An Efficient Scheduling Algorithm for Real-Time Musical Systems

Yann Ortega
Grame, 6 quai Jean Moulin
69001 Lyon - France.
Tel. (33) 78 39 32 02

Abstract: Scheduling problems are important in most of real-time musical systems. We here present an algorithm allowing to solve these problems efficiently and providing a bounded low scheduling cost per event in all 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 ordering in time various tasks, as for example, calculating and transmitting commands to musical devices or real-time animating user interfaces. When the dates of these different operations are known in advance, the existence of a scheduling mechanism, capable to sort and process the tasks by date, greatly simplifies the design of musical application.

The performances of the whole system depends greatly on the efficiency of the used algorithm, 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 techniques have been proposed and the reader can refer to [2] to get a general idea of some of them.

In this paper, we propose an original scheduling algorithm, patented by the author, initially developed in 1985 as a part of the MidiLisp project [3], revised and improved later to become one of the central parts of the multitask MidiShare system [4].

The problem definition

First of all, the problem definition must be given. We consider dated events corresponding to tasks to be done at given moments. We also consider scheduler S characterized by set E of events in wait and current date D. We want to implement three operations on this scheduler: Reset to define the initial conditions, Schedule to insert an event into the scheduler and, finally, Clock to give each time-unit, the list of ready to run events and to increment the current date.

Scheduler Definition

Reset(S)
E <- ∅
D <- 0
As the outset, the set of events in wait is empty
and the current date is zero.

Schedule(e, S)
E <- E + {e}
The scheduled event is added to those already in wait.

Clock(S) -> R
R <- {e of E | date(e) ≤ D}
All the events which dates are less or equal
to the current date are ready to run.

ICMC GLASGOW 1990 PROCEEDINGS
s ← E-R
D ← D+1
return R

These events are no longer part of those in wait.
The current date is increased.
Ready to run events are returned.

As one can see, the events scheduled too late (date (e) < d) 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
decide first consider that the event dates are absolute or 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 with the same date are chained together. This method is obviously not
realistic because of the amount of memory it needs. But its performance is excellent.

Trivial Algorithm

Reset(S)
S[0..2^{32}−1] ← o
G ← 0.
and the current date is zero. (<G> is an empty list of events)

Schedule(e,S)
I ← max(date(e),G)
append e to S[I]
The event date determines the table index
The scheduled event is added to those already in wait
at the same date. Last events are forced to the current date.

Clock(S) → R
R ← S[0]
The current date is used to find the events ready to run.
E[0] ← o
Triple events are no longer part of those in wait.
The current date is increased.
return R
Ready to run events are returned.

If we define the cost of an event as the sum of all the operations necessary for its scheduling, this cost
is quite low. 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 sure of a bounded scheduling cost, whatever the case, is very important for a real-time 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 the
performance. Moreover, it provides a bounded scheduling cost.

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

One of the employees of this firm is in charge of opening the mail every morning and classifying the
orders. He must also give the other employees the list of the day’s tasks. He must, of course, not
spend too much time classifying the arriving orders and quickly find the day’s orders. Several people
have tried to tackle the job without much success. All of them used a huge file where 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’s
activity. Classifying is quite easily done. If the order is for the current month, he puts it into the
appropriate day’s rack. If the order is for the current year, he puts it into the appropriate month’s rack.
If not, he puts the order into the appropriate year's rack.

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 day of the following month, the 31 racks are empty. Then he 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. Then Mr Nay must 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 analyze 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 amount of work accumulated at the beginning of each month and even more, at the beginning of each year. Since 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: an additional set of 31 racks for the reclassification of the following month and an additional set of 12 racks for the next year.

Every day, Mr Nay does a little reclassification work. He takes a few orders of the next year and classifies them into the second set of 12 racks. He takes a few orders of the next month and classifies them into 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 reclassification to be done. Then, he exchanges the two sets of 31 racks. As at the beginning of the next year, Mr Nay completely achieves the reclassification of this new 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 in this case, the total amount 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's method. As we have said before, we consider events with 32 bits date. We split these dates, not in three parts (day, month, year) as in the above example, but in 4 bytes numbered from 0 to 3. The expression date[0..3] represents the low order byte of the date of the event and date[3] 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 (E)
E0[0..255] <= <> At the outset, all the levels of the scheduler are empty.
E1[0..255] <= <>
E2[0..255] <= <>
E3[0..255] <= <>
A0[0..255] <= <>
A1[0..255] <= <>
A2[0..255] <= <>
O <= O
```

The current date is zero.

ICMC GLASGOW 1990 PROCEEDINGS
Schedule(S):

d <- max (date(e), D)   If the event is late, it is bounded to the current date.
if D[3] > D[2] then append e to E3(d[3])       The event is inserted
    at the level
else if d[1] > D[2] then append e to E1(d[1])
else append e to ED(d[0])

Clock(S) -> R
ResortAlternate(S), R <- ED(O[0])
    A few events are reclassified.
    The low order byte of the current date stands as an index
    to find the ready to run events.
    These events no longer are part of those in wait
    The current date is increased.
    If the end of level 0 is reached, the reclassification must be
    completed and racks be exchanged.
    The events ready to run are returned.

return R

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

ResortEvents(S) :
    if D[1]=0 then
        if D[2]=0 then
            Swap (E1, A2)
            Take all e of E3[D[3]] and
            append e to E2(date(e)[2])
            Swap (E1, A1)
            Take all e of E2[D[2]] and
            append e to E1[date(e)[1]]
            Swap (C0, A0)
            Take all e of E1[D[1]] and
            append e to ED(date(e)[0])
    if at end of level 1:
        if at end of level 2:
            Complete level 3 resort
            towards level 1.
            Swap level 1 racks.
        Complete level 2 resort
        towards level 1.
        Swap level 1 racks.
        Complete level 1 resort
        towards level 0.

ResortAlternate(S) :
    if D[3]<295 then
        take some e of E3[D[3]+1] and
        append e to A2[date(e)[2]]
    if D[2]<295 then
        take some e of E2[D[2]+1] and
        append e to A1[date(e)[3]]
        else
            take some e of A2[0] and
            append e to A2[date(e)[3]]
    if D[1]<295 then
        take some e of E1[D[1]+1] and
        append e to A0[date(e)[0]]
        else
            take some e of A1[0] and
            append e to A0[date(e)[0]]
        if not at the end of level 3:
            A few events are reclassified
            from level 3 to alternate level 2.
        if not at the end of level 2:
            A few events are reclassified
            from level 2 to alternate level 1.
        if, on the contrary, at end of level 1:
            A few events are reclassified
            from alternate level 2 to alternate level 1.
        if, on the contrary, at end of level 1:
            A few events are reclassified
            from level 1 to alternate level 0.
    if not at end of level 1:
        A few events are reclassified
        from 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

ICMC/GLASGOW 1990 PROCEEDINGS
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 wall.

As it often happens in computer science, an algorithm is a compromise between speed and memory space. Ours is not an exception for 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 fewer levels, the more efficient is the algorithm. The simplest case being of course, the use of a single level, which brings us back to the trivial algorithm of the beginning.

References

The Art of Computer Programming vol. 3.
Addison Wesley.


Proceeding of the ICMC 1986.