

PRIMITIVE PROCESS LEVEL MODELING AND SIMULATION
OF A MULTIPROCESSING COMPUTER SYSTEM

by

James Wayne Anderson

May 1974

TR-32

This paper constituted the author's dissertation for the Ph.D. degree at the University of Texas at Austin, May 1974.

This work was supported in part by National Science Foundation Grant GJ-1084.

THE UNIVERSITY OF TEXAS AT AUSTIN
DEPARTMENT OF COMPUTER SCIENCES

TABLE OF CONTENTS

CHAPTER		PAGE
1	INTRODUCTION TO THE RESEARCH	
	1.0 Introduction	
	1.1 Related Work	
	1.1.1 Trace-Driven Modeling	
	1.1.2 Directed Graphs	
	1.1.3 Summary	
	1.2 Summary of Chapters	
2	SYSTEM PROCESS GRAPHS	
	2.0 Introduction	
	2.1 Definitions	
	2.1.1 Memories	
	2.1.2 Processors	
	2.1.3 Processes	
	2.1.4 Operating Systems	
	2.2 System Process Graphs	
	2.2.1 Process States	
	2.2.2 Construction of System Process Graphs	
	2.3 Applications in the Simulation of Computer Systems	
	2.3.1 Processes Represented by Graphs in the Model	
	2.3.2 Methodology of a SPG Structured Trace-Driven Model	
	2.3.3 Properties of the SPG's	
3	THE MODELED SYSTEM AND ITS SOFTWARE PROBE	
	3.0 Introduction	
	3.1 System Description	
	3.1.1 Hardware	
	3.1.2 Operating System	

3.1.2.1	Dedicated PPU's	
3.1.2.2	Partitioning of Central Memory	
3.1.2.3	Processes at CP's	
3.1.2.4	Peripheral Processes	
3.1.2.5	Functions	
3.1.2.6	MTR	
3.1.2.7	CPM	
3.1.3	Software Probe	
4	THE SIMULATION MODEL	
4.0	Introduction	
4.1	Construction of System Process Graphs	
4.1.1	Functions	
4.1.1.1	Extended Channel Functions	
4.1.1.2	Parameterization of the Function SPG's	
4.1.2	Peripheral Process Graphs	
4.1.2.1	SPG Builder for Peripheral Processes	
4.1.3	Central Processes	
4.1.3.1	Construction of Job SPG's	
4.2	Operation of the Model	
4.2.1	Degree of Resolution	
4.2.2	CPM and MTR Processes	
4.2.2.1	CPU Scheduling	
4.2.2.2	PPU Scheduling	
4.2.2.3	Job Scheduler	
4.2.2.4	Servicing of RA+1 Calls	
4.2.2.5	Processing of Functions	
4.2.2.6	Traversals of SPG's	
4.2.3	Validation	
4.2.3.1	Job Mix	
4.2.3.2	Comparison of Performance Measures	
4.2.3.3	Validation Summary	

5	EXPERIMENTAL APPLICATIONS	
5.0	Introduction	
5.1	Experiment A - First Fit Memory Allocation	
5.1.1	Memory Allocation	
5.1.2	Experiment Description	
5.1.3	Comparison of Results	
5.1.4	Summary	
5.2	Experiment B - Reduce Punts	
5.2.1	Experiment Description	
5.2.2	Comparison of Results	
5.2.3	Summary	
5.3	Experiment C - CIO-2RD-2WD Overlay	
5.3.1	Experiment Description	
5.3.2	Comparison of Results	
5.3.3	Summary	
5.4	Experiment D - All PP Overlays on ECS	
5.4.1	Experiment Description	
5.4.2	Comparison of Results	
5.4.3	Summary	
5.5	Experiment E - Overlays on System Disk Moved to ECS	
5.5.1	Experiment Description	
5.5.2	Comparison of Results	
5.5.3	Summary	
5.6	Experiment F - Copy of System on Each Disk	
5.6.1	Experiment Description	
5.6.2	Comparison of Results	
5.6.3	Summary	
5.7	Experiment G - Centralized CIO	
5.7.1	Experiment Description	
5.7.2	Comparison of Results	
5.7.3	Summary	

CHAPTER

PAGE

5.8 Experiment H - 13 PPU's
5.8.1 Comparison of Results
5.8.2 Summary

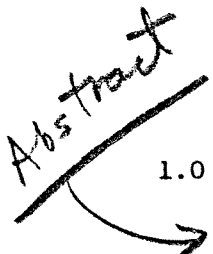
6 SUMMARY

BIBLIOGRAPHY

CHAPTER 1

INTRODUCTION TO THE RESEARCH

Abstract



1.0 Introduction

[This research consists of the modeling and simulation of a real operating system, UT-2 [5,10,12] running on a CDC 6600 computer system, at a level of detail corresponding to the primitive processes and modules in which the system is logically and physically defined.] Model-
ing and simulation at this level of detail has heretofore been regarded as an untractable task. The number of processes in a real system and the complexity of their interactions and interrelationships have precluded simulation at this level of system model definition.

The interactions and interrelationships of processes, as well as their consumption of resources, are determined by the sequences of operations they performed. These sequences can be represented explicitly by state graphs. It is through the use of representations of these state graphs that the problem of primitive process level simulation modeling is made tractable.

The source code for the various processes could be analyzed to produce detailed and complete state graphs. Howard and Alexander [11] have, indeed, done so utilizing an automatic source code analyzer. It is necessary, however, to parameterize the model at a level of detail commensurate with the resolution of the state graphs. The model would be useless due to the absence of validation and baseline performance information if adequately detailed parameterization were not possible.

The availability of an elementary level, event driven trace [12,20] in the UT-2 operating system has provided the mechanism whereby primitive process level modeling and simulation of a real system can be effected with very little compromise in the level of resolution or accuracy. Trace of an elementary process will contain only a small subset of all possible traversals through its potential state graph; i.e., those that actually occurred during the monitored operating period. This subset will be smaller and less complex than the complete state graph and, therefore, more tractable for handling and inclusion in the model. Further, the subset will be complete within the domain of performance modeling. The trace monitor not only provides a practical resolution of the state graphs of the system, it also provides the essential parameters of the model and the required baseline case for model validation.

The initial phase of this research project consists of the construction of an explicit model of the elementary system processes through analysis of trace information recorded by the UT-2 software monitor. This model is composed of directed graphs representing the actually occurring traversals of the state graphs for the various elementary processes. System primitive processes (or functions) are represented as nodes of the directed graphs. The arcs are labeled with branching probabilities and processing time requirements (see Figure 1.1 for an example). The model system was driven from the user level workload recorded by the software monitor. Each user job is itself represented by a directed graph where the nodes are requests for system services and the arcs again are labeled with processing time requirements.

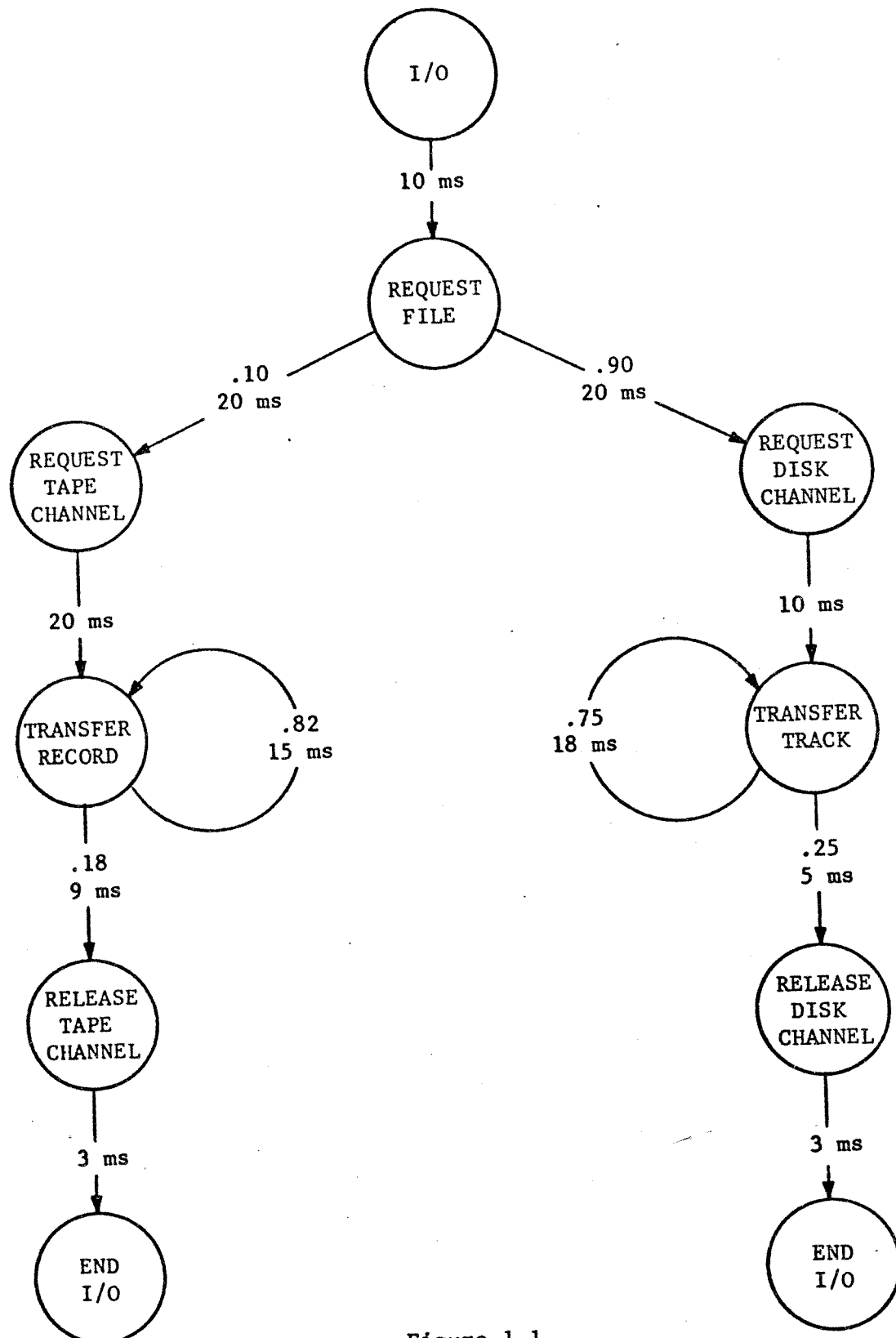


Figure 1.1
 Example of a Directed Graph for Hypothetical System I/O Process

The user requests for system services are mapped onto traversals of the directed graphs for the designated elementary processes.

The design, goals and functions of this trace driven model differs in a number of significant ways from previous models.

- a. The model is defined explicitly in terms of the elementary processes of the system and their interrelationships. It thus preserves the functioning structure of system. Most previous simulation models [1,13,17,18,19] retain only the algorithmic and conceptual structure of the system being modeled.
- b. The model is explicitly graph driven. The representation of the functional structure, therefore, can be readily altered. Such modifications require little or no re-validation of the model.
- c. All usage of system resources is resolved to elementary processes instead of just user jobs or gross level system processes.
- d. The model can provide explicit direction to the systems programmer as well as the system designer and analyzer.

These changes have profound effects on the types of analysis which can be carried out in the context of this model. It is possible to precisely define and isolate either elementary or complexes of elementary functional units for study. The effects of varying implementation and organization of elementary processes of the system can be studied. It is possible to define and effect localized changes and to observe the propagation of these changes to other areas of the

system. The function of well defined system processes can be isolated and optimized.

This resolution at the primitive or functional level contrasts to and represents a conceptual advance in the concept of operating system modeling, even over the detailed trace driven modeling studies of Brice, Sherman, et al. [1,17,18,19].

1.1 Related Work

1.1.1 Trace-Driven Modeling

Previous trace-driven modeling studies have typically concentrated on the design of modeling techniques or the application of trace-driven modeling to some particular problem or area of system performance. Sherman [17] presents a comprehensive state-of-the-art report on trace-driven modeling. The interested reader should refer to Sherman's paper as no point could be seen in repeating that information here.

Sherman [17] also constructed detailed trace-driven models of two operating systems, the UT-1 and the UT-2, and demonstrated the flexibility of the models by investigating several areas of the operating systems as candidates for possible modification. These areas included CPU scheduling, channel scheduling and deadlock prevention or deadlock detection and recovery algorithms.

In an interesting work following that of Sherman, Brice [1] used both a distribution-driven and a highly detailed trace-driven model to investigate the application of CPU scheduling techniques to I/O processing. His paper included a study of the effects of having feedback coupled schedulers for CPU and I/O processing. As a distribution-driven model is relatively cheap to execute in terms of computer resources required, Brice used it to determine parameters, isolate performance trends and to draw preliminary conclusions. The more elaborate trace-driven model was then used to verify these conclusions.

1.1.2 Directed Graphs

The use of directed graphs is not new in the study of operating

systems. Previous studies utilizing them, however, have been concerned with the correctness or analysis of operating systems.

Holt [7] and Hcbalkar [6] represented the resource demands of processes with directed graphs and used these graphs in the study of deadlock prevention, detection and recovery.

Howard and Alexander [11], as mentioned in the previous section, produced state graphs for elementary processes of the UT-2 system. Their primary objective was to develop an automated procedure for verifying that processes obeyed given ordering rules on the sequences of operations that they performed.

A model utilizing directed graphs was presented by Rodriguez [16] for use in the analysis of parallel computations on unstructured data.

Johnson [12] developed directed graphs of system processes through analysis of the information recorded by the UT-2 software monitor. These directed graphs were used to construct a hierarchial graph model of the UT-2 system and to accumulate statistics on the activities of the corresponding processes. This sequential analysis technique allowed considerable insight into the behavior of the processes of the system.

1.1.3 Summary

Johnson's work, more than any other, served as a catalyst for this current study. One of the accomplishments of this study is the application of a technique formerly reserved for correctness and analysis studies of operating systems, that of representing processes by directed graphs, to performance modeling.

1.2 Summary of Chapters

Chapter one introduces the research, reviews some related projects and presents a summary of the chapters.

Chapter two presents the definitions of terms used in the research and develops the conceptual model in which the resource demands by system processes are explicitly represented by directed graphs.

The modeled system and its software monitor are described in chapter three.

Chapter four presents a detailed description of the simulation model including its construction, operation and validation.

The experimental studies are described in chapter five.

Chapter six presents the summary of the research.

CHAPTER 2

SYSTEM PROCESS GRAPHS

2.0 Introduction

When modeling and simulating a computer system at the primitive process level, consideration must be given to the synchronization and other interactions of and among processes, particularly with respect to their use of system resources and the contention among them for those resources. This chapter develops the concept of system process graphs and describes their applicability to system modeling as a means for handling these interrelationships and interactions of processes.

2.1 Definitions

This section presents definitions of terms used in this research. Although many of these terms have been defined often in the literature, [10,12,17], their definitions are included here for completeness.

A computer has been defined as a collection of processors linked to memories, where memories are passive storage devices and processors are elements which act upon their attached memories in a discrete fashion. The term processor is used most generally to refer to a programmed processor, one whose actions are directed by interpreting a program stored in one of its memories. As this infers, a program is a sequence of instructions applicable to interpretation by a computer. A program being interpreted by a processor constitutes a process. An operating system is a set of integrated processes which allocate and control the resources of the computer system, acting as an interface between the user and machine. The memories, processors and processes constitute the resources of a computer system.

2.1.1 Memories

There are three basic types of memory in a computer system. Each processor has a few cells of local memory called registers which, while executing a program, serve various functions such as addressing specific cells in memory, holding the instruction being interpreted, containing operands and intermediate results of computations, etc. The program, or portion of program, being interpreted resides in the processor's "fast access" executable memory. This memory is not large enough to hold more than a small fraction of the total number of

programs or data files in the system. The rest are located in "low speed, large capacity" storage devices, or secondary memories, and loaded into the executable memory on demand.

2.1.2 Processors

The processors are the active elements of a computer and, under the direction of a program, modify the contents of various cells in its memories. If more than one program, or parts of more than one program, can be resident in the executable memory and the processor switched among them, the processor is said to be multiprogrammed. A multipro-cessor system, as the name implies, is a system with more than one processor. Processors in a multiprocessor system interact, typically, only through their shared memories.

2.1.3 Processes

In order for a processor to be multiprogrammed, there needs to be at least two controlling processes for that processor, one to determine those programs to be resident in executable memory and to allocate that memory to them and another process to switch the processor among them. These processes, constituting a rudimentary operating system, may or may not be resident in the executable memory of the processor they are controlling. The ability to interrupt a processor and switch it to other processes necessitates a more general definition of the term process. A process is any discrete set of executable instructions and associated data resident in some memory of the computer, having been initiated or partially interpreted by a processor. If a process has received some service by a processor and then interrupted, the

contents of the registers and other necessary memory is generally preserved so that the process can resume execution when it is again assigned the processor.

2.1.4 Operating Systems

The operating system for any computer system, even the most moderate of them, consists of more than a memory and processor supervisor. Not only does the operating system contain processes which control and allocate the resources of a system, most of these performing functions beyond the control of the user, but also various service routines which are directly or indirectly requested by the users. To the processes requesting one of these resources, the request might appear as an instruction. To the external observer, this request initiates a new process and this new process, in its execution, might make requests initiating other dependent processes. This concept of processes creating dependent processes projects a natural hierarchical structure on the family of computer system processes.

2.2 System Process Graphs

2.2.1 Process States

Once a processor is assigned to a process, execution of the process continues until one of three events occurs: (1) the processor executes a "terminate" instruction; (2) an interrupt occurs from an external process; or (3) the process blocks itself pending the occurrence of some external event.

A process is always initiated by another process. In the case of a "user job," the initiating process could be the operator entering the job into the system. Once a process has been initiated, it exists in one of the three states generally defined for a process which are:

- (1) active - occupying executable memory and executing instructions on the processor;
- (2) ready - desiring execution but not resident in executable memory or, if in executable memory, not being assigned to the processor;
- (3) blocked - not desiring further processing pending the occurrence of some external event.

While a process is active, it progresses toward completion, altering its memories (sometimes being restricted to a subset of the memories accessible by the processor) and constructing and issuing requests for system services, each of these requests initiating a dependent process.

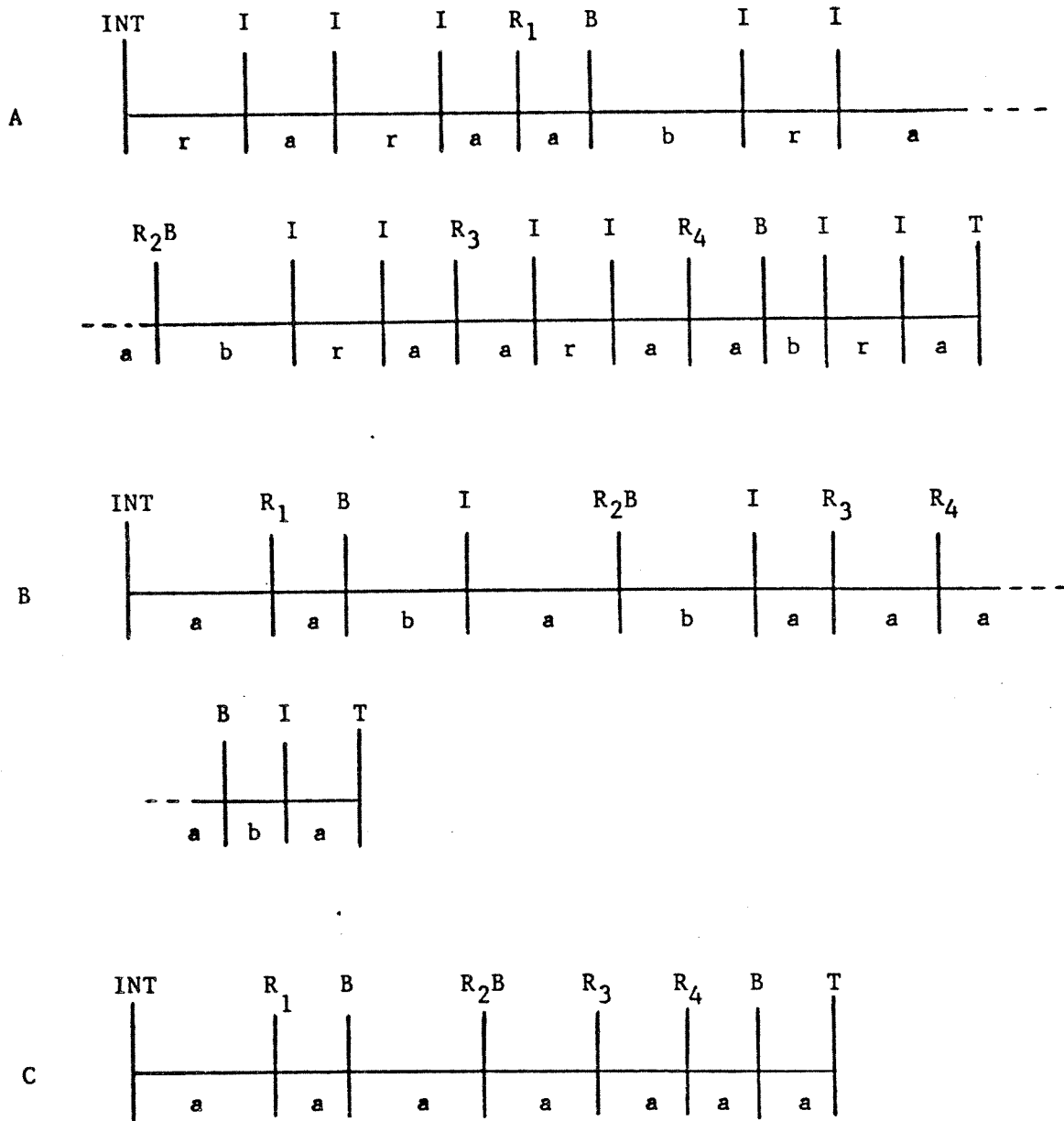
A process blocks itself, typically after initiating a dependent process, waiting on the completion of that process. As the dependent

process must compete for system resources and may, itself, initiate dependent processes, the duration of any blocked state is environment dependent. When a process enters the blocked state, one of two events usually occurs. The process can hold the processor until it is unblocked or, as is generally the case in a multiprogrammed processor, a supervisory process can interrupt the blocked process, save the necessary data for resumption, and allocate the processor to another process.

This "exchange" can also occur when an active process has the processor if the processor supervisor is so designed, that is, the supervisor assigns the processor for a time interval to each process according to some sharing scheme. If the process "exchanged" off were in an active state, its state is changed to ready. If it were blocked, its state is unchanged.

The memory manager often has the capability to swap processes out of executable memory if a "cheaper" process is initiated or enters the ready state. Usually, all executable memory assigned to the process being swapped out, along with necessary data to allow resumption of processing, is copied to secondary memory and the freed memory enters the pool of allocatable executable memory.

If a real time sequential record of the states and requests were kept for the life of a process, that is, from initiation to termination, it might appear as in Figure 2.1A. If this same process were to have a dedicated processor, it would appear as B in the same figure. As a process is unaware of its blocked periods, that state is removed resulting in C. This is now a representation of a process with all external environmental considerations removed.



INT - initiate process
 T - terminate process
 I - interrupt
 B - block
 R_i - request type i
 a - active state
 b - blocked state
 r - ready state

Figure 2.1

An alternate representation for Figure 2.1C is given in Figure 2.2. Other executions of the same process could result in dissimilar request patterns. Superimposing several records of execution, insuring only that the initiation nodes be coincident, might result in the tree structure of Figure 2.3. Allowing loops to reduce the number of nodes, the tree could be altered to appear as in Figure 2.4.

Figure 2.4 is a directed graph reflecting the service and resource demands for the observed executions of that process.

A system process graph (SPG), is a directed graph of a process, the nodes of which are requests for service or resources and the arcs represent active process time between requests. The arcs will also be tagged with their relative frequencies of traversal although this information will not always be utilized in the simulation as will be seen in section 4.2.2.6.

2.2.2 Construction of System Process Graphs

There are two methods for constructing directed graphs of system processes. The first would be by analysis of the source code of the program corresponding to each process [11]. Timing information and relative frequencies of traversal for the arcs would be extremely hard to deduce, however. Also, the examination and analysis of the source code to produce the structures, even if restricted to operating system processes, would be in itself a formidable task. The second method would be by analysis of trace tapes produced by a software probe in the computer system. This method has been used [12] to study the sequential behavior of individual processes. This is the method employed in this

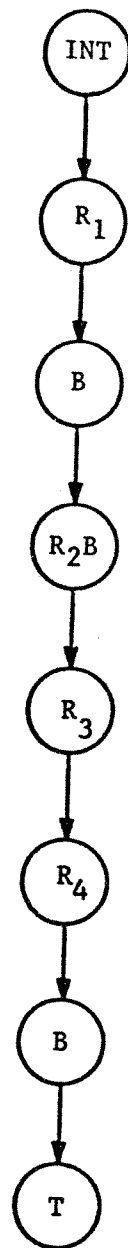


Figure 2.2

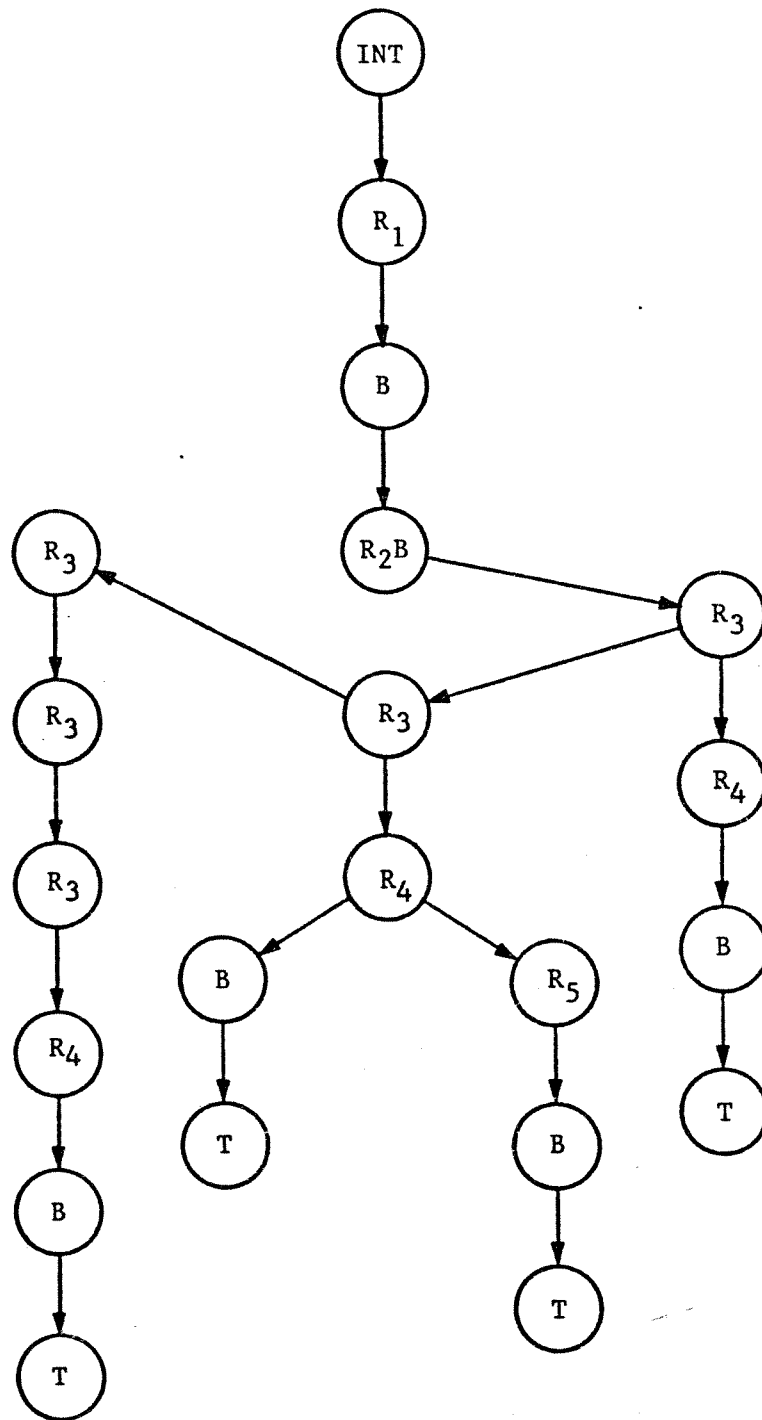


Figure 2.3

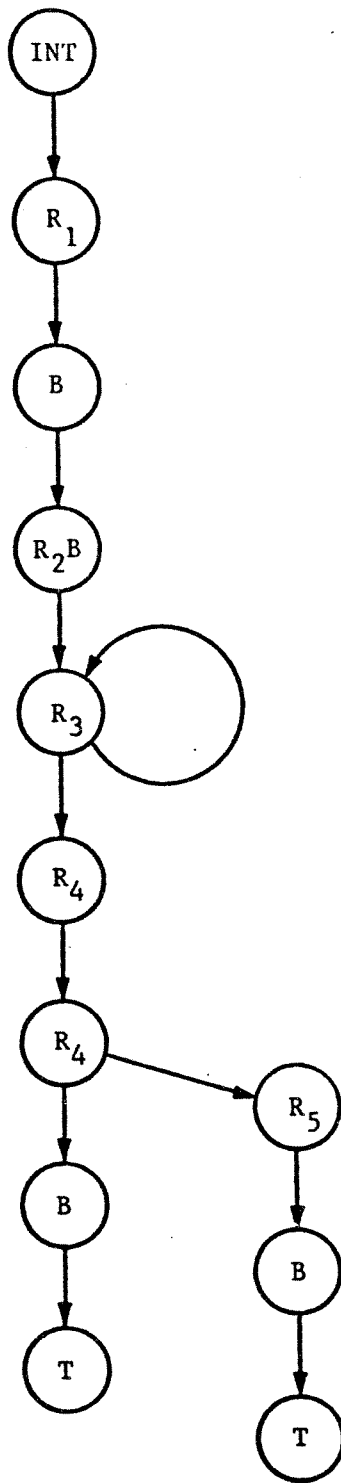


Figure 2.4

study and will be discussed in greater detail in Chapter 4.

2.3 Applications in the Simulation of Computer Systems

There have been several hierarchial approaches to designing a computer system [4,9]. Howard [9] states, "It would appear that any system can profitably be described as a hierarchy even if it was not originally designed with this idea in mind, for a hierarchial approach logically separates internal considerations from external ones and promotes modularity in system descriptions." Because of the applicability of the concept of hierarchy in the description of computer systems, SPG's are especially useful in the simulation of them. SPG's readily preserve the concepts of hierarchy and modularity, providing a means of investigating particular processes or levels of processes in the system. Further, SPG's preserve the time sequencings of states in processes and can preserve both the order and the structure of the system being simulated.

In a computer system hierarchy of processes, those corresponding to user jobs would be the highest order processes. The lowest order processes would be those which initiate no dependent processes and will be referred to as functions. Between the two lies an arbitrarily complex hierarchial family of processes.

SPG's provide an alternative method to incorporating processes into a simulation. Instead of the more conventional way of encoding a corresponding algorithm for each process to be modeled, they could be included in the form of SPG's and input as data. A model incorporating SPG's would require routines capable of simulating the system's servicing of the requests as well as one for traversing the graphs.

2.3.1 Processes Represented by Graphs in the Model

In theory, any process could be represented by a process graph. In practice, the representation of a process by a directed graph would depend upon the capability or possibility of parameterizing that graph. It is not necessary, however, to represent all processes by SPG's. Instead, the more conventional approach of representing processes by corresponding algorithms in the model could be used. Further, it is not necessary to represent processes at all levels by their individual SPG's. Several levels in the hierarchy could be combined by replacing the nodes in the highest level SPG's by the dependent processes they create, similarly replacing the nodes in the successive dependent processes until the desired combination is effected. If this combination were made during the analysis of the trace, as opposed to combining the process graphs after their structure had been realized, the nodes in the higher level SPG's would actually be replaced by that portion of the dependent process actually encountered. Combining levels would allow model simplification, retaining and isolating, by preserving the integrity of their SPG's, only those processes whose behavior or resource utilization patterns is to be studied.

2.3.2 Methodology of a SPG Structured Trace-Driven Model

Essentially, the basis of operation for most simulation models is the same. User resource demands are presented to the model in some fashion and the model attempts to handle these demands as they are handled in an actual, or some conceptualized, system.

In an SPG structured, trace-driven model, the resource demands at the highest level are separate user event traces in the form of time sequenced requests for system services, differing from other SPG's in that they cannot be initiated by other SPG's in the system. It is at this level that most of the combination of SPG's might likely occur. Typically, the Fortran compiler might not be a process whose behavior is to be isolated and studied. No SPG for it would be included separately in the model. Each user job would incorporate its particular execution of that process in its SPG.

As each user job is initiated, the model invokes the corresponding SPG and begins to traverse the graph in a chronological fashion, using the processors and memories in accordance with the supervisor processes of the modeled system. When a request is encountered, the SPG for the dependent process is invoked and attached to the parent. The model then starts the traversal of the dependent process. It is possible for several SPG's to be invoked and actively being traversed for the same parent process, that is, the parent process does not block pending the completion of dependent processes. This would be used, for example, in the case of overlapping I/O processing with computation. After the request has been serviced, the model then selects the next arc to be followed. The selection could be strictly probabilistic according to the relative frequencies of traversal for the arcs being considered. It would be more realistic if, where possible, the selection were resource or environment dependent. This will be discussed in greater detail in Chapter 4.

2.3.3 Properties of the SPG's

The SPG's provide a means of conceptualizing system processes in that they explicitly present the resource utilization patterns for their corresponding processes.

They also provide a means for constructing a model on a more deterministic base. Specifically, the arcs of the lower level SPG's may possess near invariant execution time requirements. As the lowest level SPG's create no dependent processes, their behavior should be well characterized or characterizable.

SPG structured models provide the capability to study system modifications down to the level of the lowest order SPG in the model.

Software modifications to system processes requiring changes to the structure of the associated SPG would necessitate careful analysis of the process being modified. Care must be taken to insure that the estimated execution times for any new arcs would not significantly affect the credibility of the model.

CHAPTER 3

THE MODELED SYSTEM AND ITS SOFTWARE PROBE

3.0 Introduction

To demonstrate the utility and applicability of SPG's in simulation models, one incorporating them was designed and implemented. The model, specifically, is a trace-driven simulation of a CDC 6600 running under the UT-2 operating system. In order to discuss a model, it is first necessary to present the system modeled. This chapter describes the system briefly, but in enough detail to allow understanding of the model and the manner in which it realizes the important features of the system. A more complete description of the CDC 6600 can be found in Thornton [21] or the manufacturer's reference manuals [2,3]. A quite detailed description and analysis of the UT-2 operating system is given by Johnson [12]. Although dealing with the UT-2D operating system, a successor to UT-2, the information given by Howard [10] and Wedel [22] provides further insight into the UT-2 system.

Analysis of the trace data produced by a software probe is used in this study to parameterize the model and construct the SPG's as well as provide statistics for model validation. This chapter also includes a description of the software probe and the events it records.

3.1 System Description

3.1.1 Hardware

The computer modeled is a CDC 6600 installed at the computation center at The University of Texas at Austin. Hardware resources consist of one central processor (CPU), ten peripheral processors (PPU's), 131,072 words of central memory, 500,000 words of extended core storage (ECS) and twelve I/O channels which are connected to four disks, one tape controller, various remote job entry terminals, interactive terminals and unit record equipment. A complete description of the I/O subsystem can be found in Dissly [5].

All processors have access to central memory. Only the CPU can access ECS. Each of the peripheral processors has its own memory and is capable of independent program execution. Each PPU has access to all I/O channels.

3.1.2 Operating System

The UT-2 operating system, designed at The University of Texas at Austin, is a multiprogrammed system providing both batch and interactive processing capabilities. It also provides an extensive program library. Batch jobs can enter the system from local or remote terminals as well as from any of the interactive terminals.

3.1.2.1 Dedicated PPU's

The UT-2 system requires two PPU's which are dedicated to the execution of system processes. One is dedicated to the system monitor, MTR, which exercises overall control of the system. The other is to

DSD, the process which provides communication between the operator's console and the system. When the interactive and the remote terminal systems are active, there are two more PPU's effectively dedicated to the system; one to the execution of LEI, the high speed remote terminal driver, and the other to LED, the medium speed remote entry driver and interactive controller.

3.1.2.2 Partitioning of Central Memory

A portion of central memory is reserved for the central memory resident (CMR) which consists of a resident monitor process (CPM), system tables, interprocess communication cells, and static storage for some peripheral processor programs. The remainder of central memory is partitioned to provide sixteen virtual processors or control points (CP's). A CP can consist of varying amounts of central memory and is considered to exist even when it is not occupied by a process and its size is zero. Three of the CP's are assigned permanently to the system. The interactive and low speed terminal manager (TAURUS) occupies one of them, the real time accounting process (PISCES) occupies another, and the third contains the system buffer pool (GEMINI). GEMINI never requires the CPU and, in that sense, is not actually a process. However, various dependent processes are attached to its control point and they consume system resources. For conformity, GEMINI is a special process, differing from other central processes in that it is never active. This leaves 13 CP's for allocation to user jobs.

3.1.2.3 Processes at CP's

When a central process is active, it can use the CPU to modify

the memory within its own control point area. In order to perform any I/O or gain access to other system resources or services, it must construct an appropriate request to issue to MTR. The address of the first word of a control point area is referred to as its relocation address (RA). The second word (RA+1) is reserved for communication with MTR and, consequently, all requests and messages issued by a central process to MTR are called RA+1 calls. Most RA+1 calls result in the creation of dependent processes. For example, in the UT-2 system, a user requiring the performance of I/O would issue a RA+1 call for a specific peripheral process, CIO. MTR would take this request and put it into the PPU queue, eventually assigning the CIO process to a PPU and attaching it to the calling CP. As mentioned in an earlier section, a central process can block itself. Again considering the example, suppose the process issuing the I/O request needed to wait until the I/O operation is concluded prior to continuing the active state. The "recall" flag would be set in the RA+1 call causing the request to appear as "CIOP." A central process can also block itself for a period of time by issuing a RA+1 call for RCL, specifying a time in the call. It can also block until a word is changed by issuing a RCLP call and specifying the address in the call.

3.1.2.4 Peripheral Processes

Once a peripheral process has been assigned to a PPU, that process has sole control of the PPU and its memory. Requests for other resources or services and relinquishment of granted resources must be communicated to MTR through a special cell in central memory. These

CIO - standard I/O	1AJ - advance job
2WD - write disk	2EF - process error flag
2RD - read disk	2TS - translate control statement
2PD - position disk	3AJ - special statement processor
2MT - read/write/position magnetic tape	1CD - unit record driver
2CT - read/write TAURUS CT file	1CJ - complete job
2DF - drop local file	1DB - dump trace driver
OPE - open file	1ED - TAURUS multiplexor driver
CLO - close file	1PL - plotter driver
RSF - release system files	2PL - plotter overlay
MSG - issue message to dayfile	1PS - peripheral I/O service
CPU - CP utility service	2AM - issue accounting message
DMP - dump memory	2FE - transfer file entry
EPR - TAURUS service routine	2JE - transfer job entry
LDR - loader	2TJ - translate job card
LDE - loader error processor	1RJ - resume job
LOD - loader loader	1SJ - suspend job
PCC - process control card	1SR - process system request
RCC - read control card	1SS - system I/O scheduler
PFM - permanent file manager	1TD - tape driver
RFL - request field length	1TM - write TAURUS dayfile message
SNP - snapshot of central memory	2WM - write dayfile message
1EI - high speed remote driver	
2E1 - remote driver overlay	
2E2 - remote driver overlay	

Table 3.1

Peripheral Processes

requests or messages are referred to as functions. When a peripheral process issues a function call, it blocks itself until a reply is received from MTR.

Table 3.1 lists some of the more commonly used peripheral processes. Only those processes whose names begin with a letter are explicitly invoked by RA+1 calls. The others are invoked by active peripheral processes or, indirectly, by CPM. Those processes whose names begin with a number greater than or equal to 2 are considered to be secondary overlays and in the table will follow those processes most frequently invoking them.

3.1.2.5 Functions

MTR responds to all function requests from the peripheral processes. Many of the functions are processed directly by subroutines in MTR. For others, MTR invokes CPM which contains the necessary subprocesses to handle them. A third class of functions causes MTR to check for certain system conditions and, when those conditions are met, it again invokes CPM to execute one of its subprocesses.

Tables 3.2, 3.3 and 3.4 list the system functions along with a brief description of each. The particular functions appearing in each table are determined by the manner in which they are serviced, i.e., those functions handled by MTR alone, those handled by CPM and those requiring some service from both.

3.1.2.6 MTR

All requests for system resources are handled by MTR. All processes communicate with MTR through selected cells in central memory.

Channel Reservation Functions

- RSY - reserve system channel
- RCH - reserve channel
- DCH - dereserve channel
- CCH - conditional reserve channel

Pool Peripheral Processor (PP) Control Functions

- DPP - drop (terminate use of) PP
- ABT - drop PP and abort control point
- EDR - enter delayed request for PP
- RPP - request additional PP

Control Point Status Control Functions

- RCL - unblock control point (recall)
- RCP - request central processor
- DCP - drop central processor
- CEF - change error flag

Miscellaneous Functions

- ACB - alter control point byte
- RWD - reserve word in central memory
- REV - record event
- STM - control monitor step mode

Table 3.2

MTR Functions

File Reservation Functions

- RLF - reserve local file
- RGF - reserve global file

Track Linking Functions

- RHT - reserve, extend or follow a track chain
- DHT - dereserve, truncate or excise a track chain
- CLS - compute logical sector and update physical position
- DTL - define first legal track

ECS Data Transfer Functions

- REC - read from ECS
- WEC - write to ECS

Equipment Reservation and Control Functions

- RMS - select mass storage device
- CES - change equipment status
- REQ - reserve equipment for job
- DEQ - dereserve equipment from job

Processing Service Functions

- PDE - process device error
- LPP - locate PP program for loading
- FJN - format job name

Table 3.3

CPM Functions

Job Scheduler Control Functions

- SCH - request scheduler run
- SCP - swap control point out

Central Memory Allocation

- RST - request storage size change
- PFR - pause for relocation

Table 3.4

MTR-CPM Functions

MTR polls these cells in a cyclic fashion, acting upon and responding to any messages encountered. By requiring all processes to request resources in this manner, MTR is able to exercise complete control of system resources and to synchronize interaction and competition of all processes.

MTR has two explicit schedulers. One is the PPU scheduler which is basically a FCFS scheduler. Certain peripheral process requests can move to the head of the queue, however, such as the processes requested by the job scheduler to roll jobs in and out of central memory. Further, if a job has been selected by the job scheduler to be preempted from its CP, all process requests associated with that job are removed from the queue and saved until the job is again assigned to a CP. The other scheduler is for the CPU and is effectively a round robin scheduler with an eight millesecond quantum. Once a CP is given the CPU, it holds it for its quantum unless it blocks or is interrupted by an exchange from MTR for CPM. After CPM has executed its task, another exchange occurs and the interrupted CP is given the CPU for the remainder of its quantum (or until it blocks or is interrupted again). A further divergence from a strict round robin discipline is that TAURUS gets preferential treatment. The CPU scheduler always selects the TAURUS CP when the TAURUS process is in the ready state. When a CP is exchanged off the CPU, MTR saves the contents of the registers and then assigns the CPU to next ready CP.

MTR interrupts the CPU with an exchange for CPM for one of three reasons: to process a function; to run the job scheduler; or to relocate a CP. When MTR has invoked CPM for any reason, it blocks until

it receives a response from CPM and another exchange is executed.

SCH and SCP are two of the functions requiring both MTR and CPM service. They are in effect messages requesting that the job scheduler be run. The receipt of one of them will cause MTR to set a flag indicating that the job scheduler is to be run. As soon as MTR determines that all processes carrying out the decisions of the previous execution of the job scheduler have terminated, it clears the flag and activates CPM to run the scheduler again.

RST and PFR are the other two functions requiring both MTR and CPM service. RST is a request that a certain amount of central memory be assigned to a CP. If MTR determines that the change represents a decrease in memory for the CP, CPM will be invoked once to run the central relocation process and update various table entries associated with the CP and the process assigned to it. If it represents an increase and enough free memory is available to grant the request, CPM could be activated several times to run the relocation process to satisfy this one RST. The number of times would be determined by the amount of compaction necessary to give the CP the amount of contiguous memory requested. In any event, if a CP is to be moved or have its allocated memory changed, MTR sets its move flag. Only one RST representing an increase in the amount of allocated memory can be receiving service at any given time. A CP cannot be moved or have its field length changed until all its peripheral processes have terminated or indicated that they have paused for the relocation process by issuing a PFR function. Once MTR determines that all peripheral processes have terminated or are pausing, CPM is exchanged on to the CPU and the relocation process

executed. Should not enough free memory exist to satisfy an RST request, the peripheral process making the request would be notified by MTR. This usually results in a request from that peripheral process for the job scheduler to be run which could result in the parent central process being preempted from central memory.

3.1.2.7 CPM

CPM contains subprocesses to handle those functions as indicated in Figure 3.3. The program directories, file tables, equipment tables and some peripheral process program files are located in central memory. Functions which reference or manipulate them are handled by processes in CPM.

The relocation process is located in CPM primarily because the CPU can perform core to core transfers much faster than a PPU.

The job scheduler, as previously mentioned, is a part of CPM. It runs at the request of MTR and selects those jobs to be resident in the user CP's. Scheduler decisions are implemented by the creation of resume job (1RJ) and suspend job (1SJ) peripheral processes. The cost for a batch job is determined basically by the product of its memory requirement and estimated remaining execution time (an estimate of the execution time for each batch job is supplied by the user via a control card). For interactive jobs, the time since last interaction replaces the execution time factor in computing the cost. Jobs are selected by the scheduler on a least cost basis, restricted by their ability to fit in the remaining free central memory. The UT-2 system uses dynamic memory allocation, thus a job's ability to fit in the remaining free memory

is not affected by the contiguity of the free words of central memory. Processes at CP's, except for the three assigned to the system, can be preempted (rolled out) from central memory if a cheaper job is made available to the system or if there is not sufficient memory for it and for the cheaper jobs already in central memory. A process cannot be preempted until all its existing dependent peripheral processes have terminated. A copy of the control point to be preempted, including a list of all entries in the PPU queue and a copy of the CPU registers as they were at the conclusion of the process' last active period, is saved in secondary memory, usually ECS, to allow resumption of processing when the job scheduler next selects this job.

3.1.3 Software Probe

The software event probe is a resident of MTR. When an event occurs which is to be recorded, MTR stores the event description in one of two buffers in central memory. When a buffer is full, MTR issues a request for a peripheral process, LDB, which is to be attached to the GEMINI CP. When a PPU is assigned, LDB copies the buffer to tape. In the interim, the event recorder is using the other buffer. In a highly active system, it is possible for a buffer to be re-written prior to its being copied to tape, resulting in lost buffers. Missing buffers greatly complicate analysis of the trace, perturbing the results considerably. All event tapes used in this study have less than .05% lost data. The events recorded are depicted in Table 3.5.

1. RA+1 calls
2. Function requests and responses
3. Switching of the CPU among central processes,
including CPM.
4. Movement of requests through the PPU queue
5. Assignment of PPU's
6. Job scheduler decisions

Table 3.5

Events Recorded by the Software Monitor

CHAPTER 4

THE SIMULATION MODEL

4.0 Introduction

The simulation model in this study utilizes directed graphs to reflect explicitly the resource utilization patterns of processes. Traversal of these graphs produces time sequenced events which control the simulation.

This chapter presents a detailed description of the simulation model including its construction, operation, and validation.

4.1 Construction of System Process Graphs

All SPG's used in this model were constructed through sequential analysis of the data produced by the software event monitor.

4.1.1 Functions

Due to the degree of resolution of the data recorded by the software probe, the lowest level processes for which SPG's were constructed were functions. These SPG's consist of two nodes, one for "request" and the other for "response," connected by a single arc.

No process graphs were constructed for CPM nor MTR. The programmed portion of the model consists primarily of routines simulating the execution of these two processes.

The time required for CPM and MTR to service and respond to each of the functions is explicitly presented in the trace data. For functions processed primarily by CPM, virtually all the processing time was used by CPM and very little by MTR in invoking CPM. The processing times for CPM functions, except for those accessing ECS, were generally of the magnitude of 1 millisecond or less (see Tables 4.3 and 4.4).

4.1.1.1 Extended Channel Functions

The RCH function is a request for a channel. Included in the request is a specification for the desired channel. Analysis of the trace data revealed, as expected, diverse holding times for channels linked to dissimilar devices. In order to denote the channel being selected more explicitly, the set of functions was extended, in effect, isolating the requests for and utilization of specific designated channels. A RSY is a request for the channel connected to the disk

containing the system files, commonly referred to as the system disk. A RCH can also be used to gain access to this channel. The analysis of holding times for channels revealed that the holding time for the system channel, when accessed by a RCH, varied considerably from that of the holding times for the other three disk channels (see Table 4.1). For this reason the extended set of functions was increased by one. Table 4.2 presents the definitions for the extended channel functions.

Channel	Run 1 Holding Time	Run 2 Holding Time
0	103.549	79.321
*1	163.827	99.982
2	95.521	75.837
3	95.936	67.346

*In both runs, channel #1 was connected to the system disk.

Table 4.1

Disk Channel Holding Times

RSY	Request system disk channel
RCH	Request non-system disk channel
RCS	Request (RCH) system disk channel
RTP	Request tape channel
RES	Request ECS pseudo-channel
RCX	Request for any channel not specified above

Table 4.2

Extended Functions for RCH

4.1.1.2 Parameterization of the Function SPG's

The statistics gathered for functions include the number of times each function was invoked, the average elapsed time between issuance of a request and receipt of a response for each function and the coefficient of variation for these elapsed times.

For some functions this elapsed time does not reflect accurately the active processing time required. This is true for all functions requesting a limited resource. For instance, the actual processing time used by MTR to service a RCH is negligible. The elapsed time actually indicates the time spent waiting for the desired channel to become available. However, all arcs were tagged with their measured elapsed time characteristics. How these arc times are used in the model will be discussed in a later section of this chapter. Tables 4.3 and 4.4 present function statistics gathered from the trace tapes of two system runs. These two runs will be referred to in this study as Run 1 and Run 2.

4.1.2 Peripheral Process Graphs

SPG's were constructed for all peripheral processes except MTR, as previously mentioned, and DSD. DSD drives the operators console and provides communication between the operator and the system. The software event probe sees no significant events generated by DSD directly. Therefore, the only effect of DSD reflected directly in the model is that, since DSD occupies a dedicated PPU, the number of pool PPU's is reduced by one.

The nodes of SPG's for peripheral process graphs are functions. The arcs between the nodes reflect the active process time between

Run 1

Function	Number of References	Mean Service Time*	(Coefficient of Variance) ²
RSY	6664	142.2933	3.8608
RCH	16389	36.4947	13.7842
DCH	44064	0.0000	0.0000
CCH	0	0.0000	0.0000
RMS	541	.9464	.1970
RHT	20234	1.4293	.7557
DHT	5999	1.4462	.2012
RST	10077	75.8690	7.7481
PFR	1546	39.5938	11.8291
DPP	66918	.4050	1.5176
ABT	17	.1765	4.6667
RCL	59471	0.0000	0.0000
RCP	12102	0.0000	0.0000
DCP	3180	.0003	3179.0000
REC	6383	3.7932	.6114
WEC	4230	5.4369	.3385
CES	215	.8279	.3572
REQ	7102	.9095	.2531
DEQ	3162	.9336	.2221
PDE	0	0.0000	0.0000
RPP	428	.0818	11.2286
LPP	113133	.9348	.1885
EDR	11498	0.0000	0.0000
CEF	241	.0041	240.0000
RGF	0	0.0000	0.0000
ACB	2239	.8825	.2719
RWD	32884	0.0000	0.0000
RLF	44011	.9180	.2332
SCH	8275	0.0000	0.0000
SCP	224	0.0000	0.0000
REV	0	0.0000	0.0000
STM	0	0.0000	0.0000
DTL	0	0.0000	0.0000
CLS	3583	.9911	.1829
FJN	1921	.9162	.2217
RTP	13542	10.4573	4.5077
RES	121	0.0000	0.0000
RCX	3505	7.1201	8.3023
RCS	3846	135.8679	3.6991

*Time in milleseconds

Table 4.3

Function Statistics