Comp. & Maths. with Appls., Vol. 3, pp. 1-13. Pergamon Press 1977. Printed in Great Britain

FORMALIZATION AND CONFIRMATION OF THE BOYD-EPLEY OPERATING SYSTEM MODEL

BERNARD P. ZEIGLER*
Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel and Logic of Computers Group, Dept. of Computer and Communication Sciences, University of Michigan, Ann Arbor, Mich. 48104, U.S.A.

and

DANIEL EWENCZYK
Honeywell-Ball, 94, avenue Gambetta, 75020-Paris, France

Communicated by E. Y. Rodin

(Received December 1975, and in revised form October 1976)

Abstract—An approach to operating system modelling is described in which the simultaneous and synergistic application of tractable queueing theory models and their simulatable refinements is essential. Cross checking of these models against each other and against the real system results in a degree of confirmation not achievable by relying on either analytic or simulation models alone.

1. INTRODUCTION

Models of operating systems abound in the literature[3, 7, 8, 11, 21]. What distinguishes the model proposed by Boyd and Epley[2] is its analytical simplicity derived as a consequence of its authors' attempt to capture, in abstract and general form, the main structural features of batch operating systems (OS). Because of its tractability, the Boyd-Epley model is able to establish certain simple queueing theoretical relationships among such essential quantities as flow rates, queue sizes and service times. Boyd and Epley propose six primary measurements to be made on a real OS, on the basis of which the model is able to predict all the remaining quantities. However, in practice, these primary measurements may not be directly measurable or even easily computable from available data collecting programs. In this case, a simulation model may be designed which is calibratable on the available data and whose output statistics include those required as input by the Boyd-Epley model. Of course, such a simulation may also produce estimates of the quantities predicted by the Boyd-Epley model, thus seeming to nullify its utility. However, in the first place, the Boyd-Epley predictions serve as a check on the correctness of the simulation results (simulation credibility being a constant concern in complex models). In the second place, agreement between the predicted and simulation results serves to confirm the appropriateness of the Boyd-Epley abstraction and encourages the design of observer programs which can collect the data required for its direct application.

This paper reports on an investigation into the validity of the Boyd-Epley model with respect to a class of real-life OS's (exemplified by the MVT system of the IBM 370/165 under HASP[10]). The approach is to construct a more refined model which preserves the basic structure of Boyd-Epley but in addition includes more realistic features such as allocation and scheduling on a priority basis, job class designation, and OS initiators. The more refined model is called a base model relative to the original Boyd-Epley, the latter now being referred to a lumped model (following the general modelling concepts introduced by Zeigler[15]). The formalism employed to characterize the models is that of discrete event specified systems[22], which enables concise theoretic specification of many systems and in particular of OS models.

A three-way comparison of the behaviours of the base, lumped and real system was undertaken. The base model was simulated in GPSS and calibrated against statistics gathered

*Part of this work was supported by NSF Grant No. DCR71-01997 while the first author was with the Department of Computer and Communication Sciences, The University of Michigan, Ann Arbor.

from the operation of the Technion Computing Center (Haifa, Israel, 1972). By adjusting two parameters (CPU overhead expressed as a percentage of service time, and mean I/O service time) it was found that a good fit could be obtained for overall CPU and I/O utilization rates and moreover the qualitative model behaviour was in agreement with the system operator experience.

Statistics gathered by GPSS simulation of the calibrated base model at equilibrium were analyzed and excellent agreement was obtained in comparison with those predicted by the lumped model. It was found, however, that the notion of equilibrium stated by Boyd and Epley had to be modified in order to make their predictions applicable. It is to be noted that this essential assumption of Boyd–Epley can not be tested within the model itself but can be examined with the help of the more refined simulation model.

The main point of the paper is not that we claim to provide yet another simulation model of an OS, but that we provide a methodology for OS modelling based upon the simultaneous and synergistic application of (1) simple analytic relationships (Boyd–Epley); (2) more refined simulatable models; (3) the cross checking of the lumped and base models against each other and against the real system. The results of such a methodology is a measure of validation and credible application of both models, not achievable by each one separately. An essential feature of the proposed methodology is use of the discrete event formalism for precise model description.

2. THE BOYD-EPLEY MODEL

We shall briefly describe the Boyd and Epley model and its assumptions. Then, we shall describe a more refined model suitable for simulation testing against a real OS.

Boyd and Epley divide the system resources into two categories: permanent resources and demand resources. The permanent resources are those resources required by a job step before its execution can begin. Among these are the memory space, the I/O devices which cannot be shared by several job steps and the data files owned exclusively by the job step. The permanent resources are retained by a job step throughout its whole execution whereas demand resources such as CPU units and I/O channels are requested only when needed and released after the service has been completed.

The "unit of program" is the step. No difference is made between the different steps of a job and the steps of other jobs. The names "step" and "job" will be used interchangeably in the following sections.

The main hypothesis made by Boyd and Epley is that a fixed number of jobs is always in the system (this will be discussed later). Thus, any time a step departs from the system, a step from the job queue enters the system and queues for permanent resources. Once these resources have been allocated, the step oscillates between CPU and I/O request and execution. When the last CPU task has been completed, the step releases the resources it details and exits from the system.

The job flow through the system is represented in Fig. 1. An I/O service is limited to I/O's from/to secondary storage. Therefore the system includes neither the input of jobs from card decks nor the output of the results through line printers or card punches.

At a given time a step might either be waiting for permanent resources, waiting for a CPU or executing a CPU task, waiting for an I/O channel to be free or receiving an I/O service. This led Boyd and Epley to decompose the system into 3 stages:

—the Permanent Resources Subsystem (PRS) where permanent resources are allocated.
—the CPU stage, organized as one queue and multiple servers in the case of multiprocessing.

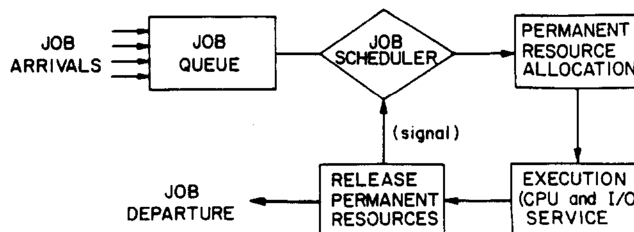

Fig. 1. Job flow through a batch processing multiprogramming system.

—the I/O stage, also organized as one queue and multiple servers, each server being a channel.

The CPU stage and the I/O stage compose the Demand Resource Subsystem (DRS).

The three-stage model is depicted in Fig. 2. Notice that the job queue is not included in the model. Boyd and Epley considered this queue as an infinite source of steps, letting a step go into the system each time another step came out. The system proper includes only the three stages defined above: the permanent resource stage and the CPU and I/O stages. In a spooling system like an OS under HASP, this means that we do not take into account the input of programs through the card reader nor their output through card punches or line printers. Jobs are considered waiting in a job queue in secondary storage before they enter the system, and the output they left on secondary storage for HASP to deal with is neglected. Furthermore no overlapping between CPU and I/O service is allowed for a given step.

The fundamental assumption made by Boyd and Epley is that at equilibrium *the number of steps in the system is always constant.* This implies that there will always be enough jobs in the job queue.

In OS/370 MVT a number of "initiators–terminators" are each assigned a certain group of jobs according to their characteristics known through Job Control Language cards. To each initiator there corresponds an input queue, and whenever an initiator terminates a job, it initiates another one from its queue. In a HASP environment, this is complicated by the fact that these queues are always almost empty, because HASP sends jobs whenever needed or rather, slightly before.

The saturation for such a system is reached when all the initiators are busy, and the system remains at its saturation point only if there always remain jobs waiting for initiation in each of the initiation queues. For this it is sufficient that the following two conditions be simultaneously met:

1. Incoming jobs must have sufficiently varied characteristics so that they are distributed among all the initiators.
2. In each group the arrival rate must be sufficiently great in order to keep all the initiators busy.

These two conditions are very restrictive, and determining if they are satisfied in reality is very difficult.

We shall define as equilibrium the state where the *mean* number of jobs in the system is constant. We shall see through simulation that this condition is easily met, and more realistic than the equilibrium definition of Boyd and Epley. Furthermore we shall include the job queue, made up of the initiation subqueues, into our system. The distribution of incoming jobs will then be considered, since they are expected to have a direct influence upon the system behavior.

In the Boyd–Epley model, steps have no priorities whatever. However, in OS/370 MVT, the priority requested by a user on his JOB card together with other parameters is transformed by HASP into an internal priority according to an algorithm taking into account all parameters. This internal priority is the job or step priority in the PRS and in the I/O stage, but by no means in the CPU stage[10].

Each time a job requests a CPU service, the time slicing feature of the IBM 370 is used, in order to compute a new CPU stage priority for the job. The priorities of user tasks within a given priority group are thus dynamically changed in order to give a higher priority to those tasks, within the group, which use the least amount of CPU time. At the Technion there is only one priority group including all the steps, thus allowing I/O bound steps to remain the smallest possible time in the CPU stage.

Furthermore in a given initiator queue, we may have steps from different HASP classes.

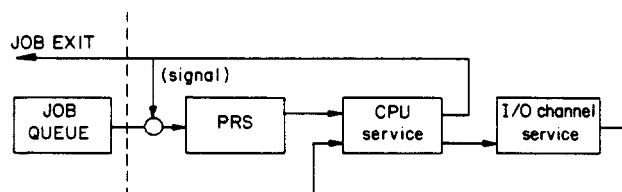

Fig. 2. Three-stage multiple-resource model.

Classes are given different priorities. When an initiator has to select a step for initiation, it scans the queue for the highest priority step within the highest priority class.

3. A MORE REFINED MODEL AND ITS FORMALIZATION

We introduce three new notions into the Boyd and Epley model, namely HASP classes, initiators and step priority. The resulting model, which we call the "base model", is then formalized as a DEVS (Discrete Event Specified System)[22]. The Boyd Epley model is then referred to as the 'lumped model'. The DEVS formalism is a way of concisely describing a simulation model. It has the advantage of being independent of any particular programming language and of not being limited by the ambiguities inherant in an informal natural language description. For more familiarization with this approach to model description the reader is referred to [25].

At the Technion computing center, users have to specify to which class their job belongs. Each class has different maximum requirements, which implies that jobs from different classes have different characteristics. We, therefore, consider it important to differentiate jobs or steps according to the class to which they belong.

The priority of a step and the possibility that a step has to preempt another step to get CPU service are also thought to be important determiners of the system behaviour (simulation results will show that as far as only flow times and throughput are concerned this is not true).

We shall not fix the number of steps in the system as constant. It will only be less than or equal to the number of initiators. We shall provide for the possibility of empty initiation queues.

An input step is defined by an identification number, a priority, a memory requirement, a class number and a number of CPU tasks to perform. We adopt Boyd's and Epley's realistic hypothesis that a step always requests a CPU service before it leaves the system.

A step keeps its own priority throughout its transit through the system, except for the allocation of an initiator which is done on a class first–priority after basis. There is therefore no dynamic change of priorities for the scheduling of CPU tasks as would be permitted by the time slicing feature just mentioned.

The job queue is now composed of as many queues as there are initiators. We intend to model only the core memory allocation; thus the Permanent Resource Subsystem (PRS) contains only one queue for the memory allocation which is fed by the initiated steps.

In order to have a general model, we shall formally describe a multiprocessor system which means that the monolithic Boyd and Epley's CPU stage will be split into components, the CPU's. However, since we shall validate simulation results against a real OS running on a monoprocessor system, only one processor has been taken into account in the simulation program.

The CPU state is organized as one queue and as many servers as there are CPU's. Since we deal with a multiprocessor system, we have to chose a scheduling algorithm. Our choice is the following: If a new step enters the CPU stage and no CPU is idle, the incoming step preempts the CPU servicing the smallest priority step among the step currently processed on the condition that the new step has a greater priority. The preempted step is put at the top of the queue of "ready" steps (those steps ready to receive a CPU service).

The I/O stage has a similar organization, i.e. one queue and multiple servers, the servers being the I/O channels, except that no channel preemption is allowed. The channels are identical. We intend to model only disk accesses. Figure 3 depicts the base model structure. (Tape accesses are not modelled because tape mounting represents a human factor which would be hard to evaluate. Furthermore tape statistics were not available at the time of the study).

The formal presentation of the base model is given in the Appendix.

4. SIMULATION OF THE BASE MODEL AND COMPARISON WITH THE REAL SYSTEM

The base model was encoded as a GPSS program* and various simulation experiments were performed. It was found that an equilibrium, as we have defined it, is always reached by the model. Calibration runs were aimed at adjusting the mean duration of the I/O tasks and the CPU overhead (due to the operating system itself and other unknown factors) in order to get a valid

*The DEVS description of the base model obviates the need for a listing of the GPSS program (this is available however in [26]).

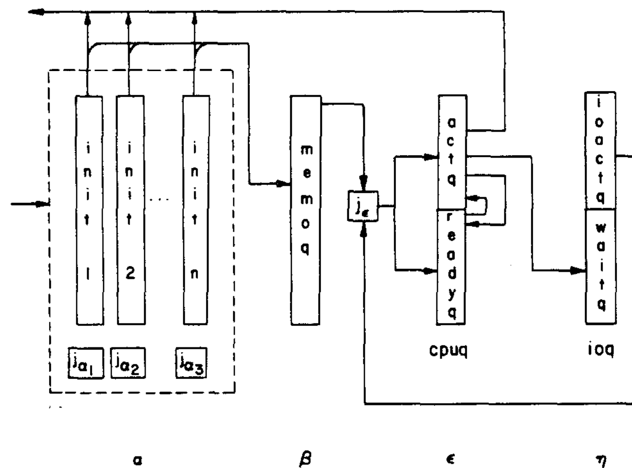

Fig. 3. The step flow through the base model. For symbol references please see the appendix.

mean time elapsed between the beginning and the end of the execution (in the CPU sense) of a step.

It was found however that the variance in this quantity is too large for validation purposes. Instead, it was found possible to match the I/O and CPU utilization of the OS/370 MVT at the Technion.

4.1 *The sources of the data*

There are two sources to the actual data used in the simulation runs: the System Management Facilities files (SMF)[19] and very few runs of AMAP.†

The data are available in the form of tables. We used only the mean CPU times, elapsed times and number of tapes and disks channel programs per step according to HASP classes, the distribution of the core used by the steps and the step priority according to classes.

The AMAP runs have given an idea of the mean duration of a disk access. The actual time to execute a tape channel program seems to vary so greatly that it was decided to simulate only disk accesses.

4.2 *The data used for simulation*

The data were collected from the SMF file of a typical period with total uptime 8.42 hours. We shall only deal with the HASP jobs and steps. During this period 1505 identified HASP jobs were registered which summed up to 12,431 steps. The mean CPU utilization‡ due to these jobs is 0.505.

A table was constructed, giving for each class the number of steps, the relative frequency, the average elapsed CPU time per step and the average number of disk accesses per step. Other tables were constructed, giving for each class the relative frequencies of user step priorities, since statistics about HASP internal priorities could not be obtained.

In addition, a table of the relative frequency of memory size requests was obtained, unfortunately not partitioned by class.

From these tables, distributions were constructed for the arrival of steps, the step class, the allocation of initiators and the allocation of the number of I/O requests.§

Since we had only the total CPU time per step and per class, it was decided to compute for each step of a given class a fixed duration for CPU tasks by dividing the mean total CPU time per step for this class by the mean number of disk accesses for this class plus one.‖

†No references: "for IBM internal use only".

‡"utilization" refers to *utilization ratio*—the ratio of the time the CPU is engaged in processing jobs to the total length of the period.

§The number of I/O requests could only be found as a mean related to the job class. Since we had no information about its distribution around the mean, we assumed identical values for different jobs of a same class. This applies to the CPU service duration.

‖The last service given to a job before it leaves the system is a CPU service. Therefore the number of CPU services of a given job is equal to the number of I/O services plus one.

The I/O duration has been chosen as a class-independant constant EXCPL. Since we have neglected all the I/O activity due to the input and output of the jobs as well as the tape accesses, we made runs with varying EXCPL in order to calibrate the system.

Another calibration parameter is CPOVH, the CPU overhead due to all unknown factors such as the operating system itself, the time-sharing system (TSO), the operators, etc... The duration of each CPU task is multiplied by the factor $1 + \text{CPOVH}/100$.

EXCPL and CPOVH are constant within each simulation run. Seventy-six runs were performed with different values of these two calibration variables. Since EXCPL and CPOVH have unknown values in the real system, no physical meaning should be attached to these two parameters.

4.3 *Simulation results*

The time unit was the hundredth of a second. A series of runs has been performed with different values of EXCPL and CPOVH. Each run simulated a real duration of 200,000 time units, i.e. 2000 seconds. A first run of 100,000 t.u. allowed the system to reach its equilibrium in the sense of a constant mean number of executing steps. After resetting the accumulated statistics, a second run of the same simulated length allowed the collection of statistics.

Between 250 and 450 steps flowed through the system depending upon the calibration parameters. The class distribution was found statistically compatible with the class distribution given as input to the program.

The statistics of the queues of the system (including both PRS and DRS) were output each 10,000 time units, i.e. each simulated 100 seconds. *It appears that the mean number of steps in the system stabilized itself rapidly to a constant dependent on both EXCPL and CPOVL* (Fig. 4). In contrast, the *number of steps in the system was not constant* as assumed originally by Boyd and Epley. This was to be expected because of the input queues addition to the model.

A study was made of the effect of CPOVH and EXCPL on CPU utilization in the model. Since the actual CPU utilization was available (as 0.505) it was possible to conclude that the appropriate parameters settings lay in the region of CPOVH = 50% and EXCPL = 40 (400 ms). Confirmation of this calibration was obtained from the following: At this parameter setting, the mean number of engaged initiators in the model (4 out of 15) agrees with the systems operators' experience. Also when the available memory is increased in the model, it shifts from memory boundedness to non-boundedness. This is also in line with management expectations for the real system.

4.4 *Comparison between base model results and the predictions made by Boyd and Epley*

We analysed a particular case in which the mean elapsed time obtained by simulation is close to the real characteristics (about 1430 t.u.).

The notations will be those adopted by Boyd and Epley in their article, except for the mean number of CPU tasks which we shall denote by $\bar{z}$. MEMOQ, REDYQ, WAITQ are the queues

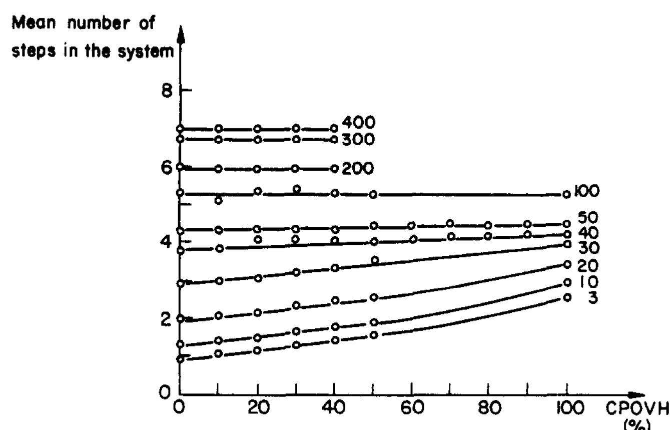

Fig. 4. Mean number of steps in the system vs CPOVH for fixed values of EXCPL.

depicted in Fig. 3. We shall call "flow-time" in a stage or in a queue, the time a step spent in this stage or queue.

All the data gathered during the simulation run appear in the left column of the following table, as well as characteristics easily computed from the former. Boyd and Epley proposed a minimal set of measurements. These appear numbered from 1 to 6. We have added a seventh measurement (2′) to account for CPU preemption. On the right column are corresponding values predicted by Boyd and Epley. It can be seen that the agreement is well within 5% (relative error). The simulation results are averages of 4 seeded runs. Student t-statistics computed fell well within the 5% confidence limits.

$$\text{EXCPL} = 45$$
$$\text{CPOVH} = 30$$
$$Sc = 1 \text{ (one CPU)}$$
$$S_I = 2 \text{ (two channels connected to disks).}$$

		Simulation Results	Boyd's and Epley's Model prediction
	mean number of steps in the system	$N = 4.25$	4.25
(1)	mean time in the system	$F = 1428$	
	mean time between step exits	341	336
PRS			
	mean number of steps in MEMOQ	$N_R = 0.04$	
	mean time in MEMOQ	12.05	13.4
DRS			
(5)	mean total flow time in DRS per step	$F_S = 1433$	1415
-CPU stage			
	mean number of steps in REDYQ	$L_c = 0.20$	0.20
	mean ready time	$W_c = 4.31$	4.3
	mean number of active or preempted steps	0.49	
	mean task duration	$\bar{c} = 9.93$	
	mean task + preemption duration	$\bar{c}' = 10.6$	10.6
	mean number of CPU tasks	$\bar{z} = 15.6$	
	CPU utilization	$\rho_c = 0.46$	
(2)	mean total CPU service time per step	$\hat{c} = 155$	
(2′)	mean total CPU service + preemption time per step	$\hat{c}' = 165.5$	
	mean flow time in CPU stage per step	233	
-I/O stage			
	mean number of active steps	1.93	
	mean number of steps in WAITQ	$L_I = 1.60$	1.62
	mean wait time in WAITQ	$W_I = 37.2$	37.2
(4)	mean number of I/O requests	14.6	
	mean I/O task time	$\bar{I} = 45.0$	45.0
	channel utilization	$\rho_I = 0.98$	
(3)	mean total I/O service time per step	$\hat{I} = 656.0$	
	mean flow time in I/O stage per task	$F_I = 82.2$	82.2
(6)	mean total flow time in I/O stage per step	1200	

5. CONCLUSIONS

A base model, constructed by refining a model suggested by Boyd and Epley was confirmed as a valid model of a real operating system. The queueing statistics obtained by simulation of the base model at equilibrium agreed well with those predicted by the Boyd–Epley lumped model, even though the latter does not take into account the initiators, classes or priorities of the base model. The concept of equilibrium however had to be modified to enable the use of the lumped model predictions. The potential use of both the lumped and base models for general application is thus confirmed.

Note should be taken however of our use of the word "confirmed". We mean that as far as the tests we were able to carry out, no discrepancy was observed in the predictions of the simulated

base model, the Boyd–Epley lumped model and the observed real system behavior. In the comparison of simulation results with operating system data, we explored the (EXCPL, CPOVH) parameter space and noting its regularities, we were able to pin down the acceptable points in this space to a small region. The criterion for acceptability was the matching of CPU and I/O utilization. The calibration was then qualitatively confirmed by comparing initiator utilization and the effect of memory size changes with the reports of facility operators.

Clearly *validation of the base model*, as opposed to confirmation, would require a much more extensive series of tests. Unfortunately, the acquisition of live operating system data is not a trivial matter and pertinent data are not available to us. Fortunately however, the essential measurements required to validate the models are known (namely the seven characteristics shown numbered in the foregoing table) and it is thus possible to guide the development of software observer systems toward incorporating capabilities for acquiring the necessary primary data. In the present circumstances, the proposed methodology of simultaneously working with both the analytically simple Boyd Epley and the more complex simulation model is necessary to achieve results not possible with either alone. We may expect the general principle of multilevel combined analytic–simulation modelling to become even more important in the future.

Our experience has also suggested further refinements which should improve the chances of validation. We now feel that the single I/O queue of the present base model should be split into several queues, each one corresponding to one channel. The data required as input should include tape accesses statistics such as tape mounting dely, access duration distribution, number of accesses per mounted tape distribution, etc. Time-sharing and APL sessions should be treated as normal HASP jobs. The calibration parameter CPOVH would thus really represent the CPU overhead due to the operating system itself. Such an enhanced model could be used to determine the influence of adding I/O channels, main memory, disk or tape units on computing system performance. We note that the foregoing modifications are refinements of the base model and so would not go beyond the essential framework of this paper.

REFERENCES

1. D. W. Barron, *Computer Operating Systems*. Chapman and Hall, London (1971).
2. D. L. Boyd and D. L. Epley, A simple model for multiple-resource allocation in operating systems, in *Computers and Automata* (J. Fox, Ed.). Wiley, New York (1971).
3. P. S. Cheng, Trade-driven system modelling, *IBM Systems J.* **8**, 4 (1969).
4. A. J. T. Colin, *Introduction to operating systems*. MacDonald/American Elsevier Computer Monographs (1971).
5. G. Cuttle and P. B. Robinson, *Executive Programs and Operating Systems*. McDonald/American Elsevier Computer Monographs (1970).
6. J. A. Cooperman and W. H. Tetzlaff, *Analysis in an OS user environment*. IBM Research RC3161 (1970).
7. H. A. Ernst, *Control Program Modelling Techniques*. IBM Research RC 3061 (1970).
8. H. A. Ernst, *Problems in Computing Systems Performance Modelling: characterization, calibration and validation*. IBM Research RC3319 (1971).
9. *General Purpose Simulation System V*. IBM Corp. SH20—0851—1 (1971).
10. *Houston Automatic Spooling Priority*. IBM Corp. 360D—05.1.014.
11. J. H. Katz, An experimental model of system/360, *Communs Ass. comput. Mach.* **10**, 11 (1967).
12. H. Katzan Jr., *Computer organization and the system* 370. Van Nostrand Reinhold (1971).
13. M. N. MacDougall, Computer system simulation: an introduction. *Computing Surveys* **2**, 3 (1970).
14. G. H. Mealy, B. I. Witt and W. A. Clark, The functional structure of OS/360, *IBM Systems J.* **5**, 1 (1966).
15. W. E. Riddle, *Hierarchical Modelling of Operating System Structure and Behavior*. Dept. of Computer and Communication Sciences, University of Michigan.
16. R. F. Rosin, Determining a Computing Center Environment, *Communs Ass. comput. Mach.* **8**, 7 (1965).
17. H. Schorr, Design principles for a high performance system, in *Computers and Automata*. Wiley, New York (1971).
18. P. H. Seaman and R. C. Soucy, Simulating Operating Systems, *IBM Systems J.* **8**, 4 (1969).
19. *System Management Facilities*. IBM Corp. GC 28–6712—6 (1972).
20. The Comtre Corp. *Operating Systems Survey* (A. P. Sayers, ed.). Auerbach (1971).
21. G. Waldbaum and H. Beilner, *SOUL: A Simulation of OS under LASP*. IBM Research RC3810 (1972).
22. B. P. Zeigler, Discrete Event Specified Systems, *Proceedings of the Eight Annual Princeton Symposium on Information Science and Systems* (1974).
23. B. P. Zeigler, Postules for a Theory of Modelling and Simulation, *Proceedings of the* 1975 *Summer Computer Simulation Conference*. AFIPS Press, Montvale, N.J. (1976).
24. B. P. Zeigler, A Conceptual Basis for Modelling and Simulation, *Internat. J. gen. Syst.* **1**, 4 (1974).
25. B. P. Zeigler, *Theory of Modelling and Simulation*. Wiley, New York (1976).
26. D. Ewenczyk, *Formalization of the Boyd and Epley Three-Stage Model of a Multiple Resource-Allocation Operating System*. MS Thesis, Computer Science Dept., Technion, Israel (1973).

APPENDIX. FORMAL DESCRIPTION OF BASE MODEL

We present the formal description of the base model. The following mnemonics are employed:

c	step class
DRS	demand resource subsystem
f	free memory
gc	number of HASP classes
gp	greatest priority
i	step identification number
in	initiator ordinal number
I^+	set of all positive integers
m	step memory requirement
$matu$	memory available to users
n	number of initiators
p	step priority
PRS	permanent resource subsystem
r	number of CPU tasks of a step
Sc	number of CPU's
S_I	number of I/O channels
x	step
Λ	empty queue

Presentation of the formal base model

The model we propose here is a Discrete Event Specified System[22]

$$M = \langle X, S, Y, \vartheta, \star, t\rangle$$

where X is the input set, S is the state set and Y the output set.

$t: S \to Re$ where $Re = R^+ \cup \{0\} \cup \{\infty\}$ and $R^+ = \{r|r > 0\}$. $t(s)$ is the maximum time the system is allowed to stay in its present state s. We define the set Q as

$$Q = \{(s, e)|s \in S, 0 \le e \le t(s)\}$$

where e is the time elapsed since the last state change. We shall call it the "elapsed time".

Then the transition specifying function is the mapping

$$\sigma: Q \times X \to S$$

where, if we call $\phi \in X$ the "non-event", and $X - \{\phi\}$ the set of external events, there is an autonomous transition function δ_ϕ such that

$$\vartheta(s, e, \phi) = \delta_\phi(s) \quad \text{where } \delta_\phi : S \to S$$

$\star: Q \to Y$ is the output function.

Such a structure M specifies a system in the same way that differential equations or sequential machines do (see [22] and [25] for explications).

The input set X

$$X = I^+ \times A \times B \times C \times D \times E$$

- I^+ set of identification numbers. It is assumed that at a given time there cannot be two steps with the same identification number.
- A set of priorities $A = \{1, 2, \ldots, gp\}$ where gp is the greatest step priority.
- B set of memory requirements $B = \{1, 2, \ldots, matu\}$ where $matu$ is the memory available to users, that is the amount of core memory which is not occupied by resident programs.
- C set of classes $C = \{1, 2, \ldots, gc\}$ where gc is the number of step classes. The class of a given step remains constant.
- $D = I^+$ number of CPU tasks. A step enters the system with a number $r \in I^+$ of CPU tasks to perform during its life in the system. Since we assume that the last task of a step is always a CPU task, the number of I/O requests is $r - 1$.
- E set of initiator numbers. We assume that steps have already been allocated an initiator number in when they enter the system, $E = \{1, 2, \ldots, n\}$

Let x be an input job step.
$x \in X - \{\phi\} \quad x = (i, p, m, c, r, in)$
i = step identification number
p = priority
m = memory requirement
c = class
r = number of CPU tasks
in = initiator number.

We shall denote by x_i a step the first component of which is i and by p_i its priority, m_i its memory requirement, c_i its HASP class, r_i the number of CPU tasks it will perform during its life in the system, and in_i the particular initiator it is assigned to. Since at a given time there is at most one step associated with a given identification number, the notation x_i will not lead to confusion.

We shall also denote by $p(x)$ the priority of the step x; $m(x)$, $c(x)$, $r(x)$ and $in(x)$ have the same straightforward meaning.

The state set S

We shall define a set of components $D = \{\alpha, \beta, \epsilon, \eta\}$.

Each component, say α, will be made up of a component state X_α and a scheduling time $\sigma_\alpha \in Re$. A component activation may be interpreted as a set of actions performed on the state that modified the latter. This may include the modification of the scheduling time of the same or other components and the possible "immediate reactivation" of the current component.

According to Zeigler [9], we shall define as "imminent event" a member of the set of events such that their scheduling time be equal to $t(s)$. A particular case is obtained if $t(s) = 0$. A selection rule has to be defined if there are several imminent events, i.e. several events scheduled to occur at the same time.

The components of the base model will be

α: the step queue

β: the permanent resource subsystem

ϵ: the CPU stage

η: the I/O stage.

A state of M will then be $(x_\alpha, \sigma_\alpha, x_\beta, \sigma_\beta, x_\epsilon, \sigma_\epsilon, x_\eta, \sigma_\eta)$. Then $t(s) = \min(\sigma_\alpha, \sigma_\beta, \sigma_\epsilon, \sigma_\eta)$ and the imminent events will be defined by the following mapping: IMM: $S \to 2^D$, $\text{IMM}(s) = \{\alpha | \alpha \in D | \sigma_\alpha = t(s)\}$.

Component α: the initiation queues

Let n be the number of initiator-terminators.

We shall have n queues, one per initiator. By convention, the top entry of a given queue is the step currently seizing the corresponding initiator. If the queue is empty (symbol Λ) the initiator is idle. The state set of α is $(X^* \times (X \cup \{\Lambda\}) \times Re)^n$, where X^* is the set of all finite length sequences of elements of x, Λ being the sequence of length 0; n the number of initiators.

α is made up of n subcomponents α_{in}, one per initiator. A typical state of α_{in} is

$$(\text{init}_{in}, j_{\alpha_{in}}, \sigma_{\alpha_{in}})$$

init_{in} is the inth queue of the component α.

In the model, one of the parameters of incoming steps is an initiator number. We, therefore, do not consider allocation of initiators internal to the model. This feature appears in the specification of input sequences where steps of a given class are given equal probabilities to join any of the initiation queues assigned to this class.

$j_{\alpha_{in}}$ is a saving place where an incoming step is stored as a result of an external event.

$\sigma_{\alpha_{in}}$ is the scheduling time of the subcomponent α_{in}.

An incoming step is immediately inserted into the corresponding queue, and if the latter is empty, directly passes to the PRS. α_{in} may also be activated when a step exits the system. If $\text{init}_{in} = \Lambda$, i.e. no step is waiting for initiation, then α_{in} deactivates itself without further action.

Component β: the PRS (Permanent Resource Subsystem)

The state set is is $(B \cup \{0\}) \times X^* \times Re$.

A typical state is $(f, \text{memo}q, \sigma_\beta)$ where f is the free memory, memo$q$ is the memory queue where steps queue in after having seized an initiator. It is organized on a priority basis. Whenever β is activated it reactivates itself unless no memory can be allocated to the top entry of memo$q$. This is to provide for the case of the release of a large amount of memory by a step. This memory may possibly be allocated by fragments to several smaller steps on the condition that they are at the top of memo$q$.

β may either

—allocate memory to the top entry of memo$q$, send this step to the CPU stage and immediately reschedule itself and ϵ, the CPU stage.

—or do nothing if no memory can be allocated.

Component ϵ: the CPU stage

The state set is

$$(X \times (R^+ \cup \{0\}))^* \times (X \cup \{\Lambda\}) \times [0, 1]^{\#c} \times Re.$$

A typical state is $(\text{cpu}q, j_\epsilon, \gamma^1, \sigma_\epsilon)$.

We assume the CPU stage contains Sc CPU's which are either busy or idle. We shall have one step queue the Sc top entries of which (if they exist) are the steps currently receiving CPU service. Since we allow preemption the steps which are not being processed by one of the CPU's are either preempted steps or have recently entered the stage and are waiting for the beginning of the execution of the task. We shall call the queue of all the steps in the stage cpu$q$, the subqueue of all waiting steps ready$q$, and act$q$ the subqueue of active steps, i.e. those currently receiving CPU service.

Formally act$q$ can be defined by

$$\begin{cases} \text{cpu}q = \text{act}q\ \text{ready}q \\ \text{length}(\text{act}q) = \min[\text{length}(\text{cpu}q), Sc] \end{cases}$$

where Sc is the number of CPU's and length the mapping

$$\text{length}: (X \times R^+)^* \to I^+ \cup \{0\}$$

length (cpu$q$) is the length of the queue cpu$q$.

Each time a step enters the CPU stage, it is given a processing time tc from a class dependent distribution $H_c(\gamma_c^1)$ where $c \in C$ is the class of the step, and γ_c^1 is a uniformly distributed random number in the range [0, 1]. The step is then inserted into cpu$q$ possibly causing a preemption.

j_ϵ is the value stored in a save cell which may contain a step coming either from the PRS or from the I/O stage, or be empty (Λ).

Let k_1 denote the number of steps currently in the CPU stage. Let $x_1, x_2, \ldots, x_{k_1} \in X$ be steps and $tc_1, tc_2, \ldots, tc_{k_1} \in R^+$ be positive real numbers. tc_i is the processing time there remained to the ith step of the queue after the completion of the last event. Notice that here x_i does not mean the step the identification number of which is i, but the ith step of the queue. A typical cpu$q$ where all CPU's are busy will have the following aspect.

$$\text{cpu}q \begin{cases} \left.\begin{matrix} (x_1, tc_1) \\ \vdots \\ (x_{Sc}, tc_{Sc}) \end{matrix}\right\} \text{act}q \\ \left.\begin{matrix} (x_{Sc+1}, tc_{Sc+1}) \\ \vdots \\ (x_{k_1}, tc_{k_1}) \end{matrix}\right\} \text{ready}q \end{cases}$$

As mentioned earlier CPU's are preemptible. Our scheduling and preemption algorithm is the following: an incoming step (the value of j_ϵ) will either seize an idle CPU, either preempt another step if no CPU is idle and its priority is high enough, or be inserted into ready$q$ if its priority is too low. This will be formally described further down.

ϵ may also transfer a step which has completed its current CPU task to the I/O stage or purge the system from this step if the completed CPU task was its last one. In this last case the permanent resources retained by the step are given back to the PRS and β is activated.

Component η: the I/O stage

The state set is

$$(X \times (R^+ \cup \{0\}))^* \times [0, 1]^{\#c} \times Re.$$

A typical state is (io$q$, γ^2, σ_η) where io$q$ is a queue with an organization similar to cpu$q$. The S_l top entries of io$q$ are the steps currently receiving channel service. We shall call ioact$q$ the subqueue of these steps. If the S_l channels are not all seized the length of ioact$q$ is smaller than S_l. More precisely, if wait$q$ is the subqueue of all the steps waiting for a free channel, then

$$\begin{cases} \text{io}q,\ \text{ioact}q,\ \text{wait}q \in X^* \\ \text{io}q = \text{ioact}q\ \text{wait}q \\ \text{length (ioact}q) = \min\,[\text{length (io}q), S_l]. \end{cases}$$

A fundamental difference between the CPU and the I/O stages is that channels may not be preempted, and steps have to wait for a channel to be free to begin and execute their I/O task.

When a new step enters this stage it is allocated a service time ti from a class dependent distribution $W_c(\gamma_c^2)$ where γ_c^2 is a uniformly distributed random number in the range [0, 1].

A typical io$q$ is given here. k_2 will denote the total number of steps in the stage.

$$\text{io}q \begin{cases} \left.\begin{matrix} (x_1, ti_1) \\ \vdots \\ (x_{S_l}, ti_{S_l}) \end{matrix}\right\} \text{ioact}q \\ \left.\begin{matrix} (x_{S_l+1}, ti_{S_l+1}) \\ \vdots \\ (x_{k_2}, ti_{k_2}) \end{matrix}\right\} \text{wait}q \end{cases}$$

Notice that $k_2 \leq S_l \Rightarrow$ wait$q = \Lambda$. η may be scheduled by ϵ. When activated, η sends the step which caused the activation to the CPU stage. ϵ is then activated.

Selection rule

We might have several imminent events (recall previous page). To avoid information loss, we adopt the following rules: rules:

—whenever ϵ and β are both imminent, ϵ will be executed before β.

—whenever ϵ and η are both imminent, ϵ will also be executed first.

The reason for these two rules is that we do not want to overwrite the contents of j_ϵ. This would happen if the allocation of permanent resources (β) or the exit from the I/O stage (η) were done before the insertion into cpu$q$ (ϵ).

Based on the above description, the transition function δ is formalized as a set of tables, accessed under control of the time advance and selection rules.

The transition function δ

$$\delta Q \times X \to S.$$

The transition function is represented as a double entry table where the actions of each component are represented. To help simplify this table let us define a few operators.

top: $X^* \to X$

top (qn) is the top entry of the queue $qn \in X^*$.

rest: $X^* \to X$

rest (qn) is the queue qn without its top entry $qn = \text{top}(qn)\ \text{rest}(qn)$.

insert: $X \times X^* \times I^+ \to X^*$

Insert (x, qn, p) is the queue obtained by inserting the step x given a priority p into the queue qn. The insertion is made on a priority basis. There is no redundancy in the fact that one of the parameters of x is its priority, and the definition of insert. The reason for including a priority parameter in insert is that the insertion of steps into the initiation queues is done on a class first-priority after basis. A step $x = (i, p, m, c, r, in)$ is not inserted according to its own priority p but according to a different priority $b(c, p)$ which also takes the HASP class into account. b is the following mapping

$$b: C \times A \to A.$$

As an example, the insertion of a step x_i into the initiation queue in yields the new queue

—x_i if $\text{init}_{in} = \Lambda$ (empty queue)
—insert $[x_i, \text{rest}(\text{init}_{in}), b(c_i, p_i)]$ if $\text{init}_{in} \neq \Lambda$.

pop: $X \times (X \times R^+)^* \to (X \times R^+)^*$

pop (x, qn) is the queue obtained when the step x pops off the queue qn. An important assumption is that there never be holes in queues. These must be thought of as linear linked lists or chains. This and our assumption that the S_t top entries of the k_2 elements of io$q$ are receiving channel service gives to pop $(x_i, \text{io}q)$ the following meaning. It is the queue resulting from the two actions

—the step x_i (the identification number of which is i) pops off io$q$
—top (wait$q$), if any, becomes the S_tth entry of io$q$ and therefore begins to receive channel service.

preempt$_t$: $X \times (X \times R^+)^* \to (X \times R^+)^*$

preempt$_t$ $(x, \text{cpu}q)$ is the new cpu$q$ resulting from the arrival of a step x into the CPU stage when each CPU is siezed $(k_1 \geq Sc)$, t units of time after the last transition.

Let x_j be the jth step in act$q$ with minimum j such that

$$p_j = \min(p_1, p_2, \ldots, p_{Sc}).$$

Here again p_j mean the priority of the jth step of act$q$.

(1) if $p > p_j$ then ready$q$ becomes $[x, H_{c(x)}(\gamma^1_{c(x)})][x_j, tc_j - t]$ ready$q$ and act$q$ becomes pop $(x_j, \text{act}q)$ where $c(x)$ is the class of the step x. This is called preemption.
(2) if $p \leq p_j$ then ready$q$ becomes insert$^*\{[x, H_{c(x)}(\gamma^1_{c(x)})], \text{ready}q, p(x)\}$ where insert* is an extension of the mapping insert: and, insert*: $(X \times R^+) \times (X \times R^+)^* \times A \to (X \times R^+)^*$ where insert* has the same effect than insert except that it operates on queues of the set $(X \times R^+)^*$ instead of X^*.

$'_t$: $(X \times R^+)^* \to (X \times R^+)^*$

actq'_t is the queue act$q$ the entries of which have had t subtracted from their second projection. For example

$$(x_1, tc_1, x_2, tc_2)'_e = (x_1, tc_1 - e, x_2, tc_2 - e)$$

The transition function δ (tables)

We shall first describe the transition caused by an external event, i.e. the input of a step. This step $x = (i, p, m, c, r, in)$ is stored into the saving cell $j_{\alpha_{in}}$ of the subcomponent α_{in} (the initiator number in). α_{in} is then activated by giving $\sigma_{\alpha_{in}}$ the value 0.

The entries of the Table A1 give $\delta(s, e, x)$. Blank entries mean unmodified parameters.

Tables A2 and A3 give the transition function $\delta_\phi(s)$, that is the law governing the system when the latter is working like an autonomous machine.*

Table A1. Transition function δ

The following components are possibly modified		External event x $x = (i, p, m, c, r, in)$
α_{in}	init_{in}	
	$j_{\alpha_{in}}$	x
	$\sigma_{\alpha_{in}}$	0
β	f	
	memo$q$	
	σ_β	$\sigma_\beta - e$
ϵ	cpu$q$	actq'_e ready$q$
	j_ϵ	
	γ^1	
	σ_ϵ	$\sigma_\epsilon - e$
η	io$q$	ioactq'_e wait$q$
	γ^2	
	σ_η	$\sigma_\eta - e$

*Due to space limitations we present the tables only for the α_{in} and β components of the sequential state. For the same reasons the descriptions of the output set and function of the model are omitted. For a complete description, please see reference 26.

Table A2. α_{in}—subcomponent number *in* of the component α

α_{ij}	If $j_{\alpha_{in}} \neq \Lambda$ and $\text{init}_{in} \neq \Lambda$ the step contained in the cell $j_{\alpha_{in}}$ is enqueued in init_{in}	If $j_{\alpha_{in}} \neq \Lambda$ and $\text{init}_{in} = \Lambda$ initiation of the step $j_{\alpha_{in}}$	If $j_{\alpha_{in}} = \Lambda$ a step has terminated	
			Initiator is free	A new step is initiated
init_{in}	insert $[j_{\alpha_{in}}, \text{rest}(\text{init}_{in}), b(c, p)]$	$j_{\alpha_{in}}$		
$j_{\alpha_{in}}$	Λ	Λ		
$\sigma_{\alpha_{in}}$	∞	∞	∞	∞
β				
f				
memo$q$		insert $[j_{\alpha_{in}}, \text{memo}q, p(j_{\alpha_{in}})]$		insert $[\text{top}(\text{init}_{in}), \text{memo}q, p[\text{top}(\text{init}_{in})]]$
σ_β	$\sigma_\beta - t(s)$	0	$\sigma_\iota - t(s)$	0
ϵ and η	The only changed parameters are cpu$q$, σ_ϵ, io$q$ and σ_η which respectively become act$q'_{t(s)}$,ready$q$, $\sigma_\epsilon - t(s)$, ioact$q'_{t(s)}$,wait$q$ and $\sigma_\eta - t(s)$			

Table A3. β—the permanent resource subsystem

α_{in}	init_{in}		
	$j_{\alpha_{in}}$		
	$\sigma_{\alpha_{in}}$	$\sigma_{\alpha_{in}} - t(s)$ $in \in [1, n]$	
β		if $m[\text{top}(\text{memo}q)] \leq f$	else
	f	$f - m[\text{top}(\text{memo}q)]$	
	memo$q$	rest(memo$q$)	
	σ_β	0	∞
ϵ	cpu$q$	act$q'_{t(s)}$,ready$q$	
	j_ϵ	top(memo$q$)	
	γ^1		
	σ_ϵ	0	$\sigma_\epsilon - t(s)$
η	io$q$	ioact$q'_{t(s)}$,wait$q$	
	γ^2		
	σ_η	$\sigma_\eta - t(s)$	

The use of these tables is straightforward. Given the present state s, $t(s)$ gives the transition time. The mapping IMM (see above) gives all the imminent events, one of which is selected with the selection rule. The imminent event thus selected leads to one of the tables, the entries of which yield the new state. For instance, if the selected imminent event is β, we look at Table A3: the new value of *memoq* is rest (*memoq*); the top entry of the previous *memoq* is entered into j_ϵ, etc.

A principle in the reading of all the tables is that the components assume their new value within a null time and only when all the changes have been performed. This implies that the variables appearing as entries are always those pertaining to the state s.