PORTABLE COMPUTERS - PORTABLE OPERATING SYSTEMS

D. Wiegandt
CERN, Geneva, Switzerland

1. Portability

> Portable: Capable of being carried by hand or on the person; capable of being moved from place to place; easily carried or conveyed.
>
> The Oxford English Dictionary

1.1 Portable hardware

Hardware development has made rapid progress over the past decade. Computers used to have attributes like "general purpose" or "universal", nowadays they are labelled "personal" and "portable". Recently, a major manufacturing company started marketing a portable version of their personal computer. But even for these small computers the old truth still holds that the biggest disadvantage of a computer is that it must be programmed, hardware by itself does not make a computer.

> "Software stands between the user and the machine."[1]

For every major innovation a manufacturer of microcomputers is faced with the problem to provide software that can be used with his new hardware. To produce an operating system from scratch is a costly task often involving large teams of software engineers. Manufacturers therefore prefer to have existing software adapted to their specific product, "ported" to the new hardware.

1.2 Portable software

A piece of software is usually called portable if the effort to transport it is much lower than the effort to rewrite it for a new environment. There are several possible approaches to transporting software [1] :

1. Write a program that emulates an existing computer on the new hardware and then simply run the application programs or even the operating

---

[1] Harlan D. Mills

system of the older machine on the new hardware. This approach was taken by IBM when they introduced the Series /360 by providing emulators for the 7090 and the 1401, and in more recent times by a company called Dynamic Microprocessor Associates who offer EM 80/86, an 8080 emulator to run under CP/M-86 on Intel 8086 microprocessors. In spite of its success, this method is usually hampered by a severe performance penalty.

2. Define an instruction set and an architecture of a virtual machine. Then, write an operating system and all the necessary programming tools like compilers, assemblers, loaders, etc. for this virtual machine. All that is left to be done now is to write target-specific emulators that emulate the virtual machine on the new hardware. This is the approach taken by manufacturers providing the UCSD[2] p-System, which is based on the UCSD implementation of PASCAL, but also supports BASIC and FORTRAN. All programs are compiled into "p-code", which is then executed by an emulator on the target microprocessor. The UCSD p-system is available on some 9 processors, including Z80, 8086, 68000 and LSI-11. There is still a performance penalty associated with this approach, but a clever choice of the intermediate "higher level" language helps substantially to reduce the overhead imposed by emulation.

3. Write the whole programming environment, i.e. language translators, operating system, utilities and so on in a higher level language, carefully isolating machine-dependent parts. Transporting this system to a new environment then amounts to

    a. writing a code generator for the high level language translator to generate code for the new target,

    b. rewriting the machine dependent parts of the operating system,

    c. finally recompiling the whole system using the adapted compiler.

This description is, of course, oversimplified, but it describes the principle of porting UNIX to a new environment. More details of this process can be found in [1] [2] [3] .

2. Operating systems for portable computers

Q: What is an elephant? A: An elephant is a mouse with an operating system.

Folk saying

---

[2] University of California at San Diego

2.1 Elements of operating systems for 8-bit microcomputers

The above folk saying is unfortunately not at all helpful to understand the purpose of an operating system. Here is a better definition:

> An operating system is an organized collection of programs and data that is specifically designed to manage the resources of a computer system and to facilitate the creation of computer programs and control their execution on that system [4] .

Operating systems usually take over when the computer is initialized, and the first manifestation of the operating system's running is some prompt appearing on the screen. This prompt usually signals that the system is waiting for the user's commands. Here we have identified one principal task of every operating system: the interpretation and execution of the user's commands.

Usually a well defined module of the operating system is assigned to this task. Let us call this module CCP, Console Command Processor. To display the prompt on the screen and to read the command from the keyboard, CCP uses the services of another important component of the operating system, a module handling basic input/output operations. Let us call this module BIOS, for Basic Input/Output System. A microcomputer usually has some secondary storage, in general floppy disk units. Handling of disk accesses and file management are delegated to a third module, called BDOS for Basic Disk Operating System.

These three modules, CCP, BIOS, and BDOS are the essential components of a very popular microcomputer operating systems, CP/M (Control Program for Microprocessors), which was written in 1973 by Gary Kildall for the Intel 8080 microprocessor. As we shall see, control program is a better name than operating system for CP/M.

CP/M fans even call it a portable operating system, because "all you need to do to adapt CP/M to new hardware is to adapt the BIOS module to the new environment". This statement is obviously only true when you continue to run on a processor that executes 8080 machine instructions, e.g. a Z80. So the portability of CP/M is comparable to the one of MVS, IBM's most important operating system, which also runs on SIEMENS, Fujitsu, Amdahl, etc.

Looking at what we can do with CP/M, we soon realize that it is a single user single task system, which may just about satisfy the needs of a user of an 8-bit microcomputer, but it will certainly fail to cope with the requirements of a user of a 16-bit micro.

Before we go on to 16-bit systems, let us summarize, which functions of an operating system we have identified so far:

1. Command interpretation

2. Input/output handling

3. Implementation of a file system

4. System services

By the term "system services" we shall understand the provision of services of more general nature by the operating system, for example the conversion of the contents of a counter, driven by a quartz oscillator or by the line frequency, into a printable ASCII string containing date and time.

2.2 Elements of operating systems for 16-bit microcomputers

So far, we have dealt with 8-bit microcomputers. If we switch our attention now to 16-bit processors, we may ask the question what new requirements for an operating system arise from this hardware evolution. The first obvious improvement is the increased addressing range of 16-bit processors. With 8-bit micros, operating system and user program could not occupy more than 64k bytes of memory. This limit has been pushed to 1Mbyte for the most restricted 16-bit chip, the Intel 8086, other chips have an even bigger addressing range. So the operating system must provide means to enable a user to efficiently use more memory, i.e. memory management facilities.

The processing power of 16-bit chips has also increased significantly compared to 8-bit chips. To make efficient use of this additional processing power, the operating system should at least provide multiprogramming capability, or even allow multiuser operation. Most of the currently popular 16-bit operating systems do not even provide true multiprogramming, MS/DOS, for instance, only allows a printer spooler to run in the background concurrently with a single user program.

A third important area is the handling of sophisticated peripheral equipment. While 8-bit micros usually have floppy disk equipment attached that allows secondary storage amounts of a few megabyte, modern 16-bit computers are delivered with Winchester disk drives with capacities of a few hundred megabytes, and the number of workstations equipped with a high resolution bitmapped display unit is increasing continuously. This evolution has to be accompanied by a corresponding evolution of the I/O handling facilities of the operating systems.

Let us summarize the additional requirements of more advanced hardware:

5. Memory management

6. Process management

7. Improved file systems and I/O facilities.

We shall now try to deepen our understanding of operating systems by studying a very interesting system that is getting more and more popular in the area of microcomputers: UNIX.

3. A portable operating system: UNIX

3.1 History

Whoever attempts to tell the history of the UNIX system describes the situation in 1969 at Bell Laboratories that gave birth to UNIX: Scientists were frustrated by the failure of the Multics project, a collaboration between Bell Laboratories, General Electric and The Massachusetts Institute of Technology, from which Bell Laboratories had withdrawn. The computing facility at their disposal was a time-shared central computer, probably with rather bad turnaround times even for small programs.

One of these programs was "space travel", a program simulating the movements of the planets in the solar system, whose authors, Ken Thompson and Dennis Ritchie decided to use an obsolete PDP-7 with a CRT display unit to run this program interactively. Unfortunately the PDP-7 provided only an assembler and a loader, so an operating system, originally a single user system, was written by Ken Thompson. The start as a single user system may also be an explanation for its name, UNIX, as opposed to Multics, which was meant to become the state-of-the-art multiuser system.

The history of UNIX is tightly related to the history of the language C. UNIX was originally written in PDP-7 assembler, then rewritten, again in assembler, for a PDP-11/20 during 1971/1972. Thompson had been working on the language B, a descendant of BCPL. Attempts to use this language to rewrite the system failed, even after the introduction of variable types in the previously typeless language. After Dennis Ritchie had added structures and global variables in 1973, the new language became known as C, and a rewrite of the system in C was successful. The first publicly available version of UNIX, the Sixth Edition, was released in 1975.

Since then, UNIX evolved in different flavours on different hardware. Two major streams can now be distinguished:

1. AT&T UNIX System V, based on Bell Laboratories Seventh Edition UNIX is available for DEC Hardware and several microprocessors, e.g. Motorola 68000.

2. Berkeley 4.2bsd UNIX, based on a former VAX version of Bell Laboratories, has been enhanced with respect to that version by the implementation of demand paging and a number of useful features and utility programs. The ULTRIX system announced recently by DEC for VAXES is based on Berkeley 4.2bsd.

For more details on the history of UNIX see for instance [5] .

3.2 The UNIX file system

The UNIX file system was the first component of the system in the development process [5] . It did not yet have exactly the same structure as today, but one principle was already firmly established:

> A file is a sequence of zero or more bytes. All additional structuring information has to be imposed and recognized by application programs.

Three categories of files are known by the UNIX file system:

1. Ordinary files
2. Directories
3. Special files

3.2.1 Ordinary files

As we have seen from the above-mentioned principle, an ordinary file is a string of bytes. An application program reading a text file will of course recognize a line structure, the convention being that all lines in a text file end with a newline character. The assembler generates, and the loader expects object files of a specific format. But in contrast to most other file systems this structure is entirely managed by the application program and not imposed by the operating system. There are no such things as records, blocks, blocking factors, blank padding and so on. Physical structuring imposed by the hardware (e.g. disk sector sizes) is completely hidden from the user.

3.2.2 Directories

A directory is the place where the link between a file name and a physical file on disk is kept. Each user has at least one directory of his own files, his "home directory". He may create any number of subdirectories to group related files together in a convenient way.

A directory is in principle an ordinary file containing directory entries and can be read like any other file, appropriate permission assumed. Modification, however, is only possible via system calls. In this way the system controls the contents of directories.

The structure of a UNIX file system is a rooted tree, the nodes being directories, the leaves being ordinary or special files. In standard UNIX (as opposed to distributed UNIX) each directory must appear as an entry in exactly one other directory, which is called its parent directory. There is one exception to this rule: the root directory (/), which is its own parent.

Each directory has at least two entries, the entry "." that points to the directory itself and the entry ".." that points to its parent directory.

When a user is logged on to a UNIX system, he is always associated with one directory or another, the working directory. The working directory is an attribute of every UNIX process. When you first log on, your home directory is your working directory, but provided you have the necessary access privileges you may change your working directory to any place in the hierarchy.

Files have names of 14 or less characters (Berkeley UNIX permits up to 255 characters in file names). Files may be specified to the system in the form of a path name. A path name is a sequence of directory names separated by slashes (/) ending in a file name. If the very first character of the path name is a slash, the search starts in the root directory (full path name), if not, the search starts in the working directory.

It is possible for an ordinary file to have more than one link, i.e. to be known under different names. These links may even be in different directories. All links to a physical file have the same status, i.e. the existence of a file is not bound to a specific directory. The file disappears only when the last link to it has been removed.

3.2.3 Special files

Each supported peripheral device has at least one special file associated with it. Special files may be read and written just like ordinary files, but here, the transfer requests result in an activation of the associated peripheral device. Special files are entries in the directory /dev. So if you want to write to magnetic tape for instance, you write to /dev/mt. Special files exist for main memory as well as for all peripherals connected to the system - disk units (partitions), terminal lines, ... Special precautions are of course taken to prevent illegal access to main memory and disk units.

The application of standard file operations on peripheral devices has the following important advantages:

1. Data transfer between a program and an ordinary file and data transfer between a program and a peripheral device is as similar as possible.

2. File names and devices names have identical syntax and meaning. A program that expects a file name as an argument can be given a device name.

3. Peripheral devices are subject to the same protection mechanisms (see below) as ordinary files.

3.3 Nonresident file systems

UNIX allows mounting of nonresident file systems. The root file system, however, has to be resident, as it contains the boot image. The mount function takes two arguments:

1. the name of an existing empty directory

2. the name of a special file which is associated with a storage medium that contains an independent file system with its own directory hierarchy.

"mount" substitutes the empty directory with the root directory of the mounted file system. There is no distinction between files on any mounted file system and files on a resident file system. The only restriction that applies is that links cannot be established between one file system hierarchy and another. This restriction has been introduced to avoid the overhead otherwise incurred with the removal of all such links when a file system is dismounted. Berkeley UNIX circumvents this difficulty by the introduction of "symbolic" links, a kind of aliasing of file names by indirect access.

3.4 Access control

Each UNIX user has a unique identification, consisting of a user number and a group number. These numbers were assigned to him by the system administrator, when the user registered. When a user logs in the login program will set the numbers associated with this user after having read the corresponding entry in the password file.

When a user creates a new file, this file is marked with the user number and the group number of its owner. Associated with all files (ordinary files, directories and special files) is a set of nine protection bits that specify

1. read permission, write permission and execute permission for the file owner

2. read permission, write permission and execute permission for other members of the owner's group

3. read permission, write permission and execute permission for all other users.

Another bit is only used with executable files, it is called the set-user-ID bit.[3] Normally, a program is executed with the user identification of its caller. If the set-user-ID bit is on, however, the program is executed with the identification of its owner. This mechanism is by the way protected by a patent

---

[3] There is also a set-group-ID bit.

held by Dennis Ritchie.

A good example for the use of the set-user-ID bit is the program passwd, a program that anybody can use to introduce or modify his own password. Clearly this program needs write access to the system file that holds all information on registered users, but this file in turn must certainly be protected against unwanted modification. The strategy followed here is the following:

1. Create the passwd file as owned by user root, a user that is necessarily present in all UNIX systems and has user-ID 0.

2. Give it read access by everybody, because the encrypted passwords stored in the file cannot be decrypted anyway.

3. Give it write access only by its owner, i.e. root.

4. Create the passwd program as a file owned by root and executable by everybody.

5. Set the set-user-ID bit for this program, so whenever this program is executed it will pretend to be executed by root.

In this way you can assure that write access to the passwd file is limited. The identity of the actual caller of the program is also always available, so the called set-user-ID program can always verify access rights and take appropriate measures if necessary.

One particular user ID is exempt from file access constraints: the super-user. The system manager uses this feature to avoid interference from the protection mechanism for tasks like file backup and restore. Data that must not be read even by the super-user will have to be encrypted. There are system utilities to do that.

3.5 Input/output primitives

The lowest level of I/O is done by system calls. All Input/Output in UNIX is done to or from files, so the same system calls are used to do I/O to disk files, terminals, any other peripheral device. Some of the system calls may not be applicable to all devices, however, e.g. it does not make sense to do positioning on terminal input or to read from a line printer. I/O system calls are functions that return integer values.

To read or write an existing file, you have to open the file by

```
filep = open(name,flag);
```

where name indicates a path name (see above), flag indicates whether the file is to be read or written, and the result of a successful operation is a small integer value called a file descriptor, which is to be used to identify the file in subsequent I/O system calls related to the same file.

It is an error to open a file that does not exist. To open a new file or to overwrite an existing one, you use the creat system call:

```
filep = creat(name,perms);
```

where perms is the protection to be associated with the newly created file. Recent versions of UNIX provide a "Create if not existent" flag for the open system call, thus combining open and creat.

Reading and writing of files is normally sequential, there is, however, a way to read or write in arbitrary order by positioning in the file before transferring data (see below). UNIX maintains a pointer for each open file to the next byte to be read or written. If n bytes are transferred, this pointer advances by n bytes. Reading or writing is done by the following functions:

```
n = read(filep, buffer, count);
n = write(filep, buffer, count);
```

A maximum of count bytes is transferred between the file specified by filep and the array specified by buffer. The result n is the number of bytes actually transferred. For write operations this should always be equal to count, except in error cases, e.g. end of tape. The result n of a read operation can be less than count if only n bytes remain before the end of the file, n = 0 indicates that the end of the file has been reached.

There is a system function that permits to move the position pointer of a file to the desired location in the file:

```
loc = lseek (filep, offset, base);
```

The pointer associated with the file specified by filep is moved by offset bytes relative to base. Base can be the beginning of the file, the current position in the file, or the end of the file. The resulting offset from the beginning of the file is returned in loc. lseek can thus be used to determine the current position by specifying an offset of zero relative to the current position.

Another system function is the function

```
res = ioctl(filep, request, arg);
```

which is primarily used to control operating characteristics of peripheral devices, e.g. baud rates of terminal lines, echoing etc.

The function close serves to terminate the treatment of a file, unlink removes the entry of a file in a directory. Note that the file is deleted only after the last link to it has been removed.

UNIX does not have a mechanism for file locking at the user level, so any number of users can simultaneously read and write (!) a file. There are, however, internal interlocks to maintain the file system's consistency in cases where files are manipulated by more than one user [6] .

3.6 Implementation of the file system

3.6.1 Structure of a file system

The following description of the UNIX file system applies to Version 7 UNIX. The availability of disk units with capacities of several hundred megabytes has led to modifications of the file system implementation to improve the efficiency and the security of the file system. The file system implemented in Berkeley 4.2bsd for instance was heavily modified compared to Version 7. For details of these modifications see [7] .

The disk space of a UNIX file system is split in 4 segments:

1. Block zero which may be a boot block, is unused by the file system.
2. Block one, the so called super block, which contains the size of the file system and the boundaries of the other regions.
3. The i-list (index list), a sequence of i-nodes (index nodes), the number of which determines the maximum number of files in this file system.
4. Storage blocks

Each of the i-nodes in the i-list is a structure that may contain a file definition. The index of an i-node in the i-list, called the i-number, plus major and minor device number (see below) uniquely identify a file. A directory entry consists of an i-number and a file name.

Each i-node in turn holds the following information about a file:

1. user and group ID of its owner;
2. the protection bits;
3. the address of 13 physical blocks of the file;
4. its size in bytes;
5. time of creation, last use, and last modification, in GMT seconds;
6. the number of links to the file (how many times it appears in directories);
7. the kind of file it is (ordinary, directory or special file).

3.6.2 Access to blocks of a file

The addresses of the physical blocks of the file held in the i-node have to be interpreted in the following way (UNIX Version 7):

- The first ten addresses point to the first ten data blocks of the file. If a file is bigger than 5120 bytes, the eleventh address points to a block that contains up to 128 addresses of further data blocks.
- Even bigger files (> 70656 bytes) require double indirection via the twelfth address (<= 8459264 bytes) or even triple indirection via address 13.

This mechanism allows files of up to 1 082 201 088 bytes. Very large files thus require up to four disk accesses to get to a single data block. This overhead is eliminated in practice by a cache mechanism (see below).

To reduce the levels of indirection for files of sizes up to 2**32 bytes, the Berkeley 4.2bsd file system uses block sizes of 4096 bytes or greater [7] .

Only blocks that have been written into are actually allocated. There is a zero address for blocks that have never been used in the i-node structure, and null bytes are returned when you attempt to read such a block. In this way UNIX supports sparsely written random access files.

3.6.3 Creation and deletion of files

Files are created by the allocation of an i-node and the construction of a directory entry that contains the file name and the number of the i-node just obtained.

Making a link to an existing file amounts to creating a directory entry containing the new file name and a copy of the i-number of the existing file.

Files are deleted by decrementing the link count in the i-node and removing the directory entry. If the link count reaches zero, all disk blocks of the file are freed and the i-node is deallocated.

Free space in the Version 7 file system is maintained by a linked list of available disk blocks. Every block in the chain contains a pointer to the next such block and up to 50 addresses of free disk blocks. A single read operation is thus sufficient to obtain 50 free blocks and a pointer indicating where to find more space. Berkeley 4.2bsd UNIX introduces a further subdivision of a file system into cylinder groups to increase locality of access with today's Gigabyte disks. The information on free blocks in a cylinder group is kept in a bitmap.

3.6.4 Special files

The above description of i-nodes holds for ordinary files and directories. For a special file, only the first address in the i-node is used and this specifies

an internal device name consisting of a major and a minor device number. The major device number is used as an index into a table holding driver entry points, the minor device number is handed over to the driver and selects, for example, a particular terminal port or a particular disk partition.

3.6.5 Path name interpretation

During the execution of an open system call, the i-node corresponding to the path name given as an argument to open has to be found. Depending on whether the path name starts with a slash, the search either starts at the root i-node, for which device number and i-number (always 2) are known , or in the working directory, whose device number and i-number are also known as a result of the last chdir (change directory) call.

For every component of the path name, the corresponding i-node is found. If it is the last component, we have reached the final i-node, else it must be a directory, otherwise the path name is illegal.

There is one special case that has to be taken care of in this search: a file belonging to a mounted file system. Each intermediate i-node is therefore checked if it is a "mounted-on" i-node. If so, the mount table is accessed that holds a correspondence between a device number i-number pair and the number of the device on which the mounted file system resides. This number replaces the current device number and the i-number of the root directory (2) replaces the current i-number. From then on the search continues in the mounted file system's hierarchy.

3.7 Block and character I/O

The I/O system of UNIX is split into two parts: the block I/O system and the character I/O system. Thompson points out in [8] that these parts should rather have been called "structured I/O" and "unstructured I/O". In addition to the minor and major device number, each device is characterized by a class: block or character.

For each class there is an array of entry points of device drivers (configuration table), which is indexed by the major device number and the function to perform, e.g. read, write, open, close. These arrays of entry points are the only connections between the system code and the device drivers. It has turned out that this clean separation facilitates the creation of new device drivers enormously.

3.7.1 Buffering with Block I/O devices

Reading and writing appear to be synchronous and unbuffered to the UNIX user. But there is a fairly complex buffering mechanism provided by the system that greatly reduces the number of physical I/O-operations. A physical transfer

is only necessary as a result of a read or write operation, when the corresponding data block is not yet in the internal buffer cache of the system. Otherwise, the required number of bytes are copied to or from the data buffer of the program, and in case of a write operation, the system buffer is marked to be written out.

The system also recognizes when a program treats a file sequentially and asynchronously pre-reads the next block. In this way the running time of most programs is significantly reduced at the expense of little system overhead.

There are, however, some problems associated with this cache mechanism:

1. Error reporting and error handling are made quite difficult by the asynchronous nature of the physical I/O operation.

2. In case of a system crash it is very likely that there are I/O operations that are logically, but not physically complete, because the corresponding system cache blocks have not yet been written out. Use of a system primitive that periodically flushes all outstanding I/O activity does not completely solve this problem.

3. The physical I/O sequence does not necessarily correspond to the logical I/O sequence. This could have disastrous effects for writes on sequential devices like magnetic tapes, thus the number of outstanding writes to tape units has to be limited to one.

3.7.2 Terminal I/O as an example of character device I/O

For the handling of character-oriented devices the system provides character lists. A character list is a queue of characters, that may be filled by one program and emptied by another. There are system routines for the handling of these character lists that will take care of automatic storage allocation and deallocation, as the need arises.

Character output from a program to a device is implemented by passing characters from the user program to the appropriate output queue until a maximum is reached, at which point the user program is put to sleep (see below). Physical I/O is started as soon as there is something on the queue and is sustained by hardware transfer completion interrupts. When the number of characters falls below a certain intermediate level, the user program is woken up and may continue to fill the queue up to the maximum value.

For a terminal there is some additional code that provides optional special actions on certain output characters, e.g. delay introduction after carriage return or form feed characters, conversion of tabulation characters to an appropriate number of spaces etc.

A terminal has two character input queues, called raw queue and canonical queue. Characters typed on the terminal are put on the raw queue.

If the terminal is in its standard operating mode (cooked mode), input editing, e.g. treatment of backspaces and some handling of special characters will take place, e.g. the necessary actions on receipt of flow control characters xoff/xon, echoing of the input etc., but the program waiting for terminal input is not notified until the input line is complete. Upon receipt of a line termination character, the contents of the raw queue are copied to the canonical queue and the user program is woken up.

If the terminal is in raw mode, characters will be passed unmodified immediately to the program reading terminal input. This mode is used for special applications like full screen editors or special devices that use a serial interface like a terminal but require a particular treatment. Note that in raw mode no output processing is done either.

An intermediate mode is cbreak mode, where all input characters are immediately made available to the user program, but flow control and handling of interrupt characters are still enabled. Output processing is done as for cooked mode.

3.8 Processes

Ritchie and Thompson [9] define an "image" to be the current state of a pseudo-computer. This image comprises memory and register contents, open files, current directory and so on. A process is said to be the execution of an "image", which must then be in main memory. During the execution of another process, it may remain in main memory or be swapped out to secondary storage if necessary.[4]

The virtual memory of a process is divided into a user part and a system part. The user part can be further subdivided into:

1. a text segment which contains the program code, is write protected and is shared between all processes executing the same code;

2. a non-shared writable data segment;

3. a user stack.

There is also a small data area in system space associated with a process that contains data needed only when the process is active, like information on open files, stack area for the system phase of the process and so on. This area can be swapped out with the process.

---

[4] Berkeley 4.2bsd UNIX implements a different kind of memory management, "demand paging" [10] .

Finally, each process is represented by an entry in the process table, which is allocated at process creation and freed at process termination. Each process in the system has a unique process-ID associated with it.

3.8.1 Process creation

Processes are created with the system call fork. The only exception to this rule exists during the boot process, where some permanently running processes (swapper and init) are created "manually". The result of the execution of

```
process_id = fork();
```

is that the running process splits into two independently executing processes. Each process has its own copy of the original memory image, and all open files are shared. The only difference between these processes is that one is considered to be the parent process, the other one the child process. This is accomplished by fork returning twice, once to the parent process, furnishing a non-zero process_id, and once to the child process returning zero.

3.8.2 Execution of programs

A process may use the system call

```
exec( file, arg1, arg2,...,argn);
```

to read in and execute the program named by file. The string of arguments arg1 ... argn is passed to that program. All code and data of the calling process are replaced from the file, but open files, current directory and process relationships are not changed, nor is the process-ID. There is no return to the calling process from the exec system call except if the program to be executed could not be found or execute permission was not granted.

3.8.3 Pipes

Communication between related processes in UNIX can be achieved using "pipes". A pipe is a unidirectional communication channel. The system calls write and read are used for the communication. A child process will inherit pipes that a parent process has set up. Pipes are implemented as internal FIFO buffers. If a process reads a pipe, which is empty, it will wait, if a process writes into a pipe which is getting too full, it is also suspended, until there is room again in the pipe. If the process on the write side closes the pipe, the reading process will see the end of file condition.

This kind of interprocess communication is a very valuable tool which is mainly exploited by the shell, the UNIX command interpreter, for chaining commands together. It is, however, in its original form, limited to communication between processes that have a common ancestor. More recent versions of UNIX implement generally usable interprocess communication facilities [11] .

3.8.4 Scheduling and swapping

User programs in UNIX are executed by a user process. When the program requires a system function to be executed, it calls the system much as if it were a subroutine. This call uses the trap mechanism of the hardware to switch to kernel mode, thus to a new environment. From then on the process is said to be a kernel process. User process and kernel process never execute simultaneously. The user phase and the kernel phase of the process each have their own stack. Note that interrupt handling also forces an interrupted user process into kernel mode. Once all kernel activity has finished, a process returns to user mode.

Processes frequently cannot proceed unless some condition is true, e.g. an I/O operation has finished, a child process has terminated etc. Such process then waits for a specific event to occur. Events in UNIX are represented by arbitrary integers. There is a convention to use addresses of tables associated with those events, for instance, the termination of a child process is associated with the process table entry of the parent process. When a child process terminates, it will signal this event with the parent's process table entry address as an argument. Signaling an event for which no process is waiting, has no effect, signaling an event for which many processes are waiting will wake all of them up. As there is no memory associated with an event, waiting on an event which has already happened results in eternal sleep. This is one of the features of UNIX that render its use for real-time purposes somewhat difficult, and allow its use in multi-processor configurations only after substantial modification.

At any time, only one process in UNIX is executing, all others have called wait on event for one reason or another. Events these processes were waiting for may have occurred in the meantime, so many of them are "runnable". Whenever the executing process relinquishes the cpu or an interruption of some kind occurs, the next process to execute is selected. The selection is done based on priorities. The priority of a process in kernel mode is determined by the code that called wait on event, waiting on disk I/O is associated with higher priority than waiting on terminal I/O and so on. Kernel processes always have higher priority than processes in user mode. Priorities of user processes are recalculated every second based on the ratio between cpu time to real time since the last check. The higher this ratio, the more the priority is reduced. Interactive processes tend to have a low cpu time to real time ratio, thus interactive response is maintained without special arrangements.

Let us summarize the scheduling policy of UNIX:

- Processes in kernel mode are picked first and processes in user mode second.
- Priorities are recalculated every second. Looping user processes, all other conditions being equal, will be scheduled round-robin with a one second time slice.
- A high priority process waking up will preempt a running low priority process.

- No process can use its priority to hog the cpu, because its priority will drop. No low priority process can be ignored for a long time, because its priority will rise.

Process switching in fact involves yet another component of the system: the swapper. The swapper process, process 0, is one of the permanently running processes generated at boot time. Whenever a process switch is initiated, control is first transferred to the swapper process. The swapper process inspects the process table to find a swapped-out process that is runnable. This process is then loaded into memory and competes with the other processes for the cpu. If there is not enough free memory available to swap the selected process in, the process table is searched for candidates to be swapped out to disk to free a sufficient amount of memory.

Two selection algorithms are used by the swapper:

- The selection criterion for swapping in is the secondary storage residence time of the process, its priority and its requirements for primary storage, with a slight penalty for big jobs.
- Candidates for being swapped out are primarily jobs waiting for a slow event (e.g. terminal I/O). Selection then depends on the time of residence in primary memory, big jobs also being treated less favourably.

This area again is a somewhat problematic area of UNIX. Small systems with little primary memory tend to do a lot of swapping, which by itself is not so bad, but can lead to bad bottleneck problems when the swapping device is also the main file storage device. Big systems often have enough primary memory to avoid total swapping and can afford to have a separate swapping disk.

Reasonable performance of a UNIX system can only be expected when certain minimal hardware requirements are fulfilled, e.g. > 1 Mbyte of main memory, reasonably fast disks (access times < 100 ms) etc. A paper dealing with these minimal requirements is [12] .

3.9 The shell

UNIX differs from most other operating systems by not having its command interpreter integrated in the operating system kernel. The UNIX command interpreter is called the shell, because it encloses the kernel like a shell encloses its interiors. It is an application program without special privileges and can easily be replaced by another command interpreter. There are nowadays different "shells" available on UNIX. All UNIX versions have the original shell, often called the "Bourne shell" after its author S. R. Bourne. Most UNIX versions have the c-shell, written at UC Berkeley by William N. Joy, which utilizes a command language resembling the language C, and has other useful features [13] .

The simplest form of a UNIX command line is a command name followed by arguments for that command:

```
command arg1 arg2 arg3
```

The components of a command line are separated by spaces. The shell will search a file with the name "command". "command" is the name of a program or an executable file. It can be any valid path name (see above). If the file is found, it is loaded and executed. The arguments specified in the command line are accessible to the program that is being executed. When the program terminates, the shell regains control and prompts for the next command.

If the program to be executed cannot be found in the current directory, the shell attempts to find it in a user-definable set of standard directories.

3.9.1 Standard I/O and redirection

All programs executed by the shell start with three open files:

1. Standard input, file descriptor 0
2. Standard output, file descriptor 1
3. Standard error output, file descriptor 2.

These files normally correspond to the user's terminal keyboard and terminal screen. All UNIX utilities use this convention.

It is, however, very easy to change the standard assignments of these file descriptors from the terminal to any other file. If an argument to the command is prefixed by a ">" character, output will, for the duration of the command, be directed to the file named after the ">".

```
ls >myfiles
```

will list all files in the working directory and write its results into a file called "myfiles". This file is either created or its previous contents are overwritten.

Similarly, if an argument is preceded by a "<", input to the program will be taken from the file named after the "<", instead of being read from the keyboard.

File descriptor 2, the standard error output, usually remains coupled to the terminal screen to prevent error messages from disappearing silently into an output file.

3.9.2 Filters

A unique feature of UNIX is the possibility to direct standard output from one command into standard input of another. A sequence of commands separated

by vertical bars "|" in a command line causes the shell to start all these commands simultaneously and to set up pipes (see above) in such a way that standard output of the program on the left of the "|" is linked by a pipe to standard input of the program on the right of the "|".

```
ls | pr -2 | lpr
```

ls lists the files in the current directory. Its results are handed to pr that formats its input onto pages with dated headings in two column format (the -2 argument). The output of pr in turn is passed to the printer spooler lpr.

Other operating systems require the use of temporary disk files to do a similar sequence of operations, e.g.

```
ls >temp1
pr -2 <temp1 >temp2
lpr <temp2
rm temp1 temp2
```

where the last step, the removal of the intermediate files is only too often forgotten.

Programs like pr which manipulate data read from standard input and produce results on standard output are called filters. There are many filters available as UNIX utilities.

3.9.3 Running a program in the background

Programs that do not need supervision and do not read from the keyboard can be run in the background. When you terminate your command line with an ampersand (&) before the carriage return, the shell will start the execution of the command, but not wait for its termination, it will instead display the process identification number of the program running in the background and prompt you for a new command. Output produced by the program in the background will still appear on the terminal if you do not redirect it. Most versions of UNIX require a special command when you want the program in the background to continue even in case you log out.

3.9.4 File name generation

Metacharacters can be used in a command line to generate lists of file names. The shell will expand an argument containing metacharacters into a list of file names and pass that list to the program invoked. Valid meta-characters are:

* matches zero or more characters in a file name, but not a leading period (.).

? matches any single character in the name of a file

[..] matches all characters within the brackets, ranges may be defined by placing a hyphen between two characters.

3.9.5 Command files

The shell itself is a command like any other, and can be called recursively. The shell reads commands from standard input and executes them. Commands that have been stored in a file can be executed by invoking the shell recursively and redirecting its standard input such that the shell reads this file:

```
sh <comfile
```

Comfile may for instance contain a series of commands to compile and run a program.

3.9.6 Other capabilities of the shell

The shell is both a command interpreter and a programming language interpreter. As a command interpreter, it executes commands that a user types in. As a programming language interpreter, it interprets statements stored in command files, commonly called shell scripts. The shell programming language comprises features like

- Variable assignment and substitution
- Control structures like if-then-else, case
- Repetitive command execution (for, while, until)
- Reading and prompting for user input
- Substitution of a command by the result of its execution

For more details please consult [5] .

3.9.7 Implementation of command interpreters

A command interpreter, normally a shell, is invoked when a user logs in successfully. This shell then sends a prompt to the screen and issues a read command to get a command line from the keyboard. When the line terminating character is entered, the shell will analyze the command line received. The arguments in the command line, if any, will be put in a form suitable for the exec system call. A child process is then created by the execution of the fork system call. This child process performs an exec system call to load and execute the desired command with the supplied arguments.

The parent process waits for the termination of its child. When this has happened, the parent process knows that the command has been executed, and branches back to prompting and reading command lines from the keyboard.

For the execution of commands in the background the parent process does not await the child's termination, but prints the result of the fork call and branches back to prompting and reading the next command.

Input/Output redirection is taken care of by the child process. Before it executes the command it makes the file descriptor 0 or 1 refer to the file named after the < or > sign. It does so by closing the file with the appropriate file descriptor and then opening the named file. By convention, the smallest file descriptor available will be returned by open, thus nicely establishing the desired connection.

Filters redirect standard I/O to pipes instead of ordinary files in a completely analogous way.

The shell normally stays in an eternal loop, attempting to read the next command. An end of file condition, however, will terminate the shell. This also covers the case of a shell reading a command file. It will terminate when the end of the command file is reached.

3.10 The UNIX programming environment

UNIX is different enough from most other operating systems that it is definitely justified to talk about a specific UNIX programming environment. There are several design concepts of UNIX that merit special attention:

1. Directories are normal files with the exception that they cannot be modified by an ordinary program. Modification of a directory entails modifications of file system information kept in i-nodes, a task which is better left to the kernel. Any ordinary program can, however, read and interpret the contents of a directory, so a directory listing program does not need any privileges.

2. A file is a sequence of bytes. At the first sight this may seem a deficiency of the system because the application program then has to worry about the structure of the data in the file, but it turns out to be a blessing. A user is not at all concerned with physical layouts of disks, things like sector size and track size are hidden from him. There is nothing like record headers or other system generated information in a file, every byte in a file has been put there by the user himself. There is no need to know the file size in advance, because space on the disk is allocated while the file is written. All files are identical in form, there are no "access methods" for different kinds of files.

3. All input/output operations deal with files. There are only a few standard i/o primitives that apply to all files, and peripheral devices are special files in UNIX (see above), which are treated in exactly the same way as disk files. A UNIX program reads data from a file and writes data to a file, and there is no need to care whether the file is actually a keyboard or a disk file.

4. Input/output can be redirected. The ease of input/output redirection together with the just mentioned equivalence of disk files and special files helps enormously in the debugging phases of a program, because one can easily run the program interactively, feeding data into the program from the keyboard and receiving its output on the screen. Once the program is considered sufficiently bugfree, I/O is redirected to disk files or pipes.

5. Tools and pipes allow problem solutions without writing application programs. UNIX comes with a rich set of small, generally useful utilities, like sort programs, pattern-matching programs etc. Quite frequently the combination of tools by pipes into a chain of small programs solves exactly the programmer's problem, relieving him from the burden of writing a special application program for a possibly single occurrence of a problem. Shell programming is also very valuable in this context.

6. There is also a reverse side of the coin. UNIX does not only have fans, but there are also people, who find it horrible [14] . One reason for this attitude is the fact that "learning UNIX" is not a matter of five minutes, the entrance threshold of the system is rather high. Commands often have cryptic names, and you can hardly use the system without a UNIX manual within reach. Another reason is that UNIX tends to be very terse, "no news is good news" seems to be the underlying philosophy. This can at least partly be explained by the fact that programs that are likely to be linked by pipes to other programs must not do things like prompt for input or even send error messages to the standard output file. This, by the way, is another good reason for leaving the standard error output file directed to the terminal screen. It cannot be denied either that UNIX was conceived at a time when hardcopy teletypes were the standard user terminal, and economy on key strokes was important. Recent versions of UNIX, however, got adapted to the world of CRT terminals, some are beginning to make use of the capabilities of modern bitmapped display terminals by provision of window management.

4. Conclusion

The preceding discussion of some UNIX internals and some UNIX features was meant to demonstrate that UNIX is without any doubt one of the most interesting systems in use for computers today. Its use for personal computers will continue to grow for a variety of reasons, limiting factors being its non-trivial demands on hardware and the fact that its multi-user capabilities may be a luxury rather than an asset for small machines. UNIX machines linked together in a local area network and running a "distributed" version of UNIX, e.g. the "Newcastle connection", could turn out to be a very interesting configuration of the future.

References

[1] P.J. Jalics and T.S. Heines, "Transporting a Portable Operating System: UNIX to an IBM Minicomputer," Comm. ACM Vol. No. 12 p. 1066 (December 1983).

[2] M. Tilson, "Moving UNIX to New Machines," BYTE Vol. 8 No. 10 p. 266 (October 1983).

[3] Nai-Ting Hsu, G. Skinner, and J. Zelitzky, "How to Port UNIX to a New Microprocessor," COMPUTER DESIGN Vol. 23 No. 6 p. 173 (June 1984).

[4] H. Katzan jr., Operating Systems, Van Nostrand (1973).

[5] S.R. Bourne, The UNIX System, Addison Wesley (1982).

[6] D.M. Ritchie, "The UNIX I/O System," UNIX Programmer's Manual Vol. 2B(January 1979).

[7] M.K. McKusick, W.N. Joy, S.J. Leffler, and R.S. Fabry, "A Fast File System for UNIX," UNIX Programmer's Manual Vol. 2C(August 1983).

[8] K. Thompson, "UNIX Implementation," Bell System Technical Journal Vol. 57 No. 6 p. 1931 (July 1978).

[9] D.M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," Bell System Technical Journal Vol. 57 No. 6 p. 1905 (July 1978).

[10] O. Babaoglu and W. Joy, "Converting a Swap-Based System to Paging in an Architecture Lacking Page-Referenced Bits," Operating System Review Vol. 15 No. 5 p. 78 (December 1981).

[11] W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher, "4.2BSD System Manual," UNIX Programmer's Manual Vol. 2C(August 1983).

[12] R.C. Gammill, "The UNIX Operating System," SIGSMALL NEWSLETTER Vol. 7 No. 3 & 4 p. 31 (December 1981).

[13] W.N. Joy, "Introduction to the C shell," UNIX Programmer's Manual Vol. 2C(August 1983).

[14] D.A. Norman, "The Trouble with UNIX," DATAMATION Vol. 27 No. 12 p. 139 (November 1981).