NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

The Core of Programming Systems

Edited by:

Guy Almes
Anita Jones
Bob Schwanke

February 2, 1976

Department of Computer Science
Carnegie-Mellon University

DEPARTMENT
of
COMPUTER SCIENCE

510.7808
C28r
76-8

Carnegie-Mellon University

The Core of Programming Systems

Edited by:

Guy Almes
Anita Jones
Bob Schwanke

February 2, 1976

Department of Computer Science
Carnegie-Mellon University

This document is divided into three parts. The Programming Systems Syllabus, prepared by Nico Habermann, Anita Jones, Mary Shaw, and Bill Wulf, with the assistance of the editors, defines the domain of the programming systems qualifier. The Programming Systems Study Guide, written by the editors, provides a guided tour of the syllabus, filling in the potholes. The Programming Systems Bibliography provides the references for both of the first two parts, with some annotations by Gerard Baudet.

TABLE OF CONTENTS

PROGRAMMING SYSTEMS SYLLABUS -- 1975

I. Data Structures
- A. Linear [Knuth 68, pp 234-304]
  1. Vectors and Arrays
     [Knuth 68, pp 295-302]
     [Hopgood 69, pp 12-15]
  2. Strings [Gries 71, pp 180-81] [Elson 75, Chap 6]
  3. Queues and Stacks [Knuth 68, pp 234-239]
  4. Representation [Knuth 68, pp 240-294]

- B. Non-Linear
  1. Trees [Knuth 68, pp 305-405]
  2. General list structures (Graphs) [Knuth 68, pp 423-434]
  3. Directed acyclic graphs
  4. Discrimination nets

- C. Accessing Techniques
  1. Associative schemes
     [Griswold 71, pp 118-120]
     [Feldman 69]
  2. Hashing functions
     [Maurer and Lewis 75]
     [Gries 71, pp 216-23]
     [Knuth 73, pp 506-549]
  3. Memo functions [Berliner 75]

- D. Records - unordered structures
  [Hoare 68]
  [Sites 72]
  [Hoare 72]
  [Dahl 68]
  [Elson 75, pp 21-27]

- E. Formal specifications
  [Liskov 75]
  [Parnas 71, 72a]

- F. Free Space Management
  [Elson 75, pp 163-181]
  [Knuth 68, pp 406-22]
  [Steele 75]

- G. Data Bases
  [Habermann 75b, Chap 9]
  [Madnick 69]
  [Lefkovitz 69]

II. PROGRAMMING LANGUAGES

A. By application area

1. Numeric/scientific

Algol [Naur 63] [Habermann 71] [Knuth 61,67]

Fortran

PL/I [Beech 70]

APL [Pakin 68]

Algol 68 [Lindsey 71,72]

PASCAL
[Wirth 71b, 75]
[Habermann 73]
[Lecarme 75]

Algol W [Sites 72]

BASIC

2. String

Snobol [Griswold 71] [Elson 75, Chap 7] [Pratt 75, Chap 15]

3. List Processing

L* [Robertson 75]
Lisp [Quam 73] [Weissman 67]

4. Simulation

Simula [Dahl 66]
Simula 67 [Dahl 72]

5. System Implementation

Bliss [Wulf 71]
L* [Robertson 75]

6. Abstraction languages

Alphard [Wulf 74a]
CLU [Schaffert 75]
EL1

7. Misc:

Cobol [Pratt 75, Chap 12]
RPG [IBM]

B. Issues

1. Type
   [Hoare 72, 73b]
   [Flon 74]
   [Liskov 74]
   [Wulf 74a]
   [Wirth 73, Chap 8]

2. Scope and Extent
   [Moses 70]
   [Elson 73]
   [Wulf 72, 74a]

3. Name Binding
   [Cosine 72, Mod 5]
   [Elson 73, Chap 5]

4. Control Constructs
   [Wulf 74a]
   [Knuth 66,74]
   [Dahl 72]
   [Dijkstra 68a]

5. Storage Management
   [Gries 71, Chap 8]
   [Knuth 68, pp 435-455]

6. Procedure mechanism
   [Wirth 73, Chap 12]
   [Gries 71, Chap 8]
   [Pratt 75, Chap 6]

7. Exception handling
   [Gries 71, Chap 14]
   [Beech 70]

8. Reference Variables [Hoare 75]

9. Compilation vs. Interpretation: consequences [Mitchell 70]

10. Concurrency
    [Dennis 66]
    [Campbell 74]
    [Hoare 74]

11. Protection
    [Wulf 74b]
    [Jones 73]

C. Methodology related to language [Wulf 76]
   1. Control mechanisms
      [Knuth 74]

   2. Abstraction mechanism
      [Flon 75]
      [Dahl 72]
      [Wulf 74a]
      CLU [Schaffert 75]

   3. Modularity [Parnas 71, 72a, 72b]

   4. Verification [Hoare 72]

III. Software Engineering [Guttag 75]

A. Motivation and Perspective
   [Weinburg 71]
   [Naur 69]
   [Buxton 70]
   [Goldberg 73a]

B. Proposed Methodologies
   1. Stepwise Refinement [Wirth 71a]
   2. Structured Programming (the original) [Dijkstra 72a]
   3. Modularity [Parnas 71, 72a, 72b]
   4. Hierarchical Design [Dijkstra 68b] [Parnas 74]
      [SRI 74] [Habermann 75a]
   5. Chief Programmer Team [Baker 72] [Brooks 75]
   6. Egoless Programming [Weinburg 71]

C. Specification [Parnas 72a] [Liskov 75]

D. Verification [Hoare 71] [Dijkstra 72a] [London 75]

E. Algorithmic analysis [Knuth 68, pp 94-103] [Aho 74]

F. Testing [Goodenough 75]

G. Software Construction Tools
   Editors
   Conversational Systems
   Job Control Languages
   [Weinburg 71]

IV. TRANSLATORS

A. Introduction
   1. Compilers <u>vs.</u> Interpreters [Gries 71, pp 2-10] [Hansen 74]
   2. Assemblers [Barron 69]
   3. Loaders [Presser 72]

B. Components [Gries 71] [Hopgood 69]
   1. lexical analysis [Gries 71, Chap 3]
   2. syntax analysis [Gries 71, Chaps 4-7, 12-13]
   3. symbol and name tables [Gries 71, Chaps 9 & 10]
      a. data structures
      b. search algorithms
      c. insertion/deletion [Knuth 73b]
      d. interaction with language properties
   4. code generation [Gries 71, Chap 17] [Wulf 75b]
   5. macros and their processing [Gries 71, Chap 19] [Wegner 68]

C. Global optimization, flow analysis, and register allocation
   [Wulf 75b]
   [Gries 71]
   [Johnsson 73]

D. Runtime issues [Randell and Russell 64]
   1. Display and stack management [Gries 71, Chap 8]
   2. Dynamic storage allocation & garbage collection
      [Knuth 68, pp 435-455]
   3. Overlay issues
   4. Exception recovery[Gries 71, pp 314-320]
   5. Debugging
   6. I/O and system interaction
   7. recursion
   8. Co-routines

E. Compiler-compilers [Feldman 68]

V. Operating Systems
  A. Concurrency and Synchronization
    1. Concept of Multiprogramming and context swaps
       [Dennis 66] [Saltzer 66]

    2. Implementation of Multiprogramming
       Interrupts [Digital 71, pp 117-120]
       Semaphores [Liskov 71]
       Procedure Calls [Organick 73, pp 31-32]
       Messages [Jensen 75]

    3. Mutual Exclusion
       MULTICS Block/Wait [Lampson 68] [Saltzer 66]
       Semaphores P/V [Dijkstra 68b]
       Up/Down [Wodon 72]
       Critical Regions [Brinch Hansen 72]

    4. Message Systems
       RC4000 [Brinch Hansen 70]
       As a basic primitive [Jensen 75]
       Hydra [Cohen 75, Section 8]

    5. Monitors [Hoare 73b,74]

    6. Path Expressions [Campbell 74]

    7. Deadlocks: concept, prevention, avoidance, recovery
       [Habermann 69] [Holt 72]

    8. Classical Problems [Courtois 71]

  B. Addressing
    1. Segmentation Concepts
       Survey [Randell 68]
       MULTICS [Dennis 65]

    2. Capability Concepts
       Codeword [Iliffe 62,68]
       Descriptor [Organick 71] [Organick 73]
       Capability [Fabry 74]

    3. Multiprogramming Problems [Needham 72]

C. Memory Management [Habermann 75b, Chap 7]
   1. Relocation/Overlaying [Lanzano 69] [Sayre 69] [Watson 70, pp 40-45]

   2. Paging [Denning 70] [Watson 70, pp 45-52] [Naur 65] [Kilburn 62]

   3. General Memory Hierarchies [Mattison 70]

D. Resource Allocation and Scheduling
   1. Processor Scheduling
      Short term [Lampson 68] [Brinch Hansen 71] [Kleinrock 70]
      Long term [Habermann 75b, Chap 6]

   2. General Allocation [Habermann 75b, Chap 7]

   3. Deadlocks [Habermann 69] [Holt 72]

E. Protection Concepts [Weinstock 73]
   1. Implementation of Domains,
      Privileged Instructions and Address Space Restriction
      [Graham 72] [Spier 73]

   2. Authority Based Protection
      MULTICS [Graham 68] [Daley 65]

   3. Capability Based Protection [Lampson 71] [Fabry 73] [Jones 73] [Jones 74]

   4. Classical Problems
      Confinement [Lampson 73]
      Mutually Suspicious Subsystems [Schroeder 72]

F. Examples of Real Systems
   (Learn some of these)
   1. THE [Dijkstra 68a]
   2. RC4000 [Brinch Hansen 70]
   3. Kernel/Domain Experiment [Spier 73]
   4. VENUS [Liskov 71]
   5. Hydra [Cohen 74]
   6. Cambridge Monitor [Meyer 70]
   7. UNIX [Ritchie 74]
   8. MULTICS [Corbato 73]

1. Data Structures

1.1 Introduction

Data structures play several roles in programming systems. First, they are major building blocks from which systems are built. Second, controlling the manipulation of data structures is a major theme in the development of structured programs and type-oriented languages. Third, most programming systems must handle enormous quantities of data; therefore the structure of the data largely determines the performance for the program. In this section we will deal with the basic anatomy, manipulation, and performance of some common data structures. We will not cover the full debate surrounding data structure manipulation until we discuss types and abstraction in programming languages, nor will we attempt quantitative measures of performance.

Every data structure has two aspects: its specification and its implementation. The two can and should be completely separable. For example, consider a structure S, a value x, and two operations a(S,x) and b(S). Operation b returns a value of the same kind as x. Suppose we assert that

$$(\forall x)(\ [\ a(S,x)\ ;\ c \leftarrow b(S)]\ \rightarrow\ c \text{ equals } x\ )$$

and, further, suppose we define a recursive sequence d:

$$d(S,x) = [\ a(S,x)\ [\ ;\ d(S,y)\ ]^*\ ;\ b(S)\ ]$$

and assert that

$$(\forall x)(\ x \text{ equals } d(S,x)\ )$$

where * denotes zero or more occurrences of the bracketed sequence, and the value of a sequence is the value of the last expression in the sequence. Then we can see that the net effect of d on the structure S is null, with respect to the operations a and b.

Now consider a vector V and a pointer P into it. The operations

$$e(V,x) = [\ V[P] \leftarrow x\ ;\ p \leftarrow p+1\ ]$$
$$f(V) = [\ p \leftarrow p-1\ ;\ \text{return } V[p]\ ]$$
$$g(V,x) = [\ e(V,x)\ [\ ;\ g(V,y)\ ]^*\ ;\ f(V)\ ]$$

quickly reveal that g leaves V undisturbed relative to e and f. But now we see that S, a, and b describe a stack, and V, e, and f implement one. Another implementation might use a linked list instead of a vector. For a more complete treatment of this exercise see [Parnas 72a].

Note the distinction between the <u>operational</u> <u>specification</u> of a structure and its <u>implementation</u>. A stack is a stack is a stack, whether it's implemented as a vector and pointer, a linked list, or a threaded tree.

Most common data structures are thoroughly described in the literature; perhaps the following map will help.

1.2 Linear structures

These include: stacks, queues, vectors, and strings. Multi-dimensional arrays are also included here because they can be mapped in a straightforward way onto simple vectors. Knuth's coverage of linear structures is nearly comprehensive [Knuth 68, pp 234-304]; one topic he doesn't discuss is dope vector, or Iliffe vector, representation of arrays, found in [Hopgood 69, pp 12-15]. Strings make their claim to prominence by (a) being necessary for I/O and useful in compilers, business data processing, and user interfaces to most large systems, and (b) being the most common domain for the practice of pattern matching, a powerful tool. String representation depends heavily on the operations it must support. If sorting is all that's involved, fixed length vectors of characters are sufficient. But if the representation must support frequent insertion and deletion, linked lists would be more appropriate. [Elson 75, Chaps 6 and 7] give a fine survey of the area, and describe SNOBOL, the best known string processing language today.

1.3 Non-linear structures

[Knuth 68, pp 305-405] treats trees and their relatives exhaustively and well. Binary trees are important because the algorithms that manipulate them are simple, and well suited for binary computers. Lists (wherein an element of a list can itself be a list) are a convenient linear representation of trees[Weismann 67, pp 5-24]. Knuth shows that binary trees can easily be used to implement any other kind of tree, as well as lists, and gives several accessing algorithms and representations.

Trees happen to be a special case of the class of structures called graphs. A graph is a set of nodes with connections between some of them. A connected graph is a graph where one can follow an unbroken sequence of connections from any node to any other node. A directed graph is one where the connections are asymmetric (usually represented on paper as single headed arrows, and on computers as pointers). Acyclic graphs are ones for which there is no sequence of connections which (a) includes no connection more than once, and (b) leads from a node to that same node. Directed acyclic graphs (dags) are the same, with the added condition that (c) the sequence follows the directionality of the connections. An acyclic graph is isomorphic to a free tree. A dag having not more than one path from one node to another is isomorphic to an oriented tree [Knuth 68, pp 362-380] [Elson 75, pp 108-117].

1.4 Accessing Techniques

Having acquired a collection of alternative representations for one's data, the programmer must choose among them. One usually finds that the most compact representations are fairly expensive to access, and that those forms most readily accessed take up an inordinate amount of space. The overall quantity of data involved also affects the choice of representation. For instance, finding a given item in a list of ten items is most easily done by a linear scan of the list. Finding one item in a list of 100,000 is a task of considerably more complexity. If time is at all important, the programmer might be willing to use twice the disk space if doing so would permit a much faster accessing algorithm. Hashing functions were developed to locate items in long lists based on standardized keys. Memo functions arose to lessen the effort required to traverse large trees.

1.4.1 Hashing functions

Table maintenance is an area where machines have inspired a truly new way of doing something that men thought they knew how to do.. Tables which will be searched often must be ordered in a way that lets any item in it be found efficiently. For small tables, any order will do, since one can afford to examine every element. For moderate sized tables, if the table is totally ordered (e.g. alphabetically) reasonable algorithms exist (e.g. binary search) for finding any element. But for large tables, both sorting and searching are too time consuming to be done frequently. A hashing function is a function which tells you where an item is, or belongs, in a table, by some simple, but not necessarily obvious, computation. It is a mapping from a (large) set of possible table element tags into a (small) set of slot locations. Being a many-to-one mapping, it is usually accompanied by a secondary function telling where to look next if the slot named by the hashing function contains a different element. [Knuth 73, pp 506-549] details a variety of search techniques and hashing functions, and gives relevant performance figures.

1.4.2 Memo functions

Trees turn out to be such useful tools that program designers have often been willing to traverse very large trees to avoid complicating their programs. A <u>memo function</u> is a tag on a subtree, or a way of recognizing subtrees, which permits one to avoid traversing that subtree. For instance, chess move trees have many congruent subtrees, because the same board position can be reached by several different move sequences. Recently, Hans Berliner has found a way to use memo functions to help search move trees. He has designed a system which, having evaluated a move and found it wanting, catalogues the move and the relevant context of it. Then as the tree search continues, when the move comes up again in a similar context, it can be rejected without reevaluation. Optimizing compilers, such as the Bliss/11 compiler [Wulf 75], treat common subexpressions in a similar way. The code generation routine finds tags on some subtrees saying "my code has already been generated; skip me".

1.5 Records

So far, all the structures discussed have been homogeneous, i.e. all nodes in a given structure had the same format. But in order to permit understandable programs, many applications require combining different kinds of information (i.e. different fields have different formats) into a single node such that nodes with different formats can be handled in a uniform way. A data base for keeping track of vehicles for United airlines might have different sizes of nodes for baggage carts, tank trucks, and DC-10's, but each of these nodes would have a field for the manufacturer and a field for its next scheduled maintenance. A record is a node whose fields are accessed by name rather than by index. [Hoare 68] is one excellent description of them; Algol W [Sites 72] has one clean implementation. Also read [Gries 71, pp 182-187].

1.6 Formal Specifications

When creating new, sophisticated data structures out of old simple ones, one often loses his grasp of how the structure behaves, especially when more than one program(mer) uses it. This has sparked a drive by some to hide all data structures inside precisely defined procedures. David Parnas was a pioneer in this work [Parnas 71, 72a]. He developed a number of principles for module design and specification, but was frustrated by the absence of languages which would enforce his boundaries. No completely satisfactory languages have been implemented yet, but several (cf. Clu, Alphard) are being designed to both enforce the boundaries and to permit formal specifications to be integrated into the program text. [Liskov 75] gives some criteria for selecting specification languages.

1.7 Free Space Management

When using a data structure which creates and deletes nodes in an unpredictable fashion, we need a facility for allocating in which to put nodes, nodes in, and reclaiming abandoned nodes for re-use. Such allocation schemes fall into two categories: (1) Reference counts, maintained for each node, reflect the number of pointers to that node. When count falls to zero the space occupied by the node is reclaimed. (2) Garbage collection: when free storage runs low, the system traces down every node in use and marks it. Any nodes remaining unmarked are recycled. Read [Elson 75, pp 163-181] for a fuller exposition.

1.8 Data Bases

A growing class of data structures, the so-called 'data bases', are characterized by very large size and by a permanence that often exceeds that of any programs that access them. When approaching the design of these large quasi-permanent data structures, care must be taken to use robust structures that can support the added weight. To borrow an example from comparative zoology, consider two similar animals that differ in linear dimensions by a factor of two. Since its weight varies as the cube of the size and the strength of its bones only with the square of their diameter, the diameter of the bones of the larger animal must more than double that of the smaller. So also the structures used in building large data bases must be carefully chosen, lest the data base collapse. Two families of differences are given here.

Due to very large size, a data base can seldom be kept within primary memory. Representation and accessing techniques must take the nature of the memory hierarchy available into account to minimize accessing delays such as the seek time on a moving-head disc. The performance of searching and updating depends heavily on how well the physical layout of data reflects the pattern of accesses. Also, since many searching techniques become much slower as the data base grows [Knuth 73], redundant structuring may be used to expedite anticipated searches.

Due to very long duration, several problems must be faced. First, as data nodes are inserted and deleted, the physical layout designed to expedite accessing may be compromised. This causes slower accessing and greater storage fragmentation. These problems necessitate either periodic restructuring of the data base or sophisticated updating routines that dynamically restructure the data. Second, the permanence of the data base increases the chance of error in the data or in the structure. These structures therefore motivate serious use of backups, checksums, and other reliability measures. [Lefkovitz 69] and [Madnick 69] are suggested references.

2. Programming Languages

2.1 Introduction

The syllabus approaches languages from three orthogonal directions: application area, intrinsic issues of language features, and the effect of language on program design. The first of these needs no explanation -- it's there for reference. To cover the second we shall attempt a short definition of each issue. To introduce you to the third, read [Wulf 76].

2.2 Data Accessing Issues

Type is the current buzzword for a language construct which only permits accessing a data structure by employing the operations defined on it. The leading languages based on types are Alphard [Wulf 74a] and Clu [Schaffert 75]. For a once-over, thorough theoretical treatment of types, read [Hoare 72]. To follow the recent developments, start with Flon's survey [Flon 74] and use its references.

Scope, extent, name binding, memory management, procedure mechanism, and reference variables are all issues which interact strongly with type implementations. Name binding is the process by which an occurrence of an identifier in a program becomes associated with a storage location. Scope rules govern these name bindings. Algol, for instance, has static scope rules: a name is bound to the declaration in the block where the name is used. If no declaration is found in that block, the next surrounding block is checked, and the next, and the next, until a declaration for the name is found. In contrast to Algol, APL's scope rules are dynamic: just as new storage is allocated each time a routine with declarations is entered, so each name is bound to the storage most recently (in time) allocated under that name. The APL style is very hard to follow; the Algol style is insufficient, and still prone to certain kinds of errors. Extent refers to the period of time over which the value of a variable persists: the extent of Fortran variables is forever; the extent of the Algol local variable is the period when the block immediately surrounding the declaration is being executed. Note that scope of an Algol own variable is the block surrounding it, but its extent is forever.

Memory management is an efficiency↔flexibility tradeoff. FORTRAN typifies a group of languages which allocate memory at load time, once and for all. This limits the amount of logically distinct data the program can use (excepting bulk storage). Algol-like languages have some such static allocation, but also allocate local storage on a "last in(to use), first out (of use)" basis. This lets the free and in-use space be separated by a single pointer. Languages like LISP and SNOBOL, which cannot limit themselves to such a LIFO discipline, must use garbage collection schemes, which, though well understood and very flexible, are expensive in time and memory.

Reference variables are variables which contain pointers to other variables. They wreak havoc with conventional scope and extent rules. If a routine with access to an own variable puts a reference to it in a global reference variable, the privacy of the own is violated. A local referred to by a reference variable will either leak private information or invalidate the contents of the reference variable when the local disappears.

```
begin
   reference R;
   real leak;
   begin
              own A;
              ...
              R ← reference to A;
              ...
   end
   ...
   leak ← value of object referred to by R;
end
```

2.3 Procedure Mechanisms

Procedure mechanisms vary among languages in a variety of ways:

1) What constitutes a procedure? Most languages have fixed beginning and ending points for procedures. But BASIC and SNOBOL have procedure calls which, like assembly language instructions, can "jump to subroutine" to any statement, and have returns which simply happen wherever they are encountered in normal execution. Cobol, furthermore, lets the procedure call, itself, define what group of statements constitutes the procedure. In most languages, a procedure call leaves no context behind when it returns, except own values and the values it returns. But co-routines, such as those available in Bliss and Simula, allow a procedure to be stopped and restarted at various entry points during its lifetime.

2) How do scope and extent rules apply to procedures? In static allocation languages we generally find no problems with names and procedures, except in relation to parameters. Algol's static scope rules apply to procedures in the same way they do to blocks. A Bliss routine inherits global and own structures, but not local ones, from its static context. APL and Lisp routines have access to the structure most recently bound to each name in the universe.

3) How are parameters passed? Passing them by value means evaluating each one once, giving the value to the routine, and never again referring to them in the caller's context. Call by reference implies a once only computation of the location of the structure, which is then used to hold the corresponding formal parameter. Call by name requires that the parameter location and value be recomputed, in the caller's context (in contextu vocatoris), each time the formal parameter is used inside the procedure. The overhead is very high. Call by result evaluates the parameter location, executes the procedure body, then

copies the formal parameter value into the actual parameter's location. Call by value-result copies the value out of the actual parameter into the formal one, does the computation, then copies the formal back to the actual.

2.4 Control Mechanisms

When Dijkstra first asserted that the goto statement was harmful to program understandability, he unleashed a torrent of creativity which produced new control constructs to replace it, and a host of critics to reject the new constructs. Even good programmers, when writing in a language with no other constructs for control but goto's and do loops, too often find themselves writing unreadable programs. Most of the things done by goto's can be done more easily and readably using do while statements, if then else clauses, and other iterative statements. One common use for goto's has been to break out of the orderly flow of control when some unusual condition occurs. To satisfy this need, Bliss offers four constructs which let the execution of a complex statement (in Bliss, an expression) terminate, whereupon execution resumes at the next statement.

Exceptions are unusual events which must be handled by unusual mechanisms. Some such mechanisms are monitors, which let the user provide his own response to events that the system can monitor continuously [Beech 70] (e.g. writes to certain memory locations, error flags). Others are facilities which let the programmer prematurely terminate processing in a dynamically defined context. Be sure to look at Bliss's signal and enable constructs. The enable statement says, "I know something about these special events. As long as I'm on the call stack, I'm available to process them." The signal statement says, "A special event has just occurred. Return control to an enable block which knows something about it, popping all intervening subroutine calls."

2.5 Concurrency and Protection

These two topics have recently found their way into programming languages, primarily because of the overall trend to write bigger systems with complex scheduling problems and multiple, fallible authors. For an introduction to these areas, see the discussion under the same headings in the Operating Systems section.

2.6 Compilation vs. Interpretation: consequences

Many language features require non-trivial bookkeeping during execution (e.g. array bounds-checking, stack management, storage allocation). These features can often be more easily implemented by an interpreter than by a compiler. More on this in section 4.1

3. Software Engineering

3.1 Introduction

Wulf's discussion [Wulf 76] of the software crisis, though oriented toward language design, also provides a good introduction to program design. The field has finally come to recognize that large programs

a) are too complex to be completely understood by one mind at one time,

b) will be modified by people who didn't design them, as often as they are used, and

c) can never be proved to be free of bugs by execution.

When the government lets a contract for software, both the government and the vendor know that the product will take longer and cost more than specified in the contract. The exceptions to this statement can be counted on your thumbs.

All this has come about because the size of the programs we want to build has exceeded the capacity of our minds to understand them. In order to master this complexity, we need tools which give the programs a framework which aids understanding. Several disciplines have arisen to meet this problem:

1) a collection of principles for good programming,

2) methods for specifying the behavior of programs,

3) techniques for proving things about programs (e.g. that they meet their behavioral specs), and

4) criteria for measuring the performance of the system produced.

3.2 Proposed Methodologies

Many of the emerging principles have properties in common. Stepwise refinement is a method that says:

a) write out the complete algorithm in a short sequence of steps you understand.

b) take up each step in succession and expand it by the same rule.

c) keep refining the steps until they can be written understandably in a conventional programming language.

This method has two attributes that recommend it. First, it permits the designer to see the effects his decisions have on program structure, and uncovers issues he might not have considered. ([Dijkstra 72a] is required reading, an elegant example of this.) Second, many of these decisions will only affect one or two of the steps; in such cases the designer can then isolate the implementation of the decision from the steps it doesn't affect. This second property conveniently introduces modularity. Programs get modified because either bugs are discovered or design decisions are changed. If a decision is changed, years after the program is written, the person who changes the program must track down every piece of code which depends on the decision. If the decision has to be made early in the initial design (when the resolution of the program is still very coarse), every part of the program potentially relies on that decision. For example, the RSX11D Fortran compiler (1974) uses PDP-11 TRAP instructions to signal all exceptional conditions. This decision must have been well known throughout the system, for there were TRAP instructions in about ten of the eighty source files. When one of the authors (R.S.) transported the compiler to Hydra, an operating system which does not support user TRAPs, he had to search all eighty files to get the TRAPs out.

On the other hand, if the designer can decompose his problem such that each design decision is important to at most two or three "steps", he greatly simplifies the structure of his program, and consequently its debugging and modification. Much of the current research in language design seeks convenient tools to enforce the boundaries of these program modules.

Hierarchical Design denotes another principle which promotes structure in programs: that the dependency relationships between modules must form a partial ordering. Not only does this make it possible to debug one module at a time, but it also permits the lower modules to be finished before the higher ones are designed, and eases replacement of old modules with redesigned ones. The Family of Operating Systems project at CMU is an attempt for one group to design a large collection of operating systems in a short period of time, by defining their modules cleanly enough that is simple to, say, replace the BATCH module with a TIMESHARING module [Habermann 75a].

The Chief Programmer Team is a management concept which states that the best way to produce software is to hire one really good programmer to do the actual programming, and support him with whatever specialists and assistants (e.g. man-machine interface expert, program librarian, secretary) he needs. Only one major system produced this way has met its deadline and budget line: the New York Times Morgue system, chiefed by Harlan Mills [Baker 72].

Egoless Programming is a reaction to the programmer who views his code as his own private domain, not subject to viewing, much less critique, by others. The egoless programmer must explain his code to at least one of his co-workers, on the premise that good structure and readability go hand in hand. Both the structure and the actual explanation will help unearth programming and design errors [Weinburg 71].

3.3 Program Specification

Program specification is the complement of data structure specification: we specify the behavior of programs in terms of how they affect variables. We still need a precise language for expressing these changes; [Liskov 75] provides criteria for evaluating language proposals, with emphasis on verifiability.

3.4 Verification

[London 75] provides a good description of the state of this art. Verification now ordinarily means an attempt to demonstrate consistency between the specification and the implementation of a program. <u>Inductive</u> <u>assertion</u> is the most widely used technique. It consists of inserting in the program assertions about program variables, and verifying them, then stringing them together into a proof of some property of the whole program. Also see Formal Specifications, under Data Structures.

3.5 Program Analysis

The performance of a program can be discerned in two ways: by analysis, as Knuth does with almost every algorithm he presents, and by testing [Goodenough 75]. Analysis of algorithms [Aho 74] is a blossoming field. Its general goal is to predict the execution time and/or the space required by a program, as a function of the amount of data it is manipulating. It gives little attention to constant factors and terms in the function, since these are quickly overshadowed by exponential and high order polynomial terms.

4. Translators

4.1 Introduction

Compilers, interpreters, assemblers, linkers, and loaders are all programming systems which help translate an abstract program into a sequence of machine operations which implement it. They fall into the core of Programming Systems because

1) they implement programs,

2) they are ubiquitous examples of large programming systems, with all the concomitant problems, and

3) some day, you too may get to write one.

Too often we draw a sharp line between compilers and interpreters, saying that compiled code is a list of machine instructions, whereas code to be interpreted is a data structure closely resembling the source program, to be fed into a hardware-software system which carries out the programmer's instructions. Two examples serve to blur the distinction:

1) Most Fortran systems, no matter how highly optimized they are, have an interpreter to handle I/O. The format statement is the program to be interpreted, the keyword of the I/O statement (e.g. read, print) directs the source and destination hookups, and the list of variables is the data the program acts on.

2) The early Fortran compilers for PDP-11 operating systems produced what is termed "threaded code". It consists of a list of procedure names, each one followed by its parameters. Each procedure called implements a Fortran construct, using the return address as a pointer to its parameters. Furthermore, each routine knows that the word following its last parameter is the address of the next routine. Therefore, instead of doing a return from subroutine when its finished, it branches to the beginning of the next routine, and advances the parameter pointer, all in one machine instruction. The reader may decide for himself whether this code is compiled or interpreted.

So, instead of trying to classify a translator as either a compiler or an interpreter, we can speak of the extent to which it compiles the code.

The choice between compilation and interpretation, even within the same system, must be made on the basis of fixed versus variable costs. The cost of a translator consists of (1) a cost "fixed" by the length of the source text, and (2) a cost that varies with the number of times each statement will be executed. Compilation has a high fixed cost and a relatively low variable cost; interpretation costs less prior to execution, but more during execution.

To see that one really can get the best of both worlds, read [Hansen 74], which describes a Fortran system which interprets the program at first, then incrementally compiles and optimizes the heavily used parts, automatically. It was superior, in CPU time used, to every other Fortran compiler to which Hansen compared it. Such selective compilation has been done manually for years, in interactive systems like Lisp and L*, which let the user request that pieces of his system be compiled. Note that interpretation is primarily used on code that will not be executed often. This category includes any code which isn't fully debugged yet, and includes partially-built programming systems. Most interactive systems are designed to permit the programmer to stop the program at certain points, change or augment the program, and continue where he left off. Thus interactive, interpretive translators are strongly conducive to program development.

Assemblers, once considered an art unto themselves, have become somewhat hard to distinguish from compilers. Their basic function is translation from assembly language to machine language, which in most cases is almost a one-to-one mapping. Macro facilities used to be available only in assemblers; now many compilers have them as well. If you want to read more about assemblers, [Barron 69] is reasonably complete. Similarly, a good way to learn about linkers and loaders is to read [Presser 72]. Briefly, a linker takes a set of machine language modules as input, implements intermodule references, (e.g. global names, common blocks), and puts everything into a format the loader can read. A loader is a program which makes a linked collection of binary modules executable. For a bare machine, this just means writing them into core. For a page-oriented system like HYDRA, it means taking a list of binary page objects and putting together the various data structures to keep track of them during execution. Linking and loading are often done by the same program.

4.2 Components

Different translators do a lot of the same kinds of things while achieving different goals. Lexical analysis (converting a string of characters into a string of language symbols) is done by all translators except loaders. All translators have symbol tables. And so on. Read [Gries 71] to learn about these common parts, then skim [Hopgood 69] to fill in the gaps. In order to study translators one must start with an understanding of languages. [Gries 71, Chap 2] gives an excellent foundation here. The translator's job can then be defined to be to parse the program, derive its intent from its syntactic structure, and then generate a program for a target machine, which carries out the intent of the original program.

4.2.1 Parsing

Parsing techniques are usually classified as either top-down or bottom-up. One parses top-down by starting with the distinguished symbol of the grammar and applying productions of the grammar to it, until he finds a sequence of productions which yield the input string. Bottom-up parsing involves searching for a set of (productions, except reversed, so they're) reductions which reduce the input string to the distinguished symbol. Read [Gries 71] selectively, learning to do these to methods by hand, and learning how LR[K] grammars aid bottom-up parsing, and how recursive descent implements top-down parsing.

Lexical analysis is the simplest, yet most time consuming, part of parsing. It is actually only a sub-part of syntax analysis, but it is handling a portion of the grammar of the language which is very regular and very easy to process. That is, while the grammar for the entire language may be phrase-structured, the set of productions which actually have terminal symbols on the right hand side may form a regular grammar, which is known to be parseable by a finite state automaton [Gries 71, Chap 3].

4.2.2 Symbol Tables

Compilers and interpreters must have a mechanism for storing and retrieving information about the identifiers the program uses, such as type of variable, location in core, dimension of array. This mechanism is called a symbol table. The accessing mechanism for this table must allow intermingled stores and retrievals, making it a potential bottleneck in the compiler. Be sure to study the section on data structure accessing mechanisms (I.C in the syllabus). Then study [Gries 71, Chap 9 & 10] to see how symbol tables are organized and used. The complexity of the symbol table varies with the intelligence of the compiler as well: [Wulf 75] describes the optimizing Bliss/11 compiler, whose symbol table is an elaborate general list structure.

4.2.3 Code Generation

Code generation is the phase of translation which starts to depend strongly on the particular idiosyncracies of the target machine. Read [Gries 71, Chap 17] to see the general methods, then look in [Wulf 75] to see a good example.

4.2.4 Macros

Macro processing [Wegner 68] permeates translation systems, yet little has been written about it. One should be aware of what sorts of macro processors are available, lest he be satisfied with the first one he uses. A macro facility allows the programmer to associate a name with a string of characters. Then every place in the text where that name occurs, the macro processor replaces the name with the string. Most systems allow the macro to have parameters. Each actual parameter is inserted in the replacement string wherever the formal parameter occurs. If the actual parameter is itself the name of a macro, there is an issue of which to expand first.

Some systems allow recursive macros. To do this they must have a way of deciding that part of the replacement string will never be executed, so that it need not be expanded. Otherwise a macro which called itself would expand forever.

A few macro processors allow a variable number of parameters on the macro. This turns out to be a shorthand way of representing a sequence of calls to the same macro with different parameters. Bliss/11 has a very elaborate macro facility, exhibiting most of the above features.

4.3 Optimization

The biggest argument given for assembly language programming is that the code is more compact and more efficient. This claim has begun to crumble at CMU in the face of the Bliss/11 compiler, which regularly produces shorter programs than assembly language programmers attempting the same task. The current shortcoming in this compiler, however, is its inability to optimize the innermost loops of programs, thus its shorter programs still run more slowly than their longer, hand coded counterparts. Read [Gries 71, Chap 18] and skim [Wulf 75] to discover the wide variety of optimization techniques available. One important subtopic is register allocation on multi-register machines, which can dramatically affect performance [Johnsson 73].

4.4 Runtime Issues

To acquire a feeling for where your program is spending its time, you need to look at the insides of a runtime system for a compiler for a common language like Algol [Randell 64]. [Gries 71] will also give some feel for the problems involved (see specific references in syllabus), but remains theoretic because it never shows a whole compiler system.

4.5 Compiler-Compilers

There are well known algorithms for generating a syntax analyzer for a language, given a precise definition of its grammar. Programs which do this are called compiler-compilers. The next breakthrough will be the discovery of both a precise notation for completely describing computers, and an algorithm which uses that description to generate optimizing code generators. Until then, compiler-compilers will only be useful in the early stages of language development, because the compilers they produce are not transportable, and produce very poor code [Feldman 68].

5. Operating Systems

5.1 Introduction

Modern programming systems generally find it essential to map a physical conventional computer onto a higher level machine and to present this machine to programmers. One reason for this is to allow the physical resources of the hardware fairly and safely to be shared among many processes. This motivation is technological and would disappear if many copies of very large computers were available. A second reason is to implement a programming environment even more attractive than a private large computer, an environment that allows information to be shared among processes. This motivation is intrinsic and is steadily increasing as the sophistication of application routines grows. This section discusses several areas in which operating systems have traditionally performed this mapping; many of the techniques discussed are being used now by applications-oriented software as well as by operating software. It concludes with a list of several examples of actual operating systems, for many systems are noteworthy, not in their solution of any particular problem, but in the approach taken in system design. The reader is then referred to the COSINE Report [COSINE 72] for further study.

5.2 Concurrency and Synchronization

Concurrent execution of processes within a computer system opens the door both to increased performance and also to several interesting problems. In a broad sense, concurrency includes overlap of i/o with computation and time-sharing of a CPU among several independent jobs, as well as the cooperation of computing processes in a single task. Thus concurrency is more the rule than the exception, and the real challenge becomes to find an example of a truly sequential process! As multiprocessing becomes more common, however, and explicit parallelism within a single applications task becomes more attractive, a solid grasp of the concepts, problems, and techniques associated with concurrency becomes essential for all computer scientists, not only for operating systems specialists.

One early effort to understand concurrency took place during the early '60's at MIT's Project MAC [Dennis 66, Saltzer 66]. In developing a time-sharing system on a 7094 and in their early design work on the more ambitious MULTICS system, they developed the techniques of 'block', 'wake', and 'context swap' essential to multiprogramming, but not without also inventing deadlock bugs [Rappaport 68].

At about the same time, Dijkstra and others began a study of the problem of 'mutual exclusion'. This problem arises whenever two or more concurrent processes share modifiable data. Each process may need to perform computations on the shared data that depend on the data not being modified by the other processes during the computation. Such a computation must appear indivisible relative to the other processes. Early contributions by Dekker [Dijkstra 68c, p 58], Dijkstra [Dijkstra 65], and Knuth [Knuth 66] presumed only the load and store memory operations. Although ingenious, these solutions were very difficult to construct, prove, or even understand; in Dijkstra's words [Dijkstra 68c, p 66], they were "tremendous mystification[s]".

Out of dissatisfaction with the early solutions, the notion of a 'semaphore' was developed [Dijkstra 68c; read it!]. The semaphore concept was a crucial step in the development of a classical school of concurrency. While solving the mutual exclusion problem handily, it also afforded natural solution of the 'producer-consumer' problem and others. [Habermann 71b] exemplifies work within this school to strengthen mathematical understanding of synchronization; this work is related to program verification. [Brinch Hansen 72] similarly exemplifies work in strengthening the programming and linguistic tools in this area.

Several other distinct synchronization concepts have also been proposed. In the design of the Danish RC4000 system [Brinch Hansen 70], the concept of a 'message system' was adopted to accomplish both interprocess communication and synchronization, thus making the use of shared memory and semaphores unnecessary. The related concept of a 'pipe' in the UNIX system [Ritchie 73] unifies the notions of interprocess communication and i/o. A second concept, due to Brinch Hansen and Hoare, is that of a 'monitor' [Hoare 73, 74]. As a Parnas module or a Simula class, a monitor consists of a set of procedures with exclusive access to a data structure. The monitor concept, however, also facilitates proper synchronization of processes calling these procedures. A recent concept due to Habermann and Campbell [Campbell 74] gives the programmer a powerful tool for specifying the synchronization of processes by 'path expressions'. All these concepts focus on making synchronization less error-prone and more programmable.

One of the more persistent problems with concurrency is that of 'deadlock' or 'deadly embrace' [Habermann 69, Holt 72]. A deadlock is said to occur whenever a process is waiting for an event that never happens. This can certainly happen due to a simple synchronization error on the part of the programmer. More subtle, however, are the deadlocks due to an optimistic resource allocator that overcommits and is unable to satisfy the needs of any of its users. While the occurrence of a deadlock can always be viewed as an error, deadlock prevention is sometimes possible only with an unduly conservative resource allocator. Many serious systems are written that allow the possibility of deadlock, the hope being that actual deadlock will occur rarely. Hence, techniques of deadlock prevention and deadlock detection and recovery are both important.

In summary, we repeat that the importance of concurrency to many areas of computing is spreading. One significant example at CMU is the Hearsay II System [Fennell 75], in which concurrency is used within a single AI task.

5.3 Address Space Issues

One increasingly important goal of operating systems is to facilitate the controlled sharing of data across several environments or domains [Spier 73]. One obstacle to this sharing is the monolithic address space of conventional computer architecture, which gives each process a single contiguous block of address space. The structuring of address space has been one of the central themes of operating system research. Most influential in this area has been the 'segmentation' concept developed at Project MAC [Dennis 65, Daley 68]. Under this concept, logically distinct data entities (e.g. programs, data bases, communication areas) comprise distinct segments. Each process' address space consists of several of these segments; the process may have different access rights (e.g. read, write, execute only) to each segment. Within the MULTICS system, data is addressed with a 36-bit address: an 18-bit segment number and an 18-bit word offset. MULTICS segmentation was not only an ambitious design in 1965, but is still more advanced than most current systems; thorough study of it is well worth the effort. The most crucial and accepted idea is the abstract data entity or segment, whose controlled sharing is enforced by the system. More controversial, however, is the issue of how these segments should be addressed. The structuring of addresses as

<segment , offset>

pairs in MULTICS solved several problems, but created several others, e.g. who allocates segment numbers? One important reason for studying MULTICS in detail is to appreciate the complexity required to make this approach work.

One significant family of alternatives stems from work done during the early '60's at Rice University and at Burroughs. Within this family, certain addresses are tagged as 'codewords' [Iliffe 62] or 'descriptors' [Organick 73]; possession of a codeword ipso facto confers rights to access the addressed data. These techniques, designed to solve the problems of dynamic storage allocation and efficient subscript checking on array accesses, have also been applied to solve the problems of shared program and data. Their importance in making computers more programmable has been pointed out in [McKeeman 67]. Modern development of these concepts have led to the capability concept which not only facilitates data and program sharing [Fabry 73], but also controls access of all objects in the system, as in Hydra [Wulf 74].

5.4 Memory Management

High speed random access memory has always been at a premium in modern computers. Even with steady improvements in memory technology, the relative cost of primary memory remains very high. The efficient multiplexing of this memory is a persistent technological problem -- one that continues to receive wide attention.

An old solution to the problem involves the overlaying of a process's allocated memory by different programs or data sets dynamically during execution [Lanzano 69]. These overlay techniques tended to be ad hoc and were seldom effectively mechanized. They required some skill on the part of the programmer and made program modification cumbersome. They also failed to allow sharing of memory among different processes.

The most pervasive modern technique for memory management stems from work done on the Atlas computer at the University of Manchester [Kilburn 62]. This technique, now called 'demand paging', divides a memory space into fixed length pages, then loads some of the pages into blocks of main memory termed page frames. Whenever the process tries to access a page not in main memory, the operating system loads the needed page for the process. This technique makes memory management invisible to the programmer. This does ease the crucial software burden felt on overlay systems, but has kept operating systems workers busy trying to find paging policies that are even reasonably efficient. The precarious performance of paging systems was a major concern during the mid-'60's [Naur 65, Belady 69, Randell 68]. While the mechanism of paging is comparatively simple, the policies required for efficiency were slow in coming. For an excellent survey of these issues, see [Denning 70]. While the memory management problem is indeed technological, the basic 'one level store' concept of demand paging has proven quite robust and shows up in cacheing and related memory management techniques at several levels of memory hierarchy [Mattison 70].

5.5 Resource Allocation and Scheduling

The allocation of the physical resources of a computer to its processes represents a family of technological issues that has always been more important in real systems than in the literature. The only unifying goal is the correct multiplexing of a resource under control of an efficient policy such that chance of deadlock is minimized.

Multiplexing of the central processor(s) is fundamental to multiprogramming. Two basic techniques are involved. The first is that of context swap, or switching of the processor from one process to another. The MULTICS implementation in [Saltzer 66], the Burroughs implementation in [Organick 73], and the survey paper in [Lampson 68] all describe some of the more elegant approaches. The second technique is that of basic interprocess communication, where 'processor' includes i/o devices. The conventional scheme involves one processor being interrupted by another processor when communication is desired, e.g. upon the completion of an i/o. At best, these interrupts appear as 'unexpected procedure calls' that can cleanly field the event. More often, however, these interrupts put the system through a precarious and error-prone piece of code that handles the interrupt in an _ad hoc_ fashion. One newer approach treats an interrupt as a V semaphore operation, which unblocks a high-priority process. This approach was first taken in the 'THE' system [Dijkstra 68b] and later pressed into the hardware of the Venus system [Liskov 71]. This unified treatment of interrupts and more normal synchronization primitives allows a cleaner structuring of the system. Brinch Hansen's paper [Brinch Hansen 71] is also useful in showing how semaphore styled synchronization can be unified with processor multiplexing.

[Kleinrock 70] and [Habermann 75b, Ch 6] should be studied for the longer-term scheduling decisions required to make processor multiplexing efficient. Long term schedulers must take the memory, processor, and i/o requirements of each process into account in choosing a reasonable policy. The conflict between response time and throughput as goals for such a scheduler must also be resolved.

The area of resource allocation above the memory and processor levels generally has been neglected in the literature, since it has few important common themes and even fewer neat solutions. Many of these resources, e.g. line printers, are not easily preemptable. They tend to be allocated in unpredictable amounts, e.g. when a compiler keeps lengthening an object file on disc. The allocation of one peripheral may also influence the allocation of another. In short, general solution tends to be elusive and the task is a potential source of deadlocks. [Habermann 75b, Ch 7] is suggested for background in this area.

5.6 Protection Concepts

The basic theme of protection is the controlled access to objects by programs in a computer system. Protection concepts are, therefore, mechanisms designed to solve intrinsic problems. While solutions to the problems vary, some common vocabulary has developed. The things to be protected are called 'objects'; traditionally, objects have been data files, but modern systems often try to treat other objects uniformly. The accessors of these objects are called 'domains' or 'environments'; traditionally, domains have been identical to user-identifications, but modern systems sometimes allow the controlled change of domain, as upon certain kinds of procedure calls; [Lampson 71] gives an excellent survey of the basic issues involved.

One mechanism important to any protection scheme is the basic domain crossing. This crossing, usually in the form of a procedure call, must perform a sophisticated change of environment, yet be absolutély reliable and efficient enough to allow for heavy use without prohibitive overhead. See [Spier 73] for an excellent treatment of the importance of these domain crossings.

One important family of protection concepts, called 'authority based protection', is the rule with conventional systems and was brought to a high level of sophistication in the MULTICS system [Daley 65, Graham 72]. Here a list of authorized domains that may access an object is associated with the object. Each domain may attempt access of any object, but the mechanism checks the domain against the authorization list and prohibits illegal access.

While authority based protection does solve many protection problems, several others of special importance to controlled sharing elude it. One such problem is the 'confinement problem' [Lampson 73], in which a domain wishes to call a procedure and insure that it passes no information to anyone but the caller. Another related problem, that of the cooperation of mutually suspicious subsystems, is discussed in [Schroeder 72].

A second family of protection concepts, called 'capability based protection', may offer the solutions to these problems. The Hydra [Wulf 74, Jones 73, 74] system at CMU is one of the few implementations at the present. Under a capability system, a list of accessing rights for various objects is associated with each domain. An access to an object may only really be attempted when a domain has a capability for the object and, even then, the access rights on the capability can be checked. The power of capability systems comes from their ability dynamically to construct new domains, whose rights are determined by its list of capabilities -- capabilities designed to meet the particular needs of the domain. Refer to [Graham 72] for a comparison of the two protection concepts.

5.7 Examples of Real Systems

Many systems should be studied, not for their solution to any single problem, but for their approach to system design and implementation. MULTICS is certainly an example of both. Their approaches to several issues have already been mentioned. Even more important, however, may be the concept of a computer utility, their use of a high-level language and formal specifications in system implementation, and their bold approach to an ambitious project, truly baroque in every sense. A comparison of [Dennis 65] and [Corbato' 72] should serve to illustrate this.

Two European systems of the late '60's are especially important examples of system design in the post-MULTICS era. The 'THE' system [Dijkstra 68b, read!] at the Technological University of Eindhoven was remarkable primarily (1) in its hierarchical system design and (2) in the carefulness of its implementation. Similarly the RC4000 system [Brinch Hansen 70] contributed (1) the 'kernel' approach to operating system design and (2) messages as an interprocess communications primitive. These were both small systems, compared to the MULTICS, but promised to be equally influential due to their elegance and careful structuring.

While operating systems are generally regarded as software which must 'cope' with given hardware, the Venus system [Liskov 71] demonstrates the value of selecting the proper primitives, then implementing them in a coherent hardware/software system.

The UNIX system [Ritchie 73] is remarkable for two reasons: first, its unified approach to file i/o, physical i/o, and interprocess communication; second, for its exceptional command language and user interface.

One very distinctive approach to system design was taken by IBM's Cambridge research group in the 'virtual machine' concept [Meyer 70]. Here a virtual machine executive, running on a real 360, presents a virtual 360 to each of its users. These users see what appears to be a real 360 and may indeed be a conventional 360 operating system or a recursive copy of the virtual machine executive. This approach has special advantages for the research environment, where many copies of experimental systems can be running simultaneously.

PROGRAMMING SYSTEMS BIBLIOGRAPHY -- 1975

The following abbreviations are used:

AFIPS	American Federation of Information Processing Societies
ACM	Association for Computing Machinery
IRE	Institute of Radio Engineers
BIT	Nordisk Tidskrift for Informations-Behandling
CACM	Communications of the ACM
JACM	Journal of the ACM
CompSurv	Computing Surveys (ACM)
FJCC	Fall Joint Computer Conference (AFIPS)
SJCC	Spring Joint Computer Conference (AFIPS)
NCC	National Computer Conference (AFIPS)
SOSP	Symposium on Operating System Principles (ACM)
CompJ	Computer Journal (British Computer Society)

Aho, Alfred, Hopcroft, John, and Ullman, Jeffrey, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.

Baker, F. T., "Chief Programmer Team Management of Production Programming", IBM SysJ, No. 1, 1972.

Barron, David, Assemblers and Loaders, American Elsevier, 1969.

Beech, D., "A Structural View of PL/1", CompSurv, Mar 70.

A complete review of the PL/1 language is presented. The goal is to answer (by the affirmative) the question: "Is the structural knowledge of PL/1 of manageable proportions?" Both the static and dynamic aspects of the language are examined.

Belady, L. A., and Kuehner, C. J., "Dynamic Space Sharing in Computer Systems", CACM, May 69.

Bell, James, "Threaded Code", CACM, Jun 73.

Berliner, Hans, "A Representation and Some Mechanisms for a Problem Solving Chess Program", (PhD Thesis), CMU-TR, May 75.

Branquart, P., Lewi, J., Sintzoff, M., and Wodon, Pierre, "The Composition of Semantics in Algol 68", CACM, Nov 71.

Brinch Hansen, Per, (ed.), RC4000 Software Multiprogramming System, A/S Regnecentralen, Copenhagen, Denmark, 1969.

Brinch Hansen, Per, "The Nucleus of a Multiprogramming System", CACM, Apr 70.

First, the concept of process is introduced precisely. Then, the system nucleus is presented as the minimum requirements for a multiprogramming system, providing an efficient environment for processes (communication, control, hierarchy).

Brinch Hansen, Per, "Short Term Scheduling in Multiprogramming Systems", 3rd SOSP, 1971.

Brinch Hansen, Per, "Structured Multiprogramming", CACM, Jul 72.

"Event queues" are new features (data structures and operations to perform on them) that are proposed to be added to a high level language in order to describe the operations encountered in a multiprogramming system (process communication, synchronization, etc.).

Brinch Hansen, Per, Operating System Principles, Prentice Hall, 1973.

Brinch Hansen, Per, "Concurrent Programming Concepts", CompSurv, Dec 73.

Language features for multiprogramming (event queues, semaphores, critical regions, monitors) are reviewed. Two principles for the choice of equivalent features are proposed: concurrent programs should be easy to understand, and assumptions about invariant relationships among program components should be checked automatically.

Brooks, Frederick, The Mythical Man-Month, Addison-Wesley, 1975.

Buxton, J. N., and Randell, Brian, (eds.) Software Engineering Techniques, Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969, NATO, Apr 70.

Campbell, Roy, and Habermann, A. Nico, "Specification of Process Synchronization by Path Expressions", International Symposium on Operating System Theory and Practice, IRIA, Apr 74, also Lecture Notes in Computer Science, Springer Verlag, 1974.

Coffman, Edward, and Denning, Peter, Operating Systems Theory, Prentice-Hall, 1973.

Cohen, Ellis, et al, Hydra User's Manual, CMU, Feb 75.

Corbato', F. J., Clingen, C. T., and Saltzer, Jerome, "Multics -- the First Seven Years", SJCC, 1972.

The goals of the Multics project are reviewed and a history of the project including a description of its (then) current status and appearance to users is presented. Then experiences gained from the project are mentioned.

COSINE Committee, "An Undergraduate Course on Operating Systems Principles", Committee on Education, National Academy on Engineering, 1972.

Courtois, P. J., Heymans, R., and Parnas, David, "Concurrent Control with 'Readers' and 'Writers'", CACM, Oct 71.

Two examples illustrate the problem of the exclusive acces to a resource shared by concurrent processes. A programmed solution using P and V operations is given for those two examples.

Dahl, O.-J., and Nygaard, K.,"Simula - An Algol-based Simulation Language", CACM, Sep 66.

Dahl, O.-J., "Discrete Event Simulation Languages", in (Genuys, ed.), Programming Languages, Academic Press, 1968.

Dahl, O.-J., and Hoare, C. A. R., "Hierarchical Program Structures", in (Dahl, Dijkstra, and Hoare) Structured Programming, Academic Press, 1972.

Daley, R. C., and Neumann, P. G., "A General Purpose File System for Secondary Storage", FJCC, 1965.

Daley, R. C., and Dennis, Jack, "Virtual Memory, Processes, and Sharing in MULTICS", CACM, May 68.

Denning, Peter, "Virtual Memory", CompSurv, Sep 70.

Virtual memory is introduced as a solution to the problem of dynamic storage allocation. Virtual memory is then defined and its possible implementations (segmentation, paging, segmentation-paging) are presented, compared and the problems they pose are mentioned. The principle of replacement algorithms is introduced and optimal paging algorithms are presented.

Dennis, Jack, "Segmentation and the Design of Multiprogrammed Computations", JACM, Oct 65.

Dennis, Jack and van Horn, Earl, "Programming Semantics for Multiprogrammed Computations", CACM, Mar 66.

Digital Equipment Corporation, PDP11/20 Processor Handbook, Maynard, Massachusetts, 1971.

Dijkstra, Edsger, "Solution to a Problem in Concurrent Programming Control", CACM, Sep 65.

Dijkstra, Edsger, "GOTO Statement Considered Harmful", CACM, Mar 68.

Dijkstra, Edsger, "The Structure of 'THE' Multiprogramming System", CACM, May 68.

Dijkstra, Edsger, "Cooperating Sequential Processes", in (Genuys, ed.), Programming Languages, Academic Press, 1968.

Dijkstra, Edsger, "A Constructive Approach of the Problem of Program Correctness", BIT, Jul 68.

Dijkstra, Edsger, "Notes on Structured Programming", in (Dahl, Dijkstra, and Hoare) Structured Programming, Academic Press, 1972.

Dijkstra, Edsger, "Hierarchical Ordering of Sequential Processes", in (Hoare and Perrott, ed.), Operating System Techniques, Academic Press, 1972.

Elson, Mark, Concepts of Programming Languages, SRA, 1973.

Elson, Mark, Data Structures, SRA, 1975.

Fabry, R. S., "Capability Based Addressing", 4th SOSP, 1973, also CACM, Jul 74.

Capability-based computers are discussed (in particular the Plessey machine) and, in general, advantages of protection systems based on the concept of capability are presented.

Feldman, Jerome, and Gries, David, "Translator Writing Systems", CACM, Feb 68.

Feldman, Jerome, and Rovner, Paul, "An Algol-Based Associative Language", CACM, Aug 69.

Fennell, Richard, "Multiprocess Software Architecture for AI Problem Solving", (PhD Thesis), CMU-TR, May 75.

Flon, Larry, "A Survey of Some Issues Concerning Abstract Data Types", CMU-TR, 1974.

Flon, Larry, "Program Design with Abstract Data Types", CMU-TR, 1975.

Freeman, Peter, Software Systems Principles: A Survey, SRA, 1975.

Goldberg, J., ed., Proceedings of a Symposium on the High Cost of Software, SRI, 1973.

Goldberg, R. P., "Architecture of Virtual Machines", NCC, 1973.

Goodenough, John, and Gerhart, Susan, "Toward a Theory of Test Data Selection", International Conference on Reliable Software, Apr 75, also SIGPLan, Jun 75.

Graham, Robert, "Protection in an Information Processing Utility", CACM, May 68.

Graham, Robert, and Dennis, Jack, "Protection -- Principles and Practice", SJCC, 1972.

Seven levels of protection systems are distinguished from the simplest (complete isolation of the programs) to the most sophisticated (complete cooperation and shared access to information). The paper gives a comprehensive treatment of one of these levels: protection systems allowing the cooperation of mutually suspicious subsystems.

Gries, David, Compiler Construction for Digital Computers, Wiley, 1971.

Griswold, R. E., Poage, J. F., and Polonsky, J. P., The Snobol4 Programming Language, Prentice-Hall, 1971.

Guttag, John, ed., "An Annotated Bibliography on Computer Program Engineering", Univ Toronto TR, Apr 75.

Habermann, A. Nico, "Prevention of System Deadlocks", CACM, Jul 69.

Algorithms for the prevention of deadlock are presented. With the knowledge of the maximum claims in resources by the different processes, the algorithms determine whether the next allocation leaves the system in a safe state (i. e., a state guaranteeing that the system can eventually grant any request).

Habermann, A. Nico, "Introduction to Algol 60 for those who have used other Programming Languages", CMU-TR, Sep 71.

Habermann, A. Nico, "Synchronization of Communicating Processes", 3rd SOSP, 1971, also CACM, Mar 72.

A formalization of the synchronization primitives is introduced and permits to derive an invariant property of the synchronization mechanisms. This property can be used to prove the program correctness of concurrent processes. This is applied to two forms of programmed mechanisms: the programming of critical sections and the programming of communication between asynchronous sequential processes.

Habermann, A. Nico, "Critical Comments on the Programming Language Pascal", CMU-TR, Oct 73.

Habermann, A. Nico, Cooprider, Lee, and Flon, Larry, "Modularization and Hierarchy in a Family of Operating Systems", 5th SOSP, 1975.

Habermann, A. Nico, Operating Systems, SRA, 1975.

Hansen, Gilbert, "Adaptive Systems for the Dynamic Run-Time Optimization of Programs", (PhD Thesis), CMU-TR, Mar 74.

Hoare, C. A. R., "Record Handling", in (Genuys, ed.), Programming Languages, Academic Press, 1968.

Hoare, C. A. R., "An Axiomatic Basis for Computer Programming", CACM, Oct 69.

Axioms are introduced with each statement of a programming language (assignment, composition, iteration) along with a rule of inference. The axioms and the deduction rule are used to prove formally the correctness of a small program.

Hoare, C. A. R., "Proof of a Program: FIND", CACM, Jan 71.

By determining invariant relations, the proofs of both the correctness and the termination of the program "Find" are given. It is concluded that the methods presented in this example can be applied, in general, to a systematic programming.

Hoare, C. A. R., "Notes on Data Structuring", in (Dahl, Dijkstra, and Hoare) Structured Programming, Academic Press, 1972.

Hoare, C. A. R., "A Structured Paging System", CompJ, Aug 73.

Hoare, C. A. R., "Hints on Programming Language Design", SIGAct-SIGPLan Conference, Oct 73.

Hoare, C. A. R., "Monitors: an Operating System Structuring Concept", CACM, Oct 74.

The concept of monitor is developed. The notion is similar to that of "class" in Simula 67 and can be used to replace critical sections. This is illustrated by several examples.

Hoare, C. A. R., "Data Reliability", SIGPLan, Jun 75.

Holt, Richard, "Some Deadlock Properties of Computer Systems", CompSurv, Sep 72.

Examples of deadlocks and solutions to the problem are first mentioned. Then, a model (based on a graph representation) is introduced. The model takes into account "reusable resources" (to describe objects shared among processes) and "consumable resources" (to describe signals or messages). Efficient detection and prevention algorithms are deduced from the model.

Hopgood, F. R. A., Compiling Techniques, American Elsevier, 1969.

Horning, J. J., and Randell, Brian, "Process Structuring", CompSurv, Mar 73.

Precise definitions are first given of the terminology in use in operating systems (process, processor, computation, etc.). Then, two methods for structuring complex systems are presented: "process combination" and "process structuring," and they are applied to various topics of computer systems (concurrency, synchronization, multiprogramming, etc.).

IBM, IBM System/360 Operating System Report Program Generator Language, IBM Systems Reference Library.

Iliffe, John, and Jodeit, Jane, "A Dynamic Storage Allocation Scheme", Comp J, Oct 62.

Iliffe, John, Basic Machine Principles, American Elsevier, 1968.

Jensen, Douglas, "A Distributed Function Computer for Real-Time Control", Symposium on Computer Architecture, Jan 75.

Johnsson, Richard, "A Survey of Register Allocation", CMU-TR, May 73.

Jones, Anita, "Protection in Programmed Systems", (PhD Thesis), CMU-TR, 1973.

Jones, Anita, and Wulf, William, "Towards the Design of Secure Systems", International Workshop on Protection in Operating Systems, IRIA, Aug 74.

Kilburn, T., et al, "One-Level Storage System", IRE Trans on Elec Comp, Apr 62.

Kleinrock, Leonard, "A Continuum of Scheduling Policies", SJCC, 1970.

A family of scheduling algorithms are presented that takes into account both the service time received by a user and the time spent awaiting for service. The family of algorithms includes all classical scheduling policies (RR, FCFS, LCFS) and, in addition, a continuum series of algorithms that can be tuned by an appropriate selection of the parameters of this family.

Knuth, Donald, and Merner, Jack, "ALGOL 60 Confidential", CACM, Jun 61.

Knuth, Donald, "Letter to the Editor", CACM, May 66.

Knuth, Donald, "The Remaining Trouble Spots in ALGOL 60", CACM, Oct 67.

Knuth, Donald, The Art of Computer Programming: Vol. 1, Fundamental Algorithms, Addison-Wesley, 1968, also 2d Edition, 1973.

Knuth, Donald, The Art of Computer Programming: Vol. 3, Sorting and Searching, Addison-Wesley, 1973.

Knuth, Donald, "Structured Programming with go to Statements", CompSurv, Dec 74.

First, a history of the "goto controversy" is surveyed. Arguments and examples in favor of both pro and con are presented. Then, structured programming is discussed. An extensive bibliography both on the goto controversy and on structured programming is also included.

Lampson, Butler, "Scheduling Philosophy for Multiprocessing Systems", CACM, May 68.

Lampson, Butler, "Protection", 5th Princeton Conference on Information Sciences and Systems, 1971.

A model of a protection system is presented. The model uses an access matrix to describe the process rights and takes into account the mechanisms to control access to information and to insure the protection. Certain techniques for the implementation of the model are discussed.

Lampson, Butler, "A Note on the Confinement Problem", CACM, Oct 73.

Lanzano, B. C., "Loader Standardization for Overlay Programs", CACM, Oct 69.

Lecarme, C., and Desjardins, P., "More Comments on the Programming Language Pascal", Acta Informatica, 1975.

Lefkovitz, David, File Structure for On-Line Systems, Hayden, 1969.

Lindsey, C. H., and van der Muellen, S. G., Informal Introduction to Algol 68, North-Holland, 1971.

Lindsey, C. H., "Algol 68 with Fewer Tears", CompJ, May 72.

Liskov, Barbara, "The Design of the Venus Operating System", 3rd SOSP, 1971, also CACM, Mar 72.

Implemented on the Venus machine (a microprogrammed computer), the Venus operating system was produced to test the influence of the machine architecture on the complexity of the software produced for this machine. The development of the system is reported and shows how the design, following Dijkstra's concept of levels of abstraction, is influenced by the machine architecture.

Liskov, Barbara, and Zilles, Stephen, "Programming with Abstract Data Types", SIGPLan, Apr 74.

A facility is proposed to allow the user of a high level language to describe his own data types: "Abstract data types" are defined by a user by means of the operations that can be performed on them. Implications for the programming language are discussed and examples show the definition and use of abstract data types.

Liskov, Barbara, and Zilles, Stephen, "Specification Techniques for Data Abstractions", SIGPLan, Jun 75.

London, Ralph, "A View of Program Verification", SIGPLan, Jun 75.

McKeeman, W. M., "Language Directed Computer Design", FJCC, 1967.

Madnick, Stuart, and Alsop, Joseph, "A Modular Approach to File System Design", SJCC, 1969, also (Freeman) Software Systems Principles, SRA, 1975.

A file system, designed on the principle of hierarchical modularity, is presented. The file system is intended to be implemented in the environment of a computer network and to handle files kept on removable volume.

Mattison, R. L., et al, "Evaluation Techniques for Storage Hierarchies", IBM SysJ, No. 2, 1970.

Maurer, W. D., and Lewis, T. G., "Hash Table Methods", CompSurv, Mar 75.

Meyer, R. A., and Seawright, L. H., "A Virtual Machine Time-Sharing System", IBM SysJ, No. 3, 1970.

Mitchell, James, "The Design and Construction of Flexible and Efficient Interactive Programming Systems", (PhD.Thesis), CMU-TR, Jun 70.

Moses, Joel, "The Function of Function in LISP; or Why the FunArg Problem should be called the Environment Problem", MAC-TR, Jun 70.

Naur, Peter, et al, "Revised Report on the Algorithmic Language ALGOL 60", CACM, Jan 63.

Naur, Peter, "The Performance of a System for Automatic Segmentation of Programs within an Algol Compiler", CACM, Nov 65.

Naur, Peter, and Randell, Brian, Software Engineering, Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968, NATO, Jan 69.

Needham, R. M., "Protection Systems and Protection Practices", FJCC, 1972.

Organick, Elliott, and Cleary, J. G., "Data Structure Model of the B6700 Computer System", SIGPLan, Feb 71.

Organick, Elliott, Computer System Organization: the B5700/B6700 Series, Academic Press, 1973.

Pakin, Sandra, APL 360 Reference Manual, SRA, 1968.

Parnas, David, "Information Distribution Aspects of Design Methodology", CMU-TR, 1971.

Parnas, David, "A Technique for Software Module Specification with Examples", CACM, May 72.

Using examples, it is shown how to write a large program by decomposing it into subprograms each with a well defined function and specification.

Parnas, David, "On the Criteria to be Used in Decomposing Systems into Modules", CACM, Dec 72.

The decomposition of a system into modules may have different results depending on the criteria used for the decomposition. Criteria are discussed to achieve an efficient implementation.

Parnas, David, "On a 'Buzzword': Hierarchical Structure", IFIP Congress, 1974, Vol. 2.

Presser, Leon, and White, John, "Linkers and Loaders", CompSurv, Oct 72.

Using examples from the IBM system 360, the linking and loading functions are presented in detail. Implementation of linkers and relocating loaders is also mentioned.

Pratt, Terence, Programming Languages: Design and Implementation, Prentice-Hall, 1975.

Quam, L. H., and Diffie, W., "Stanford Lisp 1.6 Manual", SAIL-ON-28.7, 1973.

Randell, Brian, and Russell, Lawford, Algol 60 Implementation, Academic Press, 1964.

Randell, Brian, and Kuehner, C. J., "Dynamic Storage Allocation Systems", CACM, May 68.

Randell, Brian, "A Note on Storage Fragmentation and Program Segmentation", CACM, Jul 69.

Results of a simulation to measure storage fragmentation are reported. Both internal and external fragmentation are taken into consideration and it is pointed out that in trying to decrease the external fragmentation (by reducing the number of block sizes), the internal fragmentation increases and leads to a worse utilization of the total storage.

Rappaport, Robert, "Implementing Multi-Process Primitives in a Multiplexed Computer System", MAC-TR, Nov 68.

3 8482 00582 2248

Ritchie, Dennis, and Thompson, Ken, "The UNIX Time-Sharing System", 4th SOSP, 1973, also CACM, Jul 74.

A comprehensive description of the file system of UNIX, a large time sharing system, is presented along with the user facility to handle his files.

Robertson, George, "L*(I) Reference Manual", CMU-TR, Oct 75.

Saltzer, Jerome, "Traffic Control in a Multiplexed Computer System", (PhD Thesis), MAC-TR-30, MIT, 1966.

Sayre, D., "Is Automatic 'Folding' of Programs Efficient Enough to Replace Manual?", CACM, Dec 69.

Schaffert, Craig, Snyder, Alan, and Atkinson, Russ, "The CLU Reference Manual", MAC-TR, MIT, Jun 75.

Schroeder, M. D., "Cooperation of Mutually Suspicious Subsystems in a Computer Utility", (PhD Thesis), MAC-TR-104, 1972.

Sites, Richard, Algol W Reference Manual, Stanford Computer Science Department Report STAN-CS-71-230, 1972.

Spier, Michael, Hastings, Thomas, and Cutler, David, "An Experimental Implementation of the Kernel/Domain Architecture", 4th SOSP, 1973.

The implementation of a "protective operating system framework" is reported. The implementation of this system (on a PDP-11/45) is described as being modular and uses the concepts of "kernel" (the heart of the operating system) and of "domain" (the memory space dedicated to a procedure).

SRI, "On the Design of a Provable Operating System", International Workshop on Protection in Operating Systems, IRIA, Aug 74.

Steele, Guy, "Multiprocessing Compactifying Garbage Collection", CACM, Sep 75.

Tsichritzis, Dionysios, and Bernstein, Philip, Operating Systems, Academic Press, 1974.

Watson, Richard, Timesharing System Design Concepts, McGraw-Hill, 1970.

Wegner, Peter, Programming Languages, Information Structures, and Machine Organizations, McGraw-Hill, 1968.

Weinburg, G., Psychology of Computer Programming, van Nostrand Rheinhold, 1971.

Weinstock, Chuck, "A Survey of Protection Systems", CMU-TR, Jul 73.

Weissman, Clark, LISP 1.5 Primer, Dickenson, 1967.

Wirth, Niklaus, "Program Development by Stepwise Refinement", CACM, Apr 71.

Using an example, it is shown both how to decompose a program and how to structure the corresponding data by successive approximations.

Wirth, Niklaus, "The Programming Language Pascal", Acta Informatica, 1971.

Wirth, Niklaus, Systematic Programming: An Introduction, Prentice Hall, 1973.

Wirth, Niklaus, "An Assesment of the Programming Language Pascal", SIGPLan, 1975.

Wodon, Pierre, "Still Another Tool for Synchronizing Cooperating Processes", CMU-TR, Aug 72.

Wulf, William, Russell, D. B., and Habermann, A. Nico, "Bliss: a Language for Systems Programming", CACM, Dec 71.

Bliss, superficially an Algol or a PL/1-like language, is a system implementation language. It provides an efficient access to the hardware features of the machine (initially a PDP-10) along with the possibility to define complex data structures, to write coroutines and, in addition, produces efficient code.

Wulf, William, and Shaw, Mary, "Global Variables Considered Harmful", CMU-TR, Aug 72.

A case against the use of global variable is presented. It is argued that "non local" variables lead to obscure programs and can be misused. The scope of variable in Algol is a contributing factor in this difficulty.

Wulf, William, "ALPHARD: Towards a Language to Support Structured Programs", CMU-TR, Apr 74.

Wulf, William, et al, "HYDRA: The Kernel of a Multiprocessor Operating System", CACM, Jun 74.

The design philosophy of HYDRA, the kernel of C.mmp, is presented. HYDRA provides mechanisms to handle "objects" efficiently and in a secure way.

Wulf, William, et al, Design of an Optimizing Compiler, American Elsivier, 1975.

Wulf, William, "Languages and Structured Programs", to be published in 1976.