Санкт-Петербургский государственный университет,
Математико-Механический факультет, кафедра системного программирования

# Оптимизация вычислений за счет эффективных структур данных в ОС Embox

курсовая работа студента 445 группы
Мальковского Николая Владимировича

науч. рук.: аспирант кафедры СП Бондарев Антон Владимирович

## Введение.

Целью моей курсовой работы является оптимизация работы ядра операционной системы Embox за счет добавления в нее дополнительных структур данных. В частности мною была предложена структура данных, реализующая планировщик событий - составляющая ядра операционной системы, отвечающий за обработку событий. Далее будет описан сам алгоритм, проведен анализ его эффективности, сравнение с другими алгоритмами (в частности с используемыми в Embox).

## Постановка задачи.

Задача возникла из необходимости какой-либо эффективной структуры данных обработки событий в операционных системах. Чаще всего встречаются периодичные события - таймеры(например таймер, отсчитывающий время, когда должна произойти передача управления другому потоку). В предложенном алгоритме будут рассматриваться только единичные события(в этом представлении таймеры - последовательность единичных событий). Однако для предлагаемой мной структуры не требуется, чтобы она была именно “структурой данных для обработки событий в ОС”, поэтому постановка задачи абстрагируется от понятия операционной системы. Таким образом задача будет формулироваться в виде: “Реализовать структуру данных, поддерживающую такие-то операции”.

Теперь формализуем понятие планировщика (далее будет использоваться именно это определение планировщика). Замечу, что следующее определение абстрагирует нас от понятия времени.

**Опр.** Планировщиком событий некоторая структура данных, поддерживающая две операции:

- **Tick()** - операция сама по себе ничего не делающая - является своеобразным счетчиком. Далее, я буду называть эту операцию *декрементацией*
- **Insert(n, handler)** - добавить в структуру событие, которое должно быть обработано через **n** вызовов **Tick** с помощью функции **handler**. Далее я буду называть *интервалом события* число операций декрементаций, через которое событие произойдет(с каждым **Tick**’ом оно будет уменьшаться)

Предполагается, что мы имеем дело с дискретной моделью времени, в которой оно разделено на

некоторые кванты одинакового размера. Таким образом **Tick** соответствует переходу к следующему кванту(вызывается аппаратным таймером). Об этом, разумеется, следует помнить, однако для формального описания алгоритма это замечание не нужно.

Итак, теперь наконец можно достаточно просто сформулировать задачу - реализация вышеуказанной структуры данных.

## Существующие решения.

На момент написания отсчета, все известные мне решения являются вариацией различных очередей с приоритетом.

Во-первых, два простых алгоритма со списком. В первом алгоритме все события хранятся в одном списке, добавление события - обычное добавление в список, декрементация - цикл по списку. Таким образом получается быстрое добавление, но медленные декрементация и удаление.

Во втором алгоритме все события хранятся в отсортированным по времени списку. Таким образом удаление всегда с головы списка, а декрементацию можно делать ленивой - декрментируется только голова списка.

Во-вторых, алгоритм, основанный на биномиальной куче. Операции добавления/удаления выполняются за логарифм от общего числа элементов в куче. Декрементация опять же ленивая - декрементируется только голова, при добавлении и удалении информация о декрементации распространяется на промежуточные вершины. Далее представлена таблица, показывающая временные оценки для операций этих алгоритмов.

| | Добавление в худшем случае | Добавление в среднем | Декр. в худшем случае | Декр. в среднем | Удаление |
|---|---|---|---|---|---|
| Алгоритм со списком | O(1) | O(1) | O(N) | O(N) | O(N) |
| Улучшенный алгоритм со списком | O(N) | O(N) | O(1) | O(1) | O(1) |
| Алгоритм с биномиальной кучей | O(LogN) | O(LogN) | O(1) | O(1) | O(LogN) |

* N - количество событий, которые зарегистрированы в структуре на момент выполнения операции

На данный момент в операционной системе Embox используется второй алгоритм. Хоть он и является довольно эффективным и простым в реализации, остается проблема с медленным добавлением, которая особенно проявляет себя при большой нагрузке на систему с большим количеством периодичных событий(таймеров).

Также следует отметить, что во всех трех алгоритмах время обработки одного событие(добавление, удаление и все его декрементации) зависит от общей загрузки системы.

Далее будет описан алгоритм, у которого этот недостаток отсутствует (что ведет к оптимальности по затратам вычислительных мощностей с асимптотической точки зрения), однако это приводит, к некоторым ограничениям и дополнительными затратами памяти.

**Простой алгоритм.**

Для начала, начнем с того, что ограничим сверху возможные значения интервала события

некоторым числом L(таким образом он может принимать любые целые значения из отрезка [1;L]). Структура будет состоять из одного массива размера L и указателя на некоторый элемент этого массива.

```
struct scheduler {
     list arr[L];
     int cur;
}
```

Работу структуры достаточно явно отражает следующий инвариант: **в любой момент времени все событий с интервалом K хранятся в списке** `arr[(cur+K) mod L]`. Таким образом операция декрементации получается очень простой - увеличиваем указатель `cur` на единицу по модулю L(все ячейки массива сдвигаются влево на единицу относительно указателя), при этом отправляются на обработку все события, находящиеся в списке, на который указывает `cur` после этой декрементации. Так же из этого инварианта естественным образом вводится добавление - новое событие добавляется в начало списка `arr[(cur+time) mod L]`. Далее соответствующий рисунок.

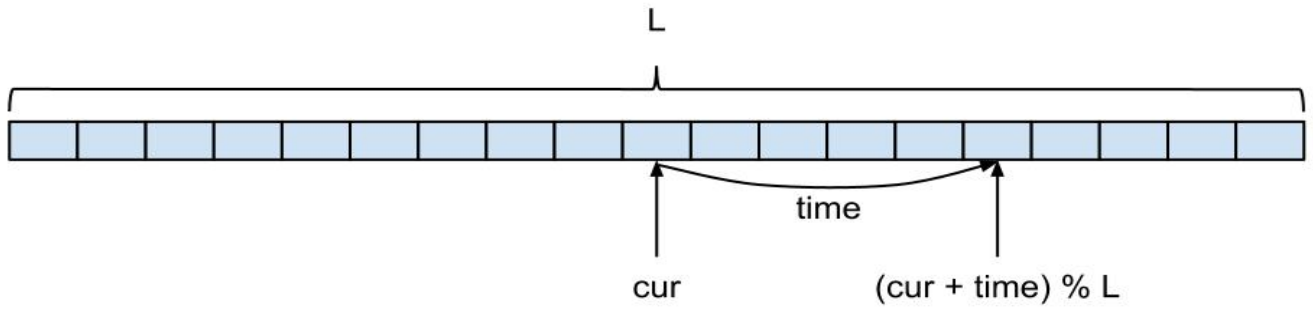


Рис. 1

Таким образом декрементация заключается в увеличении `cur`, т.е. O(1). Удаление событий из очереди происходит по группам, при этом удаляется каждый раз весь список, поэтому, вообще говоря, удалять можно просто обнуляя голову списка, но на практике нам нужно по этому списку пройтись, чтобы отправить на обработку каждое событие. Отсюда, для удаления списка из K событий делается за O(K) действий, для одного события - O(1) соответственно. Добавление - обращение к определенному элементу массива и добавление в начало списка, т.е. O(1). Исходя из этого можно заключить, что этот алгоритм является **асимптотически** оптимальным по времени работы(затратам процессорного времени), хоть это и не значит, что он будет эффективней рассмотренных ранее алгоритмов во всех случаях. Важно отметить, что обработка N событий за K единиц времени занимает O(N+K) действий, что и характеризует оптимальность(быстрее нельзя, так как это размер входных данных).

Далее приведена абстрактная реализация функций этого алгоритма.

```
Insert(integer n, handler h) begin
      insert h into arr[(cur + n) mod L]
end

Tick() begin
      cur ←(cur + 1) mod L
      while arr[cur] is not empty begin
            call arr[cur].head
            remove arr[cur].head
      end
end
```

**Основной алгоритм.**

Теперь хотелось бы как-нибудь улучшить алгоритм, чтобы сократить расходы памяти. Основная идея заключается в применении подхода так называемой корневой эвристики.

Пусть опять интервалы событий находятся в пределах [1;L]. Рассмотрим K - некоторый делитель L. Разобьем все события на два вида - события, интервалы которых меньше K и все остальные. Очевидное соображение - если зафиксировать какой-то момент времени, то все события второго вида(на этот момент времени) будут обработаны не раньше, чем через K декрементаций. События первого вида будем обрабатывать также, как и в предыдущем алгоритме. События второго уровня будет обрабатывать только один раз в K декрементаций(за K декрементаций некоторые события второго уровня стали событиями первого уровня). Оказывается, для событий второго уровня можно построить такую же структуру, как и для первого уровня.

Таким образом, структура будет выглядеть слудющим образом(далее соответствующий рисунок):

```
struct Scheduler2 {
     int inner[1..K];
     int outer[1..L/K];
     int cur_1;
     int cur_2;
}
```

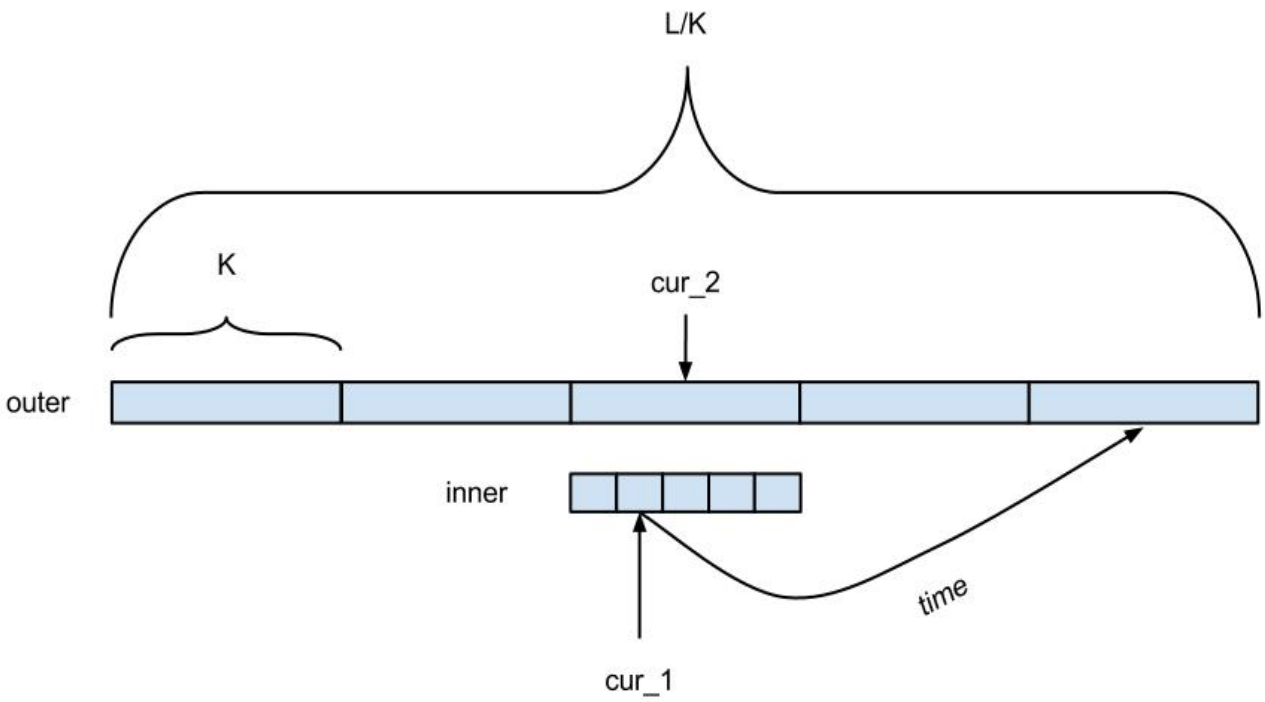


Рис. 2

Итак, есть массив inner содержащий события первого типа и массив outer, содержащий события второго типа. С событиями первого типа мы уже разобрались - обрабатываем как и в прошлом алгоритме. События второго типа будем постепенно переносить из массива outer в массив inner.

Напомню, что в любой момент времени интервал события второго типа на этот момент времени хотя бы K, тогда можно обрабатывать события второго типа **раз в K** декрементаций. А именно, те события, интервалы которых стали меньше K перенесем в массив inner. Учитывая структуру outer очевидно, что все события, которые нужно переносить при очередной K-ой декременации находятся в ячейке cur_2. При переносе этих событий, проходим по всему списку и каждое событие кладем в нужную ячейку массива inner(для этого нужно запомнить первоначальный интервал события или же его остаток по модулю K).

Далее - абстрактная реализация.

```
Insert(integer n, handler h) begin
      if(n strictly less than K) begin
            insert h into inner[(cur_1 + n) mod K]
      end else begin
            insert pair (h, (cur_1 + n) mod K) into
                        outer[(cur_2 + (cur_1 + n)/K) mod (L/K)]
      end
end

Tick() begin
      cur_1 ← (cur_1 + 1) mod K
      if(cur_1 == 0) begin
            cur_2 ← (cur_2 + 1) mod (L/K)
            for each pair (n, h) in outer[cur_2]
                  insert h into inner[n]
            clear outer[cur_2]
      end
      for each handler h in inner[cur_1]
            call h
      clear inner[cur_1]
end
```

## Эффективность алгоритма.

Напрямую из описания можно заключить следующие факты:

- Добавление события происходит за O(1)
- Удаление - O(1)
- Каждая K-ая декрементация - O(N), все остальные - O(1)

К сожаления мне так и не удалось избавиться от неоднозначности декрементации, но даже в этом случае можно утверждать, что расходы(по времени) на обработку событий меньше, чем для описанных в начале отчета алгоритмах.

Рассмотрим какое-то конкретное событие и посчитаем, сколько операций тратится на его обработку.

Каждое событие не более одного раза:

- Попадает во внешний массив(outer)
- Попадает во внутренний массив(inner)
- Перемещается из внешнего во внутренний
- Удаляется из структуры

Напоминаю, что удаление и перемещение работает таким образом, что обработка N событий занимает O(N) операций. Таким образом на одно событие уходит O(1) времени. На добавление и во внутренний и во внешний массивы также тратится O(1) времени.

Кроме этих четырех операций остается только увеличение cur_1 и cur_2, то есть на каждую декрементацию приходятся еще 1-2 присваивания. Отсюда и вытекает оценка, что обработка N событий за M декрементаций занимает O(N+M) операций.

Утверждается, что время обработки асимптотически оптимально, так как это, фактически, размер входных данных.

К сожаления есть небольшая проблема, связанная с декрементацией - каждую K-ую декрементацию происходит копирование, что может нагрузить системы. Фактически - это единственное возможное место перегрузки системы, которое создает сама структура.

Далее будет рассмотрен способ борьбы с этой проблемой.

**Амортизация.**

Основной идее здесь является нечто на подобии буферизации и предподсчета. Предлагается распределить нагрузку, которая попадает на каждую K-ую деркрементацию, равномерно по другим декрементациям(Рис. 3 и Рис. 4). Первое, что мы сделаем - увеличим массив inner в 2 раза. События

с интервалом меньше 2K теперь тоже будут попадать сразу в inner, но перемещаться из outer в inner по-прежнему будут события с интервалами меньше K. Теперь, рассмотрим момент времени, когда произошла декрементация в массиве outer. Утверждается, что к этому моменту **известны все события, которые будут перенесены при следующей декрементации outer** - они находятся в ячейке, следующей за cur_2. Важно то, что это множество событий не изменится, так как состоит из событий с интервалами из [K;2K), новые события с такими интервалами будут сразу помещены в inner.

Таким образом, если мощность этого множества - M, то к следующей декрементации outer мы должны перенести L событий, при этом известно, что всего будет M декрементаций. Значит, на каждую декрементацию приходится по M/K событий. То есть теперь мы будет при каждой декрементации переносить события из ближайшей ячейки массива outer в массив inner. После каждой декрементации outer M определяется заново.

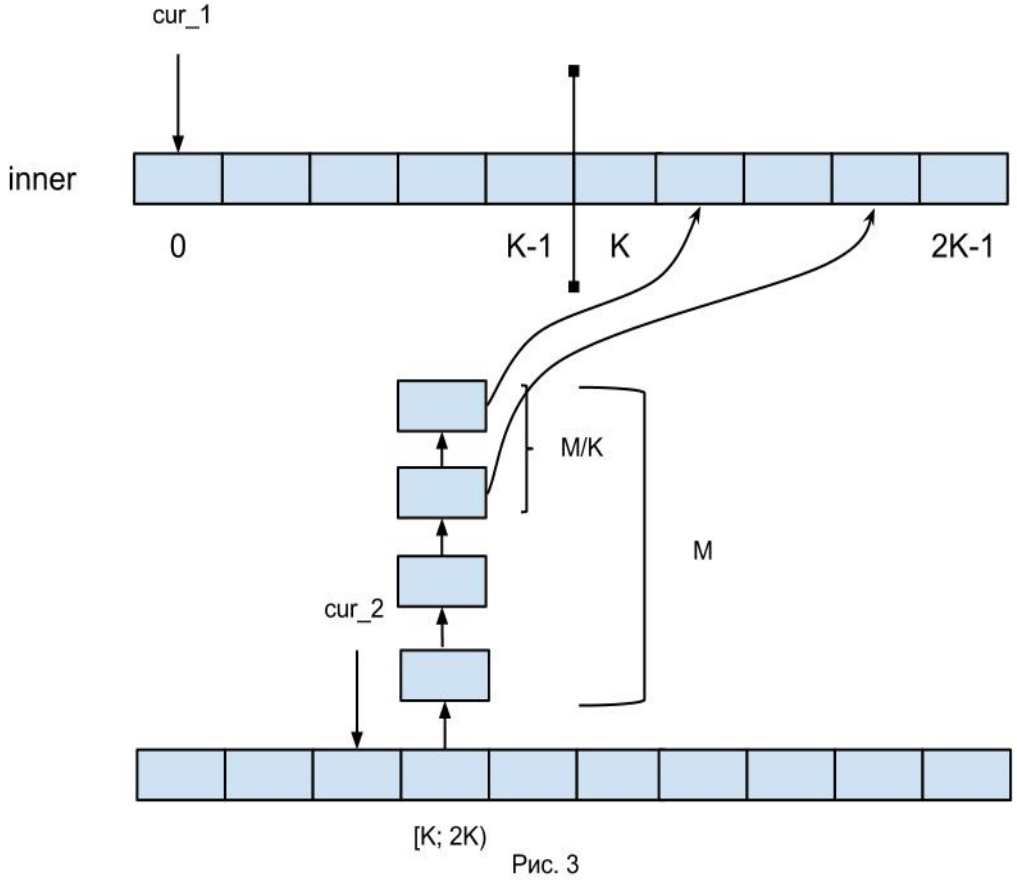

Рис. 3

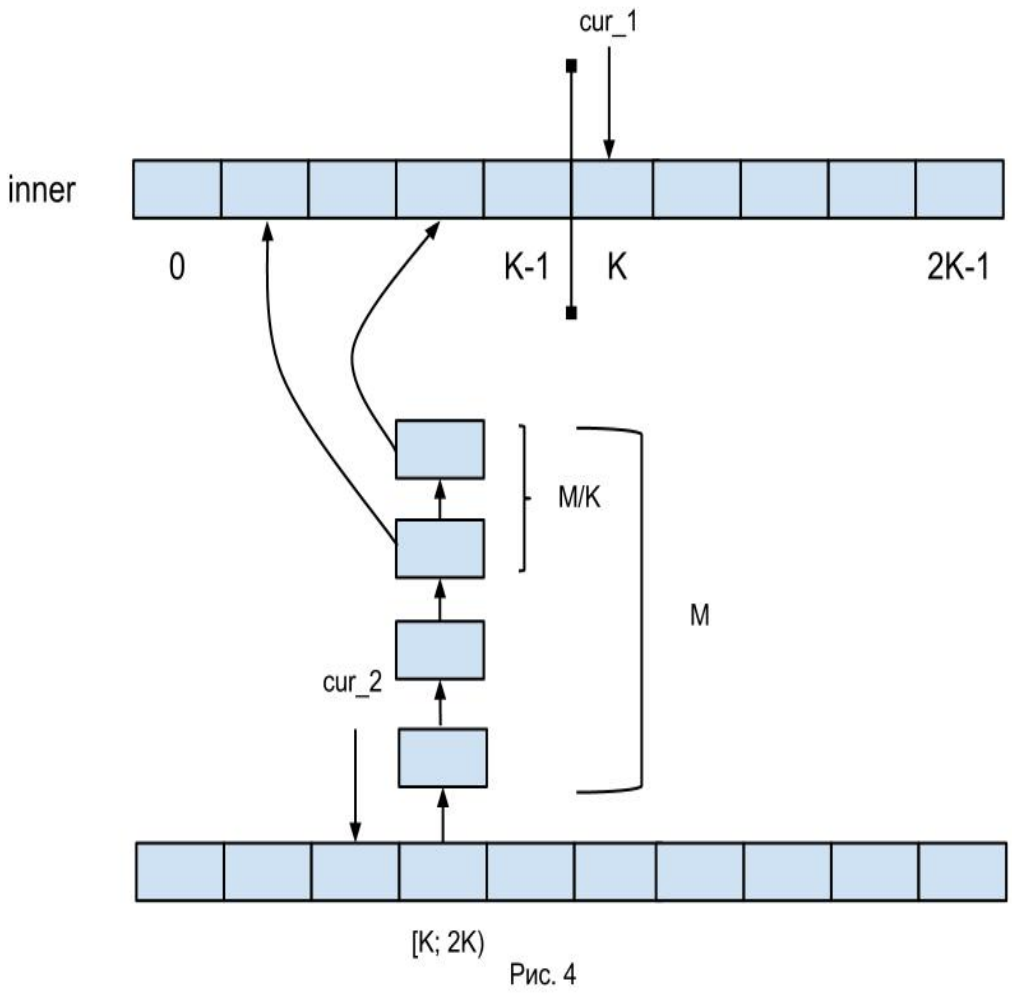

Рис. 4

Далее описано, как изменятся соответствующие операции при амортизации.

```
Insert(integer n, handler h) begin
     if(n strictly less than 2K) begin
          insert h into inner[(cur_1 + n) mod 2K]
     end else begin
          insert pair (h, (cur_1 + n) mod K) into
                    outer[(cur_2 + (cur_1 + n)/K) mod (L/K)]
     end
```

```
end
```

```
//Initially T equals to K

Tick() begin
      cur_1 ← (cur_1 + 1) mod K
      if(cur_1 == 0) begin
            cur_2 ← (cur_2 + 1) mod (L/K)
      M ← size of outer[(cur_2 + 1) mod (L/K)]
      T  ← K - T
            clear outer[cur_2]
      end

      repeat M/K times begin
            transfer h from first pair (n, h) from outer[(cur_2+1) mod (L/K)]]
                  to inner[T + n]
      end

      for each handler h in inner[cur_1]
            call h
      clear inner[cur_1]
end
```

**О выборе K.**

Последнее, что хотелось бы отметить - это выбор константы K(размер массива inner). Сейчас уже не столь важно, что от выбора константы зависит возможное кол-во операций

на обработку события(при увеличении K больше событий будут сразу попадать в inner). Сейчас главное, что от выбора K зависит размер потребляемой памяти, например при K=L мы получаем простой алгоритм с большим раходом памяти. Размер памяти в зависимости от K описывается простой функцией. Не составляет и труда найти ее минимум.

$$Space(K) = K + \frac{L}{K}$$

$$Space'(K) = 1 - \frac{L}{K^2}$$

$$Space'(K) \leq 0, K \in [0;\sqrt{L}]$$

$$Space'(K) \geq 0, K \in [\sqrt{L};L]$$

$$Space(\sqrt{L}) = 2\sqrt{L}$$

Если считать, что декрементация происходит каждую 1мс, то нескольких пары сотни байт хватает, чтобы обрабатывать события в пределах пары минут, 1Кб - час, сотня Кб - год.

## Результаты

Данный алгоритм планирования событий был реализован в ОСРВ Embox, что позволило улучшить характеристики обработки прерываний и сделать их более предсказуемыми и детерминированными, что является важным для операционных систем реального времени. Код можно посмотреть по адресу ***http***://embox.googlecode.com/svn/trunk/embox/ . Мой аккаунт на googlecode - malkovskynv@gmail.com.