INTEGER):INTEGER;
BEGIN UNIXACCESS:=UNIXACCESS(S(UNIXNAME,MODE); END;

PROCEDURE ENTRY UNIXCALL;
BEGIN CALLUNIX END;

PROCEDURE ENTRY EXIT;
BEGIN QUIT END;

PROCEDURE INITIALIZE;
VAR I: INTEGER; PARAM: ARGLIST;
 LINE: INTEGER; RESULT: RESULTTYPE;
BEGIN
 INIT OPERATOR(TYPEUSE), OPSTREAM(OPTION),
 INSTREAM(INBUFFER), OUTSTREAM(OUTBUFFER);
 INSTREAM.INITREAD; OUTSTREAM.INITWRITE;
 FOR I:= 1 TO MAXFILE DO
  INIT FILES[I](TYPEUSE, DISKUSE, CATALOG);
 INIT CODE(TYPEUSE, DISKUSE, CATALOG);
 WITH PARAM[2] DO
 BEGIN TAG:= IDTYPE; ARG:= 'CONSOLE32 ' END;
 CALL ('T3,'DO32 ', PARAM, LINE, RESULT);
 OPERATOR.WRITE('JOBPROCESS: (: 10:), 'TERMINATED (: 10:)' END;

BEGIN INITIALIZE END;
TYPE IOPROCESS = PROCESS
   (TYPEUSE: TYPERESOURCE; DISKUSE: RESOURCE;
    CATALOG: DISKCATALOG; SLOWIO: LINEBUFFER;
    BUFFER: PAGEBUFFER; REQUEST, RESPONSE: ARGBUFFER;
    STACK: PROGSTACK; IOTASK: TASKKIND);

"PROGRAM DATA SPACE = " "+2000"

CONST DATASIZE = 12000;
TYPE DATASPACE = ARRAY [1..DATASIZE] OF BYTE;
TYPE FILE = 1..1;

VAR

OPERATOR: TERMINAL; OPSTREAM: TERMINALSTREAM;
ICSTREAM: CHARESTREAM; Iofile: DATAPFILE;
CODE: PROGFILE2;
DATA: DATASPACE;

PROGRAM DRIVER(VAR PARAM: ARGLIST; STORE: PROGSTORE2;
   LEN:INTEGER; VAR DATA:DATASPACE);
ENTRY READ, WRITE, OPEN, CLOSE, GET, PUT, LENGTH,
   MARK, RELEASE, IDENTIFY, ACCEPT, DISPLAY, READPAGE,
   WRITEPAGE, READLINE, WRITELINE, READAPG, WRITEAPG,
   LOOKUP, IOTRANSFER, IOMOVE, TASK, RUN,
   UNIXOPEN, UNIXREAD, UNIXWRITE, UNIXCLOSE, UNIXCREATE,
   UNIXACCESS, UNIXCALL, EXIT;

PROCEDURE CALL(VAR ID: IDENTIFIER; VAR PARAM: ARGLIST;
   VAR LINE: INTEGER; VAR RESULT: RESULTTYPE);
   VAR STATE: PROGSTATE; LASTID: IDENTIFIER;
BEGIN
   WITH CODE, STACK DO
   BEGIN
      LINE:= 0;
      OPEN(ID, STATE);
      IF (STATE = READY) & SPACE THEN
      BEGIN
         PUSH(ID);
         DRIVER(PARAM, STORE, DATASIZE, DATA);
         POP(LINE, RESULT);
      END ELSE
      IF STATE = TOOBIG THEN RESULT:= CODELIMIT
      ELSE RESULT:= CALLERROR;
IF ANY THEN
BEGIN GET(LASTID); OPEN(LASTID, STATE) END;
END;
END;

PROCEDURE ENTRY READ(VAR C: CHAR);
BEGIN IOSTREAM.READ(C) END;

PROCEDURE ENTRY WRITE(C: CHAR);
BEGIN IOSTREAM.WRITE(C) END;

PROCEDURE ENTRY OPEN
(F: FILE; ID: IDENTIFIER; VAR FOUND: BOOLEAN);
BEGIN IOPFILE.OPEN(ID, FOUND) END;

PROCEDURE ENTRY CLOSE(F: FILE);
BEGIN IOPFILE.CLOSE END;

PROCEDURE ENTRY GET(F: FILE; P: INTEGER; VAR BLOCK: PAGE);
BEGIN IOPFILE.READ(P, BLOCK) END;

PROCEDURE ENTRY PUT(F: FILE; P: INTEGER; VAR BLOCK: PAGE);
BEGIN IOPFILE.WRITE(P, BLOCK) END;

FUNCTION ENTRY LENGTH(F: FILE): INTEGER;
BEGIN LENGTH := IOPFILE.LENGTH END;

PROCEDURE ENTRY MARK(VAR TOP: FULLWORD);
BEGIN TOP := ATTRIBUTE(ATTPHEAP) END;

PROCEDURE ENTRY RELEASE(TOP: FULLWORD);
BEGIN SETHEAP(TOP) END;

PROCEDURE ENTRY IDENTIFY(HEADER: LINE);
BEGIN OPSTREAM.RESET(HEADER) END;

PROCEDURE ENTRY ACCEPT(VAR C: CHAR);
BEGIN OPSTREAM.READ(C) END;

PROCEDURE ENTRY DISPLAY(C: CHAR);
BEGIN OPSTREAM.WRITE(C) END;

PROCEDURE ENTRY READPAGE(VAR BLOCK: PAGE; VAR EOF: BOOLEAN);
BEGIN BUFFER.READ(BLOCK, EOF) END;

PROCEDURE ENTRY WRITEPAGE(BLOCK: PAGE; EOF: BOOLEAN);
BEGIN BUFFER.WRITE(BLOCK, EOF) END;

PROCEDURE ENTRY READLINE(VAR TEXT: LINE);
BEGIN SLOWIO.READ(TEXT) END;

PROCEDURE ENTFPY WRITELINE(TEXT: LINE);
BEGIN SLOWIO.WRITE(TEXT) END;
PROCEDURE ENTRY READARG(S: ARGSEQ; VAR ARG: ARGTYPE);
BEGIN REQUEST.READ(ARG) END;

PROCEDURE ENTRY WRITEARG(S: ARGSEQ; ARG: ARGTYPE);
BEGIN RESPONSE.WRITE(ARG) END;

PROCEDURE ENTRY LOOKUP
,ID: IDENTIFIER; VAR ATTR: FILEATTR; VAR FOUND: BOOLEAN);
"PRIVATE DISK IS SEARCHED FIRST, FOLLOWED BY SYSTEM DISK
ONLY SEQCODE FILES ARE FOUND ON THE SYSTEM DISK — KSU"
BEGIN
   CATALOG.LOOKUP(ID, ATTR, FOUND, PPRIvATEDISK);
   IF NOT FOUND THEN BEGIN
      CATALOG.LOOKUP(ID, ATTR, FOUND, SYSTEMDISK);
      FOUND:= FOUND AND (ATTR.KIND = SEQCODE);
   END;
END;

PROCEDURE ENTRY IOTRANSFER
,DEVICE: IODEVICE; VAR PARAM: IOPARAM; VAR BLOCK: PAGE);
BEGIN
   IF (DEVICE = PRIVATE_DISK) OR (DEVICE = SYSTEMDISK) THEN BEGIN
      DISKUSE.REQUEST;
      IO(BLOCK, PARAM, DEVICE);
      WITH PARAM DO
      IF STATUS = ENDFILE THEN
         STATUS:= COMPLETE;
      DISKUSE.RELEASE;
   END ELSE
   ID(BLOCK, PARAM, DEVICE);
END;

PROCEDURE ENTRY IOMove(DEVICE: IODEVICE; VAR PARAM: IOPARAM);
BEGIN IO(PARAM, PARAM, DEVICE) END;

FUNCTION ENTRY TASK: TASKKIND;
BEGIN TASK:= IOTASK END;

PROCEDURE ENTRY RUN
,ID: IDENTIFIER; VAR PARAM: APGLIST;
,VAR LINE: INTEGER; VAR RESULT: REsULTTYPE);
BEGIN CALL(ID, PARAM, LINE, RESULT) END;

FUNCTION ENTRY UNIXOPEN VAR UNIXNAME:UNIX_NAME;
,MODE:INTEGER):INTEGER;
BEGIN UNIXOPEN:=UNIXOPENS(UNIXNAME,MODE) END;

FUNCTION ENTRY UNIXREAD(FILEDES:INTEGER; VAR BUF:PAGE;
,LENGTH:INTEGER):INTEGER;
BEGIN UNIXREAD:=UNIXREADS(FILEDES,BUF,LENGTH) END;

FUNCTION ENTRY UNIXWRITE(FILEDES:INTEGER; BUF:PAGE;
LENGTH:INTEGER:INTEGER;
BEGIN UNIXWRITE:=UNIXWRITES(FILEDES,BUF,LENGTH); END;

FUNCTION ENTRY UNIXCLOSE(FILEDES:INTEGER):INTEGER;
BEGIN UNIXCLOSE:=UNIXCLOSSES(FILEDES); END;

FUNCTION ENTRY UNIXCREATE(VAR UNIXNAME:UNIX_NAME;
MODE:INTEGER):INTEGER;
BEGIN UNIXCREAT:=UNIXCREATESUNIXNAME,MODE); END;

FUNCTION ENTRY UNIXACCESS(VAR UNIXNAME:UNIX_NAME;
MODE:INTEGER):INTEGER;
BEGIN UNIXACCESS:=UNIXACCESESUNIXNAME,MODE); END;

PROCEDURE ENTRY UNIXCALL;
BEGIN CALLUNIX END;

PROCEDURE ENTRY EXIT;
BEGIN QUIT END;

PROCEDURE INITIALIZE;
VAR PARAM: ARGLIST; LINE: INTEGER; RESULT: RESULTTYPE;
BEGIN
INIT OPERATOR(TYPEUSE), OPSTREAM(OPERATOR),
ISTREAM(BUFFER),
IOPFILE(TYPEUSE, DISKUSE, CATALOG),
CODE(TYPEUSE, DISKUSE, CATALOG);
IF IOTASK = INPUTASK THEN IOSTREAM.INITWRITE
ELSE IOSTREAM.INITREAD;
CALL (''IO32 ..... ', PARAM, LINE, RESULT);
OPERATOR.WRITE(''IOPROCESS: (:10:) ', 'TERMINATED (:10:) ');
END;

BEGIN INITIALIZE END;
TYPE CARDPROCESS =
PROCESS
  (TYPEUSE: TYPERESOURCE; BUFFER: LINEBUFFER);

"CARDS PROCESS HAS BEEN MODIFIED FOR THE
  USE OF A VIRTUAL READER - KSU"
  "ENDFILE OR ENDMEDIUM CAUSES THE PROCESS TO TERMINATE"

VAR OPERATOR: TERMINAL; TEXT: LINE;
  PARAM: IOPARAM; OK, ENDP: BOOLEAN;
BEGIN
  INIT OPERATOR(TYPEUSE);
  PARAM.OPERATION := INPUT;
  ENDP := FALSE;
REPEAT
  IO (TEXT, PARAM, CARDDEVICE);
  CASE PARAM.STATUS OF
    COMPLETE:
      OK := TRUE;
    INTERVENTION, FAILURE:
      BEGIN OK := FALSE; WAIT END;
    ENDFILE, ENDMEDIUM: "KSU - SUPPORTS VIRTUAL READER"
      BEGIN OK := TRUE; ENDP := TRUE; END;
    TRANSMISSION:
      BEGIN
        OPERATOR.WRITE('CARDS: (:10:)', 'ERROR (:10:) ');
        OK := FALSE;
      END
    END;
  UNTIL OK;
  BUFFER.WRITE (TEXT);
  UNTIL ENDP;
  OPERATOR.WRITE('CARDS: (:10:)', 'TERMINATED (:10: ');
END;
TYPE PRINTPROCESS =
PROCESS
  (TYPEUSE: TYPERESOURCE; BUFFER: LINEBUFFER);

VAR OPERATOR: TERMINAL; PARAM: IOPARAM;
  TEXT: LINE;

BEGIN
  INIT OPERATOR(TYPEUSE);
  PARAM.OPERATION := OUTPUT;
  PARAM.ARG := 80;
  CYCLE
    BUFFER.READ(TEXT);
    IO(TEXT, PARAM, PRINTDEVICE);
    IF PARAM.STATUS <> COMPLETE THEN
      BEGIN
        OPERATOR.WRITE('PRINTER: (:10:)', 'INSPECT (:10: ');
        REPEAT
          IO(TEXT, PARAM, VARDEVICE);
          UNTIL PARAM.STATUS = COMPLETE;
        END;
      END;
  END;
END;
TYPE LOADERPROCESS=
PROCESS(DISKUSE: RESOURCE);

CONST SOLOADR = 24;
VAR PARAM: IOPARAM; J:INTEGER; LN:FULLWORD;

PROCEDURE INITIALIZE(PAGENO: UNIV IOARG);
BEGIN
  WITH PARAM DO
  BEGIN
    OPERATION:= CONTROL;
    ARG:= 1;
  END;
END;

BEGIN
  J:=SOLOADR; INITIALIZE(J);
  "AWAIT BEL SIGNAL"
  IO(PARAM, PARAM, TYPEDEVICE);
  "LOAD SOLO SYSTEM"
  "DISKUSE.REQUEST;"
  IO(PARAM, PARAM, PRIVATEDISK);
  Diskuse.Release;
END;
VAR

TYPEUSE: TYPERESOURCE;
DISKUSE: RESOURCE; CATALOG: DISKCATALOG;
INBUFFER, OUTBUFFER: PAGEBUFFER;
CARDBUFFER, PRINTBUFFER: LINEBUFFER;
INREQUEST, INRESPONSE, OUREREQUEST, OUTRESPONSE: ARGBUFFER;
INSTACK, OUTSTACK, JOBSTACK: PROGSTACK;
READER: CARDPROCESS; WRITER: PRINTERPROCESS;
PRODUCER, CONSUMER: IOPROCESS; MASTER: JOBPROCESS;

WATCHDOG: LOADERPROCESS;

BEGIN

INIT

TYPEUSE, DISKUSE,
CATALOG (TYPEUSE, DISKUSE, CATADDR),
INBUFFER, OUTBUFFER,
CARDBUFFER, PRINTBUFFER,
INREQUEST, INRESPONSE, OUREREQUEST, OUTRESPONSE,
INSTACK, OUTSTACK, JOBSTACK,
WATCHDOG (DISKUSE),
READER (TYPEUSE, CARDBUFFER),
WRITER (TYPEUSE, PRINTERBUFFER),
PRODUCER (TYPEUSE, DISKUSE, CATALOG, CARDBUFFER,
INBUFFER, INREQUEST, INRESPONSE, INSTACK, INPUTTASK),
CONSUMER (TYPEUSE, DISKUSE, CATALOG, PRINTERBUFFER,
OUTBUFFER, OUREREQUEST, OUTRESPONSE, OUTSTACK,
OUTPUTTASK),
MASTER (TYPEUSE, DISKUSE, CATALOG, INBUFFER, OUTBUFFER,
INREQUEST, INRESPONSE, OUREQUEST, OUTRESPONSE,
JOBSTACK);

END.
APPENDIX D

SEQUENTIAL PASCAL PROGRAMS
"UTILITY PROGRAMS FOR
THE SOLO SYSTEM"

"MODIFIED TO RUN UNDER UNIX BY
MARTIN WILDE
FEBRUARY 1984"

"#############
# PREFIX #
#############"

TYPE INTEGER = SHORTINTEGER;

CONST NL = '(:10:)' ;  FF = '(:12:)' ;  CR = '(:13:)' ;
EM = '(:25:)' ;  NULL = '(:00:)' ;

CONST PAGELENGTH = 512;
TYPE PAGE = ARRAY (.1..PAGELENGTH.) OF CHAR;

CONST LINELENGTH = 132;
TYPE LINE = ARRAY (.1..LINELENGTH.) OF CHAR;

CONST IDLENGTH = 12;
TYPE IDENTIFIER = ARRAY (.1..IDLENGTH.) OF CHAR;

CONST UNIXLENGTH = 13;
TYPE UNIX_NAME = ARRAY (.1..UNIXLENGTH.) OF CHAR;

TYPE FILE = 1..2;

TYPE FILEKIND = (EMPTY, SCRATCH, ASCII, SEQCODE, CONCODE);

TYPE FILEATTR = PACKED RECORD
    KIND: FILEKIND;
    ADDR: INTEGER;
    PROTECTED: BOOLEAN;
    NOTUSED: ARRAY (.1..5.) OF INTEGER
END;

TYPE IODEVICE =
    (TYPEDEVICE, DISKDEVICE, TAPEDEVICE, PRINTDEVICE, CARDDEVICE);

TYPE IOCOMMAND = (INPUT, OUTPUT, MOVE, CONTROL);

TYPE IOARG = (WRITEOF, REWIND, UPSPACE, BACKSPACE);

TYPE IORESULT =
    (COMPLETE, INTERVENTION, TRANSMISSION, FAILURE,
    ENDFILE, ENDMEDIUM, STARTMEDIUM);
TYPE IOPARAM = PACKED RECORD
  OPERATION: IOOPERATION;
  STATUS: IORESULT;
  ARG: 0..32768
END;

TYPE TASKKIND = (INPUTTASK, JOBTASK, OUTPUTTASK);

TYPE ARGTAG =
  (NILTYPE, BOOLTYPE, INTTYPE, IDTYPE, PTRTYPE);

TYPE POINTER = @BOOLEAN;

TYPE ARGTYPE = RECORD
  CASE TAG: ARGTAG OF
    NILTYPE, BOOLTYPE: (BOOL: BOOLEAN);
    INTTYPE: (INT: INTEGER);
    IDTYPE: (ID: IDENTIFIER);
    PTRTYPE: (PTR: POINTER)
  END;

CONST MAXARG = 10;
TYPE ARGLIST = ARRAY (.1..MAXARG.) OF ARGTYPE;

TYPE ARGSEQ = (INP, OUT);

TYPE PROGRESS =
  (TERMINATED, OVERFLOW, POINTERERROR, RANGEERROR,
   VARIANTERROR, HEAPLIMIT, STACKLIMIT, CODELIMIT, TIMELIMIT,
   CALLERROR);

PROCEDURE READ(VAR C: CHAR);
PROCEDURE WRITE(VAR C: CHAR);

PROCEDURE OPEN(VAR F: FILE; ID: IDENTIFIER; VAR FOUND: BOOLEAN);
PROCEDURE CLOSE(VAR F: FILE);
PROCEDURE GET(VAR F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);
PROCEDURE PUT(VAR F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);
FUNCTION LENGTHFILE(VAR F: FILE): INTEGER;

PROCEDURE MARK(VAR TOP: INTEGER);
PROCEDURE RELEASE(VAR TOP: INTEGER);

PROCEDURE IDENTIFY(HEADER: LINE);
PROCEDURE ACCEPT(VAR C: CHAR);
PROCEDURE DISPLAY(VAR C: CHAR);

PROCEDURE READPAGE(VAR BLOCK: UNIV PAGE; VAR EOF: BOOLEAN);
PROCEDURE WRITEPAGE(VAR BLOCK: UNIV PAGE; EOF: BOOLEAN);
PROCEDURE READLINE(VAR TEXT: UNIV LINE);
PROCEDURE WRITELINE(TEXT: UNIV LINE);
PROCEDURE READARG(VAR ARGSEQ: ARGSEQ; VAR ARG: ARGTYPE);
PROCEDURE WRITEARG(VAR ARGSEQ: ARGSEQ; ARG: ARGTYPE);
PROCEDURE LOOKUP (ID: IDENTIFIER; VAR ATTR: FILEATTR;
VAR FOUND: BOOLEAN);

PROCEDURE IOTRANSFER
(DEVICE: IODEVICE; VAR PARAM: IOPARAM;
VAR BLOCK: UNIV PAGE);

PROCEDURE IOMUXE (DEVICE: IODEVICE; VAR PARAM: IOPARAM);

FUNCTION TASK: TASKKIND;

PROCEDURE RUN (ID: IDENTIFIER; VAR PARAM: ARGLIST;
VAR LINE: INTEGER; VAR RESULT: PROGRESS);

FUNCTION UNIXOPEN (VAR UNIXNAME: UNIX_NAME;
MODE: INTEGER): INTEGER;

FUNCTION UNIXREAD (FILEDES: INTEGER; VAR BUF: PAGE;
LENGTH: INTEGER): INTEGER;

FUNCTION UNIXWRITE (FILEDES: INTEGER; BUF: PAGE;
LENGTH: INTEGER): INTEGER;

FUNCTION UNIXCLOSE (FILEDES: INTEGER): INTEGER;

FUNCTION UNIXCREATE (VAR UNIXNAME: UNIX_NAME;
MODE: INTEGER): INTEGER;

FUNCTION UNIXACCESS (VAR UNIXNAME: UNIX_NAME;
MODE: INTEGER): INTEGER;

PROCEDURE CALLUNIX;

PROCEDURE EXIT;

PROGRAM P (VAR PARAM: ARGLIST);
UNIX 32

THE FOLLOWING PROGRAM IS UTILIZED UNDER SOLO TO INPUT AND OUTPUT FILES TO AND FROM UNIX.

INPUT:

WHEN A FILE IS DESIRED TO BE INPUT THAT EXISTS UNDER UNIX THE COMMAND UNIX32 (INPUT,FILENAME, KIND) IS ENTERED. THE UNIX FILE IS CHECKED TO SEE IF THE USER HAS READ ACCESS TO A FILE BEFORE IT IS READ IN. IF THE USER DOES NOT HAVE ACCESS AN ERROR MESSAGE IS PRINTED ENTAILING ACCESS DENIED. IF THE USER DOES HAVE ACCESS TO THE FILE THEN THE FILE IS READ IN AND THE FILE32 PROGRAM IS RUN TO STORE THE FILE ONTO THE VIRTUAL DISK.

OUTPUT:

CONST
MAPLENGTH = 255; "LENGTH OF A SOLO FILE"
CATPAGELength = 16; "NUMBER OF CATALOG PAGES"
NONAME = ' '; "BLANK CATALOG ENTRY"
CATFILE = 1; "CATALOG FILE NUMBER"

TYPE

FILEMAP = PACKED RECORD "TEMPORARY FILE ON VIRTUAL DISK"
  FILELENGTH: SHORTINTEGER;
  PAGESET: ARRAY (.1..MAPLENGTH.) OF SHORTINTEGER
END;

CATENTRY = PACKED RECORD "CATALOG ENTRY FOR A FILE"
  ID: IDENTIFIER;
  ATTR: FILEATTR;
  KEY, SEARCHLENGTH: INTEGER
END;

CATPAGE = ARRAY (.1..CATPAGELength.) OF CATENTRY;
"A CATALOG PAGE"

VAR

CAT: RECORD "PROGRAM VARIABLE OF A FILE"
  INDEX: INTEGER;
  CHANGED: BOOLEAN;
  BLOCK: CATPAGE
END;

CATLENGTH: INTEGER; "LENGTH OF THE CATALOG"
BUCKET: RECORD "FILES PARAMETERS"
  NAME: IDENTIFIER;
  START, LENGTH, INDEX: INTEGER
END;

KIND: FILEKIND; "TYPE OF FILE"
FOUND: BOOLEAN; "INDICATOR OF FILE IS IN CATALOG"
FILENAME: IDENTIFIER; "NAME OF THE FILE"
MAP: FILEMAP; "CONTENTS OF THE NEXT FILE"
LENGTH: INTEGER; "NUMBER OF PAGES IN A FILE"
CK: BOOLEAN; "CONTINUATION INDICATOR"
CONTINUE: BOOLEAN; "CONTINUATION INDICATOR"
DIRECTION: BOOLEAN;
"TRUE FOR INPUT AND FALSE FOR OUTPUT OPERATIONS"
UNIXNAME: UNIX_NAME; "13 CHARACTER UNIXNAME"
PROCEDURE WRITETEXT(TEXT:LINE);

VAR I:INTEGER; C:CHAR; "TEXT POINTER AND OUTPUT CHARACTER"

BEGIN "OUTPUT A LINE OF TEXT TO THE CONSOLE"
  I:=1;
  C:=TEXT[I];
  WHILE C <> '$' DO
    "OUTPUT CHARACTERS UNTIL A DOLLAR SIGN IS FOUND"
    BEGIN
      DISPLAY(C); "DISPLAY CHARACTER"
      I:=I+1;
      C:=TEXT[I]
    END;
    DISPLAY(\n) "FORCE A LINE FEED ON THE CONSOLE"
END; "END OF WRITETEXT"
"******************************************************************
*                                                          *
*      THE PROCEDURE - HELP - PRINTS HELP MESSAGES TO         *
*      THE USER WHEN AN ILLEGAL COMMAND LINE IS ENTERED.     *
*                                                          *
******************************************************************"

PROCEDURE HELP;

BEGIN  "DISPLAY HELP MESSAGES"
   WRITETEXT('TRY AGAIN$');
   WRITETEXT(' UNIX32(INPUT,FILENAME,KIND)$');
   WRITETEXT(' UNIX32(OUTPUT,FILENAME)$');
   WRITETEXT(' WHERE FILENAME:IDENTIFIERS$');
   WRITETEXT(' KIND:(SCRATCH,ASCII,SEQCODE,CONCODE)$');
END;  "END OF HELP MESSAGES"

"******************************************************************
*                                                          *
*      THE PROCEDURE - CHECKKIND - CHECKS THE FILETYPE       *
*      TO SEE IF THE FILE KIND IS LEGAL.                    *
*                                                          *
******************************************************************"

PROCEDURE CHECKKIND(ID:IDENTIFIER; VAR KIND:FILEKIND;
                  VAR OK:BOOLEAN);

BEGIN  "CHECK FILETYPES"
   OK:=TRUE;
   IF ID = 'SCRATCH' THEN KIND:=SCRATCH ELSE IF
   ID = 'ASCII' THEN KIND:=ASCII ELSE IF
   ID = 'SEQCODE' THEN KIND:=SEQCODE ELSE IF
   ID = 'CONCODE' THEN KIND:=CONCODE
   ELSE OK:=FALSE
END;  "END OF CHECKKIND"
PROCEDURE SAVEFILE;

VAR LINE:INTEGER; "LINE OF WHICH ERROR OCCURRED"
RESULT:PROGRESUL; "RESULT OF FILE EXECUTION"
LIST:ARGLIST; "PARAMETERS FOR THE FILE32 PROGRAM"

BEGIN "SAVEFILE"
  WITH LIST[1] DO
    BEGIN "SET TAG TO BE FALSE"
      TAG:=BOOLTYPE;
      BOOL:=FALSE;
    END;

  WITH LIST[2] DO
    BEGIN "SET ACTION TO CREATE"
      TAG:=IDTYPE;
      ID='CREATE';
    END;

  WITH LIST[3] DO
    BEGIN "INCLUDE FILENAME"
      TAG:=IDTYPE;
      ID=FILENAME;
    END;

  WITH LIST[4] DO
    BEGIN "INCLUDE FILELENGTH"
      TAG:=INTTYPE;
      INT=LENGTH;
    END;

  WITH LIST[5] DO
    BEGIN "SET TYPE OF FILE TO BE CREATED"
      TAG:=IDTYPE;
      CASE KIND OF
        SCRATCH:ID='SCRATCH';
        ASCII:ID='ASCII';
        SEQCODE:ID='SEQCODE';
        CONCODE:ID='CONCODE';
      END;
    END;

  WITH LIST[6] DO
    BEGIN "SET LAST TAG TO BE FALSE"
TAG := BOOLEAN;
BOOL := FALSE;
END;
"STORE PROGRAM ON VIRTUAL DISK"
RUN (FILE32 , LIST, LINE, RESULT);
IF (RESULT <> TERMINATED) OR NOT LIST[1].BOOL THEN
  WRITETEXT ("DESTINATION FILE LOST");
  "DISPLAY ANY ERROR MESSAGES"
END; "END OF SAVEFILE"
**THE PROCEDURE - GETUNIXNAME - CONVERTS THE**
**INPUTTED INTO THE PROPER FORMAT FOR A FILE OPEN UNDER**
**UNIX. UNIX REQUIRES THE FILENAME TO BE TRUNCATED BY A**
**NULL SYMBOL.**

```
PROCEDURE GETUNIXNAME(NAME:IDENTIFIER);
VAR I:INTEGER; "ARRAY INDEX"
BEGIN
    UNIXNAME := " ";
    I := 1;
    REPEAT "TRANSFER CHARACTERS FROM ARRAY TO ARRAY"
        UNIXNAME[I] := NAME[I];
        I := I + 1;
    UNTIL (I = 13) OR (NAME[I] = ' ');
    UNIXNAME[I] := NULL;
    "SET LAST POSITION TO BE A NULL FOR UNIX"
END;
```

**THE PROCEDURE - READCATPAGE - READS A PAGE FROM**
**THE CATALOG. THE PAGE TO READ IS PASSED TO IT.**

```
PROCEDURE READCATPAGE(I: INTEGER; VAR ELEM: CATENTRY);
VAR PAGENO: INTEGER; "PAGE NUMBER OF THE FILES PAGE"
BEGIN "READ A CATALOG PAGE"
    WITH CAT DO
    BEGIN
        PAGENO := (I - 1) DIV CATPAGELength + 1;
        IF INDEX <> PAGENO THEN
            BEGIN "SEARCH UNTIL PAGE FOUND"
                INDEX := PAGENO;
                GET(CATFILE, INDEX, BLOCK)
            END;
        ELEM := BLOCK(. (I - 1) MOD CATPAGELength + 1 .)
    END
END
```
FUNCTION HASH (ID: IDENTIFIER): INTEGER;

VAR KEY, I: INTEGER; "TEMPORARY VARIABLES"
C: CHAR;

BEGIN "HASH A FILENAME"
  KEY := 1;
  I := 0;
  REPEAT "CONTINUE UNTIL END OF THE NAME REACHED"
    I := I + 1;
    C := ID(.I.);
    IF C <> "'" THEN KEY := KEY*ORD(C) MOD CATLENGTH + 1
      "FOLDING OF NAME"
    UNTIL (C = "'") OR (I = IDLENGTH);
  HASH := KEY
END;

PROCEDURE GETCAT (I: INTEGER);

VAR PAGENO: INTEGER; "PAGE NUMBER OF THE FILE"

BEGIN "GET CATALOG FILE"
  WITH CAT DO
    BEGIN
      PAGENO := (I - 1) DIV CATPAGELength + 1;
      IF INDEX <> PAGENO THEN
        BEGIN "KEEP SEARCHING UNTIL PAGE FOUND"
          IF CHANGED THEN PUT(CATFILE, INDEX, BLOCK);
          INDEX := PAGENO;
          GET(CATFILE, INDEX, BLOCK);
          CHANGED := FALSE
        END
      END
***************
* THE PROCEDURE - READCAT - READS IN THE CATALOG. *

***************
PROCEDURE READCAT (I: INTEGER; VAR ELEM: CATENTRY);
BEGIN
  "INPUT CATALOG"
  WITH CAT DO
  BEGIN
    GETCAT(I);
    ELEM := BLOCK(. (I-1) MOD CATPAGELENGTH + 1.)
  END
END;

***************
* THE PROCEDURE - INITBUCKET - SETS UP THE FILES *
* PARAMETERS FOR SEARCHING OF THE CATALOG. *

***************
PROCEDURE INITBUCKET (ID: IDENTIFIER);
BEGIN
  "INITIALIZE FILES PARAMETERS"
  WITH BUCKET DO
  BEGIN
    START := HASH(ID);
    READCAT(START, ELEM);
    NAME := ELEM.ID;
    LENGTH := ELEM.SEARCHLENGTH;
    INDEX := START
  END
END;

THE PROCEDURE - SEARCHOLD - SEARCHES THE CATALOG FOR THE FILENAME AND RETURNS ITS RESULT IN THE FILES PARAMETER LIST.

PROCEDURE SEARCHOLD (ID: IDENTIFIER);

VAR MORE: INTEGER; "NUMBER OF PAGES TO SEARCH"
FOUND: BOOLEAN; "INDICATOR OF WHETHER FILE WAS FOUND"
ELEM: CATEGTRY; "FILES CATALOG ENTRY"

BEGIN "SEARCH FOR FILENAME IN CATALOG"
INIBUCKET(ID);
WITH BUCKET DO IF ID <> NAME THEN
BEGIN
  MORE := LENGTH;
  INDEX := START MOD CATLENGTH + 1;
  FOUND := FALSE;

  WHILE (MORE>0) & NOT FOUND DO
    BEGIN "LOOK FOR FILENAME"
      READCAT(INDEX, ELEM);
      NAME := ELEM.ID;
      IF ID = NAME THEN FOUND := TRUE
    ELSE
      BEGIN "NOT FOUND KEEP LOOKING"
        IF ELEM.KEY=START THEN MORE:=MORE-1;
        INDEX := INDEX MOD CATLENGTH + 1
      END
    END
  END
END
"***********************************************************************
* * THE PROCEDURE - INITCAT - OPENS THE CATALOG FILE * *
* AND SETS ITS PARAMETERS. * *
* ***********************************************************************"

PROCEDURE INITCAT;

VAR FOUND: BOOLEAN; "INDICATOR OF WHETHER FILE WAS FOUND"

BEGIN
  OPEN (CATFILE, 'CATALOG ', FOUND);
  WITH CAT DO
    BEGIN
      INDEX := 0;
      CHANGED := FALSE
    END
  END;

"***********************************************************************
* * THE PROCEDURE - INITFILES - Initializes the * *
* CATALOG FOR SEARCHING OF FILENAMES. * *
* ***********************************************************************"

PROCEDURE INITFILES;

BEGIN "INITIALIZE CATALOG"
  INITCAT;
  CATLENGTH := LENGTHFILE (CATFILE) * CATPAGELength;
END;
"***********************************************************************
*                 THE PROCEDURE - READATTR - READS THE FILES             *
*                 ATTRIBUTES.                                          *
*                                                                *
***********************************************************************"

PROCEDURE READATTR (ID: IDENTIFIER; VAR ATTR: FILEATTR);

VAR ELEM: CATENTRY; "FILES CATALOG ENTRY"

BEGIN "LOCATE FILE AND THEN READ ATTRIBUTES"
  SEARCHOLD(ID);
  WITH BUCKET DO
    BEGIN
      READCAT(INDEX, ELEM);
      ATTR := ELEM.ATTR
    END
  END;

"***********************************************************************
*                 THE PROCEDURE - CONTAINS - LOOKS FOR THE FILE IN      *
*                 THE CATALOG.                                        *
***********************************************************************"

FUNCTION CONTAINS (ID: IDENTIFIER) : BOOLEAN;

BEGIN
  SEARCHOLD(ID);
  WITH BUCKET DO CONTAINS := (NAME=ID) & (ID<>NONAME)
END;
PROCEDURE OBTAINPAGES (FILENAME: IDENTIFIER);

VAR PARAM: IOPARAM; "IO PARAMETERS"
    ATTR: FILEATTR; "FILE ATTRIBUTES"

BEGIN "OBTAINPAGES"
    FOUND:=FALSE;
    IF CONTAINS (FILENAME) THEN
        BEGIN "FILE FOUND IN SOLO CATALOG"
            FOUND:=TRUE;
            READATTR (FILENAME, ATTR);
            "OBTAIN FILE'S ATTRIBUTES"
            PARAM. ARG:= ATTR. ADDR;
            PARAM. OPERATION:= INPUT;
            IOTransfer (DISKDEVICE, PARAM, MAP);
            "OBTAIN THE FILES PAGEMAP"
        END;
    END;
END; "END OF OBTAINPAGES"
* THE PROCEDURE - CHECKARG - VALIDATES THE * PARAMETERS PASSED TO IT UPON TIME OF INVOCATION. IF * ANY OF THE PARAMETERS ARE INVALID, THE ROUTINE CALLS * THE HELP ROUTINE AND TERMINATES THE PROGRAM. *

PROCEDURE CHECKARG;

VAR
   DIRECT: ARGTYPE; "INVOCATION PARAMETERS"
   MORE: BOOLEAN; "VALIDITY INDICATOR"

BEGIN "CHECKARG"
   MORE:=FALSE;
   CONTINUE:=FALSE;
   DIRECT:=PARAM[2];
   WITH DIRECT DO "CHECK CORRECT COMMAND"
      IF TAG <> IDTYPE THEN HELP
      ELSE "FIRST ARGUMENT IS A COMMAND"
         BEGIN
            IF ID='INPUT' THEN
               BEGIN "COMMAND IS INPUT"
                  DIRECTION:=TRUE;
                  MORE:=TRUE;
                  CONTINUE:=TRUE;
               END
               ELSE "CHECK FOR OUTPUT COMMAND"
                   IF ID='OUTPUT' THEN
                        BEGIN "COMMAND IS OUTPUT"
                            DIRECTION:=FALSE;
                            OK:=TRUE;
                            CONTINUE:=TRUE;
                        END
                        ELSE "ARGUMENT IS NEITHER INPUT OR OUTPUT"
                           HELP
               END;
            IF CONTINUE THEN
               DIRECT:=PARAM[3];
               WITH DIRECT DO "CHECK FOR FILENAME ON INPUT"
                  IF TAG <> IDTYPE THEN
                     BEGIN
                        CONTINUE:=FALSE;
                        HELP
                     END
                  ELSE "TAG IS OK, OBTAIN FILENAME"
                     FILENAME:=ID;
IF (CONTINUE) AND (MORE) THEN
  BEGIN "IF INPUT THEN OBTAIN INPUT FILES FILETYPE"
  DIRECT:=PARAM[4];
  IF TAG <> IDTYPE THEN
    BEGIN
      CONTINUE:=FALSE;
      HELP
    END
  ELSE
    WITH DIRECT DO
    BEGIN
      CHECKKIND(ID,KIND,OK);
      IF NOT OK THEN
        BEGIN "FILETYPE WAS ILLEGAL"
          HELP;
          CONTINUE:=FALSE;
        END;
    END;
  END;
END: "END OF CHECKARG"
THE PROCEDURE - INPUTFILE - INPUTS A FILE FROM THE UNIX FILE SYSTEM. BEFORE THE FILE IS INPUTTED, THE ACCESS RIGHTS FOR THE USER TO READ THE FILE IS CHECKED. IF THE RIGHTS ARE DENIED, AN ERROR MESSAGE IS PRINTED.

PROCEDURE INPUTFILE;

VAR
  I, K: INTEGER;            "ARRAY POINTERS"
  LENGTH: INTEGER;         "LENGTH OF UNIX FILE"
  BUF, BUFTEMP: PAGE;      "FILE BUFFERS"
  PARAM, IOPARAM: INTEGER; "FILE DESCRIPTOR"
  LIMIT: BOOLEAN;          "INDICATOR OF SOLO LENGTH EXCEEDED"
  ID: IDENTIFIER;           "SOLO FILE NAME"
  RESULT: INTEGER;         "INDICATOR OF UNIX FILE CLOSURE"
  MODE: INTEGER;           "UNIX TYPE OF FILE MANIPULATION"
  ACCESSES: INTEGER;       "RESULT OF UNIX FILE ACCESS CHECK"
  SHORT: BOOLEAN;          "LENGTH INDICATOR"

BEGIN "INPUT FILE"
  ID := 'NEXT';
  OBTAINPAGES(ID);        "OBTAIN FREE PAGEMAP"
  LIMIT := FALSE;
  GETUNIXNAME(Filename);  "CONVERT FILENAME"
  MODE := 0;              "TEST FOR UNIX FILE IN SYSTEM"
  ACCESSES := UNIXACCESS(UNIXNAME, MODE);
  IF ACCESSES = -1 THEN "FILE NOT KNOWN TO UNIX"
      WRITE TEXT('FILE UNKNOWN TO UNIX')
  ELSE "CHECK USER ACCESS TO THE FILE"
    BEGIN "OPEN THE FILE"
      MODE := 0;
      FILEDES := UNIXOPEN(UNIXNAME, MODE);
      I := 1;
      LENGTH := 0;
      SHORT := TRUE;
      PARAM.OPERATION := OUTPUT;
      FLENGTH := UNIXREAD(FILEDES, BUF, PAGELENGTH);
      WHILE (FLENGTH = 512) AND (SHORT) DO
        BEGIN "READ PAGES IN UNTIL UNIX EOP FOUND"
PARAM_ARG:=MAP_PAGESET"I";
IOTRANSFER(DISKDEVICE,PARAM,BUF);
I:=I+1;
LENGTH:=LENGTH+1;
FLENGTH:=UNIXREAD(FILEDES,BUF,PAGELength);
IF LENGTH = 256 THEN
  SHORT:=FALSE;
END;

IF (FLENGTH < 512) AND
  (FLENGTH > 0) AND (SHORT) THEN
BEGIN "FILE BLOCK CONTAINS END OF FILE"
  K:=1;
  REPEAT "COPY CHARACTERS INTO TEMP BUFFER"
    BUTEMP"K":=BUF"K";
    K:=K+1;
  UNTIL K=FLENGTH + 1;
  BUTEMP"K":=EN; "PLACE SOLO EOF INTO BUFFER"
  K:=K+1;
  REPEAT "PLACE NULL CHARACTERS AFTER EOF"
    BUTEMP"K":=NULL;
    K:=K+1;
  UNTIL K = 513; "STORE FILE ON DISK"
  PARAM_ARG:=MAP_PAGESET"I";
  IOTRANSFER(DISKDEVICE,PARAM,BUTEMP);
  I:=I+1;
  LENGTH:=LENGTH+1;
END ELSE

  IF (FLENGTH = 0) AND (SHORT) THEN
  BEGIN "EOF AT END OF FILE BLOCK"
    BUTEMP"1":=EN; "STORE SOLO EOF"
    PARAM_ARG:=MAP_PAGESET"I";
    IOTRANSFER(DISKDEVICE,PARAM,BUTEMP);
    LENGTH:=LENGTH+1;
  END;

  IF NOT SHORT THEN
  BEGIN "SOLO FILE LENGTH EXCEEDED"
    LIMIT:=TRUE;
    WRITETEXT('SOLO FILE LIMIT EXCEEDED$')
  END;

  IF NOT LIMIT THEN
  SAVEFILE; "FILE LENGTH NOT EXCEEDED, SAVE FILE"

  RESULT:=UNIXCLOSE(FILEDES); "CLOSE UNIX FILE"
END;
END;
END; "END OF INPUTFILE"
THE PROCEDURE - OUTPUTFILE - OUTPUTS A FILE FROM * THE VIRTUAL DISK TO THE UNIX FILE SYSTEM. IF THE FILE * DOES NOT ALREADY EXIST IN THE UNIX FILE SYSTEM, THE * FILE IS CREATED WITH GENERAL ACCESS RIGHTS. OTHERWISE * THE FILE IS CHECKED TO SEE IF THE USER HAS ACCESS * RIGHTS TO WRITE TO THE FILE. IF THE USER DOES NOT, AN * ERROR MESSAGE IS PRINTED.

PROCEDURE OUTPUTFILE;

VAR
  I,T,L:INTEGER;            "ARRAY POINTERS"
  LENGTH:INTEGER;           "UNIX FILE LENGTH"
  BUF:PAGE;                 "TEMPORARY FILE BUFFER"
  PARAM:TOPARAM;            "IO PARAMETERS"
  FILEDES:INTEGER;          "UNIX FILE DESCRIPTOR"
  LIMIT:BOOLEAN;            "INDICATOR OF SOLO LENGTH EXCEEDED"
  ID:IDENTIFIER;            "FILE NAME"
  RESULT:INTEGER;           "INDICATOR OF RESULT OF UNIX CLOSE"
  MODE:INTEGER;             "ACCESS MODE"
  ACCESSSES:INTEGER;        "INDICATOR OF UNIX ACCESS RESULT"
  PERFORM:BOOLEAN;          "INDICATOR OF FILE OPENED"

BEGIN "OUTPUT FILE TO UNIX"
  PERFORM:=TRUE;
  OBTAINPAGES (FILENAME);  "GET FILES PAGE MAP"
  IF NOT FOUND THEN "FILE NOT FOUND ON DISK"
    WRITETEXT ('SOLO FILE NOT FOUND$')
  ELSE
    BEGIN "FILE FOUND"
      GETUNIXNAME (FILENAME);  "OBTAIN FILE NAME"
      MODE:=0;
      ACCESSSES:=UNIXACCESS (UNIXNAME,MODE);
      IF ACCESSSES < 0 THEN
        BEGIN "FILE NOT FOUND IN UNIX, CREATE IT"
          MODE:=2;  "SET UNIX MODE FOR READ/WRITE ACCESS"
          FILEDES:=UNIXCREATE (UNIXNAME,MODE);
        END
      ELSE
        BEGIN
          MODE:=1;
          FILEDES:=UNIXOPEN (UNIXNAME,MODE);  "OPEN FILE"
          IF FILEDES < 0 THEN
            BEGIN
              WRITETEXT ('WRITE ACCESS DENIED$');
              PERFORM:=FALSE;
            END
        END
  END

END
END;

IF PERFORM THEN
BEGIN "FILE OPENED AND USER HAS PROPER ACCESS"
I := 1;
PARAM. OPERATION := INPUT; "OBTAIN A PAGE FROM SOLO"
LENGTH := MAP. FILELENGTH;
WHILE LENGTH > 1 DO
BEGIN "SEND PAGES TO UNIX"
PARAM. ARG := MAP. PAGESET"I";
IOTRANSFER (DISKDEVICE, PARAM, BUF);
PLENGTH := UNIXWRITE (FILEDES, BUF, PAGELength);
I := I + 1;
LENGTH := LENGTH - 1;
END;

L := 1; "LAST SOLO PAGE REQUIRES SPECIAL HANDLING"
PARAM. ARG := MAP. PAGESET"I";
IOTRANSFER (DISKDEVICE, PARAM, BUF);
WHILE (BUF"L" <> EM) AND (L<512) DO
L := L + 1;
"STORE FILE BLOCKS DATA BEFORE EOF IN UNIX FILE"
 IF (L=512) AND (BUF"L" <> EM) THEN
    PLENGTH := UNIXWRITE (FILEDES, BUF, PAGELength)
ELSE
BEGIN "SOLO EOF AT THE END OF THE FILE BLOCK"
L := L - 1; "STORE ENTIRE BLOCK IN UNIX"
PLENGTH := UNIXWRITE (FILEDES, BUF, L);
RESULT := UNIXCLOSE (FILEDES);
END
END;
END; "END OF OUTPUTFILE"
PROCEDURE INITIALIZE;
BEGIN  "IDENTIFY PROGRAM AND CHECK PARAMETERS"
       IDENTIFY('UNIX= (:10:)');
       DISPLAY(NL);
       INITFILES;
       CHECKARG;
END;  "END OF INITIALIZE"

BEGIN  "START JOB"
   IF TASK = JOBTASK THEN
      BEGIN "CORRECT INVOCATION"
         INITIALIZE;
      END;
   IF CONTINUE THEN
      BEGIN "DO UNIX FILE HANDLING"
         IF DIRECTION = TRUE THEN
            INPUTFILE "INPUT A FILE FROM UNIX"
         ELSE
            OUTPUTFILE; "OUTPUT A FILE TO UNIX"
      END;
END;  "END OF UNIX32 PROGRAM"
"UTILITY PROGRAMS FOR
THE SOLO SYSTEM"

"MODIFIED TO RUN UNDER UNIX BY
MARTIN WILDE

FEBRUARY 1984"

"###
# PREFIX #
###"

TYPE INTEGER = SHORTINTEGER;

CONST NL = '(:10:)' ;  FF = '(:12:)' ;  CR = '(:13:)' ;
EM = '(:25:)' ;  NULL = '(:00:)' ;

CONST PAGELength = 512;
TYPE PAGE = ARRAY (.1..PAGELength.) OF CHAR;

CONST LINELENGTH = 132;
TYPE LINE = ARRAY (.1..LINELENGTH.) OF CHAR;

CONST IDLengTH = 12;
TYPE IDENTIFIER = ARRAY (.1..IDLengTH.) OF CHAR;

CONST UNIXLENGTH = 13;
TYPE UNIX_NAME = ARRAY (.1..UNIXLENGTH.) OF CHAR;

TYPE FILE = 1..2;

TYPE FILEKIND = (EMPTY, SCRATCH, ASCII, SEQCODE, CONCODE);

TYPE FILEATTR = PACKED RECORD
  KIND: FILEKIND;
  ADDR: INTEGER;
  PROTECTED: BOOLEAN;
  NOTUSED: ARRAY (.1..5.) OF INTEGER
END;

TYPE IODEVICE =
  (TYPEDEVICE, DISKDEVICE, TAPEDEVICE, PRINTDEVICE,
   CARDDEVICE);

TYPE IOOPERATION = (INPUT, OUTPUT, MOVE, CONTROL);

TYPE IOARG = (WRITEEOF, REWIND, UPPSPACE, BACKSPACE);

TYPE IORESULT =
  (COMPLETE, INTERVENTION, TRANSMISSION, FAILURE,
   ENDPFILE, ENDMEDIUM, STARTMEDIUM);
TYPE IOPARAM = RECORD
  OPERATION: IOOPERATION;
  STATUS: IORESULT;
  ARG: 0..32768
END;

TYPE TASKKIND = (INPUTTASK, JOBTASK, OUTPUTTASK);

TYPE ARGTAG =
  (NILTYPE, BOOLTYPE, INTTYPE, IDTYPE, PTRTYPE);

TYPE POINTER = 3 BOOLEAN;

TYPE ARGTYPE = RECORD
  CASE TAG: ARGTAG OF
    NILTYPE, BOOLTYPE: (BOOL: BOOLEAN);
    INTTYPE: (INT: INTEGER);
    IDTYPE: (ID: IDENTIFIER);
    PTRTYPE: (PTR: POINTER)
END;

CONST MAXARG = 10;
TYPE ARGLIST = ARRAY [.1..MAXARG.] OF ARGTYPE;

TYPE ARGSEQ = (INP, OUT);

TYPE PROGRESS =
  (TERMINATED, OVERFLOW, POINTERERROR, RANGEERROR,
   VARIANTERROR, HEAPLIMIT, STACKLIMIT, CODELIMIT, TIMELIMIT, CALLErrOR);

PROCEDURE READ(VAR C: CHAR);
PROCEDURE WRITE(C: CHAR);
PROCEDURE OPEN(F: FILE; ID: IDENTIFIER; VAR FOUND: BOOLEAN);
PROCEDURE CLOSE(F: FILE);
PROCEDURE GET(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);
PROCEDURE PUT(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);
PROCEDURE LENGTH(F: FILE): INTEGER;
PROCEDURE MARK(VAR TOP: INTEGER);
PROCEDURE RELEASE(TOP: INTEGER);
PROCEDURE IDENTIFY(HEADER: LINE);
PROCEDURE ACCEPT(VAR C: CHAR);
PROCEDURE DISPLAY(C: CHAR);
PROCEDURE READPAGE(VAR BLOCK: UNIV PAGE; VAR EOF: BOOLEAN);
PROCEDURE WRITEPAGE(BLOCK: UNIV PAGE; EOF: BOOLEAN);
PROCEDURE READLINE(VAR TEXT: UNIV LINE);
PROCEDURE WRITELINE(TEXT: UNIV LINE);
PROCEDURE READARG(S: ARGSEQ; VAR ARG: ARGTYPE);
PROCEDURE WRITEARG(S: ARGSEQ; ARG: ARGTYPE);
PROCEDURE LOOKUP (ID: IDENTIFIER; VAR ATTR: FILEATTR;
VAR FOUND: BOOLEAN);

PROCEDURE IOTRANSFER
(DEVICE: IODEVICE; VAR PARAM: IOPARAM;
VAR BLOCK: UNIV PAGE);

PROCEDURE IOMOVE (DEVICE: IODEVICE; VAR PARAM: IOPARAM);

FUNCTION TASK: TASKKIND;

PROCEDURE RUN (ID: IDENTIFIER; VAR PARAM: ARGLIST;
VAR LINE: INTEGER; VAR RESULT: PROGRRESULT);

FUNCTION UNIXOPEN (VAR UNIXNAME: UNIX_NAME;
MODE: INTEGER): INTEGER;

FUNCTION UNIXREAD (FILEDES: INTEGER; VAR BUF: PAGE;
LENGTH: INTEGER): INTEGER;

FUNCTION UNIXWRITE (FILEDES: INTEGER; BUF: PAGE;
LENGTH: INTEGER): INTEGER;

FUNCTION UNIXCLOSE (FILEDES: INTEGER): INTEGER;

FUNCTION UNIXCREATE (VAR UNIXNAME: UNIX_NAME;
MODE: INTEGER): INTEGER;

FUNCTION UNIXACCESS (VAR UNIXNAME: UNIX_NAME;
MODE: INTEGER): INTEGER;

PROCEDURE CALLUNIX;

PROCEDURE EXIT;

PROGRAM P (VAR PARAM : ARGLIST);
CALLUNIX 32

THE FOLLOWING PROGRAM IS UTILIZED UNDER SOLO TO COMMUNICATE TO THE UNIX ENVIRONMENT. THE PROGRAM ALLOWS THE USER TO EXECUTE UNIX FILES UNDER SOLO.

INPUT:
THE INPUT TO THE PROGRAM IS THE DESIRED UNIX COMMAND TO BE EXECUTED.

OUTPUT:
THE OUTPUT OF THE PROGRAM IS MESSAGES INFORMING THE USER OF WHICH ENVIRONMENT THAT THEY ARE EXECUTING IN. IN ADDITION THESE MESSAGES THE NORMAL UNIX PROMPT OF A DOLLAR SIGN ($) IS DISPLAYED UPON THE ENTERING OF THE UNIX SHELL.

THE FOLLOWING ROUTINE OUTPUTS A LINE OF TEXT TO THE CONSOLE. THE ROUTINE OUTPUTS CHARACTERS TO THE CONSOLE UNTIL A DOLLAR SIGN IS FOUND IN THE TEXT STRING.

PROCEDURE Writetext(Text:Line);

VAR
I: INTEGER; "INDEX OF TEXT ARRAY"
C: CHAR; "CHARACTER TO BE DISPLAYED"

BEGIN
"DISPLAY TEXT LINE"
I:=1;
C:=Text[I];
WHILE C <> '$' DO
"DISPLAY CHARACTERS UNTIL THE DOLLAR SIGN IS FOUND"
BEGIN
    DISPLAY(C); "DISPLAY CHARACTER"
    I:=I+1;
    C:=Text[I];
END;

DISPLAY(NL) "SEND A NEW LINE CHARACTER TO THE CONSOLE"
END; "END OF Writetext"
**THE EXECUTE ROUTINE PERFORMS THE ACTUAL UNIX INTERFACE. THE NECESSARY INTERFACING IS DONE IN THE CONCURRENT PASCAL KERNEL. THE ROUTINE PLACES THE USER IN THE UNIX SHELL. ONCE THE UNIX PROGRAM IS EXECUTED, CONTROL IS RETURNED TO THE SOLO ENVIRONMENT.**

PROCEDURE EXECUTE;

BEGIN "EXECUTE UNIX SHELL COMMAND"
  DISPLAY(NL);
  WRITETEXT('ENTERING THE UNIX ENVIRONMENT: $');
  DISPLAY(NL);
  CALLUNIX; "ENTER THE UNIX SHELL"
  DISPLAY(NL);
  WRITETEXT('BACK AT THE SOLO ENVIRONMENT...$');
END; "END OF PROCEDURE EXECUTE"

**THE MAIN PROGRAM INITIATES THE UNIX INTERFACE BY ACCEPTING THE JOB AND THEN CONTROL IS TRANSFERRED TO EXECUTE ROUTINE.**

BEGIN "START UNIX INTERFACE"
  IDENTIFY('CALLUNIX: (:10:)');
  IF TASK = JOBTASK THEN
    BEGIN "PLACE USER IN THE UNIX SHELL"
      EXECUTE
    END
END

END. "END OF CALLUNIX PROGRAM"
"UTILITY PROGRAMS FOR
THE SOLO SYSTEM"

"MODIFIED TO RUN UNDER UNIX
MARTIN WILDE
DECEMBER 1983"

"#########
# PREFIX #
#########

TYPE INTEGER = SHORTINTEGER;

CONST NL = '(:10:)' ; FF = '(:12:)' ; CR = '(:13:)' ;
EM = '(:25:)' ; NULL = '(:00:)' ;

CONST PAGELENGTH = 512;
TYPE PAGE = ARRAY (.1..PAGELENGTH.) OF CHAR;

CONST LINELENGTH = 132;
TYPE LINE = ARRAY (.1..LINELENGTH.) OF CHAR;

CONST IDLENGTH = 12;
TYPE IDENTIFIER = ARRAY (.1..IDLENGTH.) OF CHAR;

CONST UNIXLENGTH = 13;
TYPE UNIX_NAME = ARRAY (.1..UNIXLENGTH.) OF CHAR;

TYPE FILE = 1..2;

TYPE FILEKIND = (EMPTY, SCRATCH, ASCII, SEQCODE, CONCODE);

TYPE FILEATTR = PACKED RECORD
  KIND: FILEKIND;
  ADDR: INTEGER;
  PROTECTED: BOOLEAN;
  NOTUSED: ARRAY (.1..5.) OF INTEGER
END;

TYPE IODEVICE =
  (TYPEDEVICE, DISKDEVICE, TAPEDEVICE, VAPEDEVICE,
  CARDDEVICE);

TYPE IOOPERATION = (INPUT, OUTPUT, MOVE, CONTROL);

TYPE IOARG = (WRITEEOF, REWIND, UPSPACE, BACKSPACE);

TYPE IORESULT =
  (COMPLETE, INTERVENTION, TRANSMISSION, FAILURE,
  ENDFILE, ENDMEDIUM, STARTMEDIUM);

TYPE IOPARAM = PACKED RECORD
OPERATION: IOOPERATION;
STATUS: IORESULT;
ARG: 0..32768
END;

TYPE TASKKIND = (INPUTTASK, JOBTASK, OUTPUTTASK);

TYPE ARGTAG =
(NILTYPE, BOOLTYPE, INTTYPE, IDTYPE, PTRTYPE);

TYPE POINTER = @BOOLEAN;

TYPE ARGTYPE = RECORD
CASE TAG: ARGTAG OF
NILTYPE, BOOLTYPE: (BOOL: BOOLEAN);
INTTYPE: (INT: INTEGER);
IDTYPE: (ID: IDENTIFIER);
PTRTYPE: (PTR: POINTER)
END;

CONST MAXARG = 10;
TYPE ARGLIST = ARRAY (.1..MAXARG.) OF ARGTYPE;

TYPE ARGSEQ = (IFP, OUT);

TYPE PROGRESUT =
(TERMINATED, OVERFLOW, POINTERERROR, RANGEERROR,
VARIANTERROR, HEAPLIMIT, STACKLIMIT, CODELIMIT, TIMELIMIT,
CALLERROR);

PROCEDURE READ(VAR C: CHAR);
PROCEDURE WRITE(C: CHAR);
PROCEDURE OPEN(F: FILE; ID: IDENTIFIER; VAR FOUND: BOOLEAN);
PROCEDURE CLOSE(F: FILE);
PROCEDURE GET(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);
PROCEDURE PUT(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);
FUNCTION LENGTH(F: FILE): INTEGER;

PROCEDURE MARK(VAR TOP: INTEGER);
PROCEDURE RELEASE(TOP: INTEGER);

PROCEDURE IDENTIFY(HEADER: LINE);
PROCEDURE ACCEPT(VAR C: CHAR);
PROCEDURE DISPLAY(C: CHAR);

PROCEDURE READPAGE(VAR BLOCK: UNIV PAGE; VAR EOF: BOOLEAN);
PROCEDURE WRITEPAGE(BLOCK: UNIV PAGE; EOF: BOOLEAN);
PROCEDURE READLINE(VAR TEXT: UNIV LINE);
PROCEDURE WRITELINE(TEXT: UNIV LINE);
PROCEDURE READARG(S: ARGSEQ; VAR ARG: ARGTYPE);
PROCEDURE WRITEARG(S: ARGSEQ; ARG: ARGTYPE);
PROCEDURE LOOKUP(ID: IDENTIFIER; VAR ATTR: FILEATTR;
VAR POUND: BOOLEAN);
PROCEDURE IOTRANSFER
 (DEVICE: IODEVICE; VAR PARAM: IOPARAM;
  VAR BLOCK: UNIV PAGE);
PROCEDURE IOMOVE(DEVICE: IODEVICE; VAR PARAM: IOPARAM);

FUNCTION TASK: TASKKIND;

PROCEDURE RUN(ID: IDENTIFIER; VAR PARAM: ARGLIST;
  VAR LINE: INTEGER; VAR RESULT: PROGRESS);

FUNCTION UNIXOPEN(VAR UNIXNAME:UNIX_NAME;
  MODE:INTEGER):INTEGER;

FUNCTION UNIXREAD(FILEDES:INTEGER; VAR BUF:PAGE;
  LENGTH:INTEGER):INTEGER;

FUNCTION UNIXWRITE(FILEDES:INTEGER; VAR BUF:PAGE;
  LENGTH:INTEGER):INTEGER;

FUNCTION UNIXCLOSE(FILEDES:INTEGER):INTEGER;

FUNCTION UNIXCREAT(VAR UNIXNAME:UNIX_NAME;
  MODE:INTEGER):INTEGER;

FUNCTION UNIXACCESS(VAR UNIXNAME:UNIX_NAME;
  MODE:INTEGER):INTEGER;

PROCEDURE CALLUNIX;

PROCEDURE EXIT;

PROGRAM P(VAR PARAM: ARGLIST);
**DISKSPACE**

THE FOLLOWING PROGRAM DISPLAYS TO THE USER THE
AMOUNT OF FREE PAGES THAT ARE AVAILABLE ON THE
VIRTUAL DISK. THE PROGRAM READS THE FREE SPACE
FROM THE FREE PAGE INDICATOR VALUE THAT IS STORED
IN THE FILE CALLED NEXT ON THE VIRTUAL DISK.

INPUT:
NONE

OUTPUT:
THE NUMBER OF FREE PAGES THAT EXIST ON THE
VIRTUAL DISK.

CONST
FIVECYLINDERLIMIT = 119; "DISK PARAMETER"
FREEPAGELIMIT = 30; "DISK PARAMETER"
SETSIZE = 8; "DISK PARAMETER"
FIVECYLINDERSIZE = 120; "DISK PARAMETER"
ADDR = 152; "ADDRESS OF FREE FILE ON DISK"

TYPE
INTEGER = SHORTINTEGER; "16 BIT INTEGER"
FIVECYLINDER = SET OF 0..FIVECYLINDERLIMIT;
"SET OF CYLINDERS"
FREEPAGE = ARRAY(0..FREEPAGELIMIT,) OF FIVECYLINDER;
FREEONDISK = PACKED RECORD "FREE FILE"
FREEPAGEONDISK:FREEPAGE;
MISCELLANEOUS:ARRAY(1..SETSIZE.)
OF SHORTINTEGER
END;

VAR
FREE:FREEONDISK; "FREE FILE CONTENTS"
FIRSTPAGE:SHORTINTEGER; "FIRST PAGE OF FILE"
FIRSTFREEPAGE:SHORTINTEGER;
"VALUE OF FIRST FREE PAGE AVAILABLE"
THE ROUTINE WRITEINT OUTPUTS AN INTEGER(S) TO THE CONSOLE FOR THE USER. THE ROUTINE CONVERTS INTEGERS TO THE EQUIVALENT CHARACTER VALUE AND THEN OUTPUTS THE STRING AS CHARACTERS.

PROCEDURE WRITEINT (I:SHORTINTEGER);

VAR
J,K:INTEGER; "ARRAY POINTERS"
CHARS:LINE; "INTEGER STRING"

BEGIN "OUTPUT INTEGERS"
FOR K:=1 TO 8 DO CHAR(.,K.):=' ';
J:=ABS(I);
K:=8;
REPEAT CHAR(.,K.):=CHR((J MOD 10) + 48);
"CONVERT INTEGER TO CHARACTER"
J:=J DIV 10;
K:=K-1;
UNTIL (J=0);
CHAR(.,9.):=NL;
FOR K:=1 TO 9 DO
"OUTPUT INTEGER(S) AS A CHARACTER STRING"
DISPLAY(CHAR(.,K.));
END; "END OF WRITEINT"
PROCEDURE INITIALIZE;

VAR
   PARAM:IOPARAM;   "IO PARAMETERS"
   ADDRESS:SHORTINTEGER;   "ADDRESS OF FILE ON VIRTUAL DISK"

BEGIN "OBTAIN NUMBER OF FREE PAGES"
   IDENTIFY('VIRTUAL DISK SPACE (PAGES) : (:10:) ');
   "DISPLAY MESSAGE"
   DISPLAY(NL);
   PARAM.ARG:=ADDR;   "SET ADDRESS OF FREE FILE"
   PARAM.OPERATION:=INPUT;
   IOTRANSFER(DISKDEVICE,PARAM,FREE);   "GET FREE FILE"
   FIRSTFREEPAGE:=FREE.MISCELLANEOUS(.2.);
   WRITEINT(FIRSTFREEPAGE);   "DISPLAY NUMBER OF FREE PAGES"
END;
* THE MAIN PROGRAM INTERROGATES THE INVOCATION OF *
* THE ROUTINE AND IF CORRECT THEN DISPLAYS THE NUMBER OF *
* FREE PAGES BY CALLING THE INITIALIZE ROUTINE. *
* 
* ***********************************************

BEGIN "GET FREE PAGES"
  IF TASK = JOBTASK THEN
    BEGIN "DISPLAY NUMBER OF FREE PAGES"
      INITIALIZE
    END
  END
END. "END OF DISKSPACE PROGRAM"
APPENDIX E

THE CONCURRENT PASCAL KERNEL
/* Concurrent PASCAL UNIX Kernel */

/* Copyright 1982 Robert Young */

/* Robert Young */
/* Dept of Computer Science */
/* Kansas State University */

/* UNIX File Functions for Solo */
/* interfaces to the UNIX file */
/* system. */
/* Added by Martin Wilde 1984 */

/* UNIX CallUnix function added */
/* to permit Solo application programs */
/* to run UNIX programs. */
/* Added by Martin Wilde 1984 */

/* Program reinitialization and time functions */
/* added to permit the user to access the time */
/* and reinitialize a concurrent program. */
/* Added by Martin Wilde 1984 */

#include <stdio.h>
#include <signal.h>
#include <sgtty.h>
#include <sys/types.h>
#include <stat.h>
#include <sys/timeb.h>

#define GATEASSIST
#define PCBDUMP
/* #define SYMDEBUG */

#define BELL '\07'
#define NL '\n'
#define LF '\n'
#define CR '\r'

#define STDIN 0
#define STDOUT 1

#define currlevl(p) p->pcbldb->ldlevl+1

#define CONLEN 80
#define NDEV 8
#define CURPSLICE 100
#define MAXSLICE 1000

/* error codes */

#define BRKPTERR 0
#define INDXERR 1
/* entry codes */

#define ENTERROER 0
#define ENTTASKQ 1
#define ENTSSTOPQ 2
#define ENTSIPROG 3
#define ENTSIPROC 4
#define ENTSDEPDEV 5
#define ENTSIPROG 6
#define ENTSDELAY 7
#define ENTSDefense 8
#define ENTSIPXCH 9
#define ENTSMON 10
#define ENTSGENTR 11
#define ENTSGETXEXIT 12
#define ENTSIOREQ 13
#define ENTSUTRAC 14
#define ENTSIPXEN 15
#define ENTSIPXEX 16
#define ENTSATMTR 17
#define ENTSSETH 18
#define ENTSSTART 19
#define ENTSSTOP 20
#define ENTSWAIT 21
#define ENTSREALT 22
#define ENTSVC1 23
#define ENTSVC2 24
#define ENTSVCD 25
#define ENTSGMEM 26
#define ENTSBRKPT 27
#define ENTSVCD15 28
#define ENTSUNOPEN 29
#define ENTSUNREAD 30
#define ENTSUNWRITE 31
#define ENTSUNCLOSE 32
#define ENTSUNCREAT 33
#define ENTSUNACCESS 34
#define ENTSUNCALL 35
#define ENTSQUIT 36
#define ENTSLAST 37
/* wait types */

#define NOWAIT 0
#define QVARWAIT 1
#define GATEWAIT 2
#define EXITWAIT 3
#define IOWAIT 4
#define SVCWAIT 5
#define CLOKWAIT 6
#define DEADWAIT 7
#define PRFXWAIT 8
#define SVCPWAIT 9
#define CTRLGWAIT 10

/* wait attributes */

#define EXTWTYP 0x0001
#define PFXWLWTYP 0x0002

/* attribute codes */

#define ATTRCALL 0
#define ATTRHEAP 1
#define ATTRLINE 2
#define ATTRRSLT 3
#define ATTRRTIM 4

/* device codes */

#define TYPEDEV 0
#define DISKDEV 1
#define TAPEDEV 2
#define VARLDEV 3
#define CARLDEV 4
#define PRINTERDEV 5
#define ITAMDEV 6

/* io operation codes */

#define IOOPIN 0
#define IOOPOUT 1
#define IOOPMOVE 2
#define IOOPCTRL 3

/* io result codes */

#define IORSCMPL 0
#define IORSIREQ 1
#define IORSTAT 2
#define IORSPFAIL 3
#define IORSEND 4
#define IORSENDM 5
#define IORSBEGM 6
/* io move op codes */
#define IOMVWF 0
#define IOMVREW 1
#define IOMVUNSP 2
#define IOMVGBKSP 3
#define IOMVUNLD 4

struct ioparam {
    short ioop;
    short iostatus; /* status code */
    int ioarg;
};

/* callunix interface variables */
int (*oldhup)();
int (*oldquit)();
int (*oldintr)();

/* priorities */
#define MMAPRIOR 0
#define IOPRIOR 1
#define LOWPRIOR 2
#define SEQPRIOR 3

struct osfd {
    char fdvol[4];
    char fdfn[8];
    char fdext[3];
    char fdaclt;
};

struct svc1blok {
    char svc1fc, svc1lu, svc1sta, svc1sta2;
    char *svc1sad, *svc1lead;
    int svc1rad, svc1lx, svc1xit;
};

struct svc7blok {
    char svc7cmd, svc7mod, svc7sta, svc7lu;
    short svc7key, svc7lrc;
    struct osfd svc7fd;
    int svc7siz;
};

typedef struct pcbqueue {
    struct pcb *pcbqfrst, *pcbqlast
};

struct as {
char    *msll, *msra, *msql;
struct ld  *msldl;
struct ms  *mspl;
};

struct sd  {
  char    *sdstkbot, *sdstktop;
  char    *sdheapbt, *sdheaptp;
  short   sdseqstk;
};

struct gate {
  short   gatemes;
  short   gatelock;
  struct pcbqueue gatewaitq;
  struct pcb *gateownr;
};

struct ld  {
  int     ldll, ldra, ldql, ldldl;
  char    ldlevl, ldfill1[3];
  int     ldstl;
  struct gate ldgate;
};

struct pcb  {
  struct pcb *pcbqlnk, *pcbqlnk;
  int       pcbstat, pcbpc;
  struct sd  *pcbsdb;
  struct ld  *pcbldlb;
  short      pcberror, pcberrll, pcballr, pcbstlvl;
  int        pcbhpr[16], pcbhprbf[8], pchprbf[16];
  struct ld  pcbld;
  struct ms  *pcbgb;
  struct pcb *pcbchld, *pcbcnchld, *pcbhpar;
  short      pcbcrvlvl, pcbexvlvl, pcbwtyp, pcbpchipd, pcbslice;
  short      pcbiodev, pcbovrtm, pcbnest, pcbprior;
  int        pcbrightm, pcbqarg;
  longq      pcbsstart, pcbsstop;
  short      pcbabork;
};

#define PCBLEN 0x140

char    coninbuf[80], conotbuf[80];

struct dcb  {
  struct pcb *dcbuser;
  char    *dcbargs;
  short    dcbufin, dcbdevtp;
  short    dcbfda, dcbinlen;
  short    dcbinndx, dcbotndx;
char *dcbinbuf, *dcbotbuf;
} dchs[8] = {
    {0,0,80,TYPEDEV,-1,0,0,0,cominbuf,comoutbuf},
    /* std in and out = console */
    {0,0,512,DISKDEV,-1,0,0,0,0,0,0},
    /* dev 1 = user disk */
    {0,0,0,VARLDEV,-1,0,0,0,0,0,0},
    /* dev 2 = tape */
    {0,0,0,VARLDEV,-1,0,0,0,0,0,0},
    /* dev 3 = var len dev */
    {0,0,80,CARDDEV,-1,0,0,0,0,0,0},
    /* dev 4 = card reader */
    {0,0,132,PRINTDEV,-1,0,0,0,0,0,0},
    /* dev 5 = print dev */
    {0,0,0,VARLDEV,-1,0,0,0,0,0,0},
    /* dev 6 = var len dev */
    {0,0,0,VARLDEV,-1,0,0,0,0,0,0}
    /* dev 7 = tape 2 */
};

/* Communications Area */

struct commbloc {
    short entrcode;
    struct pcb *newpcb;
    struct pcb **gparm1,**gparm2;
    struct gate *gateparm;
    struct pcb *pcbparm;
    int intparm1;
    short intparm2;
    struct ioparam *ioparm1;
    char *ioparm2;
    int *intrslt1;
    int *intrslt2;
    int kernr7;
    struct svc1blok *svc1ptr;
    struct svc7blok *svc7ptr;
    struct id *idparm;
    int svc15ptr;
    int segmap[16];
} comm ;

struct pcb *inipcb;
struct pcb *curpcb;

extern int c_main;

/* Kernel Local Variables */

struct pcbqueue readymon, readyio, readylow, clockevent,
    reinit;
int now, period, timeparm;
short kernhalt;
struct timeb *tp;
int extwaits;
int nextpcbid;
int entrncts[ENTLAST];
int reload;
char *mem;
int memsize;
short lufd"32"; /* maps lu to unix fd */

int waittab[] = {
0, /* no wait */
PPXRELWWTYP, /* qvar */
0, /* gate */
0, /* exit */
PPXRELWWTYP+EXTWTYP, /* io */
PPXRELWWTYP+EXTWTYP, /* svc1 */
PPXRELWWTYP+EXTWTYP, /* clock */
PPXRELWWTYP, /* dead */
PPXRELWWTYP, /* prefix */
PPXRELWWTYP+EXTWTYP, /* svc 15 */
PPXRELWWTYP+EXTWTYP, /* control G */
0}; /* end of table */

/* Dump PCB */

#define PCBDUMP

dumppcb(p)
register struct pcb *p; 
{
    static char wstat[] =
        "run qvargateexitio svc1clokdeadprfxsvcf";
    fprintf(stderr,"%6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X %6X ",
p,&wstat[4*p->pchwttyp],p->pcbpc,0 /* currlevl */, p->pcbcrlvl,p->pcbstlvl,p->pcbexlvl);

dumpccbs(p)
register struct pcb *p; 
{
    register struct pcb *q;
    dumppcb(p);
    for (q=p->pcbfchld;q;q=q->pcbncchild)
        dumpccbs(q);
}
dumpcbtree ()
{
    fprintf(stderr, "-n");
    fprintf(stderr, " pcb stat pc lv cr st ex fchild nchild parent er ");
    fprintf(stderr,"line gb lb -n");
    dumpcbins(initpcb);
}
#endif
abend(reason)
{
    register char *p;
    fprintf(stderr,"kernel abend code %d-%n",reason);
    p = (char *) 0xffffffff;
    *p = reason;
};
errhalt()
{
    abend(1);
}
kernerr(sig,type,line)
int sig,type,line;
{
    short hw1, hw2;
    char *msg;
    static char *msgtab[] = {
        "Breakpoint",
        "Index Range Error",
        "Parameter Range Error",
        "Value Range Error",
        "Case Label Error",
        "Trunc Range Error",
        "Variant Tag Error",
        "Pointer Error",
        "Stack Overflow Error",
        "Heap Overflow Error",
        "10", "11", "12", "13", "14", "15",
        "Illegal Instruction",
        "Address Fault",
        "Arithmetic Fault",
        "Delay Error"};
    static char *siqtab[] = {
        "Signal 0",
        "Hangup",
        "Interrupt",
        "Quit",
        "Illegal Instr",
        "Trace Trap",
        "Abort Call",
        "
"EMT Trap",
"Float Exception",
"Kill",
"Memory Fault",
"Segmentation Fault",
"System Call Error",
"Pipe Error",
"Alarm Irpt",
"Termination",
"Signal 16";

line=0;
msg = sigtab[sig];
if (sig==4) {
    hw1 = *(short int *)curpcb->pcbp;
    hw2 = *(short int *)(curpcb->pcbp+2);
    if ((hw1 & 0xfff00)==0x8800) {
        line=hw2;
        msg = msgtab[(hw1 & 0x00f0) >> 4];
    }
}
else if (sig==0) {
    msg = msgtab[type];
};

fprintf(stderr,"Line %D Address %6X %s\n",line,
curpcb->pcbp,
msg);
#elif define SYMDEBUG
    condebug(initpcb,curpcb);
#else
    errhalt();
#endif
    if (type!=BRKPTERR)
        stopproc(curpcb,currlevl(curpcb)-1);
};

kerndead ()
{
    fprintf(stderr,"Infinite Wait Error\n");
#if define SYMDEBUG
    condebug(initpcb,curpcb);
#else
    #ifdef PCBDUMP
        dumppcb();
    #endif
        errhalt();
    #endif;
        extwaits++;
};

addq(q,p)
register struct pcbqueue *q;
register struct pcb *p;
{
    if (q->pcbqlast) q->pcbqlast->pcbfqlnk=p;
    else q->pcbfqfrst=p;
    p->pcbbqlnk=q->pcbqlast; p->pcbfqlnk=0;
    q->pcbqlast=p;
};

struct pcb *getq(q)
register struct pcbqueue *q;
{
    register struct pcb *p;
p=q->pcbfqfrst;
    if (p) {
        q->pcbfqfrst=p->pcbfqlnk;
        if (q->pcbfqfrst)
            q->pcbfqfrst->pcbbqlnk=0;
        else q->pcbqlast=0;
        p->pcbfqlnk=0;
    }
    return(p);
};

struct pcb *removeq(q, arg)
register struct pcbqueue *q;
int arg;
{
    register struct pcb *p, *lp;
    lp=0; p=q->pcbfqfrst;
    while (p)
        if (p->pcbqarg==arg) break;
        else {
            lp=p; p=p->pcbfqlnk;
        };
    if (p) {
        if (lp) lp->pcbfqlnk=p->pcbfqlnk;
        else q->pcbfqfrst=p->pcbfqlnk;
        if (q->pcbqlast==p) q->pcbqlast=lp;
    }
    return(p);
};

initq(q)
register struct pcbqueue *q;
{
    q->pcbfqfrst=0; q->pcbqlast=0;
};

runupdsl()
{
    curpcb->pcbslice+=CURRSICE;
    if (curpcb->pcbslice>MAXSLICE) {
        while(curpcb->pcbslice>=MAXSLICE) {
            curpcb->pcbslice-=MAXSLICE;
        }
    }
curpcb->pcbvtm=1;
if (!curpcb->pcbnest)
    curpcb->pcbprior=LOWPRIOR;
}

runpreempt()
{
    runudpdl();
    curpcb=0;
}

rdyenter(p)
register struct pcb *p;
{
    switch (p->pcbprior) {
    case MOMPRIOR: addq(&readymon, p);
        break;
    case IOPRIOR: addq(&readyio, p);
        break;
    case LOWPRIOR: addq(&readylow, p);
        break;
    }
    p->pcbwtyp=NOWAIT;
}

rdysselect()
{
    register struct pcb *p;
    if (curpcb) abend(1);
    for (;;)
    {
        if (!((p=getq(&readymon))
        if (!((p=getq(&readyio))
            p=getq(&readylow);
        if (p) {
            if (p->pcbexlvl)
                if (currlevl(p)>p->pcbexlvl)
                    chkpterm(p);
            else break;
        } else break;
    }
    curpcb=p;
    comm.newpcb=curpcb;
}

rdyinit()
{
    initq(&readymon);
    initq(&readyio);
    initq(&readylow);
    initq(&reinit);
rdyresched()
{
    register struct pcb *p = curpcb;
    register int resched;
    resched=0;
    if (curpcb) {
        runanwait();
        if (p->pcbprivor>MONPRIOR)
            if ((readymon.pcbqfrst) p->pchovtm)
                resched++;
        else if (p->pcbprivor>IPROR)
            if (readyio.pcbqfrst) resched++;
        else resched=0;
    }

    else resched=0;
    if (resched) {
        p->pchovtm=0;
        rdyenter(p);
        curpcb=0;
    }
};

int acgrpf(p,lcp)
register struct pcb *p;
register struct ld *lcp;
{
    if ((++(lcp->ldgate.gatesem)) & lcp->ldgate.gatelock) {
        addq(&lcp->ldgate.gatewaitq,p);
        return(0);
    }

    else {
        lcp->ldgate.gatelock=1;
        if (lcp->ldgate.gateownr) abend(999);
        lcp->ldgate.gateownr=p;
        return(1);
    }
};

replfx(ldp, p)
register struct pcb *p;
register struct ld *ldp;
{
    register struct pcb *q;
    register int ok;
    if (ldp->ldgate.gateownr!=p) abend(998);
    ldp->ldgate.gateownr=0;
    ldp->ldgate.gatesem--;
    if (ldp->ldgate.gatewaitq.pcbqfrst) {
        ok=0;
        while (ldp->ldgate.gatewaitq.pcbqfrst & ok) {

    

            

        }

    }
q=getq(&ldp->ldgate.gatewaitq);
if (q->pcbwttyp != PRFXWAIT) abend(997);
ldp->ldgate.gatesem--; 
ok=acqrfps(q, ldp);
if (ok) {
    ldp->ldgate.gatesem++; 
    ldp->ldgate.gateowner=q;
    q->pcbwttyp=NOWAIT; 
    rdyenter(q);
} /* else waiting for another gate */ ;
}
if ( !ok & & (ldp->ldgate.gatesem>=0)) 
    ldp->ldgate.gatelock=0;
else if (ldp->ldgate.gatesem>=0) 
    ldp->ldgate.gatelock=0;
};

int acqrfps(p, skipldp)
register struct pcb *p;
struct ld *skipldp;
{
    register struct ms *q, *r;
    q=p->pcbpb;
    while (q) {
        if (q->msldl != skipldp)
            if ( !acqrfp(p, q->msldl))
                break;
        q=q->mspl;
    }
    if (q)
        r=p->pcbpb;
        while (q!=r) {
            if (r->msldl!=skipldp)
                relpfx(r->msldl,p);
            r=r->mspl;
        }
    return (0);
};
    return (1);
};

relpfxs(p, skipldp)
struct pcb *p;
struct ld *skipldp;
{
    register struct ms *q;
    q=p->pcbpb;
    while(q) {
        if (q->msldl != skipldp)
            relpfx(q->msldl,p);
        q=q->mspl;
    };
};
remwwait (p)
register struct pcb *p;
{
    int wt;
    if (p->pcbwaityp == NOWAIT) abend(996);
    wt=waittab[ p->pcbwaityp ];
    if (wt & EXTWTYP)
        extwaits--;
    if (wt & PFXRELWTYP)
        if (!acqrpfs(p,(struct ld *)0)) {
            p->pcbwaityp=PRFWAIT;
            return;
        }
    if (p->pcbwaityp == CTRLGWAIT) {
        curpcb=p;
        comm.newpcb=curpcb;
        p->pcbwaityp=NOWAIT;
    }
    else {
        p->pcbwaityp=NOWAIT;
        rdyenter(p);
    }
};

entrwwait(wait)
int wait;
{
    int wt;
    if (curpcb->pcbwaityp != NOWAIT) abend(995);
    curpcb->pcbwaityp = wait;
    wt = waittab[wait ];
    if (wt & EXTWTYP)
        extwaits++;
    if (wt & PFXRELWTYP)
        relpfxs(curpcb,(struct ld *)0);
    runpreempt();
};

chkpterm (p)
register struct pcb *p;
{
    register struct pcb *q;
    if ((p->pcbexlv1 > 0 ) & (currlv1(p)!=p->pcbexlv1)) {
        q=p->pcbfchld;
        while (q) {
            if (q->pcbclvl >= p->pcbexlv1) break;
            q=q->pcbnchld;
        }
    }
    if (!q) {
        if(!((p->pcbwaityp == NOWAIT)
(p->pcbwttyp == EXITWAIT))
    abend(990);
    if (p->pcbwttyp == NOWAIT) {
        p->pcbwttyp = EXITWAIT;
        relpfxs(p, (struct ld *) 0);
    }
    if (p->pcbexlvl == p->pcbstlvl+1)
        if (!poplevels(p, p->pcbstlvl)) abend(989);
    if ((p==initpcb) & (p->pcbexlvl==1))
        kernhalt=1;
    p->pcbexlvl=0; p->pcbstlvl=255;
    remwvait(p);
}
else p->pcbwttyp = EXITWAIT;
}
else remwvait(p);
;

stoproc(p, level)
register struct pcb *p;
int level;
{
    if (p==curpcb) {
        runpreempt(); rdyenter(p);
    }
    if (p->pcbstlvl >= level) {
        p->pcbstlvl = level;
        if (((p->pcbexlvl==0) (p->pcbexlvl > p->pcbstlvl))
            p->pcbexlvl = p->pcbstlvl + 1;
        if ((p->pcbwttyp == QVARWAIT) & (currlevl(p) >= p->pcbexlvl))
            remwvait(p);
        if ((p->pcbwttyp == CLOKWAIT) & (currlevl(p) >= p->pcbexlvl)) {
            if (p!=removeq(&clockevent, (int) p)) abend(985);
            remwvait(p);
        }
    }
};

stopfam(p,level)
register struct pcb *p;
int level;
{
    register struct pcb *q;
    for (q=p->pcbfchld; q; q=q->pcbrchld)
        stopfam(q, level);
    stoproc(p, level);
}

/* I/O routines go here */
dataio(buf, ioparm, dcbp)
char *buf;
register struct ioparam *ioparm;
register struct dcb *dcbp;
{
  int len, n;
  if (dcbp->dcbfd < 0) {
      fprintf(stderr, "Device %d is not assigned-n",
              dcbp->dcbns);
      abend(540);
  }
  if (dcbp->dcbdevtp == DISKDEV)
      lseek(dcbp->dcbfd, ioparm->ioarg*dcbp->dcbbufln, 0);
  if (dcbp->dcbdevtp == VARLDEV) n = ioparm->ioarg;
  else n = dcbp->dcbbufln;
  if (ioparm->ioop == IOOPIN) {
      len = dcbp->dcbinlen = read(dcbp->dcbfd, buf, n);
  } else if (ioparm->ioop == IOOPCTRL)"
      reboot(); "
  else
      len = write(dcbp->dcbfd, buf, n);
  if (len<0) abend(541);
  else if (len==0) ioparm->iostatus = IORSEND;
  else ioparm->iostatus = IORSCLP;
  if (dcbp->dcbdevtp == VARLDEV)
      ioparm->ioarg = len;
};

consoleio(buf, iopq, dcbp)
register char *buf;
register struct ioparam *iopq;
register struct dcb *dcbp;
{
  if (iopq->ioop == IOOPIN) {
      if (dcbp->dcbinndx >= dcbp->dcbinlen) {
          dcbp->dcbinlen = read(STDIN, dcbp->dcbinbuf,
                                 dcbp->dcbbufln);
          if (dcbp->dcbinlen < 0) abend(501);
          dcbp->dcbinndx=0;
      }
      *buf = dcbp->dcbinbuf[dcbp->dcbinndx++];
      if (*buf == BELL) "
          sigasend(); "
  }
  else if (iopq->ioop == IOOPOUT) {
      dcbp->dcbotbuf[dcbp->dcbotndx++]= *buf;
      if ((*buf == 'n')
          (dcbp->dcbotndx>=dcbp->dcbbufln)) "
          if (write(STDOUT, dcbp->dcbotbuf,
                      dcbp->dcbotndx) !=
              dcbp->dcbotndx) abend(502);
          dcbp->dcbotndx=0;
else if (ioarg->iioo == IOPCCTRL) {
    sigalas();
}
else {
    fprintf(stderr,"Invalid operation to console-n");
    abend(503);
};

tapeio(buf,ioparm,dcbp)
register char *buf;
register struct ioparm *ioparm;
register struct dcb *dcbp;
{
    struct {
        int command;
        int pad[2];
    } ctlparm;

    static char cmdmap[] = {3,7,0,1,7};

    switch (ioparm->iioo) {
    case IOPPIN:
    case IOPPOUT:
        dataio(buf,ioparm,dcbp);
        break;
    case IOPMOVE:
        ctlparm.pad[0]=0;
        ctlparm.pad[1]=0;
        ctlparm.command=cmdmap[ioparm->ioarg];
        if (dcbp->dcbfd < 0) {
            fprintf(stderr,"Device %d not assigned-n",
                dcbp-dcbps);
            abend(542);
        }
        if (ioctl(dcbp->dcbfd,TIOCSETP,&ctlparm)) {
            fprintf(stderr,"Error on IOCTL to device %d-n",
                dcbp-dcbps);
            abend(543);
        }
        ioparm->iostatus = IORSCMPI;
        break;
    }
};

ioreq(buf,ioarg,dev)
register char *buf;
register struct ioparm *ioarg;
int dev;
{ register struct dcb *dcbp;
    if (dev>=NDEV) {
        fprintf(stderr,"Invalid device %d\n",dev);
        abend(520);
    }
    dcbp = &dchs[dev];
    if (curpcb->pcbnest == 0) curpcb->pcbprior=IOPPRIOR;
    ioarg->iostatus = IORSCHMPL;
    switch (dcbp->dcbdevtp) {
        case TYPEDEV:
            consoleio(buf,ioarg,dcbp);
            break;
        case DISKDEV:
        case CARDDEV:
        case PRITDEV:
            dataio(buf,ioarg,dcbp);
            break;
        case TAPEDEV:
        case VARLDEV:
            tapeio(buf,ioarg,dcbp);
            break;
        default:
            fprintf(stderr,"Invalid device %d\n",dev);
            abend(521);
    }
}

sigawait ()
{
    curpcb->pcbqarg=(int)curpcb;
    addq(&clockevent,curpcb);
    entrwait(CLOKWAIT);
}

/* The sigaload function places the calling process on the reinit queue until the control signal is sent. */
sigaload ()
{
    curpcb->pcbqarg=(int)curpcb;
    addq(&reinit,curpcb);
    entrwait(CTRLGWAIT);
}

/* The sigasend function removes the delayed process from the reinit queue. The process is then executed. */
sigasend ()
{
    register struct pcb *p
    p=getq(&reinit);
    remwwait(p);
}
sigsend(q)
register struct pcbqueue *q;
{
    register struct pcb *p;
    register int resched = 0;
    p = getq(q);
    while (p) {
        remvwait(p);
        resched++;
        p = getq(q);
    }
    if (resched) rdyresched();
}

clockinit()
{
    initq(&clockevent);
    period = 0;
}

timeinit()
{
    ftime(tp);
    now = tp->time;
}

#ifndef GATEASSIST
gateenter(q)
register struct gate *q;
{
    curpcb->pchnest++;
    curpcb->pcbprior = HIGHPRIOR;
    if (q->gatelo) {
        addq(&g->gatewaitg, curpcb);
        entrwait(GATEWAIT);
    }
    else g->gatelo = 1;
}
gateexit(q)
register struct gate *q;
{
    if (! --curpcb->pchnest)
    curpcb->pcbprior = LOWPRIOR;
    if (g->gatewaitg.pcbqfrst)
        remvwait(getq(&g->gatewaitg));
    else g->gatelo = 0;
}
#endif

#else GATEASSIST

#endif
gateentr(g)
register struct gate *g;
{
    if ((g->gatesem>0) & g->gatelo)
        addq(&g->gateloq,curpcb);
        entrwait(GATEWAIT);
    else g->gatelo=1;
}

gateexit(g)
register struct gate *q;
{
    register struct pcb *p;
    curpcb->pcbprev = LOWPRIOR;
    if (g->gateloq.pcbqfrst) {
        p=getq(&g->gateloq);
        p->prio=MONPRIOR;
        remvwait(p);
    }
    else if (g->gatesem>=0)
        g->gatelo=0;
    rdyresched();
}

gatechq(q1,q2,q)
register struct pcb **q1, **q2;
register struct gate *q;
{
    register struct pcb *p;
    *q2 = 0;
    if (*q1) kernerr(0, DELAYERR, 0);
    else {
        *q1 = curpcb;
        entrwait(QVARWAIT);
    }
    if (p) {
        p->prio=MONPRIOR;
        remvwait(p);
        rdyresched();
    }
    else {
        g->gatesem--;
        if (!g->gateloq.pcbqfrst) {
            #ifdef GATEASSIST
                if (g->gatesem>=0)
                    g->gatelo=0;
            #else
                g->gatelo=0;
            #endif
            else remvwait(getq(&g->gateloq));
        }
    }
}
gateinit(q)
register struct gate *q;
{
    initq(&q->gatewaitq);
    q->gateelock=1; q->gatesem=0;
}

entrprfx(p,ldp)
register struct pcb *p;
register struct ld *ldp;
{
    ldp->ldgate.gatesem--;
    if (! acqrf(p,ldp)) {
        relpfxs(ldp);
        /* skip one we didn't get */
        p->pcbwttyp=PRFXWAIT;
        runpreempt();
    }
}

exitprfx(p,ldp)
register struct pcb *p;
register struct ld *ldp;
{
    ldp->ldgate.gatesem++;
    ldp->ldgate.gateownr=p;
    relpfx(ldp,p);
}

initproc(p)
register struct pcb *p;
{
    register struct pcb *q, *lq;
    p->pcbruntm=0; p->pcbslice=0;
    p->pcbstart=0;
    p->pcbnest=0; p->pcbovrtm=0;
    p->pcbprior=LOWPRIOR; p->pcbabort=0;
    p->pcbparnt=curpcb; p->pcbfcchild=(struct pcb *)0;
    p->pcbnchild=(struct pcb *)0;
    if (curpcb) {
        lq=(struct pcb *)0; q=curpcb->pcbfcchild;
        while (q) {
            lq=q; q=q->pcbnchild;
        }
        if (lq) lq->pcbnchild=p;
        else p->pcbparnt->pcbfcchild=p;
    }
    p->pbcrlvl=currlvl(p);
    p->pcbwttyp=NOWAIT;
    p->pcbsstlvl=255;
    p->pbcexlvl=0;
p->pcbpchid=nextpcbid++;  
ryenter(p);  
ryresched();  

}  
initprg(p)  
register struct pcb *p;  
{  
p->pcbruntm=0; p->pcbslice=0;  
p->pcbstart=0;  
p->pcbnest=0; p->pchevrtm=0;  
p->pcbprior=LOWPRIOR; p->pcbabort=0;  
}  
endproc()  
{  
register struct pcb *p, *lp;  
if (curpcb->pcbparnt) {  
lp=(struct pcb *)0;  
p=curpcb->pcbparnt->pcbfchld;  
while (p!=curpcb) {  
lp=p; p=p->pcbnchld;  
}  
if (lp) lp->pcbnchld=p->pcbnchld;  
else curpcb->pcbparnt->pcbfchld=p->pcbnchld;  
if ((curpcb->pcbparnt->pcbexvlvl > 0) &
(curpcb->pcbparnt->pcbwttyp == EXITWAIT))  
remvwait(curpcb->pcbparnt);  
}  
if (curpcb->pcbfchld) abend(984);  
curpcb->pcbwttyp=DEADWAIT;  
if (curpcb==initpcb) kernhalt=1;  
runtimept();  
}  
endprog()  
{  
register struct pcb *p;  
p=curpcb;  
entrwai(EXITWAIT);  
p->pcbexvlvl=currlevl(p);  
chkpterm(p);  
}  
stopprg(p)  
register struct pcb *p;  
{  
if (p!=curpcb) abend(123);  
stopfam(comm.pcbparm,currlevl(curpcb));  
}  
pfxstopint()  
{  
register struct pcb *p;  
p=curpcb;  
}
entrwait(EXITWAIT);
chkpterm(p);
}

rsetproq()
{
curpcb->pcbctlvl=255;
curpcb->pcbexcvl=0;
curpcb->pcbberrln=0;
curpcb->pcberrcd=0;
}
gatedelay(q,q)
register struct pcb **q;
register struct gate *q;
{
    if (*q) kernerr(0,DELAYERR,0);
    else {
        *q = curpcb;
        entrwait(QVARWAIT);
    }
    g->gatesem--;
    if (!g->gatewaitq.pcbqfrst)
#ifdef GATEASSIST
        if (g->gatesem>=0) g->gatelnk=0;
        else ;
#else
        g->gatelnk=0;
#endif
    else
        rnvwait(getq(&g->gatewaitq));
}
gatecont (q,q)
register struct pcb **q;
register struct gate *g;
{
    g->gatesem--;
    if (!*q)
        gateexit(g);
    else {
#ifdef GATEASSIST
        (*q)->pcbprior=MOWPRIOR;
#endif
        rnvwait(*q);
        g->gatesem++;
        *q = (struct pcb *) 0;
#ifdef GATEASSIST
        curpcb->pcbprior=LOWPRIOR;
#else
        if (!--curpcb->pcbnest)
            curpcb->pcbprior = LOWPRIOR;
#endif
        rdyresched();
}
varinit()
{
    #ifdef ENTRYCOUNTS
    register int i;
    #endif
    initq(&clockevent);
    kernhalt=0;
    extwaits=0;
    nextpcbid=1;
    #ifdef ENTRYCOUNTS
    for (i=0; i<=ENTRLAST; i++)
        entrcounts[i]=0;
    #endif
}

attribute(parm)
register int parm;
{
    switch (parm) {
        case ATTRCALL: return((int) curpcb);
        case ATTRHEAP: return((int) marf());
        case ATTRLINE: return(curpcb->pcberrln);
        case ATTRSLT: return(curpcb->pcberrcd);
        case ATTRRTIM: return(curpcb->pchruntm);
        default: return(0);
    }
}

/* OS/32MT SVC 1 and 7 emulation */
reposn(fd,statptr,lseek2,lseek3,ctlarq)
int fd,lseek2,lseek3,ctlarq;
register struct stat *statptr;
{
    if ((statptr->st_mode & S_IFMT)==S_IFREG) {
        /* normal file */
        if (lseek(fd,lseek2,lseek3)<0)
            abend(460);
    }
    else { /* try at ioctl */
        if(ioctl(fd,ctlarq,0)<0)
            abend(461);
    }
}

tsve1(p)
register struct svc1blok *p;
{
    short fd;
    int len;
    struct stat statbuf;
fd = lufd[p->svc1lu];
if (p->svc1lu==0) fd=1;
if (fd<0) {
    fprintf(stderr,"invalid lu %d-%n",p->svc1lu);
    abend(440);
};
if (p->svc1fc & 0x80) { /* command function */
    fstat(fd,&statbuf);
    p->svc1sta=0x00;
    switch (p->svc1fc) {
    case 0xc0:
        /* rewind */
        reposn(fd,&statbuf,0,0,0);
        break;
    case 0xa0:
        /* bsr */
        reposn(fd,&statbuf,512,1,0);
        break;
    case 0x90:
        /* fsr */
        reposn(fd,&statbuf,1,0,0);
        break;
    case 0x88:
        /* wfm */
        reposn(fd,&statbuf,0,0,0);
        break;
    case 0x84:
        /* fsf */
        reposn(fd,&statbuf,2,0,0);
        break;
    case 0x82:
        /* hsf */
        reposn(fd,&statbuf,0,0,0);
        break;
    default:
        fprintf(stderr,"illegal svc1 func-%n");
        abend(443);
    }
}
else { /* data transfer */
    switch(p->svc1fc & 0x60) {
    case 0x00:
        /* wait */
        break;
    case 0x60:
        /* test and set */
        abend(443);
    default:
        if (p->svc1fc & 0x04)
        if (lseek(fd,512*p->svc1rad,0)==-1)
            abend(444);
        len = p->svc1ead - p->svc1sad + 1;
        if ((p->svc1lu==0) & (p->svc1fc&0x10)) fd=0;
        if (p->svc1fc & 0x20)
            len=write(fd,p->svc1sad,len);
        else len=read(fd,p->svc1sad,len);
        if ((p->svc1xf<len)<0)
            p->svc1sta=0x84;
        else if (len==0)
            p->svc1sta=0x88;
        else p->svc1sta=0x00;
break;
};

convfd (in, out)
register struct osfd *in;
register char *out;
{
    register char *p;
    register int i;

    p = out;
    if ((in->fdacl == 's') ? (in->fdact = 'S')) {
        strcpy(p, "/usr/pas32/sysfiles/");
        p += strlen(p);
    }
    for (i = 0; i <= 7; i++)
        if ((in->fdfn[i] == ' ') ? (p++) = in->fdfn[i];
    for (i = 0; i <= 2; i++)
        if ((in->fdext[i] == ' ') ? (p++) = in->fdext[i];
    * (p++) = 0;
    fprintf(stderr, "in=%16.16s out=%s-n", in, out);
};

svc7(p)
register struct svc7blok *p;
{
    int fd, lu, cmd;
    char fn[32];
    short mode, append;
    struct stat statbuf;

    if (p->svc7cmd & 0xd2) convfd(&p->svc7fd, fn);
    lu = p->svc7lu; p->svc7sta = 0;
    cmd = p->svc7cmd;
    if (cmd) {
        if (cmd & 0x80) { /* alloc */
            fd = creat(fn, 0664);
            if (fd == -1) {
                fprintf(stderr, "svc7 alloc error on %s-n", fn);
                p->svc7sta = 5; return;
            }
        }
        if (close(fd) == -1) abend(472);
    }
    if (cmd & 0x40) { /* assign */
        if (lufd[lu] == -1) {
            p->svc7sta = 2; return;
        }
        switch (p->svc7mod & 0xe0) {
            case 0x00: /* sro */
case 0x20:  /* ero */
    mode=0; append=0; break;
case 0x40:  /* swo */
case 0x60:  /* ewo */
    mode=1; append=1; break;
case 0x80:  /* srw */
case 0xa0:  /* srsw */
case 0xc0:  /* ersw */
case 0xe0:  /* erw */
    mode=2; append=0; break;
};
if ((fd=open(fn,mode))== -1) {
    fprintf(stderr,"svc7 open error on "%s", fn);
    p->svc7sta=4;
    return;
};
if (append) lseek(fd,0,2);
lufd[lu]=fd;
cmd &= 0x40;
}
if (cmd & 0x20) {  /* chap */
    abend(460);
};
if (cmd & 0x10) {  /* rename */
    abend(481);
};
if (cmd & 0x08) {  /* reprot */
    abend(482);
};
if (cmd & 0x04) {  /* close */
    if (lufd[lu]== -1) {
        p->svc7sta=2;
        return;
    }
    close(lufd[lu]);
lufd[lu] = -1;
cmd &= 0x04;
}
if (cmd & 0x02) {  /* delete */
    if (unlink(fn)) {
        fprintf(stderr,"svc7 delete error on "%s", fn);
        p->svc7sta=5;
        return;
    }
    cmd &= 0x02;
}
if (cmd & 0x01) {  /* checkpoint */
cmd &= 0x01;
}
else {  /* fetch attributes */
    if (lufd[lu]== -1) {

p->svc7sta=2;
    return;
};
    
    fstat(lufd[lu],&statbuf);
p->svc7key=0; p->svc7mod=2;
p->svc7siz=statbuf.st_size;
p->svc71rc=512; p->svc7key=0x7f7e;
};
    return;
};

/*VARARGS1*/
parmerr(msg, argv, arg2, arg3, arg4)
    *msg;
    int argv, arg2, arg3, arg4;
{
    fprintf(stderr,"Error processing options - ");
    fprintf(stderr, msg, argv, arg2, arg3, arg4);
    fprintf(stderr,"-n");
    abend(123);
    exit(999); _exit(999);
}

initmem()
{
    register struct sd *sdp;
    register int *m;
    register char *mem2;

    mem=sbrk(memsze);
    if(!mem) parmerr("insufficient memory available");
    initpcb=(struct pcb *) mem;
    mem2=mem+PCBLEN;
    sdp=(struct sd *) mem2;
    sdp->sdstkbot = mem2;
    sdp->sdstktop = (sdp->sdstkbot+sizeof (*sdp) + 7) & ~8;
    sdp->sdheapb = mem + memsize - 4;
    sdp->sdheappt = sdp->sdheapb;
    sdp->sdseqstk = 1;
    for (m= ((int *) sdp->sdstktop)+1;
        m < ((int *) sdp->sdheapb; m++)
        *m = 0;
    *m = m;
    initpcb->pcbldb = &initpcb->pcbls;
    initpcb->pcbldb->ldlevel = 0;
    initpcb->pcbpb = 0;
    initpcb->pcbstlv = 255;
    initpcb->pcbsdb = sdp;
    initpcb->pcbpc = c_main;
    initpcb->pcbqprim[2] = (int) sdp->sdstktop - 16;
    initpcb->pcballr = 0;
};
/* The reboot function reinitializes the kernel and variables. Once the kernel is reinitialized, the concurrent program is restarted from the initial process. */

reboot()
{
    free(mem);
kerninit(&comm);
curpcb=0;
rdyinit();
clockinit();
timeinit();
varinit();
initmem();
initproc(initpcb);
};

/* The quitprog function terminates the program. */
quitprog()
{
    kernhalt=1;
}

/* The unixcall function places the user in the unix shell. The process performs a fork and assigns the shell execution to this new child process. Once a command is executed by the shell, the user is returned to the concurrent pascal environment. */
unixcall()
{
    register (*savint), pid, rpid;
    int retcode;
    putchar("$"iane__)
    putchar(" ");
    if ((pid = fork()) == 0) {
        signal(SIGQUIT, oldquit);
        execl("/bin/sh", "sh", "-t", 0);
        exit(0100);
    }
    savint = signal(SIGINT, SIG_IGN);
    while ((rpid = wait(&retcode)) != pid && rpid != -1) {
        signal(SIGINT, savint);
    }

    /* The time function provides the kernel time. This function is utilized by the routines which call the realtime function. */

timeproc(sec)
    register int sec;
{ 
    ftime(tp);
    sec = tp->time - now;
    return(sec);
}

starttime()
{
    ftime(tp);
    curpcb->pcbstart=tp->time;
}

stoptime()
{
    int runtime;
    ftime(tp);
    curpcb->pcbstop=tp->time;
    runtime=curpcb->pcbstop - curpcb->pcbstart;
    curpcb->pcbruntm=curpcb->pcbruntm + runtime;
}

scanspars(argc,argv)
int argc;
register char **argv;
{
    register char *p;
    char *fnptr;
    char tempfn[16];
    int fd;
    char ftype;
    short fnum;
    int i;

    mmsize=32*1024;
    for (i=0; i<32; i++)
        lufd[i] = -1;
    argv++; argc--;
    while (argc) {
        if((*argv)[0] == '－')
            switch((*argv)[1]) {
                case 'k':
                    mmsize=0;
                    for (p = &(*(argv)[2]); *p; p++)
                        if((*p >= '0') && (*p <= '9'))
                            mmsize = 10*mmsize + (*p-'0');
                        else if ((*p == 'k'))
                            mmsize *= 1024;
                        else parmerr("invalid -k syntax");
                    break;
                case 'f':
                    fnum = (*argv)[2] - '0';
                    if ((fnum<0) || (fnum>9))
                        parmerr("invalid file number");
                    ftype = (*argv)[3];
            } }
if(!ftype) ftype='w';
switch (ftype) {
    case 'r':
    case 'w':
    case 't':
    case 'x':
        break;
    default:
        parmerr("invalid file type");
};
if (ftype == 't') {
    strcpy(tempfr, "/tmp/cpasaXXXXX");
    mktemp(tempfr);
    fnptr = tempfn;
} else {
    fnptr = argv[1];
    argv++; argc--;
};
if (ftype == 'x') {
    if((fd=open(fnptr,2))<0)
        parmerr("error opening %s",fnptr);
} else
    if (ftype != 'r')
        if ((fd=creat(fnptr,0664))<0)
            parmerr("error opening %s",tempfn);
        close(fd);
    if ((fd=open(fnptr,2))<0)
        parmerr("error opening %s",fnptr);
} else
    if ((fd=open(fnptr,0))<0)
        parmerr("error opening %s",fnptr);
    if (ftype == 't') unlink(fnptr);
    dcbs[fnnum].dcbfd = fd;
    lufd[fnnum] = fd;
    break;
default:
    parmerr("invalid option code");
} else parmerr("invalid parameter format");
argv++; argc--

};

main(argc,argv)
int argc;
char  **argv;
{
    oldquit = signal(SIGQUIT,SIG_IGN);
    kerninit(&comm);
    curpcb=0;
reload=0;
rdyinit();
clockinit();
timeinit();
varinit();
scanspars(argc, argv);
initmem();
initproc(initpcb);
while (!kernhalt) {
    if (!curpcb) rdyselect();
    starttime();
    sigsend(&clockevent);
    if (kernhalt) break;
    if ((!curpcb) & 6 and textwaits) kerndead();
    serve(curpcb);
    stoptime();
    entrncts[comm.entrcode]++;
    switch (comm.entrcode) {
    case ENTERRO:
        kernerr(comm.intparm1, 0, 0); break;
    case ENTASKQ:
        abend (333);
    case ENSTOPQ:
        pfxstopi(); break;
    case ENTPROG:
        break;
    case ENTPROC:
        initproc(comm.pcbparm); break;
    case ENPROG:
        endproc(); break;
    case ENDELAY:
        gatedelay(comm.qparm1, comm.gateparm); break;
    case ENCONT:
        gatecont(comm.qparm1, comm.gateparm); break;
    case ENEXIT:
        gateexit(comm.qparm1, comm.qgateparm); break;
    case ENREQ:
        ioreq(comm.ioparm1, comm.ioparm2, comm.intparm1); break;
    case ENTURAC:
        break;
    case ENTPFXEN:
        entprfx(curpcb, comm.ldparm); break;
case ENTFPREFIX:
    exitprfx(curpbh, comm.ldsparm); break;

case ENTFATTR:
    *comm.intsLt1.attribute = comm.intparm2; break;

case ENTFSETH:
    release(comm.intparm1); break;

case ENTFSTART:
    rsetproq(); break;

case ENTFSTOP:
    stoppgr(comm.pchparm); break;

case ENTFWAIT:
    sigwait(); break;

case ENTFREALT:
    *comm.intsLt1 = timeproc(timeparm); break;

case ENTFVSC1:
    svc1(comm.svc1ptr); break;

case ENTFVSC2:
    abend(222); break;

case ENTFVSC7:
    svc7(comm.svc7ptr); break;

case ENTFMEM:
    abend(688); break;

case ENTFBRKPT:
    kerrerr(0, BRKPTERR, comm.intparm1); break;

case ENTFVSC15:
    abend(555);

case ENTFNOPEN:
    *comm.intsLt1 = open(comm.ioparm2, comm.intparm2); break;

case ENTFNREAD:
    *comm.intsLt1 = read(comm.intparm1, comm.ioparm2, comm.intparm2); break;

case ENTFNWRITE:
    *comm.intsLt1 = write(comm.intparm1, comm.ioparm2, comm.intparm2); break;

case ENTFNCLOSE:
    *comm.intsLt1 = close(comm.intparm1); break;

case ENTFNCREAT:
    *comm.intsLt1 = creat(comm.ioparm2, 0777); break;

case ENTFNACCESS:
    *comm.intsLt1 = access(comm.ioparm2, comm.intparm2); break;

case ENTFNCALL:
    unixcall(); break;

case ENTFQUIT:
    quitproq(); break;

default:
    abend(999);
};
fprintf(stderr,"Main program has terminated\n");
}
pas.con prog unix concurrent pascal library
* copyright robert young 1982
*
******************************************************************************
* concurrent pascal library routines
******************************************************************************
*
* robert young
* department of computer science
* kansas state university
*
* these routines are the library routines supporting
* cpascal/32 under unix. they are based on those
* supplied with pascal/32 and on those used in the
* navy 16-bit system with extension to support a kernel
* written in pascal/32 and multi-level concurrent program
*
* additional user entry points added by
* patrick wayne ireland
* may 2, 1979
*
* the additional entry points are for the following:
* (1) svc1's
* (2) svc2,23's (GET ELAPSED TIME)
* (3) svc7's
*
* unix file system interface points added by
* martin wilde
* november 1983
*
* the additional entry points added are:
* (1) unix open
* (2) unix read
* (3) unix write
* (4) unix close
* (5) unix call
* (6) exit program
*
******************************************************************************
*
entry p_main task entry point
entry serve kernel process serve routine
entry kerninit kernel initialization
entry poplevels kernel pop multiple levels
entry currlevel kernel current level function
entry mark kernel interface to mark
entry release kernel interface to release
entry getmem kernel interface to get memory
entry p_conlib library base
entry c_main addr of main program
space 1
extrn C$MAIN entry to concurrent program
* extrn P$BASE          base address for conc program
extrn signal
extrn abort

    title machine definitions
*******************************************************************************
*
*    library options
*
*******************************************************************************
libgate equ 1           include library gate assist code
space 3
*******************************************************************************
*
*    machine definitions
*
*******************************************************************************
byte equ 1             byte length in addressing units
halfword equ 2         halfword
boolean equ 2          booleans are 2 bytes in pascal/32
integer equ 4          integers are 4 bytes in pascal/32
address equ 4          addresses are 4 bytes in pascal/32
real.sql equ 4         single precision real
real.db1 equ 8         double precision real
*
ubyte equ 0            shift to convert byte/byte
lhalfword equ 1        shift to convert hword/byte
lboolean equ 2         shift to convert boolean/byte
lintger equ 2          shift to convert integer/byte
laddress equ 2         shift to convert address/byte

    title register definitions
*******************************************************************************
*
*    register definitions
*
*
r0     equ 0           r0 = stack top limit
sktp   equ 0
r1     equ 1           r1 = global base
gb     equ 1
r2     equ 2           r2 = local base
lb     equ 2
r3     equ 3
r4     equ 4
r5     equ 5
r6     equ 6
r7     equ 7
r8     equ 8
r9     equ 9
r10    equ 10
r11    equ 11          r11 = library scratch
r12    equ 12          r12 = library scratch
lscr1  equ 12          r13 = library function result
lscr2  equ 11
r13    equ 13
arg2 equ 13 r13 = library argument 2
r14 equ 14 r14 = library argument 1
arg1 equ 14 r15 = user link
r15 equ 15 r15 = library link

*  *
* error codes  *
*  *
err equ x'8804' define error pseudo-op
*  *
irng.er equ 1 inner range error
prng.er equ 2 parameter range error
vrg.er equ 3 value range error
case.er equ 4 case label error
trnc.er equ 5 truncl biff range error
vtnt.er equ 6 variant tag error
pntr.er equ 7 pointer error
stck.er equ 8 stack overflow error
heap.er equ 9 heap overflow error
inst.er equ 16 illegal instruction error
memf.er equ 17 memory fault error
ovfl.er equ 18 arithmetic overflow error

*  *
title system data structures  *
*  *
ms struc stack record
*
ll ds address local link
ra ds address return address
ql ds address global link
ld1 ds address level descriptor link
mslen equ * length non-prefix stack
pl ds address prefix entry link
ends

*  *
* monitor gate  *
*  *
gate struc

gatesem ds halfword semaphore lock
gatelock ds boolean normal boolean lock
gateq ds address queue
gateowner ds address owning process
ends

*  *
* level descriptor record  *
*  *
ld struc

ld1 ds address local link
ldra ds address return address
ldql ds address global link
ldldl ds address level link
ldsd1 ds address old stack descriptor base
ldstl ds address old stack limit
ldlevl equ ldsdl old level (redefined byte)
lpgate ds gate gate for prefix
ends
space 3

* stack descriptor record
* sd struct
stackbot ds address bottom of stack
stacktop ds address top of stack
heapbot ds address top of heap
heapbot ds address bottom of heap
seqlstack ds boolean true if sequential stack
ends

* kernel communications area
* comm struct
entrcode ds halfword kernel entry code
newpcb das 1 new pcb to serve
qparm1 das 1 queue variable parameter
qparm2 das 1 queue variable parameter
gateparm das 1 gate parameter
pcbparm das 1 pcb parm (init process)
intparm1 ds integer integer value parameter
intparm2 ds halfword integer value parameter
ioparm1 das 1 ioparm parameter
ioparm2 das 1 io buffer parameter
intrslt1 das 1 integer func result addr
intrslt2 das 1 integer func result addr
kern7 das 1 kernel c stack pointer
svc1ptr das 1 address of svc1 block
svc7ptr das 1 address of svc7 block
ldparm das 1 ld parameter
svc15ptr das 1 svc 15 parm parameter
segmap ds 16*halfword segment sizes
ends

* process control block
* pcb struct
qlink das 2 queue links
stat das 1 tsw stat
pc das 1 program counter
sdh das 1 stack descriptor base
ldb das 1 level descriptor base
errcode ds halfword error code
errline ds halfword error line number
allregs ds boolean restore all regs
stoplevel ds halfword    stop intercept level
grreqs das 16     general req save area
fpreqs ds 8*real.sql snql precision fp reqs
dpreqs ds 8*real=dbl    dbl precision fp reqs
pcblad ds 1d     initial level descriptor
pcdbpb das 1      prefix entry chain base
fctild das 1      first child process
nchild das 1      next sibling process
pcbparnt das 1    parent process
pcbcrelvl ds halfword creation level
pcbexitvl ds halfword exit level
pcbwtype ds halfword wait type
pcblid ds halfword process sequence number
pcblen equ x'140'  compiler pcb length
ends

*     title library links
*
*     library routine links
*
pure

entry    crt0.o

crt0.o equ *

*  following code is crt0.s which must be at loc 0
*
.exit equ 1

entry _exit
entry environ
extrn main
extrn exit

* rearrange args on stack & call main routine
sis r7,2*adc
lda r0,2*adc(r7)
sta r0,0(r7)
la r1,3*adc(r7)
sta r1,adc(r7)
enlvp equ *
lda r0,0(r1)
bz fnndenv
ais r1,adc
b enlvp

fnndenv equ *
ais r1,adc
sta r1,2*adc(r7)   clobber orig argc
sta r1,environ
bal r15,main

* if main routine returns, exit
sta r0,0(r7)
bal r15,exit
*
_exit equ *
lda r0,0(r7)
svc 0,exit
bss
environ das 1

pure

org 0+crt0.o+x'100' start at end of udl
p_links ds 6*41 space for library links
title udl extension
*
*
low core data areas
*

bss
acomm das 1 address of kernel comm area
currrcb das 1 current pcb
kernlock ds boolean executing "DISABLED" code
ekernpend ds boolean trap pending/task disabled
kernsave das 1 save area on trap entry
kernsbot das 1 kernel stack bottom
usertsw das 2 user dispatch tsw area
kernactv ds boolean executing kernel code
kernexit ds 6 rx3 branch to exit library
lastpc das 1 last task q serve pc
*odebase dac P$BASE base address of conc program
asegmap das 1 addr of segment map

pure
title kernel entry codes
*
*
kern entry codes
*
k$error equ 0 user error
k$qserv equ 1 task queue service
k$pfsstop equ 2 pref exit stop intercept
k$cproq equ 3 conc program entry
k$ninit equ 4 process init
k$term equ 5 process termination
k$cproqx equ 6 conc program exit
k$delay equ 7 delay standard proc
k$cont equ 8 continue standard proc
k$exchange equ 9 exchange standard proc
k$minit equ 10 monitor init
k$menter equ 11 monitor entry
k$mexit equ 12 monitor exit
k$io equ 13 io standard proc
k$trace equ 14 trace
k$peater equ 15 prefix entry
k$peexit equ 16 prefix exit
k$kercal equ 17 start of user-defined
k$attr equ k$kercal+0 attribute function
k$sethp equ k$kercal+1 setheap function
k$start equ k$kercal+2 start procedure
k$stop equ k$kercal+3 stop procedure
k$wait equ k$kercal+4 wait procedure
k$realm equ k$kercal+5 realtime procedure
k$svc1 equ k$kercal+6 svc 1 procedure
k$svc2 equ k$kerca1+7 svc 2 procedure
k$svc7 equ k$kerca1+8 svc 7 procedure
k$getmem equ k$kerca1+9 get remaining memory
k$hrkpt equ k$kerca1+10 user breakpoint
k$svc15 equ k$kerca1+11 svc 15 procedure
k$unopen equ k$kerca1+12 unix open function
k$unread equ k$kerca1+13 unix read function
k$unwrite equ k$kerca1+14 unix write function
k$unclose equ k$kerca1+15 unix close function
k$uncreat equ k$kerca1+16 unix create function
k$unaccess equ k$kerca1+17 unix access function
k$unccall equ k$kerca1+18 unix call function
k$quit equ k$kerca1+19 exit concurrent program

* space 3
* start of code
* space 3
pure
p_conlib equ * start of library
db 'COPYRIGHT 1982 ROBERT YOUNG '
align 2
title p_sprog - enter sequential program

* p_sprog - sequential program entry
* called bal 1llink,p_sprog
* user program invoked by:
  * sta 1b,ms+1l(lb)
  * ai 1b,ms
  * la r3,datascap data space addr
  * li r4,datalen data space length
  * la r5,codeascap code space addr
  * li r6,parmlen parameter length
  * bal 1llink,p prqcal invoke proq thru library
  * dch n number of prefix routines
  * dch prefix1-* prefix routine offset
  * ...
  * dch prefixn-* prefix routine offset

* external routine invoked by:
  * st 1b,ms+1l(lb)
  * ai 1b,ms update local base
  * bal arg1,proc call external procedure
  * dch -1 flag no prefix

* space 3
p_sprog sta gb,gl(lb) save old global base
sta arg1,ra(lb) save ret addr to caller
ldar gb,lb set new gb
br 1llink return to seq1 program
* this should be studied in conjunction with p_prqcal
* which does most of the work for program invocation. note
* this routine must also work with calls to external pascal
* procedures which also call p_sprog. p_prqcal fixes up
* the registers to match those found on an external routine
* call

* **title** p_sprogx - sequential program exit

* **p_sprogx** - sequential program exit

* **called** bal llink,p_sprogx

  p_sprogx lda q0,q1(lb)           restore global envir
  lda llink,ra(lb)                restore return addr
  l h lsrc1,0(llink)              get prefix count
  b m sprogx2                    external routine exit
  lda lsrc1,currpcb              get pcb addr
  lda lsrc2,lq(db(lsrc1))        get ld
  lda r0,rst1(lsrc2)             restore stack limit
  lda lsrc2,lq(db(lsrc2))        get old sdb
  sta lsrc2,sdb(lsrc1)           restore sdb
  lda lsrc2,lq(db(lsrc1))        get ld again
  lda lsrc2,lq(db(lsrc2))        get old ld
  sta lsrc2,lq(db(lsrc1))        restore ld
  l h lsrc1,0(llink)             get prefix count
  aar lsrc1,lsrc1                double it
  aar llink,lsrc1                adjust return address
  sprogx2 ais llink,2             skip prefix routine count
  lda lb,l1(lb)                  restore lb
  br llink                       return to invoker

* **again, p_prqcal should be studied to see how ld is built.**

* **title** p_new - allocate space on the heap

* **p_new** - allocate space on the heap

* **called** li arg1,length      length of area
  li arg2,stacklen(lb)           current stack top
  b m llink,p_new                call lib routine
  st reslt,answer                store result

  p_new lda lsrc1,currpcb        get current pcb
  lda lsrc1,sdb(lsrc1)           get stack descriptor base
  ldar lsrc2,arg2                save stack top
  l reslt,heaptop(lsrc1)         get current heaptop
  c reslt,0(reslt)               check if its still ok
  bne new.ov                    no--heap overrun
  sr reslt,arg1                 back up by item length
  ni reslt,-8                    round to double word
  si reslt,address              back up ptr for ptr check
  clr reslt,lsrc2               did we hit stack top??
bl  new.ov           yes--heap overrun
st  reslt,0(reslt)   set pointer check word
new.ok  st  reslt,heaptop(lscr1) store new heap top
lr  r0,reslt        set stack top limit
br  ll ink          return to caller
new.ov  err  heap.er,0 heap overrun error

* items on the heap are doubleword aligned and are preceded
* by a word which points to itself. this is used for ptr
* checks and to ensure that the stack has not run into the
* heap since we do not currently store the max stack
* increment on procedure entry as was done in the 16 bit
* version. note that the heap grows downward from the top
* of the region as in the 16 bit version but differing from
* the navy pascal/32. thus heapbot is a higher address than
* heaptop. each program has a stack descriptor which
* contains the heap extents. this unlike the navy pascal/32
* code allows multiple sequential programs in one address
* space. the stack descriptor is addressed via the stack
* stack descriptor base in the pcb.

     title p_mark - mark the heap top for later release

* p_mark - mark the heap top for later release
* called bal  ll ink,p_mark
*     st  reslt,answer
p_mark li  arg1,0       set 0-length element
l dar arg2,lb        set dummy stack top
b   p_new          go allocate it and return

     title p_rel - release heap space

* p_rel - release heap space
* called l  arg1,releasepoint
*     bal  ll ink,p_rel release

p_rel  lda lscr1,currcrb get current pcb
lda lscr1,sdb(lscr1) get stack descriptor addr
cl  arg1,heaptop(lscr1) above heap top?
bl  relerr         yes
cl  arg1,heaptop(lscr1) below heap bottom?
* bnl relerr         yes--error
cl  arg1,0(arg1)    check word ok?
be  relok          yes--continue
relerr err  ptrn.er,0 pointer error
relok  l  lscr2,heaptop(lscr1) get current heap top
li  arg2,0         clear heap to zeros
reloop clr lscr2, arg1 are we to release point?
bnl redone        yes--stop clearing
st  arg2,0(lscr2)  clear a word
ai  lscr2,address  increment
b   reloop        continue clearing
redone st  arg1,heaptop(lscr1) store new heap top
lr  r0,arg1       set new stack top limit
br llink return to user

* note that the release point actually includes a check
* word. the check word will not be released.

* title p_fort - fortran function/subroutine call
* p_fort - fortran function/subroutine call interface
*
* called st lb,displ(lb)
* ai lb,displ
* la arg1,routine
* bal llink,p_fort
* dc h'PARMC'T parm count
* dc h'PARMLN' pascal parm list len
*
* p_alloc equ *
* temporary kludge
p_fort sta qb,8(lb) save gb in mark stack
sta r0,12(lb) save stack limit
ldai r3,4(llink) compute return addr
sta r3,4(lb) save return address
ldai r3,0(lb) start of pascal parms
ah r3,2(llink) add length of pascal parm
lh r4,0(llink) get parm count
ais r4,1 increment
ar r4,r4 calc 2*(n+1) for fortran
st r4,0(r3) store it before fort parm
ar r4,r4 double again to get actual
ar r4,r3 calc addr of end of parms
li r5,'23014300' get nop and front of rx3
sta r5,0(r4) store it following parm
li r5,'40000000' get rx3 format part
aai r5,p_fort save return point
sta r5,4(r4) store it following
sta lb,8(r4) ptr back to mark stack
ldai r5,12+576(r4) calc end of fortran
clar r0,r5 did we overflow stack?
bfc 8,fort2 no--fine
err stack.err,0 stack overflow error
fort2 ldar r12,r5 set fort'S WORK AREA PTR
ldai r15,2(r3) ptr to stuff from fort
br arg1 branch to fortran routine
p_fort equ *
lpa lb,-576-4(r12) get ptr to mark stack
lda gb,8(lb) get global base
lda r0,12(lb) get stack limit
lda llink,4(lb) get return addr
lda lb,0(lb) get old local base
br llink return to caller
*
* this routine provides linkage to fortran 6 routines
* including members of the fort6 run time library. the
* routines are declared 'FORTRAN' rather than 'EXTERN' which
* causes the compiler to build a second fortran-style
* parameter list in which all parameters are passed by
  * reference. this list includes space at the front for
  * insertion of the stuff fortran requires preceding the
  * list \((2(n+1))\). this routine saves the pascal environment
  * in the 16-byte mark stack area and inserts a branch to
  * p_fortr following the parameter list to regain control
  * when fortran returns. register 12 is set to point to the
  * top of a 576 byte library work area allocated on the
  * stack. on return, r12 is used to find the
  * pointer back to the mark stack for return to pascal. for
  * more information on fortran calling conventions, see
  * interdata publication 29-565 fortran vi 32-bit run time
  * library technical description.

  title p_scomp - compare sets for inclusion

  p_scomp - compare sets for inclusion

  * computes \((l \leq r)\)
  * computes \((l \text{ and } r) = l\), status will show equal if true

  * called la arg1,l
  * la arg2,r
  * bal llink, p_scomp

  * called la arg1,l  
  * la arg2,r  
  * bal llink, p_scomp

  p_scomp 1 lscr1,0(arg1) generate l and r
  n lscr1,0(arg2)
  cl lscr1,0(arg1) then compare with l
  bnzr llink
  l lscr1,integer(arg1) just repeat
  n lscr1,integer(arg2)
  cl lscr1,integer(arg1)
  bnzr llink
  l lscr1,2*integer(arg1)
  n lscr1,2*integer(arg2)
  cl lscr1,2*integer(arg1)
  bnzr llink
  l lscr1,3*integer(arg1)
  n lscr1,3*integer(arg2)
  cl lscr1,3*integer(arg1)
  br llink

  title p_sand - set and

  p_sand - set and

  * computes \(l := l \text{ and } r\)

  * called la arg1,l
  * lda l arg2,r
  * bal llink, p_sand

  p_sand 1 lscr1,0(arg2) just and a word at a time
  n lscr1,0(arg1)
st  lscr1,0(arg1)
l  lscr1,integer(arg2)
n  lscr1,integer(arg1)
st  lscr1,integer(arg1)
l  lscr1,2*integer(arg2)
n  lscr1,2*integer(arg1)
st  lscr1,2*integer(arg1)
l  lscr1,3*integer(arg2)
r  lscr1,3*integer(arg1)
st  lscr1,3*integer(arg1)
br  llink
title  p_sor - set or

*p_sor - set or

* compute l := l or r
*
called la  arg1,l
*  lda1  arg2,r
*  bal  llink,p_sor
*
p_sor  l  lscr1,0(arg2)                      just or a word at a time
  o  lscr1,0(arg1)
st  lscr1,0(arg1)
l  lscr1,integer(arg2)
o  lscr1,integer(arg1)
st  lscr1,integer(arg1)
l  lscr1,2*integer(arg2)
o  lscr1,2*integer(arg1)
st  lscr1,2*integer(arg1)
l  lscr1,3*integer(arg2)
o  lscr1,3*integer(arg1)
st  lscr1,3*integer(arg1)
br  llink
title  p_sdif - set difference

*p_sdif - set difference

* computes l := l - r
*
called la  arg1,l
*  lda1  arg2,r
*  bal  llink,p_sdif
*
p_sdif  lcs  lscr1,1                get x'FFFFFFFF'
x  lscr1,0(arg2)              generate l := l and not r
n  lscr1,0(arg1)
st  lscr1,0(arg1)
lcs  lscr1,1
x  lscr1,integer(arg2)
n  lscr1,integer(arg1)
st  lscr1,integer(arg1)
lcs  lscr1,1
x  lscr1,2*integer(arg2)
n  lscr1,2*integer(arq1)
st  lscr1,2*integer(arq1)
lcs lscr1,1
x  lscr1,3*integer(arq2)
n  lscr1,3*integer(arq1)
st  lscr1,3*integer(arq1)
br  llink
title  p_stcpy - structure copy

*  
p_stcpy - structure copy
*  
* copy a structure l := r
*  
* called la  arq1,l
  la  arq2,r
  bal  llink,p_stcy
  dch  <structure length>
*  
p_stcpy lh  lscr1,0(llink)  get byte count
  sis  lscr1,integer  decrement for first word
  ais  llink,halfword  compute return address
p_stcpy1 l  lscr2,0(arq2,lscr1) fetch word
st  lscr2,0(arq1,lscr1) store word
sis  lscr1,integer  decr count/offset
bnm  p_stcpy1  continue
br  llink  return to caller
title  p_stcmp - structure compare

*  
p_stcmp - structure compare
*  
* compare two structures
*  
* called la  arg1,l
  la  arg2,r
  bal  llink,p_stcmpx (x = 0..3)
  dch  <structure length>
*  
* the various routines correspond to the number of excess
* bytes over complete words in the structure. the
* structures must be fullword aligned. length does not
* include odd bytes.

p_stcmp0 lis lscr1,0  clear
  sh  lscr1,0(llink)  get neg length
  ais  llink,halfword  compute return address
p_stcmp0 l  lscr2,0(arq1)
cl  lscr2,0(arq2)
brer  llink
ais  arg1,integer
ais  arg2,integer
ais  lscr1,integer
bnz  p_stcmp0
lis lscr2,0  set cond code zero
br  llink
p_stcmp1 ais llink,halfword
lis lscr1,0
sh lscr1,-halfword(llink)
bz p_stcmp1a
p_stcmp1 l lscr2,0(arg1)
c1 lscr2,0(arg2)
bner llink
ais arg1,integer
ais arg2,integer
ais lscr1,integer
bnz p_stcmp1
p_stcmp1a lb lscr2,0(arg1)
c1b lscr2,0(arg2)
br llink
p_stcmp2 ais llink,halfword
lis lscr1,0
sh lscr1,-halfword(llink)
bz p_stcmp2a
p_stcmp2 l lscr2,0(arg1)
c1 lscr2,0(arg2)
bner llink
ais arg1,integer
ais arg2,integer
ais lscr1,integer
bnz p_stcmp2
p_stcmp2a lh lscr2,0(arg1)
c1h lscr2,0(arg2)
br llink
p_stcmp3 ais llink,halfword
lis lscr1,0
sh lscr1,-halfword(llink)
bz p_stcmp3a
p_stcmp3 l lscr2,0(arg1)
c1 lscr2,0(arg2)
bner llink
ais arg1,integer
ais arg2,integer
ais lscr1,integer
bnz p_stcmp3
p_stcmp3a l lscr2,0(arg2)
l lscr1,0(arg1)
srls lscr1,8
srls lscr2,8
c1r lscr1,lscr2
br llink
title p_cprog - concurrent program entry
*
* p_cprog - concurrent program entry
*
* called li arg1,stacklen stack required init proc
* bal llink p_cprog call library
*
* see p_prgcal for program invocation details.
space 3
p_cprog  sa1  lb,pcblen-mslen  offset init proc addr
         sa1  arg1,pcblen-mslen  adjust stack size also
lda  gb,lb  set global base for proc
lda  lsrc1,currpcb  get pcb addr
lda  lsrc2,sdb(lsrc1)  get stack descriptor addr
lis  arg2,0
sth  arg2,seqlstk(lsrc2)  not a seql stack now
lda  lsrc1,stacktop(lsrc2)  get old stack top
aar  lsrc1,arg1  calc new stack top
ais  lsrc1,7  round up to dw boundary
nhi  lsrc1,-8  the benefit of any reals
cla  lsrc1,heaptop(lsrc2)  overflow memory?
bnl  stackerr  yes--stack limit error
sta  lsrc1,stacktop(lsrc2)  set new stack top
lda  r0,lsrc1  stack limit for conc proq
lis  lsrc1,1  get true
sth  lsrc1,kernlock  lock the kernel
lda  lsrc1,a.comm  load comm area addr
ldai  arg1,k$cprog  conc proq entry
b  kerncall  call kernel, exit llink

* note that a conc program must be invoked by a process
* and we will use its pcb. therefore, we need not use up
* memory at the front of the global space for the pcb which
* the compiler allocated. we can also assume we are the
* only user of the data space, and thus kernel
* exclusion is not required immediately. most of the work
* is performed by p_progcall and it should be studied for
* details of entry linkage. currently parameters to conc
* programs are not supported, nor are prefixes.
*
* title p_cprogx - concurrent program exit

* p_cprogx - concurrent program exit
*
* called bal  llink,p_cprogx
*
* space 3
p_cprogx  lda  lsrc2,currpcb  get pcb addr
         lda  lb,ldb(lsrc2)  ret via ld to pcb offset
         lis  lsrc1,1  true
         sth  lsrc1,kernlock  lock the kernel
         lda  lsrc1,a.comm  get comm area addr
         ldai  arg1,k$cprogx  conc proq exit
         b  llink,kerncall  call kernel, return here
         b  p_sprogx  rest is just like seql

* note that it is assumed that the kernel will not re-serve
* the terminated initial process until all of its children
* have been terminated. when it is again served it will
* return to its caller. code in p_sprogx (sequential
* program exit) is used to restore the
* invoker'S ENVIRONMENT, ETC.
title p_prqcal - program invoke

p_prqcal - program invoke

called sta lb,ms+11(lb) save lb
  ai lb,ms update lb
  la r3,dataspac get data space addr
  li r4,datalen get data space length
  la r5,codespac get code space addr
  li r6,paramlen get param+ms length
  bal llink,p_prqcal call thru library
  dch prefix1 prefix routine count
  dch prefix1-* prefix routine offset
  ...
  dch prefixn-* prefix routine offset

program mode uses 40-byte mark stack which is large enough to build a level descriptor.

space 3

p_prqcal sta gb,ldql(lb) save gb
sta llink,ldra(lb) save return addr
lda r7,currpcb get current pcb
lda r8,ldb(r7) get current level desc base
sta r8,ldldl(lb) save old ldb in new ld
lda r9,dsd(r7) get stack desc base
sta r9,ldsd(lb) save old sdb in new ld
lb r10,ldlevel(r8) get caller level-1
ais r10,1 get caller level
stb r10,ldlevel(lb) set called proq level-1
lis r10,0 nil ptr
sta r10,ldgate+gatem(lb) clear the gate
sta r10,ldgate+gateownr(lb) clear owner
sta r10,ldgate+gatem+adc(lb) second word also
lis r10,1 true
sth r10,ldgate+gateclosed(lb) set lock flag
lcs r10,1 get -1 value
sth r10,ldgate+gatesem(lb) set sem unlocked
lb r10,seqlstatk(r9) is this seql stack?
bz prqcal no--no stack top calc necessary
ldai r10,0(r6,lb) calc current stack top
sta r10,stacktop(r9) update stack top in seql sd
prqcal1 sta r0,ldstl(lb) save stack limit in new ld
cla r3,stackbot(r3) does data space contain a sd? be prqcal2 yes--not necessary to initialize it
sta r3,stackbot(r3) set sd as bottom of stack
aar r4,r3 calc end of data space
sis r4,adc back up a word
sta r4,heapbot(r3) that is always heap bot
sta r4,heaptop(r3) and current heap top
sta r4,0(r4) initialize release point
ldai r4,sd(r3) skip over stack desc
ais r4,7 round up to nearest
nih r4,-8    doubleword boundary
sta r4,stacktop(r3)  set current stack top
lis r4,1    true
sth r4,seqlstack(r3) assume a sequential stack for now
prgcal2  lda lscr1,0 get stack top
lis lscr2,0 value for memory clear
lh lscr2,seqlstack(r3) is it a seql stack?
bz prgsker no--can't RE-USE ACTIVE CONC STACK
lda lscr2,heaptop(r3) limit for clear
sar lscr2,r4 amount of memory to clear
bnpl prgsker no space in stack--error
clar lscr2,r6 is there enough space for parms?
bz prgsker no--bad stack

prgcal2a  sta lscr1,0(r4) clear a word
ais r4,adcr incr ptr
sis lscr2,adcr decr bytes left
bp prgcal2a continue clearing stack
sta r3,sdb(r7) set new sdb
lda gb,stacktop(r3) new mark stack/parms go
ldai r4,mslen-ld(gb,r6) calc stack required
cla r4,heaptop(r3) is there enough?nbl prgsker no--error
lm r12,0(lb) get lb/ra/q1/lld from ld
stm r12,0(gb) copy onto new mark stack
sta lb,lldb(r7) set new ldb from old stack
sai r6,ld rem ld size from parmelen
bz prgcal3a nothing more to copy

prgcal3  lda r4,ld-adc(r6,lb) get a parm caller'S STACK
sta r4,mslen-adc(r6,gb) store on new stack
sis r6,4 decrem parmelen
bnz prgcal3 continue copy

prgcal3a  lda lgb set lb to new mark stack
lda gb,gl(lb) old gb for pas/32 compat
lda arg1,ra(lb) restore return addr in r14
ldar llink,arg1 and in llink
lda r0,heaptop(r3) load new stack limit
bnr r5 branch to program

* a program invocation is considered an upward
* level-crossing call. the state of the previous level is
* saved in a level descriptor, which is an extended mark
* stack. the level descriptor exists on the caller's
* STACK FOR PROTECTION, AND IS USED TO
* restore the caller'S ENVIRONMENT EITHER DUE TO A
* program termination or a prefix call. a pseudo-register
* in the pch (the level descriptor base, ldb) points to the
* current level descriptor. the called program receives a
* normal global mark stack (16-bytes) on its stack followed
* by the parameters which are copied from the old stack.
* the caller supplies through extra parameters a variable
* which contains the code (where the first location is
* assumed to contain a branch to the initial procedure),
and a variable which is to be the data space
(stack/heap) for the called program. A stack
descriptor is built in the data space which describes
the used and unused space in the data space. For a
sequential program, the stacktop/stackbot are not
explicitly maintained during execution to reduce
overhead. However, they are calculated
whenever the stack descriptor base (sdb) is to be changed.
Heaptop/bot are maintained by new/mark/release routines.
For a concurrent program, process space is allocated from
the bottom (stackbot/top) as processes are initiated, and
stacktop always points to the beginning of unused space.
A data space variable can be re-used (through a recursive
run-type prefix routine, for example) as long as all uses
are sequential. Re-use is detected by checking the first
word of the data space. Therefore, the first word should
be initialized by the user to zero to prevent
unpredictable results. Sdb is also a pseudo-register
in the pcb.
For more information, refer to my diagrams.

title p_prxcal - prefix routine call

p_prxcal - prefix routine call

called 1a r3,parm1 parameters
  la r4,parm2 ...
  sta lb,ms+ll(lb) save lb
  aai lb,ms update lb
  bal llink,p_prxcal call prefix thru library
  dch routine-id prefix routine id

prefix routine ids are 2, 4, ..., 2*n where there are n
routines.

space 3

p_prxcal sta llink,ra(lb) save return addr
sta gb,gl(lb) save global base
lda lscr1,currpcb get current pcb
lda lscr2,ldb(lscr1) get current ldb
sta lscr2,ldl(lb) current ldb prefix exit
lda gb,ldgl(lscr2) get prefix's global base
lda arg1,ldra(lscr2) get return addr
lda lscr2,lldl(lscr2) get old ldb
sta lscr2,ldb(lscr1) set as new ldb
lh lscr2,0(llink) get prefix routine id
sis lscr2,2 make 0-origin
bm prxerr error in prefix routine id
ch lscr2,0(arg1) over limit?
bnn prxerr yes--give error
aar lscr2,lscr2 double prefix number
ldai lscr2,2(arg1,lscr2) calc addr of displacement
ah lscr2,0(lscr2) calc routine addr
br lscr2 branch to prefix routine
prxerr  err  ptr.err,0  invalid prefix routine id

* prefix routines run in the global environment of the proc
* in which they are declared. however, they run in the
* caller’s stack since concurrent can’t handle recursion
* through the prefix as in solo’s run prefix routine.

* title p_prxent - prefix entry
* p_prxent - prefix entry
* called li  arq2,stacklen  max stack depth from
* bal  llink,p_prxent  call library
* space 3

p_prxent aar  arq2,lb  calc end of local space
clar  arq2,r0  did we hit limit?
bnl  prxstker  yes—stack error
lda  lscr1,currcpcb  get pcb
lda  lscr2,pchbp(lscr1)  previous top prefix
sta  lscr2,pl(lb)  set prefix link
sta  lb,pchbp(lscr1)  set new top prefix
lda  lscr2,ldl(lb)  get ld addr for gate
ifp  libgate  using gate assist code
lis  lscr1,1  constant
ahm  lscr1,ldgate+gatesem(lscr2)  attempt to enter
bnz  penter2  must do it the hard way
lda  lscr1,currcpcb  get pcb addr
sta  lscr1,ldgate+gateownr(lscr2)  set it f
br  llink  return if successful
endc ,  end of assist code
penter2 ldai  arg1,k$penter  prefix enter code
lis  lscr1,1  true
sth  lscr1,kernlock  lock kernel
lda  lscr1,accomm  get comm area
sta  lscr2,ldparm(lscr1)  ld addr is the parm
b    kernfull  call kernel, return user
prxstker equ  * here on stack error on prefix call
lda  lscr1,currcpcb  get pcb
lda  lscr2,ldl(lb)  get ldl
sta  lscr2,ldb(lscr1)  pop a level
lda  gb,ql(lb)  restore gb
lda  llink,ra(lb)  get ra
ais  llink,2  skip over routine id
lda  lb,li(lb)  pop level before err call
err  stack.err,0  stack limit on prefix ent

* title p_prxext - prefix routine exit
* p_prxext - prefix routine exit
* called bal  llink,p_prxext  call library
*
space 3

p_prxext
  lda lscr1, currpcb
  lda lscr2, ld1(lb)
  sta lscr2, ldb(lscr1)
  lda lscr2, pl(lb)
  sta lscr2, pchpb(lscr1)
  lda lscr2, ld1(lb)
  ifp libqate
  lda lscr1, currpcb
  cl lscr1, ldgate+gateownr(lscr2) is this owner?
  be prxext5
  dc h'0'
  set up for clear
  sta lscr1, ldgate+gateownr(lscr2)
  lcs lscr1, 1
  ahm lscr1, ldgate+gatesem(lscr2)
  bm prxext4
  endc
  ldai arg1, k$peeit
  lsis lscr1, 1
  sth lscr1, kernlock
  lda lscr1, acomm
  lda lscr2, ld1(lb)
  sta lscr2, ldparm(lscr1)
  bal llink, kerncall
  call kernel, return here
  prxext4
  lda lscr1, currpcb
  lda lscr2, ld1(lb)
  lbr lscr2, ldlevl(lscr2)
  ch lscr2, stoplevl(lscr1)
  prxext2
  lda qb, gl(lb)
  lda llink, ra(lb)
  b 2(llink)

* this must be extended later to support the stop kernel
* function to check for returning to a stopped level
* (return from a downward call).

  title p_init - process initialization
  * p_init - process initialization
  *
  * called sta lb, ms+ll(lb)
  * la re, travevar
  * aai lb, ms
  * bal llink, pn
  * lda lb, ll(lb)
  *
* initial code in every process except initial process

```
* pr
li r4,stackler  stack requirements
* li r3,parmlen  parameter length
* bal r5,p_init  call library
*
space 3

p_init
lis r7,1  true
sth r7,kernlock lock since updating concurrent sd
lda r7,currcpcb  get initers PCB
lda r8,sdb(r7)  get his sdb
lh r9,seqlstack(r8)  is it a conc stack?
bz initer1  no--bad news
lda r10,stacktop(r8)  where new stack starts
aar r4,r10  calc end of new stack process stack
ais r4,7  round up to dw boundary
nih r4,-8  for the sake of any reals
cla r4,heaptop(r8)  too much?
bnl initstk  yes--stack limit error
sta r4,stacktop(r8)  set new stack top
ldar r4,r10  front of stack is new pcb
sta r4,0(r6)  set pcb trace variable in initer
ldai r6,pchld(r4)  where initial ld is in pcb
lda r6,lbeh(r7)  get initers LDB
sta r6,lbeh(r4)  set the ldb for it
li r9,255  get default stop level
sth r9,stoplevl(r4)  set it
lda r9,lbeh(r7)  set old ldb
lb r9,ldelevl(r9)  get initer level
sth r9,ldelevl(r6)  this is initers LEVEL
lda r6,sdb(r7)  get initers SDB
sta r6,sdb(r4)  this is new sdb also for conc
sta r5,pc(r4)  set new pc for new process
sta r4,lb*adc+gpreqs(r4)  set his new lb
sta r4,gb*adc+gpreqs(r4)  and his new gb
lda r5,stacktop(r8)  get end of his stack
sta r5,r0*adc+gpreqs(r4)  set his new stack limit
lis r6,0  false
sth r6,allreqs(r4)  don't NEED ALLREQS RESTORE
sta r6,pchpb(r4)  null prefix chain
ldar r3,r3  any parms to copy?
bz init2a  no--good
init2
lda r5,0(r3,lb)  copy from lb with 4-byte ms
sta r5,pchlen-adc(r3,r4)  onto new stack
sis r3,adc  decr byte counter
bp init2  continue copy
init2a
equ *
lda 1scrr1,acomm  get comm area addr
sta r4,pcbparm(1scrr1)  set pcb parameter
ldai arg1,k$init  init entry to kernel
b kerncall call kernel, ret to initer via llink
initer1
dc h*0'  re-using seql stack for conc
initstk
stm r0,gpreqs(r7)  save regs in pcb
lis r8,1  true
```
sth r8, allregs(r7) force full restore
ldai r8, initstk2 where to return unlocked
sta r8, pc(r7) set in pcb
lda r3, acomm set up for common code
b testq go enable and re-serve
initstk2 err stk.err, 0 stack limit on process init

* process init allocates process space from the init'er's
* DATA space (which is really the whole current program's
* DATA SPACE).  The first part of that space is the new
* process's PCB and the pcb is followed by the init
* parameters.  The kernel is invoked to
* notify it of the new process.  The new process begins
* execution at its initial statement.
*
* title p_term - process termination
*
* p_term - process termination
*
* called bal llink, p_term call library
*
* space 3
*p_term
lis lscr1, 1 true
sth lscr1, kernlock lock kernel for call
lda lscr1, acomm get comm area addr
ldai arg1, k$term entry code
bal llink, kerncall call the kernel
dc h'0' error if he returns
title monitor procedures
*
* p_cont - continue standard procedure
*
*p_cont
lis lscr1, 1 true
sth lscr1, kernlock lock kernel
lda lscr1, acomm get comm area
sta arg1, qparm1(lscr1) queue variable addr
sta qb, gateparm(lscr1) gate address = qb
lda qb, ql(lb) get old qb
lda llink, ra(lb) set return addr from monitor
lda lb, ll(lb) get old lb
ldai arg1, k$cont continue kernel func
b kerncall call kernel
space 3
*
* p_delay - delay standard procedure
*
p_delay
lis lscr1, 1
sth lscr1, kernlock lock kernel
lda lscr1, acomm comm addr
sta arg1, qparm1(lscr1) queue var addr
sta qb, gateparm(lscr1) gate addr
ldai arg1, k$delay kernel func id
b kerncall call kernel
* p_exchnq - exchange standard procedure

```
p_exchnq lis  lscr1,1
    sth  lscr1,kernlock  lock kernel
    lda  lscr1,acomm  comm addr
    sta  arg1,gparm1(lscr1)  queue var 1 addr
    sta  arg2,gparm2(lscr1)  queue var 2 addr
    sta  gb,gateparm(lscr1)  gate addr
    lda  arg1,k$nexchnq  kernel func id
      b  kerncall  call kernel
    eject
```

* p_menter - monitor entry

```
p_menter equ *
    ifp  libgate  using gate assist
    lis  lscr1,1  increment
    ahm  lscr1,gatesem(gb)  attempt to acquire gate
    bsr  llink  return if we got it
  endc

enter3  lda  arg1,k$menter  func id
enter2  lis  lscr1,1  true
    sth  lscr1,kernlock  lock kernel
    lda  lscr1,acomm  get comm addr
    sta  gb,gateparm(lscr1)  set gate addr
    b  kerncall  go call kernel
  space 3
```

* p_mexit - monitor exit

```
p_mexit equ *
    ifp  libgate  using library gate assist
    lcs  lscr1,1  ready to release monitor
    ahm  lscr1,gatesem(gb)  release the lock
    bm  mexit4  return if no one waiting
  endc

mexit3  lda  arg1,k$mexit  func id
    lis  lscr1,1  true
    sth  lscr1,kernlock  lock it up
    lda  lscr1,acomm  get comm area
    sta  gb,gateparm(lscr1)  store gate addr
    lda  gb,ql(lb)  restore gb
    lda  llink,ra(lb)  restore return addr
    lda  lb,ll(lb)  restore lb
      b  kerncall  call kern, ret to mon caller
mexit4  lda  gb,ql(lb)  restore gb
    lda  llink,ra(lb)  restore return addr
    lda  lb,ll(lb)  restore lb
      br  llink  return to mexit caller
  space 3
```

* p_minit - monitor init
* p_minit  lda1 arg1,k$minit  func id  
b     menter2  common code  
title miscellaneous built in procedures  *

* p_io - io standard procedure  *

* p_io  lis  lscr1,1  true  
sth  lscr1,kernlock  lock kernel  
lda  lscr1,acomm  get comm area  
sta  r3,ioparm1(lscr1)  buffer addr  
sta  r4,ioparm2(lscr1)  ioparm addr  
sta  r5,intparm1(lscr1)  io device number  
ldai  arg1,k$mio  kernel func id  
b     kerncall  call kernel  
space 3  *

* p_trace - compiler generated trace  *

* p_trace  lis  lscr1,1  true  
sth  lscr1,kernlock  enter kernel  
lda  lscr1,acomm  get comm area addr  
lh  lscr2,0(llink)  get trace code  
sto  lscr2,intparm1(lscr1)  save in intparm1  
lh  lscr2,2(llink)  get trace arg  
sto  lscr2,intparm2(lscr1)  save in intparm2  
ais  llink,4  adjust return address  
ldai  arg1,k$trace  kernel entry code  
b     kerncall  go call kernel  
title p_kercal - user-defined kernel routine call  *

* p_kercal - user-defined kernel routine call  *

* called la  r3,parm1  parms  
* sta  lb,ms+ll(lb)  save lb  
* aai  lb,ms  update lb  
* bal  llink,p_kercal  call kernel thru library  
* dch  routine-id  kernel routine id (0-origin)  *

* p_kercal  lis  lscr1,1  true  
sth  lscr1,kernlock  lock the kernel  
lh  lscr2,0(llink)  get routine id  
ais  llink,2  update return addr  
aar  lscr2,lscr2  double routine id  
ldai  lscr2,kerdecod(lscr2)  get addr of offset  
lda  lscr1,acomm  get comm area addr  
ah  lscr2,0(lscr2)  calc routine addr  
br  lscr2  branch to proc-dependent routine  *

* the procedure-dependent routines follow this decoding  
table. there must be one entry per routine in the order  
in which they are declared in the user concurrent  
program. the proc-dependent routine must store  
the parameters from the regs into the appropriate
COMM LOCATIONS (COMM-addressed via lscr1), set arg2 to be
the k$ function id for the kernel, and branch to
kerncall2. are in registers r3,r4,r5 dependent
on the number of parameters used. the
function id should be handled as followed as follows:
lda1 arg1,k$xxxxx

space 3

user-defined kernel routine decode table

kerdecode equ *
dcz p_attr-* attribute

dcz p_sethp-* setheap

dcz p_start-* start

dcz p_stop-* stop

dcz p_wait-* wait

dcz p_realm-* realtime

dcz p_svc1-* svc 1

dcz p_svc2-* svc 2

dcz p_svc7-* svc 7

dcz p_getmem-* getmem

dcz p_brkpt-* breakpoint

dcz p_svc15-* svc 15

dcz p_unopen-* unix open

dcz p_unread-* unix read

dcz p_unwrite-* unix write

dcz p_unclose-* unix close

dcz p_uncreat-* unix create

dcz p_unaccess-* unix access

dcz p_uncall-* unix call

dcz p_quit-* exit program
title user-defined kernel routine interfaces

* * *
p_attr - attribute standard function

lda1 arg1,k$attr kernel routine id
sth r3,interpam2(lscr1) shortinteger value parm
lda1 lscr2,-integer(lb) addr of result
sta lscr2,intrslt1(lscr1) store func reslt addr
b kerncall2 back to common

* * *
p_sethp - setheap standard procedure

lda1 arg1,k$sethp kernel routine id
st r3,interpam1(lscr1) integer value parm
b kerncall2 common

* * *
p_start - start standard procedure

lda1 arg1,k$start
b kerncall2 common

* * *
p_stop - stop standard procedure
* p_stop  ldai arg1,k$stop kernel routine id
  st r3,pcbparm1(lscr1) first parm is pcb id
  sth r4,intparm2(lscr1) second is result code
  b kerncal2 back to common

* p_wait - wait standard procedure
* p_wait  ldai arg1,k$wait kernel routine id
  b kerncal2 back to common

* p_realtm - realtime standard function
* p_realtm ldai arg1,k$realtm kernel routine id
  ldai lscr2,-integer(1b) addr of result
  sta lscr2,intrslt1(lscr1) store func reslt addr
  b kerncal2 back to common

* p_svc1 - svc 1 procedure
* p_svc1  ldai arg1,k$ssvc1 kernel routine id
  st r3,svc1ptr(lscr1) parameter block address
  b kerncal2 back to common

* p_svc2 - svc 2,23 procedure (get elapsed time)
* p_svc2  ldai arg1,k$ssvc2 kernel routine id
  st r3,intrslt1(lscr1) parameter address
  b kerncal2 back to common

* p_svc7 - svc 7 procedure
* p_svc7  ldai arg1,k$ssvc7 kernel routine id
  st r3,svc7ptr(lscr1) parameter block address
  b kerncal2 back to common

* p_getmem - get remaining memory std procedure
* p_getmem ldai arg1,k$getmem kernel routine id
  sta r3,intrslt1(lscr1) first int (ptr) result
  sta r4,intrslt2(lscr1) second int result (len)
  b kerncal2 back to common

* p_brkpt - user breakpoint std procedure
* p_brkpt  ldai arg1,k$brkpt kernel routine id
  st r3,intparm1(lscr1) line number parm
  b kerncal2 back to common

* p_svc15 - svc 15 procedure
* p_svc15  ldai arg1,k$ssvc15 kernel routine id
  st r3,svc15ptr(lscr1) parameter block address
  b kerncal2 back to common
* * p_unopen - unix open function
* *
  p_unopen sta  r3,ioparm2(lscr1)
    sth  r4,intparm2(lscr1)
    ldai  lscr2,-integer(lb)
    sta  lscr2,inrslt1(lscr1)
    ldai  arg1,k$unopen
    b  kerncal2

* * p_unread - unix read function
* *
  p_unread sta  r3,intparm1(lscr1)
    sta  r4,ioparm2(lscr1)
    sth  r5,intparm2(lscr1)
    ldai  lscr2,-integer(lb)
    sta  lscr2,inrslt1(lscr1)
    ldai  arg1,k$unread
    b  kerncal2

* * p_unwrite - unix write function
* *
  p_unwrite sta  r3,intparm1(lscr1)
    sta  r4,ioparm2(lscr1)
    sth  r5,intparm2(lscr1)
    ldai  lscr2,-integer(lb)
    sta  lscr2,inrslt1(lscr1)
    ldai  arg1,k$unwrite
    b  kerncal2

* * p_unclose - unix close function
* *
  p_unclose sta  r3,intparm1(lscr1)
    ldai  lscr2,-integer(lb)
    sta  lscr2,inrslt1(lscr1)
    ldai  arg1,k$unclose
    b  kerncal2

* * p_uncreat - unix create function
* *
  p_uncreat sta  r3,ioparm2(lscr1)
    sth  r4,intparm2(lscr1)
    ldai  lscr2,-integer(lb)
    sta  lscr2,inrslt1(lscr1)
    ldai  arg1,k$uncreat
    b  kerncal2

* * p_unaccess - unix access function
* *
  p_unaccess sta  r3,ioparm2(lscr1)
    sth  r4,intparm2(lscr1)
    ldai  lscr2,-integer(lb)
    sta  lscr2,inrslt1(lscr1)
    ldai  arg1,k$unaccess
b kerncalc2

p_uncall - unix call function

p_uncall ldai arg1,k$uncall
  b kerncalc2

p_quit - exit concurrent program

p_quit ldai arg1,k$quit
  b kerncalc2

title pop levels kernel routine

poplevels(var p:pcb;level:integer;var result:boolean);

the pop levels routine is called by the kernel when it is
desired to pop level activation records (ld's) for a
particular process so that it returns to the statement
following the p_prgcal call at the specified
level. this is generally used when an invoked
program is to be terminated due to an error or user
request. if the process is not currently executing above
the specified level, result is set false and no popping
occurs. otherwise, ld's ARE POPPED UNTIL THE LDB POINTS
to the ld which WILL RETURN to the specified level.
the pc is set to call p_sproq which will
pop the last level. however, if the dummy ld (pcbld)
would be restored, or the dummy ld is encountered during
the search, the pc is set to call p_term to terminate
the process instead. if the requested level is
below the creation level, the pc is
also set to call p_term to terminate the process.
this routine is entered as a pascal external routine by
the kernel.

poplevels shi r7,32 alloc reg save area
stm r8,0(r7) save reqs
ldar r14,r7 local base
lda r3,0+32(r14) get pcb address
lda r4,4+32(r14) get level number to restore to
lda r6,ldbr(r3) get addr of current ld
lhi r0,0 assume we couldn't pop it
clb r4,ldlevl(r6) are we above requested level?
bp popls6 no--can't do it
ch r4,pcbcrvl(r3) pop creation level?
bm popls4 yes--terminate process instead
ldai r8,pcbld(r3) addr of dummy ld for check
popls2 lda r6,ldbr(r3) get current ld
cr r6,r8 is this the dummy ld?
be popls4 yes--terminate process
clb r4,ldlevl(r6) is this desired level?
bmm popls3 yes--go set it up
lda r6,ldld1(r6) get next lower ld
sta r6,1db(r3)  set it as current ld
b  popls2  continue search for proper level
popls3 lda r6,1db(r3)  get 1db
sta r6,1b*adc+gpreqs(r3)  loc base for p_sprocs
lda r6,p_sproq  where to resume execution
popls5 sta r6,pc(r3)  set resume addr
lis r6,0  false
sth r6,allregs(r3)  no allregs restore reqd
lis r0,1  return result
popls6 lm r8,0(r7)  restore regs
ahi r7,32  return to caller
br r15

* function currlevel(p:pcb):integer
*
* this external function provides the current execution
* level to the kernel for a specified pcb.  this is
* required since the kernel cannot access the ld.
*
currlevel lda r3,0(r7)  get pcb addr
lda r4,1db(r3)  get ld addr
lb r0,ldlevel(r4)  get level-1
ais r0,1  get current level
br r15  return to caller

* title kernel interface routines for memory management
*
* procedure mark (var top: integer)
*
* this kernel external routine performs a mark on the
* current process's heap.  this is used to implement
* attribute(heapop).
*
mark equ *
shi r7,96  alloc stack space
stm r0,32(r7)  save all regs
lda lb,currrpcb  get current pcb
lda lb,lb*adc+gpreqs(lb)  get lb val from caller's pcb
bal llink,p_mark  call pascal mark routine
lr r0,reslt  get result
lm r1,32+4(r7)  restore other regs
ahi r7,96  restore stack ptr
br r15  return to caller

* space 3
*
* procedure release (top: integer)
*
release shi r7,96  alloc stack space
stm r0,32(r7)  save regs
lr r14,r7  save base ptr
lda arg1,96+0(r7)  reset pt is first c arg
bal llink,p_rel  call pascal release routine
procedure getmem (var ptr, len: integer)

the getmem procedure allocates the rest of a concurrent
stack to the caller in the form of a pointed-to record
and its length. this kludge exists to allow naxex to be
adaptive to its region size without recomilation.

getmem lda r3,0(r7) get addr of ptr result
lda r4,4(r7) get addr of len result
lda r5,currcpcb get current process pcb
lda r6,sdb(r5) get stack descriptor
lh r1,seg!stack(r6) is this concurrent stack?
bnz getmem2 no--getmem not valid due to heap
lda r1,stacktop(r6) get current top
ais r1,7 round up to a
nhi r1,-8 dw boundary for good measure
lda r2,heaptop(r6) get upper bound
sar r2,r1 calc amt of space available
sis r2,adc back up just for good measure
bnz getmem2 none available--error
sta r1,0(r3) give address of space
sta r2,0(r4) and length
aar r1,r2 calc end of space
sta r1,stacktop(r6) new stack top
getmem3 br r15 return to kernel
getmem2 lis r1,0 get nil/zer
sta r1,0(r3) set nil ptr
sta r2,0(r4) and zero length
b getmem3 return to kernel

title library interface to kernel

kerncall - library interface to kernel

at kerncall it is assumed that llink is the return addr,
arg1 is the kernel functin code, lscr1 has the comm area
address, and r0/qb/lb have been restored. kerncall2 will
restore lb.

kerncall2 lda lb,11(lb) restore lb
kerncall sth arg1,entrcode(lscr1) set entry code
lda lscr2,currcpcb get current pcb addr
sta r0,r0*adc+qregs(lscr2) save r0
sta lb,lb*adc+qregs(lscr2) save lb
sta qb,qb*adc+qregs(lscr2) save qb
sta llink,pc(lscr2) execute at return addr
lis lscr1,0 false
sth lscr1,allreqs(lscr2) all req reste not reqd
b unserve return from kernel serve call
space 3
kernfull - full req save/restore kernel call

kernfull
sth argv1,entrcode(lscr1) save entry code
lda lscr2,currpcb get current pcb addr
stm r0,gpvals(lscr2) save all regs
sta llink,pc(lscr2) return address
lis lscr1,1 true
sth lscr1,allregs(lscr2) all regs
b unserv return from kernel serve call
space 3

unserv - return to kernel from serve call

unserv
lis llink,1 true
sth llink,kernactive pascal/32 kernel active
unserv
lda llink,acomm get comm area addr
lda r7,kern7(llink) get kern stack ptr
lm r8,0(r7) restore regs
ahi r7,32 restore stack ptr
br r15 return to kern following serve call

serve - kernel return point to library

serve
shi r7,32 alloc space on c stack
stm r8,0(r7) save regs
lh r3,kernlock are we locked?
bz *
no--find out why
lda r3,acomm restore comm area address
sta r7,kern7(r3) save kernel c stack ptr
lis r14,0 false
sth r14,kernactive not running pas/32 kernel code
testq equ *
taskq test used to be here
serve
lda r4,currpcb get old pcb addr
bz serve5 none--no svc reqd
lda r5,newpcb(r3) get new pcb to serve
clar r4,r5 same pcb?
be serve2 yes--no fp save reqd
lh r6,allregs(r4) all regs save?
bz serve5 no--no sweat
stme r0,fpvals(r4) save single precision
stmd r0,dpvals(r4) save double precision
serve
lda r5,newpcb(r3) get new pcb
bz serve2 there is none
lh r6,allregs(r5) all reg restore?
bz serve2 no--fine
lme r0,fpvals(r5) load single precision
lmd r0,dpvals(r5) load double precision
serve2
sta r5,currpcb new current pcb
ldar r5,r5 is there a new pcb?
bz wait1 no--enter wait
lh r6,allregs(r5) all reg restore?
bz serve3 no--fine
lis r6,0 false
serve4

```
sth r6, allreqs(r5)       reset allreqs

serve3

```

```
equ *
lis r6, 0
sth r6, kernlock         """" THIS CAN'T GO HERE """
lda r6, pc(r5)           get resume addr
oi r6, y'40000000'       last 2 halfwds of rx3 branch
sta r6, kernexit+2       put in branch instr
lm r0, gpregs(r5)         load regs
b kernexit                exit from kernel
```

```
lda r0, gpregs(r5)        reload r0
lda gb, gb*adc+gpregs(r5) reload gb
lda lb, lb*adc+gpregs(r5) reload lb
lda llink, pc(r5)         load return addr
lis r6, 0
sth r6, kernlock
lh r7, kernpend           running disabled?
bzr llink                 no--exit
dc h'0'                   can't get here
wait1

```

```
lis r7, 0                 false
sth r7, kernpend          not disabled
sth r7, kernlock          not locked
dc h'0'                   can't get here either
```

```
svc 9, waitsw             enter trap wait
```

```
title kerninit - kernel initialization part 2
```

```
* Kerninit (comm, stacksize, taskqsize, var init_pcb)
```

```
kerninit equ *
```

```
entry from kernel as external routine
shi r7, 32
stm r8, 0(r7)             save regs
lda r3, 0+32(r7)          get first parameter
sta r3, acomm             set comm area addr in low core
lis r6, 1                 true
sth r6, kernlock          kernel is locked now
sth r6, kernactv          and exec pas/32 kernel code
lhi r6, x'4300'           front of uncond rx3 branch
sth r6, kernexit          fill in in branch instr
```

```
lhi r6, 0
sth r6, kernpend          kernel is running enabled
```

```
ldai r3, p_links          addr for library links
```

```
ldai r4, linktab          table of links
```

```
initlnk 1                  fetch entry
```

```
bm initlnkd we're done at end of table
```

```
lhi r0, x'4300'           unconditional branch
```

```
sth r0, 0(r3)
```

```
lt r0, 0(r4)               first half of addr
```

```
ohi r0, x'4000'           make it rx3 branch
```

```
st h r0, 2(r3)            put in instr
```

```
lh r0, 2(r4)               get second half
```

```
st h r0, 4(r3)           finish instr
```

```
ais r3, 6
```

```
ais r4, 4
```

```
b initlnk                 continue in table
```

```
initlnkd equ *             end of table
```
initialize error handler

shi r7,32               get more space on stack
ldai r8,sigtab table of signals
errinit1 equ *
    lh r0,0(r8)      get signal number
    bz errinit3 we're at end of table
    st r0,0(r7)     set first parm
    lhi r0,1        set to ignore first
    sta r0,4(r7)    set it
    bal r15,signal  find out what it is set to
    thi r0,1       was it ignored?
    bnz errinit4    yes--then leave it alone
    ldai r0,p$error error handler
    sta r0,4(r7)    set this also
    bal r15,signal  initialize signal interface
errinit4 equ *
    ais r8,2       advance to next entry
    b errinit1     continue processing
errinit3 equ *
    h r0,y'80007f00' psw status in no-stack mode
    sta r0,0(r7)
    ldoi r0,errinit2
    sta r0,4(r7)    resume psr
    sta r7,8(r7)    old stack ptr
    svc 0,48       signal
    dc 0,0         switch to no-stack mode
errinit2 equ *
    ahi r7,32   resumes execution here
    lm r8,0(r7) restore regs
    ahi r7,32
    br r15      return to c kernel

error handler

entry p$error

p$error equ *
    h r0,0(r7)
    st r0,ersig signal number
    lm r8,4(r7) r0-r7
    stm r8,ersave save r0-r7
    lm r8,36(r7) r8-rf
    stm r8,ersave+32 save them also
    lda r0,72(r7)
    sta r0,erpc
    lh r0,kernactv
    bnz kernerr error in kernel--fatal
    shi r7,32
    l r0,ersiq
    st r0,0(r7) re-instate the signal
    ldai r0,p$error
    st r0,4(r7)
    bal r15,signal re-instate the signal we caught
li r0, y'80007f00' no-stack mode psw
sta r0, 0(r7)
ldai r0, error2
sta r0, 4(r7) resume psw
sta r7, 8(r7) stack
svc 0, 48 signal zero
dc 0, 0 enter no-stack mode

error2 equ *
ahi r7, 32 restore stack
lda r1, currpcb get current pcb
lm r8, ersave
stm r8, gpregs(r1)
lm r8, ersave+32
stm r8, gpregs+32(r1)
stm r0, fregs(r1)
stm r0, dregs(r1)
lhi r0, 1
sth r0, allregs(r1)
lda r0, erpc
sta r0, pc(r1)
ldai arg1, k$error
lhi r0, 1
sth r0, kernlock
lda lscr1, acomm get comm area
l r0, ersig
st r0, intparm1(lscr1) signal number
sth arg1, entrdecode(lscr1)
b unserve go to kernel

kernerr equ *
bal r15, abort go abort due to kernel error

p_null equ *
dc h'0'

stackerr err stk.er, 0 disabled stack limit

title constants
*
*
library constants
*
*
c_main dac C$MAIN
align adc

sigtab equ * table of signals to catch
dc h'2' catch irpt signal
dc h'4' illegal instr
dc h'5' trace trap
dc h'7' emt
dc h'8' arith fault
dc h'10' memory fault
dc h'11' segmentation fault
dc h'12' system call error
dc h'13' pipe error
dc h'0' flag end of table

* library link table
*
align adc
linktab equ *
dac p_sproq sequential program entry
dac p_sproqx sequential program exit
dac p_fort fortran routine call
dac p_new new standard procedure
dac p_mark mark standard procedure
dac p_rel release standard proc
dac p_scomp set compare
dac p_sand set and
dac p_sor set or
dac p_sdif set difference
dac p_stcpy structure copy
dac p_stcmp0 structure compare
dac p_stcmp1 structure compare
dac p_stcmp2 structure compare
dac p_stcmp3 structure compare
dac p_prxcal prefix call
dac p_trace compiler generated trace
do 3 extra entries
dac p_null res for pas/32 expansion

* concurrent library links
*
dac p_cproq conc program entry
dac p_init init process
dac p_term process termination
dac p_cproqx conc program exit
dac p_delay delay standard proc
dac p_cont continue standard proc
dac p_exchng exchange standard proc
dac p_minit monitor init
dac p_menter monitor entry
dac p_mexit monitor exit
dac p_prxent prefix entry
dac p_prxext prefix exit
dac p_kercal user-defined kernel call
dac p_prqcal program call
dac p_io io standard proc
do 6 reserved
dac p_null reserved for future use
dac -1 mark end of table

* impure data areas
*
impur
ersig ds adc signal number
ersave ds 16*adc general regs
erpc ds adc program counter
dac end
APPENDIX G

SOLO SYSTEM FILES
The following appendix contains a list of the UNIX files used in the Solo system and their compilation methods if further modifications are desired.

solo32.p: This is the text file which contains the Concurrent Pascal system. It is compiled with the command line:

cc pascal solo32 -o solo32

The resulting object code is placed in a file called solo32.

kernel.c: This is the modified Concurrent Pascal Kernel. The command line for its compilation is:

cc -c kernel.c

The output file has a name of kernel.o.

conlibrary.s: This is the modified Concurrent Pascal Library. The command line for its assembly is:

as conlibrary.s -o conlibrary.o

The object code is then placed in a file called conlibrary.o.

solo.vd: This is the name of Solo's virtual disk. The file contains all of the Sequential Pascal programs that used by Solo. Included in this are the seqcode files and the text files of the programs.
reloc.p : This is the Sequential Pascal Relocator program. It is compiled with the command line of:

pas32 reloc -o reloc

The object code file is given a name of reloc.

solo : This is a shell file which contains the execution commands for the Solo system. As described before it has a format of:

solo32 -k120k -f1x solo.vd -f3 printfile -f4r cardfile

pas32 : This is the modified pas32 compiler shell.

cpascal : This is the modified Concurrent Pascal compiler shell.
Solo32:
A Concurrent Pascal Operating System
with UNIX Interfaces

by

Martin Wilde

R.S., Kansas State University, Manhattan, Kansas, 1983

---------------------

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the
requirements for the Degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1984
ABSTRACT

Solo is an operating system that is written entirely in Concurrent Pascal. Application programs are written in Sequential Pascal and are utilized by Solo to support various functions of an operating system. The system is powerful enough to provide most features that a single user might need while it is small enough so that a person studying operating systems can understand the whole system in a short amount of time. With a few additional functions Solo can be expanded to a useful operating system.

The intent of the project was to reestablish the Solo operating system and implement extensions on the original system to allow interfaces to the UNIX file system and environment. Contained in the report is a description of these extensions and the necessary modifications that were made to reestablish the Solo operating system, the implementation in which Solo allows a user to have access to the UNIX file system and environment.