For example, a call causes conditions in the calling procedure before the call to be stored in the markstack in the following manner:

```
MarkStack DyNamic link (MSDYNL)  <-- MP
  "   "   IPC(MSIPC)  <-- IPC
  "   "   SEGment Pointer(MSSEG)  <-- SEGB
```

The Pascal declaration for a "markstack" is:

```
TYPE    MSCW = PACKED RECORD  { MARK STACK CONTROL }
                 MSSTAT: MSCWP;          { LEXICAL PARENT POINTER }
                 MSDYNL: MSCWP           { PTR TO CALLER'S MSCW }
                 MSIPC: INTEGER;         { BYTE INX IN RETRN CODE SEG }
                 MSSEG: BYTE;            { SEG # OF CALLER CODE }
                 MSFLAG: BYTE            { CURRENTLY UNUSED }
               END  {MSCW };
```

In addition a Static Link field becomes a pointer to the data segment of the lexical parent of the called procedure. In particular, it points to the Static Link field of parent's markstack. After the building of the data segment new values for IPC, SEGB, SP, and MP are established.

```
program pascalsystem;

var syscom : syscomrec;
    ch : char;
  segment procedure userprog;
      begin
        .
        .
        .
      end;

  segment procedure syscode;

      segment procedure printerror;
          begin
            .
            .
            .
          end;
```

```
segment procedure initialize;
    begin
      .
      .
      .
    end;

segment procedure getcmd;
    begin
      repeat
        case ch of
          'e':  editor;
          'f':  filer;
          'l':  linker;
          'x':  execute;
          'c':  compiler;
          . . .
        end { case }
      until false;
    end;
begin { syscode }
  initialize;
  getcmd;
end   { syscode };

segment procedure cspcode;
    begin
      ioinit;
      syscode;
    end;

begin (* pascal system *)
  cspcode;
end.
```

Figure 3-18. Structure of the Operating System

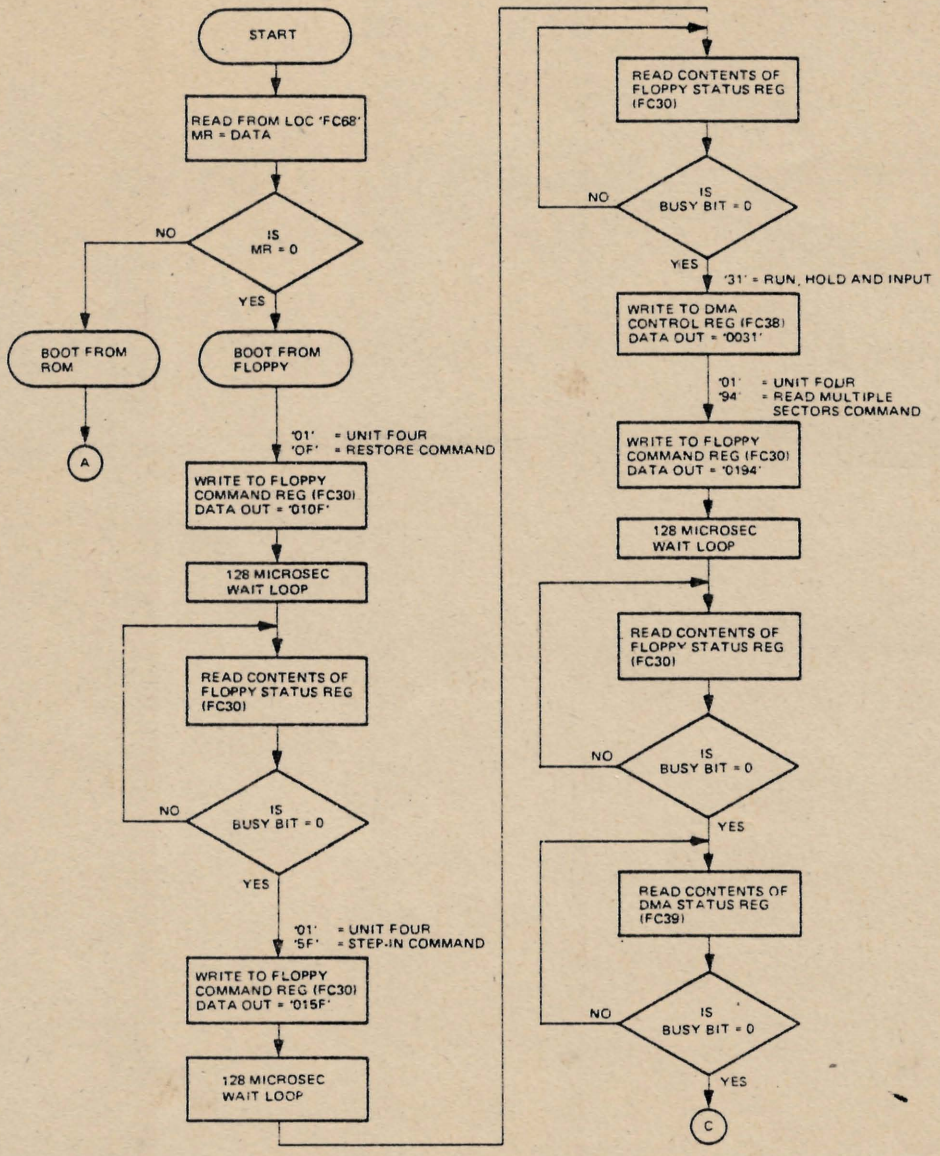
The operating system consists of a code file containing several segments and operating system tables. Some segments of the operating system are always resident. These include segment 0, the PASCALSYSTEM; 2, SYSCODE; 3, CSPCODE. Segments 4, PRINTERROR; 5, INITIALIZE; 6, GETCMD are overlaid. When a user program executes, only segments 0, 2, and 3 are resident. With only segments 0, 2, 3 resident, approximately 22,000 words of memory are available.

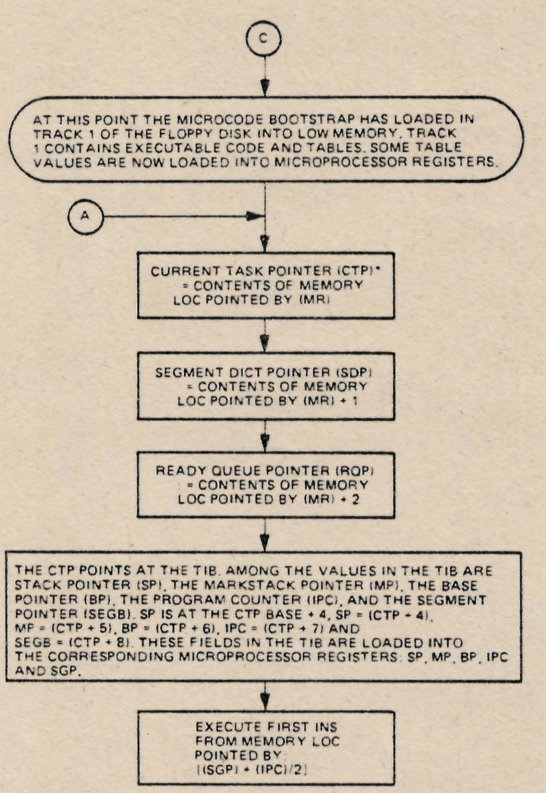
The compiler, editor, and filer are large programs that have their own code segments. During a program's execution, memory usage consists of the programs in-core code segments and the operating system's resident code and tables. When the compiler is loaded into memory in non-swapping mode, approximately 5200 words are available for use as symbol table space. In

swapping mode, this figure increases to 7300 words. When the editor is loaded in memory, approximately 11,000 words are available for text file editing. When the filer is loaded in memory, about 9000 words are available as buffer space.

3.7.2 The Bootstrap Sequence

The bootstrap sequence is initiated whenever the RESET button is pushed. The following is a flowchart describing the microcode/software instructions that are executed in order to load in and start the execution of the operating system.


START
READ FROM LOC 'FC68'
MR = DATA
IS
MR = 0
NO
YES
BOOT FROM
ROM
BOOT FROM
FLOPPY
A
'01' = UNIT FOUR
'0F' = RESTORE COMMAND
WRITE TO FLOPPY
COMMAND REG (FC30)
DATA OUT = '010F'
128 MICROSEC
WAIT LOOP
READ CONTENTS OF
FLOPPY STATUS REG
(FC30)
IS
BUSY BIT = 0
NO
YES
'01' = UNIT FOUR
'5F' = STEP-IN COMMAND
WRITE TO FLOPPY
COMMAND REG (FC30)
DATA OUT = '015F'
128 MICROSEC
WAIT LOOP
READ CONTENTS OF
FLOPPY STATUS REG
(FC30)
IS
BUSY BIT = 0
NO
YES
'31' = RUN, HOLD AND INPUT
WRITE TO DMA
CONTROL REG (FC38)
DATA OUT = '0031'
'01' = UNIT FOUR
'94' = READ MULTIPLE
SECTORS COMMAND
WRITE TO FLOPPY
COMMAND REG (FC30)
DATA OUT = '0194'
128 MICROSEC
WAIT LOOP
READ CONTENTS OF
FLOPPY STATUS REG
(FC30)
IS
BUSY BIT = 0
NO
YES
READ CONTENTS OF
DMA STATUS REG
(FC39)
IS
BUSY BIT = 0
NO
YES
C



*THE CURRENT TASK POINTER POINTS AT THE TASK INFORMATION BLOCK (TIB) THE LAYOUT OF THE TIB IS AS FOLLOWS.

```
TIB = RECORD (TASK INFORMATION BLOCK)
      REGS: PACKED RECORD
            WAITQ: TIBP; (QUEUE LINK FOR SEMAPHORES)
            PRIOR: BYTE; (TASK'S CPU PRIORITY)
            FLAGS: BYTE; (STATE FLAGS . . NOT DEFINED YET)
            SPLOW: INTEGERP; (LOWER STACK POINTER LIMIT)
            SPUPR: INTEGERP; (UPPER LIMIT ON STACK)
            SP: INTEGERP; (ACTUAL TOP-OF-STACK POINTER)
            MP: MSCWP; (ACTIVE PROCEDURE MSCW PTR)
            BP: MSCWP; (BASE ADDRESSING ENVIRONMENT PTR)
            IPC: INTEGER; (BYTE PTR IN CURRENT CODE SEG)
            SEGB: CODESEG; (PTR TO SEG CURRENTLY RUNNING)
            HANGP: SEMP; (WHICH TASK IS WAITING ON)
            XXX: INTEGER; (NOT USED)
            SIBS: SIBVEC (ARRAY OF SIBS FOR 128 255)
          END (REGS);
      MAINTASK: BOOLEAN;
      STARTMSCW: MSCWP
    END (TIB);
```

THE primitive software bootstrap loaded from track 1 now begins execution. It loads in track 0 of the floppy. The execution of track 0 which was just loaded starts the loading of the operating system. Segments 0 and 3 of the operating system are loaded into upper memory. These segments contain the I/O drivers for the operating system. At this point the operating system starts execution. It loads in segment 2 and segment 5 also into upper memory. Segment 5 performs I/O initialization. Then segment 6 is loaded into upper memory and the operating system command prompt appears. The operating system is now ready to accept user commands.

3.7.3 Registers and Operating System Tables

All registers in the Microengine are referenced by register number. The available registers and their numbers are:

-3	Ready Queue Pointer [RQP]
-2	Segment Vector Pointer [SDP]
-1	Current Task Pointer [CTP]
2	Lower Stack Pointer Limit [SPLOW]
3	Upper Limit of Stack [SPUPR]
4	Top of Stack Pointer [SP]
5	Active Mark Stack Control Word Pointer [MP]
6	Base Addressing Mark Stack Control Word Pointer [BP]
7	Program Counter [IPC]
8	Pointer to currently executing code segment [SEGB]

Registers are initialized in two ways. The first happens during the boot sequence. Refer to section 3.7.2, the Microengine Bootstrap Sequence for details. The second method is by the PMACHINE statement, a III.0 UCSD Pascal language extension which allows generation of Pascal operators. For example, the program segment below reads the value of the mark stack pointer into a Pascal variable.

```
CONST MP=5
      LPR=157; STO=196;

VAR   LMP: INTEGER;

BEGIN
  PMACHINE(~LMP,(MP),LPR,STO);
END.
```

A complete description of all the III.0 UCSD Pascal operators is found in section B.5 of this manual. The '~' in the PMACHINE statement places the address of the following identifier on top of the stack. An identifier (or expression) enclosed in parens is evaluated and the result is placed on top of the stack. An expression without '~' or parens is placed directly into the code.

It should be noted that the positive register numbers refer to values in the active TIB (Task Information Block). The Pascal declaration for the TIB is:

```
TIB = RECORD { TASK INFORMATION BLOCK }
            REGS: PACKED RECORD
                    WAITQ: TIBP; { QUEUE LINK FOR SEMAPHORES }
                    PRIOR: BYTE; { TASK'S CPU PRIORITY }
                    FLAGS: BYTE; { STATE FLAGS...NOT DEFINED YET }
                    SPLOW: ^INTEGER; { LOWER STACK POINTER LIMIT }
                    SPUPR: ^INTEGER; { UPPER LIMIT ON STACK }
                    SP: ^INTEGER; { ACTUAL TOP-OF-STACK POINTER }
                    MP: MSCWP; { ACTIVE PROCEDURE MSCW PTR }
                    BP: MSCWP; { BASE ADDRESSING ENVIRONMENT PTR }
                    IPC: INTEGER; { BYTE PTR IN CURRENT CODE SEG }
                    SEGB: ^CODESEG; { PTR TO SEG CURRENTLY RUNNING }
                    HANGP: SEMP; { WHICH TASK IS WAITING ON }
                    XXX: INTEGER; { NOT USED }
                    SIBS: ^SIBVEC { ARRAY OF SIBS FOR 128..255 }
                  END { REGS } ;
              MAINTASK: BOOLEAN;
              STARTMSCW: MSCWP
            END { TIB } ;
```

For example, the MP, the mark stack control word pointer, is register number 5 and it is word 5 in the TIB. When the microcoded operators LPR and SPR refer to positive valued registers, these values are taken from the TIB.

The Segment Vector Pointer register points at the segment vector which is an operating system table that contains information concerning all active segments in the Pascal System. The Pascal declaration for the segment vector is:

```
SEGVEC = ARRAY[0..15] OF SIBP
```

SIBP, meaning Segment Information Block pointer is a pointer to a record containing information about each active segment. The Pascal declarations for SIB and SIBP are:

```
SIBP = ^SIB;

SIB = RECORD { SEGMENT INFO BLOCK }
        SEGBASE: ^CODESEG;  { MEMORY ADDRESS OF SEG }
        SEGLENG: INTEGER;   { # WORDS IN SEGMENT }
        SEGREFS: INTEGER;   { NUMBER OF ACTIVE CALLS }
        SEGADDR: INTEGER;   { ABSOLUTE DISK ADDRESS }
        SEGUNIT: UNITNUM    { PHYSICAL DISK UNIT }
      END { SIB } ;
```

3.7.4 Concurrency Primitives and Interrupts

UCSD Pascal provides several language constructs that are useful for operating system and I/O handler development. Those pertinent to inter-task communication and I/O coordination are described below.

Tasks:

Tasks provide the basis for concurrent processing. A task is created by the PROCESS declaration, a UCSD Pascal extension. The PROCESS declaration is syntactially similar to a PROCEDURE declaration, except that it creates a task that may run concurrently with other tasks in the system. Each task in the Microengine has an associated Task Information Block (TIB) that reflects the status of the task. The TIB contains such information as the stack limits of the task, the top of stack pointer, the task's priority, the task's program counter, and a queue link for the task scheduling mechanism to be described below.

Task switching is done by the semaphore mechanism as proposed by Dijkstra. This is implemented by means of LIFO queues. Queues are associated with the currently executing task, all tasks that are ready to execute, and tasks that are waiting on a semaphore.

Semaphores:

A TIB that is to execute is placed on the current task queue. The queue has only a single task, the currently executing one, on it. TIB's that are ready to execute, but are waiting for processor allocation, are on the ready queue in priority order. When a task is to execute, it is moved from the ready queue to the current task queue. Otherwise, TIB's are chained on queues of sema- phores until the semaphore they are associated with is signalled. A semaphore variable is declared as is any other built-in data type. In order to give a semaphore an initial value, the SEMINIT statement is used. It takes as parameters a semaphore and an integer value. The integer value specifies the number of times the semaphore has been signalled. For example

```
SEMINIT(SEM,0);
```

will initialize semaphore SEM to not signalled.

Tasking primitives:

The task queues in the UCSD III.0 system are manipulated by four primitives: SIGNAL, WAIT, START, STOP. Both SIGNAL and WAIT take a single parameter, which is the semaphore variable. SIGNAL and WAIT synchronize inter-task communication. WAIT will cause a task on the current task queue to wait on the WAIT semaphore parameter, if the semaphore has never been signalled. For example: WAIT(SEM) where SEM is of type SEMAPHORE will perform a WAIT operation.

SIGNAL will cause a task waiting on the SIGNAL semaphore parameter to start executing, if the task is at a higher priority. Otherwise, the highest priority task will begin execution. SIGNAL(SEM) will perform the signal operation on a semaphore. In order to initialize or terminate tasks, the statements START and STOP are utilized. START links a TIB created by the PROCESS statement into the ready queue. It takes as parameters a PROCESS, a variable of type PROCESSID which is a pointer to a TIB, the amount of stack space the PROCESS will require and the PROCESS priority. Thus in order to activate a process, the system call would be:

```
START(PROCESS(PARAMS), PID, STACK, PRIOR);
```

STOP, when executed by a running task, causes the task to be terminated. STOP is called when a process is terminated and may not be called as a Pascal statement. Note that both SEMAPHORE and PROCESSID are predeclared types recognized by the Pascal compiler.

Interrupt Attachment:

The tasking primitives have been described. Now the final command, ATTACH, must be introduced. Its purpose is to attach I/O device signals from the I/O hardware to the software and firmware supported semaphores. Each I/O port on the Microengine has a unique interrupt address where a signal is sent indicating that an I/O operation is complete. For example, a disk transfer could have completed, or a character has been read in from a terminal and placed in the serial port data register, or a character has been sent from the printer buffer in the parallel port controller to the printer. ATTACH has two parameters, a semaphore, and an interrupt address where the semaphore is attached, thereby allowing a hardware signal to signal a semaphore. For example:

```
ATTACH(SEM,32);
```

will attach a semaphore, SEM, to the interrupt vector address to the floppy-DMA interrupt vector.