Chapitre 4

Ordonnancement pour les systèmes musicaux temps réel

4.1. Introduction

Les logiciels musicaux, en particulier les logiciels MIDI, sont souvent amenés à traiter et à produire des flots d'événements datés et organisés dans le temps. Malheureusement, l'ordre de production de ces événements n'est bien souvent pas celui souhaité pour leur restitution. C'est pourquoi il est utile de disposer d'un mécanisme permettant à l'application « d'envoyer des événements dans le futur » et se chargeant de les distribuer au bon moment.

Pour comprendre la nécessité d'un tel mécanisme, prenons l'exemple d'une application qui transforme chaque note reçue en une suite de « rebonds » à la manière d'une balle de ping pong. L'application reçoit un flot d'événements MIDI entrants, et pour chaque note reçue, génère la suite de rebonds qui lui correspond sous la forme d'une séquence de notes de même hauteur, dont le rythme va en s'accélérant pendant que l'intensité des notes diminue progressivement afin de simuler le comportement physique de la balle.

Produire ces rebonds directement dans l'ordre de restitution temporel est assez compliqué dès que les rebonds de plusieurs notes reçues s'entremêlent. Il faut garder la trace de tous les rebonds en cours de simulation et savoir en permanence la date du prochain rebond afin de produire la note correspondante le moment venu. Il serait plus

---

Chapitre rédigé par Yann ORLAREY.

simple de précalculer, à la réception de chaque note entrante, la séquence des rebonds qui lui correspond et de déléguer à un mécanisme spécifique la tâche de conserver les événements le temps nécessaire afin de les délivrer à leur date d'échéance pour le compte de l'application.

Le problème ressemble donc à un problème de tri, mais sur un ensemble d'événements qui évolue au fur et à mesure que de nouveaux événements sont placés dans l'ordonnanceur et que des événements arrivés à échéance en sont retirés. En fait, il se rapproche beaucoup du problème de la gestion des événements dans les logiciels de simulation. Du reste, de nombreuses techniques intéressantes ont été proposées dans le cadre des systèmes de simulation pour gérer l'ensemble des événements en attente (*pending event set*) [JON 86, RON 97].

Malheureusement, des contraintes et des besoins différents rendent les algorithmes utilisés en simulation peu applicables au domaine musical temps réel. D'une part, les systèmes de simulation doivent pouvoir déterminer rapidement le prochain événement à traiter alors que les systèmes musicaux peuvent se contenter de déterminer les événements qui arriveront à échéance à la prochaine unité de temps (car ils sont de toute façon appelés à chaque unité de temps). D'autre part, les systèmes musicaux ont des contraintes temps réel qui supposent l'absence de cas très défavorables et qui interdisent le recours à des opérations lourdes de réallocation mémoire ou de réorganisation des données, comme peuvent le faire les systèmes de simulation.

Dans un contexte musical temps réel, il est important que le coût d'ordonnancement d'un événement soit faible et surtout borné, et ce quelle que soit l'avance avec laquelle un événement est placé dans l'ordonnanceur (c'est-à-dire quel que soit son temps de séjour dans l'ordonnanceur) et quel que soit le nombre d'événements déjà en attente. Dans les paragraphes suivants, nous présentons un algorithme qui possède ces propriétés. Le coût d'ordonnancement d'un événement est en $O(1)$, tant en ce qui regarde l'avance avec laquelle il est ordonnancé que le nombre d'événements déjà en attente. Cet algorithme a initialement été développé dans le cadre du projet MIDILISP [BOY 86]. Il est utilisé depuis plusieurs années dans le système multitâche temps réel MidiShare [ORL 89, ORL 90].

4.2. Spécification du problème

Avant de présenter l'algorithme dans le détail, voyons plus précisément comment le problème se définit, quelles sont les opérations de base que l'on peut vouloir effectuer sur l'ordonnanceur, quels sont les critères de coût à retenir pour une utilisation temps réel, ainsi que l'utilisation typique que l'on peut en faire en prenant pour exemple le système MidiShare.

4.2.1. *Définitions*

A un instant donné, l'état d'un ordonnanceur $S = scheduler[E, D]$ est caractérisé par l'ensemble E des événements en attente et la date courante D. Chaque événement e inséré dans l'ordonnanceur est muni d'une *date d'échéance*, que nous noterons $d(e)$, et qui est la date à laquelle on veut qu'il soit retiré de l'ordonnanceur et distribué à son destinataire. Les dates sont représentées par des entiers non signés de 32 bits. Comme nous le verrons, le fait que les dates soient de taille fixe est important dans la définition de l'algorithme.

Nous appellerons *date de placement*, que nous noterons $p(e)$, la date à laquelle un événement est placé dans l'ordonnanceur, c'est-à-dire la date courante D de l'ordonnanceur au moment de l'insertion de l'événement e. L'*avance* avec laquelle un événement est placé dans l'ordonnanceur (également appelée *durée de séjour*) est la différence entre sa *date d'échéance* et sa *date de placement* : $d(e) - p(e)$. Dans le cadre d'un fonctionnement normal, les événements sont placés avec une *avance* positive : $d(e) > p(e)$. Le nombre d'événements en attente dans l'ordonnanceur $scheduler[E, D]$ à un instant donné est $|E|$.

4.2.2. *Opérations sur l'ordonnanceur*

Munis de ces quelques définitions, nous pouvons décrire plus précisément l'état initial de l'ordonnanceur ainsi que les deux opérations principales d'insertion d'un événement et de retrait des événements arrivés à échéance.

4.2.2.1. *Initialisation de l'ordonnanceur*

INITIALISATION$() \rightarrow scheduler[\emptyset, 0]$: à l'initialisation, l'ensemble des événements en attente est bien évidemment vide et la date courante de l'ordonnanceur est 0.

4.2.2.2. *Insertion d'un événement*

INSERTION$(e, scheduler[E, D]) \rightarrow scheduler[E + \{e\}, D]$: l'événement ordonnancé est ajouté à ceux déjà en attente. Pour que l'opération soit correcte, on suppose que l'événement est placé avec une avance positive, c'est-à-dire que $d(e) > D$.

4.2.2.3. *Avance de l'horloge et retrait des événements arrivés à échéance*

HORLOGERETRAIT$(scheduler[E, D]) \rightarrow scheduler[E - R, D + 1]$ and RETURN(R) avec $R = \{e | e \in E, d(e) \leqslant D + 1\}$: la date courante de l'ordonnanceur est avancée d'une unité et tous les événements arrivés à échéance sont retirés de l'ensemble des événements en attente et retournés.

4.2.3. *Coût d'ordonnancement*

Le coût d'ordonnancement d'un événement e, que nous noterons $C_o(e)$, se décompose en trois parties :

– $C_i(e)$, le coût d'insertion dans l'ordonnanceur,

– $C_m(e)$, le coût de maintient dans l'ordonnanceur jusqu'à l'échéance,

– $C_r(e)$, le coût de retrait de l'événement e de l'ordonnanceur.

Ainsi, on a $C_o(e) = C_i(e) + C_m(e) + C_r(e)$.

Dans le cadre d'une application temps réel, l'objectif sera non seulement de minimiser le coût total d'ordonnancement d'un événement, mais également de faire en sorte que ce coût soit indépendant du nombre $|E|$ d'événements en attente dans l'ordonnanceur ainsi que de l'avance $d(e) - p(e)$ avec laquelle l'événement est placé dans l'ordonnanceur. En d'autres termes, pour satisfaire aux contraintes d'une utilisation temps réel, ce coût doit être borné en toutes circonstances, en particulier lorsque la charge du système est élevée.

4.2.4. *Utilisation de l'ordonnanceur*

Le système MidiShare (voir chapitre ??) fournit un exemple typique d'utilisation d'un ordonnanceur dans un contexte multitâche temps réel. Les applications clientes de MidiShare forment un réseau de communication et, de ce point de vue, l'activité d'une application consiste à recevoir et à émettre des événements datés. Lorsqu'une application émet un événement, celui-ci est placé dans l'ordonnanceur jusqu'à sa date d'échéance avant d'être délivré aux applications destinataires.

Le premier aspect à prendre en considération dans une telle implémentation est la possibilité d'accès concurrent à l'ordonnanceur. En effet, plusieurs applications, éventuellement réparties sur des processeurs différents, peuvent avoir besoin des services de l'ordonnanceur au même moment. Plutôt que de protéger l'accès à l'ordonnanceur par des sémaphores ou d'autres techniques bloquantes, ce qui serait préjudiciable en termes de performances, la solution adoptée par MidiShare consiste à utiliser une liste LIFO intermédiaire dans laquelle sont tout d'abord placés les événements avant d'être insérés dans l'ordonnanceur. Les problèmes d'accès concurrents sont donc déportés sur cette liste intermédiaire. Mais une telle liste peut être implémentée de manière très efficace en utilisant des techniques non bloquantes (*lock-free*).

L'insertion proprement dite dans l'ordonnanceur est directement réalisée par la routine d'interruption *timer* qui cadence le système. Celle-ci commence par récupérer, par une opération atomique, la liste des événements à insérer. Ensuite, elle insère un à un chaque événement de cette liste dans l'ordonnanceur. Puis, elle appelle la méthode

HORLOGERETRAIT afin de récupérer les événements arrivés à échéance. Enfin, ces événements sont distribués aux applications destinataires.

L'enchaînement des opérations réalisées par la routine d'interruption *timer* peut être schématisé comme suit :

$$\text{TIMER=RECUPERATION.}(\text{INSERTION})*.\text{HORLOGERETRAIT.DISTRIBUTION}$$

4.3. L'algorithme d'ordonnancement

Il existe bien entendu de nombreuses façons de traiter le problème d'ordonnancement que nous venons de décrire. Une solution naïve consiste tout simplement à représenter l'ensemble E des événements en attente sous la forme d'une liste d'événements ordonnés dans le temps. Techniquement simple, cette solution ne répond pas aux critères d'efficacité nécessaires et implique un coût d'insertion $C_i(e)$ proportionnel au nombre d'événements en attente dans l'ordonnanceur. Elle n'est donc pas applicable dans des situations avec des contraintes temps réel et un grand nombre d'événements à traiter.

Il existe une solution encore plus simple (mais malheureusement inapplicable dans la pratique, pour des questions de place mémoire), extrêmement efficace en temps de calcul et qui consiste tout simplement à utiliser un tableau avec autant d'entrées que de dates possibles (2^{32} dans le cas présent). Chaque entrée contient la liste des événements en attente à cette date.

L'algorithme que nous allons présenter est une élaboration de cette idée mais nécessitant beaucoup moins de mémoire. Avant d'en faire une description formelle, nous allons étudier cet algorithme à travers l'histoire de *Monsieur Nay* et de la société *DO-IT*.

4.3.1. *L'histoire de Monsieur Nay et de la société DO-IT*

La société DO-IT est un peu curieuse. Son activité principale consiste à recevoir par courrier des ordres à effectuer à des dates précises. Récemment, par exemple, elle a reçu un ordre d'un monsieur âgé, qui pense ne plus être de ce monde d'ici quelques temps. Dans sa lettre, cette personne âgée demande à DO-IT de souhaiter, le 17 février 2018, un joyeux anniversaire à sa fille unique qui aura alors 50 ans.

Un des employés de cette société est chargé, tous les matins, de dépouiller le courrier et de classer toutes les demandes. En outre, il fournit aux autres employés la liste des tâches du jour. Il faut bien entendu qu'il ne perde pas trop de temps à classer les demandes qui arrivent et qu'il retrouve rapidement les demandes du jour. Plusieurs personnes se sont succédées à ce poste, sans beaucoup de réussite. Elles utilisaient

toutes un grand bac dans lequel les fiches étaient classées par ordre chronologique. Elles perdaient beaucoup de temps à parcourir le fichier pour classer les ordres nouvellement reçus. Cette mauvaise gestion a duré jusqu'à l'arrivée à ce poste de Monsieur Nay.

Monsieur Nay, homme organisé, a conçu un nouveau système de bacs de rangement. Il utilise 31 bacs pour les 31 jours du mois, 12 bacs pour les 12 mois de l'année et 100 bacs pour les 100 ans d'activité de la société. Le classement se fait très simplement. Si l'ordre est pour le mois en cours, il le met dans le bac du jour correspondant. Si l'ordre est pour l'année en cours, il le met dans le bac du mois correspondant. Sinon, il met l'ordre dans le bac de l'année correspondante.

Une fois son courrier classé, il n'a plus qu'à prendre les ordres qui se trouvent dans le bac du jour et à les donner aux autres employés. Evidemment, le 1 er du mois suivant, les 31 bacs sont tous vides. Il va donc reprendre tous les ordres du mois qui commence et les reclasser. Suivant le même principe, au début de chaque année, les bacs des jours et des mois sont vides. Notre employé doit donc reclasser tous les ordres de l'année qui commence dans les bacs des mois puis reclasser tous les ordres du mois de janvier dans les bacs des jours.

A ce stade, si nous analysons le coût de traitement d'un ordre, nous nous apercevons que celui-ci n'est manipulé au plus que trois fois, quelle que soit l'avance avec laquelle il est donné. Dans le pire des cas, il sera d'abord placé une fois dans le bac des années, puis une fois dans celui des mois, puis enfin dans celui des jours. De plus, ce coût est indépendant du nombre d'ordres déjà enregistrés.

Le coût de traitement d'un ordre est donc très faible. Par contre, il subsiste un problème qui est le volume de travail accumulé à chaque début de mois et surtout à chaque début d'année. Comme monsieur Nay est un homme astucieux, il a trouvé la solution : répartir son travail de reclassement tout au long de l'année. Pour cela, il utilise des bacs supplémentaires : un deuxième jeu de 31 bacs pour le reclassement du mois suivant et un deuxième jeu de 12 bacs pour le reclassement de l'année suivante.

Tous les jours, Monsieur Nay fait un peu de reclassement. Il prend quelques ordres de l'année suivante et les classe dans le deuxième jeu de 12 bacs. Il prend ensuite quelques ordres du mois suivant et les classe dans le deuxième jeu de 31 bacs. Au début du mois suivant, Monsieur Nay termine complètement le reclassement de ce nouveau mois. S'il a bien réparti son travail, il n'a que peu ou pas de reclassement à faire. Ensuite, il échange les deux jeux de 31 bacs. Suivant le même principe, en début d'année, il termine le reclassement de cette nouvelle année, échange les deux jeux de 12 bacs, puis termine le reclassement du mois de janvier et échange les deux jeux de 31 bacs.

L'analyse des coûts est la même que précédemment – un ordre n'est au plus manipulé que trois fois –, mais cette fois, la charge de travail totale est répartie plus uniformément. Bien évidemment, l'échange des bacs est, au niveau informatique, réduit à un simple échange de pointeurs.

4.3.2. *Implémentation de l'algorithme de Nay*

Voyons comment implémenter concrètement l'algorithme de Nay. Nous allons tout d'abord décrire les structures de données employées, puis les principales méthodes – en particulier l'insertion et le reclassement – dans un formalisme proche du C++.

4.3.2.1. *Structures de données*

L'ordonnanceur (`class sorter`) est organisé autour de quatre systèmes de rangements (`class storage`) dans lesquels les événements vont progresser jusqu'à leur date d'échéance. Chaque rangement dispose de deux zones de stockage (`struct zone`) : primaire et secondaire. Ces zones sont des tableaux de 256 entrées dans lesquels les événements sont organisés en préservant l'ordre d'arrivée (`class fifo`). Les événements (`struct event`) comportent, outre une date d'échéance, un lien de chaînage utilisé pour la constitution de listes chaînées.

4.3.2.1.1. Les événements

Pour les besoins de l'implémentation, un événement est une structure de données ayant essentiellement deux champs : un lien de chaînage et une date sur 32 bits non signée.

```
struct event {
    event* link;
    unsigned long date;
    ...
};
```

Le lien de chaînage sera utilisé pour réaliser des listes chaînées d'événements. Le champ date sur 32 bits indique la date d'échéance de l'événement. Nous aurons besoin d'accéder aux octets qui composent une date. Pour une date d, nous noterons PART$(d, 3)$ son octet de poids le plus fort jusqu'à PART$(d, 0)$ son octet de poids le plus faible, de sorte que $d =$ PART$(d, 3)$*$256^3 +$ PART$(d, 2)$*$256^2 +$ PART$(d, 1)$*$256 +$ PART$(d, 0)$.

4.3.2.1.2. L'ordonnanceur

L'ordonnanceur comporte, outre la date courante, quatre systèmes de bacs de rangement pour les événements en attente ainsi qu'un fifo pour les événements ordonnancés en retard (voir figure 4.1). Réunis, ils forment l'ensemble E des événements en attente dans l'ordonnanceur.

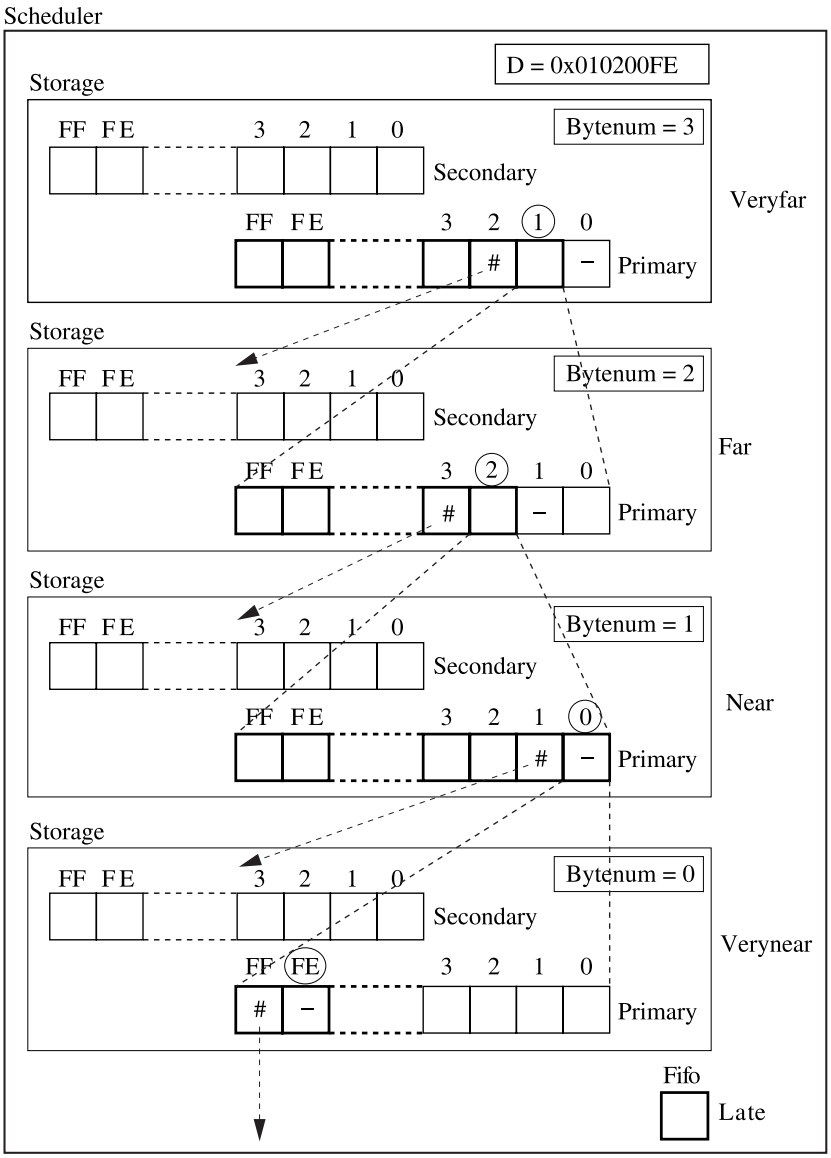

Figure 4.1. *L'ordonnanceur comporte une date courante* `D` *(ici 0x010200FE), quatre structures de rangement des événements :* `veryfar`, `far`, `near` *et* `verynear`, *ainsi qu'un fifo* `late` *pour les événements insérés en retard. Chaque rangement comporte deux zones de stockage :* `primary` *et* `secondary`, *une position temporelle courante* `period` *(matérialisée par un cercle), le fifo d'extraction* `extractable` *(matérialisé par #), ainsi que le numéro d'octet* `bytenum` *à sélectionner dans la date.*

Le principe de l'algorithme étant de maintenir les événements d'autant mieux classés qu'ils sont proches de leur date d'échéance, le rangement `veryfar` correspond aux événements les plus lointains, dont la date diffère de la date courante au niveau de l'octet de poids le plus fort, alors que le rangement `verynear` correspond aux événements presque arrivés à échéance et dont la date ne diffère de la date courante que sur l'octet de poids le plus faible.

```
class sorter {
    unsigned long D;
    storage veryfar;
    storage far;
    storage near;
    storage verynear;
    fifo late;
 public:
    sorter();
    unsigned long date();
    void insert (event* e);
    event* clock_extract();
};
```

Au cours de leur séjour dans l'ordonnanceur, les événements vont successivement passer de `veryfar` à `far`, puis de `far` à `near`, puis de `near` à `verynear`, pour enfin sortir à la date d'échéance.

L'ordonnanceur comporte deux opérations principales. La méthode `insert(e)` insère l'événement `e` dans le rangement et la case appropriés. La méthode `clock_extract()` incrémente la date de l'ordonnanceur, effectue tous les reclassements et retourne la liste des événements arrivés à échéance.

4.3.2.1.3. Les rangements

Comme indiqué au paragraphe précédent, l'ordonnanceur dispose de quatre systèmes de rangement qui contiennent les événements en attente, classés suivant l'éloignement plus ou moins grand de leur date d'échéance. Celle-ci est décomposée en quatre octets numérotés de 0 (octet de poids le plus faible) à 3 (octet de poids le plus fort). Les événements dont la date d'échéance diffère de la date courante sur l'octet 3 sont les plus éloignés et seront rangés dans `veryfar`. Ceux qui diffèrent sur l'octet 2 seront rangés dans `far` et ainsi de suite. Le champ `bytenum` de chaque rangement détermine le numéro de l'octet pris en considération par celui-ci.

On trouve également deux zones de rangement qui vont servir alternativement de zones de rangement primaire et secondaire. La zone secondaire est utilisée pour le travail progressif de reclassement des événements en provenance du

système immédiatement supérieur. Le champ `period` contient l'octet de la date courante déterminé par `bytenum`, de sorte que l'égalité suivante est toujours vérifiée : `D` = `veryfar.period`*256^3 + `far.period`*256^2 + `near.period`*256 + `verynear.period`. Enfin, le champ `extractable` indique le fifo d'où sont extraits les événements.

```
class storage {
    const int bytenum;
    zone    zn[2];
    buffer* primary;
    buffer* secondary;
    fifo*   extractable;
    int     period;
 public:
    storage(int n);
    bool tryput(event* e);
    event* extract(unsigned long cdate, int resort);
    void resort(event* l);
};
```

Les opérations principales sont :

– `tryput(e)` essaye d'insérer un événement dans le rangement. L'opération échoue si la date d'échéance de l'événement est trop proche pour être insérée à ce niveau, auquel cas il faut essayer de l'insérer au niveau inférieur ;

– `extract(date,resort)` extrait tout ou partie des événements extractibles et échange les zones de rangement si nécessaire ;

– `resort(list)` classe une liste d'événements dans la zone secondaire.

4.3.2.1.4. Les zones de rangements

Les zones de rangement sont constituées de 256 boîtes adressées donc sur un octet. Chaque boîte est un fifo afin de maintenir l'ordre de placement des événements.

```
struct zone {
    fifo box[256];
};
```

4.3.2.1.5. Les fifos

L'utilisation de fifos (*first in first out*) plutôt que de simples listes permet de préserver l'ordre d'insertion des événements dans l'ordonnanceur, en particulier lorsque ceux-ci comportent la même date d'échéance.

```
class fifo {
      event* head;
      event* tail;
      void Init();
    public:
      fifo();
      bool   Put(event* e);
      event* GetOne();
      event* GetSome(long n);
      event* GetAll();
      event* GetAllConcat(event* lst);
};
```

Le fifo fonctionne en maintenant deux pointeurs `head` et `tail` sur le premier et le dernier événement d'une liste chaînée. Cinq méthodes principales sont associées à un fifo :

– `Put(e)` permet d'ajouter un événement `e` en queue de fifo ;

– `Get()` permet d'extraire l'événement en tête du fifo ;

– `GetSome(n)` permet d'extraire sous forme de liste chaînée les `n` premiers événements en tête du fifo ;

– `GetAll()` permet d'extraire sous forme de liste chaînée tous les événements du fifo ;

– `GetAllConcat(list)` fonctionne comme `GetAll` mais concatène en plus la liste d'événements passée en argument en queue des événements du fifo (en d'autres termes, `fifo.GetAllConcat(l)` = `fifo.GetAll()++l`).

4.3.2.2. *Les méthodes d'ordonnancement*

4.3.2.2.1. L'insertion d'un événement

La méthode `insert()` insère un nouvel événement dans l'ordonnanceur en fonction de son avance.

```
void sorter insert (event* e)  {
    if (e->date <= D) {
        late.put(e);
    } else {
         veryfar.tryput(e)
        || far.tryput(e)
        || near.tryput(e)
        || verynear.tryput(e);
    }
}
```

Le premier cas traité est celui des événements e en retard ($d(e) \leqslant D$) qui sont placés dans le fifo `late` en attendant la prochaine opération d'extraction. Les événements e en avance ($d(e) > D$) sont, quant à eux, placés dans le rangement approprié en essayant successivement, grâce à la méthode `tryput()`, les rangements *veryfar*, *far*, *near* et *verynear*.

```
bool storage tryput(event* e)  {
    int i = part(e->date, bytenum);
    if (i > period) {
        primary->box[i].put(e);
        return true;
    } else {
        return false;
    }
}
```

Le placement dans `veryfar` réussit si l'octet de poids fort de la date de l'événement est supérieur à celui de la date courante : $\mathrm{part}(d(e), 3) > \mathrm{part}(D, 3)$. Ce même octet sert alors d'indice dans la zone primaire de rangement pour placer l'événement dans le fifo correspondant. Le placement échoue si $\mathrm{part}(d(e), 3) = \mathrm{part}(D, 3)$. Le rangement suivant est alors essayé en utilisant l'octet suivant de la date. A noter que l'une des quatre opérations est assurée de réussir puisque $d(e) > D$ implique qu'il existe $i \in \{3, 2, 1, 0\}$ tel que $\mathrm{part}(d(e), i) > \mathrm{part}(D, i)$.

4.3.2.2.2. L'extraction

La méthode `clock_extract()` incrémente la date de l'ordonnanceur, puis demande à `veryfar` de se mettre à jour et d'extraire des événements à propager dans le rangement suivant grâce à la méthode `extract_update()`. Les événements ainsi extraits sont ensuite distribués dans la zone secondaire de `far` par la méthode `resort()`. La même série d'opérations est accomplie pour les rangements `far`, `near` et `verynear`. Les événements extraits de `verynear` sont ceux arrivés à échéance. Ils sont combinés avec les événements ordonnancés en retard pour donner le résultat final.

```
event* sorter clock_extract()  {
  D = D+1;
  far.resort(veryfar.extract_update(D,RESORT));
  near.resort(far.extract_update(D,RESORT));
  verynear.resort(near.extract_update(D,RESORT));
  return late.getallconcat(verynear.extract_update(D,RESORT));
}
```

La méthode `resort()` reclasse les événements en provenance du niveau supérieur dans la zone secondaire par un adressage direct en fonction de l'octet de la date de l'événement qui lui correspond.

```
void storage resort(event* l)
{
    while (l) {
        event* e = l;
        l = l->link;
        secondary->box[ part(e->date,bytenum) ].put(e);
    }
}
```

La méthode `extract_update()` est plus complexe à cause du mécanisme de reclassement progressif mis en œuvre. La méthode prend deux paramètres : la (nouvelle) date courante de l'ordonnanceur et le nombre d'événements à extraire. La première étape consiste à déterminer s'il s'agit d'un changement de période ou pas. On compare pour cela l'octet concerné de la nouvelle date courante au champ `period`. Si l'on n'est pas arrivé au bout de la période, on se contente d'extraire le nombre souhaité d'événements. Si l'on est arrivé au bout de la période, il faut :

1) extraire tous les événements du fifo extractable afin qu'ils soient totalement reclassés dans le rangement inférieur ;

2) échanger les zones primaires et secondaires si l'on est passé de la période 255 à la période 0 ;

3) mettre à jour le champ extractable pour qu'il désigne le fifo correspondant à la période suivante.

Il est à noter, pour cette dernière opération, que si pour une période p le fifo correspondant à la période suivante est en général `primary->box[`p`+1]`, pour la période 255, il s'agit de `secondary->box[0]`.

```
event* storage extract_update(unsigned long cdate, int resort)
{
    int np = part(cdate,bytenum);
    if (np == period) {
        return extractable->getsome(resort);
    } else {
        event* r = extractable->getall();
        period = np;
        if (np == 255) {
            extractable = &secondary->box[0];
        } else {
```

```
                if (np == 0) {
                    zone* tmp = secondary;
                    secondary = primary;
                    primary = tmp;
                }
                extractable = &primary->box[np+1];
            }
            return r;
        }
    }
```

4.4. Performances

L'analyse du code précédent montre que l'insertion d'un événement représente au plus cinq comparaisons, quelques accès mémoire et un ajout en queue de fifo. Pendant son séjour dans l'ordonnanceur, un événement est, dans le cas le plus défavorable, déplacé trois fois avant de sortir de l'ordonnanceur, soit trois ajouts en queue de fifo. Toutes ces opérations sont indépendantes du nombre d'événements déjà en attente. Le coût maximal d'ordonnancement d'un événement C_o est donc en $O(1)$ quelle que soit la durée de séjour S de l'événement et le nombre N d'événements en attente dans l'ordonnanceur.

4.4.1. *Contexte des mesures*

Pour vérifier expérimentalement ce modèle, nous avons effectué une série de mesures sur un portable Dell C640, disposant d'un Pentium 4 de 1,8 GHz et 256 Mo de RAM, fonctionnant sous Linux Redhat 9 avec un noyau 2.4.20. Les programmes ont été compilés avec GCC 3.2.2 avec l'option d'optimisation -O3. Les mesures ont été réalisées en utilisant la fonction POSIX `clock()` qui renvoie la durée écoulée d'utilisation du processeur par le programme, exprimée en microsecondes. Les mesures portent sur le temps de calcul mis par l'ordonnanceur pour fonctionner pendant T unités de temps, avec une densité moyenne de W événements entrant et sortant à chaque unité de temps et pour une durée moyenne de séjour de $S/2$ unités de temps.

A l'initialisation du test, $N = W * S$ événements sont placés dans l'ordonnanceur avec des dates distribuées aléatoirement entre 1 et $S - 1$ unités de temps. Puis, à chaque unité de temps, et pendant T unités de temps, l'ordonnanceur est appelé afin de récupérer les événements arrivés à échéance qui sont immédiatement réordonnancés de sorte qu'il y ait en permanence N événements dans l'ordonnanceur. Le nombre total d'événements ordonnancés pendant toute la durée du test est $P = W * T$.

A titre de comparaison, nous avons également mesuré les performances d'un autre ordonnanceur basé sur l'algorithme dit du « calendrier » [BRO 88]. Le calendrier est ici un tableau découpé en *jours* et correspondant à une *année* d'activité. Lorsqu'un événement est ordonnancé, il est placé dans la case du jour souhaité indépendamment de son année. Lorsque l'on extrait les événements arrivés à échéance, on prend soin de ne sélectionner que les événements de l'année en cours et de laisser les autres en place. Simple de mise en œuvre, cet algorithme est très efficace lorsque les événements séjournent en général moins d'une année, mais ses performances se dégradent pour des durées supérieures. Pour les tests, nous avons utilisé une année de 2 048 jours afin que les deux algorithmes aient des occupations identiques en mémoire.

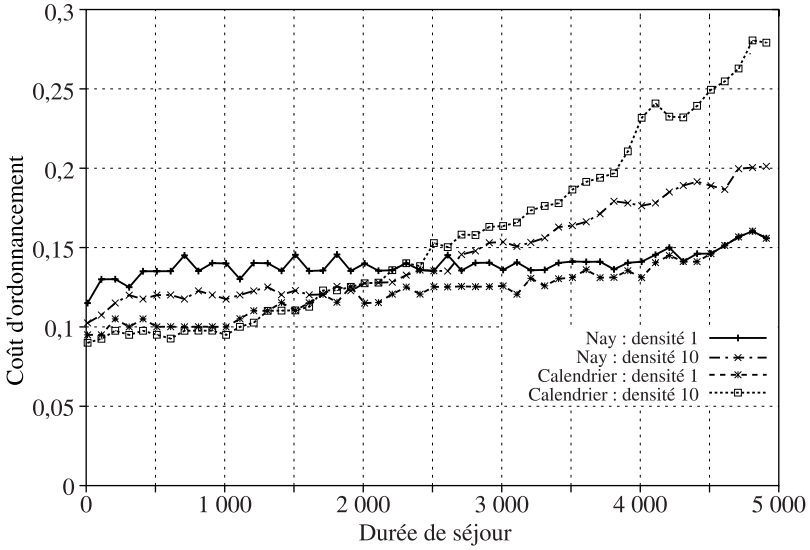

Figure 4.2. *Coût d'ordonnancement d'un événement pour des séjours courts, d'une durée maximale de 0 à 5 000 unités de temps*

4.4.2. *Performances pour des séjours courts*

La figure 4.2 montre l'évolution du coût d'ordonnancement d'un événement (en microsecondes) en fonction du temps de séjour maximal S variant entre 0 et 5 000 unités de temps (soit un temps moyen de séjour variant entre 0 et 2 500 unités de temps). Les mesures ont été effectuées pour une période de fonctionnement $T = 400\,000$. Chaque ordonnanceur est testé pour des densités de 5 et 10 événements par unité de temps. Sur l'ensemble de la période de test, cela représente respectivement 2 000 000

et 4 000 000 événements ordonnancés. Le coût porté en ordonnée est obtenu en divisant la durée du test par le nombre total d'événements ordonnancés. Il est donc légèrement supérieur au coût réel dans la mesure où il intègre une portion du coût fixe liée à l'appel de l'ordonnanceur à vide.

Le graphique montre que l'algorithme du calendrier est légèrement plus efficace pour des séjours qui tiennent dans l'*année* avec $0{,}092\,\mu s$ contre $0{,}107\,\mu s$ pour l'algorithme de Nay.

La tendance s'inverse lorsque la durée maximale de séjours commence à dépasser les 2 500 unités de temps (c'est-à-dire pour des durées moyennes de 1 250 unités de temps).

4.4.3. *Performances pour des séjours moyens*

Sur des séjours plus longs allant jusqu'à 50 000 unités de temps, les mesures montrent des coûts stables conformes au modèle pour l'algorithme de Nay, alors qu'elles montrent, pour l'algorithme du calendrier, des coûts multipliés par 60 par rapport aux cas les plus favorables (voir figure 4.3).

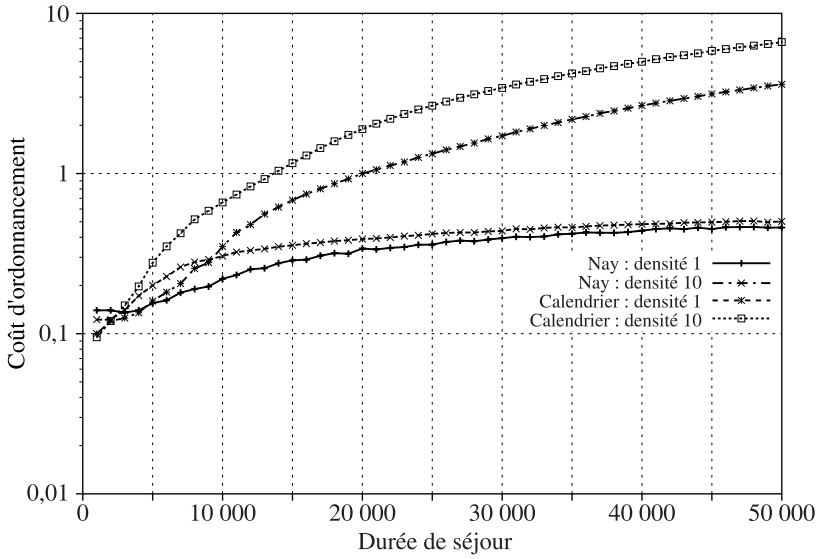

Figure 4.3. *Coût d'ordonnancement d'un événement pour des séjours allant jusqu'à 50 000 unités de temps*

4.4.4. *Performances pour des séjours longs*

Sur de longs séjours, allant jusqu'à 200 000 unités, on mesure la stabilité du coût d'ordonnancement pour l'algorithme de Nay, alors que les coûts pour celui du calendrier sont multipliés par un facteur 100 par rapport aux cas les plus favorables (voir figure 4.4).

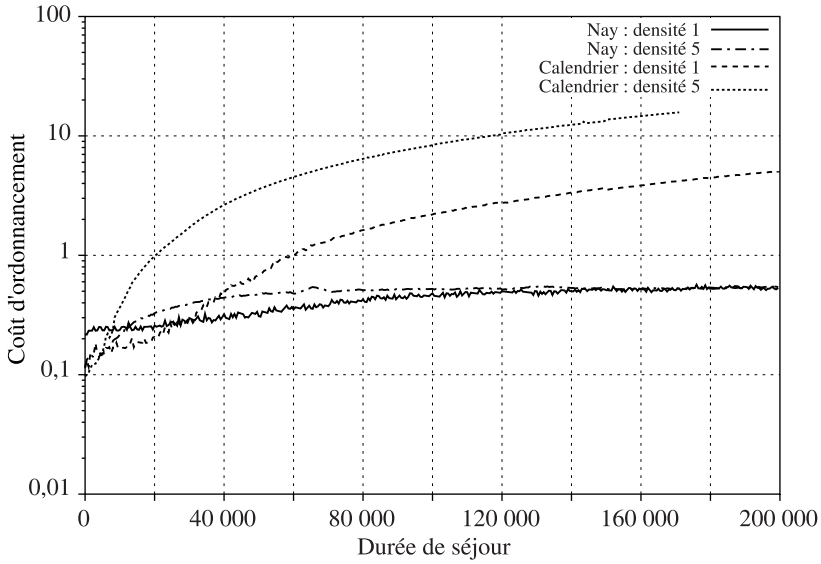

Figure 4.4. *Coût d'ordonnancement d'un événement pour de longs séjours allant jusqu'à 200 000 unités de temps*

4.5. Conclusion

Nous avons présenté un algorithme d'ordonnancement d'événements adapté à une utilisation temps réel grâce à un coût de traitement borné, indépendamment du nombre d'événements en attente et de l'avance avec laquelle les événements sont ordonnancés. Cet algorithme exploite le fait que les dates des événements peuvent être convenablement représentées par des entiers sur 32 bits et que l'ordonnanceur est appelé à chaque unité de temps.

Son principe de fonctionnement est de classer les événements d'autant mieux qu'ils sont proches de leur date d'échéance. Dans la version présentée, les dates des événements sont découpées en quatre octets et un événement est classé au plus quatre fois, par de simples adressages de tableau, avant d'arriver à échéance. D'autres choix

de découpage sont évidemment possibles suivant la nature du problème, les besoins en vitesse et la place mémoire disponible. Comme souvent en informatique, il y a un monnayage entre la vitesse d'exécution et la place mémoire utilisée. Moins il y a de niveaux, plus l'algorithme est efficace mais plus la place mémoire nécessaire est grande. Le cas extrême étant, bien entendu, l'emploi d'un seul niveau.

4.6. Bibliographie

[BOY 86] BOYNTON L., LAVOIE P., ORLAREY Y., RUEDA C., WESSEL D., « MIDI-LISP: A Lisp based music programming environment for the Macintosh », dans *Proceedings of the International Computer Music Conference*, ICMA, 1986.

[BRO 88] BROWN R., « Calendar queues: A fast 0(1) priority queue implementation for the simulation event set problem », *Communications of the ACM*, vol. 31, n° 10, p. 1220-1227, 1988.

[JON 86] JONES D.W., « An empirical comparison of priority-queue and event-set implementations », *Communications of the ACM*, vol. 29, n° 4, p. 300-311, 1986.

[ORL 89] ORLAREY Y., LEQUAY H., « MidiShare: A real time multi-tasks software module for MIDI applications », dans *Proceedings of the International Computer Music Conference*, ICMA, p. 234-237, 1989.

[ORL 90] ORLAREY Y., « An efficient scheduling algorithm for real-time musical systems », dans *Proceedings of the International Computer Music Conference*, ICMA, p. 194-198, 1990.

[RON 97] RÖNNGREN R., AYANI R., « A comparative study of parallel and sequential priority queue algorithms », *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 7, n° 2, p. 157-209, 1997.