Reprinted from:
Proceedings of
Fourth Australian
Computer Conference

Process Handling on Burroughs B6500

By J. C. CLEARY

Adelaide, South Australia
August 11-15th, 1969

Proceedings of Fourth Australian Computer Conference
Adelaide, South Australia, 1969

Process Handling on Burroughs B6500

By J. G. Cleary

Burroughs Corporation, Pasadena, California, U.S.A.

ABSTRACT: The approach to process handling embodied in B6500 hardware and software design and implementation is discussed in this paper. Hardware features necessary to the understanding of the approach are first described. Some aspects of the language ESPOL—an extended ALGOL language used for writing the B6500 Executive System—are presented. The representation of active and inactive processes by active and inactive stacks is discussed. Implementation of process interlocking facilities is described. Some aspects of B6500 core protection are discussed. There follows a description of the usage and implementation of events and the concept of "software interrupt" is introduced and discussed. Finally, it is claimed that the paper demonstrates how the basic B6500 design philosophy has assisted in the implementation of process handling. The emphasis throughout the paper is on description of a working system, rather than theoretical discussion.

KEY WORDS AND PHRASES: Process, processor, multiprogramming, multiprocessing, dynamic relocation, dependent process, active process, Algol, event, state vector, tree, locking, queue.
COMPUTING REVIEWS CATEGORIES: 4.20, 4.21, 4.22, 4.31, 4.32, 6.20, 6.21.

INTRODUCTION

The B6500 is designed for the multiprogramming/multiprocessing environment where the process, rather than the job or program, is considered as the basic processing unit.

This paper describes how the B6500 system controls the running of independent and dependent processes. An intuitive understanding of the meaning of the term "process" is assumed (but see Dahm, Gerbstadt and Pacelli, 1967; Dennis and Van Horn, 1966; Lampson, 1968).

SYSTEM ORGANISATION

The division of the B6500 system into hardware and software components was dictated by economic rather than technical considerations. Thus, the distinction between hardware and software functions is considered relatively unimportant, and will be noted only in this section.

The B6500 has been discussed elsewhere (Hauck and Dent, 1968; Hillegas, 1968) and only those details considered necessary to the understanding of the current subject are discussed here.

Hardware/Software Integration

The B6500 hardware has been designed to operate under the control of an executive program (MCP or Master Control Program), and to be programmed only in higher level languages (e.g. ALGOL, COBOL and FORTRAN)—there is no assembler. Thus, machine design must facilitate the implementation of fast compilers producing efficient machine code.

The command structure of the B6500 is Polish string and the instructions manipulate the contents of a stack. The stack not only facilitates the execution of Polish code, but also provides an efficient means of handling recursion.

Efficient compiling techniques typically produce a Polish string, usually as the first of two or more steps in the translation of source code to machine language; they also seem naturally biased towards the use of recursive procedures (Randall and Russel, 1964). Hence, one-pass compilation of efficient code is possible.

Instructions and Word Formats

The machine instruction is called an operator and is variable, in length—in 8 bit increments or "Syllables"—from 8 bits, for the more frequently used instructions, to 96 bits.

Data and Control words are 51 bits long. The tag, in the first three bits, identifies the various types as shown in Figure 1. The remaining 48 bits are data or control information.

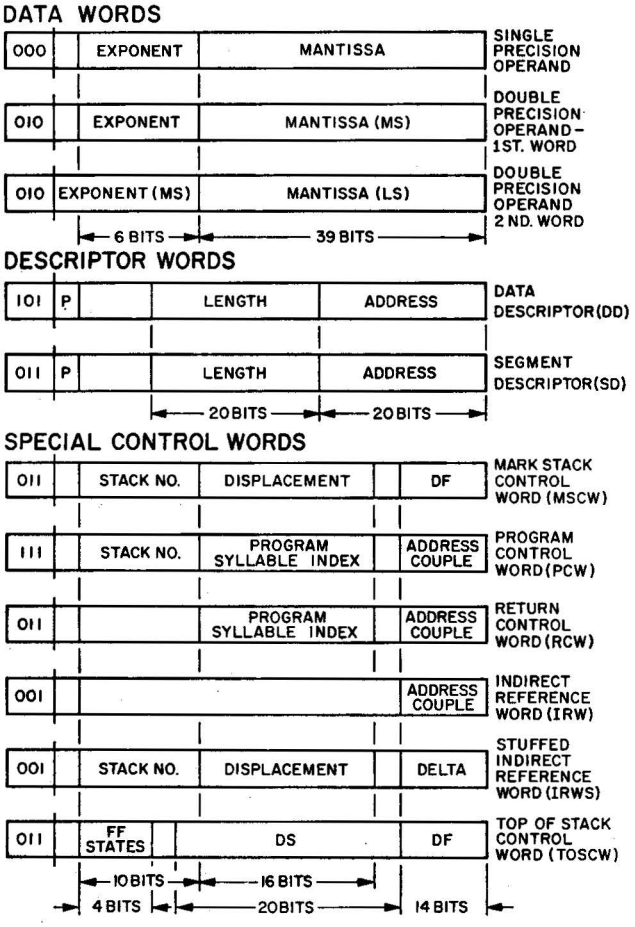

Fig. 1. B6500 Word Formats.

Stack Organisation

Each process is assigned memory for a stack in which are stored local variables, references to program procedures, data

arrays and current process status. When the process is activated, four high speed registers (A, X, B and Y) are established as the top of stack locations and register S points to the location of the last word placed in the stack memory area (See Figure 2). Registers X and Y may be regarded as double precision extensions of registers A and B and will not be considered further.

The stack operates as a last in, first out storage area. Thus, an operand is stored into register A with consequent pushdowns into register B and into the memory location pointed at by register S. Similarly, extraction of data is from register A with consequent pop-ups from B and the location referenced by S. The contents of S are incremented by one on a push-down and decremented on a pop-up.

Operators typically manipulate the A and B registers with consequent increase or decrease in stack size. The necessary push-downs and pop-ups are automatically handled by processor hardware. The Base of Stack (BOS) and Stack Limit (SL) registers define the boundaries of the process stack memory—the process is interrupted if S is set to the value of BOS or SL.

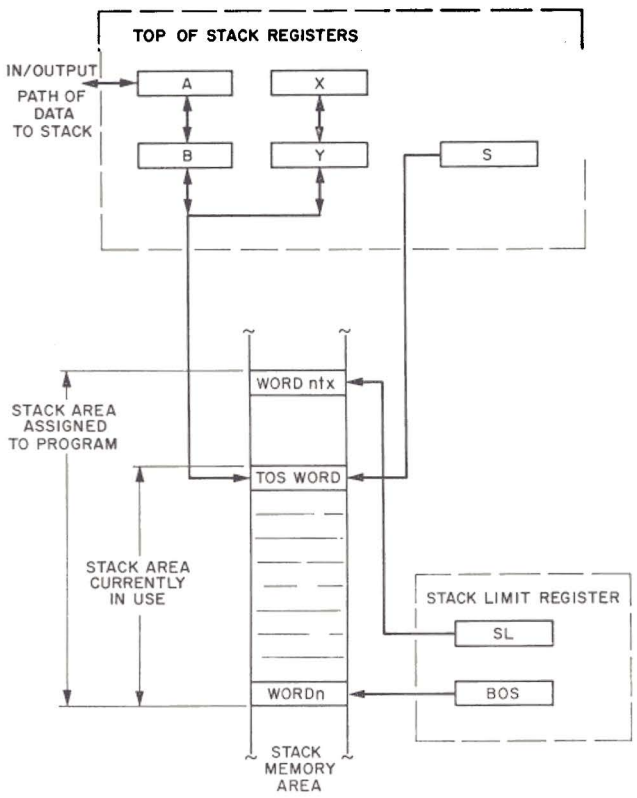

Fig. 2. Top of Stack and Stack Bounds Registers.

Addressing

Data is addressed via the Data Descriptor (DD), the Indirect Reference Word (IRW) and the Stuffed Indirect Reference Word (IRWS); code is accessed via the Segment Descriptor (SD). See Figure 1.

The Address Couple in an IRW specifies a level and a displacement—denoted for convenience as (level, displacement) or (11, δ). The level corresponds to the lexicographic level at which the referenced item is declared. Figure 4 shows how the stack is built up for the simple Algol program given in Figure 3. Note that the start of each level within the stack is marked by a Display Register (Randall and Russel, 1964) of which there are 32, corresponding to the 32 possible levels of nomenclature. Thus, V1, V2 and V3 are addressed by IRW's having the address couples (2, 2), (2, 3) and (3, 2) respectively.

```
BEGIN ————————————————— LEXICOGRAPHICAL LEVEL "2"
  REAL V1;                          ℓℓ = 2, δ = 2
  REAL V2;                          ℓℓ = 2, δ = 3
  PROCEDURE A;                      ℓℓ = 2, δ = 4
    BEGIN ———————— LEXICOGRAPHICAL LEVEL "3"
      REAL V3;                      ℓℓ = 3, δ = 2
      PROCEDURE B;                  ℓℓ = 3, δ = 3
        BEGIN ———— LEXICOGRAPHICAL LEVEL "4"
          V3 ← 3;
          V1 ← V3;
        END;
    B;
  END;
  PROCEDURE C;                      ℓℓ = 2, δ = 5
    BEGIN ————————————— LEXICOGRAPHICAL LEVEL "3"
      REAL V4;                      ℓℓ = 3, δ = 2
      PROCEDURE D;                  ℓℓ = 3, δ = 3
        BEGIN ———— LEXICOGRAPHICAL LEVEL "4"
          REAL V5;                  ℓℓ = 4, δ = 2
          V4 ← 4;
          V5 ← 5;
          A;
          V2 ← V4;
        END;
    D;
  END;
C;
END;
```

Fig. 3. Algol Program with Lexicographical Structure Indicated.

When a procedure is entered or exited, the "addressing environment" may change and hence the Display Registers may have to be reset—e.g. the address couple (3, 2) references V3 or V4 depending upon whether procedure A or C is executing. The Mark Stack Control Word (MSCW) and Return Control Word (RCW) contain information, on the lexicographic structure and dynamic history of the process, which is used in setting the Display Registers for the current addressing environment; however, a detailed discussion of this subject is beyond the scope of this paper (see Hauck and Dent, 1968).

It is sometimes necessary to access information outside the addressing environment of a procedure—i.e. in Algol terms, formal parameters must be referenced. Also referencing across stacks which are dynamically relocatable, is sometimes desirable. The IRWS (see Figure 1) locates a stack via its Stack Number, a MSCW is found at a distance above the bottom of stack given by the displacement field; the referenced item is found at a distance above the MSCW given by the delta field.

The Data Descriptor references arrays and the Segment Descriptor, code segments. In general, the address field—an absolute address—points to the starting memory (P = 1) or disk (P = 0) location of the information; an attempt to access information referenced by a Descriptor with P = 0 causes a "presence bit" interrupt and the consequent reading of the information from the disk. All addressing within the array or segment is relative to the base address and any relative address exceeding the value of the length field causes an "invalid index" interrupt.

A procedure declaration generates, within the stack, a Program Control Word (PCW—e.g. PCW-A in Figure 4). The address couple field of the PCW (see Figure 1) points to a Segment Descriptor which in turn points to the code segment for the procedure. The starting relative address of the procedure is contained in the Program Syllable Index field of the PCW. A call on the procedure generates an IRW or IRWS pointing to the PCW and thus, the relevant code is entered (see Figure 5).

The Stack Tree

An active process is represented by an active stack—i.e. a stack referenced by the S, BOS and SL registers of some processor. Stacks are used also to represent inactive processes and for other specialised purposes.

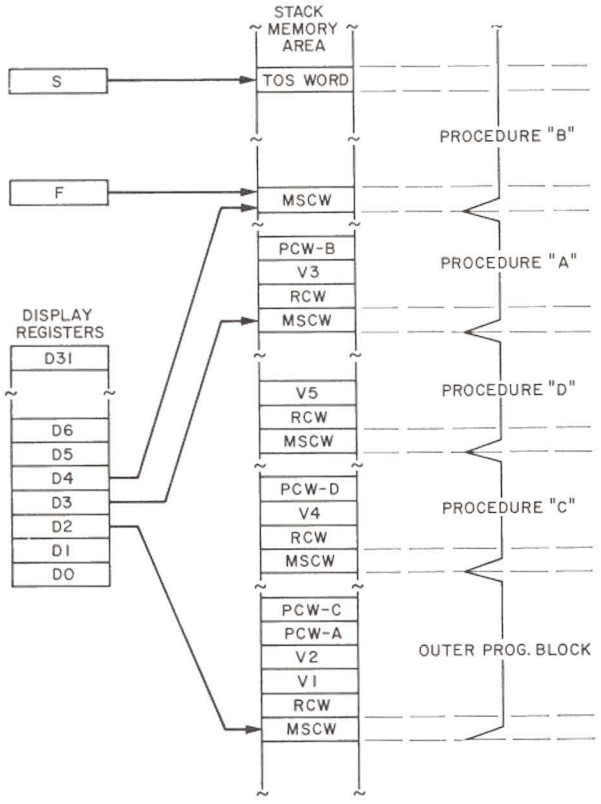

Fig. 4. Display Registers Indicating Current Addressing Environment.

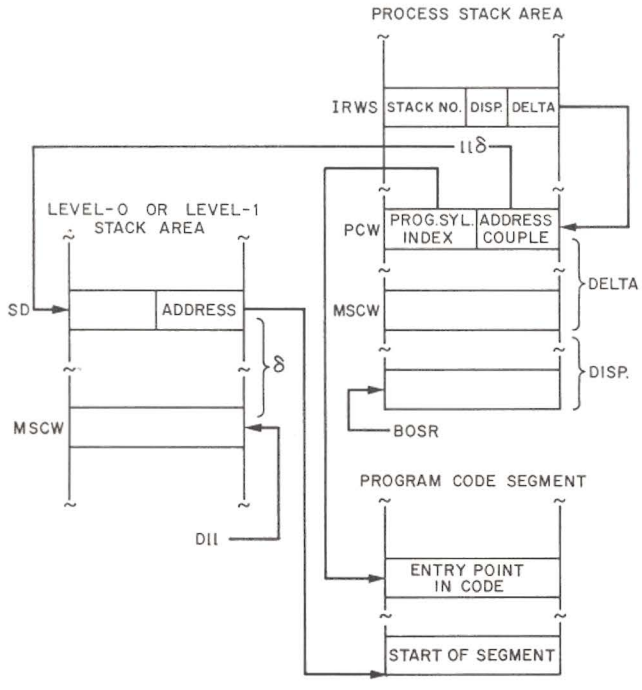

Fig. 5. Call on a Procedure. IRWS→PCW→SD→Code Area.

The simple Stack Tree—or Cactus Stack—where each job gives rise to a single process is shown in Figure 6a. The stack trunk—the level-zero stack—contains operating system global variables. For each job a level-one stack, containing all Segment Descriptors, is required. Where two or more jobs involve activation of the same code, a common level-one stack is used —all code is non-modifiable and thus re-entrant. However, jobs involving different code require creation of new level-one stacks. Finally, each process (in this case each job) is represented by a level-two stack, linked via display registers to the level-one and level-zero stacks. The level-zero and level-one stacks are inactive; the level-two stacks may be active—e.g. Stack Number Two—or inactive.

As each stack is created by the operating system, a data descriptor—the Stack Descriptor—referencing its allocated space, is entered into an array called the Stack Vector. A stack is considered as having a Stack Number—merely the relative position of its Stack Descriptor within the Stack Vector. The Stack Vector itself is referenced by the Stack Vector Descriptor, maintained in a reserved position of the level-zero stack. All references to stacks are made through the Stack Number and thus, through a Stack Descriptor no different from any other Data Descriptor—it is because of this that stacks are overlayable and dynamically relocatable.

A process may create other processes. Figure 6b illustrates the case where process L has created process M and process M has created process N. Process L and M are inactive and process N is active. Note that process N has available not only information in its own stack, but also global information in the process M, process L, level-one and level-zero stacks. On the B6500 the establishment of an independent process is equivalent to a procedure call, except that the procedure has an independent existence—i.e. has a separate stack.

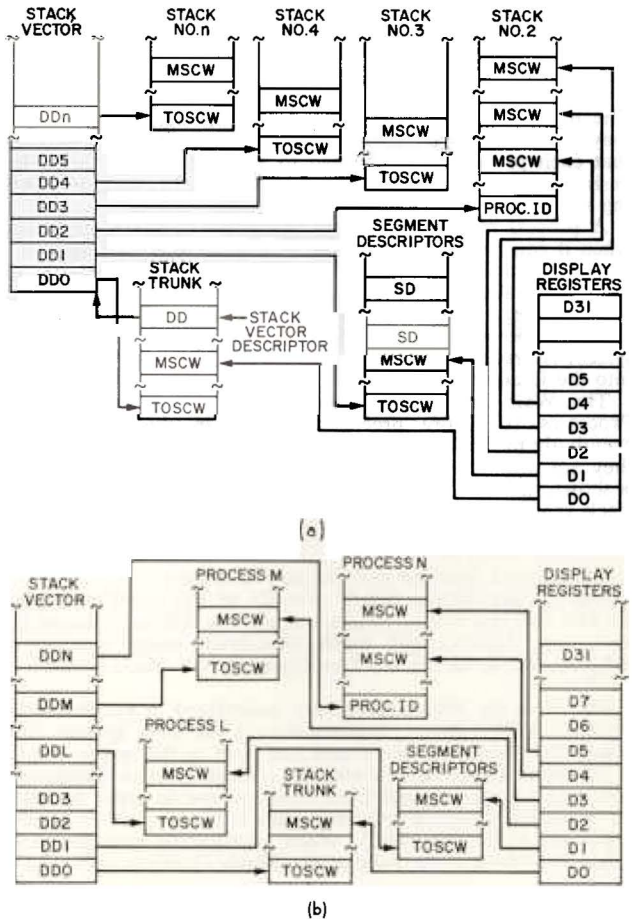

Fig. 6. (a) The Stack Tree; (b) Stack Tree with Dependent Processes.

The System Programming Language

The language ESPOL (Executive System Programming Oriented Language) is used exclusively for writing the MCP. ESPOL is an extension of Algol and is itself written in the standard B6500 Extended Algol.

It is beyond the scope of this paper to include a formal definition of the language. ESPOL facilities are introduced as needed in the following sections; however, the following features are worth noting briefly:

1. String handling facilities.
2. Memory protection overwrite facilities. Certain operations —e.g. storing into a Control Word—will usually cause a fault interrupt. These fault interrupts can be by-passed by the hardware. ESPOL code can selectively invoke or ignore the memory protection facilities.
3. Field modifications. Fields in words, any number of bits in length, may be modified or transferred to fields in other words.
4. Equivalence facilities. The same memory location may be accessed by several different identifiers of several different types.
5. Event handling.
6. Macro definition. An identifier may take on the value of a string of text. The appearance of the identifier anywhere in a program is equivalent to the appearance of the related text. A macro may be parameterised—i.e. actual parameters may be supplied with the macro identifier and will replace formal parameters in the related text.
7. Intrinsics or primitives. The appearance of an intrinsic is semantically equivalent to a procedure call; however, the procedure "declaration" is implicit—i.e. it is understood by the compiler.

It will be obvious that ESPOL is not intended for use by applications programmers; it is used exclusively for writing the operating system.

PROCESS ACTIVATION AND DEACTIVATION

The primary objective is to maximise throughput. The B6500 may have more than one central processor and it is desirable that each processor be assigned to a process at all times. It is desirable, also, that the overhead represented in process switching should be minimised.

A process may change from the active to the inactive state when it can no longer continue—e.g. when it is waiting on I/O —or when it is preempted by a process of higher priority. The stack associated with the inactive process is entered either into a "Wait Queue", behind a particular event or, on preemption, into the "Ready Queue". A process newly presented to the system is made to look as if it has been preempted and entered into the Ready Queue.

The Wait Queues are described in the section on events. Processes (or more precisely stacks) in the Wait Queues eventually move to the Ready Queue when the events on which they are waiting are caused. The structure of the Ready Queue is shown in Figure 7a where the inactive process represented by stack #500 currently has the highest, and that represented by stack #20 has the lowest priority. The queue is linked through the second word in the stack; the forward and backward links facilitate rapid insertion and deletion of entries.

The "Ready Queue Head" consists of one word containing the first and last Stack Numbers as shown. All the linkages are, of course, indirect; the Stack Descriptor being accessed by indexing the Stack Vector Descriptor by the Stack Number in the link.

Entries in the Ready Queue are maintained in priority order. Priority is determined dynamically—it includes among other things, the time that the process has been in the queue—and it is guaranteed that no process will remain inactive indefinitely. The queue is re-arranged from time to time in order to reflect the changing dynamic priorities (this is the subject of another paper yet to be published). When a processor becomes available, the process at the head of the Ready Queue is activated.

Active and Inactive Stacks

The mechanism whereby a process goes from the active through the inactive and back to the active state is best understood by considering preemption at interrupt time.

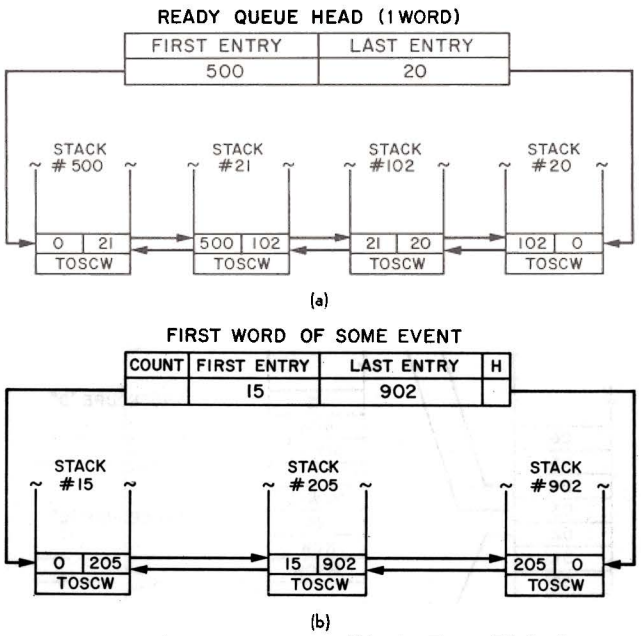

Fig. 7. (a) The Ready Queue; (b) An Event Wait Queue.

When a process—or more precisely, the processor assigned to a process—is interrupted, the hardware forces entry into an interrupt procedure. After the processing of the interrupt, the process stack is as shown in Figure 8a.

At this point, the interrupt routine checks to see if there is a process in the Ready Queue with higher current priority than the interrupted process; if so, the following occurs:

1. The active process—called hereafter "process A"—is linked into the Ready Queue.
2. The interrupt procedure executes a "Move Stack" operator. This one instruction does the following:
   (a) Stores pointers to the processor's S and F registers in the first word of process A's stack (see Figure 8b). The F register marks the topmost MSCW in the stack. The first word in the stack, which previously was a single precision operand containing the processor ID (a number 0 through 7), is now a Top of Stack Control Word (TOSCW) and is so tagged.
   The status of various processor flip-flops necessary to restore process A to the active state is also stored in the TOSCW.
   (b) The S and F registers and some processor flip-flops are set from the TOSCW for the process—"Process B"—at the head of the Ready Queue. Process B is de-linked from the Ready Queue. The Stack Descriptor, obtained via the Stack Number, for process B is used in setting the BOS and SL processor registers.
   (c) The TOSCW in process B's stack is changed to a single precision operand containing the processor ID. Process A is now inactive and process B is active and assigned the processor previously occupied by process A.
3. Note that process B was previously rendered inactive in precisely the same manner as described for process A above and its stack is now as shown in Figure 8a.

The stack for process B is now active; however, the processor is still executing code associated with the interruption of process A. To recreate process B's environment at the point of its interruption, the interrupt routine issues an Exit operator which does the following:

(a) Recreates process B's addressing environment. As mentioned earlier, the MSCW's in process B's stack are used in resetting the Display Registers. Register F points to the topmost MSCW and all the MSCW's are linked one to another (see Hauck and Dent, 1968).

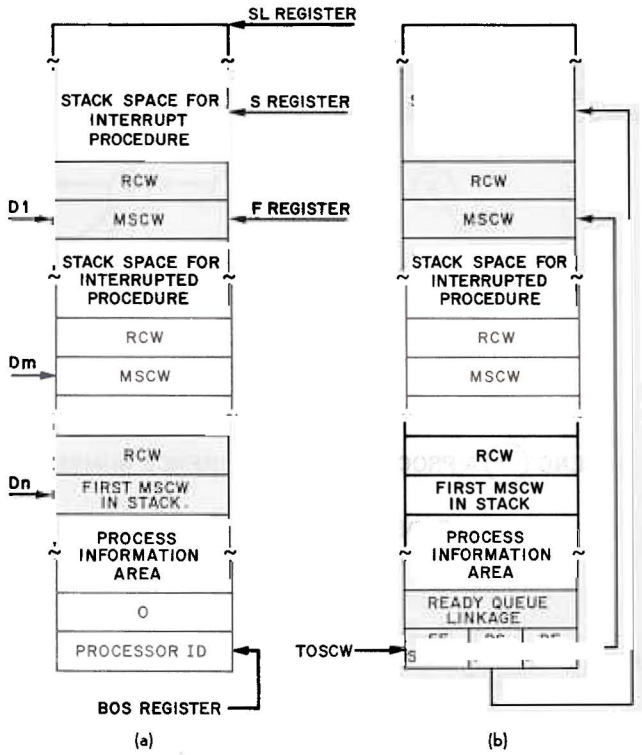

Fig. 8. (a) Active Stack; (b) Inactive Stack.

(b) Uses the topmost RCW to return control to process B's interruption point. The segment Address Couple and Program Syllable Index for the interruption point of process B's code were stored in the RCW (see Figure 1—note, an RCW is very similar to a PCW) by hardware at the time of interruption.

Note that a swap of processes between a processor and the Ready Queue; between the active and inactive states, is the essential point of the mechanism described above. Note also, that most of the work is performed by two operators—Move Stack (MVST) and EXIT. MVST is invoked in ESPOL via an intrinsic, the EXIT operator is invoked by the normal Algol procedure exit mechanism.

The Stack as the Process State Vector

A process' private information is carried entirely within its stack, thus, one may speak of a process as being represented by its stack—i.e. its level-n stack where n>1.

The stack, thus corresponds to what has been called the Process State Vector (or Process State Word—see Dennis, 1965; Dennis and Van Horn, 1966; Lampson, 1968). If one is to have processes which can be interrupted and deactivated, then one must ensure that, on reactivation, a process will continue as if never interrupted. Process state information is carried in the stack as follows:

(a) Control information necessary to restore the operating environment is held in Control Words (MSCW's, RCW's and the TOSCW).
(b) Information on dependencies with other processes is contained partly in Control Words and partly in the Process Information Area (see Figure 8).
(c) Information generated or modified by the process or pointers to such information (e.g. Data Descriptors in the case of arrays) is contained wholly within the process stack.
(d) Logging and accounting data are held in the Process Information Area.

DEPENDENT AND INDEPENDENT PROCESSES

Consider the program, written in extended Algol, shown in Figure 9a. Suppose the program initially is represented by a process active on processor 0. The statement "PROCESS A" causes procedure A to start executing as a separate process. If processor 1 is assigned to the new process—called hereafter "PROCESS A"—the Stack Tree, immediately after the initial process—"PROCESS B"—enters procedure B is shown, with the Stack Vector omitted, in Figure 9b.

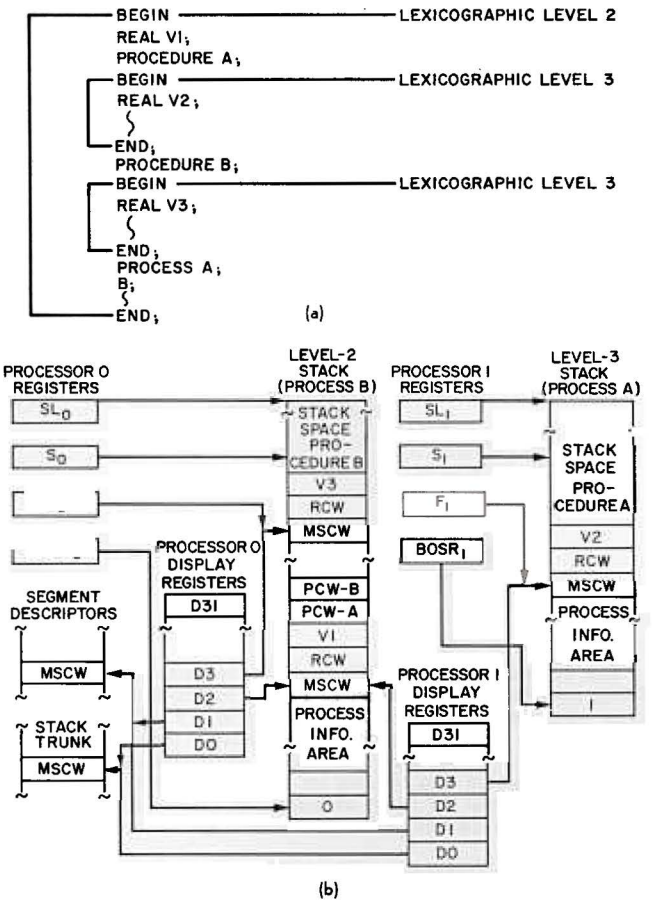

Fig. 9. (a) Program for Process Activation; (b) Multi processor Stack Tree.

Figure 9b illustrates how process activations are identical to procedure calls except that a new stack is established. It therefore follows that all the rules of scope inherent in Algol block structure apply equally to nested procedures and nested processes; e.g. V1, declared in process B is global to process A. These relationships are established by the Display Registers of the two processors—D2 for both processors points to the same MSCW in the level two stack and therefore V1 in both stacks is accessed by the address couple (2, 2). Note also, that level-one and level-zero are common to both processes.

If procedure A were declared somewhat differently, parameters could be passed to process A by a statement such as: PROCESS A (parameter 1, parameter 2 . . .). Such parameters would be accessed by Stuffed Indirect Reference Words as described above.

When process B exits, the PCW associated with procedure A is destroyed in the usual Algol block—exit fashion, hence process A will be terminated—it no longer has a creator, nor does it have a code segment. It should be noted that exit from the block where a process is *called* will not necessarily terminate that process—e.g. if procedure B were to contain the statement: "PROCESS A", then a further level-three stack corresponding to a new process—say process A'—would be created; however, process A' would not be terminated when procedure B exited. It should be further noted that termination of process A has no effect on the running of process B.

It is not always desirable that processes should destroy themselves. It may be that a creator process will synchronise itself with created processes and will determine when those dependent processes are to be terminated. Such sychronisation is usually achieved by using events. The syntax for the usage of events in extended Algol differs little from that described for ESPOL in a later section and the reader is referred to that section in order to determine how the following program gives rise to the Stack Tree shown in Figure 6b.

```
BEGIN
      EVENT E1;
      PROCEDURE M;
            BEGIN




                  HOLD;
            END;
      PROCEDURE N;
            BEGIN




                  CAUSE (E1);
            END;
      PROCESS M;
      PROCESS N;
         .

      WAIT (E1);
         .
         .
END.
```

For convenience, the outer block code may be regarded as being associated with process L. Figure 6b represents the situation when process L has reached the WAIT (E1) statement and process M has reached the HOLD statement. After CAUSE (E1) is reached, process N is terminated, however, process L—which was waiting on event E1, but has now been re-activated, may do some cleaning up before allowing itself and process M (which is suspended in a HOLD condition) to be terminated.

The Process Information Area of each stack contains links —Stack Numbers—to dependent processes. These links are to the "father", the "eldest son", the "older brother" and the "younger brother" processes (see Figure 10). It is stressed that the dynamic tree of a process family is dependent upon the sequence of process activations and not upon the location of PCW's (procedure declarations) associated with processes. Thus, the dynamic tree in Figure 10 may have been set up by a program such as:

```
BEGIN
      REAL V1;
      PROCEDURE ONE;
            BEGIN
                  PROCESS TWO;



            END;
      PROCEDURE TWO;
            BEGIN
                  PROCESS THREE;
                  PROCESS FOUR;
                  PROCESS FIVE;


                  .
            END;
      PROCEDURE THREE;    BEGIN . . . END;
      PROCEDURE FOUR;     BEGIN . . . END;
      PROCEDURE FIVE;     BEGIN . . . END;
                  PROCESS ONE;
            END.
```

Such linkages are essential if a process is to control its created processes. For example, a process may cause the termination of any or all of the processes it has initiated (via syntactical structures not important to this discussion). These links are also necessary when an operator requires the system to terminate a process and all its successors.

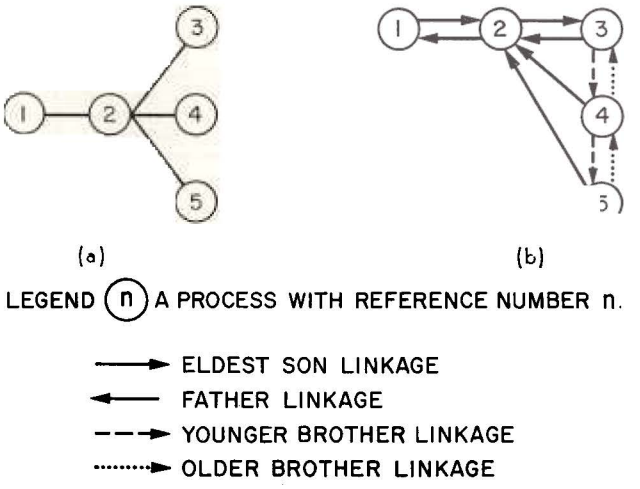

Fig. 10. Process Activation Links. (a) Dynamic Tree of Process Family; (b) Process Family Linkage.

The emphasis in this section has been upon the manipulation of processes in the extended Algol language. However, all compilers—ALGOL, FORTRAN and COBOL at present—PLI and others in the future—are themselves written in extended Algol; hence, the asynchronous and synchronous, dependent and independent process facilities required by these various languages will be available.

LOCKING

Different processes may access the same resources—code segments, files, queues, core storage, etc. It is sometimes necessary, when one process is manipulating a particular resource, that other processes should be prevented access—the resource is said to be locked. One may consider two types of lock—though the distinction is probably not important—one where a section of re-entrant code is locked, thus preventing execution of that code by more than one process, and the other where individual resources are locked. In either case, the lock must be global to all processes accessing it.

Locking Facilities in ESPOL

Locking facilities are implemented in ESPOL by the Lock Intrinsics: LOCK (<Lock Entity>), UNLOCK (<Lock Entity>), BUSY (<Lock Entity>), BUZZ (<Lock Entity>).

The Lock Entity may be a simple variable or a subscripted variable.

LOCK, UNLOCK and BUSY are type BOOLEAN intrinsics and return the value TRUE, as follows:

1. LOCK and BUSY—if the entity was previously locked,
2. UNLOCK—if the entity was previously unlocked.

LOCK leaves the entity locked, UNLOCK leaves the entity unlocked. For example, consider the following ESPOL code section:

```
IF LOCK (X) THEN GO TO SOMELABEL;
MANIPULATERESOURCE1;
MANIPULATERESOURCE2;
UNLOCK (X);
```

X—which has been declared as a global REAL—is here used as a lock on the code (shown as two procedure calls) up to UNLOCK (X). If X is not previously locked, then LOCK (X) locks it and the program proceeds, otherwise the code starting at SOMELABEL is executed. Note that UNLOCK can be used as an untyped procedure (as can also LOCK though this is not illustrated).

The code:

```
IF BUSY (X) THEN GO TO SOMELABEL;
MANIPULATERESOURCE1;
MANIPULATERESOURCE2;
```

might be used by a process which can access a resource—in this case, the code—providing some privileged process has not locked out all other processes. BUSY does not lock the resource.

It is often the case that a process can not continue until it can access a resource which is locked by some other process; it must, therefore, suspend operation until the lock is released. It may wait on an event e.g.:

```
IF LOCK (SOMELOCK) THEN WAIT (SOMEEVENT);
```

as explained below, this causes the process to go inactive until the event is caused; a new process is activated and the processor does not idle. However, for those cases where there is a probability that the processor time involved in activation and deactivation of a process will exceed the time elapsing before the locked resource becomes available, a facility for allowing a process to spin or "buzz" is required, e.g.:

```
BUZZLABEL: IF LOCK (READYQ) THEN GO
                BUZZLABEL;
```

So long as READYQ is locked, the process will not proceed; when READYQ is unlocked—by some other process, LOCK (READYQ) immediately relocks it and returns the answer TRUE and hence, the process continues. Precisely the same effect is achieved by:

```
BUZZ (READYQ);
```

A process encountering this statement will not continue until READYQ is unlocked, at which point it will relock READYQ and continue. It is guaranteed that if several processes are buzzing READYQ "simultaneously" only one will go through the lock and the others will spin until the one in control of READYQ executes the statement UNLOCK(READYQ), at which point another spinning process will go through the lock.

It is often difficult to decide whether to use WAIT or BUZZ in a particular case. However, it appears that both facilities are indispensable (though this contention may be disputed—see Lampson, 1968).

The Read With Lock Operator

The statement:

```
SPIN: IF LOCK (X) THEN GO SPIN;
```

does the following in machine code:

1. Loads a pointer to X (an IRW) into the A register and an operand with value 1 into the B register.
2. Executes a "Read With Lock" (RDLK) instruction which swaps the contents of the word pointed at by the A register with the contents of the B register. Thus, X is now 1.
3. Checks the B register. If it is a 1 (i.e. if X was previously 1) the statement is repeated, otherwise the next statement is executed.

X is considered locked when it has the value 1 and unlocked when it has the value 0. If the testing of the value of X and the setting of the value of X to 1 were two distinct operations, it would be possible for two or more separate processes to pick up X simultaneously, to find it unlocked, to lock it and then to proceed—this is precisely the situation to be avoided. RDLK ensures, because of the swap (which takes only one memory cycle) that only one process, out of any number "simultaneously" executing RDLK, will find X unlocked.

CORE PROTECTION

Memory protection is insured as follows:

1. All programming—including software systems programming—is performed in higher level—Compiler—languages. The programmer has very little control over machine code.
2. The use of Descriptors prevents access to core areas outside those assigned to a process.
3. Each word of core has three tag bits (see Figure 1). Using these control bits, single precision data, double precision data. Descriptors and Control Words are distinguished by the hardware and any attempted incorrect usage (e.g. accessing a Control Word as data) causes an interrupt.
4. Parity checking on information transfer.

EVENTS

Events are quantities which record occurrences. Typically, one process will note the occurrence—will "cause" the event—and one or more other processes will take appropriate action.

In ESPOL, event identifiers are declared in much the same manner as real, integer or boolean identifiers, e.g.:

```
EVENT E1, E2, . . . EN;
```

One may also declare event arrays—they are not however important to this discussion. Event identifiers have normal ESPOL scope and one may speak of local, global and formal events.

Events are made use of in ESPOL by means of the Event Intrinsics:

CAUSE (<Event Designator>), WAIT (<Event Designator>),
RESET (<Event Designator>), HAPPENED (<Event Designator>).

For example, consider an I/O request by an applications program. Such a request generates a call on an MCP procedure which contains the following code:

```
BEGIN
      EVENT E;

      RESET (E);
      IOREQUEST (PARAMETER 1, PARAMETER 2,
        . . . E);
      WAIT (E);
      .

END;
```

The IOREQUEST procedure puts a pointer—an IRWS—to the event into a queue associated with the particular I/O unit referenced. The WAIT statement suspends the process—sends it to sleep. At some time in the future, another process will be interrupted when the particular I/O is complete. The hardware will force entry into the interrupt routine which will look at the queue associated with the unit causing the interrupt. The pointer to the event will be accessed and the interrupt procedure will execute the statement: CAUSE (PE) where PE is the event pointer. This will cause the process which initially made the I/O request to wake-up. Note the two processes involved —one issues the I/O request and goes to sleep, the other notes (via an interrupt) that the I/O is complete and causes the first to wake up.

Any number of processes may wait on a particular event; they will all be awakened when that event is caused.

CAUSE sets the state of an event to "happened", RESET to "not happened". The boolean intrinsic HAPPENED tests this state—it is TRUE if the event has happened, FALSE if it has not happened. HAPPENED is not particularly useful; the statement:

```
IF NOT HAPPENED (E) THEN WAIT (E);
```

saves a little time when E has happened, however, the WAIT mechanism itself would recognise this happened state—it would not send the process to sleep—and hence, the above statement is equivalent to:

```
WAIT (E);
```

It should be noted that once an event has occurred—has been caused—it has theoretically happened for all time. However, it is not feasible to assign a different identifier to each expected event and the same identifier will be used to note many occurrences (in the example above, E will be used in many I/O requests); hence, the necessity for RESET.

Software Interrupts

The Software Interrupt in ESPOL provides for the interruption of a process when a particular event is caused. Software interrupts are declared via the Interrupt Declaration whose syntax, in Backus-Naur form, is given below:

<Interrupt Declaration>:: = INTERRUPT <Interrupt List>

<Interrupt List>:: = <Interrupt Segment>/<Interrupt List>, <Interrupt Segment>
<Interrupt Segment>:: = <Interrupt Identifier>:<On Part>
<On Part>:: = ON <Event Designator>, <Interrupt Statement>
<Interrupt Statement>:: = <Statement>

Consider the following code:

```
BEGIN
INTERRUPT TIMER: ON CLOCKTICK, IF CONDITION
                          THEN BEGIN HANDLE-
                       CONDITION; DISABLE (TIMER)
                       END;
  ENABLE(TIMER);
  .
  .
  .
END;
```

CLOCKTICK is a global event (whose declaration is not shown here) which is caused at fixed time intervals—the interval timer interrupt is used for this purpose. All processes with enabled Software Interrupts referencing CLOCKTICK are interrupted when it is caused. In the example, the process checks, at each CLOCKTICK that the CONDITION (a procedure or macro identifier) has occurred. If it has, the procedure HANDLECONDITION is called and the interrupt is disabled —further CLOCKTICKS will not cause interruption. Otherwise the process resumes at its point of interruption and will be interrupted by the next CLOCKTICK. A Software Interrupt does not become active until referenced by an ENABLE intrinsic.

Complex Sleeps

Statements such as: WAIT (E1 AND E2); WAIT (E3 OR E4) sometimes known as "complex sleeps", are not allowed in ESPOL. However, consider the following code (where E1—E4 are considered to have been declared globally):

```
BEGIN
      INTERRUPT I1: ON E3, DISABLE (I1, I2),
                    I2: ON E4, DISABLE (I1, I2);
      WAIT (E1); WAIT (E2); COMMENT WAIT
        (E1 AND E2);
      ENABLE (I1, I2); HOLD; COMMENT WAIT
        (E3 OR E4);
L1:
END;
```

Two consecutive WAIT statements are the obvious way to handle the "AND" condition.

HOLD sends the process to sleep unconditionally and it will be awakened only when a Software Interrupt is directed at it. Should either E3 or E4 be caused (the "OR" condition), the process will wake up and immediately disable I1 and I2 thus preventing further Software Interrupts. The process will resume at the statement labeled L1.

Implementation of Events and Software Interrupts

An event takes two words in a stack—it is a double precision operand so far as the hardware is concerned. The format of the first word is shown in Figure 7b where three processes are all waiting on the same event. The processes have been entered into the event's Wait Queue via WAIT intrinsic call. The CAUSE intrinsic will de-link the three process stacks from the event's Wait Queue and link them into the Ready Queue—that is, it will "wake up" the processes associated with the stacks. Eventually the processes will be activated when their stacks reach the head of the Ready Queue. A process may be present in either a Wait Queue or the Ready Queue, but not both. The one-bit field "H"—the happened bit—is set by CAUSE.

The second word of an event is essentially a Stuffed Indirect Reference Word (however, it has the "double precision" tag) pointing to the first process stack in an "Interrupt Queue" (see Figure 11). An interrupt declaration generates two words in the stack—a PCW referencing code for the Interrupt Statement and an IRWS pointing to the next process stack in the Interrupt Queue. ENABLE sets and DISABLE resets a bit—the "Enable State Bit"—in the PCW. A process stack may be linked into any number of Interrupt Queues.

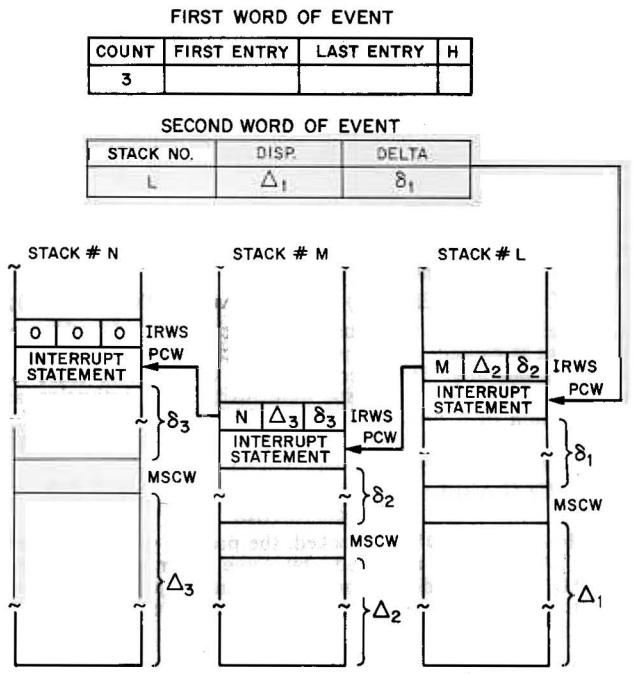

Fig. 11. Event Interrupt Queue with Three Entries.

A CAUSE (E) intrinsic call searches the Interrupt Queue for E, using the count field in the first word of E to determine the number of entries. The first stack in the queue is located via the second word in E and each subsequent stack is located via the IRWS in its predecessor. For each process stack the following occurs:

1. If the Enable State Bit is off, the interrupt is not applicable and the next stack is accessed.
2. Otherwise the IRWS from the predecessor stack is entered, together with the process Stack Number, into the "Interrupt Statement Queue".
3. The processor handling the CAUSE statement interrupts all other processors by executing the privileged operator "HEYOU". Thus, the process represented by the process stack in question is interrupted if it is active. The hardware interrupt procedure—which recognises the "Processor to Processor Interrupt" generated by HEYOU—picks up the IRWS from the Interrupt Statement Queue and causes entry into the code associated with the Interrupt Statement. Eventually, the interrupted process will return, both from the Interrupt Statement and the hardware interrupt procedure, to its point of interruption. The whole sequence is very similar to the handling of hardware interrupts—hence, the use of the term "Software Interrupt".

The above applies when the process to be interrupted is active; however, an inactive process cannot be interrupted by the operator HEYOU; moreover, during its deactivated period, more than one Software Interrupt may be directed at such a process. Thus, when a process is re-activated, one or more entries in the Interrupt Statement Queue, placed there as in (2) above, may indicate one or more outstanding Interrupt Statements and the process will execute all such statements before resuming at the point where it was deactivated.

The Intrinsic HOLD

HOLD unconditionally deactivates a process. During a CAUSE action, a process encountered in an Interrupt Queue, which is in the HOLD condition, is entered into the Ready Queue.

CONCLUSION

B6500 system design is based no certain basic beliefs, among them:

1. The desirability of the integration of hardware and software design.

2. The utility of compiler languages in the writing of programming systems.
3. The utility of the stack machine in systems implementation and—
4. The relevance of Algol block structure to systems implementation.

This paper has attempted to demonstrate how this design philosophy has assisted in the implementation of process handling in an available commercial system.

ACKNOWLEDGEMENTS

Many people have contributed, over a number of years, to the design of B6500 systems. Among the people directly concerned with the work described in this paper, special mention should be made of: B. A. Creech, B. A. Dent, E. A. Hauck, B. M. Miyakusu, W. C. Price, D. L. Kunker and R. Patel of Burroughs Corporation; R. S. Barton and D. M. Dahm, Consultants. Certain figures were reproduced from the paper by Hauck and Dent (1968) and thanks is due to these authors and to the Thompson Book Company for permission to use the diagrams.

REFERENCES

DAHM, D. M., GERBSTADT, F. H. AND PACELLI, M. M. (1967): "A System Organization for Resource Allocation", Comm. Assoc. Comp. Mach., Vol. 10, No. 12, p. 772.

DENNIS, J. B. (1965): "Segmentation and the Design of Multiprogrammed Computer Systems", Journal Assoc. Comp. Mach., Vol. 12, No. 4, p. 589.

DENNIS, J. B. AND VAN HORN, E. C. (1966): "Programming Semantics for Multiprogrammed Computations", Comm. Assoc. Comp. Mach., Vol. 9, No. 3, p. 143.

HAUCK, E. A. AND DENT, B. A. (1968): "Burroughs B6500/B7500 Stack Mechanism", Proceedings 1968 Spring Joint Computer Conference, p. 245. Thompson Book Company, Inc., Washington, D.C.

HILLEGAS, J. R. (1968): "Auerbach on Computer Technology: Burroughs Dares to Differ", Data Processing Magazine, July, 1968, issue, p. 40.

LAMPSON, B. W. (1968): "A Scheduling Philosophy for Multiprocessing Systems", Comm. Assoc. Comp. Mach., Vol. 11, No. 5, p. 347.

RANDALL, B. AND RUSSEL, L. J. (1964): "ALGOL 60 Implementation". A.P.I.C. Studies in Data Processing, No. 5, Academic Press, Inc., London, New York.

1051281

Printed in U. S. America