HAL
open science

An Efficient Scheduling Algorithm for Real-Time Musical Systems

Yann Orlarey

To cite this version:

Yann Orlarey. An Efficient Scheduling Algorithm for Real-Time Musical Systems. International Computer Music Conference, 1990, Glasgow, United Kingdom. pp.194-198. hal-02158945

HAL Id: hal-02158945

https://hal.science/hal-02158945v1

Submitted on 18 Jun 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An Efficient Scheduling Algorithm for Real-Time Musical Systems

Yann Orlarey
Grame, 9, rue du Garet
69001 Lyon - France.
Tel. +33 (0)4 72 07 37 00
orlarey@rd.grame.fr

Abstract *: Scheduling problems hold an important place in most real-time musical systems. We here present an algorithm allowing to solve these problems efficiently and ensuring a bounded low scheduling cost per event in any circumstances. Its principle is to maintain events all the better sorted out as their running time gets closer.*

Introduction

Most of real-time musical systems have to face the problem of organizing in time the various tasks to be achieved. These tasks, which can be very numerous, correspond to operations to achieved at precise dates, as for instance, calculating and transmitting commands to musical equipments or even real-time animating of user interfaces.

The date of these different operations is generally know by the system with a certain advance, of variable length, according to the nature of the operation. The existence of a scheduling mechanism allows the system to remember the tasks to be done at a given date.

The efficiency of the algorithm in use is greatly responsible for the performances of the whole system, particularly if the number of tasks to be scheduled is important. The classical sorting and searching methods [1] for maintaining events in chronological order are, for various reasons, inadequate. Several specific technics have been proposed and the reader will be able to refer to [2] to get a general idea of some of them.

In this paper, we propose an original scheduling algorithm, initially developed by the author in 1985 as part of the MidiLisp project [3], then resumed and perfected to become one of the central parts of the multitask MidiShare system [4]. This algorithm has been presented several times during informal discussions as well as during a lecture at CCRMA in 1987, but until now, it had never been presented in a written form.

Definition of the problem

First, a clear definition of the problem must be given. We consider dated events `e` which correspond to tasks to be done at a precise moments. We equally consider a scheduler `S` characterized by the set `E` of events in wait and a date `D`, the current date. We want to implement three operations on this scheduler : **Reset** defines the initial conditions, **Schedule** inserts an event into the scheduler and, at least, **Clock** gives each time-unit, the list of ready to run events and increments the current date.

We can formalize these three functions in the following way :

Scheduler Definition

Reset(S)

E ← ø	*A the outset, the set of events in wait is empty*
D ← 0	*and the date is zero.*

Schedule(e,S)

E ← E+{e}	*The scheduled event is added to those already in wait.*

Clock(S) → R

R←{e ∈ E | date(e) ≤ D}	*All the events whose date is less or equal to the current date are ready to run.*
E ← E-R	*These events no longer are part of those in wait.*
D ← D+1	*The current date is increased.*
return R	*The events ready to run are returned.*

As can be seen, the events scheduled too late (date(e)<D) have no special treatment, they are simply forced to the current date. Obviously, other strategies are possible.

To solve our problem, there exists a trivial algorithm, extremely efficient in computing time. We thus consider, that the event dates are absolutes 32 bits unsigned integer values. The trivial algorithm simply consists in using a huge table of 2^{32} entries and storing the events into this table using their date as an index. Events having the same date are chained together. This method is obviously not realistic because of the amount of memory it needs. But its performances are excellent.

Trivial Algorithm

Reset(S)

E[0..2^{32}-1] ← <>	*At the outset, the set of events in wait is empty*
D ← 0	*and the current date is zero. (<> is an empty list of events)*

Schedule(e,S)

i ← max (date(e), D)	*The event date determines the table index*
append e to E[i]	*The scheduled event is added to those already in wait at the same date. Late events are forced to the current date.*

Clock(S) → R

R ← E[D]	*The current date is used as an index to find the events ready to run.*
E[D] ← <>	*These events are no longer part of those in wait.*
D ← D+1	*The current date is increased.*
return R	*The events ready to run are returned.*

If we define the cost of an event as the sum of all the operations necessary for its scheduling, this cost is very low here. Moreover, it has two interesting characteristics : it is independent of the number of events in wait in the scheduler and it is independent of the advance with which the event is scheduled. Being insured of a bounded scheduling cost, whatever the case, is very important for a real-time musical system.

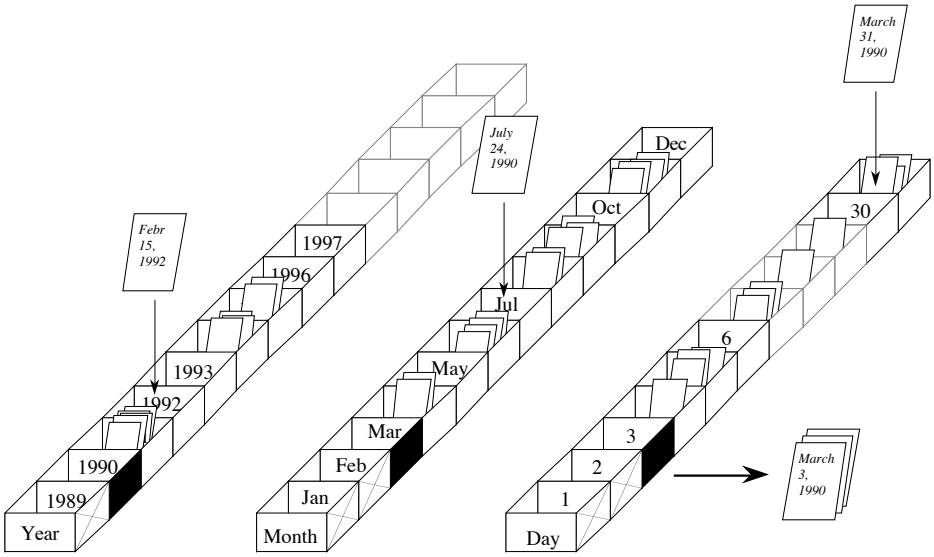
Presentation of the algorithm

The algorithm we propose here is partly derived from the one described above. It follows the same principle while considerably reducing the size of the needed memory, without too much damaging performances. Moreover, it ensures a bounded scheduling cost.

Before any formal description of the algorithm, we are going to study it by the means of the DO-IT[1]. company. This company is somewhat peculiar. Their main activity consists in receiving mail orders to be performed at precise dates. Recently, for instance, they received an order from an aged man, who expected to die soon. In his letter, he asked DO-IT to wish a happy birthday to his single daughter, on the 17th of February, 1998, when she reaches 50.

On of the employees of this firm is in charge of opening the mail every morning and to classify the orders. He must also give the other employees the list of the day's tasks. He must, of course, not spend to much time classifying the arriving orders and quickly find the day's orders. Several people have tried to tackle the job without much success. They all used a huge file in which the orders were classified chronologically. They lost too much time going through the file to classify the newly arrived orders. This bad management lasted until Mr Nay took the job.

Mr Nay was a systematic man and he imagined a new filling system. He used 31 racks for the 31 days of the month, 12 racks for the twelve months of the year and 100 racks for the 100 years of the firm activity. Classifying is quite easily done. If the order is for the current month, he puts it into the day's corresponding rack. If the order is for the current year, he puts it into the month's corresponding rack. If not, he puts the order into the year's corresponding rack.



Current year: 1990 Current month: March Current date: **March 3, 1990**

Once this done, he has but to take the orders in the day's rack and give them to the other employees. Evidently, on the first of the following month, the 31 racks are empty. He then takes all the orders of the starting month and reclassifies them. Following the same principle, at the beginning of each year, the day racks and the month racks are empty. Mr Nay must then reclassify all the year orders in the month racks, then all the orders for the month of January in the day racks.

At this stage, if we analyse the cost of the treatment of an order, we see that it is not manipulated more than three times, whatever the advance it is given with. At worst, it will be placed once in the year's racks, once in the month's racks and at last in the day's racks. Moreover, this cost is independent of the number of orders already registered.

The cost of treatment of an order is therefore very low. But there remains a problem : that is the volume of work accumulated at the beginning of each month and even more, at the beginning of each year. As Mr Nay is a clever man, he found the solution : he distributes his reclassification work all along the year. To do so, he uses supplementary racks : a second set of 31 racks for the reclassification of the following month and a second set of 12 racks for the next year.

Every day, Mr Nay does a little reclassifying. He takes a few orders of the next year and classifies them in the second set of 12 racks. He takes a few orders of the next month and classifies them in the second set of 31 racks. At the beginning of the next month, Mr Nay completely achieves the reclassification of this new month. If his work has been well distributed, there is little or no reclassifying to be done. Then, he exchanges the two sets of 31 racks. At the beginning of the next year, Mr Nay completely achieves the reclassification of this new

[1] This analogy was suggested to me by P. Desain and H. Honing, when I was explaining them this scheduling algorithm.

year. Then, he exchanges the two sets of 12 racks, ends up the reclassifying of the month of January and exchange the two sets of 31 racks.

The analysis of the costs is the same as before, an order is not manipulated more than three times, but this time, the total charge of work is distributed more uniformly. Obviously, the exchange of racks is, in the computer domain, reduced to a simple exchange of pointers.

Description of the algorithm

Let us see now a more formal description of Mr Nay method. As we said before, we consider events dated on 32 bits. We split these dates, not in three parts (day, month, year) as in the preceding example, but in 4 bytes numbered from 0 to 3. The expression date(`e`)`[0]` represent the low order byte of the date of the event `e` and date(`e`)`[0]` the high order byte. We will use a structure corresponding to that of the racks, but with 4 levels : `E0`, `E1`, `E2` and `E3` will be the mains sets of racks and `A0`, `A1`, `A2` the alternates sets of racks. Each level has 256 entries and is indexed by the corresponding byte of the date.

We can implement the three functions of the scheduler in the following manner :

Mr Nay's algorithm

Reset(S)

```
E0[0..255 ] ← <>        At the outset, all the level of the scheduler are empty.
E1[0..255 ] ← <>
E2[0..255 ] ← <>
E3[0..255 ] ← <>
A0[0..255 ] ← <>
A1[0..255 ] ← <>
A2[0..255 ] ← <>
D ← 0                   The current date is zero.
```

Schedule(e,S)

```
d ← max (date(e), D)        If the event is late, it is bounded to the current date.
if      d[3] > D[3]         then append e to E3[d[3]]       The event is inserted
elsif   d[2] > D[2]         then append e to E2[d[2]]       at the level
elsif   d[1] > D[1]         then append e to E1[d[1]]       corresponding to its date.
else    append e to E0[d[0]]
```

Clock(S) → R

```
ResortAlternate(S)          A few events are reclassified.
R ← E0[D[0]]                The low order byte of the current date stands as an index
                            to find the events ready to run.
E0[D[0]] ← <>               These events no longer are part of those in wait
D ← D+1                     The current date is increased.
if D[0]=0 then              If the end of level 0 is reached, the reclassification must be
      ResortEvents(S)       completed and racks exchanged.
return R                    The events ready to run are returned.
```

There remains to define the ResortEvents routine which complete the reclassification at the end of a period and the ResortAlternate which reclassify a few events towards the alternate levels, each time-unit.

```
ResortEvents(S) :

    if D[1]=0 then                          If at end of level 1 :
        if D[2]=0 then                          If at end of level 2 :
            Swap (E2, A2)                           Swap level 2 racks.
            Take all e of E3[D[3]] and              Complete level 3 reclassification
            append e to E2[date(e)[2]]              towards level 2.
        Swap (E1, A1)                           Swap level 1 racks.
        Take all e of E2[D[2]] and              Complete level 2 reclassification
        append e to E1[date(e)[1]]              towards level 1.
    Swap (E0, A0)                           Swap level 0 racks.
    Take all e of E1[D[1]] and              Complete level 1 reclassification
    append e to E0[date(e)[0]]              towards level 0.


ResortAlternate(S) :

    if D[3]<255 then                        If not at the end of level 3 :
        take some e of E3[D[3]+1]and        A few events are reclassified
        append e to A2[date(e)[2]]          from level 3 to alternate level 2.

    if D[2]<255 then                    If not at the end of level 2 :
        take some e of E2[D[2]+1]and        A few events are reclassified
        append e to A1[date(e)[1]]          from level 2 to alternate level 1.
    else                                    If, on the contrary, at end of level 2 :
        take some e of A2[0]and             A few events are reclassified from
        append e to A1[date(e)[1]]          alternate level 2 to alternate level 1.

    if D[1]<255 then                    If not at the end of level 1 :
        take some e of E1[D[1]+1]and        A few events are reclassified
        append e to A0[date(e)[0]]          from level 1 to alternate level 0.
    else                                    If, on the contrary, at end of level 1 :
        take some e of A1[0]and             A few events are reclassified from
        append e to A0[date(e)[0]]          alternate level 1 to alternate level 0.
```

The number of events re-sorted by ResortAlternate is obviously important for the regulation of the system. We suggest to take twice the average number of events processed by the system per time unit.

Conclusion

The algorithm we have presented here has been used for several years in different musical systems. It can be used to solve scheduling problems efficiently. Its functioning is adapted to real time context because it ensures the absence of very unfavourable cases. The total scheduling cost of an event is bounded by a small constant, whatever the advance with which the event is given and whatever the number of events already in wait.

As often in computer science, an algorithm is a compromise between speed and memory space. This one is no exception to the rule. We have used 4 levels but other choices are obviously possible according to the nature of the problem and the memory space available. The less levels, the more efficient the algorithm. The simplest case being of course, the use of one level, which brings us back to the trivial algorithm of the beginning.

References

[1] Knuth D. E. [1973] : «Sorting and Searching»
The Art of Computer Programming vol. 3.
Addison Wesley.

[2] Dannenberg R. B. [1986] : «A Real Time Scheduler/Dispatcher».
Proceedings of the ICMC 1986.

[3] Boynton L., Lavoie P., Orlarey Y., Rueda C., and Wessel D. [1986] : «MIDI-LISP: A Lisp based Music Programming Environment for the Macintosh»
Proceeding of the ICMC 1986

[4] Orlarey Y., Lequay H. [1989] : «MidiShare : A real time multi-tasks software module for Midi applications».
Proceedings of the ICMC 1989.