1974

Deadline scheduling in a multiprogrammed computer environment

John Raleigh Jordan
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/rtd
Part of the Computer Sciences Commons

Recommended Citation

This Dissertation is brought to you for free and open access by Digital Repository @ Iowa State University. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact hinefuku@iastate.edu.
INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in “sectioning” the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

Xerox University Microfilms
300 North Zeeb Road
Ann Arbor, Michigan 48106
JORDAN, John Raleigh, 1941-
DEADLINE SCHEDULING IN A MULTIPROGRAMMED
COMPUTER ENVIRONMENT.

Iowa State University, Ph.D., 1974
Computer Science

University Microfilms, A XEROX Company, Ann Arbor, Michigan

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED.
Deadline scheduling in a multiprogrammed computer environment

by

John Raleigh Jordan

A Dissertation Submitted to the Graduate Faculty in Partial Fulfillment of The Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1974
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>CHAPTER II. A REVIEW OF THE LITERATURE</td>
<td>4</td>
</tr>
<tr>
<td>CHAPTER III. THE MEETING OF DEADLINES</td>
<td>16</td>
</tr>
<tr>
<td>Probability calculations</td>
<td>16</td>
</tr>
<tr>
<td>Side entry queue</td>
<td>17</td>
</tr>
<tr>
<td>Ordering by deadline</td>
<td>22</td>
</tr>
<tr>
<td>CHAPTER IV. COMPLETION TIMES</td>
<td>26</td>
</tr>
<tr>
<td>Bounds on the completion times</td>
<td>27</td>
</tr>
<tr>
<td>Estimates of the completion times</td>
<td>43</td>
</tr>
<tr>
<td>CHAPTER V. CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH</td>
<td>52</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>55</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>58</td>
</tr>
</tbody>
</table>
CHAPTER I. INTRODUCTION

The purpose of this research is to determine the extent to which it is possible to guarantee a user of a multiprogrammed, batch-operated computer system that his job will be completed by a given deadline. The conditions under which it is possible to determine a job's completion time are studied. In particular, an upper bound on the completion time of a job is obtained. This bound is a random variable and from its distribution function the probability that the job will complete on or before its given deadline is computed. If the probability of meeting this deadline is high, the user may leave the shop with a high degree of confidence that when he returns, his job will be finished. In cases where the probability of meeting the deadline is low, the user is provided with an adjusted deadline which will be met with high probability.

The environment under study is a third generation machine, multiprogrammed with a fixed number of tasks. The cost of using the procedures developed here would be reasonable in this environment because the required computations would not add significantly to the overhead of any present-day operating system.

Historically, rapid turnaround has been a goal of multiuser computer systems. In many cases, however, rapid turnaround is not enough to enable a user to effectively plan
his activities. It is our contention that it is often more useful to give the user an estimate of when his job will be done. Better yet, if possible, let him request a completion time (deadline) and have the system grant or deny that request. If the request is granted (deadline promised) the system should be capable of delivering with a high degree of accuracy. This thesis discusses just such an approach.

Chapter II reviews the work which has been done in deadline scheduling and gives the evidence for the need of research in this area.

A reasonably fast method of computing the necessary probabilities is discussed in the first section of Chapter III.

In the second section of Chapter III a new queueing discipline or scheduling algorithm is developed which allows the system to grant deadlines more often than it can using first-come-first-serve. It is shown that the results developed under the dispatching mechanism which is used in Chapter IV are valid under this discipline.

The last section of Chapter III provides the proof that the queue may be partially ordered by deadlines which will increase the chances that the requested deadline of a newly arriving job can be met.

The first section of Chapter IV develops theoretical bounds on completion times in any multiprogrammed system with
a fixed number of tasks.

The results of simulations which lead to some surprisingly accurate heuristic estimates of (as opposed to bounds on) completion times are presented in the second section of Chapter IV.

Chapter V presents the conclusions derived from this research and contains this author's suggestions for further work in the area of deadline scheduling.
CHAPTER II. A REVIEW OF THE LITERATURE

Most of the computer literature to date which deals with scheduling is centered around the efficient use of machines (3,4,5). The cost of hardware is starting to drop radically. We must begin to consider the environment in which a computer is used.

In a very recent paper Balke (1) stated "The only reason for buying or using a computer system is to improve the performance of people against the measures applied to them. Successful computer performance measurement can occur only when the computer system is related structurally and statistically to the goals and activities of the people who buy and use it."

We take this to mean that the efficient use of the machine is not necessarily the most important factor in a computer system. Rather, the most important factor is the machine's usefulness to people.

As was outlined in the introduction, we feel that to meet deadlines will greatly enhance the effectiveness of a computer system. However we cannot expect to be able to guarantee meeting all that are requested. Conway, et al. (7) seem to agree on both of these points when they discuss

---

1Numbers in parentheses refer to bibliographic entries.
sequencing against due-dates (deadlines): "Of the various measures of performance that have been considered in research on sequencing, certainly the measure that arouses the most interest in those that face practical problems of sequencing is the satisfaction of pre-assigned job due-dates. ... Scheduling procedures of sufficient power to enforce a completely arbitrary set of due-dates do not yet exist. If they did, then it would not be necessary to worry about the manner in which the due-dates are determined. ... In some situations due-dates are exogeneous to the scheduling organization; they are set by some independent external agency and announced upon arrival of the job. ... A more typical situation is: the producing agency proposes a date, which is adjusted according to the customer's needs and competitive forces, and then the producer and consumer agree on a final due-date."

The above remarks are a clear indication for the need for work in this area and furthermore emphasize the notion that due-dates cannot be arbitrary, but are negotiated.

There are literally hundreds of papers and many books which have been written in the areas of queueing and scheduling. Queueing theory in general seeks to characterize a system by describing the waiting or flow times of jobs in a shop in terms of a distribution function. In general, the queueing results depend on the jobs' processing times being
exponentially distributed. Haaloe (19) states "'Mean waiting
time in a multiple-channel system' is a traditional problem
in the literature on queueing theory. ... Only under special
circumstances, when service time is exponentially
distributed, does the problem have an exact solution. There
is no general solution, so far, for the most trivial case:
uneven service distribution and particularly not for constant
service time."

Scheduling theory deals with finding optimal schedules
and the results are mostly for fixed, known processing times
with common job arrivals. Although none of these conditions
is present in a computing environment, some of the results
from both disciplines serve as useful guidelines in the
problem which we address in this paper.

All of the writers in pure scheduling theory who have
addressed deadline scheduling have seen the problem as one of
optimization. From 1959 through 1967, the loss functions
which are minimized measure the magnitude of the lateness or
tardiness of jobs.

McNaughten (20) in 1959 was the first author to discuss
deadline scheduling of the "multiple computer". He said
"There are at least two different requirements for scheduling
a number of tasks. One is the requirement that all tasks be
finished as soon as possible. This kind of requirement has
received attention in the literature. Another kind of
requirement which has received little, if any, attention is that the tasks be completed according to their relative urgency..." He pointed out that within the latter category there are two kinds of deadline: "The first is what might be called an absolute deadline, one in which a task has no value at all if it is not completed by the deadline. ... More often a deadline is an expectation date set more or less arbitrarily: the task is by no means without value if it passes its deadline, although in the usual case it may lose some of its value. ... The crucial point of time is called a relative deadline." The relative deadline is the one which all of the authors cited here address and is the one studied in this thesis.

McNaughten defined the loss function: "... for any task there exists a deadline d and a penalty coefficient p such that for every t, if the task finishes at time t, there is a loss function of max(0,p(t-d)); in other words, there is no loss if the task is completed on or before time d." This loss function is called tardiness when p=1. He considers scheduling several one stage tasks with fixed, known times on several processors to minimize total cost. In the case of a single computer, he shows the minimal cost is achieved by ordering the jobs in decreasing order of p(i)/a(i) where p(i) is the penalty for being late and a(i) is the fixed processing time. He does not say how one would go about
ordering the jobs in this way, however. He also addresses
the case of $m$ processors and $n$ jobs with $a(1)$, $a(2)$, ..., $a(n)$ the processing times for the $n$ jobs, allowing serial
splitting of tasks. He proves that a necessary and
sufficient condition for a schedule to exist in which all the
tasks are completed by time $s$, $s \geq 1/m \sum_{i=1}^{n} a(i)$ is that

for all $i$, $a(i) \leq s$.\footnote{The word "sum" stands for the more conventional Greek
letter Sigma.} We show in Chapter IV that when
equality holds for $s$ in the above, $s$ is a bound on the
beginning time for a new job when the tasks are not split.
He also shows that if all deadlines are zero, there exists a
schedule without splits whose cost does not exceed one in
which there are finitely many splits.

Referring to McNaughton's work, Eastman et al. (9) point
out that no efficient algorithm for finding the optimal
schedule has been found even though that optimal schedule is
known to exist. In developing an upper bound for the cost of
an optimal schedule for $n$ jobs with fixed processing times on
$m$ identical processors ($m < n$), they imply (without proof) that
when jobs are arbitrarily numbered $1, 2, \ldots, n$ and are assigned
for processing in this order that in general, a bound on the
beginning time for job $m+k$ is
\[ \frac{m+k-1}{m} \sum_{i=1}^{m+k-1} a(i), \text{ where the } a(i) \text{ are the processing times.} \]

Our approach to meeting deadlines in this paper is to find expressions for the completion times so that we may find their distribution functions and hence the probability of meeting the deadlines. These expressions are elusive in the case of a multiprogrammed computer.

Newell (22) says that in a fluid approximation, when there is more than one processor (fluid conductor) the system behaves as if there were just one server serving at the rate of the sum of the rates of the individual processors. This result does not directly apply because computer jobs are not in general capable of being divided up into small pieces so that they behave as a fluid.

Jackson (14) showed that the maximum job tardiness is minimized by sequencing the jobs in order of non-decreasing due-dates. This result applies to jobs with common arrivals and fixed processing times.

Fife (11) considers the scheduling of the processing of an initial queue of jobs and subsequent Poisson arrivals on a single processor. He defines a random variable, total cost \( C \), and finds a rule for assigning priorities which minimizes \( \mathbb{E}[C] \), the expected value of \( C \). That rule is to arbitrarily order the jobs and compute \( \mathbb{E}[C] \). Then successively interchange jobs which are adjacent and leave them
interchanged if that results in a lower $E[C]$. This then results in an optimal schedule and he shows the same result holds for the case of no more arrivals or with Poisson arrivals. We use an interchange algorithm in Chapter III when trying to find a schedule in which a newly entering job will likely complete by its deadline.

Schild and Fredman (28) provided a technique involving $m^2$ calculations for optimally scheduling $m$ tasks with arbitrary delay cost functions on a single processor.

Rothkoph (25) considers the problem of scheduling $m$ independent immediately available tasks on $n$ parallel processors. He develops a dynamic programming algorithm for a wide class of parallel processor problems when the service times are fixed integers while allowing the processors to have different processing rates.

In a later paper Rothkoph (26) addresses random processing times. He gives a formula for the cost of a schedule when the tasks have been assigned to the processors, which is linear in the processing times. He finds the expected value of the cost in terms of the expected values of the times. He then seeks that assignment to the processors which minimizes the expected cost by applying the rules of deterministic scheduling.

In 1968 Moore (21) chose to minimize a different kind of loss function, the number of late jobs. He gives a
marvelously elegant algorithm for fixed, known processing times on a single processor which, in a maximum of \( n(n+1)/2 \) simple steps (\( n \) is the number of jobs) provides the optimal schedule. It partitions the \( n \) jobs into two sets, those whose deadlines will be met (\( A \)) and those whose deadlines will not be met (\( B \)). Jobs in \( A \) are ordered by deadline and jobs in \( B \) may be ordered in any way without affecting the measure because the set \( A \) contains the maximal number of jobs that will meet their deadlines.

Moore's work was extended to random processing times on a single processor by Balut (2) in 1973. He assumes set-up and processing times are normally distributed and considers the sum of them to be a single normally distributed random variable. He states that the problem is to maximize the number of jobs in the resulting schedule subject to the \( n \) constraints, \( P[C(i) \leq d(i)] \geq 1-a \) for \( i=1,2,\ldots,n \), where \( C(i) \) and \( d(i) \) are the completion time and deadline, respectively, for job \( i \). These constraints are used in the work described in this thesis. In final form the problem is an integer programming problem with \( n \) quadratic constraints and he concludes "... the problem remains intractable in this simplified form. It is possible, however, to deduce a great deal about the nature of the solution through inspection of [the] constraints." He interprets the constraints to mean that effectively one should order the jobs according to
due-dates, compute the expected completion times in ascending order of job index number, and when the first job encountered in the sequence for which the expected completion time exceeds the due-date, select from among the jobs included in the sequence up to that point one for exclusion from the sequence. The choice of job to exclude is made after a non-linear equation for each of the jobs in the sequence is solved.

Fabrychy and Shamblin (10) sequence jobs as a function of the probability of completion by deadline. They present a dynamic algorithm intended for a large manufacturing concern with multiple machines but treat processing times for all the machines as a single random variable. They compute a new schedule at the beginning of each day's work. The sequencing rule is to order the jobs at each machine center from smallest to largest probability of completion by deadline. They conclude that this has the effect of giving approximately equal probability of completion by deadline to all jobs. This result is in harmony with that obtained in the static scheduling problem with fixed processing time when the jobs are ordered by slack time, where the result is to maximize the minimum tardiness (7).

Lauesen (17) describes an existing operating system which gives users estimates of job completion times. An estimate for the finishing time is computed when a job is
enrolled, and it is updated whenever changes in the schedule occur. The latest estimate is available from any terminal. He states it is rather precise in practice, with a tendency to be somewhat pessimistic. It is important to note that it is the latest estimate which is rather precise. His algorithm is a modified version of the Banker's algorithm as described by Haberman (12). It ignores multiprogramming and the inputs to it are upper limits on the processing times.

In his section entitled "Improvement of the Turn-around Predictions" Lauesen makes a distinction between the cases of batch jobs and time-sharing jobs. He repeats multiprogramming is ignored but states it only seriously affects the batch jobs. These are the very jobs to which deadlines are most important. A time-sharing user has decided to commit his own time to the computing job until it is finished. The batch user is the one who has other things to do and needs to know when his job will be ready for him.

Lauesen makes a courageous statement on page 7: "The algorithm as it stands pays greater attention to justice in turn-around time than to efficient resource utilization." We applaud him and his system for this.

In his concluding remarks, Lauesen states "We have had no complaints about turn-around time—not even from heavily loaded installations where the common processors (CPU and disc) are busy 22 hours a day servicing jobs (overhead
disregarded). I believe that the main reason for this is the estimated finishing times. Users are at ease and they can better stand a long turn-around time.

With the exception of the last two papers cited, all of the writers have cast the problem of deadline scheduling in an optimization framework. The philosophy of the approach taken by Moore and by Balut comes closer to the approach taken in this thesis than any of the others in that it is concerned with the number of jobs which are late. There are several reasons why their solutions are not applicable in a multiprogrammed computing environment. The first and most obvious is that they address only a single processor. The more important reason is that the notion of optimally scheduling a fixed number of jobs makes little sense in a dynamic system. Of course it is certainly possible to employ an optimal procedure with every new arrival or at the completion of every job (or both) but it is conceivable that a given job may never finish under this scheme! That is, when a user deposits a job he would be told that under the optimal procedure, if no more jobs arrive, his job will complete by its deadline with a certain probability. But if any more jobs arrive, all bets are off. We do not think this philosophy will optimize the usefulness of a computer as far as the user is concerned. We believe the approach taken in this thesis will lead to much more fruitful results sooner
than the optimization route.

At the risk of being accused of trying to optimize, we conclude this chapter by citing Koenigsberg (16). He states that delays are shorter if all servers are fed from a single queue instead of having several queues. We employ a single queue which is served by the multiprogrammed computer.
CHAPTER III. THE MEETING OF DEADLINES

Probability calculations

In Chapter IV we develop bounds for job completion times. Let \( C^*(n) \) be a bound on the completion time for a job which is at position \( n \) in the queue. \( C^*(n) \) may be expressed as a linear combination of the processing times of the first \( n \) jobs in the queue. That is,

\[
C^*(n) = \sum_{i=1}^{n} a(i) X(i), \text{ where the } X(i) \text{ are the processing times and the } a(i) \text{ are constants.}
\]

We may let \( Y(i) = a(i)X(i) \) and write

\[
C^*(n) = \sum_{i=1}^{n} Y(i).
\]

We want a fast method of determining \( P[C^*(n) \leq d] \), where \( d \) is the deadline. An approximate distribution for \( C^*(n) \) may be developed from the Central Limit Theorem. It can be shown that \( C^*(n) \) is asymptotically normal as \( n \) increases if the following conditions are satisfied (8):

1. \( Y(i) \) are independently distributed with means \( m(i) \) and variances \( s^2(i) \).
2. The third absolute moment of \( Y(i) \) about its mean, \( r^3(i) \), is finite for every \( i \).
3. If

\[
r^3 = \sum_{i=1}^{n} r^3(i)
\]
then \( \lim_{n \to \infty} r/s^2(n) = 0, \)

where \( s^2(n) = \sum_{i=1}^{n} s^2(i) \).

4. The expected effect of any single \( Y(i) \) on \( C^*(n) \) is relatively insignificant.

It is not unreasonable to expect that the above requirements are fulfilled by the processing times of computer jobs.

Given that \( C^*(n) \) is approximately normally distributed with mean,

\[
\mu = \sum_{i=1}^{n} m(i) \quad \text{and variance } s^2 = \sum_{i=1}^{n} s^2(i)
\]

it is a simple matter to transform it into a standard random variable \( Z \) which is normal with mean zero and variance one, for which tables at any confidence level are available (18). In this way we are able to determine the approximate probability that \( C^*(n) \) is less than or equal to \( d \). We are also able to find the minimum \( d^* \) such that \( P[C^*(n) \leq d^*] = 1-a \), where \( a \) is the level of error we will tolerate.

Side entry queue

In a standard queueing system the operating procedure is specified by a so-called queueing discipline such as first-come-first-serve (FCFS). We do not have a standard queueing system. Instead we have two independent processes, the scheduler and the dispatcher. The scheduler constructs
the queue and manipulates the jobs in it. The dispatcher selects jobs from the queue for processing. Because the first job in the queue may require more memory than is available in a partition which becomes free, the dispatcher searches through the queue until it finds the first job which will fit in the available partition. This is called a first-fit (FF) discipline. This dispatching mechanism is independent of the manner in which the queue is constructed.

We propose in this section a queueing discipline or scheduler which will allow us to grant the requested deadline more often than we would be able to if we used a FCFS queueing discipline.

For purposes of illustration, let us assume for the moment that all job processing times are normally distributed with given means and variances. We may characterize a job by an ordered triple, \((\text{mean}, \text{variance}, \text{deadline})\). Assume there is one job \(J(1)\) which is \((200, 10, 800)\) waiting to be processed by a machine with one partition. Assume the job in process is about to finish but before it completes, another job, \(J(2)\) which is \((10, 1, 20)\) arrives. If we simply join it to the queue as in FCFS, it will almost certainly not complete by its deadline. But if we insert \(J(2)\) in the queue in front of \(J(1)\), both of them will almost certainly finish by their deadlines. If on the other hand \(J(1)\) had been \((10, 1, 12)\), placing \(J(2)\) in front of it would jeopardize \(J(1)\)'s chances
of completing on time.

The general case is as follows. Assume there is one job being processed by a single partition machine and n-2 jobs waiting to be processed. Further assume that at the time each of the n-1 jobs was enrolled, it was guaranteed with probability at least 1-a that it would complete by its deadline. For example, each job had a very generous deadline and it was sufficient to simply join it to the queue. Let a new job J(n) arrive. The following procedure will be followed.

J(n) will be joined to the end of the queue. If it can be guaranteed completion by its deadline with probability at least 1-a, leave it at the end of the queue. If not, then begin jumping over jobs in front of it, one at a time, until the guarantee can be met, if it can. That is, we form trial schedules with J(n) in front of J(j), j=n-1,n-2,...,l, 1 \geq 1. At each jump, two probability calculations must be made. The first is to see if J(j) will complete by its deadline with probability at least 1-a when J(n) is placed in the queue in front of it. If not, then we let l=j and position J(n) immediately behind J(l) and find the minimum d*(n) such that P[C(n) \leq d*(n)] = 1-a. This d*(n) is a negotiated deadline, the best the procedure can do without jeopardizing any other job's chances. If, on the other hand, J(j) will complete by its deadline with
probability at least 1-a, then we compute the probability that J(n) will complete by being placed in front of J(j). If this probability is at least 1-a, we let l=j and stop the procedure leaving J(n) positioned in front of J(l). If not, we jump another job (subtract 1 from j) and test the probabilities with the new j. Eventually we will either position J(n) or exhaust the queue. If we exhaust the queue and J(n) still cannot be satisfied then it will be placed first in the queue with a negotiated deadline, d*(n). We have given the name of Side Entry Queue (SEQ) to this procedure because a job may enter from the "side of the line" instead of always having to go to the end of the line.

The above procedure looks perfectly innocent but there is reason to question the validity of the probability calculations. Specifically, at the time of enrollment of a job, it was promised that it would complete by its deadline with probability at least 1-a. This calculation depends on the number of jobs which will be processed before it and the procedure outlined above can change that number. We must show that the procedure maintains \( P[C(j) \leq d(j)] \geq 1-a \) for every j.
Theorem 3.1 For \( j = 1, 2, \ldots, n \) if, when \( J(j) \) is enrolled, it is true that \( P[C(j) \leq d(j)] \geq 1-a \) then the truth of this statement is not changed by using the SEQ procedure.

**Proof**

During the time a typical job \( J(j) \) is in the system, some number of jobs \( K \) will be positioned in the queue in front of it. \( K \) is a random variable with some distribution,

\[
P[K=k] = g(k), \quad k=0,1,2,\ldots,m,
\]

\( m \) is finite because only a finite number of jobs may enter the system before \( J(j) \) leaves the system. Our procedure ensures that any schedule we construct which affects \( J(j) \) has the property that \( P[C(j) \leq d(j)] \geq 1-a \). That is,

\[
P[C(j) \leq d(j) \mid K=k] \geq 1-a \quad \text{for} \quad k=0,1,\ldots,m. \quad (3.1)
\]

It is well known (18) that

\[
P[C(j) \leq d(j)] = \sum_{k=0}^{m} P[C(j) \leq d(j) \mid K=k] P[K=k]. \quad (3.2)
\]

Replacing \( P[C(j) \leq d(j) \mid K=k] \) in (3.2) by \( 1-a \) from (3.1), we may write

\[
P[C(j) \leq d(j)] \geq \sum_{k=0}^{m} (1-a) P[K=k]
\]

\[
= (1-a) \sum_{k=0}^{m} P(K=k)
\]

\[
= (1-a). \quad Q.E.D.
\]
It may be true that at some point \( P[C(j) \leq d(j)] \) will become less than 1-a but that will happen because of the stochastic nature of the processing times. We will not construct a schedule which has this property.

**Ordering by deadline**

If we simply adhere to the above outlined procedure there will still be times where we will not be able to grant users' requests. The procedure may be modified after the following result is established.

Consider a single partition machine (single server) in which two jobs, \( J(k) \) and \( J(l) \), are positioned adjacently in the queue so that \( k=l+1 \). Let the total processing time represented by the jobs in front of \( J(l) \) be \( T \). Let \( C(1,l) \) and \( C(1,k) \) denote the completion times under this schedule. Clearly,

\[
C(1,l) = T + X(l) \quad \text{and} \quad C(1,k) = T + X(l) + X(k). \tag{3.3}
\]

Assume

\[
P[C(1,l) \leq d(l)] \geq 1-a, \tag{3.5}
\]

\[
P[C(1,k) \leq d(k)] \geq 1-a \quad \text{and} \quad d(k) < d(l). \tag{3.6}
\]

Consider what happens if we interchange \( J(k) \) and \( J(l) \). Let \( C(2,l) \) and \( C(2,k) \) denote the completion times under this schedule.

\[
C(2,l) = T + X(l) + X(k) \quad \text{and} \quad C(2,k) = T + X(l) + X(k). \tag{3.8}
\]
\[ C(2,k) = T + X(k). \] 

Since \( C(2,1) = C(1,k) \), we know from 3.6 that
\[ P[C(2,1) \leq d(k)] \geq 1-a. \] 

Then 3.7 and 3.10 imply
\[ P[C(2,1) \leq d(1)] \geq 1-a. \] 

Equations 3.4 and 3.9 imply
\[ C(2,1) \leq d(1). \] 

Then 3.6 and 3.12 imply
\[ P[C(2,k) \leq d(k)] \geq 1-a. \] 

We have just proved Theorem 3.2.

**Theorem 3.2** If the probability of completion by deadline for two adjacent jobs in a queue which are scheduled to be processed in reverse order of their deadlines by a single server is at least 1-a then that probability is at least 1-a for both jobs if their positions are interchanged.

What this does for us is to allow us to place safe jobs with large deadlines farther back in the queue which will make more room for a newly entering job to fit. By using this additional procedure we may be able to insert a job after \( J(k) \) and before \( J(1) \) when we were not able to insert it after \( J(1) \) and before \( J(k) \). Clearly in the single server case, the completion times for the jobs other than \( J(k) \) and \( J(1) \) are unchanged by switching them around.

It is interesting to note that if we apply this procedure iteratively on every pair of jobs (provided the
conditions are met) we will order the jobs by their deadlines and this is the order which, in the static scheduling problem with fixed processing times, minimizes the maximum tardiness (14). It is also the order for random processing times which results when an optimal schedule is found which minimizes the number of late jobs to such an extent that no jobs will probably be late (2).

Unfortunately this simple procedure is not directly usable in the multiprogramming case. As we will see in the next chapter, in these cases the completion times are bounded in the following manner.

\[ C(1,l) \leq T + X(l) = C*(1,l) \]  \hspace{1cm} \text{(3.15)}

\[ C(1,k) \leq T + X(k) + rX(l) = C*(1,k) \]  \hspace{1cm} \text{(3.16)}

where \( 0 < r \leq 1 \). It is easy to verify that we can get the same result for \( C*(2,k) \) as we did for \( C(2,k) \) in the single server case. But the bounds for the second job in the two schedules, \( C*(2,l) \) and \( C*(1,k) \) are not equal because of the coefficient \( r \). This prevents us from making the substitutions as we did in arriving at line 3.10 in the single server case. What we can do is to let \( r=1 \) in defining the bounds on the completion times, since it will only make them larger, and regain the equality which allowed us to do the sorting. The effect will be that we will not recognize some of the times when we could legitimately interchange the jobs. We must take on faith at this point (but it will be
observed in Chapter IV) that the bound for a given job is a linear combination of random processing times of the jobs which precede it in the queue where the coefficients depend only on the size class (if there is more than one size class) and not on the order in which the jobs are processed. Thus, the bound for a given job is not changed by reordering the jobs which precede it.
CHAPTER IV. COMPLETION TIMES

The service time for a given job is determined by a random variable. If there are n jobs in the system, the processing time or service time for each job J(i) is X(i), i=1,2,...,n. The X(i) are assumed to be independent but not necessarily identically distributed. If we restrict our attention to the case in which there is a fixed number of tasks, the degree of multiprogramming does not contribute to the variance of the X(i) (6). Competition for resources such as CPU and channels will contribute to the variance of X(i). The effects of this competition on the resident processing time is assumed to be random. If it were not for this competition, the processing times would, in general, be known (15). A procedure for actually obtaining the distribution functions is outlined in the suggestions for future research in Chapter V.

When a newly entering job arrives we will want to make a statement about the probability that it will complete by its deadline. We will derive in this chapter functions which are random variables about which probabilistic statements can be made. These functions are bounds on or estimates of the completion times. They are linear combinations of the times for all the jobs in the system at the moment of entry. In order that we may treat all of the random variables the same we will assume the processor has just finished all of its
jobs and is ready to be loaded again. In practice we would be obligated to use conditional distributions for those jobs already in progress. The procedures are equivalent (7).

**Bounds on the completion times**

There are six theorems in this section dealing with bounds on the beginning times for jobs. The completion time for a job is simply its beginning time plus its processing time. The first theorem deals with the simplest multiprogrammed machine, namely two equal sized partitions. The next two theorems give bounds on the beginning time for m-partition machines with equal and unequal size partitions, respectively. The fourth theorem deals with bounds for a machine which has m different size classes of partitions with several partitions in each class. The last two theorems, with additional constraints, give improved bounds on those derived in the previous two theorems.

For i=1,2,...,n, let J(i) denote job i.

For each J(i) let

- \( X(i) \) denote the time required to process job i,
- \( B(i) \) denote the time it will begin processing, and
- \( C(i) \) denote the time it will complete processing.

\( B(i) \) and \( C(i) \) are measured from time zero. Let the jobs be arranged in a queue so they will be available for processing in increasing order of their index i.
The simplest multiprogrammed case is one in which the computer's memory is divided into two equal-sized partitions, thus allowing jobs to run in either partition. The dispatcher selects the first job in the queue as soon as a partition becomes available. Thus, the empty partitions are immediately filled with \( J(1) \) and \( J(2) \). \( J(3) \) will begin processing in the first partition which becomes available. Thus, 

\[
B(3) = \min(C(2), C(1)) = \min(X(2), X(1)).
\]

\( J(4) \) will begin when a partition first frees up after \( J(3) \) begins processing. Thus,

\[
B(4) = \min(C(3), \max(C(2), C(1)))
\]

\[
= \min(X(3) + B(3), \max(C(2), C(1)))
\]

\[
= \min(X(3) + \min(C(2), C(1)), \max(C(2), C(1)))
\]

It is clear that the expression for \( B(i) \) is going to be unmanageable. The reason for this is that each \( B(i) \) is selected from the minimum of two sums. These sums are the accumulated processing times of all the jobs which have begun before \( B(i) \). Some of them ran in one partition, some in the other. There is no way of knowing before hand at each stage which partition will become available first. Thus at stage \( i \)

\[
\text{there are } 2^{i-1} \text{ possible combinations of random variables}
\]

and one of them will be \( B(i) \). Clearly \( B(i) \) is a complicated function and if the expression for it were manageable, we could find its distribution function. As an alternative we will find a bound on \( B(i) \).
**Theorem 4.1** If \( B(1) = B(2) = 0 \), then \( C(1) = X(1) \) and \( C(2) = X(2) \) and for each \( i = 3, 4, \ldots, n \),

\( B(i) \leq B^*(i) \), where

\[
B^*(i) = \frac{1}{2} \sum_{j=1}^{i-1} X(j).
\]

**Proof.** At each stage \( i \), the minimum sum is selected as \( B(i) \). At stage \( i+1 \), the maximum sum from stage \( i \) is a candidate for selection. We will need a dummy variable, \( Y(i) \), at each stage to denote this maximum. By construction

\[ B(3) = \min(C(2), C(1)) \text{ and } \]

\[ C(3) = X(3) + B(3). \]

Let

\[ Y(3) = \max(C(2), C(1)) \]

For \( i = 4, 5, \ldots, n \), let

\[ B(i) = \min(C(i-1), Y(i-1)) \text{ and } \]

\[ Y(i) = \max(C(i-1), Y(i-1)). \]

Then

\[ B(i) = \min(C(i-1), Y(i-1)) \]

\[ \leq \frac{1}{2} (C(i-1) + Y(i-1)) \]

\[ = \frac{1}{2} (X(i-1) + B(i-1) + \max(C(i-2), Y(i-2))) \]

\[ = \frac{1}{2} (X(i-1) + C(i-2) + Y(i-2)) \]

\[ = \frac{1}{2} (X(i-1) + X(i-2) + B(i-2) + \max(C(i-3), Y(i-3))) \]

\[ = \frac{1}{2} (X(i-1) + X(i-2) + \min(C(i-3), Y(i-3)) + \max(C(i-3), Y(i-3))) \]

\[ = \frac{1}{2} (X(i-1) + X(i-2) + C(i-3) + Y(i-3)) \]
Thus we have shown there exists a very simple random
variable which bounds the beginning time for any job in a two
partition machine. The bound is half of the sum of the
processing times of all of the jobs which will run before it.

We next address the $m$ equal size partition case.

**Theorem 4.2**

For $i=1,2,\ldots,m$, let $B(i)=0$.

For $i=m+1,m+2,\ldots,n$, let $B(i) = 1/m \sum_{j=1}^{i-1} X(j)$.

Then for $i=m+1,m+2,\ldots,n$, $B(i) \leq B^*(i)$.

**Proof** Define an ordered set $U(i)$ as follows.

$U(i) = \{U(i;10), U(i;20), \ldots, U(i;m0)\}$,
where $U(i; 10)$ is the minimum element of $U(i)$ and for $1 \leq k < m$, $U(i; k^0) \leq U(i; (k+1)^0)$. Construct $U(m+1)$ from the first $m$ $X(i)$. Thus $U(m+1; 10)$ is the minimum of the first $m$ $X(i)$, the time the first available partition frees up. Thus $B(m+1) = U(m+1; 10)$ and $C(m+1) = B(m+1) + I(m+1)$. Replace $U(m+1; 10)$ by $C(m+1)$ and define

$$U(m+2) = \{U(m+2; 10), U(m+2; 20), \ldots, U(m+2; m^0)\}$$

$$= \{C(m+1), U(m+1; 20), U(m+1; 30), \ldots, U(m+1, m^0)\}.$$ 

Now $B(m+2) = U(m+2; 10)$.

In general, for $i = m+2, m+3, \ldots, n$, let

$$U(i) = \{U(i; 10), U(i; 20), \ldots, U(i; m^0)\}$$

$$= \{C(i-1), U(i-1; 20), \ldots, U(i-1, m^0)\}$$

$$= \{B(i-1) + X(i-1), U(i-1; 20), \ldots, U(i-1, m^0)\}.$$ 

Now $B(i) = U(i; 10)$, the minimum of $U(i)$ and since the minimum is less than or equal to the average,

$$B(i) \leq \frac{1}{m} \sum_{k=1}^{m} U(i; k^0)$$

$$= \frac{1}{m} \left( X(i-1) + B(i-1) + \sum_{k=2}^{m} U(i-1; k^0) \right)$$

$$= \frac{1}{m} \left( X(i-1) + U(i-1; 10) + \sum_{k=2}^{m} U(i-1; k^0) \right)$$

$$= \frac{1}{m} \left( X(i-1) + \sum_{k=1}^{m} U(i-1; k^0) \right)$$

$$= \frac{1}{m} \left( X(i-1) + X(i-2) + B(i-2) + \sum_{k=2}^{m} U(i-2; k^0) \right)$$
Thus, a bound on the beginning time for a given job is the sum of the processing times of all the jobs which must run before it averaged over the \( m \) partitions.

In the preceding theorems we have given a bound on the beginning time for an arbitrary job in the queue. Typically we will have to deal with a particular job whose beginning time will be determined only by those \( n \) jobs positioned in front of it in the queue. With this in mind we will henceforth only address the issue of the waiting time for the particular job. As we shall see, the job will be in a class and the subscript for its beginning time will denote its class rather than its position in the queue.
Many times the job mix for a computing facility has a wide variation in memory requirements among jobs. In these cases, instead of having uniform size partitions, the memory will be carved up into several size classes. This way, memory space is less likely to be wasted by having a small program run in a large partition. We address this situation next.

Let there be q size classes of jobs and let there be one partition for each class. The queue is composed of n jobs, some from each class r (r=1,2,...,q). X(r,i) will denote the processing time for a job of class r. Its position in the queue is i.

We are interested in the beginning time for a typical job from class j when it enters the system and is joined to the queue after job n. It may run in any of the partitions j,j+1,...,q. The dispatching mechanism is such that when a partition becomes available, the first job encountered searching through the queue which will fit is serviced. Denote the beginning time for this job from class j as B(j).

\[
\text{For } r=1,2,...,q, \text{ let } S(r) = \sum_{k=1}^{n} X(r,k). 
\]

S(r) is the sum of the times for all of the jobs in the queue which are from size class r.
**Theorem 4.3**

Let \( S(r) \) and \( B(j) \) be defined as above. Let

\[
B^*(j) = \frac{1}{(q-j+1)} \sum_{r=1}^{q} S(r).
\]

Then \( B(j) \leq B^*(j) \) for \( j=1,2,\ldots,q \).

**Proof.** For \( 1 \leq i \leq q \) and \( 1 \leq j \leq q \), let \( p(i,j) \) = the fraction of \( S(j) \) which is run in partition \( i \). Since a job can not run in a smaller partition than its own class, \( P \) is lower triangular. Consider the matrix equation

\[
T = PS.
\]

\( T(i) \) is the amount of time that partition \( i \) will be busy while the \( n \) jobs in the queue are processed.

For \( j=1,2,\ldots,q \),

\[
B(j) = \min_{1 \leq l \leq q} \{ T(l) \} \leq \frac{1}{(q-j+1)} \sum_{l=j}^{q} T(l).
\]

\[
T(l) = \sum_{r=1}^{q} p(l,r) S(r), \text{ and therefore}
\]

\[
B(j) \leq \frac{1}{(q-j+1)} \sum_{l=j}^{q} \sum_{r=1}^{q} p(l,r) S(r)
\]

\[
= \frac{1}{(q-j+1)} \sum_{r=1}^{q} S(r) \sum_{l=j}^{q} p(l,r). \quad (4.3.1)
\]

Now \( \sum_{l=j}^{q} p(l,r) \leq 1 \) by virtue of the fact that this sum runs down a column of the \( P \) matrix which contains fractions which in total sum to unity. Therefore
By letting $j = 1$ we see this is exactly the same bound that was obtained in the previous theorem. So the smallest jobs will tend to have the same completion times under multi-size memory classes as they had under the uniform size partition scheme. With increasing $j$, $B^*(j)$ gets larger. It was obvious this would happen since the larger jobs can run in fewer places, it will take longer to complete them.

In some cases it is desirable to dedicate more than one partition per class. This is true when the job mix is such that there typically is more expected processing time in one or more classes when compared to the other classes.

**Theorem 4.4**

Let $q = $ the number of job size classes.

For $1 \leq j \leq q$, let $m(j) =$ the number of partitions in class $j$.

Let $u(j) = \sum_{k=1}^{j-1} m(k)$, the number of partitions below class $j$.

Let $m = \sum_{j=1}^{q} m(j)$, the total number of partitions.

Let $B^*(j) = 1/(m-u(j)) \sum_{r=1}^{q} S(r)$.

Then $B(j) \leq B^*(j)$. 

\[
B(j) \leq \frac{1}{(q-j+1)} \sum_{r=1}^{q} S(r) = B^*(j). \quad \text{Q.E.D.}
\]
Proof

Let \( T = RS \), where

\[
R(1,1) \quad 0 \quad \cdots \quad 0 \\
R(2,1) \quad R(2,1) \quad \cdots \quad 0 \\
\vdots \quad \vdots \quad \ddots \quad \vdots \\
R(g,1) \quad R(g,2) \quad \cdots \quad R(g,g)
\]

where

\[
p(u(i)+1,j) \\
p(u(i)+2,j) \\
R(i,j) = \cdots \\
p(u(i)+m(i),j)
\]

\( p(u(i)+k,j) \) is the fraction of \( S(j) \) run in the \( k \)th member of the partitions of class \( i \).

Let \( h = u(j) + 1 \). By definition,

\[
B(j) = \min_{h \leq l \leq m} \{ T(l) \}
\]

\[
\leq \frac{1}{(m-u(j))} \sum_{l=h}^{m} T(l).
\]

\[
T(l) = \sum_{r=1}^{q} p(l,r)S(r) \text{ and therefore}
\]

\[
B(j) \leq \frac{1}{(m-u(j))} \sum_{l=h}^{m} \sum_{r=1}^{q} p(l,r)S(r)
\]
\[
q = \frac{1}{(m-u(j))} \sum_{r=1}^{m} \sum_{l=h}^{p(l,r)S(r)} \\
= \frac{1}{(m-u(j))} \sum_{r=1}^{q} \sum_{l=h}^{m} S(r) \sum_{l=h}^{p(l,r)}. \quad (4.4.1)
\]

But \( \sum_{l=h}^{p(l,r)} \leq 1 \) and therefore

\[
B(j) \leq \frac{1}{(m-u(j))} \sum_{r=1}^{q} S(r) = B^*(j). \quad Q.E.D.
\]

If we have only one partition in each class \( (u(j) = 1 \) for all \( j) \) then this reduces to the same bound as in Theorem 4.3. It is easy to see that again the larger jobs are less fortunate than the smaller ones.

Although we do not prove it, an argument can be made that a job from class \( j \) will be more likely to run in a partition of class \( j \) than one from class \( j+1 \); and it will be more likely to run in a partition of class \( j+1 \) than one of \( j+2 \), etc. The reason this is true is that it can not run in a class below it and the partitions above it will, in general, be busy running their own jobs. Some simulations were run to demonstrate this and it was found that the only time this was violated was when there was a vast discrepancy in the allocation of the number of partitions to the number of jobs. Specifically, the number of jobs was monotone decreasing by their size class and the number of partitions was monotone increasing by their size class. Thus the small
partitions were much overloaded causing many of the smaller jobs to be shifted to the larger partitions to be run. System designers are aware of the job memory requirements and should not let a situation like this develop. The number of partitions for each class should be decided on the basis of either expected processing times in each class (partitions in this proportion), or be biased toward the smaller jobs so as to not waste memory.

Recall that an element of the P matrix, \( p(i,j) \), is the fraction of the processing time of class \( j \) jobs which is done in partition \( i \). If we assume that jobs are less likely to run in each successively higher partition class then the columns of \( P \) will likely be monotone non-increasing below the diagonal. That is \( p(i,j) \geq p(i+1,j) \) for \( i \geq j \). The reason for this is that the time will accumulate in proportion to the number of jobs run because the processing times for the jobs are independent. In order to use this to any advantage, we will need the results pertaining to partial sums in the following lemma.

**Lemma**

Let

\[
\begin{align*}
(i) & \quad g \geq 1 \\
(ii) & \quad 1 \geq a(1) \geq a(2) \geq \ldots \geq a(q) \geq 0 \\
(iii) & \quad \sum_{i=1}^{q} a(i) = 1
\end{align*}
\]
Then for any integer $s$, $1 \leq s < q$

(a) \[
\frac{s}{q} \sum_{i=1}^{g} a(i) \geq 1 \quad \text{and} \quad (4.4)
\]

(b) \[
\sum_{i=s+1}^{q} a(i) \leq \frac{(q-s)}{q} \quad (4.5)
\]

**Proof**

Assume \[
\frac{s}{q} \sum_{i=1}^{s} a(i) < 1 \quad (4.6)
\]

Then \[
\sum_{i=1}^{s} a(i) < \frac{s}{q} \quad (4.7)
\]

Since the $a(i)$ are monotone non-increasing,

\[
\sum_{i=1}^{s} a(i) \geq sa(s) \quad (4.8)
\]

From 4.7 and 4.8, $s/q > sa(s)$

and \[
\frac{1}{q} > a(s) \quad (4.9)
\]

From 4.2 and 4.10,

\[
\frac{1}{q} > a(s) \geq a(s+1) \geq \ldots \geq a(q) \quad (4.11)
\]

From 4.3 and 4.6

\[
\sum_{i=s+1}^{q} a(i) > 1 - \frac{s}{q} \quad (4.12)
\]

Again since the $a(i)$ are monotone,

\[
\frac{(q-s) a(s+1)}{i=s+1} \geq \sum_{i=s+1}^{q} a(i) \quad (4.13)
\]

Then from 4.12 and 4.13,

\[
(q-s) a(s+1) > 1 - \frac{s}{q} = \frac{(q-s)}{q} \quad (4.14)
\]

which implies
a(s+1) > 1/q \quad (4.15)

Then from 4.11 and 4.15,

\[ \frac{1}{q} > a(s) \geq a(s+1) > \frac{1}{q} \quad (4.16) \]

and \( \frac{1}{q} > \frac{1}{q} \) is a contradiction so (a) is established.

From 4.3,

\[
s \sum_{i=1}^{q} a(i) + \sum_{i=s+1}^{q} a(i) = 1 \quad (4.17)
\]

Then \( \sum_{i=1}^{s} a(i) = 1 - \sum_{i=s+1}^{q} a(i) \) \quad (4.18)

From 4.4

\[
s \sum_{i=1}^{q} a(i) \geq s/q \quad (4.19)
\]

From 4.18 and 4.19

\[
1 - \sum_{i=s+1}^{q} a(i) \geq s/q \quad (4.20)
\]

which implies

\[
s \sum_{i=s+1}^{q} a(i) \leq 1 - s/q = (q-s)/q \quad (4.21)
\]

and (b) is established. Q.E.D.

We may now improve on the bound from Theorem 4.3.

**Theorem 4.5**

If for \( 1 \leq i \leq q-1 \) and \( 1 \leq j \leq i \),

\[ p(i, j) \geq p(i+1, j) \text{ then with} \]

\[ B^*(j) = \frac{1}{q} \sum_{r=1}^{j-1} S(r) + \frac{1}{q-j+1} \sum_{r=j}^{q} S(r), \]

\[ B^*(j) = \frac{1}{q} \sum_{r=1}^{j-1} S(r) + \frac{1}{q-j+1} \sum_{r=j}^{q} S(r), \]

\[ B^*(j) = \frac{1}{q} \sum_{r=1}^{j-1} S(r) + \frac{1}{q-j+1} \sum_{r=j}^{q} S(r), \]
\[ B(j) \leq B^*(j). \]

**Proof**  \nFrom Theorem 4.3, line 4.3.1, we have

\[ B(j) \leq \frac{1}{(q-j+1)} \sum_{r=1}^{q} S(r) \sum_{l=j}^{q} p(l,r) \]

\[ = \frac{1}{(q-j+1)} \left\{ \sum_{r=1}^{j-1} S(r) \sum_{l=j}^{q} p(l,r) \right\} \quad (4.22) \]

By construction, \( \sum_{r=1}^{l=j} p(l,r) = 1 \) if \( j \leq r \)  \quad (4.23)

Applying the lemma to the first sum of \( p(l,r) \) in 4.22,

\[ \sum_{l=j}^{q} p(l,r) \leq \frac{(q-(j-1))}{q} = \frac{(q-j+1)}{q} \text{ if } j > r. \quad (4.24) \]

Replacing the sums of \( p(l,r) \) in 4.22 by their applicable substitutions from 4.23 and 4.24 we may write

\[ B(j) \leq \frac{1}{(q-j+1)} \left\{ \sum_{r=1}^{j-1} S(r) (q-j+1)/q + \sum_{r=j}^{q} S(r) \right\} \]

\[ = \frac{1}{q} \sum_{r=1}^{j-1} S(r) + \frac{1}{(q-j+1)} \sum_{r=1}^{q} S(r) = B^*(j). \quad \text{Q.E.D.} \]

Comparing this with the result from theorem 4.3 we can see that the sum of \( S(r) \) up through \( j-1 \) is divided by the full value of \( q \), thus reducing the bound.

A similar result is achieved in the general case as follows.
Theorem 4.6

Let
\[
\begin{pmatrix}
a(1,1) & 0 & \ldots & 0 \\
a(2,1) & a(2,1) & \ldots & 0 \\
a & \vdots & \ddots & \vdots \\
a(q,1) & a(q,2) & \ldots & a(q,q)
\end{pmatrix}
\]
where \( a(i,j) = |R(i,j)| \), and \( R(i,j) \) is defined in Theorem 4.4. With \( h, m \) and \( u(j) \) also defined as in Theorem 4.4,

if for \( 1 \leq i \leq q-1 \) and \( 1 \leq j \leq i \),
\[
a(i,j) \geq a(i+1,j)
\]

\[
B^*(j) = \frac{1}{m-a(j)} \left( \frac{q}{q-j+1} \sum_{r=1}^{j-1} S(r) + \sum_{r=j}^{q} S(r) \right)
\]

then \( B(j) \leq B^*(j) \).

Proof

From Theorem 4.4, line 4.4.1,
\[
B(j) \leq \frac{1}{m-u(j)} \sum_{r=1}^{q} S(r) \sum_{l=h}^{m} p(l,r). \tag{4.25}
\]

By construction \( \sum_{l=h}^{m} p(l,r) = \sum_{l=h}^{q} a(l,r) \). \tag{4.26}

Applying the lemma to the \( a(l,r) \),
\[
\sum_{l=h}^{q} a(l,r) \leq \frac{q}{q-j+1}/q. \tag{4.27}
\]

By combining 4.25 and 4.26 we may write
\[
B(j) \leq \frac{1}{m-u(j)} \sum_{r=1}^{q} S(r) \sum_{l=j}^{q} a(l,r) = \tag{4.28}
\]
\[
\frac{1}{(m-u(j))} \left\{ \sum_{r=1}^{q-1} S(r) \sum_{l=j}^{q-1} a(l,r) + \sum_{r=j}^{q} S(r) \sum_{l=j}^{r} a(l,r) \right\} \tag{4.29}
\]

We know \( \sum_{l=j}^{q} a(l,r) = 1 \) for \( j \leq r \) \( \tag{4.30} \)

By applying 4.27 and 4.30 to 4.29 we have

\[
B(j) \leq \frac{1}{(m-u(j))} \left\{ \sum_{r=1}^{j-1} S(r) \frac{(q-j+1)}{q} + \sum_{r=j}^{q} S(r) \right\}
\]

\[
= \frac{1}{(m-u(j))} \left\{ \frac{(q-j+1)}{q} \sum_{r=1}^{j-1} S(r) + \sum_{r=j}^{q} S(r) \right\} = B^*(j). \quad Q.E.D.
\]

When compared with the result from Theorem 4.4 we see that the sum of the first \( j-1 \) \( S(r) \) is damped by a fraction less than one and a tighter bound results.

We have presented six theorems which give bounds and improved bounds on the beginning time for a job in any multiprogrammed computer with a fixed number of tasks.

Estimates of the completion times

A number of simulations were run to verify the results described earlier. These simulations led to some insights which enable us to estimate the beginning time for a newly entering job.

The simulations were run under a variety of conditions. For a given run, the number of partitions in a given class was fixed. Recall that partitions in class \( i \) are smaller than those in class \( i+1 \). The size class of a newly entering job was taken from either a uniform or triangular density.
The run times were either normal or exponential with the means sometimes the same for all classes, sometimes a function of class size. At the end of each simulation run, the P matrix was printed out and it was observed that sometimes it was exactly and other times, nearly the identity matrix. In trying to find out why this was the case, we looked at the expected times for each partition in a given class. We let $E(i) = E[S(i)]/(m(i)F)$, where

$$ F = \sum_{i=1}^{q} E[S(i)], $$ and $m(i)$ is the number of partitions in class $i$. The division by $F$ is simply to normalize the numbers so that they are less than one. Let $(E(1), E(2), \ldots, E(q))$ be denoted by $E$. We then observed that when the $E(i)$ were monotone increasing with $i$, $P$ was either the identity matrix or nearly so. Experiments 6, 7 and 11 in Table 1 are examples of this.

When the $E(i)$ were not monotone increasing with $i$, $P$ exhibited no interesting pattern, but another matrix did. This matrix, $W$, shows how much processing time has accumulated in each of the partitions. $W$ has $q$ rows. Each row $i$ has $m(i)$ entries, one for each partition in the class. It is a two dimensional representation of the $T$ vector used in the proofs of Theorems 4.3 and 4.4.

The interesting observation was that when $E(i) \geq E(i+1)$, the elements of rows $i$ and $i+1$ were nearly equal as
Table 1. P matrix observations from simulations.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>E</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(.168, .160, .176)</td>
<td>.949 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.051 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1</td>
</tr>
<tr>
<td>5</td>
<td>(.182, .162, .248)</td>
<td>.911 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.089 .990 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 .009 1</td>
</tr>
<tr>
<td>6</td>
<td>(.182, .322, .496)</td>
<td>.992 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.008 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1</td>
</tr>
<tr>
<td>7</td>
<td>(.060, .161, .496)</td>
<td>1 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1</td>
</tr>
<tr>
<td>11</td>
<td>(.262, .462, .828)</td>
<td>.985 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.015 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1</td>
</tr>
</tbody>
</table>

illustrated in Table 2.

In order to rationalize the results we sought an interpretation for E(i).

If instead of having one queue for all classes of jobs we had used one queue for each class then E(i) would have been a relative utilization expectation for each partition in the class. When E(i) are monotone increasing with i this means that each memory class is busier than the next smaller one. When they are not monotone increasing with i, it says that there is a class which is not so busy as the next
Table 2. $W$ matrix observations from simulations.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>E</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>(.500, .320, .176)</td>
<td>174.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>173.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>176.1</td>
</tr>
<tr>
<td>4</td>
<td>(.503, .160, .060)</td>
<td>87.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>86.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>88.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>86.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>86.5</td>
</tr>
<tr>
<td>8</td>
<td>(.500, .640, .176)</td>
<td>180.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>179.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>181.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>187.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>181.6</td>
</tr>
<tr>
<td>9</td>
<td>(.500, .320, .060)</td>
<td>105.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>103.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>104.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>105.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>104.9</td>
</tr>
<tr>
<td>10</td>
<td>(.262, .232, .276)</td>
<td>121.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>122.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>120.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>159.3</td>
</tr>
<tr>
<td>12</td>
<td>(.504, .496)</td>
<td>252.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>271.2</td>
</tr>
</tbody>
</table>

smaller one. If we then modify the dispatching rules so that some of the jobs may leave their own queue and join a higher one when it is not so busy there will be a tendency to equalize or level the loads. The leveling process may be illustrated by an analogy of fluid flow. Imagine there are several fluid conductors (pipes) as shown in Figure 1.
Figure 1. End view of fluid conductors.

The fluid flow is perpendicular to the page. The initial fluid level in each pipe (wavy lines) corresponds to the expected loading on each of the memory classes. Each pipe has the same cross-sectional area and is therefore capable of conducting the same amount of fluid per unit time. This corresponds to the equal job processing rate of the different partitions. There are valves between adjacent pipes which allow fluid to flow only from a lower numbered pipe to a higher numbered pipe. (There is a one-way trap door in the bottom of the source or lower level pipe.) This corresponds to jobs being able to move only to a larger memory for processing.

When the fluid levels are all increasing by their index number, all the fluid is conducted in its own pipe. But when
a lower level pipe is fuller than its neighbor, the fluid seeks its own level by forcing open the trap door and holding it open until the levels in the two pipes are the same height, the average of the initial heights (the dotted line).

If one of the middle pipes starts quite full compared to the ones on either side, some of the fluid will go to the next higher pipe. If enough goes, the middle one may fall below the one on the left of it and will be fed from there. Experiment 8 in Table 2 is an example of this.

What this means in our case is that if the $E(i)$ are increasing, a job must essentially wait only on the jobs in its class and they may be considered as being served by a single processor. When the $E(i)$ are decreasing, after the leveling process has happened, those classes for which the loads are equal may be treated as though they were in one class and all the partitions from all the classes involved may be treated as a single class. We require an algorithm which emulates the leveling process. It follows after an explanation of what it does.

We first compute the $E(i)$ but call them $L(i)$ because we may have to adjust them. For ease of description, we will refer to the partitions as pipes numbered one through $q$, arranged from left to right as in Figure 1. Beginning with the highest level (right-most) pipe, we compare adjacent loadings. As long as $L(j-1) \leq L(j)$ we do nothing since there
will be no pressure to flow from j-1 to j. But when we find a pair where \( L(j-1) > L(j) \) we know there will be a shift to the right. The shift may involve more than just the pairs j-1 and j so we enter a loop which computes the average loading of a string of adjacent pipes. This string initially has two elements, j-1 and j. If this average loading is smaller than the one to the left, we include that one in the string and recompute the average, etc. Eventually this will stop by either finding a pipe to the left of the string whose loading is less than the average of the string or we will exhaust the number of pipes. When it does stop, we must start the process from the beginning because the average of the string may now exceed the level of the pipe to the right of the string. When the process terminates, the \( L(i) \) will be monotone non-decreasing with i. There may be some level plateaus where the averaging has taken place.
Step 1. Let $F = \sum_{i=1}^{q} E[S(i)]$.

For $i=1,2,...,q$, let $L(i) = E[S(i)]/(m(i)F)$.

Step 2. Let $j = q$.

Step 3. If $L(j-1) > L(j)$ then go to Step 4. Let $j=j-1$ and if $j = 1$ then stop, otherwise go to Step 3.

Step 4. Let $j^1 = j$.

Step 5. Let $j^0 = j - 1$.

Compute $V = 1/(j^1 - j^0 + 1) \sum_{i=j^0}^{j^1} L(i)$.

If $L(j^0 - 1) \leq V$ then go to Step 6, otherwise let $j=j-1$ and go to Step 5.

Step 6. For $j=j^0,...,j^1$, let $L(j) = V$ and go to Step 2.

This procedure will compute $L(i)$ such that $L(i) \leq L(i+1)$ for $i=1,2,...,q-1$. For a string of $i$ from $i^0$ to $i^1$ where $L(i^0) = L(i^0+1) = ... = L(i^1)$, the jobs in these classes may be treated as though they were in a single class. Theorem 4.4 then applies to those in the string with the number of partitions in the single "class" determined by the number of partitions contributed by those classes in the string.

We must point out that the procedure outlined above is quite heuristic. The probable reason that it works is that the jobs in the simulations were scattered through the queue randomly with respect to size class. Of course this is what
one would expect in the normal course of events, but if the jobs would happen to be ordered by size class, it is doubtful that this procedure would predict very accurately.
CHAPTER V. CONCLUSIONS AND SUGGESTIONS

FOR FURTHER RESEARCH

We have developed, under certain assumptions, bounds for the completion times in any multiprogrammed computer with a fixed number of tasks. We have developed a scheduling algorithm or queueing discipline which increases the ability to meet deadlines over FCFS. The calculations necessary to implement the procedures would prohibitive for most ordinary job shops but would represent a trivial marginal increase in the cost of any present-day operating system and therefore the procedures are feasible.

In order to implement the procedures it is necessary to have at least approximations to the distribution functions which describe the processing times, X(i). This is basically a statistical problem in hypothesis testing and estimation. The simplest solution would be to gather lots of data from the machine on which the procedure will be implemented, plot histograms for the resident times and find a well-known distribution function which is a sufficiently good approximation. Then estimate the parameters of the distribution and test the hypothesis that it actually is that distribution. When all that is done, each job in the future would be considered to have been drawn from a random sample of independent identically distributed (iid) random variables. Some evidence of the ability to do this is
contained in (15).  This is a clean procedure and if the $X(i)$ are iid, one may be able to determine how far the bounds on the completion times are from reality. However, in the opinion of this writer, the deadlines which could be guaranteed under this procedure would likely be too large to be practical because they are monotone functions of the variance of the completion time, $C$.  We suspect, but have not shown, that a much lower variance of $C$ would result if the variances of the $X(i)$ could be made smaller by classifying the jobs into several sub-populations. For example in an academic environment, student jobs would typically have smaller means and variances when compared to the jobs at large. Another example, where there might be only one job in a sub-population, is that of data processing jobs where the dominant factor which determines how long a job is resident is the number of records in master files which do not change much.

Certainly a very interesting problem would be to determine the distribution functions for the flow times of jobs in a multiprogrammed environment with a variable number of tasks where the size and number of partitions dynamically changes according to the job stream. This problem will be much more complicated because of the potential problems of permanent blocking (13).
An intriguing thought is that the work described here might be applicable to fourth generation machines with virtual memories. Although it was probably not the intention of the designers of these systems (23), it is possible to think of the main memory being divided up into fixed sizes and the virtual memory's contents would be transferred in and out of the main frame as the program (process) dictates. Under these guidelines the work described in this paper would almost certainly apply.

The nagging problem this author has had during the time this problem has been under consideration has been what to do about the resources which are not capable of being multiplexed such as tape or disk drives. This problem, as it relates to permanent blocking, has been solved (13) but knowing a job will not be permanently blocked doesn't help much in pinpointing when it will actually complete. The tacit assumption in this thesis is that the jobs are sufficiently heterogeneous so that this is not a problem.

These are some of the thoughts for extension of this research which come to mind. Certainly as these are investigated, others will appear.
BIBLIOGRAPHY

1. Balke, Karl G. "Person-centered measurement." Presented at the seventh annual symposium on the interface between statistics and computer science, Iowa State University, October 18, 1973.

2. Balut, John Stephen. "Scheduling to minimize the number of late jobs when set-up and processing times are uncertain." Management Science 19, No. 11 (July 1973), 1283-1288.


ACKNOWLEDGEMENTS

I wish to express my sincere thanks and appreciation to Dr. C. G. Maple for his guidance and encouragement in this research and throughout my graduate program.

I also wish to extend special thanks to several members of the department of Statistics at Iowa State who have listened patiently to many of my ideas and offered many helpful suggestions. They are Doctors B. C. Arnold, D. L. Isaacson, W. J. Kennedy, G. Meeden and R. Mensing.

I take this opportunity to express my gratitude to my wife, Carolee, for her understanding and encouragement through these sometimes difficult years while I have pursued the degree.