PROGRAMMING SYSTEMS STUDY GUIDE

August, 1972

A. N. Habermann

A. K. Jones

J. M. Newcomer

This study guide provides a survey of data structures and programming system concepts. The authors feel that programming systems can be studied more fruitfully if this study is preceded by a study of data structures, since data structures are fundamental to all programming systems.

A Programming System transforms an underlying machine into one that provides facilities to write and run programs in a more suitable environment. Such an environment suppresses the "constant" aspects of programmming, i.e., those aspects which do not vary from task to task; it allows the programmer to concern himself only with those issues relevant to programming his algorithm. Such systems and their uses are:

Assemblers & Compilers
: Allow the use of symbolic notation more consistent with the description of algorithms, and provide a mapping from this notation into machine instructions.

Editors
: Provide the capability for facile manipulation of symbolic representation of text.

Loaders
: Allow separate compilation and modification of programs which may be combined only when (or as) needed.

Debugging facilities
: Permit the programmer the use of symbolic notation when investigating the state of a partially executed object program. In this sense, debuggers provide an inverse mapping of the compile function; the completeness of this mapping is usually used as a measure of the power of such a facility.

Procedure libraries
: Provide a common pool of often-used functions which can be shared among many users.

Device management systems
: Veil device characteristics irrelevant to most programs and permit multiplexing of shared devices where applicable.

File systems
: Allow programmers to share information by means of a uniform and reliable facility for storing programs and data for later use.

Multiprogramming systems
: Multiplex hardware resources--both processors and peripherals--in order to gain multiaccess and the sharing of data and equipment dynamically.

I. Data structures

There exist a number of information-containing structures which are used both in implementations and in precise descriptions of programming systems. Understanding the format of these structures, the implicit relationships between elements within a single structure, and the algorithms used to access and manipulate the structures is crucial to the understanding of programming systems.

Algorithms and concepts which are pertinent to data structures valuable for programming systems are further defined and discussed in the cited reference pages.

A. Linear Structures [Kn68 234-45]

Stack--a linear list for which all insertions and deletions are made at one end of the list.
Terms--Push, Pop, LIFO [Hop69 11], stack pointer [Kn68 240]

Queue--a linear list for which all insertions are made at one end of the list; deletions and references are made at the other end.
Terms--Insert element, Delete element, FIFO

String--an ordered sequence of elements which may be altered before or after any element. [Hop69 5]
Terms--Concatenate strings, Break strings apart, Pattern match

Representation of Structures
- Table [Kn68 240-5]
  - Table lookup, Key [Hop69 16-19]
  - Hash table [Hop69 19-28, Weg68 113-20]
- Linked list [Kn68 251-8]
- Circularly linked list [Kn68 270-2]
- Doubly linked list [Kn68 278-80]
- Vector--array [Hop69 12-15]
  - Dope vector, Iliffe vector

B. Trees--a finite set T of nodes such that a) one node is designated root(T) b) the remaining nodes are partitioned into disjoint sets, each set being a tree. [Kn68 305-13]

Traversal algorithms [Kn68 315+]

Threaded lists and trees [Kn68 319-22]

Representation of trees [Kn68 322-7]

Plex [Hop69 9-10]

Garbage collection [Kn68 406-414]

C. Problems: Using these structures and their associated algorithms is the best way of learning them. You should be conscious of using a variety of structures and efficient algorithms for accessing them in the several programming problems done in other portions of this course.

Further problems from Knuth, Chapter 2 [Kn68]

p238, problems 1,2,3
p248, problems 2,8
p266, problem 7
p294, problem 1,2
p328, problems 2,4,5

II. Assemblers, Compilers, and Loaders

Assemblers, compilers, and loaders each abstract the machine-dependent aspects of a program into a symbolic representation, and postpone as long as possible the binding time of that representation, i.e., the point at which the symbol becomes fixed to a machine representation. This abstraction is partially realized by the use of names. Assemblers use names to represent machine operations, data locations, and data itself. Compilers go further, and allow symbolic representation of computation (A←B+C) and control (for I ← 1 step 1 until N do...). Loaders allow one to defer binding names to absolute machine addresses by allowing some names to be bound relative to a base (relocatable address) and others to be left unbound. The loader then has the responsibility of making relative addresses absolute and locating (usually by library search) bindings for unbound names.

The function of assemblers, compilers, and loaders is to map some representation into another representation. For example, an assembler maps operation names (e.g., "ADD") into their machine representation (e.g., 270 octal), a compiler maps arithmetic and control statements into appropriate sequences of instructions, and a loader maps relative addresses into absolute addresses.

Assemblers

An assembler is a mapping from a set of strings in one alphabet to a set of strings in another alphabet [Mea67], i.e.,

A: alphabet* → alphabet2*

> Note: If X denotes the alphabet [a,b] then X* denotes the set of all strings composed of the symbols in X, i.e.[the null string, a, b, aa, ab, ba, bb, aaa, . . .]

As an example, opcode names are mapped from a mnemonic representation ("ADD", "MOVEM", "FSBR") to their machine code representation (octal 270, 202, and 154 respectively for the PDP 10).

A more powerful use of assemblers is to map strings representing symbolic data into its internal representation e.g., allowing the user to specify "ABC==12*5" and then mapping all occurrences of the string ABC into the integer value 60. Names may be allowed to take on values of data or instructions ("labels"), or to be declared unbound ("EXTERNAL"), and thus free ourselves from the requirement of needing to know the absolute position of the information in advance.

data names → numbers representing absolute, relative, or unbound addresses.

For further discussion of assemblers, see [Weg68 107-124].

Problem: A typical problem in one-pass asemblers is when a reference to a label is made, but the label has not been seen. We cannot pass over the source code twice (once to locate labels, once to generate code), so the address to be used for the label is unknown. Your mission, should you decide to accept, is to devise a scheme for resolving the forward reference problem:

```
      .
      .
   GOTO A
      .
      .
   A: ...
```

You may assume for simplicity that instructions are laid down in memory directly where they will be executed, and these instructions will all be available (i.e., you can look at any instruction once it is laid down; only the source may not be re-examined). You may want to implement such a scheme in Algol or APL, using some notation like ">A" meaning "GO TO A"

and ":A" defining A. Any other symbols may be assumed to be random instructions and only fill space. An array or vector may be used to represent memory.

Compilers

A compiler is a mapping from one language into another much in the manner of an assembler. However, in the case of a compiler the source language (Ls) and the target language (Lt) are not necessarily as closely related as in an assembler.

C: Ls → Lt

One of the first operations a compiler performs is the mapping of names into a more easily manipulated information structure. This is done by constructing a symbol table. A good discussion of tables and their mapping functions may be found in Hopgood [Hop69 16:28] and Wegner [Weg68 113-20]. Basically one provides a mapping function which maps names into integers. These integers may represent memory addresses, table indices, or similar information, but for a given name the mapping must be unique [note this does not eliminate hash coding, since the hash algorithm provides a unique mapping by finding an available "slot"]. One also provides an inverse mapping (usually much simpler) by which the attributes of the name may be obtained, including the original sequence of characters which constituted it.

In addition to the attribute which is the character string representing the name, we may also include in the symbol table type information (such as integer, real, etc.), scope information (local, global, own, etc. and block levels) and whatever relevant attributes may be required by the compiler for the specific language. This function is referred to as lexical analysis and the part of the compiler which performs it is the lexical analyzer. In addition to mapping names into integers, all delimiters, operators, and reserved words are mapped into some internal representation.

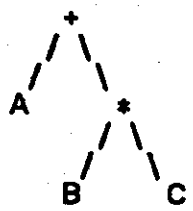
Example:

Lex: A → 128
B → 417
C → 212

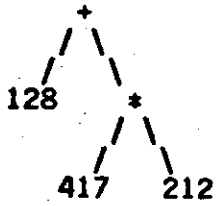
A compiler also maps infix expressions into some internal representation, typically (but not necessarily) trees. If you are interested, algorithms for accomplishing this may be found in Wegner [Weg68 237-44] and Hopgood [Hop67 45:64]. More advanced discussions may be found in Rosen [Gra67, SB67,CS67]. Thus an expression such as

A + B * C

is transformed into the tree



However, since the names have already been mapped into integers (as in the previous example) the tree is actually stored as:



We should also note that the operators "+" and "*" have been mapped into some internal representation; however, that is not relevant to this example and hence the external representation is used for convenience.

Next, the internal representations of the names are mapped onto machine addresses. The means of doing this depends on the machine, the language, the attributes of the name, etc. For a static storage organization, where all names have fixed locations, [e.g., FORTRAN], where one machine word holds the information associated with a name, and where each name is a scalar rather than an array, then we may assume we need as many locations as we have names. If we start at some arbitrary point we can assign ascending addresses for each name, e.g., the first name at location 1000, the second at 1001, etc.

In addition to the types of addresses produced by assemblers (relative, absolute, and unbound), compilers may also produce addresses which are relative to a dynamic base, e.g., a stack pointer. This type of address is used when accessing local-type names in block-structured languages such as PL/1, ALGOL or BLISS. The implementation technique , known as "display", is described in detail in [Ka67] and [RR64].

Given bindings for names, we may now proceed with code generation. In generating code the tree is traversed in some order which depends on the code generating and optimization algorithm. For trees representing simple expressions we may use endorder traversal [Kn68 316], evaluating each subtree and then applying the root operation. The first transformation of the tree (for a single-accumulator machine) would produce:

New tree:

Code:
LOAD B
MULT C

The second transformation discovers that the accumulator contains a partial result and thus adds only one more line of code:

New tree:

acc

Code:
LOAD B
MULT C
ADD A

Other methods of code generation are discussed by Graham [Gra67] and Hopgood [Hop69].

The following problems are given for the interested reader to pursue:

1) Transform the following expressions into their tree representations:
   a) A + B * C / D
   b) X + Y - Z - A
   c) A * (B + C) / D
   d) A + (B * C) - D
   e) (X + Y) * (A - B)
   f) A + (B * (C / D))

2) Note that by traversing the tree in endorder order we can obtain a linear representation of the expression. This form is known as postfix, Reverse Polish, or Polish string form (the latter names after the Polish logician Jan Lukasiewicz). Expressions are normally written in infix notation, where binary operators are placed between their operands and unary operators precede their operands. Ambiguities (such as A+B*C) are resolved by precedence rules or parentheses. In prefix noation binary operators precede their operands, and in postfix notation binary operators follow their operands. In addition to being a parentheses-free notation (since it is unambiguous), in postfix representation the operands and operators appear precisely in the form required if a stack is used for evaluation. The rules for postfix form are simple:

   1) If x is a simple operand, then it is an expression.

   2) If x and y are expressions, and * is a binary operator, then xy* is an expression.

   3) If x is an expression, and % is a unary operator, then x% is an expression.

4) All expressions may be derived by application of rules 1-3.

a) By traversing the trees derived in problem 1, obtain the postfix representation of the expressions.

b) By the use of the stack primitives PUSH and POP [Kn68 237-38], show how a stack may be used to evaluate a postfix expression.

c) Devise an algorithm for converting infix expressions to postfix expressions [RR64 149ff]. If you have time and/or interest, implement this algorithm in your favorite language.

d) Devise an algorithm for evaluating postfix expressions. If you have time and/or interest, implement this algorithm (you may implement it as part of problem c if you did that, or by itself). For operands you may allow single integers which take on their own value, e.g., 13+4* would yield 16.

3) A simple hashing function can be described as follows: take the binary integer representation of the string to be hashed (in the PDP-10 each character is 7 bits wide), and multiply it by a "hash constant". Choose a subfield of the resulting product as the index into the table. For example, assume all identifiers are three characters long (21 bits on the PDP-10). From the 42-bit product choose a subfield n bits wide. This now gives an index into a table of size 2**n.

The choice of the hashing constant is very critical in this scheme. We would like to hash the identifiers into as many unique values as possible to prevent clustering, i.e., where several different identifiers hash to the same value. If we choose a subfield of the product, say bits 14-20 (numbering the rightmost bit 0), then a hash constant of 1 (binary) will give us a hash which depends exclusively on the first character, so that CAT, COW, CON, and CLA all hash to the same value. Similarly, using hash constants of 10000000 or 100000000000000 will give hash values depending on the middle or final letters respectively.

A short programming problem: using Algol, write a program which does the following:

1) Reads a string and converts it to an integer.

2) Multiplies the integer by some hash constant (you should leave this variable so you can experiment with it).

3) Places the string in a symbol table, using a simple "if this slot is full, use the next" strategy. Be sure to allow duplicate strings to use the same slot. The table is considered circular, with the first cell following the last.

4) Keeps track of the number of tries required to find a vacant slot in the table.

Extensions:

1) Study the effects of various hash constants on the performance.

2) Study the effects of other strategies to find vacant slots (quadratic search, etc.).

3) Decide which hash constant is best for a) a table less than 50% full; b) a table more than 50% but less than 75% full; c) a table more than 75% full.

4) Decide which search strategy is best for the same conditions as in 3, above.

4) Advanced readings in compiler techniques include discussions of displays [RR64, Ka67, Wu71b]; syntax analysis [SB67, CS67, Con63, Gra67, FI63]; for hash coding [Weg68 113-20, Bell70, Day70, BK70].

Loaders

A loader is a program which takes the output of an assembler or compiler and places it in the machine memory ready for execution. Not all compilers require loaders; systems such as WATFOR and WATFIV generate code directly in memory where it is executed.

A loader may be simple or complex. A simple loader may just read in a paper tape containing addresses and data, and place the data where the addresses specify. Such a loader may not require more than a dozen or so instructions. A sophisticated loader accepts files containing programs with relative (relocatable) addresses (relocatable programs), and performs such operations as binding relocatable addresses to absolute addresses and binding unbound names to addresses (which may be themselves relocatable). Such a loader can accept a collection of programs and load them to form a single program; in addition, other collections of programs may be given as "libraries". These sets of programs are searched to locate bindings for unbound names, such as subroutine names.

Programs known as linkage editors have most of the capabilities of loaders, as well as editing capabilities which allow such operations as changing the names of programs, subroutines, entry points, etc. However, rather than producing a program in memory ready to execute a linkage editor produces another relocatable program file. This technique may be used to collect subprograms together in a single subprogram, thus reducing the overhead in loading such a collection.

Because of the lack of adequate descriptions of loaders, the authors have felt it advisable to include a short discussion here.

The most basic function of a loader is to transform relocatable addresses into absolute addresses. When the loader begins loading a collection, it initializes a variable containing the relocation offset to some initial value. This may be the first free location after the monitor (in a real-memory operating system) or zero (in a virtual memory system). While reading the program file, this value is added to each relocatable address to form an absolute address. When a program is completely loaded, the offset is incremented by the length of the program and then the next program is loaded in a similar manner.

In addition, the loader must bind the unbound addresses to absolute addresses. In order to accomplish this, two lists are maintained by the loader: those names still unbound, and those names which have been bound. When an unbound name is encountered in loading a program, the bound names list is examined. If the name appears on this list, it is given the value to which it has been bound; if it does not appear, it is added to the list of unbound names. The list of unbound names also enables the loader to locate all locations which must contain the address to which each name is bound. This is done very simply: for a single location, the unbound name list contains the address of that location. For more than one location, the

unbound name list contains the address of one of the locations; this location in turn contains the address of the next location, and so on. If an unbound name already exists in the list, then the tail of its list is set to point to the head of the current list, and the new list is pointed to by the table.

A name is bound to an address when a program containing that name is loaded. The name may be either absolute or relocatable; in the latter case the relocation offset must be added to its associated value. When a name is bound, it is added to the list of bound names (and it must not duplicate one already there with a different value). The list of unbound names is then searched to resolve occurences of this name, if any exist.

Upon completion of loading the programs specified by the user, the unbound names list should be empty. If it is not, the loader will search the libraries specified in an attempt to find bindings. If a program which contains one of the unbound names is located, the loading process begins again. Note that the new program loaded may itself contain unbound names, and further searches may be required to resolve these.

After the libraries have been searched, if any unbound names remain an error message is usually given and some standard action taken.

Problem: Study the forward reference problem under Assemblers. Given that we cannot randomly access the code produced (since it has been written on an output file) show what additions would have to be made to the assembler and/or output to allow a loader to fix the forward references at load time. Show what additional operations the loader must perform. If you want, implement this scheme as an extension of the asembler problem (i.e., you cannot access an array element lower than the current index of your memory array).

III. Modularity

Modularity is the isolation of functionally separate aspects of a system. It contributes to the understanding of complex systems, to the ease of construction of large systems and to the ease of modifying existing systems.

As an example, consider the compiler mapping discussed previously:

$$C: Ls \rightarrow Lt$$

We can break the compiler down into several components: the lexical analyzer, the syntax analyzer, and the code generator.

The function of the lexical analyzer is to scan the input text searching for delimiters, collecting names and storing them in the symbol table, and mapping all names, delimiters, reserved words, operators, etc. into some internal representation which can be processed efficiently by the syntax analyzer. The output of a lexical analyzer consists (usually) of a string of items called lexemes, which form another language, so we have:

$$Lex: Ls \rightarrow Lx$$

The syntax analyzer accepts the lexeme string, and after verifying that the stream is structured according to certain specific rules (there are many ways of doing this: [Hop69 45:64, Gra67, SB67, CS67]) produces another internal representation of the source program. This may be Polish strings, trees, a new lexeme string, or any combination of the previous.

$$Syn: Lx \rightarrow Li$$

Up to this point the representation may be independent of the machine on which the program is to run (although this is not usually so). The internal representation is now passed to the code generator, which produces the actual machine repesentation:

$$Code: Li \rightarrow Lt$$

As in this example, a program which performs a complex task may be separated into several algorithms, each dependent on the existence of a subset of the others and a well specified communication mechanism for receiving input and exporting output, but not dependent upon what might be termed the idiosyncracies of the implementations of other modules.

Each module can be constructed and except for these stated dependencies may be shown to be correct without further regard for how other modules perform their functions.

Modularity appears in many forms. A subroutine or procedure having a well defined means for receiving input and returning output may be modular. If so, it can be replaced by a new implementation observing the same assumptions about communication of input and output, without necessitating any change in the calling program. [Kn68 182-9]

Modularity is not inconsistent with extensive feedback or cooperation between modules. Consider coroutines. Unlike subroutines which provide a hierarical relationship with the caller above the called, coroutines are symmetric--each routine may call any of the others. Only a single routine is active at a time and when it is called, a coroutine continues from the point at which it last called another of the coroutines [Kn68 190-6, Weg68 324-28]. The first documented description of coroutines appeared in [Con63]. Coroutines have been the rather elegant organizing principle in two recent projects here: [Kru71] [Wu71a]

A third implementation of modularity is the process. A process embodies an instance of execution of a program. Processes may be independent or may cooperate with one another through use of an interprocess communication facility. Since each process executes at an independent rate, parallel processes are a natural vehicle for implementing complementary functions which are not necessarily performed in sequential order [Br69 18-20,27-35].

Further references may be found in [COS71 M5.6, COS71 M8.1]

Problem: Consider two tasks, production and consumption.

Producer code	Consumer code
A: secure a buffer	C: consume document
produce document	release buffer
mark document for consumption	
go to A	go to C

Assume a pool of k (>1) buffers. Write a pseudocode program to relate production and consumption as subroutines, coroutines and as parallel processes. Do not concern yourself with the problems of 'producing a document' or 'consuming a document.'

IV. Concurrent Sequential Processes

Permitting more than one process to execute concurrently in one computer system may introduce interaction between the processes--either by explicit wish of the programmer or implicitly through sharing of the computer system resources. [COS71 Module 3 (COS71 is a plan for an undergraduate course on operating systems principles. It contains both prose description of important concepts and pointers into an extensive and up to date bibliography. The whole document should be assumed to be an extension of this study guide and should be read. Where interest or lack of background require it, refer to some of the references cited in COS71 keeping in mind that this is to be a survey of the area of operating systems with an emphasis on terminology and the major concepts involved, not an in depth study of techniques to solve particular problems.)]

Two processes sequentially executing steps in their respective programs are concurrent if at some instant both are beyond the initial step but have not yet completed execution. [Dij65] [DvH66]

Associated with each process is an environment--all those variables the process may potentially reference in its next operation. If several processes share a common variable or data cell, care must be taken to insure that the effect of one reference to the cell does not invalidate a simultáneous reference to the same cell by another process. When two processes require access to the same data with the possibility that it would be modified during the access, the code used by each comprises a critical section. Two processes may not simultaneously be in the same critical section. This is the problem of mutual exclusion and requires that operations on a datum which intrinsically take some duration of time, be primitive or uninterrupted by another operation upon the same datum. [Cos M3.2]

Another form of synchronization is required when one process depends on another asynchronous process having completed some function. No assumptions may be made about the rate at which a process executes. It may therefore be necessary to delay one process pending notification that another has completed the required action. The various forms of synchronization are described in the COSINE report [COS71 M3] and by Dijkstra [Dij68b].

With synchronization is introduced the possibility of deadlock: a process is waiting for an event which will never occur--e.g. two processes, each waiting for the other to provide some information (circular wait) [Hab69]. Deadlocks occur not because of programming errors in single programs, rather because of combinations of mistakes in an attempt to cause processes to cooperate.

Processes may require a communication facility in addition to the synchronization facility so that the one or more sender processes may deposit messages in a structure called a mailbox or a message buffer. The

one or more receiver processes may remove and read these messages. Synchronization is required to provide cooperative use of the mailbox (altering the state variables of the mailbox) and to prevent an overflow of messages or an underflow (the attempt by a receiver to remove a message from the empty mailbox.) [Br69 21-6] [SO69]

To implement a problem solution using concurrent cooperatng processes, two requirements must be met:

1) create a modular implementation of an algorithm for each process
2) specify precisely the interactions between processes

We have introduced synchronization and interprocess communication to provide the well defined interactions between processes.

Besides providing a clear conceptual way of specifying complex asynchronous algorithms, parallel processes are a convenient concept for building multiprogramming operating systems--which permit efficient utilization of hardware through concurrent use of peripheral devices and processors. [Br69 18-20,27-35]

Problems:

1) Those included in Module 3 of COS71.

2) Assume that the instructions 'set a variable', 'inspect a variable' and 'exchange the values of two variables' are indivisible operations, i.e., they can be considered as "timeless". Two solutions for a problem are described below. One contains a fundamental sin against reliable and correct system design.

The problem was to program two asynchronous processes, A and B, which should operate in a specific part of their program on a common data base, but the execution of these specific parts should never overlap in real time. The given solutions are:

```
1. A:  a := 1;
   if b then
     begin while PR=BB do a:=0;
           goto
     end
   execute specific part A;
   PR := BB;
   a := 0;
   - - - -
   - - - -
   goto A;
     goto A
```

```
2. A:  a := 0;
exchange (a,LOCK)
       if a ≠ 1 then
   begin while PR=BB do;
       goto A
   end;
execute specific part A;
PR := BB;
   exchange (a,LOCK);
   - - - -
   - - - -
```

The initial value of LOCK = 1.

The program for B is obtained in both solutions by systematic changes of A, a, AA into B, b, BB. The question is to find out which of the two solutions is the bad one and to explain why.

V. Operating Systems

The purpose of an operating system is to map one machine into another.

$$O: Mr \rightarrow Mv$$

The real machine (Mr) consists of physical components and information structures (core, disks, cards; disk records, tape records, etc.) while the machine the user sees (Mv) deals in logical components and information structures (virtual memory, files, records, etc.). It is possible by such mappings to provide the user with a machine possessing more capabilities than the actual physical machine, for example, as a timesharing system provides a multiplicity of virtual machines which do not interfere with each other, or a paging system allows a user to maintain an address space larger than the amount of core storage that actually exists on the machine.

One of the prime means that an operating system has of accomplishing this mapping is by allowing postponement of the time that certain bindings are made. The actual binding time is dependent upon what is being bound: logical devices may be bound to physical devices at the time they are requested by the program, or they may be bound well in advance, such as before the program is permitted to execute at all. An address in the user's program may not be bound to a physical address until the instruction fetch cycle begins. On the other hand, file names can be bound to a set of physical records on some medium (drum, disk, tape) and retain that binding as long as the file exists [COS71 M5]. A user program may request that a file be printed on the line printer, and may receive confirmation of printing, although the printing is physically done long after termination of the user program.

Another function which operating systems perform, device management, is the mapping of physical information structures on the real machine onto logical information structures on the virtual machine. Thus a user may talk about "files" and "records" without needing to be aware of the physcial representation of such structures. Indeed, information may exist on several physically unlike media such as drum, disk, tape, or cards, and yet appear precisely the same to the user.

An operating system provides for efficient use of the base machine Mr by managing the available resources to optimize their usage. For example, a multiprogramming system optimizes processor usage by allowing some processes to proceed when others cannot and by encouraging activity to proceed in parallel on peripheral devices [COS71 M7]

In a similar manner other preemptible resources may be shared among several processes: core storage, channel usage, etc. In addition to allowing more effective use of these resources, there is the possibility of a deadlock condition ensuing [Dij68b, Hab69, Cos71 M3.6].

An important distinction must be made between a mechanism which is an algorithm to perform an operation, and a policy --an algorithm which decides if or how to apply such mechanisms. Thus we have a mechanism which gives a processor resource to a process in order to let the process execute and a scheduling policy to decide which of the competing processes should receive the processor next. We have P and V operations and a policy which dictates the selection of the next process to pass a semaphore.

In general the designer of a system fixes the mechanisms but permits policy variation (such as parameterizing a scheduler). An even more flexible design would allow changing the entire modular policy, not merely the policy parameters.

Problems: We have tried to name and roughly describe some of the major concepts and concerns of operating systems. Reading some overviews of specific systems may knit together some of these ideas. [Dij68a] and [Br70] rank among the "clearer" of such overviews.

Memory management problem--Read [Wil68 35-47] and these references from the COS71 bibliography: Denning(21) [highly recommended], Randell(70), Belady(4). Be able to define: segment, page frame, demand paging, relocation, working set, thrashing, swapping, virtual memory, placement policy, migration.

Memory protection issues---Read [Wil68 49-59].

BIBLIOGRAPHY

[Bell70] Bell, J. R. "The Quadratic Quotient Method: A Hash Code Eliminating Secondary Clustering", CACM 13,2 (Feb 70).

[BK70] Bell, J. R. & C. H. Kaman, "The Linear Quotient Hash Code", CACM 13, 11 (Nov 70).

[Br69] Brinch-Hansen, P. RC4000 Software Multiprogramming System. A/S Regnecentralen, April 1969.

[Br70] Brinch-Hansen, P. "The Nucleus of a Multiprogramming System". CACM 13 (April 70) 238. System

[Con63] Conway, M. E. "Design of a Separable Transition-diagram Compiler", CACM 6, 7 (Jul 63) pp 396-408

[COS71] COSINE Committee "An Undergraduate Course on Operating Systems Principles". Commission on Education of the National Academy of Engineering. June 71.

[CS67] Cheatam, T. E. & K. Sattley, "Syntax Directed Compiling" in Programming Systems and Languages, Saul Rosen ed.

[Day70] Day, A. C. "Full Table Quadratic Searching for Scatter Storage", CACM 13,8 (Aug 70).

[Den67] Dennis, J. B. Segmentation and the Design of Multiprogrammed Computer Systems in Programming Systems and Languages, Saul Rosen, ed.

[DvH66] Dennis, J. B., and van Horn, E. C. Programming Semantics for Multiprogrammed Computations. CACM 9,3 (Mar 66).

[Dij65] Dijkstra, E. W. "Solution of a Problem in Concurrent Programming Control". CACM 8,9 (Sept 65), 569.

[Dij67] Dijkstra, E. W. "Recursive Programming" in Programming Systems and Languages, Saul Rosen ed.

[Dij68a] Dijkstra, E. W. "The Structure of THE Multiprogramming System". CACM 11,5 (May 68) 341-6.

[Dij68b] Dijkstra, E. W. "Co-operating Sequential Processes", in Programming Languages, F. Genuys, ed.

[Fl63] Floyd, R. W. "Syntactic Analysis & Operator Precedence", JACM 10,3 (Jul 63).

[Gra67] Graham, R. M. , "Bounded Context Translation" in Programming Systems and Languages, Saul Rosen ed.

[Hab69] Habermann, A. N. "Prevention of System Deadlocks". CACM 12 (July 69).

[Hop69] Hopgood, F. R. A. Compiling Techniques

[Ka67] Kanner, H., P Kosinski, & C. L. Robinson, "The Structure of Yet Another Algol Compiler" in Programming Systems and Languages, Saul Rosen ed.

[Kn68] Knuth, D. The Art of Computer Programming Volume 1

[Kru71] Krutar, R. Conversational System Programming, Ph. D. thesis, C-MU 1971 (to be published)

[Mea67] Mealy, G. H. "A Generalized Assembly System" in Programming Systems and Languages, Saul Rosen ed.

[RR64] Randell, B. & L. J. Russell, Algol-60 Implementation

[SB67] Samelson, K. & F. L. Bauer, "Sequential Formula Translation" in Programming Systems and Languages, Saul Rosen ed.

[SO69] Spier, M. J. & E. I. Organick, "The MULTICS Interprocess Communication Facility", Proceedings of the Second Symposium on Operating Systems Principles Oct. 1969.

[Weg68] Wegner, P. Programming Languages, Information Structures, and Machine Organization, 1968

[Wil68] Wilkes, M. V. Time-Sharing Computer Systems Am. Elsevier 1968.

[Wu71a] Wulf, W. A., et. al. "A Software Laboratory" (preliminary report) C-MU, 1971

[Wu71b] Wulf, W. A., et. al. BLISS Reference Manual, C-MU, April 1971

INDEX