February 1983

Report No. STAN-CS-83-945
CSL Technical Report 206

Perseus: Retrospective on a Portable Operating System

by

Willy Zwaenepoel and Keith A. Lantz

Department of Computer Science

Stanford University
Stanford, CA 94305

Perseus: Retrospective on a Portable Operating System

Willy Zwaenepoel and Keith A. Lantz
Computer Systems Iaboratory
Departments of Computer Science and Electrical Engineering
Stanford University

Abstract

We describe the operating system Perscus, developed as part of a study into the issues of computer communications and their impact on operating system and programming language design. Perseus was designed to be portable by virtue of its kernel-based structure and its implementation in Pascal. In particular, machine-depcndcnt code is limited to the kernel and most operating systems functions are provided by server processes, running in user mode. Perseus was designed to evolve into a distributed operating system by virtue of its interprocess communication facilities, based on message-passing. This paper presents an overview of the system and gives an assessment of how far it satisfied its original goals. Specifically, we evaluate its intcrprocess communication facilities and kernel-based structure, followed by a discussion of portability. We close with a brief history of the project, pointing out major milestones and stumbling blocks along the way.

1. Introduction

In the spring of 1979 the Stanford computer science community was on the verge of acquiring a large set of new computer facilities. Until that time, the only available research machine of any note was a DecSystem-10 running a homegrown, one-of-a-kind operating system similar to TOPS-10 and known as WAITS. However, orders were outstanding for a DecSystem-2060 and two VAX-11/780's, and a grant of 20 Xerox Altos [22] together with a 3 Mbit experimental ethernet [15] was in the offing. Long-term plans called for an in-house workstation, which ultimately became known as the SUN workstation [2].

In order to investigate the issues involved in interconnecting these resources, as well to acquire additional resources, Stanford undertook with IBM a Joint Study on Distributed Computing. The purpose of the Joint Study was to investigate issues in computer communications and their impact on operating system and programming language design. In view of the hcterogcncity of machines then on order or anticipated, key attention was given to building a general-purpose operating system that would be portable over a range of target machines. The system would also support interprocess communication between machines at a higher level than that possible with traditional network architectures: that is, it would be a distributed operating system. The foundation for portability was the choice of high-level implementation language, Pascal* [9]. The foundation for distribution was the choice of message-passing as the principal means of interprocess communication.

This paper presents the result of the Joint Study, an operating system called Perseus. The Perscus system was implemented in native mode on an IBM 4331 and on a guest-level to DecSystem-20/TOPS-20 and VAX/Berkeley Unix. Unfortunately, the system did not survive (at Stanford) to fulfill its potential as a distributed operating system.

Section 2 presents an overview of the system. Sections 3 and 4 are critical reviews of the underlying interprocess communication mechanisms and overall system structure, respectively. Section 5 discusses the portability of Perseus and, related to that, the choice of implementation language. Section 6 concludes with a brief history of the project. Additional details may be found in the final report of the Joint Study [12].

2. An Overview of Perseus

Perseus was influenced primarily by Demos [1. 14, 16] and secondarily by Thoth [5, 6] and RIG [10, 13]. As such, it consists of a resident *kernel* and a set of processes. The kernel establishes a virtual machine for the processes to run on. It provides intcrprocess communication, basic process and memory management, and interfaces to the hardware at the interrupt and device register level. The kernel is the only component of the system to run in privileged mode.

All other services arc provided through *server processes*, executing in user mode. Server processes provide process management, memory managcmcnt, file storage and retrieval, time service, name service, device handling, command language interpretation and network access. Applications access the system through *user processes*. which arc, from the kcrncl's viewpoint, in no way different from server processes.

2.1. Interprocess Communication

The interproccss communication mechanism in Perseus is a very close descendant of the one in Demos. In general, one can say that the system exhibits a vast amount of functionality at the kernel level, as opposed to at the process level in systems such as Thoth.

The principal protected objects in the system arc the communication paths over which messages can bc sent. These

This work was supported by the IBM Joint Study on Distributed Computing under contracts SEL 47-79, SEL 2-80, SEL 8-81, and SEL 18-81. This paper has been submitted to *Software - Practice and Experience.*

paths arc called *links* and arc similar to the notion of ***ports*** in systems such as Accent [18,19].[1] A link has one receiver and one sender. The sender may change over time. A link is maintained by the kernel independent of any process.

In order for process ***A*** to send a message to process ***B***, A must possess a link to ***B***. This link must previously have been created by Rand sent (in a message) to A. A sender names a link through a process-local *link-id,* which is then mapped by the kernel into a system-global identifier for the link. Thus, a link-id is similar to a capability. At the time it is created, a process is given access to a set of *standard* links associated with various servers.

Each link is associated with a *channel* and a *code*. Multiple links may belong to the same channel and/or have the same code. Channels are typically used to represent a particular type of request: codes are used to distinguish particular senders. The receive operations are performed on a set of channels. When a message is received, the receiver may use the code to determine the sender.

Perseus provides both synchronous and asynchronous message communication. Four basic primitives are available for sending and receiving messages: blocking ***Send*** and ***Receive,*** and non-blocking ***ASend*** and ***AReceive***. Additional primitives exist for creating and destroying links and for obtaining status information about them.

Since messages are small and fixed size, there is an additional facility for bulk data transfer. The data transfer facility is protected through the use of links. Process ***A*** can send process ***B*** a link, containing a pointer to and the size of an area in if's address space. A can also specify whether he grants read or write access to process ***B.***

To clarify the use use of links, consider a standard scenario involving a client and the file system. Assume that a client wants to read a file. The client first sends a message over its standard link to the file system, asking the file system to open the file. Implicit in this message is a *rep&-link,* ***R,*** owned by the client, on which the file system will reply. If the file system accepts the request, it will ask the kernel to create a new link, ***F***, pointing to itself, and it will reply to the client on ***R,*** returning the newly created link ***F***. ***F*** now represents the open file in the client's world. It is used to send a message with a read request to the file system. The client passes in the message with the read request a link, ***D***, pointing to the area in its address space where it wants the file stored. The file system uses the data transfer facility over link ***D*** to transfer the file to that particular area and, finally, replies to the client on ***R.*** The client typically closes the file by deleting link ***F***; the file system will be notified of the destruction of the link and-close down the connection.

2.2. The File System

The basic unit of information with respect to the file system is a byte. To the system, every file appears as a continuous stream of bytes. with length greater than or equal to zero, no structuring whatsoever is expected. Some user programs may generate or expect files in a particular format but this is left entirely under the control of those programs. The system provides operations for creating and deleting files and arbitrary length reads and writes on files.

Directories provide the mapping between file names and files. They arc considered to bc objects owned by the file system. User access is therefore restricted through the use of special primitives and cannot be done through the normal read/write operations. The directory structure forms a rooted tree similar to the Unix directory structure [20].

The file system is implemented as a set of four cooperating processes:

1. the directory processor,
2. the basic file server,
3. the buffer manager, and
4. the disk driver.

The directory processor is responsible for maintaining working directories and for translating pathnames into the file system's internal representation of files, called file identifiers. The basic file server maintains all active files in the system. It translates read and write requests into logical blocks to be read or written, requests those blocks from the buffer manager and moves the data from (to) the client's address space to (from) the buffer manager. The buffer manager attempts to keep a pool of frequently used file blocks in memory. When the need for free buffers arises, it flushes a number of buffers out to the disk, based on an LRU strategy. Finally, the disk driver translates logical block numbers into disk addresses, schedules the requests for the disk and maintains the file descriptors and the extent trees on the disk. It interfaces to the disk through a small number of assembly routines in the kernel that implement single and multiple block transfer at the interrupt and the device vector level.

2.3. The Network Software

The goal of the networking project was to make the 4331 accessible from Stanford's ethernet, primarily as a file server. The choice of protocols was dictated by the fact that all existing machines on the ethernet used the PUP protocols [3] at the time the project began.

The 4331 is connected to the ethernet via several pieces of hardware. The ethernet transceiver is interfaced to a Multibus connected to a modified I/O port on an Series/1 minicomputer. The Series/1 in turn is connected to the 4331 via a channel attach unit.

Level 0 of the PUP hierarchy consists of the hardware from the ethernet up to and including the Series/1. The Series/1 has some software to buffer packets for performance reasons. The remaining levels of the architecture are implemented under Perseus. Level 1 and level 2 are each implemented by one process, the PUP process and the Rtp/Bsp process. An FtpServer and a TelnetServer process provide file access and virtual terminal service, respectively, for clients on remote hosts. Local clients requesting remote resources interface to the network through FtpUser and TelnetUser processes, respectively. There is some level 0 software in the kernel for handling communication between the PUP process and the Series/1.

2.4. The Switchboard

The switchboard allows a client to communicate with an arbitrary server, to which it does not have a standard link. This notion is identical to that of a *name server* in other systems. Essentially, a server registers with the switchboard by sending it a link pointing to itself and the name of the service that the server provides. A client can query the switchboard's database of servers and, on a successful query, set up a connection with a server. The switchboard allows the system to be configured in a dynamic fashion: servers can go down and come back up and re-register with the switchboard.

There may be one or more server processes for a given service, depending on the service *class*. The three classes of service are:

Static There is always exactly one server for a ***Static*** service. The switchboard binds the name of the service to a link to the server.

[1] Accent is a descendant of RIG, with changes inspired primarily by Demos and Multics.

Auto There is at most one server for an ***Auto*** service, but if the service has never been requested no server exists. The switchboard binds the name of the service to:

1. the pathnamc of the file from which the server should be started when the first request for it is rcceivcd; and
2. a link to the server, if it has already been created.

Private There is one server process per request for a ***Private*** service. The switchboard binds the name of the service to the pathnamc of the file from which the server should be started. The switchboard starts a new server process for every request it receives: therefore, it does not store the binding from name to link.

Note that these service classes are completely invisible to the client process which requests a service; all the client sees is a link to a server. Only the process which registers the service with the switchboard is concerned with the service class.

Client processes may send two kinds of requests to the switchboard:

Locate Find a link to the requested service and returns that link to the requester.

Open Find a link to the requested service and forward the ***Open*** request over that link. This is more convenient for the client than using ***Locate*** followed by an ***Open*** to the server, but it is not an essential function of the switchboard.

It was once contemplated to remove all standard links from client processes except for the link to the switchboard [8]. All connections would then be set up dynamically at runtime. Although this idea was never implemented in Pcrscus, it has since been incorporated in a much more general design for a universal directory service [11].

2.5. The Process Manager

The process manager is responsible for process creation, loading, destruction, suspension and resumption. Upon process creation, the parent (creator) receives a link to the process manager, on which it sends requests related to that particular child. This link is referred to *as the control link* for the child.

A principal function of the control link is for debugging. That is, the parent can request that a child be suspended, resumed, or destroyed as desired. The control link can be passed to another process, so the original creator need not remain the parent.

2.6. The Memory Manager and the Swapper

Memory management was implemented for the 4331, which supports multiple address spaces and demand paging. Memory management is performed by the kernel and two server processes, the mcmon, manager and the swapper. The kernel handles all parts that directly interface the hardware. It handles the page and segment tables nccdcd for virtual address translation, reads the change and rcfercncc bits, fields the address fault traps, and is responsible for locking into memory the pages needed for I/O.

The swapper moves page between disk and memory. It acts only on request from the memory manager.

The memory manager contains the algorithms for memory scheduling. The process manager notifies the memory manager when processes have to be loaded or when processes are suspended or destroyed. The mcmoty manager also receives periodic notifications from the time scrvcr in order to do periodic memory rescheduling.

2.7. The Time Server

The time server maintains the current time of day and accepts requests from other processes to be notified after a certain period of time. It performs the machine-indcpcndcnt aspects of timing, while the kernel writes the so called timer block (on the 4331) and accepts the timer interrupts.

2.8. The Command Language Interpreter

The command language interpreter forms the interface of the user to the system. It is similar to the Unix shell [4]. The Pcrscus command language is position-oriented. Essentially, the command language interpreter takes a command name from the input line and checks whether it is an executable object file or a command file (a text file containing a sequence of commands). If the command name is an executable file, the command interpreter starts a process running the designated program. Otherwise, the command interpreter spawns a new command interpreter, which reads its input from the command file and initiates other processes as necessary. Facilities for sending parameters to a process, for redirecting standard input and output and for supporting search lists were also implemented.

3. Interprocess Communication

Since the area of inter-process communication is of continuing interest, we will now devote some special attention to the subject. We perceive three predominant characteristics of the Pcrscus interprocess communication system:

- The use of links which introduces both protection and the notion of a tight connection at the IPC level.
- The provision for both synchronous and asynchronous communication.
- The distinction between small, fixed-size messages and bulk data transfer.

We discuss each of these issues in turn, in comparison to other systems and in the perspective of distributed communication.

3.1. Links

The use of links is motivated primarily by a desire to achieve a certain amount of protection. Since link-ids are capabilities, they cannot be easily fabricated. This helps prevent either buggy or malevolent processes from gaining fraudulent access to resources. In addition, the kernel access lists allow the system to know who is talking with whom and to provide event-driven notifications to processes when links pointing to them are destroyed. However, there are a number of problems with links, which we now consider.

Links arc an example of a connection-based communication system. The client first requests the scrvcr to open a connection, the server validates the rcqucst. allocates a state record for this connection and returns a unique identifier (a link-id) to the client process by which it can access the connection. Connections have a number of advantages similar to those attributed to virtual circuits. Nevertheless, wc observe three problems inherent in connection-based systems:

1. The overhead for setting up and dismantling a link is substantial. In its simplest form, it requires two kernel calls for creating and destroying the link, two extra messages for opening the connection (the open request and the reply) and one extra message for notifying the server that the client has closed the connection.
2. Connection-based systems do not recognize the fact that most client-server interactions arc of a single-shot nature,

meaning that the client usually wants a single request satisfied and is not interested in the connection afterwards. In particular, it was observed in Perscus that client-server interaction had an almost constant pattern of setting up a connection, satisfying a request and breaking down the connection. This made the overhead just discussed particularly undesirable.

3. It is not clear how well the tight connection IPC mechanisms map into a distributed world with loosely connected kernels on different machines. As a single machine implementation, the Perscus kernel takes very explicit advantage of the fact that all the information relevant to a particular connection is available in the local kernel. Distribution of this knowledge substantially complicates the algorithms involved. A simple but important example of the problems involved concerns the message sent to the server when a link pointing to it is destroyed. This message allows the server to free the resources allocated to that connection. Message transfer across a network is not entirely reliable and therefore the message might never arrive at the server, if it lives on a different host as the client. So, the server cannot entirely rely on this notification to deallocate its state record and will probably have to use some other higher level mechanism as well (e.g. timeouts). Extrapolating this argument, we came to wonder whether it would not be better to rely entirely on this higher level mechanism and get rid of the explicit notification at the IPC level. This was the approach taken in RIG.

The use of connections was dictated primarily by the decision to provide capability-like protection. The use of link-ids is notably different from systems like RIG or Thoth in which all communication ports are global identifiers. In RIG, the use of global, location-dependent identifiers was reported to be a major headache, and, indeed, was a principal motivation for the integration of ports in to Accent [18, 19]. No such problems were reported in Thoth, largely because Thoth supported an extra level of low-level naming. location-independent *logical* process identifiers. There are two issues involved here, neither of which have, in our opinion, been given a substantial treatment:

1. Is there a substantial gain in building protection in at the message level or would be it be better to provide that protection on the process level, based on end-to-end type arguments [21]? Not all checking can be done at the message level. because in general, the message level does not possess enough knowledge to do so. Therefore, the receiving process will, even in a protected system, still have to do some checking of its own. So one can wonder whether there is any merit in doing checking at the message level at all. since the effort probably will have to be (at least partially) duplicated at the process level.
2. What cost is involved in using local identifiers? What percent of the time does the kernel spend in mapping identifiers from one level to another? One must bear in mind that interprocess communication is a heavily-used utility and therefore it should be made as efficient as possible.

Accent may help answer some of these questions.

Assuming that protected communications paths (connections) are a good idea, a third potential problem with links is that they are tightly coupled with processes. That is, the only entity that can receive a message from a link is a process; in particular, communication with the kernel is via system calls distinct from the IPC machinery. This may be warranted under the argument that services with substantially different semantics and performance characteristics should be accessed with substantially different mechanisms.

In Accent, on the other hand, ports and processes are totally independent. It is the port that represents the service, not the process. Thus, a port can be used to specify kernel objects, such as interrupt handlers, or an operating system could implement ports without implementing processes. This scheme eliminates the need for virtually all system calls other than those for sending and receiving messages. Moreover, the separation of communication and function allows the service to migrate without clients being aware of the migration, and it allows for the explicit management of communication between two parties by an intermediary (such as a debugger or performance monitor).

Perscus also introduced asymmetry in the way that receivers and senders refer to links. That is, a sender refers to a link via a link-id, whereas the receiver refers to a link via a channel and a code. Accent, on the other hand, provides a symmetrical treatment of its ports. Both sender and receiver refer to a port through a port-id. This symmetry is supported by the fact that both sender and receiver have access rights to the port and, therefore, both have capabilities.

3.2. Asynchronous vs. Synchronous Communication

It is instructive to observe the typical usage pattern of the Perscus primitives. The non-blocking receive operation was never used. User processes almost invariably used blocking sends and then waited for a reply from the server they addressed. Server processes used asynchronous sends and synchronous receives.

There does not seem to be much need for asynchrony in user processes since they most often cannot proceed without the current request being fulfilled. A compiler, for instance, cannot proceed without its request to the file system to read in the source file being finished. The situation is somewhat different for server processes: the notion here is that the *service* should at all times be available. It is clear that this notion cannot be supported with a completely synchronous communication mechanism (i.e. a mechanism which has nothing else than blocking send and receives), since the server would invariably become blocked on the first request. Permanent availability of the service can be achieved in one of two ways [10]:

1. The service is performed by a single process, able to multiplex itself between different requests. This solution demands multiple addresses per process (typically one per outstanding request) and built-in asynchrony in the message system, so the server never blocks on a single request.
2. The second solution has the service performed by a variable number of processes (one per request). A single address per process is sufficient in this scheme and the processes can communicate on a synchronous basis. There is typically a supervising process, that receives the user requests, starts up a new server process and forwards the request to that process without blocking itself.

A number of systems of both flavors have been built, but no conclusive evidence seems to be available for either case. Synchronous communication makes programming easier because of its more familiar semantics, not unlike procedure calls. It incurs a penalty though increased message traffic, a higher number of processes in order to gain concurrency, and, consequently. a higher number of context switches. These disadvantages have to be weighed against the machinery needed in the kernel to support multiplexed processes and the difficulty of writing multiplexed processes (comparable to writing the necessary kernel machinery!). This machinery includes, as mentioned. the support of multiple addresses per process, a facility for mapping these addresses into entry points in the process and the support of asynchronous communication, which complicates the buffering strategy in the kernel.

Characteristics of the particular hardware at hand heavily influence the relative merits of either solution. On machines where process creation and process switching is relatively

expensive, the one-process-per-server model is likely to come out ahead in performance.

3.3. Separate Message and Data Transfer Facility

Small, fixed-size messages enormously alleviate the buffering problems in the kernel, especially in the case of asynchronous communication. The strategy also ties in well with an often observed usage pattern: A vast amount of communication is transfer of control information. Usually the amount of data in these messages is small and its contents are explicitly interpreted by the receiving process. On the other hand, there is occasional bulk transfer of untyped and uninterpreted data, as in data transfer to and from secondary storage. Using a separate facility eliminates the performance disadvantage of copying and buffering large messages. Although this distinction is usually rejected by message purists, advocating variable size messages and no data transfer facility, it was never felt to be a major restriction in the system.

It is not clear, however, whether the distinction between message and bulk data transfer maps very naturally in a distributed setting. We are currently investigating this issue in the context of the distributed operating system V [7]. Moreover, some systems have been built that eliminate the need to be concerned about message size. Accent, for example. treats messages as just another segment in virtual memory. They may be mapped directly from the sender's address space to the receiver's address space using copy-on-write page mapping [19].

4. System Structure

The distribution of system functions was based on an almost fanatic desire to keep the kernel minimal. Most process management functions, for example, are performed by the process manager. Although these functions can logically be done outside the kernel, it seems that doing so constitutes an undue duplication of effort and a major cause of inefficiency. Invariably, a process would call the process manager, the process manager would transmit the information almost literally to the kernel, the kernel then replied to the process manager and finally the process manager replied to the original requester. 'This caused two extra messages and two extra context switches for every process manipulation, with no apparent gain in functionality.

Similar decisions were taken for memory management and device handling. Manipulation of the memory map was done by the kernel but memory allocation and rescheduling was done by another server process, the memory manager., With respect to device handling, all the kernel did was field the interrupts, transform them into a message and send it off to the device handling process. In the reverse direction, the kernel would accept a message from the controlling process, write the device register and start the device.

While in the case of process management, the separation of function between kernel and process manager seems to have been a mistake. this conclusion is not so obvious in the case of device handling and memory management. In a large system, device handling code can become very extensive, and even in systems where the kernel does not run with all interrupts disabled. there are still good reasons for keeping the kernel small and manageable. I lowever, one pays an inevitable penalty in repeated context switches between the kernel and the device handlers.

A potential solution to this problem was implemented in Thoth where the so called system team ran in privileged mode and in the address space of the kernel. 'l'his obviates the need for doing frequent context switches. llowever, it blurs somewhat the distinction between kernel and non-kernel operation:

By keeping the kernel small it was possible to run it with interrupts disabled. In so doing, the kernel becomes a sequential program, easier to understand and more likely to be correct. On the other hand, this decision severely complicates a number of issues related to demand paging: The kernel has to be very careful not to step on a page that is not in memory. It raises problems when the processor is idle: The idle state is, in general. terminated by an interrupt which has to be processed in the kernel. However, one cannot wait for this interrupt in the kernel since all interrupts are masked. So control has to be transferred to a (non-kernel) idle process only to be returned to the kernel as soon as the interrupt occurs. Thus, we have become convinced that, while it is essential to keep the kernel small and simple, running the kernel with all interrupts disabled, is in general not a viable solution.

5. Portability

We have pointed out that due to the large variety of computers that were under consideration at the beginning of the Perseus project, we were strongly motivated to build a system that could be ported to different machines. Two critical issues are involved in making an operating system portable:

1. The logical design must be represented in such a way that it can readily be ported from one machine to another.
2. The logical design of the system must be efficiently implementable on different machines.

Addressing the second issue first, the system structure described in the previous section, with the system built out of a kernel and a set of system processes, proved to be very valuable. With respect to the first issue, in practice this requires the design to be represented in a suitable high-level language so that the representation can be mapped automatically onto different machines. In the remainder of this section we review the logical design of the system with respect to portability and we discuss the choice of Pascal as the implementation language. Practical experience with porting the system is presented throughout the section.

5.1. Portability of the Logical Design

The portability of the logical design of Perseus derives from its construction in terms of a kernel and a set of system processes. The kernel abstracts to a large degree the idiosyncrasies of a particular machine into a well-defined virtual machine in which the processes operate. The objects available at the interface of this virtual machine (processes, messages, generic devices) are essentially machine-independent, and therefore the processes executing in this virtual machine tend to be machine-independent. Messages are advantageous in this respect since they do not depend on the existence of shared memory segments, a facility not easy to implement in some memory architectures.

Some hardware characteristics of peripheral devices cannot be ignored totally. We attempted to mask these dependencies as much as possible by parametrizing certain processes. For example, in the case of the file system it is clear that the size of the disk blocks is of paramount importance to the proper implementation of the system. All processes that made up the file system were parametrized in terms of this number. Parametrization was also used to tune the system to certain configuration, e.g. a parameter was available in the file system buffer manager to denote the maximum number of I/O buffers to be kept in core at any given time.

More pervasive dependencies occur, which cannot be adequately parametrized. A sophisticated disk driver requires certain scheduling algorithms in order to minimize the arm movement. This is clearly very dependent on the particular disk

and the dependency cannot be captured in a single parameter but rather pervades the entire algorithm. A similar problem occurs in the memory manager: Different algorithms need to be used to take advantage of different memory management schemes. Nevertheless, -with the exception of the disk driver and the memory manager, we found that porting the server processes (process manager, switchboard, time server, network software and command language interpreter) was relatively straightforward.

Although the kernel of an operating system might, at first sight, be considered the epitomy of a machine-dependent program, this is not necessarily true. The kernel logically consists of two levels:

1. A device level which interfaces the bare hardware (device interrupts, processor traps, memory management hardware).
2. The virtual machine interface.

While the device level is necessarily machine-dependent, the next level need not be if a strict separation between the two is maintained. One cannot totally localize all machine-dependent information to the device level, but again the technique of parametrizing the kernel showed excellent results. Parameters are needed, among others, for the size of the volatile process state to be saved and for certain constants describing the size of address translation tables and address spaces. The interface between the two levels cannot be made entirely machine-independent. In particular, we found it necessary to impose the following requirements on the machine architecture in order for the higher level to work correctly:

- There needs to be an efficiently addressable memory unit, commonly called a *word*, in which both a memory address and an integer can be stored.
- The possibility must exist to turn machine interrupts off.
- The volatile storage of a process must be easily accessible.

In porting the system from the DecSystem-20 to the 4331, no major problems occurred. However, the transition from a single address space to multiple address spaces on the IBM required several changes to the kernel. The original kernel essentially made the assumption that it could address user memory as part of its own. In a multiple address space world, user pointers first had to be translated into addresses meaningful to the kernel. This change was simple but pervasive: It turned that there were a substantial number of such memory references in the kernel, much more than logically necessary.

5.2. Language Issues

One of the key decisions, to be taken early in a project like this one, is the choice of implementation language. At the time the project began, our two principal choices were Model and Pascal. Demos was written in Model, so the first option was to retarget the Model compiler, port Demos, and work from there. This route was taken by Mike Powell in developing a multi-processor descendant of Demos [17]. However, the Model language was unfamiliar to most people at Stanford and neither the Model compiler nor the Demos system were written with portability in mind. Substantial effort would have been required to overcome these deficiencies.

At about the same time, the Pascal* [9] project got underway at Stanford. Pascal* is an upward-compatible extension to standard Pascal with several features added for facilitating systems programming, including parametric types, exception handling, and separate compilation. The Pascal* compiler would be explicitly designed to be portable, using a fairly machine-independent front-end and a retargetable code generator. In addition, we had a Pascal compiler available, running on and producing code for our original machine, and accompanied by some reasonable program development tools. Thus, we decided to start programming in Pascal and, as soon as the Pascal* compiler would become available, we would upgrade our code. Pascal* being upward compatible with Pascal, it was believed that this could be done without major retrofits.

It is not our intention here to debate at length the various strengths and weaknesses of Pascal. We restrict ourselves to the context of writing operating systems. Whereas the idea of writing an operating system in a high level language has become commonplace, there is still a lot of debate as to how high-level the language should be and both commercial and research development efforts are underway in either direction.

Pascal attempts to achieve high levels of protection and machine independence and therefore has to impose some restrictions on what can be expressed in the language. Certain constructs, necessary in systems programming (mainly access to memory address and bits within a word, the ability to locate code and data at specific locations in memory and dynamic storage allocation and deallocation), fall outside of Pascal's scope and therefore have to be programmed in assembly language. A less restrictive language might allow these constructs to be expressed in the language itself, but in return it would not offer comparably reduced levels of protection and machine-independence. Conversely, Pascal's protected world requires a little more of the code to be written in assembly language, but in return, makes for a much cleaner separation between machine dependent and machine independent code.

We feel that, in view of the very small part of the system that actually requires the kind of constructs Pascal does not supply, this clean separation is to be preferred over the disadvantage of having to write a little bit more of assembly code. An interesting compromise was explored by the implementors of Perseus on VAX/Berkeley Unix. They choose to leave most of the kernel code in its original Pascal implementation but redid some of the device-level code in C. This was possible since the languages could call each other. The general impression of this approach was very favorable, maintaining both the protection and machine-independence of Pascal wherever possible, while making the device level easier to read and to maintain.

Having argued in favor of Pascal for writing an operating system, is Pascal a portable language? The constructs in the language are, with the exception of variant records, machine-independent. However, poor specification has often led to implementations that allow non-portable constructs. the size of set's being the classic example. Also, lack of expressive power in certain areas has led to local extensions and, in turn, to a proliferation of Pascal variants. Our experience is that the differences between Pascal implementations are usually easily resolved. In porting the operating system from the DecSystem-20 to the 4331 and later to the VAX, the language system was not ported with the operating system; each time we had to interface to a new compiler. It turned out that the changes required to the code were extremely small and usually lexical in nature.

In retrospect, we feel that the choice of Pascal was justified, especially in view of the projected upgrade to Pascal★. Unfortunately, the lack of availability of a Pascal* compiler for the IBM and the MC68000 proved one of the major stumbling stones for the project.

6. The Rise and Fall

As noted in the introduction, Stanford was computer poor at the time the project began. 'Thus, a principal motivation was the acquisition of additional facilities through IBM. Given the type of IBM computers available and the type of computers available from other sources. Perseus was originally targeted for a wide range of micro- and minicomputers. These machines originally included the Series/1, the HP-300 and the Xerox Alto, and grew

to include the 4331 and the Motorola MC68000. The hope was that the rctargctublc Pascal* compiler would alleviate most of the problems associated with bringing the system up on so many machines.

Ironically, thc first working version of the kernel was not implemented on a micro or mini, but on a DecSystem-2060. It used one TOPS-20 process for each Perscus process and emulated the Pcrseus intercommunication facilities through TOPS-20 messages. This first kernel only supported interprocess communication. Soon thereafter, a second version was brought up capable of dynamic process management.

In early 1980, a 4331 appeared and became the major target for the project. The kernel was ported and memory management was added, allowing processes to run in multiple address spaces. In late 1980, priority was placed on connecting the machine to the ethernet and using it as a file server for the Stanford University Network (SUN). Consequently, the PUP network protocols [3] were implemented to run under Pcrseus and a file storage system was dev eloped. A Series/1 was used as a network frontend.

In early 1981, the MC68000-based SUN workstation [2] was targeted as the machine of choice for the future, and hopes for the 4331 and Series/1 began to wane. During the bummer of 1981, Perseus was implemented at a guest-level on top of VAX/Berkeley Unix. Ironically, its m&age-passing capabilities were simulated with Accent IPC [18]. This effort signaled the end of active Pcrseus development.

Perseus came to an end due to a combination of hardware and software problems, disruptive changes in personnel, and differences of opinion at several levels. In terms of hardware, the machine of choice, the SUN workstation, never became available in the time frame of the project. Not only does the SUN perform better than the 4331, but it is trivial to connect to an ethernet. As mentioned above, a Series/1 frontend was required to connect the 4331 to the ethernet.

In terms of software, the development tools available on the 4331 were quite different from those with which the implementors were familiar. The result was that much of the development was carried out on other machines, with the resulting overhead of maintaining consistent versions of the code on multiple machines. More important, a Pascal-k code generator for the 4331 never materialized and the code generator for the SUN workstation did not materialize in time.

As for project management, in summer 1980, the original principal investigator left Stanford. One of the authors (Lantz) took over, but a lack of familiarity with the project and fundamental disagreements with some of the ideas resulted in a slowdown. Ultimately, policy differences between IBM and Stanford mitigated against continuation of the Joint Study and the project was terminated by mutual consent.

7. Concluding Remarks

The Pcrseus project was under way from 1979 through 1981. In that period of time the system was implemented both in native mode and as a guest-level operating system. We have described the logical design and the irnplcmcntation of the system. We have outlined its successes and its failures, in terms of interprocess communication, system structure, and portability. Although Pcrscus itself has been abandoned, all of these issues retain our active interest.

Acknowledgments

The authors wish to thank the many people that were involved in the Perscus project, including Forest Baskett, John Burnett, Tim Gonsalves, Thomas Gross, Mike Kenniston, Steve Kille, Guillcrmo Loyola, Noah Mendelsohn, and Marvin Theimer. We thank the IBM Palo Alto Scientific Ccntcr for supporting this research, in particular Horace Flatt, Paul Friedl, and Patrick Smith.

References

1. F. Baskett, J.H. Howard, and J.T. Montague. Task communication in DEMOS. Proc. 6th Symposium on Operating Systems Principles, ACM, November, 1977, pp. 23-32. Published *as SIGOPS Operating Systems Review* 11(5).

2. A. Bechtolsheim, F. Baskett, and V. Pratt. The SUN workstation architecture. Tech. Rept. 229, Computer Systems Laboratory, Departments of Computer Science and Electrical Engineering, Stanford University, March, 1982.

3. D.R. Boggs, J.F. Shoch, E.A. Taft, and R.M. Metcalfe. "Pup: An internctwork architecture." *IEEE Transactions on Communications COM-28*, 4 (April 1980), 612-624.

4. S.R. Bourne. "The UNIX Shell." *The Bell System Technical Journal 57*, 6 (July/August 1978), 1971-1990.

5. D.R. Cheriton. *The Thoth System: Multi-Process Structuring and Portability*. North-Holland/Elsevicr, 1982.

6. D.R. Cheriton, M.A. Malcolm, L.S. Melen, and G.R. Sager. "Thoth, a portable real-time operating system." *Comm. ACM 22*, 2 (February 1979), 105-115.

7. D.R. Cheriton and W. Zwaencpoel. The V distributed kernel and its performance. Submitted to the 9th Symposium on Operating Systems Principles, ACM, October 1983.

8. T.R. Gross, K.A. Lantz, K. Marzullo and W. Zwaenepoel. Uniform access to resources in Pcrseus. Computer Systems Laboratory, Departments of Computer Scicncc and Electrical Engineering, Stanford University, June, 1981.

9. J.L. Hennessy. Pascal*. Computer Systems Laboratory, Departments of Computer Science and Electrical Engineering, Stanford University, 1980.

10. K.A. Iantz. *Uniform Interfaces for Distributed Systems.* Ph.D. Th., University of Rochester, 1980. TR63, Department of Compu tcr Science.

11. K.A. Lantz, D.R. Chcriton, G.J. Popek and C.S. Kline. A universal directory service. To be submitted to *ACM Transactions on Computer Systems.*

12. K.A. Iantz, et al. Perseus: Final Report of the IBM Joint Study on Distributed Computing. In preparation.

13. K.A. Lantz, K.D. Gradischnig, J.A. Feldman, and R.F. Rashid. "Rochester's Intelligent Gateway." *Computer 15*, 10 (October 1982), 54-68.

14. DEMOS User's Reference Manual. Los Alamos Scientific Laboratory, 1979.

15. K.M. Metcaife and D.R. Boggs. "Ethernet: Distributed packet switching for local computer networks." *Comm. ACM 19*, 7 (July 1976), 395-404. Also CSL-75-7, Xerox Palo Alto Research Center, reprinted in CSL-80-2.

16. M.L. Powell. The DEMOS file system. Proc. 6th Symposium on Operating Systems Principles, ACM, November, 1977, pp. 33-42. Published *as SIGOPS Operating Systems Review* 11(5).

17. M. Powell. Portable operating system constructs. Department of Electrical Engineering and Computer Science, University of California - Berkeley, April, 1981.

18. R.F. Rashid. An inter-process communication facility for UNIX. Tech. Rept. CMU-CS-80-124, Department of Computer Science, Carnegie-Mellon University, March, 1980.

19. R.F. Rashid and G.G. Robertson. Accent: A communication oriented network operating system kernel. Proc. 8th Symposium on Operating Systems Principles, ACM, Dcccmber, 1981, pp. 64-75. Published *as SIGOPS Operating Systems Review* 15(5).

20. D.M. Ritchie and K. Thompson. "The UNIX timesharing system." *Bell System Technical Journal 57*, 6 (July/August 1978), 1905-1929.

21. J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. Proc. 2nd International Conference on Distributed Computing Systems, INRIA/LRI, April. 1981, pp. 509-512.

22. C.P. Thackcr, E.M. McCreight, B. W. Lampson, R.F. Sproull. and D.R. Boggs. Alto: A personal computer. In *Computer Structures: Principles and Examples*, D.P. Siewiorck, C.G. Bell, and A. Ncwcll, Ed., McGraw-Hill, 1982, pp. 549-572.