MICROPROGRAMMED IMPLEMENTATION OF A SCHEDULER

R. Chattergy
University of Hawaii
Honolulu, Hawaii

Application of microprogramming to enhance the performance of operating systems has been discussed in the literature in the past [7,1,5]. Two examples of such applications can be found in [4,6]. This paper discusses the philosophy behind the microprogrammed implementation of a scheduler, used in a large, time-shared computer incorporating several processors.

1. INTRODUCTION

This paper describes the activities of a typical microprogrammed scheduler (microscheduler) in a time-shared system with multiple processors. This description is a simplified version of the actual microscheduler in the BCC 500 computer system, designed by W. Lichtenberger, M. Pirtle, B. Lampson, J. Freeman, R. Schultz and R. Van Tuyl in 1969. A functional diagram of the system is shown in figure 1. The general philosophy of scheduling for a multiprocessing system has been discussed at length in [3]. As mentioned in [3], scheduling consists of two activities. The first is the determination of an optimal schedule based on some scheduling criterion. The second is the enforcement of that schedule on the processes in the system.

Clearly the task of selecting a scheduling criterion and determining a schedule by some algorithm is a complex and evolutionary process. The environment within which resources are allocated by scheduling often changes, forcing a change of the scheduling criterion, and in extreme cases a change of the corresponding algorithm. Hence a scheduling algorithm is unsuitable for microprogrammed implementation in a read-only memory. On the other hand, the task of enforcing a schedule, classified as a midi-primitive in [7], consists of more permanent subtasks such as, the creation and maintenance of queues, blocking and awakening of active processes, making calls on the swapper etc. These subtasks are discussed in detail in the later sections. These subtasks are even system independent in the sense that, they must be carried out in one form or another regardless of the system architecture.

The above considerations prompted the design of the scheduler in the following form. The scheduler is a hardware processor, microprogrammed to execute a set of instructions <u>in a loop</u>. The task of schedule enforcement is directly microprogrammed as part of this loop. Besides schedule enforcement, the processor also executes a machine instruction of an emulated machine in every iteration of the loop. The scheduling algorithm is implemented in software on this emulated machine. Thus the microscheduler carries out both scheduling activities, executing the scheduling algorithm written in a high-level language for flexibility, and enforcing the schedule discipline via firmware for speed of execution.

2. HARDWARE DESCRIPTION

The BCC 500 shown in figure 1, is a large time-shared computer with two processors for executing user-processes, and three special purpose processors for carrying out system management tasks (i.e., executing the operating system). All of the processors operate independently, communicate with each other via main memory, and are microprogrmamed. Figure 2 shows the arithmetic-logic unit of a microprogrammed special purpose processor and its bus structure.

All registers shown in figure 2 are twenty four bits wide. M,Q, and Z are the main registers, where M serves as the communication register with the main memory via the main memory interface. The outputs of M and Q are connected to the left Boolbox (LB), and the outputs of Q and Z go into the right Boolbox. Each Boolbox can perform any of sixteen boolean operations on its inputs. The outputs of the Boolboxes are connected to the Adder. The output of the left Boolbox goes into the Cycler.

The outputs of the Adder and the Cycler can be put into any of the seven Holding Registers, R0,...,R6. The register R0 acts as the memory address register (MAR) when main memory is accessed. The output of any of the Holding Registers can be incremented by one and hence any of these registers can be used as a counter. In addition there are sixty four Scratch Pad registers which are loaded from the X-bus and read onto the Y-bus.

The Control Memory of the microprocessor is a read-only, diode memory containing atmost 2 048, ninty-bit words. Different fields of the 90 bit micro-word control different logic circuits and in case of a branch to a subroutine, the return address is automatically stored in an auxilliary register.

3. MICROWORD

The bits and fields in the 90 bit words in the control memory are coded to generate the controlling signals necessary to operate the ALU. For example, bits 0-5 are used to set up one of a number of branch conditions to be tested for branching. The bits 8-17 are used to provide the branch address, which can also be obtained from the OS register (return from a subroutine) or the X-bus (computed go to). The bits 18-41 are used to specify a 24-bit constant which can be gated onto the X or the Y buses respectively. A detailed description of all the fields is too long. The above description should be enough to give the reader a "feel" for the system.

4. MICRO LANGUAGE

A special purpose readable reference language, called MICRO, is available for writing microprograms for the processors of the BCC 500. The MICRO language has declaration statements and statements for execution. The declaration statements can be used to define macros, give symbolic names to registers, define parameter values, define branch conditions for repeated use in the program, etc. The set of statements for execution consists of the usual assignment (including multiple) statement, memory operations statement, branch instructions, microword-field assignment statements, etc. It is impossible to discuss the language in detail here. Instead, explanatory comments enclosed between "/*" and "*/" are imbedded in the sample microprograms provided in the later sections.

5. MICROSCHEDULER INTERFACE

A simplified diagram of the interfaces among the system resources and the microscheduler is given in figure 3. In this figure, the microscheduler and the user processors are hardware processors, whereas the swapper and the scheduler are software packages run on the system processors. All processors in the system make WAKEUP calls to the microscheduler to activate processes. If a process, which has received a wakeup call, is not in the main memory, the microscheduler inserts the identity of this process into the input stack of the scheduler. The scheduler, using its scheduling algorithm, assigns a priority to the process which cannot be changed by the other processors. It puts the process in its appropriate position in a queue and makes a SWAPIN call to the swapper. In some cases such as a page-fault condition, the microscheduler can make a direct SWAPIN call to the swapper. Due to lack of adequate memory space, the swapper may fail to swap in a process. It then makes a GIVEUP call to the microscheduler, asking for the identity of a process that may be swapped out to make room. The microscheduler answers this call via a SWAPOUT call indicating the process that can be swapped out.

The microscheduler alters the work schedules of the processors by making SWITCH calls. A switch call contains the identity of the process to be worked on by the receiving processor. If a new process of preemptive priority preempts the current process on a processor, this information is sent back to the microscheduler via a RETURN call by the processor. If a running process blocks itself, the corresponding processor makes a BLOCK call to the microscheduler. The processor informs the microscheduler whether or not the blocked process should be swapped out (a policy decision made by the monitor).

All communications with the microscheduler are carried out via a stack in the main memory under suitable PROTECT and UNPROTECT mechanisms.

6. LIFE-CYCLE OF AN ACTIVE PROCESS

Figure 4 shows the life-cycle of an active process under control of the microscheduler. Consider an active process which receives a call from some other running process. The call is entered under protection into the top two words of the input stack of the microscheduler. The microscheduler periodically inspects the stack for calls from the outside. Upon finding such a call, the microscheduler checks the identity of the process for validity. If the identity is invalid, it ignores the call and deletes the entry. For a valid call, the microscheduler determines whether the call is for a wakeup or block.

WAKEUP CALL: The microscheduler merges the data word from the call into the program interrupt word of the process (PIW), stored in the process resident table. It checks to see if the process is either waiting in the microscheduler queue for a processor, or already running. In either case, nothing more needs to be done. For an interesting example of this situation see [2] pp. 271.

On the other hand, if the process is blocked, the microscheduler unblocks the process. It checks to see if the process is in the main memory. If the process is in the main memory, the microscheduler inserts it, according to it's priority, in a queue of processes waiting for processors. Note that if the inserted process has preemptive priority then it can preempt a running process. This means that the microscheduler may have to reallocate the processors. A preemptive priority structure is necessary because the system does not have a hardwired interrupt mechanism. Preemptive priorities must be assigned to processes whose non-execution can lead to loss of information.

If the process is not in the main memory, it has to be swapped in. The microscheduler then puts the process in a stack of processes waiting for the scheduler. The scheduler determines the priorities of the processes independently, and inserts them in the input queue of the swapper. In some cases, the microscheduler may make a direct request for a swapin to the swapper.

BLOCK CALL: Whenever a running process blocks, the monitor is activated. The monitor decides whether the blocked process should remain in main memory (page-fault) or be swapped out (input from terminal). This decision is passed onto the microscheduler via the block call. The microscheduler blocks the process and if so directed makes a swapout call to the swapper.

If a process is caught in a timer-trap, the microscheduler does not block it but puts it on the input stack of the scheduler for future scheduling. The scheduler changes the priority of such a process based on its scheduling criterion and sends a wakeup.

RETURN CALL: Whenever a running process is preempted of it's processor by a process with preemptive priority, the processor sends a return call to the microscheduler. The microscheduler removes the process from the run state and puts it in the microscheduler queue to wait for a processor.

7. PROCESSOR SCHEDULING

The microscheduler periodically checks the status of each processor and reallocates those processors which are either idle or can be preempted. The processors are directed to switch processes by means of the SWITCH call sent by the microscheduler. In principle, the SWITCH call provides the processor the identity of the new process to be run.

A processor has three possible states. It is either idle, or running a process, or running a process which has preempted another process. If the processor is in the last mentioned state, then the microscheduler does not send it a switch call untill the process running on it blocks. A processor is switched only if it is idle or running a process which has not preempted another process.

Whenever the microscheduler enters a new process in its queue that has a preemptive priority, it sets up a schedule flag. This flag indicates that reallocation of the processors is necessary. When the microprocessor decides to reallocate the processors, it switches the highest priority process in the microscheduler queue with a process that has blocked.

8. MICROSCHEDULER INPUT STACK

Calls to the microscheduler are placed on a stack called USIB in the microprogram. Each call consists of two words. The leftmost six bits of the first word contains an opcode identifying the call, such as 1 for wakeup, 2 for block etc. The rightmost eighteen bits of the first word contains a pointer to the first word of a process's process resident table (in effect identifies the process). The second word contains the bits to be set by the microscheduler in the process interrupt word in the resident table, as a result of the call.

9. MICROPROGRAM FOR MANAGEMENT OF USIB

```
    USIBIGIN:  PROTECT (USIB);
/* Protects stack from use by other processors...*/
                MAR ← USIBTOP, FETCH;
/* Get pointer to top of stack...................*/
                Q ← SK7 ← M, MAR ← USIBASE, FETCH;
/* Get pointer to the bottom of stack............*/
                M EOR Q, GO TO EMPTY IF LB=0, Z←LUSIBE
/* Compare top and bottom pointers stored in M   */
/* and Q by exclusive OR.  If pointers same, out-*/
/* put of left boolbox LB=0.  Branch to block    */
/* labelled EMPTY.  LUSIBE=No. of words in call..*/
                MAR ← SK7, FETCH, Z ← Q-Z;
/* Get first word from stack.....................*/
                SK7 ← Q ← M, MAR ← MAR+1, FETCH;
/* Get second word from stack....................*/
                R2 ← M;
/* Put second word in register R2................*/
                M ← Z, MAR ← USIBTOP, STORE;
/* Move pointer to top of stack down by LUSIBE   */
/* words.........................................*/
                UNPROTECT (USIB);
/* Unprotect stack.  The first word fetched from */
/* stack is in SK7 and Q.  The second word is in */
/* register R2...................................*/
                M ← Q LCY 4, Q ← 600 000 17B;
                M ← M AND Q LCY 2, CALL UERROR IF LB=0;
/* Left cycling the contents of Q through M mask-*/
/* ed by 600 000 017 and the last AND operation  */
/* leaves the opcode for the microscheduler in   */
/* the rightmost bits of M.  For a valid opcode  */
/* this must be > 0.  UERROR subroutine is called*/
/* otherwise.....................................*/
                Q ← MAXOP;
/* Maximum value of opcode is loaded in Q........*/
                CALL UERROR ON Q-M < 0. Q ← OPTAB-1;
/* Call UERROR if the opcode exceeds its maximum */
/* allowable value...............................*/
                R5 ← M+Q, Q ← R2, DGO TO USIBIGIN;
/* R5 stores the pointer to the subroutine (wake-*/
/* up, block etc.) to be used by the microschedu-*/
/* ler as a result of this call.  The subroutine */
/* is called in the next line.  Q and R2 contains*/
/* the second word of the call.  DGO TO is a de- */
/* layed branch.  The branch is executed after   */
/* execution of the next instruction is complete.*/
        MAR ← Z ← M ← SK7, CALL STKLK,.C ← 3,.TCX,.TCW;
/* STKLK causes a branch to the subroutine point-*/
/* ed at by R5.  It also saves the return address*/
/* in a stack.  Because of the delayed GO TO in  */
/* the previous line, this return address is that*/
/* of USIBIGIN.  Thus a return is made to USIBI- */
/* GIN after a subroutine such as block or wake- */
/* up has been executed.  MAR contains the ad-   */
/* dress of the PRT plus 3 (ie. the address of   */
/* the PIW), where the 3 is merged from the con- */
/* stant field of the microword by TCX and TCW...*/
      OPTAB:  GO TO WAKEUP;
              GO TO BLOCK;
              GO TO BLOCKOUT;
              GO TO GIVEUP;
/* End of microprogram for the management of USIB*/
```

10. FLOW-CHART OF THE MICROSCHEDULER

A complete description of all the microprograms is too long to be included in this paper. A flow-chart describing the operation of a simple micro-scheduler is given below.

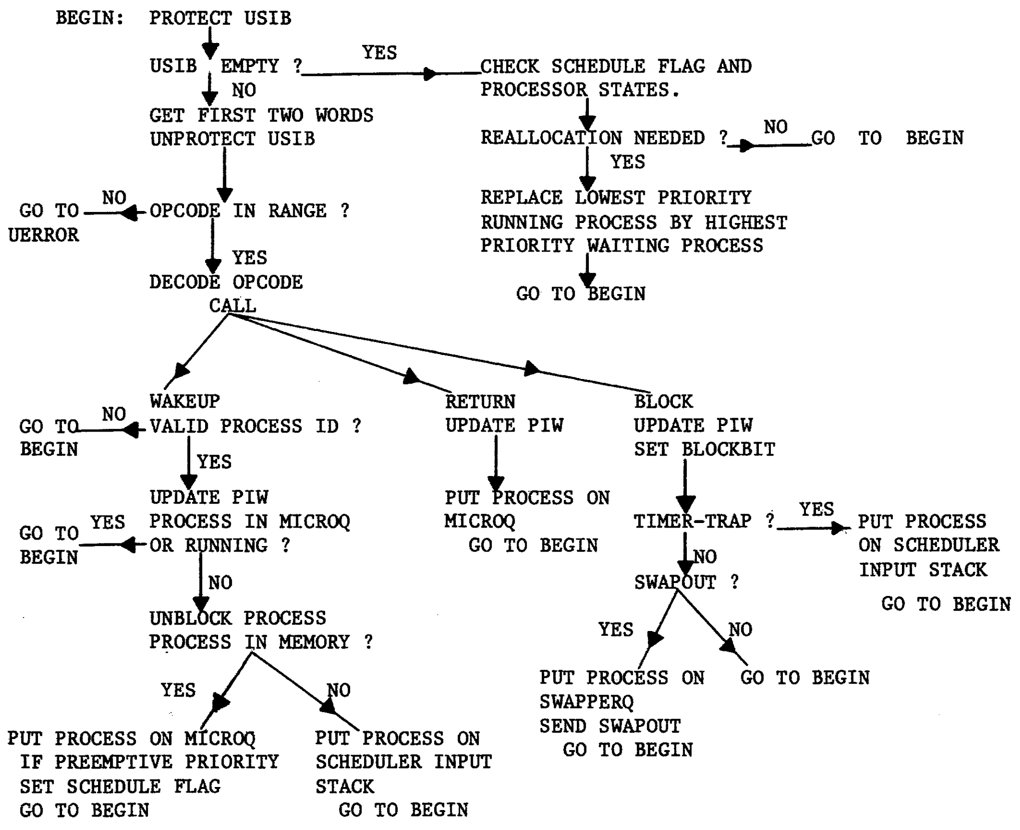
In the flow-chart, the usual housekeeping operations have been left out. Also the flow-chart does not include such operations as the management of real-time queues, which a microscheduler of a time-shared system must handle. A real-time queue is a queue of processes whose wakeup signals are specified by a real-time clock, and does not come from other processes. Basically, the microscheduler inspects its input stack periodically, and in response to calls left there by other processors it executes proper subroutines such as WAKEUP or BLOCK. It also checks the schedule flag and the states of the processors. Whenever necessary, it reallocates the processors and continues to loop around.

ACKNOWLEDGMENT

The author gratefully acknowledges encouragement and constructive criticism from Professor Wayne Lichtenberger of the department of Electrical Engineering, University of Hawaii.

REFERENCES

[1] W. H. Burkhardt, R. C. Randel, Design of operating systems with micro-programmed implementation, NTIS Report, PB-224-484, September, 1973.
[2] R. M. Graham, Principles of system programming, John Wiley, 1975.
[3] B. W. Lampson, A scheduling philosophy for multi-processing systems, Communications of the ACM, vol. 11, No. 5, May, 1968.
[4] B. H. Liskov, The design of the Venus operating system, Communications of the ACM vol. 15, No. 3, March, 1972.
[5] J. V. Sell, Microprogramming in an integrated hardware/software system, Computer Design, vol. 14, No. 1, January, 1975.
[6] W. G. Sitton, L. L. Wear, A virtual memory system for the Hewlett-Packard 2100A, preprints of the seventh annual Workshop on Microprogramming, ACM, September, 1974.
[7] A. H. Werkheiser, Microprogrammed operating systems, preprints of the third annual Workshop on Microprogramming, ACM, October, 1970.


BEGIN: PROTECT USIB
USIB EMPTY ?
YES
CHECK SCHEDULE FLAG AND PROCESSOR STATES.
NO
GET FIRST TWO WORDS UNPROTECT USIB
REALLOCATION NEEDED ?
NO
GO TO BEGIN
YES
GO TO UERROR
NO
OPCODE IN RANGE ?
REPLACE LOWEST PRIORITY RUNNING PROCESS BY HIGHEST PRIORITY WAITING PROCESS
YES
DECODE OPCODE CALL
GO TO BEGIN
WAKEUP
GO TO BEGIN
NO
VALID PROCESS ID ?
YES
RETURN UPDATE PIW
BLOCK UPDATE PIW SET BLOCKBIT
UPDATE PIW PROCESS IN MICROQ OR RUNNING ?
GO TO BEGIN
YES
PUT PROCESS ON MICROQ GO TO BEGIN
TIMER-TRAP ?
YES
PUT PROCESS ON SCHEDULER INPUT STACK GO TO BEGIN
NO
NO
SWAPOUT ?
UNBLOCK PROCESS PROCESS IN MEMORY ?
YES
NO
YES
NO
PUT PROCESS ON SWAPPERQ SEND SWAPOUT GO TO BEGIN
GO TO BEGIN
PUT PROCESS ON MICROQ IF PREEMPTIVE PRIORITY SET SCHEDULE FLAG GO TO BEGIN
PUT PROCESS ON SCHEDULER INPUT STACK GO TO BEGIN

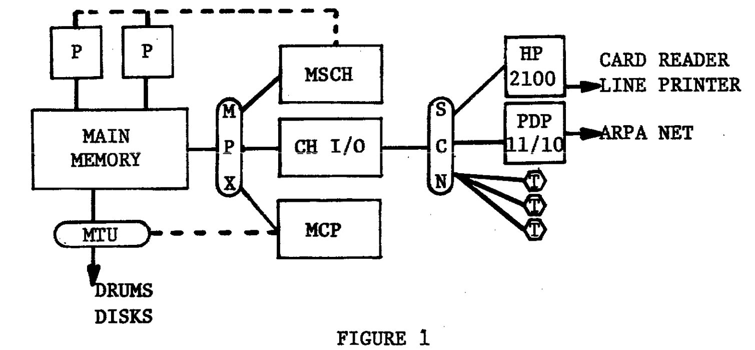


FIGURE 1

P	User Processor
MSCH	Microscheduler
CH I/O	Character I/O Processor
MCP	Memory Control Processor
MTU	Memory Transfer Unit
MPX	Multiplexer
SCN	Scanner
T	Terminal

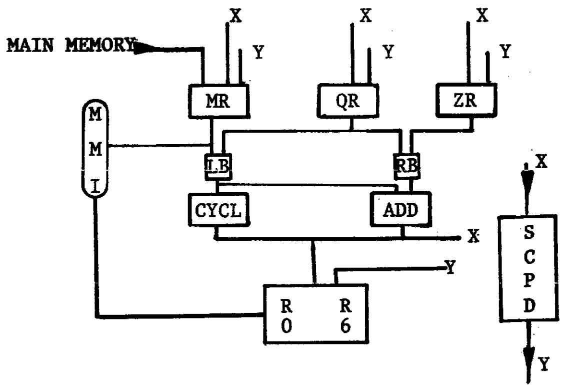


FIGURE 2

MR	M Register
QR	Q Register
ZR	Z Register
MMI	Main Memory Interface
SCPD	Scratch Pad

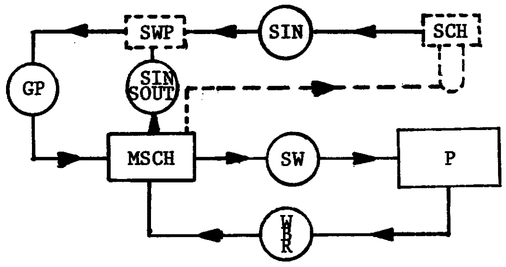


FIGURE 3

SWP	Swapper
SCH	Scheduler
SIN	Swap In
SOUT	Swap Out
GP	Giveup
SW	Switch
W	Wakeup
B	Block
R	Return

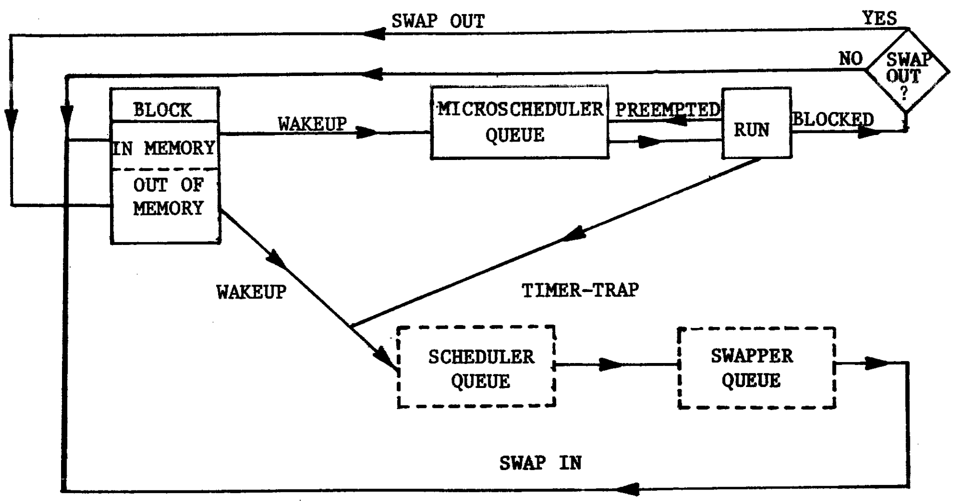


FIGURE 4