# 18 Real-Time Scheduling and Computer Accompaniment

Roger Dannenberg

## 18.1 Introduction

Some of the most interesting applications of computers in music involve real-time computer music systems. The term "real time" refers to systems in which behavior is dependent upon time. As a simple example, a program that controls a music synthesizer to perform a piece of music is a real-time program. This sort of program is perhaps the least interesting form of real-time computer music system because it ignores the possibility of live real-time interaction between human performers and computer music systems. More sophisticated approaches can be classified into at least three categories: computer music instruments, computer accompaniment systems, and interactive composition systems. Although all of these categories are inspired by traditional terminology, it should be emphasized that the special properties of the computer force a rethinking of the meaning of terms like "instrument" and "composition." This reorganization of meaning is one of the attractions of computer music.

A computer music instrument is, by analogy to acoustic instruments, a device that produces sound in response to human gesture and control. Most electronic keyboard instruments are now computer controlled; thus they provide examples of traditionally oriented real-time computer music systems. A more innovative instrument can be seen in the sequential drum of Max Mathews [11]. This instrument can play stored sequences of notes when triggered by striking a specially instrumented drum. The position and force on the drum can control different aspects of each note.

A computer accompaniment system is based on the model of traditional accompaniment in which a score is initially provided for both the solo and the accompaniment. The job of the accompanist is to synchronize with the soloist. One of the sound examples is a recording of a trumpet solo accompanied by computer [4].

While accompaniment systems are given musical materials in the form of a precomposed score, interactive composition systems use the computer actually to generate musical materials in response to input from live musicians. Other appropriate terms for this type of system include improvisation and composed improvisation [6]. A short version of *Jimmy Durante Boulevard* [5], a work of composed improvisation, is included as

From *Current Directions in Computer Music Research,* edited by Max V. Mathews and John R. Pierce, Cambridge, MA: MIT Press, 1989.

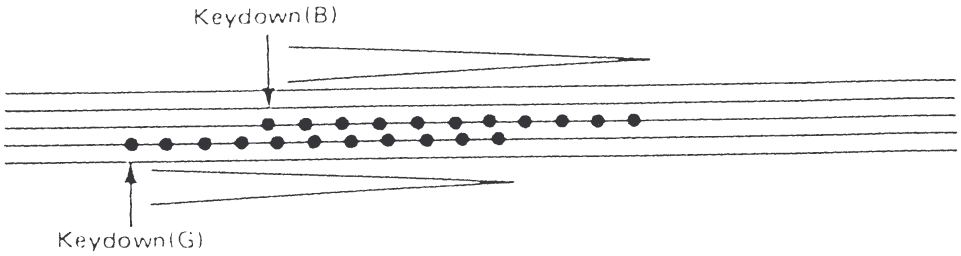


**Figure 18.1**
A musical timing diagram of a potential execution of the *echo* program. Keydown events are indicated by arrows.

another sound example (given on the accompanying compact disk). In this piece, a keyboard, flute, and trumpet are interfaced to a computer, which also controls several music synthesizers. A number of tasks run simultaneously, analyzing input from the flute and keyboard, recording material from the trumpet, computing musical material to be performed, and controlling the synthesizers.

All of these real-time systems depend upon schedulers as a means of coordinating and ordering the execution of many small tasks over the course of time. To illustrate the role of the scheduler more specifically, the following example presents a moderately difficult real-time programming task and solves it in an elegant manner.

### 18.1.1 An Example

The problem is to play a sequence of notes with diminishing loudness to simulate an echo. An echo sequence must be triggered whenever a key is pressed on a keyboard, and sequences may overlap in time. Suppose, for example, that a G and a B are pressed at times indicated by arrows in figure 18.1. The resulting sequences of events are seen to interleave in time. A scheduler is essential for the realization of programs with several independent but simultaneous real-time tasks such as this.

A program that realizes this behavior is given below. It is written in a stylized version of C that should be understandable to anyone familiar with a modern programming language like C, Pascal, or Ada. To avoid clutter, the nesting of program statements will be indicated by indentation rather than by explicit symbols. Algol (and Pascal) assignment (:=) and equality (=) symbols will be used in place of the symbols used in C (= for assignment, == for equality). The meaning should always be

clear from context. **Keywords** will be printed in **boldface**, *program identifiers* will be printed in *italics*, and CONSTANT VALUES will be printed in SMALL CAPITALS.

```
echo(ptich, loudness)
  loudness := loudness-5
  if loudness > 0
    note(pitch, loudness)
    cause(DELAY, echo, pitch, loudness)

keydown(pitch)
  note(pitch, INITIALLOUDNESS)
  cause(DELAY, echo, pitch, INITIALLOUDNESS)
```

This program is executed in an environment that continuously looks for input from a keyboard. When a key is pushed, *keydown* is called with the pitch of the key. The *keydown* routine schedules the *echo* routine to run after a short delay by calling *cause*.

The *cause* routine is critical to the behavior of the program. Its first argument is a delay and its second argument is the name of a routine. The *cause* routine schedules a call to the specified routine after the given delay. Any other parameters to *cause* are saved and passed to the specified routine when it is called. Thus, the *echo* routine will be called DELAY time units after *keydown*. The *echo* routine begins by decrementing its *loudness* parameter. If the parameter is still greater than zero, *echo* plays the given note (by calling *note*) and uses *cause* to schedule another call to *echo*. This will decrease the loudness further, play another note and schedule yet another call. This process repeats until the loudness goes to zero or below, at which time *echo* does nothing. Since *echo* does not schedule anything else at this point, the sequence of notes comes to an end.

Since each call to *echo* runs for a very short time (typically less than 1 ms), there is plenty of processing time to deal with other actions that are scheduled to occur in between the notes of an echo sequence. In particular, many overlapping echo sequences can be active at once. Each sequence uses the same *echo* routine but is characterized by a distinct *pitch* parameter. (With this version of *echo*, pressing a key a second time during a sequence will start a second sequence with the same pitch, which may be undesirable.)

This example illustrates a few important concepts. Virtually all timing in conventional real-time programs is achieved by explicit calls to service

routines like *cause*. It is usually assumed that programs execute very fast except for these calls, which usually have the effect of delaying execution. During the time execution is delayed, there is normally enough time to perform many other actions. By taking advantage of this idle time, other tasks can be processed. The scheduler (in this case *cause*) plays an important role in the management of time-dependent tasks because it is responsible for running tasks in the right sequence and at the right time.

The *cause* routine used in the *echo* program has two nice properties. It not only serves to schedule events, but it also saves parameters and passes them to the events when they are performed. Since saving and passing parameters is largely a straightforward matter of bookkeeping, only the scheduling aspects of *cause* will be considered further.

The *cause* construct is due to Douglas Collinge, who designed the language Moxie [7]. Upon learning about Moxie, this author promptly stole the central idea and the name to create Moxc, a version of Moxie based on the C programming language.[1] Moxc was used to implement *Jimmy Durante Boulevard* and runs on several personal computers.

In the remainder of this chapter, various implementations of real-time schedulers are presented. Then implementations that perform scheduling with respect to a variable-speed time reference are examined. This provides a natural way to implement musical effects such as tempo change and rubato. Finally, a more sophisticated scheduler is presented that incorporates musical knowledge to enhance its ability to adjust tempo dynamically to obtain musical results.

## 18.2 Real-Time Schedulers

The *echo* program illustrates the need to schedule events for performance at a specific time in the future. This section considers a sequence of scheduler implementations, each one containing an improvement over the previous one. The final implementation will exhibit excellent real-time behavior.

It is convenient to define some primitive operations that will be used by each scheduler. The *gettime*( ) operation reads the current real time; for example,

*t* := *gettime*( )

assigns the current time to *t*. The *setalarm*(*t*) operation causes the opera-

tion *alarm*( ) to be invoked at time *t*. If *t* is less than the current time, then *alarm*( ) is invoked immediately. If an alarm is pending due to a previous *setalarm*, then invoking *setalarm* again will cancel the effect of the previous *setalarm*. In other words, at most one alarm can be pending. This corresponds to typical real-time systems that have a hardware counter (the reading of which is modeled by *gettime*) and a hardware timer (the setting of which is modeled by *setalarm*). When the timer times out, a hardware interrupt is generated (modeled by invoking *alarm*).

Using these primitives, the goal is to implement a scheduler with the operation *schedule*(*id*, *time*), where *id* is an event identifier and *time* is the time of the event. The *schedule* operation causes the operation *event*(*id*) to occur at *time* if *time* is in the future. Otherwise, the operation takes place immediately.

The *schedule* and *setalarm* operations are similar in that they each cause another operation (*event* and *alarm*, respectively) to take place in the future. However, *schedule* is more powerful because it "remembers" multiple requests. Since multiple requests can be outstanding, *schedule* associates an identifier with each request. One use of *schedule* is to implement the *cause* routine used in the *echo* program. In this case, *id* would be the address of a block of memory containing a routine entry point and parameters. The *event* operation would run the indicated routine with the saved parameters. Another typical use of *schedule* is to reactivate sleeping processes. In this case, *id* would be the address of the process descriptor that is to be reactivated. Thus, *schedule* is a general building block that can be used in a number of ways.

Two important observations to keep in mind are that (1) the scheduler must keep track of an arbitrary number of pending *schedule* requests, and that (2) the requests do not necessarily arrive in the same order in which they must be satisfied. Thus, a scheduler must have some way to remember a set of pending requests and a method for sorting requests into time order.

In the implementations that follow, the same notational conventions seen in the *echo* program example will be used. Because the C language notation for structures is rather cumbersome, the following conventions will be followed. A structure with elements A, B, . . . , C is created by calling *new*(A, B, . . . , C). The fields FIELD1, FIELD2, . . . , FIELDN of a structure *s* are denoted by *s*.FIELD1, *s*.FIELD2, . . . , *s*.FIELDN.

### 18.2.1 Implementation 1

A straightforward implementation of the scheduler wakes up and runs at every increment of time and looks at the pending requests to see if one should be satisfied. The data structure consists of *requests*, a list of pairs of *id*s and *time*s, which is initially empty. The variable *t* is used to compute the next time at which *alarm* should be invoked. The scheduler is initialized by setting *t* and invoking *setalarm* (*setalarm* will immediately generate an interrupt that calls *alarm*):

```
initialize( )
  set requests to EMPTY
  t := gettime( )
  setalarm(t)
```

The *schedule* operation adds an *id* and *time* to the *requests* list:

```
schedule(id, time)
  insert new(id, time) into requests
```

The *alarm* operation searches through the list of requests looking for any whose time has come. It then increments *t* and calls *setalarm* so that *alarm* will be invoked every unit of time.

```
alarm( )
  for each r in requests
    if r.TIME <= gettime( )
      remove r from requests
      event(r.ID)
  t := t + 1
  setalarm(t)
```

**Note**: Since *schedule* and *alarm* operate on the same variables, it is *essential* that *alarm* not be invoked by an interrupt during the execution of *schedule*. In order to simplify this presentation, it is assumed throughout that the executions of *schedule* and *alarm* are always mutually exclusive.

The *schedule* operation has the nice property that it takes a fixed amount of time, assuming *requests* is implemented as a linked list [1]. However, this scheduler suffers from two problems. First, the *alarm* operation must look at every pending request every time it is invoked. As the number of requests goes up, so does the computational cost of *alarm*. Second, *alarm* is invoked

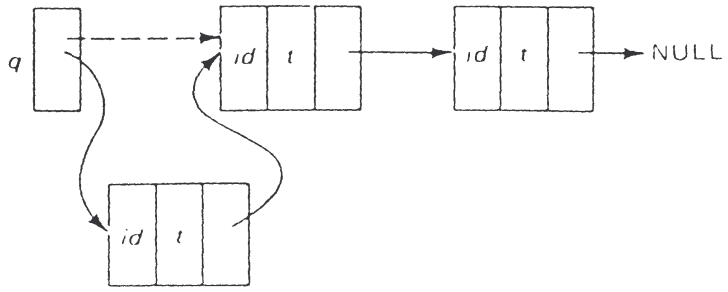


**Figure 18.2**
Inserting a new element into a linked list.

even when there are no requests to be satisfied. This might be tolerable if the time needed to execute *alarm* were small, but that is not the case here. The next implementation uses a *priority queue* to reduce the cost of *alarm* operations.

### 18.2.2 Implementation 2

A *priority queue* is a data structure that contains a set of items, each of which has a *priority*. In this case, items will be requests ([id, time] structures), and the priority of an item will be determined by its time component. Priority queues have an *insert* operation, which adds an item to the queue, an *inspect* operation, which returns the [id, time] structure with the highest priority (earliest time), and a *remove* operation that removes the item with the highest priority. Since priorities are static (that is, the priority cannot change after the insert operation), an efficient implementation is to represent the queue as a linked list, sorted by increasing time values. Each list element is a structure with three fields: ID, TIME, and NEXT, where the NEXT field is a link to the next list element. As explained above, *new*(*id*, *time*, *next*) will allocate and initialize a new linked list node. To insert a new *id* and *time* in the list, the *new* function is called to allocate a structure to hold the *id* and *time* and remember (in the NEXT field) a reference to the remainder of the list. Figure 18.2 illustrates the operation of inserting a new item into a list, an operation that will be used throughout this chapter. The operation shown is

$q := new(id, t, q)$

and the value of $q$ before the assignment is indicated by a dotted line. NULL represents a pointer to the empty list.

For the scheduler, the queue is initially a list with one node whose time component is infinity (this simplifies other parts of the implementation).

```
newqueue( )
   return new(0, infinity, NULL)
```

An implementation of *insert* is

```
insert(queue, id, time)
   if time < queue.TIME
      return new(id, time, queue)
   pointer := queue
   while t >= pointer.NEXT.TIME
      pointer := pointer.NEXT
   pointer.NEXT := new(id, time, pointer.NEXT)
   return queue
```

To complete the priority queue implementation, *inspect* and *remove* operations must be provided. In the implementation below, *inspect* returns data from the front of the queue, and *remove* returns a reference to the rest of the queue. Notice also that the problems of storage reclamation are ignored to simplify this presentation:

```
inspect(queue)
   return new(queue.ID, queue.TIME)

remove(queue)
   return queue.NEXT
```

A new scheduler can be implemented using a priority queue. This time, *requests* is initialized as a priority queue:

```
initialize( )
   requests := newqueue( )
   t := gettime( )
   setalarm(t)
```

The *schedule* routine is as follows:

```
schedule(id, time)
   requests := insert(requests, id, time)
```

Now, since the request with the earliest time is at the front of the queue, *alarm* only needs to look at the front of the list of requests.

```
alarm( )
  r := inspect(requests)
  while gettime( ) >= r.TIME
    event(r.ID)
    requests := remove(requests)
    r := inspect(requests)
  t := t + 1
  setalarm(t)
```

This implementation solves the first problem of Implementation 1, namely, the *alarm* operation now takes time proportional to the number of requests ready to be satisfied plus a small fixed overhead. This is quite good, since satisfying the requests by calling *event* is likely to dominate the total computation cost. This implementation still suffers from the fact that a small fixed cost is incurred every time unit because *alarm* is invoked whether or not there are pending requests.

Even though this scheduler is a great improvement over Implementation 1, a new problem has been introduced. Recall that in Implementation 1, the *schedule* operation took a fixed amount of time. In the new implementation, the *schedule* operation takes time proportional to the number of pending requests in the worst case. This is because the *insert* operation may have to scan the entire queue in order to find the right place to insert a new item. Thus, *alarm* is now efficient at the cost of making *schedule* rather inefficient. Nevertheless, most real schedulers are essentially identical to Implementation 2.

### 18.2.3 Implementation 3

One way to improve the previous scheduler is to use a faster implementation of priority queues. Since faster implementations are not common knowledge, at least not as common as they should be, a short digression on fast priority queues is in order to present one algorithm for priority queues.

A data structure called a *heap* provides a fast way to implement a priority queue [3]. The time required to insert and remove elements is proportional to the logarithm of the number of elements in the queue. A heap is a complete (or full) binary tree in which each node stores a value that is less than or equal to the values of its children. Heaps are typically stored in an array where the first element (at index 1) is the root. The children of a node at index $i$ are at array locations $2i$ and $2i + 1$. Figure 18.3 illustrates a heap and its array representation.

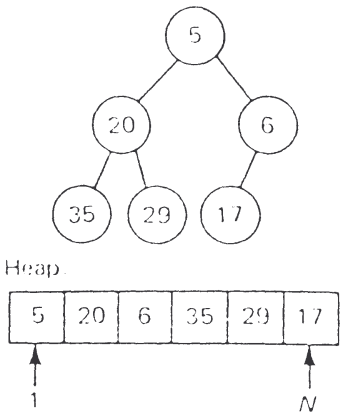


**Figure 18.3**
A heap data structure uses an array to represent a binary tree.

To treat the heap as a priority queue, *insert* and *remove* operations are needed. Let $H[1]$ through $H[N]$ be a heap. Initially, $N = 0$ and the heap is empty. To insert an element into the heap, the element is added to the end of the heap and is "bubbled up" by iteratively exchanging the element with its parent until the heap property is satisfied. To remove an element from the heap, the first element of the array is taken. To restore the heap property, the last element of the heap is moved to the beginning of the array (the top of the tree) and "bubbled down" as follows: the element is swapped with the smallest of its children until all of the element's children are greater than or equal to the element. The routines follow:

```
initialize( )
  N := 0

insert(time)
  N := N + 1
  H[N] := time
  i := N
  while i > 1
    parent := floor(i/2)
    if H[parent] <= time
      return
    swap(H[i], H[parent])
    i := parent

remove( )
  H[1] := H[N]
```

```
    N := N - 1
    i := 1
    child := 2 * i
    while child <= N
      if child + 1 <= N
        if H[child + 1] < H[child]
          child := child + 1
      child is now the index of the least child
      if H[i] <= H[child]
        return
      swap(H[i], H[child])
      i := child
      child := 2 * i

inspect( )
    return H[1]
```

In these routines, only the times are stored in the heap. For use in a scheduler, an event must be associated with each time. This is a straightforward extension to make once the algorithms are understood.

Notice that the size of array $H$ sets an upper limit to the number of events that can be stored on the heap. A priority queue based on $2 \cdot 3$ trees [1] offers similar high performance without an intrinsic upper bound on the queue size.

Be redefining *newqueue*, *insert*, *inspect*, and *remove*, the scheduler can be improved without changing either *schedule* or *alarm*! Since this is a straightforward substitution, no further implementation details are presented here.

Using a heap for the priority queue changes the cost of both the *schedule* and the *alarm* operations. The *schedule* operation costs are reduced from something proportional to $n$ to something proportional to $\log n$. This improvement is quite significant when $n$ is large. The *alarm* operation, which formerly cost a fixed amount per satisfied request, now costs something proportional to $\log n$ per satisfied request. This is a small price to pay considering the savings made in the *schedule* operation.

### 18.2.4 Implementation 4

More improvements are possible. Implementation 4 incorporates an optimization that avoids the *alarm* operation unless necessary. The trick is to

use *setalarm* to invoke *alarm* only at the proper time. Both *schedule* and *alarm* must be changed, and *setalarm* is not called when the system is initialized:

```
initialize( )
   requests := newqueue( )

schedule(id, time)
   requests := insert(requests, id, time)
   r := inspect(requests)
   setalarm(r.TIME)

alarm( )
   r := inspect(requests)
   while gettime( ) >= r.TIME
      event(r.ID)
      requests := remove(requests)
      r := inspect(requests)
   setalarm(r.TIME)
```

Notice that both *schedule* and *alarm* end by calling *setalarm*(*r*.TIME), where *r*.TIME is the time of the earliest pending request, as determined by *inspect*.[2] Thus, *alarm* will always be invoked when the next pending request is ready, but never earlier.

Implementation 4 saves a fixed cost at every time increment for which no request is ready. If the unit of time is very short, this can be a significant savings. Many real implementations use this technique to optimize Implementation 2. In most computer music applications, however, a time resolution of several milliseconds is adequate.[3] Therefore, the overhead of invoking the *alarm* operation at every unit of time could amount to less than 1% of the computing resources in a carefully written scheduler.

In a real-time computer music system, frequent operations that consume only a small amount of processing time are not as problematic as less frequent operations that involve significant computation. Following this line of reasoning, Implementation 4 has not led to a substantial improvement: an unimportant aspect of the scheduler has been optimized while significant overheads remain in the form of the priority queue operations invoked by *schedule* and *alarm*. Implementations 5 and 6 will incorporate a strategy that largely removes this problem.

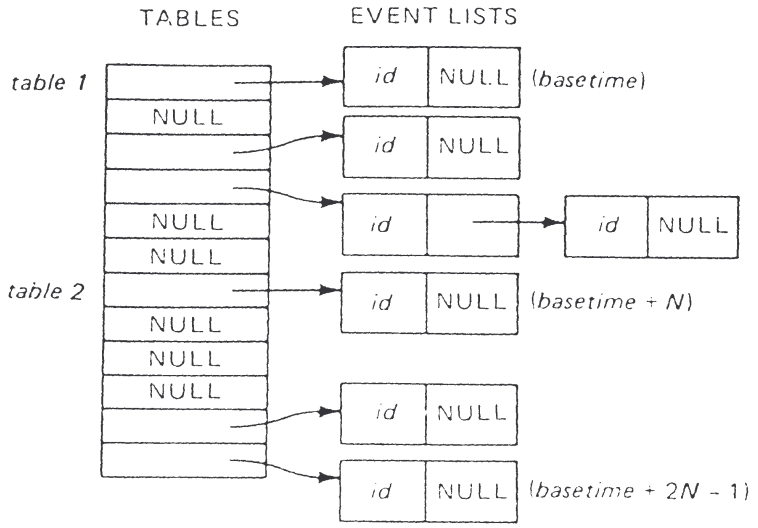


**Figure 18.4**
*table 1* and *table 2*, with lists of events scheduled for times between *basetime* and *basetime* $+2N-1$.

### 18.2.5 Implementation 5

Implementation 5 is quite similar to Implementation 3, but a different method is used to avoid the overhead of the priority queue. The idea is to use the fact that at a time resolution of several milliseconds, a separate list of requests can be maintained for each unit of time for several seconds into the future. Since there is a separate list for each unit of time (see figure 18.4), scheduling an event amounts to inserting the event *id* in the proper list. This always takes a small constant amount of time. Furthermore, performing events is very fast. At each unit of time it is only necessary to advance to the next list and perform all of the events in the list. Thus, there is a small constant amount of time necessary per event and per time unit to perform events. This is much better than the performance of the previous scheduler when there are many events waiting in the queue.

The scheduler presented in this section will provide constant time scheduling and constant time event dispatching, but it will only allow scheduling a finite amount into the future. This limitation will be removed in the next section.

For reasons that will become clear later, two tables (arrays) of lists called *table1* and *table2* are used, and each will store $N$ lists, named *table1*[0] ... *table1*[$N-1$] and *table2*[0] ... *table2*[$N-1$]. The operation *swap*(*table1*,

*table2*) exchanges the contents of *table1* and *table2*.[4] When the scheduler is started, each table is filled with empty lists, *basetime* is initialized to the current time, and *settime* is used to invoke *alarm*:

```
initialize( )
  for i := 0 to N − 1
    table1[i] := NULL
    table2[i] := NULL
  basetime := gettime( )
  t := basetime
  setalarm(basetime)
```

Now, *table1*[*i*] (for any *i* between 0 and $N - 1$) will be a list of requests scheduled for time *basetime* + *i*, and *table2*[*i*] will hold requests scheduled for time *basetime* + $N$ + *i*. In figure 18.4, *table1* and *table2* are shown. The scheduled time for several events is indicated in parentheses to the right of the events. Each table entry stores a possibly empty list of structures with two fields, *id* and *next*. Note that the time of an event is implied by the choice of table entry, so there is no need to store the time in the lists.

At time *basetime* + $N$, all requests in *table1* will have been satisfied, so *table1* can be reused for future events. This is accomplished by swapping *table1* and *table2* and adding $N$ to *basetime*. Figure 18.5 illustrates the correspondence between table entries and time before and after a swap operation. Notice how *table1* is renamed and relocated in time to become the new *table2*.

The implementation of *alarm* is simple. First, if the end of *table1* is reached, swap tables. Then get the list of pending requests corresponding to the current time and call *event* for each *id* in the list. Finally, clear the current table entry to allow it to be reused when the tables are swapped.

```
alarm( )
  if t = basetime + N
    swap(table1, table2)
    basetime := t
  requests := table1[t − basetime]
  while requests <> NULL
    event(requests.ID)
    requests := requests.NEXT
```

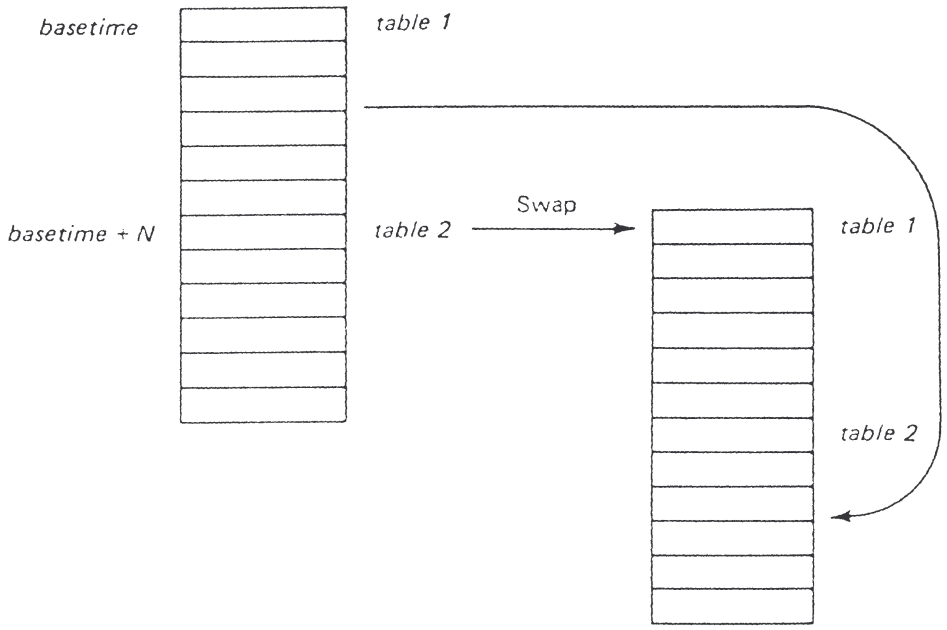


**Figure 18.5**
Data structures for Scheduler Implementation 5.

```
table1[t − basetime] := NULL
t := t + 1
setalarm(t)
```

The *schedule* operation locates the right list and inserts the *id*:

```
schedule(id, time)
  if time < basetime + N
    table1[time − basetime] :=
      new(id, table1[time − basetime])
  else if time < basetime + (2 * N)
    table2[time − (basetime + N)] :=
      new(id, table2[time − (basetime + N)])
  else error( )
```

Note that just before the swap operation, all lists in *table1* are empty as a consequence of *alarm*. This makes *table1* ready to be reused as the new *table2*.

Both *alarm* and *schedule* now take a *constant* amount of time per request. The only problem with this scheduler is that it does not allow us to schedule

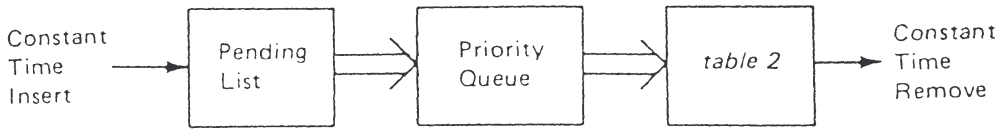


**Figure 18.6**
Flow of event request data from scheduling time to performance time.

events for further than $N$ time units into the future. (The maximum time at which an event can be scheduled is *basetime* + $2N - 1$, and the current time can be as great as *basetime* + $N - 1$; the difference is $N$.) Notice that if a request is scheduled further ahead of time than this, there will be no table entry to receive it. However, if requests are always made with times less than $N$ time units into the future, *schedule* will always work, and its real-time characteristics are almost ideal (the only possible reservation being the overhead incurred by invoking *alarm* when there are no requests to be met). The last implementation will extend this one to allow requests to be made at arbitrary times in the future.

### 18.2.6 Implementation 6

The key idea of this implementation is to use the previous implementation to handle all near-term requests and to add a fallback strategy for long-term requests. Notice that the scheduler has at least $N$ units of time to deal with any far-term request, so it is possible to delegate most of the work to a background process that runs when there is no other work to do. Since practical real-time systems have a large amount of idle time, this strategy is quite reasonable: time-critical (near-term) scheduling operations will execute in constant time, and non-time-critical (long-term) scheduling operations will take more processing time, but will take advantage of otherwise idle processing time.

The algorithm works as follows: any request that cannot be immediately entered into a table is put on a simple linked list called *pending*. Note that this takes only a fixed amount of time. In the background, a process uses idle processor time to remove items from the *pending* list and insert them into a priority queue. It also takes items from the queue and inserts them into tables as this becomes possible. The flow of data for events scheduled after *basetime* + $2N$ is illustrated in figure 18.6. The double arrows represent transfers that take place in the background.

The timing constraints on the background process are simplest to understand if the requirements are made a little stronger than absolutely neces-

sary. At the moment just after *basetime* is incremented, *table2* is empty and represents lists of events that are to take place in the interval from *basetime* + $N$ to *basetime* + $2N - 1$. The *pending* list may contain requests for this interval, but no more requests for the interval will be added to *pending* because any new request for that interval will be inserted directly into the table. Thus, when *basetime* is incremented, there are $N$ time units in which to insert the *pending* list into the priority queue and then to transfer to *table2* everything in the queue with a time earlier than *basetime* + $2N$. Since new requests with times of *basetime* + $2N$ or greater might be scheduled while this background task is running, it is convenient to use two lists, *pending1* and *pending2*. Requests will be added to *pending1* and removed from *pending2*. A *swap* operation will exchange them when *basetime* is incremented.

Figure 18.7 illustrates timing relationships. During the time interval labeled 1, events are moved from *pending2* to the priority queue and then to *table2*. Meanwhile, any event scheduled for time interval 2 is placed on *pending1*. At *basetime* + $N$, a swap occurs, exchanging *pending1* and *pending2*, *table1* and *table2*, and adding $N$ to *basetime* so that the whole sequence repeats. The resulting scheduler is given below:

```
alarm( )
  if t = basetime + N
    swap(table1, table2)
    swap(pending1, pending2)
    basetime := t
  requests := table1[t - basetime]
  while requests <> NULL
    event(requests.ID)
    requests := requests.NEXT
  table1[t - basetime] := NULL
  t := t + 1
  setalarm(t)

schedule(id, time)
  if time < basetime + N
    table1[time - basetime] :=
      new(id, table1[time - basetime])
  else if time < basetime + (2 * N)
    table2[time - (basetime + N)] :=
      new(id, table2[time - (basetime + N)])
```

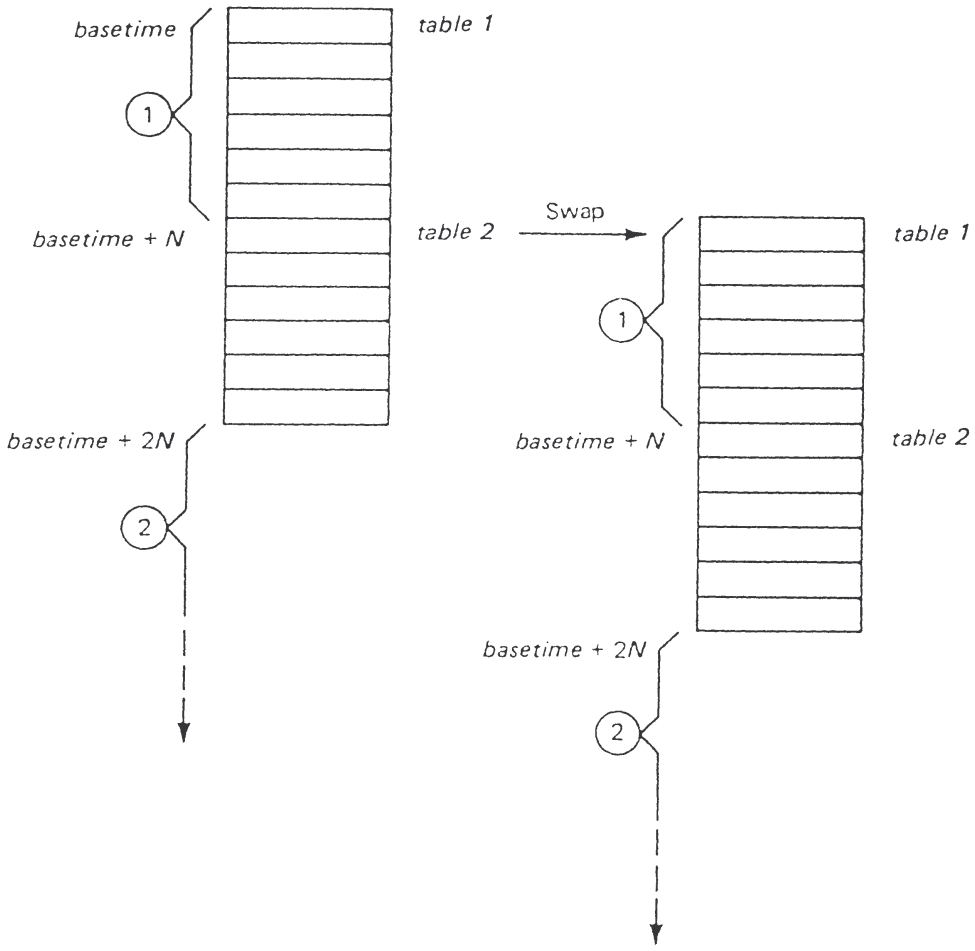


Figure 18.7
Data structures for Scheduler Implementation 6.

```
    else
      pending1 := new(id, time, pending1)

background(starttime)
  mybase := starttime
  while TRUE
    mybase := mybase + N
    while mybase > basetime
      do nothing
    while pending2 <> NULL
      insert(queue, pending2.ID, pending2.TIME)
```

```
    pending2 := pending2.NEXT
  q := inspect(queue)
  while q.TIME < mybase + (2 * N)
    schedule(q.ID, q.TIME)
    queue := remove(queue)
    q := inspect(queue)
  if mybase <> basetime
    error( )
```

The important variables are

| | |
|---|---|
| *table1* | events scheduled for *basetime* <= *t* < *basetime* + *N* |
| *table2* | events scheduled for *basetime* + *N* <= *t* < *basetime* + 2*N* |
| *pending2* | temporarily holds events scheduled during *basetime* − *N* <= *t* < *basetime* for times greater than *basetime* + *N* |
| *pending1* | temporarily holds events scheduled during *basetime* <= *t* < *basetime* + *N* for times greater than *basetime* + 2*N* |

Initialization is as for the previous scheduler, with a few additions:

```
initialize( )
  for i := 0 to N − 1
    table1[i] := NULL
    table2[i] := NULL
  basetime := gettime( )
  t := basetime
  setalarm(basetime)

  pending1 := NULL
  pending2 := NULL
  queue := newqueue( )
  start background(basetime)
```

The background process is started and is passed the initial value of *basetime* as a parameter. The background process uses this value to determine when to perform a cycle of its outer loop that moves requests from *pending2* to *queue*, and then from *queue* to *table2*. After these operations, *background* checks to make sure it has completed its task within *N* time units. If it has, *mybase* will still equal *basetime*.

For computer music applications, this scheduler is superior to those considered earlier. The *schedule* and *alarm* operations take a constant amount of time to execute in all cases. When events are scheduled far in the future, there is an additional computational expense proportional to $\log n$ per event, but this expense is delegated to a background process. During each regular interval of $N$ time units (where $N$ is an arbitrary number), the background process must enqueue all requests pending from the previous interval and dequeue all requests pending for the next interval. This interval can be made large if desired in order to minimize the effect of "bursts" of scheduling requests.

Notice that if all events are scheduled for times greater than *basetime* + $2N$, then this scheduler will do slightly more work than Scheduler Implementation 4. The extra work arises from moving each event on and off of both a pending list and a table. In addition, the *alarm* routine must be called every unit of time. However, even in the worst case this scheduler still has a significant advantage because events can be scheduled and dispatched in constant time. This is very important in music where events often come in bursts—for example, at the beginning of a chord with many notes. The high performance during bursts of scheduling or dispatching activity more than makes up for the extra work performed by the background process.

The memory space required by the scheduler is proportional to $N + M$, where $N$ is the time interval size and $M$ is the number of pending requests. There is no way to get around $M$ in any scheduler, and the memory space due to tables of size $N = 1{,}000$ might typically be 8,000 bytes, one-sixteenth of a single 1M-bit memory chip. At a time resolution of 1 ms, this would give an interval time of one second.

Experienced programmers may recognize that a circular buffer could be used in place of the double-buffering scheme of two tables used here. The double-buffering scheme is used here because it makes it easier to understand the requirements that must be met by the background process.

(After this chapter was completed, a paper was independently published by Varghese and Lauck [12] that develops algorithms similar to those in this section. Rather than use a background process to schedule future events, all events are immediately entered into a table at the event time mod $N$. The alarm routine must examine entries in the table and invoke only event requests whose time matches the current time. This is simpler than Implementation 6 but slightly more expensive.)

Scheduler Implementation 6 was designed because other existing scheduler algorithms did not deliver the performance desired for real-time computer music systems. The present design overcomes significant problems associated with other schedulers.

## 18.3 Scheduling with Virtual Time

In each of the schedulers discussed in the previous section, times are referenced to a single clock that is presumed to correspond to real (physical) time. In computer music programs, it is often convenient to have a time reference or references that do not correspond to real time. Consider the conductor's baton, which (among many other functions) measures time in beats.

Because tempo may vary, time as measured in beats may not have a fixed linear relation to real time. By analogy, one can imagine a software scheduler that uses a nonlinear or variable speed time reference.

This concept can be extended to incorporate several simultaneous but independent time references, analogous to having several conductors conducting at different tempi. A further extension is the composition (in the mathematical sense) or nesting of time references [9]. As an intuitive introduction to this concept, imagine taking a recording of a rubato passage of music and varying the playback speed. The resulting tempo will be a composite of two functions, or *time maps*, that map from one time reference to another: the playback speed and the original tempo. In this section, various ways to implement virtual-time schedulers will be considered, starting with a simple extension to the last scheduler in the previous section.

### 18.3.1 A Single-Reference, Virtual-Time Scheduler

The simplest virtual-time scheduler contains a single time reference that can be made to advance at any positive speed with respect to real time. For now, it is assumed that the speed of virtual time relative to real time can be changed within the program by calling *setspeed*($s$), where $s$ is the new speed of the virtual time. The speed variation can be implemented in either hardware or software.

Usually, real-time computer systems have a programmable real-time clock that generates an interrupt every $N$ cycles of a very fast (often 1–10

MHz) system clock. In terms of the schedulers of section 18.2, an interrupt corresponds to calling *alarm*( ). For example, if the system clock period is 1 $\mu$s and the nominal time unit used by the scheduler is 1 ms, then $N$ would be 1 ms/1 $\mu$s = 1,000. If $N$ is changed to 900, the interrupt period will be 0.9 ms. Thus, virtual time (the time reference used by the scheduler) would go faster.

In cases where a hardware solution is not possible, software can be used. The software solution described here will not produce truly periodic intervals like the ones generated in hardware. Instead, the approach will produce the correct average period in the long run. The actual advances of virtual time (or calls to *alarm*) will occur on transitions of the real-time clock, which limits the time resolution of the system. This is typically not a problem since the real-time clock interval is small. A schematic of the software solution follows:

```
Initially:
  d := 0.0

On hardware interrupt:
  d := d + s
  while d >= 1.0
    d := d - 1.0
    alarm( )
```

Both $d$ and $s$ are floating point numbers. Note that if $s$, the speed of virtual time, is exactly 1, then *alarm* is called on every interrupt. The speed can be arbitrary; for example, if $s = 0.71$, then *alarm* will be called 71 times out of every 100 interrupts, and the calls will be spaced fairly uniformly. In general, during an interval of $t$ units of real time, *alarm* is called approximately $st$ times. Thus, the ratio of virtual to real time approaches exactly $st/t = s$, as desired. Also notice that if $s$ is greater than 1.0, *alarm* will at least sometimes be called more than once in response to a single interrupt. This is necessary to get an average *alarm* rate greater than the interrupt rate.

Rather than use floating point numbers as indicated above, fixed point numbers or integers are often used for greater efficiency. The same program can be written with only integer operations. In the version below, the speed $s$ is set to an integer scaled to 1,000 times the desired speed of virtual time. For example, if the desired speed is one-half, $s$ would be 500, and *alarm*( ) would be called on every other hardware interrupt. (The choice of

one thousand is arbitrary. Larger numbers allow greater accuracy in the representation of fractions but require larger integers.) Here is the code:

Initially:
  $d := 0$
  ONE := 1000

On hardware interrupt:
  $d := d + s$
  **while** $d >$ ONE
    $d := d -$ ONE
    *alarm*( )

### 18.3.2 Multiple Reference Schedulers

Problems arise if it is necessary to schedule according to several time references. A simple, but potentially expensive, approach is to operate a separate scheduler for each time reference. The interrupt routine presented above is rewritten as follows:

Initially:
  $d_i := 0.0$ for each $i$

On hardware interrupt:
  for each $i$
    $d_i := d_i + s_i$
    **while** $d_i >= 1.0$
      $d_i := d_i - 1.0$
      $alarm_i$( )

In this approach, there is a separate speed ($s_i$) for each scheduler. Time for scheduler $i$ advances when $alarm_i$ is called. This approach is practical only when the number of schedulers is small because the cost is proportional to the number of schedulers.

### 18.3.3 An Efficient Compromise

It seems wasteful to compute the advance of each virtual clock at every unit of real time, but this is necessary because the speed of a virtual clock can change at any moment. If changes to $s$ are restricted, this problem can largely be eliminated. One possible restriction is to require complete knowledge of how $s$ will change in the future. It is then possible to compute

the real time to which any virtual time will correspond. If the desired real time can be computed when an event is scheduled, there is no need for special virtual-time schedulers.

Unfortunately, for most interactive real-time programs it is too restrictive to require future knowledge of the behavior of $s$. For example, $s$ might be controlled by a slider in real time. Another possible restriction is to allow $s$ to change only with some small advance notice. This would allow a conversion of each virtual time to a real time shortly before the real time occurs.

This idea can be applied to Scheduler Implementation 6 from the previous section. Recall that in scheduler 6, a background process pulls events from a queue and enters them into *table2* while events in *table1* are activated in sequence. For this new scheduler, there will be a separate queue for each virtual-time reference, and the background process will pull events from each queue, translating virtual times into real times and inserting events into *table2*. Once an event is entered into a table, its real time cannot be changed, which is another way of saying $s$ must be known in advance. The worst-case advance notice (or latency, depending on one's point of view) is twice the table size.

The implementation of this scheduler begins with routines for converting virtual time to real time. The calculation assumes that virtual time moves forward at a rate $s$ from the last time at which $s$ was changed:

```
virttoreal(vtime, i)
    return realref_i + (vtime − virtref_i) * s_i
```

The parameter $i$ indicates which virtual-time reference is to be used. For each reference, $s_i$ is the speed, $realref_i$ is the time at which $s_i$ was last changed, and $virtref_i$ is the virtual time at which $s_i$ was changed. Figure 18.8 illustrates this graphically. To change $s$, the following routine is called:

```
setspeed(speed, i)
    time := gettime( )
    virtref_i := virtref_i + (time − realref_i)/s_i
    realref_i := time
    s_i := speed
```

The *setspeed* routine computes the current virtual time based on previous values of $virtref_i$, $realref_i$, $s_i$, and the current real time. Then, $virtref_i$, $realref_i$, and $s_i$ are updated.

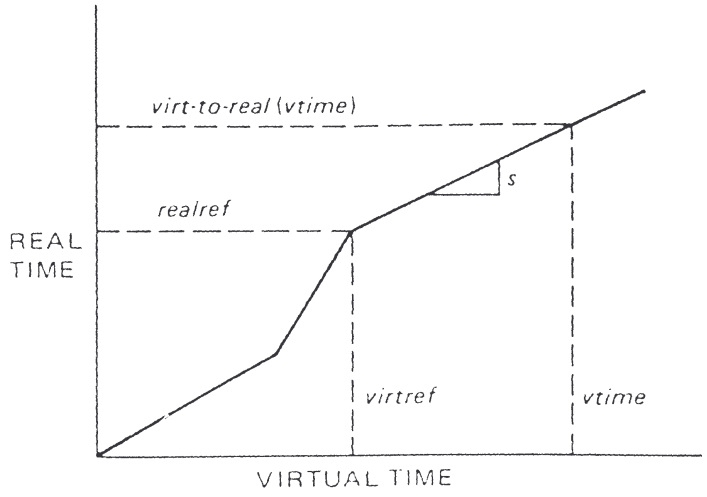


**Figure 18.8**
The virtual- to real-time calculation.

A new routine, *vschedule*, can now be written.

```
vschedule(id, vtime, i)
  time := virttoreal(vtime, i)
  if time <= gettime( )
    event(id)
  else if time < basetime + N
    table1[time - basetime] :=
      new(id, table1[time - basetime])
  else if time < basetime + (2 * N)
    table2[tme - (basetime + N)] :=
      new(id, table2[time - (basetime + N)])
  else pending1 := new(id, vtime, i, pending1)
```

This routine is similar to *schedule* except it takes an extra parameter ($i$) that specifies which virtual-time reference to use. If the event is expected to happen during the times represented by *table1* or *table2*, then the event is scheduled for a particular real time. Otherwise, the event is put on the pending queue to be handled by the background process, which is now presented:

```
background(starttime)
  mybase := starttime
  while TRUE
```

```
mybase := mybase + N
while mybase > basetime
   do nothing
while pending2 <> NULL
   i := pending2.I;
   insert(queue_i, pending2.ID, pending2.VTIME)
   pending2 := pending2.NEXT
for each i
   q := inspect(queue_i)
   time := virttoreal(q.VTIME, i)
   while time < mybase + 2 * N
      schedule(q.ID, time)
      queue_i := remove(queue_i)
      q := inspect(queue_i)
      time := virttoreal(q.VTIME, q.I)
if mybase <> basetime
   error( )
```

This code is based on that of Scheduler Implementation 6. Notice that the pending queue now specifies which time reference, and therefore which priority queue in which to insert the event. Notice also that now there can be more than one priority queue, so the background process must examine each one.

What has been accomplished with this new algorithm? In the beginning of this section, it was found that computation costs were proportional to the number of virtual-time references. This was true because each virtual time was updated at every unit of real time in order to schedule events properly. The new scheduler saves work by converting all virtual times to real times. There is still a computation cost proportional to the number of virtual-time references, but now this happens on each iteration of the background process rather than every unit of real time. Thus the new scheduler is much more efficient.

The disadvantage of this approach is that there is some latency between the time $s$ changes and the time at which this affects the real time of an event. Barry Vercoe has described programs with this property, presenting them as a model of human physiology [13]. The latency due to fixing performance times slightly in advance of real time is analogous to human reaction time.

To get an idea of the magnitude of the latency, a reasonable implementation might use real time units of 5 ms and tables of length 16. This would give a worst case latency of $5 \times 16 \times 2 = 160$ ms or 0.16 s. Note that this number reflects the worst-case delay before a new $s$ takes effect; if $s$ changes by some percentage $p$, then the maximum timing error will be roughly $(p/100) \times 0.16$ s. Also note that these numbers are arbitrary and there seems to be a wide range of reasonable choices. The main trade-off is that as the table gets smaller, latency goes down, but so does the amount of time available in the background process to handle a burst of events. Overall, this approach is interesting, but not very satisfying. It is fairly complex, yet an implementation is likely to suffer from too much latency or situations where the background process fails to make its deadline.

### 18.3.4 Yet Another Scheduler

Another implementation is worth considering. In this scheduler, there will be no latency except that due to the processor falling behind when there are many events to schedule or activate. This scheduler will not be as efficient as the previous one.

The idea is based on Implementation 4 in the previous section. Recall that Implementation 4 uses a priority queue for events and that it uses *setalarm* so that the scheduler does not work until it is time to activate the next event. Now imagine having one priority queue for each virtual-time reference and allowing each one to set an individual alarm. The alarm will be set with the anticipated real time of the next event. (Speed changes will be dealt with later.)

In practice, there may not be an individual hardware timer for each virtual time reference, but this is exactly the problem that schedulers solve! The virtual-time schedulers will use a single real-time scheduler to schedule themselves, and the real-time scheduler will use the hardware timer as always.

Assume that there is an Implementation 6 scheduler that implements the operation *schedule*($i$, *time*) that results in a call to *valarm*($i$) at the indicated time. The only change necessary to Implementation 6 is to replace the call to *event* with a call to *valarm*.

The virtual-time scheduler is presented below:

*vschedule*(*id*, *vtime*, $i$)
  $r :=$ *inspect*($requests_i$)

```
requests_i := insert(requests_i, id, vtime)
if vtime < r.VTIME
    schedule(i, virttoreal(vtime, i))
```

The object is to make sure that *valarm* will be called at the real time corresponding to the next item in the *requests* queue. If the virtual time of the new request (*vtime*) is less than the earliest time of any other event in *requests* (*r.vtime*), then *schedule* is called with the real time corresponding to *vtime*, the new earliest virtual time. The implementation of *valarm* detects and ignores any extra requests:

```
valarm(i)
    r := inspect(requests_i)
    vtime := r.VTIME
    while virttoreal(r.VTIME, i) <= gettime( )
        event(r.ID)
        requests := remove(requests_i)
        r := inspect(requests_i)
    if vtime <> r.VTIME
        schedule(i, virttoreal(r.VTIME, i))
```

Extra requests are detected by the while loop. If it is not yet time to perform the next event in the queue, nothing happens. If one or more events in the queue are performed, then *vtime* will no longer equal *r.vtime* and *valarm* finishes by scheduling the anticipated real time of the next event in the queue. To complete the implementation, here is *setspeed*:

```
setspeed(speed, i)
    s_i := speed
    r := inspect(requests_i)
    schedule(i, virttoreal(r.VTIME, i))
```

This scheduler is interesting because it always schedules *valarm* for the next known event time for each virtual-time reference. Since the real-time scheduler is so efficient, there is little overhead in scheduling extra *valarm* events. Notice that it is never a problem to schedule an extra *valarm* because *valarm* checks to see that it is time for an event before performing it. On the other hand it is important always to have at least one *valarm* scheduled for the real time of the next event, so each call to *setspeed* also schedules *valarm* for the new predicted time of the next event.

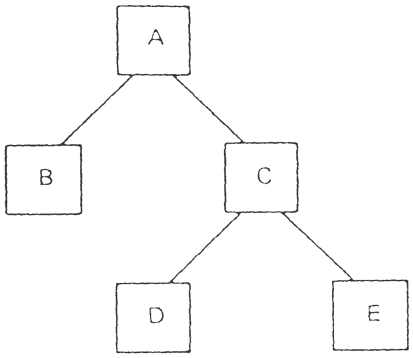


**Figure 18.9**
A hierarchy of virtual-time schedulers.

In this scheduler, a call to *vschedule* and a subsequent *valarm* has a cost proportional to the log of the number of events in the *request* queue. There is also an added cost of calling *schedule*, but this cost is essentially fixed except for potential background processing. Finally, each call to *setspeed* adds the cost of calling *schedule*.

### 18.3.5 Composition of Time Maps

After exploring real-time schedulers and virtual-time schedulers, there *is* still one area left to consider. A logical extension to the scheduler is one in which a virtual-time scheduler uses another virtual time as a reference. This allows arbitrary nesting of virtual-time references.

The implementation of such a system uses a tree of schedulers where the root is a real-time scheduler and other nodes are virtual-time schedulers. Figure 18.9 illustrates a tree of virtual-time references. References E and D use C as a time reference, and B and C use A (real time) as a reference. Each scheduler uses its parent to schedule an alarm when it is time for the next event to be activated. Thus, the parent scheduler determines the reference time for its children. It can be seen that the previous scheduler is a two-level tree with this structure.

In a multilevel tree, the cost of performing an event is the sum of the cost at each level. For example, in figure 18.9, the performance of an event scheduled with scheduler E begins with an event scheduled with A, the real-time scheduler. This first event invokes the *valarm* routine of scheduler C, which in turn runs the *valarm* routine of scheduler E. The *valarm* routine of E finally performs the scheduled event. The idea of scheduling alarms that

turn out to be useless because of a subsequent speed change is still applicable in a multilevel tree of schedulers.

To reduce the overhead of dispatching events that are many levels deep, it is possible to collapse any tree to two levels. For example, in figure 18.9, D and E would be moved to the level of B and C. To do this, it is necessary to modify *setspeed* to update all affected time references. For example, changing the speed of C must indirectly affect D and E. Finally, the *virttoreal* function must be modified to compute the correct composition of time maps. This approach makes dispatching faster at the expense of a more expensive *setspeed* operation.

Another extension of virtual-time schedulers is to consider using a continuously varying speed. For example, a smooth acceleration might be effected by changing the speed linearly instead of in steps. To incorporate this capability, the only change necessary is in the function *virttoreal*. The real time of a future event at virtual time $V_f$ will be the integral of the speed function $s(v)$ from the current virtual time $V_c$ to $V_f$. If $s(v)$ can be restricted to a polynomial, then the *virttoreal* function will consist of evaluating the integral of $s$, also a polynomial [9].

### 18.3.6 Related Work

The idea of nested virtual-time schedulers is not new, and implementations have been described by Jaffe and by Anderson and Kuivila. Jaffe's article [9] considers a non-real-time system oriented toward computer-aided composition. The system by Anderson and Kuivila [2] is a real-time implementation in which the delay of the next event must be specified as each event is scheduled. Executable processes can be used to specify speed changes, resulting in a very flexible means for specifying time maps.

The work described here makes two contributions to existing methods. First the real-time scheduling algorithm is a significant improvement over the use of heaps or other "fast" methods. Second, all of the virtual-time scheduling algorithms described in this chapter allow events to be scheduled in arbitrary order, and it is unnecessary to know the time of the next event, or whether the next scheduled event will be before or after the present one.

In fairness to the previous work, it is often the case that one knows when the next event will be scheduled, or at least one can often know the order of events. This leads to greater efficiency in the referenced works, insuring that there are no extra calls to *vtime*.

## 18.4 Computer Accompaniment

All of the schedulers presented so far deal only with time and events and could be used for, say, controlling machinery as well as music. In this section, a scheduler that assumes a musical context is examined. This assumption enables the scheduler to incorporate musical knowledge, resulting in a very sophisticated (but not very general) scheduler.

*Computer Accompaniment*[5] is a task similar to that of a human accompanist, who listens to another performer, reads music, and performs another part of the music in a synchronized fashion. Computer accompaniment does not involve any improvisation or composition. Instead, the computer is given a score containing a description of the music to be played by each performer. In this model, the *soloist* considers only his or her (or its!) part of the score and determines the tempo of the performance. The (computer) *accompanist* dynamically adjusts its timing to match that of the soloist.

Computer accompaniment has been accomplished by a computer system that can be divided roughly into two parts: the "analytical" part that uses a fast algorithm to perform pattern matching on a symbolic representation of the score and the solo, and the "musical" part that attempts to produce a sensible performance in the face of continual adjustments in tempo. Figure 18.10 separates these parts further into a number of separate tasks that are described below.

### 18.4.1 Analytical Tasks

The "analytical" part has the job of following the soloist in the score. This part consists of two concurrent tasks. The first task, called the *listener*, is a preprocessor of input from the soloist. Listening in this context means converting sound into a symbolic form to be used by the next task. A single melodic line from, say, a trumpet or flute is processed in real time to obtain time-varying pitch information, which is then quantized to obtain the discrete pitches of a musical scale. Alternatively, a music keyboard whose pitch output is inherently symbolic can be used. In either case, the listener task sends a schematic representation of the soloist's performance to the next task. The delay between the onset of a note and its detection by the listener must be small to achieve responsive accompaniment.

This second task, called the *matcher*, compares the actual performance to the expected performance as indicated in the score. The objective of the

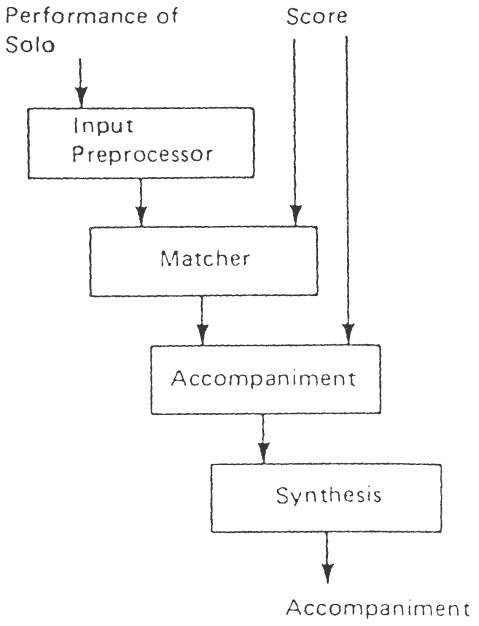


**Figure 18.10**
Block diagram of a computer accompaniment system.

comparison is to find a correspondence between the performance and the score, thereby relating real time to the timing indications in the score. In accompaniment, the goal is to construct a time map in real time in order to make the virtual times of score events take place at the real times of the performance events. Since either the soloist or the listener task may make mistakes (the listener task makes mistakes because pitch and attack detection, especially in a noisy acoustic environment, is inherently error prone), the matcher must be tolerant of missing notes, extra notes, or notes whose pitches are wrong. Furthermore, the timing of notes will vary from one performance to the next. To deal with this kind of "fuzzy" match, a real-time adaptation of dynamic programming is used. The output of the matcher is a sequence of reports that occur whenever the matcher is fairly certain that a note performed by the soloist corresponds to a particular note in the score. A matcher for monophonic inputs is described in [8] and two matchers for polyphonic inputs are described in [4].

### 18.4.2 Musical Tasks

The "musical" part contains a third task, called the *accompanist*, which controls the timing of the accompaniment. Note that most of the accom-

paniment is determined by the score, and timing is the only dimension that varies from one performance to the next. (Other parameters, such as loudness, could be varied as well.) Typically the accompanist will output commands that mean something like "the violin should now begin playing C-sharp," and a synthesizer handles the actual sound generation. The main problem in the accompaniment task is to adjust the timing of the accompanist in a musical fashion.

One approach to providing timing would be to use a virtual-time reference to schedule accompaniment events. Calls to *setspeed* could be used to keep the accompaniment in synchronization with the performance. Although this scheme would work reasonably well, it would be unable to handle jumps to new positions in the score in a musical way. A better approach is to write a scheduler that incorporates musical knowledge. The scheduler gets the clock speed and offset relative to real time from the matcher task. Adjustments must be made carefully if they are to sound musical, and a rule-based approach is used to program the accompanist. For example, one rule says that if the virtual clock is behind the soloist by a moderate amount, it is better to catch up by playing very fast than by skipping part of the accompaniment. Note that this is not as simple as the virtual-time schedulers seen in the previous section. In order to enhance the quality of the accompaniment, the accompanist can repeat parts of the score if virtual time jumps backward, skip parts of the score if virtual time jumps forward by large amounts, and generally alter timing according to the context of the score. The accompanist can be called a *knowledge-based* scheduler because it schedules events based on knowledge of musical interpretation as well as the passage of time.

### 18.4.3 Implementation

The implementation of the accompanist relies on an underlying real-time scheduler of the sort described in the first section. The overall structure of the accompanist is fairly simple: the accompanist calculates the real time at which the next accompaniment will occur, and an event is scheduled to reactivate the accompanist at that time. Normally, the accompanist will then suspend itself and when the proper time is reached, the accompanist will be reactivated. It then performs the next accompaniment event, and the cycle is repeated.

The accompanist is also reactivated when input arrives from the matcher. Recall that the matcher reports when the soloist plays a note in the score;

therefore, the current real time is known to correspond to the virtual time of the matched note in the score. The accompanist can now recalculate the speed of virtual time, the current virtual time, and the real time of the next accompaniment event. An event is scheduled to reactivate the accompanist at that time.

If the performance time of the matched note is not close to the expected time for that note, the accompanist must take a corrective action. A rule that matches the current situation will determine the action; for example, if the virtual time skips far ahead, the accompanist will stop any notes currently sounding and skip to a new location in the score.

Left to its own, the accompanist always has a single reactivation event scheduled while it is asleep. However, the matcher may intervene with some information that changes that accompanist's notion of when it must next do something. Rather than changing the time of the previously scheduled wakeup event, it schedules a new one. This is analogous to setting another alarm clock after deciding to wake up at a different time. Since the accompanist knows when it should wake up, it can ignore alarms that go off at other times.

### 18.4.4 Thinking Ahead

One consequence of the accompanist implementation is that the analytical score following tasks are only weakly coupled to the musical accompaniment tasks. This is as it should be, since an accompaniment performance should make a certain musical sense even without the solo it is meant to support. The partial independence allows the accompanist to continue its performance even in the absence of input from the soloist. This is essential for situations where the accompaniment has many notes against a sustained note or rest in the solo part.

Because of this partial independence, it is not true that the accompaniment system must lag behind the soloist. To avoid this potential problem, the latency of the system is determined experimentally and subtracted from the scheduled real time of every accompaniment event. Therefore, the accompaniment anticipates every event by an amount equal and opposite to the latency in the system, just as a tuba player must anticipate in order to compensate for the latency of his instrument. If the soloist and accompanist are to play two notes simultaneously, it is only after the soloist's note is processed that the accompanist can know whether its timing was correct. If not, the discrepancy is used to update the virtual-time

reference so that the next note will be timed more correctly. The resemblance to human performance is striking.

### 18.4.5 Related Work

Computer accompaniment systems have been developed independently by Dannenberg [4, 8], Vercoe [13, 14], and by Lifton [10]. The paper by Bloch and Dannenberg [4] discusses matchers for monophonic and polyphonic performances in detail. The paper by Vercoe and Puckette [14] describes the idea of learning performance timing through rehearsal, and Lifton's paper [10] describes a system for the accompaniment of vocal music.

## 18.5 Conclusions

Computer music instruments, accompaniment systems, and interactive composition systems are opening new doors for performers and composers. Schedulers are critical components in these systems, and this chapter has presented a collection of schedulers for computer music applications. Scheduler Implementation 6 is a particularly efficient real-time scheduler with a constant cost per scheduled event plus some background processing for events scheduled far into the future. Several virtual-time schedulers were also described. Finally the concept of a knowledge-based scheduler was explored in the context of computer accompaniment systems.

As real-time music systems increase in complexity, the idea of knowledge-based schedulers may very well evolve into the notion of expert performance systems, that is, computer programs that model the behavior of human performers. Composers who use these systems will enjoy the flexibility of computer-generated sounds, without giving up all of the advantages of having human performers play their works. Composers who write their own expert performance systems will be able to explore and develop new standards of performance practice tuned to their own personal musical goals.

## Acknowledgments

Most of this chapter grew out of conversations with Ron Kuivila in between concerts and lectures at the 1985 International Computer Music Conference and the Second STEIM Symposium on Interactive Composi-

tion in Live Electronic Music. It was during these conversations that the idea for a fast scheduler was born. Barly Truax, Simon Fraser University, Michel Waisvicz, and STEIM deserve special thanks for organizing these conferences, which generated such a sharing of ideas. John Maloney furnished many helpful comments and spotted some errors in an earlier draft. The author would also like to thank the Computer Science Department and the Center for Art and Technology at Carnegie-Mellon University for their support of this work.

## Notes

1. Moxc is available as part of the CMU MIDI Toolkit from the Center for Art and Technology, Carnegie-Mellon University, Pittsburgh, PA 15213.

2. The *requests* queue is initialized with a request whose time is infinity, so *inspect* will always return a value. It is assumed that *setalarm(infinity)* is defined to disable the alarm indefinitely.

3. Sound travels on the order of a foot per millisecond.

4. In an actual implementation, *table1* and *table2* would be blocks of memory accessed through pointer variables. The *swap* operation merely exchanges the contents of the pointer variables. Thus *swap* is very fast.

5. Computer Accompaniment is the subject of a pending patent.

## References

[1] Aho, Hopcroft, and Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] David P. Anderson and Ron Kuivila. A Model of Real-Time Computation for Computer Music. In *Proceedings of the 1986 International Computer Music Conference*, pp. 35–41. Computer Music Association, 1986.

[3] Jon Bentley. Programming Pearls. *Communications of the ACM* 28(3):245–250, 1985.

[4] Joshua J. Bloch and Roger B. Dannenberg. Real-Time Computer Accompaniment of Keyboard Performances. In *Proceedings of the 1985 International Computer Music Conference*, pp. 279–290. Computer Music Association, 1985.

[5] Xavier Chabot, Roger Dannenberg, and Georges Bloch. A Workstation In Live Performance: Composed Improvisation. In *Proceedings of the 1986 International Computer Music Conference*, pp. 57–60. Computer Music Association, 1986.

[6] Joel Chadabe. Interactive Composing: An Overview. *Computer Music Journal* 8(1):22–27, 1984.

[7] D. J. Collinge. MOXIE: A Language for Computer Music Performance. In *Proceedings of the 1984 ICMC*, pp. 217–220. Computer Music Association, 1984.

[8] Roger B. Dannenberg. An on-Line Algorithm for Real-Time Accompaniment. In *Proceedings of the 1984 International Computer Music Conference*, pp. 193–198. Computer Music Association, 1984.

[9] David Jaffe. Ensemble Timing in Computer Music. *Computer Music Journal* 9(4):38–48, 1985.

[10] John Lifton. Some Technical and Aesthetic Considerations in Software for Live Interactive Performance. In *Proceedings of the 1985 International Computer Music Conference*, pp. 303–306. Computer Music Association, 1985.

[11] Max V. Mathews and Curtis Abbott. The Sequential Drum. *Computer Music Journal* 4(4):45–59, 1980.

[12] G. Varghese and T. Lauck. Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, published as *Operating Systems Review* 21(5): 25–38, ACM Order No. 534870, 1987.

[13] Barry Vercoe. The Synthetic Performer in the Context of Live Performance. In *Proceedings of the 1984 International Computer Music Conference*, pp. 199–200. Computer Music Association, 1984.

[14] Barry Vercoe and Miller Puckette. Synthetic Rehearsal: Training the Synthetic Performer. In *Proceedings of the 1985 International Computer Music Conference*, pp. 275–278. Computer Music Association, 1985.