Xenix and the Motorola 68000 family

Bill Bateson looks at Xenix and Xenix 68000

The main features of the Xenix time-sharing system are discussed, paying particular attention to those aspects of the operating system which relate most closely to the hardware on which the system is running. In particular the architecture of the CPU, the memory management hardware, and the disc system are discussed.

microsystems Unix memory management

Bill Bateson is a systems programmer in the software products group of UK firm Logica. He started his career in the computing industry in 1979 with SPL International working on reactor simulation programs for the UK Atomic Energy Authority. Bateson is a graduate of Bristol University, UK, with a degree in mathematics and computing science. He is currently engaged in adapting Xenix to run on various 68000-based microsystems.

WHAT IS XENIX?

Xenix family tree

Xenix is an enhanced version of the Unix time-sharing system, and its relationship to Unix can best be illustrated by means of a family tree (Figure 1).

Unix itself was first written and implemented by Ken Thompson (of Bell Labs) on DEC PDP 7 in 1969. His motivation to write the system was a general dissatisfaction with the available computer facilities. With Unix, he aimed to produce a system that provided a powerful development environment for programmers geared towards interactive use. This first attempt was successful enough to gain the interest of the second godfather of Unix, Dennis M Ritchie. Between them, they developed Unix from its first single-user version to the commonly available multiuser system.

The essential features of Unix are as follows.

A hierarchical file system with demountable volumes A Unix file system may be considered as a tree structure containing three types of node (Figure 2).

- Ordinary files
  These can logically be regarded as one-dimensional arrays of characters.
- Directories
  These are the branching points on the tree structure. They contain the information necessary to map the names of the files immediately below them onto the files themselves. This information is in the form of a simple pointer which says where to find the disc block(s) belonging to the file. Because it is simply a pointer, an identical copy of the pointer can appear in another directory, which gives rise to the concept of a link, ie a single file with two entries in the directory structure. Directories are stored in the same way as ordinary files, but cannot be written to by unprivileged programs, so that the system controls their contents.
- Special files
  These are the most unusual feature of the Unix file system. Each I/O device on the Unix system has an associated special file. These files can be read or written to just like an ordinary file, but the request results in activation of the specified device, causing data to be transferred without reference to any file system structure. Thus to write to a floppy disc, for example, one may write to the special file associated with the floppy disc drive exactly as if it were an ordinary Unix file.

The root of the file system is the / or 'root' directory, and this directory is always held on a particular device

Software Products Group, Logica UK Ltd, 64 Newman Street, London W1, UK
Presented at *Applying the 68000 family*, City Conference Centre, London, UK, 30 October 1984

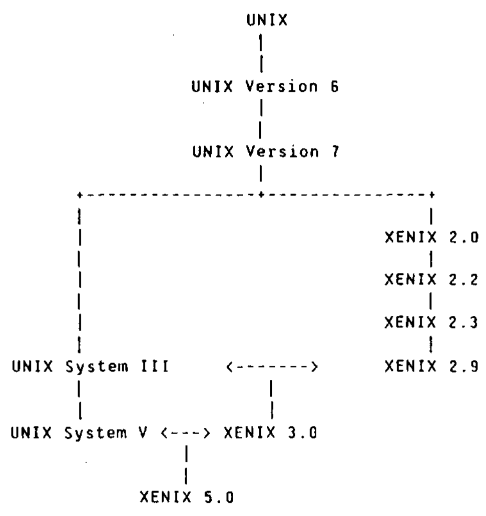

Figure 1. Xenix family tree

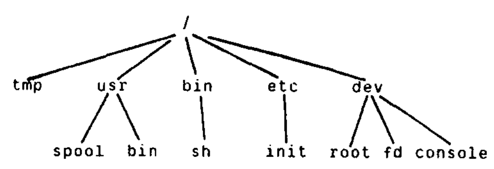

Figure 2. Unix hierarchical file system

for a given Unix configuration. However, it is not necessary that the whole file system hierarchy should reside on that same device, and it is possible to mount other file systems on other devices below any empty directory on a file system. For example, the subtree below the /usr directory is usually held on a separate logical device from the remainder of the tree, but when it is mounted as above, the fact that it is held on a separate device is completely transparent to the user.

Compatible file, device and interprocess I/O All I/O calls have the same form whether they are to files, devices or other processes. The file is first opened by the open() system call

filep = open(name, flag)

where name may be an ordinary file, or the special file relating to the device to be opened, and flag indicates the mode required for the file (read/write/append/etc.). filep is a file descriptor used in all subsequent accesses to the file.

filep can now be used as arguments to the read(), write() and other I/O system calls in place of the file name.

If communication is required with another process, the pipe() system call is used to obtain two file descriptors (one for writing to the pipe, and a second for reading from it)

pipe(fielpr, filepw)

Thus, a process will write to the pipe by writing to the filepw file descriptor, and another process can read from the pipe by reading from the filepr file descriptor.

System command language selectable on a per user basis Communication with the system is carried out by a command-line interpreter program, or shell. This is essentially a program capable of reading and executing commands from a terminal. The standard shell generally associated with Unix is the Bourne shell, but this is not hard wired into the system, and can easily be changed so that each user could have his/her own shell. For example, to allow users restricted access to a system, a shell could be devised to limit the number of commands available to the user. A particular example of this could be to make a user whose shell was the chess program; anyone logging into this user could then have access only to the chess program.

A wide range of software tools and compilers The Unix system includes a large number of programs useful to Unix users such as

- system maintenance utilities
- program debuggers
- text formatters
- editors

as well as C, FORTRAN and many other compilers.

A high degree of portability The Unix kernel is almost entirely written in the C programming language. This means that once a C compiler has been written for a new CPU, the problem of porting Unix is a far simpler task than it would be for an assembler-written operating system. Furthermore, if the port has already been done to a particular CPU and memory management unit (MMU), the task of adapting for a different disc or serial port becomes relatively straightforward.

It is this portability that has led to the proliferation of Unix and its current availability on a large number of machines.

The first version of Xenix was essentially AT&T Unix version 7 with many bugs fixed and the first Xenix enhancements, eg automatic file recovery (the file system is automatically checked and corrected on startup after a system crash). This was followed in 1982 by Xenix 2.3 which included further Xenix enhancements such as record locking (which enables an arbitrary number of bytes at an arbitrary position in a file to be locked) and semaphores (for improved interprocess communication).

More recently Xenix 3.0, based on AT&T Unix system III, has been introduced. This is a significantly improved version, in terms of both its usability and its functionality. New features include better system administration utilities, the menu-driven visual shell, improved text processing facilities, and shared data.

Microsoft has plans to produce a further version of Xenix compatible with Unix system V, although the exact nature of this product is as yet unreleased.

Aims of Xenix

Whilst Unix has had a rather haphazard history during which little effort has been made towards making it a coherent commercial package, Xenix has always been aimed at the commercial user. In particular, all releases have been made upwards compatible, so for example all programs that run on Xenix 2.3 will run unmodified on Xenix 3.0 — this applies to binary as well as source programs. Xenix also forms part of an integrated product range Microsoft, and can be used in conjunction with other Microsoft products such as Windows and MS-DOS.

Finally Xenix aims to continue its domination of the Unix market (at present 77% of all installed Unix and Unix-like systems are Xenix), and expects to expand its present 70 000 installed systems to 250 000 by the end of 1985.

WHAT DOES XENIX REQUIRE FROM ITS HARDWARE?

Ideal Xenix machines

For Xenix to run successfully on a system, there are certain requirements of the hardware configuration some of which are necessary, and others simply desirable. The following sections describe in general terms the requirements from each of the basic components of a computer system on which it is intended to run Xenix.

CPU This must be able to run programs in two distinct modes: supervisor mode, in which no restrictions are placed on an executing program, and user mode, in which the following restrictions apply

- certain instructions are not executable (eg the instruction to change between user and supervisor modes or any instructions related to interrupt handling)
- the program is not permitted to access I/O devices (this may be implemented by the memory management unit)

In the case of the Intel 8086 chip, supervisor/user modes are not implemented on chip and must be emulated by external hardware.

It is necessary for the CPU to be able to access a physical address space of at least 512 kbytes (this is the minimum main memory requirement for Xenix 3.0), plus any further addresses required to map I/O devices.

The 68000 family fulfills both of these requirements, having 16 Mbytes of logical address space, all of which is directly accessible (many other chips have segmented architectures capable of easily accessing only 64 kbytes at a time).

A desirable feature (especially for CPUs with limited direct addressing) is a set of output signals which enable the MMU to determine the type of memory access being performed. This enables the MMU to translate the addresses it receives differently for the four possible combinations of user data, user program, supervisor data and supervisor program.

The interrupt handling on the CPU is also an important factor in the performance of the system. Although Xenix can be implemented on a system without interrupts from the I/O devices (provided there is a source of clock interrupts to the CPU), it is extremely inefficient. If the devices do not interrupt when they need to be serviced, Xenix must regularly poll them to determine their state. For serial lines, this means a very high rate of polling if the normal transmission rate of 9600 baud is to be maintained, and similarly the disc device must be polled regularly if it is to achieve a reasonable throughput. The process of polling devices is very wasteful of CPU time, and any system without interrupts will be extremely slow compared with a similar system operating with interrupts.

MMU Essentially, the requirements of the memory management system are as follows

- relocation — the MMU must be able to relocate a program with fixed addresses anywhere in main memory
- protection — the MMU must be able to protect areas of memory from being accessed and, in particular, from being written to. Should an access be made to nonexistent or protected memory, the MMU should cause some sort of trap to be generated on the CPU.

The MMU system will generally be set up so that when the CPU is running in user mode, the addresses output by the CPU will be mapped by one set of mapping registers, and when it is in supervisor mode, a second set of registers will be used. In both cases, a program executing in supervisor mode should have unrestricted access to all memory. The effect of this is to allow the Xenix kernel (which operates in supervisor mode) to access all memory, while the user programs running under Xenix (which run in user mode) can only access the memory allocated to them by the Xenix kernel.

Main memory At least 512 kbytes of RAM are required for a Xenix 3.0 system. Xenix itself requires approximately 130 kbytes; the remainder is required to allow several processes to be simultaneously mapped, thereby helping to minimize swapping.

Discs A 10 Mbyte hard disc is the minimum requirement for a small Xenix system. The disc is divided into two distinct areas: the swap area (used to hold copies of processes whose execution has been suspended so they can be swapped out of main memory to make way for higher priority processes), and the Xenix file system area (this will be further subdivided on most systems) which holds the Xenix software and user programs.

Backup device Some form of backup device is required if user and system files are to be protected against hardware or software malfunction. This will usually be a magnetic tape or floppy disc drive.

ROM monitor Except for systems that are to have Xenix adapted to run on them (see the section on Xenix adaptions), the only requirements of a monitor are to be able to read and execute the primary bootstrap program (usually held on the first block of the main disc). This program will then cause Xenix to bootstrap itself into life.

Memory management on Xenix

At any one moment during normal use of a Xenix system, there exist in memory a number of user programs, or processes, at various stages of execution. In order that these processes can coexist without interfering with another, the memory management system must enforce strict protection to allow an active process access only to the limited region of memory assigned to it. Any accesses outside the allocated regions must be trapped and appropriate action taken according to the exact nature of the violation.

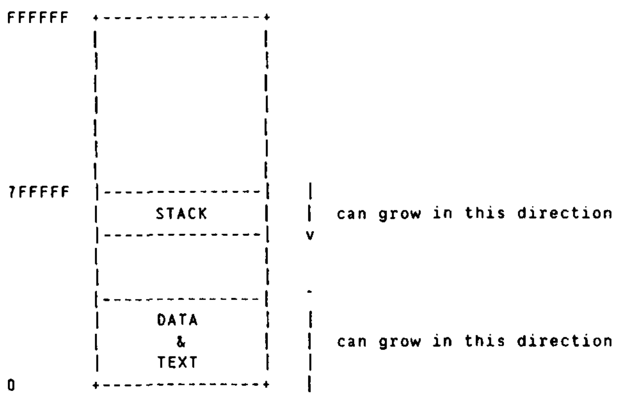

Figure 3. Normal executable structure

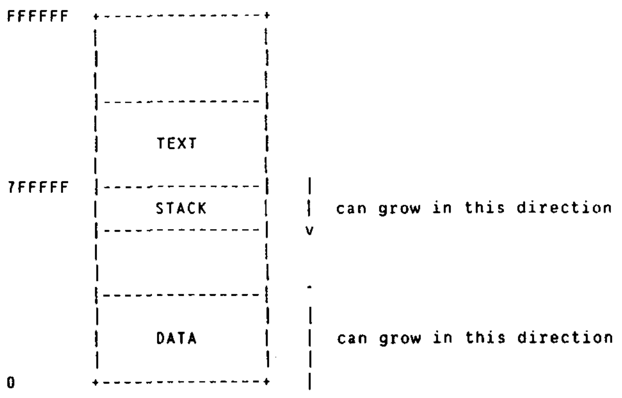

Figure 4. Pure text structure

To understand why the memory management system works as it does it is first necessary to give some background of how Xenix structures its processes. The term process is used to describe a compiled and link-edited program together with the associated system table entries required to sustain the program as it runs. All programs that run under Xenix 68000 are structured in one of two ways.

Normal executable Under this scheme the virtual address space for the program is split into two independent regions: the first containing the program text and data for the program, and the second containing the stack (see Figure 3).

Pure text Under this scheme, the virtual address space of a program is divided into three distinct regions: the first containing the program text, the second containing the data and the third containing the stack (see Figure 4).

The text segment is set to be 'read only' by the MMU so that it can be shared among a number of processes without danger of being corrupted. Thus for a commonly used program, such as the shell, only one copy of the text segment is held in main memory at any one time; after the first shell becomes active, all other instances of the shell will only require data and stack segments to be mapped, as they will share the first shell's text segment.

The precise addresses at which these regions start may vary between different implementations of Xenix 68000, but will be fixed for a particular implementation.

Under Xenix both the stack and the data regions can be grown beyond their initial allocation. In the case of the stack, the growth occurs automatically when the bounds of the allocated stack are exceeded. The data region, however, can be grown by user processes using a system call to the Xenix kernel.

The facility that enables Xenix to grow these program regions is provided by the MMU system and an example of how this works follows.

First consider an example of how Xenix maps the logical address space of a process onto physical memory. Figure 5 shows two processes A and B on the left, and on the right are shown the positions of the regions which comprise the processes as they are mapped in physical memory. Note that the regions are fairly well scattered around physical memory, which will generally be the case on a system with a normal multiuser workload. For the purposes of this example it has been assumed that each region belonging to a process is contiguously mapped — we shall see later that this is not always necessary. To illustrate the problem of growth, we shall consider how each region grows its stack.

Process A will make a stack reference below the base address of its stack region (eg as a result of a C subroutine call). This will be detected as an invalid reference by the MMU, causing a trap which suspends the user process and switches the CPU to supervisor mode. On discovering that the reference was below the stack (a valid way to grow the stack), Xenix will first check whether the physical memory below the currently allocated stack region is free. In this case it is, so Xenix will allocate a further chunk of memory from the current top-of-stack position to a suitable point below the referenced address to allow for further growth. When the memory management has been set up to map this new larger region, the trap can be terminated and the user process resumed.

If the same thing happens to process B however, Xenix will discover that the memory immediately

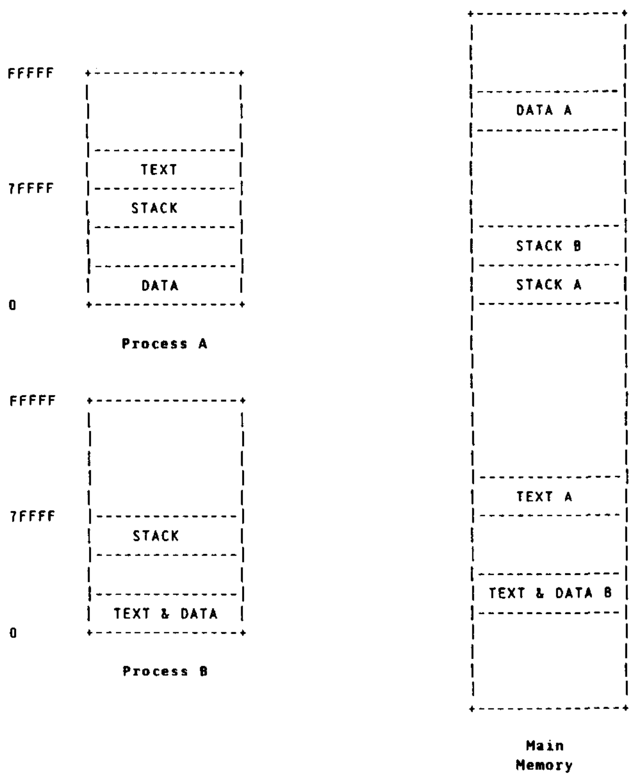

Figure 5. Xenix mapping logical address space onto physical memory

below the current stack is in use (by process A). In this case, the only way to grow is to swap out the stack region onto the swap disc, allocate a new larger region (say below the Text and data region for process B), and swap that stack area back into the new larger region, continuing execution from there.

The exact method by which Xenix maps its processes depends on the MMU in question. For the purposes of this document, we shall consider the two most common MMU systems in use on 68000 Xenix systems which implement the two basically different approaches to memory management. These are the MC68451 MMU which implements a segmented scheme, and the paged MMU scheme (an example of which is used on the Sun 68000 board).

Segmented scheme using the MC68451 MMU

The 68451 MMU is capable of mapping segments of memory which are powers of 2 in size, with a minimum segment size of 256 bytes. Each MMU has 32 descriptors, each of which can be programmed to map a logical segment of size 2^n (which must start on a logical boundary which is itself a multiple of 2^n) to a physical segment of the same size starting at any physical address which is also on a boundary which is a multiple of 2^n. Although these restrictions may seem to be fairly severe, in practice there is a scheme which makes reasonably efficient use of the MMU. This scheme is known as the 'buddy' system and works as follows.

Physical memory is divided into a number of equally sized pages. Let us assume that these pages are 2 kbytes in size.

Looking at Figure 6 we can see that memory can be divided up into pairs of pages, or buddies. Each pair of buddies can be merged to form a single unit, which will in turn have a buddy.

This process of merging buddies continues until all memory has been accounted for by a few (usually one or two) large segments. For a 768 kbyte main memory when all possible buddies have been merged, for example, there will be one 512 kbyte segment and one 256 kbyte unit.

As memory is used, these large segments become split into their constituent buddies and the free segments released are kept on free lists, one for each size of segment. So for the above example we would have a free list for segments of size $2^{11}, 2^{12}, 2^{13}, \ldots, 2^{19}$.

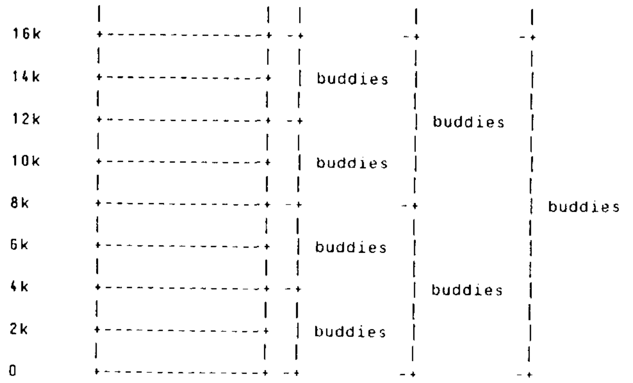

Figure 6. Memory divided into 'buddies'

To allocate a region of size 2^n, we must first search the free-segment list for a segment of that size. If no such segment exists, we look for a segment of size 2^{n+1}. If we find one, it can be split into its two buddies, each of size 2^n. One can be used, and the second returned to the free list for segments of size 2^n.

Had we had not been able to find a segment of size 2^{n+1}, we could have tried to find a segment of size 2^{n+2} and split that down to the required 2^n segment.

If we reach the maximum-segment-size free list and find it empty still without having found the required segment free, we must swap out a process, free its memory (merging any buddies if possible), and try again. Provided there are large enough segments to map a process when all memory has been freed (apart of course from the memory occupied by Xenix itself), then we shall always be able to map a process in this manner.

The problem with this system is that it requires a fairly clever algorithm to make best use of free memory and, however well they are implemented, a lot of CPU time can be used in deciding the best way to map a process. Conversely, if the algorithm is too simple, memory will be wasted resulting in excessive swapping.

Having said this, the system does work well in practice. It also has the distinct advantage of being a standard MMU on one chip. This makes implementing Xenix on a system with a 68451 far simpler than if a nonstandard MMU is used.

Paged memory management

The paged system of memory management overcomes the problem of memory fragmentation and can be implemented with much simpler software. The basic principle is as follows.

Memory is divided into a number of equally sized pages, say for example 2 kbyte pages. These pages are maintained on a single free list, or they are attached to a list belonging to the process to which they are allocated.

The logical address space too is divided up into 2 kbyte pages, and each logical page can be mapped onto any physical page by means of a set of mapping registers — one for each page in logical memory. These registers are usually in very fast RAM to minimize the overhead of mapping pages.

The problem of allocating memory in this system is far easier than in the segmented scheme. We simply take as many pages from the free list as are required. If there are not enough pages available, a low-priority process can be swapped out and its pages returned to the free list, making them available for the process requiring memory.

Xenix disc system

As stated before, a Xenix disc performs two separate functions and is consequently divided into two logically separate regions. The first region is reserved for Xenix file systems, and the second is used for swapping. Because of their different functions, each region demands different qualities from the disc.

The swap area is usually accessed in a sequential manner with multiple sector transfers being the norm. Consequently a high transfer rate is desirable to optimize swapping.

The Xenix file system, however, is structured in such a way that the reading of a file can require several reads, none of which is likely to be in consecutive blocks. As a result, a fast seek time is desirable to obtain the best performance for this area.

Xenix 68000 regards its discs as a sequence of directly addressable 512 byte blocks. The disc controller should ideally be capable of initiating a read or write on a block with a single command, ie the controller should implement implied seeks.

Ideally the controller will also be able to perform direct memory access (DMA). This saves part of the time required to transfer blocks between memory and the disc controller (although the bus will still be taken up while the copy is in progress).

IMPLEMENTING XENIX

Extra requirements for Xenix adaptations

To adapt Xenix to a new machine, there are a number of additional requirements to those stated in the previous section.

The only hardware requirement is that there must be at least two serial ports on the system. The first must be configured as the console device for communication with the monitor (and eventually Xenix), and the second must be configured as a serial link to the host computer which is used to develop, compile and download prototype Xenix kernels to the target machine.

From the software angle, the machine must be provided with a reasonably sophisticated ROM monitor program which must incorporate at least the following abilities

- to read data (ie programs) sent from the host down the serial link, and to transfer it to any given memory location
- to display and alter memory
- to display and modify the CPU's registers

The following features are required to make debugging simpler

- a method of setting breakpoints
- a monitor command to read and write a specified number of blocks between main memory and a specified section of the disc
- the ability to trace the execution path of a program over a specified number of instructions
- a disassembler

All the above features are provided by the VERSAbug monitor which is supplied as part of the Motorola VERSAmodule board.

Some interesting Xenix adaptations

The final part of this discussion is devoted to outlining a few of the Xenix 68000 adaptations that Logica has done over the last year.

Eagle

This system is a 68000-based single-user system aimed at the office market. The memory management system is of the paged variety, and the discs used are a small (10 Mbyte) Winchester and a 5¼-in floppy drive.

The interesting feature of this system is that all accesses to the I/O devices and the memory management system are performed via calls to a set of monitor routines. Consequently the resultant Xenix system is independent of the exact hardware on which it runs provided the monitor interface to the devices is the same.

Fix

This is also a single-user system which has a similar principle of accessing devices by means of monitor calls. However, the CPU is a 68010, the MMU is a 68451, and the application is rather unusual for a Xenix system. It is used as a host system to monitor and control an intelligent subsystem which automatically performs chemical analysis of various substances.

The host system automatically downloads the control programs to the subsystem, waits for the analysis to be completed, and then processes the results of the analysis producing a printed report. The advantage of having Xenix on the host system is that it provides an ideal environment for maintaining and developing the programs used to perform the analysis.

Beta

This is a high-performance dual-processor system aimed at the general commercial market. It features a tightly coupled 68010/68000 pair of CPUs with two 68451 MMUs and a 16081 floating point processor.

The 68010 is the main processor and runs a specially adapted version of Xenix, whilst the 68000 is the I/O processor (see Figure 7). The I/O processor runs a custom-built multitasking system consisting of a control task and a number of device driver tasks which compete cooperatively for the CPU.

All the external devices (except for a hexadecimal LED display used for diagnostic purposes) are connected to the I/O processor, and the two processors communicate by a combination of shared memory and interprocessor interrupts.

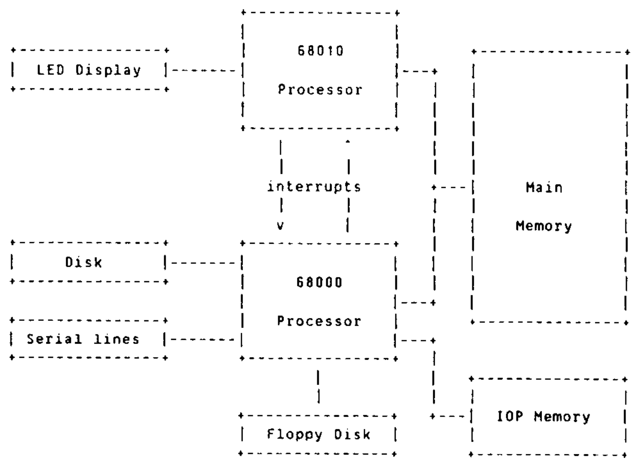

Figure 7. Beta dual-processor system

Brutus

This is also a dual-processor system aimed at a similar market to Beta and is produced by Sagem (the largest non-state-owned electronics company in France). The main differences are in the way in which the two processors communicate and in the function of each processor. The main processor (a 68010 with 68451 MMU) drives most of the I/O devices (disc, tape, floppy and two serial lines), and the I/O processor handles the rest of the terminal lines. Interprocessor communication is achieved by a simple interface which allows the main processor to pass characters to and from the terminals attached to the I/O processor.

CONCLUSIONS

Processors in the Motorola 68000 series are cheap, and are available in large quantities. Leaving aside the competition from the Intel 80286 and National Semiconductor 32000 series, the 68000/68010 processors provide the best basis for a fast and efficient Xenix implementation.