ASYNCHRONOUS COOPERATIVE MULTIPROCESSING WITHIN MULTICS

by

PRAKASH GURUNATH HEBALKAR

B.TECH(HON.S), Indian Institute of Technology, Bombay

(1966)

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREES OF

MASTER OF SCIENCE AND ELECTRICAL ENGINEER

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1968

Signature of Author_____
Department of Electrical Engineering, May 17, 1968

Certified by _____
Thesis Supervisor

Accepted by_____
Chairman, Departmental Committee on Graduate Students

ASYNCHRONOUS COOPERATIVE MULTIPROCESSING WITHIN MULTICS

by

PRAKASH GURUNATH HEBALKAR

Submitted to the Department of Electrical Engineering on May 17, 1968 in partial fulfillment of the requirements for the Degrees of Master of Science and Electrical Engineer

ABSTRACT

Existing computing systems do not permit cooperative multiprocessing, i.e. computations consisting of several concurrently operating processes which use shared data-bases actively. When all data references are fully synchronised to avoid conflict, such computation is possible; however such synchronisation is nearly impossible in existing environments.With asynchronous operation of the processes, output-functionality i.e. the property that the results of a computation are reproducible, becomes the chief consideration and also the chief problem with current systems.

Several models for output-functional cooperative multiprocessing have been proposed. One of these has been chosen and a sub-system for MULTICS is proposed that ensures output-functionality of multiprocess computations. The operation of processes on segments is controlled, using the access checking mechanism, so that conflicts are avoided. Procedures for use in this control of access and for the creation and deletion of processes in a convenient manner are described. The procedures are defined in sufficient detail to permit relatively easy coding.

The sub-system proposed ensures that all computations performed within that environment will have reproducible results under minimal restrictions. Multiprocessing in the sub-system requires fairly substantial computation on the part of each process for efficiency, as the access control is implemented in software.

THESIS SUPERVISOR: Jack B. Dennis

TITLE: Associate Professor of Electrical Engineering

ACKNOWLEDGEMENT

The author is indebted to Professor J. B. Dennis for suggesting the topic treated herein as well as for several fruitful discussions and valuable suggestions. Several MULTICS system programmer's deserve credit for making the mystery that is MULTICS understandable. The author would also like to thank Carla Marceau, Suhas Patil and Donald Slutz for reading through the manuscript at various stages and giving their comments. Barbara Mutnick did a fine job of typing the thesis.

Work reported herein was supported, in part, by Project MAC, an M.I.T. Research Program sponsored by the Advanced Projects Agency, Department of Defense, under Office of Naval Research Contract No. Nonr-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government.

TABLE OF CONTENTS

Chapter 1

The concept of multiprocessing marks a major advance in contemporary ideas on computing. By multiprocessing, as the term is used here, is meant the "simultaneous" execution of several processes in parallel. It includes simple time sharing, although the execution of several processes is not strictly simultaneous there, because several processes are in different stages of completion at any instant with the specific processes involved also changing dynamically--a situation analogous to that where several processes are actually running simultaneously. The term is thus used in a broad sense and does not necessarily imply the actual use of several processors.

Even the earlier time-sharing systems had several advantages to offer and these in fact changed the entire attitude of people towards computer usage. Some principal advantages of time-sharing are the reduction in investment and operating cost to the small user while retaining the availability of a powerful facility at all times, the reduction in response time from hours to minutes and the associated advantage of (essentially) interacting with the computer. This interactive use increased the possibilities of computer usage considerably, both in terms of uses technically feasible and in terms of user acceptability. Later systems provided facilities for on-line storage of files of programs for use as and when required, and for permitting users to obtain files of other users for their use.

The newest systems such as MULTICS[1] offer several new advantages. Thus, the use of segmentation in NULTICS provides the user with an extremely large address space. This, combined with the possibility of making inter-segment references using symbolic names, as well as several other lesser facilities permit much greater programming generality. Further, the multiprocessor configuration provides for increased reliability and a greater measure of continuity of ser-

vice. Several protection mechanisms are provided. The modular design permits easy and almost indefinite expansion of facilities.

There is, however, an important deficiency in all these systems. As interactive use of computers became popular, a question occured to several people viz, if two users could use the same machine "simultaneously", would it be possible for user programs to interact? Such interaction is very desirable both for different humans and for different devices to compute simultaneously on data or programs supplied by some other user. Provision of such a facility will open up the possibility of the use of computers in a whole new class of problems. Since the two different cooperating processes need not belong to different computations (or jobs or "users"), parallel execution of the processes of a single computation is also a possibility. The case where the procedures or data shared are unmodifiable is relatively easy to handle and many systems do permit this either by sharing the same physical code (as in MULTICS) or by copying programs (as in CTSS)[2]. The more interesting facility is, however, a planned sharing of modifiable sections. Here the only case permitted so far is a scheme where the individual processes are laboriously pre-checked for correct synchronisation and for prevention of errors on account of reading off of incorrect data (say) because of differences in execution times of the processes. Moreover even with such pre-checking there is no guarantee that synchronisation will occur, because there is a large number of unknowns as, for instance, in a multiprocessing environment. The case where processes run asynchronously and must still cooperate has not been implemented yet.

The principal problem when asynchronous processes cooperate is one of functionality, i.e. of ensuring that the sequence of outputs produced by a given computation depends only on the input-data and on the specification of the computation, not on non-intrinsic (and perhaps unknown) factors such as the relative speeds of execution of the various parts of the computation. It is easily shown that functionality is ensured if determinacy is ensured; deter-

minacy is the property of a computation whereby the sequence of values of any variable (or element in the address space) is completely and uniquely defined by the input data and the specification of the computation (variations in other factors affecting the computation notwithstanding).

Several models have been proposed for determinate asynchronous multiprocessing. Van Horn's model [3] is one such model that also seems relatively easy to implement in existing systems. For this reason this model is picked for implementation here. MULTICS provides a suitable basis for such an implementation because it provides several useful facilities in varying measures such as segmentation, protection mechanisms, facilities for creation of several processes in a computation, facilities for inter-process communication, etc. The goal is thus to describe a sub-system operating within the framework of MULTICS that permits determinate asynchronous cooperative multiprocessing, use being made of Van Horn's model to guide the implementation.

The chapters that follow describe the model and the implementation. Chapter 2 describes briefly Van Horn's model, which he terms Machines for Coordinated Multiprocessing, and relevant portions of MULTICS. Chapter 3 describes aspects of the implementation that relate to a situation where a number of processes and segments are in existence. Chapter 4 generalises to the case where processes and segments can be created and destroyed. The last chapter illustrates how this system might be used to solve problems.

Chapter 2

Part 1

MACHINES FOR COORDINATED MULTIPROCESSING

A Machine for Coordinated Multiprocessing (MCM) is an abstract device that is similar to common computing systems but also possesses additional desirable properties. It consists (cf. Fig 2.1) principally of cells, a Count Matrix (CM) and a Scheduler.

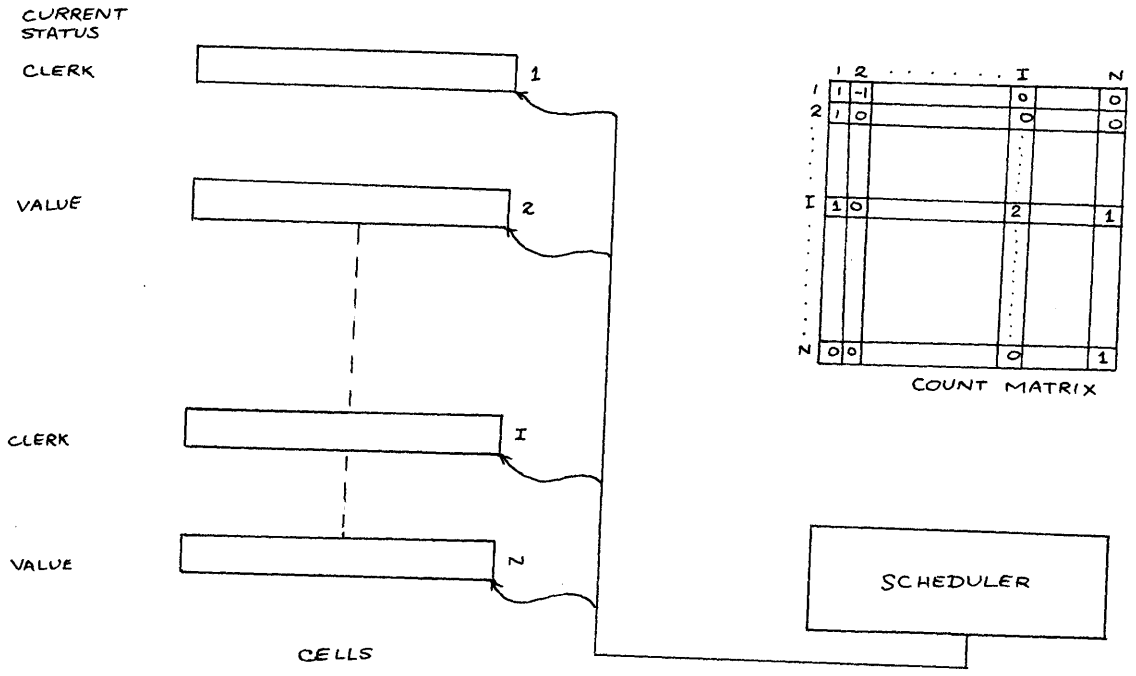
2.1 Cells

Cells, which are identified by names, can perform two types of functions. A cell that only stores a value passively is a value cell; on the other hand, a clerk cell is active and can perform operations. Whether a cell is a clerk cell or a value cell at any instant is determined by the contents of the CM. It is therefore possible for a cell to change its function with time. The number of clerk cells increases and/or decreases with time; the total number of cells in the system is, however, fixed (say N).

Certain cells are permanently designated output cells. When an output cell is written into, the value is both stored and put out. There is thus a one-to-one correspondence between output cells and the output streams of an MCM.

2.2 Transactions

The basic operations performed by a clerk cell are called transactions. There are five types of transactions. Two of these are the get and put transactions. The transaction get which is "get of i replace f(•)", causes replacement of the contents of cell x (the clerk cell performing the get) with f(c(i)) where c(•) is the content function for cells. The content of a cell is called a word. A put transaction is "put of i with v replace w" and it causes the



MCM Essentials

Figure 2.1

clerk cell to hold the word w and cell i to hold the word v. The particular f(•), v or w to be used by a cell is determined by its transaction table which lists the transactions to be performed corresponding to the contents of the cell (it is meaningful only when the cell is a clerk cell).

In the descriptions of get and put, i is the operand name, v is the operand word, f(•) is the replacement function and w is the replacement word. In the put transaction, if i = x, v is ignored. The use of get and put is illustrated by the instruction STO Y which stores the accumulator into location Y. It can be mimicked by the following instructions (x is the clerk cell and value cell K contains the STO Y instruction):

```
get of x replace f (•)
put of y with A replace Z.
```

Here the initial contents of x are equivalent in ordinary machine terms to Instruction Register- (IR), Location Counter- (LC), Instruction Cycle- and Address Register- information. LC contains "k". The function f(•) is such that f(x) is the put instruction (found in k) and the equivalents of incremented LC information and Execution Cycle information. Cell A corresponds to the Accumulator and Z is the word which will cause x to "fetch the next instruction" in the usual sense, i.e. Z = C(x) with cycle information modified to indicate an instruction cycle.

The above example also serves to point out the need for a clerk cell to possess certain capabilities. For instance, when cell x performs a transaction it needs to read its own contents to determine what transaction to perform. It also needs to write into itself the new word that specifies what transaction it is to perform next. Again, to perform a get of i (put of i) it needs to read from (write into) cell i. One of the restrictions on the activity of clerk cells is that they need capability or permission to perform any reading or writing. For this reason and because a clerk cell needs to write into itself

as an essential part of its activity, clerk cells are defined to be precisely those cells that have write capability for themselves. The next section describes the Count Matrix which is an important adjunct of the capability concept.

2.3 Capabilities and the Count Matrix

The Count Matrix is a table that defines the capabilities of each cell for every cell in the MCM. It is an NXN array, each element of which is an integer and is called a count. If the count at position (x,i) is greater than zero, cell x is said to have read capability for cell i. If, in addition, cell x is the only cell having read capability for cell i, it is said to posses write capability for cell i. Thus, if the only entry in the nth column of CM that is positive (greater than zero) is the nth one, then n is a clerk cell.

It was said earlier that a cell can change function with time. In terms of the Count Matrix this implies that some means of altering CM entries must exist. These means are provided by the remaining three transactions a cell can perform, viz, the send, done and bye transactions.

"Send of i to e replace w" is the form of the send transaction. The parameter "e" is called the sendee name; "i" is the operand name and "w" the replacement word. When a cell n performs such a transaction it increments the count at (e,i) and causes itself to hold w. The sendee may not be the clerk call itself. In order to perform a send, cell x must have write capability for itself and read capability for cell i.

"Done of i replace w" decreases the count at (n,i) by one and causes the clerk cell performing the done, i.e. n, to hold w. A cell need only be a clerk cell to perform a "done". A bye transaction is "bye to e replace w". When cell n performs a bye it simultaneously decrements the count at (n,n) and increments the count at (e,n); it then causes cell n to hold w. It is clear that cell n must have write capability for itself (i.e. be a clerk cell) in order

perform a bye. Bye serves to change the status (or function) of a clerk cell n to value-status.

Some salient features of the capability mechanism are, then, that a cell can neither give itself a capability nor be deprived of its capability by another cell. Further, by means of send's and done's a value cell can be made a clerk cell while done's, send's and bye's make it possible for a clerk cell to become a value cell.

2.4 The Computation State

The computation state of an MCM is defined by the information contained in the MCM's cells and count matrix. The computation state changes at discrete time instants when transactions take place. Transactions occur in zero time. The MCM thus starts out with an initial computation state, defined by the initial contents of the N cells and the CM, and enters successive computation states after each instant. The computation state at the instant a transaction is taking place is undefined. The computation being performed by an MCM is complete when the computation state reaches a steady state value (i.e. no transactions are possible).

2.5 The Scheduler

The computation state changes because of transactions taking place at well defined instants of time. At these instants the scheduler sends out go pulses to a subset of the set of all clerk cells. The cells in this subset then perform their transactions, after which the MCM is in a new state. It is the job of the scheduler to select this subset of clerkcells using the information in the CM and a set of rules.

A clerk cell is said to be enabled when it possesses all the capabilities it requires to perform the transaction it would perform if it received a go

pulse. The set of enabled cells constitutes the enable set. The choice collection is a subset of the power set of the enable set, this subset satisfying the condition that no element in it contains the names of two (or more) cells that would modify the same element of the CM. The scheduler picks some one element (a set) of the choice collection and sends simultaneous go-pulses to the elements of this set at the next instant. It then re-constructs the enable set and choice collection for use at the next instant. The only other restriction on the scheduler is that it be reasonable, i.e. each enabled clerk cell must receive a go-pulse within a finite time of its becoming enabled.

The above rules imposed on the scheduler's choice ensure that simultaneously performed transactions do not conflict, i.e. during a change of computation state

1. each cell is written into by no more than one clerk cell;
2. each cell that is both read from and written into is written by the same clerk cell that reads the cell and
3. each element of the Count Matrix is altered by no more than one clerk cell.

2.6 Determinacy of MCM's

An MCM is well-defined if and only if during no computation performed by the MCM is an evaluation of a transaction table or a replacement function ever attempted with an argument for which no value is defined. Van Horn has shown that the computation performed by a well-defined MCM is always determinate and, therefore, functional. This is precisely why a computation performed in the proposed sub-system, which models an MCM, is determinate.

Part 2

MULTICS

This part of the chapter gives a brief description of MULTICS. The reader is referred to reference [4] for further details. MULTICS is the acronym for Multiplexed Information and Computing System. As its name implies, it is a computing system for general use. A user is one who is using the system to perform a computation. The activity of a processor in executing a sequence of instructions constitutes a process. More precisely, "A process is a locus of control within an instruction sequence" [5]. It is thus an abstraction of the activity of a processor. Consequently, it is possible for a process to exist (albeit suspended) at an instant when no physical processor is carrying out the activities of the process (as for instance in time-sharing). A computation is defined by a set of instruction sequences and data bases. It can be performed by one process or by several processes acting concurrently. MULTICS is a multiprocessing system i.e. several processes can be (and usually are) in existence at any instant. It also happens to be a multi-processor system. It permits multiprocess- or concurrent- computation, i.e. a user's computation can require concurrent activity by several processes on the data bases.

The space in which all the objects (information) defining a computation (i.e. instructions and data) reside or appear is the address space of the user. Thus all objects can be referred to by their location in the address space. The activities of a process consist of the execution of instructions in the address space using data objects from the space. The address space is thus an abstraction which is to physical memory what a process is to the activity of a physical processor. With the concepts of a process and address space explained, the description of the components can be taken up.

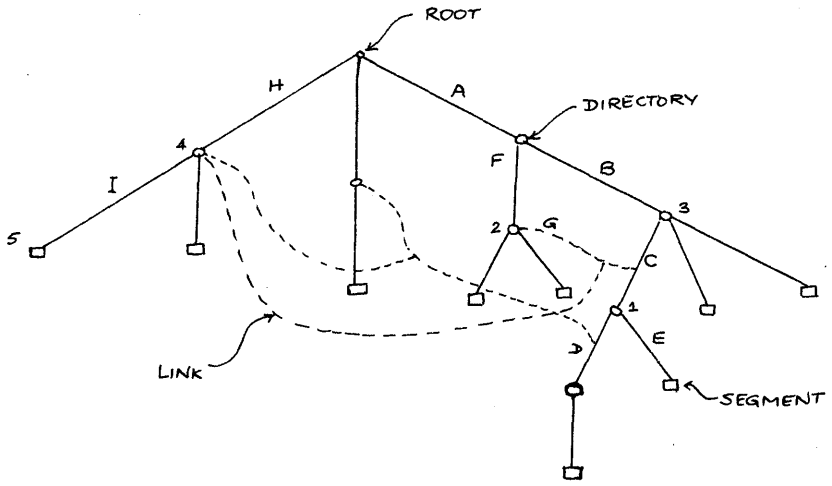
2.7 Organisation of Information

An important aspect of the address space seen by a MULTICS user is that this space is composed of segments i.e. distinguishable and identifiable pieces of linearly organised data, each piece being potentially infinite in length (actually a segment can be only as long as 256 K words because of physical restrictions on memory size). Thus, any object in the address space has a two dimensional address, viz a segment name and a relative location in the segment. In secondary storage segments are what are usually called files. There is thus a one-to-one correspondence between files in secondary storage and segments. One of the functions of the file system is to make the physical location of a piece of data irrelevant to the user i.e., a data object is always referred to as if it were a segment, it being the responsibility of the file system to bring the object into active (core) memory if it is not there already. All the data objects appear in one common heirarchical structure of nodes and branches called the directory heirarchy (cf. Fig. 2.2). All segments are terminal nodes in this tree-like structure and vice versa. Intermediate nodes and the root of this heirarchy are directories. In this sense the structure is a tree with leaves which are segments and nodes which are directories.* Directories and segments appear at the end of branches emanating from other (parent) directories (the root directory has no parent).

There are also links between pairs of intermediate nodes. The structure is thus a network rather than a pure tree. Any object in the heirarchy is identified by a tree-name i.e. a sequence of names of branches and/or links leading to the object from the root node. The tree-name of a directory is also called a path-name. The tree-name of a node is the concatenation of the path-

---

* As all information appears as segments, directories are also segments, but they will not be so termed in order to avoid confusing terminology.



Directory Structure

Figure 2.2

(Adapted from [4])

name of the directory in which the file resides and the entry-name of the entry (branch) in that directory which points to the node, i.e. path-name> entry-name where ">" indicates concatenation (eg. A > B or A > B > C > E).

A branch in a directory points to a data-object (either a directory or a segment). In addition, associated with every branch is an Access-Control-List (ACL) which is a list of user-id's (user-identification's) and associated access modes for the object pointed to by the branch. There are five possible access types, an access mode being a meaningful combination of these. The types are "WATER" i.e. write (W), append (A), trap (T), execute (E) and read (R). All except T indicate the kind of access or use permitted. "T" indicates that control is to trap to a procedure that returns a mode consisting of some substring of REWA. The mode given in the ACL is the apparent mode. This is also the effective mode when T is absent. When T is present in the apparent mode, the mode returned by the trap procedure is the effective mode.

Various primitives to make/delete entries in the data structure, to change access rights, to add/delete links exist. These will be indicated in the following discussion as the need arises.

Two directories in the whole heirarchy are particularly interesting from the point of view of the user. These are the working directory and the process directory. It should be noted that these names are conceptual and relate to usefulness; they are not actual (entry-or path-) names of directories. Every process has its own distinct process directory. Several processes may share the same working directory, however. Ordinarily, all the segments a user creates and uses (other than MULTICS segments) appear in one of the following three directories: the working directory, the process directory and the MULTICS command-and-subroutine library. A process is given a fresh process directory when it is created, and cannot change its own process directory: The process directory is destroyed with the process. The working directory is also

assigned at creation time, but can be changed by the user by means of a MULTICS primitive. Ordinarily, a user should have no use for this primitive. The nebulous nature of the working-directory- and process-directory- concepts will lessen in section 2.8.

A final fact about the information structure relates to copying of segments. When a non-directory entry is created in a directory, the user who created it can specify whether all users of this entry should share a unique version (segment) or get separate copies for their use. In the latter case, when a process asks for (a segment corresponding to) the entry, file system routines copy the entry into a segment in the process directory of this process and this is the segment the process uses.

2.8 Use of the directory hierarchy

At first it appears that use of a data-object requires that its tree name be known in order to locate it in the heirarchy. That is indeed the way to make access to elements of the structure. However, it is possible to leave some of the work to system routines, making it possible for a process to demand an object by merely specifying its entry name (the last component of the tree name). The job of finding the correct object in the heirarchy is that of the Search Module.

The search module takes a proferred name and looks in different directories for an object with that name; when an object is found, the tree-name of that object is returned as the (complete) identifier of the object. The sequence of directories to be searched is indicated by search rules which can be specified by the user. When no rules are specified by a user, MULTICS provides a set of rules, a characteristic of which is that the process directory is the first searched with the working directory next. In this sense segments which are "private" to a process are found before those with the same name elsewhere.

From the description of the search module given above, it is clear that, as the same entry name can appear in several directories, the segment sought, when an entry name is used, need not be the same as the segment found by the search module. The user must, therefore, have a means of ensuring that the correct segment is found. One way, of course, would be for the user to use the tree-name of the segment. However, the tree-name is long and cannot be used in the address part of an instruction if only because of the inconvenience. The solution MULTICS provides is to permit the user to associate specific segments with procedure segments so that when control is in that procedure segment those pre-associated segments are found when just entry-names are used. This process of association is called relating. The segments are daughters of the (mother) procedure segment. The means whereby these relations are stored and remembered are described in the following paragraphs.

When a segment is related to a procedure segment a relationship segment is created in the directory heirarchy. This segment contains the actual path name of the mother segment and for each daughter, among other things, the call name*, the tree-name, a global usage switch and a create switch. The global usage switch indicates whether the daughter segment is available to users other than the creator as well. The create switch indicates whether this daughter segment exists or is to be created. That a segment is a relationship segment for another procedure segment is indicated by a switch on the branch pointing to the former segment. The system arranges that a reference to the procedure results in the relationship segment being found. The information in this segment is used to set up information in the Segment Name Table, as described below, and to bring up the procedure segment itself. The relationship segment is thus seen to be a repository of association information.

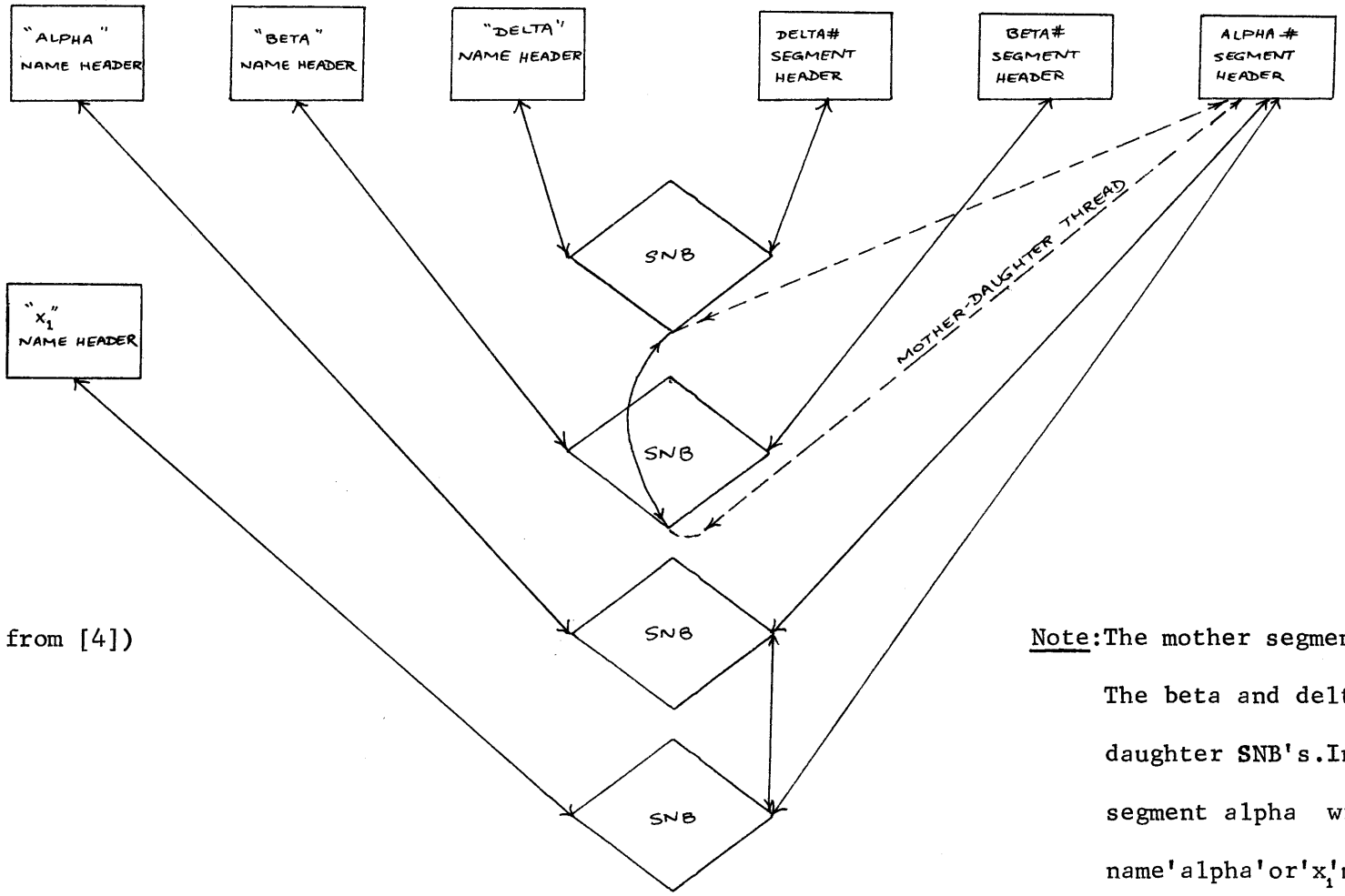
---

*A segment can be assigned any number of (call-) names for purposes of calls, by the use of the "initiate" primitive. The name of a segment need not be a call name. Thus "call alpha$alpha" may call a procedure in segment beta.

The use of a procedure using related segments is illustrated by the following example: Let a user decide to relate segments tall-talk and snob to the procedure "do-nothing" which is in segment alpha. Then suppose the name of the relationship segment as returned by the relate command is beta. The user then refers to the procedure as "beta$do-nothing" and not as "alpha$do-nothing". In fact neither this user nor another need be aware (for calling purposes) of what segment (alpha) the procedure resides in or that beta is a relationship segment. It appears as if it is in segment beta and in fact the creator gives another user a link (or access) to beta and not alpha. This conceptual inelegance is a result of the peculiarity of the relation mechanism in MULTICS.

The Segment Name Table (SNT) is a table listing associations of symbolic names and segment numbers (segments are identified by numbers in the hardware). Every process has its own SNT as a segment in its process directory. This table serves as a quick means of converting symbolic names to addresses during the process of linking which is described in section 2.10.

The basic element of the SNT is a Segment Name Block (SNB). There are three threads linking SNB's (cf. Fig. 2.3). One set of threads links all SNB's associated with the same symbolic name. There is thus a set of two-directional pointers linking the Name Header, which contains the symbolic name, and the associated SNB's. The Name Header contains a pointer to the last SNB created in its list, in addition to other things. There is a similar thread linking all SNB's associated with the same segment number (it will be recalled that a segment may have several call names). Thus, there is a Segment Header containing the segment number, tree-name, unique-id, etc. of this segment and in each SNB a pair of pointers constituting the link. The third thread lists all daughter SNB's. It consists of one pointer in each SNB linking SNB's for daughter names and an entry in each SNB indicating the segment number of the mother segment (cf. the concept of relationships explained above). An SNB



(Adapted from [4])

Note: The mother segment is alpha The beta and delta SNB's are daughter SNB's. Initiating the segment alpha with either the name 'alpha' or 'x_1' makes 'beta' and 'delta' known calling names.

Representation of SNT Entries

Figure 2.3

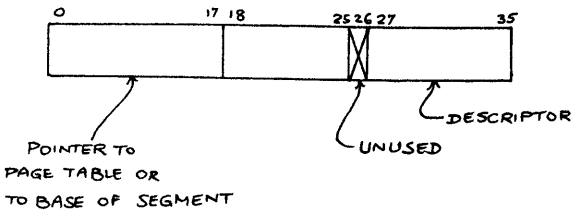
also contains a ring number (cf. Section 2.11). It therefore indicates an association of a symbolic name, a segment number and a ring. A global switch in every SNB is interpreted as indicating availability to procedures other than the mother.

Segment-and Name-Tables in the SNT contain pointers to the segment- and name-headers respectively. They are used to make access to the information in the SNT.

2.9 The Mapping to Physical Memory

It was mentioned in section 2.7 that the virtual memory (or address space) of a user appears segmented, with two dimensional addresses. Since the physical memory is limited in size and since different users may use (share) the same segment while knowing it by different names, it is necessary to provide a suitable mapping from virtual memory to physical memory. This mapping is defined by the Descriptor Segment of a process. This is a table that contains pointers to physical blocks of memory (frequently to page tables as many segments are paged--this paging, however, is of no concern to the user) that correspond to segments in the address space. An index in this table, the segment number, is as good an identification of the segment in the address space of this process as its name. The emphasis on the last two words of the previous sentence is because each process has its own descriptor segment and, because of the method of construction of this table (cf. Section 2.10), two processes can have different segment numbers for the same segment.

An entry in the descriptor segment of a process is called a descriptor word (cf. Fig. 2.4). A descriptor word contains a pointer, as mentioned above, a descriptor and the length of the segment (this information is used to detect out-of-bounds references and is, therefore, set at the current length or maximum permissible length, depending on whether the user is not or is per-



Descriptor Word

Figure 2.4

mitted to append to the current contents of the segment). The descriptor contains paging information and the user's access mode for this segment (cf. section 2.7).

The above description does not indicate the mapping from segment name to segment number. Once an entry in the SNT is created, it provides this mapping. The construction of this table is described in section 2.10.

All memory references in a process are in the form segment number|offset. The addressing mechanism uses the first part of such an address, as an index in the descriptor segment to get the pointer to the base of the segment in core; this pointer together with the offset, defines the location intended.

Now, because of dynamic allocation of the memory (the address space seen by a user is much larger than the available physical memory) a segment may be transferred to secondary storage. In this event the corresponding descriptor word for every process is marked with a (missing-) segment fault indicator so that any attempt to refer to this segment initiates activity to load the segment into core memory. However, in such a case, the only information available about such a segment is its segment number (this is available in the machine conditions stored at the instant of the fault). More information is required to relate this to an object in the directory heirarchy and to its location in secondary storage. This information is provided by the Known Segment Table (KST) and the Active Segment Table (AST).

All segments are sharable because every process uses the (one and only) directory heirarchy. It is improper to have several copies of a segment in core. As a result, several segment numbers (in different processes, of course) really refer to the same physical entity. The mapping from segment number to physical location in secondary storage is therefore split up into a (per-process) mapping from segment number to a (system-wide) unique identification for the segment and one from the latter to a location in secondary storage. The

KST provides the first mapping and the AST the second one.

The KST contains entries indexed by segment number. An entry contains the unique identifier for this segment (this is unique throughout the system and over several years), the user's effective mode, the protection list (cf. section 2.11), a list of symbolic names for this segment (if it is a directory segment), the segment number of the directory segment in which this segment is an entry and the index of this segment in that directory, the date and time the branch was last modified, etc. The date and time when the branch was last modified is compared with the corresponding item in the AST to determine possible changes in access privileges of this process (made by another process). If the two items agree the KST access information is used for insertion into the descriptor, else this information is recomputed. The information about the parent directory is required to recover segments that have been deactivated (i.e. entries for these segments in the AST have been removed to make room for other entries). It is sometimes necessary to be able to refer to entries in the KST by the unique identifier and, for this reason, a hash coding scheme is used based on the unique identifier.

The AST is indexed by unique identifier. An AST entry contains much information; only a part of this will be indicated. One item is the maximum segment length. Others are the page table address, the date and time when the branch was last modified, a pointer to the location of the file in secondary storage, a loaded switch, a process trailer pointer and the unique identifier for the segment. All the terms except the process trailer and loaded switch have been explained already. The process trailers are threaded into a list and are entries that indicate how to make access to the descriptor segment word (sdw) for this segment in all the processes using this segment. This information is required when the access control information for a segment is changed as segment fault indicators have to be inserted in all these

sdw's to ensure that descriptors are reevaluated. A process trailer is removed when for instance the process makes the segment unknown (i.e. removes the KST entry). All process trailers are deleted when a segment is unloaded from core. The loaded switch indicates when a segment may be unloaded. When there is no space for additional entries in the AST, the entry for the least used segment is removed.

A segment is known when a KST entry for it exists, otherwise it is unknown. Similarly a segment is active if an AST entry for it exists, otherwise it is inactive. Also, it is loaded if it has a page table in core. A segment can be known and inactive or active and unknown (to a process but known to some other process). A loaded segment is necessarily active.

If a segment fault occurs, the KST is consulted to get the unique identifier for the segment. This is used to look for an AST entry. If an entry is found, the page table address and maximum length information can be used directly, access control information being checked for time of modification as indicated earlier. If no AST entry exists, the AST entry for the parent directory is sought. The KST is used to get the call name for the immediately superior directory if no AST entry for it is found. This directory can then be sought using the Search Module, etc., and the original segment can be activated. This chain can continue but must terminate as the root directory is always known.

2.10 External References

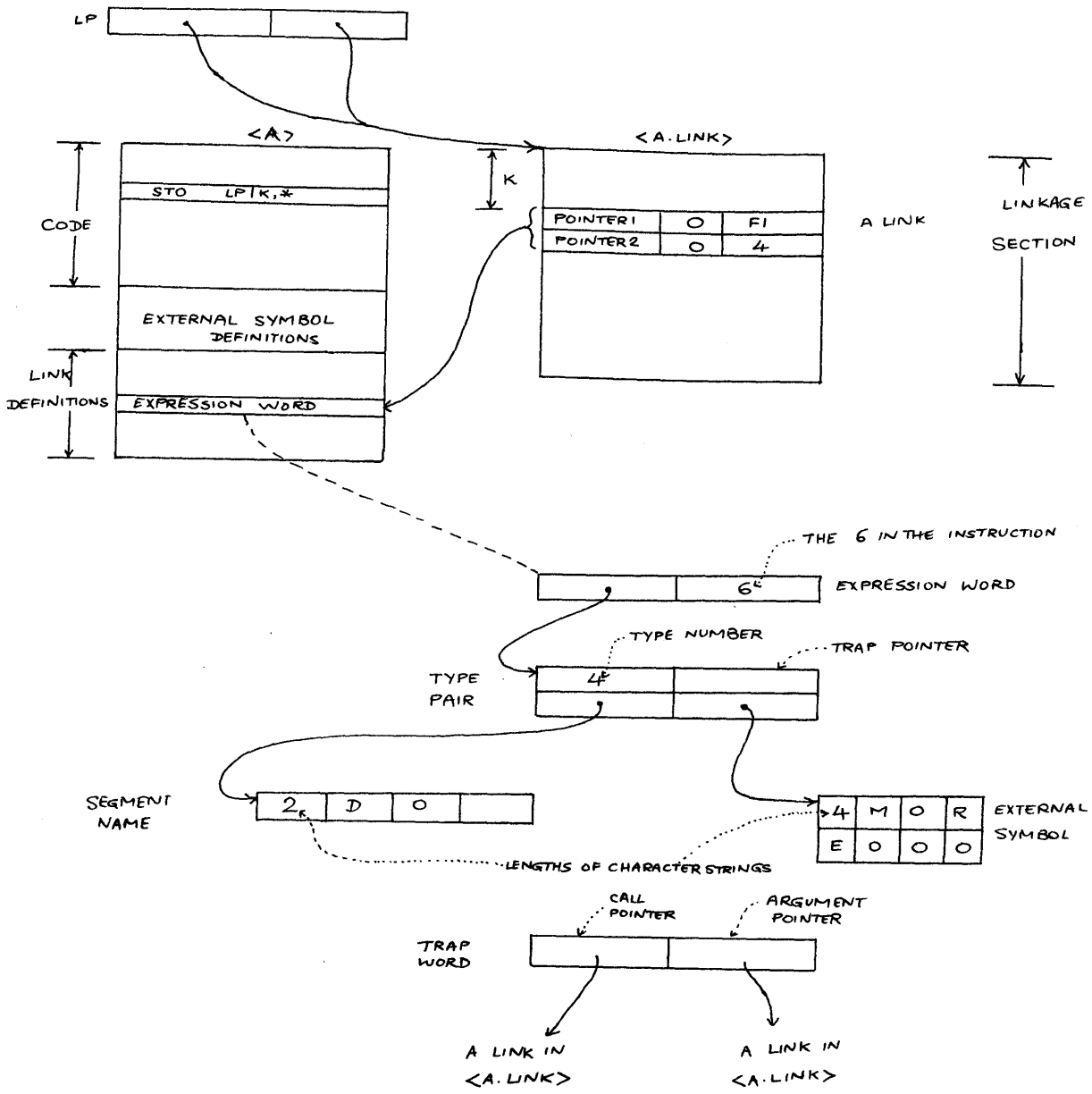
The preceding section described how the mapping from segment number to physical memory takes place. The construction of the SNT entries which map segment names to segment numbers will be described here. The broader problem is to translate references such as alpha|income to segment number|offset. This operation is called linking.

There are two types of references to an external segment (a segment other than the procedure segment) that an instruction can make. One is the "itb" or indirect to base reference. In this case a base register number (one of eight of which only two are available to the user) and an offset are supplied. The contents of the base register (presumably set up correctly earlier) are used as an index to the descriptor segment and addressing takes place as usual. The its or indirect to segment reference uses a pair of words. One word contains a segment number and the other a numeric offset. The reference takes place as usual.

An external reference almost always is compiled as an "its" instruction. The pair of words in this case is called a link. All the links are put into the linkage section which is in the linkage segment for the procedure segment. The linkage segment for a procedure "do" is called do-link. It is clear that at compilation time it is impossible to know the segment number for a segment. The compiled form of an instruction is shown in Figure 2.5.

In Fig. 2.5 lp is the link pointer pair of base registers that points (by segment number and offset) to the linkage section for the current procedure, "a", The "lp|k,*" indicates "indirect through the word at relative location k with respect to the word pointed to by lp). At this location is found a two word link. The "fi" tag is a "fault indicator" which results in transfer of control to a routine called the "Linker" when the link is first used. The Linker does the job of converting the link to the form shown in Fig. 2.6 where "do#" is the segment number for the segment "do". The "4" in the link and type pair is an addressing type which is of no interest here.

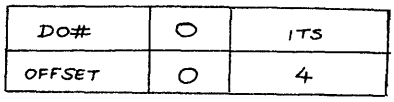
The Linker uses the pointers in the link to fetch the expression word which points to the type pair. In case the trap pointer is non-null, a trap-before-link situation is indicated and the Linker uses the pointers in



Compiled or raw form of a link and associated information for

'sto do|more+6'

Figure 2.5



Completed link for 'sto do|more+6'

Figure 2.6

the trap word to construct and issue a call to a user-specified procedure. Upon return the Linker proceeds to complete the link. It gets the symbolic segment name and calls the Segment Management Module (an interface with the File System) primitive "smm$get-segment" with this symbolic name and a pointer to the procedure segment that caused the linkage fault.

The primitive "smm$get-segment" searches the SNT for an entry which is related (or if no such entry exists, a global entry) to the procedure segment and in the same ring. If such an entry exists it immediately yields the segment number of the correct segment (which is in its segment header) and this is returned to the Linker. When no such entry exists, "smm$get-segment" calls the Search Module to get the tree-name and unique identifier of the segment. It then calls the primitive "estbl-seg" to return a pointer (or equivalently a segment number) to the segment. "Estbl-seg" itself uses the KST to get a segment number for the unique identifier (this is why the hash coding of the KST by unique-identifier is necessary) if an entry exists. If no such entry exists "estbl-seg" has to call the primitive "make-known" with the tree-name of the segment as argument. "Make-known" establishes a KST entry for the segment using the branch information in the directory heirarchy and the next available segment number. Thus "smm$get-segment" ultimately gets a segment number for the name. It uses this information to construct an SNT entry and returns the segment number to the Linker.

Some additional work is required when the segment for which "smm$get-segment" is asked to give a segment number is a relationship segment (this information is returned by the Search Module along with the unique identifier and path name). In fact in this case the routine makes known (by a call to "make-known") and reads the segment found. It makes a complete entry for the procedure segment in the SNT as before. It also puts in SNB's for the re-

lated (call) names found in the relationship segment but does not thread them to segment headers. It then makes the relationship segment unknown (by a call to "make-unknown") and returns. The remaining threading (to segment headers) for daughter SNB's will be done when those call names are actually used. In that event, the SNB's give the tree name of the correct segment (it will be recalled that SNB's contain tree names) so that the aim of relating is fulfilled (as the tree name is not obtained by calling the Search Module).

The Linker, it will be recalled, has now obtained a segment number. The offset part of the link is obtained from the expression word and the external symbol definitions. The link is complete and the Linker returns. The indirect reference now proceeds. It is possible, however, that this is the first reference to this segment in this process. In this case it should be clear from the previous discussion that the correct descriptor word for this segment has not yet been set up. By convention every word in a descriptor segment is initially set up with a segment-fault indicator (a template descriptor segment is used) for all segments. As a result, the first use results in a missing segment fault and ultimately a call to "segfault" which sets up the descriptor word.

Segfault uses the KST to get the unique identifier for the segment number for which a fault arose. The unique identifier indexes the AST to yield an entry which gives the page table address (a page table may have to be set up in this process). The KST entry also yields the segment length and access control information for this segment. Segfault can now put in all this information into the descriptor word.

One noticeable aspect of all the operations described above is that no activity takes place unless explicitly demanded. This is a basic principle in MULTICS.

Passing reference to the Associative Memory is in order. This is a sixteen word memory which is used in a last-used-first-out manner to contain the sixteen most used descriptor segment words and the associated segment numbers (actually also Page Table Words but this may be ignored here). This memory saves memory references to the descriptor segment, page table, etc., and thus speeds up the addressing mechanism when a segment is used frequently.

2.11 Protection

The existence of only one information structure (viz the directory heirarchy) in the whole system ensures that all its elements are available to all users. This sharing has, however, to be controlled to ensure that privacy is guaranteed where required. This is the function of the protection mechanism. Part of the mechamism is in the access control list and in descriptors. This part ensures accessibility of segments in an appropriate mode to each user. However, no distinction is made between procedures attempting such access. The latter kind of protection is provided by the ring structure.

The ring structure defines a classification of all the segments in the system. There are sixty-four rings and they define a heirarchy of protection with inner rings representing rings of greater protection. The ring-numbers are assigned outwards from the core beginning with zero. Associated with every segment is an ordered set of three ring numbers called the protection list. The three numbers must be in non-decreasing order. The first pair defines the access bracket while the second pair defines the call bracket.

A segment is accessible to all procedure segments in the access bracket or in lower rings with the mode specified by the owner of the segment. A procedure segment cannot be read or written but can be called at pre-specified entry points (called gates) by procedures in the call bracket.

From the above description it is clear that the ring protection mechanism affords intra-user protection in addition to inter-user protection. Sensitive and vital information and procedures are put into lower rings. Multics system routines, for instance, are assigned to rings 0 to 31 and user procedures can only be in rings 32 to 63 so that the system is protected from the user. Moreover, a finer sub-division of the protection is made with most sensitive procedures in ring 0 (the hard-core ring), less sensitive procedures in ring 1 (the administrative ring), etc.

The state vector of a process at any instant includes the ring number of the currently executing procedure. This number is used to validate all references (as for instance in looking for SNT entries). A ticklish situation arises with inward calls as a request may be made to make access to data not in the caller's domain of access but in that of the called procedure. In this case all arguments are validated by the "gate-keeper" which checks the legitimacy of the call. In the case of outward calls arguments must be copied into the outer ring for availability there. This discussion glosses over implementational complications as they are not of importance here. (For instance no outward calls or input-output activity from ring 0 are permitted so as to ensure that ring 0 is entered only for a bounded time -- this is not true of other rings).

2.12 Processes

It will be recalled that a computation consists of one or more processes operating on the information structure. A process can have one of three states. A process is running if a processor is currently allocated to it. It is ready if it is merely awaiting allocation of a processor as soon as one becomes available(it is on the ready list). A process is blocked or idle if it cannot be allocated a processor unless some actions (such as Input-Output activity) take place. Clearly means must exist to change the state of process. These are

provided by the primitives "block", "wakeup", and "quit". The first primitive makes the process using it blocked. The wakeup primitive makes a blocked process ready. Quit makes a ready or running process blocked (as soon as any masks on interrupts are removed).

In very broad terms, a scheduler picks up processes on the ready list and allocates processors to them as processors get freed. The switching of a processor to a different process is done by setting up the state vector for the new process. For instance, in order to ensure that the correct address space is set up, the descriptor segment must be loaded and the descriptor base register set to point to it. Several other segments such as the KST, etc.,have to be loaded too. A process is made ready by putting it on the ready list.

Each process is identified by an identifier (process-id) which is unique throughout the system and over several years (it is obtained by reading a clock, say). Processes are further grouped into process-groups with their process-group-id's.

This grouping is for accounting and administrative purposes. It also corresponds to a computation because all user-created processes are associated with the process group of the creating process. In this sense a process group also corresponds to the computation initiated at a console. Processes in a group typically share certain attributes and objects such as a working directory.

The creation of a process consists of setting up a few basic segments a process requires (such as the KST, the process directory, etc.), initialising them and creating entries in certain tables (such as the working directory table which specifies the working directories of processes). Two primitives, "create-wp" and "delete-wp", permit a user process to create and delete processes. The acronym "wp" stands for "working process" and is a term used for all processes of a user's computation. Processes which perform only system work are called system processes.

A process consists of the execution of a sequence of procedures. All Multics procedures are pure i.e. they do not modify themselves. In order to allow recursive execution of pure procedures, a stack is used for allocation of variables and to record arguments, return addresses, etc. The stack is also useful for implementation of the block structure of PL/I. Recursivity of procedures is a desirable property and it is essential for the proper operation of Multics (for example "segfault" in trying to get fresh access information on branches might refer to a directory segment which might be missing and thus cause a recursive call to segfault).

The base register pair "sp" (stack pointer) points to the current frame of the stack while "ap" (argument pointer) points to the base of the argument list. As lp, sp and ap are dedicated base register pairs, only the remaining one is available to a user.

2.13 Interprocess Communication

Some means of communication between processes is required for multi-process computation. The Interprocess Communication Module which consists of the Event Channel Manager and Wait Coordinator modules provides this communication by controlled use of the "block" and "wakeup" primitives.

The basic element of the communication mechanism is an event channel in a shared segment called the Event Channel Table. A process expecting to receive communication sets up an event channel and conveys the channel name and the process-id to the process which will send the message (e.g., putting this information in some prearranged spot). The process anticipating a message can then wait for an event to be signalled on the channel. This is the basic principle of communication. Some of the details are given below.

Event channels are of two sending types viz device-signal channels and communication channels. The former are connected with input-output devices

and interrupts and will be of no significance here. Channels also have two receiving types viz event-wait-and event-call- types. In the case of the first type, channels have to be explicitly interrogated, whereas the second type of channel causes a prespecified procedure of the receiving process to be called when an event is signalled over it. Further channels can be of event-count or event-queue mode. In the case of the former a counter is incremented every time an event is signalled on the channel. In the case of the latter additional information identifying each event (event-id's) and the senders' process-id's are supplied; read-out of the channel thus yields a sequence (queue) of event-id's instead the event-id of the first event signalled since the channel was reset (as with event-count mode).

The primitives "ecm$create-ev-chn" and "ecm$delete-ev-chn" serve to create and delete event channels. It is possible to make a channel accessible to other process groups by explicitly calling certain primitives. The two primitives above are available in all rings but a channel is accessible to other process groups only in ring 0. The primitives "ecm$set-ev" and "ipgecm$set-ev" put event-id's (anything-clock times, say) into channels to signal events to processes in the same and in other process groups respectively. The rings of availability are as above. The primitive "wc$wait" permits a user process to block awaiting signalling of an event on a specified channel by a process belonging to the same process group.

2.14 Miscellaneous Aspects of Multics

A very brief reference to fault handling and input-output activity is in order. The latter is quite straightforward and not of particular significance to the system being designed. The reader is referred to reference [4] for details. Faults are converted into conditions (in the PL/I sense) as soon as possible and treated by condition handlers.

Because segments are shared, races in access to segments can arise and so can incongruities (eg., one process reading from a data-object while another is writing into it). For this purpose Multics provides a facility to lock data objects. Four words are used as a lock. The primitive to lock an object is "locker$wait". When the locker is invoked (in process A say) it creates an event-queue-mode channel (having an event channel name "a" say). It then attempts to set the lock word (the first of the four) using its own process-id for a locking value (the lock is set if the lock word is non-zero). If the attempt is successful (the lock was not set earlier) it puts "a" into "x.channel" (the remaining three words) and returns to the caller. If the attempt is unsuccessful it reads an event channel name (say "b") out of x.channel. It then sends process B (which has locked "x") an event signal over event channel "b" using "a" as an indicator and returns. The primitive "locker$reset" unlocks an object. It compares process-id for the current process with that in the lock word. An error return is made if there is a mismatch. Otherwise, it resets the lock structure. It then reads event channel "b" and using the event indicator read ("a") as an event channel name signals process A (as the object has been unlocked). It does this until channel b is exhausted, deletes event channel "b" and returns.

Locking in the basic file system is slightly different and permits several processes to read an object while a process attempting to write-lock the object causes inhibition of requests to read and a wait for the number of current readers to drop to zero before the object is locked.

Part 3

ADAPTATION OF THE MCM MODEL

The reader has seen MCM's and a typical large multiprocessing system as represented by Multics. It is clear that if the aim is to implement a subsystem for determinate multiprocessing, an interpretation of the principles of MCM's in the context of systems is necessary. Such an interpretation is given here.

Because they can perform operations, processes and clerk cells are analogous. Similarly data-units such as segments are the elements in systems that correspond to value cells. Transactions alter the contents of cells and correspond to the operations performed by instructions. Transaction tables correspond to the instruction repertoire. There is a difference, however, as there is no correlation between the number of processes and that of segments in the system while the total number of cells in an MCM is fixed and clerk cells can become value cells or vice versa. However, it seems clear that the aspect of MCM's that is of consequence is the discipline. As long as this discipline is followed determinacy is guaranteed. The non-convertibility of processes and data-units is thus irrelevant; for, processes and data-units can be created and destroyed just as the number of clerk and value cells can change.

Again, the part of the count matrix which displays the capabilities of value cells for other cells serves only (while the cells are value cells) to detect change of status of these cells. As processes and data-units are distinguishable in the system, the equivalent of the count matrix need only display the capabilities of processes for data-units and the bye transaction is unnecessary.

The aim, then, is to discipline the interaction of processes and data-units using the CM and the other rules governing MCM behaviour, the results proved for MCM's ensuring that computations performed in such a framework are determinate.

Chapter 3

The case where a number of processes and data units are in existence and their numbers do not change with time will be considered in this chapter. For the purpose of this discussion it will be assumed that these processes and data units have been created somehow and are in existence. The chief implementational problem in this case is that of permitting controlled access to the data units and of permitting transfer of capabilities for these data units.

To ensure determinacy it is necessary to impose strict control on all the data units in the processes. However, it will be seen later that the checking of capabilities consumes some time. In the case of MULTICS routines and data units which will be used very often, such control of access is a heavy drain on system* resources. Again, if MULTICS were non-determinate (actually non-functional), even an ordinary one-process computation would have non-determinate results, thus making MULTICS useless! It seems reasonable, therefore, to assume that such is not the case and that control of access by MULTICS procedures to MULTICS data bases is unnecessary. Consequently, only those data bases which the user creates need be guarded; in other words only those data units in rings relatively outer to a certain ring (r_{min}) need be guarded.

3.1 The Unit of Access Control

Before the mechanism of checking access is examined, the data unit which is to be used for this purpose must be determined. This unit should ideally be a word of the address space. However, the obvious inefficiency and large com-

*The term "system" will be used throughout the sequel to mean the system proposed here.

putational demands of such a scheme make it impracticable. For example, the count matrix would have to have a large number of columns, and a large number of sends and dones would become necessary. The unit has, therefore, to be larger than one word. It could be a page of virtual memory, but the number of pages is still too large to permit efficient system operation. The obvious choice is, therefore, a segment. This choice is also suggested by the fact that MULTICS itself checks access control information on a per segment basis only. Moreover, a strong motivation is that the segment is logically the next larger data unit after the single word.

3.2 The Mechanism of Access Checking

The chief problem of access checking is that of preventing all references to a segment* that are of a type not permissible according to Van Horn's rules. That is, it is necessary to prevent say read references to a segment when the process concerned lacks read capability for that segment. Since all user procedures in the system will be required to be pure, the only accesses to be controlled are those to external segments i.e., to segments other than the procedure segment** itself. This last fact suggests two possible schemes for access checking which are described before.

The first scheme utilises the fact that references to external segments occur via links in the linkage section of that procedure segment. It will be recalled (from Chapter 2) that links are originally raw and set up so that the first attempt to use a link results in a fault to the linker. This first scheme uses the trap-before-link feature of the linking mechanism to trap control to a system procedure which picks up the symbolic segment name using the pointer in the still raw link. The procedure then consults the Count Matrix (CM) to

---

*The discussion concerns only segments because other data objects such as scalar quantities cannot be shared between processes (except when explicitly passed as arguments). This is a MULTICS restriction.

** a MULTICS convention requires all procedures to be in one segment only.

determine the capability of the current process for the segment. Clearly the kind of reference sought to be made must be known for the procedure to take appropriate action. This is done by requiring the compiler to mark all write-references by a non-zero value in the middle field of the second world of the pair of words constituting the link (cf. Fig. 2.6). The system procedure thus knows the type of reference sought and the type permitted by the contents of the CM. If the two types differ, the system procedure sets up an event-signal channel and calls the MULTICS primitive "wc$wait". The process is thus blocked (because it lacks the capability it seeks) until it receives the capability sought. If the process possesses the capability it seeks, the system procedure returns, permitting linking as usual. Subsequent references continue unhindered until the process loses its capability.

The other part of this scheme concerns the action taken when a process loses its capability. Every time the process calls the "send" or "done" system primitives, the primitives check whether write-or read- capabilities for a segment are lost. In either case they call a system routine that puts directed-fault 5 tags in place of "its" in every link of the corresponding type (i.e., read or write reference) that has the segment number of this segment in it. The system procedure called upon acceptance of the directed fault 5 checks CM for the Van Horn capability and either, replaces the "directed-fault 5" tag by "its", or sets up an event-signal channel and calls the MULTICS wait coordinator, depending on whether the process posseses or lacks the capability sought respectively.

The scheme described above suffers from a major defect, for a user can dispose of the old linkage section and get a new one or change his present linkage section. What is worse is that a reference to an external segment need not use a linkage section at all; rather a process can get pointers directly from the MULTICS primitive "generate-ptr". This defect thus precludes

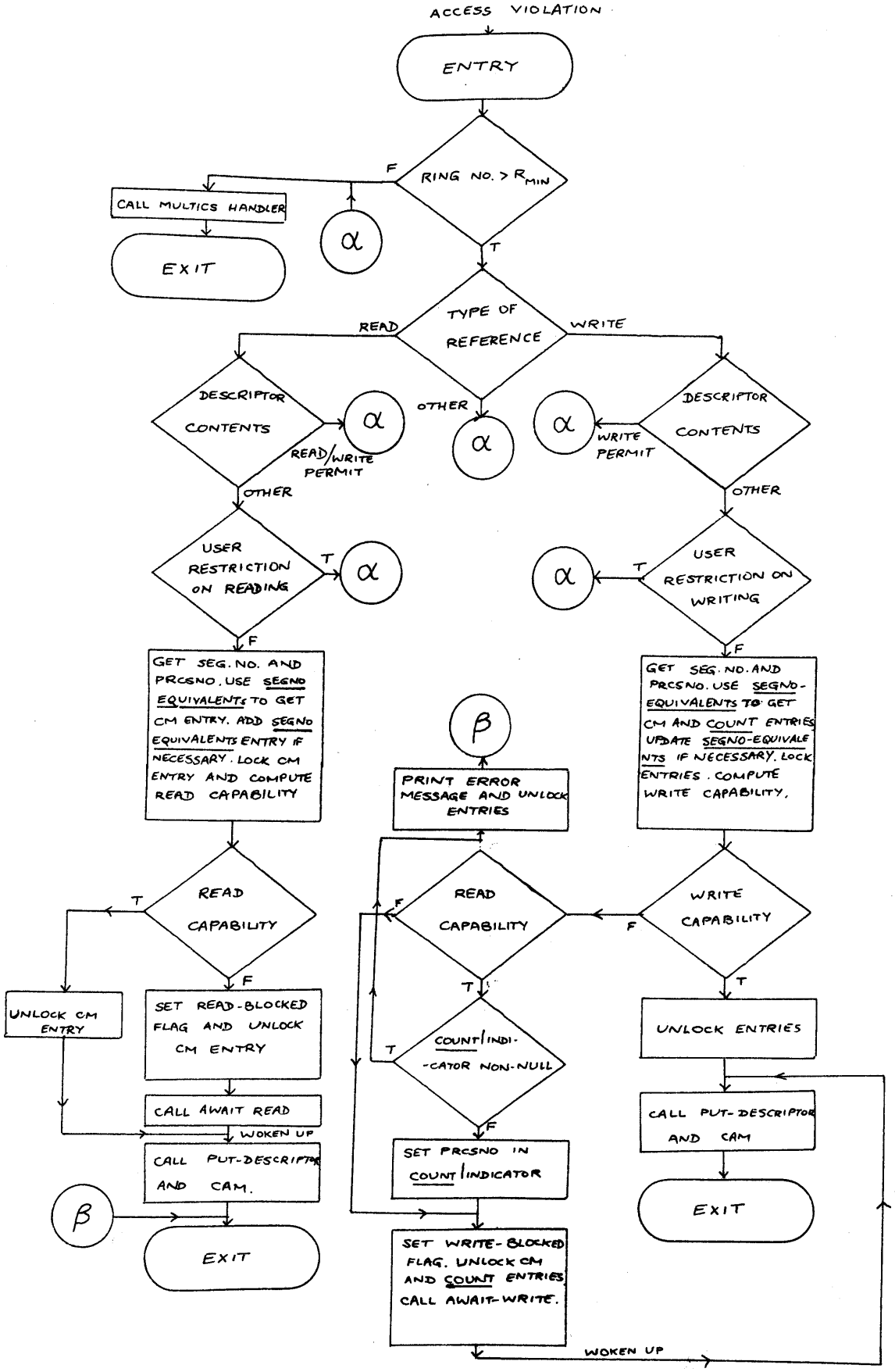
use of this scheme. The other scheme, which is also adopted for use, will now be described in some detail.

The reader will recall that except when the address of the page table (or the page itself) is found during readout of the associative memory, all references to external segments take place via the descriptor segment. Also, the descriptor word for a segment has a descriptor field which specifies access restrictions, if any, for this process. This field, and the associated fault-handler therefore, serve as an excellent way to check access. Thus, suppose all fields are initially set to "directed fault 5*". Then, the first time a segment is used, control traps to a system routine (cf. Fig. 3.1) called "Access-Checker" (AC for short) which checks whether the process has read or write capability for this segment by consulting the CM. If the process has read capability only, AC sets bit 35 of the descriptor "on", indicating read capability only. If the process has write capability it obviously has read capability too and both these are indicated by turning "on" bits 30 and 35. If the process lacks either capability it calls the system routine "await-read" in which it stays blocked as explained later.

Consider the situation where the segment descriptor of a process indicates permission only to read while the process tries to write; a hardware "illegal procedure fault-access violation sub-condition" results. The fault handler signals the Condition (in the PL/I sense) known as Access Violation. Suppose it is arranged that the procedure called upon occurrence of an Access Violation Condition is the system procedure AC. AC can determine if the ring number of the segment is less than r_{min}. If it is, AC calls the routine Multics handler. Otherwise, AC checks the descriptor field to determine what permission is

---

*This will be changed later (in section 3.5).



Access Checker
Figure3.1

indicated by the descriptor. It then determines* if the fault is due to an attempt to write (read) into the segment, and if so, it gets the path name of the segment and checks the CM for write (read) capability. If such capability is present, AC sets the descriptor field to indicate this and returns. If not, AC calls the system primitive "await-write" ("await-read") in which the process blocks. If, however, the descriptor showed permission to write (read), a hardware malfunction has occurred, and so AC calls the regular MULTICS handler for the Access Violation Condition.

Clearly, just as a mechanism for issuing permission is necessary, so too is a means of denying it. When a process calls the system primitives "send" and "done" which implement the corresponding Van Horn primitives, these routines also check to see if write capability or read capability, respectively, are lost**. When a capability is lost, they call the ring 0 system procedure "put-descriptor" to change the descriptor for that segment in the descriptor segment for this process to indicate read-permission or no permission according as the process has only read capability or no capability at all, respectively.

From the preceding three paragraphs it is clear that all references to data segments occuring through the descriptor segment of a process occur only if the process has the capability (in the Van Horn sense) demanded. It was mentioned that only pure procedures are to be used by the user. An Access Violation Condition for a procedure segment is therefore genuine and so is passed on to the regular MULTICS handler.

As regards references using data from the associative memory, one may note that since the descriptor field is picked up from the corresponding des-

---

*It is possible to determine if the instruction causing the fault was attempting to write into the segment from the snapshot of machine conditions stored in the process-data segment at the instant of fault.

**The details of how this is done are explained in sections 3.3 and 3.4.

criptor segment word, the only matter of concern is loss of a capability. This situation is taken care of by instituting calls to the MULTICS ring 0 abaolute mode procedure "cam" to clear the associative memory when a mere restrictive descriptor is inserted into a descriptor segment word.

3.3 The Count Matrix

The reader will recall that in MCM's the Count Matrix displays the capabilities of a cell for all cells in the MCM. The Count Matrix has a potentially indefinite extent in either direction when one translates the concept in terms of processes and segments (cf. Part III, Ch. 2). For, with processes creating processes, (cf. Chapter 4) the number of processes in the system has no upper bound (barring physical restrictions) and the same holds for segments, which too can be created.

It is convenient to handle one dimension by using segments, which naturally have a length that is (theoretically) infinite. The matrix would thus be a collection of segments. The question as yet unanswered is whether a segment should correspond to a row or to a column. In this context one observes that a process usually uses several segments so that at any instant the Count Matrix has many more columns than rows. A scheme in which segments implement columns will therefore suffer from the defect that there would be a very large number of segments each with very few entries. For reasons of efficient use of storage, therefore, a row wise implementation of the CM seems more desirable. It would appear that since determination of write capability requires examination of all the entries in a column, a one segment per column is convenient. However, it is possible, as is shown later, to determine write capability by examining just two entries instead of m entries (where m is the current number of processes). Toward this end a count of the number of positive entries in a column is stored in a separate segment.

The Count Matrix (cf. Fig. 3.2) is implemented as m + 1 segments where,

again, m is the current number of processes in the system. There is one segment for each row of Van Horn's Count Matrix, i.e. for each process, its name being prcsno-row* where prcsno is the number that identifies the process (not the MULTICS process-id) as explained in Chapter 4. The extra segment called count contains counts of processes with read capability for each segment.

Each entry of count is seven words long. The first four are required to lock the entry. The fifth contains the count and the sixth (the count-indicator) contains the process-id of a process blocked awaiting write capability but possessing read capability. The seventh (the created-switch) is a switch for CM initialisation. Each entry of prcsno-row also consists of seven words. Four are for locking of the entry, the fifth is the Van Horn count, the sixth is a flag indicating whether or not this process is blocked awaiting read capability for this segment (the read-blocked flag) and the seventh is a similar flag indicating blocking for lack of write capability (the write-blocked flag).

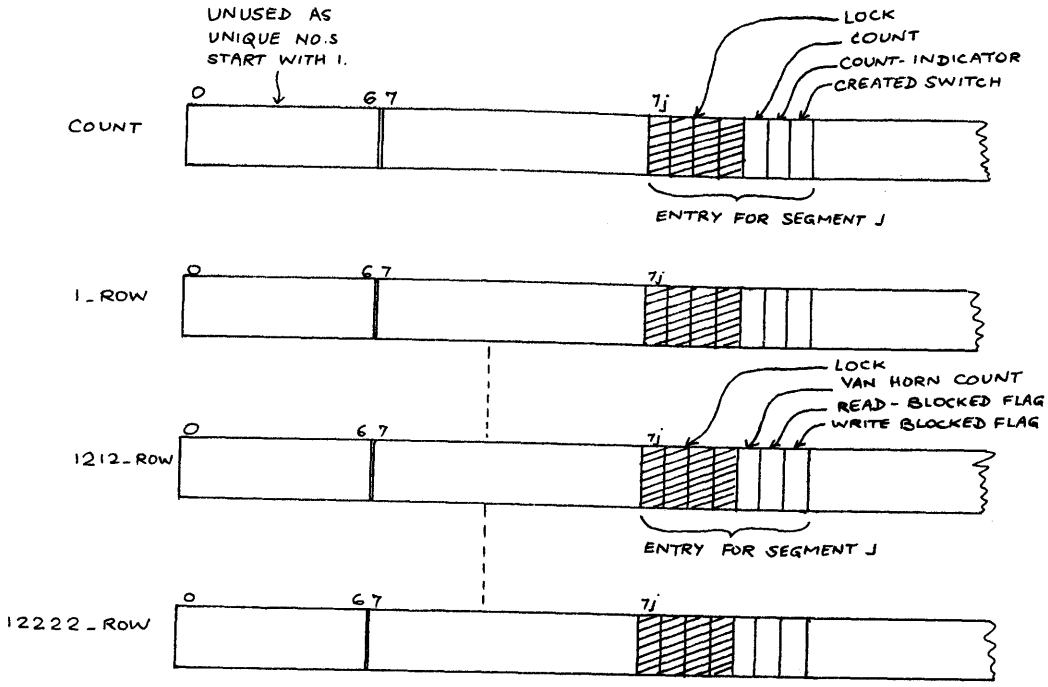
Clearly some indication must be left when a process lacks read capability and blocks. A good place is the corresponding CM entry because a "send" to that entry is the only action that could lead to revival of the process. This is why a CM entry contains a read-blocked flag. The write-blocked flag serves a similar function for the case where a process is blocked for lack of write capability.

Two indices are required in order to make access to an entry in the count matrix. Because of the way the segments are named, the identification of a process, prcsno,** serves as a convenient first index. The second index is

---

* Names of segments are underlined, those of procedures are in quotation marks.

** The user will not know the MULTICS process-id (which is created at process creation time) at the time the program is written. Prcsno provides a means of identifying processes (which is unique in a computation) even when a program is being written.



Structure of the Control Matrix

Figure 3.2

not so obvious, however. A segment is identified either by a symbolic name or a segment number. In either case the identification must be converted to a numeric offset within the segment prcsno-row. The segment number for a given segment is, in general, different for different processes and, moreover, unknown at the time the program is written. The user can, therefore, specify only a symbolic segment name which must be converted to an appropriate numeric offset, the second index.

The conversion of a symbolic segment name to numeric index can be done fairly easily. A system routine ("send", "done", etc.,) takes the symbolic segment name and gets the corresponding unique-id from the directory heirarchy using the MULTICS primitive "getsegstatus". It then looks up a segment, the equivalence-table, containing equivalences between unique-ids and (unique) numbers. If an entry corresponding to this unique-id exists it uses the numeric value. Otherwise, it adds an entry consisting of the unique-id and the next highest unused multiple of 7 (since each entry in CM is 7 words long) beginning with 1 X 7. This table, then, provides the numeric offset to be used within a CM segment.

Now getting the unique-id and searching the table are both time consuming, so that to repeat those operations each time is a waste of resources. A means of avoiding this is therefore necessary. For instance a segment, the symbol table, which is unique for each process* (unlike the CM and equivalence table both of which are clearly shared by all processes), can be created when the process is created. This table lists symbolic segment names and corresponding numeric offsets to be used in CM segments. Only this table need be searched in subsequent references. In fact, even this effort can be saved by implementing the Symbol Table not as a segment but in the manner described a little later.

---

*It resides in the process directory.

A similar scheme is required for the use of the procedure AC which only gets segment numbers upon access-violation. A per process segment, segno-equivalents (cf. Fig. 3.3), containing the numeric index for the CM entry corresponding to each segment number, is therefore created and is maintained by AC. It is a table that is accessible by segment number and numeric index.

The segment symbol table is eliminated as follows: Every user procedure is required to have for each segment a declaration for an integer-type (in the ALGOL sense) variable with the name "segment name", where "segment name" is the symbolic name the user wishes to use for a segment. This declaration should be placed in a suitable block of the procedure, in most cases the outermost block. This variable corresponds to one entry of symbol table mentioned above. It is passed as an argument to the routines "send" and "done" which extract the value of this variable. It if is zero (i.e., undefined), the routine goes through the search described above. The numeric offset, the unique number, found is then made the value of the variable. Thus a unique number is always available and easily available for subsequent references.

In case a procedure is known to change a CM entry for a given segment only infrequently, the user may, if he so desires, omit the declaration for the integer variable. A symbolic segment name is then used as the argument in calls to "send" and "done". A call to "send" (cf. Fig. 3.4) is, thus:

call send(segnamevar, seg-name-char, prcsno, process-name)

where seg-name-var is the integer-type variable mentioned above

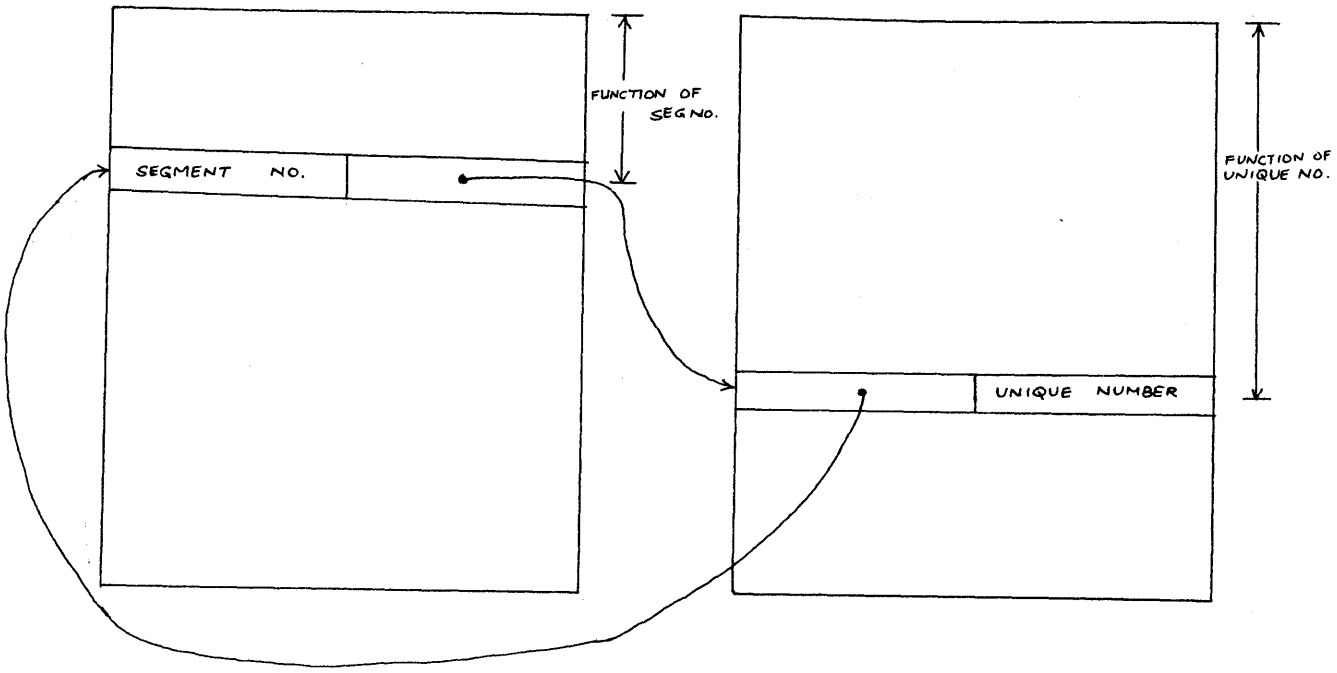
seg-name-char is the character-string symbolic segment name

prcsno and process name will be explained in Chapter 4.

Clearly only one of the first two arguments need be provided, the other is superfluous. Similarly, a call to "done" (cf. Fig. 3.5) is

call done (seg-name-var, seg-name-char)

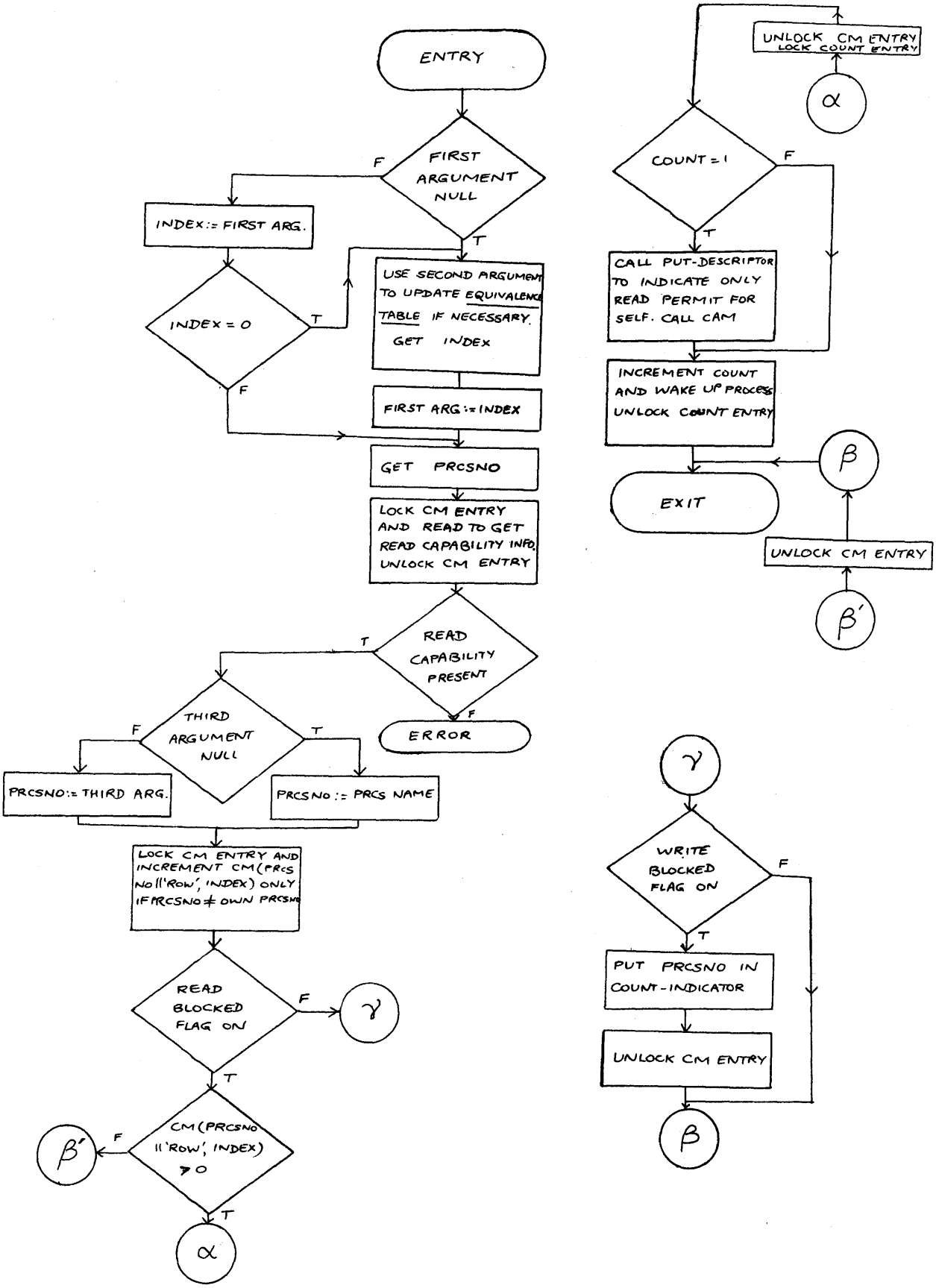
In this case the system routine "done" determines which row of CM to use by



The Segment segno-equivalents

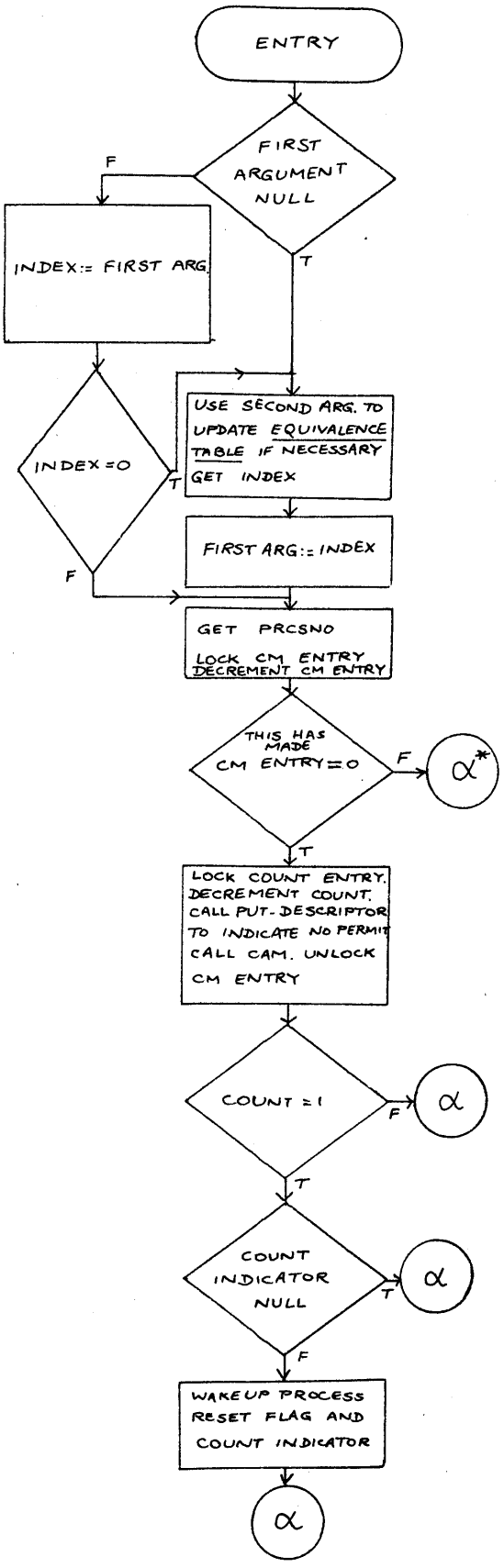
(Both tables are parts of one segment)

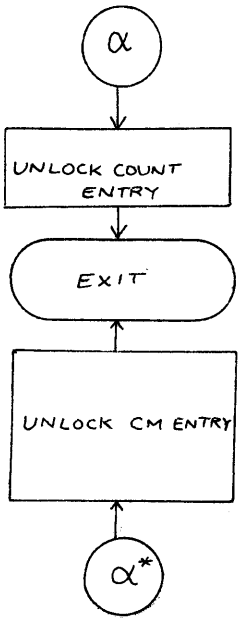
Figure 3.3



"Send"

Figure 3.4





"Done"

Figure 3,5

looking in the segno-equivalents segment for this process.

Sections 3.2 and 3.3 have described the mechanism of checking access according to Van Horn's rules. The details of just how processes are blocked awaiting receipt of capability and how they are revived will be given in the following section.

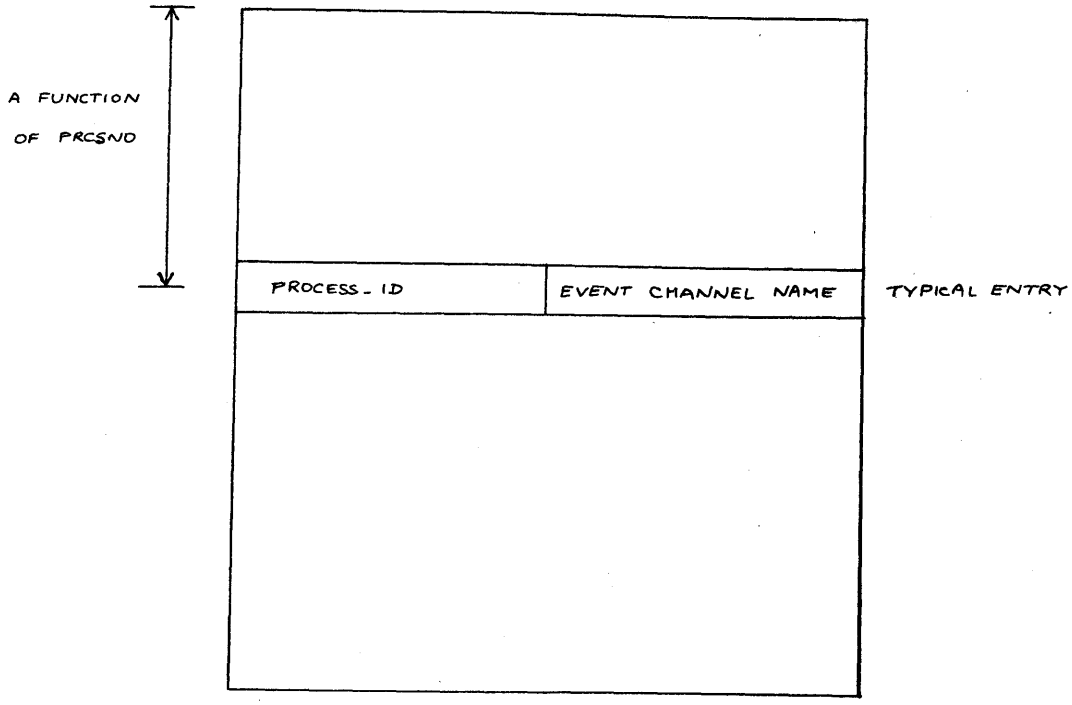
3.4 Signalling Considerations

It was indicated in Chapter 2 that inter-process communication in MULTICS occurs by use of the Interprocess Communication Module. For the kind of signalling sought, one event-channel per process is necessary. The event-channel has to be created by the receiving process. This channel creation can be done either at the time of process creation or at the time a process decides to await the arrival of capability. Since the creation occurs only once, it seems reasonable to designate it to the process-creation routine. In the following it will therefore be assumed that there is a communication channel of receiving type "event-wait channel" and mode "event-count-mode" for each process, and that the system wide segment channel table (cf. Fig. 3.6) contains the event channel name and process-id corresponding to each prcsno. Consider the system routines "await-read" and "await-write" mentioned in Section 3.2 (cf. Figs. 3.7, 3.8). Their main function is to indicate that the process is blocked for lack of a specific type of capability and to call the wait coordinator so as to wait on the event channel for this process.

```
call wc$wait(chn-list, ev-ind, sts)
```

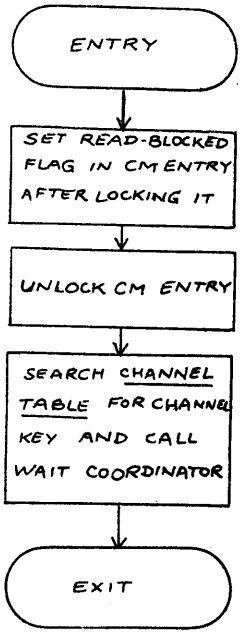
where	chn-list	is a single element list consisting of the event channel name
	ev-ind	is an indicator returned--it has no use here
	sts	is return status information

The next few paragraphs indicate where and how the two system primitives leave the indication.



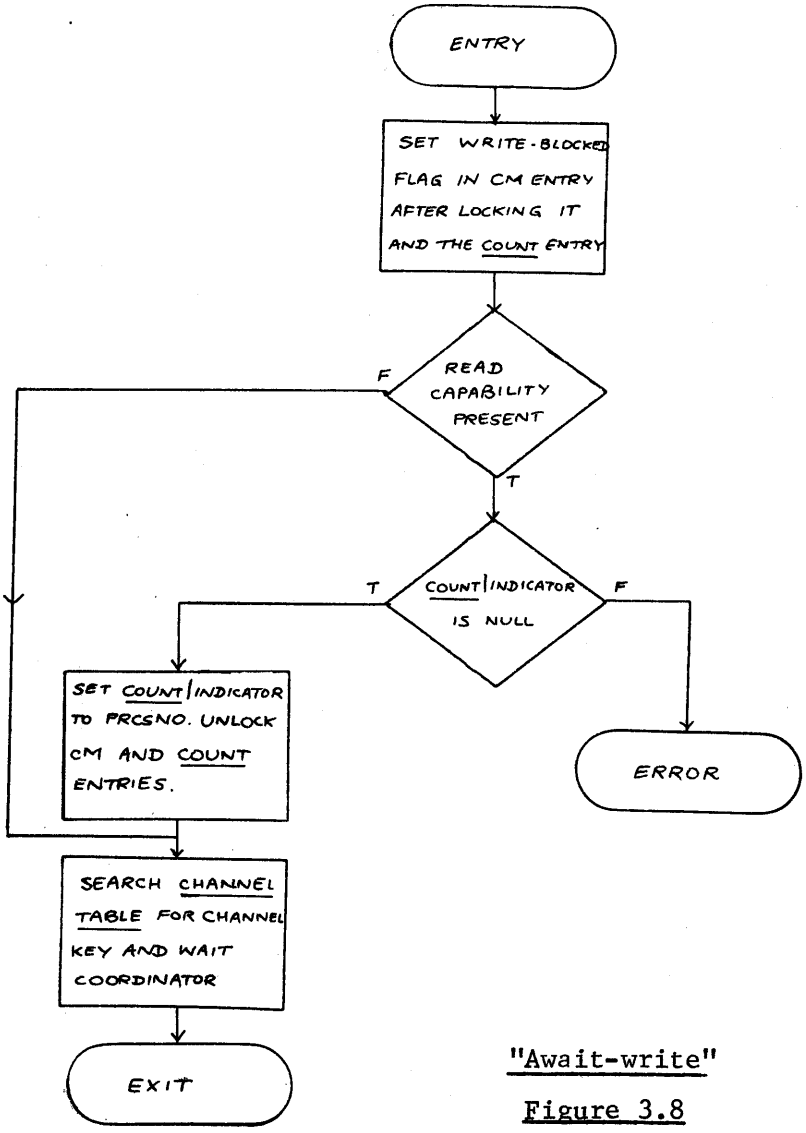
Channel Table

Figure 3.6



"Await-read"

Figure 3.7



"Await-write"

Figure 3.8

In the case of "await-read", the indication is made by setting the read-blocked flag in the appropriate CM entry. In the case of "await-write", though, it seems that the indication should not be left only in a CM entry; for a process can receive write capability by some other process giving up read capability for that segment, the CM entry for the former process not being referred to or altered at all. The information has thus to be "global" (in contra-distinction to the "local" nature of the "await-read" indication) at least in part. The count-indicator in count provides this part of the indication and the write-blocked flag, the rest of it.

The "send" and "done" procedures do more than just their Van Horn function. For instance, "send" checks the prcsno-row entry for the sending process and say segment j (i.e., the segment whose associated unique number is j). If the entry is greater than zero, it increments the Van Horn count in the corresponding entry for the sendee process provided the sendee process is not the process itself. In addition, if the incremented Van Horn count has become positive and if the read-blocked flag in the same entry is on, "send" resets the flag, looks up channel table for the channel key* for the sendee process and calls the MULTICS Event Channel Manager with

```
        call ecm$set-event (rec-prcs, ev-chn, ev-id, sts)
where   rec-prcs is the process-id of the receiving process
        ev-chn   is the channel name
        ev-id    is any identifier (say 11...1) for the event
        sts      is return status information
```

"Send" can also cause loss of write capability. It is therefore required to check the j^{th} entry in count whenever the incremented Van Horn count

---

*the combination of channel name and process-id

of the sendee process becomes positive. If this entry of count is one, the sender possessed write capability for segment j (prior to the send) which it has now lost. If the count is greater than one then no capability has been lost. "Send" increments the j^{th} entry of count to indicate that one more process has read capability for segment j. Further, if write capability was lost, "send" calls "put-descriptor" to reset the descriptor field of the appropriate entry in the descriptor segment of the sending process to indicate only a read permit (and clears the associative memory as mentioned earlier).

"Done" is required to decrement by one the CM entry for the current process and the segment concerned (segment j say). If thereby this CM entry becomes zero, "done" must reduce the count in the j^{th} entry of count by one, as this process has lost read capability for segment j, and reset the descriptor for this process by calling "put-descriptor" and "cam". If, further, that count becomes one, some process has gained write capability for segment j and it could be blocked in "await-write" awaiting write capability for this segment. "Done" must determine which process, if any, this is and revive it. It therefore looks up the count-indicator in the j^{th} entry of count which indicates if a process having read capability is blocked for lack of write capability for segment j, and the process-id of this process (a zero process-id means no such process exists). If count-indicator is non-zero, "done" consults channel table to get the channel key corresponding to this process. It then calls the Event Channel Manager with

```
call ecm$set-event(rec-prcs,ev-chn,ev-id,sts)
```

where the arguments are the same as before. The write-blocked flag in the CM entry of the revived process is then reset and the count-indicator for that segment is cleared.

The function of "await-write" in terms of leaving an indication is now obvious. First, it sets the write-blocked flag in the CM entry for the cur-

rent process and that segment (segment j say). Further if the current process posseses read capability for segment j, "await-write" examines count-indicator in the j^{th} entry in count. If that entry is zero, the presno of the current process is put in, and a call to "wc$wait" made as before. Suppose, however, that the entry is non-null. Then there is already a process blocked in "await-write" and it has read capability. The same is true of the current process. Clearly neither process can ever be able to proceed, i.e., the two process are in a "deadly embrace" [6] as a result of a programming error. Here then is the first situation encountered so far which requires special error indication. Such indication might be say, the typing out at a console of an error message with the prcsno's of the processes involved and the aborting of all processes in the computation.

The reader is reminded that it is perfectly legitimate for m-1 processes (where m is the current number of processes) to be blocked for lack of write capability, provided only one of them has read capability at a time. This does not necessarily guarantee that a process will not be blocked indefinitely as a result of incorrect programming; this situation will require separate error handling. Such a situation can arise if the count entry for the segment becomes zero while a process is still blocked, or because the only process with write capability for the segment is destroyed (cf. Chapter 4). The detection of the former case is part of the activity of "done".

It is possible for a process to gain read capability after it blocks for lack of write capability. Consequently, "send" (which gives this read capability) checks the write-blocked flag, and when it is set, examines count-indicator. If it is zero, "send" puts in the prscno of the write-blocked process. It can find count-indicator non-zero whereupon the error action mentioned above is taken.

The final versions of "send", "done", "await-read" and "await-write" that arise from the discussion above are those shown in figures 3.4 to 3.8.

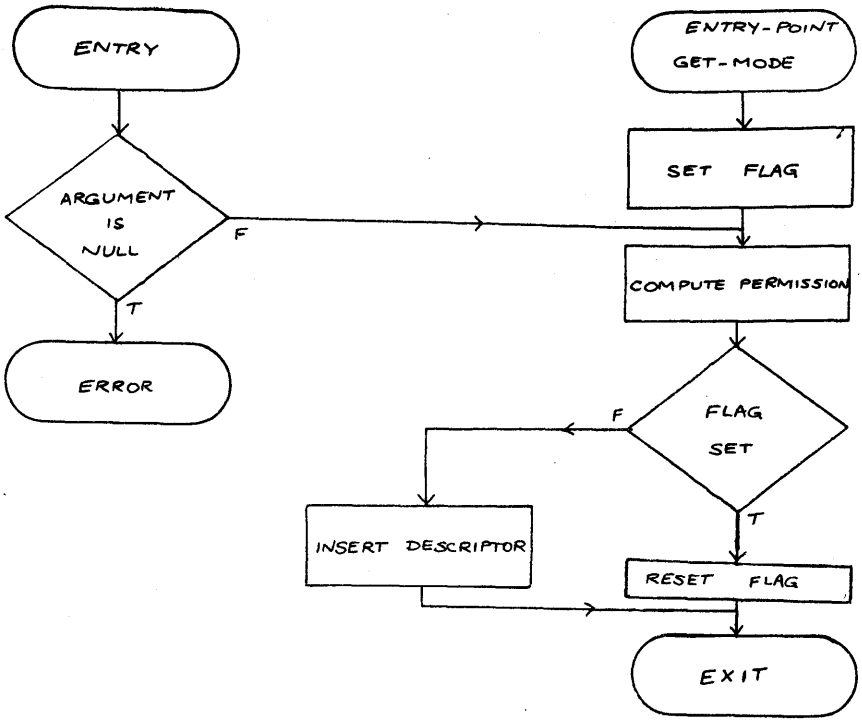
3.5 More Changes to MULTICS Routines

In Section 3.2 the condition-handler in MULTICS for the Access Violation Condition was changed to be AC so as to catch these faults and determine if they are genuine or simulated. Some more routines need changes too, but this discussion was postponed up to this section in order to present more impoortant aspects of the access-checking scheme. For instance, MULTICS "make-unknown" which deletes KST entries must be made to delete corresponding entries in the segment segno-equivalents of that process. For, the segment numbers of segments made unknown are reusable by other segments and therefore incorrect use of the pre-existing segno-equivalents entry for that segment number can occur later unless the entry is deleted. When a segment fault is taken, the handler calls the MULTICS primitive "seg-fault". One of the functions of "seg-fault" is to put a descriptor field into the descriptor word for that segment. This can be disastrous as the Van Horn capability has not been considered at all in this process! It is necessary to change "segfault" so that it calls the system primitive "put-descriptor" to do the job of putting in the descriptor field of the descriptor word. In this connection one must remember that the user-imposed MULTICS restrictions on access override any other constraints. The effective MULTICS mode for each segment must therefore be stored in a table, (the user restriction segment) indexed by unique number, which is used each time the descriptor field of any descriptor word is inserted. The most restrictive of the Van Horn capability and the MULTICS effective mode stored in the table, called the '"permission"', is what is put into the descriptor field. "Put-descriptor" takes one argument viz a segment number. It uses segno-equivalents to get the unique number for this segment and computes the capa-

bility of the current process for that segment. That together with the user restriction gives the permission, which is inserted in the descriptor. The user-restriction segment that was mentioned above has to be filled in. This could be done either by MULTICS "segfault" or MULTICS "makeknown". It turns out that MULTICS "makeknown" is better suited; for, several MULTICS routines (eg., "delete-segment") need to check the effective mode of a user with respect to a segment. This mode is of course meaningless unless modified by comparison with the Van Horn capability. The mode is obtained from the Known Segment Table (that obtained from the directory branch is used only to update Active Segment Table and Known Segment Table entries). It is necessary, therefore, to modify suitably the Known Segment Table (KST) entry. A good scheme seems to be to put in a TRAP mode for all entries. In this way control traps to the entry point "get-mode" (in put-descriptor) which returns the permission (cf. Fig. 3.9).

Now MULTICS "make-known" makes the KST entries and so it has to be modified to put in the TRAP mode in the new KST entry (except when the mode is only E i.e., "execute" or if the ring number > rmin) while putting the MULTICS mode in the user-restrictions-table. Similarly, when MULTICS "segfault" consults the AST entry to check for obsolescence of the KST entry, it must update the user-restrictions entry and not the KST entry.

MULTICS procedures like "delete-seg" read out the permission and if it is not of the kind sought, signal an error instead of blocking the process awaiting receipt of the capability. To eliminate this incorrect behavior, the MULTICS function "check-access" and primitive "hcs$check-access" must be changed to take one more argument viz, an expected permission, except when permission to execute is sought. Then, if the lack of permission results from a lack of Van Horn capability (this comparison is only made if the ring number is $\geq$ rmin), "check-access" calls "await-read" or "await-write" as appropriate. If, however, the lack of permission is because of user restrict-



"Put-descriptor"

Figure 3.9

ions, an error-status return is appropriate. The "append" mode is so returned if permitted by the user and if write capability exists.

Lastly, the primitives "delete-entry" and "change-name", which can be called by the user to delete a segment or to change the name of an entry in a directory, are two of four routines which directly call ring zero file system primitives to achieve their effect. The file system primitives check write access for the segment and the directory respectively, using the information in the directory heirarchy and not in the KST. Consequently, it is not the permission but the MULTICS access that is used. To overcome this problem these two routines, "append-branch" and "append-link" must be changed to call check-access before calling the file system primitives. "Truncate-seg" does not present such problems as it calls a routine which uses KST access information.

3.6 Protection of System Data Bases

It is obvious that a user could (intentionally, perhaps) change the contents of the system data bases (as, for example, the CM entries) or procedures and thus make the computation non-determinate. These data bases and procedures must therefore be protected from the user. In terms of MULTICS concepts, the data bases must lie in inner rings with respect to user procedures. They must have access brackets that end at or below r_{min} (as defined at the beginning of this chapter), and system procedures must be of the execute-only type. Furthermore, as the system routine "put-descriptor", has to make access to the descriptor segment, it must be in ring zero. The other procedures can be in any ring below ring r_{min}, preferably as far out as possible.

One aspect that was implicit in the discussion of the previous sections will now be made explicit. This is the prevention of races in the access to data. For instance when a process is reading a CM entry, another process must be prevented from changing it. This requires write-locking of that entry while a process is reading it. The mechanism for locking is provided by

MULTICS as described in Chapter 2. This is why each CM entry and count entry has four words for use of the locking mechanism. The duration of locking by the procedure AC will now be indicated.

In the case of AC, when it is trying to determine read capability, it refers to the CM entry. If the capability is missing, it will be recalled that it sets the "read-blocked" flag and calls "await read". Now, if another process should increment (and make positive) that CM entry between the time when the first process reads the CM and the time it sets the flag and calls "await-read", the blocked process misses the reviving action. The CM entry must, therefore be locked by AC before it is read and unlocked after the read-blocked flag is set.

When AC is testing for write-capability it uses a CM entry and a count-entry. AC must, therefore lock the CM entry and then read it. If the CM entry is negative or zero, it must get the write-blocked flag, unlock the CM entry and call "await-write". If the CM entry is positive, AC must lock the count entry and read it. If that entry is one, AC can unlock the CM and count entries and return with the knowledge that the process has write capability (there is no possibility of mishap since only this process has read capability for the segment and it is operating in AC, so that no "sends" or "dones" are possible). If that entry is greater than one, however, AC must set the write-blocked flag in the CM entry, insert the prcsno in count-indicator, unlock the count entry and call "await-write". If the count-entry is zero error-status information must be returned (or put into a segment for copying later) and perhaps all the processes in the computation aborted. "Send", "done", "put-descriptor", etc., also use the locking mechanism in a similar way. Hereafter, the details of locking and unlocking will be specified without such a detailed explanation.

Chapter 4

The previous chapter discussed how processes and segments in the system are controlled in accordance with Van Horn's discipline after they come into existence. This chapter discusses problems presented by the creation of processes and segments and presents a set of solutions.

4.1 The Fork

Processes are created by an operation known as forking. Conway [7] defines fork as a meta-instruction that starts a new process executing at a label. It can be interpreted more generally as the operation of creating a sub-computation to be executed in parallel. In particular, the sub-computation could consist of a single process. As an external procedure defines a computation, the execution of an external procedure in parallel corresponds to the execution of a sub-computation in parallel. The case where this external procedure is of the conventional type is a case where the sub-computation consists of a single process. If it uses a fork, one has a generalised sub-computation consisting of several processes. With this association of external procedures and sub-computations, it will be seen that a fork is quite reasonably interpreted, in the context of processes executing procedures, as a call to set a new process executing an external procedure with certain arguments.

As a counterpart of fork, one needs an operation whereby a parallel process is terminated. Dennis and Van Horn [5] suggest 'quit' as such an operation. They also indicate that Conway's [7] join serves as a mechanism whereby a process can continue just when some processes have terminated. In terms of the system being implemented it turns out that a quit is achieved very simply by a return of the procedure called at a fork. There are several variations of the basic join [5] which is "join t,w" where "t is the word name of a count to be decremented and w the word name of an instruction word to be

executed if the count becomes zero..."[5]. Translated in terms of processes in the system and reduced to essentials this means join awaits the termination of a process and the process then proceeds to execution of the next instruction. Such a join is "join (process-name, prcsno)". If further it is made a convention that the process, termination of which is awaited, is the most recently created one, no argument is necessary and one obtains "join". Finally, "join (n)" awaits the termination of the n most recently created processes. All three join's are provided for in this system, although the first one is sufficient.

4.2 Creation of Segments

When a data segment is created, there are three attributes which are of importance. These are the symbolic name, the directory in which the segment resides and the access bracket. The second attribute has only one of two possibilities. If the segment is meant to be potentially accessible to all processes, it is ordinarily put in the working directory (which, it turns out, is usually common to all of a user's processes). If the segment is meant to be private to this process, it is put in the process directory. As for the access bracket, it need only have a lower limit higher than r_{min}, being anything the user chooses, otherwise. The first attribute is not that easy to handle. This is made clear in the following paragraph.

Let it be assumed that the system naively takes the character string specified by the user as the name to be assigned to a shared segment. Suppose, further, that this string is "alpha". As long as the computation runs by itself no ambiguities as to which segment is referred to by "alpha" arise, assuming that the user has not multiply defined segments with the same name. Suppose, however, that another user decides to incorporate this computation as a sub-computation. The principle of modularity requires that the user be not required to know details of the embedded sub-computation. It is quite pos-

sible, therefore, that the user has also used alpha as a name for a shared segment. Because of the common working directory, a multiple definition of the name alpha and consequent ambiguity result. The MULTICS concept of relating segment names to procedures provides the solution to this problem.

From the above discussion it follows immediately that a call to the system procedure "createsegment" (cf. Fig. 4.1) is

call createsegment (name, type1, type2, access-bracket, calling-procedure).

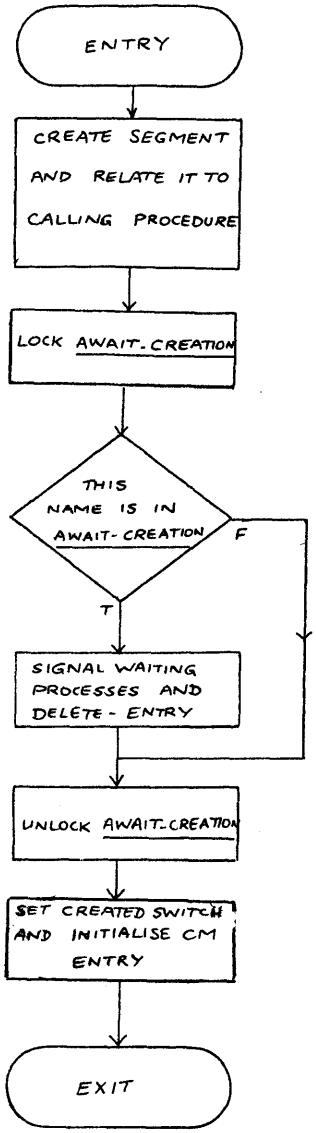
where

name	is the name to be assigned to the segment
type 1	is "directory type" (0) or "non-directory type" (1) for the segment.
type 2	is "private" (0) or "shared" (1)
access-bracket	is the ring access bracket (r_1, r_2) $r_1, r_2 > r_{min}$
calling-procedure	is the symbolic name of the procedure making the call.

This routine calls MULTICS "append-branch":

call append-branch (dir, name, type 1, RAW, or 01, 256) REAW,

which creates a related entry named "name" in "dir" (specified as the working directory or process directory--depending on "type 2"--whose names are easily found using the MULTICS functions "wdir" and "pdir" respectively). The entry is of type "type 1" with the user permitted mode REAW (read, execute, append, write) if it is a directory entry and mode RAW if it is a non-directory entry. The maximum length of the segment is 256 K words. Since arrangement has been made in the above creation of an entry for all processes to get the same segment and not a copy in their own process directories, the entry corresponds to one segment. Effectively, then, one segment with the specified name has been created, as was desired.



"Createsegment"

Figure 4.1

"Createsegment" then relates this entry to the calling procedure by using the MULTICS primitive for relating. It then creates the appropriate access bracket and call bracket for the segment by a call to a MULTICS routine:

call set-protection (path-name, plist, glist, ac names)

where path-name is the path name of the segment relative to the working directory or root directory.

plist is (a_1, a_2, a_2)--the access bracket (a_1, a_2) and the call bracket (a_2, a_2)--if $a_1 > r_1$ $a_2 > r_1$ else, (r_1, a_2, a_2) or (r_1, r_1). $r_1 = r_{min}$.

glist, acnames } are irrelevant here and are therefore null above.

This routine ("createsegment") is to be used for the creation of data segments only. Creation of procedure segments and specification of access rights of other users is by calls to MULTICS routines. Also, if the segment is to be used by other procedures, the user must relate it to these procedures so as to guarantee that they get the correct segment when they use that name.

4.3 Deletion of Segments

In essence, the deleting of segments is a very simple affair. MULTICS provides the primitive "delete-entry" which is called thus:

call delete-entry (dir, entry, courtesy flag)

where dir is the directory where the entry is to be found

entry is the entry in this directory which corresponds to the segment

courtesy flag is a flag which indicates whether or not this routine should wait until all processes reading from this segment finish. This flag can be specified as 2, meaning do not show any courtesy (as the system ensures that no other process is reading it, by the definition of Van Horn write capability).

Write-permission is required for dir and entry.

4.4 Miscellaneous Aspects of Segment Creation

With creation of segments permitted within the system, one can conceive of a situation wherein a process makes reference to a segment but the segment has not been created yet because of delays in the process which was to create it. As the system stands, the MULTICS search module would merely signal an error because the segment is not found. Such action is clearly erroneous. The process should instead block*, awaiting creation of the segment. This is ensured by the mechanism indicated in the next paragraph.

A segment, await creation, is added to the working directory and to each process directory. These segments are created at the first call to "fork" (or when a user changes his working directory) and upon creation of the process respectively. This segment is used to record the segment names for which a futile search results. The MULTICS search module is changed so that it locks the await-creation segment and makes an entry for the missing segment in it. An entry in await-creation consists of the segment name, process-id and the event-channel-id of the process. The search module then calls the MULTICS "we$wait" primitive after unlocking the segment. The procedure "createsegment" locks await-creation in the appropriate directory (determined by its third argument) and checks to see if the name of the newly created segment is found as an entry. It deletes the entry if found and signals on the channel specified by the entry by calling the MULTICS primitive "ecm$set-event", unlocks the segment and returns.

Another fact deserves mention. This is that the working directory and process directories need not be subject to Van Horn's discipline. They are

---

*From the point of view of practicality it is probably more appropriate at this point to ask the user if the process should wait or not (the failure to find the segment could be, for instance, the result of a typographical error in the name of the segment). In the latter case the computation should probably be abandoned.

not used as data segments, and the individual segments in them created by the user are disciplined as indicated in Chapter 3, so that no non-determinacy results. Moreover, if these segments were subject to the discipline the user would be burdened by the large number of "sends" and "dones" required. Moreover, these calls cannot even be specified a priori when data-dependent process creation is required. In short, the pattern of such capability transfers is complex. For this reason these directories must be in rings interior to r_{min}. No user procedure can delete the process-directory as the directory in which it resides is not write-accessible in rings outside r_{min}.

It should be noted that when MULTICS routines are used to create segments, in no case can any of these segments have any number in the call or access bracket lower than the ring from which an attempt to set the protection list is made. Thus all such segments are guaranteed to be in rings outside r_{min} and subject to the Van Horn discipline and no non-determinacy is possible.

4.5 The Naming of Processes

A naming scheme for processes should be such that the user knows the name of a process at the time the program is written (to program sends, for instance), and yet no conflict arises due to the nesting of computations. Such a scheme will now be described.

One way of ensuring that a name is always associated with a pre-specified object irrespective of any other uses of the same name is by use of a tree structure. The advantage offered by this structure is that a name is not used by itself to identify an object; rather it is qualified by the names on the branches on a path to it from the root of the tree. In this way two or more objects can be represented by the same name provided their qualifiers are different.

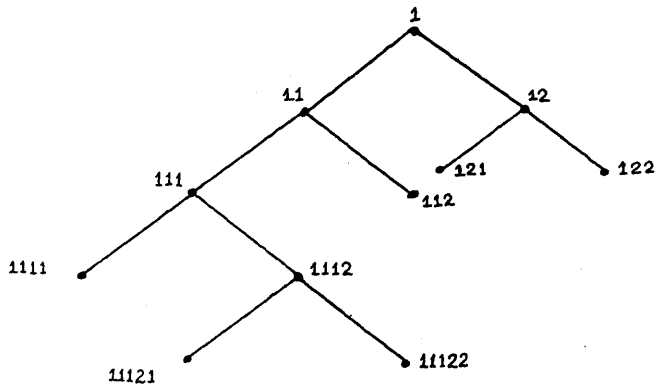
In the case of the fork, it will be recalled that only one new process is created by a fork. The heirarchy of process creation is thus a bifurcated tree

whose terminal nodes represent processes that are currently active. If the branches of the tree are numbered 1 or 2 according as the branch lies to the left or to the right, one arrives at a set of unique names for each current process. Such a tree is shown in Figure 4.2.

An interesting property of this structure is that the path-name (set of labels on a path from the root of the tree to a node) of a node for a son process is always the path-name of the father with the last "1" stripped off and replaced by 2. Further even if the whole tree is made a sub-tree of a bigger tree (this corresponds to embedding of a computation in a bigger one), the same rule holds. This algorithm therefore provides a means of naming processes which results 1) in predictable and distinct names for all active processes and 2) in rules for determining the name of any node (or process) whose position relative to some node is known, the rules being independent of the depth of the nodes below the root (i.e., independent of the degree of nesting of computations).

The rules for determining the names of processes during creation and deletion activity are, 1) The name of the first process is 1. 2) At a fork the name of the created process is the current name of the creating process concatenated with 2. 3) The new name of the creating process after a fork is its old name concatenated with 1. 4) At a join, the last 1 in the name of the process doing the join is stripped off only if the son process has terminated. The name resulting from deconcatenation is examined again for possible removal of 1's until no further removal is possible. The process performing the join can then proceed.

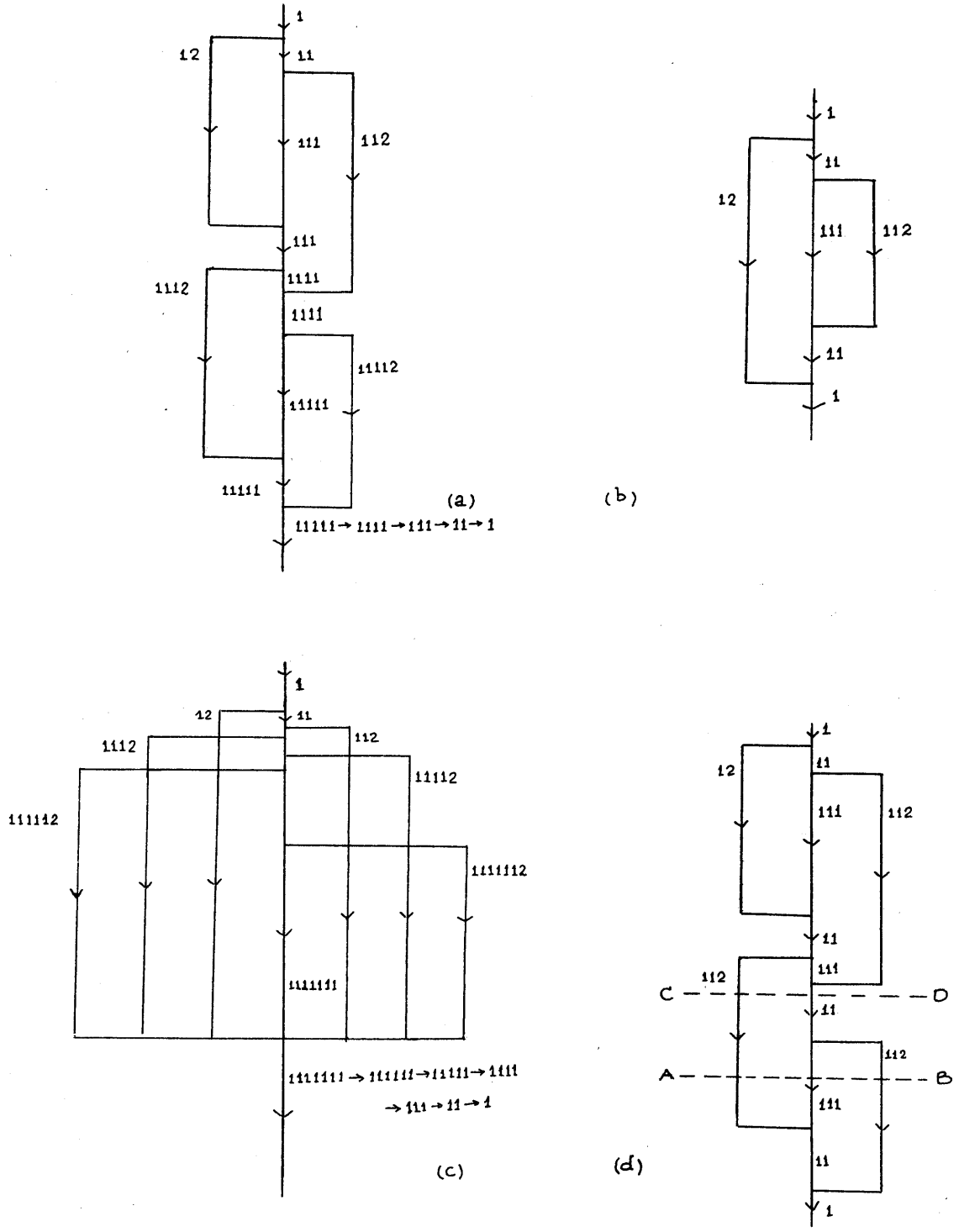
The "only" restriction in rule 4 is required because a join of the first kind, i.e. "join (process-name)" is permitted. Figures 4.3 (a) to (c) show typical diagrams showing names of processes when forks and the three kinds of join are performed. Figure 4.3 (d) shows the same sequences as in Figure 4.2 (a)



TERMINAL NODES INDICATE CURRENT NAMES OF EXISTING PROCESSES.

Process Creation Heirarchy and Naming

Figure 4.2



Three Kinds of Join's

Figure 4.3

but with rule 4 above modified to exclude the "only". The situation at the line AB shows two processes having the same name. Also the usual rules for finding the name of a son process do not apply (for instance at CD). This demonstrates the need for the "only" in rule 4.

This scheme makes it very simple for a user to program his sends and joins as the name of a process is now predictable. Further, such statements as "Send capability for segment j to my creator's creator" can be implemented too since all that is necessary is to strip off the last two digits of the name of the sending process to get the sendee's process name. Clearly the name of any node in the heirarchy can be created from a process' name by such truncation and concatenation.

Three questions can be asked about this scheme. Firstly, does the large number of names for a process imply that the sender must determine the sendee's current name? Secondly, does this scheme preclude symbolic process names? Finally, how can such a scheme be implemented? These questions are answered below.

It will be recalled that the reason for adopting this scheme is to permit reference to rows of the CM for sends and dones. It is vital, therefore, that the name (actually, number) of a process as specified by the above scheme be used in the name of the corresponding row segment of the CM. Now, since this segment is created when the process is, the number of a process at the time it was created is what is used to name the segment. This unique and fixed number is all that is required to do a send/done to the corresponding process, not the current number of the process. If, however, the process number specified for a send/done is obtained by stripping off some digits, such a number could contain several 1's at the right hand end. These 1's are also stripped (by the system procedure "send" or AC) to get the number of the process at birth and the name of the corresponding CM segment.

The facility of symbolic names is easily introduced. The reader will recall the scheme of integer variables used to record the unique numbers assigned to segments. The same scheme can be used here. The user declares an integer variable with the symbolic name that is to be given to a process. This variable is passed as an argument to "fork" which stores the number of the new process in it. Any subsequent call to "send", "done" with this variable as an argument results in an evaluation of that variable to get the process number which is then used to make access to CM. The number itself can be given as an argument too. Of course, this symbolic name is meaningful only in the process which passed it as an argument.

Finally, the implementation of the scheme can now be specified. A location called "prcsno" in a per-process private data area (such as the segno-equivalents segment) is used to contain the current number of a process. It is initially set upon creation of the process. It is continuously updated as forks or joins occur. At a fork it is used to find the number of the new process and to create a CM segment with the corresponding name.

In passing it may be mentioned that the CM segments are related to the system procedures at the time of creation so that the user can use those same names for his own segments if he wants to. (He can relate those segments to his own procedures too.) This procedure is standard for all segments created for system purposes.

4.6 Implementation of the Fork

Fork is implemented as an external procedure. A call to "fork" is:

call fork (entrypt-name, prcsname, arg-list)

where	entrypt-name	is the symbolic name of the external procedure to be called by the new process
	prcsname	is an integer type variable that will provide the facility of a symbolic name for the process. This argument can be null

arg-list — is a list of the arguments to be passed to the external procedure.

The implementation of the procedure is described a little later.

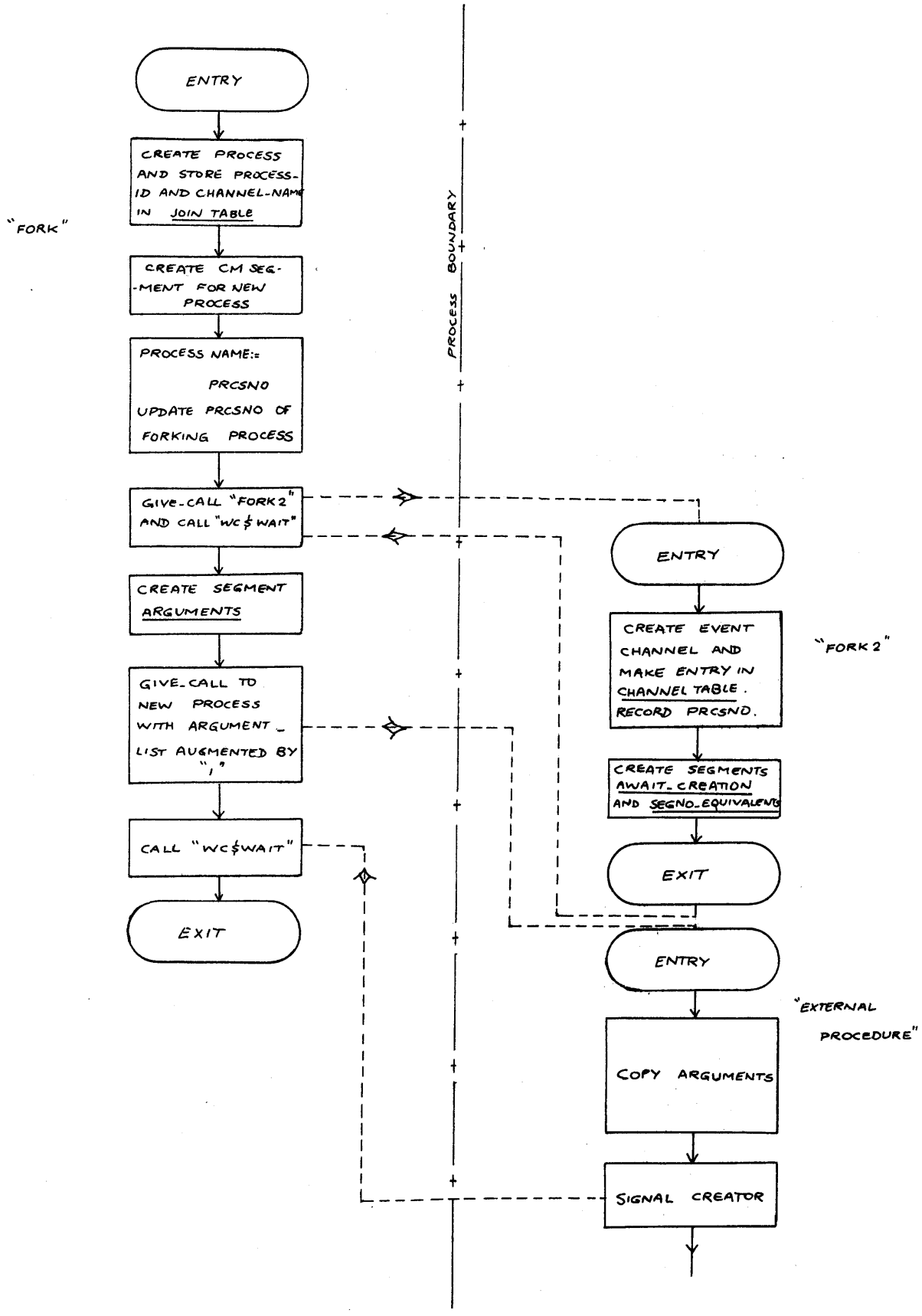
In connection with the creation of private segments for use by the new process, it is noted that an external procedure is complete, in the sense that the only information given to it from outside is that constituted by its arguments and shared data. The creation of any private data structures (such as segments) is thus the responsibility of the procedure. "Fork" does not create any private segments (other than for system purposes) for use by the new process.

One of the MULTICS routines used by "fork" is "give-call" which makes another process call the procedure specified by the give-call with the specified arguments. It is a peculiarity of this procedure that the arguments are not copied; rather, pointers to these arguments are created. In this way the two processes use the same memory location(s) for each argument. It is very possible, therefore, that a race between these processes could result, leading to possible non-determinacy. What is required, therefore, is the "call by value" feature of ALGOL. Implementation of this feature is discussed in section 4.7.

The activities of "fork" are quite numerous and the reader may wish to consult Fig. 4.4 throughout the discussion. The first task of "fork" is to create a process. This is done by a call to MULTICS "create-wp". One of the arguments is prcsno. This is obtained by concatenating the name of the creating process found at segno-equivalents|prcsno with a 2. The call to "create-wp" is therefore

call create-wp (prcsno, process-id, event-channel-name, status);

The last three arguments are returned by "create-wp". The first three arguments are then stored as an entry in the segment join-table. They are re-



"Fork"

Figure4.4

quired by the procedure implementing the join.

It will be recalled that the CM segment for the new process has to be created by "fork". "Fork" thus takes the prcsno of the new process and creates a segment named "prscno-row" in the working directory by a call to MULTICS "append-branch". The MULTICS mode of the user is such as to permit reading, writing and appending (RWA).

Next, "fork" stores the prcsno of the new process in its second argument, the integer type variable that will be used in calls to other procedures such as "join" to identify the new process. The entry at prcsno in the segno-equivalents segment of the forking process is updated by concatenating its present contents with a "1".

At this stage the forking process must start up initiation activity in the new process and block awaiting completion of that activity. For this purpose "fork" makes the new process call "fork 2" with the prcsno of the new process as argument and calls the MULTICS Wait Coordinator to await a completion signal from the new process on the event channel created by "create-wp".

Fork 2 is an external procedure which first creates an event channel for use by other processes to wake up this process when it receives some capability (cf. Section 3.4). This is done by the call:

call ecm$create-ev-chn(ev-chn,mode,signl-ring)

where
- ev-chn — is the event channel name returned by the procedure
- mode — is "0" (boolean) to indicate event-count mode
- signl-ring — is the ring number from which "wc$wait" and "ecm$delete-ev-chn" can be called i.e., the ring in which "join", "await-read" and "await-write" reside.

The channel-name returned, together with the process-id, i.e., the channel key, is stored in channel table as the value of an entry whose key is the

prcsno of this process (the argument received by "fork 2"). "Fork 2" stores its arguments at prcsno in the segno-equivalents segment of this process after it creates the segments await-creation (cf. sec. 4.4) and segno-equivalents (cf. sec. 3.3) in the process-directory. "Fork 2" then returns.

The forking process is awakened by the above return and "fork" creates a segment named arguments in the process-directory of the new process. It then issues a call to MULTICS "gvclf$give-call":

call gvclf$give-call("proc-b-name",prcs-b-id,rtn-chn,stat-rtn,a,a....a)

where	proc-b-name	is the entry point specified by the first argument of the call to "fork"
	prcs-b-id	is the process-id of the newly created process (found in join table)
	rtn-chn	is the channel-name returned by "create-wp"
	stat-rtn	is return status information
	a,...,a	is an argument list (consisting of the argument-list provided to "fork" and an "on" switch--cf. sec. 4.7).

Fork then returns.

Some additional work is required of "fork" (before the last give-call), the very first time it is called. This is the creation of several segments for the first process in the system and for system use. Thus prcsno-row for the first process, await-creation, join-table, channel-table, equivalence-table, user-restrictions-table and count have to be created in the working directory. Segments await-creation and segno-equivalents have to be created in the process directory of the first process. An indication of whether or not a call to fork is the first is provided by some variable in a shared segment.

4.7 Call by Value at a Fork

The reasons why a give-call to the new process at a fork should be a call by value have been explained in section 4.6. Implementation of such a

call is not easy.

At first it appears that "fork" can be made to copy the values of the arguments in arguments and then use pointers to these values in the give-call. However, this solution is deceptive as "fork" cannot know the nature of an argument. The copying has thus to be done in the external procedure itself. That is, a compiler should put in code at the start of every procedure to copy the arguments in arguments and use pointers to these values as arguments. However, as ordinary PL/I calls are calls by name, this code should not be executed ordinarily. A switch to decide whether a procedure is called at a fork or in a conventional manner is therefore required. A convenient way of obtaining this switch is described in the next paragraph.

Suppose the compiler is further modified so that it puts in code for the construction of one extra zero argument, in addition to those provided,in the argument list created at a call to a procedure (say alpha). Then if n is the number of arguments expected by alpha, the $n + 1^{st}$ argument in the argument list handed to it is ordinarily zero. At a fork, however, it is the $(n + 2)^{th}$ argument that is zero while the $(n + 1)^{th}$ is "1." The $(n + 1)^{th}$ argument therefore serves as a switch for the copying of arguments.

One additional consideration is that of ensuring that the forking procedure does not continue before the arguments have been copied. This is achieved by having "fork" call the Wait Coordinator to wait on its channel (whose key appears in channel table). The called external procedure signals on this channel after copying is complete. This is shown in Figure 4.4.

4.8 Implementation of the joins

The three kinds of join are implemented as entry points in the procedure segment join. The basic action of a join is to update the name of the process calling join and to destroy terminated processes. It is the event which de-

cides when this action is to occur that distinguishes the three joins.

"Join1" uses its arguments, i.e., either prcsno or process-name, to look up join table for the channel key of the channel for the process whose termination is awaited. It then calls the MULTICS primitive "wc$wait" to await a signal on this channel. The signal is sent by a MULTICS procedure when the process terminates, i.e., when control reaches a "return" statement in the external procedure that was called at the fork which created the process. Join1 then calls the system procedure "update" (cf. Fig. 4.5) and returns.

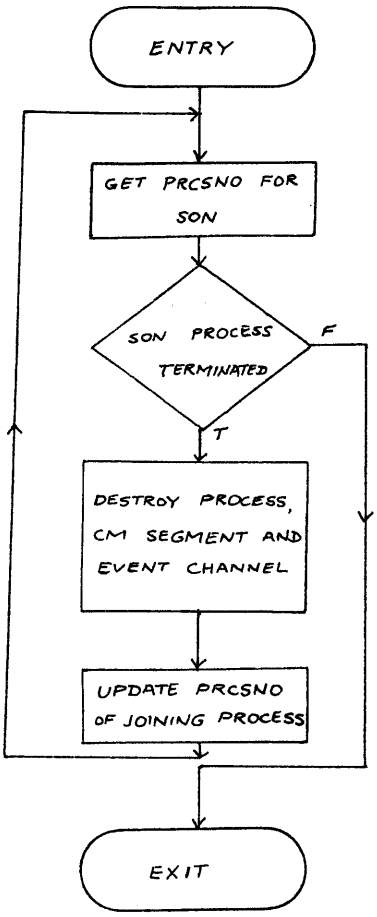
Join2 uses the prcsno of the calling process to determine prcsno for the son-process. (Join2 does not take any arguments.) It uses this prcsno to get a channel key from join table. It then awaits signalling on this channel by the son process. Finally, it calls "update" and returns. "Join3" merely calls "join2" n times where n is its argument.

Update checks if the son process has terminated (using MULTICS "we$test-event"). If it has, update calls the MULTICS procedure "destroy-wp" to destroy the terminated process:

call destroy-wp (process-id, status)

where	process-id	is the process-id of the son process (found in join-table)
	status	is return status information.

It then calls MULTICS "delete-entry" to destroy the CM segment for this process. The event channel for this process is destroyed too. It then updates the current name of the joining process by deleting a trailing 1. Update repeats the above procedure until a stage is reached where a son-process has not terminated. It then returns. The repetition is necessary as preceding calls to "update" by join1's may not have had any success in deletion of 1's at all.



"Update"

Figure 4.5

4.9 Input-Output

No special comments on Input-Output activity are required as the usual access checking mechanism is effective.

4.10 Initialisation of the Control Matrix

The foregoing sections have described the system almost completely. However, the problem that is still unsolved is that of initialisation of the Count Matrix. This problem can be stated as, "If a process cannot give itself additional capability (it must be given by some other process), how does the first process get its capabilities?"

There are two ways of solving the above problem First, the MULTICS routine that created the first process could be required to set up the initial CM. However this does not take care of segments created by the first process. Also, it is not possible to specify a priori all the segments that the first process will use. Some form of "incremental initialisation" is therefore required. One possibility is to have a convention that the first process to attempt to access a segment is presumed to have the capability and a CM entry is made for it. This produces a problem of races between the first process and any other process in the sense that the latter may have been intentionally denied capability for the segment (by not being given the capability) but gets it all the same merely because it attempts access first.

A good convention is to assume that the first process always has capability for a segment unless the segment has been created by another process in this computation. Any process that creates a segment is simultaneously granted capability for it. No other process has capability for this segment unless explicitly granted it. In other words the system routine Access Checker determines if a process lacks a capability. If it does, the routine finds out if its name is "1" (the first process) and if it is, checks the count entry

for the segment to see if the created-switch is on. If the switch is on, this is a genuine case of lack of capability and the usual procedure is followed. If the switch is off, it is turned on and the CM entry for this process and segment is initialised to "1". When a process creates a segment, the system routine "createsegment" (cf. Fig. 4.1) creates an entry in count and sets the created-switch on while initialising the creator's CM entry for that segment to "1". In this way the CM is initialised "incrementally".

All aspects of the system have been described at this stage. The next chapter shows some examples of how the system may be used.

Chapter 5

5.1 Verification

It is perhaps appropriate at this stage to verify in an informal manner that the system design given in the two previous chapters does, in fact, satisfy all the conditions of Van Horn's model. It is clear that because of the access checking mechanism only enabled processes can proceed with their computational activities. Further, the locking mechanism used for CM entries ensures that these processes also belong to the choice collection (effectively). The definitions of activity of "done" and "send" in sections 3.3 and 3.4 ensure that an enabled process begins computational activity in a finite time after it becomes enabled. It can, therefore, be concluded that a computation carried out within this system posseses all the properties of an MCM computation (provided the MULTICS locking mechanism gives every process waiting for a locked CM entry a chance to use the entry in a finite time).

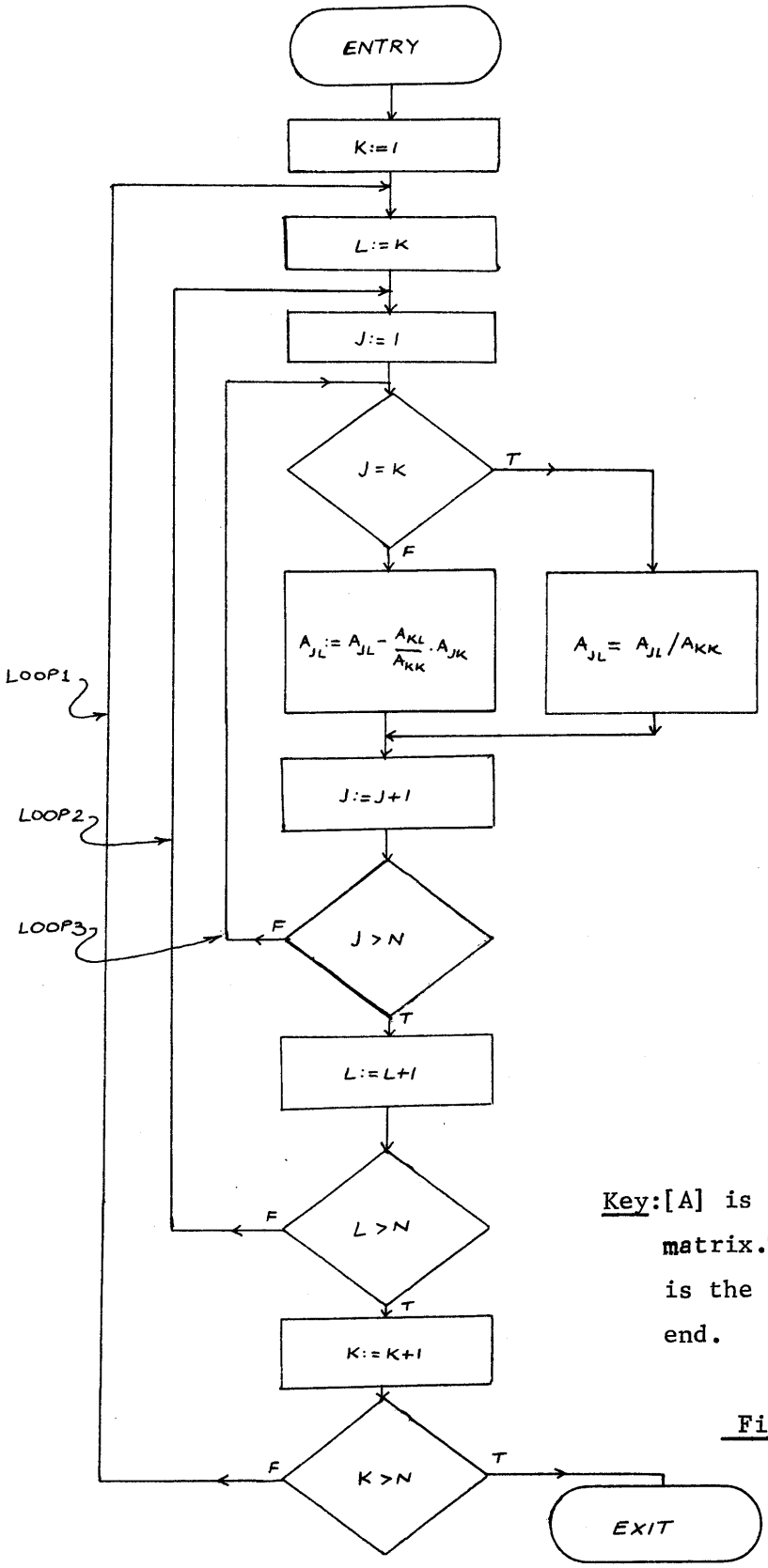
5.2 An Example of Use of this System

The example chosen for purposes of illustration is the Gauss-Jordan method of solution of a set of simultaneous linear equations of the form

$$[A'][X] = [C]$$

where [A'] is an N x N coefficient matrix,[X] is an Nx1 vector of variables and [C] is an Nx1 vector of constants. The algorithm is shown in Fig.5.1. It will be noted that after the k^{th} pivot (A_{kk}) is used, the k^{th} column of the augmented matrix [A] ([A'] with [C] added on as one more column) is not used at all, i.e. it is as good as if it were deleted. This fact is useful in understanding the destruction of segments in the program given at the end of this chapter.

For the purposes of this program, a matrix is assumed to be made up of column segments. The program is written in PL/I. It is not claimed that the



Key:[A] is the augmented matrix.The N+1th col. is the solution at the end.

Figure 5.1

program indicates the best way to implement the algorithm; it is chosen for illustrative purposes only. It will be noted that Loop 2 is executed in parallel in the program. The symbol for concatenation in the program is the set of two vertical bars.

Several comments about the program are in order. SEGNM is the name of a segment containing the array [A]. The two calls to "createsegment" in the loop create non-directory segments in the working directory. These segments are related to the procedure and reside in ring 64. "Copy1" is a procedure that copies the j^{th} part of SEGNM (i.e. the j^{th} column of [A]) into 'C'||J. "Transform" carries out the transformation inside Loop 2 on a column of the augmented matrix ('C'||J) and copies the transformed column into the 'D'||J segment. 'C'||J is left unchanged. "Call copy('S','D'||I+1)" copies the last column of the augmented matrix, i.e. the solution, into the segment S. NM is a function that returns prcsno for this process. DNM gives prcsno for the parent process.

5.3 Embedding

To illustrate the embedding of computations it is proposed to show a circuit analysis program that analyses networks and prints out solutions. It uses the program of section 5.2. Setting up of the equations and analysis are done concurrently. The program is given at the end of the chapter. "Set-up-eqns" sets up the node equations for the network (specified on a graphic input device, say) and returns the number of variables, N, and the set of loop equations. "Print" prints out the solution. The logical function "condition" indicates when to stop. The rest of the functions are as in section 5.2.

It will be noted that the writer of "Cct-analyser" need not know that "Lin-eqn-solver" is a multi-process computation. There is clearly an embedding

of a multi-process computation within a multi-process computation, here.

5.4 Conclusion

The preceding discussion has, the author hopes, illustrated that programming a multiprocess computation in the proposed system is not difficult. Actual implementation of this subsystem in MULTICS as a working sub-system runs into two "problems". Firstly, one of the system procedures has to be in ring zero. This is not normally permissible but this objection can be easily overcome as that procedure is simple and, therefore, easily debugged. Moreover, the ring zero procedure is required because MULTICS fails to distinguish between processes working for the same user for access control purposes. Secondly, changes in compilers are necessary in order to implement the call-by-value feature required by "fork". This is a direct result of the inadequacies of existing languages in the context of multi-processing, and is unavoidable without language reform.

One assumption and two restrictions are required in order to assert that all of Van Horn's conditions for determinacy are met. The assumption is that MULTICS ensures determinacy of single-process computation and this seems reasonable. One restriction is that the MULTICS locking mechanism use a round-robbin or similar scheme that ensures every process waiting for access to an entry a chance to do so in a finite time. The other restriction requires a user not to use a directory that is write-accessible outside the ring r_{min} as the parent directory of a segment he/she wishes to use as a working directory. Neither this restriction nor the restrictions on the user that all procedures be pure, that all shared data-bases must be segments and that Input-Output activity is restricted to segments seem to constitute too stiff a price to pay for the determinacy gained.

```
LIN_EQN_SOLVER:PROCEDURE(SEGNM,K,S,N);
               DECLARE K(*),N FIXED,(SEGNM,S) CHARACTER(6);
               J=1;
            L1:CALL CREATESEGMENT('C'||J,1,1,'64','LIN_EQN_SOLVER');
               CALL CREATESEGMENT('D'||J,1,1,'64','LIN_EQN_SOLVER');
               CALL COPY1('C'||J,SEGNM,J);
               J=J+1;
               IF J<=N THEN GO TO L1;
               I=1;
            L2:J=1;
            L3:CALL FORK(TRANSFORM,0,'C','D',I,J);
               CALL SEND(0,'C'|||I,NM||(J)'2',0);
               CALL SEND(0,'C'||J,NM||(J)'2',0);
               CALL SEND(0,'D'||J,NM||(J)'2',0);
               CALL DONE(0,'D'||J);
               J=J+1;
               IF J<=N THEN GO TO L3;
               CALL JOIN3(N);
               IF I=N THEN GO TO L5;
               J=1;
            L4:CALL COPY('C'||J,'D'||J);
               J=J+1;
               IF J<=N THEN GO TO L4;
               CALL DELETE_ENTRY(WDIR,'C'|||I,2);
               CALL DELETE_ENTRY(WDIR,'D'|||I,2);
               I=I+1;
               GO TO L2;
            L5:CALL COPY('S','D'|||I+1);
               CALL DELETE_ENTRY(WDIR,'D'|||I+1,2);
               CALL DELETE_ENTRY(WDIR,'C'|||I+1,2);
               CALL DONE(0,SEGA);
               CALL SEND(0,'S',DNM 0);
               CALL DONE(0,'S');
               RETURN;
  CCT_ANALYSER:PROCEDURE;
               DECLARE(N1,N) FIXED,K(100),K1(100);
               CALL CREATESEGMENT('SEGA',1,1,'64','CCT_ANALYSER');
               CALL CREATESEGMENT('SEGB',1,1,'64','CCT_ANALYSER');
               CALL SET_UP_EQNS('SEGA',N,K);
               CALL CREATESEGMENT('S',1,1,'64','CCT_ANALYSER);
            L1:CALL FORK('LIN_EQN_SOLVER',0,'SEGA',K,'S',N);
               CALL SEND(0,'SEGA',NM||2,0);
               CALL SEND(0,'S',NM||2,0);
               CALL DONE(0,'S');
               CALL SET_UP_EQNS('SEGB',N1,K1);
               CALL JOIN;
               CALL PRINT('S');
               CALL COPY('SEGA','SEGB');
               IF CONDITION THEN RETURN;
               GO TO L1;
               END
```

References

[1] Corbato,F.J.,et.al.,"A New Remote-Accessed Man-Machine System," a set of six papers,AFIPS Conf. Proc. 27(FJCC 1965),Spartan Books,Washington,D.C., 1965,pp.185-247.

[2] Crisman,P.A.,editor,The Compatible Time-Sharing System:A Programmer's Guide, second edition, M.I.T.Press, Cambridge,Mass.,1965.

[3] Van Horn,E.C.,"Computer Design for Asynchronously Reproducible Multiprocessing," Ph.D. Dissertation,M.I.T.,Cambridge,Mass.,1966.

[4] Graham,R.M.,editor,MULTICS System Programmers' Manual, Project MAC,M.I.T., Cambridge,Mass.1967.

[5] Dennis,J.B.,and Van Horn,E.C.,"Programming Semantics for Multiprogrammed Computations," Communications of the ACM 9, 3 (March 1966),pp 143-155.

[6] Dijkstra, E.W.,Cooperating Sequential Processes,Department of Mathematics, Technological University,Eindhoven,Netherlands,Sept.1965.

[7] Conway,M.E.,"A Multiprocessor System Design," AFIPS Conf. Proc. 24 (FJCC 1963), Spartan Books,Baltimore Md.,pp139-146.

Abbreviations Used

AFIPS:American Federation of Information Processing Societies.

FJCC:Fall Joint Computer Conference.

ACM:Association for Computing Machinery.

M.I.T.:Massachusetts Institute of Technology.

Conf.:Conference.

Proc.:Proceedings.