О. В. КАРАВАЕВА

ОПЕРАЦИОННЫЕ СИСТЕМЫ. УПРАВЛЕНИЕ ПАМЯТЬЮ И ПРОЦЕССАМИ

Учебное пособие

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ

«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт математики и информационных систем

Факультет автоматики и вычислительной техники

Кафедра электронных вычислительных машин

О. В. КАРАВАЕВА

ОПЕРАЦИОННЫЕ СИСТЕМЫ. УПРАВЛЕНИЕ ПАМЯТЬЮ И ПРОЦЕССАМИ

Учебное пособие

Киров

2017

УДК 004.451(07)
К21

Рекомендовано к изданию методическим советом факультета автоматики и вычислительной техники ВятГУ

Допущено редакционно-издательской комиссией методического совета ВятГУ в качестве учебного пособия для студентов направлений 09.03.01 «Информатика и вычислительная техника», 09.03.04 «Программная инженерия» всех профилей подготовки

Рецензенты:
кандидат технических наук, доцент,
и. о. заведующего кафедрой автоматики и телемеханики ВятГУ
В. И. Семеновых

кандидат технических наук, доцент кафедры математических
и естественно-научных дисциплин КФ МФЮА
Т. А. Анисимова

Караваева, О. В.

К21 Операционные системы. Управление памятью и процессами: учебное пособие / О. В. Караваева. – Киров: ВятГУ, 2017. – 68 с.

УДК 004.451(07)

Издание предназначено для выполнения лабораторных работ по дисциплине «Операционные системы».

Авторская редакция
Тех. редактор Д. В. Дедюхина

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Операционная система – комплекс взаимосвязанных программ, предназначенных для управления ресурсами компьютера и организации взаимодействия с пользователем. Основной задачей операционной системы является управление всеми ресурсами вычислительной системы.

Многозадачная операционная система занимается переключением процессора с одного процесса на другой, обеспечивая его эффективную загрузку, а также отслеживает конфликты при обращении к общим ресурсам.

Критерий эффективности, в соответствии с которым операционная система организует управление ресурсами компьютера, может быть различным и зависит от назначения вычислительной системы, частью которой она является.

Операционная система должна корректно предоставлять ресурсы всем процессам по их запросам. Для этого ОС должна знать, какие ресурсы есть в системе и каким образом их можно выделять процессам.

Вычислительная система называется многопроцессорной, если она содержит несколько процессоров, работающих с общей ОП (общее поле оперативной памяти) и управляется одной общей операционной системой. Часто в МПС организуется общее поле внешней памяти. Под общим полем понимается равнодоступность устройств. Так, общее поле памяти означает, что все модули ОП доступны всем процессорам и каналам ввода-вывода (или всем периферийным устройствам в случае наличия общего интерфейса); общее поле ВЗУ означает, что образующие его устройства доступны любому процессору и каналу.

1. ОСНОВНЫЕ ВИДЫ РЕСУРСОВ И СПОСОБЫ ИХ РАСПРЕДЕЛЕНИЯ

Все ресурсы можно разделить на две большие группы.

Попеременно (квазипараллельно) разделяемый ресурс, или монопольно используемый ресурс – это ресурс, который в один момент времени может принадлежать только одному процессу.

Параллельно разделяемый ресурс – это ресурс, который в один момент времени могут использовать несколько процессов.

На рисунке 1.1 представлена совокупность ресурсов, которыми управляет операционная система.

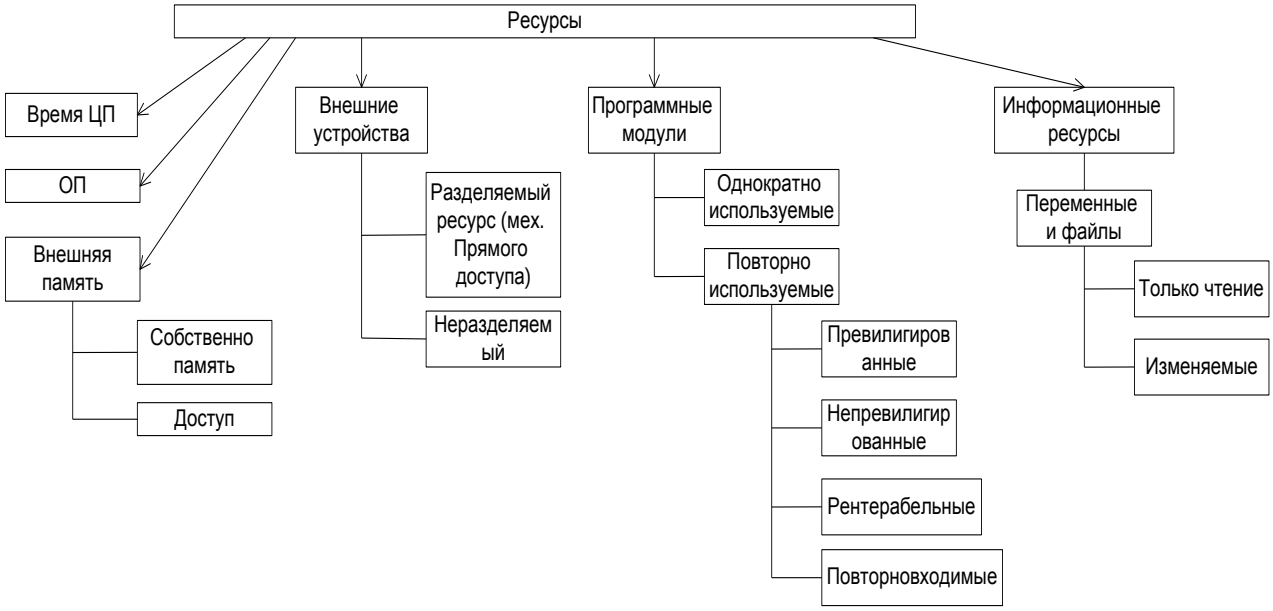

Рис. 1.1. Виды ресурсов

Одним из важнейших ресурсов является сам процессор, точнее – процессорное время. Процессорное время делится между процессами, исполняемыми на вычислительной системе таким образом, чтобы процессы получили процессорного времени, например, пропорционально их приоритету. Процессорное время – ресурс, разделяемый квазипараллельно. Процессор не может быть разделен между различными процессами, так как все современ-

ные массовые процессоры имеют фон Неймановскую архитектуру, которая использует единственный поток команд.

Вторым видом ресурсов вычислительной системы можно считать память. Оперативная память может быть разделена и одновременным способом (то есть в памяти одновременно может располагаться несколько процессов или, по крайней мере, текущие фрагменты, участвующие в вычислениях), и попеременно (в разные моменты оперативная память может предоставляться для разных вычислительных процессов). Память – очень интересный вид ресурса. Дело в том, что в каждый конкретный момент времени процессор при выполнении вычислений обращается к очень ограниченному числу ячеек оперативной памяти. С этой точки зрения желательно память разделять для возможно большего числа параллельно исполняемых процессов. С другой стороны, как правило, чем больше оперативной памяти может быть выделено для конкретного текущего процесса, тем лучше будут условия для его выполнения. Поэтому проблема эффективного разделения оперативной памяти между параллельно выполняемыми вычислительными процессами является одной из самых актуальных.

Когда говорят о внешней памяти (например, память на магнитных дисках), то собственно память и доступ к ней считаются разными видами ресурса. Каждый из этих ресурсов может предоставляться независимо от другого. Но для полной работы с внешней памятью необходимо иметь оба этих ресурса. Собственно внешняя память может разделяться одновременно, а доступ к ней – попеременно.

Если говорить о внешних устройствах, то они, как правило, могут разделяться параллельно, если используются механизмы прямого доступа. Если же устройство работает с последовательным доступом, то оно не может считаться разделяемым ресурсом.

Очень важным видом ресурсов являются программные модули. Прежде всего, рассматриваются системные программные модули, поскольку именно

они обычно и являются программными ресурсами и в принципе могут быть распределены между выполняющимися процессами.

Как известно, программные модули могут быть однократно и многократно (или повторно) используемыми. Однократно используемыми называют такие программные модули, которые могут быть правильно выполнены только один раз. Это означает, что в процессе своего выполнения они могут испортить себя: либо повреждается часть кода, либо – исходные данные, от которых зависит ход вычислений. Очевидно, что однократно используемые программные модули являются неделимым ресурсом. Более того, их обычно вообще не распределяют как ресурс системы. Системные однократно используемые программные модули, как правило, используются только на этапе загрузки ОС. При этом следует иметь в виду тот очевидный факт, что собственно двоичные файлы, которые обычно хранятся на системном диске и в которых и записаны эти модули, не портятся, а потому могут быть повторно использованы при следующем запуске ОС.

Повторно используемые программные модули, в свою очередь, могут быть непривилегированными, привилегированными и реентерабельными.

Привилегированные программные модули работают в так называемом привилегированном режиме, то есть при отключенной системе прерываний, так, что никакие внешние события не могут нарушить естественный порядок вычислений. В результате программный модуль выполняется до своего конца, после чего он может быть вновь вызван на исполнение из другой задачи (другого вычислительного процесса). С позиций стороннего наблюдателя по отношению к вычислительным процессам, которые попеременно (причем, возможно, неоднократно) в течение срока своей «жизни» вызывают некоторый привилегированный программный модуль, такой модуль будет выступать как попеременно разделяемый ресурс. Структура привилегированных программных модулей изображена на рисунке 1.2. Здесь в первой секции программного модуля выключается система прерываний. В последней секции, напротив, включается система прерываний.

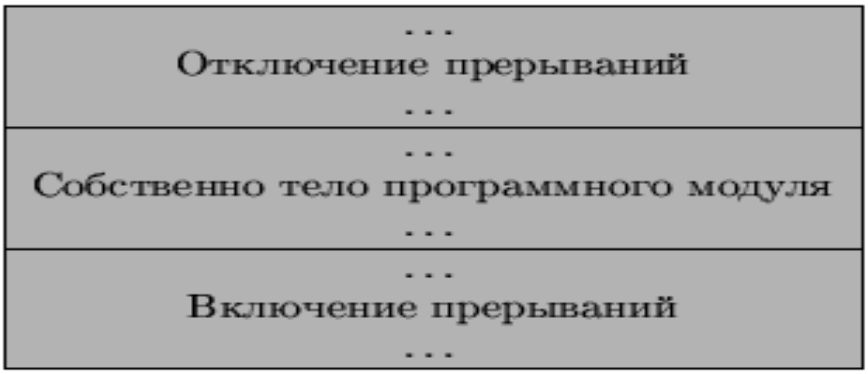

Рис. 1.2. Структура привилегированного программного модуля

Непривилегированные программные модули – это обычные программные модули, которые могут быть прерваны во время своей работы. Следовательно, в общем случае их нельзя считать разделяемыми, потому что если после прерывания выполнения такого модуля, исполняемого в рамках одного вычислительного процесса, запустить его еще раз по требованию другого вычислительного процесса, то промежуточные результаты для прерванных вычислений могут быть потеряны.

В противоположность этому, реентерабельные программные модули (reenterable) допускают повторное многократное прерывание своего исполнения и повторный их запуск по обращению из других задач (вычислительных процессов). Для этого реентерабельные программные модули должны быть созданы таким образом, чтобы было обеспечено сохранение промежуточных вычислений для прерываемых вычислений и возврат к ним, когда вычислительный процесс возобновляется с прерванной ранее точки. Это может быть реализовано двумя способами: с помощью статических и динамических методов выделения памяти под сохраняемые значения

Кроме реентерабельных программных модулей существуют еще повторно входимые (от reentrance). Этим термином называют программные модули, которые тоже допускают свое многократное параллельное использование, но в отличие от реентерабельных их нельзя прерывать. Повторно входимые программные модули состоят из привилегированных секций и повторное обращение к ним возможно только после завершения какой-нибудь из

таких секций. После выполнения очередной привилегированной секции управление может быть передано супервизору, и если он предоставит возможность выполняться другому процессу, то возможно повторное вхождение в рассматриваемый программный модуль. Другими словами, в повторно входимых программных модулях четко предопределены все допустимые (возможные) точки входа. Следует отметить, что повторно входимые программные модули встречаются гораздо чаще реентерабельных (повторно прерываемых). Наконец, имеются и информационные ресурсы, то есть в качестве ресурсов могут выступать данные. Информационные ресурсы могут существовать как в виде переменных, находящихся в оперативной памяти, так и в виде файлов. Если процессы используют данные только для чтения, то такие информационные ресурсы можно разделять. Если же процессы могут изменять информационные ресурсы, то необходимо специальным образом организовывать работу с такими данными.

В данном учебном пособии рассматривается управление основными ресурсами системы, к которым относятся оперативная память и процессорное время.

2. УПРАВЛЕНИЕ ПАМЯТЬЮ

Оперативная память – это важнейший ресурс любой вычислительной системы, поэтому от выбранных механизмов распределения памяти между выполняющимися процессорами очень сильно зависит эффективность использования ресурсов системы, ее производительность и возможности, которыми могут пользоваться программисты при создании своих программ.

Функциями ОС по управлению памятью в мультипрограммных системах являются:

– отслеживание (учет) свободной и занятой памяти;

– первоначальное и динамическое выделение памяти процессам приложений и самой операционной системе и освобождение памяти по завершении процессов;

– настройка адресов программы на конкретную область физической памяти;

– полное или частичное вытеснение кодов и данных процессов из ОП на диск, когда размеры ОП недостаточны для размещения всех процессов, и возвращение их в ОП;

– защита памяти, выделенной процессу, от возможных вмешательств со стороны других процессов;

– дефрагментация памяти [1].

Процесс может быть загружен в раздел равного или большего раздела.

Как правило, программист обращается к памяти с помощью некоторого набора логических имен, которые чаще всего являются символьными, а не числовыми, и для которого отсутствует отношение порядка. Другими словами, в общем случае множество переменных неупорядоченно, хотя отдельные переменные и могут иметь частичную упорядоченность (например, элементы массива). Имена переменных и входных точек программных модулей составляют пространство имен.

С другой стороны, существует понятие физической оперативной памяти, собственно с которой и работает процессор, извлекая из нее команды и данные и помещая в нее результаты вычислений.

2.1. Виртуальная память

Системное программное обеспечение должно связать каждое указанное пользователем имя с физической ячейкой памяти, то есть осуществить отображение пространства имен на физическую память компьютера. В общем случае это отображение осуществляется в два этапа: сначала системой программирования, а затем операционной системой (с помощью специальных программных модулей управления памятью и использования соответствующих аппаратных средств вычислительной системы). Между этими этапами обращения к памяти имеют форму виртуального или логического адреса. При этом можно сказать, что множество всех допустимых значений виртуального адреса для некоторой программы определяет ее виртуальное адресное пространство или виртуальную память.

Виртуальное адресное пространство – это максимально доступное приложению адресное пространство. Объём виртуального адресного пространства зависит от архитектуры компьютера и операционной системы. Он зависит от архитектуры компьютера, так как именно архитектура определяет, сколько бит используется для адресации. Он также зависит от операционной системы, так как в зависимости от реализации операционная система может накладывать дополнительные ограничения, помимо ограничений архитектуры.

Прилагательное «виртуальное» применительно к виртуальному адресному пространству означает, что это общее число доступных приложению уникально адресуемых ячеек памяти, но не общий объём памяти, установленной в компьютере, или выделенной в конкретный момент времени данному приложению.

В системе с виртуальной памятью используемые программами адреса, называемые виртуальными адресами, транслируются в физические адреса в памяти компьютера.

Термин виртуальная память фактически относится к системам, которые сохраняют виртуальные адреса во время исполнения. Так как второе отображение осуществляется в процессе исполнения задачи, то адреса физических ячеек могут изменяться. При правильном применении такие изменения могут улучшить использование памяти, избавив программиста от деталей управления ею, и даже увеличить надежность вычислений.

2.2. Простое непрерывное распределение и распределение с перекрытием

Простое непрерывное распределение – это самая простая схема, согласно которой вся память условно может быть разделена на три части:

- область, занимаемая операционной системой;
- область, в которой размещается исполняемая задача;
- незанятая ничем (свободная) область памяти.

Изначально являясь самой первой схемой, она продолжает и сегодня быть достаточно распространенной. Эта схема предполагает, что ОС не поддерживает мультипрограммирование, поэтому не возникает проблемы распределения памяти между несколькими задачами. Программные модули, необходимые для всех программ, располагаются в области самой ОС, а вся оставшаяся память может быть предоставлена задаче. Эта область памяти при этом получается непрерывной, что облегчает работу системы программирования. Привязка виртуальных адресов программы к физическому адресному пространству осуществляется на этапе загрузки задачи в память.

Чтобы для задач отвести как можно больший объем памяти, операционная система строится таким образом, что постоянно в оперативной памяти располагается только самая нужная ее часть (ядро). Остальные модули ОС могут быть обычными диск-резидентными (или транзитными).

Такая схема распределения влечет за собой два вида потерь вычислительных ресурсов – потеря процессорного времени, потому что процессор простаивает, пока задача ожидает завершения операций ввода/вывода, и потеря самой оперативной памяти, потому что далеко не каждая программа использует всю память, а режим работы в этом случае однопрограммный. Однако это очень недорогая реализация и можно отказаться от многих функций операционной системы. В частности, такая сложная проблема, как защита памяти, здесь вообще не стоит.

Если есть необходимость создать программу, логическое (и виртуальное) адресное пространство которой должно быть больше, чем свободная область памяти, или даже больше, чем весь возможный объем оперативной памяти, то используется распределение с перекрытием (так называемые оверлейные структуры). Этот метод распределения предполагает, что вся программа может быть разбита на части – сегменты. Каждая оверлейная программа имеет одну главную часть и несколько сегментов, причем в памяти машины одновременно могут находиться только ее главная часть и один или несколько не перекрывающихся сегментов.

Пока в оперативной памяти располагаются выполняющиеся сегменты, остальные находятся во внешней памяти. После того как текущий (выполняющийся) сегмент завершит свое выполнение, возможны два варианта. Либо он сам обращается к ОС с указанием, какой сегмент должен быть загружен в память следующим. Либо он возвращает управление главному сегменту задачи, и уже тот обращается к ОС с указанием, какой сегмент сохранить (если это нужно), а какой сегмент загрузить в оперативную память, и вновь отдает управление одному из сегментов, располагающихся в памяти. Сегменты кода, как правило, не претерпевают изменений в процессе своего исполнения, поэтому при загрузке нового сегмента кода на место отработавшего последний можно не сохранять во внешней памяти, в отличие от сегментов данных, которые сохранять необходимо.

Первоначально программисты сами должны были включать в тексты своих программ соответствующие обращения к ОС и тщательно планировать, какие сегменты могут находиться в оперативной памяти одновременно, чтобы их адресные пространства не пересекались. Однако с некоторых пор эти вызовы система программирования стала подставлять в код программы сама, автоматически, если в том возникает необходимость. Так, в известной и популярной системе программирования *Turbo Pascal*, начиная с третьей версии, программист просто указывал, что данный модуль является оверлейным. И при обращении к нему из основной программы модуль загружался в память и ему передавалось управление. Все адреса определялись системой программирования автоматически, обращения к *DOS* для загрузки оверлеев тоже генерировались системой *Turbo Pascal*.

2.3. Распределение памяти статическими и динамическими разделами

Для организации мультипрограммного режима необходимо обеспечить одновременное расположение в оперативной памяти нескольких задач (целиком или частями). Самая простая схема распределения памяти между несколькими задачами предполагает, что память, незанятая ядром ОС, может быть разбита на несколько непрерывных частей (разделов). Разделы характеризуются именем, типом, границами.

Разбиение памяти на несколько непрерывных разделов может быть фиксированным (статическим), либо динамическим (то есть процесс выделения нового раздела памяти происходит непосредственно при появлении новой задачи).

Выделяют разделы с фиксированными границами и разделы с подвижными границами

2.3.1. Разделы с фиксированными границами

Разбиение всего объема оперативной памяти на несколько разделов может осуществляться единовременно или по мере необходимости операто-

ром системы. Однако и во втором случае при выполнении разбиения памяти на разделы вычислительная система более ни для каких целей в этот момент не используется. Пример разбиения памяти на несколько разделов приведен на рисунке 2.1.

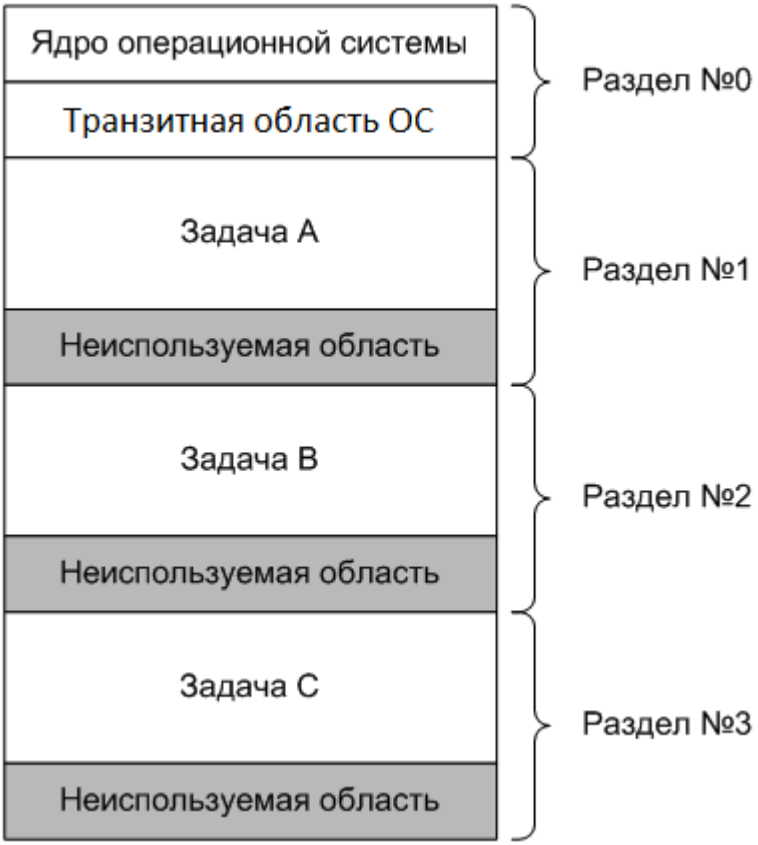

Рис. 2.1. Распределение памяти разделами
с фиксированными границами

В каждом разделе в каждый момент времени может располагаться по одной программе (задаче). В этом случае по отношению к каждому разделу можно применить все те методы создания программ, которые используются для однопрограммных систем. Иногда в некотором разделе размещалось по несколько небольших программ, которые постоянно в нем и находились. Такие программы назывались ОЗУ-резидентными (или просто резидентными).

При небольшом объеме памяти и, следовательно, небольшом количестве разделов увеличить количество параллельно выполняемых приложений (особенно когда эти приложения интерактивны и в основном ожидают опе-

раций ввода/вывода) можно за счет свопинга (swapping). При свопинге задача может быть целиком выгружена на магнитный диск (перемещена во внешнюю память), а на ее место загружается либо более привилегированная, либо просто готовая к выполнению другая задача, находившаяся на диске в приостановленном состоянии. При свопинге из основной памяти во внешнюю (и обратно) перемещается вся программа, а не ее отдельная часть.

Серьезная проблема, которая возникает при организации мультипрограммного режима работы вычислительной системы – это защита как самой ОС от ошибок и преднамеренного вмешательства задач в ее работу, так и самих задач друг от друга.

Одной из простейших, но достаточно сильных мер является введение регистров защиты памяти. В эти регистры ОС заносит граничные значения области памяти раздела текущей исполняющейся задачи. При нарушении адресации возникает прерывание, и управление передается супервизору ОС. Обращения к ОС за необходимыми сервисами осуществляются не напрямую, а через команды программных прерываний, что обеспечивает передачу управления только в предопределенные входные точки кода ОС.

Основным недостатком такого способа распределения памяти является наличие порой достаточно большого объема неиспользуемой памяти (см. рис. 2.1). Неиспользуемая память может быть в каждом из разделов. Поскольку разделов несколько, то и неиспользуемых областей получается несколько, поэтому такие потери стали называть фрагментацией памяти. В отдельных разделах потери памяти могут быть очень значительными, однако использовать фрагменты свободной памяти при таком способе распределения не представляется возможным. Желание разработчиков сократить столь значительные потери привело их к следующим двум решениям:

– выделять раздел ровно такого объема, который нужен под текущую задачу;

– размещать задачу не в одной непрерывной области памяти, а в нескольких областях.

Второе решение реализовалось в нескольких способах организации виртуальной памяти.

2.3.2. Разделы с подвижными границами

Чтобы избавиться от фрагментации, можно попробовать размещать в оперативной памяти задачи плотно, одну за другой, выделяя ровно столько памяти, сколько задача требует. Специальный планировщик (диспетчер памяти) ведет список адресов свободной оперативной памяти. При появлении новой задачи диспетчер памяти просматривает этот список и выделяет для задачи раздел, объем которого либо равен необходимому, либо чуть больше. При этом модифицируется список свободной памяти. При освобождении раздела диспетчер памяти пытается объединить освобождающийся раздел с одним из свободных участков, если таковой является смежным. Данный метод реализован в лабораторной установке.

При этом список свободных участков может быть упорядочен либо по адресам, либо по объему. Выделение памяти под новый раздел может осуществляться одним из трех способов:

- первый подходящий участок;
- самый подходящий участок;
- самый неподходящий участок.

В первом случае список свободных областей упорядочивается по адресам. Диспетчер памяти просматривает этот список и выделяет задаче раздел в той области, которая первой подойдет по объему.

Правило *«первый подходящий»* приводит к тому, что память для небольших задач преимущественно будет выделяться в области младших адресов и, следовательно, это будет увеличивать вероятность того, что в области старших адресов будут образовываться фрагменты достаточно большого объема.

Способ *«самый подходящий»* предполагает, что список свободных областей упорядочен по возрастанию объема этих фрагментов. В этом случае

при просмотре списка для нового раздела будет использован фрагмент свободной памяти, объем которой наиболее точно соответствует требуемому. Однако оставшийся фрагмент оказывается настолько малым, что в нем уже вряд ли удастся разместить какой-либо еще раздел и при этом этот фрагмент попадет в самое начало списка. Поэтому в целом такую дисциплину нельзя назвать эффективной.

Как ни странно, самым эффективным правилом является последнее, по которому для нового раздела выделяется *«самый неподходящий»* фрагмент свободной памяти. Для этой дисциплины список свободных областей упорядочивается по убыванию объема свободного фрагмента. Очевидно, что если есть такой фрагмент памяти, то он сразу же и будет найден, и поскольку этот фрагмент является самым большим, то, скорее всего, после выделения из него раздела памяти для задачи оставшаяся область памяти еще сможет быть использована в дальнейшем.

Однако очевидно, что при любой дисциплине обслуживания будет наблюдаться сильная фрагментация. При этом возможны ситуации, когда из-за сильной фрагментации памяти диспетчер задач не сможет образовать новый раздел, хотя суммарный объем свободных областей будет больше, чем необходимо для задачи. В этой ситуации возможно организовать так называемое «уплотнение памяти», для чего все вычисления приостанавливаются, и диспетчер памяти корректирует свои списки, перемещая разделы в начало памяти (или, наоборот, в область старших адресов). Недостатком этого решения является потеря времени на уплотнение и невозможность при этом выполнять сами вычислительные процессы.

3. УПРАВЛЕНИЕ ЗАДАЧАМИ В ОПЕРАЦИОННЫХ СИСТЕМАХ

Важнейшей частью ОС, непосредственно влияющей на функционирование вычислительной машины, является подсистема управлениями процессами.

Процесс – это абстракция, описывающая выполняющуюся программу. Для ОС процесс представляет собой единицу работы, заявку на потребление системных ресурсов. Подсистема управления процессами планирует выполнение процессов, т. е. распределяет процессорное время между несколькими одновременно существующими в системе процессами, а так же занимается созданием и уничтожение процессов, обеспечивает процессы необходимыми системными ресурсами, поддерживает взаимодействие между процессами.

ОС поддерживает в памяти специальные информационные структуры, в которые записывает, какие ресурсы выделены каждому процессу. Она может назначить процессу ресурсы единоличного пользования или в совместное пользование с другими процессами. Некоторые из ресурсов выделяются процессу при его создании, а некоторые динамически, по запросам во время выполнения. Ресурсы могут быть предписаны процессу на все время его жизни или только на определенный период. При выполнении этих функций подсистема управления процессами взаимодействует с другими подсистемами ОС, ответственными за управление данными ресурсами.

Создание задачи сопряжено с формированием соответствующей информационной структуры, а ее удаление – с расформированием. Создание и удаление задач осуществляется по соответствующим запросам от пользователей или от самих задач. Задача может породить новую задачу. При этом между задачами появляются «родственные» отношения. Порождающая задача называется «отцом», «родителем», а порожденная – «потомком». Родитель может приостановить или удалить свою дочернюю задачу, тогда как потомок не может управлять родителем.

Решение вопросов, связанных с тем, какой задаче следует предоставить процессорное время в данный момент, возлагается на специальный модуль операционный системы, чаще всего называемый диспетчером задач. Вопросы же подбора вычислительных процессов, которые не только можно, но и целесообразно решать параллельно, возлагаются на планировщик процессов.

Операционная система выполняет следующие основные функции, связанные с управлением процессами:

- создание и удаление задач;
- планирование процессов и диспетчеризация задач;
- синхронизация задач, обеспечение их средствами коммуникации.

3.1. Планирование и диспетчеризация процессов

Планирование процессов (задач) – это определение очередности получения ресурсов вычислительной системы для процессов при их активизации. Планирование процесса связано с его переводом из состояния бездействия в состояние готовности. Такой перевод осуществляется однократно на интервале существования процесса.

Стратегия планирования определяет, какие процессы планируются на выполнение для того, чтобы достичь поставленной цели.

Основными задачами алгоритма планирования процессов являются:

- равнодоступность – предоставление каждому процессу справедливой доли времени центрального процессора;
- принуждение к определенной политике – наблюдение за выполнением определенной политики;
- баланс – поддержка загруженности всех составных частей системы [3].

Известно большое количество различных стратегий выбора процесса, которому необходимо предоставить процессор. Среди них, прежде всего, можно назвать следующие стратегии:

– по возможности заканчивать вычисления (вычислительные процессы) в том же самом порядке, в котором они были начаты;

– отдавать предпочтение более коротким процессам;

– предоставлять всем пользователям (процессам пользователей) одинаковые услуги, в том числе и одинаковое время ожидания.

Диспетчеризация процессов (задач) – это определение очерёдности получения процессора для процессов (задач), находящихся в состоянии готовности, с целью их выполнения. Диспетчеризация процесса связана с его переводом из состояния готовности в состояние выполнения (счёта). Диспетчеризация, для конкретного процесса, может выполняться многократно, т. е. процесс может несколько раз переходить из состояния готовности в состояние выполнения и обратно на интервале своего существования. Так как в каждый такт процессорного времени могут выполняться команды только одной задачи, диспетчеризация предполагает создание и модификацию очереди готовых к выполнению задач (процессов). Элементами такой очереди (как и других очередей в вычислительной системе) на «физическом уровне» являются *дескрипторы задач*.

Дескриптор задачи – это специальная информационная структура, в которой хранятся характеристики задачи необходимые для целей управления со стороны ОС. Первоначально, дескриптор задачи формируется на этапе её трансляции. Перед выполнением задачи, такой дескриптор загружается в оперативную память совместно с её кодом и данными. Информация о задаче, которая хранится в дескрипторе, разделяется на несколько групп, и часть её динамически изменяется на интервале существования задачи. Рассмотрим эти группы:

– информация по идентификации задачи (имя задачи, тип задачи);

– информация о ресурсах, которые необходимы задаче для её выполнения и о ресурсах, которые используются в настоящее время (идентификаторы необходимых внешних устройств);

– информация о текущем состоянии задачи (содержимое некоторых регистров процессора);

– информация о родственных связях задачи (имена родительского процесса и процессов-предков);

– информация, необходимая для целей планирования и диспетчеризации (адресные ссылки на соседние дескрипторы, находящиеся в той же очереди, приоритет задачи, ссылки на объекты синхронизации).

Обычно, дескриптор задачи имеет размер порядка нескольких десятков машинных слов. Есть два основных способа размещения дескрипторов в оперативной памяти. Первый, более ранний, состоит в том, что ОС выделяет в системной области памяти специальный раздел под таблицу дескрипторов. Такая таблица имеет фиксированный размер и фиксированный начальный адрес. Такой способ прост с точки зрения управления и сложности реализации, но имеет существенный недостаток – ограничение параллелизма в вычислительной системе. Причиной этого недостатка является ограниченный размер таблицы дескрипторов. Второй способ, более современный, состоит в том, что ОС выделяет подходящую свободную область оперативной памяти под очередной загружаемый дескриптор задачи. Методологически, такой способ не ограничивает параллелизм функционирования вычислительной системы.

Диспетчеризация сводится к следующему:

– сохранение контекста текущего потока, который требуется сменить;

– загрузка контекста нового потока, выбранного в результате планирования;

– запуск нового потока на выполнение.

ОС выполняют диспетчеризацию потоков, совместно с аппаратными средствами процессора.

Всем потокам должно принадлежать разное время выполнения:

– ни один поток не будет занимать процессор дольше определенного времени;

– максимальное быстродействие выполнения;

– равное время всем процессам.

Возможные состояния процессов и переходы между ними представлены на рисунке 3.1.

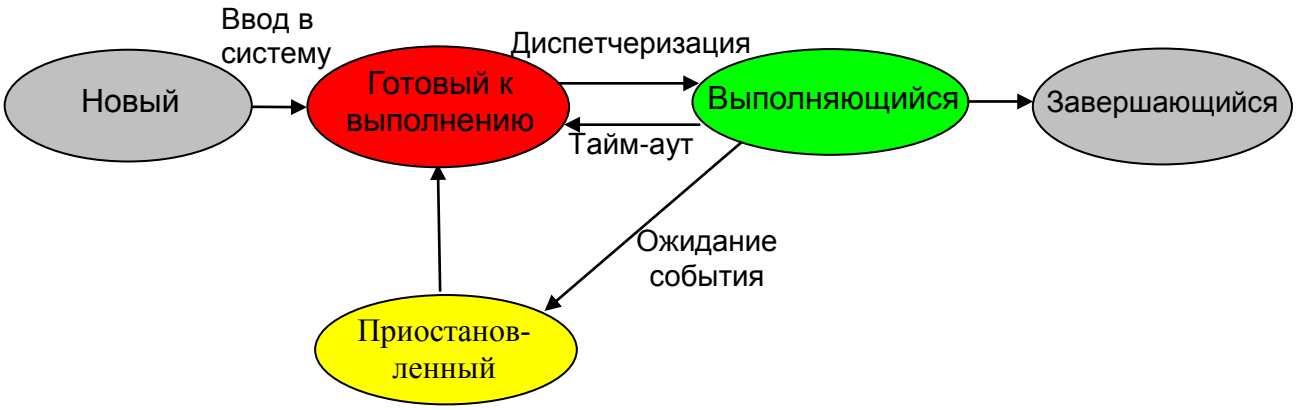

Рис. 3.1. Состояние процессов

При работе ОС может возникнуть ситуация, когда все процессы, находящиеся в памяти, будут заблокированы, то есть в памяти не будет ни одного готового к выполнению процесса.

Решением такой проблемы является свопинг, когда ОС переносит один из блокированных процессов на диск, помещая его в очередь приостановленных процессов. Затем ОС загружает новый процесс или другой процесс из очереди приостановленных процессов и продолжает его выполнение.

При выполнении процесс может быть прерван (по таймеру, в результате ошибки и т. п.), а после обработки прерывания операционной системой переходит снова в состояние готовности к выполнению. Если в процессе выполняется синхронный ввод-вывод, либо процесс должен ожидать наступления некоторого события (например, определенного момента времени), процесс переходит в состояние ожидания. При завершении ввода-вывода или при наступлении ожидаемого события процесс не получает сразу же квант процессорного времени, а переходит в состояние готовности к выполнению.

Процесс переходит в завершенное состояние при завершении работы программы процесса.

3.2. Планировщики

Операционная система при управлении процессами обеспечивает их поочередное выполнение. Эту задачу решает планировщик ОС. Для управления процессами ОС организует следующие очереди:

Очередь заданий (job queue) – содержит множество всех процессов в системе. В нее попадает каждый новый процесс и остается в ней в течение всего пребывания в системе.

Очередь готовых процессов (ready queue) – наиболее часто используемая и изменяемая очередь, содержащая множество всех процессов, находящихся в основной памяти и готовых к выполнению. В нее попадает каждый новый процесс, который система допускает к выполнению, а также каждый процесс после выполнения ввода-вывода или наступление ожидаемого события.

Очереди процессов, ожидающих ввода-вывода (device queues) – множества процессов, ожидающих результата работы устройств ввода-вывода (для каждого устройства организуется своя очередь).

Управление процессами операционной системой и поведение процессов в системе можно рассматривать как миграцию между различными очередями [4].

В операционной системе диспетчеризация процессов выполняется обычно несколькими планировщиками, каждый из которых имеет свою периодичность вызовов и свою определенную задачу, которую он решает.

Долговременный планировщик (планировщик заданий) определяет, какие процессы должны быть перемещены в очередь готовых процессов. В большинстве современных операционных систем, долгосрочный планировщик отсутствует.

Кратковременный планировщик (планировщик процессора) – определяет, какие процессы должны быть выполнены следующими и каким процессам должен быть предоставлен процессор [4].

Каждый планировщик имеет свои особенности поведения, как и каждый процесс.

Кратковременный планировщик вызывается очень часто, по крайней мере не реже, чем по истечение очередного кванта времени процессора. Поэтому он должен быть очень быстрым, максимально эффективно реализованным. Понятно, что недопустимо, например, если время работы этого планировщика окажется сравнимым с размером самого кванта времени – слишком велики будут накладные расходы.

Долговременный планировщик вызывается относительно редко, так как система не столь часто принимает решения о переводе процесса в очередь готовых процессов. Поэтому он может быть сравнительно медленным, не столь эффективно реализованным.

Однако, поскольку основной задачей системы в целом остается обслуживание как можно большего числа процессов, именно долговременный планировщик определяет степень (коэффициент) мультипрограммирования – число процессов, которое обслуживает система в единицу времени.

3.3. Дисциплины диспетчеризации

Дисциплина диспетчеризации – это некоторое основное правило, реализующее очерёдность предоставления (выделения) процессора (процессорного времени) готовым к выполнению задачам (процессам). Любая конкретная дисциплина диспетчеризации выполняет две взаимосвязанные функции – выделение процессорного времени конкретной задаче (процессу), и создание и модификация очереди готовых к выполнению задач (обслуживание очереди). Дисциплина диспетчеризации реализуется специальной компонентой ОС – *диспетчером (диспетчером задач)*.

Известно большое количество правил (дисциплин диспетчеризации), в соответствии с которыми формируется список (очередь) готовых к выполнению задач.

Для сравнения алгоритмов диспетчеризации обычно используются некоторые критерии:

– загрузка центрального процессора (CPU utilization). В большинстве персональных систем средняя загрузка процессора не превышает 2–3 %, доходя в моменты выполнения сложных вычислений и до 100 %. В реальных системах, где компьютеры (например, серверы) выполняют очень много работы, загрузка процессора колеблется в пределах от 15–40 % (для легко загруженного процессора) до 90–100 % (для тяжело загруженного процессора);

– пропускная способность центрального процессора (CPU throughput). Пропускная способность процессора может измеряться количеством процессов, которые выполняются в единицу времени;

– время оборота (turnaround time). Для некоторых процессов важным критерием является полное время выполнения, то есть интервал от момента появления процесса во входной очереди до момента его завершения. Это время названо временем оборота и включает время ожидания во входной очереди, время ожидания в очереди готовых процессов, время ожидания в очередях к оборудованию, время выполнения в процессоре и время ввода-вывода;

– время ожидания (waiting time). Под временем ожидания понимается суммарное время нахождения процесса в очереди готовых процессов;

– время отклика (response time). Для интерактивных программ важным показателем является время отклика, или время, прошедшее от момента попадания процесса во входную очередь до момента первого обращения к терминалу.

Очевидно, что простейшая стратегия краткосрочного планировщика должна быть направлена на максимизацию средних значений загруженности и пропускной способности, времени ожидания и времени отклика.

Правильное планирование процессов в значительной степени влияет на производительность всей системы. Можно выделить следующие главные причины, приводящие к снижению производительности системы:

– накладные расходы на переключение процессора. Они определяются не только переключениями контекстов задач, но и (при переключении на потоки другого приложения) перемещениями страниц виртуальной памяти, а также необходимостью обновления данных в кэше (коды и данные одной задачи, находящиеся в кэше, не нужны другой задаче и будут заменены, что приведет к дополнительным задержкам);

– переключение на другую задачу в тот момент, когда текущая задача выполняет критическую секцию, а другие задачи активно ожидают входа в свою критическую секцию (см. главу 7). В этом случае потери будут особо велики (хотя вероятность прерывания выполнения коротких критических секций мала).

В случае мультипроцессорных систем применяются следующие методы повышения производительности системы:

– совместное планирование, при котором все потоки одного приложения (неблокированные) одновременно ставятся на выполнение процессорами и одновременно снимаются с выполнения (для сокращения переключений контекста);

– планирование, при котором находящиеся в критической секции задачи не прерываются, а активно ожидающие входа в критическую секцию задачи не ставятся на выполнение до тех пор, пока вход в секцию не освободится;

– планирование с учетом так называемых подсказок (hints) программы (во время ее выполнения), например, в известной своими новациями ОС Mach имелось два класса таких подсказок: во-первых, указания (разной степени категоричности) о снятии текущего процесса с процессора, во-вторых, указания о том процессе, который должен быть выбран взамен текущего.

3.3.1. Классификация дисциплин диспетчеризации

Различают два больших класса дисциплин обслуживания - бесприоритетные и приоритетные. При бесприоритетном обслуживании выбор задачи производится в некотором заранее установленном порядке без учета их относительной важности и времени обслуживания. При реализации приоритетных дисциплин обслуживания отдельным задачам предоставляется преимущественное право попасть в состояние исполнения. Перечень дисциплин обслуживания и их классификация приведены на рисунке 3.2.

Одним из основных методов гарантии обслуживания является использование динамических приоритетов.

Диспетчеризация с динамическими приоритетами требует дополнительных расходов на вычисление значений приоритетов исполняющихся задач, поэтому во многих ОС реального времени используются методы диспетчеризации на основе статических (постоянных) приоритетов. Хотя надо заметить, что динамические приоритеты позволяют реализовать гарантии обслуживания задач.

Существует два основных типа процедур планирования процессов – вытесняющие (preemptive) и невытесняющие (non-preemptive).

Non-preemptive multitasking – невытесняющая многозадачность – это способ планирования процессов, при котором активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление планировщику операционной системы для того, чтобы тот выбрал из очереди другой, готовый к выполнению процесс.

Preemptive multitasking – вытесняющая многозадачность – это такой способ, при котором решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается планировщиком операционной системы, а не самой активной задачей [2].

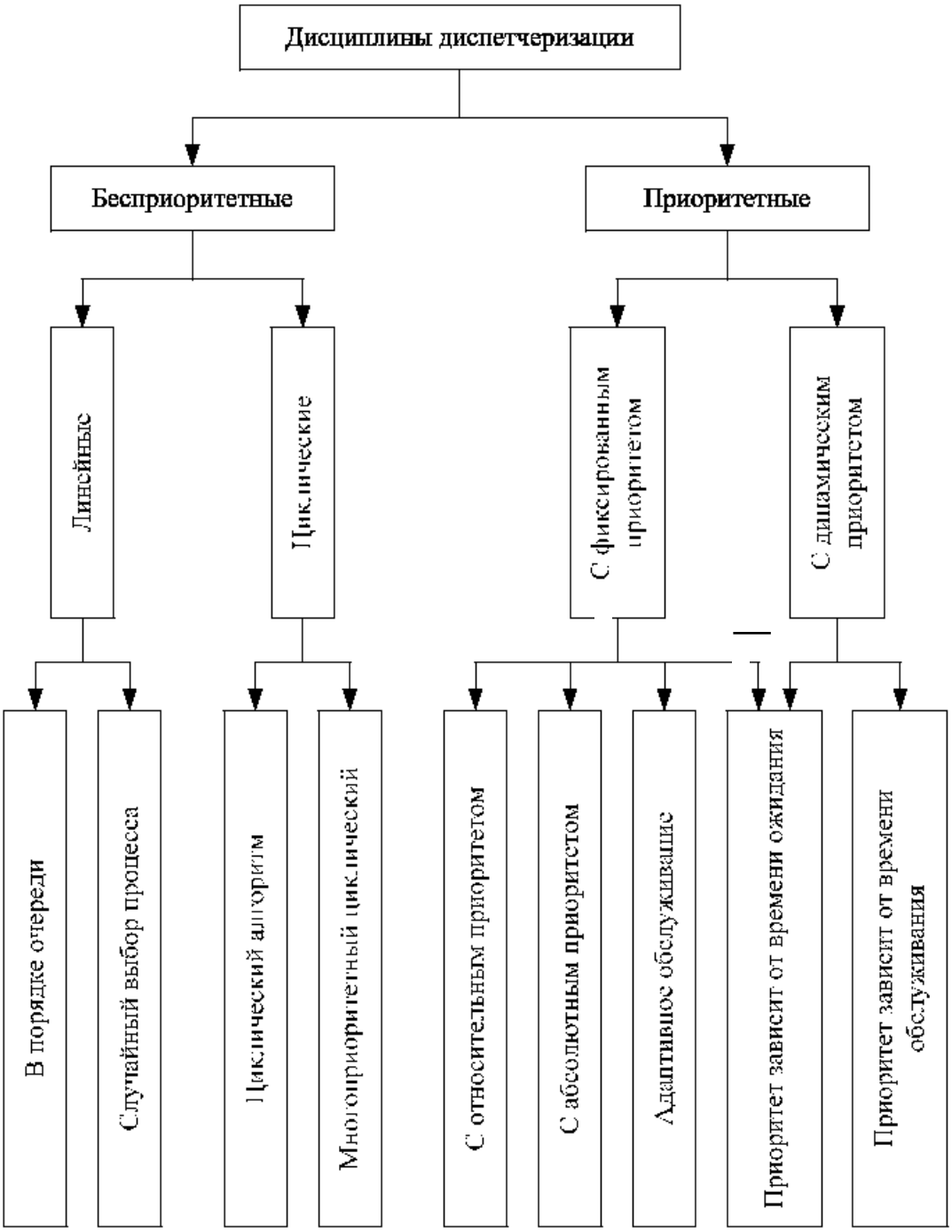

Рис. 3.2. Дисциплины диспетчеризации

Вытесняющая и невытесняющая многозадачность – это более широкие понятия, чем типы приоритетности. Приоритеты задач могут как использоваться, так и не использоваться и при вытесняющих, и при невытесняющих способах планирования. Так в случае использования приоритетов дисциплина относительных приоритетов может быть отнесена к классу систем с невытесняющей многозадачностью, а дисциплина абсолютных приоритетов – к классу систем с вытесняющей многозадачностью. А бесприоритетная дисциплина планирования, основанная на выделении равных квантов времени для всех задач, относится к вытесняющим алгоритмам. Основным различием между preemptive и non-preemptive вариантами многозадачности является степень централизации механизма планирования задач. При вытесняющей многозадачности механизм планирования задач целиком сосредоточен в операционной системе, и программист пишет свое приложение, не заботясь о том, что оно будет выполняться параллельно с другими задачами. При этом операционная система выполняет следующие функции: определяет момент снятия с выполнения активной задачи, запоминает ее контекст, выбирает из очереди готовых задач следующую и запускает ее на выполнение, загружая ее контекст. При невытесняющей многозадачности механизм планирования распределен между системой и прикладными программами. Прикладная программа, получив управление от операционной системы, сама определяет момент завершения своей очередной итерации и передает управление ОС с помощью какого-либо системного вызова, а ОС формирует очереди задач и выбирает в соответствии с некоторым алгоритмом (например, с учетом приоритетов) следующую задачу на выполнение. Такой механизм создает проблемы как для пользователей, так и для разработчиков.

Для пользователей это означает, что управление системой теряется на произвольный период времени, который определяется приложением (а не пользователем). Если приложение тратит слишком много времени на выполнение какой-либо работы, например, на форматирование диска, пользователь не может переключиться с этой задачи на другую задачу, например, на тек-

стовый редактор, в то время как форматирование продолжалось бы в фоновом режиме. Эта ситуация нежелательна, так как пользователи обычно не хотят долго ждать, когда машина завершит свою задачу. Поэтому разработчики приложений для невытесняющей операционной среды, возлагая на себя функции планировщика, должны создавать приложения так, чтобы они выполняли свои задачи небольшими частями. Например, программа форматирования может отформатировать одну дорожку дискеты и вернуть управление системе. После выполнения других задач система возвратит управление программе форматирования, чтобы та отформатировала следующую дорожку. Подобный метод разделения времени между задачами работает, но он существенно затрудняет разработку программ и предъявляет повышенные требования к квалификации программиста. Программист должен обеспечить «дружественное» отношение своей программы к другим выполняемым одновременно с ней программам, достаточно часто отдавая им управление. Крайним проявлением «недружественности» приложения является его зависание, которое приводит к общему краху системы. В системах с вытесняющей многозадачностью такие ситуации, как правило, исключены, так как центральный планирующий механизм снимет зависшую задачу с выполнения. Однако распределение функций планировщика между системой и приложениями не всегда является недостатком, а при определенных условиях может быть и преимуществом, потому что дает возможность разработчику приложений самому проектировать алгоритм планирования, наиболее подходящий для данного фиксированного набора задач. Так как разработчик сам определяет в программе момент времени отдачи управления, то при этом исключаются нерациональные прерывания программ в «неудобные» для них моменты времени. Кроме того, легко разрешаются проблемы совместного использования данных: задача во время каждой итерации использует их монопольно и уверена, что на протяжении этого периода никто другой не изменит эти данные. Существенным преимуществом невытесняющих систем является более высокая скорость переключения с задачи на задачу. Примером эффективного

использования невытесняющей многозадачности является файл-сервер NetWare, в котором, в значительной степени благодаря этому, достигнута высокая скорость выполнения файловых операций.

3.3.2. Невытесняющие дисциплины диспетчеризации

Диспетчеризация без перераспределения процессорного времени, то есть *невытесняющая многозадачность* – это такой способ диспетчеризации процессов, при котором активный процесс выполняется до тех пор, пока он сам не отдаст управление диспетчеру задач для выбора из очереди другого, готового к выполнению процесса или треда.

Для пользователей это означает, что управление системой может теряться на некоторый произвольный период времени, который определяется процессом выполнения приложения, а не системой, старающейся всегда обеспечить приемлемое время реакции на запросы пользователей. Подобный метод разделения времени между задачами существенно затрудняет разработку программ и предъявляет повышенные требования к квалификации программиста.

Однако распределение функций диспетчеризации между системой и приложениями не всегда является недостатком, а при определенных условиях может быть и преимуществом, потому что дает возможность разработчику приложений самому проектировать алгоритм распределения процессорного времени, наиболее подходящий для данного фиксированного набора задач. Примером эффективного использования не вытесняющей многозадачности является сетевая операционная система *Novell NetWare*, в которой в значительной степени благодаря этому достигнута высокая скорость выполнения файловых операций.

Дисциплина FCFS

Самой простой в реализации является дисциплина *FCFS (first come – first served)*, согласно которой задачи обслуживаются «в порядке очереди»,

то есть в порядке их появления. Те задачи, которые были заблокированы в процессе работы (попали в какое-либо из состояний ожидания, например, из-за операций ввода/вывода), после перехода в состояние готовности ставятся в эту очередь готовности перед теми задачами, которые еще не выполнялись (рис. 3.3).

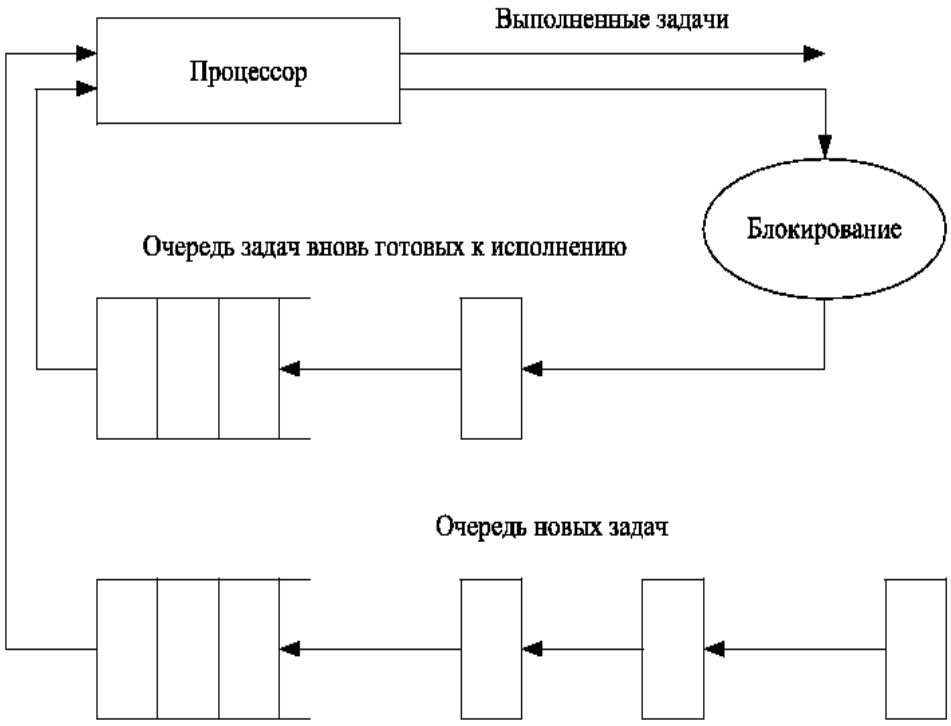

Рис. 3.3. Дисциплина диспетчеризации FSCS

Эта дисциплина обслуживания не требует внешнего вмешательства в ход вычислений, при ней не происходит перераспределение процессорного времени.

К достоинствам этой дисциплины, прежде всего, можно отнести простоту реализации и малые расходы системных ресурсов на формирование очереди задач.

Однако эта дисциплина приводит к тому, что при увеличении загрузки вычислительной системы растет и среднее время ожидания обслуживания, причем короткие задания (требующие небольших затрат машинного времени) вынуждены ожидать столько же, сколько и трудоемкие задания.

Дисциплины SJN, SRT

Избежать недостатка дисциплины *FCFS* (равноправна по отношению как к «длинным» так и к «коротким» процессам) позволяют дисциплины SJN и SRT.

Дисциплина обслуживания *SJN (shortest job next)* требует, чтобы для каждого задания была известна оценка в потребностях машинного времени.

Необходимость сообщать ОС характеристики задач, в которых описывались бы потребности в ресурсах вычислительной системы, привела к тому, что были разработаны соответствующие языковые средства. В частности, язык JCL (Job control language, язык управления заданиями) был одним из наиболее известных. Пользователи вынуждены были указывать предполагаемое время выполнения, и для того, чтобы они не злоупотребляли возможностью указать заведомо меньшее время выполнения (с целью получить результаты раньше других), ввели подсчет реальных потребностей. Диспетчер задач сравнивал заказанное время и время выполнения и в случае превышения указанной оценки в данном ресурсе ставил данное задание не в начало, а в конец очереди.

Дисциплина обслуживания SJN предполагает, что имеется только одна очередь заданий, готовых к выполнению. И задания, которые в процессе своего исполнения были временно заблокированы (например, ожидали завершения операций ввода/вывода), вновь попадают в конец очереди готовых к выполнению наравне с вновь поступающими. Это приводит к тому, что задания, которым требуется очень немного времени для своего завершения, вынуждены ожидать процессор наравне с длительными работами.

Для устранения этого недостатка и была предложена дисциплина *SRT (shortest remaining time),* в которой потоки, которым осталось меньше всего времени до завершения, помещаются в начало очереди.

3.3.3. Вытесняющие дисциплины диспетчеризации

Диспетчеризация с перераспределением процессорного времени между задачами, то есть *вытесняющая многозадачность* – это такой способ, при

котором решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается диспетчером задач, а не самой активной задачей. При вытесняющей многозадачности механизм диспетчеризации задач целиком сосредоточен в операционной системе, и программист может писать свое приложение, не заботясь о том, как оно будет выполняться параллельно с другими задачами.

При этом операционная система выполняет следующие функции: определяет момент снятия с выполнения текущей задачи, сохраняет ее контекст в дескрипторе задачи, выбирает из очереди готовых задач следующую и запускает ее на выполнение, предварительно загрузив ее контекст.

При истечении кванта процессорного времени задача возвращается в очередь.

Дисциплина RR

Для интерактивных вычислений желательно, прежде всего, обеспечить приемлемое время реакции системы и равенство в обслуживании, если система является мультитерминальной (многозадачной). При этом можно пожелать, чтобы некоторые приложения, выполняясь без непосредственного участия программиста, но тем не менее гарантированно получали необходимую им долю процессорного времени.

Для решения подобных проблем используется дисциплина обслуживания, называемая *RR (round robin, круговая, карусельная)*, и приоритетные методы обслуживания. Дисциплина обслуживания RR предполагает, что каждая задача получает процессорное время порциями (квантами времени, q). После окончания кванта времени q задача снимается с процессора, и он передается следующей задаче. Снятая задача ставится в конец очереди задач, готовых к выполнению (рис. 3.4).

Для оптимальной работы системы необходимо правильно выбрать закон, по которому кванты времени выделяются задачам.

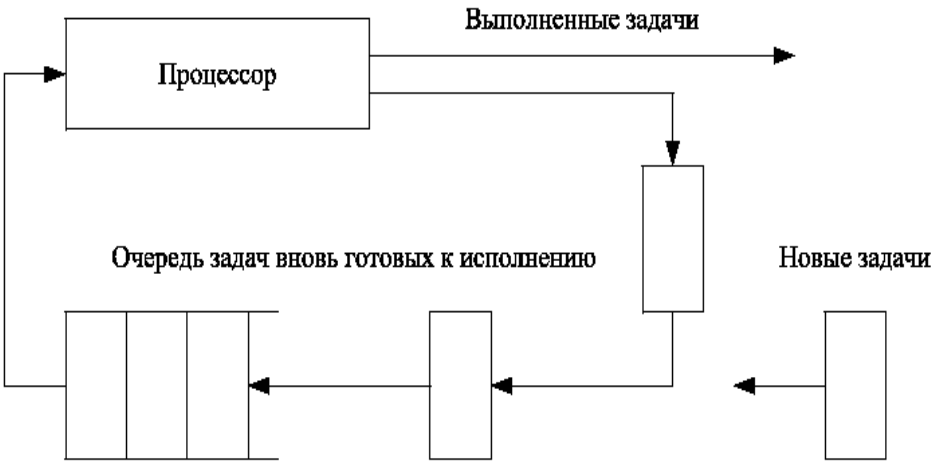

Рис. 3.4. Карусельная дисциплина диспетчеризации (round robin)

Величина кванта времени выбирается как компромисс между приемлемым временем реакции системы на запросы пользователей (с тем, чтобы их простейшие запросы не вызывали длительного ожидания) и накладными расходами на частую смену контекста задач. Если величина q велика, то при увеличении очереди готовых к выполнению задач реакция системы станет плохой. Если же величина мала, то относительная доля накладных расходов на переключения между исполняющимися задачами станет большой и это ухудшит производительность системы.

В своей простейшей реализации дисциплина карусельной диспетчеризации предполагает, что все задачи имеют одинаковый приоритет. Если же необходимо ввести механизм приоритетного обслуживания, то это, как правило, делается за счет организации нескольких очередей. Процессорное время будет предоставляться в первую очередь тем задачам, которые стоят в самой привилегированной очереди. Если она пустая, то диспетчер задач начнет просматривать остальные очереди.

Недостаток Round-robin: процессы с интенсивным вв/выв (т. е. заблокированные в ожидании вв/выв) полностью не используют свой квант времени, поэтому процессы с интенсивным использованием ЦП получают преимущество.

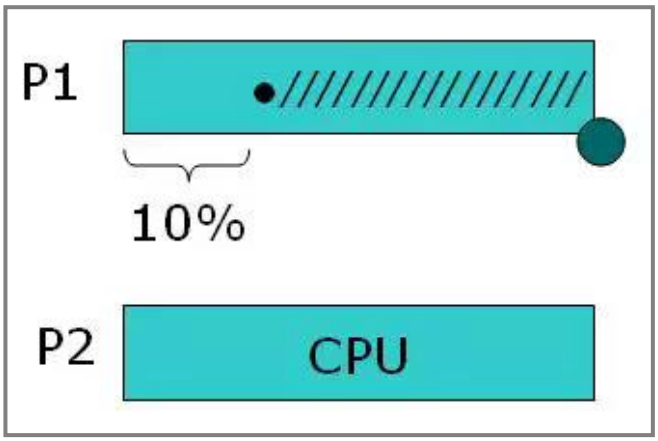

Рис. 3.5. Недостаток Round-robin

Например, есть два процесса P1 и P2.

Процесс P1 ожидает ввод/вывод в точке (●), пока этот вв/выв не завершится часть отмеченного штриховкой времени процесс P1 потратит «впустую», он вытеснится только в зеленой точке по истечении кванта времени.

P2 в это время активно использует ЦП, например, считает.

Проблема RR в том, что не учитываются задержки, и полезное время работы P1 составляет только 10 %.

Диспетчеризация задач с использованием динамических приоритетов и многоуровневые очереди

Алгоритмы SJN и SRT представляют собой частные случаи приоритетного планирования. При приоритетном планировании каждому процессу присваивается определенное числовое значение – приоритет, в соответствии с которым ему выделяется процессор. Процессы с одинаковыми приоритетами планируются в порядке FCFS. Для алгоритма SJN в качестве такого приоритета выступает оценка требуемого времени. Чем меньше значение этой оценки, тем более высокий приоритет имеет процесс..

Алгоритмы назначения приоритетов процессов могут опираться как на внутренние параметры, связанные с происходящим внутри вычислительной системы, так и на внешние по отношению к ней. К внутренним параметрам относятся различные количественные и качественные характеристики про-

цесса такие как: ограничения по времени использования процессора, требования к размеру памяти, число открытых файлов и используемых устройств ввода-вывода, отношение средних продолжительностей операций ввода-вывода к времени CPU и т. д. Алгоритмы SJN и SRT используют внутренние параметры. В качестве внешних параметров могут выступать важность процесса для достижения каких-либо целей, стоимость оплаченного процессорного времени и другие политические факторы.

Планирование с использованием приоритетов может быть как вытесняющим, так и невытесняющим. При вытесняющем планировании процесс с более высоким приоритетом, появившийся в очереди готовых процессов, вытесняет исполняющийся процесс с более низким приоритетом. В случае невытесняющего планирования он просто становится в начало очереди готовых процессов.

При выполнении программ, реализующих какие-либо задачи контроля и управления (что характерно, прежде всего, для систем реального времени), может случиться такая ситуация, когда одна или несколько задач не могут быть реализованы (решены) в течение длительного промежутка времени из-за возросшей нагрузки в вычислительной системе. Потери, связанные с невыполнением таких задач, могут оказаться больше, чем потери от невыполнения программ с более высоким приоритетом. При этом оказывается целесообразным временно изменить приоритет «аварийных» задач (для которых истекает отпущенное для них время обработки). После выполнения этих задач их приоритет восстанавливается. Поэтому почти в любой ОС реального времени имеются средства для изменения приоритета программ. Есть такие средства и во многих ОС, которые не относятся к классу ОСРВ. Введение механизмов динамического изменения приоритетов позволяет реализовать более быструю реакцию системы на короткие запросы пользователей, что очень важно при интерактивной работе, но при этом гарантировать выполнение любых запросов.

Планирование процессов в ОС UNIX

Краткая диаграмма состояний процессов в операционной системе UNIX изображена на рисунке 3.6.

Рис. 3.6. Состояния процессов в операционной системе UNIX

Как видим, состояние процесса исполнение расщепилось на 2 состояния: исполнение в режиме ядра и исполнение в режиме пользователя. В состоянии исполнение в режиме пользователя процесс выполняет прикладные инструкции пользователя. В состоянии исполнение в режиме ядра выполняются инструкции ядра операционной системы в контексте текущего процесса (например, при обработке системного вызова или прерывания). Из состояния исполнение в режиме пользователя процесс не может непосредственно перейти в состояния ожидание, готовность и закончил исполнение. Такие переходы возможны только через промежуточное состояние исполняется в режиме ядра. Точно также запрещен прямой переход из состояния готовность в состояние исполнение в режиме пользователя. Приведенная выше диаграмма состояний процессов в Linux не является полной. Она показывает только состояния, для понимания которых достаточно уже

полученных знаний. Полную диаграмму состояний процессов в операционной системе UNIX можно найти в книге Баха «Архитектура операционной системы UNIX» (рис. 6.1.).

В системе UNIX System V Release 4 реализована вытесняющая многозадачность, основанная на использовании приоритетов и квантования.

Все процессы разбиты на несколько групп, называемых классами приоритетов. Каждая группа имеет свои характеристики планирования процессов.

Созданный процесс наследует характеристики планирования процесса-родителя, которые включают класс приоритета и величину приоритета в этом классе. Процесс остается в данном классе до тех пор, пока не будет выполнен системный вызов, изменяющий его класс.

В UNIX System V Release 4 возможно включение новых классов приоритетов при инсталляции системы.

Рассмотрим как реализован механизм динамических приоритетов в ОС UNIX, которая, как известно, не относится к ОСРВ. Приоритет процесса вычисляется следующим образом. Во-первых, в вычислении участвуют значения двух полей дескриптора процесса – p_nice и p_cpu. Первое из них назначается пользователем явно или формируется по умолчанию с помощью системы программирования. Второе поле формируется диспетчером задач (планировщиком разделения времени) и называется системной составляющей или текущим приоритетом. Другими словами, каждый процесс имеет два атрибута приоритета, с учетом которого и распределяется между исполняющимися задачами процессорное время: текущий приоритет, на основании которого происходит планирование, и заказанный относительный приоритет, называемый nice number (или просто nice).

Схема нумерации (числовых значений) текущих приоритетов различна для различных версий UNIX. Например, более высокому значению текущего приоритета может соответствовать более низкий фактический приоритет планирования. Разделение между приоритетами режима ядра и задачи также зависит от версии. Рассмотрим частный случай, когда текущий приоритет

процесса варьируется в диапазоне от 0 (низкий приоритет) до 127 (наивысший приоритет). Процессы, выполняющиеся в режиме задачи, имеют более низкий приоритет, чем в режиме ядра. Для режима задачи приоритет меняется в диапазоне 0–65, для режима ядра – 66–95 (системный диапазон). Процессы, приоритеты которых лежат в диапазоне 96–127, являются процессами с фиксированным приоритетом, не изменяемым операционной системой, и предназначены для поддержки приложений реального времени. Диапазон приоритетов процесса представлен на рисунке 3.7.

Приоритетный класс	**Выбор планировщика**	**Глобальное значение приоритета**
Реальное время (real time)	первый	127 . . . 96
Системные процессы (system)	. . . .	95 66
Процессы разделения времени (time-shared)	. . . . последний	65 0

Рис. 3.7. Диапазон приоритетов процесса

Планировщик использует следующие характеристики для процессов разделения времени:

ts_globpri	содержит величину глобального приоритета
ts_quantum	определяет количество тиков системных часов, которые отводятся процессу до его вытеснения

ts_tqexp	системная часть приоритета, назначаемая процессу при истечении его кванта времени
ts_slpret	системная составляющая приоритета, назначаемая процессу после выхода его из состояния ожидания; ожидающим процессам дается высокий приоритет, так что они быстро получают доступ к процессору после освобождения ресурса
ts_maxwaite	максимальное число секунд, которое разрешается потреблять процессу; если этот квант времени истекает до кванта ts_quantum, то, следовательно, считается, что процесс ведет себя по-джентльменски, и ему назначается более высокий приоритет
ts_lwait	величина системной части приоритета, назначаемая процессу, если истекает ts_maxwait секунд

Для процессов разделения времени в дескрипторе процесса proc имеется указатель на структуру, специфическую для данного класса процесса. Эта структура состоит из полей, используемых для вычисления глобального приоритета:

ts_timeleft	число тиков, остающихся в кванте процесса
ts_cpupri	системная часть приоритета процесса
ts_uprilim, ts_upri	верхний предел и текущее значение пользовательской части приоритета. Эти две переменные могут модифицироваться пользователем
ts_nice	используется для обратной совместимости с системным вызовом nice. Она содержит текущее значение величины nice, которая влияет на результирующую величину приоритета. Чем выше эта величина, тем меньше приоритет

В версии SVR4 нет поддержки многонитевой (multithreading) организации процессов на уровне ядра, хотя и есть два системных вызова для организации нитей в пользовательском режиме. Во многих коммерческих реализациях UNIX, базирующихся на кодах SVR4, в ядро включена поддержка нитей за счет собственной модификации исходных текстов SVR4.

Обслуживаемые по алгоритму FIFO потоки реального времени имеют наивысшие приоритеты и не могут вытесняться другими потоками, за исключением такого же потока реального времени FIFO с более высоким приоритетом, перешедшего в состояние готовности [3].

Процессу, ожидающему недоступного в данный момент ресурса, система определяет значение приоритета сна, выбираемое ядром из диапазона системных приоритетов и связанное с событием, вызвавшее это состояние. Когда процесс пробуждается, ядро устанавливает значение текущего приоритета процесса равным приоритету сна. Поскольку приоритет такого процесса находится в системном диапазоне и выше, чем приоритет режима задачи, вероятность предоставления процессу вычислительных ресурсов весьма велика. Такой подход позволяет, в частности, быстро завершить системный вызов, выполнение которого, в свою очередь, может блокировать некоторые системные ресурсы.

После завершения системного вызова перед возвращением в режим задачи ядро восстанавливает приоритет режима задачи, сохраненный перед выполнением системного вызова. Это может привести к понижению приоритета, что, в свою очередь, вызовет переключение контекста.

Текущий приоритет процесса в режиме задачи p_priuser, как мы только что отметили, зависит от значения nice number и степени использования вычислительных ресурсов p_cpu:

p_priuser = a * p_nice – b * p_cpu

Задача планировщика разделения времени – справедливо распределить вычислительный ресурс между конкурирующими процессами. Для принятия решения о выборе следующего запускаемого процесса планировщику необходима информация об использовании процессора. Эта составляющая приоритета уменьшается обработчиком прерываний таймера по каждому «тику» таймера. Таким образом, пока процесс выполняется в режиме задачи, его текущий приоритет линейно уменьшается.

Каждую секунду ядро пересчитывает текущие приоритеты процессов, готовых к запуску (приоритеты которых меньше некоторого порогового значения, в нашем примере эта величина равна 65), последовательно увеличивая их. Это осуществляется за счет того, что ядро последовательно уменьшает отрицательную компоненту времени использования процессора. Как результат, эти действия приводят к перемещению процессов в более приоритетные очереди и повышают вероятность их последующего запуска.

Возможно использование следующей формулы:

p_cpu = p_cpu / 2.

Это правило проявляет недостаток нивелирования приоритетов при повышении загрузки системы. Происходит это потому, что в таком случае каждый процесс получает незначительный объем вычислительных ресурсов и, следовательно, имеет малую составляющую p_cpu, которая еще более уменьшается благодаря формуле пересчета величины p_cpu. В результате степень использования процессора перестает оказывать заметное влияние на приоритет, и низкоприоритетные процессы (то есть процессы с высоким значением nice number) практически «отлучаются» от вычислительных ресурсов системы.

В некоторых версиях ОС UNIX для пересчета значения p_cpu используется другая формула:

p_cpu – p_cpu * (2 * load) / (2 * load + l).

Здесь параметр load равен среднему числу процессов, находившихся в очереди на выполнение за последнюю секунду, и характеризует среднюю загрузку системы за этот период времени. Такой алгоритм позволяет частично избавиться от недостатка планирования по формуле p_cpu = p_cpu/2, поскольку при значительной загрузке системы уменьшение p_cpu при пересчете будет происходить медленнее.

Пример диспетчеризации процессов представлен на рисунке 3.8. Если предположить, что первым запускается процесс А и ему выделяется квант времени, он выполняется в течение 1 секунды: за это время таймер посылает

системе 60 прерываний и столько же раз программа обработки прерываний увеличивает для процесса А значение поля, содержащего показатель ИЦП (использование центрального процессора) с 0 до 60. По прошествии секунды ядро переключает контекст и, произведя пересчет приоритетов для всех процессов, выбирает для выполнения процесс В. В течение следующей секунды программа обработки прерываний по таймеру 60 раз повышает значение поля ИЦП для процесса В, после чего ядро пересчитывает параметры диспетчеризации для всех процессов и вновь переключает контекст. Процедура повторяется многократно, сопровождаясь поочередным запуском процессов на выполнение [5].

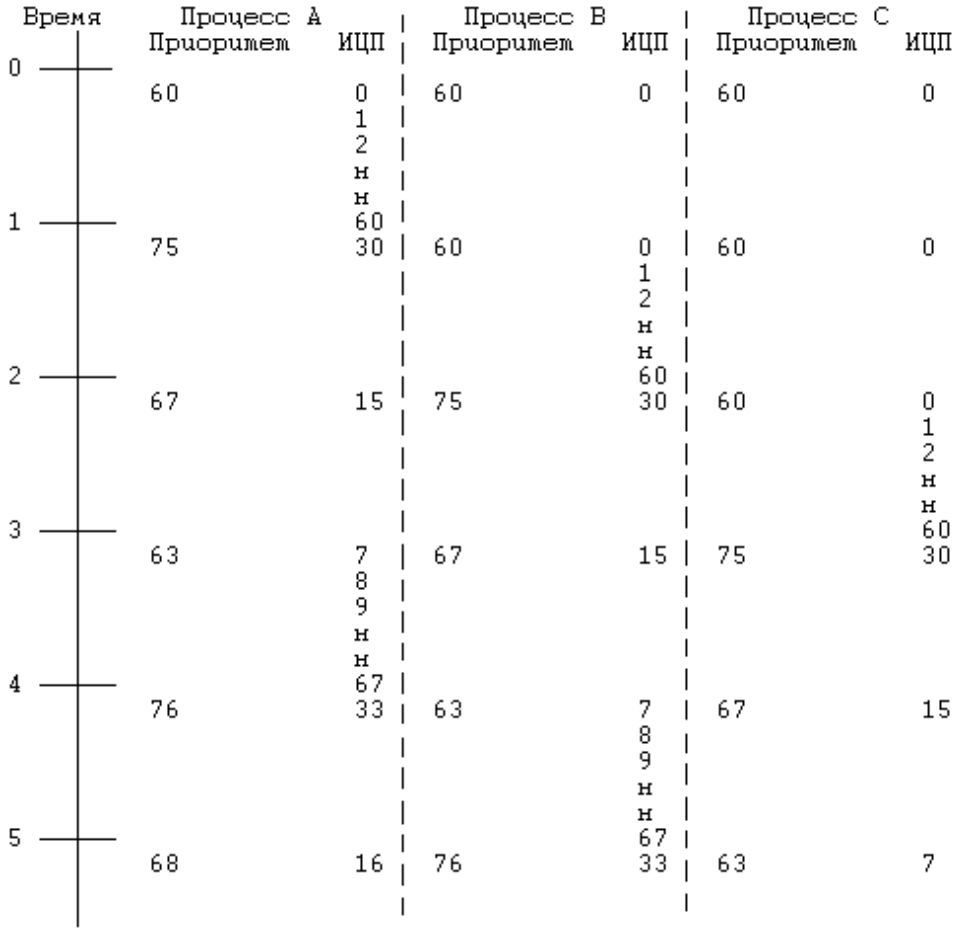

Рис. 3.8. Пример диспетчеризации процессов

Описанные алгоритмы планирования позволяют учесть интересы низкоприоритетных процессов, так как в результате длительного ожидания оче-

реди на запуск приоритет таких процессов увеличивается, соответственно увеличивается и вероятность запуска. Эти алгоритмы также обеспечивают более вероятный выбор планировщиком интерактивных процессов по отношению к вычислительным (фоновым). Такие задачи, как командный интерпретатор или редактор, большую часть времени проводят в ожидании ввода, имея, таким образом, высокий приоритет (приоритет сна). При наступлении ожидаемого события (например, пользователь осуществил ввод данных) им сразу же предоставляются вычислительные ресурсы. Фоновые процессы, потребляющие значительные ресурсы процессора, имеют высокую составляющую p_cpu, как следствие, менее высокий приоритет.

Планирование процессов в ОС Windows NT

В Windows NT реализована вытесняющая многозадачность, при которой операционная система не ждет, когда поток сам захочет освободить процессор, а принудительно снимает его с выполнения после того, как он израсходовала отведенное ему время (квант), или если в очереди готовых появилась поток с более высоким приоритетом. При такой организации разделения процессора ни одна поток не займет процессор на очень долгое время.

В ОС Windows NT поток в ходе своего существования может иметь одно из шести состояний (рис. 3.9). Жизненный цикл потока начинается в тот момент, когда программа создает новый поток. Запрос передается NT executive, менеджер процессов выделяет память для объекта-потока и обращается к ядру, чтобы инициализировать объект-поток ядра. После инициализации поток проходит через следующие состояния:

Готовность. При поиске нити на выполнение диспетчер просматривает только нити, находящиеся в состоянии готовности, у которых есть все для выполнения, но не хватает только процессора.

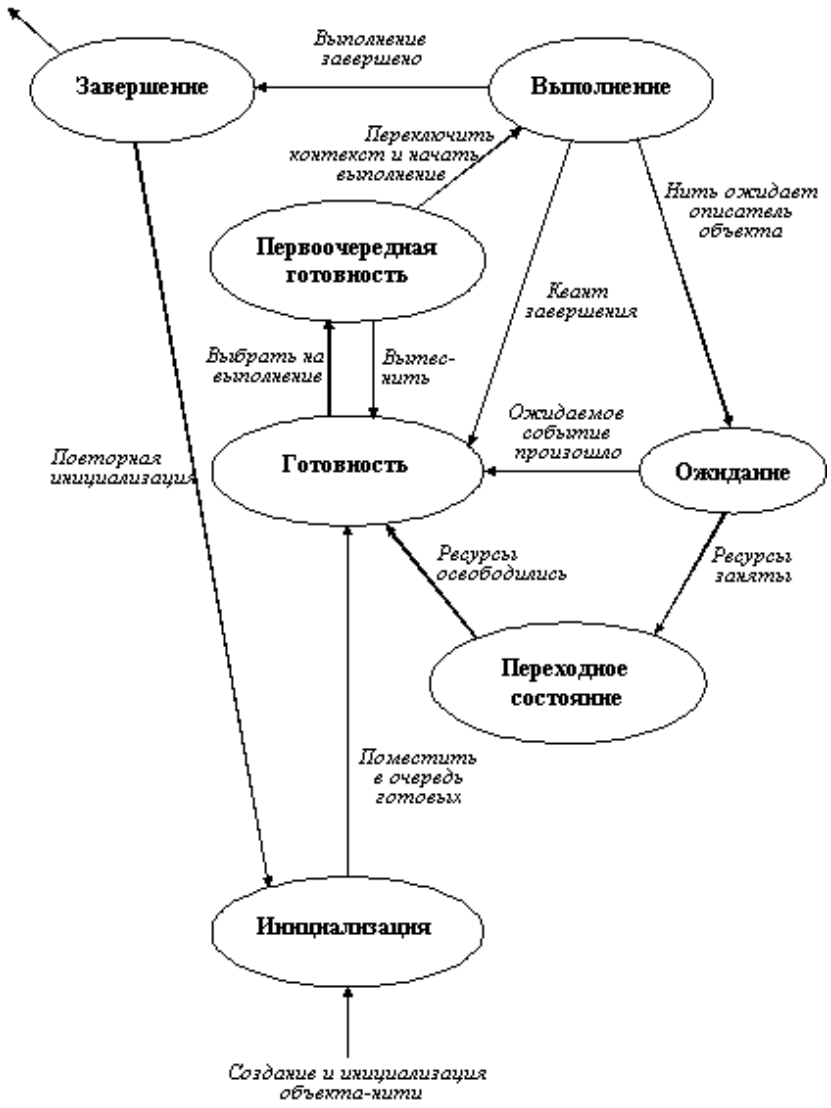

Рис. 3.9. Граф состояний потоков в Windows NT

Первоочередная готовность (standby). Для каждого процессора системы выбирается одна поток, которая будет выполняться следующей (самая первая поток в очереди). Когда условия позволяют, происходит переключение на контекст этой нити

Выполнение. Как только происходит переключение контекстов, поток переходит в состояние выполнения и находится в нем до тех пор, пока либо ядро не вытеснит ее из-за того, что появилась более приоритетная поток или закончился квант времени, выделенный этой нити, либо поток завершится вообще, либо она по собственной инициативе перейдет в состояние ожидания.

Ожидание. Поток может входить в состояние ожидания несколькими способами: поток по своей инициативе ожидает некоторый объект для того, чтобы синхронизировать свое выполнение; операционная система (например, подсистема ввода-вывода) может ожидать в интересах нити; подсистема окружения может непосредственно заставить поток приостановить себя. Когда ожидание нити подойдет к концу, она возвращается в состояние готовности.

Переходное состояние. Поток входит в переходное состояние, если она готова к выполнению, но ресурсы, которые ей нужны, заняты. Например, страница, содержащая стек нити, может быть выгружена из оперативной памяти на диск. При освобождении ресурсов поток переходит в состояние готовности.

Завершение. Когда выполнение нити закончилось, она входит в состояние завершения. Находясь в этом состоянии, поток может быть либо удалена, либо не удалена. Это зависит от алгоритма работы менеджера объектов, в соответствии с которым он и решает, когда удалять объект. Если executive имеет указатель на объект-поток, то она может быть инициализирована и использована снова.

Диспетчер ядра использует для определения порядка выполнения нитей алгоритм, основанный на приоритетах, в соответствии с которым каждой нити присваивается число – приоритет, и нити с более высоким приоритетом выполняются раньше нитей с меньшим приоритетом. В самом начале поток получает приоритет от процесса, который создает ее. В свою очередь, процесс получает приоритет в тот момент, когда его создает подсистема той или иной прикладной среды. Значение базового приоритета присваивается процессу системой по умолчанию или системным администратором. Поток наследует этот базовый приоритет и может измепоток его, немного увеличив или уменьшив. На основании получившегося в результате приоритета, называемого приоритетом планирования, начинается выполнение нити. В ходе выполнения приоритет планирования может меняться.

Аналогичные механизмы имеют место и в таких ОС, как Windows NT или OS/2. Правда, алгоритмы изменения приоритета задач в этих системах иные. Например, в Windows NT каждый поток (тред) имеет базовый уровень приоритета, который лежит в диапазоне от двух уровней ниже базового приоритета процесса, его породившего, до двух уровней выше этого приоритета.

На рисунке 3.10 показан динамический приоритет потока, нижней границей которого является базовый приоритет потока, а верхняя – зависит от вида работ, исполняемых потоком.

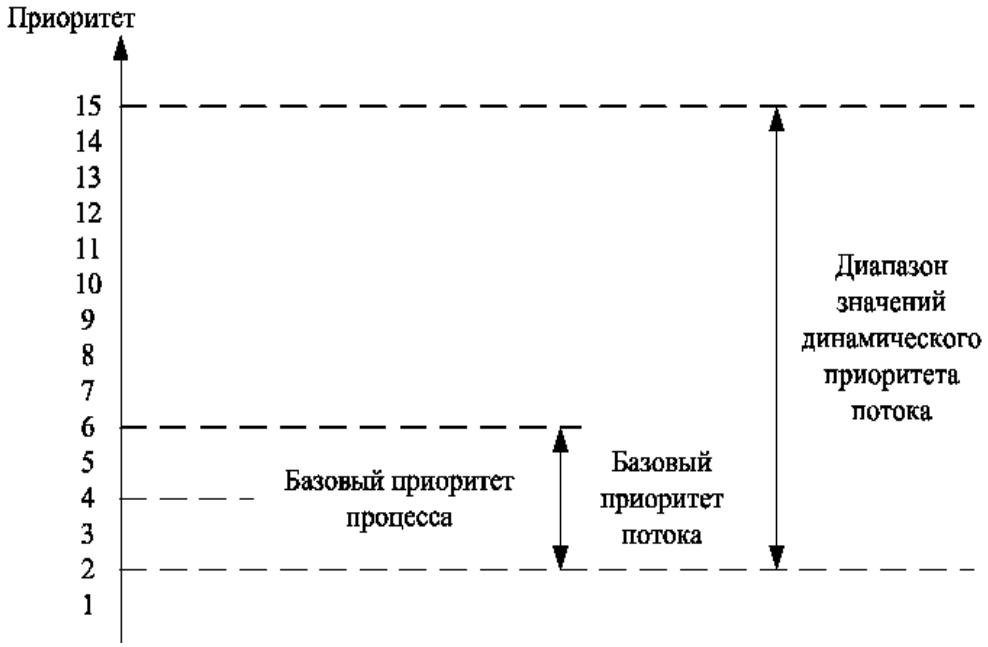

Рис. 3.10. Схема динамического изменения приоритетов в Windows NT

Для определения порядка выполнения потоков диспетчер использует систему приоритетов, направляя на выполнение потоки с высоким приоритетом раньше потоков с низкими приоритетами.

Существует группа очередей – по одной для каждого приоритета. *Windows NT* поддерживает 32 уровня приоритетов; потоки делятся на два класса: реального времени и переменного приоритета. Потоки реального времени, имеющие приоритеты от 16 до 31 – это высокоприоритетные потоки, используемыми программами с критическим временем выполнения, то есть требующие немедленного внимания системы (по терминологии *Microsoft*).

Диспетчер задач просматривает очереди, начиная с самой приоритетной. При этом если очередь пустая, то есть, нет готовых к выполнению задач с таким приоритетом, осуществляется переход к следующей очереди. Следо-

вательно, если есть задачи, требующие процессор немедленно, они будут обслужены в первую очередь. Для собственно системных модулей, функционирующих в статусе задач, зарезервирована очередь с номером 0.

Большинство потоков в системе относятся к классу переменного приоритета с уровнями приоритета (номером очереди) от 1 до 15. Эти очереди используются потоками с переменным приоритетом, так как диспетчер задач корректирует их приоритеты по мере выполнения задач для оптимизации отклика системы. Диспетчер приостанавливает исполнение текущего потока после того, как тот израсходует свой квант времени. При этом если прерванный тред – это поток переменного приоритета, то диспетчер задач понижает его приоритет на единицу и перемещает в другую очередь. Таким образом, приоритет потока, выполняющего много вычислений, постепенно понижается (до значения его базового приоритета). С другой стороны, диспетчер повышает приоритет потока после освобождения задачи (потока) из состояния ожидания. Величина добавки зависит от типа события, которого ожидал заблокированный тред. Так, например, поток, ожидавший ввода очередного байта с клавиатуры, получает большую добавку к значению своего приоритета, чем процесс ввода/вывода, работавший с дисковым накопителем. Однако в любом случае значение приоритета не может достигнуть 16.

Планирование процессов в OS/2

В операционной системе OS/2 схема динамической приоритетной диспетчеризации несколько иная, хотя и похожа на рассмотренную. В OS/2 имеются четыре класса задач. И для каждого класса задач имеется своя группа приоритетов с интервалом значений от 0 до 31. Итого, 128 различных уровней и, соответственно, 128 возможных очередей готовых к выполнению задач (тредов, потоков).

Класс задач, имеющих самые высокие значения приоритета, называется критическим (time critical). Этот класс предназначается для задач, которые мы в обиходе называем задачами реального времени, то есть для них должен

быть обязательно предоставлен определенный минимум процессорного времени. Наиболее часто встречающимися задачами, которые относят к этому классу, являются задачи коммуникаций (например, задача управления последовательным портом, принимающим биты с коммутируемой линии, к которой подключен модем, или задачи управления сетевым оборудованием). Если такие задачи не получат управление в нужный момент времени, то сеанс связи может прерваться.

Следующий класс задач имеет название приоритетного. Поскольку к этому классу относят задачи, которые выполняют по отношению к остальным задачам роль сервера, то его еще иногда называют серверным. Приоритет таких задач должен быть выше, это будет гарантировать, что запрос на некоторую функцию со стороны обычных задач выполнится сразу, а не будет дожидаться, пока до него дойдет очередь на фоне других пользовательских приложений.

Большинство задач относят к обычному классу, его еще называют регулярным, или стандартным. По умолчанию система программирования порождает задачу, относящуюся именно к этому классу. Наконец, существует еще класс фоновых задач, называемый в OS/2 остаточный. Программы этого класса получают процессорное время только тогда, когда нет задач из других классов, которым сейчас нужен процессор. В качестве примера такой задачи можно привести программу проверки электронной почты.

Внутри каждого из вышеописанных классов задачи, имеющие одинаковый уровень приоритета, выполняются в соответствии с дисциплиной round-robin. Переход от одного треда к другому происходит либо по окончании отпущенного ему кванта времени, либо по системному прерыванию, передающему управление задаче с более высоким приоритетом (таким образом, система вытесняет задачи с более низким приоритетом для выполнения задач с более высоким приоритетом и может обеспечить быструю реакцию на важные события).

OS/2 самостоятельно изменяет приоритет выполняющихся программ независимо от уровня, установленного самим приложением. Этот механизм называется повышением приоритета. Операционная система изменяет приоритет задачи в следующих трех случаях:

– увеличение приоритета активной задачи (foreground boost). Приоритет задачи автоматически увеличивается, когда она становится активной. Это снижает время реакции активного приложения на действия пользователя по сравнению с фоновыми программами;

– увеличение приоритета ввода/вывода (input/output boost). По завершении операции ввода/вывода задача получает самый высокий уровень приоритета ее класса. Таким образом обеспечивается завершение всех незаконченных операций ввода/вывода;

– увеличение приоритета «забытых» задач (starvation boost). Если задача не получает управление в течение достаточно долгого времени, диспетчер задач OS/2 временно присваивает ей уровень приоритета, не превышающий критический. В результате переключение на такую «забытую» программу происходит быстрее. После выполнения приложения в течение одного кванта времени его приоритет вновь снижается до остаточного. В сильно загруженных системах этот механизм позволяет программам с остаточным приоритетом работать хотя бы в краткие интервалы времени. В противном случае они вообще никогда бы не получили управление.

4. ПЛАНИРОВАНИЕ В МНОГОПРОЦЕССОРНЫХ СИСТЕМАХ

Многопроцессорные системы представляют собой основной путь построения ВС сверхвысокой производительности. При создании таких ВС возникает много сложных проблем, таких как:

– распараллеливание вычислительного процесса (программ) для эффективной загрузки процессоров системы;

– преодоление конфликтов при попытках нескольких процессоров использовать один и тот же ресурс системы (например, некоторый модуль памяти) и уменьшение влияния конфликтов на производительность системы;

– осуществление быстродействующих экономичных по аппаратурным затратам межмодульных связей.

Указанные вопросы необходимо учитывать при выборе структуры МПС.

Планирование в операционных системах – это предоставление доступа к процессору процессов или потоков в зависимости от алгоритмов планирования.

Часть операционной системы, отвечающая за планирование, называется планировщиком.

В различных операционных системах используются разные алгоритмы планирования. Это зависит от назначения системы, ее возможностей и других факторов, в том числе и от видения разработчиков.

Планирование бывает необходимо в разные моменты времени в зависимости от ситуации, складывающейся в системе.

Например, планировщик начинает свою работу при создании или завершении процесса, при блокировании процесса на операции ввода-вывода или на семафоре, при возникновении прерывания от таймера или от внешнего устройства и т. д.

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 [3].

У планирования в многопроцессорных системах те же цели, что и в однопроцессорных. Оно должно обеспечивать максимальную производительность и минимальное время реагирования для всех процессов. Кроме того, система должна гарантировать соблюдение приоритетов процессов. В отличие от алгоритмов планирования, используемых в однопроцессорных системах, алгоритмы для многопроцессорных систем должны определять не только порядок выполнения процессов, но и то, на каких процессорах эти процессы должны выполняться. За счет этого растет сложность алгоритмов. Алгоритмы планирования для многопроцессорных систем должны гарантировать, что процессоры в системе не будут простаивать, если есть процессы, ожидающие выполнения.

Когда нужно определить, на каком процессоре должен выполняться процесс, планировщик должен принять во внимание несколько моментов: например, в некоторых стратегиях планирования целью является максимальное распараллеливание работы, полное использование возможностей системы по одновременному выполнению программ. Взаимодействующие процессы в системах часто объединяются в задачи. Параллельное выполнение процессов задачи позволяет повысить производительность, поскольку процессы выполняются действительно одновременно. Существует несколько алгоритмов планирования с временным разделением, пытающихся воспользоваться такой параллельностью при планировании размещения взаимодействующих процессов на разные процессоры. Это позволяет процессам более эффективно организовать свою одновременную работу. Другие стратегии сосредотачиваются на привязке задач к процессорам, т. е. взаимосвязи процесса и процессора, его локальной памяти и кэша.

Процесс, привязанный к процессору, выполняется на этом процессоре в течении большей части своего жизненного цикла. Алгоритмы планирования, пытающиеся разместить процесс на один процессор на все время выполнения этого процесса, обеспечивают мягкую привязку, а алгоритмы, размещающие процесс всегда на одном и том же процессоре, обеспечивают жесткую привязку.

Алгоритмы планирования с пространственным разделением пытаются добиться максимальной привязки задач к процессорам, направляя взаимодействующие процессы для выполнения на один процессор. Эти алгоритмы подразумевают, что взаимодействующие процессы будут работать с одними и теми же данными, и эти данные скорее всего удастся разместить в кэше процессоров и в их локальной памяти. Поэтому алгоритмы планирования с пространственным разделением увеличивают частоту успешных обращений к кэшу и локальной памяти. Однако они могут ограничивать производительность, поскольку при их использовании взаимодействующие процессы обычно не могут исполняться одновременно.

Алгоритмы планирования в многопроцессорных системах обычно делятся на задачно-ориентированные и задачно-независимые.

Во многих алгоритмах планирования для многопроцессорных систем процессы организуются в глобальную очередь выполнения. Каждая такая очередь содержит все процессы в системе, которые готовы к выполнению. Глобальные очереди выполнения можно использовать для сортировки процессов по их приоритетам, по задачам, к которым они относятся, или по тому, какой процесс выполнялся последним.

В системах могут также использоваться процессорные очереди выполнения. Использование таких очередей характерно для слабо связанных систем, в которых нужно добиться как можно большего уровня успешных обращений к кэшу и локальной памяти. В процессорных очередях выполнения процессы ассоциируются с определенным процессором, и система реализует заданный алгоритм планирования для каждого процессора. В некоторых си-

стемах используются узловые очереди выполнения. В таких системах в каждом узле может находиться по несколько процессоров. Эти очереди эффективны в системах, в которых процессы связаны с определенными группами процессоров. С проблемой планирования связана и проблема миграции процессов, т. е. переноса процессов из одной процессорной или узловой очереди в другую.

4.1. Задачно-независимые алгоритмы планирования

Задачно-независимые алгоритмы требуют для своего использования минимальных накладных расходов, поскольку они не пытаются добиться максимального распараллеливания или воспользоваться привязкой задачи к процессору. Эти алгоритмы размещают задачи и процессы для выполнения на любом доступном процессоре. В общем случае, алгоритм планирования для однопроцессорных систем можно использовать как задачно-независимый алгоритм планирования для многопроцессорных систем.

К основным задачно-независимым алгоритмам относятся FCFS, круговой алгоритм и SPF.

4.1.1. Алгоритм планирования FCFS

FCFS расшифровывается как first-come-first-served («первым пришел – первым обслужен»). В этом алгоритме используется одна глобальная очередь. Если появляется свободный процессор, планировщик размещает на этом процессоре первый процесс из очереди выполнения. Процесс выполняется до тех пор, пока он не завершится или не произойдет ошибка.

При использовании данного алгоритма все процессы выполняются в том порядке, в котором они поступают в систему, поскольку ни один из процессов не имеет преимущества перед остальными. Поэтому при использовании алгоритма FCFS исключено бесконечное ожидание. Алгоритм использует одну глобальную очередь процессов. Однако такое отношение можно счесть и неоптимальным, поскольку долго выполняющийся процесс

заставляет ждать все остальные процессы, включая и быстро выполняющиеся. Кроме того, может возникнуть ситуация, при которой процессу с высоким приоритетом придется ожидать окончания выполнения процесса с низким приоритетом.

Пример планирования с использованием данной дисциплины представлен на рисунке 4.1.

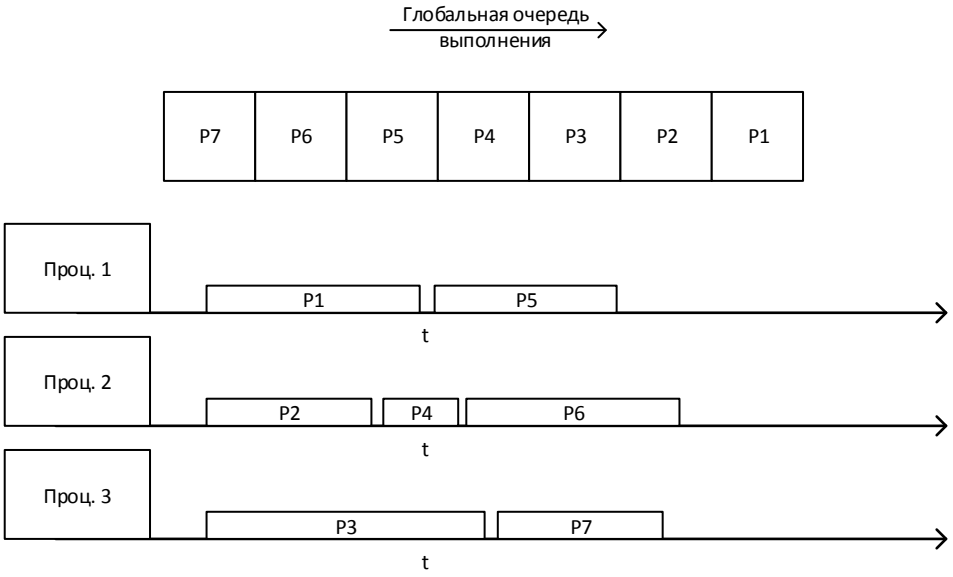

Рис. 4.1. Планирование с использованием дисциплины FCFS

Обычно алгоритм FCFS непригоден для планирования интерактивных процессов, поскольку он не может обеспечить малого времени реагирования. Однако этот алгоритм прост в реализации, и в нем нет опасности бесконечного откладывания выполнения процесса – как только процесс встал в очередь на выполнение, никакой процесс не сможет попасть в очередь впереди него.

4.1.2. Круговой алгоритм планирования

Круговой алгоритм планирования процессов (round-robin process scheduling) помещает все готовые к выполнению процессы в глобальную очередь выполнения. Этот алгоритм в многопроцессорных системах работает почти так же, как и в однопроцессорных – процесс выполняется в течение максимум одного полного кванта времени, а затем приходит очередь другого

процесса. Ранее выполнявшийся процесс помещается в конец глобальной очереди выполнения. Использование этого алгоритма также предотвращает бесконечное откладывание выполнения процесса, но он не обеспечивает высокую степень параллельности или хорошей привязки к процессорам, поскольку игнорирует связи между процессами.

Пример планирования с использованием данной дисциплины представлен на рисунке 4.2.

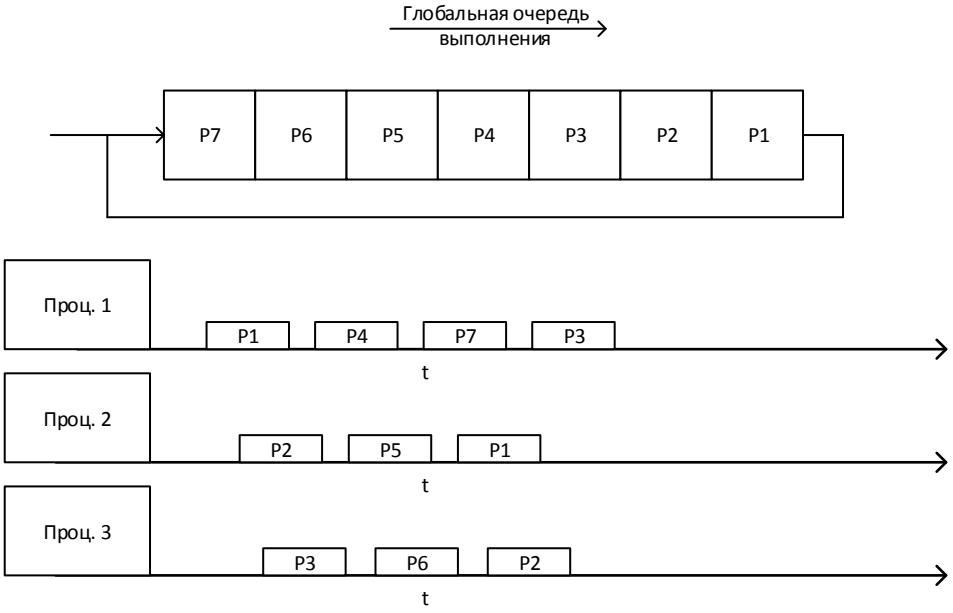

Рис. 4.2. Планирование с использованием круговой дисциплины планирования

4.1.3. Алгоритм планирования SPF

SPF расшифровывается как shortest-process-first («кратчайший процесс – первым»). При использовании этого алгоритма планирования первым запускается процесс, на полное выполнение которого требуется меньше всего времени. Как вытесняющая, так и невытесняющая версии данного алгоритма демонстрируют меньшее время ожидания для интерактивных процессов, чем алгоритм FCFS, поскольку интерактивные процессы обычно являются «короткими». Однако при использовании алгоритма SPF запуск на выполнение «длинного» процесса может бесконечно откладываться, если в системе один

за другим появляются более «короткие» процессы. Как и все задачно-независимые алгоритмы планирования, SPF не обращает внимания на параллелизм и привязку к процессорам. Планирование с использованием данной дисциплины представлено на рисунке 4.3.

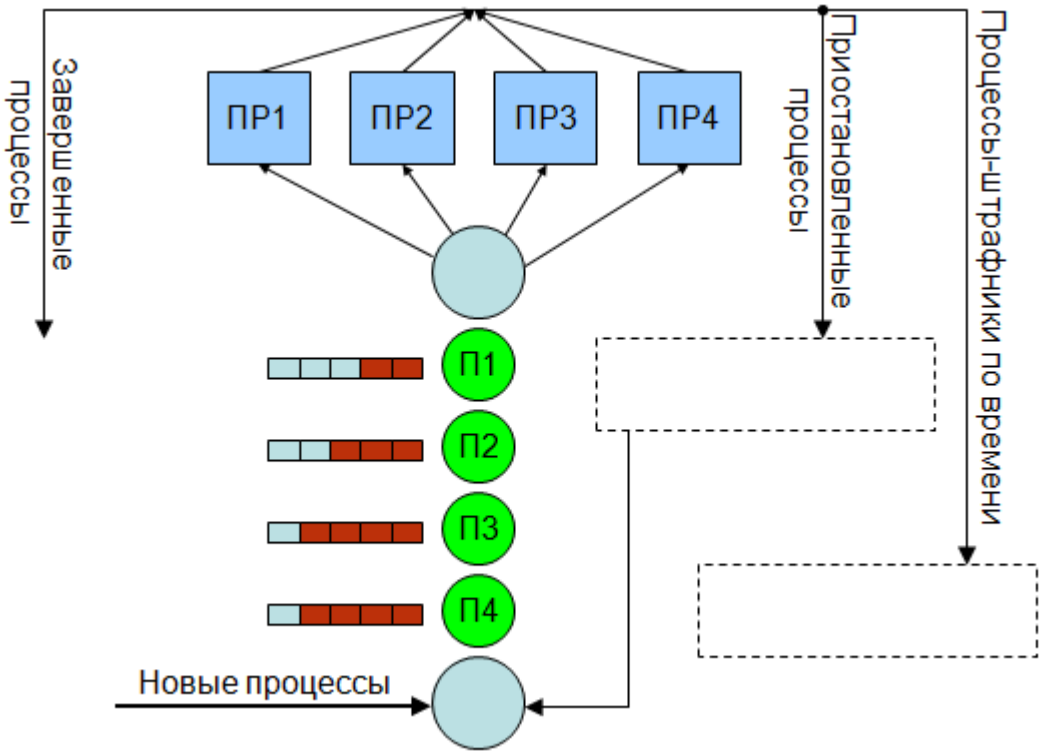

Рис. 4.3. Планирование с использованием дисциплины SPF

4.2. Задачно-ориентированные алгоритмы планирования

Задачно-ориентированные алгоритмы оценивают свойства каждой задачи и пытаются достичь максимального её распараллеливания или мягкой привязки к процессору, таким образом повышая производительность за счет снижения накладных расходов. Но они не учитывают моментов, специфичных для планирования процессов в многопроцессорных системах. Например, если два процесса, интенсивно взаимодействующие друг с другом, будут выполняться не одновременно, то существенную часть времени они будут просто ожидать ответа на свои запросы. Следовательно, производительность системы упадет. Кроме того, в большинстве многопроцессорных систем у каж-

дого процессора есть собственный кэш. Процессы из одной задачи часто обращаются к общим данным в памяти, и расположение этих процессов на одном и том же процессоре может увеличить частоту попаданий в кэш, что приведет к повышению производительности работы с памятью. В целом задачно-ориентированные алгоритмы нацелены на достижение максимального распараллеливания и привязки к процессорам за счет увеличения сложности алгоритмов.

К оосновным задачно-ориентированным алгоритмам относятся SNPF, круговой алгоритм, алгоритм совместного планирования и динамическое разделение.

4.2.1. Алгоритм планирования SNPF

SNPF расшифровывается как smallest-number-of-processes-first («наименьшее количество процессов – первым»). Этот алгоритм может быть как вытесняющим, так и невытесняющим, но в обоих вариантах он использует глобальную очередь выполнения. Приоритет задачи в такой очереди будет обратно пропорционален количеству процессов в этой задаче. Если задачи с одинаковым количеством процессов конкурируют, пытаясь занять один и тот же процессор, задача, которая дольше ожидала, получает приоритет. В невытесняющей версии алгоритма SNPF, когда процессор становится доступным, алгоритм выбирает процесс из задачи в начале очереди выполнения и позволяет ему выполняться до завершения. В вытесняющей версии, если в очереди появляется новая задача с малым количеством процессов, она получает высокий приоритет и ее процессы начинают выполняться так скоро, как это возможно. Алгоритмы SNPF улучшают параллелизм, поскольку процессы из одной задачи часто могут выполняться одновременно. Однако эти алгоритмы не улучшают привязку к процессорам. Кроме того, при их использовании возможна ситуация, когда выполнение задачи с большим количеством процессов будет откладываться бесконечно.

Планирование с использованием данной дисциплины представлено на рисунке 4.4.

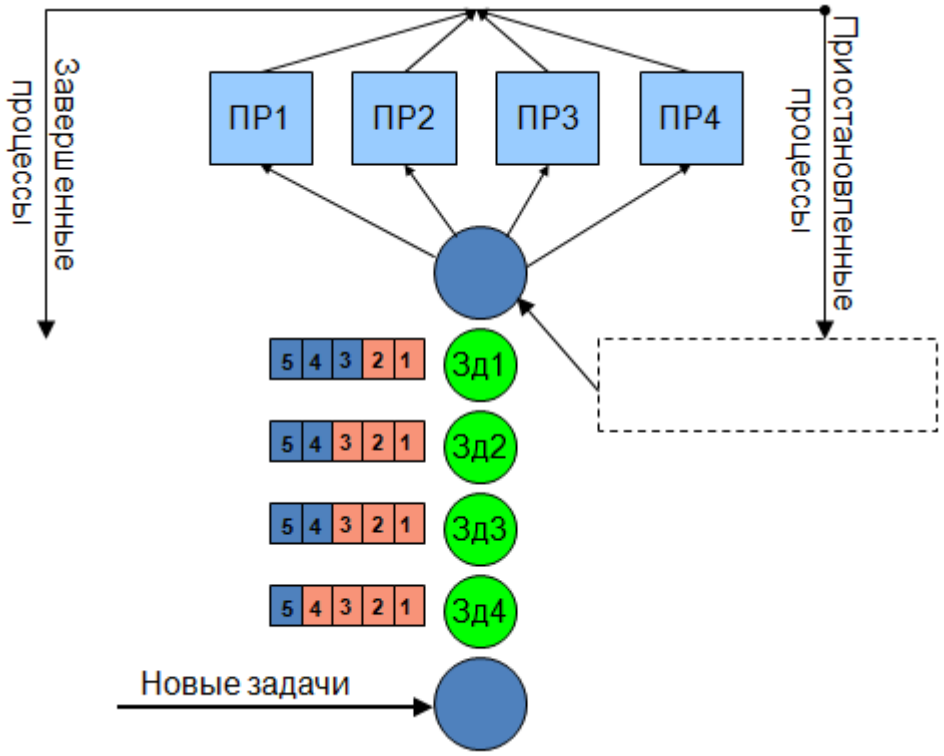

Рис. 4.4. Планирование с использованием дисциплины SNPF

4.2.2. Круговой алгоритм планирования

Круговой алгоритм планирования задач (round-robin job scheduling) помещает все готовые к выполнению задачи в глобальную очередь выполнения. Из этой очереди каждая задача назначается для выполнения группе процессоров (хотя и необязательно одной и той же группе каждый раз, когда до задачи доходит очередь выполняться). У каждой задачи есть собственная очередь процессов. Если в системе есть p процессоров, и она использует кванты продолжительности q, то задача получает в общей сложности p x q процессорного времени при ее отправке на выполнение. Обычно в задаче не точно p процессов, каждый из которых расходует целиком по одному кванту (то есть выполнение процесса может блокироваться до истечения кванта времени). Поэтому в алгоритме кругового планирования процессы задачи отправляются на выполнение, пока задача не израсходует в общей сложности p x q вре-

мени, пока она не выполнится до конца или пока не заблокируются все ее процессы. Кроме того, алгоритм может поровну разделить p x q времени между всеми процессами задачи, позволяя каждому процессу выполняться, пока он не израсходует свой лимит времени, не завершится или не заблокируется. Альтернатива: если в задаче больше, чем p процессов, то алгоритм может выбрать p процессов для выполнения в течение кванта времени длиной q.

Как и в задачно-независимой версии алгоритма, бесконечное откладывание выполнения задачи исключено. Кроме того, поскольку процессы из одной задачи будут выполняться одновременно, этот алгоритм обеспечивает хороший уровень параллелизма. Однако дополнительные накладные расходы на переключение контекста могут снизить производительность системы.

Планирование с использованием данной дисциплины представлено на рисунке 4.5.

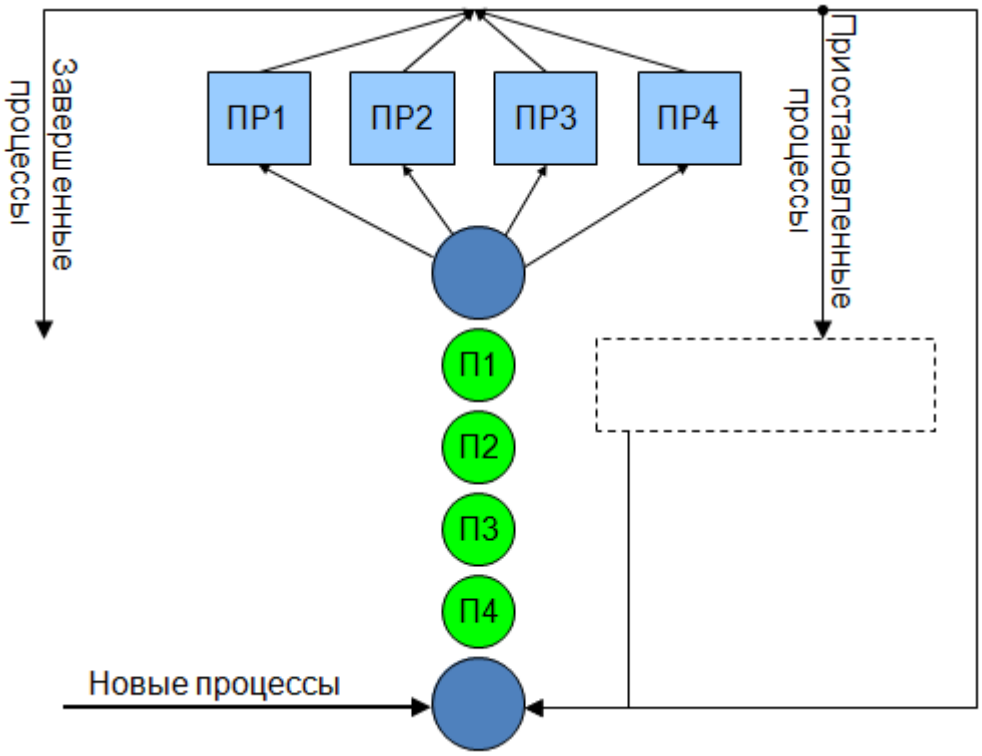

Рис. 4.5. Планирование с использованием кругового алгоритма планирования

4.2.3. Алгоритмы компланирования

Алгоритмы копланирования, или совместного планирования (coscheduling или gang scheduling) используют глобальную очередь выполнения с циклическим доступом. Цель этих алгоритмов – выполнять процессы из одной задачи одновременно, вместо того, чтобы обеспечивать их максимальную привязку к процессорам. Есть несколько реализаций алгоритмов копланирования – матричные, непрерывные и неразделяющие. В данной работе рассматривается только неразделяемая реализация, поскольку она лишена некоторых слабых мест матричных и непрерывных реализаций.

Неразделяющий алгоритм копланирования (undivided coscheduling algorithm) помещает процессы из одной задачи в смежные позиции в глобальной очереди выполнения (рис. 4.6). Планировщик использует «окно», размер которого равен количеству процессоров в системе. Все процессы, попадающие в окно, выполняются параллельно в течение максимум одного кванта. Выполнив обработку группы процессов, окно перемещается к следующей группе, которая тоже выполняется параллельно в течение одного кванта. Чтобы добиться максимальной загрузки процессоров, используется следующий подход: если процесс, попавший в окно, не может выполняться, окно расширяется на один процесс вправо, и в течение одного кванта выполняется также следующий процесс в очереди, даже если он не принадлежит к задаче, выполняемой в данный момент.

Рис. 4.6. Алгоритм компланирования

Поскольку в алгоритмах копланирования используется циклический доступ к очереди выполнения, при их использовании исключено бесконечное откладывание выполнения задачи. Кроме того, процессы из одной задачи часто выполняются одновременно. Поэтому алгоритмы копланирования позволяют программам, рассчитанным на параллельное выполнение, получать преимущество от работы в многопроцессорных системах. К сожалению, процесс может выполняться на разных процессорах каждый раз, когда до него доходит очередь, поэтому привязка его к процессору будет слабой.

4.2.4. Алгоритм динамического разделения

Динамическое разделение (dynamic partitioning) сводит к минимуму потери производительности вследствие промахов в кэше, пытаясь поддерживать максимальную привязку процессов к процессорам. Планировщик поровну распределяет процессоры системы между задачами. Количество процессоров, которое достанется каждой задаче, всегда меньше или равно количеству доступных для выполнения процессов в этой задаче.

В качестве примера можно рассмотреть систему с 32 процессорами, в которой должно выполняться три задачи – первая с 8 процессами, доступными для выполнения, вторая – с 16 процессами, а третья с 20. Если бы планировщик поровну разделял процессоры между задачами, одна задача получила бы 10 процессоров, а две остальные – по 11 каждая. В рассматриваемом примере в первой задаче только 8 процессов, поэтому планировщик выделит ей 8 процессоров и поровну разделит оставшиеся 24 процессора между двумя оставшимися задачами (по 12 процессоров каждой) (пример 1 на рис. 4.7). Поэтому каждая задача будет всегда выполняться на одних и тех же процессорах, если только в системе не появятся новые задачи. Этот алгоритм можно усовершенствовать так, что каждый процесс будет всегда выполняться на одном и том же процессоре. Если в каждой задаче содержится по одному процессу, то динамическое разделение превратится в обычный круговой алгоритм планирования.

Если в систему поступают новые задачи, система динамически изменяет распределение процессоров. Предположим, что в систему поступила четвертая задача, включающая 10 доступных для выполнения процессов (пример 2 на рис. 4.7). Первой задаче, как и раньше, выделяется 8 процессоров, но вторая и третья задачи передают часть своих процессоров четвертой задаче. Процессоры будут поровну распределены между задачами – по 8 процессоров каждой задаче. Алгоритм будет изменять выделение процессоров для каждой задачи каждый раз, когда начинается или завершается выполнение задачи, или процесс в задаче будет изменять свое состояние – из выполнения в ожидание и обратно. Хотя количество процессоров, достающихся задаче, изменяется, задача все равно выполняется на подмножестве или надмножестве процессоров, доставшихся ей изначально, и привязка к процессорам улучшается. Чтобы динамическое разделение было эффективным, прирост производительности от повышения уровня попаданий в кэше должен перевесить падение производительности от переразделения.

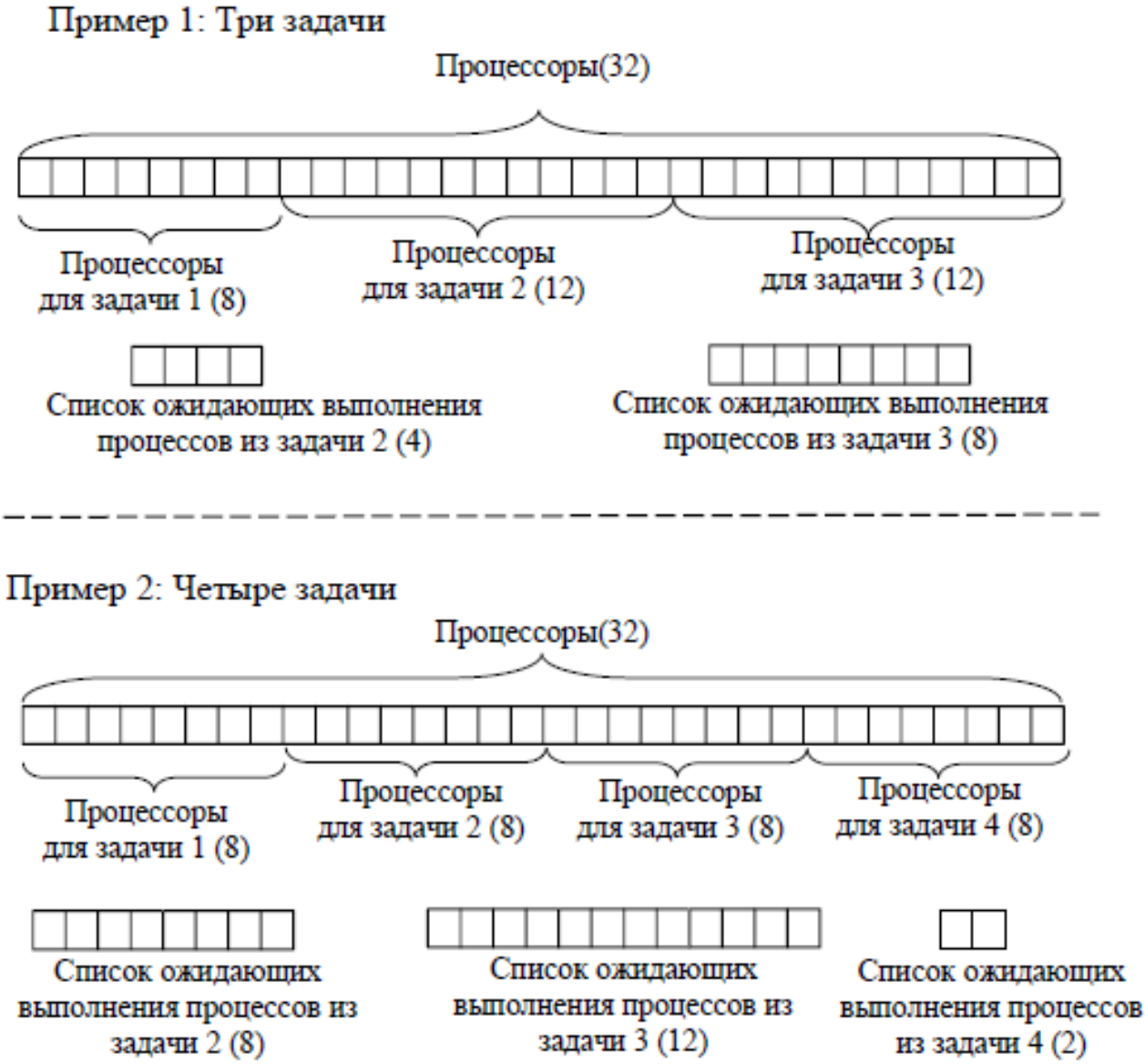

Рис. 4.7. Динамическое разделение процессоров

ЗАКЛЮЧЕНИЕ

Управление ресурсами составляет важную часть функций любой операционной системы, в особенности мультипрограммной. Операционная система должна управлять всеми ресурсами вычислительной машины таким образом, чтобы обеспечить максимальную эффективность ее функционирования.

Многопроцессорные системы в настоящее являются основными решателями большинства современных задач с использованием вычислительной техники.

Знание процессов, происходящих внутри современных компьютеров, в первую очередь изучение алгоритмов работы разнообразных модулей операционных систем, а так же знание механизмов распределения ресурсов, несомненно, поможет профессионалам-программистам создавать программные продукты, эффективные, надежные, отвечающие современным требованиям и способные решать задачи любой сложности.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Назаров, С. В. Современные операционные системы [Текст] : учеб. пособие / С. В. Назаров, А. И. Широков. – Москва : Интернет-Университет информационных технологий : Бином. Лаборатория знаний, 2011. – 279 с. – (Основы информационных технологий).

2. Олифер, В. Г. Сетевые операционные системы [Текст] : учеб. пособие / В. Г. Олифер, Н. А. Олифер. – 2-е изд. – Санкт-Петербург : Питер, 2009. – 672 с.

3. 1. Таненбаум, Э. Современные операционные системы [Текст] : пер. с англ. / Э. Танебаум, Х. Бос. – 4-е изд. – Санкт-Петербург : Питер, 2015. – 1119 с. – (Серия «Классика computer science»).

4. Управление процессами. Планирование и диспетчеризация процессов [Электронный ресурс] : лекция. – Режим доступа: http://www.intuit.ru/studies/courses/641/497/lecture/11280?page=1. – 10.01.2017.

5. Планирование выполнения процессов [Электронный ресурс]. – Режим доступа: http://citforum.ru/operating_systems/bach/glava_78.shtml. – 10.11.2017.

Учебное издание

Караваева Ольга Владимировна

ОПЕРАЦИОННЫЕ СИСТЕМЫ. УПРАВЛЕНИЕ ПАМЯТЬЮ И ПРОЦЕССАМИ

Учебное пособие

Подписано в печать 19.01.2017. Печать цифровая. Бумага для офисной техники. Усл. печ. л. 3,91. Тираж 5 экз. Заказ № 4101.

Федеральное государственное бюджетное образовательное учреждение высшего образования «Вятский государственный университет».

610000, г. Киров, ул. Московская, 36, тел.: (8332) 74-25-63, http://vyatsu.ru