Inter-Office Memorandum

To	CSL, SSL	Date	February 2, 1973
From	Butler Lampson	Location	Palo Alto
Subject	Design notes for Alto Operating System	Organization	PARC

XEROX

This memo is a tentative specification of the operating system for Alto which Gene McDaniels is writing. Comments and suggestions are solicited.

We begin by considering what features we will need in six months, and then try to find a subset with which to start. Stoy and Strachey's OS6 system (Computer Journal, July and October 1972) is the source for a good deal of the terminology and some of the ideas.

1. Basic capabilities

disk files

names and directories; versions;

sequential and random access;

disk scavenging, syntax check and recovery procedures;

partitioning the disk to allocate separately either:

consecutive cylinders for swapping or overlay areas, or the two disks, since one is removable.

input-output

streams which can mediate transfers between programs and disk files, i-o devices, or other programs, more or less as in Stoy and Strachey;

drivers for keyboard, mouse and display;

driver for whatever communication facility we provide in the hardware.

command language, i.e., a way to call in programs for execution

storage allocation and overlay facilities for core.

2. Some numbers

Memory: A 'standard' machine is 48Kx16 bits. A 'big' machine is 64K.

A full display buffer is 32K, but the amount of memory used by the display is directly proportional to the amount of display surface covered, and this can be controlled quite finely, in fact, in arbitrary rectangles which start at the left edge of the screen.

Disk: 2 packs (1 removable), each with 2 surfaces and 406 cylinders. Each track is 12 sectors of 256 data words, an 8-word label, and a 2-word header. 9744 sectors/pack.

Rotation time: 25 ms. Seek time: 8+3* sqrt (dt). Average access time to one megabyte on consecutive cylinders is about 30 ms.

A bit table for 1 pack is 2.4 pages (of 256 words).

Transfer rate is 133 ms/16K or .5 sec/big memory.

Time to read every sector on one pack is 22 seconds.

Display: 820 x 620 raster. Using 5x7 characters with 2-bit displacement for descenders we get ~100 characters/line, 80 lines. In this font, 72 character lines take up 270 words each. Obviously, prettier fonts take more space.

3. Locating files

Every file has a unique tag called its serial number. We assume 32 bits for this number; one of these bits distinguishes directories from other files. Every file also has a file number, which is an index into a master file table (MFT) which maps the file number into the disk address of the leader page for the file.

A directory entry contains some naming information (discussed in more detail in section 6) and a three-part address:

file number	FN
serial number	SN
leader page address	LA (optional)

To get the leader page:

read the page at LA;

if its label (see below) says that it is the leader page for file SN, (PN=Ø) we are done;

otherwise, read entry FN in MFT and read the page at that address.

if its label says "leader for SN" we are done (and can optionally update LA in the directory).

otherwise the file no longer exists.

The motivation for MFT is to decouple the file system from the naming system. It permits the entire file, including its leader page, to be moved around freely without any reference to the directories.

4. File structure

A file consists of a leader and a body. The leader occupies one (or more) pages of its own, disjoint from other leaders and from the body. It contains:

title, a string which is not used by the operating system but may help to identify the file if all its directory entries get lost

access control information: (no access, read-only, read-write), (scratch, normal, permanent)

history: date created, date written, date last read

serial number, version number

file number

locator, which tells how to get to the body

Two forms seem reasonable for the locator: sequential and random. A sequential locator is just the disk address of the first page of the file. Each page then points to the next and previous ones. A random locator is a list of disk addresses for the pages of the file. Pros and cons of these two schemes:

Sequential pro: no need to write the header when closing the file

header always fits on one page

Sequential con: random access is costly (but the incremental scheme described in section 5 below mitigates the severity of this problem).

Proposal: we should use a sequential file structure initially. I can't see any reason why we can't add or switch to a random structure later.

Each page on the disk consists of three records:

a 2 word header:

partition (see section 7)
disk address

an 8 word label

NP	ptr (disk address) to next page of file
PP	ptr to previous page
SN	serial number of file (2 words)
VN	version number
PN	page number in file (starting with 1)
	2 unused words

a 256 word data record

A disk controller command specifies a disk address and an operation for each of the three records of the addressed sector chosen from the list: read, check, write. The only constraint is that writing, once started, must continue for the whole sector. A check operation compares disk data with core data and signals an error when it finds a disagreement, except that if the core data word is -1 there is no error and the disk data replaces the -1. This allows the check to be a simple kind of pattern match. Each record has a checksum which is checked on a read or check operation, generated on a write. Any error stops the controller immediately.

Controller commands can be chained together, and the controller can transfer to or from consecutive sectors without restrictions. Again, one error stops the controller, preventing it from proceeding down the chain. Each command is marked when it is completed. Details of the controller interface are documented in the Alto manual.

The capabilities sketched above allow full speed transfers from consecutive sectors of a sequentially structured file (by reading the label into core so that its final word overwrites the last word of the next command). The pattern-match feature allows SN, VN and PN to be checked while NP is read, so that writes can be done into an already-allocated sequential file without giving up any checking.

5. Operations on files

Either the file number or the leader address will do as identification for a file, together with SN, VN. The operations we need are:

open (i.e. make stream; see section 8 below) for input by byte; word or page

open for output by byte, word or page

open for random access (maximum file size). This returns a value which can be used as a parameter to three other operations:

move core to file (core address, file address, count)

move file to core (same arguments)

close

intended implementation: construct a page number to disk address map in core, and fill it in incrementally as pages of the file are touched in performing move operations

delete body

delete file

read heading

create file

set access controls.

6. Directories

A directory is a file which contains a list of entries. Each entry contains:

NAME	a string
TYPE	which may be file or link a value which depends on the type

For a file, the value is a pair: <serial number, list of triples: <version number, file number, leader address>>.

For a link the value is another string which is taken as the full name of the file being referenced. This form of indirect addressing may be iterated to a reasonable depth.

The form of the file name in NAME is:

identifier ['.' identifier]

Here an identifier is a string of printing characters not including ( ) [ ] . , ; or blank. The second identifier is intended to be thought of as a type, and procedures which operate on directories will implement the usual * conventions for both parts of the name.

A full file name has the form:

['>'] [directory name $ ('>' directory name)] file name ['!' version number]

Here "directory" is just a file name. Each name is looked up in the current directory, which is set initially to a value specified as a parameter to the lookup procedure and is changed whenever a directory name is encountered in the full name. The initial > sets the current directory to the system directory.

Operations on directories are:

enter (directory, name, type, value)	deletes the entry if the type is NIL
lookup (directory, name)	which returns NIL or a three-part address, following links and using the latest version in case of ambiguity
lookuponce (directory, name)	which returns NIL or a type and value
entries from (directory)	which returns a stream which delivers successive directory entries

Initially we can do without versions, links and multiple directories.

7. Partitions, slices and allocation

A sector on the disk is specified by an address with four components:

cylinder: $0 \leq c < C$ C=406

surface:	$0 \leq s < S$	S=2
pack:	$0 \leq p < P$	P=2
record:	$0 \leq r < R$	R=12

A track is a group of sectors with the same c, s and p. We will use virtual addresses for disk sectors, computing the virtual address by:

$$v = ((c P + p) S + s) R + r$$

so that increasing v scans first, all the records of a track, then all the tracks on one pack in one cylinder, then all the packs, and finally the cylinders. Hence abs(v1 - v2) small implies short seek time between v1 and v2.

A slice is a set of sectors (c, p, s, r) such that:

$C\ min \leq c \leq C\ max$	$lC = C\ max - C\ min + 1$
$P\ min \leq p \leq P\ max$	lP etc.
$S\ min \leq s \leq S\ max$	
$R\ min \leq r \leq R\ max$	

It takes 8 numbers to specify a slice. We can get a reasonably efficient function which delivers successive virtual disk addresses in a slice by observing that any slice consists of kernels (each of which may be part of a track, a track, a pack cylinder (all the tracks with fixed c and p), or a cylinder) within which a consecutive range of virtual addresses is in the slice. The kernels are characterized by an origin kØ = (C min, P min, S min, R min), a length l and the shortest interval i between the start of kernels. If the interval between consecutive kernels is always i, the slice is simple. For a simple slice, v is in the slice if (v-kØ) mod i <l. Most interesting slices are simple.

A partition is a triple <partition number N, slice 1, slice 2>. All the sectors in the slices have the partition number written in the header. The partition has a serial number <0,N> and a leader which is exactly like a file leader except that the locator is two slices. There are two things to do with a partition: allocate files in it, or use it as one big file. Files are not allowed to have pages in more than one partition.

Basic operations on partitions:

create partition (number) creates an empty partition

transfer slice (slice, from partition, to partition). All the pages must be free

delete partition (partition). The partition must be empty

Note the constraint that a partition can only consist of two slices.

We need two operations to move pages relative to slices:

block transfer (from partition, to partition) copies all the pages and frees the old copies

flush (slice, partition) copies all the pages out of the slice into the partition. The slice may be contained in the partition

These operations have to update the locator information, and the MFT if leaders are moved. They don't have to update directories.

We can do without all this initially.

8. Disk scavenger

The purpose of the disk scavenger is to check the correctness of all the structured information used by the file and directory system, to reconstruct secondary data kept for efficiency (such as the allocation bit table) from primary data (headers and labels), to report discrepancies (such as pages for which there is no leader) and to leave the disk in a consistent state.

The following steps will suffice (hopefully).

a) Read every label and construct two tables:

1) file table FT, with one entry for each file, containing its VN, SN, header address and newly assigned FN

2) disk table DT, with one entry for each disk page, containing the file system address of the page: (FN,PN)

In the process check the validity of all the pointer chains using a technique described below. Also check that each file has a header page with sensible information on it.

b) Reconstruct the allocation bit table from DT

c) Reconstruct MFT from FT

d) Sort all unaccounted-for pages by file system address, constructing leader pages and dummy data blocks and fixing up pointer chains if necessary. Log all the problems found, identifying the file by SN, VN, and title if it had a leader page. DT can now be discarded

e) Read all the directories, log and delete entries for non-existent files, update FN and LA for each entry. Keep a reference count, for each file, of directory entries referring to it

f) Log all files with no directory entries and either delete them or optionally put them into a special directory (or the main directory in the initial system) with a name derived from the title

The main trick is to make FT and DT fit into core, which may be only 48K, and to make only one pass over the disk. We index DT by virtual disk address, adjusted if necessary so as to agree with the order in which pages are read. FT will be hashed on (SN, VN) into a linear hash table whose size is determined by the available space. Position in this table will do for FN until step (a) is complete; the table can then be compacted and DT adjusted if desired.

A DT entry may have several forms:

address:	FN, 12 bits at most, for a limit of 4,000 files
	PN, 3 bits. Files with more than 8 pages have dummy FN's assigned from a randomly chosen empty FT entry for each additional 8 pages
chained:	CP, 15 bits. Points to DT entry for previous page of the file. Pages with PN=Ø are never chained
special:	chained, but toward the next page. Only for pages not yet read, special entries are marked as address, since pages not yet read can't be address
empty:	Ø
free:	1

Address and chained entries are distinguished by the sign bit. Each DT entry thus occupies exactly one word.

All DT entries start out empty. When a label is read the following processing has to be done.

a) create a FT entry for the file if one doesn't already exist. Compute the file system address A of the page

b) the DT entry for the page may be empty, chained or special. If it is empty make it address. If it is chained, follow the chain until an address A' is found and check that A-A' = the number of links traversed (or whatever). If it is special, do a similar check, make it address and make the page it pointed to chained

c) examine the DT entry for the pages pointed to by the two pointers in the label. For each one, if it is address, chained or special, check that it has the correct address, using the method of (b) for a chained entry. Make a following address or special page chained. If it is free give an error. If it is empty, that page hasn't been read yet. Make it chained or special

When all is done, all the segments of files which are held together with pointers in labels are held together in DT. It is easy to check for incomplete pointer chains, and to read off all the pages of every file from the CH words in FT.

A real FT entry contains:

SN: 32 bits

VN: 15 bits

CH: address of last file page currently in ST. This can be updated as DT in constructed, or, if space is tight, it can be derived later by a single pass through DT, room being made for it by moving VN into the end of the chain, which normally contains a file address. This only works for a well-formed chain, which has the PN of the end = Ø

REAL: 1 bit = 1, indicating that this is a real file

A dummy FT entry has REAL = Ø and contains a pointer to the corresponding real entry plus the page number in the real file of page Ø in the dummy

9. Streams

A stream is like a coroutine. It is implemented as a structure which points to the functions which implement the various operations, and may also contain variables which are local to the stream and represent its

state (e.g. a pointer to a buffer block). Following Stoy and Strachey we have the following operations on streams:

next (stream)	delivers the next item. If items are >1 word, it takes another argument which points to a place to put the data
out (stream, data)	
end of (stream)	is true if the end of input has appeared. It is strictly up to the stream to define this
reset (stream)	rewinds a disk file or magtape and does other appropriate things for other devices
close (stream)	gets rid of it

This summary leaves open sticky questions about special operations which may be needed by streams like the keyboard, and the handling of control functions in general. Further thought is required, but probably we can do without these things initially.

10. Display (to be written)

Keyboard and mouse (to be written)

Memory allocation and control of programs (to be written)