A TASK-SCHEDULING ALGORITHM FOR A MULTIPROGRAMMING COMPUTER SYSTEM

K. L. Krause*, V. Y. Shen, and H. D. Schwetman
Purdue University
West Lafayette, Indiana 47907

Abstract

This paper presents a description and analysis of a task scheduling algorithm which is applicable to third generation computer systems. The analysis is carried out using a model of a computer system having several identical task processors and a fixed amount of memory. The algorithm schedules tasks having different processor-time and memory requirements. The goal of the algorithm is to produce a task schedule which is near optimal in terms of the time required to process all of the tasks. An upper bound on the length of this schedule is the result of deterministic analysis of the algorithm. Computer simulations demonstrate the applicability of the algorithm in actual systems, even when some of the basic assumptions are violated.

I. Introduction

The algorithm which assigns competing tasks to available processors and memory is a critical component in third generation computer systems. Given a collection of tasks awaiting processing, this algorithm produces a schedule which assigns tasks (or pieces of tasks) to the available resources during the ensuing periods of time. One measure of the performance of this algorithm is the amount of time required to complete the produced schedule.

The algorithm presented in this paper is such a task-scheduling algorithm. This algorithm is constructed so as to have three important properties:

1. It is applicable in a realistic operating environment,
2. It produces "good" schedules, and
3. Its behavior can be analyzed using deterministic methods.

Furthermore, the algorithm produces its schedule in a reasonable (less than exponential) number of steps. The analysis which accompanies the description of the algorithm provides three theorems which give bounds on the length of the schedule produced, given a set of tasks and a system configuration.

Our basic model of a multiprogramming/multiprocessing system contains n identical and independent processors (or perhaps virtual processors.) It has M units (pages) of memory used to hold tasks which have been assigned to a processor. Each task (T_j) requires a fixed amount of memory (m_j units) and t_j units of processor time. We shall assume that all tasks are available for scheduling at the time the schedule is produced. An optimal schedule is defined to be a schedule which completes all tasks in minimal time. This model is intended to correspond to a real system with M units of memory and which can support n degrees of multiprogramming. Each task would have different but fixed (and known) memory and processor time requirements.

The basic system model with memory constraint is a departure from similar models which have appeared in the literature [1,2,3]. The model deals with memory as an absolute constraint and not in a memory management sense (cf. [4]). We also assume that tasks may be transferred in and out of memory with negligible delay (cf. [5]).

The paper presents two versions of the algorithm; each version is analyzed and the bounds on the schedule-length provided. The first version is a simplified one in which all tasks require the same amount of processor time. The second version is the more general one, in which each task can have different time requirements. The applicability of the algorithm is demonstrated by using a simulation model of a computer system. The results of this simulation are compared with data gathered from an actual system (the CDC 6500 in use at the Purdue University Computing Center.)

II. The Unit-Time Model

As a first step toward the analysis of the computation model, we shall assume that all tasks have the same amount of processor time requirement. The following example shows that the scheduling strategy used may make a great difference in the resulting completion time.

Example 1. Given a system with four processors and 100 units of memory, and a list of tasks with memory requirements as shown below:

$(J_1, J_2, \ldots, J_{16})$ =
(5,5,5,5,9,9,10,10,33,33,35,35,51,51,52,52)

---

* Current address: Air Force Weapons Laboratory, Kirtland AFB, Albuquerque, NM 87117

A schedule of the tasks placed according to the order in the list is shown in Figure 1

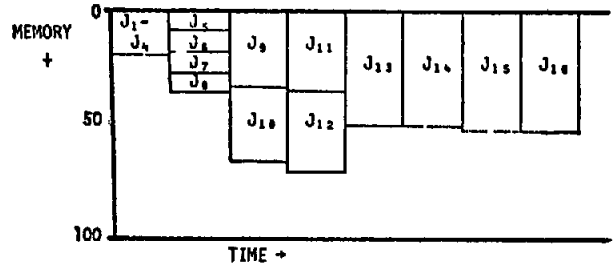

Figure 1. A bad schedule for Example 1.

Figure 1 could be considered as a two-dimensional Gantt chart. A vertical rectangle represents a single time slice, each with 100 units of memory and four processors available. The time required to complete the 16 tasks is eight units. If we place the tasks according to the following list, a shorter schedule results and is shown in Figure 2.

Task List =
(52,33,10,5,52,33,10,5,51,35,9,5,51,35,9,5)

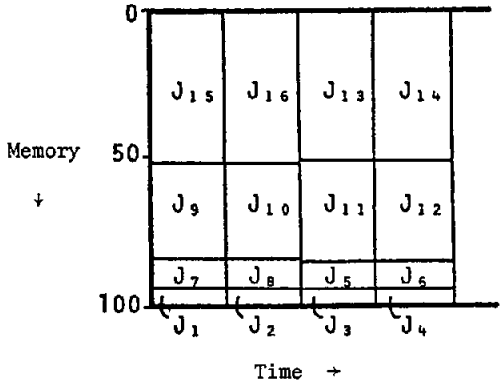

Figure 2. An optimal schedule for Example 1.

The schedule shown in Figure 2 has the shortest completion time for the list of tasks since both the processors and memory are fully utilized in each time slice. It is not difficult to find the optimal schedule in this case, and the optimal schedule is known to exist due to the finiteness of the problem. However, it is generally accepted that the generation of an optimal schedule for an arbitrary environment requires an exhaustive process. Such method would be inappropriate to apply to computer scheduling since it might take an exponential number of steps.

The task list of Example 1 is interesting since it suggests a simple method to find a good schedule. Let us reorder the list of tasks according to descending memory requirement. The following list is obtained:

$(T_1,T_2,\ldots,T_{16})$ =
(52,52,51,51,35,35,33,33,10,10,9,9,5,5,5,5)

The optimal schedule of Figure 2 is relabeled and shown in Figure 3.

Figure 3 shows that a good schedule can be obtained using a systematic method. Such a method starts by making an estimate of the number of time slices the final schedule may take.

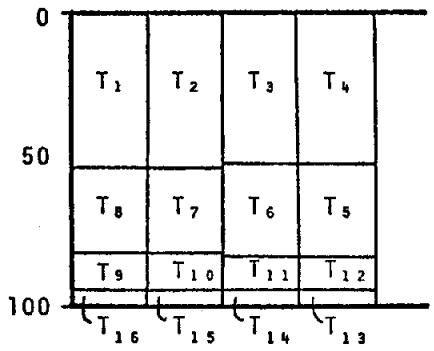

Figure 3. Figure 2 relabeled.

An optimistic starting point is the lower bound of the length of any schedule N_L, where

$$N_L = \left\lceil \max\left(k/n, \sum_{j=1}^{k} m_j/M\right)\right\rceil \qquad (1)$$

In Eq. (1), the system is assumed to have n processors and M units of memory, the task list has k tasks, and the memory requirement of the j^{th} task is denoted by m_j. $\lceil x \rceil$ represents the smallest integer greater than or equal to x. The placement strategy assigns the largest remaining task to the time slice with the maximum amount of available memory. This simple strategy enables the placement of all tasks in N_L time slices in Figure 3, and we can see that an optimal schedule is obtained. If N_L time slices are not sufficient to place all tasks, another guess must be made and the schedule reconstructed. The following is a formal description of the scheduling strategy.

Algorithm 1.

The proposed algorithm assigns tasks from a task list ordered on memory requirement (m_j), largest requirement first. There are k tasks to be assigned. The tasks are assigned across time slices denoted by the index i. Each time slice has two associated variables, n_i - the number of processors in use and c_i - the amount of available memory. M is the total number of memory units at each time slice and n is the number of processors. N_a will denote the number of time slices used by the algorithm to assign the given tasks. The final value of N_a is determined by iteratively attempting to place all the tasks in an incremental number of time slices.

0) Set $N_a = \left\lceil \max\left(k/n, \sum_{j=1}^{k} m_j/M\right)\right\rceil$

1) For $i = 1,2,\ldots,N_a$ perform the following steps:

1.1) Assign the i^{th} task to the i^{th} time slice.

1.2) Set $n_i = 1$ and $c_i = M - m_i$.

2) Set $j = N_a + 1$

3) Set $I = \max (i \mid c_i \geq c_\ell$ for all ℓ, $1 \leq \ell \leq N_a$ and $n_i \neq n)$

4) If $c_I < m_j$ then set $N_a = N_a + 1$ and go to step 1.

5) Assign the j^{th} task to the I^{th} time slice.

6) Set $n_I = n_I + 1$, $c_I = c_I - m_j$, and $j = j + 1$

7) If $j > k$ then terminate, otherwise go to step 3.

Step 3 of the algorithm deterministically selects the time slices with the identical amount of available memory. If the time slice with the maximum amount of available memory can not accommodate the next task, Step 4 indicates that we can increase the original estimate by one time slice and try again.

The amount of time required by Algorithm 1 to produce a schedule is bounded by a second degree polynomial in k/n [6]. The formal description permits the algorithm to be analyzed using deterministic methods. The results of this analysis can be used to reduce the number of iterations required by the algorithm.

The following theorem, which is the result of deterministic analysis, provides an upper bound on the completion time for the schedule produced by this strategy.

Theorem 1. If the number of time slices required by Algorithm 1 is N_a and the number of time slices required for an optimal schedule is N_{op} then

$$N_a < (\tfrac{4}{3})\, N_{op} + 1. \qquad (2)$$

The proof of this theorem is very involved and is reported in [6]. We can demonstrate that this bound is approachable for a large set of tasks by considering a task list with two classes of tasks, those with memory requirements approximately M/2 and those with requirements approximately M/4. By proper adjustment of the memory requirements we can construct an optimal schedule which is the same length as the lower bound and an algorithmically produced schedule that approaches 4/3 of this length. The task sizes are chosen in such a manner that the optimal schedule would use 100% of the memory resource and the algorithmic schedule would be the maximum possible, i.e. a pair in each time slice. Consider an arbitrary number of classes of time slices denoted by $A_1, A_2, \ldots, A_r$. The widths or number of time slices represented by each class will be denoted by $N_1, N_2, \ldots, N_r$ respectively. The optimal schedule would appear as follows (all time slices within a denoted class are identical):

A_1	A_2	A_3	A_4		A_{r-2}	A_{r-1}	A_r
M/2+ε	M/2-4ε	M/2-10ε	M/2-22ε		M/2-bε	M/2-(2b+2)ε	M/4
M/2-ε	M/4+2ε	M/4+5ε	M/4+11ε	...	M/4+(b/2)ε	M/4+(b+1)ε	M/4
	M/4+2ε	M/4+5ε	M/4+11ε		M/4+(b/2)ε	M/4+(b+1)ε	M/4
							M/4

where $\varepsilon > 0$

and $$b = 2^{r-3} + \sum_{q=1}^{r-3} 2^q \text{ for } r \geq 2 \qquad (3)$$

Note that the space requirement at each time slice is M. (i.e. all of memory.)

$$N_{op} = \sum_{i=1}^{r} N_i = \sum_{j=1}^{k} m_j / M \qquad (4)$$

where m_j is the memory requirement for each of the k individual tasks.

We restrict ε at this point by

$$M/2 - (2b+2)\varepsilon > M/4 + (b+1)\varepsilon \qquad (5)$$

to guarantee the first tasks in A_{r-1} are larger than the others

or $$\varepsilon < M/12(b+1) \qquad (6)$$

This restriction is imposed so we can explicitly depict the algorithmic schedule. The final algorithmic schedule S, would be

M/2+ε	M/2-ε	M/2-4ε	M/2-10ε	...	M/2-bε	M/2-(2b+2)ε
M/4	M/4+2ε	M/4+5ε	M/4+11ε		M/4+(b+1)ε	M/2-(2b+2)ε

By looking at the number of tasks associated with each subclass above we have:

$$N_a = N_1 + N_1 + N_2 + N_3 + \ldots + N_{r-2} + N_{r-1}/2 \qquad (7)$$

In order to satisfy this exact schedule we must have the following relationships:

$$N_r = N_1/4$$

$$N_2 = N_1/2$$

$$N_3 = N_2/2 = N_1/2^2$$

$$N_4 = N_3/2 = N_1/2^3$$

$$\vdots$$

$$N_{r-1} = N_{r-2}/2 = N_1/2^{r-2}$$

It then follows from (4) that

$$N_{op} = \sum_{i=1}^{r} N_i = N_1 \left(\sum_{i=0}^{r-2} (\tfrac{1}{2})^i + \tfrac{1}{4} \right) \qquad (8)$$

and from (7) that

$$N_a = N_1 \left(2 + \sum_{i=1}^{r-3} (\tfrac{1}{2})^i + (\tfrac{1}{2})^{r-3}\right) \qquad (9)$$

Now let $r \to \infty$; it follows from (8) and (9) that

$$N_{op} \to (9/4)\, N_1$$

and $$N_a \to 3 \cdot N_1$$

therefore $N_a \to (4/3)\, N_{op}$.

III. The Variable-Time Model

Algorithm 1 of the previous section can be extended easily to cover cases when the tasks have different processor time requirements. The assumption to be made is that a task requiring t units of processor time may be broken into t sub-tasks, each requiring one time unit. This assumption is reasonable if the computer system being modeled has a preemptive-resume feature. Each sub-task is then assigned using the same strategy as the unit-time model, with the added constraint that no two sub-tasks of the same task may be placed in the same time slice.

Algorithm 2.

As in the previous algorithm, we have a list of tasks ordered on memory requirement (m_j), largest requirement first. There are k tasks to be assigned, each with t_j time units. The sub-tasks (of one time unit) are assigned across time slices denoted by the index i. Each time slice has two associated variables, n_i - the number of processes in use and c_i - the amount of available memory. M is the total number of memory units at each time slice and n is the number of processors. The algorithm starts by estimating the length of the schedule (N_a) to be the lower bound, and increments N_a iteratively until all tasks are placed.

0) Set $N_a = \left\lceil \max \left(\sum_{j=1}^{k} t_j/n, \sum_{j=1}^{k} m_j t_j/M, \max_j(t_j) \right) \right\rceil$

1) (First-round assignment)
For $j = 1,2,\ldots,b$ where

$$b = \{q \mid \sum_{j=1}^{q} t_j \leq N_a \text{ and } \sum_{j=1}^{q+1} t_j > N_a\}$$

perform the following steps (start with the first time slice):

1.1) Assign the j^{th} task to the next t_j time slices.

1.2) Set $n_i = 1$ and $c_i = M - m_j$ for each of the t_j slices.

2) Set $j = b + 1$.

3) Set $I = \max \{i \mid c_i$ is the maximum of a set of c_d's, where $n_d \neq n$ and no part of the j^{th} task has been assigned to the d^{th} slice$\}$.

4) If $c_I < m_j$ then set $N_a = N_a + 1$ and go to Step 1.

5) Assign a unit of the j^{th} task to the I^{th} time slice.

6) Set $n_I = n_I + 1$, $c_I = c_I - m_j$, and $t_j = t_j - 1$.

7) If $t_j > 0$, go to Step 3.

8) If $j = k$ then terminate.

9) Set $j = j + 1$ and go to Step 3.

Steps 3 through 7 attempt to assign the j^{th} task to the t_j time slices that have the maximum amount of memory available. If this is not possible, Step 4 indicates that we can increase the original estimate by one time slice and try again. Algorithm 2 produces a schedule in a finite number of steps due to the finiteness of the problem. The similarities between Algorithms 1 and 2 allow the analytical method used in the proof of Theorem 1 to be extended. The bound on the time required to produce a schedule mentioned in the discussion following Algorithm 1 also applies to Algorithm 2 [6].

Theorem 2. If the number of time slices required by Algorithm 2 is N_a, and the number of time slices in the optimal schedule is N_{op}, then

$$N_a < (2 - \frac{2}{n})N_{op} + 1. \qquad (10)$$

The proof of the theorem can be found in [6].

We can demonstrate that this bound is approachable by scheduling the following set of tasks. Note that the requirement of a task is represented by the 2-tuple (m,t).

Name	Number	Requirement
J_1, J_2	2	$(\frac{M}{2} - \varepsilon, \frac{N_{op}}{2} - 1)$
J_m	$(n-2)N_{op} + 2$	$(\frac{2\varepsilon-\delta}{n-3}, 1)$
J_f	1	(δ, N_{op})

The variables ε and δ are chosen so that $(\frac{M}{2} - \varepsilon) > n(\frac{2\varepsilon-\delta}{n-3})$ and $(\frac{2\varepsilon-\delta}{n-3}) > \delta$, and all fractional expressions yield integer values. The optimal schedule has N_{op} time slices and is shown in Figure 4. Note that

$$(n-2)N_{op} + 2 = (n-3)(\frac{N_{op}}{2} - 1) + (n-1)(\frac{N_{op}}{2} + 1).$$

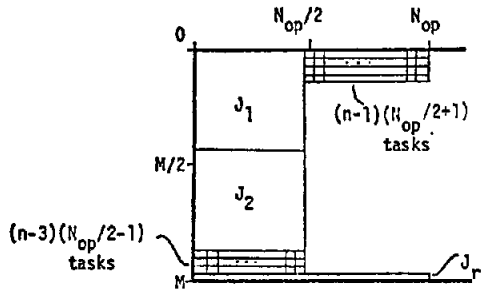

Figure 4. The Optimal Schedule

The next-to-final schedule (N_a-1 time slices) is shown in Figure 5. Note that two units of J_f can not be placed but one additional time slice in the schedule is adequate to assign all tasks by the algorithm.

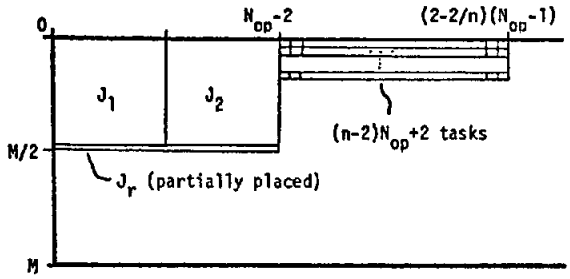

Figure 5. The schedule with (N_a-1) time slices.

It is evident from the figure that

$$N_a - 1 = N_{op} - 2 + \frac{(n-2)N_{op} + 2}{n}$$

$$= (2 - \frac{2}{n})(N_{op} - 1)$$

which approaches (10) when N_{op} is large.

The bound given by Theorem 2 is not very impressive. It indicates the possibility that the length of the schedule produced by Algorithm 2 could be nearly twice as long as the optimal schedule. However, further analysis of the behavior of the algorithm shows that the situation of Figure 5 occurs only when there are many jobs with small memory requirements (J_m's and J_f) and the smallest job (J_f) has extremely large time requirement. In the next-to-last iteration of the algorithm portions of the smallest job (J_f) can not be assigned because all time slices have either n task units assigned or parts of J_f assigned already. This situation does not occur during simulation studies using benchmark jobs on the CDC 6500 and can be detected easily by the algorithm if it does occur. The following theorem seems to cover a large number of scheduling environments.

Theorem 3. In the iterative process of Algorithm 2, if there exists some time slice with less than n task units assigned but no task unit of the final task, then

$$N_a < \frac{4}{3} N_{op} + 1 \qquad (11)$$

The proof of the theorem can be found in [6].

The example used to demonstrate the approachability of Theorem 1 may be used for Theorem 3, as it is a special case of the variable-time model.

Theorems 2 and 3 may be used to reduce the iterative aspect of Algorithm 2. For example, in situations where Theorem 3 applies, Step 4 of Algorithm 2 can be modified to change the estimated length of the schedule in steps. Note that the lower bound of the length of the schedule is computed in Step 0, and the upper bound is established by Theorem 3. (This is possible since the proof of Theorem 3 assumes N_{op} to be that computed by Step 0 of Algorithm 2.) Thus using a bisection method we can bring the length of the schedule to approximate the true result of Algorithm 2 with few iterations.

IV. Simulation Results

The schedule completion times produced by Algorithm 2 (denoted N_a) depend on certain assumptions implicit in the computational model. The most obvious assumption is that tasks do not change their memory requirement during execution. Also, while Theorem 3 bounds the schedule-completion times produced by the algorithm, it does not provide information about the typical completion times.

In order to assure ourselves that the assumptions are not unreasonable and that the expected performance was significantly better than the predicted bound, we implemented a simulated system which used the job placement strategy of Algorithm 2. The job descriptions processed by the simulator were automatically produced by analyzing data gathered from the Purdue-MACE/CDC 6500 computer system in use at the Purdue University Computing Center [7]. The data allowed us to compute the dynamic memory and time requirements for all of the jobs processed by the system during periods of observation. Thus, we were able to use realistic data for our simulation study of the behavior of Algorithm 2. As the data were processed to produce the required job descriptions, the actual schedule completion times realized by the production system were recorded. Those actual times served as a basis for judging the performance of the algorithm.

The data was gathered during two periods of production on the Purdue system and analyzed as a series of 50 second intervals. The results of the data analysis and the simulation runs using this data are presented in Table 1.

The Purdue-MACE system uses a priority-driven job scheduler with a preemptive-resume feature. Thus in Table 1, the number of tasks observed during a 50-second interval is the number observed during some phase of execution, not necessarily the number of tasks completed or initiated. The entries labeled T are three schedule-completion times. T(LOWER) is the time equivalent of the initial value of N_a in Algorithm 2. This is a lower bound on the schedule-completion time and serves as an approximation to N_{op}. T(ACTUAL) is the observed schedule-completion time (approximately 50 seconds.) T(SIM) is the schedule-completion time produced by Algorithm 2 (really the time equivalent of the final value of N_a of Algorithm 2.)

M(ERROR) is an indication of the magnitude of the error in the memory actually used as compared to the approximation used by the simulator (and Algorithm 2.) The space-time integral for each

TEST TAPE NO. 1

02/06/73. 13.30.00.

INTERVAL	NO.TASKS	T(LOWER)	T(ACTUAL)	T(SIM)	M(ERROR)
1	30	44.10	50.00	44.10	.07
2	29	48.90	50.00	49.35	.20
3	23	52.77	49.99	52.89	.38
4	22	49.95	49.95	49.95	.35
5	25	48.00	50.00	48.15	.19
6	25	49.20	49.61	49.20	.21
7	19	48.75	49.41	48.75	.30
8	20	47.10	49.59	47.10	.12
9	21	43.80	41.69	43.95	.16

TEST TAPE NO. 2

03/20/73. 13.48.56.

INTERVAL	NO.TASKS	T(LOWER)	T(ACTUAL)	T(SIM)	M(ERROR)
1	26	48.30	50.00	48.30	.08
2	30	44.10	50.00	44.25	.06
3	32	47.85	49.99	48.30	.19
4	22	43.50	49.99	43.65	.03
5	17	50.55	50.00	50.70	.22

Table 1

Comparison of Results Observed on System and Produced by Scheduling Algorithm

task's memory-time requirement was computed as part of the data analysis. The approximation used by the simulator is the product of the initial memory requirement (at the onset of a 50 second interval) and the time-in-core for each task (see Figure 6.) The total memory error for each interval is the sum of the absolute differences between the actual and approximate space-time integrals for each task. The percentages in the M(ERROR) column of Table 1 are the total errors as percentages of the actual value of the space-time integrals for the intervals.

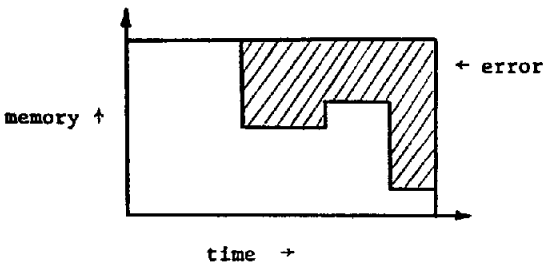

Figure 6. Illustration of Memory Error for Single Task

It can be seen that when the errors are small, the schedule produced by Algorithm 2 is better (shorter) than that produced by the priority-driven scheduler of the Purdue-MACE system. Larger errors cause the simulation to display completion times which are worse than those produced by the system. The obvious cause of worse performance in the event of larger errors is that several tasks may decrease their memory requirement allowing more tasks to be fit into memory concurrently in the actual system than in the simulated system. Such a decrease is quite common in the Purdue-MACE system, since most jobs begin with a compilation step and end with an execution step, at a reduced memory requirement.

The simulation results suggest that errors caused by changes in a task's memory requirement do not severely degrade the results of the strategy. They also suggest that the expected completion times are closer to the lower bound than the upper bound ((4/3)(50) + 1 ≃ 67.7 seconds.)

V. Remarks

Theorem 3 and the simulation results demonstrate that our task scheduling algorithm (Algorithm 2) has the desired properties mentioned in the introduction. In particular, we have provided a lower bound and an upper bound on the length of the schedule produced by the algorithm (Step 0, Algorithm 2 and Equation (11), Theorem 3.) It should be noted that the proofs of the theorems are all long and that no methodology which constructs such proofs has yet been developed. This leads us to believe that similar analysis of more complicated models would be very difficult.

The simulation results presented in section IV suggest that the scheduling algorithm may be of practical interest because its performance, while not necessarily optimal, is bounded. In an actual implementation, the system would maintain a list of tasks with information about the estimated time and memory requirements. The scheduler would be called to produce a schedule (really a time-space map) for the list. The system would then activate and de-activate tasks, according to this schedule. The scheduler would be called to produce a new schedule whenever the actual state of the system deviates from the expected or predicted state, e.g. when a task terminates prematurely or changes its memory requirement.

In spite of the promising results shown in the simulation study, it may not be desirable to implement the proposed strategy. For example, the strategy requires that the system has an efficient preemptive-resume feature. The strategy also requires accurate estimates of the time and memory requirements for all of the tasks, and it does not handle tasks requesting operator intervention, such as the mounting of a magnetic tape. Although the scheduler requires only an algebraic number of steps, the cost of implementation and computation may still be high in some environments. The most serious shortcoming of the strategy is that it does not consider other scheduling goals, such as task turn-around time or task priority.

Most schedulers in existing systems can be classified as demand schedulers. In particular, they produce a schedule for only the next one or two slices of time. Our algorithm represents a major departure from this existing approach in that it implements a "look-ahead" strategy. The simulation results show that when the memory-requirement errors are small, this look-ahead strategy can produce a noticeably improved schedule (e.g. a 5-10% reduction in schedule-completion times) when compared to the demand strategy in the Purdue system. This observation suggests that, as might be expected, look-ahead (perhaps they could be called two-dimensional) task schedulers may be a promising family of schedulers which deserve further examination.

REFERENCES

1. Graham, R. L. "Bounds on Multiprocessing Timing Anomalies," SIAM J. on Applied Math. (17,2) Mar. 1969, (416-429).

2. Kleinrock, L. "A Continuum of Time-Sharing Scheduling Algorithms," Proceedings SJCC (36) 1970, (453-458).

3. Muntz, R. R., Coffman, E. G., Jr. "Preemptive Scheduling of Real Time Tasks on Multipro - cessor Systems," JACM (17,2) Apr. 1970, (324-338).

4. Denning, P. J. "The Working Set Model for Program Behavior," CACM (11,5) May, 1968, (323-333).

5. Chen, Y. E., Epley, D. L. "Memory Requirements in a Multiprocessing Environment," JACM (19,1) Jan. 1972, (57-69).

6. Krause, K. L. "Analysis of Computer Scheduling with Memory Constraints," Ph.D. Dissertation, Computer Sciences Dept., Purdue University, July, 1973.

7. Abell, V. A., S. Rosen, and R. E. Wagner, "Scheduling in a General Purpose Operating System," Proceedings FJCC (37) 1970, (89-96).