**Московский государственный институт радиотехники, электроники и автоматики (Технический Университет)**

**Проект по курсу**

## **“Вычислительные комплексы, системы и сети ЭВМ”**

**Многопроцессорный вычислительный комплекс на основе коммутационной матрицы с симметричной обработкой заданий всеми процессорами**

**Группа: АВ-33-93**

**Студент: Липин В.В.**

### Москва

**1998**

# **1. Общая часть**

# **1.1 Содержание**

1. Общая часть
   1. Содержание
   2. Задание
   3. Введение
2. Аппаратная организация МПВК
   1. Структурная схема МПВК
   2. Функциональная схема элемента коммутационной матрицы
   3. Организация оперативной памяти
      1. Память с расслоением
      2. Применение кода Хэмминга в модулях памяти
   4. Организация резервирования и восстановления при отказе любого компонента МПВК
3. Организация функционирования ОС на МПВК
   1. Симметричная многопроцессорная обработка (SMP)
   2. Нити
      1. Подходы к организации нитей и управлению ими в разных вариантах ОС UNIX
   3. Семафоры
      1. Определение семафоров
      2. Реализация семафоров
   4. Тупиковые ситуации
   5. Предотвращение тупиковых ситуаций

# Линейное упорядочение ресурсов

## Иерархическое упорядочение ресурсов

## Алгоритм банкира

* 1. Защита информации

1. Литература

# **1.2 Задание**

## **(вариант №16)**

Разработать многопроцессорный вычислительный комплекс по следующим исходным данным:

* использовать матрицу с перекрестной коммутацией;
* количество процессоров – 8;
* количество блоков ОЗУ – 4;
* количество каналов ввода-вывода – 4;

Требуется разработать структурную схему коммутационной матрицы и функциональную схему элемента коммутационной матрицы.

Описать функционирование ОС для организации многопроцессорной обработки.

Описать организацию резервирования и восстановления вычислительного процесса при отказе любого компонента многопроцессорного вычислительного комплекса.

# **1.3 Введение**

Разработка многопроцессорных (МПВК) и многомашинных (ММВК) вычислительных комплексов, как правило, имеет свой целью повышение либо уровня надежности, либо уровня производительности комплекса до значений недоступных или труднореализуемых (реализуемых с большими экономическими затратами) в традиционных ЭВМ.

На большинстве классов решаемых задач для достижения высокой производительности наиболее эффективны МПВК. Это связано с большой интенсивностью информационных обменов между подзадачами, которая приводит к слишком высоким накладным расходам в ММВК. ММВК, в принципе, позволяют достичь много большей производительности благодаря лучшей масштабируемости, однако это преимущество проявляется только при соответствия решаемых задач условию максимального удлинения независимых ветвей программы, что не всегда возможно.

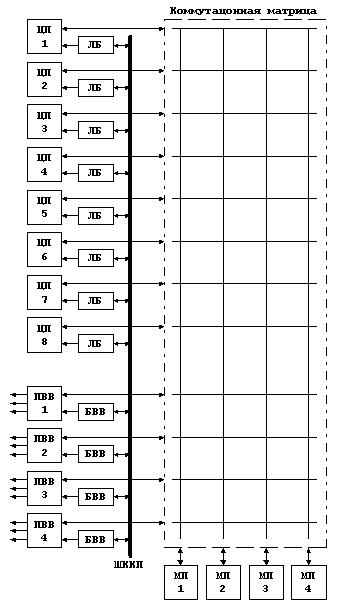
Указанный в задании МПВК с матрицей перекрестной коммутации позволяет достичь наибольшей производительности, что связано с минимизацией вероятности конфликтов при доступе к компонентам комплекса. При построении МПВК на основе доступа с использованием одной или нескольких общих шин конфликты доступа намного более вероятны, что приводит к заметному снижению производительности по сравнению с МПВК на основе матрицы перекрестной коммутации.

Исходя из этих соображений было решено проектировать МПВК по критерию максимальной производительности, меньше уделяя внимания высокой отказоустойчивости комплекса. Это решение также обосновывается и тем, что современные микроэлектронные изделия обладают вполне достаточной надежностью для подавляющего большинства коммерческих применений, что делает экономически необоснованным существенное усложнение комплекса с целью достижения высокой отказоустойчивости.

**2. Аппаратная организация МПВК**

2.1 Структурная схема МПВК

В МПВК с перекрестной коммутацией все связи осуществляются с помощью специального устройства – коммутационной матрицы. Коммутационная матрица позволяет связывать друг с другом любую пару устройств, причем таких пар может быть сколько угодно – связи не зависят друг от друга. Структурная схема МПВК приведена на рисунке:



Коммутационная матрица выполняет передачу данных между процессорами и памятью, а также между процессорами ввода-вывода и памятью. Коммутируются только внутренние шины МПВК, основное назначение которых – высокоскоростная передача данных, для этих шин нет смысла добиваться высокой протяженности проводников или стандартизации с целью упрощения подключения дополнительных устройств. Высокоскоростной обмен с периферийными устройствами осуществляется посредством процессоров ввода-вывода, которые являются контроллерами периферийных высокоскоростных шин, к которым, в свою очередь и подключаются контроллеры соответствующих устройств. На роль таких периферийных шин подходят, например, VME (применяется в МПВК фирмы Digital Equipment Company), SBus (применяется в МПВК фирмы Sun Microsystems) или PCI (применяется в МПВК, построенных на процессорах фирмы Intel семейства x86).

В SMP совместимой системе прерывания управляются контроллерами APIC (Advanced Programmable Interrupt Controller), БИС которых серийно выпускаются многими производителями микроэлектронных изделий (например DEC, Sun, IBM, Texas Instruments). Данные контроллеры обладают распределенной архитектурой, в которой функции управления прерываниями распределены между двумя функциональными блоками: локальным (ЛБ) и ввода-вывода (БВВ). Эти блоки обмениваются информацией через шину, называемую шиной коммуникаций контроллера прерываний (ШККП). Устройство ввода-вывода определяет появление прерывания, адресует его локальному блоку и посылает по шине ШККП. Блоки APIC совместно отвечают за доставку прерывания от источника прерываний до получателей по всей системе. Использование такой организации дополнительно увеличивает расширяемость за счет разгрузки разделения между процессорами нагрузки по обработке прерываний. Благодаря распределенной архитектуре, локальные блоки или блоки ввода-вывода могут быть реализованы в отдельной микросхеме или интегрированы с другими компонентами системы.

В МПВК подобной структуры нет конфликтов из-за связей, остаются только конфликты из-за ресурсов. Возможность одновременной связи нескольких пар устройств позволяет добиваться очень высокой производительности комплекса. Важно отметить и такое обстоятельство, как возможность установления связи между устройствами на любое, даже длительное время, так как это совершенно не мешает работе других устройств, зато позволяет передавать любые массивы информации с высокой скоростью, что также способствует повышению производительности комплекса.

Кроме того, к достоинствам структуры с перекрестной коммутацией можно отнести простоту и унифицированность интерфейсов всех устройств, а также возможность разрешения всех конфликтов в коммутационной матрице. Важно отметить и то, что нарушение какой-то связи приводит не к выходу из строя всего устройств, т.е. надежность таких комплексов достаточно высока. Однако и организация МПВК с перекрестной коммутацией не свободна от недостатков.

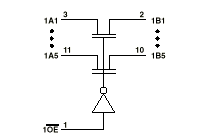
Прежде всего - сложность наращивания ВК. Если в коммутационной матрице заранее не предусмотреть большого числа входов, то введение дополнительных устройств в комплекс потребует установки новой коммутационной матрицы. Существенным недостатком является и то, что коммутационная матрица при большом числе устройств в комплексе становится сложной, громоздкой и достаточно дорогостоящей. Надо учесть то обстоятельство, что коммутационные матрицы строятся обычно на схемах, быстродействие которых существенно выше быстродействия схем и элементов основных устройств – только при этом условии реализуются все преимущества коммутационной матрицы. Это обстоятельство в значительной степени усложняет и удорожает комплексы.

# **2.2 Функциональная схема элемента коммутационной матрицы**

Коммутационная матрица (см. раздел “Структурная схема МПВК”) представляет собой прямоугольный двумерный массив из коммутационных элементов, установленных в местах пересечения шин процессоров и памяти (по структурной схеме МПВК). Функции каждого из этих элементов просты – при наличии управляющего сигнала должна быть обеспечена двусторонняя связь между шинами со стороны процессора и шинами со стороны памяти. При отсутствии управляющего сигнала связь должна отсутствовать, сигналы на шинах должны распространяться далее.

Не представляет существенных проблем синтез такого блока на стандартных логических элементах, однако каждый такой блок содержит два (как минимум) последовательно соединенных логических элемента, что вносит достаточно ощутимую задержку. Такое решение входит в противоречие с требованием повышенного быстродействия элементов коммутационной матрицы и приводит к необходимости повышения скорости работы за счет применения логики высокого быстродействия, что не всегда возможно или желательно.

Выходом представляется применение специализированных интегральных микросхем, выпускаемых некоторыми производителями микроэлектроники. У Texas Instruments это ИМС SN74CBT3384 (разрядность 10 бит), SN74CBT16244 (разрядность 16 бит), SN74CBT16210 (разрядность 20 бит), у Cypress Semiconductor - CYBUS3384 (два коммутатора в одном корпусе с разрядностью 5 бит каждый), у Sun Microelectronics - STP2230SOP. Данные ИМС имеют достаточное быстродействие (задержка распространения 5.2 – 10.0 нс, что соответствует максимальной тактовой частоте 190 – 100 МГц) и достаточно невысокую цену (несколько долларов за единицу в партиях от 1000 штук).

Все ИМС этого семейства имеют практически одинаковую функциональную схему:

Видно, что шина данных коммутируется полевым транзистором с изолированным затвором, на который подается управляющее напряжение со входа управления. Элемент полностью симметричен по входу и выходу данных, что делает его применение весьма удобным.

# **2.3 Организация оперативной памяти**

Память МПВК должна удовлетворять требованиям высокого быстродействия и высокой надежности. Несмотря на достаточно высокие характеристики по этим показателям, обеспечиваемые современной элементной базой, с помощью относительно несложных и недорогих методов можно достичь существенно более высоких показателей быстродействия и надежности.

С целью повышения быстродействия имеет смысл применить расслоение памяти по адресам на 4 модуля (разбиение на 4 модуля оговорено в задании, так что целесообразно применить данное разбиение для улучшения быстродействия МПВК). Более подробно расслоение по адресам будет рассмотрено ниже.

Для повышения надежности модулей памяти было решено применить корректирующие коды. Наиболее распространен код Хэмминга позволяющий исправлять одиночные и обнаруживать двойные ошибки. Более подробно его применение рассматривается ниже.

## **2.3.1 Память с расслоением**

Наличие в системе множества микросхем памяти позволяет использовать потенциальный параллелизм, заложенный в такой организации. Для этого микросхемы памяти часто объединяются в банки или модули, содержащие фиксированное число слов, причем только к одному из этих слов банка возможно обращение в каждый момент времени. Как уже отмечалось, в реальных системах имеющаяся скорость доступа к таким банкам памяти редко оказывается достаточной. Следовательно, чтобы получить большую скорость доступа, нужно осуществлять одновременный доступ ко многим банкам памяти. Одна из общих методик, используемых для этого, называется расслоением памяти. При расслоении банки памяти обычно упорядочиваются так, чтобы N последовательных адресов памяти i, i+1, i+2, . . . , i+N-1 приходились на N различных банков. В i-том банке памяти находятся только слова, адреса которых имеют вид k\*N + i (где 0 < k < M-1, а M число слов в одном банке). Можно достичь в N раз большей скорости доступа к памяти в целом, чем у отдельного ее банка, если обеспечить при каждом доступе обращение к данным в каждом из банков. Имеются разные способы реализации таких расслоенных структур. Большинство из них напоминают конвейеры, обеспечивающие рассылку адресов в различные банки и мультиплексирующие поступающие из банков данные. Таким образом, степень или коэффициент расслоения определяют распределение адресов по банкам памяти. Такие системы оптимизируют обращения по последовательным адресам памяти, что является характерным при подкачке информации в кэш-память при чтении, а также при записи, в случае использования кэш-памятью механизмов обратного копирования. Однако, если требуется доступ к непоследовательно расположенным словам памяти, производительность расслоенной памяти может значительно снижаться.

Обобщением идеи расслоения памяти является возможность реализации нескольких независимых обращений, когда несколько контроллеров памяти позволяют банкам памяти (или группам расслоенных банков памяти) работать независимо.

Если система памяти разработана для поддержки множества независимых запросов (как это имеет место при работе с кэш-памятью, при реализации многопроцессорной и векторной обработки), эффективность системы будет в значительной степени зависеть от частоты поступления независимых запросов к разным банкам. Обращения по последовательным адресам, или, в более общем случае, обращения по адресам, отличающимся на нечетное число, хорошо обрабатываются традиционными схемами расслоенной памяти. Проблемы возникают, если разница в адресах последовательных обращений четная. Одно из решений, используемое в больших компьютерах, заключается в том, чтобы статистически уменьшить вероятность подобных обращений путем значительного увеличения количества банков памяти. Например, в суперкомпьютере NEC SX/3 используются 128 банков памяти.

Подобные проблемы могут быть решены как программными, так и аппаратными средствами.

### 2.3.2 Применение кода Хэмминга в модулях памяти

С целью повышения общей надежности модулей оперативной памяти было решено применить хранение данных в коде Хэмминга, который за счет избыточности позволяет корректировать одиночные ошибки и обнаруживать ошибки большей кратности. Код Хэмминга получил широкое распространение благодаря простоте кодеров и декодеров, а также минимальной избыточности.

Так как большинство современных высокопроизводительных микропроцессоров имеют разрядность 64 бита, необходимо обеспечить разрядность памяти не менее 64 бит. Этой разрядности соответствует код Хэмминга (72, 64), что означает наличие 64 информационных разрядов и 8 контрольных. Можно рассчитать эффект от применения корректирующего кодирования:

Пусть вероятность отказа одиночного модуля памяти разрядностью 1 бит за некоторое время P1=10-5, тогда вероятность отказа одного из необходимых 64 модулей памяти за то же время Pобщ=64\*P1=6.4\*10-4. В случае применения кода Хэмминга для потери информации необходимы две ошибки в 72-х разрядах: Pобщ=(72\*P1)\*(71\*P1)= 5.112\*10-7. Таким образом надежность возрастает более чем на три порядка при увеличении стоимости на 12.5%. Разумеется эти рассуждения верны лишь в случае пренебрежимо малой вероятности отказов и стоимости кодера и декодера по сравнению с модулями памяти, однако в случае реальных объемов памяти это действительно так.

**2.4 Организация резервирования и восстановления при отказе любого компонента МПВК**

Одной из целей создания МПВК является достижение повышенной надежности вычислительной системы. Повышение надежности функционирования в МПВК достигается проще, чем в однопроцессорных ЭВМ, однако требует усложнения организации как аппаратуры МПВК, так и системного программного обеспечения, а для достижения наилучшего эффекта такой модификации нередко требует и прикладное программное обеспечение. При правильной организации МПВК отказы не приводят к потере данных или прекращению вычислительного процесса, а либо вообще не сказываются на работе МПВК, либо вызывают постепенную деградацию вычислительной мощности. Меры по обеспечению отказоустойчивости свои для каждого компонента МПВК.

Отказы оперативной памяти были рассмотрены выше. Отказы в коммутационной матрице также как и отказы оперативной памяти целесообразно маскировать применением кодов, корректирующих ошибки. Наиболее сложны для обнаружения и восстановления информации отказы процессоров, однако в случае наличия встроенных средств самоконтроля или при периодическом прогоне на каждом процессоре контрольных тестов, задача обнаружения решается довольно легко. Восстановление вычислительного процесса состоит в исключении операционной системой отказавшего процессора из списка доступных и запуске на другом процессоре пострадавшего процесса с последней контрольной точки. Отказы процессоров ввода-вывода обнаруживаются довольно легко, восстановление ввиду наличия 4-х идентичных по функциям ПВВ также несложно – необходимо лишь подключение критичных для работы МПВК периферийных устройств ко всем ПВВ с соответствующей коммутацией. Отказы самих периферийных устройств – наиболее простая и хорошо проработанная часть организации отказоустойчивости. Наиболее часто применяется тот или иной вид резервирования на уровне устройств – нагруженное, ненагруженное или скользящее.

Необходимо отметить, что поддержки операционной системы не требует в данном случае лишь организация отказоустойчивости оперативной памяти, коммутационной матрицы и некоторых периферийных устройств. Все остальные решения требуют той или иной поддержки на уровне ядра операционной системы МПВК.

# Также стоит подчеркнуть, что, несмотря на возможности организации отказоустойчивости МПВК, все наиболее серьезные реализации отказоустойчивых вычислительных систем представляют собой ММВК. Это объясняется большей изолированностью, а следовательно и к меньшему воздействию отказавших модулей на другие модули в ММВК.

# **3. Организация функционирования ОС на МПВК**

ОС МПВК должна выполнять все функции ОС однопроцессорной ЭВМ, однако дополнительно она должна обеспечивать эффективное использование ресурсов многопроцессорного комплекса (см. разделы “Симметричная многопроцессорная обработка”, “Нити”). Так как проблемы реализации ОС более серьезны, чем в однопроцессорном варианте, необходимо уделить особое внимание обеспечению корректного конкурентного доступа (предлагается использование семафоров – см. “Семафоры”). Для избежания простоев системы необходимо избегать тупиковых ситуаций (см. разделы “Тупиковые ситуации”, “Предотвращение тупиковых ситуаций”). Так как эффективных алгоритмов разрешения тупиков без потери информации, в общем случае, не существует, данные методы не рассматриваются (это в основном уничтожение процессов, начиная с наименее приоритетных процессов по какому-либо определенному заранее критерию).

В общем можно сказать, что ОС многопроцессорных ЭВМ похожи по функциям и реализации на ОС однопроцессорных ЭВМ, однако они выполняют некоторые дополнительные функции, а реализация основных функций часто более сложна.

**3.1 Симметричная многопроцессорная обработка  
(Symmetric Multiprocessor Processing – SMP)**

Для многопроцессорной обработки всегда требуются и соответствующие аппаратные платформы, и операционные системы. Однако ОС могут использовать многопроцессорные платформы несколькими разными способами. При асимметричной многопроцессорной обработке процессы прикладных программ назначаются конкретному процессору на аппаратной платформе. Нити каждого процесса должны ждать, пока назначенный им процессор освободится. Такой метод, как правило, менее эффективен, чем симметричный метод. Симметричная многопроцессорная обработка предполагает, что все процессоры имеют одинаковые возможности.

Под симметричной многопроцессорной обработкой обычно понимают равноправное разделение между процессорами различных задач, в числе которых в асимметричных системах могут быть ввод-вывод или числовые вычисления. В SMP системе все процессоры функционально идентичны, и любая задача может выполняться на любом процессоре. SMP ОС запускает процесс (нить) из общей очереди на первом освободившемся процессоре, как только выполнение процесса (нити) после приоритетного прерывания ОС возобновляется. В SMP-модели нагрузка динамически распределяется между процессорами, так что невозможна ситуация, в которой одни центральные процессоры перегружены, в то время как другие ничем не заняты.

Хотя все процессоры в SMP системе функционально идентичны, выделяется два их типа: загрузочный процессор (BSP) и прикладные процессоры (АР). Какой процессор играет роль загрузочного, определяется аппаратными средствами или совместно аппаратурой и BIOS. Это сделано для удобства и имеет значение только во время инициализации и выключения. BSP-процессор отвечает за инициализацию системы и за загрузку ОС. АР-процессор активизируется после загрузки ОС.

Симметричность имеет два важных аспекта: симметричность памяти и ввода-вывода. Память симметрична, если все процессоры совместно используют общее пространство памяти и имеют в этом пространстве доступ с одними и теми же адресами. Симметричность памяти предполагает, что все процессоры могут исполнять единственную копию ОС. В таком случае любые существующие системы и прикладные программы будут работать одинаково, независимо от числа установленных в системе процессоров. Требование симметричности ввода-вывода выполняется, если все процессоры имеют возможность доступа к одним и тем же подсистемам ввода-вывода (включая порты и контроллеры прерывания), причем любой процессор может получить прерывание от любого источника. Некоторые многопроцессорные системы, имеющие симметричный доступ к памяти, в то же время являются асимметричными по отношению к прерываниям устройств ввода-вывода, поскольку выделяют один процессор для обработки прерываний. Симметричность ввода-вывода помогает убрать потенциально узкие места ввода-вывода и тем самым повысить расширяемость системы.

Системы, обладающие симметричностью памяти и ввода-вывода, позволяют обеспечить расширяемость аппаратных средств, а также стандартизовать программные средства.

Есть две общие реализации SMP, известные как сильносвязанная и слабосвязанная. Сильносвязанная реализация базируется на схеме, согласно которой процессоры совместно используют данные из пула общих ресурсов, прежде всего, из общей памяти. Слабосвязанные системы используют механизм обмена сообщениями между процессами для совместного использования ресурсов, когда это необходимо. В некоторых слабосвязанных системах каждый процессор может даже иметь свой собственный контроллер диска и другие подсистемы.

Наиболее целесообразно использовать симметричную сильносвязанную модель. Тогда ОС обеспечивает мощную поддержку симметричной многопроцессорной обработки, так как планировщик в ядре ОС функционирует на уровне нити, поэтому сервер может назначить две нити одного процесса различным процессорам. Это особенно полезно для прикладных программ баз данных, где запросы могут быть расщеплены на нити и распределены между процессорами, что ведет к значительному увеличению производительности.

Чтобы полнее воспользоваться преимуществами SMP при организации многозадачности, выполнение нитей процесса контролируется с помощью приоритетных прерываний. Приоритетное прерывание позволяет операционной системе поддерживать контроль над программами, какую программу и когда запускать, так что сбившиеся программы не могут поработить систему и вызвать проблемы. При приоритетных прерываниях - постоянный, занимающие микросекунды запуск и остановка нескольких программ - как только возобновляется выполнение нити, ОС может назначить ее другому процессору.

Основным преимуществом такой архитектуры является то, что прикладные программы имеют в своем распоряжении столько центральных процессоров, сколько имеется в наличии у сервера. Так как операционная система занимается планированием работы процессоров, прикладным программам нет необходимости знать о количестве имеющихся процессоров. Операционная система назначит каждую нить первому свободному процессору. Планировщик позволяет распределять нагрузку и в конечном итоге выполнять программы точно с той же скоростью, с какой несколько центральных процессоров и могут с ними справиться.

Одним из преимуществ SMP-платформ, является их масштабируемость. Компания, реализовавшая однопроцессорную систему с таким сервером, в будущем будет иметь возможность реализовать SMP с той же самой версией ОС. Если задачи растут постепенно, необязательно выкладывать сразу все деньги. Напротив, можно приобрести сначала сервер только с одним процессором, а позднее добавлять дополнительные процессоры. Если масштабируемость реальна, то по мере того, как развивается информационная система и, соответственно, требования пользователей к мощности системы, можно добавить еще один процессор без какого-либо изменения программного обеспечения.

Для достижения масштабируемости важно использовать также асинхронные операции. При асинхронном вводе/выводе процессу не надо ожидать завершения чтения или записи, прежде чем он приступит к выполнению другой задачи. Каждый процесс создается с использованием единственной нити, которая выступает отдельным блоком при выполнении процессором команд программы. Программы могут запускать новые нити по мере потребности, и ОС назначает и контролирует их без участия высокоуровневой прикладной системы.

Прикладная программа может использовать несколько нитей, не нарушая основной поток программы. Программа может создать отдельную нить для каждой операции. Каждая из этих нитей, или операций, с точки зрения пользователя, может выполняться на отдельном процессоре без контроля со стороны прикладной программы.

Такая архитектура имеет еще одно преимущество. Накладные расходы связаны скорее с процессом, чем с нитью. Кроме того, нити создаются быстрее, чем процессы, и они совместно используют память внутри процесса, так что нитям нет необходимости осуществлять какую-либо специальную передачу данных. Помимо этого, все ресурсы процесса доступны всем нитям внутри процесса.

Конечно, для работы с текстовыми редакторами и электронными таблицами мощь многопроцессорной архитектуры не потребуется. Однако более актуальным она является в рамках сетевого сервера или сервера базы данных, ответственного за оперативную обработку транзакций, поступающих от большого числа пользователей.

При выборе прикладной программы необходимо выяснить, использует ли продукт все преимущества ОС на уровне нити, а не на уровне процесса. Определяя производительность и гибкость какой-либо системы как единого целого, необходимо иметь в виду, что аппаратное обеспечение, сетевая операционная система и прикладное программное обеспечение работают вместе.

Необходимо иметь в виду, что эффективность не растет линейно при добавлении еще одного процессора. Для любой SMP-системы выгода от дополнительного процессора постепенно сходит на нет с добавлением каждого последующего процессора.

Производительность не растет линейно, поскольку ОС должна управлять каждым процессором и, следовательно, взаимодействием процессора с внутренними вызовами и периферийными устройствами на шине. Когда нить в однопроцессорной системе не может более выполняться до осуществления некоторого условия, процессор маскирует программное прерывание так, что никакой другой процесс не может воспользоваться данным ресурсом. Затем он сохраняет состояние нити, чтобы выполнение кода могло возобновиться при осуществлении условия.

Когда есть только один процессор, довольно просто сохранять описание уровней прерывания и масок, контролирующих доступ к структурам данных ОС. С добавлением каждого нового процессора эта задача становится все более трудной. Операционная система для SMP-платформы должна уточнить, что только один процессор в данный момент выполняет сегмент кода, который меняет глобальную структуру данных.

В системе с одним процессором маскированное прерывание предотвращает использование процессором ресурса. Но в SMP-среде этот механизм не дает возможности гарантировать, что различные процессы не будут иметь доступа к тому же самому ресурсу через другое прерывание.

В SMP ОС целесообразно использовать метод взаимоблокировки для управления прерываниями между процессорами. По сути, взаимоблокировка является программной процедурой, которая блокирует доступ второго процессора к уже занятому ресурсу. Например, когда ядро хочет получить доступ к защищенной области, такой как очередь отложенных вызовов процедур, оно должно "приобрести" замок, который связан с очередью. Если замок находится в распоряжении какого-либо процессора, то другой процессор пытается получить замок до тех пор, пока его не освободит другой процессор.

Такой метод позволяет предотвратить порчу процессорами глобальных структур данных на уровне ядра. Однако при непродуманной реализации, он может привести к тому, что процессоры будут бездействовать в течение длительного периода, ожидая освободившийся замок. Этот метод хорошо работает, когда выполняются небольшие фрагменты кода. Такой код наиболее часто используется в функциях ядра, которые не вызывают внешние процедуры, не вытесняются из памяти и не генерируют прерываний. Таким образом, во многих случаях взаимоблокировка не действует, в то время как ядро должно синхронизировать нити между процессорами.

Ядро также может управлять нитью, назначая ей одно из трех состояний: готова, выполняется или ждет. Когда нить ждет результатов запроса, ядро изменяет состояние нити с "выполняется" на "ждет" и удаляет ее из очереди на выполнение. После того как нить получила ожидаемую ею информацию, ядро изменяет состояние нити с "ждет" на "готова" и возвращает ее в очередь. Нить выполняется, когда появляется возможность.

Хотя это объяснение поверхностно, оно все же показывает, насколько сложным для операционной системы оказывается управление синхронизацией нескольких процессов. По мере добавления новых процессоров к системе накладные расходы на управление конфликтами возрастают, и это уменьшает отдачу от ОС, ориентированных на симметрично-многопроцессорную обработку.

**3.2 Нити (threads)**

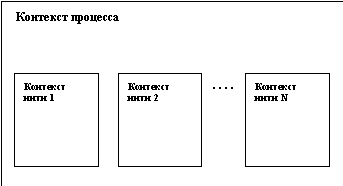
Понятие "легковесного процесса" (light-weight process), или, как принято называть его в современных вариантах ОС, "thread" (нить, поток управления) давно известно в области операционных систем. Интуитивно понятно, что концепции виртуальной памяти и потока команд, выполняющегося в этой виртуальной памяти, в принципе, ортогональны. Ни из чего не следует, что одной виртуальной памяти должен соответствовать один и только один поток управления. Поэтому, например, в ОС Multics допускалось (и являлось принятой практикой) иметь произвольное количество процессов, выполняемых в общей (разделяемой) виртуальной памяти.

Понятно, что если несколько процессов совместно пользуются некоторыми ресурсами, то при доступе к этим ресурсам они должны синхронизироваться (например, с использованием семафоров, см. раздел “Семафоры”). Многолетний опыт программирования с использованием явных примитивов синхронизации показал, что этот стиль "параллельного" программирования порождает серьезные проблемы при написании, отладке и сопровождении программ (наиболее трудно обнаруживаемые ошибки в программах обычно связаны с синхронизацией). Это явилось одной из причин того, что в традиционных вариантах ОС понятие процесса жестко связывалось с понятием отдельной и недоступной для других процессов виртуальной памяти. Каждый процесс был защищен ядром операционной системы от неконтролируемого вмешательства других процессов. Многие годы это считалось одним из основных достоинств системы (впрочем, это мнение существует и сегодня).

Однако связывание процесса с виртуальной памятью порождает, по крайней мере, две проблемы. Первая проблема связана с так называемыми системами реального времени. Такие системы, как правило, предназначены для одновременного управления несколькими внешними объектами и наиболее естественно представляются в виде совокупности "параллельно" (или "квазипараллельно") выполняемых потоков команд (т. е. взаимодействующих процессов). Однако если с каждым процессом связана отдельная виртуальная память, то смена контекста процессора (т. е. его переключение с выполнения одного процесса на выполнение другого процесса) является относительно дорогостоящей операцией. Поэтому традиционный подход препятствовал использованию системы в приложениях реального времени.

Второй (и может быть более существенной) проблемой явилось появление так называемых симметричных мультипроцессорных компьютерных архитектур (SMP - Symmetric Multiprocessor Processing). В таких компьютерах физически присутствуют несколько процессоров, которые имеют одинаковые по скорости возможности доступа к совместно используемой основной памяти. Появление подобных машин на мировом рынке, естественно, вывело на первый план проблему их эффективного использования. Понятно, что при применении традиционного подхода к организации процессов от наличия общей памяти не очень много толка (хотя при наличии возможностей разделяемой памяти об этом можно спорить). К моменту появления SMP выяснилось, что технология программирования все еще не может предложить эффективного и безопасного способа реального параллельного программирования. Поэтому пришлось вернуться к явному параллельному программированию с использованием параллельных процессов в общей виртуальной (а тем самым, и основной) памяти с явной синхронизацией.

Что же понимается под "нитью" (thread)? Это независимый поток управления, выполняемый в контексте некоторого процесса. Фактически, понятие контекста процесса изменяется следующим образом. Все, что не относится к потоку управления (виртуальная память, дескрипторы открытых файлов и т. д.), остается в общем контексте процесса. Вещи, которые характерны для потока управления (регистровый контекст, стеки разного уровня и т. д.), переходят из контекста процесса в контекст нити. Общая картина показана на рисунке:



Как видно из этого рисунка, все нити процесса выполняются в его контексте, но каждая нить имеет свой собственный контекст. Контекст нити, как и контекст процесса, состоит из пользовательской и ядерной составляющих. Пользовательская составляющая контекста нити включает индивидуальный стек нити. Поскольку нити одного процесса выполняются в общей виртуальной памяти, все нити процесса имеют равные права доступа к любым частям виртуальной памяти процесса, стек (сегмент стека) любой нити процесса в принципе не защищен от произвольного (например, по причине ошибки) доступа со стороны других нитей. Ядерная составляющая контекста нити включает ее регистровый контекст (в частности, содержимое регистра счетчика команд) и динамически создаваемые ядерные стеки.

Приведенное краткое обсуждение понятия нити кажется достаточным для того, чтобы понять, что внедрение в ОС механизма легковесных процессов требует существенных переделок ядра системы. (Всегда трудно внедрить в программу средства, для поддержки которых она не была изначально приспособлена.)

**3.2.1 Подходы к организации нитей и управлению ими в разных вариантах ОС UNIX**

Хотя концептуально реализации механизма нитей в разных современных вариантах практически эквивалентны (да и что особенное можно придумать по поводу легковесных процессов?), технически и, к сожалению, в отношении интерфейсов эти реализации различаются. Мы не ставим здесь перед собой цели описать в деталях какую-либо реализацию, однако постараемся в общих чертах охарактеризовать разные подходы.

Начнем с того, что разнообразие механизмов нитей в современных вариантах ОС UNIX само по себе представляет проблему. Сейчас достаточно трудно говорить о возможности мобильного параллельного программирования в среде UNIX-ориентированных операционных систем. Если программист хочет добиться предельной эффективности (а он должен этого хотеть, если для целей его проекта приобретен дорогостоящий мультипроцессор), то он вынужден использовать все уникальные возможности используемой им операционной системы.

Для всех очевидно, что сегодняшняя ситуация далека от идеальной. Однако, по-видимому, ее было невозможно избежать, поскольку поставщики симметричных мультипроцессорных архитектур должны были как можно раньше предоставить своим покупателям возможности эффективного программирования, и времени на согласование решений просто не было (любых поставщиков прежде всего интересует объем продаж, а проблемы будущего оставляются на будущее).

Применяемые в настоящее время подходы зависят от того, насколько внимательно разработчики ОС относились к проблемам реального времени. (Возвращаясь к введению этого раздела, еще раз отметим, что здесь мы имеем в виду "мягкое" реальное время, т. е. программно-аппаратные системы, которые обеспечивают быструю реакцию на внешние события, но время реакции не установлено абсолютно строго.) Типичная система реального времени состоит из общего монитора, который отслеживает общее состояние системы и реагирует на внешние и внутренние события, и совокупности обработчиков событий, которые, желательно параллельно, выполняют основные функции системы.

Понятно, что от возможностей реального распараллеливания функций обработчиков зависят общие временные показатели системы. Если, например, при проектировании системы замечено, что типичной картиной является "одновременное" поступление в систему N внешних событий, то желательно гарантировать наличие реальных N устройств обработки, на которых могут базироваться обработчики. На этих наблюдениях основан подход компании Sun Microsystems.

В системе Solaris (правильнее говорить SunOS 4.x, поскольку Solaris в терминологии Sun представляет собой не операционную систему, а расширенную операционную среду) принят следующий подход. При запуске любого процесса можно потребовать резервирования одного или нескольких процессоров мультипроцессорной системы. Это означает, что операционная система не предоставит никакому другому процессу возможности выполнения на зарезервированном(ых) процессоре(ах). Независимо от того, готова ли к выполнению хотя бы одна нить такого процесса, зарезервированные процессоры не будут использоваться ни для чего другого.

Далее, при образовании нити можно закрепить ее за одним или несколькими процессорами из числа зарезервированных. В частности, таким образом, в принципе можно привязать нить к некоторому фиксированному процессору. В общем случае некоторая совокупность потоков управления привязывается к некоторой совокупности процессоров так, чтобы среднее время реакции системы реального времени удовлетворяло внешним критериям. Очевидно, что это "ассемблерный" стиль программирования (слишком много перекладывается на пользователя), но зато он открывает широкие возможности перед разработчиками систем реального времени (которые, правда, после этого зависят не только от особенностей конкретной операционной системы, но и от конкретной конфигурации данной компьютерной установки). Подход Solaris преследует цели удовлетворить разработчиков систем "мягкого" (а, возможно, и "жесткого") реального времени, и поэтому фактически дает им в руки средства распределения критических вычислительных ресурсов.

В других подходах в большей степени преследуется цель равномерной балансировки загрузки мультипроцессора. В этом случае программисту не предоставляются средства явной привязки процессоров к процессам или нитям. Система допускает явное распараллеливание в пределах общей виртуальной памяти и "обещает", что по мере возможностей все процессоры вычислительной системы будут загружены равномерно. Этот подход обеспечивает наиболее эффективное использование общих вычислительных ресурсов мультипроцессора, но не гарантирует корректность выполнения систем реального времени (если не считать возможности установления специальных приоритетов реального времени).

Отметим существование еще одной аппаратно-программной проблемы, связанной с нитями (и не только с ними). Проблема связана с тем, что в существующих симметричных мультипроцессорах обычно каждый процессор обладает собственной сверхбыстродействующей буферной памятью (кэшем). Идея кэша, в общих чертах, состоит в том, чтобы обеспечить процессору очень быстрый (без необходимости выхода на шину доступа к общей оперативной памяти) доступ к наиболее актуальным данным. В частности, если программа выполняет запись в память, то это действие не обязательно сразу отображается в соответствующем элементе основной памяти; до поры до времени измененный элемент данных может содержаться только в локальном кэше того процессора, на котором выполняется программа. Конечно, это противоречит идее совместного использования виртуальной памяти нитями одного процесса (а также идее использования памяти, разделяемой между несколькими процессами).

Это очень сложная проблема, относящаяся к области проблем "когерентности кэшей". Теоретически имеется много подходов к ее решению (например, аппаратное распознавание необходимости выталкивания записи из кэша с синхронным объявлением недействительным содержания всех кэшей, включающих тот же элемент данных). Однако на практике такие сложные действия не применяются, и обычным приемом является отмена режима кэширования в том случае, когда на разных процессорах мультипроцессорной системы выполняются нити одного процесса или процессы, использующие разделяемую память.

После введения понятия нити трансформируется само понятие процесса. Теперь лучше (и правильнее) понимать процесс ОС UNIX как некоторый контекст, включающий виртуальную память и другие системные ресурсы (включая открытые файлы), в котором выполняется, по крайней мере, один поток управления (нить), обладающий своим собственным (более простым) контекстом. Теперь ядро знает о существовании этих двух уровней контекстов и способно сравнительно быстро изменять контекст нити (не изменяя общего контекста процесса) и так же, как и ранее, изменять контекст процесса. Последнее замечание относится к синхронизации выполнения нитей одного процесса (точнее было бы говорить о синхронизации доступа нитей к общим ресурсам процесса - виртуальной памяти, открытым файлам и т. д.). Конечно, можно пользоваться (сравнительно) традиционными средствами синхронизации (например, семафорами). Однако оказывается, что система может предоставить для синхронизации нитей одного процесса более дешевые средства (поскольку все нити работают в общем контексте процесса). Обычно эти средства относятся к классу средств взаимного исключения (т. е. к классу семафороподобных средств). К сожалению, и в этом отношении к настоящему времени отсутствует какая-либо стандартизация.

**3.3 Семафоры**

Поддержка операционной системы в многопроцессорной конфигурации может включать в себя разбиение ядра системы на критические участки, параллельное выполнение которых на нескольких процессорах не допускается. Нижеследующие рассуждения помогают понять суть данной особенности. При ближайшем рассмотрении сразу же возникают два вопроса: как использовать семафоры и где определить критические участки.

Если при выполнении критического участка программы процесс приостанавливается, для защиты участка от посягательств со стороны других процессов алгоритмы работы ядра однопроцессорной операционной системы используют блокировку.

Механизм установления блокировки:

/\* операция проверки \*/

выполнять пока (блокировка установлена)

{

приостановиться (до снятия блокировки);

};

установить блокировку;

Механизм снятия блокировки:

снять блокировку;

вывести из состояния приостанова все процессы, приостановленные в результате блокировки;

Блокировки такого рода охватывают некоторые критические участки, но не работают в многопроцессорных системах, что видно из приведенного рисунка:



Предположим, что блокировка снята, и что два процесса на разных процессорах одновременно пытаются проверить ее наличие и установить ее. В момент t они обнаруживают снятие блокировки, устанавливают ее вновь, вступают в критический участок и создают опасность нарушения целостности структур данных ядра. В условии одновременности имеется отклонение: механизм не сработает, если перед тем, как процесс выполняет операцию проверки, ни один другой процесс не выполнил операцию установления блокировки. Если, например, после обнаружения снятия блокировки процессор A обрабатывает прерывание и в этот момент процессор B выполняет проверку и устанавливает блокировку, по выходе из прерывания процессор A так же установит блокировку. Чтобы предотвратить возникновение подобной ситуации, нужно сделать так, чтобы процедура блокирования была неделимой: проверку наличия блокировки и ее установку следует объединить в одну операцию, чтобы в каждый момент времени с блокировкой имел дело только один процесс.

**3.3.1 Определение семафоров**

Семафор представляет собой обрабатываемый ядром целочисленный объект, для которого определены следующие элементарные (неделимые) операции:

* Инициализация семафора, в результате которой семафору присваивается неотрицательное значение;
* Операция типа P, уменьшающая значение семафора. Если значение семафора опускается ниже нулевой отметки, выполняющий операцию процесс приостанавливает свою работу;
* Операция типа V, увеличивающая значение семафора. Если значение семафора в результате операции становится больше или равно 0, один из процессов, приостановленных во время выполнения операции P, выходит из состояния приостанова;
* Условная операция типа P, сокращенно CP (conditional P), уменьшающая значение семафора и возвращающая логическое значение "истина" в том случае, когда значение семафора остается положительным. Если в результате операции значение семафора должно стать отрицательным или нулевым, никаких действий над ним не производится и операция возвращает логическое значение "ложь".

Определенные таким образом семафоры, безусловно, никак не связаны с семафорами пользовательского уровня.

**3.3.2 Реализация семафоров**

Дийкстра показал, что семафоры можно реализовать без использования специальных машинных инструкций. Здесь представлены реализующие семафоры функции, написанные на языке Си.

struct semaphore

{

int val[NUMPROCS]; /\* замок - 1 элемент на каждый процессор \*/

int lastid; /\* идентификатор процессора, получившего семафор последним \*/

};

int procid; /\* уникальный идентификатор процессора \*/

int lastid; /\* идентификатор процессора, получившего семафор последним \*/

Init(semaphore)

struct semaphore semaphore;

{

int i;

for (i = 0; i < NUMPROCS; i++)

semaphore.val[i] = 0;

}

Pprim(semaphore)

struct semaphore semaphore;

{

int i,first;

loop:

first = lastid;

semaphore.val[procid] = 1;

forloop:

for (i = first; i < NUMPROCS; i++)

{

if (i == procid)

{

semaphore.val[i] = 2;

for (i = 1; i < NUMPROCS; i++)

if (i != procid && semaphore.val[i] == 2)

goto loop;

lastid = procid;

return;

/\* успешное завершение, ресурс можно использовать \*/

}

else if (semaphore.val[i])

goto loop;

}

first = 1;

goto forloop;

}

Vprim(semaphore)

struct semaphore semaphore;

{

lastid = (procid + 1) % NUMPROCS; /\* на следующий процессор \*/

semaphore.val[procid] = 0;

}

Функция Pprim блокирует семафор по результатам проверки значений, содержащихся в массиве val; каждый процессор в системе управляет значением одного элемента массива. Прежде чем заблокировать семафор, процессор проверяет, не заблокирован ли уже семафор другими процессорами (соответствующие им элементы в массиве val тогда имеют значения, равные 2), а также не предпринимаются ли попытки в данный момент заблокировать семафор со стороны процессоров с более низким кодом идентификации (соответствующие им элементы имеют значения, равные 1). Если любое из условий выполняется, процессор переустанавливает значение своего элемента в 1 и повторяет попытку. Когда функция Pprim открывает внешний цикл, переменная цикла имеет значение, на единицу превышающее код идентификации того процессора, который использовал ресурс последним, тем самым гарантируется, что ни один из процессоров не может монопольно завладеть. Функция Vprim освобождает семафор и открывает для других процессоров возможность получения исключительного доступа к ресурсу путем очистки соответствующего текущему процессору элемента в массиве val и перенастройки значения lastid. Чтобы защитить ресурс, следует выполнить следующий набор команд:

Pprim(семафор);

команды использования ресурса;

Vprim(семафор);

В большинстве машин имеется набор элементарных (неделимых) инструкций, реализующих операцию блокирования более дешевыми средствами, ибо циклы, входящие в функцию Pprim, работают медленно и снижают производительность системы. Так, например, в машинах серии IBM 370 поддерживается инструкция compare and swap (сравнить и переставить), в машине AT&T 3B20 - инструкция read and clear (прочитать и очистить). При выполнении инструкции read and clear процессор считывает содержимое ячейки памяти, очищает ее (сбрасывает в 0) и по результатам сравнения первоначального содержимого с 0 устанавливает код завершения инструкции. Если ту же инструкцию над той же ячейкой параллельно выполняет еще один процессор, один из двух процессоров прочитает первоначальное содержимое, а другой - 0: неделимость операции гарантируется аппаратным путем. Таким образом, за счет использования данной инструкции функцию Pprim можно было бы реализовать менее сложными средствами:

struct semaphore

{

int lock;

};

Init(semaphore)

struct semaphore semaphore;

{

semaphore.lock = 1;

}

Pprim(semaphore)

struct semaphore semaphore;

{

while (read\_and\_clear(semaphore.lock));

}

Vprim(semaphore)

struct semaphore semaphore;

{

semaphore.lock = 1;

}

Процесс повторяет инструкцию read and clear в цикле до тех пор, пока не будет считано значение, отличное от нуля. Начальное значение компоненты семафора, связанной с блокировкой, должно быть равно 1. Как таковую, данную семафорную конструкцию нельзя реализовать в составе ядра операционной системы, поскольку работающий с ней процесс не выходит из цикла, пока не достигнет своей цели. Если семафор используется для блокирования структуры данных, процесс, обнаружив семафор заблокированным, приостанавливает свое выполнение, чтобы ядро имело возможность переключиться на контекст другого процесса и выполнить другую полезную работу. С помощью функций Pprim и Vprim можно реализовать более сложный набор семафорных операций, соответствующий тому составу, который определен в разделе “Определение семафоров”. Для начала дадим определение семафора как структуры, состоящей из поля блокировки (управляющего доступом к семафору), значения семафора и очереди процессов, приостановленных по семафору. Поле блокировки содержит информацию, открывающую во время выполнения операций типа P и V доступ к другим полям структуры только одному процессу. По завершении операции значение поля сбрасывается. Это значение определяет, разрешен ли процессу доступ к критическому участку, защищаемому семафором.

Алгоритм операции P:

/\*

Операция над семафором типа P

входная информация:

(1) семафор;

(2) приоритет;

выходная информация:

0 - в случае нормального завершения;

1 - в случае аварийного выхода из состояния приостанова по cигналу, принятому в режиме ядра;

\*/

{

Pprim(semaphore.lock);

уменьшить (semaphore.value);

если (semaphore.value >= 0)

{

Vprim(semaphore.lock); вернуть (0);

}

/\* следует перейти в состояние приостанова \*/

если (проверяются сигналы)

{

если (имеется сигнал, прерывающий нахождение в состоянии приостанова)

увеличить (semaphore.value);

если (сигнал принят в режиме ядра)

{

Vprim(semaphore.lock);

вернуть (-1);

}

в противном случае

{

Vprim(semaphore.lock);

longjmp;

}

}

}

поставить процесс в конец списка приостановленных по семафору;

Vprim(semaphore.lock);

выполнить переключение контекста;

проверить сигналы (см. выше);

вернуть (0);

}

В начале выполнения алгоритма операции P ядро с помощью функции Pprim предоставляет процессу право исключительного доступа к семафору и уменьшает значение семафора. Если семафор имеет неотрицательное значение, текущий процесс получает доступ к критическому участку. По завершении работы процесс сбрасывает блокировку семафора (с помощью функции Vprim), открывая доступ к семафору для других процессов, и возвращает признак успешного завершения. Если же в результате уменьшения значение семафора становится отрицательным, ядро приостанавливает выполнение процесса, используя алгоритм, подобный алгоритму sleep: основываясь на значении приоритета, ядро проверяет поступившие сигналы, включает текущий процесс в список приостановленных процессов, в котором последние представлены в порядке поступления, и выполняет переключение контекста.

Алгоритм V

/\*

Операция над семафором типа V

входная информация: адрес семафора

выходная информация: отсутствует

\*/

{

Pprim(semaphore.lock);

увеличить (semaphore.value);

если (semaphore.value <= 0)

{

удалить из списка процессов, приостановленных по семафору, первый по счету процесс;

перевести его в состояние готовности к запуску;

}

Vprim(semaphore.lock);

}

Операция V получает исключительный доступ к семафору через функцию Pprim и увеличивает значение семафора. Если очередь приостановленных по семафору процессов непустая, ядро выбирает из нее первый процесс и переводит его в состояние "готовности к запуску". Операции P и V по своему действию похожи на функции sleep и wakeup. Главное различие между ними состоит в том, что семафор является структурой данных, тогда как используемый функциями sleep и wakeup адрес представляет собой всего лишь число. Если начальное значение семафора - нулевое, при выполнении операции P над семафором процесс всегда приостанавливается, поэтому операция P может заменять функцию sleep. Операция V, тем не менее, выводит из состояния приостанова только один процесс, тогда как однопроцессорная функция wakeup возобновляет все процессы, приостановленные по адресу, связанному с событием. С точки зрения семантики использование функции wakeup означает: данное системное условие более не удовлетворяется, следовательно, все приостановленные по условию процессы должны выйти из состояния приостанова. Так, например, процессы, приостановленные в связи с занятостью буфера, не должны дальше пребывать в этом состоянии, если буфер больше не используется, поэтому они возобновляются ядром. Еще один пример: если несколько процессов выводят данные на терминал с помощью функции write, терминальный драйвер может перевести их в состояние приостанова в связи с невозможностью обработки больших объемов информации. Позже, когда драйвер будет готов к приему следующей порции данных, он возобновит все приостановленные им процессы. Использование операций P и V в тех случаях, когда устанавливающие блокировку процессы получают доступ к ресурсу поочередно, а все остальные процессы - в порядке поступления запросов, является более предпочтительным. В сравнении с однопроцессорной процедурой блокирования (sleep-lock) данная схема обычно выигрывает, так как если при наступлении события все процессы возобновляются, большинство из них может вновь наткнуться на блокировку и снова перейти в состояние приостанова. С другой стороны, в тех случаях, когда требуется вывести из состояния приостанова все процессы одновременно, использование операций P и V представляет известную сложность. Если операция возвращает значение семафора, является ли она эквивалентной функции wakeup?

while (value(semaphore) < 0)

{

V(semaphore);

};

Если вмешательства со стороны других процессов нет, ядро повторяет цикл до тех пор, пока значение семафора не станет больше или равно 0, ибо это означает, что в состоянии приостанова по семафору нет больше ни одного процесса. Тем не менее, нельзя исключить и такую возможность, что сразу после того, как процесс A при тестировании семафора на одноименном процессоре обнаружил нулевое значение семафора, процесс B на своем процессоре выполняет операцию P, уменьшая значение семафора до –1.

Процесс A продолжит свое выполнение, думая, что им возобновлены все приостановленные по семафору процессы. Таким образом, цикл выполнения операции не дает гарантии возобновления всех приостановленных процессов, поскольку он не является элементарным. Рассмотрим еще один феномен, связанный с использованием семафоров в однопроцессорной системе. Предположим, что два процесса, A и B, конкурируют за семафор. Процесс A обнаруживает, что семафор свободен и что процесс B приостановлен; значение семафора равно -1. Когда с помощью операции V процесс A освобождает семафор, он выводит тем самым процесс B из состояния приостанова и вновь делает значение семафора нулевым. Теперь предположим, что процесс A, по-прежнему выполняясь в режиме ядра, пытается снова заблокировать семафор. Производя операцию P, процесс приостановится, поскольку семафор имеет нулевое значение, несмотря на то, что ресурс пока свободен. Системе придется "раскошелиться" на дополнительное переключение контекста. С другой стороны, если бы блокировка была реализована на основе однопроцессорной схемы (sleep-lock), процесс A получил бы право на повторное использование ресурса, поскольку за это время ни один из процессов не смог бы заблокировать его. Для этого случая схема sleep-lock более подходит, чем схема с использованием семафоров. Когда блокируются сразу несколько семафоров, очередность блокирования должна исключать возникновение тупиковых ситуаций. В качестве примера рассмотрим два семафора, A и B, и два алгоритма, требующих одновременной блокировки семафоров. Если бы алгоритмы устанавливали блокировку на семафоры в обратном порядке, как следует из рисунка,



последовало бы возникновение тупиковой ситуации; процесс A на одноименном процессоре захватывает семафор SA, в то время как процесс B на своем процессоре захватывает семафор SB. Процесс A пытается захватить и семафор SB, но в результате операции P переходит в состояние приостанова, поскольку значение семафора SB не превышает 0. То же самое происходит с процессом B, когда последний пытается захватить семафор SA. Ни тот, ни другой процессы продолжаться уже не могут. Для предотвращения возникновения подобных ситуаций используются соответствующие алгоритмы обнаружения опасности взаимной блокировки, устанавливающие наличие опасной ситуации и ликвидирующие ее. Тем не менее, использование таких алгоритмов "утяжеляет" ядро. Поскольку число ситуаций, в которых процесс должен одновременно захватывать несколько семафоров, довольно ограничено, легче было бы реализовать алгоритмы, предупреждающие возникновение тупиковых ситуаций еще до того, как они будут иметь место. Если, к примеру, какой-то набор семафоров всегда блокируется в одном и том же порядке, тупиковая ситуация никогда не возникнет. Но в том случае, когда захвата семафоров в обратном порядке избежать не удается, операция CP предотвратит возникновение тупиковой ситуации:



если операция завершится неудачно, процесс B освободит свои ресурсы, дабы избежать взаимной блокировки, и позже запустит алгоритм на выполнение повторно, скорее всего, тогда, когда процесс A завершит работу с ресурсом. Чтобы предупредить одновременное обращение процессов к ресурсу, программа обработки прерываний, казалось бы, могла воспользоваться семафором, но из-за того, что она не может приостанавливать свою работу (см. главу 6), использовать операцию P в этой программе нельзя. Вместо этого можно использовать "циклическую блокировку" (spin lock) и не переходить в состояние приостанова, как в следующем примере:

while (! CP(semaphore));

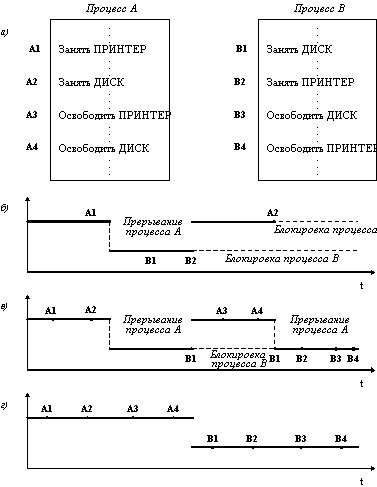
Операция повторяется в цикле до тех пор, пока значение семафора не превысит 0; программа обработки прерываний не приостанавливается и цикл завершается только тогда, когда значение семафора станет положительным, после чего это значение будет уменьшено операцией CP. Чтобы предотвратить ситуацию взаимной блокировки, ядру нужно запретить все прерывания, выполняющие "циклическую блокировку". Иначе выполнение процесса, захватившего семафор, будет прервано еще до того, как он сможет освободить семафор; если программа обработки прерываний попытается захватить этот семафор, используя "циклическую блокировку", ядро заблокирует само себя. В качестве примера обратимся к рисунку:



В момент возникновения прерывания значение семафора не превышает 0, поэтому результатом выполнения операции CP всегда будет "ложь". Проблема решается путем запрещения всех прерываний на то время, пока семафор захвачен процессом.

3.4 Тупиковые ситуации

Рассмотрим пример тупика. Пусть двум процессам, выполняющимся в режиме мультипрограммирования, для выполнения их работы нужно два ресурса, например, принтер и диск. На рисунке показаны фрагменты соответствующих программ.

Пусть после того, как процесс А занял принтер (установил блокирующую переменную), он был прерван. Управление получил процесс В, который сначала занял диск, но при выполнении следующей команды был заблокирован, так как принтер оказался уже занятым процессом А. Управление снова получил процесс А, который в соответствии со своей программой сделал попытку занять диск и был заблокирован: диск уже распределен процессу В. В таком положении процессы А и В могут находиться сколь угодно долго.

В зависимости от соотношения скоростей процессов, они могут либо совершенно независимо использовать разделяемые ресурсы (г), либо образовывать очереди к разделяемым ресурсам (в), либо взаимно блокировать друг друга (б). Тупиковые ситуации надо отличать от простых очередей, хотя и те и другие возникают при совместном использовании ресурсов и внешне выглядят похоже: процесс приостанавливается и ждет освобождения ресурса. Однако очередь - это нормальное явление, неотъемлемый признак высокого коэффициента использования ресурсов при случайном поступлении запросов. Она возникает тогда, когда ресурс недоступен в данный момент, но через некоторое время он освобождается, и процесс продолжает свое выполнение. Тупик же, что видно из его названия, является в некотором роде неразрешимой ситуацией.

В рассмотренных примерах тупик был образован двумя процессами, но взаимно блокировать друг друга могут и большее число процессов.

Проблема тупиков включает в себя следующие задачи:

* предотвращение тупиков,
* распознавание тупиков,
* восстановление системы после тупиков.

Тупики могут быть предотвращены на стадии написания программ, то есть программы должны быть написаны таким образом, чтобы тупик не мог возникнуть ни при каком соотношении взаимных скоростей процессов. Так, если бы в предыдущем примере процесс А и процесс В запрашивали ресурсы в одинаковой последовательности, то тупик был бы в принципе невозможен. Второй подход к предотвращению тупиков называется динамическим и заключается в использовании определенных правил при назначении ресурсов процессам, например, ресурсы могут выделяться в определенной последовательности, общей для всех процессов.

В некоторых случаях, когда тупиковая ситуация образована многими процессами, использующими много ресурсов, распознавание тупика является нетривиальной задачей. Существуют формальные, программно - реализованные методы распознавания тупиков, основанные на ведении таблиц распределения ресурсов и таблиц запросов к занятым ресурсам. Анализ этих таблиц позволяет обнаружить взаимные блокировки.

Если же тупиковая ситуация возникла, то не обязательно снимать с выполнения все заблокированные процессы. Можно снять только часть из них, при этом освобождаются ресурсы, ожидаемые остальными процессами, можно вернуть некоторые процессы в область свопинга, можно совершить "откат" некоторых процессов до так называемой контрольной точки, в которой запоминается вся информация, необходимая для восстановления выполнения программы с данного места. Контрольные точки расставляются в программе в местах, после которых возможно возникновение тупика. Из всего вышесказанного ясно, что использовать семафоры нужно очень осторожно, так как одна незначительная ошибка может привести к останову системы. Для того чтобы облегчить написание корректных программ, было предложено высокоуровневое средство синхронизации, называемое монитором. Монитор - это набор процедур, переменных и структур данных. Процессы могут вызывать процедуры монитора, но не имеют доступа к внутренним данным монитора. Мониторы имеют важное свойство, которое делает их полезными для достижения взаимного исключения: только один процесс может быть активным по отношению к монитору. Компилятор обрабатывает вызовы процедур монитора особым образом. Обычно, когда процесс вызывает процедуру монитора, то первые несколько инструкций этой процедуры проверяют, не активен ли какой-либо другой процесс по отношению к этому монитору. Если да, то вызывающий процесс приостанавливается, пока другой процесс не освободит монитор. Таким образом, исключение входа нескольких процессов в монитор реализуется не программистом, а компилятором, что делает ошибки менее вероятными.

В распределенных системах, состоящих из нескольких процессоров, каждый из которых имеет собственную оперативную память, семафоры и мониторы оказываются непригодными. В таких системах синхронизация может быть реализована только с помощью обмена сообщениями.

3.5 Предотвращение тупиковых ситуаций

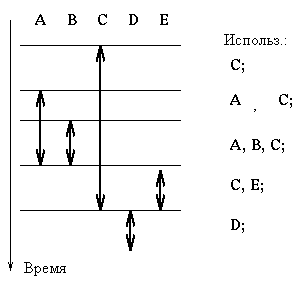
Для предотвращения тупика необходимо использовать ресурсы таким способом, при котором мы не можем войти в тупик. В реальной жизни аналог такого решения это “левые повороты слишком опасны, так что мы делаем только правые повороты”. Это требует больше времени, чтобы добраться до места назначения, но такой метод работает. В терминах тупиков, мы можем удерживать использование ресурсов так, чтобы не волноваться о возможности возникновения тупиков. Здесь мы рассмотрим эту идею на нескольких примерах.

# **3.5.1 Линейное упорядочение ресурсов**

Пусть все ресурсы полностью упорядочены от 1 до r. Мы можем наложить следующее ограничение: процесс не может запрашивать ресурс Rk, если он удерживает ресурс Rh и при этом k < h.

Просто видеть, что, используя это правило, мы никогда не будем входить в тупики. (Для доказательства применим метод доказательства от противного).

Приведем пример того, как применяется это правило. Пусть есть процесс, который использует ресурсы, упорядоченные как A, B, C, D, E, следующим способом:



Тогда процесс может делать следующее:

захватить (A); захватить (B); захватить (C);

использовать C;

использовать A, C;

использовать A, B, C;

освободить (A); освободить (B); захватить (E);

использовать C и E;

освободить (C); освободить (E); захватить (D);

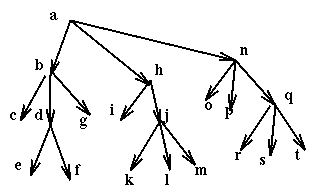
использовать D;

освободить (D);

Стратегия этого типа может использоваться, когда мы имеем несколько ресурсов. Эту стратегию просто применять, при этом степень параллелизма уменьшается не слишком сильно.

## **3.5.2 Иерархическое упорядочение ресурсов**

Другая стратегия, которую мы можем использовать в случае, если ресурсы иерархически структурированы, должна блокировать их в иерархическом порядке. Пусть удерживание ресурсов представлено организованным в дерево. Мы можем блокировать любой узел или группу узлов в дереве. Ресурсы, в которых мы заинтересованы - узлы в дереве, обычно самого нижнего уровня в древовидном представлении иерархии. Тогда следующее правило гарантирует предотвращение тупиков: узлы, в настоящее время блокированные процессом, должны найтись на всех путях от корня до желательных ресурсов. Пример использования этого правила, с блокировкой одиночного ресурса одновременно:



Тогда если процесс хочет использовать ресурсы e, f, i, k он должен использовать команды в следующей последовательности:

блокировка (a);

блокировка (b);

блокировка (h);

освобождение (a);

блокировка (d);

освобождение (b);

блокировка (i);

блокировка (j);

освобождение (h);

блокировка (k);

освобождение (j);

блокировка (e);

блокировка (f);

освобождение (d);

## **3.5.3 Алгоритм банкира**

Одна из причин, по которой этот алгоритм не используется в реальном мире широко – чтобы использовать его, операционная система должна знать максимальное количество ресурсов, в которых каждый процесс будет нуждаться когда-либо. Следовательно, например, запущенная на выполнение программа должна объявить, что она будет нуждаться не более чем, скажем, 400КБ памяти. Операционная система сохранит ограничение 400КБ и будет использовать его в вычислениях с целью предотвращения тупика.

Алгоритм Банкира пытается предотвращать тупик, путем предоставления или отказа предоставления ресурсов системы. Каждый раз, когда процесс нуждается в каком либо неразделяемом ресурсе, этот запрос должен быть одобрен банкиром.

Банкир - консервативен. Каждый раз, когда процесс делает запрос ресурса (“просит ссуду”), банкир осторожно рассматривает “банковские книги” и пытается определять, может или нет состояние тупика возникнуть в будущем, если запрос ссуды будет одобрен.

Алгоритм симулирует предоставление запрошенного ресурса и затем просматривает возникающее в результате выполнения запроса состояние системы.

После предоставления ресурса в системе останется некоторое количество этого ресурса свободным. Далее, проверяем другие процессы в системе. Мы требовали, чтобы каждый из них установил максимальное количество всех ресурсов системы, в которых они будут нуждаться, чтобы завершить выполнение, следовательно, мы знаем, сколько каждого ресурса каждый процесс удерживает и требует.

Если банкир имеет достаточно свободного ресурса чтобы гарантировать, что хотя бы один процесс может завершиться, тогда он может брать ресурс, удерживаемый этим процессом, и добавляет это к свободному объему ресурса. В этот момент банкир может рассматривать теперь больший свободный объем и делать попытку проверки, что другой процесс может завершиться, если требование будет выполнено. Если банкир может гарантировать, что все процессы в системе завершатся, то он одобряет рассматриваемый запрос.

Если, с другой стороны, в данный момент банкир не может гарантировать, что любые процессы завершатся, потому что недостаточно свободного ресурса удовлетворить самое малое требование, то может наступить состояние тупика. Это называется небезопасным состоянием. В этом случае рассматриваемый запрос будет отклонен и запрашивающий процесс обычно блокируется.

Эффективность алгоритма Банкира зависит значительно от его реализации. Например, если “банковские книги” сохраняются сортированными по размерам требований процессов, то добавление новой информации о процессах к таблице или уменьшение таблицы упрощено. Однако если таблица сохраняется в неупорядоченном виде, то добавление новой записи приводит к снижению эффективности таблицы.

# **3.6 Защита информации**

Защита информации подразделяется на несколько практически независимых направлений, два из которых представляются наиболее значимыми: поддержание целостности данных при многозадачной и многопроцессорной обработке, а также защита от несанкционированного доступа к данным. Без реализации защиты по этим двум направлениям круг задач, эффективно решаемых в данной вычислительной системе, резко сужается. Другие источники угрозы для обрабатываемых данных либо значительно менее существенны, либо имеют очень специфические методы решения, непригодные в общем случае, как, скажем, защита от ошибочных действий законного, но неквалифицированного пользователя.

Поддержание логической целостности данных при конкурентном доступе к ним должно осуществляться как на уровне ОС, так и на уровне прикладных программ. В самой ОС существует множество структур данных требующих такой защиты. Метод решения – это, в первую очередь, корректная разработка как системного, так и прикладного ПО. Наиболее простым и универсальным является использование блокировок с помощью семафоров (см. “Семафоры”). Для упрощения разработки прикладного ПО целесообразно вынести заботу о целостности данных на системное программное обеспечение, так, например большие массивы однородных данных имеет смысл хранить с использованием реляционных систем управления базами данных (РСУБД, RDBMS), при разработке которых уже учтены возможные проблемы поддержания логической целостности данных.

Стоит отметить, что данный аспект защиты данных, безусловно, имеет решения, однако осознание их необходимости в каждом конкретном случае и корректная реализация требуют достаточной квалификации разработчика, а также некоторого дополнительного расхода системных ресурсов.

Защита от несанкционированного доступа к данным в общем случае представляет собой более сложную задачу, решение которой может привести к противоречию с основными функциями вычислительной системы. Можно сказать, что удобство пользования вычислительной системой обратно пропорционально степени ее защищенности от несанкционированного доступа.

Основная модель защиты от несанкционированного доступа в настоящее время это доступ на основе определения прав групп и отдельных пользователей на использование ресурсов, каждый из которых имеет список управления доступом, который определяет какой вид доступа допустим для данного пользователя. По умолчанию доступ запрещен, при конфликте записей в списке выбирается наиболее ограничивающий вид доступа.

Наиболее важные требования к подсистеме безопасности таковы:

* Владелец ресурса должен иметь возможность управлять доступом к ресурсу.
* ОС должна защищать объекты от несанкционированного использования другими процессами. Например, система должна защищать память так, чтобы ее содержимое не могло читаться после освобождения процессом, и после удаления файла не допускать обращения к данным файла.
* Перед получением доступа к системе каждый пользователь должен идентифицировать себя, вводя уникальное имя входа в систему и пароль. Система должна быть способна использовать эту уникальную идентификацию для контроля действий пользователя.
* Администратор системы должен иметь возможность контроля связанных с безопасностью событий. Доступ к этим контрольным данным должен быть ограничен администратором.
* Система должна защищать себя от внешнего вмешательства типа модификации выполняющейся системы или хранимых на диске системных файлов.

Прежде чем пользователь сможет сделать что-либо в системе, он должен произвести вход в систему с указанием имени и пароля. В случае успешного входа в систему создается уникальный маркер доступа данного пользователя, содержащий информацию о его правах, используемую ОС для проверки запросов доступа к ресурсам. Данный маркер доступа копируется для всех порожденных пользователем процессов в системе, таким образом, любой процесс имеет полномочия, определенные ранее для владельца процесса.

Такое построение подсистемы безопасности в ОС позволяет выполнить все вышеперечисленные требования, однако существует возможность непосредственного доступа к данным путем электрического подключения к системе (например, к линии связи удаленного терминала) или приема электромагнитных излучений ЭВМ. Решение по предотвращению таких возможностей должно приниматься в каждом конкретном случае, однако наиболее общие методы это экранирование ЭВМ и линий связи, их физическая защита, а также шифрование информации при ее передаче и хранении (а возможно и при ее обработке).

# **4 Литература**

1. “Вычислительные комплексы, системы и сети”  
   А. М. Ларионов, С. А. Майоров, Г. И. Новиков  
   Ленинград, ЭНЕРГОАТОМИЗДАТ, 1987  
   (к разделу “Структурная схема МПВК”)
2. “The design of the UNIX operating system”  
   Maurice J. Bach  
   Prentice-Hall, 1986, ISBN 0-13-201757-1 025  
   (к разделу “Семафоры”)
3. “Сетевые операционные системы”  
   Н. А. Олифер, В. Г. Олифер,  
   Информационно-аналитические материалы Центра Информационных Технологий  
   (к разделу “Тупики”)
4. Курс лекций “Introduction to Distributed Systems and Networks” (CIS 307)  
   Giorgio Ingargiola, Associate Professor of Computer and Information Science Department.  
   Университет г. Темпль (шт. Филадельфия, США)  
   (к разделу “Предотвращение тупиков”)
5. “Операционная система UNIX”  
   С. Д. Кузнецов  
   Информационно-аналитические материалы Центра Информационных Технологий  
   (к разделу “Нити”)
6. “Сетевые ОС для SMP-платформ”  
   Е. Ленгрен  
   “Открытые Системы” № 2(10)/95 стр. 16  
   (к разделу “Симметричная многопроцессорная обработка”)
7. “Спецификация многопроцессорных систем компании Intel”  
   А.А. Мячев  
   “Открытые Системы” № 3(11)/95 стр. 56  
   (к разделу “Симметричная многопроцессорная обработка”)
8. “Ресурсы Windows NT Server и Workstation версия 3.5”  
   Microsoft Press, 1995  
   (к разделу “Защита информации”)