

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет

вычислительной техники

Кафедра

«Вычислительная техника»

Направление подготовки: 09.04.01 «Информатика и вычислительная техника»
Магистерская программа: «Вычислительные машины, комплексы, системы и сети»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

на тему

**Моделирование ядра многопроцессорных
операционных систем с аппаратной
поддержкой**

Студент Кор 03.06.2020 Королев Богдан Дмитриевич
(подпись, дата) (ФИО полностью)
Руководитель Р.А. Бикташев
(подпись, дата) (фамилия, инициалы)
Нормоконтролёр А.В. Кучин
(подпись, дата) (фамилия, инициалы)
Рецензент доцент каф. ВЛН А. И. Мартышкин 03.06.2020
ПензГТУ (должность, место работы) (подпись, дата)
(фамилия, инициалы)

Работа допущена к защите (протокол зас. кафедры от 03.06.2020 № 12)

Заведующий кафедрой М.А. Митрохин
(подпись) (фамилия, инициалы)

Работа защищена с отметкой отлично (протокол заседания ГЭК от 16.06.2020 № 10)

Секретарь ГЭК Е.И. Гурин
(подпись) (фамилия, инициалы)

Пенза 2020

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет

вычислительной техники

Кафедра

«Вычислительная техника»

«Утверждаю»

Заведующий кафедрой

 М.А. Митрохин

«08» ноября 2018г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
МАГИСТРА

1. Студент Королев Богдан Дмитриевич гр. 18ВВм1 факультета
вычислительной техники, направления 09.04.01

2. Тема работы «Моделирование ядра многопроцессорных операционных
систем с аппаратной поддержкой».

Тема утверждена приказом ПГУ №1012/о от "8" ноября 2018 г.

3. Руководитель магистерской диссертации к.т.н., профессор Бикташев Р. А.

4. Задание на работу (назначение разработки, исходные данные и т.п.)

Назначение разработки: моделирование ядра многопроцессорных
операционных систем с аппаратной поддержкой.

Разработать модели планировщиков задач для многопроцессорных
систем и измерить их временные характеристики.

5. Перечень подлежащих разработке вопросов

1. Анализ алгоритмов планирования и диспетчеризации задач в
многопроцессорных вычислительных системах.

2. Разработка автоматных моделей на основе языка НДА и VHDL-моделей приоритетных планировщиков/диспетчеров задач с глобальной и локальными приоритетными очередями для получения их вероятностно-временных характеристик.

3. Проведение вычислительных экспериментов с использованием разработанных моделей.

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1.	Анализ временных характеристик современных операционных систем	10%	26.09.2018- 15.01.2019	
2.	Изучение методов планирования в многопроцессорных системах	15%	16.01.2019- 22.03.2019	
3.	Анализ требований на разработку	10%	23.03.2019- 14.06.2019	
4.	Разработка VHDL -моделей	15%	15.06.2019- 10.10.2019	
5.	Тестирование моделей	15%	11.10.2019- 5.01.2020	
6.	Отладка моделей	10%	6.01.2020- 17.03.2020	
7.	Замер временных характеристик моделей	15%	18.03.2020- 9.05.2020	
8.	Оформление выпускной работы	10%	10.05.2020- 10.06.2020	

Дата выдачи задания "3" октября 2018 г.

Руководитель магистерской диссертации  3.10.2018 Бикташев Р. А.
(подпись, дата)

Задание к исполнению принял студент  3.10.2018 Королев Б. Д.
(подпись, дата)

Работу к защите допустить
Декан факультета  3.10.2018 Фионова Л.Р.
(подпись, дата)

РЕФЕРАТ

Магистерская работа содержит 165 листов, 49 рисунков, 13 источников, 10 приложений.

МНОГОПРОЦЕССОРНАЯ СИСТЕМА, ОС, ДИСПЕТЧЕР, ПЛАНИРОВЩИК, ДИСПЕТЧЕРИЗАЦИЯ, ПРИОРИТЕТНОЕ ПЛАНИРОВАНИЕ, ГЛОБАЛЬНАЯ ОЧЕРЕДЬ, ЛОКАЛЬНАЯ ОЧЕРЕДЬ, МОДЕЛЬ.

Цель работы – получение временных характеристик и сравнительная оценка планировщиков/диспетчеров задач с аппаратной реализацией.

В результате проведенной работы были разработаны VHDL-модели планировщиков задач с глобальной и локальными приоритетными очередями вместе с модулями сбора статистики для оценки вероятностно-временных характеристик моделей.

Методы моделирования диспетчеров задач многопроцессорных систем могут быть применены в учебном процессе кафедры «Вычислительная техника» Пензенского государственного университета при осуществлении основной образовательной программы по дисциплинам «Вычислительные системы» для магистрантов, которые обучаются по направлению «Информатика и вычислительная техника».

Содержание

Введение	5
1 Обзор методов планирования	10
1.1 Временные характеристики существующих систем реального времени	10
1.1.1 Время реакции системы (латентность)	10
1.1.2 Время переключения контекста	10
1.1.3 Дедлайн	11
1.1.4 Джиттер	11
1.2 Существующие алгоритмы приоритетного планирования	11
1.2.1 Алгоритмы, основанные на приоритетах (классификация по принципу выбора процесса на выполнение из очереди)	11
1.2.2 Смешанные алгоритмы планирования (квантование с приоритетами)	13
1.3 Алгоритмы, основанные на приоритетных очередях задач	14
1.3.1 Разделение времени	15
1.4 Анализ схем аппаратного приоритетного выбора готовых к выполнению задач	18
1.4.1 Схема приоритетного выбора «полное дерево»	20
1.4.2 Схема последовательного приоритетного выбора	21
1.4.3 Схема гибридного приоритетного выбора	22
1.5 Выводы	24
2 Моделирование алгоритмов приоритетного планирования на языке недетерминированных автоматов	25
2.1 Возможности языка НДА для формального описания алгоритмов управления параллельными процессами и ресурсами	25
2.2 Формализация алгоритмов управления параллельными процессами на основе использования механизма «рандеву»	32
2.3 Формализация алгоритмов управления параллельными процессами с использованием механизма монитора и согласующего кольцевого буфера ...	37
2.4 Выводы	44
3 Аппаратная реализация алгоритмов приоритетного планирования	46

3.1 Проблемы управления ресурсами и процессами в многопроцессорных системах	46
3.2 Структура аппаратного планировщика с глобальной очередью	56
3.3 Моделирование аппаратного планировщика с глобальной очередью на языке VHDL	59
3.4 Структура аппаратного планировщика с локальными очередями	72
3.5 Моделирование аппаратного планировщика с локальными очередями на языке VHDL	74
3.6 Получение временных диаграмм и временных характеристик	84
3.7 Выработка критериев сравнительной оценки планировщиков	92
3.8 Выводы	96
Заключение	97
Список используемых источников	99
Приложение А Модуль блока генерации задач	101
Приложение В Модуль блока управления очередью планировщика с глобальной очередью	104
Приложение С Модуль блока приоритетной очереди планировщика с глобальной очередью	107
Приложение D Модуль блока синхронизации и назначения задач планировщика с глобальной очередью	121
Приложение E Модуль блока выбора процессора планировщика с глобальной очередью	128
Приложение F Модуль блока процессора	131
Приложение G Модуль блока назначения задач планировщика с локальными очередями	134
Приложение H Модуль блока локального диспетчера с приоритетной очередью планировщика с локальными очередями	137
Приложение I Модуль выбора очереди планировщика с локальными очередями	151
Приложение J Презентация	153

Введение

При разработке и совершенствовании вычислительных систем (ВС) необходимо проводить исследования ее характеристик. Для изучения характеристик ВС применяют программные модели, не являющиеся физическим устройством, а описанные данными и алгоритмами в памяти ЭВМ. Следовательно, их построение и изменение производится без финансовых затрат.

Проблема эффективной диспетчеризации пакетов задач является одним из наиболее важных аспектов эффективного использования вычислительных ресурсов при распараллеливании вычислительного процесса на уровне задач. Использование оптимальных или, по крайней мере, приближенных расписаний для реализации крупных пакетов задач на многопроцессорных и многомашинных ВС позволяет сократить время исполнения таких пакетов и уменьшить при этом затраты машинного времени.

Таким образом, разработка и исследование различных методов, моделей и алгоритмов, позволяющих строить эффективные расписания для различных типов пакетов задач и структур параллельных ВС приобретает особую актуальность.

В данной работе предлагается метод выбора приоритетного планировщика задач для встроенных многоядерных систем реального времени, основанный на имитационных моделях массового обслуживания, который позволяет оценивать как характеристики производительности планировщиков задач, так и их функционирование.

Задача планирования процессов в многопроцессорной системе, в отличие от однопроцессорной, является двумерной. В период работы функции назначения задачи осуществляется функция выделения целевого процессора,

вследствие чего возникают дополнительные издержки времени. Помимо этого, в планировщике задач с разделением во времени формируется процесс перезагрузки кэш-памяти, который взаимосвязан с переключением задач, когда прерванная задача с достаточно высокой вероятностью способна быть направлена для продолжения обслуживания в иной процессорный узел. Данный процесс повышает частоту кэш-промахов, и в абсолютно любом случае отрицательно воздействует на производительность многопроцессорной системы. Другое противоречие в диспетчерах задач с разделением в пространстве, которое требуется разрешить, взаимосвязано с тем, что автоматическое поддержание различного уровня нагрузки процессорных узлов невозможно. Решение представленной задачи лежит в плоскости принудительного обеспечения баланса методами планировщика/диспетчера, что в любом случае порождает дополнительные временные расходы на исполнение определенной программы балансирования. Представленные выше факторы неизбежно формируют рост трудности программной реализации ядра операционных систем, приводят к уменьшению общей производительности многопроцессорной системы, а помимо этого уменьшают надежность и безопасность.

Во время обращения к критическим ресурсам операционной системы, а помимо этого и в процессе планирования и диспетчеризации задач к данным ресурсам возникают очереди процессов, которые ожидают выполнения иных процессов. По этой причине для оценки вероятностно-временных характеристик многопроцессорных систем применяют методы теории массового обслуживания.

Большая часть задач проектирования многопроцессорных вычислительных систем связаны с распределением ресурсов среди конкурирующих требований, притом, что моменты возникновения данных требований носят случайный характер. Это обстоятельство приводит к невозможности предсказать времена выполнения задач в многопроцессорных

системах, что особенно важно для систем реального времени. Решением данных проблем выступает аналитическое, имитационное и функционально-логическое моделирование, дающее возможность ускорить процесс проектирования, в частности, аппаратных планировщиков задач в составе ядра специализированных многопроцессорных операционных систем, а также уменьшить себестоимость проекта. Представленные аспекты обуславливают актуальность выбранной темы исследования.

Объект исследования – алгоритмы планирования и диспетчеризации задач в многопроцессорных вычислительных системах.

Предмет исследования – математическое и функционально-логическое моделирование планировщиков/диспетчеров задач в составе ядра многопроцессорных операционных систем с использованием языка недетерминированных автоматов (НДА), систем массового обслуживания и языка моделирования аппаратуры VHDL.

Цель данной работы – получение временных характеристик и сравнительная оценка планировщиков/диспетчеров задач с аппаратной реализацией.

С целью достижения поставленной цели в диссертационной работе решаются следующие задачи:

1) разработка автоматной модели на основе языка НДА и на её основе VHDL-модели приоритетного планировщика/диспетчера задач и оценка характеристик аппаратной реализации;

2) разработка имитационной модели приоритетного планировщика/диспетчера задач для получения его вероятностно-временных характеристик;

3) разработка функционально - логической модели планировщика/диспетчера задач для оценки возможности реализации аппаратными средствами;

4) проведение вычислительных экспериментов с использованием разработанных моделей.

Методы исследований. Для того чтобы решить поставленные задачи в диссертационной работе применены методы:

- теории недетерминированных автоматов.
- теории аналитического и имитационного моделирования;
- теории вероятностей;

Научная новизна работы состоит в следующем:

1) разработаны на основе языка НДА автоматные модели приоритетных планировщиков/диспетчеров задач с разделением времени с локальными и глобальными очередями в составе многопроцессорной системы.

2) созданы на основе НДА модели VHDL – программы аппаратной реализации планировщиков/диспетчеров задач с разделением времени с локальными и глобальными очередями в составе многопроцессорной системы и произведена оценка характеристик этих моделей;

3) разработан программный модуль сбора статистики в составе VHDL – программы для оценки вероятностно-временных характеристик подсистемы «диспетчер задач – процессорный узел».

Практическая значимость работы.

Применение предложенных аналитической и программной моделей обеспечивает сравнительную оценку временных характеристик планировщика/диспетчера задач на этапе эскизного проектирования, без проведения дорогостоящих физических экспериментов.

Реализация и внедрение результатов работы. Методы моделирования диспетчеров задач многопроцессорных систем будут применены в учебном процессе кафедры «Вычислительная техника» Пензенского государственного университета при осуществлении основной образовательной программы по

дисциплинам «Вычислительные системы» для магистрантов, которые обучаются по направлению «Информатика и вычислительная техника».

Достоверность результатов работы подтверждается имитационным моделированием, применением известных методов математического моделирования, а помимо этого докладами на научных конференциях и публикацией в научных журналах.

На защиту выносятся.

1. Алгоритмы приоритетного планирования и диспетчеризации с глобальной и локальной очередями задач в составе многопроцессорных операционных систем.
2. Программный комплекс, включающий в себя VHDL-программу функционально-логического моделирования для оценки функционирования, а также программу имитационного моделирования и определения вероятностно-временных характеристик многопроцессорных приоритетных планировщиков задач.
3. Результаты комплексных исследований вероятностно-временных характеристик планировщиков задач с использованием разработанных математических методов и комплекса программ.

1 Обзор методов планирования

Для определения методов планирования моделируемых планировщиков задач необходимо выяснить требуемые временные характеристики операционных систем реального времени. На основе материалов данного раздела будут определены вычисляемые в последующем временные характеристики моделируемых диспетчеров задач.

1.1 Временные характеристики существующих систем реального времени

1.1.1 Время реакции системы (латентность)

Латентность приводится как основной параметр почти всеми производителями систем реального времени. Она выражается в скорости системы реакции на входные воздействия. Внешние сигналы генерируются датчиками, передаваясь на управляющий компьютер оповещают его о случившемся событии, который генерирует запрос на прерывание. Получив сигнал прерывания, система запускает программу обработки прерывания. Интервал времени от события на объекте до первой инструкции обработчика этого события называется латентностью системы. Латентность также необходимо оценивать для худшей для системы ситуации, при которой процессоры загружены, происходят другие прерывания, и система выполняет какие-либо действия.

1.1.2 Время переключения контекста

Так как в ОСРВ заложена возможность одновременного выполнения различных задач на нескольких процессорах, необходимо вычислять накладные расходы системы при переключении процессора с одной задачи на другую.

Время, которая система затрачивает на передачу управления от одной задачи к другой и называется временем переключения контекста.

1.1.3 Дедлайн

Дедлайн – крайний срок выполнения задачи. В ОСРВ задачи должны быть выполнены к строго определенному моменту времени, иначе это может приводить к катастрофическим последствиям или выполнение задачи просто теряет смысл. В СРВ может одновременно поступать несколько задач, в таком случае каждая ни одна из них не должна обрабатываться дольше заданного для этой задачи времени.

1.1.4 Джиттер

Джиттер - разброс значений времени отклика. Различают джиттер запуска – время от готовности исполнения задачи до начала её исполнения, а также джиттер вывода – время от готовности исполнения задачи до окончания выполнения задачи. Независимо от пришедшего в систему сигнала в идеале времена обработки должны быть одинаковыми. Таким образом, ОСРВ джиттер должен быть максимально минимизирован.

1.2 Существующие алгоритмы приоритетного планирования

1.2.1 Алгоритмы, основанные на приоритетах (классификация по принципу выбора процесса на выполнение из очереди)

Приоритет – число, характеризующее степень важности процесса, а, следовательно, его количество привилегий. Чем выше приоритет процесса, тем меньше времени он будет проводить в очередях. Приоритет может быть целым или дробным, положительным или отрицательным числом. Существуют системы, в которых более привилегированные процессы обладают большим

значением приоритета и наоборот, системы, в которых чем меньше значение приоритета у процесса, тем выше сам приоритет процесса.

Приоритет также может назначаться или изменяться администратором системы при порождении процесса или вычисляться самой ОС по определенным правилам. По возможности изменения приоритета задач в течение их жизни различают ОС с динамическими и фиксированными приоритетами. В системах с динамическими приоритетами приоритет процесса может меняться при обращении к ОС, при изменении его пользователем с помощью определенных команд или другими процессами.

Существует несколько разновидностей алгоритмов приоритетного планирования: с относительными приоритетами и с абсолютными приоритетами.

При обоих случаях из очереди процессов выбирается наиболее приоритетный, однако процедура определения момента смены этого процесса различна.

В системах с относительными приоритетами процесс, поступивший на обслуживание в процессор, обслужится даже если в это время в очереди появится другой более приоритетный процесс. Схема выбора и обслуживания процесса в системе с относительными приоритетами показана на рисунке 1.

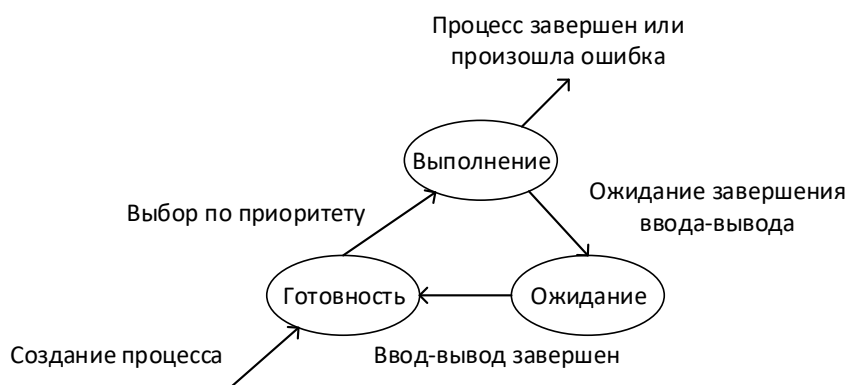


Рисунок 1 – граф состояний алгоритма обслуживания с относительными приоритетами

В системах с абсолютными приоритетами, если во время выполнения задачи в процессоре в очереди появится более приоритетная задача, выполнение текущей задачи будет приостановлено и начнется выполнение более приоритетной задачи, а текущая задача переходит в состояние готовности и будет выгружена обратно в очередь задач (рисунок 2).

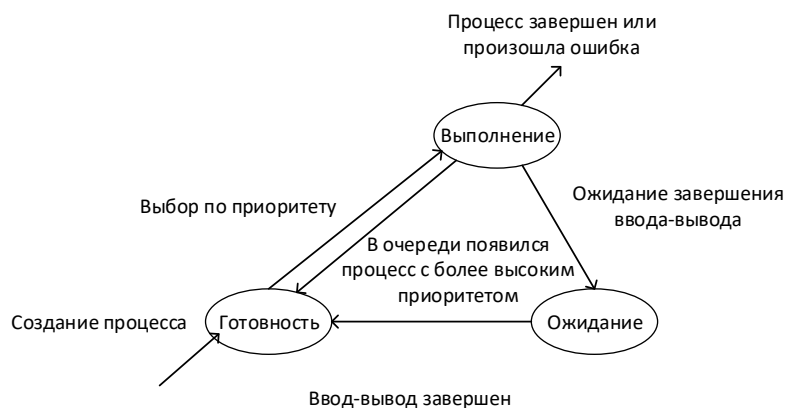


Рисунок 2 – граф состояний алгоритма обслуживания с абсолютными приоритетами

1.2.2 Смешанные алгоритмы планирования (квантование с приоритетами)

Во многих операционных системах (Windows NT и Unix System V Release 4) алгоритмы планирования построены на приоритетах в сочетании с алгоритмами квантования. Например, в основе планирования в диспетчере лежит квантование, но величина кванта и порядок его смены определяется приоритетностью процессов. В вышеприведенных примерах систем использована дисциплина вытесняющей многозадачности, основанная на использовании абсолютных приоритетов и квантования. Обработка задачи процессором в такой системе может быть отложена по причине смены кванта времени или поступлением в систему более высокоприоритетной задачи (Рисунок 3).

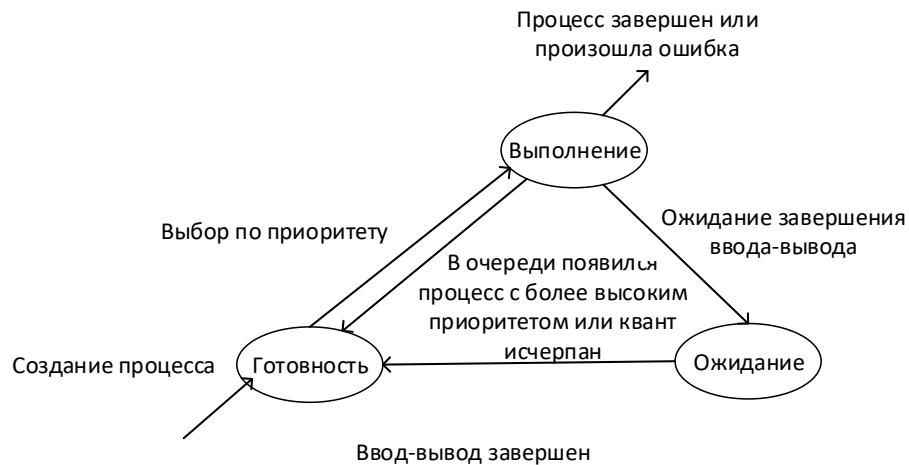


Рисунок 3 - граф состояний алгоритма обслуживания квантования с приоритетами

1.3 Алгоритмы, основанные на приоритетных очередях задач

Перед тем как перейти к рассмотрению вопросов планирования работы мультипроцессоров, нужно четко определить, что именно подвергается планированию. Во времена, когда все процессы были однопоточными, ответ был один: следовало планировать процессы, поскольку больше планировать было нечего. Но все современные операционные системы поддерживают многопоточные процессы, что значительно усложняет планирование. Играет роль и то, с какими потоками мы имеем дело: с потоками в пространстве ядра или с потоками в пользовательском пространстве. Если потоки образуются с помощью библиотеки, находящейся в пользовательском пространстве, и ядро о них ничего не знает, то планирование осуществляется, как и всегда, на основе процессов. Если ядро даже ничего не знает о существовании потоков, то вряд ли оно сможет заниматься их планированием. А вот с потоками в пространстве ядра все обстоит по-другому. Здесь ядро знает обо всех потоках и может выбирать из тех потоков, которые принадлежат процессу. В этих системах ядро стремится выбрать для выполнения поток, а процесс, которому он

принадлежит, играет в алгоритме выбора потока лишь незначительную роль (или вообще не играет никакой роли). Далее речь пойдет о планировании потоков, но, разумеется, в системах с однопоточными процессами или потоками, реализованными в пользовательском пространстве, объектом планирования являются процессы. Что именно подвергать планированию — процессы или потоки, не является единственным вопросом, касающимся планирования. В однопроцессорной системе планирование ведется в одном измерении. Здесь нужно многократно отвечать лишь на один вопрос: какой поток должен быть запущен следующим? В мультипроцессорных системах планирование ведется в двух измерениях. Планировщик должен решить, какой поток запускать и на каком центральном процессоре следует это сделать. Это дополнительное измерение существенно усложняет планирование на мультипроцессорах. Другим усложняющим фактором является то, что в некоторых системах все потоки не связаны друг с другом в силу принадлежности к разным процессам и не могут ничего сделать друг с другом. В других системах они сведены в группы, где все они принадлежат одному и тому же приложению и работают вместе [1].

1.3.1 Разделение времени

Обратимся к случаю планирования независимых потоков. Простейший алгоритм планирования для работы с независимыми потоками заключается в поддержке для готовых к работе потоков единой структуры данных для всей системы, возможно, в виде простого списка, но, скорее всего, в виде набора списков для потоков с разными приоритетами [1] (Рисунок 4).

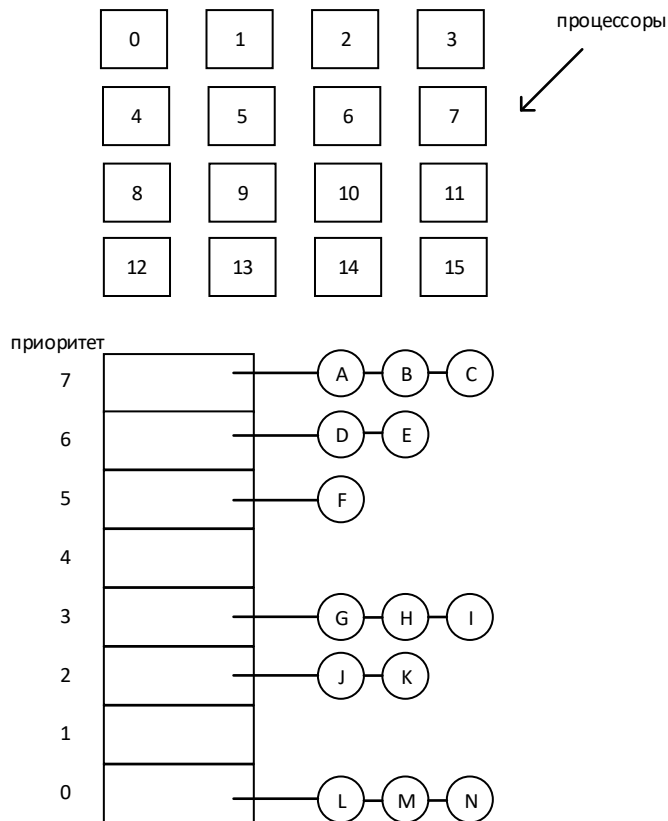


Рисунок 4 – единая структура данных для планирования работы мультипроцессора

Здесь показаны 16 центральных процессоров, которые в данный момент заняты работой, и набор из готовых к работе 14 потоков, имеющих разные приоритеты. Первым процессором, завершившим свою работу (или столкнувшимся с блокировкой своего потока), станет центральный процессор 4, который заблокирует очереди планируемых потоков и выберет поток А, имеющий наивысший приоритет (Рисунок 5).

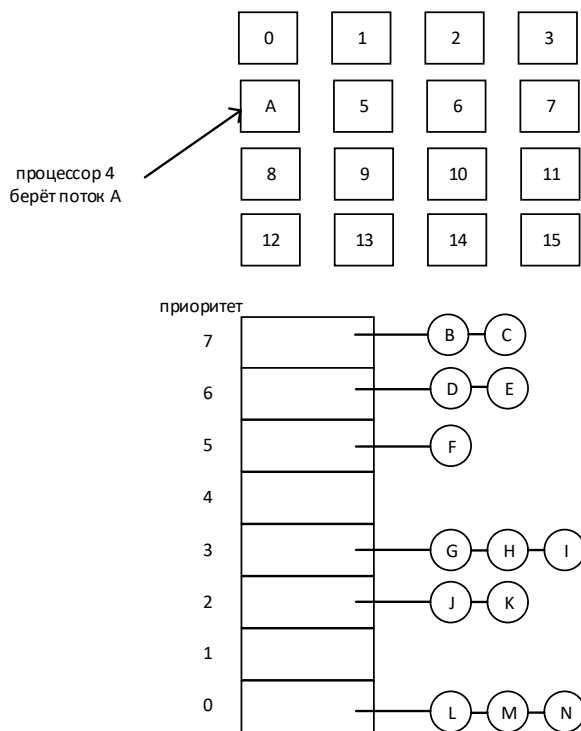


Рисунок 5 – блокировка свободного процессора процессом с наивысшим приоритетом

Затем освободится центральный процессор 12, который выберет поток В (Рисунок 6).

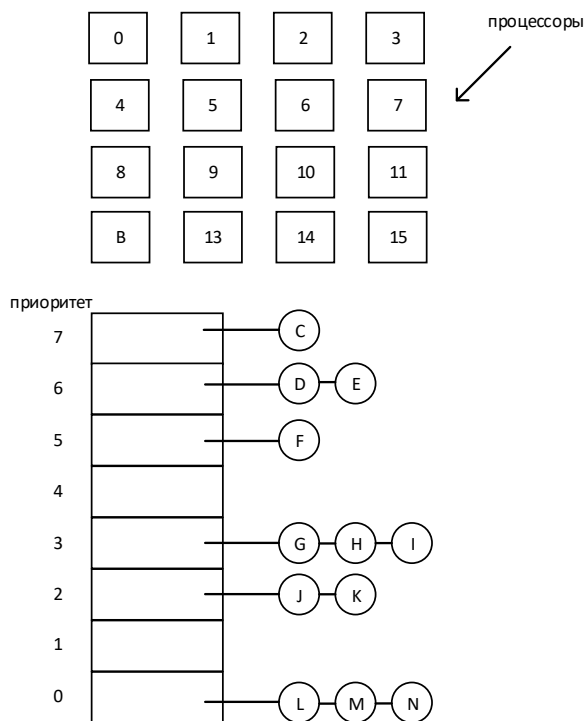


Рисунок 6 – последующая блокировка свободного процессора следующим процессом с наивысшим приоритетом

Наличие единой структуры данных позволяет процессорам работать в режиме разделения времени. Это явление обеспечивает сбалансированность нагрузки, проявляющуюся в отсутствии недостатка задач у одного процессора при перегрузке другого. Однако данная структура имеет ряд недостатков. Во-первых, потенциально возможная конкуренция процессоров к структуре данных при росте процессоров в системе. Во-вторых, блокировка процессов при ожидании ввода-вывода и переключении контекста.

1.4 Анализ схем аппаратного приоритетного выбора готовых к выполнению задач

Поскольку разрабатываемые планировщики задач предназначены для использования во встраиваемых системах реального времени, важно, чтобы они обеспечивали функции, используемые в этой предметной области. Приоритетное планирование также может быть представлено как планирование, управляемое событиями, так как блок планирования выбирает готовую задачу с наивысшим приоритетом, для таких событий, как окончание кванта процессорного времени, блокировка задачи и т.д.

Приоритеты для каждой задачи в алгоритме приоритетного планирования могут быть назначены с использованием либо схемы статического приоритета, либо динамического приоритета. В схеме со статическим приоритетом присваивают приоритеты априори с использованием алгоритма фиксированного приоритета, такого как Rate Monotonic (RM) или Deadline Monotonic (DM). Алгоритм RM и DM позволяют автоматически назначать (изменять) приоритеты во время выполнения задачи. Однако такая схема является сложной для реализации и для начала будут использованы статические фиксированные приоритеты, когда назначенные приоритеты задач не изменяются во время выполнения, если не изменяются временные параметры задач.

В схеме динамического приоритета алгоритм планирования динамически назначает приоритеты во время выполнения на основе параметров задачи, таких как время выпуска, крайний срок, время выполнения и т.д. Примером динамических алгоритмов являются алгоритмы "Первый самый ранний срок" (EDF) и "Первый самый низкий срок" (LLF).

Очевидно, что наилучшая реализация динамического алгоритма планирования в аппаратно-ориентированном планировщике реального времени, однако, пользуясь преимуществом того, что планирование может выполняться параллельно исполняемой программе без ее загрузки, решено реализовать сначала простой алгоритм со статическими фиксированными приоритетами. Причина такого решения заключается в том, что оно упрощает работу по разработке планировщика, поскольку этот проект посвящен разработке аппаратного планировщика, и не направлен на усовершенствование алгоритмом планирования.

Аппаратный планировщик состоит из простых модулей, которые требуют минимального объема оборудования, но имеет возможность добавления функций, таких как динамическое назначение приоритетов с использованием, например, алгоритмов EDF или LLF, RM или DM для автоматического назначения фиксированных приоритетов во время выполнения задачи.

Будем считать, что фиксированный приоритет означает неизменность, приоритета каждой задачи во время выполнения. Т.е. если несколько задач с одним и тем же приоритетом готовы к выполнению, то задача, которая пришла первой, будет сохранять ее до тех пор, пока не будет вытеснена задачей с более высоким приоритетом или не будет заблокирована по каким-то причинам. В том случае, если две или более задачи с одним и тем же приоритетом становятся готовыми одновременно, будет выбрана задача с наименьшим идентификатором приоритета.

В этом разделе рассматриваются различные структуры общего планировщика, что позволяет выбрать наиболее подходящий для этого проекта.

1.4.1 Схема приоритетного выбора «полное дерево»

Планировщик «полное дерево», показанный на рисунке 7, является полностью параллельным алгоритмом планирования. Это потребует отдельного блока для управления состоянием каждой задачи и вывода необходимых сигналов, таких как «готовность» (Ready), «идентификатор задачи» (ID) и «приоритет» (Priority), в приоритетный блок планировщика. Для N задач дерево планировщика состоит из N+1 модулей сравнения и выбора. Этот метод является наиболее быстрым с точки зрения сравнения и выбора следующей задачи для выполнения. Время выполнения алгоритма планирования будет равно $O(s + \log_2(N))$, где s - количество тактовых циклов, необходимых управляющему автомату для выполнения полного обновления состояния задачи.

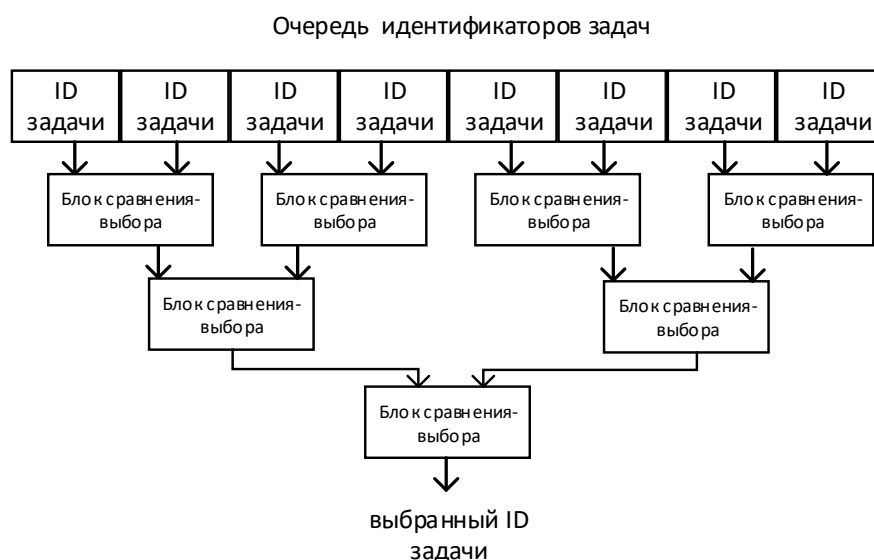


Рисунок 7 – схема приоритетного выбора «полное дерево»

Алгоритм «полное дерево» имеет одно большое ограничение. Для его реализации потребуется большой объём оборудования, даже если количество задач в очереди относительно мало. Причина этого заключается в том, что каждый модуль сравнения-выбора должен реализовать модуль сравнения как для приоритета, так и для идентификатора, которые с точки зрения затрат оборудования являются дорогостоящими логическими узлами. Быстрый синтез

с использованием 64 местной очереди и 4 битов для приоритета приводит к использованию ресурса для 2^6 необходимых модулей сравнения – выбора что составит более 5% логических блоков и триггеров кристалла. Масштабирование количества задач в очереди до 128 или даже 256 приводит 10% превышению только для реализации планировщика.

1.4.2 Схема последовательного приоритетного выбора

Концепция последовательного планировщика, показанная на рисунке 8, является намного менее производительной, но и менее затратной по использованию оборудования кристалла ПЛИС по сравнению с планировщиком «полное дерево».

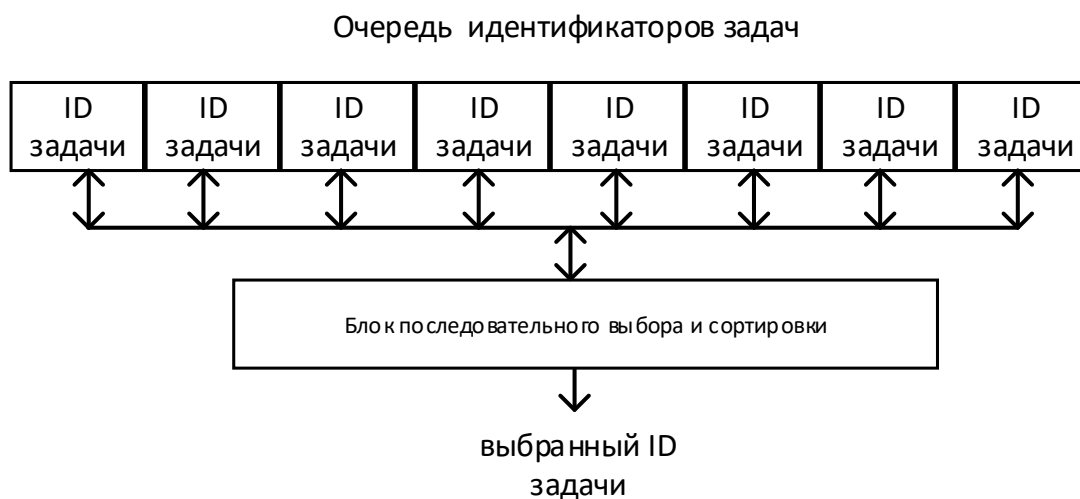


Рисунок 8 – схема последовательного приоритетного выбора

Последовательный планировщик состоит только из одного блока управления, который последовательно обновляет (сортирует) задачи в очереди. Такой планировщик не использует преимущества параллелизма, которые возможны при реализации планировщика в аппаратном обеспечении, что приводит к самой низкой производительности, поскольку все задачи в очереди должны быть обновлены, а среди готовых задач должна быть выбрана та, которая имеет самый высокий приоритет. Следовательно, планировщик не может работать быстрее, чем он может обслуживать все задачи в течение одного периода.

Единственным реальным положительным аспектом этой концепции является тот факт, что она использует наименьшие затраты оборудования, так как необходим только один автомат управления и задачи могут храниться в памяти RAM, что намного эффективнее, чем при использовании регистров для формирования очереди, как это необходимо для планировщика «полное дерево».

Время выполнения алгоритма планирования будет равно $O(S(N) + C)$, N – число задач в очереди, S – число тактовых циклов, необходимых блоку (автомату) управления для выполнения полного обновления состояния очереди и оценки того, имеет ли текущая задача более высокий приоритет, чем последняя. C – сумма количества тактовых циклов, необходимых блоку управления для подготовки к выбору задач перед обновлением очереди. Таким образом, время выполнения последовательного планировщика увеличивается линейно с количеством задач в очереди.

1.4.3 Схема гибридного приоритетного выбора

Планировщики «полное дерево» и последовательные планировщики являются двумя возможными крайностями по критерию «производительность-затраты оборудования». Концепция гибридного планировщика является комбинацией этих двух и объединяет в себе достоинства обоих и устраняет недостатки. Рисунок 9 иллюстрирует концепцию, которая основана на наличии двух или более последовательных блоков управления приоритетным выбором для сортировки (обновления) задач в очереди. После того как каждый из блоков выбрал задачу с наивысшим приоритетом в своем модуле задач, для выбора задачи с общим наивысшим приоритетом используется небольшое дерево «сравнения – выбора» модулей.

Эта структура медленнее, но, с другой стороны, использует меньше логики, чем концепция полного дерева. Она использует больше логики, чем последовательный планировщик, но также быстрее, если количество задач

каждого модуля управления задачами меньше, чем общее количество задач в очереди.

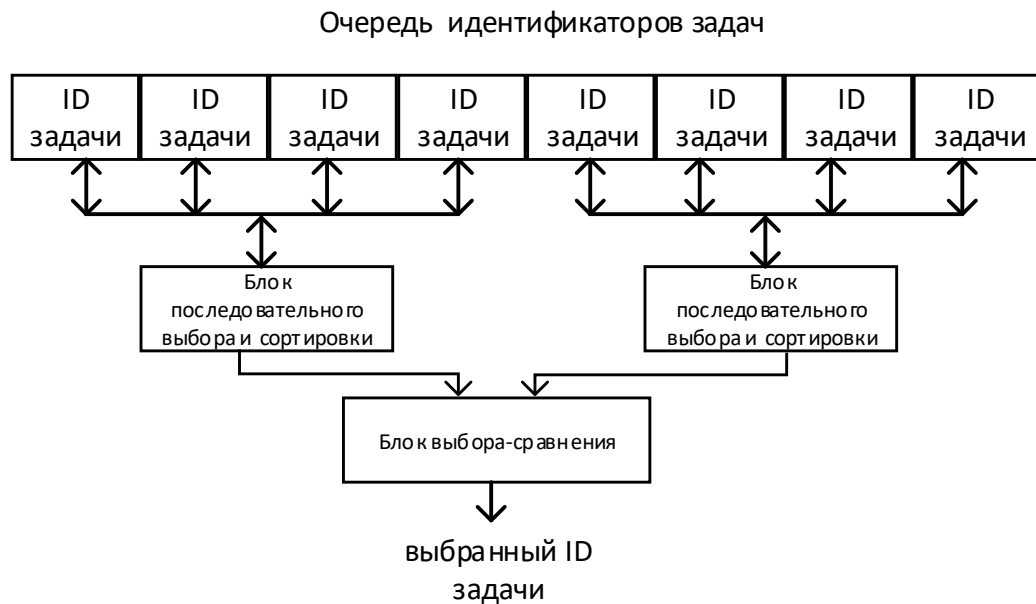


Рисунок 9 – схема гибридного приоритетного выбора

Потенциальным недостатком этого метода является тот факт, что небольшое количество задач в каждом модуле может привести к неэффективному использованию единственного блока памяти для хранения задач, поэтому потребуется распределенная память, состоящая из некоторого множества блоков памяти.

Время выполнения всего планировщика будет равно $O(S + \log_2(N))$, S - нумерация тактовых циклов, необходимых для сортировки (обновления) закрепленных за ним задач, это также минимальный период для таймера. N - количество модулей выбора-сравнения.

Поскольку в представляемой работе исследуется принципиальная возможность реализации приоритетного планировщика для систем реального времени, то в качестве критерия выбора был взят критерии «затраты оборудования» и «простота реализации». По этим критериям наибольшую эффективность показывает схема последовательного приоритетного выбора, который и был использован при реализации данного проекта.

1.5 Выводы

Изучены основные характеристики систем реального времени, проанализированы существующие алгоритмы приоритетного планирования и диспетчеризации задач. Произведен анализ схем аппаратного приоритетного выбора готовых к выполнению задач. Показано, что в многопроцессорных системах преимущественно применяют алгоритмы с временным и пространственным разделением задач. Выявлено, что существуют аппаратные реализации алгоритмов приоритетного планирования, которые выполняются с применением планировщиков задач с аппаратной поддержкой, что является основной целью диссертационной работы.

Основными задачами диссертационного исследования следует считать:

- провести формализацию алгоритмов диспетчеризации и разработку автоматных моделей диспетчеров задач с глобальной и локальными очередями задач на основе логики недетерминированных автоматов;
- на основе автоматных моделей разработать функциональные модели устройства диспетчеризации на языке проектирования аппаратуры VHDL и получить их временные характеристики.

2 Моделирование алгоритмов приоритетного планирования на языке недетерминированных автоматов

Вторая глава нацелена на предметное рассмотрение вопроса моделирования приоритетных планировщиков задач. В ней дается обзор распространенных на практике методов, алгоритмов и способов реализации важных частей планировщиков задач, их моделирование на НДА, и представление в виде математических моделей. Оцениваются пути их эффективного применения, с учетом преимуществ и недостатков. Взятые алгоритмы и методики являются отражением теоретических положений, изложенных в первой главе. На основе материалов данной главы сформируется основа для реализации диспетчеров задач с приоритетными очередями.

2.1 Возможности языка НДА для формального описания алгоритмов управления параллельными процессами и ресурсами

В данной работе предлагается методы описания и выбора приоритетного планировщика задач для встроенных многоядерных систем реального времени, основанные на языке недетерминированных автоматов (НДА), имитационных моделях массового обслуживания и языках функционально-логического моделирования, которые позволяют оценивать как характеристики производительности планировщиков задач, так и их функционирование, а также решать вопросы повышения производительности и надежности ядра многопроцессорных систем реального времени.

Язык недетерминированных автоматов используется для формального описания алгоритмов управления процессами и ресурсами в параллельных

вычислительных системах. Известно, что формальные методы должны обеспечивать комплексное решение задач спецификации, разработки, верификации, реализации и анализа сложных систем управления взаимодействующими параллельными процессами и ресурсами в вычислительных системах. Математический аппарат формальных методов описания управляющих автоматов (УА) должен обладать при этом высокой эффективностью, позволяющей решать повышения производительности и надежности систем управления [3].

Под высокой эффективностью математического аппарата формального языка описания УА понимается выполнение следующих основных требований:

- простота, наглядность, компактность и удобство использования для заказчика и разработчика;
- высокая степень выразительности при описании сложных вариантов алгоритмов, в том числе с отражением временного фактора, а также включающих как синхронные, так и асинхронные процессы с отсутствием тупиковых ситуаций, когда имеется зависимость реализуемых событий по данным и ресурсам;
- наличие методов оптимизационных равносильных преобразований, связанных с минимизацией, композицией, декомпозицией и верификацией УА на моделях и возможностью простой трансформации описания с одного языка на другой;
- простой переход от формального аналитического представления к структурной реализации УА, в том числе аппаратно (схемно или микропрограммно), программно или в виде их комбинаций;
- математический аппарат должен быть ориентирован на описание параллельно-последовательных алгоритмов;
- при простоте математического аппарата языка должно быть обеспечено описание широкого круга применений по управлению процессами и

объектами, начиная с торговых автоматов и кончая операционными системами с разделяемыми ресурсами.

Основное достоинство моделей недетерминированных автоматов заключается в скрытом в них параллелизме, который проявляется в том, что под действием одного и того же входного сигнала допускается переход от одного частного события к нескольким событиям, одновременное существование которых в соответствии с УА возможно. Таким образом, описание переходов в УА при использовании моделей НДА выполняется не в терминах состояний детерминированных автоматов, а в терминах частных событий НДА, что и обеспечивает такую же простоту, компактность и удобство описания УА в параллельных системах.

Формальный язык представления УА на основе логики НДА можно рассматривать и как вариант базового языка, который включает две составляющие: графическую форму для наглядного представления алгоритмов в виде граф-схем с параллельными ветвями (ГСАП) и аналитическую в виде системы рекуррентных канонических уравнений (СКУ), описывающих все реализуемые в алгоритме частные события. При этом под частными событиями, в зависимости от уровня детализации УА, понимается выполнение некоторых актов обработки информации, к которым могут относиться выполнение микроопераций, операций, подпрограмм, аппаратных сигналов, функциональных блоков и т.д.

Для формального аналитического представления сложных УА может быть использована модель конечного цифрового НДА Мура в виде стандартной системы рекуррентных предикатных формул вида (1), описывающих все реализуемые в алгоритме частные события S_j [3]:

$$S_j^{Yj}(t+1) = \bigvee_{i,j} X_{i,j}(t) \& S_i(t) \vee \bigvee_j X_{j,j}(t) \& S_j(t), \quad (1)$$

$$j = \overline{0, m}, S_0(0) = 1,$$

где $X_{i,j}$ – частный входной сигнал, представляющий собой сочетание (конъюнкцию) элементарных двоичных входных сигналов из структурного алфавита $[X]$, взятых с отрицанием или без отрицания, под действием которых осуществляется переход от события S_i к событию S_j ;

$S_i(t)$ – сокращенное обозначение выражения, описывающего частное событие S_i , непосредственно предшествующее частному событию S_j ; (каждая предикатная формула $S_i(t)$ должна быть либо тождественно истинной предикатной формулой, либо удовлетворять условию $S_i(t) \neq S_j(t)$ и определяться также выражением вида (1));

Y_j – частный выходной сигнал, представляющий собой сочетание элементарных двоичных выходных сигналов из структурного алфавита $[Y]$, отмечающих событие S_j .

В системе канонических уравнений (1) первая часть каждого уравнения формализует описание условий первоначального появления или зарождения события S_j , а вторая – формализует описание условий возобновления или сохранения события S_j .

Таким образом, простая каноническая форма языка НД СКУ определяет только причинно-следственные связи между событиями S_i и S_j , т.е. связь по управлению и информационную связь. Для такой формы представления событий предполагается, что событие S_j может выполняться сразу же после события S_i (связь по управлению условно или безусловно), когда это событие выработало информацию (данные) для реализации события S_j , т.е. эти события связаны информационно (внутренняя информационная связь). Учитывая отмеченные выше логические возможности простой канонической формы представления частных событий (1), следует отметить, что она характерна в основном только для синхронного режима работы.

Одним из достоинств языка НДА является отражение временного фактора в управляющих алгоритмах, определяющих поведение системы управления во времени. Одной из особенностей систем логического управления является асинхронность режима работы, которая может быть реализована за счёт расширения входного алфавита $[X]$ путем ввода двоичных входных сигналов, которые сигнализируют о готовности входной информации (исходных данных) для ее обработки и готовности блоков системы для приема результатов обработки, а также сигналов о завершении обработки исходных данных [3].

Для асинхронного режима работы характерно использование ждущих и временных логических условий. В этом случае переход от события S_i к событию S_j в зависимости от значения ждущего или временного логического условия осуществляется путем ввода дополнительного пустого события S_r , которое имитирует ожидание появления истинности некоторого логического условия S_p , чтобы осуществить переход к событию S_j . В данном случае под логическим условием S_p можно понимать или выполнение некоторого временного события, например, $t_n \leq T \leq t_b$, или окончание каких-либо действий, которые вызывают переход к событию S_j . Тогда описываемые события будут иметь вид:

$$S_j(t+1) = S_r S_p \quad (2)$$

$$S_r(t+1) = (S_i \vee S_r) \overline{S_p}$$

Анализируя выразительные возможности структуры модели НД СКУ (2) для описания УА, определяющих поведение системы во времени, когда порядок выполнения событий описывается без привлечения времени в явном виде, можно утверждать, что язык НД СКУ является по существу одной из разновидностей языка временной темпоральной логики типа CTL [3].

Учитывая отмеченные выразительные возможности языка НД СКУ, свойственные языку временной темпоральной логики, этот язык может быть успешно использован для верификации управляющих алгоритмов. Для этого исследуемые свойства управляющего алгоритма описываются в виде формальной модели на языке логики НД СКУ и выполняют ее проверку путём моделирования этой модели, преобразованной в программный код, например, на языке VHDL [5].

Выше было показано, что недетерминированные автоматы с успехом моделируют параллельные процессы. Параллельным процессам присуще свойство конфликтности, которое заключается в том, что при доступе к общему ресурсу возникает борьба между процессами. Такие процессы называются несовместимыми. Если для двух несовместимых событий S_i и S_j , принадлежащих двум параллельным ветвям алгоритма управления, и если не допускается их одновременного зарождения, то описанию таких событий будут соответствовать уравнения

$$\begin{aligned} S_i(t+1) &= S_{i,3}\bar{S}_j \vee S_i, S_{i,c}, \\ S_j(t+1) &= S_{j,3}\bar{S}_i \vee S_j, S_{j,c}, \end{aligned} \quad (3)$$

где $S_{j,3}$ - сокращенное обозначение описания условия зарождения события S_j , $S_{j,c}$ - сокращенное обозначение описания условий сохранения события S_j . Если для двух несовместимых событий S_i и S_j , принадлежащих двум параллельным ветвям алгоритма управления, условия их зарождения могут появиться в один и тот же момент времени, то описанию таких событий будут соответствовать уравнения (4), но с учетом относительного приоритета одного события над другим. Тогда описание таких событий, предотвращающее конфликтную ситуацию, при условии более высокого приоритета, например, события S_i над событием S_j , будет иметь вид:

$$S_i(t+1) = S_{i,3} \bar{S}_j \vee S_i S_{i,c} \quad (4)$$

$$S_j(t+1) = S_{j,3} \bar{S}_i S_{npi} \vee S_j S_{j,c},$$

где введение в первую часть второго уравнения события S_{npi} , учитывающего условие приоритетности события S_i над событием S_j , обеспечивает отсутствие конфликтной ситуации.

Предлагаемая обобщённая каноническая форма представления алгоритма УА в виде НД SKU (2) позволяет естественно и просто переводить описание алгоритма на языки программирования, например, на язык VHDL, когда требуется выполнить моделирование алгоритма для его верификации или аппаратную реализацию, например, на ПЛИС [5]. Действительно, при представлении исходного УА в виде программы на языке VHDL, правые части всех уравнений исходной SKU НДА преобразуются в коды, которые имеют такой же вид, что и исходные уравнения, но только с другими обозначениями переменных и операций над ними, которые указываются в начале программы. Например, для события типа S_k^i входа процесса в критический интервал имеем [4]:

$$S_k^i(t+1) = S_{вп}^i S_{вз}^i S_{пр}^i \vee S_k^i \bar{S}_p^i, \quad (5)$$

$$sk \leq (svp \text{ and } svz \text{ and } spr) \text{ or } (sk \text{ and } (\text{not } sp)).$$

В то же время к предлагаемой стандартной форме представления УА в виде НД SKU может быть преобразовано описание управляющих алгоритмов, представленных на широко известных начальных языках, к числу которых относится язык операторных схем алгоритмов с параллельными ветвями (ГСАП) [4].

2.2 Формализация алгоритмов управления параллельными процессами на основе использования механизма «рандеву»

В данном разделе рассматривается методика формальной спецификации алгоритма управления межпроцессорного взаимодействия в классической задаче «о спящем парикмахере», которая является примером классических задач синхронизации процессов. Эта задача иллюстрирует отношение типа «клиент-сервер», которое имеет место между процессами в многопроцессорных вычислительных системах, где для нее используется особый тип синхронизации, называемый «рандеву».

2.1.1 Аналитическая модель алгоритма управления процессами в задаче «о спящем парикмахере» на основе использования СНДА

Рассматривая посетителей и парикмахера как взаимодействующие процессы, для которых посетитель – это клиент, запрашивающий сервис у парикмахера, а парикмахер – это сервер, обеспечивающий данный сервис, можно представить данный тип взаимодействия как пример отношений «клиент-сервер».

Действия, когда парикмахер дожидается прихода посетителя, а посетитель – освобождения парикмахера, необходимо синхронизировать. Эту встречу посетителя и парикмахера называют рандеву. При этом посетителю необходимо ждать, пока парикмахер закончит его стричь, после чего он освобождает кресло парикмахера и ждет открытия двери парикмахером. Парикмахер, в свою очередь, перед тем как закрыть дверь, должен подождать, пока уйдет посетитель. Таким образом, парикмахер и посетитель проходят через последовательность синхронизированных этапов, начинающихся с рандеву [3].

Для формального описания алгоритма взаимодействия процессов в данной задаче будем использовать язык систем канонических уравнений. Для этой цели введем следующие основные частные события, реализуемые в данном алгоритме:

S_3^k - событие, определяющее приход нового клиента в парикмахерскую;

$S_{оч}^k$ - событие, определяющее наличие места в очереди к парикмахеру;

S_0^k и $S_{зд}^n$ - события, определяющие нахождение клиента и парикмахера в салоне парикмахерской, соответственно;

$S_{бп}^k$ и $S_{бп}^n$ - события, свидетельствующие о том, что клиент будит парикмахера, а парикмахер будит клиента, соответственно;

$S_{кп}^k$ и $S_{ок}^n$ - события, свидетельствующие о том, что клиент сел в кресло парикмахера, а парикмахер освободил свое кресло, соответственно;

S_c^k и S_c^n - события, свидетельствующие о том, что клиент и парикмахер спят, соответственно;

$S_{гт}^k$ и $S_{гт}^n$ - события, свидетельствующие о том, что клиент и парикмахер готовы к обслуживанию, соответственно;

S_n^k и S_k^n - события, свидетельствующие о начале и окончании стрижки, соответственно;

$S_{ок}^k$ и $S_{од}^n$ - события, свидетельствующие о том, что клиент освободил кресло парикмахера, а парикмахер открыл выходную дверь, соответственно;

S_y^k и $S_{зд}^n$ - события, свидетельствующие о том, что клиент ушел из парикмахерской, а парикмахер закрыл выходную дверь, соответственно.

На основании словесно представленного алгоритма управления процессами в задаче о спящем парикмахере и введенных событий, реализуемых в этом алгоритме, система канонических уравнений, описывающих эти события, и соответствующий им граф недетерминированного автомата (НДА) будут иметь следующий вид:

- для процесса – клиент:

$$\begin{aligned}
S_0^k(t+1) &= S_3^k S_{оч}^k \\
S_c^k(t+1) &= S_0^k \overline{S_c^n} \vee S_c^k \overline{S_{ок}^n} \\
S_{оп}^k(t+1) &= S_0^k S_c^n \\
S_{кп}^k(t+1) &= S_c^k S_{ок}^k \vee S_{оп}^k S_{ок}^n \\
S_{гт}^k(t+1) &= S_{кп}^k S_{зд}^n \vee S_{гт}^k S_{гт}^n
\end{aligned} \tag{6}$$

- для процесса – сервер (парикмахер):

$$\begin{aligned}
S_c^n(t+1) &= S_0^n \overline{S_0^k} \vee S_0^n \overline{S_{оп}^k} \\
S_{ок}^n(t+1) &= S_0^n S_0^k S_c^k \\
S_{ок}^n(t+1) &= (S_0^n S_0^k \overline{S_0^k} \vee S_c^n) S_{оп}^k \\
S_{гт}^n(t+1) &= (S_{ок}^n \vee S_{ок}^n) S_{кп}^k \vee S_{гт}^n \overline{S_{гт}^k}
\end{aligned} \tag{7}$$

Система канонических уравнений, описывающая события после randevu:

$$\begin{aligned}
S_h(t+1) &= (S_{гт}^k S_{гт}^n) \\
S_{од}^k(t+1) &= S_k \\
S_k(t+1) &= S_h \\
S_y^k(t+1) &= S_{ок}^k S_{од}^k \\
S_{ок}^k(t+1) &= S_k \\
S_{PL}^n(t+1) &= S_{од}^m S_y^k
\end{aligned} \tag{8}$$

В приведенных выше системах канонических уравнений и графе НДА были опущены события, соответствующие пустым событиям обратной связи в циклах из логических условий. Эта связь символизирует ожидание некоторого

события S_i истинности логического условия, при котором реализуется переход к событию S_j , являющемуся первым преемником события S_j . Например, рассмотрим переход от события $S_{ок}^k$ к событию S_y^k , для которого выход условной вершины $S_{од}^n$ соединен с ее входом.

Граф недетерминированного автомата алгоритма управления взаимодействующими процессами в задаче «спящий парикмахер» имеет следующий вид:

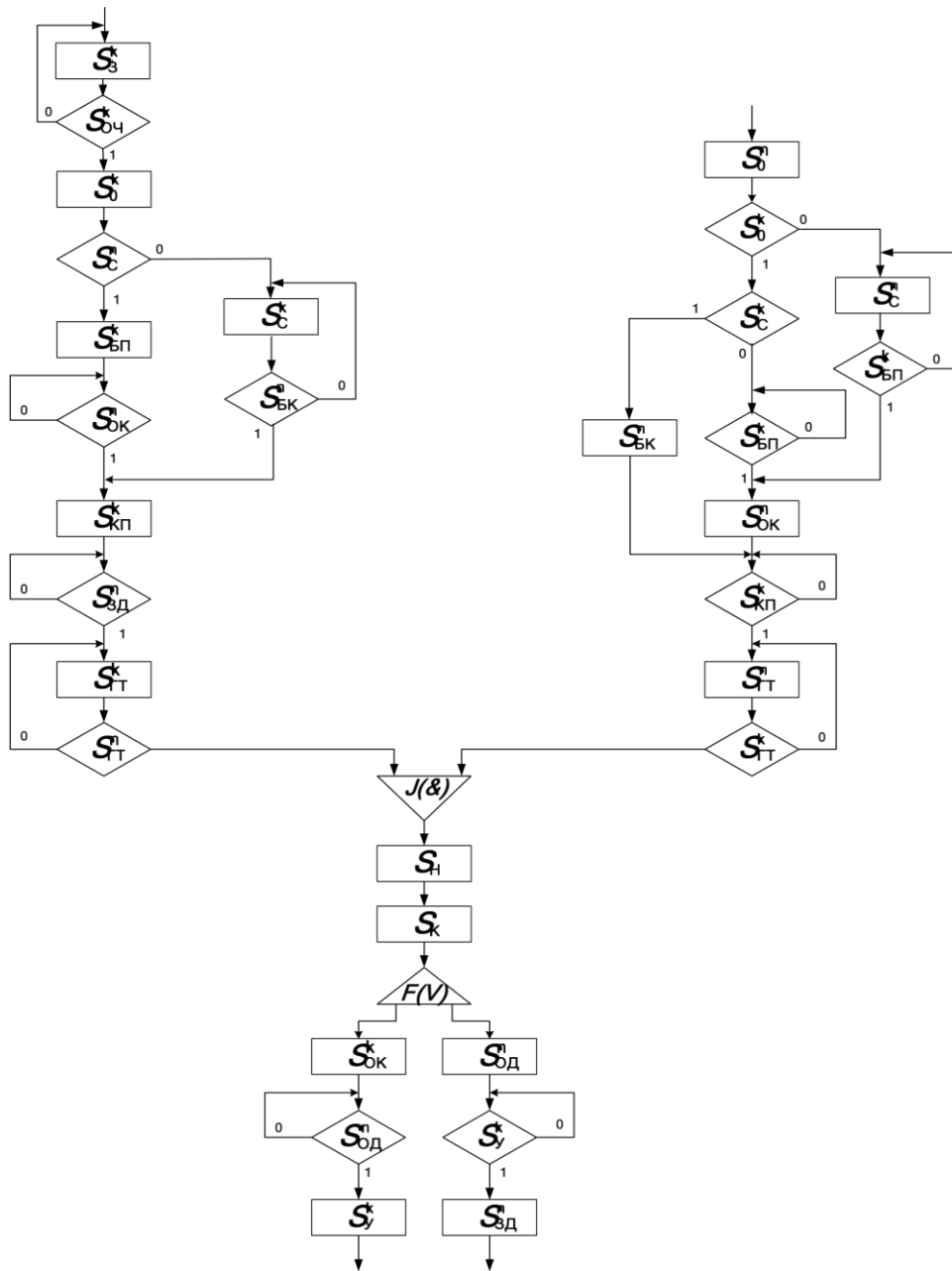


Рисунок 10 - Граф недетерминированного автомата алгоритма управления взаимодействующими процессами в задаче «спящий парикмахер»

Введя пустую операторную вершину в обратную связь и обозначив ее символом S_e , получим следующие уравнения:

$$S_e(t+1) = (S_{ок}^k \vee S_c) S_{од}^n \quad (9)$$

$$S_y^k(t+1) = (S_{ок}^k \vee S_c) S_{од}^n$$

Данный алгоритм и системы канонических уравнений будут положены в основу построения планировщиков задач с глобальной и локальными

очередями, которые основаны на механизме синхронизации «рандеву». Однако этот алгоритм характерен для вычислительных систем с одним процессором. Для многопроцессорной системы этот алгоритм расширяется в алгоритм работы парикмахерской, где аналоги процессоров - это коллектив парикмахеров, а аналоги задач – клиенты.

2.3 Формализация алгоритмов управления параллельными процессами с использованием механизма монитора и согласующего кольцевого буфера

В многопроцессорных системах обмен данными между параллельными ветвями алгоритма приводит к необходимости использовать межпроцессорные взаимодействия, проявляющиеся в обмене данными между несколькими процессами (потоками). Для такого обмена существуют различные механизмы: именованные и неименованные каналы, разделяемая память. Данные механизмы, реализованные программно, негативно сказываются на латентности, пропускной способности и загруженности системы, поэтому последнее время намечается тенденция к аппаратной реализации данных функций.

Любой процесс обмена происходит с использованием буфера некоторого объема, причем буфером может быть либо разделяемая область в оперативной памяти, либо объект операционной системы. В каждом случае процесс-передатчик формирует данные и записывает их в буфер, т.е. является производителем, процесс-приемник читает их из него, т.е. является потребителем [3].

Будем считать, что используется многопоточная модель программирования, реализующая существование потока-передатчика, записывающего в кольцевой буфер и потока-приемника, осуществляющего чтение из кольцевого буфера. Данная постановка усложняет процедуру

управления процессами и требует дополнительных средств ее реализации, однако является наиболее общей.

Для решения задач управления взаимодействующими параллельными процессами используют, как было отмечено ранее, различные механизмы синхронизации процессов, в том числе и механизм монитора, который был впервые предложен в 1974 г. Хоаром и Бринчем Хансеном для организации синхронизации процессов при обращении к общим критическим ресурсам.

Применение такого механизма синхронизации процессов по сравнению с механизмами критических интервалов с использованием семафоров позволяет существенно повысить надежность и компактность программного обеспечения реализации информационных систем.

В данном разделе рассматривается методика формального описания алгоритма управления взаимодействующими параллельными процессами при обмене сообщениями с использованием механизма мониторов и кольцевого буфера в качестве общего критического ресурса. В связи с этим рассмотрим вначале методику функционирования кольцевого согласующего буфера при обмене сообщениями между производителем и потребителем, позволяющую определить условия, для которых буфер будет полон, пуст или занимает промежуточное состояние.

2.3.1 Функционирование кольцевого согласующего буфера при обмене сообщениями между производителем и потребителем

Производитель помещает сообщения в разделяемый кольцевой буфер, потребитель извлекает их оттуда. Буфер содержит очередь уже помещенных, но еще не извлеченных сообщений. Эта очередь может быть представлена связным списком или массивом [3].

Для представления очереди сообщений используются две целочисленные переменные, которые указывают, соответственно, на первую заполненную и первую пустую ячейку буфера. Эти переменные организуются с помощью

счетчиков буфера записи СчБз и счетчика буфера чтения СчБчт. Показания счетчика СчБз будет определять номер первой незаполненной (пустой) ячейки, а счетчика СчБчт – номер первой заполненной ячейки. Для этой цели при инициализации системы управления в указанные счетчики записываются единицы, т.е. при $t = 0$ / $\text{СчБз} = 1$; $\text{СчБчт} = 1$.

Если принять количество ячеек в кольцевом буфере, например, за $n = 8$, то счетчики буфера будут работать по модулю 8. Тогда при переполнении счетчиков может быть образован сигнал переполнения, который может быть использован для определения состояний буфера, для которых буфер будет или полон, или пуст.

Для варианта кольцевого буфера событие, когда буфер будет полон ($S_{\text{бп}} = 1$), будет иметь место при условиях равенства значений счетчиков буфера ($\text{СчБз} = \text{СчБчт} > 1$) и был перенос из счетчика буфера записи ($\text{СчБз} = 0$).

Событие, когда буфер будет пуст ($S_{\text{бo}} = 1$), будет иметь место также при условии равенства значений счетчиков буфера ($\text{СчБз} = \text{СчБчт}$) и не было сигнала переноса из СчБчт или при условии, когда в процессе работы счетчики буфера примут исходное состояние ($\text{СчБз} = \text{СчБчт} = 1$) и был перенос из СчБчт.

Рассмотрим некоторые варианты записи и чтения в кольцевой буфер.

Представим буфер массивом, в котором имеется $n = 8$ ячеек (рисунок 11), а в каждую ячейку записывается одно сообщение.

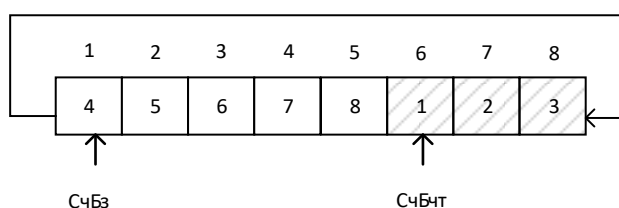


Рисунок 11 – произвольное положение массива в буфере

Будем считать, что номер ячейки буфера совпадает с номером сообщения. Тогда верхний ряд цифр (рисунок 11) указывает на исходное состояние буфера перед началом работы системы, когда все ячейки пусты.

Кольцевой буфер при записи информации работает следующим образом. Такт записи в одну ячейку буфера состоит из двух частей: в первой части осуществляется непосредственная запись в ячейку, номер которой указывает счетчик СчБз, во второй части производится кольцевой сдвиг содержимого буфера влево. На рисунке 11 показаны результаты записи трех сообщений, когда положение счетчика СчБз стало равным 4, так как в исходном состоянии в СчБз была записана 1. Таким образом, ячейка буфера под номером 4 будет первой пустой ячейкой после сообщений в конце очереди.

2.3.2 Формализация основных событий, реализующих алгоритм управления параллельными процессами в решаемой задаче «производитель-потребитель» при использовании механизма монитора и языка СНДА

Описание таких событий (S_M^i) будет состоять из двух частей: первая составляющая определяет зарождение события, определяющего начальный вход i -го процесса в монитор ($S_{M,3}^i$), а вторая – его сохранение в мониторе ($S_{M,c}^i$), т.е. имеем

$$S_M^i(t+1) = S_{M,3}^i \vee S_{M,c}^i \quad (10)$$

$$S_{M,3}^i = S_{вп}^i S_{вз}^i S_{пр}^i$$

где $S_{вп}^i$ – событие, определяющее прием заявки i -го процесса на обслуживание; $S_{вз}^i$ – событие, обеспечивающее взаимоисключение событий S_M^i ; $S_{пр}^i$ – событие, определяющее приоритет i -го процесса [3].

$$S_{M,c}^i = S_M^i \overline{S_n^i} \overline{S_{кр,3}^i} \quad (11)$$

где S_n^i – событие, свидетельствующее об окончании одноразовой операции i -го процесса с общим критическим ресурсом; $S_{кр,3}^i$ – событие, свидетельствующее о том, что критический ресурс для i -го процесса занят (буфер полон - $S_{бп}=1$) или (буфер пуст - $S_{бo}=1$).

Рассмотрим формальное описание отдельных составляющих в уравнении (10).

Событие $S_{вп}^i$ обеспечивает реализацию третьего требования к механизму монитора, основанного на том, что заявка i -го процесса на обслуживание воспринимается и сохраняется только в том случае, когда данный процесс не находится в мониторе.

Тогда это событие формализуется следующим образом:

$$S_{вп}^i(t+1) = (S_3^i \vee S_{вп}^i) \overline{S_M^i} \overline{S_{мп}^i} \quad (12)$$

где S_3^i – заявка i -го процесса; $S_{мп}^i$ – событие, определяющее повторный вход i -го процесса в монитор, вышедшего из монитора не обслуженным, т.е. это событие обеспечивает четвертое требование к механизму монитора и будет иметь следующий вид:

$$S_{мп}^i(t+1) = S_{мн}^i S_{кр,0}^i \vee S_{мп}^i S_n^i \quad (13)$$

где $S_{мн}^i$ – событие, свидетельствующее о выходе i -го процесса из монитора не обслуженным; $S_{кр,0}^i$ – событие, свидетельствующее об освобождении критического ресурса для операции с ним i -го процесса после выполнения предыдущей операции.

Событие $S_{вз}^i$ является основным событием, определяющим вход в монитор в одно и то же время только одного i -го процесса, и определяется взаимоисключением событий входа процессов в монитор на основе их несовместимости с i -м процессом. Это событие представляется следующей формулой:

$$S_{вз}^i = \wedge (\overline{S_M^\alpha} \vee \overline{S_{мп}^\alpha}) \quad (14)$$

$$\vee \alpha [\alpha \neq i]$$

Событие $S_{пр}^i$ обеспечивает однозначность входа процессов в монитор, используя их приоритетность, которая определится следующей формулой:

$$S_{пр}^i = S_{пр}^i(0) \vee S_{вп}^i S_{кр,0}^i \overline{S_{мн}^i} \quad (15)$$

где $S_{\text{пр}}^i(0)$ – событие, определяющее приоритет i -го процесса в начальный период. Введение этого события объясняется тем обстоятельством, что в начальный период обращения процессов к общему ресурсу могут быть истинны все события типа $S_{\text{вз}}^i$ и $S_{\text{вп}}^i$.

В дальнейшем при описании всех событий алгоритма управления при обращении к критическому ресурсу будем базироваться в качестве примера на вариант решения задачи взаимодействия для двух процессов: один производитель и один потребитель, заявки которых будут фиксироваться в двух очередях – одна начальная очередь, а другая очередь ждущих заявок процессов, вышедших из монитора не обслуженными. Для этого варианта полная система уравнений, определяющая доступ процессов к критическому ресурсу и выполнение операций над ними, будет иметь следующий вид:

$$\begin{aligned}
S_{\text{м}}^1(t+1) &= S_{\text{вп}}^1 S_{\text{вз}}^1 S_{\text{пр}}^1 \vee S_{\text{м}}^1 (S_{\text{н}}^1 \vee S_{\text{кр,з}}^1) \\
S_{\text{м}}^2(t+1) &= S_{\text{вп}}^2 S_{\text{вз}}^2 S_{\text{пр}}^2 \vee S_{\text{м}}^2 (\overline{S_{\text{н}}^2} \vee \overline{S_{\text{кр,з}}^2}) \\
S_{\text{мп}}^1(t+1) &= S_{\text{мн}}^1 S_{\text{кр,0}}^1 \vee S_{\text{мп}}^1 \overline{S_{\text{н}}^1} \\
S_{\text{мп}}^2(t+1) &= S_{\text{мн}}^2 S_{\text{кр,0}}^2 \vee S_{\text{мп}}^2 \overline{S_{\text{н}}^2} \\
S_{\text{вп}}^1(t+1) &= (S_{\text{з}}^1 \vee S_{\text{вп}}^1) (\overline{S_{\text{м}}^1} \vee \overline{S_{\text{мп}}^1}) \\
S_{\text{вп}}^2(t+1) &= (S_{\text{з}}^2 \vee S_{\text{вп}}^2) (\overline{S_{\text{м}}^2} \vee \overline{S_{\text{мп}}^2}) \\
S_{\text{пр}}^1 &= S_{\text{пр}}^1(0) \vee S_{\text{вп}}^1 S_{\text{кр,0}}^1 \overline{S_{\text{мн}}^1} \\
S_{\text{пр}}^2 &= S_{\text{пр}}^2(0) \vee S_{\text{вп}}^2 S_{\text{кр,0}}^2 \overline{S_{\text{мн}}^2} \\
S_{\text{пр}}^1(0) &= S_0^1 x_n S_{\text{вп}}^1 \\
S_{\text{кр,з}}^1 &= S_{\text{бп}} \\
S_{\text{кр,з}}^2 &= S_{\text{бо}} \\
S_{\text{мн}}^1(t+1) &= S_{\text{м}}^1 S_{\text{кр,з}}^1 \vee S_{\text{мн}}^1 \overline{S_{\text{кр,0}}^1} \\
S_{\text{мн}}^2(t+1) &= S_{\text{м}}^2 S_{\text{кр,з}}^2 \vee S_{\text{мн}}^2 \overline{S_{\text{кр,0}}^2}
\end{aligned} \tag{16}$$

$$S_{кр,0}^1(t+1) = S_n^2 \vee S_{кр,0}^1 \overline{(S_{мп}^1 \vee S_{м}^1)}$$

$$S_{кр,0}^2(t+1) = S_n^1 \vee S_{кр,0}^2 \overline{(S_{мп}^2 \vee S_{м}^2)}$$

$$S_{рз}^1(t+1) = (S_{м}^1 \vee S_{мп}^1) S_{кр,с}^1 \vee S_{рз}^1 \overline{(S_{бп} \vee S_{кз})}$$

$$S_{рч}^2(t+1) = (S_{м}^2 \vee S_{мп}^2) S_{кр,с}^2 \vee S_{рз}^2 \overline{(S_{бп} \vee S_{кз})}$$

$$S_{кр,с}^1 = \overline{S_{бп}} \vee S_{бо}$$

$$S_{кр,с}^2 = S_{бп} \vee \overline{S_{бо}}$$

$$S_n^1(t+1) = S_{рз}^1(S_{кз} \vee S_{бп})$$

$$S_n^2(t+1) = S_{рз}^2(S_{кч} \vee S_{бо})$$

где $S_{рз}^1$ и $S_{рч}^2$ – события, определяющие выполнение операции записи и чтения, соответственно; $S_{кр,с}^1$ и $S_{кр,с}^2$ – комбинационные события, определяющие возможность выполнения операции записи и чтения, соответственно; $S_{кз}$ и $S_{кч}$ – события, свидетельствующие об окончании операции записи и чтения порций информации, определяемых счетчиками $S_{ч*з}$ и $S_{ч*чт}$, соответственно.

Учитывая представленное аналитическое описание всех основных частных событий (уравнения (16)), реализующих управляющий алгоритм взаимодействия процессов в решаемой задаче, можно для наглядности представить рассматриваемый алгоритм в виде графа СНДА (рисунок 12, 13)

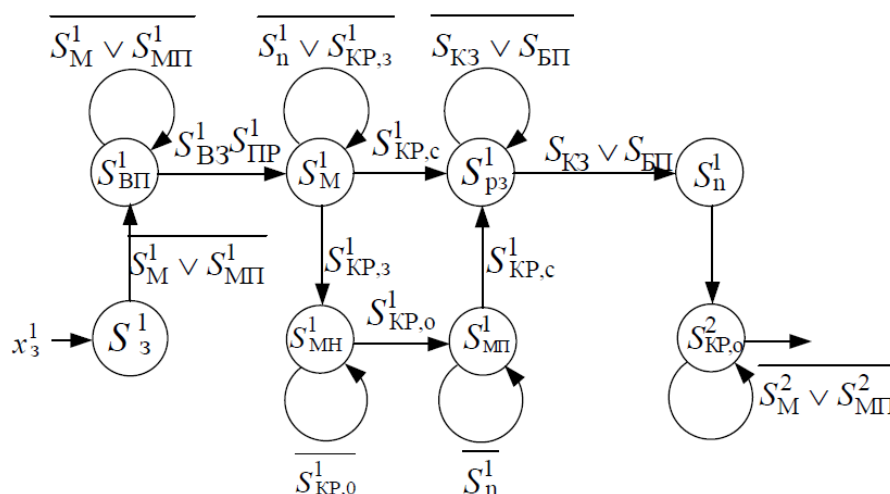


Рисунок 12 - граф СНДА алгоритмов управления процесса производителя при обращении его к общему ресурсу (буферу сообщений) с использованием механизма монитора

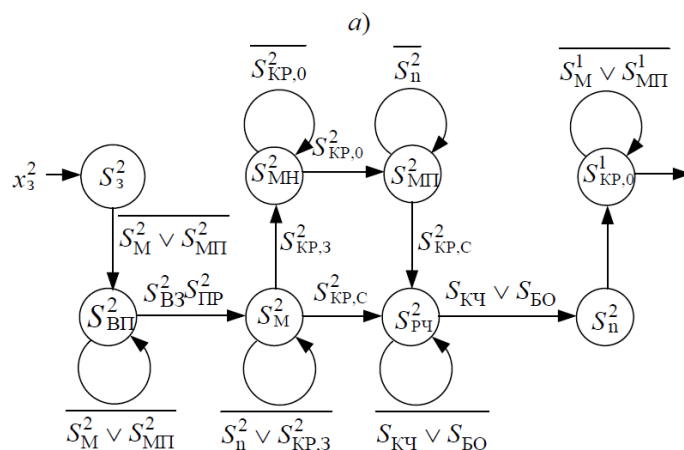


Рисунок 13 - граф СНДА алгоритмов управления процесса потребителя при обращении его к общему ресурсу (буферу сообщений) с использованием механизма монитора

Изученные СКУ для задачи «производители - потребители» будут использованы в дальнейшем при построении СКУ для процесса «клиент», реализуемого приоритетной очередью задач до момента рандеву для планировщиков, проектируемых в 3й главе.

2.4 Выводы

1. Изучены возможности языка НДА для формального описания алгоритмов управления параллельными процессами и ресурсами.

2. Рассмотрены НДА-модели в виде системы канонических уравнений и графа автомата, реализующего алгоритмы управления диспетчеризацией с временным и пространственным разделением задач в многопроцессорных системах, такие как механизм «рандеву», концептуальная модель «задачи о спящем парикмахере», алгоритмы управления параллельными процессами в задаче «читатели-писатели» и с использованием механизма монитора и согласующего кольцевого буфера
3. Полученные канонические уравнения, описывающие поведение автомата, будут трансформированы на язык проектирования аппаратуры и использованы для описания планировщиков на языке VHDL.

3 Аппаратная реализация алгоритмов приоритетного планирования

3.1 Проблемы управления ресурсами и процессами в многопроцессорных системах

Значительная часть функций межпроцессорного взаимодействия, такие как планирование и диспетчеризация процессов, управление памятью, обработка прерываний в современных операционных системах реализуется программным путем. Высокая частота использования этих функций приводит к большим непроизводительным временным затратам, достигающим нескольких десятков процентов общего времени выполнения процессов.

Уже достаточно давно наметилась тенденция к переходу от программной реализации алгоритмов ОС к аппаратной, которая вызвана стремлением к увеличению отношения производительность/стоимость ЭВМ и обусловлена высоким развитием микроэлектроники. Действительно, аппаратная реализация алгоритмов ОС позволяет уменьшить время его выполнения и, следовательно, повысить производительность ЭВМ. С другой стороны, стоимость дополнительного оборудования для аппаратной реализации, благодаря прогрессу элементной базы, с каждым годом падает. Кроме того, аппаратная реализация алгоритмов ОС позволяет в определенной степени уменьшить трудности создания программного обеспечения, а также улучшить совместимость различных версий ОС для компьютеров разных платформ. Кроме того, в значительной степени устраняется трудоемкая операция отладки нижних уровней ОС и повышается корректность всего программного обеспечения, благодаря лучшей формализации функций, реализованных аппаратным способом по сравнению с программным способом.

Издержки операционной системы, реализованной программно, наиболее сильно оказывают влияние на функционирование систем реального времени, так как именно системы реального времени требуют наиболее быстрой реакции на входное воздействие. В связи с этим возникает необходимость увеличить производительность ОС именно в системах жесткого реального времени.

Традиционное управление процессами и ресурсами, осуществляемое ядром операционной системы (ОС) на основе системного программного обеспечения, приводит к значительным замедлениям выполнения приложений, и, следовательно, к существенным потерям производительности вычислительной системы. Эта проблема вызвана наличием трудоемких компонент ядра ОС, к которым относятся: диспетчеры задач, связанные с переключением процессов; планировщики задач, связанные с выделением и освобождением ресурсов; средства взаимодействия процессов, такие как каналы передачи сообщений; примитивы синхронизации и другие API-функции.

В однопроцессорных операционных системах межпроцессорные коммуникации (IPC) ориентированы на локальные взаимодействия, т.е. между адресными пространствами на той же самой машине. Средства IPC традиционно реализуются в ядре, однако у такого подхода существуют проблемы. Приложения, которые используют быстровыполняющиеся потоки имеют ограниченную эффективность из-за высокой трудоемкости вызова функций ядра и переключения контекста процессора. На мультипроцессорах с разделяемой памятью проблема решается перемещением функций ядра в пользовательскую область, однако при этом может страдать безопасность и надежность системы из-за средств тестирования, которые ограничены в своих возможностях.

Эффективность функций планирования и диспетчеризации во многом обусловлена скоростью переключения процессов. Для многопроцессорных систем характерен более частый вызов планировщика и диспетчера.

Следовательно, планировщик и диспетчер в многопроцессорных системах должны работать быстрее, для того чтобы не создавать задержек. Однако имеет место быть обратная ситуация, когда накладные расходы возрастают с увеличением процессоров в системе из-за возникающих конфликтов за доступ к диспетчеру задач, в результате чего появляются очереди запрашивающих процессоров. Устранение конфликтов решается с помощью дополнительной синхронизации работы процессоров. Задача планирования и диспетчеризации в многопроцессорных системах, в отличие от однопроцессорных, является двумерной, так как помимо функции назначения задачи необходимо выполнять функцию выбора текущего процессора. Кроме того, в планировщиках с общей очередью существует явление перезагрузки кэш-памяти процессоров, когда задача, обрабатываемая на одном процессоре, в следующий раз выполняется на другом. Эта ситуация называется кэш-промахом и негативно сказывается на производительности системы.

Анализ зарубежных и отечественных публикаций за последние несколько лет показывает, что главные недостатки программного обеспечения, на котором базировались традиционные операционные системы, в т.ч. реального времени, могут быть устранены путем реализации функций ядра аппаратными средствами. Считается, что основной недостаток аппаратно-ориентированных операционных систем заключается в низкой гибкости, под которой понимается отсутствие возможности настройки функций ядра на изменяющиеся условия обслуживания разнохарактерных пользовательских запросов (высокоприоритетные, низкоприоритетные, бесприоритетные, высокоскоростные, низкоскоростные, фоновые и др.). При выполнении таких приложений применяют разные критерии, например, минимизация времени выполнения работ, максимизация реакции системы и др., что требует настройки (реконфигурации) ОС на оптимальный метод планирования (временное

разделение, пространственное разделение, круговой выбор (алгоритм RR и его модификации), приоритетный выбор (алгоритмы RM, EDF, LSTF) и др.).

Существует два подхода к аппаратно-ориентированной реализации алгоритмов управления процессами и ресурсами в многопроцессорных (многоядерных) системах: микропрограммная и схемная.

Микропрограммный способ предполагает внесение обоснованно выбранных функций управления процессами и ресурсами в систему команд целевого процессора. В настоящее время микропрограммно реализуют неизменяемые функции ОС, которые способны работать в пользовательском пространстве и могут вызываться выполняющимся приложением. Микропрограммно реализуют примитивы типа «тестирование и блокировка», «выборка и инкремент», «выборка и добавление» и др. Они позволяют выполнять синхронизацию процессов методом активного ожидания (спин-блокировки) в пространстве пользователя. В некоторых языках программирования микропрограммно реализуют такие высокоуровневые механизмы синхронизации как, например, мониторы. Функции же управления задачами (планировщики, диспетчеры) чаще вызываются автоматически по таймеру и реализуются как процедура, выполняющаяся в пространстве ядра. Обычно такие функции требуют перенастройки для оптимизации вычислительного процесса под конкретный класс прикладных задач, что затрудняет использование микропрограммного подхода.

Перенесение части функций с ядра ОС на управляющий сопроцессор или устройство управления позволяет освободить ядро от планирования задач и функций переключения контекста, обмена сообщениями и синхронизации процессов, что увеличивает скорость выполнения API-функций. Тщательная отладка в ПЛИС данных компонентов гарантирует безопасное функционирование и надежность ОС. Повышение производительности системы происходит за счет:

1) более быстрого выполнения функций ОС по синхронизации и планированию задач при аппаратной реализации.

2) освобождения процессоров от функций ОС, при котором они выполняют только полезную производительную нагрузку.

Архитектура аппаратного ядра многопроцессорной операционной системы

Существует несколько вариантов включения ядра ОС в многопроцессорную систему: 1) с помощью системной общей шины (рисунок 14); 2) с помощью отдельной (периферийной) шины (рисунок 15).

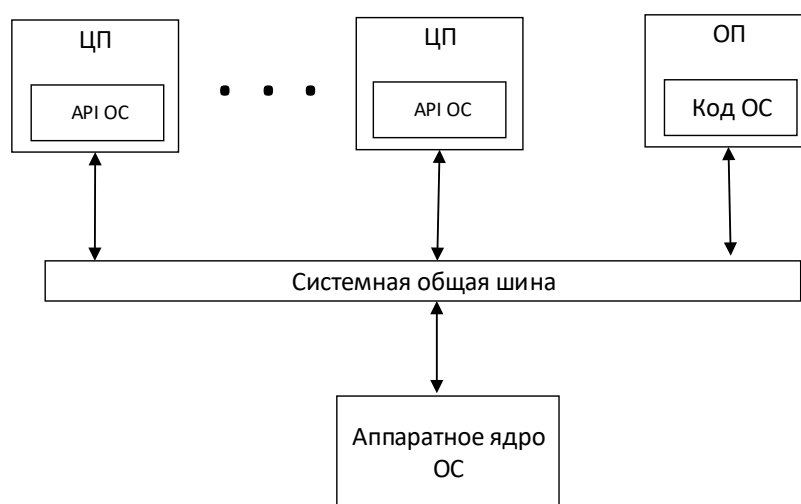


Рисунок 14 - интерфейс аппаратного ядра ОС с общей шиной

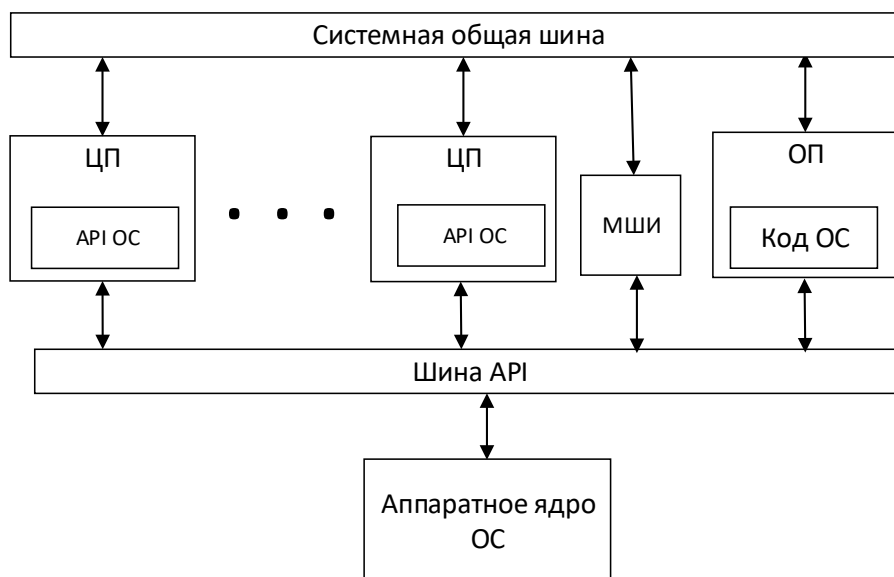


Рисунок 15 - интерфейс аппаратного ядра ОС с отдельной шиной

Первый вариант структурной схемы использует единственную общую шину для взаимодействия между процессорами, памятью системы, а также между API (Application Programming Interface) операционной системы (API ОС) и аппаратным ядром ОС. API ОС является частью операционной системы, обеспечивающей программный интерфейс между выполняющимися приложениями через ядро ОС. API определяет функциональность, которую предоставляет программа операционной системы, при этом API позволяет абстрагироваться от того, как именно эта функциональность реализована. Самой важной частью API операционных систем является множество системных вызовов. Положительной стороной такой архитектуры является то, что единый интерфейс дает возможность использования типовых средств (стандартного интерфейса) для включения аппаратного ядра в многопроцессорную систему. Однако она имеет недостаток, заключающийся в последовательном характере взаимодействия через общую шину, поскольку только два устройства могут связываться на любом промежутке времени. Это обстоятельство не обеспечивает параллелизм связей, что может создать высокий трафик, приводящий к перегрузке общей шины.

В архитектуре отдельной (специальной, периферийной) шины взаимодействие процессоров с оперативной памятью (ОП) по системной шине осуществляется параллельно с взаимодействием API ОС с модулем аппаратного ядра – по специальной шине API. В такой архитектуре системная шина не нагружена дополнительным трафиком от операционной системы, что позволяет увеличить общую производительность многопроцессорной системы. Однако для обеспечения связи аппаратного ядра ОС с программным кодом ОС, хранящимся в оперативной памяти системы, требуется сопряжение шины API с оперативной памятью через некоторый межшинный интерфейс. При этом проблема облегчается тем обстоятельством, что межшинный интерфейс входит в состав аппаратуры некоторых процессоров. Так, например, в soft-процессор Microblaze фирмы Xilinx встроена стандартная шина AXI4 для связи процессоров с общей

памятью и потоковая шина AXI-Stream для связи периферии с той же памятью методом файлового обмена. Здесь потоковая шина работает в режиме межшинного интерфейса.

Устройство управления процессами и ресурсами (УУПР) является аппаратным ядром операционной системы, которое состоит из трех основных частей: интерфейса ядра, блока управления процессами (БУП) и блока управления ресурсами (БУР). На рисунках 16, 17 и 18 представлены возможные способы структурной реализации УУП. Интерфейс ядра включает логику, необходимую для декодирования инструкций API операционной системы и поддержания взаимодействия между узлами.

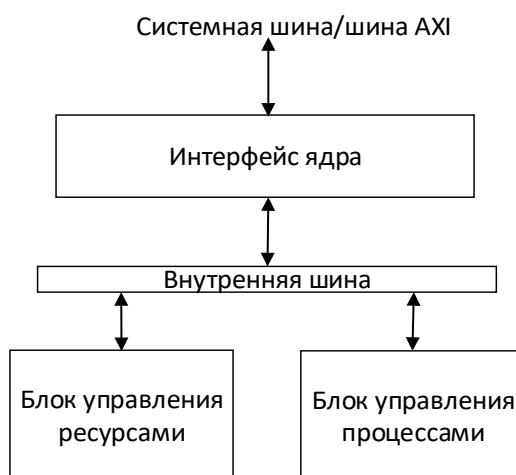


Рисунок 16 – структура аппаратного ядра ОС с общей шиной

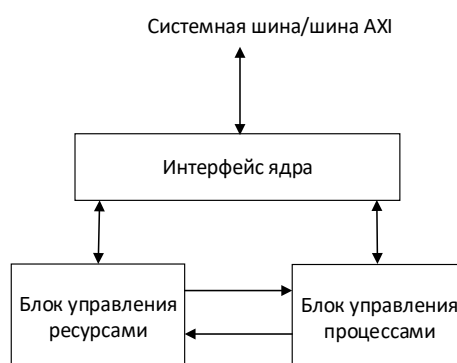


Рисунок 17 - структура аппаратного ядра ОС с использованием непосредственных связей

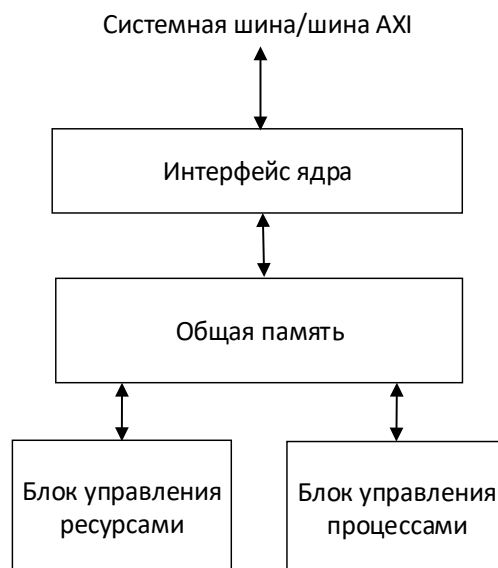


Рисунок 18 - структура аппаратного ядра ОС с общей памятью

Первый вариант (рисунок 16) использует единственную общую шину для взаимодействия между блоками УУПР, а также между API ОС и аппаратным ядром ОС. Положительной чертой данной архитектуры является возможность использования типовых средств подключения аппаратного ядра к многопроцессорной системе. Недостатком является последовательный характер взаимодействия блоков через общую шину, что ограничивает производительность аппаратного ядра.

Вариант реализации, представленный на рисунке 17, заключается в создании непосредственных связей между внутренними модулями. Пропускная способность такой архитектуры значительно выше, как и сложность реализации.

Архитектура с общей памятью, представленная на рисунке 18 характеризуется совместным использованием памяти и регистров, принадлежащим блокам БУП и БУР. Каждый блок имеет ряд регистров и модулей памяти для записи и чтения данных, а также возможность чтения и записи данных из других модулей памяти благодаря единому адресному пространству. Это позволяет создать достаточно простой интерфейс для каждого блока, состоящий из регистров и логики доступа к памяти.

API ОС связывается с аппаратным ядром с помощью записи в соответствующие регистры. Так как регистры и сегменты памяти способны поддерживать сразу несколько писателей и читателей, все модули ядра способны связаться друг с другом. Данная обеспечивает параллелизм в связях как между блоками ядра, так и между аппаратурой ядра и API ОС. В то же время блоки ядра могут получать доступ только к частям регистров и памяти каждого из других модулей, в отличие от API-интерфейса, способного получать доступ ко всей карте распределения памяти.

Основные модули блока управления процессами (рисунок 19) это: диспетчер, планировщик задач, таймер и сторожевой таймер. В его составе также имеется таблица процессов, которая является RAM-памятью для хранения дескрипторов задач, готовых к выполнению. Дескрипторы содержат необходимые данные для управления сохранением и восстановлением контекста.

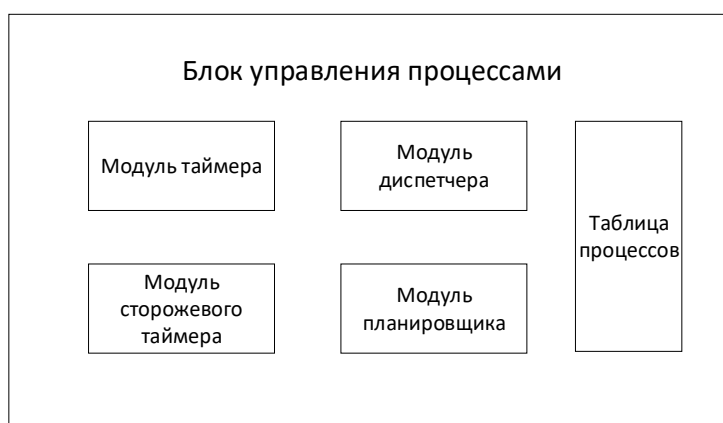


Рисунок 19 – состав блока управления процессами

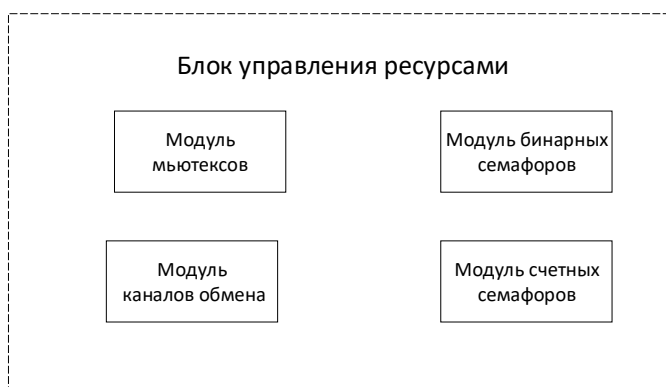


Рисунок 20 – состав блока управления ресурсами

Блок управления ресурсами (рисунок 20) имеет в своем составе средства управления ресурсами (механизмы синхронизации процессов), обеспечивающие взаимодействие параллельных процессов при доступе к общим (разделяемым) ресурсам, основанное на критических интервалах: бинарные и счетные семафоры, мьютексы, очереди сообщений. БУР может включать также и другие механизмы синхронизации, например, флаги событий, почтовые ящики и т.д.

В последующих разделах будут рассмотрены вопросы структурной реализации аппаратного ядра многопроцессорных операционных систем с использованием средств функционального моделирования языке VHDL. При проектировании устройств аппаратной поддержки важен выбор математического аппарата для формализации, обеспечивающего простоту описания, верификации и структурной реализации алгоритмов управления процессами и ресурсами. Наиболее подходящим аппаратом является язык логики недетерминированных автоматов. Использование этого языка позволило разработать НДА- модели алгоритмов взаимодействия параллельных процессов, основанные на критических интервалах (модуль мьютексов), и модели очередей сообщений (модуль каналов). Из средств управления процессами разработаны модули диспетчера и частично планировщика задач, реализация которых показана

Разработанные модели планировщиков позволяют продемонстрировать работоспособность алгоритмов, рассмотренных в главе 2 и теоретических положений, изложенных в главе 1.

При разработке решались практические задачи присущие многим планировщикам: генерация задач, поступающих в планировщик, с определенной интенсивностью по нужному закону распределения приоритетов, реализация приоритетных очередей, осуществление механизма “рандеву” для получения нужными процессорами нужных задач, имитация процессорами

выполнения выбранных задач. Для моделирования использовались актуальные технологии, язык моделирования VHDL и среда моделирования Xilinx 14.7.

Аппаратная реализация функций синхронизации позволяет в значительной степени увеличить надежность операционной системы, которая наиболее важна для специализированных систем и систем реального времени. На рисунке 21 представлен подход, при котором планировщик и диспетчер задач выполнены в виде независимого устройства (специализированного процессора), входящего в состав многопроцессорной системы.

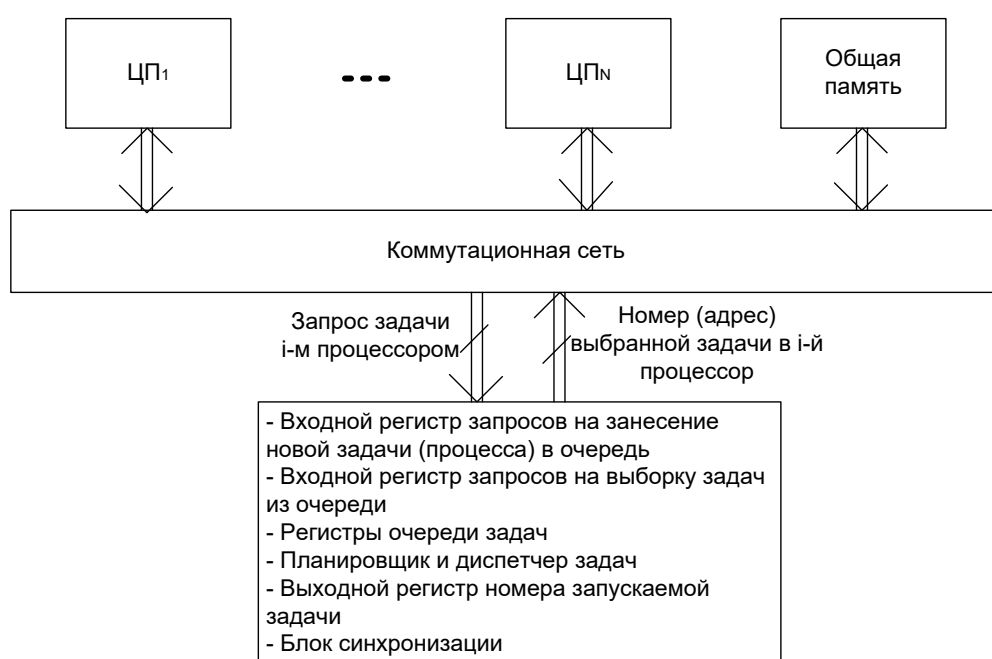


Рисунок 21 - схема подключения аппаратного планировщика-диспетчера задач в многопроцессорную систему

3.2 Структура аппаратного планировщика с глобальной очередью

Выделяют два вида планировщиков задач в n -процессорной системе: с глобальной очередью и с локальной очередью.

Планировщик с глобальной очередью динамически перераспределяет поток задач между всеми процессорами, в результате чего задачи выполняются на любом из n процессоров. Общая схема планировщика задач с глобальной очередью представлена на рисунке 22.

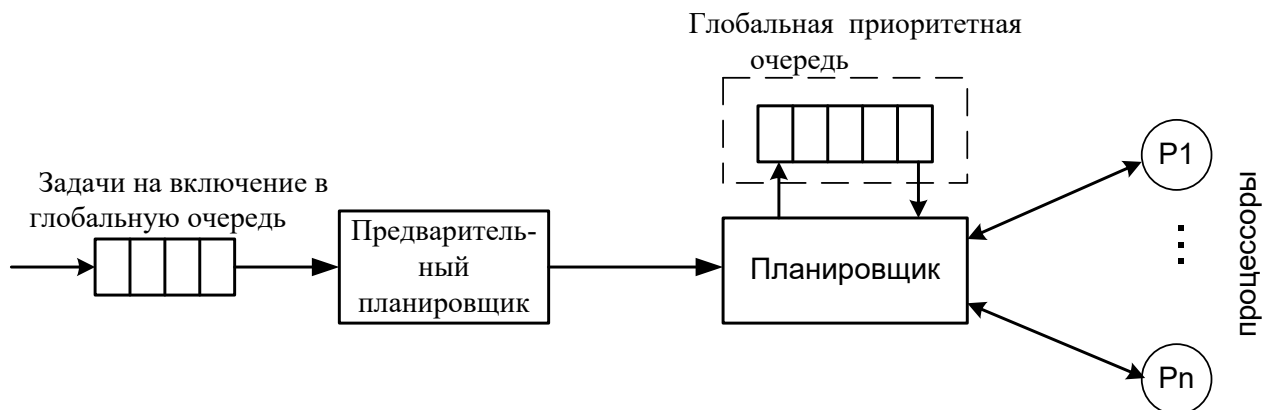


Рисунок 22 - общая схема планировщика задач с глобальной очередью

В модели с глобальной очередью заявки поступают в блок управления очередью, который при возможности добавляет задачи в блок приоритетной очереди или сигнализирует блок очереди выдать задачу в блок синхронизации и назначения задач, который через блок управления свободными процессами получает информацию о доступных процессорах и пересылает им задачи.

Аппаратная поддержка алгоритма диспетчеризации задач осуществлена в несколько упрощенном варианте, а непосредственно, в части постановки задач в очередь, выбора задачи с целью обслуживания в процессоре, помимо этого синхронизации действий серверной части (процессоров), кроме того и клиентской (задачи) на базе метода «рандеву», выбор обслуживающего процессора. Иные функции, к которым относятся ровно как сохранение, так и возобновлении контекста, перемещение из очереди готовых задач в очередь ожидающих и многое другое, осуществляются классическим путем и в представленной модели никаким образом не рассматриваются.

Устройство аппаратной поддержки исследовано, основываясь на примере многопроцессорной системы, которая состоит из четырех процессоров, соединенных общей шиной. Имеется два варианта подключения диспетчера к процессорам. В первом варианте для передачи сигналов между диспетчером и процессорами применяется общая шина, что формирует нежелательный трафик. Во втором варианте каждый процессор имеет свой интерфейс взаимодействия с диспетчером. Несмотря на увеличение сложности процессорного блока из-за увеличения выводимых контактов для необходимых сигналов на рисунке 23 представлен 2й вариант.

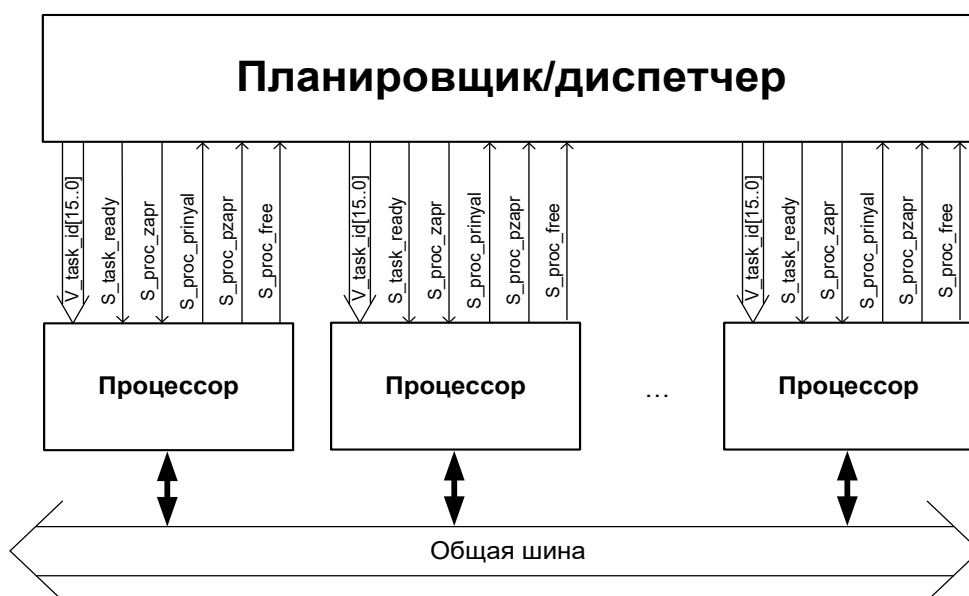


Рисунок 23 -общий вид многопроцессорной системы с аппаратной поддержкой планировщика/диспетчера задач.

Преимуществом данной организации также является еще и тот факт, что при выходе из строя одного из процессоров система не выйдет из строя целиком, а сохранит работоспособность при некотором уменьшении эффективности.

3.3 Моделирование аппаратного планировщика с глобальной очередью на языке VHDL

В ходе выполнения работы в среде Xilinx была создана модель диспетчера задач с глобальной очередью на языке VHDL, которая была дополнена блоками сбора статистики для получения временных характеристик модели. Модель имеет 4 блока процессоров и приоритетную очередь на 16 мест.

Для моделирования событий планировщика с глобальной очередью была разработана система канонических уравнений, описывающих события для заявки в очереди (17).

Таблица 1 – события для заявки в очереди

Обозначение события	Описание частного события	Сигналы на схемах
S_I^t	Ожидание занесения задачи в очередь	$S_task_arrived$
S_Q^t	Задача в очереди	$S_task_in_queue$
S_{FQ}^t	В очереди имеются свободные места	$\overline{S_prior_full}$
S_{GPj}^t	Задача готова к обслуживанию в j -м процессоре, j -й процессор помещается в пул занятых	$S_task_ready_j$
S_{ZPj}^t	Запрос j -го процессора диспетчером	$S_proc_zapr_j$
S_S^{pj}	j -й процессор помещается в пул свободных	$S_proc_free_j$
S_{PZ}^{pj}	j -й процессор выдал сигнал подтверждение запроса	$S_proc_pzapr_j$
S_{PT}^{pj}	j -й процессор задачу принял	$S_proc_prinyal_j$
S_S^p	В процессорном пуле имеются свободные процессоры	S_proc_free
S_{PR}^{pj}	Приоритет j -го процессора	Внутренний сигнал, на схеме не обозначен
S_{OF}^t	Удалить задачу из очереди	Внутренний сигнал, на схеме не обозначен

Продолжение таблицы 1

S_{SL}^{pj}	j -й процессор выбран для обслуживания задачи	$S_{proc_sel_j}$
S_A^{pj}	j -й процессор выполняет обслуживание задачи	proc_work1..proc_work_4
S_{RT}^t	Выдача результата выполнения текущей задачи	Внутренний сигнал, на схеме не обозначен
S_{TO}^{pj}	Снять задачу с исполнения	Внутренний сигнал, на схеме не обозначен
S_E^{pj}	Выполнение задачи закончено	Внутренний сигнал, на схеме не обозначен
S_O^{pj}	j -й процессор находится в режиме ожидания	Внутренний сигнал, не обозначен

На основе введенных выше событий система канонических уравнений, описывающих данные события, будет иметь вид:

- процесс «клиент», который реализуется очередью задач до момента randevу:

$$\begin{aligned}
 S_I^t(t+1) &= x_z \vee \overline{S_I^t S_{FQ}^t} \\
 S_Q^t(t+1) &= S_I^t S_{FQ}^t \vee S_Q^t \overline{S_{SL}^{pj}} \\
 S_{ZPj}^t(t+1) &= S_Q^t S_{SL}^{pj} \vee S_{ZPj}^t \overline{S_{PZ}^{pj}} \\
 S_{GPj}^t(t+1) &= S_{ZPj}^t S_{PZ}^{pj} \vee S_{GPj}^t \overline{S_{PT}^{pj}}
 \end{aligned} \tag{17}$$

где x_z - сигнал запроса задачи на вхождение в очередь.

- процесс «сервер», который реализуется процессором до момента randevу:

$$\begin{aligned}
 S_S^P(t+1) &= (S_S^{p1} \vee S_S^{p2} \vee \dots \vee S_S^{pN}) \vee S_S^P \overline{S_{ZP}^t} \\
 S_O^{pj}(t+1) &= S_{ZP}^t \vee S_O^{pj} \overline{S_{SL}^{pj}}
 \end{aligned}$$

$$S_{SL}^{pj}(t+1) = \bigvee_{\forall j \in N} S_O^{pj} S_{PR}^{pj} \quad (18)$$

$$S_{PZ}^{pj}(t+1) = S_{SL}^{pj} S_{ZPj}^t \vee S_{PZ}^{pj} \overline{S_{GPj}^t}$$

$$S_{PT}^{pj}(t+1) = S_{PZ}^{pj} S_{GPj}^t$$

- события, происходящие после randevу:

$$S_A^{pj}(t+1) = S_{PT}^{pj} S_{GPj}^t \vee S_A^{pj} \overline{S_E^{pj}}$$

$$S_{TO}^{pj}(t+1) = S_A^{pj} S_E^{pj} \vee S_{TO}^{pj} \overline{S_{RT}^t}$$

$$S_{RT}^t(t+1) = S_A^{pj} S_E^{pj} \vee S_{RT}^t \overline{S_{TO}^{pj}} \quad (19)$$

$$S_{OF}^t(t+1) = S_{TO}^{pj} S_{RT}^t$$

$$S_S^{pj}(t+1) = S_{TO}^{pj} S_{RT}^t \vee S_S^{pj} \overline{S_{GPj}^t}$$

На основе данных систем канонических уравнений (17, 18, 19) построен граф НДА алгоритма планирования/диспетчеризации задач в многопроцессорной системе с глобальной очередью задач [5] (рисунок 24).

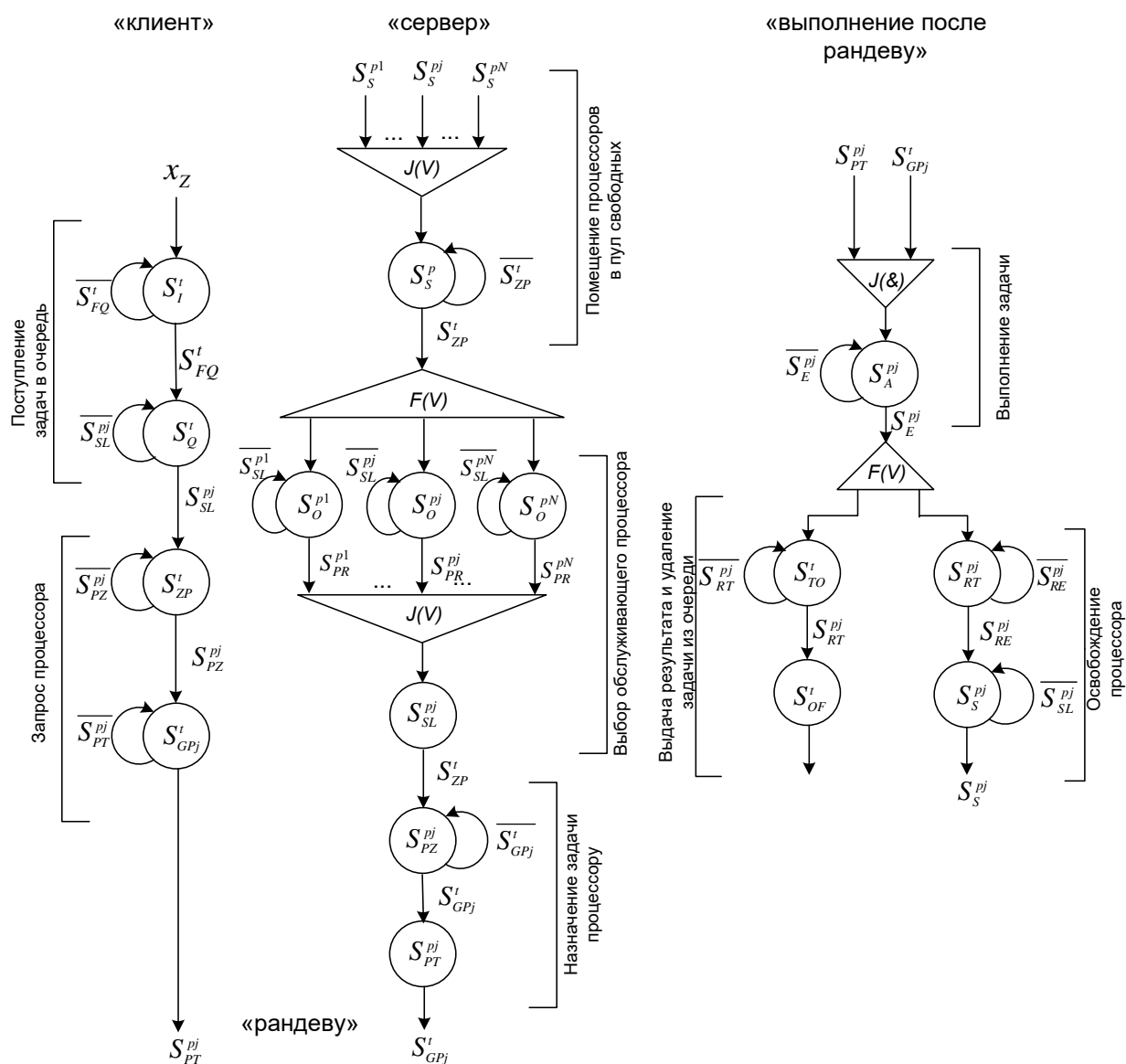


Рисунок 24 - граф НДА алгоритма планирования/диспетчеризации задач в многопроцессорной системе с глобальной очередью задач

Интерфейс взаимодействия процессора с диспетчером представлен на рисунке 25.

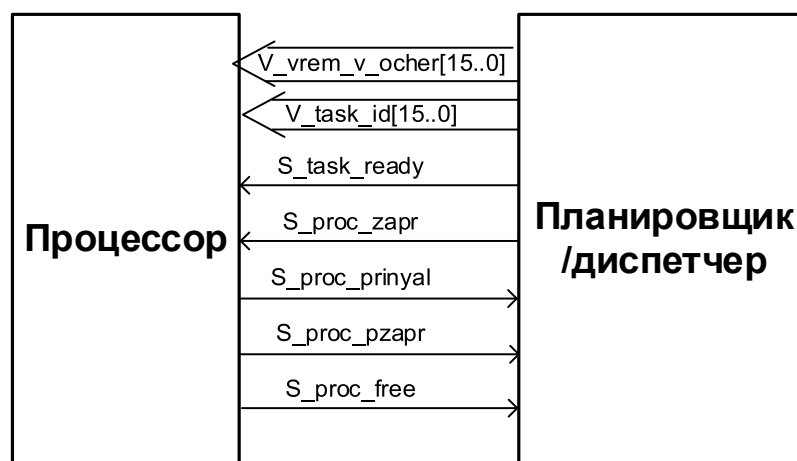


Рисунок 25 - интерфейс взаимодействия процессора с диспетчером

Клиентский процесс формируется таким образом: если существует задача, ожидающая обслуживания, то диспетчер с помощью сигнала S_proc_sel блока выбора процессора выбирает свободный процессор и запрашивает его установкой на нем сигнала S_proc_zapr . Процессор, получивший сигнал запроса, покидает спящий режим, выходя из пула свободных процессоров и снимая активный сигнал S_proc_free , идущий в блок выбора процессора и посылает диспетчеру сигнал подтверждения запроса S_proc_pzapr , сообщая, что готов к приему новой задачи. После того как процессор назначен, диспетчер передает адрес задачи и её время, проведенное в очереди для статистики по шинам $V_task_id[15..0]$ и $V_vrem_v_ocher[15..0]$, выставляя также сигнал V_task_ready , гарантируя что данные на установлены на шинах. Выбранный процессор, получая сигнал V_task_ready , считывает данные с шин и закрепляет их во внутреннем регистре, после чего подтверждает принятие данных выставлением активного уровня сигнала $V_proc_prinyal$ и начинает обработку задачи.

Серверный процесс формируется следующим образом. У процессора, не занятого обслуживанием задач, выставлен сигнал S_proc_free , говорящий о нахождении процессора в спящем режиме. При поступлении в очередь новой задачи планировщик выбирает свободный процессор из пула свободных согласно приоритетному алгоритму, при котором наивысшим приоритетом

обладает 1й процессор. Выбранный процессор выходит из спящего состояния, кроме того диспетчер получает из очереди адрес задачи и время, проведенное задачей в очереди, с помощью сигнала V_task_read и передает его процессору. Процессор, получив адрес задачи, выставляет сигнал S_proc_prinyal и снимая все остальные сигналы.

После вышеперечисленного планировщик может перейти к обработке следующей задачи. После выполнения задачи процессор вновь уведомляет диспетчер установкой сигнала S_proc_free, оказываясь в пуле свободных процессоров.

Функционально устройство планирования и диспетчеризации состоит из четырёх блоков:

- блок управления очередью задач;
- блок приоритетной очереди задач;
- блок выбора процессора с целью обслуживания очередной задачи;
- блок синхронизации и назначения задач согласно процессорам.

Блок управления очередью задач необходим для приема идентификаторов задач в систему. Он осуществляет анализ на наличие свободных мест в очереди задач, и в случае если свободные места имеются, то помещает идентификатор новой задачи в блок приоритетной очереди. Помимо этого, реализует выборку идентификатора самой первой высокоприоритетной задачи, поступившей в очередь, а затем помещает его в свободный процессор.

Блок управления очередью создает очередь задач и отслеживает ее состояние, используя 2 сигнала: S_prior_full и S_prior_empty. Оба сигнала поступают в блок управления очередью из блока приоритетной очереди. Сигнал S_prior_full означает, что очередь заполнена, так как заполнены все регистры, хранящие идентификаторы задач, вследствие чего новые идентификаторы задач, поступающие в планировщик, будут игнорироваться и не будут записываться в очередь. Сигнал S_prior_empty означает, что очередь пуста и из нее невозможно взять задач. О наличии задач в очереди сообщает

сигнал `S_task_in_queue`, поступающий из блока управления очереди в блок синхронизации. При активном уровне сигнала `S_task_queue` блок синхронизации начинает выборку задачи из очереди и запрос свободного процессора на обслуживание этой задачи.

В блок управления очередью из блока генерации задач идут 2 шины: `V_task_id[15..0]` и `prior_of_task_gen[3..0]`, содержащие идентификатор и приоритет задачи соответственно. Сигнал `S_task_arrived` сообщает о поступлении новой задачи в блок управления очередью. Если в очереди есть свободные места, то блок управления очередью выставляет адрес и приоритет задачи на входные шины очереди и сигнализирует блок приоритетной очереди о необходимости записать задачи сигналом `S_wrreq`. Запрос задачи из очереди осуществляется блоком синхронизации с помощью сигнала `S_prior_read`, поступающим в блок управления очередью. При поступлении сигнала `S_prior_read` блок управления очередью формирует сигнал `S_rdreq`, оповещающий очередь о необходимости прочитать из нее и выдать адрес наиболее приоритетной задачи в блок синхронизации.

Блок приоритетной очереди задач осуществляет хранение и сортировку поступающих идентификаторов задач. Идентификатор представляет собой часть контекста задачи, и способен состоять из нескольких полей, одно из которых указывает на адрес задачи в памяти многопроцессорной системы. После запроса от блока управления очередью и считывания адреса и приоритета задачи внутри очереди выполняется функция сортировки “пузырьком” Sort, которая пошагово сравнивает приоритет текущей задачи с приоритетом задачи, стоящей перед текущей задачей и, если приоритет текущей задачи выше, переставляет задачи в очереди местами, либо же извлекает идентификатор запускаемой задачи из вершины списка с целью передачи его в блок синхронизации и назначения задач. Также блок очереди вычисляет время нахождения каждой задачи в очереди и передает его по шине

V_vrem_v_ocher[15..0] в блок синхронизации и назначения задач для последующего вычисления времени нахождения задачи в планировщике.

В разработанных планировщиках было принято, что задачи имеющие меньшее значение приоритета имеют более высокий приоритет и наоборот. Таким образом, задачи 0 приоритета имеют высший приоритет, а 15го приоритета – низший. Очередь состоит из двух массивов регистров: массив приоритетов задач и массив идентификаторов задач (рисунок 26).



Рисунок 26 – структура приоритетной очереди

При добавлении задачи в очередь по шине V_task_id[15..0] поступает идентификатор задачи, по шине prior_of_task[3..0] поступает приоритет задачи. После записи новой задачи в очередь, её приоритет сравнивается с последней задачей в очереди и, если значение приоритета новой задачи ниже (приоритет выше), то идентификаторы задач меняются местами, так происходит до тех пор, пока задача не встанет в конец подсписка задач, выше ее по приоритету.

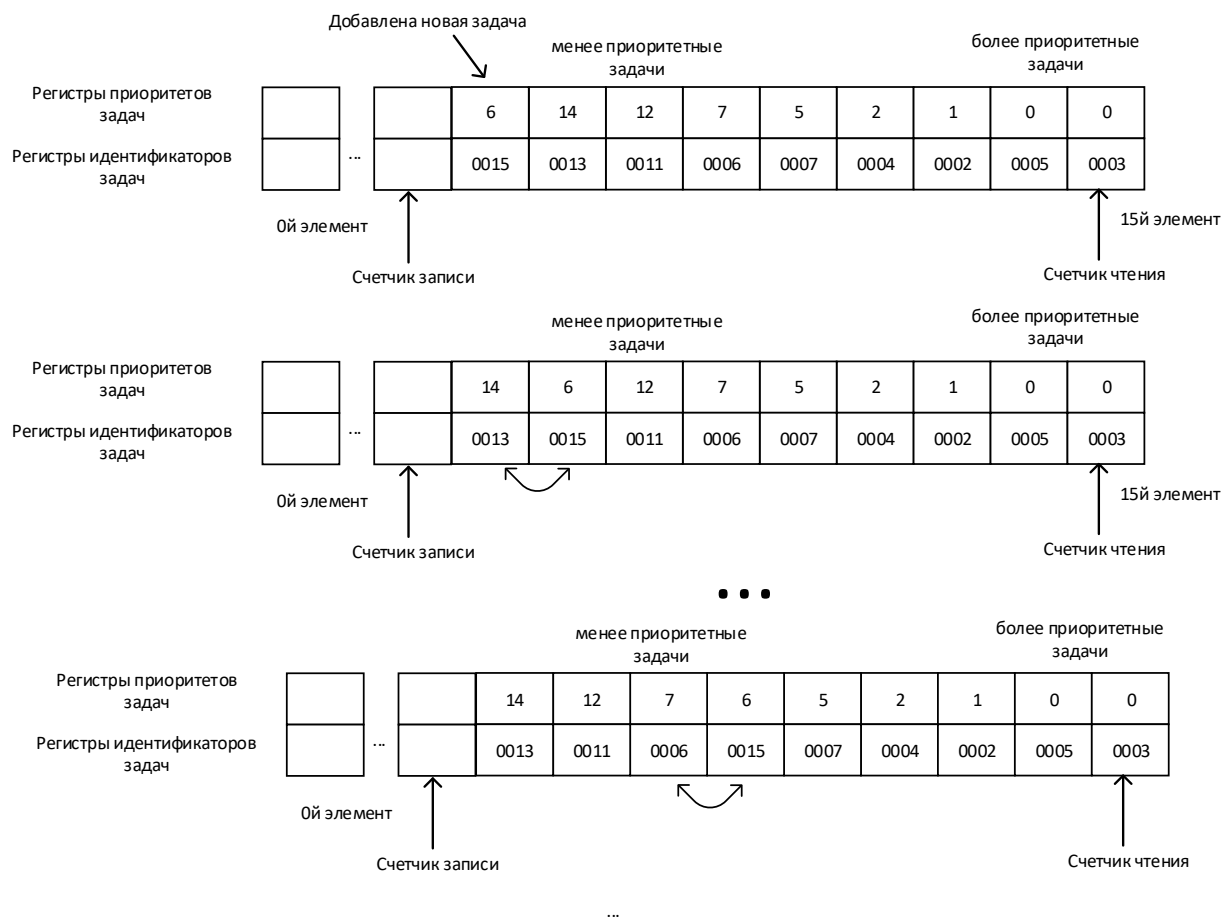


Рисунок 27 – добавление и сортировка элементов очереди

Листинг 1 Функция сортировки блока приоритетной очереди

```

procedure Sort is --сортирует очередь по убыванию приоритета
(слева большее значение приоритетов, справа меньшее)
    variable tempQueueElement : STD_LOGIC_VECTOR(15 DOWNTO 0);
    variable tempPriorElement : STD_LOGIC_VECTOR(3 DOWNTO 0);
    begin
        for k in (queue'length - num_elements_in_queue) to
(queue'length-1) loop
            if(MyPriorsOfTasks(k)(3 downto 0) < MyPriorsOfTasks(k+1)(3
downto 0)) then
                tempQueueElement(15 downto 0):=MyQueue(k)(15 downto 0);
                tempPriorElement(3 downto 0):=MyPriorsOfTasks(k)(3 downto
0);
                MyQueue(k)(15 downto 0):=MyQueue(k+1)(15 downto 0);

```

```

MyPriorsOfTasks(k) (3 downto 0) := MyPriorsOfTasks(k+1) (3
downto 0);
MyQueue(k+1) (15 downto 0) := tempQueueElement(15 downto 0);
MyPriorsOfTasks(k+1) (3 downto 0) := tempPriorElement(3 downto
0);

end if;
end loop;
end procedure;

```

При чтении задач из приоритетной очереди всегда читается 15й (крайний правый) элемент, имеющий наивысший приоритет. После чтения из очереди выполняется сдвиг всех её элементов вправо, чтобы затереть только что вышедший элемент.

Блок выбора процессора принимает все сигналы «Свободен» (*S_proc_free*) от всех процессоров в системе. Он определяет свободные в системе процессоры, с помощью сигналов *S_proc_free_1...S_proc_free_4*, идущих от каждого процессора. Если хоть один из сигналов имеет активный уровень, значит в системе имеются свободные процессоры и сигнал *S_proc_free*, идущий в блок синхронизации будет также иметь активный уровень, оповещая систему о возможности выполнения задач. Для определения, какой именно процессор следующим должен быть загружен задачей используется схема взаимоисключения и схема приоритетов, результатом которой служат сигналы *S_proc_sel_1...S_proc_sel_4*, идущие в блок синхронизации. В один момент времени только один из этих сигналов имеет активный уровень или ни один, если все процессоры загружены.

Блок синхронизации и назначения задач исследует сведения о наличии ожидающих задач в очереди, а помимо этого и свободных процессоров, которые назначаются с целью обработки данных задач. Указанный блок, применяя механизм синхронизации «рандеву», реализует взаимодействие очереди задач с процессорами согласно описанному выше интерфейсу. Блок выбора процессора создает сведения о том, какой из свободных процессоров

выбран с целью обслуживания текущей задачи. Идентификатор принятой на обслуживание задачи передается блоком приоритетной очереди задач по шине *V_task_id*. Помимо этого, блок управления очередью задач уведомляет о состоянии очереди и реализует выборку идентификатора задач из блока приоритетной очереди согласно требованию блока синхронизации и назначения задач.

Схема созданной модели диспетчера показана на рисунке 28.

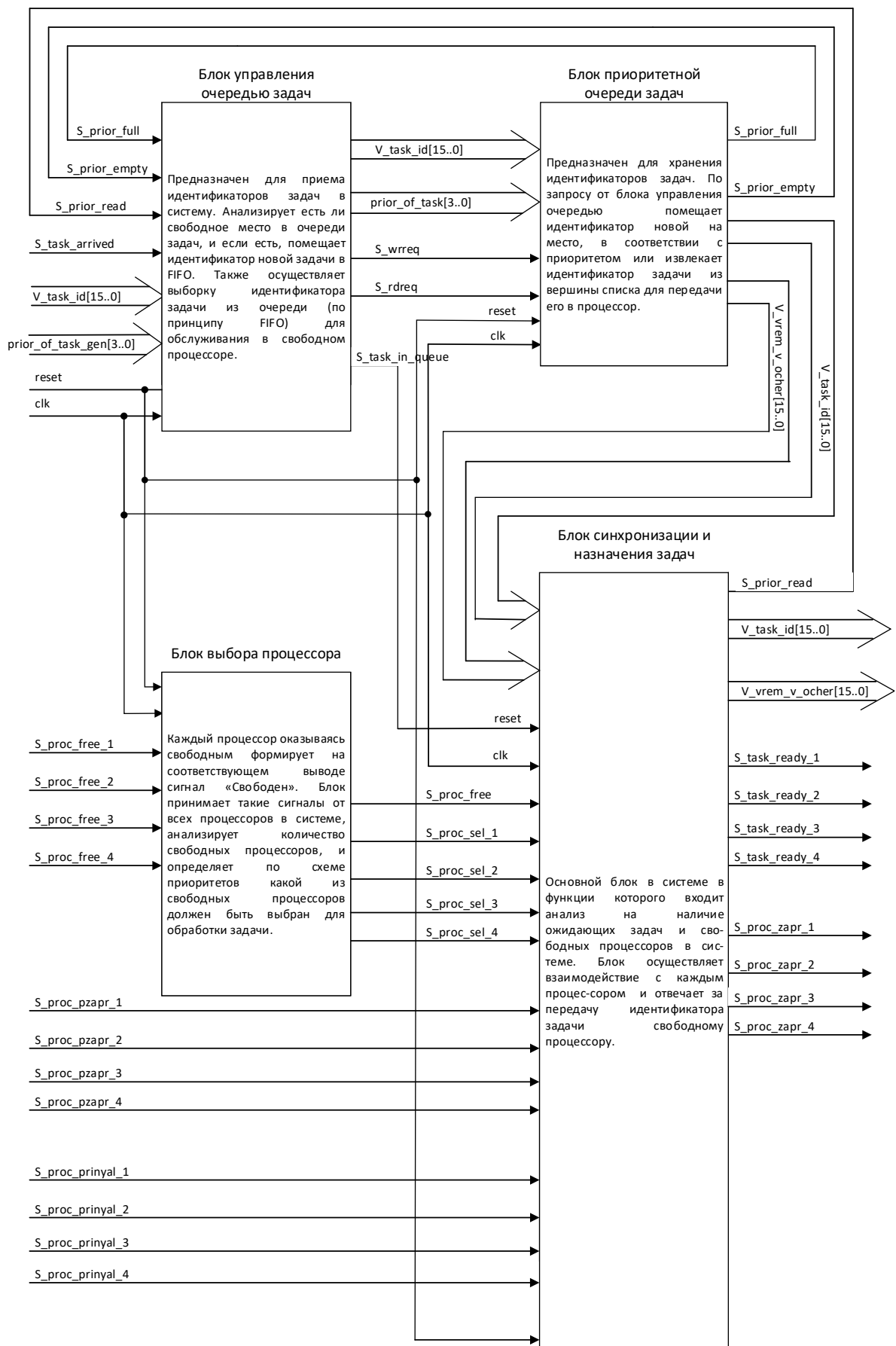


Рисунок 28 - схема созданной модели планировщика с глобальной очередью в среде Xilinx

При поступлении заявки в систему блок управления очереди, если сигнал блока очереди `S_prior_full` установлен в '0', выставляет на вход блока очереди задач сигнал `S_wrreq` в '1', говорящий, что нужно считать задачу из блока управления очередью, а также устанавливает на шину `V_task_id_o` 16-битный идентификатор задачи и на шину `prior_of_task` 4-битный приоритет задачи.

При поступлении сигнала из блока синхронизации и назначения задач в блок управления очередью `S_prior_read` блок управления очередью, при отсутствии у него на входе сигнала с очереди `S_prior_empty`, выставляет на вход блока очереди сигнал `S_rdreq`, после чего блок очереди задач выставляет на шину `V_task_id_bs` адрес первой в очереди задачи и функцией `Sdvig_ocher_vpravo` смещает все элементы очереди ближе к началу, а последний элемент затирает нулями. Вместе с этим блок управления очередью выставляет на вход блока синхронизации и назначения задач сигнал `S_task_in_queue`, говорящий, что задача из очереди помещена вход `V_task_id_bs`.

Полная модель системы планировщика с глобальной очередью задач включает также блок генерации задач, генерирующий заявки в систему с нужной интенсивностью и блок процессоров, имеющих интерфейс связи с блоком выбора процессоров и блоком синхронизации задач, а также имитирующих работу над полученными задачами.

Схема планировщика с глобальной очередью вместе с блоком генерации задач и блоками процессоров представлена на рисунке 29.

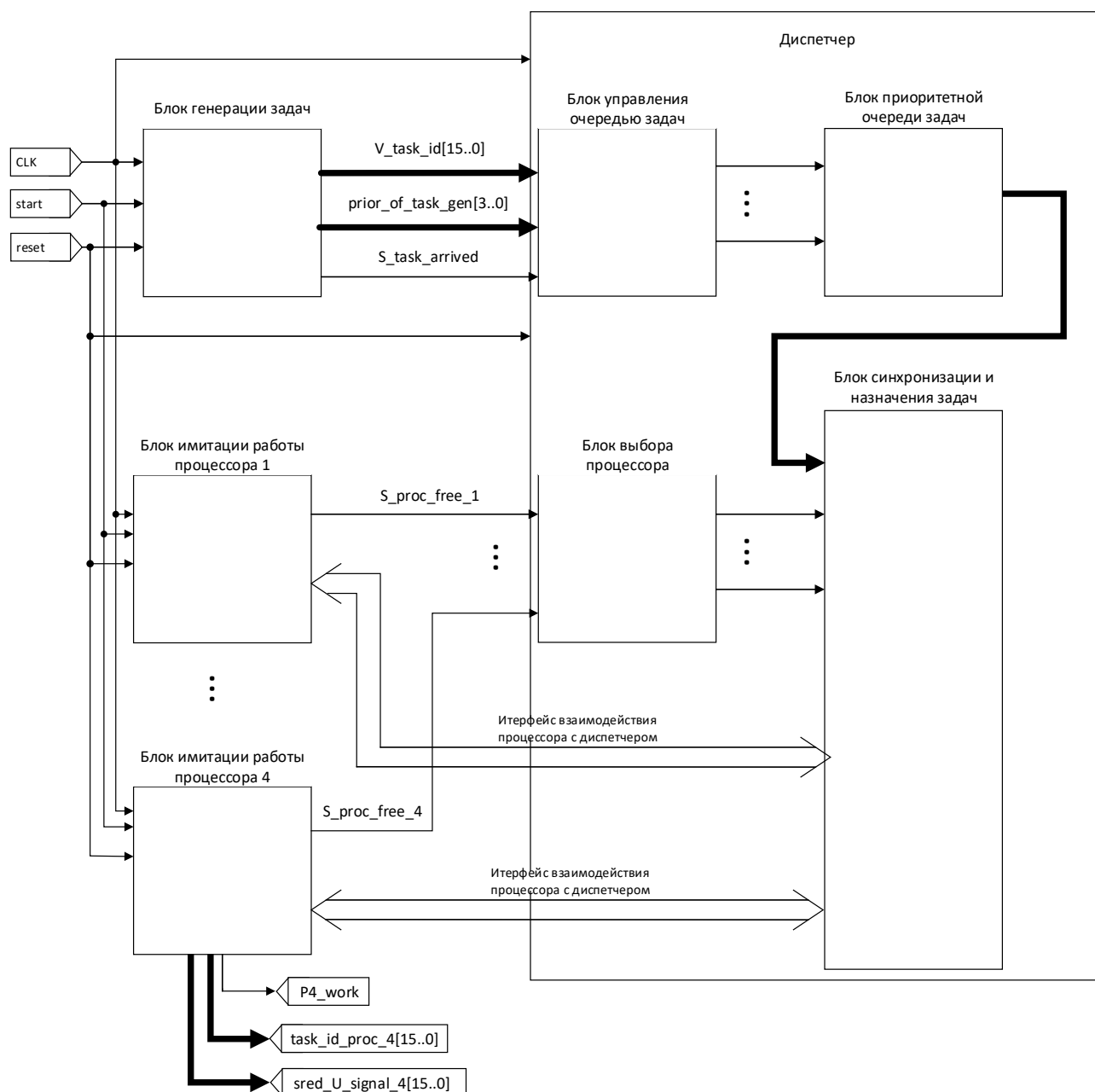


Рисунок 29 – общий вид структурной схемы с глобальной приоритетной очередью

3.4 Структура аппаратного планировщика с локальными очередями

Важной причиной снижения производительности является высокая вероятность перезагрузки кэш в системах с квантованием. Так при переключениях контекста незавершенная задача помещается в конец очереди,

после чего может быть назначена диспетчером не в тот процессор, в котором она обрабатывалась ранее и где кэш полон блоками программы и данных, а в любой другой. Поэтому потребуется выгрузка содержимого кэш из предшествующего процессора и загрузка кэш текущего, на что значительно тратится процессорное время.

В планировщиках/диспетчерах с локальными очередями у каждого процессора имеется своя очередь задач. Функция диспетчера в этом случае ограничивается лишь выборкой задачи из очереди и помещением её в процессор. При переключении контекста по кванту времени задача остается в той же очереди, в которой находилась заранее, из-за чего перезагрузки кэша процессора не требуется. В системе действует сразу N процессоров, задачи из очередей выбираются бесконфликтно, что благотворно сказывается на производительности. Каждая очередь оповещает блок выбора очереди о наличии в ней свободных мест, в результате вновь поступившая задача в систему записывается в наименее заполненную очередь.

Диспетчер задач в данном случае распределен и является независимым аппаратным устройством в составе многопроцессорной системы (рисунок 30).

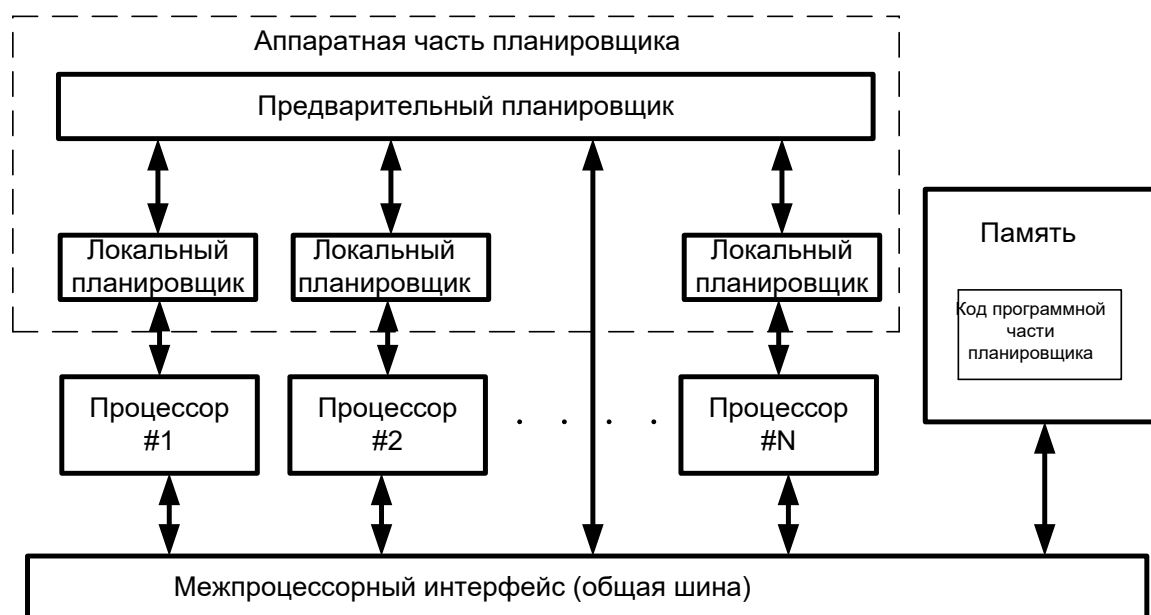


Рисунок 30 - схема подключения устройства управления процессами в многопроцессорную систему

Аппаратно устройство управления состоит из двух частей: предварительный планировщик и локальный планировщик. Предварительный планировщик передает задачу локальному планировщику. В реальной системе это процедура сопровождается выделением раздела области в общей памяти и присвоением задаче уникального идентификатора, однозначно указывающего на задачу в памяти. Идентификатор помещается в одну из локальных очередей.

Модель многопроцессорной системы с локальными очередями с использованием аппаратного устройства управления процессами представлена на рисунке 31. Для упрощения программной модели принято, что часть функций планировщика и диспетчера задач, которые связаны с сохранением и восстановлением контекстов задач, а также прерыванием по истечении кванта процессорного времени реализуются традиционно и в модели не рассматриваются.

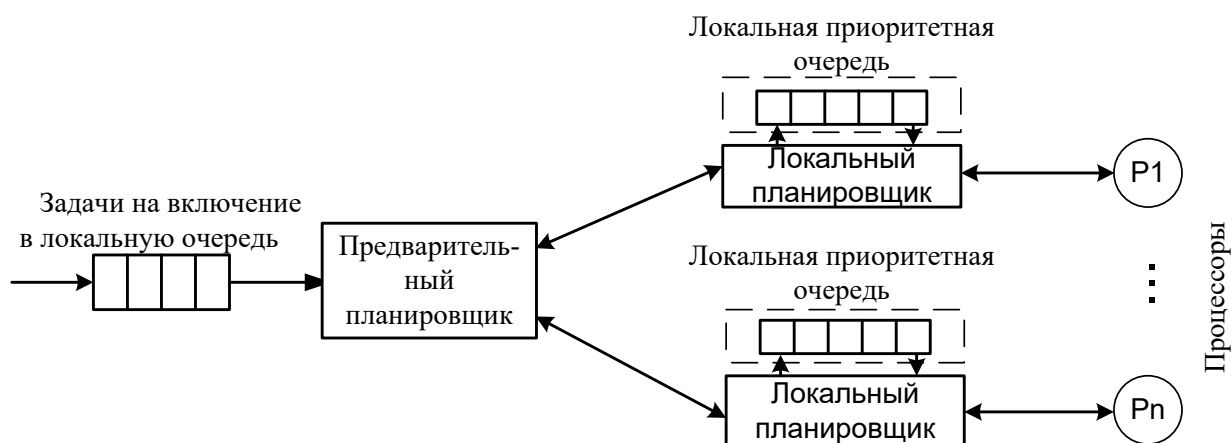


Рисунок 31 - модель многопроцессорной системы с локальными очередями

3.5 Моделирование аппаратного планировщика с локальными очередями на языке VHDL

В ходе выполнения работы в среде Xilinx была создана модель диспетчера задач с локальной очередью на языке VHDL. В модели

присутствует 4 блока процессора и 4 приоритетных очереди на 4 места каждая. Данная схема была доработана и дополнена вспомогательными модулями, такими как блок генерации задач, генерирующий задачи, распределенные по экспоненциальному закону, блоки процессоров, принимающие задачи на выполнение и имитирующие их обработку и блоки сбора статистики для получения временных характеристик модели.

Для моделирования событий планировщика с локальными очередями также была разработана система канонических уравнений, описывающих события для заявки в очереди.

Таблица 2 – события для заявки в очереди

Обозначение события	Описание частного события	Сигналы на схеме
S_I^t	Ожидание поступления новой задачи	$S_task_arrived$
S_U^{Qj}	Функция состояния очереди (количество задач в очереди j -го процессора)	$S_use [5...0]$
$\overline{S_{FQj}^t}$	В j -й локальной очереди имеются свободные места	————— S_prior_fullj
S_{FQ}^t	Все очереди заполнены	S_prior_full
S_{Qj}^t	Задача в очереди j -го процессорного узла	$S_task_in_queue$
S_{SL}^{Qj}	Выбор очереди с минимальным числом задач и занесение новой задачи в выбранную очередь	S_wrreg
S_{ZPj}^t	Запрос j -го процессора локальный диспетчером	$S_proc_zapr_j$
S_{GPj}^t	Задача готова к обслуживанию в j -м процессоре	$S_task_ready_j$
S_S^{pj}	Свободен j -й процессор	S_proc_free
S_{PZ}^{pj}	Подтверждение запроса j -м процессором	$S_proc_pzapr_j$

Продолжение таблицы 2

S_{PT}^{pj}	Задачу j -й процессор принял	$S_{proc_prinyal_j}$
S_O^{Qj}	Ожидание поступления задачи в очередь j -го процессора	Внутренний сигнал, на схеме не обозначен
S_A^{pj}	Выполняет обслуживание задачи j -й процессор	$proc_work1..proc_work_4$
S_{TO}^t	Снять задачу с исполнения	Внутренний сигнал, на схеме не обозначен
S_{OF}^t	Удалить задачу из очереди	Внутренний сигнал, на схеме не обозначен
S_{SZ}^{pj}	Процессор в режиме ожидания	Внутренний сигнал, на схеме не обозначен
S_E^{pj}	Выполнение задачи закончено	Внутренний сигнал, на схеме не обозначен
S_{RT}^{pj}	Выдача результата выполнения текущей задачи	Внутренний сигнал, на схеме не обозначен

Система канонических уравнений, описывающих события, представленные в таблице выше, будет иметь следующий вид:

- процесс «клиент», который реализуется локальным диспетчером до момента randevu:

$$\begin{aligned}
 S_I^t(t+1) &= x_Z \vee S_I^t \overline{S_{FQ}^t} \\
 S_O^{Qj}(t+1) &= S_{FQ}^t \vee S_O^{Qj} \overline{S_{SL}^{Qj}}; \\
 S_{SL}^{Qj}(t+1) &= \bigvee_{\forall j \in N} S_O^{Qj} \overline{S_U^{Qj}} \\
 S_Q^t(t+1) &= S_{SL}^{Qj} \vee S_Q^t \overline{S_{SZ}^{pj}} \\
 S_{ZPj}^t(t+1) &= S_Q^t S_{SZ}^{pj} \vee S_{ZPj}^t \overline{S_{PZ}^{pj}}
 \end{aligned} \tag{20}$$

$$S_{GPj}^t(t+1) = S_{ZPj}^t S_{PZ}^{pj} \vee S_{GPj}^t \overline{S_{PT}^{pj}}$$

где x_Z - поступление запроса задачи на постановку её в очередь.

- процесс «сервер», который реализуется процессором до момента randevу:

$$\begin{aligned} S_{SZ}^{pj}(t+1) &= S_S^{pj} \vee S_{SZ}^{pj} \overline{S_Q^t} \\ S_{PZ}^{pj}(t+1) &= S_{SZ}^{pj} S_Q^t \vee S_{PZ}^{pj} \overline{S_{ZPj}^t} \\ S_{PT}^{pj}(t+1) &= S_{PZ}^{pj} S_{ZPj}^t \vee S_{PT}^{pj} \overline{S_{ZPj}^t} \end{aligned} \quad (21)$$

-СКУ, которая описывает события после randevу:

$$\begin{aligned} S_A^{pj}(t+1) &= S_{PT}^{pj} S_{GPj}^t \vee S_A^{pj} \overline{(S_E^{pj} \vee S_{PK}^{pj})} \\ S_{TO}^t(t+1) &= S_A^{pj} S_E^{pj} \vee S_{TO}^t \overline{S_{RT}^{pj}} \\ S_{RT}^{pj}(t+1) &= S_A^{pj} S_E^{pj} \vee S_{RT}^{pj} \overline{S_{TO}^t} \\ S_{OF}^t(t+1) &= S_{RT}^t S_{TO}^{pj} \\ S_S^{pj}(t+1) &= S_{RT}^t S_{OF}^{pj}. \end{aligned} \quad (22)$$

Выбор очереди для записи в неё вновь поступившей задачи выполняется в соответствии с выражением:

$$S_{SL}^{Qj} = \min_{(\forall j \in N)} \{S_U^{Qj}\} \quad (23)$$

Выражение 23 означает выбор очереди с минимальным количеством занятых мест, где N – число очередей в многопроцессорной системе.

Уравнениям 20, 21, 22 соответствует граф НДА (рисунок 32), который содержит клиентскую и серверную части алгоритма управления взаимодействующими процессами, которые описывают события до и после randevу. Момент randevу описан оператором J (&).

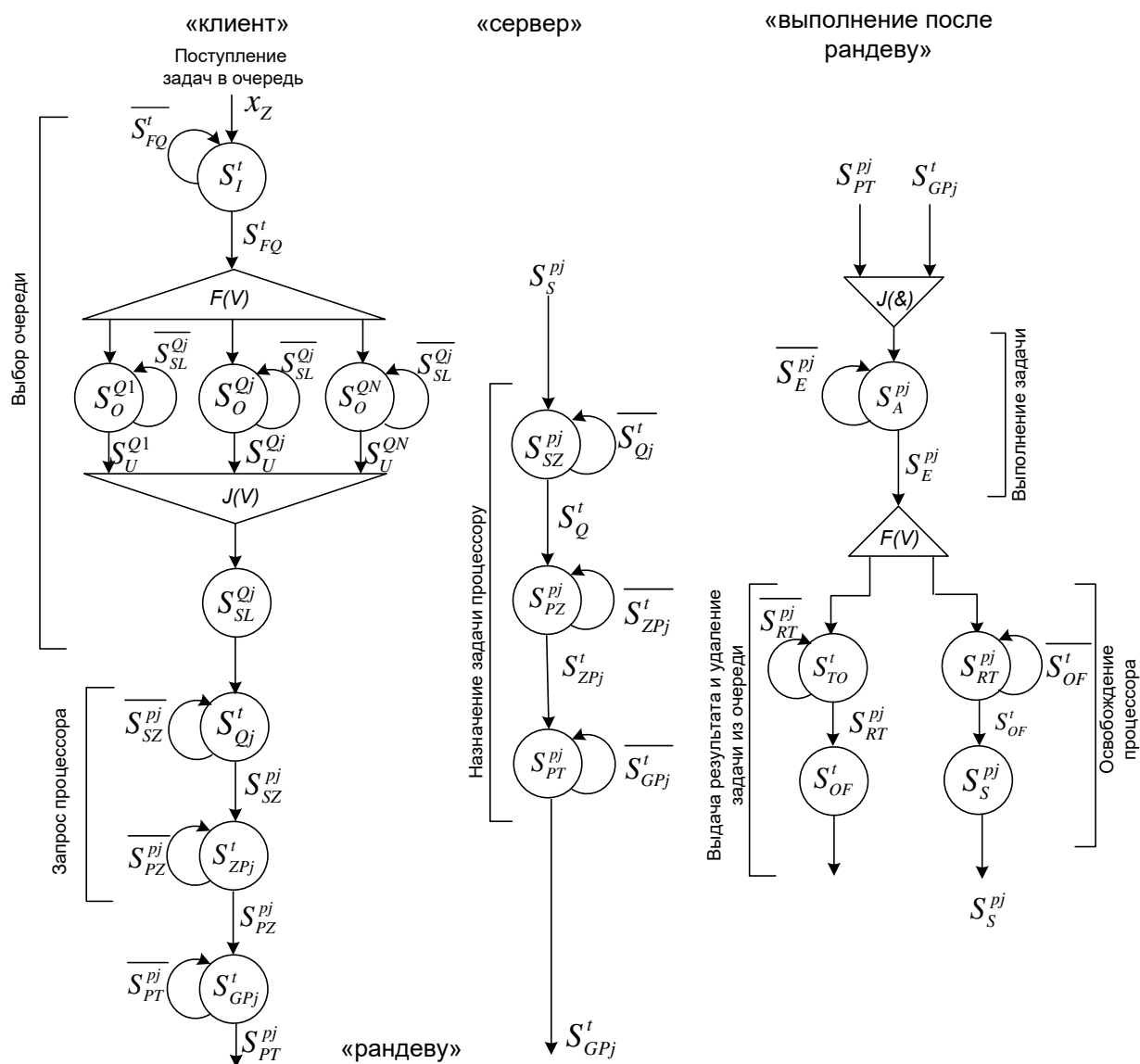


Рисунок 32 - граф НДА алгоритма планирования/диспетчеризации задач в многопроцессорной системе с локальными очередями задач

Интерфейс взаимодействия устройства локального диспетчера с каждым из процессоров и с планировщиком задач представлен на рисунке 33.



Рисунок 33 - интерфейс взаимодействия устройства локального диспетчера с каждым из процессоров и с планировщиком задач

Функционирование представленных выше устройств происходит по следующему алгоритму. Если блок назначения задач получил новую задачу, он помещает ее идентификатор и приоритет на шины $V_task_id[15..0]$ и $V_task_prior[3..0]$ соответственно. Локальный диспетчер, получив на своем входе сигнал S_wrreq , помещает идентификатор задачи и приоритет в очередь, где задача устанавливается в нужное место с помощью функции $Sort()$. Если в очереди локального диспетчера имеется задача, то локальный диспетчер, при условии, что процессор свободен и выставил сигнал S_proc_free в активный уровень, выставляет на выходе сигнал S_proc_zapr , запрашивая процессор. Незанятый процессор снимает сигнал S_proc_free и выставляет сигнал S_proc_pzapr , подтверждая запрос. После этого локальный диспетчер высвобождает из очереди задачу с наивысшим приоритетом, находящуюся в голове очереди, и выставляет её идентификатор и время, проведенное в очереди, на шины $V_task_id[15..0]$ и $V_vrem_v_ocher[15..0]$, а также сигнал S_task_ready , гарантирующий нахождение данных на шинах. Процессор, получив сигнал S_task_ready , считывает данные с шин $V_task_id[15..0]$ и $V_vrem_v_ocher[15..0]$. После этого процессор выставляет сигнал $S_proc_prinyal$, говорящий о принятии данных задачи и начале её обработки.

Алгоритм работы устройства управления процессами содержит две части: клиентскую и серверную. Клиентская часть осуществляет постановку задач в приоритетную очередь, серверная часть выбирает самую приоритетную задачу из головы очереди и запускает её на обслуживание в процессор. Действия клиента и сервера синхронизируются по схемам «писатели-читатели» и «рандеву» [4], изученным во 2й главе, причем последняя предполагает наличие выбранной задачи и готовность процессора к обслуживанию этой задачи.

Аппаратно устройство управления процессами представлено в виде трех типов блоков:

- блок назначения задач;
- N локальных диспетчеров с приоритетными очередями, где N – число процессоров в многопроцессорной системе;
- блок выбора очереди.

Блок локального диспетчера с приоритетной очередью хранят готовые к выполнению идентификаторы задач. Число этих блоков равно числу процессоров в системе. Блоки локальных диспетчеров имеют меньший размер очередей, однако за счет того, что их количество больше общее число мест в очередях одинаково как для планировщика с глобальной очередью, так и для планировщика с локальными очередями и равно 16. Записывать задачи в очередь может только локальный диспетчер с помощью сигнала S_wrreq , поступающего с блока назначения задач по отдельной линии. Чтение задач из очереди осуществляется также только локальным планировщиком с помощью сигнала S_rdreq , поступающим с процессора по индивидуальным линиям. Адрес и приоритет задачи передаются по общим шинам $V_task_id[15..0]$ и $V_task_prior[3..0]$ соответственно, причем только в одну из очередей одновременно. Локальный диспетчер получает данные с шины только если получил сигнал S_wrreq . Если в очереди имеется хоть одна задача процессор

читает её из своей очереди с помощью локального диспетчера, если задач в очереди нет – процессор переходит в режим ожидания.

Сигналы `S_prior_full` поступает из блоков локальных диспетчеров в блок назначения задач и оповещает его о заполненной той или иной очереди, вследствие чего о невозможности записи в нее задач.

Блок выбора очереди предназначен для оповещения блока назначения задач о номере очереди с наименьшим количеством задач. Каждый блок локального диспетчера с приоритетной очередью по собственной шине `V_use[4..0]` передает количество задач в данной очереди. Блок выбора очереди выбирает очередь с наименьшим числом задач и выставляет номер нужной очереди на шину `id[2..0]`, ведущую в блок назначения задач, который считывает номер очереди и в соответствии с ним пытается записать адрес задачи в нужную очередь. Если все очереди заполнены, то идентификаторы последующих задач, поступающих в систему, игнорируются пока, какая-то из очередей не освободится.

Схема созданной модели планировщика с локальными очередями в среде Xilinx изображена на рисунке 34.

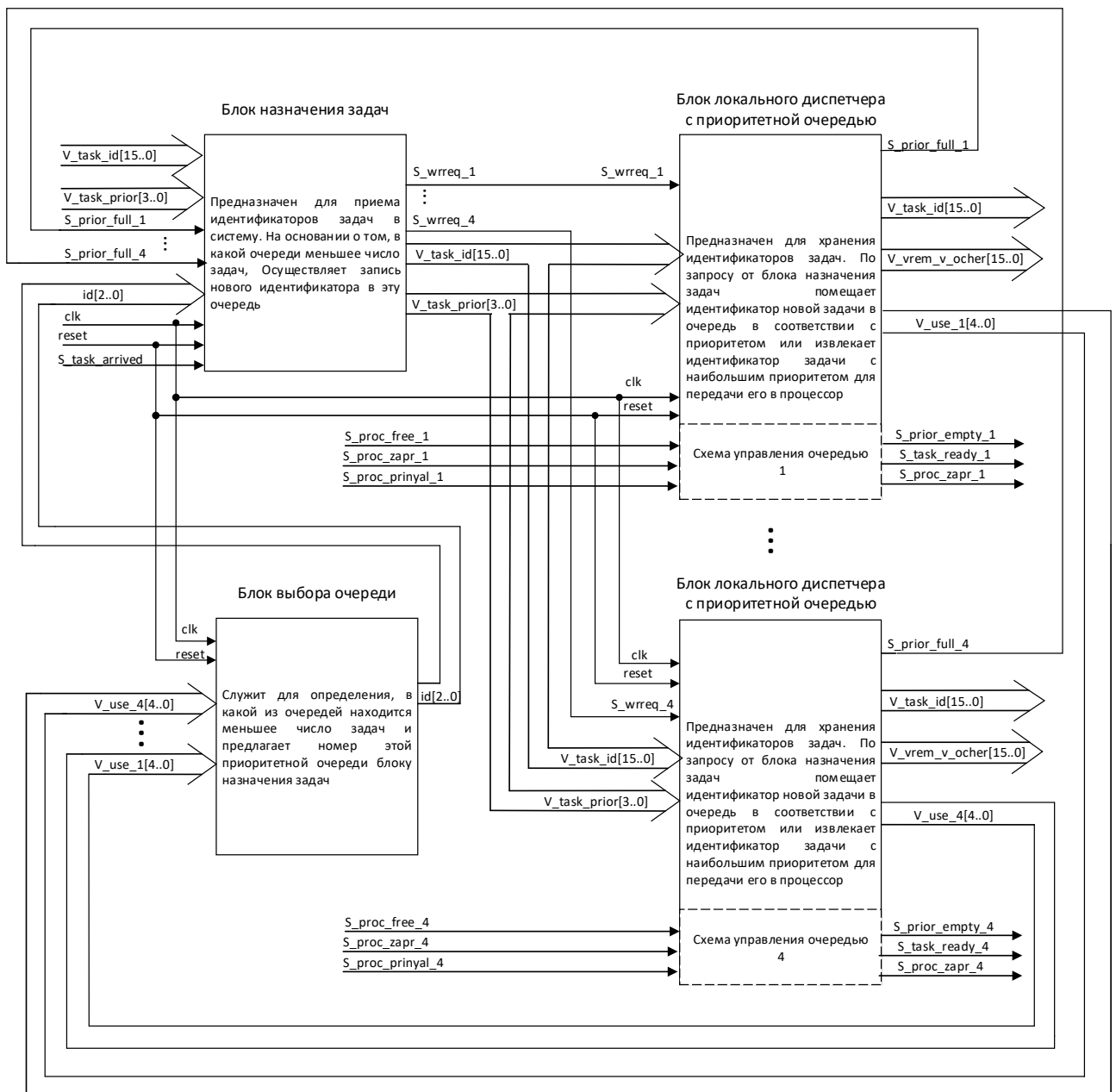


Рисунок 34 - схема созданной модели планировщика с локальными очередями в среде Xilinx

При поступлении заявки в систему блок назначения задач исходя из значения на входе $id[2..0]$ выбирает очередь задач и, если сигнал с этой очереди $S_prior_full_n$ не равен '1', устанавливает на ее входе сигнал S_wrrreq_n , а на шине $V_task_id[15..0]$ адрес задачи и на шине $V_task_prior[3..0]$ приоритет задачи. После чего выбранная очередь записывает к себе эту задачу и вызывает функцию Sort, которая сортирует очередь по убыванию приоритета (последовательно сравнивается текущая задача со следующей и, если у текущей

задачи значение приоритета меньше, задачи меняются местами). Функция Sort в планировщике с локальными очередями аналогична функции Sort в планировщике с глобальной очередью.

С помощью выхода $V_use[4..0]$ каждая очередь оповещает блок выбора очереди о количестве элементов в ней.

При попадании заявки в очередь, и наличия сигнала свободного процессора $S_proc_free_n$, блок очереди запрашивает готовность процессора сигналом $S_proc_zapr_n$, после чего ожидает подтверждения запроса от процессора $S_proc_pzapr_1$ и выставляет сигнал S_task_ready , $V_task_id [15..0]$ (адрес задачи) и $V_vrem_v_ocher[15..0]$ (время нахождения задачи в очереди для определения общего времени нахождения задачи в планировщике). После подтверждения принятия запроса процессором $S_prinyal_n$, происходит ожидание освобождения процессора - сигнала $S_proc_free_n$.

Схема планировщика с локальными очередями вместе с блоком генерации задач и блоками процессоров представлена на рисунке 35.

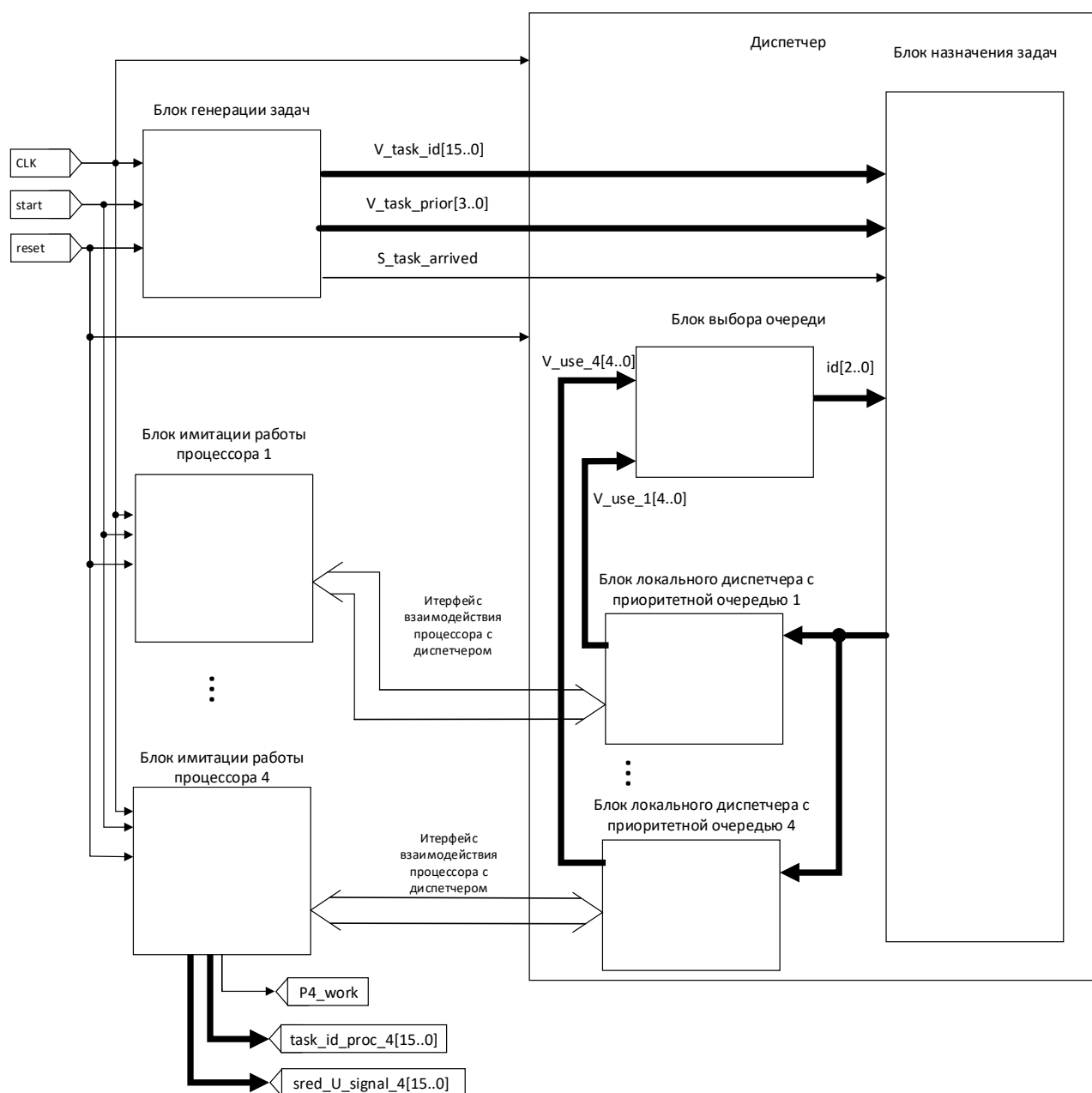


Рисунок 35 - общий вид структурной схемы с локальными приоритетными очередями

3.6 Получение временных диаграмм и временных характеристик

Для разработанных схем планировщиков с глобальной и локальными очередями в среде Xilinx были построены временные диаграммы, в которых

измеряются такие временные характеристики планировщиков как латентность (в условиях бесконфликтности), среднее время нахождения в очереди заявок каждого приоритета (время отклика планировщика), время ожидания обслуживания задачи (время отклика системы).

Для моделирования принято, что интенсивность потока заявок на входе планировщика (диспетчера) задач изменяется от 0.04 до 0.07 зад/ед. времени (числом задач на один процессорный такт), а среднее время выполнения (время обслуживания) задачи в процессоре равна 52 ед. времени (процессорных тактов), таким образом интенсивность обслуживания равняется $1/52=0,0192$ зад/ед. времени. Измерения проводились на временных интервалах, когда загрузка составляла среднее значение ($p=0.5$) и высокое значение ($p=0.9$). Загрузка достигалась изменением интенсивности входящего потока задач в диапазоне от 0.04 до 0.07 зад/ед. времени. Число процессоров в многопроцессорной системе принято равным четырем. Модельное время (общее число экспериментов) составляет 5 ед. миллионов тактов (200 ед. миллионов нс. при периоде синхросигнала в 40 нс.), что позволило получить достаточно статистически стабильные результаты.

Таким образом, эксперименты проведены как в средненагруженном, так и высоконагруженном режиме системы реального времени. Это дает возможность оценить как сроки выполнения задач (дедлайн) разного приоритетного уровня, так и джиттер задач в многопроцессорной системе с глобальной и локальными очередями задач.

На рисунках 36 и 37 изображены временные диаграммы работы планировщика с глобальной очередью в условиях средней ($p=0.5$) и высокой ($p=0.9$) загрузки соответственно.

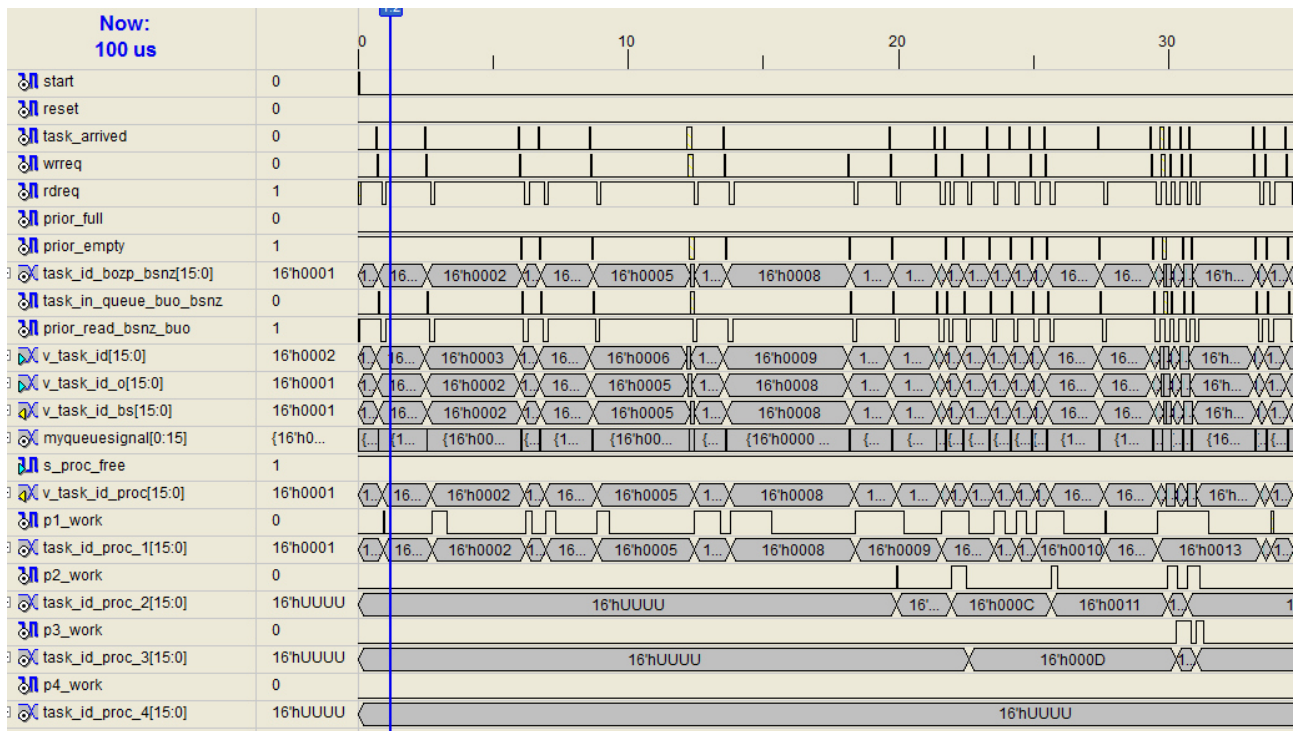


Рисунок 36 - временная диаграмма работы планировщика с глобальной очередью в условиях средней загрузки

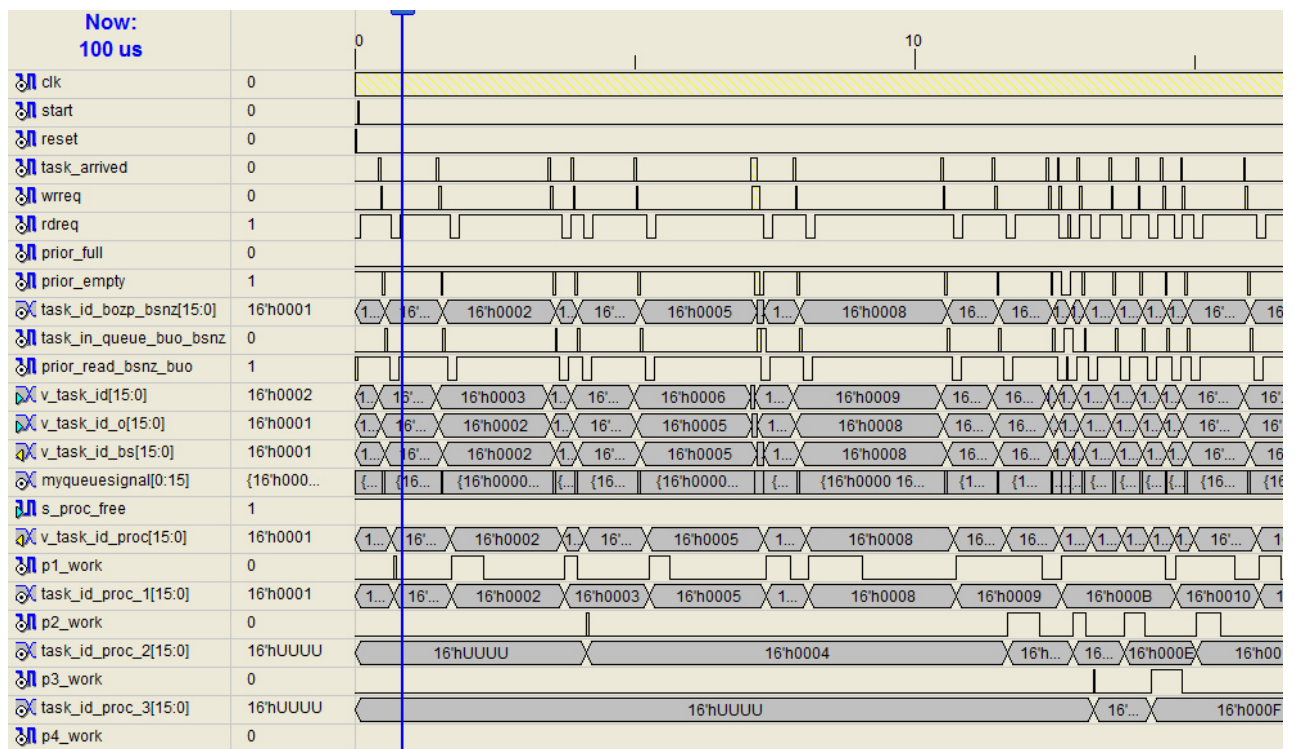


Рисунок 37 - временная диаграмма работы планировщика с глобальной очередью в условиях высокой загрузки

На рисунках 38 и 39 изображены временные диаграммы работы планировщика с локальной очередью в условиях средней ($p=0.5$) и высокой ($p=0.9$) загрузки соответственно.

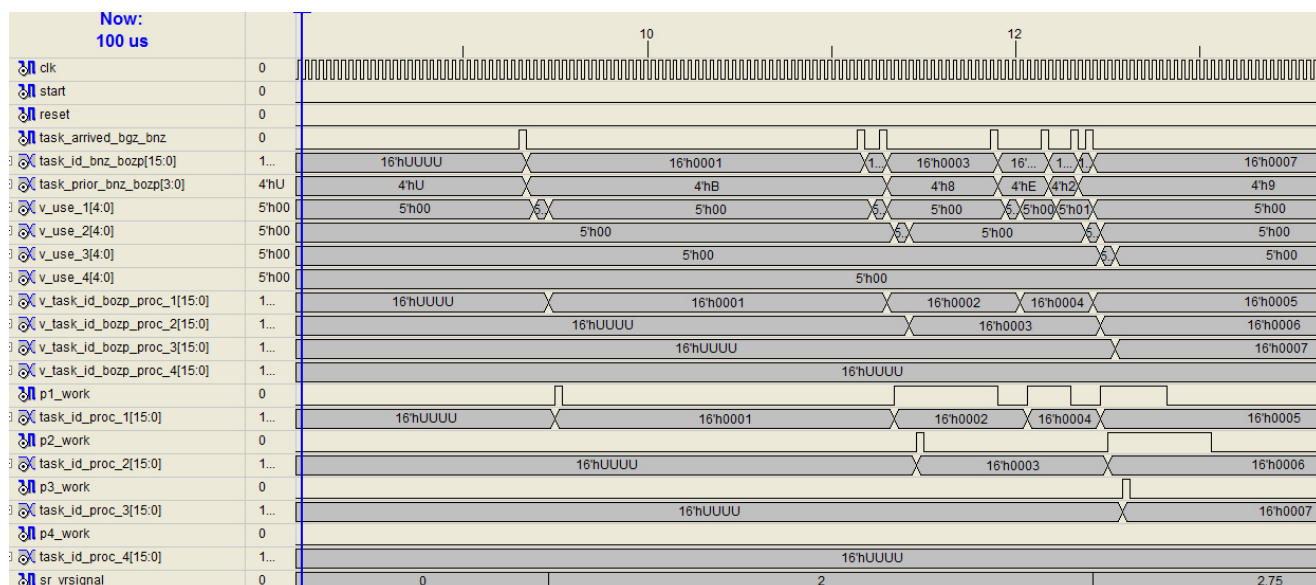


Рисунок 38 - временная диаграмма работы планировщика с локальной очередью в условиях средней загрузки

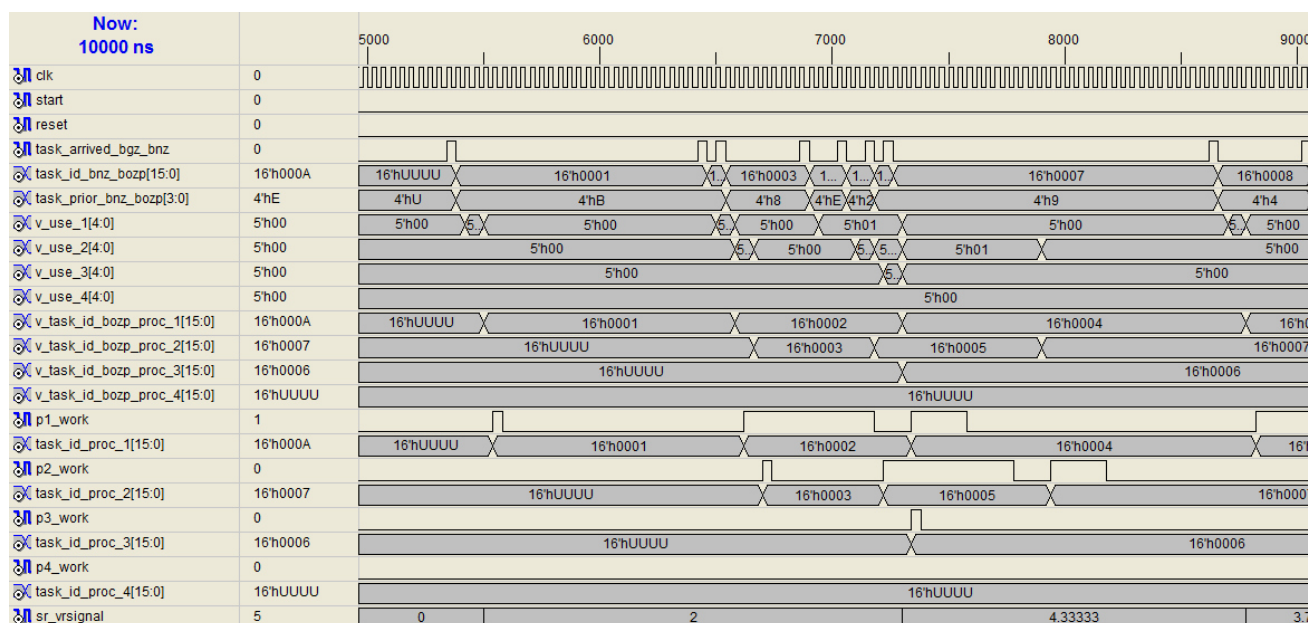


Рисунок 39 - временная диаграмма работы планировщика с локальной очередью в условиях высокой загрузки

После моделирования в среде Xilinx были измерены такие временные характеристики систем обслуживания, как латентность (в условиях

бесконфликтности), среднее время в очереди заявок каждого приоритета, время ожидания обслуживания задачи. Эксперименты проводились в условиях средней ($\rho_{цп} = 0,5$) и высокой ($\rho_{цп} = 0,9$) загрузки процессоров.

Вероятность поступления x задачи в планировщик задавалась экспоненциальными законом распределения по формуле [4]:

$$x = -\frac{1}{\alpha} \ln(1 - s) \quad (24)$$

Где s – значение равномерно распределённой в интервале $(0; 1)$ случайной величины, α – интенсивность входного потока задач.

Загрузка процессоров вычислялась по следующей формуле:

$$\rho_{цп} = \frac{\alpha}{k * \mu} \quad (25)$$

Где μ – интенсивность обработки задач одним процессором, α – интенсивность входного потока задач, k – количество процессоров в системе.

Время латентности в условиях бесконфликтности планировщиков вычислялось как время между возрастающим фронтом сигнала `task_arrived`, отвечающего за поступление первой задачи в планировщик, и возрастающим фронтом сигнала `p1_work`, говорящим о начале обработки первой задачи первым процессором.

На рисунке 40 показано, что латентность планировщика с глобальной очередью задач равна 7 тактов.

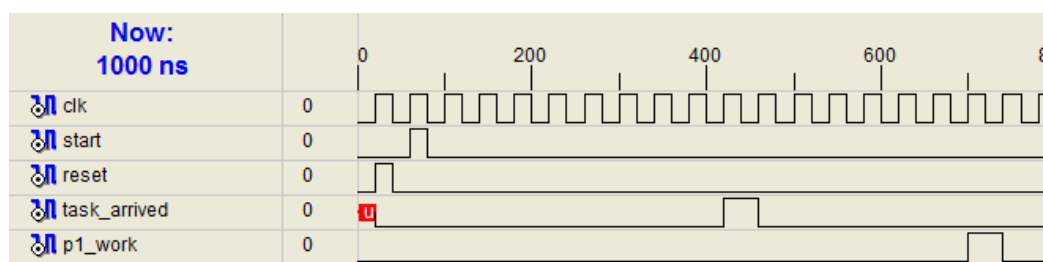


Рисунок 40 – латентность планировщика с глобальной очередью задач

На рисунке 41 показано, что латентность планировщика с локальными очередями задач равна 5 тактов.

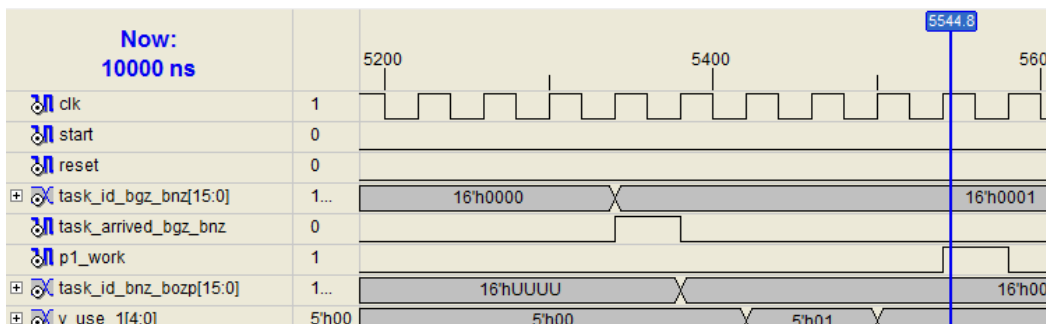


Рисунок 41 - латентность планировщика с локальными очередями задач

Для вычисления среднего времени пребывания задачи в приоритетной очереди для планировщика с глобальной очередью задач был создан сигнал `sr_vrsignal`, а также сигналы `sr_vrsignal0prior..sr_vrsignal15prior` для определения среднего времени нахождения заявок 0..15 приоритета в очереди соответственно.

В условиях средней загрузки процессоров ($\rho_{цп} = 0,5$) планировщика с глобальной очередью задач среднее время пребывания задачи в очереди составило 1,535 такта.

<code>sr_vrsignal</code>	1.53549
<code>sr_vrsignal_0prior</code>	1.375
<code>sr_vrsignal_1prior</code>	1.4127
<code>sr_vrsignal_2prior</code>	1.40426
<code>sr_vrsignal_3prior</code>	1.48611
<code>sr_vrsignal_4prior</code>	1.625
<code>sr_vrsignal_5prior</code>	1.51136
<code>sr_vrsignal_6prior</code>	1.35484
<code>sr_vrsignal_7prior</code>	1.31667
<code>sr_vrsignal_8prior</code>	1.63492
<code>sr_vrsignal_9prior</code>	1.50794
<code>sr_vrsignal_10prior</code>	1.4375
<code>sr_vrsignal_11prior</code>	1.60563
<code>sr_vrsignal_12prior</code>	1.89286
<code>sr_vrsignal_13prior</code>	1.77049
<code>sr_vrsignal_14prior</code>	1.6
<code>sr_vrsignal_15prior</code>	1.70588

Рисунок 42 – среднее время пребывания задач в очереди планировщика с глобальной очередью задач в условиях средней загрузки процессоров

В условиях высокой загрузки процессоров ($\rho_{цп} = 0,9$) планировщика с глобальной очередью задач среднее время пребывания задачи в очереди составило 2,64 такта.

sr_vrsignal	2.6402
sr_vrsignal_0prior	1.89855
sr_vrsignal_1prior	2.1068
sr_vrsignal_2prior	1.86667
sr_vrsignal_3prior	1.84874
sr_vrsignal_4prior	1.97938
sr_vrsignal_5prior	3.528
sr_vrsignal_6prior	2.25234
sr_vrsignal_7prior	2.17797
sr_vrsignal_8prior	2.07018
sr_vrsignal_9prior	3.24272
sr_vrsignal_10prior	2.88991
sr_vrsignal_11prior	3.04167
sr_vrsignal_12prior	4.02778
sr_vrsignal_13prior	3.65657
sr_vrsignal_14prior	2.3719
sr_vrsignal_15prior	3.45714

Рисунок 43 - среднее время пребывания задач в очереди планировщика с глобальной очередью задач в условиях высокой загрузки процессоров

Для вычисления среднего времени пребывания задачи в приоритетной очереди для планировщика с локальными очередями задач для каждой очереди были созданы сигналы sr_vrsignal, а также сигналы sr_vrsignal0prior... sr_vrsignal15prior для определения среднего времени нахождения заявок 0...15 приоритета в очереди соответственно. Для нахождения среднего времени пребывания задач в очереди следует найти среднее арифметическое между значениями сигналов sr_vrsignal для каждой очереди.

В условиях средней загрузки процессоров ($\rho_{\text{цп}} = 0,5$) планировщика с локальными очередями задач среднее время пребывания задачи в планировщике составило $(7,272 + 5,394 + 3,576 + 2)/4 = 4,56$ такта.

sr_vrsignal	7.272
sr_vrsignal	5.3945
sr_vrsignal	3.57692
sr_vrsignal	2

Рисунок 44 - среднее время пребывания задач в очереди планировщика с локальными очередями задач в условиях средней загрузки процессоров

В условиях высокой загрузки процессоров ($\rho_{\text{цп}} = 0,9$) планировщика с локальными очередями задач среднее время пребывания задачи в планировщике составило $(9,451 + 6,756 + 4,126 + 2,714)/4 = 5,761$ такта.

 sr_vrsignal	9.45143
 sr_vrsignal	6.75686
 sr_vrsignal	4.12687
 sr_vrsignal	2.71429

Рисунок 45 - среднее время пребывания задач в очереди планировщика с локальными очередями задач в условиях высокой загрузки процессоров

Для вычисления среднего времени обслуживания задач для планировщика с глобальной и локальными очередями задач были созданы сигналы sred_u_signal1..sred_u_signal4, показывающие среднее время нахождения задачи в планировщике после обработки 1..4м процессорами соответственно. Для нахождения среднего времени обслуживания задач для планировщика с глобальной очередью необходимо найти среднее арифметическое значений sred_u_signal1..sred_u_signal4.

В условиях средней загрузки процессоров ($\rho_{\text{цп}} = 0,5$) планировщика с глобальной очередью задач среднее время пребывания задачи в планировщике составило $(18,226 + 18,789 + 18,642 + 23,8)/4 = 19,864$ такта

 sred_u_signal_1	18.2261
 sred_u_signal_2	18.789
 sred_u_signal_3	18.6429
 sred_u_signal_4	23.8

Рисунок 46 - среднее время выполнения задач в планировщике с глобальной очередью задач в условиях средней загрузки процессоров

В условиях высокой загрузки процессоров ($\rho_{\text{цп}} = 0,9$) планировщика с глобальной очередью задач среднее время пребывания задачи в планировщике составило $(18,662 + 18,989 + 21,02 + 23,68)/4 = 20,587$ такта.

 sred_u_signal_1	18.6632
 sred_u_signal_2	18.9893
 sred_u_signal_3	21.0202
 sred_u_signal_4	23.68

Рисунок 47 - среднее время выполнения задач в планировщике с глобальной очередью задач в условиях высокой загрузки процессоров

В условиях средней загрузки процессоров ($\rho_{цп} = 0,5$) планировщика с локальными очередями задач среднее время пребывания задачи в планировщике составило $(21,269 + 20,38 + 20,692 + 9)/4 = 17,835$ такта.

 sred_u_signal_1	21.2693
 sred_u_signal_2	20.3807
 sred_u_signal_3	20.6923
 sred_u_signal_4	9

Рисунок 48 - среднее время выполнения задач в планировщике с локальными очередями задач в условиях средней загрузки процессоров

В условиях высокой загрузки процессоров ($\rho_{цп} = 0,9$) планировщика с локальными очередями задач среднее время пребывания задачи в планировщике составило $(23,237 + 20,941 + 19,074 + 21,928)/4 = 21,295$ такта.

 sred_u_signal_1	23.2371
 sred_u_signal_2	20.9412
 sred_u_signal_3	19.0746
 sred_u_signal_4	21.9286

Рисунок 49 - среднее время выполнения задач в планировщике с локальными очередями задач в условиях высокой загрузки процессоров

3.7 Выработка критериев сравнительной оценки планировщиков

На основе измерений, сделанных ранее, было выполнено сравнение основных характеристик промоделированных планировщиков с глобальной и локальными очередями: латентность в условиях бесконфликтности

планировщика с глобальной очередью больше латентности планировщика с локальными очередями на 40% (7 ед.вр. против 5 ед.вр.), а среднее время ожидания заявки в очереди в планировщике с глобальной очередью ниже, чем в планировщике с локальными очередями в 2,97 раза (1,535 ед.вр. против 4,56 ед.вр.) раза при средней загрузке и в 2,18 раза (2,64 ед.вр. против 5,761 ед.вр.) раза при высокой загрузке. Время обслуживания задач планировщика с глобальной очередью больше времени обслуживания задач планировщика с локальными очередями при средней загрузке на 11% (19,864 ед.вр. против 17,835 ед.вр.) и при высокой нагрузке процессоров меньше на 3% (20,587 ед.вр. против 21,295 ед.вр.) обслуживания задач локального планировщика.

Результаты замера характеристик для средней и высокой нагрузок показаны в таблицах 3 и 4 соответственно.

Таблица 3 – Временные характеристики при средней загрузке процессоров ($p=0.5$)

характеристика	глобальная очередь	локальная очередь
латентность (ед.вр.)	7,00	5,00
ср вр в очер	1,54	4,56
ср вр в очер 0 приор	1,38	4,31
ср вр в очер 1 приор	1,41	4,32
ср вр в очер 2 приор	1,40	4,36
ср вр в очер 3 приор	1,49	4,42
ср вр в очер 4 приор	1,63	4,46
ср вр в очер 5 приор	1,51	4,48
ср вр в очер 6 приор	1,35	4,52
ср вр в очер 7 приор	1,32	4,53
ср вр в очер 8 приор	1,63	4,59
ср вр в очер 9 приор	1,51	4,62
ср вр в очер 10 приор	1,44	4,63
ср вр в очер 11 приор	1,61	4,61
ср вр в очер 12 приор	1,89	4,65
ср вр в очер 13 приор	1,77	4,67
ср вр в очер 14 приор	1,60	4,66
ср вр в очер 15 приор	1,71	5,00
общее среднее время выполнения задачи в планировщике	19,864	17,835

Таблица 4 - Временные характеристики при высокой загрузке процессоров ($p=0.9$)

характеристика	глобальная очередь	локальная очередь
латентность (ед.вр.)	7	5
ср вр в очер	2,64	5,761
ср вр в очер 0 приор	1,898	5,221
ср вр в очер 1 приор	2,106	5,234
ср вр в очер 2 приор	1,866	5,242
ср вр в очер 3 приор	1,848	5,543
ср вр в очер 4 приор	1,979	5,435
ср вр в очер 5 приор	3,528	5,553
ср вр в очер 6 приор	2,252	5,645
ср вр в очер 7 приор	2,177	5,765
ср вр в очер 8 приор	2,07	5,783
ср вр в очер 9 приор	3,242	5,834
ср вр в очер 10 приор	2,889	5,926
ср вр в очер 11 приор	3,041	5,902
ср вр в очер 12 приор	4,027	6,112
ср вр в очер 13 приор	3,656	6,136
ср вр в очер 14 приор	2,371	6,263
ср вр в очер 15 приор	3,457	6,352
общее среднее время выполнения задачи в планировщике	20,587	21,295

Таким образом, по критерию производительности глобальный планировщик имеет лучшие показатели, так как в среднем имеет меньшее время ожидания задач в приоритетной очереди при примерно схожем общем времени выполнения задач. Кроме того, джиттер задач глобального планировщика значительно меньше джиттера задач локального планировщика.

Листинги программ всех модулей приведены в приложении.

3.8 Выводы

Разработаны функциональные модели диспетчеров с глобальной и локальными приоритетными очередями на языке VHDL. Модели были протестированы в условиях средней и высокой загрузки процессоров. Были измерены такие временные характеристики моделей как латентность в условиях бесконфликтности, среднее время ожидания задач в приоритетной очереди, а также среднее время ожидания задач каждого приоритета, среднее время нахождения задач в планировщике.

Модель планировщика с глобальной очередью задач показала лучшие результаты по производительности, чем модель планировщика с локальными очередями по критерию среднего времени отклика планировщика (задержки заявок в очереди). Джиттер задач глобального планировщика меньше джиттера задач локального планировщика. По критерию времени ответа системы (дедлайн) планировщики с глобальной и локальными очередями показали примерно равные результаты. Результат можно объяснить лучшей балансировкой загрузки процессоров, которая в глобальных планировщиках осуществляется автоматически и отсутствием балансировки в отдельном планировщике.

Заключение

Итогом проведенной работы являются следующие результаты: исследован материал, касающийся методов планирования планировщиков многопроцессорных систем, а именно:

- изучены основные характеристики операционных систем реального времени, такие как латентность, время переключения контекста, дедлайн, джиттер;
- рассмотрены существующие алгоритмы приоритетного планирования;
- изучены алгоритмы, основанные на приоритетных очередях задач;

Исследован вопрос моделирования алгоритмов приоритетного планирования на языке недетерминированных автоматов. Отдельно изучена проблема формализации алгоритмов управления параллельными процессами на основе использования механизма «рандеву», рассмотрена концептуальная модель «задачи о спящем парикмахере». Рассмотрены алгоритмы управления параллельными процессами в задаче «читатели-писатели», с использованием механизма монитора и согласующего кольцевого буфера. В итоге, на основе изученного материала, предложены реализации планировщиков задач с глобальной и локальными приоритетными очередями.

Дополнительно, выполнен анализ промоделированных планировщиков, выявлены их достоинства и недостатки, получены их основные временные характеристики, такие как латентность, среднее время отклика планировщика (ожидания заявки в очереди), среднее время ответа системы (дедлайн), а также определен джиттер задач в каждом из планировщиков. На его основе был сделан вывод о большей эффективности планировщика с глобальной очередью задач перед планировщиком с локальными очередями.

Общий вывод заключается в том, что в многопроцессорных системах жесткого реального времени следует применять преимущественно планировщики с глобальной очередью, т.к. по сравнению с планировщиками с локальными очередями при примерно равном дедлайне в системах с глобальной очередью джиттер задач намного меньше. В системах мягкого реального времени можно применять любой из них, так как по показателю времени ответа они показывают примерно одинаковые результаты.

Полученные результаты обладают практической и научной ценностью. Они могут быть применены при проектировании и разработке современных планировщиков задач многопроцессорных операционных систем. Дальнейшее развитие данной работы может заключаться в модификации данных планировщиков задач для систем реального времени и моделировании их на системах на кристалле фирмы Xilinx серии Zynq 7000.

Список используемых источников

1. Таненбаум Э. Бос Х. Современные операционные системы. 4-е изд. — СПб.: Питер, 2015. — 1120 с.
2. Таненбаум Э. Многоуровневая организация ЭВМ. – М.: Мир, 1979. – 547 с.
3. Вашкевич Н.П. Недетерминированные автоматы и их использование для реализации систем параллельной обработки информации / Н.П.Вашкевич, Р.А. Бикташев. – Пенза: Изд-во ПГУ, 2016. - 394с.
4. Вашкевич Н.П. Недетерминированные автоматы в проектировании систем параллельной обработки [текст]/ Н.П.Вашкевич: учеб. Пособие. - Пенза: изд-во Пенз.гос.ун-та, 2004. – 280 с.
5. Вашкевич Н.П. Аппаратная реализация функций синхронизации параллельных процессов при обращении к разделяемому ресурсу на основе ПЛИС [текст] / Н.П.Вашкевич, Р.А.Бикташев, Е.И.Гурин // Известия вузов. Поволжский регион. Технические науки. № 2, 2007, с.3-12.
6. Алиев Т.И. Основы моделирования дискретных систем. – СПб: СПбГУ ИТМО, 2009. – 363 с.
7. Бикташев Р. А., Королев Б.Д. Сравнительный анализ производительности средств управления задачами методом моделирования. // Сборник научных статей Всероссийской межвузовской научно-практической конференции. Под редакцией Л.Р. Фионовой. 2019 с.
8. Кларк Э.М. Верификация моделей программы: Model Checking [текст]. Перевод с англ./ Э.М.Кларк, О.Грамберг, Д.Пелед. – М.: МЦНМО, 2002. – 416 с.

9. Зотов, В. Ю. Проектирование цифровых устройств на основе ПЛИС фирмы Xilinx в САПРWebPACK ISE / В. Ю. Зотов. – М.: Горячая линия– Телеком, 2003. – 624 с.
- 10.Пранявичус Г.И. Модели и методы исследования вычислительных систем. – Вильнюс: Мокслас, 1982. – 227 с.
- 11.Т. Р. Baker, “A comparison of global and partitioned EDF schedulability tests for multiprocessors,” Florida State University, Tech. Rep. TR-051101, 2005.
- 12.К. Chakma / A Hierarchical Scheduling Approach for Symmetric Multiprocessing Based Real Time Systems on VxWorks / K. Chakma, S. Debbarma, N. Kar, N. Debbarma, and T. Debbarma // Lecture Notes on Software Engineering. - February 2013.-Vol. 1, No. 1, pp. 61-65.
- 13.Spartan-3E FPGA Starter Kit Board User Guide [Электронный ресурс]. – Режим доступа:
https://www.xilinx.com/support/documentation/boards_and_kits/ug-230.pdf

Приложение А

Модуль блока генерации задач

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.math_real.all;
entity block_gen_zadach is
PORT(
CLK                : IN STD_LOGIC;
start              : IN STD_LOGIC;
reset              : IN STD_LOGIC;
V_task_id          : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
prior_of_task_gen  : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
S_task_arrived     : OUT STD_LOGIC

);
end block_gen_zadach;
architecture block_gen_zadach_arch of block_gen_zadach is
begin
process(clk, reset)
variable start_enabled      : STD_LOGIC:='0';
variable reg_task_id        : STD_LOGIC_VECTOR(15 DOWNTO 0);

variable lambda :real :=0.07;
variable rand: real;--случайное число 0..1
variable seed1: positive :=52;--сиды для ГСЧ
variable seed2: positive :=56;--сиды для ГСЧ
```

```

variable randPrior: real;--случайное число 0..15
variable range_of_rand : real := 15.0;

variable interval_mejd_zad : real; -- интервал между задачами
variable temp : integer :=0;
begin
  if ((clk='1') and clk'event) then
    if start = '1' then
      start_enabled := '1';
    end if;
    if(temp = 0) then
      uniform(seed1, seed2, rand);--случ 0..1
      interval_mejd_zad := -1.0 / lambda * (LOG( 1.0 - rand));
    end if;
    if(temp >= integer(interval_mejd_zad)) then
      start_enabled := '1';
      temp:=0;
    else
      start_enabled := '0';
      temp := temp + 1;
    end if;
    if(start_enabled = '1') then
      reg_task_id := reg_task_id+'1';
      V_task_id <= reg_task_id;
      uniform(seed1, seed2, rand);--случ 0..1
      prior_of_task_gen <=
conv_std_logic_vector(integer(rand*range_of_rand),prior_of_task_ge
n'length);
      S_task_arrived <= '1';

```

```

else
    S_task_arrived <= '0';
end if;

if reset = '1' then
    V_task_id <= "0000000000000000";
    reg_task_id := "0000000000000000";
    prior_of_task_gen    <="0000";
    start_enabled := '0';
    S_task_arrived <= '0';
    uniform(seed1, seed2, rand);--случ 0..1
end if;

end if;

end process;

end block_gen_zadach_arch;

```

Приложение В

Модуль блока управления очередью планировщика с глобальной очередью

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity block_upr_ocher is
PORT(
CLK
      : IN STD_LOGIC;
V_task_id
      : IN STD_LOGIC_VECTOR(15 DOWNT0 0);
prior_of_task_gen
      : IN STD_LOGIC_VECTOR(3 DOWNT0 0);
S_task_arrived
      : IN STD_LOGIC;
S_prior_full
      : IN STD_LOGIC;
S_prior_empty
      : IN STD_LOGIC;
S_prior_read
      : IN STD_LOGIC;
reset
      : IN STD_LOGIC;
V_prior_data_o
      : OUT STD_LOGIC_VECTOR(15 DOWNT0 0);
S_wrreq
      : OUT STD_LOGIC;
S_rdreq
      : OUT STD_LOGIC;
S_task_in_queue
      : OUT STD_LOGIC;
prior_of_task
      : OUT STD_LOGIC_VECTOR(3 DOWNT0 0));
end block_upr_ocher;

architecture block_upr_ocher_arch of block_upr_ocher is
SIGNAL reg_task_id
      : STD_LOGIC_VECTOR(15 DOWNT0 0);
SIGNAL reg_prior_of_task
      : STD_LOGIC_VECTOR(3 DOWNT0 0);
SIGNAL S_task_arrived1
      : STD_LOGIC;
SIGNAL S_task_arrived2
      : STD_LOGIC;
```

```

begin
process(clk, reset) begin
    if (clk='1' and clk'event) then
        S_task_in_queue <= not S_prior_empty;
        reg_task_id <= V_task_id;
        reg_prior_of_task <= prior_of_task_gen;
        if S_task_arrived ='1' then
            if S_prior_full='0' THEN--если очередь не заполнена
                S_wrreq<='1';
            else
                S_wrreq<='0';
            end if;
            V_prior_data_o<=reg_task_id;
            prior_of_task <= reg_prior_of_task;
        else
            S_wrreq<='0';
        end if;
        IF S_prior_read='1' THEN
            S_rdreq<='1';
        ELSE
            S_rdreq<='0';
        END IF;
    end if;
    if (reset = '1') then
        V_prior_data_o  <="UUUUUUUUUUUUUUUUUU";
        prior_of_task   <="UUUU";
        S_wrreq         <= '0';
        S_rdreq         <= '0';
        S_task_in_queue <= '0';
    end if;

```

```
end process;  
end block_upr_ocher_arch;
```

Приложение С

Модуль блока приоритетной очереди планировщика с глобальной очередью

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity block_oher_zad_prior is
PORT(
CLK                : IN STD_LOGIC;
V_task_id_o        : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
prior_of_task      : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
S_wrreq            : IN STD_LOGIC;
S_rdreq            : IN STD_LOGIC;
reset              : IN STD_LOGIC;
S_prior_full       : OUT STD_LOGIC;
S_prior_empty      : OUT STD_LOGIC;
V_task_id_bs       : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
V_vrem_v_oher      : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
);
end block_oher_zad_prior;

architecture block_oher_zad_prior_arch of
block_oher_zad_prior is
    type queue is array (0 to 15) of std_logic_vector (15 downto
0);--временные хранилища приоритета и адреса (для сортировки)
    type priors_of_tasks is array (0 to 15) of std_logic_vector
(3 downto 0);
    type vrem_v_oher is array(0 to 15) of integer;
```

```

    SIGNAL MyQueueSignal: queue := (others => (others => '0'));-
-сигнал для отслеживания

    SIGNAL MyPriorsOfTasksSignal : priors_of_tasks := (others =>
(others => '0'));-сигнал для отслеживания

    SIGNAL Sr_vrSignal : real ;
    SIGNAL Sr_vrSignal_0Prior : real ;
    SIGNAL Sr_vrSignal_1Prior : real ;
    SIGNAL Sr_vrSignal_2Prior : real ;
    SIGNAL Sr_vrSignal_3Prior : real ;
    SIGNAL Sr_vrSignal_4Prior : real ;
    SIGNAL Sr_vrSignal_5Prior : real ;
    SIGNAL Sr_vrSignal_6Prior : real ;
    SIGNAL Sr_vrSignal_7Prior : real ;
    SIGNAL Sr_vrSignal_8Prior : real ;
    SIGNAL Sr_vrSignal_9Prior : real ;
    SIGNAL Sr_vrSignal_10Prior : real ;
    SIGNAL Sr_vrSignal_11Prior : real ;
    SIGNAL Sr_vrSignal_12Prior : real ;
    SIGNAL Sr_vrSignal_13Prior : real ;
    SIGNAL Sr_vrSignal_14Prior : real ;
    SIGNAL Sr_vrSignal_15Prior : real ;

begin

    process(clk, reset)

        variable tempQueueElement      : STD_LOGIC_VECTOR(15 DOWNT0
0);

        variable tempPriorElement      : STD_LOGIC_VECTOR(3 DOWNT0
0);

        variable num_elements_in_queue : integer :=0;--количество
элементов в очереди

```

```

variable MyQueue: queue := (others => (others => '0'));--
массив адресов задач

variable MyPriorsOfTasks : priors_of_tasks := (others =>
(others => '0'));--массив приоритетов задач

variable delayFromBSNZ :integer :=2;

variable delayTick:integer :=0;

variable tempVrem_v_ocher : integer:=0;

variable MyVrem_v_ocher : vrem_v_ocher :=(others => 0);--
массив времен заявок в очер

variable Sr_vr : real := 0.0; --сред число тактов заявки в
очер

variable sumVremVOcher : integer :=0; --суммарное время в
очереди выполненных задач

variable countVipolnZad : integer :=0; --суммарное число
выполненных задач

variable sumVremVOcher0Prior : integer :=0;--суммарное время
в очер зад приоритета 0..15

variable sumVremVOcher1Prior : integer :=0;
variable sumVremVOcher2Prior : integer :=0;
variable sumVremVOcher3Prior : integer :=0;
variable sumVremVOcher4Prior : integer :=0;
variable sumVremVOcher5Prior : integer :=0;
variable sumVremVOcher6Prior : integer :=0;
variable sumVremVOcher7Prior : integer :=0;
variable sumVremVOcher8Prior : integer :=0;
variable sumVremVOcher9Prior : integer :=0;
variable sumVremVOcher10Prior : integer :=0;
variable sumVremVOcher11Prior : integer :=0;
variable sumVremVOcher12Prior : integer :=0;
variable sumVremVOcher13Prior : integer :=0;
variable sumVremVOcher14Prior : integer :=0;

```

```

variable sumVremVOcher15Prior : integer :=0;

variable countVipolnZad0Prior : integer :=0;--количество
выполненных задач 0..15 приоритета
variable countVipolnZad1Prior : integer :=0;
variable countVipolnZad2Prior : integer :=0;
variable countVipolnZad3Prior : integer :=0;
variable countVipolnZad4Prior : integer :=0;
variable countVipolnZad5Prior : integer :=0;
variable countVipolnZad6Prior : integer :=0;
variable countVipolnZad7Prior : integer :=0;
variable countVipolnZad8Prior : integer :=0;
variable countVipolnZad9Prior : integer :=0;
variable countVipolnZad10Prior : integer :=0;
variable countVipolnZad11Prior : integer :=0;
variable countVipolnZad12Prior : integer :=0;
variable countVipolnZad13Prior : integer :=0;
variable countVipolnZad14Prior : integer :=0;
variable countVipolnZad15Prior : integer :=0;
variable Sr_vr0Prior : real := 0.0;--среднее число тактов
заявки 0..15 приор в очер
variable Sr_vr1Prior : real := 0.0;
variable Sr_vr2Prior : real := 0.0;
variable Sr_vr3Prior : real := 0.0;
variable Sr_vr4Prior : real := 0.0;
variable Sr_vr5Prior : real := 0.0;
variable Sr_vr6Prior : real := 0.0;
variable Sr_vr7Prior : real := 0.0;
variable Sr_vr8Prior : real := 0.0;
variable Sr_vr9Prior : real := 0.0;

```

```

variable Sr_vr10Prior : real := 0.0;
variable Sr_vr11Prior : real := 0.0;
variable Sr_vr12Prior : real := 0.0;
variable Sr_vr13Prior : real := 0.0;
variable Sr_vr14Prior : real := 0.0;
variable Sr_vr15Prior : real := 0.0;
procedure UvelVremVOcher is
begin
    for n in (queue'length-1) downto (queue'length -
num_elements_in_queue) loop
        MyVrem_v_ocher(n) := MyVrem_v_ocher(n) + 1;
    end loop;
end procedure;
procedure PereschitatSredVrVOcher is
begin
    sumVremVOcher := sumVremVOcher +
MyVrem_v_ocher(queue'length-1);--расчет сред врем в очер общий
    countVipolnZad:= countVipolnZad + 1;
    Sr_vr := real(sumVremVOcher) / real(countVipolnZad);
    if(MyPriorsOfTasks(queue'length-1)(3 downto 0) = "0000")
then--расчет сред врем в очер для задач приор 0..15
        sumVremVOcher0Prior := sumVremVOcher0Prior +
MyVrem_v_ocher(queue'length-1);
        countVipolnZad0prior:= countVipolnZad0Prior + 1;
        Sr_vr0Prior := real(sumVremVOcher0Prior) /
real(countVipolnZad0Prior);
        elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"0001") then
            sumVremVOcher1Prior := sumVremVOcher1Prior +
MyVrem_v_ocher(queue'length-1);
            countVipolnZad1prior:= countVipolnZad1Prior + 1;

```

```

        Sr_vr1Prior      :=      real (sumVremVOcher1Prior)      /
real (countVipolnZad1Prior);

        elsif (MyPriorsOfTasks (queue'length-1) (3      downto      0)      =
"0010") then

                sumVremVOcher2Prior      :=      sumVremVOcher2Prior      +
MyVrem_v_ocher (queue'length-1);

                countVipolnZad2prior:= countVipolnZad2Prior + 1;

                Sr_vr2Prior      :=      real (sumVremVOcher2Prior)      /
real (countVipolnZad2Prior);

        elsif (MyPriorsOfTasks (queue'length-1) (3      downto      0)      =
"0011") then

                sumVremVOcher3Prior      :=      sumVremVOcher3Prior      +
MyVrem_v_ocher (queue'length-1);

                countVipolnZad3prior:= countVipolnZad3Prior + 1;

                Sr_vr3Prior      :=      real (sumVremVOcher3Prior)      /
real (countVipolnZad3Prior);

        elsif (MyPriorsOfTasks (queue'length-1) (3      downto      0)      =
"0100") then

                sumVremVOcher4Prior      :=      sumVremVOcher4Prior      +
MyVrem_v_ocher (queue'length-1);

                countVipolnZad4prior:= countVipolnZad4Prior + 1;

                Sr_vr4Prior      :=      real (sumVremVOcher4Prior)      /
real (countVipolnZad4Prior);

        elsif (MyPriorsOfTasks (queue'length-1) (3      downto      0)      =
"0101") then

                sumVremVOcher5Prior      :=      sumVremVOcher5Prior      +
MyVrem_v_ocher (queue'length-1);

                countVipolnZad5prior:= countVipolnZad5Prior + 1;

                Sr_vr5Prior      :=      real (sumVremVOcher5Prior)      /
real (countVipolnZad5Prior);

```

```

        elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"0110") then
            sumVremVOcher6Prior := sumVremVOcher6Prior +
MyVrem_v_ocher(queue'length-1);
            countVipolnZad6prior:= countVipolnZad6Prior + 1;
            Sr_vr6Prior := real(sumVremVOcher6Prior) /
real(countVipolnZad6Prior);
        elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"0111") then
            sumVremVOcher7Prior := sumVremVOcher7Prior +
MyVrem_v_ocher(queue'length-1);
            countVipolnZad7prior:= countVipolnZad7Prior + 1;
            Sr_vr7Prior := real(sumVremVOcher7Prior) /
real(countVipolnZad7Prior);
        elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"1000") then
            sumVremVOcher8Prior := sumVremVOcher8Prior +
MyVrem_v_ocher(queue'length-1);
            countVipolnZad8prior:= countVipolnZad8Prior + 1;
            Sr_vr8Prior := real(sumVremVOcher8Prior) /
real(countVipolnZad8Prior);
        elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"1001") then
            sumVremVOcher9Prior := sumVremVOcher9Prior +
MyVrem_v_ocher(queue'length-1);
            countVipolnZad9prior:= countVipolnZad9Prior + 1;
            Sr_vr9Prior := real(sumVremVOcher9Prior) /
real(countVipolnZad9Prior);
        elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"1010") then

```

```

        sumVremVOcher10Prior      :=      sumVremVOcher10Prior      +
MyVrem_v_ocher(queue'length-1);
        countVipolnZad10Prior:= countVipolnZad10Prior + 1;
        Sr_vr10Prior      :=      real(sumVremVOcher10Prior)      /
real(countVipolnZad10Prior);
        elsif(MyPriorsOfTasks(queue'length-1)(3      downto      0) =
"1011") then
        sumVremVOcher11Prior      :=      sumVremVOcher11Prior      +
MyVrem_v_ocher(queue'length-1);
        countVipolnZad11Prior:= countVipolnZad11Prior + 1;
        Sr_vr11Prior      :=      real(sumVremVOcher11Prior)      /
real(countVipolnZad11Prior);
        elsif(MyPriorsOfTasks(queue'length-1)(3      downto      0) =
"1100") then
        sumVremVOcher12Prior      :=      sumVremVOcher12Prior      +
MyVrem_v_ocher(queue'length-1);
        countVipolnZad12Prior:= countVipolnZad12Prior + 1;
        Sr_vr12Prior      :=      real(sumVremVOcher12Prior)      /
real(countVipolnZad12Prior);
        elsif(MyPriorsOfTasks(queue'length-1)(3      downto      0) =
"1101") then
        sumVremVOcher13Prior      :=      sumVremVOcher13Prior      +
MyVrem_v_ocher(queue'length-1);
        countVipolnZad13Prior:= countVipolnZad13Prior + 1;
        Sr_vr13Prior      :=      real(sumVremVOcher13Prior)      /
real(countVipolnZad13Prior);
        elsif(MyPriorsOfTasks(queue'length-1)(3      downto      0) =
"1110") then
        sumVremVOcher14Prior      :=      sumVremVOcher14Prior      +
MyVrem_v_ocher(queue'length-1);
        countVipolnZad14Prior:= countVipolnZad14Prior + 1;

```

```

        Sr_vr14Prior      :=      real(sumVremVOcher14Prior)      /
real(countVipolnZad14Prior);

        elsif(MyPriorsOfTasks(queue'length-1)(3      downto      0)      =
"1111")then

                sumVremVOcher15Prior      :=      sumVremVOcher15Prior      +
MyVrem_v_ocher(queue'length-1);

                countVipolnZad15Prior:= countVipolnZad15Prior + 1;

                Sr_vr15Prior      :=      real(sumVremVOcher15Prior)      /
real(countVipolnZad15Prior);

        end if;

end procedure;

procedure Sdvig_ocher_vpravo is --смещает очередь вправо
begin
        for n in (queue'length-1)      downto      (queue'length      -
num_elements_in_queue + 1)loop
                MyQueue(n)(15      downto      0) := MyQueue(n-1)(15      downto      0);

                MyPriorsOfTasks(n)(3      downto      0) := MyPriorsOfTasks(n-1)(3
downto 0);

                MyVrem_v_ocher(n) := MyVrem_v_ocher(n-1);

        end loop;

        MyQueue(queue'length - num_elements_in_queue)(15      downto      0)
:="0000000000000000";--бывший последний элемент делаем пустым

        MyPriorsOfTasks(queue'length      -      num_elements_in_queue)(3
downto 0):="0000";

        MyVrem_v_ocher(queue'length - num_elements_in_queue):=0;

end procedure;

procedure Sort is --сортирует очередь по убыванию приоритета
(слева большее значение приоритетов, справа меньшее)
variable tempQueueElement      : STD_LOGIC_VECTOR(15      DOWNTO      0);
variable tempPriorElement      : STD_LOGIC_VECTOR(3      DOWNTO      0);
begin

```

```

    for k in (queue'length - num_elements_in_queue) to
(queue'length-1) loop
        if(MyPriorsOfTasks(k)(3 downto 0) < MyPriorsOfTasks(k+1)(3
downto 0)) then
            tempQueueElement(15 downto 0):=MyQueue(k)(15 downto 0);
            tempPriorElement(3 downto 0):=MyPriorsOfTasks(k)(3 downto
0);

            MyQueue(k)(15 downto 0):=MyQueue(k+1)(15 downto 0);
            MyPriorsOfTasks(k)(3 downto 0):=MyPriorsOfTasks(k+1)(3
downto 0);

            MyQueue(k+1)(15 downto 0):=tempQueueElement(15 downto 0);
            MyPriorsOfTasks(k+1)(3 downto 0):=tempPriorElement(3 downto
0);

        end if;
    end loop;
end procedure;
begin
    if(clk='1' and clk'event) then
        if(S_rdreq = '1' ) then --запись в блок синхронизации назн
задач
            if(num_elements_in_queue > 0) then --очередь не пуста
                if(delayTick > delayFromBSNZ)
                    V_task_id_bs <= MyQueue(queue'length-1)(15 downto 0);-
--записать последний элемент в блок синхр
                    V_vrem_v_ocher <=
conv_std_logic_vector(MyVrem_v_ocher(queue'length-
1),V_vrem_v_ocher'length);
                    PereschitatSredVrVOcher;--пересчитываем среднее время
в очереди
                    Sdvig_ocher_vpravo;--всю очередь сдвинуть вправо
                    num_elements_in_queue:=num_elements_in_queue-1;
                    delayTick:=0

```

```

        end if;
    end if;
end if;

if(S_wrreq = '1') then--чтение из блока управления очередью
    if(num_elements_in_queue < queue'length) then
        MyQueue(queue'length -1 - num_elements_in_queue)(15 downto
0):=V_task_id_o(15 downto 0); --добавить элемент слева
        MyPriorsOfTasks(queue'length -1 - num_elements_in_queue)(3
downto 0):=prior_of_task(3 downto 0);
        MyVrem_v_ocher(queue'length -1 -
num_elements_in_queue):=0;
        num_elements_in_queue:=num_elements_in_queue+1;
        Sort;
    end if;
end if;

if(num_elements_in_queue=queue'length) then
    S_prior_empty <= '0';
    S_prior_full <= '1';
end if;

if(num_elements_in_queue < queue'length and
num_elements_in_queue /= 0) then
    S_prior_empty <= '0';
    S_prior_full <= '0';
end if;

if(num_elements_in_queue=0) then
    S_prior_empty <= '1';
    S_prior_full <= '0';
end if;

UvelVremVOcher;--увеличить время в очереди для всех задач

```

```

delayTick:=delayTick+1;

MyQueueSignal(0)(15 downto 0) <= MyQueue(0)(15 downto 0);
MyQueueSignal(1)(15 downto 0) <= MyQueue(1)(15 downto 0);
MyQueueSignal(2)(15 downto 0) <= MyQueue(2)(15 downto 0);
MyQueueSignal(3)(15 downto 0) <= MyQueue(3)(15 downto 0);
MyQueueSignal(4)(15 downto 0) <= MyQueue(4)(15 downto 0);
MyQueueSignal(5)(15 downto 0) <= MyQueue(5)(15 downto 0);
MyQueueSignal(6)(15 downto 0) <= MyQueue(6)(15 downto 0);
MyQueueSignal(7)(15 downto 0) <= MyQueue(7)(15 downto 0);
MyQueueSignal(8)(15 downto 0) <= MyQueue(8)(15 downto 0);
MyQueueSignal(9)(15 downto 0) <= MyQueue(9)(15 downto 0);
MyQueueSignal(10)(15 downto 0) <= MyQueue(10)(15 downto 0);
MyQueueSignal(11)(15 downto 0) <= MyQueue(11)(15 downto 0);
MyQueueSignal(12)(15 downto 0) <= MyQueue(12)(15 downto 0);
MyQueueSignal(13)(15 downto 0) <= MyQueue(13)(15 downto 0);
MyQueueSignal(14)(15 downto 0) <= MyQueue(14)(15 downto 0);
MyQueueSignal(15)(15 downto 0) <= MyQueue(15)(15 downto 0);

MyPriorsOfTasksSignal(0)(3 downto 0) <=MyPriorsOfTasks(0)(3
downto 0);

MyPriorsOfTasksSignal(1)(3 downto 0) <=MyPriorsOfTasks(1)(3
downto 0);

MyPriorsOfTasksSignal(2)(3 downto 0) <=MyPriorsOfTasks(2)(3
downto 0);

MyPriorsOfTasksSignal(3)(3 downto 0) <=MyPriorsOfTasks(3)(3
downto 0);

MyPriorsOfTasksSignal(4)(3 downto 0) <=MyPriorsOfTasks(4)(3
downto 0);

MyPriorsOfTasksSignal(5)(3 downto 0) <=MyPriorsOfTasks(5)(3
downto 0);

MyPriorsOfTasksSignal(6)(3 downto 0) <=MyPriorsOfTasks(6)(3
downto 0);

```

```

    MyPriorsOfTasksSignal(7)(3 downto 0) <=MyPriorsOfTasks(7)(3
downto 0);

    MyPriorsOfTasksSignal(8)(3 downto 0) <=MyPriorsOfTasks(8)(3
downto 0);

    MyPriorsOfTasksSignal(9)(3 downto 0) <=MyPriorsOfTasks(9)(3
downto 0);

    MyPriorsOfTasksSignal(10)(3 downto 0)
<=MyPriorsOfTasks(10)(3 downto 0);

    MyPriorsOfTasksSignal(11)(3 downto 0)
<=MyPriorsOfTasks(11)(3 downto 0);

    MyPriorsOfTasksSignal(12)(3 downto 0)
<=MyPriorsOfTasks(12)(3 downto 0);

    MyPriorsOfTasksSignal(13)(3 downto 0)
<=MyPriorsOfTasks(13)(3 downto 0);

    MyPriorsOfTasksSignal(14)(3 downto 0)
<=MyPriorsOfTasks(14)(3 downto 0);

    Sr_vrSignal <= Sr_vr;

    Sr_vrSignal_0Prior <= Sr_vr0Prior;
    Sr_vrSignal_1Prior <= Sr_vr1Prior;
    Sr_vrSignal_2Prior <= Sr_vr2Prior;
    Sr_vrSignal_3Prior <= Sr_vr3Prior;
    Sr_vrSignal_4Prior <= Sr_vr4Prior;
    Sr_vrSignal_5Prior <= Sr_vr5Prior;
    Sr_vrSignal_6Prior <= Sr_vr6Prior;
    Sr_vrSignal_7Prior <= Sr_vr7Prior;
    Sr_vrSignal_8Prior <= Sr_vr8Prior;
    Sr_vrSignal_9Prior <= Sr_vr9Prior;
    Sr_vrSignal_10Prior <= Sr_vr10Prior;
    Sr_vrSignal_11Prior <= Sr_vr11Prior;
    Sr_vrSignal_12Prior <= Sr_vr12Prior;
    Sr_vrSignal_13Prior <= Sr_vr13Prior;

```

```

        Sr_vrSignal_14Prior <= Sr_vr14Prior;
        Sr_vrSignal_15Prior <= Sr_vr15Prior;
    end if;

    if(reset = '1') then
        num_elements_in_queue:=0;
        S_prior_full    <= '0';
        S_prior_empty <= '1';
        V_task_id_bs <="UUUUUUUUUUUUUUUUUU";
        delayTick:=3;
    end if;
end process;
end block_oher_zad_prior_arch;

```

Приложение D

Модуль блока синхронизации и назначения задач планировщика с глобальной очередью

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all ;
entity block_syncron_naznach_zadach is
PORT(
CLK                : IN STD_LOGIC;
V_task_id_bs       : IN STD_LOGIC_VECTOR(15 DOWNT0 0);
V_vrem_v_ocher     : IN STD_LOGIC_VECTOR(15 DOWNT0 0);--время в
очереди из БОЗП
S_task_in_queue    : IN STD_LOGIC;
S_proc_free        : IN STD_LOGIC;
S_proc_sel_1       : IN STD_LOGIC;
S_proc_sel_2       : IN STD_LOGIC;
S_proc_sel_3       : IN STD_LOGIC;
S_proc_sel_4       : IN STD_LOGIC;
S_proc_pzapr_1     : IN STD_LOGIC;
S_proc_pzapr_2     : IN STD_LOGIC;
S_proc_pzapr_3     : IN STD_LOGIC;
S_proc_pzapr_4     : IN STD_LOGIC;
S_proc_prinyal_1   : IN STD_LOGIC;
S_proc_prinyal_2   : IN STD_LOGIC;
S_proc_prinyal_3   : IN STD_LOGIC;
S_proc_prinyal_4   : IN STD_LOGIC;
```

```

reset                                : IN STD_LOGIC;
S_prior_read                        : OUT STD_LOGIC;
V_task_id_proc                     : OUT STD_LOGIC_VECTOR(15 DOWNT0 0);
S_task_ready_1                     : OUT STD_LOGIC;
S_task_ready_2                     : OUT STD_LOGIC;
S_task_ready_3                     : OUT STD_LOGIC;
S_task_ready_4                     : OUT STD_LOGIC;
S_proc_zapr_1                      : OUT STD_LOGIC;
S_proc_zapr_2                      : OUT STD_LOGIC;
S_proc_zapr_3                      : OUT STD_LOGIC;
S_proc_zapr_4                      : OUT STD_LOGIC;
V_vrem_v_ocher_proc                : OUT STD_LOGIC_VECTOR(15 DOWNT0 0)--
время в очер + в БЧЗ для процов
);
end block_synchron_naznach_zadach;

architecture block_synchron_naznach_zadach_arch of
block_synchron_naznach_zadach is
    signal reg_task_id : STD_LOGIC_VECTOR(15 DOWNT0 0);--хранит
адр задачи
begin
    process(clk, reset)
        variable zadacha_prinyata_v_obrabotku : STD_LOGIC := '0';
        variable zadacha_zaproshena : std_logic := '0';
        variable nomer_vzaimodeistv_processora: integer :=0;--номер
процессора, с которым в данный момент осуществляется обмен
запросами
        variable reg_vrem_v_ocher : STD_LOGIC_VECTOR(15 DOWNT0
0):="000000000000000000";--хранит время задачи в очер
        variable need_update_vrem : std_logic := '1';

```

```

begin

    if (clk='1' and clk'event) then

        if(S_proc_free = '1' and zadacha_prinyata_v_obrabotku =
'0') then--если есть своб процы и нет задач готовых обслужиться

            S_prior_read <='1'; --запросить задач

        else

            S_prior_read <='0'; --не запрашивать задач

        end if;

        if(S_task_in_queue = '1' ) then

            reg_task_id <= V_task_id_bs;

            if(need_update_vrem = '1') then

                reg_vrem_v_oher := V_vrem_v_oher;

                need_update_vrem := '0';

            end if;

            zadacha_prinyata_v_obrabotku := '1';

        end if;

        if(zadacha_prinyata_v_obrabotku = '1') then

            --запрашиваем выбранный процессор на принятие задачи, если
            есть своб процы И он выбран И еще НЕ дал подтв запроса И не принял
            задачу

            if(S_proc_free = '1' and S_proc_sel_1 = '1' and
S_proc_pzapr_1='0' and S_proc_prinyal_1 = '0' and
zadacha_zaproshena = '0' and nomer_vzaimodeistv_processora=0) then

                S_proc_zapr_1 <= '1';

                zadacha_zaproshena:='1';

                nomer_vzaimodeistv_processora:=1;

            elsif(S_proc_free = '1' and S_proc_sel_2 = '1' and
S_proc_pzapr_2='0' and S_proc_prinyal_2 = '0' and
zadacha_zaproshena = '0' and nomer_vzaimodeistv_processora=0) then

```

```

S_proc_zapr_2 <= '1';
zadacha_zaproshena:='1';
nomer_vzaimodeistv_processora:=2;
elseif(S_proc_free = '1' and S_proc_sel_3 = '1' and
S_proc_pzapr_3='0' and S_proc_prinyal_3 = '0' and
zadacha_zaproshena = '0' and nomer_vzaimodeistv_processora=0) then
S_proc_zapr_3 <= '1';
zadacha_zaproshena:='1';
nomer_vzaimodeistv_processora:=3;
elseif(S_proc_free = '1' and S_proc_sel_4 = '1' and
S_proc_pzapr_4='0' and S_proc_prinyal_4 = '0' and
zadacha_zaproshena = '0' and nomer_vzaimodeistv_processora=0) then
S_proc_zapr_4 <= '1';
zadacha_zaproshena:='1';
nomer_vzaimodeistv_processora:=4;
end if;
if(S_proc_pzapr_1='1' and nomer_vzaimodeistv_processora=1)
then
V_task_id_proc <= reg_task_id;
S_proc_zapr_1 <= '0';
S_task_ready_1 <='1';
elseif(S_proc_pzapr_2='1' and
nomer_vzaimodeistv_processora=2) then
V_task_id_proc <= reg_task_id;
S_proc_zapr_2 <= '0';
S_task_ready_2 <='1';
elseif(S_proc_pzapr_3='1' and
nomer_vzaimodeistv_processora=3) then
V_task_id_proc <= reg_task_id;
S_proc_zapr_3 <= '0';
S_task_ready_3 <='1';

```

```

        elsif(S_proc_pzapr_4='1'                                and
nomer_vzaimodeistv_processora=4) then
        V_task_id_proc <= reg_task_id;
        S_proc_zapr_4 <= '0';
        S_task_ready_4 <='1';
        end if;

        if(S_proc_prinyal_1='1' and nomer_vzaimodeistv_processora =
1 ) then
        S_task_ready_1 <='0';
        zadacha_zaproshena:='0';
        zadacha_prinyata_v_obrabotku:='0';--обнуляем переменную,
не дающую запрашивать новых задач
        nomer_vzaimodeistv_processora:=0;
        need_update_vrem := '1';
        elsif(S_proc_prinyal_2='1'                                and
nomer_vzaimodeistv_processora = 2 ) then
        S_task_ready_2 <='0';
        zadacha_zaproshena:='0';
        zadacha_prinyata_v_obrabotku:='0';
        nomer_vzaimodeistv_processora:=0;
        need_update_vrem := '1';
        elsif(S_proc_prinyal_3='1'                                and
nomer_vzaimodeistv_processora = 3 ) then
        S_task_ready_3 <='0';
        zadacha_zaproshena:='0';
        zadacha_prinyata_v_obrabotku:='0';
        nomer_vzaimodeistv_processora:=0;
        need_update_vrem := '1';
        elsif(S_proc_prinyal_4='1'                                and
nomer_vzaimodeistv_processora = 4 ) then

```

```

    S_task_ready_4 <='0';
    zadacha_zaproshena:='0';
    zadacha_prinyata_v_obrabotku:='0';
    nomer_vzaimodeistv_processora:=0;
    need_update_vrem := '1';

    end if;
end if;

reg_vrem_v_oher:=
conv_std_logic_vector(CONV_INTEGER(unsigned(reg_vrem_v_oher(15
downto 0))) + 1,reg_vrem_v_oher'length);

V_vrem_v_oher_proc <= reg_vrem_v_oher;
end if;
if (reset = '1') then
    S_prior_read<='0';
    S_task_ready_1<='0';
    S_task_ready_2<='0';
    S_task_ready_3<='0';
    S_task_ready_4<='0';
    S_proc_zapr_1 <='0';
    S_proc_zapr_2 <='0';
    S_proc_zapr_3<='0';
    S_proc_zapr_4<='0';
    zadacha_zaproshena:='0';
    zadacha_prinyata_v_obrabotku:='0';
    nomer_vzaimodeistv_processora:= 0;
    reg_vrem_v_oher := "0000000000000000";
end if;
end process;
end block_syncron_naznach_zadach_arch;

```


Приложение Е

Модуль блока выбора процессора планировщика с глобальной очередью

```
library IEEE;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity block_vibora_proc is
PORT (
CLK                : IN STD_LOGIC;
S_proc_free_1      : IN STD_LOGIC;
S_proc_free_2      : IN STD_LOGIC;
S_proc_free_3      : IN STD_LOGIC;
S_proc_free_4      : IN STD_LOGIC;
reset              : IN STD_LOGIC;
S_proc_free        : OUT STD_LOGIC;
S_proc_sel_1       : OUT STD_LOGIC;
S_proc_sel_2       : OUT STD_LOGIC;
S_proc_sel_3       : OUT STD_LOGIC;
S_proc_sel_4       : OUT STD_LOGIC);
end block_vibora_proc;

architecture block_vibora_proc_arch of block_vibora_proc is
begin
    process(clk, reset) begin
        if ((clk='1') and clk'event) then
            S_proc_free<= S_proc_free_1 OR S_proc_free_2 OR
S_proc_free_3 OR S_proc_free_4;-- OR S_proc_free_5 OR
S_proc_free_6 OR S_proc_free_7 OR S_proc_free_8;
```

```

    if(S_proc_free_1 = '1') then
        S_proc_sel_1 <= '1';
        S_proc_sel_2 <= '0';
        S_proc_sel_3 <= '0';
        S_proc_sel_4 <= '0';
    elsif(S_proc_free_2 = '1') then
        S_proc_sel_1 <= '0';
        S_proc_sel_2 <= '1';
        S_proc_sel_3 <= '0';
        S_proc_sel_4 <= '0';
    elsif(S_proc_free_3 = '1') then
        S_proc_sel_1 <= '0';
        S_proc_sel_2 <= '0';
        S_proc_sel_3 <= '1';
        S_proc_sel_4 <= '0';
    elsif(S_proc_free_4 = '1') then
        S_proc_sel_1 <= '0';
        S_proc_sel_2 <= '0';
        S_proc_sel_3 <= '0';
        S_proc_sel_4 <= '1';
    end if;
end if;

if(reset = '1') then
    S_proc_free    <= '1';
    S_proc_sel_1   <= '1';
    S_proc_sel_2   <= '0';
    S_proc_sel_3   <= '0';
    S_proc_sel_4   <= '0';
end if;

end process;

```

```
end block_vibora_proc_arch;
```

Приложение F

Модуль блока процессора

```
library IEEE;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.math_real.all;
entity proc1 is PORT(
CLK
      : IN STD_LOGIC;
V_task_id      : IN  STD_LOGIC_VECTOR(15 DOWNT0 0);
S_task_ready_1 : IN STD_LOGIC;
S_proc_zapr_1   : IN STD_LOGIC;
reset
      : IN STD_LOGIC;
V_vrem_v_ocher_proc : IN STD_LOGIC_VECTOR(15 DOWNT0 0);
S_proc_pzapr_1  : OUT STD_LOGIC;
S_proc_prinyal_1: OUT STD_LOGIC;
S_proc_free_1   : OUT STD_LOGIC);
end proc1;

architecture proc1_arch of proc1 is
signal P1_work      : STD_LOGIC := '0';
signal task_id_proc_1 : STD_LOGIC_VECTOR(15 DOWNT0 0);
signal sred_U_signal_1 : real := 0.0;
begin
process(clk, reset)
variable workTicks : integer := 50; -- сколько возрастающих
синхросигналов (+1) проц будет обраб задачу
variable thisWorkTick : integer := workTicks + 1;
variable seed1: positive := 24; -- сиды для ГСЧ
variable seed2: positive := 12; -- сиды для ГСЧ
variable rand: real; -- случайное число 0..1
```

```

variable mu : real := 1.0 / 52.0; --мат ожидание времени
исполнения задачи

variable sredU : real :=0.0;
variable sumU : integer :=0;
variable kol_obrab_zadach : integer :=0;
procedure calculateSredU is
begin
    sumU := sumU + CONV_INTEGER(unsigned(V_vrem_v_oher_proc(15
downto 0))) + thisWorkTick;
    kol_obrab_zadach := kol_obrab_zadach+1;
    sredU := real(sumU) / real(kol_obrab_zadach);
    sred_U_signal_1<=sredU;
end procedure;
begin
    if ((clk='1') and clk'event) then
        if (thisWorkTick < workTicks and P1_work = '1') then
            thisWorkTick := thisWorkTick+1;
        else
            if(S_proc_zapr_1 = '1') then --получили запрос от
диспетчера
                S_proc_free_1 <= '0'; --вышли из пула свободных процов
                S_proc_pzapr_1 <= '1'; -- подтвердили запрос
            end if;
            if(S_task_ready_1 = '1' or P1_work = '1') then
                if(thisWorkTick = workTicks) then
                    P1_work <= '0'; --закончить обработку задачи
                    S_proc_free_1 <= '1';--встать в пул свободных процов
                    S_proc_prinyal_1 <='0';
                    uniform(seed1, seed2, rand);
                    workTicks:= integer(-1.0 / mu * (LOG( 1.0 - rand)));

```

```

        thisWorkTick:= workTicks + 1;
    else
        task_id_proc_1 <= V_task_id; --сохранили адр задачи
        calculateSredU;
        S_proc_prinyal_1 <= '1';      --оповестили диспетчер о
получении адреса
        P1_work <= '1';
        S_proc_pzapr_1 <= '0';
        thisWorkTick:=0;
    end if;
end if;
end if;
end if;
if (reset = '1') then -- освободить процессор
    S_proc_pzapr_1 <= '0';
    S_proc_prinyal_1 <= '0';
    S_proc_free_1 <= '1';
    P1_work <='0';
    task_id_proc_1 <= "UUUUUUUUUUUUUUUUUU";
    uniform(seed1, seed2, rand);
    workTicks:= integer(-1.0 / mu * (LOG( 1.0 - rand)));--экспон
закон
    thisWorkTick:=workTicks + 1;
end if;
end process;
end procl_arch;

```

Приложение G

Модуль блока назначения задач планировщика с локальными очередями

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity block_nazn_zadach is
PORT(
CLK                : IN STD_LOGIC;
S_task_arrived     : IN STD_LOGIC;
V_task_id          : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
V_task_prior       : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
id                 : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
S_prior_full_1     : IN STD_LOGIC;
S_prior_full_2     : IN STD_LOGIC;
S_prior_full_3     : IN STD_LOGIC;
S_prior_full_4     : IN STD_LOGIC;
V_task_id_o        : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
V_task_prior_o     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
S_wrreq_1          : OUT STD_LOGIC;
S_wrreq_2          : OUT STD_LOGIC;
S_wrreq_3          : OUT STD_LOGIC;
S_wrreq_4          : OUT STD_LOGIC);
end block_nazn_zadach;
architecture block_nazn_zadach_arch of block_nazn_zadach is
signal AllQueuesFullSignal : STD_LOGIC;
begin
```

```

process (clk)
variable AllQueuesFull : std_logic := '0';
variable TickOfAnalyzeQueues : std_logic := '0';
begin
    if ((clk='1') and clk'event) then
        if (S_task_arrived = '1') then
            AllQueuesFull := S_prior_full_1 and
S_prior_full_2 and S_prior_full_3 and S_prior_full_4;
            if (AllQueuesFull = '0') then
                if (id="000") then
                    S_wrreq_1 <= '1';
                    S_wrreq_2 <= '0';
                    S_wrreq_3 <= '0';
                    S_wrreq_4 <= '0';
                elsif (id="001") then
                    S_wrreq_1 <= '0';
                    S_wrreq_2 <= '1';
                    S_wrreq_3 <= '0';
                    S_wrreq_4 <= '0';
                elsif (id="010") then
                    S_wrreq_1 <= '0';
                    S_wrreq_2 <= '0';
                    S_wrreq_3 <= '1';
                    S_wrreq_4 <= '0';
                elsif (id="011") then
                    S_wrreq_1 <= '0';
                    S_wrreq_2 <= '0';
                    S_wrreq_3 <= '0';
                    S_wrreq_4 <= '1';

```

```

        end if;

        V_task_id_o <= V_task_id;
        V_task_prior_o <= V_task_prior;
        AllQueuesFullSignal<=AllQueuesFull;

    else

        AllQueuesFullSignal<='1';

    end if;

    else--если задача не пришла не заставлять никакую
очередь писать

        S_wrreq_1 <= '0';
        S_wrreq_2 <= '0';
        S_wrreq_3 <= '0';
        S_wrreq_4 <= '0';

    end if;

    end if;

end process;

end block_nazn_zadach_arch;

```

Приложение Н

Модуль блока локального диспетчера с приоритетной очередью планировщика с локальными очередями

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity block_oher_zad_prior1 is
PORT(
CLK                : IN STD_LOGIC;
V_task_id_o        : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
V_task_prior_o     : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
S_wrreq_1          : IN STD_LOGIC;
reset              : IN STD_LOGIC;
S_proc_prinyal_1   : IN STD_LOGIC;
S_proc_free_1      : IN STD_LOGIC;
S_proc_pzapr_1     : IN STD_LOGIC;
S_prior_full_1     : OUT STD_LOGIC;
V_task_id_bs_1     : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
V_use_1            : OUT STD_LOGIC_VECTOR(4 DOWNTO 0);
V_vrem_v_oher     : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
S_proc_zapr_1      : OUT STD_LOGIC;
S_task_ready_1     : OUT STD_LOGIC
);
end block_oher_zad_prior1;

architecture block_oher_zad_prior1_arch of
block_oher_zad_prior1 is

```

```

    type queue is array (0 to 3) of std_logic_vector (15 downto
0);--временные хранилища приоритета и адреса (для сортировки)

    type priors_of_tasks is array (0 to 3) of std_logic_vector (3
downto 0);

    type vrem_v_ocher is array(0 to 3) of integer;

    SIGNAL MyQueueSignal: queue := (others => (others => '0'));-
-сигнал для отслеживания

    SIGNAL MyPriorsOfTasksSignal : priors_of_tasks := (others =>
(others => '0'));-сигнал для отслеживания

    SIGNAL Sr_vrSignal : real ;

    SIGNAL Sr_vrSignal_0Prior : real ;

    SIGNAL Sr_vrSignal_1Prior : real ;

    SIGNAL Sr_vrSignal_2Prior : real ;

    SIGNAL Sr_vrSignal_3Prior : real ;

    SIGNAL Sr_vrSignal_4Prior : real ;

    SIGNAL Sr_vrSignal_5Prior : real ;

    SIGNAL Sr_vrSignal_6Prior : real ;

    SIGNAL Sr_vrSignal_7Prior : real ;

    SIGNAL Sr_vrSignal_8Prior : real ;

    SIGNAL Sr_vrSignal_9Prior : real ;

    SIGNAL Sr_vrSignal_10Prior : real ;

    SIGNAL Sr_vrSignal_11Prior : real ;

    SIGNAL Sr_vrSignal_12Prior : real ;

    SIGNAL Sr_vrSignal_13Prior : real ;

    SIGNAL Sr_vrSignal_14Prior : real ;

    SIGNAL Sr_vrSignal_15Prior : real ;

begin

    process(clk, reset)

```

```

        variable tempQueueElement          : STD_LOGIC_VECTOR(15
DOWNT0 0);

        variable tempPriorElement          : STD_LOGIC_VECTOR(3 DOWNT0
0);

        variable tempVrem_v_ocher : integer:=0;
        variable num_elements_in_queue : integer range 0 to 4
:=0;--количество элементов в очереди

        variable MyQueue: queue := (others => (others => '0'));-
-массив адресов задач

        variable MyPriorsOfTasks : priors_of_tasks := (others =>
(others => '0'));-массив приоритетов задач

        variable MyVrem_v_ocher : vrem_v_ocher :=(others => 0);--
массив времен заявок в очер

        variable Sr_vr : real := 0.0; --сред врем заявки в очер

        variable sumVremVOcher : integer :=0; --суммарное время в
очереди выполненных задач

        variable countVipolnZad : integer :=0; --суммарное число
выполненных задач

        variable sumVremVOcher0Prior : integer :=0;--суммарное
время в очер зад приоритета 0..15

        variable sumVremVOcher1Prior : integer :=0;
        variable sumVremVOcher2Prior : integer :=0;
        variable sumVremVOcher3Prior : integer :=0;
        variable sumVremVOcher4Prior : integer :=0;
        variable sumVremVOcher5Prior : integer :=0;
        variable sumVremVOcher6Prior : integer :=0;
        variable sumVremVOcher7Prior : integer :=0;
        variable sumVremVOcher8Prior : integer :=0;
        variable sumVremVOcher9Prior : integer :=0;
        variable sumVremVOcher10Prior : integer :=0;
        variable sumVremVOcher11Prior : integer :=0;

```

```

variable sumVremVOcher12Prior : integer :=0;
variable sumVremVOcher13Prior : integer :=0;
variable sumVremVOcher14Prior : integer :=0;
variable sumVremVOcher15Prior : integer :=0;

variable countVipolnZad0Prior : integer :=0;--количество
выполненных задач 0..15 приоритета

variable countVipolnZad1Prior : integer :=0;
variable countVipolnZad2Prior : integer :=0;
variable countVipolnZad3Prior : integer :=0;
variable countVipolnZad4Prior : integer :=0;
variable countVipolnZad5Prior : integer :=0;
variable countVipolnZad6Prior : integer :=0;
variable countVipolnZad7Prior : integer :=0;
variable countVipolnZad8Prior : integer :=0;
variable countVipolnZad9Prior : integer :=0;
variable countVipolnZad10Prior : integer :=0;
variable countVipolnZad11Prior : integer :=0;
variable countVipolnZad12Prior : integer :=0;
variable countVipolnZad13Prior : integer :=0;
variable countVipolnZad14Prior : integer :=0;
variable countVipolnZad15Prior : integer :=0;

variable Sr_vr0Prior : real := 0.0;--среднее число тактов
заявки 0..15 приор в очер

variable Sr_vr1Prior : real := 0.0;
variable Sr_vr2Prior : real := 0.0;
variable Sr_vr3Prior : real := 0.0;
variable Sr_vr4Prior : real := 0.0;
variable Sr_vr5Prior : real := 0.0;
variable Sr_vr6Prior : real := 0.0;

```

```

variable Sr_vr7Prior : real := 0.0;
variable Sr_vr8Prior : real := 0.0;
variable Sr_vr9Prior : real := 0.0;
variable Sr_vr10Prior : real := 0.0;
variable Sr_vr11Prior : real := 0.0;
variable Sr_vr12Prior : real := 0.0;
variable Sr_vr13Prior : real := 0.0;
variable Sr_vr14Prior : real := 0.0;
variable Sr_vr15Prior : real := 0.0;
variable TaskWroteToProc : integer;
procedure UvelVremVOcher is
begin
    for n in (queue'length-1) downto (queue'length -
num_elements_in_queue) loop
        MyVrem_v_ocher(n) := MyVrem_v_ocher(n) + 1;
    end loop;
end procedure;
procedure PereschitatSredVrVOcher is
begin
    sumVremVOcher := sumVremVOcher +
MyVrem_v_ocher(queue'length-1);
    countVipolnZad:= countVipolnZad + 1;
    Sr_vr := real(sumVremVOcher) / real(countVipolnZad);
    if(MyPriorsOfTasks(queue'length-1)(3  downto  0) =
"0000") then--расчет сред врем в очер для задач приор 0..15
        sumVremVOcher0Prior := sumVremVOcher0Prior +
MyVrem_v_ocher(queue'length-1);
        countVipolnZad0prior:= countVipolnZad0Prior +
1;

```

```

        Sr_vr0Prior    :=    real(sumVremVOcher0Prior)    /
real(countVipolnZad0Prior);

        elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"0001") then

                sumVremVOcher1Prior    :=    sumVremVOcher1Prior    +
MyVrem_v_ocher(queue'length-1);

                countVipolnZad1prior:=    countVipolnZad1Prior    +
1;

                Sr_vr1Prior    :=    real(sumVremVOcher1Prior)    /
real(countVipolnZad1Prior);

        elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"0010") then

                sumVremVOcher2Prior    :=    sumVremVOcher2Prior    +
MyVrem_v_ocher(queue'length-1);

                countVipolnZad2prior:=    countVipolnZad2Prior    +
1;

                Sr_vr2Prior    :=    real(sumVremVOcher2Prior)    /
real(countVipolnZad2Prior);

        elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"0011") then

                sumVremVOcher3Prior    :=    sumVremVOcher3Prior    +
MyVrem_v_ocher(queue'length-1);

                countVipolnZad3prior:=    countVipolnZad3Prior    +
1;

                Sr_vr3Prior    :=    real(sumVremVOcher3Prior)    /
real(countVipolnZad3Prior);

        elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"0100") then

                sumVremVOcher4Prior    :=    sumVremVOcher4Prior    +
MyVrem_v_ocher(queue'length-1);

                countVipolnZad4prior:=    countVipolnZad4Prior    +
1;

```

```

        Sr_vr4Prior    :=    real(sumVremVOcher4Prior)    /
real(countVipolnZad4Prior);

        elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"0101") then

                sumVremVOcher5Prior    :=    sumVremVOcher5Prior    +
MyVrem_v_ocher(queue'length-1);

                countVipolnZad5prior:=    countVipolnZad5Prior    +
1;

                Sr_vr5Prior    :=    real(sumVremVOcher5Prior)    /
real(countVipolnZad5Prior);

        elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"0110") then

                sumVremVOcher6Prior    :=    sumVremVOcher6Prior    +
MyVrem_v_ocher(queue'length-1);

                countVipolnZad6prior:=    countVipolnZad6Prior    +
1;

                Sr_vr6Prior    :=    real(sumVremVOcher6Prior)    /
real(countVipolnZad6Prior);

        elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"0111") then

                sumVremVOcher7Prior    :=    sumVremVOcher7Prior    +
MyVrem_v_ocher(queue'length-1);

                countVipolnZad7prior:=    countVipolnZad7Prior    +
1;

                Sr_vr7Prior    :=    real(sumVremVOcher7Prior)    /
real(countVipolnZad7Prior);

        elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"1000") then

                sumVremVOcher8Prior    :=    sumVremVOcher8Prior    +
MyVrem_v_ocher(queue'length-1);

                countVipolnZad8prior:=    countVipolnZad8Prior    +
1;

```

```

        Sr_vr8Prior    :=    real(sumVremVOcher8Prior)    /
real(countVipolnZad8Prior);

        elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"1001") then

                sumVremVOcher9Prior    :=    sumVremVOcher9Prior    +
MyVrem_v_ocher(queue'length-1);

                countVipolnZad9prior:=    countVipolnZad9Prior    +
1;

                Sr_vr9Prior    :=    real(sumVremVOcher9Prior)    /
real(countVipolnZad9Prior);

        elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"1010") then

                sumVremVOcher10Prior    :=    sumVremVOcher10Prior    +
MyVrem_v_ocher(queue'length-1);

                countVipolnZad10Prior:=    countVipolnZad10Prior    +
1;

                Sr_vr10Prior    :=    real(sumVremVOcher10Prior)    /
real(countVipolnZad10Prior);

        elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"1011") then

                sumVremVOcher11Prior    :=    sumVremVOcher11Prior    +
MyVrem_v_ocher(queue'length-1);

                countVipolnZad11Prior:=    countVipolnZad11Prior    +
1;

                Sr_vr11Prior    :=    real(sumVremVOcher11Prior)    /
real(countVipolnZad11Prior);

        elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"1100") then

                sumVremVOcher12Prior    :=    sumVremVOcher12Prior    +
MyVrem_v_ocher(queue'length-1);

                countVipolnZad12Prior:=    countVipolnZad12Prior    +
1;

```

```

        Sr_vr12Prior := real(sumVremVOcher12Prior) /
real(countVipolnZad12Prior);

        elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"1101") then

            sumVremVOcher13Prior := sumVremVOcher13Prior +
MyVrem_v_ocher(queue'length-1);

            countVipolnZad13Prior:= countVipolnZad13Prior +
1;

            Sr_vr13Prior := real(sumVremVOcher13Prior) /
real(countVipolnZad13Prior);

            elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"1110") then

                sumVremVOcher14Prior := sumVremVOcher14Prior +
MyVrem_v_ocher(queue'length-1);

                countVipolnZad14Prior:= countVipolnZad14Prior +
1;

                Sr_vr14Prior := real(sumVremVOcher14Prior) /
real(countVipolnZad14Prior);

                elsif(MyPriorsOfTasks(queue'length-1)(3 downto 0) =
"1111") then

                    sumVremVOcher15Prior := sumVremVOcher15Prior +
MyVrem_v_ocher(queue'length-1);

                    countVipolnZad15Prior:= countVipolnZad15Prior +
1;

                    Sr_vr15Prior := real(sumVremVOcher15Prior) /
real(countVipolnZad15Prior);

                end if;

            end procedure;

        procedure Sdvig_ocher_vpravo is --смещает очередь вправо
begin

```

```

        for n in (queue'length-1) downto (queue'length -
num_elements_in_queue + 1) loop
            MyQueue(n) (15 downto 0) := MyQueue(n-1) (15
downto 0);

            MyPriorsOfTasks(n) (3 downto 0) :=
MyPriorsOfTasks(n-1) (3 downto 0);

            MyVrem_v_ocher(n) := MyVrem_v_ocher(n-1);
        end loop;

        MyQueue(queue'length - num_elements_in_queue) (15
downto 0) := "0000000000000000"; --бывший последний элемент делаем
пустым

        MyPriorsOfTasks(queue'length -
num_elements_in_queue) (3 downto 0) := "0000";

        MyVrem_v_ocher(queue'length -
num_elements_in_queue) := 0;

    end procedure;

    procedure Sort is --
сортирует очередь по убыванию приоритета (слева большее значение
приоритетов, справа меньшее)
        variable tempQueueElement : STD_LOGIC_VECTOR(15 DOWNT0 0);
        variable tempPriorElement : STD_LOGIC_VECTOR(3 DOWNT0 0);
        begin
            for k in (queue'length - num_elements_in_queue) to
(queue'length-1) loop
                if(MyPriorsOfTasks(k) (3 downto 0) < MyPriorsOfTasks(k+1) (3
downto 0)) then
                    tempQueueElement(15 downto 0) := MyQueue(k) (15 downto 0);
                    tempPriorElement(3 downto 0) := MyPriorsOfTasks(k) (3 downto
0);

                    MyQueue(k) (15 downto 0) := MyQueue(k+1) (15 downto 0);
                    MyPriorsOfTasks(k) (3 downto 0) := MyPriorsOfTasks(k+1) (3
downto 0);

                    MyQueue(k+1) (15 downto 0) := tempQueueElement(15 downto 0);

```

```

    MyPriorsOfTasks(k+1)(3 downto 0):=tempPriorElement(3 downto
0);
    end if;
    end loop;
end procedure;
begin
    if(clk='1' and clk'event) then
        if(S_wrreq_1 = '1') then--чтение из блока назн
задач
            if(num_elements_in_queue < queue'length)
then
                MyQueue(queue'length -1 -
num_elements_in_queue)(15 downto 0):=V_task_id_o(15 downto 0); --
добавить элемент слева
                MyPriorsOfTasks(queue'length -1 -
num_elements_in_queue)(3 downto 0):=V_task_prior_o(3 downto 0);--
и его приоритет
                MyVrem_v_ocher(queue'length -1 -
num_elements_in_queue):=0;
                num_elements_in_queue:=num_elements_in_queue+1;
                Sort;-- отсортировать очередь
            end if;
        end if;
        if(S_proc_free_1 = '1' ) then
            if(num_elements_in_queue > 0) then
                S_proc_zapr_1 <='1';
                S_task_ready_1 <='0';
            end if;
        end if;
    end if;
end if;

```

```

        if (S_proc_pzapr_1 = '1' and
num_elements_in_queue > 0 and TaskWroteToProc > 0) then
            V_task_id_bs_1 <= MyQueue(queue'length-
1)(15 downto 0);--записать последний элемент в проц
            V_vrem_v_ocher <=
conv_std_logic_vector(MyVrem_v_ocher(queue'length-
1),V_vrem_v_ocher'length);
            PereschitatSredVrVOcher;--пересчитываем
среднее время в очереди
            Sdvig_ocher_vpravo;--всю очередь сдвинуть
вправо

num_elements_in_queue:=num_elements_in_queue-1;
            S_task_ready_1 <= '1';
            S_proc_zapr_1 <='0';
            TaskWroteToProc:=0;
        else
            TaskWroteToProc:= TaskWroteToProc + 1;
        end if;
        if (S_proc_prinyal_1= '1') then
            S_task_ready_1 <= '0';
            S_proc_zapr_1 <='0'; -- на всяк случ
        end if;
        if (num_elements_in_queue=queue'length) then
            S_prior_full_1 <= '1';
        else
            S_prior_full_1 <= '0';
        end if;

        V_use_1
<=conv_std_logic_vector(num_elements_in_queue,V_use_1'length) ; --
сообщаем кол во эл тов в очер

```

```

UvelVremVOcher;--увеличить время в очереди для всех
задач

--для проверки сигналов на симуляции
MyQueueSignal(0)(15 downto 0) <= MyQueue(0)(15
downto 0);

MyQueueSignal(1)(15 downto 0) <= MyQueue(1)(15
downto 0);

MyQueueSignal(2)(15 downto 0) <= MyQueue(2)(15
downto 0);

MyQueueSignal(3)(15 downto 0) <= MyQueue(3)(15
downto 0);

Sr_vrSignal <= Sr_vr;
Sr_vrSignal_0Prior <= Sr_vr0Prior;
Sr_vrSignal_1Prior <= Sr_vr1Prior;
Sr_vrSignal_2Prior <= Sr_vr2Prior;
Sr_vrSignal_3Prior <= Sr_vr3Prior;
Sr_vrSignal_4Prior <= Sr_vr4Prior;
Sr_vrSignal_5Prior <= Sr_vr5Prior;
Sr_vrSignal_6Prior <= Sr_vr6Prior;
Sr_vrSignal_7Prior <= Sr_vr7Prior;
Sr_vrSignal_8Prior <= Sr_vr8Prior;
Sr_vrSignal_9Prior <= Sr_vr9Prior;
Sr_vrSignal_10Prior <= Sr_vr10Prior;
Sr_vrSignal_11Prior <= Sr_vr11Prior;
Sr_vrSignal_12Prior <= Sr_vr12Prior;
Sr_vrSignal_13Prior <= Sr_vr13Prior;
Sr_vrSignal_14Prior <= Sr_vr14Prior;
Sr_vrSignal_15Prior <= Sr_vr15Prior;
end if;

if(reset = '1') then

```

```

        num_elements_in_queue:=0;
        S_prior_full_1  <= '0';
        V_task_id_bs_1 <="UUUUUUUUUUUUUUUUUU";
        S_task_ready_1 <='0';
        S_proc_zapr_1  <= '0';
        V_use_1 <="00000";
        TaskWroteToProc:=0;
        Sr_vr :=0.0;
    end if;
end process;
end block_oher_zad_prior1_arch;

```

Приложение I

Модуль выбора очереди планировщика с локальными очередями

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
entity block_vibora_ocher is
PORT(
CLK : IN STD_LOGIC;
V_use_1 : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
V_use_2 : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
V_use_3 : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
V_use_4 : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
id : OUT STD_LOGIC_VECTOR(2 DOWNTO 0)
);
end block_vibora_ocher;
architecture block_vibora_ocher_arch of block_vibora_ocher is
type usesArray is array (0 to 3) of integer;
begin
process (CLK)
variable myArray : usesArray := (others => 0);--массив
количеств элементов в очередях
variable minUseValue : integer := 4;
variable minUse: integer := 0;
begin
myArray(0) := CONV_INTEGER(unsigned(V_use_1(4 downto
0))) );
```

```

myArray(1)    := CONV_INTEGER(unsigned(V_use_2(4 downto
0)));
myArray(2)    := CONV_INTEGER(unsigned(V_use_3(4 downto
0)));
myArray(3)    := CONV_INTEGER(unsigned(V_use_4(4 downto
0)));

for i in 0 to myArray'length-1 loop--поиск минимального
количества элементов в очередях
    if(myArray(i) < minUseValue) then
        minUseValue := myArray(i);
        minUse := i;
    end if;
end loop;
id <=conv_std_logic_vector(minUse,id'length);
minUseValue:=4;
minUse:=0;
end process;
end block_vibora_oher_arch;

```

Приложение J

Презентация

Моделирование ядра многопроцессорных операционных систем с аппаратной поддержкой

Выполнил студент группы 18ВВм1 Королев Б. Д.
Научный руководитель: к.т.н., профессор Бикташев Р. А.

Объект исследования – алгоритмы планирования и диспетчеризации задач в многопроцессорных вычислительных системах.

Предмет исследования – математическое и функционально-логическое моделирование планировщиков/диспетчеров задач в составе ядра многопроцессорных операционных систем с использованием языка недетерминированных автоматов (НДА), систем массового обслуживания и языка моделирования аппаратуры VHDL.

Цель данной работы – получение временных характеристик и сравнительная оценка планировщиков/диспетчеров задач с аппаратной реализацией.

С целью достижения поставленной цели в диссертационной работе решаются следующие задачи:

- 1) разработка автоматной модели на основе языка НДА и на её основе VHDL-модели приоритетного планировщика/диспетчера задач и оценка характеристик аппаратной реализации;
- 2) разработка имитационной модели приоритетного планировщика/диспетчера задач для получения его вероятностно-временных характеристик;
- 3) разработка функционально - логической модели планировщика/диспетчера задач для оценки возможности реализации аппаратными средствами;
- 4) проведение вычислительных экспериментов с использованием разработанных моделей.

Научная новизна работы состоит в следующем:

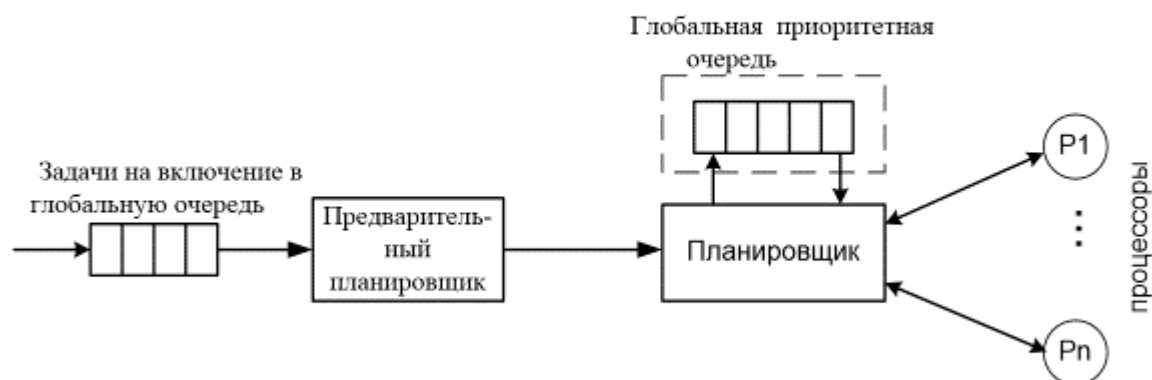
- 1) разработаны на основе языка НДА автоматные модели приоритетных планировщиков/диспетчеров задач с разделением времени с локальными и глобальными очередями в составе многопроцессорной системы.
- 2) созданы на основе НДА модели VHDL – программы аппаратной реализации планировщиков/диспетчеров задач с разделением времени с локальными и глобальными очередями в составе многопроцессорной системы и произведена оценка характеристик этих моделей;
- 3) разработан программный модуль сбора статистики в составе VHDL – программы для оценки вероятностно-временных характеристик подсистемы «диспетчер задач – процессорный узел».

Практическая значимость работы.

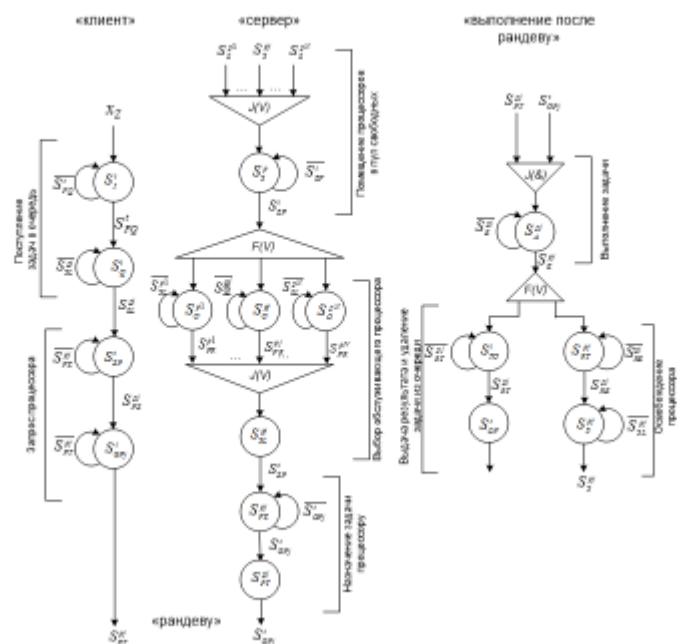
Применение предложенных аналитической и программной моделей обеспечивает сравнительную оценку временных характеристик планировщика/диспетчера задач на этапе эскизного проектирования, без проведения дорогостоящих физических экспериментов.

Реализация и внедрение результатов работы. Методы моделирования диспетчеров задач многопроцессорных систем будут применены в учебном процессе кафедры «Вычислительная техника» Пензенского государственного университета при осуществлении основной образовательной программы по дисциплинам «Вычислительные системы» для магистрантов, которые обучаются по направлению «Информатика и вычислительная техника».

Планировщик с глобальной очередью задач



Граф НДА алгоритма диспетчеризации задач приоритетного планировщика с глобальной очередью в многопроцессорной системе



Структура приоритетной очереди задач



Добавление новой задачи в приоритетную очередь

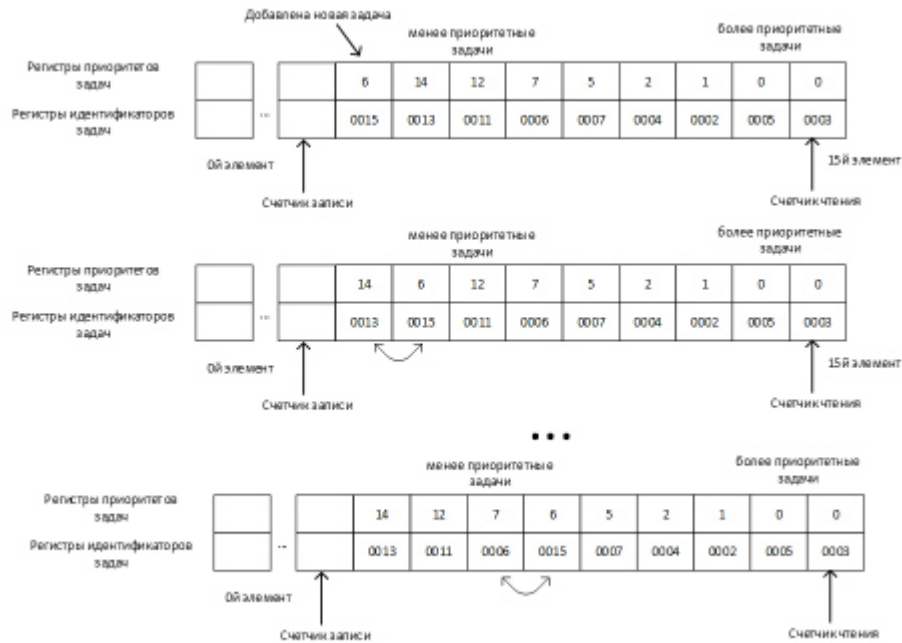
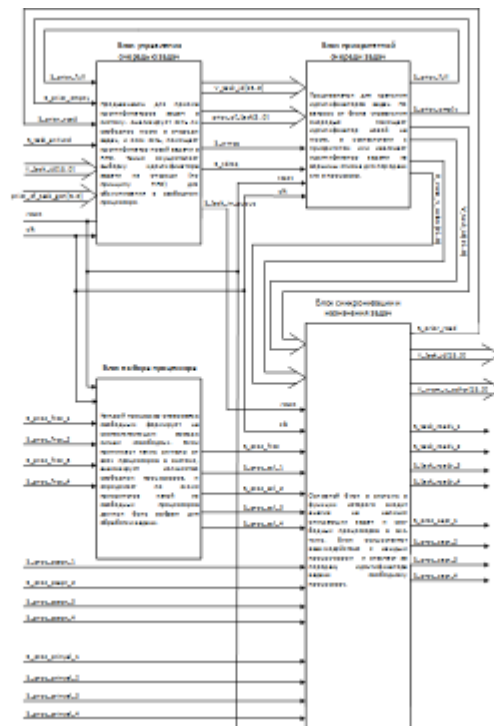
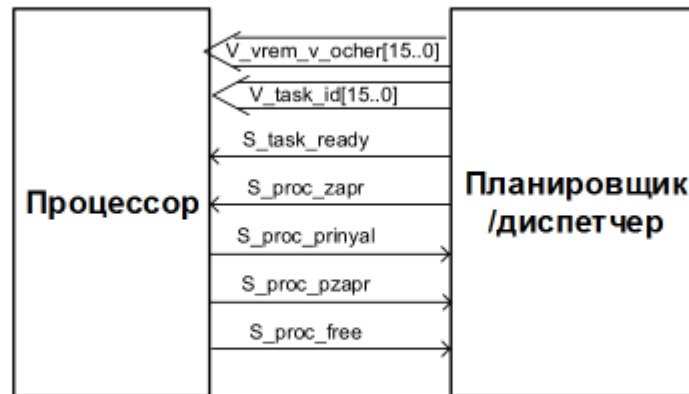


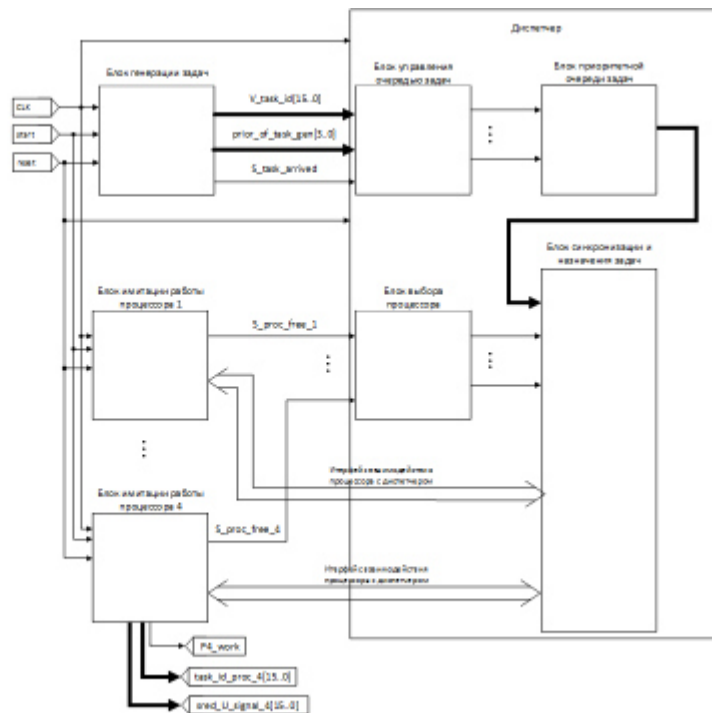
Схема созданной модели планировщика с глобальной приоритетной очередью в среде Xilinx



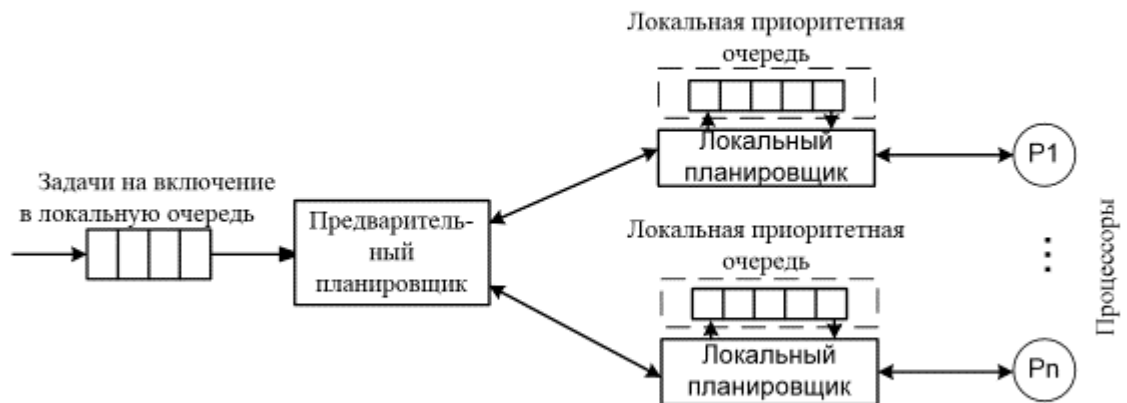
Интерфейс взаимодействия процессора с диспетчером в планировщике с глобальной очередью задач



Общий вид структурной схемы с глобальной приоритетной очередью



Планировщик с локальной очередью задач



Граф НДА алгоритма диспетчеризации задач приоритетного планировщика с локальной очередью в многопроцессорной системе

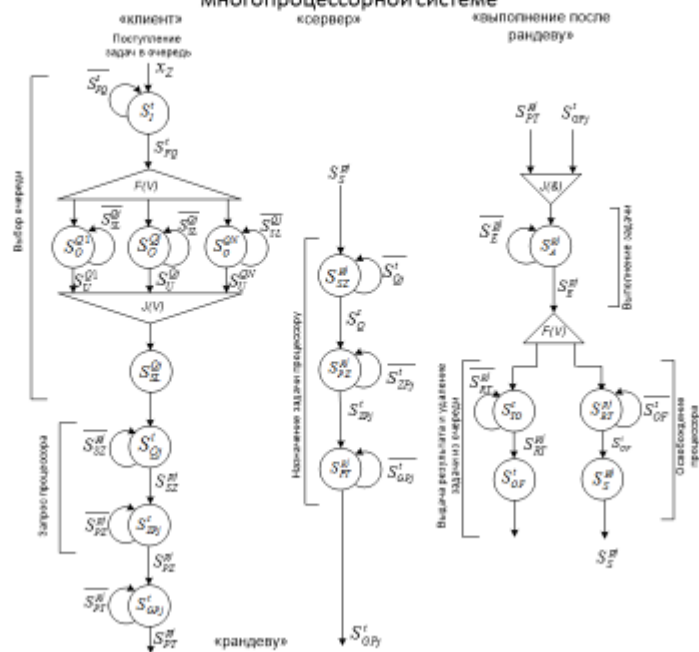
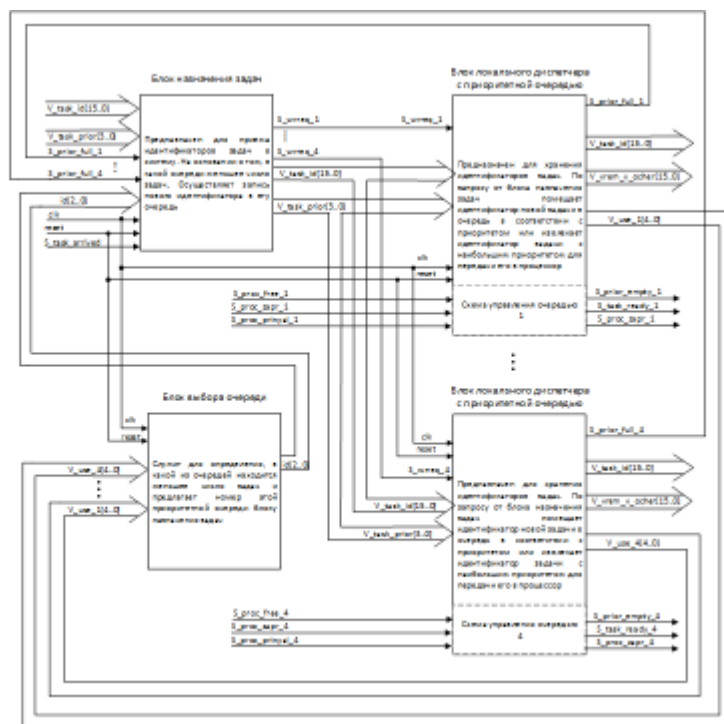


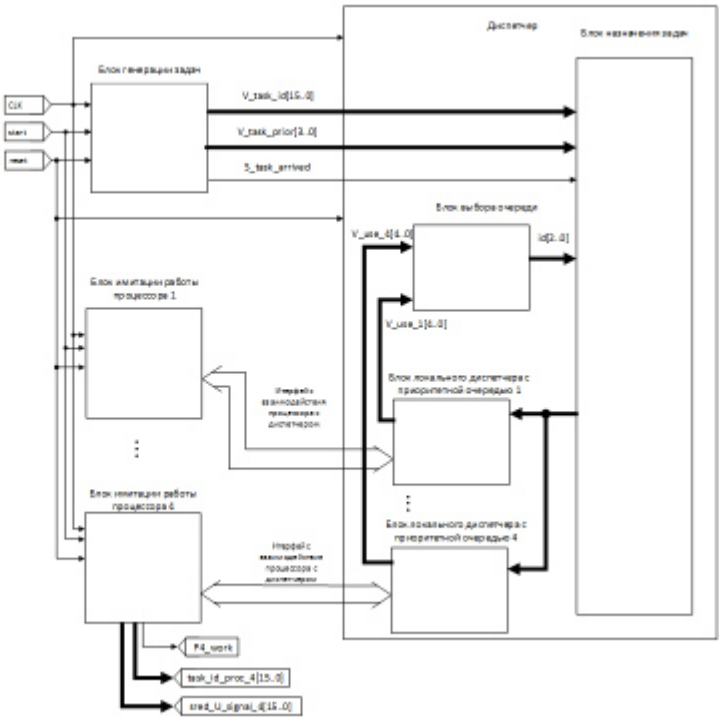
Схема созданной модели планировщика с локальными приоритетными очередями в среде Xilinx



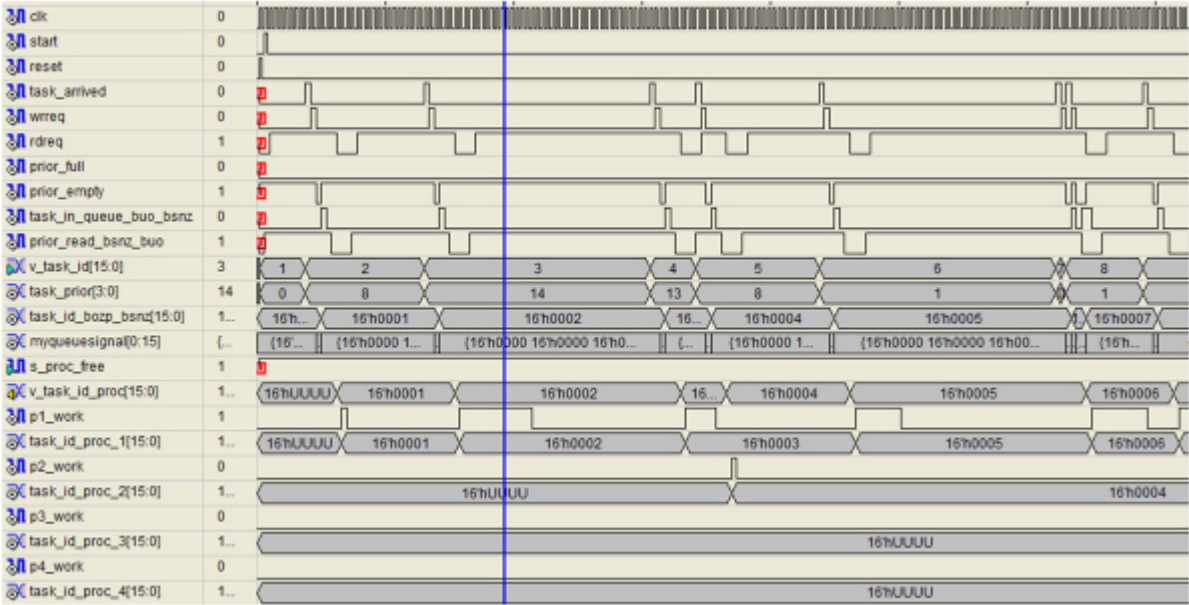
Интерфейс взаимодействия процессора с диспетчером в планировщике с локальными очередями задач



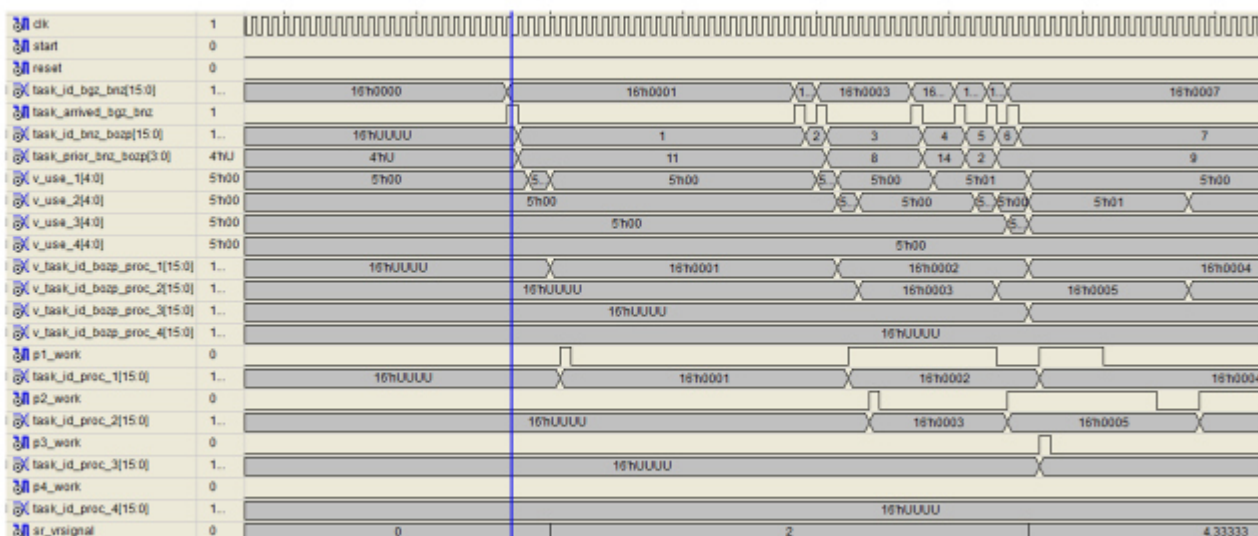
Общий вид структурной
схемы с локальными
приоритетными
очередями



Временная диаграмма работы планировщика с глобальной очередью задач



Временная диаграмма работы планировщика с локальными очередями задач



Временные
характеристики при
средней загрузке
процессоров ($\rho=0.5$)

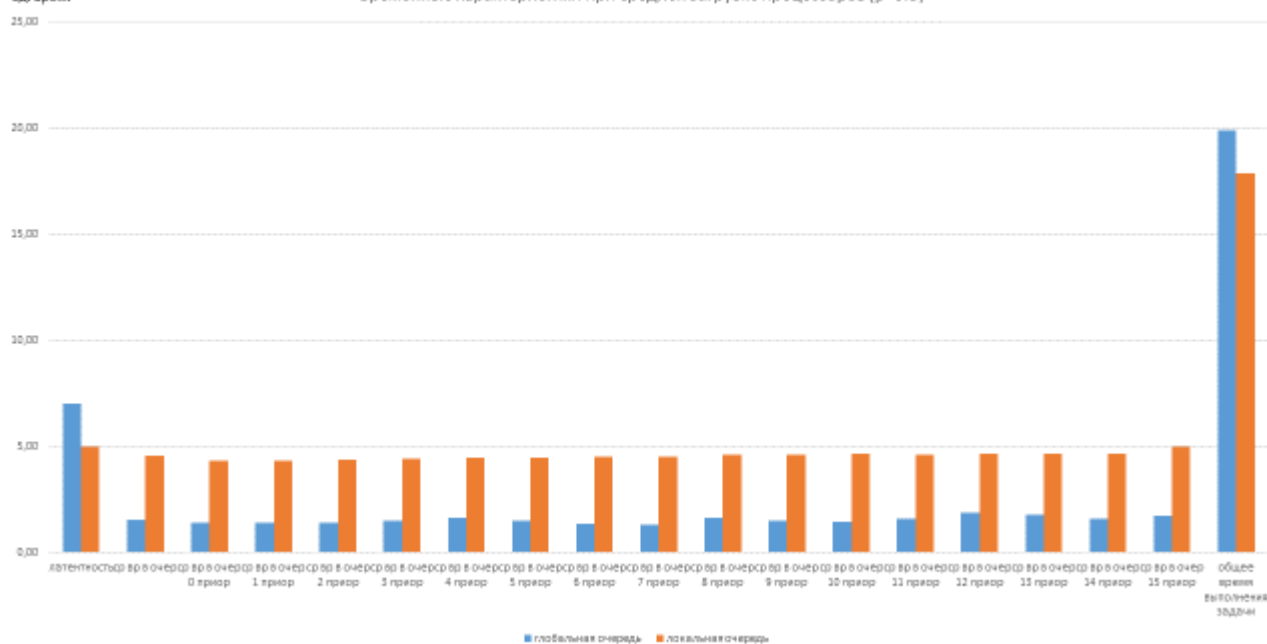
	глобальная очередь	локальная очередь
латентность (ед.вр.)	7,00	5,00
ср вр в очер	1,54	4,56
ср вр в очер 0 приор	1,38	4,31
ср вр в очер 1 приор	1,41	4,32
ср вр в очер 2 приор	1,40	4,36
ср вр в очер 3 приор	1,49	4,42
ср вр в очер 4 приор	1,63	4,46
ср вр в очер 5 приор	1,51	4,48
ср вр в очер 6 приор	1,35	4,52
ср вр в очер 7 приор	1,32	4,53
ср вр в очер 8 приор	1,63	4,59
ср вр в очер 9 приор	1,51	4,62
ср вр в очер 10 приор	1,44	4,63
ср вр в очер 11 приор	1,61	4,61
ср вр в очер 12 приор	1,89	4,65
ср вр в очер 13 приор	1,77	4,67
ср вр в очер 14 приор	1,60	4,66
ср вр в очер 15 приор	1,71	5,00
общее среднее время выполнения задачи в планировщике	19,864	17,835

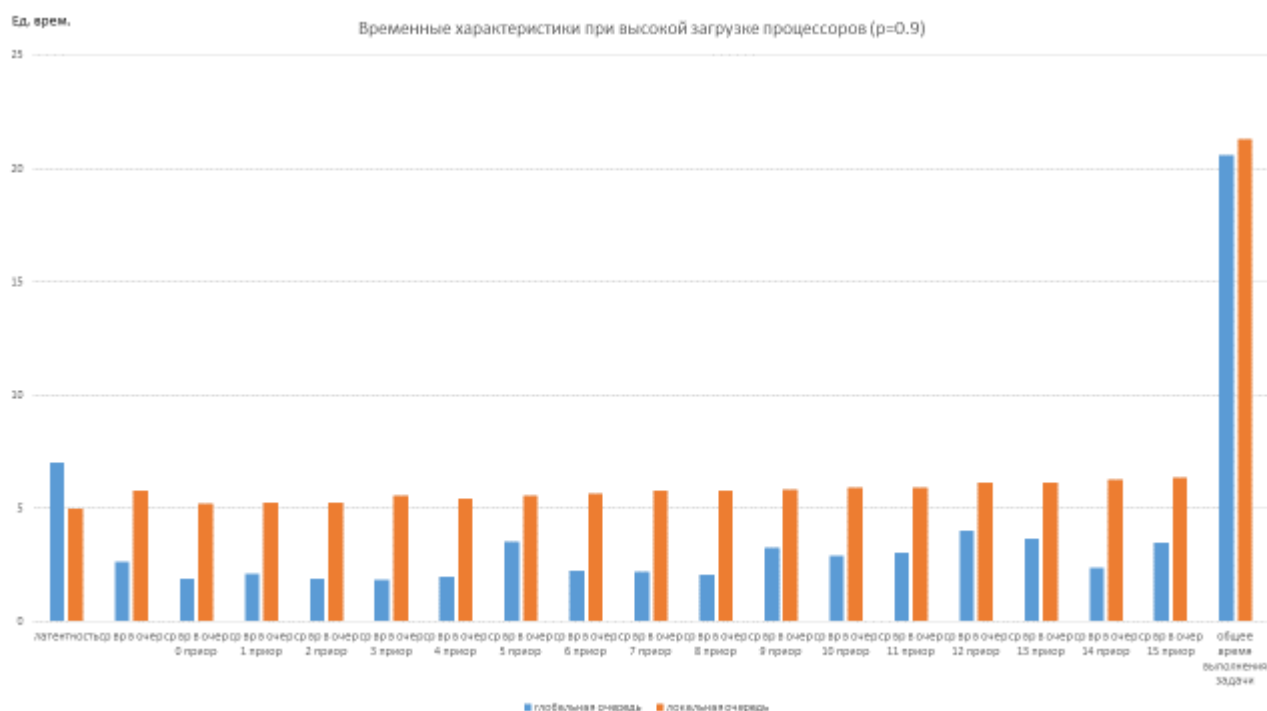
Временные
характеристики при
высокой загрузке
процессоров ($p=0.5$)

	глобальная очередь	локальная очередь
латентность (ед.вр.)	7	5
ср вр в очер	2,64	5,761
ср вр в очер 0 приор	1,898	5,221
ср вр в очер 1 приор	2,106	5,234
ср вр в очер 2 приор	1,866	5,242
ср вр в очер 3 приор	1,848	5,543
ср вр в очер 4 приор	1,979	5,435
ср вр в очер 5 приор	3,528	5,553
ср вр в очер 6 приор	2,252	5,645
ср вр в очер 7 приор	2,177	5,765
ср вр в очер 8 приор	2,07	5,783
ср вр в очер 9 приор	3,242	5,834
ср вр в очер 10 приор	2,889	5,926
ср вр в очер 11 приор	3,041	5,902
ср вр в очер 12 приор	4,027	6,112
ср вр в очер 13 приор	3,656	6,136
ср вр в очер 14 приор	2,371	6,263
ср вр в очер 15 приор	3,457	6,352
общее среднее время выполнения задачи в планировщике	20,587	21,295

Ед. врем.

Временные характеристики при средней загрузке процессоров ($p=0.5$)





Выводы

Латентность в условиях бесконфликтности планировщика с глобальной очередью больше латентности планировщика с локальными очередями на 40% (7 ед.вр. против 5 ед.вр.), а среднее время ожидания заявки в очереди в планировщике с глобальной очередью ниже, чем в планировщике с локальными очередями в 2,97 раза (1,535 ед.вр. против 4,56 ед.вр.) раза при средней загрузке и в 2,18 раза (2,64 ед.вр. против 5,761 ед.вр.) раза при высокой загрузке. Время обслуживания задач планировщика с глобальной очередью больше времени обслуживания задач планировщика с локальными очередями при средней загрузке на 11% (19,864 ед.вр. против 17,835 ед.вр.) и при высокой нагрузке процессоров меньше на 3% (20,587 ед.вр. против 21,295 ед.вр.) обслуживания задач локального планировщика.

Модель планировщика с глобальной очередью задач показала лучшие результаты по производительности, чем модель планировщика с локальными очередями по критерию среднего времени отклика планировщика (задержки заявок в очереди). Джиттер задач глобального планировщика меньше джиттера задач локального планировщика. По критерию времени ответа системы (дедлайн) планировщики с глобальной и локальными очередями показали примерно равные результаты. Результат можно объяснить лучшей балансировкой загрузки процессоров, которая в глобальных планировщиках осуществляется автоматически и отсутствием балансировки в раздельном планировщике.