MULTIPROGRAMMING: AN OVERVIEW

by

CHARLOTTE LAND CRAWFORD

B. A., Texas Christian University, 1963

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1971

Approved by:

Tom L. Gallagher
Major Professor

TABLE OF CONTENTS

INTRODUCTION

Multiprogramming is the result of the computer users' desire to make more efficient use of the equipment and improve throughput and response times. Multiprogramming exists in many forms, but the basic idea is that two or more programs share the resources of the computer.

The purpose of this paper is to present an overview of multiprogramming: what it is, how it works.

Five important properties of a multiprogramming computing system are:

> (1) Computation processes are in concurrent operation for more than one user. . . .
> (2) Many computations share pools of resources in a flexible way. . . .
> (3) Individual computations vary widely in their demands for computing resources in the course of time. . . .
> (4) Reference to common information by separate computations is a frequent occurrence. . . .
> (5) An M[ultiprogrammed] C[omputing] S[ystem] must evolve to meet changing requirements. . . .[1]

The advantage of using a multiprogramming system can be demonstrated by considering two programs, a foreground program and a background program. The foreground program has high usage of card readers, printers, tapes, disks, drums (input-output), and the background program primarily uses the processor capabilities (compute-bound). The foreground program receives priority use of the central processor but relinquishes it frequently because of input-output (I/O) requests. When the foreground program is interrupted, the background program processes until either the foreground program I/O request is satisfied or the background program issues an I/O request.

This method works even when one program is not processor bound if enough I/O devices exist in the system to service a number of programs. The

---

[1]Jack B. Dennis and Earl C. Van Horn, "Programming Semantics for Multiprogrammed Computations," Communications of the ACM, IX (March, 1966), 144.

central processor is shared, and the I/O devices tend to reach their maximum capability.

The elapsed time required to execute a single program is greater than when that program runs alone. However, an overall reduction in turn-around time for the total job stream results because long jobs incrementally process to completion.

This improvement is accomplished at a price. More hardware is necessary to detect idle programs and more software to switch control among the programs. Critchlow states that this increase must be less than the increase in output of useful work if a net gain in efficiency is to be achieved.[2]

In 1959 Codd proposed a set of six conditions which a multiprogramming system must satisfy if it is to be generally accepted and used. They are still applicable today.

1. Independence of preparation. Programs may be independently written and compiled.

2. Minimum information from programmer. A programmer need not provide additional information about his program for successful processing.

3. Maximum control by programmer. While certain features of the machine must be placed out of the programmer's direct control, no reduction in the effective logical power available to him must result.

4. Noninterference. Programs must not be allowed to interfere with one another.

5. Automatic supervision. Some method must be used to control I/O, detect machine malfunctions and programming errors, perform scheduling, and

---

[2]A. J. Critchlow, "Generalized Multiprocessing and Multiprogramming Systems," AFIPS Conference Proceedings: 1963 Fall Joint Computer Conference, XXIV (Baltimore: Spartan Books, 1963), 107.

keep accounting records. (This method is known today as an operating system.)

6. Flexible allocation of time and space. The needs of the program should dictate allocation of core and I/O devices.[3]

OPERATING SYSTEM

An operating system consists of special control and service programs to perform most of the routine supervisory functions of the machine operator. It eliminates the need for operator intervention during processing and permits sequential execution of independent programs. It handles linking between program segments and any necessary program relocation. It supervises queues, if the system uses them, and determines job selection. One important function is to supervise I/O: file label checking, buffering, blocking, error detection, recovery procedures. It must be able to handle interrupts from external devices, e.g., inquiry stations, remote terminals, display devices, or other computers.

Most operating systems contain:

1. Programs to compile and assemble source code
2. Programs for sorting data
3. Programs to manage I/O for all types of devices and data set organization
4. Programs to determine space availability on direct-access devices
5. Programs to diagnose and analyze I/O device failures
6. Programs to determine the disk location of all applications programs as well as those named above

---

[3]E. F. Codd and others, "Multiprogramming STRETCH: Feasibility Considerations," Communications of the ACM, II (November, 1959), 14.

7. Programs to write messages to the operator

8. Programs to maintain a catalog of data sets

9. The supervisor, which controls and schedules all other programs.[4]

The supervisor informs programs when to begin execution and regains control of the central processor when an interrupt occurs. The supervisor consists of many programs, called routines. These routines include:

1. Scheduler, which oversees the entire administration of the operating system.

2. I/O routines, which control all I/O operations.

3. Task supervisor, which determines the status and condition of each task in core.

4. Task dispatcher, which chooses the next task to use the central processor.

5. Main storage supervisor, which maintains a table of available core storage.

6. Time supervisor, which maintains time of day and calculates the amount of time used by each task.

7. Contents supervisor, which maintains a list of programs in core at any given time.

8. Interrupt handling routines, which determine the specific cause of an interrupt. Interrupts are of the following types: (a) external--the time allotted to the executing program is gone or an external signal is received; (b) program--the executing program contains an error; (c) I/O--an I/O operation has taken place; (d) supervisor call--an instruction in the active program requests that central processor control be returned to the supervisor; (e)

---

[4]Harry W. Cadow, OS/360 Job Control Language (Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1970), pp. 120-121.

machine check--a central processor malfunction occurs.

The operating system usually resides permanently in a protected (read only) section of main memory. In some systems, certain routines may reside on an auxiliary storage device and be brought into core as needed; this usually represents a substantial hardware and software overhead. Resident control programs can occupy significant amounts of the available core memory. Many users have not been willing to tolerate this loss of resources for an operating system. It is, however, possible for the user to design the operating system to fit his individual needs. The advent of direct access storage facilitates efficient segmentation techniques, which may significantly reduce the sizes of resident control routines.

One of the major requirements of multiprogramming is that the job stream be able to be read continuously. Information on job control cards is placed in a job queue and each set of data cards recorded on an auxiliary storage device as a separate data set.[5] One method of accomplishing this is called spooling, where peripheral I/O operations for card readers and printers are programmed to proceed concurrently with the normal job stream. The system stays well ahead of the card reader and maintains a backlog on disk for the printer so no active program is delayed.[6]

The operating system can control use of the central processor's time in three ways. One is a time-slicing system, which automatically allots so many units of time (a quantum) to each active job. In a queueing system it is possible that the quantum may vary from queue to queue. Time-slicing is an attempt to more equally allot usage of the central processor.

---

[5]Ibid., p. 152.

[6]Robert F. Rosin, "Supervisory and Monitor Systems," Computing Surveys, I (March, 1969), 49.

A subset of time-slicing is dynamic dispatching. In each queue job priority is determined by the amount of I/O required--the more I/O the higher the priority.

Instead of time-slicing, a system may use the roadblocking technique for central processor usage. In roadblocking, once a job has gained control of the central processor, it does not have to relinquish the central processor until it requires I/O. This technique may exist in systems with or without priorities.

QUEUEING

Multiprogramming is not effective without priority among programs and interrupt facilities so that higher priority tasks can assume control. Priorities are implemented through the use of queues.

Two types of queueing disciplines exist: head of line and preemptive. In a head of line queue, a resource services the request with the highest priority after completing the current request. In a preemptive queue, a request is interrupted whenever a higher priority request enters the queue.

Queueing models based on these disciplines are determined by considering the way jobs arrive and the amount of service time required. These models are the round robin, the multiple-level feedback, and the processor-sharing model. The latter is a theoretical model.[7]

In the round robin model (RR), jobs are serviced on a first-in first-out (FIFO) basis, with each job allocated a quantum of time. If its service demand is greater than the quantum, the job is put on the rear of the queue and will loop through until its service demand is met.

---

[7]E. G. Coffman, "Studying Multiprogramming Systems," Datamation, XIII (June, 1967), 49-54.

The multiple-level feedback model is the most general. A hierarchy of queues exists with a time-slice which may vary according to the level of the queue. If the service request of a job is not fulfilled at the end of the quantum, the job is put at the end of the next higher level queue (priority decreases with higher queue level). Assuming a finite number of queue levels, at the highest level, if a job still needs service, no feedback occurs. It loops at the highest level as long as necessary but receives service only after all lower-level queues are empty.

In this model, at the end of any time-slice, the next job to be serviced comes from the highest priority, non-empty queue.

Internal queue structure in this model is first in line, which can be determined either by FIFO or by the original entry queue level.

Another model which has received analysis is the theoretical processor-shared (PS) or pure time-sharing (PTS) model. If the time-slice approaches zero, then the round robin and feedback models become PS models. These models allow two or more jobs to share the processor simultaneously. Four different PS models exist: simple PS (round robin), PS with priorities (job priority determines rate at which service is received), preemptive PS (infinite level feedback model without priorities), and preemptive PS with priorities (infinite level priority model with jobs ordered by priority within the queues).

Kleinrock has introduced a generalized queueing model which can be applied to various scheduling algorithms.[8] Assuming no preassigned priorities in this model (i.e., all jobs join the queue with zero priority), a job awaiting service gains priority at rate, α. While receiving service, priority

---

[8]Leonard Kleinrock, "A Continuum of Time-Sharing Scheduling Algorithms," AFIPS Conference Proceedings: Spring Joint Computer Conference, XXXVI (Montvale, New Jersey: AFIPS Press, 1970), 453.

changes at a different rate, β. If a job is removed from service before completion, the priority again changes at rate α, etc. All jobs possess the same parameters α and β, and at all times jobs of highest priority are serviced equally.

Depending on the relative values of α and β, six scheduling algorithms are discussed by Kleinrock.[9]

In the first come first served (FCFS) scheduling algorithm, jobs in the queue gain priority at a rate which is below the rate of jobs in service. That is, $0<\alpha\leq\beta$. Only when one job finishes can another receive service.

If jobs in service gain no priority at all (i.e., $\beta=0$), then each job entering the queue will immediately receive service. This is the processor-sharing round robin (RR) algorithm.

In the selfish round robin (SRR) algorithm jobs are serviced in a processor-sharing RR manner. The rate gain in priority of those jobs in the queue is such that they will eventually catch up to those already in service (i.e., $0\leq\beta\leq\alpha$). However, those already receiving service are "selfishly" trying to keep the service for themselves. In this case, each job spends some time in the queue and then joins the group being serviced in an RR manner.

In the last come first served (LCFS) algorithm, priority decreases with time, both for the queued jobs and for the active ones. However, queueing jobs lose priority faster than ones receiving service (i.e., $\alpha\leq\beta<0$). Obviously any new job will have highest priority and so gains control until his service demand is satisfied or a new job demands service. If a job processes to completion, the queued job with highest priority becomes active.

A modification of this model occurs when $\alpha<0<\beta$, that is, queueing

---

[9]Ibid., pp. 454-455.

jobs lose priority but active jobs gain priority. If a new job finds the active job with a negative priority, it gains control and processes to completion since the service priority increases while queueing priority decreases, meaning no queueing job can catch up. Service then goes to the queueing job with highest priority. Since queueing jobs lose priority, all priorities are negative, including the one next receiving service. If this job can keep processor control until its priority is positive, it is said to "seize" control and can process to completion. This algorithm is called LCFS with seizure.

The bulk service algorithm is a special case where all jobs in the queue retain zero priority, i.e., $\alpha=0<\beta$. All queueing jobs are taken into service simultaneously and begin to gain priority. Any new job must queue until all the active jobs are finished, and then the entire queue again receives service "in bulk".

In the LCFS with pickup case an active job loses priority while a queueing job either loses at a slower rate or gains priority. The range is $\beta<0$, $\beta<\alpha$. Consequently, queueing jobs are "picked up" by the active jobs when the priorities are equal, and the entire group then receives service.

Figure 1 illustrates the range of scheduling algorithms discussed above. The FCFS, LCFS, and RR models are well known and solved. The two regions describing the LCFS with seizure and LCFS with pickup are not solved. Kleinrock gives an analysis of the SRR. A job whose demand for service is greater than average is discriminated against in the SRR model as compared to a FCFS model, while a job whose service demand is less than average is treated preferentially in the SRR model as compared to a FCFS model.[10]

---

[10]Ibid., p. 454, 458.

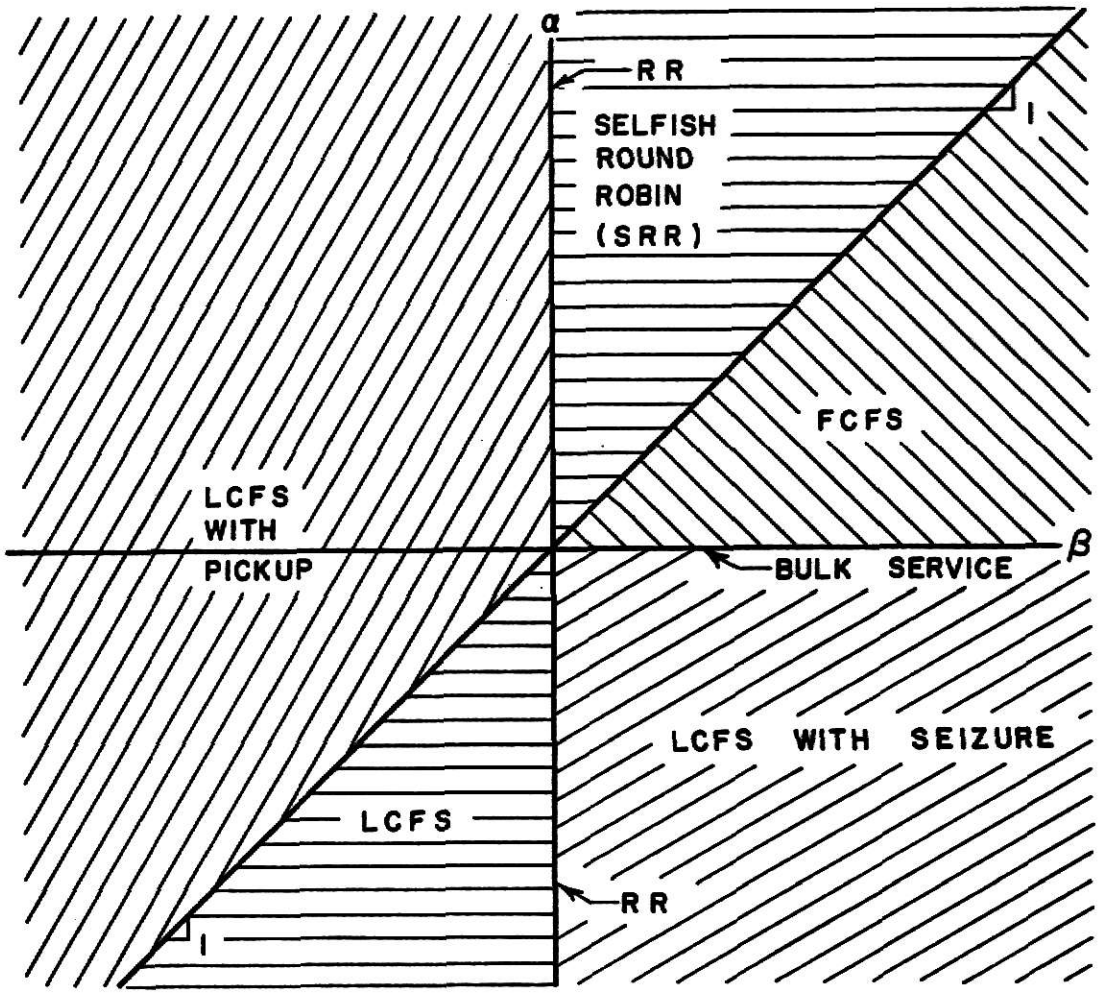

Fig. 1. Structure of Kleinrock's generalized queueing model.

CORE ALLOCATION

Core can be allocated (i.e., divided) either statically or dynamically. In static allocation, main memory is divided into two prime areas, one for the operating system and the other for partitions, which are fixed areas of core. The maximum number of partitions is determined at system generation time, but the actual number of partitions used may vary. Each partition may service more than one job queue. Partitions may be time-sliced among themselves, and if a partition services remote terminals, it also may be time-sliced internally. Each partition can contain only one task at any given time.

In dynamic allocation of core, a variable number of tasks may execute concurrently. A fixed area of main storage contains the resident part of the system. The remainder of core is available for tasks. One method used is to initiate a task with highest priority within a queue provided there is enough available core. This assumes only one job from each queue can be resident and competing for resources at any one time. A disadvantage to this method is that core may become fractured. That is, suppose several large jobs and many small jobs are running concurrently with some medium sized jobs queueing. The small jobs run to completion, leaving enough core for a medium sized task, except that the space now available is in small segments scattered within core. During any one period of systems operation, this situation may become worse.

Two solutions have been suggested and implemented. One is called virtual memory, in which any program can be subdivided into small sections, usually of different sizes, and scattered about in core. A disadvantage to this system is its cost; the links (both hardware and software) necessary to connect these sections are expensive to buy and use. The other solution is dynamic memory relocation, in which tasks are reassigned in memory to make the available space contiguous. The jobs are removed from memory onto an

auxiliary storage device and then loaded back into core. This method has the disadvantage of degrading the system while the exchange takes place.

Another method of dynamic core allocation is called paging. A page is the basic unit of storage and transmission. Core is physically divided into sections, and a page may contain part of a program (also called a page) or data. A program is not necessarily loaded into contiguous pages, so a page table is kept which shows where each page of program or data is located.

Most systems use a page on demand strategy. That is, one page is mapped into main memory originally, and this page demands any other pages it may need for execution. These pages in turn may demand other pages. Eventually a page is called when core is "full", so one of these pages must be "turned out" (copied back onto auxiliary storage) to make room for the new page. The problem becomes one of which page to replace.

Several page replacement algorithms exist to determine which page is to be replaced. These are random selection, first-in first-out, and least recently used. The simplest of these algorithms is a random selection of pages, but it results in high page traffic between core and auxiliary storage.

The FIFO algorithm replaces the page which has been least recently paged. The logic for this algorithm is that programs execute instructions in sequence, so the first page called will be least likely to be needed. One argument against this is that many programs are set up as calls to subroutines, other programs, etc., so that the presence of the first page of the program is vitally necessary for execution. If demand for core is quite intense, another disadvantage of this algorithm is apparent. The list of pages in memory is processed rapidly, and many of the deleted pages are still needed by their programs. This causes an excess of page faults as each program is trying to get the pages needed for execution. It is possible that the page requesting

a new page will be overwritten.

The Least Recently Used (LRU) algorithm replaces the page which has gone the longest time without being referenced. This method works well, except when overload occurs, at which time its behavior degenerates to that of the FIFO algorithm.

Wegner has suggested several factors which determine the efficiency of paging:

1. Page tables and other necessary software add much to a system's overhead.

2. Processes seem to require a large number of pages over short time sequences resulting in many page faults.

3. The time required by a process to acquire enough pages in memory to run for a reasonable length of time without page faults tends to be quite long. This factor also shows that paging requires a considerable real time investment.[11]

Separate studies conducted by Fine, Jackson, and McIsaac[12] and O'Neill[13] seem to show that demand paging leads to very inefficient computer utilization.

Fine, et al., conclude that a program tends to demand pages at a very rapid rate until a sufficiency of pages is acquired, but then the program

---

[11]Peter Wegner, "Machine Organization for Multiprogramming," Proceedings of the 22nd National Conference of the ACM (Washington: Thompson Book Co., 1967), p. 148.

[12]Gerald H. Fine, Calvin W. Jackson, and Paul V. McIsaac, "Dynamic Program Behavior under Paging," Proceedings of the 21st National Conference of the ACM (Washington: Thompson Book Co., 1966), pp. 224-226.

[13]R. W. O'Neill, "Experience Using a Time-Shared Multiprogramming System with Dynamic Address Relocation Hardware," AFIPS Conference Proceedings: Spring Joint Computer Conference, XXX (Washington: Thompson Book Co., 1967), pp. 611-621.

often does not run very long. If a program does run for an extended time, its sufficiency of pages is a large fraction of total pages required.[14]

The idea of the working set model, introduced by Denning, is an attempt to reduce excessive paging. The working set is defined to be the set of most recently used pages of a process or the minimum set of pages which must be present in core for a process to operate without unnecessary page faults.[15]

The working set has two important properties which show how it differs from the other paging algorithms:

1. A process is active if and only if its working set is completely in main memory.

2. It is applied individually to each program in a multiprogramming system.[16]

Denning defines a working set memory allocation policy as one in which a process is active only if enough unused space exists in core to contain its working set.[17]

De Meis and Weizer describe implementation of the working set theory on an RCA Spectra 70/46 with Time-Sharing Operating System. A count of pages used in execution of a task during its previous active period is used as the working set of that task. The task is not activated until the number of main memory pages not in use at any instant at least equals the size of the working set. The conclusion reached is that "the primary value of the working

---

[14]Fine, et al., loc. cit.

[15]Peter J. Denning, "The Working Set Model for Program Behavior," Communications of the ACM, XI (May, 1968), p. 326.

[16]Peter J. Denning, "Thrashing: Its Causes and Prevention," AFIPS Conference Proceedings: Fall Joint Computer Conference, XXXIII, Part I (Washington: Thompson Book Co., 1968), p. 917.

[17]Ibid., p. 916.

set concept lies in aiding the operating system to intelligently schedule the use of main memory based on a reasonable prediction of the memory requirements of the tasks in its load."[18]

Thrashing is defined as "excessive overhead and severe performance degradation or collapse caused by too much paging."[19] It will turn a shortage of memory space into a dedication of the processor to obtain appropriate pages. Denning believes that the poor performance of paged systems is due not to program behavior but the amount of time necessary to access a page in auxiliary storage.[20]

The missing page probability is the probability that when a page is referenced by a process it is not in main memory. This probability is a good way to measure a paging algorithm's performance since the probability is lower if necessary pages are normally resident in core.

Denning shows that the FIFO and LRU algorithms, since they are applied globally across core, permit the missing page probability to fluctuate as the total demand for memory fluctuates.[21] This fluctuation is a direct result of the length of time necessary to move a page between auxiliary storage and main memory (traverse time).

Denning claims "that applying a paging algorithm globally to a collection of programs may lead to undesirable interaction among them."[22]

It is possible for programs to interact if the paging algorithm is close to saturation. Large programs will not get as much space as they need,

---

[18]W. M. De Meis and N. Weizer, "Measurement and Analysis of a Demand Paging Time-Sharing System," Proceedings of the 24th National Conference of the ACM (Washington: Thompson Book Co., 1969), p. 205.

[19]Denning, op. cit., p. 915. [20]Ibid.

[21]Ibid., p. 921. [22]Ibid., p. 917.

and the space each program acquires will depend on its demands as compared to the others in core.

These globally applied algorithms exhibit great susceptibility to thrashing. Dennings solutions are: (1) use of the working set model algorithm to make a program independent of the others' demands for space; (2) slow-speed core memory between drum and main memory to cut down on traverse time by making speeds between memory levels more compatible; and (3) sufficient main memory to contain the desired number of working sets.[23]

IBM has designed a new hardware device, called a cache, which is a small, high-speed buffer contained in the central processor unit. This type memory unit is installed on the IBM 360/85 and 370 series. It seems that this is a uniquely hardware-implemented paging system.

The cache is used to hold the contents of those parts of main storage currently being used; however, it is not addressable by a program. When a program references data not contained in the cache, the portion of main memory containing the referenced data must be loaded into the cache, overwriting some other portion.

Both main storage and the cache are divided into sectors, and during operation a correspondence is set up between the cache sectors and main storage sectors. Most main memory sectors are not assigned a cache sector because there are only a limited number of cache sectors. The cache sectors are assigned dynamically during operation to the main storage sectors currently being used by the program. In order to decide which cache sector to reassign, an activity list is kept and the sector which has been least recently referenced is reassigned.

---

[23]Ibid., p. 922.

Each sector is divided into sixteen blocks, and the blocks are loaded on demand. Therefore, if a program only references one or two blocks of a main memory sector, these are the only ones loaded into the cache sector. When main memory is updated the cache is also updated, so no cache sector reassignment is necessary.

PROTECTION

Protection of the system from the user and the user from the system has evolved as operating systems have become more sophisticated. Additional hardware implementation has been necessary. Registers to hold acceptable upper and lower bound addresses and registers to reflect protection status for fixed portions of core have been developed. Facilities also exist to make a certain area read only, write only, or execute only. Interrupt facilities prevent attempts to execute hardware-dependent I/O operations, to set the special registers (interval timer or protection registers) or to execute illegal instructions. Privileged instructions exist which only the supervisor can execute. It is possible for the supervisor to complete an operation the user requested but did not have the privilege to execute. Thus the user must rely on the system, insuring system integrity.[24]

CONCLUSION

Multiprogramming is a logical development of computer usage. It is an attempt to use more efficiently the computer equipment and improve throughput and response times by having two or more programs execute concurrently.

The implementation of multiprogramming has run into many obstacles in

---

[24]Rosin, op. cit., p. 52.

the course of its development. Many different methods exist to implement the idea of multiprogramming. Some form of dynamic core allocation seems to be preferable, but whether paging can be improved to an acceptable point remains to be seen.

Unfortunately it is possible for the amount of time used by the supervisor in maintaining the job flow to be as great as that required to process applications programs. Perhaps investigation should be made to determine whether some activities supported by the system cannot be accomplished by other methods.

It may be possible to design an operating system which results in small overhead for jobs not using or requiring many system resources but which accommodates jobs requiring extended functions.

An upper limit exists on what a multiprogramming system can accomplish. As the system approaches this upper limit, marginal improvements require vast amounts of time and effort. It is not possible to do more than use all of the machine all of the time!

BIBLIOGRAPHY

1. Books

Cadow, Harry W. OS/360 Job Control Language. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1970.

Wilkes, M. V. Time-Sharing Computer Systems. New York: American Elsevier Publishing Co., 1968.

2. Papers

Bouvard, J. "Perspective on Operating Systems," Comparative Operating Systems: A Symposium. New York: Brandon/Systems Press, Inc., 1969.

Brawn, Barbara S., and Frances G. Gustavson. "Program Behavior in a Paging Environment," AFIPS Conference Proceedings: 1968 Fall Joint Computer Conference. Vol. XXXIII, Part II, pp. 1019-1032. Washington: Thompson Book Co., 1968.

Cantrell, H. N., and A. L. Ellison. "Multiprogramming System Performance Measurement and Analysis," AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference. Vol. XXXII, pp. 213-221. Washington: Thompson Book Co., 1968.

Coffman, E. G., and R. R. Muntz. "Models of Pure Time-Sharing Disciplines for Resource Allocation," Proceedings of 24th National Conference of the ACM. Pp. 217-228. New York: ACM Publication P-69, 1969.

Critchlow, A. J. "Generalized Multiprocessing and Multiprogramming Systems," AFIPS Conference Proceedings: 1963 Fall Joint Computer Conference. Vol. XXIV, pp. 107-126. Baltimore: Spartan Books, 1963.

De Meis, W. M., and N. Weizer. "Measurement and Analysis of a Demand Paging Time Sharing System," Proceedings of 24th National Conference of the ACM. Pp. 201-216. New York: ACM Publication P-69, 1969.

Denning, Peter J. "Effects of Scheduling on File Memory Operations," AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference. Vol. XXX, pp. 9-21. Washington: Thompson Book Co., 1967.

_____. "Thrashing: Its Causes and Prevention," AFIPS Conference Proceedings: 1968 Fall Joint Computer Conference. Vol. XXXIII, Part I, pp. 915-922. Washington: Thompson Book Co., 1968.

Fenichel, Robert R. "An Analytic Model of Multiprogrammed Computing," AFIPS Conference Proceedings: 1969 Spring Joint Computer Conference. Vol. XXXIV, pp. 717-721. Montvale, New Jersey: AFIPS Press, 1969.

Fine, Gerald H., Calvin W. Jackson, and Paul V. McIsaac. "Dynamic Program

Behavior Under Paging," Proceedings of 21st National Conference of ACM. Pp. 223-228. Washington: Thompson Book Co., 1966.

Kleinrock, Leonard. "A Continuum of Time-Sharing Scheduling Algorithms," AFIPS Conference Proceedings: 1970 Spring Joint Computer Conference. Vol. XXXVI, pp. 453-458. Montvale, New Jersey: AFIPS Press, 1970.

Kuehner, C. J., and B. Randell. "Demand Paging in Perspective," AFIPS Conference Proceedings: 1968 Fall Joint Computer Conference. Vol. XXXIII, Part II, pp. 1011-1018. Washington: Thompson Book Co., 1968.

McCredie, John W., Jr. "Measurement Criteria for Virtual Memory Paging Rules," Proceedings of 24th National Conference of ACM. Pp. 193-199. New York: ACM Publication P-69, 1969.

O'Neill, R. W. "Experience Using a Time-Shared Multi-Programming System with Dynamic Address Relocation Hardware," AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference. Vol. XXX, pp. 611-621. Washington: Thompson Book Co., 1967.

Peters, Bernard. "Security Considerations in a Multi-Programmed Computer System," AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference. Vol. XXX, pp. 283-286. Washington: Thompson Book Co., 1967.

Steel, Thomas B., Jr. "Multiprogramming--Promise, Performance and Prospect," AFIPS Conference Proceedings: 1968 Fall Joint Computer Conference. Vol. XXXIII, Part I, pp. 99-103. Washington: Thompson Book Co., 1968.

Wegner, Peter. "Machine Organization for Multiprogramming," Proceedings of 22nd National Conference of ACM. Pp. 135-150. Washington: Thompson Book Co., 1967.

3. Periodicals

Arden, B. W., and others. "Program and Addressing Structure in a Time-Sharing Environment," Journal of the Association for Computing Machinery, XIII (January, 1966), 1-16.

Chang, Wei, and Donald J. Wong. "Analysis of Real Time Multiprogramming," Journal of the Association for Computing Machinery, XII (October, 1965), 581-588.

Codd, E. F. "Multiprogram Scheduling: Parts 1 and 2. Introduction and Theory," Communications of the ACM, III (June, 1960), 347-350.

_____. "Multiprogram Scheduling: Parts 3 and 4. Scheduling Algorithm and External Constraints," Communications of the ACM, III (July, 1960), 413-418.

_____, and others. "Multiprogramming STRETCH: Feasibility Considerations," Communications of the ACM, II (November, 1959), 13-17.

Coffman, E. G. "Studying Multiprogramming Systems," Datamation, XIII (June, 1967), 47-54.

_____, R. R. Muntz, and H. Trotter. "Waiting Time Distributions for Processor-Sharing Systems," Journal of the Association for Computing Machinery XVII (January, 1970), 123-130.

_____, and L. C. Varian. "Further Experimental Data on the Behavior of Programs in a Paging Environment," Communications of the ACM, XI (July, 1968), 471-474.

Denning, Peter J. "The Working Set Model for Program Behavior," Communications of the ACM, XI (May, 1968), 323-333.

Dennis, Jack B. "Segmentation and the Design of Multiprogrammed Computer Systems," Journal of the Association for Computing Machinery, XII (October, 1965), 589-602.

_____, and Earl C. Van Horn. "Programming Semantics for Multiprogrammed Computations," Communications of the ACM, IX (March, 1966), 143-155.

Flores, Ivan. "Multiplex Programming," Science & Technology, September, 1969, pp. 6-13.

Liptay, J. S. "Structural Aspects of the System/360 Model 85 II The Cache," IBM Systems Journal, VII, 1 (1968), 15-21.

Maher, R. J. "Problems of Storage Allocation in a Multiprocessor Multiprogrammed System," Communications of the ACM, IV (October, 1961), 421-422.

Rosin, Robert F. "Supervisory and Monitor Systems," Computing Surveys, I (March, 1969), 37-53.

Saltzer, Jerome H., and John W. Gintell. "The Instrumentation of Multics," Communications of the ACM, XIII (August, 1970), 495-500.

Smith, John L. "Multiprogramming under a Page on Demand Strategy," Communications of the ACM, X (October, 1967), 636-646.

Stevens, David F. "On Overcoming High-Priority Paralysis in Multiprogramming Systems: A Case History," Communications of the ACM, XI (August, 1968), 539-541.

4. Reference Manuals

Control Data Corporation 3600 Computer System, Drum SCOPE Reference Manual, No. 60059200, Rev. A, 1965.

General Electric 625/635 System Manual, No. CPB-371B, 1965.

IBM System/360 Operating System: Concepts and Facilities, Form C28-6535-4, 1968.

IBM System/360 Operating System: Control Program with MFT, Program Logic Manual, Program Number 360S-CI-505, Form GY27-7128-5, June 1970.

IBM System/360 Operating System: MFT Guide, Form C27-6939-5, 1965.

IBM System/360 Operating System: MVT Guide, Form GC28-6720-1, 1970.

UNIVAC 1108 Multiprocessor System, System Description, 1968.

ACKNOWLEDGEMENTS

The author wishes to thank Dr. Tom L. Gallagher for his suggestion of the topic and constructive criticisms of this report. The author especially acknowledges the guidance, suggestions, and interest of Mrs. Elizabeth A. Unger.

Financial assistance from the National Institute of Health under the Department of Health, Education, and Welfare is gratefully acknowledged.

MULTIPROGRAMMING: AN OVERVIEW

by

CHARLOTTE LAND CRAWFORD

B. A., Texas Christian University, 1963

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1971

ABSTRACT

The purpose of this paper is to present an overview of multiprogramming with an emphasis on those components and methodologies found in contemporary multiprogramming systems. The properties, advantages, and disadvantages are presented, and the conditions multiprogramming must satisfy to be acceptable are discussed. The concepts and methodologies presented are the operating system, queueing, core allocation, and protection.

Some possible improvements in multiprogramming technology are noted. At this time multiprogramming seems to be the most reasonable way for general purpose computing systems to produce work efficiently.