INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313 761-4700 800 521-0600

Order Number 9315467

Scheduling periodic tasks among multiple processors

Hsu, Jane Chien Yueh, Ph.D.

City University of New York, 1993

U·M·I

300 N. Zeeb Rd.
Ann Arbor, MI 48106

SCHEDULING PERIODIC TASKS

AMONG

MULTIPLE PROCESSORS

by

Jane C. Hsu

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1993

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

12/22/92

Date

Professor Erdal A. Akkoyunlu

Chair of Examining Comittee

1/11/93

Date

Professor Stanley Habib

Executive Officer

Professor Frank Beckman

Professor Efstathos Zachos

Professor Domingo Rodriguez

Supervisory Comittee

THE CITY UNIVERSITY OF NEW YORK

Abstract

SCHEDULING PERIODIC TASKS AMONG MULTIPLE PROCESSORS

by

Jane C. Hsu

Advisor: **Professor Eralp A. Akkoyunlu**

This dissertation addresses the problem of scheduling periodic tasks on a multiple processor system. Individual requests of each of N task types arrive periodically with known fixed period and service requirement, and each request of a given task type must be completed before the next arrival of a request of its type. The tasks are processed on a system of M equivalent processors, $M \geq 2$. Though a task may be scheduled on more than one processor, it may run on only one processor at a time. For systems in which the total processing demand does not exceed that available ("feasible systems"), algorithms are presented that always produce a viable schedule. These algorithms are of two types. The "Queue-Based" Algorithms partition tasks into queues and assign queues to processors for servicing. The "Flat" Algorithms implement a modified deadline algorithm augmented with auxiliary deadline-type constraints that serve to prevent the typical failure modes of the standard deadline algorithm in the multiple processor setting. The Queue-Based Algorithms typically require tasks to be shared by and switched between processors, while the Flat Algorithms require additional preemptions of active tasks. For both classes of algorithms, the amount of switching required by the schedules produced is quantified and modifications which result in reduced processor switching are given.

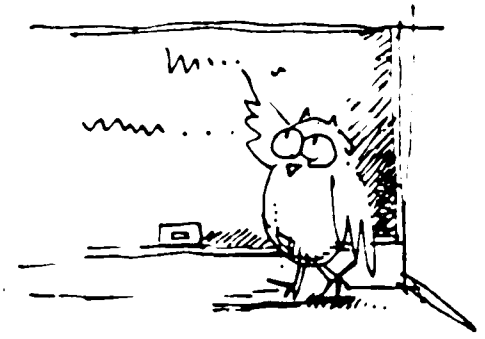
Acknowledment

There are many people to whom I wish to express my appreciation. First, of course, are my parents for all of their love and caring. Then, my husband, for all of his patience and support.

For his commitment in seeing me through to the end, I am forever indebted to my advisor, Professor Eralp Akkoyunlu. He introduced me to distributed systems reaearch, and he was always eager to help, even when he was sailing faraway seas. From start to finish he kept me on-course by calling, faxing, etc. to check on my progess.(I was also checking on him, to make sure he did not just disappear!) In Professor Akkoyunlu's absence, Professor Stathis Zachos could always be counted on to make sure I did not skip(or finish) any meals. I will always appreciate his taking an interest in my work. Thank you also to Professor Domingo Rodriguez for his encouragement and for traveling from Puerto Rico to sit on my committee. Professor Frank Beckmen admitted me to the Ph.D. program and has always been one of the nicest and most helpful people I know at the Graduate Center. There are four other friends who have helped me in many different ways over the last few years: Mr. Joe Driscoll, Mr. Jacob Weiss, Mr. Yung-Chang Hsu, and Ms. Shu Ling Chen. Finally, my thanks to the good ship *YOSUN*, for safely returing Professor Akkoyunlu from his adventure around the world.

Table of Contents

Part I



Background and Feasibility

Chapter 1

Introduction

This research dissertation falls in the general area of scheduling. The specific problem considered is that of scheduling periodic tasks on a multiple processor system. We require that each individual task must be completed before the next periodic request. Only one processor may work on an individual request at any given time.

The problem of scheduling periodic tasks under the constraint that each individual request of a given task be completed before the next periodic request is encountered in hard real-time environments where the service requirements of each task and its recurrence rate are known in advance. Examples of such environments include aircraft and manufacturing systems, where periodic sensor data must be processed and control signals sent before the arrival of the next packet of sensor data. In telecommunications, transmission lines/channels must be shared between data and voice users. Voice users need the channel in a strictly periodic fashion; data users may get the channel only when voice users are not present.

The general problem of scheduling tasks on multiple processor systems has received increasing attention in the research community as such systems become more prevalent and real-time applications proliferate. A compendium of on-going research, available techniques, and new approaches is provided by Van Tilborg and Koob [23, 24]. From the Foreword of [23]:

Of course, the general problem of scheduling resources optimally is NP-hard, and the addition of deadline timing constraints offers no relief from that fact of life. Somewhat surprisingly, however, research over the past few years has demonstrated that there are many practical, sometimes subtle and counter-intuitive, techniques that can be used to extract both predictable performance from real-time systems and also better overall ability to satisfy deadlines.

The real-time scheduling problem has been studied in the literature in various forms [23]. Typically the schedule to be derived is expected to satisfy all constraints, without any room for error. However, some researchers consider models which tolerate an occasional missed deadline [19]. This is a reasonable approach for some applications, such as updating a raster display, where a missed update is unlikely to have any effect provided it does not occur too often. In some studies [16] the tasks to be scheduled are composed of a set of periodic, predictable tasks mixed with a stream of randomly arriving aperiodic tasks. This model is important in applications where there are two kinds of chores: a set of periodic updates, to which is superimposed the need to respond to unanticipated events. In this dissertation, however, a schedule will only be considered valid if all deadlines are met, and all tasks will be periodic, i.e., there are no aperiodic or randomly arriving tasks to be serviced.

At present there is no known algorithm which is guaranteed to generate a valid schedule for periodic tasks with deadline in feasible multiproces-

sor applications. The term *feasible* refers here to the assumption that the required total computational load of the multiple tasks does not exceed the available processing capacity. The goal of this dissertation is to develop, analyze, and evaluate new algorithms for scheduling periodic tasks with deadline in feasible multiple processor systems. To be valid, a schedule must meet all deadlines. In addition, a valid schedule must not schedule a task to run on more than one processor at any given time. Thus, the systems addressed in this dissertation are assumed to satisfy the following restrictions:

R1 No task requires more processing than can be provided by a single processor.

R2 No task may run on more than one processor at any given time.

Clearly, **R1** is necessary for **R2** to be satisfiable. In addition, throughout this dissertation we will assume that tasks are independent. By this we mean that the processing of individual requests of a given task does not depend on the processing of other tasks. We will also assume that the run-time of individual requests of a given task is constant and does not depend on which processor executes the request. Thus, the processors are assumed identical (i.e., equally powerful) with respect to the tasks. Finally, our initial model assumes that the time required to switch a processor from one task to another is zero. The importance of this assumption will be clear later on when we present our algorithms for solving such scheduling problems. As we will see, scheduling peri-

odic tasks on multiprocessor systems often requires pre-empting tasks actively being processed by newly arriving tasks.

Our research has focused on two classes of scheduling algorithms:

- **Queue-based algorithms**, which impose a structure by partitioning the set of tasks into queues, and restricting processor sharing to at most a single task being serviced by two *neighboring* processors.

- **Flat algorithms**, where a modified deadline algorithm is applied to the unstructured set of tasks.

Queue-based algorithms, such as "Algorithm A" presented in Chapter 3, provide a straightforward methodology for generating the desired schedules. Similar to the approach described in [7], Algorithm A works in two stages: a task assignment stage and a task scheduling stage. Algorithm A assigns most tasks to individual processors and limits the sharing of any task to at most two *neighboring* processors, i.e., processors with successive labels. A cyclic schedule is then derived for each processor. Within each cycle the tasks are processed in order with each task getting its proportionate share of processing time. In addition, Algorithm A never schedules a task for service on more than one processor at the same time. Algorithm A produces a valid schedule whenever it is possible to do so, and therefore we say that Algorithm A is optimal. In addition, in schedules generated by Algorithm A a task may be shared by at most two processors. Finally, Algorithm A yields a sched-

ule feasible on the minimum number of processors, while the algorithms presented in [7] may require as many as twice that number. As noted earlier, we will first work under the assumption that the time required for a processor to switch from one task to another is zero. In actuality, the time required to perform such a switch, though small, is not zero and so each switch has a cost in terms of lost processing. Thus, as part of our analysis we study the switching properties (e.g., the maximum number of switches required) of Queue-Based Algorithms and Flat Algorithms. In particular, we quantify the worst-case cost of switching, and develop algorithmic modifications to reduce this cost.

Our second class of algorithms, the Flat Algorithms, involves the application of a modified deadline algorithm to the unstructured set of tasks. It is well known (see Liu and Layland [20]) that the deadline algorithm always generates a valid schedule for feasible single processor systems. However, it is equally well known that the straightforward deadline algorithm can fail in the multiple processor setting. In the Flat Algorithms we introduce techniques for modifying the deadline algorithm, primarily through the incorporation of additional deadline constraints, in order that a valid schedule is guaranteed. In addition, we derive bounds for the switching rate of the Flat Algorithms.

The dissertation is organized as follows. In Chapter 2 we introduce notation, basic concepts and assumptions, and give more detailed descriptions of commonly studied approaches to scheduling single and multiple processor systems. In Chapter 3 we describe Algorithm A, a partition-

based algorithm, and give the necessary and sufficient conditions for it to generate a viable schedule for feasible systems. We illustrate the type of schedules generated by Algorithm A by examples involving two processor systems. In Chapter 4 we consider the switching properties of the schedules generated by Algorithm A and develop upper bounds on the number of switches required. We show these bounds may be improved by various modifications and extensions. Chapter 5 describes the first of these more general queue-based algorithms, the Fair algorithm and its "Flip-Flop" variant. Chapter 6 gives the second queue-based algorithm, the Balanced algorithm, which has additional good properties. In Chapter 7 we introduce the Flat Algorithms, our primary examples being the Paris and KL Algorithms. In Chapter 8 we study the Flat algorithms in detail and establish the optimality of the Paris and KL Algorithms. In Chapter 9 we derive bounds on the switching rate of the Flat Algorithms and show that Paris Algorithm is event-driven in the multiprocessor setting. In Chapter 10 we present the results of our exhaustive simulation studies of the switching rates of the various scheduling algorithms. In Chapter 11 we summarize our results and pose problems for future research.

Chapter 2

Notation, Basic Concepts, and Background

2.1 Tasks

The systems and scheduling problems considered in this dissertation involve a set S of N periodic tasks and a setΠ of M processors. In its most general form, a periodic task $J \in S$ is characterized by the quadruple

$$J = (p, e, s, d)$$

where

p is the task period,

e is the computation time required during each period,

s is the task offset (i.e., the arrival times for the individual task requests are $s + kp, k \geq 0$), and

d is the deadline (i.e., the e units of service must be completed within d units of time after the arrival of the task request).

A very important special case occurs when $s = 0$ and $d = p$ for all $J \in S$. Such a system is called *synchronous*. Each task in a synchronous system

is represented as the ordered pair $J = (e, p)$. In a synchronous system, each request of a given task must be completed before the arrival of the next request of the task. Thus, a task J requires exactly e units of processing during the interval

$$[kp, \quad (k+1)p), \qquad k \geq 0$$

In this dissertation we will work only with synchronous systems. We note that the modifications required to apply our results to asynchronous systems are not straightforward, and that the problem of scheduling asynchronous systems is generally more difficult than scheduling synchronous systems.

For synchronous systems we define for each task

$$\mu = \frac{e}{p}$$

as a measure of the computing power (i.e., the fraction of one processor) required by the task. We confine our attention to systems of N tasks

$$S = \{A = (e_A, p_A)\}$$

which satisfy the restriction

R1 $e_A \leq p_A$ holds for all tasks $A \in S$.

and so the relation

$$0 \leq \mu_A \leq 1$$

holds for every task A. This is equivalent to assuming that no single task requires more computation than can be supplied by a single processor. In addition, we will assume that individual tasks are not parallelizable, and so impose the restriction

R2 No task may be scheduled to run on more than one processor at the same time.

A system for which there exists a valid schedule is called feasible. The relation

$$\sum_{A \in S} \mu_A \leq M$$

bounds the total amount of processing required to service the tasks by the number of processors and is clearly a necessary condition for feasibility. Our goal is to develop algorithms for generating these schedules. We say a scheduling algorithm is *optimal* when it generates a valid schedule for any system that satisfies this relation.

2.2 Schedules in General

A schedule σ is a mapping

$$\sigma : \Pi \times T \rightarrow S \cup \{idle\}$$

$\sigma(k,t)$ remains constant in intervals of the form $[t_1, t_2)$ and $\sigma(k,t)$ denotes the task which is active on processor k at time t.

In priority-driven scheduling each task is assigned a priority which determines all scheduling decisions: high priority tasks are serviced before lower priority tasks. *Fixed priority* algorithms [7, 18] associate a single unchanging priority with each task. *Dynamic priority* [15, 17, 20] algorithms, on the other hand, allow for the priority of a task to change between different periodic arrivals of the task, as well as during a given period.

Both *preemptive* and *non-preemptive* schedules have been considered in the literature. In preemptive scheduling, a task can be interrupted and remain temporarily inactive while a higher priority task receives service. In non-preemptive algorithms [26], a task which has been activated always runs to completion without interruption. These algorithms are obviously less flexible, in that they fail for some systems which can be scheduled by preemptive algorithms. Nonetheless, such an approach may be acceptable in applications where, for example, many tasks share common data in such a way that a partially completed transaction would have to be protected from other tasks.

2.3 Single Processor Scheduling Algorithms

The single processor scheduling problem has been studied in the literature in various forms [23]. In real-time scheduling, also referred to as deadline-driven scheduling, a schedule must satisfy all task deadlines to be acceptable. Alternatively, one may allow schedules which miss deadlines provided the results obtained from tasks which are terminated before completion are sufficiently precise [19]. Scheduling with both lateness and earliness penalties has also been considered [3].

The problem of scheduling periodic tasks on a single processor has received considerable attention. The three best known algorithms for the one-processor case are the Rate-Monotonic Algorithm, the Deadline Algorithm, and the Slacktime Algorithm:

- The *Rate-Monotonic Algorithm* [7] (for scheduling periodic tasks) gives highest priority to tasks with the highest recurrence rate (i.e., the smallest period). It is a fixed priority scheduling algorithm and is not pre-emptive.

- The *Deadline Algorithm* [2, 4, 18] schedules for immediate processing (i.e., gives highest priority to) the task with the earliest deadline. It is preemptive.

- The *Slacktime Algorithm* [21] schedules for immediate processing (i.e., gives highest priority to) the task with the smallest slacktime. It is

preemptive.

The slacktime of a task at a given time t is defined as follows: if d_0 is the time of the next deadline and e_0 is the remaining service which must be provided to the task before this next deadline, the slacktime at time t is the quantity $(d_0 - t) - e_0$. Thus, the Deadline Algorithm selects the task with the most pressing deadline to run next, while the Slacktime Algorithm selects the task for which we have the least surplus (i.e., *slack*) time remaining. For both algorithms, in the case of a tie all such tasks are run in round-robin fashion.

Recall that a scheduling algorithm is said to be optimal if it produces a valid schedule in all cases where it is possible to do so, i.e., for all feasible systems. Liu and Layland have shown that the Deadline Algorithm always generates a valid schedule for feasible single processor systems [20, Theorem 7] and so the Deadline Algorithm is optimal in the single processor setting. However the straightforward Deadline Algorithm is subject to failure in the multiple processor setting. We will see examples of how this can happen in the next section.

2.4 Pitfalls when Scheduling on Multiple-Processor Systems

Algorithms which are optimal with one processor may fail for two-processor systems. For example, the two-processor system

$$A = B = (2,4), \qquad C = (4,5)$$

can be scheduled by the Slacktime Algorithm but not by the Deadline Algorithm. This is because the Deadline Algorithm will ignore C until it is too late. Conversely, the system

$$A = B = C = (2,4), \qquad D = (4,8)$$

can be scheduled by the Deadline Algorithm, but not by the Slacktime Algorithm. This is because the Slacktime Algorithm would run a round-robin of A, B, and C during the interval $[0,3)$ using both processors, leaving one processor idle during the interval $[3,4)$, causing failure. Note that the slacktime algorithm has to service *all* tasks with the smallest slacktime in a round-robin, otherwise it would fail to schedule the two-processor system

$$A = B = C = (2,3)$$

2.5 Multiprocessor Scheduling Algorithms

The multiple processor scheduling problem has also received considerable attention in the literature [23]. The use of algorithms found to be effective for scheduling single processors, such as the Rate-Monotonic

Algorithm, the Deadline Algorithm, and the Slacktime Algorithm, for scheduling multiple processor systems has proven to be of enduring interest. While we have seen in the previous section that the simple application of these algorithms to multiple processor systems can lead to disaster, they are of great utility when suitably augmented.

Most algorithms proposed for scheduling on multiple processors generally fall into one of two categories;

- *Partitioning algorithms* which distribute the tasks among the processors and then apply an optimal algorithm (such as a Deadline Algorithm) locally.

- *Non-partitioning algorithms* which keep all tasks in one pool and select the task to be processed next according to an assigned priority.

Perhaps the first partitioning algorithm to appear in the literature was that of Horn [13]. He considered both single and multiprocessor scheduling with the objective of minimizing maximum lateness and total delay. His approach to partitioning involved assigning tasks to processors in a cyclic fashion. Partitioning algorithms for scheduling real-time multiprocessor systems have been studied by Dhall and Liu [7]. They proposed a partitioning strategy whereby a task is assigned to a processor provided it will be scheduled to run on that processor before its deadline by a locally applied single-processor scheduling algorithm.

Questions relating to the complexity of scheduling periodic tasks with deadlines on a multiprocessor system have also been addressed in the literature. Leung and Whitehead [18] have shown that in the fixed-priority case, the problem of determining whether or not such a schedule exists is generally NP-hard and that partitioning and non-partitioning techniques are incomparable in the sense that there are examples in which one method is successful but the other one fails.

Chapter 3

Algorithm A

In this chapter we introduce a partition-based scheduling algorithm we call *Algorithm A*. Algorithm A has a fixed system interval we call a scheduling *quantum*. Algorithm A assigns most tasks to individual processors and limits the sharing of any task to at most two neighboring processors, i.e., processors with successive labels. Most tasks will be assigned to one processor only, and any sharing will be limited to at most one task by each pair of neighboring processors. A cyclic schedule is then derived for each processor. This schedule has a period equal to the system quantum. The task periods are assumed, without loss of generality, to take integer values (in some unspecified unit of time). Within each cycle, the tasks are processed in order with each task getting its proportionate share of processing time. Finally, Algorithm A never schedules a shared task for service on more than one processor at the same time.

Recall that in order for a system be feasible it must satisfy the condition

$$\sum_{A \in S} \mu_A \leq M$$

since M is the total computing power of M processors over a unit interval of time while the summation on the left hand side is the amount of processing required by the tasks over the same timespan. We show in

this chapter that Algorithm A always produces a viable schedule for such a system, in effect giving a constructive proof that the condition is also sufficient. Since Algorithm A always generates a valid schedule for a feasible system, we say it is optimal. As we will see, Algorithm A has complexity linear on the number of tasks [28].

As mentioned earlier, for now we will work under the assumption that the time required for a processor to switch from one task to another is zero. In actuality, the time required to perform such a switch is small but not zero. Thus, as part of our analysis we study the switching properties (e.g., the maximum number of switches required) of Algorithm A.

3.1 System Period and Quantum

The definition and analysis of Algorithm A require us to assume, without loss of generality, that task periods take integer values (in some unspecified unit of time). We define the *system period*

$$T = LCM\{p_A : A \in S\}$$

and the system *quantum*

$$q = GCD\{p_A : A \in S\}$$

The following two facts are obvious, but will prove useful as we continue our development.

Fact 3.1.1: T is the system period in the sense that any schedule which is valid for the interval $[0, T)$ can be used to generate the steady state schedule, i.e.,

$$\sigma(k, t) = \sigma(k, t \bmod T)$$

Thus, it suffices to generate a schedule for the interval $[0, T)$.

Fact 3.1.2: If σ is a valid schedule for a system, then so is σ^{REV}, the reverse of σ where

$$\sigma^{REV}(k, t) = \sigma(k, T - t)$$

Thus, viable schedules are typically not unique.

3.2 Algorithm A

Algorithm A works in two steps: a task assignment step and a task scheduling step. Most tasks are uniquely assigned to one processor, though each processor may share at most one task with each of its neighbors. A cyclic schedule is then derived for each processor. Within each cycle the tasks are processed in order with each task getting its proportionate share of processing time.

At the task assignment step, each processor is assigned a set of local

tasks for which it supplies all the computing required. In addition, there are at most (and in general) $M-1$ **shared** tasks

$$\{SHARED^k, ..., 0 \leq k < M\}$$

Each shared task $SHARED^k$ will receive its processing time from the two neighboring processors π^k and π^{k+1}. Each processor runs the tasks it has been assigned in a round robin during each scheduling quantum q. That is, during each time interval $[jq, (j+1)q)$, $j = 0, 1, 2, ...,$ processor k services, in order, the task it shares with processor $k-1$, its local tasks, and the task it shares with processor $k+1$, allocating to each the appropriate amount $(\mu_i \times q)$ of service.

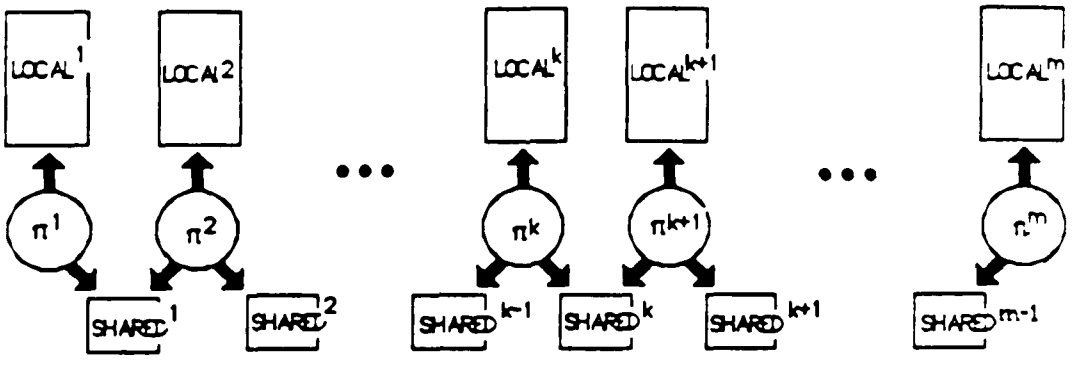


Figure 3.2.1: Topology of Algorithm A

Step 1: Partition the tasks into queues.

We first define the partial sums that represent the cumulative processing

requirement of tasks $1, 2, ..., h$, for $h = 1, 2, ..., N$. Let

$$s_h = \sum_{i=1}^{h} \mu_i, \quad 0 \leq h \leq N$$

The idea is that each time s_h exceeds the integer $k, k = 1, 2, ..., M-1$, the task associated with the exceedance is to be shared by processors k and $k+1$, while the intervening tasks are assigned to a single processor. More explicitly, letting $h(k)$ denote the value of h such that

$$s_{h(k)-1} < k < s_{h(k)}, \quad k = 1, 2, ..., M-1$$

and setting $h(0) = 1$ and $h(M) = N+1$, then task $J_{h(k)}$ is shared by processors k and $k+1$, while the tasks $J_{h(k-1)+1}, J_{h(k-1)+2}, ..., J_{h(k)-1}$ are assigned to (i.e., local to) processor $k, k = 1, 2, ..., M$. Thus, each processor k is associated with three queues:

1. the queue consisting of the task shared with processor $k-1$,

2. the queue consisting of the its local tasks, and

3. the queue consisting of the task shared with processor $k+1$.

These queues are defined as follows:

a. for $j = 2k-1$, the local queue Q_j of processor k is:

$$LOCAL^k = Q_j = \{J_i : \quad 1 < i < N : \quad k-1 \leq s_{i-1} \leq s_i \leq k\}$$

b. for $j = 2k$, Q_j consists of a single task shared by processors k and $k+1$:

$$SHARED^k = Q_j = \{J_i : \quad s_{i-1} < k < s_i\}$$

Step 2: Define the scheduling sequence.

During each scheduling quantum q processor k services, in order

a. the shared queue $SHARED^{k-1}$,

b. its local queue $LOCAL^k$, and

c. the shared queue $SHARED^{k+1}$.

Step 3: Define the processing times.

For processor $\pi^k, 1 \leq k \leq M$, we define the values μ_L^k and μ_R^k, the processing required by the (part of) the task it shares with its left-hand neighbor π^{k-1} and its right-hand neighbor π^{k+1}, respectively, as follows:

$$\mu_L^k = \begin{cases} 0, & \text{if } k = 0 \text{ or } SHARED^{k-1} = \emptyset, \\ s_i - (k-1), & \text{if } SHARED^{k-1} = J_i. \end{cases}$$

Similarly,

$$\mu_R^k = \begin{cases} 0, & \text{if } k = M \text{ or } SHARED^k = \emptyset, \\ k - s_i - 1, & \text{if } SHARED^k = J_i. \end{cases}$$

Thus, for each shared task $A = SHARED^k$ we have

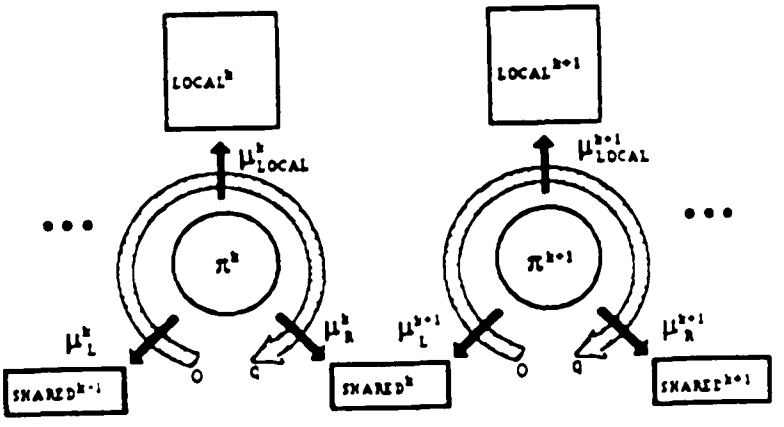
$$\mu_L^k + \mu_R^{k+1} = \mu_A$$

During an interval of duration q, each processor k allocates each of its queues the appropriate amount of service, which for $k = 1, 2, ..., M$ equals

a. μ_L^k for the shared queue $SHARED^{k-1}$,

b. $\sum_{i=h(k-1)+1}^{h(k)-1} \mu_i$ for the local queue $LOCAL^{k-1}$, and

c. μ_R^{k+1} for the shared queue $SHARED^k$.



Step 4:

We are now able to produce a schedule σ_0 for the interval $[0, q)$. This

is the collection of individual schedules for each processor k

$$\sigma_0[0,q) = \{\sigma_0^k([0,q)), 1 \leq k \leq M\}$$

so that $\sigma_0[k,t) = \sigma_0^k(t)$. To form the schedule σ_0^k for processor k, we define

$$t_L^k = q \times \mu_L^k \quad and \quad t_R^k = q \times (1 - \mu_R^k).$$

For $i : J_i \in LOCAL^{k-1}$, let

$$t_i^k = q \times (s_{i-1} - (k-1)).$$

We now specify σ_0^k, the schedule for π^k over the interval $[0,q)$, as follows:

1. $\sigma_0^k([0,t_L^k)) = SHARED^{k-1}$

2. If $LOCAL^{k-1} = \emptyset$, then $\sigma_0^k([t_L^k, t_R^k)) = \{idle\}$.

Otherwise, for $i : J_i \in LOCAL^{k-1}$, $\sigma_0^k([t_i^k, t_i^k + q\mu_i)) = J_i$.

Note that if $J_N \in LOCAL^{k-1}$ then $\sigma_0^k([q(s_N - (k-1)), q)) = \{idle\}$.

3. $\sigma_0^k([t_R^k, q)) = SHARED^k$.

Step 5: Iterate the quantum schedule.

The final schedule σ is generated by repetition of the schedule σ_0 generated for the interval $[0,q)$. In particular, the schedule for the system

period T is

$$\sigma_T = \{\sigma_T^k, 1 \leq k \leq M\}$$

where

$$\sigma_T^k(t) = \sigma_0^k(t \bmod q), \quad 0 \leq t \leq T$$

is the local schedule of processor k. *End of Algorithm A*

Fact 3.2.1 Optimality Theorem

For any feasible system satisfying R1, Algorithm A produces a valid schedule.

Proof:

The proof that Algorithm A produces a valid schedule σ follows from assertions 1, 2, and 3 which follow.

1. σ_0 is well defined, i.e., σ_0^k, as specified in Step 4 is well-defined. Specifically,

   a. $0 \leq t_L^k \leq t_R^k \leq q$,

   b. $t_L^k = t_{i0}^k$, $i0 = \min \{i : J_i \in LOCAL^{k-1}\}$

   c. $t_i^k + q\mu_i = t_{i+1}^k$, $\forall i : J_i, J_{i+1} \in LOCAL^{k-1}$

   d. $t_{ix}^k + q\mu_{ix} = t_R^k$, $ix = \max \{i : J_i \in Q2k-1\}$

These all follow directly from the definition of

$$s_i, t_i, \qquad 1 \leq i \leq N$$

and

$$t_L^k, t_R^k, \qquad 1 \leq k \leq M$$

2. Each task $J_i \in S$ receives exactly e_i units of computation time during each period p_i. This is easy to see. First, each task J_i receives $q\mu_i$ units of service during each quantum q (if $J_i = SHARED^k$, this total is made of $q\mu_R^k$ units from processor k, and $q\mu_L^{k+1}$ units from processor $k+1$). Then, since q divides p_i, each period p_i consists of exactly (p_i/q) quanta of length q. Thus the total service received by J_i during each period is

$$(p_i/q)q\mu_i = e_i$$

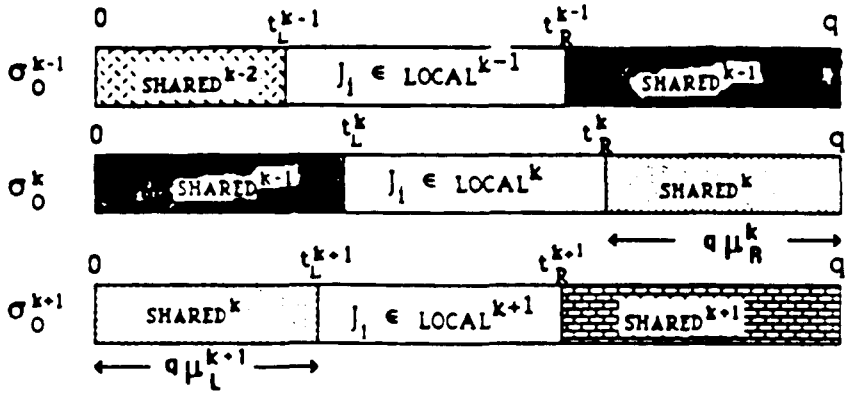
3. No (shared) task is scheduled on two processors at the same time. This is a straightforward consequence of the constraint

$$\mu_i \leq 1, \quad 1 \leq i \leq N$$

from which

$$t_L^{k+1} \leq t_R^k, 1 \leq k < M$$

immediately follows. QED



No overlap for SHARED^k since :

$$q\mu_L^{k+1} + q\mu_R^k - q\mu_1 \leq q$$

Figure 3.2.2 Scheduling System of Algorithm A

Part II

Queue-Based Algorithms

Chapter 4

Switching Rates And Event-Driven Schedules

In a practical system, dispatching a task involves an overhead which was ignored in the idealized model, for which the constraint

$$\mu = \sum_{A \in S} \mu_A \leq M \tag{4.1}$$

is sufficient. This overhead, caused by process switching (when one task is interrupted in favor of another) consists of a fixed number ϵ of processor cycles wasted during each switch. This loss of processing can be meaningful if

a. $\mu \approx M$, or

b. $e_i \approx \epsilon$ or $p_i \approx \epsilon$.

In such cases, schedules which minimize the rate of process switching become highly desirable. At the same time, the computational complexity of scheduling algorithms which compute schedules which are efficient from the point of view of switching rate tends to be high: We suspect that finding the best schedule (e.g., a valid schedule with the minimum number of switches) is NP-hard.

In this chapter we introduce basic quantities and notation used in studying the switching properties of scheduling algorithms. We count as a switch both preemptions of one task by another task on an active processor and the run-initialization of a task on a previously inactive processor. We study the tradeoff between the computational complexity of the switching algorithms and the (switching) efficiency of the schedules they produce. Complexity theory provides us with well-understood measures for assessing the computational complexity of our scheduling algorithms. To obtain a meaningful measure for the switching rate of our schedules, we consider the number of switches in the fundamental system period

$$T = LCM\{p_A : A \in S\}.$$

This is reasonable because if σ_0 is a schedule for the interval [0,T), then σ^* , the infinite repetition of σ_0, is a schedule for [0, ∞).

4.1 Switching Basics

We will use the quantity

$$L_0 = \sum_{A \in S} \frac{T}{p_A},$$

the number of task arrivals during the interval [0,T), as a yardstick by

which to measure the switching performance of the schedule σ_0 . L_0 is an appropriate measure because

a. it is easy to compute and depends only on fixed, known task parameters, and

b. it is, in fact, a lower bound on the number of switches in the interval [0,T) since each arriving task must be dispatched.

Note that L_0 is a loose lower bound. For example, the system

$$S = \{A, B, C\}, \qquad A = B = C = (2, 3)$$

has $T = 3$ and $L_0 = 3$. Yet any schedule must have 4 switches in the system period [0,3).

We define one class of efficient schedules which we call **event-driven.** An event consists of

- the arrival of a task, or
- the completion of a task.

A schedule is event-driven if,

- every process switch is associated with an event, and

- each event is associated with at most one switch.

It is clear that, for an event-driven schedule, the number of switches in the interval [0,T) is bounded from above by $2L_0$.

FACT 4.1.1 : In the single processor case, the Deadline Algorithm is event-driven.

Proof: In the deadline algorithm, a task is dispatched only

when it arrives, or

when some other task completes. *QED*

FACT 4.1.2: In the single processor case, the Deadline Algorithm is not minimal.

Proof: The Deadline Algorithm applied to the system

$$S = \{A, B\}, \quad A = (1,2), \quad B = (4,8)$$

yields the schedule

$$\sigma = ABABABBA$$

with 7 switches; but there is a schedule

$$\sigma' = ABBAABBA$$

with 6 switches. *QED*

4.2 Switching Properties of Algorithm A

We will now derive an upper bound for the number of switches required by Algorithm A during a system period [0,T). In studying the switching rates of partition-based and queue-based algorithms such as Algorithm A, we distinguish between two types of switching:

- **A queue switch (global switch)** occurs when a processor preempts a task in one queue in favor of a task in another queue.

- An **internal switch (local switch)** occurs when a processor preempts one task for another task of the same queue.

It is fairly straightforward to calculate the maximum number of switches required by Algorithm A. Recall that Algorithm A constructs a cyclic schedule on subintervals of duration q (the system quantum) given by

$$q = GCD(\{p_A : A \in S\})$$

and so there are $\frac{T}{q}$ quanta in one system period. Since q is bounded below by 1, there are at most T quanta in a system period. During each system quantum, each of the (at most) $M - 1$ shared tasks is run twice (once by each processor by which it is shared), while the remaining tasks

are run once. Thus, the total number of switches for Algorithm A in the interval [0, T) is bounded above by

$$T(N + M - 1)$$

As we will see, it is possible to reduce the rate of switching from that required by Algorithm A.

Chapter 5

The Queue-Based Algorithms

Our simplest queue-based algorithm is Algorithm A. Algorithm A produces a valid schedule whenever it is possible to do so, i.e, for all feasible systems. In addition, Algorithm A never schedules a task for service on more than one processor at the same time. As noted earlier, we are working under the assumption that the time required for a processor to switch from one task to another is zero. In actual applications, the time required to perform such a switch is small but not zero. As we have shown in the previous chapter, Algorithm A involves a high level of process switching, requiring as many as $T(N + M - 1)$.

In this chapter we will introduce other queue-based algorithms with reduced amounts of switching. As a class, the queue-based algorithms generalize Algorithm A by relaxing the requirement that the *system interval* be fixed. That is, the notion of system quantum introduced in the context of Algorithm A is replaced a notion of system interval which is not fixed but rather dynamically defined in terms of the task deadlines. The payoff for using an alternative system interval definition is a reduction in the number of switches. Like Algorithm A, queue-based algorithms impose structure by partitioning the tasks into queues, but in the more general queue-based algorithms the notion of a shared task is replaced by the notion of a shared queue being serviced by neighboring processors. We will characterize the switching properties of queue-based

algorithms in terms of their worst case performance with upper bounds on the number of switches required.

5.1 Queue-Based Algorithms With Reduced Switching

The queue-based algorithms generalize Algorithm A by relaxing the requirement that the system interval be periodic. That is, the notion of system quantum introduced in the context of Algorithm A is replaced a notion of system interval which is not fixed but rather dynamically defined in terms of the task deadlines. For example, for the Fair Algorithm, any two successive task deadlines creates a system interval.

Like Algorithm A, queue-based algorithms impose structure by partitioning the tasks into queues. Having allocated tasks to queues and queues to processors, a deadline-based schedule (not a cyclic schedule) is generated for each queue. This schedule operates on two levels:

- a global schedule which allocates the processors to each queue,
- a local, deadline-based schedule internal to each queue.

Possible advantages of the queue-based approach are

- The tasks run in a highly localized setting, so that the databases they require can be localized. Most tasks run on a single processor, and even those that are shared are shared by only two processors.

• The scheduling algorithms are efficient and generate schedules without excessive computational overhead, even for $M > 2$.

• The rate of processor switching is reasonable, i.e., less than for Algorithm A.

5.2 The Fair Algorithm

The basic queue-based algorithm is the Fair Algorithm. All queues are partitioned in the same way as the local queues are defined in Algorithm A. The basic unit for scheduling is the system interval between two successive deadlines $[\, t_{i-1}, t_i)$. This is convenient because the sequence in which tasks are scheduled within a system interval is not important.

Let

$$\mu_r = \sum_{A \in Q_r} \frac{e_A}{p_A}, \qquad 1 \leq r \leq 2M - 1$$

be the load of the queue Q_r .

During an interval $[\, t_{i-1}, t_i\,)$, the amount of time allocated to Q_r, is

$$\mu_r(t_i - t_{i-1})$$

As in Algorithm A, during each interval, processor π_k services its queues in a cyclic fashion in the prescribed order:

- $SHARED^{k-1}(Q_{2k-2})$ first,
- $LOCAL^k(Q_{2k})$ next,
- $SHARED^{k+1}(Q_{2K})$ last.

It is easy to see that, if the tasks in each queue are run in deadline order, every deadline will be met. It will again be helpful to use the quantum

$$q = GCD\{e_A, p_A : A \in S\})$$

in generating task schedules internal to each queue. For each queue Q_r, $1 \leq r \leq 2M - 1$, we also define

w_{ir} : the number of quanta allocated to queue Q_r in the interval $[t_{i-1}, t_i)$, and

W_{ir} : the total **number of quanta** given the queue Q_r in the interval $[\ 0, t_i\ )$, so that

$$W_{ir} = \sum_{k=0}^{i} w_{kr}.$$

The most important thing about the Fair Algorithm is that it gives a *fair share* of processing service to the queues. That is, in each interval in

which computing power is allocated to the queues, the amount actually given will be approximately equal to the ratio μ for the queue. In other words, we allocate an amount of processing to the queue between the (integer) *floor* and *ceiling* of μ. However, we require that when each task meets its deadline it must have received sufficient computation time for completion.

The Fair Algorithm:

For i, r such that $1 \leq r \leq 2M-1, \qquad 0 < t_i \leq T,$

a. During the interval $[\ t_{i-1}, t_i)$, queue Q_r is allocated w_{ir} quanta, computed as follows:

$$W_{0r} = 0$$

$$\lfloor (\mu_r \frac{t_i}{q}) \rfloor \leq W_{ir} \leq \lceil (\mu_r \frac{t_i}{q}) \rceil$$

$$\sum_{r=1}^{2M-1} W_{ir} = \lceil (t_i \frac{\mu}{q}) \rceil$$

$$w_{ir} = W_{ir} - W_{i-1r}$$

b. During this interval, for $1 \leq k \leq M$, processor π_k services its queues in the order:

- $SHARED^{k-1}(Q_{2k-2})$first,
- $LOCAL^k(Q_{2k-1})$ next,
- $SHARED^k$ last.

c. The internal scheduling of a queue is by deadline.

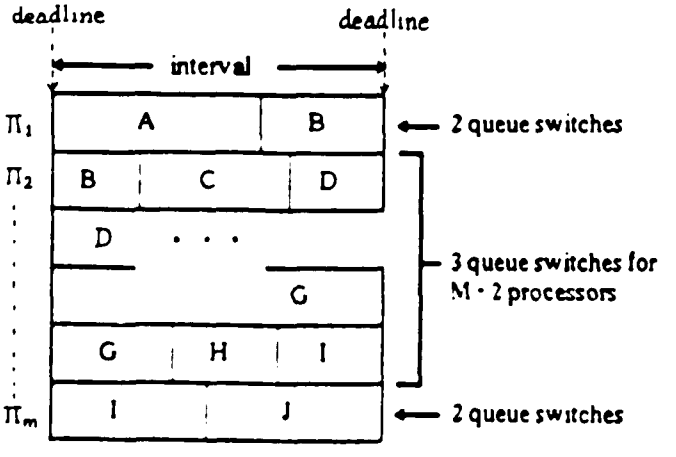


Figure 5.2.2 Fair Algorithm

Fact 5.2.1 : The Fair Algorithm is optimal.

Proof :

1. During each interval, each queue receives the integral part of its fair share of service. Since this is true at each deadline, and queues are run in deadline order, every deadline is met.

2. Restriction $R1$ is satisfied because

- each shared queue $SHARED^k(Q_{2k})$ is run

  early on processor π_{k+1}

  late on processor π_k

- there is no conflict involving Q_{2k} because the load of the queue $SHARED^k(Q_{2k})$ is

  $\mu_{2k} \leq 1$

Q.E.D.

5.3 Switching Rate of the Fair Algorithm

In this section we derive the switching properties of the Fair Algorithm, in particular an upper bound on the number of switches made in the interval [0,T).

Fact 5.3.1 The number of intervals between distinct consecutive deadlines in [0,T) is bounded from above by $L_0 - (N - 1)$.

Proof: There are L_0 task arrivals, but all N tasks have a coinciding deadline at T.

Q.E.D.

Fact 5.3.2 The quantity $2L_0 - N - (2M - 1)$ is an upper bound on the total number of local switches in the interval [0,T).

Proof : Since the local scheduling of each queue is done independently, we have $2M - 1$ local systems (queues) each running a deadline algorithm. The quantity L_0 gives the total number of task arrivals in the interval [0,T), and so in this interval there are $2L_0$ task arrival/completion events. These $2L_0$ events are partitioned among the queues. However, certain events cause no (local) processor switching:

- All initial arrivals, except one in each queue : $N - (2M - 1)$
- The last arrival in each queue : $(2M - 1)$
- The last completion in each queue : $(2M - 1)$

Q.E.D.

Fact 5.3.3 In an algorithm which

- allocates service time to every queue in every interval, and
- uses a Deadline Algorithm within each queue,

each local switch in a queue either

- coincides with a queue switch, or
- corresponds to a completion event,

so that the cost of local switching is $L_0 - (2M - 1)$

Proof : A switch due to an arrival can only occur when the queue is first executed upon the arrival of the task. Also, in each queue, there can be at most one switch due to an arrival at each deadline. If each interval contains a queue switch for each queue, all of the arrival switches will coincide with queue switches.

$Q.E.D.$

Since it is reasonable to assume that in an interesting system

$$M < N \ll L_0$$

we shall find it convenient to simplify (and loosen) these bounds as follows:

- The number of intervals in $[0,T) < L_0$.
- The total number of local switches in $[0,T) < 2L_0$.

For the Fair Algorithm, during each interval,

- processors 1 and M service two queues each, while
- the remaining $M-2$ processors serve three queues each

for a total of $2M-2$ queue switches *inside* each interval. In addition, there are M queue switches at the *seams* between the intervals. The number of queue switches is therefore $3M-2$ per interval. The upper bound on total switching is then

$(3M-2)L_0$	queue switches
$+ \quad L_0$	local switches
$(3M-1)L_0$	total switches

5.4 Modifications of the Fair Algorithm: The Flip-Flop Fair Algorithm

The Fair Algorithm is the basic queue-based algorithm, and Algorithm A is a version of the Fair Algorithm in which the shared queues are

restricted to having a single task. Other variations on the Fair Algorithm are possible and may serve to reduce the amount of switching required. The first modification we describe involves reversing the order of task servicing in every other service interval. This will eliminate virtually half of the queue switching, as shown in the following theorem. We call the algorithm that incorporates this modification the *Flip-Flop Fair Algorithm* .

Fact 5.4.1 If step (b) of the Fair Algorithm is modified to:

(b') During the i^{th} interval, for i *odd* , processor $\pi_k, (1 \leq k \leq M)$ services its queues in the following order:

- $SHARED^{k-1}(Q_{2k-2})$ first,
- $LOCAL^k(Q_{2k-1})$ next,
- $SHARED^k(Q_{2k})$ last.

while for i *even*, the order is reversed to

- $SHARED^k(Q_{2k})$ first,
- $LOCAL^k(Q_{2k-1})$ next,
- $SHARED^{k-1}(Q_{2k-2})$ last.

Then the total number of switches in [0,T) is bounded by $2ML_0$.

Proof : The intervals will now concatenate seamlessly. There will be a total of $2M-2$ internal queue switches, and at most $2L_0$ local switches, for a total switches in $(0,T)$ is $(2M-2)L_0 + 2L_0 = 2ML_0$.

Q.E.D.

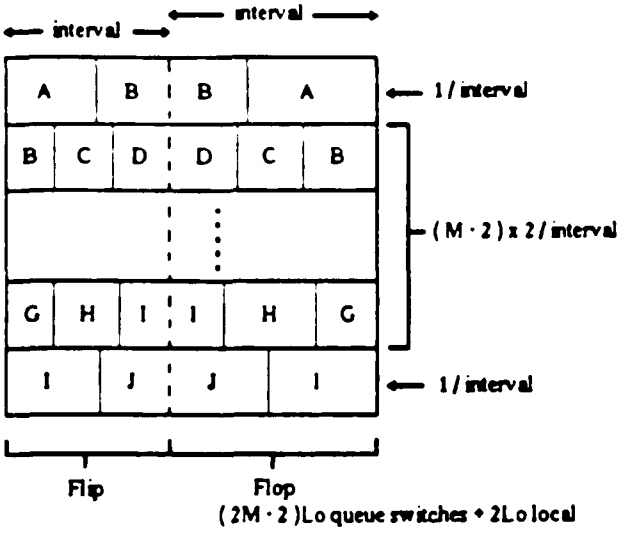


Figure 5.4.1 The Flip-Flop Fair Algorithm

Chapter 6

The Balanced Algorithm

With the Fair Algorithm and its Flip-Flop variant we have made significant progress in reducing the rate of switching to its minimum, mainly by reducing the amount of queue-switching. As we have seen, switching performance can generally be improved by either

- increasing the length of the scheduling interval, or
- rearranging the topology of the queues.

Additional improvements of this type are implemented in the Balanced Algorithm. The Balanced Algorithm requires that the load of the queues be more or less even. Where it does yield a viable schedule, that schedule has extremely good switching properties.

In the Balanced Algorithm we eliminate the distinction between shared and dedicated queues, as well as the topology of the Fair Algorithm.

6.1 Uniform Queue Structure

For the Balanced Algorithm, we begin by defining queues in terms of partial cumulative sums of the task processing loads μ_A, $A \in S$, in the same manner in which every individual queues are defined for Algorithm A. However, we now impose in additional condition that for any two

queues the sum of their required processor loadings must be greater than 1. For any pair of queues Q_r and Q_s, with $1 \leq r \leq 2M-1$, $1 \leq s \leq 2M-1$, we have that

$$(\mu_r > 0 \quad and \quad \mu_s > 0) \Rightarrow \mu_r + \mu_s > 1 \tag{6.1.1}$$

For the Balanced Algorithm we require that the number of queues satisfying (6.1.1) in fact exactly equals $2M-1$. Note that,

- for small M this is easy to achieve (it is inevitable for M equal to 1 or 2.)

- for large M, it is easy to achieve if the μ of the individual tasks is small.

6.2 The Scheduling Interval

Having generated $2M-1$ queues, the Balanced Algorithm now assigns queues to processors and sets the processor schedules. Associated with each queue is a *queue deadline*, the first (i.e., next immediate) deadline for a task in the queue. For the Balanced Algorithm, the scheduling interval consists of the first (i.e., most immediate) M (not necessarily distinct) queue deadlines. Thus, compared with previous queue-based algorithms described in earlier chapters, the Balanced Algorithm extends the length of a scheduling interval by taking M deadlines at a time for a scheduling interval.

6.3 Topology

During each scheduling interval, the Balanced Algorithm splits the $2M-1$ queues into two groups: the *foreground* queues and the *background* queues. The $M-1$ foreground queues include all queues having deadlines inside the scheduling interval. The M remaining queues are in the background.

Assume that for the k_{th} scheduling interval, queues $Q_1 \ldots Q_{M-1}$ are in the foreground, and $Q_M \ldots Q_{2M-1}$ are in the background. In each scheduling interval, processors π_1 thru π_{M-1} run foreground queues Q_1 thru Q_{M-1}, respectively, with high priority and background queues Q_M thru Q_{2M-2}, respectively, with low priority. Processor π_M runs background queues Q_M thru Q_{2M-2} in sequence except that it has low priority in any conflict: if some processor π_1 thru π_{M-1} is already running one of these background queues, then π_M has low priority to run that particular queue. Processor π_M will then run the next background queue for which no conflict exists with any processor π_1 to π_{M-1}, and return to run any background queue skipped due to such a conflict at the earliest possible time. Finally, processor π_M runs its background queue Q_{2M-1} whenever it would otherwise be idle (i.e., with lowest priority of all of the queues it processes). Thus, for processors π_1 to π_{M-1} the background queues are shared with processor π_M while the foreground queues run in a dedicated fashion.

Within each scheduling interval, queues and tasks receive their fair

share of processing. That is, letting S_k denote the duration of the k-th scheduling interval and μ_j the load of queue Q_j, then during scheduling interval k queue j receives $S_k \times \mu_j$ units of processing time.

At the end of a scheduling interval, the next scheduling interval is determined, again in terms of the of the first (i.e., most immediate) M (not necessarily distinct) queue deadlines. The foreground and background queues are redefined in terms of these deadlines, i.e., as $M - 1$ queues having deadlines inside the scheduling interval, and the queues/tasks receive their fair share of processing time as appropriate for the duration of this new scheduling interval.

The topology and scheduling sequence of the Balanced Algorithm is illustrated in Figure 6.3.1.

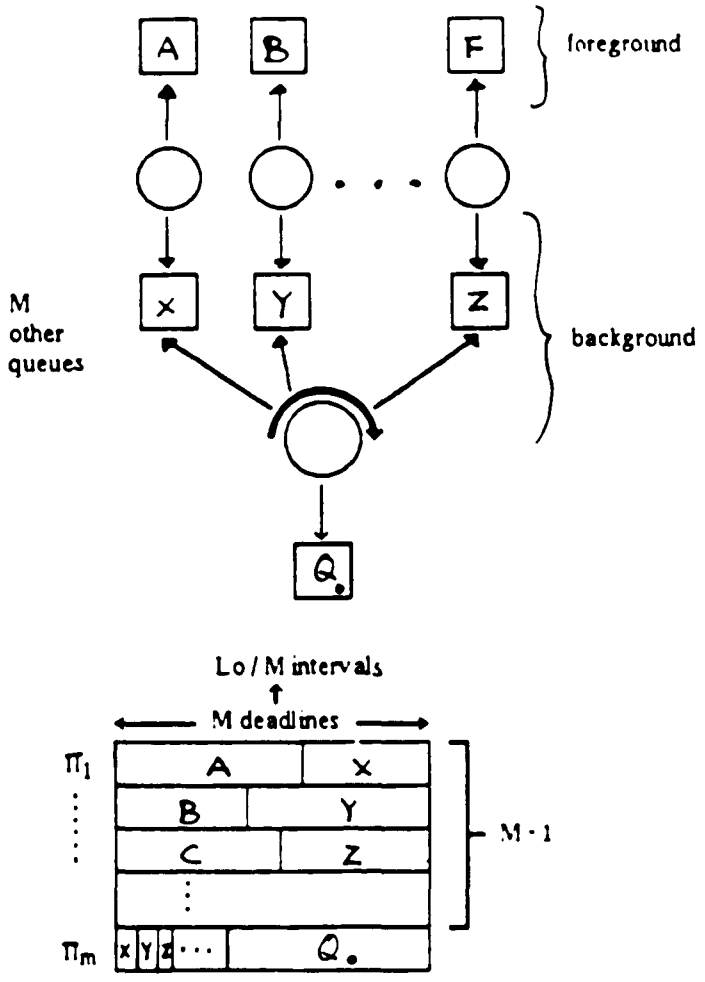


Figure 6.3.1: Balanced Algorithm

6.4 The Balanced Algorithm

Step 1 The scheduling interval $[\ \tau_{i-1}, \tau_i)$ of the Balanced Algorithm will stretch over M (not necessarily distinct) deadlines.

For each deadline t_u we define the set

$$S_u = \{A : A \in S, \quad \exists k : t_u = kp_A\}$$

of tasks which have a deadline which coincides with t_u. We use this to define the scheduling intervals $[\ \tau_{i-1}, \tau_i\ )$ in [0,T).

First, we set

$$\tau_0 = t_0 = 0$$

and

$$\tau_{i-1} = t_v \quad for \quad \tau_{i-1} < T.$$

We define s as

$$s = min\{r : M \leq \sum_{u=v}^{r} |S_u|\}$$

and finally,

$$\tau_i = min(t_s, T).$$

Thus, for $1 \leq r \leq 2M - 1, \quad i : 0 \leq \tau_i \leq T.$

Step 2 During the interval $[\ \tau_{i-1}, \tau_i\ )$, each queue Q_r is allocated w_{ir} quanta, computed as follows:

$$W_{0r} = 0$$

$$\lfloor (\tau_i \frac{\mu_r}{q}) \rfloor \leq W_{ir} \leq \lceil (\tau_i \frac{\mu_r}{q}) \rceil$$

$$\sum_{r=1}^{2M-1} W_{ir} = \lceil (\tau_i \frac{\mu}{q}) \rceil$$

$$w_{ir} = W_{ir} - W_{i-1r}$$

Step 3 Processor allocation during the interval is done by the Tiling Algorithm described below.

Step 4 The internal scheduling of each queue is by deadline.

The Tiling Algorithm

The Tiling Algorithm does the actual scheduling of the tasks during the interval $[\tau_{i-1}, \tau_i\ )$. This is done by trying an initial configuration which meets all the deadlines, but may include scheduling conflicts, i.e., a pair of processors may be assigned the same task at the same time.

Any such conflicts are then eliminated in a sequence of transformations until a valid schedule is obtained.

1. Preliminaries We partition the tasks into two sets U_i and V_i, where

$$|U_i| = M - 1, \quad |V_i| = M$$

where the *urgent* set U_i includes all queues with deadlines **inside** the interval $[\tau_{i-1}, \tau_i)$. For $1 \leq r \leq 2M - 1$,

$$\exists A, k : (T_A \in Q_r, \tau_{i-1} < kp_i \leq \tau_i) \quad \Rightarrow \quad (Q_r \in U_i)$$

Let $Q_0 \in V_i$ be some queue such that

$$\forall r : Q_r \in V_i \quad \Rightarrow \quad \mu_r \leq \mu_0 \qquad (6.4.1)$$

2. Initial Configuration

Initially,

• Each of the first $M - 1$ processors is assigned two queues Q_r and Q_s where

$$Q_r \in U_i, \quad Q_s \in V_i - \{Q_0\}$$

The processor allocates w_{ir} units to Q_r, and the remaining

$$q_i - w_{ir}$$

quanta to Q_s. It gives high priority to the urgent queue Q_r, running Q_s in the *background* until Q_r receives its full allocation for the interval.

• Processor M has to supply the balance of the service due to the queues Q_s in $U_i - \{Q_0\}$. For each such queue Q_r, if (Q_r, Q_s) is the pair assigned to the same processor π_m, then

$$w_{is'} = q_i - (w_{ir} - w_{is})$$

is the contribution of processor M to Q_s. We construct a *schedule stub* Ω_0, a string which includes a substring δ'_s of length $w_{is'}$ for each $Q_s \in V_i - Q_0$.

$$\Omega_0 = CONCAT(s : Q_s \in V_i - \{Q_0\} : \hat{\sigma}_s)$$

The initial schedule for the processor π_M is $\Omega_0 Q_0$.

Fact 6.4.1 No deadlines are missed in the i^{th} interval.

Proof :

1. At the end of each interval, each queue receives its fair share of service time. Thus, at the start of the i^{th} interval, each queue has received its fair share.

2. Similarly, a queue $Q_s \in V_i$ has no internal deadlines in the i^{th} interval. By the end of the interval, it will have once again received its fair share of service.

3. Queues $Q_r \in U_i$ may have internal deadlines in the i^{th} interval. Queues in U_i are serviced with high priority during the i^{th} interval. Therefore Q_r will receive at least its fair share at each internal deadline and cannot cause failure.

Q.E.D.

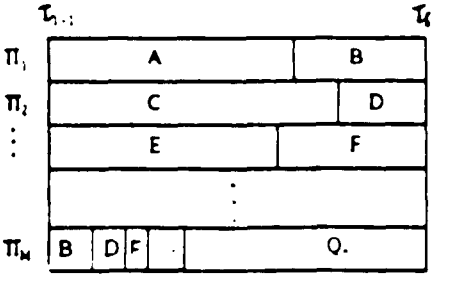


Figure 6.4.1 Initial Configuration

Fact 6.4.2 If each queue $Q_r \in U_i$ can be serviced without interruption, the initial allocation is a valid schedule and the tiling is complete.

Proof : The first $M-1$ processors each schedule a pair of disjoint queues, so that the only possible conflicts would involve the stub Ω_0. But the relations

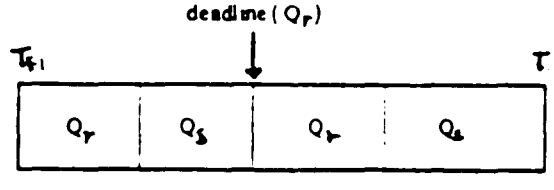
$$\mu_0 + \mu_r \geq 1, \quad Q_r \in U_i \tag{6.4.2}$$

preclude this possibility.

Q.E.D.

It is possible that some $Q_r \in U_i$ cannot receive its whole allotment for the interval without interruption. In this case, the associated processor π_j must interleave Q_r and Q_s, invalidating the assumptions of Fact 6.4.2 above and perhaps violating restriction $R1$ by having Q_s running simultaneously on π_j and π_M.

Fact 6.4.3 Any extra switching caused by this interleaving is already accounted for as part of the local switching for Q_r.



Fact 6.4.4 Each such interleaving is associated with an internal deadline of the interval which involves Q_r. There can therefore be at most $M-1$ such interruptions in the interval (each involving U_i).

The following transformation, applied repeatedly, will result in a feasible schedule for the interval. Let τ be the first instant where there is a clash in the tentative schedule for the interval.

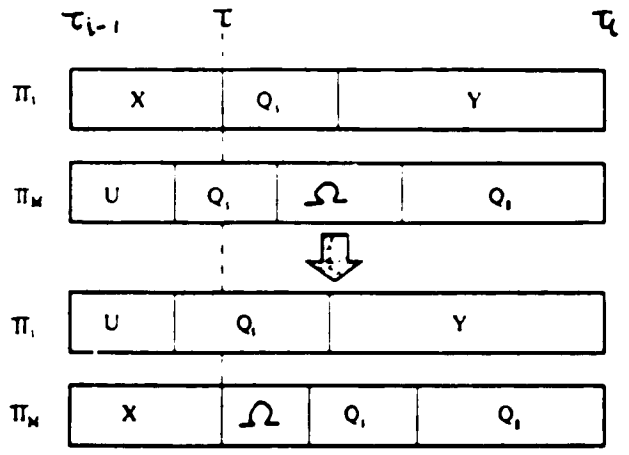


Figure 6.4.3 Transformation (a)

a. Assuming that Ω, the tail of the stub excluding Q_s is not empty, the transformation (a) above will eliminate any conflict at time τ.

b. In case the remainder of the stub is exhausted, the transformation below will eliminate the conflict at time τ.

Q_0 can *cover* any remaining conflicts with Q_s. Figure 6.4.3 shows the general case, where the schedule for π_j includes more occurrences of Q_s,

starting at τ'. In any case, since Q_0 was selected to have

$$\forall s : Q_s \in V_i \Rightarrow \mu_s \leq \mu_0 \qquad (6.4.3)$$

all $Q_s - conflicts$ between π_j and π_M can be eliminated.

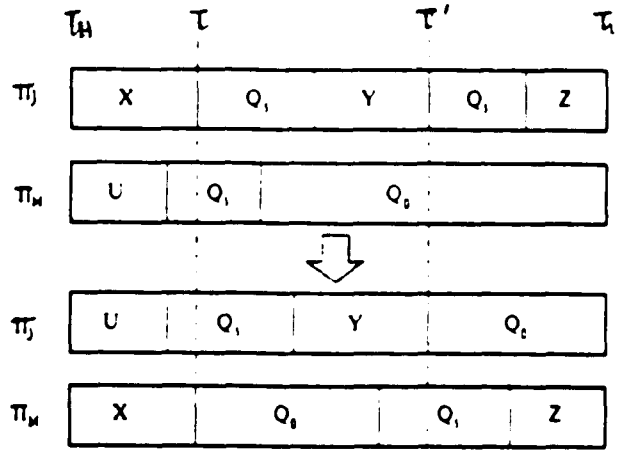


Figure 6.4.4 Transformation (b)

Fact 6.4.5 The transformation step

1. Maintains Fact 6.4.1 invariant.

2. Leaves the number of processor switches in the interval invariant.

3. Increases the length $[\tau_{i-1}, \tau)$ of the conflict-free section of the tentative schedule.

Proof : By construction. In particular a queue $Q_r \in U_i$ retains the time slices it was assigned.

6.5 Amount of Swithching in the Balanced Algorithm

Fact 6.5.1 The Balanced Algorithm is optimal. The number of switches in the interval [0,T) is bounded by

$$(5 - \frac{3}{M})L_0.$$

Proof: The optimality of the Balanced Algorithm follows from Facts 6.4.1 and 6.4.5. Since Fact 6.4.5 states that the transformation step does not change the amount of processor switching, we use the initial configuration to compute the bound.

We see from Figure 6.4.1 that in the initial configuration

a. Each interval includes $2(M-1)$ internal queue switches,

b. Since we have no choice in selecting the set Ui, it is possible that $M-1$ queue switches occur at the join of two intervals. (Since there are $2M-1$ queues, any two subsets of size M will always have at least one common element.)

c. The total number of queue switches in each of the $\frac{L_0}{M}$ intervals is $3(M-1)$.

d. As before, we have an additional $2L_0$ local switches.

e. The total is therefore

$$(5 - \frac{3}{M})L_0.$$

Q.E.D.

6.6 Disscussing of Switching Rate

As M is increased, the bound approaches $5L_0$. A measure of interest is the ratio

$$\frac{L_0}{M}$$

which refers to the activity of a single processor. This ratio is the average number of arrivals per processor. Our main result states that the switching bound is less than 5 times the number of arrivals per processor. This compares well with the deadline algorithm for a single processor, where the bound is twice the number of arrivals. The switching cost of multiple processors is therefore to increase the bound on the amount of processor switching by a constant factor less than 2.5.

6.7 Dedicating Processors to Queues

One of the attractive features of the earlier queue-based algorithms (the Fair Algorithm, and its variants) is the static nature of the queue-processor assignment logic, in that queues are assigned to one or two

processors and are serviced by those processor(s) henceforth. With the Balanced Algorithm and the dynamic nature of the scheduling interval and queue-processor assignments we lose this unique association of queues to processors. As a result, we cannot simply "hard-wire" queues to processors and associated peripherals, e.g., local memory, output busses, etc.

Nevertheless, we can still associate one queue to a particular processor. Say we wish to uniquely assign queue "X" to the "*red*" processor. Then, in a given scheduling interval, if queue X is in the foreground, it is simply assigned to run on the red processor. If it is not in the foreground, it becomes the "*special*" background queue $2M - 1$ and the red processor becomes the "*special*" processor π_M to which it is assigned.

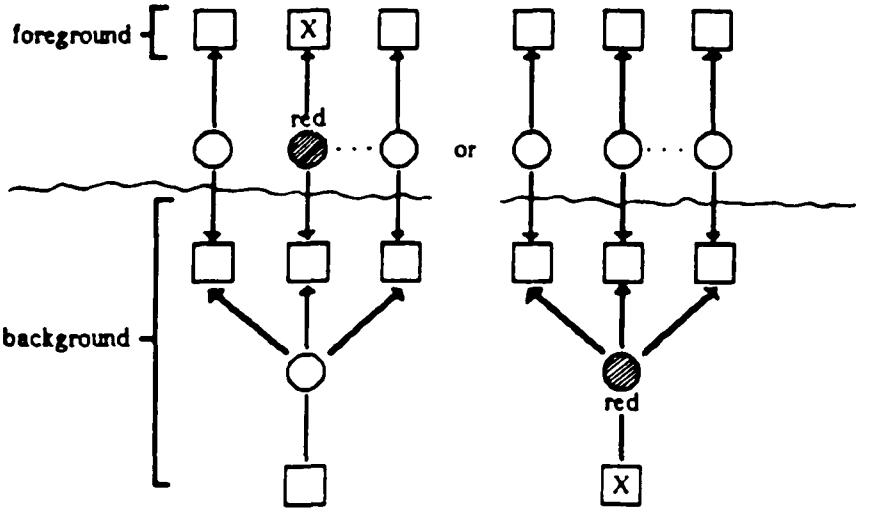


Figure 6.7.1 Dedicating Processors to Queues

6.8 Generalizing the Balanced Algorithm

The discussion of the unique role played by the *"red"* processor π_M in allowing a specific queue to be assigned to a specific processor suggests a generalization of the Balanced Algorithm. In essence, the Balanced Algorithm splits the processors into $M-1$ *"blue"* processors and one *"red"* processor. In fact, we may split the processors up arbitrarily. We illustrate the required modifications by describing below the Balanced Algorithm with two *"red"* processors and $M-2$ *"blue"* processors. The changes required for r *"red"* processors and $M-r$ *"blue"* processors will be obvious. The payoff is that with r *"red"* processors we can uniquely assign r queues to specified processors.

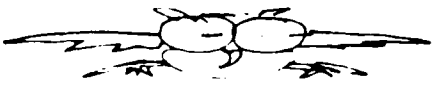
Modified Scheduling Interval: The scheduling interval consists of the first (i.e., most immediate) $M-1$ (not necessarily distinct) queue deadlines.

Modified Topology: The total $2M-1$ queues are split into $M-2$ foreground queues having deadlines and $M+1$ background remaining queues. In each scheduling interval, processors π_1 thru π_{M-2} run foreground queues Q_1 thru Q_{M-2}, respectively, with high priority and background queues Q_{M-1} thru Q_{2M-4}, respectively, with low priority. Processors π_{M-1} and π_M run queues Q_{2M-3} and Q_{2M-2}, respectively, and background queues Q_{M-1} thru Q_{2M-4} as a group, avoiding conflicts with processors π_1 thru π_{M-2}.

Thus, processors π_{M-1} and π_M are "*red*" processors, to which queues

Q_{2M-3} and Q_{2M-2}, respectively, are uniquely assigned.

Part III



Flat Algorithms

Chapter 7

The Flat Algorithms

It is well known that the Deadline Algorithm, which is optimal with one processor, is not optimal with multiple processors. This is illustrated by the following example.

Example 7.1: A two processor system with tasks $\{A, B, C\}$, where $A = B = (2, 4)$ and $C = (7, 8)$ has a valid schedule:

$$\begin{array}{l} AABBAABB \\ CCCCCCC- \end{array}$$

However, a schedule generated by the Deadline Algorithm would start with the prefix

$$\begin{array}{l} AA.... \\ BB.... \end{array}$$

which cannot be completed for a valid schedule.

The inadequacy of the Deadline Algorithm in Example 1 is a result of not giving sufficient early attention to task C, the task with the least immediate deadline. If we wish to use a deadline-based scheduling algorithm in the multiprocesor setting, we might consider augmenting

the deadline algorithm with a process for introducing additional effective deadlines designed to prevent tasks with distant deadlines from being ignored until it is too late to run them successfully. Thus, if task C has to receive e units of service by its deadline t, and if t_1 is a time which satisfies

$$t - e < t_1 < t$$

then C has an effective deadline t_1 before which it has to receive

$$e - (t - t_1)$$

units of service. In Example 1, C needs 7 units of service by t=8. This can be refined to:

$$C : 3 \text{ units by } t_1 = 4 \text{ ; } 4 \text{ additional units by } t = 8$$

While this type of refinement will succeed in Example 1 (basically, an added deadline is required for deadline-based scheduling here), we shall demonstrate below that it is not enough in general.

The following is a more interesting example of how the Deadline Algorithm can fail in the multiple processor setting:

Example 7.2: A two-processor system with tasks

$$S = \{A, B, C, D\}, \quad A = (1,2), \quad B = (1,3), \quad C = (4,6), \quad D = (5,10)$$

This system has period $T = 30$. Consider the initial part of the schedule obtained by applying the Deadline Algorithm to this system:

01	2	3	45
AC	A	B	AD
BD	C	C	$C-$

At time t=5, the only task on hand to be run is D, and so (since a task may only run on one processor at a time, restriction **R1**) one of the processors will be idle during the interval [5,6). However, a valid schedule for this system cannot have any idle cycles since

$$\mu = 2$$

and so the deadline algorithm again fails. In contrast to Example 1, at time t=6 there is still sufficient time left to run each individual job; rather, there is no longer enough time to run all of the jobs.

It is not hard to see that any schedule for Example 2 must satisfy the condition:

D receives at least 3 units of service in the interval [0,6). (7.1)

The Paris Constraints are a formalization of such restrictions. In the two-processor case, the Paris constraints in fact amount to additional deadlines imposed on a task to force compliance with restrictions similar to (7.1) above.

7.1 The Two Processor Case: Paris Constraints

For the important special case of only two processors ($M = 2$), the Paris constraints associated with a deadline k are given by a set of recurrence relations. Let

w_i^- represent the work at hand immediately before the deadline t_i, and

w_i^+ be the work at hand immediately after the deadline t_i.

For $A \in S$, the quantity $t_i \bmod p_j$ represents the time elapsed since the arrival of task A. Then

$$b_{iA} = max(0, \quad e_A - t_i \bmod p_j)$$

is a lower bound on the work remaining to be done on behalf of task A at time i. We define the auxiliary variables

$$a_i = \sum_{A: t_i \bmod p_j = 0} e_A$$

and

$$b_i = \sum_{A \in S} b_{iA}$$

Then

a_i is the total work which arrived at i, and

b_i is a lower bound on w_i^-.

Consider now the i'th interval [i-1, i). It is easy to see that w_i^-, w_i^+ satisfy the following recurrence relations:

$$w_0^- = 0$$

$$w_{i-1}^+ = w_{i-1}^- + a_{i-1}, \quad i > 0$$

$$w_i^- = max(w_{i-1}^+ - M \times (t_i - t_{i-1}), \quad b_i), \quad i > 0$$

The final relation describes the requirement that M processors all be kept busy to the extent that there are a sufficient number of tasks and available work to keep them busy.

Let α_{iA} denote the unfinished work for A at deadline t_i. The constraint C_{ij} for task A at deadline t_i is expressed in terms of α_{iA} :

$$C_{iA}: \begin{cases} \alpha_{iA} = 0, & \text{if } t_i \bmod p_A = 0; \\ \alpha_{iA} \leq w_i^-, & \text{otherwise.} \end{cases}$$

The first part of the constraint simply states that all tasks meet the usual (regular) deadlines. The second part of the constraint essentially defines additional deadlines for each task. These new deadlines require that each task receive as much processing time as possible in each interval. The rationale for this constraint is as follows:

Consider the situation at the end of the interval [i-1, i), just before the arrival of new tasks. Denote this instant by t_i^-. Also, let

$$\beta_i^- = \sum_{j=1}^{N} \alpha_{iA} \qquad (7.1.1)$$

denote the actual work left at t_i^-. We assert

a. $\beta_i^- \geq w_i^-$

b. $\beta_i^- > w_i^- \Rightarrow$ the interval [i-1, i) will have idle cycles.

c. $(\exists A : A \in S : \alpha_{iA} \leq w_i^-) \Rightarrow (\beta_i^- = w_i^-)$ or

$$(\exists B : B \in S, B \not\equiv A : \alpha_{iB} > 0)$$

d. $(\exists A, B \in S : A \not\equiv B, \alpha_{iA} > 0, \alpha_{iB} > 0) \Rightarrow$ no idle cycles in [i-1, i)

From which we conclude that

$$(\forall A : A \in S : \alpha_{iA} \leq w_i^-) \Rightarrow (\beta_i^- = w_i^-)$$

The key point is (d): if there are two tasks left at the end of the interval $[\, t_{i-1}, t_i \,)$, then there cannot be any idle cycles inside the interval, since the unassigned task (which has, of course, a deadline) replaces the idle cycle without clashing with the task that is scheduled on the active processor.

Let us now return to our two examples to see how the new constraints enable the generation of viable schedules.

Example 7.1 Revisited: A two processor system with tasks $\{A, B, C\}$, where $A = B = (2, 4)$ and $C = (7, 8)$. Recall that a schedule generated by the standard deadline algorithm would start with the prefix

$$AACC....$$
$$BB--....$$

resulting in failure to complete task C. The additional constraints defined above prevent this failure. Note first that at time $t = 0$ a total of 11 units of work arrived. Since the system has two processors ($M = 2$), the constraint requires that by time $t = 4$ only 3 units of this work

remains. That is, the constraint requires that no task, in particular task C, have more than 3 units of work remaining at time $t = 4$. Thus, under the constraint task C has a deadline at $t = 4$ to have received 4 units of processing. Now, all three tasks have deadlines at $t = 4$. A deadline algorithm will successfully schedule this augmented system.

$$\begin{array}{c} CCCCCCC- \\ ABABABAB \end{array}$$

Example 7.2 Revisited: A two processor system has tasks

$$S = \{A, B, C, D\}, \quad A = (1, 2), \quad B = (1, 3), \quad C = (4, 6), \quad D = (5, 10).$$

Recall that the initial part of the schedule obtained by applying the standard deadline algorithm to this system is:

01	2	3	45
AC	A	B	AD
BD	C	C	$C-$

The occurrence of an idle cycle in a two processor system for which $\mu = 2$ means the schedule must eventually fail. The additional constraints defined above prevent this failure as well. Note first that up to time $t = 6$ a total of 14 units of work has arrived (three arrivals of task A, two arrivals of task B, and one arrival each of tasks C and D). Since the system has two processors ($M = 2$), the constraint requires that by time $t = 6$ only 2 units of this work remains. That is, the constraint

requires that no task, in particular task D, have more than 2 units of work remaining at time $t = 6$. Thus, under the constraint task D has a deadline at $t = 6$ to have received 3 units of processing. Adding this constraint yields the following deadline-algorithm schedule:

01	2	3	45
AC	A	C	CD
BD	C	D	AB

It may seem somewhat mysterious as to why we can reduce the deadline constraint

C_1: The total work remaining $\leq w^-$.

to the seemingly weaker constraint

C_2: Work remaining for each task $\leq w^-$.

and so obtain a useful augmented system of deadlines. While it is clear that $C_1 \Rightarrow C_2$, in the context of deadline-based scheduling the two constraints are in fact equivalent. This is shown in the following proof.

Fact 7.1.1 : In a two processor system scheduled by a standard deadline algorithm, C_1 and C_2 are equivalent.

Proof :

a. It is clear that $C_1 \Rightarrow C_2$.

b. To see that $C_2 \Rightarrow C_1$, suppose that C_1 is false (and so the total amount of work remaining exceeds w^-) and C_2 is true (and so the amount of work remaining for each task is less than or equal to w^-). Then

1. There is an idle cycle in the scheduling period, and

2. More than one task remains unfinished at the deadline.

This is a contradiction, because under a deadline algorithm, the one of the unfinished tasks would be used to fill in the idle cycle. Q.E.D.

7.2 The General Multiple Processor Case

With more than two processors ($M > 2$), the reasoning is basically the same, except that the constraints

$$\textit{unfinished work at time } t_i \leq w_i^-$$

must hold for every subset of size $(M - 1)$ of the set:

$$S_i = \{A : A \in S, \quad t_i \text{mod} p_A > 0\}$$

This is because in the multiple processor setting, the equivalent of Assertion (d) would require at least M unfinished tasks at the end of the period to guarantee no idle cycles in the period: the idle cycle on one

processor occurs concurrently with up to $M-1$ tasks scheduled on the remaining processors. To avoid a potential clash, we need a pool of at least M tasks to choose from. This suggests that the generalization of the Paris Constraints to $(M > 2)$ processors would lead to a linear program. A bound on the number of constraints required in this setting is

$$\binom{N}{M-1}$$

for each interval. The two processor case is therefore a serendipidous simplification, and owes its efficiency to the fact that $M-1=1$ so that the constraints reduce to N restrictions on the N individual tasks. As we have seen, these can be incorporated into the deadline table without causing any expansion of the table.

The Constraints

The Paris constraints have the effect of postponing the idle cycles in the schedule. We now explore this notion further and prove that application of the Paris constraints results in a schedule in which idle cycles are postponed to the maximum extent possible.

Consider a feasible system S of tasks. Let T be the system period, and let k be the number of intervals between successive deadlines in [0,T). T and k are determined by S and are independent of any particular schedule.

We confine our attention to viable schedules Σ where the unit of allo-

cation **quantum** q

$$q = GCD(\{e_A, p_A : T_A \in S\})$$

is the **GCD** of the system parameters. Then every schedule $\sigma \in \Sigma$ will have the same number r of idle quanta in the period [0,T) .

For each schedule $\sigma \in \Sigma$, define the $k - tuple$

$$(r_1, r_2, \ldots, r_k)$$

where r_i is the number of idle quanta in the $i'th$ interval between deadlines. Each such $k - tuple$ defines an equivalence class on the set Σ in the following way: we say two schedules in Σ are (idle-cycle) equivalent if they are associated with the same $k - tuple$ $(r_1, r_2, \ldots, r_k)$.

Define a lexicographic order $\prec$ such that, if σ and σ' are schedules characterized by k-tuples $(r_1, r_2, \ldots, r_k)$ and $(r'_1, r'_2, \ldots, r'_k)$, we say that

$$\sigma' \prec \sigma, \quad iff \exists_j : 0 \leq j \leq i \quad : \quad r'_i = r_i, i < j, r'_j < r_j$$

So, for example, $(0,0,0,3) \prec (0,0,1,2)$.

Consider the equivalence class $[\sigma_0]$ such that, for all $\sigma \in \Sigma$, either

$\sigma \in [\sigma_0]$ or else $\sigma_0 \prec \sigma$. Then $[\sigma_0]$ is the set of viable schedules for which the idle cycles appear as late as possible. In the following sequence of theorems, we will show

1. that the set $[\sigma_0]$ is not empty,

2. that if a schedule does not satisfy the Paris constraints then it is not an element of $[\sigma_0]$, and

3. that if a viable schedule satisfies the Paris constraints then it is an element of $[\sigma_0]$.

Fact 7.2.1 : For a feasible system, the equivalence class $[\sigma_0]$ is not empty.

Proof : Since both the number of intervals in [0,T) and the total number r of idle quanta is fixed, there can only be a finite number of equivalence classes. Thus,

$$\exists \sigma \quad \Rightarrow \quad \sigma \in [\sigma_0] \quad or \quad (\exists \sigma' : \sigma' \prec \sigma, \quad (\forall \sigma'' : \sigma' \prec \sigma'' \quad or \quad \sigma'' \in [\sigma']))$$

QED

Fact 7.2.2: Every schedule in $[\sigma_0]$ satisfies the Paris Constraints.

Proof : Assume the contrary: that $\sigma \in [\sigma_0]$ does not satisfy the Paris constraints. Let i be the first deadline where σ violates the Paris Constraints.

1. From these assumptions we have

$$\exists A : A \in S\alpha_{iA} > w_i^- \qquad (7.2.1)$$

$$\forall \sigma' : \quad \sigma' \in [\sigma] \quad \textbf{or} \quad \sigma \prec \sigma' \qquad (7.2.2)$$

2. From (7.2.1) and (7..2.2) we have that

$$\beta_i > w_i^- \qquad (7.2.3)$$

3. From (7.2.3) and Assertion (b) above we conclude that the i'th interval contains idle cycles.

4. Let d_A be the next deadline of T_A :

$$d_A = t_i + p_A - t_i \bmod p_A$$

Since $\alpha_{iA} > 0$,

$$\exists \tau : t_i \leq \tau \leq d_A : T_j \in \sigma[\tau]$$

and there is an occurence of A in $[\ t_i, d_A\ )$.

5. Since i is the first deadline where σ violates the Paris Constraints,

$$\alpha_{i-1,A} \leq \bar{w_{i-1}}$$

and, using (7.2.1) we have

$$\alpha_{iA} > b_{iA}$$

From this, we conclude that the interval $[\ t_{i-1}, t_i)$ contains fewer than

$$t_i - t_{i-1}$$

quanta of A . There must exist a quantum τ' in the interval $[\ t_{i-1}, t_i\ )$ during which A is not scheduled.

$$\exists \tau' : t_{i-1} \leq \tau' \leq t_i : \sigma[\tau'] = (X, Y), \quad X \neq Y, X \neq T_j, Y \neq T_j$$

6. Also, either

a. $X = idle$ or $Y = idle$, or

b. $\exists \tau'' : t_{i-1} \leq \tau'' \leq t_i : \sigma[\tau''] = (W, idle)$

We construct a schedule σ' which differs from σ as follows:

a. If case a holds, and, say $X = idle$:

$$\begin{array}{ll} \sigma[\tau'] = (idle, Y), & \sigma[\tau] = (T_j, Q) \\ \sigma'[\tau'] = (T_j, Y), & \sigma'[\tau] = (idle, Q) \end{array}$$

b. If case b holds, then since $X \neq Y$, then either $X \neq W$ or $Y \neq W$. Say $X \neq W$,

$$\begin{array}{lll} \sigma[\tau''] = (W, idle), & \sigma[\tau'] = (X, Y), & \sigma[\tau] = (T_j, Q) \\ \sigma'[\tau''] = (W, X), & \sigma'[\tau'] = (T_j, Y), & \sigma'[\tau] = (idle, Q) \end{array}$$

In either case, σ' is a valid schedule for the system S . Also, $\sigma' \prec \sigma$ since

$$r'_i < r_i \qquad \forall k : 0 \leq k < i : r'_k = r_k$$

which contradicts (7.2.2).

Fact 7.2.3: If $\sigma \in \Sigma$ satisfies the Paris Constraints, then $\sigma \in [\sigma_0]$.

Proof : Assume the contrary: Suppose σ satisties the Paris Constraints but that there exists $\sigma' \prec \sigma$. Then

$$\exists j : 0 \leq j \leq i | r'_i = r_i, \quad i < j, \quad r'_j < r_j$$

Since σ satisfies the Paris Constraints, by (b), we get

$$\beta_j^- = w_j^- \tag{7.2.4}$$

Since $r'_i = r_i$ for $0 \leq j \leq i$, we have

$$w'_{j-1^+} = w_{j-1^+} \tag{7.2.5}$$

$$w'_{j^-} = w_j^- \tag{7.2.6}$$

and since $r'_j < r_j$, we have

$$\beta_j^- < \beta'_{j^-} \tag{7.2.7}$$

From (7.2.6) and (7.2.7) we conclude that

$$\beta_j^- > w_j^-$$

which contradicts (7.2.4). *QED*

7.3 The Flat Algorithms

The Flat Algorithms we consider here obey the Paris Constraints. The structure of our algorithms is

1. Build a constraint table.

2. Traverse the constraint table in some order to construct a schedule.

The first step amounts to specifying the **least** equivalence class of schedules $[\sigma_0]$ described above. The second step selects **one** member of this class of schedules.

In chapter 8 we show that there is an effective way of constructing a schedule from the constraint table. Note that the original, simple constraint table where the only constraints are the basic deadlines of each task also specifies a set of schedules (in fact, it specifies the set of **all** schedules.) In a sense then, the scheduling problem is simply finding an effective strategy for satisfying these constraints in the course of constructing a schedule.

The KL Algorithm

As shown above, the Paris Constraints have the effect of postponing idle cycles which might otherwise occur in the early part of the schedule. In practice, they confine our search to the *'least'* equivalence class $[\sigma_0]$ of schedules. If, however, the system under consideration satisfies the

relation

$$\mu = M$$

so that a valid schedule cannot contain any idle cycles, all schedules σ will be in the same equivalence class $[\sigma_0]$. In this case, the Paris constraints will be feasibility constraints, in the sense that they do not exclude any valid schedule from consideration. Example 7.2 is an example where every schedule must obey the Paris Constraints.

The question of whether a group of valid schedules is excluded from consideration is an important one, since we are searching for schedules with a certain property (that being a low switching rate.) It is therefore comforting that the Paris Constraints can be modified so as not to rule out any schedule. The idea is to *'augment'* the given system S into a new system S' by adding suitable *'dummy'* tasks so that the load of the augmented system S' is

$$\mu' = M$$

There is then a many-to-one mapping between the set Σ' of schedules of S' and the set Σ of schedules of S : the dummy tasks are simply place holders for idle periods in the schedules for S.

This modification of the Paris Constraints actually simplifies the algorithm. For S, the number of idle processor cycles for the interval [0,T) is:

$$r = M \times T - \sum(\frac{T}{p_j} : \quad 1 \leq j \leq N)$$

Logically, $S' = S \cup R$, where R is a set of r identical tasks whose parameters are $(1, T)$. Each of these dummy tasks stands for a single idle cycle; this circumvents any problems with restriction $R1$.

In practice, we need not represent the dummy tasks in our deadline table at all. Their sole effect is to add r to the initial work at hand. In addition, since the augmented system S' cannot have idle cycles, the recurrence relations are simplified. In the two processor case, they are

$$w_0^- = r$$

$$w_i^+ = w_i^- + a_i, \quad i \geq 0$$

$$w_i^- = w_{i-1}^+ - M \times (t_i - t_{i-1}), \quad i > 0$$

where, as before,

$$a_i = \sum_{A \in S} (e_A \mid j : t_i \bmod p_j = 0)$$

and the constraint for task A at deadline t_i is still

$$C_{iA} : \begin{cases} \alpha_{iA} = 0, & \text{if } t_i \bmod p_A = 0; \\ \alpha_{iA} \leq w_i^-, & \text{otherwise.} \end{cases}$$

Chapter 8

The Optimality of the Paris Algorithm

In this chapter we show that the Deadline Algorithm applied to the set of Paris Constraints

$$A \in S: \quad i : 0 \leq i < T: \quad \alpha_{iA} \leq c_{iA}$$

is optimal, in the sense that it is an effective procedure for constructing a schedule. It is more expedient to work with the KL Algorithm, and no generality is lost. We therefore confine our attention to fully loaded systems ($\mu = 2$). Our approach is to show that any feasible system has a schedule in deadline order (in the Paris sense of deadline.) We proceed as follows:

First we define permutations which transform one schedule into another and establish sufficient conditions for the existence of such permutations. We then define a lexicographic ordering of schedules for a given system. This ordering has the property that, assuming that they exist, deadline schedules for the system would precede all others. We then show that for a feasible system, every schedule which violates the deadline property is preceded by another.

8.1 Definitions and Transformations

We begin by defining some mechanisms for transforming one schedule into another and by stating certain basic facts about such transforma-

tions. Let σ be a schedule for a system S. We define, for $k : 0 \leq k < T$, subschedules σ_{0k} and σ_{kT} such that,

$$\sigma = \sigma_{0k} \quad \sigma_{kT}$$

and,

σ_{0k} is the schedule for the interval [0,k)

σ_{kT} is the schedule for the interval [k,T)

We extend this notation and write σ_k for the "*slice*" $\sigma_{k,k+1}$

We also let the ordered $N - tuple$

$$\Theta_k = (\alpha_{k1}, \ldots, \alpha_{kN})$$

be the state of the system in terms of the unfinished work at k.

Where necessary, we use superscripts $+$, $-$ to avoid ambiguity at the discontinuity. Thus, we write,

$$\Theta_0^+ = (e_1, \ldots, e_N) \quad \text{and} \quad \Theta_T^- = (0, \ldots, 0)$$

A state Θ_k is valid at k if there exists a schedule

$$\sigma = \sigma_{0k} \quad \sigma_{kT}$$

such that the system is in state Θ_k at time k. In this case, σ_{0k} is a valid prefix, and σ_{kT} is a valid suffix, provided they are well-formed (each slice σ_i contains 2 distinct tasks).

Well-Formed σ_{0k} ::

$$\forall i : 0 \leq i < k : \sigma_i = \{X, Y\} \Rightarrow X \not\equiv Y$$

Equivalently, a state Θ_k is valid iff $\mathbf{Q}_k$ holds:

$$\mathbf{Q}_k : \quad \Theta_k = (\alpha_{k1}, \ldots, \alpha_{kN})$$
$$\forall A : A \in S :$$
$$b_{kA} \leq \alpha_{kA} \leq min(\forall i : k < i \leq t_{kA} : c_{iA} - (i - k))$$

The prefix σ_{0k} is valid iff

$$\mathbf{P}_k \wedge \mathbf{Q}_k \wedge \text{Well} - \text{Formed}\sigma_{0k}$$

where

$$\mathbf{P}_k : \quad \forall i : 0 \leq i < k :$$
$$\forall A \in S : b_{iA} \leq \alpha_{iA} \leq c_{iA}$$

We use t_{kA} as shorthand for the next deadline of A following k:

$$t_{kA} = k + p_A - k \bmod p_A$$

and,

$$SLACK_{kB} :: \forall i : k < i \leq t_{kB} : \quad \alpha_{kB} < c_{iB} + (i - t_{kB})$$

Also for $0 \leq k < T$, we use Θ_k to define sets R_k and L_k :

$$L_k = \{A : \alpha_{kA} > b_{kA}\}$$
$$R_k = \{B : SLACK_{kB}\}$$

A task in L_k has received more than the minimum amount of service at k with respect to its Paris Constraints. Similarly, a task in L_k has received less than the maximum amount of service. L_k and R_k constrain the possible transformations allowed on a schedule: a task in R_k can be speeded up (i.e., receive more service in the interval [0, k), whereas a task in L_k can be delayed (i.e., receive fewer quanta in [0, k).)

8.2 Pertinent Facts

Fact 8.2.1

$$a. \quad W \in L_m, W \notin \sigma_m \Rightarrow W \in L_{m+1}$$
$$b. \quad Z \in R_m, Z \in \sigma_m \Rightarrow Z \in R_{m+1}$$

Proof : These follow directly from the definition of R_k and L_k.

Fact 8.2.2

$$\sigma_j = \{Y, Z\}, Y \in L_j, Z \in R_{j+1}$$
$$\Rightarrow \exists W : W \in L_{j+1} - \{Z\}$$

Proof :

$$Z \in R_{j+1} \Rightarrow SLACK_{j+1Z}$$
$$\Rightarrow \alpha_{j+1Z} < w_{j+1}^{-}$$

It follows that during [j, j+1),

$$\exists W : W \not\equiv Z : \alpha_{j+1,W} > 0$$

Now, since W is available during [j, j+1),

$$W \not\equiv Y \Rightarrow b_{j+1,W} < \alpha_{j+1,W}$$
$$\Rightarrow W \in L_{j+1}$$

Also, since $Y \in L_j, Y \in \sigma_j$,

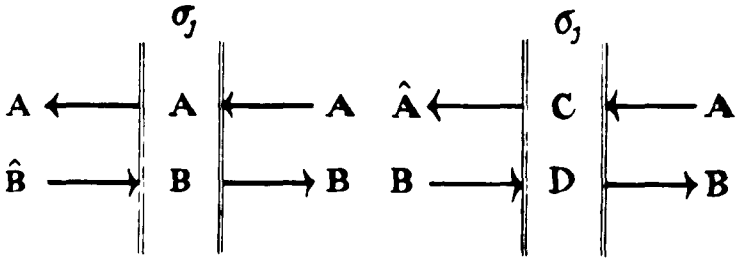
$$W \equiv Y \Rightarrow W \in L_{j+1}$$

QED

8.3 Permutations

We now obtain sufficient conditions for obtaining one schedule as a permutation of another. A permutation consists of a chain of exchanges

between adjacent slices. The proposition (8.3.1) below is a requirement for each link in the permutation.



$$\exists A, B : B \in R_{j+1}, A \in L_{j+1} - \{B\} : (A \in \sigma_j \vee B \notin \sigma_j)$$
$$\Rightarrow \exists \hat{A}, \hat{B} :$$
$$\hat{B} \in R_j,$$
$$\hat{A} \in (\{A\} \cup \sigma_j) \cap L_j - \{\hat{B}\} :$$
$$A \in \sigma_j \Rightarrow \hat{A} \equiv A$$
$$B \notin \sigma_j \Rightarrow \hat{B} \equiv B \qquad (8.3.1)$$

We write

$$\hat{\sigma}_{0k} = (\sigma_{0k} : A \leftrightarrow B)$$

for the permutation obtained by modifying the prefix σ_{0k} by removing one unit of service for task B, and adding one unit of service to A (assuming that such a permutation exists.)

Similarly, to describe a permutation for a suffix, we write

$$\hat{\sigma}_{kT} = (A \leftrightarrow B : \sigma_{kT})$$

We compose permutations to describe the resulting schedule

$$\hat{\sigma}_{0k} = (\sigma_{0k} : A \leftrightarrow B : \sigma_{kT})$$

8.4 Another Lexicographic Ordering of Schedules

Consider a (possibly augmented) fully loaded system and its set of Paris Constraints. If σ is a valid schedule for this system, it must satisfy the Paris Constraints (Fact 7.2.2). We can associate each entry in this schedule with a specific Paris deadline.

Let

$$\sigma_i = \{X, Y\}$$

denote the $i'th$ slice of the schedule, ie., the pair of tasks (including possibly dummies) scheduled during the $i'th$ quantum.

For this slice σ_i we define the ordered pair

$$C_i = (x_i, y_i), \quad x_i \leq y_i$$

where x_i and y_i are the Paris deadlines for the tasks scheduled in σ_i.

We define a *deadline ordering* $\ll$ between two such pairs

$$C = (x, y) \quad \text{and} \quad \hat{C} = (\hat{x}, \hat{y})$$

in the usual way:

$$\begin{aligned} 1.\quad & C \ll \hat{C} \quad \text{iff} \quad x < \hat{x} \\ & \qquad\quad \text{or} \quad x = \hat{x} \quad \text{and} \quad y < \hat{y} \\ 2.\quad & C \approx \hat{C} \quad \text{iff} \quad x = \hat{x} \quad \text{and} \quad y = \hat{y} \end{aligned}$$

We extend this ordering to valid schedules of a given KL system:

If σ and $\hat{\sigma}$ are 2 such schedules, we say

$$\begin{aligned} 1.\quad & \sigma \ll \hat{\sigma} \quad \text{iff} \quad \exists i : i < T : C_i \ll \hat{C}_i \\ & \qquad\qquad\qquad\quad \forall j : 0 \leq j < i, \quad C_j \approx \hat{C}_j \\ 2.\quad & \sigma \approx \hat{\sigma} \quad \text{iff} \quad \forall i : 0 \leq i < T : C_i \approx \hat{C}_i \end{aligned}$$

We are now ready to prove the following theorem.

Fact 8.4.1 Optimality Theorem

Every feasible two processor system has a valid Paris Deadline schedule.

Proof : For a feasible system, the deadline-order schedules, if they exists, must belong to the equivalence class.

$$[\sigma_0] : \forall \sigma : \sigma \notin [\sigma_0] \Rightarrow \sigma_0 \ll \sigma$$

The proof will be by contradiction. We shall contradict the proposition:

There exists a feasible KL system which has no valid deadline schedule.

Assume that such a system exists. The following 2 statements flow from this assumption.

1. There exists a schedule σ such that, every schedule $\hat{\sigma}$ obeys the relation

$$\sigma \ll \hat{\sigma}$$

2. σ is not in (Paris-)deadline order.

It follows from (2) that the schedule σ contains 2 slices σ_i and σ_j which violate the deadline order, so that for some i, j,

for $0 \leq i < j < T$,

$$\sigma_i = \{X, Z\}, \quad \sigma_j = \{Y, W\}, \quad C_i = (x, z), \quad C_j = (y, w),$$
$$X \neq Y, \quad y < z,$$
$$\forall k : \quad i < k \leq j+1 : \quad Z \in R_{j+1}$$
$$\forall k : \quad i < k \leq j : \quad Y \in L_k$$

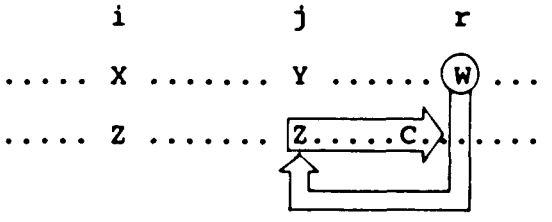
Further, let i be the smallest index for which such an i exists, and j be the largest index which forms the above pattern.

We now proceed to construct a schedule $\hat{\sigma}$ such that $\hat{\sigma} \ll \sigma$

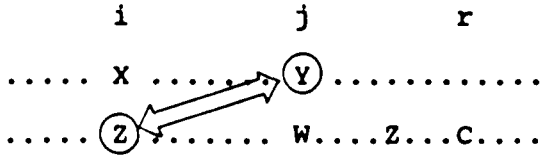
- We must assume that $W \equiv Z$, otherwise the swap $Y \leftrightarrow Z$ will give us $\hat{\sigma}$.

- Similarly, we must assume the relation $x \leq y$ must hold, for otherwise the simple swap $X \leftrightarrow Y$ would give us $\hat{\sigma}$.

To recap, we have $\sigma_i = \{X, Z\}, \quad \sigma_j = \{Y, Z\}$, where $x \leq y < z$.



We shall construct $\hat{\sigma}$ by delaying task Z in σ_j (essentially, shifting the string of $Z's$ at σ_j to the right, and "*bubbling*" some other task W into σ_j.)



so that the swap $Y \leftrightarrow Z$ between σ_i and σ_j can be performed.

```
          i           j           r
..... X ..... Z ............
..... Y ..... W....Z...C....
```

To prove that there exists

$$\hat{\sigma} \ll \sigma$$

we use the Tail Permutation Lemma, whose proof is deferred to the next 2 sections:

$$Z \in R_{j+1} \wedge \exists W : W \in L_{j+1} - \{Z\}$$

$$\Rightarrow \quad \exists \hat{\sigma}_{j+1,T} : \quad \hat{\sigma}_{j+1,T} = [W \leftrightarrow Z : \quad \hat{\sigma}_{j+1,T}]$$

Since $\sigma_j = \{Y, Z\}$, $Y \in L_j$, and $Z \in R_{j+1}$ by assumption, we use Fact 8.2.2 to conclude that there exists W which satisfies the premise.

We now specify $\hat{\sigma}$.

$$\hat{\sigma} = \sigma_{0i}, \quad \hat{\sigma}_i, \quad \sigma_{i+1,j}, \quad \hat{\sigma}_j, \quad \hat{\sigma}_{j+1,t}$$

where

$$\hat{\sigma}_i = \{X, Y\}, \quad \hat{\sigma}_j = \{Z, W\}$$

Since the permutation leaves the prefix σ_{0i} invariant and $\hat{C}_i \ll C_i$, we conclude

$$\hat{\sigma} \ll \sigma$$

so that

$$\sigma \notin [\sigma_0]$$

contradicting the assumption.

8.4 More Preliminaries

In this section we develop 2 preliminary facts to be used in the proof of the Tail Permutation Lemma in the next section.

Fact 8.4.1 Fairness

Every Paris Constraint of the form

$$\alpha_{iA} \leq c_{iA}$$

imposed on some task A at time i obeys the condition

$$\frac{e_A - c_{iA}}{i \bmod p_A} \leq \mu_j$$

Proof : The condition above is a statement that a Paris Constraint cannot require that a task receive more than its *fair* rate of service during any interval [0, i). To prove this, we simply note that

- for a fully loaded system ($\mu = 2$), the Paris Constraints are feasibility conditions, and

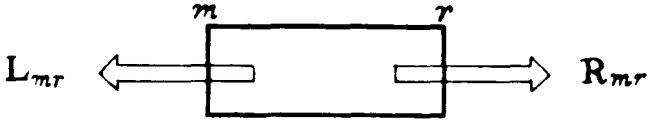
- there exist optimal fair algorithms such as Algorithm A (or any other fine grained processor-sharing algorithm).

QED

Let σ be a schedule, and let σ_{mr} be a segment of this schedule for a proper subinterval $[m, r)$ of the system period $[0, T)$. Also, let

$$S_{mr} = Uk: \quad m \leq k < r: \quad \sigma_{ek}$$

be the set of tasks scheduled during the interval $[m, r)$, and,



$$L_{mr} = L_m \cap S_{mr}$$

$$R_{mr} = S_{mr} \cup R_r$$

Fact 8.4.2

$$L_{mr} \neq \emptyset \vee R_{mr} \neq \emptyset$$

If any of the following holds

1. $\exists Z : Z \notin S_{mr}$

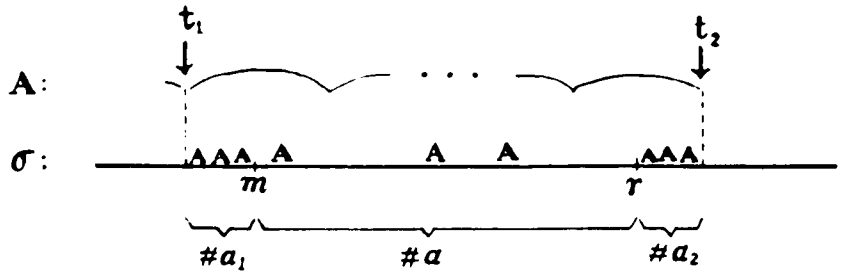
2. $\exists Z : Z \in S_{mr} : \mu_Z < 1$

3. $m > 0$

Proof

The proof is by contradiction. Assume that for all A,

$$A \in S_{mr} \Rightarrow A \notin L_m, \quad A \notin R_r$$



and consider any task A (figure above) where

a. t_1 is the latest arrival of A s.t. $t_1 \leq m$

b. t_2 is the earliest Paris Deadline for A s.t. $t_2 \geq r$

and $[t_1, t_2)$ covers $[m, r)$

Let $\#a_1$, $\#a$ and $\#a_2$ denote the number of occurences of A in $[t_1, m)$, $[m, r)$ and $[r, t_2)$ respectively. Since $A \notin L_m$

$$\#a_1 = min(e_A, m - t_1)$$

so that

$$t_1 < m \Rightarrow \frac{\#a_1}{(m - t_1)} > \mu_A$$

Similarly, since $A \notin R_r$, we use Fact 8.4.1 to write

$$t_2 > r \Rightarrow \frac{\#a_2}{(t_2 - r)} > \mu_A$$

and we conclude

$$\mu_A \geq \frac{\#a}{(r - m)}$$

$$A \in S_{mr} \Rightarrow \frac{\#a}{(r - m)} \leq \mu_A,$$

$$t_1 < m \vee r < t_2 \Rightarrow \mu_A > \frac{\#a}{(r - m)}$$

Finally, summing over all tasks in the interval,

$$\sum_{A \in S_{mr}} \mu_A \geq \sum_{A \in S_{mr}} \frac{\#a}{r - m} = 2 \qquad (8.4.3)$$

Given 8.4.3, and any one of the conditions listed, the assumption is contradicted:

1. $\exists Z : Z \notin S_{mr} \Rightarrow \mu_Z + \sum_{A \in S_{mr}} \mu_A > 2$

2. $\exists Z : Z \in S_{mr} : \mu_Z < 1 \Rightarrow$

$$\exists A : A \in S_{mr} : \mu_A > 1$$

3. $m > 0 \Rightarrow \exists Z : t_1 < m$

$$Z \notin S_{mr} \Rightarrow \mu_Z + \sum_{A \in S_{mr}} \mu_A > 2$$

$$Z \in S_{mr}: \quad \mu_Z < 1 \Rightarrow$$

$$\exists A: \quad A \in S_{mr}: \quad \mu_A > 1$$

$$QED$$

8.5 On the Existence of Permutations

The basic support for the Optimality Theorem flows from the Fact below.

Fact 8.5.1 Tail Permutation Lemma

If σ_{mT} s.t.

$$Z \in R_m, \quad \exists W : W \in L_m - \{Z\}$$

then

$$\exists \hat{\sigma}_{mT} = W \leftrightarrow Z : \sigma_{mT}$$

Proof

The proof will be by induction on the length $T - m = k$.

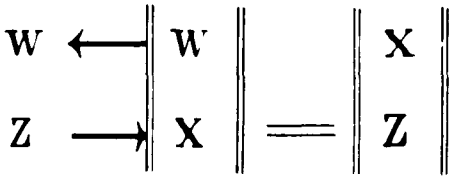
- Base : $k = 1$

Using Fact 8.2.1, we have.

$$\sigma_{mT} = \sigma_m,$$

$$Z \in R_m \Rightarrow Z \notin \sigma_m$$

$$W \in L_m \Rightarrow W \in \sigma_m$$



so that we conclude

$$\hat{\sigma}_{mT} = W \leftrightarrow Z : \sigma_m = \sigma_m \cup \{Z\} - \{W\}$$

- Induction Step :

We assume the statement is valid for all permutations s.t. $T - m < k$. Let σ_{mT} be a suffix where $T - m = k$. There are 2 cases to consider.

Case 1 : $Z \in \sigma_m$:

From Fact 8.2.1*a* :

$$Z \in R_{m+1}$$

From (8.3.1) :

$$\exists \hat{W} \in L_{m+1} \wedge (W \notin \sigma_m \Rightarrow W \equiv \hat{W})$$

$$W \longleftarrow \left\| \begin{matrix} X \\ Y \end{matrix} \right\| \longleftarrow \hat{W} : W \equiv X \vee W \equiv \hat{W}$$
$$Z \longrightarrow$$

$$\begin{aligned}
\hat{\sigma}_{mT} &= W \leftrightarrow Z : \sigma_{mT} \\
&= W \leftrightarrow Z : \sigma_m : \hat{W} \leftrightarrow Z : \sigma_{m+1,T} \\
&= \hat{\sigma}_m, \hat{\sigma}_{m+1,T}
\end{aligned}$$

where we appeal to the induction hypothesis for the existence of

$$\hat{\sigma}_{m+1,T} = \hat{W} \leftrightarrow Z : \sigma_{m+1}$$

and

$$\hat{\sigma}_m = \sigma_m \cup \{\hat{W}\} - \{W\}$$

Case 2 : $Z \notin \sigma_m$:

Let

$$r \geq m, L_{mr} - \{Z\} = \emptyset, W \in L_{mr+1} - \{Z\}, \qquad (8.5.2)$$

$$\begin{aligned}
\mathbf{V} : \forall k : m < k \leq r : \\
&\exists A_k : A_k \in R_{mk} \cup \{Z\}, \\
&\quad A_k \in \sigma_{k-1} \vee A_k \equiv A_{k-1}
\end{aligned}$$

We set

$$A_m = Z$$

and state

$$\neg \mathbf{V} \Rightarrow \exists \hat{\sigma}_{mT} \quad \text{s.t.} \quad Z \in \hat{\sigma}_m \qquad (8.5.3)$$

To prove this, assume the premise.

$$\exists u : m < u \leq r : R_{mu} = \emptyset \quad \wedge \quad Z \notin R_u$$

From Fact 8.4.2, we have,

$$L_{mu} \neq \emptyset$$

and, from (8.5.2),

$$Z \in S_{mu}$$

Let i be the smallest index s.t. $i > m$ and $Z \in \sigma_i$. Since $Z \notin S_{mi}$, and $L_{mi} = \emptyset$, we can write Fact 8.4.2.

$$A_m = Z,$$

$$\forall k : \quad m < k \leq i :$$

$$\exists A_k \in R_{mk}, \quad A_k = \sigma_{k-1} \vee A_k \equiv A_{k-1}$$

and the sequence of transformations

$$\hat{\sigma}_{0m} = \sigma_{0m}$$

$$m < k \leq i : \hat{\sigma}_{0k} = \sigma_{0k} : Z \leftrightarrow A_k$$

is valid.

If $A_i \in \sigma_i$, we also have to transform the tail $\sigma_{i+1,T}$ We have (Fact 8.2.1)

$$A_i \in R_{i+1}.$$

Also, since

$$Z \in \sigma_i \quad \text{and} \quad Z \in L_i,$$

From Fact 8.2.2

$$\exists B_{i+1} \in L_{i+1}$$

Since $i > m$, we again use the induction hypothesis to write

$$\hat{\sigma}_{iT} = \begin{cases} \sigma_i : B_{i+1} \leftrightarrow A_i : \sigma_{i+1,T}, & \text{if } A_i \in \sigma_i \\ \sigma_{iT}, & \text{otherwise} \end{cases}$$

and

$$\hat{\sigma} = \hat{\sigma}_{0i}, \quad (Z \leftrightarrow A_i : \hat{\sigma}_{iT})$$

this proves (8.5.3). Since

$$\neg \mathbf{V} \Rightarrow Z \in \hat{\sigma}_m$$

reduces to Case 1, we assume **V**.

First, if necessary, we transform the tail. As before, we appeal to the induction hypothesis ($r > m$), and write,

$$\hat{\sigma}_{r+1,T} = \begin{cases} B_{r+1} \leftrightarrow A_r : \sigma_{r+1,T} & \text{if } A_r \in \sigma_r \\ \sigma_{r+1,T}, & \text{otherwise} \end{cases}$$

Finally, the entire transformation is

$$\hat{\sigma}_{mT} = \hat{\sigma}_{mr} \quad \hat{\sigma}_r \quad \hat{\sigma}_{r+1,T}$$

where

$$\hat{\sigma}_r = \begin{cases} \sigma_r \cup \{B_{r+1}\} - \{W\}, & \text{if } A_r \in \sigma_r \\ \sigma_r \cup \{A_r\} - \{W\}, & \text{otherwise} \end{cases}$$

$$\hat{\sigma}_m = \sigma_m \cup \{W\} - \{Z\}$$

$$\forall k: \quad m < k < r : \hat{\sigma} = \hat{\sigma}_k \cup \{A_k\} - \{A_{k+1}\}$$

QED

Chapter 9

Switching Rates of Flat Algorithms

In this chapter we will derive bounds on the switching rates for the Flat Algorithms. Our analysis depends on the fact, which we prove, that the Flat Algorithms are event-driven. By this we mean that its transitions from processing one task to another (i.e., processor switches) are not spontaneous but instead are initiated by arrival events (i.e., the arrival of a task for processing at the beginning of its processing period) and completion events (i.e., the completion of processing of a task before or at the end of its processing period). We start by proving that the deadline algorithm is event-driven in the single processor case. We then address the multiprocessor case, and show that the deadline algorithm is event-driven in the multiple processor setting as well. Finally we show that the Paris Algorithm is event-driven.

9.1 The Single Processor Deadline Algorithm

In this section we will derive a bound on the switching rate of the standard deadline algorithm. The following fact will be extremely useful:

Fact 9.1.1 For single processor systems, the deadline algorithm is event-driven.

Proof: There are two kinds of process switches in a single processor

deadline algorithm.

1. A switch occurs whenever the currently running task completes, allowing another task to be processed.

2. A preemptive switch occurs whenever an arriving task has an earlier deadline than the currently running task.

   A completion event always causes a process switch (unless the processor becomes idle.) An arrival event may cause at most one preemptive process switch. The proof is complete since no other process switches occur in a single processor deadline algorithm.

QED

Fact 9.1.1 allows us to note the following consequences:

1. If L_0 is the number of arrivals in the system period [0,T), the quantity $2L_0$ is an upper bound for the number of process switches in [0,T).

2. The upper bound above is realistic, since there are systems whose deadline schedule approaches it. For example, the system $A = (1,2)$, $B = (4,8)$ has a deadline schedule $ABABABAB$ with $2L_0 - 2 = 8$ switches.

3. The deadline algorithm does not minimize the number of switches. The above example above has a schedule $ABBAABBA$ with $\frac{3L_0-1}{2} = 6$

switches.

4. Small modifications to the standard deadline algorithm can reduce the number switches. Comparing the schedules in the preceding two items, we see that by modifying the basic deadline schedule we can obtain a schedule with the minimum number of switches for the example above.

9.2 The Multiple Processor Deadline Algorithm

There are two major differences between the single processor and multiple processor cases of the deadline algorithm.

1. In the multiprocessor case, the deadline algorithm is not optimal in that there are feasible systems which have no deadline algorithm.

2. In addition to *"free"* (i.e., unforced) decisions to switch processes, the multiprocessor deadline algorithm includes forced process switches caused by the fact that some task (or some queue) can become *"urgent"* in that it must run immediately even though no arrival or completion event has occurred. This occurs, for example, in the system

$$A = (2,4), \quad B = (2,4), \quad C = (7,8)$$

whose deadline schedule includes a forced switch at time $t = 1$:

ACCCCCCC
BBA-AABB

Surprisingly, in spite of these differences, the multiple processor deadline algorithm is still event-driven in the sense that all switches can still be identified with a unique (i.e., distinct) arrival or completion event. Thus, as in the single processor case, number of switches required by the multiple processor deadline algorithm is bounded above by $2L_0$.

In addition to the standard switches inherent to the single processor deadline algorithm, the multiple processor deadline algorithms (such as the Flat algorithms which implement the Paris constraints) include two additional types of switches:

1. switches to tasks which become urgent in that if they do not receive processing immediately they will fail, and

2. Paris Constraint switches propes.

We show that these "forced" process switches do not invalidate this property, so that each switch can still be identified with a unique arrival or completion event.

Fact 9.2.1 A multiprocessor deadline algorithm which includes urgency switches is event-driven.

Proof : We arrive at our result by increments. First we consider the non-optimal class of deadline algorithms which allow urgency-based switching.

When the system is running,each task is in one of three possible states at any given time:

- it is **Idle** : there is no work that remains to be done on that task until the next arrival,

- it is **Urgent** : unless the task begins processing immediately it will fail,

- it is **Ready** : work remains to be done on the task but it has some *"slack"* and can wait.

For a feasible system, the initial state of a task is either

Idle : $\mu = 0$,

Ready : $0 < \mu < 1$, or

Urgent : $\mu = 1$.

Note that

1. A task which is initially **Idle** or **Urgent** will never change state.

2. A task whose initial state is **Ready** may change state but will always return to **Ready** state.

3. There are no direct transitions

Idle $\rightarrow$ **Urgent**, **Urgent** $\rightarrow$ **Idle**

Thus, during each period of the system, a task may go through a sequence of transitions

(Ready→ Urgent → Ready || Ready → Idle → Ready)*

While the transition **Ready → Urgent** seems spontaneous, in that it is not directly linked to an event (arrival or completion), each such transition corresponds one-to-one with a transition **Urgent → Ready.** This latter transition occurs at a deadline so that it is linked to *two* events: a completion and an arrival.

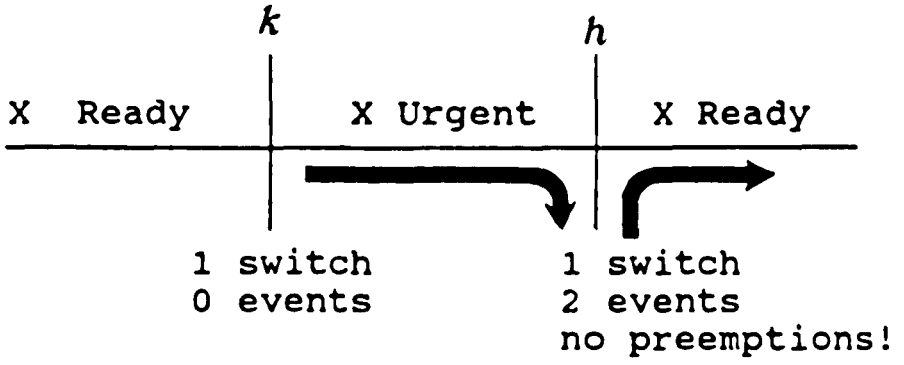


Figure 9.1 event, switch.

Each of the transitions Ready → Urgent and Urgent → Ready causes a process switch, and there are two events associated with the pair. We select the following mapping,

a. the Ready → Urgent switch at time k is mapped to the arrival at time h,

b. the Urgent→ Ready switch at time h is mapped to the completion at time h.

To summarize, each transition of system state is the result of an arrival or completion event. A deadline algorithm causes three kinds of process switches, depending on the event which triggers the switch.

Completion Switches:

- The completed task has a state change

  Ready → Idle or Urgent → Ready.

Arrival Switches:

- A newly arrived task preempts a currently running task. This occurs when the current task is Ready, and the new task has an earlier deadline. This type of preemption is never delayed, but occurs immediately upon the arrival of the preempting task.

- A (previously arrived) waiting task has a state change Ready → Urgent and so preempts a Ready task with an earlier deadline. This type of preemption does not occur in single processor systems.

The key is to note that the types of preemptions above occur on disjoint arrival events: a simple preemption can only be mapped to an arrival which occurs at that moment, but urgent switches are mapped to arrivals which occur at a different (later) time.

9.3 Switching Rate of The Paris Algorithm

Next, we consider the Paris Algorithm in its general form. We derive bounds on the switching rate of the Paris Algorithm, which is optimal (Chapter 8).

Fact 9.3.1: The Paris Algorithm is event driven.

Proof: To prove the proposition, we show that we can extend the mapping between process switches and events (arrivals and completions), so that we maintain the desirable properties:

- every process switch is associated with an event,

- each event is associated by at most one switch.

Note first that the Paris Algorithm is a deadline algorithm which works off the augmented set

$$C = \{C_{ij} : i \in J, j \in T\}$$

of Paris Constraints, and involves the same three types of process switching discussed above. We can partition this set of constraints into two subsets

$$C = C' \cup C''$$

where C' is the set of basic (original) constraints, and C'' is the addi-

tional (Paris) Constraints. We have already shown that every switch in a deadline algorithm driven by the set C' of basic constraints, including urgency deadlines, is event-driven: i.e., the mapping of process switches into arrival and completion events has the desired properties. All we need is to show that the additional switches due to the set C'' of secondary constraints map into a disjoint set of events, and that this mapping itself satisfies the properties above.

Each Paris Constraint can be mapped to a completion event which is not the image of any process switch associated with the set C'. To prove this assertion, note that some of the Paris Constraints do not cause any switching and are redundant.

Example 9.3.1: Consider again the system

$$A = (1,2) \quad B = (1,3) \quad C = (4,6) \quad D = (5,10) \quad p = 30$$

and a particular schedule, where the Paris Constraints are emphasized.

ACCCCB	|ACABAD	|ACCCCA	|AC	|CCCA	|ACACCC |
BDADAD	|BDDCCC	|BDABDD	|DB	|ABDD	|BDDDAB |

obtained drom the Paris deadline table below.

deadline			*A*	*B*	*C*	*D*		
0	→	[	0	0	0	0	]	
2	→	[	1	0	0	0	]	
3	→	[	0	1	0	0	]	
4	→	[	1	0	0	0	]	
6	→	[	1	1	4	3	]	← ParisConstraint : *D*
8	→	[	1	0	0	0	]	
9	→	[	0	1	0	0	]	
10	→	[	1	0	0	2	]	
12	→	[	1	1	4	1	]	← ParisConstraint : *D*
14	→	[	1	0	0	0	]	
15	→	[	0	1	0	0	]	
16	→	[	1	0	0	0	]	
18	→	[	1	1	4	3	]	← ParisConstraint : *D*
20	→	[	1	0	1	1	]	← ParisConstraint : *C*
21	→	[	0	1	0	0	]	
22	→	[	1	0	0	0	]	
24	→	[	1	1	3	2	]	← ParisConstraint : *D*
26	→	[	1	0	0	0	]	
27	→	[	0	1	0	0	]	
28	→	[	1	0	0	0	]	
30	→	[	1	1	4	3	]	

It is easy to verify that two of the five constraints in C'' did not cause any additional switching in the schedule above, while the remaining three constraints (involving task D at deadlines 6, 18, and 24) each caused an additional switch. To show that these switches are associated with events which are disjoint from events linked to other switches, we note that each constraint in C'' serves to prevent one or more idle cycle in the interval preceding that particular deadline. Thus, if the given constraint were eliminated, the interval would contain an idle cycle and a processor transition BUSY → Idle caused by a completion event. This completion event is not linked to any other process switch. Recall that

a BUSY → Idle transition for a processor is NOT a process switch.

We now display the mappings between process switches and events for the three nonredundant Paris Constraints on D at times 6, 18, and 24. In each case, we display the prefix which would result if the constraint in question were removed. In each case, the transition to an idle processor is caused by the completion of task C, so that the extra switch can be associated with this event. For example, in case (1) below, each of the eight switches in the invalid prefix is mapped 1-1 to the following events:

- the first three arrivals and completions of A

- the first arrival and completion of B

and no switch is associated to the completion of task C.

(1) at time = 6:

invalid prefix

ACCCC-
BDABAD

valid prefix

ACCCCB
BDADAD

(2) at time = 18:

invalid prefix

ACCCCB|ACABAD|ACCCC-|
BDADAD|BDDCCC|BDABAD|

valid prefix

ACCCCB|ACABAD|ACCCCA|
BDADAD|BDDCCC|BDABDD|

(3) at time = 24:

invalid prefix

$$ACCCCB|ACABAD|ACCCCA|AC|CCC-|$$
$$BDADAD|BDDCCC|BDABDD|DB|ABAD|$$

valid prefix

$$ACCCCB|ACABAD|ACCCCA|AC|CCCA|ACACCC|$$
$$BDADAD|BDDCCC|BDABDD|DB|ABDD|BDDDAB|$$

9.4 Discussion

As we have shown above, the Paris Algorithm is event-driven. This means that the amount of switching in one system period[0,T) is bounded from above by $2L_0$, where L_0 is the number of arrivals in the interval. In this context, we note the following.

1. The bound is within a constant factor of 2 of the theoretical lower bound L_0 which, as we have seen, is not in general achievable.

2. The bound is independent of the number of processors, and is the same as the bound for a single processor deadline algorithm.

3. As in the single processor case, there are systems which reach the bound.

4. There exist systems for which the Paris Algorithm (or any deadline algorithm) does not minimize the amount of switching.

5. With minor cosmetic changes, such as,

- using the interval between successive deadlines instead of a single quantum as the basic scheduling step,

- paying attention to continuity in assigning tasks to processors during the interval,

we can improve substantially on this bound. For the example above, the number of arrivals is $L_0 = 33$. The resulting schedule

has 35 switches (i.e., 1.061 L_0)

6. We note that the small changes mentioned above improve the performance of the worst examples of the deadline algorithm. For example, the single processor system

$$A = (1,2), \qquad B = (1,8)$$

whose deadline schedule $ABABABAB$ has $2L_0 - 2 = 8$ switches also has the schedule $ABBAABBA$ with 6 switches. This last is obtained by allocating scheduling time for each interval of length 2 between deadlines (an A and a B each time,) and then tiling the schedule for continuity.

Part IV

Numerical Results and Conclusion

Chapter 10

Simulation Results

In conjunction with the theoretical analysis of the scheduling algorithms presented in the preceding chapters, a great number of simulation studies were performed. The initial motivation for these studies was to develop an understanding of the problems and pitfalls of various approaches. In particular, while a proposed algorithm might appear optimal for logical reasons, simulation studies enabled us to find counter-examples (i.e., feasible task sets for which a proposed algorithm failed to generate a viable schedule) and so avoid effort wasted in analysis. Such counter-examples helped sharpen our reasoning and arrive at better (i.e., optimal) algorithms.

In this chapter we present the results of simulation studies of the switching properties of the queue-based and flat algorithms for the two-processor case ($M = 2$). These have been gathered up in one chapter to facilitate comparison of the various algorithms. For the queue-based algorithms, previous chapters have presented theoretical bounds on the number of switches required by analysis of the worst case. For the Paris Algorithm, we have shown that the bound is $2L_0$. It is of interest to know the number of switches required in general, i.e., average performance. Simulation studies such as those presented here represent an attempt to evaluate average-case performance.

For each of the scheduling algorithms, the same procedure was used

to generate a set of tasks. This procedure consisted of the following sequence of steps:

Step 0: Set $n = 1$, $\mu_{total} = 0$.

Step 1: Generate two random integers between one and twelve, inclusive. Set e_n equal to the smaller of these integers, p_n equal to the larger, and $\mu_n = \frac{e_n}{p_n}$.

Step 2: If $\mu_{total} + \mu_n > 2$, STOP.

Step 3: If $LCM\{p_i, 1 \leq i \leq n\} > 1024$ STOP.

Step 4: Set $\mu_{total} = \mu_{total} + \mu_n$, $N = n$, $n = n + 1$, and GOTO Step 1.

Thus, the period and required processing time of each task is an integer drawn randomly from the set$\{1, 2, ..., 12\}$ (Step 1). Tasks are added until the total processing load first exceeds two (Step 2). For convenience (i.e., ease of examining the schedules actually generated), the system period is restricted to be less than 1024 (Step 3).

For each algorithm, more than 2,000,000 task sets were used in testing. Recall that for any algorithm, the minimum number of switches in a system period is L_0, the number of task arrivals in a system period. For the Fair Algorithm $1.4L_0$ switches were required on average. This is substantially less than the analytically derived worst-case upper bound

of $2ML_0 = 4L_0$. In comparison, for the Paris Algorithm $1.2L_0$ switches were required on average while for the Bali Algorithm $1.05L_0$ switches were required on average.

Chapter 11

Conclusion

11.1 Summary of Results

In this dissertation we have considered the problem of scheduling periodic tasks on real-time (i.e., deadline-driven) multiple processor systems. In order for a schedule to be valid each individual task must be completed before the next periodic request (i.e., we tolerate no missed deadlines). The processors are assumed identical, that is, equally powerful with respect to the tasks to be seviced. The service requirements of the tasks and their recurrence rate are assumed to be known in advance. The tasks are assumed to be independent, meaning that the processing of individual requests of a given task does not depend on the processing of other tasks. The task/processor system is assumed to be feasible in that the total processing load does not exceed that available. Finally, we assume that no task requires more processing than can be provided by an individual processor, and we allow only one processor to work on an individual request at any given time.

We have focused on two classes of scheduling algorithms:

- **Queue-based algorithms**, which partition the set of tasks into queues and assign queues to processors, with most tasks/queues being serviced by a single dedicated processor and the remaining tasks/queues being serviced by at most two processors,

- **Flat algorithms**, whereby a modified deadline algorithm is applied to the unstructured set of tasks augmented with additional deadline constraints which serve to 1) ensure that tasks with more distant deadlines are not deferred processing until it is too late and 2) keep all processors busy (i.e., delay idle cycles) as far into the system period as possible.

Our initial analysis assumed that the time required to switch a processor from one task to another is zero. We show that under this assumption the proposed algorithms are optimal in that they yield a valid schedule for any feasible system. In actuality, the time required to perform such a switch, though small, is not zero and so each switch has a cost in terms of lost processing. Thus, as part of our analysis we study the switching properties (e.g., the maximum number of switches required), of the queue-based and the flat algorithms. In particular, we quantify the worst-case cost of switching, and develop algorithmic modifications to reduce this cost. Finally, we use Monte Carlo simulations to estimate the switching performance one might reasonable expect to observe in practice. These estimate are significantly smaller than the upper bounds, and are in fact much closer to the lower bound (the arrival rate of tasks).

11.2 Areas for Future Research

The many assumptions made regarding tasks and processors suggest a number of areas for future research. The following areas are, to us, among the more interesting.

11.2.1 Real-Time Scheduling on Systems of Unequally Powerful Processors

Throughout this dissertation we have worked under the assumption that the processors are identical, that is, equally powerful with respect to each of the tasks. Suppose we relax this assumption and instead allow the power of each processor to arbitrary. Scheduling on such a system would appear to be far more difficult than on a system of identical processors.

While it is not our intention to give a thorough treatment of such systems, it would appear that perhaps the algorithms developed in this dissertation can be easily extended to the unequal-processor case. To see how this is done, let us first consider a two processor system in which processor π_2 is r times as powerful as processor π_1 where r is an integer. It is easy to see that this unequal-processor system is equivalent to (i.e., has the same processing power as) a system of $r + 1$ identical processors, each of which has the power of processor π_1. Thus, for scheduling purposes, we model this system of two unequally powerful processor in terms of a virtual system of $r + 1$ equally powerful processors, with the tasks of the first virtual processor assigned to π_1 and the tasks of the last r virtual processors running in sequence on π_2. Now, instead of being an integer, suppose that r is a rational number, that is, $r = \frac{v}{w}$, where v and w are integers. Such a system is equivalent to a system of $v + w$ identical processors, where the first v processors have total power equal to π_1 and the last w processors have total power equal to

π_2. This approach may be extended to $M > 2$ processors. Doing so involves defining, in effect, a "virtual processor power quantum" p such that all system processors π_i have power $r_i \times p$ where r_i is an integer. This yields a virtual system model with $r_1 + r_2 + ... + r_M$ unit-quantum processors.

11.1.2 Real-Time Scheduling of Dependent Tasks

Throughout this dissertation we have assumed that the tasks are independent, meaning that the processing of individual requests of a given task does not depend on the processing of other tasks. Suppose we relax this assumption and are instead given a set of task processing order specification which require that certain tasks must have completed processing before certain other tasks may begin being processed. The first question one must face for such a situation is whether or not a viable schedule even exists. This appears to be a very difficult problem. The difficulty of this problem is illustrated by our familiar example of a two-processor system with tasks $A = B = (2, 4)$ and $C = (7, 8)$. While this a viable schedule for this system exists when the tasks are independent, no viable schedule exists if we require that 1 (A or B preceed C, or 2) C preceed A or B. (Note that a viable schedule does exist if we require that A preceed B or B preceed A.)

Again, it is not our intention to give a thorough treatment of this problem but merely to suggest avenues of approach. With respect to the Fair Algorithm, it may be possible to satisfy fairly simple precedence

constraints by carefully assigning tasks to foreground and background queues as appropriate. With respect to the Flat Algorithms, perhaps it is possible to model precedence constraints in a way that allows them to be handled as a modification and/or augmentation (by adding new individual task constraints) of the deadline table.

11.2.3 Dynamic On-Line Schedule Reconfiguration

Throughout this dissertation we have assumed that the service requirements of the tasks and their recurrence rate are be known in advance. However, suppose the service requirements and/or recurrence rates increase or that new tasks are added to the system, but in such a way so that the system reamins feasible. Then that it is likely that the existing schedule will have to be reconfigured on-line. How can this be done so as to minimize disruption, e.g., the assignment of tasks to a different processor. What are the characteristics of schedules which are, in some sense, maximally tolerant of changes in tasking?

References

[1] A. A. Bertossi and M. A. Bonuccelli, Preemptive scheduling of periodic jobs in uniform multiprocessor systems, Information Prcessing Letters, vol. 16, pp. 3-6, 1983.

[2] S. R. Biyabani, J. A. Stankovic, and K. Ramamritham, The intergration of deadline and criticalness in hard real-time scheduling. Proc. IEEE Real-Time Systems Symposium, pp. 152-160, 1988.

[3] T. C. E. Cheng and H. G. Kahlbacher, A proof for the longest-job-first policy in one-machine scheduling, Naval Research Logistics, vol. 38, pp. 715-720, 1991.

[4] E. G. Coffman, Jr., Introduction to deterministic scheduling theory, *Computer and Job-Shop Scheduling Theory*, E. G. Coffman, Jr., Ed., Wiley, New York, pp. 1-50, 1976.

[5] E. G. Coffman, Jr., M. R. Garey and D. S. Johnson, An application of bin-packing to multiprocessor scheduling, SIAM Journal Computing, vol. 7, pp. 1-17, 1978.

[6] M. Dertouzos, Control robotics: the procedural control of physical processes, Proc. of the IFIP Congress, pp. 807-813, 1974.

[7] S. K. Dhall and C. L. Liu, On a real-time scheduling problem, Operations Res., vol. 26, pp. 127-140, 1978.

[8] J. Du and J. Y-T. Leung, Minimizing mean flow time with release time and deadline constraints, Proc. IEEE Real-Time Systems Sympo-

sium, pp. 24-32, 1988.

[9] M. R. Garey and D. S. Johnson, Scheduling tasks with non-uniform deadlines on two processors, Journal ACM, vol. 23, pp. 461-467, 1976.

[10] M. R. Garey and D. S. Johnson, Two processor scheduling with start-times and deadlines, SIAM Journal Computing, vol. 6, pp. 416-426, 1977.

[11] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, pp. 236-243, Freeman, New York, 1979.

[12] K. S. Hong and J. Y-T. Leung, Preemptive scheduling with release times and deadlines, Univ. of Texas at Dallas Technical Report UTDCS 7-87, 1987.

[13] W. A. Horn, Some simple scheduling algorithms, Naval Research Logistics Quarterly, vol. 21, pp. 177-185, 1974.

[14] E. L. Lawler, Recent results in the theory of scheduling, *Mathematical Programming: The State of the Art*, A. Bachem et. al. Eds., Springer, 1982.

[15] E. L. Lawler and C. U. Martel, Scheduling periodically occurring tasks on multiple processors, Information Processing Letters, vol. 12, pp. 9-12, 1981.

[16] J. P. Lehoczky, L. Sha, J. K. Strosnider and H. Tokuda, Fixed priority scheduling theory for hard real-time systems, *Foundations of Real-Time Computing: Scheduling and Resource Management*, A. M.

Van Tilborg and G. M. Koob, Eds., Kluwer Academic, Massachusetts, 1991.

[17] J. Y-T. Leung and M. L. Merrill, A note on preemptive scheduling of periodic real-time tasks, Information Processing Letters, vol. 11, pp. 115-118, 1980.

[18] J. Y-T. Leung and J. Whitehead, On the complexity of fixed-priority scheduling of periodic real-time tasks, Performance Evaluation, vol. 2, pp. 237-250, 1982.

[19] J. W. S. Liu, K-J. Lin, and S. Natarajan, Scheduling real-time periodic jobs using imprecise results, Proc. IEEE Real-Time Systems Symposium, pp. 252-260, 1987.

[20] C. L. Liu and J. W. Layland, Scheduling algorithms for multiprogramming in a hard real-time environment, Journal ACM, vol. 20, no. 1, pp. 46-61, 1973.

[21] C. L. Liu, J. W. S. Liu, and A. L. Liestman, Scheduling with slack time, Acta Informatica, vol. 17, pp.31-41, 1982.

[22] A. K. Mok, Fundamental design problems of distributed systems for the hard real-time environment, Ph.D. dissertation, EECS Department, MIT, May 1983.

[23] A. K. Mok, The design of real-time programming systems based on process models, Proc. IEEE Real-Time Systems Symposium. pp. 5-17, 1984.

[24] A. M. Van Tilborg and G. M. Koob, Eds., *Foundations of Real-Time*

Computing: Scheduling and Resource Management, Kluwer Academic, Massachusetts, 1991.

[25] A. M. Van Tilborg and G. M. Koob, Eds., *Foundations of Real-Time Computing: Formal Specifications and Methods*, Kluwer Academic, Massachusetts, 1991.

[26] C. M. Woodside and D. W. Craig, Local non-preemptive scheduling policies for hard real-time distributed systems, Proc. IEEE Real-Time Systems Symposium, pp. 12-16, 1987.

[27] J. Xu and D. L. Parnas, Scheduling processes with release times, deadlines, precedence, and exclusion relations, IEEE Transactions on Software Engineering, vol. 16, no. 3, pp. 360-369, 1990.

[28] S. Zachos, personal communication, May, 1989.