STEPS TOWARD A GENERAL PURPOSE TIME-SHARING SYSTEM USING LARGE CAPACITY CORE STORAGE AND TSS/360

By

Richard E. Fikes
Hugh C. Lauer
Albin L. Vareha, Jr.

Carnegie-Mellon University
Pittsburgh, Pennsylvania
March, 1968

This paper has been submitted for publication at the 1968 National Conference of the Association for Computing Machinery.

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense: Contract SD-146 (monitored by the Air Force Office of Scientific Research) and in part by a grant from the Mellon family and the Richard King Mellon Foundation. Distribution of this document is unlimited.

ABSTRACT

This paper is a progress report of an effort at Carnegie-Mellon University to determine how a large capacity core storage facility (LCS) can be used to reduce the demand paging overhead costs in the IBM System/360 Time Sharing System (TSS/360) and in similar general purpose time-sharing systems. A discussion is presented of how the number of paging operations and the cost of each paging operation can be reduced by using LCS as both a swapping device and an extension of executable core. Two problems which arise are considered. One is the design of an effective method for determining whether pages should execute in HSS or in LCS. An analysis of this problem is presented and possible solutions are discussed. The second problem is the development of efficient core management algorithms for LCS. A new algorithm for releasing core is presented and compared with two existing algorithms. Finally, results from a feasibility implementation of the ideas in a pre-release version of TSS/360 are presented as a demonstration of the validity of using LCS to reduce paging overhead.

TABLE OF CONTENTS

INTRODUCTION

It is our desire at the Carnegie-Mellon University Computation Center to provide to the university community a general purpose computer utility capable of supporting the wide range of academic and research programs which are currently being advanced. In particular, we wish to develop an integrated system of processors, storage devices, communication channels, and software which will:

- Accept conversational, real-time, and batch tasks at any given time and process them concurrently.
- Provide languages and facilities for writing, debugging, and executing programs conversationally.
- Contain a file system suitable for safely storing source code, object code, data, and output.
- Accommodate the addition of new languages, new software subsystems, and new types of conversational terminals.
- Accept tasks ranging in size from very small student programs and desk calculator work to very large tasks which tax all of the facilities of the system.

The first attempts to provide systems having these capabilities resulted in Project MAC's CTSS at Massachusetts Institute of Technology[1] and the Q-32 Time-Sharing System at System Development Corporation.[2] Other early efforts produced JOSS at the RAND Corporation[3] and BASIC at Dartmouth College.[4] The latter two systems emphasized conversational computing, ease of operation, and file facilities, but were dedicated systems in that they provided only a single algebraic language.

Today's commercial systems have adopted many of the features developed by these early time-sharing projects, but none can qualify as the desired general purpose computer utility. For example, RUSH[5] provides a time-shared conversational environment, but is a dedicated system. Some of the large operating systems such as SCOPE II for the CDC 6600,[6] GECOS for the GE 625,[7] and IBM's OS/360[8] provide more complete programming environments and perhaps remote job entry from terminals, but are lacking in conversational capabilities and file facilities.

The so-called "second generation time-sharing systems" - namely, IBM's TSS/360,[9] MULTICS for the GE 645 at Project MAC,[10] and the SDS Sigma 7 system[11] - represent the first deliberate attempts to incorporate all of the capabilities set forth above into a single hardware-software system. The difficulties which these systems are experiencing discourage many people, but we believe they contain concepts in their design which can form a nucleus for the desired system. These concepts include virtual memory, program segmentation, sharable segments, reentrant program code, hardware reference bits and change bits, demand paging, and configuration partitioning.[12] With this point of view, Carnegie-Mellon has undertaken a study and redesign of the TSS/360 system in an attempt to bring the desired general-purpose computer utility closer to reality. This paper is a report of the results and progress of that effort.

THE IBM 360/67 SYSTEM WITH A LARGE CAPACITY STORE

Figure 1 indicates some of the elements of the Carnegie-Mellon 360/67 hardware configuration. The CPU and each channel access memory as a homogeneous, directly addressable core of 8,912,896 bytes. The only difference between the high speed store (HSS) and the large capacity store (LCS) is in cycle time and access time. Thus the disk, tapes, data cell, and peripherals can read from or write to either store.

The storage channel is a device which reads from one memory location and writes into another, effecting a core-to-core data transfer. In normal operation it is limited in speed by the memory's data transmission rate and by the interference on the memory bus controller due to access requests from other devices (including the CPU). The transmission rate of two way interleaved 8 μsec LCS is two million bytes per second and therefore represents the maximum transmission rate to or from LCS which the storage channel may achieve. Interference degradation varies widely depending on the demands for the memory bus controller. Actual measurements show that if the storage channel is transferring data from HSS to LCS and the CP is executing from HSS, then the storage channel transmission rate ranges from 1.2 to 1.8 million bytes per second. This rate drops significantly when the CPU and/or other devices are accessing LCS.

A storage hierarchy is formed in the system by HSS, LCS, the disks, and the data cell. The storage at each level is larger, slower, and less expensive per bit than the storage at the previous level. Figure 2a illustrates an ideal organization of such a hierarchy in which information could flow back and forth between adjacent levels and the CPU could directly access information at any level. This organization cannot yet be achieved,

but our system does allow the approximation to the ideal illustrated by Figure 2b. Here we see that information can pass between levels 1 and 2 via the storage channel and between levels 2 and 3 via the selector channel. But for information to pass from a disk to the data cell it must first be moved either to LCS or HSS since no path exists between levels 3 and 4. The CPU does not have direct access to levels 3 and 4 (i.e., information must be transferred to core before being referenced), but it can access instructions and data stored in either levels 1 or 2.

It is this availability of two levels of executable storage which distinguishes the CMU configuration from the standard IBM 360/67, GE 645, and SDS Sigma 7 in which storage level 2 is a high-speed drum. We examine in the next section the implications of this distinction with regard to the problems which plague the second generation time-sharing systems.

THE NEED FOR LCS

At this writing, none of the second generation time-sharing systems work satisfactorily, and there is no single problem or group of problems to which the failure can be attributed. However, a dominant problem seems to be the extremely high overhead required to support each task. Because of this overhead, only a few conversational tasks can be given acceptable service at any one time. The primary concern of our work with TSS/360 has been to reduce this overhead and thereby increase the capacity of the system.

To understand the sources of this overhead let us review the demand paging mechanism which is present in all three systems. When a task executes, only that local portion of its program and data which is relevant

at a given time (the working set) need reside in core storage. The remainder may reside on a lower level storage device (the swapping device). What the program "sees" is not physical core but a two dimensional address space called a virtual memory. A hardware mapping mechanism uses a set of tables (called page tables) for each task to translate an access of a virtual memory address into an access of a physical core location. If the block (or page) of virtual memory containing the address being accessed does not reside in physical core, then an interrupt is generated and the system monitor causes the page to be read into core (paged-in) from the swapping device. When the transfer is complete, the monitor sets the page tables to indicate that the page is now part of the task's working set. When a page is no longer needed it can be paged out to the swapping device and deleted from the working set. Thus working sets vary dynamically and active portions of many independent tasks can simultaneously reside in core.

Core can be thought of as being allocated to a task in units of page-seconds, where one page-second is one page of core (4096 bytes in TSS/360) allocated for one second. The page-second costs associated with a demand paging mechanism were analyzed in an earlier paper by Hugh Lauer.[13] He showed that if t seconds are required to transfer a page from a swapping device into core and if a task requires a working set of size n during a time-slice of computing, then a total of

$$\frac{n(n+1)}{2}\, t = \left(\frac{n^2}{2} + \frac{n}{2}\right) t \qquad \text{page-seconds} \qquad (1)$$

of core must be allocated to the task just to transfer its pages from the swapping device. That is, if T is the length of a time-slice, then the system expends

$$\left(\frac{n^2}{2}+\frac{n}{2}\right)\frac{t}{T} \qquad (2)$$

page-seconds of core to prepare for each second that a task computes. Each such page-second spent in preparation represents one second during which a page is assigned to a dormant task.

We have attacked the overhead problem in TSS/360 in two ways:

- by reducing the number of paging operations required to support each task, and
- by reducing the costs associated with each paging operation.

We have attempted to achieve these reductions by using LCS both as a swapping device and as an extension of physical core. That is, we associate with each page of virtual memory a marker indicating if that page is to be accessed by the CPU (i.e., executed) while residing in HSS or while residing in LCS. Pages which are executed in LCS use a disk as a paging device and migrate back and forth between LCS and the disk. Pages which are executed in HSS use LCS as a paging device and migrate back and forth between HSS and LCS.

LCS is a more favorable swapping device than a drum since LCS has no rotational delay before data transfer can begin as does a drum. This means that using LCS as a paging device for the virtual memory pages which execute in HSS reduces the value of t in equation (1) and therefore reduces the cost in HSS page-seconds of paging operations. The disk is a slower paging device than a drum, but in this case the cost is in the much more plentiful LCS page-seconds. We obtain another important reduction in HSS page-second cost by executing some of the task's working set in LCS and thereby reducing

the value of n in equation (1). Thus, the paging operations which require allocation of a scarce resource - namely, HSS page-seconds - have been reduced in cost.

By allowing some virtual memory pages to execute from LCS we have increased the size of executable core available in the system. In the standard configuration without LCS it is necessary to assume that a task begins each time slice with none of its virtual memory pages in core. With the addition of LCS it is possible to allow working set pages to remain in executable core between time-slices. This implies a large reduction in the number of paging operations since a task need not page-in its entire working set at the beginning of each time-slice.

Thus, by using LCS we have achieved our goals of reducing the number of paging operations required for each task and the cost of each paging operation. Two problems arise which will be discussed in the following sections. The first is that the LCS access time is more than ten times that of HSS, so that when some of a task's virtual memory pages are executed in LCS it requires longer to do a given amount of processing. Hence the system designers must carefully consider the issue of deciding whether each virtual memory page will execute in LCS or in HSS. The second problem is that because of the size of LCS the core management algorithms used for HSS are inadequate for LCS and new ones must be devised.

THE ALLOCATION OF VIRTUAL MEMORY PAGES

For each virtual memory page the system must decide whether that page should be executed in LCS or in HSS. Given that a page is residing in LCS, consider the relative costs of these two alternatives. If a page is executed in LCS there is a processing cost in that each access to LCS takes longer than an access to HSS. In our configuration an LCS access takes approximately 3.6 μsecs longer than an access to HSS. There is no direct HSS page-second cost incurred when a page is accessed in LCS, but the extra processor time required by the task will mean that the task's pages which are executed in HSS may have to remain in HSS for a longer period. Hence, the cost of leaving a page in LCS to be executed there is in processor time and amounts to the difference in HSS and LCS access time multiplied by the number of accesses to the page.

If a page is transfered into HSS when it is accessed by the processor then there is both an HSS page-second cost and a processing time cost. These costs occur because of the processing time required to perform a page-in operation to HSS and because the task and the HSS core pages allocated to it must remain dormant while the operation takes place. In TSS/360 when the virtual memory page is accessed and found not to be in HSS, an interrupt is generated and the following processing occurs:

- The interrupt is recognized, the virtual memory page is located in LCS, a core page is allocated in HSS for the virtual memory page, a request is made to the storage channel to perform the transfer into HSS, and the task which initiated the access to the virtual memory page is marked inexecutable.
- The storage channel performs the transfer and generates an interrupt.

. The storage channel interrupt is recognized, the successful completion of the page-in operation is recorded, and the task is marked ready to execute.

These operations take on the order of $\frac{1}{2}$ to 1 msec of processor time and $\frac{n}{2}$ to n page-msecs of HSS in TSS/360, where n is the number of HSS core pages allocated to the task at the time of the page-in operation.

We may define for each virtual memory page an access density as follows:

> The access density of a virtual memory page is the average number of accesses to the page per time slice, where only those time slices during which the page is accessed at least once are considered in computing the average.

Given this definition, we would expect a page of program code containing loops to have a greater access density than a page of code having no loops; also, we might expect a page of data to have a smaller access density than a page of program code. Based on the cost evaluations given in the preceding paragraphs the first criterion for deciding whether or not to execute a given page in LCS or to transfer it into HSS is the access density of the page. There exists a threshold access density value, AD_t, for which the processing overhead to move a page into HSS is equal to the additional cost of making AD_t accesses to LCS rather than to HSS. In TSS/360 this threshold is on the order of 200 accesses per time-slice per page. Hence, if the access density of a page is less than AD_t then the page should be left in LCS; if it is greater, then the page is a candidate for being paged-in to HSS.

The second criterion for making the LCS to HSS paging decision is based on the availability of HSS page-seconds in the system. The system must either maintain an estimate of the number of HSS core pages needed during each time slice and not begin a time slice until that number is available, or allow HSS core pages to be "stolen" from a task while the task is in a time slice. In either case, as the number of virtual memory pages in the system which are executed in HSS increases, so does the amount of time each task must wait to begin each time slice and/or for page-in operations to complete. Hence, the total number of virtual memory pages in the system competing for HSS space must be limited.

We may now make some comments on ways in which the system can mark each virtual memory page to be executed in HSS or in LCS. Y. C. Chen has considered this problem in a multiprogramming environment where each page of a task allocated in HSS remains there until the task terminates.[14] He suggests that for scientific programs all data and all code which form program loops be allocated in HSS, with the remaining code allocated in LCS. His conclusions are based on the first criterion discussed above in which the access density of a page is the determining factor. His analysis differs from ours in that he is considering the total number of accesses to each page during the lifetime of the task while we must consider only the number of accesses during one time slice of computing. Hence we would expect an optimal scheme for our system to leave more virtual memory pages in LCS than does Chen's. In particular, it seems that an algorithm which transfers to HSS all executable program code and leaves in LCS all data and all interpretable program code (data to an interpreter) would be closer to optimal for the type of system we are considering because it

would tend to place the most densely accessed virtual memory pages in HSS.

The question then arises whether the 'executable code - data' division we have suggested is an acceptable one with respect to the second criterion, i.e., the availability of HSS page-seconds. With this division, as the number of tasks in the system increases so will the number of virtual memory pages competing for HSS core pages. We have no way of predicting at what point this increase will cause enough overhead to significantly effect system performance; that it will be a problem, there is no doubt.

When we designed the first implementation of these ideas in TSS/360 we had the additional constraint that the effort require no more than one man-month of work. The scheme chosen was to execute in HSS all sharable virtual memory pages and execute non-sharable pages in LCS. In TSS/360 a sharable page is one which may be included in the address space of more than one task. In particular, all system code and system data is sharable, e.g., the monitor, the assembler, compilers, interpreters, system catalog. We have imposed the restriction in this implementation that a user can specify virtual memory pages as sharable only if he is given special permission to do so. Hence, we essentially allow the system to execute in HSS and require all user programs to execute in LCS. We do not claim that this division is optimal nor will it be our final choice, but it has several desirable properties as follows:

- The system pages are almost entirely reentrant, executable code, and therefore represent a subset of the pages which would be allocated in HSS by the 'executable program code - data' division suggested above.

- A system page is accessible by all tasks and therefore may be accessed more densely than a page of code assigned to only one task.
- There are approximately 200 systems pages in TSS/360 and this number does not increase measurably as the number of tasks in the system increases. Hence, the demand for HSS space is not so great as to cause excessive paging and the accompanying overhead.

The major problems with this division are that user tasks with heavy processing requirements will receive degraded service since all of their program code will be executed from LCS and that certain system pages with low access densities are paged to HSS unnecessarily. These problems might be solved by allowing compilers, assemblers, or system programmers to specify to the system which pages will be heavily accessed. We might also monitor the behavior of programs to be optimized, particularly system programs, under typical operating conditions in order to develop activity tables for the system to use in making the LCS or HSS decisions.

Finding optimal decision algorithms for assigning pages is but one part of the task of implementing LCS in TSS/360. It is a problem into which we hope to gain more insight as we gain experience with the system.

CORE MANAGEMENT

A major portion of the core management problem in a paging environment such as we have been considering is how to make core pages available when they are needed. That is, in general there will not be enough LCS and HSS space to contain all the virtual memory pages of all the tasks in the system. Hence, at any given time some virtual memory pages which are executed in LCS must be "swapped out" on the disk and some virtual memory pages which are executed in HSS must be "swapped out" in LCS. There must be an algorithm in the system for determining when to swap pages out and which pages to swap out. In this section we will examine two existing algorithms for performing this function and present a new one for the CMU-TSS system.

In the release 1 version of TSS/360 (TSS-1) the mechanism that frees core-pages is activated at the end of each time slice.[15] All private (non-sharable) virtual memory pages accessed during the time slice are swapped out at that time. If a page has been altered during the time slice (indicated by a hardware change bit for each core page), then a page-out operation is initiated to store a valid copy of the page on the swapping device. If a page has not been altered during the time slice, then the current copy of the page on the swapping device is still valid and no page-out operation is required. This method of freeing core space assumes that in most cases the core pages assigned to a task will be needed by the system before that task begins its next time slice. Hence, it is not profitable to incur the cost of a more elaborate mechanism which would selectively free core based on the needs of the system.

A second typical mechanism for freeing core pages is the one proposed for the MULTICS system.[16] This mechanism uses the principle that a core page which has been referenced frequently and/or recently is more likely to be referenced again before a page which has not been referenced as frequently or as recently. In the hardware there exists for each core page a reference bit such that whenever an access is made to a core page the reference bit for that page is set. Instructions exist for reading these bits and resetting them. With this facility the system can monitor the accesses being made to each core page.

In the MULTICS proposal there is a ranking routine which is called periodically to maintain a chain of pointers to core pages ordered with respect to the frequency and recency with which each page has been referenced. The ordering is maintained by testing the reference bit of each page represented in the chain. If a reference bit for a page is set, its pointer is moved to the end of the chain and the reference bit is cleared; if the reference bit is not set, no action is taken. Whenever the number of available core pages drops below a threshold, a replenisher routine is called to provide available core. The routine removes a given number of pointers from the top of the ranking routine's core page chain and frees the corresponding pages for use by the system. (Note: Ranking pages according to frequency and recency of use is not to be confused with determining the access density of a page. So far as the ranking routine is concerned, one access per page in an interval of time counts as much as one million accesses to the page in the same interval of time. The access density is concerned with the number of accesses in an interval, and it cannot be measured with any hardware commercially available at this time.)

When we considered the core management problem for our system we found both of the approaches outlined above inadequate. The first scheme where all private core pages are freed at the end of each time slice is satisfactory for HSS core pages and we will continue to use it. But for the 2048 LCS pages, the probability is quite high that the system will not need to free the LCS core pages assigned to a task while the task is between time slices. This increased probability is worth exploiting since if a core page can remain assigned to a task and the task references that page in its next time slice, then the system has saved a possible page-out operation and page-in operation, and the task can obtain a time slice uninterrupted by a wait for the page-in operation to complete. Therefore, we wish the system to free LCS pages only when there is actual demand for them in the system.

The second scheme, where a chain of core pages is maintained, has the desired property for LCS of freeing core only when it is needed but was rejected because it requires an excessive amount of overhead processor time in a system with a large number of core pages. In the 360/67 it would take approximately 35 μsecs. to process each LCS core page during the scan made by the ranking routine. This processing includes testing the reference bits and moving the pointers in the chain if necessary. This implies that the scan of a chain containing pointers to 2000 LCS pages would require 70 msecs. of processor time. Executing this scan just twice per second would tie up the processor 14% of real time.

Even if an accurate ordering based upon frequency and recency of use could be inexpensively maintained, it is not clear that ordering is the correct basis for determining what core pages to free. The problem with the ordering is that a page chosen to be freed may be assigned to a task

which has been between time slices (and therefore not referencing the page), but is just beginning a new time slice in which it will attempt to reference the page. One would like to order the core pages based upon the time at which they will next be referenced and release those with the longest projected inactive time (PIT) first. Although the system does not have such a projection on each core page, there is information in the system concerning the PIT of each task. Since a non-sharable virtual memory page belonging to a task can be referenced only when the task is in a time slice, a chain of tasks ordered with respect to the PIT of each task can serve as a partial ordering of the core pages which are allocated to non-sharable virtual memory pages. Then when a request is made to release core pages to the system, core assigned to the task at the top of this core release chain can be freed. We now consider the problem of defining a task ordering to serve as a basis for such a core release procedure.

At any given time a task in the system is in one of three states: active, scheduled, or unscheduled. A task is in the active state during its time slice. A task is in the scheduled state when it is ready to begin its next time slice and has been assigned a place in the scheduling queue. (We are assuming that at any given time the first task in this scheduling queue will be the next task to begin a time slice.) A task is in the unscheduled state when its time slice has been terminated and it is waiting for an input/output operation to complete or for a message to be input from the user's console. We are assuming an internal scheduling scheme such that when a task issues an input/output request and must wait for the completion of the operation, it is allowed to remain in its time

slice for a specified period of time. If at the end of that period the task is still waiting, then its time slice is terminated and it enters the unscheduled state. Hence in most cases, tasks which issue a request for an input/output operation on a high speed device can remain active until the operation completes. Figure 3 illustrates the movement of tasks in the system from one state to another.

The tasks in the active state may be said to have PITs of zero since they are in a time slice. When a task enters the active state it is removed from the core release chain. This prevents core pages assigned to active tasks from being stolen by the system. When a task completes its time slice and enters the scheduled or unscheduled state it is returned to the core release chain.

When a task enters the scheduled state it is queued to begin its next time slice. This queue defines the ordering of the scheduled tasks in the core release chain; that is, if task x is behind task y in the scheduling queue, then task x is placed above task y in the chain.

Tasks in the unscheduled state have unknown PITs and the system has little or no information to use in predicting when such a task can be scheduled. We have placed all unscheduled tasks above the scheduled tasks in the chain even though when a task enters the scheduled state it may be placed in front of other scheduled tasks in the scheduling queue. In so doing we have assumed that the probability of unscheduled task x beginning its next time slice before scheduled task y is less than one-half.

It remains to specify an ordering among the unscheduled tasks. When a task enters the unscheduled state,we have chosen to enter it immediately above the highest scheduled task in the core release chain. This means

that the longer a task remains unscheduled the greater the probability of it being at the top of the chain and hence having to give up core pages. This ordering has the effect of giving preferred service (i.e., faster response time) to the active terminal user at the expense of the user who is sitting idle in a long "think time" or at the expense of a task waiting for completion of an input/output operation on a slow speed device.

We now have defined for all tasks not in the active state an ordering to be used for freeing LCS core pages when the need exists. Figure 4 illustrates this ordering. The freeing operation consists of removing the task at the top of the chain and freeing core pages assigned to that task.

We have further refined this core release algorithm based upon the following two considerations. First, note that it is possible for a core page to remain assigned to a task which is no longer referencing the page. This situation can occur for a compute bound task or for a task whose user is active at his terminal, since such a task has a low probability of reaching the top of the core release chain while it is between time slices. Hence it may be advisable to release those pages assigned to a task x which are not being referenced before releasing those pages assigned to a task y which are being referenced even though task x is below task y in the core release chain. The second consideration is that if a copy of the information in an LCS core page exists on the disk and the page is unchanged, then it can be freed without a page-out operation. Since at any given time there will be a large pool of freeable pages in the system from which to obtain core, a significant amount of overhead can be saved by freeing pages which do not require a page-out operation before freeing those which do.

We have responded to these two considerations by decomposing the core release chain into four chains. When a task leaves the active state it is entered into chain 1 using the same ordering relationships described above. When the system needs core it removes the top task from chain 1 and does the following:

- Each core page assigned to the task which is unreferenced and has a valid copy on the disk is freed.
- The reference bit is cleared for each core page assigned to the task which is referenced and has a valid copy on the disk.
- The task is appropriately entered into chain 2.

If chain 1 is empty when the system needs core, then it removes the top task from chain 2 and does the following:

- Each core page assigned to the task which is unreferenced and does not have a valid copy on the disk is freed, and a page-out operation from LCS to disk is initiated.
- The reference bit is cleared for each core page assigned to the task which is referenced and does not have a valid copy on the disk.
- The task is appropriately entered into chain 3.

If chains 1 and 2 are both empty when the system needs core, then it removes the top task from chain 3 and does the following:

- Each core page assigned to the task which has a valid copy on the disk is freed.
- The task is appropriately entered into chain 4.

If chains 1, 2, and 3 are all empty when the system needs core, then it removes the top task from chain 4 and does the following:

> Each remaining core page assigned to the task is freed, and page-out operations from LCS to disk are initiated.

This scheme first selects all freeable core pages which have not been referenced and for which a page-out operation is not needed. It then frees all pages which have not been referenced and for which a page-out operation is needed. It then frees those pages which have been referenced, but do not need page-out operations. Finally, the referenced pages for which page-out operations are needed are freed. Figure 5 illustrates the movement of tasks in the core release chains. A referenced core page, as referred to here, is one which was accessed by the task to which it is assigned after the last time the core release mechanism selected the task from the top of a chain and cleared the page's reference bit. This selection will occur at varying intervals for each task depending upon core demands in the system and the activity of the task, but the task will receive at least one time slice during each interval.

The way in which the system determines when LCS space needs to be freed is as follows. Each time a LCS page is allocated a test is made to determine if the number of available LCS pages plus the number of page-out operations in progress to the disk is less than a threshold. If the sum is less than the threshold, then a request is made to the core release mechansim to free core pages. Each time the core release mechanism is activated it releases at least k LCS pages, where k is a system parameter.

Consider the overhead involved in this core release algorithm. In contrast to the MULTICS proposal, no periodic scanning of a 2000 entry

core page chain is required. When the system needs core the only pages considered are those which are candidates for being freed, i.e., those assigned to tasks at the top of the core release chains. Each time a task is removed from the top of a core release chain and an attempt is made to free pages assigned to it, a scan of the list of core pages assigned to the task is made. If only one core release chain were used and all the pages assigned to the top task in the chain were released when the system needed core, then every page in the list being scanned would be freed. In the four chain scheme, there is an extra cost in that multiple scans of the core pages assigned to a task may be necessary and extra chain maintenance is involved. This is still a quite small amount of processing time to expend in gaining additional reduction in disk paging operations and task wait times.

Chain maintenance in this core release scheme, another source of overhead, consists of entering tasks into and removing tasks from the core release chains. Removing a task from a chain requires but a few instructions. Entering a non-scheduled task into a chain can be facilitated by keeping a pointer to the highest scheduled task in the chain. The non-scheduled task is then entered immediately above the highest scheduled task. Entering a scheduled task into a chain requires sorting it into its appropriate place among the scheduled tasks. This is the only chain maintenance operation requiring any appreciable amount of CP time.

Thus, we have an inexpensive LCS page release scheme which steals pages from tasks only when there is demand in the system for them and which orders the release of pages based on the projected inactive time of the page, the history of use of the information in the page, and the cost required to free the page.

The algorithm which we have presented does not contain a provision for the release of core pages containing sharable virtual memory. Since we have proposed that most sharable virtual memory pages should be allocated in HSS, this is not a significant problem in our system. It would be desirable to have associated with each sharable virtual memory page the number of tasks which are using the page. Then whatever mechanism is used to free private pages could also be used for sharable pages with the provision that when a sharable core page is to be released, its task counter is decremented instead and it is released only when that counter goes to zero. If such task counters cannot be implemented, as is the case in TSS/360, then one may adopt a scheme which periodically scans through a chain of the sharable core pages and releases those pages which are not being referenced.

EXPERIENCE WITH CMU-TSS

The previous two sections have dealt with the problems of interfacing LCS with TSS/360. In this section, we present a brief progress report of our experiences with the system.

Carnegie-Mellon received its first pre-release version of TSS/360 in the spring of 1967. Modifications were made in that system to use LCS, in place of the drum, as a swapping device in order to reduce the cost of each paging operation. All pages were executed in HSS and no changes were made in the core release algorithms. These modifications yielded only a modest improvement in system performance. It became clear to us that if the number of requests for paging operations was not drastically reduced, our attempts to reduce the cost of each paging operation would be pointless.

A later pre-release version of TSS/360 (version 58) was modified so that the processor directly accessed instructions and data from both HSS

and LCS. In this system sharable virtual memory pages were executed in HSS and private virtual memory pages were executed in LCS, as discussed above. We did not attempt to implement the LCS core release algorithm of the previous section in this version. Instead, we modified the TSS/360 core release algorithm so that it would not release core pages which were being referenced unless they were needed by the system. The details of the modified algorithm are described in an appendix. This core release mechanism was crude but was easy to implement and sufficiently effective to allow testing of the system with a small number of tasks.

An experiment was devised to compare the performance of the CMU-TSS system with the standard, drum-oriented TSS/360. We prepared a paper tape which contained the input messages for a terminal session. The task logged on, loaded the assembler, entered 83 lines of source code, requested an assembly, and logged off. The TSS Assembler performed a syntax check on each line as it was input and reported to the user any errors before accepting the next line. Runs were made on both systems by running one, two, four, and eight identical copies of the paper tape on teletypes. An internal logging routine recorded all interrupts, overhead, and terminal interactions on magnetic tape.

Figure 6 indicates the response times obtained. Response times during the syntax check (i.e., the time between the carriage return at the teletype and the request for the next line of source code) for the unmodified TSS/360 were extremely poor. With eight tasks on the system, average response time was greater than 30 seconds. In the modified system, the average was less than 0.1 second, independent of the number of tasks in the system.

Other results from the experiments indicate the potential of the modified system to service more than 8 tasks. We know that in the modified system each task had 25 private pages assigned in LCS and no paging of shared pages was necessary between HSS and LCS. There are approximately 200 sharable virtual memory pages in the system, so that 1800 core pages of LCS are available for private virtual memory pages. This implies that 72 conversational tasks having comparable memory requirements could be run without doing any paging at all. We also know from the experiments that CPU idle time was a linearly decreasing function of the number of tasks, and that with eight tasks in the system the CPU was idle 60% of the time. Most of this idleness occurred when all tasks were inputting text and there was no work to do. It appears that much of this time can be recovered when the system load goes up.

We do not claim that our experiments were precisely controlled and error-free or that the tasks used comprise a representative system load, but they do indicate orders of magnitude. The fact that we achieved an improvement of two orders of magnitude in response time demonstrated to us the validity of our hypothesis that the number of paging operations is a crucial factor in system overhead and that allowing the processor to directly access LCS is a feasible way of reducing overhead.

SUMMARY AND PROJECTION

We have explored the overhead costs of demand paging in TSS/360 and in similar general purpose time-sharing systems. We have discussed how that overhead can be reduced by incorporating LCS into the system. LCS reduces the total number of paging operations and the cost of handling

each paging operation, but it creates two new problems. One is the design of an effective method for determining whether pages should execute in HSS or LCS. We have presented an analysis of this problem and discussed possible solutions. The second problem is the development of an efficient algorithm for releasing LCS core pages. A new algorithm was presented and compared with the MULTICS and TSS/360 core release techniques. Finally, we demonstrated the validity of our hypothesis with a feasibility implementation of the ideas in a pre-release version of TSS/360.

With so many variables to consider, we cannot confidently extrapolate our results, nor can we predict the saturation point of the TSS/360 system. But looking ahead, we can see two new bottlenecks. One is the slow paging rate of the IBM 2314 disk. If we assume optimal scheduling of the disk seeks, we can achieve a maximum rate of 40 page transfer operations per second. The actual paging rate is less because disk channel time must be shared with accesses to system and user files and because the allocation of disk space results in sub-optimal scheduling of seeks. As the load on the system grows and LCS page-seconds become a scarce resource, the slow disk paging rate will become a factor with which to reckon.

The other bottleneck is the design of the memory bus controller. When the system load increases and more core accesses are demanded, the CPU and the storage channel experience severe degradation in speed. Thus service capacity is limited. We are currently considering this problem in detail.

Both problems, and others which we have not yet appreciated, keep us from attaining the ultimate in a general purpose time-sharing system. But the CMU-TSS system is a significant step toward that goal.

APPENDIX

The core release algorithm implemented for the experiments with the pre-release version of TSS/360 is shown in Figure 7. xxxT1 and xxxT2 are threshold values defined for HSS and for LCS such that xxxT1<xxxT2. HSSAVAIL and LCSAVAIL indicate the number of core pages available for assignment in HSS and LCS respectively. When xxxAVAIL drops below xxxT1, then core pages in xxx are freed at the end of each time slice until xxxAVAIL is built up to xxxT2. In addition, regardless of the value of LCSAVAIL, LCS pages which are not being referenced are freed at the end of each teletype interaction.

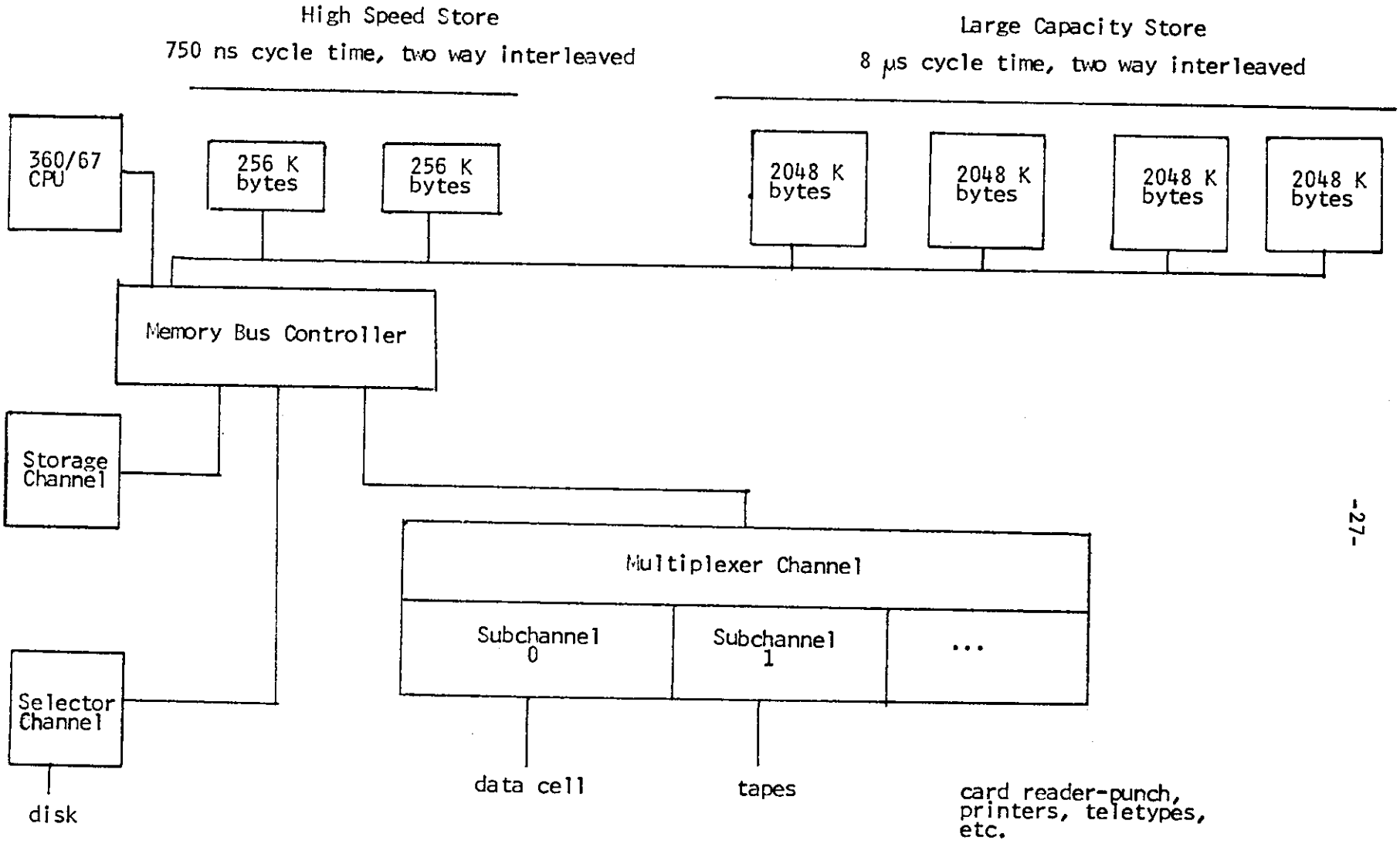

Figure 1. Hardware Configuration

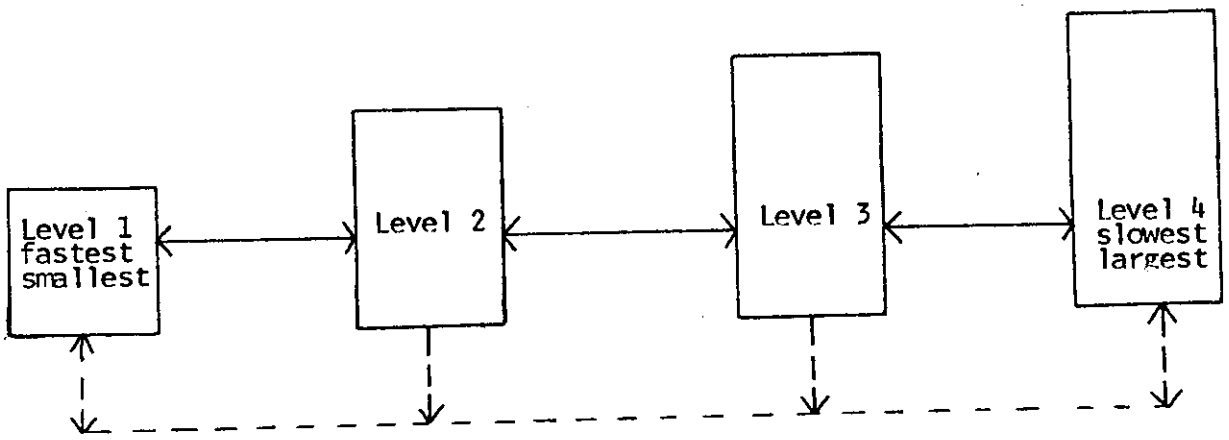

Figure 2a. Ideal Storage Organization

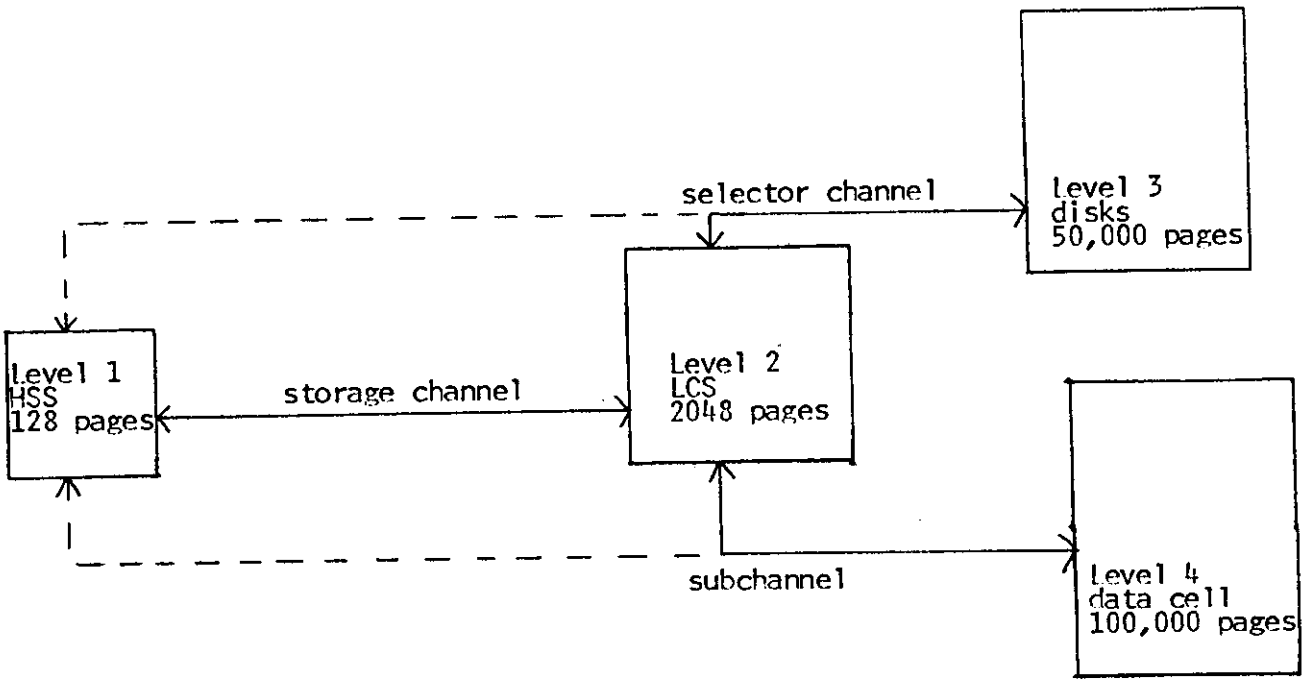

Figure 2b. Storage Organization of the CMU 360/67 Configuration

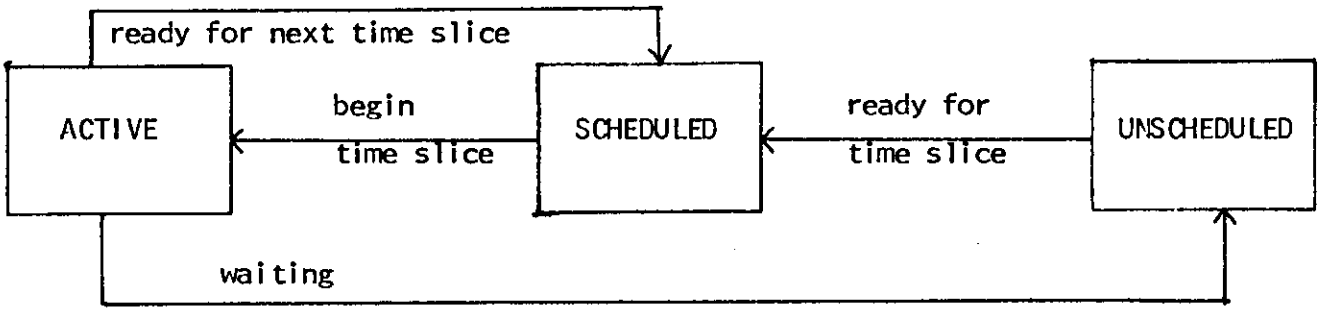

Figure 3. Task Movement Between States

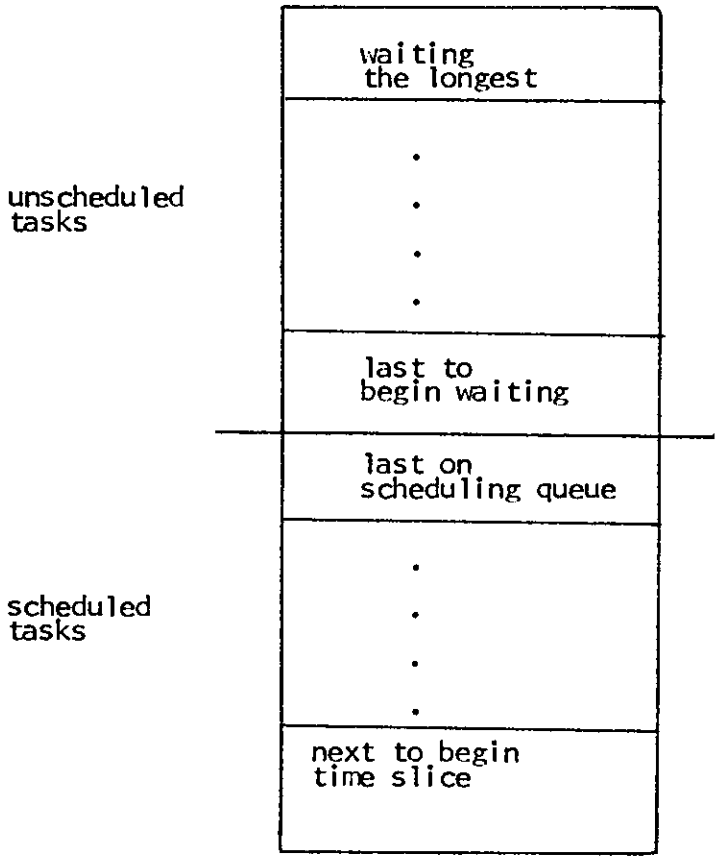

Figure 4. Core Release Chain

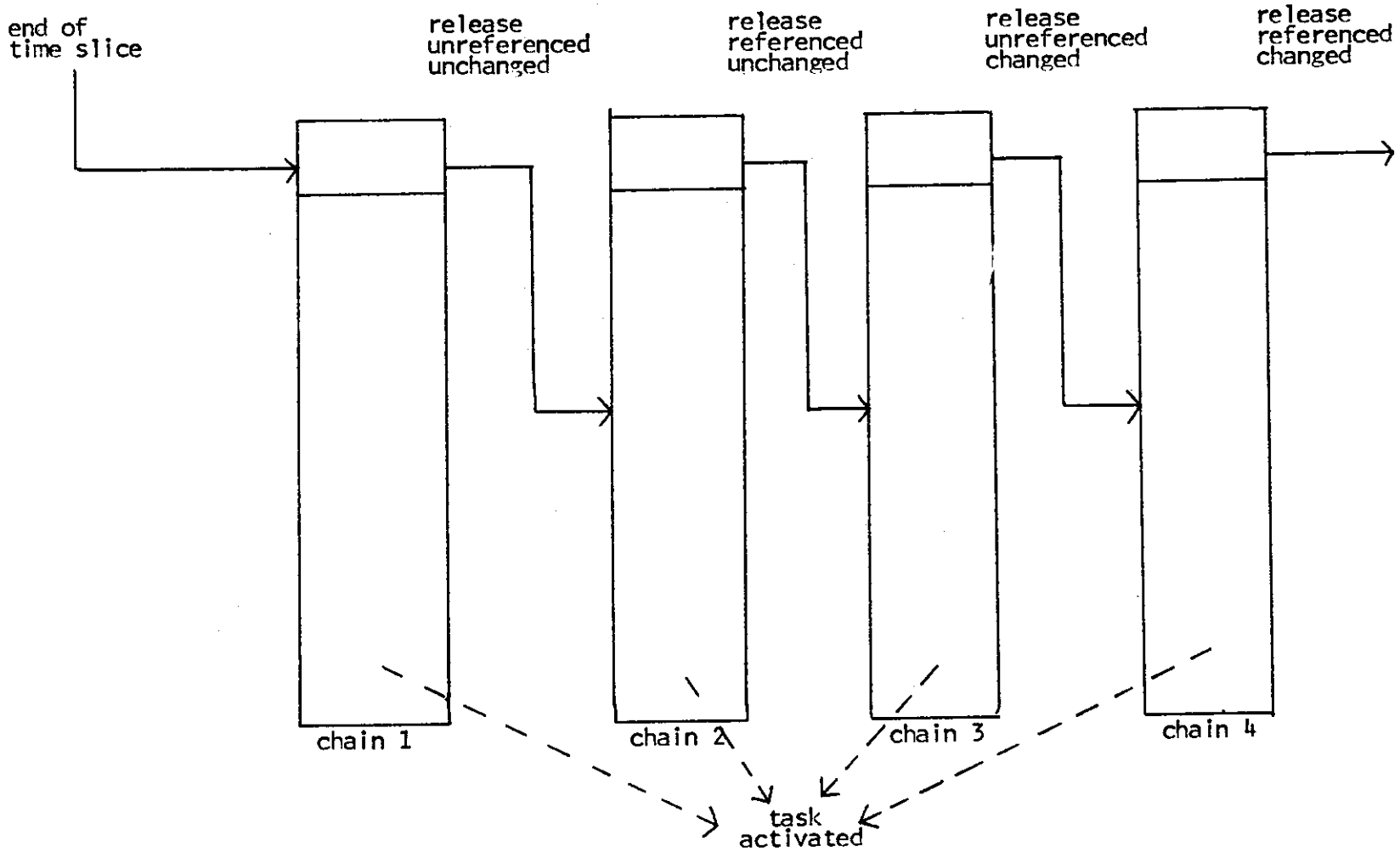

Figure 5. Task Movement in the Core Release Chains

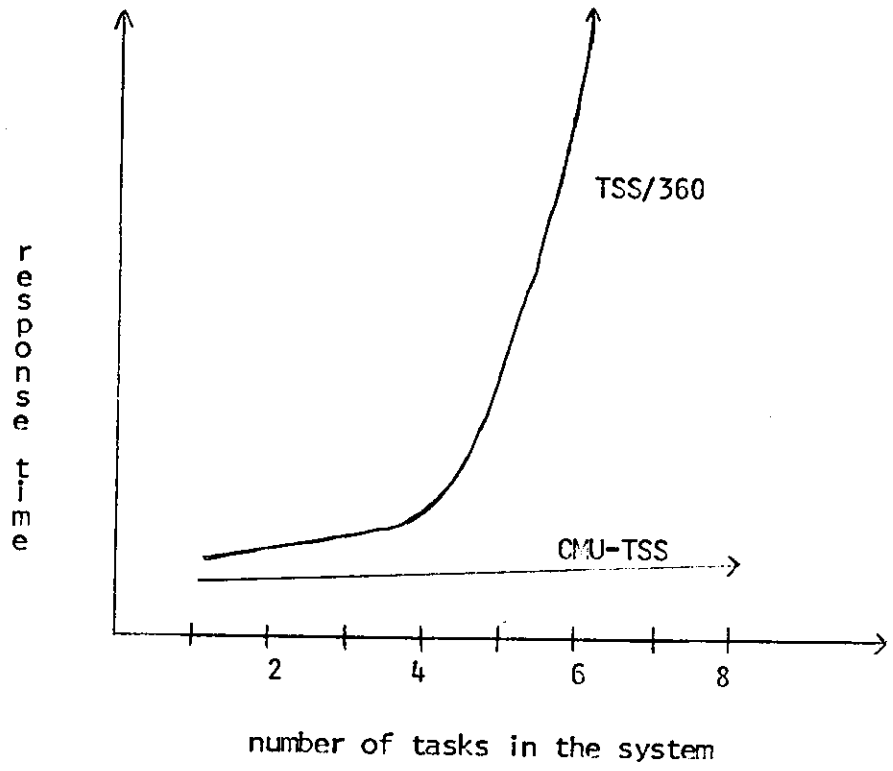

Figure 6. Response Times From The Experiment

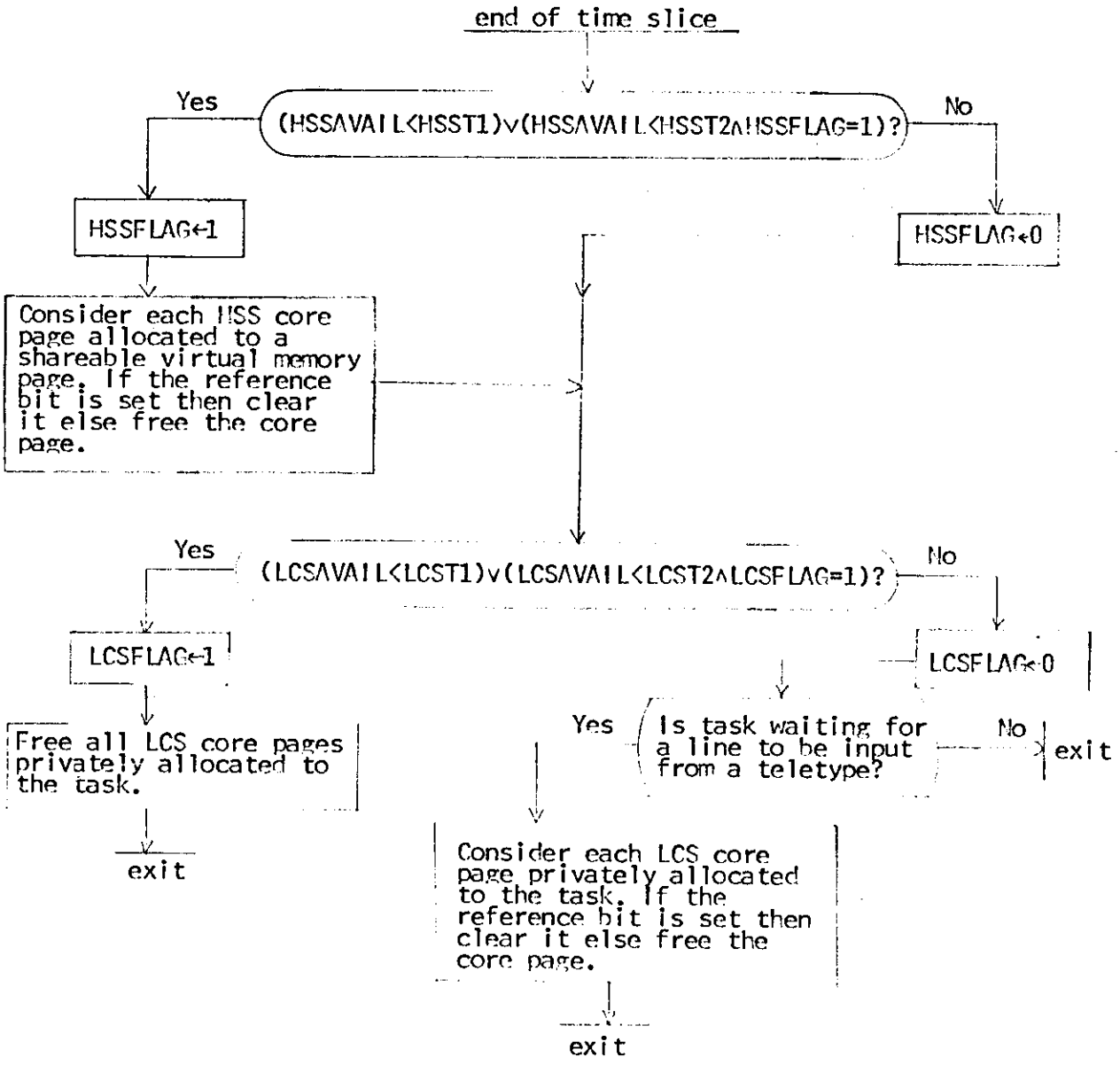

Figure 7. Core Release Algorithm Implemented for the Experiment

References

1. Corbato, F. J., et al, The Compatible Time-Sharing System, Project MAC User's Manual.

2. Schwartz, J. I., Coffman, E. G., and Weissman, C., "A General Purpose Time-Sharing System," Proc. Spring Joint Computer Conference, 1964.

3. Shaw, J. C., "Joss: A Designer's View of an Experimental On-line Computing System," Proceedings Fall Joint Computer Conference, 1964.

4. Lochner, Kenneth M., "The Evolving Time-Sharing System at Dartmouth College," Computers and Automation, Sept. 1965.

5. Allen-Babcock Corporation, The RUSH System, System Manual.

6. Control Data Corporation, SCOPE II for the CDC 6400/6600, System Manual.

7. General Electric Corporation, GE Comprehensive Operating System, System Manual.

8. IBM Corporation, OS/360: Concepts and Facilities, System Manual SRL #C28-6535.

9. IBM Corporation, TSS/360: Concepts and Facilities System Manual SRL #C28-2003.

10. Corbato, F. J., et al, A New Remote Accessed Man-Machine System, Session 6, Proc. Fall Joint Computer Conference, 1965.

11. Mendelson, M. J., and England, A. W., "The SDS SIGMA 7: A Real-Time, Time-Sharing Computer," Proc. Fall Joint Computer Conference, 1966.

12. Corbato, F. J., et al, A New Remote Accessed Man-Machine System, Session 6, Proc. Fall Joint Computer Conference, 1965.

13. Lauer, H. C., "Bulk Core in a 360/67 Time-Sharing System," Proc. Fall Joint Computer Conference 1967.

14. Chen, Y. C., and Hsieh, S. C., "Selective Transfer Analysis," IBM Research Report RC 1926, October 25, 1967.

15. MULTICS Systems Programmers' Manual, Section BG. 6, Project MAC, Massachusetts Institute of Technology, 1966.

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
Carnegie-Mellon University Department of Computer Science Pittsburgh, Pennsylvania 15213	UNCL
	2b. GROUP

3. REPORT TITLE

STEPS TOWARD A GENERAL PURPOSE TIME-SHARING SYSTEM USING LARGE CAPACITY CORE STORAGE AND TSS/360

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

5. AUTHOR(S) (First name, middle initial, last name)

Fikes, Richard E, Hugh C. Lauer, and Albin L. Vareha, Jr.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
March 1968	36	15

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
SD-146 ARPA	
b. PROJECT NO. 9718	
c. 6154501R	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d. 681304	

10. DISTRIBUTION STATEMENT

--Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Air Force Office of Scientific Research (SRI) 1400 Wilson Boulevard Arlington, Virginia 22209

13. ABSTRACT

This paper is a progress report of an effort at Carnegie-Mellon University to determine how a large capacity core storage facility (LCS) can be used to reduce the demand paging overhead costs in the IBM System/360 Time Sharing System (TSS/360) and in similar general purpose time-sharing systems. A discussion is presented of how the number of paging operations and the cost of each paging operation can be reduced by using LCS as both a swapping device and an extension of executable core. Two problems which arise are considered. One is the design of an effective method for determining whether pages should execute in HSS or in LCS. An analysis of this problem is presented and possible solutions are discussed. The second problem is the development of efficient core management algorithms for LCS. A new algorithm for releasing core is presented and compared with two existing algorithms. Finally, results from a feasibility implementation of the ideas in a pre-release version of TSS/360 are presented as a demonstration of the validity of using LCS to reduce paging overhead.

DD FORM 1 NOV 65 1473

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT