A STUDY OF STACK ARCHITECTURE IN CONTROL PROGRAM DESIGN

by

Joseph H. Austin, Jr.

November 14, 1968

Abstract

A model of an event-driven multi-tasking control program is examined to determine which information structures are appropriate to control program design. These are compared with the capabilities of the stack mechanism of the Burroughs B6500. It is concluded that many requirements of the control program are not satisfied by the stack concept. A more general information structure and possible implementation are proposed.

A STUDY OF STACK ARCHITECTURE IN CONTROL PROGRAM DESIGN

PART I: INTRODUCTION

This paper is concerned with pushdown stack architecture from the viewpoint of the control program. First we will examine the basic von-Neumann machine and the motivation for stack architecture. In Part II we will examine a model of the control program to discern what information and control structures are appropriate to it. Part III will compare these with features available in the Burroughs stack machine. Part IV will suggest extensions to machine architecture to simplify the control progran

I. Basic von-Neumann Machine[1]

The basic addressing structure of a von-Neumann computer is given in Figure I.1:

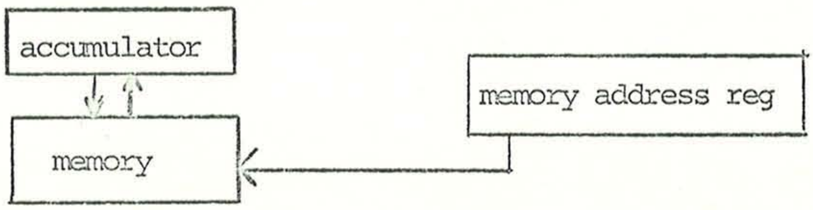


Figure I.1: von Neumann machine

This machine has a single accumulator which is implicitly addressed.

II. Motivation for the Stack Machine

An early problem with this machine organization was the necessity for saving and reloading the contents of the single accumulator. The problem could be solved by providing an "infinite" number of accumulators; thus intermediate results could remain in an accumulator as long as needed. Theoretical studies revealed that, for operations of common occurrence in scientific programming, such as evaluation of arithmetic expressions, compilation of "Algol-like" languages, and subroutine calls, the sequence of required accumulator save and restore operations corresponded to the pattern of a pushdown stack. Accordingly, when Burroughs Corporation designed the B5500 for efficient implementation of ALGOL, it included a hardware-simulated "semi-infinite" set of accumulators organized as a pushdown stack.

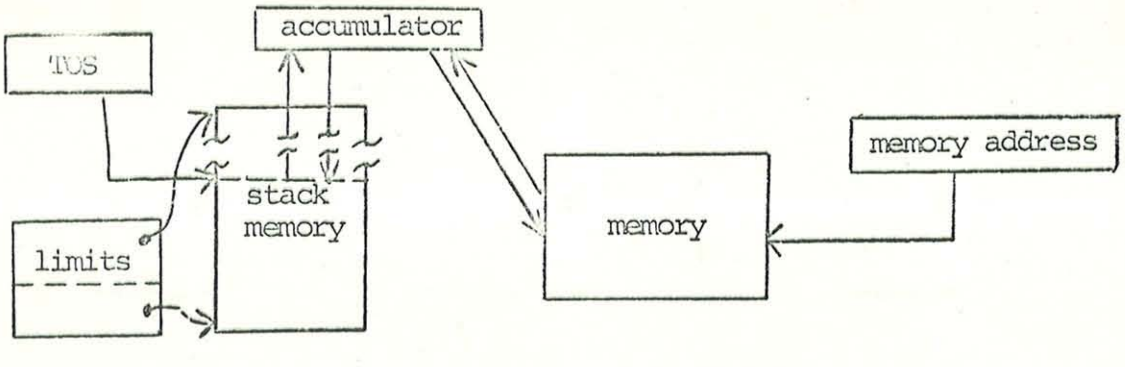


Figure I.2: Stack machine

[1]e.g., the IBM 650. See reference (1).

In this machine, real (hardware) accumulators are augmented by a vector of core locations which serve as logical accumulators. An additional top-of-stack (TOS) register maintains the address of the core location corresponding to the juncture of real and simulated accumulators. Each store or load on the accumulator causes an implicit pushdown or pop-up of the stack.

III. Emergence of the Control Program

Currently time-sharing and communications-oriented computing systems are growing in popularity. Due to price/performance economies of scale, these systems are typically implemented on large-scale computers, each with an elaborate control program to enable many users to share system resources concurrently.

We shall now consider whether stack architecture, of proven usefulness in executing Algol-type programs, is suitable for the execution of a heavily-used event-oriented control program.

PART II: THE CONTROL PROGRAM

I. Model of the Control Program

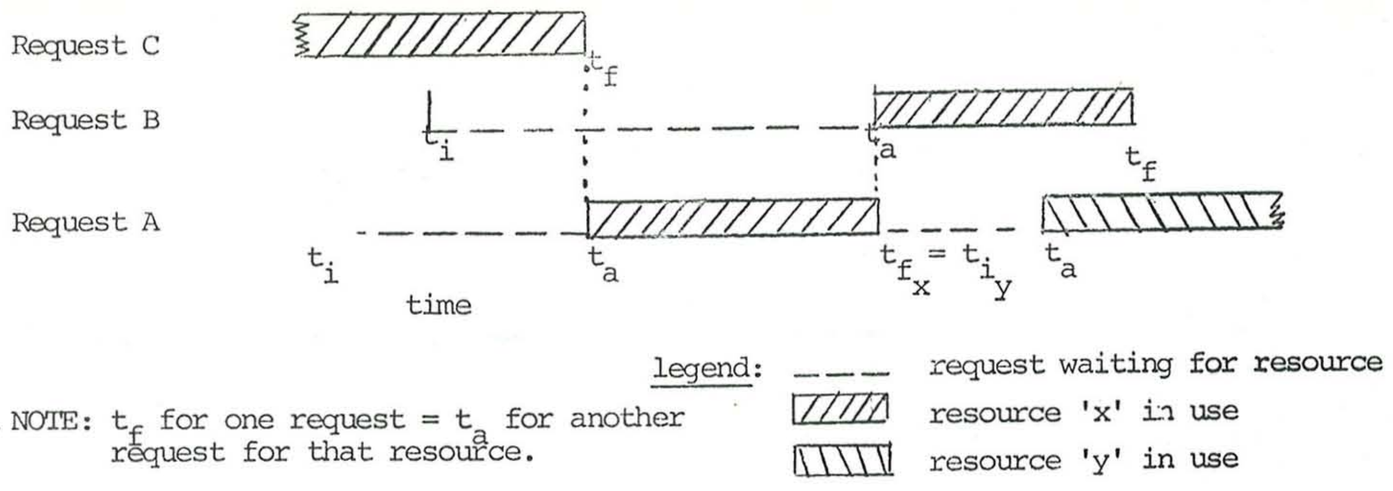
1. Introduction

The control program is a software-implemented extension of the computer architecture. This extension includes definition of pseudo op-codes[2] and simulation of features not implemented in hardware, such as I/O handling and interrupt servicing. The major function of interest in this paper is that of "multiplexing" the computer system to achieve multiprogramming and dynamic reallocation of resources.

2. Primary Function

The primary function of the control program is allocation of resources in response to events. Resources include control of CPUs and channels, memory and auxiliary storage, I/O devices and programs. The allocation process is performed by routines collectively known as the control program or supervisor.

A model of the supervisor thus characterized is the "job shop." Requests enter the system, require one or more resources, and are ultimately serviced, freeing the resources. The basic operation of the model is illustrated below:



NOTE: t_f for one request = t_a for another request for that resource.

[2]For example, supervisor call routines for IBM System/360.

There are three significant events in this basic cycle: t_i is the time of arrival of a request, t_a is the time the resource becomes available, t_f is the time the request is serviced and the resource freed. The cycle may be repeated for other resources.

3. Design Problems

The information and control requirements of the supervisor fall into three broad categories.

A. Queuing Problem:

Whenever an asynchronous event (request) requires a resource that is currently in use by another request, the control program must assure that both requests are serviced. It may:

a) pre-empt the resource, saving (stacking) the status of the current user, or
b) queue the request until the resource becomes available.

B. Inventory Problem:

The control program must maintain a record of the status of all system resources and locate these records when a request or completion event occurs.

C. Control Problem:

The control program must analyze each event to determine its logical significanc locate the appropriate requests and resources, and give control to the appropriate routines. This latter is typical of the allocation problem when the resource is a program.

II. Logical Data Structures Required by the Control Program

1. Basic Data Structure

The basic logical data structure in control program design is the set. In general, it is variable in size and may be ordered. Types of set organizations are distinguished by where insertions and deletions are made.

2. Definitions

Types of sets and their typical uses are as follows:

stack: insertions and deletions made at the same end (LIFO).
use -- preserve status of a pre-empted resource.

queue (unqualified): insertions at one end; deletions at the other (FIFO).
use -- hold a request until a resource becomes available.

ordered queue: insertions anywhere on the basis of a key; deletions from one end.
use: "priority" queueing.

list: insertions and deletions anywhere on a key basis; variable size.
use -- maintain inventory of a variable resource.

table: same ordering properties list, but fixed size and format.

others (tree, plex, etc.): additional properties indicate multi-membership or nestin structure.
use -- record of resources that are members of more than one set, or subsets of a larger set.

NOTE: The term "random" access or arrival will refer to an event requiring access to a point other than one of the "ends" of these set structures, even though these events have a "non-random" statistical distribution.

3. Data Structures for Particular Supervisor Functions

The types of data structures suitable for the queuing and inventory problems depend on the relative order of the events representing the entry of a resource or request into a set and those requiring later access to or removal of that element. In general, one will want to order the set in a way which will maximize access efficiency for all three events in the basic cycle. In some cases, the required structures can be restricted by constraining the natural order of events, e.g., by disabling unwanted interrupts.

We can examine the impact of access requirements on data structure by investigating particular cases.

Supervisor services can be divided into four main areas:

A. Task[3] management, concerned with creation and deletion of tasks and allocation of control of the CPUs.

B. Input-Output management, concerned with the creation and deletion of I/O requests and allocation of channels and devices.

C. Storage management, concerned with the allocation of main and auxiliary storage to tasks and I/O operations.

D. Timer management, concerned with multiplexing the interval timer.

A. Task Management (see Figure II.1)

The history of a task is characterized by (a) creation by a higher task, (b) alternating periods of ready, active (control CPU), and waiting, and (c) eventual termination, notifying the higher-level task. The main problems of concern to the information structures are task creation/termination and dispatching (allocating CPU control).

1. Creation and termination

We will consider the situation in which a new (daughter) task is created by an existing (mother) task. Then there are two tasks competing for the resources originally controlled by the mother task alone. In particular, this is an implicit request for control of the CPU. The mother task may wish to transfer, share, or reserve to itself other resources (I/O devices, access to variables). The new task may subsequently acquire additional resources of its own.

Resource inventory problem: the inventory data structure must reflect the new disposition of resources. For this purpose resources fall into two categories:

a) non-sharable resources: (e.g., card reader) may be kept or transferred. The control program must record which task owns the resource. If the resource is transferred, it must also record the disposition of the resource when the task terminates, e.g., a stack of previous owners.

b) sharable resources: (e.g., data in storage). The control program must record each task that has access to the resource, e.g., a tree of possible users.

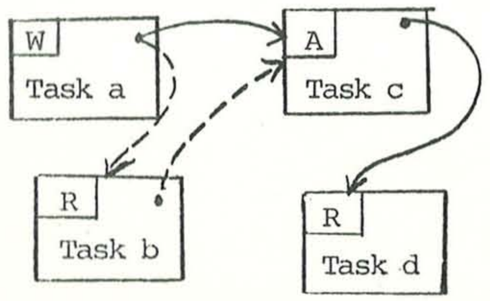
[3]By a <u>task</u> we mean a program module which may (logically) be executed in parallel with other program modules, e.g., independent multi-programmed jobs. Ref.(10)p.329.

DOS[4] Task Scheduler (table organization):

W	supervisor
W	error recovery
A	tele-processing
W	spool
R	batch

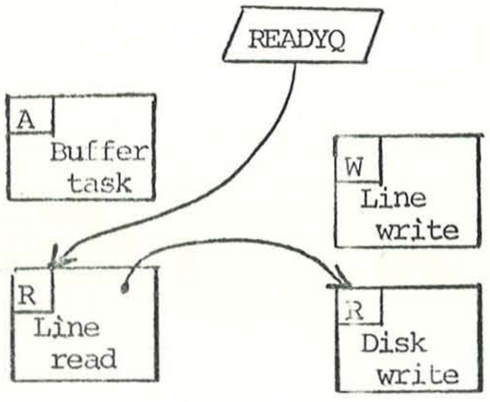
- fixed number of tasks
- predetermined priority
- dispatch algorithm scans table from top until first ready task is found
- interrupt action finds entry by key transform and changes status code

OS[5] Task Scheduler (list organization):



- variable number of tasks
- dispatch algorithm scans linked list for highest priority ready program
- when a higher priority task becomes ready, the active task is effectively "stacked" in the list
- a TCB is created and inserted by priority for a new task (dotted arrow)

QTAM[6] Dispatcher (ordered queue organization)



- fixed number of pre-defined tasks
- a chained list is maintained of ready tasks only
- dispatch algorithm selects top task on ready queue
- interrupt activity finds tasks at predetermine fixed core locations

Status codes: A = active R = ready W = waiting

Figure II.1: Examples of Task Management

[4]Disk Operating System, a control program for IBM System/360, ref.(6).

[5]Operating System/360, a control program for IBM System/360, ref.(7,8).

[6]Queued Telecommunications Access Method, an IOCS for remote terminals on IBM System/360, ref.(5).

Request element: task activity record.

A record of the existence and status of the task, which is also a request for a CPU, is created with the task. The data structure appropriate to this set of requests is influenced by the frequency of creation. If the number of tasks is relatively stable (e.g. MFT)[7], a table may be appropriate; if the number of tasks varies greatly, a variable length structure may be required.

2. Dispatching

This concerns allocation of CPUs to ready tasks.

There may be one or more CPUs in the system. If the number is small, the inventory problem is negligible. If the number is large, CPU allocation will be similar to channel allocation, discussed below.

Ready task set: The structure of the set of CPU request depends on the dispatching algorithm (t_a vs. t_i) and the order tasks relinquish the CPU (t_f vs. t_a). Typical cases of the former are:

Dispatch algorithm	Data Structure
round robin	table or list (see creation/termination)
first in first out	queue
priority	ordered queue
pre-emptive (LIFO)	stack

If there is only one CPU, the last task to receive control will be the first to release it. Otherwise, t_f will be unrelated to t_a, and an "active task set" must be organized.

Waiting task set: The logical structure of the set of tasks waiting for an event (or set of active I/O programs) depends on the order the waits will be satisfie In general, this is "random" with respect to the order of entering the wait state, bu could be constrained otherwise, e.g., by selective disabling of I/O interrupts.

B. Input-Output Management (see Figure II.2)

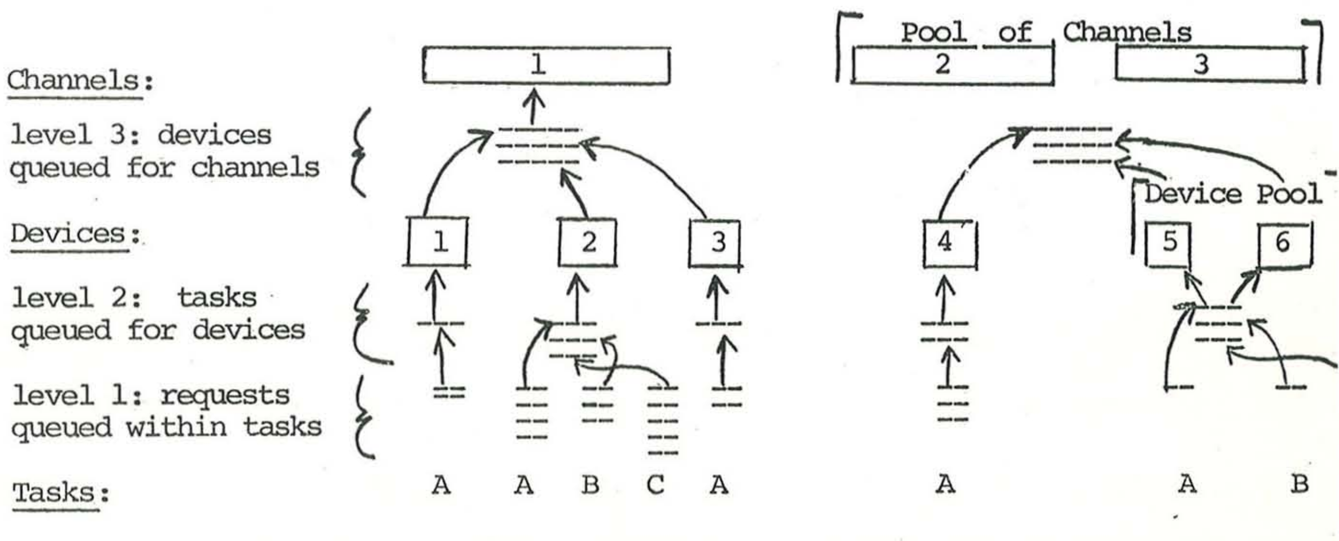
The model we are considering assumes I/O is performed by logically independent channels, a resource similar to a CPU. Hence, I/O requests are similar to tasks. Similarities have already been seen in connection with CPU inventory and waiting/active tasks.

The history of an I/O operation is characterized by (a) receipt of an I/O request, (b) allocation of a device and channel, (c) completion of the request, and (d) purging the request and notification of the requesting task. Items (a) and (d) are similar to task creation/termination.

Resource inventory: Devices, and channels (and CPUs) must be assigned to particular requests. If only a particular one may satisfy the request, the selection is trivial. If there is a pool of similar resources, they may be organized in any convenient order, since the particular choice makes no difference.

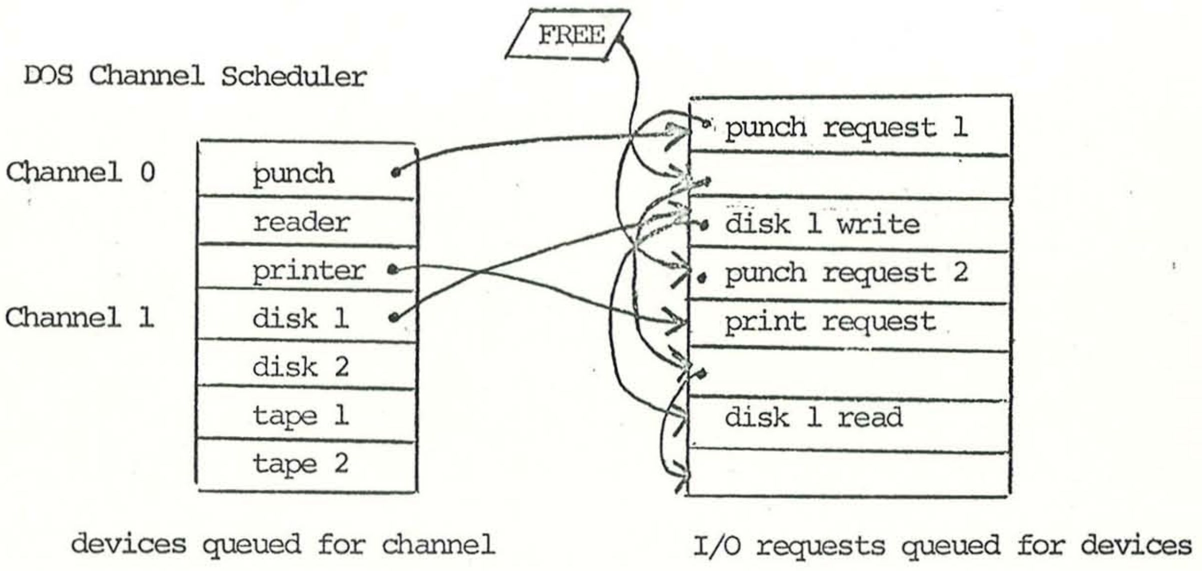
I/O requests: Request records are created by the requesting tasks. The distinctive feature of their data organization is the requirement of multi-level queuing, as illustrated below:

[7] Multiprogramming with a Fixed number of Tasks, one of the scheduling options of IBM's Operating System.



Channels:
level 3: devices queued for channels
Devices:
level 2: tasks queued for devices
level 1: requests queued within tasks
Tasks:
Pool of Channels
1
2
3
Device Pool
1
2
3
4
5
6
A
A
B
C
A
A
A
B

Figure II.2a: Multiple-level Queuing



° level one and level two queues are merged in FIFO order
° each level two queue is a chained list with head at device entry
° each level one queue is a fixed table with an entry for each device
° the channel scheduler scans the device entries in round-robin order

Figure II.2b: Example of Input-Output Management

Level 1: requests from a task for a particular device: ordinarily these requests must be queued FIFO. This requirement can be eliminated by forbidding the user to issue two requests to the same device at a time, and the queue can be merged at level 2.

Level 2: all requests for a particular device: this queue arises for sharable devices, e.g. DASD, and is typically organized to optimize access time, i.e., an ordered queue.

Level 3: all requests for the same channel: at this level, the top entry on each device queue has the characteristics of a task -- ready if the device is free, active if the channel is allocated to it, and waiting if the channel is released but the device is still busy.

NOTE: A distinct difference between I/O queuing and task queuing is that a pre-emptive (LIFO) discipline cannot be used with I/O. This is because the mechanical motion of I/O devices prevents their being "frozen" for later resume.

C. Storage Management (see Figure II.3)

Storage management problems are similar for both core and auxiliary storage. Again the appropriate organization depends on the order of t_f vs. t_a.

The simplest case is that of "automatic" storage, where requests for allocation and release of storage follow the LIFO order of subrouting calls and returns. The stack is an ideal implementation for this case.

Another simple case is that in which large requests may be satisfied by multiple non-contiguous extents, which is possible when access will always be serial, e.g. sequential DASD datasets, stacks, queues, and chained lists. A simple list organization of free storage is adequate in this case.

In the general case, the "best" storage allocation algorithm is an unsolved problem. The usual approach is to reduce the problem to one of the simple cases given above. Time-sharing machines (e.g. S/360 Model 67, GE 645) use address translation hardware to make physically discontiguous regions logically contiguous, permitting a chained-list allocation algorithm. Other systems (GE 635) use a stack allocation and relocate programs in memory to fill in holes that develop in the bottom of the stack.

D. Timer Management

The supervisor model assumes the simulation of many pseudo-timers using a single hardware interval timer. This is realistic due to the unpredictable requirements and difficulty of synchronizing many real timers.

In this case requests are inherently oriented toward an ordered queue. Requests are like tasks in that an asynchronous event is scheduled.

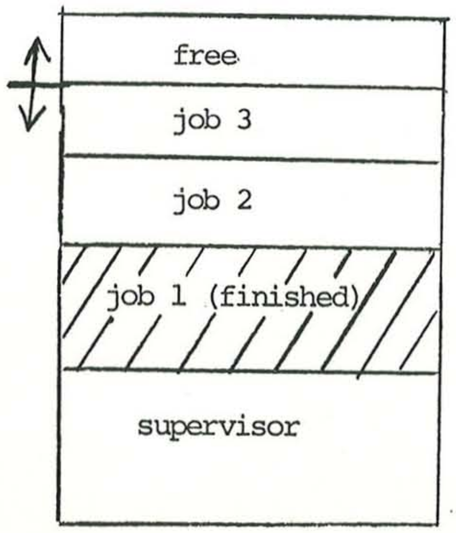
4. Data Structure for Program Control.

There are two main types of program organization appropriate to the internal control of the supervisor. They are distinguished by the order in which routines (programs) start and complete. If the program is considered to be a resource, the status information to be preserved includes the location counter and data pertinent to the program.

A. Subroutine organization: transfer to and from programs is such that routines are executed in a LIFO order. A stack organization is appropriate for saving the status of temporarily inactive routines.

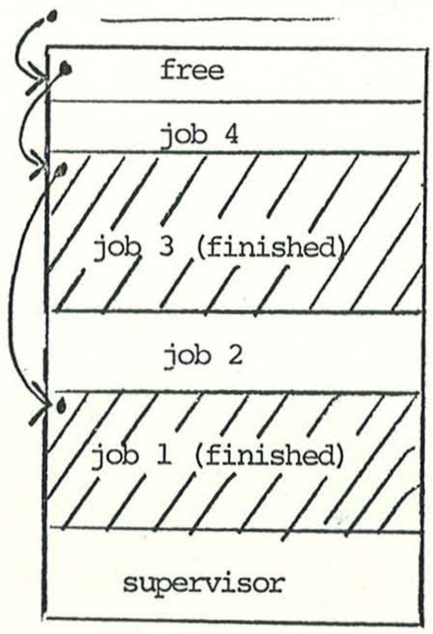
B. Task Organization: transfer of control from one routine to another depends on the sequence of external events, and may be asynchronous. Separate tasks may logically run in parallel; each requires an independent activity record for saving its status.

If each task is composed of subroutines, this leads to the requirement of multiple independent stacks.



Stack allocation (MFT-1)[8]

Storage becomes trapped at the bottom of the stack. Jobs 3 and 2 must finish before idle storage from job 1 is availabl



List organization of free core (MVT)[9]

Storage becomes fragmented. Active storage for job 2 divides a large segment of free storage from jobs 1 and 3.

Figure II.3: Examples of Storage Management

[8]Early versions of MFT, also called "Sequential Partition Scheduler" and "Option 2," operated as illustrated here.

[9]Multiprogramming with a Variable number of Tasks, another scheduling option of IBM's Operating System.

PART III: CONTROL PROGRAM FUNCTIONS ON A STACK MACHINE

I. Functions of the Stack

Recalling the basic design of the stack machine (B5500) discussed in Part I, we can summarize the applicability of the main stack functions as follows.

1. Evaluation of Polish expressions: there is no peculiar applicability of this function to supervisor programming.

2. Subroutine control: Figure III.2 illustrates the operation of the stack in a calling sequence. In this instance the stack serves as an extension of the instruction counter, not the accumulator. The program status is embedded in the stack (Mark Stack and Return Words); but note that these entires are chained together and do not rely on the TOS register for their stack organization.

3. Memory allocation: the stack implements a LIFO storage allocation algorithm for automatic variables as required by the program.

II. Extended features in the B6500/7500

Figure III.1 illustrates extensions to the basic stack design to aid certain control program requirements.

1. Task control. The extended machine allows multiple stacks for multiple tasks. The "stack vector array" holds the address of each task's stack and assists in memory inventory and task switching.
2. Shared reference to variables: A "stack" of display registers references each parameter area accessible to the current routine. This is an implementation of resource sharing (data) and maintains one branch of the task generation tree.

III. Additional Hardware Features

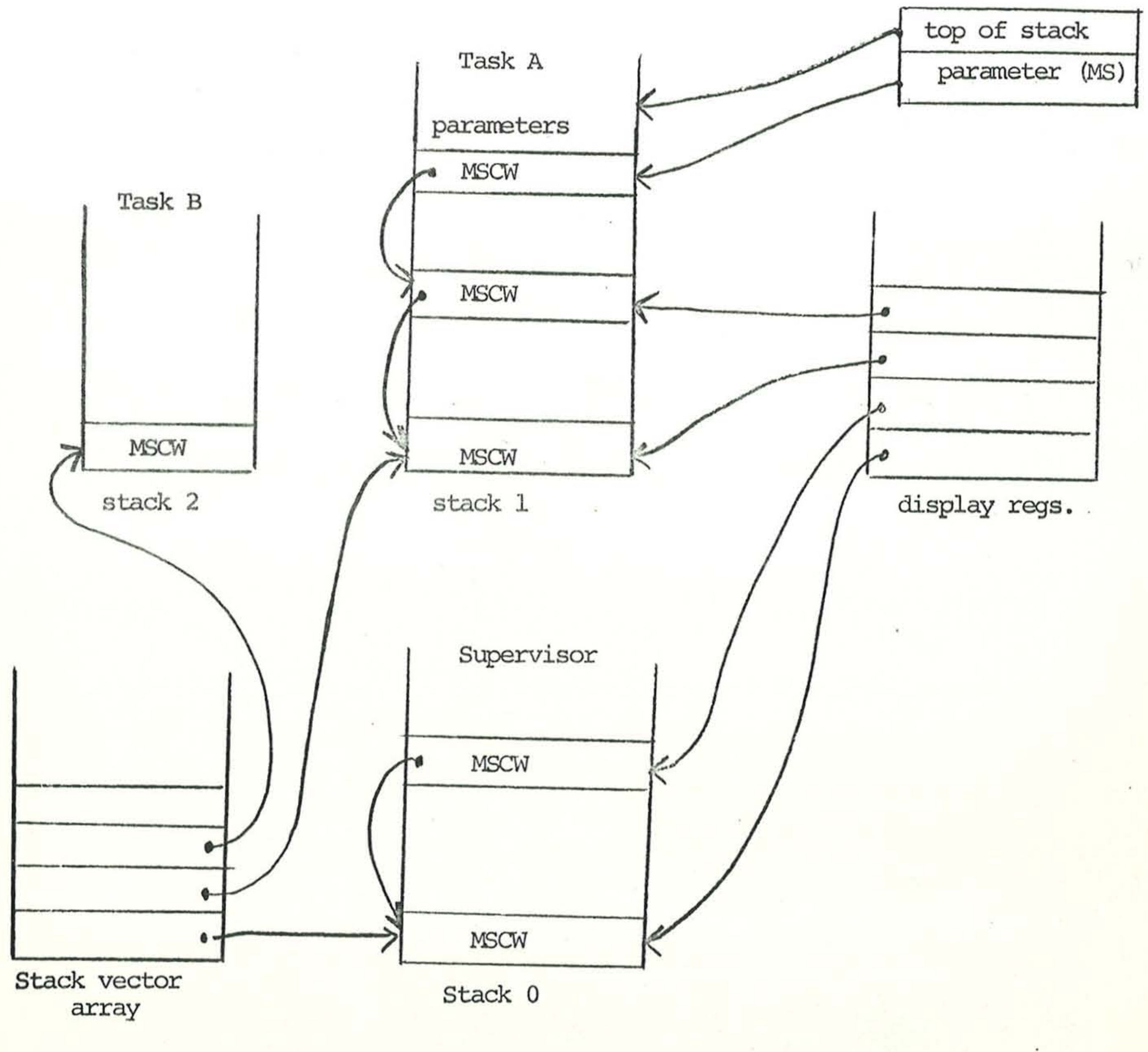
Features of the B5500 unrelated to the stack mechanism but useful in control program operations are the following:

1. Chained list search (Link List Lookup): This operation scans a chained list of data items of the form / key | link / until it finds a key greater than or equal to a reference key, and then creates a descriptor (pointer) for that element. The link may be embedded in larger structures for a generalized chained structure.
2. Table Scan (Flag Bit Search): This operation scans a vector until it finds a word with a flag bit on. This adds efficiency to table-oriented queuing and searching.

IV. Comparison with System/360

Figure III.3 illustrates equivalent S/360 programs for some Burroughs B5500 operations. Figure III.4 is a comparison of the two based on the adjusted number of main memory cycles required for equivalent operations. Note that whereas the number of accesses for data operands is comparable, the S/360 programs require many more accesses for instruction fetching, particularly in "tight loop" operations. Furthermore, a significant part of S/360 "overhead" is due to testing for exceptional conditions such as end of stack.

We would also expect programmer efficiency and debugging to be superior on the Burroughs machine due to provision of function-oriented primitive operations rather than requiring the programmer to write short instruction sequences to perform operations on set structures. On the other hand, the examples illustrate that the S/360 instruction set provides sufficient flexibility to implement any set structure we have considered, whereas the Burroughs operations are more restrictive.



Registers in extended machine:

TOS: top of stack register (temporary variable addressing)

MS: mark stack register (parameter addressing and return)

Display registers: base registers for non-local variables

Mark Stack Control Words: (MSCW) preserve the calling sequence chain

Stack Vector Array: address of base of each independent stack

Figure III.1: B6500/7500 Extensions to Basic Stack Machine

B5500 Operation

Mark Stack:

Store MSCW to mark end of callers stack.

Operand Call (or descriptor call):

Store parameters for called program.

Operand call to subroutine descriptor:

Save return address and branch to subroutine.

Literal call:

Save space in stack for local variables.

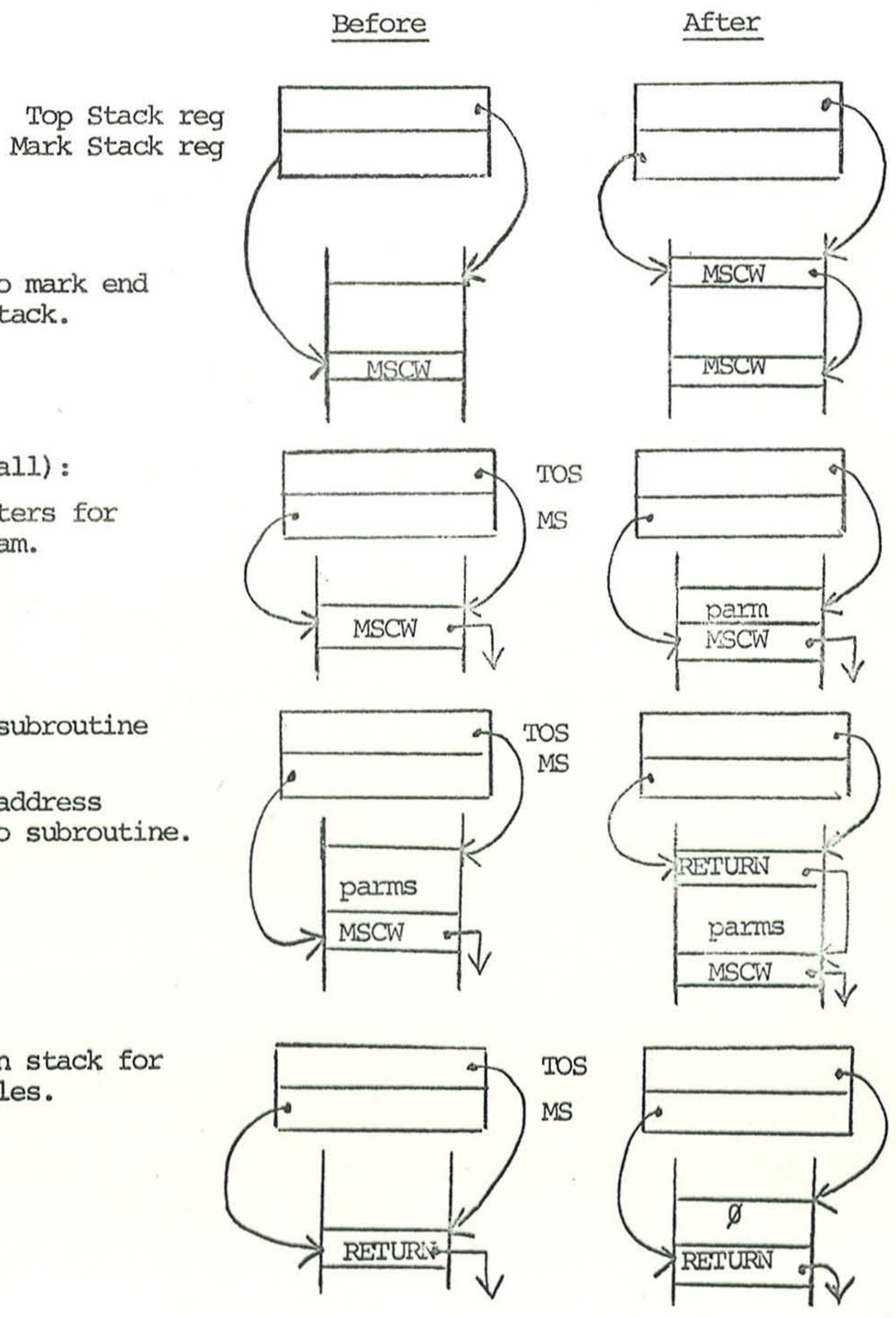


Figure III.2: B5500 Calling Sequence

B5500 Operation: Stack Pushdown

On the Burroughs stack machines, the top two (logical) positions of the stack are implemented by a pair of hardware registers A and B. Thus, the TOS register, which points to the top of the core-memory portion of the stack, actually points to the third word of the logical stack.

The basic stack pushdown operation is implicit in every "operand call" operation (equivalent to LOAD ACCUMULATOR); pushdown is equivalent to temporarily storing the contents of the accumulator prior to loading.

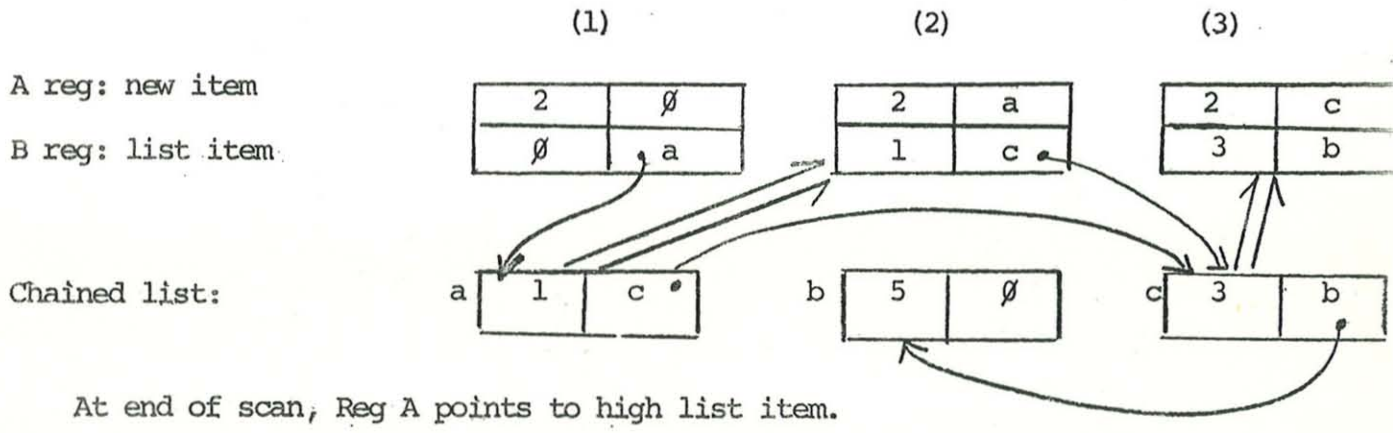
The operation proceeds as follows:

1. The TOS register is incremented to point to the next higher position (which is logically empty) in the core-memory portion of the stack.
2. The TOS register is compared with the limits registers, and an interrupt occurs when the stack memory limits are exceeded.
3. The B-register contents are stored in the new memory stack word, and the A-register contents are shifted to the B-register. The A-register is then ready to receive operands from memory.

S/360 equivalent: (let UL,LL = limit registers)

```
    LA  TOS,4           increment stack pointer
    CR  TOS,UL          test memory limit
    BH  error           if limit exceeded, cause interrupt
    ST  B,0(TOS)        store B-reg in stack
    LR  B,A             move A contents to B register
```

Figure III.3a: Pushdown Operation



At end of scan, Reg A points to high list item.

Equivalent 360 program:

Registers: A,X B,Y

A	X
B	Y

(360 must use 2 registers to simulate Burroughs registers).

```
      LA   Y              put start address in second half of B reg
loop  LR   X,Y            move link up to A reg
      LM   B,Y,0(X)       move next list item to B reg
      CR   B,A            compare keys of list item and new item
      BL   loop           continue scan if low comparison
```

Figure III.3b: Linked List Lookup

B5500 Operation: Flag bit search

This operation scans a table until it finds the first operand with a flag bit set.

Equivalent S/360 program:

Program begins with address of TABLE-4 in register A.

```
loop LA  A,4            set address to next table entry
     TM  flag,0(A)      test table entry for flag bit
     BNO loop           if bit off, repeat  for next entry
```

The program ends with the address of the flagged entry in register A.

Figure III.3c: Flag Bit Search

V. Conclusions

This survey of control-program information structures reveals a high incidence of cases where data cannot or should not be accessed in a last-in-first-out order. Hence we must conclude that stack organization, while effective for problems for which it was originally designed, is irrelevant to many of the basic processing operations of the control program.

Furthermore, even in those instances where LIFO data structures are useful in the control program, and particularly in multi-programming several tasks which may individually use a stack to advantage, the original concept of a single "semi-infinit stack must be modified. What is required is a separate stack for each task or data structure. This can be accomplished by a chained implementation of separate stacks or a chained tree structure of multiple-extent stacks.

Finally, we notice that control program efficiency can be improved by providing specialized operations to process commonly occurring data structures, such as chained lists and tables. This suggests that the success of the Burroughs design is due not alone to the stack concept *per se*, but to the efficient hardware implementation of all relevant data-structure operations.

OPERATION	MACHINE	INSTRUCTION ACCESSES	OPERAND ACCESSES	TOTAL
Pushdown	B5500	0*	1	1
	S/360	5**	1	6
Link List Lookup	B5500	1	2n	2n+1
	S/360	4n+1	3n-1***	7n
Flag Bit Search	B5500	1	n	n+1
	S/360	3n	2n-1***	5n-1

n = number of items scanned

*This operation is implicit in other operation codes

**Two of these fetches are required to test for memory bounds

***The extra operand access is required for the branch address

Figure III.4: Comparison of B5500 and S/360 Operations

PART IV: POSSIBLE HARDWARE EXTENSIONS

I. Introduction

We have seen that the basic data structure appropriate to the control program is the ordered[10] set. We might inquire whether a generalized ordered-set mechanism, on the order of the stack mechanism, would be possible.

II. Multiple Stacks, Multiple Extents

We first postulate the availability of a "stack" for each data structure we desire. This mechanism is implemented in the B6500. To avoid core fragmentation problems, we could allocate these as chained extents of a fixed size. The access mechanism (always serial) would automatically "branch" when it reached the end of an extent.

III. Queue Structure

Figure IV.1 illustrates the behavior of data in a stack vs. that of data in a queue. Note the following:

a) the stack has only one access point; the queue has two.
b) the active areas of both structures vary in size, but the location of the active area of the queue "climbs" through memory while the stack area stays "tied down."

Hence a hypothetical queue machine would require the following:[11]

a) an extra register to identify the bottom of the queue
b) a "wrap-around" mechanism so requests for high addresses would be filled with low addresses, e.g., multi-extent mechanism.

IV. Ordered Queue Implementation (see Figure IV.2)

A disadvantage of a vector representation for an ordered queue is that items must be moved for insertions and deletions. This need not be a handicap, however, since a serial key scan (also required for a chained representation) could perform the move "on the fly" by storing each operand one location higher or lower after making an unsuccessful key comparison. The data fetched and stored (key + link for chained, key + data or pointer for vector) would be the same in each case.

V. Chained List Operations

The efficiency of using chained representations could be improved by including operations like the Link List Lookup. The improved machine would have:

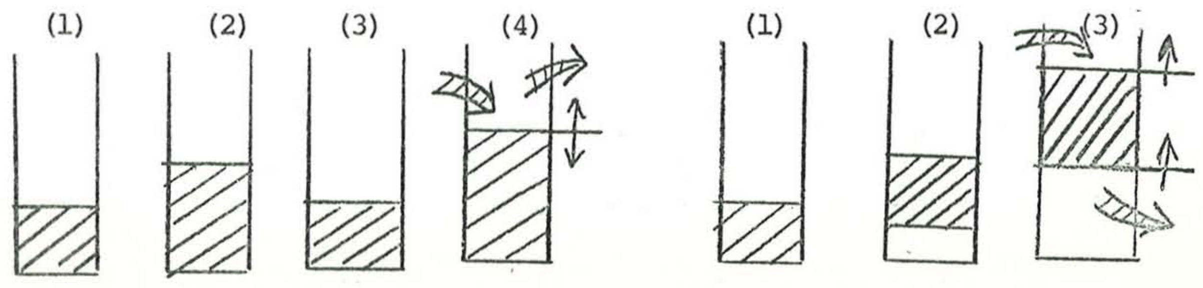
a) a special link-list format, key + link, recognized by hardware
b) link-list op-codes, such as scan, insert, delete. Of particular importance here would be condition codes or interrupts to signal exceptional conditions, such as end of list, null list, broken chain, etc.

(Note that a similar concept was involved in converting floating-point arithmetic from subroutines to hardware.)

---

[10] i.e., well-ordered under the operation of concatenation. This may be either physical contiguity or logical (chained) linking.

[11] The G.E. circular list is an example of this structure.



Shaded area represents active entries.

Stack: one boundary moves both ways

Queue: two boundaries move the same way

Figure IV.1: Comparison of Stack and Queue Operation

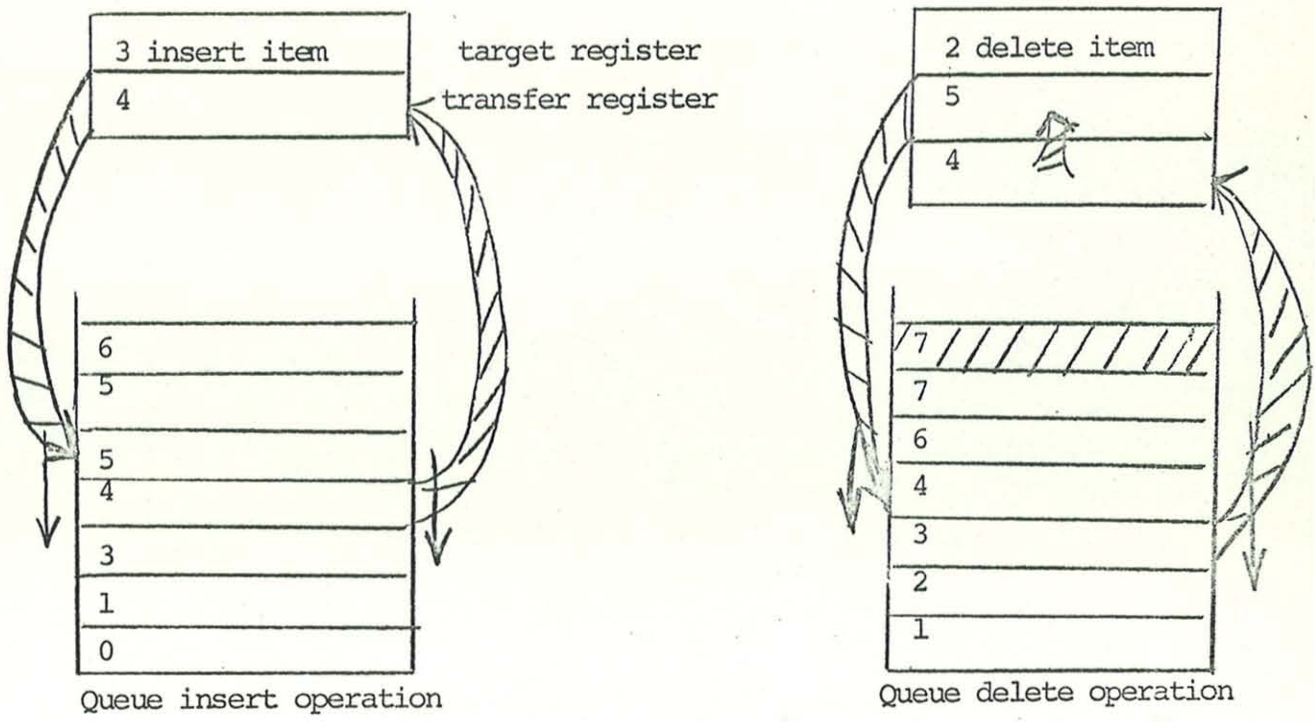


Figure IV.2: Possible Implementation of Ordered Queue Operations

PART V: CONCLUSIONS

The basic properties of stacks and their utility in algorithmic computation are well known.[12] The question that must be asked is whether this theoretical suitability carries over into an environment which is inherently multi-tasking, highly parallel, and event-driven.

The examples of solving the control program problems suggest that the stack is not the most useful mechanism. Particularly compelling is the fact that the Burroughs architecture augments the stack mechanism with hardware that is oriented toward other structures in order to implement its own control program. Accordingly, we may conclude that the predominance of data structures other than stacks, particularly ordered lists, represents a basic requirement for supervisor programming, and is not just a make-shift due to lack of stack hardware.

We may expect that, just as the stack organization emerged as the "natural" structure for Algol-type programming, a new "natural" structure will emerge to simplify the problems of supervisor programming.

For a discussion of these properties, see a recent text on advanced computer programming, such as Wegner, ref.(10).

REFERENCES

1. Brooks, F.P. and Iverson, K.E., Automatic Data Processing, John Wiley and Sons, 1963, pp. 167-77.

2. Burroughs Corporation, "Burroughs B5500 Information Processing System Reference Manual," 1964.

3. ____________________, "A Narrative Description of the Burroughs B5500 Disk File Master Control Program," 1968.

4. Hauck, E.A. and Dent, B.A., "Burroughs' B6500/B7500 Stack Mechanism," Proceedings of the 1968 AFIPS Spring Joint Computer Conference, (vol. 32).

5. IBM Corporation, "IBM System/360 Disk Operating System, Queued Telecommunications Access Method Program Logic Manual," Form Y30-5002, 1968.

6. ______________, "IBM System/360 Disk Operating System, Supervisor and Physical and Logical Transients," Form Y24-5084-3, 1968.

7. ______________, "IBM System/360 Operating System, Concepts and Facilities," Form C28-6535-2, 1968.

8. ______________, "IBM System/360 Operating System, Supervisor and Data Management Services," Form C28-6646-0, 1967.

9. ______________, "IBM System/360 Principles of Operation," Form A22-6821-6,1967.

10. Wegner, P., Programming Languages, Information Structures, and Machine Organization McGraw-Hill, 1968. (In particular, pp. 51-56, 324-332.)