Discussion Paper No. 112

MANAGEMENT OF OPERATING SYSTEM SUBPROGRAMS

by

Jair M. Babad,[*] V. Balachandran,[**] and
Edward A. Stohr[**]

October 1974

[*] Graduate School of Business, University of Chicago

[**] Graduate School of Management, Northwestern University

Abstract

This paper is concerned with the problem of optimally assigning operating system programs to different storage devices such as core, drum, or disc. We assume that each program can be stored on the devices either as one page stored completely on one device or as a number of pages stored possibly on different devices. Given the costs of storage and accessing for the storage devices, and a constraint set representing time and storage requirements, a solution is found by mathematical programming techniques. Finally, the procedures developed in the paper are demonstrated using actual data for some of the system library routines at the Northwestern University Computing Center.

Key Words: OPTIMIZATION, COMPUTER SCIENCE, OPERATING SYSTEMS.

I. Introduction

The management of a modern computer system is a complicated operation, and special programs, called operating systems, have been designed and implemented for this task. An operating system is a collection of many subprograms, each of which is responsible for one or more specific operations such as input-output, accounting, job scheduling, resource management, source language translation, loading of job modules, and so forth. Some of the most frequently used of these subprograms may be maintained within the core memory of the computer. However, due to the limited size of the computer's main memory, many of the operating system's subprograms are stored on auxiliary devices such as drums or discs, and are moved into the core memory only when they are needed. The determination of the storage device on which a subprogram is to be stored is based on the frequency of use, the storage requirements and the transfer rate of information from various storage devices into the main core. These decisions are usually made according to some heuristic procedures, due to the lack of appropriate analytic models. Their impact on the computer system's performance, though, might be very noticeable, and improper decisions might degrade performance considerably. In order to remedy this situation we present here an appropriate model for the storage management of operating system subprograms. In this model we take into account the various cost elements that are related to the usage of such subprograms, and, subject to access time and storage constraints, we seek the decisions which minimize the total expected cost. We also provide an example of an application of our model to a large scale operating system.

The present paper represents an application of the model found in Babad [1] and contains an extension to the case where subroutine calls are dependent on the prior history of jobs. The philosophy we adopt is also similar to that in Babad, Balachandran, and Stohr [2] in that our model attempts to find the most cost-effective solution for a given set of processing requirements. This contrasts with the approach adopted, for example, in Ramamoorthy and Chandy [9], where cost is held fixed and the objective is to allocate programs to devices to minimize processing time. Another point of difference with their work is that they are concerned with fixed-length program segments or "blocks" whereas we allow for variable-length segments. For similar reasons, our work also contrasts with the approach of Grossman and Silverman [6], who deal with the problem of placing records on a disc in order to minimize the average access time.

The approach we adopted in this paper is to allocate subprograms to various storage devices with the objective of minimizing the subprograms' access and transfer costs. This allocation results in a storage hierarchy and is related to earlier work on memory hierarchies, e.g., Salasin [10], Ramamoorthy and Chandy [9], Mattson et al. [8], and Gecsei and Lukes [4]. However, we concentrate on the optimal allocation of the programs to the various storage devices rather than the selection of an optimal storage structure.

Our discussion starts with the case where program calls are independent of the prior history of the process ("independent subprograms"). In Section 2 we present the general model and detailed notation. Several simplified versions of the general model are discussed in Section 3. In Section 4 the model is extended to allow for dependent subprograms. The application of our model to an actual operating system is described in Section 5, while in Section 6 we discuss the possibility of future applications and enhancements of the model.

2. Cost Analysis of Operating Systems

In the cost analysis that is presented in this section, we are concerned with the storage and access costs of an operating system and ignore development and fixed costs. Specifically, the objective will be to minimize the total expected costs of the system by a proper selection of storage devices for storing the system's subprograms. Several other interpretations of the model for different objectives will be discussed later in Section 5.

Let R be the number of subprograms in the operating system, and let M_i be the length of the ith one $(1 \leq i \leq R)$. Since an operating system is not subjected to many frequent changes, we might consider R and the M_i's as fixed exogenous data. These subprograms are used and called by K different types of user jobs, such as compilers, job accounting, statistical processing, etc. However, the subprograms are assumed to be independent in the sense that one subprogram may not call another one. With each job-type k we associate a value factor $v_k (1 \leq k \leq K)$, which takes into account the relative importance of the job and the number of times it is processed per time period. In the simplest case, v_k will be the relative frequency of the job times the total number of processed jobs during one time period. In addition, we associate with each job-type k a selection set S_k, which is the subset of the system's subprograms that are requested by this job-type.

The operating system's subprograms may be stored on various storage devices, indexed by $j = 0, 1, \ldots, J$. We assume that this indexing is according to the transfer rate of data from the storage devices into the core memory. Thus the core memory is indexed as the zero device, a drum might be the first device, and a disc, say, might be the second device. Storage of a unit of data on device j costs the system C_j money units per time period, while the cost of accessing and reading data from device j into core includes two elements: a fixed cost A_j and a transfer cost (per storage unit transferred) of B_j. For example, in a random access disc device A_j is the cost of the time required to calculate the address of a needed subprogram, to move the arm to this address, and to wait for the rotational delay of the disc. Due to our indexing order, the costs A_j and B_j will (usually) be nondecreasing functions of j, with $A_0 = B_0 = 0$ (since the information on device 0 is already in the core memory). Similarly, the costs C_j are a nonincreasing function of j, because the fast devices are more limited in their storage capacity and cost more to purchase and operate than the slower devices.

As in Babad [1], we initially consider the following general storage scheme for the operating system's subprograms: A subprogram i, say, is broken into many nonoverlapping segments, denoted by $b_{ij} (0 \leq j \leq J)$, where segment b_{ij} is stored on the jth device, and $\Sigma_{j=0}^{j=J} b_{ij} = M_i$. Due to the stability of the operating system, this storage scheme is static rather than dynamic; that is, once the segments are determined, the storage assignments are not to be changed unless a (considerable) change in the job mix or the operating system's specifications occurs.

When a job-type k, say, is processed by the system, all the subprograms in its selection set S_k have to be accessible in the core memory. The operating system thus checks a directory of its subprograms, determines where their segments are stored, and transfers these segments from the various storage devices into the core memory. Suppose i is one of the subprograms in S_k. If $b_{ij} = 0$, no segment of i is stored on device j, and this device is not accessed for subprogram i. On the other hand, if $b_{ij} > 0$ the jth storage device has to be accessed for subprogram i, with access cost of $A_j + B_j b_{ij}$. Let

$$\delta_{ij} = \begin{cases} 1 & \text{if } b_{ij} = 0 \\ 0 & \text{if } b_{ij} > 0 \,. \end{cases} \tag{1}$$

The total access cost of subprogram i is then

$$AC_i = \Sigma_{j=0}^{j=J} (1 - \delta_{ij})(A_j + B_j b_{ij}) \tag{2}$$

and the total access cost of job-type k, when processed once, is

$$TC_k = \Sigma_{i \in S_k} AC_i = \Sigma_{i \in S_k} \Sigma_{j=0}^{j=J} (1 - \delta_{ij})(A_j + B_j b_{ij}) \,. \tag{3}$$

Taking into account the value factors of the various job-types, we get as the total expected access cost during a time period:

$$TAC = \Sigma_{k=1}^{k=K} v_k TC_k = \Sigma_{k=1}^{k=K} v_k \Sigma_{i \in S_k} \Sigma_{j=0}^{j=J} (1 - \delta_{ij})(A_j + B_j b_{ij}) \,. \tag{4}$$

Consider now the storage costs. On storage device $j (0 \leq j \leq J)$ are stored the segments b_{ij} of all the subprograms $1 \leq i \leq R$. Thus the total storage on device j which is devoted to the operating system's subprograms is:

$$L_j = \Sigma_{i=1}^{i=R} b_{ij} \tag{5}$$

and the total storage cost per time period is:

$$TSC = \Sigma_{j=0}^{j=J} C_j L_j = \Sigma_{j=0}^{j=J} C_j \Sigma_{i=1}^{i=R} b_{ij} \,. \tag{6}$$

The total operating cost of the operating system during a time period is thus

$$T = TAC + TSC \tag{7}$$

with TAC and TSC given by (4) and (6), respectively. This cost is clearly a function of the specific storage scheme used, as given by the b_{ij}'s. Our aim is to minimize T, subject to the subprogram constraints:

$$\Sigma_{j=0}^{j=J} b_{ij} = M_i \qquad \text{for} \quad i \leq i \leq R \tag{8}$$

$$b_{ij} \geq 0 \qquad \text{for} \quad 1 \leq i \leq R \quad \text{and} \quad 0 \leq j \leq J \tag{9}$$

and b_{ij} integer (since storage is measured in storage units). Also, from (1) we have

$$b_{ij} + \delta_{ij} \geq 1 \quad \text{for} \quad 1 \leq i \leq R \quad \text{and} \quad 0 \leq j \leq J. \tag{10}$$

Additional constraints might arise from limitations on available storage. Specifically, suppose that the total available storage on device j is $D_j \geq 0$. Thus,

$$L_j = \Sigma_{i=1}^{i=R} b_{ij} \leq D_j \qquad \text{for} \quad 0 \leq j \leq J \tag{11}$$

where, to be consistent, we set D_j to be suitably large if the available storage on device j is not restricted; e.g., in such a case we might set $D_j = \Sigma_{i=1}^{i=R} M_i$.

Further, the requirements of the operating system might dictate a short access time for some subprograms; for example, interruption handlers and real-time communication subprograms are restricted in their access time. So let f_j be the fixed time required to access device j, and t_j be the transfer time (per storage unit transferred) from storage device j into core memory. The access time constraints are derived in a similar way to the derivation of the access costs, and, denoting by T_i (≥ 0) the maximal allowed access time for subprogram i, we have

$$\Sigma_{j=0}^{j=J} (1 - \delta_{ij})(f_j + t_j b_{ij}) \leq T_i$$

or

$$\Sigma_{j=0}^{j=J} (t_j b_{ij} - f_j \delta_{ij}) \leq T_i - \Sigma_{j=0}^{j=J} f_j \quad \text{for} \quad 1 \leq i \leq R \tag{12}$$

since $(1 - \delta_{ij})b_{ij} = b_{ij}$ by (1). Clearly, for nonbinding time constraints we might set T_i to be very large, e.g.,

$$T_i = \Sigma_{j=0}^{j=J} (f_j + t_j M_i).$$

Our problem is thus to find a storage scheme--as given by the b_{ij}'s--which will minimize T as given by (7), subject to the constraints (8)-(12). However, some simplifications can be introduced in (7). Specifically, let

$$\epsilon_{ik} = \begin{cases} 1 & \text{if } i \in S_k \\ 0 & \text{otherwise} \end{cases} \quad \text{for} \quad 1 \leq i \leq R \quad \text{and} \quad 1 \leq k \leq K\,. \tag{13}$$

Equation (7) then becomes

$$T = \Sigma_{j=0}^{j=J} C_j \Sigma_{i=1}^{i=R} b_{ij} + \Sigma_{k=1}^{k=K} v_k \Sigma_{i=1}^{i=R} \epsilon_{ik} \Sigma_{j=0}^{j=J} (1 - \delta_{ij})(A_j + B_j b_{ij}) \tag{14}$$

$$= \Sigma_{j=0}^{j=J} \Sigma_{i=1}^{i=R} \left[C_j b_{ij} + (1 - \delta_{ij}) A_j \Sigma_{k=1}^{k=K} v_k \epsilon_{ik} + (1 - \delta_{ij}) b_{ij} B_j \Sigma_{k=1}^{k=K} v_k \epsilon_{ij})\right].$$

But $(1 - \delta_{ij})b_{ij} = b_{ij}$ by (1), and $\Sigma_{k=1}^{k=K} v_k \epsilon_{ik}$ is a constant determined by the data of the problem. Let then:

$$V_i = \Sigma_{k=1}^{k=K} v_k \epsilon_{ik} \quad \text{for} \quad 1 \leq i \leq R. \tag{15}$$

Ignoring the constant term $\Sigma_{j=0}^{j=J} \Sigma_{i=1}^{i=R} A_j V_i$ in (14) which does not affect the optimization, we finally get the following integer programming formulation of our problem:

$$\left.\begin{array}{ll}
\text{Minimize} \quad T = \Sigma_{j=0}^{j=J} \Sigma_{i=1}^{i=R} [(C_j + B_j V_i) b_{ij} - A_j V_i \delta_{ij}] & \\
\text{subject to:} \quad \Sigma_{j=0}^{j=J} b_{ij} = M_i & \text{for} \quad 1 \leq i \leq R \\
\Sigma_{i=1}^{i=R} b_{ij} \leq D_j & \text{for} \quad 0 \leq j \leq J \\
\Sigma_{j=0}^{j=J} (t_j b_{ij} - f_j \delta_{ij}) \leq T_i - \Sigma_{j=0}^{j=J} f_j & \text{for} \quad 1 \leq i \leq R \\
b_{ij} + \delta_{ij} \geq 1 & \text{for} \quad 1 \leq i \leq R \text{ and } 0 \leq j \leq J \\
b_{ij} \geq 0, \text{ integer}, \ \delta_{ij} = 0 \text{ or } 1 \text{ for } 1 \leq i \leq R \text{ and} & \\
\qquad 0 \leq j \leq J. &
\end{array}\right\} \tag{16}$$

Notice that in this formulation we effectively eliminated the various job-types, which only enter the formulation implicitly via the V_i's.

3. Nonsegmented Programs

Many computing systems require that each subprogram be entirely stored on one storage device. The main reasons for such a requirement are the reduction in the subprogram directory size (with the associated reduction in directory search overhead) and the resulting simplicity of maintenance of the operating system. In this case, it seems appropriate to replace the formulation of (16) by a simpler representation of the problem. Let then:

$$x_{ij} = \begin{cases} 1 & \text{if} \quad b_{ij} = M_i \\ 0 & \text{if} \quad b_{ij} = 0 \end{cases} \quad \text{for} \quad 1 \leq i \leq R \text{ and } 0 \leq j \leq J. \tag{17}$$

Thus in this case $x_{ij} = 1 - \delta_{ij}$, b_{ij} equals $M_i x_{ij}$, and from (16) we get the following integer programming formulation:

$$\text{Minimize:} \quad T = \Sigma_{j=0}^{j=J} \Sigma_{i=1}^{i=R} U_{ij} x_{ij} \tag{18-i}$$

subject to:

$$\Sigma_{j=0}^{j=J} x_{ij} = 1 \qquad \text{for} \quad 1 \leq i \leq R \tag{18-ii}$$

$$\Sigma_{i=1}^{i=R} M_i x_{ij} \leq D_j \qquad \text{for} \quad 0 \leq j \leq J \tag{18-iii}$$

$$\Sigma_{j=0}^{j=J} (f_j + t_j M_i) x_{ij} \leq T_i \qquad \text{for} \quad 1 \leq i \leq R \tag{18-iv}$$

$$x_{ij} \geq 0, \text{ integer} \quad \text{for} \quad 1 \leq i \leq R \text{ and } 0 \leq j \leq J \tag{18-v}$$

where $U_{ij} = (C_j + B_j V_i) M_i + A_j V_i$ is the cost per time unit of storing and accessing subprogram i on device j.

In particular, consider the very common case in which the storage hierarchy consists of the core memory and a drum or disc--i.e., when only two levels of storage hierarchy are allowed. In this case $J = 1$, and we can replace the double indexing of (18) by a single index. Specifically, let:

$$y_i = x_{i0}. \tag{19}$$

The objective function is then:

$$\Sigma_{i=1}^{i=R} (U_{i0} y_i + (1 - y_i) U_{i1})$$

$$= \Sigma_{i=1}^{i=R} U_{i1} + \Sigma_{i=1}^{i=R} (U_{i0} - U_{i1}) y_i.$$

Ignoring the first term, which is a constant and does not affect the optimization, we have the formulation:

$$\left.\begin{array}{ll} \text{Minimize: } & \Sigma_{i=1}^{i=R} (U_{i0} - U_{i1})y_i \\ \text{subject to: } & \Sigma_{i=1}^{i=R} M_i y_i \leq D_0 \\ & E_{i=1}^{i=R} M_i(1 - y_i) \leq D_1 \\ & [(f_0 + t_0 M_i) - (f_1 + t_1 M_i)]y_i \leq T_i - (f_1 + t_1 M_i) \\ & \qquad\qquad\qquad \text{for } \ 1 \leq i \leq R \\ & y_i = 0 \ \text{ or } \ 1 \qquad\qquad \text{for } \ 1 \leq i \leq R. \end{array}\right\} \quad (20)$$

In this case, though, some simple observations may considerably reduce the size of the problem. First, we notice that if an access constraint is binding for a particular value of i, then subprogram i has always to remain in core. Thus, all subprograms for which these constraints are binding are immediately assigned to the core memory--and deleted from the problem. The remaining constraints of this type can also be dropped from the formulation since they are nonbinding. Second, it might be assumed, as is usually the case, that the capacity of secondary storage (drum or disc) will not be binding, so that:

$$\Sigma_{i=1}^{i=R} M_i \leq D_1.$$

The problem is then reduced into a knapsack problem with R items, where item i has value $u_{i0} - u_{i1}$, length M_i, and the available space is D_0. Many algorithms (e.g., Gilmore and Gomory [5]) are available for this standard problem.

4. Extension to Dependent Subprograms

One of the main assumptions of the preceding development was the independence of the system's subprogram. Under this assumption, no inter-relations exist between subprograms, and one subprogram may not call another. This is clearly not the case for many systems, and in this section that assumption is relaxed. Specifically, we assume that when subprogram i is executed for jobs of type k, it may call any other subprogram, say j. Subprogram j, in turn, may call another subprogram, and so forth. Further, we assume that if the called subprogram is not stored in the core memory, it will be brought in from the storage device on which it is stored (this assumption is justified as an approximation, since otherwise the model must be expanded--and complicated--by the introduction of memory directories, cost of searching such directories, the "process history," etc.)

Let us then denote by $p_{in}(k)$ the probability that subprogram n will be called by subprogram i when a job of type k is processed, and consider a (fixed) storage scheme as given by a set of b_{ij}'s. The total operating cost with dependencies will differ from the previously considered cost due to the increase in accessing costs. We thus have to modify TAC of equation (4) with modified accessing costs. But the access costs are separable into job-types costs (TC_k), and therefore it is enough to consider here a single job-type, say k. As is clear from the contexts and equation (3), if a subprogram i, say, is independent with respect to job-type k, then the modified cost for accessing it equals the cost earlier considered in (3). Such a program satisfies

$$\Sigma_{n=1}^{n=R} P_{in}(k) = 0$$

and might be ignored in the following discussion. Thus, without loss of generality, we assume that for each subprogram $i \in S_k$, we have:

$$0 < \Sigma_{n=1}^{n=R} p_{in}(k) < 1. \tag{21}$$

Notice that the probabilities in (21) do not sum up to 1, since each subprogram in S_k might be terminated without calling in any other subprogram (otherwise, if i always calls j, we may combine i and j into one subprogram).

In the following discussion it will be convenient to adopt matrix notation. Thus, let:

$$\left.\begin{array}{ll} \underline{S}_k^0 = & \text{row vector with } R \text{ elements, whose ith element} \\ & \underline{S}_k^0(i) \text{ equals 1 if } i \in S_k, \text{ and is zero otherwise.} \\ \underline{AC} = & \text{column vector with R elements, whose ith element is} \\ & AC_i \text{ as given by (2).} \\ I = & \text{an identity matrix of size } R \times R. \\ P_k = & \text{the matrix whose } (i, n) \text{ element is } p_{in}(k). \end{array}\right\} \tag{22}$$

Using this notation, the total access cost for job-type k of equation (4) is given (for independent programs) by

$$TC_k = \underline{S}_k^0 \; I \; \underline{AC}. \tag{23}$$

However, the programs are not independent. First, the programs selected by $\underline{S}_k^0$ are called into core memory. Each of these might, in turn, call some other programs, and the expected number of calls of program n is given by:

$$\Sigma_{i=1}^{i=R} \; \underline{S}_k^0(i) \; p_{in}(k) \qquad 1 \leq n \leq R.$$

Thus,

$$\underline{S}_k^1 = \underline{S}_k^0 P \tag{24}$$

is the vector whose ith element specifies the expected calls of subprogram i by the system's programs which were initially used during the processing of job-type k. Similarly,

$$\underline{S}_k^2 = \underline{S}_k^1 P$$

specifies the expected number of subprogram selections for processing, due to calls by programs chosen by $\underline{S}_k^1$. Thus, the total expected number of subprograms' calls for processing of job-type k is given by:

$$\left.\begin{aligned} \underline{S}_k^\infty &= \underline{S}_k^0 + \underline{S}_k^1 + \underline{S}_k^2 + \ldots \\ &= \underline{S}_k^0 I + \underline{S}_k^0 P + \underline{S}_k^0 P^2 + \ldots \\ &= \underline{S}_k^0 (I + P + P^2 + \ldots) \\ &= \underline{S}_k^0 (I - P)^{-1}. \end{aligned}\right\} \tag{25}$$

Notice that the infinite matrix sum in (25) converges because the norm of the P matrix is less than 1 due to (21). The modified access cost is then given by:

$$\begin{aligned} TC_k &= \underline{S}_k^\infty I \, \underline{AC} \\ &= \underline{S}_k^0 (I - P)^{-1} \underline{AC} \, . \end{aligned} \tag{26}$$

In view of (25) and (26) we can reinterpret ϵ_{ik}. Thus, we redefine ϵ_{ik} as the expected number of calls to subprogram i during the processing of job-type k. ϵ_{ik} is then 0 if $i \notin S_k$, 1 if $i \in S_k$ and is independent, and the ith elment of $\underline{S}_k^\infty$ otherwise. The model, as given by (14)-(16), is then valid without any further changes.

5. Application and Results

In order to gain experience with the use of the theory presented in the first four sections, a pilot study has been carried out using actual data for the operating system of the CDC6400 at the Vogelback Computing Center, Northwestern University. The present system configuration at Vogelback consists of a CDC6400 Computer and associated peripheral equipment. It provides time sharing and batch processing services for educational and research purposes. The 6400 system has a large central processor (CP) with a 65,536 60-bit word memory (CM) and ten associated peripheral processors (PP's), each having a 4,096 12-bit word memory. The major function of the CP is to perform the computational tasks required by user programs. The PP's act as independent processors, are responsible for the overall management of the system, in particular the management of user programs. They are also responsible for the transmission of data between the various I/O devices and CM, and between the disc and magnetic tape units and CM. The flows of information and control in the CDC 6400 system are shown in Figure 1.

The auxilliary storage devices include "Extended Core Storage" (ECS) with a capacity equivalent to 251,904 CM words, 3 CDC 844-21 disc pack units with a total storage capacity equal to approximately 30 million CM words, and a CDC 6603-II disc unit of 7.5 million CM words. The ECS is a large ferrite core memory which communicates only with the CM. At present it contains the highest priority system information (including many of the PP subprograms), time-sharing user jobs, and some user temporary files. The 844-disc packs contain the less frequently used PP programs, the remaining "libraries" of operating system programs, and user permanent files. The 6603 disc unit contains lower priority information and will not be discussed further here.

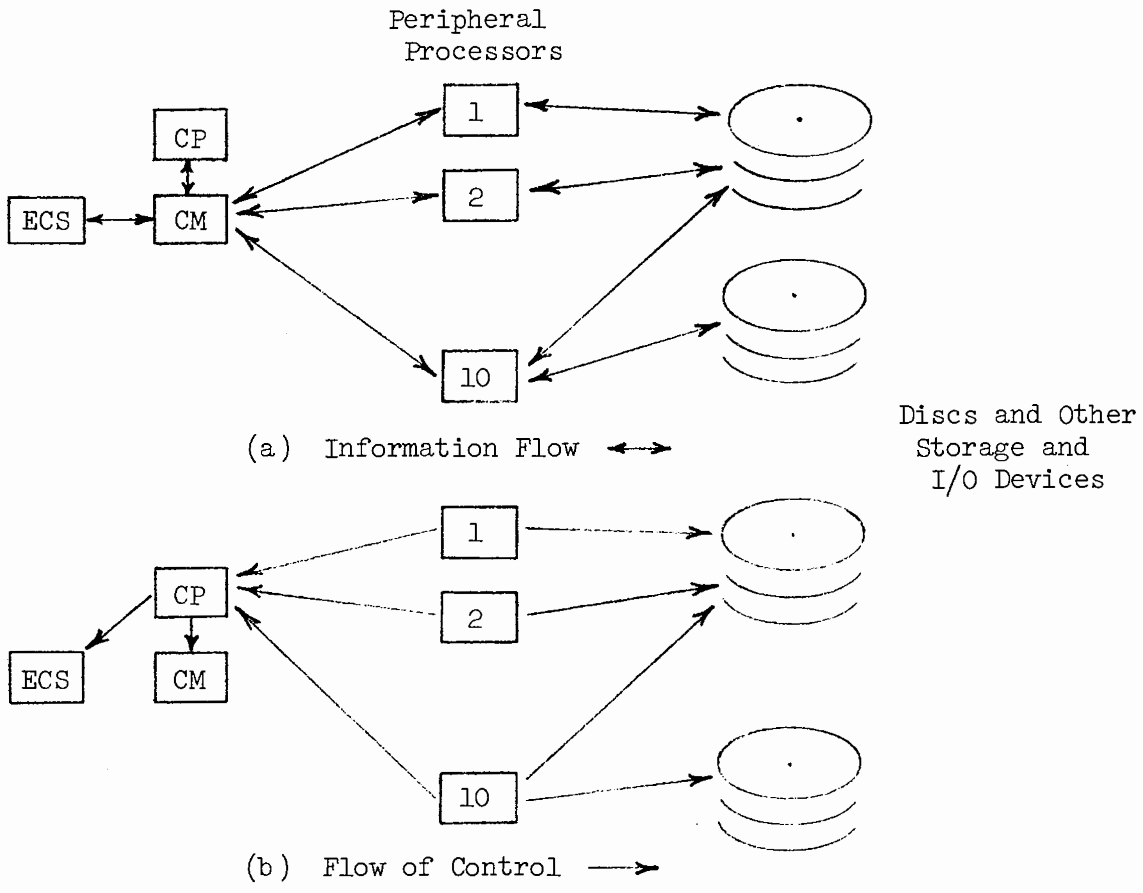

Fig. 1.--CDC 6400 System--Flows of Information and Control

Requests by a user program for system programs or data files are first referred to a PP which identifies the location of the desired information. If the information is stored on disc, the PP will arrange for its transmission to CM. If the desired information is stored in ECS, control is returned to the CP, which then arranges for the transfer of the data to CM.

The theory presented earlier can be used to investigate the storage assignment of both PP programs (presently residing on ECS and disc) and other system programs, including the libraries of compiler subroutines (presently residing only on the disc). Data on the frequency of use of the PP programs is not presently available, so the study was carried out for one of the system program "libraries." The chosen library contained 116 subroutines which are used by the RUN Fortran compiler. Some of these subprograms are used for input, output, and control and are called every time a RUN-compiled program is executed. Other subroutines, such as SQRT and TAN, are part of the regular FØRTRAN library of subprograms and have less frequent usage. The most frequently used subroutines in the library are executed an average of once every 77 seconds. The subprogram lengths vary from 10 to 1,032 words.

Specifically, our analysis was concerned with the consequences of allowing some of the RUN library subroutines presently stored on disc to be stored in either CM or ECS. Since the available data concerned the frequency of use of individual subroutines (rather than jobs as we have defined them), the analysis in Section 4 concerning "dependent" subprograms could be bypassed without loss. Thus, our problem was in the form of equation (18) with $J = 2$. Values of $D_0 = 200$ and $D_1 = 2,000$ words were chosen as desirable upper bounds on the amount of CM and ECS, respectively, which could be allocated to the RUN subroutines without significant effects on other aspects of system performance. An analysis was also carried out with $D_0 = 400$ and $D_1 = 4,000$ to test the sensitivity of the solution to these constraints. No capacity limit was imposed for the disc storage. Since all the programs are presently resident on the slowest of the three access devices, it was not considered necessary to include any of the timing constraints for individual programs.

In deriving the cost data for the problem, an attempt was made to charge the system subprograms at the same rate as user programs. In this way, system subprograms will be allocated to higher performance storage devices if, roughly speaking, the economic benefits exceed the costs. The Vogelback Computation Center uses the following formula for computing job cost:

$$JC = P\left[7 + 3\frac{M_J}{M_{CM}}\right]\left[T_{CP} + .2\left(1 + \frac{M_J}{M_{CM}}\right)T_{PP}\right],$$

where P is a priority factor ($P = 1$ for most jobs), M_{CM} is the capacity (in words) of the CM, M_J is the total amount of CM required for the job, and T_{CP} and T_{PP} are the respective amounts of CP and PP time used (in minutes). For each device, j, access costs of h_j dollars per second and storage costs of c_j dollars per word per second were computed which were consistent with actual user costs (as given by the formula above) to run a "typical" source program job and to store information in auxiliary storage devices. The resulting values for the A_j's, B_j's, and C_j's are shown in Table 1. Also shown in the table are the access times and transmission rates for the various system components. The access times, f_j, allow for an "average" amount of software overhead in addition to the hardware-related access delays.

TABLE 1

DATA FOR CDC 6400 SYSTEMS

	Device		
	CM	ECS	844-Disc
j	0	1	2
f_j (seconds)	1×10^{-6}	27.1×10^{-6}	32.3×10^{-3}
t_j (seconds per word)	0	$.2 \times 10^{-6}$	13.85×10^{-6}
A_j (\$)	1.29×10^{-7}	2.26×10^{-6}	6.07×10^{-4}
B_j (\$ per word)	0	1.67×10^{-8}	2.60×10^{-7}
C_j (\$ per word per second)	2.41×10^{-6}	3.01×10^{-7}	3.62×10^{-10}

Data on the frequency of use of the subprograms was collected over a period of approximately two months. Only 100 of the 116 subprograms in the library were used during this period, and so the analysis was carried out for these subprograms. The resulting problem was a zero-one integer program with 102 constraints and 300 variables.

The solution of such a large integer programming problem would normally be a formidable task. However, if we ignore constraint set (18-iv), the problem can be transformed to an ordinary transportation problem by defining new decision variables, $Y_{ij} = M_i X_{ij}$. The optimal tableau for this problem will include a "dummy" program for the inequality constraints (18-iii) and will have $R + J$ cells in the basis (where, as before, R is the number of programs and J is the number of devices). Let the rows of the tableau correspond to programs. Since each row, i, should have a cell in the basis, at most $J - 1$ rows may have more than one cell in the basis. This shows that at most $J - 1$ programs may be assigned to two or more devices. Efficient algorithms are available to restore the restriction that there should be only one basic cell in each row (i.e., that programs may not be assigned to more than one device), (see Balachandran [3]). Alternatively, an approximate solution to problem (18) (without (18-iv)) can be obtained by neglecting the integer restraint and solving the resulting linear program. Again, at most $J - 1$ programs will be assigned to more than one device. Since the ratio of the number of programs to the number of devices will usually be very large, the linear programming solution should, intuitively, be very close to the optimal integer solution. The resulting non-integer solution can be adjusted by hand, or, if necessary, the exact integer solution can be obtained by running additional linear programming problems and employing a "branch-and-bound" technique (see Land and Doig [7]). Moreover, in some cases the value of the capacity limits, D_j, may be flexible, and a feasible integer solution might be obtained by this means.

Another property of (18) (without(18-iv)) is that if subprogram i is allocated to storage device j rather than to $j + 1$, the cost-saving is a decreasing function of the ratio $r_{ij} = \frac{U_{i,j} - U_{i,j+1}}{M_i}$. Furthermore, program i should not be assigned to device j if $r_{ij} > 0$. This provides a convenient heuristic procedure for allocating subprograms to devices which is similar to the ranking technique employed during the solution of knapsack problems [5]. The procedure is as follows:

(0) Construct a list in which the negative r_{ij}'s are ranked in increasing order of magnitude. Let $L_j = D_j$, $0 \leq j \leq J$.

(1) If the list is empty, stop. Otherwise, select i and j such that r_{ij} is at the head of the list.

(2) (a) If $M_i \leq L_j$, assign subprogram i to device j. Set $L_j = L_j - M_i$. Delete all r_{ij}'s, $0 \leq j \leq J$ from the list. Go to (1).

(b) If $M_i > L_j$, remove r_{ij} from the head of the list. Go to (1).

The approximate solutions obtained by linear programming methods as described below are consistent with this heuristic method.

The fact that efficient exact (and approximate) solution procedures are available for (18) is significant since they enable solutions to be obtained for actual operating systems which may involve several thousand subprograms.

In our study, three different objective functions were considered. The first objective was to minimize the expected processing times for the subroutines (a "throughput" criterion). In this case, the values of the storage costs, C_j, were set equal to zero, and the values of f_j and t_j were substituted for the costs A_j and B_j, respectively, in the computation of the U_{ij} coefficients in (18-i). The second objective which was studied was that of minimizing the direct costs to the users of the system. This was accomplished by using the costs A_j and B_j shown in Table 1 with the storage costs, C_j, again set to zero. Two runs were made using this objective, the first with capacity constraints $D_0 = 200$ and $D_1 = 2,000$, and the second with capacity constraints $D_0 = 400$ and $D_1 = 4,000$. The third objective considered was that of minmizing the total costs of the system. In this case, the U_{ij} coefficients were calculated using the values of A_j, B_j, and C_j given in Table 1.

The linear programming solutions for the four problems are summarized in Table 2.

TABLE 2

LINEAR PROGRAMMING SOLUTIONS

		Constraints		Number of Subprograms Allocated to Devices*			
Problem	Criterion	D_0	D_1	CM	ECS	Disc	Total
1	Maximum Throughput	200	2000	4	17	79	100
2	Minimum User Costs	200	2000	4	17	79	100
3	Minimum User Costs	400	4000	5	30	65	100
4	Minimum Total Costs	200	2000	0	1	99	100

*Adjusted to the nearest whole number.

In this experiment, the optimal allocation of subprograms to devices was the same for the maximum throughput (Problem 1) and minimum user costs (Problem 2) cases. Thus, the present pricing policy of the Vogelback Center is consistent with the goal of encouraging maximum throughput of jobs through the system. As expected from the nature of the objective functions, the constraints for CM and ECS capacity were binding for the first three problems. This was not so when the objective was to minimize total system costs. In this case, only one subprogram was assigned to ECS and all others to the disc. Thus, the present policy of assigning all of these subprograms to disc storage is very nearly optimal from the point of view of minimizing total system costs.

The savings for the optimal solutions listed above relative to the present allocation of subprograms are summarized in Table 3.

TABLE 3

SAVINGS RELATIVE TO PRESENT ALLOCATION OF SUBPROGRAMS TO DEVICES

Problem	Criterion	Constraints D_0	Constraints D_1	Percent Saving Relative to Current Solution
1	Maximum Throughput	200	2000	68.9
2	Minimum User Costs	200	2000	68.5
3	Minimum User Costs	400	4000	92.7
4	Minimum Total Costs	200	2000	12.3

These results show that very high relative savings may be obtained by shifting a small number of programs to the higher performance storage devices. Thus, the expected total time spent processing these subprograms can be reduced by approximately 70% if the optimal solution to problem 1 is adopted. If can be seen also that significant relative savings can be made if the capacity constraints are relaxed (compare problems 2 and 3). Although all relevant factors affecting system performance cannot be taken into account, it does seem reasonable to expect that substantial increases in system efficiency might be obtained if analytical procedures such as the above are employed in place of the current trial-and-error methods.

6. Conclusions

In this paper, we presented a model for the storage management of operating systems subprograms and the application of the model to a large-scale operating system. It was shown that the model objectives can be interpreted in several different ways, and thus can accommodate various computing systems, like university systems and commercial systems. Interrelations between programs were taken into account, with the simplifying assumption that called programs are always fetched from the storage device on which they are stored. The removal of this assumption is a question open for further research. The model as presented was a static model; an open question is the determination of a storage policy for a dynamic environment, in which demand patterns for programs' execution vary in time; this facet of the problem is related to the pricing policy of the computer system, which has a major effect on the demand pattern, and thus also on the storage management. Finally, it should be noted that our model, with minor modifications, may be applied to other process and storage control problems.

Acknowledgements

We wish to express our gratitude to Dr. Melvyn H. Schwartz, Manager, Software Development at the Vogelback Computation Center, for his valuable help and advice during the writing of this paper.

References

[1] Babad, J. M., "A Record and File Partitioning Model," Report No. 7358, Center for Research in Business and Economics, University of Chicago, December, 1973.

[2] Babad, J. M., Balachandran, V., and Stohr, E. A., "Cost Evaluation of Storage Schemes," Report No. 7416, Center for Research in Business and Economics, University of Chicago, April, 1974.

[3] Balachandran, V., "An Integer-Generalized Transportation Model for Optimal Job Assignment in Computer Networks," Discussion Paper No. 58, Center for Mathematical Studies in Economics and Management Science, Northwestern University, October, 1973.

[4] Gecsei, J., and Lukes, "A Model for the Evaluation of Storage Hierarchies," IBM System Journal, 13, 2, pp. 163-178, 1970.

[5] Gilmore, P. C., and Gomory, R. E., "The Theory and Computation of Knapsack Functions," Operations Research, 14, 1045-1074.

[6] Grossman, D. D., and Silverman, H.F., "Placement of Records on a Secondary Storage Device to Minimize Access Time," Journal of the Association for Computing Machinery, 20, 3, pp. 429-439, July, 1973.

[7] Land, A. H., and Doig, A. G., "An Automatic Method for Solving Discrete Programming Problems," Econometrica 28, pp. 497-520, 1960.

[8] Mattson, R. L., Gecsei, J., Slutz, D. R., and Traiger, I. L., "Evaluation Techniques for Storage Hierarchies," IBM System Journal, 9, 2, pp. 78-117, 1970.

[9] Ramamoorthy, C.V., and Chandy, K. M., "Optimization of Memory Hierarchies in Multiprogrammed Systems," Journal of the Association for Computing Machinery, 17, 3, pp. 426-445, July, 1970.

[10] Salasin, J., "Hierarchical Storage in Information Retrieval," Communication of the Association for Computing Machinery, 16, 5, pp. 291-295, May, 1973.