Operations Systems

R.S. Gaines Editor

Thoth, a Portable Real-Time Operating System

David R. Cheriton, Michael A. Malcolm, Lawrence S. Melen, and Gary R. Sager
University of Waterloo

Thoth is a real-time operating system which is designed to be portable over a large set of machines. It is currently running on two minicomputers with quite different architectures. Both the system and application programs which use it are written in a high-level language. Because the system is implemented by the same software on different hardware, it has the same interface to user programs. Hence, application programs which use Thoth are highly portable. Thoth encourages structuring programs as networks of communicating processes by providing efficient interprocess communication primitives.

Key Words and Phrases: portability, real time, operating systems, minicomputer

CR Categories: 3.80, 4.30, 4.35

1. Introduction

This paper describes a portable real-time operating system called Thoth which has been developed at the University of Waterloo as part of a research study into the feasibility of portable operating systems. Thoth supports multiple processes, dynamic memory allocation, device-independent input/output, a file system, multiple terminals, and swapping. It is currently running on two minicomputers with quite different architectures (Texas Instruments 990 and Data General NOVA).

This research is motivated by the difficulties encountered when moving application programs from one system to another; these difficulties arise when interfacing with the hardware and system software of the target machine. The problems encountered interfacing with new system software are generally more difficult than those of interfacing with new hardware because of the wide variety of abstract machines presented by the compilers, assemblers, loaders, file systems and operating systems of the various target machines. We have taken the approach of developing portable system software and porting it to "bare" hardware. The same system software is used on different hardware, thus the same abstract machine is available to application programs. Thus most application programs which use Thoth are portable if not machine independent.

Most previous work on software portability has focused on problems of porting programs over different operating systems as well as different hardware. To our knowledge, this is the first time an entire system has been designed for portability. Our experience indicates that this approach is practical both in the cost of porting the system and its time and space performance.

An earlier experiment in operating system portability has been reported by Cox [4]. More recently, the UNIX operating system [13] has been moved from a PDP-11/45 to an INTERDATA 7/32; this port was done independently by Miller [11] at the University of Wollongong, and Johnson and Ritchie [7] at Bell Telephone Laboratories.

The design of Thoth strives for more than portability. A second design goal is to provide a system in which programs may be structured using many small concurrent processes. We have aimed for efficient interprocess communication and inexpensive processes to make this structuring technique attractive.

A third design goal is that the system meet the demands of real-time applications. To help meet this goal, the system guarantees that the worst-case time for response to certain external events (interrupt requests) is bounded by a small machine-dependent constant.

A fourth design goal is that the system be adaptable to a variety of real-time applications. A range of system configurations is possible: A stand-alone application program can use a version of the kernel which supports dynamic memory allocation and interprocess communication. Larger configurations support process destruction, a device-independent input-output system, a tree-structured file system, multiple terminals, and swapping.

Thoth as described in this paper has evolved through several versions. It was originally developed using a

Authors' present addresses: D. Cheriton, Department of Computer Science, University of British Columbia, Vancouver, B.C., Canada V6T 1W5; M.A. Malcolm and G.R. Sager, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1; L.S. Melen, Transport Canada, Air Traffic Services, Place de Ville, Ottawa, Canada, K1A 0N8.

A version of this paper was presented at the Sixth Symposium on Operating Systems Principles, West Lafayette, Indiana, November 16–18, 1977.

The remaining papers appear in Operating Systems Review (ACM SIGOPS Newsletter), Vol. 11, No. 5 (Special Issue). This special issue is available prepaid from ACM, P.O. Box 12105, Church Street Station, New York, NY 10249; ACM or SIGOPS members $9.00, all others $12.00.

compiler and other support software running on a Honeywell 6050. The first version was running on a Data General NOVA in May 1976. It was ported to a Texas Instruments 990 in August 1976 using the Honeywell for software development. Since that time, new versions have been developed for either the TI or the NOVA, then ported to the other machine when complete. Larger configurations of Thoth can be used to develop software; in particular, all system maintenance and development are now done using a multiterminal Thoth system.

2. The Thoth Machine

Thoth implements an abstract machine referred to as the *Thoth machine*. The Thoth machine is implemented as a base language and a set of system functions implemented in this language.

The base language, described by Braga [1], models the hardware facilities available on a large number of machines. It was designed to conceal hardware idiosyncrasies while avoiding being a barrier between the programmer and the hardware. This is a stack-oriented language derived from B [6], which is a descendant of BCPL [12]. As in BCPL, programs in our base language are written as a set of functions and data modules which have global scope. Variables local to a particular function, including its parameters, are dynamically allocated on a stack so that functions can be reentrant.

The language includes statements for disabling and enabling hardware interrupts to provide indivisible executions of sections of code. There is also a **twit** statement which provides a way of inserting assembly language directly into the code. This makes it possible to insert special I/O instructions, and makes the presence of such machine-dependent code obvious. The name of the statement was chosen to suggest its low-level nature, and to discourage its indiscriminant use. These statements are used almost exclusively for implementing primitive functions in the operating system, and are seldom necessary or desirable in user programs.

Under Thoth, a function is invoked as either a subroutine or as a separate process. When invoked as a subroutine, the function uses the caller's stack. When invoked as a process, a new stack is allocated separate from that of the invoking process.

A program comprises a tree of processes which interface with the Thoth machine via system function calls. This tree of "user processes" is actually a subtree of the tree of system processes.

With small configurations, the system functions are all linked with the user program into an executable core image. We have used the convention of beginning all global system names with a "." so the user can avoid inadvertently replacing a system function or data module by starting identifiers with some other character.

The remainder of this section describes the system functions.

Memory allocation. A contiguous vector of memory is allocated by the function call

vec = .Alloc_vec(size)

This returns a pointer to a vector of size+1 words, which can be indexed as vec[0] through vec[size]. The allocation is done by means of a next-fit algorithm based on the boundary-tag method of Knuth [8]. The vector is returned to the free list by

.Free(vec)

Process creation. A process is created with a specified stack size by

id = .Create(funct, stack_size)

where funct is a pointer to the function to be invoked as a process. A unique nonzero process id is returned which is used in future references to the new process. The created process becomes a direct descendant of its creator in the tree of processes. The process is created in the *embryonic* state and cannot execute until it is readied:

.Ready(id, argument_list)

.Ready passes the (optional) arguments to the new process and makes it eligible for execution.

An optional third argument can be passed to .Create to specify the priority of the new process; the default priority level is 0.

CPU allocation. Processes are allocated the CPU as follows. A process eligible for execution is said to be *ready*. A process which is not ready is said to be *blocked*. Of the highest priority ready processes, the process ready for the longest time is allocated the CPU, and is said to be *active*.

The active process relinquishes the CPU either by blocking or by being preempted when a higher priority process becomes ready. The latter is caused either by a hardware interrupt or by an action of the active process. The active process may block by attempting to communicate with another process or by waiting for an interrupt to occur. Certain system functions, such as input and output primitives, use interprocess communication and may block the active process.

It follows that on a single processor machine, a process executes indivisibly with respect to processes of the same or lower priority until it blocks. This *relative indivisibility* is a useful property of Thoth processes.

The priority of a process remains fixed throughout its lifetime. The *highest* user priority is 0; lower priorities are greater than 0. The root of a subtree of user processes normally has priority 0. Thoth system processes have higher priorities than user processes with the exception of a default process which has lower priority than all user processes and is always ready.

Interprocess communication. There are four primitives for passing messages between processes. All messages are 8 words in length.

A process sends a message to another process by

id = .Send(msg, id)

The contents of the 8-word msg vector is sent to the process specified by id. The sending process blocks until the receiving process has received the message and sent back an 8-word reply with .Reply. The reply message overwrites the original msg vector.

If the receiving process does not exist, .Send returns 0 and the msg vector remains unchanged. Normally .Send returns the id of the process which sent the reply.

The receiving process uses

id = .Receive(msg)

or

id = .Receive(msg, id)

The receiving process blocks, if necessary, to receive an 8-word message in its msg vector. When the optional id parameter is present, the message must come from the specified process. When the id parameter is not present, the first process sending to the receiving process will satisfy the receive. The id of the sending process is returned for later use in .Reply:

.Reply(msg, id)

The 8-word reply containing in the msg vector is sent to the specified process awaiting a reply from the receiving process. The sending process is readied upon receiving the reply, and the replying process does not block.

An attempt to receive from a nonexistent process results in an undefined message and a 0 being returned by .Receive. A .Reply to a nonexistent process is a null operation.

Instead of replying to a sender, the receiving process can forward the message, possibly changing its contents, to another process:

.Forward(msg, from_id, to_id)

The process specified by from_id must be blocked awaiting a reply from the forwarding process. The effect of .Forward is the same as if the from_id process had performed a .Send to the process specified by to_id of the 8-word message in the forwarding process's msg vector. The forwarding process does not block.

The interprocess communication primitives can be used for synchronizing and eliminate the need for primitives like semaphores. These primitives have changed a number of times as the system has evolved. Their semantics and efficiency have considerable impact on the system and seem worthy of further study.

Interrupts. Interrupts are handled by system processes. These processes use the function call

.Await_interrupt(device_id)

to block until an interrupt occurs for the specified device. Such an interrupt can only occur when no processes of the same or higher priority are active. Hence an interrupt causes its associated process to become both ready and active.

Process destruction. Any process can destroy a process (possibly itself) by

.Destroy(id)

The destroyed process ceases to exist in the sense that its id becomes invalid, it can no longer execute, and its stack and all memory it has allocated are returned to the free list. When a process is destroyed, all of its descendants are also destroyed.

For any process blocked doing a .Send to or .Receive from a process which is destroyed, the result is the same as if the .Send or .Receive had been executed for a nonexistent process. In particular, when a process is destroyed, all blocked processes attempting the send to or receive from the deceased process become ready.

Teams. Each process belongs to a *team* which is a set of processes sharing a common address space and a common free list of memory resources. Processes on the same team can share data. Processes on different teams cannot share data, but they can communicate via the interprocess communication primitives. There are two types of teams: resident and transient. Processes on *resident* teams remain in memory and are higher priority than those on transient teams. *Transient* teams may be swapped to secondary storage when primary memory becomes scarce. Transient teams can be created dynamically while resident teams are created only when the system is initialized. The priority, and hence the relative indivisibility property, of a process on a transient team applies only with respect to other processes on the same team.

Teams allow the use of physical memories larger than the logical address space on machines with memory management hardware, and greater concurrency via swapping on machines with a secondary storage device suitable for swapping.

Clock. Machines with a source of periodic interrupts can be configured to support a clock abstraction. The current date and time of day is maintained.

The clock can be used by a process to block for a period of time. This is done by either

.Sleep(time_and_date)

or

.Delay(seconds)

A process invoking .Sleep will block until the current time and date becomes equal to or later than that specified by the time_and_date vector. A process invoking .Delay will block until the specified number of seconds has elapsed. Using an optional second argument to .Delay, a process can sleep for as little as one clock interrupt (which is machine dependent). In both cases, the process then becomes ready.

The current date and time of day can be obtained by

.Get_time(time_and_date)

It can be set by

.Set_time(time_and_date)

Input/output. The Thoth input/output system provides a reasonably uniform interface with peripheral devices and files so they can be used interchangeably by most programs. Each device is assigned a unique name. For example, terminals are named "$tty0", "$tty1", ... , the disks are named "$disk0", ... , etc. The random-access memory can be treated as an input/output device, called "$mem". This allows the use of input/output editing functions on strings of characters stored in memory.

A file or device is accessed by

fcb = .Open(pathname, mode)

where pathname is a string specifying either the name of a device or the pathname of a file (which will be defined later) and mode is a string specifying the mode of access (read, write, append or read/write). Append mode is equivalent to write mode on devices and is defined further for files in the next section. In the remainder of this section, we will use the word *file* to mean "file or device."

The function .Open returns a pointer to a *file control block* which contains a description of the accessed file and any necessary buffer(s); this pointer serves as an identifier for the accessed file. The process is said to *own* the fcb and no other process can use it, although other processes can still access the file using separate fcbs.

Each open file has a *current byte position.* When a file is opened (except for append mode), this current byte position is initialized to 0, the beginning of the file.

An fcb can be used to transfer data to or from files after it is *selected.* An fcb is *selected for input* by

.Select_input(fcb)

An fcb is *selected for output* by:

.Select_output(fcb)

These two functions verify the fcb ownership and access mode.

Data is transferred one byte at a time from a file selected for input by

data = .Get ( )

.Get returns the current byte right-adjusted and zero-padded on the left. Similarly, data is transferred to a file selected for output by

.Put(data)

.Put writes the rightmost byte of the data word to the current byte in the file. Both .Put and .Get have the effect of incrementing the current byte position.

These data transfers are implemented with device-dependent buffering schemes. To guarantee that all bytes transferred by .Put have actually been output to the file

.Flush( )

flushes the buffers of the file selected for output.

For devices and files on devices which allow direct access to bytes,

.Seek(fcb, where, how)

changes the current byte position. The "how" parameter specifies the interpretation of the "where" parameter. The three possibilities are: *absolute byte* which sets the current byte position to the where-th byte in the file; *relative byte* which increments the current byte position by where, which may be negative; *absolute block* which sets the current byte position to the first byte in the where-th block of the file. Absolute block seeking applies only to devices and files on devices for which the concept of a block is appropriate.

.Close(fcb)

flushes output (if necessary), removes access to the file and releases memory used for the fcb. When a process is destroyed, all of its accessed files are automatically closed.

File system. Thoth can be configured to support a file system on target machines with one or more direct access secondary storage devices. The file system is structured as a tree in which each node is a file. In addition, each node may have substructure consisting of one or more descendant nodes.

Each node has a name consisting of up to 32 characters and all the immediate descendants of a node have unique names. The root node of the tree has the unique name *. A node is specified by a *pathname* which is a sequence of names separated by the / character. A pathname describes a path through the tree. For example, the pathname * refers to the root node, and the pathname

*/src/fsys/seek

refers to the node named seek which is an immediate descendant of fsys, which is an immediate descendant of src, and src is immediately under the root.

The nodes which are direct descendants, or *sons*, of a given node, their *father*, are ordered. The file system provides functions which return the pathname of the father, the next brother or the first son of the node specified by a given pathname. This enables a program to traverse a subtree of the file system.

Each process has an associated *current node* which is inherited from its parent; it may be changed by

.Set_current_node(pathname)

The concept of current node allows abbreviated references to files in the subtree rooted at the current node. A pathname starting with "/" describes a path starting at the current node. The current node is referred to by @ . For example, if the current node is */src/fsys then the file */src/fsys/seek can also be referred to as either

/seek

or

@/seek

Two functions are used to modify the file system tree structure. A new node is created by

.Make_node(pathname)

The space occupied by a file is reclaimed and, if it is a leaf, the node is deleted by

.Remove_node(pathname)

A file system structure can be *grafted* as a subtree to the file system by

.Graft(pathname, device_name)

The root node of the file system structure on the device specified by device_name can subsequently be referred to as pathname. Nodes in the grafted file system structure can be referred to as described above using pathname as the name of the root node.

.Ungraft(pathname)

ungrafts the file system structure rooted at pathname making the root of the previously grafted substructure inaccessible using pathname. Grafts allow the use of multiple and removable secondary storage devices.

The contents of a file is regarded as an infinite sequence of bytes. Initially all of the bytes are null. The function .Put, described above, is used to modify individual bytes in a file when it is open with write or append access. A file is physically represented by one or more *blocks*, where the size of a block depends on what is convenient or efficient for the particular device. All bytes after the last physical block are null by definition. Files open with write or append access are automatically grown to contain the data written. A file open with write access can be explicitly changed to a specified size by

.Change_file(fcb, size_in_blocks)

after which it will only become smaller by another call to .Change_file or by removing the file, although the file will still be grown as required to contain data written.

The current byte position is determined by

position = .Where(fcb)

The file *mark* is a byte position in the file which is remembered over accesses to the file. The mark is initialized to 0 when the file is created. The mark of the file selected for output is set to the current byte position by

.Mark( )

When a file open with only write or append access is closed, the mark is set to the current byte position. The current byte position is set to the mark by

.Seek_mark(fcb)

or when a file is opened for append access.

.At_mark(fcb)

returns 1 if the current byte position is equal to the mark, and 0 otherwise. The concept of mark generalizes that of "end of file."

Environment enquiry. Environment enquiry is the facility for a program to access parameters describing the target machine. The availability of these parameters may make an otherwise machine dependent program machine independent. Some parameters, such as the number of bits per word, the number of bits per byte, etc., are known at compile time while others, especially those describing the system configuration, are only available at execution time.

Parameters known at compile time are provided in the form of predefined manifest constants. A *manifest constant* is a base language identifier defined to represent a string of text. Every occurrence of a manifest constant is replaced by its definition via textual substitution in the source code during compilation.

Parameters known only at execution time are made available as global variables, or through system function calls.

3. Portability of Thoth

In this section the notion of portable system software is made more precise. We then characterize the set of machines over which Thoth is considered to be portable and discuss the work entailed in porting the system.

With certain programs, such as compilers, assemblers, loaders, etc., one can distinguish the *host machine* on which the program executes from the *target machine* for which the output of the program is intended. We say that a program is *portable* over a set of machines if it costs significantly less to modify it for each machine than to implement and maintain separately. This cost should include the cost of running the program throughout its lifetime; hence, a program that is easy to convert for a new machine but grossly inefficient may not be considered portable. If moving portable software to new host machines requires no modification, it is said to be *machine independent*. Portable software is said to be *machine invariant* if it requires no modification for new target machines. Software that is not machine invariant is said to be *machine specific*.

Portable software has advantages in development, maintenance and adaptability. It is usually less expensive to develop one program for several machines than to customize an entirely separate program for each machine. Moreover, one can justify better design and documentation because of wider applicability. Also, it is easier to maintain one well-designed program than several programs.

Operating system portability is aimed at a problem most prevalent with minicomputers. The decision to acquire a new minicomputer must be largely predicated on the software available for the machine unless substantial time and resources are allocated to software development. Moreover a major task for the manufacturer of a new machine is to develop software for it. Porting Thoth to a new machine provides a substantial body of system software plus a growing collection of machine-independent application programs. This reduces the cost of providing software for a new machine and in the case of owning several different minicomputers, greatly reduces the software maintenance cost when all the machines are running Thoth.

Three main portability problems were addressed during the development of Thoth. The first problem was to design an abstraction of a minicomputer that could be efficiently realized on a large number of machines. The dual of this problem is that of choosing the domain of target machines so that a reasonable (and efficient) abstraction is possible. The second problem was to represent the abstraction in such a form as to minimize the effort required to implement it on target machines. The third problem was to design and implement software tools to automate as much of the implementation as possible.

Implementability over a specific domain of machines was a major consideration during the design of the Thoth Machine abstraction described in Section 2. Some desirable ideas were not incorporated due to the apparent difficulty of implementing them on certain machines. Similarly, some possible target machines were rejected due to their lack of hardware to efficiently implement abstractions thought to be fundamental to any reasonable system.

A high-level language has been used to represent most of the system so that most of the translation into machine code can be done by a compiler. The language has been designed to encourage the use of machine-independent constructs; however machine-specific code can, and sometimes must, be used. To document machine dependencies, the software is divided into components which are stored in separate files, each of which contains either a single function, a set of related global data modules, or a set of related manifest definitions. Each file is classified as either machine-invariant or machine-specific; this classification is implied by the subtree of the file system in which the file is stored. The machine-specific components which need to be modified during a port are thus isolated from machine-invariant code, and easy to find.

The tree structure of the Thoth file system is used to structure the source files to reflect functionality as well as machine dependency. The subtree containing all source files is rooted at */src; immediately below this node are subtrees containing major functional components such as kernel, input/output, file system, etc. Each of these subtrees contains machine-invariant source files immediately below its root plus a further subtree of machine-specific code for each target machine type. Thus, for example, there is a subtree rooted at */src/kernel/nova containing the kernel source files which are specific to Nova computers.

The tree structure of the file system has proved invaluable for managing the over 2000 files of rapidly changing source code. The management of these files is a nontrivial job that will become more difficult as Thoth is ported to new machines.

Some components of the system which are nearly machine-invariant are rendered machine-invariant by replacing machine dependencies with manifests. Then only the manifest definitions occur in a machine-specific source file. For example, the process which is activated by each real-time clock interrupt executes the following function:

```
.Chronographer(timer)
  {
    extrn .Time_vec, .Wake_time_vec, .Time_mods;
    auto i;
    repeat
      {
        START_CLOCK;
        .Await_interrupt(RTC);
        STOP_CLOCK;
        ++.Time_vec[5];
        for(i=5; .Time_vec[i] >= .Time_mods[i];)
          {
            .Time_vec[i] = 0;
            ++.Time_vec[--i];
          }
        if(i == DAYS && .Time_vec[i] == 365 &&
          YEARS[.Time_vec] & 3) \ non-leap year test
          {
            DAYS[.Time_vec] = 0;
            ++YEARS[.Time_vec];
          }
        if(.Compare_vec(.Wake_time_vec, .Time_vec, 5) != 1
            && STATE[timer] == RECEIVE_BLOCKED)
          {
            disable;
            MESSAGE[timer] = WAKE_UP;
            BLOCKED_ON[timer] = 0;
            .Add_ready(timer);
            enable;
          }
      }
  }
```

The manifests START_CLOCK and STOP_CLOCK must be defined for each different target machine. The machine-specific manifest definitions for the Nova are:

```
#START_CLOCK   = twit(.NIO.|.IS., .RTC.);
#STOP_CLOCK    = twit(.NIO.|.IC., .RTC.);
```

For the TI 990 the definitions are:

```
#START_CLOCK   = twit(.CKON.)
#STOP_CLOCK    = twit(.CKOF.)
```

Thoth is also made more portable by the readability of its source code. The most accurate documentation of a program is usually its source; internal and external documentation is often out of date. For this reason, our language design has been strongly influenced by aesthetic considerations which are often dismissed as "syntactic sugar." Since the readability of source code depends heavily on the style of coding, members of the project have spent considerable time discussing detailed style issues as well as reading each other's code. This has resulted in our adopting a uniform style by consensus, making it easier to read each other's code.

The machine-specific components of the system which must be changed during a port may be viewed as *interfaces* between system abstractions and machine hardware. The main interface is provided by the compiler which maps the machine-independent high-level language constructs into machine instructions. The code generation phase of the compiler is thus an important interface which must be changed during a port. Other interfaces are represented by assembly code or **twit** statements in the high-level language. Besides the compiler, there are three main interfaces.

The first interface is in the primitive operations of readying, blocking and preempting processes. The form of these primitives is machine-invariant, but a small amount of interface code must be written to load and store the volatile environments of processes as they acquire and relinquish the CPU. The abstraction of interrupts is a more complicated aspect of process preemption and activation; this abstraction is implemented by an assembly coded module called the *interrupt handler*.

A second interface is between the input/output system and the hardware interfaces. For character-oriented devices, such as teletypes, simple functions which "output a character and wait for an interrupt", or "wait for an interrupt and then input a character" are implemented using **twit** statements. A direct access secondary storage device is treated as an indexed sequence of fixed-size blocks which may be read or written randomly by referring to the index of the desired block. (The block size may vary from one device to another.) This abstraction is easily implemented in the device handlers including having several logical devices per physical device as in UNIX [13]. For purposes of program loading and swapping, the random access device handlers also support reading and writing of multiple contiguous blocks, which is supported directly by the hardware on many machines.

A third interface can be used when appropriate memory mapping hardware is available. This consists of a small number of functions for changing memory maps.

Implementing these interfaces is relatively straightforward because, except for the memory mapping interface, no design decisions need to be made. The interface functions have simple well-defined semantics. In most cases, the interface functions from a previous implementation for another machine can be used as *prototypes* in which only the machine-specific parts must be changed, and they serve as a model for the new implementation.

Interface code for the NOVA implementation includes 216 assembly language instructions and 201 **twit** instructions. For the TI 990 implementation we have 638 assembly language instructions and 334 **twit** instructions.

Thoth domain. It seems impractical to design system software to be portable over all computers. We have therefore restricted our attention to a subset of machines which we call the *Thoth domain.* The characterizing machine properties given below are based on assumptions made about the target machine in the base language and in the machine independent parts of the operating system. As such, a machine not in the Thoth domain could still be a target machine for Thoth. Porting to such a machine may require changes to machine independent code or tolerating some degree of inefficiency. That is, the characterization of the Thoth domain is not meant to exclude machines as much as to document assumptions.

A machine in the Thoth domain must allow a unit of storage called a *word,* which is some fixed number (one or more) of consecutive storage units totaling at least 16 bits. It must be possible to indivisibly access or modify the contents of a word (i.e. it must not be possible for an interrupt to occur during an access or modification). It should be possible to efficiently address consecutive words using consecutive integers, called word pointers. It must be possible to store any word pointer in a word. These concepts of words and word pointers are fundamental to the base language and are therefore necessary for any Thoth implementation.

Thoth does not use the concept of byte pointer because it does not appear to have an efficient implementation on some machines; several machines cannot store a byte pointer in a word (e.g. Honeywell Level 6, Rolm 1602, MODCOMP, Tandem 16). However, on these machines, a byte may be addressed as a non-negative offset relative to a word address. This is the abstraction used for accessing bytes in the base language.

Every machine in the Thoth domain has a single processor. This assumption is used to achieve relative indivisibility as defined in Section 2. Porting to a multiprocessor machine would require additional process synchronization; this seems feasible in some cases. However, our experience has been limited to single processor machines. Preferably the processor uses binary two's complement integer representation and arithmetic but relatively minor changes to the code would circumvent this restriction. It is assumed that the processor can be interrupted by the device interfaces, but that it is able to execute code to prevent interrupts from occurring (i.e. disable interrupts). This allows the implementation of the **enable** and **disable** statements of the base language and the interrupt abstraction described in Section 2. It is preferable that the processor be able to selectively prevent devices or groups of devices from interrupting and **enable** interrupts without negating any selective disabling.

The base language requires an efficient means of implementing a stack. Each function invocation uses a part of the process' stack, called a *stack frame,* which is used to store local variables, temporary results, and to receive arguments. Words and bytes in a stack frame may be referenced in an arbitrary order. Generally, a dedicated index register suffices for an efficient stack implementation. So-called "hardware stack" features found on a number of minicomputers are generally not appropriate for implementing this type of stack.

Porting Thoth. Porting the complete Thoth system to new hardware includes porting a compiler, assembler, loader, and library editor in addition to code which implements the system primitives discussed in Section 2.

An "assembler generating kit" is used to build an assembler for a specific target machine from a prototype assembler. In our experience, it takes roughly 10 man hours to produce the assembler this way (see [10]). The assembler produced is not particularly fast and lacks some features but it is adequate because the compiler generates relocatable load code directly and only a small part of the system is implemented in assembly code. Assembly code is used for initializing interrupt vectors, adapting the hardware interrupt structure to that of Thoth and implementing intrinsic functions called by compiled code. It has also been used to implement efficient versions of frequently used functions.

The relocating linking loader is machine invariant as well as machine independent; i.e. it requires no modification for use with a new target machine in the Thoth domain. Its design, discussed in [2], is based on a relocatable load code format suitable for machines in the Thoth domain. The load code is a sequence of *directives* which are executed by the loader to produce an executable module. The initial directives specify target machine parameters relevant to the loading process. Because the load code format is the same for all target machines, the library editor is also machine invariant.

The base language compiler consists of five phases which communicate via intermediate representations of source programs stored in files. Phases 1 and 2 do lexical and syntactic analysis. Phase 3 performs machine invariant global and local optimizations and modifies expression trees to facilitate code generation. Phases 4 and 5 do code generation. All phases are machine independent and the first 4 phases are machine invariant. Phase 5 requires substantial modification to generate code for a new machine. Converting the compiler to generate working code seems to take a few weeks although converting it to generate quality code may take up to several months depending on the complexity of the target machine. Most of this time is spent designing the stack and code bursts for expression evaluation which is one of the most time-consuming and challenging parts of porting Thoth.

After the compiler is ported, a simple version of the system can be running in a few days. Additional device drivers have been added to the system in times ranging from under one hour to two weeks.

Accurately quantifying the effort required to port the system is difficult because the system has been under development during the times we have ported it and the porting has been done by the designers/implementors. In general, the effort required is dependent on the knowledge and ability of the porters, the architecture, configuration and documentation of the target machine and the software support available for performing the port. Thus, although we feel confident that porting Thoth requires considerably less effort than implementing similar software from scratch, we refrain from stricter estimates of time requirements until more experience has been gained.

4. Performance of Thoth

Measurements of operating system performance are important to the design of application programs, particularly real-time applications. Such measurements are also useful for evaluating the efficiency of the system design and implementation.

Space performance. The user can tailor the configuration of Thoth to a particular application. The *configuration table* is a matrix of constants and function names. This table is used during system initialization to create system processes and data structures. Some of the entries in the configuration table determine what input/output devices will be supported. The basic Thoth configuration includes support for the following:

dynamic memory allocation
process creation
interprocess communication
input/output
$tty0

Optional entries in the configuration table cause functions to be loaded from the library to support

clock
process destruction
additional devices
file system
multiple teams and swapping

Thoth may be used in two different modes: multiple team and single team. In a multiple team system, the system code is loaded into primary memory before (i.e. separately from) any of the user teams. In a single team system, all system code is linked with the user code as a single core image. A configuration which allows multiple teams also provides the ability to load executable teams from files, swap teams when the primary memory resource becomes scarce, enforce an equitable sharing of the CPU resource among the teams, and reclaim resources from teams which terminate either normally or abnormally. For these reasons, any configuration which includes multiple teams must also include the file system, clock and process destruction options. If the system is configured to run a single user team "stand alone," there is no interdependence of the options; thus, the user's requirements alone dictate the options to be included.

The space requirements for a null user team are given in Table I; the figures are separated into requirements for code and data. Code includes instructions and external variables, while data includes team descriptors, process descriptors, stacks and buffers allocated during system initialization.

A substantial decrease in size can be realized by eliminating the input/output primitives and support for $tty0. The code size for a stripped version of Thoth can be reduced to approximately 2000 words for the Nova/2 and 2700 words for the TI 990. The stripped versions are indicative of the sizes possible for specialized control applications.

The difference in code size between the TI 990 and Nova results mainly from the fact that the base language is word-oriented and typeless. Since the TI 990 is byte-addressed, every use of a word pointer requires conversion to a byte address. These conversions are not required on the Nova because it is a word-addressed machine.

Time performance. Many real-time application programs are best structured as networks of processes passing data via interprocess communication primitives. Hence the speed of the communication primitives determines the maximum capacity of the network. Since Thoth itself relies on this structuring philosophy, the efficiency of these primitives is important to the efficiency of other aspects of the system.

Measurement of the time required for the communication primitives is accomplished with a Thoth team. The team's root process creates two processes which repeatedly call matching sets of communication functions. One of the processes counts the number of calls made and, after a fixed time has elapsed, the parent process divides this number into the elapsed time to obtain an estimate of the time per communication. The results of these measurements are presented in Table II. This meaurement technique includes all overhead for dispatching, handling clock interrupts, and some artifact due to loop control and counting.

Similar techniques obtain estimates of the time required to create, ready and destroy a process and the time required for byte I/O primitives (.Get and .Put). These times are also included in Table II. Input and output which is block, rather than byte, oriented can be done in a portable manner using facilities not described in Section 2. Such an approach is used by the file Copy utility; to copy a file of 10,000 bytes to a new file on the same disk drive requires 6 seconds on our TI 990/10 using a DS-25 moving head disk and 4 seconds on our Nova/2 with a fixed head disk. Most of this time is spent creating and opening the files. To copy a portion of a disk pack containing a 9.4 million-byte file system from one drive to another takes 116 seconds on our TI 990/10 using two T-25 disk drives.

Real-time response. One measure of the effectiveness of a real-time system is the maximum amount of time required to respond to an interrupt. For this reason, all algorithms in Thoth which require the disabling of interrupts have been designed to disable for an amount of time which can be bounded by a machine-dependent constant. Unfortunately, a rather sophisticated hardware monitor is required to measure this aspect of system performance. In the case of the Nova/2, however, the measurements are available through the use of a Fortran program designed to simulate execution of programs on the Nova/2 and gather detailed hardware level information about the program executed [14]. The timings from the simulator are considered comparable to those in Table II because the Nova simulator reproduces the behavior of the timing programs to within 3 percent of our Nova/2. This simulator proved to be of great use during the early stages of Thoth development, and tests are still run occasionally.

The observed results from programs run on the simulator are presented in Table III; the data represents worst-case disable times and their associated functions.

Table I. Sizes of Thoth configurations.

	Nova		TI 990	
	Code	Data	Code	Data
Basic configuration	3595*	1024	5778	1166
Clock	336	142	227	198
Process destruction	356	84	446	270
Memory I/O	127	0	312	0
File system	4743	415	5627	602
Each additional tty	24	152	25	721
Multiple teams	2828	†	4396	†

* All sizes are given in 16-bit words. The total space required by a given-configuration can be computed by adding the requirements of the options to those of the basic system. Note that since the second tty shares code with the first, it requires only the space for an entry in the configuration table and data for additional driver processes.

† The data size for multiple teams is decided when the system is generated; the data space required depends on the maximum number of user teams and processes and the number of devices supported by the system. Systems currently running use from 3600 to 6300 words.

Table II. Times in microseconds for Thoth primitives.

	Nova/2	TI 990/10
.Send/.Receive/.Reply	656	1862
.Forward	323	1025
.Create/.Ready/.Destroy	5000	22700
.Put	88	176
.Get	69	137

Table III. Worst case disable times in microseconds for the Nova/2. (The associated function name indicates where the disabled code begins.)

Disable time	System function
297	.Send
266	.Free
254	.Receive
187	clock processes
162	.Ready
111	.Alloc_vec
107	.Await_interrupt
58	.Create
54	interrupt supervisor
22	.Kill
17	.Reply

Of the functions presented in Table III, all but the interrupt handler are machine-independent. Since the interrupt handler and the machine-dependent functions called by the other functions all have a similar structure in both the Nova and TI 990 implementations, it is reasonable to estimate disable times for the TI 990 from the ratios of speeds indicated in Table II.

The simulator also provides response times to interrupts from each of the devices. *Response time* for a specific device is defined to be the elapsed real time from "device completion" until the CPU "services" the resulting interrupt. For the Nova computer, *device completion* occurs when the device sets its DONE flip-flop to 1, and the CPU *services* the interrupt by setting the device's DONE flip-flop to 0. The Nova is capable of selectively disabling devices for interrupts; this allows the use of multiple priority levels for interrupt handler processes. When a device's handler process is active, interrupts are selectively disabled for devices with handler processes of the same or lower priority; those with handler processes of higher priority may still interrupt and thereby preempt the lower priority process.

The best response time for the $tty0 output is 68 microseconds. The best response time for the real-time clock is also 68 microseconds. The worst cases occur when both devices complete just as a lower priority process enters the worst-case disabled section of code (see Table III). Since the real-time clock handler process (which executes the .Chronographer code given in Section 3) has a higher priority than the $tty0 output handler, the worst case time for the clock is 68 + 297 = 365 microseconds, assuming there are no other devices with a same or higher priority handler process. With this assumption the worst case time for the $tty0 output is 68 + 68 + 297 + $x > 433$ microseconds, where x is the time spend in .Chronographer from the STOP_CLOCK through the .Await_interrupt function call. Unfortunately, we have no convenient way of measuring the worst case value for x which would occur at the beginning of some year. This is consistent with the observed worst-case response times of 316 microseconds for the real-time clock and 594 microseconds for $tty0 output.

5. Concluding Remarks

The original objectives of developing Thoth were to investigate the feasibility of portable operating systems and to provide a tool for teaching real-time programming of minicomputers. To these ends, we feel Thoth has been highly successful.

As a research project, Thoth has demonstrated the feasibility of a portable operating system for a specified class of machines. Thoth has also facilitated writing machine-independent utility, communication and application programs. The approach of porting the entire system to a "bare machine" has many advantages but carries the penalty of requiring that other software either be discarded or ported to Thoth. Because Thoth currently supports only the one language, porting "foreign" software to Thoth involves a complete rewrite.

As a pedagogic tool, Thoth has been used at several levels. Undergraduates taking a course in real-time programming of minicomputers have written Thoth programs to control a model train set and to race model cars on a slot car track interfaced to a minicomputer. Because Thoth is written in a high-level language, students have been able to read the source code to gain insight into operating system structure. In this respect, portability and pedagogy seem complementary; exemplary and portable techniques both strive for general applicability. At the graduate level, Thoth has been studied and critically analyzed as part of a weekly seminar. It has been used for various graduate programming projects, and for several faculty members' research.

A commercial firm is applying Thoth to real-time control problems. Thoth is also proving to be valuable for our own minicomputer applications.

A major deficiency with the system currently is the language. Because it is untyped, there are no unsigned or double precision integers thus limiting the maximum positive integer on most machines to 32767. For example, files can easily be too big for .Where (see Section 2) to always return a meaningful value. Of more consequence is the lack of a pointer type. Because all pointers are word pointers, the use of a pointer on a byte-addressable machine involves conversion to a byte address on every use with the resulting penalty in speed and size of code. We plan to introduce types into the language to solve these problems. It is interesting to note that our desire for types in the language is based entirely on considerations of efficiency and portability over diverse machines.

Much remains to be done in the development of Thoth and the study of portable operating systems in general. We are currently working on several problems. First, more work is required on a portable abstraction for the use of memory management and protection hardware. The system running on the TI 990/10 uses the memory mapping hardware option but we have yet to use such hardware on other machines. Second, there is the continual effort to correct deficiencies and inefficiencies in the design of the system and extend its functionality. That is, we continue to address problems of developing an operating system in addition to the portability considerations. Finally, more experience with porting the system is required to fully evaluate the portability of Thoth. The system is currently being ported to the Honeywell Level 6 minicomputer and we plan to port to the PDP-11 later on.

Acknowledgments. We gratefully acknowledge the support given this project by the National Research Council of Canada. This research could not have been done without the facilities and environment of the University of Waterloo's Mathematics Faculty Computing Facility directed by Professor W. Morven Gentleman. A number of individuals have made notable contributions to Thoth and the Portable System Software Project. Included in this group are Mike Afheldt, Bert Bonkowski, Otmar Bochardt, Reinaldo Braga, Morven Gentleman, Sam Henning, Tom Miller, Alfedo Piquer, Patricio Poblete, Laurie Rapsey, Gary Stafford, Ian Telford, and Fred Young. We also thank John Corman for keeping our hardware running. We have drawn from many sources in the design of the Thoth Machine described in Section 2. The main influence has been other operating systems, particularly: Brinch-Hansen's RC 4000 system, Multics, Data General's RTOS, Honeywell's GCOS, and Bell Laboratories' UNIX.

A note on the name: in Egyptian mythc ogy [9], Thoth ruled Egypt for 3226 years. He was endowed with complete knowledge and wisdom, inventing all arts and sciences including arithmetic, geometry, astronomy, soothsaying, magic, medicine, drawing, and writing. In some stories, creation was accomplished by the sound of his voice. After his death, Thoth went to the skies where he became god of letters, god of wisdom, messenger for the gods, upholder of justice, and searcher after truth. He measured time, divided the world, kept divine archives, and was patron of history. When Egyptians died, Thoth weighed their hearts and proclaimed them "guilty" or "not guilty." He then revealed the magic formulae needed to traverse the underworld in safety.

Received August 1977

References

(Note. References [3] and [5] are not cited in the text.)

1. Braga, R.S.C. Eh ref. manual. Res. Rep. CS-76-45, Dept. of Computer Sci., U. of Waterloo, November 1976.
2. Braga, R.S.C., Malcolm, M.A., and Sager, G.R. A portable linking loader. Symp. on Trends and Applications 1976: MICRO and MINI Systems (an IEEE/NBS conf.), May 1976, pp. 124–128.
3. Brinch-Hansen, P. The nucleus of a multiprogramming system. *Comm. ACM 13*, 4 (April 1970), 238–241, 250.
4. Cox, G.W. Portability and adaptability in operating system design. Ph.D. Th., Purdue U., Indiana, 1975.
5. Feiertag, R.J., and Organick, E.I. The Multics input-output system. Proc. Third Symp. on Oper. Sys. Princ., Oct. 1971, pp. 35–41 (available from ACM, New York).
6. Johnson, S.C., and Kernighan, B.W. The programming language B. Bell Lab. Comput. Sci. Tech. Rep. No. 8, January 1973.
7. Johnson, S.C., and Ritchie, D.R. Personal communications, 1977.
8. Knuth, D.E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1973.
9. Lons, Veronica. *Egyptian Mythology.* The Hamlyn Pub. Group, Ltd., 1968.
10. Malcolm, M.A., and Stafford, G.J. The Thoth assembler writing kit. Res. Rep. CS-77-14, Dept. of Computer Sci., U. of Waterloo, October 1977.
11. Miller, R. UNIX—a portable operating system. Proc. of the Australian Universities Computer Sci. Seminar, Feb. 1978, pp. 23–25.
12. Richards, M. BCPL: a tool for compiler writing and system programming. Proc. Spring Joint Computer Conf., 1969, pp. 557–566.
13. Ritchie, D.M., and Thompson, K. The UNIX time sharing system. *Comm. ACM 17*, 7 (July 1974), 365–375.
14. Sager, G.R. Emulation for program measurement/debugging. In *Minicomputer Software*, J.R. Bell and C.G. Bell, Eds. North-Holland Pub. Co., Amsterdam, 1976, pp. 107–123.

Operating Systems

R. S. Gaines
Editor

Synchronization with Eventcounts and Sequencers

David P. Reed
Rajendra K. Kanodia
Massachusetts Institute of Technology

Synchronization of concurrent processes requires controlling the relative ordering of events in the processes. A new synchronization mechanism is proposed, using abstract objects called eventcounts and sequencers, that allows processes to control the ordering of events directly, rather than using mutual exclusion to protect manipulations of shared variables that control ordering of events. Direct control of ordering seems to simplify correctness arguments and also simplifies implementation in distributed systems. The mechanism is defined formally, and then several examples of its use are given. The relationship of the mechanism to protection mechanisms in the system is explained; in particular, eventcounts are shown to be applicable to situations where confinement of information matters. An implementation of eventcounts and sequencers in a system with shared memory is described.

Key Words and Phrases: process synchronization, interprocess communication, distributed systems, security models, mutual exclusion, semaphores

CR Categories: 4.30, 4.32

This research was performed in the Computer Systems Research Division of the M.I.T. Laboratory for Computer Science. It was sponsored in part by Honeywell Information Systems, Inc., and in part by the Air Force Information Systems Technology Applications Office (ISTAO), and by the Advanced Research Projects Agency (ARPA) of the Department of Defense under ARPA order 2641, which was monitored by ISTAO under Contract F19628-74-C-0193.

Authors' present addresses: D.P. Reed, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge MA 02139; R.K. Kanodia, Bell Laboratories, Holmdel, NJ 07733

A version of this paper was presented at the Sixth Symposium on Operating Systems Principles, West Lafayette, Indiana, November 16–18, 1977.

The remaining papers appear in Operating Systems Review (ACM SIGOPS newsletter), Vol. 11, No. 5 (Special Issue). This special issue is available prepaid from ACM, P.O. Box 12105, Church Street Station, New York, NY 10249; ACM or SIGOPS members $9.00, all others $12.00.