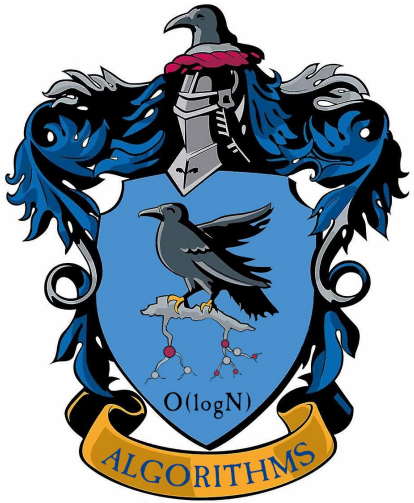
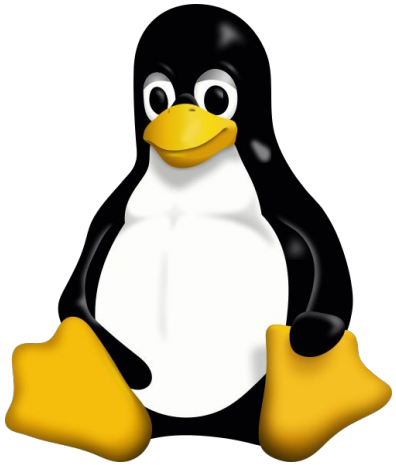
Алгоритмы + OSE

Планирование: от FCFS до EEVDF



+



Задача планирования

Вводные: знаем, что такое процесс операционной системы, имеем низкоуровневые примитивы для работы с процессами, в том числе переключение контекста.

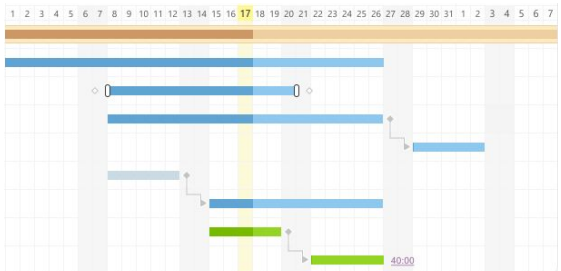
Задача планирования

Вводные: знаем, что такое процесс операционной системы, имеем низкоуровневые примитивы для работы с процессами, в том числе переключение контекста. Компонента операционной системы, выделяющая процессам время на CPU, называется планировщиком.

Задача планирования

Вводные: знаем, что такое процесс операционной системы, имеем низкоуровневые примитивы для работы с процессами, в том числе переключение контекста. Компонента операционной системы, выделяющая процессам время на CPU, называется планировщиком.

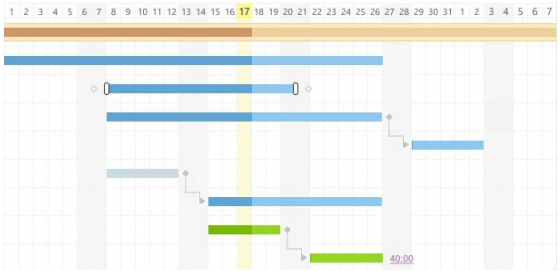
Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.



Задача планирования

Вводные: знаем, что такое процесс операционной системы, имеем низкоуровневые примитивы для работы с процессами, в том числе переключение контекста. Компонента операционной системы, выделяющая процессам время на CPU, называется планировщиком.

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

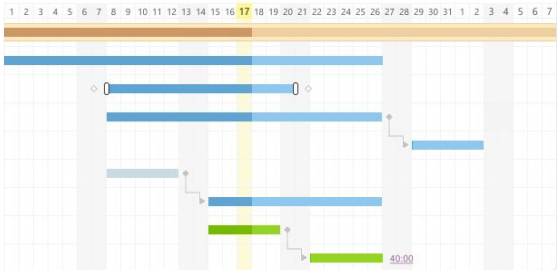


На самом деле не так важно, что именно мы планируем: процессы, потоки, произвольные таски

Задача планирования

Вводные: знаем, что такое процесс операционной системы, имеем низкоуровневые примитивы для работы с процессами, в том числе переключение контекста. Компонента операционной системы, выделяющая процессам время на CPU, называется планировщиком.

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.



На самом деле не так важно, что именно мы планируем: процессы, потоки, произвольные таски, но иногда будем делать поправку на OS-специфику

Задача планирования

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Задача планирования

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Начнем с более частной задачи, введя ограничения на то, что планируем, а потом начнем приближаться к реальности, отказываясь от введенных ограничений.

Задача планирования

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. Время выполнения всех заданий одинаково;
2. Все задания поступают в одно и то же время;

Задача планирования

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. Время выполнения всех заданий одинаково;
2. Все задания поступают в одно и то же время;

3. Задания дорабатывают до конца непрерывно;
4. Задания используют только CPU (никакого IO);

Задача планирования

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. Время выполнения всех заданий одинаково;
2. Все задания поступают в одно и то же время;

3. Задания дорабатывают до конца непрерывно;
4. Задания используют только CPU (никакого IO);

5. Время работы каждого задания известно заранее.

Задача планирования

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. Время выполнения всех заданий одинаково;
2. Все задания поступают в одно и то же время;

3. Задания дорабатывают до конца непрерывно;
4. Задания используют только CPU (никакого IO);

5. Время работы каждого задания известно заранее.

Такая постановка задача имела практический смысл до интерактивных систем, а для нас – хороший старт

Задача планирования

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

А что будем оптимизировать?

Задача планирования

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Метрики планирования:

1) Для любой задачи k введем оборотное время (как время окончания минус время поступления):

$$T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$$

Задача планирования

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Метрики планирования:

1) Для любой задачи k введем оборотное время (как время окончания минус время поступления):

$$T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$$

Будем оптимизировать среднее оборотное время: $T_{turnaround} = \frac{\sum_k T^k_{turnaround}}{N}$

Задача планирования

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Метрики планирования:

1) Среднее оборотное время; $T_{turnaround} = T_{completion} - T_{arrival}$

О чем еще стоит подумать?

Задача планирования

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Метрики планирования:

1) Среднее оборотное время; $T_{turnaround} = T_{completion} - T_{arrival}$

2) Справедливость планирования: задачи получают равное или пропорциональное количество процессорного времени.

Задача планирования

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Метрики планирования:

1) Среднее оборотное время; $T_{turnaround} = T_{completion} - T_{arrival}$

2) Справедливость планирования: задачи получают равное или пропорциональное количество процессорного времени.

Формальное определение,
например, через индекс Джейна: $\mathcal{J}(x_1, x_2, \ldots, x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot \sum_{i=1}^{n} {x_i}^2}$

Задача планирования

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Метрики планирования:

1) Среднее оборотное время; $T_{turnaround} = T_{completion} - T_{arrival}$

2) Справедливость планирования: задачи получают равное или пропорциональное количество процессорного времени.

С практической точки зрения нам важно отсутствие голодания процессов (чтобы не было ситуации, когда процесс вообще не получает процессорного времени).

Задача планирования

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Метрики планирования:

1) Среднее оборотное время; $T_{turnaround} = T_{completion} - T_{arrival}$

2) Справедливость планирования

3) ??? -> добавим позже

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. Время выполнения всех заданий одинаково;
2. Все задания поступают в одно и то же время;
3. Задания дорабатывают до конца непрерывно;
4. Задания используют только CPU (никакого IO);
5. Время работы каждого задания известно заранее.

Метрики:

1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$
2. Справедливость;

Предложите политику?

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. Время выполнения всех заданий одинаково;
2. Все задания поступают в одно и то же время;
3. Задания дорабатывают до конца непрерывно;
4. Задания используют только CPU (никакого IO);
5. Время работы каждого задания известно заранее.

Метрики:

1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - \cancel{T^k_{arrival}}$
2. Справедливость;

Предложите политику?

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. Время выполнения всех заданий одинаково;
2. Все задания поступают в одно и то же время;
3. Задания дорабатывают до конца непрерывно;
4. Задания используют только CPU (никакого IO);
5. Время работы каждого задания известно заранее.

Метрики:

1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - \cancel{T^k_{arrival}}$
2. Справедливость;

Предложите политику?

В наших ограничения вариантов не очень много, выбираем в любом порядке задания и выполняем. Например, в порядке FIFO (FCFS – first* come, first served)

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. Время выполнения всех заданий одинаково;
2. Все задания поступают в одно и то же время;
3. Задания дорабатывают до конца непрерывно;
4. Задания используют только CPU (никакого IO);
5. Время работы каждого задания известно заранее.

*почти в одно и то же время

Метрики:

1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - \cancel{T^k_{arrival}}$
2. Справедливость;

Предложите политику?

В наших ограничения вариантов не очень много, выбираем в любом порядке задания и выполняем. Например, в порядке FIFO (FCFS – first* come, first served)

FCFS

Время

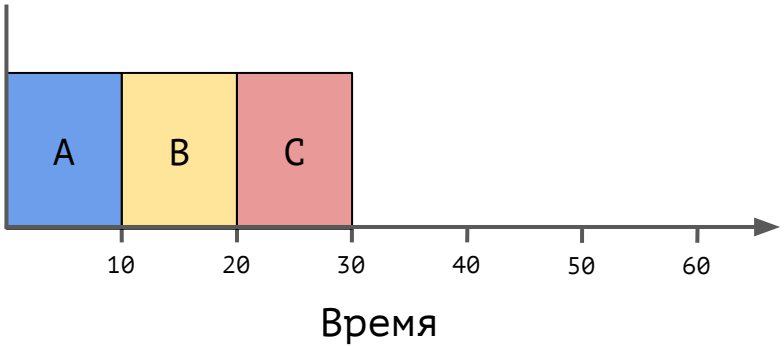
Поступают задачи A,B,C, каждая продолжительностью 10 миллисекунд

FCFS

Поступают задачи A,B,C, каждая продолжительностью 10 миллисекунд

допустим, именно в таком порядке (но приблизительно в одно время)

FCFS



Поступают задачи A,B,C, каждая продолжительностью 10 миллисекунд

допустим, именно в таком порядке (но приблизительно в одно время)

По FCFS берем в работу просто все подряд, получаем среднее оборотное время =

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. Время выполнения всех заданий одинаково;
2. Все задания поступают в одно и то же время;
3. Задания дорабатывают до конца непрерывно;
4. Задания используют только CPU (никакого IO);
5. Время работы каждого задания известно заранее.

*почти в одно и то же время

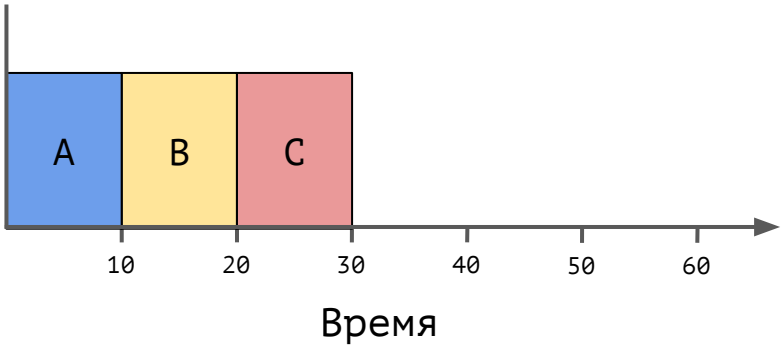
Метрики:

1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - \cancel{T^k_{arrival}}$
2. Справедливость;

Предложите политику?

В наших ограничения вариантов не очень много, выбираем в любом порядке задания и выполняем. Например, в порядке FIFO (FCFS - first* come, first served)

FCFS

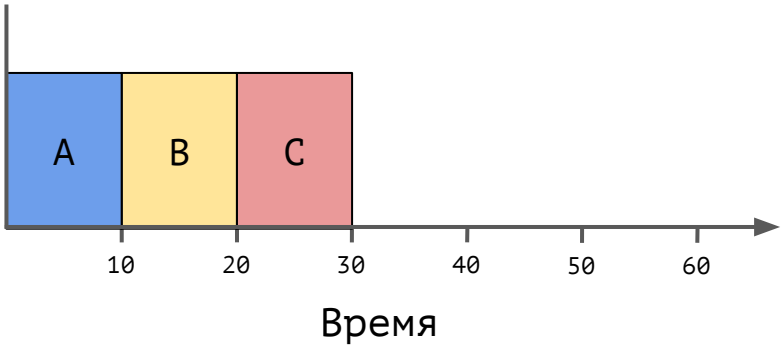


Поступают задачи A,B,C, каждая продолжительностью 10 миллисекунд

допустим, именно в таком порядке (но приблизительно в одно время)

По FCFS берем в работу просто все подряд, получаем среднее оборотное время = (10 + 20 + 30) / 3 = 20

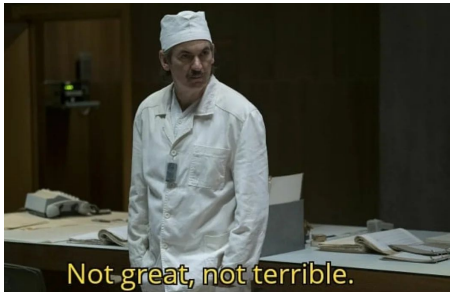
FCFS



Поступают задачи A,B,C, каждая продолжительностью 10 миллисекунд

допустим, именно в таком порядке (но приблизительно в одно время)

По FCFS берем в работу просто все подряд, получаем среднее оборотное время = (10 + 20 + 30) / 3 = 20



Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. Время выполнения всех заданий одинаково;
2. Все задания поступают в одно и то же время; ← *почти в одно и то же время
3. Задания дорабатывают до конца непрерывно;
4. Задания используют только CPU (никакого IO);
5. Время работы каждого задания известно заранее.

Метрики:

1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - \cancel{T^k_{arrival}}$
2. Справедливость;

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~
2. Все задания поступают в одно и то же время; ← *почти в одно и то же время
3. Задания дорабатывают до конца непрерывно;
4. Задания используют только CPU (никакого IO);
5. Время работы каждого задания известно заранее.

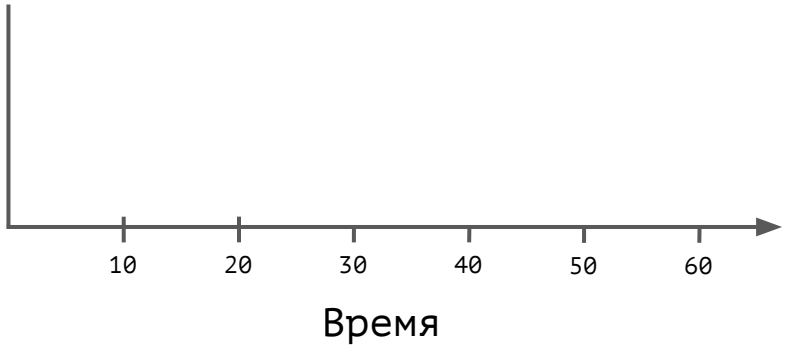
Метрики:

1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - \cancel{T^k_{arrival}}$
2. Справедливость;

Что изменится?

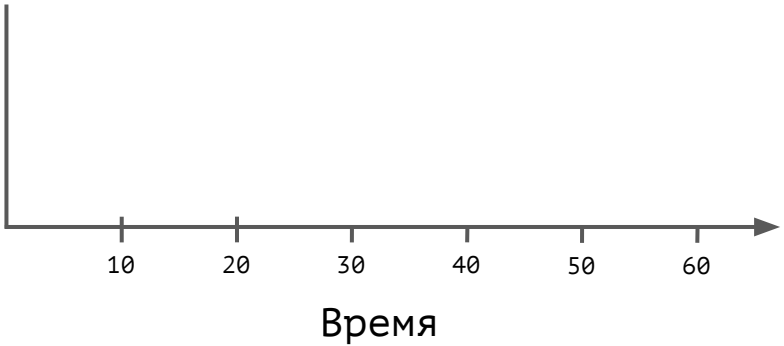
Постройте контрпример к FCFS?

FCFS



Пусть сначала поступает задача A, длительностью **30** ms.

FCFS

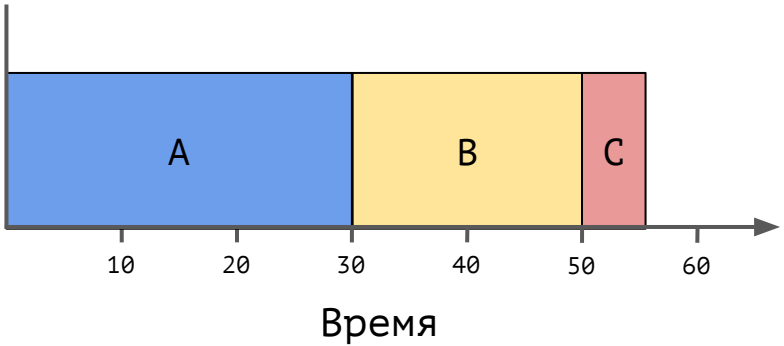


Пусть сначала поступает задача A, длительностью 30 ms.

Чуть погодя, задача B - длительностью 20 ms.

Наконец, задача C, длительностью 10 ms.

FCFS



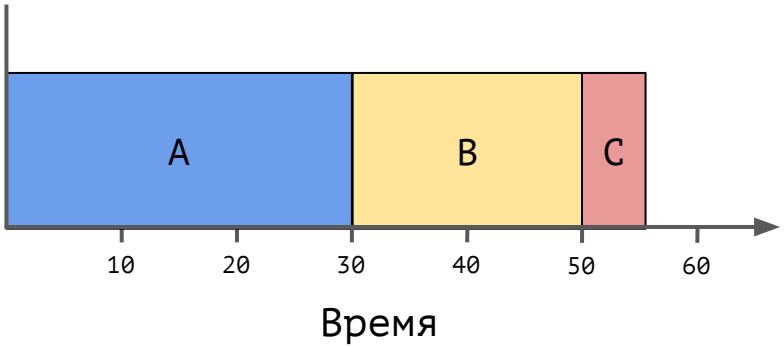
Пусть сначала поступает задача A, длительностью 30 ms.

Чуть погодя, задача B - длительностью 20 ms.

Наконец, задача C, длительностью 5 ms.

По FCFS всех по порядку, получаем среднее оборотное время =

FCFS



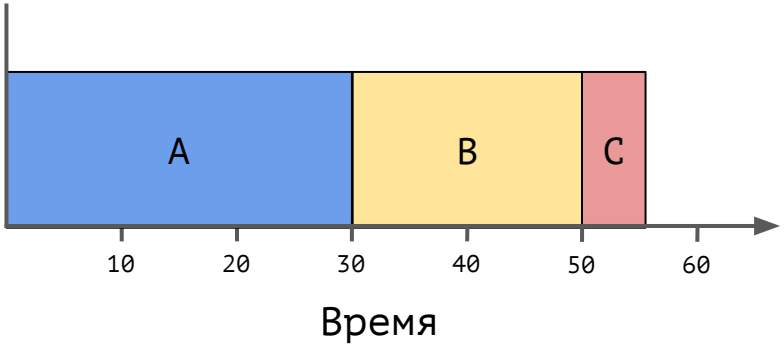
Пусть сначала поступает задача A, длительностью 30 ms.

Чуть погодя, задача B - длительностью 20 ms.

Наконец, задача C, длительностью 5 ms.

По FCFS всех по порядку, получаем среднее оборотное время = (30 + 50 + 55) / 3 = 45

FCFS



Пусть сначала поступает задача A, длительностью 30 ms.

Чуть погодя, задача B - длительностью 20 ms.

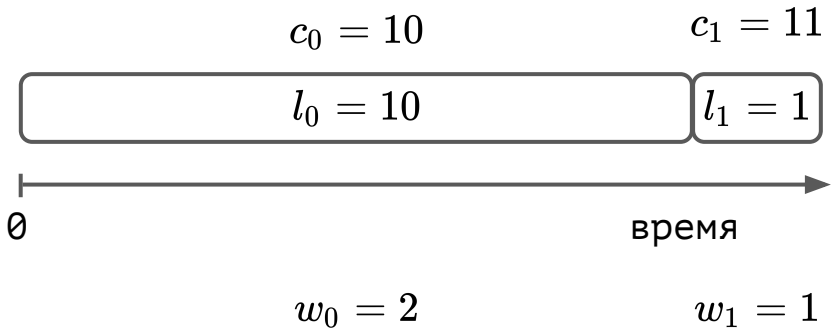
Наконец, задача C, длительностью 5 ms.

По FCFS всех по порядку, получаем среднее оборотное время = (30 + 50 + 55) / 3 = 45

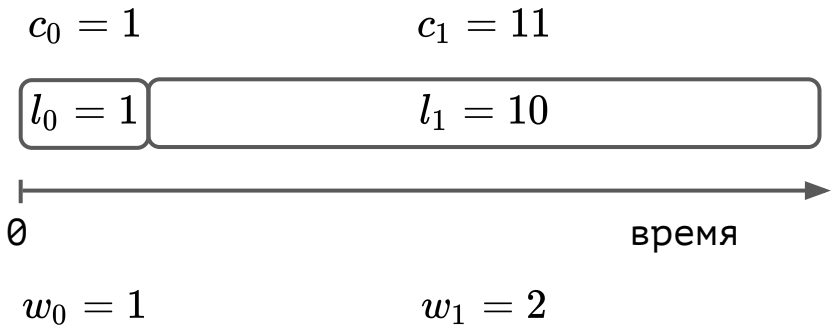
А как лучше?

Ничего не напоминает?

Задача планирования: пример



$$\sum c_j * w_j = 20 + 11 = 31$$



$$\sum c_j * w_j = 1 + 22 = \underline{23}$$

Всегда ли достаточно ставить вперед наивысший приоритет? Нет.

Задача планирования: пример

Рассмотрим два частных случая:

1. Пусть все задания одной длительности

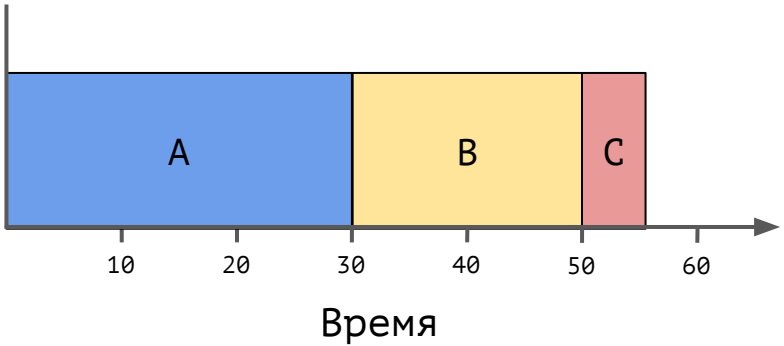
   (тогда с более высоким приоритетом исполняем раньше)

2. Пусть у всех заданий одинаковый приоритет, но разная длительность

   (тогда более короткие исполняем раньше)

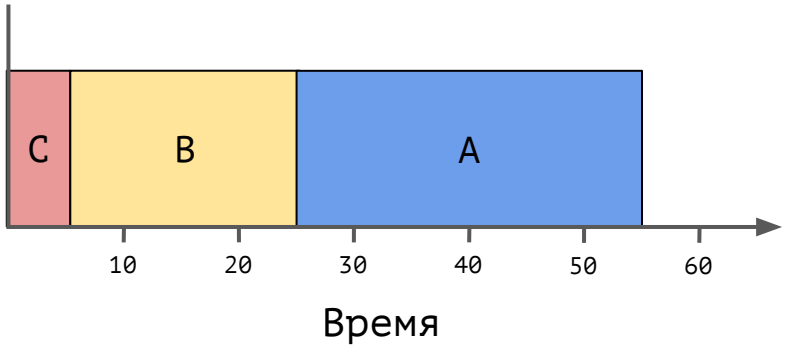
Значит нужно для каждого задания получить метрику, которая растет при росте приоритета и уменьшается при росте длины.

FCFS



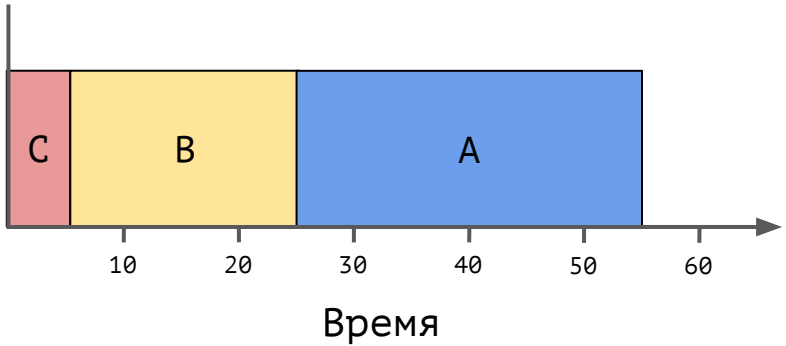
Попробуем поставить вперед более короткие задачи, т.е. порядок:
C -> B -> A

FCFS



Попробуем поставить вперед более короткие задачи, т.е. порядок:
C -> B -> A

SJF



Попробуем поставить вперед более короткие задачи, т.е. порядок: C -> B -> A

Тогда среднее оборотное время = (5 + 25 + 55) / 3 = 28.3

Куда лучше, чем FCFS. Такая политика называется SJF (shortest job first), и она является оптимальной для указанной постановки задачи (док-во как для любого жадника)

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~
2. Все задания поступают в одно и то же время; ← *почти в одно и то же время
3. Задания дорабатывают до конца непрерывно;
4. Задания используют только CPU (никакого IO);
5. Время работы каждого задания известно заранее.

Метрики:

1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - \cancel{T^k_{arrival}}$
2. Справедливость;

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~
2. ~~Все задания поступают в одно и то же время;~~
3. Задания дорабатывают до конца непрерывно;
4. Задания используют только CPU (никакого IO);
5. Время работы каждого задания известно заранее.

Метрики:

1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$
2. Справедливость;

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~
2. ~~Все задания поступают в одно и то же время;~~
3. Задания дорабатывают до конца непрерывно;
4. Задания используют только CPU (никакого IO);
5. Время работы каждого задания известно заранее.

Метрики:

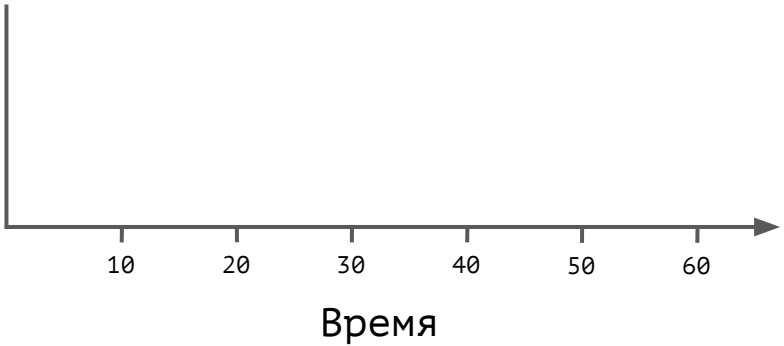
1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$
2. Справедливость;

Что изменится?

Постройте контрпример* к SJF?

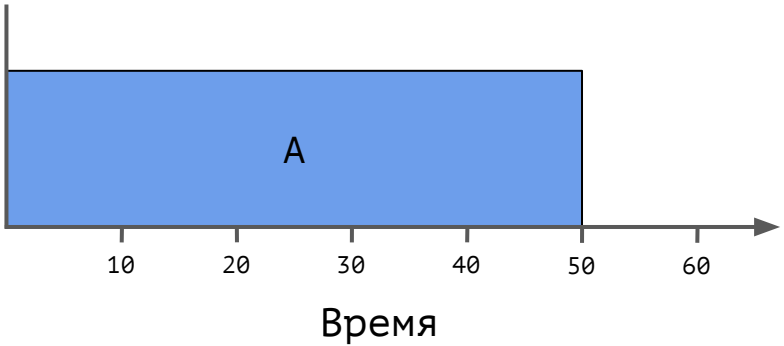
*не совсем контрпример, скорее неудачные входные данные

SJF



Пусть во время 0 поступает зада A, длиной 50 ms.
Больше пока никого нет.

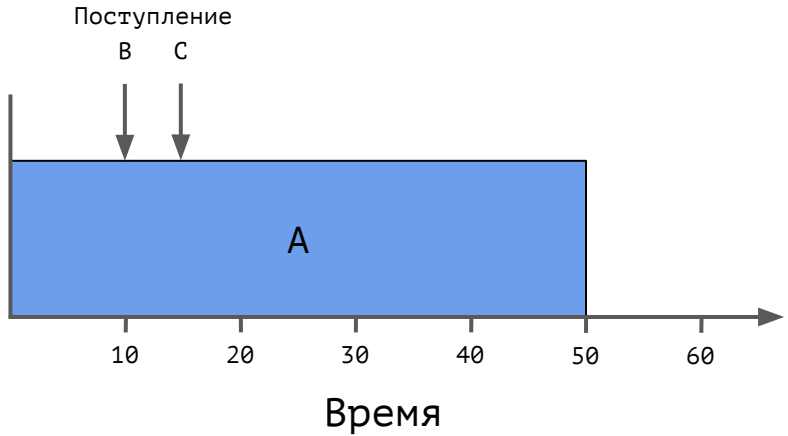
SJF



Пусть во время 0 поступает зада A, длиной 50 ms.
Больше пока никого нет.

Согласно SJF ставим это задачу на исполнение.

SJF

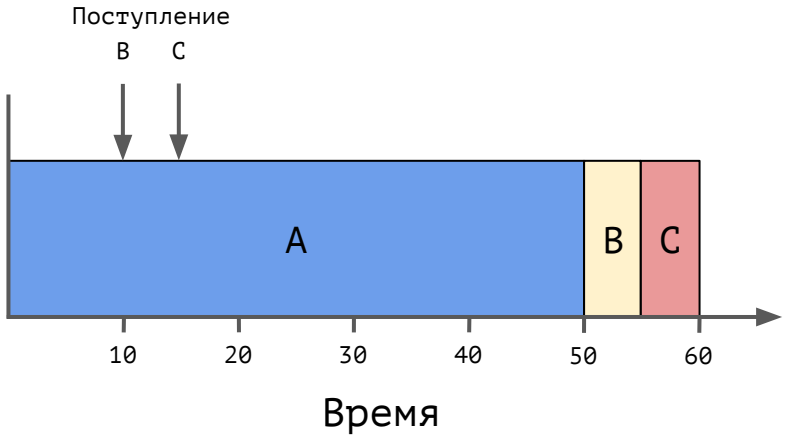


Пусть во время 0 поступает зада A, длиной 50 ms. Больше пока никого нет.

Согласно SJF ставим это задачу на исполнение.

Пусть во время 10 и 15 поступают две задачи B и C по 5 ms каждая.

SJF



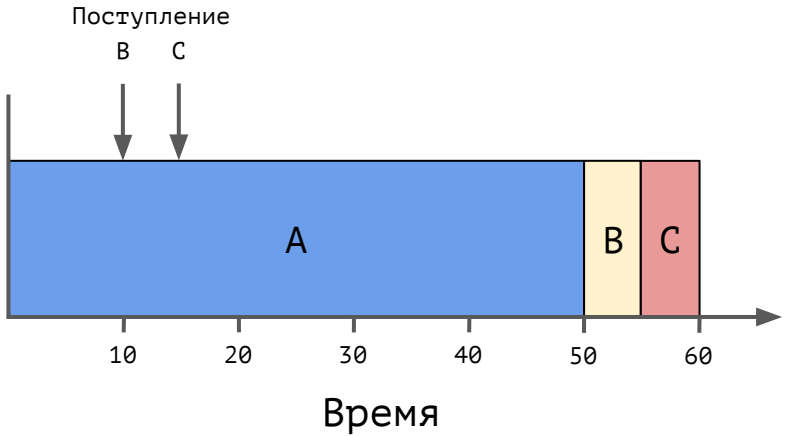
Пусть во время 0 поступает зада A, длиной 50 ms. Больше пока никого нет.

Согласно SJF ставим это задачу на исполнение.

Пусть во время 10 и 15 поступают две задачи B и C по 5 ms каждая.

Согласно нашим ограничениям, можем поставить их только в конец.

SJF



Тогда среднее оборотное время = (50 + 55 + 60) / 3 = **55**

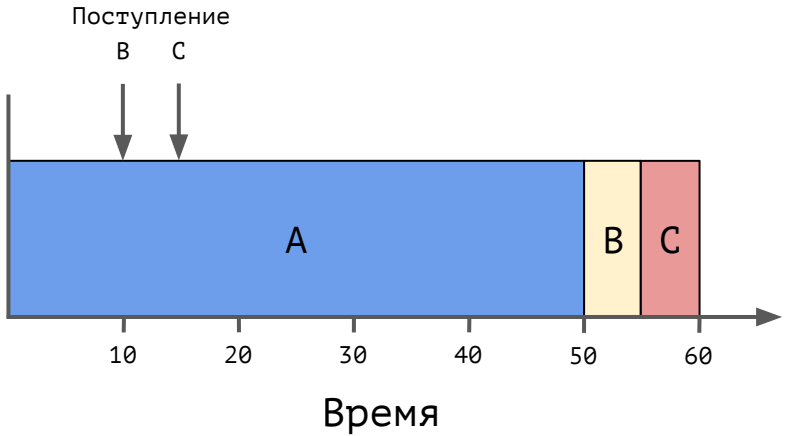
Пусть во время 0 поступает зада A, длиной 50 ms. Больше пока никого нет.

Согласно SJF ставим это задачу на исполнение.

Пусть во время 10 и 15 поступают две задачи B и C по 5 ms каждая.

Согласно нашим ограничениям, можем поставить их только в конец.

SJF



Тогда среднее оборотное время = (50 + 55 + 60) / 3 = **55**

А как лучше?

Пусть во время 0 поступает зада A, длиной 50 ms. Больше пока никого нет.

Согласно SJF ставим это задачу на исполнение.

Пусть во время 10 и 15 поступают две задачи B и C по 5 ms каждая.

Согласно нашим ограничениям, можем поставить их только в конец.

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~
2. ~~Все задания поступают в одно и то же время;~~
3. Задания дорабатывают до конца непрерывно;
4. Задания используют только CPU (никакого IO);
5. Время работы каждого задания известно заранее.

Метрики:

1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$
2. Справедливость;

Что изменится?

Постройте контрпример* к SJF?

*не совсем контрпример, скорее неудачные входные данные

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~
2. ~~Все задания поступают в одно и то же время;~~
3. Задания дорабатывают до конца непрерывно;
4. Задания используют только CPU (никакого IO);
5. Время работы каждого задания известно заранее.

Метрики:

1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$
2. Справедливость;

Что изменится?

Постройте контрпример* к SJF?

*не совсем контрпример, скорее неудачные входные данные

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~
2. ~~Все задания поступают в одно и то же время;~~
3. ~~Задания дорабатывают до конца непрерывно;~~
4. Задания используют только CPU (никакого IO);
5. Время работы каждого задания известно заранее.

Умеем прерывать процессы (задания)

Метрики:

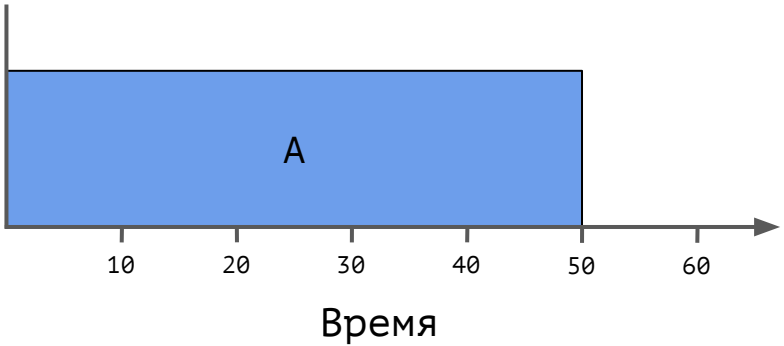
1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$
2. Справедливость;

Что изменится?

Постройте контрпример* к SJF?

*не совсем контрпример, скорее неудачные входные данные

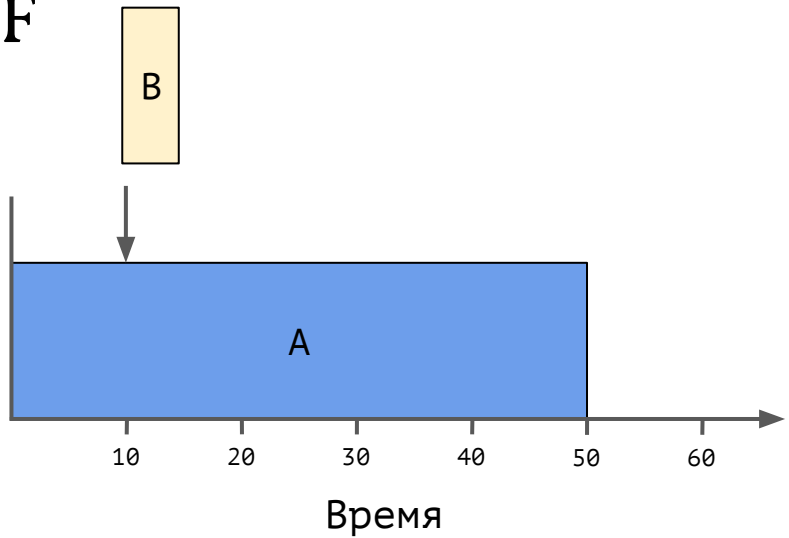
SJF



Пусть во время 0 поступает зада A, длиной 50 ms.
Больше пока никого нет.

Ставим его на CPU.

SJF

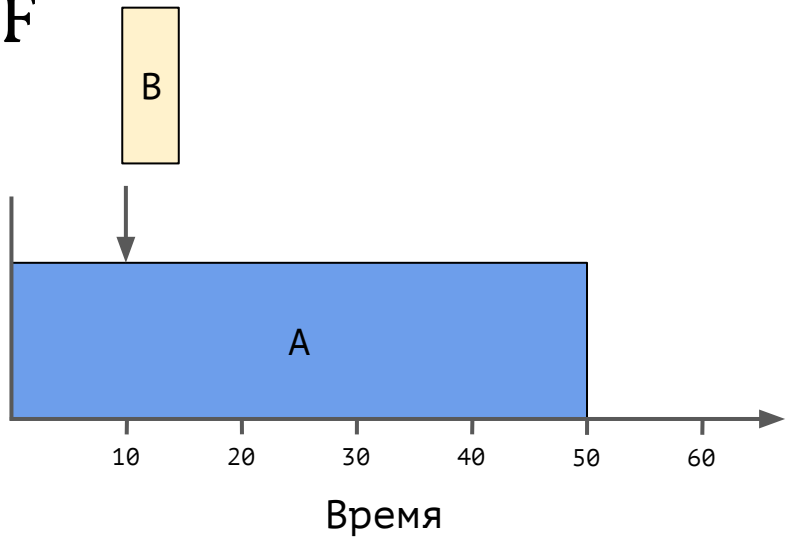


Пусть во время 0 поступает зада A, длиной 50 ms. Больше пока никого нет.

Ставим его на CPU.

В момент 10 появляется новое задание B. Его длительность - 5 ms.

SJF



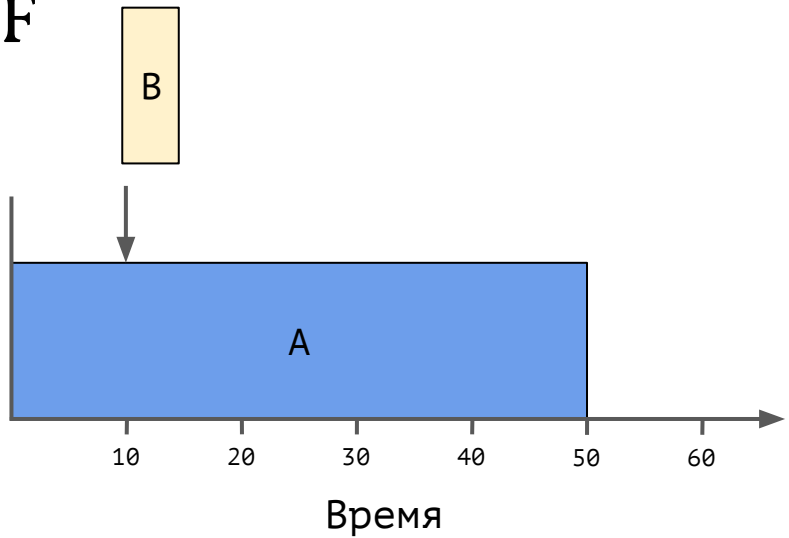
Пусть во время 0 поступает зада A, длиной 50 ms. Больше пока никого нет.

Ставим его на CPU.

В момент 10 появляется новое задание B. Его длительность - 5 ms.

Стоит ли прервать A и поставить на исполнение B?

SJF



Пусть во время 0 поступает зада A, длиной 50 ms. Больше пока никого нет.

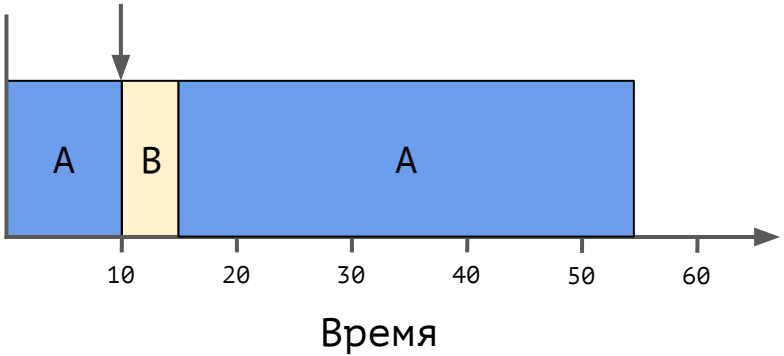
Ставим его на CPU.

В момент 10 появляется новое задание B. Его длительность - 5 ms.

Стоит ли прервать A и поставить на исполнение B?

Да! Если мы так сделаем, то только выиграем в плане оборотно времени.

SJF



Пусть во время 0 поступает зада A, длиной 50 ms. Больше пока никого нет.

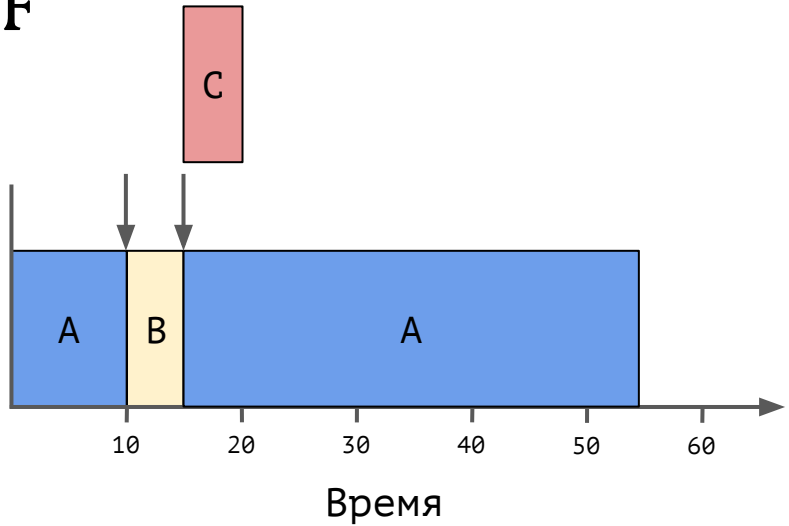
Ставим его на CPU.

В момент 10 появляется новое задание B. Его длительность - 5 ms.

Стоит ли прервать A и поставить на исполнение B?

Да! Если мы так сделаем, то только выиграем в плане оборотно времени.

SJF



Пусть во время 0 поступает зада A, длиной 50 ms. Больше пока никого нет.

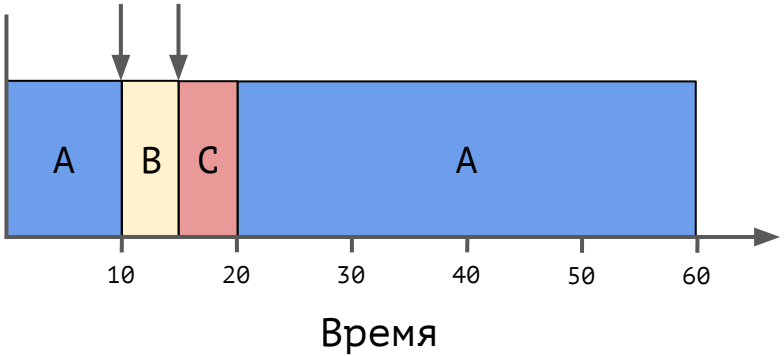
Ставим его на CPU.

В момент 10 появляется новое задание B. Его длительность - 5 ms.

В момент 15 появляется новое задание C. Его длительность - 5 ms.

Сново принимаем решение о прерывании.

SJF



Тогда среднее оборотное время = **(60 + (15-10) + (20-15)) / 3 = 23.3**

Пусть во время 0 поступает зада A, длиной 50 ms. Больше пока никого нет.

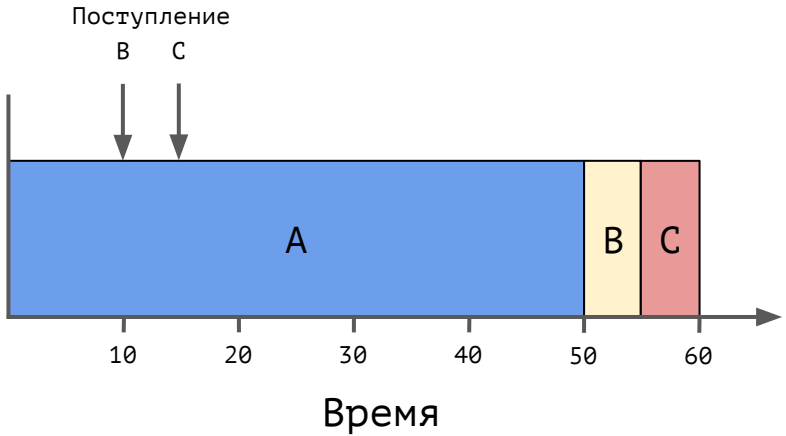
Ставим его на CPU.

В момент 10 появляется новое задание B. Его длительность - 5 ms.

В момент 15 появляется новое задание C. Его длительность - 5 ms.

Сново принимаем решение о прерывании.

SJF



Тогда среднее оборотное время = (50 + 55 + 60) / 3 = **55**

А как лучше?

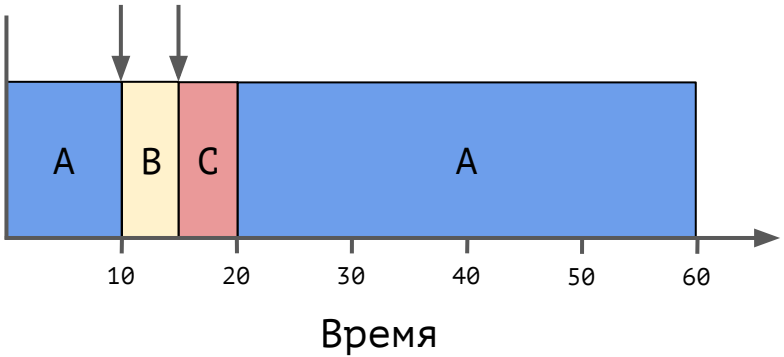
Пусть во время 0 поступает зада A, длиной 50 ms. Больше пока никого нет.

Согласно SJF ставим это задачу на исполнение.

Пусть во время 10 и 15 поступают две задачи B и C по 5 ms каждая.

Согласно нашим ограничениям, можем поставить их только в конец.

SJF

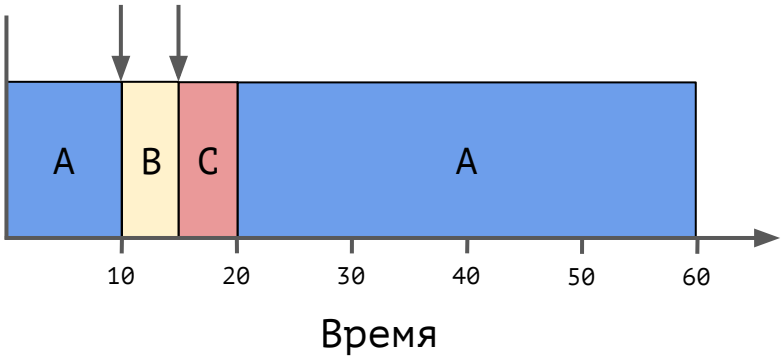


Обобщим: каждый раз при поступлении нового задания принимаем решении о снятии текущего процесса с CPU.

Кого нужно поставить?

Тогда среднее оборотное время = (**60** + **5** + **5**) / 3 = **23.3**

SJF



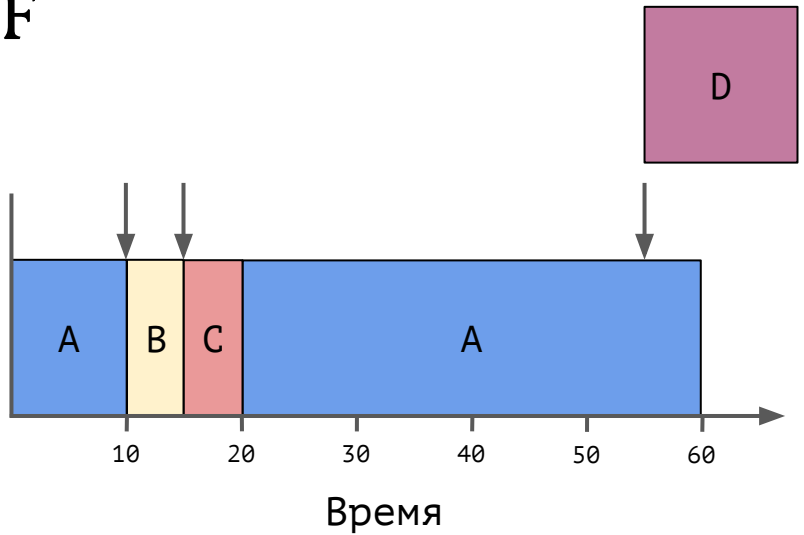
Обобщим: каждый раз при поступлении нового задания принимаем решении о снятии текущего процесса с CPU.

Кого нужно поставить?

Задание с наименьшим временем до окончания.

Тогда среднее оборотное время = (**60** + **5** + **5**) / 3 = **23.3**

SJF



Тогда среднее оборотное время = (**60** + **5** + **5**) / 3 = **23.3**

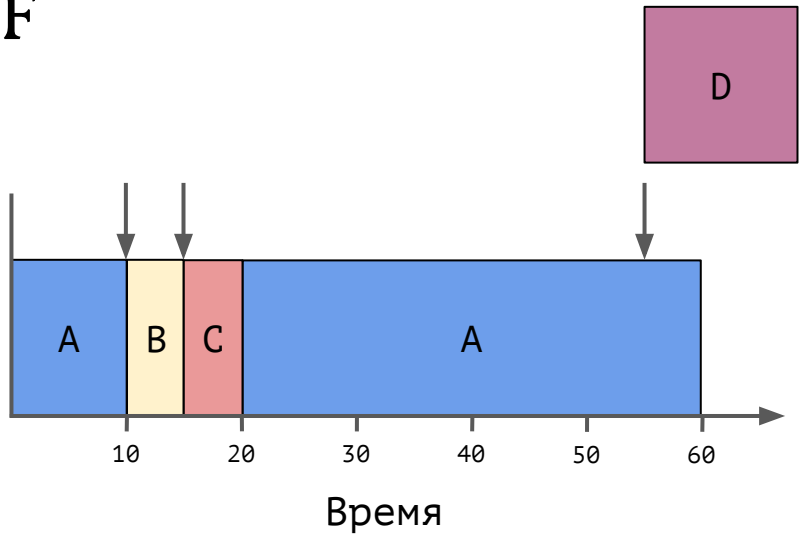
Обобщим: каждый раз при поступлении нового задания принимаем решении о снятии текущего процесса с CPU.

Кого нужно поставить?

Задание с наименьшим временем до окончания.

Если бы задание D длины 15 пришло бы в момент 55, нужно ли было бы прерывать A?

SJF



Тогда среднее оборотное время = (**75** + **5** + **5** + **15**) / 4 = **25**

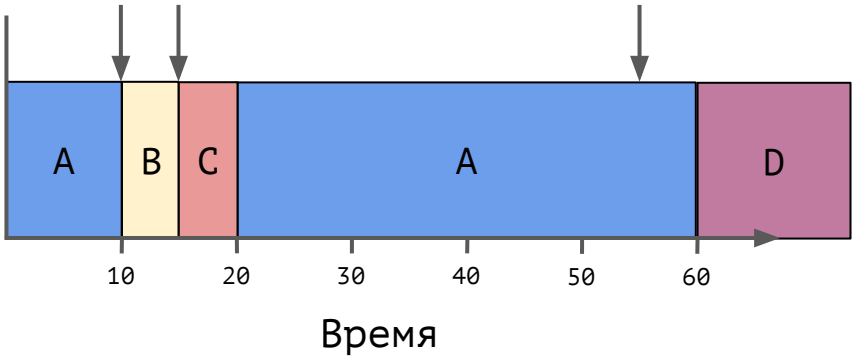
Обобщим: каждый раз при поступлении нового задания принимаем решении о снятии текущего процесса с CPU.

Кого нужно поставить?

Задание с наименьшим временем до окончания.

Если бы задание D длины 15 пришло бы в момент 55, нужно ли было бы прерывать A? Нет, так мы только проиграем в оборотном времени!

SJF



Тогда среднее оборотное время = (**60** + **5** + **5** + **20**) / 4 = **22.5**

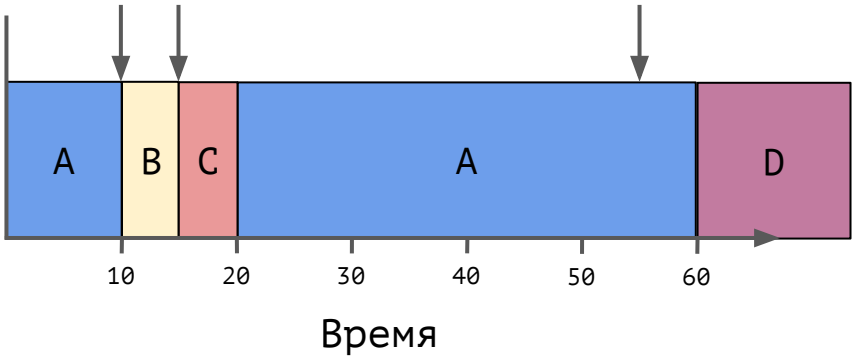
Обобщим: каждый раз при поступлении нового задания принимаем решении о снятии текущего процесса с CPU.

Кого нужно поставить?

Задание с наименьшим временем до окончания.

Если бы задание D длины 15 пришло бы в момент 55, нужно ли было бы прерывать A? Нет, так мы только проиграем в оборотном времени! Оставляем A.

STCF



Тогда среднее оборотное время = (**60** + **5** + **5** + **20**) / 4 = **22.5**

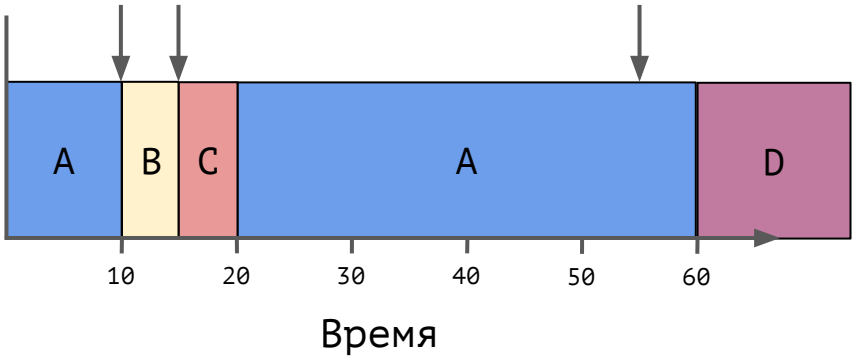
Обобщим: каждый раз при поступлении нового задания принимаем решении о снятии текущего процесса с CPU.

Кого нужно поставить?

Задание с наименьшим временем до окончания.

Такая политика называется STCF (Shortest-Time-To-Completion)

STCF



Тогда среднее оборотное время = (**60** + **5** + **5** + **20**) / 4 = **22.5**

Обобщим: каждый раз при поступлении нового задания принимаем решении о снятии текущего процесса с CPU.

Кого нужно поставить?

Задание с наименьшим временем до окончания.

Такая политика называется STCF (Shortest-Time-To-Completion)

Тоже жадник и тоже оптимален в наших ограничениях

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~
2. ~~Все задания поступают в одно и то же время;~~
3. ~~Задания дорабатывают до конца непрерывно;~~
4. Задания используют только CPU (никакого IO);
5. Время работы каждого задания известно заранее.

Умеем прерывать процессы (задания)

Метрики:

1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$
2. Справедливость;

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

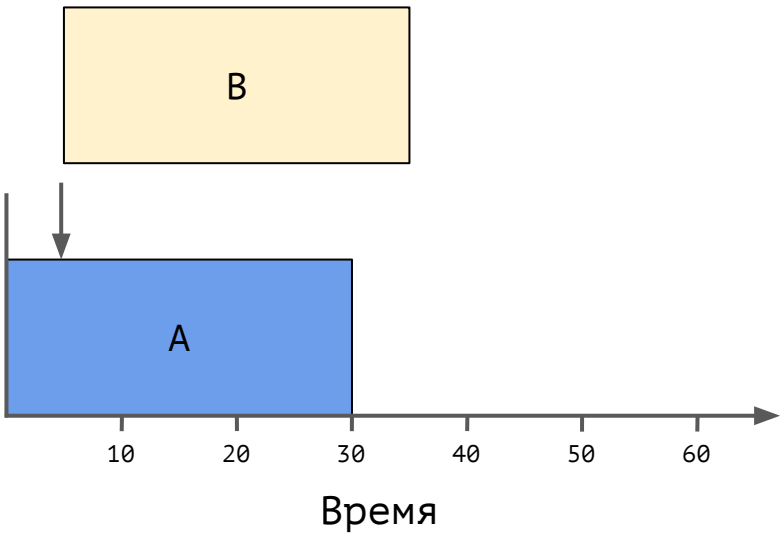
1. ~~Время выполнения всех заданий одинаково;~~
2. ~~Все задания поступают в одно и то же время;~~
3. ~~Задания дорабатывают до конца непрерывно;~~
4. Задания используют только CPU (никакого IO); ←
5. Время работы каждого задания известно заранее.

Метрики:

1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$
2. Справедливость;

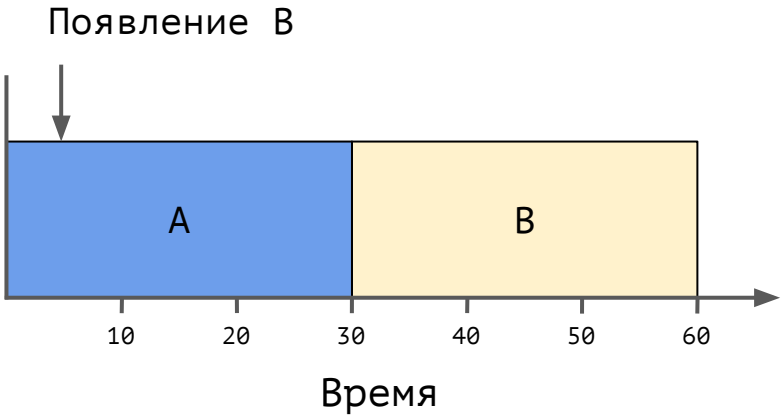
Как бы это учесть? Ведь во время работы с IO можно было бы занять CPU чем-то еще

STCF



Пусть в момент 0 появляется задание A длины 30 ms. A в момент 5 появляется задание B длины 30 ms.

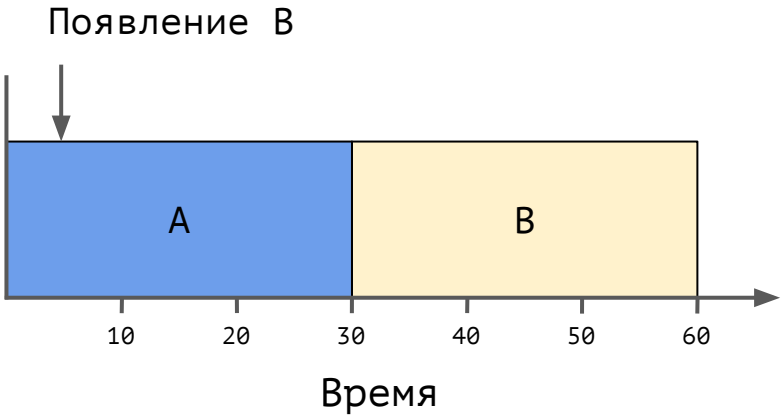
STCF



Пусть в момент 0 появляется задание A длины 30 ms. A в момент 5 появляется задание B длины 30 ms.

Согласно STCF мы просто ставим B после A.

STCF

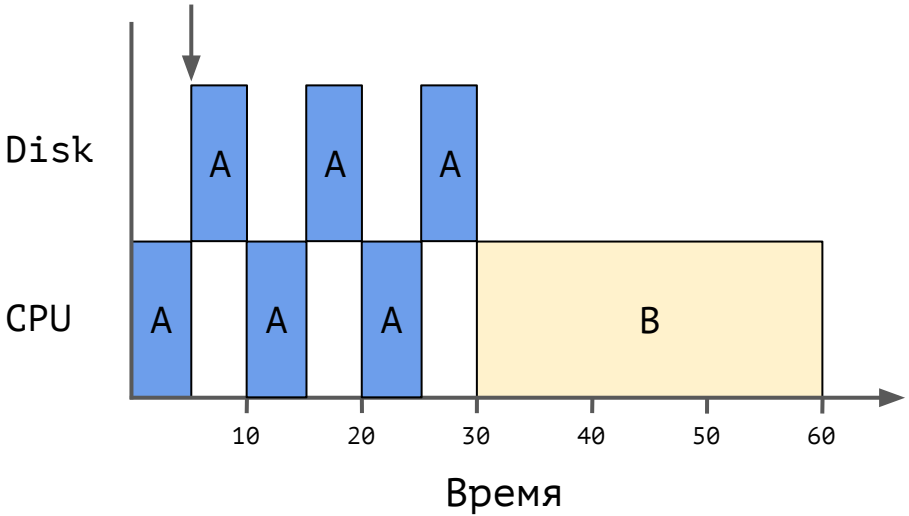


Пусть в момент 0 появляется задание A длины 30 ms. A в момент 5 появляется задание B длины 30 ms.

Согласно STCF мы просто ставим B после A.

Но пусть теперь A каждые 5 ms начинает ждать диск (тоже по 5 ms)

STCF

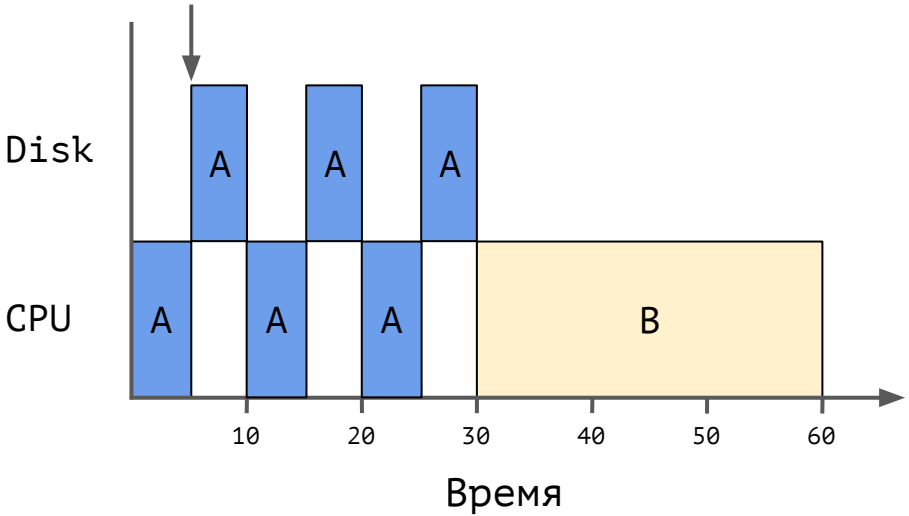


Пусть в момент 0 появляется задание A длины 30 ms. A в момент 5 появляется задание B длины 30 ms.

Согласно STCF мы просто ставим B после A.

Но пусть теперь A каждые 5 ms начинает ждать диск (тоже по 5 ms)

STCF



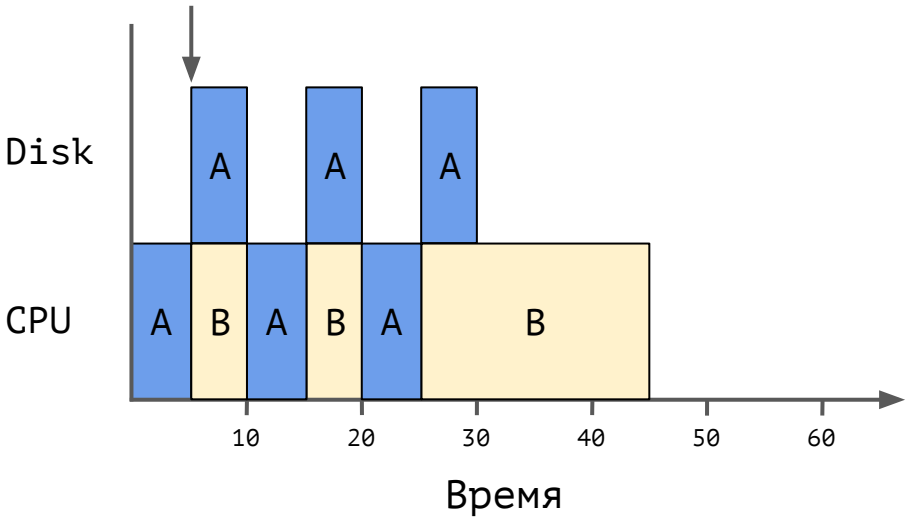
Пусть в момент 0 появляется задание A длины 30 ms. A в момент 5 появляется задание B длины 30 ms.

Согласно STCF мы просто ставим B после A.

Но пусть теперь A каждые 5 ms начинает ждать диск (тоже по 5 ms)

Контролируем моменты запроса IO и возврата (процесс из заблокирован переходит в готов)

STCF+



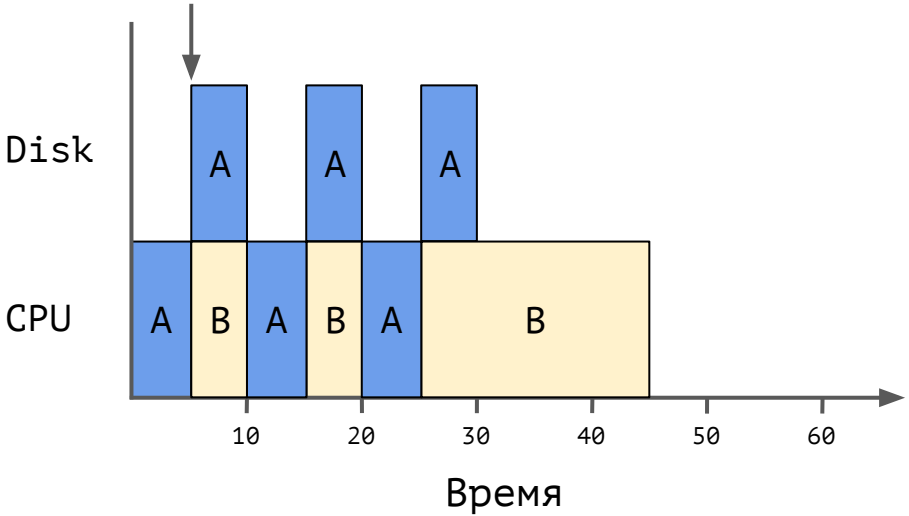
Пусть в момент 0 появляется задание A длины 30 ms. A в момент 5 появляется задание B длины 30 ms.

Согласно STCF мы просто ставим B после A.

Но пусть теперь A каждые 5 ms начинает ждать диск (тоже по 5 ms)

Контролируем моменты запроса IO и возврата (процесс из заблокирован переходит в готов)

STCF+



Пусть в момент 0 появляется задание A длины 30 ms. A в момент 5 появляется задание B длины 30 ms.

Согласно STCF мы просто ставим B после A.

Но пусть теперь A каждые 5 ms начинает ждать диск (тоже по 5 ms)

Контролируем моменты запроса IO и возврата (процесс из заблокирован переходит в готов)

Достигается довольно легко: достаточно в STCF считать непрерывные части задач на CPU отдельными заданиями

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~
2. ~~Все задания поступают в одно и то же время;~~
3. ~~Задания доработывают до конца непрерывно;~~
4. Задания используют только CPU (никакого IO); ←
5. Время работы каждого задания известно заранее.

Метрики:

1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$
2. Справедливость;

Учитываем, разбивая задачи на непрерывно исполняемые на CPU части

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~
2. ~~Все задания поступают в одно и то же время;~~
3. ~~Задания дорабатывают до конца непрерывно;~~
4. ~~Задания используют только CPU (никакого IO);~~
5. Время работы каждого задания известно заранее.

Метрики:

1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$
2. Справедливость;

Учитываем, разбивая задачи на непрерывно исполняемые на CPU части

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~
2. ~~Все задания поступают в одно и то же время;~~
3. ~~Задания дорабатывают до конца непрерывно;~~
4. ~~Задания используют только CPU (никакого IO);~~
5. Время работы каждого задания известно заранее.

Метрики:

1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$
2. Справедливость;
3. Время отклика: время первого запуска минус время появление задания в системе

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

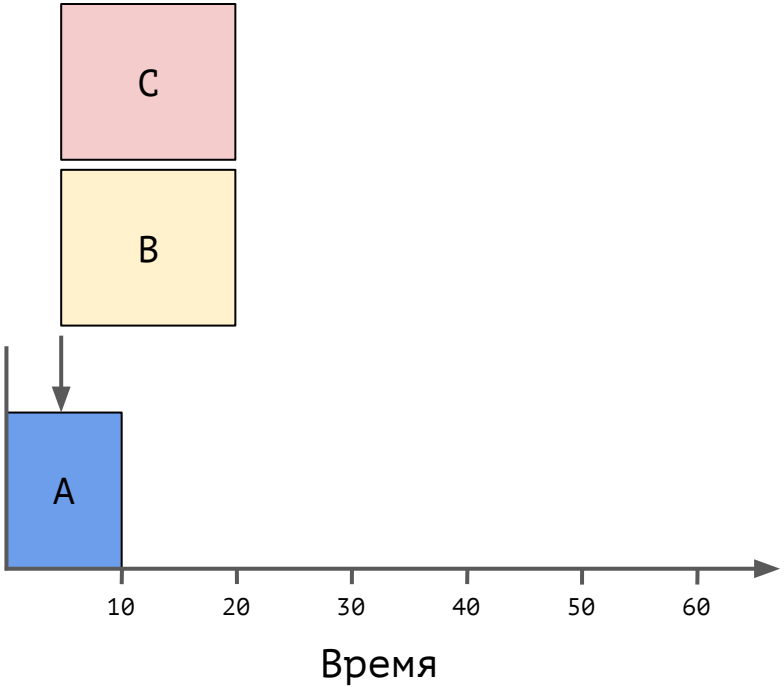
Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~
2. ~~Все задания поступают в одно и то же время;~~
3. ~~Задания дорабатывают до конца непрерывно;~~
4. ~~Задания используют только CPU (никакого IO);~~
5. Время работы каждого задания известно заранее.

Метрики:

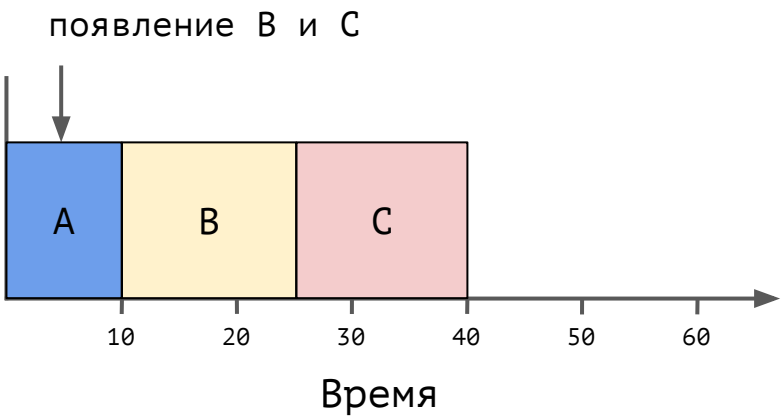
1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$
2. Справедливость;
3. [Среднее] время отклика: $T^k_{response} = T^k_{firstrun} - T^k_{arrival}$

Смысл: не хочется заставлять пользователя ждать, пока вся очередь рассосется



Пусть в момент 0 появляется задание A длины 10 ms. A в момент 5 появляются задания B и C длины 15 ms.

STCF

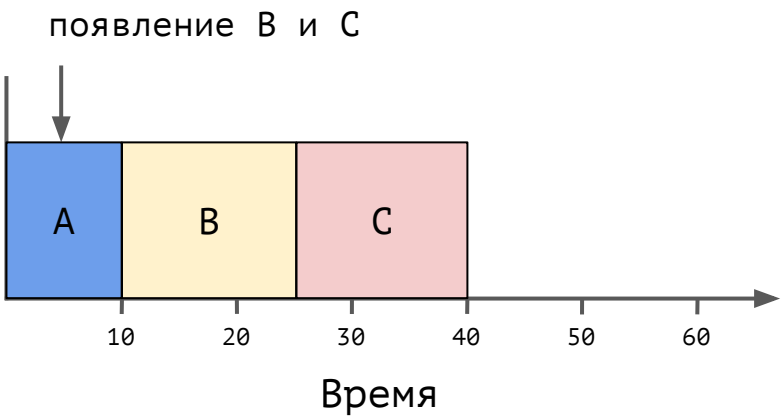


Пусть в момент 0 появляется задание A длины 10 ms. A в момент 5 появляются задания B и C длины 15 ms.

Согласно STCF ставим B и C в конец.

Оборотное время - норм ((10 - 0) + (25 - 5) + (40 - 5) / 3) = 21.6,

STCF

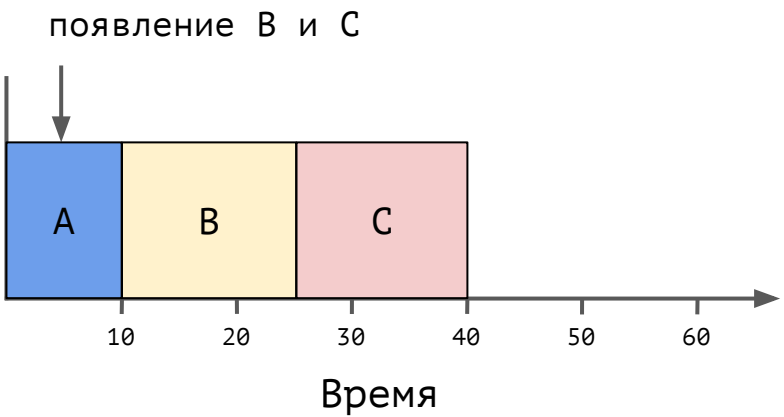


Пусть в момент 0 появляется задание A длины 10 ms. A в момент 5 появляются задания B и C длины 15 ms.

Согласно STCF ставим B и C в конец.

Оборотное время - норм ((10 - 0) + (25 - 5) + (40 - 5) / 3) = 21.6, но вот время отклика = ((0 - 0) + (10 - 5) + (25 - 5)) / 3 = 6.6

STCF



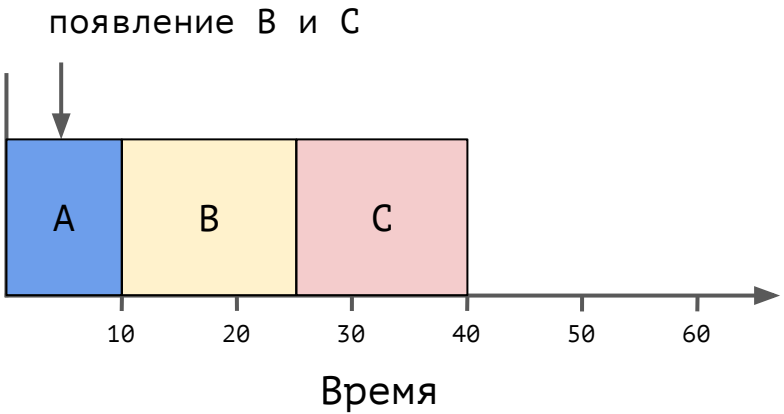
Пусть в момент 0 появляется задание A длины 10 ms. A в момент 5 появляются задания B и C длины 15 ms.

Согласно STCF ставим B и C в конец.

Оборотное время - норм (21.6), но вот время отклика (6.6)

Пользователю задачи C прям грустно, долго ждать

STCF



Какую политику предложите для максимизации среднего времени отклика?

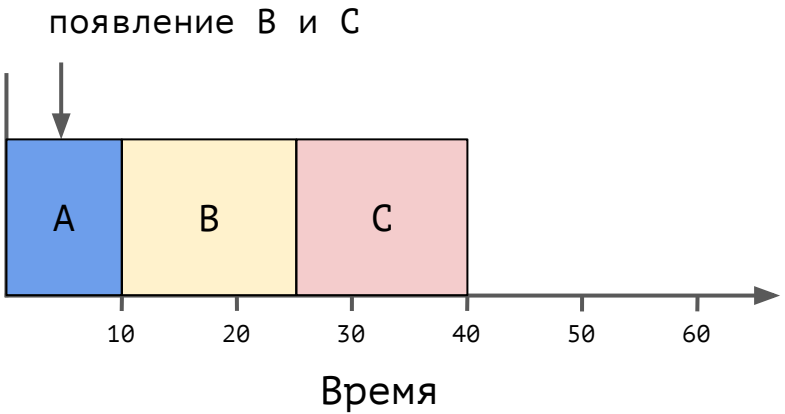
Пусть в момент 0 появляется задание A длины 10 ms. A в момент 5 появляются задания B и C длины 15 ms.

Согласно STCF ставим B и C в конец.

Оборотное время - норм (21.6), но вот время отклика (6.6)

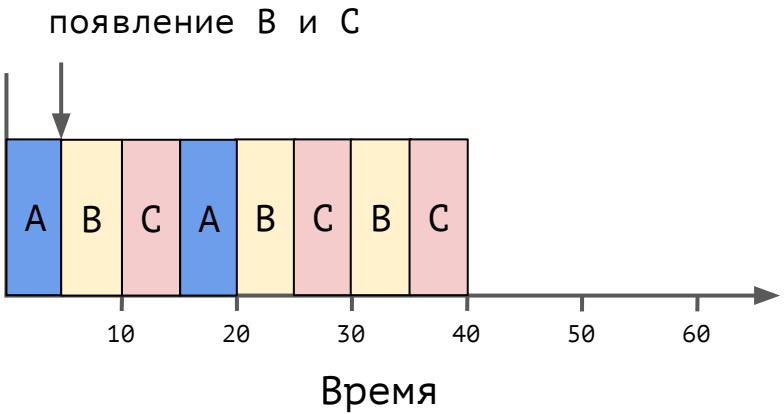
Пользователю задачи C прям грустно, долго ждать

Циклическое планирование (Round-Robin, RR)



Выберем промежуток времени, кратный разрешающей способности таймера и назовем его квантом планирования. Пусть он будет равен 5ms.

Циклическое планирование (Round-Robin, RR)

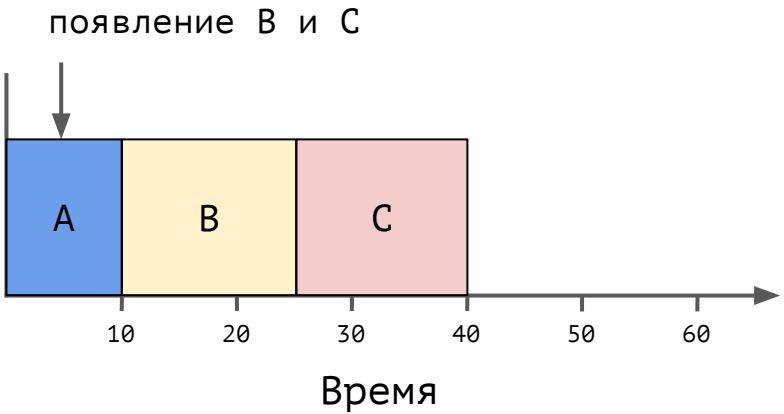


Выберем промежуток времени, кратный разрешающей способности таймера и назовем его квантом планирования. Пусть он будет равен 5ms.

Раз в 5ms будем снимать с CPU процесс (задание) и ставить следующий по списку.

Среднее время отклика = ((0-0) + (5-5) + (10-5))/3 = 1.6

STCF



Какую политику предложите для максимизации среднего времени отклика?

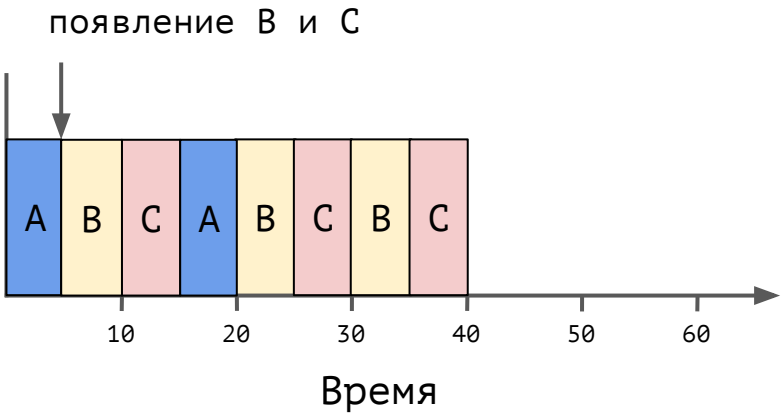
Пусть в момент 0 появляется задание A длины 10 ms. A в момент 5 появляются задания B и C длины 15 ms.

Согласно STCF ставим B и C в конец.

Оборотное время - норм (21.6), но вот время отклика (6.6)

Пользователю задачи C прям грустно, долго ждать

Циклическое планирование (Round-Robin, RR)

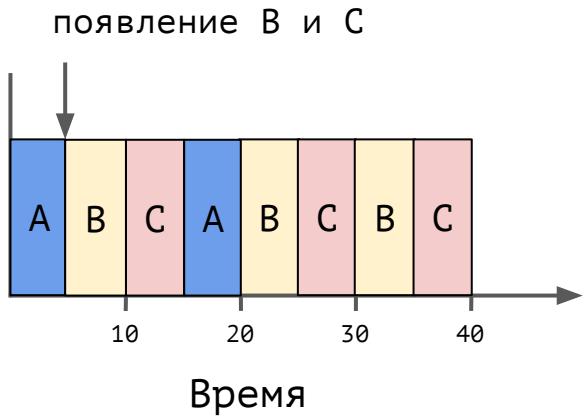


Выберем промежуток времени, кратный разрешающей способности таймера и назовем его квантом планирования. Пусть он будет равен 5ms.

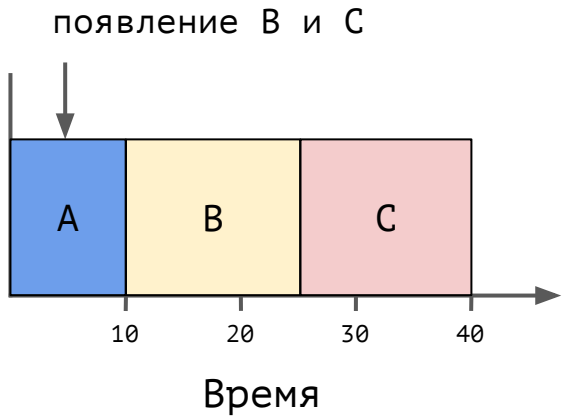
Раз в 5ms будем снимать с CPU процесс (задание) и ставить следующий по списку.

Среднее время отклика = 1.6
Среднее оборотное = ((20-0) + (35-5) + (40-5))/3 = 28.3

Циклическое планирование (Round-Robin, RR)

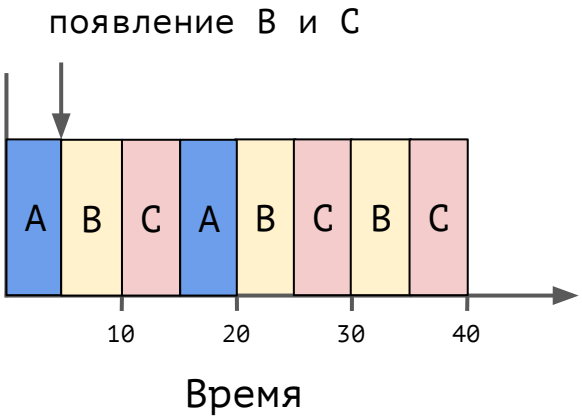


Среднее оборотное время = 28.3
Среднее время отклика = 1.6

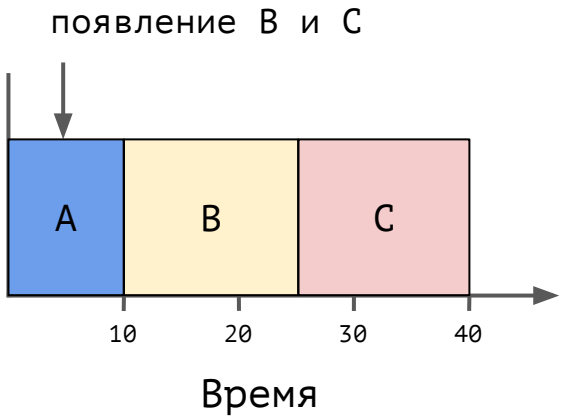


Среднее оборотное время = 21.6
Среднее время отклика = 6.6

Циклическое планирование (Round-Robin, RR)



Среднее оборотное время = 28.3
Среднее время отклика = 1.6



Среднее оборотное время = 21.6
Среднее время отклика = 6.6

Естественный trade off между отзывчивостью и производительностью (дополнительно не забываем о стоимости переключения контекста)

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~
2. ~~Все задания поступают с одно и то же время;~~
3. ~~Задания дорабатывают до конца непрерывно;~~
4. ~~Задания используют только CPU (никакого IO);~~
5. Время работы каждого задания известно заранее.

Метрики:

1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$
2. Справедливость;
3. [Среднее] время отклика: $T^k_{response} = T^k_{firstrun} - T^k_{arrival}$

RR дает и хорошее время отклика и безусловную справедливость (ценой оборотного времени)

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~ FCFS
2. ~~Все задания поступают с одно и то же время;~~ SJF
3. ~~Задания доработывают до конца непрерывно;~~ STCF
4. ~~Задания используют только CPU (никакого IO);~~ STCF+
5. Время работы каждого задания известно заранее.

Метрики:

1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$
2. Справедливость;
3. [Среднее] время отклика: $T^k_{response} = T^k_{firstrun} - T^k_{arrival}$

RR дает и хорошее время отклика и безусловную справедливость (ценой оборотного времени)

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~
2. ~~Все задания поступают с одно и то же время;~~
3. ~~Задания дорабатывают до конца непрерывно;~~
4. ~~Задания используют только CPU (никакого IO);~~
5. Время работы каждого задания известно заранее. ← ???

Метрики:

1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$
2. Справедливость;
3. [Среднее] время отклика: $T^k_{response} = T^k_{firstrun} - T^k_{arrival}$

Как избавиться от оракулов, предсказывающих время работы задач?

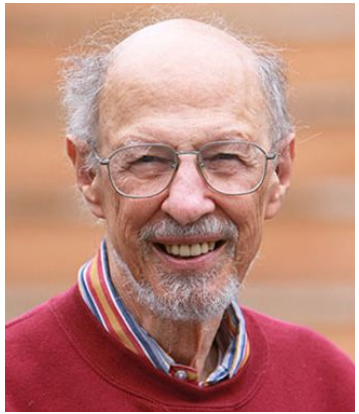
Многоуровневая аналитическая очередь (MLFQ)

Multi-level Feedback Queue

Многоуровневая аналитическая очередь (MLFQ)

Multi-level Feedback Queue

В том числе за нее
Фернандо Корбато
получил премию Тьюринга



Многоуровневая аналитическая очередь (MLFQ)

Идеи:

1. Не знаем время работы и даже характер поведения задач заранее, поэтому не можем заранее выбрать между STCF (оборотное время) и RR (время отклика).

2. Будем принимать решения на лету, в зависимости от наблюдаемого поведения задач (аппроксимируя тот или иной планировщик)

Многоуровневая аналитическая очередь (MLFQ)

Реализация:

1. Время квантуется, после каждого кванта принимается решение о следующем исполняемом задании;

Многоуровневая аналитическая очередь (MLFQ)

Реализация:

1. Время квантуется, после каждого кванта принимается решение о следующем исполняемом задании;

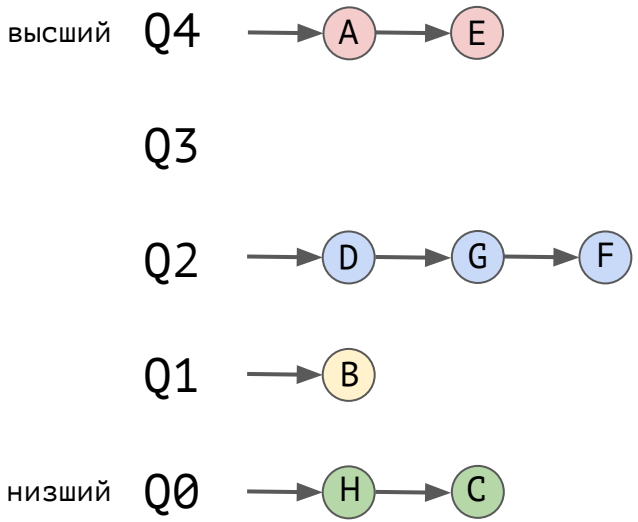
2. Каждому заданию присваивается приоритет, который меняется по ходу работы системы по некоторым правилам;

3. Для каждого приоритета заводится очередь из заданий с этим приоритетом.

Многоуровневая аналитическая очередь (MLFQ)

Реализация:

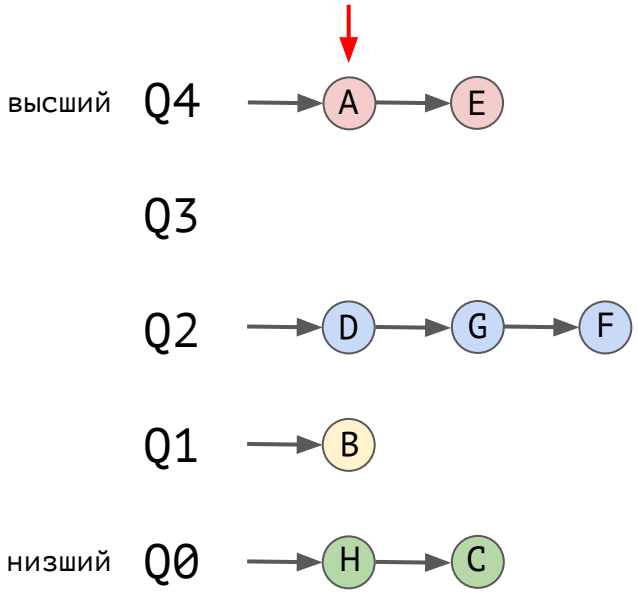
1. Время квантуется, после каждого кванта принимается решение о следующем исполняемом задании;
2. У каждого задания - приоритет, который меняется по ходу работы системы по некоторым правилам;
3. Для каждого приоритета очередь из заданий с этим приоритетом.



Многоуровневая аналитическая очередь (MLFQ)

Реализация:

1. Время квантуется, после каждого кванта принимается решение о следующем исполняемом задании;
2. У каждого задания - приоритет, который меняется по ходу работы системы по некоторым правилам;
3. Для каждого приоритета очередь из заданий с этим приоритетом.



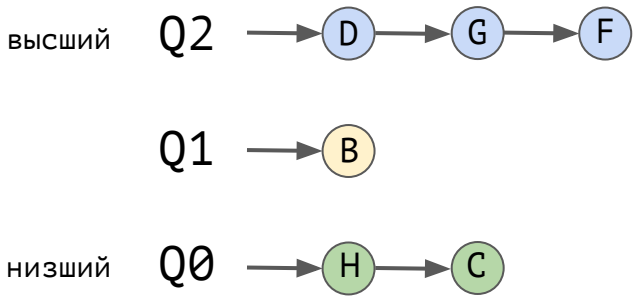
При взятии следующего задания:

1. Выбираем наиболее приоритетную очередь,
2. Если в очереди много заданий, используем на них **RR**

MLFQ: изменение приоритетов

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



MLFQ: изменение приоритетов

Пусть приходит таск A продолжительности 50ms

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .

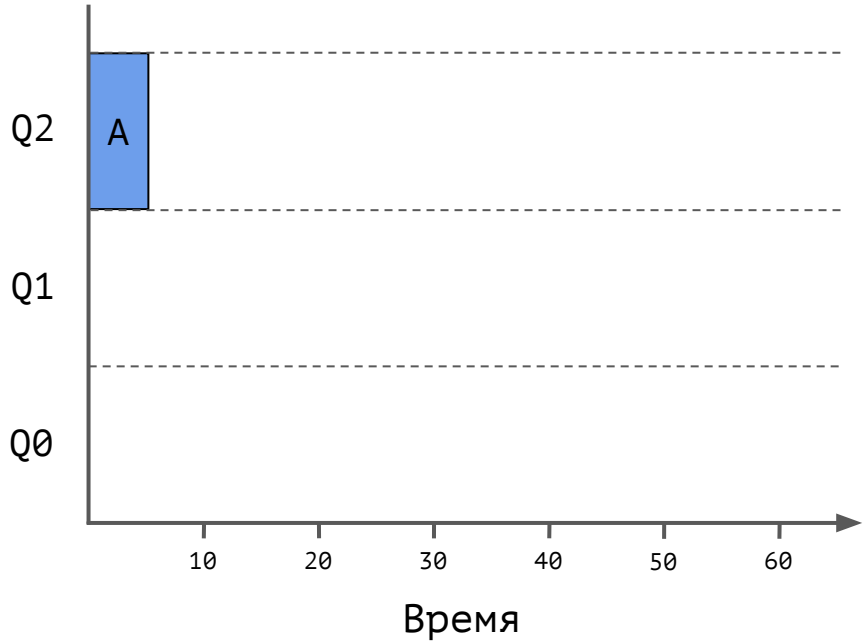
Пример 1: долгий таск тонет (период квантования 5ms)

MLFQ: изменение приоритетов

Пусть приходит таск A продолжительности 50ms,

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



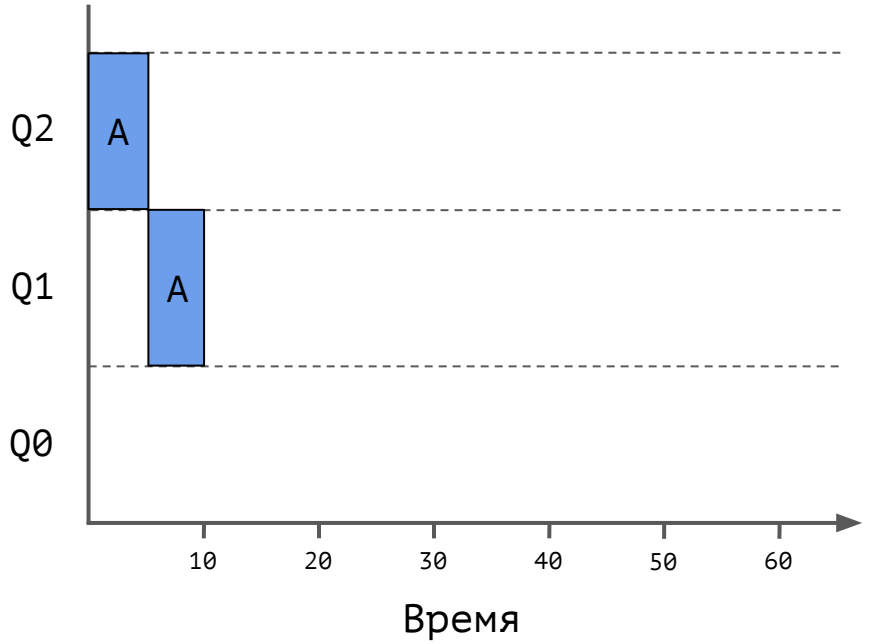
Пример 1: долгий таск тонет (период квантования 5ms)

MLFQ: изменение приоритетов

Пусть приходит таск A продолжительности 50ms, за первый квант не заканчивается => понижает приоритет

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



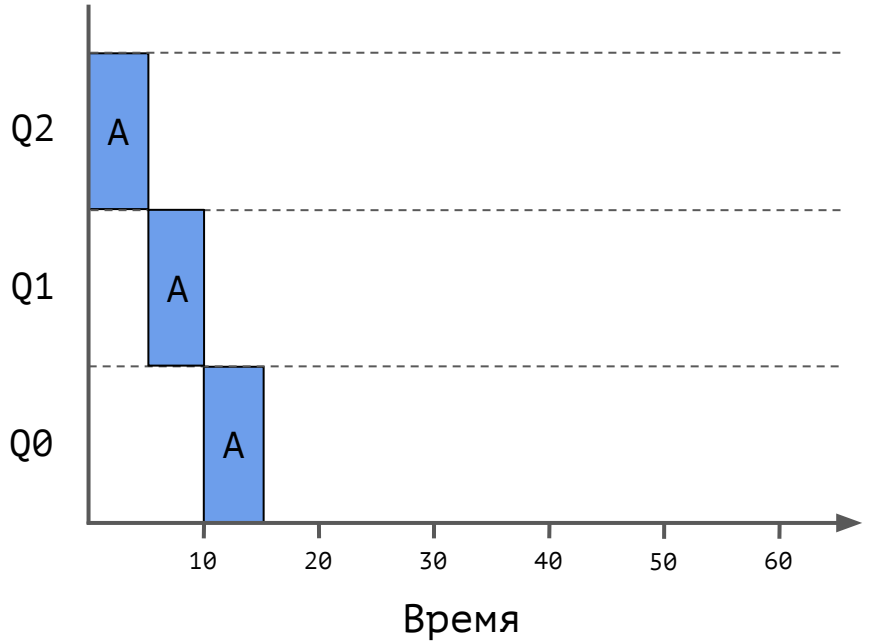
Пример 1: долгий таск тонет (период квантования 5ms)

MLFQ: изменение приоритетов

Пусть приходит таск A продолжительности 50ms, за первый квант не заканчивается => понижает приоритет => после второго кванта опять теряет

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



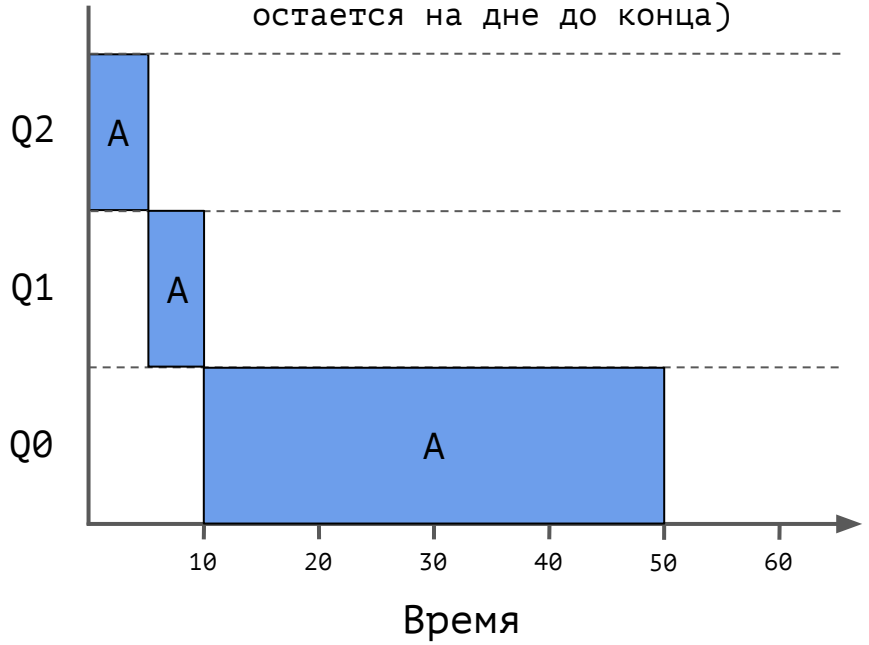
Пример 1: долгий таск тонет (период квантования 5ms)

MLFQ: изменение приоритетов

Пусть приходит таск A продолжительности 50ms, за первый квант не заканчивается => понижает приоритет => после второго кванта опять теряет (и остается на дне до конца)

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



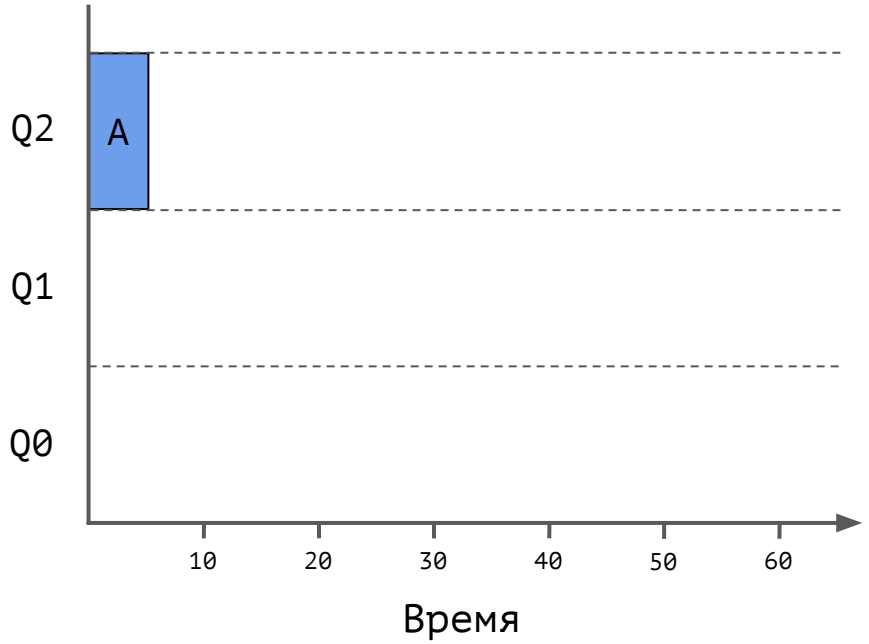
Пример 1: долгий таск тонет (период квантования 5ms)

MLFQ: изменение приоритетов

Пусть во время 0 приходит таск A продолжительности 50ms

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



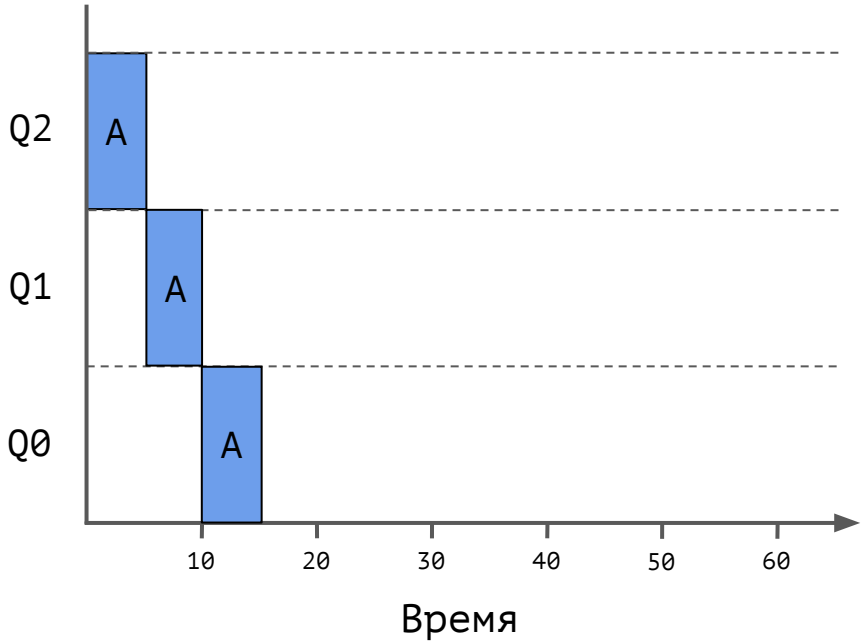
Пример 2: долгий таск тонет и приходит короткое задание (период квантования 5ms)

MLFQ: изменение приоритетов

Пусть во время 0 приходит таск A продолжительности 50ms, а во время 15 приходит B, продолжительности 10ms

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



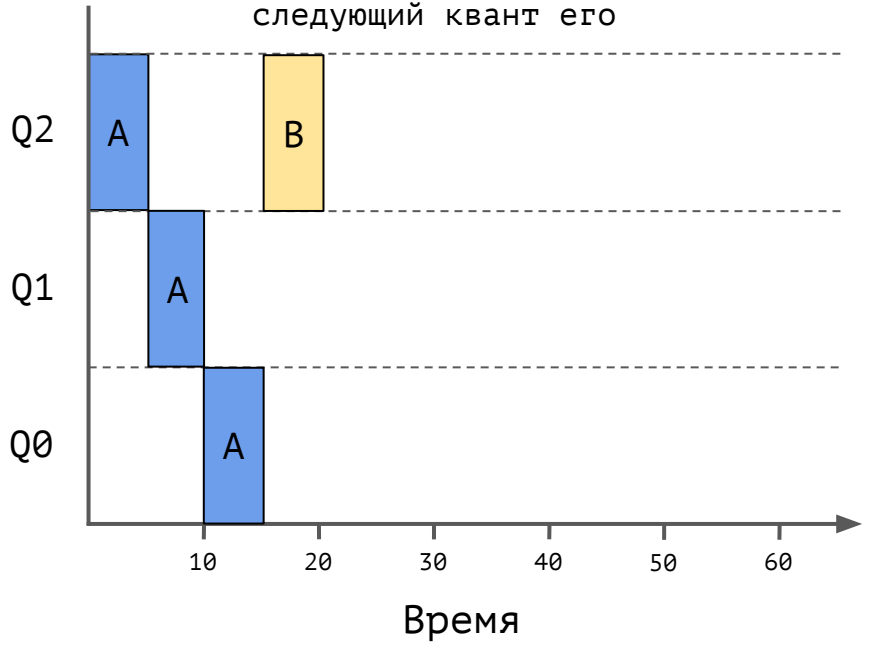
Пример 2: долгий таск тонет и приходит короткое задание (период квантования 5ms)

MLFQ: изменение приоритетов

Пусть во время 0 приходит таск A продолжительности 50ms, а во время 15 приходит B, продолжительности 10ms. У него будет приоритет выше => следующий квант его

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



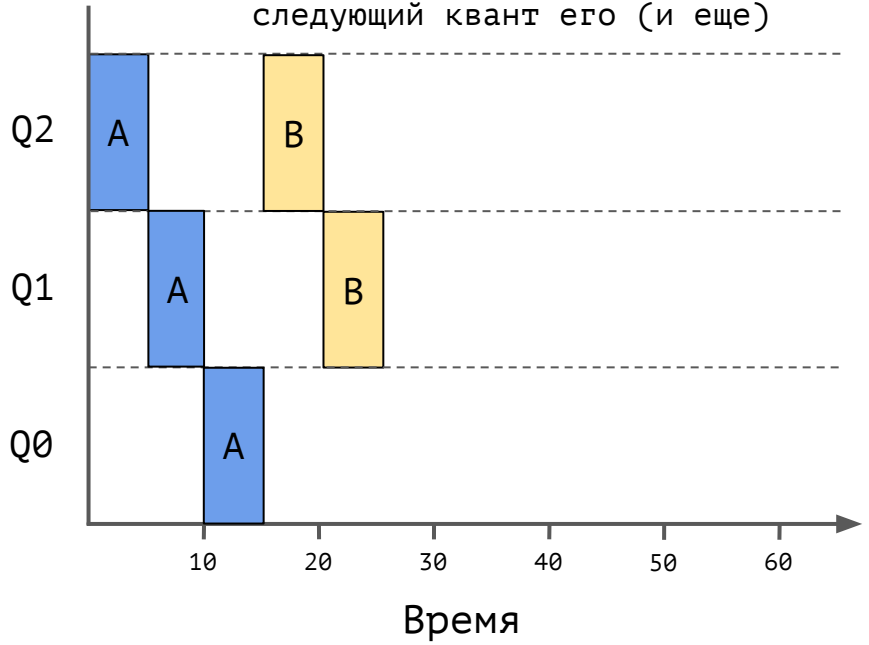
Пример 2: долгий таск тонет и приходит короткое задание (период квантования 5ms)

MLFQ: изменение приоритетов

Пусть во время 0 приходит таск A продолжительности 50ms, а во время 15 приходит B, продолжительности 10ms. У него будет приоритет выше => следующий квант его (и еще)

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



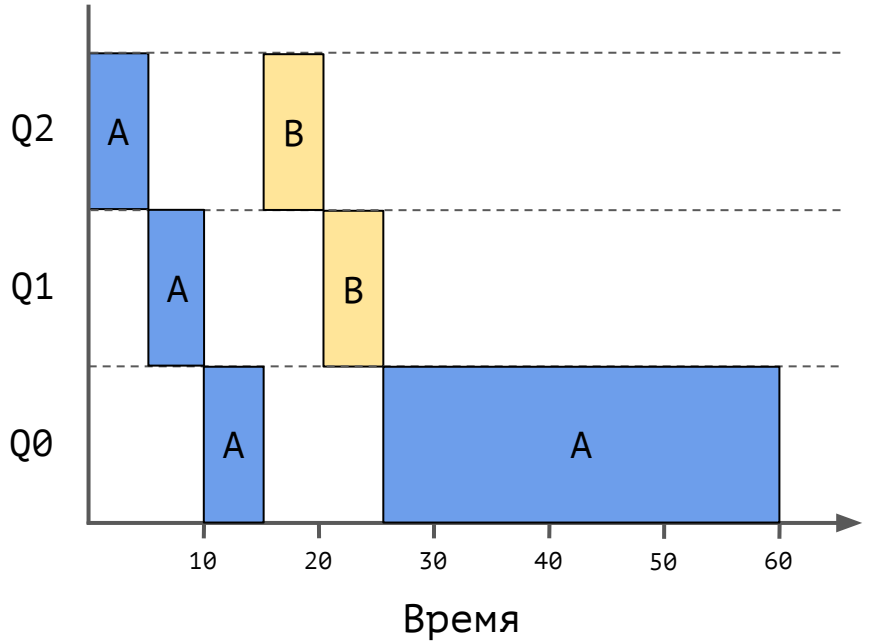
Пример 2: долгий таск тонет и приходит короткое задание (период квантования 5ms)

MLFQ: изменение приоритетов

Пусть во время 0 приходит таск A продолжительности 50ms, а во время 15 приходит B => … => и только потом исполнение возвращается к долгому A

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



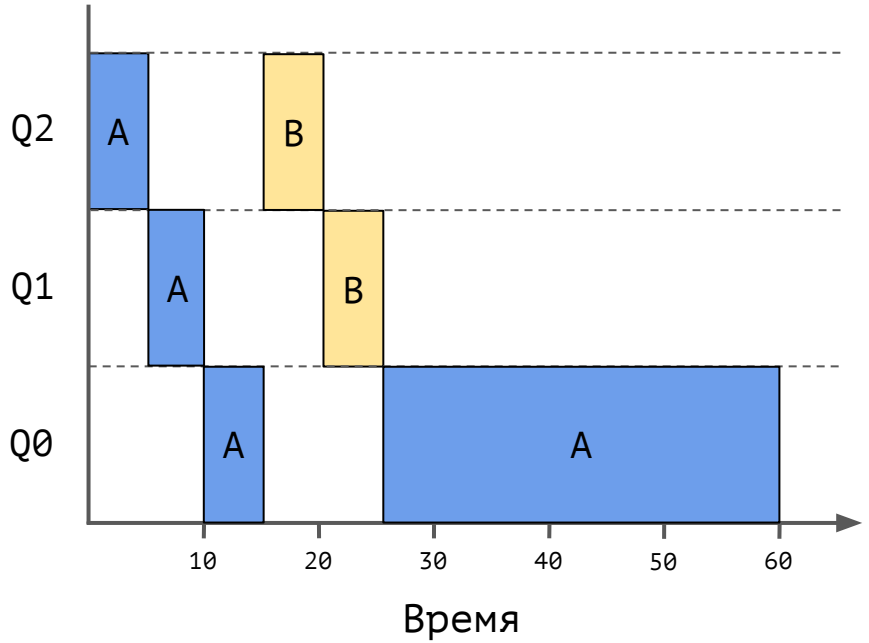
Пример 2: долгий таск тонет и приходит короткое задание (период квантования 5ms)

MLFQ: изменение приоритетов

Пусть во время 0 приходит таск A продолжительности 50ms, а во время 15 приходит B => … => и только потом исполнение возвращается к долгому A

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



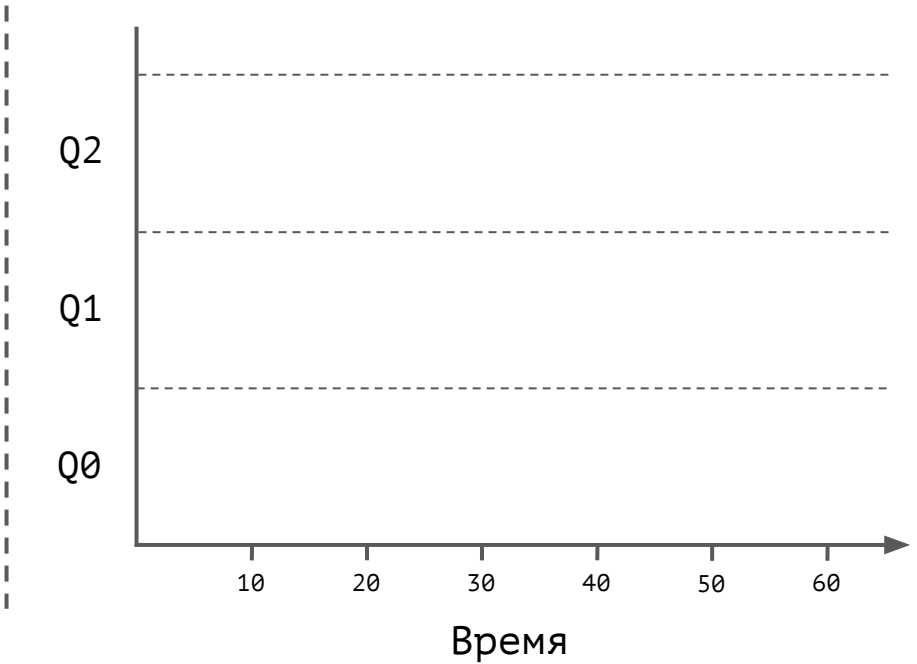
Пример 2: долгий таск тонет и приходит короткое задание (период квантования 5ms)

Таким образом, мы поступили абсолютно, как STCF, что хорошо 👍

MLFQ: изменение приоритетов

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .

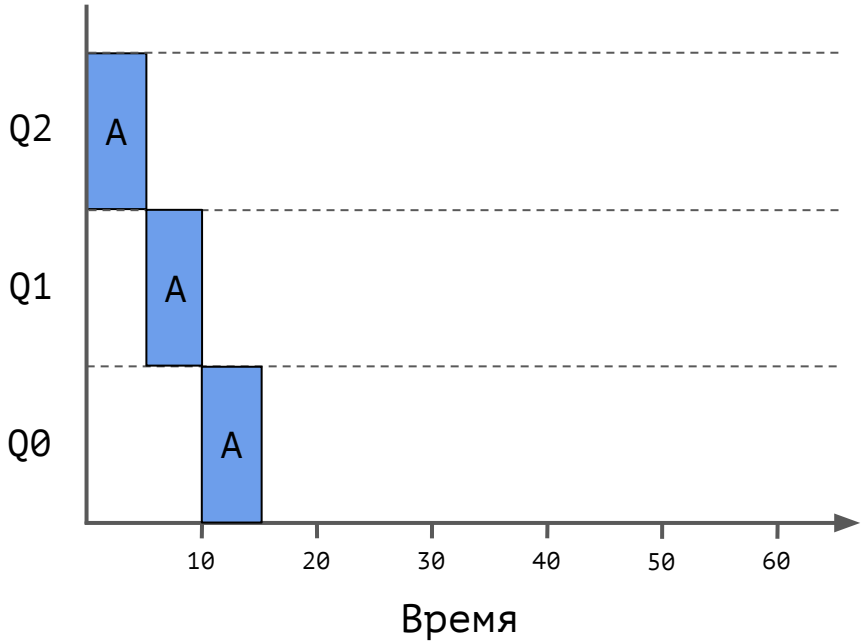


MLFQ: изменение приоритетов

Пусть во время 0 приходит таск A продолжительности 50ms, а во время 15 приходит B

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



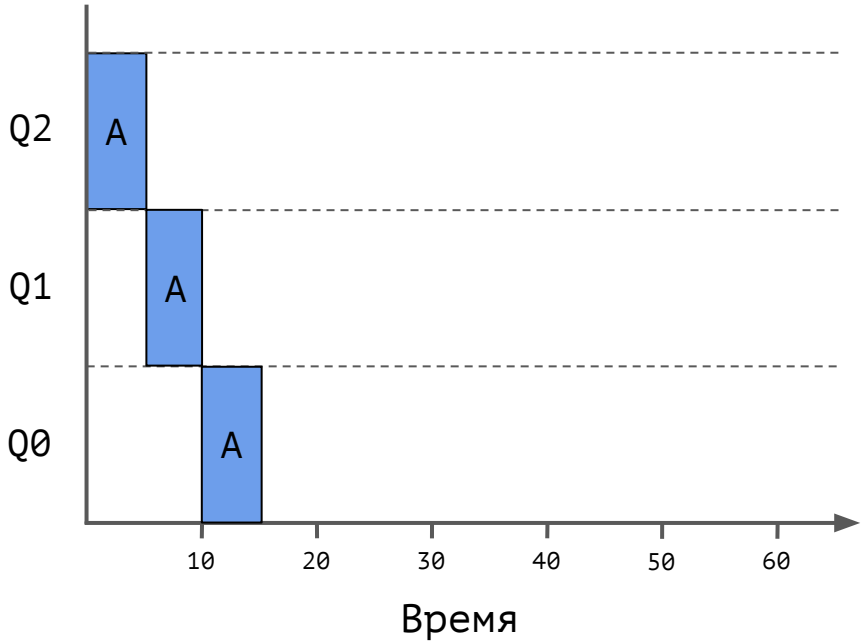
Пример 3: долгий пакетный таск и долгий интерактивный таск

MLFQ: изменение приоритетов

Пусть во время 0 приходит таск A продолжительности 50ms, а во время 15 приходит B, который работает 100ms, но каждые 2ms начинает ждать IO еще 3 ms.

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



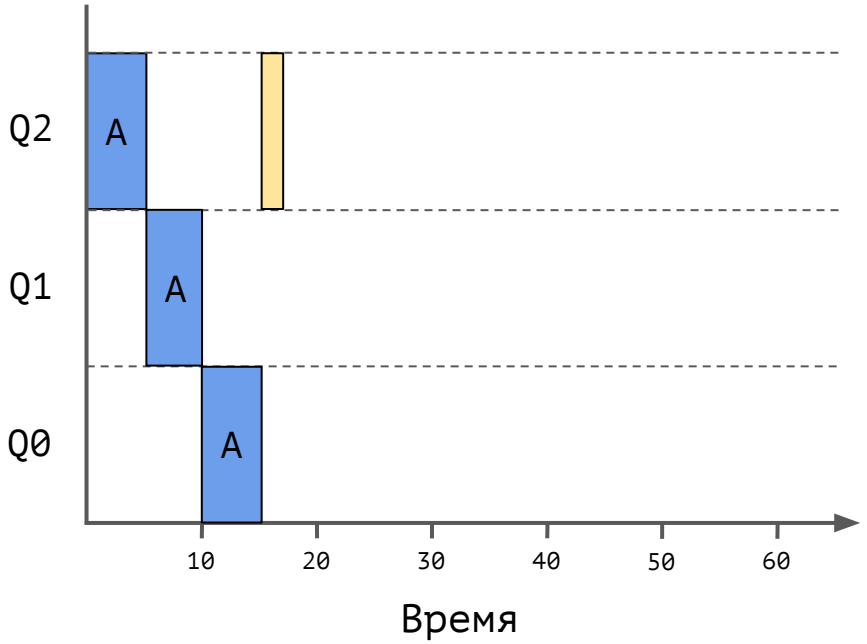
Пример 3: долгий пакетный таск и долгий интерактивный таск

MLFQ: изменение приоритетов

Пусть во время 0 приходит таск A продолжительности 50ms, а во время 15 приходит B, который работает 100ms, но каждые 2ms начинает ждать IO еще 3 ms.

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



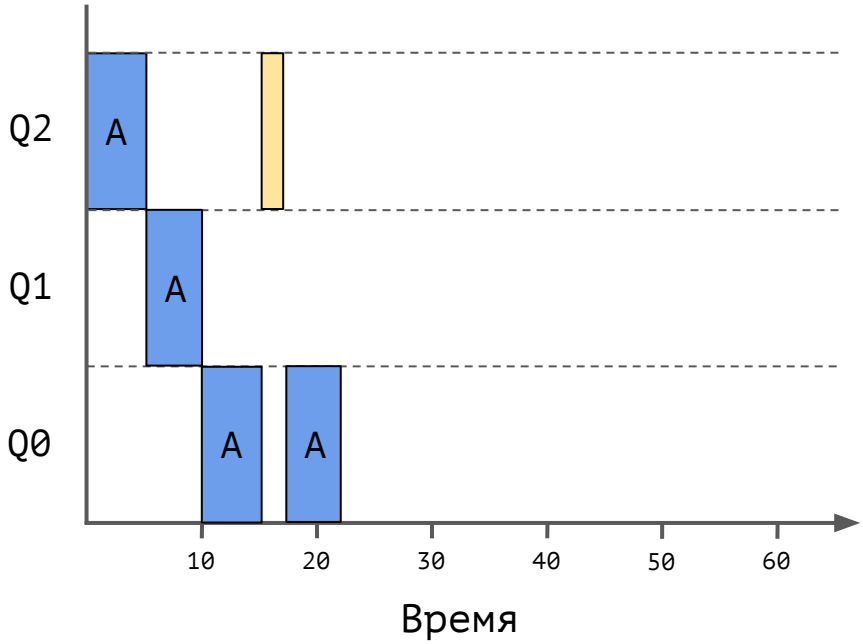
Пример 3: долгий пакетный таск и долгий интерактивный таск

MLFQ: изменение приоритетов

… B, который работает 100ms, но каждые 2ms начинает ждать IO еще 3 ms => он не отрабатывает весь квант, поэтому остается в Q0

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



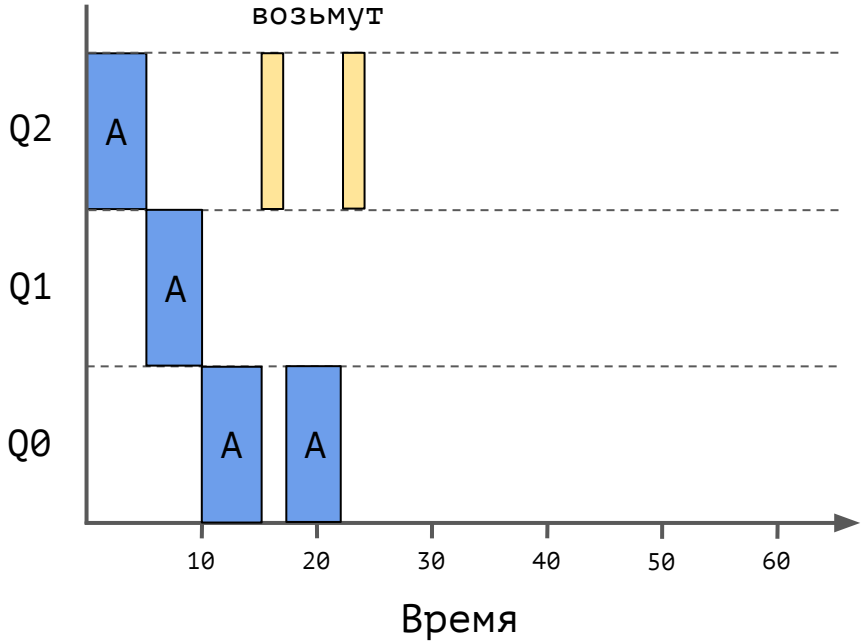
Пример 3: долгий пакетный таск и долгий интерактивный таск

MLFQ: изменение приоритетов

… B, который работает 100ms, но каждые 2ms начинает ждать IO еще 3 ms => он не отрабатывает весь квант, поэтому остается в Q0 => при следующем выборе его снова возьмут

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



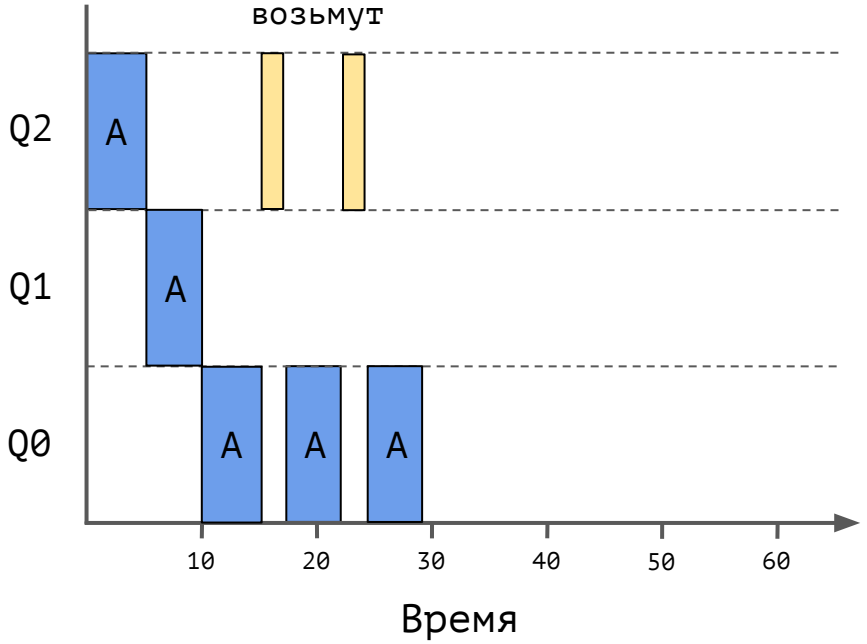
Пример 3: долгий пакетный таск и долгий интерактивный таск

MLFQ: изменение приоритетов

… B, который работает 100ms, но каждые 2ms начинает ждать IO еще 3 ms => он не отрабатывает весь квант, поэтому остается в Q0 => при следующем выборе его снова возьмут

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



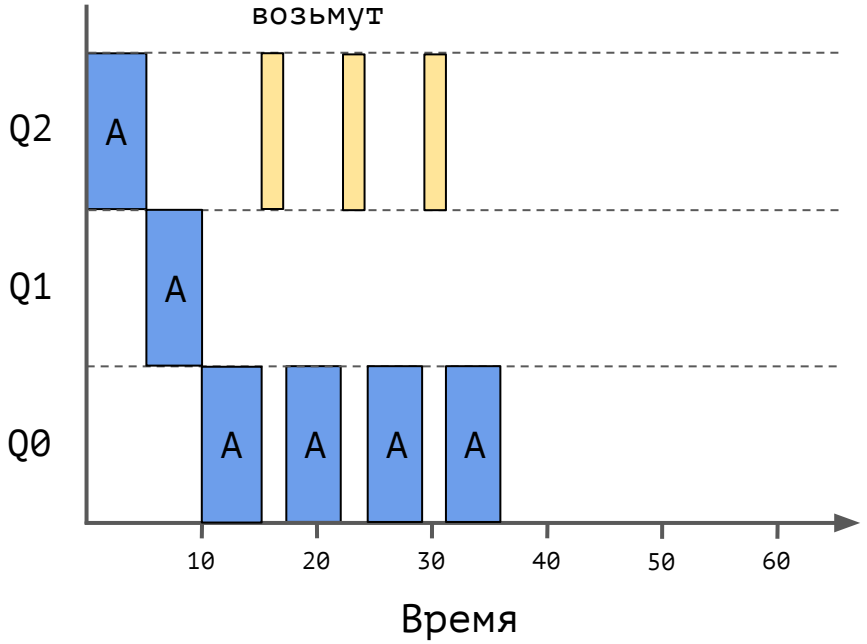
Пример 3: долгий пакетный таск и долгий интерактивный таск

MLFQ: изменение приоритетов

… B, который работает 100ms, но каждые 2ms начинает ждать IO еще 3 ms => он не отрабатывает весь квант, поэтому остается в Q0 => при следующем выборе его снова возьмут

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



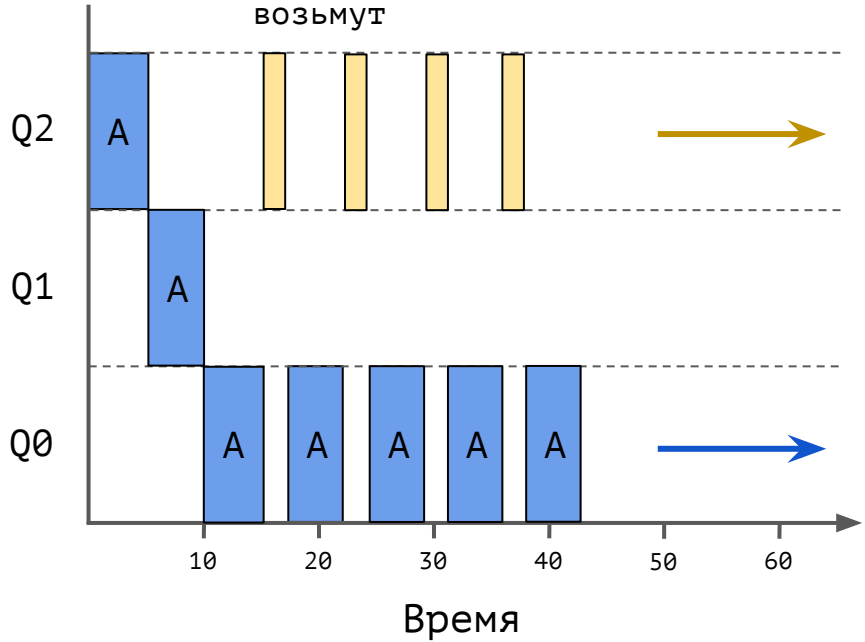
Пример 3: долгий пакетный таск и долгий интерактивный таск

MLFQ: изменение приоритетов

… B, который работает 100ms, но каждые 2ms начинает ждать IO еще 3 ms => он не отрабатывает весь квант, поэтому остается в Q0 => при следующем выборе его снова возьмут

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



Пример 3: долгий пакетный таск и долгий интерактивный таск

Интерактивная задача все время получает процессорное время => адаптировались

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~ FCFS
2. ~~Все задания поступают с одно и то же время;~~ SJF
3. ~~Задания доработывают до конца непрерывно;~~ STCF
4. ~~Задания используют только CPU (никакого IO);~~ STCF+
5. Время работы каждого задания известно заранее.

Метрики:

1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$
2. Справедливость;
3. [Среднее] время отклика: $T^k_{response} = T^k_{firstrun} - T^k_{arrival}$

RR дает и хорошее время отклика и безусловную справедливость (ценой оборотного времени)

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~ FCFS
2. ~~Все задания поступают с одно и то же время;~~ SJF
3. ~~Задания дорабатывают до конца непрерывно;~~ STCF
4. ~~Задания используют только CPU (никакого IO);~~ STCF+
5. ~~Время работы каждого задания известно заранее.~~ MLFQ

Метрики:

👍 1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$

2. Справедливость;

👍 3. [Среднее] время отклика: $T^k_{response} = T^k_{firstrun} - T^k_{arrival}$

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~ FCFS
2. ~~Все задания поступают с одно и то же время;~~ SJF
3. ~~Задания доработывают до конца непрерывно;~~ STCF
4. ~~Задания используют только CPU (никакого IO);~~ STCF+
5. ~~Время работы каждого задания известно заранее.~~ MLFQ

Метрики:

👍 1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$

2. Справедливость;

👍 3. [Среднее] время отклика: $T^k_{response} = T^k_{firstrun} - T^k_{arrival}$

Ну и плюс: MLFQ позволяет интерактивным задачам оставаться в топе, т.е. не только в начале время отклика хорошее, но и дальше.

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~ FCFS
2. ~~Все задания поступают с одно и то же время;~~ SJF
3. ~~Задания доработывают до конца непрерывно;~~ STCF
4. ~~Задания используют только CPU (никакого IO);~~ STCF+
5. ~~Время работы каждого задания известно заранее.~~ MLFQ

Метрики:

👍 1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$

2. Справедливость;

👍 3. [Среднее] время отклика: $T^k_{response} = T^k_{firstrun} - T^k_{arrival}$

Ну и плюс: MLFQ позволяет интерактивным задачам оставаться в топе, т.е. не только в начале время отклика хорошее, но и дальше. А есть ли какие-нибудь проблемы?

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~ FCFS
2. ~~Все задания поступают с одно и то же время;~~ SJF
3. ~~Задания доработывают до конца непрерывно;~~ STCF
4. ~~Задания используют только CPU (никакого IO);~~ STCF+
5. ~~Время работы каждого задания известно заранее.~~ MLFQ

Метрики:

👍 1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$

2. **Справедливость** ;

👍 3. [Среднее] время отклика: $T^k_{response} = T^k_{firstrun} - T^k_{arrival}$

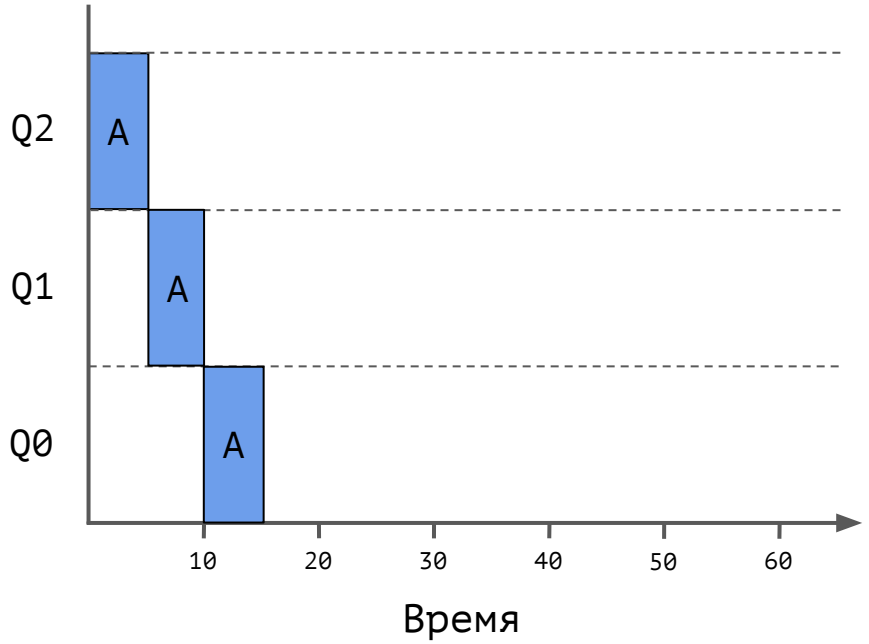
Ну и плюс: MLFQ позволяет интерактивным задачам оставаться в топе, т.е. не только в начале время отклика хорошее, но и дальше. А есть ли какие-нибудь проблемы?

MLFQ: изменение приоритетов

Пусть во время 0 приходит таск A продолжительности 200ms, а во время 15 приходит B, который работает 10ms

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



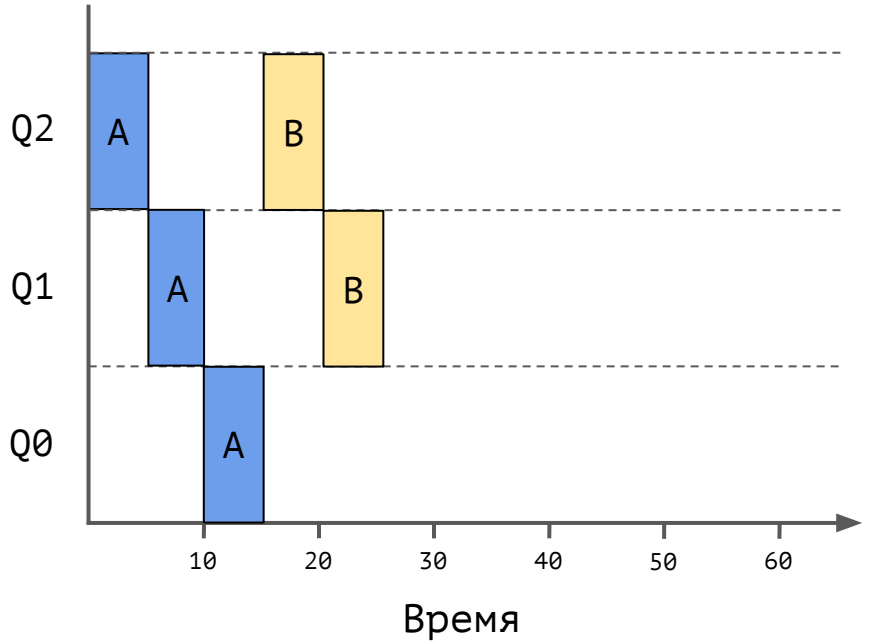
Пример 4: голодание долгого таска

MLFQ: изменение приоритетов

Пусть во время 0 приходит таск A продолжительности 200ms, а во время 15 приходит B, который работает 10ms

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



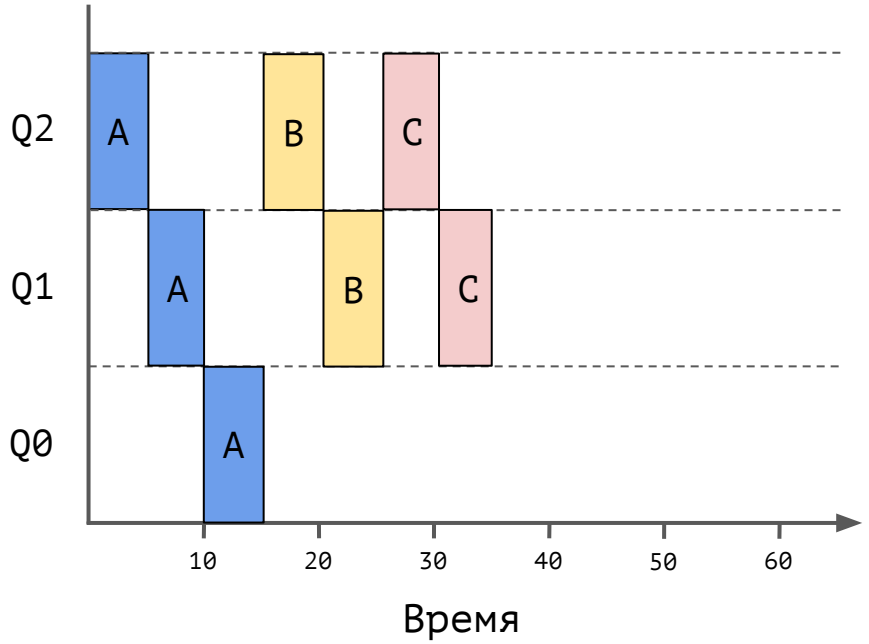
Пример 4: голодание долгого таска

MLFQ: изменение приоритетов

Пусть во время 0 приходит таск A продолжительности 200ms, а во время 15 приходит B, который работает 10ms, а потом таск C, который работает 10ms,...

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



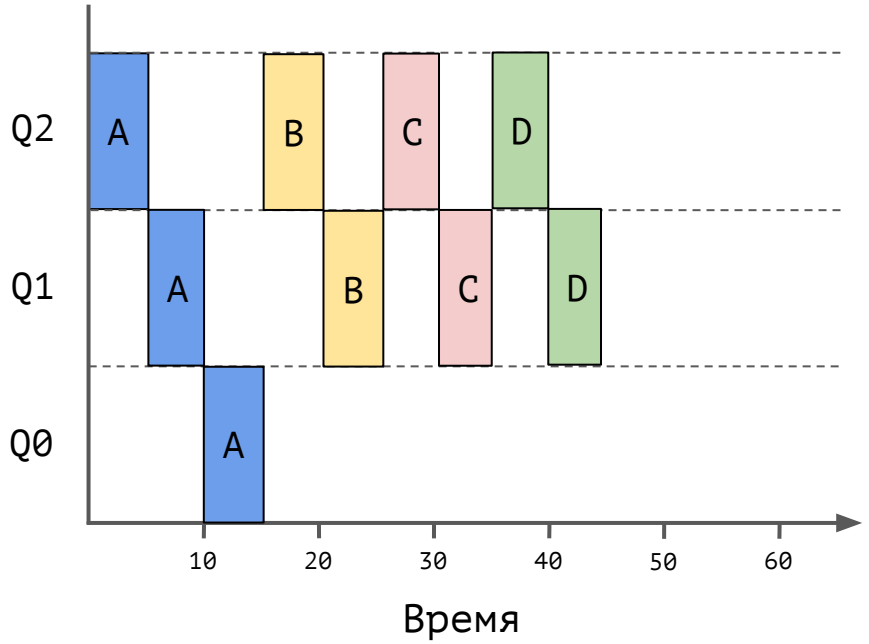
Пример 4: голодание долгого таска

MLFQ: изменение приоритетов

Пусть во время 0 приходит таск A продолжительности 200ms, а во время 15 приходит B, который работает 10ms, а потом таск C, который работает 10ms,...

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



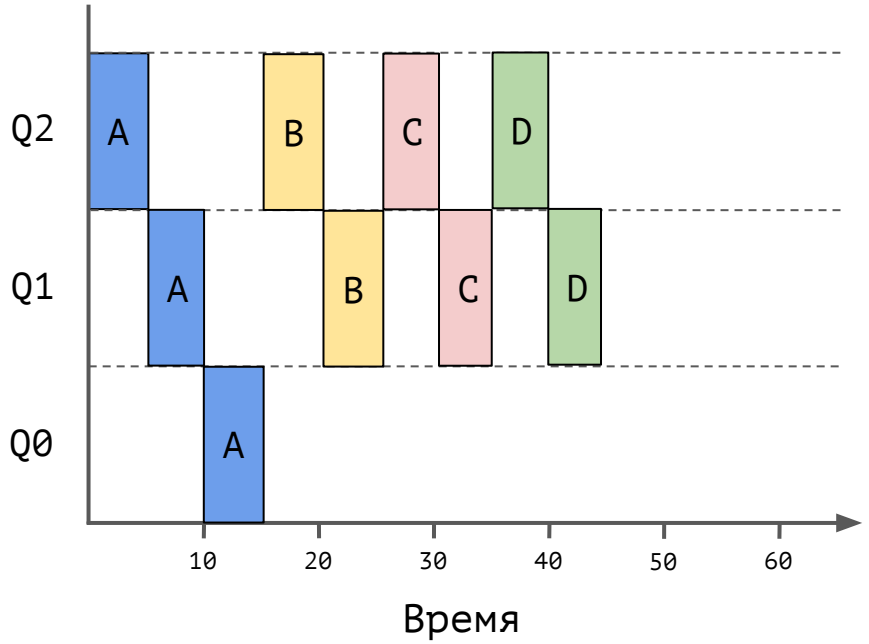
Пример 4: голодание долгого таска

MLFQ: изменение приоритетов

Пусть во время 0 приходит таск A продолжительности 200ms, а во время 15 приходит B, который работает 10ms, а потом таск C, который работает 10ms,...

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



Пример 4: голодание долгого таска

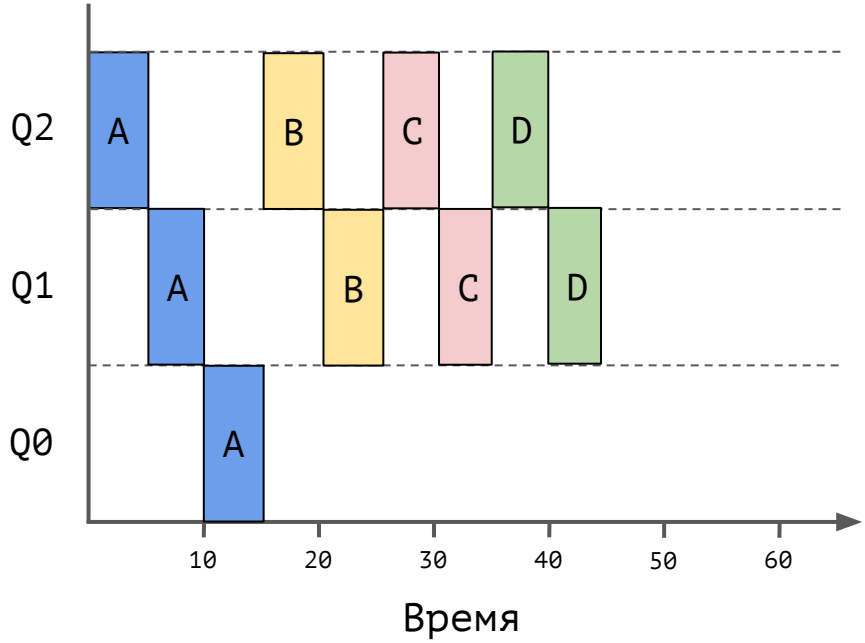
Задача A может так вообще никогда не получить процессорное время => голодание. Что делать?

MLFQ: изменение приоритетов

Пусть во время 0 приходит таск A продолжительности 200ms, а во время 15 приходит B, который работает 10ms, а потом таск C, который работает 10ms,...

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .
4. Раз в некоторый промежуток времени поднимать всем таскам приоритет до



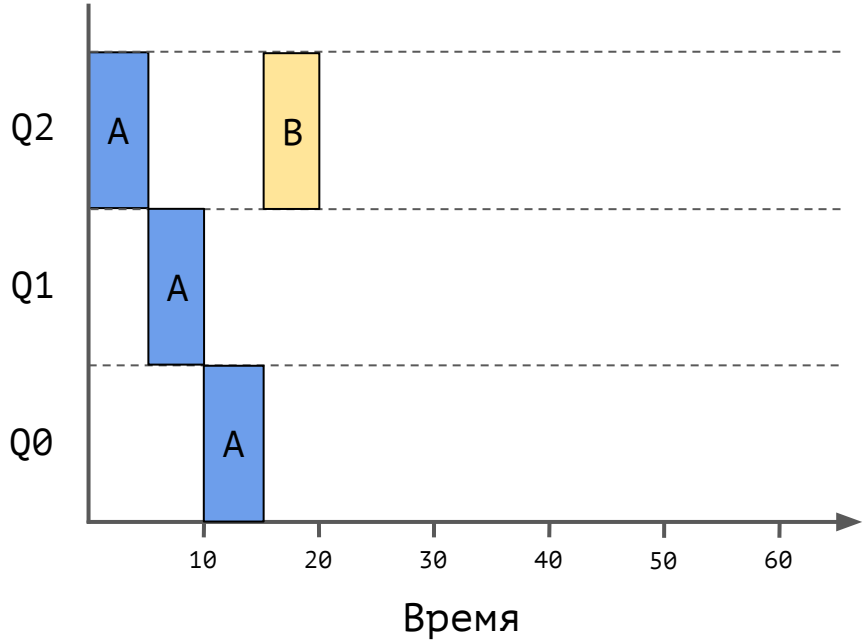
Пример 4: голодание долгого таска

MLFQ: изменение приоритетов

Пусть во время 0 приходит таск A продолжительности 200ms, а во время 15 приходит B, который работает 10ms, а потом таск C, который работает 10ms,...

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .
4. Раз в некоторый промежуток времени поднимать всем таскам приоритет до максимума



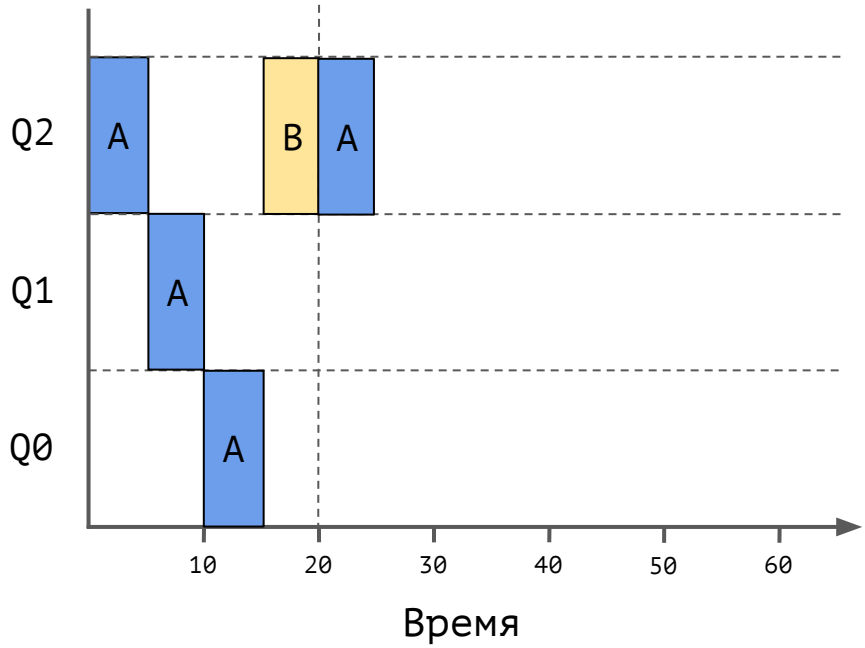
Пример 5: спасение голодающих (решение принимается раз в 20 ms)

MLFQ: изменение приоритетов

… а во время 15 приходит B, который работает 10ms, а потом таск C, который работает 10ms, но во время 20ms A снова получает высший приоритет!

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .
4. Раз в некоторый промежуток времени поднимать всем таскам приоритет до максимума



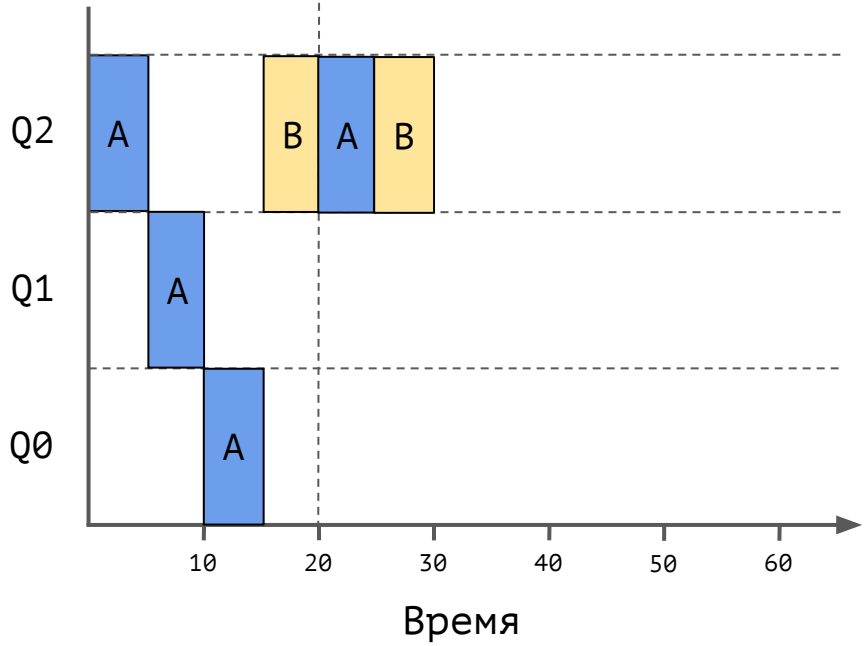
Пример 5: спасение голодающих (решение принимается раз в 20 ms)

MLFQ: изменение приоритетов

… а во время 15 приходит B, который работает 10ms, а потом таск C, который работает 10ms, но во время 20ms A (ну и B) снова получают высший приоритет!

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .
4. Раз в некоторый промежуток времени поднимать всем таскам приоритет до максимума



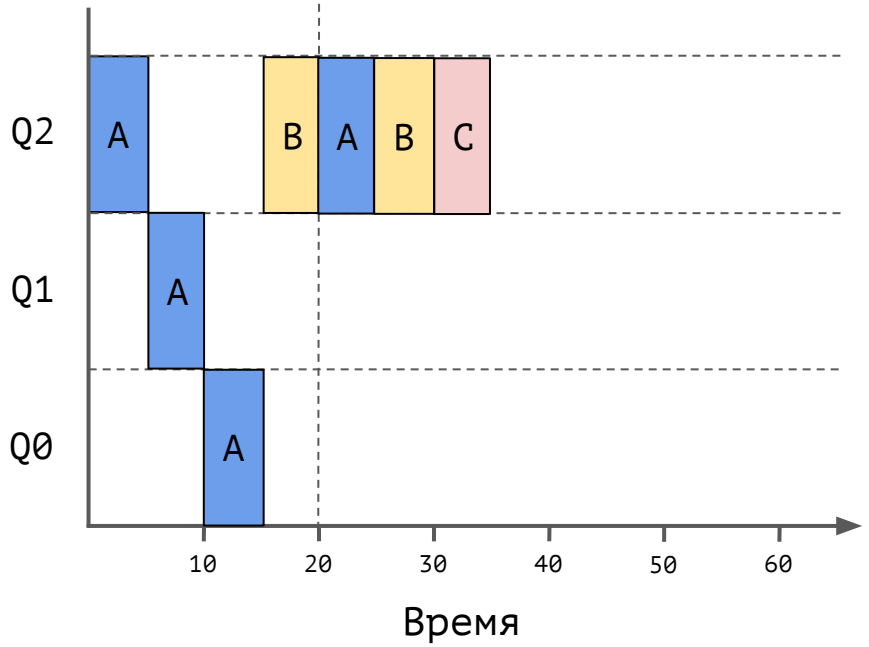
Пример 5: спасение голодающих (решение принимается раз в 20 ms)

MLFQ: изменение приоритетов

… а во время 15 приходит B, который работает 10ms, а потом таск C, который работает 10ms, но во время 20ms A (ну и B) снова получают высший приоритет!

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .
4. Раз в некоторый промежуток времени поднимать всем таскам приоритет до максимума



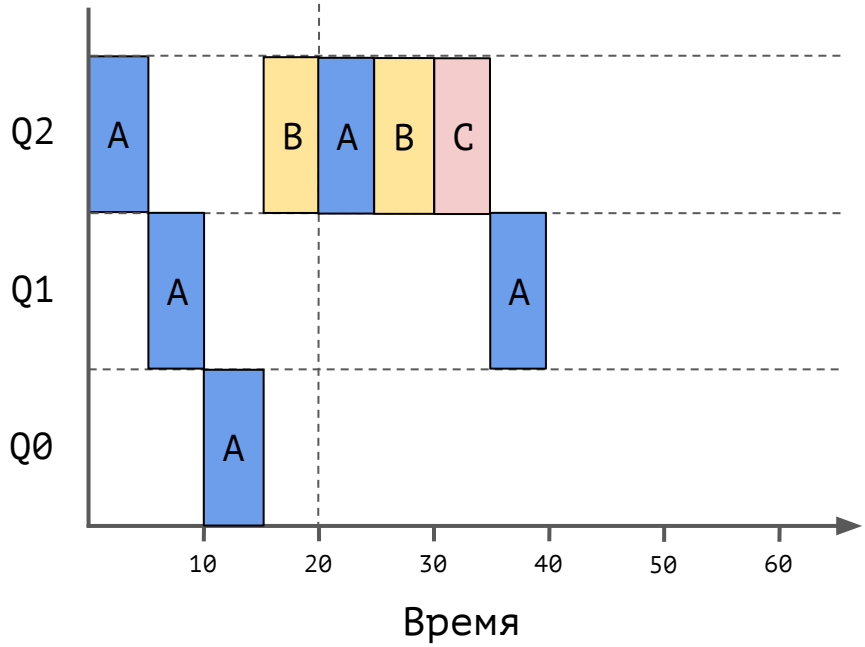
Пример 5: спасение голодающих (решение принимается раз в 20 ms)

MLFQ: изменение приоритетов

… а во время 15 приходит B, который работает 10ms, а потом таск C, который работает 10ms, но во время 20ms A (ну и B) снова получают высший приоритет!

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .
4. Раз в некоторый промежуток времени поднимать всем таскам приоритет до максимума



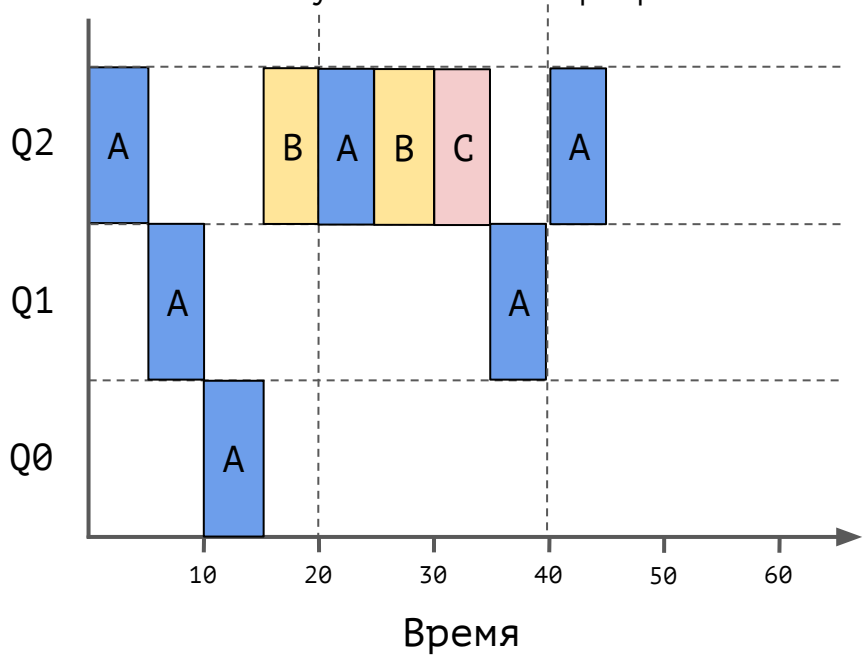
Пример 5: спасение голодающих (решение принимается раз в 20 ms)

MLFQ: изменение приоритетов

… а во время 15 приходит B, который работает 10ms, а потом таск C, который работает 10ms, но во время 40ms A снова получает высший приоритет!

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .
4. Раз в некоторый промежуток времени поднимать всем таскам приоритет до максимума



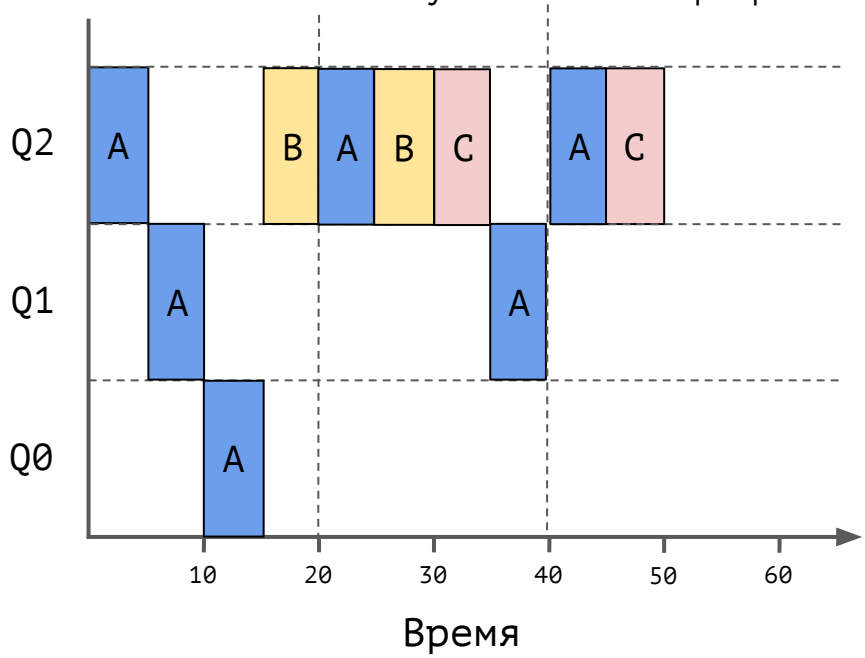
Пример 5: спасение голодающих (решение принимается раз в 20 ms)

MLFQ: изменение приоритетов

… а во время 15 приходит B, который работает 10ms, а потом таск C, который работает 10ms, но во время 40ms A (ну и C) снова получают высший приоритет!

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .
4. Раз в некоторый промежуток времени поднимать всем таскам приоритет до максимума



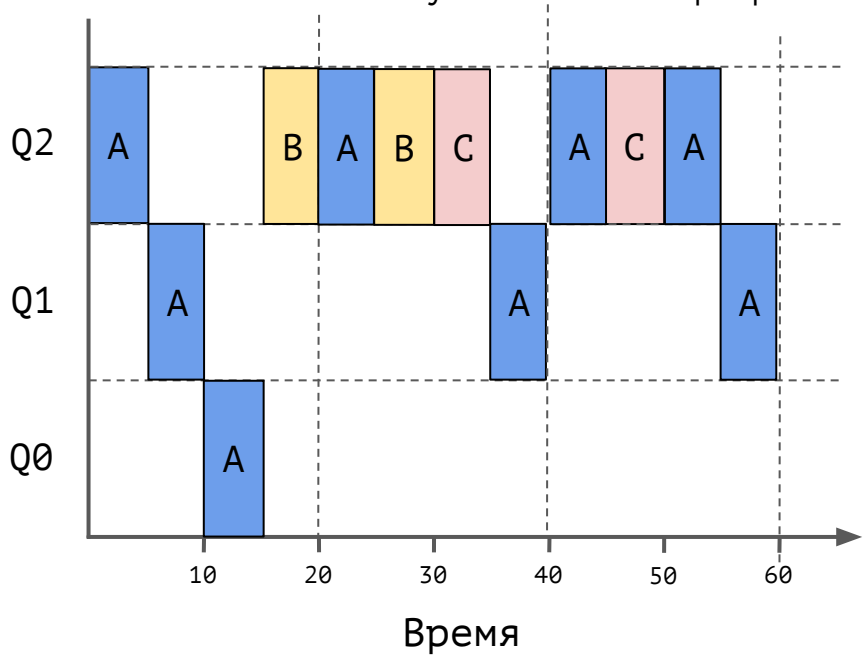
Пример 5: спасение голодающих (решение принимается раз в 20 ms)

MLFQ: изменение приоритетов

… а во время 15 приходит B, который работает 10ms, а потом таск C, который работает 10ms, но во время 40ms A (ну и C) снова получают высший приоритет!

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .
4. Раз в некоторый промежуток времени поднимать всем таскам приоритет до максимума



Пример 5: спасение голодающих (решение принимается раз в 20 ms)

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~ FCFS
2. ~~Все задания поступают с одно и то же время;~~ SJF
3. ~~Задания доработывают до конца непрерывно;~~ STCF
4. ~~Задания используют только CPU (никакого IO);~~ STCF+
5. ~~Время работы каждого задания известно заранее.~~ MLFQ

Метрики:

👍 1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$

2. **Справедливость** ;

👍 3. [Среднее] время отклика: $T^k_{response} = T^k_{firstrun} - T^k_{arrival}$

Ну и плюс: MLFQ позволяет интерактивным задачам оставаться в топе, т.е. не только в начале время отклика хорошее, но и дальше. А есть ли какие-нибудь проблемы?

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~ FCFS
2. ~~Все задания поступают с одно и то же время;~~ SJF
3. ~~Задания доработывают до конца непрерывно;~~ STCF
4. ~~Задания используют только CPU (никакого IO);~~ STCF+
5. ~~Время работы каждого задания известно заранее.~~ MLFQ

Метрики:

👍 1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$

👍 2. Справедливость;

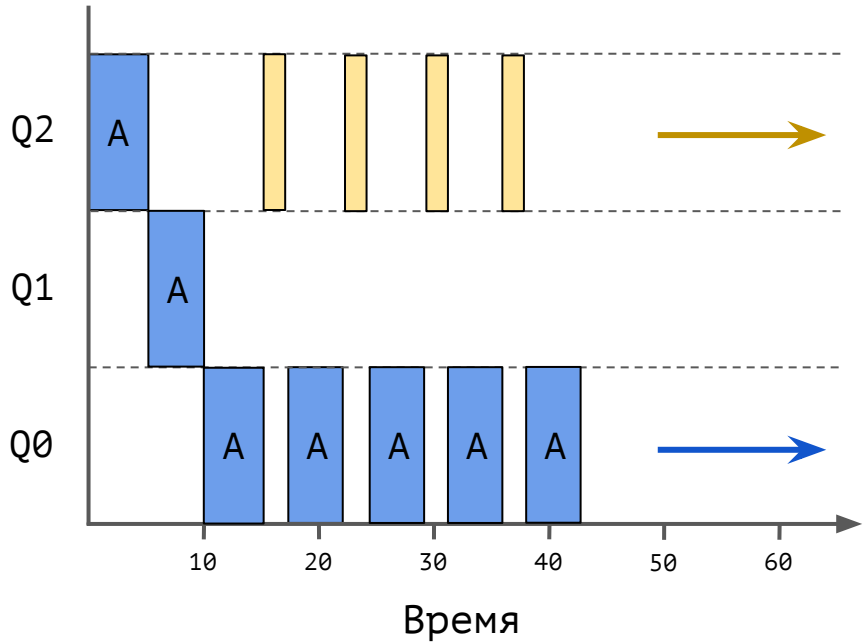
👍 3. [Среднее] время отклика: $T^k_{response} = T^k_{firstrun} - T^k_{arrival}$

Ну и плюс: MLFQ позволяет интерактивным задачам оставаться в топе, т.е. не только в начале время отклика хорошее, но и дальше. А есть ли еще какие-нибудь проблемы?

MLFQ: защита

Правила изменения приоритетов:

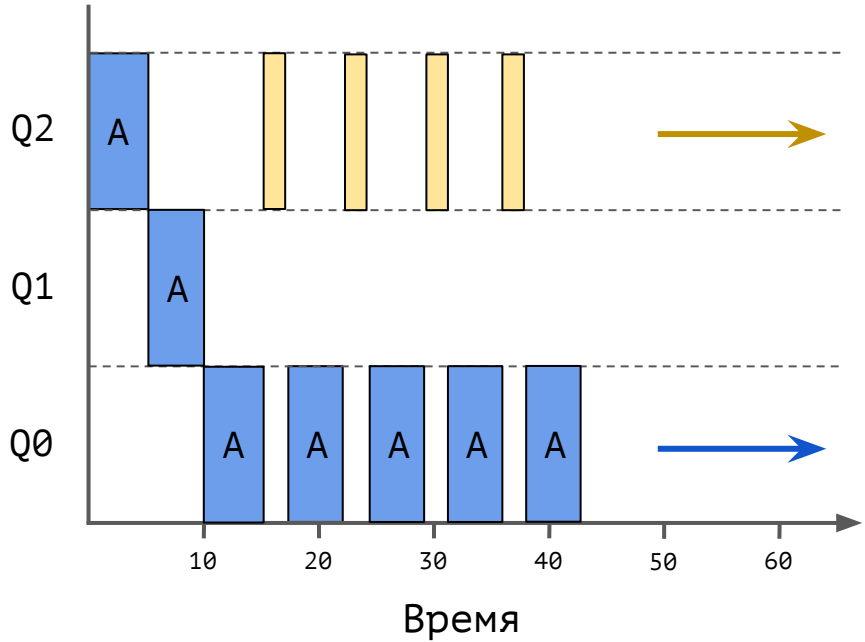
1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .



MLFQ: защита от псевдо-интерактивных задач

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Если задание отработало весь свой квант времени, его приоритет **понижается** ;
3. Если задание добровольно отдало CPU (но не прекратилось), его приоритет **остается прежним** .

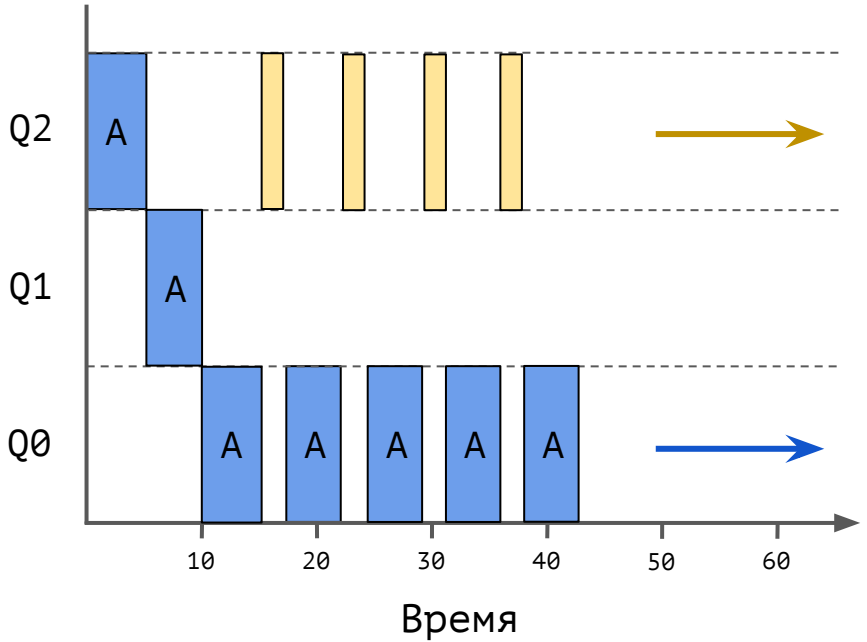


Зная вот этот пункт, можно абьюзить планировщик: ненадолго отдавать CPU в самом конце кванта и не терять приоритет

MLFQ: защита от псевдо-интерактивных задач

Правила изменения приоритетов:

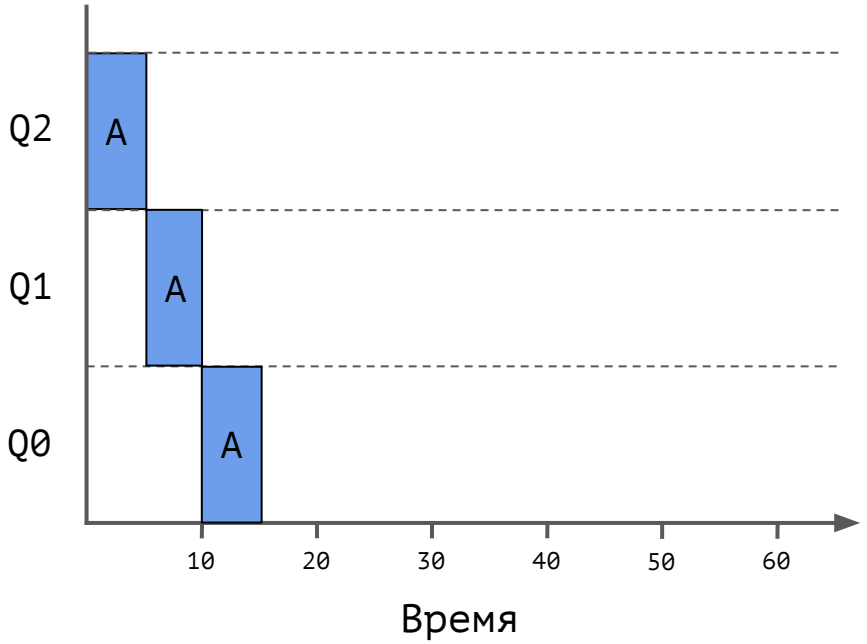
1. Задания поступают с наивысшим приоритетом;
2. Как только задание отреботало весь квант на текущем приоритете (неважно, сколько раз оно отдавало CPU), понижать приоритет



MLFQ: защита от псевдо-интерактивных задач

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Как только задание отреботало весь квант на текущем приоритете (неважно, сколько раз оно отдавало CPU), понижать приоритет

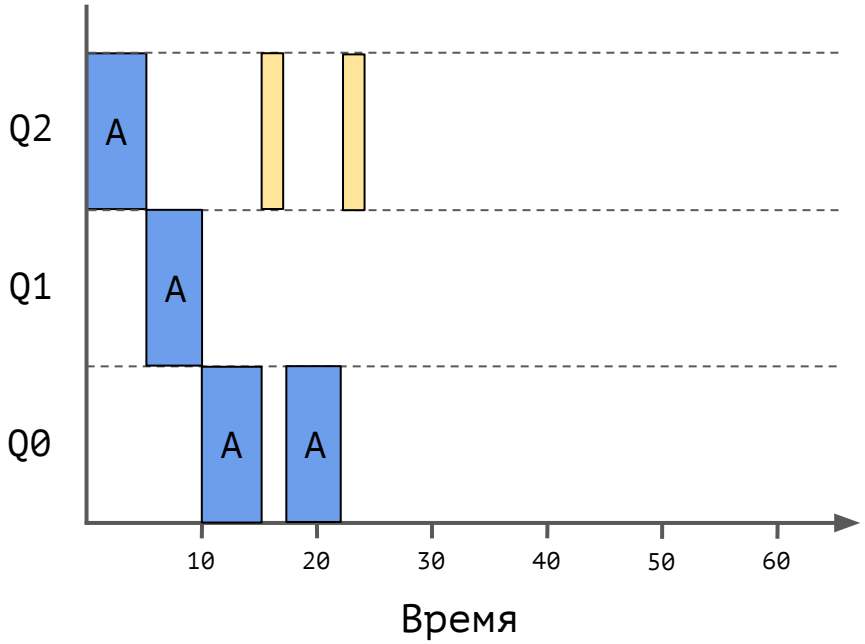


Пример 6: долгий пакетный таск и долгий интерактивный таск (оба тонут)

MLFQ: защита от псевдо-интерактивных задач

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Как только задание отреботало весь квант на текущем приоритете (неважно, сколько раз оно отдавало CPU), понижать приоритет

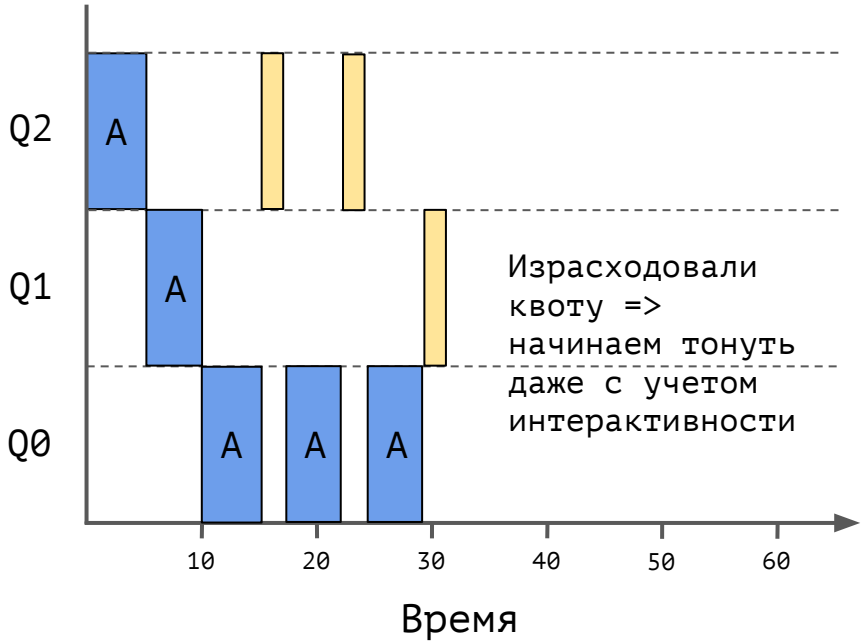


Пример 6: долгий пакетный таск и долгий интерактивный таск (оба тонут)

MLFQ: защита от псевдо-интерактивных задач

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Как только задание отреботало весь квант на текущем приоритете (неважно, сколько раз оно отдавало CPU), понижать приоритет

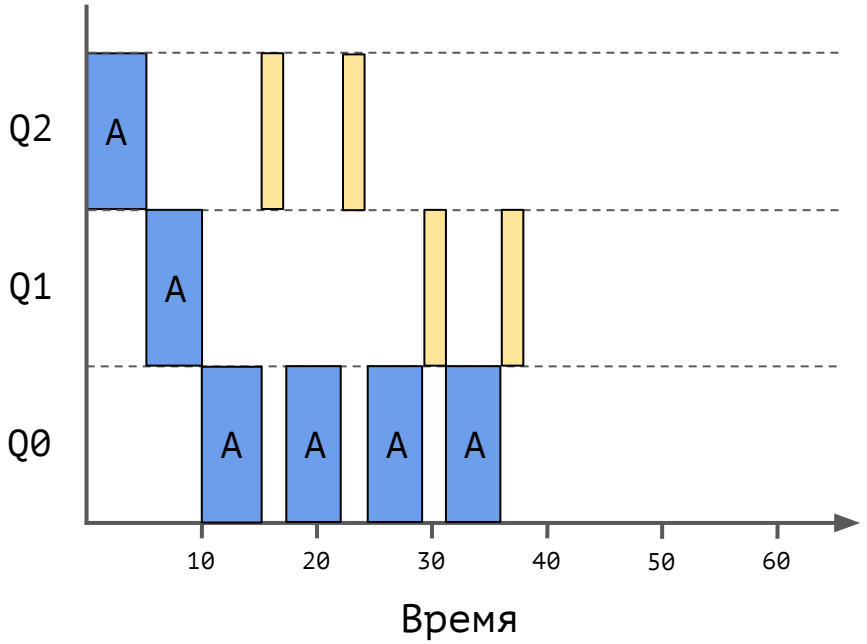


Пример 6: долгий пакетный таск и долгий интерактивный таск (оба тонут)

MLFQ: защита от псевдо-интерактивных задач

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Как только задание отреботало весь квант на текущем приоритете (неважно, сколько раз оно отдавало CPU), понижать приоритет

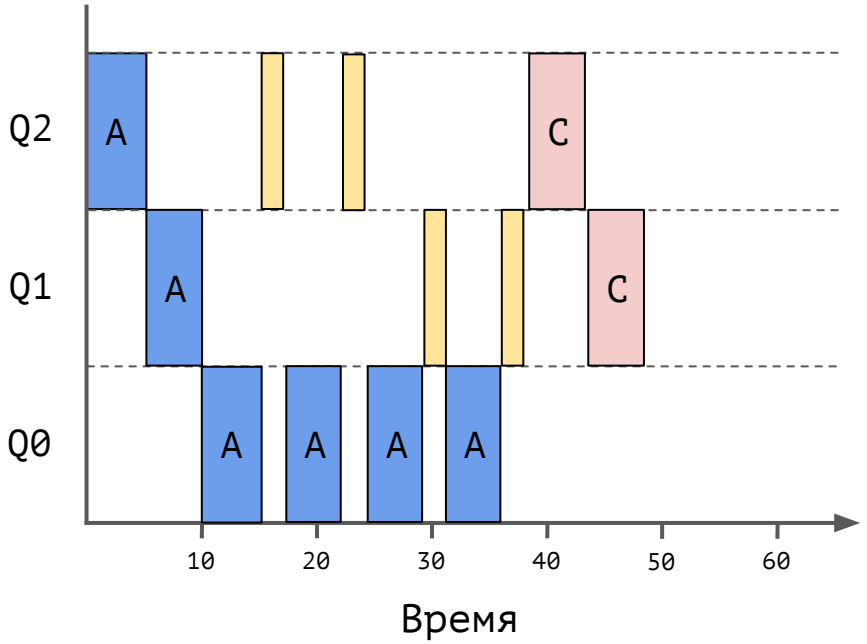


Пример 6: долгий пакетный таск и долгий интерактивный таск (оба тонут)

MLFQ: защита от псевдо-интерактивных задач

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Как только задание отреботало весь квант на текущем приоритете (неважно, сколько раз оно отдавало CPU), понижать приоритет

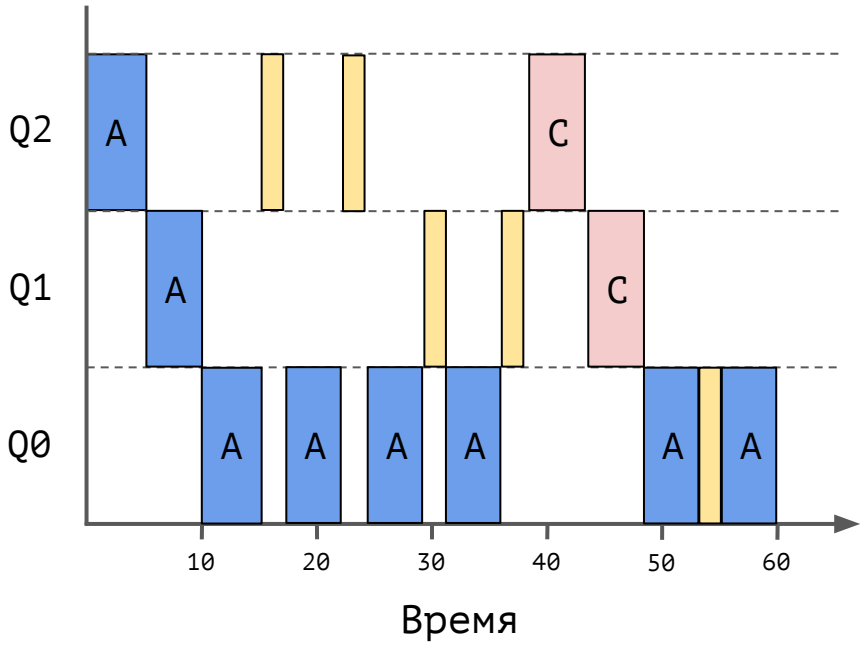


Пример 6: долгий пакетный таск и долгий интерактивный таск (оба тонут)

MLFQ: защита от псевдо-интерактивных задач

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Как только задание отреботало весь квант на текущем приоритете (неважно, сколько раз оно отдавало CPU), понижать приоритет



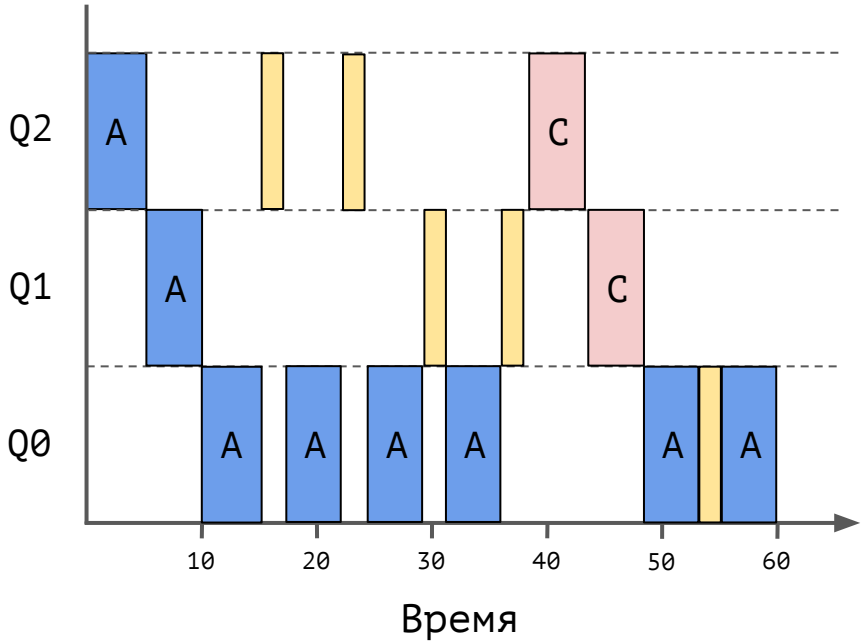
Пример 6: долгий пакетный таск и долгий интерактивный таск (оба тонут)

Такое заабьюзить не получится, все равно утонете.

MLFQ: защита от псевдо-интерактивных задач

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Как только задание отреботало весь квант на текущем приоритете (неважно, сколько раз оно отдавало CPU), понижать приоритет



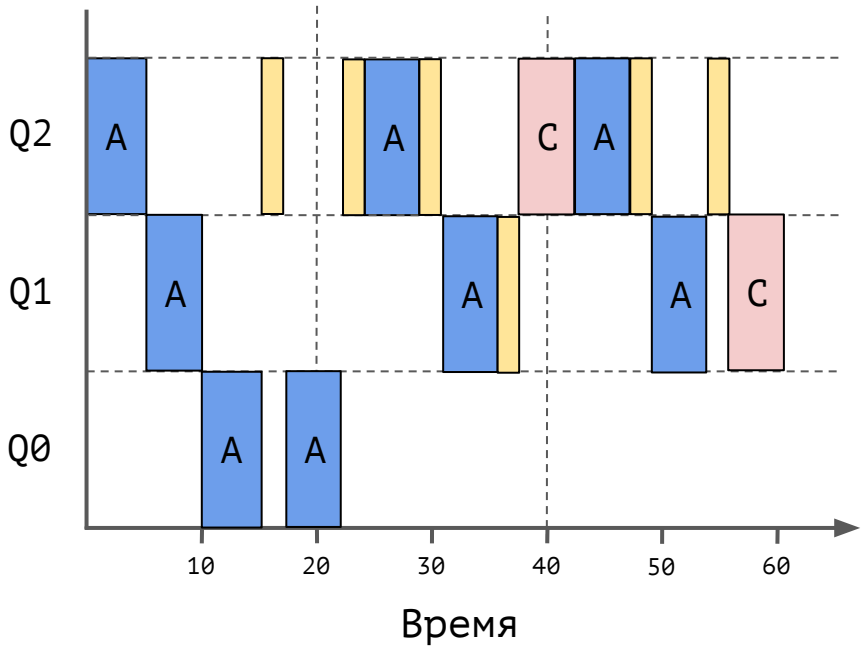
Пример 6: долгий пакетный таск и долгий интерактивный таск (оба тонут)

Такое заабьюзить не получится, все равно утонете. Но как же справедливость??

MLFQ: защита от псевдо-интерактивных задач

Правила изменения приоритетов:

1. Задания поступают с наивысшим приоритетом;
2. Как только задание отреботало весь квант на текущем приоритете (неважно, сколько раз оно отдавало CPU), понижать приоритет
3. Раз в некоторый промежуток времени поднимать всем таскам приоритет до максимума

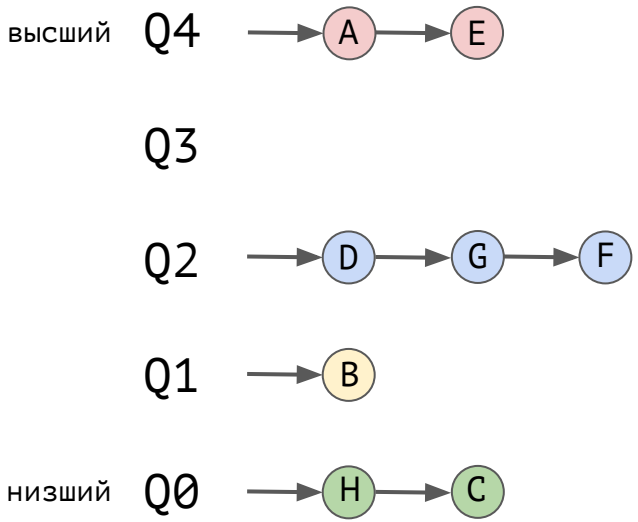


Пример 7: долгий пакетный таск и долгий интерактивный таск (оба тонут, но голодающих спасают)

MLFQ: замечания по реализации

Замечания:

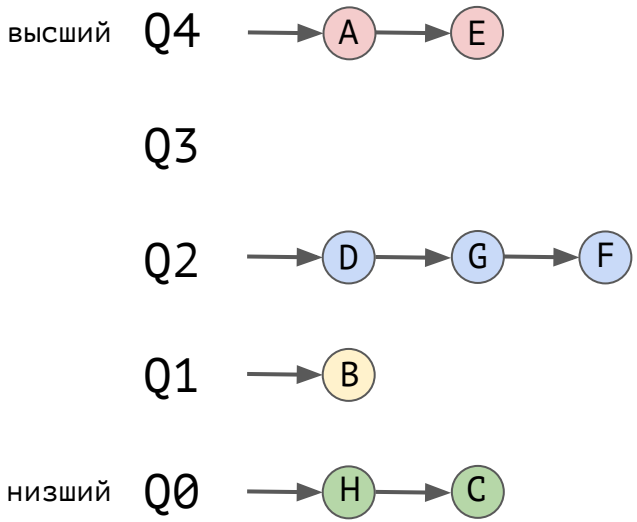
- Длительность кванта, время поднятия приоритетов, количество очередей - все это настраивается и подбирается под конкретную задачу;



MLFQ: замечания по реализации

Замечания:

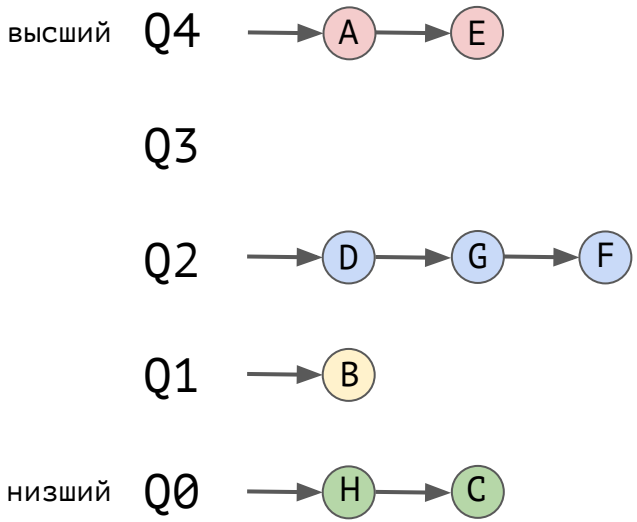
- Длительность кванта, время поднятия приоритетов, количество очередей - все это настраивается и подбирается под конкретную задачу;
- Длительность кванта может отличаться в зависимости от приоритета (ниже - больше)
- Некоторые приоритеты могут быть зарезервированы под особые нужды ОС



MLFQ: замечания по реализации

Замечания:

- Длительность кванта, время поднятия приоритетов, количество очередей - все это настраивается и подбирается под конкретную задачу;
- Длительность кванта может отличаться в зависимости от приоритета (ниже - больше)
- Некоторые приоритеты могут быть зарезервированы под особые нужды ОС



- Неустаревающая классика: использовали во многих старых OS (Solaris, BSD Unix, Windows NT), идеи используют и сейчас

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Ограничения глобальной задачи:

1. ~~Время выполнения всех заданий одинаково;~~ FCFS
2. ~~Все задания поступают с одно и то же время;~~ SJF
3. ~~Задания доработывают до конца непрерывно;~~ STCF
4. ~~Задания используют только CPU (никакого IO);~~ STCF+
5. ~~Время работы каждого задания известно заранее.~~ MLFQ

Метрики:

👍 1. [Среднее] оборотное время: $T^k_{turnaround} = T^k_{completion} - T^k_{arrival}$

👍 2. Справедливость;

👍 3. [Среднее] время отклика: $T^k_{response} = T^k_{firstrun} - T^k_{arrival}$

Ну и плюс: MLFQ позволяет интерактивным задачам оставаться в топе, т.е. не только в начале время отклика хорошее, но и дальше.

Вам не кажется, что все как-то слишком сложно?

Вам не кажется, что все как-то слишком сложно?

Очереди, политики сменов приоритетов, оптимизация оборотно времени и времени отклика...

Вам не кажется, что все как-то слишком сложно?

Очереди, политики сменов приоритетов, оптимизация оборотно времени и времени отклика... и ведь про все это нужно думать на каждом кванте или даже чаще!

Вам не кажется, что все как-то слишком сложно?

Очереди, политики сменов приоритетов, оптимизация оборотно времени и времени отклика... и ведь про все это нужно думать на каждом кванте или даже чаще!

Давайте попробуем еще раз.

Задача планирования

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Задача пропорционального планирования

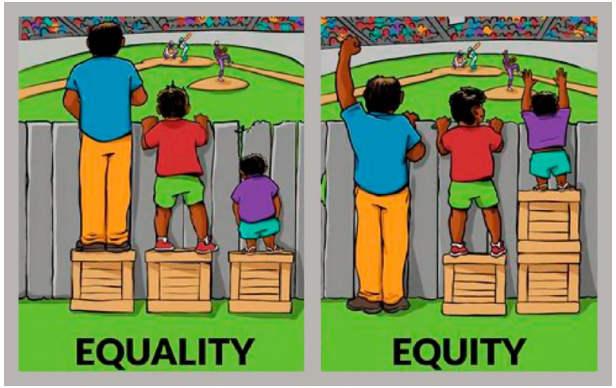
Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Идея: попробуем гарантировать, что каждое задание получает определенную долю процессорного времени.

Задача пропорционального планирования

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Идея: попробуем гарантировать, что каждое задание получает определенную долю процессорного времени. В частном случае это задача равномерного планирования, но такое далеко не всегда нужно.



Лотерейное планирование

Идея: попробуем гарантировать, что каждое задание получает определенную долю процессорного времени.

Реализация: раздадим процессам лотерейные билеты. Если процесс должен занимать CPU чаще других (например из-за более высокого приоритета), у него будет больше билетов.

Лотерейное планирование

Идея: попробуем гарантировать, что каждое задание получает определенную долю процессорного времени.

Реализация: раздадим процессам лотерейные билеты. Если процесс должен занимать CPU чаще других (например из-за более высокого приоритета), у него будет больше билетов.

После очередного кванта проводим лотерею! Какой процесс в нее выиграл, тот и занимает CPU.

Лотерейное планирование

Пример: пусть есть три процесса A, B и C. A и B выдали по 25 билетов (от 0 до 24 и от 25 до 49 соответственно), а второму - 50 (от 50 до 99)

Лотерейное планирование

Пример: пусть есть три процесса A, B и C. A и B выдали по 25 билетов (от 0 до 24 и от 25 до 49 соответственно), а второму - 50 (от 50 до 99)

Каждый квант выбираем случайное число от 0 до 99.

13 4 95 49 20 8 83 98 26 82 94 67 61 14 33 16 57 70 91 36 48

Лотерейное планирование

Пример: пусть есть три процесса A, B и C. A и B выдали по 25 билетов (от 0 до 24 и от 25 до 49 соответственно), а второму - 50 (от 50 до 99)

Каждый квант выбираем случайное число от 0 до 99.

```
13 4 95 49 20 8 83 98 26 82 94 67 61 14 33 16 57 70 91 36 48

 A A        A A                        A     A
        B              B                   B              B  B
     C          C  C     C  C  C  C             C  C  C
```

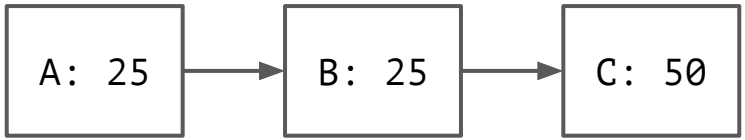
Получили то, что хотели: C исполнялся в ~2 раза чаще

Лотерейное планирование

Пример: пусть есть три процесса A, B и C. A и B выдали по 25 билетов (от 0 до 24 и от 25 до 49 соответственно), а второму - 50 (от 50 до 99)

Как реализовать?

Храним список процессов, для каждого количество его билетов.

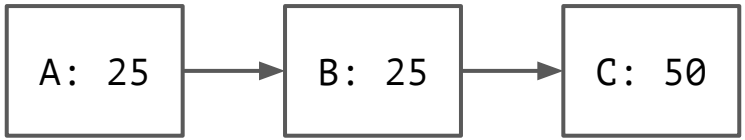


Лотерейное планирование

Пример: пусть есть три процесса A, B и C. A и B выдали по 25 билетов (от 0 до 24 и от 25 до 49 соответственно), а второму - 50 (от 50 до 99)

Как реализовать?

Храним список процессов, для каждого количество его билетов. Для принятия решения: 1) генерируем случайное число от 0 до количества билетов -1 => 2) проходим по списку, каждый раз прибавляя количество билетов => 3) как только превысили выбранное число - нашли победителя



Лотерейное планирование

Реализация: раздаем билеты и каждый раз проводим лотерею, чтобы понять, какой процесс займет CPU

Плюсы такого подхода?

Лотерейное планирование

Реализация: раздаем билеты и каждый раз проводим лотерею, чтобы понять, какой процесс займет CPU

Плюсы такого подхода:

1. Он справедлив! (в пределе)

Лотерейное планирование

Реализация: раздаем билеты и каждый раз проводим лотерею, чтобы понять, какой процесс займет CPU

Плюсы такого подхода:

1. Он справедлив! (в пределе)
2. Он очень прост! (а значит быстр)

Лотерейное планирование

Реализация: раздаем билеты и каждый раз проводим лотерею, чтобы понять, какой процесс займет CPU

Плюсы такого подхода:

1. Он справедлив! (в пределе)
2. Он очень прост! (а значит быстр)

3. Новые процессы могут моментально начать участвовать в лотереях (изменив шансы остальным)

Лотерейное планирование

Реализация: раздаем билеты и каждый раз проводим лотерею, чтобы понять, какой процесс займет CPU

Плюсы такого подхода:

1. Он справедлив! (в пределе)
2. Он очень прост! (а значит быстр)

3. Новые процессы могут моментально начать участвовать в лотереях (изменив шансы остальным)

4. Процессы могут проводить операции с билетами: передавать их друг другу, устраивать инфляцию, вводить валюту

Лотерейное планирование

Реализация: раздаем билеты и каждый раз проводим лотерею, чтобы понять, какой процесс займет CPU

Плюсы такого подхода:

1. Он справедлив! (в пределе)
2. Он очень прост! (а значит быстр)

3. Новые процессы могут моментально начать участвовать в лотереях (изменив шансы остальным)

4. ... операции с билетами

5. Он случайный!

Лотерейное планирование

Реализация: раздаем билеты и каждый раз проводим лотерею, чтобы понять, какой процесс займет CPU

Плюсы такого подхода:

1. Он справедлив! (в пределе)
2. Он очень прост! (а значит быстр)

3. Новые процессы могут моментально начать участвовать в лотереях (изменив шансы остальным)

4. ... операции с билетами

5. Он случайный! (нет контрпримеров)

Лотерейное планирование

Реализация: раздаем билеты и каждый раз проводим лотерею, чтобы понять, какой процесс займет CPU

Минусы такого подхода?

Лотерейное планирование

Реализация: раздаем билеты и каждый раз проводим лотерею, чтобы понять, какой процесс займет CPU

Минусы такого подхода:

1. Как раздавать билеты?? Просто верить пользователям?

Лотерейное планирование

Реализация: раздаем билеты и каждый раз проводим лотерею, чтобы понять, какой процесс займет CPU

Минусы такого подхода:

1. Как раздавать билеты?? Просто верить пользователям?
2. Что там с IO? Как учитывать IO-характер задач при лотерее?

Лотерейное планирование

Реализация: раздаем билеты и каждый раз проводим лотерею, чтобы понять, какой процесс займет CPU

Минусы такого подхода:

1. Как раздавать билеты?? Просто верить пользователям?
2. Что там с IO? Как учитывать IO-характер задач при лотерее?

3. Не всем подходит недетерминированность.

Лотерейное планирование

Реализация: раздаем билеты и каждый раз проводим лотерею, чтобы понять, какой процесс займет CPU

Минусы такого подхода:

1. Как раздавать билеты?? Просто верить пользователям?
2. Что там с IO? Как учитывать IO-характер задач при лотерее?

3. Не всем подходит недетерминированность. Впрочем, это легко исправить.

Шаговое планирование

Реализация: раздаем билеты процессам…

Шаговое планирование

Реализация: раздаем билеты процессам, но лотереи больше не проводим. Вместо этого для каждого процесса определяем шаг: величину обратно-пропорциональную количеству билетов.

Шаговое планирование

Реализация: раздаем билеты процессам, но лотереи больше не проводим. Вместо этого для каждого процесса определяем шаг: величину обратно-пропорциональную количеству билетов.

Например, пусть у процессов A, B и C было билетов 100, 50 и 250 соответственно. Тогда их шаги будут 100, 200 и 40.

Шаговое планирование

Реализация: раздаем билеты процессам, но лотереи больше не проводим. Вместо этого для каждого процесса определяем шаг: величину обратно-пропорциональную количеству билетов.

Например, пусть у процессов A, B и C было билетов 100, 50 и 250 соответственно. Тогда их шаги будут 100, 200 и 40.

Для каждого процесса введем текущий прогресс. За каждый квант работы его прогресс увеличивается на величину шага.

Шаговое планирование

Реализация: раздаем билеты процессам, но лотереи больше не проводим. Вместо этого для каждого процесса определяем шаг: величину обратно-пропорциональную количеству билетов.

Например, пусть у процессов A, B и C было билетов 100, 50 и 250 соответственно. Тогда их шаги будут 100, 200 и 40.

Для каждого процесса введем текущий прогресс. За каждый квант работы его прогресс увеличивается на величину шага.

При принятии решения о следующем процессе вместо лотереи просто берем процесс с наименьшим прогрессом.

Шаговое планирование: пример

Прогресс A (шаг 100)	Прогресс B (шаг 200)	Прогресс C (шаг 40)	Текущий процесс
0	0	0	A

Шаговое планирование: пример

Прогресс A (шаг 100)	Прогресс B (шаг 200)	Прогресс C (шаг 40)	Текущий процесс
0	0	0	A
100	0	0	B

Шаговое планирование: пример

Прогресс A (шаг 100)	Прогресс B (шаг 200)	Прогресс C (шаг 40)	Текущий процесс
0	0	0	A
100	0	0	B
100	200	0	C

Шаговое планирование: пример

Прогресс A (шаг 100)	Прогресс B (шаг 200)	Прогресс C (шаг 40)	Текущий процесс
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C

Шаговое планирование: пример

Прогресс A (шаг 100)	Прогресс B (шаг 200)	Прогресс C (шаг 40)	Текущий процесс
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C

Шаговое планирование: пример

Прогресс A (шаг 100)	Прогресс B (шаг 200)	Прогресс C (шаг 40)	Текущий процесс
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A

Шаговое планирование: пример

Прогресс A (шаг 100)	Прогресс B (шаг 200)	Прогресс C (шаг 40)	Текущий процесс
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C

Шаговое планирование: пример

Прогресс A (шаг 100)	Прогресс B (шаг 200)	Прогресс C (шаг 40)	Текущий процесс
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C

Шаговое планирование: пример

Прогресс A (шаг 100)	Прогресс B (шаг 200)	Прогресс C (шаг 40)	Текущий процесс
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	A

Шаговое планирование: пример

Прогресс A (шаг 100)	Прогресс B (шаг 200)	Прогресс C (шаг 40)	Текущий процесс
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	A

Добились того же, что и с лотереями, но без рандома.

Проблемы?

Шаговое планирование: пример

Прогресс A (шаг 100)	Прогресс B (шаг 200)	Прогресс C (шаг 40)	Текущий процесс
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	A

Добились того же, что и с лотереями, но без рандома.

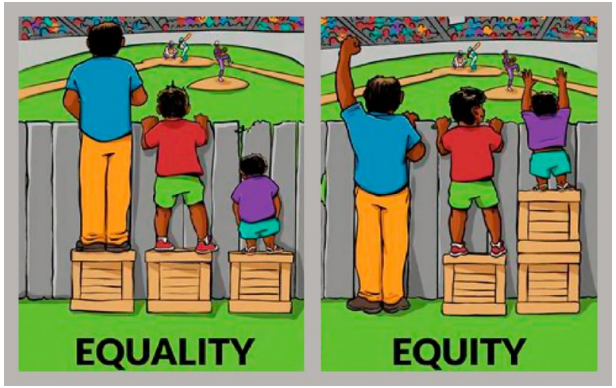
Проблемы?

Очень трудно вводить новые задачи: нужно всем как-то пересчитать шаги (а может и прогресс?)

Задача пропорционального планирования

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Идея: попробуем гарантировать, что каждое задание получает определенную долю процессорного времени. В частном случае это задача равномерного планирования, но такое далеко не всегда нужно.



Задача пропорционального планирования

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Идея: попробуем гарантировать, что каждое задание получает определенную долю процессорного времени. В частном случае это задача равномерного планирования, но такое далеко не всегда нужно.

Примеры: лотерейное и шаговые планирования. Имеют ряд проблем именно в качестве планировщика OS, но чудесно хороши в задачах, где вы просто делите ресурс на заранее известное количество игроков (выделяете WSL-у на Windows 25% ресурсов)

Вполне равномерный планировщик (CFS)

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Вполне равномерный планировщик (CFS)

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Идеи: применим идеи шагового планировщика, но:

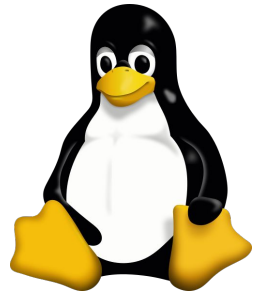
1. добавим адаптивности,
2. сделаем поправки на OS-специфику (включая IO),
3. формализуем приоритеты процессов,
4. побеспокоимся о производительности

Вполне равномерный планировщик (CFS)

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Идеи: применим идеи шагового планировщика, но:

1. добавим адаптивности,
2. сделаем поправки на OS-специфику (IO),
3. формализуем приоритеты процессов
4. побеспокоимся о производительности



Так появляется Вполне равномерный планировщик (Completely Fair Scheduler - CFS) - главный планировщик в Linux с 2007 по 2023 годы.

Вполне равномерный планировщик (CFS)

Основное устройство:

1. Для каждого процесса вводится его состояние: vruntime (virtual runtime).

Вполне равномерный планировщик (CFS)

Основное устройство:

1. Для каждого процесса вводится его состояние: vruntime (virtual runtime). Это то, сколько процесс всего отработал (виртуального) времени (аналогия с прогрессом шагового планировщика).

Вполне равномерный планировщик (CFS)

Основное устройство:

1. Для каждого процесса вводится его состояние: vruntime (virtual runtime). Это то, сколько процесс всего отработал (виртуального) времени (аналогия с прогрессом шагового планировщика).

2. В момент принятия решения о следующем активном процессе CFS выбирает его с минимальным vruntime.

Вполне равномерный планировщик (CFS)

Основное устройство:

1. Для каждого процесса вводится его состояние: vruntime (virtual runtime). Это то, сколько процесс всего отработал (виртуального) времени (аналогия с прогрессом шагового планировщика).

2. В момент принятия решения о следующем активном процессе CFS выбирает его с минимальным vruntime.

3. Кванты планирования больше не фиксированы. Они зависят текущего количества процессов, т.е. определяются динамически.

Вполне равномерный планировщик (CFS)

Основное устройство:

1. Для каждого процесса вводится его состояние: vruntime (virtual runtime). Это то, сколько процесс всего отработал (виртуального) времени (аналогия с прогрессом шагового планировщика).

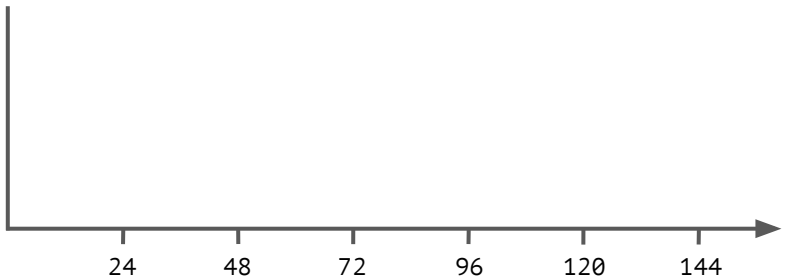
2. В момент принятия решения о следующем активном процессе CFS выбирает его с минимальным vruntime.

3. Кванты планирования больше не фиксированы. Они зависят текущего количества процессов, т.е. определяются динамически. Для этого используют параметр shed_latency (его делят на кол-во процессов*).

*пока не говорим про приоритеты

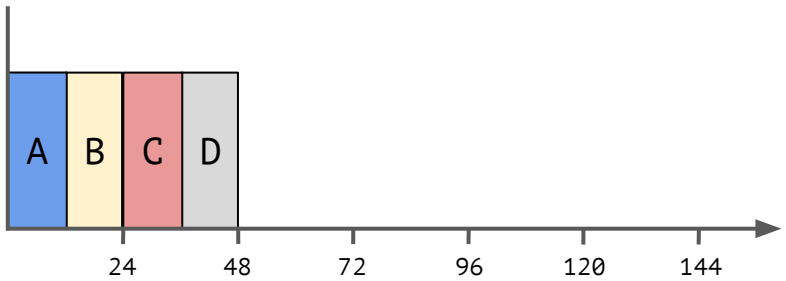
Вполне равномерный планировщик (CFS)

Пример: пусть в момент 0 стартуют процессы A, B, C, D. При этом shed_latency = 48ms.



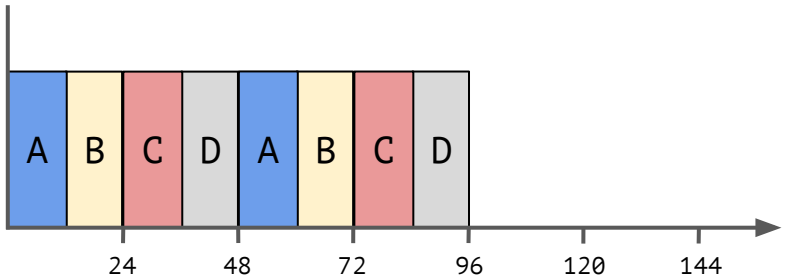
Вполне равномерный планировщик (CFS)

Пример: пусть в момент 0 стартуют процессы A, B, C, D. При этом shed_latency = 48ms. Каждый из них будет получать квант по 12 ms



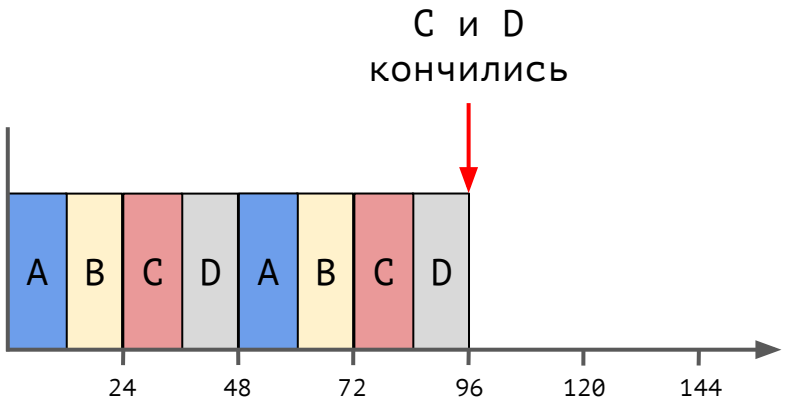
Вполне равномерный планировщик (CFS)

Пример: пусть в момент 0 стартуют процессы A, B, C, D. При этом shed_latency = 48ms. Каждый из них будет получать квант по 12 ms. А выбирать их будут равномерно по наименьшему vruntime.



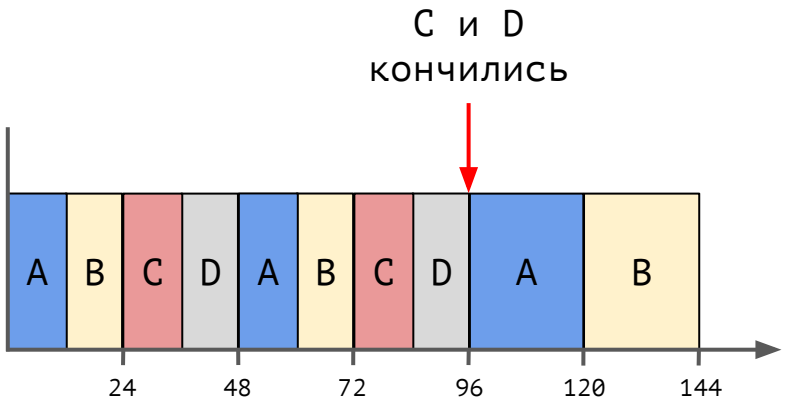
Вполне равномерный планировщик (CFS)

Пример: пусть в момент 0 стартуют процессы A, B, C, D. При этом shed_latency = 48ms. Каждый из них будет получать квант по 12 ms. А выбирать их будут равномерно по наименьшему vruntime. Пусть C и D кончились к моменту 96.



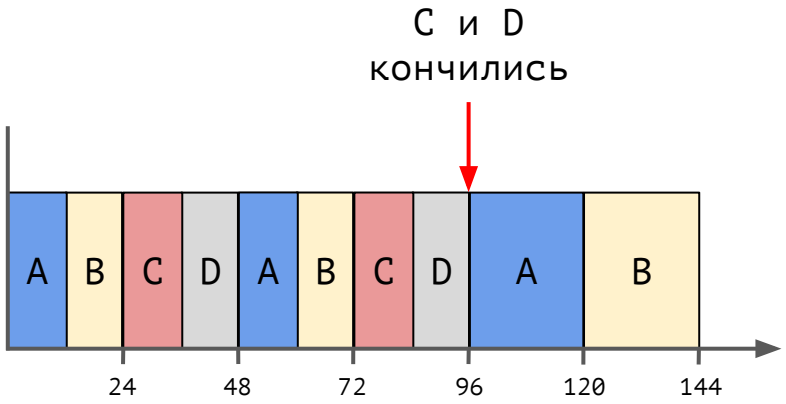
Вполне равномерный планировщик (CFS)

Пример: пусть в момент 0 стартуют процессы A, B, C, D. При этом shed_latency = 48ms. Каждый из них будет получать квант по 12 ms. A выбирать их будут равномерно по наименьшему vruntime. Пусть C и D кончились к моменту 96. Тогда A и B начинают получать кванты по 24ms.



Вполне равномерный планировщик (CFS)

Пример: пусть в момент 0 стартуют процессы A, B, C, D. При этом shed_latency = 48ms. Каждый из них будет получать квант по 12 ms. A выбирать их будут равномерно по наименьшему vruntime. Пусть C и D кончились к моменту 96. Тогда A и B начинают получать кванты по 24ms.



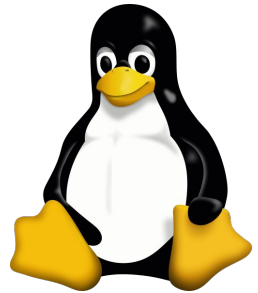
Замечание: есть ограничение снизу на размер кванта (min_granularity), чтобы не свалиться в бесконечный context-switch

Вполне равномерный планировщик (CFS)

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Идеи: применим идеи шагового планировщика, но:

1. добавим адаптивности,
2. сделаем поправки на OS-специфику (IO),
3. формализуем приоритеты процессов
4. побеспокоимся о производительности



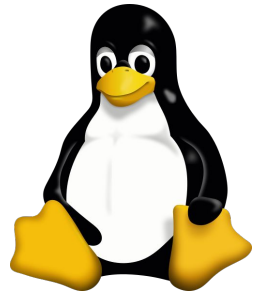
Так появляется Вполне равномерный планировщик (Completely Fair Scheduler - CFS) - главный планировщик в Linux с 2007 по 2023 годы.

Вполне равномерный планировщик (CFS)

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Идеи: применим идеи шагового планировщика, но:

1. добавим адаптивности,
2. сделаем поправки на OS-специфику (IO),
3. формализуем приоритеты процессов
4. побеспокоимся о производительности



Так появляется Вполне равномерный планировщик (Completely Fair Scheduler - CFS) - главный планировщик в Linux с 2007 по 2023 годы.

Вполне равномерный планировщик (CFS)

Приоритеты: у каждого процесса в Unix есть уровень nice. Он может принимать значения от -20 до +19, по умолчанию равен 0. Чем ниже nice, тем выше приоритет процесса.

Вполне равномерный планировщик (CFS)

Приоритеты: у каждого процесса в Unix есть уровень nice. Он может принимать значения от -20 до +19, по умолчанию равен 0. Чем ниже nice, тем выше приоритет процесса.

По уровню nice можно получить значение весовой функции:

```
static const int prio_to_weight[40] = {
    /* -20 */     88761,     71755,     56483,     46273,     36291,
    /* -15 */     29154,     23254,     18705,     14949,     11916,
    /* -10 */      9548,      7620,      6100,      4904,      3906,
    /*  -5 */      3121,      2501,      1991,      1586,      1277,
    /*   0 */      1024,       820,       655,       526,       423,
    /*   5 */       335,       272,       215,       172,       137,
    /*  10 */       110,        87,        70,        56,        45,
    /*  15 */        36,        29,        23,        18,        15,
  };
```

Вполне равномерный планировщик (CFS)

Приоритеты: у каждого процесса в Unix есть уровень nice. Он может принимать значения от -20 до +19, по умолчанию равен 0. Чем ниже nice, тем выше приоритет процесса.

По уровню nice можно получить значение весовой функции.

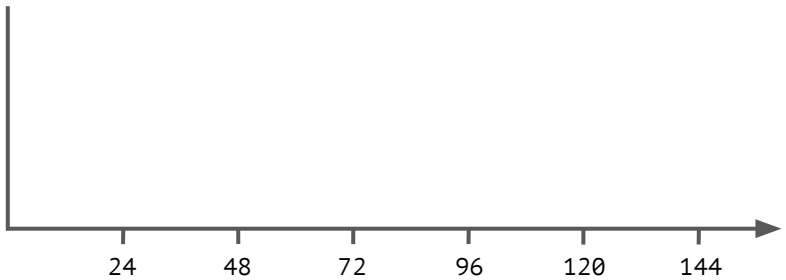
Длительность квантов и изменение vruntime пересчитывается с учетом весов (поэтому vruntime и виртуальное).

$$time\_slice_k = \frac{weight_k}{\sum_{i=0}^{n-1} weight_i} * sched\_latency$$

$$vruntime_i = vruntime_i + \frac{weight_0}{weight_i} * runtime_i$$

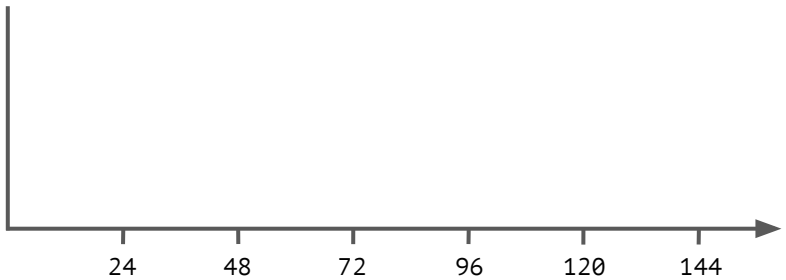
Вполне равномерный планировщик (CFS)

Пример: пусть в момент 0 стартуют процессы A, B. При этом shed_latency = 48ms, у A уровень nice равен -5, а у B дефолтный - 0.



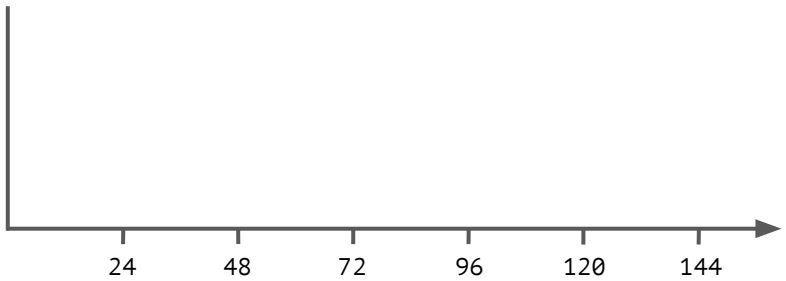
Вполне равномерный планировщик (CFS)

Пример: пусть в момент 0 стартуют процессы A, B. При этом shed_latency = 48ms, у A уровень nice равен -5, а у B дефолтный - 0. Тогда weight_a = 3121, а у weight_b = 1024.



Вполне равномерный планировщик (CFS)

Пример: пусть в момент 0 стартуют процессы A, B. При этом shed_latency = 48ms, у A уровень nice равен -5, а у B дефолтный - 0. Тогда weight_a = 3121, а у weight_b = 1024. Тогда time_slice_a ~ (3/4) * 48ms = 36ms; time_slice_b ~ (1/4) * 48ms = 12ms. При этом vruntime будет компенсирован (тоже в три раза)!

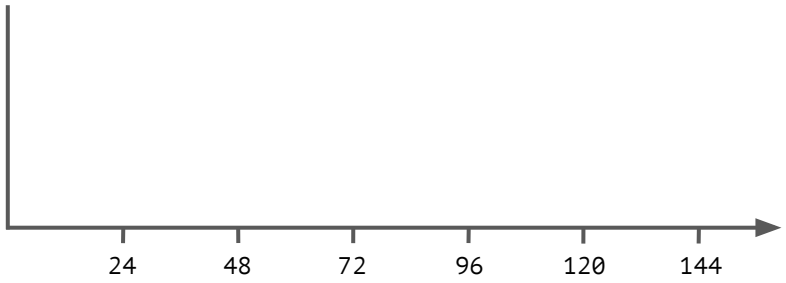


$$time\_slice_k = \frac{weight_k}{\sum_{i=0}^{n-1} weight_i} * sched\_latency$$

$$vruntime_i = vruntime_i + \frac{weight_0}{weight_i} * runtime_i$$

Вполне равномерный планировщик (CFS)

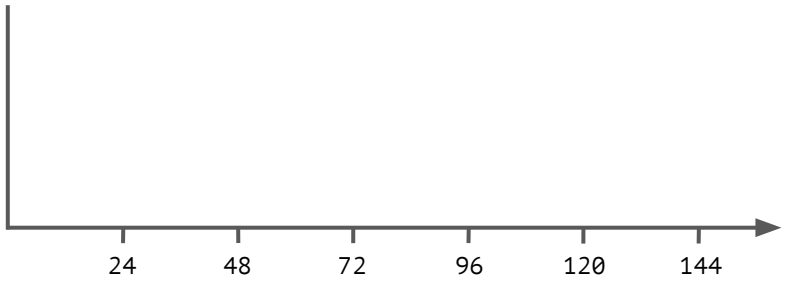
Пример: пусть в момент 0 стартуют процессы A, B. При этом shed_latency = 48ms, у A уровень nice равен -5, а у B дефолтный - 0. … time_slice_a = 36ms; time_slice_b = 12ms. При этом vruntime будет компенсирован в три раза.



vruntime A	vruntime B	Текущий процесс

Вполне равномерный планировщик (CFS)

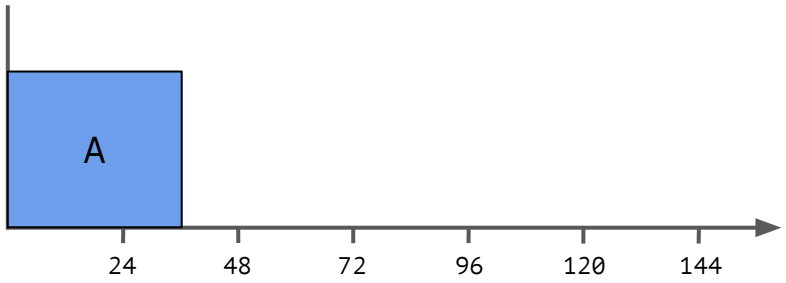
Пример: пусть в момент 0 стартуют процессы A, B. При этом shed_latency = 48ms, у A уровень nice равен -5, а у B дефолтный - 0. … time_slice_a = 36ms; time_slice_b = 12ms. При этом vruntime будет компенсирован в три раза.



vruntime A	vruntime B	Текущий процесс
0	0	A

Вполне равномерный планировщик (CFS)

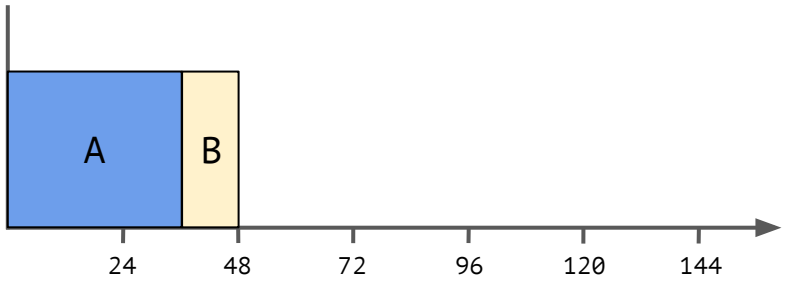
Пример: пусть в момент 0 стартуют процессы A, B. При этом shed_latency = 48ms, у A уровень nice равен -5, а у B дефолтный - 0. … time_slice_a = 36ms; time_slice_b = 12ms. При этом vruntime будет компенсирован в три раза.



vruntime A	vruntime B	Текущий процесс
0	0	A
36	0	

Вполне равномерный планировщик (CFS)

Пример: пусть в момент 0 стартуют процессы A, B. При этом shed_latency = 48ms, у A уровень nice равен -5, а у B дефолтный - 0. … time_slice_a = 36ms; time_slice_b = 12ms. При этом vruntime будет компенсирован в три раза.

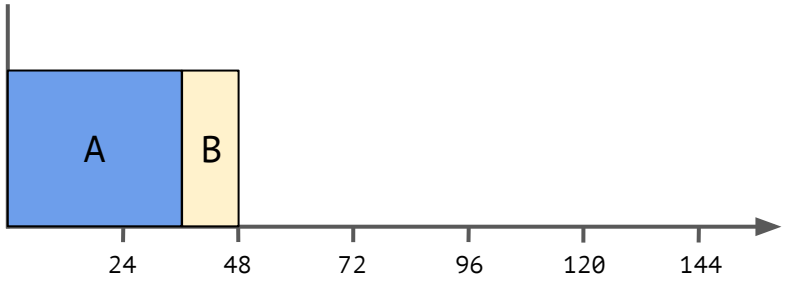


vruntime A	vruntime B	Текущий процесс
0	0	A
36	0	B

Вполне равномерный планировщик (CFS)

Пример: пусть в момент 0 стартуют процессы A, B. При этом shed_latency = 48ms, у A уровень nice равен -5, а у B дефолтный - 0. … time_slice_a = 36ms; time_slice_b = 12ms. При этом vruntime будет компенсирован в три раза.

$$vruntime_i = vruntime_i + \frac{weight_0}{weight_i} * runtime_i$$

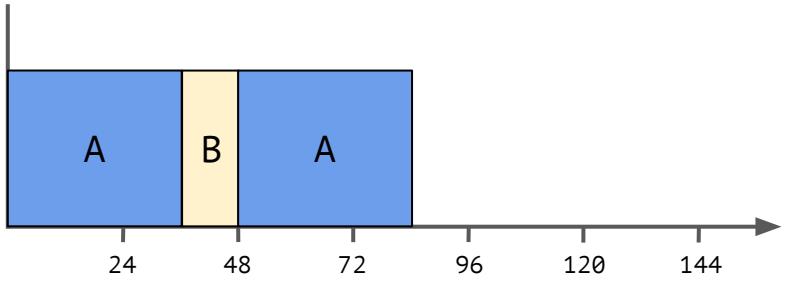


vruntime A	vruntime B	Текущий процесс
0	0	A
36	0	B
36	36	

потому, что увеличили в 3 раза!

Вполне равномерный планировщик (CFS)

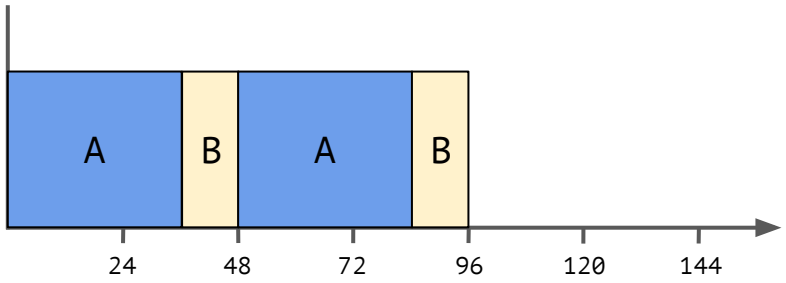
Пример: пусть в момент 0 стартуют процессы A, B. При этом shed_latency = 48ms, у A уровень nice равен -5, а у B дефолтный - 0. … time_slice_a = 36ms; time_slice_b = 12ms. При этом vruntime будет компенсирован в три раза.



vruntime A	vruntime B	Текущий процесс
0	0	A
36	0	B
36	36	A
72	36	

Вполне равномерный планировщик (CFS)

Пример: пусть в момент 0 стартуют процессы A, B. При этом shed_latency = 48ms, у A уровень nice равен -5, а у B дефолтный - 0. … time_slice_a = 36ms; time_slice_b = 12ms. При этом vruntime будет компенсирован в три раза.



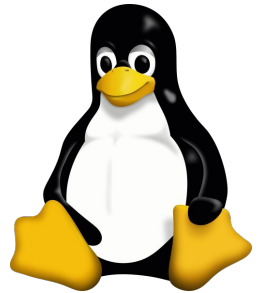
vruntime A	vruntime B	Текущий процесс
0	0	A
36	0	B
36	36	A
72	36	B
72	72	

Вполне равномерный планировщик (CFS)

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Идеи: применим идеи шагового планировщика, но:

1. добавим адаптивности,
2. сделаем поправки на OS-специфику (IO),
3. формализуем приоритеты процессов
4. побеспокоимся о производительности



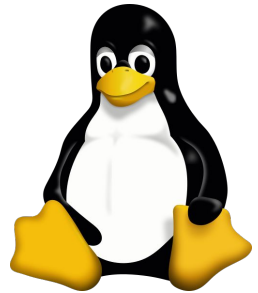
Так появляется Вполне равномерный планировщик (Completely Fair Scheduler - CFS) - главный планировщик в Linux с 2007 по 2023 годы.

Вполне равномерный планировщик (CFS)

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Идеи: применим идеи шагового планировщика, но:

1. добавим адаптивности,
2. сделаем поправки на OS-специфику (IO),
3. формализуем приоритеты процессов
4. побеспокоимся о производительности



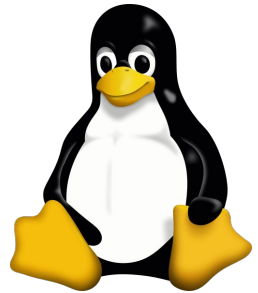
Так появляется Вполне равномерный планировщик (Completely Fair Scheduler - CFS) - главный планировщик в Linux с 2007 по 2023 годы.

Вполне равномерный планировщик (CFS)

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Идеи: применим идеи шагового планировщика, но:

1. добавим адаптивности,
2. сделаем поправки на OS-специфику (IO),
3. формализуем приоритеты процессов
4. побеспокоимся о производительности



Так появляется Вполне равномерный планировщик (Completely Fair Scheduler - CFS) - главный планировщик в Linux с 2007 по 2023 годы.

Вполне равномерный планировщик (CFS)

Производительность: по ходу работы CFS нам нужно все время брать задачу с минимальным vruntime.

Какую структуру данных для этого использовать?

Вполне равномерный планировщик (CFS)

Производительность: по ходу работы CFS нам нужно все время брать задачу с минимальным vruntime.

Какую структуру данных для этого использовать?

- (Bin)Heap? Можно быстро посмотреть минимум, а достать/добавить за O(logN). Проблемы?

Вполне равномерный планировщик (CFS)

Производительность: по ходу работы CFS нам нужно все время брать задачу с минимальным vruntime.

Какую структуру данных для этого использовать?

- (Bin)Heap? Можно быстро посмотреть минимум, а достать/добавить за O(logN). Проблемы?

  Реализуется массивом, а его нужно иногда релоцировать… во время переключения потоков? Непозволительно долго.

Вполне равномерный планировщик (CFS)

Производительность: по ходу работы CFS нам нужно все время брать задачу с минимальным vruntime.

Какую структуру данных для этого использовать?

- (Bin)Heap? Можно быстро посмотреть минимум, а достать/добавить за O(logN). Проблемы: релокации
- Сбалансированное дерево? А какое?

АВЛ деревья VS Красно-чёрные деревья

АВЛ:

+ операции за O(logN)
+ глубина ~1.5*logN
+ добавление требует O(1) поворотов

– удаление требует O(logN) поворотов

Красно-**чёрные** :

+ операции за O(logN)
+ добавление требует O(1) поворотов
+ удаление требует O(1) поворотов

– глубина ⩽ 2*logN

Вполне равномерный планировщик (CFS)

Производительность: по ходу работы CFS нам нужно все время брать задачу с минимальным vruntime.

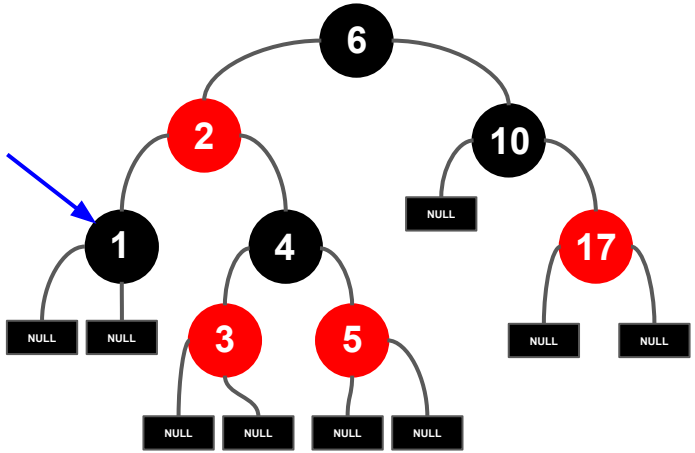
Какую структуру данных для этого использовать?

- (Bin)Heap? Можно быстро посмотреть минимум, а достать/добавить за O(logN). Проблемы: релокации
- Сбалансированное дерево:
  - AVL: больше действий на удалении
  - Красно-черные: берем ✅

Вполне равномерный планировщик (CFS)

Производительность: по ходу работы CFS нам нужно все время брать задачу с минимальным vruntime.

Поддерживаем Красно-черное, где ключ - это vruntime. Храним ссылку на минимальный элемент, чтобы сэкономить на поиске (остальные операции все равно O(logN), но все-таки).

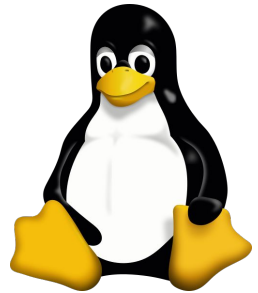


Вполне равномерный планировщик (CFS)

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Идеи: применим идеи шагового планировщика, но:

1. добавим адаптивности,
2. сделаем поправки на OS-специфику (IO),
3. формализуем приоритеты процессов
4. побеспокоимся о производительности



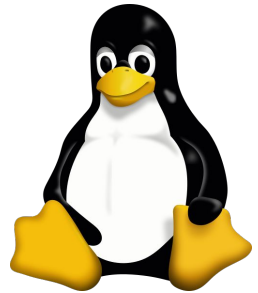
Так появляется Вполне равномерный планировщик (Completely Fair Scheduler - CFS) - главный планировщик в Linux с 2007 по 2023 годы.

Вполне равномерный планировщик (CFS)

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Идеи: применим идеи шагового планировщика, но:

1. добавим адаптивности,
2. сделаем поправки на OS-специфику (IO),
3. формализуем приоритеты процессов
4. побеспокоимся о производительности



Так появляется Вполне равномерный планировщик (Completely Fair Scheduler - CFS) - главный планировщик в Linux с 2007 по 2023 годы.

Вполне равномерный планировщик (CFS)

Поддержка IO: пусть процесс надолго уснул (ждет ответа от IO). Очевидно, мы уберем его из красно-черного дерева, пока он не проснется.

Но что будет с его vruntime, когда он наконец проснется?

Вполне равномерный планировщик (CFS)

Поддержка IO: пусть процесс надолго уснул (ждет ответа от IO). Очевидно, мы уберем его из красно-черного дерева, пока он не проснется.

Но что будет с его vruntime, когда он наконец проснется? Он будет очень сильно отставать от vruntime работавших все это время процессов =>

Вполне равномерный планировщик (CFS)

Поддержка IO: пусть процесс надолго уснул (ждет ответа от IO). Очевидно, мы уберем его из красно-черного дерева, пока он не проснется.

Но что будет с его vruntime, когда он наконец проснется? Он будет очень сильно отставать от vruntime работавших все это время процессов => а значит его раз за разом будут ставить на CPU! Как исправить?

Вполне равномерный планировщик (CFS)

Поддержка IO: пусть процесс надолго уснул (ждет ответа от IO). Очевидно, мы уберем его из красно-черного дерева, пока он не проснется.

Но что будет с его vruntime, когда он наконец проснется? Он будет очень сильно отставать от vruntime работавших все это время процессов => а значит его раз за разом будут ставить на CPU! Как исправить?

Чтобы этого избежать, при пробуждении процесса ему выставляется наименьшее значение vrtuntime из красно-черного дерева.

Вполне равномерный планировщик (CFS)

Поддержка IO: пусть процесс надолго уснул (ждет ответа от IO). Очевидно, мы уберем его из красно-черного дерева, пока он не проснется.

Но что будет с его vruntime, когда он наконец проснется? Он будет очень сильно отставать от vruntime работавших все это время процессов => а значит его раз за разом будут ставить на CPU! Как исправить?

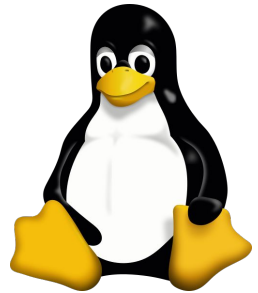
Чтобы этого избежать, при пробуждении процесса ему выставляется наименьшее значение vrtuntime из красно-черного дерева (если он, конечно, отстал от него).

Вполне равномерный планировщик (CFS)

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Идеи: применим идеи шагового планировщика, но:

1. добавим адаптивности,
2. сделаем поправки на OS-специфику (IO),
3. формализуем приоритеты процессов
4. побеспокоимся о производительности



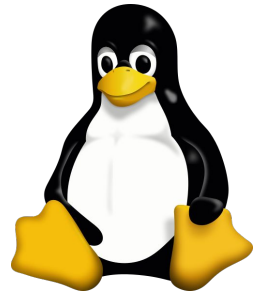
Так появляется Вполне равномерный планировщик (Completely Fair Scheduler - CFS) - главный планировщик в Linux с 2007 по 2023 годы.

Вполне равномерный планировщик (CFS)

Глобальная задача: разработать высокоуровневые политики для планирования процессов в операционной системе.

Идеи: применим идеи шагового планировщика, но:

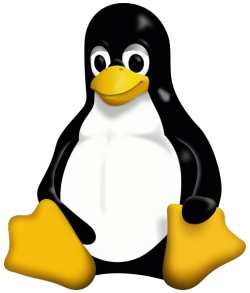
1. добавим адаптивности,
2. сделаем поправки на OS-специфику (IO),
3. формализуем приоритеты процессов
4. побеспокоимся о производительности



Так появляется Вполне равномерный планировщик (Completely Fair Scheduler - CFS) - главный планировщик в Linux с 2007 по 2023 годы.

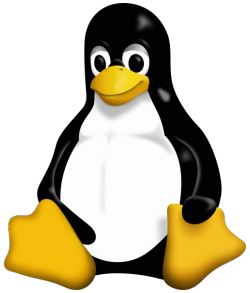
Вполне равномерный планировщик (CFS)

Текущий статус: CFS очень хорошо справлялся с равномерным распределением времени доступа к CPU между процессами, но была одна проблема



Вполне равномерный планировщик (CFS)

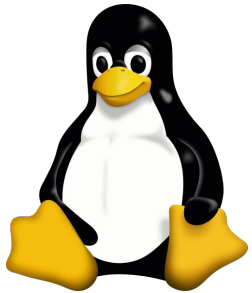
Текущий статус: CFS очень хорошо справлялся с равномерным распределением времени доступа к CPU между процессами, но была одна проблема: latency-sensitive задачи



Вполне равномерный планировщик (CFS)

Текущий статус: CFS очень хорошо справлялся с равномерным распределением времени доступа к CPU между процессами, но была одна проблема: latency-sensitive задачи

Представьте, что у вас появляются задачи, которым обычно не нужно много времени CPU, но при этом им нужно очень быстро и часто (но не очень надолго) захватывать CPU.

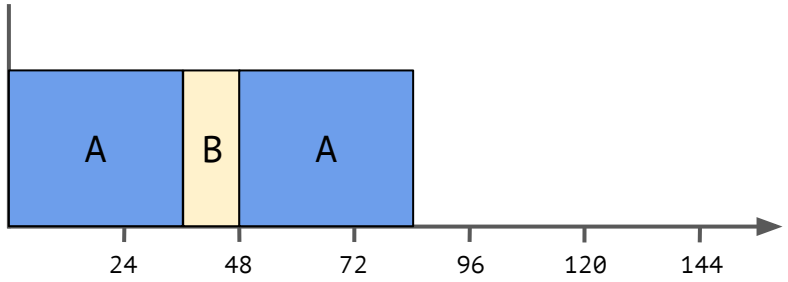


Вполне равномерный планировщик (CFS)

Текущий статус: CFS очень хорошо справлялся с равномерным распределением времени доступа к CPU между процессами, но была одна проблема: latency-sensitive задачи

Представьте, что у вас появляются задачи, которым обычно не нужно много времени CPU, но при этом им нужно очень быстро и часто (но не очень надолго) захватывать CPU.

умеем так



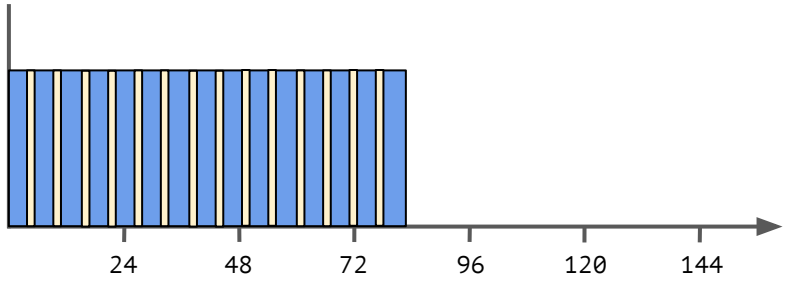
vruntime A	vruntime B	Текущий процесс
0	0	A
36	0	B
36	36	A
72	36	

Вполне равномерный планировщик (CFS)

Текущий статус: CFS очень хорошо справлялся с равномерным распределением времени доступа к CPU между процессами, но была одна проблема: latency-sensitive задачи

Представьте, что у вас появляются задачи, которым обычно не нужно много времени CPU, но при этом им нужно очень быстро и часто (но не очень надолго) захватывать CPU.

а хотим так



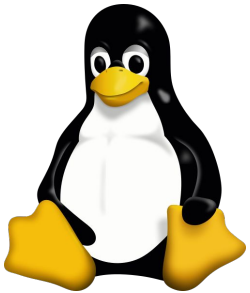
vruntime A	vruntime B	Текущий процесс
??	??	A
??	??	B
??	??	A
??	??	B

Вполне равномерный планировщик (CFS)

Текущий статус: CFS очень хорошо справлялся с равномерным распределением времени доступа к CPU между процессами, но была одна проблема: latency-sensitive задачи

Представьте, что у вас появляются задачи, которым обычно не нужно много времени CPU, но при этом им нужно очень быстро и часто (но не очень надолго) захватывать CPU.

Повышать приоритет => выдавать слишком много времени



Вполне равномерный планировщик (CFS)

Текущий статус: CFS очень хорошо справлялся с равномерным распределением времени доступа к CPU между процессами, но была одна проблема: latency-sensitive задачи

Представьте, что у вас появляются задачи, которым обычно не нужно много времени CPU, но при этом им нужно очень быстро и часто (но не очень надолго) захватывать CPU.

Повышать приоритет => выдавать слишком много времени;

Нужен еще какой-то механизм, который бы влиял на частоту загрузки на CPU (он назывался latency nice patches)

Вполне равномерный планировщик (CFS)

Текущий статус: CFS очень хорошо справлялся с равномерным распределением времени доступа к CPU между процессами, но была одна проблема: latency-sensitive задачи

Представьте, что у вас появляются задачи, которым обычно не нужно много времени CPU, но при этом им нужно очень быстро и часто (но не очень надолго) захватывать CPU.

Повышать приоритет => выдавать слишком много времени;

Нужен еще какой-то механизм, который бы влиял на частоту загрузки на CPU (он назывался latency nice patches). Но все это было хрупко и не очень эффективно, поэтому в 2023 году CFS заменили на...

Earliest Eligible Virtual Deadline First (EEVDF)

Earliest Eligible Virtual Deadline First (EEVDF)

Общая идея: сохраняем почти все идеи CFS, но чуть иначе гарантируем честность, а вместо vruntime в красно-черном дереве используем виртуальный дедлайн.

Earliest Eligible Virtual Deadline First (EEVDF)

Общая идея: сохраняем почти все идеи CFS, но чуть иначе гарантируем честность, а вместо vruntime в красно-черном дереве используем виртуальный дедлайн.

1) Как гарантируем честность? Для каждой задачи поддерживаем параметр lag: насколько vruntime меньше или больше "среднего" vruntime по всей очереди.

Earliest Eligible Virtual Deadline First (EEVDF)

1) Как гарантируем честность? Для каждой задачи поддерживаем параметр lag: насколько vruntime меньше или больше "среднего" vruntime по всей очереди.

Пусть кванты для всех по 30ms, приоритеты равны

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0

Earliest Eligible Virtual Deadline First (EEVDF)

1) Как гарантируем честность? Для каждой задачи поддерживаем параметр lag: насколько vruntime меньше или больше "среднего" vruntime по всей очереди.

Пусть кванты для всех по 30ms, приоритеты равны

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
30ms		0		0	

Earliest Eligible Virtual Deadline First (EEVDF)

1) Как гарантируем честность? Для каждой задачи поддерживаем параметр lag: насколько vruntime меньше или больше "среднего" vruntime по всей очереди.

Пусть кванты для всех по 30ms, приоритеты равны

в идеале, каждый работает по 10ms

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
30ms		0	+10ms	0	+10ms

Earliest Eligible Virtual Deadline First (EEVDF)

1) Как гарантируем честность? Для каждой задачи поддерживаем параметр lag: насколько vruntime меньше или больше "среднего" vruntime по всей очереди.

Пусть кванты для всех по 30ms, приоритеты равны

в идеале, каждый работает по 10ms

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
30ms	-20ms	0	+10ms	0	+10ms

Earliest Eligible Virtual Deadline First (EEVDF)

1) Как гарантируем честность? Для каждой задачи поддерживаем параметр lag: насколько vruntime меньше или больше "среднего" vruntime по всей очереди.

Пусть кванты для всех по 30ms, приоритеты равны

в идеале, каждый работает по 10ms

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
30ms	-20ms	0	+10ms	0	+10ms
	переработка		недоработка		недоработка

Earliest Eligible Virtual Deadline First (EEVDF)

1) Как гарантируем честность? Для каждой задачи поддерживаем параметр lag: насколько vruntime меньше или больше "среднего" vruntime по всей очереди.

Пусть кванты для всех по 30ms, приоритеты равны

в идеале, каждый работает по 10ms

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
30ms	-20ms	0	+10ms	0	+10ms

Earliest Eligible Virtual Deadline First (EEVDF)

1) Как гарантируем честность? Для каждой задачи поддерживаем параметр lag: насколько vruntime меньше или больше "среднего" vruntime по всей очереди.

Пусть кванты для всех по 30ms, приоритеты равны

в идеале, каждый работает по 10ms

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
30ms	-20ms	0	+10ms	0	+10ms
30ms		30ms		0	

Earliest Eligible Virtual Deadline First (EEVDF)

1) Как гарантируем честность? Для каждой задачи поддерживаем параметр lag: насколько vruntime меньше или больше "среднего" vruntime по всей очереди.

Пусть кванты для всех по 30ms, приоритеты равны

в идеале, каждый работает по 10ms

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
30ms	-20ms	0	+10ms	0	+10ms
30ms	-10ms	30ms	-10ms	0	+20ms

Earliest Eligible Virtual Deadline First (EEVDF)

1) Как гарантируем честность? Для каждой задачи поддерживаем параметр lag: насколько vruntime меньше или больше "среднего" vruntime по всей очереди.

Пусть кванты для всех по 30ms, приоритеты равны

в идеале, каждый работает по 10ms

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
30ms	-20ms	0	+10ms	0	+10ms
30ms	-10ms	30ms	-10ms	0	+20ms
30ms	0	30ms	0	30ms	0

Earliest Eligible Virtual Deadline First (EEVDF)

1) Как гарантируем честность? Для каждой задачи поддерживаем параметр lag: насколько vruntime меньше или больше "среднего" vruntime по всей очереди.

Пусть кванты для всех по 30ms, приоритеты равны

в идеале, каждый работает по 10ms

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
30ms	-20ms	0	+10ms	0	+10ms
30ms	-10ms	30ms	-10ms	0	+20ms
30ms	0	30ms	0	30ms	0
60ms	-20ms	0	+10ms	0	+10ms

Earliest Eligible Virtual Deadline First (EEVDF)

Общая идея: сохраняем почти все идеи CFS, но чуть иначе гарантируем честность, а вместо vruntime в красно-черном дереве используем виртуальный дедлайн.

1) Как гарантируем честность? Для каждой задачи поддерживаем параметр lag: насколько vruntime меньше или больше "среднего" vruntime по всей очереди.

Earliest Eligible Virtual Deadline First (EEVDF)

Общая идея: сохраняем почти все идеи CFS, но чуть иначе гарантируем честность, а вместо vruntime в красно-черном дереве используем виртуальный дедлайн.

1) Как гарантируем честность? Для каждой задачи поддерживаем параметр lag: насколько vruntime меньше или больше "среднего" vruntime по всей очереди. Берем в работу только те задачи, у которых лаг неотрицательный.

Earliest Eligible Virtual Deadline First (EEVDF)

Общая идея: сохраняем почти все идеи CFS, но чуть иначе гарантируем честность, а вместо vruntime в красно-черном дереве используем виртуальный дедлайн.

1) Как гарантируем честность? Для каждой задачи поддерживаем параметр lag: насколько vruntime меньше или больше "среднего" vruntime по всей очереди. Берем в работу только те задачи, у которых лаг неотрицательный.

2) Считает для задач виртуальные дедлайны: время, когда в ближайший раз задача могла бы завершить ближайший свой квант.

Earliest Eligible Virtual Deadline First (EEVDF)

Общая идея: сохраняем почти все идеи CFS, но чуть иначе гарантируем честность, а вместо vruntime в красно-черном дереве используем виртуальный дедлайн.

1) Как гарантируем честность? Для каждой задачи поддерживаем параметр lag: насколько vruntime меньше или больше "среднего" vruntime по всей очереди. Берем в работу только те задачи, у которых лаг неотрицательный.

2) Считает для задач виртуальные дедлайны: время, когда в ближайший раз задача могла бы завершить ближайший свой квант, начиная с того момента, как у нее неотрицательный лаг!

Earliest Eligible Virtual Deadline First (EEVDF)

Общая идея: сохраняем почти все идеи CFS, но чуть иначе гарантируем честность, а вместо vruntime в красно-черном дереве используем виртуальный дедлайн.

1) Как гарантируем честность? Для каждой задачи поддерживаем параметр lag: насколько vruntime меньше или больше "среднего" vruntime по всей очереди. Берем в работу только те задачи, у которых лаг неотрицательный.

2) Считает для задач виртуальные дедлайны: время, когда в ближайший раз задача могла бы завершить ближайший свой квант, начиная с того момента, как у нее неотрицательный лаг! При том кванты для latency sensitive задач ставятся маленькими (зависит latency nice)

Earliest Eligible Virtual Deadline First (EEVDF)

1) Считает для задач виртуальные дедлайны: время, когда в ближайший раз задача могла бы завершить ближайший свой квант, начиная с того момента, как у нее лаг >= 0

Пусть кванты:
A – 60 ms
B – 24 ms
C – **12 ms**

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0

Earliest Eligible Virtual Deadline First (EEVDF)

1) Считает для задач виртуальные дедлайны: время, когда в ближайший раз задача могла бы завершить ближайший свой квант, начиная с того момента, как у нее лаг >= 0

Пусть кванты:
A – 60 ms
B – 24 ms
C – **12 ms**

vdeadlines:

A – 60 ms
B – 24 ms
C – 12 ms

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0

Earliest Eligible Virtual Deadline First (EEVDF)

1) Считает для задач виртуальные дедлайны: время, когда в ближайший раз задача могла бы завершить ближайший свой квант, начиная с того момента, как у нее лаг >= 0

Пусть кванты:
A – 60 ms
B – 24 ms
C – **12 ms**

vdeadlines:

A – 60 ms
B – 24 ms
C – 12 ms

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
0		0		12ms	

Earliest Eligible Virtual Deadline First (EEVDF)

1) Считает для задач виртуальные дедлайны: время, когда в ближайший раз задача могла бы завершить ближайший свой квант, начиная с того момента, как у нее лаг >= 0

Пусть кванты:
A – 60 ms
B – 24 ms
C – **12 ms**

Работает C, считаем лаги

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
0	+4ms	0	+4ms	12ms	-8ms

Earliest Eligible Virtual Deadline First (EEVDF)

1) Считает для задач виртуальные дедлайны: время, когда в ближайший раз задача могла бы завершить ближайший свой квант, начиная с того момента, как у нее лаг >= 0

Пусть кванты:
A – 60 ms
B – 24 ms
C – 12 ms

Работает B, считаем лаги

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
0	+4ms	0	+4ms	12ms	-8ms
0		24ms		12ms	

Earliest Eligible Virtual Deadline First (EEVDF)

1) Считает для задач виртуальные дедлайны: время, когда в ближайший раз задача могла бы завершить ближайший свой квант, начиная с того момента, как у нее лаг >= 0

Пусть кванты:
A – 60 ms
B – 24 ms
C – 12 ms

Работает B, считаем лаги

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
0	+4ms	0	+4ms	12ms	-8ms
0	+12ms	24ms	-12ms	12ms	+0ms

Earliest Eligible Virtual Deadline First (EEVDF)

1) Считает для задач виртуальные дедлайны: время, когда в ближайший раз задача могла бы завершить ближайший свой квант, начиная с того момента, как у нее лаг >= 0

Пусть кванты:
A – 60 ms
B – 24 ms
C – 12 ms

Кто работает дальше?

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
0	+4ms	0	+4ms	12ms	-8ms
0	+12ms	24ms	-12ms	12ms	+0ms

Earliest Eligible Virtual Deadline First (EEVDF)

1) Считает для задач виртуальные дедлайны: время, когда в ближайший раз задача могла бы завершить ближайший свой квант, начиная с того момента, как у нее лаг >= 0

Пусть кванты:
A – 60 ms
B – 24 ms
C – 12 ms

vd[A] = 0 + 60

vd[C] = 12 + 12

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
0	+4ms	0	+4ms	12ms	-8ms
0	+12ms	24ms	-12ms	12ms	+0ms

Earliest Eligible Virtual Deadline First (EEVDF)

1) Считает для задач виртуальные дедлайны: время, когда в ближайший раз задача могла бы завершить ближайший свой квант, начиная с того момента, как у нее лаг >= 0

Пусть кванты:
A – 60 ms
B – 24 ms
C – 12 ms

Работает C, считаем лаги

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
0	+4ms	0	+4ms	12ms	-8ms
0	+12ms	24ms	-12ms	12ms	+0ms
0		24ms		24ms	

Earliest Eligible Virtual Deadline First (EEVDF)

1) Считает для задач виртуальные дедлайны: время, когда в ближайший раз задача могла бы завершить ближайший свой квант, начиная с того момента, как у нее лаг >= 0

Пусть кванты:
A – 60 ms
B – 24 ms
C – 12 ms

Теперь вариантов вообще нет кроме A!

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
0	+4ms	0	+4ms	12ms	-8ms
0	+12ms	24ms	-12ms	12ms	+0ms
0	+16ms	24ms	-8ms	24ms	-8ms

Earliest Eligible Virtual Deadline First (EEVDF)

1) Считает для задач виртуальные дедлайны: время, когда в ближайший раз задача могла бы завершить ближайший свой квант, начиная с того момента, как у нее лаг >= 0

Пусть кванты:
A – 60 ms
B – 24 ms
C – 12 ms

Теперь вариантов вообще нет кроме A!

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
0	+4ms	0	+4ms	12ms	-8ms
0	+12ms	24ms	-12ms	12ms	+0ms
0	+16ms	24ms	-8ms	24ms	-8ms
60ms		24ms		24ms	

Earliest Eligible Virtual Deadline First (EEVDF)

1) Считает для задач виртуальные дедлайны: время, когда в ближайший раз задача могла бы завершить ближайший свой квант, начиная с того момента, как у нее лаг >= 0

Пусть кванты:
A – 60 ms
B – 24 ms
C – 12 ms

Теперь вариантов вообще нет кроме A!

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
0	+4ms	0	+4ms	12ms	-8ms
0	+12ms	24ms	-12ms	12ms	+0ms
0	+16ms	24ms	-8ms	24ms	-8ms
60ms	20-60+16	24ms		24ms	

Earliest Eligible Virtual Deadline First (EEVDF)

1) Считает для задач виртуальные дедлайны: время, когда в ближайший раз задача могла бы завершить ближайший свой квант, начиная с того момента, как у нее лаг >= 0

Пусть кванты:
A – 60 ms
B – 24 ms
C – 12 ms

Теперь вариантов вообще нет кроме A!

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
0	+4ms	0	+4ms	12ms	-8ms
0	+12ms	24ms	-12ms	12ms	+0ms
0	+16ms	24ms	-8ms	24ms	-8ms
60ms	-24ms	24ms		24ms	

Earliest Eligible Virtual Deadline First (EEVDF)

1) Считает для задач виртуальные дедлайны: время, когда в ближайший раз задача могла бы завершить ближайший свой квант, начиная с того момента, как у нее лаг >= 0

Пусть кванты:
A – 60 ms
B – 24 ms
C – 12 ms

Теперь вариантов вообще нет кроме A!

A		B		C	
vtime	lag	vtime	lag	vtime	lag
0	0	0	0	0	0
0	+4ms	0	+4ms	12ms	-8ms
0	+12ms	24ms	-12ms	12ms	+0ms
0	+16ms	24ms	-8ms	24ms	-8ms
60ms	-24ms	24ms	+12ms	24ms	+12ms

Earliest Eligible Virtual Deadline First (EEVDF)

Общая идея: сохраняем почти все идеи CFS, но чуть иначе гарантируем честность, а вместо vruntime в красно-черном дереве используем виртуальный дедлайн.

1) Как гарантируем честность? Для каждой задачи поддерживаем параметр lag: насколько vruntime меньше или больше "среднего" vruntime по всей очереди. Берем в работу только те задачи, у которых лаг неотрицательный.

2) Считает для задач виртуальные дедлайны: время, когда в ближайший раз задача могла бы завершить ближайший свой квант, начиная с того момента, как у нее неотрицательный лаг! При том кванты для latency sensitive задач ставятся маленькими (зависит latency nice)

Earliest Eligible Virtual Deadline First (EEVDF)

Общая идея: сохраняем почти все идеи CFS, но чуть иначе гарантируем честность, а вместо vruntime в красно-черном дереве используем виртуальный дедлайн.

1) Как гарантируем честность? Для каждой задачи поддерживаем параметр lag: насколько vruntime меньше или больше "среднего" vruntime по всей очереди. Берем в работу только те задачи, у которых лаг неотрицательный.

2) Считает для задач виртуальные дедлайны: время, когда в ближайший раз задача могла бы завершить ближайший свой квант, начиная с того момента, как у нее неотрицательный лаг! При том кванты для latency sensitive задач ставятся маленькими (зависит latency nice). Именно эти дедлайны использует в качестве ключей в кч-дереве.

Earliest Eligible Virtual Deadline First (EEVDF)

Общая идея: сохраняем почти все идеи CFS, но чуть иначе гарантируем честность, а вместо vruntime в красно-черном дереве используем виртуальный дедлайн.

1) Как гарантируем честность? Для каждой задачи поддерживаем параметр lag: насколько vruntime меньше или больше "среднего" vruntime по всей очереди. Берем в работу только те задачи, у которых лаг неотрицательный.

2) Считает для задач виртуальные дедлайны: … Именно эти дедлайны использует в качестве ключей в кч-дереве.

3) Все это с учетом nice и проточек под IO

Earliest Eligible Virtual Deadline First (EEVDF)

Общая идея: сохраняем почти все идеи CFS, но чуть иначе гарантируем честность, а вместо vruntime в красно-черном дереве используем виртуальный дедлайн.

Как помогает с latency: у задач, чувствительных к latency, высчитываются маленькие кванты => более ранние дедлайны => они быстрее попадают на CPU (пусть и не надолго)

Осталось за кадром

- Детали реализации EEVDF (раз, два, три)
- Real-time планировщики в Linux (FCFS/RR)
- Планирование в условии многих ядер CPU (прибивание процессов к CPU, балансировка нагрузки, кража работы)

Takeaways

- Два подхода к разработке планировщиков: оптимизация оборотного времени/времени отклика или равномерное планирование;
- Всегда важно учитывать IO и интерактивные задачи;
- Применяются классические алгоритмы и особенно структуры данных, но с поправкой на практику