An Open Operating System for a Single-User Machine[1,2]

Butler W. Lampson

Xerox Research Center
Palo Alto, California 94304

Robert F. Sproull

Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

The file system and modularization of a single-user operating system are described. The main points of interest are the openness of the system, which establishes no sharp boundary between itself and the user’s programs, and the techniques used to make the system robust,

1. Introduction

In the last few years a certain way of thinking about operating systems has come to be widely accepted. According to this view, the function of an operating system is to provide a kind of womb (or, if you like, a virtual machine) within which the user or her program can live and develop, safely insulated from the harsh realities of the outside world [2, 5, 13]. One of the authors, in fact, was an early advocate of such “closed” systems [12]. They have a number of attractive features:

- When the hardware is too dreadful for ordinary mortals to look upon, concealment is a kindness, if not a necessity.
- Useful and popular facilities can be made available in a uniform manner, with the name binding and storage allocation required to implement them kept out of the way.
- The system can protect itself from the users without having to make any assumptions about what they do (aside from those implicit in the definition of the virtual machine).
- A more robust facility can perhaps be provided if all of the underlying structure is concealed.

[1] This paper was presented at the 7th ACM Symposium on Operating Systems Principles, November 1979 and published in *ACM Operating Systems Review* **11**, 5 (Dec. 1979), pp 98-105. This version was created from that one by scanning and OCR; it may have errors.

[2] Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

On the other hand, a good deal may be lost by putting too much distance between the user and the hardware [4], especially if she needs to deal with unconventional input-output devices. Furthermore, a lot of flexibility is given up by the flat, all-or-nothing style of these systems, and it is extremely difficult for a user to extend or modify the system because of the sharp line which is drawn between the two.

In this paper we explore a different, more "open" approach: the system is thought of as offering a variety of facilities, any of which the user may reject, accept, modify or extend. In many cases a facility may become a component out of which other facilities are built up: for example, files are built out of disk pages. When this happens, we try as far as possible to make the small components accessible to the user as well as the large ones. The success of such a design depends on the extent to which we can exploit the flexibility of the small components without destroying the larger ones. In particular, we must pay a great deal of attention to the robustness of the system, i.e., recovery from crashes and resistance to misuse.

Another aspect of our system is that the file system and communications are standardized at a level below any of the software in the operating system. In fact, it is the representation of files on the disk and of packets on the network that are standardized. This has permitted programs written in radically different languages (BCPL [4], Mesa [8], Lisp [7] and Smalltalk [10]; the former came first, and is the host language of the software described in this paper) and executed using radically different instruction sets (implemented in writeable microcode) to share the same file system and remote facilities. In doing so, they do not give up any storage to an operating system written in a foreign language, or any cycles in switching from one programming environment and instruction set to another at every access to disk storage or communications.

The price paid for this flexibility is that any change in these representations requires changing several pieces of code, written in several languages and maintained by several different people; the cost of this rewriting is so high that it is effectively impossible to make such changes. Thus the approach cannot be recommended when processor speed and memory are ample; standardization at a higher level is preferable in this case. In our situation, however, the policy has made many major applications of the machine possible which would otherwise have been completely infeasible. Furthermore, we have found that these restrictions have caused few practical problems, in spite of the fact that the range of uses of the system has been far greater than was initially anticipated.

In a multi-user system, of course, there must be compulsory protection mechanisms that ensure equitable and safe sharing of the hardware resources, and this consideration sets limits to the openness that can be achieved. Within these limits, however, much can be done, and indeed the facilities discussed below can be provided in a protected way without any great changes, although this paper avoids an explicit analysis of the problem by confining itself to a single-user system.

To describe an entire system in this way would be a substantial undertaking. We will confine ourselves here to the disk file system, the method of doing world swapping, and the way in which the system is constructed out of packages.

2. Background

The operating system from which the examples in this paper are drawn was written for a small computer called the Alto [16], which has a 16-bit processor, 64k words of 800 ns memory, and one or two moving-head disk drives, each of which can store 2.5 megabytes on a single removable pack and can transfer 64k words in about one second. The machine and system can also support another disk with about twice the size and performance. The machine has no virtual memory hardware. The processor executes an instruction set that supports BCPL, including special instructions for procedure calls and returns.

The system is written almost entirely in BCPL, and in fact this language is considered to be one of the standard ways of programming the machine. The compiler generates ordinary machine instructions, and uses no runtime support routines except for a small body of code that extends the instruction set slightly.

Only one user at a time is supported, and peripheral equipment other than the disk and terminal is infrequently used. As a result, the current version of the system has only two processes, one of which puts keyboard input characters into a buffer, while the other does all the interesting work. The keyboard process is interrupt-driven and has no critical sections; hence there are no synchronization primitives and no scheduler other than the hardware interrupt system. As a result, the system does not control processor allocation, and in fact gets control only when a user program calls some system facility. The system does control storage allocation to some extent, both in main memory and on the disk, in order to make it possible for the user’s programs to coexist and to call each other.

Thus the system can reasonably be viewed as a collection of procedures which implement various potentially useful abstract objects. There is no significant difference between these system procedures and a set of procedures that the user might write to implement his own abstract objects. In fact, the system code is made available as a set of independent subroutine packages, each implementing one of the objects, and these packages have received a great deal of independent use, in applications which do not need all the services of the system and cannot afford its costs.

There are several kinds of abstract object: input-output streams, files, storage zones, physical disks. All of these objects are implemented in such a way that they can be values of ordinary variables; since BCPL is a typeless language this means that each object can be represented by a 16-bit machine word. In many cases, of course, this word will be a pointer to something bigger.

The streams are copied wholesale from Stoy and Strachey’s OS6 system [15], as are many aspects of the file system. We give a summary description here for completeness. A stream is an object that can produce or consume items. Items can be arbitrary BCPL objects: bytes, words, vectors, other streams etc. There is a standard set of operations defined on every stream:

Get an item from the stream;

Put an item into the stream (normally only one of these is defined);

Reset, which puts the stream into some standard initial state (the exact meaning of this operation depends on the type of the stream);

Test for end of input;

and a few others. These operations are invoked by ordinary BCPL procedure calls.

A stream is thus something like a Simula class [6]. It differs from a class in that the procedures that implement the operations are not the same for all streams, and indeed can change from time to time, even for a particular stream. A stream is represented by a record (actually a BCPL vector) whose first few components contain procedures that provide that stream's implementation of the standard operations. The rest of the record holds state information, which may vary from stream to stream (e.g. word counts, pointers to buffers, disk addresses, or whatever is appropriate). The size of the record is not fixed, but rather is determined entirely by the procedure that creates the stream.

It is also possible for the record to contain procedures that implement non-standard operations (e.g. set buffer size, read position in a disk file, etc.). Alternatively, arbitrary procedures can be written which perform such operations on certain types of stream. In both cases the procedure receives the record which represents the stream as an argument, and can store any permanent state information in that record. Of course, a program that uses a non-standard operation sacrifices compatibility, since it will only work with streams for which that operation is implemented.

This scheme for providing abstract objects with multiple implementations is used throughout the system. Each abstract object is defined by the operations that can be invoked on it; the semantics of each operation are defined (more or less rigorously). Any number of concrete implementations are possible, each providing a concrete procedure for each of the abstract operations. Hierarchical structures can be built up in this way. For instance, the procedure to create a stream object of concrete type "disk file stream" takes as parameters two other objects: a disk object which implements operations to access the storage on which the file resides, and a zone object which is used to acquire and release working storage for the stream.

3. Pages and files

The system organizes long-term storage (on disk) into *files*, each of which is a sequence of fixed-size *pages*; every page is represented by a single disk sector. Although a file is sufficient unto itself, one normally wants to be able to attach a string name to it, and for this purpose an auxiliary *directory* facility is provided. Since the integrity of long-term storage is of paramount importance to the user, a *scavenging* procedure is provided to reconstruct the state of the file system from whatever fragmented state it may have fallen into. The requirements of this procedure govern much of the system design. The remainder of this section expands on the outline just given.

3.1 Pages

The simplest object that can be used for long-term storage is a page. It consists of:

an *address*—one word which uniquely specifies a physical disk location (H);

a *label*, which consists of:

F:	a file identifier—two words (A);
V:	a version number—one word (A);
PN:	a page number—one word (A);
L:	a length (the number of bytes in this page that contain data)—one word (A);
NL:	a next link—one word (H);
PL:	a previous link—one word (H);

a *value*—256 data words (A).

The information that makes up a page is of two kinds: *absolutes* (A) and *hints* (H). The page is completely defined by the absolutes. The hints, therefore, are present solely to improve the efficiency of the implementation. Whenever a hint is used, it is checked against some absolute to confirm its continued validity. Furthermore, there is a recovery operation that reconstructs all the hints from the absolutes.

Thus a page has a unique *absolute name*, which is the file identifier, version number and page number (represented by (FV, *n*), where *n* is the page number and FV is the file identifier and version), and it has a *hint name*, which is the address. The *full name* (FN) of a page is the pair (absolute name, hint name). The links of the page (FV, *n)* are the addresses of the pages whose absolute names are (FV, *n*-1) and (FV, *n*+1), or NIL if no such pages exist. The basic operations on a page are to read and write the data, and to read the links, given the full name. Note that it is easy to go from the full name of a page to the full names of the next and previous pages.

3.2 Files

A file is a set of pages with absolute names (FV, 0), (FV, 1) (FV, *n*). The name of page (FV, 0) is also the name of the file. The basic operations on files are

create a new, empty file;
add a page to the end of a file;
delete one or more pages from the end;
delete the entire file.

Thus, if page (FV, *n)* exists, pages (FV, *i*) exist for all *i* between 0 and *n*. Hence, if the address of one page of a file is known, every page can be found by following the links. If (FV, *n*) is the last page of the file, then pages (FV, *i*) for $i<n$ have L=512 (i.e., they are full of data), and the last page has L<512.

The data bytes of the file are contained in pages 1 through *n*. Page 0 is called the *leader* page, and contains all the properties of the file other than its length and its data:

dates of creation, last write, and last read (A);
a string called the *leader name*, discussed in Section 3.5 (A);
the page number and disk address of the last page (H);
a *maybe consecutive* flag (H).

3.3 Representation of pages

The physical representation of a page on the disk is called a *sector*, and consists of three parts:

> a *header*, which contains the disk pack number (different for each removable pack) and the disk address;
>
> a label, which contains the seven words specified in Section 3.1;
>
> a value, which contains the 256 data words.

A single disk operation can perform read, check or write actions independently on each of these parts, with the restriction that once a write is begun, it must continue through the rest of the sector. A check action compares data on the disk with corresponding data taken from memory, word by word, and aborts the entire operation if they don't match. If a memory word is 0, however, it is replaced by the corresponding disk word, so that a check action is a simple kind of pattern match.

The system uses these facilities to make the disk a rather robust storage medium. Disk pages are always accessed by their full names. The label of a sector is always checked before it is written, and is written on only three occasions:

> When the page is freed—its full name must be given, and the check is that the label is the right one. Then ones are written into label and value, to ensure that any attempt to treat the page as part of a file will fail with a label check error.
>
> The first time the page is written after it has been allocated—the check is that the page is free. Then the proper label for the page is written.
>
> In order to change the length of the file—the label of the last page is read and checked. Then it is rewritten, possibly with new values of L and NL.

This scheme costs a disk revolution each time a page is allocated or freed, but it makes accidental overwriting of a page quite unlikely. On any other write the label is checked, at no cost in time. The check action is distinguished from a read so that a subsequent write operation can be aborted before anything is written, without taking an extra revolution. The label is also checked on reads, of course. If the checks succeed, it is certain that the hint (address) used to access a disk page actually leads to the page specified by the absolute part of the full name. As we shall see below, it is also possible to find a page from the absolute name alone (though not very efficiently) and to reconstruct all the hint names from the absolute names.

A disk contains a file called the *disk descriptor* with a standard name and disk address. In it are:

> the *allocation map, a* bit table indicating which pages are free (H);
>
> the disk *shape*, i.e., number of tracks, surfaces, and other information needed to parameterize the disk routines for a particular model of disk (A);

the name of the root directory (H).

Note that the allocation map is a hint because the absolute information about which pages are free is contained in the labels. If the map says that a page is free, the allocator marks it busy when allocating it, and when the label check described above fails, the allocator is called again to obtain another page. Thus a page improperly marked free in the map results in a little extra one-time disk activity. A page improperly marked busy will never be allocated; such lost pages are recovered by the Scavenger described in Section 3.5.

Our implementation of the disk descriptor is slightly different from the description above. The main directory has a standard name and disk address, and points to the disk descriptor file. This scheme arose because the disk descriptor was added to the file system data structure when it became apparent that the disk shape must be recorded in some way. The description given above reflects the logical relationship between the disk descriptor and the main directory (i.e., that's how we should have done it).

3.4 Directories

So far we have constructed a data storage facility based on pages, and an allocation facility based on files. Allocation is not provided at the page level because losing track of pages is too easy, which in turn is because a page, being of fixed size, cannot be a logical unit of storage. A file, on the other hand, can be of arbitrary size and hence is a suitable receptacle for a collection of data that the user views as a unit (as long as it isn't too small). Since a file is a logical unit, moreover, it should have a logical name, i.e., a string name, as well as its unique file identifier, and the name should be interpreted in some context, so that names can be assigned independently without fear of conflict.

This is the familiar line of reasoning which leads to a tree-structured directory hierarchy [5]. Our system takes a somewhat different tack (following OS6) because of our desire to treat files as independent objects in their own right. We take the view that any operation on a file can be performed with no more than a knowledge of its full name (which is the full name of its first page), and that a separate mechanism exists for associating names with files. This is done by a file called a *directory,* which contains a set of pairs (string, full name). A file may appear in any number of directories. Since there is nothing special about a directory from the point of view of the file system, it is possible to have a tree, or indeed an arbitrary directed graph, of directories. We do need to be able to identify all the directories for the scavenging procedure described below, and to this end we reserve a subset of the file identifiers for directory files.

A further implication (and here we part company with OS6) is that it must be possible to recover some logical name from the file itself, so that the file can survive even if the directory entries for it are lost or scrambled. This is the purpose of the leader name in the leader page; its significance should be apparent from the roles that it plays in scavenging. The information in the leader page is considered to be absolute, since this is a name by which the file can be located even if all the directory entries for it are destroyed. Directory entries, by contrast, are taken less seriously, although they are not entirely redundant and hence cannot be treated as pure hints. If a directory is destroyed, we don't lose any files, but we do lose some information, namely the information that a certain set of files was referenced from that directory by a certain set of names.

3.5 Scavenging

By reading all the labels on the disk, we can check that all the links are correct (reconstructing any that prove faulty), obtain full names for all existing files, and produce a list of free pages. To do this, all we have to do is create a list of all the labels not marked free and sort it by absolute name. If there is enough main storage to hold a table with 48 bits per sector, a suitable choice of data structure allows this processing to be done without any auxiliary storage. This is in fact the case for the machine's standard disks. Larger disks require this list to be written on a specially reserved section of the disk.

We can then read all the directories and verify that each entry points to page 0 of an existing file, fixing up the address if necessary and detecting entries which point elsewhere. If any file remains unaccounted for by directory entries, we can make a new entry for it in the mail directory, using its leader name. This is the sole function of the leader name.

This entire process is called scavenging, and it takes about a minute for a 2.5 megabyte disk. When it is complete, all hints have been recomputed from absolutes, and any inconsistencies (incomplete files, null directory entries, nameless files, etc.) have been detected. The question of what to do with the inconsistencies is beyond the scope of this paper. During scavenging any permanently bad pages are marked in the label with a special value so that they will never be used again.

As we have noted, scavenging cannot fully reconstruct lost directories. This could be accomplished by writing a journal of all changes to directories and taking an occasional snapshot of all the directories. By applying the changes in the journal to the snapshot we would get back the current state. This is of course a standard technique by which the integrity of any database may be safeguarded. For the reasons already mentioned, we do not consider our directories important enough to warrant such attentions. If the user disagrees, he is free to modify the system-provided procedures for managing directories, or to write his own.

We have also written a more elaborate scavenger that does an in-place permutation of the file pages on the disk so that the pages of each file are in consecutive sectors. This arrangement typically increases the speed with which the files can he read sequentially by an order of magnitude over what is possible if the pages have become scattered.

3.6 Using hints

The purpose of hints is to increase performance. For example, if a program possesses the full name (FV, *i*) of a file page and the hint address, it can access the page directly without going through a directory lookup and without scanning down the chain of data blocks. If this direct access fails (i.e., the page at that hint address turns out not to be (FV, *i*)), the program has several options:

> It may have a full name for some other portion of the file (typically, the leader page) which is correct. Then it can follow links from that page, still avoiding the directory

lookup. Hint addresses can also be kept for every *k*-th page of the file to reduce the number of links that must be followed.

If this fails, it may look up the FV in a directory to obtain the proper disk address.

If this fails, it may look up the string name of the file in a directory to obtain a new FV and disk address.

Finally, it may invoke the Scavenger to reconstruct the entire file system and all the directories, and then retry one of the earlier steps.

Note that such a hint can be expanded to name a particular byte within the file system, simply by augmenting a full name with a byte position within the page.

The hint mechanism can also be used advantageously for files that are thought to be allocated consecutively. A program is free to assume that a file is consecutive and, knowing the address a_i of page i, to compute the address of page j as a_i+j-i. The label check will prevent any incorrect overwriting of data, and will inform the program whether the disk access succeeds.

Many programs use a collection of auxiliary files to which they need rapid access. The editor, for example, uses two scratch files, a journal file, a file of messages etc. When these programs are "installed", they create the necessary files and store hints for them in a data structure that is then written onto a *state file*. Subsequently the program can start up, read the state file, and access all its auxiliary files at maximum disk speed. If a hint fails, e.g. because a scratch file got deleted or moved, the program must repeat the installation phase. It doesn't matter where hints are stored, and the system makes no effort to keep them up to date. It simply insures that when a hint fails, no damage is done, and the program using the hint is informed so that it can take corrective action.

4. Communication between programs

A key objective of most operating systems is to foster communication between separate programs, often written in different programming languages or environments. But how can communication be provided by an open operating system that allows the programmer to reject all facilities of the system? The most conservative solution is to allow communication only through disk files, since the file structure must be observed by all programs. For example, a command scanner may write the command string typed by the user on a file with a standard name, and may then invoke a program that will execute the command. Ordinary disk files can be used in this way to pass data from one program to another. The disk file structure must also serve as a way to invoke an arbitrary program, that is, to "transfer control" from one program to another. Because of the openness of the operating system, the called and calling programs may have little or nothing in common.

These transfers of control are achieved by defining a convention for restoring the entire state of the machine from a disk file; this allows an arbitrary program to take control of the machine. The files that describe the machine state can be used to implement several control disciplines. A coroutine structure is commonly used: a program first records its state on one disk file, and then restores the machine state from a second file. The original program resumes execution when the machine state Is restored from the first file.

The interprogram communication mechanism has found many uses. Examples are:

—Bootstrapping. A hardware bootstrap button causes the state of the machine to be restored from a disk file whose first page is kept at a fixed location on the disk. This *boot* file may be written by a linker that writes programs and data in the file, arranged so that they will constitute a running program when the machine state is restored from the file. Alternatively, the file may have been written by saving the state of a running program that will be resumed each time the machine is bootstrapped.

—Debugging. When a breakpoint is encountered or when the user strikes a special DEBUG key on the keyboard, the state of the machine is written on a disk file, and the machine state is restored from a file that contains the debugger. The debugging program may examine or alter the state of the faulty program by reading or writing portions of the file that was written as a result of the breakpoint. The debugger can later resume execution of the original program by restoring the machine state from the file. The original program and the debugger thus operate as coroutines.

—Checkpointing. A program may occasionally save its state on a disk file. It may then be interrupted, either by a processor malfunction or by user action (e.g., bootstrapping the machine). The computation may be resumed later by restoring the machine state from the checkpoint file.

—Activity switching. The coroutine structure is used to switch between several tasks that are part of a single application. One example is a *printing server,* a program that accepts files from a local communications network and prints them. The program is divided into two tasks: a *spooler* that reads files from the network and queues them in a disk file, and a *printer* that removes entries from the queue and controls the hardware that prints them. Because each of these tasks has considerable internal state and operates in a different environment, they communicate using the state save/restore mechanism. Whenever the spooler is idle but the queue is not empty, it saves its state and calls the printer. Whenever the printer is finished or detects incoming network traffic, it stops the printer hardware, saves its state, and invokes the spooler. This scheme easily allows printing to be interrupted in order to respond quickly to incoming files.

In many systems, state-restoring mechanisms would be extremely dangerous, as they could lead to severely damaged disk file structures. This may happen if a program is saved when it contains copies in memory of parts of the disk data structure (e.g., parts of a directory, or an allocation map that indicates free blocks) and it is never restored, or fails to update this information to reflect the actual disk state after it is restored. The label-checking machinery described in section 3.3 prevents catastrophic damage if the information is incorrectly updated. Moreover, hints that are saved and restored are usually still valid, and can be used to re-read quickly whatever disk data may have been cached in main memory.

4.1 InLoad and OutLoad

Two procedures, *InLoad* and *OutLoad,* are the ones most commonly used to operate on disk files containing machine states. Each routine takes as argument the full name of a disk file, and requires about a second to complete its operation:

> (*written, message*) : = *OutLoad*(*OutFN*)
> *InLoad*(*InFN, message*)

OutLoad writes the current machine state on the file, and returns with the *written* flag true. Note that the program counter saved on the output file is inside the *OutLoad* procedure itself. The *InLoad* procedure restores the state of the machine from the given file, and passes a *message* (about 20 words) to the restored program. The effect is that *OutLoad* returns again, this time with *written* false and with the message that was provided in the *InLoad* call. Code for a coroutine linkage thus looks like:

> . . .
> *messageToPartner* = parameters to pass in coroutine call;
> (*written, messageFromPartner*) := *OutLoad*(*myStateFN*);
> **if** *written* **then** *InLoad*(*partnerStateFN, messageToPartner*);
> *messageFromPartner* contains parameters passed to me;
> . . .

If the parameters in the coroutine call will not fit in the small message vector, the vector is used to pass the full name of a place on the disk where the parameters have been written. Often the message contains a *return address,* that is, the full name of a file to restore upon return. In the example above, a return address can be provided by copying *myStateFN* into *messageToPartner* before the *InLoad* call.

The *InLoad* and *OutLoad* procedures, although quite small (about 900 words), are nevertheless subject to obliteration by a sufficiently errant program. For linkage to debuggers, it would be preferable if these routines were protected in some way. Machines without memory protection hardware should probably implement the procedures in read-only memory (or in processor microcode that cannot be damaged). We fashioned a partial solution to this problem with a special emergency bootstrap program, containing only the *OutLoad* procedure, that writes most of the machine state onto a disk file. Unfortunately, this method could not preserve some of the most vital state (e.g., processor registers).

5. Organization of the open system

The operating system is a collection of commonly used subroutine packages that are normally present in memory for the convenience of user programs. The system provides streams for disk files, keyboard input and display output; routines for reading and writing disk pages directly; the *OutLoad* and *InLoad* procedures; a free storage allocator; BCPL runtime routines; and storage for a good deal of handy data, such as hints for frequently-used files, the user’s name and password, etc. The subroutine packages are written almost entirely in BCPL, with consistent conventions for storage allocation and for object invocation.

5.1 Invoking programs

The system includes a procedure for invoking BCPL programs that execute under the operating system. Code for the program is read from a disk stream and loaded into low memory addresses. All references to operating system procedures are bound, using a fixup table contained in the code file. Finally, the program is invoked by calling a single entry routine. The program may terminate either by calling the program loader to read in another program and thus overlay the first program, or by returning from the main procedure. If the program returns, the system loads and runs a standard *Executive* program. The Executive accepts user commands from the keyboard and executes them, often by calling the loader to invoke a program the user has requested.

Programs that run under the operating system may also be invoked from an entirely different programming environment. The *InLoad* procedure is invoked on the file that contains the operating system state, which causes the system to be loaded and initialized. The message vector passed to *InLoad* may contain the name of a file containing the program to be invoked. A stream is opened on this file, and the program is loaded and run.

5.2 Junta

The packages that form the operating system are organized to support its openness. A program that prefers not to use the standard procedures provided by the system, or that needs to use the memory space occupied by them, may request that some or all system procedures be deleted from memory. The procedure that removes procedures is called *Junta* because it forcibly takes over the machine. When a program terminates, a *CounterJunta* procedure is called to restore the standard procedures from the *InLoad/OutLoad* context for the operating system.

The system is organized into several levels of services, so that a program may select the procedures it wishes to retain. Procedures are arranged so that the lowest level, which contains the most commonly used services, is at the very top of memory. Less ubiquitous services are in levels with higher numbers, located lower in memory. The highest level number to be retained is passed as an argument to *Junta*, which removes all higher-numbered levels and frees the storage they occupy. The *CounterJunta* procedure restores all levels that were removed, and reinitializes any data structures they contain.

The facilities in each level are summarized below:

1. *OutLoad/InLoad*, *CounterJunta*
2. Keyboard input buffer.
3. Hints for important files.
4. BCPL runtime procedures (e.g., stack frame allocator).

5,6. Disks (code and data for the disk object for the standard disk).

7. Zones (code for the standard free-storage object).
8. Disk streams (code for standard disk stream objects).
9. Disk directories.
10. Keyboard streams (code for standard keyboard stream object).
11. Display streams (code for standard display stream objects).

12. The program loader and Junta procedure.
13. System free storage, including room for default display stream, disk streams, etc.

The keyboard input buffer is present nearly always, so that any characters typed ahead by the user when running one program are saved for interpretation by the next,

The *Junta* and *CounterJunta* procedures give the programmer simple but precise control over the operating system facilities retained in memory. Unlike more elaborate mechanisms such as swapping code segments, this scheme guarantees the performance of the resident system.

There is no distinction between procedures and data of the user and those of the system. The storage allocator, for example, will build zone objects to allocate any part of memory, whether in the system free storage region or not. The routine that creates a disk stream object requires two other objects as parameters: a zone to allocate space for the stream data structure (defaulted to the system free storage zone), and a disk object used to invoke disk transfers (defaulted to the "standard disk"). It is common for a program using a large non-standard disk to include a package that implements only the disk object for the special disk hardware, and to open streams on files using the standard operating system disk stream implementation.

A programmer desiring even more flexibility is encouraged to remove most of the system with *Junta* and to incorporate copies of the standard packages in his own program, placed wherever he wants. A common reason for this approach is that scarce memory space forces the programmer to overlay program structures. For example, a file server program that uses only the non-standard big disk nevertheless uses the standard disk stream package, organized in overlays. The display, keyboard, and storage-allocation packages have been assembled to form an operating system for use without a disk, used to support diagnostics or other programs that depend on network communications rather than on local disk storage.

The success of the operating system as a collection of subroutine packages depends primarily on the design of the packages. Retaining common procedures in memory simply saves disk space by reducing the size of many programs. Retaining common data structures in memory saves initialization time. The *Junta* procedure is a convenience, but not a necessity. It is the considerable effort that was devoted to refining the subroutine packages that makes them useful both as a cohesive operating system and as separate packages. Like any language design, this is a non-trivial undertaking whose difficulty is belied by the apparent simplicity of the result,

6. Conclusion

We have described the design of three major parts of a small operating system. In each case considerable trouble has been taken to make all the facilities accessible to users, in the sense that they can build up their own macro-operations from the primitives in the system if those provided by the system prove unsatisfactory. In the treatment of files we emphasized the methods used to minimize the probability that data will be destroyed. and to permit full automatic recovery after a crash. The program communication facilities emphasize reasonable communication between programs that take over the entire machine and organize it in different ways. The organization into levels accommodates programs which don't go quite so far, but still need a great deal of control over their own destiny; here proper design of the packages which make up the system to permit stand-alone operation is crucial.

The measures taken to make the file system robust, in which the label checking is crucial, have worked extremely well. Many thousands of file systems currently exist, running on hundreds of machines without any centralized maintenance. The incidence of complaints about lost information is negligible. The hint schemes have also worked well in making it possible to obtain high performance from the system. Because of inadequate explanation of the proper use of hints, however, many programmers did not understand exactly what was a hint and what was absolute, or how to recover properly from failure of a hint. As a result, it is more common than it should be for a program to crash with the message "Hint failed, please reinstall," rather than automatically invoking the proper recovery procedure.

The open character of the system has also been successful, and has fostered a large number of different programming environments that work together quite harmoniously. The Junta has been used frequently to remove standard handlers for human input/output that simulate a teletype terminal, so that experimental programs can control interaction very carefully, or in novel ways.

The disadvantages of openness are about what one would expect. Since it is not possible to virtualize the system, there is no practical way to change the representation or functionality of the file system or communications. Furthermore, there is no way to intercept all accesses to the file system, display, or whatever and direct them to some other device, such as a remote file system. This could be done only by changing the machine's microcode.

Acknowledgements

Many of the facilities described above were first implemented by Gene McDaniel. The current implementation was done by the authors, and has since been improved by David Boggs. The original implementations of *InLoad/OutLoad* and of the Scavenger are due to Jim Morris.

References

1. Bensoussan, A., et al., "The Multics virtual memory," *Comm. ACM* **15**, 5 (May 1972).
2. Bobrow, D. G. et al., "Tenex, a paged time sharing system for the PDP-10," *Comm. ACM* **15**, 3 (March 1972).
3. Bobrow, D. G. and B. Wegbreit, "A model and stack implementation of multiple environments," *Comm. ACM* **16**, 10 (Oct 1973).
4. Brinch Hansen, P., *Operating Systems Principles*, Prentice-Hall, New York, 1973.
5. Corbato, F. J. et al., "An introduction and overview of the Multics system," *Proc. AFIPS Conf.* **27** (1965 FJCC).
6. Dahl, O-J. and C. A. R. Hoare, "Hierarchical program structures," in *Structured Programming*, Academic Press, New York, 1972.

7. Deutsch, L. P., "Experience with a microprogrammed Interlisp system," *IEEE Trans, Computers* **C-28**, 10 (Oct 1979).
8. Geschke, C. M., J. H. Morris Jr., and E. H. Satterthwaite, "Early experience with Mesa," *Comm. ACM* **20**, 8 (Aug 1977).
9. Hoare, C. A. R. and R. M. McKeag, "A survey of store management techniques," in *Operating Systems Techniques*, Academic Press, New York, 1972.
10. Ingalls, D., "The Smalltalk-76 programming system: Design and implementation," *Fifth ACM Symposium on Princip1es of Programming Languages*, Tucson, Arizona, Jan 1978.
11. Knuth, D. E. *The Art of Computer Programming,* vol. 1, Addison-Wesley, Reading, Mass., 1968.
12. Lampson, B. W. et a!, "A user machine in a time-sharing system," *Proc IEEE* **54**, 12 (Dec 1966).
13. Meyer, P. A. and L. H. Seawright, "A virtual machine time-sharing system," *IBM Systems Journal* **9**, 3 (July 1970).
14. Richards, M., "BCPL: A tool for compiler writing and system programming," *Proc. AFIPS Conf.* **35** (1969 SJCC).
15. Stoy, J. E. and C. Strachey, "OS6—An experimental operating system for a small computer," *Computer Journal* **15**, 2 and 3.
16. Thacker, C.P. et. al., "Alto: A personal computer," to appear in *Computer Structures: Readings and Examples,* Sieworek, Bell and Newell, eds., McGraw-Hill, 1979.