A STUDY OF PLACEMENT ALGORITHMS

IN A NON-PAGED ENVIRONMENT

by

Victor David Cruz

May 1976 TR-58

This report constituted the author's Master's thesis in Computer Sciences at The University of Texas at Austin.

DEPARTMENT OF COMPUTER SCIENCES

THE UNIVERSITY OF TEXAS AT AUSTIN

TABLE OF CONTENTS

Chapter		Page
1	INTRODUCTION	1
	1.1 Introduction	1
	1.2 Trace-Driven Simulation	3
	1.3 Memory Management	5
	1.3.1 Goals of Memory Management	6
	1.3.2 Techniques of Allocation Management	8
	1.3.2.1 Partitioned Allocation	8
	1.3.2.2 Relocatable Partitioned Allocation	13
	1.3.2.3 Paged Memory Management	15
	1.3.2.4 Segmented Memory Management	17
	1.3.2.5 Remarks	20
	1.3.3 Memory Management on the CDC 6400	20
	1.3.4 Summary of Remaining Chapters	23
2	RELATED WORKS	25
3	DESCRIPTION OF THE SYSTEM	31
	3.1 Hardware	31
	3.2 Operating System	32
	3.2.1 Peripheral Processors	32
	3.2.2 Central Memory	33
	3.2.3 MTR	35
	3.2.3.1 Communication With Other Processes	35
	3.2.3.2 Job Rollins and Rollouts	35
	3.2.3.3 Expansion Algorithm	37
	3.2.3.4 Servicing Storage Requests	39
	3.2.4 CPM	45
	3.2.5 Software Trace	47

Chapter		Page
4	THE SIMULATION MODEL	49
	4.1 Model Design.	50
	4.1.1 Operation Domain	53
	4.1.2 Scheduler and Physical Mechanism Coupling Effects	72
	4.1.3 Necessary Events	79
	4.2 Model Operation	80
	4.3 Model Validation.	84
	4.3.1 Design Consequences.	85
	4.3.2 Qualitative Description of Deviations.	89
	4.3.3 Detection of Deviations.	100
	4.3.4 Quantitative Accounting of Deviations.	103
5	EXPERIMENTAL STUDIES	107
	5.1 The Algorithms.	107
	5.2 The Performance Metrics	109
	5.3 Compaction Overhead Results	112
	5.3.1 Jobs Moved per Storage Increase.	113
	5.3.1.1 System Algorithm.	113
	5.3.1.2 First-Fit-1	116
	5.3.1.3 First-Fit-2	117
	5.3.1.4 Best-Fit-1.	117
	5.3.1.5 Best-Fit-2.	118
	5.3.1.6 Summary	118
	5.3.2 Total Field Length Moved per Storage Increase	118
	5.3.2.1 System Algorithm.	119
	5.3.2.2 First-Fit-1	119
	5.3.2.3 First-Fit-2	125
	5.3.2.4 Best-Fit-1.	125
	5.3.2.5 Best-Fit-2.	126
	5.3.2.6 Summary	126
	5.3.3 CPU Time Spent Moving Jobs	127
	5.3.4 Number of Moves per Residence.	131

Chapter Page

5.3.4.1 System Algorithm. 132
5.3.4.2 First-Fit-1 132
5.3.4.3 First-Fit-2 135
5.3.4.4 Best-Fit-1. 135
5.3.4.5 Best-Fit-2. 136
5.3.4.6 Summary 136

5.3.5 Further Studies. 136
5.3.6 Summary. 143

5.4 Memory Utilization. 144

5.4.1 Distribution of Holes. 146
5.4.2 Approximation Technique. 147
5.4.3 Results. 153

6 SUMMARY. 156

APPENDIX 159

BIBLIOGRAPHY 163

CHAPTER 1

INTRODUCTION

1.1 Introduction

Main memory, the storage that may be directly accessed by a central processor (CPU) for instructions or as data, is often the critical and limiting resource of a computer system's performance. This is particularly true of multiprogramming computer systems being used in the support of large scale interactive time-sharing applications. The workloads of these systems are typically very dynamic in nature. The broad range of job sizes together with the rapidly changing memory demands of a system of this type, combine to make the efficient operation of the memory management function of the operating system crucial.

This research is a detailed, empirical study of the memory management function of an operating system for a computer system which is primarily for small interactive jobs. Particular attention is centered on the CPU overhead for implementing the allocation mechanism of the memory manager.

The hardware of the CDC 6000 series machines requires that the memory partition assigned to a user job lies in a single contiguous block. The policy portion of memory management is essentially

the memory scheduling task. It is charged with deciding which of the competing jobs should be granted the use of central memory. The allocation mechanism must then physically accomplish the task of choosing and allocating specific portions of memory for those user jobs which policy has dictated are to be memory resident. The allocation mechanism is also charged with removing from memory those jobs which policy indicates are to be removed. It is frequently necessary, so that memory may be more fully utilized, to move a job from one location in memory to another to coalesce several blocks of vacant memory into a single larger block. The breaking up of memory into several blocks of vacant space is called "external fragmentation" (1). The process of moving jobs to coalesce free memory blocks is called compaction. The amount of compaction necessary is strongly dependent upon the allocation mechanism's choice of location for those jobs which are to be placed in memory.

The following section provides a discussion of the problems attendent with the memory management function as well as a brief survey of several approaches to solving these problems.

A non-paged memory system was chosen for this work because of the detailed data for the CDC 6000 series equipment that was available from the UT-2D operating system and also because of the lack of reported work involving empirical studies of memory management for non-paged systems.

An in-depth trace-driven simulation model of memory management is constructed. The primary metric of performance is the amount of

CPU processor time consumed in the task of performing memory compaction. Different algorithms for choosing the location in main memory to which incoming jobs are assigned are simulated and measurements made to characterize their impact on compaction overhead. An attempt is also made to determine side effects (both harmful and beneficial) which result from the choice of placement algorithms.

1.2 Trace-Driven Simulation

Trace-driven modeling has been successfully used to gain insight into a number of areas. In (2) and (3) the problems of CPU scheduling and deadlock prevention were investigated. In (4), Sherman's work stresses the technique of trace-driven modeling and its broad applicability rather than the solution to specific problems. This work is an excellent reference for the reader who is interested in a comprehensive report on trace-driven modeling techniques. Browne, Lan and Baskett (5) have used a trace-driven model to study the effects of the interaction of job scheduling with CPU scheduling. Brice (6) has also used trace-driven modeling in his studies of feedback-driven scheduling systems. Anderson (7) uses a trace-driven technique to model a complete operating system at the primitive process level.

Trace-driven modeling has two disadvantages.

1) Sample size. The trace data is recorded on a medium of finite size. The time span covered is therefore, a function of the amount of data recorded for each event and the amount of activity in

the system. The time span recorded in this study is generally about 20 minutes.

2) Representability. Since the sample is not large, the possibility exists that the sample of jobs recorded is not representative of jobs usually run on the system. The trace-driven model can only predict what would happen if the same set of jobs is submitted to the model. If the set of jobs recorded is not representative, the trace-driven model might not make a prediction valid for the normal job stream. The representability of the trace data must be justified independently.

The primary advantage of this technique is the accuracy and detail which it is capable of disclosing when used in modeling computer systems. Distribution driven simulations typically make simplifying assumptions concerning workload characteristics such as memory size requirements and the memory residence time of requests. They generally assume some idealized distribution (such as exponential) for both of these factors. It is usually also necessary to assume independence of successive requests for memory as well as independence of request size and request duration. Trace-driven simulation allows the use of real workload characteristics gathered from an environment which truly exists. Although some of the assumptions made by distribution driven simulations may be reasonable, the use of trace-driven simulations makes it unnecessary to make any assumptions concerning either the independence of successive memory

requests or the independence of request size and request duration. Since more detail can be included in the simulation model, a richer set of measures for comparison to those of the actual system is generated. This provides the opportunity for greater insight into the interrelations between processes of the operating system.

1.3 Memory Management

The need for dynamic memory management arises from the requirement that the rapidly changing memory requirements of programs and processes be mapped onto an essentially linear physical address space. Evolution of programming techniques such as list processing were among the first reasons for attacking the problem. Newell (8) described the method whereby elements of a list were kept in part of a word and the remainder of the word used to point to the successor element of the list. This allows the construction of logical linear address spaces from non-contiguous physical locations. Comfort (9) modified this earlier method by allowing for blocks of words as list elements. In cases where the elements of the list are larger than can be stored in a single word, this presents an effective reduction of pointer overhead. However, this introduces the problem of fragmentation. The physically available blocks may eventually become too small for some requests even though the total amount of vacant memory is sufficient. This is the so-called "external fragmentation" problem.

The problem of allocating memory so that fragmentation is minimal is a central goal of memory management. Knowlton (10) presents a method where memory is allocated only in blocks of set sizes. This method allows for a very efficient method of reclaiming blocks and recombining them to avoid fragmentation. However, the problem of "internal fragmentation" (1), space which is unused because of mismatches between the physical block sizes and the logical block requirements, is evident in this method. Knuth (11) presents a more thorough discussion of this method which he calls the "buddy method." All of these works were concerned primarily with the management of a program's utilization of its data storage area.

Multiprogramming of computers yields a memory management problem of a very similar nature. The operating system must manage the sharing of the central memory among all its data blocks, the user programs. This study is limited to treatment of the memory management portion of a multiprogramming operating system.

1.3.1 Goals of Memory Management

The primary task of the memory manager is not merely to execute a policy which attains a high memory utilization. Its goals are to efficiently manage memory in such a manner that the overall system's goals are met. These may be high throughput or quick response. From a slightly different point of view, this could be stated as a policy of managing memory well enough so that the

availability of memory is not the limiting factor in the performance of the computer system.

The process of managing memory requires resources. Work done managing memory is normally classified as overhead. With this in mind, the idealized goal of the memory management function of the operating system can be stated as: the management of memory in a manner such that the availability of memory is not a factor in constraining system performance, but without the usage of other resources to a degree that the consumption of these other resources for memory management becomes a factor in system performance.

The preceding paragraph implies that no management is necessary if there is an infinite amount of memory. This is essentially correct as the performance of any management policy would introduce overhead of management decisions when the availability of memory couldn't possibly become a factor in limiting system performance. In a limited memory environment, any use of idle resources for the purpose of improving memory utilization is justified. Care must be taken to avoid over-management or it can become more expensive than can be recovered from improvements in memory utilization.

Memory management in a multiprogrammed operating system is specifically concerned with three functions:

1. Status--all locations are either available or allocated. Maintaining an accurate record for each memory location is a requirement.

2. Policy--determining which user jobs should get how much memory and when. This is the function of the job scheduler.

3. Mechanism--mapping of the logical requests of the jobs onto the linear address space of physical memories.

The subfunctions of the mapping of logical job requirements onto physical memory are allocation and relocation. The allocation problem is the assignment of physical memory to the tasks or jobs. The units of assignment may be larger than the unit cell of the memory and the units assigned may or may not be in physically contiguous locations. Relocation is the movement of logical blocks from one physical address block to another physical address block.

1.3.2 Techniques of Allocation Management

Several techniques have been suggested for performing the process of managing memory so that its availability has minimal impact on system performance. All have advantages and disadvantages. The following sections describe briefly some of these techniques, all used in a multiprogramming environment.

1.3.2.1 Partitioned Allocation

In this technique, main memory is partitioned into separate regions. Each partition holds a separate job as shown in Figure 1.1. Very little specialized hardware is necessary. The major function of the manager is selecting an empty partition into which an incoming job's address space is written. When the job terminates, the partition is designated "available."

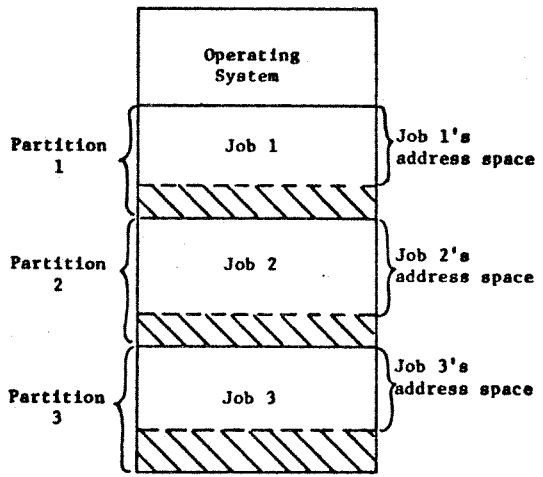


Figure 1.1 Fixed Partitioning

A desirable feature is a mechanism for preventing a job from accessing memory outside its own partition. This can be done more than one way. If the partition is allocated in a contiguous block, two bounds registers may be set to indicate the partitions bounds for the job. Whenever a job is reassigned, the partition registers must be changed.

Alternatively, a protect key may be associated with each partition. The partition is a logical concept and need not be physically allocated in a contiguous block. If the partition is divided into several blocks, the hardware must associate the several blocks. The protect key provides this mechanism by maintaining the same key for each block of the partition. When a job is allocated a partition it is informed of the appropriate protect key.

Two versions of the technique are most common. Fixed or static partitions and variable or dynamic partitions. Each has points for and against its usage.

In the static partition version, memory is divided at startup time and there are a fixed number of partitions of non-varying size (Fig. 1.1). Note again that the partitions are drawn as physically contiguous blocks for convenience and that they are only logically contiguous. A job is assigned to a vacant partition which is large enough for the maximum amount of memory which the job could need. The management of memory by this technique is simple and relatively inexpensive to implement. Its major disadvantage is that a large portion of memory may be wasted due to the unused space

in each partition in use. This is the "internal fragmentation" problem. Also, no job may be larger than the largest partitioned region.

In the dynamic partition version of partitioned allocation, partitions are created during processing time so that they match the job sizes. Thus, the size of partitions, as well as the number of partitions, vary dynamically with the requirements of the jobs currently desiring memory. This version requires more status information than the static version. It is necessary to maintain records of size, location and access restrictions of all partitions, both free and allocated. Thus, the management of this version of partitioned allocation is more complex than the static version. However, this version allows more efficient utilization of memory. Figure 1.2 shows a sequence of requests satisfied by using the dynamic partitioning technique. It seems obvious that memory is much more efficiently utilized than could have been had by executing the same sequence of requests under the static partitioning technique. Note also, that in fact, the sequence of requests might be impossible in the static version if the largest possible job was 64 units.

The dynamic version has a different problem, however. When memory becomes fragmented with alternate busy and free space, the allocation mechanism must decide which free partition to allocate. There are several policies for deciding this. Some of these will be detailed in a later chapter. At this time it is sufficient to say

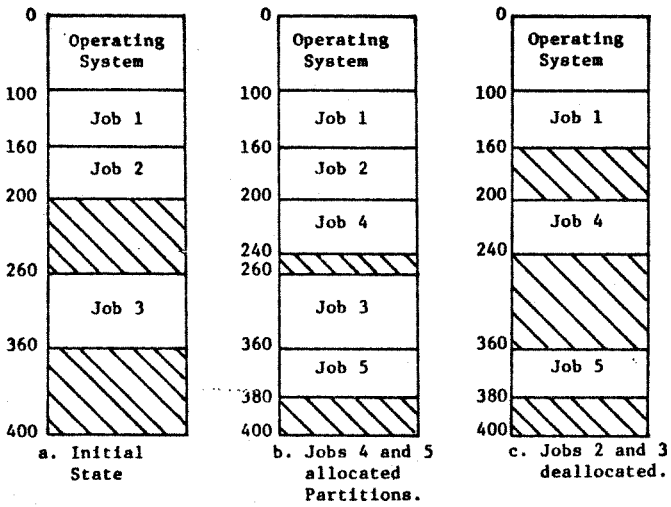


Figure 1.2 Dynamic Partitioning

that the choice of this policy has a significant impact on the performance of this technique of memory management. Also part of the fragmentation problem is the difficulty of identifying adjacent free areas and merging them. This is necessary to prevent memory fragmenting into what appears to be a multitude of small partitions. A final aspect of the fragmentation problem is the frequent case which arises of a job requiring a block of space which is larger than any available partition even though there exists sufficient space to accomodate the job in two or more available partitions (the "compaction" problem).

1.3.2.2 Relocatable Partitioned Allocation

The technique of relocatable partitioned allocation of memory is similar in many respects to the dynamic version of partitioned allocation. The relocatable partition technique provides a mechanism by which the contents of a job's partition may be relocated in memory. Dynamic memory management requires relocation if jobs are allowed to be interrupted before they run to completion. This process has the ability of combining all the free areas in memory into one contiguous area and removing the final problem of fragmentation described in the previous section. When relocation of jobs is employed to coalesce jobs, the technique is called compaction. Figure 1.3 shows an example of the usage of this technique.

The concept of moving a job's partition is simple. The act of moving a job and guaranteeing that it will still run correctly is

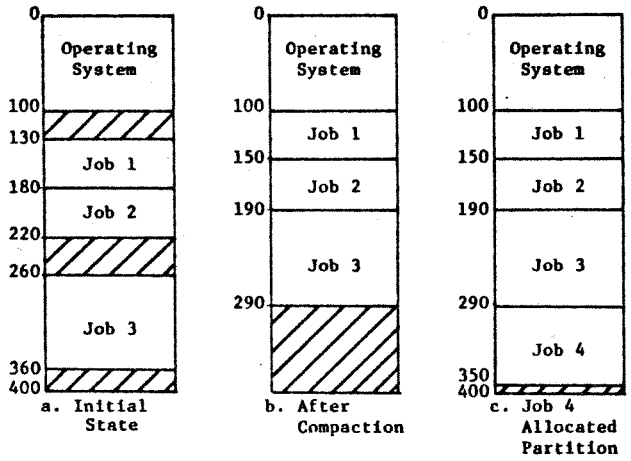


Figure 1.3 Relocatable Partitioning

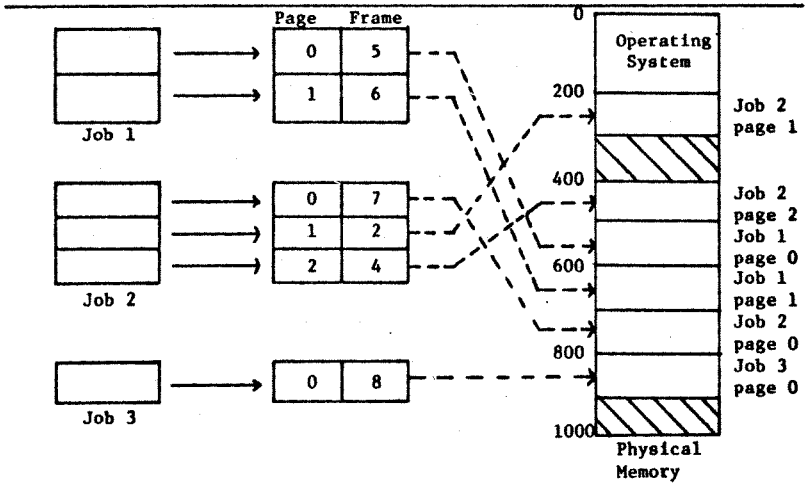


Figure 1.4 Page Mapping

more involved. To operate correctly, all items which are sensitive to location must be modified. This is not a simple operation and in general requires special hardware to perform the operation.

Memory protection is provided by means of base and bound registers indicating the upper and lower bounds of the partition. Attempts to access beyond this range are illegal and are detected by the hardware.

This technique is not without problems. Compaction time may be very substantial. This is impacted by the decision of the allocation mechanism in choosing an available partition. Also, as in the variable partition version of section 1.3.2.1 some memory will remain unused, even though it is compacted, because the amount of available space may be less than the desired partition.

1.3.2.3 Paged Memory Management

The fragmentation problem of the previous allocation technique is attributable to the requirement that a job's partition lies in a contiguous block. The paged management technique attempts to avoid the external fragmentation problem by removing the contiguity requirement.

In this allocation technique, both physical memory and the user jobs' address space are divided into pieces of equal size called pages. The physical pages of memory of the machine are called page frames. Special hardware to provide the capacity to map a job's logical address space into a physical address in memory is needed.

Any page from any user job can be placed in any page frame in memory. Thus, the maximum number of jobs in memory is not limited as in fixed partitioning. Page frames are not permanently associated with other page frames to form unchanging partitions as for fixed partitioning. It is necessary to provide fast mapping of memory references as was also necessary in the relocatable partition allocation technique. The pages remain logically contiguous from the point of view of the user job but may be scattered as necessary throughout physical memory. Figure 1.4 shows an example of this mapping.

The choice of page size is very important to this method of management. If the page size is too small, then the amount of special hardware increases since the special hardware is associated with each page regardless of page size. If the page is too long, this technique is essentially the same as the partitioned technique of section 1.3.2.1.

This technique solves the fragmentation without resorting to the physical movement of partitions. The mapping function performed in the hardware allows for non-contiguous storing of the jobs' partitions. There is however a cost/time tradeoff in the hardware mapping. Storage of the map in high-speed registers is exhorbitantly expensive. Storing the map in low-cost (relative to registers) memory is so slow as to become unacceptable. This effectively requires at least two memory fetches for every memory reference and so processing speed is cut by about 1/2. It is not intended that a thorough discussion of this tradeoff be presented in

this study. It is mentioned because it must be considered as one of the factors concerning desirability of this technique. Though this method avoids the fragmentation problem of the other techniques, the necessary page mapping hardware increases computer cost and slows down performance.

This technique does not eliminate the fragmentation problem completely. Instead, it is transformed. Dynamic partitioning suffers from lost space between partitions or "external fragmentation." Paging transforms the external fragmentation problem to one of internal fragmentation. The wasted free space is within those pages of memory which are allocated. Since page frames of memory are of fixed length, if a job's partition is not a multiple of page size, an extra page frame must be allocated for the last portion. Thus, there is internal fragmentation in those pages which are not filled.

1.3.2.4 Segmented Memory Management

The first techniques required contiguous blocks of memory for allocation to a job. The paging technique allows the job to be divided into pages of equal size while remaining logically contiguous. Segmentation is a technique for managing programs which have been divided into logical groups called segments. A segment might be a sub-routine, array or data area. Thus, each job consists of several segments.

Segmented management is similar in concept to paging, in that the segments need not be allocated contiguous to each other.

A job's segments may be placed anywhere in memory where there is a vacant area of sufficient size. Segmentation is really dynamic partitioning with the partition further broken down by logical function. Memory references are to a specific segment and to a location within that segment. It is the function of the hardware to map the references to physical addresses. Figure 1.5 shows a typical memory configuration under segmentation management.

This technique does not suffer the problem of internal fragmentation like paging. The partitions allocated are the same size as the segments. Segmentation does, however, have the earlier problem of external fragmentation. It is solved by compaction just as in the relocatable partitioned method. The fragmentation is not generally as large a problem for segmentation. Since the job's segments can be individually placed, they can usually be placed in the available spaces more successfully than can an entire job. The segments are typically smaller and thus, will fit in more places. Still, compaction is occasionally necessary.

Segmentation also has many of the same disadvantages as paging, such as higher hardware cost, more overhead for address mapping and increased management complexity. Hardware costs are high because of the necessary address mapping function and additional memory space for necessary tables.

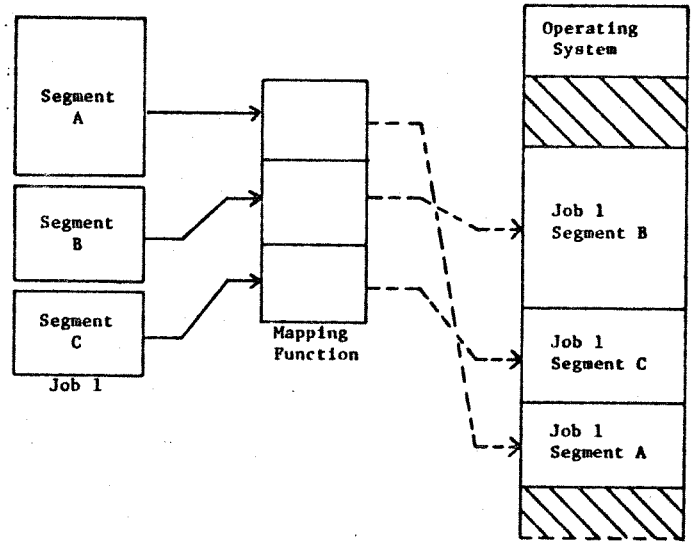


Figure 1.5 Segmented Management

1.3.2.5 Remarks

This section has presented a very brief description of several memory allocation management techniques. The primary differences have been stressed and difficulties with each method described. This is so the reader may have a broader view of the overall problem of memory management.

Each of these techniques has attempted to attain higher memory utilization. Note also that while the more flexible methods allow higher utilization, they also incur higher costs, both in hardware and in resource overhead necessary in the performance of the management function.

1.3.3 Memory Management on the CDC 6400

This work is a detailed study based on empirical data gathered from an actual, running system. The system from which all the data is gathered is on a CDC 6400 computer. Since all the data is from this one system, only the memory management technique used on the CDC 6400 is studied. The technique in use is a version of the relocatable partitioned allocation described in section 1.3.2.2. A more detailed description of this one technique is now presented.

As mentioned earlier, although the concept of moving a job's partition is simple, the requirements necessary to effect this movement are more complex. In order to remove the problems of location dependence all jobs are written such that they "think" they are located at location 0 in memory. Prior to job initiation, a

base register, called the job's RA for "reference address," is set to the address at which the job's partition begins. The contents of this relocation register are then added to each memory reference made by the job to obtain the actual physical memory address. A bounds register which contains the length of the partition is also set at the time the job begins to run. This base-bound register pair defines a contiguous partition of memory. This partition of a job will, henceforth, be referred to as a job's "field length." Since all memory references in the program are relative to the contents of the base register, relocation of the job in memory is now simplified. It can be effected by direct movement of the contents of memory from cell to cell with no translation of addresses necessary. It is then only necessary to change the contents of the base register to reflect the job's new location in memory.

When "external fragmentation" is so great that memory is too fragmented to directly allocate memory to a requesting job, compaction of memory is performed with the use of the relocation mechanism. At other times, there may be more than one partition which can be allocated to an incoming job. The choice of partitions, or holes as they will usually be referred to, can have a definite impact on the fragmentation. This in turn bears directly on the amount of compaction which is necessary. Figure 1.6 shows a sequence of job placements and the compaction necessary. Figure 1.7 shows the same sequence, but the jobs are placed in memory differently.

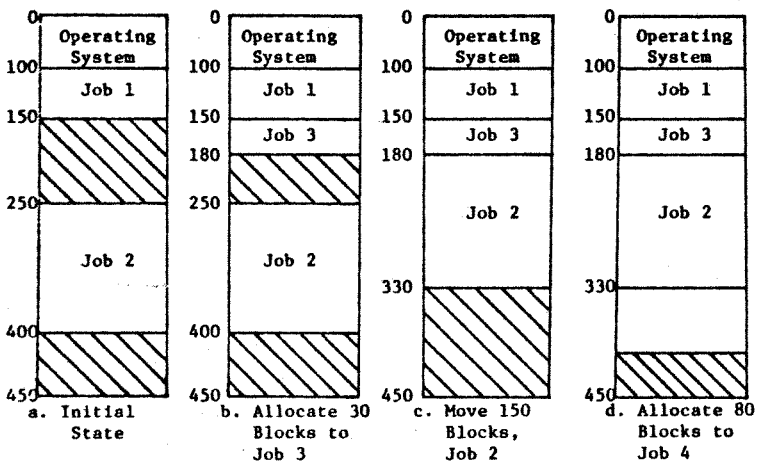


Figure 1.6

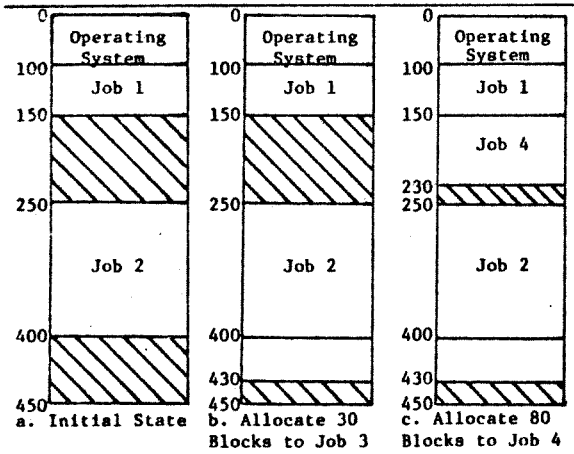


Figure 1.7

Note that the amount of compaction performed in accomplishing the same sequence is quite different. Though the memory utilization is approximately equal for these two cases, the amount of overhead is not. Memory utilization alone cannot be used as a performance metric.

In section 1.3.1, the idealized goal of memory management was stated: to manage memory in a manner such that the availability of memory is not a factor in constraining system performance, but without the usage of other resources to a degree that the consumption of these other resources for memory management becomes a factor in system performance. The example shown in Figures 1.6 and 1.7 points out one area where reducing overhead seems possible. Changing the policy for selecting the location in memory which is allocated new jobs can reduce the overhead of compaction. This is the goal of this study. Several different strategies for job placement will be simulated, using the data from the CDC 6400, to determine which yields the best results. The goal is to determine an algorithm which reduces memory compaction overhead.

1.3.4 Summary of Remaining Chapters

Chapter Two is a survey of previous works in the field of memory management which attempted studies of fragmentation.

Chapter Three is a description of the University of Texas computing facility and the UT-2D operating system.

The design, operation and validation of the simulation model are presented in Chapter Four.

In Chapter Five the placement algorithms simulated are described. The experiments performed and the results obtained are then presented.

Chapter Six presents a summary of the research.

CHAPTER 2

RELATED WORKS

Systems employing non-paged memory management are in widespread use. However, a relatively small number of published works exist which explicitly investigate placement algorithms for non-paged systems.

Collins' (12) work was one of the very earliest projects. Collins investigates the effects of different algorithms for assigning memory on fragmentation of memory. The metric used to compare the algorithms was the number of assignments made before memory was too fragmented to allow the next assignment. He reports which of the algorithms tested yielded the best results but reports no details. He also determined that allowing the size of requested blocks to exceed 10% of available memory was undesirable, as it "caused early cessation" of the sequence of allocation. This restriction would not be appropriate to most time-sharing systems. Collins does not state what assumptions were made concerning independence of successive requests nor independence of request size and duration.

The most widely referenced work is that of Knuth (11). He simulated several placement algorithms and compared results. His comparisons were based on two figures. The first comparison was

on the amount of work required to make a choice of locations. The free blocks were kept on a linked list and the number of blocks checked before making a choice was recorded and compared. The second, and more important, metric was memory utilization. Knuth did not simulate compaction, feeling it to be too expensive. Thus, he measured memory utilization as the degree to which memory filled before a request was encountered which was for a block larger than any free block. He varies several parameters including request size distribution, and request duration, reports the results of comparing the obtained measures, and offers an explanation for the results. He does not, however, include many quantitative measures concerning the actual performance. In this work, Knuth assumes that successive requests were independent and that request duration was independent of request size.

The work of Margolin, et al (13) is a trace-driven simulation (he refers to it as "emulation"). This work attempts to set up chains of blocks of storage of the same size as the "n" most commonly requested sizes. One pool of memory is set aside for allocating memory for the requests for block sizes not reserved by chains. The experiments performed were of varying the number of chains set aside and the size of the free pool. Comparisons of performance are on the basis of the amount of time spent searching chains to find a block to be allocated.

These three works were primarily concerned with allocation of blocks from a pool of available space for use as storage elements

in a list processing application. The following works are more closely related to this study because they are examining storage algorithms as directly related to time-sharing multiprogrammed computer systems.

Fenton and Payne (14) have performed simulation experiments of different placement algorithms in a non-paged multiprogramming environment. They base their comparisons on a ratio of time taken to complete a sequence of requests. The base for the ratios used is the time to complete a sequence, assuming that the free space is always in one contiguous block. This is equivalent to compacting completely after every release of an allocated block. If this compaction time is assumed to be 0, then no blocking for memory can occur as the result of fragmentation. When simulating the placement algorithms, if no block is large enought to satisfy the request, it must wait until a block is deallocated. If this block can then coalesce with already present free blocks to form a block of sufficient size, the request is satisfied. This will obviously cause the algorithms with blocking to run longer than the base case. The ratio of the time to complete the base case to the time to complete the simulated algorithm is used to compare algorithms. Request sizes were generated from an idealized approximation to the published measurements of segment sizes in (15) and (16). Request duration was assumed to be independent of request size since no work relating size and duration of requests was published.

In a more recent article, Shore (17) employs a distribution driven simulator. He assumes size and duration of request to be independent and generates request sizes from several simple theoretic distributions. The performance measure on which comparisons are made is a time-memory-product efficiency.

The most recent article concerned with the study of memory management in a non-paged environment of which the author is aware, is that of Agrawala and Bryant (18). They have constructed several simulation models. Varying request distributions and memory scheduling algorithms, Agrawala and Bryant report on a selection of metrics including memory utilization and measures of the fragmentation of memory. The distribution of requests used was uniform. They report results under both assumptions of independence and dependence of request size and duration. The effects of compaction are not included in their models.

The work of this study is trace-driven. For this reason, no assumptions concerning independence of request size and duration are necessary. Any dependencies which exist are automatically simulated, thus any errors which might have been introduced by these assumptions are eliminated.

In describing these earlier works, the performance metric used by each is stressed. No measure for judging placement algorithms has appeared as the best measure to use.

All of the previous works have ignored the possibility of actually using compaction as part of the fragmentation solution. The general consensus seems to have been that compaction would be too slow and much too expensive to even consider as a realistic possibility. It has simply been assumed that compaction would deteriorate performance and consume resources to too great a degree to be practicable.

The UT-2D operating system uses compaction effectively as a part of its memory management function. This high level of system performance (see Chapter 4) with compaction being periodically performed, shows that compaction is a viable operation in the process of memory management. Still, compaction does consume resources and the amount of compaction is heavily impacted upon by the amount of fragmentation. In UT-2D, compaction is performed by the CPU. If the placement algorithm is inferior, then the CPU overhead in the performance of compaction will increase. Therefore, the primary metric by which placement algorithms are judged in this work, is the amount of CPU processing time consumed in the performance of compaction. If this can be reduced, then the use of compaction becomes an even more acceptable tool for use in solving the problem of memory management in non-paged environments.

As a secondary result of this study, memory utilization is investigated. The details necessary to produce quantitative results of the same level of detail as in the study of compaction are not

included. However, results concerning changes in the job scheduler are obtained. These results are used to generate an estimate of the upper limit on the obtainable memory utilization.

CHAPTER 3

DESCRIPTION OF THE SYSTEM

The trace-driven simulation in this study is of a CDC 6400 running under the UT-2D operating system. Before discussing the simulation, it is necessary to describe the system. This chapter is a brief description of the system, detailing only those portions of the system necessary for a clear understanding of the simulation model and the features of the system which were simulated. A more complete description of the system hardware can be found in Thornton (19). A quite detailed description of UT-2D, its structure and its workings can be found in Wedel (20) and in Howard (21). The software event recorder is described by Howard and Wedel (22). The tape of events produced by the recorder is used to drive the simulator in its modeling of the system.

3.1 Hardware

The system described is the CDC 6600-6400 dual mainframe system of the University of Texas at Austin. Each mainframe runs more or less autonomously. The 6600 has ten peripheral processors (PPs), 131,072 words of central memory (CM), and 12 I/O channels. The 6400 has 7 PPs, 65,536 words of CM and 11 I/O channels. The

machines share access to 503,808 words of extended core storage (ECS), 4 CDC 808 disks, an 8-spindle CDC 841 disk system, a single tape controller, various remote job entry terminals and unit record equipment. All processors have access to central memory but only the CPU has access to ECS. All data used in the study was obtained from the 6400 subsystem.

3.2 Operating System

The operating system, UT-2D, is a locally written, general purpose multiprogramming system. It provides both batch and interactive service with most of the batch work being done on the CDC 6600. The CDC 6400 handles the majority of the interactive workload. The interactive workload is more dynamic in nature, performing more storage moves. For this reason, it was chosen over the batch system for study.

3.2.1 Peripheral Processors

The CDC 6400 has seven peripheral processors, three of which are dedicated to the system. One is used to support the operator's console. A second is dedicated to the system monitor, MTR, which maintains control of the overall system. The third PP is assigned to the multiplexor driver. The remaining four PPs are the pool PPs. They are allocated on a first come first serve (FCFS) basis, as needed for the performance of system functions, most notably I/O and the allocation and deallocation of memory. For the purpose of this

study, the use of the PPs in memory management is most important. Further discussion of the PPs' role in memory management is given in section 3.2.3.

3.2.2 Central Memory

Central Memory is partitioned into two areas. The low-order end of memory is reserved for the central memory resident, CMR. CMR consists of Central Processor Monitor, CPM, and various system tables and communications buffers. The second area is the portion available to the 16 virtual processors called Control Points (CPs). All use of the central memory and the central processor, other than by CMR, is by a CP. Each control point dynamically varies its partition size as well as its location in memory. The allocation of that portion of CM for which the CPs are competing is the item in which this study is most interested.

Of the sixteen control points, three are dedicated to the system. One is occupied by the interactive terminal manager, SATURN. SATURN is permanently located immediately above the low-order end of memory. The other two system CPs are occupied by PISCES and GEMINI. PISCES is the real time accounting process and GEMINI contains the system buffer pool. These two CPs will occasionally vary in size and location. However, due to the algorithm used for relocation of CPs, they will always remain above any CPs which are active for user jobs. Figure 3.1 shows a typical memory

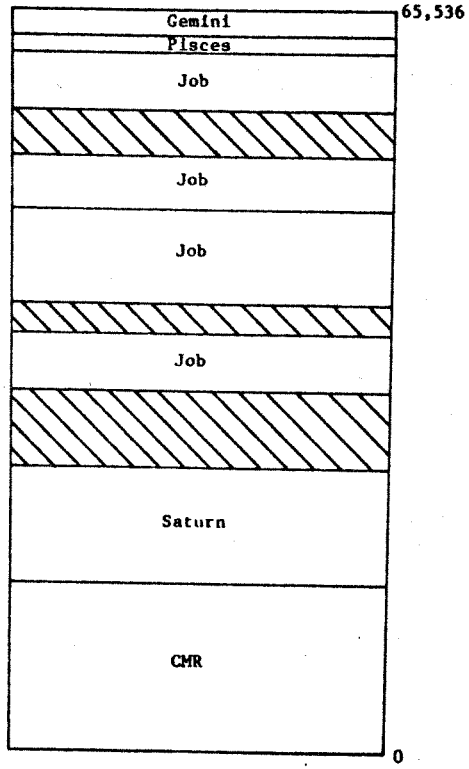


Figure 3.1

configuration. The algorithm for dynamic relocation of control points is given in section 3.2.3.3.

3.2.3 MTR

3.2.3.1 Communication With Other Processes

The system monitor, MTR, handles all requests for system resources. Communication with MTR is through a portion of the low-order end of memory. This block of communication cells is used like mail boxes. The mail boxes are polled in a regular, cyclic manner by MTR, which responds appropriately for all messages found. It is in these slots that the job scheduler files requests to roll jobs in and out of memory.

MTR also polls the communication cells of the CPs. These cells, at relative location 1 for each CP, are called RA+1 cells (Reference Address + 1). A control point requests service of MTR by posting an RA+1 call. MTR, which runs in a PP, also polls the RA+1 cells of the CPs. If a CP wished to request a storage increase ("RST up"), it would post the appropriate message in its RA+1 cell. MTR would then note the request and schedule it for service by a PP.

3.2.3.2 Job Rollins and Rollouts

When a request to rollin a job is found by MTR, a resume job (1RJ) is initiated at a PP. During the rollin procedure the PP will issue to MTR an "RST up". The first step in processing of an "RST up" is to insure that sufficient space is available to grant the request.

Although the CP will require the space in a single contiguous block, the check at this point is only if sufficient free space exists in all of CM. Whether it is in one block or more is not yet of concern. If sufficient space is not available, MTR will inform the requesting PP of the situation. If there is sufficient space then MTR will designate the hole immediately above SATURN as the specific hole into which the CP is to be placed. It is quite possible that this hole will not be large enough to satisfy the storage request. In fact, occasionally the hole will be of size 0. At this point, the CP being rolled in is considered to be of length 0 and to start at the top of the hole above SATURN. Since the CP now has both a location and a length, it could be processed just as any CP in memory which has requested a storage increase. This is what the system does. MTR treats an "RST up" for a job rolling in, as no different from an "RST up" from a CM resident job, once the location of the job being rolled in is established.

MTR rolls a job out by initiating a suspend job (1SJ) process at a PP. The PP will issue an "RST down" to zero for the control point rolling out. The job's field length is written out to ECS (the swapping medium), memory is released and the job is detached from the control point. After the rollout is completed, the CP is either attached to a new job or remains idle. If it is attached to a new job, then the rollin process is initiated. If the CP remains idle, it does not contend for memory and presents no interference to any other CPs in memory.

3.2.3.3 Expansion Algorithm

This section describes the algorithm used by the UT-2D system to determine the movement of CPs necessary to accomplish an "RST up". The purpose of executing this algorithm is the creation of a gap, adjacent to the requesting control point, large enough to satisfy the new storage requirement. Figure 3.2 is a block diagram of the algorithm used to pick the next CP to be moved. The following paragraphs explain certain aspects of this algorithm which need particular attention.

If the designated move is into a hole above the control points, then the change in location which is determined is such that the CP will be moved all the way to the top of the hole. If the move is into a hole below the control point, then the determined move will be only enough so that the sum of all gaps above is just sufficient to satisfy the request. In either case, the amount by which the CP is to be moved is calculated at the time the algorithm is run. This figure is not recalculated at the time the actual move takes place. This method of determining new job locations, used in conjunction with the policy of placing new jobs in memory below all other user jobs, has the effect of packing CPs toward the upper end of memory. This tends to produce holes in the lower end. It also moves the jobs which are resident in memory the longest, to the top, where, hopefully, they will not need to be moved as often.

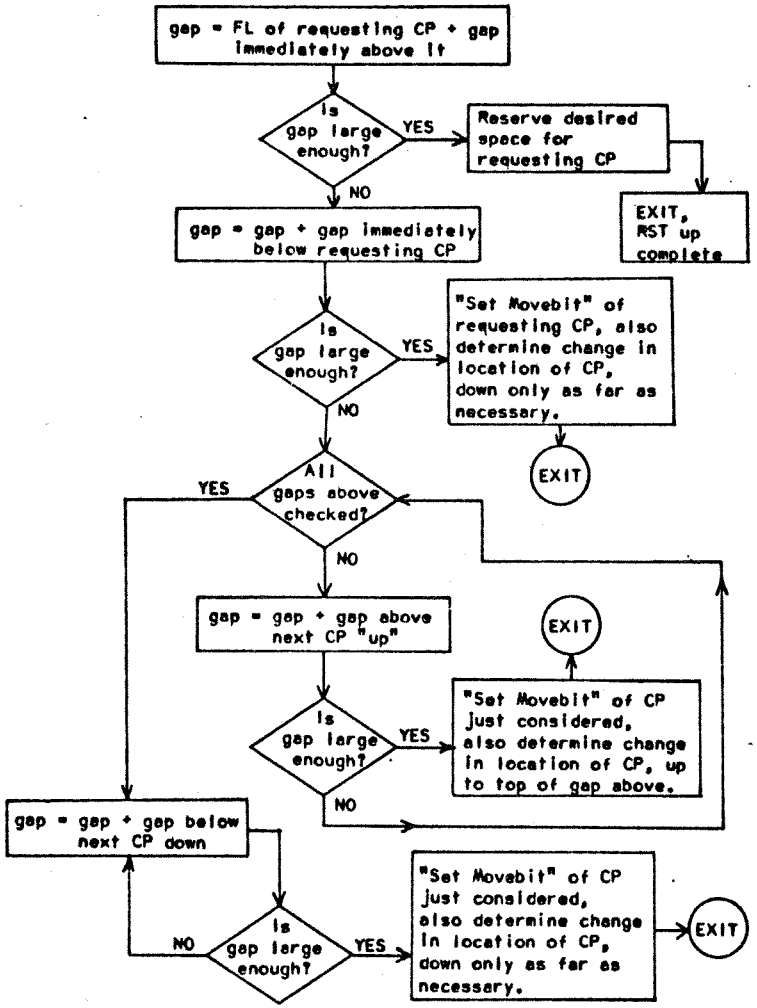


Figure 3.2 Expansion Algorithm

The algorithm is run once, to determine the "next" CP which needs to be moved. It may return the first time designating the requesting CP as the next one. If another CP is marked, then after that CP has successfully been moved, the algorithm will be restarted. The algorithm is not biased by the last decision.

This algorithm does not actually move the CPs, but rather marks the CP which has been chosen. A CP which has been chosen as the "next" CP to be moved is said to "have its movebit set." After the move is complete, the "movebit" is cleared.

A control point may have the CPU and/or one or more PPs active. No storage moves can take place on the field length of a control point which has a processor active. The actual move will take place whenever the CP is idle, i.e., has no PPs (or the CPU) active. PPs which have a long expected duration, periodically pause to allow MTR to initiate a storage move. The mechanism for the pause is a monitor function call, Pause for Relocation (PFR). PFR is a request to the system monitor to delay the PP if a storage move is scheduled for the CP to which the PP is attached.

3.2.3.4 Servicing Storage Requests

As part of its function of system resource allocation, MTR has explicit control over the scheduling of requests for PP service. This is basically a FCFS scheduler. There are two exceptions to this rule which move to the head of the queue. The first of these is the process which writes to tape the events recorded by the software

monitor. The second and more important exception is the process requested by the job scheduler to roll a job out of memory. This is done to free memory as quickly as possible since it is highly probable that there is a ready job which can make use of this memory.

Obviously, one part of PP scheduling is the scheduling and servicing of storage requests. The RST functions may be either increases or decreases. MTR will only process one storage increase at a time. If another "RST up" is requested while MTR is already busy processing an earlier "RST up," the new "RST up" will be postponed for later processing. However, any number of storage decreases may proceed in parallel with the processing of an "RST up".

The servicing of an "RST down" is a relatively simple procedure. As soon as all PPs for a CP are idle, that is, all peripheral processes must have either terminated or paused for relocation by issuing a PFR function, the "RST down" can be processed. Since accomplishing this could not possibly require the movement of any CP, it is not necessary to execute the compaction algorithm of the previous section. There is one case when the processing of an "RST down" is postponed. Any control point which requests a decrease in storage while it has its "movebit" set, will be postponed until after the move is completed.

Porcessing an "RST up" is a much more complicated process. As indicated earlier, MTR must first verify that sufficient space is available to allow the requesting CP to expand to the requested new

size. Should the space be unavailable, a reply indicating this is made to the calling PP. Assuming the space is available, MTR will begin the process of creating the necessary hole in memory to satisfy the new storage requirement. This is done by executing the algorithm of the previous section to mark the CP which needs to be moved.

MTR selects only one control point at a time to be moved, rather than an entire sequence of CPs which need to be moved. If no CP changes its storage requirements between the first running of the algorithm and the completion of the "RST up," then the moves which are determined are the same as if the entire sequence were determined originally. This sequence will be referred to as the "original sequence."

A control point must be idle before it can be moved. There is usually a waiting time between the marking of a CP to be moved and the actual movement of the job. This is to wait for the PPs to either terminate or pause. During this time there is said to be a "move pending." It is possible, since MTR allows the processing of storage decreases in parallel with storage increases, for another CP to either reduce its field length or rollout while there is a "move pending." This reduction of storage requirements by control points other than the CP marked to be moved, while there is a "move pending," can have the following effects.

An "RST down" may produce a hole which makes it unnecessary to move any other CPs. This produces the effect of cutting off the remainder of the "original sequence" as no longer necessary. It may

actually render the "move pending" unnecessary, but once the "movebit" is set, the system must perform the move. Figure 3.3 shows an example of this.

The "RST down" could produce a hole which will change the "original sequence." An analysis of the expansion algorithm will show that if CPs which are both above and below the requesting CP will need to be moved, then the lowest of these will be moved first. If, while waiting for this move, a control point above the requesting CP should rollout, then the "original sequence" will be changed. Remembering that future decisions of the expansion algorithm are unbiased by past decisions, the planned movement of any other control points below the requesting CP may be discarded. It is also possible that the movement of some CPs above the requesting CP will be cancelled. Figure 3.4 shows an example of this case.

The expansion algorithm performs two functions. It selects the next CP to be moved and it predetermines the new location for the selected CP. Predetermining the new location occasionally results in actions which do not satisfy the intent of the expansion algorithm. The control point which is waiting to be moved is said to be "idling down" and will be called the idling CP. It is possible that the CP immediately above the idling CP is rolled out before the "move pending" is completed. When the idling CP is finally moved to its predetermined location, the intent of the expansion algorithm will not be satisfied. The intent is that the move will place the

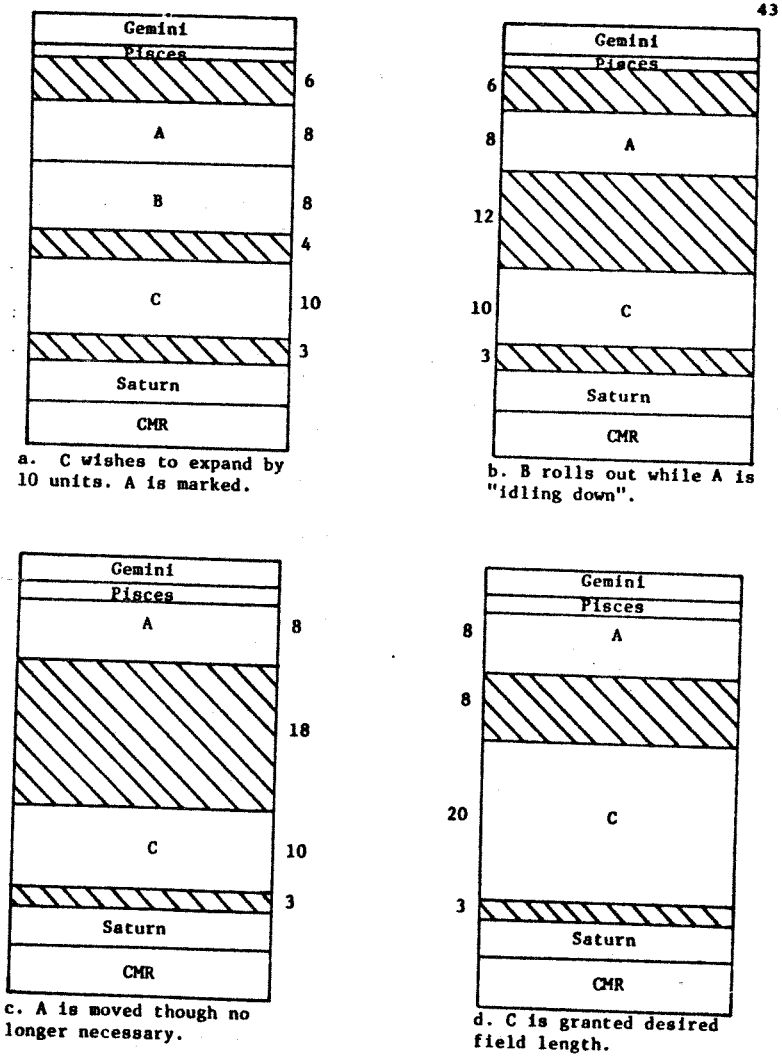
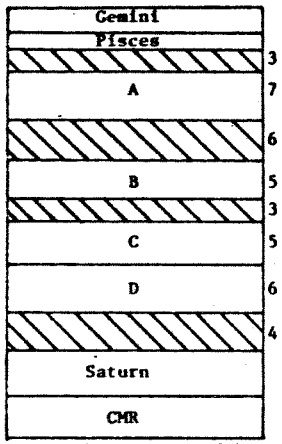
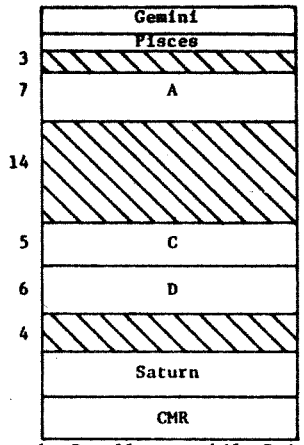


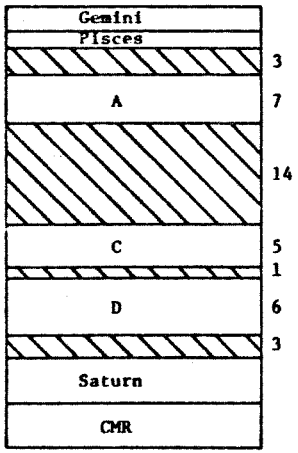
Figure 3.3



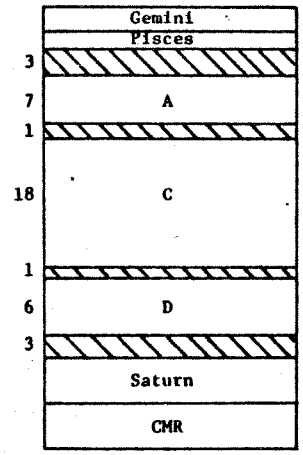
a. C wishes to expand by 13 units. D is marked.



b. B rolls out while D is "idling down".



c. D moves though no longer necessary.



d. C increases. A and C didn't move.

Figure 3.4

CP at the top of the hole. The job will instead leave a hole where the rolled out job was because the move is preset. An example of this sequence is shown in Figure 3.5.

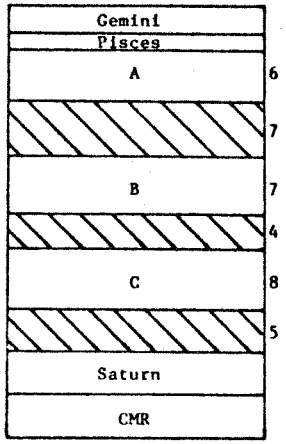
3.2.4 CPM

The central processor monitor, CPM, can be viewed as a dumb but strong branch of MTR. CPM is initiated solely by MTR. The program directories, file tables, equipment tables and some PP program files are located in CM. CPM contains subprocesses to handle those functions which reference or manipulate these areas of memory.

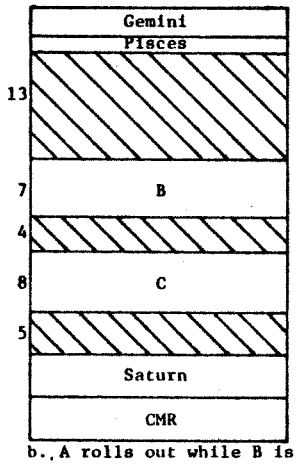
In the previous section, reference is made frequently to the movement of a control point. The actual relocation process is located in CPM. This is primarily because the CPU can perform core to core transfers much faster than a PP. Thus, the actual sequence of events to move a CP is:

1) MTR determines that the CP to be moved is idle;
2) MTR exchanges the CPU to the relocation process of CPM;
3) CPM performs the storage move;
4) MTR exchanges the CPU back to the control point which was interrupted.

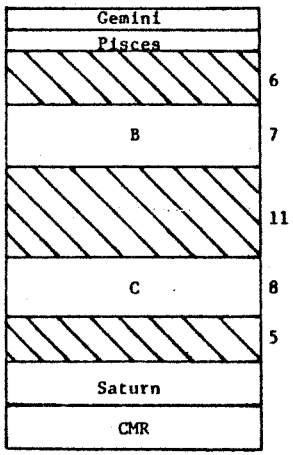
The job scheduler is also a part of CPM. It runs at the request of MTR and selects the jobs which should be resident in the user CPs. Scheduler decisions are implemented by the creation of resume job (1RJ) and suspend job (1SJ) peripheral processes.



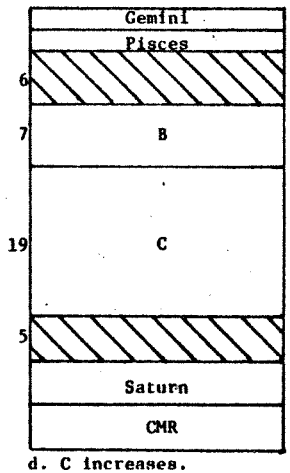
a. C wishes to expand by 11 units. B is marked.



b. A rolls out while B is "idling down".



c. B is moved but leaves hole above it.



d. C increases.

Figure 3.5

3.2.5 Software Trace

The software event trace recorder is resident in MTR. MTR has a large circular buffer in which events are recorded. The events which are recorded are characterized in Table 3.1. When the buffer has at least one physical tape record (512 words) full of recorded events, MTR issues a request for a peripheral process, 1DB. This request will be serviced as soon as there is an available PP. When the PP is assigned, 1DB will copy blocks of 512 words to tape until there is less than a tape record of events recorded in the buffer. As mentioned in an earlier section, MTR promotes the request for 1DB to the head of the PP queue. Even so, if the system is highly active, the buffer may fill before a PP becomes available. This results in "lost events." These "lost events" can perturb the results significantly. They also make it quite difficult to correctly track the location and memory requirements of the control points. The event tapes used in this study were specially chosen for their porperties of high activity and few lost events.

TABLE 3.1

1. Allocation of the CPU among CPs and CPM.
2. Messages and requests from a CP to MTR.
3. Movement of requests for PP service through the PP queue.
4. Assignment of PPs.
5. Requests and releases by PPs of other resources and services.
6. Job scheduler decisions

CHAPTER 4

THE SIMULATION MODEL

The topic of this study is the overhead of physical memory management on a computer which requires the entire field length of a job to be resident in central memory and which does not support paging. Attention was focused on this aspect of system operation by the fact that from 6% to 8% of the CPU resource, of the mainframe executing the interactive workload, was required for storage compaction. The design of a model to investigate the effects of different physical storage management procedures requires the determination of the interaction of the physical storage management with the other system procedures and algorithms. The goal is to have the simplest model capable of returning a quantitatively accurate assessment of the physical storage management functions.

The model is a trace-driven model. The determination to be made is what system functions must be explicitly included in the model. This determination is made by an analysis of coupling and feedback between the physical management process and the most closely related process, the scheduler of jobs to central memory.

4.1 Model Design

The wealth of information provided by the UT-2D software monitor is sufficient to provide a simulation of extreme detail of nearly every aspect of the operating system. This study, however, is only investigating a rather specific function of the operating system. It therefore, seemed unlikely that all of the available detail was required. The optimal system model will return performance measures to the level of accuracy to which they are resolved by system measurement and will retain sufficient representation of the total system environment so that changes in the component of interest interact validly with the system environment.

The function of the UT-2D operating system which is simulated in this study is memory management. Memory management is logically divisible into two separate sub-tasks: priority scheduling of jobs for allocation of memory (policy management) and physical management of the jobs in memory. Unfortunately, although these tasks are separate, they are not independent. Policy scheduling is a logically complex task.

This work is interested in the physical or mechanism level of memory management, including:

1) attachment of central memory to newly arriving jobs (rollins),
2) detachment of central memory for departing jobs (rollouts), and

3) changes in the central memory requirements of currently resident jobs.

Figure 4.1 depicts the relationship of the sub-tasks, including the coupling or feedback effect between the policy and mechanism level of memory management.

One would like to treat the physical mechanism level of memory management independently of the policy scheduling task. To do this, it is necessary that the following two points are both satisfied. First, it must be shown that the domain in which the physical manager is currently operating and will be operating is a domain which can be classed as representative of reasonable priority job schedulers in meeting reasonable goals. Second, the coupling effect between the two tasks must be weak enough or of small enough magnitude, such that, changing the mechanism algorithm would not perturb the operation of the policy scheduling task to a significant degree, and vice versa. Once these two points are shown, then the simulation of the memory management function can be performed as portrayed in Figure 4.2. Only the physical mechanism need be explicitly modeled. The policy scheduling task will be implicitly performed by accepting the decisions made by the system as recorded on the event trace tape. This makes it possible to merely regard the set of requests for memory and the associated residence times as a sequence to drive the simulation of the physical mechanism.

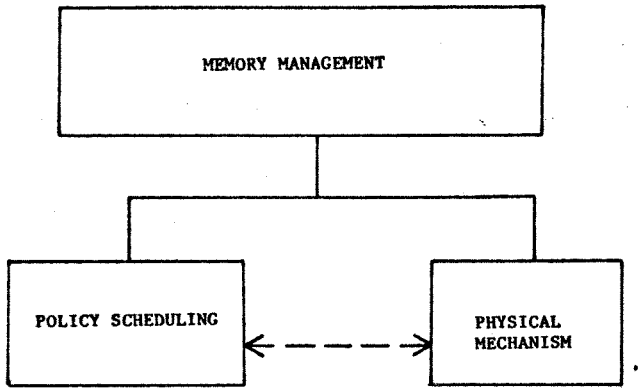


Figure 4.1

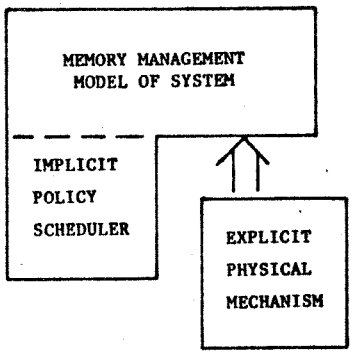


Figure 4.2

4.1.1 Operation Domain

This section presents a description of the domain of operation which the policy scheduler provides to the physical mechanism. Included are statistics by which the job load can be characterized and statistics concerning the efficiency at which the entire system is operating.

The policy scheduler is the predominant factor in determining the level of operation of the system. The decision was made that it is sufficient to accept the decisions of the policy scheduler as they are recorded on the trace. This avoids the necessity of explicitly modeling the job scheduler. Although this will reduce the detail to which some measures of memory management can be quantitatively predicted, it produces a simpler model which does return the means for a quantitatively accurate assessment of the metric of primary concern, CPU utilization in the performance of storage moves. The analysis made in determining that the explicit modeling of the policy scheduler could be omitted, is also presented in this section.

All data used in this study was obtained from measurements recorded on three tapes. The measurements were taken at mid-afternoon for periods of 14, 28 and 18 minutes respectively. Past experience in other studies, has shown that during the mid-afternoon the system is moderately to heavily utilized and that this period places greatest demands for efficient physical memory management.

Since the system being modeled is for general purpose computing in a university environment, the range of field lengths

for jobs ranges over the full spectrum allowed by the operating system, from a minimum of 0.625 K to a maximum of 32 K (K=1024). Memory is allocated in blocks of 64 words (100_8). Most computation involves file handling routines, which are relatively short and jobs employing the text editor. These jobs are generally in the range of about 4 to 8 K. Also well used is the FORTRAN compiler which occupies about 16 K. Further usage is provided by an assortment of jobs executing programs in any of a number of languages including ALGOL, LISP and PASCAL to name just a few. In general, at any point in time there is a large mixture of jobs present. Figure 4.3 gives a time weighted distribution of the field lengths of ready jobs. The job scheduler, which will be described shortly, favors shorter jobs; thus, simply recording the memory requirements of the jobs which run will not give a true picture of the distribution of jobs which wish to run. What was done instead, was to determine how much time was spent waiting for memory by jobs of each field length. When normalized, so that the sum of all waiting times is one, what results is the probability density function for field lengths of ready jobs. This is the distribution given in Figure 4.3. Each point represents the probability that there is a job of that field length which is ready to rollin, at any point in time. It is an approximation of the distribution of the field lengths of jobs from which the job scheduler must choose.

Figures 4.4, 4.5 and 4.6 show the distribution of residence times for each of the three tapes. As can be seen, the shape of the

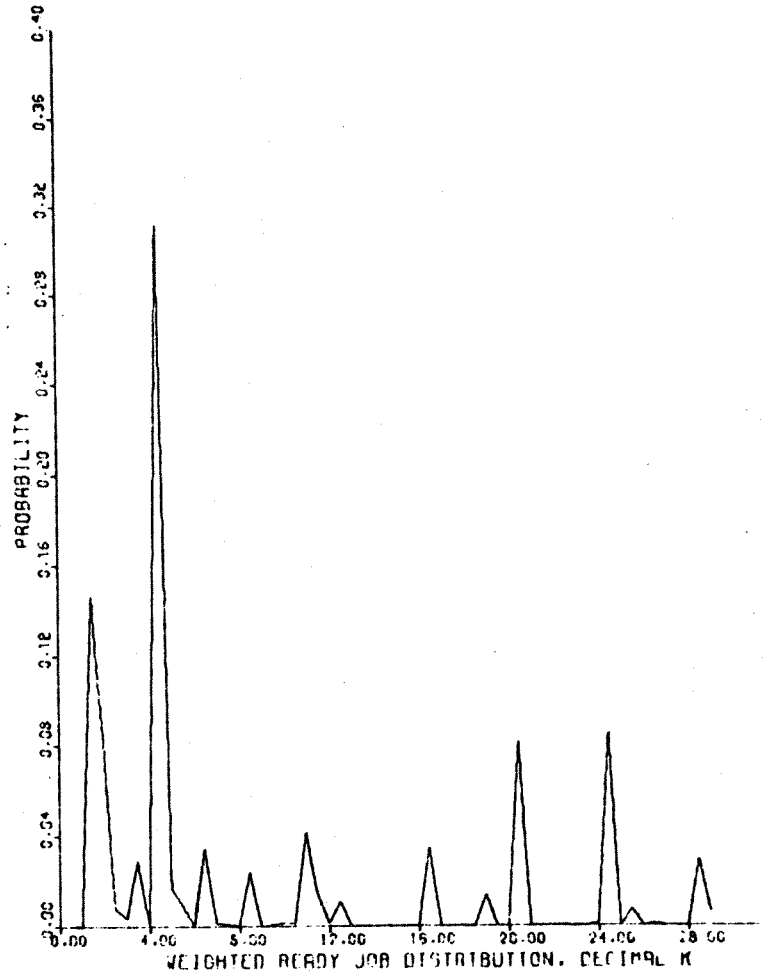


Figure 4.3A (Tape 1)

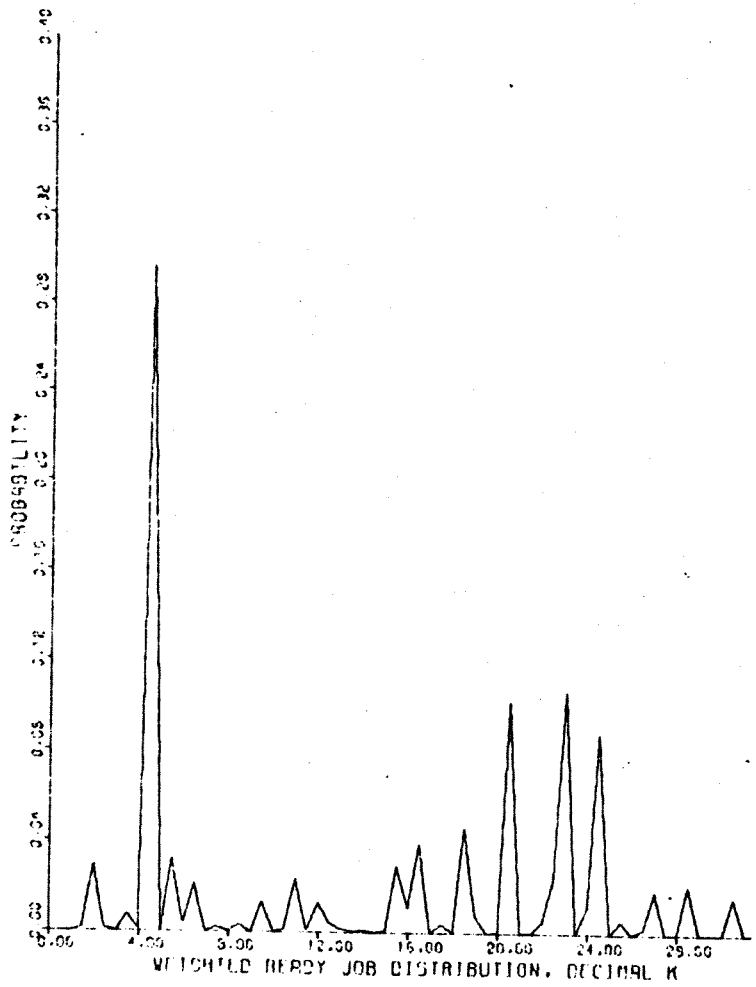


Figure 4.3B (Tape 2)

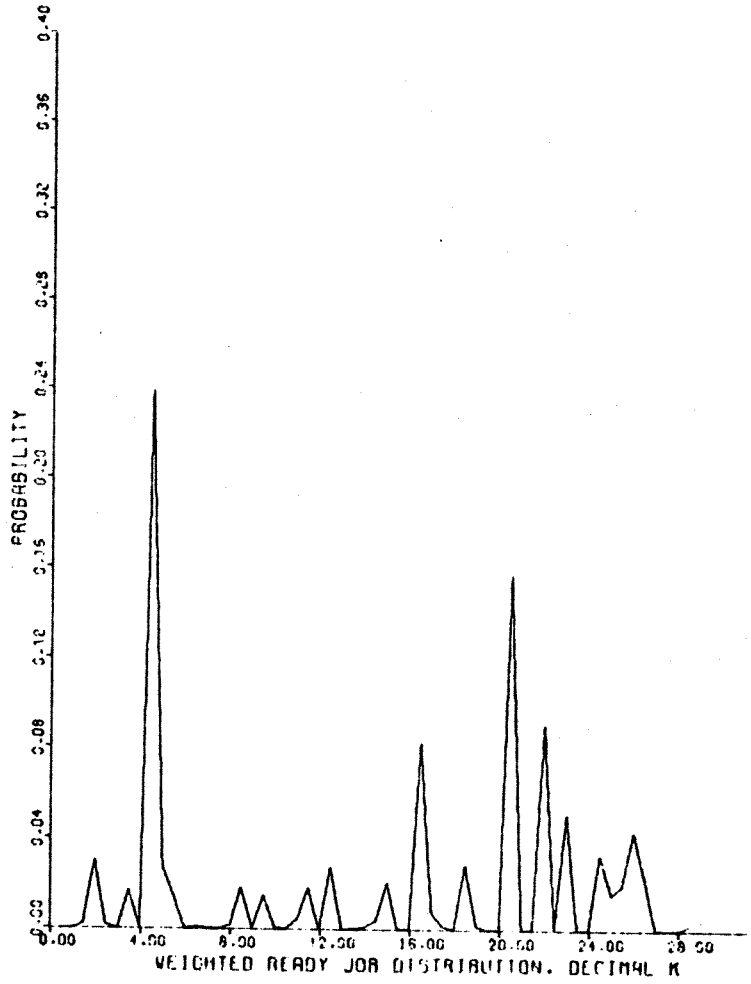


Figure 4.3C (Tape 3)

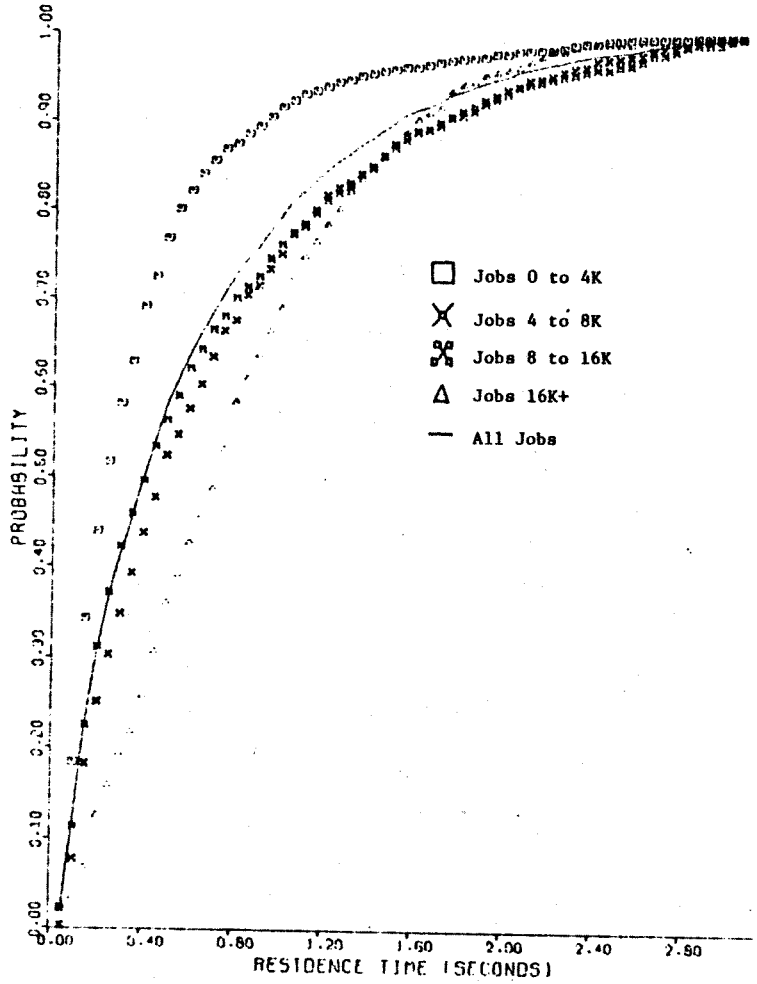


Figure 4.4 (Tape 1)

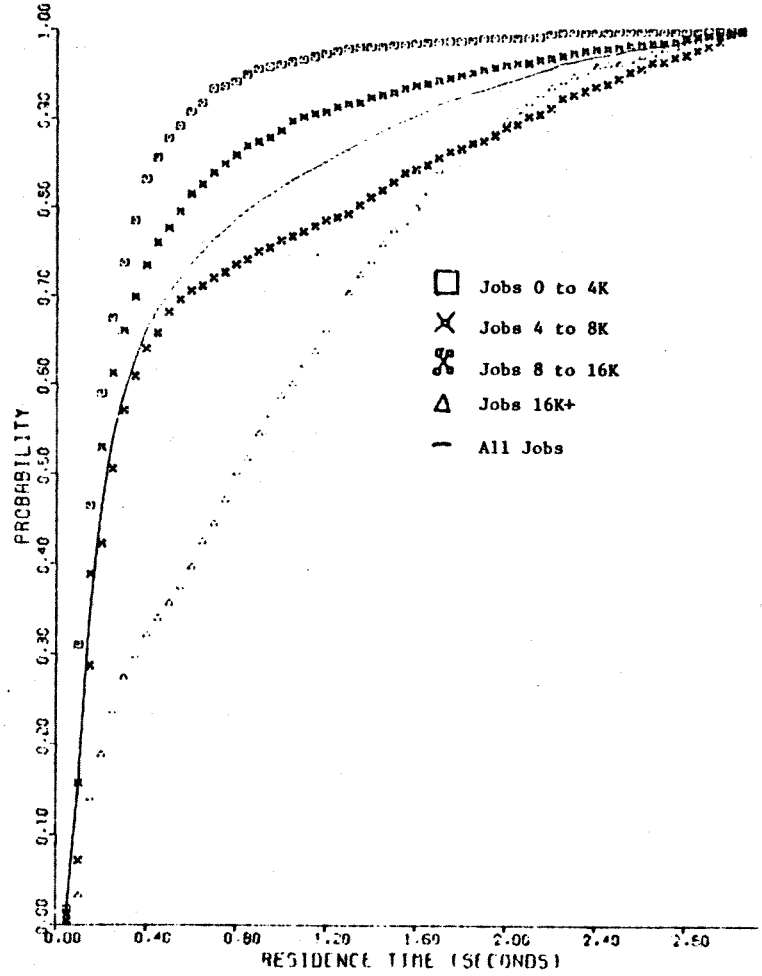


Figure 4.5 (Tape 2)

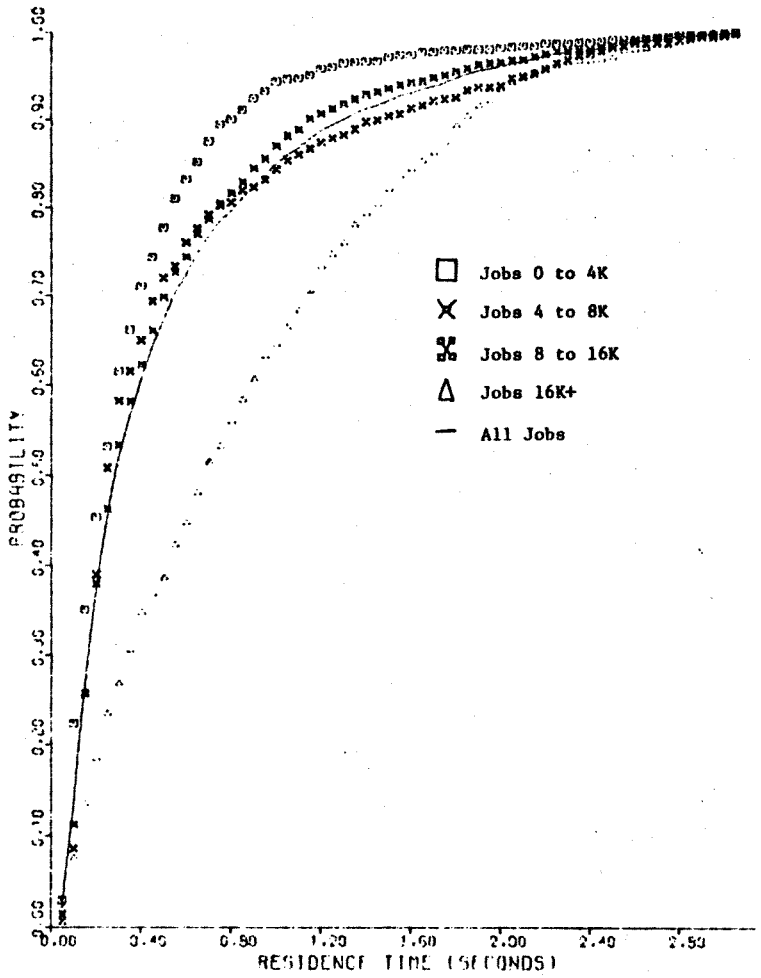


Figure 4.6 (Tape 3)

curves is similar for all field lengths but the median increases as the job size increases. This shows that larger jobs tend to remain in memory longer. This is generally attributable to two factors.

1) The smallest jobs are generally small file handling routines. They typically require only a small burst and then are ready to leave memory. These jobs vacate memory before the job scheduler finds any necessity to preempt them. Rather than being ejected from memory, they more typically must notify the system monitor that they have no more need of memory.

2) The larger jobs are the other extreme. These are jobs such as compilations or production programs which may require several seconds of time to complete. They typically, will remain in memory just as long as the scheduler will allow.

The majority of the jobs have a relatively small memory requirement and are resident in memory on the order of 150 milliseconds. Fewer than 2% of the jobs are over 16 K and remain in memory for over 2 seconds.

The next step in describing the operation field for the physical mechanism is to describe the policy sceduler and its effects. The policy scheduler decides which jobs are to be allocated memory. It, therefore, determines the level of operation for the entire system. If the policy scheduler makes poor decisions, the system may be underloaded even though there are many jobs ready for memory. It is, therefore, crucial that the job scheduler maintain a policy which keeps the remainder of the system busy, as well as selecting

jobs in a manner consistent with achieving goals set forth for a good job scheduler. These goals are: short response time for the majority of users and efficient management of resources, particularly memory and the central processor.

The UT-2D priority scheduler which selects those jobs which should be allocated main memory is a least cost first scheduler. Cost is defined to be a product of time and field length. The time used is TM time, a locally defined quantity. It is a function of CP time used and the number of I/O records accessed. The workload being processed is the interactive workload. The scheduler does not use total accrued TM time in determining cost. For jobs in memory, the time used is "TM time accrued during this memory residence." Thus, a job which has just been rolled in will have priority over a job of the same size which has run for some time. For jobs not in memory, the time used is "TM time since last terminal wait plus 1 second." Thus, a job which has just terminated a "think period" will have priority over a job of the same size which has run for some time, but was preempted and is now waiting for memory. This scheme for determining the time factor of a job's cost will tend to avoid the thrashing of swapping jobs in and out. It will tend to guarantee that a job which is rolled in will remain for a period of time. The scheduler works by maintaining the highest priority jobs in memory, reordering jobs periodically and preempting jobs as their priority drops and other jobs of higher priority arrive.

Since the scheduler favors shorter jobs, the distribution of the jobs which actually run is somewhat different from the distribution of ready jobs as seen in Figure 4.7. The spike at 4 K is still evident, but the relative number of smaller jobs is noticeably increased. The effect of this on the physical mechanism is merely to provide a different job mix to be allocated memory. Since all that is being shown is that the operating domain of the physical mechanism is realistic and can be used as representative, small perturbations in the job field length distribution present no difficulties for the modeling study.

When the scheduler is run, it causes several PP requests to be created for the running of 1RJ and 1SJ processes, which are the primary generators of storage requests. The remainder of storage requests are generated by RFL (Request Field Length) functions for jobs already resident and desiring to expand. One measure of system activity is the frequency with which the job scheduler is run. Table 4.1 shows that the scheduler is run about six times per second. It is run "as often as necessary," thus its decisions are essentially continuous. However, it does not run infinitely fast. Thus there should be slightly more than this number of RSTs per second. In fact, measurements indicate in excess of three rollins and three rollouts per second. When jobs which change field lengths while resident are also considered, this results in storage changes being satisfied at a rate of about seven per second, as indicated in Table 4.2.

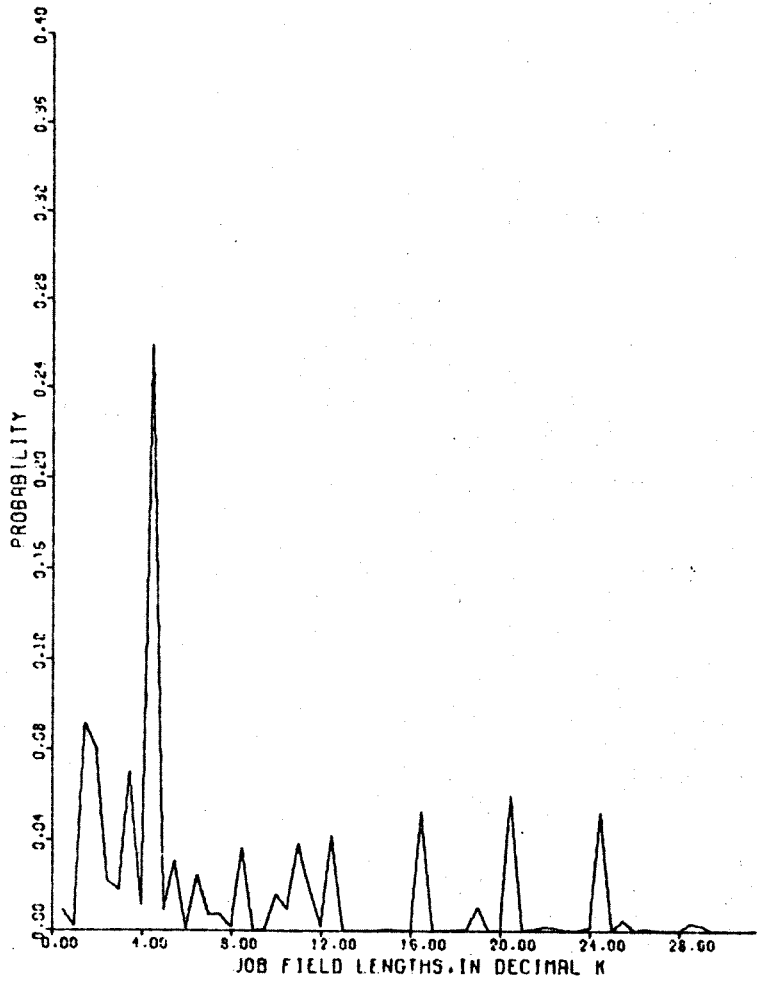


Figure 4.7A (Tape 1)

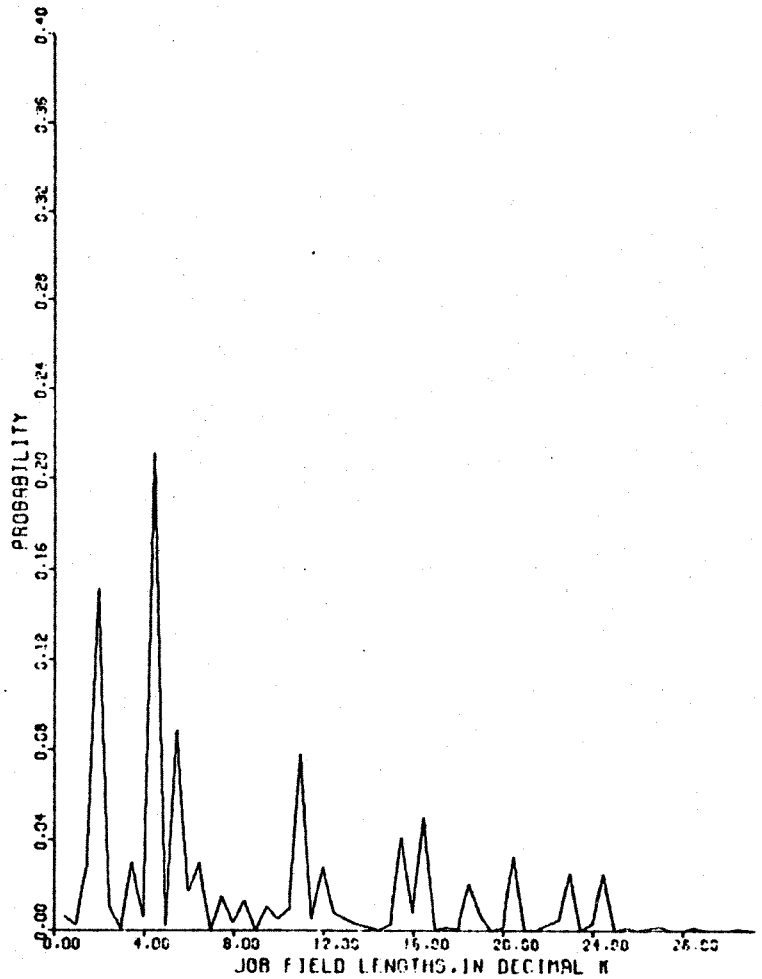


Figure 4.7B (Tape 2)

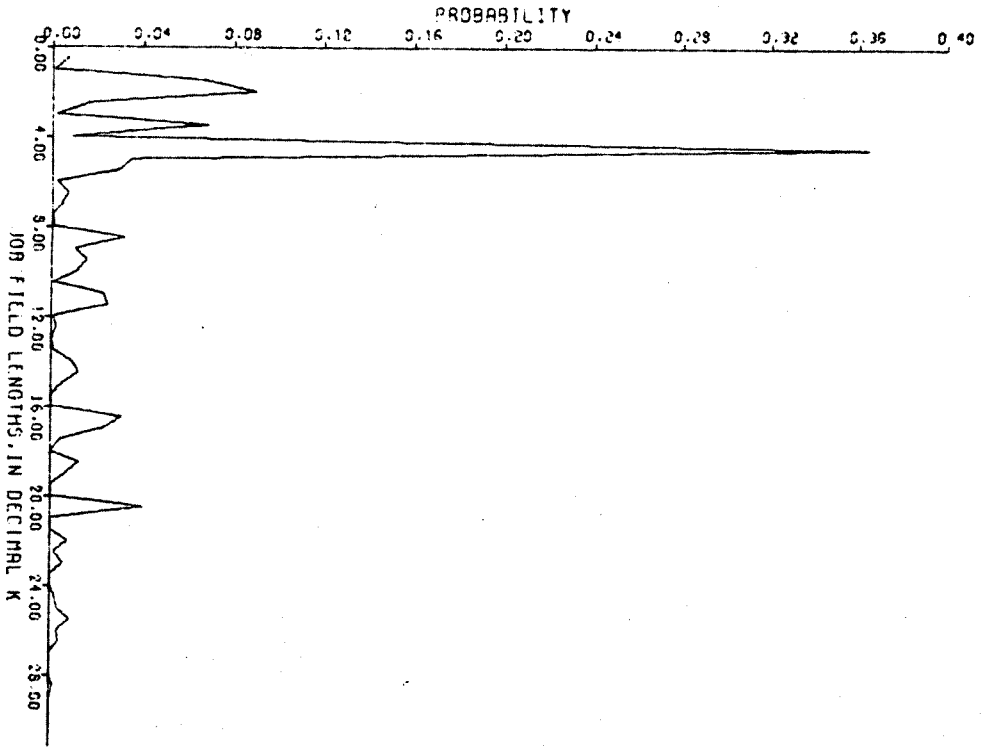


Figure 4.7C (Tape 3)

TAPE	UTILIZATION	RATE (CALLS/SEC)	MEAN (msec/CALL)	SQUARED COEF. OF VARIATION (K**2)	MAX (msec)
1	0.058	6.260	9.330	0.011	58
2	0.052	5.670	9.261	0.014	56
3	0.061	6.104	10.011	0.026	56

Table 4.1 Scheduler Runs

TAPE	UTILIZATION	RATE (CALLS/SEC)	MEAN (msec/CALL)	SQUARED COEF. OF VARIATION (K**2)	MAX (msec)
1	0.330	7.878	41.968	5.882	1327
2	0.193	6.305	30.671	7.737	1496
3	0.266	7.340	36.259	7.615	1521

Table 4.2 Satisfied RSTs

The workload being processed on the machine being modeled, is the interactive workload. The chief goal of the system in processing this workload is to provide fast response to the majority of users. The success of the memory manager is, therefore, best measured by the response time provided to the users. Although the secondary goal of efficient use of resources is not necessarily satisfied, it will probably be adequately met if the primary goal is achieved. As an example, it should be evident that high CPU utilization will not necessarily provide fast response time, but good response time could not be provided to a system of moderate to heavy utilization with low CPU utilization. Table 4.3 presents data for response time for jobs as measured on the system for each of the three tapes respectively. Response time for these measurements was defined to be the time from notification of the job scheduler that the job was ready and desired access to memory, to the time when the job begins the rollout procedure in a "terminal wait" mode. The measurements are subdivided by ranges of TM time in milliseconds consumed in active processing. As can be seen, the average response time for all the jobs, including the less than 3% which require over one second of TM time in a single burst, is between three and five seconds for all tapes measured. Also note that on Tape 1, 80% of the jobs has a mean response time of 1864.99 ms., on Tape 2, 83% had mean response times of 1357.52 ms., and on Tape 3, 83% had mean response times of 1773.72 ms. It seems evident that the system, on

Table 4.3 Response Times

Tape 1	UTIL	RATE	COUNT	MEAN	K**2	COUNT>0	MEAN>0	K**2>0	MAX
RESPONSE TIMES	7.754	1.833	1537	4229.552	18.553	1537	4229.552	18.553	305222
0-3	1.668	0.706	592	2334.193	12.443	592	2334.193	12.443	88160
4-15	0.669	0.557	467	1200.042	8.610	467	1200.042	8.610	53660
16-63	0.421	0.205	172	2055.458	8.127	172	2055.458	8.127	67212
64-255	1.821	0.201	169	9034.745	18.314	169	9034.745	18.314	305222
256-1023	1.125	0.107	90	10485.351	3.585	90	10485.351	3.585	176235
1024-4095	1.138	0.044	37	25804.925	2.103	37	25804.925	2.103	214538
4096-16383	0.752	0.010	9	70084.852	0.752	9	70084.852	0.752	213599
16384+	0.177	0.001	1	1489921.386	IND	1	1489921.386	IND	148921

Tape 2									
RESPONSE TIMES	6.790	1.953	3292	3476.797	39.654	3292	3476.797	39.654	627172
0-3	0.419	0.750	1265	559.216	72.501	1265	559.216	72.501	130191
4-15	0.668	0.612	1032	1059.270	31.278	1032	1059.270	31.278	150016
16-63	1.144	0.266	450	4285.638	63.613	450	4285.638	63.613	627172
64-255	0.664	0.136	230	4720.663	26.415	230	4720.663	26.415	344825
256-1023	0.730	0.112	189	6513.622	6.626	189	6513.622	6.626	222936
1024-4095	1.535	0.049	83	31181.675	2.897	83	31181.675	2.897	293815
4096-16383	1.668	0.025	43	65385.969	1.791	43	65385.969	1.791	563331

Tape 3									
RESPONSE TIMES	10.343	2.257	2404	4581.369	56.021	2404	4581.369	56.021	691527
0-3	0.231	0.449	713	345.752	3.618	713	345.752	3.618	109118
4-15	0.552	0.766	816	720.896	2.428	816	720.896	2.428	143794
16-63	1.655	0.415	442	3989.132	37.483	442	3989.132	37.483	380278
64-255	2.746	0.277	296	9880.137	20.342	296	9880.137	20.342	430241
256-1023	1.781	0.097	104	18235.501	18.333	104	18235.501	18.333	691527
1024-4095	2.021	0.024	26	82780.386	3.318	26	82780.386	3.318	441361
4096-16383	1.354	0.006	7	206048.374	1.110	7	206048.374	1.110	496812

the average, is providing excellent service to the majority of users, particularly the smaller users for whom the system is primarily intended.

Table 4.4 presents figures concerning CPU allocation. The primary goal of the memory scheduler is to provide fast response. It is thought that CPU utilization will normally fall within "acceptable" bounds. Due to the fact that the system is managing a very active time-sharing system, characterized by mostly small service requirement, the overhead is relatively high and constant varying from only 37% to 40% of the total CPU allocation. The CPU utilization by users varies over a range of 23% to 40%. This variance is due mostly to perturbations in the job load as evidenced by the fact that the sum of idle and user time is nearly constant, varying from 57% to 60%, thus indicating that the unused user time goes to idle.

To recap this section, what has been shown is that the domain of operation presented by the policy scheduler is not a strange, atypical job mix. The CPU is operating in a range which is acceptable for the type of work done and the service provided to the majority of the users would be quite acceptable for most installations.

The UT-2D policy sheduler has been shown to be representative of a broad class of schedulers. Thus, results obtained in this study should have application to other operating systems working in similar environments. Since the UT-2D scheduler's decisions are

Tape 1 STATISTIC	UTIL	RATE	COUNT	MEAN	K**2	COUNT>0	MEAN>0	K**2>0	MAX
CPU ALLOCATION	0.996	709.995	595215	1.404	7.909	511105	1.635	6.650	169
IDLE	0.284	147.344	123524	1.930	4.171	92980	2.565	2.892	17
TAURUS	0.019	19.022	15947	1.017	2.306	12013	1.350	1.490	16
PISCES	0.001	3.911	3279	0.509	0.573	2664	0.626	0.277	3
USERS	0.292	184.720	154858	1.585	4.773	122844	1.998	3.579	17
CP MTR	0.396	354.996	297607	1.121	14.413	280604	1.189	13.532	169
Tape 2 CPU ALLOCATION	0.999	643.254	1084165	1.553	6.705	915253	1.839	5.504	114
IDLE	0.201	97.911	165023	2.059	4.052	117458	2.893	2.596	17
TAURUS	0.023	16.880	28451	1.376	2.704	21326	1.836	1.776	16
PISCES	0.000	0.510	861	0.533	0.660	666	0.689	0.284	2
USERS	0.401	206.325	347748	1.943	4.110	255495	2.645	2.754	17
CP MTR	0.372	321.626	542082	1.159	12.314	520308	1.207	11.779	114
Tape 3 CPU ALLOCATION	0.998	566.836	710062	1.496	6.819	599292	1.773	5.599	125
IDLE	0.353	164.227	174873	2.150	3.954	126752	2.966	2.591	17
TAURUS	0.027	22.476	24359	1.211	2.923	17281	1.707	1.783	16
PISCES	0.000	1.074	1144	0.524	0.731	835	0.718	0.263	1
USERS	0.230	145.241	154456	1.587	4.688	112090	2.190	3.122	17
CP MTR	0.385	333.417	355030	1.157	11.761	342334	1.200	11.305	125

Table 4.4 CPU Allocation

representative, it was hoped that the explicit simulation of these decisions would be unnecessary. If the coupling effect between the policy scheduler and physical mechanism is weak enough, quantitatively accurate results may be obtained from the model by accepting the decisions of the policy scheduler as recorded on the trace tapes. This coupling effect is discussed in the following section.

4.1.2 Scheduler and Physical Mechanism Coupling Effects

The representativeness of the UT-2D job scheduler was shown in order to allow the possibility of using its decisions to drive the simulation model. However, the physical mechanism and the policy scheduler are not totally independent. It is possible that the operations of the physical mechanism indirectly affect the operation of the policy scheduler. This section will first describe how this is possible and then present data concerning the magnitude of this coupling effect.

The purpose of the policy scheduler is to determine which jobs should be allocated memory. This determination is based upon some functional evaluation of the "desirability" of the job as perceived by the job scheduler. It therefore seems evident that the running of the job scheduler, to determine what decisions are necessary, is only required whenever a job changes one of the factors by which its "desirability" is judged or when the status of the resource being allocated, the amount of available memory, changes.

The physical management task of memory management is the implementation of the decisions (explicit or implicit) of the job scheduler. All rollins, rollouts and storage moves are performed by the physical mechanism. All storage moves are caused by jobs which request a storage increase. While some of these storage increases are the result of jobs which are already in memory and desire to expand, most of the storage moves are performed to satisfy an "RST up" (a storage increase) for a rollin. Since all rollins are the result of decisions by the policy scheduler, it can be seen that the rate at which the physical mechanism must alter the memory configuration is largely determined by the frequency with which the job scheduler must run. The frequency of running the job scheduler is determined by the rate at which jobs go through the cycle of requiring and releasing memory. Part of this cycle is of course the time spent waiting for and in memory, and this can be effected by the efficiency of the physical mechanism. Thus, it has been shown that the physical mechanism can effect the rate at which jobs traverse from requesting to releasing memory. This effects the frequency of running the job scheduler which in turn, effects the frequency of running the physical mechanism to implement the decisions, and the coupling effect between physical mechanism and policy scheduler is evident. If this coupling effect is large, then changing the physical mechanism could perturb the scheduler's operation to an extent too great to allow the simulation model to accept the decisions as recorded on the tape. This is not the case as will now be shown.

There are several processes which take place in the processing of a scheduler's decision. However, all of these processes remain unchanged in the simulation model except for one. This one exception is the amount of memory which is moved to satisfy an "RST up." The simulation model changes the algorithm for the placement of new jobs in memory. Since this changes the physical location of some of the jobs in memory, it ultimately results in different jobs being moved. The algorithms modeled are improvements over the current algorithm so their effect is to decrease the average amount of work which the physical mechanism must perform to implement the decisions of the policy scheduler.

Presented in Table 4.5 are statistics which indicate the nature of the approximately 40% CPU overhead by monitor of which storage move is apart. On all three tapes, the storage move overhead is on the order of 6% to 8%. If all this time could be eliminated, then scheduler decisions could be implemented sooner. They might be moved up by the total amount of time eliminated up to that point. This would represent the greatest possible improvement. This is obviously impossible because this would imply the existence in memory of gaps of sufficient size to allocate the requested memory without moving jobs, but if this were possible, the system would have done so. Realistically, the best that could be expected is for some redistribution of the gaps which would allow more of the jobs to be directly allocated and possibly reduce the number of jobs moved when compaction is necessary.

	UTIL	RATE	COUNT	MEAN	K**2	COUNT>0	MEAN>0	K**2>0	MAX
Tape 1									108
CP MONITOR	0.399	758.024	635480	0.525	25.975	403379	0.827	16.122	
DO FUNC	0.002	23.064	19336	0.094	1.579	7495	0.244	0	0
SCHEDULE	0.058	6.260	5248	9.330	0.011	5248	9.330	0.011	58
MOVE	0.078	10.669	8945	7.403	5.283	7391	8.960	4.192	108
COMM	0.018	0.477	400	37.833	0.012	400	37.833	0.012	46
EXCHANGE	0.036	85.128	71366	0.434	0.109	71344	0.434	0.108	1
CP FTN	0.104	150.734	126366	0.693	7.493	88137	0.993	4.923	21
PP FTN	0.044	125.696	105376	0.351	0.587	85667	0.432	0.290	1
CPM EXIT	0.047	354.998	297608	0.135	1.561	136862	0.293	0.178	1
DISPLAYS	0.006	0.996	835	6.903	0.000	835	6.903	0.000	7
Tape 2									114
CP MONITOR	0.372	693.933	1169581	0.537	23.503	808827	0.776	15.945	
DO FUNC	0.002	22.532	37977	0.088	1.748	13820	0.244	0	0
SCHEDULE	0.052	5.670	9557	9.261	0.014	9538	9.279	0.012	56
MOVE	0.061	8.073	13607	7.676	5.454	11012	9.485	4.223	114
COMM	0.016	0.391	660	42.520	0.001	660	42.520	0.001	50
EXCHANGE	0.034	79.616	134188	0.432	0.099	134165	0.432	0.099	1
CP FTN	0.097	111.482	187897	0.871	4.916	144236	1.135	3.541	22
PP FTN	0.051	143.540	241929	0.358	0.573	212217	0.408	0.380	5
CPM EXIT	0.048	321.627	542083	0.149	1.219	281496	0.288	0.152	1
DISPLAYS	0.008	0.998	1683	8.600	0.000	1683	8.600	0.000	9
Tape 3									105
CP MONITOR	0.345	725.501	772530	0.532	22.568	520750	0.789	14.886	
DO FUNC	0.001	21.422	23024	0.046	1.818	8168	0.244	0	0
SCHEDULE	0.061	6.104	6500	10.011	0.026	6478	10.045	0.022	56
MOVE	0.063	9.613	10237	6.616	5.650	8342	8.118	4.419	105
COMM	0.018	0.398	424	46.155	0.003	424	46.155	0.003	53
EXCHANGE	0.034	82.125	87449	0.421	0.108	87415	0.421	0.108	1
CP FTN	0.105	111.495	118723	0.856	4.195	94699	1.074	3.143	20
PP FTN	0.057	159.725	170079	0.358	0.567	146559	0.416	0.350	4
CPM EXIT	0.045	333.419	355032	0.135	1.432	167603	0.286	0.148	1
DISPLAYS	0.008	0.997	1062	8.575	0.000	1062	8.575	0.000	9

Table 4.5 CP Monitor