OS LEC 04

isagila

Собрано 08.04.2024 в 18:17

Содержание

1. Лекции

1.1. Лекция 24.02.07.

Эволюция понятия «операционная система»

Def 1.1.1. Операционная система это базовое системное программное обеспечение, управляющее работой вычислительного узла и являющееся посредником между аппаратным обеспечением, прикладным программным обеспечением и пользователем.

Принципы архитектуры фон Неймана:

1. Однородность памяти (и код, и данные находятся в единой памяти).
2. Адресность (линейная система адресов) и произвольный доступ к ячейкам памяти.
3. Принцип программного управления.
4. Принцип кодирования (для всего используется двоичное кодирование).

I. Программы—диспетчеры

Одной из задач, решаемых на этом этапе, была задача повторного использования кода, автоматизации загрузки и линковки. Было замечено, что некоторые участки кода часто повторяются, и чтобы не переписывать их заново, была предложена идея выделить в оперативной памяти некоторый участок, в который заранее положить необходимый код, а в основной программе лишь ссылаться на эти функции. В рамках этой идеи удобно использовать программы, которые подставят нужные адреса (слинкуют) заготовленные функции в основной программный код. Также необходимо неким образом обеспечить передачу параметров в эти функции и возможность взаимодействовать с их результатом. Отсюда и возникла потребность в программном обеспечении, которое будет автоматически решать эти задачи.
Следующая задача, которая возникла на этом этапе, это задача оптимизации взаимодействия с устройствами хранения и ввода-вывода. Дело в том, что не всегда все данные можно сохранить в оперативной памяти, а иногда это просто невыгодно или ненужно. Таким образом, нужно неким образом подгружать необходимые данные в оперативную память (из хранилища или с потокового ввода) и выгружать обработанные данные (в хранилище или в потоковый вывод). На тот момент это можно было делать только используя центральный процессор, т.к. в противном случае может возникнуть рассинхронизация: например, операция записи в оперативную память завершится позже, чем требуется, и попытка работать с этими не до конца загруженными данными приведет либо к сбою, либо к некорректной обработке этих данных.
Однако понятно, что это не самый оптимальный способ. Оптимальнее было бы сделать например так: процессор работает с одним блоком памяти, в то время как другой, независимый блок оперативной памяти подгружается из хранилища (или освобождается). Чтобы применить эту идею, введем дополнительно контроллер, который свяжем и с хранилищем, и с оперативной памятью. Разрешим этому контроллеру независимо от ЦП заниматься откачкой и подкачкой данных между хранилищем и оперативной памятью.
Но тогда возникает другая проблема: мы не можем прогнозировать время выполнения этих операций, т.к. большинство хранилищ используют технологии, которые это не позволяют. Чтобы решить эту проблему нужно связать (синхронизировать) работу контроллера и работу центрального процессора. Таким образом появляется механизм прерываний: сначала ЦП дает управляющую команду контроллеру на закачку определенного блока данных. По окончании этой операции контроллер инициирует прерывание, процессор приостанавливает выполнение текущей операции и обрабатывает это прерывание. В ходе обработки прерывания можно (например) изменить некоторый флаг, с помощью которого основная программа поймет о том, что данные готовы к обработке. Данная модель получила название spooling (обеспечение взаимодействия с периферийным устройством параллельно с работой основного вычислительного процесса).

Def 1.1.2. Прерывание это сигнал, поступающий от внешнего устройства к центральному процессору, сообщающий о наступлении некоторого события, в результате которого процессор приостанавливает выполнение текущего набора команд и передает управление подпрограмме — обработчику прерывания.

Последняя задача в рамках этапа программ—диспетчеров это появление однопрограммной пакетной обработки. Т.к. программы становились больше и иногда возникала потребность в том, чтобы переиспользовать некоторые специфичные части кода, то появилось разделение на пакеты. Теперь в память загружалась не вся программа (т.к. она не могла полностью разместиться в оперативной памяти), а лишь некоторое количество пакетов: пока один пакет выполнялся, другой пакет (или несколько пакетов) подгружались в память. Появляется понятие очереди из пакетов, и возникает необходимость в управлении этой очередью: например, некоторые пакеты по тем или иным причинам должны получить приоритет. Также нужно следить за тем, чтобы нужные пакеты всегда были подгружены и не происходило обращения к еще не загруженным частям кода. Эти, а также некоторые другие задачи возлагаются на программы—диспетчеры.
Итого, задачи решаемые на данном этапе:

1. Задача повторного использования кода, автоматизации загрузки и линковки.
2. Задача оптимизации взаимодействия с устройствами хранения и ввода-вывода.

3. Задача однопрограммной пакетной обработки.

II. Мультипрограммные операционные системы

Разные программы имеют разные требования к ЦП и к контроллеру: есть программы, которые используют много памяти, но мало процессорного времени, а есть программы, которые наоборот, требуют мало памяти, но сильно нагружают процессор. Однако вычислительный узел должен быть универсален и не должен подстраиваться лишь под одну задачу. Итого получается, что в зависимости от текущей программы либо процессор, либо контроллер будет простаивать. Это неэффективно, из-за чего появляется идея сделать так, чтобы пока одна программа использует ресурсы ЦП, другая могла бы использовать контроллер, чтобы погрузить необходимые ей данные в оперативную память. Таким образом ни ЦП, ни контроллер не будут простаивать впустую, а эффективность работы узла будет определяться тем, сколько программ одновременно смогут использовать необходимые им ресурсы.
Однако такой подход влечет за собой массу проблем: если раньше была единая очередь и порядок выполнения команд был очевиден, то теперь необходимо переключаться между программами, следить за тем, чтобы программы не использовали чужие участки памяти, организовывать работу с хранилищем, следить за уровнем доступа к ресурсам и так далее. Решением этих проблем является операционная система, которая занимает позицию между аппаратным обеспечением, программным обеспечением и пользователями.

1.2. Лекция 24.02.14.

Как уже было сказано на предыдущей лекции, для эффективного использования ресурсов потребовалось одновременно выполнять (или по крайней мере удерживать в памяти) несколько программ. Однако количество ядер меньше, чем количество процессов, которые необходимо исполнять (на тот момент речь шла вообще об процессорах с одним ядром, которые могут выполнять одновременно только одну программу). Соответственно, возникает задача об обеспечении разделения времени процессора. Таким образом, мы сначала даем одной программе (пакету) некоторое время использовать процессор, потом выполняем переключение (которое тоже занимает какое-то время) и далее предоставляем ресурс процессора другой программе. Время, которое мы предоставляем каждой программе, может быть одинаковым, а может и различаться — об этом мы поговорим позже.
Для реализации этих новых механизмов потребовались некоторые дополнительные вещи. В частности, потребовался таймер, который мог бы вызывать еще один вид прерываний (до этого существовали только прерывания ввода-вывода). Стоит отметить, что механизм прерываний по таймеру не так прост: если в результате обработки этого прерывания было принято решение переключиться на исполнение другого процесса, то необходимо неким образом сохранить текущее состояние регистров процессора, чтобы потом иметь возможность вернуться к этому процессу и продолжить его выполнять.
Однако ЦП это не единственный ресурс. Вторая серьезная задача это обеспечение разделения памяти, т.к. если несколько программ находятся в памяти одновременно, то необходимо каким-то образом обеспечить корректную адресацию, ведь на этапе разработки программного обеспечения нельзя знать, как и где оно будет размещено в памяти. Решением этой проблемы является виртуализация памяти — механизм виртуальной памяти. Каждая программа внутри себя отсчитывает адреса от некоторого виртуального нуля, а операционная система предоставляет механизм пересчета этих виртуальных адресов в физические, причем этот пересчет может осуществляться как в момент загрузки программы в оперативную память, так и при каждом обращении (или не при каждом — стратегия пересчета может быть и другой).
Задача обеспечения разделения памяти повлекла за собой задачу обеспечения защиты программ от деятельности других программ. Для решения этой задачи появляется еще одно прерывание, которое обеспечивает защиту памяти и прерывает работу процессора, если происходит попытка обратиться к памяти, принадлежащей другому приложению. Стоит отметить, что защищать от других программ нужно не только память, но и другие ресурсы.
Следующей задачей является планирование использования ресурсов и исполнения программ. Данная задача является более сложной, нежели планирование очереди пакетов. Помимо этого стоит учитывать, что планирование исполнения программы тесно связано с планированием ресурсов, которые необходимо выделить этой программе. Также нужно не забывать про синхронизацию: некоторые ресурсы являются неразделяемыми, и поэтому нельзя давать к ним доступ поочередно. Например, если две программы хотят что-то напечатать на принтере, то ЦП нельзя просто переключаться между ними и печатать по несколько символов от каждой программы.
Еще одной задачей на этапе мультипрограммных операционных систем является задача обеспечение универсального доступа к устройствам хранения. Для решения этой задачи появляется файлово-каталожная система и модель прав доступа к разным файлам и каталогам.
Итого, основные задачи данного этапа:

1. Обеспечение разделения времени процессора.
2. Обеспечение разделения памяти.
3. Защита программ от действий других программ.
4. Планирование выполнения и синхронизация выполнения.
5. Обеспечение универсального доступа к устройствам хранения.

Появляется механизм виртуальной машины. Каждое приложение работает как будто на своей виртуальной версии вычислительного узла и ничего не знает про другие приложения и про физические ресурсы вычислительного узла. Операционная система становится тем уровнем абстракции, который разделяет аппаратное обеспечение и программное обеспечение, позволяя им взаимодействовать только через нее, но не напрямую. Чтобы поддерживать эту абстракцию требуются некоторые строгие интерфейсы взаимодействия. Интерфейсом взаимодействия с аппаратным уровнем становится механизм привилегированного режима (только код ядра операционной системы имеет доступ к управлению физическими ресурсами вычислительного узла). В обратную стороны мы получаем механизм прерываний. С точки зрения программного обеспечения также появляется механизм взаимодействия — механизм системных вызовов (это «просьба» к ядру операционной системы о выполнении некоторой привилегированной операции или предоставлении некоторого аппаратного ресурса).
Одной из первых операционных систем считается операционная система МСР (1963 год).

III. Сетевые операционные системы

Узким местом становятся операции ввода-вывода. В связи с развитием качества связи появляется понятия удаленного терминала и многотерминальности. Теперь терминал совмещает в себе как функции ввода, так и функции вывода. Из-за удаленности терминала появляется проблема идентификации: если раньше четко можно было отследить, кто работает с вычислительным узлом (т.к. для этого необходимо было физически находится рядом), то теперь сделать это не так просто. Появляется потребность в дополнительных механизмах идентификации, аутентификации и авторизации. Помимо территориального разделения терминалов появляется идея о том, чтобы каким-то образом связать несколько вычислительных узлов. Эта идея возникла для того, чтобы разделять выполнение задач между несколькими вычислительными узлами и не позволять одному из узлов простаивать в то время как другой узел полностью загружен. Таким образом появляются сетевые операционные системы.

IV. Универсальные (мобильные открытые) операционные системы

На данном этапе, т.к. почти под каждый вычислительный узел нужна своя операционная система, то возникает плохая переносимость: ПО, разработанное для одной операционной системы, не всегда может работать на другой операционной системе. Таким образом от операционной системы требуется универсальность, а значит она должна поддерживать разработку приложений на языке высокого уровня, что позволит абстрагироваться от прямого доступа к ресурсам. Для того, чтобы это осуществить, необходимо, чтобы сама операционная система была написана на языке высокого уровня. Решение этой проблемы было найдено в 1969 году (Томсон, Керниган и Ритчи). Им стал язык `C` и операционная система `UNICS`. Первая редакция этой операционной системы пишется на ассемблере и не имеет встроенного компилятора языка высокого уровня. Одновременно с этим разрабатывается интерпретируемый язык `B`, и к 1972 году на этом языке переписывается `UNICS` (вторая редакция). Также разрабатывается компилируемый язык `C`, а код, написанный на `B`, постепенно переписывается на `C`. Итого к 1973 году появляется редакция `UNICS` с встроенным компилятором `C`. Далее, в конце 1973 года, появляется четвертая редакция, в которой ядро полностью написано на `C`, а к 1975-ому году и все утилиты также переписываются на `C` — это и становится пятой редакцией переименованной операционной системы `Unix`. Последняя редакция выходит в 1978 году. Далее уже появляются операционные системы называемые `*nix` системами в знак того, что они многое унаследовали от `Unix`.
Одной из систем на основе `Unix` является `BSD`, которая в некотором виде дожила и до наших дней (`FreeBSD`, `OpenBSD`, `NetBSD` и т.д.). На основе `BSD` была сделана операционная система `SunOS`, которая позже станет `Solaris`, а далее и `OpenSolaris`. Также на основе `Unix` появляются и проприетарные решения такие как `HP-UX`, `AIX`, `IRIX` и другие. Помимо этого стоит отметить `SystemV`, которая появилась как попытка связать `Unix`, развивающийся на тот момент `Solaris` и ветку `BSD` решений.
В 1983 году появляется проект `GNU`. Идея этого проекта заключается в том, чтобы создать свободное ПО и свободную операционную систему, на которой это ПО будет работать. Основатель `GNU` Ричард Столлман выделяет четыре свободы:

1. Свобода использовать программное обеспечение.
2. Свобода изучать и адаптировать программное обеспечение.
3. Свобода распространять программное обеспечение.
4. Свобода улучшать и публиковать программное обеспечение.

Из последнего пункта вытекает идея copyleft-а. Эта лицензия говорит о том, что если данное ПО интегрируется в другой проект или модифицируется, то этот проект также должен иметь copyleft лицензию. Итого все производные проекты и производные от них проекты также будут свободными (т.е. с copyleft лицензией).
Акроним `GNU` рекурсивно расшифровывается как Gnu is Not Unix. В рамках этого проекта создается компилятор `gcc`, пишутся и переписываются библиотеки языка `C`, однако остается проблема с ядром: написать с нуля ядро новой операционной системы оказывается сложной задачей. В 1991 публикуется операционная система `Linux`, которая несмотря на то, что была основана на операционной системе `Minix`, обладала собственным, концептуально новым ядром. Эндрю Таненбаум (автор `Minix`) высказывает резкую критику в сторону новой операционной системы, отмечая монолитность ядра и невозможность переноса на другие архитектуры помимо 8086. Для развития `Linux` все чаще начинает использоваться ПО, разработанное в рамках проекта `GNU`. Таким образом появляется операционная система, которая сейчас называется `GNU/Linux`.

Еще одним примером *nix системы является появившаяся в 1989 году операционная система NeXTSTEP. В 1997 году она интегрируется вместе с некоторыми наработками из FreeBSD в проект Darwin, который позже становится родоначальником операционных систем семейства macOS.

1.3. Лекция 24.02.21.

Цель существования современной операционной системы заключается в том, что она должна обеспечить производительность, надежность и безопасность выполнения пользовательских программ, эксплуатации аппаратного обеспечения, хранения и доступа к данным (в том числе по сети) и диалога с пользователем.
Т.к. операционная система это сложное и комплексное ПО, то чтобы лучше его понять, сначала надо поговорить об его архитектуре. Выделяют несколько уровней архитектуры программного обеспечения:

1. Функциональная архитектура.

   Этот уровень описывает всю совокупность функций, выполняемых операционной системой.

2. Системная архитектура.

   Реализация операционной системы это программно-аппаратный комплекс: есть аппаратные компоненты, которые являются неотъемлемой частью операционной системы, а есть программные компоненты, которые могут быть как свободными, так и проприетарными библиотеками.

3. Программная архитектура.

   Т.к. операционная система содержит некоторые программные компоненты, то нужно учитывать, что эти компоненты также обладают собственной структурой.

4. Архитектура данных.

   Предыдущий пункт говорит нам о сложности организации кода, а значит этот код работает с не менее сложноорганизованными данными, поэтому имеет смысл говорить об архитектуре данных.

Функциональная архитектура. Функции операционной системы

1. Управление разработкой и исполнением пользовательского ПО.
   (a) Предоставление возможности и API для написания программного обеспечения, совместимого с этой операционной системой.
   (b) Предоставление возможности загрузить и выполнить написанное приложение, а также обеспечить ему доступ к требуемым в процессе исполнения ресурсам.
   (c) Обнаружение и обработка ошибок, возникающих в ходе выполнения написанного ПО.
   (d) Высокоуровневый доступ к устройствам ввода-вывода.
   (e) Управление хранилищем данных: обеспечение высокоуровневого доступа к данным и их безопасности.
   (f) Мониторинг ресурсов.

2. Оптимизация использования ресурсов.

   Обозначим $k_1, k_2, \ldots$ — критерии оптимальности использования соответствующего ресурса. Т.к. в распоряжении вычислительного узла обычно находится не один, а множество ресурсов, то чаще всего оптимизировать все критерии одновременно невозможно. Из-за этого операционные системы используют разные принципы оптимизации, например:

   (a) Суперкритерий (свертка).

   Пусть $\widehat{k} = \alpha k_1 + \beta k_2 + \gamma k_3 + \ldots$ при условии, что $\alpha + \beta + \gamma + \ldots = 1$, т.е. используется взвешенная сумма критериев. Выбирается та стратегия, у которой суперкритерий $\widehat{k}$ максимален.

   (b) Условный критерий.

   В некоторых ситуациях значения какого-либо критерия (или нескольких критериев) обязательно должны находится в некотором диапазоне. Тогда сначала ищется «область» в которой выполнены требуемые условия к критериям, а потом уже оптимизируются остальные критерии.

   Стоит помнить, что операционная система это открытая система: пользователи открывают новые приложения, закрывают старые, запускают на обработку большие объемы данных или наоборот, бездействуют — в общем, помимо того, что требуется решать сложную задачу многокритериальной оптимизации, также нужно учитывать текущий контекст. Операционные системы обычно для решения этой проблемы используют ту или иную реализацию цикла Деминга (PDCA). Данный цикл состоит из четырех этапов:

   (a) Планирование. На этом этапе формируются некоторые значения коэффициентов $\alpha, \beta, \gamma, \ldots$.
   (b) Выполнение. На этом этапе операционная система принимает решения согласно выбранному плану.

(c) Проверка. На данном этапе происходит проверка сделанных решений на соответствие некоторым целевым показателям.

(d) Действия. На данном этапе нужно каким либо образом исправить несоответствие полученного результата плану.

Далее система уходит на новый цикл: на новом этапе планирования может быть выбрана другая стратегия, т.к. ситуация поменялась, или та же самая, если она хорошо себя зарекомендовала.

3. Поддержка администрирования и эксплуатации вычислительного узла.

   Операционная система должна предоставлять средства для диагностики, системного администрирования (восстановление после сбоев), восстановления поврежденных файлов (резервное копирование).

4. Поддержка развития самой операционной системы.

   Операционные системы, как чрезвычайно сложное ПО, проектируются и создаются в течение очень долго времени, а значит и использоваться будут также длительное время (процесс перехода на новую ОС связан с риском и определенными затратами). Это значит, что ОС должна быть открыта к изменениям: за время ее использования появятся новые программные и аппаратные решения, будет написано новое ПО, и ОС должна быть спроектирована так, чтобы у этих продуктов была возможность работать в рамках этой ОС (либо должна быть возможность добавить их поддержку). В современных ОС эта функция обычно реализуется с помощью средств автоматического обновления.

Функциональные подсистемы

1. Подсистема управления процессами.
   (a) Планировщики.
   (b) Структуры данных, отвечающие за хранение данных о процессах.
2. Подсистема управления памятью.
   (a) Механизм виртуализации памяти.
   (b) Защита памяти одного процесса от других процессов.
   (c) Распределение данных по памяти с разной скоростью доступа.
3. Подсистема управления файлами.
   (a) Преобразование символьных имен файлов в адреса их физического хранения.
   (b) Механизм управления каталогами.
4. Подсистема управления внешними устройствами.
5. Подсистема защиты данных и администрирования.
   (a) Идентификация, аутентификация, авторизация пользователя.
   (b) Аудит операционной системы, действий пользователя, поведения приложений, сетевой активности и т.д.
6. API.
7. Подсистема пользовательского интерфейса.
   (a) Интерфейс командной строки.
   (b) Графический пользовательский интерфейс (GUI).

1.4. Лекция 24.02.28.

Системная архитектура

Т.к. требования, предъявляемые к операционной системе, противоречивы и операционная система является открытой системой, то в области операционных систем нет «идеального» решения, которое бы удовлетворило всем требованиям — всегда приходится искать компромисс. Исходя из этого существуют разные архитектурные решения, имеющие разные преимущества и недостатки. Главным моментом, определяющим архитектуру ОС, является ответ на вопрос: что и как будет выполняться в ядре, а что будет вынесено за его пределы.

Def 1.4.1. Ядром операционной системы называется та часть ее кода, которая отличается двумя характеристиками присущими только ей (в совокупности) и никому больше: резидентность и привилегированный режим. Причем если резидентность может быть присуща другому ПО, то привилегированный режим это характеристики только ядра операционной системы.

Def 1.4.2. Резидентность ядра означает то, что его код находится в оперативной памяти всегда, в течении всего периода эксплуатации операционной системы, и как правило в неизменных адресах.

Def 1.4.3. Код, выполняемый в привилегированном режиме, не ограничивается проверками на доступ к адресам памяти.

Таким образом с одной стороны хочется, чтобы код ядра был как можно меньше: так он будет требовать меньше оперативной памяти, а также повысится надежность, ведь чем больше кода имеет доступ к привилегированному режиму, тем больше вероятность возникновения ошибки, а ошибка на таком уровне будет стоить очень дорого. С другой стороны встает вопрос безопасности: чем меньше будет ядро, тем больше функций придется выполнять вне привилегированного режима, а значит придется переключаться между режимами, чтобы обеспечить взаимодействие компонентов ОС. Если же бóльшая часть компонентов ОС помещена в ядро, то они могут взаимодействовать между собой напрямую, что увеличивает быстродействие системы. Помимо этого чем меньше код ядра, тем больше существует возможностей вмешаться в работу той части ОС, которая не защищена привилегированным режимом.
Помимо ядра существуют и другие принципы, которые влияют на архитектуру операционных систем:

1. Принцип модульной организации.

   Операционная система, как и любое сложное ПО, должна быть представлена в виде совокупности модулей с изолированной функциональностью.

2. Принцип функциональной избыточности.

   Операционная система должна обладать функционалом бóльшим, чем нужно каждому конкретному пользователю здесь и сейчас.

3. Принцип функциональной избирательности.

   Архитектура операционной системы должна позволять нам выбирать между функциями, предоставляемыми этой ОС. Причем должна быть возможность делать выбор на разных уровнях: например, какую-то функцию можно временно отключить, чтобы не расходовать на нее ресурсы, а какую-то функциональность можно получить, если поставить дополнительный пакет.

4. Принцип параметрической универсальности.

   Операционная система должна выносить во внешнюю среду как можно больше своих параметров управления.

5. Концепция многоуровневой иерархической вычислительной системы.

   Операционная система обычно делится на слои, и, обеспечив интерфейс взаимодействия между слоями, мы получаем возможность изменять один из слоев, не затрагивая остальные.

6. Принцип разделения модулей операционной системы.

   Модули операционной системы делятся на модули ядра и модули, относящиеся к вспомогательным функциям.

Архитектуры операционных систем

I.a Монолитная архитектура

Несмотря на название в монолитной архитектуре можно выделить три слоя (не всегда явно, но обычно это так): главная программа, сервисы и утилиты. Главная программа представлена одним модулем и умеет взаимодействовать с сервисами, а сервисы уже взаимодействуют с утилитами по принципу «многие-ко-многим», т.е. один сервис может взаимодействовать с несколькими утилитами и одна утилита может использоваться несколькими сервисами. Для чего же нужно такое разделение?
Утилиты обеспечивают нам работу с аппаратной частью, каждая их них реализует некоторый протокол взаимодействия с контроллером соответствующего экземпляра аппаратного обеспечения. Главная программа же является интерфейсом для взаимодействия с пользовательским ПО — ее задачей является получение системных вызовов. Механизм системных вызовов работает следующим образом: программа помещает в некоторой (но не произвольной) части выделенной ей памяти идентификатор системного вызова и необходимые аргументы и инициирует программное прерывание. Далее управление передается ядру операционной системы (в данном случае слою главной программы) и происходит анализ: какая программа инициировала системный вызов, какие аргументы были переданы и т.д. Из-за того, что утилиты работают с аппаратной частью, то существует еще слой сервисов, которые умеют принимать решения и исполнять их с помощью утилит. Главная программа после «расшифровки» системного вызова вызывает один или несколько сервисов, которые уже непосредственно занимаются его выполнением. Результат работы сервиса передается сначала главной программе, а потом и инициировавшему системный вызов приложению.
К преимуществам этой архитектуры относится быстродействие (сервисы могут вызывать утилиты без переключения режима), безопасность (все решения принимаются на уровне ядра). Из недостатков можно отметить проблемы с надежностью и повышенные затраты памяти.

I.b Многослойная архитектура

Со временем из-за большего количества функций, возложенных на операционную систему, количество сервисов довольно сильно увеличилось и появилась идея о разделении среднего слоя сервисов на несколько. Это концепция многослойной архитектуры. **Нельзя считать, что это новая отдельная архитектура** (т.к. даже в рамках монолитной архитектуры можно выделить слои): это скорее концепция, которая является логическим продолжением монолитной архитектуры и далее позволит нам перейти к другим видам архитектур. Причем стоит отметить, что многослойную архитектуру строили по-разному, и она менялась с течением времени. Далее будет рассмотрен лишь один из вариантов ее трактовки. Данную архитектуру удобно представить в виде концентрических окружностей. Тогда если смотреть от центра наружу, то порядок слоев будет такой:

1. Аппаратное обеспечение.
2. Средства аппаратной поддержки ядра.
   (a) Система прерываний.
   (b) Средство для поддержки привилегированного режима.
   (c) Средство поддержки виртуальной памяти.
   (d) Смена контекстовых регистров.
   (e) Системный таймер.
   (f) Защита памяти.
3. Машинно-зависимые модули (hardware abstraction layer, HAL).
4. Базовые механизмы ядра.

   Этот слой отвечает за исполнение решений, принятых менеджерами ресурсов.
5. Менеджеры ресурсов.

   На этом слое реализованы основные алгоритмы принятия решений: модули для составления расписаний (schedulers) и модули, занимающиеся задачами размещения (allocators).
6. Интерфейс системных вызовов (API).

Стоит отметить, что первые два внутренних слоя реализованы на аппаратном уровне. Слои 2 и 3 (их имеет смысл рассматривать в паре) обеспечивают возможность установки ОС на ту или иную платформу.

Замечание 1.4.4. Если все вышеперечисленные слои относятся к ядру, то такая архитектура все еще будет называться монолитной.

Еще одним серьезным недостатком монолитного ядра является то, что при любом изменении аппаратного обеспечения (и не только его) требуется перекомпиляция ядра. Даже когда появились решения, требующие не пересборки ядра, а лишь перезапуска ОС, это проблема все осталась актуальной: например, если ОС развернута на сервере, то ее перезапуск приведет к тому, что пользователи некоторое время не будут иметь доступа к серверу (что не всегда допустимо).
Развитием монолитного ядра стало модульное ядро. Его преимуществом является то, что зачастую (но не всегда) можно добавить, удалить или заменить модуль без перезапуска ядра.
Также недостатком монолитного ядра являются проблемы с созданием распределенных решений. Пусть есть несколько физических вычислительных узлов и требуется балансировать нагрузку между ними. В случае монолитной архитектуры у каждого узла будут свои менеджеры ресурсов и, следовательно, будет очень сложно этого достичь.

II. Микроядерная архитектура

Идея заключается в том, чтобы взять многослойную архитектуру и часть внешних слоев вынеси из режима ядра (kernel mode) в пользовательский режим (user mode). В режиме ядра остается слой базовых механизмов и более внутренние слои, т.е. слои, отвечающие за непосредственное исполнение решений, а вот слои, принимающие решения, переходят в пользовательский режим. Если в многослойной архитектуре решения принимали менеджеры ресурсов, то в микроядерной архитектуре это делают серверы (суть та же, но название другое). Теперь если приложению нужно совершить системный вызов, то оно сначала обращается в ядро, ядро отдает запрос соответствующему серверу (или нескольким), возвращаясь в пользовательский режим. После выполнения запроса сервер отвечает ядру, а оно передает этот ответ приложению, которое изначально инициировало системный вызов.
Основным преимуществом является то, что мы уменьшаем объемы памяти, выделяемые для операционной системы. Другим преимуществом является удобство в построении распределенных систем. К недостаткам можно отнести количество переключений между режимами, однако это не главная проблема. Основной проблемой является надежность и безопасность: т.к. серверы находятся в пользовательском режиме, то другое пользовательское ПО может помешать их работе или перехватить какие-либо данные.

Замечание 1.4.5. Гибридное ядро это то ядро, которое можно пересобрать так, что часть функций поменяет свое расположение (перейдет из режима ядра в пользовательский режим или наоборот).

III. Наноядерная архитектура

Идея наноядерной архитектуры заключается в том, чтобы взять микроядерную архитектуру и еще больше функций вынеси из ядра. Как правило при этой архитектуре в ядре остается только обработка прерываний, но иногда там дополнительно реализуются низкоуровневые планировщики с простым алгоритмом планирования. Наноядра в основном используют в гипервизорах для систем виртуализации.

IV. Экзоядерная архитектура

Данная архитектура является развитием идеи распределенной ОС и попыткой построить гетерогенную (т.е. реализованную над разнородным оборудованием) распределенную ОС. В данной архитектуре принятие решений и межпроцессное взаимодействие остаются в режиме ядра, а взаимодействие с оборудованием разрешается напрямую.

1.5. Лекция 24.03.06.

Управление процессами

Def 1.5.1. Процесс это совокупность набора исполняющихся команд, ассоциированных с ним ресурсов и контекста исполнения, находящихся под управлением операционной системы.

Процесс для операционной системы представлен в виде некоторой структуры данных (process control block (PCB), дескриптор процесса). Стоит отметить, что процесс это не то же самое, что и программа: одна программа может порождать несколько процессов. Это весьма логичный и понятный вариант: допустим в некотором комплексном ПО один процесс занимается обработкой GUI, а другой взаимодействием по сети. Однако обратная ситуация также возможна: несколько программ могут исполняться в рамках одного процесса. Допустим, мы запустили некоторое приложение, и оно выполнило системный вызов. Тогда часть времени в рамках одного процесса выполнялся непосредственно код приложения, а часть времени — код ядра. Т.к. ядро это отдельная программа (даже скорее несколько программ), то получается, что в рамках одного процесса выполнялось несколько программ.
Однако одного понятия процесса недостаточно для эффективного манипулирования ресурсами. Предположим, что у нас есть большая растровая картинка, которую надо каким-либо образом обработать, причем обработка каждого конкретного пикселя может происходить независимо от обработки других пикселей. В этом случае эффективно обрабатывать данную картинку параллельно и по частям, однако концепция процессов мешает этому: картинка должна быть помещена в адресное пространство только одного процесса, а другие процессы не должны иметь к ней доступ. Для решения этой проблемы появилась следующая идея: пусть внутри одного процесса будет разрешено создавать несколько наборов команд и связанных с ними контекстов.

Def 1.5.2. Пара из набора команд и связанного с ним контекста в рамках одного процесса называется потоком (thread).

Итого в рамках одного процесса может существовать несколько потоков, причем все потоки имеют доступ в общему адресному пространству процесса. Стоит отметить, что в современных ОС единицей диспетчеризации является именно поток, а не процесс. Однако в связи в этим возникают некоторые проблемы, а именно: переключение между потоками возможно только через переключение в режим ядра, т.к. диспетчеризацией потоков занимается ОС (и делает она это в режиме ядра). Причем т.к. за переключение потоков отвечает ОС, то она может делать это не так, как задумано разработчиком ПО. Это в свою очередь может негативно сказаться на производительности.
Для решения этой проблемы появился еще один уровень иерархии (ниже потоков), который называется волокно (fiber) и предоставляет пользовательскую многопоточность вне инструментария ОС. Теперь любой поток представлен в виде множества волокон.

Замечание 1.5.3. Существуют разночтения в термине fiber. Помимо «волокна» он иногда трактуется как «легковесный поток», а иногда как «green thread» (зеленый поток).

Как и в случае потока, каждое волокно содержит в себе некоторый набор команд и контекст их выполнения, но разница в том, что управление переключением между волокнами (и его планирование) берет на себя код этого потока, а не ОС. Плюсами такого подхода является решение проблем, описанных выше, а вот к минусам можно отнести то, что приходится самостоятельно реализовывать алгоритмы планирования и переключения между волокнами.
У операционной системы есть механизм прерываний по таймеру, который позволит переключать потоки, а у волокон нет такого механизма. Основным решением этой проблемы стало решение кооперативной многозадачности: его идея заключается в том, что само волокно в какой-то момент отдаст управление следующему волокну, либо волокну, являющемуся диспетчером. Также стоит отметить, что у волокон может быть поддержка на уровне ОС: не с точки зрения планирования, а точки зрения предоставления API для разработки приложений с использованием этого механизма.
Однако даже этих трех уровней иерархии оказалось недостаточно, чтобы эффективно управлять процессами в операционной системе. Представим себе работу браузера: если для каждой вкладки сделать отдельный поток, то возникнет проблема с безопасностью. Т.к. потоки в рамках процесса браузера имеют общее адресное пространство, то одна вкладка будет иметь возможность получить доступ к данным другой вкладки (причем это может быть как преднамеренно, так и вследствие некоторой ошибки). Отсюда возникает другая идея: выстроим иерархию процессов,

где будет корневой процесс, а все вкладки будут им порождены. Но тогда возникает следующая проблема: пусть мы открыли какое-то одно приложение и 99 вкладок в браузере, тогда с точки зрения операционной системы 99% процессов это процессы браузера и лишь 1% это процессы другого приложения. Однако планировщик учитывает их всех на одном уровне.
Отсюда возникает идея о том, что нужно научиться ограничивать доступ к ресурсам для некоторых групп процессов. Для этого в иерархии над процессами появляется еще один уровень. В разных операционных системах он называется по-разному: в Windows это job (задание, работа), в Linux это cgroup (контрольная группа). Суть заключается в том, что в рамках этой группы процессов можно установить определенные квоты на использование тех или иных ресурсов.

Основные функции подсистемы управления процессами

1. Создание.

   В операционной системе любой процесс порождается другим процессом, он не создается абстрактно извне. Таким образом любой процесс имеет родительский процесс. Как именно появляется первый процесс мы не будем рассматривать в рамках этого курса, лишь скажем что какое-то решение есть, и он как-то появляется. Отсюда получается, что в операционной системе есть некоторая иерархия процессов (и в разных ОС она будет отличаться), где каждый процесс представлен некоторой структурой данных. В разных ОС она будет разной, но в общем виде эта структура содержит следующие поля:

   (a) Информация по идентификации процесса. Сюда входят PID (уникальный идентификатор процесса), PPID (идентификатор родительского процесса), UID (идентификатор пользователя, запустившего процесс).

   (b) Информация по состоянию процесса (статус и контекст).

   (c) История (она сильно зависит не только от типа ОС, но и от ее версии и планировщика).

   Как же создается процесс? Сначала рассмотрим пользовательские процессы в Linux (процессы ядра рассматривать не будем). Они образуют дерево процессов, где корнем является процесс с PID = 1 и PPID = 0. Порождение новых процессов происходит методом клонирования (fork), т.е. полным копированием адресного пространства. PID для нового процесса выдается ОС, PPID определяется как PID процесса, который клонировали, а другие свойства (например UID) наследуются от родительского процесса. Таким образом ни один дочерний процесс не будет иметь больше прав, чем его родитель, что выгодно с точки зрения безопасности.

   После клонирования сегмент кода склонированного процесса заменяется на код необходимого приложения, и мы получаем полноценный новый процесс. После завершения работы дочерний процесс «отчитывается» родителю об этом, и родитель (с помощью системного вызова) может считать и обработать код его завершения. Если же родительский процесс внезапно нештатно завершается (а его дочерние процессы продолжают работать), то новым родителем для «осиротевших» процессов становится корневой процесс, т.к. дерево процессов должно оставаться связным. Есть так же и другая интересная ситуация: если родительский продолжает работать, но не может корректно обработать код завершения дочернего процесса, то такой дочерний процесс становится «зомби»-процессом. Он вроде бы и завершил свою работу, но все равно остается в дереве процессов. Подробнее об этом будет рассказано в следующей лекции.

   В Windows работает другой механизм. Корневым процессом является диспетчер процессов, который несет ответственность за создание всех новых процессов, т.е. если какому-либо из процессов потребовалось создать потомка, то он через системный вызов обращается к диспетчеру процессов и просит создать новый процесс. Таким образом нет никакого дерева процессов: все процессы как бы являются потомками диспетчера процессов. Плюсом такого решения является централизованный тотальный контроль за появлением процессов. С другой стороны права дочернего процесса определяются не родительским процессом, а диспетчером процессов, поэтому формально можно создать процесс с правами выше, чем у родительского процесса. Для предотвращения этого у ОС есть механизмы защиты, но потенциальная возможность их обойти все равно остается.

2. Обеспечение ресурсами.

   Любой процесс с самого начала уже обеспечен некоторыми ресурсами: например, ему выделено адресное пространство. Такие ресурсы называются статическими и будут оставаться с процессом до его завершения. Однако есть еще и динамические ресурсы: например, в процессе своего выполнения процесс может потребовать дополнительную память или доступ к файлу. Даже процессорное время тоже можно считать динамическим ресурсом. Итого задача обеспечения ресурсами сводится к задачам планирования ресурсов (задача о расписании) и аллокации (задача о рюкзаке).

3. Изоляция.

   В последующих лекциях про память будет обсуждаться вопрос об обеспечении изоляции памяти. Также в последующих лекциях при обсуждении синхронизации частично будет затронута тема изоляции.

4. Планирование.

   Планированию далее будет посвящена отдельная лекция, т.к. это весьма многоуровневый процесс: необходимо планировать процессорное время, очереди на ввод-вывод, рождаемость процессов и т.д.

5. Диспетчеризация (переключение процесса между различными состояниями).

   Простейшая диспетчеризация заключается в том, что процесс переключается между состояниями «работает» и «ждет». Смена состояний происходит в три шага:

   (a) Сохранение контекста текущего процесса.

   (b) Загрузка контекст другого процесса.

   (c) Смена состояний этих двух процессов.

   Стоит отметить, что последний шаг должен быть атомарным, в противном случае мы получаем неуправляемое состояние системы: если произойдет прерывание, то у нас одновременно будет существовать либо два работающих процесса, либо вообще ни одного. Для удовлетворения этого требования существуют разные хитрые механизмы, которые будут рассмотрены позднее. Понятно, что у процесса может быть больше двух состояний: об этих состояниях и о переходах между ними мы поговорим на следующей лекции.

6. Взаимодействие.

   В рамках этого курса будем касаться этой темы достаточно слабо, т.к. взаимодействие процессов существенно отличается в разных операционных системах.

7. Синхронизация.

   Этой теме также будет посвящена отдельная лекция (даже чуть больше), в рамках которой мы поговорим про семафоры, мьютексы, тупики и т.п.

8. Уничтожение.

   Уничтожение процесса это не такой простой процесс, как кажется на первый взгляд: процесс мог владеть определенными ресурсами, которые возможно нужно корректно «завершить». Например, процесс что-то записал в файл, но т.к. современные файловые системы чаще всего используют механизм отложенной записи, то ОС нужно принудительно инициировать запись на диск, ведь после уничтожения процесса данные, которые он хотел записать, станут недоступны. Также может быть такое, что процесс владел некоторыми неразделяемыми ресурсами и блокировал доступ к этим ресурсам для других процессов — после уничтожения эти блокировки нужно снять.

   Помимо этого необходимо завершить все дочерние процессы: проверить, что они корректно завершились, и в противном случае каким-либо образом это обработать. Еще нужно отчитаться родительскому процессу о своем завершении и предоставить код завершения.

1.6. Лекция 24.03.13.

Диспетчеризация процессов

Под диспетчеризацией процессов подразумевается переключение процессов между различными состояниями. Мы будем рассматривать возможные состояния и переходы между ними в виде графа (или конечного автомата), где вершинами будут являться состояния процесса, а ребрами — возможные переходы между ними. Сначала посмотрим на самую простую модель (рис. 1.6.1).

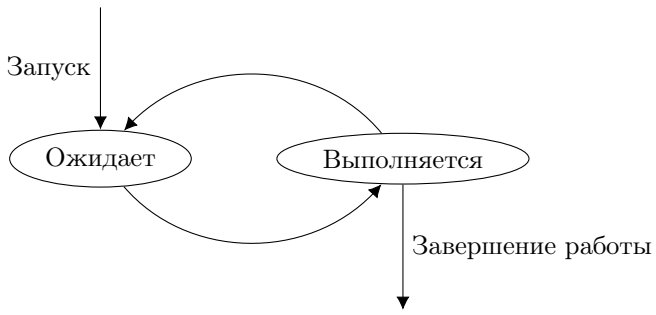

Рис. 1.6.1: Простейший граф состояний процесса

После создания процесс попадает в состояние ожидания, потом, когда до него доходит очередь, он начинает выполняться. Далее возможно два варианта: процесс либо выполнится и завершится, либо снова уйдет в ожидание, причем причины ожидания могут быть различны. Возможно, процессу потребовался некоторый неразделяемый ресурс, который сейчас занят, или он был вытеснен более приоритетным процессом.
Однако у этой модели есть некоторые недостатки. Главным из них является то, что в состоянии ожидания мы никак не учитываем причину, т.е. чего собственно говоря ждет процесс. Из-за этого может возникнуть следующая ситуация: мы достали ожидающий процесс из очереди и начали его исполнять, однако выяснилось, что этот процесс не дождался того,

что ему требовалось, и поэтому сразу же вновь ушел в ожидание. Переключение с процесса на процесс требует времени, а тут мы сделали это два раза, да еще и без всякой пользы — понятно, что этот момент можно оптимизировать.
Появляется мысль о том, чтобы использовать различные очереди для процессов, которые находятся в ожидании какого-либо внешнего события, и для процессов, которые готовы исполняться, но мы их прервали. На графе состояний и переходов это отразится следующим образом (рис. 1.6.2): мы введем новое состояние «Готов», в котором будут находиться процессы, готовые к исполнению. В это же состояние будут попадать только что созданные процессы, т.к. мы предполагаем, что любой процесс изначально готов к исполнению. Теперь из состояния «Выполняется» есть два пути: в состояние «Готов» и в состояние «Ожидает». Выбор одного из этих путей будет определяться тем, какое именно событие прервало исполнение этого процесса.

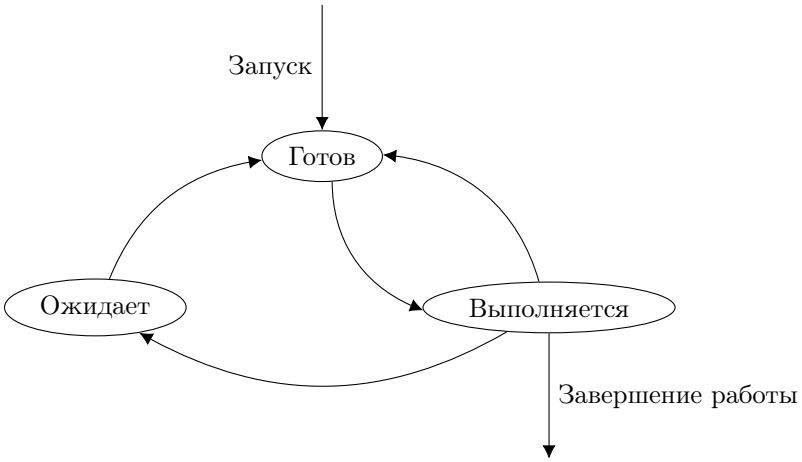

Рис. 1.6.2: Трехуровневая модель состояний

Данная трехуровневая модель состояний реализуется любой операционной системой, однако даже этих трех уровней может не хватать, тогда в зависимости уже от специфики операционной системы могут добавляться новые уровни. Здесь стоит заметить, что порождение процесса это не мгновенная операция: например, в Linux нужно скопировать уже существующий процесс и поместить код нового процесса в адресное пространство. Также процессу при создании могут потребоваться некоторые ресурсы, которые тоже надо получить. Таким образом мы приходим к тому, что рождение процесса это отдельное состояние, которое также должно быть отражено в графе состояний (рис. 1.6.3).

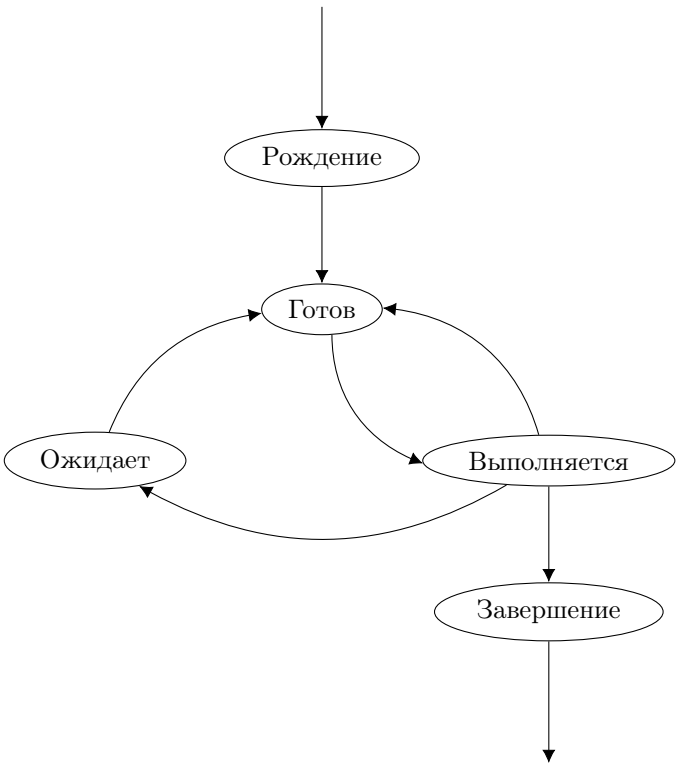

Рис. 1.6.3: Новые состояния: «Рождение» и «Завершение»

По абсолютно аналогичным соображениям мы также добавляем состояние «Завершение» в граф состояний: например, процесс при завершении отправил `SIGCHLD` своему родителю, чтобы тот его обработал. Родительский процесс может в

это время находится в состоянии ожидания, поэтому текущий процесс должен будет будет дождаться, пока родитель обработает его сигнал. В это время он будет находиться в состоянии «Завершение».
В Linux (**только с точки зрения представления результата пользователю**) состояния «Готов» и «Выполняется» оба обозначаются как `running`, т.к. не имеет смысла показывать какой конкретно процесс сейчас выполняется — переключение процессов происходит настолько быстро, что человек просто не успеет прочитать эту информацию, а процесс уже переключится. А вот состояние «Ожидает» представлено в двух видах: прерываемое ожидание и непрерываемое. В случае непрерываемого ожидания у процесса существует конкретная операция, результат выполнения которой он ждет, и прерывать это ожидание некорректно с той точки зрения, что это может поменять контекст выполнения, а это в свою очередь может привести к ошибке. Обращения к такому процессу необходимо буферизовать и передать их процессу только после завершения ожидания. Прерываемое ожидание (как понятно из названия) это ожидание, которое может быть прервано аппаратным прерыванием или сигналом от другого процесса. Оно характерно для интерактивных процессов, например для веб-серверов.
Также есть еще один момент, характерный для Linux'а (и не только). Допустим, что процесс в состоянии «Завершение» послал сигнал `SIGCHLD` своему родителю, а тот находится в непрерывном ожидании и не выходит из него (но не завершается). В этом случае текущий процесс спустя некоторое время переходит в состояние «Зомби» и становится zombie процессом — это еще одно состояние в графе состояний. В это состояние можно попасть единственным образом (описанным выше) и из него нет выходов.
Необходимо добавить несколько слов про состояние «Завершение». Оно может быть как корректным (процесс выполнил свою работу и ему больше нечего делать), так и аварийным. Обработка аварийных ситуаций чаще всего происходит с помощью аппаратных прерываний: ядро ловит прерывание, запускает соответствующий обработчик, который посылает процессу сигнал, чтобы тот мог неким образом обработать его. Это позволит родительскому процессу узнать код завершения данного процесса и понять, что пошло не так.
При работе в многопоточном режиме может возникать ситуация рассинхронизации потоков: допустим, один поток пытается использовать данные, которые должны быть вычислены другим потоком, однако другой поток еще не успел ничего вычислить (и мы вручную никак не проконтролировали эту ситуацию). В итоге мы должны перейти из состояния «Выполняется» в состояние «Завершение» с некоторым кодом ошибки. Однако если мы попытаемся повторить последнюю операцию через некоторое время, то второй поток (возможно) успеет провести необходимые вычисления и ошибки не произойдет. Для обработки таких случаев вводится новое состояние «Исключетельная ситуация» (рис. 1.6.4) — и мы можем решать сколько попыток давать процессу, прежде чем окончательно отправить его на завершение.

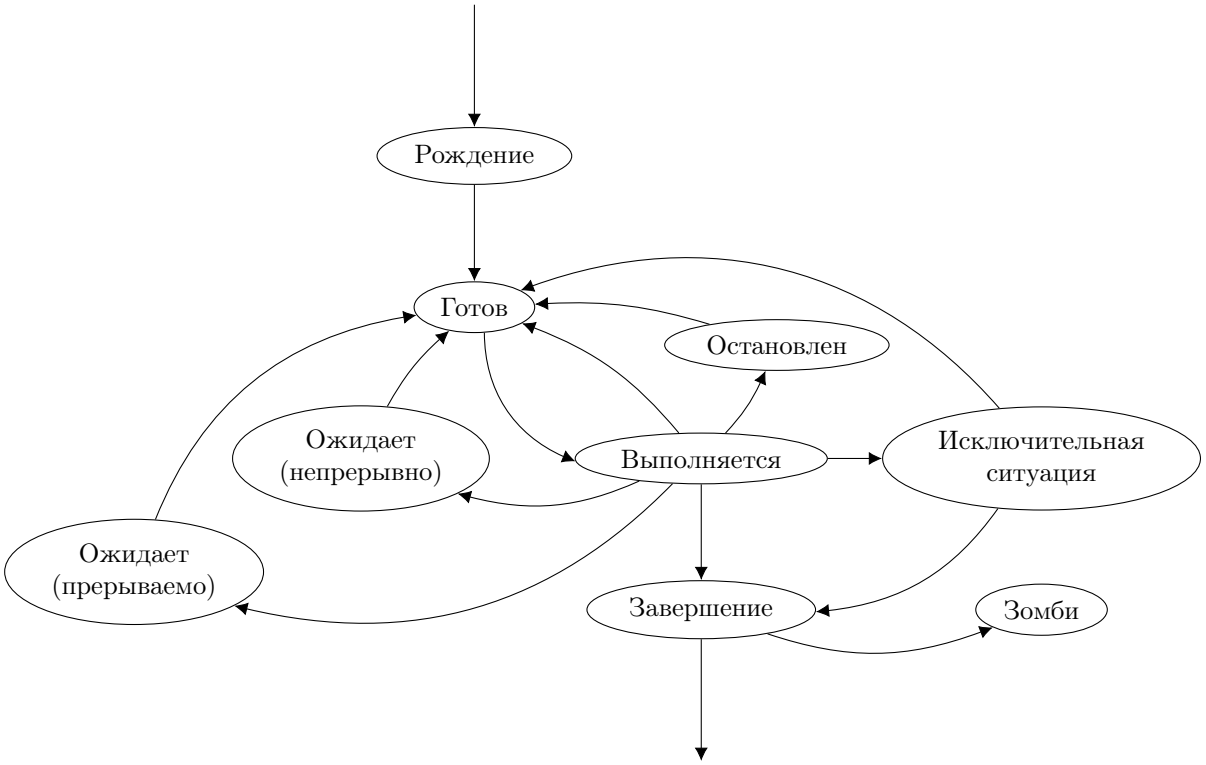

Рис. 1.6.4: Расширенный граф состояний процесса

Похожим образом вводится состояние «Остановлен», только если критические ситуации обрабатывались автоматически, то остановка процесса это обычно результат действий человека. Допустим, администратор видит, что некоторый процесс внезапно начал использовать чрезмерно много ресурсов (хотя не должен). Он может приостановить процесс и разобраться, в чем причина такого поведения, после чего решить эту проблему и возобновить работу процесса. В Linux для остановки и возобновления работы процесса используются сигналы `SIGSTOP` и `SIGCONT`.

Задача планирования

Под планированием вычислительного процесса понимается распределение времени процессора между выполняющимися заданиями или процессами, причем данное распределение времени должно удовлетворять некоторым критериям (скорость исполнения, время отклика и т.п.). Из процессов, претендующих на нахождение в состоянии «Выполняется», формируется очередь, которой и управляет ОС. Однако эта очередь не единственная: существуют и другие устройства, к которым также есть свои очереди. Помимо этого т.к. граф состояний процессов стал значительно более сложным (рис. 1.6.4), чем был изначально (рис. 1.6.1), то теперь на каждый переход из состояния в состояние также есть своя очередь. Итого операционная система мало того, что должна управлять всеми этими очередями, так она еще и должна делать это в совокупности.
Пусть есть некоторое количество процессов, которым необходимо предоставить какой-либо ресурс. Операционная система некоторым образом выстраивает очередь из этих процессов, и они начинают исполняться. Однако система открытая: пока процессы исполняются могут произойти некоторые события, которые приведут к тому, что изначальный план станет неоптимальным или даже невыполнимым. Допустим, что для того, чтобы этого избежать, мы будем прерываться после каждого выполненного процесса и осуществлять перепланирование. Таким образом у нас будет крайне маленький горизонт планирования: фактически, мы будем планировать лишь на один шаг вперед. Из плюсов такого решения можно отметить то, что мы будем довольно точно учитывать все изменения, которые будут происходить в системе, и следовательно делать выбор весьма оптимально. Однако планирование тоже требует времени и ресурсов — если мы будет так часто его осуществлять, то больше времени потратим на него, нежели на выполнение реальных процессов. В противовес такому подходу мы можем иметь большой горизонт планирования, т.е. мы может составить план на относительно длинный период времени и придерживаться его, не тратя время на перепланирование. Таким образом мы снижаем накладные расходы на перепланирование, однако повышается риск (особенно ближе к концу этого плана) того, что система будет работать не оптимально.
Итого мы имеем ситуацию выбора: у каждого из вариантов есть свои плюсы и минусы, и нельзя сказать, что один из них однозначно лучше другого в общем случае. Исходя из этого каждая операционная система (как правило) имеет несколько планировщиков и разные горизонты планирования: краткосрочное планирование, среднесрочное планирование и долгосрочное планирование. Еще раз посмотрим на рис. 1.6.4. Переход «Готов» — «Выполняется» находится в краткосрочном планировании: здесь используются весьма простые алгоритмы, и даже если они будут давать не самый оптимальный результат, то ничего страшного не произойдет, ведь перепланирование будет происходить довольно часто. К долгосрочному планированию относится переход «Рождение» — «Готов», т.е. принятие решения о самой возможности создания нового процесса, ведь новый процесс надолго изменит состояние всей системы: он будет требовать ресурсы, его необходимо будет учитывать в очередях и т.д. Таким образом решение о создании нового процесса это важное и необратимое решение, поэтому для него требуется именно долгосрочное планирование.
Пусть у нас есть некоторое количество процессов, которые находятся в прерываемом ожидании. Мы можем предположить (по каким-либо причинам), что часть из этих процессов будут находится в этом состоянии еще долго и переместить их данные в swap, чтобы освободить оперативную память для других процессов. Принятие подобных решений относится к среднесрочному планированию (на самом деле к среднему планированию относится не только это). Стоить помнить, что деление на кратко-, средне- и долгосрочное планирование весьма условно и зависит от операционной системы. Однако помимо этих трех уровней планирования отдельно выделяют очереди ввода-вывода, т.к. устройств ввода-вывода может быть несколько. Этот уровень планирования отличается тем, что если процесс попал в непрерываемое ожидание, то он встал в очередь к какому-либо устройству, и пока он не выйдет из этой очереди, то этот процесс можно не учитывать. Контролировать этот выход мы не можем: это не наше решение, в отличие от среднесрочного планирования, где именно мы решаем, убирать данные процесса в swap или нет.

Критерии планирования

1. Критерий справедливости.

   Гарантировать каждому процессу равную долю процессорного времени (или любого другого ресурса).

2. Критерий эффективности.

   Максимально эффективно использовать все предоставленные ресурсы.

3. Критерий сокращения полного времени выполнения.

4. Критерий сокращения времени ожидания.

5. Критерий сокращения времени отклика.

   Данный критерий важен для интерактивных систем.

Свойства алгоритмов планирования

1. Алгоритм планирования должен быть предсказуем.

   При многократном запуске на одних и тех же данных мы должны получать одинаковый (или достаточно близкий) результат.

2. Алгоритм планирования должен иметь минимальные накладные расходы.

3. Алгоритм планирования должен быть масштабируемым.

1.7. Лекция 24.03.20.

Параметры планирования

1. Статические параметры системы.

   К ним относятся предельные значения имеющихся ресурсов: количество доступных ядер, максимально доступный объем памяти и т.д.

2. Динамические параметры системы.

   К ним относятся доступные на текущей момент ресурсы: сколько сейчас свободно оперативной памяти, какая в среднем нагрузка на ЦП (например, за последние несколько минут) и т.д.

3. Статические параметры процесса.

   У каждого процесса есть собственные ограничения. К ним относятся, например, ограничения, связанные с правами доступа. Допустим, мы знаем, что у процесса нет прав на доступ к определенному устройству, а значит он никогда не сможет попасть в очередь к этому устройству. Мы можем учесть это при планировании. Помимо это у процесса есть «важность», которая также влияет на планирование.

4. Динамические параметры процесса.

   Они описывают то, как процесс использует ресурсы. К этой категории относится, например, CPU-burst — это то, сколько времени будет исполняться процесс до того момента, как он либо завершится, либо уйдет в ожидание, при условии, что мы дадим ему возможность выполняться на ЦП без прерываний. Ясно, что CPU-burst это всегда эвристика, потому что точное значение определить мы не можем. Также к этой категории относится IO-burst. Допустим, что процесс выполнил какие-то вычисления и ушел в ожидание ввода-вывода. Время, которое он будет там находится, и называется IO-burst. Как и в случае с CPU-burst мы не можем узнать точное значение этого параметра, а лишь можем дать ему примерную оценку.

Алгоритмы планирования

Алгоритмы планирования можно разделить на два больших класса, а именно: вытесняющее и не вытесняющее планирование. В случае не вытесняющего планирования если мы дали процессу выполняться, то он только сам может приостановить или завершить свое исполнение. А в случае вытесняющего планирования мы имеем некоторый механизм, который позволит нам прервать выполнение текущего процесса для того, чтобы передать ресурс процессора следующему процессу. Этот механизм может быть разным: прерывание по таймеру, прерывание из-за появления более приоритетного процесса и т.д.
Для сравнение эффективности алгоритмов планирования мы будем использовать следующие показатели:

1. $\tau_{\text{полн}}$ — полное время исполнения всех процессов, от начала исполнения первого процесса до конца исполнения последнего.

2. $\tau_{\text{исполн}}$ — среднее время исполнения. Для каждого процесса сложим время, которое он действительно исполнялся, и время, в течение которого он был готов исполняться, но находился в состоянии ожидания. Теперь найдем среднее значение такой величины для всех процессов — это и будет среднее время исполнения.

3. $\tau_{\text{ожид}}$ — среднее время ожидания.

I. First Came First Served (FCFS)

Из названия алгоритма понятно, что он предполагает выполнение процесс в том порядке, в котором они появлялись.

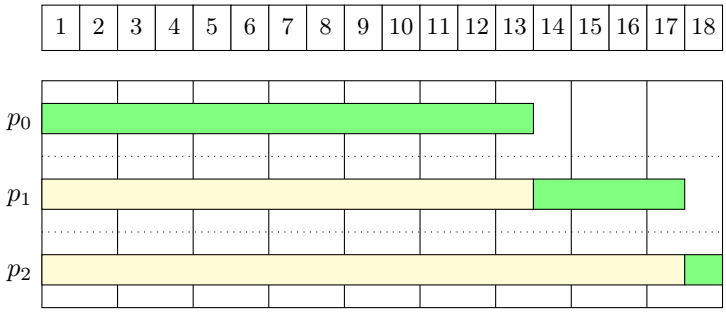

Рис. 1.7.1: FCFS: $0 \to 1 \to 2$

Пример 1.7.2. Пусть у нас три процесса и известны их CPU-burst: $p_0 = 13$, $p_1 = 4$ и $p_2 = 1$. Значения приведены в условных единицах, т.к. в данном примере нам не очень важны именно единицы измерения. Допустим, что мы будем обслуживать процессы в порядке $0 \to 1 \to 2$. Тогда процесс их исполнения можно проиллюстрировать как показано на рис. 1.7.1. Вычислим характеристики для этого случая:

$$\tau_{\text{полн}} = 18 \qquad \tau_{\text{исполн}} = \frac{13 + 17 + 18}{3} = 16 \qquad \tau_{\text{ожид}} = \frac{0 + 13 + 17}{3} = 10$$

Пример 1.7.3. А теперь представим, что процессы были расположены в порядке $2 \to 1 \to 0$ (см. рис. 1.7.4) и еще раз вычислим характеристики.

$$\tau_{\text{полн}} = 18 \qquad \tau_{\text{исполн}} = \frac{1 + 5 + 18}{3} = 8 \qquad \tau_{\text{ожид}} = \frac{0 + 1 + 5}{3} = 2$$

Несмотря на то, что общее время исполнения не изменилось (и не могло измениться), ясно второй вариант будет лучше, потому что ожидающие процессы также используют ресурсы: например, они занимают место в оперативной памяти. Таким образом, если процессы меньше ждут, то оперативная память освобождается быстрее, а значит мы быстрее сможем использовать ее для других задач.

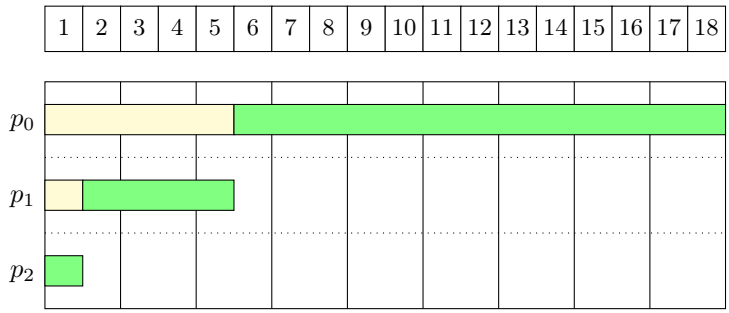

Рис. 1.7.4: FCFS: $2 \to 1 \to 0$

Исходя из этих примеров может показаться, что логично всегда «пропускать» маленькие процессы вперед. Однако это не так: пусть у нас есть некоторый большой процесс, который пропустил несколько маленьких процессов. Пока маленькие процессы исполнялись, появились новые маленькие процессы — возникает вопрос, что с ними делать? Если постоянно пропускать процессы вперед очереди, то большой процесс будет ждать очень долго (а в худшем случае до него очередь так и не дойдет).

II. Round Robin (RR)

Предыдущий алгоритм был не вытесняющим — а теперь давайте рассмотрим его вытесняющую версию. Идея заключается в том, что мы выбираем некоторый отрезок времени, и по очереди даем всем процессам выполняться выбранное количество времени.

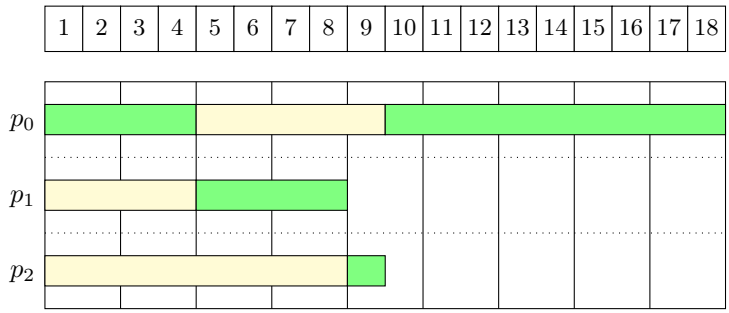

Рис. 1.7.5: RR: $0 \to 1 \to 2$, $q = 4$

Пример 1.7.6. Вернемся к предыдущему примеру. Пусть у нас все также есть три процесса с известными CPU-burst: $p_0 = 13$, $p_1 = 4$ и $p_2 = 1$. Сначала рассмотрим порядок выполнения $0 \to 1 \to 2$ при кванте исполнения $q = 4$ (рис. 1.7.5) и вычислим характеристики.

$$\tau_{\text{полн}} = 18 \qquad \tau_{\text{исполн}} = \frac{18 + 8 + 9}{3} \approx 11.6 \qquad \tau_{\text{ожид}} = \frac{5 + 4 + 8}{3} \approx 5.6$$

Можно заметить, что несмотря на то, что мы выбрали «плохой» порядок процессов, итоговые характеристики получились все равно лучше, чем в 1.7.2. При этом важно отметить, что если бы мы выбрали «хороший» порядок процессов ($2 \to 1 \to 0$), то получили бы такой же результат, что и в 1.7.3 (не будем это отдельно иллюстрировать, т.к. там будет полностью такая же ситуация и характеристики).

Пример 1.7.7. А теперь попробуем уменьшить квант исполнения до $q = 1$ и посмотрим, что из этого выйдет (рис. 1.7.8). Вычислим характеристики.

$$\tau_{\text{полн}} = 18 \qquad \tau_{\text{исполн}} = \frac{18+9+3}{3} = 10 \qquad \tau_{\text{ожид}} = \frac{5+5+2}{3} = 4$$

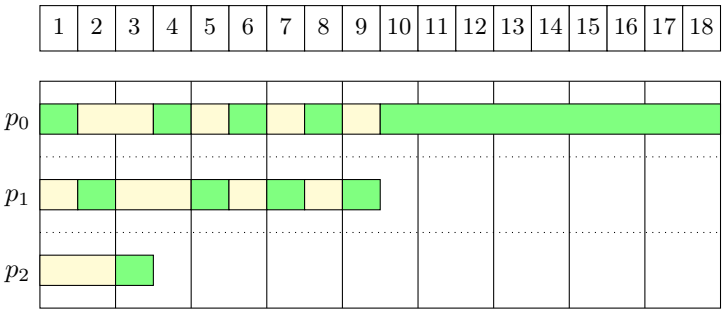

Рис. 1.7.8: RR: $0 \to 1 \to 2$, $q = 1$

Казалось бы, что из примера следует, что чем меньше квант исполнения, тем ближе наше распределение времени ЦП к идеальному, однако это не совсем так. Дело в том, что переключение между процессами не мгновенное: необходимо сохранить контекст одного процесса, загрузить контекст другого, обновить статусы процессов и т.д. Таким образом в FCFS мы выполняем всего два переключения между процессами (меньше просто не получится), а вот в RR переключений становится уже больше: 3 при $q = 4$ и целых 9 при $q = 1$.
Итого, если схематично изобразить график зависимости реального времени исполнения от величины кванта, то получится некоторая парабола, ветви которой направлены вверх. Понятное дело, что при бесконечно большом кванте мы фактически получаем FCFS (который как мы уже видели в 1.7.2 не идеален), а при кванте бесконечно близком к нулю оптимального решения мы не получим, т.к. будем иметь слишком большие накладные расходы на переключение между процессами. Значит задача поиска оптимального кванта времени это тоже отдельная задача оптимизации, причем решение это задачи сильно зависит от CPU-burst процессов, которые хотят исполняться. Однако динамически вычислять квант это слишком дорого: мы потратим много времени на вычисление, а ситуация быстро изменится и наши вычисления станут неактуальными.

III. Shortest job first (SJF)

Этот алгоритм существует в вытесняющем и не вытесняющем вариантах, однако чаще всего используется именно вытесняющий вариант, в котором через некоторые кванты времени происходит проверка на существование более приоритетного процесса.

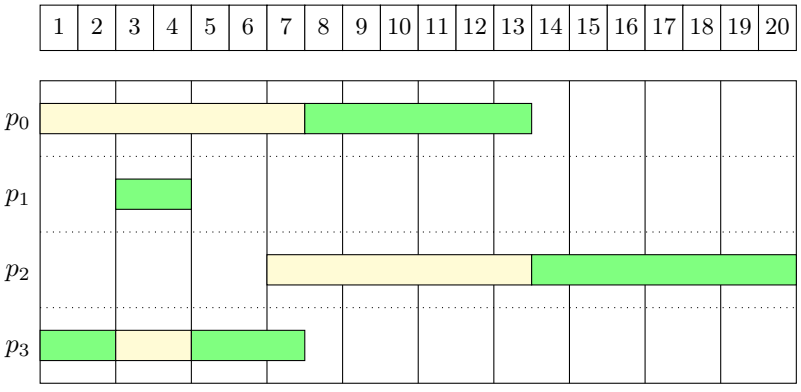

Рис. 1.7.9: SJF

Пример 1.7.10. Пусть у нас есть четыре процесса с известными CPU-burst: $p_0 = 6$, $p_1 = 2$, $p_2 = 7$ и $p_3 = 5$. Для наглядности они стартуют не одновременно, а с задержкой: $t_0 = 0$, $t_1 = 2$, $t_2 = 6$ и $t_3 = 0$. Проиллюстрируем то, как будут исполняться процессы в этом случае (рис. 1.7.9).

Как мы видим этот алгоритм (в вытесняющем варианте) объединяет в себе идею предыдущего алгоритма и упорядочивание очереди по возрастанию. Однако здесь мы снова возвращаемся к тому, что если всегда пропускать маленькие процессы вперед, то до больших процессов очередь может просто не дойти. Про такие процессы говорят, что они находится в состоянии «голодания», т.е. они готовы исполняться, но ОС не предоставляет им право на исполнение, и они висят в ожидании.

IV. Гарантированное планирование

Обозначим N — количество процессов в системе, T_i — время сеанса i-ого процесса (т.е. время с момента его готовности до текущего момента), а τ_i — время исполнения i-ого процесса (сколько времени мы давали этому процессу исполняться). Если предыдущие алгоритмы действовали с позиции эффективности, то этот алгоритм действует с позиции справедливости, а именно: мы хотим, чтобы выполнялось $\tau_i \sim \frac{T_i}{N}$. Если мы вычислим отношение этих двух величин

$$R_i = \frac{\tau_i}{T_i} \cdot N$$

то получим коэффициент R_i, который называется коэффициентом справедливости (для i-ого процесса). Далее если мы возьмем процесс с наименьшим коэффициентом справедливости (т.е. процесс, который мы больше всего «обделили») и дадим ему исполняться, то его коэффициент справедливости будет увеличиваться, в то время как коэффициент справедливости других процессов будет уменьшаться (т.к. у них будет расти знаменатель, а числитель будет оставаться таким же). Таким образом через некоторое время самый «обделенный» процесс перестанет быть таковым, и его место займет другой процесс.
Итого алгоритм заключается в том, что мы берем процесс с наименьшим коэффициентом справедливости и даем ему исполняться в течении некоторого кванта времени. Далее мы прерываем этот процесс, снова находим процесс с наименьшим коэффициентом справедливости, предоставляем ему возможность исполнятся и так по кругу. Таким образом мы повысим справедливости распределения времени, но потеряем в эффективности. Однако у этого алгоритма есть и другие проблемы. Первая из них заключается в накладных расходах: вычислять коэффициенты справедливости, являющиеся вещественными числами, а потом еще и сравнивать их для поиска наименьшего это недешевая операция (особенно учитывая то, что делать это нужно весьма часто). Второй проблемой является неустойчивость ко «взлому»: можно сделать так, что процесс будет запущен, но не будет требовать времени исполнения, а будет просто ничего не делать. Потом, когда время его сеанса T_i возрастет достаточно, он начнет свои вычисления, причем из-за большого T_i, которое он накопил, данный процесс будет получать приоритет и фактически на время своего исполнения полностью вытеснит все остальные процессы.

V. Многоуровневые очереди

Во всех предыдущих алгоритмах мы использовали некоторый программно вычисляемый приоритет: в SJF это был CPU-burst (мы всегда выбирали процесс с наименьшим CPU-burst), при гарантированном планирование это был коэффициент справедливости (опять же, мы выбирали процесс с наименьшем коэффициентом справедливости). А что если приоритет будет задаваться извне? Допустим, при создании каждого процесса пользователь будет сопоставлять ему некоторое целое число, которое мы будем считать приоритетом этого процесса. Здесь сразу же возникает проблема с дискретностью шкалы приоритетов: пусть у меня есть процессы с приоритетами 2 и 3. Я хочу создать процесс, приоритет которого будет меньше, чем у первого процесса, но больше, чем у второго. Однако я не могу сделать приоритет 2.5 в силу того, что приоритет это целое число.
Эту проблему можно попытаться решить следующим образом: пусть мы изначально будем выдавать приоритеты с шагом в (например) 10, а потом если потребуется, то будем выдавать промежуточные приоритеты, например 25 или 22 и т.д. Однако это не решает проблему полностью: все равно в какой-то момент мы упремся в то, что захотим выдать приоритет, который не будет являться целым числом. Делать приоритет не целым числом мы не хотим, т.к. работа с нецелыми числами будет иметь повышенные накладные расходы. К тому же, если вдруг возникнет два процесса с одинаковым приоритетом, то какой из них нужно выбрать? Получается, что нужно вводить некоторые дополнительные правила.
Вместо этого была предложена следующая идея: пусть мы изначально зададим некоторый фиксированный набор приоритетов (например от 1 до P), который можно выдать процессу, и позволим нескольким процессам иметь одинаковый приоритет. Таким образом для каждого значения приоритета сформируется своя очередь. Мы будем поступать следующим образом: сначала выполним все процессы с первым приоритетом, потом все процессы со вторым приоритетом и т.д. Если в какой-то момент появляется новый процесс с более высоким приоритетом, чем исполняемый в данный момент, то мы прерываем текущий процесс и переключаемся на появившийся процесс. Если внутри одного приоритета есть несколько процессов, то мы используем например Round Robin для того, чтобы определить очередность выполнения процессов.
Однако у такого похода есть проблема, о которой мы уже говорили. Пусть у нас есть процесс с самым низким приоритетом. Существует вероятность того, что до этого процесса выполнение никогда не дойдет, потому что постоянно будут появляться новые процессы с более высоким приоритетом. Да, они будут исполняться, но на их место будут приходить все новые и новые процессы. Таким образом наш процесс будет находиться в состоянии «голодания». Для того, чтобы как-то этого избежать, мы можем поступить следующим образом: если процесс за некоторое заранее отведенное время (timeout) ни разу не получил возможность исполняться, то мы повышаем его приоритет не единичку. Таким образом мы как бы создаем «лифт», благодаря которому низкоприоритетные процессы через некоторое время смогут повысить свой приоритет и получить возможность исполняться. После того, как процесс получил свой квант времени исполнения, он «отбрасывается» в очередь, соответствующую его изначальному приоритету. После этого он снова вынужден ждать и постепенно повышать свой приоритет, чтобы вновь получить возможность исполняться.
Итого, низкоприоритетные процессы вынуждены пропорционально больше ждать, чем высокоприоритетные процессы, но при этом они не голодают и спустя некоторое время гарантированно получат возможность исполняться. Казалось бы это хорошее решение, однако у него все еще есть недостатки. Мы разобрались с проблемой голодания, но не разобрались с проблемой эффективности: если мы изначально неправильно распределим приоритеты, то общая эффективность

нашей системы будет низкой. Для того, чтобы это исправить, появляется идея многоуровневых очередей с обратной связью. Пусть мы также имеем несколько очередей и для каждой очереди выбираем квант непрерывного выполнения, причем очереди с более низким приоритетом соответствует бóльший квант времени. Новорожденный процесс изначально попадает с первую очередь. Если, когда до него дойдет время, он успеет выполнится за квант времени, соответствующий этой очереди, то мы оставим его в ней. Если же он не успеет этого сделать, то мы его прервем и отправим в очередь с более низким приоритетом. Таким образом, процессы, требующие много времени исполнения, будут попадать во все более и более низкоприоритетные очереди, но при этом когда до них дойдет время, то они будут получать все больше и больше непрерывного времени исполнения.
Такой подход пытается объединить в себе идеи SJF и идеи гарантированного планирования. Однако у нас остается проблема с голоданием: если процесс попал в низкоприоритетную очередь, то он, конечно, получит много времени непрерывного исполнения, когда до него дойдет очередь, но что, если эта очередь никогда до него не дойдет? С этого момента развитие ОС пошло таким образом, что люди пытались скомбинировать уже имеющиеся алгоритмы и найти некоторый компромисс в требованиях, выдвигаемых к этим алгоритмам. Конкретные идеи, реализованные в разных ОС, будут рассмотрены на следующей лекции.

1.8. Лекция 24.??.??.

Сначала сформулируем некоторые требования, которыми мы выдвигаем к нашему решению в области планирования процессов:

1. Поддержка внешнего управления приоритетами.

   У пользователя должна быть возможность указать более приоритетный процесс.

2. Эффективное использование ресурсов.

   Здесь речь идет не только про ресурс центрального процессора, но и про оперативную память и другие ресурсы. Основное требование заключается в том, что если есть процессы, которые готовы исполняться, то они должны исполняться, и ЦП не должен простаивать. Также нужно, чтобы процессы как можно быстрее покидали оперативную память, чтобы освободить ее для других процессов. Помимо этого необходимо минимизировать накладные расходы. Например, на переключение между пользовательским режимом и режимом ядра, на переключение между процессами и т.д. К этому пункту также можно отнести эффективное использование кэша процессора.

3. Минимальные накладные расходы.

   Необходимо сделать так, чтобы сам алгоритм принятия решения об упорядочивании очереди не затрачивал слишком много ресурсов. Причем здесь необходимо помнить о том, что зачастую сам алгоритм выбора довольно быстрый и не требует много ресурсов, однако ему для работы требуются некоторые заранее собранные данные о процессах, некоторые предпосчитанные значения, и вот их вычисление может быть достаточно ресурсозатратным. Также как обычно не стоит забывать о том, что ресурсом является не только процессорное время, но и, например, оперативная память: алгоритм планирования, требующий слишком много памяти нас вряд ли будет устраивать.

4. Минимальные риски возникновения тупиков.

   Подробнее про тупики мы поговорим на следующих лекциях, сейчас рассмотрим лишь небольшой пример. Пусть у нас есть процесс p_1, который сейчас использует ресурс R. Спустя некоторое время более приоритетный процесс p_0 тоже начинает требовать ресурс R. Возникает проблема: процесс p_0 не может выполняться, т.к. ему нужен ресурс R. Отобрать этот ресурс у p_1 мы не можем: например, ресурс R это принтер и процесс p_1 уже начал на нем что-то печатать. В таком случае попытка отобрать ресурс может привести к потере целостности данных. Позволить процессу p_1 завершить использование ресурса R мы также не можем: в таком случае мы нарушим наши же правила о приоритетности процессов. Так и возникает тупик: оба процесса не могут выполняться и просто бесконечно ждут.

Для удовлетворения первого требования хорошо подойдут многоуровневые очереди, которые мы уже рассмотрели. Для второго требования хорошо подойдет алгоритм SJF, а также адаптируемые кванты непрерывного выполнения (чтобы минимизировать различные переключения). Также для того, чтобы попадать в кэш, хорошо было бы иметь раздельные очереди к разным ядрам (но при этом должна быть возможность переходить между ними). Для выполнения третьего пункта нужно пользоваться по возможности целочисленной арифметикой, а также использовать алгоритмы за $O(n)$ в худшем случае (а лучше что-то побыстрее). Также мы хотим, чтобы структуры данных используемые в алгоритме, занимали разумное количество памяти. Частичным решением для требования четвертого пункта является гарантированное планирование: если все процессы будут получать примерно поровну процессорного времени, то шанс возникновения тупиков уменьшится (п.а. а почему?).

Планирование в Windows

Понятное дело, что нельзя говорить просто Windows — нужно указывать версию, однако т.к. Windows это проприетарная система, то мы можем лишь полагаться на открытую документацию (и на некоторое reverse engineering исследования). Вдобавок к этому, будем считать, что общие концепции планирования не сильно различаются от версии к версии.
Планирование в Windows основывается на идее многоуровневых очередей: их 32 штуки, и чем больше номер очереди, тем приоритетнее процесс. Если в рамках одной очереди есть несколько процессов, то они переключаются между собой

при помощи Round Robin'а. Все очереди разделены на два класса: с 0-ой по 15-ую это динамические (переменные) приоритеты, а с 16-ой по 31-ую это процессы реального времени. Особенность процессов реального времени заключается в том, что ОС не меняет выданные им приоритеты. А вот у процессов с изначальными приоритетами от 0 до 15 приоритет может меняться (решение об изменении приоритета принимает ОС), однако он не может выйти за границы своего класса, т.е. не может стать больше 15-ти. Стоит сказать несколько слов про нулевую очередь. Это очередь обнуления страниц: т.е. когда ОС не занята другими процессами, то она занимается тем, что обнуляет страницы памяти, чтобы данные уже завершившихся процессов не были получены новыми процессами.

Откуда пошло название процессы «реального времени»? В прошлом была идея о том, что можно создать механизм процессов реального времени, т.е. процессов для которых время отклика не будет превышать некую заранее определенную константу. Однако позже выяснилось, что осуществить эту идею в универсальной операционной системе практически невозможно: это может быть связано, например, с тем, что данные процесса находятся в файле подкачки, доступ к которому не очень быстрый, или с непопаданием в кэш процессора и т.д. Таким образом реализовать ОС с процессами реального времени можно, но для этого придется пожертвовать оптимальностью использования ресурсов и универсальностью системы.

Для внешнего управления приоритетами существует 6 классов приоритетов процессов, причем каждому классу соответствует номер очереди:

1. Процесс реального времени (realtime) — 24-ая очередь.
2. Высокий (high) — 13-ая очередь.
3. Выше нормального (above normal) — 10-ая очередь.
4. Нормальный (normal) — 8-ая очередь.
5. Ниже нормального (below normal) — 6-ая очередь.
6. Бездействующий (idle) — 4-ая очередь.

Стоит отметить, что приоритет realtime может задать только администратор, у обычного пользователя нет такой возможности. Здесь необходимо вспомнить о том, что процесс это множество потоков, и у нас есть возможность отдавать приоритет определенным потокам. Для этого существуют уровни насыщения:

1. Time critical (+15).
2. Highest (+2).
3. Above normal (+1).
4. Normal (±0).
5. Below normal (−1).
6. Lowest (−2).
7. Idle (−15).

Как это работает? Мы создаем процесс и задаем ему некоторый приоритет из первого списка. Это базовое значение приоритета для всех его потоков. Далее с помощью системного вызова можно задать некоторое насыщение для конкретного потока. Это насыщение прибавляется к базовому приоритету и получается итоговое значение приоритета для данного потока. Здесь важно помнить о том, что с помощью насыщения потока нельзя перевести его из одного класса в другой, т.е. даже если изначально приоритет был high, и мы использовали насыщение time critical, то итоговый приоритет все равно будет равен 15. Аналогично насыщение idle опустит realtime-приоритетный процесс лишь до 16-ого уровня.

Мы поговорили про внешнее управление приоритетами, теперь можно поговорить и про внутреннее. Операционная система может менять динамический приоритет в зависимости от состояния процесса. «Хорошим» для ОС является интерактивный процесс: такой процесс часто уходит в ожидание ввода-вывода и не тратит много процессорного времени. Это согласуется с идеей SJF, но как это реализовано? Каждый процесс, который выходит из различных операций ввода-вывода, получает временное повышение приоритета. Оно зависит от типа завершенной операции ввода-вывода и версии ОС, например, для клавиатуры это может быть порядка +6 (но перейти из одного класса в другой таким образом все равно нельзя). Также временное повышение приоритета получает процесс, который ждал освобождения какого-либо ресурса. Это нужно для того, чтобы процесс побыстрее поработал с этим ресурсом и освободил его, таким образом уменьшая вероятность возникновения тупика. Помимо этого приоритет получает текущее активное (выбранное) окно.

Понятно, что все эти повышения приоритета не вечные, а временные. В Windows для этого вводится некоторый квант времени, и если по истечении двух таких квантов времени процесс самостоятельно не ушел в ожидание, то мы прерываем выполнение и понижаем ему приоритет. Если, когда до него снова дошла очередь, он опять не уложился в два кванта времени, то мы опять понижаем ему приоритет, и так далее, пока его приоритет не вернется к изначальному значению.

Для борьбы с голоданием в Windows существует следующее решение: если какой-то процесс не исполнялся в течении 4-ех секунд, то ему на один квант дается сразу 15-ая очередь, после чего он возвращается в свою исходную очередь. Это также помогает снизить риски, связанные с тупиками.

Планирование в Linux. Планировщик $O(1)$.

До 2.6 ядра Linux использовался планировщик, который работал за $O(n)$: мы проходили по всем процессам и для каждого процесса считали некоторую характеристику, после чего выбирали процесс с наилучшей характеристикой. В ядре 2.6 появился планировщик, который работает за $O(1)$. Также отличием этого планировщика стало то, что он имеет свои системы очередей к каждому ядру, в отличие от старого планировщика с одной очередью. Как же устроен этот планировщик?

К каждому ядру есть свой набор из 140 пар очередей. Как и в Windows, они разбиты на два класса: с 1-ой по 100-ую это процессы реального времени, а со 101-ой по 140-ую это пользовательские (или динамические) процессы. В отличие от Windows, в Linux более приоритетными являются очереди с меньшим номером. Внутри второго (пользовательского) класса процессов пользователь может свободно менять приоритеты. Для это существуют показатель `nice`, который принимает значения от -20 до $+19$, где значению -20 соответствует 101-ая очередь, а значению $+19$ — последняя, 140-ая очередь.

В рамках одной очереди очередность определяется с помощью аналога FIFO, но при этом каждому процессу предоставляется некоторый квант непрерывного исполнения, который зависит от номера очереди: в 101-ой очереди он будет равен 200 мс, а в 140-ой — всего 10 мс. Если в течении этого кванта процесс успел уйти в ожидание, то когда он вернется из ожидания, то встанет в конец той же очереди с оставшимся квантом времени исполнения. Если же он не ушел в ожидание за отведенный ему квант, то он перемещается в парную очередь (как мы помним в Linux 140 *nap* очередей). Набор из 140 очередей, который мы сейчас обрабатываем, называется активным (active), а набор из 140 парных очередей, в которые мы убираем не ушедшие в ожидание процессы, называется неактивным (non-active). Стоит отметить, что когда мы убираем процесс реального времени в парную очередь, то он гарантированно попадает в очередь с тем же приоритетом, а вот пользовательский процесс может попасть в другую очередь (изменение приоритета может быть от -5 до $+5$, но перейти в класс realtime процессов все равно нельзя) в зависимости от вычисленного значения интерактивности.

Преимущество такого подхода заключается в том, что мы избавляемся от голодания: постепенно разбирая очереди из процессов, мы так или иначе дойдем до 140-ой очереди и предоставим процессам в ней время на исполнение. Причем это время будет меньше, чем время для высокоприоритетных процессов. В более поздних версиях этого планировщика появился механизм, благодаря которому высокоприоритетные интерактивные процессы после своего кванта исполнения могут попасть не в парную очередь, а в конец текущей очереди. Для оценки такой возможности вводится еще один параметр, который показывает, не будут ли остальные процессы ждать слишком долго из-за того, что мы позволили какому-то процессу выполняться дважды. Так или иначе, даже учитывая эту возможность, в какой-то момент все активные очереди будут разобраны. В этот момент произойдет обмен: активные очереди станут неактивными, а неактивные — активными. Таким образом новые неактивные очереди будут пустыми, а в новых активных очередях будут находится процессы, готовые к исполнению.

Для быстрого поиска очереди, в которой есть хотя бы один процесс, используется битовый вектор. Отсюда и название $O(1)$, потому что этот поиск происходит за константу. Однако стоит помнить о том, что для обеспечения эффективного использования ресурсов, нам необходимо вычислять коэффициент интерактивности, который является вещественным числом. Так же нам необходимо выполнить разные дополнительные проверки и вычисления после того, как процесс отработал свой квант времени. Это и стало причиной критики данного планировщика.

Перед тем, как перейти к другому планировщику, обратим внимание на еще один момент. Когда процесс порождает новый поток, то он попадает в ту же очередь, что и его родитель. Из-за этого может возникнуть ситуация, в которой высокоприоритетный процесс постоянно порождает много потоков, а низкоприоритетные процессы в это время голодают. Для решения этой проблемы было придумано следующее решение: когда процесс порождает новый поток, он делит свое оставшееся время исполнения с этим потоком. Таким образом, сколько бы потоков не породил процесс, они суммарно будут исполняться не больше, чем изначальный квант времени, выделенный процессу.

Планирование в Linux. Планировщик CFS.

CFS расшифровывается как Completely Fair Scheduler (абсолютно справедливый планировщик). Изначально у абсолютно справедливого планирования была проблема с тем, что коэффициент справедливости это вещественное число, и упорядочивать процессы по вещественному числу это слишком затратно. Однако с развитием этой идее появилось следующее предложение: для каждого процесса мы вводим две величины — время исполнения (`execution_time`) и максимальное время исполнения (`max_execution_time`). Далее мы выбираем процесс с наименьшим временем исполнения и даем ему возможность исполняться время, равное `max_execution_time`. Если он уйдет в ожидание, то после возвращения, мы позволим ему выполняться в течении оставшейся части предоставленного ему времени. Когда процесс потратит весь свой `max_execution_time`, то мы вставим его в очередь процессов в соответствии с его новым `execution_time`.

Для внешнего управления приоритетами мы можем менять то, как вычисляется `max_execution_time`. Изначально он вычисляется как время ожидания поделенное на количество процессов. Однако при помощи параметра nice (который остался ровно таким же ради совместимости) мы можем домножать `max_execution_time`на некоторое значение, тем самым давая больше времени более приоритетным процессам. Итого мы сортируем процессы уже по

целочисленной переменной `execution_time`, что уже лучше чем в изначальной идее справедливого планирования. Казалось бы, асимптотика все равно останется равной $O(n)$, т.к. мы вынуждены искать место, в которое нужно вставить отработавший процесс. Однако если представить очередь в виде красно-черного дерева, то такая вставка будет работать за $O(\log n)$. Если вдобавок к этому хранить указатель на самый левый элемент дерева, то мы всегда за $O(1)$ сможем узнать, какой процесс необходимо сейчас выполнять. Итого получается довольно удобный подход, в котором не нужно долго и сложно считать интерактивность и разбираться с очередями.
Напоследок несколько слов про многопроцессорность и кэши в рамках планирования в Linux. У обоих рассмотренных планировщиков к каждому ядру существует своя очередь, однако раз в некоторый промежуток времени (≈ 200 мс) происходит перебалансировка процессов между процессорами. Также отдельно в CFS вводится такой параметр как гранулированность — это специальная величина, определяющая нижнюю границу для `max_execution_time` (чтобы не было слишком частых переключений между процессами). Помимо этого нужно сказать, что многие параметры планирования в Linux, о которых мы упомянули, можно менять, тем самым настраивания планировщик под конкретную задачу или специфику работы.

1.9. Лекция 24.??.??.

Взаимодействие процессов

Взаимодействие процессов бывает внешним и внутренним. Под внешним взаимодействием понимается возможность процессов обменяться управлением или данными. Внешнее взаимодействие инициируется нами, а значит мы точно знаем, чего нам хочется. В Linux внешнее взаимодействие по данным можно осуществить, например, с помощью pipe, передав вывод одного процесса на вход другому. Внешнее взаимодействие по управлению осуществляется через сигналы: один процесс может послать сигнал другому процессу, а тот, после обработки сигнала, изменит свое поведение. В целом, со внешним взаимодействием нет особых проблем: ОС нужно лишь осуществлять буферизацию, т.к. когда один процесс посылает другому сигнал, то адресат может быть не готов его принять. В этом случае необходимо составить очередь и, когда процессу будет предоставлено время исполнения, передать ему сигналы из этой очереди.
Значительно более сложной задачей является задача вынужденного взаимодействия, когда процессы начинают конкурировать за неразделяемый ресурс. Самый простой пример, который здесь можно привести это пример с принтером: один процесс отправил часть данных для печати в принтер, но потом его вытеснил другой процесс, который тоже отправил часть своих данных в принтер. В итоге принтер распечатает не то, что мы хотели, если вообще что-то распечатает. Более неприятными являются ситуации, в которых неразделяемый ресурс это, например, какие-то структуры файловой системы. В такой ситуации конкуренция двух процессов может привести к потере данных во всей файловой системе. Таким образом возникает проблема, которая получила название взаимоисключение, т.е. мы должны обеспечить взаимоисключение доступа процессов к какому-либо неразделяемому ресурсу.
Мы не можем при рождении процесса блокировать все ресурсы, которые потенциально могут быть им использованы — в таком случае, нам, чтобы, например, поработать с принтером, придется закрыть все приложения, которые умеют с ним работать, кроме одного, которое на самом деле хочет что-то напечатать. Однако нам и не нужно такое жесткое решение: достаточно блокировать ресурс только на то время, когда приложение его действительно использует. Из исходного кода приложения четко можно выделить часть, которая непосредственно работает с неразделяемым ресурсом. Такую часть стали называть критической секцией кода относительно этого неразделяемого ресурса. Важно помнить, что одно приложение может взаимодействовать с несколькими неразделяемыми ресурсами, и, как следствие, иметь несколько критических секций относительно разных ресурсов. Более того эти секции могут даже пересекаться. Таким образом задача взаимоисключения сводится к тому, что мы должны обеспечить невозможность двух и более процессов одновременно находится в критических секциях относительно одного и того же неразделяемого ресурса.
Эту задачу можно попытаться решить следующим образом: в код каждого процесса перед входом в критическую секцию и после выхода из нее необходимо добавить так называемые пролог и эпилог. В прологе процесс сообщает ОС о том, что входит в критическую секцию и собирается использовать неразделяемый ресурс. ОС либо позволяет ему это сделать, либо приостанавливает его, т.к. ресурс в данный момент занят. В эпилоге процесс сообщает о том, что он вышел из критической секции и больше не нуждается в неразделяемом ресурсе. Однако не все так просто: помимо взаимоисключения нам также необходимо обеспечивать прогресс. Под прогрессом мы будем понимать невозможность ситуации, при которой процесс не может использовать свободный желаемый ресурс. Но и это еще не все. Помимо этих двух требований мы также должны обеспечивать отсутствие голодания: например, пусть есть три процесса. Если два из них по очереди используют ресурс, а третий все время ждет, то говорят, что этот процесс «голодает». Мы должны не допускать подобных ситуаций. Четвертным требованием, выдвигаемым к механизмам взаимодействия процессов, является обеспечение отсутствия тупиков. Самым простым вариантом тупика является ситуация, при которой два процесса заблокировали свои ресурсы и нуждаются в ресурсах друг друга, из-за чего не могут продолжить исполнение и просто бесконечно долго ждут. Однако могут быть и круговые тупики, в которых процессы по кругу бесконечно ждут друг друга.
Итого, требования, выдвигаемые к механизму взаимодействия процессов, выглядят так:

1. Взаимоисключение.
2. Обеспечение прогресса.
3. Отсутствие голодания.

4. Отсутствие тупиков.

Первый подход, который появился, это решить данную проблему на аппаратном уровне. Сложности, которые у нас возникают, обусловлены вытесняющей многозадачностью, однако мы не можем от нее отказаться, т.к. она обеспечивает эффективность, справедливость и т.д. Однако мы может сделать так, чтобы процесс в своем прологе посылал ОС сигнал, благодаря которому ОС на аппаратном уровне блокировала бы прерывания этого процесса. А по завершении критической секции, в эпилоге, процесс бы посылал другой сигнал, и ОС отменяла бы блокировку прерываний. Это простое, но с тем довольно опасное решение: во-первых, мы полностью ломаем работу планировщика — как только процессу потребовался неразделяемый ресурс, мы тут же вынуждены отменить текущее планирование и позволить процессу пользоваться этим ресурсом столько, сколько ему требуется. Во-вторых, если в критической секции произойдет ошибка и процесс не дойдет до эпилога, то блокировка прерываний останется, и нельзя будет ничего сделать с таким процессом.

Такой подход получил название однопрограммного режима, и несмотря на свою опасность в большинстве ОС есть фрагменты кода, где он используется. Например, он используется при диспетчеризации (переключении) процессов, т.к. она должна быть атомарной, а если во время смены регистрового контекста произойдет прерывание, то мы потеряем целостность данных. Однако это код ядра, и он хорошо протестирован, но для всего остального аппаратное решение не годится, и надо искать программное решение. В чем сложность программного решения? Сложность заключается в том, что процессы изолированны, а значит не могут сами «договориться» об очередности использования ресурса. Для этого их необходимо использовать структуры данных внутри ОС: сначала один процесс напишет о себе какую-то информацию, а затем другой процесс прочитает ее. Такие программные решения получили название алгоритмов взаимоисключения. Далее мы рассмотрим некоторые из них. Ниже будет использоваться псевдокод, который стоит воспринимать лишь как общую идею алгоритма, а не его конкретную реализацию.

I. Замóк

```
1  shared int lock = 0;
2
3  P_i() {
4    ... // some code before
5
6    while (lock) {}; // prologue
7    lock = 1;
8
9    ... // critical section
10
11   lock = 0; // epilogue
12
13   ... // some code after
14 }
```

Здесь под `shared` подразумевается то, что к этой переменной имеют доступ несколько процессов (в реальности это реализовано через системные вызовы). Под `P_i` имеется в виду i-ый процесс, причем количество процессов может быть любым — алгоритм от этого не зависит.

В чем может быть проблема такого алгоритма? Т.к. у нас вытесняющее планирование, то в любой момент может произойти прерывание. Допустим, что процесс p_0 был прерван после выполнения строчки 6, но перед выполнением строчки 7. После этого процесс p_1 также решил зайти в критическую секцию и успешно это сделал (так как замок был «открыт»). Затем p_1 был прерван в процессе выполнения своей критической секции, и управление вернулось к p_0. Он начал свое исполнение со строчки 7, закрыл замок (который, впрочем, уже был закрыт) и также вошел в критическую секцию. Таким образом нарушилось условие взаимоисключения.

II. Строгое чередование

```
1  shared int turn = 0;
2
3  P_i() {
4    ... // some code before
5
6    while (turn != i) {}; // prologue
7
8    ... // critical section
9
10   turn = 1 - i; // epilogue
11
12   ... // some code after
13 }
```

Данный код написан для двух процессов ($i \in \{0, 1\}$), однако он может быть легко расширен и на большее число процессов. Теперь, т.к. пролог и эпилог состоят из одной команды, то не будет проблем с прерываниями и условие взаимоисключения будет соблюдено. Однако возникает проблема, связанная с условием прогресса: пусть процесс p_0 выполнил свою критическую секцию и передал «ход» процессу p_1. Допустим, что процесс p_1 в это время спит или не хочет использовать этот ресурс. Далее процессу p_0 вновь потребовался неразделяемый ресурс, однако он не может

получить его, т.к. ход ему может передать только p_1, который не собирается этого делать. В итоге мы получаем ситуацию, при которой ресурс свободен, однако в силу нашего алгоритма, никто его не использует.

III. Флаги готовности

```
1  shared int ready[2] = { 0, 0 };
2
3  P_i() {
4    ... // some code before
5
6    ready[i] = 1; // prologue
7    while (ready[1 - i]) {};
8
9    ... // critical section
10
11   ready[i] = 0; // epilogue
12
13   ... // some code after
14 }
```

Вместо того, чтобы делать одну переменную для всех процессов, появилась идея делать по одной переменной на каждый процесс. В коде выше рассматривается ситуация для $n = 2$ процессов, однако алгоритм может быть масштабирован и на большее число процессов. Проблема данного подхода заключается в потенциальной возможности тупика. Пусть p_0 выполнил строчку 6 и тут же произошло прерывание. После этого p_1 выполнил свою строчку 6 и встал в ожидание на строчке 7, т.к. p_0 уже успел поднять свой флаг готовности. После этого произошло еще одно прерывание и управление вернулось к p_0. Он тоже встал в ожидание на строчке 7, т.к. p_1 поднял свой флаг готовности. В итоге оба процесса будут бесконечно друг друга ждать — нарушается условие прогресса и условие отсутствия тупиков. Здесь стоит отметить, что если в предыдущем подходе была возможность выхода из ожидания (процесс p_1 мог когда-нибудь выполнить свою критическую секцию и передать «ход» обратно p_0), то здесь такой возможности нет. Процессы будут ждать друг друга бесконечно и ни один из них так и не выполнит свою критическую секцию.

IV. Алгоритм Петерсона

В этот момент появилась идея о том, что программного решения может и не существовать. Люди стали пытаться доказать, что требуемый алгоритм нельзя построить, однако этого не получилось сделать. Только в 1981-ом году появляется решение данной задачи, которое получает название по имени автора.

```
1  shared int ready[2] = { 0, 0 };
2  shared int turn = 0;
3
4  P_i() {
5    ... // some code before
6
7    ready[i] = 1; // prologue
8    turn = 1 - i;
9    while (ready[1 - i] && turn == 1 - i) {};
10
11   ... // critical section
12
13   ready[i] = 0; // epilogue
14
15   ... // some code after
16 }
```

Этот алгоритм так же называется алгоритмом «вежливого чередования»: в прологе, после того как процесс заявляет о своем намерении использовать ресурс, он передает «ход» другому процессу и встает в ожидание до тех пор, пока «ход» в руках другого процесса и тот претендует на использование ресурса. Данный алгоритм решает проблему взаимоисключения, обеспечивает прогресс и отсутствие тупиков. Его проблема заключается в том, что если процессов не 2, а больше, то они должны «вежливо передавать ход» по кругу, а это долго. Помимо этого условие в строчке 9 станет не таким простым, и на его вычисление потребуется время. Также проблема возникает, если рождается новый процесс, претендующий на данный ресурс: его нужно как-то встроить в круг уже существующих процессов. Итого данный алгоритм решает поставленную проблему, но его накладные расходы могут оказаться слишком большими.

V. Аппаратная поддержка взаимоисключений

Т.к. найти эффективного программного решения не получилось, то люди вернулись к идее об аппаратном решении проблемы. Как мы уже выяснили выше, полностью переходить в однопрограммный режим это опасно, однако нам это и не нужно: нам хотелось бы, чтобы (например) в алгоритме «Замóк» не происходило прерываний между строчками 6 и 7. Для этого была разработана специальная сложная инструкция процессора, которая проверяет переменную, и, если она равна нулю, то возвращает `true` и одновременно с этим (атомарно) устанавливает значение 1 в эту переменную. Если же проверяемая переменная не равна нулю, то просто возвращается `false`. Используя полученную инструкцию, можно слегка скорректировать код самого первого алгоритма и получить следующее:

```
shared int lock = 0;

P_i() {
  ... // some code before

  while (test_and_set(&lock)) {}; // prologue

  ... // critical section

  lock = 0; // epilogue

  ... // some code after
}
```

То, что мы получили, называется мьютекс (mutex). Однако на самом деле мьютекс это лишь частный случай другой, более сложной конструкции, которая называется семафор. Дело в том, что не всегда ресурс бывает только в двух состояниях («занят» или «свободен»), иногда ресурс может быть частично занят и требуется знать, на сколько именно он занят. Для этого и существует идея семафоров, предложенная Дейкстрой еще в 1965-ом году.

1.10. Лекция 24.??.??.

Мьютексы и семафоры

На прошлой лекции мы поговорили о том, что существует механизм замка́, который с аппаратной поддержкой решает поставленную задачу. Зачем же тогда нужны семафоры? Дело в том, что иногда бывает нужно, чтобы не один процесс мог использовать ресурс, а несколько. Т.е. n процессов одновременно могут использовать какой-то ресурс, а $(n+1)$-ый уже нет. Вторым примером, когда требуется более сложный механизм взаимоисключения, чем замо́к, является работа с буфером. Рассмотрим следующую ситуацию:

`command1 | command2`
`stdout` $\rightarrow$ `stdin`

Пусть первый процесс посылает байты (по одному), а второй их принимает. Тогда нам нужно обеспечить синхронизацию этих двух процессов: если первый процесс записал байт, то пока второй его не считает, первый процесс не должен записывать следующий байт — в противном случае мы просто потеряем информацию. Если же второй процесс считал байт, то он не должен продолжать считывать, а должен дождаться, пока первый процесс запишет новый байт. Ясно, что пересылать байты по одному это неудобно: нам придется много раз переключаться между процессами, открывать и закрывать замки. Для решения этой проблемы логично использовать буфер, но тогда возникает три проблемных места. Во-первых, не должно возникать ситуации, при которой одновременно один процесс пишет, а другой — читает. Ведь запись это не мгновенная ситуация, поэтому может возникнуть проблема, когда второй процесс считает данные, которые были еще не до конца записаны. Во-вторых, нам необходимо предотвратить переполнение буфера. В-третьих, мы должны предотвратить попытку чтения из пустого буфера. Последние две проблемы возникают из-за того, что планировщик ничего не знает про то, что процессы «общаются» через буфер, поэтому может возникнуть ситуация, при которой одному из процессов предоставляется больше времени исполнения, в результате чего другой процесс либо не успевает считать, либо не успевает записать.
Приведенные примеры объединяет следующее: есть некоторая константа, и пока она не превышена, то процесс может работать с ресурсом. Если же она превышается, то процесс должен остановиться. Механизм, который решает эту задачу и называется семафором. Семафор S это целая неотрицательная переменная, над которой разрешены две **атомарные** операции:

```
p(S) { // check
  while (S == 0) blocked;
  S = S - 1;
}

v(S) { // increment
  S = S + 1;
}
```

где под `blocked` может иметься в виду ожидание (wait), сон (sleep) или нечто другое. Теперь мы можем написать решение возникшей выше задачи с буфером, используя семафоры.

```
Semaphore mutex = 1;
Semaphore empty = n; // n is size of buffer
Semaphore full = 0;

produce() { // process 1
  while (true) {
    ... // produce data
    p(empty);
    p(mutex);
    ... // put data
```

```
      v(mutex);
      v(full);
    }
  }

  consume() { // process 2
    while (true) {
      p(full);
      p(mutex);
      ... // read data
      v(mutex);
      v(empty);
      ... // consume data
    }
  }
```

Теперь немного подробнее про семафор с именем `mutex`. Вообще, `mutex` это сокращение от **mut**ual **ex**clusion (взаимное исключение). Тогда замок, который мы рассматривали на прошлой лекции, тоже можно считать мьютексом. Теоретически это так, но на практике разница заключается в программно-аппаратной реализации.

1. Spinlock.

   Он похож на код, приведенный в конце прошлой лекции. Это так называемое активное ожидание (цикл бездействия), т.е. процесс будет выполнять инструкцию `test_ad_test` до тех пор, пока замок закрыт. Такой подход эффективен в случае коротких критических секций.

2. Двоичный семафор.

   Эффективен для больших критических секций: процессы, ожидающие открытия семафора, (обычно) будут отправляться в непрерываемое ожидание. Тогда процессы, вышедшие из этого ожидания, будут получать временное повышение приоритета для того, чтобы избежать инверсии приоритетов (ситуации когда высокоприоритетный процесс не может исполняться, т.к. нуждается в ресурсе, занятым низкоприоритетным процессом).

3. Мьютекс.

   Для того, чтобы избежать инверсии приоритетов, также используется наследование приоритетов: низкоприоритетный процесс временно получает тот же приоритет, что и высокоприоритетный процесс, нуждающийся в ресурсе. Однако для этого нам необходимо знать владельца семафора, поэтому он не решает проблему инверсии приоритетов. Под мьютексом же сейчас подразумевают двоичный семафор, для которого нам известен владелец, т.е. мы не просто знаем, что замок закрыт, но также знаем, кто его закрыл.

Здесь мы видим, что задача взаимоисключения частично конфликтует с задачей планирования: нам приходится повышать приоритет некоторым процессам (хотя с точки зрения планирования этого не требовалось) для того, чтобы избежать блокировки.

Тупики

Рассмотрим следующую ситуацию: пусть у нас есть два процесса p_0 и p_1, которые для своей работы используют ресурсы R_1 и R_2, причем они делают это так, что критические секции ресурсов пересекаются. Изначально процесс p_0 занял ресурс R_1, а процесс p_1 — ресурс R_2. Спустя некоторое время (не обязательно одновременно) процессу p_0 потребовался ресурс R_2, а процессу p_1 — ресурс R_1. Процессы встали в тупик: исполняться дальше они не могут, т.к. им требуются ресурсы друг друга. Отпустить ресурсы они также не могут, т.к. их критическая секция еще не завершилась. В итоге они будут бесконечно находиться в состоянии ожидания. Для иллюстрации проблемы тупиков Дейкстра предложил простой и понятный пример, получивший название «Проблема обедающих философов».
Пусть за круглым столом сидит 5 философов и перед каждым из них стоит блюдо. Между соседними философами лежит по одной вилке (итого на столе лежит 5 вилок). Для того, чтобы есть свое блюдо, философу требуются обе вилки, лежащие непосредственно рядом с ним. Философ может либо есть свое блюдо, либо будучи сытым находиться в размышлениях, либо будучи голодным ожидать освобождения вилок.
Если предположить, что процесс еды короткий, а процесс размышлений длинный, при этом философы становятся голодными в разное время (а не все одновременно), то никаких особых проблем не возникает. Сложности появляются тогда, когда возникает состояние борьбы за ресурсы (race condition) — в данном случае когда все философы приблизительно в одно и то же время становятся голодными. Далее у них есть несколько стратегий.

I. Deadlock

Пусть алгоритм действий каждого философа будет такой:

1. Дождись освобождения левой вилки и возьми ее.
2. Дождись освобождения правой вилки и возьми ее.
3. Поешь и отпусти обе вилки.

В этом случае может возникнуть ситуация при которой все философы почти одновременно возьмут левую вилку, после чего ничего не смогут делать, т.к. правая для каждого философа вилка будет недоступна, и они будут бесконечно долго ждать ее освобождения. Это называется deadlock-ом.

II. Livelock

Пусть алгоритм действий каждого философа будет такой:

1. Дождись освобождения левой вилки и возьми ее.
2. Если правая вилка занята, то отпусти левую вилку и после небольшой паузы вернись к шагу 1.
3. Если правая вилка свободна, то возьми ее, поешь и отпусти обе вилки.

В этом случае может возникнуть ситуация при которой все философы почти одновременно возьмут левую вилку, после чего (т.к. правая для каждого философа вилка будет занята) также одновременно отпустят ее. А потом снова возьмут и так далее. В итоге этот процесс теоретически может быть бесконечным. Это называется livelock-ом, здесь в отличие от deadlock-а процессы что-то делают, но эти действия непродуктивны (ни один из философов так в итоге и не поест).

III. Рандомизация

Для того, чтобы избежать проблемы livelock-а, можно попробовать использовать случайное время ожидания. В целом, это является решением, однако потенциально возможность возникновения тупика все еще остается (если случайные времена ожидания окажутся «неудачными»). Также не стоит забывать про накладные расходы на эту рандомизацию.

IV. «Официант»

Введем «официанта» (операционную систему), который будет наблюдать за столом со стороны и контролировать действия философов, запрещая или разрешая им есть согласно некоторым своим правилам.

Замечание 1.10.1. На самом деле можно обойтись и без официанта, однако в этом случае необходимо разрешить философам «смотреть» на тех, кто сидит рядом с ними. Т.е. философ должен принимать решение учитывая состояние своих соседей.

Условия возникновения тупиков

Для возникновения тупика необходимо **одновременное** выполнение четырех условий:

1. Mutual exclusion (взаимоисключение).

   Если ресурс используется одним процессом, то другой процесс не может его использовать.

2. Hold and wait (ожидание ресурса).

   Если процесс взял ресурс, он имеет право его не отдавать и при этом просить какие-либо следующие ресурсы.

3. No preemption (неперераспределяемость).

   Если мы выделили процессу некоторый ресурс, то мы не может его отобрать: процесс может вернуть его только самостоятельно.

4. Circular wait (круговое ожидание).

   Процессы встали в круговое ожидание относительно двух или более ресурсов.

Методы предотвращения тупиков

1. Ничего не делать и игнорировать проблему тупиков.
2. Пытаться предотвращать тупики. Т.е. организовывать работу ОС таким образом, чтобы перечисленные выше 4 условия никогда не выполнялись одновременно.
3. Обнаружение тупиков и восстановление после тупика. Т.е. мы не предотвращаем тупик, но имеет возможность его найти и каким-либо образом исправить.

В рамках данного курса мы не будем рассматривать алгоритмы обнаружения тупиков, т.к. они весьма трудоемки и носят скорее теоретический характер. Причем в современных ОС они практически не используются, ведь вероятность возникновения тупика весьма мала, а вот затраты на его детектирование и исправления довольно существенны. Современные ОС стараются предотвращать тупики (хотя, как мы узнаем далее, до конца это сделать невозможно), либо просто их игнорировать: да, пользователь потенциально может потерять какие-то данные, но это лучше, чем тратить половину вычислительных мощностей на детектирование и устранение тупиков. Такие большие затраты возникают в силу того, что в системе сотни процессов и тысячи ресурсов, поэтому количество возможных колец ожидания огромно. Можно пытаться находить тупики по странному поведению процессов, например, если процесс долго не отвечает, то есть подозрение на тупик. Однако понять, действительно ли там тупик или нет, весьма сложно: проще сказать пользователю о том, что процесс завис и предложить ему либо завершить его, либо еще подождать.

Однако лучше все-таки пытаться предотвращать тупики: для этого нам необходимо, чтобы хотя бы одно из условий возникновения тупика не выполнялось. Сразу стоить отметить, что универсального решения нет, однако для некоторых частных случаев такое возможно. Это еще больше сокращает вероятность появления тупика. Что же мы можем сделать с каждым из условий возникновения тупика?

1. Mutual exclusion.

   В некоторых случаях нам может помочь буферизация. Приведем простой пример с принтером: вместо того, чтобы реально печатать на принтере мы будем буферизовать данные, посылаемые процессом, и ставить их в очередь. После этого мы будем посылать процессу сигнал о том, что данные приняты, и он может освободить принтер. А когда уже очередь действительно дойдет до печати этих данных, то мы пошлем процессу еще один сигнал (либо о том, что печать успешна, либо об ошибке). Такой подход можно применять только тогда, когда нам не очень важен первый отклик от ресурса: например, если мы хотим отправлять данные по сети, то может возникнуть такая ситуация, что данные не отправятся, а мы уже сообщили процессу, что они отправлены, и он их у себя удалил.

2. Hold and wait.

   Мы можем разрешить процессу при запуске (или когда он в первый раз просит тот или иной ресурс) сразу же просить все необходимые ему ресурсы. Стоить помнить, что это скорее всего будет неэффективно с точки зрения использования ресурсов, поэтому не везде можно использовать такой подход.

3. No preemption.

   Можно разрешить ОС «отбирать» ресурсы у процессов (предварительно забуферизовав их контекст, чтобы потом можно было вернуться к этому моменту и продолжить исполнение). Однако это не всегда возможно: не у каждого процесса можно отобрать ресурс так, чтобы не нарушилась его работа.

4. Circular wait.

   Мы можем пронумеровать все процессы и ресурсы. Далее сделаем так, чтобы процесс мог брать ресурсы только с номером бóльшим, чем у него уже есть. Например, если у процесса есть 1-ый и 3-ий ресурсы, то он не может взять 2-ой ресурс, но может брать ресурсы с номером 4 и больше. Для того, чтобы процессы могли получать ресурсы с меньшим номером, мы иногда будем перенумеровывать ресурсы по кругу (сдвигая нумерацию на единичку), при этом проверяя не приводит ли это к тупику. Проблемы такого решения в накладных расходах: во-первых, не очень понятно, как пронумеровать все ресурсы, т.к. они появляются и исчезают динамически. Во-вторых, нужно тратить время на перерасчеты, сдвиги нумерации, проверки на возможность тупика и т.д.

Напоследок рассмотрим еще один пример взаимодействующих процессов, который немножко отличается от уже приведенных. Он связан с устройствами ввода-вывода и с их спецификой в том плане, что из них можно читать и на них можно писать. При этом во время записи любые другие операции должны быть запрещены (двум процессам нельзя одновременно писать, а также нельзя одновременно читать и писать). Однако несколько процессов могут одновременно читать с этих устройств. Разумеется мы можем сделать так, чтобы с этим ресурсом в один момент времени взаимодействовал только один процесс. Однако это может быть неэффективно: пусть есть некоторый ресурс, который часто и много читают, но редко в него пишут. Если все операции выстроить в очередь, то придется потратить значительно больше времени, чем если позволять параллельное чтение.
Однако если разрешить параллельное чтение, то может возникнуть проблема, получившая название «Проблема читателей и писателей». Ее суть проиллюстрирована на рис. 1.10.2.

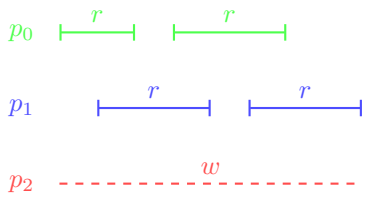

Рис. 1.10.2: Проблема читателей и писателей

Таким образом процесс p_2 не может ничего записать из-за того, что процессы p_0 и p_1 по очереди читают данные. Для того, чтобы этого избежать, создается единая очередь. Далее ищется первый запрос на запись, и все запросы на чтение, которые пришли раньше него, распараллеливаются. После окончания их работы, производится запись, и все повторяется с начала: ищется следующий запрос на запись, и все чтения до него распараллеливаются.
Такое решение будет плохо себя показывать в случае, если запросы на чтение и запись чередуются. Чтобы этого избежать, существуют некоторые подходы по пересортировке очереди так, чтобы стремиться группировать запросы на чтение (например, создавать группы фиксированного размера). Однако не всегда запросы можно переставлять местами, поэтому это лишь частное решение.

1.11. Лекция 24.??.??.

Управление памятью

Управление памятью это не менее важная задача, чем управление процессами. Причем если управление процессами определяет общую производительность вычислительного узла, то управление памятью влияет на мультипрограммность: исходя из архитектуры фон Неймана исполняемый код взаимодействует с данными в оперативной памяти. А т.к. оперативная память это конечный ресурс, то она ограничивает количество одновременно существующих процессов. Вводят даже такое понятие, как коэффициент мультипрограммности — количество процессов, которые могут существовать (но не обязательно выполняться) одновременно. Эта характеристика нас интересует, т.к. например, для веб-сервера нам важно знать, сколько запросов в единицу времени он может обработать (а это напрямую зависит от того, сколько процессов может существовать одновременно).
Относительно памяти можно сформулировать три эмпирических закона (или свойства):

1. Чем меньше время доступа к памяти, тем дороже бит такой памяти.
2. Чем выше емкость памяти, тем дешевле она обходится (в пересчете на один бит).
3. Чем выше емкость памяти, тем выше время доступа к ней.

Таким образом мы имеем либо быструю, но маленькую и дорогую, либо медленную, но большую и дешевую память. Отсюда возникает задача о распределении данных по разным видам памяти, что влияет на устройство памяти современных компьютеров и усложняет ее архитектуру относительно архитектуры фон Неймана.

Вид памяти	Регистры CPU	L1 кэш	L2 кэш	RAM	...	HDD/SSD
Объем	$\approx$ байты	$\approx 10 - 100$ Кб	$\approx$ Мб	$\approx$ Гб	...	$\approx$ Тб
Время доступа	≈ 0.1 нс	≈ 0.5 нс	≈ 5 нс	≈ 50 нс	...	≈ 10 мс

Логично, чтобы в более быстрой памяти хранилось то, к чему мы часто обращаемся. Так, например, в кэше обычно хранится таблица страниц, чтобы быстро вычислять физические адреса. Однако кэши обычно находятся под полным контролем ОС, поэтому мы будем больше говорить про RAM и HDD/SSD.
Пусть у нас есть некоторое большое приложение. Полностью хранить его в оперативной памяти это не очень выгодно: часть его кода может использоваться весьма редко. Хранить часть кода в оперативной памяти, а остальное подгружать с диска при необходимости тоже не очень удобно: нужно будет разбираться с адресацией. Для решения этой проблемы появилось решение получившее название «виртуализация». Его суть заключается в том, что для приложения выделяется отдельное адресное пространство, но часть этого пространства хранится в RAM, а часть на диске. При этом сохраняется возможность перемещать данные между разными устройствами хранения в рамках этого адресного пространства. Такая процедура обмена данных в RAM с данными на диске получила название swapping.
Далее есть несколько стратегий swapping-га: например, можно осуществлять swapping для всего адресного пространства процесса, а можно только для отдельных его фрагментов. У каждого из этих подходов есть свои преимущества и недостатки. Если swap-ать все адресное пространство, то тогда будет обеспечена целостность данных и сохранена адресация для этого процесса. С другой стороны, swap-ать память частями намного удобнее: нет необходимости копировать большие участки памяти только ради того, чтобы подгрузить пару функций. Стоит отметить, что при swapping-ге частями тоже есть свои нюансы: можно копировать частями одинакового размера (который тоже может варьироваться), а можно разного, можно копировать отдельно код и отдельно данные, а можно все вместе и т.д.
Также встает вопрос о том, как организовать доступ к данным, которые сейчас находятся на диске. В Windows это осуществляется с помощью файла подкачки, в Linux — с помощью раздела подкачки. У Windows файл подкачки это по сути обычный файл, и доступ к операциям swapping-га происходит через стандартные механизмы файловой системы. Из плюсов такого решения можно отметить то, что не нужно писать новый код (можно использовать стандартный код для работы с файлами на диске), а также то, что мы можем довольно гибко управлять размером файла подкачки. Минусом же является то, что для работы с файлом подкачки нам не нужен весь огромный инструментарий файловой системы, но используя его мы получаем некоторые накладные расходы, что негативно сказывается на производительности. Также есть некоторые проблемы с надежностью: сбой в файловой системе повлияет и на файл подкачки. Linux же предлагает другой вариант. На диске выделяется отдельный swap раздел, на котором устанавливается некоторая специфичная файловая система. Это повышает надежность, безопасность и быстродействие такого решения. Минусом же является то, что для изменения размера необходимо переразмечать диск. Это не очень сложная задача, однако сделать это в условиях, например, высоконагруженного сервера уже не так просто и связано с некоторыми рисками.
Теперь стоит поговорить про еще одну проблему: в рамках концепции виртуальной памяти нам необходимо каким-то образом пересчитывать виртуальные адреса в физические, и если до этого мы опускали этот момент, то сейчас настала пора с ним разобраться. Когда мы работаем с памятью, то имеем дело с несколькими видами адресов (рис. 1.11.1).
Для преобразования в физические адреса есть две основных стратегии: перемещающий загрузчик и динамическое преобразование. Перемещающий загрузчик пересчитывает все адреса в физические один раз при загрузке приложения. Это, конечно, увеличивает время загрузки, но с другой стороны в дальнейшем приложение будет использовать сразу физические адреса и не придется ничего пересчитывать. Также это повышает безопасность: еще на этапе загрузки можно отследить все потенциальные обращения за границы выделенной памяти. Однако все это работает только в случае, если приложению выделяется непрерывный участок памяти, который никак не будет изменяться в ходе его исполнения. Ясно, что в реальности это не так: память мало того, что может выделяться участками, так она еще

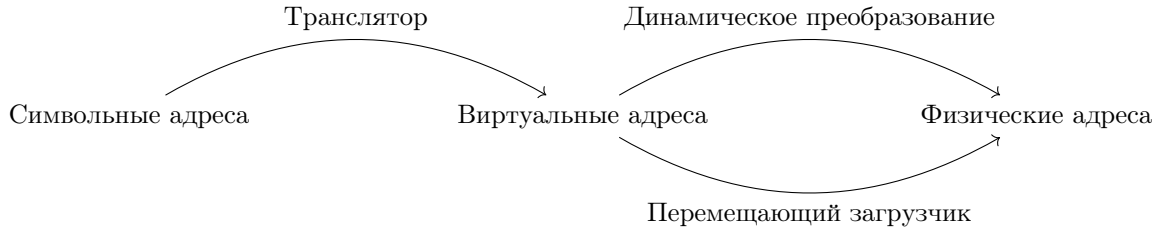

Рис. 1.11.1: Преобразование адресов

и может перемещаться в процессе работы приложения (тот же swapping, например). Из-за этого время на загрузку приложения еще больше увеличится, а также такому загрузчику придется менять адреса во всем коде приложения в случае перемещения хотя бы одной из его страниц памяти, что также очень негативно скажется на производительности. В случае фрагментированности памяти удобнее использовать динамическое преобразование, суть которого заключается в том, что физический адрес будет вычисляться только в момент обращения к переменной. Такой подход решает проблемы, которые были у перемещающего загрузчика, однако имеет накладные расходы: работа с памятью это весьма частая операция, поэтому придется много времени тратить на постоянный пересчет адресов. Итого мы имеем два граничных решения со своими плюсами и минусами, но хотим найти некий компромисс между ними. В процессе поисков этого компромисса появились разные методы выделения памяти.

Методы выделения памяти

I. Без использования внешней памяти

1. С фиксированными разделами.

   Каждому процессу на все время его существования выделяется постоянное непрерывное адресное пространство.

   (a) Одинаковые разделы.

   Сделаем так, чтобы количество разделов было k-ой степенью двойки. Тогда для того, чтобы получить из виртуального адреса физический, достаточно будет в первые k бит записать номер страницы (раздела), в котором находится исполняемое приложение. Однако отсюда следует, что виртуальное пространство в таком случае не может превышать размер одной страницы, иначе часть битов его виртуального адреса пересечется с битами, отвечающими за номер страницы в физическом адресе. Выделять несколько страниц на один процесс тоже нельзя: это затруднит (и, как следствие, замедлит) пересчет адресов. Значит нам придется подбирать размер страницы так, чтобы всем процессам хватило памяти, но при этом общее количество страниц осталось приемлемым.

   Появилось так называемое «overlay-ное программирование», в котором программист сам подгружает/отгружает код приложения из хранилища и следит за адресацией.

   (b) Разные разделы.

   Исходя из того, что все процессы разные, сделаем много маленьких разделов, несколько разделов побольше, пару разделов еще побольше и т.д. Далее для каждого процесса будем стараться подбирать подходящий ему раздел. Проблема заключается в том, что мы заранее не знаем время жизни процесса, а так же то, изменит ли он свои потребности в памяти или нет. Логично выделять процессу минимально подходящую по размеру страницу, однако здесь есть некоторые сложности.

   Проиллюстрируем это простым примером. Пусть в некоторый момент времени в системе появилось много маленьких процессов, и мы раздали им все маленькие страницы памяти, а также несколько больших страниц (потому что других свободных страниц не осталось). Далее процессы, использующие маленькие страницы, быстро завершились, а вот процессы, которые получили большие страницы, продолжают работать (допустим, это концептуально другие процессы, выполняющие иные задачи). В итоге получается, что у нас есть несколько больших страниц памяти, которые в течении длительного времени будут использоваться процессами, не нуждающимися в таком объеме памяти. Иными словами, оперативная память будет использоваться неэффективно.

   Можно попробовать решить эту проблему с помощью очередей: к каждой группе разделов одинакового размера можно сделать очередь, и если все страницы данного размера заняты, то ставить новый процесс в очередь. Однако здесь может возникнуть голодание, с которым, впрочем, можно пытаться бороться примерно также, как мы это делали в алгоритмах планирования. Помимо этого стоит помнить, что количество больших разделов ограничено, а это влияет на максимально возможное число больших процессов, которые можно породить. Также не стоит забывать, что из-за того, что разделы имеют разный размер, усложняется и замедляется работа перемещающего загрузчика. Кроме этого, мы имеем накладные расходы на хранение самих очередей, на исполнение алгоритмов по перемещению процессов между ними и т.п.

2. С динамическими разделами.

Можно попробовать поступить следующим образом: при рождении будем находить и выделять процессу столько памяти, сколько он потребовал. При таком подходе не будет проблем с тем, что (например) родилось три больших процесса, но есть всего две большие страницы памяти, а остальную память нельзя использовать (даже если она свободна), т.к. она зарезервирована под маленькие страницы. Проблема такого решения заключается в том, что со временем память фрагментируется и может возникнуть ситуация, при которой суммарно свободно много памяти, однако создать новый процесс мы не можем, т.к. нет **непрерывного** свободного участка памяти нужного размера. Таким образом получается, что чем меньше процессу нужно памяти, тем больше шансов у него родится (т.к. найти маленький непрерывный свободный участок памяти значительно проще), что несколько несправедливо.

Для решения этой проблемы логично «уплотнить» память: переместить занятые участки так, чтобы между ними не было пустых промежутков. Однако это имеет свои накладные расходы: как минимум перемещающему загрузчику нужно будет снова пересчитать все адреса. Также на все время перемещения нужно будет остановить процесс, чтобы у него не нарушилась целостность данных. Ясно, что делать подобные перемещения слишком часто это неэффективно, отсюда появляются разные стратегии перемещений.

Самая простая из них это стратегия перемещения «по удалению», т.е. после освобождения любого участка памяти мы будем сдвигать все процессы, которые были «правее» него, так, чтобы заполнить освободившееся пространство. Однако проблема этой стратегии в том, что при освобождении памяти из самого начала придется перемещать все процессы в системе, что довольно медленно.

Есть стратегия перемещения «по рождению», которая заключается в том, что пока процесс может родиться, мы выделяем ему память в произвольном месте. Любые перемещения начинаются в тот момент, когда мы уже не можем найти достаточный непрерывный участок памяти для процесса, однако всей свободной памяти хватает для его рождения. Причем в идеале мы стараемся подвинуть минимальное количество процессов так, чтобы новому процессу хватило места. Однако здесь не стоит забывать про накладные расходы на алгоритмы по расчету таких перемещений.

Две рассмотренные стратегии также имеют еще один недостаток: они непредсказуемы (неравномерны), т.е. выделение памяти может быть быстрым в течении какого-то времени, а потом резко замедлиться (например из-за того, что началось какое-то обширное перемещение). Исходя из этого появляется стратегия фонового перемещения: мы выделяем отдельный низкоприоритетный фоновый процесс, который занимается перемещением участков памяти. Причем можно делать (например) так, чтобы он перемещал только те процессы, которые сейчас спят, чтобы их остановка ни на что не влияла.

В итоге мы видим, что выделение памяти без использования внешней памяти удобно тогда, когда нет серьезной динамики в жизненном цикле процесса, т.е. процессы рождаются редко, но на длительный период времени. Также оно может быть удобно, когда мы заранее что-то знаем про процессы (например, про их количество, требования памяти и т.п.). Таким образом применение этого подхода во встроенных системах позволяет добиться значительного прироста в производительности благодаря использованию перемещающего загрузчика вместо динамического преобразования.