This paper describes an experimental algorithm for allocating use of a central processing unit to perform separate data processing tasks in a multitasking system. The algorithm, which may control only a subset of the tasks being performed by the system, appears to improve run time for some work loads.

Tasks with a recent history of using input/output facilities are given preference. This heuristic treatment of tasks is carried over to the algorithm itself, which modifies its own characteristics based on its overall effectiveness in handling the tasks under its control.

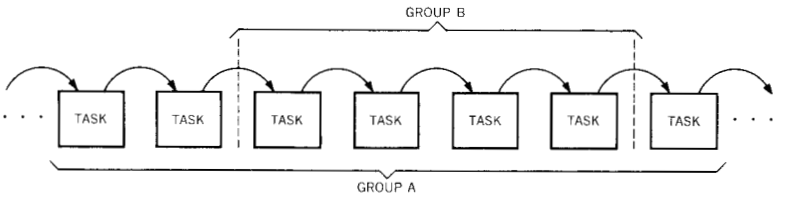
A heuristic approach to task dispatching

by K. D. Ryder

In any multiprogramming system, some rule is needed to define how CPU time is allocated among the *tasks*[1] competing for this resource. The particular form of this rule varies, depending on the goal of the system. If the goal is simply to guarantee the availability of the CPU to a few selected tasks whenever they require this resource, then a priority-based algorithm may suffice. If the objective is to improve overall throughput by making maximum use of total system resources, then a different approach is needed. A variety of techniques and combinations of techniques may be used.

This paper describes an experimental algorithm for allocating CPU time among tasks, a process we refer to as *dispatching*. The algorithm is designed to operate on a subset (or possibly the entire set) of the tasks that have been initiated in the system. The objective of this algorithm is to enable this subset of tasks to use system resources more efficiently, so that more work is completed per unit of time. One way to improve this rate (throughput) is to distinguish between those tasks that are dependent primarily on the central processing unit (CPU) and those tasks that are dependent primarily on input/output (I/O) operations. By giving preferential treatment to those tasks that use the I/O facilities more heavily, I/O and CPU operations can be overlapped. This phenomenon occurs because once an I/O operation has been initiated for a task, the CPU is

Figure 1 Task queue



generally not needed for that task until the I/O operation has been completed. Thus the CPU is available for use in performing other tasks. With this increased utilization of computing system resources, overall system throughput can be expected to improve correspondingly.

Not all tasks need be executed under control of this algorithm; other dispatching rules may coexist to govern the operation of the system in performing other tasks.

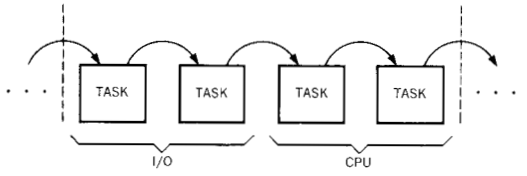
The dispatching algorithm under discussion has a number of unusual features. Foremost among them is its heuristic nature. Not only does the algorithm alter the handling of each task as the task's characteristics are determined, but the algorithm also alters itself based on its effectiveness in handling the totality of tasks under its control.

Algorithm operation

Assume that all tasks form a queue, as shown in Figure 1. We will call this set of tasks group A. The subset of tasks affected by the algorithm constitutes group B. The rules governing the positioning and dispatching of tasks not in group B are of little concern here, but let us assume that the task queue is in general searched from left to right. In competing for the CPU, tasks to the left of group B have preference over group B tasks; tasks to the right of group B tasks have the lowest preference. Group B tasks have no preference over each other beyond that inherent from their positions in the queue. A task is considered to be eligible for dispatching unless it is awaiting the completion of some event, such as an I/O operation.

In the context of this algorithm, the distinction between I/O-oriented and CPU-oriented tasks is reflected in the subdivision of group B tasks into two subgroups, as shown in Figure 2. I/O-oriented tasks are grouped to the left and CPU-oriented tasks are grouped to the right. This ensures that I/O-oriented tasks are offered use of the

Figure 2 Group B tasks



CPU first. After channel activity has been initiated for such tasks, typically use of the CPU can be relinquished until the channel operation has been completed.

determining task characteristics

Previous attempts at designing dispatching algorithms have taken into account the characteristics of tasks by compiling an historical record of each task throughout its execution.[2-4] However, task characteristics may change. It is possible that a long-time historical record that dictates dispatching decisions may always be out-of-date and hence incorrect and that only recent history should govern dispatching decisions. The algorithm described in this paper is based on that assumption. The I/O versus CPU determination is made as follows. As each task in group B is dispatched, it is monitored for a predetermined time interval. This action has three possible consequences:

1. The task may use the CPU for the entire time interval.
2. The task may voluntarily relinquish control of the CPU (as when waiting for completion of I/O activity).
3. The CPU may be preempted for a task higher in the task queue (for example, after an I/O operation for such a higher-priority task is completed).

Unused portions of time intervals are not saved.

In case number 1, the task is marked as CPU-oriented; in case number 2, the task is marked as I/O-oriented; in case number 3, the previous designation of the task remains unchanged. Hence, an essentially binary distinction is made among tasks in group B, and a short-time historical record is used to make this distinction. The algorithm tracks each task's activity as closely as possible and says, in effect, that the most pertinent history is the most recent history. It assumes that the task most likely to be I/O-oriented the next time it is dispatched is the task that was I/O-oriented the last time it was dispatched.

Each time a task in group B relinquishes CPU control, it becomes a candidate for a change in its relative position within the task

Table 1 Dispatching algorithm behavior

Original task status	*Reason for loss of CPU control*	*New task status*	*Action taken*
I/O	voluntary surrender	unchanged	Search down queue for new task to dispatch.
	time interval ended	CPU	Move task to head of CPU subgroup; search down queue from old location of task.
	preemption for another task	unchanged	Dispatch preempting task.
CPU	voluntary surrender	I/O	Move task to bottom of I/O subgroup; search down queue from old location of task.
	time interval ended	unchanged	Move task to bottom of CPU subgroup; search down queue from old location of task.
	preemption for another task	unchanged	Move task to bottom of CPU subgroup; dispatch preempting task.

queue. If a task was previously marked I/O and its characteristics are not changing, no movement occurs. If a task was previously marked CPU and remains CPU, it is shifted to the bottom of the CPU subgroup within group B. If a task formerly marked CPU is now being marked I/O, it is moved to the bottom of the I/O subgroup within group B. Conversely, an I/O task being changed to CPU status is queued at the top of the CPU subgroup. These situations are presented in Table 1.

Note that the shifting of tasks within group B biases the left-to-right search of the task queue in a number of ways. First, I/O tasks that are marked I/O tend to migrate higher in the queue as other I/O tasks change status and drop out to the CPU subgroup. A task that changes from I/O to CPU status is, however, treated preferentially over all CPU tasks the next time it is dispatched. Similarly, a CPU task being switched to the I/O subgroup has the lowest preference of all I/O tasks. These mechanisms aid in making a finer distinction between tasks that have relatively constant characteristics and those characterized by many

status changes. The cyclic movement of tasks within the CPU subgroup ensures that all CPU-oriented tasks share in any available CPU time. Such tasks thus have a high probability of being allowed to exhibit the need for status changes; potential I/O tasks are not locked at the bottom of the CPU subgroup indefinitely.

It may be seen that the relationship of the algorithm to each task implies an essentially heuristic treatment of that task. The behavior of the task is observed, and its handling is altered accordingly. The algorithm itself has important heuristic characteristics. As it operates on the totality of tasks in the system at any given time, the algorithm adjusts itself to provide maximum effectiveness relative to the entire group of tasks under its control.

self-adjusting characteristics

The self-adjusting characteristics of the algorithm are governed by six parameters:

1. An initial time interval that is assigned to group B tasks as they are dispatched
2. An incremental time that can be added to or subtracted from the original time interval
3. A lower limit on the adjusted time interval value
4. An upper limit on the adjusted time interval value
5. A predetermined value expressing the desired ratio of X to Y, where X = total number of times group B tasks used their entire time interval and Y = total number of times group B tasks have been dispatched (i.e., a ratio indicating whether enough CPU-oriented tasks are being identified)
6. A statistics interval used to determine the frequency with which the algorithm adjusts itself

Throughout a statistics interval, all group B tasks are dispatched with the same time interval value. Counts are kept of the X and Y values. When the statistics interval concludes, the ratio of X to Y is computed and compared to the value of parameter 5. If the calculated value is lower, the resolution of the algorithm is not adequate to detect enough CPU-oriented tasks. Accordingly, the current value of parameter 1 is decreased by the magnitude of parameter 2. The likelihood of identifying CPU tasks is thus increased for the next statistics interval. In a similar manner, the value of parameter 1 is increased if too few I/O-oriented tasks are being identified by the algorithm. An attempt is always made to perform some differentiation relative to any mix of tasks presently in group B. The upper and lower limits serve to keep the adjusted value of parameter 1 within reasonable bounds. The counters X and Y are set to zero at the start of each statistics interval.

The six parameters used to drive the algorithm appear to be sensitive in some degree to the environment in which the algorithm is operating. Behavior of the algorithm seems to depend on the

combination of the values chosen for these parameters and such other factors as central processing unit speed, job characteristics, system configuration, etc. Thus, optimum parameter values for one set of conditions would not be optimum for another. Our choice of parameter values was largely intuitive, with seemingly satisfactory if not optimum results. For example, the parameter values used in making the set of measurements on the System/360 Model 65 were as follows:

Parameter 1, the initial time interval, was 150 milliseconds. This value was chosen to allow enough time both to accomplish some useful work and to monitor task characteristics, but not so long as to needlessly delay determining the characteristics of other tasks.

Parameter 2, the incremental time, was 5 milliseconds. This value, which like others is dependent on CPU speed, seemed large enough to sense changes in task mix at our statistics interval (parameter 6). Of course, if the algorithm adjusted itself to changing conditions less frequently, parameter 2 ought to be larger to approach optimum performance in fewer statistics intervals. But if parameter 2 becomes too large, there is a loss in resolution in separating I/O-oriented from CPU-oriented tasks.

We established the lower limit of the time interval (parameter 3) at 50 milliseconds. Had this interval been too small, excessive overhead could have been incurred in attempting to distinguish among tasks that are essentially all I/O-oriented.

The upper limit on the time interval, parameter 4, was 500 milliseconds. A large interval here reduces the overhead involved in switching among tasks that are all essentially CPU-oriented. However, unless an upper bound is established, changing the status of some tasks to I/O would be intolerably delayed.

We used a task-switching ratio, parameter 5, of 1:2. This provided a gross sort of control over behavior of the algorithm with changing characteristics of the entire group of tasks.

A statistics interval, parameter 6, of one second was used. We felt that this period allowed the algorithm to adjust itself sufficiently often to respond efficiently to changing characteristics of the task group. A shorter statistics interval would have allowed a smaller adjustment value (parameter 2), and thus closer tracking of the mix of task characteristics, but it would have increased overhead.

Performance testing

A prototype algorithm based on the ideas presented in this paper has been implemented and incorporated into a System/360 Oper-

Table 2 System/360, Model 65 measurements

Job type	*Simultaneous group B tasks*	*Run time (sec)*	*Percent change with heuristic dispatcher*			
			Total run time	*Wait time*	*CPU time*	*CPU-I/O overlap time*
FORTRAN	4	784	−16	−45	+1	+47
COBOL	3	1298	+ 1	0	+2	+ 5
Mixed FORTRAN-COBOL	4	2154	−11	−25	+2	+33

ating System capable of multiprogramming with a variable number of tasks (MVT). Test results obtained using this system in three models of System/360 computers are summarized in Tables 2, 3, and 4.

The first column in each of the tables indicates the kind of work load. The second column indicates the number of group B tasks that existed simultaneously. The total amount of time required for a single run of the work load is recorded in the next column. It should be noted that the described work load was usually executed more than once, sometimes as often as four times. The run times (with one exception noted later) recorded in the tables are typical rather than average. All run times recorded in the tables are for systems using multiprogramming but not using the heuristic dispatcher. The remaining columns all indicate the percent change in the quantity being considered when the heuristic dispatcher is used.

Each of the total work loads used to obtain the data in Table 2 were run two to four times, with variations in run time of ± 2 percent. For example, run times for the FORTRAN work load with the dispatching algorithm operative were 651, 662, 665, and 667 seconds.

The FORTRAN jobs included 13 different technical programs derived from actual customer work loads. Each job appeared twice in the work load and was composed of a FORTRAN compilation step, a linkage editing step, and an execution step. The programs were designed to solve problems such as: heat transfer, mechanical design, capacitor analysis, diode curves (including matrix inversion), banking transactions, missile stability, spectrum analysis, man-machine interaction, missile range and thrust, double integrals, missile impact spotting, and transportation analysis.

In order to reveal the type of work involved in this FORTRAN work load, it was run on a different system configuration using a

Table 3 System/360, Model 50 measurements

Job type	Simultaneous group B tasks	Run time (sec)	Percent change with heuristic dispatcher: Total run time	Wait time	CPU time	CPU-I/O overlap time
FORTRAN	2	1649	−8	−63	+1	+71
COBOL	3	2178	−5	−29	+4	+16
Mixed FORTRAN-COBOL	2	5584	−7	−43	+2	+28

Table 4 System/360, Model 195 measurements

Job type	Simultaneous group B tasks	Run time (sec)	Percent change with heuristic dispatcher: Total run time	Wait time	CPU time	CPU-I/O overlap time
Special FORTRAN work load A	6	880	−10	−57	0	+60
Special FORTRAN work load B	8	114	− 8	−41	+3	not available
Special FORTRAN work load C	6	136	−11	−35	+2	+28

System/360 Model 65 but without multiprogramming. The total run time was 1050 seconds. The CPU was used for 746 seconds of this time, leaving the CPU in the wait state for 29 percent of the time.

The COBOL jobs were intended to be a representative selection of commercial applications. Each of the six different COBOL jobs appeared once in the work load, yielding a total of 7 COBOL compilation steps, 21 linkage editing steps, and 24 execution steps (including 3 sorting steps). Note the slight increase in total run time here. The COBOL work load was also run on a Model 65 configuration without multiprogramming. In this case, total run time was 1565 seconds, and the CPU was in the wait state 76 percent of the time. In doing this work load in a Model 65 configuration using multiprogramming, it appears that there is insufficient demand for the CPU to allow a meaningful increase in CPU-I/O overlap, and overhead resulting from use of the algorithm is added to run time.

The figures in the final row of Table 2 are for the mixture of the FORTRAN and COBOL work loads. This work load was also run on a Model 65 configuration without multiprogramming. Total run time was 3665 seconds, with the CPU in the wait state for 49 percent of the time.

Table 3 contains the results of running the same work loads on a System/360 Model 50. In this case, each of the FORTRAN jobs and each of the COBOL jobs was run once. Note that run time for the COBOL work load is improved using the algorithm in the Model 50. Both of these work loads were then combined for the mixed FORTRAN-COBOL run except that the FORTRAN jobs appeared twice.

For the System/360 Model 195 runs recorded in Table 4, three special FORTRAN work loads were used. Special work load A consists of 18 jobs, with each job containing a FORTRAN compilation, linkage editing step, and execution step. This total work load was run twice. Variations in results were about ± 1 percent, and figures in the table are averages. Special work loads B and C were each run at least twice, but results are representative rather than average. Special work load B is a set of twenty more FORTRAN jobs; each job includes a FORTRAN compilation and execution (using the System/360 Operating System loader). The ten distinct jobs in special work load C each appear three times; these jobs, which each consist of a FORTRAN compilation and execution, also used the loader. These work loads are sufficient to avoid trivially short throughput times on a computer as fast as the Model 195.

Special work load B was run on the System/360 Model 195 without multiprogramming. Total run time was 209 seconds, and the CPU was in the wait state for 64 percent of this time. Thus on this fast computer, this work load appears to be I/O-oriented.

Summary comment

The work described in this paper suggests that throughput gains are possible for some work loads if dispatching is controlled by a self-adjusting algorithm that takes into account the changing characteristics of tasks both singly and as a group. Minimum benefits can be anticipated when the characteristics of the tasks are homogeneous; if all tasks are heavily I/O-oriented or all tasks are heavily CPU-oriented, no throughput improvement can be anticipated. In fact, additional overhead will be incurred handling time intervals and manipulating task queues. Under these conditions, the only benefit is the cyclic dispatching of all CPU-oriented tasks, which prevents any single task from monopolizing use of the CPU.

Maximum benefits can be anticipated for a mixture of heavily

CPU-oriented and heavily I/O-oriented tasks. It is under these conditions that the algorithm can strive for maximum CPU-I/O overlap.

ACKNOWLEDGMENTS

Several individuals made significant contributions to the development and measurement of the heuristic dispatcher. Of particular note was the work of Miss M. J. Alleger, who not only contributed to the detailed design but also implemented the prototype algorithm. Credit also goes to W. T. Hall and W. O. Birkett, Jr., for independent measurements of the prototype.

CITED REFERENCES

1. B. I. Witt, "The functional structure of OS/360, Part II, Job and task management," *IBM Systems Journal* 5, No. 1, 12–29 (1966).
2. B. S. Marshall, "Dynamic calculation of dispatching priorities under OS/360 MVT," *Datamation* (August 1969).
3. M. Mikelsons, "A flexible task scheduling scheme for a real-time environment," IBM Research Report RC 2428, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (March 6, 1969).
4. C. R. Attanasio, P. W. Markstein, and C. E. Shanesy, "A dispatching algorithm for a conversational, high-capacity computational subsystem for OS/360 MVT," IBM Research Report RC 2483, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (May 23, 1969).