МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего профессионального образования

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ

ОСНОВЫ ОРГАНИЗАЦИИ ОПЕРАЦИОННЫХ СИСТЕМ

Учебное пособие

Санкт-Петербург
2012

Востриков А.А., Кучин Н.В.

Основы организации операционных систем: Учеб. пособие/ СПбГУАП. СПб., 2011.

Утверждено
редакционно-издательским советом университета
в качестве учебного пособия

1.Основные понятия и архитектура операционных систем

1.1 Обзор развития операционных систем

Операционные системы (ОС), как основной и необходимый компонент программного обеспечения современных вычислительных систем, появились значительно позднее первых компьютеров. Их необходимость была определена в процессе развития и эволюции вычислительной техники. Кратко рассмотрим этот процесс начиная с первых вычислительных систем.

Системы *ориентированные на использование перфокарт*.

Пользователь ЭВМ, используя специальное устройство, готовил свою задачу в виде колоды перфокарт, на которой размещал код программы и исходные данные к ней. Затем, в отведённое для него время, используя устройство считывания с перфокарт и пульт управления, вводил код программы в оперативную память (ОП) ЭВМ и запускал задачу на выполнение. Далее, манипулируя клавишами пульта управления, останавливал и перезапускал свою программу по мере надобности. После получения результатов пользователь покидал машинный зал. Такая схема работы на ЭВМ имела следующие свойства:

- в каждый момент рабочего времени ЭВМ использовалась для выполнения только одной пользовательской (прикладной) программы, т. е. реализовывался однопрограммный режим работы;
- управление вычислениями осуществлялось вручную, т. е. отсутствовала автоматизация управления;
- пропускная способность вычислительной системы была крайне низкой.

Пропускная способность вычислительной системы определяется количеством выполненых задач в единицу времени. Низкая пропускная способность таких систем связана с тем, что значительная часть общего времени решения задачи тратилась на выполнение операций ввода/вывода, а процессор, как основной ресурс вычислительной системы, в это время простаивал и не выполнял никакой полезной работы. Поэтому, для повышения эффективности использования вычислительных систем необходимо было сократить время, которое тратилось на ввод/вывод.

Системы ориентированные на магнитные ленты(МЛ).

Пользователь, перед выходом на ЭВМ, переносил свои данные и код программы на МЛ. Вычислительная система снабжалась накопителями на магнитной ленте (НМЛ), которые представляли устройства ввода/вывода с более высоким быстродействием. Схема работы в таких системах мало отличалась от систем с перфокартами. Разница заключалась лишь в том, что информация считывалась в оперативную память ЭВМ и записывалась из неё с большей скоростью. Как и раньше, реализовывался однопрограммный режим работы, а управление вычислениями осуществлялось вручную. Пропускная способность системы оставалась низкой, хотя и несколько выше, чем в системах с перфокартами. Следующим этапом развития вычислительных систем стали системы пакетной обработки данных.

Системы пакетной обработки данных.

Оператор создавал пакет заданий, последовательно записывая их на МЛ. Задания отделялись друг от друга специальными маркерами. После составления пакета и установки его на НМЛ оператор, используя пульт управления, запускал пакет на выполнение. Для того чтобы пакет заданий мог последовательно

выполняться на ЭВМ, в её памяти должна храниться специальная программа – монитор пакетной обработки. Такой монитор отслеживал момент окончания выполнения очередного задания и загружал в ОП следующее задание. Кроме того, монитор фиксировал время выполнения каждого задания и реагировал на ряд нештатных ситуаций (попытка деления на ноль, зацикливание и т. п.). По своему назначению монитор пакетной обработки являлся управляющей программой, а по способу реализации – системной. Можно сказать, что такой монитор являлся простейшей операционной системой.

Характерной особенностью систем пакетной обработки является автоматизация управления вычислениями (функционирование монитора). Но эффективность работы оставалось весьма низкой из-за однопрограммного режима выполнения задач и значительных простоев процессора. Очевидно, для того чтобы повысить эффективность функционирования вычислительной системы, необходимо минимизировать простои процессора. Поэтому, следующим этапом стал этап создания мультипрограммных вычислительных систем. Именно на этом этапе появились полноценные операционные системы.

Мультипрограммные вычислительные системы.

Для минимизации простоев процессора необходимо выполнить три условия:

- необходим механизм разделения процессора (процессорного времени) между выполняемыми программами;
- в ОП необходимо одновременно хранить коды нескольких программ;
- необходимо обеспечить одновременную работу устройств ввода/вывода и центрального процессора.

Реализация этих условий значительно усложнила вычислительные системы. Если говорить об аппаратной части(hardware), то в составе компьютеров появились специальные устройства – каналы(контроллеры, микроконтроллеры). Эти устройства предназначены для управления обменом между ОП и соответствующими устройствами ввода/вывода. Они могут функционировать определённое время независимо и асинхронно относительно центрального процессора, что способствует выполнению третьего условия. Но для обеспечения всех трёх условий, определяющих мультипрограммный режим работы вычислительной системы, изменений в аппаратной части было недостаточно. Необходимо было создать комплекс системных программных средств, которые бы распределяли процессор между пользовательскими программами (диспетчер задач), выделяли бы области памяти для хранения кодов нескольких программ одновременно (планировщик памяти), осуществляли планирование операций ввода/вывода (супервизор ввода/вывода), координировали бы параллельную работу различных устройств из состава системы (механизм прерываний) и т. д. Кроме того, на данном этапе эволюции вычислительных систем очень важным требованием стало требование минимизации возможности непосредственного доступа к ресурсам вычислительной системы со стороны пользовательских программ и пользователей. Реализация такого требования значительно повышает надёжность функционирования вычислительной системы. Другими словами системные программы должны обеспечивать интерфейс между пользовательскими программами с одной стороны и аппаратурой с другой. Комплекс системных программных средств, позволяющий обеспечивать вышеуказанные требования, составляет основу современных ОС. С появлением локальных и глобальных вычислительных сетей появились сетевые ОС.

Вычислительные системы на основе сетей.

Сетевые ОС отличаются от ОС для локальных компьютеров тем, что в их составе существуют механизмы обеспечивающие доступ к удалённым(сетевым) ресурсам. Сетевую ОС можно представить как совокупность служб, каждая из которых позволяет выполнять определённый набор функций с конкретным типом ресурса (файловая служба, почтовая служба и т. д.). Каждая служба разделяется на клиентскую и серверную части. Клиентская часть по требованию создаёт запрос к удаленному ресурсу, а серверная часть выделяет удаленный ресурс с учётом прав доступа. В составе сетевой ОС также имеется транспортная подсистема, позволяющая передавать информацию между различными узлами сети. Функционирование транспортной подсистемы опирается на совокупность протоколов более низкого уровня. Все современные версии ОС являются сетевыми.

1. 2 Назначение и функции ОС

Операционная система это комплекс взаимосвязанных системных программных модулей и соответствующих таблиц, предназначенных для эффективного (в некотором смысле) распределения ресурсов вычислительной системы во время выполнения пользовательских программ, а также для организации интерфейса между пользователями и прикладными программами с одной стороны и аппаратурой с другой.

Интерфейс между пользователями и аппаратурой поддерживается на основе использования и реализации в составе ОС командного языка запросов. В настоящее время такой язык имеет форму графических оболочек.

Интерфейс между пользовательскими программами(приложениями) и аппаратурой обеспечивается с помощью функций из состава интерфейса прикладного программирования – API(Application Programming Interface).

Перечислим основные функции ОС как распределителя ресурсов:

- распределение процессора (процессорного времени) между активными процессами (задачами) и потоками вычислительной системы – диспетчеризация задач;
- организация оперативной памяти для осуществления мультипрограммного режима (организация виртуальной памяти, защита различных областей памяти от несанкционированного доступа);
- организация ввода/вывода (планирование ввода/вывода, управление микроконтороллерами и внешними устройствами, организация файлов и каталогов на магнитных дисках);
- организация механизма прерываний как основного средства координации функционирования различных компонент вычислительной системы;
- синхронизация доступа со стороны процессов (задач) к общим (разделяемым) ресурсам системы;
- защита данных и администрирование (организация процедуры логического доступа в систему, определение прав доступа при обращении к ресурсам системы, аудит системы, резервирование ресурсов);
- организация доступа к удалённым ресурсам в вычислительных сетях.

1.3 Понятие процесса, граф состояний процесса.

Для понимания принципов функционирования современных ОС необходимо определить понятия *процесса (задачи), потока, ресурса* и *прерываний.*

Понятие *процесс (задача)* является одним из основных при рассмотрении ОС. *Процесс (задача)* – это некоторый комплекс действий, связанный с выполнением отдельной программы в вычислительной системе. *Процесс* одновременно является носителем данных и выполняет операции, связанные с их обработкой. Процесс может находиться в одном из четырёх состояний – бездействия, готовности, выполнения (счёта), ожидания (блокировки).

В состоянии *бездействия* процесс не потребляет (не использует) никаких ресурсов вычислительной системы, кроме ресурса, необходимого для хранения описания процесса в некоторой форме.

В состоянии *готовности* процессу выделяются все необходимые ресурсы или могут быть выделены в любой момент по требованию, кроме процессора.

В состоянии *выполнения* процессу выделяется процессор (процессорное время), т. е. выполняются команды соответствующей программы.

В состоянии *ожидания* процесс ждёт необходимый ему ресурс, ранее выделенный некоторому другому процессу, или наступления некоторого события.

Во время своего развития процессы могут многократно переходить из одного состояния в другое. Такие переходы осуществляются в соответствии с графом переходов:

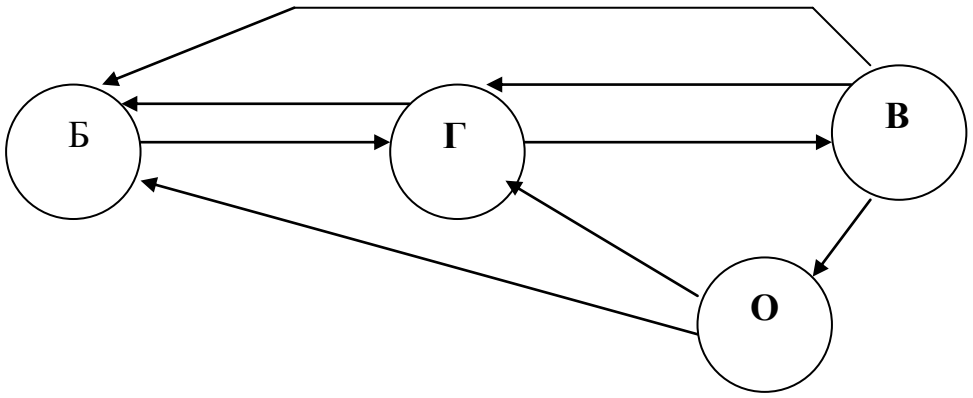


Рис 1.

Переход из одного состояния в другое может инициироваться пользователем, приложением или операционной системой, но контроль за такими переходами всегда осуществляет ОС. Это связано с тем, что любой переход предполагает выделение процессу конкретных ресурсов или освобождение ресурсов, а это является прерогативой ОС.

Состояния готовности, выполнения и ожидания являются *активными* состояниями, так как в этих состояниях процессы потребляют ресурсы

вычислительной системы и конкурируют за них. Состояние бездействия, с этой точки зрения, считается *пассивным* состоянием.

Часто, в рамках конкретного процесса, можно выделить несколько подпроцессов (потоков), причём такие потоки могут выполняться параллельно относительно друг друга. Принципиальное различие между процессами и потоками (threads) состоит в том, что разные процессы имеют высокую степень обособленности друг от друга в отличии от потоков конкретного процесса. Высокая степень обособленности разных процессов выражается в том, что каждому процессу выделяется своё адресное пространство, файлы, окна, семафоры и другие ресурсы. Такая обособленность нужна, чтобы защитить процессы друг от друга и снизить конкуренцию за общие ресурсы. Потоки конкретного процесса не имеют собственных ресурсов, при своём развитии они используют ресурсы, выделенные их процессу, кроме ресурса процессорного времени. Граф переходов для потоков имеет такой же вид, как и граф переходов для процессов.

1. 4. Классификация процессов

Для более полного понимания сущности процессов рассмотрим их классификацию по ряду классификационных признаков.

Временные ограничения.

В соответствии с этим признаком процессы разделяются на процессы реального времени, интерактивные процессы и пакетные процессы (фоновые).

Процессы реального времени – это процессы, требующие своего полного выполнения к некоторому моменту времени. Другими словами, такие процессы должны выполняться в вычислительной системе как можно скорее, так как, по своей сути они являются управляющими процессами, вырабатывающими управляющее воздействие на некоторый объект.

Интерактивные процессы – это процессы, время выполнения которых должно укладываться в определённый интервал времени. Интерактивные процессы характерны для многопользовательских вычислительных систем, где сущность процесса состоит в выполнении некоторой функции по запросу пользователя.

К выполнению *пакетных (фоновых) процессов* временные требования не предъявляются.

Генеалогический признак.

В вычислительной системе процессы могут требовать порождения (активизации) других процессов. Процесс, задающий такое требование, называется *порождающим (родительским),* а создаваемый по требованию процесс – *порождённым (дочерним).*

Динамический признак.

Два процесса относительно друг друга будут считаться *параллельными*, если интервалы их существования пересекаются во времени. В противном случае, процессы относительно друг друга будут считаться *последовательными*.

Интервалом существования процесса будем называть интервал времени между моментом активизации процесса (переход из состояния бездействия в одно из активных состояний), и моментом его окончания.

Принадлежность к центральному процессору.

Внутренние процессы связаны с выполнение команд соответствующей программы на процессоре. *Внешние* процессы – это процессы, которые

развиваются на других устройствах вычислительной системы (устройства ввода/вывода, микроконтроллеры). Внешние процессы могут развиваться асинхронно и независимо относительно работы центрального процессора.

Принадлежность к ОС.

Системные процессы – это процессы, связанные с выполнением модулей из состава ОС. *Пользовательские (прикладные)* процессы – это процессы, связанные с выполнением пользовательских программ и приложений.

Признак связности.

Процессы называются *взаимосвязанными,* если между ними существуют функциональные, управляющие, информационные или временные связи. В противном случае процессы будут считаться *изолированными.*

Взаимосвязанные параллельные процессы называют *взаимодействующими.* Такие процессы выполняют некоторую общую работу, используя, при этом, общие (разделяемые) ресурсы. Параллельные процессы, которые используют общие ресурсы, не выполняя общей работы в системе, называются *конкурирующими.*

1. 5. Ресурсы вычислительных систем, их классификация.

С точки зрения вычислительных систем, ресурсом является средство вычислительной системы, которое может быть выделено процессу (задаче) на определённый период времени для его успешного развития и выполнения.

Для более отчётливого понимания сущности ресурсов необходимо их определять в соответствии с различными классификационными признаками. Рассмотрим некоторые из них.

Реальность существования.

В соответствии с этим классификационным признаком все ресурсы вычислительной системы можно разделить на *физические* и *виртуальные.*

Физический ресурс – это ресурс, который реально существует и имеет конкретные характеристики. Виртуальный (мнимый) ресурс представляет собой некоторую модель, в основе которой лежит соответствующий физический ресурс. Как модель, виртуальный ресурс, реализуется в некоторой программно-аппаратной форме, и в этом смысле он существует. Виртуальный ресурс может иметь свойства и характеристики, значительно отличающиеся от свойств и характеристик соответствующего физического ресурса. Примером такого отличия является разница в характеристиках и в возможностях виртуальной и физической памяти компьютера.

Форма реализации.

В соответствии с этим признаком ресурсы разделяются на *аппаратные* (твёрдые) и *мягкие* (остальные). К мягким ресурсам относятся, прежде всего, *программные* ресурсы. К аппаратным ресурсам относятся процессор, оперативная память, магнитные диски и другие устройства ввода/вывода, а к программным – библиотечные модули, драйверы, компиляторы и т. д.

Время существования

Ресурс, существующий в системе до момента порождения процесса и доступный для использования на всём интервале существования процесса, называется *постоянным. Временной* ресурс может появляться и исчезать в системе динамически в течение интервала существования процесса. К временным ресурсам относятся различного рода сигналы, включая сигналы

прерываний, а также различные сообщения, которыми процессы обмениваются между собой.

Структура.

Ресурс называется *простым,* если не содержит составных элементов и рассматривается при распределении процессам как единое целое. *Составной* ресурс характеризуется некоторой структурой, т. е. имеет в своём составе некоторые однотипные элементы, обладающие одинаковыми характеристиками. Простые ресурсы могут находиться только в двух состояниях – “занят”, “свободен”, а для составных ресурсов характерны три состояния – “занят”, “свободен”, “частично занят”. Примером простого ресурса, с точки зрения операционных систем, является процессор, а составного – оперативная память.

Возможность восстанавливаемости.

Ресурс называется *воспроизводимым (системным),* если после его использования некоторым процессом, он может быть использован другим процессом. *Потребляемый* ресурс – это ресурс, который после своего использования некоторым процессом исчезает из системы. К потребляемым ресурсам, прежде всего, относятся сигналы и сообщения.

1. 6. Прерывания и порядок их обработки.

Механизм прерываний является основным координирующим механизмом в вычислительной системе, так как с его помощью решается задача параллельного функционирования различных устройств системы и корректного реагирования на особые события, которые в ней могут происходить. Прерывание – это принудительная передача управления от выполняемой программы к системе (а через неё к специальной программе обработки прерывания – обработчику прерывания), происходящая при возникновении определённого события. Основная идея прерываний – реализация асинхронного режима работы и распараллеливание работы отдельных устройств вычислительной системы.

Все виды прерываний, можно разделить на внутренние и внешние прерывания. Внутренние прерывания определяются событиями, связанными с работой центрального процессора (попытка деления на ноль, переполнение разрядной сетки, попытка сформировать неправильный адрес в оперативной памяти и т. п.). Внешние прерывания связаны с работой таймера и других устройств ввода/вывода. Внешние прерывания носят асинхронный характер по отношению к работе центрального процессора.

С другой стороны, все виды прерываний можно разделять на аппаратные и программные. Программные прерывания, по своей форме, представляют собой программное обращение к необходимому обработчику прерываний. Всё множество обработчиков прерываний являются резидентными модулями из состава операционной системы.

Порядок обработки прерываний идентичен для различных архитектур вычислительных систем и состоит в следующем:

1. Прежде всего, выполняется до конца текущая команда прерываемой программы.
2. Устанавливается факт прерывания и определяется его тип.
3. Запоминается состояние прерываемого процесса (сохраняется контекст задачи), т. е. сохраняется, на момент прерывания, в определённом

месте памяти содержимое регистров процессора (прежде всего счётчика команд).

4. Передаётся управление обработчику прерываний. Для этого, в счётчик команд заносится начальный адрес обработчика прерываний, а в соответствующие регистры – информация о данном обработчике.
5. При необходимости, сохраняется дополнительная информация о прерываемой программе.
6. Обработка прерывания, т. е. выполнение команд обработчика прерывания.
7. Восстановление информации о раннее прерванном процессе (этап, обратный этапу 5), и возврат в прерванную программу.

Этапы 1-4 реализуются аппаратно, а шаги 5- 7 – программно.

Сигналы, вызывающие прерывания, могут возникать в различных устройствах вычислительной системы, более того, они могут возникать одновременно. Поэтому, возникает задача определения очерёдности обработки прерываний. Данная задача решается на основе разных приоритетов для различных типов прерываний. Приоритеты различных типов прерываний, в порядке их уменьшения, определяются следующим образом:

- прерывания от средств контроля процессора (наивысший приоритет);
- прерывания системного таймера;
- прерывания от магнитных дисков;
- прерывания сетевого оборудования;
- прерывания от терминалов;
- программные прерывания.

Учёт приоритетов может быть встроен в технические средства, а также определяться ОС, т. е. очерёдность обработки прерываний осуществляется аппаратно-программным путём. Кроме того, в составе современных вычислительных систем есть средства, позволяющие проигнорировать сигналы прерываний или отложить их обработку (маскирование прерываний). Наличие таких средств, позволяет управлять очерёдностью обработки прерываний в соответствии с различными правилами (*дисциплины обслуживания прерываний*). Рассмотрим наиболее характерные из них:

- с *относительными приоритетами*, т. е. обслуживание не прерывается даже при наличии запросов с более высокими приоритетами. После окончания обслуживания данного запроса, обслуживается запрос с наивысшим приоритетом;
- с *абсолютным приоритетом*, т. е. всегда обслуживается запрос с наивысшим приоритетом;
- по *принципу стека*, или, как говорят по дисциплине LCFS (last come first served – последним пришёл – первым обслужен), т. е. запросы с более низким приоритетом могут прерывать обработку запросов с более высоким приоритетом.

Последние две дисциплины предполагают возможность прерывания работы обработчиков прерываний, т. е., в этих случаях, обработка прерываний носит многоуровневый характер.

В состав ОС входит специальная компонента – *супервизор прерываний (диспетчер прерываний)*, которая реализует некоторую дисциплину обслуживания прерываний. Такой супервизор использует специальные программно-аппаратные механизмы включения, отключения и маскирования прерываний.

1. 7. Архитектура ОС.

Современные ОС имеют модульную структуру, которая позволяет увеличивать их возможности развития, расширения и переноса на новые платформы. Единой архитектуры ОС не существует, но существуют универсальные подходы к структуированию ОС. Модули ОС разделяются на две группы: ядро ОС и вспомогательные модули.

Модули ядра ОС выполняют основные функции по распределению ресурсов в вычислительной системе. Они используются часто, и должны работать как можно быстрее. Поэтому, такие модули оформляются как *резидентные* (резидентные модули – это модули, коды которых постоянно находятся в оперативной памяти). Следовательно, при загрузке системы, ядро ОС всегда помещается в специально выделенную область оперативной памяти.

Вспомогательные модули ОС выполняют функции, не требующие частого использования (сжатие, архивирование, копирование и т. п.). Они оформляются как дискрезидентные модули (дискрезидентные модули – это модули, которые постоянно хранятся во внешней памяти и загружаются в оперативную память только на время своего выполнения).

Модули, входящие в состав ОС, можно классифицировать по кратности использования. *Однократно* используемые модули – это модули, которые могут быть правильно выполнены только один раз, так как, при своём выполнении, они “портят” свою собственную память. Примером такого модуля является абсолютный загрузчик системы. Модули *многократного приме*нения (повторно используемые модули) могут быть, в свою очередь, *непривилегированными, привилегированными* и *реентерабельными.*

Непривилегированные программные модули – это модули, которые могут быть прерваны во время своей работы, причём, промежуточные результаты их работы не сохраняются.

Привилегированные программные модули не могут быть прерваны во время своего выполнения, т. е. они работают при отключённой системе прерываний. Такие модули, начав работать, выполняются до конца.

Реентерабельные программные модули допускают повторное многократное прерывание своего выполнения и повторный их запуск, причём при каждом таком прерывании происходит запоминание промежуточных результатов в некоторой специально отведённой для этого области памяти. Такие модули обычно состоят из трёх секций. Первая секция предназначена для выделения памяти под промежуточные результаты выполнения. Вторая секция представляет собой непосредственный код программного модуля. Третья секция – это секция освобождения памяти, которая использовалась для хранения промежуточных результатов. Первая и третья секции работают как привилегированные секции, а вторая – как непривилегированная. Примерами реентерабельных модулей является ряд драйверов из состава ОС.

Также в составе ОС существуют *повторно входимые* модули. Такие модули допускают своё многократное параллельное использование, но их нельзя прерывать. Повторно входимые модули состоят из привилегированных секций, каждая из которых имеет свою собственную точку входа (начальный адрес). После выполнения очередной такой секции, управление может быть

передано системой другой секции того же модуля или повторно той же самой секции.

Привилегированный режим.

Ни одно приложение, при своём функционировании, не должно иметь возможности без контроля со стороны ОС получать ресурсы вычислительной системы. Поэтому, ОС должна иметь полномочия (привилегии) по распределению ресурсов. Обеспечение таких привилегий для ОС осуществляется за счёт средств аппаратной поддержки, которые поддерживают два режима работы вычислительной системы – *пользовательский* и *привилегированный (режим ядра)*. Так как ядро ОС выполняет основные её функции, именно оно должно работать в привилегированном режиме. В пользовательском режиме работают пользовательские программы и некоторые дискрезидентные утилиты из состава ОС. В пользовательском режиме запрещается выполнение некоторых инструкций (команд), связанных с распределением ресурсов вычислительной системы (переключение процессора, управление вводом/выводом, механизмы распределения и защиты памяти и т. д.). Переход из пользовательского режима в привилегированный инициируется соответствующим системным вызовом из состава API, а осуществляется аппаратными средствами. Наличие привилегированного режима функционирования вычислительной системы повышает её устойчивость и надёжность, так как распределение ресурсов происходит под жёстким контролем ОС. С другой стороны, наличие привилегированного режима несколько снижает производительность системы, что видно из рис. 2.

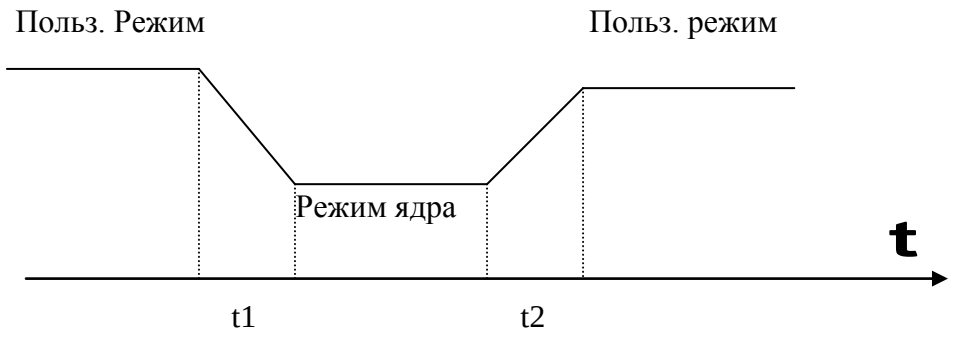


Рис. 2

Потеря производительности связана с тем, что на переход из пользовательского режима в привилегированный и обратно тратится определённое время (интервалы t1 и t2). Чем больше в пользовательской программе системных вызовов, тем больше таких переходов.

1. 8. Структура ядра ОС

Ядро ОС представляет собой сложный многофункциональный комплекс, состоящий из большого числа различных, взаимосвязанных модулей. Поэтому, структура ядра ОС носит многослойный характер, где каждый слой реализует

функции определённого типа. С некоторой долей условности можно выделить пять основных слоёв.

Средства аппаратной поддержки ОС. Часть функций ОС может выполняться аппаратными средствами. Поэтому такие средства можно считать частью ОС. Они связаны с организацией вычислительных процессов (средства поддержки привилегированного режима, система прерываний, средства переключений контекстов процессов, средства защиты памяти и т. п.).

Машинно-зависимые компоненты ОС. Этот слой образуют программные модули, в которых отражается специфика аппаратной платформы компьютера. В идеале этот слой должен полностью экранировать вышележащие слои ядра от особенностей аппаратуры. Это позволяет разрабатывать вышележащие слои ядра на основе машинно-независимых модулей, существующих в единственном экземпляре для всех типов аппаратных платформ.

Базовые механизмы ядра. Этот слой выполняет наиболее примитивные операции ядра (программное переключение контекстов процессов, диспетчеризация прерываний, перемещение страниц из памяти на диск и обратно и т. п.). Эти модули не принимают решений о распределении ресурсов – они только отрабатывают принятые “наверху решения”.

Менеджеры ресурсов. Этот слой состоит из мощных функциональных модулей, реализующих стратегические задачи по управлению основными ресурсами вычислительной системы. Обычно на данном слое работают менеджеры (диспетчеры) процессов, ввода-вывода, файловой системы и оперативной памяти. Каждый из менеджеров ведёт учёт свободных и используемых ресурсов определённого типа, и планирует их распределение в соответствии с запросами приложений. Внутри слоя менеджеров существуют тесные взаимосвязи, отражающие тот факт, что для выполнения процессу требуется доступ к нескольким ресурсам – процессору, оперативной памяти, возможно, к внешней памяти или устройству ввода-вывода.

Интерфейс системных вызовов. Этот слой является самым верхним слоем ядра и взаимодействует непосредственно с приложениями и системными утилитами, образуя прикладной программный интерфейс операционной системы – API. Системные вызовы, в свою очередь, обращаются к менеджерам ресурсов, причём, для выполнения одного системного вызова, возможно, необходимо несколько таких обращений.

Такое разбиение ядра ОС на слои является достаточно условным. В реальной системе количество слоёв и распределение функций между ними может быть иным. Выбор числа слоёв является ответственным и трудным делом: увеличение числа слоёв ведет к некоторому замедлению работы ядра за счёт дополнительных накладных расходов на межслойное взаимодействие, а уменьшение числа слоёв ухудшает расширяемость и логичность системы.

1. 9. Микроядерная архитектура ОС

Микроядерная архитектура является альтернативой классическому способу построения операционной системы. Суть микроядерной архитектуры состоит в следующем. В привилегированном режиме остаётся работать только небольшая часть ОС, называемая *микроядром.* Микроядро защищено от остальных частей ОС и приложений. В состав микроядра обычно входят машинно-зависимые модули, а также модули, выполняющие базовые функции

ядра по управлению процессами, обработке прерываний, управлению виртуальной памятью, пересылке сообщений и управлению устройствами ввода-вывода, связанные с загрузкой или чтением регистров устройств. Набор функций микроядра обычно соответствует функциям слоя базовых механизмов обычного ядра. Такие функции невозможно, как правило, выполнить в пространстве приложений.

Все остальные более высокоуровневые функции ядра оформляются в виде приложений, работающих в пользовательском режиме. По своему назначению такие приложения предназначены для выполнения запросов других приложений (создание процесса, выделение памяти, проверка прав доступа к ресурсу и т. п.). Именно поэтому менеджеры ресурсов, вынесенные в пользовательский режим, называются серверами ОС, то есть модулями, основным назначением которых является обслуживание запросов локальных приложений и других модулей ОС. Очевидно, что для реализации микроядерной архитектуры необходимым условием является наличие в операционной системе удобного и эффективного способа вызова процедур одного процесса из другого. Поддержка такого механизма и является одной из задач микроядра.

Схематично механизм обращения к функциям ОС, оформленным в виде серверов, выглядит следующим образом. Клиент (приложение или другой компонент ОС) запрашивает выполнение некоторой функции у соответствующего сервера, посылая ему сообщение. Непосредственная передача сообщений между приложениями невозможна, так как их адресные пространства изолированы друг от друга. Микроядро, выполняющееся в привилегированном режиме, имеет доступ к адресным пространствам каждого из приложений и поэтому может работать в качестве посредника. Микроядро сначала передаёт сообщение, содержащее имя и параметры вызываемой процедуры нужному серверу, затем сервер выполняет запрошенную операцию, после чего микроядро возвращает результаты клиенту с помощью другого сообщения. Таким образом, работа микроядерной ОС соответствует известной модели клиент-сервер, в которой роль транспортных средств выполняет микроядро.

Преимущества микроядерной архитектуры определяются следующими свойствами:

- высокая степень переносимости, которая обусловлена тем, что весь машинно-независимый код изолирован в микроядре, поэтому для переноса системы на новый процессор требуется меньше изменений;
- высокая степень расширяемости, так как добавление новых функций и изменение существующих значительно упрощается из-за того, что все модификации можно оформлять и реализовывать в пользовательском режиме;
- повышенная надёжность ОС, так как все сервера выполняются в своих адресных пространствах, изолированных друг от друга, и, кроме того, сервера не имеют непосредственного доступа к аппаратуре.

Недостатком микроядерной архитектуры ОС является более низкая производительность. Это связано с тем, что при выполнении запроса клиента необходимо, по крайней мере, дважды переходить в привилегированный режим (см. рис. 3). Увеличение числа переходов из пользовательского режима в привилегированный режим и обратно, влечёт за собой дополнительные временные затраты на выполнение запросов к ОС.

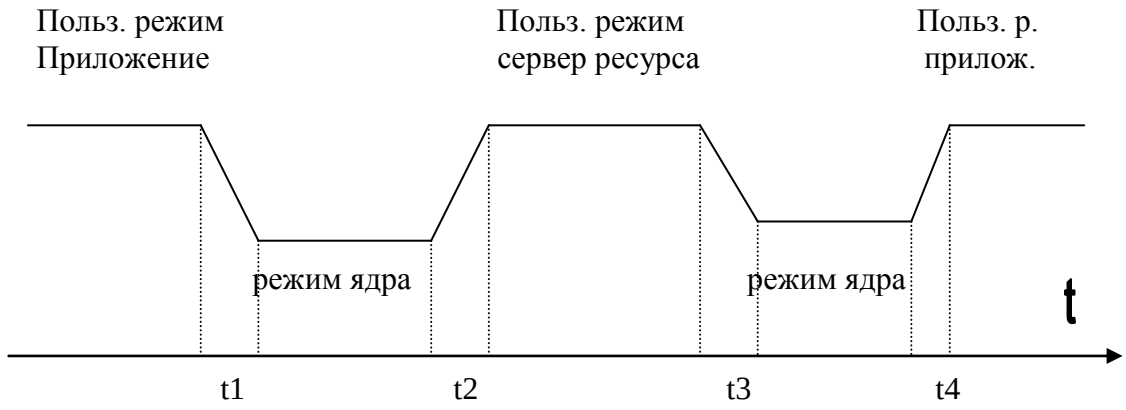


Рис. 3.

2. Планирование и диспетчеризация задач и процессов.

2.1. Планирование и диспетчеризация процессов. Дескрипторы задач.

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.

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.

2.2 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)*. Рассмотрим наиболее важные дисциплины диспетчеризации.

1. FCFS (first come – first served – первым пришёл, - первым обслужен) – прежде процессор получает та задача, которая раньше перешла в состояние готовности. Данная дисциплина проста в реализации, равноправна по отношению как к “длинным ” так и к “коротким” процессам, среднее время пребывания в очереди готовности весьма значительное.
2. SJN (shortest job next – следующий с кратчайшим заданием) – прежде процессор получает та задача, которая имеет минимальное заказное время обслуживания. Данная дисциплина требует, чтобы для каждой задачи была известна оценка потребности в машинном времени, значение которой задаётся как параметр задачи. Такая дисциплина более сложна в реализации по сравнению с FCFS, она дискриминационна по отношению к “длинным процессам”, среднее время пребывания в очереди готовности меньше чем для FCFS. SJN имеет существенный недостаток. Задачи, которые были временно заблокированы (например, ожидали завершения ввода/вывода), в результате попадут в конец очереди готовности, даже если для их выполнения требуется небольшое процессорное время.
3. SRT (shortest remaining time) – раньше процессор получает та задача, которая имеет меньше всего времени для своего завершения. Это время определяется как разность между заказанным временем обслуживания и тем

процессорным временем, которая задача уже получила. SRT свободна от недостатка, характерного для SJN. SRT сложна в реализации и дискриминационна по отношению к “длинным” процессам.

Рассмотренные дисциплины диспетчеризации являются *невытесняющими*, в отличие от *вытесняющих* дисциплин, которые будут описаны далее. Вытесняющей дисциплиной диспетчеризации будем называть такую дисциплину, которая предполагает возможное прерывание выполнения текущей задачи с целью предоставления процессора другой готовой к выполнению задаче. Рассмотрим некоторые основные вытесняющие дисциплины диспетчеризации:

4. RR (round robin) – циклическая (карусельная) дисциплина. Диспетчер выделяет готовой к выполнению задаче некоторый квант процессорного времени (интервал мультиплексирования). Если задача не успевает выполниться в течение этого кванта, диспетчер переводит её обратно в конец очереди готовности и выделяет следующий квант процессорного времени для другой готовой задачи. Данная дисциплина является дискриминационной по отношению к длинным процессам. Её удобно использовать в многопользовательских вычислительных системах, где требуется обслуживать большое число запросов, поступающих с различных рабочих станций системы.
5. Дисциплины на основе *абсолютных приоритетов* задач. Каждая задача имеет приоритет, выраженный конкретным значением, который не меняется на всём интервале существования задачи. Прежде процессор будет получать та готовая задача, которая в данный момент имеет максимальный приоритет по отношению к другим готовым задачам. Данная дисциплина характерна для систем реального времени, она дискриминационна по отношению к длинным процессам и не даёт *гарантий обслуживания* для таких процессов.
6. Дисциплины на основе *динамических приоритетов* задач. Для каждой задачи задаётся начальное значение приоритета, которое затем изменяется во времени. Таким образом, приоритет задачи есть функция времени. Конкретный вид таких функций может быть разный, но общая их направленность состоит в том, что, чем дольше задача находится в очереди готовности, тем выше становится её приоритет. Это позволяет гарантировать обслуживание как “коротких” так и “длинных” процессов.
7. Дисциплины с несколькими очередями. Диспетчер поддерживает несколько очередей готовых к выполнению задач. Каждая очередь обслуживается по своей дисциплине. Такой диспетчер сложен в реализации, так как в его составе должен быть дополнительный механизм переключения с одной очереди готовности на другую. Более простой способ реализации диспетчера (статический) предполагает, что задача, попав в некоторую очередь готовности, там и остаётся до своего полного выполнения. Более сложным способом реализации (динамическим) является способ, при котором задача может переходить из одной очереди готовности в другую на интервале своего существования.

2. 3. Дисциплины планирования

Планирование – это определение очерёдности выделения ресурсов системы задачам (процессам), которые хотят активизироваться и начать своё выполнение. Планирование связано с работой совокупности менеджеров

ресурсов ОС (планировщик памяти, супервизор ввода/вывода, файловая система и т. д.).

Наиболее естественной дисциплиной планирования является FCFS. Содержательный смысл этой дисциплины, в данном случае, состоит в следующем: необходимые ресурсы, прежде всего, выделяются той задаче, для которой раньше появилась требование активизации.

Другой возможной дисциплиной планирования является SJN. Эта дисциплина предполагает предпочтительное выделение необходимых ресурсов задаче с наименьшей оценкой времени её выполнения по сравнению с соответствующими оценками других задач.

3. Управление оперативной памятью.

Оперативная память является важнейшим ресурсом любой вычислительной системы. Она состоит из фиксированного числа ячеек, предназначенных для хранения кодов программ и данных. Команды программы могут выполняться на процессоре, если их коды и операнды предварительно загружены в оперативную память. Некоторая часть оперативной памяти всегда отводится для хранения ядра ОС, которое обычно загружается по младшим адресам памяти. Большая часть оперативной памяти используется для хранения пользовательских программ и данных. Проблема управления оперативной памятью со стороны ОС сводится к решению двух взаимосвязанных задач:

- эффективное (в некотором смысле) распределение оперативной памяти между несколькими активными задачами с целью обеспечения мультипрограммного режима;
- защита памяти от несанкционированного доступа, т. е. контроль и запрет попыток адресации к “чужим”, или, запрещённым областям памяти.

Для понимания способов организации оперативной памяти, необходимо знать методы отображения символьных имён переменных, которыми пользуются программисты, в конкретные физические адреса ячеек оперативной памяти.

3.1. Пространства и отображения, виртуальное адресное пространство.

Программист, при написании программы, определяет необходимые ему переменные с помощью символьных (логических) имён. Другими словами, он задаёт некоторое подмножество переменных, используя *логическое адресное пространство.* Это пространство ограничено правилами синтаксиса конкретного языка программирования. Логические имена переменных можно интерпретировать как символьные адреса. Задача вычислительной системы состоит в отображении этих адресов в конкретные физические адреса. Такое отображение, в общем случае, осуществляется в два этапа (см. рис. 4). Первый этап реализует система программирования (транслятор), на котором логические имена переменных преобразуются в виртуальные адреса. Конкретный вид таких адресов зависит от архитектуры вычислительной системы, конкретного транслятора и ряда других факторов. Совокупность возможных виртуальных адресов образует *виртуальное адресное пространство*. Второй этап реализуется операционной системой совместно с аппаратурой. На этом этапе

виртуальные адреса отображаются в физические адреса оперативной памяти. Конкретные значения физических адресов зависят от размера физической памяти, а в совокупности определяют *физическое адресное пространство*.

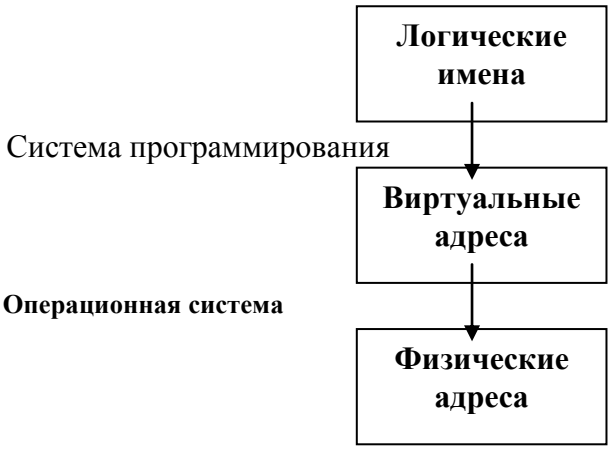


Рис. 4.

При изучении данного вопроса необходимо выделить четыре возможных варианта отображения.

При первом варианте предполагается, что виртуальное адресное пространство тождественно физическому адресному пространству. Поэтому, система программирования целиком реализует отображение логических имён в физические адреса. Этот случай характерен для отображения так называемых “абсолютных программ”. В командах таких программ в явном виде указываются их физические адреса и физические адреса операндов. Задача операционной системы сводится к обычной загрузке в память по указанным адресам.

Второй вариант предполагает эквивалентность пространства логических имён и виртуального адресного пространства. В этом случае задача отображения целиком лежит на операционной системе. Такая операционная система носит специфический характер и называется *интерпретатором.* Интерпретатор последовательно транслирует в машинный код и выполняет текущие операторы программы. При этом он создаёт свои внутренние таблицы, в которых определяется соответствие между логическими именами переменных и физическими адресами.

При третьем варианте отображение осуществляется в два этапа. При этом конкретному виртуальному адресу жестко соответствует конкретный физический адрес. Отображение осуществляется до начала выполнения задачи, т. е. код задачи не меняет своего местоположения в памяти на всё время её выполнения.

При четвёртом варианте отображение также осуществляется в два этапа, причём, конкретному виртуальному адресу могут соответствовать несколько физических адресов. Это означает, что физические адреса определяются на этапе выполнения задачи, а её код может менять своё местоположение в

оперативной памяти. Это вариант соответствует различным способам организации *виртуальной памяти*.

3.2. Распределение разделами.

Та часть оперативной памяти, которая предназначена для хранения пользовательских задач и приложений, распределяется на несколько разделов. Каждый раздел предназначен для единовременного хранения кода и данных одной задачи. Загружаемой в оперативную память логической единицей является задача. Число разделов может быть различным в разные моменты работы вычислительной системы, но экспериментально доказано, что для того чтобы процессор был загружен на 90% , необходимо иметь в оперативной памяти 4-5 задач. Число задач одновременно загруженных в оперативную память называется коэффициентом мультипрограммирования. Существует несколько стратегий (дисциплин) распределения памяти разделами. Рассмотрим основные дисциплины.

Распределение фиксированными разделами

Распределение оперативной памяти на разделы производится на этапе конфигурации вычислительной системы. Каждый раздел имеет фиксированный начальный адрес и фиксированный размер. Заданное распределение не может изменяться во время работы вычислительной системы вплоть до новой её конфигурации. Такая дисциплина распределения памяти имеет очевидный недостаток – значительные области памяти оказываются неиспользованными (*фрагментация памяти*). Причиной высокого уровня фрагментации памяти является несоответствие размеров выделенных разделов и размеров кодов загружаемых задач (см. рис. 5).

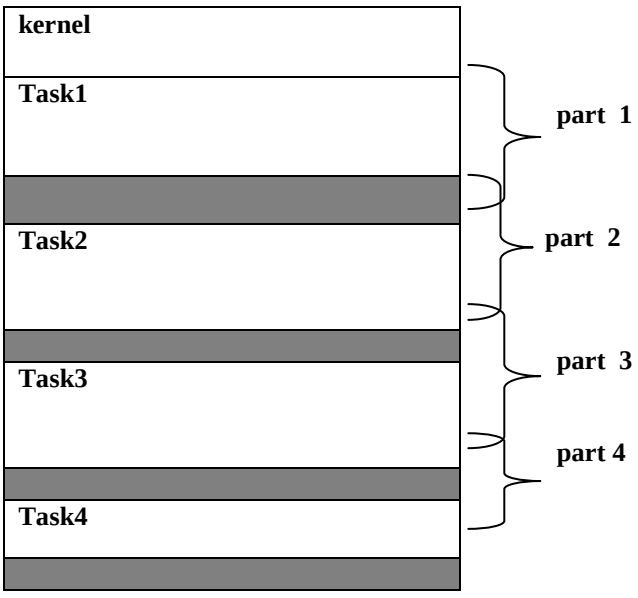


Рис. 5.

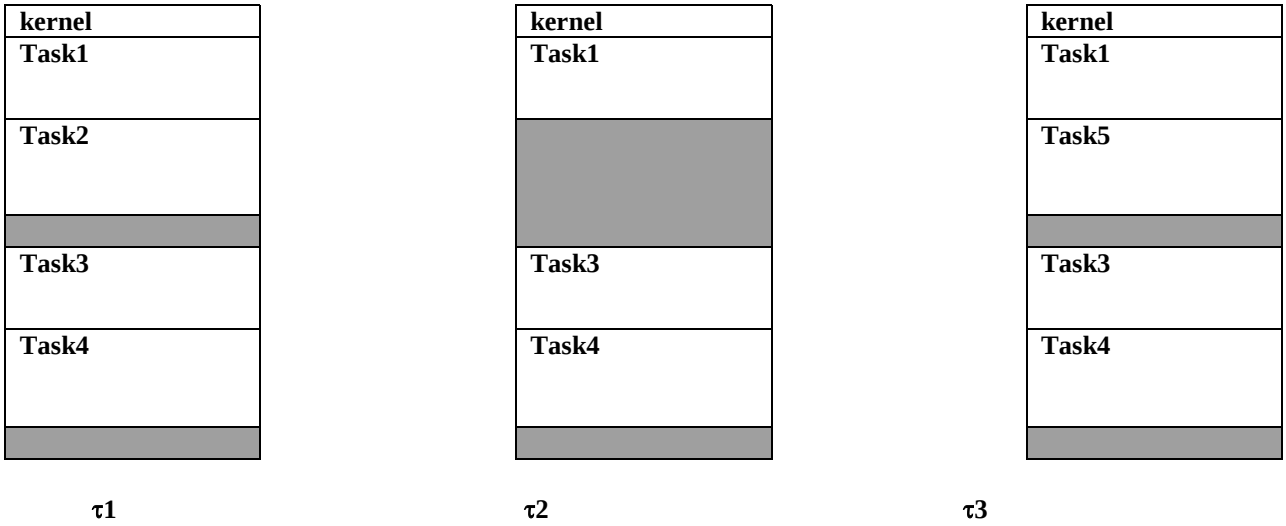
Затемнённые области на рис. показывают незаполненные полезным кодом участки памяти. Единственным достоинством такой дисциплины распределения памяти является простота управления со стороны ОС. Для снижения уровня фрагментации памяти используется распределение разделами с подвижными границами.

Распределение разделами с подвижными границами

Основная идея данной дисциплины распределения заключается в том, что в памяти выделяется раздел, размер которого равен размеру очередной загружаемой задачи. Следовательно, такое выделение возможно только во время работы вычислительной системы. Поэтому, в составе ОС необходимо иметь некоторое средство, позволяющее оперативно предоставлять память под загружаемые задачи. Такое средство будем называть планировщиком памяти.

Планировщик памяти входит в состав ядра ОС. Он создаёт и поддерживает в своей собственной памяти список свободных областей. Элементы такого списка состоят из двух полей. В первом поле хранится значение начального адреса свободной области памяти, а во втором значение размера этой области. Просматривая список, планировщик определяет свободную область памяти, подходящую по размеру для очередной загружаемой задачи, после чего производится непосредственная загрузка кода задачи в память. После каждой загрузки планировщик модифицирует (корректирует) вышеуказанный список. Просмотр списка с целью определения подходящей свободной области памяти может осуществляться двумя способами - “первый подходящий” или “наиболее подходящий”. Первый способ предполагает поиск любой свободной области памяти, размер которой не меньше размера кода загружаемой задачи. Второй способ предполагает поиск свободной области памяти, размер которой наиболее близок размеру кода загружаемой задачи и при этом не меньше, чем размер этого кода. При первом способе требуется просматривать, в среднем, половину списка, а при втором - весь список. Кроме того, планировщик реализует функцию освобождения памяти после выполнения очередной задачи, модифицируя список свободных областей памяти.

При данной дисциплине распределения памяти уровень фрагментации и коэффициент мультипрограммирования могут значительно изменяться во времени, что иллюстрируется на рис. 6.



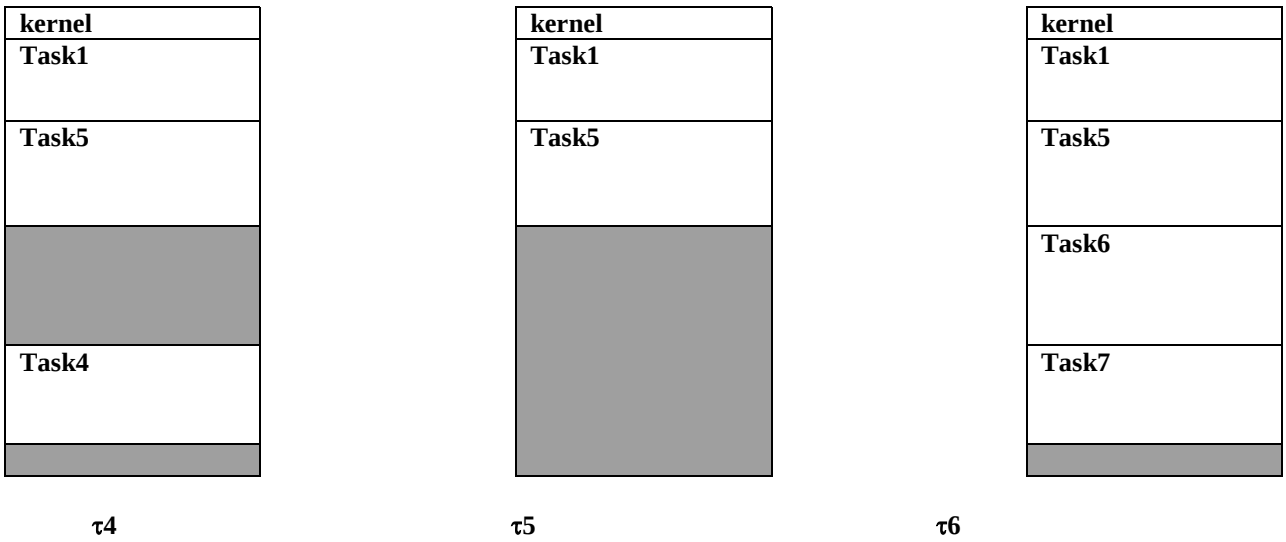


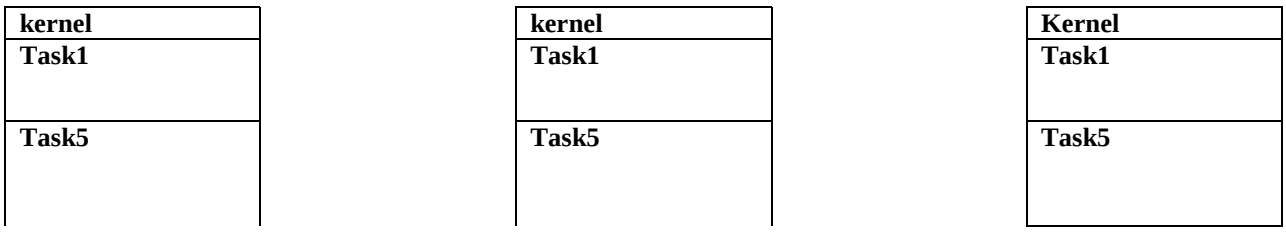
Рис. 6.

В момент **τ1** в памяти находятся коды четырёх задач. В момент **τ2** закончилось выполнение задачи **task2** и планировщик освободил память, достаточную для загрузки задачи **task5** (момент **τ3**). В момент **τ4** закончилось выполнение задачи **task3**, планировщик освободил память, но размеры свободных областей недостаточны по размеру для загрузки задачи **task6.** Поэтому, очередная загрузка произойдёт только после выполнения **task4** и соответствующего освобождения памяти (момент **τ5**). При этом, размер свободной области памяти оказался достаточным для загрузки кодов сразу двух задач – **task6** и **task7** (момент **τ6**).

Данная дисциплина распределения является средством локальной оптимизации, так как оптимизируется местоположение в памяти для конкретной задачи, но не распределение всей памяти в целом.

Распределение подвижными разделами

Для оптимизации распределения всей памяти в целом и уменьшения уровня фрагментации можно использовать метод смещения (сдвига) по адресам кодов загруженных задач с целью получения непрерывных областей памяти, размеры которых будут достаточны для загрузки новых задач. Данный метод иллюстрируется на рис. 7.



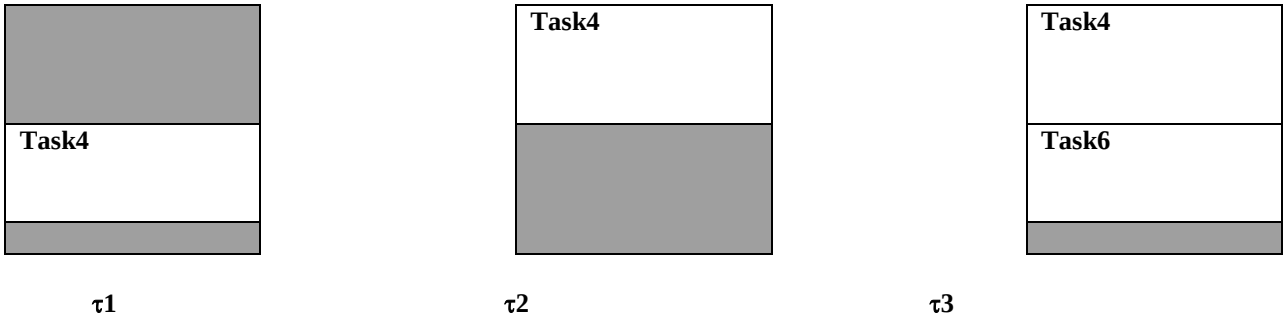


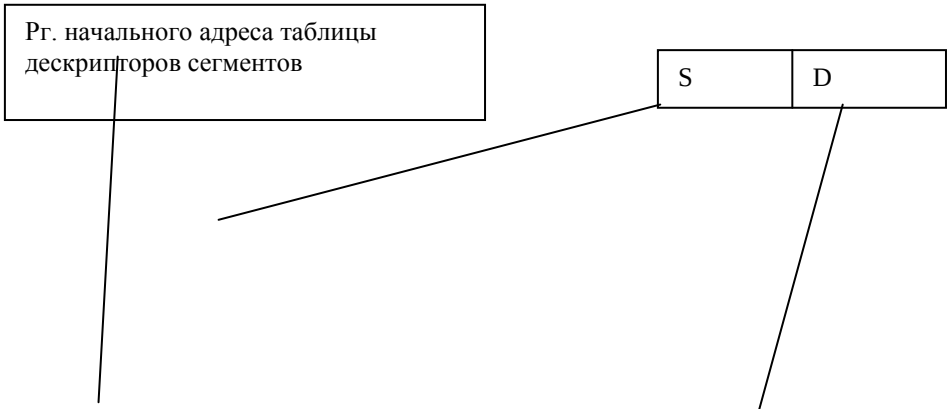
Рис. 7.

В момент **τ1** в памяти имеется две свободные области, но ни одна из них недостаточна по размеру, чтобы загрузить следующую задачу. Планировщик делает смещение кода задачи task4, образуя одну, большую по размеру, непрерывную свободную область памяти (момент **τ2).** Поэтому появляется возможность загрузить следующую задачу task6 (момент **τ3).**

Смещение в памяти кода задачи, которое выполняет планировщик, должно быть согласовано по времени с окончанием операции ввода/вывода для этой задачи (в данном случае – task4). Если такое согласование отсутствует, то результаты ввода/вывода могут записаться на "старое" место в памяти, и задача будет разрушена. С другой стороны, такое согласование противоречит принципам мультипрограммирования, в соответствии с которыми внешние устройства должны работать независимо и асинхронно относительно центрального процессора. Поэтому данная дисциплина распределения разделами не нашла практического применения.

3.3. Сегментная организация памяти.

Сегментная организация памяти – это один из способов организации виртуальной памяти. Транслятор подготавливает задачу в виде совокупности сегментов, в общем случае разного размера. Эти сегменты хранятся во внешней памяти. Адресация внутри сегментов носит непрерывный характер. Виртуальные адреса, после трансляции, имеют вид – (s,d), где s – виртуальный номер сегмента, а d – смещение относительно начала сегмента. Предполагается, что в общем случае, при первоначальной загрузке сегментов задачи в оперативную память, не все из них могут попасть в неё, и часть сегментов задачи может остаться во внешней памяти. Отображение виртуальных адресов в физические происходит на этапе выполнения задачи и имеет вид (см. рис. 8):



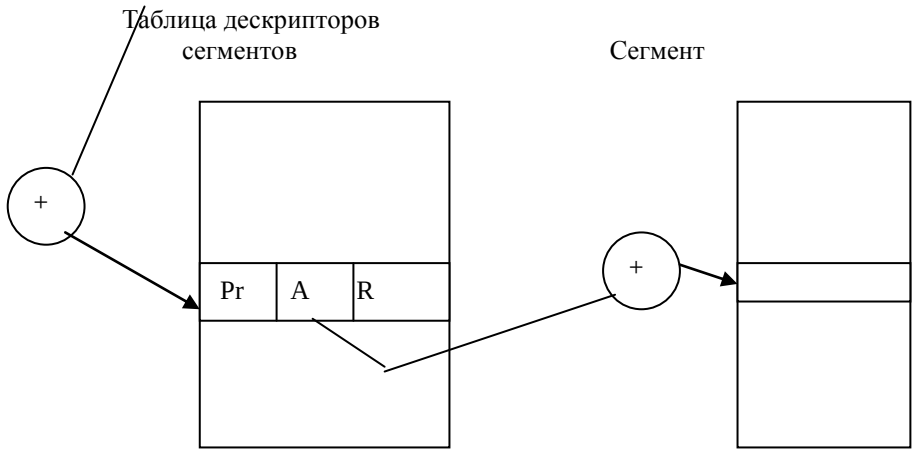


Рис. 8.

При загрузке задачи в оперативную память, ОС, совместно с аппаратурой, создаёт в памяти таблицу дескрипторов сегментов задачи. Первоначально дескрипторы сегментов задачи создаются на этапе её трансляции. Каждый дескриптор имеет поля, содержимое которых может изменяться во время выполнения задачи. На рис. показаны некоторые из них:

- Pr – битовый признак присутствия/отсутствия сегментов в оперативной памяти (0 – сегмент отсутствует в памяти, 1 – сегмент присутствует в памяти);
- A – начальный адрес сегмента;
- R – размер сегмента.

В специальном регистре процессора хранится значение начального адреса таблицы дескрипторов сегментов выполняемой задачи. При переключении на выполнение новой задачи ОС меняет содержимое этого регистра. Отображение виртуальных адресов в физические адреса производится в два этапа. На первом этапе определяется адрес конкретного дескриптора в таблице путем сложения начального адреса таблицы, хранящегося в регистре, и значения S. Если сегмент отсутствует в памяти (pr=0), то генерируется соответствующее прерывание и вызывается программа подкачки сегмента (своппинг). На втором этапе определяется значение физического адреса путём сложения значений A и I. Перед выполнением этой операции выполняется проверка условия I<=R. Если это условие выполняется (физический адрес определяется корректно и не выходит за адресуемый сегмент), то определение физического адреса внутри страницы разрешается. В противном случае генерируется соответствующее прерывание, и адресация к странице откладывается.

Рассмотрим свойства сегментной организации памяти. Во первых необходимо отметить, что загружаемым в память объектом является сегмент некоторой задачи а не её код целиком. Кроме того, при данной организации памяти обеспечивается автоматическое перекрытие сегментов в памяти (на место в памяти, где хранился некоторый сегмент, возможна загрузка нового необходимого сегмента). Уровень фрагментации памяти может быть достаточно высоким. Это связано с тем, что при перекрытии сегментов в памяти могут образовываться значительные по размеру незаполненные области памяти из-за разных размеров сегментов.

3.4. Страничная организация памяти

Страничная организация памяти – это ещё один из способов организации виртуальной памяти. Транслятор подготавливает задачу в виде совокупности виртуальных страниц одинакового размера. Эти страницы хранятся во внешней памяти. Адресация внутри страниц носит непрерывный характер. Виртуальные адреса, после трансляции, имеют вид – (p,i), где i – виртуальный номер страницы, а i – смещение относительно начала страницы. Предполагается, что в общем случае, при первоначальной загрузке страниц задачи в оперативную память, не все из них могут попасть в неё, и часть страниц задачи может остаться во внешней памяти. Физическая память рассматривается как совокупность физических страниц того же размера как и размер виртуальные страницы. Отображение виртуальных адресов в физические происходит на этапе выполнения задачи и имеет вид (см. рис. 9).

При загрузке задачи в оперативную память, ОС, совместно с аппаратурой, создаёт в памяти таблицу дескрипторов страниц задачи. Первоначально дескрипторы страниц задачи создаются на этапе её трансляции. Каждый дескриптор имеет поля, содержимое которых может изменяться во время выполнения задачи. На рис. 9 показаны некоторые из них:

- Pr – битовый признак присутствия/отсутствия страниц в оперативной памяти (0 – сегмент отсутствует в памяти, 1 – сегмент присутствует в памяти);
- F – физический номер страницы.

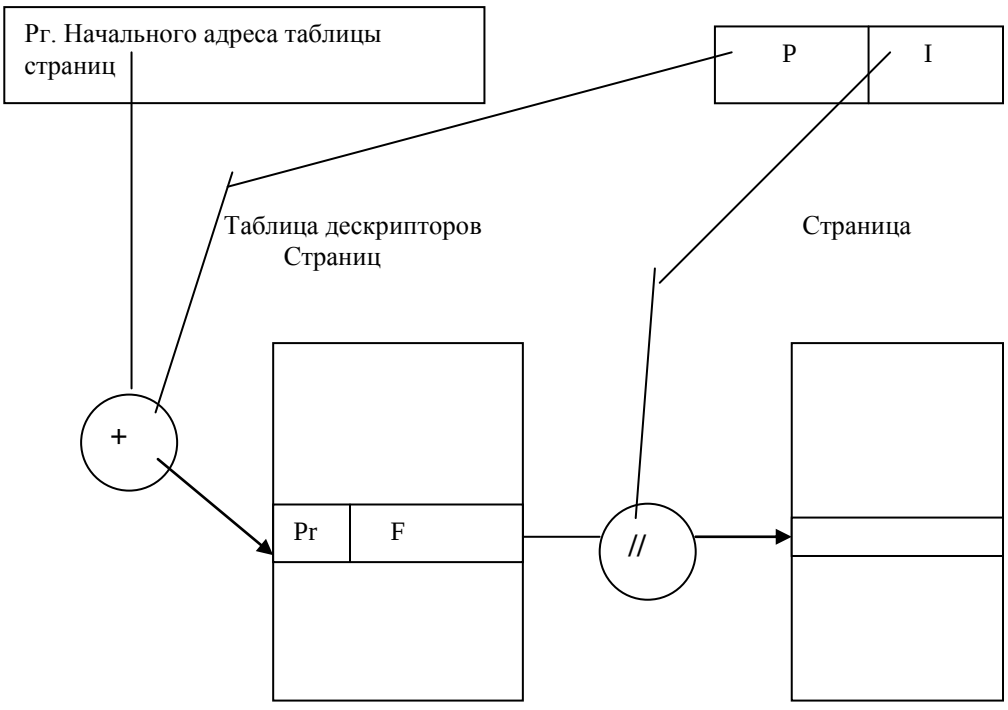


Рис. 9.

В специальном регистре процессора хранится значение начального адреса таблицы дескрипторов страниц выполняемой задачи. При переключении на выполнение новой задачи ОС меняет содержимое этого регистра. Отображение виртуальных адресов в физические адреса производится в два этапа. На первом этапе определяется адрес конкретного дескриптора в таблице путем сложения начального адреса таблицы, хранящегося в регистре, и значения P. Если страница отсутствует в памяти (pr=0), то генерируется соответствующее прерывание и вызывается программа подкачки страницы (свопинг). На втором этапе определяется значение физического адреса внутри страницы путём реализации операции *приписывания* (//). Данная операция может быть представлена в виде F*L+I, где L – размер страницы. Так как умножение выполняется медленно, значение L выбирается кратным степени двойки. В этом случае умножение заменяется на быструю операцию сдвиг влево на число разрядов равное степени двойки.

Рассмотрим свойства страничной организации памяти. Прежде всего , необходимо отметить, что загружаемым в память объектом является страница некоторой задачи а не её код целиком. Кроме того, при данной организации памяти обеспечивается автоматическое перекрытие страниц в памяти (на место в памяти, где находилась некоторая страница, возможна загрузка новой необходимой страницы). Уровень фрагментации памяти при страничной организации значительно ниже, чем при сегментной организации памяти. Но для страничной организации памяти характерна *внутренняя фрагментация*. Внутренняя фрагментация памяти образуется из-за незаполненности последней страницы кода задачи. В среднем, последняя страница задачи незаполнена полезным кодом на 50%. Поэтому важным вопросом является выбор значения размера страниц. Выбор размера страниц основывается на двух противоречивых соображениях. Чем больше размер страницы, тем выше уровень внутренней фрагментации. С другой стороны, если размер страницы делать небольшим, увеличивается интенсивность свопинга, что может значительно увеличить время выполнения отдельных задач и понизить эффективность работы вычислительной системы в целом. Поэтому размеры страниц выбираются в пределах

$$2^{8} <= L <= 2^{12} \text{ байт.}$$

3. 5. Свопинг. Стратегии свопинга.

Свопинг (swaping) - это необходимый механизм при организации виртуальной памяти, так как с его помощью реализуется автоматическое перекрытие страниц или сегментов в оперативной памяти. Свопинг состоит из двух процедур – процедуры *подкачки* страниц (сегментов) из внешней памяти в оперативную, и процедуры *откачки* страниц (сегментов) из оперативной памяти во внешнюю. Для этого ОС выделяет специальную область внешней памяти, где могут храниться временно откачанные страницы или сегменты задач. Существуют различные стратегии (дисциплины) подкачки и откачки, которые требуют специального рассмотрения.

Стратегии подкачки. Принципиально могут рассматриваться только два метода подкачки необходимых страниц (сегментов) – это *опережающая подкачка* и *подкачка по требованию.*

Опережающая подкачка основывается на том, что поведение программ при их выполнении можно предсказать. Следовательно, можно заранее определить – какие страницы или сегменты задачи необходимо сразу загрузить в оперативную память. В действительности, поведение программ, в подавляющем большинстве случаев, непредсказуемо. Поэтому идея опережающей подкачки не нашла практического воплощения.

Подкачка по требованию состоит в загрузке необходимой страницы (сегмента), когда к данной странице произошло фактическое обращение. Данная стратегия реализуется относительно просто, но при этом она способствует увеличению интенсивности свопинга, что является недостатком стратегии.

Стратегии откачки (вытеснения). Откачка страниц необходима, когда в оперативной памяти отсутствуют свободные физические страницы. В этом случае необходимо выбрать занятую физическую страницу для её откачки во внешнюю память. Существует доказанное положение о том, что прежде всего необходимо откачивать такую страницу, следующее обращение к которой будет нескоро. На основе данного положения были реализованы различные стратегии откачки, отличающиеся друг от друга сложностью реализации и показателями качества.

Случайная откачка предполагает случайный выбор страницы для откачки. Данная стратегия проста в реализации, но плохо согласуется с вышеуказанным положением. При случайной откачке слишком часто будут откачиваться нужные страницы.

FIFO – “первым подкачан – первым откачан”. При данной стратегии прежде всего откачивается та страница, которая дольше остальных находится в оперативной памяти. Если обращения к памяти носят последовательный характер, то откачиваемая страница становится ненужной на достаточно длительный период времени. Тем не менее, при такой стратегии часто удаляются нужные страницы. Основным достоинством FIFO является простота реализации, так как планировщику памяти достаточно следить за очередью физических страниц, упорядоченных по времени подкачки.

LRU (least recently used) – определяет факт давности использования страниц, т. е. прежде всего, откачивается та страница, к которой не было обращений в течение некоторого интервала времени. Данная стратегия даёт хорошие результаты, но сложна в реализации. Сложность реализации объясняется тем, что ОС, совместно с аппаратурой, должна создавать и оперативно изменять список страниц, которым не было обращений в течение некоторого заданного интервала времени.

LFU (least frequently used) – определяет факт частоты использования страниц, т. е. прежде всего, откачивается та страница, к которой было меньше всего обращений за заданный интервал времени. Планировщик, однако, не должен учитывать страницы, которые были подкачаны в память только что. Иначе такие страницы станут первыми кандидатами на откачку. Данная стратегия также достаточно сложна в реализации, но даёт хорошее качество с точки зрения ранее указанного положения.

Механизм своппинга необходим при организации виртуальной памяти, но так как при своей реализации он требует значительного времени (обмен с внешней памятью) желательно минимизировать его использование.

3.6 Сегментно-страничная организация памяти.

При сегментно-страничной организации памяти транслятор подготавливает задачу как совокупность виртуальных сегментов, в общем случае, разной длины. Каждый виртуальный сегмент состоит из целого числа виртуальных страниц одинакового размера. Виртуальные адреса представляют собой тройки типа (s,p,i), где s – задаёт виртуальный номер сегмента, p – виртуальный номер страницы внутри сегмента, i – смещение относительно начала страницы. Физическая память представляется как совокупность физических страниц той же длины, что и виртуальные, поэтому, загружаемой в память единицей является страница. При загрузке очередной задачи в оперативную память операционная система совместно с аппаратурой заводит в памяти таблицу дескрипторов сегментов задачи, а для каждого такого сегмента – таблицу дескрипторов страниц. При активизации задачи в специальный регистр процессора заносится начальный адрес соответствующей таблицы дескрипторов сегментов. Другими словами, в этом регистре всегда находится значение адреса таблицы дескрипторов сегментов той задачи, которая выполняется в данный момент времени. Схема отображения виртуальных адресов в физические адреса во время выполнения программы представлена на рис. 10.

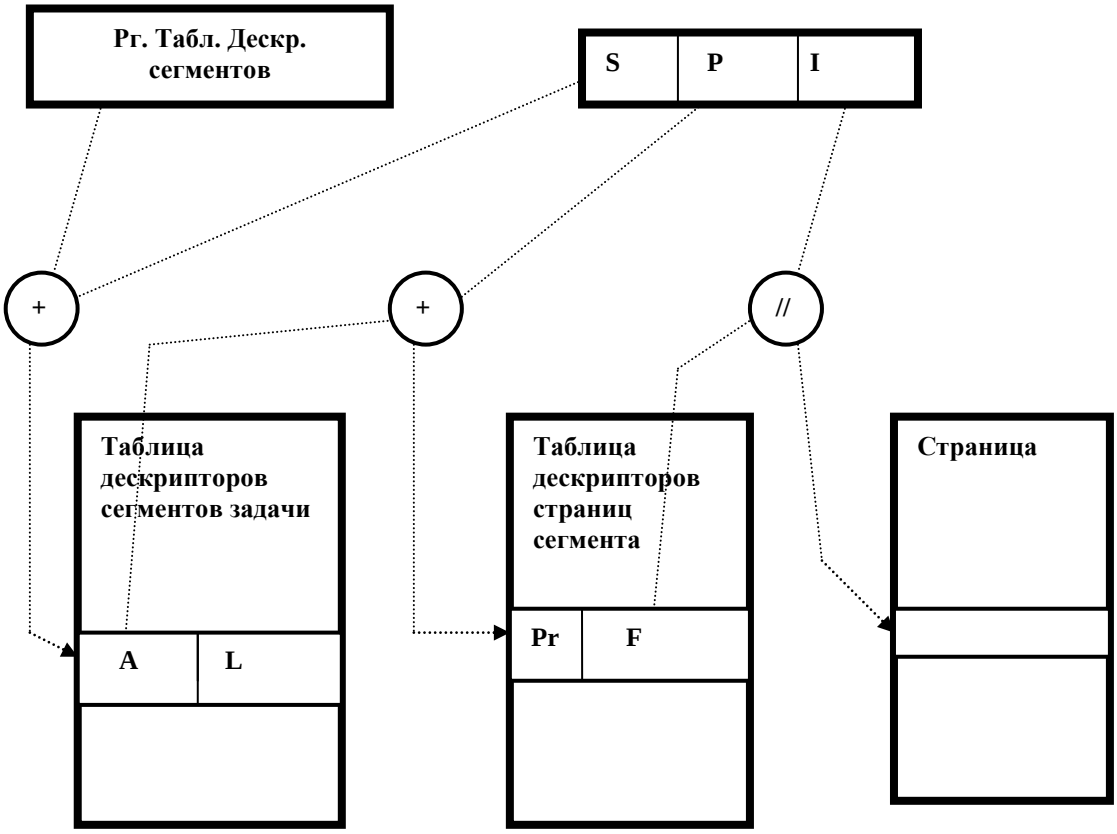


Рис. 10.

Здесь A – начальный адрес таблицы дескрипторов страниц сегмента с виртуальным номером S, L – размер сегмента в станицах, Pr – признак присутствия/отсутствия страницы в памяти, F – физический номер страницы. Отображение виртуальных адресов в физические адреса производится в три этапа – обращение к соответствующей таблице дескрипторов сегментов с целью определение нужного дескриптора сегмента, обращение к таблице дескрипторов страниц для определения необходимого дескриптора страницы, и обращение к соответствующей физической странице путём выполнения операции приписывания. Такая схема отображения реализуется недостаточно быстро, поэтому используется дополнительный механизм отображения – ассоциативная (кэш) память.

Ассоциативная память это память, выборка из которой осуществляется по значению некоторого подаваемого на вход ключа, а не по адресу, как это делается обычно. Каждая ячейка такой памяти состоит из двух полей – поля аргумента (X) и поля функции (Y). При её использовании, параллельно на вход всех ячеек посылается значение ключа (Key). Если значение ключа совпадает со значением аргумента, на выходные шины памяти выдаётся значение функции (см. рис. 11).

Если использовать ассоциативную память как дополнительный механизм применительно к сегментно-страничной организации памяти, то в качестве аргумента и функции имеет смысл использовать следующие соотношения:

$$X = (S, P), \quad Y = F.$$

В этом случае пара (S, P) однозначно определяет виртуальную страницу в сегменте, а F задаёт номер соответствующей физической страницы.

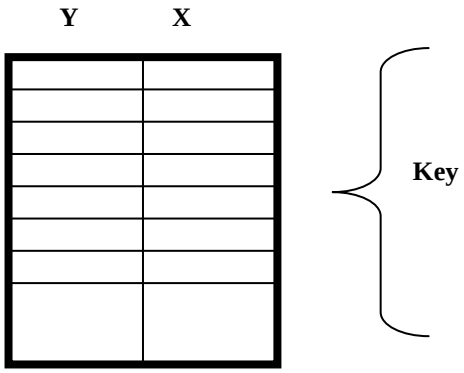


Рис. 11.

Для того чтобы понять, как механизм ассоциативной памяти дополняет основную схему отображения виртуальных адресов в физические адреса, представим следующую ситуацию. Производится первое обращение к некоторой странице. В этом случае срабатывает основная схема определения физического адреса. Но, одновременно, в ячейку ассоциативной памяти занесутся соответствующие значения аргумента и функции. Вероятность того, что следующее обращение будет к той же самой странице очень велика. И , если это так, то при следующем обращении сработает механизм ассоциативной памяти, определится значение F, затем быстро реализуется операция приписывания и определится значение физического адреса. Пока будут происходить обращения к данной странице, будет срабатывать быстрый

механизм ассоциативной памяти. При обращении к новой странице, аргумент и функция которой не прописаны в ассоциативной памяти, будет срабатывать основная схема определения физического адреса внутри страницы.

Ассоциативная память имеет относительно небольшой размер, поэтому возможна ситуация когда в памяти нет незаполненных ячеек а произошло обращение к новой странице. В этом случае необходимо решать задачу замещения ячеек ассоциативной памяти. Данная задача решается на основе использования дисциплины замещения LFU, то есть выбирается для замещения та ячейка памяти, в которой хранится информация (значения аргумента и функции) о странице с наименьшей частотой обращения за некоторый период времени.

4. Методы синхронизации параллельных процессов

4.1. Проблемы синхронизации параллельных процессов.

Функционирование мультипрограммной вычислительной системы характерно тем, что в её среде одновременно развиваются несколько параллельных процессов. В своём развитии параллельные процессы часто используют одни и те же ресурсы системы, т. е. разделяют их. Часть таких разделяемых ресурсов требуют только последовательного использования со стороны процессов, т. е. в каждый момент времени только один процесс может использовать разделяемый ресурс. Такие ресурсы называются *критическими*. Для того чтобы обеспечить последовательное использование критических ресурсов необходимо синхронизировать доступ к ним.

Задача синхронизации, в общем случае, состоит в следующем. Если несколько процессов хотят пользоваться критическим ресурсом в режиме разделения, им следует синхронизировать свои действия таким образом, чтобы такой ресурс всегда находился в распоряжении не более чем одного из них. Если один процесс пользуется в данный момент критическим ресурсом, то все остальные процессы, которым нужен этот ресурс, должны получить отказ в доступе и ждать, пока он не освободится. Если в системе не предусмотрена защита от одновременного доступа процессов к критическим ресурсам, в ней могут возникать ошибки, которые трудно обнаружить и исправить. Основной причиной возникновения таких ошибок является то, что процессы развиваются с разными скоростями, причём эти скорости самим процессам неподвластны и неизвестны друг другу. Рассмотрим несколько примеров, как отсутствие синхронизации при доступе к критическим ресурсам влияет на результаты выполнения процессов.

Пусть два конкурирующих процесса P1 и P2 асинхронно увеличивают значение общей переменной X, предварительно считывая её значение в свои локальные области памяти R1 и R2.

P1: (1) R1:=X; (2) R1:=R1+1; (3) X:=R1;

P2: (4) R2:=X; (5) R2:=R2+1; (6) X:=R2;

Поскольку процессы P1 и P2 могут иметь различные скорости выполнения, то может иметь место любая последовательность выполнения операций во времени. Например, если в промежуток времени между выполнением операций 1 и 3 будет выполнена хотя бы одна из операций 4-6, то значение переменной X будет не (X+2), а (X+1). Если предположить, что

процессы P1 и P2 осуществляли продажу билетов, и переменная X фиксировала количество уже проданных билетов, то в результате некорректного взаимодействия было бы продано несколько билетов на одно и то же место.

В качестве второго примера приведём пару процессов, которые изменяют различные поля записей служащих какого либо предприятия. Процесс АДРЕС изменяет домашний адрес служащего, а процесс СТАТУС – его должность и зарплату. Каждый процесс копирует всю запись в свою рабочую область памяти. Пусть каждый процесс должен обработать запись ИВАНОВ. Предположим, что после того как процесс АДРЕС скопировал запись ИВАНОВ в свою рабочую область, но до того как он записал скорректированную запись обратно, процесс СТАТУС скорректировал первоначальную запись ИВАНОВ в свою рабочую область. Изменения, выполненные тем из процессов, который первым запишет скорректированную запись назад в файл СЛУЖАЩИЕ, будут потеряны и, возможно, никто не будет знать об этом.

Чтобы предотвратить некорректное исполнение конкурирующих процессов вследствие нерегламентированного доступа к разделяемым переменным, необходимо ввести такое понятие как *взаимное исключение*, которое не позволит двум или более процессам параллельно обращаться к разделяемым переменным. Кроме реализации в ОС средств, организующих взаимное исключение, в ней должны быть средства, синхронизирующие работу взаимодействующих процессов. Использование таких средств позволит взаимодействующим процессам корректно обмениваться данными, чтобы правильно выполнялась их общая работа. Типичным примером взаимодействующих процессов, использующих разделяемые ресурсы, является задача ПОСТАВЩИК-ПОТРЕБИТЕЛЬ. Решение данной задачи будет рассмотрено в следующих лекциях.

Таким образом, при организации различного рода взаимодействующих процессов необходимо решать проблему корректного доступа к общим переменным, которые идентифицируют критические ресурсы с программной точки зрения. Те места в программах, в которых происходит обращение к критическим ресурсам, называются *критическими интервалами* или *критическими секциями* (critical section). Решение этой проблемы заключается в организации такого доступа к критическому ресурсу, когда только одному процессу разрешается входить в свою критическую секцию.

Когда какой-либо процесс находится в своём критическом интервале, другие процессы могут продолжать своё выполнение, но без входа в свои критические секции. Взаимное исключение необходимо только в том случае, когда процессы обращаются к разделяемым, общим данным. Если же процессы выполняют операции, которые не приводят к конфликтным ситуациям, они должны иметь возможность выполняться параллельно. Когда процесс выходит из своего критического интервала, то одному из остальных процессов, ожидающих входа в свои критические секции, должно быть разрешено продолжить выполнение, т. е. разрешено войти в свой критический интервал.

Задача взаимного исключения может быть сформулирована следующим образом:

- в любой момент времени только один процесс должен находиться в своей критической секции;
- ни один процесс не должен находиться в своей критической секции бесконечно долго;
- ни один процесс не должен бесконечно долго ждать входа в свою критическую секцию.

Процесс, находящийся вне своей критической секции, не должен блокировать выполнение других процессов, ожидающих входа в свои критические интервалы. Если два процесса одновременно хотят войти в свои критические интервалы, то принятие решения о том, кто из них должен это сделать, не должно бесконечно откладываться. Если процесс, находящийся в своём критическом интервале, завершается естественным или аварийным путём, режим взаимоисключения должен временно отменяться для того, чтобы один из других процессов смог войти в свою критическую секцию.

Для решения проблемы синхронизации параллельных процессов было предложено множество различных подходов, которые стали основой для программно-аппаратных средств из состава современных ОС.

4.2. Синхронизация параллельных процессов с помощью команды “Проверка и установка”.

Команда “Проверка и установка” входит в состав системы команд компьютеров, имеющих различные архитектуры. Данная команда имеет аббревиатуру TS или TAS (test and set), и имеет два операнда. При выполнении команды последовательно реализуются два действия: сначала значение второго операнда присваивается первому операнду, а затем второй операнд устанавливается в единицу. Команда TS является неделимой операцией, т. е. её выполнение не может быть прервано. Рассмотрим решение задачи взаимного исключения с помощью команды TS на примере двух параллельных процессов. Описание решения приводится в нотации языка параллельный Pascal.

```
Var  common, local1, local2;
Begin
  Common:=0;
Parbegin
  P1:   while true do
          Begin
            …
            local1:=1;
            while local1=1 do TS(local1, common);
            CS1;
            Common:=0;
            …
          End;
 And
 P2:   while true do
          Begin
            …
            local2:=1;
            while local2=1 do TS(local2, common);
            CS2;
            Common:=0;
            …
          End;
Parend
End.
```

Ключевые слова parbegin и parend означают, что между ними имеется описание параллельных процессов. Ключевое слово and означает связку между описаниями конкретных параллельных процессов. Бесконечный цикл while true do означает, что процесс выполняется неопределённое время.

Переменная common является общей для процессов P1 и P2. Её первоначальное значение равно нулю. Предположим, что первым начнёт выполняться процесс P1. Перед тем как войти в свою критическую секцию (CS1), он установит свою переменную local1 в единицу и войдёт в цикл while. Этот цикл P1 выполнит один раз, так как начальное значение common равнялось нулю. При выполнении цикла common установится в единицу (см. действия команды TS). Затем P1 войдёт в свой критический интервал (CS1). Если в этот момент управление получит P2 и захочет войти в свой критический интервал (CS2), то он сначала установит свою переменную local2 в единицу, а затем войдёт в цикл while и зациклится здесь (см. действия команды TS). Если через некоторое время управление получит P1 и выполнит свой критический интервал до конца, а также установит common в ноль, процесс P2, при получении управления, сможет войти в свой критический интервал. Другими словами, пока один из процессов находится в своем критическом интервале, другой процесс не может войти в свой критический интервал, т. е. задача взаимного исключения решается.

Основным недостатком предложенного решения является эффект “активного ожидания”. Активное ожидание проявляется в том, что процессы выполняя цикл while (ожидая разрешения на вход в свой критический интервал), впустую потребляют процессорное время.

4.3. Семафорные примитивы Дейкстры.

Семафор – это переменная специального типа, над которой определены две операции – *закрытие семафора* (P) и *открытие семафора* (V). Обобщённый смысл операции P состоит в декрементации числового поля семафора и в проверке значения этого поля семафора. Если это значение оказывается меньше некоторой величины (чаще всего нуля), процесс, вызвавший данную операцию блокируется и помещается в очередь к семафору.В противном случае процесс продолжает своё выполнение. Операция V связана с инкрементацией числового поля семафора. При этом, если выполняется некоторое условие, один из ранее заблокированных процессов деблокируется, т. е. покидает очередь к семафору и переходит в состояние готовности. Обычно семафор логически связывается с некоторым критическим ресурсом. Поэтому заблокированные процессы, находящиеся в очереди к семафору, косвенно ожидают доступа к критическому ресурсу.

Операции P и V являются неделимыми операциями, т. е. их выполнение не может прерываться. Действия по блокированию и деблокированию процессов реализуют модули из состава ядра ОС.

Допустимыми значениями числовых полей семафоров являются целые числа. Различают два вида семафоров – числовые и двоичные. Числовые семафоры – это семафоры, числовые поля которых могут принимать любые целые значения в некотором заданном диапазоне. Двоичные семафоры – это семафоры, числовые поля которых могут принимать только два значения :

единица и ноль. Существуют различные реализации семафорных примитивов. Они отличаются друг от друга по различным параметрам (тип семафоров, диапазон изменения значений числовых полей семафоров, логика операций, дисциплина выбора процесса при его деблокировании и т. д.). Рассмотрим некоторые алгоритмы работы семафорных примитивов. Сначала представим вариант алгоритма реализации операций P и V для числовых семафоров:

```
P(S):   S:=S-1;
        If  S<0 then <заблокировать процесс, и поместить
                      его в очередь к семафору>
V(S):   S:=S+1;
        If  S<=0 then  <деблокировать один из ранее заблокированных
                       процессов>
```

Алгоритм работы семафорных примитивов P и V для двоичных семаров может выглядеть следующим образом:

```
P(S):   if S=1 then  begin S:=0;  L:=0) end
         Else
          Begin
            L:=L+1;
            <заблокировать процесс>;
          End;

V(S):   if  (S=0) and (L>0) then
         Begin
           <деблокировать один из процессов>;
           L:=L-1;
         End;
           If  L=0 then S:=1;
```

Здесь L – длина очереди заблокированных процессов.

Решение задачи взаимного исключения с помощью семафорных примитивов для двух параллельных процессов можно представить следующим образом:

```
Var  s: semaphor;
Begin  s:=1;
Parbegin
  P1:  while true do begin
         … P(s);  CS1;  V(s);  …
       end;
   and
 P2:  while true do begin
         … P(s);  CS2;  V(s);  …
       end;
parend
end.
```

Предположим, что первым начнёт выполняться процесс P1. Прежде чем войти в свой критический интервал (CS1), он вызовет операцию P(s). После её выполнения значение числового поля семафора станет равным нулю, но P1 войдёт в свой критический интервал. Если в этот момент процесс P2 получит управление и захочет войти в свой критический интервал (CS2), он сначала вызовет выполнение операции P(s) и заблокируется по семафору s. Если через

некоторое время процесс P1 опять получит управление и выполнит свой критический интервал до конца, а также вызовет операцию V(s), процесс P2 деблокируется и сможет войти в свой критический интервал. Задача взаимного исключения будет решена, так как только один из двух процессов будет находиться в своём критическом интервале.

Использование семафорных примитивов для решения задач синхронизации имеет значительное достоинство: в случае блокировки, процессы попадают в состояние пассивного ожидания, не требуя использования процессорного времени.

4.4. Задача “Поставщик - Потребитель”.

Взаимодействуют два процесса – *Поставщик и Потребитель*. Поставщик создаёт сообщения и записывает их в пул буферов (для каждого сообщения свой буфер). Потребитель считывает сообщения из пула буферов для дальнейшего анализа и обработки. Параллельные действия по записи в пул и чтению из него запрещены. Пул буферов имеет конечную размерность. Поэтому, оба процесса должны корректно изменять число свободных и занятых буферов в пуле, как при записи очередного сообщения, так и при его считывании из пула. Решение данной задачи, основанное на использовании семафорных примитивов, представлено ниже:

```
Const N=100;
Var    S, SS, SN: semaphore;
Begin
  S:=1;  SS:=0; SN:=N;
Parbegin
 Поставщик:     while true do begin
                 … { генерация собщения}
                 P(SN);   P(S);
                 … {запись сообщения}
                 V(SS);   V(S);
                 End;
 And
 Потребитель:  while true do begin
                P(SS);   P(S);
                …{ чтение сообщения}
               V(SN);    V(S);
                … {обработка сообщения}
               End;
Parend
End.
```

Здесь N – число буферов в пуле, S – двоичный семафор, используемый для регулирования доступа к пулу буферов как при записи так и при считывании, SS и SN – числовые семафоры, используемые как счётчики числа занятых и свободных буферов в пуле соответственно.

Для того чтобы понять представленное решение рассмотрим несколько возможных ситуаций развития взаимодействующих процессов.

Пусть пул буферов частично занят (числовые поля SS и SN имеют целые положительные значения), и управление получает Поставщик. Он создаст новое сообщение, последовательно выполнит P(SN) и P(S), и начнёт запись. Если в этот момент управление получит Потребитель, то он выполнит P(SS), затем P(S), и здесь заблокируется по семафору S. Потребитель сможет деблокироваться только после того, как Поставщик закончит запись и выполнит операцию V(S). Другими словами, пока очередная запись в пул не закончена чтение из пула невозможно, и наоборот.

Пусть пул буферов пуст, а управление получил Потребитель. Он выполнит операцию P(SS) и тут же заблокируется по семафору SS (числовое поле семафора SS к этому моменту равнялось нулю). Другими словами, чтение из пула запрещено, если он не содержит сообщений.

Пусть пул буферов полностью заполнен сообщениями, а управление получил Поставщик. Он выполнит операцию P(SN) и заблокируется по семафору SN (значение числового поля семафора SN к этому моменту будет равняться нулю). Другими словами, запись в пул запрещается, если он полностью заполнен сообщениями.

4.5. Задача “Читатели - Писатели”.

Некоторая база данных используется двумя типами параллельных процессов – *Читателями и Писателями*. Читатели считывают информацию из базы данных, а Писатели модифицируют содержимое базы данных. Модификация содержимого базы данных должна выполняться в монопольном режиме, т. е., если некоторый Писатель получил доступ к базе, то остальные Писатели и Читатели должны ждать до тех пор, пока этот Писатель закончит свои действия с базой данных. Читатели относительно друг друга могут считывать информацию из базы параллельно, но, при этом, любая запись в базу должна быть запрещена. Ниже представлено решение данной задачи, основанное на использовании семафорных примитивов.

```
Var  S, W : semaphore;
       count_r: integer;
Procedure Reader;
 Begin
   P(S);   count_r:=count_r+1;
   If count_r=1 then P(W);
   V(S);
  Read_DB;
  P(S); count_r:=count_r-1;
  If count_r=0 then V(W);
  V(S);
 End;
Procedure Writer;
 Begin
   P(W);   write_DB;  V(W);
 End;
Begin
  S:=1;   W:=1;   count_r:=0;
End.
```

При доступе к базе данных Читатели используют процедуру Reader, а Писатели процедуру Writer. Здесь W – двоичный семафор, используемый для регулирования доступа к базе, S – двоичный семафор, используемый для корректного доступа к переменной count_r, count_r – счётчик параллельно выполняемых Читателей.

Монопольность записи в базу обеспечивается следующим образом. Предположим, что для некоторого Писателя осуществляется запись в базу (write_DB), и в этот момент управление получит другой Писатель. Для него, при выполнении процедуры Writer, сначала выполнится оператор P(W), и этот Писатель заблокируется по семафору W. Следующие Писатели будут также блокироваться по семафору W. Если же в этот момент управление получит Читатель, то для него последовательно выполнятся операции P(S), инкрементация счётчика читателей, и по условию (count_r = 1) операция P(W) (см. процедуру Reader). Здесь Читатель заблокируется по семафору W. Следующие Читатели будут блокироваться по семафору S.

Обеспечение параллельности чтения с одновременным запретом на запись в базу можно представить следующим образом. Предположим, некоторый Читатель занимается чтением из базы (read_DB), и в этот момент управление получает новый Читатель. Он также сможет выполнять свои действия по чтению содержимого базы данных (см. текст процедуры Reader). Если же в этот момент управление получит Писатель, то для него выполнится операция P(W), и Писатель заблокируется по семафору W, так как для первого Читателя операция P(W) уже выполнялась. Модификация содержимого базы данных сможет возобновиться только после того, когда закончит чтение последний из параллельно выполняемых Читателей (см. условие count_r = 0 в процедуре Reader). Для того, чтобы обновление базы данных происходило быстрее, Писатели имеют более высокий приоритет по сравнению с Читателями, т. е. при выполнении операции V(W), прежде всего, деблокируются Писатели.

Тем не менее, при интенсивном потоке запросов на чтение, обновление базы данных может значительно задерживаться во времени, что является недостатком представленного решения задачи. Рассмотрим решение свободное от вышеуказанного недостатка.

```
Var  S, W, R : semaphore;
       count_r: integer;
Procedure Reader;
 Begin
   P(R);  P(S);   counr_r:=counr+1;
   If count_r=1 then P(W);
   V(S);  V(R);
  read_DB;
  P(S); countr:=count_r-1;
  If count-r=0 then V(W);
  V(S);
 End;
Procedure Writer;
 Begin
   P(R);  P(W);   write_DB;  V(W);  V(R);
 End;
```

```
Begin
  S:=1;   W:=1;   R:=1;  count_r:=0;
End.
```

Для данного решения характерно использование ещё одного двоичного семафора R. Предположим, что N Читателей занимаются чтением содержимого базы данных, т. е. каждый из них находится в некоторой точке вычислений комплекса действий, определённых как read_DB. В этот момент управление получает Писатель, и для него начинает выполняться процедура Writer. Сначала выполнится операция P(R), а затем P(W), и Писатель заблокируется по семафору W, так как первый из Читателей уже выполнял эту операцию. Если далее управление получит N+1-й Читатель, то для него выполнится операция P(R), и он заблокируется по семафору R. После того как первые N Читателей закончат чтение, последний из них выполнит операцию V(W), и ранее заблокированный Писатель деблокируется и сможет модифицировать содержимое базы данных. Другими словами, с помощью семафора R временно блокируется поток запросов на чтение, что способствует ускорению обновления содержимого базы данных.

4.6. Задача с ожиданием.

Содержательная постановка задачи состоит в следующем. Для того чтобы первый процесс P1 смог выполнить свою работу целиком, он должен передать управление второму процессу P2 и дождаться его окончания. Решение такой задачи с помощью семафорных примитивов может выглядеть следующим образом.

```
Var S: semaphore;
Begin
   S:=0;
Parbegin
 P1:    …
        Exec(P2);
        P(S);
        …
     And
 P2:    …
        V(S);
Parend
End.
```

При решении используется двоичный семафор S, причём, начальное значение числового поля этого семафора равняется нулю. После того как P1 выполнит часть своих действий, он инициирует запуск процесса P2 с помощью директивы Exec. Если P1 далее продолжит своё выполнение, то после реализации операции P(S), он заблокируется по семафору S. P1 сможет деблокироваться и продолжить своё выполнение только после того, как P2 выполнит все свои действия и операцию V(S).

4.7. Почтовые ящики

Почтовые ящики относятся к высокоуровневым средствам организации взаимодействий между параллельными процессами. Почтовые ящики предназначены для хранения и передачи сообщений между взаимодействующими процессами. Для хранения посланного, но ещё не полученного сообщения необходимо выделить некоторый буфер в памяти, он будет исполнять роль почтового ящика.

Если процесс P1 хочет общаться с процессом P2, то P1 просит ОС образовать почтовый ящик, который свяжет эти два процесса так, чтобы они могли передавать друг другу сообщения. Для того чтобы послать процессу P2 какое-то сообщение, процесс P1 просто помещает это сообщение в почтовый ящик, откуда процесс P2 может его взять в любое время. При использовании почтового ящика процесс P2 в конце концов обязательно получит сообщение, когда обратится за ним.

Если объем передаваемых данных велик, то эффективнее не передавать их непосредственно, а отправлять в почтовый ящик сообщение, информирующее процесс получатель о том, где их можно найти (адресная ссылка).

Почтовый ящик может быть связан только с парой процессов (отправитель и получатель), а может быть связан и с большим числом процессов. Почтовый ящик, связанный только с процессом получателем, облегчает посылку сообщений от нескольких процессов в фиксированный пункт назначения. Если почтовый ящик не связан с конкретными процессами, то сообщение должно содержать идентификаторы отправителя и получателя.

Почтовый ящик – это информационная структура, поддерживаемая ОС. Она состоит из головного элемента (заголовка), в котором находится информация о характеристиках почтового ящика, и из нескольких буферов (гнёзд), в которые помещаются сообщения. Размер каждого гнезда и количество гнёзд обычно задаются при образовании почтового ящика.

Правила работы почтовых ящиков могут разные в зависимости от их сложности. В простейшем случае сообщения передаются только в одном направлении. Процесс P1 может посылать сообщения до тех пор, пока имеются свободные гнёзда. Если все гнёзда заполнены, то процесс P1 вынужден ждать когда хотя бы одно из гнёзд освободится. Аналогично, процесс P2 может получать сообщения до тех пор, пока имеются заполненные гнёзда. Если сообщений нет в почтовом ящике, P2 будет ждать их появлений. Такие почтовые ящики называются *однонаправленными*. Реализацией однонаправленного почтового ящика является решение задачи “Поставщик-Потребитель”.

Двунаправленный почтовый ящик, связанный с парой процессов, используется для подтверждения сообщений. Если используется множество гнёзд, то каждое из них может хранить либо сообщение, либо подтверждение (ответ). Чтобы гарантировать передачу подтверждений, когда все гнёзда заняты, подтверждение на сообщение помещается в то же гнездо, которое использовалось для хранения сообщения, и оно уже не используется для хранения другого сообщения до тех пор, пока подтверждение не будет получено. Из-за того, что некоторые процессы не забрали свои сообщения, связь может быть приостановлена. Если каждое сообщение снабдить пометкой времени появления в почтовом ящике, то системная управляющая процедура

может уничтожать старые сообщения и освобождать гнёзда. Процессы могут быть также остановлены в связи с тем, что другие процессы не смогли послать им сообщения или подтверждения. Если время поступления сообщений в почтовый ящик регистрируется, то управляющая процедура может им периодически посылать процессам пустые сообщения или подтверждения, чтобы они не ждали слишком долго.

При реализации почтовых ящиков используются низкоуровневые средства, например - семафорные примитивы и т. п.

Рассмотрим интерфейс операций с двунаправленным почтовым ящиком, которые могут использовать процессы при своём взаимодействии:

1. Send_message (receiver, message, buffer) – посылка сообщения (message) получателю (receiver) через почтовый ящик (buffer). Процесс, выдавший данную операцию, продолжает своё выполнение.
2. Wait_message (sender, message, buffer) – процесс, выдавший данную операцию ждёт до тех пор, пока в почтовом ящике не появится сообщение от отправителя (sender) и не перепишется в собственную память получателя (message).
3. Send_answer (result, answer, buffer) – записывает ответ (answer) в тот буфер, в который было записано сообщение.
4. Wait_answer (result, answer, buffer) – блокирует процесс, выдавший данную операцию до тех пор, пока в почтовом ящике не появится ответ и не перепишется в собственную память процесса (answer). Значение переменной result определяет, каков ответ: от процесса получателя или пустой от ОС.

Совокупность представленных операций с двунаправленным почтовым ящиком позволяет организовать различные варианты взаимодействия двух или более процессов. Почтовые ящики удобны для обмена сообщениями, но их затруднительно использовать для решения задачи взаимоисключения при доступе к критическим ресурсам.

4. 8. Мониторы Хоара

Монитор – это пассивный набор разделяемых переменных и повторно входимых процедур доступа к ним, которым процессы пользуются в режиме разделения, причём в каждый момент времени им может пользоваться только один процесс. Монитор можно представить как комнату, от которой есть только один ключ. Если какой-то процесс намеревается воспользоваться этой комнатой и ключ находится снаружи, то этот процесс может отпереть комнату, войти в неё и воспользоваться одной из процедур монитора. Если ключа снаружи нет, то процессу придется ждать, пока тот, кто пользуется комнатой в данный момент, не выйдет из неё и не отдаст ключ. Кроме того, никому не разрешается в комнате оставаться навсегда.

Рассмотрим, например, некоторый ресурс, который выделяет соответствующий планировщик. Каждый раз, когда процесс желает получить этот ресурс, он должен обратиться к планировщику. Процедуру планировщик разделяют все процессы, и каждый из них может обратиться к планировщику в любой момент. Но планировщик не в состоянии обслуживать более одного процесса одновременно. Такая процедура планировщик представляет собой пример монитора.

Таким образом, монитор – это механизм организации параллелизма, который содержит как данные, так и процедуры, необходимые для динамического распределения конкретного общего ресурса или группы общих ресурсов. Процесс, желающий получить доступ к разделяемым переменным, должен обратиться к монитору, который либо предоставит доступ, либо откажет в нём. Вход в монитор находится под жёстким контролем – здесь осуществляется взаимоисключение процессов, так как в каждый момент времени только один процесс может пользоваться монитором. Процессам, которые хотят войти в монитор, когда он уже занят, приходится ждать, причём режимом ожидания управляет сам монитор. При отказе в доступе монитор блокирует обратившийся к нему процесс и определяет условие, по которому процесс ждёт. Проверка условия выполняется самим монитором, который и деблокирует ожидающий процесс.

Внутренние переменные монитора могут быть либо глобальными (используемыми всеми процедурами монитора), либо локальными (используемыми только в своих процедурах). Обращение к этим переменным возможно только изнутри монитора. При первом обращении к монитору инициализируются начальные значения переменных. При последующих обращениях используются те значения переменных, которые остались от предыдущего обращения.

Если процесс обращается к некоторой процедуре монитора и обнаруживается, что соответствующий ресурс уже занят, эта процедура монитора выдаёт команду WAIT с указанием условия ожидания. В этом случае, процесс, переводящийся в режим ожидания, будет ждать момента, когда необходимый ресурс освободится. Со временем процесс, который занимал данный ресурс, обратится к монитору, чтобы возвратить ресурс системе. В этом случае монитор выдаёт команду извещения (сигнализации) SIGNAL, чтобы один из ожидающих процессов мог получить данный ресурс и войти в монитор. Если монитор сигнализирует о возвращении ресурса, и в это время нет процессов, ожидающих такого ресурса, то он вносится в список свободных ресурсов.

Чтобы гарантировать, что процесс, находящийся в ожидании некоторого ресурса, со временем его получит, считается, что ожидающий процесс имеет более высокий приоритет, чем новый процесс, пытающийся войти в монитор.

Рассмотрим пример монитора для выделения одного ресурса.

```
Monitor resourse;
Condition free;   (*  условие - свободный*)
Var busy: Boolean;
Procedure Request; (* запрос*)
Begin
If busy then WAIT (free);
Busy := true;
Выдать ресурс;
End;
Procedure Release; (*  освобождение*)
Begin
Взять ресурс;
Busy := false;
SIGNAL (free);
```

```
End;
Begin
Busy := false;
End.
```

Единственный ресурс динамически запрашивается и освобождается процессами, которые обращаются к процедурам Request и Release. Если процесс обращается к процедуре Request в тот момент, когда ресурс используется, значение переменной busy будет true и Request выполнит операцию WAIT(free). Эта операция заблокирует обратившийся процесс, и он будет помещён в конец очереди процессов, ожидающих доступа к монитору. Когда процесс, использующий ресурс, обратится к процедуре Release, операция монитора SIGNAL деблокирует процесс, находящийся в начале очереди, не позволяя исполняться никакой другой процедуре внутри того же монитора. Этот деблокированный процесс готов возобновить выполнение процедуры Request.

Использование монитора в качестве средства синхронизации и связи освобождает процессы от необходимости явно разделять между собою ресурсы, так как доступ к разделяемым переменным всегда ограничен телом монитора. Поскольку мониторы входят в состав ядра ОС, то разделяемые переменные становятся системными переменными. Этот факт автоматически исключает критические интервалы, так как в каждый момент монитором может пользоваться только один процесс.

Использование мониторов имеет ряд преимуществ по сравнению с низкоуровневыми средствами:

- локализация разделяемых переменных внутри тела монитора позволяет избавиться от малопонятных программных конструкций в синхронизируемых процессах;
- мониторы дают возможность процессам совместно использовать программные модули, представляющие критические секции (если несколько процессов совместно и одинаково используют некоторый разделяемый ресурс, то в составе монитора достаточно иметь одну копию соответствующей процедуры работы с этим ресурсом).

5. Понятие тупика и методы борьбы с тупиками в вычислительных системах

5.1. Понятие тупика, примеры тупиков, условия существования тупиков.

При параллельном выполнении процессов в вычислительной системе могут возникать ситуации, при которых два или более процессов всё время находятся в заблокированном состоянии. Самым простым является случай, когда каждый из двух процессов ожидает ресурс, занятый другим процессом. Из-за такого ожидания ни один из процессов не может продолжит своё выполнение и освободить в конечном итоге ресурс, необходимый другому процессу. Такая ситуация называется тупиком или клинчем. Говорят, что в мультипрограммной системе процесс находится в состоянии тупика, если он ждёт событие, которое никогда не произойдёт. Тупики чаще всего возникают из-за конкуренции параллельных процессов за ресурсы вычислительной системы, но иногда к тупикам приводят ошибки в программировании.

При рассмотрении проблемы тупиков целесообразно понятие ресурсов вычислительной системы обобщить и разделить их все на два класса – повторно используемые или системные ресурсы (SS – System Resource) и потребляемые или расходуемые ресурсы (CR – Consumable Resource).

Системный ресурс (SR) есть конечное множество единиц со следующими свойствами:

- число единиц ресурса постоянно;
- каждая единица ресурса или доступна, или распределена одному и только одному процессу;
- процесс может освободить единицу ресурса, только, если он раньше получил эту единицу, т. е. никакой процесс не может оказывать какое-либо влияние ни на один ресурс, если он ему не принадлежит.

Примерами таких ресурсов являются компоненты аппаратуры (оперативная память, внешняя память, внешние устройства, процессор), а также программные и информационные компоненты (файлы, таблицы, переменные и т.п.).

Расходуемый ресурс (CR) отличается от ресурса типа SR в нескольких важных отношениях:

- число доступных единиц некоторого ресурса типа CR изменяется по мере того как приобретаются и освобождаются отдельные его элементы выполняемыми процессами, а также число единиц ресурса является потенциально неограниченным;
- процесс типа “Производитель” увеличивает число единиц ресурса, освобождая одну или более единиц;
- процесс типа “Потребитель” уменьшает число единиц ресурса, сначала запрашивая, а затем приобретая одну или более единиц.

Примерами таких ресурсов являются синхронизирующие сигналы, сигналы прерываний, сообщения, которыми обмениваются процессы.

Рассмотрим несколько примеров образования тупиков.

Пример тупика на ресурсах типа CR.

Пусть три процесса Пр1, Пр2, Пр3 вырабатывают сообщения М1, М2, М3 по кольцевой схеме. Процесс Пр1 является потребителем сообщения М3, процесс Пр2 должен получить сообщение М1, а Пр3 – М2. Таким образом, каждый из процессов одновременно является и “Поставщиком ” и “Потребителем”, и вместе они образуют кольцевую схему передачи сообщений друг другу. Если связь между процессами устанавливается в следующем порядке:

```
Пpi :    send_message (Пpk, Mi , Пяk);
         Wait_message ( Пpj, Mj , Пяi);
```

(где i=1, 2, 3 k=2, 3, 1 j=3, 1, 2), то никаких сложностей при взаимодействии процессов не возникает, и тупика не будет. Однако перестановка этих двух операций местами приводит к тупику:

```
Пр1:     wait_message(Пр3, М3, Пя1);
         Send_message(Пр2, М1, Пя2);

Пр2:     wait_message(Пр1, М1, Пя2);
         Send_message(Пр3, М2, Пя3);

Пр3:     wait_message(Пр2, М2, Пя3);
         Send_message(Пр1, М3, Пя1);
```

В самом деле, ни один из процессов не сможет послать сообщение до тех пор, пока сам его не получит, а это событие никогда не произойдёт, поскольку ни один процесс не может этого сделать.

Пример тупика на ресурсах типа CR и SR.

Пусть некоторый процесс Пр1 должен обменяться сообщениями с процессом Пр2 и каждый из них запрашивает некоторый ресурс R, причём Пр1 требует три единицы этого ресурса для своего выполнения, а Пр2 – две единицы и только на время обработки сообщения. Всего же у системы имеется только четыре единицы ресурса R. Запрос на ресурс R можно реализовать через процедуру монитора Request(R, N) – запрос N единиц ресурса R, и Release(R, N) – освобождение N единиц ресурса R. Обмен сообщениями производится через почтовый ящик Пя. Фрагменты программ процессов Пр1 и Пр2 будут выглядеть следующим образом:

```
Пр1:  …
      Request (R,3);
      …
      send_message (Пр2, сообщение, Пя);
      wait_answer (ответ, Пя);
      …
      Release (R, 3);
      …

Пр2:  …
      Wait_message ( Пр1, сообщение, Пя);
      Request (R, 2);
       Обработка сообщения;
      Release (R, 2);
      Send_answer (ответ, Пя);
      …
```

Эти два процесса всегда будут попадать в тупик. Процесс Пр2, если будет выполняться первым, сначала ожидает сообщение от Пр1, после чего будет заблокирован при запросе ресурса R, часть единиц которого уже будет отдана Пр1. Процесс Пр1, завладев тремя единицами ресурса R , будет заблокирован на ожидании ответа от Пр2, которого никогда не получит, так как для этого Пр2 нужно получить две единицы ресурса R, что невозможно. Тупик можно избежать лишь при условии, что на время ожидания ответа от Пр2 процесс Пр1 будет отдавать хотя бы одну единицу ресурса R, которым он владеет.

Пример тупика на ресурсах типа SR.

Предположим, что существуют два процесса Пр1 и Пр2, разделяющих два ресурса типа SR: R1 и R2. Пусть взаимное исключение доступа к этим ресурсам реализуется с помощью двоичных семафоров S1 и S2 соответственно. Начальные значения семафоров равны единице. Процессы Пр1 и Пр2 обращаются к ресурсам следующим образом:

```
Пр1:    …              Пр2:    …
     1: P(S2);                5: P(S1);
        …                        …
     2: P(S1);                6: P(S2);
        …                        …
```

3: V(S1); 7: V(S1);
… …
4: V(S2); 8: V(S2);
… …

Здесь несущественные детали с точки зрения доступа к ресурсам опущены. Оба семафора первоначально установлены в единицу. Пространство возможных вычислений приводится на рис. 12.

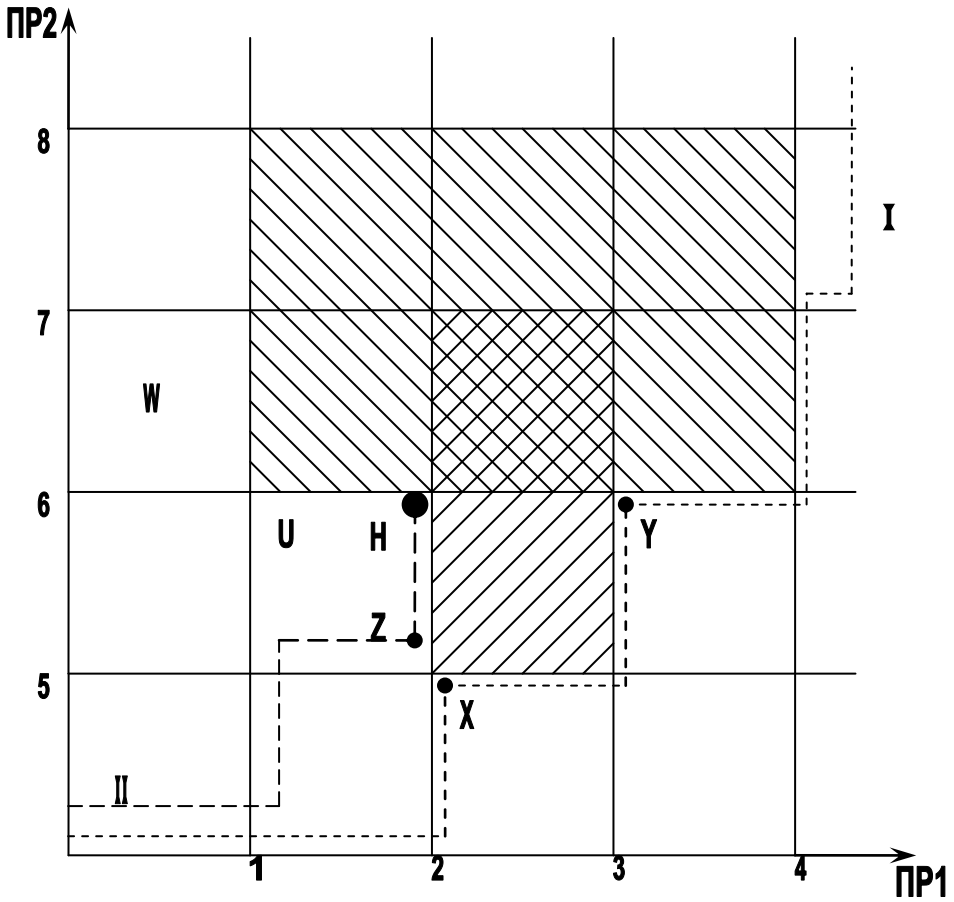


Рис. 12

Горизонтальная ось задаёт выполнение Пр1, а вертикальная Пр2. Вертикальные линии пронумерованные от 1 до 4, соответствуют операторам 1-4 процесса Пр1. Аналогично, горизонтальные линии, пронумерованные от 5 до 8, соответствуют операторам 5-8 процесса Пр2. Точка на плоскости определяет состояние вычислений в некоторый момент времени. Так, точка W соответствует ситуации, при которой Пр1 начал выполнение, но не достиг оператора 1 , а Пр2 выполнил операторы 5 и 6, но не достиг оператора 7. По мере выполнения процессов точка может двигаться горизонтально вправо, если выполняется Пр1, и вертикально вверх, если выполняется Пр2. Горизонтальные и вертикальные линии делят пространство вычислений на 25 прямоугольников (клеток), каждый из которых задаёт определённое состояние вычислений.

Заштрихованные состояния являются *недостижимыми* из-за взаимного исключения Пр1 и Пр2 при доступе к ресурсам R1 и R2.

Рассмотрим последовательность выполнения 1-2-5-3-6-7-8, представленную траекторией I. Когда Пр2 запрашивает R1 (оператор 5), R1 недоступен (семафор S1 закрыт). Поэтому, Пр2 заблокирован в точке X. Как только Пр1 достигнет оператора 3, Пр2 деблокируется по R1. Аналогично, в точке Y Пр2 будет заблокирован при попытке доступа к R2 (оператор 6). Как только Пр1 достигнет оператора 4, Пр2 деблокируется по R2. Данная последовательность выполнения процессов не приведёт к тупиковой ситуации, так как оба процесса рано или поздно получат доступ к ресурсам и смогут нормально завершиться.

Если же, например, выполняется последовательность 1–5-2-6, то Пр1 заблокируется в точке Z при выполнении оператора 2, а Пр2 заблокируется в точке H при выполнении оператора 6. При этом, Пр1 будет ждать, когда Пр2 выполнит оператор 7, а Пр2 будет ждать, когда Пр1 выполнит оператор 4. Оба процесса попадут в тупик и не смогут закончить своё выполнение. При этом, все ресурсы, которые получил Пр1, будут недоступны для Пр2, и наоборот. Тупик будет неизбежным, если выполнение процессов зашло в состояние U. Такое состояние называется **опасным** состоянием, а точка H является точкой тупика.

Для того чтобы возник тупик, необходимо, чтобы одновременно выполнялись четыре условия:

- взаимного исключения, при котором процессы осуществляют монопольный доступ к ресурсам;
- ожидания, при котором процесс, запросивший ресурс, ждёт до тех пор, пока его запрос не будет удовлетворён, при этом удерживая ранее полученные ресурсы;
- отсутствия перераспределения, при котором ресурсы нельзя отобрать у процесса, если они уже ему выделены;
- кругового ожидания, при котором существует замкнутая цепь процессов, каждый из которых ждёт ресурс, удерживаемый его предшественником в цепи.

Все эти четыре условия выполняются в точке H – точке тупика.

5.2. Методы борьбы с тупиками

Борьба с тупиковыми ситуациями основывается на одной из трех стратегий:

- предотвращение тупика;
- обход тупика;
- обнаружение тупика с последующим восстановлением работоспособности системы.

Предотвращение тупика основывается на предположении о его чрезвычайно высокой стоимости, поэтому лучше потратить дополнительные ресурсы системы, чтобы исключить вероятность его возникновения при любых обстоятельствах. Предотвращение можно рассматривать как запрет существования опасных состояний. Поэтому, дисциплина, предотвращающая тупик, должна гарантировать, что какое-либо из четырёх условий, необходимых для его наступления, не может

возникнуть.

Условие взаимного исключений можно подавить путем разрешения неограниченного разделения ресурсов. Это удобно для повторно входимых программ и ряда драйверов, но совершенно неприемлемо *к* совместно используемым переменным в критических интервалах.

Условие ожидания можно подавить, предварительно выделяя ресурсы. При этом, процесс может начать исполнение, только получив все необходимые ресурсы заранее. Следовательно, общее число затребованных параллельными процессами ресурсов должно быть не больше возможностей системы. Поэтому предварительное выделение момент привести к снижению эффективности работы вычислительной системы в целом. Необходимо также отметить, что предварительное выделение зачастую невозможно, так как необходимые ресурсы становятся известны процессу только после начала исполнения.

Условие отсутствия перераспределения можно исключить, позволяя операционной системе отнимать у процесса ресурсы. Для этого в операционной системе должен быть предусмотрен механизм запоминания состояния процесса с целью последующего восстановления. Перераспределение процессора реализуется достаточно легко, в то время как перераспределение устройств ввода-вывода крайне нежелательно.

Условие кругового ожидания можно исключить, предотвращая образование цепи. Это обеспечивается иерархическим выделением ресурсов. Все ресурсы образуют некоторую иерархию. Процесс, затребовавший ресурс на одном уровне, может затем потребовать ресурсы только на более высоком уровне. Процесс может освободить ресурсы на данном уровне только после освобождения всех ресурсов на всех более высоких уровнях. После того как процесс получил, а потом освободил ресурсы данного уровня, он может запросить ресурсы на том же самом уровне. Пусть имеется процессы Пр1 и Пр2, которые могут иметь доступ к ресурсам R1 и R2, причем R2 находится на более высоком уровне иерархия. Когда Пр1 захватил R1, то Пр2 не может захватить R2, так как доступ к нему проходит через доступ к R1, который уже захвачен Пр1. Таким образом, создание замкнутой цепи исключается. Иерархическое выделение ресурсов часто не дает никакого выигрыша, если порядок использования ресурсов, определенный в описании процессов, отличается от порядка уровней иерархии. В этом случае ресурсы будут использоваться крайне неэффективно.

Обход тупика можно интерпретировать как запрет входа в опасное состояние. Если ни одно из упомянутых четырех условий не исключено, то вход в опасное состояние можно предотвратить, при наличии у системы информации о последовательности запросов, связанных с каждым параллельным процессом. Доказано, что если вычисления находятся в любом неопасном состояния, то существует по крайней мере одна последовательность состояний, которая обходит опасное. Следовательно, достаточно проверить, не приведет ли выделение затребованного ресурса сразу же к опасному состоянию. Если да, то запрос отклоняется. Если нет, его можно выполнить. Определение того, является ли состояние опасным, или нет, требует анализа последующих

запросов процессов. Часто бывает так, что последовательность запросов, связанных с каждым процессом, неизвестна заранее. Но если заранее известен общий запрос на ресурсы каждого типа, то выделение ресурсов модно контролировать, В этом случае необходимо для каждого требования, в предположении, что оно удовлетворено, определить, существует ли среди общих запросов от всех процессов некоторая последовательность требований, которая может привести к опасному состоянию. Данный подход является примером контролируемого выделения ресурса.

Классические решение этой задачи известно как алгоритм банкира Дейкстры . Основным накладным расходом стратегии обхода тупика с помощью контролируемого выделения ресурса является время выполнения алгоритма, так как он выполняется при каждом запросе. Причем, алгоритм работает наиболее медленно, когда система близка к тупику. Необходимо отметить, что обход тупика неприменим при отсутствии информации о требованиях процессов на ресурсы.

Распознавание тупика основано на анализе модели распределения ресурсов. Один из алгоритмов использует информацию о состоянии системы, содержащуюся в двух таблицах: таблиц: таблице текущего распределения (назначения) ресурсов RATBL и таблице заблокированных процессов PWTBL (для каждого вида ресурса может быть свой список заблокированных процессов). При каждом запросе на получение или освобождении ресурсов содержимое этих таблиц модифицируется, а при запросе на ранее выделенный ресурс анализируется в соответствии со следующим алгоритмом:

1. Запрос от процесса Y на занятый ресурс I.

2. Поместить номер ресурса I в PWTBI. в строке с номером процесса Y.

3. Использовать I в качестве смешения а RATBL чтобы найти помер процесса К, который владеет ресурсом.

4. Использовать К в качестве смещения в PWTBL.

5. Проверить, ждет ли процесс К освобождения какого-либо ресурса I’. Если нет, то перейти к пункту 6, в противном случае - к пункту 7.

6. Перевести Y в состояние ожидания и выйти из алгоритма.

7. Использовать I’ в качестве смещения в RATBL, чтобы найти номер блокирующего его процесса К'.

8. Проверить К' = Y ? Если да, то перейти к пункту 9, в противном случае - к пункту 11.

9. Проверить, вся ли таблица PWI'BL просмотрена. Если да, то переход к пункту 6, в противном случае – к пункту 10.

10. Присвоение К := К' и переход *к* пункту 4.

11. Вывод о наличии тупика.
12. Конец алгоритма.

Для иллюстрации описанного алгоритма распознавания тупика рассмотрим следующую последовательность событий:

1. Пр1 занимает ресурс R1.

2. Пр2 занимает ресурс R3.

3. ПрЗ занимает ресурс R2.

4. Пр2 занимает ресурс R4.

5. Пр1 занимает ресурс R5.

В результате RATBL принимает вид:

Ресурсы	Процессы
1	1
2	3
3	2
4	2
5	1

б. Пусть Пр1 пытается занять R3. поэтому в соответствии с описанным алгоритмом - Y = 1, I =3, К =2 , процесс К не ждет никакого ресурса I', поэтому Пр1 блокируется по R3.

7. Далее, пусть Пр2 пытается занять R2; Y =2, I=2, К=3; Процесс К не ждет никакого ресурса, поэтому Пр2 блокируется и R2.

8. И, наконец, пусть ПрЗ пытается получить R5; Y = 3, I=5, К=1, I'= 3, К'= 2, К' не равен Y, поэтому берем К=2, I'=2, К'=3. В этом случае К'= Y, т.е. тупик определен. Таблица PWTBL имеет следующий вид:

Процесс	Ресурс
1	3
2	2
3	5

Равенство Y=K’означает, что существует замкнутая цепь взаимоисключающих и ожидающих процессов, т. е. выполняются все четыре условия существования тупика.

Распознавание тупика требует дальнейшего восстановления. Восстановление модно интерпретировать как запрет постоянного пребывания в опасном состоянии. Существуют следующие методы восстановления:

- принудительный перезапуск системы, характеризующийся потерей информации о всех процессах, существовавших до перезапуска в системе;
- принудительное завершение только тех процессов, которые находились в тупике;
- принудительное последовательное завершение процессов, находящихся в тупике, с последующим вызовом алгоритма распознавания после каждого завершения процесса вплоть до исчезновения тупика;
- перезапуск процессов, находящихся в тупике, с некоторой контрольной точки, т.е. из состояния, предшествовавшего запросу на ресурс;
- перераспределение ресурсов с последующим последовательным перезапуском процессов, находящихся в тупике.

Основные издержки восстановления составляют потери времени на повторные вычисления, которые могут оказаться весьма существенными. К сожалению, в ряде случаев восстановление может стать невозможным: например, исходные данные, поступившие с каких-либо датчиков, могут уже измениться, а предыдущие значения будут безвозвратно потеряны.

6. Задачи ОС по управлению файлами и устройствами.

Организация параллельной работы устройств ввода-вывода и процессора. Каждое устройство ввода-вывода снабжено специализированным блоком управления – контроллером. Контроллер взаимодействует с драйвером – системным программным модулем, предназначенным для управления устройством ввода-вывода. Контроллер периодически принимает от драйвера данные, или выдает ему порции данных, а также принимает от драйвера управляющие команды, которые определяют действия с данными. Под управлением контроллера внешнее устройство может некоторое время выполнять свои операции автономно, не требуя внимания со стороны центрального процессора. Это время зависит от многих факторов – объёма передаваемой информации, степени интеллектуальности соответствующего контроллера, быстродействия устройства ввода-вывода и. т. д.

Процессы, происходящие в контроллерах, протекают между выдачами управляющих команд независимо от ОС. От подсистемы ввода-вывода требуется спланировать в реальном масштабе времени (в котором работают внешние устройства) запуск и приостановку большого числа разнообразных драйверов, обеспечив приемлемое время реакции каждого драйвера на независимые события, происходящие при работе внешних устройств и соответствующих драйверов. С другой стороны, необходимо минимизировать загрузку процессора задачами ввода-вывода, оставив как можно больше процессорного времени на выполнение пользовательских потоков. Данная

задача обычно решается на основе многоуровневой приоритетной схемы обслуживания по прерываниям. Для обеспечения приемлемого уровня реакции все драйверы(или части драйверов) распределяются по нескольким приоритетным уровням в соответствии с требованиями ко времени реакции и временем использования процессора. Для реализации такой схемы обычно используется **диспетчер прерываний**.

Согласование скоростей обмена и кэширование данных. При обмене данными всегда возникает задача согласования скорости.

В подсистеме ввода-вывода для согласования скоростей обмена широко используется буферизация данных в оперативной памяти. В системах, где обеспечение высокой скорости ввода-вывода является первоочередной задачей (управление в реальном времени, услуги сетевой файловой службы), большая часть оперативной памяти отводится не под коды прикладных программ, а под буферизацию данных. Однако разница между скоростью работы внутренних процессов и скоростью работы внешних устройств может оказаться весьма значительной, и оперативной памяти, отведённой под буфер, может просто не хватить. В этом случае в качестве буфера используется специальный дисковый файл – спул-файл. Например, с помощью спул файла осуществляется организация вывода данных на принтер (объём выводимой информации может превышать несколько десятков мегабайт). Также, в настоящее время, в качестве буфера активно используется собственная память внешних устройств (контроллеры графических станций), что значительно ускоряет вывод графики на экран мониторов.

Буферизация данных позволяет также сократить количество операций ввода-вывода, что повышает скорость ввода-вывода.

Разделение устройств и данных между процессами. Устройства ввода-вывода могут предоставляться процессам как в монопольное так и в совместное (разделяемое) использование. При этом, ОС должна обеспечивать контроль доступа к устройствам ввода-вывода. Такой контроль осуществляется путем проверки прав доступа пользователя или группы пользователей, от имени которых действует процесс, на выполнение той или иной операции над внешним устройством. Например, определённой группе пользователей разрешено захватывать последовательный порт в монопольное владение, а другой группе нет.

ОС может контролировать доступ не только к устройству в целом, но и к отдельным порциям данных, хранимым или отображаемым этим устройством. Диск является типичным примером устройства, для которого необходимо контролировать доступ не к устройству в целом, а к отдельным каталогам и файлам. При выводе на графический дисплей отдельные окна экрана также представляют собой ресурсы, к которым необходимо обеспечить тот или иной вид доступа со стороны выполняемых в системе процессов. Для каждого файла и каталога можно задать индивидуальные права доступа, а для их совместного использования необходимо задать режим разделения устройства в целом.

Одно и то же устройство в разные периоды времени работы вычислительной системы может использоваться как в разделяемом так и в монопольном режиме. ОС должна предоставлять эти устройства в обоих режимах, осуществляя отслеживание процедур захвата и освобождения монопольно используемых устройств, а в случае совместного использования оптимизируя последовательность операций ввода-вывода для различных

процессов в целях повышения общей производительности (пример – оптимизация затрат времени на перемещения головок диска).

При разделении устройства между процессами может возникнуть необходимость в разграничении порции данных двух процессов друг от друга. Обычно такая потребность возникает при использовании так называемых последовательных устройств, данные в которых не адресуются (пример - принтер). Для таких устройств система организует очередь заданий на вывод, причём, каждое задание представляет порцию данных, которую нельзя разрывать. Для хранения очереди заданий используется спул-файл, который одновременно согласует скорости работы принтера и оперативной памяти и позволяет разбить данные на логические порции. Так как спул-файл находится на разделяемом устройстве (диск) прямого доступа, то процессы могут параллельно выполнять ввод на принтер, помещая данные в свой раздел спул файла.

Обеспечение удобного логического интерфейса между устройствами и остальной частью системы. Разнообразие устройств ввода-вывода делают актуальной функцию по созданию унифицированного интерфейса между этими устройствами и приложениями. В качестве основы такого интерфейса выступает файловая модель устройств ввода-вывода, когда любое устройство рассматривается прикладным программистом как набор байт, с которым можно работать с помощью унифицированных системных вызовов (read, write и т.п.), задавая имя устройства и смещение от начала последовательности байт.

Привлекательность модели файла-устройства состоит в её простоте и унифицированности для устройств любого типа. Однако, при программировании операций ввода-вывода для некоторых устройств, такая модель является слишком “бедной”. Поэтому файловая модель используется только в качестве базиса, над которым подсистема ввода-вывода строит более содержательную модель (например – вывод графической информации).

Поддержка широкого спектра драйверов и простота включения нового драйвера в систему. Достоинством подсистемы ввода-вывода любой универсальной ОС является наличие разнообразного набора драйверов для наиболее популярных периферийных устройств.

Чтобы ОС не испытывала недостатка в драйверах, необходимо наличие чёткого, удобного и открытого интерфейса между драйверами и другими компонентами ОС. Открытость интерфейса драйверов, т. е. доступность их описания для независимых разработчиков ПО, является необходимым условием успешного развития ОС.

Драйвер, с одной стороны, взаимодействует с модулями ядра ОС, а с другой - с контроллерами внешних устройств. Поэтому существует два типа интерфейсов: <драйвер-ядро> и <драйвер-устройство>. Интерфейс <драйвер-ядро> должен быть стандартизирован в любом случае, а интерфейс <драйвер-устройство> имеет смысл стандартизировать тогда, когда подсистема ввода-вывода не разрешает драйверу непосредственно взаимодействовать в аппаратурой контроллера, а выполняет эти операции самостоятельно. В этом случае драйвер становится независимым от аппаратной платформы. Подсистема ввода-вывода может поддерживать несколько различных типов интерфейсов.

Динамическая загрузка и выгрузка драйверов. Современные ОС должны иметь в своём составе механизм, позволяющий осуществлять динамическую

загрузку и выгрузку драйверов. Такая возможность позволяет оперативно изменять набор необходимых драйверов в системе, а также экономить системную область памяти в случае выгрузки драйвера устройства, необходимость в котором отпала по тем или иным причинам.

При изменении состава драйверов, отсутствие такой возможности, требует повторной компиляции ядра ОС, что значительно усложняет модификацию ОС.

Поддержка нескольких файловых систем. Популярность файловой системы часто приводит к её миграции из "родной" ОС в другие операционные системы, – например, файловая система FAT появилась персонально в MS-DOS, но затем была реализована в OS/2, семействе MS Windows и многих реализациях UNIX. Ввиду этого поддержка нескольких популярных файловых систем со стороны подсистемы ввода-вывода также очень важна. Архитектура подсистемы ввода-вывода должна достаточно просто включать в её состав новые типы файловых систем, без необходимости изменения её кода. Для этого в ОС имеется специальный слой программного обеспечения, отвечающий за решение данной задачи.

Поддержка синхронных и асинхронных операций ввода-вывода. Операция ввода-вывода может выполняться по отношению к программному модулю, запросившему эту операцию, в синхронном или асинхронном режиме. Синхронный режим означает, что программный модуль приостанавливает свою работу до тех пор, пока операция ввода-вывода не будет завершена, а при асинхронном режиме программный модуль продолжает выполняться одновременно с операцией ввода-вывода. Операция ввода вывода может быть инициирована как пользовательским так системным процессом. Подсистема ввода-вывода должна предоставлять возможности выполнения как синхронных так и асинхронных операций ввода-вывода. Системные вызовы ввода-вывода обычно оформляются как синхронные процедуры, так как пользовательские процессы должны дождаться результатов выполнения операций ввода-вывода для своего дальнейшего корректного развития. Внутренние вызовы операций ввода-вывода из модулей ядра ОС обычно выполняются в виде асинхронных процедур, так как кодам ядра нужна свобода в выборе дальнейшего поведения после запроса ввода-вывода.

Список библиографических источников

1. *Гордеев А.В., Кучин Н.В.* Проектирование взаимодействующих процессов в операционных системах: Учеб. Пособие. Л.:ЛИАП,1991.72с.
2. *Гордеев А.В.* Операционные системы: Учебник.- СПб.:Питер, 2004. 416с.
3. *Гордеев А.В., Молчанов А.Ю.* Системное программное обеспечение: Учебник – СПб.:Питер,2002. 736с.
4. *Кейлингерт П.* Элементы операцинных систем. Введение для пользователей/Пер. с англ. Б. Л. Лисса и С. П. Тресковой. - М.:Мир, 1985. 295с.
5. *Олифер Н.А., Олифер В.Г.* Сетевые операционные системы: Учебник – СПб.: Питер, 2001. 544с.
6. *Соловьёв Г.Н., Никитин В.Д.* Операционные системы ЭВМ: Учеб. пособие – М.: Высшая школа, 1989. 255с.