To the Graduate Council:

I am submitting herewith a thesis written by Hsiang Y. Liao entitled "The Multiprogramming System of DEC-10 MUMPS." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Kevin C. O'Kane, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Vice Chancellor
Graduate Studies and Research

THE MULTIPROGRAMMING SYSTEM

OF

DEC-10 MUMPS

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Hsiang Y. Liao

December 1982

Dedication

This thesis is dedicated
to my parents, Mr. and Mrs. Samuel Kuo Y. Liao,
and to my wife, Mrs. Carol Pai-chin Lee Liao

ACKNOWLEDGMENTS

The author wishes to express his deep appreciation to Dr. Kevin C. O'Kane, his major professor, for his constant support and generous help throughtout the course of his graduate study. He is especially grateful for his excellent guidance and for providing his sophisticated UT-MUMPS system for this study.

Additional thanks go to the other members of his comittee, Dr. J. Michael Moshell and Dr. David W. Straight, for their interest and for their serving on the research committee.

The author would like to thank his friends Dr. Edwin W. McClain and Mr. Steven M. Satterfield for their constant friendship and helpfulness in reading and improving the manuscript.

Finally, the author is most appreciative to his family, especially to his brother H. L. Liao, for their encouragement and support so that the graduate study could be completed.

ABSTRACT

The project modified the UT-MUMPS DEC-10 version from a single user nonmultiprogramming system to a single user multiprogramming system. The system installed the multitasking facilities that support the user in designing, coding, and running his multitasking program, which increases the program efficiency by permitting overlap between the terminal input and the operations performed by the CPU.

The thesis describes the design considerations and implementation of the MUMPS multiprogramming system. It specifies the additional command of the UT-MUMPS system that supports the user in writing his multitasking program. The thesis gives an experimental example of the MUMPS multitasking program that runs by this multiprogramming system and points out the possibilities for further improvement of this system.

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

The UT-MUMPS interpreter system, created by Dr. Kevin C. O'Kane, has several different versions. The DEC-10 version is a nonmultiprogramming system. Since the MUMPS program has its conversational feature, in order to provide fast turnaround time of the user program, it is appropriate to overlap the I/O operation with the operations that are processed by CPU. The purpose of this project is to modify the DEC-10 monoprogramming MUMPS system into a multiprogramming system by means of multiprogramming simulation. This simulation makes the modified system perform as a multiprogramminng system, which supports the overlap between the terminal input and CPU operation. This system is a single user multiprogramming system; that is, one user can run several task programs simultaneously. The system processes the tasks in a multiprogramming fashion, with tasks being serviced on the highest priority and on a first-come first-served basis. The essence of the concurrent program is that of dividing a program into several tasks which run asychronously. In order to support the user in designing a concurrent program, the second purpose of this project is to provide the multitasking facilities for the user.

The design method of this multiprogramming system is a mixture of the top-down and the bottom-up manners. The

design includes determining the major modules which are to be inserted into the DEC-10 nonmultiprogramming MUMPS system to modify the system into a multiprogramming system. It further includes designing and writing the primitive operations for the modules and going up to determine the contents of those modules. Finally it involves iterating several times to adjust the modules and the primitives to each other.

The UT-MUMPS is a complex system. In order to adopt the system and to modify it, one must be aquainted with the system. A comprehensive overview of this system is given in the first chapter of this thesis. It describes the organization and logical structure of the system. The second chapter discusses the design consideration of the MUMPS multiprogramming system and the multitasking facilities, as well the strategies for their implementation. The third chapter describes the directives of the multiprogramming system that are designed for the user. Finally, the fourth chapter specifies how to use the MUMPS multiprogramming system with multitasking facilities. An experimental example of the multitasking program is provided in this chapter also.

CHAPTER I

SYSTEM ENVIRONMENT OF DEC-10 MUMPS

1. MUMPS OVERVIEW

"MUMPS" stands for Massachusetts General Hospital Utility Multiprogramming System. Although the MUMPS language was originally designed for bio-medical application, it is now being used in finance, industry, and even in office automation. It is one of the four ANSI standard computer languages, the other three being FORTRAN, COBOL, and PL/I. The other three are traditional "compile, load, and go" languages. In contrast, MUMPS is an interpretive language. It has a flexible nature, an interactive nature, and a high level language interpreter. There are several advantages to the MUMPS language:

1. User programs run directly in high-level language. With a traditional program language (e.g., FORTRAN) the user has to "compile and link" its source code into object code; then the machine runs the object code. The source code of MUMPS program does not need "compiling and linking" steps; it can load and run directly.

2. Writing and debugging a program can be done in a single process. With a conventional compiled language, when an error occurs the programmer must rewrite the correct code, recomple, relink, and rerun the program.

The MUMPS programmer can fix the program immediately with high level MUMPS language and resume debugging the program at the point where the error occured.

3. The total program size and the run-time memory of the MUMPS program compared to a compiler language are much smaller and save much programming time. For example, when one COBOL payroll program was rewritten in MUMPS, the items mentioned above (from code line number to programming time) of MUMPS version were approximately 8% of the COBOL version.*

4. The MUMPS language is flexible and powerful and has no requirement for pre-defined data (even no data declaration statement). Most programming languages require for pre-defined data types, lengths, or formats that must be worked around later. MUMPS not only has full range of fixed and floating point computation facilities but also provides extensive string processing and pattern matching facilities.

5. There are Z-type commands and Z-type special functions which are handled by routines INPUT and SVAR respectively. The user can design his own commands and functions and can attach them to the interpreter directly by tailoring or modifying those routines.

---

*"A Linguistic Comparision of MUMPS and COBOL" Thomas Maunnecke, Veterans Administration Hospital, Loma Linda, California, paper presented at the National Computer Conference, 1980.

MUMPS is a database-oriented language with the feasibility to create the interactive and conversational programs. The database of MUMPS is a disk-file-oriented data set, which is organized in a heirarchical tree structure. Since it is similar to the sparse matrix technique to allocate storage space only as needed; When no data is present, space is not allocated for empty data fields, so it uses less storage than regular structure, and the information is convenient to store or to retrieve. The MUMPS database is an integrated part of the language processor, there is no need for datatype of information. The procedures for modifying the data in a database are the same as those for modifying the data in user programs; therefore, it is not necessary to have a separate database administrator.

The main disadvantage of MUMPS is that it needs more CPU time. That is a general feature of an interpretive language, but it is not a problem for some minicomputer users.

The UT-MUMPS interpreter system, created by Dr. Kevin C. O'Kane at the University of Tennessee, is written in FORTRAN. (1) It was designed primarily for use on Digital Equipment Corporation DEC-10 hardware operating under TOPS-10, on PDP-11 hardware operating under RT-11 V3B and RSX-11M (and related systems such as IAS, RSX-11D, and

RSX-11S), and on IBM 360/370 systems under any operating system supporting FORTRAN-G. The RT-11 version of this system is a multiple user system. The DEC-10, IBM, and RSX-11M version are single user systems.

The version of the system on which this study is based is DEC-10 MUMPS. The following section will make a brief survey on this system.

2. Logical Structure of UT-MUMPS

The overall picture of UT-MUMPS DEC-10 version is shown in Figure 1-1.

In this diagram the first block ".RUN MUMPS" is a command typed by the user to start the system. A DEC-10 MUMPS user, under the TOPS-10 operating system, has free dynamic access to any device and file of the DEC system. The "INIT.MPS" routine, written in the MUMPS language and loaded from a disk file, is a "LOGON" procedure; which tells user: "The MUMPS system program is running."

The system base program is a small but necessary program. It is an internal driver program of the system, used to control the run-time execution. The system base program has the following functions that one can trace from Figure 1-1:

1. During system start-up, it brings the system into direct mode to read and execute the initial command line

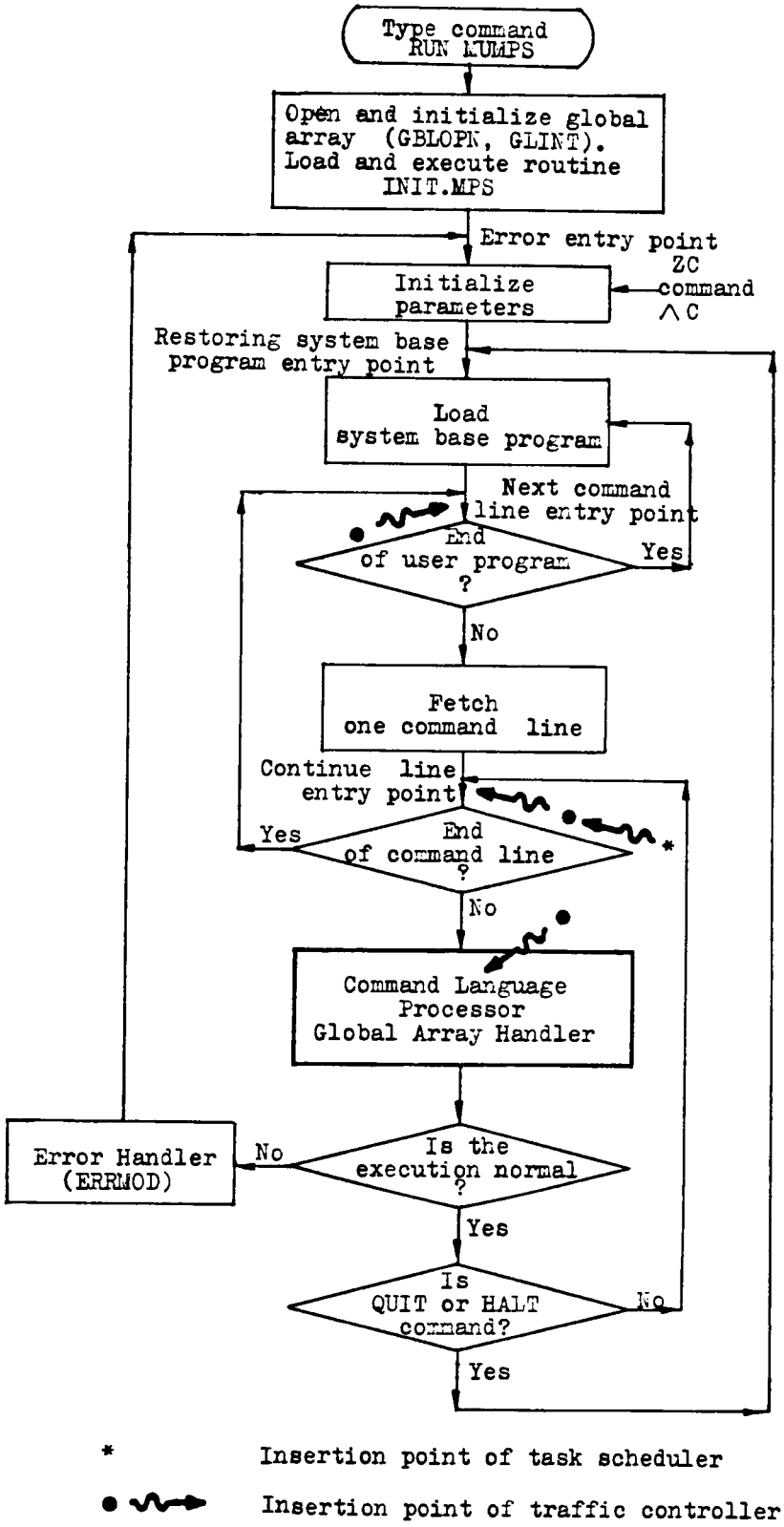


Figure 1-1. Logical Structure of the DEC-10 MUMPS System

which is typed by the user directly.

2. When the execution of the user programs is finished, or when it is forced by user to stop (for example, use QUIT, HALT, or ZC command), the system returns to direct mode automatically.

3. If the user program has crashed, the system does not exit from MUMPS; it restores the system base program and returns to direct mode automatically.

Why can the system base program fulfill those functions? It is not dificult to find the answer from its content. The system base program is written by MUMPS language as:

```
<tab>W !,">" R x X x G +00
```

where W !,">" prints the ">" character at the user's terminal to indicate it is in direct mode. R x reads an input command line, typed by user, into variable x. X x executes the command string line represented by variable x. G +00 transfers unconditionally to the beginning of the system base program, where the first "0" is a string character and the second one is a numeric used for a string delimiter.

If a user program is loaded into the user buffer space and the user types a ZRUN command, the system starts to execute the user program. With each cycle execution it fetches and executes one command line. In this process the

command language processor and global array processor are the major routines to fulfill this task.

3. Command Language Processor

The command language processor is the major part of MUMPS system program. Figure 1-2 shows its logical structure. Here as with the general rule, all command statements have to pass through the scanner and parser routines. (2) The reason that the processor distinguishes Z-type commands and non-argument form commands from other types commands and makes them by-pass of PARSE routine is to save execution time.

The PARSE routine, an expression handler, is the major routine of the language processor. It processes the argument of command with general form expression. Whenever there is a global array involved, the global array handler (GLOBAL routine) will be invoked. The symbol table handler (SYM routine) processes the local variables, putting the entries and values of each symbol into the symbol table.

There are two special modules in this command processor to process specifically the Z-type commands and the $Z-type functions that are designed by user; the INPUT routine and the SVAR routine handle the Z-type commands and the $Z-type functions respectively. If the user adds his own subroutines to these two modules or modifies a subroutine in

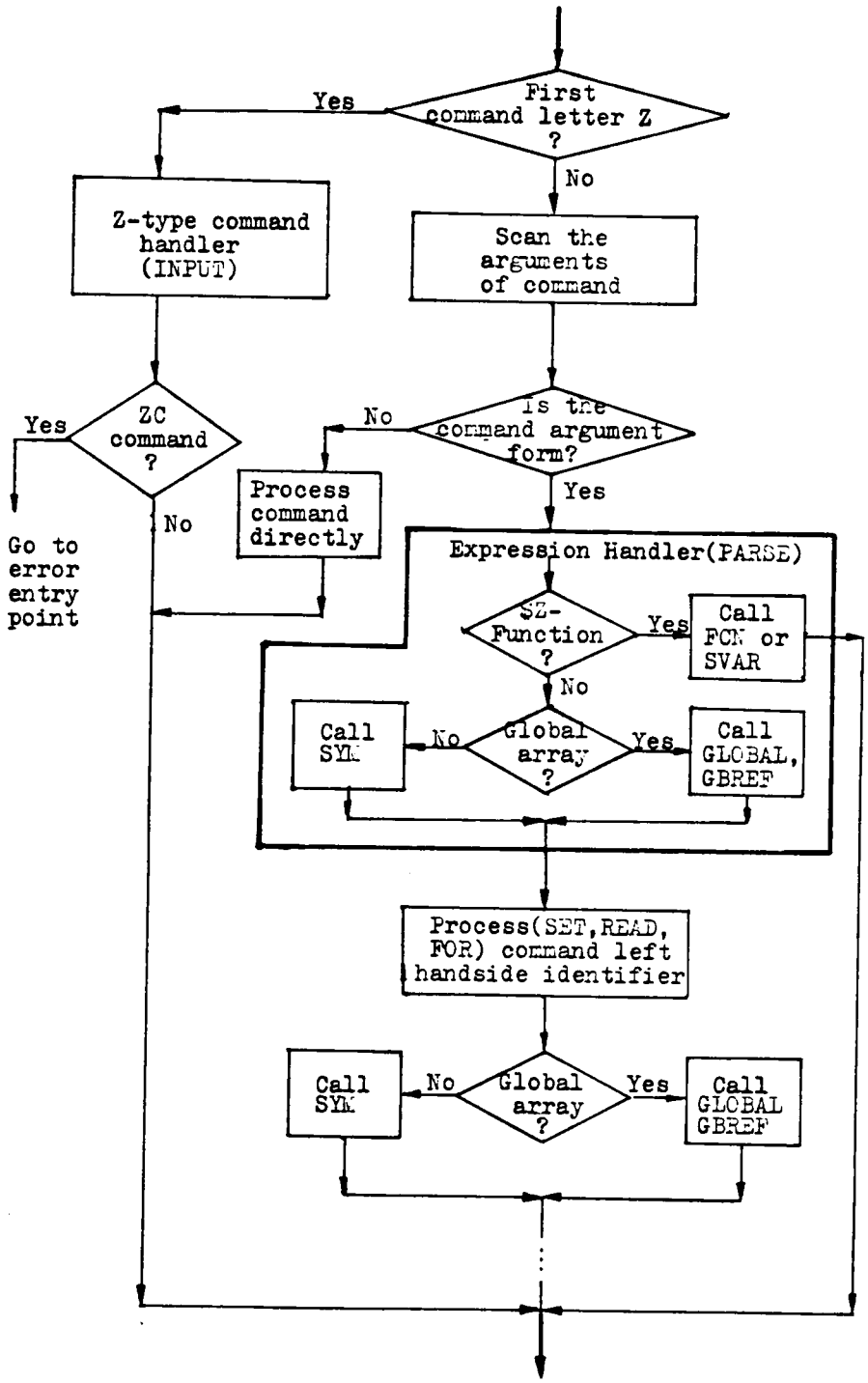


Figure 1-2. Logical Structure of the Command Language Processor

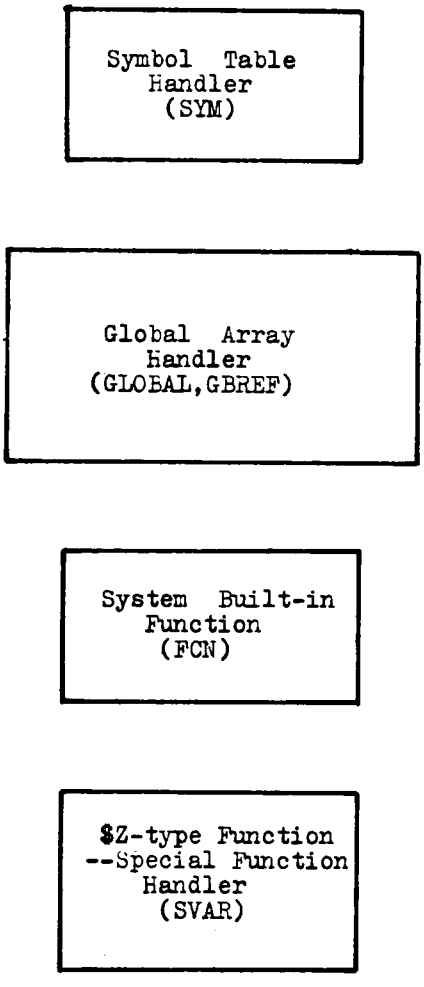


Figure 1-2. (Continued)

these modules, it is necessary to recompile, after compiling, then they link to the main interpreter directly.

4. Global Array Handler

The global array of DEC-10 MUMPS is disk-file-oriented database. It is unique to the MUMPS language. For example, A(1,43,5) is a global array which is symbolized by the circumflex "^." The system views the variable as a external file with tree structure, where "A" is the root of the tree and where the indices specify the path description to the leaves. The role of the global array processor in this system is to manage the global array data set with the hierarchical tree structure in the disk files. There are two major functions of MUMPS global array handler:

1. to initialize the global array disk file by routines GBLOPN and GBLINT.

2. to manage the tree structure database and execute the global array directives by routines GLOBAL, GBLREF, INSRT, and DEL.

5. User Storage Area

The MUMPS system allocates a user storage area of 4,000 characters. Figure 1-3 shows its memory organization. The system base program is loaded at the beginning address with a length of 22 characters. Following in order are (1) the

user programs, (2) the direct mode command, (3) the programs loaded as a result of the DO command, and (4) the dynamic stack of the parser. They all grow from low address toward high address, but only the symbol table grows downwards from the top of the user storage area.

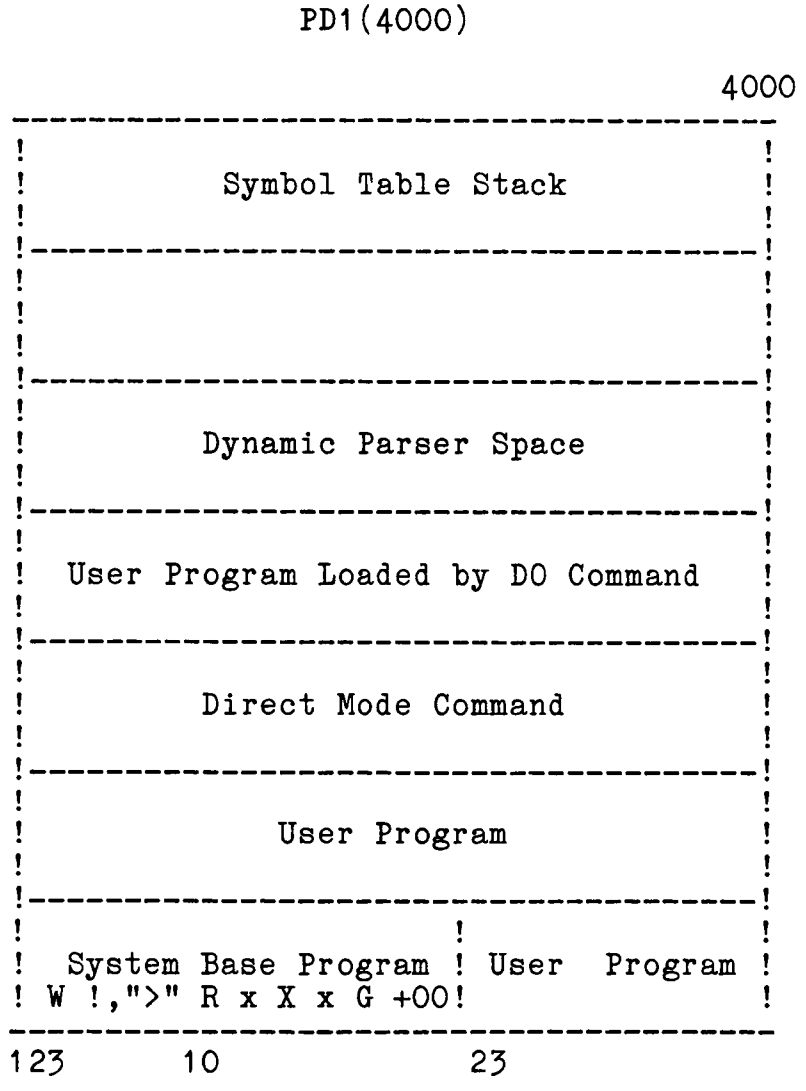


Figure 1-3 DEC-10 MUMPS User Storage Area Layout

6. RT-11 MUMPS

The UT-MUMPS RT-11 version is a multi-terminal system. This system can activate up to four terminals (including the console), with each terminal executing its own MUMPS programs. The RT-11 version is designed for operation on a small computer system where multiprogramming operating system is unavailable. The design of DEC-10 multiprogramming system in this project is adopted from the RT-11 version.

The RT-11 version system simulates many of the functions that belong to a traditional multi-user operating system. Consequently, it makes the RT-11 perform as a true multi-user system. The round-robin is the basic policy for time-sharing the various user programs. Each running user is allocated a time slice of one second. It places the user in a wait state mainly due to READ, HANG, and expired time slice conditions. The system has a swap management portion to swap the inactive programs to the disk file (named SWAP.MPS), in the meanwhile swapping in the active programs from SWAP.MPS. To facilatate the inter task communication, the RT-11 system provides communication facilities to send a message from one user partion to another user partion and receive a message (if any) from a communication vector.

CHAPTER II

MULTIPROGRAMMING SUPPORT OF DEC-10 MUMPS

1. Multiprogramming System Overview

In general sense, a multiprogramming system is an operating system that enables a computer to do several things concurrently. The concept developed rapidly in 1960s from early batch processing systems, such as ATLAS(1961), to large complex operating systems, such as MULTICS(1965) and OS360(1966). (7) There are good reasons to promote the developement of multiprogramming system. First, by executing several jobs simultaneously, multiprogramming and time-sharing systems can make use of the computer resources more efficiently. Second, real time transaction systems, such as a patient record system in a hospital, allow a number of users to access a single database simultaneously and to obtain responses in "same" time. The experience with early multiprogramming systems shows that the reliability of these systems is a problem (especially on machine level language). For example, OS360(1966) required 5,000 man-years of development effort. Because of its magnitude, OS360 had reliability problems. (3) This situation inspired computer scientists to search for new concepts.

One of the first concepts to emerge in an attempt to improve the design and implementation of multiprogramming

systems was the "process." A process (or "task") is a program module that consists of a data structure and a sequence of actions performed during execution of a sequential program. Dividing a program into several processes that are executed asychronously is an important idea for multiprogramming systems. A problem arises when processes have to share computer resources or cooperate on a common task, and then those processes must be executed synchronously. This is known as process synchronization. Dijkstra(1968) noticed in THE system that it is essential to perform the operations on a common variable strictly one at a time. Computer scientists have made efforts to invent safe methods for synchronizing processes which share resources or cooperate on a common purpose.

There are three major strategies for process synchronization: 1. Process synchronization by semaphore, 2. Process synchronization by monitor, 3. Message communication.

The semaphore, a integer signal associated with a computer resource invented by Dijkstra(1968), is an important idea for process synchronization. There is no problem in using the semaphore method to implement the process synchronization on a multiprogramming system. The synchronization mechanism of a monitor is similar to semaphore. The monitor, a set of program-defined

procedures, has a more structured format for system programs. (6) Hansen found that the semaphore alone does not fulfill the requirements of safety and efficiency in a dynamic environment where the process can appear and disappear at any time. (4) He suggested communicating between processes directly as a synchronization mechanism for two processes which interact each other.

These concepts, from process dividing to process synchronization, are essential to the design and implementation of a multiprogramming operating system.

PL/I extended the concept of multiprogramming and transfered the technique of concurent programming to the user by means of multitasking facilities. (5) User can use the multitasking facilities to implement the user application program as a concurrent program.

2. Considerations

The operating system, under which MUMPS runs, may be a multiprogramming system or a monoprogramming system. In the multiprogramming environment several jobs run simultaneously. The MUMPS system program plus the MUMPS user program as a whole entity is a member of the job group. All the operations of the MUMPS program are executed strictly in serial as a synchronous procedure. Since the MUMPS program has an interactive nature and the I/O

operation of DEC-10 MUMPS is a terminal oriented operation, the terminal input or output operations appear frequently in the MUMPS user program. When MUMPS interpreter executes the terminal I/O, the host time-sharing system has to relinquish CPU control from the MUMPS system program to other irrelevant jobs and to wait for the user response and the I/O operations of a channel. From the viewpoint of the MUMPS user program, the DEC-10 host multiprogramming operating system has nothing to do with the reducing of turnaround time of this job itself.

It is possible to reduce the turnaround time and raise the execution efficiency of a MUMPS program, because of several reasons:

1. One can view the MUMPS system program as a sub-level operating system and the user program as several tasks to be managed by the sub-level operating system.

2. It is possible to divide a MUMPS user program into smaller tasks* (or processes) that are executed asychronously.

3. It is possible to overlap the I/O operation with the tasks divided and cooperate sychronously in a common task. In other words, MUMPS system as a job member of the

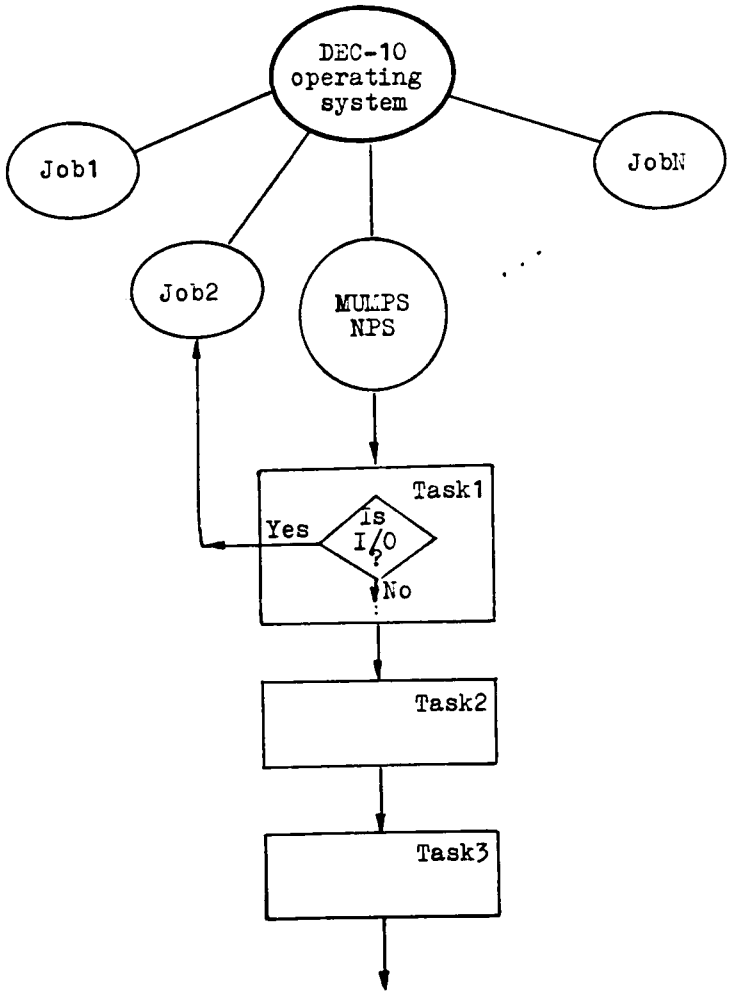
---

*Hereafter we prefer using the terminology "task" to express one of the processes or one of the sub-programs that are defined by user.

In other words, MUMPS system as a job member of the DEC-10 time-sharing system can still grasp the CPU control in hand while it executes the terminal I/O operations in the user program.

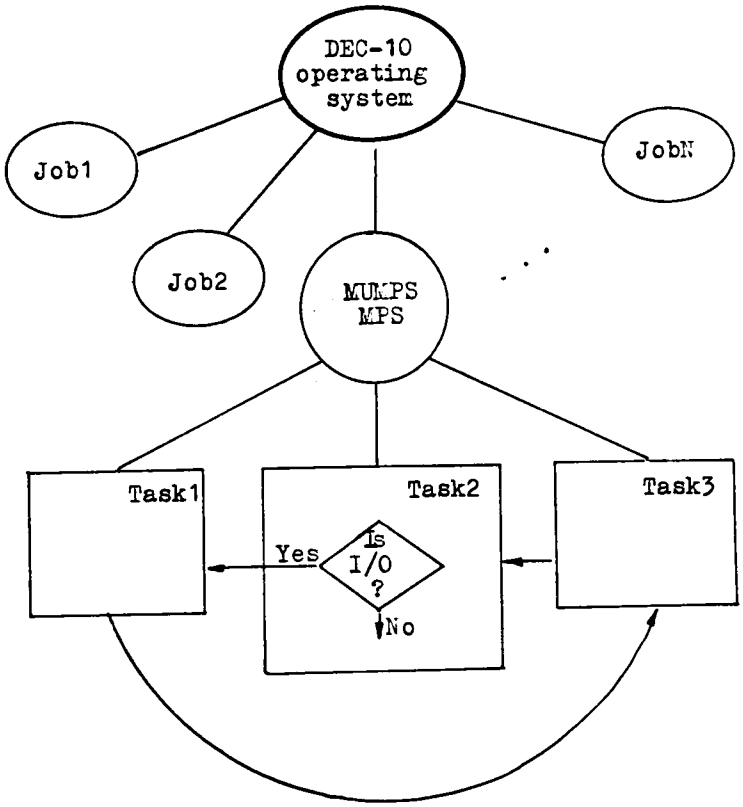
These idea can be put into practice by two strategies. The first is the modification of the DEC-10 MUMPS system from single user nonmultiprogramming system to a single user multiprogramming system. The environment of a DEC-10 user is rather convenient, the user having free dynamic access to any system device of file. It does not need to handle some processes like spoolin and spoolout. The main task of this multiprogramming system is to simulate a simple operating system with a process supervisor and I/O handler. The second part is installing the necessary multiprogramming facilities for the user, which support the user designing and writing a concurrent program. In the DEC-10 MUMPS multiprogramming environment the user utilizes this facilities to design, to code, and to run the programs with overlap of I/O operation.

Figure 2-1 illustrates the process structure of DEC-10 MUMPS system before and after modification. Thus, the practical meaning of MUMPS multiprogramming system is the running of several independent user programs with I/O overlap or the using of multiprogramming facilities to run a single concurrent program with I/O overlap. In this project



NPS--Nonmultiprogramming System

Figure 2-1. Process Structure of the DEC-10 MUMPS



MPS--Multiprogramming System

Figure 2-1. (continued)

it is assumed that the only I/O operation to be overlapped is terminal input.

The following sections describe the design and implementation of the DEC-10 MUMPS multiprogramming system.

3. State Model

Two different classes of process state are concerned--higher and lower. The higher is the state of MUMPS system program in the time-sharing system. It is "pending" state and "running" state. The "pending" state is defined as a state on which there is no task of "ready" state or "running" state in existence. It means that all tasks are in "waiting" state. When the "pending" state occurs, CPU control transfers from MUMPS to the host time-sharing system.

The user program, running in the MUMPS multiprogramming system, may consist of several tasks which are created by user. The lower class state is the state of task in the MUMPS multiprogramming system. This system defines four kinds of task states:

1. "Waiting" state. It is defined as one of the following four states:
    a. The final state of initialization of this task.
    b. The elapsed of time slice for this task.
    c. Starting execution of a I/O command.

d. Waiting for completion of external events or waiting for a message sent from other task.

2. "Completed" state. If all operations assigned by the task program are completed this task is said to be in "completed" state.

3. "Ready" state. If a task is not in "completed" state then its "ready" state is the exclusion of "waiting" state. It means the task is ready to be executed on CPU.

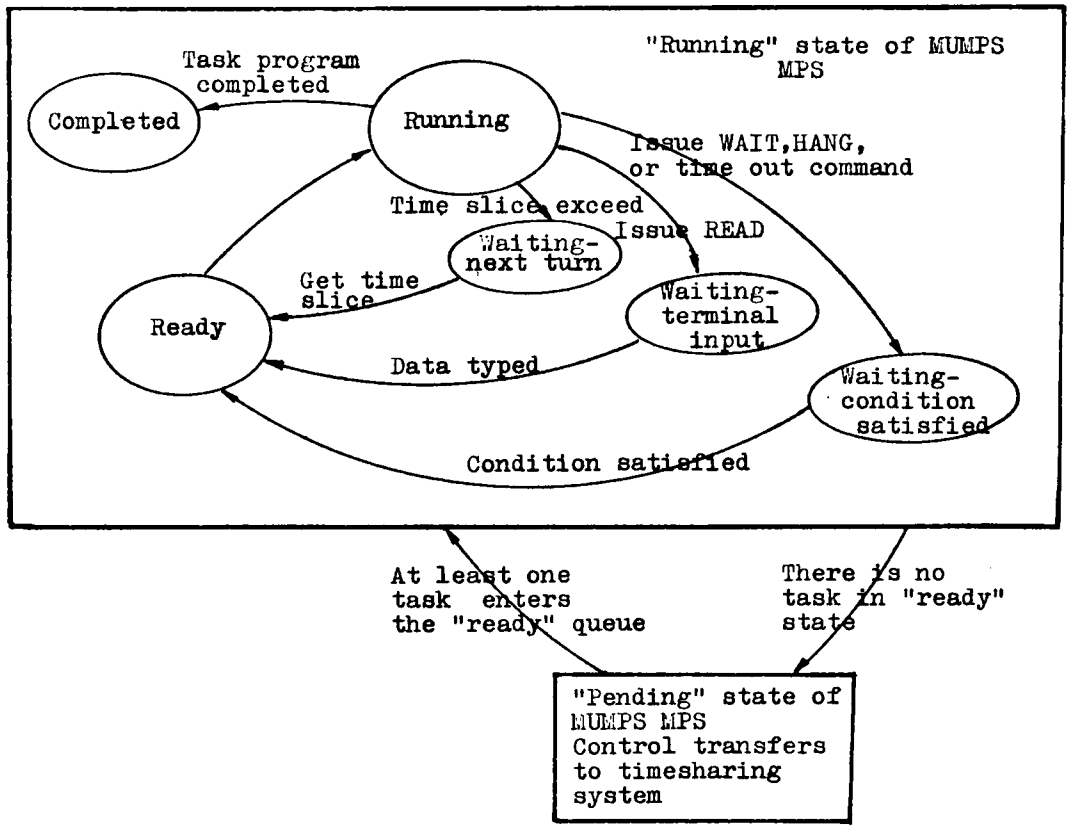
4. "Running" state. A task executing on the CPU is said to be in "running" state. When the state of a task changes from "ready" to "running," the "running" state task will receive CPU control immediately.

Figure 2-2 illustrates the state transition of this state model.

4. Data Structure of Task

The information of a task is completely characterized by its data structure. With FORTRAN language one can use variety of variables to describe the status of a task, and the index of a variable simply represents the identification of a task. Therefore a task is identified by an integer number. There are several characteristic variables in the implementation:

PR(ID): This is the priority variable of a task with task identification ID. Priority can be predefined in task



MPS--Multiprogramming System

Figure 2-2. State Transition of the MUMPS Multiprogramming System

initialization or assigned dynamically during execution. All priorities are relative values.

TSLICE(ID): This is the time slice of task with task identification ID. It is assigned in task initialization and can be modified dynamically.

MWAIT(ID): This is an integer code to indicate the reason of waiting.

RUNTYP(ID): This is an integer code to indicate the operation type.

MTSWAP: This is the central swap vector of task state. The major items that characterize the execution of a task are contained in this vector. It includes the user storage area (as shown in Figure 1-3) and system information of this task, such as program pointer PD1CUR,....

ARSWAP(ID): This is the storage area of central swap vector for task ID. The contents of MTSWAP would swap out to ARSWAP(ID) when the task ID is suspended from "running" state. ARSWAP(ID) has the same structure as MTSWAP. This is the major array and needs much space. The central swap vector is about 5.1K words, and MUMPS system needs about 33K words. Therefore, DEC-10 1090 configuration has enough memory to swap the central swap area for each task. It is not necessary to use disk file as swap storage area. From the viewpoint of memory there is no difficulty in designing the system with the task number in 10 order unless there is

need to increase the user program area in later design. The maximum task number set by this project is four.

There is no variable to describe the state of a task directly. It can be determined from state queues.

5. Structure of Queues

There are two queues used to record the information of states and the order of tasks--"ready" queue (symbol READYQ) and "waiting" queue (symbol WAITQ). Actually the "ready" queue is used as a scheduling table. The queue is an integer array beginning from first element. Each element contains the identification(ID) of a task which has the "ready" state or "waiting" state. To indicate the length of a queue there is an integer pointer for each queue. Figure 2-3 illustrates the structure of "ready" queue and "waiting" queue. This example shows the following facts:

1. There are three tasks in "ready" state with task identification 3, 2, 4.

2. The task with identification 1 is in "waiting" state.

The primitive routines applied to these two queues are very common as queue (or chain), dequeue (or unchain), sort queue (by priority), and rearrange queue (to exclude blank).

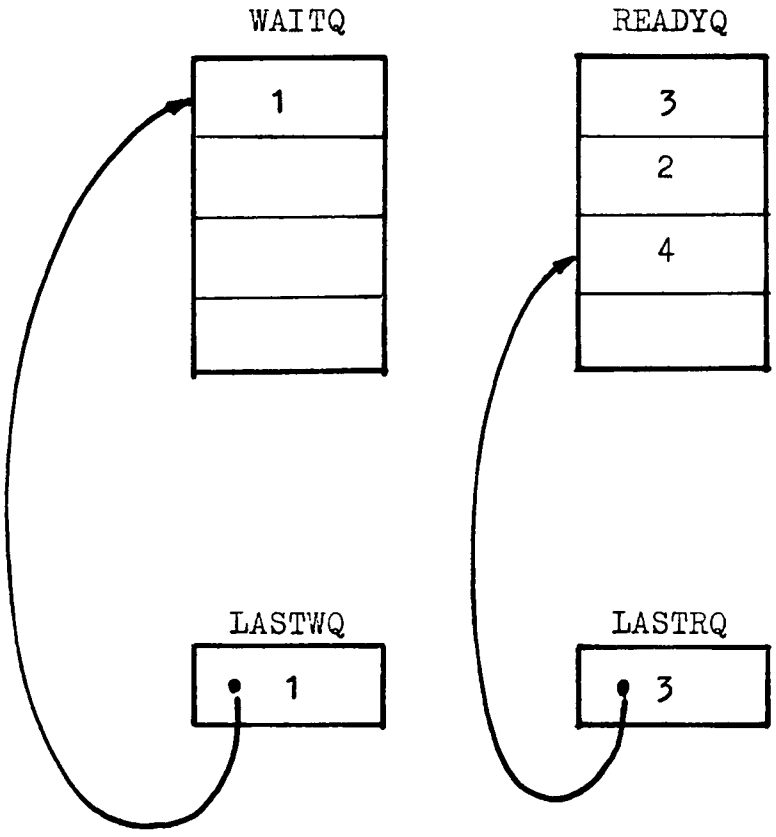


Figure 2-3. "Waiting" Queue and "Ready" Queue

6. Design and Implementation of Dispatcher

Based on the UT-MUMPS DEC-10 version, the MUMPS multiprogramming system is simulated by adding a multi-task dispatcher to the nonultiprogramming system. This project adapted and developed the multi-user dispatcher of the RT-11 version as the basic scheme of the dispatcher in the multiprogramming simulation.

The multi-task dispatcher has the ability to perform the following functions:

1. Initializing of tasks.

2. Keeping track and update of the status of tasks, resources, and external events. It includes the functions to simulate the interrupt and interrupt handler. The major interrupts are timer (time slice exceed) interrupt, I/O (terminal read) interrupt, and WAIT, HANG, or time out command interrupt. Traffic controller routine performs those functions.

3. Scheduling a task to control CPU, meanwhile relinquishing and deallocating CPU from a task. Those functions are performed by a task scheduler routine.

Figure 1-1 shows the natural insertion points of these routines, where "." and "*" represent traffic controller and task scheduler respectively. The task scheduler performs its activities along with the traffic controller. It requires the work cycle of traffic controller and task

scheduler to perform their activities not only at the interrupts mentioned above, it also performs at the end of execution of each command word, at the point where a task is finished or issued a cancel command, or at a task issued a changing priority or time slice command. For each cycle the traffic controller checks and updates the status of tasks, then the task scheduler routine plays its scheduling role immediately.

The following sections describe those routines in more detail.

7. Initialization Routine

The initialization routine is invoked by user command. It serves the purpose of defining a user task in the system. The definition of a task at the intial time:

TASK(I)={ARSWAP(I),PR(I),TSLICE(I),MWAIT(I)}

where{ } is a set. I is the identification of a task. the meaning of the variables is the same as in the previous sections.

The operations of a initialization routine are as followings:

1. Acording to the argument in user's command, set up the identification of the task with an integer number, intializing the priority variable and time slice variable of the task.

2. Load the user's program from a disk file created for this task and copying the system base program into memory area ARSWAP which corresponds to this task identification.

3. Initializing parameters both of the system and the task, for example, PD1LEN (length of the task program), PD1CUR (starting pointer of the task program).

4. Chaining the identification of this task on the "waiting" queue, puting the task in waiting state. Updating the pointor of the last element of waiting queue.

5. Setting a flag of this task on to express this task as defined.

8. Traffic Controller

The traffic controller serves the purpose of creating a multiprogramming environment. It runs between tasks to keep track and update of their status. In order to do so, the traffic controller has to recognize what kind the user's command is, using instructions of DEC-10 MACRO-11 INCHRS or INCHRW to sense whether or not the user's terminal input is ready, using timer routine SECNDS to calculate how much the time slice of running task is consumed and how much the waiting time of events remain, and keep track of the inter task comunication vector to perform the services for the message communication.

The overall logical structure to the traffic controller is shown in Figure 2-4.

The traffic controller checks the status of tasks of "running" and "waiting" state cyclically. A "running" state task in the system would be "interrupted" to be a "waiting" state task if the task issued a READ, ZWAIT, or if the HANG command or the time slice of this task has expired; the "running" task becomes temporarily ineligible to run for reasons refered to above. In the meanwhile the traffic controller unchains the "running" task from the "ready" queue and chains it to the "waiting" queue. Similarily, the "running" task would be deleted from the "ready" queue if the task issued a cancel command or the task program is finished.

After updating and tracking the running task, the traffic controller calls for routine CKWAIT to track and update the status of all tasks which are chained in the "waiting" queue. If the task in the "waiting" queue is satisfied of pending conditions corresponding to this task, then he switches the state of this task from "waiting" state to "ready" state. In the meanwhile he unchains the task from the "waiting" queue and chains it to the "ready" queue. He deletes the identification from "ready" queue when the task is in "completed."

Routine CKWAIT is a "waiting" state handler which

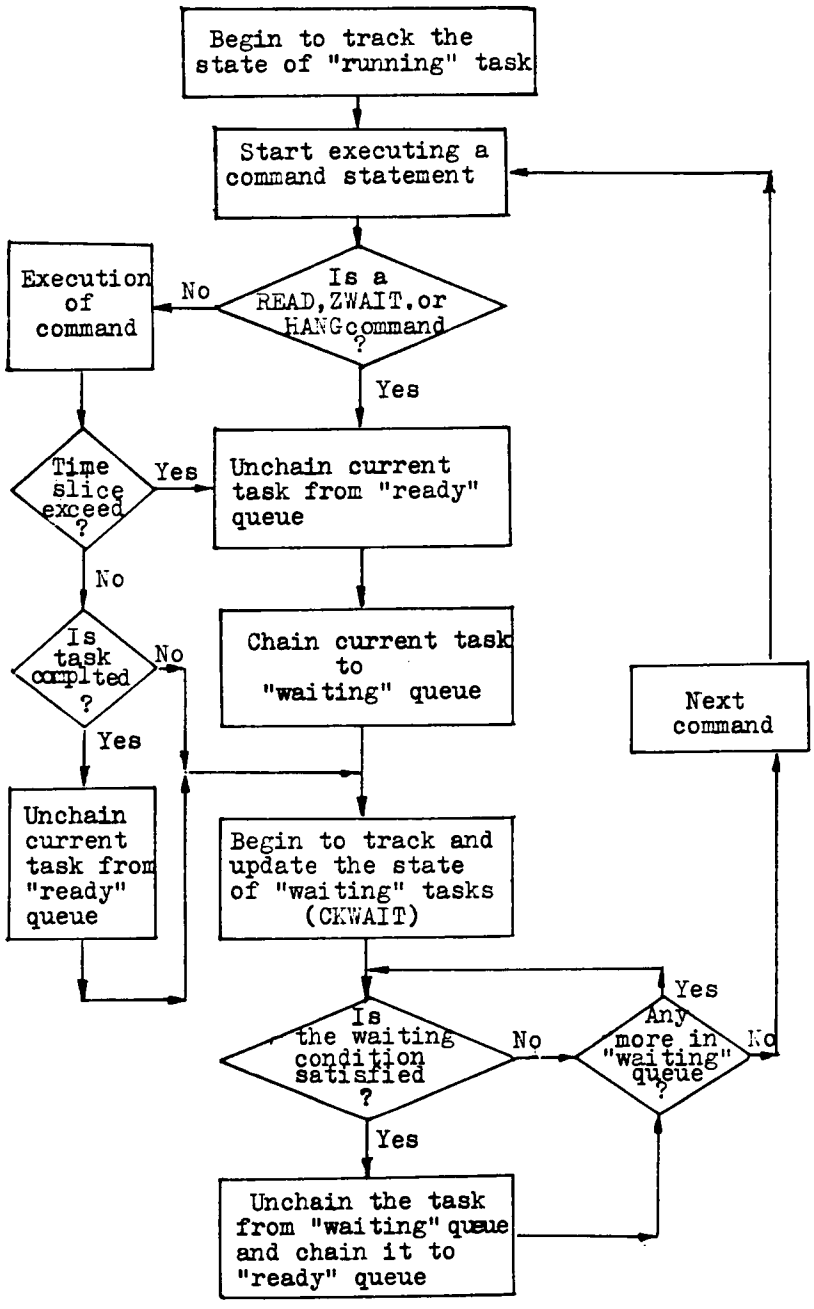


Figure 2-4. Logical Structure of the Traffic Controller

handles the criterion of state transition and makes decision to switch the state from "waiting" to "ready." Its operations simulate the functions of an interrupt handler.

In the MUMPS multiprogramming system there are eight kinds of "waiting" states which are concerned with different corresponding MWAIT codes. According to the MWAIT code routine, CKWAIT performs the different functions as follows:

1. MWAIT=1 is corresponding to situations such as waiting for task initialization or time slice excess. The handler puts this task on "ready" queue immediately, which will join the next turn of the round robin.

2. MWAIT=3 is waiting for line oriented READ, and MWAIT=4 is waiting for direct mode READ command character input. In both cases CKWAIT senses the user's terminal periodically, and as soon as the first input character is typed, it changes this task to "ready" state.

3. MWAIT=5 and MWAIT=6 are similar to MWAIT=4 with time out respectively. In addition to sensing user's terminal input CKWAIT, it checks the system timer to see if the time-out is expired. If so, it switches this task to "ready" state.

4. MWAIT=7 means that the task is waiting for a message from any task or from a specific task. Routine CKWAIT checks the message flag MSGFLG and the message queue MSGQ to determine if a message is sent. In addition, CKWAIT

checks the timer to see if the waiting time is exceeding the maximum value set by user. If the message is sent from the desired task or if the maximum waiting time is exceeded, it changes the state to "ready."

5. MWAIT=8 and MWAIT=9 are the cases in which the task is waiting for completion of either one (code 8) or several (code 9) tasks. Routine CKWAIT searches "ready" queue and "waiting" queue. If the identification of the event task is not found, it implies that the event is completed and changes the waiting task to "ready" state.

9. Task Scheduler

The task scheduler schedules and runs the task that is eligible to be run. It shares the CPU among the several tasks listed on "ready" queue at any one time. The task scheduler is guided by the following policies:

1. Highest priority. The scheduler insures the highest priority task receives the CPU. This is a preemptive policy. When an important (thus higher priority) task becomes ready after being "interrupted," it will preempt the CPU from a lower priority running task. All priorities have relative values between 0 and 9.

2. Round robin. In the case of highest priority ties, it is reasonable to schedule the tied tasks on a FIFO basis. Each task in turn is running to its own time slice limit, e.g., five seconds.

Figure 2-5 shows the logical structure of the task scheduler of multiprogramming MUMPS system.

To implement the scheduling policies, it first uses routine PRSORT to sort the "ready" queue by priority; then it picks out the first element from it as the "running" state task.

For the sake of convenience in explaining the swap management, its functions are included in the task scheduler. When the current new "running" task is not the same previous "running" task, the scheduler has to deallocate the old "running" task by swaping out the central swap vector of the "running" task state to the corresponding storage area ARSWAP(old task). In the meanwhile, the task scheduler allocates the current "running" task. It swaps in the task state vector from storage area corresponding to the current "running" task ARSWAP(new task) into the central swap vector area MTSWAP. Since the central swap vector contains everything necessary to continue the task program, the task shall be executed smoothly.

10. Message Communication

In a system of parrallel, cooperating tasks, a mechanism must be provided for the sychronization of two tasks during a transfer of information. In general, the following command formats of inter task communication are available as multitasking facilities for user interface:

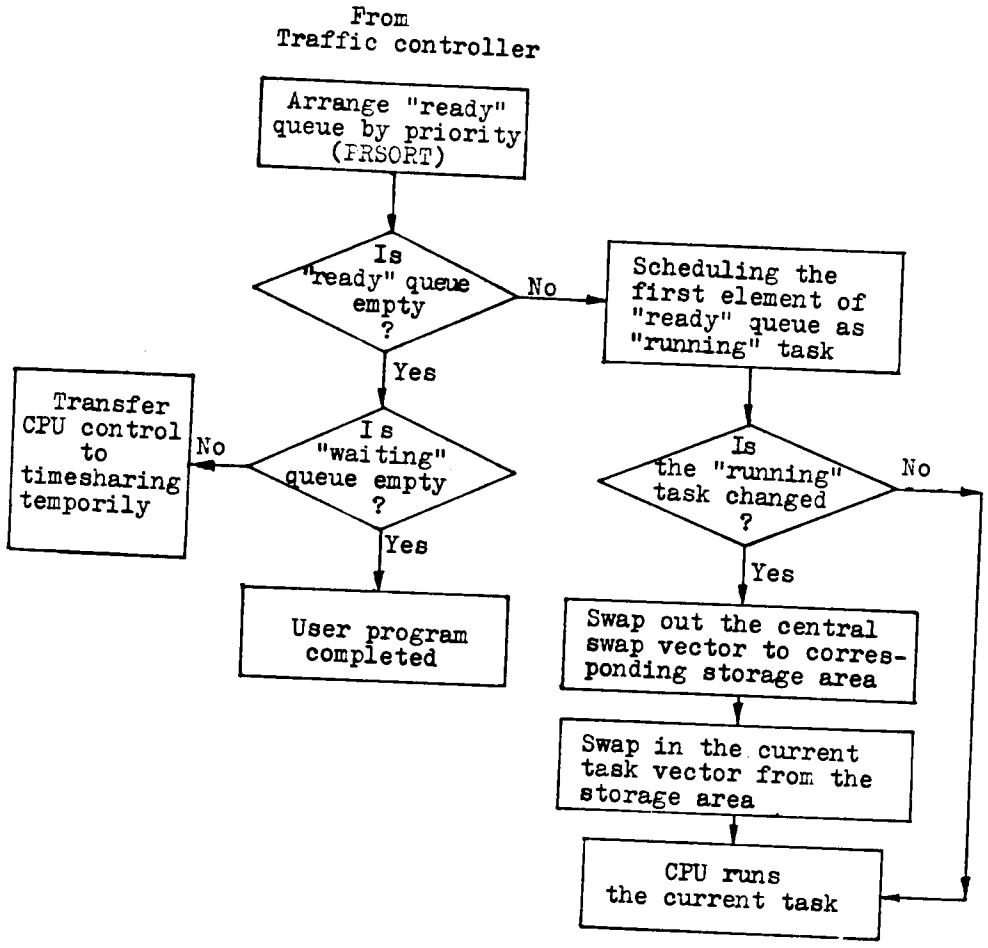


Figure 2-5. Logical Structure of the Task Scheduler

SEND message(receiver,message)

WAIT message(sender,message)

RECEIVE message(sender,message)

To implement these communication facilities, the following variables (or data) structure of message-related variables (or data) are used:

MSG--message buffer, containing the inter task communication vector.

MSGFLG--message flage, indicating whether or not the message buffer is occupied.

MSGQ--message queue, listing the task identification number of the message receiver.

LASTMQ--pointing to the last task identification number in the message queue MSGQ.

MFROM--pointing to the task that sent the message.

FRMTSK--pointing to a specific task that is expected by a requesting task to send a message.

On the basis of data structure, it is easy to implement the above user interface. "SEND message" copies a message into the buffer (MSG), turns flag on (MSGFLG=1), puts the identification of receive task number on message queue (MSGQ), and lists the identification of sender on MFROM. "RECEIVE message" is activated if message flag is on and the receiver listed on MSGQ is identical to the current running task. The receiver displays a message on the screen and turns off the message flag. The "WAIT message" delays the

requesting task until an appropriate message sender arrives onto its queue. With this command the requesting task is allowed to run only if message flag is on, if the receiver listed on MSGQ is the same as the identification of requesting task, and if the message sender on MFROM is identical to the specific task (on FRMTSK) which is required by the requesting task (if the command requires the sender is a specific task).

The MUMPS multiprogramming system implements these operations in modules INPUT, SVAR, and CKWAIT. The implementation sets only one message buffer. This means that every task can only send or receive one message at a time.

This communication scheme has an advantage in that it does not need to be aware of the complete knowledge of the existence of other tasks until it receives message from them. (3) Since the multiprogramming system is dynamic in terms of tasks that can appear and disappear at any time, in order to prevent a requesting task from waiting the message from a task which has disappeared or if there is no message sent to this requesting task, the implementation gives a maximum waiting time to the "WAIT message" command; the requesting task is allowed to run if the waiting time is exceed the maximum waiting time.

CHAPTER III

DIRECTIVES FOR MUMPS MULTITASKING FACILITIES

1. Commands and Functions of MUMPS Language

MUMPS language does not demand rigid structure; the major components of a MUMPS user program are command and function. Each statement begins with unique command word which is abbreviated to a single character for simplicity. In general case (there are exceptions) the syntax of a MUMPS statement may be expressed as:

<command word>[<post conditional>] <argument>[,<argument>,...]

post conditional=:logical expression

argument=:expression!identifier!label!file specification.

where [ ] expresses "option," and ! means "or."

The ANSI MUMPS system permits the user to incorporate his self-designed commands and functions as Z-type commands and $Z-type functions. UT-MUMPS DEC-10 version contains 14 Z-type commands and 6 $Z-type functions to increase the repertoire of MUMPS directives. (1) In terms of function; Z-type commands consist of system commands (e.g. ZRUN, ZLOAD,...ect.) and editor commands (e.g. ZK, ZL,...ect.). Compared to normal MUMPS command the syntax of Z-command is simpler; in general it has the format:

<numeric argument>Z<command word> <literal argument>

where the numeric argument is an integer number and the

literal argument is keywords, file specification, or character string.

The syntax of the special built-in function has the form:

$Z<symbol word>[(<argument>,<argument>,...)]

where the argument:=number, string, or expression.

The MUMPS multiprogramming system supports several Z-type commands and Z-type functions. They form the major parts of the UT MUMPS multitasking facilities.

2. Z-type Commands

There are seven Z-type commands which are supported by the MUMPS multiprogramming system. Following are the detail specifications of each command. As a convention in the discussion the capital letter expresses the command word and the small letter word expresses the specification.

1. ZINIT command.

The ZINIT command has the following format:

ZINIT <task name><task ID> [P<priority> T<time slice>]

where the symbols are the same as the convention above. There are three arguments; the task name can be any literal of length three and the identification must be an integer number which satisfies the requirement of being larger than 0 and smaller than 5. Each task has its unique identification number. It is necessary to note that the

task name and identification number must match the DEC-10 file name of this task. Priority means the relative priority number of this task; the lowest priority is 0 and the highest is 9. Time slice stands for time slice of this task, it is a positive integer number and less than 100 with the unit of seconds. The default of the priority and time slice are the priority number 2 and 5 seconds respectively.

The ZINIT command will cause the user created task program to be loaded into the swap vector storage area of the MUMPS system; in the meanwhile assigning the priority variable and the time slice variable associated with this task; then chaining this task identification on "waiting" queue. It notifies the system to create a multiprogramming environment. The ZLOAD command still is legal in the initialization of multiprogramming system, it is used to create a nonmultiprogramming environment and load the user program in the system.

Examples:

a. >ZINIT TSK2 P4 T10

b. >ZINIT TSK3 P3

c. >ZINIT TSK4

The command statements in the above examples create three different tasks that will be executed in a multiprogramming environment with multitasking facilities. In example (a) the task with name TSK2 has priority 4 and 10

seconds time slice; in example (b) TSK3 has priority 3 and default time slice (e.g., 5 seconds) and their DEC-10 file names are TSK2.MPS and TSK3.MPS respectively.

2. Priority command (abbreviation: P).

The ZP command is used to modify the priority of a task dynamically. It has the following format:

ZP <task ID> <priority>

Example:

> ZP 2 3

causes the priority of TSK2 in example (a) changing from 4 to 3.

3. Time slice command (abbreviation: T).

The ZT command is used to reset the time slice of a task dynamically. The ZT command has a similar format with ZP command:

ZT <task ID> <time slice>

Example:

> ZT 2 3.

It changes the time slice of TSK2 in example (a) to 3 seconds regardless of its initial value.

4. Cancel command (abbreviation: C).

The ZC command cancels a task from user's program. When executing this command, the system searches the task identification in the "waiting" queue and "ready" queue, then deletes it from the queue if found. The ZC has the

following format:

ZC <task ID>

Example:

> ZC 4.

It causes the task--TSK4 of example (c) to be cancelled.

5. Event command (abbreviation: EVT).

When the EVT command is used in a task, it causes the task declared to have the event attribute and is associated with an event variable. The EVT command has the following format:

ZEVT EVT1=<task name><task ID>[,EVT2=<task name><task ID>,...]

where EVT1 and EVT2 are event variables, the maximum number of event variable is four.

Example:

> ZEVT EVT1=TSK3,EVT2=TSK4.

It illustrates that there are two events to be concerned; one of them is TSK and another one is TSK4. They are represented by event variables EVT1 and EVT2 respectively.

6. Wait event command (abbreviation: WAIT EVT).

The WAIT EVT command will cause the requestor task which issues the command to wait for the completion of the event that is specified in this command. When the WAIT EVT command is executing, the system checks the completion variable of that event periodically; the requestor task will be active after the completion variable is set. The

following is the format of WAIT EVT command:

ZWAIT <event variable> [<operand><event variable>]

where operand is & or /, representing "and" or "or" respectively.

Example:

d. > ZWAIT EVT1.

e. > ZWAIT EVT1&EVT2.

f. > ZWAIT EVT1/EVT2.

In example (e) the task, issued ZWAIT command, has to wait the completion of both EVT1 and EVT2. In example (f) the task has to await the completion of EVT1 or EVT2.

7. Wait message command (abbreviation: WAIT MSG).

The WAIT MSG command will cause the requestor task which issues the command to wait for a message which is sent to it. The WAIT MSG has the following format:

ZWAIT MSG[<task ID>[:<time>]]

where the task ID option means the specific message sender is requested; if it is absent the message sender can be any task. The time option requests the maximum waiting time in units of seconds; if this option is absent the default maximum waiting is 5*time slice (to wait two round robin cycles approximately).

Example:

g. > ZWAIT MSG2:20.

h. > ZWAIT MSG:20.

i. > ZWAIT MSG.

In example (g) the requestor task is waiting the message sent by TSK2 with the maximum waiting time being 20 seconds. In example (h) the requestor task is suspended to await a message which may be sent by any task. The example (i) means the maximum waiting time is about 50 seconds (the requestor time slice is assumed to be 10 seconds).

3. Special Built-in Functions

There are five additional built-in functions supported by MUMPS multiprogramming system. They are $Priority, $Time slice, $Completion, $Zsend, and $Zreceive. followings are the specification of each command.

1. $Priority (abbreviation: $P).

The $P command returns the priority of a specific task. It has the format:

$P <task ID>.

For example, $P2 in (a) of last section returns 4.

2. $Time slice (abbreviation: $L)

The $L command returns the time slice of a specific task with the format:

$L <task ID>

3. $Completion (abbreviation: $C)

The $C commaand returns a Boolean "true" or "false" to represent the completion or incompletion of the specific

task. The completion function has the following format:

$C(<task ID>).

4. $ZSend (abbreviation: $ZS). The task, invoking $ZS, sends a message to a specified receiver task. When the command is executed, the $S function loads the string of message into the inter-task communication vector and lists the receiver task on the message queue. The function returns a value of 1 if the transaction is successful. Since the system only provides one message buffer, the message is not loaded if there is a pending message in the buffer already; in this case the transaction is not successful, the function returns the 0 value. The $ZS function has the following format:

$ZS(<task ID>,"<TEXT>")

where the task ID is the task ID of message receiver, and text is the string of message, in quotes, being sent.

For example, TSK2, in example (a) and (b) in the last section, issues the following command:

$ZS(3,"communication test")

means that TSK2 sent the message "communication test" to TSK3.

5. $ZReceive (abbreviation: $ZR).

The receiver task issues a command with $ZR function to read the message from inter-task communication vector. The function returns a value of 0 if no message has been sent to

the requesting task; or it returns a string containing a message sent to the requesting task. The message flag is cleared when the message has been received; then the communication vector can be used again. For example, after TSK2 issued the message mentioned above, the following statement of TSK3:

```
W !,$ZR
```

will display message--"communication test" on screen.

CHAPTER IV

SPECIFYING USE OF MULTITASKING FACILITIES

1. User Program

There are two categories of user program: nonmultitasking programs and multitasking programs. The nonmultitasking program is a single task program which consists of synchronous operations; The multitasking facility is of no use for this kind of program.

The necessary conditions to design a multitasking program for the multiprogramming MUMPS system can be described as:

1. In the task program there is a READ command (with or without time out) to input data from terminal, or a command (HANG or WAIT) to temporarily suspend the operation for the applicational requirement of the user.

2. There are asynchronous operations in the task.

Satisfying thes two conditions simultaneously makes it possible to take advantage of using multitasking facilities in this system. That is, to increase the program efficiency by permitting overlap between the operations which need waiting time (e.g., terminal input) and the operations performed by CPU. The cooperation and synchronyzation of tasks are vital important for the success of writing a multitasking program.

2. Procedures to Run a Multitasking Program

To facilatate the DEC-10 MUMPS user the procedures of running a MUMPS multitasking program are designed with conformity with the regular DEC-10 version, except the using of ZINIT command. (1)

1. Preparation of the modules. If the user wishes to extend or modify the Z-type command, the Z-type function, or the global file name and extend, he should recompile the routines related, then link edit the modules. The commands of this procedure can be typed as:

```
.DO <COMPIL>.MIC
.DO <LNKR>.MIC
```

where the <COMPIL> and the <LNKR> are the names of DEC-10 compile routine and link routine respectively.

2. Set up the search path. User types following command to access the main modules of MUMPS:

```
.PATH <disk name where MUMPS.EXE resided>
.PATH /LIB:[<PN>,<COUNT>]
```

where the MUMPS.EXE is the module of object code of MUMPS system program and <PN> <COUNT> are project number and acount number respectively.

3. Start the MUMPS multiprogramming system. The user types the following commands to initiate the MUMPS interpreter and to create the global array file:

```
.RUN MUMPS
```

>ZG

4. Initialization of task. For multitasking programs or multiple independent programs, the user types the ZINIT command to initialize each task program (detail refer to the specification described in Chapter III). For single task program it is legal both to use ZINIT or ZLOAD command (in this situation the ZLOAD command is more efficient than ZINIT command).

5. Execution of program. User types ZRUN command to run the user program as the regular DEC-10 MUMPS. After a user program is terminated, the system returns to direct mode; The user may then run other programs or type ^C or ZE to exit from MUMPS system.

3. Experiment of a MUMPS Multitasking Program

This experimental program is a result extended from a single task program in the UT MUMPS manual. (1)

There are four tasks which conprise this program. The identification of the first task is TSK1 ; it creates several elements in the global arrays--"PATH" and "PATH1". Actually, each element of them is similar to a subroutine. "PATH1" contains the subroutines to collect the data and put them in a database. "PATH" contains the subroutines to extract and display the elements of the database. The second task is named TSK2. TSK2 uses elements of "PATH1"

and "PATH" created by TSK1 to store and retreive the data. This is the real-time part of the program. The third task with identification of TASK3 is to calculate statistics (e.g., GPA of a student and the overall grade average of all students). This task uses collected data to process background calculations. The last task--TASK4 is used to issue a report on all collected data and calculated statistics. All real-time READ statements are in TSK2; it's purpose is to raise the efficiency (e.g., the turnaround time of the program) by means of overlap asychronous task with TSK2.

There are two schemes to carry out the task overlap:

1. Execute TSK1 first, after all the subroutines (i.e., global array sets) of the program are set up, then execute TSK2 and TSK3 simultaneously. The last one to be executed is TSK4. This scheme sets TSK1 with the highest priority and uses the ZWAIT MSG and ZWAIT EVT commands to control the sychronyzation of tasks.

2. Execute TSK1, TSK2, and TSK3 simultaneously. That is first set up the parts of the subroutines which are necessary immediately; the creation of the remainder of TSK1 would utilize the waiting time of READ operation. In this scheme the priority of four tasks are the same value. The time slice of TSK1 should take a suitablely small value. The flowcharts and overlap maps of two schemes are shown as

Figure 4-1 and Figure 4-2 respectively. The programs and its execution are listed in appendix.

4. Conclusions

The preceding chapters have described the design and the implementation of a DEC-10 MUMPS multiprogramming system with the installation of multitasking facilities. The experiment of a four tasks user program with three tasks overlapping demostrated how the MUMPS multiprogramming system provides the necessary facilities which make it feasible for the user to perform his independent programs with terminal input overlap or his multitasking program with both terminal input overlap and tasks synchronization.

The following are the possible improvements of this project:

1. Increasing the number of communication buffers, thus making it possible to send the number of than one message.

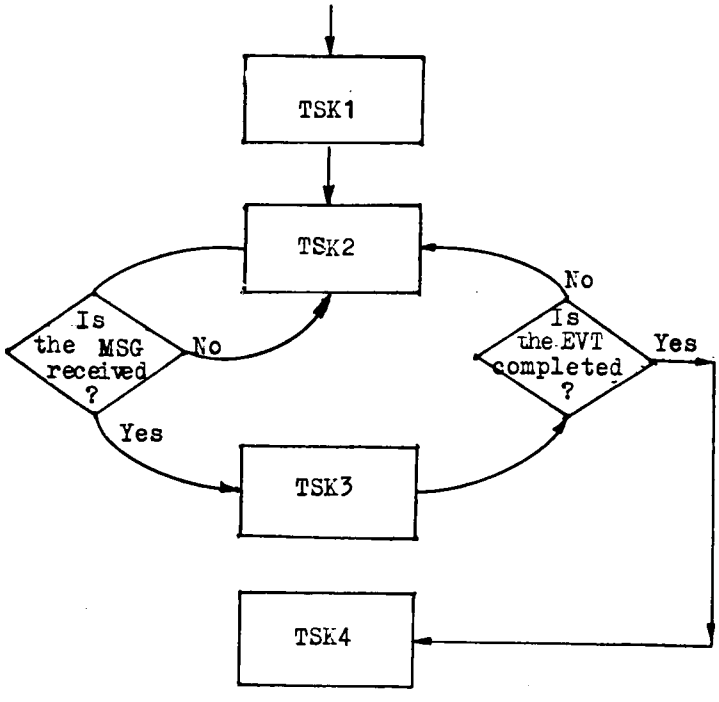
2. Synchronizing the terminal input between tasks.*

3. Adding more I/O items than terminal input.

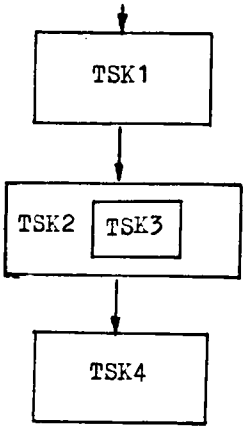
4. Modifying single user DEC-10 MUMPS with multi-tasking facilities to multi-user with those facilities.

---

*This may not gain much advantages for single user system; since the user tasks are easy to be blocked when more than one task has READ operations.

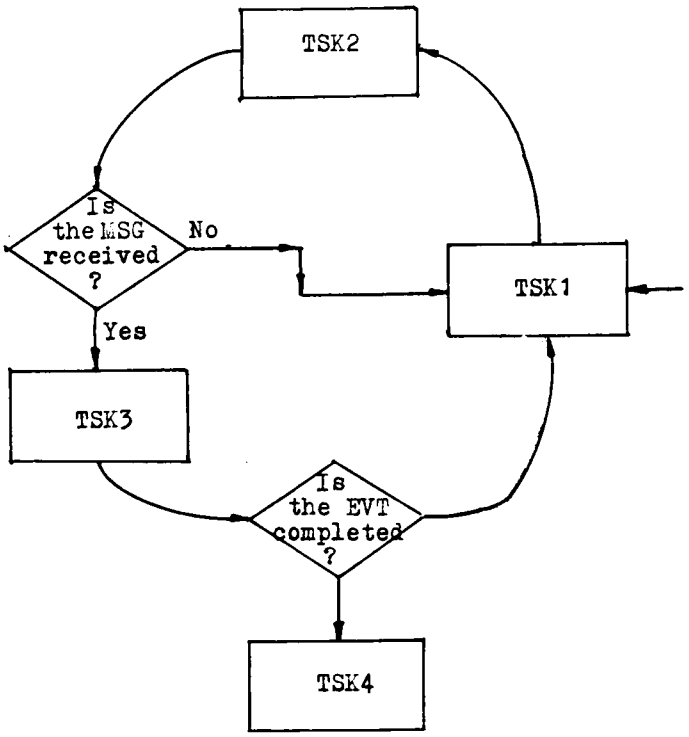


A. Logical structure

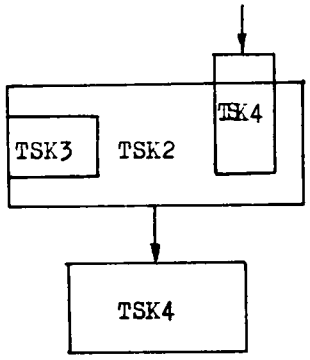


B. Overlap map

Figure 4-1. The Flowchart and Overlap Map of Scheme 1



A. Logical structure



B. Overlap map

Figure 4-2. The Flowchart and Overlap Map of Scheme 2

LIST OF REFERENCES

LIST OF REFERENCES

(1) Kevin C. O'Kane, UT-MUMPS User's Guide for Versions Running Under the Tops-10, Computer Science Department, The University of Tennessee, Knoxville, 1981.

(2) Kevin C. O'Kane, Lecture of System Programming, Computer Science Department, The University of Tennessee, Knoxville, 1981.

(3) P. B. Hansen, "A Keynote Address on Concurrent Programming," Computer, Volume 12, Number 5, May, (1979), 50-56.

(4) P. B Hansen, "The Nucleus of a Multiprogramming System," CACM, Volume 13, Number 4, April, (1970), 238-241.

(5) IBM System/360 Operating System PL/I (F) Language Reference Manual IBM, 1972.

(6) R. E. Bryant and J. B. Dennis, "Concurrent Programming," Research Direction in Software Technology, Edited by Peter Wegner, The MIT Press, (1979), 584-610.

(7) S. E. Madnik and J. J. Donovan, Operating System, McGraw-Hill Book Company, New York, 1974.

APPENDICES

APPENDIX A

EXAMPLE OF THE MUMPS MULTITASKING PROGRAM

```
 LOG 136407,245370
JOB 23 University of Tennessee TTY11
[LGNTCI Timesharing will cease in 1 hours 50 minutes]
Password:
22:05   21-Aug-82       Sat
No mail

.RUN MUMPS


UNIVERSITY OF TENNESSEE MUMPS 22:05:18 21-Aug-82


>ZG

ENTER FILE EXTENT - DEFAULT=   20
20

>ZINIT TSK1 P2 T2

>ZL

        S ^PATH1("NAME")="R ""NAME? "",^A(ID,1)"
        S ^PATH1("ADDRESS")="R ""STREET? "",^A(ID,2),""CITY? "",^A(ID,3)"
        S ^PATH1("TELEPHONE")="R ""TELEPHONE? "",^A(ID,4)"
        S ^PATH1("GRADES")="F I=1:1 I '$D(^A(ID,5,I,1)) R ""DEPT? "",^A(ID,5,I,1
),""TRM?"",^A(ID,5,I,2),""GRADE?"",^A(ID,5,I,3) QUIT"
        S ^PATH1("GPA")="FOR I=1:1:^TLE(ID) SET ^GPA(ID)=^GPA(ID)+^A(ID,5,I,3)"
        S ^PATH1("GPA2")="S ^GPA(ID)=^GPA(ID)/^TLE(ID)"
        S ^PATH1("AVG")="S ^SUM=^SUM+^GPA(ID),^AVG=^SUM/^COUNTER"
        S ^PATH("NAME")="W !,""NAME: "",?20,^A(ID,1)"
        S ^PATH("ADDRESS")="W !,""ADDRESS:"",?20,^A(ID,2),!,?20,^A(ID,3)"
        S ^PATH("TELEPHONE")="W !,""TELEPHONE:"",?20,^A(ID,4)"
        S ^PATH("GRADES")="F I=1:1 Q:'$D(^A(ID,5,I,1))  W !,^A(ID,5,I,1),?40,^A(
ID,5,I,2),^A(ID,5,I,3)"
        S ^PATH("GPA")="W !,""GPA:"",?40,^GPA(ID)"
        S ^PATH("AVG")="W !,""TOTLE STUDENTS NUMBER= "",^COUNTER,?20,""AVG="",^A
VG"
```

```
>ZINIT TSK2 P2 T99

>ZL

A       READ !,"READY?",*START W !,!
        IF START="N" GOTO A
        SET ^COUNTER=0.0
X       READ "STORE OR RETRIEVE OR ISSUE STATEMENT? ",I
        IF I="I" GOTO I
        IF I="S" READ !,"ENTER ID NBR",ID GOTO S
        IF I'="R" GOTO X
        READ !,"ENTER ID NBR",ID
        IF '$DATA(^A(ID,1)) WRITE !,"ID NOT FOUND" GOTO X
        XECUTE ^PATH("NAME")
        XECUTE ^PATH("GRADES")
        GOTO X
S       SET ^TLE(ID)=0.0
        XECUTE ^PATH1("NAME")
G       READ "GRADES? ",K IF K="Y" XECUTE ^PATH1("GRADES") SET ^TLE(ID)=^TLE(ID)
+1.0 GOTO G
        SET ^ID=ID,^FLG(ID)="FIN"
        SET ^COUNTER=^COUNTER+1.0
        IF $ZS(3,"MSG TSK2 TO TSK3: GPA CALCULATION-") GOTO X
        GOTO X
I       IF $ZS(3,"MSG TSK2 TO TSK3: END OF TSK2 !!!") GOTO E2
E2
```

```
>ZINIT TSK3 P2 T99

>ZL

        SET ID=-1
        SET ^SUM=0.0
L       SET ID=$NEXT(^FLG(ID))
        IF ID=-1 GOTO W
        ZWAIT MSG:50
        W !,"TSK3- GPA CALCULATIONS START:",!,"THIS IS THE MSG RECEIVED :$ZR=",$
ZR
        IF $C4 GOTO F
        SET ^GPA(ID)=0.0
        X ^PATH1("GPA")
        X ^PATH1("GPA2")
        X ^PATH1("AVG")
W       ZWAIT MSG:50
        GOTO L
F       W !,!,"END OF ALL TASKS !!!"
>ZINIT TSK4 P2 T99

>ZL

        ZEVT EVT1=TSK2
        ZWAIT EVT1
        W !,!,"TSK4-                ***** REPORT *****"
        SET ID=-1
R       SET ID=$NEXT(^FLG(ID))
        IF ID=-1 GOTO E
        X ^PATH("NAME"),^PATH("GRADES")
        X ^PATH("GPA")
        GOTO R
E       X ^PATH("AVG")
        IF $ZS(3,"MSG TSK4 TO TSK3: END OF TSK4") GOTO E4
E4
```

APPENDIX B

EXECUTION OF THE MUMPS MULTITASKING PROGRAM

```
>ZRUN

READY?Y

STORE OR RETRIEVE OR ISSUE STATEMENT?   S

ENTER ID NBR    123
NAME?   LEE
GRADES?         Y
DEPT?   CS
TRM?    A /
GRADE?  4
GRADES?         Y
DEPT?   MATH
TRM?    A /
GRADE?  4
GRADES?         Y
DEPT?   CHEM
TRM?    A /
GRADE?  3
GRADES?         N
STORE OR RETRIEVE OR ISSUE STATEMENT?
TSK3- GPA CALCULATIONS START:
THIS IS THE MSG RECEIVED :$ZR=MSG TSK2 TO TSK3: GPA CALCULATION-        S


ENTER ID NBR    124
NAME?   LUO
GRADES?         Y
DEPT?   CS
TRM?    A /
GRADE?  3
GRADES?         Y
DEPT?   MATH
TRM?    A /
GRADE?  4
GRADES?         N
STORE OR RETRIEVE OR ISSUE STATEMENT?
TSK3- GPA CALCULATIONS START:
THIS IS THE MSG RECEIVED :$ZR=MSG TSK2 TO TSK3: GPA CALCULATION-        I


TSK4-               ***** REPORT *****
NAME:                LEE
CS                                      A /4
MATH                                    A /4
CHEM                                    A /3
GPA:                                    3.6666667
NAME:                LUO
CS                                      A /3
MATH                                    A /4
GPA:                                    3.5
TOTLE STUDENTS NUMBER= 2  AVG=3.5833333
TSK3- GPA CALCULATIONS START:
THIS IS THE MSG RECEIVED :$ZR=MSG TSK2 TO TSK3: END OF TSK2 !!!

END OF ALL TASKS !!!
```

APPENDIX C

LIST OF Z-TYPE COMMAND AND SPECIAL FUNCTION FOR THE MUMPS MULTITASKING FACILITIES

Z-type Commands

- ZINIT Command
- Priority Command
- Time Slice Command
- Cancel Command
- Event Command
- Wait Event Command
- Wait Message Command

Special Functions of Multitasking Facilities

- $Priority
- $Time Slice
- $Completion
- $ZSend
- $ZReceive

VITA

Hsiang Y. Liao was born in Foochow, Fukien, China, on February 22, 1935. He graduated from Anglo-Chinese High School, Foochow, in September of 1952. In 1956, he received a Bachelor of Science degree in Meteorology from the Nanking University at Nanking, China.

In September of that same year, he served at Meteorological Research Institute, Peking, as an Assistant Research Meteorologist. From 1963 he taught and engaged research at the Nanking University as a Lecturer and an Associate Professor in Meteorology.

He began study to attain a Master of Science degree in Computer Science at The University of Tennessee, Knoxville in September of 1980. He was granted his Master of Science degree in Computer Science in December of 1982.