INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE: CENTRATOM TRIESTE

UNITED NATIONS INDUSTRIAL DEVELOPMENT ORGANIZATION

INTERNATIONAL CENTRE FOR SCIENCE AND HIGH TECHNOLOGY
c/o INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS 34100 TRIESTE (ITALY) VIA GRIGNANO, 9 (ADRIATICO PALACE) P.O. BOX 586 TELEPHONE 040-224572 TELEFAX 040-224575 TELEX 460449 APH I

SMR/643 - 3

**SECOND COLLEGE ON**
**MICROPROCESSOR-BASED REAL-TIME CONTROL -**
**PRINCIPLES AND APPLICATIONS IN PHYSICS**
**5 - 30 October 1992**

---

# *INTRODUCTION TO REAL-TIME OPERATING SYSTEMS*

**C. VERKERK**
**Computing and Networks Division**
**CERN**
**Geneva**
**Switzerland**

---



MAIN BUILDING Strada Costiera, 11 Tel. 22401 Telefax 224163 / 224559 Telex 460392 ADRIATICO GUEST HOUSE Via Grignano, 9 Tel. 224241 Telefax 224531 Telex 460449
MICROPROCESSOR LAB. Via Beirut, 31 Tel. 224471 Telefax 224600 GALILEO GUEST HOUSE Via Beirut, 7 Tel. 22401

# Introduction to real time operating systems.

September 28, 1992

C. Verkerk
CERN
Geneva

Real time College, Trieste 5 — 30 October 1992

## Introduction.

- These lectures give a basic introduction to real-time operating systems.
- We will concentrate on **principles** and show various solutions to some of the problems.
- For concrete examples, we will - nearly always - use OS-9.
- The lectures will largely follow the presentation given in the book:
  A. Tanenbaum: "Operating Systems, Design and Implementation", Prentice-Hall Int.,1987, ISBN 0-13-637331-3.



## Introduction.

- What is an operating system? Why do we need it? What can we do with it?
- A computer is useful only when it executes a program. Certain parts of this program must handle directly pieces of hardware. This is not simple and very often beyond the average programmer or user. Just try to think of disk I/O!





## Introduction.

- Thus one view of an operating system is that it shall provide a **reasonable interface** to the user. In other words, it should provide to the programmer a set of **easily understandable commands**, for instance for doing I/O. All idiosynchracies of the CPU architecture and of interface chips should be hidden.
- Another view of an operating system is that it should provide **resource management**, resolving possibly conflicting user requests.





## Introduction.

- One can distinguish different types of **interactive** operating systems:
  - simple, single user (MSDOS, Flex)
  - multitasking (Macintosh, GEM, OS2)
  - multi-user, time sharing (UNIX, VM, VMS, OS-9, LynXOS)
  - real-time Kernels (iRMK, AMX, VRTX, etc.)
- As you will have to start soon working with OS-9, we will look first at its outside, before delving into the internal workings of operating systems.

## Overview of OS-9.

The main characteristics of OS-9 are:

- A **real-time, multi-tasking, multi-user** operating system, providing the necessary process scheduling.
- provides **management of system resources:** memory, input/output devices and CPU-time.
- highly **modular** structure, so that a wide range of applications can be covered, from minimal embedded systems (entirely in ROM) to level II multi-user system.





## Overview of OS-9.

- **device-independent I/O** system, expandable.
- a simple, somewhat Unix-like, user interface (**shell**).
- As in UNIX, the system provides, amongst other things: *hierarchical file structure, with file modes; input/output re-direction and pipes and filters; shell scripts.*
- the system is **fast and small** (entirely written in assembly language).





## Overview of OS-9.

The system comes with a comprehensive set of **utility commands** (enhanced and expanded by users) including:

- an editor with macro facilities (**edit**)
- an assembler (**asm**)
- a debugger (**debug**)





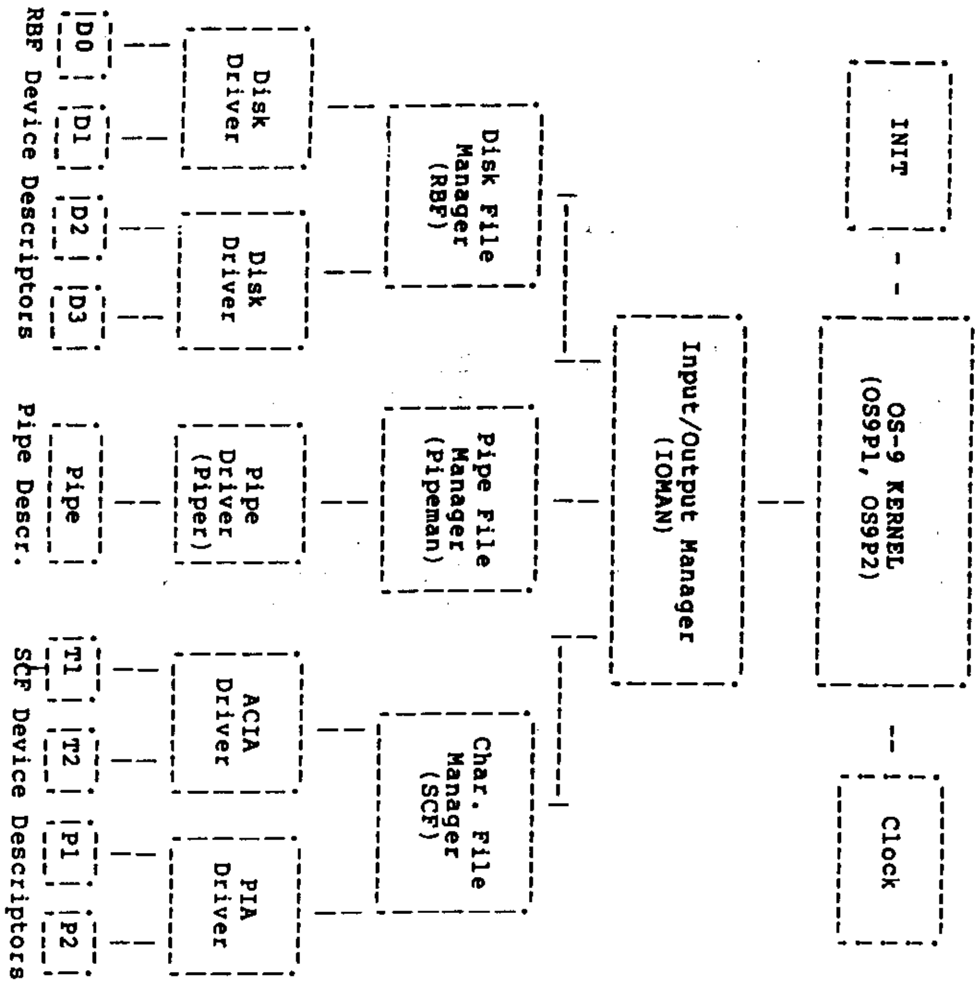
## Overview of OS-9.

It can be further expanded with:

- an interpreter of structured Basic (**Basic09**)
- a **Pascal** compiler
- a **C compiler** (**cc1** is the base module)

The Microprocessor Laboratory has acquired licences for OS-9 itself and for the C compiler.

**OS-9 COMPONENT MODULE ORGANIZATION**



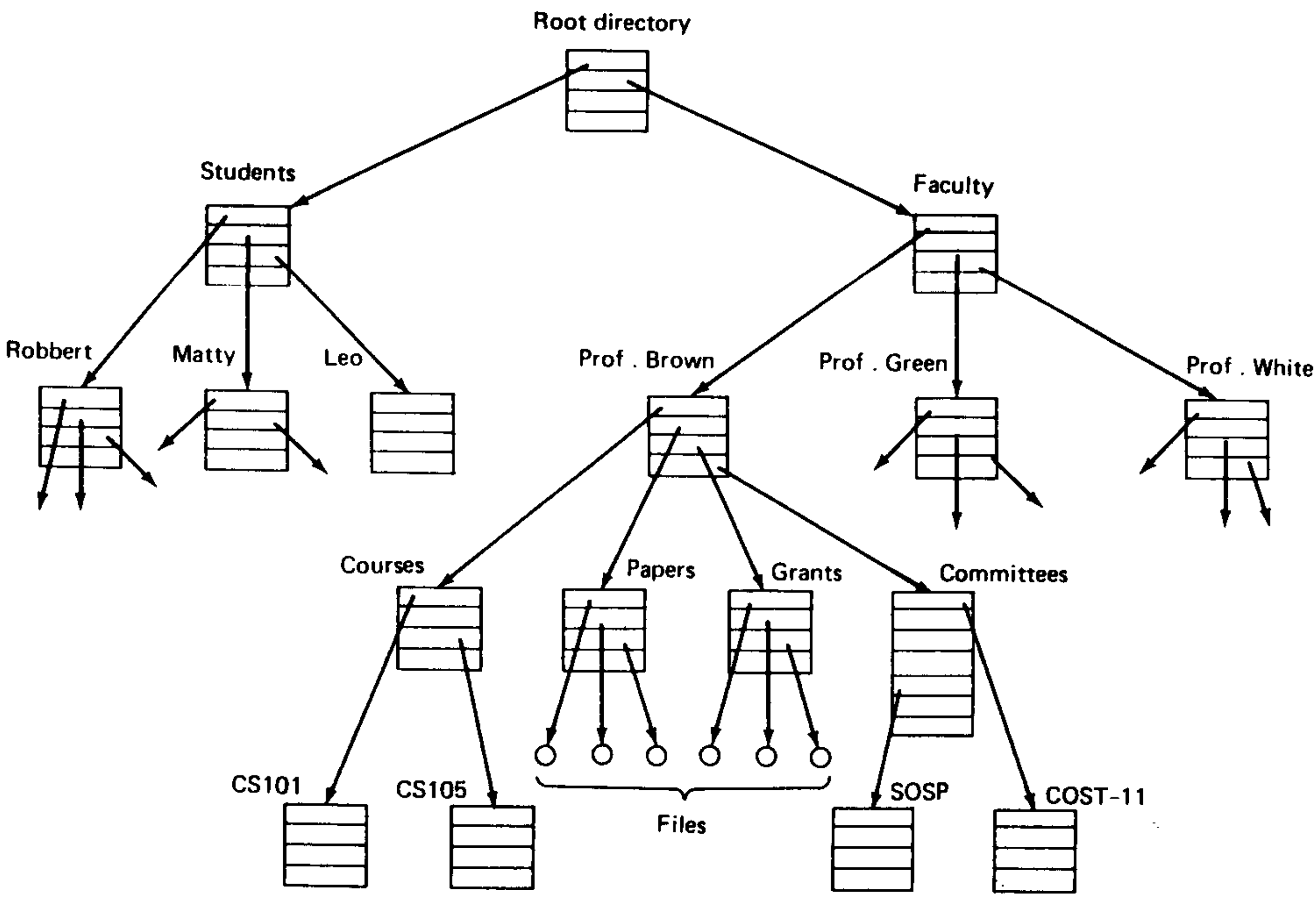




Fig. 1.6

4

## Overview of OS-9.

- The file system is organized **hierarchically**, as in UNIX, but the physical device where the files reside must be specified.
- The **root** is therefore /d0 , /d1 , /r1 , /h0 etc.
- The **root directory** may contain **files**, or it may contain **directories**.
- Each directory may contain **sub-directories**, etc, etc to any depth.
- A **tree structure** is the result.





## Overview of OS-9.

- To reach one of the leaves (a file) you follow a **path**:

  /d0/SRC/C/ACCOUNT/BILL.C

  This is a full pathname.

- One of the directories is the **working directory**. For instance C above. The same file "bill.c" can then be reached following a **partial path**:

  ACCOUNT/BILL.C

  If "ACCOUNT" were the working directory, the pathname "bill.c" would be enough.





## Overview of OS-9.

- Two different files may have the same name, provided they belong to different directories.
- The **working directory** may be designated: . , the **parent** directory: .. So if C were my working directory, and there also existed a file:

  /d0/SRC/PASCAL/STATIS/GAUSS.P

  I could reach it with:

  ../PASCAL/STATIS/GAUSS.P



6



## Overview of OS-9.

- OS-9 has **two** working directories:
  - working **data** directory
  - working **execution** directory
- **chd, chx** commands change the data and execution directories respectively
- **cd, cx** do the same
- **pwd, pwx** show what they are.

```
free /r0

"ICTP 0002 - JuniOS9 ROMdisk 3.0" created on: 92/08/16
Capacity: 2,554 sectors (1-sector clusters)
10 Free sectors, largest block 10 sectors

/r1> dir e /r0

   directory of /r0  10:32:12

Owner Last modified attributes sector bytecount name
----- ------------- ---------- ------ --------- ----------
   0  92/07/14 1825  ------wr       6      2762 OS9Boot
   0  92/08/16 1545  ----r-wr      2F       4E7 startup
   0  92/08/16 1552  d-ewrewr      35       E60 CMDS
   0  92/08/16 1553  d-ewrewr      45       520 DEFS
   0  92/08/16 1553  d-ewrewr      4C        A0 LIB
   0  92/08/16 1554  d-ewrewr      4E       160 PROC
   0  92/08/16 1555  d-ewrewr      51        C0 SYS


/r1> dir e /r0/CMDS

   directory of /r0/CMDS  10:32:36

Owner Last modified attributes sector bytecount name
----- ------------- ---------- ------ --------- ----------
   0  90/04/15 1125  --e-rewr      53      16D4 Asm
   0  92/03/15 1120  --e-rewr      6B       7C8 Debug
   0  90/04/15 1125  --e-rewr      74      1509 Edit
   0  90/04/15 1125  --e-rewr      8B       2C3 acia51
   0  90/04/15 1125  --e-rewr      8F       24B alias
   0  90/04/15 1125  --e-rewr      93      31F4 ar
   0  90/04/15 1125  --e-rewr      C6       285 attr
   0  90/04/15 1125  --e-rewr      CA       4C2 backup
   0  92/05/17 1227  --e-rewr      D0       275 bawk
   0  90/04/15 1126  --e-rewr      D4       278 binex
   0  90/04/15 1126  --e-rewr      D8        54 build
   0  90/04/15 1126  --e-rewr      DA      50FA c.asm
   0  90/04/15 1126  --e-rewr     12C      37FB c.link
   0  90/04/15 1126  --e-rewr     165      2E2C c.opt
   0  90/04/15 1127  --e-rewr     195      7B76 c.pass1
   0  90/04/15 1127  --e-rewr     212      6444 c.pass2
   0  90/04/15 1127  --e-rewr     278      27A9 c.prep
   0  90/04/15 1127  --e-rewr     2A1        38 c1
   0  90/04/15 1127  --e-rewr     2A3        3A c2
   0  90/04/15 1127  --e-rewr     2A5      11A9 call
   0  90/04/15 1128  --e-rewr     2B8      1D68 cb
   0  90/04/15 1128  --e-rewr     2D7      1F09 cc1
   0  90/04/15 1128  --e-rewr     2F8      1019 cmp
   0  90/04/15 1128  --e-rewr     30A       1BF cobbler
   0  92/02/24 1547  --e-rewr     30D       2DE copy
   0  90/04/15 1128  --e-rewr     311       2DC copyold
   0  90/04/15 1128  --e-rewr     315      15C2 cp
   0  90/04/15 1128  --e-rewr     32C        61 crypt
   0  90/04/15 1128  --e-rewr     32E      2352 cxref
   0  90/04/15 1129  --e-rewr     353        FD date
   0  90/04/15 1129  --e-rewr     355      27C6 dcheck
   0  90/04/15 1129  --e-rewr     37E       196 ddir
   0  90/04/15 1129  --e-rewr     381        A5 del
   0  90/04/15 1129  --e-rewr     383       27C deldir
   0  90/04/15 1129  --e-rewr     387       2A2 dir
   0  90/04/15 1129  --e-rewr     38B      24F2 disasm
   0  90/04/15 1129  --e-rewr     3B1        84 display
   0  90/04/15 1130  --e-rewr     3B3      19B7 dsave
   0  90/04/15 1130  --e-rewr     3CE       545 dterm
   0  90/04/15 1130  --e-rewr     3D5       180 dump
   0  90/04/15 1130  --e-rewr     3D8        22 echo
   0  90/04/15 1130  --e-rewr     3DA        D2 error
   0  90/04/15 1130  --e-rewr     3DC       1BA exbin
   0  90/04/15 1130  --e-rewr     3DF      28B1 field
   0  90/04/15 1130  --e-rewr     409      131B find
   0  90/04/15 1131  --e-rewr     41E      13DE findfunc
   0  92/03/08 1051  --e-rewr     433        92 fixdir
   0  92/03/08 1356  --e-rewr     435        9A fixdir2
   0  92/05/14 2242  --e-rewr     437       177 fgrep
   0  90/04/15 1131  --e-rewr     43A       DD1 format
   0  90/04/15 1131  --e-rewr     449       2D1 free
   0  90/04/15 1131  --e-rewr     44D       12D go
   0  90/04/15 1131  --e-rewr     450      28CC grep
   0  92/05/14 2242  --e-rewr     47A        93 head
   0  90/04/15 1131  --e-rewr     47C       B57 help
   0  90/04/15 1131  --e-rewr     489       430 hx
   0  90/04/15 1131  --e-rewr     48F       6E7 ident
   0  90/04/15 1132  --e-rewr     497       1E3 idir
   0  90/04/15 1132  --e-rewr     49A      3656 kermit
   0  90/04/15 1132  --e-rewr     4D2      1146 lib
   0  90/04/15 1132  --e-rewr     4E5      125C line
   0  90/04/15 1132  --e-rewr     4F9        2C link
   0  90/04/15 1132  --e-rewr     4FB        4F list
   0  90/04/15 1132  --e-rewr     4FD        24 load
   0  92/06/15 1155  --e-rewr     4FF       3FB login
   0  90/04/15 1133  --e-rewr     504      3056 ls
   0  90/04/15 1133  --e-rewr     536        22 makdir
   0  90/04/15 1133  --e-rewr     538       5A4 make
   0  90/04/15 1133  --e-rewr     53F       34B makrom
   0  90/04/15 1133  --e-rewr     544       1CA mdir
   0  90/04/15 1133  --e-rewr     547      28F9 mdump
   0  90/04/15 1134  --e-rewr     571      2938 mem_save
   0  90/04/15 1134  --e-rewr     59C        68 merge
   0  90/04/15 1134  --e-rewr     59E       128 mfree
   0  90/04/15 1134  --e-rewr     5A1        A8 mrename
   0  90/04/15 1134  --e-rewr     5A3       A35 mv
   0  90/04/15 1134  --e-rewr     5AF       2CB os9gen
   0  90/04/15 1134  --e-rewr     5B3        38 p1
   0  90/04/15 1134  --e-rewr     5B5        3A p2
   0  90/04/15 1135  --e-rewr     5B7      2736 patch
   0  90/04/15 1135  --e-rewr     5E0      2C70 plist
   0  90/04/15 1135  --e-rewr     60E        F6 printerr
   0  90/04/15 1135  --e-rewr     610       226 procs
   0  90/04/15 1135  --e-rewr     614        4E purge
   0  90/04/15 1135  --e-rewr     616       130 pwd
   0  90/04/15 1135  --e-rewr     619       130 pxd
   0  90/04/15 1135  --e-rewr     61C      1D78 rdump
   0  90/04/15 1136  --e-rewr     63B       11D rename
   0  90/04/15 1136  --e-rewr     63E      1441 repl
   0  90/04/15 1136  --e-rewr     654        67 save
   0  90/04/15 1136  --e-rewr     656      2795 sdir
   0  90/04/15 1136  --e-rewr     67F       118 setime
   0  90/04/15 1136  --e-rewr     682       4FA shellold
   0  90/04/15 1136  --e-rewr     688      1FCB simon
   0  92/08/16 1625  --e-rewr     6A9      4C89 sled
   0  90/04/15 1137  --e-rewr     6F7        4D sleep
   0  90/04/15 1137  --e-rewr     6F9       5D3 sort
   0  90/04/15 1137  --e-rewr     700       1E3 spaces
   0  90/04/15 1137  --e-rewr     703      166E spint
   0  90/04/15 1137  --e-rewr     71B       3DE strip
   0  90/04/15 1138  --e-rewr     720        38 t1
   0  90/04/15 1138  --e-rewr     722        3A t2
   0  92/06/07 1114  --e-rewr     724       171 tail
```

```
 0  90/04/15 1138  --e-rewr     727        F8D tc
 0  90/04/15 1138  --e-rewr     738         7D tee
 0  90/04/15 1138  --e-rewr     73A        2D6 tmode
 0  90/04/15 1138  --e-rewr     73E        18F trlf
 0  90/04/15 1138  --e-rewr     741         82 tsmon
 0  90/04/15 1138  --e-rewr     743         38 unlink
 0  90/04/15 1139  --e-rewr     745        172 verify
 0  90/04/15 1139  --e-rewr     748       11F2 wc
 0  90/04/15 1139  --e-rewr     75B         F9 words
 0  90/04/15 1139  --e-rewr     75D        44A xmode

/r1> dir e /r0/LIB

   directory of /r0/LIB  10:33:03

Owner Last modified attributes sector bytecount name
----- ------------- ---------- ------ --------- ----------
  0  90/04/15 1420  ----r-wr     8C4       5C5B clib.l
  0  90/04/15 1420  ----r-wr     922        284 cstart.r
  0  90/04/15 1421  ----r-wr     926       933B klib.l

/r1> dir e /r0/DEFS

   directory of /r0/DEFS  10:33:14

Owner Last modified attributes sector bytecount name
----- ------------- ---------- ------ --------- ----------
  0  90/04/15 1414  ----r-wr     763         4D arg.h
  0  90/04/15 1414  ----r-wr     765         33 bool.h
  0  90/04/15 1414  ----r-wr     767        46F cmacros.h
  0  90/04/15 1414  ----r-wr     76D        3B3 ctype.h
  0  90/04/15 1414  ----r-wr     772        2E1 defsfile
  0  90/04/15 1414  ----r-wr     776        1D0 dir.h
  0  90/04/15 1414  ----r-wr     779        2D1 direct.h
  0  90/04/15 1415  ----r-wr     77D        1A7 dstart.h
  0  90/04/15 1415  ----r-wr     780        B5A errno.h
  0  90/04/15 1415  ----r-wr     78D        E7B li.equates
  0  90/04/15 1415  ----r-wr     79D        192 lowio.h
  0  90/04/15 1415  ----r-wr     7A0        159 mat.h
  0  90/04/15 1415  ----r-wr     7A3         DD math.h
  0  90/04/15 1415  ----r-wr     7A5         4F memory.h
  0  90/04/15 1415  ----r-wr     7A7        247 modes.h
  0  90/04/15 1416  ----r-wr     7AB        B09 module.h
  0  90/04/15 1416  ----r-wr     7B8       1892 os9.h
  0  90/04/15 1416  ----r-wr     7D2       2A3E os9defs
  0  90/04/15 1416  ----r-wr     7FE       185F os9defs.a
  0  90/04/15 1416  ----r-wr     7FF        947 os9iodefs
  0  90/04/15 1416  ----r-wr     823       12D2 os9rbfdefs
  0  90/04/15 1416  ----r-wr     837        A0F os9scfdefs
  0  90/04/15 1416  ----r-wr     843        BAC os9sysdefs.li
  0  90/04/15 1417  ----r-wr     850        607 prec.h
  0  90/04/15 1417  ----r-wr     858        2AD rof.h
  0  90/04/15 1417  ----r-wr     85C         18 setjmp.h
  0  90/04/15 1417  ----r-wr     85E        10C sets.h
  0  90/04/15 1417  ----r-wr     861        C78 sgstat.h
  0  90/04/15 1417  ----r-wr     86F        183 signal.h
  0  90/04/15 1417  ----r-wr     872        4FE stdio.h
  0  90/04/15 1417  ----r-wr     878        134 stdlib.h
  0  90/04/15 1417  ----r-wr     87B        1E6 strings.h
  0  90/04/15 1418  ----r-wr     87E        36F sysdefs
  0  90/04/15 1418  ----r-wr     883        6B2 sysmem.h
  0  90/04/15 1418  ----r-wr     88B        CC1 systype
  0  90/04/15 1418  ----r-wr     899        179 time.h
```

7

```
  0  90/04/15 1418  ----r-wr     89C        35B utime.h
  0  90/04/15 1418  ----r-wr     8A1         46 utmp.h
  0  90/04/15 1427  ----r-wr     8A3       1F30 sgtty.h

/r1> dir e /r0/PROC

   directory of /r0/PROC  10:33:28

Owner Last modified attributes sector bytecount name
----- ------------- ---------- ------ --------- ----------
  0  90/04/15 1422  ----r-wr     9BB         EB a.com
  0  90/04/15 1422  ----r-wr     9BD        186 c.com
  0  90/04/15 1422  ----r-wr     9C0        15C c.com0
  0  90/04/15 1422  ----r-wr     9C3         40 comp.template
  0  90/04/15 1422  ----r-wr     9C5          8 dummy.r
  0  90/04/15 1422  ----r-wr     9C7        7BB exit
  0  90/04/15 1422  ----r-wr     9D0        25F filemake
  0  90/04/15 1422  ----r-wr     9D4        13B flink
  0  90/04/15 1423  ----r-wr     9D7        268 makeit

/r1> dir e /r0/SYS

   directory of /r0/SYS  10:33:37

Owner Last modified attributes sector bytecount name
----- ------------- ---------- ------ --------- ----------
  0  92/03/15 0950  ----r-wr     9DB        DAF errmsg
  0  90/04/15 1424  ------wr     9EA         F3 password
  0  90/04/15 1424  ----r-wr     9EC         70 Bootlist.stndaln
  0  90/04/15 1424  ----r-wr     9EE         7E Bootlist.wrksttn

/r1>
```

## ROM-RAM disk.

- Operating OS-9 from floppy disks is relatively slow. A very useful in-house enhancement has been the development of a ROM-RAM disk.
- The **640K ROM disk** is the exact equivalent of an entire floppy. It is device **/r0**.
- It contains the entire system and all the **utility commands** (many more than provided by Microware), the **DEFS** and **#include** files, the **C library** and a set of **procedure files.**





## ROM-RAM disk.

- The **160K RAM disk** is used to hold the **working directory**. Its use speeds up considerably the execution of most commands (in particular edit, asm and cc1). Its name is **/r1**.





## Shell.

- The **Shell** is the **interactive user-interface** to the system.
- You type a command, the shell will take the necessary steps to execute it.
- In addition to basic command line processing, the shell has functions for:
  - I/O redirection (including pipes and filters)
  - memory allocation
  - multitasking (e.g. concurrent execution)
  - procedure file execution
  - execution control (with built-in commands)







## Shell.

- A command line consists of:
  - a **"verb"** (name of a program, shell script or built-in command)
  - **parameters** to be passed to the program
  - **execution modifiers** to be processed by the shell.

EXAMPLES:

```
ASM MYFILE L -O >/P1 #12K
```

## Shell.

- **Execution modifiers** are:
  - # *memory allocation*
  - ; *sequential execution*
  - ! *pipe*
  - | *pipe*
  - < *redirect standard input*
  - > *redirect standard output*
  - >> *redirect error output*
  - & *run in background (concurrently).*
- Commands can be **grouped** using **(** and **)**.

```
OS9: (DIR CMDS; DIR SYS)! SORT ! ECHO
```



## Shell.

- **Built-in Shell commands**:
  - **chd** **<pathlist>** change data directory (**cd** *does the same*)
  - **chx** **<pathlist>** change execution directory (**cx** *does same*)
  - **ex name** execute name instead of shell.
  - **w** wait
  - *** text** comment (script)
  - **kill** **<proc ID>** abort process
  - **setpr** **<proc ID><priority>** change priority
  - **x, −x** abort, do not abort, on error
  - **p, −p** prompt on, off
  - **t, −t** copy, do not copy input lines to output





## Shell.

- A command may be run in **background** (by ending the command line with &): As soon as execution has started, control comes back to the shell: the **shell prompt** OS9: appears. You may now run another command in background, or in **foreground**.
- **Interaction** is only possible with the program in **foreground**. The following makes sense:

```
OS9: LIST LONGFILE >/P1 &
OS9: EDIT MYFILE
```





## Shell.

This does not make sense:

```
OS9: EDIT MYFILE &
OS9: LIST LONGFILE >/P1
```

- Shell may be instructed to execute two or more commands in sequence:

```
OS9: CHD/R1; COPY /D1/HELP/ATTR ATTR; EDIT ATTR
```

- The vast majority of OS-9 programs are **re-entrant**. The OS-9 C Compiler produces re-entrant code. Thus, if two users are simultaneously editing (each his own file), only one copy of "edit" will be in memory. The two **data spaces** are of course separate.

## Shell.

- All commands use a default size of working space (minimum **one page of 256 bytes**). The shell can override the default with:

```
OS9: EDIT #20K MYFILE
OR OS9: EDIT #80  MYFILE.
```

- Shell will look for a command in the current execution directory. If not found, it looks in the data directory. If it finds there a file with the requested name, it will try to execute it as a **shell script** or **procedure file**. A shell script contains one or more command lines (and comments) that the shell will interpret.



## Unified I/O System.

- As in UNIX, Input/Output **devices** are treated as **files** in OS-9, making I/O **device-independent** from the user's point of view.
- OS-9 can handle various types of devices:
  - **random block** file (disks)
  - **sequential block** file (tape)
  - **sequential character** file (terminals, printers)
  - **pipes**.





## Unified I/O System.

- **Device-independency** is achieved through a **layered structure**:
  - **IOMan** at the top, manages all requests. It passes a request on to the appropriate **file manager**.
  - For each type of device there is such a **file manager**:

```
RBF FOR DISKS
SCF FOR CHARACTER DEVICES
PIPEMAN FOR PIPES.
```

  A file manager can handle different types of devices (rbf handles floppies as well as hard disks).





## Unified I/O System.

- For each specific **hardware controller** there is a **device driver**:

```
RTFDC FOR FLOPPIES
ROAMER FOR THE ROM-RAM DISK
ACIA AND ACIA51 FOR TERMINALS, PRINTERS
PIPER FOR PIPES,
ETC.
```

- Every individual device has a **device descriptor**. Examples of those: d0, d1, r0, r1, term, p1, t2 etc.

## Unified I/O System.

- A **path** is to be opened to (a file on) a device before one can perform I/O transfers.
- There are three special paths, which are always open:
  - **standard input (stdin)** from the keyboard,
  - **standard output (stdout)** to the screen,
  - **standard error (stderr)**, usually to the screen.





## Unified I/O System.

- Input and Output may be **redirected**, adding **<pathname** or **>pathname** on the command line:

```
OS9: LIST MYFILE >/P1
OS9: DIR E >/R1/KEEPDIR
OS9: SORT </R1/UNSORTED >/R1/SORTED
```

- >> redirects stderr.





## Unified I/O System.

- A **pipe connects the output** of a program **to the input** of another program:

```
OS9: DIR E ! SORT
```

  You may continue:

```
OS9: DIR ! WORDS ! SORT
```

- A filter is a program which **reads from stdin**, transforms the data, and **writes the result to stdout**. Filters are very useful for use in pipes. Sort, words, wc are examples of filters.





## Memory Modules:

- All programs must conform to the standard **memory module** format, otherwise they cannot be loaded.
- A memory module has a **header**, followed by the program code, and a CRC.
- Header contains the following:
  - synchronization bytes (87CD)
  - length of module (in bytes)
  - pointer to module's name string
  - type/language byte
  - revision/attribute byte
  - checksum.

**EXECUTABLE MEMORY MODULE FORMAT**

| Relative Address | Usage | Check Range |
|---|---|---|
| $00 | Sync Bytes ($87CD) | header parity / module CRC |
| $01 | | |
| $02 | Module Size (bytes) | |
| $03 | | |
| $04 | Module Name Offset | |
| $05 | | |
| $06 | Type ! Language | |
| $07 | Attributes ! Revision | |
| $08 | Header Parity Check | |
| $09 | Execution Offset | |
| $0A | | |
| $0B | Permanent Storage Size | |
| $0C | | |
| $0D | (Add'l optional header extensions located here) | |
| | . . . . . . . . | |
| | Module Body object code, constants, etc. | |
| | CRC Check Value | |

## 4.2.0 MODULE HEADER DEFINITIONS

The first nine bytes of all module headers are identical:

| MODULE OFFSET | DESCRIPTION |
|---|---|
| $0,$1 = | Sync Bytes ($87,$CD). These two constant bytes are used to locate modules. |
| $2,$3 = | Module Size. The overall size of the module in bytes (includes CRC). |
| $4,$5 = | Offset to Module Name. The address of the module name string relative to the start (first sync byte) of the module. The name string can be located anywhere in the module and consists of a string of ASCII characters having the sign bit set on the last character. |
| $6 = | Module Type/Language Type. See text. |
| $7 = | Attributes/Revision Level. See text. |
| $8 = | Header Check. The one's compliment of the vertical parity (exclusive OR) of the previous eight bytes. |

## Memory Modules:

- Depending on the module type, the following information may also be contained in the header:
  - pointer to execution entry point
  - size of static storage area required.
  - pointers to other name strings.
- The **mod** assembler directive sets up the header.
- The **emod** assembler directive generates a 3-byte CRC for the module.
- Modules are loaded in high memory addresses.





## Memory Modules:

- Static storage areas associated with modules are allocated at low memory addresses.
- A directory of modules in memory is kept in page 2 of memory.
- A module **must be written in Position Independent Code**. Thus no relocation is needed.
- In order to make modules **sharable** between several processes, they should be **re-entrant**.





## Memory Modules:

- Re-entrancy is assured if you:
  - *don't store variables in the code* segment of the module.
  - use *program counter relative addressing* for addresses inside the module.
  - keep all variables in the **direct storage page**. Address those variables using either *direct page* addressing or *indexed addressing* (using the *U register*).
- The C compiler produces position independent, re-entrant code. It adds the header and CRC to the code.





## Limitations.

The system has of course its limitations:

- memory is **restricted to 64K.** Care must be exercised when compiling programs.
- **shell is rather restricted** (it is very small!):
  - no *environment* and other *shell variables*,
  - no *wildcards*,
  - no *conditionals* or *loops* in shell scripts.
  - no *formal parameters* for shell scripts.

### 4.2.1 Type/Language Byte

The module type is coded into the four most significant bits of byte 6 of the module header. Eight types are predefined by convention, some of which are for OS-9's internal use only. The type codes are:

```
     Code  Name   Meaning
     ----  ------ -------
     $10   Prgrm  Program module
     $20   Sbrtn  Subroutine module
     $30   Multi  Multi-module (for future use)
     $40   Data   Data module
  $50-$B0   User-definable
     $C0   Systm  OS-9 System module
     $D0   FlMgr  OS-9 File Manager module
     $E0   Drivr  OS-9 Device Driver module
     $F0   Devic  OS-9 Device Descriptor module
```

NOTE: 0 is not a legal type code

The four least significant bits of byte 6 describe the language type as listed below:

```
0  Data (non-executable)
1  Objct  6809 object code
2  ICode  BASIC09 I-code
3  PCode  PASCAL P-code
   The following are currently not implemented:
4  CCode  C I-code
5  CblCode  COBOL I-code
6  FrtnCode  FORTRAN I-code
```

The purpose of the language type is so high-level language run-time systems can verify that a module is of the correct type before execution is attempted. BASIC09, for example, may run either I-code or 6809 machine language procedures arbitrarily by checking the language type code.

```
                *****
                *  LIST UTILITY COMMAND
                *  Syntax: list <pathname>
                *  COPIES INPUT FROM SPECIFIED FILE TO STANDARD OUTPUT

0000 87CD004E                 mod LSTEND,LSTNAM,PRGRM+OBJCT,
                                  REENT+1,LSTENT,LSTMEM
000D 4C6973F4     LSTNAM   fcs    "List"

                * STATIC STORAGE OFFSETS
                *
00C8              BUFSIZ   equ    200         size of input buffer
0000                       ORG    0
0000              IPATH    rmb    1           input path number
0001              PRMPTR   rmb    2           parameter pointer
0003              BUFFER   rmb    BUFSIZ      allocate line buffer
00CB                       rmb    200         allocate stack
0193                       rmb    200         room for parameter lis
025B              LSTMEM   EQU    .

0011 9F01         LSTENT   stx    PRMPTR      save parameter ptr
0013 8601                  lda    #READ.      select read access mod
0015 103F84                os9    I$OPEN      open input file
0018 252E                  bcs    LIST50      exit if error
001A 9700                  sta    IPATH       save input path number
001C 9F01                  stx    PRMPTR      save updated param ptr

001E 9600         LIST20   lda    IPATH       load input path number
0020 3043                  leax   BUFFER,U    load buffer pointer
0022 108E00C8              ldy    #BUFSIZ     maximum bytes to read
0026 103F8B                os9    I$RDLN      read line of input
0029 2509                  bcs    LIST30      exit if error
002B 8601                  lda    #1          load std. out. path #
002D 103F8C                os9    I$WRLN      output line
0030 24EC                  bcc    LIST20      Repeat if no error
0032 2014                  bra    LIST50      exit if error
0034 C1D3         LIST30   cmpb   #E$EOF      at end of file?
0036 2610                  bne    LIST50      branch if not
0038 9600                  lda    IPATH       load input path number
003A 103F8F                os9    I$CLOS      close input path
003D 2509                  bcs    LIST50      ..exit if error
003F 9E01                  ldx    PRMPTR      restore parameter ptr
0041 A684                  lda    0,X
0043 810D                  cmpa   #$0D        End of parameter line?
0045 26CA                  bne    LSTENT      ..no; list next file
0047 5F                    clrb
0048 103F06       LIST50   os9    F$EXIT      ... terminate

004B 95BB58                emod               Module CRC

004E              LSTEND   EQU    *
```

## Limitations.

- **Inter-process communication and synchronization** of user processes is largely left to the user.
- There are however tricks to overcome some of these limitations. Also some commands allow wildcards, parameters or loops.
- In the present version of OS-9 Microware's **shell** has been enhanced:
  - **prompt** reflects current directory
  - a **history** file is kept
- A more powerful shell (**shell+**, originally for level II) exists



## Levels of Abstraction.

- One can view a computer at different levels of abstraction:
  - at the bottom is **hardware**.
  - a microprogram (or hardwired logic) acting on the hardware defines a **machine language**.
  - the **operating system** provides a more convenient **interface** fo the user. It defines a **virtual machine**.
  - on top of the operating system we have **system utilities** such as : a command interpreter, compilers, editors, etc.





## Levels of Abstraction.

  - the top most layer is made up of the application programs.
- The operating system itself can (and usually is) also be **layered**.
- The operating system may in fact have one of four possible structures:
  - monolithic
  - layered
  - virtual machine
  - server-client.
- In the server-client concept some functions, traditionally part of the operating system, are pushed upward to higher layers (e.g. file-server).





## Levels of Abstraction.

- The client-server model is well adapted to distributed systems.
- On most machines, the operating system runs in **kernel (or protected) mode**, whereas the layers above it run in **user mode**. Certain machine instructions cannot be executed in user mode.
- Evolution of operating systems: "plugboards", batch, multi-programming, time-sharing. With PCs and workstations evolution toward user-friendliness and network operating systems.

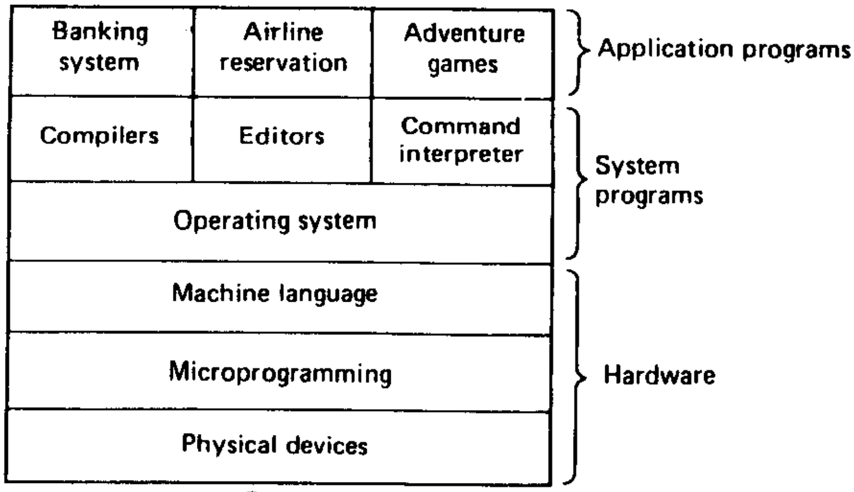



**Fig. 1-1.** A computer system consists of hardware, system programs and application programs.

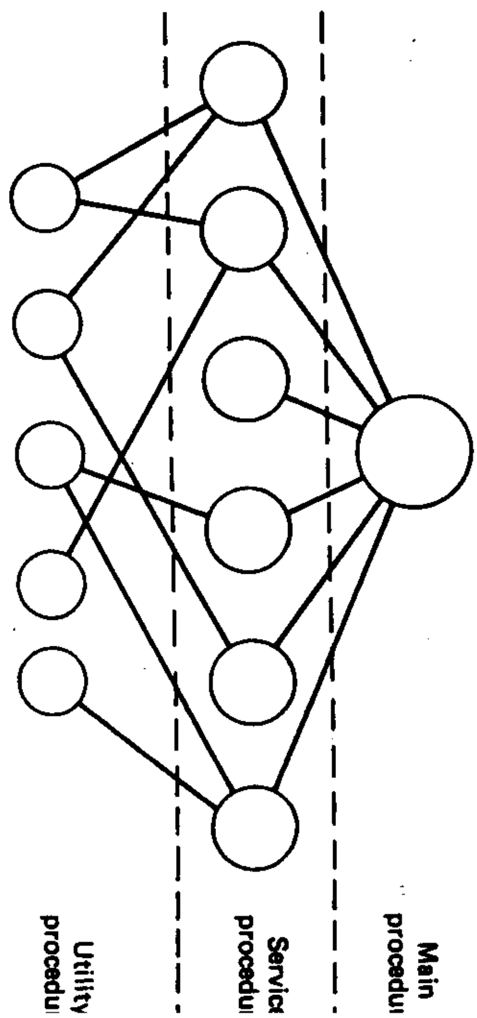



**Fig. 1-19.** A simple structuring model for a monolithic system.

| | |
|---|---|
| 5 | The operator |
| 4 | User programs |
| 3 | Input/output management |
| 2 | Operator-process communication |
| 1 | Memory and drum management |
| 0 | Processor allocation and multiprogramming |

**Fig. 1-20.** Structure of the THE operating system.

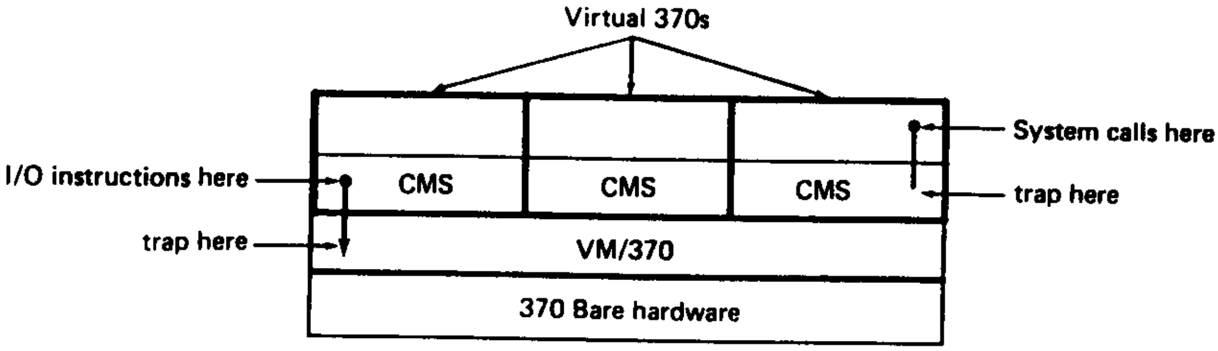



**Fig. 1-21.** The structure of VM/370 with CMS.

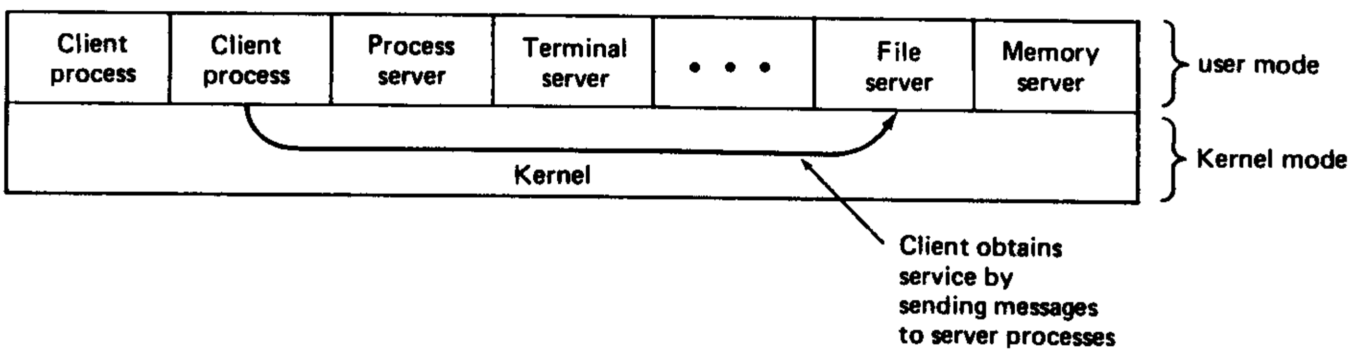
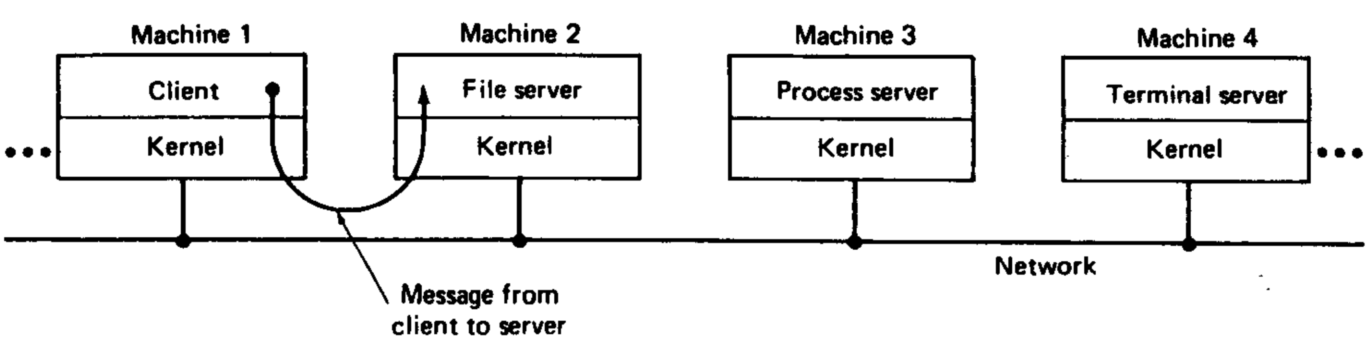



**Fig. 1-22.** The client-server model.



## System Calls and Processes.

- Programs communicate with the operating system through **system calls** (or **service requests**).Implementation varies, but it usually works through a **software interrupt**.
- Key concept is the **process**. A process is a program in execution; it consists of the executable program, its data and stack, program counter, stack pointer and all other registers and other information needed to run the program.



## System Calls and Processes.

- The existence of processes implies that there must exist system calls for **creating** and **terminating** processes. The **shell** or **command interpreter** will create a process when the user has typed in a line. For instance, it may create a process that will run the compiler. When that process has finished its job, it will execute a system call to terminate itself.
- The shell itself is also a process. In the example, the compiler is a **child** process of the shell. When it terminates it returns to the **parent**.





## System Calls and Processes.

- A process may be suspended and later resumed. All information must be saved before suspension!
- Processes will need memory to run in. When a process dies, the memory becomes free for another process to use.
- Processes will often create child processes to communicate with the outside world, e.g. perform input-output.





## System Calls and Processes.

- It is now easy to see that an operating system has four main functions:
  - **process management**
  - **memory management**
  - **Input/Output management**
  - **Interrupt handling** and time management.
- All these functions have their specific system calls. They are implemented as a disjoint **set of programs**, which make use of **common utility routines** (for instance for adding or deleting items from a table).

## Kernel of OS-9.

- Consists of two modules, both in ROM: OS9p1 and OS9p2.
- **Kernel** takes care of:
  - system initialisation
  - Processing of service requests (system calls)
  - memory management
  - process scheduling
  - interrupt processing.
- **Service requests** are made via SWI2, followed by an identification number:

```
SWI2
FCB    $06
```



## Kernel of OS-9.

- Or, in assembly, and using "OS9sysdefs":

```
OS9  F$EXIT
```

- Or, in C, using functions in the library:

```
EXIT(0)
```

- Two types of service requests to the kernel:
  - **user mode** system calls can be made from any program.
  - **privileged system mode** calls can only be made from within system routines.





## Kernel of OS-9.

- Therefore two dispatch tables in page 1 of memory.
- **Input/output system calls** are another category, not handled by the kernel (but by **IOMan**)
- After initialisation a contiguous block of memory is free. The kernel can allocate pieces of it to program modules and direct page storage areas.





## Service Requests of OS-9.

- There are:
  - **30 user mode** (of which 6 level II)
  - **40 privileged mode** (28 level II)
  - **17 I/O service requests.**
- They are used to
  - allocate, de-allocate: bits in a bitmap, 64 byte memory blocks or memory pages.
  - load, link, unlink modules
  - fork, chain, suspend, terminate processes
  - parse path names, compare names.
  - send signals and signal intercepts.
  - manipulate process IDs, priorities, process queues.
  - a few miscellaneous.

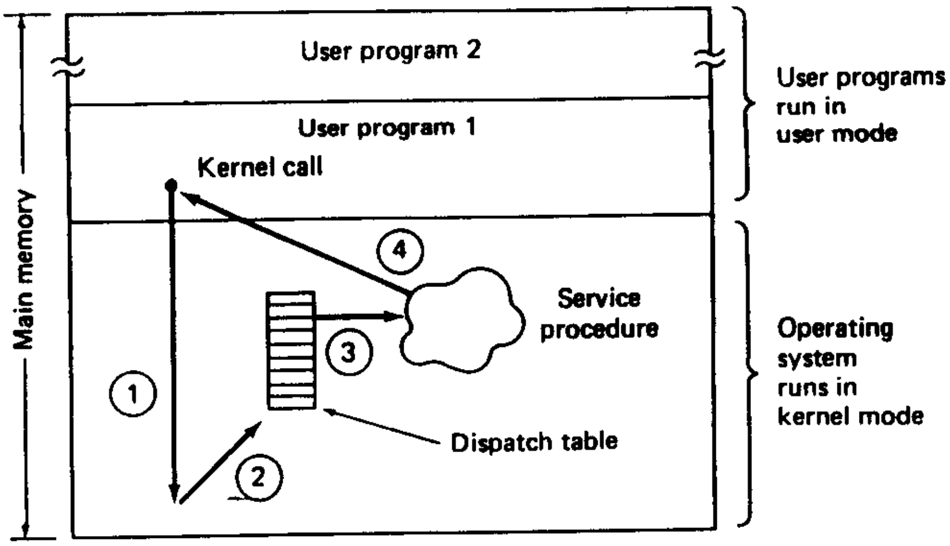



**Fig. 1-18.** How a system call can be made: (1) User program traps to the kernel. (2) Operating system determines service number required. (3) Operating system locates and calls service procedure. (4) Control is returned to user program.



User Mode Service Requests

## System Mode Privileged Service Requests



*NOTE: F$SSVC is a user mode function, although its code > $27





## INPUT/OUTPUT SERVICE REQUESTS



### STANDARD I/O PATHS

0 = Standard Input
1 = Standard Output
2 = Standard Error Output

### FILE ACCESS CODES

READ = $01
WRITE = $02
UPDATE = READ + WRITE
EXEC = $04
PREAD = $08
PWRIT = $10
PEXEC = $20
SHARE = $40
DIR = $80

### MODULE TYPES

$10 = Program
$20 = Subroutine Module
$30 = Multi-Module
$40 = Data
$C0 = System Module
$D0 = File Manager
$E0 = Device Driver
$F0 = Device Descriptor

### MODULE LANGUAGES

$0 = Data
$1 = 6809 Object code
$2 = BASIC09 I-Code
$3 = Pascal P-Code
$4 = Cobol I-code

### MODULE ATTRIBUTES

$8 - Reentrant

## Service Requests of OS-9.

- We can now look in more detail at an example in assembly language and C.
- The **C library** (/r0/LIB/clib.l) contains:
  - standard C library routines. These will often make use of the available service requests. For instance **malloc( )** will eventually translate into **F$SRqMem**.
  - routines which are the **equivalent of the service requests**, if they are **OS-9 specific**.
- **stdio.h** will help in translating into OS-9 I/O service requests.
- Try this: **lib clib.l**.





## Processes.

- A single CPU can only do one thing at a time. Rapid switching from one task to another creates the illusion that the CPU is doing many things in parallel.
- The **process model** helps to understand what happens and to keep track of things going on.
- All runnable software (including the OS) is organised in **sequential processes**.





## Processes.

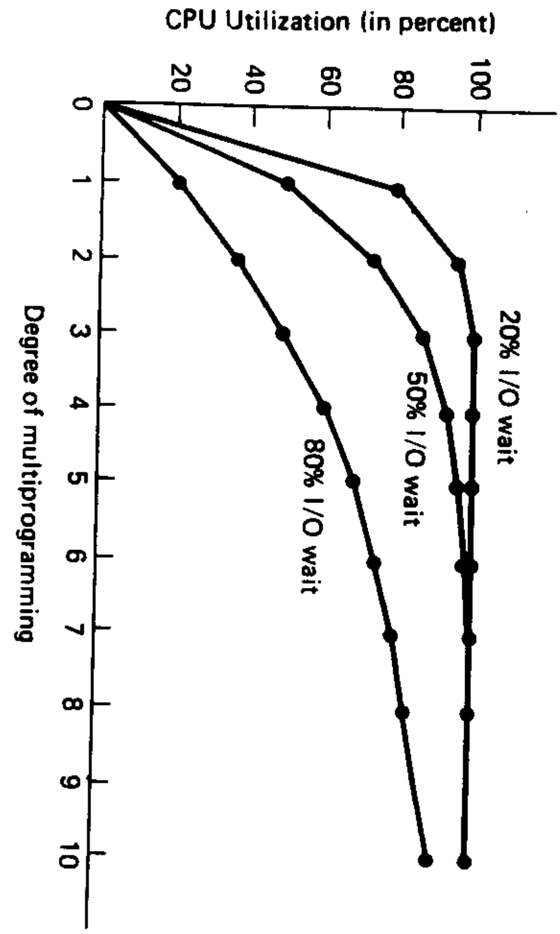
- A **process** is an *executing program,including its input and output and its state (e.g. the contents of machine registers and the values of the program's variables).*
- The operating system must be able to **create** a process when needed and to destroy it when finished. In UNIX,Minix and OS-9 a process is created with the **FORK** system call.
- A process may issue one or more FORKs, and create **child process**(es). The child(s) may again create other childs, etc.



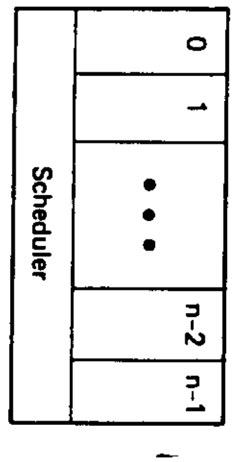


## Processes.

- When the system is booted, at some stage a process is forked, which will start things going. In a multiprogramming system, one process per terminal may be started; each will wait till someone logs in.
- In OS-9, at the end of the boot **SysGo** is called, which forks **shell**. Shell will wait for a command from the user. SysGo continues to exist as a process, although it has nothing more to do.
- In the model, processes are independent entities, but often they need to interact with each other (through a pipe or otherwise).




4-2. CPU utilization as a function of the number of processes in memory.



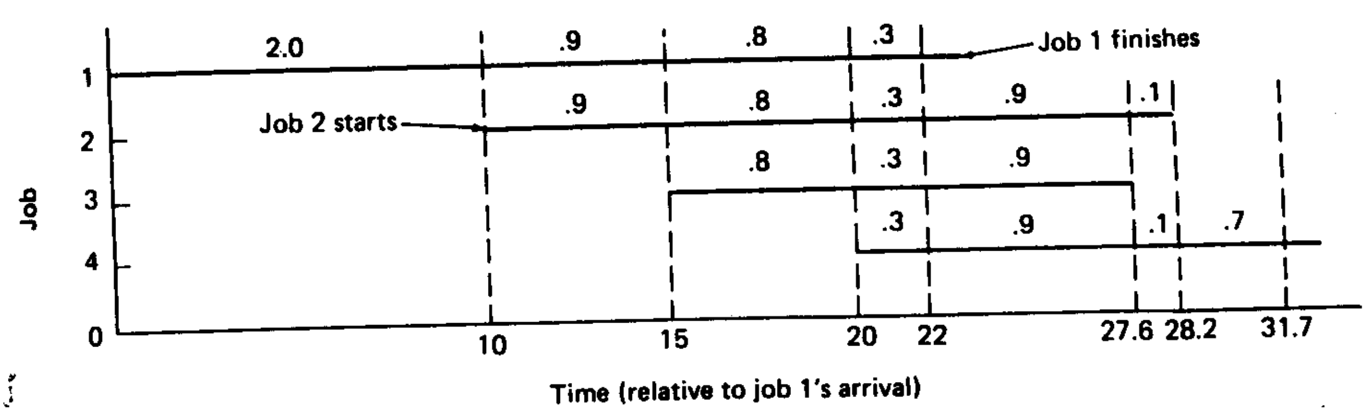

The lowest layer of a process-structured operating system handles in-
nd does scheduling. The rest of the system consists of sequential

| Job | Arrival time | CPU minutes needed |
|---|---|---|
| 1 | 10:00 | 4 |
| 2 | 10:10 | 3 |
| 3 | 10:15 | 2 |
| 4 | 10:20 | 2 |

(a)

| | #Processes 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| CPU idle | .80 | .64 | .51 | .41 |
| CPU busy | .20 | .36 | .49 | .59 |
| CPU/process | .20 | .18 | .16 | .15 |

(b)




(c)

```
*****
*   LIST UTILITY COMMAND
*   Syntax: list <pathname>
*   COPIES INPUT FROM SPECIFIED FILE TO STANDARD OUTPUT

0000 87CD004E                 mod LSTEND,LSTNAM,PRGRM+OBJCT,
                                  REENT+1,LSTENT,LSTMEM
000D 4C6973F4   LSTNAM   fcs    "List"

   * STATIC STORAGE OFFSETS
   *
00C8             BUFSIZ   equ    200          size of input buffer
0000                      ORG    0
0000             IPATH    rmb    1            input path number
0001             PRMPTR   rmb    2            parameter pointer
0003             BUFFER   rmb    BUFSIZ       allocate line buffer
00CB                      rmb    200          allocate stack
0193                      rmb    200          room for parameter list
025B             LSTMEM   EQU    .

0011 9F01        LSTENT   stx    PRMPTR       save parameter ptr
0013 8601                 lda    #READ.       select read access mode
0015 103F84               os9    I$OPEN       open input file
0018 252E                 bcs    LIST50       exit if error
001A 9700                 sta    IPATH        save input path number
001C 9F01                 stx    PRMPTR       save updated param ptr

001E 9600        LIST20   lda    IPATH        load input path number
0020 3043                 leax   BUFFER,U     load buffer pointer
0022 108E00C8             ldy    #BUFSIZ      maximum bytes to read
0026 103F8B               os9    I$RDLN       read line of input
0029 2509                 bcs    LIST30       exit if error
002B 8601                 lda    #1           load std. out. path #
002D 103F8C               os9    I$WRLN       output line
0030 24EC                 bcc    LIST20       Repeat if no error
0032 2014                 bra    LIST50       exit if error
0034 C1D3        LIST30   cmpb   #E$EOF       at end of file?
0036 2610                 bne    LIST50       branch if not
0038 9600                 lda    IPATH        load input path number
003A 103F8F               os9    I$CLOS       close input path
003D 2509                 bcs    LIST50       ..exit if error
003F 9E01                 ldx    PRMPTR       restore parameter ptr
0041 A684                 lda    0,X
0043 810D                 cmpa   #$0D         End of parameter line?
0045 26CA                 bne    LSTENT       ..no; list next file
0047 5F                   clrb
0048 103F06      LIST50   os9    F$EXIT       ... terminate

004B 95BB58               emod                Module CRC

004E             LSTEND   EQU    *
```

FILE : exer.c
PAGE : 1
TIME : 04/26/90 11:59

---

```
/*****************************************************************/
/*  Program to exercise Colombo board                            */
/*  Mario Trujillo                                               */
/*  exer.c                                                       */
/*****************************************************************/

#include <modes.h>
char *colombo;
int w;
int n,i,c;
main()
{
colombo="/c1";
w=2;
n=creat(colombo,w);
i=1234;
c=2;
write(n,&i,c);
}
```

## Processes.

- A common situation is that a process does not find input ready, when it needs it. It should then **block**, until input becomes available.
- IT must **not** do **busy-wait**! This occupies the CPU uselessly.
- Processes can be in one of **three states**:
  - **running** (e.g. using the CPU at this precise moment in time )
  - **blocked** (e.g. waiting for an external event or another process)
  - **ready** (e.g. ready to run, waiting to be scheduled for execution)

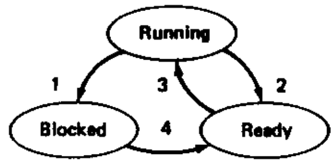




## Processes.

Note that OS-9 classifies process states somewhat differently.

- The diagram shows the possible transitions between the three states.








## Processes.

- The process model allows us to think in terms of user processes, disk processes, terminal I/O processes, etc, without having to consider **interrupts** and interrupt handling. (Obviously running various **processes concurrently** needs **interrupts**, generated by **I/O devices** and by a **clock**).
- Inside the operating system **kernel** is the **scheduler**, which takes care of the **interrupt handling** and schedules processes for execution. The rest of the operating system can be nicely structured in processes.





## Processes.

- The remaining part of the kernel only needs to contain initialization and utility routines used by other parts of the system (or the user).
- The scheduler uses a process table of some sort to manage the various processes. Every process which may run has an entry in this table. It contains **all the information** needed to restart the process exactly where it left off and with all states and conditions restored as they were at the moment the process was interrupted.

## Processes.

- The **scheduling algorithm** tries to satisfy one or more of the following -contradictory- criteria:
  - **fairness**
  - **efficiency**
  - **response time**
  - **turn around**
  - **throughput**





## Processes.

- For a real-time system controlling equipment, **response time** is generally the most important. If the system must respond within a given limit of time, we call it a **hard real-time** system.
- The scheduler cannot predict what a process is going to do, thus a **clock** must interrupt the system regularly, so that the scheduler may intervene.
- **Preemptive scheduling** allows temporary suspension of a running process, in contrast to **run to completion**.





## Processes.

- Many scheduling algorithms exist:
  - round robin: all processes get a time slice in turn.
  - priority: each process has an initial priority assigned. At each clock the scheduler may decrement the priority of the running process.
  - multiple priority classes. Within a class priority scheduling is applied.
  - shortest job first: run times must be known in advance, or estimated from past behaviour.
  - policy driven: a goal is fixed and the scheduler tries to live up to it (fairness, response time).
  - Two-level: needed when part of the runnable processes are on disk.





## Processes.

- Processes need often to communicate with each other. **Inter-process communication** is usually done with **messages** (called **signals** in OS-9).
- When a message is sent to a process, it will be delivered to it by the operating system. It is the responsability of the receiving process to interpret the message. Some messages are interpreted by the operating system itself (e.g. kill, wakeup).

## Processes in OS-9.

- **Kernel handles creation, scheduling etc. of processes.**
- All information on processes are kept in **Process Descriptors**: 64-byte structures (in level I). Details in **OS9sysdefs**.
- A process is in one of three possible states:
  - **active** (it can be run)
  - **waiting** (for a child to terminate, or for a signal)
  - **sleeping** (suspended for a given time, or until a signal is received).





## Processes in OS-9.

- Processes are queued in **three queues**, corresponding to these states. **Highest priority** process is at the **head** of the queue.
- Obviously a process may move from one queue to another.
- A new process is created with a **fork** system call. Main argument of fork is the name of the module. The module is loaded, a process descriptor set up and data storage allocated. Then the new (**child**) process is **put into the active queue**.





## Processes in OS-9.

- Every process has a unique **process ID**, which can be used by other processes (for inter-process communication).
- A process terminates when executing an **Exit** service request.





## Process Scheduling in OS-9.

- All **active processes** are assured of getting **some CPU time**.
- High-priority processes get more.
- Process **scheduling** is done at each clock tick.
- When a process is put into the active queue it enters with its "**age**" equal to its priority.
- The active process with the **oldest age** will be selected for execution; all other active processes will have their **age increased**.

## Inter-process Communication in OS-9.

- There is only **one mechanism** in OS-9 for **inter-process communication**: ***sending and receiving a signal.***
- A signal can be sent **to any process**. It consists of a **single byte**. A signal is sent using the **F$Send** service request and specifying the **process ID** of the receiving process.





## Inter-process Communication in OS-9.

- A signal is noted in the process descriptor of the receiving process. If it was **sleeping or waiting**, the receiving process becomes **active** and thus eligible for execution.
- If the receiving process has taken no special measures, to treat the signal, it will simply be **killed**.
- To process a signal properly, the receiving process must contain a **signal intercept routine** and the **address** of it must have been **communicated to the kernel** (with **F$Itcp**).





## Inter-process Communication in OS-9.

- the signal intercept routine can examine the signal code (in the B register) and take action.
- the signal codes defined are:
  - 0 = **kill** (non-interceptable)
  - 1 = **wakeup** (wakes up sleeping process, does not vector through intercept routine)
  - 2 = **keyboard abort**
  - 3 = **keyboard interrupt**
  - 4 – 255 **user definable.**





## Inter-process Communication in OS-9.

- the signal intercept routine must be short and **end with RTI.**
- an attempt to send a signal to a process which has already a signal pending will result in an error.
- We will come back later to this very important topic of inter-process communication and **synchronisation** (when we look at device drivers).

## Interrupts in OS-9.

- Interrupts are vectored through addresses in page 0 of memory.
- NMI and FIRQ are not used by OS-9, and are vectored to a RTI instruction.
- SWI, SWI2 and SWI3 are further vectored through addresses **local to the process** (specified in the process descriptor).
- The **clock** routine **changes the IRQ vector**, to its own address. So it gets quick service. It passes control to the IOMan's polling system when the interrupt did **not** originate from the clock.





## Interrupts in OS-9.

- This technique can be used for other interrupt sources as well (with moderation!).
- The **logical interrupt polling system** is prioritized.
- Each interrupting device has an entry in the **interrupt polling table** :
  - polling address (device's status register)
  - mask byte (selects relevant bit)
  - flip byte (selects positive or negative logic)
  - address of interrupt service routine
  - static storage address.
  - priority (0 is lowest, 255 highest)





## Interrupts in OS-9.

- A device is entered in the table with the F$IRQ service request.
- An interrupt service routine must end with RTS, **not** with RTI.





## Mutual Exclusion.

- In most **real-time systems various tasks run concurrently**, some of them may be **interrupt driven**.
- **Synchronisation** of the tasks is necessary.
- Problems arise if several tasks **access a variable** and can **modify its value**. (or claim a sharable resource).
- Example: real-time clock.
  - an interrupt driven routine maintains the time of the day in 3 bytes in memory: hh, mm, ss.

## Mutual Exclusion.

- another, independent task may read these bytes and display them.
- Now suppose hh:mm:ss have the values 11:59:59 when the second task is reading the time.
- Suppose a clock interrupt occurs when the task has read hh, but before it was able to read mm.
- What will be displayed?

- The access to the shared variable(s) is a **critical section** of the program and the two processes must **mutually exclude** access to this critical section.

- Access to hardware control registers of devices is also critical.





## Mutual Exclusion.

- Examples, where synchronisation and mutual exclusion are needed:
  - **car park** (of a supermarket) with several entrance gates and one or more exit gates, where barriers must be operated. Problems become serious when the car park is full.
  - **client-server** model (particularly relevant for real-time systems), where a server produces items and puts them into a buffer. The client takes items out of the buffer and consumes them. The critical sections are the updating of the buffer pointers. As long as the buffer is not full or empty no great harm is done. Serious problems when **buffer is full or empty**.





## Mutual Exclusion.

- Easy solution which **always works**: *disable interrupts before* entering and *enable interrupts after* leaving the critical section.

- This is **inadmissible** in a general purpose multi-user system. It may be dangerous in a **hard real-time system**.

- A **flag**, which is set by one process and read by the other **does not work**. Why?





## Mutual Exclusion.

CRITICAL REGION, USING NORMAL VARIABLE: FLAG

```
        PROCESS 1                 PROCESS 2
TEST1   *                 TEST2   *
        LDA FLAG                  LDA FLAG
        BEQ GOON1                 BEQ GOON2
        GO TO SLEEP               GO TO SLEEP
        BRA TEST1                 BRA TEST2
GOON1   INCA              GOON2   INCA
        STA FLAG                  STA FLAG
        USE RESOURCE              USE RESOURCE
        CLR FLAG                  CLR FLAG
```

## Mutual Exclusion.

- An **indivisible** (or **atomic**) **test-and-set** instruction is needed to guard the entrance to a critical section. TAS instruction will test a variable and if it is equal to zero, will set it to one. If it is already one it will leave it unchanged. The whole operation must be **uninterruptable**.
- 6809 has no TAS, but **LSR using a memory location** can replace it.





## Mutual Exclusion.

CRITICAL REGION, USING LSR.

```
        PROCESS 1                 PROCESS 2
TEST1   *                 TEST2   *
        LSR FLAG                  LSR FLAG
        BCS GOON1                 BCS GOON2
        GO TO SLEEP               GO TO SLEEP
        BRA TEST1                 BRA TEST2
GOON1   USE RESOURCE      GOON2   USE RESOURCE
        LDA #1                    LDA #1
        STA FLAG                  STA FLAG
```





## Mutual Exclusion.

- What does a process do when access to a critical section is denied to it? (Or when a server finds the buffer full?)
- Easiest and very **inefficient solution**: **busy-wait**.
- **Better solution:** The process which cannot get access should **go to sleep**, and be **woken-up** later, when access becomes free.





## Mutual Exclusion.

- It is important to realize that the TAS (or equivalent) instruction is in itself **not enough** to handle all possible situations. It is useful help given by the hardware.
- Several mechanisms have been invented:
  - Dekkers' algorithm (very complicated)
  - Dijkstra's **semaphore**
  - **Event counters**
  - **Monitors** (a **language construct**)
  - **Message passing**
  - Ada rendez-vous (another language construct).

```
procedure lock (var my_flag, his_flag :boolean; me :boolean);
begin
        my_flag := true;

        if his_flag then  {other process in or entering region}
                if turn = me then
                        while his_flag do;          {wait for other}
                                {process to leave or to concede}
                                {clear to enter}
                else
                        begin   {concede to other process}
                                my_flag := false;
                                while turn <> me do;      {wait till}
                                        {other process leaves}
                                my_flag := true;
                                while his_flag do;        {try again}
                                        {now clear to enter}
                        end
end;

procedure unlock (var my_flag : boolean; me : boolean);
begin
        turn    := not me;          {priority to other process}
        my_flag := false {release lock}
end;
```

Fig. 2.2 - Dekker's algorithm.

An example of the use of the protocol to resolve access is given below; it assumes that the value of *turn* is *true*, thus giving priority to *Process_1*. The example illustrates the most difficult case of simultaneous access attempts.

| *Process_1* | *Process_2* |
|---|---|
| *set own flag* | *sets own flag* |
| *test other flag - true* | *test other flag - true* |
| | |
| *test turn - me* | *test turn - not me* |
| *test other flag - true* | *reset own flag* |
| *test other flag - false* | *test turn - not me* |
| *{enter region}* | *test turn - not me* |
| ... | *test turn - not me* |
| *{leave region}* | *test turn - not me* |
| | |
| *set turn not me* | *test turn - me* |
| *reset own flag* | *set own flag* |
| ... | *test other flag - false* |
| ... | *{enter region}* |
| | ... |
| | *etc.* |

If *Process_1* had attempted to reenter the region immediately after it had exited and while *Process_2* was still executing the entry protocol, the value of the variable *turn* would have forced it to wait for the other process, thus ensuring fair access.

```
#define N   100                  /* number of slots in the buffer */
int count = 0;                   /* number of items in the buffer */

producer()
{
  while (TRUE) {                 /* repeat forever */
        produce_item();          /* generate next item */
        if (count == N) sleep(); /* if buffer is full, go to sleep */
        enter_item();            /* put item in buffer */
        count = count + 1;       /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);      /* was buffer empty? */
  }
}


consumer()
{
  while (TRUE) {                 /* repeat forever */
        if (count == 0) sleep(); /* if buffer is empty, got to sleep */
        remove_item();           /* take item out of buffer */
        count = count - 1;       /* decrement count of items in buffer */
        if (count == N-1) wakeup(producer);    /* was buffer full? */
        consume_item();          /* print item */
  }
}
```



```
#define N 100                       /* number of slots in the buffer */
typedef int semaphore;              /* semaphores are a special kind of int */
semaphore mutex = 1;                /* controls access to critical region */
semaphore empty = N                 /* counts empty buffer slots */
semaphore full = 0;                 /* counts full buffer slots */

producer()
{
  int item;

  while (TRUE) {                    /* TRUE is the constant 1 */
        produce_item(&item);        /* generate something to put in buffer */
        down(empty);                /* decrement empty count */
        down(mutex);                /* enter critical region */
        enter_item(item);           /* put new item in buffer */
        up(mutex);                  /* leave critical region */
        up(full);                   /* increment count of full slots */
  }
}


consumer()
{
  int item;

  while (TRUE) {                    /* infinite loop */
        down(full);                 /* decrement full count */
        down(mutex);                /* enter critical region */
        remove_item(&item);         /* take item from buffer */
        up(mutex);                  /* leave critical region */
        up(empty);                  /* increment count of empty slots */
        consume_item(item);         /* do something with the item */
  }
}
```

```c
#define N     100               /* number of slots in the buffer */
typedef int event_counter;      /* event_counters are a special kind of int */
event_counter in;               /* counts items inserted into buffer */
event_counter out;              /* counts items removed from buffer */

producer()
{
  int item, sequence = 0;

  while (TRUE) {                /* infinite loop */
        produce_item(&item);    /* generate something to put in buffer */
        sequence = sequence + 1; /* count items produced so far */
        await(out, sequence - N); /* wait until there is room in buffer */
        enter_item(item);       /* put item in slot (sequence-1) % N */
        advance(in);            /* let consumer know about another item */
  }
}


consumer()
{
  int item, sequence = 0;

  while (TRUE) {                /* infinite loop */
        sequence = sequence + 1; /* number of item to remove from buffer */
        await(in, sequence);    /* wait until required item is present */
        remove_item(&item);     /* take item from slot (sequence-1) % N */
        advance(out);           /* let producer know that item is gone */
        consume_item(item);     /* do something with the item */
  }
}
```

**Fig. 2-12.** The producer-consumer problem using event counters.

: a monitor written in an imaginary language, pidgin Pascal.

**monitor** *example*
  **integer** *i*;
  **condition** *c*;

  **procedure** *producer*(*x*);
  .
  .
  .
  **end**;

  **procedure** *consumer*(*x*);
  .
  .
  .
  **end**;
**end monitor**;

**Fig. 2-13.** A monitor.

```
#define N 100                   /* number of slots in the buffer */

producer()
{
  int item;
  message m;                    /* message buffer */

  while (TRUE) {
        produce_item(&item);     /* generate something to put in buffer */
        receive(consumer, &m);   /* wait for an empty to arrive */
        build_message(&m, item); /* construct a message to send */
        send(consumer, &m);      /* send item to consumer */
  }
}

consumer()
{
  int item, i;
  message m;

  for (i = 0; i < N; i++) send(producer, &m);   /* send N empties */
  while (TRUE) {
        receive(producer, &m);   /* get message containing item */
        extract_item(&m, &item);        /* take item out of message */
        consume_item(item);      /* do something with the item */
        send(producer, &m);      /* send back empty reply */
  }
}
```

**Fig. 2-16.** The producer-consumer problem with *N* messages.

```
producer
while more_data do
    begin
        <produce data>
        wait (lock);
        ...
        <insert_data_into_buffer>
        ...
        signal (lock);
        signal (data_available)
    end;
and
consumer
while more_data do
    begin
        wait (data_available);
        wait (lock);
        ...
        <extract_data_from_buffer>;
        ...
        signal (lock);
        <consume_data>;
    end.
```

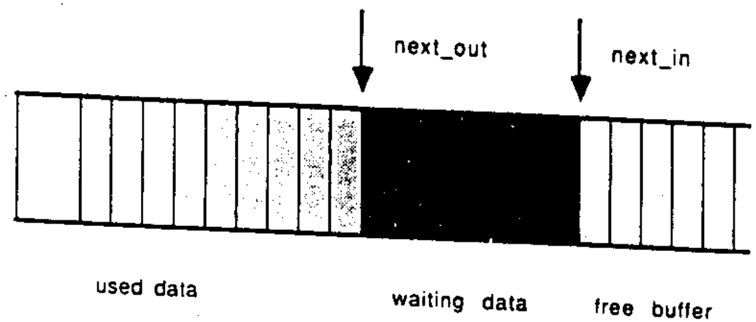



Fig. 2.4 - The unbounded buffer.



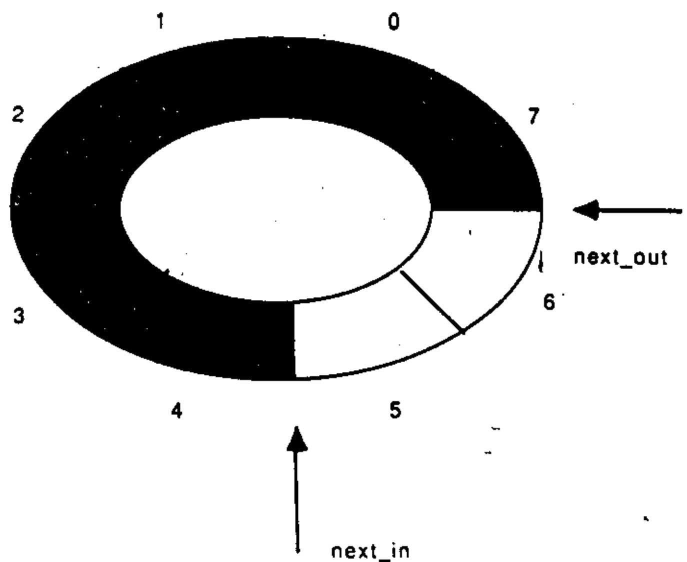



Fig. 2.5 - A cyclic buffer.

```
        producer

    while more_data do
        begin
            <produce_data>;
            wait (room_available);
            wait (lock);
            ...
            <insert_data_in_buffer>;
            ...
            signal (lock);
            signal (data_available);
        end;
and
        consumer

    while more_data do
        begin
            wait (data_available);
            wait (lock);
            ...
            <extract_data_from_buffer>;
            ...
            signal (lock);
            signal (room_available);
            <consume_data>;
        end;
```

## Mutual Exclusion.

- Note that these methods will require in most cases considerable effort by the programmer.
- Also note that **none of these methods** will automatically **prevent deadlock** or **starvation.**
- Simple example of deadlock:
  - process A holds resource X and needs Y
  - process B holds resource Y and needs X.





## Mutual Exclusion in OS-9.

- In OS-9 the **only mechanism** to obtain synchronization is **message passing.**
- It relies on **disabling interrupts** for short periods (done by OS-9).
- We will see how this mechanism is used in OS-9 when we look at the **anatomy of a device driver**.
- Making use of the LSR instruction, we can **implement general semaphores** in OS-9. We could put them in the **C library** or even implement them as **new service requests**.





## Input-Output.

- We usually distinguish between **block devices** and **character devices**.
- The hardware of the device is interfaced to the computer via a **device controller** or **adapter**. In most cases a specialized chip: disk controller, ACIA, PIA, etc.
- The device is controlled by writing **commands** into registers of the controller, by reading back **status** information. Data is transfered by reading/writing from/to the controller's **data register(s)**.





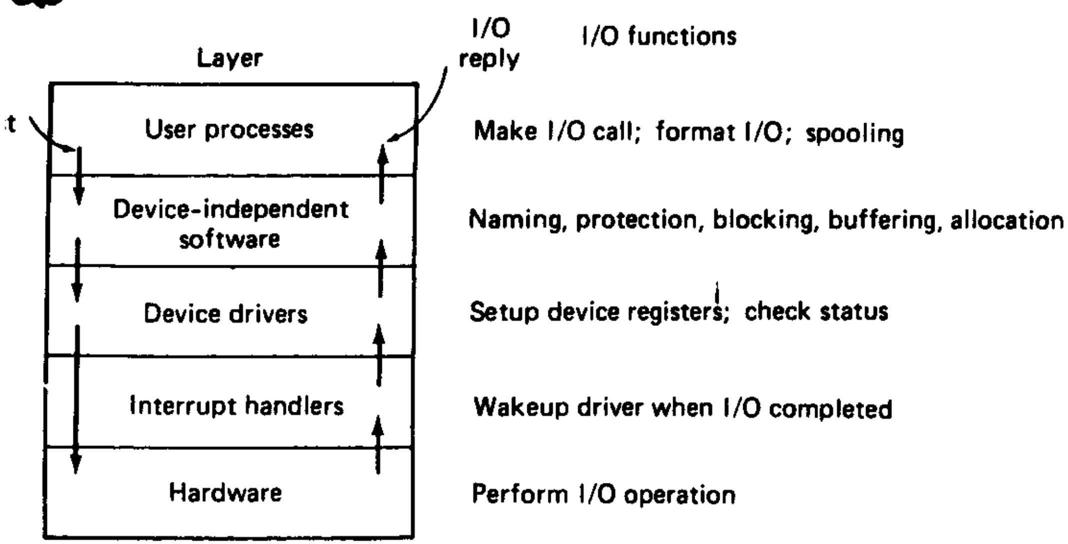
## Input-Output.

- Reading and writing the registers of a controller is done with special I/O instructions (Intel), or using **memory-mapping**. In the latter case, normal load and store instructions are used (DEC, Motorola).
- **Direct Memory Access** is done entirely by the controller. The transfer is set up by software. Data do not transit through the CPU. The completion of a **block transfer** usually generates an interrupt.

| |
|---|
| Uniform interfacing for the device drivers |
| Device naming |
| Device protection |
| Providing a device-independent block size |
| Buffering |
| Storage allocation on block devices |
| Allocating and releasing dedicated devices |
| Error reporting |

**Fig. 3-5.** Functions of the device-independent I/O software.

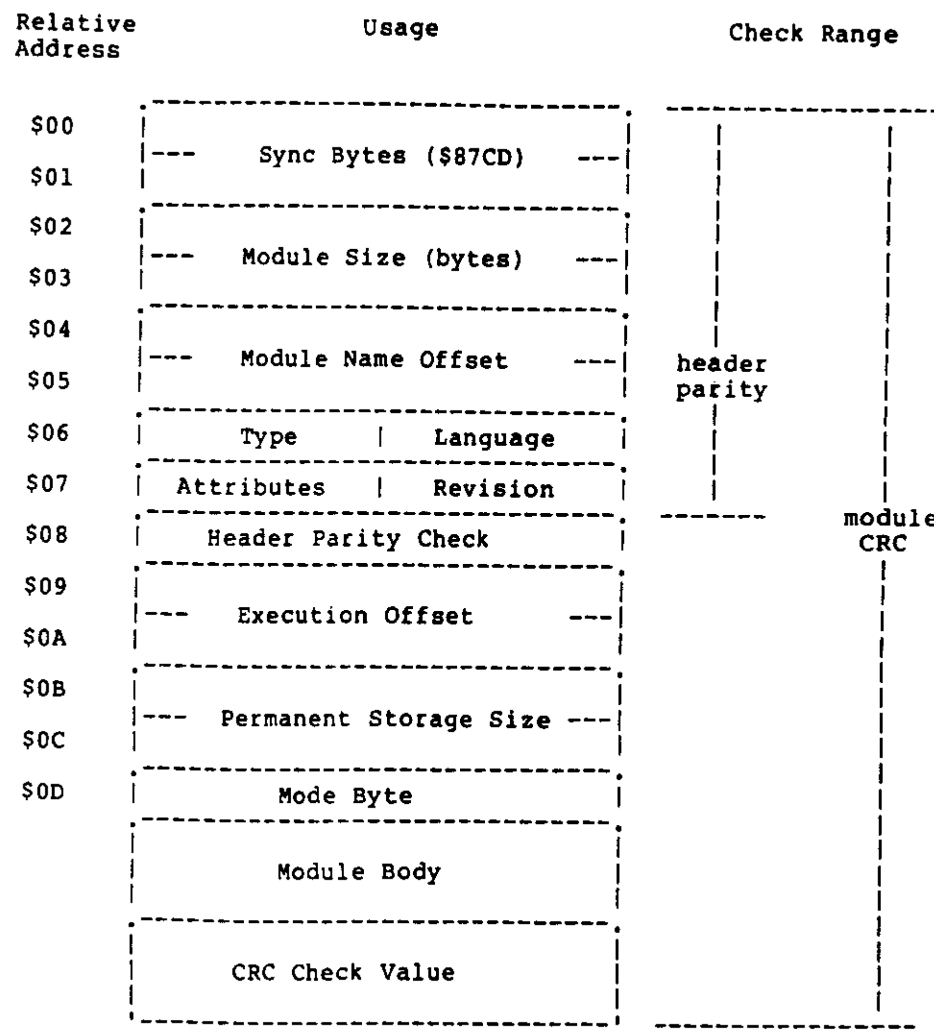
38




**ig. 3-6.** Layers of the I/O system and the main functions of each layer.



**DEVICE DRIVER MODULE FORMAT**




$D Mode Byte - (D S PE PW PR E W R)

## I/O Software.

- I/O software should fulfill two goals:
  - **hide** the peculiarities of the **hardware**
  - present a nice, clean and **regular interface** to the user.
- This leads naturally to a layered structure:
  - **interrupt handlers**
  - **device drivers**
  - **device-independent operating system software**
  - **user-level software**.





## I/O Software.

- Another responsibility of I/O software is to handle errors.
- I/O software should also take account of the type of device: **sharable** (disks) or **dedicated** (printers, terminals).
- **device independence** means that the user should see no difference between writing to a file on a hard disk or writing on a printer.
- The **device driver's task** is to **receive abstract orders** from the software above and then to see to it that the job gets done.





## I/O software.

- This means: translating the request into commands to the controller ( start motor, move arm, set up DMA, etc., etc.) and issue them. If the execution of the command takes time, the driver must **block** (go to **sleep**), until an interrupt will cause it to be woken up. Finally errors are checked and status information passed back to the caller.





## I/O software.

- The **device-independent I/O software** is responsible for:
  - uniform interfacing to the drivers
  - translating device names into selection of the appropriate driver
  - protection
  - buffering
  - storage allocation on disks
  - allocation and release of dedicated devices
  - error reporting.

## I/O software.

- Library functions, which are linked into user-programs, do the remaining part of the input-output. Some library routines simply pass parameters on to a system call (e.g. *read, write*), others do more work (e.g. *printf, scanf*).
- A final part of I/O software is a **spooling system**. Files to be printed are put in the spooling directory. A printer **daemon** is the only process allowed to access the printer.





## Deadlocks.

- In a multi-programming system, where non-sharable resources are allocated to processes, **deadlock** situations may occur.
- Deadlocks have been extensively studied, but the subject is not very important for real-time control or embedded systems, where dynamic allocation of non-sharable resources is rare.





## Deadlocks.

- Different aspects of the problem are:
  - detection and recovery
  - prevention (by imposing rules on the processes)
  - avoidance (using an algorithm to make the right choice when resources must be allocated (the Banker's algorithm).





## Input/Output in OS-9.

- Device independence is obtained by splitting into **four levels**:
  - **IOMan** manages **all** input/output
  - **File Managers** handle a class of devices, without regard to device characteristics.

EXAMPLES: RANDOM BLOCK FILE MANAGER (RBF)
SEQUENTIAL BLOCK FILE MANAGER (SBF)
SEQUENTIAL CHARACTER FILE MAN (SCF)
PIPE MANAGER (PIPEMAN)

## Input/Output in OS-9.

- **Device drivers** for doing low-level I/O transfers from/to a specific type of hardware controller (disk controller, ACIA)
- **Device descriptors** specify characteristics of individual devices.

- File managers are re-entrant and can handle a whole class of devices with similar operational characteristics.

- Responsible for buffering of data, mass-storage allocation and directory services, processing of data stream.



## Input/Output in OS-9.

- A file manager has many entry points:
  - Create
  - Open
  - MakDir
  - ChgDir
  - Delete
  - Seek
  - Read
  - Write
  - ReadLn
  - WriteLn
  - Getstat
  - Putstat
  - Close





## Device Drivers in OS-9.

- Device drivers are **re-entrant** and can control several **hardware controllers** of the **same type**.

BUT: ACIA FOR THE MOTOROLA 6850 CHIP AND ACIA51 FOR THE SIGNETICS 6551





## Device Drivers in OS-9.

- Device driver has **six entry points**:
  - Initialize
  - Read
  - Write
  - Get device status
  - Set device status
  - terminate

- Parameters passed and precise actions depend on the file manager and the hardware controller.

- We will treat in more detail later the synchronisation problem.

```
***************
*  IOEXEC
*    Execute Device's Read/Write routine

*  Passed: (A)=output char (write)
*          (X)=Device Table entry ptr
*          (Y)=Path Descriptor ptr
*          (U)=offset of routine (D$Read, D$Write)
* Returns: (A)=Input char (read)
*          (B)=error code, CC set if error
* Destroys B,CC

IOEXEC pshs a,x,y,u save registers
 ldu V$STAT,x get static storage for driver
 ldx V$DRIV,x get driver module address
 ldd M$EXEC,x and offset of execution entries
 addd 5,s offset by read/write
 leax d,x absolute entry address
 lda ,s+ restore char (for write)
 jsr 0,x execute driver read/write
 puls x,y,u,pc return (A)=char, (B)=error

 emod Module CRC
Size equ * size of Sequential File Manager
```





DEVICE DESCRIPTOR MODULE FORMAT

| MODULE OFFSET | DEVICE DESCRIPTOR MODULE FORMAT | | |
|---|---|---|---|
| $0, $1 | Sync Bytes ($87CD) | header parity | module CRC |
| $2, $3 | Module Size | header parity | module CRC |
| $4, $5 | Offset to Module Name | header parity | module CRC |
| $6 | $F (TYPE) ! $1 (LANG) | header parity | module CRC |
| $7 | Attributes ! Revision | header parity | module CRC |
| $8 | Header Parity Check | | module CRC |
| $9, $A | Offset to File Manager Name String | | module CRC |
| $B, $C | Offset to Device Driver Name String | | module CRC |
| $D | Mode Byte | | module CRC |
| $E, $F, $10 | Device Controller Absolute Physical Address (24 bit) | | module CRC |
| $11 | Initialization Table Size | | module CRC |
| $12,$12.N | (Initialization Table) | | module CRC |
| | (Name Strings etc) | | module CRC |
| | CRC Check Value | | |

## Universal Path Descriptor Definitions

| Name | Addr | Size | Description |
|---|---|---|---|
| PD.PD | $00 | 1 | Path number |
| PD.MOD | $01 | 1 | Access mode: 1=read 2=write 3=update |
| PD.CNT | $02 | 1 | Number of paths using this PD |
| PD.DEV | $03 | 2 | Address of associated device table entry |
| PD.CPR | $05 | 1 | Requester's process ID |
| PD.RGS | $06 | 2 | Caller's MPU register stack address |
| PD.BUF | $08 | 2 | Address of 256-byte data buffer (if used) |
| PD.FST | $0A | 22 | Defined by file manager |
| PD.OPT | $20 | 32 | Reserved for GETSTAT/SETSTAT options |

The 22 byte section called "PD.FST" is reserved for and defined by each type of file manager for file pointers, permanent variables, etc.

The 32 byte section called "PD.OPT" is used as an "option" area for dynamically-alterable operating parameters for the file or device. These variables are initialized at the time the path is opened by copying the initialization table contained in the device descriptor module, and can be altered later by user programs by means of the GETSTAT and SETSTAT system calls.

"PD.OPT" and "PD.FST" sections are defined for each file manager in the assembly language equate file (OS9SCFDefs for SCFMAN and OS9RBFDefs for RBF).

```
MODULE
OFFSET              ORG $12

        TABLE    EQU .    beginning of option table
$12     IT.DVC   RMB 1    device class (0=scf 1=rbf 2=pipe 3=sbf)
$13     IT.UPC   RMB 1    case (0=both, 1=upper only)
$14     IT.BSO   RMB 1    back space (0=bse, 1=bse,sp,bse)
$15     IT.DLO   RMB 1    delete (0=bse over line, 1=cr)
$16     IT.EKO   RMB 1    echo (0=no echo)
$17     IT.ALF   RMB 1    auto line feed (0= no auto lf)
$18     IT.NUL   RMB 1    end of line null count
$19     IT.PAU   RMB 1    pause (0= no end of page pause)
$1A     IT.PAG   RMB 1    lines per page
$1B     IT.BSP   RMB 1    backspace character
$1C     IT.DEL   RMB 1    delete line character
$1D     IT.EOR   RMB 1    end of record character
$1E     IT.EOF   RMB 1    end of file character
$1F     IT.RPR   RMB 1    reprint line character
$20     IT.DUP   RMB 1    dup last line character
$21     IT.PSC   RMB 1    pause character
$22     IT.INT   RMB 1    interrupt character
$23     IT.QUT   RMB 1    quit character
$24     IT.BSE   RMB 1    backspace echo character
$25     IT.OVF   RMB 1    line overflow character (bell)
$26     IT.PAR   RMB 1    initialization value (parity)
$27     IT.BAU   RMB 1    baud rate
$28     IT.D2P   RMB 2    attached device namestring offset
$2A     IT.XON   RMB 1    xon character
$2B     IT.XOFF  RMB 1    xoff character
$2C     IT.STN   RMB 2    offset to status routine
$2E     IT.ERR   RMB 1    initial error status
```

## Device Descriptor Modules.

- Non-executable: contain tables.
- Information in a device descriptor:
  - name of device
  - name of device driver
  - name of file manager
  - hardware controller address
  - initialization parameters
- The initialization parameters are copied to the **path descriptor** when a path to the device is opened. They can be changed using **I$Getstt** and **I$SetStt**. (For instance, you may change control characters for terminal, or turn page pause on or off, etc.).





## Memory management.

- The **aim of memory management** is to make **best use of available memory** and **to keep the CPU busy**.
- Two classes:
  - **without swapping** or **paging**: a process stays in memory until finished.
  - **with swapping or paging**: processes are moved between memory and disk, during "execution" of the process.
- The simple mono-programming case is of no interest.





## Memory management.

- Multiprogramming is more complicated:
  - p = probability of process being idle (waiting for I/O). With n processes in memory $p^n$ is probability that CPU is idle. For p=0.8 (not unusual at all!) , n must be 10 for idle time to be less than 10%.
- A multiprogramming system without swapping will need a large memory, which can be divided into **fixed size partitions** (not necessarily all of the same size) or **variable size partitions**.





## Memory management.

- In all cases programs must be **relocated** as the memory address where it will run is not known at compile-time.
- Also, the partitions should be **protected**, to avoid that a bug in program A destroys program B in memory. Protection needs special hardware (**base** and **limit** registers for instance).

## Memory management.

- Fixed partition schemes may under-use memory, variable partition schemes will leave "holes" in memory when processes finish.
- The holes will be filled only partially by new processes. Memory **fragmentation** may occur, where all holes are too small to receive a reasonable program. Memory **compaction** combines all holes into one large hole.
- An extra complication is that data and stack areas may grow during execution (think of malloc()). So a process may grow out of its seams.





## Swapping.

- Some of these problems may be aleviated if processes may be swapped from memory to disk (when they have to wait for I/O for instance) and brought back into memory later.
- Variable partitions may again be used. To keep track of where things are and of free memory space, different techniques are used:
  - bit maps. Each bit in the bit map represents a fixed size of memory.
  - linked lists. The list is sorted by address and links processes and holes.
  - buddy system.





## Swapping.

- When a process must be brought into memory, the **memory manager** must find a hole where to put it. Four algorithms:
  - first fit
  - next fit
  - best fit
  - worst fit.





## Virtual Memory.

- Total size of program, data and stack may exceed size of memory. Keep those parts needed now in memory and the rest on disk. When a piece now on disk is needed, bring it into memory, throwing out (maybe) a piece no longer needed.
- When these things happen without the user being aware of it, we have a **virtual memory** system.
- **Virtual memory** and multiprogramming go very well together: when process A is swapped out, because it is waiting for I/O, another process may run.

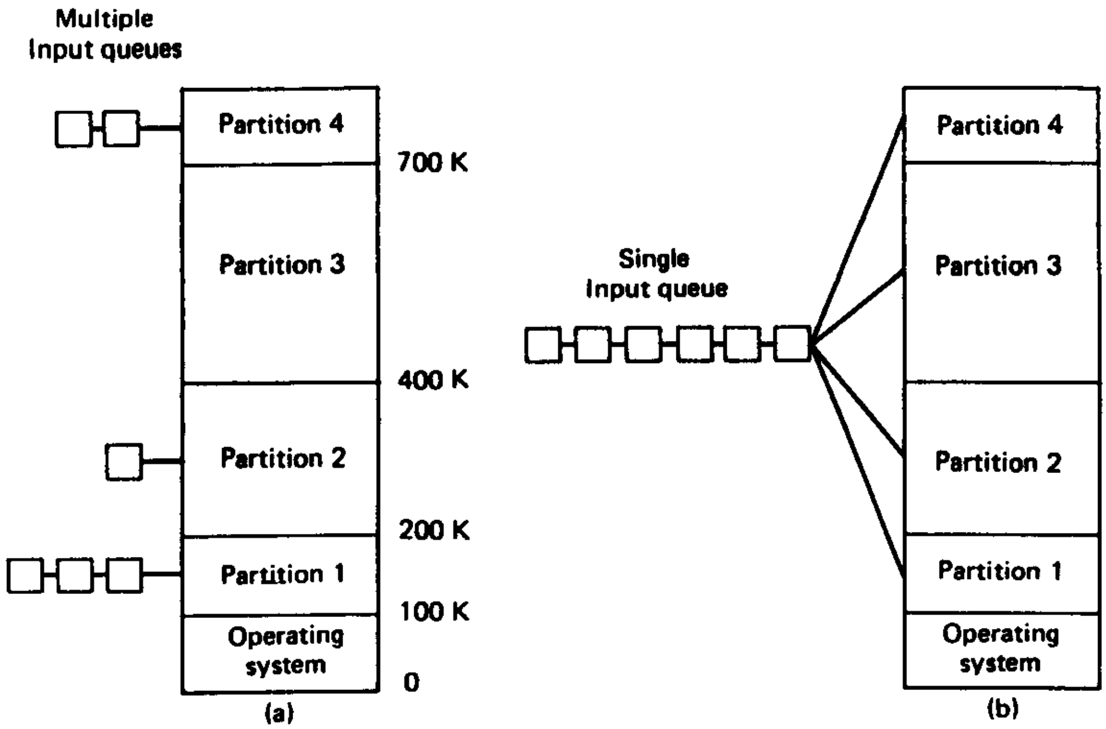



**Fig. 4-4. (a) Fixed memory partitions with separate input queues for each partition. (b) Fixed memory partitions with a single input queue.**

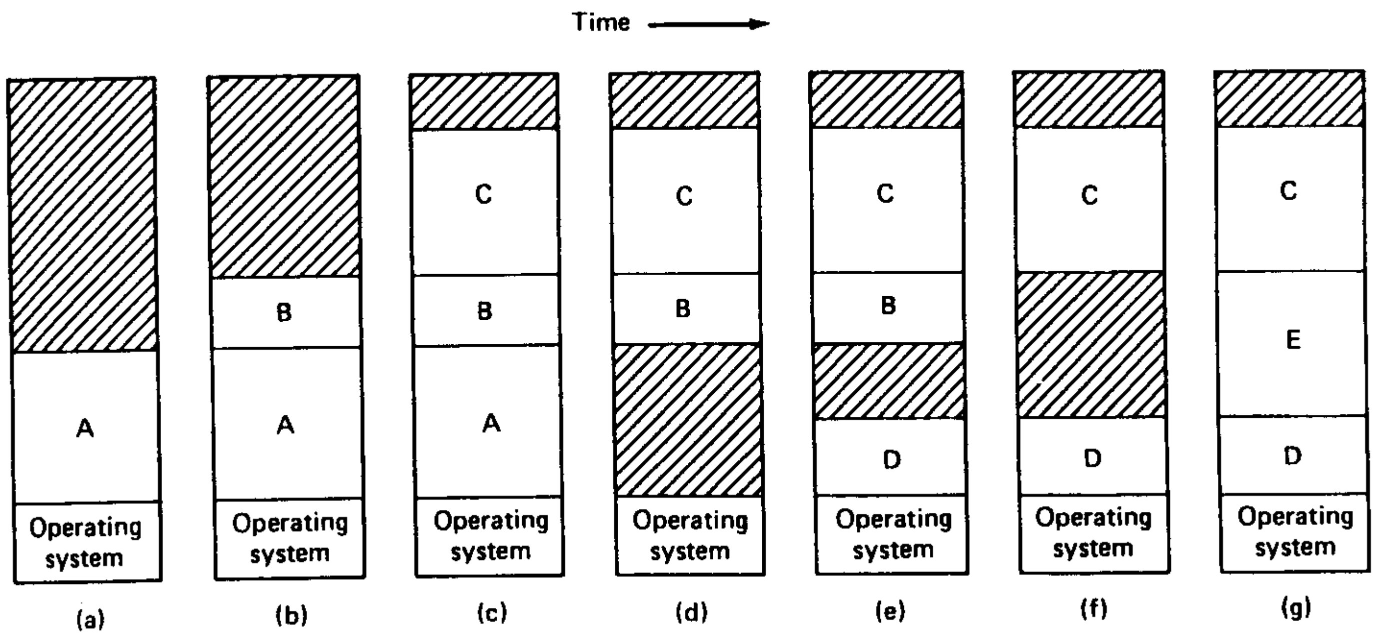



**Fig. 4-5.** Memory allocation changes as processes come into memory and leave it. The gray regions are unused memory.

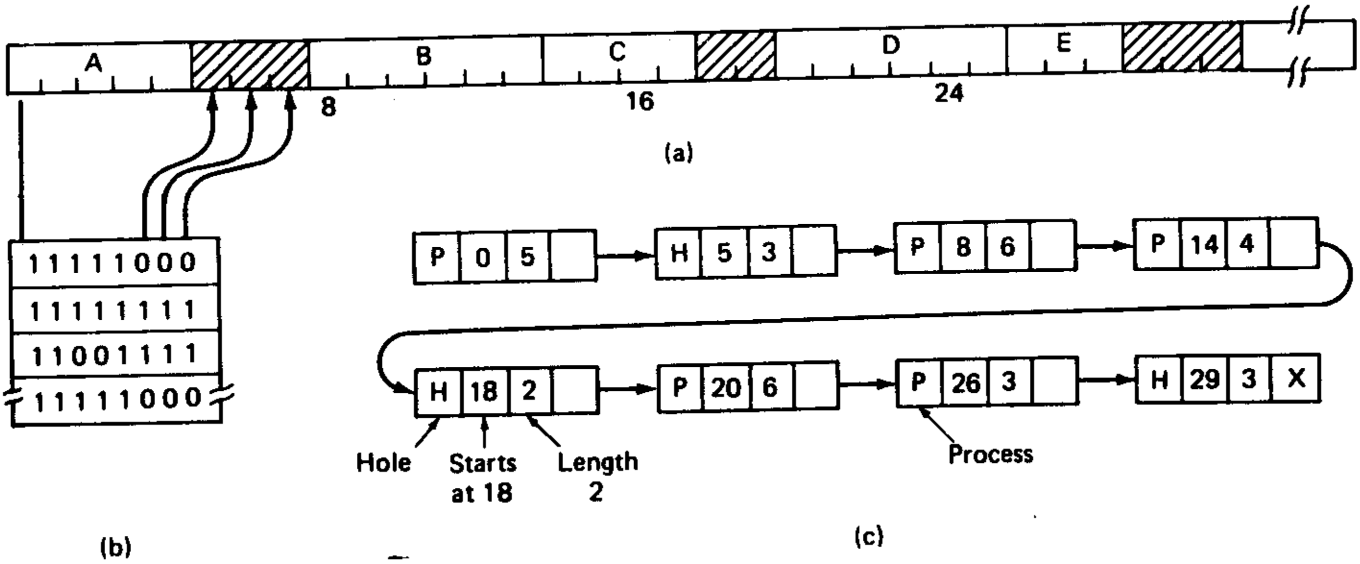



**Fig. 4-7.** (a) A part of memory with five processes and 3 holes. The tick marks show the memory allocation units. The shaded regions (0 in the bit map) are free. (b) The corresponding bit map. (c) The same information as a linked list.

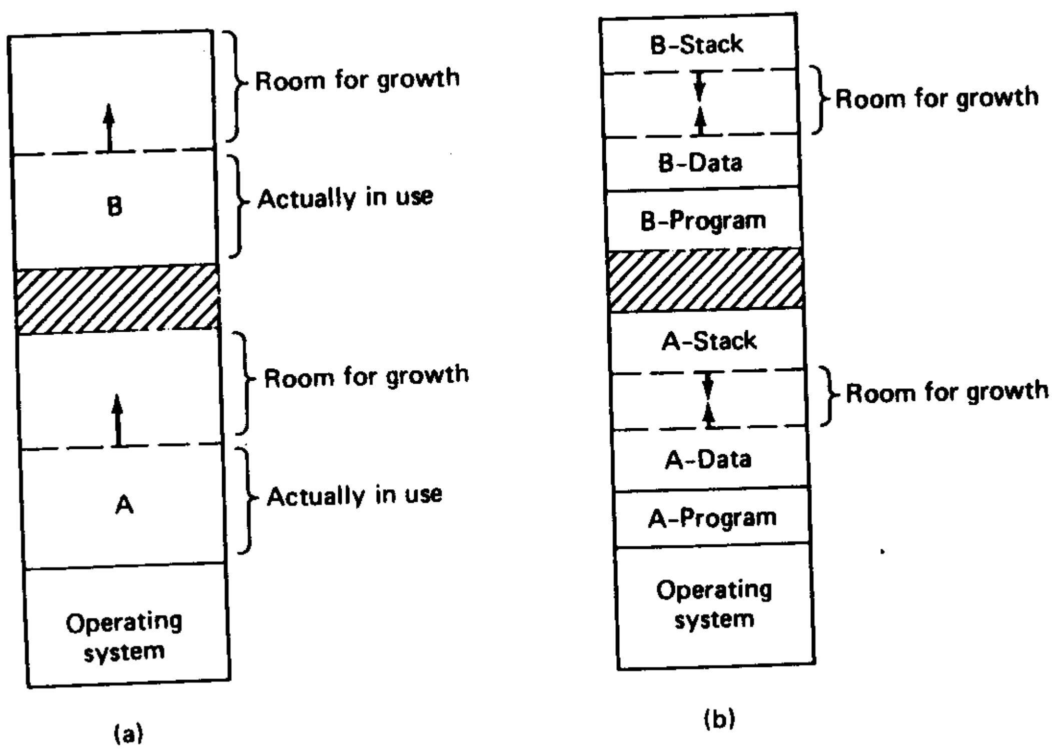



**Fig. 4-6.** (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

## Virtual Memory.

- Most **virtual memory** systems use **paging Virtual addresses** (the addresses the program uses) are translated into **physical addresses** by a **Memory Management Unit**.
- A **page fault** occurs when the program issues a virtual address in the range of an **unmapped** page (e.g. for which no physical address exists). This page is now brought into memory. If it is necessary to make room, another rarely used page is written to disk.





## Virtual Memory.

- Several **page replacement algorithms** exist to choose the page to be thrown out.
- The ideal, but unrealisable, algorithm would throw out the page that will not be used before long in the future.
- Realisable algorithms are:
  - not-recently-used page replacement (NRU)
  - first-in first-out replacement (FIFO)
  - least recently used page replacement (LRU).





## Memory Management in OS-9.

- Memory is **allocated** when:
  - a module is loaded
  - a new process is created (forked)
  - a process requests more memory
  - OS-9 needs more I/O buffers or needs to expand its data structures.
- Memory is **de-allocated** when the link count of a module goes to zero.







## Memory Management in OS-9.

- Level II makes use of **hardware MMU**.
- Level I uses a **first fit algorithm**.
- Memory fragmentation is a potential problem in a multi-user system. For a single user fragmentation is less of a problem. It can often be avoided by loading device drivers first!
- Modules in memory have a **link count**. A module can be removed from memory only when its link count is zero.

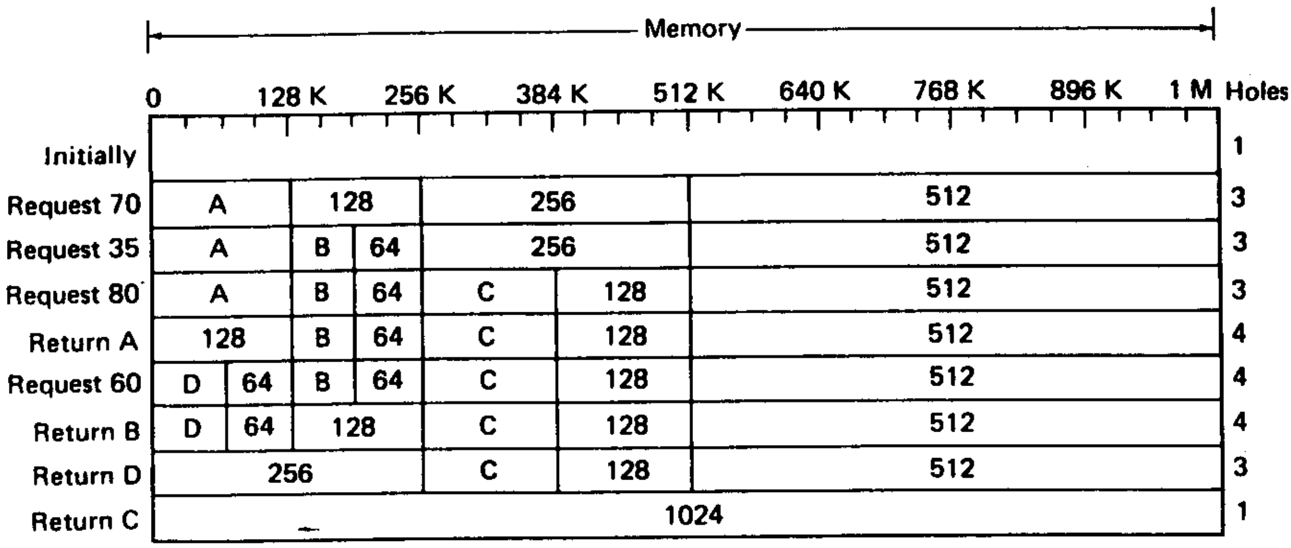



**Fig. 4-9.** The buddy system. The horizontal axis represents memory addresses. The numbers are the sizes of unallocated blocks of memory in K. The letters represent allocated blocks of memory.

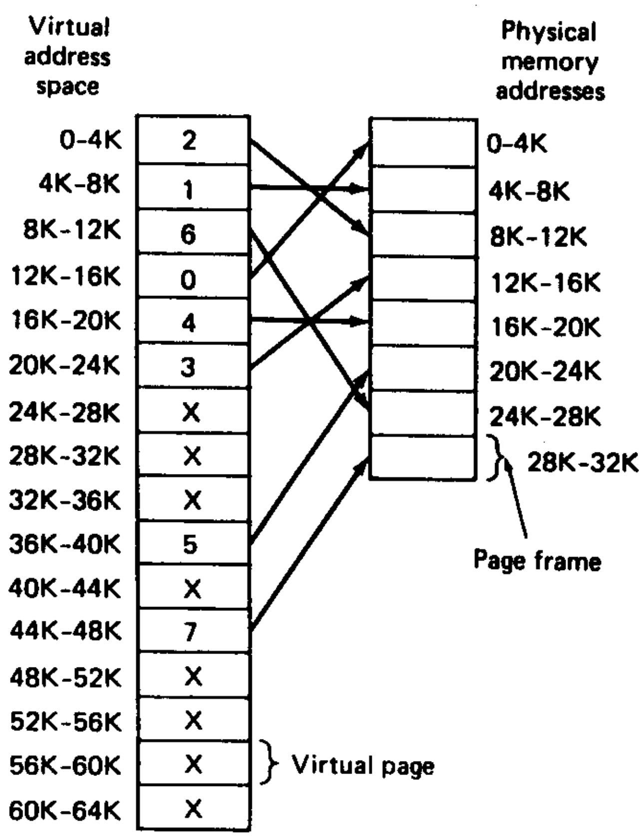



**Fig. 4-11.** The relation between virtual addresses and physical memory addresses is given by the page table.

## Path Descriptors

- 64 byte structures allocated and deallocated by IOMan when a path is opened or closed.
- First 10 bytes have same meaning for all paths.
- Then 22 bytes defined by file manager (see **OS9rbfdefs** and **OS9scfdefs**).
- Finally 32 option bytes, copied from device descriptor and alterable with I$SetStt. (see OS9rbfdefs and OS9scfdefs)





## Logical and Physical disk structure.

- A disk is divided into 256 byte sectors, with **Logical Sector Numbers (LSN)**.
- **rbf** uses **LSNs**, which are translated by the device driver into **physical location**: side, track, sector.
- **entire sectors** are transfered.
- Track 0, side 1 on a disk is (nearly) always single density and 10 sectors.
- All other tracks are usually double density and 16 sectors.
- **LSN0**(side 1, track 0, sector 0) is the **identification sector**.





## Logical and Physical disk structure.

- LSN1 and usually also LSN2 contain the **sector allocation map**. **One bit per sector**: "1"= in use, "0"=free.
- The **root directory** immediately follows the allocation map. Usually **LSN3**.
- Every file starts with a **file descriptor sector**, followed by the necessary number of sectors to contain the information.
- First byte of FD sector contains the **file attributes**:

```
D  S  PE  PW  PR  E  W  R
```





## Logical and Physical disk structure.

- A **directory** is like any other file, only difference is that **D is set**.
- An **entry** in a directory file is **32 bytes**: 29 for the **name**, and 3 for **LSN** of the FD sector of the file.
- The **RAM disk** is set up by copying 4 sectors (LSN0–3; ID, map, FD of root directory and root directory) into RAM on the ROM-RAM disk board.
- These sectors are copied from the top of the ROM memory (capacity of the ROM = 2560 sectors, of a floppy disk = 2554 sectors).

### 6.1.1 Identification Sector

Logical sector number zero contains a description of the physical and logical characteristics of the volume which are established by the "format" command program when the media is initialized. The table below gives the OS-9 mnemonic name, byte address, size, and description of each value stored in this sector.

| name | addr | size | description |
|---|---|---|---|
| DD.TOT | $00 | 3 | Total number of sectors on media |
| DD.TKS | $03 | 1 | Number of sectors per track |
| DD.MAP | $04 | 2 | Number of bytes in allocation map |
| DD.BIT | $06 | 2 | Number of sectors per cluster |
| DD.DIR | $08 | 3 | FD sector of root directory |
| DD.OWN | $0B | 2 | Owner's user number |
| DD.ATT | $0D | 1 | Disk attributes |
| DD.DSK | $0E | 2 | Disk identification (for internal use) |
| DD.FMT | $10 | 1 | Disk format: density, number of sides |
| DD.SPT | $11 | 2 | Number of sectors per track. |
| DD.RES | $13 | 2 | Reserved for future use |
| DD.BT | $15 | 3 | Starting sector of bootstrap file |
| DD.BSZ | $18 | 2 | Size of bootstrap file (in bytes) |
| DD.DAT | $1A | 5 | Time of creation: Y:M:D:H:M |
| DD.NAM | $1F | 32 | Volume name |
| DD.OPT | $3F | 32 | Path descriptor options |



### 6.1.3 File Descriptor Sectors

The first sector of every file is called a "file descriptor", which contains the logical and physical description of the file. The table below describes the contents of the descriptor.

| name | addr | size | description |
|---|---|---|---|
| FD.ATT | $0 | 1 | File Attributes: D S PE PW PR E W R |
| FD.OWN | $1 | 2 | Owner's User ID |
| FD.DAT | $3 | 5 | Date Last Modified: Y M D H M |
| FD.LNK | $8 | 1 | Link Count |
| FD.SIZ | $9 | 4 | File Size (number of bytes) |
| FD.Creat | $D | 3 | Date Created: Y M D |
| FD.SEG | $10 | 240 | Segment List: see below |

The attribute byte contains the file permission bits. Bit 7 is set to indicate a directory file, bit 6 indicates a "nonsharable" file, bit 5 is public execute, bit 4 is public write, etc.

## Anatomy of a device driver.

- When we want to use OS-9 for a real-time control application, it is very likely that we have to add one or more device drivers for a **special device**. Hopefully we will not need a special file manager.
- The nature of the special device has to be studied, in order to decide which file manager (rbf, sbf, scf) is best suited.
- The **interrupt service routine** is physically part of the device driver, but **logically** it is an **independent entity**.





## Anatomy of a device driver.

- If the driver will receive only **kill** or **wake-up signals**, no signal intercept routine is needed.
- If one wants to send other messages, such as "keyboard abort" or *menu choices*, an **intercept** routine is needed.
- It is extremely important to identify the **critical sections**, not only of the driver(s), but of the entire application.





## Anatomy of a device driver.

- *Disabling interrupts* may be *too dangerous* if the application is highly *time-critical*. Other mechanisms (TAS, semafhores) must then be used, which disable interrupts for very short periods only, or not at all.
- The **Getstat** and **Putstat** entry points of the driver merit attention. They allow to implement **special, device dependent functions**), which can be enterely **user-defined**.





## Anatomy of a device driver.

- The file manager will pass the function code and the register stack to the driver when an I$SetStt or I$GetStt system call is executed.
- Note that the **C-functions Getstat and Setstat are limited**: they perform I$GetStt and I$SetStt for a few function codes only.
- Microware's C library contains the flexible functions:

```
    OS9 ("SYSTEM CALL NAME", "ADDRESS OF REGISTER
ARRAY"
FOR EXAMPLE:
    OS9 (I_GETSTT, &REG)
```

## Anatomy of a SCFdevice driver.

- A **SCFdevice driver** in OS-9 follows the **client-server** model (In fact two: one for read, one for write).
- Remember the six entry points of a device driver. **Init** and **Term** need no particular comments.
- **GetStat** and **Putstat** open many possibilities, particularly for special purpose drivers.
- **Read** and the **interrupt service routine** form the **client-server**. Two **asynchronous processes** inside the driver.



Introduction to real time operating systems

## Anatomy of a SCFdevice driver.

- **Read** is the **client**; **interrupt service routine is the server**. (Similarly *Write is server, interrupt service* routine *the client*).
- Read and Write use **circular buffers**, separate for Read and Write.
- **Synchronization and mutual exclusion** obtained with the OS-9 mechanisms:
  - when stuck, **go to sleep** (and not busy-wait)
  - **wake-up** of suspended process provoked **by interrupt service routine**, by **sending a signal** to suspended process
  - signal received by **Intercept routine**.

## 7.4 SCF DEVICE DRIVER STORAGE DEFINITIONS

An SCF-type device driver module contains a package of subroutines that perform raw I/O transfers to or from a specific hardware controller. These modules are reentrant, so one copy of the module can simultaneously run several different devices that use identical I/O controllers. For each "incarnation" of the driver, IOMAN will allocate a static storage area for that device driver. IOMAN determines that a new incarnation of the device driver is needed when an attach occurs for a device with a different port address. The size of the storage area is given in the device driver module header. Some of this storage area is required by IOMAN and SCF, the device driver is free to use the remainder for variables and buffers. This static storage is defined in OS9 IODEFS and OS9 SCFDEFS as:

```
OFFSET              ORG 0

 $0     V.PAGE   RMB 1    port extended address
 $1     V.PORT   RMB 2    device base address
 $3     V.LPRC   RMB 1    last active process id
 $4     V.BUSY   RMB 1    active process id (0 = not busy)
 $5     V.WAKE   RMB 1    process id to reawaken
        V.USER   EQU .    end of OS9 definitions

 $6     V.TYPE   RMB 1    device type or parity
 $7     V.LINE   RMB 1    lines left until end of page
 $8     V.PAUS   RMB 1    pause request (0 = no pause)
 $9     V.DEV2   RMB 2    attached device static storage
 $B     V.INTR   RMB 1    interrupt character
 $C     V.QUIT   RMB 1    quit character
 $D     V.PCHR   RMB 1    pause character
 $E     V.ERR    RMB 1    error accumulator
 $F     V.XON    RMB 1    X-on character
$10     V.XOFF   RMB 1    X-off character
$11     V.RSV    RMB 12   reserved
$1D     V.SCF    EQU .    end of scf definitions
```



**NAME:** INIT

**INPUT:** (Y) = address of device descriptor module
(U) = address of device static storage

**OUTPUT:** NONE

**ERROR OUTPUT:** (CC) = C BIT SET
(B) = ERROR CODE

**FUNCTION:** INITIALIZE DEVICE AND ITS STATIC STORAGE

Usually this routine has three basic operations to do:

1. Initialize the device static storage.

2. Place the driver IRQ service routine on the IRQ polling list by using the OS9 F$IRQ service request.

3. Initialize the device control registers (enable interrupts if necessary).

**NOTE:** Prior to being called, the device static storage will be cleared (set to zero) except for V.PAGE and V.PORT which will contain the 24 bit device address. There is no need to initialize the portion of static storage used by IOMAN and SCF.

**NAME:** READ

**INPUT:** (Y) = address of path descriptor
(U) = address of device static storage

**OUTPUT:** (A) = character read

**ERROR OUTPUT:** (CC) = C bit set
(B) = error code

**FUNCTION:** GET NEXT CHARACTER

This routine should get the next character from the input buffer. If there is no data ready, this routine should copy its process ID from V.BUSY into V.WAKE and then use the F$SLEEP service request to put itself to sleep indefinately.

Later when data is received, the IRQ service routine should put the data in the buffer, then check V.WAKE to see if any process is waiting for the device to complete I/O. If so, the IRQ service routine should send a wakeup signal to it.

NOTE: Data buffers for queueing data between the main driver and the IRQ service routine are NOT automatically allocated. If any are used, they are defined in the device's static storage area.

OS-9 SYSTEM PROGRAMMER'S MANUAL
Sequential Character File Manager

**NAME:** WRITE

**INPUT:** (A) = char to write
(Y) = address of the path descriptor
(U) = address of device static storage

**OUTPUT:** NONE

**ERROR OUTPUT:** (CC) = C bit set
(B) = error code

**FUNCTION:** OUTPUT A CHARACTER

This routine places a data byte into an output buffer and enables the device output interrupt. If the data buffer is already full, this routine should copy its process ID from V.BUSY into V.WAKE and then put itself to sleep.

Later when the IRQ service routine transmits a character and makes room for more data in the buffer, it checks V.WAKE to see if there is a process waiting for the device to complete I/O. If there is, it sends a wake up signal to that process.

Note: This routine must ensure that the IRQ service routine will start up when data is placed into the buffer. After an interrupt is generated the IRQ service routine will continue to transmit data until the data buffer is empty, and then it will disable the device's "ready to transmit" interrupts.

Note: Data buffers used for queueing data between the main driver and the IRQ routine are NOT automatically allocated. If any are used, they should be defined in the device's static storage.

**NAME:** TERM

**INPUT:** (U) = ptr to device static storage

**OUTPUT:** NONE

**ERROR OUTPUT:** (CC) = C bit set
(B) = Appropriate error code

**FUNCTION:** TERMINATE DEVICE

This routine is called when a device is no longer in use, defined as when its use count in the device table becomes zero. In Level One systems, the termination routine is not called until the link count of the driver, descriptor, or file manager also reaches zero, and the module is being removed from the system memory directory. It must perform the following:

1. Wait until the output buffer has been emptied (by the IRQ service routine).

2. Disable device interrupts.

3. Remove device from the IRQ polling list.

**NOTE:** LI - Modules contained in the BOOT file will NOT be terminated.
LII - Any I/O devices that are not being used will be terminated.

**NAME:** GETSTA
SETSTA

**INPUT:** (A) = function code
(Y) = address of path descriptor
(U) = address of device static storage

**OUTPUT:** Depends upon function code

**FUNCTION:** GET/SET DEVICE STATUS

This routine is a wild card call used to get (set) the device parameters specified in the I$GETSTT and I$SETSTT service requests. Most SCF-type requests are handled by IOMAN or SCF. Any codes not defined by them will be passed to the device driver.

In writing getstat/setstat codes, it may be necessary to examine or change the register stack which contains the values of the 6809 registers at the time the OS9 service request was issued. The address of the register packet may be found in PD.RGS, which is located in the path descriptor. Note that Y is a pointer to the path descriptor and PD.Rgs is the offset in the path descriptor. The following offsets may be used to access any particular value in the register stack:

| OFFSET | MNEMONIC | | | MPU REGISTER |
|---|---|---|---|---|
| $0 | R$CC | RMB | 1 | condition code register |
| $1 | R$D | EQU | . | D register |
| $1 | R$A | RMB | 1 | A register |
| $2 | R$B | RMB | 1 | B register |
| $3 | R$DP | RMB | 1 | DP register |
| $4 | R$X | RMB | 2 | X register |
| $6 | R$Y | RMB | 2 | Y register |
| $8 | R$U | RMB | 2 | U register |
| $A | R$PC | RMB | 2 | program counter |

Sample access:

```
ldx PD.RGS,y
ldd R$Y,x
```

the IRQ polling sequence via an F$IRQ system call.

```
ldd V.Port,u get address to poll
leax IRQPOLL,pcr point to IRQ packet
leay IRQSERVC,pcr point to IRQ service routine
OS9 F$IRQ add dev to poll sequence
bcs Error abnormal exit if error
```

Step 2: Whenever a driver program must wait for the hardware, it should call a sleep routine. The sleep routine will copy V.Busy to V.Wake, then it will go to sleep for some period of time.

Step 3: When the driver program "awakens", it will check whether it awakened because of an interrupt or a signal sent from some other process. The usual way to accomplish the check is with the V.Wake storage byte. The V.Busy byte is maintained by the file manager to be the process ID of the process using the driver. When V.Busy is copied into V.Wake, then V.Wake becomes a flag byte and an information byte. A non-zero Wake byte indicates there is a process awaiting an interrupt. The value in the Wake byte indicates what process should be awakened by the sending of a wakeup signal. The following code will indicate a technique to accomplish this:

```
        lda V.Busy,u get proc ID
        sta V.Wake,u arrange for wakeup
        andcc #^IntMasks clear the way for interrupts
Sleep50 ldx #0 or any tick time desired.
        OS9 F$Sleep await an IRQ
        ldx D.Proc get process desc ptr (if signal test)
        ldb P$Signal,x is signal present? (if signal test)
        bne SigTest bra if so (if signal test)
        tst V.Wake,u IRQ occur?
        bne Sleep50 bra if not
```

Note that the code labelled "if signal test" is only necessary if the driver wishes to return to the caller if a signal is sent without waiting for the device to finish. Also note that IRQs (and FIRQs) must be masked between the time a command is given to the device and the moving of V.Busy to V.Wake. If they are not masked, it is possible for the device IRQ to occur and the IRQSERVC routine to become confused as to sending a wakeup signal or not.

Step 4: When the device issues an interrupt, the routine address given in the F$IRQ will be called. This routine is called as if it were a portion of the interrupt handler in the system. The interrupts are masked, the routine should be as short as possible, and the routine should return to the caller via RTS, since the system poller has called it via JSR and will do the RTI when done. The IRQSERVC routine may want to verify that an interrupt has occurred for the device. It will need to clear the interrupt and retrieve any data in the device. Then the V.Wake byte is used to communicate back to the main driver routine. If V.Wake is non-zero, it should be cleared (indicating a true device interrupt), and its contents used as the process ID for and F$Send system call sending a wakeup signal to the process. Some sample code follows:

```
ldx V.Port,u get device address
tst ???? is it real interrupt from this device?
bne IRQSVC90 bra to error if not
lda Data,x get data from device
sta 0,y store data in buffer (simplified example)
lda V.Wake,u get process ID
beq IRQSVC80 bra if none
clr V.Wake,u clear it as flag to main routine
ldb #S$Wake get wakeup signal
OS9 F$Send send signal to driver
IRQSVC80 clrb clear the carry bit (this indicates all is well)
rts
```

**NAME:** IRQ SERVICE ROUTINE

**FUNCTION:** SERVICE DEVICE INTERRUPTS

Although this routine is not included in the device drivers branch table and not called directly from SCF, it is an important routine in device drivers. The main things that it does are:

1. Service the device interrupts (receive data from device or send data to it). This routine should put its data into and get its data from buffers which are defined in the device static storage.

2. Wake up any process waiting for I/O to complete by checking to see if there is a process ID in V.WAKE (non-zero) and if so send a wakeup signal to that process.

3. If the device is ready to send more data and the output buffer is empty, disable the device's "ready to transmit" interrupts.

4. If a pause character is received, set V.PAUS in the attached device static storage to a non-zero value. The address of the attached device static storage is in V.DEV2.

5. If a keyboard abort or interrupt character is received, signal the process in V.LPRC (last known process) if any.

When the IRQ service routine finishes servicing an interrupt, it must clear the carry and exit with an RTS instruction.