Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования «Алтайский государственный технический университет им. И. И. Ползунова»

Факультет информационных технологий

Кафедра прикладной математики

Отчет защищен с оценкой ___

Преподаватель _____ (подпись)

«___» _____ 2023 г.

Отчет

по лабораторным работам № 1-2

по дисциплине «Проектирование операционных систем»

Студент гр. 8ПИ-21

Потапов Д.П

Преподаватель

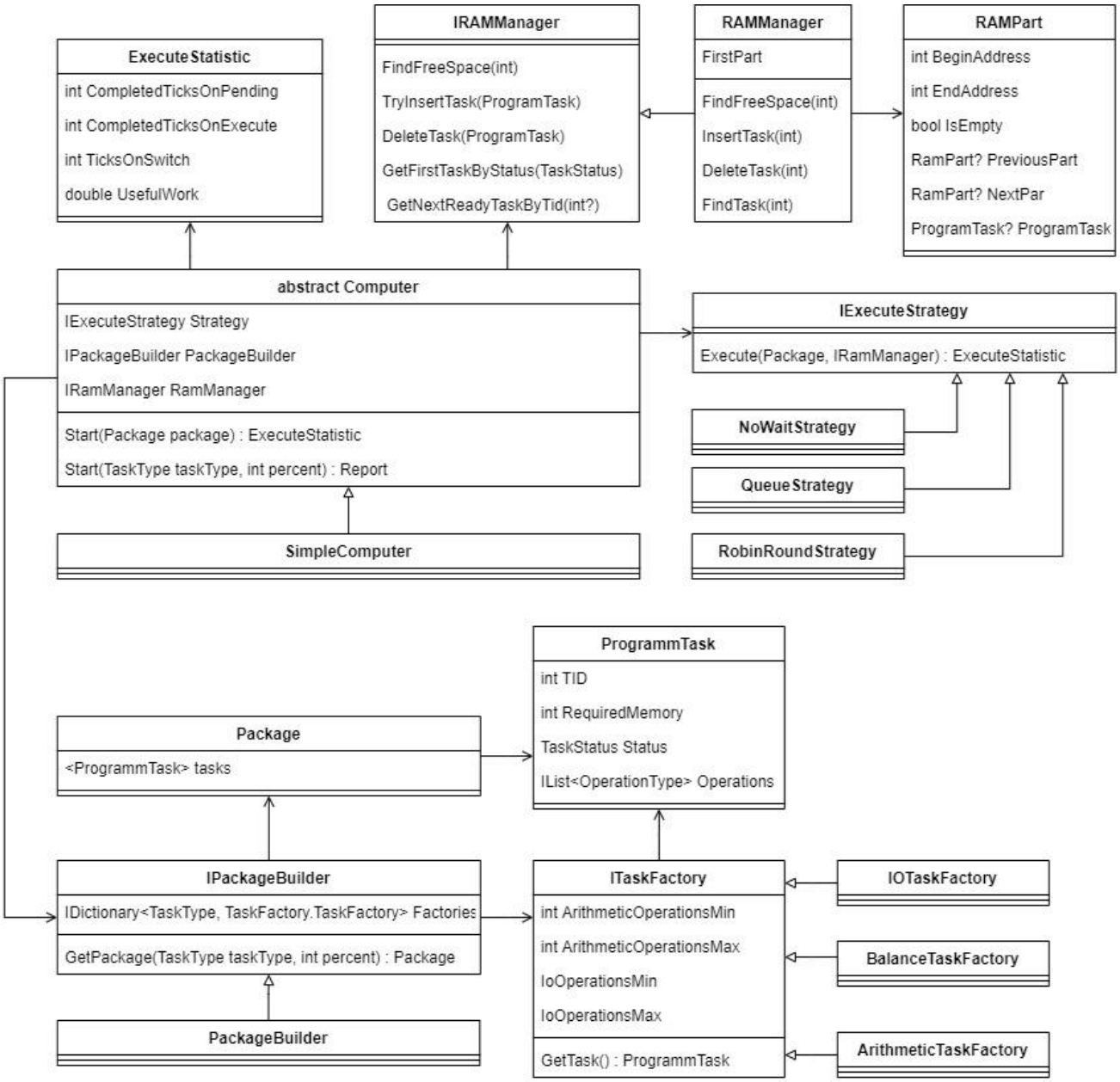
Боровцов Е.Г.

Барнаул 2023

Разработанное ПО

https://github.com/sablist99/PackageManager

Структура разработанного ПО



Описание ПО

В ходе работы было разработано программное обеспечение, моделирующее поведение операционной системы, выполняющей задания в пакетном режиме.

Предложено три стратегии обработки задач:

- Очередь – все задания выполняются строго по порядку

- Очередь, не ожидающая ввода-вывода – задания выполняются по порядку, но если требуется ввод-вывод, то происходит смена выполняемой задани

- RobinRound – каждой задаче выдается квант времени, который она может выполняться. Смена происходит в том случае, если нужен ввод-вывод или квант закончился.

Задачи генерируют **фабрики**. В зависимости от типа фабрики используются разные пороговые значения количества **операций**. Всего есть три категории:

- Small (1-10)

- Medium (11-20)

- Large (21-30)

```
public class ArithmeticTaskFactory : TaskFactory
{
    Ссылок: 2
    public override int ArithmeticOperationsMin { get => LargeQuantityMin; }
    Ссылок: 2
    public override int ArithmeticOperationsMax { get => LargeQuantityMax; }
    Ссылок: 2
    public override int IoOperationsMin { get => SmallQuantityMin; }
    Ссылок: 2
    public override int IoOperationsMax { get => SmallQuantityMax; }
    Ссылок: 5
    public override ProgramTask GetTask() => base.GetTask();
}
```

```
public class BalanceTaskFactory : TaskFactory
{
    Ссылок: 2
    public override int ArithmeticOperationsMin { get => MediumQuantityMin; }
    Ссылок: 2
    public override int ArithmeticOperationsMax { get => MediumQuantityMax; }
    Ссылок: 2
    public override int IoOperationsMin { get => MediumQuantityMin; }
    Ссылок: 2
    public override int IoOperationsMax { get => MediumQuantityMax; }
    Ссылок: 5
    public override ProgramTask GetTask() => base.GetTask();
}
```

```
public class IoTaskFactory : TaskFactory
{
    Ссылок: 2
    public override int ArithmeticOperationsMin { get => SmallQuantityMin; }
    Ссылок: 2
    public override int ArithmeticOperationsMax { get => SmallQuantityMax; }
    Ссылок: 2
    public override int IoOperationsMin { get => LargeQuantityMin; }
    Ссылок: 2
    public override int IoOperationsMax { get => LargeQuantityMax; }
    Ссылок: 5
    public override ProgramTask GetTask() => base.GetTask();
}
```

Для генерации **пакета** используется **Builder**, создающий пакет по типу задач и их процентом соотношении:

```
public interface IPackageBuilder
{
    Ссылок: 4
    public IDictionary<TaskType, AbstractTaskFactory> Factories { get; set; }
    Ссылок: 4
    public Package GetPackage(TaskType taskType, int percent);
}
```

То есть, например, GetPackage(TaskType.Arithmetic, 60) создаст пакет с 60% вычислительных операций. Оставшееся место займут сбалансированные задачи и задачи с преобладанием операций ввода-вывода.

Компьютеру на вход поступают **стратегия выполнения** и характеристики **пакета** (тип задач и процентная составляющая).

В качестве **оперативной памяти** выступает связка RamManager + RamPart. По структуре память представляет из себя двусвязный список. Где каждый элемент – **участок памяти**, характеризующийся начальным и конечным адресом, задачей (или признаком ее отсутствия), ссылками на предыдущий и следующий участками памяти.

```
Ссылок: 12
public class RamPart
{
    Ссылок: 9
    public int BeginAddress { get; set; } = 0;
    Ссылок: 9
    public int EndAddress { get; set; }
    Ссылок: 6
    public bool IsEmpty { get; set; } = true;
    Ссылок: 14
    public RamPart? PreviousPart { get; set; } = null!;
    Ссылок: 25
    public RamPart? NextPart { get; set; } = null!;
    Ссылок: 14
    public ProgramTask? ProgramTask { get; set; }
}
```

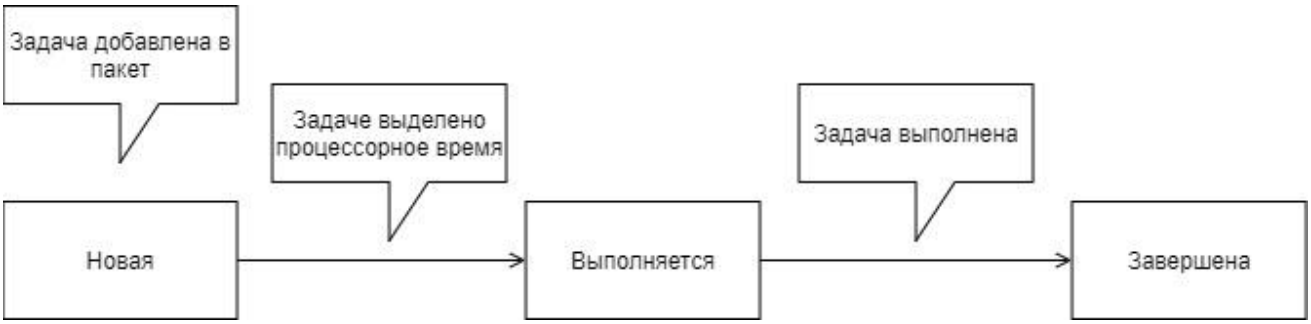
```
Ссылок: 7
public interface IRamManager
{
    Ссылок: 5
    public (bool, RamPart?) FindFreeSpace(int size);

    Ссылок: 4
    public bool TryInsertTask(ProgramTask task);

    Ссылок: 4
    public bool DeleteTask(ProgramTask task);

    Ссылок: 2
    public ProgramTask? GetFirstTaskByStatus(TaskStatus status);

    Ссылок: 2
    public ProgramTask? GetNextReadyTaskByTid(int? tid);

    Ссылок: 3
    protected RamPart? GetPartByTaskTid(int tid);

    Ссылок: 2
    protected bool ContainsTask(int tid);

    Ссылок: 2
    protected void InsertTask(RamPart freeSpace, ProgramTask task);
}
```

Так же предусмотрены настроечные параметры:

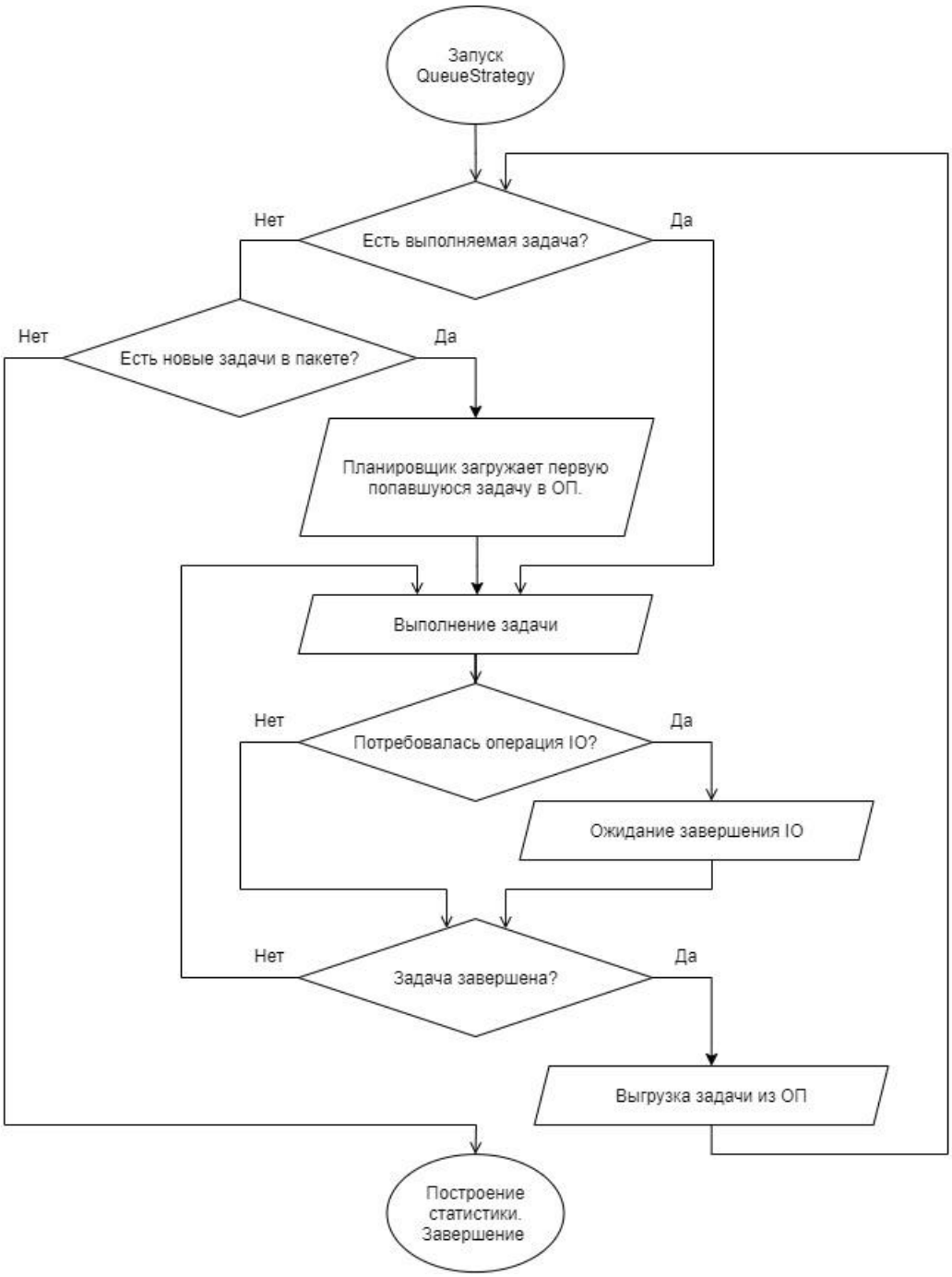
```
// Пороговые значения занимаемой задачами памяти
public const int RequiredMemoryMin = 50;
public const int RequiredMemoryMax = 500;

/// <summary>
/// Количество тиков на операцию ввода-вывода
/// </summary>
public static int PendingIOCost = 5;

/// <summary>
/// Количество тиков на переключение задачи
/// </summary>
public const int SwitchTaskCost = 2;

/// <summary>
/// Количество ОП в компьютере в МБ
/// </summary>
public const int RAMCapacity = 4096;

/// <summary>
/// Место, занимаемое ОС, в МБ
/// </summary>
public const int OperationSystemWeight = 1024;

/// <summary>
/// Количество задач, на которых проводится тестирование
/// </summary>
public static int TaskCount = 100;
```

Стратегии. Очередь

Статусы задач:

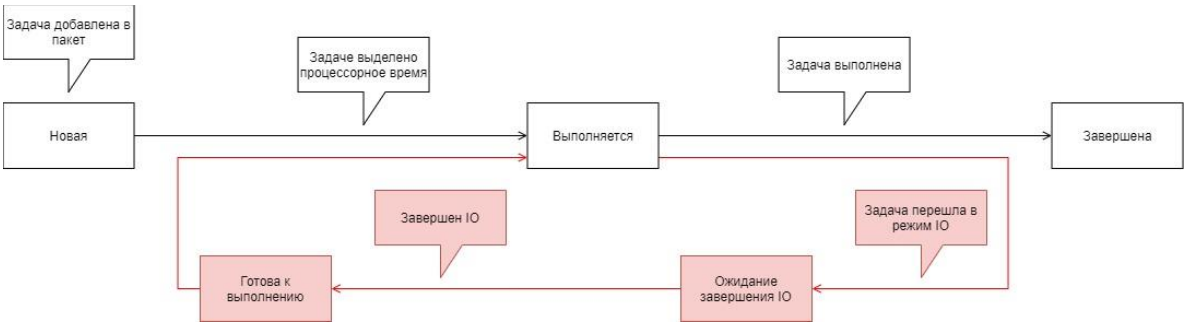


Алгоритм:



Стратегии. Очередь без ожидания

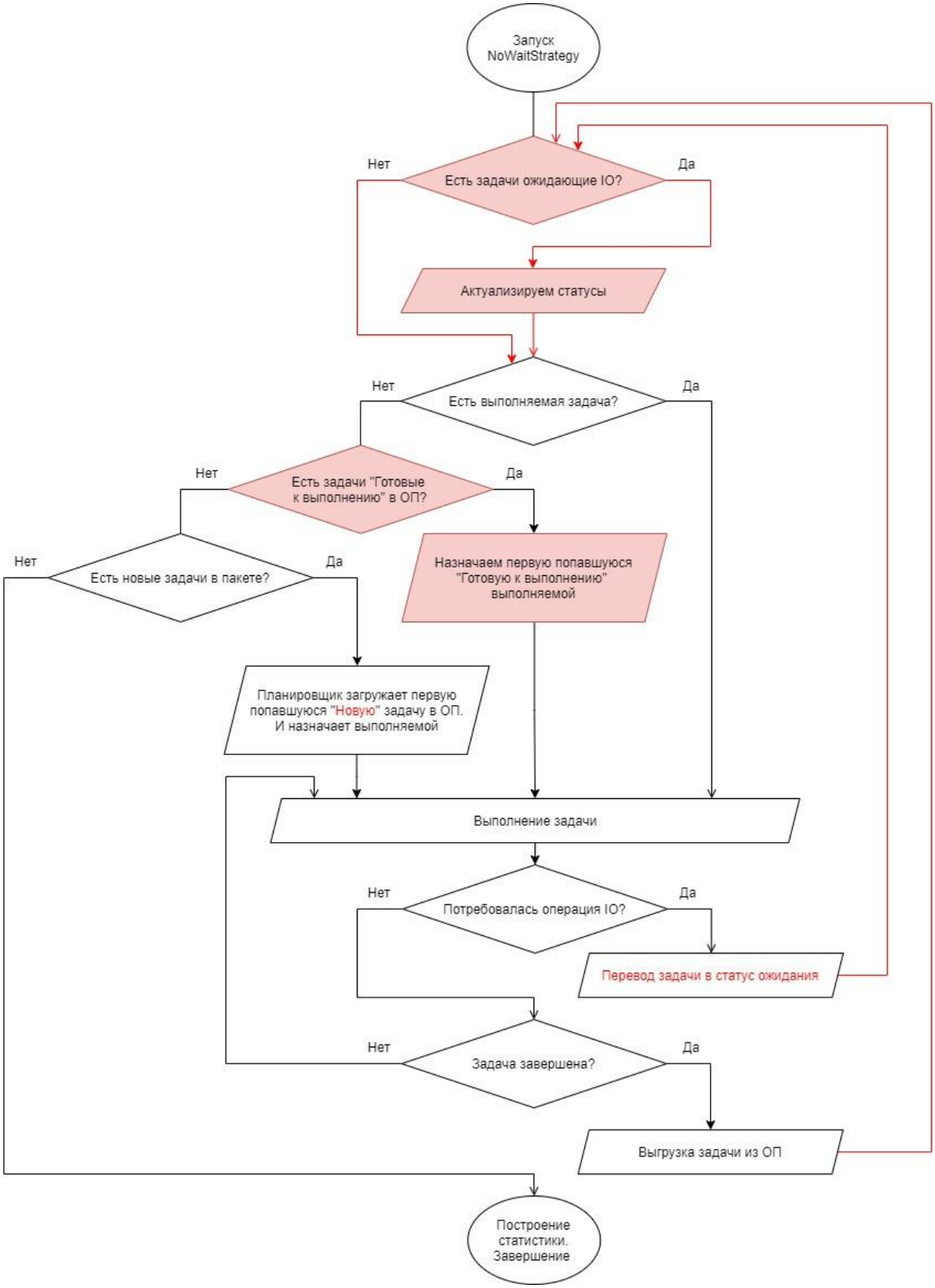
Статусы задач:



По сути, отличается тем, что теперь в случае начала операций ввода-вывода, процессор переключается на другую задачу в ОП. При этом, теперь каждый тик актуализируются статусы задач, который находятся в стадии ввода-вывода. Когда ввод-вывод заканчивается, они переходят в статус «Готова к выполнению». Процессор берет на исполнение «Первую попавшуюся» задачу из ОП в статусе «Готова к выполнению»:

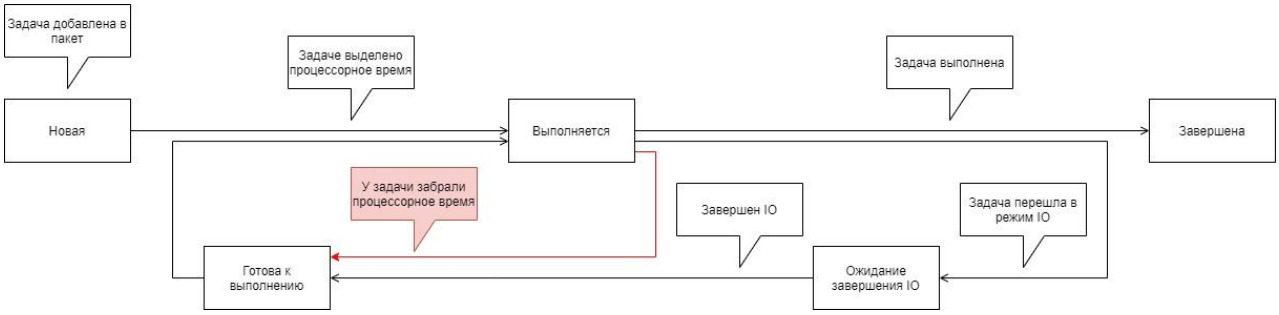
```
ссылок: 2
public ProgramTask? GetFirstTaskByStatus(Data.TaskStatus status)
{
    var currentPart = firstPart;
    do
    {
        if (currentPart.ProgramTask != null && currentPart.ProgramTask.Status == status)
        {
            return currentPart.ProgramTask;
        }

        currentPart = currentPart.NextPart;
    }
    while (currentPart != null);
    return null;
}
```

Алгоритм:


Запуск NoWaitStrategy
Нет
Есть задачи ожидающие IO?
Да
Актуализируем статусы
Нет
Есть выполняемая задача?
Да
Нет
Есть задачи "Готовые к выполнению" в ОП?
Да
Нет
Есть новые задачи в пакете?
Да
Назначаем первую попавшуюся "Готовую к выполнению" выполняемой
Планировщик загружает первую попавшуюся "Новую" задачу в ОП. И назначает выполняемой
Выполнение задачи
Нет
Потребовалась операция IO?
Да
Перевод задачи в статус ожидания
Нет
Задача завершена?
Да
Выгрузка задачи из ОП
Построение статистики. Завершение

Стратегии. RobinRound

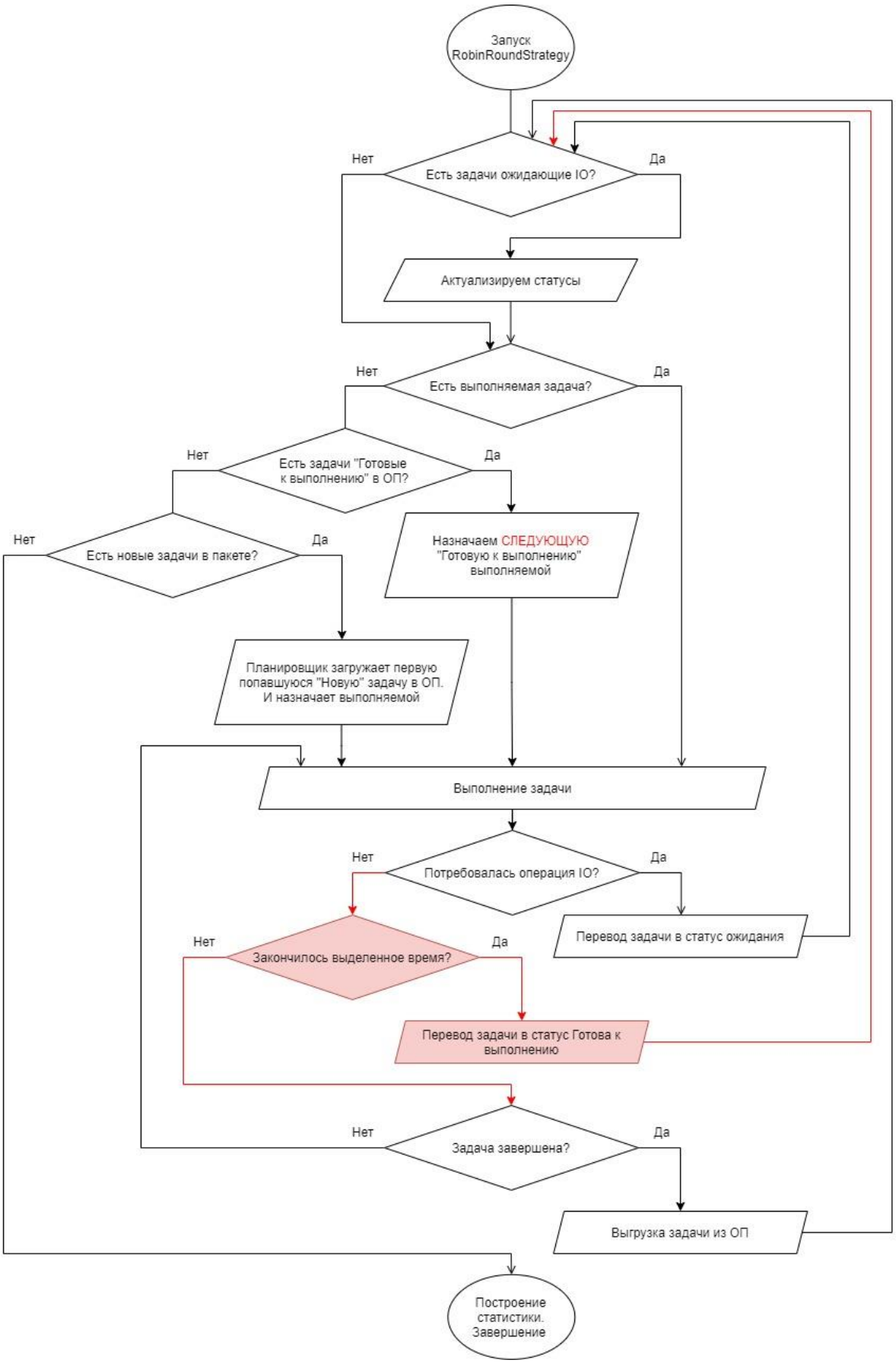
Статусы задач:



От «Очередь без ожидания» отличается тем, что теперь переключение на другую задачу может произойти в том случае, если закончился выделенный квант на выполнение. По умолчанию этот квант равен 10 тиков. Но в анализе будут рассмотрены и другие случаи. При этом задачи теперь выполняются не по принципу ОЧЕРЕДИ. При переключении берется не «Первая попавшаяся», а «Следующая» задача из ОП:

```
Ссылок: 2
public ProgramTask? GetNextReadyTaskByTid(int? tid)
{
    if (tid == null)
    {
        return null;
    }
    var sourcePart = GetPartByTaskTid((int)tid);
    var currentPart = sourcePart;
    if (currentPart == null)
    {
        // Если не смогли найти исходную задачу в памяти, то вообще ничего не возвращаем
        return null;
    }
    do
    {
        currentPart = currentPart.NextPart;
        if (currentPart == null)
        {
            // Это обеспечивает цикличность
            // FP -> P1 -> P2 -> SP -> null
            currentPart = firstPart;
        }
        if (currentPart.ProgramTask != null && currentPart.ProgramTask.Status == Data.TaskStatus.Ready)
        {
            return currentPart.ProgramTask;
        }
    }
    while (currentPart != sourcePart);
    return null;
}
```

Алгоритм:


Запуск RobinRoundStrategy
Нет
Есть задачи ожидающие IO?
Да
Актуализируем статусы
Нет
Есть выполняемая задача?
Да
Нет
Есть задачи "Готовые к выполнению" в ОП?
Да
Нет
Есть новые задачи в пакете?
Да
Назначаем СЛЕДУЮЩУЮ "Готовую к выполнению" выполняемой
Планировщик загружает первую попавшуюся "Новую" задачу в ОП. И назначает выполняемой
Выполнение задачи
Нет
Потребовалась операция IO?
Да
Перевод задачи в статус ожидания
Нет
Закончилось выделенное время?
Да
Перевод задачи в статус Готова к выполнению
Нет
Задача завершена?
Да
Выгрузка задачи из ОП
Построение статистики. Завершение

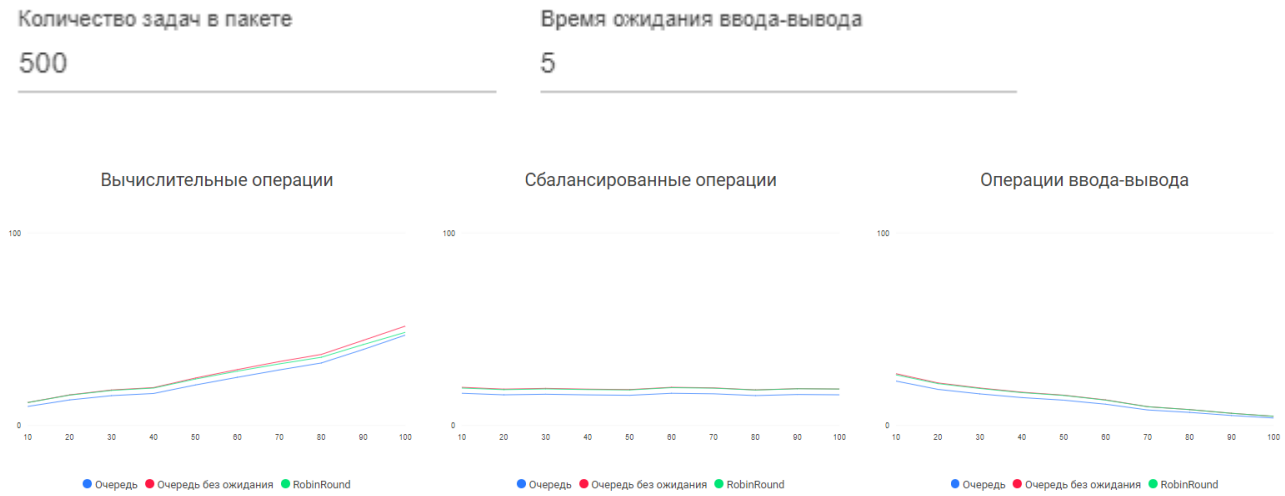
Анализ

В рамках одного графика для разных стратегий всегда использовался один и тот же пакет

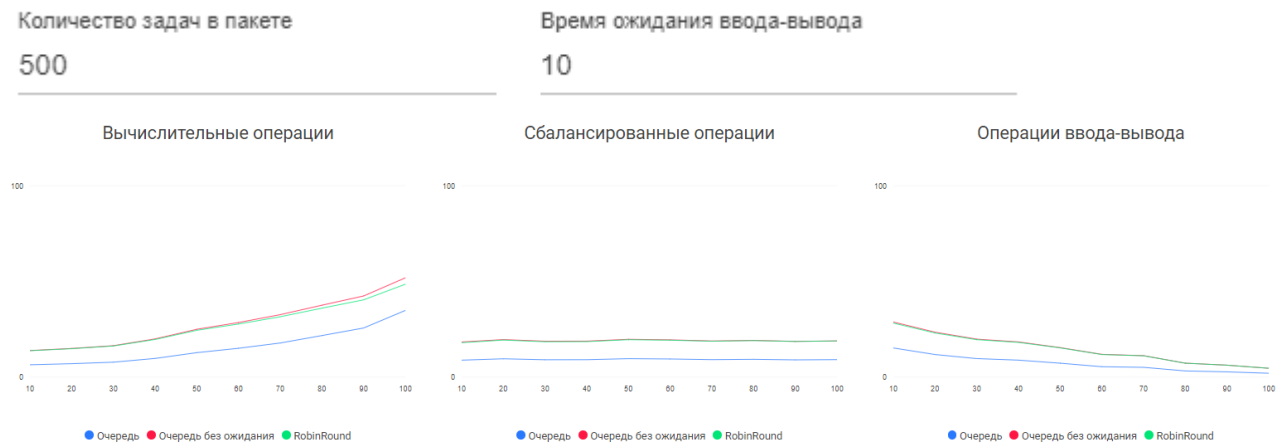
Построим графики зависимости КПД системы от типа операций и их относительного количества в пакете

(X - относительное количество операций в пакете; Y - КПД системы)

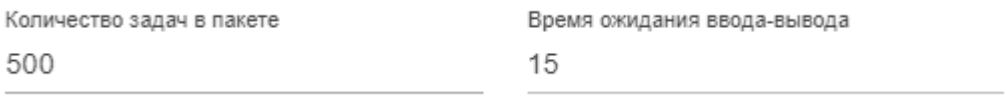
Тест 1.



Тест 2.



Тест 3.

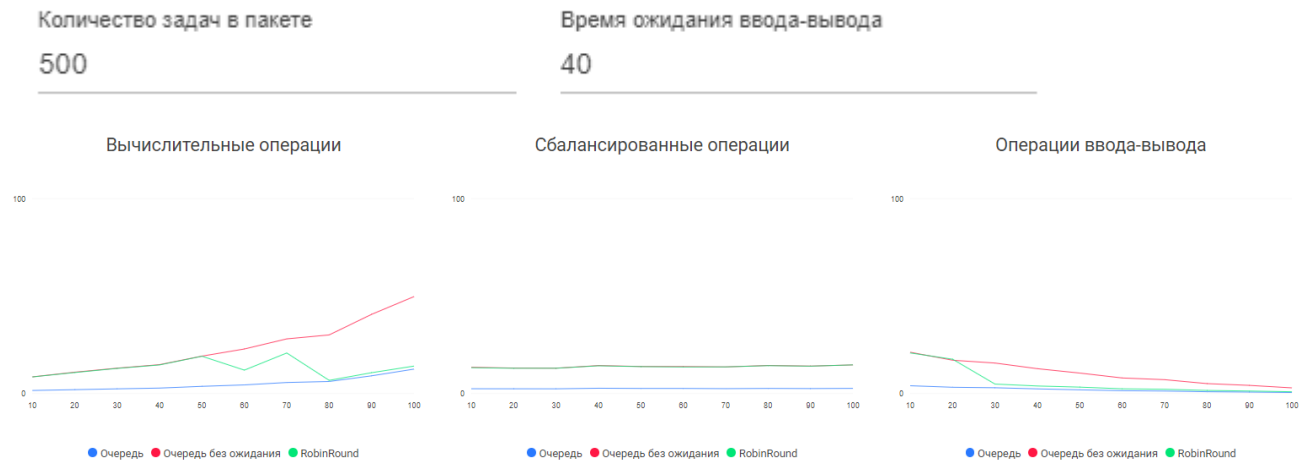


По графикам видно, что, при преобладании вычислительных операций, лучше справляется «Очередь без ожидания». Это можно объяснить тем, что при маленьком кванте, выделенному стратегии RobinRound, процессор тратит лишнее время на выполнение переключения задач.

При сбалансированном количестве задач и при преобладании задач ввода-вывода стратегии показывают практически идентичные показатели. Потому что «Очередь без ожидания» будет совершать те же переключения между задачами из-за того, что операции ввода-вывода будут встречаться чаще.

Наглядный пример, как производительность RobinRound может «просесть».
Тест 4.

Тест 5.



Думаю, что это связано с тем, что «Очередь без ожидания» целенаправленно добивает задачи по порядку. А «RobinRound» работает надо всеми помаленьку. И в какой-то момент времени получается так, что все задачи в ОП находятся в стадии «Ожидания ввода-вывода». Соответственно, «RobinRound» вынужден простаивать. В то время, как «Очередь без ожидания» просто подгружает новые задачи в ОП и работает над ними.

Тогда возникает вопрос, а как оптимально подобрать квант времени для RobinRound?

Тест 6.

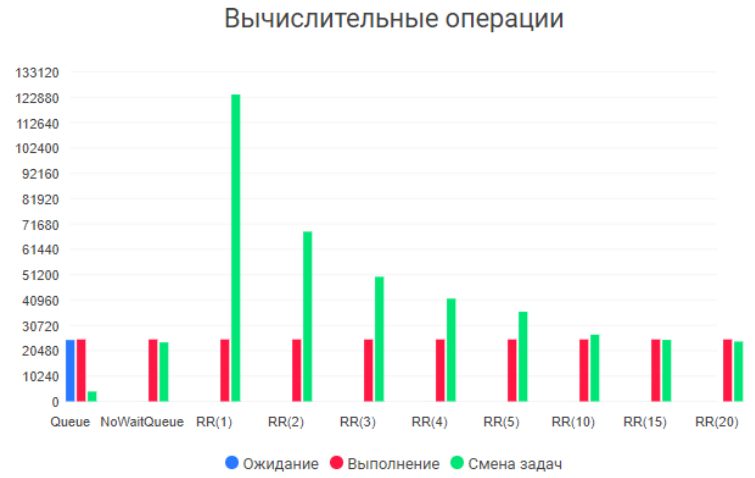
Примечание – красный столбец будет всегда одной высоты, потому что это «Выполнение» - полезная работы, не зависимо от того, какая стратегия используется.

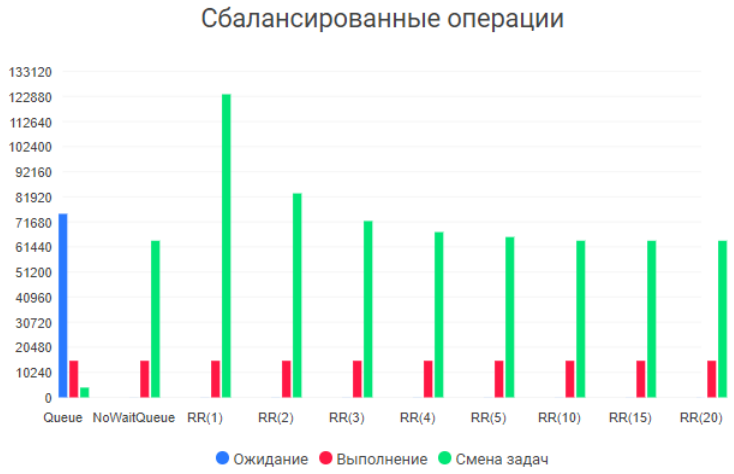
Количество задач в пакете

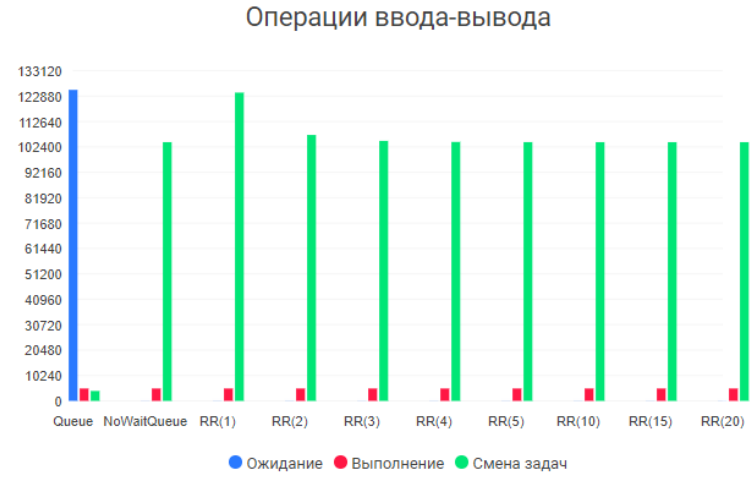
500

Время ожидания ввода-вывода

5





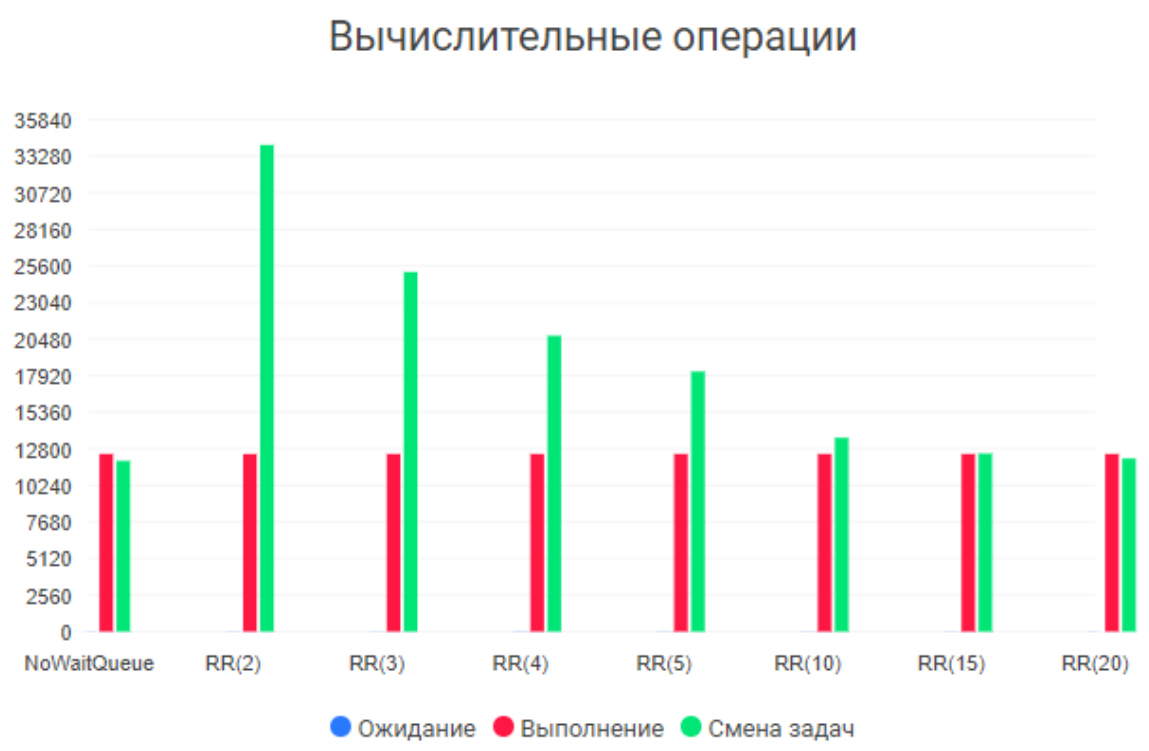


Глядя на диаграммы, можно сделать выводы:

1) Делается необоснованно много переключений между задачами при кванте равном 1.
2) При преобладании вычислительных операций имеет смысл увеличивать квант, чтобы как можно больше работать над задачей, не тратя время на переключение. Тем более, что вероятность встретить операцию ввода-вывода мала.
3) При преобладании операций ввода-вывода квант практически ни на что не влияет, потому что переключение происходит до того, как он истечет
4) При преобладании операций ввода-вывода, не зависимо от стратегии, КПД будет низким из-за частого переключения задач или простоя
5) При быстрых операциях ввода-вывода практически нет простоя системы

Для больше наглядности уберу из выборки алгоритмы Queue и RR(1)

Подтвердилась гипотеза, что «Очередь без ожидания» тратит меньше времени на переключения при вычислительных операциях:



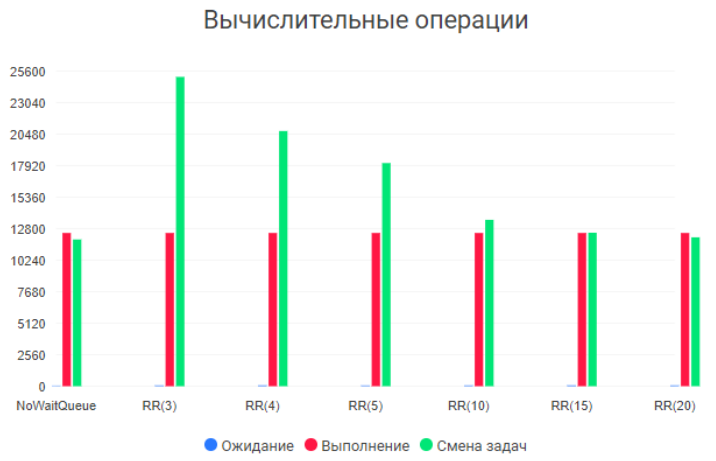
Смоделируем простой системы, увеличив длительность операции ввода-вывода

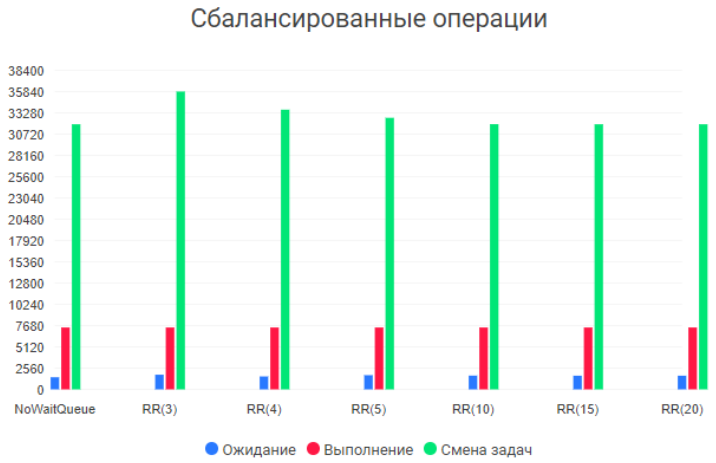
Количество задач в пакете

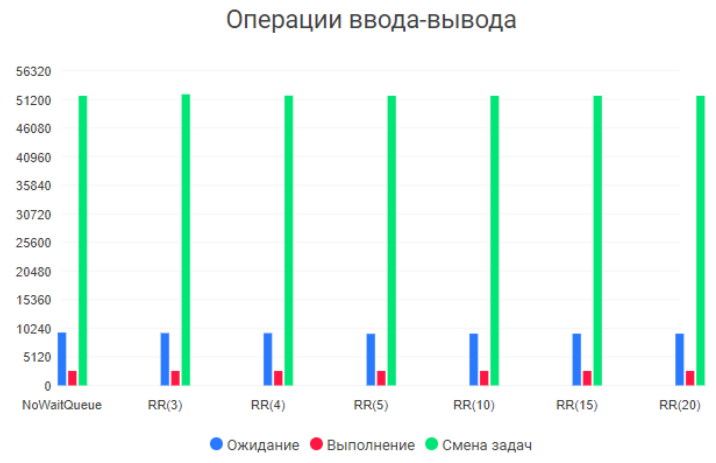
500

Время ожидания ввода-вывода

20







Интересно, что при операциях ввода-вывода, простой занимает больше времени, чем полезная работа. При этом стратегия не имеет значения.

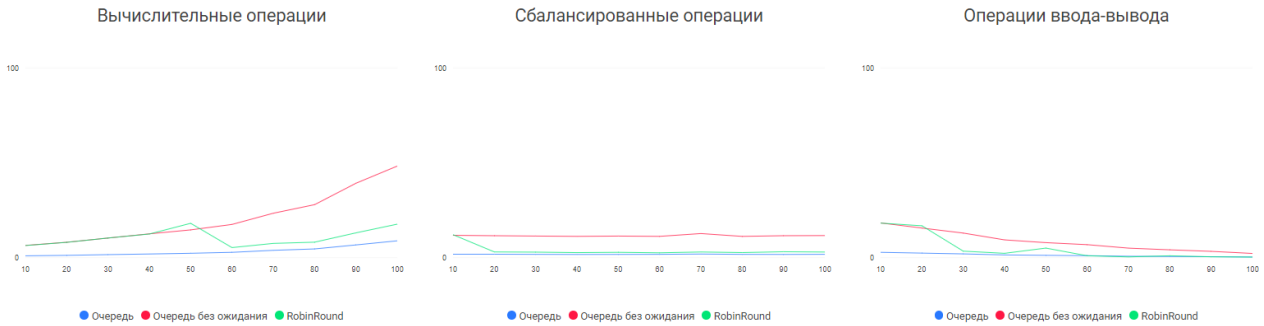
Усугубим ситуацию, создав простой при вычислительных операциях

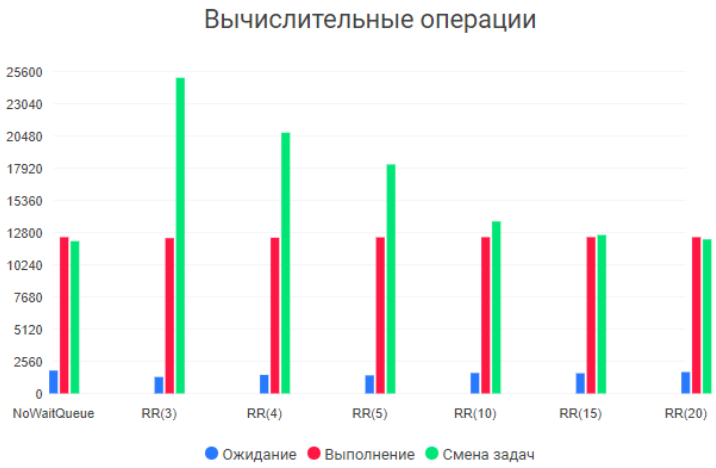
Количество задач в пакете

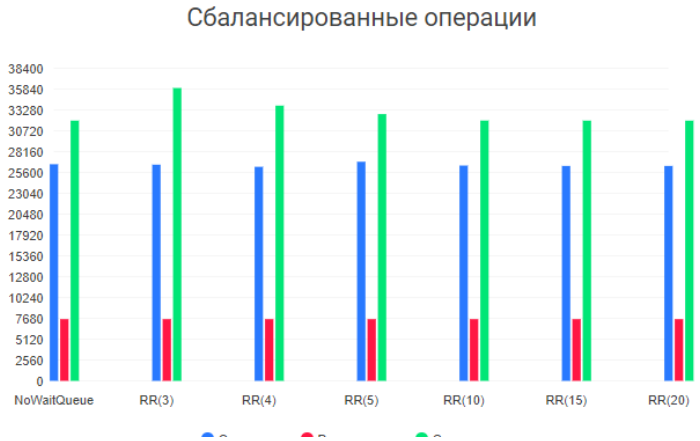
500

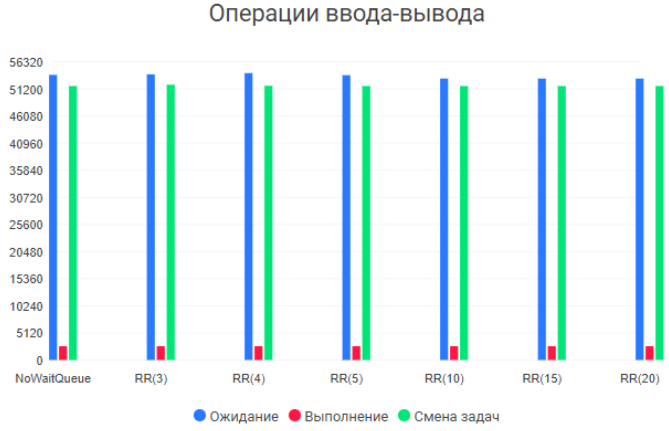
Время ожидания ввода-вывода

60









В дополнение к вышеперечисленным фактам можно сказать, что алгоритм «Очередь без ожидания» более устойчив к операциям ввода-вывода, чем «Очередь» и «RobinRound»