"MARY RUNSYS - A Portable Operating System for Program Development on Small Computers"

DATE 30.8.76

NO. OF PAGES 296 + 362 p. of App.

PROJECT Operating Systems and Programming Languages

AUTHOR Reidar Conradi

This report contains a thesis submitted for the degree of lic.techn. at the above university

ABSTRACT: It describes a medium-sized, multi-user operating system for program development, to run on PDP-11 like mini-computers. It is written entirely in MARY (18K source lines; 45 K instructions, whereof 7 K resident), and is currently implemented for the Norwegian NORD-SM machines. A prototype runs on a 48 K SM4-machine. The system is Burroughs-inspired: process implementation is based on cactus stacks, i.e., an OS-call behaves like an ordinary RECURS proc call. The synchronization primitives are rather general. The memory management is segment-based, and offers a virtual memory facility for code (being reentrant) as well as data. A hierarchical file system, displaying first-class synchronization properties, is available. A MARY-like JC-language is implemented, giving interactive users easy access to editor-, file system-, MARY-MARY compiler- and loader-facilities. Batch jobs may also be initiated.

3 KEYWORDS
MARY RUNSYS
Operating systems
Virtual memory
To my friends who made the prospects of a post-doctoral life worthwhile.
Preface

This thesis describes a portable, medium-sized operating system, MARY RUNSYS. The work has been carried out in the years 1973-1976, when employed as a Research Scientist at the Computing Centre of the University of Trondheim, Norway (RUNIT), and the last year as an Assistant professor at the Division of Computing Science at the same university.

MARY is an ALGOL60-like System Implementation Language, developed by RUNIT. The original intention behind RUNSYS was (as the name may suggest) to provide a simple olay-system to run a segmented, bootstrapped MARY-compiler on some Norwegian 16 bits mini-computers. However, the ambitions have grown considerably by the years, from a modest stand-alone system to a general multiprogramming system, offering common OS-functions such as a file system, an editor, a loader and a job control interpreter - along with the above compiler. The system a multi-user one, utilizing reentrant olayable code (like the Burroughs machines). It can support both demand and batch jobs.

Total size: 45 K of instructions (7 K resident), not counting the compiler. 18 K lines of MARY source code. 2 manyears have totally been spent on its implementation.

RUNSYS is primarily intended to support software development on machines, that resemble the Norwegian NORD and SM minis (PDP11-derivatives). The system has more or-less been running for 2 years, and has constantly been modified and expanded. It exists, for the time being, as a prototype on a 48 K SM4-machine.

The system is designed to combine well-structuredness, orthogonality and portability (it's written entirely in MARY) with efficiency and user-orientation. It will, hopefully, display practical (and commercial) qualities in the future, in addition to the present academic ones.
I will now survey the following chapters, thereby focusing the reader's attention on selected RUNSYS highlights.

Chapter 1 contains some introductory remarks about the current state of systems programming, with a particular reference to operating systems on small machines.

Chapter 2 gives the historical background for and the basic principles behind MARY RUNSYS.

Chapter 3 describes the implementation of processes (tasks) in RUNSYS. Each task possesses, in true Burroughs-fashion, a local stack on which shared, recursive (and hence reentrant) procedures may allocate their data incarnations. A hierarchical task structure is thus mapped by a cactus stack. Two synchronization procedures (ACQUIRE and RELEASE), implementing generalized semaphores and resources; and the facilities for task management (FORKing, READYQ-rotation) are also described.

Chapter 4 outlines the organization of primary storage: Memory is divided into variably-sized segments, being SAVE (resident) or OLAY (able). OLAY segments, containing code (i.e. olayable, reentrant MARY routines) or data (usually editor texts), are always accessed via SAVE descriptors. The implementation of the resulting virtual memory system, along with some rather interesting performance data and a working-set model, is scrutinized.

Chapter 5 summarizes low-level task management (interrupt-handling), and the treatment of peripheral devices. The synchronization of full-duplex TTYs is worth attention. Exchange of formatted I/O from normal tasks is also explained.

Chapter 6 deals with the file system in RUNSYS. This is hierarchically organized, supporting named subfiles within (parent) files—and so on recursively. It offers excellent synchronization properties: A subfile, e.g. containing a symbolic program, can be selectively locked without blocking the parent file.
Chapter 7 displays the structure of the implemented JC-language, being a subset of MARY. Ex. F:~G ed H marycomp R load OLAYFILE;.
Common usage of the editor (offering context-sensitive as well as fixed-column operations), the loader and the MARY-MARY compiler is also demonstrated.

Chapter 8 surveys the implementation of batch tasks (the backlog), via the START-command. RUNSYS initialization (recommended reading), -configuration and -booting are also treated.

Chapter 9 tells about the testing of RUNSYS (e.g. how reproducible test runs were obtained), and the experience gained during the testing phase.

Chapter 10 comments on expected RUNSYS security. Particularly, it elaborates the inherent software protection granted by the MARY language, which enables user tasks to call user- as well as system routines in a safe way. Thus, hardware protect systems and monitor modes are abolished/superfluous.

Chapter 11 contains a status report on what's currently implemented the sizes of system modules, etc.

Chapter 12 examines the tools that were used to construct the system. In particular, the MARY language is critically reviewed. An error statistics from the implementation of RUNSYS is also commented upon.

Chapter 13 reveals some thoughts on the relation between hardware design and operating systems, with a specific reference to RUNSYS.

Chapter 14 discusses the portability of RUNSYS, which is found to be promising for machines, that among other things are stack-oriented and don't have hardware relocation or protection mechanisms.
Chapter 15 tries to evaluate the entire RUNSYS; e.g. some comments on Hoare-monitors versus Burroughs-like cactus-stacks are supplied. Some areas for future work is outlined and a few comparative remarks with other systems is given.

Chapter 16 concludes the work that has been done.

A survey of the MARY project, the MARY language and two examples of test runs are supplied in Appendices A-C.

The entire source code of MARY RUNSYS can be found in a separate Appendix D. Competent readers ought to have a closer look at some selected parts. I will particularly recommend modules REIDAR. RSDECS, REIDAR.TREATINTRPT, REIDAR.SPACESYNCH1, REIDAR.TASKGEN, REIDAR.IODRIV1, EDIT.DELETE and ANN.NEWFILESYS.

I hope that you will enjoy reading this thesis. It has been written in an enthusiastic, conversating style, although I hope I have not any place drawn impermissible conclusions about the capabilities of the system.

There is no danger of mathematical overkill by reading it. A possible threat to its readability for the uninitiated student may be some paragraphs of MARY code, describing selected algorithms and data structures. The reasons for relying on MARY for such purposes are:

1) MARY is a convenient notation language for me to use. Indeed, no assembler or flowcharts will be found.

2) I'm interested in defining automatic, user-oriented facilities to enable normal programs to perform OS-operations (such as synchronization) in a safe way. An extensible programming language (like MARY) lends itself as a natural tool. Yet, as we shall see later, we might sometimes have appreciated an even more high-level language.
Some abbreviations also occur, which may be unfamiliar: booted (bootstrapped), misc. (miscellaneous), incl. (including/included), parms (parameters), deCs (declarations), specs (specifications), fork (create a new process), minis (mini-computers), U1108 (UNIVAC 1108), RECURS (recursive, stack-generator-driven).

Furthermore, the term "disk" may later on be interchanged with "drum", when referring to swap media. 1"k" may have been inconsistently used to denote both "1000" and "1024", and "core" may sloppily have been used synonymously with "primary storage".

A draft of this thesis was written in Nov. 1974. A lot of RUNSYS code has been put into operation since then. Besides, I have given implementation-work first priority compared to thesis writing (a RUNSYS feature shall work, before it is "described"). This decision may, however, be strategically unfavourable from an academic point of view. Although the majority of the more fundamental modifications have been taken into account, there may be minor mismatches between hereby described and currently implemented RUNSYS features.

However, in order to get this thesis finished before deadline (and to reduce the suicide rate of Norwegian secretaries), the degree of perfectionism/polishing must come to an end. The demand of a professional text-documentation system is recognized, but there is none available.

Then it only remains to wish potential readers good luck with the wading-through of this piece of text. Any comments will be sincerely appreciated by the author.
Acknowledgements

I wish to thank previous and present members of the MARY group at RUNIT: Mark Rain, Per Holager, Geir Green and Ole Solberg, for stimulating discussions and a genuine interest in my work. M. Rain and P. Holager deserve a special honour for implementing the pilot version of the MARY compiler, thus providing an adequate tool for systems programming.

Kristen Rekdal and Tor Stålhane have read drafts of this thesis, and their pinpointing comments helped me clarify many obscure points.

I also wish to thank RUNIT and the local Division of Computing Science for a very generous supply of computing power.

Finally, I'm indebted to my secretaries Eli Hagen (in particular), Judith R. Johansen, Kari E. Marsdal and Ingrid Bache for excellent typing and layout work.

Trondheim,
15. August 1976,

Reidar Conradi
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>1</td>
</tr>
<tr>
<td>Table of contents</td>
<td>7</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>17</td>
</tr>
<tr>
<td>1.1 A plea for flexible, user-oriented operating systems</td>
<td>17</td>
</tr>
<tr>
<td>1.2 On implementation languages</td>
<td>19</td>
</tr>
<tr>
<td>1.3 On MARY</td>
<td>21</td>
</tr>
<tr>
<td>2. Basic principles behind MARY RUNSYS</td>
<td>23</td>
</tr>
<tr>
<td>2.1 Why MARY RUNSYS? Historical background</td>
<td>23</td>
</tr>
<tr>
<td>2.2 Design criteria behind MARY RUNSYS</td>
<td>24</td>
</tr>
<tr>
<td>2.3 On implementation of processes</td>
<td>27</td>
</tr>
<tr>
<td>3. Basic process of TASK implementation</td>
<td>31</td>
</tr>
<tr>
<td>3.1 Layout of TASK-objects and data stacks</td>
<td>31</td>
</tr>
<tr>
<td>3.1.1 TASK-objects, main attributes</td>
<td>31</td>
</tr>
<tr>
<td>3.1.2 Register saveareas (NORD-SM)</td>
<td>32</td>
</tr>
<tr>
<td>3.1.3 Data stack administration (NORD-SM)</td>
<td>33</td>
</tr>
<tr>
<td>3.2 Life cycle of a task</td>
<td>34</td>
</tr>
<tr>
<td>3.2.1 Task queuing and task priorities</td>
<td>34</td>
</tr>
<tr>
<td>3.2.2 Software interrupts (WAIT-instructions)</td>
<td>35</td>
</tr>
<tr>
<td>3.2.3 Task states</td>
<td>36</td>
</tr>
<tr>
<td>3.3 On synchronization</td>
<td>37</td>
</tr>
<tr>
<td>3.3.1 Implementation of synchronization primitives</td>
<td>39</td>
</tr>
<tr>
<td>3.3.2 Semaphore and resource modes in MARY</td>
<td>40</td>
</tr>
<tr>
<td>3.3.3 The basic synchronization procedures, ACQUIRE and RELEASE</td>
<td>42</td>
</tr>
<tr>
<td>3.3.4 Comments on the implementation of synchronization primitives</td>
<td>48</td>
</tr>
<tr>
<td>3.3.4.1 Generality and simplicity</td>
<td>48</td>
</tr>
<tr>
<td>3.3.4.2 Security, technical aspects</td>
<td>48</td>
</tr>
<tr>
<td>3.3.4.3 Efficiency</td>
<td>49</td>
</tr>
<tr>
<td>3.3.4.4 Code residency and reentrancy</td>
<td>49</td>
</tr>
<tr>
<td>3.3.4.5 Derived synchronization primitives</td>
<td>49</td>
</tr>
</tbody>
</table>
3.3.4.6 Variants of the primitives. 51
3.3.4.7 Higher-level primitives. 51
3.3.4.8 Macros for initializing semaphores and resources. 52
3.3.4.9 Macros for declaring critical regions á la Brinch Hansen. 53
3.3.4.10 Basic, low-level synchronization. 55
3.4 Deadlock and security aspects of synchronization. 57
  3.4.1 Language protection. 58
  3.4.2 Task tree protection and related subjects. 58
  3.4.3 Consistency- and deadlock testing. 60
  3.4.3.1 Data structures revisited. 60
  3.4.3.2 CREATOR consistency. 62
  3.4.3.3 Deadlock problems of synchronization. 62
3.5 Software interrupts, complete presentation. 64
  3.5.1 The REQUESTHANDLER. 65
3.6 Task creation and termination. 67
  3.6.1 Task creation. 67
    3.6.1.1 Critical block trap. 68
  3.6.2 Task termination. 70
    3.6.2.1 Causing task termination. 70
    3.6.2.2 The TERMINATE routine. 71

4. Basic allocation of primary storage (memory). 74
  4.1 General remarks. 74
  4.2 On storage allocation in RUNSYS. 75
  4.3 General storage allocation mechanisms offered by RUNSYS. 76
    4.3.1 Space generators needed to execute MARY programs. 76
    4.3.2 System-controlled space generators. 76
  4.4 Basic space generators in RUNSYS, the MEMLINK-formalism. 77
    4.4.1 MEMLINK organization. 78
    4.4.2 Allocation and deallocation of MEMLINKs. 79
    4.4.3 How to use the MEMLINK facility. 79
4.5 Implementation of OLAY-code.
  4.5.1 The virtual code problem. 81
  4.5.2 Practical implementation of OLAY-code
      (initial version). 81
    4.5.2.1 General set-up. 81
    4.5.2.2 Implementation details. 83

4.6 Implementation of OLAY-data.
  4.6.1 The virtual data problem. 86
  4.6.2 Practical implementation of OLAY-data. 87
    4.6.2.1 General set-up. 87
    4.6.2.2 Data structures. 88
    4.6.2.3 Utility macros. Usage. 90

4.7 Memory shortage and -deadlocks. Virtual memory performance.
  4.7.1 On memory shortage and memory deadlocks. 92
  4.7.2 General comments on factors, that influence the performance
      of virtual memory systems. 93
  4.7.3 Empirical data on segment sizes. 94
    4.7.3.1 Distribution of SAVE segment sizes. 94
    4.7.3.2 Distribution of OLAY-code segment sizes. 96
    4.7.3.3 External fragmentation. 101
  4.7.4 Garbage collect strategies for OLAY segments. 101
    4.7.4.1 Replacement policies for OLAY-code. 101
    4.7.4.2 Replacement policies for OLAY-data. 103
  4.7.5 Initial performance of the segmentation system. - What did we
      learn? 104
    4.7.5.1 The starting point. 104
    4.7.5.2 Putting constant SAVE-data into OLAY-code. 106
    4.7.5.3 A hierarchical space allocation strategy. The balancing point.
      106
    4.7.5.4 Reduced buffering overhead. 108
    4.7.5.5 Lost disk revolutions. Putting the segment size in the descriptor.
      108
    4.7.5.6 Multiprogrammed space allocation and disk IO. 109
    4.7.5.7 Reduction of allocation-times for new segments. 110
4.7.5.8 Reduction of memory deadlocks, caused by SAVE segments. 110
4.7.6 Performance data on the segmentation system. 111
4.7.7 A mathematical model of the VM-system. 114
4.7.7.1 A simple working-set model. 114
4.7.7.2 Some reflections on system parameters. 118
4.8 Concluding remarks about the virtual memory system in RUNSYS. 120
4.8.1 On system performance. 120
4.8.1.1 Current status. 120
4.8.1.2 Future tuning work. 120
4.8.2 Convenience of usage. 122
4.8.2.1 Ideas on OLAY-data in MARY. 122
4.8.2.2 On uniformity of data accesses. 123
4.8.3 A few comments on paging versus segmentation. 124
4.8.3.1 On run-time efficiency. 125
4.8.3.2 On space utilization. 125
4.8.3.3 Multiprogramming qualities. 125
4.9 A fast free-list system for very small data areas. 126
4.10 Aspects of a general heap implementation for operating systems. 127

5.1 Active system tasks in RUNSYS after system generation. 132
5.2 TASK-mode for NORD1 and SM3/4. 133
5.3 Interrupt-handling. 136
5.3.1 Basic task structures on NORD1 and SM3/4 with a 16 level interrupt system. 136
5.3.2 Basic task structures on SM3/4 with a single level interrupt system. 139
5.4 The dispatcher, TASKCHANGER (NORD-SM variant). 140
5.5 The CLOCKINTERUPT-task. 141
5.6 Basic IO-handling (NORD-SM). DEVICE-objects and IO-drivers. 141
5.7 Basic exchange of IO-messages, IOREQUESTs. 144
5.8 TTY-driver interface. 146
  5.8.1 TTY-synchronization. 146
  5.8.2 Administration of IO-buffers and IOREQUESTs from TTY-drivers. 147
  5.8.3 INPUT-driver actions. 148
  5.8.4 OUTPUT-driver actions. 149
5.9 Drivers for the line printer, card reader, paper tape punch and paper tape reader. 149
5.10 The disk driver. 150
5.11 The IFILE-mode. IO-interface routines for formatted IO, PUTIMAGE AND GETIMAGE. 150
  5.11.1 The IFILE-mode. 150
  5.11.2 PUTIMAGE and GETIMAGE routines. 151
  5.11.3 Standard input and output IFILEs. 152
  5.11.4 Interface with mass storage text-FILEs. 154

6. The MARY RUNSYS file system, FILESYS. 156
  6.1 Features of a basic file system. 156
  6.2 Design criteria for the basic file system in MARY RUNSYS. 157
    6.2.1 Recursive files. Subfiles. 157
    6.2.2 File privacy and other features. 159
  6.3 Implementation of FILESYS. 159
    6.3.1 Organization of mass storage. 159
    6.3.1.1 Contiguous static files, alternative A. 159
    6.3.1.2 Linked (scattered) dynamic files, alternative B. 160
    6.3.1.3 Some remarks on alternatives A and B. 162
    6.3.2 File structures in memory. The FILE-mode. 163
    6.3.3 File types. 166
    6.3.4 File names. 166
  6.4 User interface routines. 167
    6.4.1 Technical security when using files. 167
    6.4.2 Available file macros. 167
  6.5 File synchronization. Prevention of deadlocks. 171
7. MARY_RUNSYS JOB_CONTROL LANGUAGE. Local JOB-INTERPRETER-tasks.

7.1 On JCLs. The MARY_RUNSYS JCL. 174
7.2 The JOBINTERPRETER. 175
7.3 Miscellaneous JC-functions and related subjects. 176
  7.3.1 The VADD-facility. 176
  7.3.2 The editor (ED). 178
  7.3.2.1 Editor functions. 178
  7.3.2.2 Editor implementation. 180
  7.3.2.3 Format of text (i.e. SYMBOLIC)-files. 181
7.4 The MARY-MARY compiler. 182
7.5 The MARY loader. 183

8. Miscellaneous facilities. 185

8.1 The MSECOND-task. 185
8.2 Job scheduling. General resource allocation. 185
  8.2.1 Resource allocation in RUNSYS. 185
  8.2.2 Batch and demand tasks. The IDLEDEVICE-HANDLER-task. 186
  8.2.3 Backlog organization. 188
  8.2.4 The START-command. 188
8.3 Initialization and booting of RUNSYS. 189
  8.3.1 RUNSYS loading. 189
  8.3.2 Initialization actions during system generation. 190
  8.3.3 RUNSYS boot. 192

9. System testing. 193

9.1 Fighting irreproducibility. 193
9.2 Debugging tools. 194
  9.2.1 IO-routines. 194
  9.2.2 Dump and trace facilities. 197
9.3 MINIRUNSYS. 199
9.4 Testing of RUNSYS. 200
  9.4.1 Compilation of RUNSYS. 200
  9.4.2 Testing of RUNSYS initialization. 200
  9.4.3 Testing of simple user- and system tasks. 201
  9.4.4 Testing of OLAY-code. 201
  9.4.5 Testing of general IO-drivers and associated interface. 202
9.4.6 Testing of JOBINTERPRETER functions.
9.4.7 Testing of FILESYS functions.
9.4.8 Testing of EDITOR-, loader- and misc. functions.
9.4.9 Testing of multiple TTYs.
9.4.10 Testing of clock functions.
9.4.11 Testing of interrupt-driven I/O.
9.4.12 Normal, daily production testing.

9.5 General comments on the testing.
9.5.1 Time spent on testruns.
9.5.2 On run-time errors.
9.5.3 Experiences with operating system testing.

10. RUNSYS security. User and system protection.
10.1 Treatment of hardware errors.
10.2 Security features offered by the MARY language.
10.3 Software errors, treatment and prevention.
10.3.1 Errors in user programs.
10.3.2 Errors in system programs.
10.3.3 "External" (deadlock-like) error conditions.

11. Current status of RUNSYS.
11.1 Non-implemented features.
11.1.1 Described, but not yet implemented features.
11.1.2 Planned, but not yet implemented features.
11.2 Some quantitative measures.
11.2.1 Size of system modules.
11.2.2 Selected system characteristics.
11.3 Examples of test runs.

12. An evaluation of the tools used.
12.1 General impression of the available tools.
12.1.1 Programming productivity.
12.1.2 Error-proneness.
12.1.2.1 Error statistics.
12.1.2.2 Comments on the errors committed.
12.2 Evaluation of the MARY language.
  12.2.1 Machine-nearness. 221
  12.2.2 Run-time efficiency. 222
  12.2.2.1 Data space efficiency. 223
  12.2.2.2 Code efficiency. 223
  12.2.3 Expressive power. 226
  12.2.3.1 MARY's data types. 226
  12.2.3.2 Macros in MARY. 226
  12.2.3.3 Areas in MARY. 227
  12.2.3.4 UNSAFE MARY. 227
  12.2.4 Structured programming in MARY. 228
  12.2.5 Improvement proposals to MARY. 229
  12.2.5.1 Missing facilities in MARY. 230
  12.2.5.2 Design flaws in MARY. 230
  12.2.5.3 How to structure data and algo-
      rithms in a good SIL (System
      Implementation Language)? 232
  12.2.5.4 Concurrent execution of program
      statements. 233

13. Some thoughts on hardware design and operating systems. 235
  13.1 A plea for adequate interrupt-handling mechanisms. 235
  13.2 Addressing structures. 235
  13.3 Monitor mode and memory protection. 237
  13.4 Some aspects of multi-processor systems. 238

14. Portability of RUNSYS. 239
  14.1 Portability in general. 239
  14.2 Sources of nonportability. 239
    14.2.1 Low-level machine peculiarities. 239
    14.2.2 Machine-dependencies, definable in the
        implementation language (MARY). 239
    14.2.3 Machine-dependencies, not expressable in
        the programming language. 240
      14.2.3.1 Different addressing structures. 240
      14.2.3.2 One or several CPUs. 241
      14.2.3.3 Non-standard I0-equipment. 242
    14.2.4 Design goals. 242
    14.2.5 To sum up. 242
    14.2.6 A small case example. 243
15. Evaluating RUNSYS.

15.1 On the implementation of processes.
   15.1.1 General remarks.
   15.1.2 On OS-interface, monitors versus cactus stacks.

15.2 Multi-user qualities.
15.3 Efficiency.
15.4 Security.
15.5 Design flaws, previous and present cnems.
15.6 General usefulness of the system.
   15.6.1 Application areas.
   15.6.2 A MARY machine?
   15.6.3 Possible target machines.

15.7 A few comparative remarks about other systems.

16. Conclusion.

16.1 Lessons learned.
16.2 Future work.
16.3 Aftermath.

Reference list.

Appendix A. The MARY project.

Appendix B. A very quick survey of the MARY language.

B.1 Language structure.
   B.1.1 Primaries.
   B.1.2 Serial clauses.
   B.1.3 IF-, CASE- and DO-clauses.
   B.1.4 Expression language.

B.2 Operators and expressions.
   B.2.1 The store-operator, '='.
   B.2.2 The BOOLean AND- and OR-operators.
   B.2.3 Variants of the store-operator.
   B.2.4 The GO-operator.
   B.2.5 Shift operators.

B.3 Declarations.
   B.3.1 Cell declarations ("variable" declarations).
   B.3.2 Mode declarations.
   B.3.3 Routine declarations (a special kind of cell declarations).
B.3.4 Macro- and operator declarations. 275
B.3.5 Misc. constructs. 275
B.3.6 Coercions (type conversions). 276
B.3.7 UNSAFE MARY. 277
B.3.7.1 The PROTING coercion, AMODE ➔ PROTEAN. 277
B.3.7.2 AT, SYN and NOINIT in cell declarations. 277
B.3.7.3 LOCATE for allocation of code. 277
B.3.7.4 The EXECUTE builtin function. 278
B.3.7.5 Miscellaneous builtin functions. 278
B.3.7.6 Misc. details. 278

Appendix C. Test runs.
C.1 Using the MARY RUNSYS JC-language. 279
C.2 User-controlled producer-consumer tasks. 281

Appendix D. Source code of RUNSYS.
D.1 Table of contents of Appendix D. 288
(The Appendix itself is in a separate cover).
1. Introduction.

1.1. A plea for flexible, user-oriented operating systems.

Operating systems have been available for about 20 years. Their functions have been to improve utilization of shared computer resources, and to make it easier for programmers to write and execute programs.

By "operating system" is meant not only the basic monitor, but the whole class of surrounding systems software: compilers, file system and so on.

I am particularly interested in facilities offered for program development, and want to emphasize the following features of an operating system:

* Possibilities for interactive communication. As a part of this, a good context-sensitive editor, and a mechanism to allow dynamic construction and execution of new streams of job control-images.

* A "hierarchical" file system. The user must be able to define subfiles of existing files for storage of programs and data.

* A sufficient set of programming languages must be available. In addition to ALGOL, FORTRAN, an appropriate MOL (preferably not assembler!) and COBOL; we will need a general language being at least as powerful as PL/I, SIMULA or ALGOL68. Of course we regard MARY as a candidate here; it's also a MOL.

The reasons why computers rarely are equipped with this kind of human-oriented software, are many:
* It is costly: More manyears are needed on systems programming (particularly when this has to be done in assembler), and more hardware is probably needed to support it (additional primary storage, a disk) — compared to some minimal, traditional system.

* Some of the above functions (like interactiveness) have not until quite recently been recognized as vital or necessary by computer manufacturers and their customers.

* Recent high-level programming languages have not reached widespread acceptance. This has to do with human psychology, language weaknesses and compatibility problems. It's also expensive to develop and market a new language.

Hardware prices are steadily decreasing. Micro-processors will probably accelerate this development. The costs of human man-power are increasing, relatively and absolutely.

The current prices of mini- and micro-processor equipment should no longer prevent customers from bying sufficiently large computer configurations to enable adequate software support (like compilers for high-level languages). An example: The price of a 16 K core module is $20% of a programmer's annual salary.

Many small organizations find themselves in the situation that they have bought a mini-computer for Nkr. 100.000-500.000, but they can only program it in assembler or FORTRAN.

The CPU capacity of a mini (NORD, SM, PDP) is not significantly less than that of a mammoth machine (a factor, say, 3-8) but its price is only 1-5% of the latter. If we were able to construct a suitable software environment on the minis (at a reasonable price), a significant saving in price/performance is expected.
The software crisis on mini-computers has another aspect:

These machines are very often used in dedicated applications, like real-time control and communication networks. Relevant systems software for these applications will, for reasons explained later, usually require assembly programming. Thus, because of the very application area of the minis, assembler tends to be the dominant programming language.

Regrettably, not only the minis offer poor software environments to their users. Many large and medium size computers also do. Example: Neither IBM360 nor CDC3300 have appropriate mechanisms for storage of symbolic programs on files. They don't support interactive terminals in a flexible way either (but this has partly historic reasons). On the other hand, computers as the U1108 offers excellent support systems for software development.

1.2. On implementation languages.

I suspect that the main reason why operating systems don't cope with our needs, is the inadequate tools these systems are constructed with, i.e. the implementation language, usually assembler. In other words: The most difficult programs are constructed with the poorest of tools.

Traditionally, the productivity of systems programming in assembler is \( \frac{1}{5} \) of ALGOL/SIMULA-standards, and maintenance is even more laborious. Also, the work is duplicated for every machine in question because assembler code is not portable.

Systems software has until now rarely been written in high-level languages. Some exceptions: A PL/I-variant was used in MULTICS [Corbató, Vyssotsky, 1965]; various languages have been used for compiler-writing. This situation is about to turn. Nearly all big computer manufacturers are developing and using their private, semi-secret implementation languages (PLUS for UNIVAC, PL/S for IBM).
Let us indicate the possible gains of using high-level languages in systems programming by referring to Burroughs who launched their B5500 in the middle 60s. Burroughs has no assembler on their machines, due to a remarkable hardware for executing ALGOL60-programs. They are using two ALGOL60-dialects, called ESPOL and Burroughs' Extended Algol, as their implementation languages. The entire operating system for their B5500 required about 15 manyears, and is even today regarded as one of the very best user-oriented, time-sharing systems on the market. See [Burroughs, 1970] for ESPOL-documentation.

Then why haven't systems programmers been using high-level programming techniques in all these years? Simply because existing "high-level" programming languages were unsuitable for systems programming. The first successful attempt to bridge the gap between classic high-level languages and assembler was Wirth's PL360 [Wirth, 1968]. Since then much of the activity in the field of Programming Languages has concentrated upon Machine-Oriented and System Implementation Languages (MOLs and SILs).

Let us systematize the reasons why traditional high-level languages are insufficient for systems programming:

* They are not machine-oriented enough; e.g. for interrupt-handling, basic I/O and register operations.

* They don't utilize hardware resources in an efficient way; e.g. they generate excessive code and offer no mechanisms for packing data to the machine words.

* They lack expressive power. I.e. a lot of program and data structures needed in systems programming cannot be described; e.g. common blocks and general pointer variables.

I would like to stress the importance of powerful data structuring facilities. The reason why FORTRAN- and ALGOL-programmers have to resort to assembly programming, is
usually that they cannot express the relevant data operations
in FORTRAN or ALGOL, which don't support structured data,
ALGOL68-like union types and similar niceties.

We should distinguish between 4 classes of "implementation
languages":

1. MOLs, Machine-Oriented Languages. The remaining 3 classes
   are often subclasses of this class.
   Ex. Assembler, NPL [ND, 1973].

2. PL-languages, descendants of PL360. Using ALGOL60 or PL/I
   as host languages.
   PL-languages with PL/I as a host language are
   usually termed PL/M, PL/S, etc.

3. MOHLLs, Machine-Oriented High-level Languages.
   Ex. BLISS [Wulf et al., 1971], MARY [Rain et al., 1973].

4. SILs, System Implementation Languages.
   Ex. LIS [Ichbiah et al., 1973], ALPHARD [Wulf, 1974a].
   These have emerged in recent years, because what we need
   is not only machine-orientation, but a formalism for desc-
   ribing large and highly complex data- and program structure.
   We will naturally regard MARY as a SIL too.

1.3. On MARY.

MARY is a portable machine-oriented high-level programming
language, developed at RUNIT (The Computing Centre at the
University of Trondheim) in the years 1971-1976. A summary
of activities in the MARY project can be found in Appendix A.
Although a lot of MARY documentation is available elsewhere
[Conradi, Holager, 1975], a brief survey seems justified and
is supplied in Appendix B. This will enable advanced readers
to understand the forthcoming algorithms, to evaluate more independently the general RUNSYS design, and to participate in the concluding MARY discussions.

The reader is urged to study Appendix B, before proceeding with chapter 2.
2. Basic principles behind MARY RUNSYS.

2.1. Why MARY RUNSYS? Historical background.

While writing the MARY pilot compiler (in NUALGOL), we soon realized that we could not run the booted MARY-MARY compiler on a medium-size NORD-SM configuration; that is, 16-24 K words of primary storage and no mass storage. We would need some kind of overlaying of code/data, i.e. a disk/drum and some software backup to supervise program execution. In other words, a monitor that supported virtual memory.

The existing operating systems on the NORD-SMs (SINTRANI, KOS) offered no suitable software environments for olay-able programs. They were too much oriented towards process control applications, and facilities offered for space administration were inadequate. The task of designing an olay-system to suit MARY is particularly difficult, because of recursive procedures. We may change program "segment" not only on entry, but also on exit of a procedure. Further, we aim at a multi-user application, utilizing shared reentrant code. I.e. a classic tree-structure for program segmentation won't work.

Also, available hardware on these machines gives software people little help in implementing an olay-system. This is probably not true for the NORD10 with hardware paging, but assisting software was not released autumn 1974.

Furthermore, we didn't care too much for the idea of shuffling large amounts of code from disk to fixed locations ("partitions") in memory.

Luckily, Mark Rain, having spent 2 years at Burrcughes, had received many stimulating ideas on the design of segmentation systems and on operating systems in general.

Thus we decided to write a general, flexible monitor to support execution of OLAY-segmented MARY programs, based on Burroughs-principles. The design philosophy behind the resulting MARY RUNSYS is elaborated in section 2.2.
Mark Rain wrote drafts of the basic routines for process synchronization, process administration, space allocation and interrupt-handling - amounting to ~800 MARY source lines. He also prepared the pilot compiler for generation of OLAY routines. And, of course, he generously gave us ideas on how to do things and how not to.

2.2. Design criteria behind MARY RUNSYS.

When starting from scratch with no other contraints than to produce a small operating system to support compilation and execution of (segmented) MARY programs, we were tempted beyond human resistance.

First, we saw an opportunity to prove MARY's claimed qualities through practical experience. This self-fullfilling activity is dealt with elsewhere (sec. 12.2).

More important is, that because we had free hands, we intended to design and implement an operating system

* along the latest principles for structured programming (top-down design, modularization, parameterization), and

* according to the most recent strategies in the field of operating systems (orthogonal design, a general implementation of processes, a hierarchical file system). Cf. [Hoare,Perrott,197]

The following guidelines have been applied:

* The system should be used on relatively small computers on which primary storage and IO-channel capacity constitute the bottlenecks. We have not been afraid of spending CPU cycles, if this reduced the demand of primary storage.

* It should support an olay-system for code as well as data. Code/data may be specified as SAVE (memory-resident) or OLAYable.
* It should have a general and orthogonal design.

We don't think of generality in the sense that it should offer facilities for every thinkable demand, but rather that the primitive functions of the system should be generalized in order to minimize the number of such.

Our most scarce resource is manpower. If we only have to write one procedure for each independent function, we will save a lot of writing, testing and maintenance.

For instance, all RUNSYS processes have the same set-up; no distinction between user and system processes is made (except that I0-drivers are "triggered" in a special way). Likewise, only one set of synchronization primitives exists.

A user program/process is able to perform explicit synchronization and to create new processes (FORKing). Calls on monitor functions are implemented as ordinary calls on recursive procedures - and so on inside monitor procedures, when they require new monitor functions. See sec. 2.3 for further details.

* It should be flexible and parameterized; Easy to change and maintain; Easy to add new facilities.

Nobody really disagrees on these matters, but the degree of flexibility may be limited by efficiency reasons and our limited financial support.

* It should not have an accounting system.

* It should give protection against malicious or careless users. Users should not be able to crash other programs, including the basic operating system. This security is obtained by restricting MARY programmers to SAFE MARY (inherent software protection), by implementing a safe FORTRAN-variant for FORTRAN users, by performing complete parameter checks in all user-callable system routines, etc....
* No memory protect system has been assumed. Likewise for hardware relocation mechanisms, including hardware paging.

* Compatibility. The majority of existing, relocatable user program (written in assembler or FORTRAN on the NORD-SMs) should be executable under MARY RUNSYS. However, the above security aspects may kill this opportunity on a multi-user installation, and some interface problems (IO) will exist.

* The system should be as portable as possible. Only basic IO and interrupt-handling represent genuine machine-dependent code sections which must be rewritten for a new machine.

* A fruitful cooperation/interface with the MARY compiler should be established. E.g. all MARY code is supposed to be reentrant, so-called OLAY routines must be correctly generated by the compiler, etc.

* The system should be strongly oriented towards multiprogramming. Normal users will be operating from interactive terminals, doing editing of programs, file-system operations, compilations, linking/loading execution of user programs.

Programs requiring big hunks of statically allocated data (like common blocks) will receive bad service.

* Severe real-time requirements will probably be hard to meet, without tailoring the system to some degree.

* The file system should support hierarchic files (files inside files), and have very good multiprogramming properties.

* Normal formatted IO should be standardized:
  IO on a TTY, on standard read/print-files or on arbitrary text files should look identical from user programs.
2.3. On implementation of processes.

We have adopted the Burroughs' philosophy with one data stack per process. In principle, there is no distinction between user and system processes. A process is allowed to "FORK" new processes. Thus we get a cactus stack for the entire process tree. Since this implementation scheme seems to be unknown to large parts of the programming community, some general remarks are justified.

One of our main goals has been to keep the total amount of instructions down (we save programming time and code space). Since we are operating in a multiprogrammed environment, reentrancy becomes a central issue: There should exist only one copy of a (shared) routine simultaneously in memory, while the number of data incarnations may vary.

ALGOL60 was the first programming language to introduce reentrant code, by general recursive procedures working on a data stack. ALGOL60-implementations will require that the actual machine has an efficient indirect addressing facility, something like a base-, index- or stack-register. Most machines offer this, including our minis.

Reentrant procedures, using a data stack, also minimize the demand of data space, as only data cells belonging to the actual number of called procedures need be allocated. However, we are charged with extra data administration, but CPU is not a critical resource.

MARY supports RECURSive procedures, i.e. pieces of reentrant code that grab data incarnations from a stack generator on procedure entry, and deallocate them on procedure exit. (Note that the MARY term for an ALGOL60-"procedure" is "routine".)

Every RUNSYS process, system- or user-defined, is "implemented" by a stack (plus a small data area for saved registers and misc. process information). When a process executes a RECURS procedure P, this procedure will allocate its local data and
parameters on top of the actual stack. If there at a given time exist several data incarnations of P, we conclude that P is called from more than one process, or that it is called truly recursively (directly or indirectly) by a single process.

This process implementation solves a lot of problems in an elegant and efficient way:

* **No distinction** between normal procedure calls and monitor (or inter-monitor) calls is made. In fact, the concept of a "monitor", as a black box separated from user programs, vanishes. This uniformism did, for instance, simplify the testing considerably; see sec. 9.3 on MINIRUNSYS.

The "operating system" is simply a collection of normal RECURS procedures, some system data and an interrupt-handling section. No special "monitor calls" via software interrupts are needed.

See also sec. 15.1.2 for a general discussion of Hoare-monitors versus cactus-stacks.

* **Temporary working-space** problems for system routines are solved by letting the calling task supply the necessary (stack) space.

* **Normal scope rules** in MARY (or ALGOL60) give procedures, executed by cooperating parallel child processes, access to (shared) global data in textually enclosing blocks, executed by the parent process. This solves a few data allocation and protection problems, associated with process communication.
Consider the following example:

BEGIN
---
SEMA(BUFFER) VACANT:=...; \% Shared semaphore.
PROC PRODUCER=% RECURS:(INT I:=0; --VACANT-- )
PROC CONSUMER=% RECURS:(CHAR C:="*"; --VACANT-- )
} VACANT is used by the procedures.
---
FORK(---,PRODUCER()); \% Creating two child
FORK(---,CONSUMER()); \% processes.
---
--VACANT--
END;

Data stacks at run-time:

![Diagram of data stacks]

- CHILD1: Data area for PRODUCER.
- CHILD2: Data area for CONSUMER.
- PARENT: "VACANT".
- A shared PARENT-datum, accessible by CHILD1 and CHILD2 for interprocess communication.

Note 1: No other process than the owner process itself, and possible children, may (directly) access local stack data.
Note 2: If a parent process terminates (abnormally) before all its child processes are dead, the children will be automatically aborted in a "lavine"-like way, and an error message is given. Thus child processes are guaranteed that shared parent data (like semaphores) cannot be deallocated behind their backs.

A general cactus stack, being the consequence of this process implementation, may look like:

A digression: A fragmented cactus stack, caused by FORKing, corresponds to SIMULAs heap allocation of procedure incarnations, enforced by recursive co-routines.
3. Basic process-<i>or TASK</i> implementation.

A process basically "consists" of a data object of mode TASK (see below). In RUNSYS terminology we often use "task" to denote a process.

A more formal process definition, like process="program in execution", can be found in [Dennis, Horn, 1966].

A TASK-object will contain register savearea, task priority, task identification, pointers to parent-, sibling- and child-tasks, pointers to opened files, pointers to data objects for formatted IO (so-called IO-files), and a few more exotic attributes.

A RUNSYS task may <i>acquire resources</i>: CPU, raw primary storage (to accommodate a data stack, user routines, various data objects) and a variety of so-called SHARED-objects through calls on synchronization routines (e.g. for file access).

Tasks may be executed truly parallel or quasi-parallel, depending on the hardware. Concurrent execution is probably a correct term.

3.1. Layout of TASK-objects and data stacks.

3.1.1. TASK-objects, main attributes.

<table>
<thead>
<tr>
<th>TASK-OBJECT:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register savearea, see sec. 3.1.2.</td>
</tr>
<tr>
<td>SAVELIMIT</td>
</tr>
<tr>
<td>TIMELIMIT</td>
</tr>
<tr>
<td>PRIORITY</td>
</tr>
<tr>
<td>NAME, process id.</td>
</tr>
<tr>
<td>NEXTTASKLINK</td>
</tr>
<tr>
<td>PARENT</td>
</tr>
<tr>
<td>SIBLING</td>
</tr>
<tr>
<td>CHILD</td>
</tr>
<tr>
<td>OPENEDFILES</td>
</tr>
<tr>
<td>INPFILE</td>
</tr>
<tr>
<td>OUTPFILE</td>
</tr>
<tr>
<td>RESOURCENAME</td>
</tr>
<tr>
<td>&lt;misc.&gt;</td>
</tr>
</tbody>
</table>

(Max. amount of SAVE-space grant (Max. amount of CPU-time granted)

Parent task

Sibling task

List of descr. of opened files.

Standard IO-files for formatted IO.

List of acquired SHARED-objects, see sec. 3.3.

Possible semaphore on which the task is waiting.
A complete mode declaration of TASK is given in sec. 5.2.

A TASK-object occupies ≈ 40 words on NORD-SM. If this is extravagant for low-level system tasks, e.g. if we have many I/O-drivers in the system, some selected task attributes (like file pointers) may be allocated in separate data objects. The TASK-objects must then contain pointers to such objects.

3.1.2. Register saveareas (NORD-SM).

Since the NORD-SMs have rather few registers, 4 fixed memory locations must be allocated to administrate the data stack, and to keep track of the current SAVE/OLAY-code segment:

STACKDESCRIPTOR, a small display

STACKTOP
STACKLIMIT
STACKBASE

(See sec. 3.1.3 below for further details.)

CURRENTCODESEGDESCR

Pointer to current code-segment descriptor. Needed because of OLAY-code, see sec. 4.5.

These 4 cells behave like registers and must be saved/unsaved on task change/interrupts.

The register savearea has the following layout on NORD-SM:

<table>
<thead>
<tr>
<th>P-reg.</th>
<th>X-reg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>STACK-DESCRIP-TOR</td>
<td></td>
</tr>
<tr>
<td>CURRENTCODESEGDESCR</td>
<td></td>
</tr>
</tbody>
</table>


12 words
The position of the first two cells (P- and X-reg.) is compatible with hardware formats required by the 16 level interrupt systems on these machines.

Note 1: Only the register savearea-part of TASK-objects is machine-dependent.

Note 2: On a multiple-CPU version of KS-500 (the successor SM-machine), the 4 extra adm. cells cannot reside in common, fixed memory locations. Instead, each CPU must dynamically locate its cell-incarnations by indexing a shared, global array with its local CPU-number.

3.1.3. Data stack administration (NORD-SM).

The following stack structure applies:

RUNNING TASK:

Global cells and registers:

STACK-DESCRIPTOR

B-reg.

STACKDESCRIPTOR:
STACKTOP
STACKLIMIT
STACKBASE

B(ase)REG:

Data stack of running task:

Stack limit

First available stack cell

Current stack incarnation

Stack base

Copied on save/unsqueeze actions.
Stack addresses are biased by 128 to utilize the full hardware addressing range of 256 words:

The total stack size of a task is determined on task creation, i.e. by the FORK-operation.

A programmer can specify a stack size by writing:

```
OPTION STACKSIZE <size> NOITPO
```

in his main program. The loader will then fork a user task with prescribed stack size.

Observe that the machine registers may be regarded as a bottom level stack area.

A normal stack size for user tasks will be 150-600 words, depending on the depth of procedure calls and on the data size of procedure incarnations. System tasks require stacks of 25-500 words. The smallest demand is for some IO-drivers; the largest is for a TASKSINK-task for killing other tasks.

3.2. Life cycle of a task.

3.2.1. Task queueing and task priorities.

The NEXTTASKLINK-attribute in TASK-objects is used for task queueing. A task cannot stand in more than one queue simultaneously, so this is OK.
Tasks are assigned fixed priorities between 0 and 255; 0 being the highest. A child task cannot receive higher priority than its parent. User tasks will normally have STANDARDPRIORITY (=2 for the moment).

The CPU-queue is administrated as a Round-Robin queue. The clock causes task rotation.

3.2.2. Software interrupts (WAIT-instructions).

A task may perform so-called software interrupts. These are implemented as WAIT-instructions, containing a suitable offset in the rightmost part.

Ex. REQUEST(NORMALTERMINATION); %% Really UNSAFE MARY!

Sec. 3.5 surveys the available offsets/parameters.

On a multi-level interrupt system, a WAIT-instruction is the normal give-up-priority instruction. On a single level system, we have used an illegal instruction (which gives interrupt on execution) for this purpose.

The compiler will emit WAIT-instructions to flag standard MARY run-time errors (like empty pointers, illegal subscripts).

Software interrupts may be regarded as "monitor" calls with one parameter, the instruction offset. Thus we violate our own principle on uniformity of monitor calls! However, we must be able to do this kind of interrupts, e.g. to obtain detachment after linking ourselves into a semaphore queue. It's also a simple way for the compiler to arrange program abortion by emitting one WAIT-instruction.

I am chagrined to admit that the very low-level monitor calls must go via WAIT-instructions, not as ordinary procedure calls. But since these "calls" always imply process change, it seems naturally that they also are implemented specially. Perhaps they should not even look like ordinary procedure calls, cf. REQUEST(--)?
3.2.3. Task states.

A task, T, may be in one of the following three states:

* Waiting for no other resources than CPU. Then it's a member of the CPU queue, our READYQ:

```
READYQ:
```

* Currently receiving CPU, i.e. it is the RUNNING task:

```
RUNNING:
```

* Waiting in some semaphore task queue:

```
SOME SEMAPHORE:
```

System tasks, waiting for external interrupts (IO, clock), also conform the last set-up. The semaphore they're waiting on doesn't appear as a real semaphore. But "somewhere" (in the device descriptions, interrupt level heads) there is a task pointer, which points at a waiting driver. I.e. we have a semaphore task queue, consisting of a single task!

(Observation by Per Holager.)

However, externally triggered system tasks must, because of hardware peculiarities, often run on higher interrupt levels than normal tasks, being system- or user-defined. For this reason, they are assigned CPU in a special way and are not incorporated in the READYQ-RUNNING formalism. A dispatcher task also behaves a bit differently. See ch. 5 on interrupt-handling for further details.

Let's have a closer look at normal task behaviour, from birth to death.
Task creation:
FORK-operation by parent task;
the child is inserted into the READYQ.

A RUNNING task has released
a resource to a semaphore;
thereby triggering a waiting
task by putting it into
the READYQ.

Dispatcher selects
new RUNNING task.

READYQ

SEMAPHORE
QUEUE

If no available
resource exists,:
the RUNNING task is
hung up.

RUNNING

Clock
causes
task change.

(actions caused by RUNNING)

Task termination:
The RUNNING task has
executed a WAIT-instruction with
abortive effect
A special task is then triggered
to kill the task properly.

Thus, the usual task cycle is:

READYQ — RUNNING (- SEMAPHORE QUEUE) — READYQ.

Sec. 3.3 on synchronization and sec. 3.5 on task termination
will elaborate these matters.

3.3. On synchronization.

Synchronization primitives are essential to regulate
interaction between cooperating or competing sequential
processes. Whether such processes are executed by
several CPUs (true parallelism) or by a single one (quasi-
parallelism) has consequences only for the technical imple-
mentation of such primitives.
In the fifties and early sixties, with poor interrupt facilities in hardware and inadequate understanding of synchronization in general, many more or less ingenious ad-hoc mechanisms were developed to perform process synchronization.

Today, the theory of parallel processes, and resulting synchronization problems, should be well understood. However, the intellectual effort needed to design and implement synchronization primitives is not insignificant.

There are many ways of implementing such primitives:

1) The classic P-V formalism of Dijkstra [Dijkstra, 1968a], using integer semaphores.

2) The WAIT-SIGNAL mailbox system (of unknown origin). If there is no waiting process, the resource is discarded.

3) The event-based WAIT-CAUSE system of Burroughs [Burroughs, 1969]. A process may trigger all processes, waiting on an event.

4) The message system of RC-4000 [Brinch Hansen, 1970]. Resembles a semaphore system, but fixed-frame messages with explicit senders/receivers are exchanged.

The following demands must be met by a general synchronization mechanism:

* It must be safe, i.e. ensure mutual exclusion between competing processes when accessing shared resources.

* It should be deadlock-free. Deadly embrace should be avoided; alternatively detected and resolved. Acquired resources should be released on task termination.
* It must support binary/integer semaphores (distributing "control", or access to a shared resource), as well as a queue-system for exchange of similar but arbitrary data objects (message delivery functions).

* It should allow conditional waiting on a semaphore, if there is no resource. See sec. 3.3.4.6.

* All inter-process communication should go via calls on the synchronization primitives. That is, processes should be anonymous; they should not "know" of each other explicitly, e.g. through dangling TASK pointers.

Furthermore, process interaction should be channelled through a limited number of "gates"/"mailboxes"/"semaphores", being exclusively accessible from the cooperating processes. A process may wait on more than one semaphore, but not necessarily at the same time.

Some security aspects of synchronization are discussed in sec. 3.4.

3.3.1. Implementation of synchronization primitives.

We have chosen to implement a Dijkstra-type semaphore, with the extension that the associated resource supply is not an integer, but a corresponding number of queued resource elements. A resource may be of any (data) type, but resources administrated by a given semaphore must be identical (at least in some respect).

This generalized semaphore implementation is due to Per Holager [Holager, 1973], and was initially implemented by Mark Rain. It was tested and modified by me.

Thus, our sempahore is a double queue-head. It maintains a queue of waiting tasks and a queue of released resources, the supply. See figure on the next page.
One of the queues is always empty. Regrettably, this fact cannot be utilized to save space on our implementation (the wordlength is too short).

The name of our semaphore-variant constituted a problem. We wanted a short mnemonic name, and agreed upon SEMA.

3.3.2. Semaphore and resource modes in MARY.

The formal mode declarations of semaphores and resources are as follows:

\[
\begin{align*}
\text{MODE} & \quad \text{TASK,} \\
& \quad \% \text{A modal,} \\
& \quad \% \text{serving as a forward declaratio} \\
& \quad \% \text{SHARED()}, \\
& \quad \% \text{The same for parameterized} \\
& \quad \% \text{SEMA()}, \\
& \quad \% \text{modes, so-called "templates".} \\
& \quad \text{RESOURCEHEAD (M)=}[\text{REF SHARED (PROTEAN)} \text{ NEXT,}] \\
& \quad \text{REF TASK CREATOR,} \\
& \quad \text{REF SEMA (M) RECEIVER}, \\
& \quad \text{SHARED(M)=}[\text{VAL RESOURCEHEAD(M) HEAD,} \% \text{Note the} \\
& \quad \% \text{M VALUE}], \% \text{VAL-status.} \\
& \quad \text{SEMA(M)=}[\text{VAL REF TASK CREATOR,} \% \text{Another VAL.} \\
& \quad \text{TASKQ,} \\
& \quad \text{VAL REF SHARED(M) RESOURCEQ}]; \% \text{- and another}
\end{align*}
\]

The TASK-mode is still unspecified, but this is OK.

A semaphore declaration may look like:

\[
\begin{align*}
\text{MODE} & \quad \text{BUFFER=[S]CHAR;} \quad \% \text{Some resource mode.} \\
\text{SEMA(BUFFER) SUPPLY:=NIL;} \quad \% \text{Semaphore to administrate} \\
& \quad \% \text{SHARED (BUFFER)s.} \\
\text{INITSEMAPHORE(SUPPLY);} \quad \% \text{Initializing the system-controlled} \\
& \quad \% \text{CREATOR-attribute.}
\end{align*}
\]
Data structure:

A SEMA (BUFFER)-object, named SUPPLY:

- VAL-status gives protection.
- SUPPLY.CREATOR: Pointer to creating task.
- SUPPLY.TASKQ: Waiting tasks, if any.
- SUPPLY.RESOURCEQ: Released resources, if any.
- TASK-objects: Waiting tasks, if any.
- SHARED(BUFFER)-objects (see below):
  - System-controlled HEAD-part
  - User-controlled VALUE-part.

A corresponding resource declaration may be:

```
SHARED(BUFFER) B := (NIL, 'MARY*');
INITRESOURCE (B); % Initializing the system-controlled % HEAD-attribute.
```

The resource B has mode SHARED(BUFFER), and its VALUE-field can be used as an ordinary variable. See next page.

Data structure:

Not modifiable because of VAL-status.

- B.HEAD.NEXT: Link-field, used for semaphore queueing and to link acquired resources onto owner tasks.
- B.HEAD.CREATOR: Initial creator task.
- B.HEAD.RECEIVER: Next semaphore to receive this resource; needed for deadlock treatment.
- B.VALUE: Fully controlled by the user.

Note: The CREATOR-field of SEMA(M)-objects and the HEAD-field of SHARED(M)-objects are introduced solely to increase the security of the operating system.
3.3.3. The basic synchronization procedures, ACQUIRE and RELEASE.

To get hold of a resource, we write:

```
REF SHARED (BUFFER) RB=ACQUIRE (BUFFER, SUPPLY);
OUTTEXT(RB.VALUE); % Using it.
```

To free it back to a semaphore, we write:

```
RELEASE (BUFFER, SUPPLY, RB);
```

These two procedures correspond to Dijkstra's P- and V-operations, or "Passeren" and "Vrijgeven" as their full Dutch names are.

Note that a mode (e.g. BUFFER) must be supplied in the proc calls. MARY-procedures may work on classes of modes.

Then the procedures, slightly simplified:

```
PROC ACQUIRE=% SAVE RECURS (MODE M,
                   REF SEMA (M) S)       
                   REF SHARED (M):
BEGIEN
  <Check parameter consistency. Dead-lock situation?>
  REF SHARED1(M) R NOINIT; % For SHARED1, see below.
  REMOVESEMAVAL(M,S); % See note below.
  WHILE (DISABLEINTERRUPTS;
    NIL:=S.RESOURCEQ=:+R EMPTY)
  DO
    S:= RUNNING.SEMAPTR;
    INTOQ (RUNNING,S.TASKQ); % A small macro.
    ENABLEINTERRUPTS;
    REQUEST (DUMMYREQUEST); % Give up
  OD;
```

Wake-up point, when triggered by a RELEASE-call.
R.HEAD.NEXT := S.RESOURCEQ;
ASSIGNRESOURCE(R); % Link R onto RUNNING's resource list.
ENABLEINTERRUPTS;
R COerce (REF SHARED (M))
END; % OF ACQUIRE.

PROC RELEASE =% SAVE RECURS (MODE M,
    REF SEMA(M) S,
    REF SHARED (M) R):
BEGIN

  <Check parameter consistency.>
  REMOVESEMAVAL(M,S); REMOVESHAREDVAL(M,R);

  DISABLEINTERRUPTS;
  REF REF SHARED1(M) LAST := S.RESOURCEQ;
  WHILE LAST := REF SHARED1(M) EMPTY NOT DO
    LAST.HEAD.NEXT := REF REF SHARED1(M) := LAST; % Locate last % queue element.
  OD;

  DEASSIGNRESOURCE(R); % Link R onto the semaphore's
  R := (LAST) := R.HEAD.NEXT; % supply queue.

  REF TASK T := SEARCHPRIORITYQUEUE(S.TASKQ);
    % Delinks the first queued task of % lowest/best priority.
  NIL := T.SEMAPTR;
  INTOQ(T, READYQ);
  ENABLEINTERRUPTS;

  IF T.PRIORITY < RUNNING.PRIORITY THEN
    REQUEST(IDLE); % Go to sleep for a while, i.e. % into the READYQ.
  FI;
END; % OF RELEASE.
Note 1: Some UNSAFE data operations must be performed, in order to override the VAL-status of RESOURCEQ- and HEAD.NEXT-attributes. Two additional modes, SEMA1 and SHARED1, without VAL-specifications have been introduced to simplify the programming. (Not shown.)

Two conversion macros, REMOVESEMAVAL and REMOVESHAREDVAL, and the COERCE-macro from appendix B perform the technical mode transformations:

Ex. The REMOVESEMAVAL-macro. (A simple, inefficient version).

```
DEFINE REMOVESEMAVAL(M,S)=
    REF SHARED (M) DUMMY = FREE S;
    REF SHARED1(M) S =
        FREE DUMMY COERCE(REF SHARED1(M)) $
```

Note 2: Some of the called procedures, like SEARCHPRIORITYQUEUE, apply similar de-VALing tricks.

Note 3: Observe that the UNSAFE coercions:

```
REF SHARED (ANYMODE) REF SHARED (PROTEAN)
```

allowed linking of resources of arbitrary modes into lists.

Note 4: The list handling (of task- and resource queues) might have been generalized. This is postponed since necessary "dynamic" template-modes are not yet implemented in MARY.

Further technical comments, e.g. on disabling the interrupt system, have been put in section 3.3.4.
Example. A software classic: A cooperating PRODUCER-CONSUMER pa:

![Diagram of PRODUCER and CONSUMER tasks with semaphores]

Arrows indicate the cycling of resources (buffers) between a PRODUCER and CONSUMER task, via the semaphores FILLED and VACANT.

MARY PROGRAM.

%% The synchronization stuff is supposed to be predeclared, %
%% i.e. supplied in a system "prelude" or library.

BEGIN
  MODE BUFFER= [5] CHAR;
  SEMA (BUFFER) FILLED:=NIL,
    VACANT:=NIL;
  [3] SHARED(BUFFER) POOL:=NIL; % Pool of 3 resources.

  PROC PRODUCER=\% RECURS:
  BEGIN
    DO
      REF SHARED (BUFFER) B1 =ACQUIRE(BUFFER, VACANT);
      RELEASE (BUFFER, FILLED, B1);
    OD;
  END; % OF PRODUCER.
PROC CONSUMER=% RECURS:
   BEGIN
      DO
         REF SHAREDBUFFER = ACQUIRE(BUFFER, FILLED);
         RELEASE(BUFFER, VACANT, B2);
      OD;
   END; % OF CONSUMER.

INITSEMAPHORE(FILLED);
INITSEMAPHORE(VACANT);
FOR P IN POOL
   DO
      RELEASE(BUFFER, VACANT, INITRESOURCExP));
   OD;

FORK(<misc. info>, PRODUCER());
FORK(<misc. info>, CONSUMER());
------
<Main program is executed by the parent task, in parallel
with the forked child tasks.>
------
END; % OF MAIN PROGRAM.

It may be clarifying to sketch some of the data structures
involved. A snapshot just after the FORK-operations, when
the PRODUCER-task is filling its first buffer and the
CONSUMER-task is waiting inside ACQUIRE on a filled buffer,
is shown below:

- The tasks are named M (Main program-task), P and C for short.
Relevant data structures:

**TASK-OBJECTS:**

**DATA STACKS:**

- Remarks:
  - RECEIVER-attributes of semaphores are simply NIL'ed.
  - Normal inter-task links and down-stack links are not shown.
Two SHAREDBUFFER-resources are owned by the VACANT-semaphore; the P-task has the third.

The C-task is waiting on the FILLED-semaphore inside ACQUIRE, cf. its data stack and saved PREG-value.

The P- and M-tasks will be rotating between being RUNNING and sleeping in the READYQ (not shown).

Note the co-routine-like behaviour of cooperating parallel processes, when they perform cyclic synchronization operations.

3.3.4. Comments on the implementation of synchronization primitives.

3.3.4.1. Generality and simplicity.

The sketched scheme is general. It is simple to use. The coding of ACQUIRE and RELEASE is compact and straightforward. In the security-checking procedures (deadlock and consistency tests) we need only consider one type of data structure for semaphores and resources.

3.3.4.2. Security, technical aspects.

The scheme is safe against trivial coding errors. In MARY we can (as demonstrated) declare procedures that work on classes of modes, as long as the procedures only operate on RET's or ROW's, i.e. they don't have to know the actual size of the referred objects. The compiler will naturally check all procedure calls against inconsistent parameters.

The system-controlled fields of semaphores and resources possess VAL-status to prevent unauthorized changes from normal users. The synchronization procedures must admittly do some UNSAFE coercions (only available for system routines), but they are rather harmless.
In addition, the '=:='-operator will be redefined for SEMA- and SHARED-modes. Otherwise, we might have copied entire SHARED-objects into another, thus destroying system-controlled link fields.

A further discussion on security, with emphasis on common synchronization errors, is given in sec. 3.4.

3.3.4.3. Efficiency.

The implementation is efficient. Particularly the ACQUIRE-operation is fast.

The lists of waiting tasks and released resources are real FIFO queues:

- ACQUIRing task is inserted here.
- SEMA-object:
- First resource to be ACQUIRED.
- RELEASEd resource is inserted here.
- Triggered task is delinked here.

Thus, list chasing is left to the RELEASE procedure, which probably is less time-critical than ACQUIRE.

3.3.4.4. Code residency and reentrancy.

The synchronization procedures are SAVE (i.e. memory-resident) and reentrant. They must be SAVE for obvious reasons. Reentrancy is achieved by the RECURS mechanism for stack allocation, as we have too few registers on the NORD-SMs to manage without extra working cells in memory.

3.3.4.5. Derived synchronization primitives.

Let us demonstrate how 2 classes of common synchronization functions can be incorporated into the ACQUIRE-RELEASE formalism.
1) The classic integer or binary Dijkstra semaphore, controlling access to a shared datum or distributing abstract control of some kind.

The shared datum-situation is trivial; we have a semaphore with only one resource. (See later.) To distribute "control", we must introduce a dummy resource of mode SHARED(VOID):

Ex.

```
SEMA(VOID) ACCESS:=NIL;
SHARED(VOID) DUMMY:=NIL;
<Initialize ACCESS and DUMMY.>
-----
ACQUIRE (VOID,ACCESS);
    <crit. region> Only one process at a time may be inside this region.
RELEASE (VOID,ACCESS,DUMMY);
-----
```

The notation in this example may be sweetened by using macros, see later.

If more than one process at a time should be granted access to something, it is natural to initialize a semaphore with a corresponding number of "resources". If this "something" has mode M, these dummy "resources" for distributing access rights might be SHARED(REF M)-objects.

If the maximum number of processes that can share an attribute is dynamic (cf. "concurrent readers and writers"), we must probably introduce some higher-level synchronization facilities. However, the hierarchic structure of FILESYS prevents this situation to arise; see sec. 6.5.
2) When a waiting process is activated, it will often test on some "flags" to find out exactly which condition(s) triggered it. This wake-up information is more easily passed via a resource pointer in our ordinary formalism. This may, however, be less efficient.

3.3.4.6. Variants of the primitives.

We may need variants of ACQUIRE and RELEASE.

* Specifically, we may want to allow conditional waiting on a resource in ACQUIRE, so that we are not automatically detached if there are no resources. This is a useful remedy; e.g. if there are no more TTY-input buffers, or if an interactive user tries to lock a file, already locked by somebody else.

* Furthermore, we may want the RELEASE procedure to insert resources first in the resource queue, i.e. to assign resources some sort of priorities. This can be utilized to give TTY-input echoes priority over ordinary output lines.

The above functions are implemented simply by adding an extra BOOL-parameter in the parameter lists of ACQUIRE and RELEASE. The extensions are trivial and not shown.

3.3.4.7. Higher-level primitives.

Higher-level synchronization mechanisms are easily implementable by defining additional procedures and macros. Macros may also give the synchronization stuff a more elegant syntax.

* Ex. WAITON and CAUSE procedures.

A CAUSE-call will trigger all processes, having performed WAITON.

A dummy resource, belonging to the CAUSing process, will be needed
CAUSE consists of:

\[
\text{RELEASE (~ , ~ , ~);}
\]
\[
\text{REQUEST(IDLE); \quad \% \ Go to sleep, so that one of the}
\]
\[
\text{\quad \% \ waiting tasks may acquire the}
\]
\[
\text{\quad \% \ resource.}
\]
\[
\text{ACQUIRE (~ , ~ );}
\]

WAITON consists of:

\[
\text{ACQUIRE (~ , ~ );}
\]
\[
\text{RELEASE (~ , ~ );}
\]

If the tasks involved have equal priority and the READYQ
is administrated as a Round-Robin queue, these Burroughs-
inspired procedures will do their job properly.

3.3.4.8. Macros for initializing semaphores and resources.

It is a nuisance to have to initialize semaphores and resources
in a special way, cf. sec. 3.3.2. And if we don't, the ACQUIRE-
or RELEASE procedure will arrange task termination. Really,
the MARY language should have offered some automatic facilities.

Since we have to manage without extra help from the compiler,
the following macros should be used:

Ex. Initialization macros.

\[
\text{DEFINE INITSEMA (S) = NIL;}
\]
\[
\text{\quad INITSEMAIPHORE(S)$,}
\]
\[
\text{\quad INITSHARED(R,V)=(NIL,V);}
\]
\[
\text{\quad INITRESOURCE(R)$;}
\]
\[
\text{SEMA (BUFFER) SUPPLY:=INITSEMA(SUPPLY);} 
\]
\[
\text{\quad SHARED (BUFFER) B:=INITSHARED (B, 'MARY"');}
\]

The reason why we cannot use the initialization procedures
directly, e.g. to write:

\[
\text{SEMA(BUFFER) SUPPLY:=INITSEMAIPHORE(SUPPLY);}
\]

is that the name "SUPPLY" is not syntactically available before
the semicolon. Maybe our compiler people can fix this, but
accessing a cell before it has been properly declared is potentially dangerous. See also MARY TEXTBOOK, p. 67.

Note: These macros are only applicable in declarations of simple variables. For instance, the array declaration:

\[
[3] \text{SHARED (BUFFER) POOL:=INITSEMA (POOL)};
\]

will give a compilation error.

- It seems that we need a more powerful language!

3.3.4.9. Macros for declaring critical regions à la Brinch Hansen.

See [Brinch Hansen, 1973a], pp. 83-86.

Ex. Critical region macros.

\[
\text{MODE SHARED\textsc{DATUM}(M)=} \\
\quad \{ \text{SHARED(M) DATUM,} \quad \frac{\%}{\%} \text{Mode for shared} \\
\quad \text{SEMA(M) ACCESS}; \quad \frac{\%}{\%} \text{variables.}
\]

\[
\text{DEFINE INITSHARED\textsc{DATUM}(D,V)=} \\
\quad \{ ((\text{NIL,V}),\text{NIL}); \quad \frac{\%}{\%} \text{The initial value.} \\
\quad \text{INITRESOURCE (D.DATUM);} \\
\quad \text{INITSEMAPHORE (D.ACCESS);} \\
\quad \text{RELEASE(MODE(D.DATUM.VALUE), D.ACCESS, D.DATUM)} ;,
\]

\[
\text{INSPECT, VAR REGION, BODY NOIGER=} \quad \frac{\%}{\%} \text{Macro delimiters} \\
\quad \frac{\%}{\%} \text{are underlined.}
\]

\[
\text{BEGIN} \\
\quad \text{MODE M = MODE(VAR.DATUM.VALUE);} \\
\quad \text{REF SHARED(M) PTR=ACQUIRE(M,VAR.ACCESS);} \\
\quad \{ \text{VAL REF M VAR=PTR.DATUM.VALUE;} \\
\quad \text{BODY; \% VAR may now be used as} \\
\quad \frac{\%}{\%} \text{an ordinary variable of mode M.} \\
\quad \} \\
\quad \text{RELEASE (M,VAR.ACCESS,PTR);} \\
\quad \text{END} ;
\]
SHAREDDATUM(INT) N:=INITSHAREDDATUM(N,10);
---
INSPECT N
REGION
---
Critical region
ININT():=N;  %% N is re-declared as a VAL REF INT
            %% inside the critical region.
---
NOIGER;

Critical regions in this fashion may of course be nested. However, synchronization-calls don't always conform to an onion-structure:

Ex. Merged synchronization calls.
ACQUIRE (-,S1);
ACQUIRE (-,S2);
RELEASE (-,S1,-);
RELEASE (-,S2,-);

Finally, to implement "conditional semaphore-expressions" by several linked ACQUIRE-calls in some AWAIT <Bool expr.>-formalism is also possible, but assumes "clairvoyant" semaphores in order to prevent deadlocks.

Maybe the reader has been confused and even bored by all this MARY artistry. However, I think these second-order synchronization facilities are important, because:

1) It's necessary to explore the application area of our synchronization primitives. Are they powerful enough?
2) It's desirable to declare the best set of high-level synchronization tools - to ease the programmer's work in general, and to assist normal users in this field, since such RUNSYS tools are available to everyone.
3.3.4.10. Basic, low-level synchronization.

We have introduced synchronization primitives to ensure mutual exclusion between parallel processes, when doing operations on shared data. But how do we guarantee unique access to a semaphore inside the primitives? In fact, we have only pushed the basic synchronization problem one level away from the user, pretending the semaphore operations are atomic or uninterruptable.

There are, in practice, two safe ways of doing operations on a (shared) semaphore:

1. The classic solution is to "freeze" interrupts, and thereby monopolizing the CPU, during semaphore update-operations. This was done inside ACQUIRE and RELEASE previously.

   **Ex. Low-level critical region, version 1. (For SM4).**

   ```
   DEFINE DISABLEINTERRUPTS=EXECUTE(4'D142')$, %NB,
   ENABLEINTERRUPTS=EXECUTE(4'D102')$; % macros!
   {
      DISABLEINTERRUPTS;
      <Critical region for semaphore updating.>
   }
   ENABLEINTERRUPTS;
   ```

   It is assumed that interrupts are disabled for very short intervals only.

2. On a multiple-CPU machine, the above method won't work. It's often impossible (and certainly undesirable) to halt all the other CPUs, merely to achieve the necessary synchronization effect.

   A more selective method must be found. That is, we only want to prevent that more than one CPU is doing operations on the same semaphore simultaneously.
Of course, hardware designers have devised a standard solution to this dilemma, in form of an uninterruptable memory-register swap-instruction (or something similar). Note that this solution leaves the basic synchronization problem to hardware (the memory module or the CPU-memory bus). That is, only one CPU may do this swap-operation on a given memory location simultaneously.

This swap-operation has its MARY-equivalent in the ':=:'-operator:

```
Ex. 5:=:I  %% Puts 5 into I and
       %% yields I's previous value as a result.
       %% In one instruction for one-word
       %% objects.
```

Thus:

One of the cells in the semaphore (e.g. the CREATOR-field) must be used temporarily to synchronize the CPUs.

A unique data value that cannot occur naturally in this cell, is used to mark the semaphore as "locked".

```
Ex. Low-level critical region, version 2.

REF TASK DUMMYVALUE=<some unique value>;
------
REF SEMA (M) S = <actual semaphore>;
REMOVESEMAVAL (M,S);
REF TASK OLDCREATOR:=NIL;

WHILE DUMMYVALUE:=:=S.CREATOR:=OLDCREATOR ISNT DUMMYVALUE
DO OD;  %% Loop until unlocked.
{<Critical region for semaphore updating.>}
OLDCREATOR:=S.CREATOR;
```

By this solution, the CPUs will actively run in waiting loops on low-level synchronization conflicts. But even if the waiting time is short, we may experience serious CPU-memory access problems. In fact, on the Hydra system [Wulf et al., 1973] they had to implement selective inter-processor wake-up interrupts to prevent waiting CPUs from generating memory accesses to shared semaphore cells and loop instructions.
The kind of "raw" synchronization, described in point 1) and 2), is also used elsewhere in low-level procedures. It is needed if the overhead involved (time/space) by calling ACQUIRE/RELEASE is intolerable, or if it simply is impossible to use ACQUIRE/RELEASE. For instance, some system tasks don't operate in the RUNNING-READYQ formalism.

Warning: When using the above mechanism, we should program carefully to avoid undesired effects. Disable-enable pairs don't work like ordinary parentheses:

Ex. Nested disable-enable pairs.

PROC P=% SAVE RECURS:
BEGIN
    DISABLEINTERRUPTS;
    ----- Interrupts are now enabled!
Q();
    ENABLEINTERRUPTS;
END; % OF P.

PROC Q=% SAVE RECURS:
BEGIN
    DISABLEINTERRUPTS;
    -----
END; % OF Q.

3.4. Deadlock and security aspects of synchronization.

Ordinary users have full access to forking and synchronization facilities in RUNSYS. As the number of tasks, semaphores and resources are dynamic, unexpected or unintentional situations will occur; and the operating system must be prepared to tackle them.

Security precautions fall into three categories:

1. Preventive actions
2. Discovery of errors
3. Recovery from errors
3.4.1. Language protection.

As described in sec. 3.3.4.2, the MARY language offers some protection against inconsistent use of semaphores and resources (mode checks, VAL-status).

However - and this is important - it is the user's responsibility to insert necessary synchronization calls (ACQUIRE - RELEASE pairs) when updating shared data; and to insert them correctly. MARY is a serial language with no semantic rules to prevent a procedure from doing operations on shared, global data. I.e. the compiler doesn't know whether the code of a procedure is executed by the parent task, by a child task, or both.

Even if a variable is declared as a SHARED (M)-object, any task executing a procedure which has textual access to this variable, may "diddle" with the user-controlled VALUE-field of the object - without preceding ACQUIRE (---) calls.

3.4.2. Task tree protection and related subjects.

Our hierarchic task structure guarantees shielded access and safe data allocation of shared semaphores/resources. A parent task can never be deallocated before all its children have been properly terminated, thus preventing wild pointers to non-existent parent semaphores/resources.

That is, in principle we may have a water-tight system. But in practice, we may do programming errors. E.g., we may pass a pointer to a child-resource onto a parent-semaphore. This is really an up-stack pointer, i.e. a violation of scope rules for pointers. This problem is well-known from ALGOL68, but MARY does not yet support run-time checks on inter-stack pointer-assignments. Furthermore, we can allocate semaphores/resources in private heap objects (MEMLINK-objects, see sec. 4.4) which also get deallocated on task termination.
Therefore, **CREATOR**-attributes have been introduced in semaphores and resources to improve RUNSYS security. It also gives better error reporting if we explicitly know the original creator task.

The following task tree **restrictions** apply:

A task may only use semaphores/resources, whose **CREATORs** are **at least as global** as the task itself.

The **CREATOR** of a resource must be at least as global as the semaphore it is released to.

Still, a few problems remain:

**Ex. IO-communication.**

Consider the following task tree:

![Task Tree Diagram]

Some system task

Some user or system task

An IO-driver

IO-communication is requested.

User tasks

If user task T1 and IO-driver D want to perform synchronization, the relevant semaphores and resources must belong to A.

This may cause problems for T1, if T1 has to create a new resource (e.g. a buffer) before releasing it to a semaphore in A, thus triggering D.
That is, we need a **global space generator** for resources with possibilities to redefine task ownerships (CREATOR-fields). (A general heap mechanism would have done the job, but there are some delicate problems with garbage collections in a multi-process environment. See sec. 4.10.)

A similar space generator for semaphores is also desirable.

For the time being we must rely on the "manual" MEMLINK-generator to obtain dynamic data objects.

In real-world operating systems, the resource-ownership conflict in the above IO-example can be solved:

- by letting IO-buffers be statically pre-allocated, or
- by not letting a user task die during IO-operations.
  E.g. in RUNSYS, a task waiting on a semaphore can not be terminated (except for a time-out/deadlock abortion).

### 3.4.3. Consistency- and deadlock testing.

#### 3.4.3.1. Data structures revisited.

**System-controlled attribute in SEMA-objects:**

- **CREATOR** - points at allocating task.

**System-controlled attributes in SHARED-objects:**

- **NEXT** - used for resource queueing,
- **CREATOR** - same as for SEMA-objects,
- **RECEIVER** - points at the semaphore to which the resource is expected to be released.

These attributes have been formally declared in sec. 3.3.2.
Relevant attributes of TASK-objects:
RESOURCENLIST - list head for all acquired resources,
SEMAPTR - actual semaphore on which we're waiting,
PARENT, CHILD, SIBLING - self-explanatory.

Ex. Typical situation for two cooperating parallel processes, T1 and T2.

T1: is holding R, which later will be released to S, thereby triggering T2.

The RECEIVER-attribute can be set by writing:

S WILLRECEIVE R; % A system operator is invoked

Generally, the RECEIVER-attribute should be initialized before a RELEASE call. The attribute is checked, but not touched by ACQUIRE and RELEASE. It is basically used by a deadlock-detect routine and by task termination routines. It may freely be used by programs, that need this kind of return information. For instance, RECEIVER-attributes in IO-"messages", delivered to IO-drivers, point at a semaphore on which waiting tasks may hang. (Observe that inter-task communication goes via semaphores; the interacting tasks are hidden to each other).

Furthermore, we may consider to introduce OWNER-attributes in SHARED-objects to avoid time-consuming searching among acquired resources of a task, when establishing resource ownerships.
3.4.3.2. **CREATOR consistency.**

Both ACQUIRE and RELEASE will test that the CREATORS of actual semaphores and resources are compatible with each other, and with the tasks involved.

It may be practical to let RELEASE use "RUNNING" as an initial value to the CREATORE-field of resources/semaphores, if the field is empty. This will save explicit initialization calls, when a (parent) task is initializing a local semaphore with local resources.

3.4.3.3. **Deadlock problems of synchronization.**

Synchronization deadlocks will usually block a user's internal task tree. That is, they are caused by deadlocks on user-controlled resources, as the distribution mechanisms for **shared** system resources (like mass storage FILES or formatted I0FILEs) are practically deadlock-free. Thus we don't do any great harm by letting the user pay for his misfortune, by task **abortion**.

Deadlocks are attempted detected by the ACQUIRE-routine; alternatively by a **time-out** function (max. 5 minutes long-wait).
Ex. Deadlock situation, detected during ACQUIRE(--,S1).

Here, the RUNNING task and task T2 are blocking each other. There may, however, be some other tasks that possess resources with RECEIVERS set to S1 and S2. Hence this possibility must also be investigated. (We assume that all RECEIVER attributes have been properly initialized).

The choice is, which of T2 and RUNNING should be killed: The "least global" one; both; or maybe the entire task tree of the user? I guess the former alternative should be preferred.

A common but rather trivial deadlock situation is that we're trying to acquire an already acquired resource. This happens particularly within FILESYS routines, and is taken care of by a higher-level synchronization function, LOCKFILE.

Perhaps such a facility should be generalized for all SHARED-objects, so that nested ACQUIRE-RELEASE calls will have no effect. Note that nested synchronization calls are only a problem for SHARED-objects, that distribute control.

Lastly, there are important resource types (raw space on primary and secondary storage, CPU) that cannot be expressed as SHARED-objects in the conventional ACQUIRE/RELEASE formalism. The derived memory deadlock problems are treated in sec. 4.7, while sec. 8.2 deals more generally with resource allocation among competing tasks.
3.5. Software interrupts, complete presentation.

As mentioned earlier, we need a software interrupt- or "executive" request-mechanism to give up priority. The macro

\[
\text{REQUEST (<interrupt-type>);}
\]

will emit a WAIT-instruction with a proper offset.

The request-types are declared as elements of an ordinary MARY SET-mode:

\[
\text{SET REQUESTTYPE=} \langle \text{NORMALTERMINATION, ERRORTERMINATION, SOURCELANGUAGEERR, SUBSCRIPTRANGEERR, --- MAXTIMEERR, MAXMEMERR, IDLE, DUMMYREQUEST, ---} \rangle;
\]

**Semantics of software interrupts:**

1) DUMMYREQUEST causes no other actions than "detachment". It is used if the task is already hung up in some waiting queue (by ACQUIRE or inside IO-drivers.)

2) IDLE implies that RUNNING will be put back into the READYQ.

2) All other request-types cause termination of the RUNNING task.

Request-types 2) and 3) are only meaningful for tasks that cannot be triggered by external interrupts.

When a software interrupt has occurred, the interrupt-handling section will check whether the request-type is different from DUMMYREQUEST. If it is, it must be further investigated, and necessary information is passed via a global REQUESTOR-variable to the dispatcher (see sec. 5.4); saying that RUNNING has executed a software interrupt. Then we do NIL=>RUNNING, and pass control to the above dispatcher. The reason why it's the dispatcher's job to do (a part of) the remaining work, is simply that we cannot call RECURS procedures from the interrupt-handling section. It has no stack and we cannot afford the time.
Thus, the dispatcher checks whether the "REF TASK" REQUESTOR-variable is non-empty, and releases the actual task onto a REQUESTEVENT-semaphore, to trigger a task called REQUESTHANDLER (see below).

Note, no other tasks (except system drivers) may have executed software interrupts in the meantime, as we have not yet chosen a new RUNNING candidate.

3.5.1. The REQUESTHANDLER.

The REQUESTHANDLER-task is executing the REQUESTHANDLER-procedure, which goes like:

```
DO
  REF SHARED(REF TASK)T = ACQUIRE (REF TASK, REQUESTEVENT);
  ----- 
  IF <interrupt-type>=IDLE
   THEN INTOQ(T.VALUE,READYQ);
   ELSE <terminate the task.>
   FI;
  ----- 
OD;
```

Note that task-"resources" formally are SHARED (REF TASK)-objects, being separately allocated. This simplifies programming elsewhere.

The REQUESTHANDLER-task (RH), being the very task that arranges task rotation on REQUEST(IDLE)-calls, rotates like any other task. When triggered, it is first put into the READYQ. Then it becomes RUNNING. Finally it is hung up on the REQUESTEVENT-semaphore; and so on. Let's study an example to better understand how tasks rotate in the READYQ:

Ex. READYQ-rotation.

Suppose two tasks, T1 and T2, both are executing:

```
{ DO
  REQUEST(IDLE);
  OD;
```
T1, T2 and RH are assumed to have equal priority.

Suppose T1 is RUNNING, T2 is in the READYQ, and RH is hung up on its semaphore; cf. situation (1).

<table>
<thead>
<tr>
<th>Situation:</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUNNING:</td>
<td>T1</td>
<td>T2</td>
<td>RH</td>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>READYQ:</td>
<td>T2</td>
<td>RH</td>
<td>~</td>
<td>T2</td>
<td></td>
</tr>
<tr>
<td>The semaphore REQUESTEVENT's TASKQ:</td>
<td>RH</td>
<td>RH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.RESOURCEQ:</td>
<td>T1</td>
<td>T1,T2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

First T1 executes REQUEST(IDLE), causing RH to be put into the READYQ and T1 to be passed as a resource to the REQUESTEVENT-sempahore. T2 becomes new RUNNING. Situation (2).

Then T2 executes REQUEST(IDLE). RH becomes new RUNNING, and T2 has been released as a second resource to the REQUESTEVENT-sempahore. Situation (3).

Now RH will acquire both T1 and T2, and put them back into the READYQ. Situation (4) after RH has been hung up again and T1 is RUNNING, is identical with situation (1).

The example fully demonstrates the generality of our synchronization primitives. Even TASK-objects can be passed as resources between tasks.

Note: The current functions of REQUESTHANDLER are rather modest, and may easily be carried out by the dispatcher itself. However, in case we want to implement new request-types later, the generality of the existing set-up may be needed.
3.6. Task creation and termination.

3.6.1. Task creation.

Tasks are created by a FORKER-routine, which allocates and initializes a new TASK-object and its data stack. The new task is inserted into the READYQ.

The FORKER-routine is callable via a FORK-macro, which sets up the correct code environment for a new task:

```
DEFINE FORK(TI,CODE)=

(----

IF FORKER(TI,..)          \% TI represents Task
THEN
  CODE;                   \% Information, see later.
  \% Executed by
  REQUEST(NORMALTERMINATION)\% the forked task.
FI $;
  \% The parent task continues
  \% after "TI".
```

The created child task wakes up inside the FORKER-routine. This is accomplished by initializing its PREG, BREG, STACK-DESCRIPTOR and CURRENTCODESEGDESCR properly. The child task will then leave FORKER with TRUE as result, execute the code supplied, and terminate by performing REQUEST (NORMALTERMINATION) - if it hasn't been killed already.

The TI-parameter is a pointer to a data object of mode TASKINFO, which specifies some attributes of the new task.

<table>
<thead>
<tr>
<th><strong>TI</strong></th>
<th><strong>priority</strong></th>
<th><strong>timeLimit</strong></th>
<th><strong>saveLimit</strong></th>
<th><strong>StackSize</strong></th>
<th><strong>Windups</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>An integer ∈ [0,255].</td>
<td>Max. no. of milliseconds CPU-time grant.</td>
<td>Max. amount of SAVE-space granted, incl. data sta</td>
<td>An integer, usually ∈ [200,600].</td>
<td>List of data objects, describing routines (and their parms) to be executed upon task termination, see sec. 3.6.2.2. Only allowed for system routines to supply.</td>
</tr>
</tbody>
</table>
The TIMELIMIT- and SAVELIMIT-attributes of the parent task are accordingly decremented. If one of them gets negative, the parent is killed. After child task termination, (the remaining parts of) the child's LIMIT-resources are returned to the parent.

When forking the bottom-level/basic user task, i.e. the local JOBINTERPRETER-task, a global system variable named AVAILABLESAVEMEM is also decremented. I.e. some SAVE-space is pre-reserved to prevent memory deadlocks. If there is not enough available SAVE-space, an error message is produced, or the backlog routines may delay task creation for a while.


The CODE-parameter in FORK (--,CODE) may be any MARY-expression, but it's usually a procedure call.

Ex. FORK invocations.

\[
\text{FORK (--, \texttt{READ (BUFF, <file>); WRITE (BUFF, <file>)},\quad \text{CODE-text}}
\]

\[
\text{FORK (--, \texttt{CONSUMER()},\quad \text{CODE-text}}
\]

The CODE-text has textually access to variables in the parent's program.

As stated in sec. 2.3, a parent task should not be allowed to terminate before all child tasks are dead/killed.

However, due to normal scope rules in MARY, we need more selective mechanisms, preventing parent tasks to exit a routine before all "local" child tasks are dead. - Again we face the problem of up-stack data references!

Thus, a trap called CRITICALBLOCKTRAP will be prepared when a routine invokes the FORK-macro:
Ex. Data stack of a parent task, having just forked two child tasks.

Data stack growth

Normal stack driver for RECURS routines.

Current routine-incarnation of parent task.

Inserted trap

CRITINFO-object, containing critical block trap data.

TASK-object of first child.

TASK-object of second child.

Misc.

STATIC LINK

DYNAMIC LINK

RETURN ADDR.

Parms and local data.

RETURN ADDR is altered to point at REQUEST(CRITICALBLOCKERK)-instr., causing task termination.

Pointing indirectly at critical block trap of parent task.

Next available stack location, STACKTOP.

Comments.

The first FORK-call by a routine will cause allocation of a data object on the stack of mode CRITINFO (see figure). The normal return address of the routine incarnation is set to point at a software interrupt instruction in this object, as long as the
number of active local childs is greater than zero. The return address is restored when the last child is terminated.

Note that CRITINFO- and TASK-objects are allocated on the parent's stack, via a LOC generator for stack allocation. The stack is cut back on exit of the actual routine.

In case we want to let a parent task wait on termination of all local child tasks (to avoid critical block trap-abortion), there exist some system routines to do this. They will construct descriptions of RELEASE-routines and insert such into the WINDUPS-lists of the child tasks. The parent task will then be triggered upon child task termination. In some systems this facility is termed a JOIN-operation, being an inverse FORK-operation.

3.6.2. Task termination.

3.6.2.1. Causing task termination.

The only way to terminate a task is to let it execute a software interrupt instruction, with offset $\xi$(IDLE, DUMMYREQUEST).

A task may terminate normally, i.e. execute a REQUEST (NORMAL-TERMINATION), being emitted by the FORK-macro or by the compiler before final END in a main program.

However, abnormal task terminations will frequently occur. They may be caused by run-time traps, prepared by the compiler (e.g. for empty pointers and invalid subscripts), or by enforced execution of software interrupt instructions.

In fact, a standard killing method is to alter the saved PREG of a non-active task to point at some abortive software interrupt instruction.
This is used:

* on critical block errors - really the return address of the actual routine is altered,
* to kill possible child tasks when a parent task is killed,
* on max time abortions,
* in case of long-wait abortions,
* on TTY "break"-commands.

When REQUESTHANDLER treats a software interrupt, it will release the task involved to the TASKSINKQUEUE-semaphore, if the request-type wasn't IDLE. This semaphore controls a task called TASKSINK, executing the termination actions.

The reason why we cannot let REQUESTHANDLER do the termination job, is that we are expecting output messages (e.g. why the task was aborted), and REQUESTHANDLER is in no position to wait on IO!

Likewise, the task actually being terminated cannot do it. Its data stack may still be used by active children, and the very termination reason may be stack overflow!

3.6.2.2. The TERMINATE routine.

This routine is executed by TASKSINK. Before TERMINATE is called, TASKSINK has arranged for forthcoming output texts to be directed to the print file of the terminating task. Special actions must be carried out in case of printfile overflow, or if the printfile is currently locked by the terminating task.

The TERMINATE routine is an OLAY routine. Its main function is to free all possible resources of the actual task back to their respective "owners" (the system, its parent).

Before some system resources can be released, they must be updated or prepared for deassignment, so that the data structure of such resources become consistent. For instance, the last written buffer of an opened file must be correctly output.
For user-defined resources, such consistency-updating is impossible or at least very difficult.

**TERMINATE actions in chronological order:**

* Task-restart checking.

As the terminating task may be in the middle of a "critical update operation", it should possibly be put back into the READYQ to complete the operation. See sec. 10.3.3 in the security chapter for further details.

* Killing of all child tasks.

A task tree must be terminated bottom-up. Child tasks are routinely killed by causing them to wake up in abortive software interrupt traps.

We may execute the TERMINATE-routine *recursively* to kill all children; or it may be done in a *step-wise* manner, by only aborting first-generation child tasks directly, and let their termination kill the grandchildren, etc..

Anyhow, we face a minor synchronization problem: How to trigger termination of the parent task, when all the child tasks have been successfully terminated?

Solution: On termination of the last child (the one that clears the CHILD-pointer in the parent task), we test whether a flag called KILL is set in the parent task. If so, we release the parent (for the second time) to the TASKSINKQUEUE-semaphore. This time TASKSINK succeeds in killing the parent. This elegant solution is due to Harald Børseth, [Børseth, 1975].

* Printing of task status.

The termination cause and misc. task information will be written out.

* Closing of IOFILES.

All private IOFILES for formatted IO are "closed", i.e. the last text buffer is written out.
* Closing of opened mass storage files.

These files hang in the OPENEDFILES-list of the task. Mass storage files are hierarchically organized, and care must be taken to unlock/close them in postfix order. A simple iterative algorithm, inspecting the file levels, will suffice.

* Releasing acquired SHARED-objects.

These are queued up in the RESOURCENAME of the task. They are released to their corresponding RECEIVERS, and presumably in a consistent shape. Trivial deadlocks, caused by abnormal termination of tasks that have locked shared resources (like read/print-files of a parent task), can thus be avoided.

As mentioned briefly in sec. 3.4.3.4, (locked) mass storage files constitute a subgroup of acquired SHARED-objects, which require special treatment. Indeed, our SHARED-objects represent a generalization of resource descriptions, which in other systems have been restricted to mass storage files and similar system resources.

* Removing task object from the task tree, and incrementing LIMIT-attributes of the parent task.

* Adjusting critical block trap of the parent task.

* Executing routines appended in the WINDUPS-list.

* Deallocation of MEMLINK-objects.

Finally all private heap objects, incl. the task's data stack, can be released: We scan all MEMLINKs and deallocate those with matching OWNERS.

The AVAILABLESAVEMEM-variable will be incremented if the task is a bottom-level user task.
4. Basic allocation of primary storage (memory).

4.1. General remarks.

Administration of primary memory is one of the main functions of an operating system. Programs cannot be executed and data cannot be accessed directly, if we lack available memory space to store such.

On large computers, as the Ul108, space problems don't seem to exist for ordinary users. On relatively small computers (with 16 - 32 K memory) the very problem is that the amount of code and data exceeds the available memory, or even worse: the addressing range (32 or 64K). In the latter case we cannot simply buy more core modules to solve the space problems.

Often, we must implement some kind of a virtual memory system. But virtual memory is no ultimate solution to space problems. "Trashing" is a well-known phenomenon. Any method that can reduce the number of disk requests or the amount of resident code/data, must be seriously considered. For instance, reentrant code may prove advantageous.

Various memory management techniques have been implemented in operating systems, depending on the underlying hardware, the need for efficiency and flexibility, and the need for virtual memory.

In our opinion, design and implementation of space allocation primitives are non-trivial work. Furthermore, because of the diversity in hardware addressing modes and in the application areas of systems, it is difficult to develop general/portable solutions. IO and interrupt-handling can be put into "black boxes", but hardware addressing peculiarities penetrate the entire operating system; if we want to utilize hardware efficiently.
4.2 On storage allocation in RUNSYS

We're operating under the following constraints:

* The inherent generality of RUNSYS (in contrast to a dedicated real-time system) will require a powerful primitive for dynamic storage allocation. Data objects will constantly be created and destroyed.

* Memory, disk/drum-channel capacity and manpower are our scarce resources. We have plenty of CPU-time.

* Reentrancy is required in our multiprogramming environment. I.e. allocated code segments (resident or olayable ones) must be sharable between tasks.

* The actual computers are assumed to offer poor hardware mechanisms to support virtual memory. Thus, we have to develop software solutions for this. See later.
4.3 General storage allocation mechanisms offered by RUNSYS

4.3.1 Space generators needed to execute MARY programs

A programming language describes, explicitly or implicitly, constructs which imply allocation of storage during execution of programs. E.g. ALGOL- and SIMULA-programs need run-time systems for respectively stack- and heap-allocation of data objects. Run-time systems like this may be regarded as a part of the underlying operating system.

Thus MARY RUNSYS must support:

* Allocation of data in static data areas (e.g. COMMON blocks) and on user stacks.

* Allocation of general data objects - with mechanisms for automatic garbage collection and preferably compaction. Such a "heap" facility is standard in languages like LISP, SIMULA and ALGOL68. The implementation techniques are (or should be) well-known. Details of possible heap implementations in MARY are discussed in [Conradi, 1974a], and one of them is planned to be realized. See sec. 4.10.

* Allocation of SAVE (memory-resident) as well as OLAY (otlayable) code, where only the descriptor for a piece of code needs reside in memory.

4.3.2 System-controlled space generators

In addition RUNSYS must support:

* Allocation of data objects for various system purposes (I/O-buffers, FILE-objects). This is not and automatic heap facility, and should therefore only be used by the operating system.

* Allocation of "pedal"-driven OLAY-data objects to be used for compiler- and link-editor dictionaries, editor texts and similar things. In other words: "software paging" for data.
4.4. Basic space generators in RUNSYS, the MEMLINK-formalism.

The allocation mechanisms described in the two previous sections are all implemented through a basic primitive for space allocation:

Memory is regarded as one big pool of raw storage, from which segments (MEMLINKs) of variable length are allocated.

A segment may contain:

* Nothing, it's free.
* A data block (statically allocated data, a classic COMMON block).
* A user's data stack.
* A special system-controlled heap object (e.g. an I0-buffer).
* A contiguous data area to be used for ordinary heap objects.
* A (block of) SAVE routine(s).
* A block of OLAY-code descriptors.
* A block of OLAY-data descriptors.
* An OLAY-code routine.
* An OLAY-data object.

As seen from the basic operating system, all resident segments are allocated as system-controlled SAVE segments, which may be used for data stacks, OLAY-descriptors or whatever.

The MARY compiler will emit machine-instructions in form of code segments of varying size. A segment represents an ordinary MARY routine or main program. The user may indicate whether the code should be "SAVE" or "OLAY". SAVE routines should only be used by the operating system. "OLAY" is assumed when the allocation specification is omitted.

All accesses to OLAY segments (code or data) go via their segment descriptors, being SAVE for obvious reasons. OLAY-code and OLAY-data are treated more thoroughly in secs. 4.5. - 4.6.
As an optimization, the loader will assemble OLAY-descriptors and SAVE-code segments from user programs in larger blocks to reduce storage fragmentation.

4.4.1. MEMLINK organization.

The TYPE belongs to the set MEMTYPE and may be:
{FREEMEM, SAVEMEM, OLAYCODEMEM, OLAYDATAMEM, or some very special system types.}

The entire physical memory (e.g. 32K) is organized as:

Initially, there exists one big FREEMEM-segment. Later, a rather fragmented situation is dominant.
4.4.2. Allocation and deallocation of MEMLINKs.

Allocation.

When a request for space is made, a first-fit search for a big-enough FREEMEM segment is initiated. It seems (see sec. 4.8) that we get less (external) fragmentation if we start searching from the same place every time. Thus a dummy SAVE segment of effective length zero is used as a reference point. Such a dummy segment is also located across the "32 K-jump" in a 16-bit address space. The latter limits the maximum segment size to 32 K on machines with 48/64 K.

If the located free MEMLINK-candidate is too large, it is split into two parts unless the remainder is less than MINSPACE (= 16 words, incl. MEMLINK-head).

If the current task tries to exceed its SAVELIMIT, being decremented if the segment type is SAVEMEM or OLAYDATAMEM; or if there is global shortage of space, we may be in trouble. See sec. 4.7 on garbage collection and memory deadlocks.

Deallocation.

The actual MEMLINK-segment will be concatenated with possible FREEMEM neighbours and the task's SAVELIMIT properly incremented. An error test prohibits deallocation of already free'd segments.

4.4.3. How to use the MEMLINK facility.

Two routines, GETSPACE and FORGETSPACE, callable as:

PROC SAVE RECURS (INT SIZE, MEMTYPE TYPE) REF MEMLINK
GETSPACE=EXTERNAL;

PROC SAVE RECURS (REF MEMLINK SPACE) FORGETSPACE=EXTERNAL;

carry out allocation and deallocation of MEMLINKs.

These routines are, however, hardly suited for practical use in this form. Some interface macros - SGEN, SGENROW, SDEGEN and SDEGENROW - have been defined:
Ex. Using space allocation macros (in system routines).

```plaintext
MODE BUFFER = [100] BITS;
REF BUFFER B = SGEN (BUFFER);
INT N:=ININT();
ROW PACK CHAR T = SGENROW (PACK CHAR, N); % Allocating a BUFFER-object.

-----
SDEGEN (BUFFER, B);
SDEGENROW (PACK CHAR, T); % Deallocating.
% Analogously.
```

A memory snapshot after the initialization of B and T:

Stack data:  

```
[ Stack data: ]
```

MEMLINK data:

```
[ MEMLINK data: ]
```

Please consult the program listing for definitions of the above macros (Appendix D, page 6, lines 186-204).

Some auxiliary MEMLINK routines also exist; e.g. for changing TYPE and/or OWNER of a segment, causing possible adjustments of the actual tasks' SAVELIMITs.

**Note 1:** This primitive heap mechanism is potentially UNSAFE, as we may continue to use a heap pointer (e.g. B) after the MEMLINK-object has been deallocated! Thus MEMLINK routines are only (directly) accessible by the system. Normal programs must use the LOC generator for stack allocation or the forthcoming automatic heap generator.

**Note 2:** The GETSPACE routine will always zero-fill newly allocated SAVEMEM-objects, so that the initial values of pointers, and similar "sensitive" entities, will be NIL.
4.5. Implementation of OLAY-code.

4.5.1. The virtual code problem.

When designing the booted MARY compiler, it became quite clear that we needed a mechanism for virtual code memory, as long as we insisted on a one-pass compiler.

That is, only the set of routines being currently "used", ought to reside in memory. Analogy: Only the set of data incarnations for the actually called routines have to be allocated on the stack.

We have chosen to implement virtual memory for code in form of software demand segmentation. There are two reasons for this:

1) Actual hardware can not be trusted to support paging or other delights.
2) Segmentation is a well-known technique (Burroughs 5000-6000 series, HP 3300, project SUE), easily implementable in software.

Code segmentation also utilizes the fact that a MARY OLAY routine can be placed anywhere in memory without fixup (self-relocation), and it can be overwritten without requiring write-out on disk (reentrancy).

See discussion in sec. 4.8.

4.5.2. Practical implementation of OLAY-code (initial version).

4.5.2.1. General set-up.

An OLAY-code segment is accessed via its descriptor:
OLAY-CODE DESCRIPTOR:

MEMADDR
DISKADDR

OLAY-CODE SEGMENT:

Either:

MEMLINK-head.

OLAY-code 'segment, prepared by the compiler/loader.

OR:

System routine:

MAKEPRESENT routine, fetching the relevant segment.

<data size>, occupying the first cell in all RECURS routines, is the size of stack incarnations of this very routine. Needed in case of PROC variables.

Thus, we always jump indirectly via the descriptor when we want to execute an OLAY routine. If the routine happens to reside in memory, we end up inside the desired routine. Otherwise, the memory address in the descriptor has been replaced by the address of a SAVE routine called MAKEPRESENT, which will arrange allocation, initialization and calling of the requested OLAY routine.

OLAY-code segments (e.g. file system routines) may be shared between several tasks. Every TASK-object contains a current code segment descriptor pointer. This attribute behaves like a hardware register (cf. the stack descriptor), and is saved and unsaved from the global system-variable CURRENTCODESEGDESCR on task change.

Upon entering OLAY routines, CURRENTCODESEGDESCR is updated to represent the new routine and a hardware register (XREG in our case) contains a relative address, OFFSET, in the OLAY routine. The first executable instruction in OLAY routines is RADD X,P; causing OFFSET to be added to the program counter. On normal entry of a routine, OFFSET is 0. On returning to a calling routine, OFFSET is the difference from segment start to the actual calling place.
All internal jumps and references to local constant cells in OLAY-code segments must be code-relative. (This is very easy to implement on the NORD-SM's, by using standard program-relative addressing and by implementing "long" jumps through \{RADD <register>,P\}-instructions.) Furthermore, no "bad" GOTOs to global labels are allowed from MARY routines. Hence, changes of code segments can only occur in routine calls and exits.

All references to an OLAY-code segment are relative to its segment descriptor. A general program address in OLAY-code will be represented by a tuple:

\[(\text{ref-to-segment descriptor}, \text{offset from segment start})\].

4.5.2.2. Implementation details.

* OLAY-code descriptors are made as small as possible because they are memory-resident. On the NORD-SMs their size is two 16-bit words.

OLAY-code descriptor:

\begin{center}
\begin{tabular}{|l|}
\hline
Memory address of code \\
Disk sector address of code \\
\hline
\end{tabular}
\end{center}

* The segment length is not directly accessible. On disk, it is the first data cell in the first disk sector containing the segment. Possible vacant cells at the end of a sector are not used; a segment always starts at the beginning of a sector:

\begin{center}
\begin{tabular}{l}
An OLAY-code segment \\
Disk sector \text{0} \\
\hline
Another OLAY-code segment \\
Disk sector \text{1} \\
\hline
\end{tabular}
\end{center}

This allocation method yields a 60% disk utilization for OLAY-code on SM4 (176 words sector length).
Note that the first sector must be read before the remaining ones, in order to know how much space to allocate for the segment. The median segment length is about 120 words (see sec. 4.7), which is less than the physical sector length. I.e. optimization/overlapping of buffer space for the last half-filled sector is hardly worthwhile.

* SAVE routines are treated exactly as before (SAVE calling SAVE, SAVE returning to SAVE or OLAY, etc.). Thus, the same compiled SAVE routine can be run together with OLAY routines; or separately in stand-alone programs - without extra run-time overhead.

* The interrupt-handling and task-switching mechanism in RUNSYS is made as simple as possible, i.e. only a plain memory address (current FREG-value) is needed to specify the restart address of an interrupted OLAY or SAVE routine. This implies that the current OLAY-code segment of the following tasks must not be deleted:

  . the RUNNING task,
  . all tasks in the READYQ,
  . all tasks that have generated untreated software interrupts. (We need the offsets from the WAIT-instructions involved. The interrupt may also be a REQUEST(IDLE), which causes RUNNING to be put straight back into the READYQ.)

The code segments of the above listed tasks may in fact be regarded as a very minimal system working set.

In this way, it is likely that there will exist at least one task to receive CPU, when others are waiting on the disk after segment faults.

Let us make the restarting of "OLAY tasks" quite clear:

a) If a task rotates in the READYQ (by executing REQUEST(IDLE) or because of clock intervention), its current OLAY-code segment is never deallocated.
b) If it hangs on a semaphore and then is put back into the READYQ, it will wake up inside ACQUIRE being SAVE. Thus "dormant" tasks never occupy space for OLAY-code.

c) Externally triggered tasks (i.e. drivers) will/can never execute OLAY-code.

* Alternative garbage collect strategies will be discussed in sec. 4.7.
A few comments only:
If we want to compact OLAY-code segments during garbage collects, a trivial updating of corresponding PREG-values in TASK-saveareas must be performed; cf. point a) above.

* OLAY routines must be RECURSive, and they can only call other reentrant routines; i.e. SAVE or OLAY RECURS-routines, or (small) SAVE routines that only use the registers. In the latter case we don't update CURRENTCODESEGDESCR, so we're sure the OLAY-code segment doesn't disappear while we're inside the small SAVE routine. If we choose to compact OLAY-code segments, then LREG (the hardware return-address-register) instead of PREG, may require updating in TASK-saveareas.

* Layout of stack incarnations of RECURS routines:

For SAVE routines:

\[
\begin{array}{c}
\text{STATIC LINK} \\
\text{DYNAMIC LINK} \\
\text{RETURN ADDR} \\
\text{Parms and local data.}
\end{array}
\]

For OLAY routines:

\[
\begin{array}{c}
\text{STATIC LINK} \\
\text{DYNAMIC LINK} \\
\text{SEGDESCR-pointer} \\
\text{RETURN OFFSET} \\
\text{Parms and local data.}
\end{array}
\]

Driver Absolute memory address.

Driver OLAY-code tuple address.

STATIC and DYNAMIC LINKs ought to be well-known from ALGOL60-implementations.
* "Freezing" of interrupts during entries and exits of OLAY routines is not needed. Only the MAKEPRESENT-routine does some "raw" synchronizing, during updates of a segment descriptor after segment fault.

Further details on implementation of OLAY-code (e.g. how OLAY routines call SAVE routines) are given in [Conradi, 1974b].


4.6.1. The virtual data problem.

Hardware is neither supposed to ease automatic detection of data segment faults; like giving interrupts on chasing empty pointers/descriptors. It is not possible (at least for the moment) to design efficient software mechanisms to do the job, because of the randomness of data accesses.

When (or if) COERCION declarations are implemented in MARY, perhaps we could define a special DEREF-coercion for "HEAP-modes", to cause necessary actions to be taken if an OLAY-data descriptor were empty.

The fundamental problem is to implement the addressing-, allocation-, initialization- and accessing-operation in one atomic operation. If we separate the segment-checking step from the accessing step, how long should we "lock" the segment after it has been checked?

For the time being, we must rely on a pedal-driven OLAY-facility for data. That is, we must always indicate explicitly that a data segment is going to be used, before we can use it:

Ex. GETSEG(DESCR); % Fetching and locking.
DESCR.VALUE=....; % Usage.
This is an extremely UNSAFE method and only available for system routines. It is used for compiler- and loader dictionaries, and for editor texts. Garbage collect strategies and empirical results for OLAY-data will be described in sec. 4.7.

4.6.2. Practical implementation of OLAY-data.

4.6.2.1. General set-up.

Consider the situation that we have a large collection of rather small OLAY-data objects or segments (with few inter-object pointers), which are allocated in a "virtual" array on mass storage:

```
Virtual array:
  First virtual data object, with index 1.
  Space
  Actual object with index J.
    CURRENTADDR, address of last recently referenced disk sector.
      Disk sector_1, second OLAY-data page.
    Disk sector_0, first OLAY-data page.
    NEXTINDEX, next available location.
```

We can, of course, not afford to have one data object per disk sector. Furthermore, data objects are never allocated across sector boundaries. This limits the maximum object size to one disk sector.

The disk sectors don't have to be contiguous. On the other hand, they should not be scattered to much, as this probably will increase the access time.
A data object is identified by its relative position, its index, in the virtual array. When we want to access an object with index J, the system evaluates a corresponding disk sector address, checks whether the disk sector (OLAY-data page) is in memory or not, allocates and inputs the page if necessary, and builds a real physical pointer of prescribed mode to the OLAY-data object on this page. The last referenced page will always be "locked" i.e. memory-resident, so this is OK.

A NEXTINDEX-variable (see above figure) facilitates a primitive stack-space generator for virtual data objects.

4.6.2.2. Data structures.

We will need some descriptor tables in the user's data area to administrate OLAY-data pages in memory.

Tentative declarations for SM4:

```
DEFINE
   TARGET(NORD,SM)=INNORD NORD OUTNORD INSM SM OUTSM $,
   DISKSECTORSIZE=TARGET(175,256) $;

MODE
   SECTORADDRESS=SET(1 TO $495),  %"1" MARKS AN EMPTY ADDRESS
   PAGENUMBER=SET(1 TO 2**10-1),  %"SIMILARLY"
   FILEINFO,
   USERFILE=REF VAL FILEINFO,
   OLAYPAGETABLEHEAD,

Page-head mode -

   OLAYPAGEHEAD=(:REF OLAYPAGETABLEHEAD TABLEHEAD,
                  PACK SET(0 TO 2**10-1) PAGERELADDR,
                  PACK SET(0 TO 2**5-1) NLOCKS,
                  PACK BOOL WRITTEN,
                  INT FREQ),  %"USAGE COUNT"

Page mode -

   OLAYPAGE=(:OLAYPAGEHEAD HEAD,
              (:DISKSECTORSIZE) INT DATA),

   OLAYPAGEDESC=(:REF OLAYPAGE PAGE,
                  SECTORADDRESS SECTADDR),

Page-table-head mode -

   OLAYPAGETABLEHEAD=(:USERFILE OLAYDATAFILE,
                      SECTORADDRESS CURRENTADDR,
                      PAGENUMBER TOPPAGE,
                      INT NEXTINDEX,
                      ROW (J) OLAYPAGEDESC PAGETABLE),

OLAY-data segment descriptor mode -

   OLAYDATADESC(M)=(:REF OLAYPAGETABLEHEAD TABLEHEAD,
                     INT INDEX,
                     REF M VALUE);
```
Ex. Assume that we have declared:

```
USERFILE DATAFILE =ALLOC('F',200);  % (Somewhat simplified).
[200] OLAYPAGEDESCRIPTABLE := NIL;  % Basic page table.
OLAYPAGETABLEHEAD TABLEHEAD := NIL;  % Page table head.
INITOLAYDATATABLE(TABLEHEAD,DATAFILE,TABLE);
-----
```

```
MODE DICTENTRY=[[12] CHAR NAME, ----];  % Mode of OLAY-data object.

OLAYDATADESCRIPTABLE DICTENTRY DESCRIPTABLE := (TABLEHEAD, 400, NIL);  % OLAY-data
-----

GETSEG(DESCRIPTOR);
OUTTEXT(DESCRIPTOR, VALUE, NAME);
}  % Accessing the object.
```

Snapshot of relevant data structures when accessing above
virtual DICTENTRY-object with INDEX 400. This index corresponds to
relative address 47 in relative sector 2, which e.g. is mapped to
physical sector 52.

OLAY-data page in memory:

- MEMLINK-head of
type OLAYDATAMEM.
- OLAYPAGEHEAD-
object.
- Contents of a SM-
disk sector (176
words) = the real
page.
- Actual OLAY-data
segment of mode
DICTENTRY.
Comments.

A missing page has an empty reference (NIL) in the page table.

The most recently used page, whose sector address is that of CURRENTADDR, cannot be deleted. Pages with NLOCKS>0 (see later) are also locked. Remaining "deletable" pages may require write-out on disk, if their WRITTEN-flag is TRUE, before they can be deallocated.

4.6.2.3. Utility macros. Usage.

Five macros are available to systems programmers for OLAY-data management:

* ALLOCSEG.

Ex. Allocation of a new DICTENTRY-object.

ALLOCSEG(OPERANDSIZE DICTENTRY, DESCR);  % Updates NEXTINDEX
  % and CURRENTADDR.
  % If the corresponding page is not
  % allocated, it will
  % be. The WRITTEN-
  % flag is set to TRUE.

DESCR.VALUE is now pointing into an OLAY-data page at a
data object of mode DICTENTRY. DESCR_INDEX contains its
its relative virtual address/index.
"Zero" is never used as an index value and may therefore
flag an empty index.

* GETSEG and PUTSEG.

When we want to access this object later, we write:

GETSEG(DESCR);  % Updates CURRENTADDR and DESCR.VALUE. If
  % the corresponding page is not in memory,
  % it will be allocated and initialized.

or:

PUTSEG(DESCR);  % As for GETSEG, but sets the WRITTEN-
  % flag to TRUE.
Note: Systems programmers must remember to explicitly insert ALLOCSEG-, GETSEG- and PUTSEG-calls before accessing virtual data objects. The corresponding pages will thereby be checked and temporarily locked, since CURRENTADDR is updated.

* LOCKSEG and UNLOCKSEG.

Sometimes, we may want to "freeze" more than the last referenced CURRENTADDR-page.

Some locking-control macros are available:

LOCKSEG (DESCR); % Incrementing the NLOCKS-attribute
      % of the page by 1.
      % (NLOCKS is initially 0.)

UNLOCKSEG (DESCR); % Decrementing the NLOCKS-attribute
      by 1.

This facility may, for instance, be used to ensure that dictionary entries for the most common MARY symbols don't get swapped during MARY compilations.

A case example.

The MARY compiler will have a large amount of its dictionary allocated in this way. The internal representation of modes contains a lot of mutual pointers, so mode descriptions are not suited for virtual storage. Virtual data objects are mostly symbol table entries, which contain pointers to misc. memory-resident objects.

Such dictionary entries are linked in hash chains, and the internal hash links will be indices in the virtual array. Pre-defined MARY symbols should be sorted after frequency of usage to reduce look-up times on virtual storage. The associated hash table should be rather large (≥128 words); for the same reason.

As the compiler's dictionary behaves like a stack, we will need a special routine, CUTBACKOLAYDATATABLE (<tablehead>,<new NEXTINDEX>) to forget the top-most objects.
Note that several simultaneous MARY compilations may share a common dictionary of standard MARY symbols. If a user program tries to "FREE" such a symbol, the compiler must take care not to delete its dictionary entry to other users.

4.7 Memory shortage and -deadlocks. Virtual memory performance.

4.7.1 On memory shortage and memory deadlocks.

If there is global shortage of space, a garbage collect of OLAY segments must be initiated. If this doesn't free enough memory space to satisfy the actual space request, we might have entered a deadlock situation on primary storage (see below).

In RUNSYS, memory space occupied by SAVE segments and some selected OLAY segments is not immediately re-usable. As mentioned, the global variable AVAILABLESAVEMEM and the task attribute SAVELIMIT have therefore been introduced to avoid space saturation.

A fixed amount of "OLAY-code space" (p.t. 1000 words) is also subtracted from AVAILABLESAVEMEM, when forking a new task, to make sure there is always room for requested OLAY routines.

(The code buffer in the MARY compiler can only hold ≤ 1000 words, so we're guaranteed that OLAY-code segments don't grow too large.)

Thus, in principle we should have a safe system: memory resources are pre-allocated upon task creation. In practice, however, we may experience difficulties due to memory fragmentation combined with non-compactability of resident/locked memory segments. That is, there may be enough free space, but it is not available as a (large enough) contiguous area. Such memory deadlocks may threaten safe running of ordinary user tasks, as well as the entire operating system. According to [McKeag, 1971], ≈ 10% of all system failures on some Burroughs 5500s (about 1 per week) were caused by such deadlocks. RUNSYS data on this is given in sec. 4.7.5.8.
Even if we don't run into abortive memory deadlocks, the phenomenon of *trashing* remains unsolved. Remedies against this are usually an upper limit on the number of active tasks and clever replacement algorithms for OLAY segments. See secs. 4.7.3 - 4.7.6.

4.7.2 General comments on factors, that influence the performance of virtual memory systems

There are a few general, well-established truths about VM-systems (particularly the more ambitious ones), which explain why many of them have not fulfilled their implementor's great expectations, in spite of substantial technical finesse:

- All components of a VM-system (dispatcher, jobcontroller, replacement algorithms, organization of primary and secondary storage, compilers, loader, disk/drum access algorithms) must be closely coordinated in order to achieve an efficient system.

- Optimizing garbage collect/replacement algorithms may be intolerably time- and space-consuming. That is, we may have to rely on a FIFO-, rather than a LRU-strategy for page/segment replacements.

- Often, the most effective way to improve system performance (i.e. to reduce the amount of trashing) is to buy more primary storage or a faster swap-medium.

- Any user/system program can be (manually) "tuned" towards a significantly better performance; e.g. through a more functional organization of subroutines and data. (IBM OS-manuals are full of advice on how to write programs under their VM-systems.) Re-design like this may, however, be rather costly and only worth the effort for systems programs.

These rather discouraging statements apply to both paging- and segmentation systems.
There is, however, one point I would like to stress, which is genuine for RUNSYS-like segmentation systems:

Since the olay-system is completely software-driven, it is fairly easy to parameterize, modify and monitor it.

With these general comments in mind, we shall have a closer look on garbage collect strategies and virtual memory performance in RUNSYS.

Most of the following data on the segmentation system origin from measurements of the stand-alone MARY-MARY compiler, run under MINIRUNSYS (see sec. 9.3). Only sparse hard data are available on full RUNSYS and on OLAY-data.

4.7.3 Empirical data on segment sizes

Note: Segment sizes in this section refer to incoming space requests. The real segments may be slightly larger (4-5 words (?) of internal fragmentation), if their unused tails are less than MINSPACE (=16) words.

4.7.3.1 Distribution of SAVE segment sizes

The data collected reflect the total number of allocated SAVE segments (410) in a RUNSYS test, where a small 10-lines MARY-program was edited, compiled and executed.

Note that an OLAY-data page is counted as a SAVE segment of length (4+3+176)=183 words on the SM4.

The total number of allocated OLAY-code segments in the same test was =1500.
Table 4.1  Histogram of allocated SAVE segments.
N = 410 segments.
Mean ≈ 220 words.
Median ≈ 25 words.

Technical notes

The size-unit is 32 words (as the machine doesn’t have a hardware divide instruction).

The leftmost size-column, with size-index 32, represents segment sizes $\in [0,31]$; And so on upwards.

The rightmost, with size-index 1056, represents segment sizes $\in [1024, \infty]$. 
Remarks

The distribution is rather uneven. E.g. it might be considered to have a global pool of TTY-output buffers (cf. TTYBUFFERPOOL in sec. 5.8.2).

If the compiler's dictionary hadn't been segmented, we would have registered one large 7K-segment, instead of 14 512-words segments.

4.7.3.2 Distribution of OLAY-code segment sizes

Table 4.2 Histogram of all OLAY-code segments in a stripped RUNSYS-version (without the MARY-MARY compiler). Static counts, i.e. merely counting the olay-file segments.
N = 209 segments. Total code size = 31 000 words.
Mean = 149 words. Median ≈ 95 words.
Olay-file = 294 sectors (51% utilization).
Table 4.3.1  Histogram of all OLAY-code segments in MARY-MARY stand-alone compiler.
N = 652 segments. Total code size = 118 900 wo;
Mean = 182 words. Median ≈ 110 words.
Olay-file = 1052 sectors (62% utilization).

Remark: The dotted, hatched histogram is table 4.3.2 below, being a subset of table 4.3.1.

Table 4.3.2  Histogram of OLAY-code segments in MARY-MARY stand-alone compiler.
Static usage counts, obtained by compiling a standard, 45-lines MARY test-program.
That is, which part of the compiler got activated?
N = 403 (out of 652, cf. above).
Total activated code size = 87 770 words
(out of 118 900).
Mean = 218 words. Median ≈ 130 words.
Table 4.3.3

Histogram of OLAY-code segments in MARY-MARY stand-alone compiler.
Allocation counts; obtained as for table 4.3.2, given a core store of 48 K & LRU-replacement policy.

That is, how many times were segments (re)allocated/fetched from the olay-file?

N = 1286 segment faults, i.e. 883 excessive ones.
Total fetched code size = 246 000 words.
Mean = 191 words. Median ≈ 110 words.

Table 4.3.2 (static usage counts, a subset of table 4.3.3) is indicated by dotted, hatched columns.

Remark:
1286 segment faults during 24 000 OLAY proc calls (see next table), gives a fault rate of \( \frac{1}{40} \) per proc change.
If segment/proc changes had been completely randomized, the fault rate would have been \( \approx \frac{2}{3} \).
Table 4.3.4  Histogram of OLAY-code segments in MARY-MARY stand-alone compiler.

Dynamic usage counts, obtained as for table 4.3.3.

That is, how many times were corresponding routines referenced.

\( N = 48664 \) (i.e. \( \approx 24000 \) OLAY proc calls).

Table 4.3.3 (allocation counts, a subset of table 4.3.4) is indicated by dotted, hatched columns. It is scaled by a factor 5 to illustrate the relative proportions.
Table 4.3.5 contains data quite similar to those in table 4.3.4, but more selective:

Table 4.3.5  *Usage and allocation counts for individual OLAY routines in MARY-MARY stand-alone compiler.*

Obtained as for tables 4.3.2 and 4.3.4. Only 44.2 K available core store, because of 5 K extra, resident tracing stuff. LRU-replacement policy gives 2142 segment faults (856 more than if 49.2 K memory).

The table is sorted according to usage counts.

<table>
<thead>
<tr>
<th>Routine no.</th>
<th>Routine name</th>
<th>Dynamic usage counts</th>
<th>Cum. dyn. usage counts</th>
<th>(Usage Alloc) Ratio</th>
<th>Allocation counts</th>
<th>Cum. alloc. counts</th>
<th>Segment sizes</th>
<th>Cum. seg. sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GETCARDLINE</td>
<td>3743</td>
<td>3743</td>
<td>220</td>
<td>17</td>
<td>17</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>2</td>
<td>DICTGENERAPE</td>
<td>3503</td>
<td>7246</td>
<td>1752</td>
<td>2</td>
<td>19</td>
<td>153</td>
<td>228</td>
</tr>
<tr>
<td>3</td>
<td>DICTROGEN</td>
<td>2658</td>
<td>9904</td>
<td>204</td>
<td>13</td>
<td>32</td>
<td>492</td>
<td>720</td>
</tr>
<tr>
<td>4</td>
<td>NEXITIVITY</td>
<td>1838</td>
<td>11742</td>
<td>230</td>
<td>8</td>
<td>40</td>
<td>799</td>
<td>1519</td>
</tr>
<tr>
<td>5</td>
<td>GIVEREG. (?)</td>
<td>1322</td>
<td>13064</td>
<td>342</td>
<td>4</td>
<td>44</td>
<td>183</td>
<td>1702</td>
</tr>
<tr>
<td>6-10</td>
<td>5 next ones</td>
<td>4670</td>
<td>17734</td>
<td>104</td>
<td>45</td>
<td>99</td>
<td>811</td>
<td>2513</td>
</tr>
<tr>
<td>11-20</td>
<td>10 &quot;</td>
<td>7376</td>
<td>25110</td>
<td>64</td>
<td>116</td>
<td>215</td>
<td>2219</td>
<td>4734</td>
</tr>
<tr>
<td>21-50</td>
<td>30 &quot;</td>
<td>10949</td>
<td>36059</td>
<td>39</td>
<td>284</td>
<td>497</td>
<td>5387</td>
<td>10127</td>
</tr>
<tr>
<td>51-100</td>
<td>50 &quot;</td>
<td>7042</td>
<td>43101</td>
<td>12</td>
<td>573</td>
<td>1070</td>
<td>12447</td>
<td>22574</td>
</tr>
<tr>
<td>101-200</td>
<td>100 &quot;</td>
<td>3987</td>
<td>47088</td>
<td>6.3</td>
<td>635</td>
<td>1705</td>
<td>29003</td>
<td>51577</td>
</tr>
<tr>
<td>201-403</td>
<td>203 &quot;</td>
<td>1576</td>
<td>48664</td>
<td>3.6</td>
<td>437</td>
<td>2142</td>
<td>36193</td>
<td>87770</td>
</tr>
<tr>
<td>1-403</td>
<td>Σ</td>
<td>-</td>
<td>48664</td>
<td>23</td>
<td>-</td>
<td>2142</td>
<td>-</td>
<td>87770</td>
</tr>
</tbody>
</table>

The allocation counts in table 4.3.3 and 4.3.5 really belong to sec. 4.7.6. (performance data), but they may be more easily analyzed here.

Remark:

The compiler contains many recursive calls; i.e. locps in the "reference string". The ideal situation is probably to let the most heavily used routines be resident all the time, and let the more rarely used ones compete about the remaining space. Cf. usage/fetch ratios.
4.7.3.3 External fragmentation

This has been acceptable.

The amount of free, unusable space upon garbage collects has typically been ≈ 3 - 8%. This is primarily due to the "deletability" of OLAY segments, enabling us to free relatively large contiguous areas; if required.

4.7.4 Garbage collect strategies for OLAY segments

Given the total set of "deleteable" OLAY segments; which ones should actually be deallocated/replaced: All of them, the "Least-Recently-Used" one(s), or some other subset?

A complicating factor is that OLAY-code segments are variably-sized, so that more than one segment may have to be deleted.

In my opinion, replacement algorithms should be rather straightforward, easy to parameterize and not excessively time- or space-consuming. The delicate trade-off between garbage collect efforts and non-optimal replacement choices will be elaborated in sec. 4.7.7.

4.7.4.1 Replacement policies for OLAY-code

As described in sec. 4.5.2.2, OLAY-code segments for RUNNING- and READYQ-tasks must be temporarily locked; all others are potentially deletable. Furthermore, they are reentrant and self-relocatable.

Six algorithms have been tested:

A) The first one is a plagiarism of Burroughs' initial solution for their 5500 system: When we discover shortage of free space, we wipe out all OLAY segments that may be deleted, whereafter we start refilling the store with segments from disk on demand. According to Mark Rain in private conversations, this wipeout-algorithm was only 15% less efficient than the best replacement algorithm they managed to implement.
B) The second algorithm is an elaboration of the first one:

A **frequency** counter, FREQ, is associated with each (resident) OLAY-code **segment**. It is incremented by 1 on "using" a segment, i.e. by calling the corresponding OLAY routine or returning to it from another routine. The initial value of this counter is set to OLAYCODEPOOLSIZE=(the current number of resident OLAY-code segments).

On space shortage, all segment counters are decremented by DECR=OLAYCODEPOOLSIZE//16 (see sec. 4.7.5.3 on the "balancing" point). Deletable OLAY-code segments with counter-values \( \leq 0 \) are then deallocated.

If a space request, L, still cannot be met, a **hole** H of adjacent free or deletable segments will be located and converted into a large, free segment. H is defined by:

\[
\min(\sum_{i \in H} FREQ_i \cdot \text{length}_i), \text{ where } \sum_{i \in H} \text{length}_i \geq L.
\]

Thus the hole with the **smallest**, weighted size is selected.

**Note:** FREEMEM-segments are given zero-valued counters/weights. OLAY-data pages already possess a usage counter (FREQ).

This algorithm will keep track of much-used OLAY segments and protect them from being deleted. FREQ is a kind of **dynamic usage** counter (recording accumulated usage), which steadily gets decremented to obtain a Least-Recently-Used (LRU) effect.

The algorithm is appealing, as updating overhead on subroutine entries and -exits requires only 2.5=10 instructions on SM4 (\( \approx 30\mu s \)).

C) The third algorithm is a FIFO-algorithm. Two link-fields are added to all resident OLAY-code segments, to allow a doubly linked list of segments. The list is updated on segment allocation and -deallocation.

The reason why we need a (doubly) **linked** list of segments, is that we may have to delete more than one segment; in which case a circulating-pointer-algorithm will be too coarse and hence inefficient.
Upon garbage collects, segments are assigned weights/frequencies, according to their relative position in the segment list. Then we proceed as under Algorithm B.

A linear weight function is presently used:

\[ \text{FREQ}_i = i, \text{ i.e. oldest segment will receive weight 1.} \]

D) The fourth algorithm is a true Least-Recently-Used algorithm. The basic data structures and garbage collect strategies are as under Algorithm C. The only difference is that we also update the segment list upon (re-) entering OLAY routines. This updating must, regrettably, be carried out with interrupts disabled. About 300 extra \( \mu \)s are needed per proc:

E) The fifth algorithm is a cheaper LRU-variant: A global LRUTOP-counter (initially 100) is incremented by 1 and stored into the previous FREQ-attribute during OLAY-enter and -exit calls. The FREQ-values and the LRUTOP-variable are decremented by LRUTOP/8 (see sec. 4.7.5.3) in each garbage collect. The running FREQ-values are then used in a delete-and search-strategy, as under Algorithm B.

F) The sixth algorithm is a mixture of B and (C,D or E). A combined weight function is constructed; and we proceed as under Algorithm B.

Empirical results over run-time efficiencies and a primitive working-set model can be found in secs. 4.7.6 - 4.7.7.

4.7.4.2 Replacement policies for OLAY-data

The pages being locked, i.e. the CURRENTADDR-page and the ones with NLOCKS>0 (because of LOCKSEG-calls), cannot be touched. The remaining pages are potentially deletable. They are either "written" or "non-written" ones.
Written, deletable pages require special treatment: An OLAY-data page is local to a task. Only the owner task may copy such a page out on disk (to get rid of the write-status, so the page can be deallocated). No other task can perform such writing, as the original owner task e.g. may get killed during the write-operation, thus deallocating its private OLAY-data pages. Such writing is also rather time-consuming and therefore undesirable in global garbage collects, performed by alien tasks.

The actual garbage collect algorithms resemble those for OLAY-code: We may either wipe out all irrelevant OLAY-data pages; or only a subset, determined by recent usage. The resident pages possess counter- or link-fields, being updated on GETSEG- and PUTSEG-calls, to provide such information.

Summing it up:
In global garbage collects, only non-written pages can be deleted. They are treated more or less like OLAY-code segments.

Local garbage collects are then required to delete the written ones, and this should be done before the owner task's SAVELIMIT gets negative. Extra tests in the page-checking routines see to this.

4.7.5 Initial performance of the segmentation system - What did we learn?

4.7.5.1 The starting point

The initial design of data structures has been described in sec. 4.7.2, and we were using Algorithm A, the wipeout algorithm.

After some time we discovered that the MARY-MARY compiler, being run in a stand-alone MINIRUNSYN-environment, performed real bad. With s=the ratio (available core storage/virtual code space) \( \approx 0.10 \), it compiled \( \approx 12 \) instructions per minute, while the NUALGOL-MARY compiler on the U1108 compiled about 60 per second.
That is, 300 times slower! (However, the U1108-figure is net CPU-time. The total time spent may be 2 - 5(?) times higher.)

The relative capability and speed of the respective CPUs account for a factor 5, which leaves us a factor ≈60. I.e. the extrapolated CPU-utilization on a 32K SM4-machine with a 50 ms-disk were ≤2%!

There are, of course, additional factors which must be considered, but the general, qualitative picture was undeniable: The system trashed beyond our worst expectations; and the success of the entire project was threatened.

If we had done some coarse modelling on the system in the first place, the above figures would not have come as a shock to us. Actually, many implementors of VM-systems have found themselves in a similar situation: Virtual memory meant "virtual" savings.

In other words, we had to look for targets of improvement. A re-design process like this seems to be typical for real-world operating systems; parts of a system must often be re-written several times to yield satisfactory performance. Luckily, we had the time to carry out this work to some extent.

Some background data on the MARY-MARY compiler:

As of July 1976, the stand-alone version of the compiler initially occupies ≈6 K SAVE-space and 119 K OLAY-code. In addition comes ≈12 - 15 K dynamic SAVE-data for data stack (min. 2 K), dictionary (min. 6 K), code buffer (min. 1 K), etc.. This leaves us ≈12 - 15 K, out of 32.8 K, for segmented execution of a 119 K compiler. - No wonder the system will trash! And note that a single user, executing the compiler, will also cause the full RUNSYS-system to trash.

In the following, I shall comment on implemented modifications in the segmentation system in the period April 1975 - July 1976, included "tuning" of relevant OLAY programs like the compiler.
4.7.5.2 Putting constant SAVE-data into OLAY-code

Ex. Output of a constant text.

PROC SAVE RECURS (ROW VAL CHAR T) OUTTEXT=EXTERNAL;
OUTTEXT('*****');

Here, the text constant '*****' will be allocated as a 5-words
SAVE-data object.

By introducing a macro:

Ex. DEFINE OUTOLAYTEXT(T)=([T LEN] CHAR NAMENAME=T;
OUTTEXT(NAMENAME));$;

OUTOLAYTEXT('*****'); %& New output call.

the constant text will be put into OLAY-code, copied into a
temporary, stack-allocated array NAMENAME and then passed to the
OUTTEXT-routine.

That is, by spending CPU and (transient) OLAY-code space, we
are able to cut down the SAVE-space demand.

The accumulated savings for RUNSYS, incl. the compiler, is
\(\approx 7 \, \text{K} \) - without having to remove vital debug and trace facilities.
As the minimum SAVE-space demand of a typical RUNSYS-configuration
is \(\approx 10 \ldots 12 \, \text{K} \), the net effect is substantial.

4.7.5.3 A hierarchical space allocation strategy. The balancing
point

Without too much pain, initial versions of the replacement
algorithms B-F were implemented and tested.

Although we now spend additional CPU-cycles in updating recent-
usage information (a full LRU-chain updating increases the OLAY-
proc-call overhead from 250 \(\mu\)s to 550 \(\mu\)s), the number of segment
faults went down radically (about 30 \ldots 50\%). However, no signi-
ificant throughput improvement was observed. Why so?
The explanation was very simple, and accidentally discovered after I had put in some extra accounting code in the garbage collect routine: **Space allocation takes time!** Naturally.

Evidently.

Empirical results, supported by theoretical considerations in sec. 4.7.7, indicates that we should **free far more space** than actually **needed** on space shortage, so that we can avoid further garbage collects for some time. The "balancing" **point** seems to be \( \approx 3 - 7\% \) of the storage being available for OLAY segments.

Furthermore, it is beneficial first to **wipe out** a "least-used" 5% - **fractile** in a fast, initial search (by scanning the least-used tail of the existing LRU- or FIFO-list of OLAY-code segments); **before** a more ambitious, time-consuming hole-identification procedure is attempted.

Some data:

The initial fractile-wipeout-freeing consumes \( \approx 0.4 \) ms per OLAY segment freed, while an exhaustive hole-search takes \( \approx 0.55 \) ms per resident OLAY segment.

A typical number of OLAY-code segments is \( \approx 100 \). I.e. a fractile-wipeout freeing costs \( \approx 2.5 \) ms and a hole-search \( \approx 55 \) ms. Empirically, about 75% of the garbage collects free enough space in the initial scan, giving an average of \( \approx (2.5 \cdot 4 + 55)/4 \) ms = 16 ms.

As mentioned, we only have to perform a garbage collect per \( \approx 100 \cdot 0.05 = 5 \) segments. The average garbage collect overhead then becomes \( \approx 16/5 = 3 \) ms. The (disk access time+misc. overhead) constitutes \( \approx 51 \) ms. I.e. **total segment fetch-time**:

Previously, \( \approx (55+51)\text{ms}=106 \) ms.

Now, \( \approx (3+51)\text{ms}=54 \) ms.

Note that these large garbage collect times primarily are caused by variably-sized segments. In a paging system, a hole-search procedure is superfluous.
This "hierarchical" space allocation strategy accounts for the greatest observed improvement in the VM-system until this day.

4.7.5.4 Reduced buffering overhead

Superfluous (dynamic) allocation and self-copying of I0-buffers were discovered.

An average of 5 ms per segment fetch were gained.

4.7.5.5 Lost disk revolutions. Putting the segment size in the descriptor

In the initial system, we had to read in the first OLAY-code sector, to obtain the segment size, before we could allocate a segment. About 1/3 of the (allocated) OLAY-code segments are bigger than on disk sector (176 words). This meant that we were likely to loose one full disk revolution (40 ms), as we probably would not be ready in 2.5 ms, when the second sector did pass the disk head.

We might try to allocate OLAY segments in every second/third disk sector, but this will generally increase the access time. Besides, hardware people have already put logically consecutive sectors in every second physical one on the SM4.

The only sensible solution is to let the segment size be known before the first sector is read, i.e. stored in the descriptor. That is, we allocate first, and then the entire segment can be read in one throw.

Because of space economy and compatibility, it is beneficial to let the size of OLAY-code descriptors remain unchanged (2 words). Thus we must interpret their contents a bit differently, depending on whether the referred OLAY-code segment is memory-resident or not:
The LENGTH-attribute is alternatively in the descriptor or in the segment (which may be slightly larger than needed).

Again, we're charged with extra CPU-cycles on OLAY-enter and -exit routines (≈50 μs per call), but it's worth it. The average segment fetch-time dropped by ≈15 ms.

4.7.5.6 Multiprogrammed space allocation and disk IO

Once the segment size is known on beforehand, we may overlap space allocation/garbage collection with reading of the first segment sector (into a static buffer) in the stand-alone MINIRUNSYS system. That is, we can afford a rather sophisticated space allocation algorithm, as long as it is faster than the average olay-sector access time (≈45 ms). Empirically, the average segment fetch-time sunk by 5 ms.

For full RUNSYS, the overlapping of space allocation and sector fetching is granted automatically by the multiprogramming system. However, the discovery that garbage collects may be rather time-consuming (10 - 70 ms) led to the implementation of a special space administration task, with the best possible priority.
Otherwise the interrupt system might have been turned off unduly long by an arbitrary task during a garbage collect, being initiated and executed by this task. Likewise, if we try to regulate space-administration-"access" via traditional semaphores, a poor-priority task may start a garbage collect, which then proceeds intolerably slowly.

Furthermore, a separate space administration process, e.g. executed by some other CPU, opens the way for a more or less continuous garbage collection behind the scene, so that there always is some free space available on short notice. See also later in sec. 4.)

The average allocation-time increased slightly by this solution (≈3 ms), but the average stack size of most tasks could be reduced by ≈50 words, as they can no longer initiate (and thus "pay" fcr) garbage collects.

4.7.5.7 Reduction of allocation-times for new segments

The first-fit scanning of arbitrary MEMLINK-segments, to locate a big-enough hole, was optimized by linking all FREEMEM-segments into a separate, doubly-linked list.

The time spent per segment is ≈50 µs in both alternatives. Given ≈200 segments (SAVE, OLAY and FREE), with an average of ≈20 FREE ones, the average search time will drop from:

\[ \frac{200}{2} \cdot 50 \, \mu s = 5 \, \text{ms} \]

to:

\[ \frac{20}{2} \cdot 50 \, \mu s = 500 \, \mu s. \]

Not bad!

4.7.5.8 Reduction of memory deadlocks, caused by SAVE segments

Empirically, (small) SAVE segments sooner or later got scattered around so much, that it became difficult to accomodate segments with size ≥5% of the available pool (20 - 50 K). This happened in spite of the fact that we started free-list searching for SAVE segments from the one end of memory, and OLAY-code segments from the other.
Therefore, SAVE segments are now allocated from the lower end of memory, by forcing deallocation of anything deletable on our way, until a big enough hole has been found/created. This "plowing" strategy may sound brutal, but most SAVE segments are very small. I.e. little resident OLAY-space will be lost; or we manage to locate an existing hole with no freeing at all. The main principle is to keep all the (small) SAVE segments at the lower end of memory, so that larger segments can be allocated elsewhere.

Since this "search-delete" strategy was implemented, (SAVE) segments with sizes up to 20% of the total pool size can easily be roomed.

4.7.6 Performance data on the segmentation system

As mentioned, the initial thruput data were utterly disappointing. The following data have been collected by letting the stand-alone MARY-MARY compiler compile our standard MARY test-program on a 64K SM4-machine. The relations between CPU- and segment fetch-times for varying store sizes and garbage collect strategies have been studied.

Some background information (see also p.105):

The compiled MARY test-program consists of 45 lines, generating 135 instructions.

I0-time (card reader, line printer) = 19 s is subtracted from the measured compilation times.

Resident OLAY-code segments are slightly larger than the routines contained, because of system overhead (≈10 words).

Initial SAVE-space demand of stand-alone compiler: 6K. The remaining memory space constitutes the MEMLINK-pool, whereof 14.3 K is occupied by SAVE segments. As mentioned before, these compilations each require ≈24,000 OLAY proc calls, executed by 403 different routines (87.8 K code).

About 100 extra µs per OLAY proc call are presently devoted to tracing purposes. I.e. such a proc call costs (550+100)µ s = 650 µ s.

The 550µs-figure consists of: 250µs (basic) + 300 µs (LRU).

NB: Only 30 µs extra, if FREQ-repl.policy.

Since this is a stand-alone version (space allocation and segment fetching are overlapped), we can afford to free a bit less than ≈5% space per garbage collect. Only the "full" garbs, that follow possibly insufficient "5%-factile" garbs, do degrade system performance.
<table>
<thead>
<tr>
<th>Core-size</th>
<th>Mem-link-save pool size</th>
<th>Replacement algorithm</th>
<th>CPU-time: proc-calls + disk (s)</th>
<th>CPU-time: cache faults (s)</th>
<th>Fetch-time: seg faults (garb.)</th>
<th>Fetch-time: seg faults (garb.) (aver.)</th>
<th>Cache faults</th>
<th>Segments</th>
<th>FREE</th>
<th>OILAY</th>
<th>Total code size (on garb.)</th>
<th>WHEREOF FREED PER Garb.</th>
<th>WHEREOF FREED PER Garb.</th>
<th>WHEREOF FREED PER Garb.</th>
<th>WHEREOF FREED PER Garb.</th>
<th>WHEREOF FREED PER Garb.</th>
<th>WHEREOF FREED PER Garb.</th>
</tr>
</thead>
<tbody>
<tr>
<td>32.8k</td>
<td>14.3k</td>
<td>LRU</td>
<td>0.8k</td>
<td>0.2k</td>
<td>12.5k</td>
<td>12.5k</td>
<td>258k</td>
<td>123k</td>
<td>25k</td>
<td>5k</td>
<td>12.5k (s)</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>26.8k</td>
<td>14.3k</td>
<td>LRU</td>
<td>0.8k</td>
<td>0.2k</td>
<td>12.5k</td>
<td>12.5k</td>
<td>258k</td>
<td>123k</td>
<td>25k</td>
<td>5k</td>
<td>12.5k (s)</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>41.0k</td>
<td>35.0k</td>
<td>LRU</td>
<td>0.8k</td>
<td>0.2k</td>
<td>12.5k</td>
<td>12.5k</td>
<td>258k</td>
<td>123k</td>
<td>25k</td>
<td>5k</td>
<td>12.5k (s)</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>49.2k</td>
<td>43.2k</td>
<td>LRU</td>
<td>0.8k</td>
<td>0.2k</td>
<td>12.5k</td>
<td>12.5k</td>
<td>258k</td>
<td>123k</td>
<td>25k</td>
<td>5k</td>
<td>12.5k (s)</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>65.5k</td>
<td>59.5k</td>
<td>LRU</td>
<td>0.8k</td>
<td>0.2k</td>
<td>12.5k</td>
<td>12.5k</td>
<td>258k</td>
<td>123k</td>
<td>25k</td>
<td>5k</td>
<td>12.5k (s)</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
</tbody>
</table>

Note: Only 4 of the 6 previously described replacement algorithms possess interesting/distinguishable qualities, so results for algorithms F(cheap LRU) and F(combined strategies) are not shown.
Diagram 4.3, showing throughput data from table 4.4.

Min. fetch-time to load 403 segments.

Total CPU-time.

CPU-time, if everything could have been SAVE (see later).

\[
\begin{array}{cccccc}
0.1 & 0.2 & 0.3 & 0.4 & 0.5 & 0.6 \\
32K & 40K & 48K & 64K (core size) & \\
\end{array}
\]

\[
s = \frac{\text{available memory size}}{\text{total needed compiler size}} = \frac{\text{core size} - (6 + 14.3)K \text{SAVE-space}}{87.8K} - \alpha
\]

\[
\alpha = \text{space lost because of unused free space (5 - 10%).}
\]

Remarks.

As shown, LRU is the best one for most core sizes. It is relatively more superior, the larger s becomes.

The fact that FIFO is better than LRU, given 40 K storage, may be because LRU is behaving rather poorly on cyclic segment-reference patterns.
4.7.7 A mathematical model of the VM-system

4.7.7.1 A simple working-set model

An exhaustive statistical model, based on measured or estimated transition probabilities between segments, has not been attempted. The necessary effort is prohibitive, and the results of such analysis will probably only have theoretical/phenomenological value, since an implemented, selective look-ahead analysis is out of the question.

Instead, I shall present a simple variant of the working-set model, introduced by [Denning, 1968]:

The execution of a virtual memory program consists of CPU-execution of some resident code, followed by a waiting period of length $T^*$ upon segment faults - and so on.

The average CFJ-instruction time of compiled MARY-code is $\tau=3.5 \mu s$ (SM4).

The total number of instructions executed is $N$. A fully resident program will then require $N \cdot \tau$ units of CPU-time.

The probability for segment faults per executed instruction is called $p$, being a function of the previous $s$.

Total time $T$ to execute a program:

$$T = N \cdot \tau + s \cdot M \cdot (T'' + t_s) \cdot (N - N_I) \cdot [\tau + p(s) \cdot T^*]$$

$N_I$ = number of instructions executed, until the store has been filled up and equilibrium reached.

$M$ = the average number of different compiler segments, referenced during a compilation. $M \approx 400$.

$M$ constitutes the virtual code space, measured in terms of OLAY-code segments. Average size of resident segments: $\approx 200$ words.

$T^* = T'' + t_s$.

$T''$ = disk access time (for average two sectors) + misc. segment update overhead.

$= T_D + T_s \approx 49 \text{ ms} + 1.5 \text{ ms} = 50.5 \text{ ms}$. 
\[ t_S = \text{space allocation time + average garbage collect time} = t_A + t_G = 0.5 \text{ ms} + t_G. \]

We have that:

\[
\frac{N_I}{N} \approx \frac{s \cdot M}{s \cdot M + (N - N_I) \cdot p(s)} = \frac{\text{number of segment faults, until equilibrium}}{\text{number of segment faults, totally}}.
\]

The garbage collect function can be estimated as:

\[ t_G(f) \approx \frac{c \cdot s \cdot M \cdot n \cdot \frac{1}{s \cdot f \cdot M}}{f} = \frac{c \cdot n}{f}; \text{ i.e. independent of } s! \]

\[ f = \text{percentage of segments freed during garbage collects.} \]

\[ f \text{ is the sought parameter.} \]

\[ n = \text{average number of instructions executed per segment in garbage collects. If } f = 5 - 10\%, n \text{ will be } \approx 50. \]

The ratio \( \frac{1}{s \cdot f \cdot M} \) is the average number of times per space request that we actually have to perform a garbage collect:

Ex. \( s = 0.25, f = 0.07 \Rightarrow \text{the ratio becomes } \frac{1}{7}, \text{ cf. sec. 4.7.5.3.} \)

Naturally, the p-function will depend heavily on the actual replacement policy. \( p(s) \) may, according to [Coffman, Varian, 1968], be approximated by an exponential function for "large" \( s(> 0.3?) \):

\[ p(s) \approx a \cdot e^{-bs} \]

Empirical studies lead to \( a = \frac{1}{500} \) and \( b = 7 \), given LRU-replacement.

Since we don't utilize the entire store, the exponent must be corrected:

\[ p(s) = p(s,f) \approx a \cdot e^{-bs(1-f/2)} \]

The \( N \) also ought to have been adjusted, as we spend a lot of CPU-time updating recent-usage-information in proc calls. More about this later.
We now have:

\[
T \approx N_\text{T} \cdot t + s \cdot M \cdot T' + (N-N_\text{T}) \cdot a \cdot e^{-bs(1-f/2)}(T' + \frac{t'}{2})
\]

where \( T' = T'' + t' \approx 51 \text{ ms.} \)
\( t' = \tau' \approx 175 \mu \text{s.} \)

We are interested in \([T(f)]_{\text{min}}\):

\[
\frac{dT}{df} = 0 \implies \gamma(s,f)[(T' + t')^{-\frac{1}{2}} - \frac{t'}{2T'}]\]

\[
bsT' f^2 + t' bs f - t' = 0
\]

\[
f = \frac{-t' bs + \sqrt{(t' bs)^2 + 4bsT't'}}{2bsT'}
\]

Setting \( x = bsT' \):

\[
f = \frac{-t' x + \sqrt{(t' x)^2 + 4xT't'}}{2xT'} = \frac{-0.175 \cdot x + \sqrt{0.03 \cdot x^2 + 35.7 \cdot x}}{102 \cdot x}
\]

\[
\approx \frac{-0.175 \cdot x + 6.0 \sqrt{x}}{102 \cdot x}
\]

\[
s = 0.5 \quad \Rightarrow \quad f \approx \frac{-0.5 + 11.4}{356} = \frac{1}{32} \approx 3.125 \%
\]

\[
s = 0.31 \quad \Rightarrow \quad f \approx \frac{1}{22} \approx 4.545 \%
\]

\[
s = 0.20 \quad \Rightarrow \quad f \approx \frac{1}{22} \approx 4.545 \%
\]

Measurements indicate \( \approx 6\% \) as a balancing point for \( s = 0.31 \).

This is because the \( \frac{t'}{T'} \) quotient is a bit too optimistic concerning the number of garbage collects. E.g. if we always free 6 segments, we don’t avoid garbage collects more than average 4 times, because segment sizes are unevenly distributed.
Example 1.
MARY-MARY stand-alone compiler, data from previous tables.

Core size = 49152 words, i.e. $s = 0.31$ (27.3 K of 87.8 K).

$f = 5\%$.

$N \approx 6300000$, $N_I = 600000$ (10% of $N$).

$M = 403$, $s \cdot M = 125$.

$p(s = 0.31, f = 5\%) = \frac{1}{600} \cdot e^{-2.05} = \frac{1}{4500}$

I. e. $\frac{5700000}{4500} = 1150$ predicted segment faults, in steady-state.

$T \approx 6300000 \cdot 3.5 + 125 \cdot 51000 + 1150 \cdot (51000 + \frac{175}{6.05})$ (in $\mu s$)

$= (20.5 \cdot 10^6 + 64.5 \cdot 10^6 + 3.9 \cdot 10^6) \mu s = 99\text{ seconds}$ (in good accordance with measurements).

or:

$T = N \cdot 2.4.5$

Note that the 20.5s CPU-time figure includes 2.5s of proc call trace, and that no IO-time but disk-IO is considered.

Compilation speed: 1.7 instructions per second; in trace-free steady-state.

CPU-utilization (mono-programmed system):

$\tau = \frac{1}{4.5} \approx 0.22\%$

However, be aware that proc calls of SAVE routines require only $\approx 90 \mu s$, while OLAY proc calls cost $\approx 550 \mu s$. That is, the above $N$ might be reduced to $\approx 1/3$, since $\approx 80\%$ of the CPU-time are OLAY proc calls. The overall SAVE-code effect, including system overhead, may amount to $\approx 10$, i.e. 15 - 20 instructions per second. See later also.
But alas, a compiler on, say, 100 K resident code + data tables cannot be run on our minis. The addressing range is limited to 64 K. The alternative is e.g. a 3 pass compiler with each pass occupying ≈40 K (resident?) code, see 4.8.1.2.

Example 2,

A hypothetical calculation: throughput time for the wipeout algorithm.

If we insert $f = 100\%$, we will roughly get a model of this algorithm. 48 K storage is assumed.

\[
p(s \approx 0.32, f = 100\%) = \frac{1}{600} e^{-1.05} \approx \frac{1}{1700}, \text{ or } \approx 330 \text{ segm. faults.}
\]

\[
T = (20.5 - 6.0) s + (3300 + 130) \cdot 51 s \approx 15 s + 175 s = 190 s.
\]

This is ≈50s more than measured; mainly because the \((1-f/2)\)-factor is a bit too pessimistic of the space utilization for large \(f\).

4.7.7.2 Some reflections on system parameters

a) With a 48 K core store, a 5%-change in the OLAY-code pool (i.e. 1.5 K of 30 K) leads roughly to an opposite 10%-change in the number of segment faults.

b) With a 64 K core store, we experienced 158 excessive segment faults above the 403 minimum limit. This can hardly be done better; remember \(s \approx 0.50\). Projected steady-state compilation speed for stand-alone compiler: 135/35.5 = 4 instrs./s.

c) Assume that we get a fast drum with ≈5 ms access time. The estimated, steady-state throughput time for our 45-lines test program, given 64 K and FIFO-replacement algorithm, will then be: 11s + (813 - 220) \cdot 0.005 s ≈ 14 s. I.e. ≈2.5 times faster than the present 50 ms-disk, or ≈10 instructions/s.

d) The above calculation demonstrates the relation between space administration overhead (incl. proc call overhead) and accumulated segment fetch-times:
Assume 24000 OLAY proc calls and a 64 K machine. The difference in proc call overhead between LRU and FIFO is \( \approx 300 \) \( \mu \)s per call, or 7 s totally.

FIFO, however, implies 252 segment faults more than LRU (813 compared to 561), or \( \approx 1 \) s totally the other way.

That is, a cheaper replacement algorithm pays off when the swap media is fast.

e) Since the OLAY proc call overhead is considerable, it might be worthwhile to implement it in hardware, through micro-programs.

If such overhead could be reduced to \( \approx 1/5 \) and given the previous 5 ms drum, projected steady-state compilation time (64 K & FIFO) for the previous test program would be:
\[
(6.5/5+4.5+600 \cdot 0.005)s \approx 9s. \quad \text{That is,} \quad \approx 15 \text{ instrs./s.}
\]

f) We have not considered the effect of extra dictionary space for (large) MARY programs, containing many preludes. A reduction in the available OLAY-code pool by 5 - 20 K may be expected in some cases.

g) For full RUNSYS, 6K extra SAVE-space is required, compared to a stand-alone compiler-system. This will reduce the efficiency of the system accordingly (study the curve!). Misc. system overhead and unknown multi-user effects of global, merged LRU/FIFO- chains containing partly shared code segments, may also lead to non-optimal thruput. Presently, I have no data on this.

However, letting 2 (batch) compilations run in parallel on a 64K machine (9 K additional SAVE-space for the second one), plus a couple of interactive editor runs, might bring the CPU-utilization up to 60-80\%. That is, \( \approx 3 \) instrs/s.

h) Optimizing VM-routines occupy \( \approx 1-1.5K \) extra SAVE-space.

On a 48K stand-alone system, this will increase the fault rate by \( \approx 10\% \).
4.8 Concluding remarks about the virtual memory system in RUNSYS

4.8.1 On system performance

4.8.1.1 Current status

Indeed, the system trashes when executing the MARY-MARY compiler. CPU-utilization may however be normalized, by relying on a multi-user effect to balance the system. In fact, with a high degree of time-sharing, \((\text{total virtual memory})/(\text{physical memory})\) - ratio may be \(\approx 5\), yet CPU-utilization may be as high as 50 - 80%.

Generally, with 48 K and 64 K machines and LRU-replacement policy, system thruput appears to be acceptable even for compilation purpose (1 - 3 instrs./sec.). This is, in my opinion, a respectable achievement considering the circumstances: 119 K compiler code and a 50 ms disk.

Memory deadlocks seem to cause negligible problems, if the maximum size of (SAVE) segments is less than \(\approx 2000\) words. At worst, the most greedy user task will be killed, as system tasks never acquire SAVE segments \(\approx 700\) words.

4.8.1.2 Future tuning work

The system has already been tuned by a factor \(\approx 3\). It is not likely that I should come up with radically improved VM-techniques in the future; such as to implement "clairvoyant" replacement algorithms to optimize lost disk revolutions; or otherwise reduce the disk access time by letting the loader "pack" the olay file or "sort" it according to usage patterns. However, as mentioned under point g) in sec. 4.7.7.2, there may exist unidentified bottlenecks (some interface modules?) in the VM-system under full RUNSYS. A future monitored test of RUNSYS, while this is running for some time with all the bells and whistles on, will possibly reveal such.

I have a hunch that future tuning efforts will have to be concentrat on possible reorganization of existing OLAY programs, the MARY-MARY compiler in particular.
Re-design topics:

a) Space-optimization of existing routines. E.g. a proc trace of ≈30 words is currently included in most routines; yielding ≈15 K totally or ≈10%.

b) Local re-design of module interfaces. E.g. the most-used compiler routine (GETCARDLINE) can easily be optimized; eliminating ≈2000 OLAY proc calls in the previous test program. I.e. ≈1 s of CPU-time totally, or ≈5%.

c) Utilization of the OLAY-data facility for dictionary entries, and to pack dictionary texts. This may reduce the dictionary-size to 50 - 60% of its current value.

d) Global re-design of the entire compiler. As late as February 1975, the compiler people thought the compiler could fit into 65 K; now it is 119 K!

Furthermore, our present one-pass top-down compiler organization implies many recursive calls, so that large parts of the compiler "rotates" in memory when running under space shortage. (Such "loop structures" are notoriously difficult to handle in VM-systems.) Other parts of RUNSYS, like FILESYS, show a much more transient usage pattern.

To improve upon the addressing locality in the compiler, a multi-pass compiler organization might be attempted. However, in view of the exceptionally low number of segment faults on 64 K, of rather poor multiprogramming qualities of such a compiler (if several users execute their own pass of the compiler, the net effect is negative!), and of the high re-write costs (2 - 3 manyears?), it may not be worthwhile. On the other hand, our compiler people may find that a multi-pass compiler organization exhibits highly favourable qualities when compiling an ALGOL68-like language. Anyhow, this is strictly not my responsibility to decide or to elucidate further.
4.8.2 Convenience of usage

Presently, RUNSYS offers safe and automatic mechanisms for OLAY-code. A similar, user-oriented system for OLAY-data will require that an olay heap facility is implemented in MARY, so that the compiler can insert necessary segment-update directives.

4.8.2.1 Ideas on OLAY-data in MARY

OLAY-data ought to be implemented through so-called master descriptors, one per segment. In addition comes user-accessible descriptor pointers. - See also sec. 13.2.

Tentative declarations:

```
MODE MASTERDECRIPTOR(M)=[INT DISKADDR,LENGTH,
    REF M OBJ,
    BOOL WRITTEN],
REFHEAP(M)=[REF VAL MASTERDECRIPTOR(M)DESCR],
DICTENTRY=[[6] PACK CHAR NAME, ----];
REFHEAP(DICTENTRY) PTR1:=NEWOBJECT(DICTENTRY),
    % Must generate a new master descriptor.
    PTR2:=----;
IF PTR1.NAME[1]='A'
    THEN ----
    FI;

PTR1.NAME:=PTR2.NAME; % Copy 6 PACKed CHARs.
PTR1:=PTR2; % Copy master descriptor pointer.
```
Note: Explicit pointers to data within OLAY segments must be forbidden. Such data are non-REFABLE:

Ex. ROW PACK CHAR R:=PTR1.NAME; %% Illegal!!

On performing a selection, like PTR1.NAME, the compiler must insert calls on segment checking routines behind our back. This can be accomplished by re-declaring the DEREF-coercion for REFHEAP(--) modes. For the '!=':-operator, extra update operations on write-flags must be carried out. Thus the store-operator for these modes have to be re-declared.

Then the locking problems:

As all MARY operators can take maximum two operands, it's sufficient to lock the two most recently accessed segments. It may even suffice to lock one segment at a time, if we are willing to sacrifice some run-time efficiency.

To alter the data allocation-strategy from "OLAY" to "SAVE", we need only redefine REFHEAP to:

\[
\text{DEFINE} \\
\text{REFHEAP}(M) = \text{REF M} $; \quad \text{%% Now a macro.}
\]

- and then recompile.

The inverse conversion from "SAVE" to "OLAY" will be more laborious, as all data objects now must be indirectly referenced via special descriptors; cf. above example. See also next section.

4.8.2.2 On uniformity of data accesses

Data may be allocated:

1) in the machine registers,
2) in static data areas,
3) on the stack.
4) in resident heap objects,  
5) in olayable/virtual heap objects, or  
6) on ordinary mass storage files.

Alternatives 1 - 4 can be rather uniformly treated in MARY programs. That is, a program doesn't have to be modified significantly (we only change some declarations), when another data allocation strategy is adopted.

For the time being, a considerable re-write of user/system programs under RUNSYS may be needed to realize alternative 5. In a paging system, this would not have been necessary, if we restrict ourselves to the virtual address space (see below also).

The distinction between alternative 5 and 6 (between virtual memory data and file data) is not clear:

Ex. If the address space is sufficiently extravagant, such as in the MULTICS system with 32 bits addresses, a programmer is almost relieved from explicit file management.

Ex. In RUNSYS, the implementation of virtual (OLAY-)data objects and file data exhibit overlapping/redundant functions, like a linear address space. The difference lies in how "directly" a given datum can be accessed. Study secs. 4.3 and 6.4!

4.8.3 A few comments on paging versus segmentation

The subject is fascinating, and relatively well treated in the literature, cf. [Denning, 1970].

In practical life, paging tends to dominate over segmentation, because it's cheaper to implement in hardware. In our case, a software segmentation system was the best we could invent, given the current minis.

Digression: I have not yet read about any really successful paging system (disregarding the pioneer implementations), while there are several famous segmentation systems (the Burroughs- machines and their derivatives).
4.8.3.1 On run-time efficiency

Traditional objections to the efficiency of paging systems all boil down to matters of artificial page partitions (non-locality of page accesses). Attempts e.g. to put code and associated data on the same pages, to improve addressing locality, will ruin code reentrancy.

The drawbacks of segmentation systems have been:

1) complicated space allocation algorithms, because segments have different lengths, and

2) that we probably need a fixed-head drum to obtain satisfactory efficiency, because the average segment length may get rather small (≈50 48-bits words on the B5500).

4.8.3.2 On space utilization

[Randell, 1969] claims that "internal fragmentation" in paging systems (usually 20 - 30% unused space, because of too large pages) is more serious than "external fragmentation" in segmentation systems (≈10 - 15% unusable hole-space, if segment lengths are <10% of the available storage).

In segmentation systems, the maximum available segment length may be rather small (4 - 10 K in our system; 1024 words on B5500). This may, as described before, lead to memory deadlocks.

On the other hand, indirect references through segment descriptors may sometimes give a larger virtual addressing range (>64K?) than paging does.

4.8.3.3 Multiprogramming qualities

In segmentation systems, code segments for standard systems software will normally be sharable between processes. That is, utilization of reentrant code may reduce the required amount of (resident) code by a factor 2-5.

Such a sharing can often be difficult to realize on paging systems, as several page tables must be updated on page faults.
4.9 A fast free-list system for very small data areas.

Regrettably, no rules without exception.

We do need a fast space generator for small data objects (<16 words) in addition to the MEMLINK-mechanism. This is supplied by a free-list system:

```
1
2
...

AREAHEADS-vector

Vacant two-word objects.
```

```
Linked pool of large areas (~100 words each), which later are divided into smaller ones of length <16 words. The large ones may initially be allocated as ordinary MEMLINKs.
```

Such a tailored space generator is used for SHARED (REF TASK)-objects when passing TASKs to REQUESTHANDLER and TASKSINK; it is used for 2-word FILEINFO-objects to link all opened files of a task into a list; and similar system purposes.

Two routines, GETAREA and FORGETAREA, administrate these kinds of data objects. If GETAREA runs out of space, a system crash may occur. Necessary precautions will be taken to prevent this, see sec. 8.1.
4.10. Aspects of a general heap implementation for operating systems.

A few modern high-level languages (SIMULA, ALGOL68) have implemented a general and automatic heap facility for data objects. The implementations are reliable and surprisingly efficient, and relieve programmers from a substantial part of the problems of space administration.

It seems that some of the most difficult problems in practical construction of operating systems have to do with administration of primary storage.

In RUNSYS, we have a primitive heap function in our MEMLINK-system. Data objects of different lengths and types can be allocated and deallocated (either explicitly, during garbage collects, or because of task terminations).

The system is not safe - we may still access deallocated objects. We must take care to ensure correct globalness (OWNER-fields) when passing MEMLINK-objects between tasks. Deallocation is not automatic for SAVE objects, and we can only compact OLAY-code segments (minimal updating problems).

For stack objects we also experienced scope problems with up-stack pointers. Cf. the preventive function of CREATOR-attributes in semaphores and resources, and the critical-block-trap mechanism.

- What a mess merely to maintain the available pool of primary storage!

Since MARY in principle can support an automatic and efficient heap facility for ordinary programs (including compilers) [Conradi, 1974a], why not let everything be heap, i.e. let the heap be shared by all processes including the operating system! A data stack will now behave like a normal heap object. Data objects belonging to a terminated process will automatically be cleared on the first garbage collect. It's a fascinating idea.
Comments:

* Pointer representation.

If we want a data stack to behave like an ordinary heap object (i.e. it may be compacted), we ought to implement stack pointers (or pointers in general) as descriptors: [start address, offset], because they can point into the middle of an object/stack. However, indirect addressing will get rather slow on common hardware, and we need more space for pointer objects.

* The garbage collect time may be intolerable for real-time events.

However, we don't aim at a dedicated application, and if I/O drivers possess a sufficient amount of static buffers, is this really critical? Consider the savings in memory if we were able to compact the free space.

Some figures:

The SIMULA garbage collector on the Ul108 (cycle time ≈ 1.0 μsec) uses about 0.1 second to garbage collect a pool of 16 K 36-bits words, when the average size of heap objects is ≈ 20 words, i.e. ≈ 800 data objects. It's normal to spend 10-20% of the total CPU-time in garbage collects, if we have at least 30% free space afterwards.

The data objects in RUNSYS are expected to be much larger, 50-200 words on the average. This will reduce the garbage collect time.

A rough estimate for our Norwegian minis (cycle time ≈ 1.5 μsek) indicates that they would essentially halt every 3-4 seconds for 0.3 second at a time. As primary storage (and disk-channel capacity) constitute the bottlenecks in our system, we can afford to spend this extra CPU-time.
* Difficulties with garbage collects in a multiprocess environment.

For a single program/process the compiler has full control over allocation of registers in case of garbage collects. But how do we update pointers residing in registers, if we compact a shared global heap? (Assume that stacks are non-compactable.)

Solution: We must try to avoid using the registers for indirect addressing. If a load-store operation via a register is required, we must update a local register-map for the actual process before and after the load-store. On a multi-register machine we may, of course, reserve some registers for heap operations.

Disabling of interrupts (or likewise) during such operations is highly undesirable.

Redefinitions of the DEREF-coercion and of the store operator (':=:') for heap pointers may free the MARY compiler from this kind of register administration.

Note: Whether we have a machine with one or several CPUs, is in principle irrelevant.

Digression: The ACM student award lecture for 1975 describes an algorithm, that allows garbage collection to be performed by an asynchronous process. (Allocation of new data objects must be synchronized; and pointer assignments may imply updating of status information in the referenced data objects.) See [Steele, 1975].

* Why compact?

Compaction seems to cause updating problems, e.g. for data in the registers.

We will also experience difficulties in updating code segments - in memory or on mass storage. Likewise, the suggested static buffers for real-time processes are introduced because of problems in this area.

Furthermore, the extra space recovered through a compaction
procedure may not be worth the effort - the store is often ≈90% filled up by then.

Maybe what we need is two types of heap objects, one that can be compacted and one that cannot?

Memory now consists of two areas, growing towards each other:

![Diagram showing compactable and non-compactable heap objects]

- **Compactable heap objects:**
  - "normal" heap objects,
  - location-independent code segments.

- **Non-compactable heap objects:**
  - stacks, SAVE-code segments,
  - IO-buffers, misc. descriptor blocks.

Now, time-critical IO-processes may be run nearly as before. They don't call heap generators explicitly, but are given non-compactable heap buffers from ordinary processes. One of these have to pay for the necessary overhead when the heap is full.

References to compactable heap objects may for instance go via descriptors in non-compactable objects.

This partitioning of memory into a compactable and a non-compactable part resembles the current MEMLINK organization. However, we don't have a powerful enough hardware to implement necessary "descriptors" efficiently.

**Concluding remarks.**

The hardware of any machine offers a variety of accessing and relocation mechanisms.

It would have been nice to have a single space allocation primitive on a machine; to be used for data and code, to be used for virtual memory, to be used for classic heap storage by the operating system and by ordinary users.
If hardware and software people could cooperate better, may be these perspectives are not completely unrealistic? The Burroughs machines may give us some ideas in this direction. See also hardware discussion in ch. 13.

Supporters of hardware paging may argue that paging solves both the heap and the virtual memory problem. But e.g. the SIMULA IBM370-implementation performs normal garbage collects on virtual memory objects. And the operating system on a paged machine surely has its own heap generators for system purposes.

Interrupt-handling. Binary and formatted IO.

Much effort has been devoted to shield the remaining operating system from hardware peculiarities. Only the initialization routine(s), interrupt-handling section and bottom-level system tasks (IO-drivers, clock handler, dispatcher) "know" how the underlying hardware works.

In RUNSYS less than 2% of the MARY source text deals with these things. For instance, only ≈250 lines are different in the NORD1-version for a 16 level and the SM4-version for a single level interrupt system.

Inevitably, this chapter contains some very technical details, but emphasis has been put on displaying the general structure of low-level task management and IO-interface. References to NORD-SM will usually appear as examples.

5.1. Active system tasks in RUNSYS after system generation.

Active user-task (executing a normal program, a filesys routine, the compiler, etc., forked by the user.

Basic user task. (One per active TTY-user.)

IO-processes:

- PAPER-TAPE DRIVER
- PAPER-READER DRIVER
- CARD-READER DRIVER
- LINE-PRINTER DRIVER
- DISK-Driver
- TTYIN DRIVER
- TTYOUT DRIVER
- REQUEST HANDLER
- SPACE ADMINISTRATION
- CONSOLE
- TASK SINK
- PRINT FILES
- IDLE DEVICE HANDLER
- CLOCK INTERRUPT
- BAD INTERRUPT
- POWER FAILURE
- ENDO THE WORLD

One set per TTY.

TASK CHANGER, the dispatcher
The main functions of the indicated system tasks ought to be self-explanatory.

A few remarks only:

CONSOLE takes care of operator console communication.

IDLEDEVICEHANDLER supervises all external devices.

Ex. When a TTY has been logged in, IDLEDEVICEHANDLER will fork a JOBINTERPRETER-task to undertake further actions.

IDLEDEVICEHANDLER also wakes up every M'th second, primarily to treat deadlocks. See sec. 8.2.

If we want to support a spooling system for batch processing, some additional tasks are needed.

Some of the IO-tasks are really pairs: E.g. we have a line printer control task, PRINTFILES, to chew printfiles and deliver output lines to the real LINEPRINTERDRIVER.

5.2. TASK-mode for NORD1 and SM3/4.

We shall now describe the TASK-mode in RUNSYS rather thoroughly, as the following sections on interrupt-handling assumes knowledge of it.

General modes.

MODE DEVICE,
  TASK,
  IOPFILE,
  BYTE=SET(0 TO 255),
  DLIST(M)=(REF DLIST(M) FORWARD,BACK, M VALUE), ** FOR LIST
  LIST(M)=(REF LIST(M) NEXT, M VALUE), ** PROCESSING
GLOBAL SET DECLARATIONS

SET EXECREQUEST= % FOR SOFTWARE INTERRUPTS.
(NORMALTERMINATION,ERRORTERMINATION,
SOURCELANGUAGEERR,SUBSCRIPTRANGEERR,STACKOVERFLOWERR,
DATAPRECISIONERR,ILLEGALRECURSIONERR,EMPTYREFERENCEERR,
INVALIDLENGTHERR,RUNTIMEMODEERR,DUMMYREQ,EMPTYPROCERR,IDL,
DUMMYREQUEST,CRITICALBLOCKERR,MAXTIMEERR,MAXMEMERR),
MEMTYPE= % FOR SPACE ALLOCATION.
(FREEMEM,SAVEMEM,OLAYCODEMEM,OLAYCPUQCODEMEM,
OLAYEXECCODEMEM,OLAYDATAMEM);

GLOBAL MODE DECLARATIONS

MODE MEMLINK=DLIST(:MEMTYPE TYPE,REF TASK OWNER), % FOR SPACE ALLO
EVENT(M)=((REF TASK TASKS,LIST(REF M) DATA), % SEMAPHORE-MODE
TASKINFO=((BYTE PRIORITY,FIXED TIMELIMIT, % NEEDED WHEN
INT SAVELIMIT,STACKSIZE), % FORKING TASKS.

Task modes.

MODE FILEINFO,
ACTION(M1)= % DESCRIBING SPECIAL TASK-TERMINATION PROCS
% IN WINDUPS=CHAIN.
(:PROC OLAY RECURS (REF M1 PARM) VOID ACTIONS,
REF M1 INFOPARM);

DECLARATIONS FOR NORD-1, SM-3/4

MODE
INTERRUPTLEVEL=SET(0 TO 15),
ESTRUCT=((BITS A,D),
FSTRUCT=((BITS T,A+D),
SEGMENTDESCRIPTOR= % OLAY-CODE-DESCRIPTOR MODE.
 (:LABEL COREADDR,
PACK SET(0 TO 2**14-1) DISKADDR,
PACK BOOL UPDATED, RESIDENT),
RETURNDESCRIPTOR=((REF SEGMENTDESCRIPTOR SEGOESCR,INT OFFSET),
CRITINFO= % CRITICAL-BLOCK-TRAP MODE.
 (:INT COUNTER,
REF RETURNDESCRIPTOR RETURNPOSITION,
RETURNDESCRIPTOR RETURNCOPY),
STACKDESC=((INT STACKTOP,STACKLIMIT,STACKBASE));
DECLARATIONS FOR SM-4

MODE SUBENVIRONMENTSAVEAREA=(STACKDESC STACK,
   REF SEGMENTDESCRIPTOR CODESEGDESCRIPTOR,
   FSTRUCT LSB, F),

OUTSM

DECLARATIONS FOR NORD-1, SM-3

MODE

ENVIRONMENTSSAVEAREA=
   ([LABEL P, BITS X,
      SUBENVIRONMENTSSAVEAREA SUB],
   TASK=(ENVIRONMENTSSAVEAREA SAVEAREA,
      REF TASK NEXTTASKLINK,
      LINK FIELD FOR
      TASK QUEUEING,
      CHILD,
      PARENT,
      SIBLING,
      REF CRITINFO CRITICAL,
      REF LIST(ACTION(PROTEAN)) WINDUPS,
      CHAIN OF SPECIAL
      TASK-TERM, PROCS.
      FIXED TIMELIMIT,
      INT SAVELIMIT,
      REF DEVICE DEVICE,
      PACK BOOL DIRECTLYTRIGGERED,
      SAVE-MEMORY LIMIT,
      ASSOC. DEVICE, IF ANY
      TRUE (I.E., LEVELNO
      >> TASKLEVEL),
      IF TRIGGERED ON
      EXTERNAL INTERRUPTS,
      KILL,
      SYSTEMPROCESS,
      PACK BYTE PRIORITY,
      PACK SET(0 TO 2**4-1) TASKTREELEVEL,
      PACK BOOL ABORT,
      (REF IFILE INPFILE,OUTFILE) IO,
      STANDARD IFILES,
      REF FILEINFO OPENEDFILES,
      REF PROTEAN RESOURCEHEAD,
      (:10) PACK CHAR NAME,
      PACK INTERRUPTLEVEL LEVELNO,
      PACK BOOL JCSTATE,
      PACK EXE_CREQUEST REQCODE,
      REF EVENT(PROTEAN) SEMAPTR,
      FIXED LASTTIMEACTIVE,
      LABEL RESUMEADDR,
      INT NOOFUPDATES,
      REF VAL FILEINFO STANDARDFILE);
As stated in sec. 3.1.2, only the ENVIRONMENTSAVEAREA-mode and its submodes are machine-dependent.

The DEVICE-mode will be described in section 5.6 on basic I/O.

5.3. Interrupt-handling.

On any machine, a hardware interrupt causes roughly the following actions:

<Save registers.>
<Detect interrupt.>
<Select new task to receive CPU, e.g., the dispatcher. Update system data if necessary.>
<Unsave registers and start new task.>

The interrupt system must be disabled throughout these actions.

How and by whom (hardware/software) these activities are carried out, is extremely machine-dependent. E.g., does hardware utilize special task-or savearea vectors? Where must the software interrupt-handling section be located in memory? How many interrupt levels do we have?

Interrupt-handling on NORD1 and SM3/4 will now be treated in some detail. After this, it should be easy to see which modifications are needed in RUNSYS on other machines.

5.3.1. Basic task structures on NORD1 and SM3/4 with a 16 level interrupt system.

4 bits in a STATUS-register indicate the currently executing level. This level is determined by two hidden hardware registers: A Priorit Interrupt Detect and a Priority Interrupt Enable register.
Allocation of hardware levels for RUNSYS tasks:

LEVELHEADS-vector, at hardware addresses 16-31.

Externally triggered tasks.

Increasing priorities.

15  →  POWERFAILURE-task.
13  →  CLOCKINTERRUPT-task.
11  →  Current IO-driver for input, if any
07  →  Current IO-driver for output, if any
02  →  RUNNING-task, normal user or system task.
01  →  TASKCHANGER-task, the dispatcher.
00  →  ENDOPTHEWORLD-task, a security trap.

Levels 7, 11, 13 and 15 are hardware-wired to support special drivers. Observe that RUNNING contains the value of LEVELHEADS [2].

When a RUNNING task on level 2 executes a software interrupt (i.e. a WAIT-) instruction, it gives up priority; and we fall down to level 1 - straight into the arms of the dispatcher, which will select and start a new RUNNING task. If no new RUNNING candidate exists, the dispatcher goes in a waiting loop - the idle loop of the machine.

Two global cells, CURRENTTASK and SYSTEMTASK, will represent respectively the currently executing task and a possible system task on interrupt level >2. One or both of RUNNING and SYSTEMTASK will be non-empty. CURRENTTASK=RUNNING ⇒ SYSTEMTASK=NIL.

Note: We don't really need the globals RUNNING, SYSTEMTASK and CURRENTTASK, but they simplify the programming.
On sensing interrupt, hardware will automatically save current
PREG and XREG into the task head of the executing level, and
jump to address 32, with XREG pointing at the new level's task
head.

**Interrupt-handling section at address 32:**

1. Save remaining registers (+ STACKDESCRITOR + CURRENTCODE-
SEGDESCR).

2.a. If we have fallen down-level, no special actions; unless
the old level were 2: Then NIL:=: RUNNING:=: REQUESTOR, so that
the dispatcher can take further actions to handle the
software interrupt.

2.b. If we have gone up-level and the new level is an IO-level
(7 or 11), some detective work must be carried out. That
is, which device caused the interrupt?

Associated with IO-levels 7 and 11, there are two daisy chains
of device descriptions, containing IOIS-instructions
for sensing IO-interrupts. By "executing" this
chain, it is possible to identify the actual device (or one
of them). Such device descriptions also contain a pointer to
an OWNER-task, being the next IO-driver to be triggered.

2.b.1. We may choose to perform the daisy chain-searching inside
the interrupt-handling section itself, and then trigger the
actual IO-driver by putting a pointer to its TASK-object into
LEVELHEADS [7 or 11].

2.b.2. We may also "ask" the last active IO-driver on level 7 or
11 to do the searching for us. This means that all IO-drivers
on these levels must give up priority inside some restart-
routine; which first performs an IOIS-scan, and then "triggers"
the real IO-driver on the same level.
3. Unsave registers (+ 4 other cells) and trigger new task through a special hardware instruction. Note that XREG, pointing at the new task head after point 1, might have been altered in the meantime; e.g. if we have selected a new IO-driver.

Naturally, the corresponding MARY code is programmed to achieve optimal run-time efficiency. There is hardly one superflous instruction compared to assembler.

5.3.2. Basic task structures on SM3/4 with a single level interrupt system.

All tasks now run on the same level. There is no hardware LEVELHEADS-vector.

The globals RUNNING, SYSTEMTASK and CURRENTTASK have the same functions as before. On sensing interrupts, hardware will save the current PREG in location 0 and jump to address 1.

Interrupt-handling section at address 1:

1. Save registers (+ misc.) into CURRENTTASK's savearea.
2. Analyze hardware STATUS-register for power failure, illegal instruction (simulated software interrupt), etc. Perform NIL::RUNNING::REQUESTOR if necessary. Analyze IOIS-chains for possible IO-interrupts (same detective work as before).
3. Select new CURRENTTASK:
   3a. This may be either the POWERFAILURE-task, the CLOCKINTERRUPT-task or an IO-driver. That is, a SYSTEMTASK.
   3b. Otherwise, we check whether RUNNING ≠ NIL, and continue with this task.
   3c. Now, only the dispatcher can be triggered. SYSTEMTASK is updated.
4. Unsave registers (+ misc.) and restart the new CURRENTTASK. The restarting implies a hardware trick; please consult the program listing (TREATINTRPT-routine, l. 179).
Note 1: The interrupt-level priorities on a 16 level system are partly mapped by the selecting sequence for CURRENTTASK in point 3a, 3b and 3c.

Note 2: We may get a new interrupt inside a system driver (SYSTEMTASK), i.e. during processing of the previous interrupt. Possible solutions:

1) All SYSTEMTASKs are executed with interrupts entirely disabled. We must, however, take care so we don't detect a powerfailure-interrupt too late. And we must not call routines, that enable the interrupt system themselves.

2) We briefly analyze the new interrupt and queue the corresponding driver into a SYSTEMTASKQ, before restarting the current SYSTEMTASK. This SYSTEMTASKQ must then be investigated in point 3a above, when selecting a new SYSTEMTASK.

Observe that triggered tasks in our previous multi-level system - hanging on misc. interrupt-levels >2 - now will be explicitly queued up in SYSTEMTASKQ.

5.4. The dispatcher, TASKCHANGER (NORD-SM variant).

Functions:

1. First it will release a possible REQUESTOR-task to the REQUESTEVENT-semaphore, thus triggering the REQUESTHANDLER-task.

2. Then it will go in an idle-loop until SEARCHPRIORITYQUEUE (READYQ) =: RUNNING =: LEVELHEADS[2] yields a non-empty value.

On a 16 level interrupt system, the dispatcher runs on level 1. It must explicitly trigger level 2 (by fiddling the PID-register) to start the new RUNNING task.
On a single level system, it will simply execute a software interrupt-instruction, which passes control to the new RUNNING task.

Note: No "dynamic" task priorities are implemented, such as letting IO-bound tasks have better priorities than CPU-bound ones.

5.5. The CLOCKINTERRUPT-task.

Regrettably, there is only one clock on the NORD-SMs, so cur SYSTEMTIME goes a bit slower than the wallclock time. We spend some un-accountable cycles on interrupt-handling and in the CLOCKINTERRUPT routine itself.

Functions:

1. TIMEQUANT=:+SYSTEMTIME, where TIMEQUANT=40ms.

2. Then a TICKTOCKQ is inspected, and possible scheduled tasks will be triggered. (There exists a TIMETUNNEL-function, by which a task may request sleeping for a specified time period.)

3. Lastly TIMEQUANT=-:RUNNING.TIMELIMIT, and we check whether we have a max. time abortion.

5.6. Basic IO-handling (NORD-SM). DEVICE-objects and IO-drivers.

Necessary device information is contained in so-called DEVICE-objects, being established on system generation.
DEVICE-objects (for NORD-SM) are organized as follows:

The order by which DEVICE-objects are linked in their respective daisy chain, will assign IO-drivers a kind of mutual priorities.

There is one DEVICE-object per device. All special IO-instructions for a device are located in this object.

When, say, a TTY-output driver wants to transfer a physical character, it must execute one of the special IO-instructions (IOAP <devno>) in the DEVICE-object. (However, it is a bit dirty to execute "data" cells as code.)

Thus, all driver routines will be reentrant, and we can add new (similar) devices dynamically to the system on system generation without recompiling existing IO-routines.
An IO-driver communicates with ordinary tasks via a NEWREQUESTS-semaphore in the DEVICE-object.

**Note 1:** There is exactly one IO-driver per device, the OWNER-task. (A TTY is regarded as two devices.) When an IO-interrupt has occurred we end up somewhere in the interrupt-handling section, via a JPL-instruction, with (LREG-value + 1) pointing at the address of the associated IO-driver. This task can now be triggered.

As mentioned in sec. 3.2.3, we may regard the DEVICE-object as a **semaphore**, being given a "resource" by hardware, thus triggering its only queued task, the OWNER-task.

**Note 2:** Only an IO-driver knows how a device really works.
(Which instructions and which formats to use). Other tasks may only know things like disk sector length and TTY-buffer size.

**Note 3:** Some IO-drivers may be "triggered" both externally and internally: Consider disk transfers and TTY output lines. It will simplify the programming to let these IO-drivers behave like "ordinary" tasks when they are software-triggered, so that necessary synchronization can be done by classic ACQUIRE-RELEASE calls.

Thus, an IO-driver that doesn't expect external interrupts, (not a TTY-input driver!), may hang itself up in a normal semaphore-queue through an ACQUIRE call.

This can be accomplished by a slight modification of the dispatcher: When it discovers that an IO-driver (which of course has the very highest priority) has been put into the READYQ because of some RELEASE call, it is fairly simple to trigger the new task as a special SYSTEMTASK (the interrupt level stands in the TASK-object), instead of a normal RUNNING task.

However, this kind of uniform synchronization may be less efficient than a more dirty, ad-hoc solution.
Note 4: On SM3/4, hardware indicates a so-called missing-connect interrupt, if we execute IO-instructions of dis-connected devices.

This dilemma can be solved by replacing IOIS-instructions in DEVICE-objects of dis-connected devices by dummy-instructions (JPL,I 5). Some additional testing is also required. This trick is credited to Per Øvrebø.

Note 5: If we cannot identify a device by scanning the relevant IO-chain, we have an unknown device (or a temporarily "deleted" one in case of dis-connect).

The task BADINTERRUPTTRAP will investigate this situation more thoroughly. Consult the program listing.

Note 6: On machines with "device vectors" of some kind, e.g. the PDPl1, daisy chains to identify IO-interrupts are superfluous.

5.7. Basic exchange of IO-messages, IOREQUESTs.

All data transfers to/from external devices are incorporated in a common formalism. The basic data structure in this scheme is an IOREQUEST-object.

When an IO-transfer is requested by a task, the task will fetch, initialize and release an IOREQUEST-package to the NEWREQUESTS-semaphore in the relevant DEVICE-object. The associated IO-driver inspects the package, performs the requested actions, and releases the package to the supplied RECEIVER-semaphore.

A TTY-input driver for a full-duplex TTY will, of course, live a more independent life. It will attempt to read new input lines before they really are requested by some task.
A SHARED(IOREQUEST)-object looks like:

- Semaphore on which a waiting process (if any) hangs.
- Actual device

<table>
<thead>
<tr>
<th>NEXT</th>
<th>CREATE</th>
<th>RECV</th>
<th>SYSTEM data in SHARED-objects, HEAD-field.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CREATOR</td>
<td>RECV</td>
<td>IOREQUEST-object, VALUE-field.</td>
</tr>
<tr>
<td></td>
<td>DEVICE</td>
<td>&lt;misc.info&gt;</td>
<td>REQ-subfield, 3 16-bits words (NORD-SM).</td>
</tr>
<tr>
<td>type-field</td>
<td>IO-request specification, according to type-field.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In MARY the REQ-field is declared as a "dynamic template", which corresponds roughly to an ALGOL68-union. See MARY TEXTBOOK, pp. 178-182.

The REQ-field will be filled in by the requesting task, and contains a datum of mode:

* BYTE, For papertape IO.
* CHAR, For TTY echoes.
* ROW PACK (VAL) CHAR, For normal formatted IO.
* REF [REF PROTEAN MEMADDR, SECTORADDRESS STARTSECT, INT LENGTH], For disk transfers.
* BITS, For error messages.

Standard I0-interface routines (like PUTIMAGE, GETIMAGE and DSKIO), being called implicitly via system routines, are responsible for user-administration of IOREQUEST-objects.

Three aspects of IOREQUEST-communication will be further commented on in secs. 5.8-5.9:

* How is interface between a TTY-input and a TTY-output driver organized?
* How do we allocate I0-buffers and SHARED (IOREQUEST)-objects?
* How is formatted I0 organized for normal tasks? How is interface with mass storage text-files established?
5.8. TTY-driver interface.

5.8.1. TTY-synchronization.

Cooperating input and output TTY-drivers must be synchronized. How we actually implement this synchronization depends on whether a TTY operates in full-duplex, half-duplex or simplex mode; and on how flexible we want the system to be, e.g. do we allow concurrent reading and writing?

We shall describe the RUNSYS solution for the most general case: a full-duplex TTY on which we may read and write simultaneously.

A TTY-output driver will write two kinds of texts:

1. Ordinary output-lines, prepared by the user.
2. Echo characters and special output-texts, prepared by the TTY-input driver.

Texts of type 2 will have priority over ordinary output lines of type 1. A change of output-text is only allowed when the current text is exhausted.

A minor problem occurs when printing echo-characters: They are passed one after another in separate IOREQUEST-packages, and the TTY-output driver must continue to print characters from the same input-line until end-of-line.

This can be accomplished, if the input driver RELEASEs a dummy IOREQUEST in front of the resource queue of the output driver's NEWREQUESTS-semaphore. I.e. this semaphore always "contains" the number of waiting output-lines. When ACQUIRED, such a dummy IOREQUEST directs the output driver to obtain a complete echo-sequence of IOREQUESTs from another semaphore, FILLED ECHOES. Then we return to the normal semaphore for further instructions.

The input driver must use a variant of the RELEASE-routine (described in sec. 3.3.4.6), which can put new resources in front of a semaphore's resource queue.

Per Holager contributed significantly to this solution.
5.8.2. Administration of IO-buffers and IOREQUESTs from TTY-drivers

Normally, an input- or output-text is represented by an IOREQUEST-object, pointing at a separate IO-buffer. Echo-texts usually don't have separate IO-buffers.

**IO-BUFFER FLOW DIAGRAM:**

- Circles indicate tasks; squares indicate semaphores.
- DEVIN and DEVOUT represent pointers to a TTY-device pair.

TTYBUFFERPOOL and AVAILABLEECHOES are global semaphores. The others are local for the actual TTY. Note that echo- and input-buffers (and corresponding IOREQUESTs) rotate in closed circuits. Output-text buffers, being variably-sized, are however dynamically created and destroyed.
5.8.3. INPUT-driver actions.

General remarks.

The total number of input-buffers in TTYBUFFERPOOL is determined on system generation, and is 2 per TTY. The buffers are fixed-sized. They are not trimmed to get rid of rightmost blanks, as this involves allocation of new buffers.

A local TTY-semaphore named BUFFERS has been introduced to prevent a single input-driver from monopolizing all available buffers. BUFFERS contains initially 3-4 IOREQUESTs. (Possible deadlocks, in spite of this, are resolved through a long-wait abortion of the actual user task.)

Driver actions:

1. If there are insufficient resources in TTYBUFFERPOOL/
   BUFFERS, we emit a 'WAIT'-message to the teletype. This
   keeps the user busy for a while!

2. When we have succeeded in getting a new TTY-buffer (after
   receiving the first character on a new line), we obtain an
   IOREQUEST-object from AVAILABLEECHOES, to use as a dummy
   resource to DEVOUT, NEWREQUESTS.

3. Then we start filling our buffer with input characters.
   If there are no more IOREQUESTs (to hold echo characters)
   on AVAILABLEECHOES, the last character is simply ignored.
   The "echo supply" in AVAILABLEECHOES is initially 4* (no. of TTYs

Special actions are carried out when deleting the last input
character/line, or when sensing a BREAK-command to kill the
actual user task.

Note: We assume that there exists a "conditional" variant of
the ACQUIRE routine, so that we don't get hung up automati-
cally, if there are no available semaphore-resources.
5.8.4. OUTPUT-driver actions.

These ought to be straight-forward:

The output-driver alternates between writing echo-lines (for full-duplex and simplex TTYs) and ordinary output-lines. The buffers of the latter ones are deallocated after the output operation. As for input, a local BUFFERS-semaphore regulates the maximum number of output buffers.

Three details:

1. On carriage-return, necessary line-feeds etc. are generated.
2. The driver keeps track of the number of characters printed on a line, and performs an extra line-shift if required.
3. No provisions exist for "overwriting" or for partial printing of lines without terminating carriage-returns. This is, however, not really difficult to implement.

5.9. Drivers for the line printer, card reader, paper tape punch and paper tape reader.

Driver routines for these devices are very simple, and resemble "degenerated" TTY-input or TTY-output drivers. They will acquire IOREQUESTs from <actual device>.NEWREQUESTS on what to do, and release them to the supplied RECEIVER-semaphore.

The card reader is presently available to any task, but will later be controlled exclusively by a spooling system.

The line printer is primarily used by a PRINTFILES-task to output printfiles of terminated batch tasks. It may also be used by system routines for debugging.

The line printer and the card reader both require special treatment by their driver routines to meet the short interrupt-response times. Consult program listing for further details.

The paper tape punch and -reader are free for anyone to use. This ought to be changed later.
5.10. The disk driver.

No disk-head optimization is implemented yet. For the time being, a plain FIFO-queue (automatically granted by standard ACQUIRE/RELEASE-routines) is used for disk requests.

A DISKIO-routine:

PROC SAVE RECURS (BOOL READACTION, REF PROTEAN MEMPTR, SECTORADDRESS DISKADDR, INT LENGTH)
  BITS DISKIO=EXTERNAL;

is called to interface the disk driver from system routines; i.e. FILESYS, loader, OLAY-data and OLAY-code routines.

5.11. The IOFILE-mode. IO-interface routines for formatted IO, PUTIMAGE and GETIMAGE.

5.11.1. The IOFILE-mode.

All formatted IO passes via an IOFILE-object, which administrates the current text buffer and establishes contact with the actual IO-driver.

Layout of IOFILE-objects (somewhat simplified):

<table>
<thead>
<tr>
<th>IOFILE-object:</th>
<th>Current text buffer:</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMAGE --------</td>
<td>THIS IS A TEXT......</td>
</tr>
<tr>
<td>CURRENT MAX</td>
<td>Buffer indices.</td>
</tr>
<tr>
<td>DEVICE</td>
<td>Buffer length=MAX</td>
</tr>
<tr>
<td>EOF flag</td>
<td>Controlling access of this IOFILE.</td>
</tr>
<tr>
<td>IOFILEACCESS-</td>
<td>TEXTFILE-field of mode SYMBOLICFILE.</td>
</tr>
<tr>
<td>semaphore</td>
<td>Used when interfacing mass storage text-files, see secs. 5.11.4 and 7.3.2.1.</td>
</tr>
<tr>
<td>corresponding</td>
<td></td>
</tr>
<tr>
<td>dummy resource.</td>
<td></td>
</tr>
<tr>
<td>Current file addr, DA</td>
<td></td>
</tr>
<tr>
<td>FILE</td>
<td></td>
</tr>
<tr>
<td>PREV</td>
<td></td>
</tr>
</tbody>
</table>

Associated device, if any.

Opened mass storage text file.

Previous "IOFILE" descr.

Etc.
5.11.2. **PUTIMAGE and GETIMAGE routines.**

Let the actual IODEVICE be named "RFILE".

When a user task executes "PUTINT (RFILE, I, W);", PUTINT will prepare an output string of W characters and call PUTTEXT to do the remaining write-operations.

PUTTEXT will, if necessary, call PUTIMAGE to output the current text buffer (RFILE.IMAGE), if RFILE.CURRENT+W > RFILE.MAX.

Thus, PUTIMAGE is the bottom-level IO-routine, called by user tasks. But, of course, PUTIMAGE only prepares an IOREQUEST-object and passes this to some IO-driver, which then initiates the physical IO-operations.

For TTYs, PUTIMAGE/GETIMAGE will acquire/release resources, i.e. SHARED(IOREQUEST)s, via the NEWREQUESTS-semaphore in the RFILE.DEVICE-object.

PUTIMAGE will normally not wait for an output-text to be output (see below).

Initially RFILE.CURRENT (position of next character to be written/read) is greater than RFILE.MAX, causing a line-shift in the first IO-call.

**PUTIMAGE actions.**

As mentioned, we must limit the number of output buffers.

a) For TTYs, the BUFFERS-semaphore in the associated DEVICE-object takes care of this. BUFFERS will usually contain 2 - 5 IOREQUEST-objects.

Text buffers are allocated as normal MEMLINK-objects. A buffer is trimmed and its OWNERShip adjusted, before it is coupled onto an IOREQUEST-object, being released to the output driver.
Only one output buffer, the current RFILE.IMAGE-buffer need to be allocated during user write-operations, if the filled ones already have been emptied and deallocated by the output driver.

Note, we may consider to share output- as well as input-buffers in TTYBUFFERPOOL, but this makes buffer-trimming difficult.

b) For output on mass storage text-files, only one text buffer at a time is allocated. When full, it is stored into the current file-buffer. When this disk-interface buffer has been filled up (with ≈10 text buffers), we must wait.

**GETIMAGE actions.**

They are rather trivial:

For TTYs, we acquire input texts from RFILE.DEVICE.NEWREQUESTS, and release them to TTYBUFFERPOOL/RFILE.DEVICE.BUFFERS afterwards.

For text-files, the operations are roughly the inverse of those for output, but special actions are carried out on VADD-commands, see later.

### 5.11.3 Standard input and output IOFILES.

Every user task has access to two standard IOFILES, named INFILE and OUTFILE.

A TASK-object contains three IOFILE-pointers:

```plaintext
IO.INPFILE
IO.OUTPFILE
IO.MISCIIOFILES
```

The standard IOFILES.

List of auxiliary IOFILES.

INFILE and OUTFILE are really macros. They will call some system routines, yielding CURRENTTASK.IO.

When a user task has been forked after TTY log-in, its corresponding IOFILE-objects are established.
Note 1. Children of a user task will automatically use their parent's I0FILES as their standard I0FILES. That is, we must synchronize. For this purpose, there exists an I0FILEACCESS-semaphore in all I0FILE-objects. The calling hierarchy of I0-routines is centralized so that GETCHAR does the synchronization for input, and PUTTEXT (and PUTPTEXT) does the synchronization for output:

Ex.

Output call hierarchy. Input call hierarchy.

```
PUTREAL
|   | GETREAL
|---|---
|   |
PUTINT
|   | GETINT
|---|---
|   |
PUTCHAR
|   | FIRSTNONBLANK
|---|---
|   |
PUTTEXT
|   | GETCHAR
|---|---
|   |
PUTBASICCHAR
|   | GETIMAGE
|---|---
|   |
PUTIMAGE
```

This set-up ensures that e.g. CURRENT- and MAX-fields are updated consistently.

Note that we synchronize on text items, not on text images. That is, if several tasks concurrently execute:

```
100:=:I;
PUTPTEXT(OUTFILE,'I=v');
PUTINT(OUTFILE, I, 5);
NEWLINE(OUTFILE);
```

on the same OUTFILE, the resulting output is completely interwoven. However, the strings 'I=v' and 'v100' are not split.
Note 2. All formatted IO goes via the same IO-routines (PUTTEXT, etc.) in the same I0FILE-formalism; regardless of whether it is performed by system tasks or user tasks, or whether it is performed by user routines or system routines.

Ex.
If a system routine, e.g. a file system routine, discovers an error situation, it can immediately write out an error message which is directed to the user's OUTFILE; instead of returning some cryptic status-word back to the requesting program.

Ex. The TASKSINK routine always redefines its standard I0FILEs to those of the task getting killed. Thus, forthcoming termination-messages will get out exactly where we want them to

Note 3. In order to simplify common IO-calls, some additional routines and operators have been introduced:

Ex.
OUTINT(I, 5);  \% means PUTINT(OUTFILE, I, 5);
I \Rightarrow 5;  \% means the same.
'I\Rightarrow v';  \% means PUTTEXT (OUTFILE, 'I\Rightarrow v');
'\Rightarrow';  \% means NEWLINE (OUTFILE);

Consult the technical documentation [MARY group, 1974] for further details.

5.11.4. Interface with mass storage text-FILEs.

Although ordinary files on mass storage have not yet been treated (see ch. 6), we shall comment a bit more on interfacing formatted I0FILEs with standard MARY text-FILEs.

For tasks that are not associated with an interactive terminal, their IO must (should!) go via mass storage.
Examples: Tasks administrated by a spooling system; tasks being "started" by another task and having their job control images on mass storage text-files.

The JOBINTERPRETER of such tasks will put non-empty FILE-pointers into their standard IOFILEs, so that GETIMAGE and PUTIMAGE routines are able to direct I0-requests to the proper devices. When a new text buffer is to be transferred, a call on the system routine IMAGElO will be performed. IMAGElO will convert a normal text buffer to the special text representation on mass storage, or vice versa. IMAGElO is also used by the editor.

Generally, we may introduce auxiliary IOFILEs, in addition to the standard ones. Such IOFILEs are linked into the MISClOFILESLIST of the task when created. They always establish contact with a mass storage text-file:

Ex. Directing debug data to a special text-file.

    REF IOFILE AUXFILE=CREATEOUTFILE ( 'F.BIGFILE', <size> );
    PUTPTEXT(AUXFILE, 'DEBUG DATA:');
    ------

A similar facility is used by compilers to read/write on non-standard IOFILEs, cf. U1100's CSF('\WADD---')-function.
6. The MARY_RUNSYS file system, FILESYS.

6.1. Features of a basic file system.

The basic file system on a machine constitutes an important part of the operating system. Its structure has consequences for the entire system. Keywords: multiprogramming properties, utilization of mass storage, file access times.

Requirements of a basic file system:

* Facilities for allocation and deallocation of raw storage, temporary or permanent, linked or contiguous, on one or several device types. Named subfiles should be permitted.

* Protection mechanisms: Classic synchronization and deadlock-detection between multiple users, security mechanisms against software and hardware errors (e.g. backup), and against unauthorized usage (e.g. access keys).

* User-callable interface-routines, e.g. for treating symbolic text files. How "high-level" such functions might be, before we have introduced a second-order file system, is a matter of definition.

* Satisfactory utilization of mass storage, channel capacity, CPU, and primary storage.

* Modular, orthogonal design in order to ease programming, testing, maintenance and enhancements.

Of course, some of these requirements are slightly contradictory, e.g. access time versus utilization of mass storage. The need for a very general file system may also be less on a small machine than on a large one. That is, static contiguous files may suffice on a mini-computer.
6.2. Design criteria for the basic file system in MARY RUNSYS.

6.2.1. Recursive files. Subfiles.

Our most scarce resource has been (and still is) programming manpower. A simple recursive or hierarchical file structure was therefore adopted: A file consists of a table of contents (dictionary) plus the subfiles contained, and so on.

The bottom-level files have no dictionaries; they are so-called atomic files, in contrast to their compound parent files. An atomic file may accommodate a symbolic text, a relocatable program, or it may be used purely as a raw data-file by some second-order file system.

The file names of the different levels are separated by '.' (dot), like in 'F.A.X'.

Example.

![Diagram of file system hierarchy]

The files F and F.A are compound, i.e. they primarily consist of dictionaries. F.B, F.C, F.A.X and F.A.Y are single atomic files.
Advantages of this type of file hierarchy:

* **Naturalness.** Many applications demand a tree-like organization of mass storage. Think of public versus private program files in programming projects.

* The tree structure implies **generality** and **simplicity.** We save a lot of coding work by using identical routines for all levels on the tree; e.g. the same routines can be used to maintain the master file's catalogue, as well as dictionaries of ordinary user files.

One of the reasons for implementing a general hierarchic file structure was the striking **unorthogonality** in the file system on U1108. Here we have data files (containing symbolic texts), program files (containing "elements" which may be symbolic texts or translated programs) and raw, atomic files. In addition there may exist several "versions" of a file or an element. At least 4 different formats for symbolic texts are used. And so on. - However, don't misunderstand me. U1108's file system exhibits many valuable properties, particularly for program development.

* **System-supported named subfiles** eliminate a lot of error-prone storage administration from user programs. Packing to release local file **space becomes an easy task.** Hierarchic, desentralized subfiles also reduce the average size of file dictionaries. E.g. the master file on the MASTER op.sys. on CDC3300 at the Univ. of Oslo contains about 10,000 entries, because the system doesn't allow more than one file level.

* **Synchronization** of files is done at the most local subfile level. That is, we don't have to lock the entire file F when reading from subfile F.A. Indeed, we may very well read from F.A, write on another subfile F.B and allocate a new sub-file F.C - simultaneously. Compound files are only locked temporarily during dictionary updates, see sec. 6.5.

This **selective locking of files** yields very good multiprogramming performance. Furthermore, for "read-only" files (such as symbolic text-files) **several** tasks may safely access them at the same time.

U1108's file system, for instance, is hopelessly inefficient in this area  [UNIVAC, 1975].
* The hierarchic structure makes it easy to separate the logical file functions from the underlying space allocation principles (a parent file provides space for its subfiles). In fact, conditional compilation of ≈15% of the source text decides whether linked or contiguous file allocation should be used.

6.2.2. File privacy and other features.

Our ambitions on other aspects of the file system are considerably less. Once we have a small, clean system with good multi-programming performance, we're happy.

I can only hope the system is modular enough (cf. the above 15%-figure) to permit implementation of access keys, private versus public files, external devices with different sector lengths, and so on.

6.3. Implementation of FILESYS.

6.3.1. Organization of mass storage.

Two alternatives, called A and B, have been implemented. In both alternatives, a file dictionary is organized as a linear array of dictionary entries. No hashing is attempted, since the number of subfiles is expected to be ≈100, and since dictionary searching only is performed during allocation and opening of files. (On SM4, a disk sector can room 11 dictionary entries.)

6.3.1.1. Contiguous static files, alternative A.

A file, and all its subfiles, is allocated in a fixed number of consecutive storage locations (usually sectors):

```
F:  'A' | 'B' | 'B' | -- | F.A | F.B | F.B
```

A star (*) indicates a deleted subfile.

The size of a file dictionary is pre-set to about 1/30 of the total file size.
All file addresses can easily be made relative to the start of the file itself (or the parent file). This makes it easy to copy or to pack files; we simply shuffle the actual file sectors unchanged from one place to another.

Evaluation of method A:

Internal file access will be fast due to contiguous space allocation.

We will, however, get trouble with utilization of storage. The only way to free the space of a deleted subfile is to pack the parent file. This is a bothersome operation, especially when applied to the master file of the entire disk or drum!

We may reduce the demand of packing by allocating some extra sectors at the end of an atomic subfile, in case of later expansions. Likewise, we may allow renaming of a deleted subfile into a new subfile, if the space requirements are compatible. Furthermore - as a compound file has a dynamic upper tail - we may use this area as a scratch area first, and later "freeze" it into a new subfile of correct length.

Nevertheless, the staticness of compound files is a nuisance, and temporary files (like large printfiles) don't fit very well into this scheme.

6.3.1.2. Linked (scattered) dynamic files, alternative B.

Here, all subfiles of a file are mapped onto logically increasing sector addresses. The physical storage is organized in track units, which contain a power-of-2 sectors. The available tracks are administrated by a global bitmap, which is small enough to be memory-resident. A typical track size will be 16-64 sectors.

Ex. A disk containing 6500 sectors, a track size of 16 sectors, and a wordlength of 16 bits implies a bitmap of =25 words. - Peanuts!
A sector address on a linked file can be split into:

```
track no | rel. sector no.
```

A compound file must contain a table of track addresses, to be used for address transformations.

In order to have a dynamic upper file-limit on all file levels, only the dictionary of a compound subfile can reside physically together with atomic subfiles on the same level. The real contents of its subfiles must be allocated somewhere else, so we get a "free" file tail.

Example.

```
F's dictionary, G's dict.

F.A, only a dictionary F.B* F.B

F.A.X F.A.Y

Parent file of F, e.g. the master file.

Etc.

Subfiles of F. This is the real file area of F.

Subfiles of F.A.
```

Subfiles (dictionaries, atomic files), that physically reside within the parent file, are statically allocated. Only the tail of a parent file is dynamic, cf. dotted rectangles above.

The basic dictionary entry and the track address-table of a compound file are stored in consecutive locations in the parent's dictionary. Thus file entries for compound files are variably-sized:

Dictionary of parent file of F and G:
(not shown in above example)

```
F's track table

G's track table
```

File entry for F. File entry for G.
Evaluation of method B:

The main and important feature is the dynamic space allocation. We may also diddle with the global track map to "hide" damaged disk areas.

The tricks mentioned under method A to avoid file packings, are also valid here.

Drawbacks: (possibly) longer file access times, and larger space demand for opened files.

6.3.1.3. Some remarks on alternatives A and B.

The two sketched file implementations both assume that the master file - the root of the file tree - is treated a bit differently, as it has no proper parent file. A dummy dictionary, containing only the master file entry, must be placed somewhere on the disk/drum, e.g. in one of the lower "system sectors":

As mentioned, conditional compilation of FILESYS is used to select between contiguous or linked files. Usage of file routines from normal programs is unaffected by this change.

For contiguous files, necessary file space is allocated at file creation-time.

For linked files, the space of a compound file increases and decreases dynamically (see also sec. 6.3.3). When a new subfile is allocated, the parent file may be expanded - if there are sufficient free system tracks left. When the parent file is deleted or packed, possible vacant tracks are released to the system.

The existing RUNSYS-version uses linked files.
6.3.2. File structures in memory. The FILE-mode.

When using a file, some basic file information must be brought into memory. The necessary data objects are allocated and initialized on file opening.

The mode of a dictionary entry is called FILEHEAD, and contains file type, size, name, relative address in the parent file, number of subfiles, etc..

For every opened file there exists a FILE-object in memory, containing a copy of the FILEHEAD-object and a pointer to a copy of a possible track table. In addition, it holds some temporary control information (an access semaphore, parent- and child pointers a buffer pointer).

Memory structure:

```
FILE-object.
  Copy of dictionary entry.
  FILEHEAD-object, 16 words on SM.
  FILETRACKS
    Control information.
    Track addresses
      Track table, if the file is a linked compound one
    RBUFF
      Current file Buffer
      FILEBUFFER-object; containing the last file-sector being accessed, its sector address and a read/write flag. Partly user-controlled for atomic files.
```
The data structure for the set of opened files constitutes a tree:

FILE-objects:

Every task possesses a list of files, being opened by this task. The list is chased on task termination to close files that are still open (temporary files will be deleted). The list elements are small FILEINFO-objects. List elements of closed files cannot be reused, see figure below.

Declarations:

```plaintext
MODE
FILEHEAD=[([10] PACK CHAR FILENAME,-----)],
FILE=[FILEHEAD HEAD, -------], % See below.
LIST(M)= [REF LIST(M) NEXT, M VALUE],
FILEINFO=LIST(REF FILE), % List element.
TASK=[---, REF FILEINFO OPENEDFILES, STANDARDFILE,---];
```

Data structures:

RUNNING task:

Some other task:

A truly shared FILE-object.
All accesses to an opened file go via FILEINFO-objects. A NOOFUSERS-attribute in FILE-objects keeps track of the number of incoming pointers from FILEINFO-objects. This counter is manipulated through OPENFILE- and CLOSEFILE-calls, see below.

**FILE-mode, main attributes:**

```plaintext
MODE FILE= [FILEHEAD HEAD, % Dict. entry.
  ROW (0) TRACKADDRESS FILETRACKS, % Possible track table
  REF FILE PARENT, CHILD, SIBLING,
  INT NOOFUSERS, % No. of open calls
  SHARED (REF FILE) CONTROL;
  INT NLOCKS,
  REF TASK OWNER,
  <buffer, misc.>];
  } Needed for synchronization.
```

When a file is requested opened - by OPENFILE (<parent>, <file name> ) - the tree structure of the already opened files is first scanned. If the file happens to be open, only a new FILEINFO-object is created and NOOFUSERS incremented by 1. Otherwise, a search in file directories on mass storage will be required. A new FILE-object, file track table and -buffer (for COMPOUND files) will be prepared; if the sought file really exists.

Note that the parent file must be previously opened, and that a user can perform additional OPENFILE-operations on the same file.

CLOSEFILE will inversely deallocate the associated file objects (except the FILEINFO-object), if NOOFUSERS has been decremented to zero.

Note, no locking operations on the file semaphore is performed on file opening!

A user program is only allowed to perform explicit reading/ writing on atomic RAWDATA files, see secs. 6.4-6.5.
6.3.3. File types.

Possible file types: COMPOUND, SYMBOLIC, RELOCATABLE, ABSOLUTE, MISCTYPE, RAWDATA.

Allocation states: CONTIGUOUS, SCATTERED.
Duration types: PERMANENT, TEMPORARY.

Subfiles of TEMPORARY files cannot be PERMANENT.

Atomic subfiles of CONTIGUOUS files cannot be SCATTERED, and vice versa. The "SCATTERED"-specification is ignored, if we are using allocation method A. If we are using allocation method B (linked files), a CONTIGUOUS compound file will be statically allocated in consecutive tracks on mass storage.

Files used for OLAY-code and OLAY-data ought to be CONTIGUOUS for reasons of efficiency.

6.3.4. File names.

The FILENAME may consist of max. 10 characters. It is blankfilled to the right if it contains less than 10. RELOCATABLE and ABSOLUTE files have '$' and '$$', respectively, as suffixes in their file names.

Ex.

'PROGRAM ' Name of symbolic program.
'PROGRAM$ ' Name of compiled program.
'PROGRAM$$ ' Name of link-edited program.

The first 6 characters in a task name are automatically used as file name of a standard prefixing parent file. This parent file is accessible via a STANDARDFILE routine. Its file name is set by a BEGIN-command, see sec. 7.1 on the job control language.
6.4. User interface routines.

When using the file system, the programmer should include a library *prelude* in his program. The prelude contains declarations of modes, macros, operators and routines for file operations.

6.4.1. Technical security when using files.

It is desirable to have pointers to FILE- or FILEINFO-objects from user programs, so we don't have to deal with symbolic file names all the time. (Special "file indices" into local task tables may cause ambiguities. - How should child tasks distinguish between file indices of its different parent tasks?) However, general pointers into system-controlled data objects might cause disaster, if the user doesn't do what he is supposed to!

At this point, MARY offers full protection against faulty data accesses. The file prelude declares a mode USERFILE:

```
MODE FILEINFO, % A modal, an undefined mode.
USERFILE=REF VAL FILEINFO; % Note the VAL!
```

Thus VAL-status will prevent users from modifying a FILEINFO-object. (And they don't know the inner structure of it anyway.)

Since all usage of file pointers in normal MARY programs is on the REF-level (i.e. no de-refing), the incompleteness of the FILEINFO mode does not affect compilation of programs.

Elegant, isn't it?

6.4.2. Available file macros.

The following macros give access to FILESYS routines:

Let <file> denote a USERFILE-variable.
* OPEN (<parent file>, <file name>).

If <parent file> is NIL, the STANDARDFILE of the actual task is first substituted; thereafter the master file, if the requested file still cannot be found. This searching strategy is credited to [Burroughs, 1969].

If the <file name> is composite (like F.A.X), prefixing unopened files (F and F.A) will automatically be opened, if needed

* CLOSE (<file>).

If NOOFUSERS=1, then all CHILDren of this file must have been closed previously. A task cannot close files, being opened solely by other tasks (parents, children). The last file buffer is output, if necessary. Temporary files will also be deleted.

* ALLOC (<parent file>, <file name>, <misc.types>, <no. of sect.>.

If the <parent file> is NIL, then either the STANDARDFILE is used, or some prefixing parent file in case the <file name> is composite. An existing file with the same file name will be deleted, if possible. The allocated file is automatically opened.

* DELETE (<file>).

NOOFUSERS must be 1 and all CHILDren closed. As for closing, the actual file must have been opened by this task.

* COPY (<file1>, <file2>).

Copies the contents of <file1> over to <file2>. The types of the two files must match. NOOFUSERS of <file2> must be 1 and its CHILDren closed. COPY is not available to user programs, only the JOBCONTROLLER. COPY will not copy the contents of possible subfiles of <file1> that are COMPOUND ones, as this implies recursion.
* COMPRESS (<file>).

Means COPY (<file>, <file>). I.e. self-copying implies packing. Regrettably the reserved MARY-word "PACK" cannot be used.

* INFO (<file>). Prints some file information.
* CONTENTS (<file>). Prints the subfile table.
INFO and CONTENTS are only available to the JOBCONTROLLER.
* LOCKFILES (<ROW of files>),
* UNLOCKFILES (<ROW of files>).
Locks/unlocks the files supplied for exclusive use. The files must be opened by this very task. Only allowed for RAWDATA files, if called by user routines.

* GETVALUE
* PUTVALUE(<file>, <ptr. to FILEADDRESS-variable>, <memory ptr.>, <length>).

Transfers data to/from a given file, being locked on beforehand. Thus, user routines can only do IO on RAWDATA files.

The file is simulated as a (kind of virtual) word-addressable array with base zero.

A FILEADDRESS is a tuple: (sector address, relative offset), in order to override the hardware-defined range of ordinary integers (±2¹⁵).

In the offset exceeds DISKSECTORSIZE-1, it is automatically truncated and the sector address incremented.

After the file transfer, the offset is incremented by the <length>-parameter. Nothing is written out in the buffer by such a call.

The requested file-slice may cross sector boundaries.

A PUTVALUE-call with sector address = (1-) will empty the current file buffer.

Note 1: The GETVALUE-macro is potentially UNSAFE, but user programs may (presently) access it.
Note2: File buffers must be explicitly allocated (possibly by system interface routines) for atomic files, cf. the next two macros.

* ASSOCIATEBUFFERANDLOCK (<possible old buffer>, <file>),
* DEASSOCIATEBUFFERANDUNLOCK (<file>, <buffer deallocate option>).

These two macros are used when performing I/O on atomic files. Their functions ought to be self-explanatory (see below example). They are needed to ensure private file buffers when several tasks share an opened atomic file. E.g. the VADD-command will often read common preludes from shared text-files.

Remark: Our motive behind implementing a primitive virtual memory facility for files (the word-addressable function) is to prevent user- and system programs from duplicating identical buffer-management routines.

Ex. Simple user program, doing file operations.

BEGIN
   WADD,M M*ARY.FILEDECS % WADD-facility on U1108.
   USERFILE F =ALLOC (NIL,'A',(RAWDATA,---),25);
   MODE DATA=[[10] PACK CHAR NAME, INT NO,---];
   FILEADDRESS ADDR:=(0,0);
   ASSOCIATEBUFFERANDLOCK(NIL,F);
   TO 100
   DO
      REF DATA PTR =<-->
      <DATA-value>:= (PTR);
      PUTVALUE(F,ADDR,PTR.OPERANDSIZE DATA);
      --- Yields the size of a DATA-objec
   OD;
   DEASSOCIATEBUFFERANDUNLOCK(F,TRUE);
   CLOSE (F);
END FINISH

At last, the highlights of MARY FILESYS.

Our hierarchic file structure, with possibilities for selective locking of subfiles, eliminates the majority of common synchronization conflicts.

Most file operations are also unary, i.e. only one file at a time needs to be locked. Example: (temporary) locking of a parent file during allocation of a new subfile. However, files can be locked explicitly by the users, thus causing deadlocks.

The following ordering principles apply to file lockings:

* A parent file must always be locked before its subfiles, etc.

* Compound files on the same subtree-level are sorted after some unique criterion, e.g. memory address of FILE-objects, and locked in this order. Atomic files are locked after all compound files have been locked, and in a similar order.

Thus, tree-level is the most significant sorting criterion; memory address comes next. User programs cannot lock compound files.

Hence deadlocks can never occur in file lockings, cf. [Habermann, 1969]. But, of course, if we consider a larger set of resources (including memory space), the situation is not deadlock-free.

The file locking itself is done in the normal way, by ACQUIRE-calls on the file semaphore.
Unlocking of files can be done in arbitrary order.

If several LOCKFILES-calls are made by a task, the currently locked files of this task must be temporarily unlocked. Then old and new files are re-sorted to establish a new deadlock-free locking sequence.

This possibility of implicit unlocking and locking of already locked files is the reason why user programs cannot lock compound files. Thus file updating routines are free to lock such files - one at a time, parent files before their subfiles - without causing deadlocks.

The only (system) routine that needs to lock more than one file simultaneously, is the COPYFILE-routine, performing a binary file operation. Here a full LOCKFILES-operation on the two files supplied will be required.

Note that safe look-up of subfile-objects in the tree of FILE-objects is granted automatically, if the actual parent file is locked. No separate locking of the file tree, as such, is needed.

Also note that a file must be explicitly opened by a task - thus incrementing the NOOFUSERS-attribute of the file - before this task can perform locking, closing, deleting, etc. on it. This guarantees that a file, opened by some parent task, cannot by closed (thus loosing its FILE-object) by a naughty child task.

As mentioned in sec. 3.4.3, we may get trivial deadlocks if a task attempts to lock the same file several times. For instance, ALLOCFILE calls OPENFILE; both are directly callable by user programs and both will try to lock the actual parent file.

That is, our synchronization operator must be insensitive to nested lock calls. We have therefore introduced a NLOCKS-attribute in FILE-objects to keep track of the arithmetic difference between lock and unlock calls. Lock and unlock operations will be dummy operations, unless <file>.NLOCKS is stepped from 0 to 1, or from 1 to
Another FILE-attribute, OWNER, is added for the sake of efficiency to eliminate searching among acquired task resources.

Lastly, let us mention that file synchronization in other systems might be organized after the "concurrent readers and writers"-scheme in [Courtois et al., 1971]. Because of our subfile structure and "virtual file" facility, this multiple-semaphore solution is neither required nor applicable. File lockings in our system are absolute; no distinction is made between the different operation-types (reading, writing).

Furthermore, most files (e.g. SYMBOLIC and RELOCATABLE ones) are read-only after initialization. Since no other files can be locked during read-operations from read-only files, we don't have to go through a complete LOCKFILES-operation (to avoid deadlocks) when using such files. A read-only file needs only to be locked temporarily during the read-operation.

Remark: If the editor or the compiler wants to substitute the entire contents of a read-only file by a new file-contents, these processors must require that the actual file is not opened by some other task already (cf. the COPY-operator for files). During the following over-write operation, the file-type is temporarily altered to RAWDATA.
7. MARY RUNSYS JOB CONTROL LANGUAGE. Local JOBINTERPRETER-tasks.

7.1. On JCLs. The MARY RUNSYS JCL.

Any general operating system seems to support its own job control language, of varying complexity and weirdness. Existing JCLs are seldom standardized, portable or easy to learn or use.

Our approach to the subject has been a very pragmatic one. We want a flexible, straight-forward system, which is cheap to implement and oriented towards interactive use.

We have chosen to implement a MARY-like JCL:

* Commands are implemented as _operators_ in the MARY sense (no priority, expression language). Each JC-operator will activate a corresponding system routine (e.g. the compiler). Indeed, why should proc calls look differently in JCLs than in ordinary programming languages? Analogy: an operating system may be regarded as the "implementation" of a JCL.

* We have _free format_ on JC-commands, and semicolon is used as a separator. The VADD-command is an exception.

* Command images should start with a 'V' (or 'Œ'). The GETIMAGE-routine will enter EOF-state when reading an input image with 'V' in column 1, unless we are in "JCL-mode".

Ex. MARY JCL-program.

```
V (REIDAR, <SAVE-space limit>, ---)  
% 'REIDAR' is the name of the  
% prefixing STANDARDFILE. The  
% files below are usually subfil  
% of 'REIDAR'.
V begin                                    
V F alloc (100, compound);                
V F.A alloc (5, symbolic);               
V B=:F.A;                                 
V F info contents;                       
V F.A ed F.A;                             
{ editor commands                        
```
V F.A marycomp F.R link
V F.PROG load;
} input-data to F.PROG
V F pack contents;
Vadd F.ACTIONS
V REIDAR.F.JOBDECK start (PRINTFILE,---);
Vend;

Comments:

A. Operators requiring more than two operands, must get their operands in form of displays: (−, −, ···, −).

B. Default values presently constitute a problem. Maybe a general macro facility will give the necessary flexibility?

Observe that the compiler, linker and loader automatically will add suffixing ' $' -characters in file names of compiled/linked programs. I.e. the real file name of ' F.R' above is ' F.R$ ', which should be used in ' =:' - and DELETE-commands.

C. The expressive power of the current JCL is very limited: No nesting of expressions; no declaration facilities; no control structures to permit looping or branching.

Idea: Why not let our JCL be an interactive and interpretive variant of MARY? (Proposed by Mark Rain).

In other words, there is plenty of future work to do in this area.

7.2. The JOBINTERPRETER.

This is a routine being executed by the user's bottom-level, JOBINTERPRETER-task, being forked after TTY log-in. It chews syntactic items from input images, decodes JC-operators and calls requested system routines: FILESYS routines, the editor, misc. compilers the linker, or the loader. The (text−) operands of JC-operators are accessible by the called routines.
We need only add a new operator-name and -description, when we want a new JC-function.

The JOBINTERPRETER will execute the "called" routine as a separate process, to better control stack size and handle error terminations:

Ex. Compilation, activated from the JOBINTERPRETER.

COMPILE-task

A forked child task to run the compiler.

JI-task

Basic JOBINTERPRETER-task, forked on TTY log-in.

When the child task terminates (normally or abnormally), the JOBINTERPRETER-task is woken up and a new JC-call is prepared.

Maybe we should allow several children to be executed in parallel, but I doubt it's worthwhile.

7.3. Miscellaneous JC-functions and related subjects.

7.3.1. The VADD-facility.

Although this function belongs to the general IO-functions (the GETIMAGE-routine carries out the actions), it is natural to describe it here.

In order to parameterize input as much as possible, we have plagiarized an U1108-like VADD-mechanism: Parts of an input text may be read from specified mass storage files, being opened and closed automatically. VADD-commands may be nested.

Ex. Input text or file:

```
begin
Vadd F.PRELUDE
list (int) h:=nil;
end;
```

Expands to contents of F.PRELUDE:

```
mode list(m)==;
Vadd M.ARY.PRELUDE
```

F.PRELUDE:
This mechanism is well-suited for implementation of MARY preludes, being common source texts (declaration skeletons).

Implementationally, a stack is needed to represent recursive ADD-invocations. The stack is represented as a linked list of special file descriptions.

**Layout of file descriptions:**

- Previously pushed file description
- Current file description

```
File addr., DA
File ptr., FILE
Prev. descr., PREV
```

Objects of moc SYMBOLICFILE.

The original file description resides within the IFILE-object of the task's INFILE; the others must be dynamically created and destroyed. The PREV-pointer in the INFILE-object selects the most recently pushed file description:

**Ex. Relevant data structures from previous example.**

```
Current:
INFILE, original

TASK-object: (for TTY):
```

Stack element for F.PRELUDE: for M.ARY.PRELUDE, being the current one:

When the current mass storage file is exhausted, its PREV-candidate is investigated, etc.. If this is empty and we don't have input from TTY, we enter EOF-state unconditionally.

For obvious reasons, 'WADD' must stand in cols. 1-4.
When reading from text-files, a private file buffer for this task is allocated by GETIMAGE. It is used to hold the current disk sector of the actual text-file, i.e. file buffers of stacked files are re-used and temporarily overwritten.

Ex. Sketch of relevant GETIMAGE-actions.

```
REF FILEBUFFER B:=NIL;
USERFILE F = INFILE.TEXTFILE.FILE;
----
DO
   ASSOCIATEBUFFERANDLOCK (B,F) := B;
   IMAGEIO (---);
   DEASSOCIATEBUFFER
   ANDUNLOCK(F,FALSE);
```

Note: The current text-file(F) is only temporarily locked. Other tasks are free to use it between our locking periods.

7.3.2. The editor (ED).

7.3.2.1. Editor functions.

An editor, originally written by two students [Grodås, Gulbrandsen, 1974] as a project thesis, has been adopted by RUNSYS. The editor has been extensively revised and extended by the author.

Some editor features:

* The editor offers traditional commands for manipulation of lines and character strings:

Position-, insert-, delete- and modify-commands;

Lines can be copied, moved and split;

Character strings can be located in the text.

* The current text position is a pair (line, column). Thus we may, for instance, easily delete cols. 57-70 in the next 10 lines.
* Old line numbers are kept by default (the described LNO-attribute)

When the position of a line is altered, it is automatically given the same LNO as that of a preceding old line, plus a relative SUBNO. We can also explicitly renumber all text lines.

Ex. Assignment of line numbers.

Old lines: Inserted lines: Renumbered lines:

\[
\begin{align*}
1.0 & \quad & 1.0 \\
2.0 & \quad & 2.0 \\
3.0 & \quad & 3.0 \\
4.0 & \quad & 4.0 \\
\ldots & \quad & \ldots \\
\end{align*}
\]

Deleted Ones
\[
\begin{align*}
1.1 & \\
1.2 & \\
1.3 & \\
1.4 & \\
\ldots & \\
\end{align*}
\]

* Editor commands are made simple, and can be orthogonally combined. Line numbers and repetition counts are distinct; the latter ones always start with a '+' or '-'. (The delete command, D, represents an exception: Dw means to delete w columns from the current position.)

More than one command per line is permitted.

* A simple DO-construct for repetition is implemented. 'DO' can be replaced by '('; a nested 'DO' by '('. Rightmost 'OD'-symbols can usually be omitted. '$$' and '$$$' can be used to specify unlimited repetition counts, until end-of-text and end-of-line respectively.

Some more compact notations for the most-used repetition constructs ought to be implemented.

Ex. Simple editor commands.

F'MARY' Locate the text 'MARY'.
D'AB';**' Delete 'AB', insert '***' instead of.
C/AB/**/ The same, but not unless 'AB' is four
5 'NEW LINE, NUMBER 5.1' Insert a new text line after line no. 5. ('5' is a position command.)
0 DO +10 P OD
0 ( $ P D
6.5,3 D2
+3
1 2 M4
1 2 A^4
0($((%$ C/*/!)/
0($,2 (((10 C/*/!)/

Print the 10 first lines.
Print and delete all lines.
Delete cols. 3-4 in line 6.5.
Position 3 lines forwards.
Move lines 1-2 after line 4.
Add a copy of lines 1-2 after line 4.
Substitute all occurrences of '*' by '!'.
Substitute all occurrences of '*' by '!', in cols. 2-10 only.

7.3.2.2. Editor implementation.

A list of line descriptors or entries will reside in memory.

Format of line entry-object:

```
SUC
file address
```

Successor line entry-object.
Rel. file addr. of virtual line object.

The list is allocated in an array of line entries:

An extra scratch table is allocated to hold possible text expansions. Default size for this table is 100 entries, but this may be altered by the user. Vacant line entries are linked in a free list.

The OLAY-data facility is used to access the real text lines. Before starting the editor; input-, output- and scratch-files (for OLAY-data objects) must be established.
Note 1: Relative positioning in the text is much, much faster than absolute positioning according to fixed line numbers, as we only have to chase memory-resident text-links in the first case.

Note 2: Since an editor is completely I/O-bound, we have not been afraid of spending CPU-cycles to save memory space. E.g. we use a singly linked list of line entries.

Maybe even a sequential array of file addresses could be an acceptable text representation, in spite of the update costs?

When an edit-operation is completed, the new text is written out in correct order on its destination file. If the file isn't big enough, the editor will discover this and assist the user in allocation of another file.

On NORD-SM, the editor occupies ≈7 K OLAY-code and ≈100 words SAVE-space. Each active user requires a data stack of ≈500 words, some text tables (1 word per line at best) and max. 2 file buffers.

7.3.2.3. Format of text (i.e. SYMBOLIC)-files.

A text line is stripped for leading and trailing blanks before storing. Some control information is inserted in front of the text string:

Line object on file:

```
bl | L | LNO | SUBNO | 'STORED TEXT'
```

Line head, 3 words on NORD-SM. Packed text

The head contains:

bl: no. of leading blanks.
L: length of stripped text.
LNO: main line no.
SUBNO: sub line no. \{ mainly used by the editor. \}

A blank line has L=1, bl=0.
Line objects are stored consecutively within a file, but can never cross sector boundaries. Thus (a copy of) a text-file can immediately be used by the editor as its virtual OLAY-data array.

File layout:

First disk sector                           Last disk sector

```
  | line   | obj       | obj       | ...  | obj       | obj       |
```

Total # of lines.                  An end-of-sector
mark, (L=0, b1=0).

A final eof-mark, (L=0, b1=0).

An average MARY source line has ≈25-35 characters after stripping. SM4's sector size of 176 words and wordlength of 16 bits indicate ≈176/(3+30/2)) = 9-10 lines per sector. I.e. we waste (only) ≈6% of the file space by not allowing lines to overlap sector boundaries. On U1108's FASTRAND-drum (sector size 28 words, 36-bit words), the figures are 4-5 lines per sector and 10% unused space.

As mentioned, there is one format for symbolic texts on external media in RUNSYS.

7.4. The MARY-MARY compiler.

Its structure, functions and usage are described elsewhere [Solberg, 1976]. The main implementor has been Ole Solberg, who has done a terrific job!

The generated code is in a so-called Binary Relocatable Format (BRF). It is stored on a RELOCATABLE file, being simulated "paper tape" on disk.

The compiler is a large beast on the NORD-SMs: ≈119 K OLAY-code and ≈2.5K SAVE-space (OLAY-code descriptors, SAVE routines, constant tables). User-dependent SAVE-data, data stack included, occupy from 8K SAVE-space and upwards. In addition, a shared 6K SAVE-space dictionary of standard MARY-symbols will be allocated, if at least one compilation is active.
7.5. The MARY loader.

The current NORD-SM loaders don't allow fixup of partial words, so we had to write our own loader; to work on a slightly modified BRF. This work has been carried out by Geir Green, after an original set-up of Mark Rain.

A highly desirable link-editor has not yet been implemented (see later). Our MARY-loader is a two-pass affair:

1. A preliminary pass to build dictionaries of entry points, external references and code/data segments.

2. A fixup pass to load fixup'ed SAVE segments into acquired memory locations and OLAY segments out on an olay file.

When the loader successfully has loaded a program, it will (UNSAFEly) construct a PROC variable describing the loaded program, and simply call this variable.

The stack size needed to execute the loaded program should be supplied when calling the loader: 

\[
\text{optional}
\]

Ex. (F.R,F.SCR) load (F.OUTP, <stack size>);

Run-time stack organization of user tasks during loading and execution phase:

\[
\begin{align*}
\text{Data stack} & \quad \text{created by JOBINTERPRETER when forking the loader task} \\
\text{Basic loader routine, performing some initialization stuff.} & \quad \text{The loader program is active.} & \quad \text{Back in basic loader routine, which will return to the JOBINTERPRETER.}
\end{align*}
\]
The loader's dictionary may get rather large, say 400 names & 10-15 words. It is segmented in the normal OLAY-data formalism, e.g. on the above F.SCR-file.

A user-description of the loader can be found in [Green et al., 1976].

On NORD-SM, the loader occupies ≈4K OLAY-code and ≈100 words SAVE-space. Each user will need temporary file buffers and misc. tables of ≈600 words totally, a stack of program-dependent length (the default value is 500 words), plus SAVE-space for the loaded programs.

Remarks.

Our two-pass loader is not very efficient, neither in terms of file access time, nor in terms of occupied file space. For instance, OLAY-code segments are stored in two versions on mass storage: as BRF-code, and as fixup'ed executable OLAY-code.

The current loader functions should be split into:

Link-editor functions: Build name tables, prepare code segments for loading.

Load-and-go functions: Allocate memory space, update and load relevant segments.

The mass storage format (BRF) for compiled programs should similarly be split into two parts:

```
Code and data segments, Name table, Fixup info. \} A RELOCATABLE file.
```

Now, OLAY-code segments are stored once on mass storage. They can be updated where they stand in the file in a reversible way, as we have a separate fixup table.

For MARY, it's natural to incorporate a mode checking facility in the link-editor; to verify correspondence between separately compiled modules. This is essential for the security of MARY programs, as a compiler only can check consistency within modules, cf. [Langmaack, 1973].

A draft of a mode checking link-editor for MARY is described in a recent Masters thesis [Sætermoen, 1976].
8. Miscellaneous facilities.

8.1. The MSECONDrask.

This process wakes up every M'th second to inspect system status.

Functions:
* MSECOND will check long-wait conditions.
All tasks possess a LASTTIMEACTIVE-attribute, being reset by the dispatcher. When a task (or a group of tasks) have not been active the last 5 minutes and are hanging on semaphores, we assume that we have detected a deadlock. The user tasks involved will be killed: They are removed from their respective semaphore queues, put into the READYQ and caused to wake up in abortive software interrupt traps.

* MSECOND will, if needed, increase the supply of small freelist objects, described in sec. 4.9, to avoid space shortage.

* MSECOND will also keep an eye on the amount of trashing and possibly reduce job priorities of critical tasks. (Max. 2 concurrent compilations?)
MSECOND may generally be used to monitor the system. It did, for instance, supply us with statistics of relevant system data during the test period.

Aftermath: The above functions have been incorporated in the IDLEVICEHANDLER-task to save task space.

8.2 General resource allocation. Job scheduling.

8.2.1 Resource allocation in RUNSYS.

Consider the following resource types:

1. Peripheral devices, like tape drives.
   Administrated by permanent, global SHARED-objects.

2. Raw, mass storage file space.
   Administrated track-wise by a global bit map.

   Administrated by dynamically created, global SHARED-objects.
Note: A shared OLAY-system-file is permanently allocated, thus eliminating most local swap-files.

4. Memory space.
Administrated by local/global MEMLINK-objects, local SAVELIMITs and a global AVAILABLESAVEMEM-variable.

5. Misc. temporary resources, possibly user-created ones.
Administrated by local (for the user's task tree) SHARED-objects.

6. CPU-time.
Administrated by local TIMELIMITs and clock intervention.

Disregarding group 1 and 2 -resources, the only non-preemptive resource type that deserves explicit deadlock-preventive actions by a global job-scheduler in RUNSYS is group 4, memory space - or more precisely: SAVE memory space. This is our critical resource, and the only one being considered (via the AVAILABLESAVEMEM-variable) before new user tasks are initiated. A general resource-allocation strategy to ensure safe, deadlock-free running of all processes is much too ambitious for our system, cf. [Habermann, 1969] and [Holt, 1972].

That is, a global job-scheduler in RUNSYS is (and may be) very simple; see later.

8.2.2. Batch and demand tasks. The IDLEDEVICEHANDLER-task.

Although most user activities are assumed to be carried out interactively, a backlog facility to run batch tasks will be needed. Such a batch task may be initiated from another task by a START-command, or its control images may be fetched from cards and then activated by a future spooling system.
Since the IDLEDEVICEHANDLER-task already was responsible for forking of new demand user tasks after TTY log-in, we decided to let IDLEDEVICEHANDLER also administrate the batch ones. Maybe "JOBMANAGER" had been a more appropriate task name now.

In other words: Demand tasks will be forked at once, if there is enough available SAVE-space; or the user will receive 'NO MORE DEMAND USERS ACCEPTED' and log-in is rejected.

Descriptions of future batch tasks are first linked into a backlog-queue. Task candidates are then selected from this, when sufficient SAVE-space becomes available.

Task priorities for batch tasks are lower than those for demand tasks (3 for batch, 2 for demand). Batch tasks do all their IO on mass storage files, and corresponding printfiles should be released to the PRINTFILES-task upon task termination.

Sketch of management of batch and demand tasks:
8.2.3. Backlog organization.

A backlog queue-entry contains essentially:

| INPFILENAME  |
| OUTPFILENAME |
| SAVE-limit   |
| Stacksize    |
| TIME-limit   |

The backlog queue will be scanned when a new queue element is inserted, and after termination of bottom-level user tasks. The first queued task candidates, whose SAVE-space requests can be accommodated, will then be forked as real tasks.

Comments.

No dynamic queue priorities have been implemented to prevent infinite backlog-ness.

Observe that a batch task may not necessarily succeed in obtaining its SAVE-space resources (in case they are grabbed as very large data objects), even if the task has been forked under this assumption. Cf. sec. 4.7.

8.2.4. The START-command.

A Job Control command, START, will activate a batch task.

Set up:

```plaintext
<read file> start (<print file>, <SAVELIMIT>,
<stack size>, <TIMELIMIT>);
```

Ex.

```plaintext
F.READFILE start (F.PRINTFILE, 2000, 400, 10000);
```

Later on, the system is expected to supply a printfile automatically. However, printfiles may grow unpleasantly large:

Ex. 100 line printer pages will require a printfile of \( \approx 700 \) sectors on our SM4, while the disk capacity is 6500 sectors.
8.3. Initialization and booting of RUNSYS.

8.3.1. RUNSYS loading.

A slightly modified version of the loader (in fact the original stand-alone loader), with stripped IO- and space allocation-routines, is used to load stand-alone programs, e.g. RUNSYS itself.

Initialization routines for RUNSYS will be placed in a memory area, later being overwritten by MEMLINK-segments. (These routines must be SAVE, as we cannot yet use the OLAY facility for code.)

Layout of memory after loading of RUNSYS on a 32K NORD-SM:

Increasing addresses

- \( \approx 12K \) -
  - Resident RUNSYS-routines and -data.

- \( \approx 3K \) -
  - System data generated during initialization.
  - Init. routines (transient ones).
  - Data stack for init. program.

- \( \approx 8K \) -
  - Loader's dictionary.

- \( \approx 3K \) -
  - Stand-alone loader.

The loader's dictionary is saved on disk for later use. We need it to fix up references to system routines, when loading user programs under RUNSYS.
8.3.2. Initialization actions during system generation.

The actions are, roughly, to initialize system tasks and misc. system data (e.g. descriptions of peripheral devices).

**Service functions:**

A set of small IO-routines (INITGETTEXT, INITPUTTEXT), not using the interrupt system, are used for TTY-communication during initialization. One of the connected TTYs is used as a console. Its device number must be set on the operator's panel before starting.

There also exists a primitive space generator to allocate space for system stacks, DEVICE-objects, etc. on top of the existing resident RUNSYS-area (see previous figure). This is done to save MEMLINK-overhead for data objects that are never deallocated.

**Actions:**

On starting, the MASTER CLEAR-button has been pushed, i.e. the interrupt system is off.

1. The available memory size can, at least on the NORD-SMs, be determined automatically by observing address truncation of non-existing memory addresses.

2. Then, device information (device names, device numbers, device types) are read in from the console and necessary data structures generated.

3. Then, we're ready to fork our system tasks:

Until now, no real "processes" exist - only a running stand-alone program, which executes code both in permanent and transient RUNSYS routines.

First, we must create a proper RUNNING-task out of the initialization program. This is done by some ad-hoc method. At the same time the interrupt system is turned on.
The standard way of forking system tasks is as follows:

3A. Take a local copy of the global STACKDESCRIPTOR.
3B. Allocate a new TASK-object, T, and a new data stack. Set STACKDESCRIPTOR to represent the new stack.
3C. Initialize the READYQ with the current task, i.e. RUNNING its
3D. Then T:=RUNNING and the new task can be "entered", in its correct stack environment, by calling its associated routine. This routine will ultimately hang itself up in a semaphore (or detach itself in another way). Then the dispatcher will be called to pick up a new RUNNING-candidate from the READYQ, and we're back where we started.
3E. Unsave the STACKDESCRIPTOR (cf. point 3A) and continue forking.

The first system task to be forked is the dispatcher itself!

The second is REQUESTHANDLER. Software interrupts will now be properly handled.

Then the remaining tasks can be created. These fork operations are really nested, as one of the system tasks (IDLEDEVICEHANDLER) may fork its own IO-drivers, but this causes minor problems.

Note that creation of system tasks is done as automatic as possible. Actual storage locations of system data are not determined before (a new version of) RUNSYS has been loaded. The number of system tasks will depend on the number of devices supplied.

Also note that available permanent RUNSYS routines (e.g. synchronization primitives) are heavily used during initialization, thus reducing the size of transient initialization routines.

4. It remains to write out a disk-copy of the memory area occupied by RUNSYS, for later use by our hardware boot (see sec. 8.3.3 below).

5. Then the basic MEMLINKs can be established.
6. Finally, IDLEDEVICEHANDLER will allow interrupts on all relevant devices (FALSE=:SYSTEMLOADPHASE, see sec. 9.2), open the master file in FILESYS - and we're ready for the first TTY log-in.

A complete RUNSYS-generation takes \( \approx 10 \) minutes.

8.3.3. RUNSYS boot.

The described initialization sequence need only be performed when a new device is added to or deleted from the system, or when a new RUNSYS-"release" is introduced. Normally, a special hardware boot gets the system on the air in no time.

The boot basically consists of a short sequence of instructions to be inserted into specified memory locations. When this sequence is executed, it will read in a real read-routine from mass storage, which then will input the entire RUNSYS. (The read-routine may overwrite itself; it's reentrant.)

Control is now passed to point 4 in the initialization algorithm of RUNSYS; or more precisely, just after a similar write-out on disk. I.e. RUNSYS wakes up in its last initialization phase.

On NORD-SM, the machine's hardware loader is used to put the basic boot sequence (of \( \approx 10 \) words) into appropriate memory locations. A small paper tape, containing this information, has been prepared for the purpose. The entire boot operation takes \( \approx 3 \) seconds.

Similar boot tapes have successfully been used for stand-alone versions of the loader and the compiler.
"Testing shows the presence, not the absence of bugs", E.W. Dijkstra

9. System testing.

When testing an operating system, and particularly its inner kernel, we're faced with some very unpleasant facts which ordinary users normally don't experience:

1. Irreproducibility of error conditions, due to the randomness of external interrupts and of a dynamic work-load.

2. It is vulnerable to (low-level) errors, as we have limited or no background support.

3. Monitoring of system functions may itself change these very functions (a classic dilemma in philosophy of science).

4. Monitoring may not only slightly alter the system we're supervising. The entire system may crash if we are not careful. E.g. if we attempt to write out debug texts from an IO-, synchronization- or space allocation routine, we probably wind up in the same routine once again, because it is recursively invoked during output operations. This tail-biting can be avoided if low-level system routines call special primitive output routines, see sec. 9.2.1.

9.1. Fighting irreproducibility.

In the first testing period, it was decided not to perform IO via interrupts and not to turn on the clock. Thus the only way to cause process-change was via software interrupts. That is, inside ACQUIRE, by REQUEST (IDLE), or by a compiler-emitted WAIT-instruction (causing abortion).

This again implied that task behaviour became deterministic (assuming the number of tasks is program-controlled), and we got reproducible testruns. It was not until nearly all other RUNSYS-features had been properly tested, that external interrupts were
allowed. About 50 new source lines in IO-drivers and initialization
routines were needed to perform the change. Everything
was prepared to handle such interrupts elsewhere.

I dare hardly think of how I could have tested the system - with
a reasonable amount of effort - unless this method had been adopted.
For instance, *25 serious* system crashes caused by compiler errors
were discovered during the testing period (see sec. 12.1.2).

9.2. Debugging tools.

9.2.1. IO-routines.

As mentioned, some *primitive* IO-routines were (and still are) used to
obtain debug information from selected low-level system routines
(to avoid tail-biting). These IO-routines perform no buffering,
they turn off the interrupt system during write-out of a text
parameter, they hang in IOT-loops (on our NORD-SMs) to output a
character, etc.. In fact, they are identical to those used during
RUNSYS initialization.

In order to *parameterize* formatted IO as much as possible, ordinary
IC-routines - like OUTTEXT and ININT - will inspect a global
BCOL-variable named SYSTEMLOADPHASE, to decide whether to call the
primitive IO-routines or the permanent ones:

Ex. Standard OUTTEXT-routine.

```pascal
PROC OUTTEXT=% SAVE RECURS (ROW VAL CHAR T):
BEGIN
---
IF SYSTEMLOADPHASE
THEN
  INITPUTTEXT (T); % Primitive, initial IO-routine.
ELSE
  PUTTEXT (OUTFILE,T); % Permanent IO-routine.
FI;
---
END; % OF OUTTEXT.
```
The permanent IO-routines (like PUTTEXT) will activate "waiting-loop-driven" or "interrupt-driven" IO-drivers, depending on the actual RUNSYS version; see below.

Hierarchy of proc calls for output:

```
OUTTEXT,    permanent SAVE-routine.
SYSTEMLOADPHASE
=TRUE    =FALSE

INITPUTTEXT,    PUTTEXT,    permanent OLAY-routine.
transient SAVE-routine;

PUTIMAGE,    permanent OLAY-routine.

Waiting-loop-driven IO-DRIVER, if test
version of RUNSYS.

Interrupt-driven IO-DRIVER, if final RUNSYS version.
```

Note 1. INITPUTTEXT must not be called after initialization is completed, as its memory area is freed to the MEMLINK pool. Inversely, PUTTEXT being an OLAY-routine must not be called before the MEMLINK pool is established.

Note 2. Waiting-loop-driven IO-DRIVERS in the test version of RUNSYS are not as inefficient as one might fear:

Ex. Waiting-loop section in IO-DRIVERS on NORD-SM.

```
CHAR CH AREG:= <next character>; %% if output.
BEGIN & OK: %% EXIT LABEL for this clause.
   DO
      TO 50 %% Try max. 50 times per activation.
      DO
         IF IOT (<dev.no>) %% Basic IO-instruction, yields TRUE
         THEN
            GO OK;
         FI;
      OD;
      OD;
      REQUEST (IDLE); %% Go to sleep for a while.
```
<AREG contains a new character>; % if input.

Note that IO-DRIVERS of this type must **not** have higher priority than other tasks, including user tasks. Otherwise they will completely monopolize CPU in their active periods. On the other hand, in the absence of a clock, any other task must give up priority regularly, e.g. through REQUEST (IDLE) or ACQUIRE-calls, to grant CPU to IO-DRIVERS.

Additional remarks.

The described uniformism of IO-calls can be extended even further:

If we, in a stripped RUNSYS-package called MINIRUNSYS, also support a global variable named SYSTEMLOADPHASE (=FALSE), we can use almost the same IO-routines as in RUNSYS. Only the DISKIO-, GETIMAGE- and PUTIMAGE-routines from RUNSYS need be replaced by new routines, which perform active waiting on IO. INITPUTTEXT, etc. should be replaced by dummy routines. MINIRUNSYS is particularly used for separate testing of system routines, see sec. 9.3.

Likewise, if we let the global variable SYSTEMLOADPHASE be TRUE, we can easily implement a set of very small SAVE IO-routines for stand-alone programs, if we happen to have space problems. This time PUTTEXT, etc. are replaced by dummy routines.

Note that declarations and calls of IO-routines in system or user programs are unaffected of this. The difference lies in which system routines we order the loader to fetch, and in the value of the BOOLEAN SYSTEMLOADPHASE -variable.
Summary of IO-alternatives:

- **OUTTEXT**: permanent SAVE-routine.
- **SYSTEMLOADPHASE**
  - TRUE
  - FALSE
- **INITPUTTEXT**: transient SAVE-routine. May also be used as a stand-alone SAVE-routine.
- **PUTTEXT**: permanent OLAY-routine.
- **PUTIMAGE₁**: OLAY-routine for MINIRUNSYS.
- **PUTIMAGE₂**: permanent OLAY-routine for RUNSYS
- **interrupt-driven IO-DRIVER**: for final RUNSYS version.
- **Waiting-loop-driven IO-DRIVER**: for test version of RUNSYS.

Admittedly, this elegant parameterization and uniformism in IO-calls was not the first solution to our IO-interface problem. It cost a lot of sweat, but it's definitely worth it. (The intermediate routines, like OUTTEXT, are needed anyway to look up the current task's OUTFILE before calling PUTTEXT.)

Note that all IO-routines are declared as **RECURS** routines, callable by any task.

### 9.2.2. Dump and trace facilities.

We must often rely on tailored dumps to obtain useful debug information from routines.

A DUMP-macro for printing the hex-contents of an arbitrary data object is however available:
Ex. Using DUMP in a system routine.

    REF TASK RUNNING:=--;
    -----  
    DUMP ('RUNNING=', RUNNING:-TASK);  %% Dumping contents of
    %% TASK-object, pointed at
    %% by RUNNING.

Switches on the operator's panel decide whether to activate the
dump, and on which device it should be printed (TTY, line printer).

More important, however, are some ENTER- and EXIT-macros, to be
called on routine entry and exit. They will print the supplied
routine name and a hex-dump of its local data area, including
parameters.

Ex. Using ENTER and EXIT.

    PROC  P=%%--:
    BEGIN
        ENTER('P');
        ----
        EXIT('P');
    END;  %% OF P.

(When the ENTERTRACE- and EXITTRACE-operators are implemented in
MARY, we don't have to insert ENTER- and EXIT-calls manually.
See MARY TEXTBOOK, p. 167.) As for DUMP, switches on the panel
control activation of the macros.

Our "PROCTRACE"-macros have been a great help when testing the
system. They make it easy to trace proc calls and to output
(some of the) data being interchanged between modules. As a
matter of fact, a common way to run a test program (e.g. RUNSYS
itself) has been to activate all possible trace onto the line
printer, and let it run until I could hear the printer stop.
Then, either an error (e.g. a normal MARY run-time error) had
occurred - and I knew where and in which context by inspecting
the printout; or the program had terminated normally - in which case I had to check the results.

9.3. MINIRUNSYS.

This is a set of routines for IO, debugging, space allocation and OLAY-code management; thus simulating the later RUNSYS environment. ACQUIRE, etc. are replaced by dummy routines. MINIRUNSYS occupies ≈3K SAVE code.

Set-up:

```
  MINI-RUNSYS                PROGRAM
       Initial               TO BE
       linkage               TESTED.

      Full
      RUNSYS

  Final
  linkage
```

Calls on system routines (IO, debugging, space allocation) behave as we are running under full RUNSYS, except that everything happens synchronously - there is no multiprogramming.

Such separate testing assumes that monitor/system routines are called via normal RECUCRS proc.calls, and not through special hardware monitor calls. For instance, the entire FILESYS is plug-compatible with MINIRUNSYS; without requiring recompilation.

The normal stand-alone loader is used to link MINIRUNSYS onto our actual program.

The importance of this arrangement for convenient and separate testing of system routines (FILESYS, EDITOR, JOBINTERPRETER, etc.) can hardly be overestimated.
9.4. Testing of RUNSYS.

Given MINIRUNSYS, reproducibility in the absence of external
interrupts and the described debug mechanisms, RUNSYS testing and
debugging caused no major difficulties. Serious system crashes
were rare and concentrated in the initial testing phase. In this
period, the NUALGOL-MARY compiler was far from bug-free, and vital
primitives for space allocation and task management were not
thoroughly tested.

A summary of errors found, and their causes, is given in sec. 9.5.

No really original test principles will be reported in the
following. Interested readers are advised to consult [Dijkstra,

The test phases of RUNSYS, in chronological order:

9.4.1. Compilation of RUNSYS.

We may regard the compilation as the very first test of a
RUNSYS-routine. The compiler will flag all static mode errors
and prepare run-time tests for the rest.

9.4.2. Testing of RUNSYS initialization.

Initially, SYSTEMLOADPHASE was TRUE, so the initial IO-routines
were invoked.
A. First we verified that we actually were inside the main initiali-
   zation routine, by printing something like 'HERE I AM'.

B. Then the basic interrupt-handling (save/unsave operations) was
tested. This happened when making a proper task out of the
initialization program.

C. After this, READYQ-management and ACQUIRE/RELEASE-mechanisms
were tested, during creation of system tasks.
(See also sec. 8.3.2 on initialization actions.)
9.4.5. Testing of general IO-drivers and associated interface.

Permanent IO-routines (like PUTTEXT) had also been tested under MINIRUNSYS for a long time. When linking these routines onto RUNSYS, only the bottom-level PUTIMAGE- and GETIMAGE-routines had to be substituted by their RUNSYS-equivalents, conforming to the IOREQUEST-formalism for exchange of IO-messages.

The following testing was now rather delicate:

We were about to put FALSE into SYSTEMLOADPHASE, thus ordering OUTTEXT to activate PUTTEXT instead of INITPUTTEXT. As explained earlier, we must then remove (or passivate) all debug dumps from routines like ACQUIRE, RELEASE, GETSPACE, TASKCHANGER, REQUESTHANDLER etc. to avoid unpleasant recursions. Neither could debug data be output to our traditional TTY, being the actual test device.

Hence, we had to cheat a little. The tricks were:

1) to "turn on/off" SYSTEMLOADPHASE in selected low-level routines,
2) to direct debug data to the line printer.

This worked out perfectly, and the testing proceeded as follows:

A. TTY-output was tested.
B. TTY-input was tested.
C1. TTY log-in was tested.
C2. Forking of a corresponding JOBINTERPRETER was tested.
       From this point, test programs didn't have to be a tail of the initialization routines, as we might fork a general test program after TTY log-in, instead of a JOBINTERPRETER.
D. Testing of the remaining devices (paper tape reader and -punch, card reader, line printer) was no sweat.

Let us emphasize, once again, that external interrupts were not yet permitted. However, asynchronous buffered IO became available. A touch of irreproducibility was thereby introduced, but we didn't notic
9.4.3. Testing of simple user- and system tasks.

A. In order to test space allocation and user task-creation, -synchronization and -termination, a set of test routines were initially appended to the main initialization routine. This was done because no OLAY-facilities, IO-drivers or a RUNSYS loader were (yet) available. The only way to implement "user programs" was to let the initialization routine call test routines in this fashion.

Note that we must protect initialization- and test-routines from being overwritten, by not including their memory areas in the free MEMLINK pool, cf. figure in sec. 8.3.1.

After this point, general task administration ought to be in good shape.

B. Then the disk driver was tested, by executing some simple DISKIO-calls which generated and released IOREQUEST-packages to it.

This driver was the very first IO-driver to be tested, and we could use our primitive IO-routines throughout the test. (Note, the disk driver was not interrupt-driven yet.)

9.4.4. Testing of OLAY-code.

Primitives for OLAY-code management (the MAKEPRESENT-routines) had been thoroughly tested in separate test programs, using the MINIRUNSYS package.

Thus we simply modified the above RUNSYS test routines towards OLAY-code, and called them as under point A in sec. 9.4.3. Again, it was easy to test and debug the entire calling mechanism for OLAY routines, being shared between several tasks.
9.4.6. Testing of JOBINTERPRETER functions.

JC-interpreter functions were well tested under MINIRUNSYS.

Initially no FILESYS-calls were allowed. We concentrated on the
general JC-functions: forking, execution and termination of
arbitrary sub-tasks.

9.4.7. Testing of FILESYS functions.

FILESYS had also been tested under MINIRUNSYS. It remained to
check file-synchronization mechanisms and all-over behaviour in
a multiprogrammed environment. Some buffer-allocation errors for
ATOMIC files were found.

9.4.8. Testing of editor-, loader- and misc. functions.

Both the editor and loader had undergone serious stand-alone tests,
and the testing became easy. The OLAY-data facility was now run in
its correct environment. The booted MARY compiler might also
have been tested, but was not available at that time.

A digression: Most system routines (still) have ENTER- and
EXIT-macros inserted in their code. Thus, if we
want to activate a general procedure trace (by
turning on panel switches), the generated output
dumps will automatically be directed to the current
task's OUTFILE (e.g. TTY) on system calls.

These built-in trace functions have been very help-
ful during debugging.

9.4.9. Testing of multiple TTYs.

At last, more than one TTY was connected to the system. The line
printer and other peripheral devices became no longer directly
accessible. Special printfiles and similar mechanisms must be used
At this point, some system errors admittedly were discovered, particularly deadlocks. But the major work dealt with tuning the system towards a better performance. For instance, the virtual memory system underwent substantial modifications.

9.4.10. Testing of clock functions.

Now we were ready to implement the real time clock, and went back to point A in sec. 9.4.3 to obtain the most primitive task environment we could get.

The real danger in this area was programming errors in disable-enable pairs (missing or nested pairs) in low-level synchronization operations. Such synchronization errors were likely to pop up as second-order run-time errors later. - Really nasty!

Note that SYSTEMLOADPHASE initially was TRUE, so we were able to produce nice debug dumps. In this case, intervals between registered clock interrupts might get rather long, as our initial I0-routines disable the interrupt system during basic I0-operations.

The best cure against bugs is prevention, and I read very, very carefully through all low-level system routines. Maybe because I did so, the result was beyond my wildest expectations:

No such synchronization errors were found.

9.4.11. Testing of interrupt-driven I0.

Technically, it was straight-forward to adapt I0-drivers towards external interrupts, instead of a waiting-loop system.

However, missing-connect interrupts on the SM4 are still not satisfactorily treated.

The testing itself caused minor problems. We had apparently verified all relevant disable-enable pairs by testing the clock.

An operating system is never finished.

However, we don't want to compete with EXEC-8 on the U1108 in generality. A lot of user requests have simply been ignored.

Normal error fixing and maintenance have, of course, been carried out according to the author's capabilities and spare time. See ch. for a status report.

9.5. General comments on the testing.

Data on committed design- and programming errors have been assembled during the first and intermediate phase of the development period. These phases cover 4 months out of a 19-month development period.

An error statistics for these months is presented and technically analyzed in sec. 12.1.2. In this section we shall give some general comments on the experience gained during system tests.

9.5.1. Time spent on testruns.

About 15% of my working time has been spent on actually running the system, or parts of it. Although the system is not completely tested, this 15%-figure seems to be much lower than similar estimates from assembler-coded systems. A paper by [Boehm, 1973] indicates that about 50% of the time should be spent on testruns.

9.5.2. On run-time errors.

The table in sec. 12.1.2 shows that about 12% of the errors were discovered/caught at run-time.

The corresponding logical or semantic errors were often rather trivial, e.g. simple interface or initialization errors. This observation coincides with experience from implementing large systems in general. [Wulf, 1974b]: The difficult problem is not the coding of individual modules, but to interface hundreds of modules into a cooperating unit.
Very often, we had "forgotten" to treat a rare case properly, or we didn't update consistently all the calling contexts of a routine when the functions of this routine were altered. Similarly, when testing new features, the most serious bugs were frequently found in the test programs, that hadn't quite managed to "simulate" the environments prescribed.

"Interface" errors of this type are very high-level semantic errors, and we will need a much more powerful programming tool than any existing programming language to cope with them.

However, even if an error itself is "trivial" (it is not a serious design error), it may not be harmless, and it may not be trivial to find.

9.5.3. Experiences with operating system testing.

As mentioned previously, testing of operating systems can be theoretically and practically difficult.

In the very first testing phase we actually had to prick in instructions and use raw dumps of large memory areas to find out what really happened. But similar old-fashioned assembly methods, like snap-shots and break-points, were rarely needed later.

When testing a new module/facility, we routinely prepared a rather exhaustive test program and let it run; first under MINIRUNSYS, later under full RUNSYS. Standard MARY run-time traps flagged most of the errors, and debug macros supplied us with relevant data on error terminations.

If the system really crashed, we immediately read in the stand-alone loader (into the top-most memory locations), to use its built-in dump-commands to inspect selected system data (TASK-objects, data stacks).
Errors causing **system crash** were primarily:

* Second-order errors (dangling pointers, erroneous synchronization)
* Inadequate treatment of abnormal task terminations,
* Resource allocation errors, usually memory shortage.

**Abnormal task terminations** were (and are) a hard nut to crack. E.g. the risk of introducing deadlocks, when a killed task has acquired some shared resources, lead to the inclusion of RECEIVER-attributes in RESOURCE-objects.

System design, coding and testing is a dialectic and iterative process. Hopefully it converges towards a better system. See ch. 10 on security and ch. 15 on a general evaluation.
10. RUNSYS security. User and system protection.

Everybody wants a safe system, of course.

* The system should tackle hardware errors, in a reasonable way.

* It should behave benevolently in normal and abnormal software situations. Emphasis should be put on preventing abnormal situations (deadlocks, memory shortage) from occurring.

* It should distribute safe access of shared resources (memory space, mass storage space, device rights) between multiple users.

For instance, a user task should not be able to "crash" other tasks (system or user ones). Actually, it should hardly be allowed to crash itself in a destructive manner, because of the side effects.

10.1. Treatment of hardware errors.

* Power failure. A special POWERFAILRE task receives control if the power fails. It will save and mark some selected system data, before halting. On the SM4, for instance, we have only ≈200 μs to prepare necessary restart actions, so we must not disable interrupts for excessive time periods! Current IO-operations (reading a card, printing a line, transferring a disk sector) must be re-initiated on system restart.

This facility is not yet implemented.

* Disk errors.

The file system, in cooperation with the disk driver, is expected to take necessary actions. A status word, supplied by the user, will routinely be set on software or hardware file errors. Damaged disk areas can be "hidden" by manipulating the disk's track map; if we have linked files.
If a file is destroyed, a backup version must be supplied.

Our main worry is the lack of appropriate backup media to replace paper tape. The presently available SM4-machine has no tape drives. Even a 9.6 K-baud line between SM4 and U1108 will need ≈1 hour to load a 1M-word disk-pack from SM4 to standard U1108 files.

For the time being, each group of users has their private disk-pack, with a copy of the system in front. This desentralized system will at least reduce the consequences of hardware file errors.

10.2. Security features offered by the MARY language.

The majority of system programs and all user programs are written in SAFE MARY, offering full security against destructive programming errors (see below). In addition, a mode checking link-editor will be needed to ensure consistency between separately compiled modules.

MARY is a fully typed language. The compiler can perform most of the mode checking at compile-time. Some tests must be postponed in form of run-time traps (software interrupts) to check:

* empty pointer errors,
* set range errors,
* subscript range errors (really a variant of set range error
* stack overflow errors,
and a few more exotic errors.

These tests are in fact very cheap (≈5% of the code) and virtually indispensable to prevent second-order errors.

Currently there remains one non-implemented run-time trap, namely scope-checks on "pointer"-assignments (REFs, ROWs, PROCs and LABELs). See MARY TEXTBOOK, p. 77.

This has, however, not caused any unflagged run-time error in ≈60.000 lines of MARY code (disregarding the special situation for pointers into stacks of abnormally killed child tasks, see sec.3.4.2
Thus, assuming a mode checking link-editor and scope checks on pointer-assignments, normal MARY programs cannot behave destructively. SAFE MARY forbids dirty type conversions, SYNing of data, omission of required data initializations, and execution of IO- or EXECUTE-instructions.

Observe that this program security is not waterproof, unless the compiler is (reasonably) bug-free(!).

Some technical questions arise in this area: How to prevent smart programmers from using the UNSAFE version of the MARY compiler? How to prevent normal programs from calling reserved system programs? (System routines on U1108 have an exclusive '$' in their names to forbid this.) How should mode checking between already loaded system routines and new user programs be organized? I cannot suggest any foolproof system.

System routines will usually have run-time traps inserted in their code as an extra security precaution, even if they are inherently UNSAFE.

UNSAFE MARY is primarily used when programming basic IO, interrupt-handling and space allocation. Sec. 12.2.3.4 contains a statistics on the usage of UNSAFE constructs.

10.3. Software errors, treatment and prevention.

10.3.1. Errors in user programs.

The built-in security of MARY programs, described in the previous section, grossly reduces the amount and consequences of local errors in user programs. In the worst case, a user task has to be terminated.

Note that run-time errors, caught by compiler-emitted run-time traps, may "pop up" in quite different routines than those where the underlying logical errors were committed.
10.3.2. Errors in system programs.

In the following, we shall investigate the consequences of software errors in system routines and -tasks.

As mentioned in sec. 10.2, system routines normally contain run-time traps for standard MARY errors. However, it can be very dangerous for the system to let system routines fall into such traps, as these routines can be executed by system tasks.

The best cure against errors is always prevention: Robust design and careful programming. Furthermore, we must try to discover inconsistent data structures before they manifest themselves as abortive run-time errors. The causes of inconsistencies are often meaningless parameters to system routines. Thus we should always check that a file is not deleted/closed when a user wants to access it; and so on. MARY RUNSYS offers a lot of advanced features (for forking and synchronization) to ordinary users, and we must check extra carefully that these facilities are correctly used.

Such error checks can be rather labourious, but they are necessary not only to prevent system crashes, but also to provide users with relevant diagnostics in error situations.

10.3.3. "External" (deadlock-like) error conditions.

These are typically max. time error, max. memory error, output file overflow error, or error termination because some parent task gets killed (e.g. because of critical block error). Stack overflow error, being a normal MARY run-time error, should also be included in this category.

Such abnormal deadlock- terminations may create difficulties, if we are in the middle of an updating operation on a system resource, e.g. a file dictionary. That is, we ought to give the task - if possible - some extra CPU-quantums or memory words, so that the updating can be carried out consistently.
An extra TASK-attribute of mode INT, called NOOFUPDATES, has therefore been introduced to delay abnormal task terminations, if we're inside a "critical updating" region (or a nested one).

However, in case of stack overflow there is little we can do to delay fatal task terminations. Routines like ALLOCFILE and DELETEFILE must therefore explicitly test whether the remaining stack space is sufficient to room a full file-update operation.

Maybe these problems derive from our "stack-implementation" of tasks; letting arbitrary tasks execute system routines to update system data? But to leave all critical updating e.g. to special system tasks may be rather bothersome; and it doesn't really eliminate, say, max. time abortions. See sec. 15.1.2.

Note 1: The function of NOOFUPDATES resembles that of a semaphore - in the sense that it regulates entry and exit of critical "regions". But it always works on a single task; no "synchronization" in the conventional meaning is involved/required. In fact, the actual (system) resources are usually locked already.

Note 2: The ED-processor on U1108 may serve as a good example on recovery/prevention from common error situations, such as max. time- and max. pages abrtions.
11. Current status of RUNSYS.

First, some non-implemented features and general system data are summarized. Then, 2 real-world test runs, illustrating the present power of RUNSYS, are presented.

11.1 Non-implemented features.

The following list only includes the major ones. On the other hand, there are features being implemented, or implemented more generally, which have not been properly documented.

11.1.1 Described, but not yet implemented features.

* The SEMA- and RESOURCE-formalism for synchronization is not fully implemented. SEMAphores are presently called EVENTS, and some small, separately allocated list objects are used to maintain resource-queues; cf. [Holager, 1973].

Hence automatic task-termination-release mechanisms are not available, and explicit updating actions on system resources (IO-buffers, device controls) must be carried out on task terminations.

* The treatment of powerfailure and misssing-connect inter-
rupts (SM4) is very rudimentary.

* The editor presently requires 6 memory-resident words to represent a text line. The mass storage format is also slightly different than previously outlined.

* A mode checking link-editor is not ready.

* A hierarchical, deadlock-free file locking algorithm, the LOCKFILES routine, is not implemented. Possible file deadlocks are resolved by task abortions (long-wait abortion, or abortion because a requested atomic file already is being used for some other purpose.)
* The surveillance functions performed by the IDLEDEVICE-HANDLER-task, e.g. to prevent trashing, are not fully implemented.

* Error detection and -recovery in system routines, that perform updating of system resources (particularly in IO- and FILESYS-routines), are not first class.

11.1.2 Planned, but not yet implemented features.

* Miscellaneous FILESYS utilities: A possibility to erase the subfile space of a COMPOUND file without deletion of its file entry; Adequate backup facilities; Reuse of vacant subfile-entries to avoid packing of COMPOUND files; Automatic mechanisms to generate private, temporary scratch files (needed by the loader, editor and compiler).

* A system-generator for read- and printfiles, thus facilitating a spooling system.

* High-level (file-) mechanisms for handling external devices; like card reader, paper tape reader and -punch.

* Expansion of console communication for instrumentation and monitoring purposes.

* Better error messages (and edited "dumps" of system data) on task terminations. Current system diagnostics are much too cryptic.
11.2 Some quantitative measures.

11.2.1. Size of system modules.

(m) means "coded by me", if common modules.

All figures are in thousands.

<table>
<thead>
<tr>
<th>Module</th>
<th>Source lines</th>
<th>SAVE-code &amp; data</th>
<th>OLAY-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common RUNSYS preludes:</td>
<td>1.45</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Interrupt-handling:</td>
<td>0.3</td>
<td>0.28</td>
<td>-</td>
</tr>
<tr>
<td>System drivers (I0, clock):</td>
<td>0.9</td>
<td>1.8</td>
<td>-</td>
</tr>
<tr>
<td>Task administration and</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>synchronization:</td>
<td>1.0</td>
<td>0.60</td>
<td>1.5</td>
</tr>
<tr>
<td>Space administration:</td>
<td>1.05</td>
<td>1.75</td>
<td>0.2</td>
</tr>
<tr>
<td>Proc-call adm. routines:</td>
<td>0.8</td>
<td>0.4</td>
<td>-</td>
</tr>
<tr>
<td>OLAY-code management:</td>
<td>0.7</td>
<td>0.75</td>
<td>-</td>
</tr>
<tr>
<td>OLAY-data management:</td>
<td>0.55</td>
<td>0.10</td>
<td>1.05</td>
</tr>
<tr>
<td>IO-interface routines:</td>
<td>0.65</td>
<td>0.25</td>
<td>1.6</td>
</tr>
<tr>
<td>Formatted-IO routines:</td>
<td>0.65</td>
<td>0.75</td>
<td>1.7</td>
</tr>
<tr>
<td>High-level task management,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>incl. job control interpreter:</td>
<td>0.9</td>
<td>0.2</td>
<td>3.8</td>
</tr>
<tr>
<td>FILESYS routines:</td>
<td>2.9</td>
<td>0.18</td>
<td>8.7</td>
</tr>
<tr>
<td>LOADER routines:</td>
<td>0.65(m)+1.6</td>
<td>0.15</td>
<td>4.4</td>
</tr>
<tr>
<td>EDITOR routines:</td>
<td>2.45</td>
<td>0.1</td>
<td>8.0</td>
</tr>
<tr>
<td>COMPILER routines:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.9(m)+23</td>
<td>+2.5</td>
<td>+118</td>
</tr>
<tr>
<td></td>
<td>15.9(m)+24.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initialization routines:</td>
<td>0.8</td>
<td>3.0</td>
<td>-</td>
</tr>
<tr>
<td>System boot:</td>
<td>0.1</td>
<td>0.2</td>
<td>-</td>
</tr>
<tr>
<td>Misc. MINIRUNSYS routines:</td>
<td>(2.0)</td>
<td>(3.5)</td>
<td>(2.0)</td>
</tr>
</tbody>
</table>

Total sizes:

\[
\begin{align*}
\text{Total sizes:} & \quad 18.7(m)+24.6 & 10.5 & 31.0 \\
& & +2.5 & +118
\end{align*}
\]

(Approx. 7 K of MARY code for test- and utility-purposes have also been written, but they seem immaterial here).
About 10% of the source code are preludes (common declarations). Another 10% is needed to reference these.

A printout of the entire RUNSYS is supplied as a separate Appendix D.

11.2.2 Selected system characteristics.

Interrupt response time: 0.4 ms.
A full change of the RUNNING task costs: 1-1.5 ms.
Size of system data per extra TTY: ≈ 300 words.
Extra SAVE-data per (inter)active user: ≈ 2-15K.
Overhead of SAVE proc calls: ≈ 90 μs.
Overhead of OLAY proc calls: ≈ 550 μs.

11.3 Examples of test runs.

Two complete job streams can be found in Appendix C.

1. The first one demonstrates some common filesys, editor and JC-operations.

2. The second one will load and execute a precompiled test program, which performs some user-controlled forking and synchronization operations, à la the producer-consumer pair from sec. 3.3.3.
12. An evaluation of the tools used.

While chapter 15 contains an evaluation of the entire system, this chapter discusses the tools that were used to construct the system. Particularly the MARY language will be commented upon, as RUNSYS originally was developed to support segmented execution of our MARY-MARY compiler (or generally, any large MARY program), and since RUNSYS is one of the first bigger programs coded entirely in MARY.

In the following, we shall focus attention on three major sources of errors/inadequacies:

* Ull08 standard software.
* The NUALGOL-MARY compiler and the MARY stand-alone loader.
* The MARY language in general, and its SIL-qualities in particular.

12.1 General impression of the available tools.

12.1.1 Programming productivity.

Some introductory remarks:

The available tools must have been rather good, as I've produced \(\approx 18,000\) lines of debugged and reasonably well documented MARY code in about 19 man-months, design and tuning included. And it's not trivial programs, neither in size nor in complexity. 4 months of thesis-writing comes on top of this.

The yielded amount of code is about 45K, i.e. about 80 instructions per man-day (7 days per week).

These figures coincide well with today's standard production rates for writing structured programs in high-level languages.

The productivity is about 4 - 7 times higher than normal assembler standard, and we expect an even greater gain in the maintenance phase.
Reasons: Assuming the number of coded lines per day is constant, we benefit automatically by the normal expansion factor of 2-5 between source lines and generated instructions. Furthermore, we commit fewer coding errors; and we don't have to split design and programming artificially, by using two different notation languages (flowcharts and assembler). Lastly, because of increased individual productivity, we can manage with much smaller teams, thus reducing managerial overhead.

As mentioned before, program development has mainly been done on our Ull08. After successful compilations, paper tapes were prepared, and the compiled programs loaded on the minis via the MARY stand-alone loader. Our pilot compiler, written in NUALGOL/ASSEMBLER, was used to compile MARY programs on Ull08.

Doing program development for small, "naked" machines from larger, better equipped machines is clearly economically advantageous. However, if adequate software had been available on the minis (in fact the driving motive behind MARY and MARY RUNSYS), a considerable economic profit is expected: Hardware prices differ by a factor ≈10 - 40. If the over-all performance rates differ by a factor ≈5-10 the other way, the price/performance gain will be ≈4 - 5.

Our (the MARY group's) general impression is that good interactive support systems (a context-sensitive editor, available(!) fast CRT-terminals) increase our programming productivity by at least 50%, compared to a classic key-punch-oriented batch environment. Although existing "backup" software on the minis (e.g. the loader) deserves improvement, this didn't hamper program development significantly, as only 15-20% of our total time was spent on test-runs.

12.1.2. Error-proneness.

The error statistics, announced in sec. 9.4, is first given as a background material.
12.1.2.1. Error statistics.

Total number of errors: ≈600, in about 4000 source lines.

<table>
<thead>
<tr>
<th>Error group</th>
<th>Error frequencies</th>
<th>Sums</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. U1108-exec errors:</td>
<td></td>
<td>11%</td>
</tr>
<tr>
<td>1.1. File errors:</td>
<td>2%</td>
<td></td>
</tr>
<tr>
<td>1.2. Compiler control card errors:</td>
<td>4%</td>
<td></td>
</tr>
<tr>
<td>1.3. Editing errors:</td>
<td>4%</td>
<td></td>
</tr>
<tr>
<td>1.4. Misc.:</td>
<td>1%</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>11%</td>
</tr>
<tr>
<td>2. Compilation errors:</td>
<td></td>
<td>60%</td>
</tr>
<tr>
<td>2.1. Clerical or trivial coding errors:</td>
<td>8.5%</td>
<td></td>
</tr>
<tr>
<td>2.2. Mode conflict errors:</td>
<td>8.5%</td>
<td></td>
</tr>
<tr>
<td>2.3. Missing decl. errors:</td>
<td>11%</td>
<td></td>
</tr>
<tr>
<td>2.4. Inconsistent prelude errors:</td>
<td>14%</td>
<td></td>
</tr>
<tr>
<td>2.5. Compiler inadequacies (Fixup trouble, register lockings):</td>
<td>15%</td>
<td></td>
</tr>
<tr>
<td>2.6. Compiler errors (rejection of legal constructs):</td>
<td>2%</td>
<td></td>
</tr>
<tr>
<td>2.7. Misc.:</td>
<td>1%</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>60%</td>
</tr>
<tr>
<td>3. Linking and loading errors:</td>
<td></td>
<td>11%</td>
</tr>
<tr>
<td>3.1. Trivial loading errors:</td>
<td>9%</td>
<td></td>
</tr>
<tr>
<td>3.2. Inconsistent modes of separately compiled data or routines:</td>
<td>1%</td>
<td></td>
</tr>
<tr>
<td>3.3. Misc.:</td>
<td>1%</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>11%</td>
</tr>
<tr>
<td>4. Run-time errors:</td>
<td></td>
<td>12%</td>
</tr>
<tr>
<td>4.1. Detected by standard run-time traps (e.g. for empty pointers):</td>
<td>6%</td>
<td></td>
</tr>
<tr>
<td>4.2. Second-order errors (like dangling pointers) caused by UNSAFE MARY constructs:</td>
<td>1%</td>
<td></td>
</tr>
<tr>
<td>4.3. Compiler errors:</td>
<td>2%</td>
<td></td>
</tr>
<tr>
<td>4.4. Synchronization errors, mostly deadlocks:</td>
<td>0.5%</td>
<td></td>
</tr>
<tr>
<td>4.5. Other errors, not causing crash:</td>
<td>2.5%</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>12%</td>
</tr>
<tr>
<td>5. Logical errors found through code re-reading, after a program has been taken into use:</td>
<td></td>
<td>6%</td>
</tr>
<tr>
<td>5.1. By correcting other errors:</td>
<td>1%</td>
<td></td>
</tr>
<tr>
<td>5.2. By inspection of the program itself:</td>
<td>5%</td>
<td></td>
</tr>
<tr>
<td>5.3. By studying relevant background material:</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6%</td>
</tr>
<tr>
<td>Sum:</td>
<td></td>
<td>100%</td>
</tr>
</tbody>
</table>


12.1.2.2. Comments on the errors committed.

A. Many errors are, of course, caught by normal proofreading of programs and don't appear in the list. See also error group 5.

B. Group 1, U1108-exec errors, are mainly file locking errors (trying to access unopened files) or trivial editing errors (interchanging edit and input "mode", incorrect splitting of lines). Many of these errors could have been avoided by proper redesign of U1108's file system and editor.

C. Group 2, compilation errors, are often rather trivial ones; there are few grave mode conflict errors (group 2.2). This indicates that once the notation is reasonably good, we are not likely to do very many semantic errors.

Prelude errors (inconsistent VADD-cards, group 2.4) can be greatly reduced by a minor modification in the MARY syntax.

The bothersome fixup and register locking errors (group 2.5) are mainly caused by some silly one-pass restrictions of the MARY compiler.

D. Generally, error reporting and -recovery is not first class on our pilot compiler. It has not yet reached a fairly bug-free state, cf. error groups 2.6 and 4.3! As a matter of fact, implementation of an operating system is not very pleasant on an experimental compiler. Garwick's statement that "a language is defined by its compiler", do contain a kernel of truth. [Garwick, 1964].

Yet, we managed to survive without ulcers.

E. Group 3, linking and loading errors, were relatively innocent; but it's rather annoying to restart a RUNSYS loading because of typing wrong TTY-characters to the loader. Some kind of parameterization of loader input commands, via paper tape or cards, ought to be implemented.
It is surprising that we experienced a very small percentage (1% in group 3.2) of mode interface errors. Other reports [Klunder, 1973] indicate that ≈25% of all run-time errors (i.e. 4% in our case) fall into this category. In MARY, modes of data blocks and external procedures must be written out in full, e.g. in preludes. Thus the compiler will at least flag the majority of mode-mismatches.

Nevertheless, a future, mode checking link-editor is essential to ensure over-all security when executing user programs.

F. **Group 4**, run-time errors, account for only ≈12% of the total number of errors.

Disregarding compiler errors (2% in group 4.3), only 1% of the errors caused **serious system crashes** (group 4.2). The remaining errors were easy to locate and correct.

Observe that half of the errors found at run-time (6% in group 4.1) were caught by compiler-generated **run-time traps**. If these traps hadn't been inserted, the number of second-order errors would probably have increased drastically - from 1% to 7%!

Really, these results pay a great tribute to MARY. My personal experience is, that once a MARY program compiles, the major job is done. The previous figures indicate ≈1 run-time error per 60 source lines.

### 12.2. Evaluation of the MARY language.

Its **practical usefulness**, as a "general" programming language, has already been demonstrated.

We shall now explore some more specific language qualities, namely:

* Machine-orientation.
* Run-time efficiency.
* Expressive power, particularly for describing large systems
* Orientation towards "structured programming".
* Multiprogramming qualities.
Some new language constructs will also be suggested. They may not be easily implementable in any existing language.

Portability matters will be discussed in ch. 14.


This has been satisfactory: MARY gives access to all relevant hardware registers. The compiler can be made to emit almost any sequence of machine instructions, e.g. by using the locator directive (':=') to specify working registers. Naturally, low-level register fiddling requires more writing effort by the programmer.

Observe that most instructions are accessible through normal MARY operators, like shift-operators from IMPLEMENTATION PRELUDE.

MARY has a FREE-mechanism to allow re-use of declared entities (e.g. registers), without exiting the current scope. This facility is heavily used in machine-near programming.

A few exotic registers (interrupt-priority registers) and instructions (basic IO, interrupt system control) are only available via explicit EXECUTE-calls.

Pure machine-orientation is rare; it's often linked with efficiency, see below.

12.2.2. Run-time efficiency.

Time and space efficiency will sometimes be contradictory qualities. The latter one tends to be preferred on mini-computers.
12.2.2.1. Data space efficiency.

This was achieved through standard PACKing, and is judged to be satisfactory.

The only relevant data space optimization, not currently implemented, is to pack type-fields of template-objects together with the template data. (See Appendix B.3.2). This would, for instance, have given current OLAY-code descriptors a more logical and safe notation.

12.2.2.2. Code efficiency.

First a few general remarks.

Usually, a short routine executes faster than its voluminous counterpart. Yet, the question of open versus closed subroutines (macros versus procedures) is interesting. In hard time-critical regions we cannot afford to call routines, so in-line code must be used. But normally, and particularly when we want to save memory space, few instructions and heavy subroutinization will be valid criteria for code efficiency. Only the first one is genuinely language-dependent.

Note that if an implementation language yields, say, twice as much code as assembler, the entire program swells up unt tolerably, even if the time-critical 8%-portion of the code is located and improved to assembler standard.

Clearly, the efficiency of a program is very much determined by its general design, but this doesn't mean that local code optimization is irrelevant and not appreciated.

Code efficiency of RUNSYS routines has been obtained in several ways:

* by appropriate design of data and algorithms.

The subject falls partly outside the scope of this discussion. However, some design choices are clearly language-dependent. E.g. MARY's general pointer concept penetrates the entire system. Its impact on efficiency is described below.
by utilizing MARY constructs, that encouraged compact code.

I would like to mention:

- Indirect addressing through pointers provides faster data access than conventional array indexing.
- Constant tables can be generated at compile-time.
- Expression language, equal operator priorities, special MARY operators (like '=:', ':=', ':=+', ':=-') and so-called "operator expressions" allow the programmer to optimize the code manually in a straightforward manner.
- MARY's FOR-ROW construct gives very efficient array scanning.
- Explicit use of registers may eliminate a lot of load-store instructions.
- Non-RECURS and "register" routines offer fast proc-call sequences.
- Optimizing ("McCarthy") AND- and OR-operators improve BOOLEAN tests.

by letting the compiler optimize the code.

Regrettably, only peep-hole optimization is implemented on the current two MARY compilers.

Some figures:

Several test samples - [Madsen, 1972], [Rekdal, 1974] and private examples - indicate that MARY generates between 0 and 30% more code than normal good assembly code. The extra overhead derives from accessing packed data objects, superfluous load-store operations, restrictions on the use of integers as pointers, overhead in proc calls, inserted run-time traps, reentrancy considerations, etc..

On the other hand, the small time-critical MARY-routines for handling interrupts and for entering and leaving RECURS routines have the same number of instructions as their assembler variants on NORD-SM.
Visual inspection of normal MARY-generated code on the NORD-SMs indicates that a global optimization scheme [Wulf et al., 1975] would have eliminated about 20% of it — mainly by getting rid of load-store pairs. I will personally judge existing peep-hole optimization to be credited for a 10% reduction of the code.

On a multi-register machine, like the U1108, manual allocation of data in registers will be necessary to compete with assembler, unless we can rely on global optimization.

"Hand optimization" (declaring extra temporaries and putting data in registers) seldom gave more than a 10% code reduction in RECURS routines on the NORD-SMs. Reasons: Expressions tend to be very simple (cf. [Knuth, 1971]), and our minis have very few register none of them survive normal proc calls.

Actually, I discovered that the greatest savings in code space were on the module level, in routines that weren't segmented properly. E.g. an editor module shrunk to 1/3 of its original size by introducing appropriate subroutines for common sub-functions

Thus, emphasis has been put on a clear straight-forward style, not on tricky code.

Note that reentrant (i.e. RECURSive) and OLAYable MARY routines may significantly reduce the required amount of resident code.

The machine code generated is, as mentioned, not always a relevant measure of code efficiency. For instance, the hidden overhead inside system routines on entry and exit of SAVE RECURS-routines takes about 30 instructions, which is about 3 times as much as required by a stripped stack organization.
12.2.3. Expressive power.

See also sec. 12.2.5 on language improvements.

12.2.3.1. MARY's data types.

MARY is gorgeously equipped with classic data types: sets, subsets, arrays, structures, general pointers, ROWs, procedures, labels and templates. VALing and PACKing are additional features.

It is usually straight-forward to describe a given data structure in MARY. However, to modify a data structure radically, e.g. to alter the representation of a table from a linked list to an array, is not easy. On the other hand, to alter the internal representation of structured data - e.g. to PACK or not to PACK - is no sweat. (It's not in assembler, unless we have a powerful macro facility.)

MARY permits user-defined operators. Both operators and routines can work on classes of modes. I.e. we get a powerful tool for describing general algorithms.

All data accesses are completely mode-checked by the compiler, giving a high degree of security, cf. error statistics.

12.2.3.2. Macros in MARY.

Macros are virtually indispensible in an implementation language, as they make it much easier to write well-structured parameterized programs. In RUNSYS, macro texts are routinely used to hide away machine-oriented or tricky code sequences; study the RSDECS.prelude in Appendix D. Macro declarations constitute ≈40-45% of RUNSYS preludes.

The mode-macro interface in MARY deserves special attention. E.g. the SGEN-macro for space- allocation will apply the OPERANDSIZE-builtin on its mode parameter M, to evaluate the data space demand of an M-object before calling a GETSPACE- routine.
Certainly, macros have their drawbacks, if used in a cryptic manner, cf. [Conradi, 74], pp. 12-13.

12.2.3.3. Areas in MARY.

The AREA formalism for data is very elegant, and provides a high degree of security and portability when using registers.

12.2.3.4. UNSAFE MARY.

A good measure of the inherent power of a language is to count how many times we had to step "outside" the language in order to do efficiently what we wanted. Such "cheating" may occur on two levels: Either the language is not low-level enough, or it is not general enough to describe conceptually what we're doing (e.g. it may lack union-like data types).

The relative usage of UNSAFE-constructs (see Appendix B.3.7) on NORD-SM is probably a good indicator of language violations:

<table>
<thead>
<tr>
<th>PUNNING-operations, i.e.</th>
<th># of times used (static counts)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROTING coercions via the COERCE- and similar macros</td>
<td>400</td>
</tr>
<tr>
<td>LOCATE and AT:</td>
<td>40</td>
</tr>
<tr>
<td>SYN:</td>
<td>50</td>
</tr>
<tr>
<td>NOINIT:</td>
<td>150</td>
</tr>
<tr>
<td>EXECUTE- and misc. IO- instructions:</td>
<td>50</td>
</tr>
<tr>
<td>Resetting RANGE and CHECK toggles:</td>
<td>30</td>
</tr>
<tr>
<td>Accessing BREG, PREG and STATUSREG:</td>
<td>80</td>
</tr>
<tr>
<td>≃800 (out of 18.7K source lines).</td>
<td></td>
</tr>
</tbody>
</table>

Note that UNSAFE-constructs are absolutely necessary, if we want to abolish assembler. But their usage has been restricted to a minimum, often inside system macros and operators. Surprisingly few errors did originate from their application - cf. error group 4.2 in sec. 12.1.2.1 - probably because a good notation for doing dirty things means a lot.
12.2.4. Structured programming and MARY.

ADVANTAGES:

* MARY's arsenal of declarative mechanisms are, maybe, our most important structuring tool. Preludes and separate compilation are essential for program modularization. Variant-generation is achieved through conditional compilation, using macros to select COM-MOC pairs.

Note that we may shuffle declarations of local variables into the innermost possible scope (e.g. inside loops), without introducing extra block-entry overhead.

* Initializations in variable-declarations make programs more readable.

* Program names may consist of up to 63 significant characters.

* End-of-line comments are practical.

* An uncrunch-option on the compiler causes programs to be reformatted with proper indentation.

* No (explicit) labels have been used in RUNSYS, except in very low-level programs. In fact, labels are mostly used as label variables in small system routines for entering/entering RECURS routines. MARY's IF-, CASE- and DO-clauses and special exit constructs make labels superfluous elsewhere.

DISADVANTAGES:

* As declaration of program symbols must precede their application, top-down programming will not always imply a "top-down" way of physically writing the programs.

* Expression language and operator expressions may, if used intensively, blur program readability. This has not represented a practical problem.

* Many common MARY symbols (':=', '=:', ':=', '=:+') look alike. It doesn't seem to cause any confusion.
* Automatic coercions, like DEREFing on the right hand sides of assignments (':='), may cause misunderstanding of the underlying semantics. - I've done two such errors:

Ex. Derefering of right-hand sides in assignments.

```
INT I;
REF INT RI:=NIL;
-----
I:=RI;            % Modifies RI.
I:=(RI);         % The same. Modifies the INT-variable,
I:=RI:-INT;      % that RI points at.
```

However, to require a special DEREF-symbol - like '↑' in PASCAL, and possibly an inverse REF-symbol for construction of pointers, seems like over-specifying.

Admittedly, coercions may be hard to grasp. But once understood, few errors are later committed.

* The only serious objection to readability of real-world MARY programs is the huge amount of declarations we must remember, to understand what's going on inside expressions. This problem is typical for any language offering powerful declarative mechanisms, and can only be solved by a de-luxe cross-ref generator and by reducing the name space of available data items and operations. See next section.

This declarative complexity is rather a property of the system we're describing, than of the language itself; but any SIL ought to contain mechanisms to describe complex systems comprehensively.

12.2.5. Improvement proposals to MARY.

First we shall treat some technical details. Then the language question will be discussed a bit more generally.
12.2.5.1. Missing facilities in MARY.

There are still a few non-implemented features in MARY: frozen templates, slicing of PACKed ROWs, space generators. See MARY TEXTBOOK sec. 17.4 for an implementation status.

Some tentative new facilities could be:

* A prelude should be allowed to contain EXTERNAL-declarations of data or routines, that are ENTRY-declared in the very module using the prelude. This will significantly reduce trivial declaration conflicts in shared preludes.

* A syntactic construct to "open" a struct-object, like INSPECT in SIMULa or INlTH in PASCAL.

* Better ways for treatment of abnormal exits of routines, see [Rain, 1973].

* A suitable formalism to specify non-standard routine interfaces, cf. [Wulf, 1973].

12.2.5.2. Design flaws in MARY.

There are some "holes" in MARY, or at least in the current implementation. We shall concentrate on the more serious ones.

* Data being explicitly allocated in named areas (like registers) should always be initialized in SAFE MARY, even if they have "innocent" modes (like INT). Otherwise, we might make non-portable assumptions on the initial values of such variables:

  Ex. 100*J:=K;  % On NORD-SM, AREG will contain the value of K.
  INT I AREG:=0;  %% Initialization should be required!

* Scope checks on stack pointers can, in general, only be carried out on the "block"/routine level, not on "local scopes" inside blocks. Thus stack cells, to which we are building pointers, should never overlap any other cell.
Ex. Scope violation in pointer assignments.

```plaintext
PROC P=\% (REF INT X, REF REF INT Q):(X=\%:Q:=REF INT);
  REF INT RI:=NIL;
  BEGIN
  INT I;
  P(I, RI); % Calling a routine, P, that
  END; % performs I:=RI;

BEGIN
  CHAR C:=\'*\'; % C may overlap I.
END;
```

In this example either the REFing coercion of I in P(I, RI) should be forbidden; or, preferably, C should not be overlapped with I. However, the latter solution is difficult to implement on a one-pass compiler.

* SET-variables, whose internal value space doesn't include "zero", will presently receive illegal values in NIL-assignments:

Ex. Meaningless NIL-assignments to SET-variables.

```plaintext
SET (5- TO 3-) I:=NIL; % I gets an illegal valu
SET ('A' TO 'Z') C:=NIL; % Analogously.
[INT A, SET (10 TO 100)B]V:=NIL; % Likewise.
```

* Assignments to multiple-word data objects (>3 words on NORD-SM) may result in temporarily inconsistent values (see also next section):

Ex. Dubious multiple-word assignments.

```plaintext
MODE M=[10] ROW CHAR;
M X:=NIL,
  Y:=NIL;
<Fork child tasks, that also are accessing Y>;
X:=\%Y; % Intermediate values of
  \% Y may be meaningless!
NIL:=\%Y; % Likewise.
```
12.2.5.3. How to structure data and algorithms in a good SIL (System Implementation Language)?

This subject is tremendous, and I will only touch upon it.

The problem for a SIL (or a MOHLL) is to achieve a sufficient level of abstraction, without losing machine-orientation or run-time efficiency.

That is, is it possible to describe implementation details (e.g. data representations) orthogonally to logical data properties, thus gaining flexibility and portability?

Furthermore, we face a problem of restricting access to data and routines. For instance, normal scope rules for program names are much too liberal [Shaw, Wulf, 1973].

A large research activity in programming languages is currently devoted to these problems; cf. [Palme, 1976], [Krieg, 1974], [Wulf, 1974], [Liskov, 1974], [DeRemer, Kron, 1974], [Wang, 1974] an the work in IFIP WG 2.4.

There seems to be a common agreement that a data type should contain, in addition to classic data attributes, a set of algorithms (routines, operators, macros) that work on these attributes. I.e. SIMULA's class concept for concatenation of data and code has been rediscovered. See [Dahl, Nygaard, Myhrhaug, 1969].

Only a subset of the attributes and algorithms of a data type should be declared accessible outside the data type. Thus selected data attributes (e.g. system-controlled ones) can be protected from undesired changes, and normal programs don't have to make any assumptions about the internal representation of data.
Ex. We should be able to define a data type, STACK, with two primitive operators, POP and PUSH, being the only operations allowed on STACK-objects. How a stack is maintained is only the business of POP and PUSH.

SIMULA's hierarchic prefix-mechanism may add further elegance to this scheme, see also [Digernes, 1972] pp. 81-98.

Lastly, we might want to define allocation- and deallocation-operations for data, e.g. for SHARED- and SEMA-objects. This is another SIMULA idea.

In this area, MARY has a weak point. Not only is access of data/routines too generous (although VAL-status, modals and separate compilation provide some protection), but the physical layout of data objects influences the processing algorithms too much.

Generally speaking, how should we restrict access of any declared name, not merely local names inside data type declarations? I.e. which programs may access variable X or routine R? Let's hope we know some of the answers in a few years.

Anyhow, we will always need a good XREF-processor; to keep track of where a name is declared; as what; where it is used; in what context (a variable may be evaluated, assigned to, REFed, DEREFed, used as a routine parameter, etc.); and preferably to draw graphs over routine call-hierarchies.

12.2.5.4. Concurrent execution of program statements.

Any programming language, including a SIL, is a tool for expressing algorithms to be executed on a computer. How the underlying hardware or operating system really works, is usually irrelevant for the semantics of normal program operations (data accessing, looping, branching, subroutine calls, data allocation).
It's therefore not surprising that the semantics of concurrency or parallelism rarely can be expressed in existing programming languages. One may argue that the timeslicing of one CPU or the usage of several CPUs introduces no conceptually new features - except the need of synchronization. But this is exactly what we want to express!

MARY is a serial language, capable of utilizing the computational power of monoprogrammed von Neumann-machines; and not the CDC6000, ILLIAC or STAR. Although MARY is oriented somewhat against multiprogramming (reentrant code), there are other languages that offer more explicit constructs for describing concurrency:

Ex. ALGOL68 has collateral clauses, parbegin-parend clauses, and sema variables.

Ex. Concurrent PASCAL [Brinch Hansen, 1973] has shared variables and a variety of critical region-statements, but many problems are unsolved: How to resolve deadlocks, and how to formalize general proc calls within critical regions? I regard Brinch Hansen's PASCAL-version more as a teaching toy, than as a practical tool for writing operating systems.

Frankly, I don't have the faintest idea on how FORK-, ACQUIRE- and RELEASE-operations could be safely and efficiently implemented as a part of MARY or any other SIL. I guess that manual insertion of synchronization calls also will be needed in the nearest future.

Note: Not all shared resources need to be synchronized upon accessing. Recursive routines are reusable, and will only require a new data incarnation per activation. Constant (VAL) data can also be safely accessed by multiple users.

Generally, a datum being modified by a single task and read by several ones, needs no (software) synchronization; if the modify operation can be expressed as an atomic (hardware) store-operation. Examples: PARENT- and TIMELIMIT-attributes of TASK-objects.
13. Some thoughts on hardware design and operating systems.

A compiler will tackle the majority of hardware idiosyncrasies. Yet, I got rather mad sometimes, being forced to compensate in software for what really should have been done in hardware.

I am not the first person to complain about insufficient collaboration between hardware and software people, but I fear I will not be the last.

13.1. A plea for adequate interrupt-handling mechanisms.

* On NORD-SM, daisy-chain searching is required to identify IO-interrupts. This can easily be repaired by letting the A-register, also being saved by hardware, indicate the actual device number.

* The best solution is to associate a routine with every source of external interrupts through an interrupt vector. Thus on sensing interrupts, we automatically wake up inside the correct routine with some interrupt information as its parameters.

13.2. Addressing structures.

The basic addressing structures on any machine have a profound effect on the implementation of software on this machine. In the following, I shall briefly describe some common addressing modes and discuss their relevance to operating systems, with a special reference to RUNSYS:

* Base/index registers and hardware stacks support code reentrancy.

* Program-relative addressing and an accessible P-register make it easy to write location-independent code.
* A **uniform** addressing mode - for system and user processes, for code and data - makes process **interaction** easy. E.g. any task may execute a shared piece of code without bother-some address transformations; likewise for data inter-

* **Relocation registers**, usually combined with a memory **protect** system, may do a decent job; **If** there is a clear division between the "system" (not needing relocation) and user tasks; **If** user tasks have their data and code contiguously allocated in memory; **If** user tasks don't share data and code; **If** there is no dynamic virtual memory system (other than to shuffle a task's **entire** memory area in and out, cf. the arbitrary place-

If some of the above **If's** are not satisfied, we may experience unpleasant address transformations between tasks; as mentioned above.

* **Paging** with **fixed** page sizes will provide good "external" memory utilization, and facilities for relocation **and** virtual memory.

As when using relocation registers, communication between tasks may be difficult; if they don't share the same page tables.

Efficiency considerations on paging systems can be found in sec. 4.8.

* **Segmentation**, i.e. partitioning of memory into segments of **variable** length, eliminates some of the drawbacks of paging: Segment boundaries become natural, it's often easy to utilize code reentrancy and inter-task communication is straight-forward.

Memory segments, containing code or data, ought to be admin-

*strated by hardware **descriptors** (see also [Organick, 1973]):
Ex. Burroughs-like descriptors.

This means that a pointer, or a generalized memory address, becomes a descriptor, occupying more than one hardware word on 16-bits machines (probably 3 words). The entire instruction repertoire must be altered correspondingly, giving interrupts on segment faults.

If extra status-bits on hardware words could have been used to indicate pointers, it would have simplified garbage collects.

The efficiency of segmentation systems is dealt with elsewhere (secs.4.7-4.8).

13.3. Monitor mode and memory protection.

Since MARY gives full security against wrong memory accesses and un-authorized instructions, there is no need for a special monitor mode or a memory protect system! Admittedly, a few system routines written in UNSAFE MARY will do "dirty" things, but they must under any circumstances be able to do so in "monitor mode", where there is no hardware protection.

The non-existing distinction between user- and system routines/tasks assumes, for instance, that FORTRAN-routines are not allowed to do illegal indexing and similar things.
13.4. Some aspects of multi-processor systems.

Note: a "processor" may be a CPU, as well as an I0-channel.

The recent development in micro-processors have made multi-processing more fascinating than ever. However, some classic problems related to processor-communication and -utilization ought to be analyzed, before we configure the hardware:

* How to resolve access conflicts to swappable memory pages/segments? E.g. how to "lock" a page temporarily, to prevent it from being swapped, during an I0-transfer via some non-standard I0-channel?

* How to ensure efficient memory access, given typical processor-memory architectures: uni-bus, multi-bus, switch-bar?

* The CPUs may work on shared data and they may exchange messages. Underlying inter-processor synchronization can however be rather delicate; cf. sec. 3.3.4.10.

What I'm trying to say is, that bying a lot of cheap micro-processors is no standard solution to increased computing capacity. Memory-processor interface and general software design (task implementation, synchronization primitives, sharedness of routines/data) must be closely coordinated to obtain an efficient system.

I would personally estimate RUNSYS to be well-suited to run on multi-CPU configurations, if the CPUs are equivalent in respect to priority and addressing modes. Sec. 14.2.3.2 in the portability chapter contains further details.
14. Portability of RUNSYS.


Portability has many aspects: Do we mean portability of compilers for language X, or of programs written in X? How high do we tolerate transfer costs to be, before a program becomes "nonportable". Shall we say 20% of the original development costs?

Portability and efficiency tend to be contradictory qualities: generality versus machine-orientation. Thus, systems programs are costlier to port than user programs.

The advantages of portability should be well-known: Lower production costs of programs; higher reliability as there is only one (symbolic) version to maintain.

Satisfactory portability is not easy to get at. Although we may get a lot of help from the actual implementation language, a "portable" program system must be designed with portability in mind from the very beginning, e.g. by proper modularization and parameterization (using macros).

14.2. Sources of nonportability.

We ought to distinguish between 4 levels of nonportability of operating systems in general and MARY RUNSYS in particular:

14.2.1. Low-level machine peculiarities.

This is typically I/O- and interrupt-handling. It cannot be described semantically in MARY, but represents a rather small and well isolated part of the total code (§2%).


Problems associated with different word lengths - like data packing and accuracy of INT-operations - are solved by letting the compiler perform packing/unpacking, and by relying on MARY's
capability to redefine the mode INT and all INT-operators ('+', '-', '>', '<', etc.). We must, however, be careful in pointer arithmetic, if the address space is larger than the set of positive INTs.

Yet, a more powerful data type concept than MARY's, as described in sec. 12.5, can make data descriptions even more implementation-independent.

Incompatible instruction sets are handled by the compiler itself, and through implementation-defined operators.

Most register usage is automatically portable in MARY.

The majority of proc call sequences (to RECURS, non-RECURS and register routines) are definable in MARY, thus portable. See MARY TEXTBOOK, sec. 16.4.

Note: Although a piece of MARY code appears to be technically portable, e.g. by removing register specs in variable decs, it may lose its efficiency. Later tuning, like insertion of new register specs, may improve upon this.

14.2.3. Machine-dependencies, not expressable in the programming language.

14.2.3.1. Different addressing structures.

Necessary background material has been discussed in sec. 13.2. We may conclude that RUNSYS is expected to be highly portable onto machines:

that are stack-oriented,

that facilitate location-independent code, and

that don't have paging, relocation registers, memory protection or a special monitor state - that is, they offer a uniform addressing and operational mode.
Hardware features like paging might, of course, be disabled or ignored.

On a Burroughs-like descriptor-machine, it will be natural to transplant our OLAY-code mechanisms from software to hardware, thus gaining efficiency. In fact, we got most of our ideas on OLAY from Burroughs. OLAY-data can be efficiently implemented as a special heap facility on such machines.

A possible application of hardware paging in RUNSYS is, as mentioned, for virtual data objects. But we will run into difficulties, if we try to exchange virtual data addresses between tasks, e.g. from user tasks to I0-drivers.

Finally, a PDP8-like hardware (only two addressable "pages") is definitely not suited to host RUNSYS.

14.2.3.2. One or several CPUs.

Necessary changes from one to several CPUs are rather trivial; if the CPUs are "indifferent" to each other (no master-slave system). Only the basic task-management needs adjustment; arbitrary tasks don't notice the difference. This flexibility originates from our uniform (stack-)implementation of tasks.

The modifications are:

* to use a different mechanism for low-level synchronization operations (using a core-swap or a test-and-set instruction),

* to implement global task variables, like RUNNING and STACKDESCRIPTOR, slightly differently.

* to apply some ad-hoc tricks during initializations, e.g. which CPU should execute the initialization code?

Furthermore the interrupt-handling must be slightly modified, e.g. which CPU receives which I0-interrupts? The existing READYQ-organization can, however, be used almost unchanged.
14.2.3.3. Non-standard I/O-equipment.

Peripheral devices represent a constant source of surprises for systems programmers.

Just consider the number of "carriagereturn-linefeed" combinations for TTY's or the described missing-connect interrupts on SM4.

14.2.4. Design goals.

Different design goals certainly make operating systems incompatible and hence nonportable. No matter how "portable" a batch-oriented op.sys. on machine X has been designed, it will not be well-suited for real-time applications on machine Y.

The general usefulness of RUNSYS will be discussed in the next chapter.

14.2.5. To sum up.

I hope, although I have not proved it through practice, that RUNSYS will be fairly simple to move from the current SM4-machine to NORD1, KS-500 (with several CPUs), NORD10 (not using the hardware paging), PDP11/20, INTEL8080 and a lot of similar mini- and micro-computers.

Estimated amount of work: between 6 and 30 man-weeks to transport the entire RUNSYS, given a MARY compiler for the new machine, which will require ≃2 additional man-years.

The machine should have at least a primary storage of 32K (preferably 48K) 16-bits words, and a drum/disk with a ≥1 M-words storage capacity and an average access time of ≤40 ms.
14.2.6. A small case example.

An early version of MINIRUNSYS (≈2 K source lines), containing some rather "hairy" machine-oriented sections, was April 1975 moved from SM4 to NORD10, where it was interfaced to run under the SIN-TRAN II virtual operating system. (We had to use the segmentation system as a "dynamic loader", on top of the underlying paging system, to be able to run the MARY-MARY compiler in a 64 K virtual address space.)

This transfer of MINIRUNSYS required 1½ weeks of work (by me) and revealed 7 errors:

* 2 coding errors in low-level OLAX-enter and -exit routines.

* 2 illegal-instruction errors, as SINTRAN II didn't tolerate DISABLEINTERRUPTS-ENABLEINTERRUPTS pairs.

* 2 errors caused by inconsistent parameter transfers (wrong use of registers) to system routines.

* 1 buffer-size error.

The last 3 errors could be accounted for as misunderstandings between myself and the other implementators.

One of the reasons behind this rather successful porting operation is that we have parameterized the source code to obtain "variant-generation" via alternative macro expansions.
15. Evaluating RUNSYS

Detailed evaluations can be found in the respective chapters. This one will concentrate on the more general issues.

15.1 On the implementation of processes

15.1.1 General remarks

The process or task concept of RUNSYS is general and rather orthogonally implemented. User (-controlled) tasks are easily created (FORKed). A RUNSYS task may acquire a diversity of resources (memory space, mass storage files, misc. SHARED-objects). The tools available for process interaction (ACQUIRE-RELEASE routines) are general, safe and easy to use. All in all, the traditional monitor facilities in RUNSYS seem satisfactory.

Since the organization of OS-interface from user processes (or system processes not being low-level drivers) still arouses academic interest and controversy, some further comments on our Burroughs-inspired cactus-stacks will be given.

15.1.2 On OS-interface, monitors versus cactus stacks

In the OS-literature the word "kernel" or "monitor" is frequently used to denote that part of an operating system being allowed to perform forking, synchronization, space allocation, etc.. That is, that part of the system that updates (shared) system resources. To protect the access of such resources/data structures, (user) tasks will have to enter/exit the monitor via special (hardware) monitor calls. Furthermore, many machines must operate in a special "monitor mode" to be able to execute certain privileged instructions. (Note, however, a possible . design-tautologi here!) Thus, "inside" such a monitor, updating of vital system resources can safely be carried out.
The design of "monitors" may vary. We have the monolithic ones, where only one process at a time may be allowed to enter. Prof. C.A.R. Hoare has defined a more sophisticated type [Hoare, 1974], where each group/module of related (system-?) routines and data may be declared to constitute a monitor. That is, monitor access-lockings can be made more selective. Provisions also exist to allow other processes to take over temporarily, if the current inside-monitor-process may have to wait, e.g. on I0-transfers.

Still, I will regard the synchronization/protection mechanism of the above monitors to be unnecessarily restrictive and therefore inefficient (what if we have several CPUs?, or consider the selective parent-file lockings during file openings in RUNSYS). Given a high-level language to guarantee that calls on system routines conform to the declarations of such routines, I cannot see why selected system routines be necessity have to be executed in an exclusively locked manner inside monitors; alternatively by system tasks only (if this is how monitors are being implemented).

- Surely we need synchronization of critical regions, but not as coarse as this.

Apart from being inefficient and non-orthogonal (monitor routines behave differently than other routines), a monitor-solution is faced with the traditional problems such as:

* How to allocate temporary working-space for directly/indirectly called monitor routines? This must be done dynamically, if we allow user tasks to "leave" a Hoare-monitor temporarily. This is the well-known recursive-coroutine dilemma, which in our case is solved by a cactus stack. (On Ull08, the standard system heap generator EXPOOL is used to provide proc incarnations).
* We must prevent a sequence of updating operations on system resources to be aborted before completion, because of resource shortage (CPU-time, memory space). For instance, during a file packing, the actual monitor routine will probably charge the requesting task for the CPU-time spent. If the initial amount of such is insufficient, what then? Cf. what happens on the U1108, if we experience a max-time abortion during a VPACK-operation (Put thy thrust in the file backup?)

In RUNSYS, the NOOFUPDATES-attribute has been introduced to delay task abortions in such cases. However, I guess a monitor-solution can handle such resource deadlocks better than RUNSYS does — and may do.

The only, really annoying drawback yet discovered in RUNSYS during updates of system resources is this: The ownerships of truly global data objects, being allocated by some arbitrary (user) task, need adjustment. E.g. FILE-objects, created by an OPENFILE-routine may have a longer life-time than the task that performs the initial opening operation. Again, a classic monitor will probably handle such situations more naturally. However, a general heap facility (see sec. 4.10) would have eliminated the problem.

Finally, let me emphasize one aspect of RUNSYS routines/tasks, that have proved particularly advantageous:

Cactus-stack-implementation of tasks implies that an arbitrary system RECURS routine (e.g. LOCKFILES) may be called in the same way (no recompilation required of the callee or the called routine), and itself executed with unchanged semantics(with identical calls on synchronization routines), regardless of whether the routine is executed:

strictly sequentially (e.g. by the MINIRUNSYS stand-alone system),

in normal quasi-parallelism, or

in true parallelism (by several CPUs).
The uniformism in this set-up meant, for instance, that substantial parts of RUNSYS could be separately tested in calm, reproducible environments. See chapters 9 and 10.

Technical note: Cactus stacks will require RECURS routines to be implemented so that they don't assume contiguous stacks; cf. the discontinuity between stacks of parent and child tasks, and also SIMULA's heap generator for proc incarnations.

15.2 Multi-user qualities

They are indeed excellent.

Code is reentrant thus sharable, and usually OLAYable. Example. Of the 9 K FILESYS code only the OLAY-code descriptors (≈ 150 words), and a FILEIO-routine (≈400 words) need be resident once files have been allocated/opened/locked.

As mentioned, hierarchical FILESYS-files will reduce locking conflicts to a minimum.

Furthermore, there exists a START-command to initiate (new) batch tasks.

Lastly, don't let us forget the time-sharing facilities of the system, granting normal users access to practically all computational resources of the actual machine.

15.3 Efficiency

Performance data for the virtual segmentation system have been given elsewhere.

The system will trash, if only a single user executes the MARY-MARY compiler. A future multi-pass compiler will possibly reduce the amount of trashing, if there on the average is maximum one compilation concurrently active.

If the users mainly perform edit and file operations, trashing is no problem on 32-48K machines.
The system has been tested with 3 interactive terminals and 2 batch compilation jobs concurrently active. The response times were found to be acceptable for interactive purposes.

The SM4-implementation can handle ≈/UU interrupts/second, which on a 64 K machine will support 5-6 interactive 300-bauds terminals doing meaningful operations.

See also sec. 16.2 on future work.

15.4 Security

A few comments only.

There still remain some problems with unintentional task abortions (resource deadlocks, insufficient parameter checking in system routines). However, the system has not yet been subjected to a permanent users' work-load, so I really don't know its practical robustness.

On the other hand, I have the last 1 1/2 years personally not registered a single software crash of the disk's master file. The entire FILESYS seems almost immune against faulty file operations, like locking/writing on closed files.

15.5 Design flaws, previous and present ones

No really drastic global re-designs have yet been required, due to inherent design failures.

Admittedly, a lot of (local) tuning and improvements have been carried out. For instance, OLAY-code management has been revised once, basic space allocation three times, IO-drivers twice, task termination routines twice, FILESYS once (when introducing linked files), etc.

Furthermore, I fear/hope that there remain some undiscovered bottlenecks in the booted MARY compiler, apart from the well-known trashing anomaly:
The number of proc calls needed to compile one machine-code instruction is approx. 180, which seems rather high even if CPU-time is regarded to be plentiful. - But, once again, this compiler has been developed beyond my direct influence, so I am not the right person to articulate such a criticism.

15.6 General usefulness of the system

15.6.1 Application areas

RUNSYS represents a fascinating playtoy/simulation tool for concurrent processes. It is, however, designed and intended to be more than just a private hobby project for the implementor.

Since the MARY-MARY compiler presently is incapable of producing reliable code (July 1976), the full power of RUNSYS cannot yet be utilized. This again implies, that the system cannot be properly tested and debugged, as no other language processors are (yet) running under the system.

The system is, however, fully applicable as a multi-user editor/file machine. In fact, with a reasonably fast TTY-line between the lab's SM4 and the University's main U1108 it can be (and partly is) used as a local program-editing machine.

Although the major effort in RUNSYS has been put in creating high-level software environments for program development, the generality of the system should make it fairly well-suited for traditional mini-computer applications, such as communications control and front-end functions. Inadequate efficiency may, however, limit the application area for real-time functions, that expect ≥700 interrupts per second (present, untailed SM4-version).
15.6.2 A MARY machine?

The system is coded in MARY; i.e. assuming a reliable MARY compiler for the actual target machine. The system is expected to support at least a MARY-MARY compiler; or more generally to facilitate segmented, overlayed execution of any (large) MARY program.

However, it's nothing that prevents people from writing other language processors (preferably in MARY), and letting these run under the system, and letting the translated programs be loaded and executed afterwards. We must only provide that all code (including OS-interface operations) being executed by RUNSYS, has been software-checked to guarantee benevolent actions (cf. also "SAFE" MARY). Since this requirement is rather severe and the system interface probably intricate, it is possible that we in practice have eliminated non-MARY programs.

15.6.3 Possible target machines

The basic machine requirements (stack-orientedness, self-relocation, absence/avoidance of monitor mode) have been dealt with in the portability chapter (ch. 14). Although the portability prospects may look promising, little practical work has been done in this area.

The major obstacle, when trying to move RUNSYS to another target machine, will probably be to finance the costs (≈ 2 manyears) of the underlying MARY compiler. However, when deciding a possible new MARY-implementation, we should also consider the substantial amount of standard MARY systems software (incl. RUNSYS) becoming available at a moderate price.

15.7 A few comparative remarks about other systems

RUNSYS is a high-level operating system (OS), intended to run on fairly small machines. Maybe the ambitions of the system are too high, compared to the resources (manpower, hardware equipment) available? Presently, these Norwegian
minis are mostly run under some modest real-time systems: SINTRAN I [ND, 1972], KOS [KV, 1973a].

However, on the NORD10-variant with hardware paging a general, virtual OS (SINTRAN III, [ND, 1975a]) has been developed, thus demonstrating that an OS of RUNSYS-like ambitions is far from unrealistic on these machines. I would, rather say it represents a sound way to exploit their full computational potential—as emphasized in the introductory chapter.

In recent years quite a few medium-sized, orthogonally-designed operating systems have emerged in the literature, although in varying state of completeness and practical usefulness. Cf. VENUS [Liskov, 1972], UNIX [Ritchie, Thompson, 1974], BOSS 2 [Lauesen, 1975], CAL [Lampson, Sturgis, 1976]. Obviously, RUNSYS represents a trend in the development of Operating Systems.

As a contrast, we have the existing gigantic operating systems found on the U1100-series, IBM 360/370-series, CDC 6000-7000 series, and so on. However, the code size of such systems (200-500 K) is not that much larger than a complete RUNSYS- and compiler-configuration (≈160 K). This indicates that the task of implementing reasonably general operating systems is not excessively resource-demanding, given the tools proper. Cf. also the initial comments on the Burroughs systems.

I will not give any further comparisons (functional or technical) between RUNSYS and other systems. There is not time for me to do so; neither can RUNSYS in its present state be subjected to realistic benchmarks. I also regard it to be beyond the scope of this thesis.
16. Conclusion

16.1 Lessons learned

When looking back on what has been done, there are two factors that strike me:

1) Writing systems software in a high-level language, compared to assembler, is like manufacturing books before and after Gutenberg. I have had an intense feeling of creative joy by mastering a high-level language to implement such complex systems as operating systems really are. The previously described rewrite-, tuning- and enhancement work would have been unthinkable without a MARY-equivalent language.

2) The relative magnitude between a simple monitor and a reasonably general operating system is \( \approx 20 \). (I started with \( \approx 800 \) drafted lines of basic monitor functions; now I have \( > 18 \) K lines, and there is still undone work). Clearly, I've underestimated the necessary effort to come up with an efficient, functional and reliable system. However, I have a feeling that I now have passed the mountain top and can begin to look into the promised land.

16.2 Future work

I have presently spent \( \approx 4 \) months on tuning the olay-system, as I regard efficiency to be a crucial issue. Still, there remains some global tuning-work, which will require usage of the existing proc-trace facility.

However, future efforts should be concentrated on improving the system's user-orientedness: automatic scratch-files, improved error detection to avoid user-task abortions, and similar features.

I also have plans to adapt the system for more real-time-oriented applications (computer graphics, net-work functions),
just to test its generality, but nothing concrete has been done. Qui vivra, verra.

As for implementations on other machines, I'm presently limited to the NORD-SMs because of the MARY compiler. A RUNSYS-version for KS500 (a multi-CPU SM-machine) will probably be running before Christmas this year (1976), while a NORD1/NORD12-version will have to wait a bit further on.

16.3 Aftermath

When looking back on the work I have carried out, I cannot be completely satisfied - the work is not finished, neither quantitatively nor qualitatively. However, there is not much work that remains (3-4 man-months), before what we may call a production-version is ready.

Working on MARY, and later on RUNSYS, has undoubtedly represented the most creative and stimulating intellectual period in all my life. It's almost so that one forgets about the pile of sweat that has been transpired during the last three years. - And isn't that one of the things that makes life worthwhile?

*
References.

0. [Boehm, 1973]

1. [Brinch Hansen, 1970]

2. [Brinch Hansen, 1973a]

3. [Brinch Hansen, 1973b]

4. [Burroughs, 1969]

5. [Burroughs, 1970]

6. [Børseth, 1975]

7. [Coffman, Varian, 1968]
8. [Conradi, 1974a]

9. [Conradi, 1974b]

10. [Conradi, 1974c]

11. [Conradi, Holager, 1975]

12. [Corbató, Vyssotsky, 1965]

13. [Courtois et al., 1971]


15. [Denning, 1968]
16. [Denning, 1970]
P. Denning: "Virtual Memory", Computing Surveys, Vol. 2.,

17. [Dennis, Horn, 1966]
J. Dennis, E. van Horn: "Programming Semantics for Multi-
programmed Computations", CACM 9, 3 (March 1966),
p. 143.

18. [DeRemer, Kron, 1974]
F. DeRemer and H. Kron: "Programming-in-the-large versus
Programming-in-the-small", UCLA Santa Cruz,
Sept. 1974, 35 p. Presented at the meeting of

19. [Digernes, 1972]
T. Digernes: "An Implementation Language for the UNIVAC 1108".
Masters Thesis, Norwegian Institute of Technology,

20. [Dijkstra, 1968a]
E.W. Dijkstra: "Cooperating Sequential Processes".
In: F. Genuys (ed.): "Programming Languages",

21. [Dijkstra, 1968b]
E.W. Dijkstra: "The Structure of THE Multiprogramming System",

22. [Garwick, 1964]
J.V. Garwick: "The definition of programming languages by
their compilers". In: T.B. Steel (ed.): "Formal
language description languages for computer pro-
23. [Green et al., 1976]
G. Green et al.: "MARY LINKING LOADER. Users Guide",

24. [Grodås, Gulbrandsen, 1974]
M. Grodås, O. Gulbrandsen: "Editor for SM-4", Prosjektoppgave
ved IDB, NTH, mai 1974, 200 s.

25. [Habermann, 1969]
A.N. Habermann: "Prevention of System Deadlocks",
CACM 12, 7 (July 1969), pp. 373-385.

26. [Hoare, 1974]

27. [Hoare, Perrott, 1972]
C.A.R. Hoare, R.H. Perrott (Eds.): "Operating Systems Principle:

28. [Holager, 1973]
P. Holager: "A Consolidation of Synchronization Primitives",

29. [Holt, 1972]
Computer Surveys, Vol. 4, No. 3 (Sept. 1972),
pp. 179-196.

30. [Ichbiah et al., 1973]
J.D. Ichbiah, J.P. Rissen and J.C. Heliard: "The two-level
approach to data independent programming in the
LIS system implementation language", paper at
the IFIP/TC-2 Working Conference in Trondheim,
31. [Klunder, 1973]
   J. Klunder: "Experiences with SPL", paper at the IFIP/TC-2

32. [Knuth, 1971]
   D.E. Knuth: "An Empirical Study of FORTRAN Programs",
   Software - Practice and Experience", Vol. 1,
   1971, pp. 105-133.

33. [Krieg, 1974]
   B. Krieg: "Increasing the Flexibility and Reliability of
   the Class Concept", Technische Universität München,
   the meeting of IFIP WG 2.4 in Berlin, Dec. 1974.

34. [KV, 1973a]
   System - KOS", KV-1-1901-7303, KV, Kongsberg,

35. [KV, 1973b]
   Kongsberg Våpenfabrikk: "SM General Purpose Computers.
   Programmers Instruction Manual", KV-5-1575-7305,

36. [KV, 1974]
   Kongsberg Våpenfabrikk: "KS 500 General Purpose Computers.
   Programmers Instruction Manual", D-1-2029-7411,
   KV, Kongsberg, Norway, 1974.

37. [Kvitsand, 1971]
   S. Kvitsand: "Documentation of the Programming Language
   PL 1108", Masters Thesis, Norwegian Institute

38. [Lampson, Sturgis, 1976]
   B. Lampson, H. Sturgis: "Reflections on an Operating System
   (About CAL.)
39. [Langmaack, 1973]

40. [Lauesen, 1975]
S. Lauesen: "A Large Semaphore Based Operating System", CACM 18, 7 (July 1975), pp. 377-389. (About BOSS2)

41. [Liskov, 1972]
B. Liskov: "The Design of the VENUS Operating System", CACM 15, 3 (March 1972), pp. 144-149.

42. [Liskov, 1974]

43. [Madsen, 1972]

44. [MARY group, 1974]
MARY group: "How to use MARY's IO-packages", Internal (RUNIT-)note, August 1974, 10 p.

45. [McKeag, 1971]
R. M. McKeag: "Burroughs B5500 Master Control Program", The Queen's University of Belfast, Northern-Ireland, 1971.

46. [ND, 1970]
47. [ND, 1972]
Revision 2, ND-60.026.01, ND, Oslo, Feb. 1972.
(Contains SINTRAN I-information too.)

48. [ND, 1973]
Norsk Dataelektronikk: "NORD PL, Users' Guide", Revision 8,
Publ. No. ND-60.047.01, Oslo, Aug. 1973, approx.
30 p.

49. [ND, 1975a]
Norsk Dataelektronikk: "SINTRAN III. Users' Guide", (Pre-
liminary), ND-60.050.01, ND, Oslo, 1975 (?).

50. [ND, 1975b]
Norsk Dataelektronikk: "Introduction to NORD-10", Revision 7,
ND, Oslo, July 1975.

51. [Organick, 1973]
E. Organick: "Computer System Organization. The B5700/B6700

52. [Palme, 1976]
J. Palme: "New Feature for Module Protection in SIMULA",

53. [Rain, 1973]
M. Rain: "A Possible Solution of the Subroutine Calling
Interface Problem in Machine-Oriented Languages",
paper at the IFIP/TC-2 Working Conference in

54. [Rain et al., 1973]
M. Rain et al.: "MARY—A Portable Machine-Oriented Programming
39. [Langmaack, 1973]

40. [Lauesen, 1975]
S. Lauesen: "A Large Semaphore Based Operating System", CACM 18, 7 (July 1975), pp. 377-389. (About BOSS2)

41. [Liskov, 1972]
B. Liskov: "The Design of the VENUS Operating System", CACM 15, 3 (March 1972), pp. 144-149.

42. [Liskov, 1974]

43. [Madsen, 1972]

44. [MARY group, 1974]
MARY group: "How to use MARY's IO-packages", Internal (RUNIT-)note August 1974, 10 p.

45. [McKeag, 1971]
R. M. McKeag: "Burroughs B5500 Master Control Program", The Queen's University of Belfast, Northern-Ireland, 1971.

46. [ND, 1970]
55. [Randell, 1969]

56. [Rekdal, 1974]

57. [Ritchie, Thompson, 1974]

58. [Shaw, Wulf, 1973]

59. [Solberg, 1976]

60. [Steele, 1975]

61. [Sætermoen, 1976]

62. [UNIVAC, 1975]
Sperry Rand Corp.: "SPERRY UNIVAC 1100 Series Executive System Programmer Reference (UP 4144)", Sperry Rand, USA, 1975.

62a. [Van Vijngaarden, 1975]
63. [Wang, 1974]

64. [Wichmann, 1970]

65. [Wirth, 1968]

66. [Wulf, 1973]

67. [Wulf, 1974a]

68. [Wulf, 1974b]

69. [Wulf et al., 1971]

70. [Wulf et al., 1973]
71. [Wulf et al., 1975]

Appendix A. The MARY project.

The project started in late 1970. The existing software situation on the Norwegian NORD-SM mini-computers ("minis") motivated RUNIT (The Computing Centre at the University of Trondheim) and KV (Kongsberg Våpenfabrikk, the SM-manufacturer), with support from NTNFi (The Royal Norwegian Council for Technical and Scientific Research), to initiate a project to produce a MOL for the NORD-SMs.

These mini-computers have a wordlength of 16 bits. They have 6 general registers including an accumulator, an index register and a base register. Their interrupt systems have 16 or 1 level(s). It is easy to write reentrant code for them. The NORD1, NORD20, NORD10, SM3, SM4 and KS-500 have almost identical instruction sets, so it should be worthwhile to construct one compiler for all these machines, and use compiler options to indicate the actual target computer. For machine descriptions, see [ND, 1970], [ND, 1975b], [KV, 1973b] and [KV, 1974].

After some studies in the field of programming languages ALGOL68 was chosen as a host language, because of its general data types and orthogonal structure. The concepts stolen from ALGOL68 were polished, rationalized and extended. In addition, ideas from PL360 (on register allocation) and PASCAL (on scalar types) were incorporated. Of course, some theoretical innovations of our own were also merged into the language.

In parallel with the language design a one-pass pilot compiler in NU ALGOL/U1108-ASSEMBLER was developed. The language structure changed rapidly and violently the first two years (creating an enormous amount of compiler bugs), but since autumn 1972 only minor changes have been made in the language.

The pilot compiler consists of 23K of NU ALGOL source lines +2K of assembly code. It occupies 65K code (unsegmented) +16K of data. It's a large beast, even on our UNIVAC 1108! It is now quite stable and is used by more than 15 active MARY programmers, including 4 from the MARY implementation group.
A MARY-MARY production compiler (one-pass, 24K source lines, 120K instructions) to run on the Norwegian minis has been written, but is not yet fully debugged.

In order to execute large MARY programs on the minis (including the compiler), a so-called 'OLAY' feature for automatic code segmentation has been implemented.

A multiprogramming system, MARY RUNSYS (being the subject of this thesis), has been written to support execution of OLAY-segmented MARY programs. RUNSYS will need at least 32K primary storage to run an OLAYed MARY compiler on the minis.

Because of inadequate loaders on the NORD-SMs (we can't do fixup on partial words), a separate MARY linking loader had to be written.

About one manyear of additional effort remains (as of July 1976) before a complete "MARY system" is available, including a basic multiprogramming system, a MARY compiler, a file system, a job control interpreter, an editor and miscellaneous service functions. A lot of extra software is on the way, mostly written by senior students.

It seems appropriate to survey the MARY activities up till this date.

The MARY implementation group has consisted of:

Mark Rain, autumn 1970 - spring 1973: Language design, pilot compiler writing, some monitor functions, PR-activities.

Per Holager, spring 1971 - : Language design, compiler writing, documentation, teaching.


Ole Solberg, autumn 1972 - : MARY-MARY compiler.

Reidar Conradi, spring 1973 - summer 1975: MARY RUNSYS, language design, documentation, teaching.
By July 1976, about 15 man-years had been spent on the project.

**Basic MARY software produced by the above group:**

The MARY-MARY compiler, about 24K source lines.
Remaining MARY RUNSYS, about 18K source lines.

**Student activities, completed ones:**

* A Masters thesis drafting a MARY link-editor, 1000 source lines.
* A Masters thesis on a SNOBOL interpreter, 2500 source lines.
* A project thesis on an EXEC-8 like file system (2 students), 3500 source lines.
* A project thesis on a context-sensitive editor (2 students), 1500 source lines.
* A summer project on a small stand-alone file system, 1500 source lines.
* A Masters thesis on tolerance limits for machine tools, 3000 source lines.
* A project thesis on a small RC-4000 like monitor (3 students), 2000 source lines.
* A Masters thesis on a simple multi-user query system, 2000 source lines.
* A Masters thesis on a MARY compiler for the U1108 (3 students), approx. 8000 (new) source lines.
* A Masters thesis on a dedicated on-line system, including a second-order file system, a report generator, a FORTRAN compiler, a command analyzer and some extra monitor functions (7 students). Approx. 9000 source lines.
* A project and a Masters thesis on a MARY compiler for the INTEL8080 (2 students), approx. 10000 (new) source lines.
* A project thesis on communication software to interface U1108 and SM4, approx. 2000 source lines.
* A Masters thesis on a mode checking link-editor for MARY, approx. 3000 source lines.

Lastly, let me add personally, that without the knowledge, brain, working capacity and never failing enthusiasm of Mark Rain, the MARY project would have ended 3½ years ago with a traditional MOL as a result. - This would probably have pleased our top managers.
We're now in the situation that we have a rather powerful tool for writing efficient, reliable and portable systems software. We had originally no intention of writing more MARY software than the MARY-MARY compiler and some basic service routines.

However, in order merely to run the compiler on the minis, a good chunk of additional software was needed. Also, potential customers/users were reluctant to use a new and strange language, until they saw the benefits demonstrated and the system running.

Hopefully, this picture is about to change. The language has proved to be a tool for practical use (see discussion in sec. 12.2), and the entire system should be running within a few months.

"There are three ways to lose money: women, gambling and engineers. The two former are the more comfortable ones, the latter the most certain", bankier Rotschild.

Appendix B. A very quick survey of the MARY language.

B.1. Language structure.

Ex. A small demonstration program.

BEGIN
    INT SUM:=0;
    [50] INT VECTOR:=NIL; % Declarating an INT array
    % of 50 elements.
    FOR ELEM IN VECTOR % Scanning the array.
        WHILE INT():=ELEM>0 % '=' is a store-operator.
            DO ELEM + SUM:=SUM; OD; % Computing the sum of all
            % vector elements.
    END;

B.1.1. Primaries.

Simple constants: 1, '*', TRUE.
Displays: (1,2,3), 'ABC', 2.0, 4'FF00'.
Simple variables: IDENT, X.
Selections: P.SUC, HEAD.NAME.
Array elements and-slices: A[I], A[I TO J].
Proc calls: P(I,J), ININT().
Casts, i.e. explicit type conversions: RI:-INT.
Empty clauses: (Nothing).
Closed clauses, i.e. serial clauses
    BEGIN---END.
    (---).
embedded in parentheses:
    IF---FI.
    CASE---ESAC.

B.1.2. Serial clauses.

A serial clause is a sequence of expressions/declarations/DO-clauses, separated with semicolons. Declarations may be mixed with expressions and DO-clauses.
Ex. Serial clause.

```
INT N:=100,
SUM;
N*N+10:=SUM;
OUTINT(SUM,7);  % Writing: '101010'.
```

B.1.3. IF-, CASE- and DO-clauses.

IF- and CASE-clauses are closed clauses. A DO-clause is not, since it always yields no value (=VOID).

Ex. IF-clause.

```
IF A>B
THEN
   OUTINT(A,7);
ELSE
   ERROR();
FI;
```

Ex. CASE-clause.

```
CHAR CH:= INCHAR();
CASE CH IN
   'A' TO 'Z': --, % Selecting a CHAR-value.
   '0' TO '9': --,
   ','','': --
OUT
   --
ESAC;
```

Ex. DO-clauses.

```
WHILE N<10
   DO
   OD; % A simple WHILE-construct.
```
FOR I IN 1 TO N  % The controlled loop variable, 
WHILE A>B  % I, is automatically declared.
  DO  ---
  OD;

FOR ELEM IN VECTOR  % Scanning the VECTOR-array.
  DO
    OUTINT (ELEM,7);
  OD;

B.1.4. Expression language.
Every language construct, including classic statements and 
declarations, has/yields a value of a specified data type or 
mode. The empty value has mode VOID.

Ex.  3+
BEGIN INT I:=INTINT();
  I*I
END +K:=J;
% This value is yielded by the 
% closed clause.
% I.e. 3+<some INT-value>+K:=J;

Ex.  IF I>J
    THEN
      0:=I;
      J  % Yielding value of J,
    ELSE
      I  % or value of I
      FI  :=K;  % - to be stored into K.

B.2. Operators and expressions.

MARY has no operator priorities. Expressions are evaluated 
from left to right, within the same parenthesis level. Monadic 
operators trail their operands. Operators may be declared and 
re-declared, see  B.3.4.
Ex.  I+J*5  \hspace{1cm} \%\text{ Means } (I+J)^5.  \\
    J=*I  \hspace{1cm} \%\text{ Means } (J-)^*I.  \\
    I=J \text{ AND } (K<0)  \hspace{1cm} \%\text{ Parentheses must be used!}

Some important operators will now be commented upon.

B.2.1. The store-operator, '='.

This is MARY's assignment operator. It yields the value being stored.

\textbf{Ex.}  50=:I; I=:J;  \hspace{1cm} \%\text{ Means } 50=:I; \ 2500=:J;  \\
\textbf{Ex.}  \text{ WHILE ININT()=:I>0 DO -- OD; } \%\text{ Treating only non-negative }  \\
\hspace{1cm} \%\text{ integers.}

B.2.2. The Boolean AND-and OR-operators.

They leave the second operand unevaluated, if the first operand is FALSE and TRUE, respectively.

\textbf{Ex.}  IF PTR ISNT NIL AND (PTR.VALUE = KEY) THEN---FI;

B.2.3. Variants of the store-operator.

\textbf{Ex.}  I=:+SUM;  \hspace{1cm} \%\text{ Increments SUM by I.}
\textbf{Ex.}  I \text{ MAXAB } J;  \hspace{1cm} \%\text{ J is set to the larger. The suffix }  \\
\hspace{1cm} \%\text{ 'AB' means 'And Becomes'.}
\textbf{Ex.}  \text{ INCR+N=:N=:INDEX; } \hspace{1cm} \%\text{ ':' is like '='}, but yields  \\
\hspace{1cm} \%\text{ the former right-hand value as result.}

B.2.4. The GO-operator.

GO is an ordinary language operator, performing program jumps.

\textbf{Ex.}  \text{ GO ROUTINE; } \hspace{1cm} \%\text{ Exits current routine.}  \\
\hspace{2cm} \text{ GO LOOP; } \hspace{1cm} \%\text{ Exits current loop.}  \\
\hspace{4cm} \%\text{ 'ROUTINE' and 'LOOP' are instantaneous }  \\
\hspace{4cm} \%\text{ labels, declared by the compiler.}
B.2.5. Shift operators.

Ex.  

| BITS   | B:=4'FF00',            | % 'FF00' is a display of hex bytes. |
|        | BB:=4<<B;              | % Left circular shift operation, 4 times. |
|        | 4:=->B;                | % Right logical shift of B 4 times. |

B.3. Declarations.

All items must be declared before use (one-pass restriction).


Ex. Simple cell decs.

| INT     | I:= ININT(),          | % Initializations are allowed (and |
|        |                         | % sometimes required).             |
|         | FIVE=5,                | % A READ-ONLY integer.             |
|         | PROD=I*FIVE;           | % FIVE has mode VAL INT.           |
|         |                         | % Analogously.                     |
| REF     | INT RI:=I;            | % A pointer to integers.           |
| REF     | VAL INT RVI:=FIVE;     | % A pointer to constant integers,  |
|         |                         | % namely VAL INTs.                |

Cells are allocated by default on the stack. Other allocation strategies must be specified explicitly:

Ex. Location specifications.

| [1000] INT TABLE IN COMMON:= NIL; % A common block, named 'TABLE' |
| INT A AREG:=I;          | % 'A' resides in AREG, a   |
|                         | % machine register.       |
| [0 TO 15] REF TASKHEAD HEADS AT 16; % 'HEADS' is put at absolute |
|                         | % addresses 16-31. "UNSAFE" |
|                         | % MARY!                  |
| A:=I*FIVE;             | % ':=' means that 'A' (i.e. ARE |
|                         | % should be used as a    |
|                         | % working register.      |
B.3.2. Mode declarations.

A declared mode name (like INT) can be used instead of the mode description itself.

* SET modes.

Ex. SET VOID=(), % The empty set.
       PROTEAN=(NIL), % This is a universal mode.
       BOOLEAN=(FALSE,TRUE), % Set of logical values.
       FIXED=(.,0,1,2,3,4,...,2147483647), % Available set of integers.
       CHAR=(.,'A','B','C',............), % The set of characters.
       WEEKDAYS=(MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY,
               SATURDAY,SUNDAY); % A user-defined set.

* SUBSET modes.

Ex. MODE INT=SET (2**15- TO 2**15-1), % Note constant limits!
    WEEKEND=SET (SATURDAY TO SUNDAY);

* Multiples (arrays).

Ex. MODE BUFFER= [80] CHAR,
       BITS= [16] PACK BOOL, % Mapping the binary contents of a machine word.
       MATRIX= [10][10] INT; % A multiple of multiples.

* Structures (data packages, records).

Ex. MODE NODE= [ REF NODE NEXT,
            PACK CHAR INFO, PACK BOOL ACTIVE,
            INT NO,
            [10] PACK VAL CHAR NAME];

* Pointers.

Ex. MODE POINTER= REF BUFFER;
* ROWs (array descriptors for selecting a slice or subpart of an array)

Ex. MODE STRING = ROW CHAR;
STRING T := NAME;  % Now is T:

NAME [3 TO 4] := T;  % Now is T:

% T[1] := NAME[3], after the last % assignment.

* PROCs.

Ex. MODE PROCMODE = PROC RECURS (INT A, B) INT;

% The described PROC takes two INT parms, yields an INT, and % is recursive.

* Labels.

Ex. MODE LABELMODE = LABEL;  % Very rarely used!

* Templates, parameterized modes.

Ex. Static templates.

MODE LIST(M) = [REF LIST(M) NEXT, M VALUE];
LIST (INT) HEAD := (NIL, l);  % Textual substitution of % 'LIST(INT)' is performed.

Ex. Dynamic templates, useful in general list processing.

MODE DLIST(M = (VOID, INT, LONGINT, REAL))
= [REF DLIST() SUC, PRED,
[4] CHAR NAME,
M INFO ] ;

DLIST(INT) INTOBJ := (NIL, NIL, 'YRAM', 1974);  % A hidden type-fie:
% (to indicate 'INT' % will also be allo-
% cated.

REF DLIST() TAIL := INTOBJ;

A special CASE-clause exists for selecting between dynamic template variants. See MARY TEXTBOOK, p. 181.
B.3.3. Routine declarations (a special kind of cell declarations).

All parameters are transferred "by value". A value may be of any mode, incl. REFS and PROCs.

```plaintext
  optional proc.mode
Ex. PROC 'RECURS (INT A,B) INT' MAX=% RECURS (INT A,B) INT:
    IF A>B THEN A ELSE B FI;  % Routine body, a primary.
    MAX(I,10):=J;
```

B.3.4. Macro- and operator declarations.

Invocation of a macro or an operator implies textual substitution of the corresponding macro/operator text.

```plaintext
Ex. Macro decs.
    DEFINE SIZE=10$,         % No macro parms.
        MAX(A,B)=IF A>B THEN A ELSE B FI$;  % Two parms.
```

In MARY, macros can be used to obtain conditional compilation via the _COM-MOC_ formalism for comments, cf. MARY TEXTBOOK, p.146.

```plaintext
Ex. Operator decs.
    OP =:+(VAL INT I, INT A)=(A+I:=A)$,
        AND (VAL BOOL B1,B2)= IF B1 THEN B2 ELSE FALSE FI$,
    - (VAL INT I, VAL VOID V)="monadic minus"$;
```

Declarations of _AREAs_ (e.g. machine registers) and space generators will not be described. Please consult MARY TEXTBOOK, ch. 15. Some aspects of space allocation (incl. HEAP generators) are, however, treated in sec. 4.10.

B.3.5. Misc. constructs.

* _MODE_ (expr.). Yields the mode of the supplied expression.
* _OPERANDIZE_ <mode>. Yields the data size in words of a <mode>-object.
Ex. Mode functions.

```
INT I;
OPERANDSIZE MODE(I)    %% Yields 1.
OPERANDSIZE [100] INT  %% Yields 100.
```

* FREE <symbol> . An anti-declaration: Frees the supplied symbol and yields its value, if it's a cell.

Ex. Register freeing.

```
INT A AREG;
----
INT SAVEA:= FREE A;  %% Frees the A-register, and yields
                      %% its value.
```

B.3.6. Coercions (type conversions).

A coercion can only be applied in a limited number of contexts, please consult MARY TEXTBOOK ch. 14.

Common coercions:

- **DEREFING**, indirect addressing. E.g. `REF INT ⇒ INT`.
- **REFING**, construction of a pointer. E.g. `INT ⇒ REF INT`.
- **DEROWING**, indexing/slicing via a `ROW descr`. E.g. `T[I]`.
- **ROWING**, construction of a `ROW descr`. E.g. `NAME[3 TO 4]`.
- **WIDENING**, expanding a SET value. E.g. `INT ⇒ LONGINT`.
- **NARROWING**, shrinking a SET value. E.g. `LONGINT ⇒ INT`.
- **VOIDING**, discarding a value (before semicolon). E.g. `5=:I;`.
- **PROTING**, squeezing a specified value into a universal PROTEAN value. E.g. `PROTEAN P:=5;`.
- **DEPROTING**, interpreting a PROTEAN value as a specified value. E.g. `P=:RI;`.
- **DEREFING, DEROWING, ROWING and NARROWING** may imply generation of run-time tests on the coercends.
- **PROTING** is only allowed in UNSAFE MARY, see below.
B.3.7. UNSAFE MARY.

MARY contains an unportable superset, UNSAFE MARY, enabling us to do otherwise illegal operations, e.g. "dirty" type conversions. Only selected systems programs have to be written in UNSAFE MARY. It may happen that such code sections (e.g. space generators) are portable in spite of their unsafeness, but this is not guaranteed by the language. UNSAFE MARY contains the following extensions:

B.3.7.1. The PROTING coercion, AMODE $\Rightarrow$ PROTEAN.

Since the inverse DEPROTING coercion, PROTEAN $\Rightarrow$ AMODE, already exists in the language, we are now able to convert any mode into another mode. PROTING is mostly used for "pointer conversions":

Ex. Dirty type conversions ("punning").

```assembly
DEFINE COERCEM = : - PROTEAN : - M$;  % A type conversion macro.
% ':-' is a cast symbol.
REF INT RI := 5 COERCEM (REF INT);  % RI now contains the
% INT-address 5.
```

B.3.7.2. AT, SYN and NOINIT in cell declarations.

Ex. UNSAFE data allocations.

```assembly
[1000] REF NODE TABLE NOINIT;  % Omitting
% required initializations.
INT START := 10;
REF [1000] INT PTR SYN START NOINIT;  % START and PTR are
% mapped on to each other.
NIL := (PTR);
LABEL LASTADDR AT 0 NOINIT;  % Clearing locs. 10-1009.
% Resides in location 0.
```

B.3.7.3. LOCATE for allocation of code.

Ex.

```assembly
LOCATE
----- }
% This code-section starts at absolute
% address 32.
AT 32;
```
B.3.7.4. The EXECUTE builtin function.

EXECUTE is used to emit an arbitrary bit pattern into the code stream.

Ex. SM4-instructions.

\[
\text{DEFINE } \text{DISABLEINTERRUPTS}=\text{EXECUTE}(\text{'D142'}); \\
\text{ENABLEINTERRUPTS}=\text{EXECUTE}(\text{'D102'}); \\
\text{DISABLEINTERRUPTS;} \{ \\
\text{<crit.code section>} \} \{ \\
\text{ENABLEINTERRUPTS;} \} \\
\text{%%% Note the effect of macros for } \\
\text{%%% program readability and -parameter} \\
\text{%%% ization.} \\
\]

B.3.7.5. Miscellaneous builtin functions.

On NORD-SM there are 8 IO-instructions and a WAIT-instruction available in this way.

Ex. Basic IO-operations.

\[
\text{CHAR C AREG:=1';} \{ \\
\text{DO OD UNTIL IOT(3);} \} \{ \\
\text{%%% Waiting for TTY-output.} \\
\text{%%% Go to sleep.} \\
\]

B.3.7.6. Misc. details.

RESETting of the CHECK and RANGE , to avoid run-time tests on empty pointers and likewise, is only permitted in UNSAFE MARY.

On NORD-SM: PREG, BREG and STATUSREG are only available in UNSAFE MARY.
Appendix C, part 1: Common JEDEC editor operations.

TTY1
*** NEW TASK: USER TASK

MARY RUNSYS AT YOUR SERVICE
SAVE LIMIT, INCREMENT (DECIMAL) AND PROJECT ID:

> 14000 RC
JOB CONTROLLER IS ACTIVE
READY
> @BEGIN;
> @ *** CONTENTS;
*** NEW TASK: USER TASK
2 SUBFILES OF: ***
  1: RC  2: FIL
  2: LTS  2: FIL

USER TASK
NORMAL TASK TERMINATION

> @ RC CONTENTS INFO;
*** NEW TASK: USER TASK
9 SUBFILES OF: RC
  1: SCR  5: DAT
  2: RELOC  6: DAT
  3: READFILE  3: DAT
  4: PRINTFILE  20: DAT
  5: G  3: DAT
  6: F  1: DAT
  7: TESTPROG  1: DAT
  8: DEMO  7: DAT
  9: GLAYFILE  7: DAT

USER TASK
NORMAL TASK TERMINATION
*** NEW TASK: USER TASK
FILE INFORMATION FOR: RC
FILEHEADPOS: 0
LEVEL: 1

MAXNOFS SECTORS: 2
STARTADDR: 16
NOFIRSTDATASECTION: 53

USAGE:
32
NOSECSUBFILES: 9
NOFSUBFILESUSED: 9

NEXTTRACK:
32 33 34 35
NOSECSUBFILES: 3

USER TASK
NORMAL TASK TERMINATION

(Contd.)
> @ G DELETE: RC PACK CONTENTS;
*** NEW TASK: USER TASK

USER TASK
NORMAL TASK TERMINATION
*** NEW TASK: USER TASK

USER TASK
NORMAL TASK TERMINATION
*** NEW TASK: USER TASK
8 SUBFILES OF: RC

1: SCR 5 DAT
2: RELOC 6 DAT
3: READFILE 3 DAT
4: PAINTFILE 20 DAT
5: F 1 DAT
6: TESTPROG 1 DAT
7: DEMO 7 DAT
8: OLAYFILE 7 DAT

USER TASK
NORMAL TASK TERMINATION

> @ LTS CONTENTS-IS: RC=+LTS CONTENTS;
*** NEW TASK: USER TASK
0 SUBFILES OF: LTS

USER TASK
NORMAL TASK TERMINATION
*** NEW TASK: USER TASK

USER TASK
NORMAL TASK TERMINATION
*** NEW TASK: USER TASK
8 SUBFILES OF: LTS

1: SCR 5 DAT
2: RELOC 6 DAT
3: READFILE 3 DAT
4: PAINTFILE 20 DAT
5: F 1 DAT
6: TESTPROG 1 DAT
7: DEMO 7 DAT
8: OLAYFILE 7 DAT

USER TASK
NORMAL TASK TERMINATION

A delete-pack operation.

Copying a compound file.

(Contd.)
@ TESTPROG ED TESTPROG;
*** NEW TASK: USER TASK
SCRATCH FILE:
> SCA
  0  0
> L 3
  3  0,17
> C /3/5/
*TEXT NOT FOUND*
> 0 C /3/4/
FOR I IN 1 TO 4
> +2 P
    FOR J IN 1 TO 10 DO OUTEX(); OD;
S  0
> C /10/7/
FOR J IN 1 TO 7 DO OUTEX(); OD;
> 0 DO +12 P
BEGIN
PROC SAVE RECURS OUTEX=EXTERNAL;
FOR I IN 1 TO 4
    FOR J IN 1 TO 7 DO OUTEX(); OD;
    INT K=I+10;
OD;
INT L=10;
(:10) INT DUMMY=NIL;
EXECUTE('A' 'F800');
END FINISH
10  0
>
J

USER TASK
NORMAL TASK TERMINATION

Editing a MARY-program
i.e. a SYMBOLIC file.

(Contd.)
>@ TESTPROG COMP RELOC;
*** NEW TASK: USER TASK

SIZE:

IO-DEVICES:
LIST, ETC.: OUTPUT ADR.: 

MARY COMPILER VERSION 3.0 (MARY/MARY) OF 5-4-1976
COMPUTER: SM 4
COMPILED ON 0/10-1975 AT 00:00:00.

X=EXTERNAL;

00: 00:
00003:004

UTHEX(); OD;

00:000D:00A
00:0010:003
00:0013:003
00:0015:002
00:001C:007
00:001D:001

NO ERRORS FOUND.
33 INSTRUCTIONS Emitted
11 SYMBOLIC IMAGES COMPILED, WITH ASYNTACTIC ENTITIES.
LAST OUTPUT ADR.: 0000

USER TASK
NORMAL TASK TERMINATION

> @ RELOC LOAD OLOADFILE;
*** NEW TASK: USER TASK
SCRATCH FILE:

@SCR

MARY COMPATIBLE BRF LOADER
>1000

>FS
*
0001

CURRENTLOCATION: 421A
OLOADCODESIZE: 0000 0000
NOOFLAYSEGMENTS: 0000
*
*
OUTHEX SIZE 8BITS 0005 DEF
OUTHEX 0229 DEF
RECENTADGRECENT 18AD DEF

CURRENTLOCATION: 421A
*
*
0801080108010801080108010801080108010801080108010801080108010801080108010801080108010801080108010801080108010801080108010801080108010801080108010801080108010801
USER TASK
NORMAL TASK TERMINATION

(Contd.)
> @ AC•READFILE ED READFILE:
*** NEW TASK: USER TASK
SCRATCH FILE:

> SCR
  0 0
  0 + P + P

> 0 0 + 0

>(AC CONTENTS:
  1 0
@BEGIN:
  2 0

> + P
@AC CONTENTS:
  3 0

> C /AC CONTENTS/G ALLOC 5/
  0 G ALLOC 5;

> + P
@ G COMP RELOC; RELOC LOAD OLAYFILE;
  4 0

> C /G/TESTPROG=G/
@ TESTPROG=G COMP RELOC; RELOC LOAD OLAYFILE;

> C ///
@ TESTPROG=G COMP RELOC RELOC LOAD OLAYFILE;

> + P
  1000
  5 0

> + P
SCR
  6 0

> 5 5 M6
V P + P
  1000
  6 1
S
  7 0

> - - P
SCR
  6 0

> 0 DJ +12 P
*MEANINGLESS POSITION*
  0 AC
@BEGIN:
  0 G ALLOC 5;
@ TESTPROG=G COMP RELOC RELOC LOAD OLAYFILE;
SCR
  1000
S
@AC END:

> 9 0

> - 3

V P
  0 G ALLOC 5;

> + C /RELOC //
@ TESTPROG=G COMP RELOC LOAD OLAYFILE;

> U

USER TASK
NORMAL TASK TERMINATION

> @ AC•READFILE START (AC•PRINTFILE,1&000,750);
*** NEW TASK: USER TASK
*** NEW TASK: USER TASK

USER TASK
NORMAL TASK TERMINATION

(Contd.)
33 INSTRUCTIONS EMITTED
11 SYNTACTIC IMAGES COMPILED, WITH 64 SYNTACTIC ENTITIES.
LAST OUTPUT FOR: 5000.
i.e. relocate load cวยfie.

i.e. job termination.

BEGIN
PROC PARENTPROC =% OLAY RECURS:
BEGIN
  MODE BUFFER = (: ( : 10 ) CHAR IMAGE, INT LENGTH );
  EVENT ( BUFFER ) VACANT = NIL ;
  FILLED = NIL ;
  (: 3 ) BUFFER BUFFERPOOL = NIL ;
  FOR P IN BUFFERPOOL
  DO
    RELEASE ( BUFFER, VACANT, P );
  OD ;

  (: EVENT ( INT ) SEM, INT RES ) FINISHED = NIL ;
PROC FLAG =% OLAY RECURS ( REF INT DUMMY ) :
  RELEASE ( INT, FINISHED, SEM, DUMMY ) ;
PROC PRODUCER =% OLAY RECURS ( ROW VAL ROW VAL CHAR TEXT ) :
BEGIN
  CREATENWUPSENTRY ( INT, FLAG, FINISHED, RES );
  ' PRODUCER IS ACTIVE ' ++ CCR = > ;
  FOR P IN TEXT
  DO
    REF BUFFER PB = ACQUIPE ( BUFFER, VACANT );
    P LEN G = RA . LENGTH ;
    P = PB . IMAGE ( 1 TO P LEN G );
    RELEASE ( BUFFER, FILLED, RR );
    ' * ' ++ CCR = > ;
  OD ;
END; %% OF PRODUCER .
PROC CONSUMER =% OLAY RECURS :
BEGIN
  CREATENWUPSENTRY ( INT, FLAG, FINISHED, RES );
  ' CONSUMER IS ACTIVE ' ++ CCR = > ;
  TO B
  DO
    REF BUFFER PB = ACQUIPE ( BUFFER, FILLED );
    RA . IMAGE ( 1 TO RA . LENGTH ) = > ; ' * ' ++ CCR = > ;
    RELEASE ( BUFFER, VACANT, RR );
  OD ;
END; %% OF CONSUMER .
EMITCONSTANTS:
' START FORKING ' ++ CCR = > ;
'TASKINFO II = ( STANDARD , PRIORITY , 2 ++ 11, 1200, 500 ) ;
FORK ( II , PRODUCER ( ' MARY ', ' HAN ', ' LAM ', ' LITTLE ', ' LAM ' ) ) ;
FORK ( II , PRODUCER ( ' HELLO ', ' MARY ', ' YUU ' ) ) ;
1 = : ++ 1, PRIORITY ;
FORK ( II , CONSUMER ) ;
TO 3
DO ; %% WAIT FOR ALL CHILDREN TO DIE .
    ACQUIPE ( INT, FINISHED, SEM ) ;
OD ;
'HUMRAH!! ' ++ CCR = > ;
END; %% OF PARENTPROC .
PARENTPROC ( ) ;
END FINISH

(Contd.)
** Job initiation **

MARY RUNSYS AT YOUR SERVICE
SAVELIMIT-INCREMENT (DECIMAL) AND PROJECT ID:

1000 RC
JOBCONTROLLER IS ACTIVE
READY

➢ TEMPO LOAD OLAYFILE;
*** NEW TASK: USER TASK
SCRATCH FILE:

➢ SCR

MARY COMPATIBLE BRF LOADER

1000 (Max. SAVE-space demand of
loaded program)

*

5
*

0005

CURRENTLOCATION: 36CS
OLAYCODESIZE: 0000 01A1
NOOPOLAYSEGMENTS: 0004
HISTOGRAM:
32DF8 : 0001 0000 0000 0002 0000 0001 0000 0000 0000 0000 0000
32E4 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
32FO : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

START FORKING

*** NEW TASK: USER TASK
*** NEW TASK: USER TASK
PRODUCER IS ACTIVE
****** NEW TASK: USER TASK
PRODUCER IS ACTIVE
CONSUMER IS ACTIVE
MARY *HAD *A *LITTLE *HELLO *LAMB MARY LOU
USER TASK
NORMAL TASK TERMINATION

USER TASK
NORMAL TASK TERMINATION

USER TASK
NORMAL TASK TERMINATION
HURRAH!! ***

USER TASK
NORMAL TASK TERMINATION

➢ RC END;
JOB TERMINATED

RC TASK
NORMAL TASK TERMINATION

** Loading precomp cons-prod program (see previous page.) **

- and executing it:

Output from cons-prod program.

** Job terminated **
APPENDIX B1

GENERAL RUNSYS PRELUDES.

PAGE 1, LINDA RSDECS
PAGE 6, REIDAR BASICDECS

- REL LINE 8: INDEBUG (MACRO)*
- REL LINE 9: OUTDEBUG (MACRO)*
- REL LINE 36: FORCE (MACRO)*
- REL LINE 37: COERCE (MACRO)*
- REL LINE 38: FULL (MACRO)*
- REL LINE 39: GLOBAL (MACRO)*
- REL LINE 45: LEAVE (MACRO)*
- REL LINE 57: RETURN (MACRO)*
- REL LINE 68: BYTE (MODE)*
- REL LINE 69: DLIST (MODE)*
- REL LINE 73: LIST (MODE)*
- REL LINE 77: ADJUST (OP)*
- REL LINE 79: ADJUSTAB (OP)*
- REL LINE 86: SEMITRACE (OP)*
- REL LINE 94: OUTOLAYTEXT (MACRO)*
- REL LINE 99: INFILE (MACRO)*
- REL LINE 100: OUTFILE (MACRO)*
- REL LINE 104: => (OP)*
- REL LINE 154: CCR (MACRO)*
- REL LINE 155: TARGET (MACRO)*
- REL LINE 156: DISABLEINTERRUPTS (MACRO)*
- REL LINE 157: ENABLEINTERRUPTS (MACRO)*
- REL LINE 158: EMITCONSTANTS (MACRO)*
- REL LINE 159: LOADREG (MACRO)*
- REL LINE 163: BUILDRW (MACRO)*

PAGE 9, REIDAR BASICDECS
PAGE 10, REIDAR DEBUGDECS

- REL LINE 26: DUMP (MACRO)*
- REL LINE 38: ENTER (MACRO)*
- REL LINE 39: EXIT (MACRO)*

PAGE 12, REIDAR DEBUGDECS
PAGE 13, REIDAR TASKDECS

- REL LINE 8: STANDARDPRIORITY (MACRO)*
- REL LINE 9: FORK (MACRO)*
- REL LINE 21: SDEGEN (MACRO)*
- REL LINE 23: SGEN (MACRO)*
- REL LINE 25: SGENROW (MACRO)*
- REL LINE 38: MULTIPLEADDR (MACRO)*
- REL LINE 39: SDEGENROW (MACRO)*
- REL LINE 45: EXECREQUEST (MODE)*
- REL LINE 51: METYPE (MODE)*
- REL LINE 59: MEMLINK (MODE)*
- REL LINE 60: EVENT (MODE)*
- REL LINE 61: TASKINFO (MODE)*
- REL LINE 71: REQUEST (MACRO)*

PAGE 15, REIDAR TASKDECS
PAGE 16, REIDAR TASKDECS2

- REL LINE 11: INTOQ (MACRO)*
- REL LINE 12: OUTOFQ (MACRO)*
- REL LINE 24: AREALINK (MODE)*
- REL LINE 25: ACTION (MODE)*
- REL LINE 46: SEGMENTDESCRIPTOR (MODE)*
- REL LINE 50: PROCOBJECT (MODE)*
- REL LINE 51: RETURNDESCRIPTOR (MODE)*
TABLE OF CONTENTS FOR APPENDIX D

- REL* LINE 52: CRITINFO (MODE).
- REL* LINE 56: STACKDESC (MODE).
- REL* LINE 86: TASK (MODE).

PAGE 18, REIDAR*TASKDODECS
- REL* LINE 11: MINSPACE (MACRO).
- REL* LINE 13: FREEMEMLINK (MODE).

PAGE 19, REIDAR*SPACEDDEC5
- REL* LINE 11: MINSPACE (MACRO).
- REL* LINE 14: FREEMEMLINK (MODE).

PAGE 20, REIDAR*SPACEDDEC5
- REL* LINE 26: IOINSTRUCTION (MODE).
- REL* LINE 29: INTERRUPTLEVEL (MODE).

PAGE 21, REIDAR*SPACEDODECS2
- REL* LINE 15: DINSERTAFTER (MACRO).
- REL* LINE 8: COREADDRESS (MODE).

PAGE 22, REIDAR*LISTDDEC5
- REL* LINE 11: EXECUTEINSTR (MACRO).
- REL* LINE 13: MOVE (MACRO).
- REL* LINE 16: BASEOFFSET (MACRO).
- REL* LINE 26: IOINSTRUCTION (MODE).
- REL* LINE 29: INTERRUPTLEVEL (MODE).
- REL* LINE 30: MEMREFERENCE (MODE).

PAGE 23, REIDAR*INITSPDEC5
- REL* LINE 16: TRIGGERTASK (MACRO).
- REL* LINE 49: EXECREQ (MACRO).

PAGE 24, REIDAR*INITSDDEC5
- REL* LINE 7: DEVICETYPE (MODE).
- REL* LINE 13: IOREQUESTTYPE (MODE).
- REL* LINE 20: SYMBOLICFILE (MODE).
- REL* LINE 23: IOFILE (MODE).
- REL* LINE 34: DEVICETITLEENTRY (MODE).
- REL* LINE 54: DISKSECTORSIZE (MACRO).
- REL* LINE 56: DEVICE (MODE).
- REL* LINE 76: DISKBUFFER (MODE).

PAGE 25, REIDAR*MACHDODC5
- REL* LINE 8: TTYBUFFER (MODE).
- REL* LINE 14: BACKLOGININFO (MODE).
- REL* LINE 18: BASICREQUEST (MODE).
- REL* LINE 29: IOREQUEST (MODE).
- REL* LINE 34: TTYDEVICE (MODE).

PAGE 26, REIDAR*MACHDODECS
- REL* LINE 10: IOREAD (MACRO).
- REL* LINE 11: IOWRITE (MACRO).

PAGE 27, REIDAR*INTPTDDEC5
- REL* LINE 20: DISKINFO (MODE).

PAGE 28, REIDAR*INTPTDDEC5
- REL* LINE 7: TICKTOCKENTRY (MODE).
- REL* LINE 19: TIMEQUANTUM (MACRO).

PAGE 29, REIDAR*DEVDDEC5

PAGE 30, REIDAR*DEVDDECS2

PAGE 31, REIDAR*DEVDDEC5

PAGE 32, REIDAR*DEVDDECS2

PAGE 33, REIDAR*DISKIO
- REL* LINE 10: IOREAD (MACRO).
- REL* LINE 11: IOWRITE (MACRO).

PAGE 34, ANN*DISKINFO

PAGE 35, REIDAR*CLOCKDDEC5

PAGE 36, REIDAR*CLOCKDDEC5

PAGE 37, REIDAR*CLOCKDDEC5

PAGE 38, REIDAR*MAINDDEC5
- REL* LINE 26: ALLOWEXTERNALINTERRUPTS (DATA).
- REL* LINE 36: AREAHEADS (DATA).
- REL* LINE 42: REQUESTEVENT (DATA).
- REL* LINE 43: TASKSINKQUEUE (DATA).
- REL* LINE 45: SYSTEMLOADPHASE (DATA).
<table>
<thead>
<tr>
<th>PAGE 70, REIDAR*TASKGEN</th>
<th>REL* LINE 133: ACQUIREDEVICE (PROC)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>REL* LINE 147: RELEASEDEVICE (PROC)*</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PAGE 79, REIDAR*IODRIV</th>
<th>REL* LINE 61: SETCRITICALEXIT (PROC)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>REL* LINE 210: FORKER (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 301: CONSOLE (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 336: ABORTTASK (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 513: TASKSINK (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 523: ENDOFTHENEWSWORLD (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 537: POWERFAILURE (PROC)*</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PAGE 95, REIDAR*INITIODRIV</th>
<th>REL* LINE 43: DISKDEVICE (DATA)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>REL* LINE 41: IDREQUESTS (DATA)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 42: AVAILABLEECHOES (DATA)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 43: TTYBUFFERPOOL (DATA)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 62: ADDENVNUM (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 192: TTYINPUTDRIVER (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 373: TTYOUTPUTDRIVER (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 428: LINEPRINTERDRIVER (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 459: CARDREADERDRIVER (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 573: PAPERTAPEDRIVER (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 611: DISKDRIVER (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 784: BACKLOGQUEUE (DATA)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 785: PRINTFILESEQ (DATA)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 791: IDLEDEVICEHANDLER (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 941: PRINTFILES (PROC)*</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PAGE 98, REIDAR*IOINTERFACE</th>
<th>REL* LINE 71: INITIDLEDEVICEHANDLER (PROC)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>REL* LINE 62: TTYIN (DATA)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 63: TTYOUT (DATA)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 64: CREADER (DATA)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 65: PRINTER (DATA)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 66: TREADER (DATA)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 67: TPUNCH (DATA)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 69: INITINFIL (DATA)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 70: INITOUTFILE (DATA)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 158: REALOWNERTASK (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 183: CREATENEWFILE (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 258: FORGETBUFFER (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 390: GETIMAGE (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 511: PUTIMAGE (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 614: DISKIO (PROC)*</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PAGE 109, REIDAR*IOPROCS</th>
<th>REL* LINE 52: HEXNUMBERS (DATA)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>REL* LINE 81: GETCHAR (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 93: FIRSTNONBLANK (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 101: gettext (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 116: GETPTEXT (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 151: PUTTEXT (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 165: PUTPTTEXT (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 183: PUTCHAR (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 188: NEWLINE (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 198: GETBYTE (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 201: PUTBYTE (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 208: FILLTEXT (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 213: FILLFORMATBUFFER (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 228: DEFORMATINT (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 274: GETINT (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 280: INTPWER (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 292: FORMATINT (PROC)*</td>
<td></td>
</tr>
<tr>
<td>REL* LINE 326: PUTINT (PROC)*</td>
<td></td>
</tr>
</tbody>
</table>
**TABLE OF CONTENTS FOR APPENDIX D**

- REL* LINE 349: GETHEX (PROC)*
- REL* LINE 367: PUTHEX (PROC)*
- REL* LINE 414: TRACETULITY (PROC)*

**PAGE 117, REIDAR INTERRUPT**
- REL* LINE 28: REQUESTHANDLER (PROC)*
- REL* LINE 75: TASKCHANGER (PROC)*

**PAGE 120, REIDAR INITI**
- REL* LINE 45: INITBOOTTASK (DATA)*
- REL* LINE 230: INITFORKER (PROC)*
- REL* LINE 415: SAVEBREG (DATA)*
- REL* LINE 472: INITIALIZE (PROC)*
- REL* LINE 515: LETGO (PROC)*

**PAGE 129, REIDAR TREATINTPRPT**
- REL* LINE 29: INTERRUPTHANDLING (PROC)*
- REL* LINE 167: SYSTEMTASKQ (DATA)*
- REL* LINE 240: CHANGENVIRONMENTLABEL (DATA)*
- REL* LINE 280: LOADENVIRONMENTLABEL (DATA)*

**OLAY MANAGEMENT AND MINIRUNSYS CODE**

**PAGE 135, LINDA OLAYCODEDEFS**
- REL* LINE 5: STACKDRIVER (MODE)*
- REL* LINE 9: OLAYCODEHEAD (MODE)*

**PAGE 137, LINDA NEDISKIO23**
- REL* LINE 15: DISKPOOL (DATA)*
- REL* LINE 22: DISKIO (PROC)*

**PAGE 139, LINDA OLAYCODE3**
- REL* LINE 22: OLAYCODEBUFFER (DATA)*
- REL* LINE 25: SEGCOUNT (DATA)*
- REL* LINE 33: OLAYCODESTAT (DATA)*
- REL* LINE 193: MAKEPRESENT (PROC)*
- REL* LINE 210: OLAYPROCDATASIZE (PROC)*

**PAGE 143, LINDA MINIRUNSYS4**
- REL* LINE 22: BASICTASK (DATA)*
- REL* LINE 612: FORGETSPACE (PROC)*
- REL* LINE 639: GETSPACE (PROC)*

**PAGE 157, LINDA INTRINSICS3**
- REL* LINE 63: SAVECODEDESCR (DATA)*
- REL* LINE 65: CURRENTCODESEGDESCR (DATA)*
- REL* LINE 67: STACKDESCRIPTOR (DATA)*
- REL* LINE 69: SEGMENTCALLCOUNT (DATA)*
- REL* LINE 73: OLAYCODESTART (DATA)*
- REL* LINE 138: MAKEPRESENTENTERADDRESS (DATA)*
- REL* LINE 201: OLAYENTER (PROC)*
- REL* LINE 254: OLAYEXIT (PROC)*
- REL* LINE 310: SAVETOOLDAYEXITLABEL (DATA)*
- REL* LINE 349: RECENTROURECENT (PROC)*
- REL* LINE 411: RECEXRUORECEX (PROC)*
- REL* LINE 454: EXITWITHOUTRECURSCUTOBACK (PROC)*
- REL* LINE 500: OLAYTOSAVERCURSENTER (PROC)*
- REL* LINE 567: OLAYTOSAVEENTER (PROC)*
- REL* LINE 607: RECEVENTROUREVENT (PROC)*
- REL* LINE 678: OLAYTOSAVEPROCRCURSENTER (PROC)*
- REL* LINE 741: OelayPROCRCURSENTER (PROC)*
- REL* LINE 808: SLICEROUSLICE (PROC)*
- REL* LINE 876: COPYROTOCOPY (PROC)*
- REL* LINE 930: KLOBBERROUKLOBBER (PROC)*
- REL* LINE 976: UNPAKBROUENPAKB8 (PROC)*
- REL* LINE 1023: UNPAKBROUENPAK4 (PROC)*
- REL* LINE 1071: UNPAKBROUENPAK2 (PROC)*
TABLE OF CONTENTS FOR APPENDIX A

PAGE 177, LINDA OLAYDATADEFS
- REL* LINE 1120: UNPAKRIOUUNPAKI (PROC)*
- REL* LINE 1170: INSERTROUINSERT (PROC)*

PAGE 177, LINDA OLAYDATADECs
- REL* LINE 2: ALLOCSEG (MACRO)*
- REL* LINE 3: PUTSEG (MACRO)*
- REL* LINE 4: GETSEG (MACRO)*
- REL* LINE 5: LOCKSEG (MACRO)*
- REL* LINE 6: UNLOCKSEG (MACRO)*
- REL* LINE 21: OLAYPAGE (MODE)*
- REL* LINE 27: OLAYPAGETABLEHEAD (MODE)*
- REL* LINE 33: OLAYDATADESCN (MODE)*

PAGE 178, LINDA OLAYDATA
- REL* LINE 51: TRYDELETEOLAYDATA PAGE (PROC)*
- REL* LINE 189: ALLOCDATABASE (PROC)*
- REL* LINE 294: PUTGETDATABASE (PROC)*
- REL* LINE 342: LOCKDATABASE (PROC)*
- REL* LINE 402: INITOLAYDATATABLE (PROC)*

PAGE 179, LINDA OLAYDATA

PAGE 187, LINDA DISKBOOTRS
- REL* LINE 76: WRITEPROGRAMS (PROC)*

PAGE 189, ANN LOADER
- REL* LINE 4: BRFLLOADER (PROC)*

PAGE 223, ANN JC
- REL* LINE 60: JOBINTERPRETER (PROC)*

PAGE 234, ANN TEXTDECS
- REL* LINE 7: MAKETEXT (MACRO)*
- REL* LINE 15: STRING (MODE)*
- REL* LINE 16: TEXT (MODE)*
- REL* LINE 18: = (OP)*
- REL* LINE 21: LENG (OP)*
- REL* LINE 26: UNPACKTOPACK (OP)*
- REL* LINE 29: PACKTOUNPACK (OP)*

PAGE 235, ANN FILEDECS

PAGE 237, ANN MINIFILEDECS

PAGE 239, ANN FILESYSDECS

PAGE 240, ANN REALFILEDECS
- REL* LINE 7-10: INSEQ, OUTSEQ, INSCAT, OUTSCAT,
- REL* LINE 15: SELECT (ALL MACROS)*
- REL* LINE 30: SECTORADDRESS (MODE)*
- REL* LINE 41: REALSECTORADDRESS (MACRO)*
- REL* LINE 46: TRACKADDRESS (MODE)*
- REL* LINE 66: FILETYPE (MODE)*
- REL* LINE 67: FILESTATUS (MODE)*
- REL* LINE 68: FILEALLOCATION (MODE)*
- REL* LINE 69: FILEDURATION (MODE)*
- REL* LINE 73: FILEINFO (MODE)*
- REL* LINE 74: USERFILE (MODE)*
- REL* LINE 75: FILELEVEL (MODE)*
- REL* LINE 77: DISKADDRESS (MODE)*
- REL* LINE 78: FILECLASS (MODE)*
- REL* LINE 84: FILEHEAD (MODE)*
- REL* LINE 111: FILEBUFFER (MODE)*
TABLE OF CONTENTS FOR APPENDIX D - 294 -

PAGE 243, ANN.FILESYS
PAGE 244, ANN.FSROUTDEC50
PAGE 245, ANN.FSROUTDEC51
PAGE 246, ANN.FSROUTDEC52
PAGE 247, ANN.FSROUTDEC53
PAGE 248, ANN.HENFILESYS

- REL. LINE 114: FILE (MODE)
- REL. LINE 22: BASICFILE (DATA)
- REL. LINE 59: FILEIO (PROC)
- REL. LINE 100: OPENFILE (PROC)
- REL. LINE 333: CLOSEFILE (PROC)
- REL. LINE 612: ALLOCFILE (PROC)
- REL. LINE 747: DELETEFILE (PROC)
- REL. LINE 876: COPYFILE (PROC)
- REL. LINE 1103: INFODEF (PROC)
- REL. LINE 1188: CONTENTSFIL (PROC)

PAGE 268, ANN.FSROUT1

- REL. LINE 40: PRINTFILENAME (PROC)
- REL. LINE 57: NOODOTS (PROC)
- REL. LINE 80: SPLIT (PROC)
- REL. LINE 107: SEARCHNAME (PROC)
- REL. LINE 153: FILEOK (PROC)
- REL. LINE 177: FILESECTORADDRESS (PROC)
- REL. LINE 197: ALLOCFILEBUFFER (PROC)
- REL. LINE 271: MAKEFILEINFO (PROC)
- REL. LINE 309: INITFILE (PROC)
- REL. LINE 353: FILEOVERFLOW (PROC)
- REL. LINE 411: TRANSFERIOVALUE (PROC)
- REL. LINE 530: INITBASICFILE (PROC)

PAGE 278, ANN.FSROUT2

- REL. LINE 11: AVAILABLETRACKS (DATA)
- REL. LINE 14: TRACKACCESS (DATA)
- REL. LINE 31: TRANSFERTRACKE (PROC)
- REL. LINE 57: INPUTTRACKTABLE (PROC)
- REL. LINE 83: PRINTTRACKTABLE (PROC)
- REL. LINE 151: ACQUIRETRACK (PROC)
- REL. LINE 183: RELEASETRACK (PROC)
- REL. LINE 209: GETSECTORADDRESS (PROC)

PAGE 283, ANN.FSROUT3

- REL. LINE 29: LOCKFILE (PROC)
- REL. LINE 83: UNLOCKFILE (PROC)
- REL. LINE 125: LOCKPARENTALSO (PROC)
- REL. LINE 164: LOCKFILES (PROC)
- REL. LINE 184: ASSOCIATEBUFFERANDLOCK (PROC)
- REL. LINE 213: DEASSOCIATEBUFFERANDUNLOCK (PROC)
- REL. LINE 242: STANDARDFILE (PROC)
- REL. LINE 257: CLOSEALLFILES (PROC)

PAGE 289, ANN.TEXTPROCS

- REL. LINE 41: COMPARETEXTS (PROC)
- REL. LINE 71: COPYTEXTS (PROC)
- REL. LINE 93: PACKUNPACK (PROC)

EDITOR CODE:

PAGE 291, EDIT*EDITDEC5

- REL. LINE 31: NUMBERONLINE (MODE)
- REL. LINE 32: SPACEINFO (MODE)
- REL. LINE 33: LINE (MODE)

PAGE 292, EDIT*EXDEFINE

PAGE 293, EDIT*OLAYDATADEC5
<table>
<thead>
<tr>
<th>PAGE 294, EDIT•PACKDEFINES</th>
<th>295</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAGE 295, EDIT•PROCDEFINES</td>
<td></td>
</tr>
<tr>
<td>PAGE 296, EDIT•SMALLPROCS</td>
<td></td>
</tr>
<tr>
<td>- REL• LINE 31:</td>
<td>DIGIT (PROC)*</td>
</tr>
<tr>
<td>- REL• LINE 47:</td>
<td>COMPARERI ONGS (PROC)*</td>
</tr>
<tr>
<td>- REL• LINE 73:</td>
<td>LIMITCHECK (PROC)*</td>
</tr>
<tr>
<td>- REL• LINE 95:</td>
<td>STRIP (PROC)*</td>
</tr>
<tr>
<td>- REL• LINE 123:</td>
<td>BLANKFILL (PROC)*</td>
</tr>
<tr>
<td>- REL• LINE 143:</td>
<td>FINDCHARACTER (PROC)*</td>
</tr>
<tr>
<td>- REL• LINE 168:</td>
<td>GENERATELINE (PROC)*</td>
</tr>
<tr>
<td>- REL• LINE 204:</td>
<td>GENERATETEXT (PROC)*</td>
</tr>
<tr>
<td>- REL• LINE 231:</td>
<td>GIVENUMBERS (PROC)*</td>
</tr>
<tr>
<td>- REL• LINE 261:</td>
<td>ADJUSTBIGPARTS (PROC)*</td>
</tr>
<tr>
<td>- REL• LINE 284:</td>
<td>INTOLINE (PROC)*</td>
</tr>
<tr>
<td>- REL• LINE 311:</td>
<td>INSERTNEWLINE (PROC)*</td>
</tr>
<tr>
<td>- REL• LINE 336:</td>
<td>COMPARELINENOS (PROC)*</td>
</tr>
<tr>
<td>- REL• LINE 368:</td>
<td>WRITEPOS (PROC)*</td>
</tr>
</tbody>
</table>

| PAGE 303, EDIT•PACK        |
| - REL• LINE 28:            | STORE (PROC)* |
| - REL• LINE 91:            | UNPACK (PROC)* |

| PAGE 305, EDIT•POSITION    |
| - REL• LINE 347:            | POSITION (PROC)* |

| PAGE 312, EDIT•INSERT      |
| - REL• LINE 26:             | READ (PROC)* |
| - REL• LINE 100:            | INSERT (PROC)* |

| PAGE 315, EDIT•DELETE      |
| - REL• LINE 71:             | PRINTACTION (PROC)* |
| - REL• LINE 113:            | PRINT (PROC)* |
| - REL• LINE 244:            | DELETELINE (PROC)* |

| PAGE 320, EDIT•FIND        |
| - REL• LINE 31:             | FINDACTION (PROC)* |
| - REL• LINE 72:             | FIND (PROC)* |
| - REL• LINE 107:            | LOCATEITEM (PROC)* |

| PAGE 323, EDIT•MOVE        |
| - REL• LINE 34:             | SPLITLINE (PROC)* |
| - REL• LINE 64:             | RESEQUENCE (PROC)* |
| - REL• LINE 94:             | RENUMBER (PROC)* |
| - REL• LINE 165:            | MOVELINES (PROC)* |
| - REL• LINE 206:            | COPYLINES (PROC)* |

| PAGE 328, EDIT•EDITIO      |
| - REL• LINE 39:             | IMAGEIO (PROC)* |
| - REL• LINE 85:             | EDITIO (PROC)* |
| - REL• LINE 283:            | INPUTTEXT (PROC)* |
| - REL• LINE 301:            | OUTPUTTEXT (PROC)* |
| - REL• LINE 321:            | HELPUSER (PROC)* |

<p>| PAGE 334, EDIT•MAIN        |
| - REL• LINE 28:             | FIRSTLINE (DATA)* |
| - REL• LINE 28:             | LASTLINE (DATA)* |
| - REL• LINE 29:             | CURRENT (DATA)* |
| - REL• LINE 29:             | LASTCURRENT (DATA)* |
| - REL• LINE 30:             | FREELIST (DATA)* |
| - REL• LINE 31:             | NEWOBJECT (DATA)* |
| - REL• LINE 32:             | COLUMN (DATA)* |
| - REL• LINE 34:             | PRINTLINE (DATA)* |
| - REL• LINE 35:             | PRINTLASTLINE (DATA)* |
| - REL• LINE 36:             | WORKBUFFER (DATA)* |
| - REL• LINE 37:             | WORKBUFFINDEX (DATA)* |
| - REL• LINE 38:             | COMMANDLINE (DATA)* |
| - REL• LINE 39:             | INSERTBUFFER (DATA)* |
| - REL• LINE 40:             | READBUFFER (DATA)* |
| - REL• LINE 43:             | NAMETABLE (DATA)* |
| - REL• LINE 46:             | CURRENTLINEDESCR (DATA)* |</p>
<table>
<thead>
<tr>
<th>PAGE</th>
<th>SECTION</th>
<th>LINE</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>339</td>
<td>BRIT.SMRUNSYS</td>
<td>8</td>
<td>INRUNSYS (MACRO)*</td>
</tr>
<tr>
<td>340</td>
<td>BRIT.RSDECS</td>
<td>9</td>
<td>OUTRUNSYS (MACRO)*</td>
</tr>
<tr>
<td>342</td>
<td>BRIT.IODECS</td>
<td>87</td>
<td>STANDALONE (MACRO)*</td>
</tr>
<tr>
<td>343</td>
<td>EDIT.BIGMAIN</td>
<td>31</td>
<td>DICTIONARYPOOL (DATA)*</td>
</tr>
<tr>
<td>345</td>
<td>EDIT.TAKESPACE</td>
<td>24</td>
<td>TAKESPACE (PROC)*</td>
</tr>
<tr>
<td>347</td>
<td>EDIT.COMPILE1</td>
<td>9</td>
<td>GETOPTIONS (PROC)*</td>
</tr>
<tr>
<td>351</td>
<td>EDIT.PUTDISCBYTE</td>
<td>24</td>
<td>BRFILE (DATA)*</td>
</tr>
<tr>
<td>353</td>
<td>EDIT.IO1</td>
<td>12</td>
<td>GETCARDLINE (PROC)*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
<td>GETPAPERTAPELINE (PROC)*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18</td>
<td>PRINTEROUTPUT (PROC)*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>21</td>
<td>TTYOUTPUT (PROC)*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24</td>
<td>GETTELETYPEDLINE (PROC)*</td>
</tr>
<tr>
<td>354</td>
<td>EDIT.SAVEOUTPUT</td>
<td>18</td>
<td>SAVEINPUT (PROC)*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>27</td>
<td>SAVEOUTPUT (PROC)*</td>
</tr>
<tr>
<td>355</td>
<td>BRIT.DICTPAGEDECS</td>
<td>3</td>
<td>DICTPAGESIZE (MACRO)*</td>
</tr>
<tr>
<td>356</td>
<td>BRIT.DICTDECS</td>
<td>64</td>
<td>GLOBALDICT (DATA)*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>117</td>
<td>ESTABLISHDICTIONARY (PROC)*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>168</td>
<td>DICTIONARY (PROC)*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>238</td>
<td>DICTHIGHER (PROC)*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>249</td>
<td>DICTGENERATE (PROC)*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>298</td>
<td>RELEASEUPPERDICTPAGES (PROC)*</td>
</tr>
</tbody>
</table>