INTRODUCTION

The Operating System is a single user system providing concurrent processes, events, exceptions, device independent I/O in a hierarchical file system, and management of code and data segmentation. This manual is intended for applications programmers who deal directly with the Operating System.

The Operating System falls naturally into four categories: file management, process management, memory management, and process communication. In each of the four chapters describing these portions of the Operating System, there is an overview of the subject that explains the terms and concepts used in the system calls. The system calls themselves are then described in some detail. A fifth chapter describes system startup procedures. The Appendices describe the Operating System interface and error codes.

FILE OVERVIEW

INTRODUCTION

The File System provides device independent I/O, reliable storage with access protection, uniform file naming conventions, and configurable device drivers.

A file is an uninterpreted stream of eight bit bytes. A file that is stored on a block structured device resides in a catalog and has a name. For each such file the catalog contains an entry describing the file's attributes including the length of the file, its position on the disk, and the last backup copy date. Arbitrary application-defined attributes can be stored in an area called the file label.

Each file has two associated measures of length, the Logical End of File (LEOF) and the Physical End of File (PEOF). The LEOF is a pointer to the last byte that has meaning to the application. The PEOF is a count of the number of blocks allocated to the file. The pointer to the next byte to be read or written is called the file marker.

To handle input and output, applications do not need to know the physical characteristics of a device. Applications that do, however, can increase the I/O performance by causing file accesses on block boundaries. Each Operating System call is synchronous in that the I/O requested is performed before the call returns. The actual I/O, however, is asynchronous and is always performed in the context of an Operating System process.

To reduce the impact of an error, the file system maintains a high level of distributed, redundant information about the files on storage devices. Duplicate copies of critical information are stored in different forms and in different places on the media. All the files are able to identify and describe themselves, and there are usually several ways to recover lost information. The scavenger program is able to discover and reconstruct damaged directories from the information stored with each file.

FILE NAMES

All the files known to the Operating System at a particular time are organized into a tree of catalogs. At the top of this tree is a predefined catalog with names for the highest level objects seen by the system. These include physical devices, such as a printer or a modem, and the volume names of any disks that are available.

Any object catalogued in the file system can be named by specifying the volume in which the file resides and the file name. The names are separated by the character "-". Because the top catalog in the tree has no name, all complete pathnames begin with "-".

For example,

-PRINTER names the physical printer,

-LISA-FORMAT.TEXT
names a file on a volume named LISA.

The file name can contain up to 32 characters. If a longer name is specified, the name is truncated to 32 characters. Accesses to sequential devices use a dummy filename that is ignored but must be present in the pathname. For example, the serial port pathname

-RS232B

is illegal, but

-RS232B-XYZ

is accepted, even though the -XYZ portion is ignored. Certain device names are predefined:

RS232A	Serial Port 1
RS232B	Serial Port 2
UPPER	Upper Twiggy drive (Drive 1)
LOWER	Lower Twiggy drive (Drive 2)
DEV0, DEV6, DEV7, DEV8	Bit bucket (byte stream is flushed into oblivion)

Upper and lower case are significant in file names: 'TESTVOL' is not the same object as 'TestVol'. Any ASCII character is legal in a pathname, including the non-printing characters.

THE WORKING DIRECTORY

It is sometimes inconvenient to specify a complete pathname, especially when working with a group of files in the same volume. To alleviate this problem, the operating system maintains the name of a working directory for each process. When a pathname is specified without a leading "-", the name refers to an object in the working directory. For example, if the working directory is -LISA the name FORMAT.TEXT refers to the same file as -LISA-FORMAT.TEXT. The default working directory name is the name of the boot volume directory.

DEVICES

The Lisa hardware supports a variety of I/O devices including the keyboard, mouse, clock, two Twiggy disk drives, two serial ports, a parallel port, and three expansion I/O slots. The screen, keyboard, and mouse are accessed through LisaGraf and the Window Manager. The other devices are handled by the Operating System.

Device names follow the same conventions as file names. Attributes like baud rate and print intensity are controlled by using the DEVICE_CONTROL call with the appropriate pathname.

All device calls are synchronous from the process point of view. Within the Operating System, however, I/O operations are asynchronous. The process doing the I/O is blocked until the operation is complete.

Each device has a permanently assigned priority. From highest to lowest the priorities are:

```
Serial Port 1   (RS232A)
Serial Port 2   (RS232B, the leftmost port)
I/O Slot 0
I/O Slot 1
I/O Slot 2
Speaker
10 ms system timer
Keyboard, mouse, soft-off switch, battery powered clock
CRT vertical retrace interrupt
Parallel Port
Twiggy 1        (UPPER)
Twiggy 2        (LOWER)
Video Screen
```

The Operating System maintains a Mount Table which connects each available device with a name and a device number. The Device Driver associated with a device knows about the device's physical characteristics such as sector size and interleave factors for disks.

STRUCTURED DEVICES

On structured devices, such as disk drives, the File System maintains a higher level of data access built out of pages (logical names for blocks), label contents, and data clusters (groups of contiguous pages). Any file access ultimately translates into a page access. Intermediate buffering is provided only when it is needed. Each page on a structured device is self-identifying, and the page descriptor is stored with the page contents to reduce the destructive impact of an I/O error. The eight components of the page descriptor are:

```
Version number
Volume identifier
File identifier
Amount of data on the page
Page name
Page position in the file
Forward link
Backward link
```

Each structured device has a Media Descriptor Data File (MDDF) which describes the various attributes of the media such as its size, page length, block layout, and the size of the boot area. The MDDF is

created when the volume is initialized.

The File System also maintains a bitmap of which pages on the media are currently allocated, and a catalog of all the files on the volume. Each file contains a set of file hints which describe and point to the actual file data. The file data need not be allocated in contiguous pages.

THE VOLUME CATALOG

On a block structured device, the volume catalog provides access to the files. The catalog is itself a file which maps user names into the internal files used by the Operating System. Each catalog entry contains a variety of information about each file including:

- name
- type
- internal file number and address
- size
- date and time created or last modified
- file identifier
- safety switch

The safety switch is used to avoid accidental deletions. While the safety switch is on, the file cannot be deleted. The other fields are described under the LOOKUP file system call.

The catalog can be located anywhere on the media, and the Operating System may even move it around occassionally to avoid wear on the media.

LABELS

An application can store its own information about file attributes in an area called the file label. The label allows the application to keep the file data separate from information maintained about the file. Labels can be used for any object in the file system. The maximum label size is 488 bytes.

LOGICAL AND PHYSICAL END OF FILE

A file contains some number of bytes recorded in some number of physical blocks. Additional blocks might be allocated to the file, but not contain any file data. There are, therefore, two measures of the end of the file called the logical and physical end of file. The logical end of file (LEOF) is a pointer to the last stored byte which has meaning to the application. The physical end of file (PEOF) is a count of the number of blocks allocated to the file.

In addition, each open file in each process has a pointer associated with it called the file marker that points to the next byte in the file to be read or written. When the file is opened, the file marker points to the first byte (byte number 0). The file marker can be positioned implicitly or explicitly using the read and write calls. It cannot be positioned past LEOF, however, except by a write operation that appends data to a file.

When a file is created, an entry for it is made in the catalog specified in its pathname, but no space is allocated for the file itself. When the file is opened by a process, space can be allocated explicitly by the process, or automatically by the operating system. If a write operation causes the file marker to be positioned past the Logical End Of File (LEOF) marker, LEOF and PEOF are automatically extended. The new space is contiguous if possible, but not necessarily adjacent to the previously allocated space.

FILE ACCESS

There are several modes in which an application can perform input, output, or device control operations. Applications are provided with a device independent byte stream interface. A specified number of bytes is transferred either relative to the file marker or at a specified byte location in the file. The physical attributes of the device or file are not seen by the application, except that devices that do not support positioning can only perform sequential operations.

Applications that know the block size for structured devices can optimize performance by performing I/O on block boundaries in integral block multiples. This mode bypasses the buffering of parts of blocks that the system normally performs. Data transfers take place directly between the device and the computer memory. Although data transfers occur in physical units of blocks, the file marker still indicates a byte position in the file.

A file can be open for access simultaneously by multiple processes. All write operations are completed before any other access to the file is permitted. When one process writes to a file the effect of that write is immediately visible to all other processes reading the file. The other processes may, however, have accessed the file in an earlier state and not be aware of the change until the next time they access the file. It is left up to the applications to insure that processes maintain a consistent view of a shared file.

Each time a file is opened, the Operating System allocates a file marker for the calling process and a run-time identification number called the refnum. The process uses the refnum in subsequent calls to refer to the file. Each operation using the refnum affects only the file marker associated with it. The refnum is global only if a process has opened the file with global access. The LEOF and PEOF values, however, are always global attributes of the file, and any change to these values is immediately visible to all processes accessing that file.

Processes can share the same file marker. In this access mode (global access) each of the processes uses the same refnum for the file. When a process opens a file in global access mode, the refnum it gets back can be used by any process. Note that [Global_Access] access allows the same file to be opened globally by any number of processes, creating any number of simultaneously shared refnums. [Global_Access,Private] access opens a file for global access. but allows no other process to Open that file. Applications must be aware of all the side effects that global accesses cause. For example, processes making global accesses to a file cannot make any assumptions about the location of the file marker from one access to the next.

Even if the access mode is not global, more than one process can have the same file open simultaneously. Each process, in this case, has its own refnum and file marker. A write operation to the file, however, is immediately visible to all readers of that file.

PIPES

Because the Operating System supports multiple processes, a mechanism is needed for interprocess communication. This mechanism is called a pipe. A pipe is very similar to any other object in the file system — it is named according to the same rules, and can have a label. A pipe also implements a byte stream that queues information in a first-in-first-out manner for the pipe reader. Unlike a file, however, a pipe can have only one reader at a time, and once data is read from a pipe it is no longer available in the pipe.

A pipe can only be accessed in sequential mode. Only one process can read data from a pipe, but any number of processes can write data into it. Because the data read from the pipe is consumed, the file marker is always zero. If the pipe is empty and no processes have it open for writing, End Of File is returned. If any process does have it open for writing, the reading process is suspended until data arrives in the pipe, or until all writers close the pipe.

When a pipe is created, its physical size is 0 bytes. You must allocate space to the pipe before trying to write data into it. To avoid deadlocks between the reading process and the writers, the Operating System does not allow a process to read or write an amount of data greater than half the physical size of the pipe. For this reason, you should allocate to the pipe twice as much space as the largest

amount of data in any planned read or write operation.

A pipe is actually a circular buffer with a read pointer and a write pointer. All writers access the pipe through the same write pointer. Whenever either pointer reaches the 'end' of the pipe, it wraps back around to the first byte. If the read pointer catches up with the write pointer, the reading process blocks until data is written or until all the writers close the pipe. Similarly, if the write pointer catches up with the read pointer, a writing process blocks until the pipe reader frees up some space or until the reader closes the pipe. Because pipes have this structure, there are certain restrictions on some operations when dealing with a pipe. These restrictions are discussed below under the relevant file system calls.

For massive data transfers, it is recommended that shared files or data segments be used rather than pipes.

FILE SYSTEM CALLS

This section describes all the operating system calls that pertain to the file system. A summary of all the Operating System calls can be found in Appendix A. The following special types are used in the file system calls:

```
Pathname = STRING[255];
E_Name = STRING[Max_Ename];      (* Max_EName = 32 *)
Accesses = (DRead, DWrite, Append, Private, Global_Access);
MSet = SET OF Accesses;
IoMode = (Absolute, Relative, Sequential);
```

The fs_info record and its associated types are described under the LOOKUP call.

PROCESSES

INTRODUCTION

A process is a piece of executable code that can be run at the same time as other processes. Although processes can share code and data, each process has its own stack. In most systems, including the one supported by the Operating System, the parallel or concurrent execution of the processes is simulated by using re-entrant code and a scheduler. The scheduler allows each process to run until some condition occurs. At that time, the state of the running process is saved, and the scheduler looks at the pool of ready-to-run processes for the next one to be executed. When the first process later resumes execution, it merely picks up where it left off in its execution.

The status of a process depends on its scheduling state, execution state, and memory state. The memory manager handles the process memory state. If any code or data segments need to be swapped in for the process to execute, the memory manager is called before the process is launched by the scheduler.

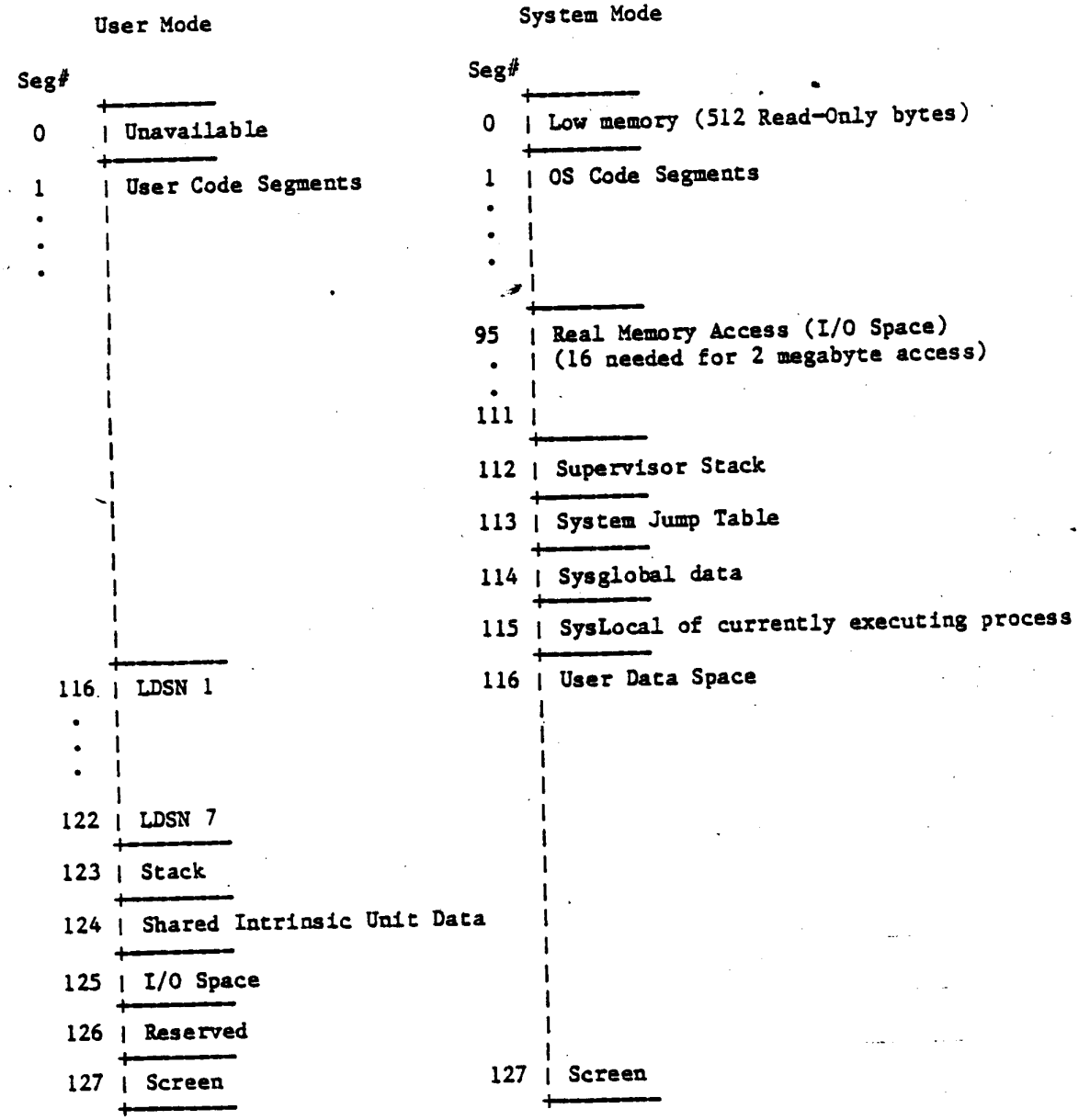
The process execution state depends on whether the process is executing in user mode or in system mode. In system mode, the process executes Operating System code in the hardware domain 0. In user mode, the process executes user code in domains 1, 2, or 3.

The process scheduling state has four possibilities. The process is "running" if it is actually engaging the attention of the CPU. If it is ready to continue execution, but is being held back by the scheduler, the process is said to be "ready". When it has completed its task and has exitted its outer block, it is "terminated". A process can also be "blocked". In the blocked state, the process is ignored by the scheduler. It cannot continue its execution until something causes its state to be changed to "ready". Processes commonly become blocked while awaiting completion of I/O. Certain Operating System calls distinguish between a process that is blocked by an I/O operation, and a process that is blocked because it has been suspended by some other process.

PROCESS STRUCTURE

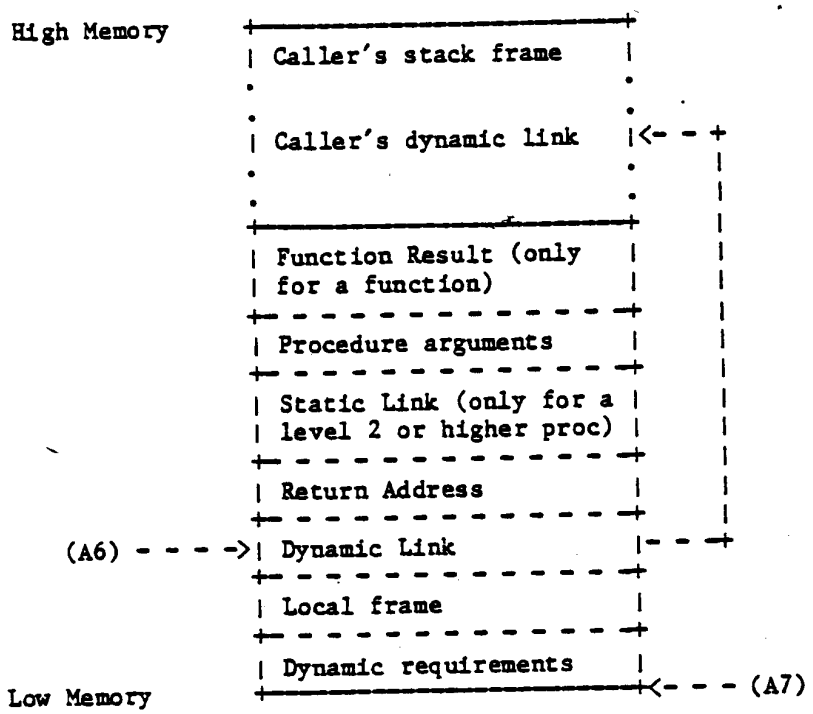
A process is a program. It can use up to 7 data segments and 116 code segments simultaneously. When a process is instantiated, the Operating System creates a Process Control Block (PCB) for it. The PCB contains the process state, global id, and a pointer to a record of the process's current needs. These include pointers to its code and data segments, its stack, an area to save registers, and so on. When a process calls the Operating System, the data segments and stack of the process are remapped into domain 0 where the Operating System executes. The address space layout of system and user processes is set up to make this remap as efficient as possible:

PROCESS ADDRESS SPACE LAYOUT



During execution, the process stack is:

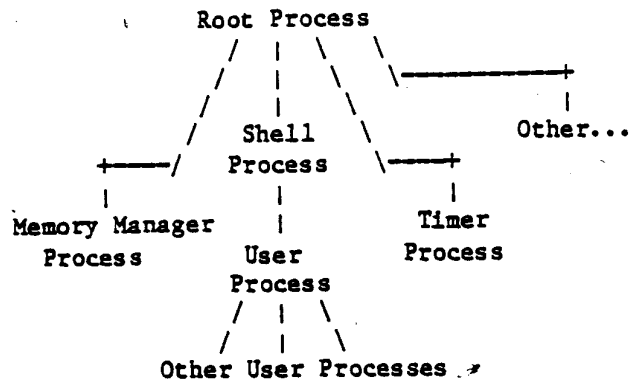
PROCESS STACK LAYOUT



Each process has an associated priority, an integer between 1 and 255. The process scheduler usually executes the highest priority ready process. The higher priorities (200 to 255) are reserved for Operating System and Filer processes.

PROCESS HIERARCHY

When the system is first started, several system processes exist. At the base of the process hierarchy is the root process which handles various internal Operating System functions. It has at least three sons, the memory manager process, the timer process, and the shell process. The memory manager process handles code and data segment swapping. The shell process is a simple command interpreter which you can use to run programs and create other processes. In the final Lisa system, the shell process will be the Filer. The timer process handles timing functions such as timed event channels.



Any other system process (the Network Control Process, for example) is a son of the root process.

PROCESS CREATION

When a process is created, it is placed in the ready state, with a priority equal to that of the process which created it. All the processes created by a given process can be thought of as existing in a subtree. Many of the process management calls can affect the entire subtree of a process as well as the process itself.

PROCESS CONTROL

Three system calls are provided for explicit control of a process. These calls allow a process to kill, suspend (block), or activate any other user process in the system. Process handling calls are not allowed on Operating System processes.

PROCESS SCHEDULING

Process scheduling is based on the priority established for the process. The system usually attempts to execute the highest priority ready process. Once it is executing a process loses the CPU only under the following conditions:

* The running process becomes blocked (during I/O, for example).
* The running process lowers its priority below that of another ready process or sets another process's priority to be higher than its own.
* The running process yields the CPU to another process.
* The running process activates a higher priority process or suspends itself.

* The running process makes any Operating System call when a higher priority ready process exists.

* The running process causes code to be swapped or its stack to be expanded.

Because the Operating System currently cannot seize the CPU from an executing process except in the cases noted above, background processes should be liberally sprinkled with YIELD_CPU calls.

When the scheduler is invoked, it saves the state of the current process and selects the next process to run by examining its pool of ready processes. If the new process requires code or data to be swapped in, the memory manager process is launched. If the memory manager is already working on a process, the scheduler selects the highest priority process in the ready queue that does not need anything swapped.

PROCESS TERMINATION

A process terminates when it hits its 'END.' statement, when it calls TERMINATE_PROCESS, when some process calls KILL_PROCESS on it, when its father process terminates, or when it runs into an abnormal condition. When a process terminates, a "terminate" exception condition is signalled on the calling process and all of the processes it has created. A process can declare an exception handler for this condition to insure that its house is in order before its demise.

Termination involves the following steps:

1. Signal the SYS_TERMINATE exception on the current process.

2. Execute the user's exception handler (if any).

3. Send the SYS_SON_TERM event to the father of the current process if a local event channel exists.

4. Instruct all sons of the current process to terminate.

5. Close all open files, data segments, and event channels.

6. Wait for all the sons to finish termination.

7. Release the PCB and return to the scheduler.

A process can protect itself from termination by disabling the "terminate" exception. Under normal circumstances, however, a process should cooperate with the Operating System by viewing the terminate exception as an opportunity to clean up its act before it is terminated. If a process disables the terminate exception and then, illogically, calls TERMINATE_PROCESS, the Operating System forces the process to terminate.

MEMORY MANAGEMENT OVERVIEW

INTRODUCTION

Each process has a set of code and data segments which must be in physical memory during execution of the process. The transformation of the logical address used by the process to the physical address used by the memory controller to access physical memory is handled by the memory management unit (MMU).

A LIMITED HARDWARE PERSPECTIVE

Addresses in LISA have three parts: a domain (context) number, a hardware segment number, and an offset. A hardware segment is a contiguous logical address space with a distinct address protection. The hardware mapping registers determine each hardware segment's type, length (in pages of 512 bytes), and origin in physical memory. The segment type (ReadOnly, ReadWrite, or Stack) controls access to that segment.

Each segment can have up to 128 Kbytes of memory. The Operating System provides data segments larger than 128 Kbytes by allocating adjacent MMU registers to a single logical segment. 128 segments are mapped by a single domain, so each of the four domains provides a cache of an entire segment map. The Operating System runs in domain 0; application programs can operate in domains 1, 2, or 3. The use of domains speeds up process switching.

DATA SEGMENTS

Each process has a data segment that the Operating System automatically allocates to it for use as a stack. The stack segment's internal structures are managed directly by the hardware and the Operating System.

A process can require additional data segments for such things as heaps and process to process communication. These added requirements are made known to the Operating System at run time. The Operating System views all data segments except the stack as linear arrays of bytes. Therefore, allocation, access, and interpretation of structures within a data segment are the responsibility of the process.

The 68000 hardware requires that all data segments that are part of the process's working set be in physical memory and mapped by hardware segment registers during execution of the process. It is the responsibility of the process to ensure that this requirement is met.

THE LOGICAL DATA SEGMENT NUMBER

Besides the stack segment, a process can have up to seven data segments in its working set at any given time. Other data segments can be available to the process, but not actually be members of the working set. To inform the Operating System that it wants a certain data segment to be available, the process associates that segment with a "logical data segment number" (LDSN). When the process wants the data segment placed in memory and made a member of the working set, it "binds" that segment to its associated LDSN. The LDSN, which has a valid range of 1 to 7, is local to the calling process. The process uses the LDSN to keep track of where a given data segment can be found. More than one data segment can be associated with the same LDSN, but only one such segment can be bound to an LDSN at any instant and thus be a member of the working set of the process.

SHARED DATA SEGMENTS

Cooperating processes can share data segments. The segment creator assigns the segment a unique name (a file system pathname). All processes that want to share that data segment must then use the same segment name. If the shared data segment contains address pointers to segments, then the cooperating processes must also agree upon a common LDSN to be associated with the segment. This LDSN is transformed by the Operating System into a specific mapping register, so all logical data addresses referencing locations within the data segment are consistent for all processes sharing the segment.

As an example of the use of shared data segments, consider the following situation: a process creates five other processes and wants to use a different data segment for communication with each of them. The process can associate and bind the five data segments with LDSN values 1 to 5. Since it can access all five segments at will, this method can have performance advantages, but all five data segments must be in memory during execution. If on the other hand, the process associates all five data segments with the same LDSN, only one such segment must be in memory at any time, but the process must bind and unbind the segments to the LDSN whenever a specific segment is needed. The application designer must weigh the advantages and disadvantages of each method for the application being developed.

PRIVATE DATA SEGMENTS

Data segments can also be private to a process. In this case, the maximum size of the segment can be greater than 128 Kbytes. The actual maximum size depends on the amount of physical memory in the machine and the number of adjacent LDSN's available to map the segment. The process gives the desired segment size and the base LDSN to use to map the segment. The Memory Manager then uses ascending adjacent LDSN's to map successive 128 Kbyte chunks of

the segment. The process must insure that enough consecutive LDSN's are available to map the entire segment.

Suppose a process has a data segment already bound to LDSN 2. If the program tries to bind a 256 Kbyte data segment to LDSN 1, the Operating System returns an error because the 256 Kbyte segment needs two consecutive free LDSN's. Instead, the program should bind the segment to LDSN 3 and the system implicitly also uses LDSN 4. If the program has no bound LDSN's, it can start its heap segment at LDSN 1, and as the heap grows, it can expand upward through the 7 LDSN's.

CODE SEGMENTS

Division of a program into multiple code segments (swapping units) is dictated by the programmer. If a program is so divided, the Linker creates a jump table to insure that intersegment procedure references are handled properly. The MMU registers can map up to 116 code segments. The allocation of the register numbers is given in the Process Structure section of the Process chapter.

A JSR, RTS, or JMP.L to a non-resident code segment causes a bus error which results in a trap to the Operating System (a software implementation of absence traps). The Operating System brings the code segment into physical memory and returns control to the process, allowing the procedure reference to continue.

THE PROCESS STACK

Because the Operating System sometimes needs to scan the stack of a process, certain conventions must be observed:

* Register A7 is the stack pointer of the process.
* Register A6 is the link register for the process stack.
* All procedures must execute the LINK instruction using A6 as the link register before any local data is placed on the stack or another procedure call is executed.

These conventions are obviously hidden from the programmer's view in high level languages, but must be followed by assembly language programmers.

Stack expansion is handled automatically by the Operating System.

SWAPPING

When a process executes, the following segments are required to be in physical memory and mapped by mapping registers:

* The current code segments being executed

* All the data segments in the process working set.

The Operating System insures that this minimum set of segments is in physical memory before the process is allowed to execute. If a required segment is not in memory, a segment swap-in request is initiated. In the simplest case, this request only requires the system to allocate a block of physical memory and to read in the segment from the disk. In a worse case, the request may require that other segments be swapped out first to free up sufficient memory. A clock algorithm is used to determine which segments to swap out or replace.

EXCEPTIONS and EVENTS

Processes have several ways to keep informed about the state of the world. Normal-process-to process communication and synchronization can be handled using events or shared data segments. An abnormal condition can cause an exception (interrupt) to be signalled which the process can respond to in whatever way it sees fit.

EXCEPTIONS

Normal execution of a process can be interrupted by an exceptional condition (such as division by zero or address error). Some of these conditions are trapped by the hardware, some by the system software, and others can be signalled by the process itself. Exceptions have character string names, some of which are predefined and reserved by the Operating System.

When an exception occurs, the system first checks the state of the exception. The three exception states are:

* Enabled
* Queued
* Ignored

If the exception is enabled, the system next looks for a user defined handler for that exception. If none is found, the system default exception handler is invoked. It usually aborts the current process.

If the state of the exception is queued, the exception is placed on a queue. When that exception is subsequently enabled, this queue is examined, and if any exceptions are found, the appropriate exception handler is entered. Processes can flush the exception queue.

If the state of the exception is ignored, the system still detects the occurrence of the exception, but the exception is neither honored nor queued.

Invocation of the exception handler causes the sceduler to run, so it is possible for another process to run between the signalling of the exception and the execution of the exception handler.

SYSTEM DEFINED EXCEPTIONS

Certain exceptions are predefined by the Operating System. These include:

* Division by zero (SYS_ZERO_DIV). Default handler aborts process.

* Value out of bounds (SYS_VALUE_OOB). Default handler aborts process.

* Overflow (SYS_OVERFLOW). Default handler aborts process.

* Process termination (SYS_TERMINATE). This exception is signalled when a process terminates, or when there is a bus error, address error, illegal instruction, privilege violation, or line 1010 or 1111 emulator error. The default handler does nothing.

Except where otherwise noted, these exceptions are fatal if they occur within Operating System code. The hardware exceptions for parity error, spurious interrupt, and power failure are also fatal.

EXCEPTION HANDLERS

A user-defined exception handler can be declared for a specific exception. This exception handler is coded as a procedure, but must follow certain conventions. Each handler must have two input parameters: Environment_Ptr and Exception_Ptr. The Operating System ensures that these pointers are valid when the handler is entered. Environment_Ptr points to an area in the stack containing the interrupted environment: register contents, condition flags, and program state. The handler can access this environment and can modify everything except the program counter and register A7. The Exception_Ptr points to an area in the stack containing information about the specific exception.

Each exception handler must be defined at the global level of the process, must return, and cannot have any "Exit" or "Global Goto" statements. Because the Operating System disables the exception before calling the exception handler, the handler should re-enable the exception before it returns.

If an exception handler for a given exception already exists when another handler is declared for that exception, the old one becomes disassociated. There is no notion of block structured declaration of exception handlers.

An exception can occur during the execution of an exception handler. The state of the exception determines whether it is queued, honored, or ignored. If the second exception has the same name as the exception that is currently being handled and its state is enabled, a nested call to the exception handler occurs.

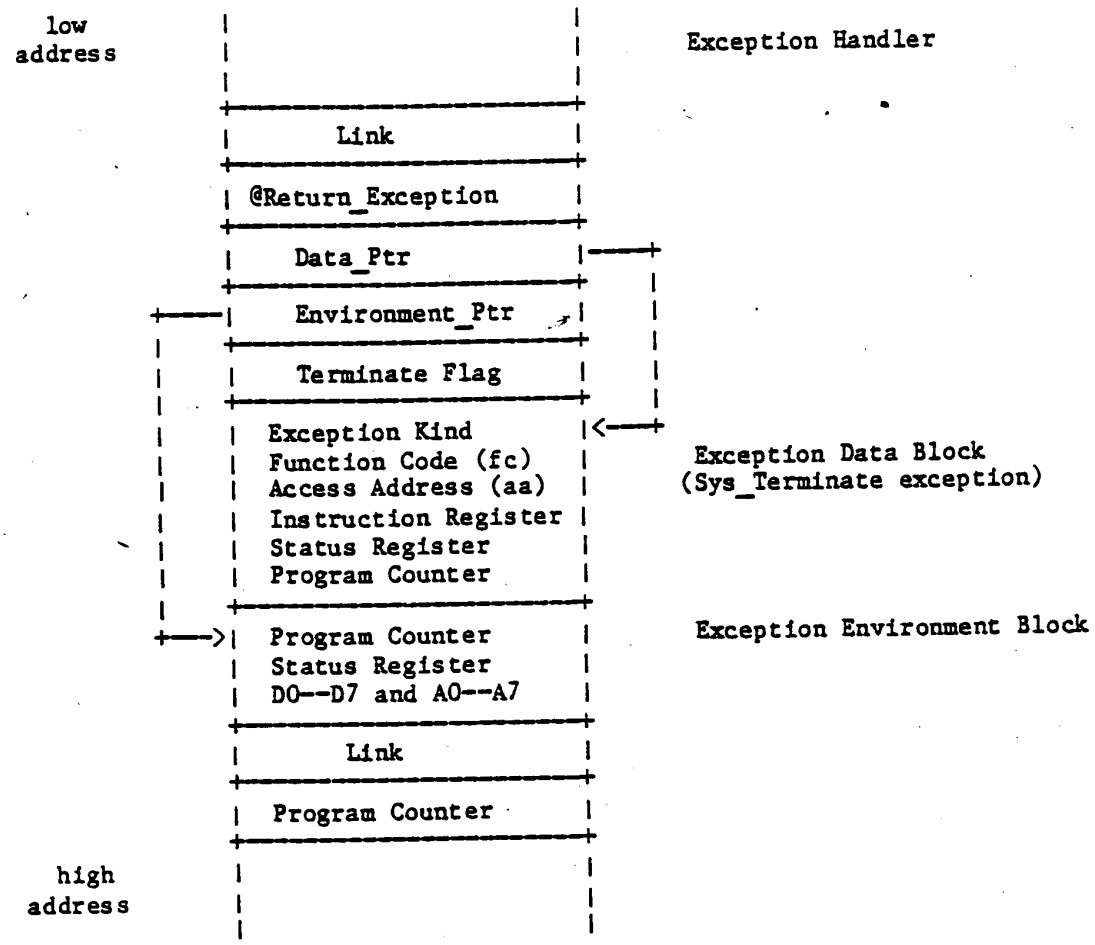
There is an "exception occurred" flag for every declared exception; it is set whenever the corresponding exception occurs. This flag can be examined and reset. Once the flag is set, it remains set

until FLUSH_EXECP is called.

The following code fragment gives an example of exception handling.

```
PROCEDURE Handler(Env_Ptr:p_env_blk;
                  Data_Ptr:p_ex_data);
VAR ErrNum:INTEGER;
BEGIN
(* Env_Ptr points to a record containing the program counter, *)
(* and all registers.  Data_Ptr points to an array of 12 longints *)
(* that contain the event header and text if this handler is *)
(* associated with an event-call channel (see below) *)
.
.
.
ENABLE_EXCEP(errnum,excep_name);
.
.
.
END;
.
(* this is either in a different segment or at the top level *)
.
Excep_name:='EndOfDoc';
DECLARE_EXCEP_HDL(errnum,excep_name,@Handler);
.
.
.
SIGNAL_EXCEP(errnum,excep_name,excep_data);
.
.
.
```

At the time the exception handler is invoked, the stack is:



The Exception Data Block given here reflects the state of the stack upon a SYS_TERMINATE exception. The term_ex_data record described in the Interface appendix gives the various forms the data block can take. The status register and program counter values in the data block reflect the true (current) state of these values. The same data in the Environment block reflects the state of these values at the time the exception was signalled, not the values at the time the exception actually occurs.

In the case of a bus or address error, the PC can be 2 to 10 bytes beyond the current instruction. The PC and A7 cannot be modified by the exception handler.

When a disabled exception is re-enabled, a queued exception may be signalled. In this case, the exception environment reflects the state of the world at the time the exception was re-enabled, not the time at which the exception occurred.

EVENTS

An event is a piece of information sent by one process to another, generally to help cooperating processes synchronize their activities. An event is sent through a kind of pipe called an event channel. The event is a fixed size data block consisting of a header and some text. The header contains control information; the identifier of the sending process and the type of the event. The header is written by the system, not the sender, and is readable by the receiving process. The event text is written by the sender; its meaning is defined by the sending and receiving processes.

There are several predefined system event types. The predefined type "user" is assigned to all events not sent by the Operating System.

EVENT CHANNELS

Event channels can be viewed as a higher-level approach to pipes. The most important difference is that event channels deal with fixed size data blocks, whereas pipes can handle an arbitrary byte stream.

An event channel can be globally or locally defined. A global event channel has a globally defined pathname catalogued in the file system, and can be used by any process to handle user defined events. A local event channel, however, has no name and is known only by the Operating System and the process that opened it.

A local event channel is automatically created when a process is created. This channel can be opened by the father process to receive system generated events pertaining to its son.

There are two types of event channels: event-wait and event-call. If the receiving process is not ready to receive the event, an event-wait type of event channel queues an event sent to it . An event-call type of event channel, however, treats its event as an exception. The exception name must be given when the event-call event channel is opened, and an exception handler for that exception must be declared. When an event is sent to an event-call event channel, the Operating System signals the associated exception. If the process reading the event-call channel is suspended at the time the event is sent, the event is queued and is executed when the process becomes active.

When an event channel is created, the Operating System preallocates enough space to the channel for typical interprocess communication. If SEND_EVENT_CHN is called when the channel does not have enough space for the event, the calling process is blocked until enough space is freed up.

The following code fragment uses event-wait channels to handle process synchronization:

```
PROCESS A                          PROCESS B
---------                          ---------

Open Chn_1 to receive;             Open Chn_1 to send;

Open Chn_2 to send;                Open Chn_2 to receive;

REPEAT                             REPEAT

    Send to Chn_2;                     Wait for Chn_2;

    Wait for Chn_1;                    Send to Chn_1;

UNTIL AllDone;                     UNTIL AllDone;
```

The order of execution of the two processes is the same regardless of the process priorities. In the following example using event-call channels, however, the process priorities do affect the order of execution.

```
PROCESS A                          PROCESS B
---------                          ---------

Declare Excep_1;                   Declare Excep_2;

Open Chn_1 to receive Excep_1;     Open Chn_1 to send;

Open Chn_2 to send;                Open Chn_2 to receive Excep_1;

Send Chn_2;

PROCEDURE Handler;                 PROCEDURE Handler;
   Send Chn_2;                         Send Chn_1;
   Yield_Cpu;                          Yield_Cpu;
```

THE SYSTEM CLOCK

A process can read the system clock time, convert to local time, or delay its own continuation until a given time. The year, month, day, hour, minute, second, and millisecond are available from the clock. The system clock is in Greenwich mean time.

EXCEPTION MANAGEMENT CALLS

The event and exception management routines use several special types and constants. To save space and reduce redundancy, these types are defined only in Appendix A, and are referred to in the rest of this chapter without much further comment.

SYSTEM CONFIGURATION AND STARTUP

SYSTEM STARTUP

Startup is a multi-step operation. After the startup request is generated, code in the bootstrap ROM executes. This code runs a series of diagnostic tests, and signals by a beep that all is well.

The ROM next selects a boot device. The default boot device is the Twiggy drive 1, but this can be overridden by the keyboard or by parameter memory. The ROM passes the memory size, the boot device position, and the results of the diagnostics to the loader found on the boot device.

The loader allocates physical memory and loads three types of Operating System segments needed during Startup, including the configurable device drivers. It creates a pseudo-outer-process, enters the Operating System, and passes to Startup a physical address map and some parameter data.

Startup inherits the unmapped address space of the loader, initializes the memory map, initializes all the Operating System subsytems, creates the system process, then destroys the pseudo-outer-process (itself), passing control to the highest priority process. At this point the boot process is complete and the outer shell process or the Filer is in control.

SELF-DIAGNOSTICS

The self-test code in ROM performs an overall diagnostic check at power-up and then executes the bootstrap routine from the disk.

The first tests initialize various system controls; MMU registers, contrast control, parity logic, etc. You should hear a beep notifying you that the startup tests have begun. A checksum is done on the ROM itself, then all of the RAM in the system is tested for shorts and address uniqueness. The Memory Management Unit is also tested in this manner.

Parts of the video and parity generator/checker circuitry are tested next. The keyboard and mouse interfaces are tested by checking various modes of the Versatile Interface Adapter operation, and by running a ROM/RAM test of all the processors used in the interfaces. Meanwhile, the disk controller is running its own tests of ROM and RAM. Finally, the RS232 port and the clock are tested.

CUSTOMIZING YOUR SYSTEM

The features and design of the system configuration program have not yet been defined.