A distributed function computer with dedicated processors

R. Chattergy* and U. W. Poocht

Recent advances in hardware technology have led to marked decreases in the costs of processors in computer systems. Reduced costs of processors have led to the design and implementation of distributed function computer architectures, using multiple dedicated processors. This paper outlines the hardware, modular operating system and protection features of such a distributed function computer, designed for time shared operation.

(Received October 1977)

1. Introduction

The concept of distributed function architecture appears to embrace a large variety of computer designs. Therefore, it is essential to clearly specify the exact design under consideration. We make use of the classification scheme given by Joseph (1974) for this purpose. According to this classification, the design under discussion should be called a distributed processing element system. It consists of functionally distributed, heterogeneous, multiple processors, which are capable of simultaneously executing independent processes. Several other computers in this category are listed by Joseph (1974) and by Jensen (1975).

The architecture of this time shared computer system was largely determined by a design team working at the University of California, Berkeley, under Project Genie, funded by the Advanced Research Projects Agency, US Department of Defense (Lichtenberger and Pirtle, 1965; Lampson, Lichtenberger and Pirtle, 1966). The primary objective for the design of this time shared computer system was to provide interactive computational facilities for a large number of users (much larger than were commonly serviced by time shared systems around 1968/1969) with modest computational needs, large data files and fast terminal response requirements. The software on the system creates a transaction oriented computational environment tailored to the needs of individual groups of users. This is one of the causes that leads to the design of distributed function systems (Joseph, 1974 p. 24). The requirement of providing fast terminal responses to a large number of users is met by the use of multiple processors simultaneously executing independent user jobs. However, the use of multiple processors alone does not create a distributed function computer. The definition of a multiprocessor system given by Enslow (1974 pp. 19-21) and the ANSI standards, require that a multiprocessor system must operate under an integrated operating system. In the computer under discussion, the operating system is distributed among several special purpose processors, which during execution of tasks communicate with each other via random access memory. In other words, there is neither a master/slave relationship among the processors, nor a separate executive for each processor, nor a 'floating' executive moving from processor to processor. This is why the computer discussed in this paper can be classified as a distributed function system.

2. System architecture

The architecture of the system, shown in **Fig. 1,** consists of two processors for executing user processes, and three special purpose dedicated processors for carrying out the system management task (i.e. executing the operating system). All of the processors operate independently and communicate with each other via random access memory. The RAM unit has four ports: two of the ports are dedicated to the two user processors, another is dedicated to the memory transfer unit and the last one is shared by the three operating system processors.

All five processors are constructed by microprogramming a basic hardware processor. However, since the nature of computation used in system programs is considerably different from that in the user programs, the microprogrammed system processors are not identical to the microprogrammed user processors. For example, the system processors do not have facilities for multiple precision floating point arithmetic operations or hardware memory mapping mechanisms which are used in the user processors. Most of the data and programs used by the dedicated system processors are stored in dedicated read-only memories (ROMs) shown in Fig. 1. The system processors access the RAM only to share some system tables, and for this reason they can share a single port without unduly high memory contention problems.

A major problem in large multiprocessor systems has to do with the size of the RAM needed to store all active user processes waiting for processors. With the cost of semiconductor RAMs decreasing at a rapid pace, this may not be a serious problem in the future. In the system under discussion, the size of the shared RAM is reduced by making use of swapping. Most of the active user processes reside on drums; the RAM

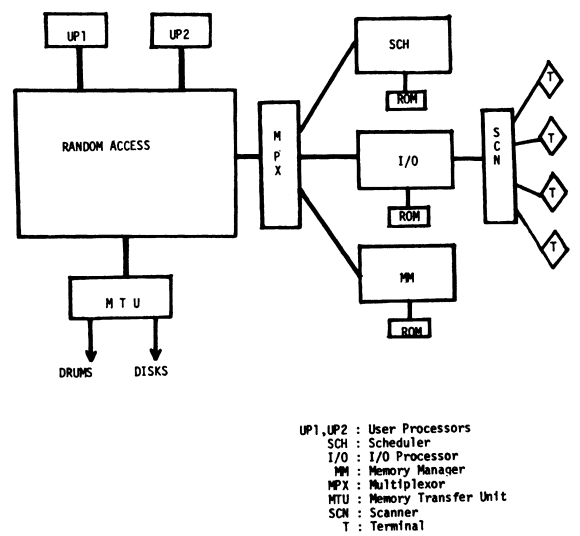

Fig. 1 System architecture

*Department of Electrical Engineering, University of Hawaii
†Computing Science Division, Texas A & M University, Industrial Engineering Department, College Station, Texas 77843, USA

contains enough active processes to keep the user processors busy. When a user process stops execution for any reason, the memory manager decides whether it should be swapped with an active process on the drum. The memory transfer unit is kept continually busy by the memory manager, carrying out this swapping operation. Swapping, with as little delay as possible, being a critical part of this system, a memory port is dedicated to the memory transfer unit. Because of two user processors and the continuous swapping activity, memory contention is a serious problem in this system. The contention problem is resolved by using a dynamic priority assignment scheme which has been discussed in detail in Pirtle (1967).

3. Computational environment

The computational environment created by the software, for each user group, is process oriented. The process oriented environment is similar to that described by Farber (1973; 1974) and all computational activities supported by the software are carried out by creating and executing processes. For our purpose it is sufficient to define a process as a program along with the resources necessary for its execution. A detailed discussion of operating systems in terms of such interactive processes can be found in Graham (1975). The processes in the operating system communicate with each other by means of messages left in mailboxes in the shared random access memory. A process is normally *blocked* and is *awakened* by the arrival of messages from other processes or external sources. The messages direct the awakened process to carry out certain activities, such as the transfer of a user process from auxiliary store into random access memory.

The process, after acquiring a processor, generates its own messages, and upon termination blocks itself. The scheduler assigns the processor to the process. This approach allows a flexible and distributable operating system that can be implemented on multiple dedicated processors. The life cycle of a typical process is illustrated in **Fig. 2.** 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 microprogrammed scheduler (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 a block.

For a 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.

If on the other hand the process is blocked, the microscheduler unblocks the process. It checks to see if the process is in the random access memory. If the process is in RAM, the microscheduler places the process according to its priority into a queue of processes waiting for processors. This placement is in a pre-emptive priority fashion, thus the microscheduler may have to reallocate the processors.

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

Whenever a running process blocks, the monitor is activated. The monitor decides whether the blocked process should remain in main memory or be swapped out. This decision

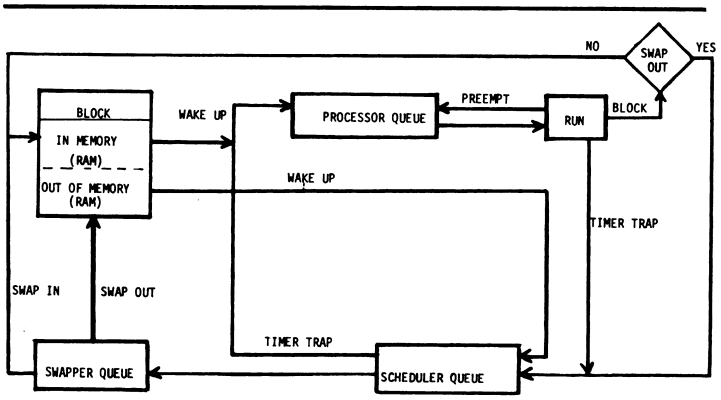

Fig. 2 Life cycle of a process

is passed onto the microscheduler via the block call. The microscheduler blocks the process and if so directed makes a swap-out call to the swapper.

If a process is caught in a timer trap, the microscheduler places 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 call.

Whenever a running process is pre-empted of its processor by a process with pre-emptive priority, the processor sends a return call to the microscheduler. The microscheduler removes the process from the running state, and places it in the microscheduler queue to wait for a processor.

4. Operating system

The nucleus of the operating system consists of certain primitive operations (tasks) distributed among the three dedicated special purpose processors, the Scheduler, the Memory Manager, and the Input/Output Processor. For example, the scheduler executes microprogrammed tasks for allocating processors to processes and other related activities. The memory manager similarly executes tasks for managing the hierarchical memory consisting of the RAM, the drums and the discs. The input/output processor manages all communications with the terminals. The primitive operations of the operating system are augmented and made more convenient for use by means of utility programs. For example, the input/output processor creates and maintains logical information channels, using its primitive operations. Each terminal is connected to such a logical input and output channel. A utility program is used to read from or write to a channel, a string of characters. The input/output processor subsequently transfers these characters through the channels to their proper destinations. Utility programs also create terminal interfaces with command languages and facilities for buffering character strings. For a more detailed discussion of the input/output processor design see Heckel and Lampson (1977).

The major responsibility of an operating system is to allocate system resources, in connection with which it makes and enforces decisions. The separation of the decision making activity from the activities of enforcing these decisions on the processes, is fundamental to the design of tasks for an operating system (Balzar, 1973). Once the system architecture and the process states are clearly specified, the activities required to enforce the operating system's decisions can be clearly defined and used to identify the needed tasks. For example the scheduler, using some criterion and algorithm, may set up an optimal schedule for processes waiting for processors. No matter what this schedule is, it essentially assigns a priority (pre-emptive or not) to each waiting process. The enforcement of this schedule is then achieved by allocating processors to processes according to their priorities. This activity is dependent on the hardware features of the system and can be decom-

```
PROGRAM DED_SCHEDULER;
      BEGIN
        INTEGER SCHEDULER_FLAG,REQUEST_LATCH;
        BOOLEAN INPUT_MAILBOX_LOCK,TIMER_TRAP,USER_PROCESSOR_IDLE,PROCESS_TEST;
        BOOLEAN PROCESS_RAM,PROCESS_STATUS_WORD,SWAP_OUT;

            PROCEDURE PROCESS_MESSAGE;
                BEGIN
                    INPUT_MAILBOX_LOCK=TRUE;
                    FETCH(MESSAGE);
                    INPUT_MAILBOX_LOCK=FALSE;
                    OPCODE=DECODE(MESSAGE);
                    WAKEUP;
                    BLOCK;
                END PROCESS_MESSAGE;

            PROCEDURE BLOCK;
                BEGIN
                    UPDATE(PROCESS_STATUS_WORD);
                    IF TIMER_TRAP THEN QUEUE(SWAP_OUT,PROCESS);
                    IF SWAP_OUT THEN QUEUE(SWAP_OUT,PROCESS);
                    SCHEDULER_FLAG=1;
                END BLOCK;

            PROCEDURE WAKEUP;
                BEGIN
                    UPDATE(PROCESS_STATUS_WORD);
                    IF PROCESS_TEST THEN GO TO RETURN;
                    IF PROCESS_RAM THEN BEGIN
                                        QUEUE(PROCESSOR,PROCESS);
                                        IF PRIORITY='PREEMPTIVE' THEN SCHEDULER_FLAG=1;
                                        END
                                   ELSE QUEUE(SCHEDULER,PROCESS);
                    RETURN:
                END WAKEUP;

            PROCEDURE ALLOCATE;
                BEGIN
                    COMMENT - ALLOCATE PROCESSOR TO HIGHEST PRIORITY PROCESS
                              WAITING;
                    .....
                    .....
                END ALLOCATE;

            PROCEDURE UPDATE(ITEM);
                TYPE DECLARATION OF ITEM;
                BEGIN
                    COMMENT- SET PROCESS_TEST IF PROCESS IN PROCESSOR
                             QUEUE OR RUNNING;
                    COMMENT- SET PROCESS_RAM IF PROCESS IN RAM;

                    .....
                    .....
                END UPDATE;

        PROCEDURE PROCESSOR_ALLOCATE;
            BEGIN
                IF USER_PROCESSOR_IDLE THEN ALLOCATE ELSE PREEMPT;
                SCHEDULER_FLAG=0;
            END PROCESSOR_ALLOCATE;

        PROCEDURE PREEMPT;
            BEGIN
                COMMENT - IF WAITING PROCESS HAS PREEMPTIVE PRIORITY,
                          THEN PREEMPT THE LOWEST PRIORITY RUNNING PROCESS;
                .....
                .....
            END PREEMPT;

        PROCEDURE QUEUE(Q_NAME,Q_ITEM);
            TYPE DECLARATION OF Q_NAME,Q_ITEM;
            BEGIN
                COMMENT- ENTER Q_ITEM INTO QUEUE DESIGNATED BY Q_NAME;
                ......
                ......
            END QUEUE;


    COMMENT - MAIN PROGRAM LOOP;
       FOR I=1 WHILE TRUE DO
                        BEGIN
                            IF REQUEST_LATCH=1 THEN PROCESS_MESSAGE
                                               ELSE
                            IF SCHEDULER_FLAG=1 THEN PROCESSOR_ALLOCATE;
                        END;
    .....
    .....
    .....

    END MAIN PROGRAM.
```

Fig. 3 Dedicated scheduler

posed into tasks which will remain constant throughout the life of the system. Some of these tasks of the scheduler are discussed in the next paragraph.

The microscheduler, in response to messages left in its 'mailbox', *wakes up* processes (see **Fig. 3**). The microscheduler is a microprogrammed processor whereas the swapper and the scheduler are software modules run on the system processors. All processors in the system make WAKE UP calls to the microscheduler to activate processes. If a process, which has received a WAKE UP call, is not in 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 places the process into its appropriate position in the queue managed by the scheduler, and makes a SWAPIN call to the swapper. Because of inadequate memory space, the swapper may fail to swap-in a process. It then makes a GIVUP call to the microscheduler, requesting the identity of a process which may be swapped out to provide memory. The microscheduler responds by generating a SWAPOUT call indicating the process that can be swapped out.

The microscheduler periodically checks the status of each processor and reallocates those processors which are either idle, or can be pre-empted. The processors are directed to switch processes by way of the SWITCH call sent by the microscheduler. The SWITCH call provides the processor with the identity of the new process to be run.

A processor has three possible states. It is either idle, running a process or running a process which has pre-empted another process. If the processor is in the last mentioned state, then the microscheduler does not send it a switch call until the process running on it blocks. A processor is switched only if it is idle or running a process which has not pre-empted another process. Whenever the microscheduler enters a process in its queue that has a pre-emptive priority, it sets up a schedule flag. This flag indicates that reallocation of the processors is necessary. When the microscheduler reallocates the processors, it switches the highest priority process in the microscheduler queue with a process that has blocked.

In summary, when a process *blocks*, the scheduler removes it from the chain of processes in the run state and releases its processor. It *allocates* processors to waiting processes, makes *swap-in* calls for awakened processes not in main memory and appropriate *swap-out* calls for processes to be put on the drums.

The scheduler shares common data bases and communicates with other processors in the system via the RAM. To simplify the communications protocol among the system processes, the tasks are designed to be *noninterruptible*. After completing a task, a system processor normally checks its mailbox for urgent messages that may need attention. In order to be able to respond quickly to urgent messages, each task is kept short and fast. The high speed of execution is attained by microprogramming each task and by using very high speed (15 MIPS) system processors. Response time to messages become particularly critical for the memory manager which may receive messages from the memory transfer unit in a fraction of a millisecond. In such cases, long tasks are decomposed into chains of short subtasks. Each subtask, after completion, leaves a message in memory private to the processor, identifying the next subtask to be performed if urgent messages are not waiting for attention. The microprogrammed tasks of each resource manager are stored in local ROMs to reduce excessive memory contention problems at the RAM. A simplified flow chart of the scheduler with its associated tasks is shown in Fig. 3. Each management processor, while using a shared data base, sets a bit in a lock-register in order to protect the shared data base from simultaneous use (and/or access) by other processors. Each bit in the lock corresponds to a group of shared data bases. A processor can set its lock bit to one only if no other processor has the same bit set to one. Whenever a processor places a message in the mailbox of another processor, it sets to one a request latch in the receiving processor. *This action does not interrupt the receiving processor* but merely informs the receiving processor, when it has completed its current task, that its mailbox has a message in it.

5. Discussion

Each dedicated processor for resource management can be thought of as a hardware module designed for a specific job. This clearly encourages a modular design of the resource management software with strictly enforced (hardware)

module boundaries. Such modular design of course simplifies the problems of system testing, maintenance, and performance measurements. To test the operation of a resource management processor, its private ROM is loaded with the microprograms, the shared data structures are created in the RAM and messages are sent to the system processor. The results of the operation of the resource manager can be seen by observing the changes in the data in the RAM. Errors in operation can often be tracked and isolated to individual processors and their associated software. One management processor can thus be used to monitor the performance of another processor or perform system wide tests.

This modular design also simplifies the problems of providing a secure computational environment for the users. A user cannot gain complete control of the system by entering the privileged mode via some software error. Most of the resource management programs are microcoded and stored in inaccessible ROMs, private to each processor. The only way a user triggers the attention of the operating system is by sending messages to the monitor. If the messages are checked carefully, the chances of a user gaining complete control of a distributed operating system are rather small.

A distributed function computer system is subject to the same problems of reliability as those of a centralised computer system. In case of hardware failures, the system has to be reconfigured using standby hardware modules. To mitigate the effects of software failure, the processes should be designed using recovery blocks (Randell, 1976). These ideas for reliability improvement are fairly new and were not around when this system was designed. In this system, if a user processor fails the system does not fail completely but if a system processor fails it leads to total system failure. Because of the architecture of the system however, it is relatively simple to add a software based system manager who takes over only when any of the dedicated system processors fail.

6. Summary

In the past, a major constraint in designing novel computer systems has been the cost of the hardware. With recent advances in technology hardware costs have decreased, making it feasible to design multiprocessor systems. Such multiprocessor systems operate under integrated operating systems. We have attempted to show that important benefits can be achieved by using distributed operating systems.

The design described here is easy to understand and implement. Each hardware module carries out a specific and simple set of noninterruptible tasks, and communicates with other modules via a simple message protocol. Note that some of the simplicity of the design, such as the absence of nested interrupts, is possible because of yet another hardware feature, namely the very high speed of the dedicated system processors. The speed of these processors also enables the programmer to design more reliable systems by using redundant codes and checks (of states and messages). To summarise, cheaper and faster microprogrammed processors make it possible to design simpler, modular, distributed operating systems suitable for use in distributed processing.

Acknowledgement

The computer design discussed in this paper is a simplified version of the BCC 500 computer, designed and implemented by Wayne Lichtenberger, Butler Lampson and others of the Berkeley Computer Corporation during 1969-1971. The computer is currently operational at the Department of Electrical Engineering, University of Hawaii, under the direction of Professor Wayne Lichtenberger.

References

BALZER, R. M. (1973). An Overview of the ISPL Computer System Design, *CACM*, Vol. 16 No. 2, pp. 117-122.

ENSLOW, P. H. Jr. Ed. (1974). *Multiprocessors and Parallel Processing*, John Wiley, New York.

FARBER, D. J., *et al*, (1973). The Distributed Computing System, *Proc. IEEE Computer Society Seventh Annual Intl. Conference*, pp. 31-34.

FARBER, D. J. (1974). Software Considerations in Distributed Architectures, *Computer*, Vol. 7, No. 3, pp. 31-35.

GRAHAM, R. M. (1975). *Principles of Systems Programming*, John Wiley, New York.

HECKEL, P. G. and LAMPSON, B. W. (1977). A Terminal-Oriented Communication System, *CACM*, Vol. 20, No. 7, pp. 486-494.

JENSEN, E. D. (1975). A Distributed Function Computer for Real-Time Control, *Proc. Second Annual Symposium on Computer Architecture*, pp. 176-182.

JOSEPH, E. C. Ed. (1974). Innovations in Heterogeneous and Homogeneous Distributed Function Architecture, *Computer*, Vol. 7, No. 3, pp. 17-24.

LAMPSON, B. W., LICHTENBERGER, W. W., and PIRTLE, M. W. (1966). A User Machine in a Time-Sharing System, *Proc. IEEE*, Vol. 54, No. 12, pp. 1766-1773.

LICHTENBERGER, W. W. and PIRTLE, M. W. (1965). A Facility for Experimentation in Man-Machine Interaction, *Proc. AFIPS Conf.*, Vol. 27, pt. I, pp. 589-598.

PIRTLE, M. W. (1967). Intercommunication of Processors and Memory, *Proc. AFIPS*, Vol. 31, pp. 621-629.

RANDELL, B. (1976). System Structure for Software Fault Tolerance, in *Current Trends in Programming Methodology*, Ed. Yeh, R. T. Chapter 7, pp. 195-219, Prentice-Hall.

Book review

Programming and Problem-solving in Algol 68, by A. J. T. Colin, 1978; 251 pages. (*Macmillan*, £4·50)

As the title suggests, this is a programming textbook rather than an ALGOL 68 language primer. Emphasising that there is more to programming than knowledge of a programming language, Professor Colin introduces the student to algorithm design and evaluation, problem specification, program design and documentation in a practical and approachable style.

Each new concept is illustrated by several worked examples, which also serve to introduce common algorithms with which the student should be familiar, e. g. sorting and searching. Each chapter ends with a set of graded execrises, many of which have worked solutions at the end of the book.

I was particularly impressed to see a clear discussion of the precision of real arithmetic: a subject which is clearly relevant to most computing, but which is often either absent from programming texts or wreathed in numerical analysis.

Only as much ALGOL 68 as one might expect to find in a first year undergraduate course is covered. Omissions include computing with references, operator definitions, transput with formats, heap storage, unions, GOTO and labels. The limited coverage of ALGOL 68 makes the book unsuitable for a programmer wishing to learn ALGOL 68. It can, however, be recommended to any student following an introductory course in programming.

ANNE ROGERS (Bath)