Оглавление

Общая архитектура системы

1.1 База стандартов Linux

(Linux Standards Base, www.linuxbase.org) представляет собой проект, ставящий целью стандартизацию самой операционной системы Linux для того, чтобы приложения, написанные для **одного дистрибутива**, соответствующего стандарту LSB, точно также компилировались и выполнялись **в любом другом LSB-совместимом дистрибутиве**.

База стандартов Linux содержит общие стандарты компонентов операционной системы, включая библиотеки, форматы пакетов и дистрибутивов, команд и утилит.

Например, стандарт LSB определяет стандартную структуру файловой системы. База стандартов Linux также включает архитектурно-зависимые стандарты, необходимые для сертификации на соответствие LSB.

Желающие протестировать и сертифицировать дистрибутив на соответствие базе стандартов Linux могут приобрести необходимые для этого

утилиты и пройти сертификацию, обратившись к организации LSB (опять-таки, -за определенную плату).

До последнего времени задача соответствия стандартам имела достаточно низкий приоритет для разработчиков ядра, поскольку в первую очередь они стремились расширить функциональные возможности и надежность работы Linux. Вследствие этого, большинство версий ядра не согласуются ни с одним из наборов стандартов.

1.2 Архитектура системы

Хотя **ядро Linux является монолитным** по своей природе, последние усовершенствования, внесенные для обеспечения масштабируемости ядра, включают в себя возможности модульности, подобные тем, которые поддерживаются операционными системами с микроядром. Linux часто называют UNIX-подобной операционной системой, поскольку в ней существуют многие службы, характерные для таких UNIX-систем, как UNIX System V от компании AT&T или BSD, разработанной в Беркли.

Ядро Linux состоит из шести основных подсистем:

- **подсистема управления процессами,**
- **подсистема взаимодействия между процессами,**
- **подсистема управления памятью,**
- **подсистема управления файловой системой,**
- **подсистема управления операциями ввода/вывода и**
- **сетевая подсистема.**

Все шесть подсистем имеют доступ к системным ресурсам.

Процессы в Linux могут выполняться в режиме ядра либо пользовательском режиме.

Пользовательские процессы выполняются в пользовательском режиме, поэтому доступ к службам ядра они получают через интерфейс системных вызовов.

Когда от пользовательского процесса поступает разрешенный системный вызов (в пользовательском режиме), **ядро обрабатывает системный вызов в режиме ядра от имени процесса пользователя.**

Если запрос пользователя некорректен (например, процесс пытается осуществить запись в файл, который не был открыт), ядро возвратит пользовательскому процессу код ошибки.

1.3 Основные подсистемы ядра

Диспетчер процессов (process manager) является главной подсистемой Linux, отвечающей за создание процессов, обеспечение доступа к процессору (процессорам) системы и удалению процессов из системы по завершению их работы.

Подсистема взаимодействия процессов (interprocess communication, IPC) ядра включает средства взаимодействия процессов. Эта подсистема работает совместно с диспетчером процессов, обеспечивая совместный доступ к информации и передачу сообщений с помощью разнообразных методов.

Подсистема управления памятью обеспечивает процессам доступ к памяти. Linux выделяет каждому процессу виртуальное адресное пространство, которое делится на **пользовательское адресное пространство и пространство ядра.**

Включение пространства ядра в контекст каждого процесса уменьшает затраты на переключение между режимами ядра и пользователя, поскольку ядро может получить доступ к своим данным из виртуального адресного пространства любого процесса.

Пользователи получают доступ к файлам и каталогам, перемещаясь по единому дереву каталогов. Корень дерева каталогов называется **корневым (root) каталогом.**

Из корневого каталога пользователи могут перейти к любой доступной файловой системе.

Пользовательские процессы запрашивают данные файловой системы через интерфейс системных вызовов.

Когда системе нужен доступ к файлу или каталогу дерева каталогов, взаимодействие осуществляется через **интерфейс виртуальной файловой системы (virtual file system, VFS)**, обеспечивающий единый способ доступа ко всем файлам и каталогам, размещенным в **неоднородных файловых системах** (например, ext2 и NFS). Виртуальная файловая система передает запросы конкретной файловой системе, которая отвечает за схему размещения и место хранения данных.

Основываясь на модели UNIX, операционная система Linux взаимодействует с устройствами, как с файлами, то есть использует те же механизмы доступа к данным, что и при работе с файлами.

Когда пользовательские процессы обмениваются данными с устройством, ядро передает запросы интерфейсу виртуальной файловой системы, которая перенаправляет их интерфейсу ввода/вывода. Интерфейс ввода/вывода передает запросы далее, драйверам устройств, выполняющим операции ввода/вывода для системного оборудования.

В Linux существует **сетевая подсистема**, позволяющая процессам обмениваться данными с компьютерами по сети. Для отправки и приема пакетов сетевая подсистема использует сетевое оборудование системы, получая доступ к нему через интерфейс ввода/вывода.

Сетевая подсистема позволяет приложениям и ядру инспектировать и модифицировать пакеты, проходящие по сетевым уровням системы с помощью интерфейса фильтрации пакетов.

Данный интерфейс позволяет реализовать **брандмауэры, маршрутизаторы и другие сетевые средства.**

2 Управление процессами и потоками

Подсистема управления процессами важна для обеспечения эффективной реализации многозадачного режима в Linux.

Хотя в первую очередь она отвечает за выделение процессоров для обслуживания процессов, подсистема управления процессами также занимается передачей сигналов, загрузкой модулей ядра и приемом прерываний.

2.1 Организация процессов и потоков

В системе Linux и процессы, и потоки называют задачами (**task**).

Изнутри они представляют собой единую структуру данных.

При загрузке ядра обычно запускается процесс **init**, который использует ядро для создания всех остальных задач.

Задачи создаются путем вызова системной функции **clone.**

Любые обращения к fork или vfork преобразуются в системные вызовы clone во время компиляции.

Функция **fork** создает дочернюю задачу, виртуальная память для которой выделяется по принципу копирования при записи (**copy-on-write**).

Когда дочерний или родительский процесс пытается выполнить запись в страницу памяти, записывающая программа создает собственную копию страницы в памяти.

Копирование при записи может привести к снижению быстродействия в том случае, когда процесс использует процедуру **execve** для загрузки новой программы сразу после **fork**.

Например, если родительский процесс начнет свое выполнение до запуска дочернего процесса, копирование при записи будет осуществляться при модификации родительским процессом любой страницы памяти.

Поскольку дочерний процесс не использует родительские страницы (если в начале работы им была сразу вызвана функция **execve**), данная операция абсолютно бесполезна и только приводит к увеличению накладных расходов.

Поэтому в Linux поддерживается вызов функции **vfork**, позволяющей повысить быстродействие при вызове дочерними процессами процедуры **execve**.

Процедура **vfork** приостанавливает работу родительского процесса в том случае, когда дочерний процесс вызывает функции **execve** или **exit**, чтобы

обеспечить загрузку дочерним процессом новых страниц до того, как родительский процесс начнет выполнять бесполезные операции копирования при записи.

Процедура **vfork** позволяет еще больше повысить производительность благодаря тому, что при ее вызове таблицы страниц родительского процесса не копируются в дочерний, поскольку новые таблицы страниц создаются тогда, когда дочерний процесс вызывает функцию **execve**.

2.2 Потоки Linux и системные вызовы clone

Поддержка потоков в Linux организована при помощи системного вызова процедуры **clone**, позволяющей вызывающему процессу задавать общий доступ к виртуальной памяти для потока, информацию о файловых системах, файловые дескрипторы и/или обработчики сигналов.

Регистры процессора, стек и других данных, индивидуальны и локальны по отношению к потокам (TSD), тогда как адресное пространство и открытые дескрипторы файлов по отношению к потокам представляются глобальными в их процессе.

Хотя **clone** и может создавать поток; эта процедура не полностью соответствует спецификациям POSIX по потокам.

Например, несколько потоков, которые были созданы путем вызова системной процедуры **clone** с настройками максимального разделения ресурсов, в то же время могут поддерживать ряд структур данных, не являющихся разделяемыми между всеми потоками процесса, таких как структуры прав доступа.

При вызове процедуры **clone** из процесса ядра (т.е. процесса, выполняющего программный код ядра), создается поток ядра (kernel thread), отличающийся от остальных потоков тем, что он может обращаться непосредственно к адресному пространству ядра.

В ядре в виде потоков реализованы несколько демонов. Демонами называются службы, пребывают в спящем режиме до тех пор, пока ядро не

разбудит их для выполнения таких задач, как сохранение страниц либо планирование программных прерываний. Эти задачи обычно связаны с обслуживанием и потому выполняются регулярно.

Способ реализации потоков в Linux характеризуется рядом преимуществ.

- Например, потоки Linux позволяют упростить код ядра и уменьшить накладные расходы, поскольку им хватает единственной копии структуры данных для управления задачами.
- Более того, хотя потоки Linux не обладают такой же переносимостью, как потоки POSIX, они предоставляют программистам возможность более гибко и тонко управлять разделением ресурсов между задачами.

2.3 Реализация потоков Linux

Операционная система Linux поддерживает потоки в ядре довольно интересным способом. В основе реализации системы Linux лежат идеи из системы 4.4BSD

Сердцем реализации потоков в системе Linux является новый системный вызов **clone**, отсутствующий во всех остальных версиях системы UNIX.

Формат обращения к нему выглядит следующим образом:

pid = clone(function, stack_ptr. sharing_flags. arg);

Системный вызов clone создает новый поток либо в текущем процессе, либо в новом процессе, в зависимости от флага sharing_flags.

Если новый поток находится в текущем процессе, он совместно использует с остальными потоками адресное пространство и любое изменение каждого байта в адресном пространстве любым потоком тут же становится видимым всем остальным потокам процесса.

С другой стороны, если адресное пространство не используется совместно, тогда новый поток получает точную копию адресного пространства, но последующие изменения в памяти уже не видны остальным потокам. Таким образом, здесь используется та же семантика, что и у системного вызова **fork.**

В обоих случаях новый поток начинает выполнение функции **function** с аргументом **arg** в качестве параметра. Также в обоих случаях новый поток получает свой собственный стек, при этом указатель стека инициализируется параметром **stack_ptr**.

Параметр **sharing_flags** представляет собой битовый массив, обеспечивающий существенно более тонкую настройку совместного использования, нежели используется в традиционных системах **UNIX**.

У этого флага определены пять битов, перечисленные в таблице. Каждый бит управляет одним из аспектов совместного использования, и каждый из битов может быть установлен независимо от остальных битов.

Флаг	Значение в 1	Значение в 0
CLONE_VM	Создать новый поток	Создать новый процесс
CLONE_FS	Общие рабочий каталог, каталог root и umask	Не использовать их совместно
CLONE_FILES	Общие дескрипторы файлов	Копировать дескрипторы файлов
CLONE_SIGHAND	Общая таблица обработчика сигналов	Копировать таблицу
CLONE_PID	Новый поток получает старый PID	Новый поток получает новый PID

Биты массива **sharing_flags**

Бит **CLONE_VM** определяет, будет ли виртуальная память (то есть адресное пространство) использоваться совместно со старыми потоками или будет копироваться.

Если этот бит установлен, новый поток просто помещается вместе со старыми потоками, так что системный вызов **clone** создает новый поток в существующем процессе.

Если бит сброшен, новый поток получает свое собственное адресное пространство. Это означает, что результат команды процессора **STORE** не виден остальным потокам. Такое поведение подобно поведению системного вызова **fork**. Создание нового адресного пространства равнозначно определению нового процесса.

Бит **CLONE_FS** управляет совместным использованием рабочего каталога и каталога **root**, а также флага **umask**. Даже если у нового потока свое

собственное адресное пространство, при установленном бите **CLONE_FS** старый и новый потоки будут совместно использовать рабочие каталоги. Это означает, что обращение к системному вызову с**hdir** одним из потоков изменит рабочий каталог другого потока, несмотря па то что у другого потока есть свое собственное адресное пространство.

В системе UNIX обращение к системному вызову **chdi**r потоком всегда изменяет рабочий каталог всех остальных потоков этого процесса, но никогда не меняет рабочих каталогов других процессов. Таким образом, этот бит обеспечивает разновидность совместного использования, недоступную в UNIX.

Бит **CLONE_FILES** аналогичен биту **CLONE_FS**. Если он установлен, то новый поток пользуется теми же дескрипторами файлов, что и старые потоки. Таким образом, обращение к системному вызову **1seek** одним потоком становится видимым для других потоков, что также обычно справедливо для потоков одного процесса, но не для потоков различных процессов.

Аналогично бит **CLONE_SIGHAND** разрешает или запрещает совместное использование таблицы обработчиков сигналов старым и новым потоками. Если таблица общая даже у потоков в различных адресных пространствах, тогда изменение обработчика в одном потоке повлияет и на другой поток.

Наконец, бит **CLONE_PID** указывает, получит ли новый поток свой собственный PID или будет использовать PID своего родительского потока. Это свойство нужно при загрузке системы. Процессам пользователя не разрешается использовать этот бит.

Такая детализация в вопросе совместного использования стала возможна благодаря тому, что в системе Linux для различных вопросов используются различные структуры данных.

Таблица процессов и структура пользователя просто содержат указатели на эти структуры данных, поэтому легко создать новый элемент таблицы для каждого клонированного потока и сделать так, чтобы он указывал либо на старую структуру, управляющую планированием потоков, памятью или еще чем-либо, либо на копию такой структуры.

2.4 Планирование потоков

2.4.1 Очереди планировщика

В состав подсистемы управления процессами входит **планировщик процессов** (process scheduler), обеспечивающий доступ процессов к процессору в течение нужных промежутков времени.

Диспетчер процессов хранит список всех задач в виде **двух структур данных.**

Первая структура представляет собой кольцевой список, каждая запись которого содержит указатели на предыдущую и последующую задачу (очередь выполнения (**run queue**). Обращение к этой структуре происходит в том случае, когда ядру необходимо проанализировать все задачи, которые должны быть выполнены в системе.

Второй структурой является хэш-таблица. При создании задачи ей присваивается уникальный идентификатор процесса (process identifier, PID). Идентификаторы процессов передаются хэш-функции для определения местоположения процесса в таблице процессов. Хэш-метод обеспечивает быстрый доступ к специфическим структурам данных задачи, если ядру известен ее PID.

Каждая задача таблицы процессов представляется в виде структуры **task_struct**, служащей в роли **дескриптора** процесса (т.е. **блока управления процессором (PCB).**

В структуре **task_struct** хранятся переменные и вложенные структуры, описывающие процесс.

Например, в переменной **state** содержатся сведения о текущем состоянии задачи. [Первоначально ядро писалось на языке С, поэтому для представления программных сущностей в нем широко применяются структуры.]

Другие важные переменные, специфические для определенной задачи, позволяют планировщику вычислять время выполнения на процессоре.

Эти переменные определяют приоритеты задач, необходимость выполнения в режиме реального времени, и то, какой алгоритм планирования реального времени должен использоваться в этом случае.

Структуры, являющиеся вложенными по отношению к структуре **task_struct**, могут содержать дополнительные сведения о задаче.

Одна из подобных структур, **mm_struct**, описывает выделяемую для задачи память. (например, местоположение таблицы страниц в памяти и число задач, совместно использующих адресное пространство).

Дополнительные структуры, являющиеся вложенными по отношению к структуре **task_struct**, содержат такую информацию, как значения регистров, хранящие контекст выполнения задачи, обработчики сигналов и права доступа для задачи.

Обращение к этим структурам осуществляется с помощью нескольких подсистем ядра помимо диспетчера процессов.

2.4.2 Граф состояний потока

Задача переходит в состояние **running (выполнения)** после того, как её будет выделен процессор

При блокировке задача переходит в состояние **sleeping (спячки)**, а при приостановке работы в состояние останов **(stopped)**.

Состояние **zombie (зомби)** показывает, что выполнение задачи закончилось, однако она еще не была удалена из системы.

Например, если процесс состоит из нескольких потоков, он будет пребывать в состоянии зомби, пока все потоки не получат уведомление о завершении работы основного процесса.

Задача в состоянии **dead (смерти)** может быть удалена из системы.

Состояния **active (активный)** и **expired (неактивный)** используются при планировании выполнения процесса, и поэтому они не сохраняются в переменной **state.**

2.4.3 Диспетчеризация потоков

Основной целью планировщика процессов Linux является выделение для выполнения каждой задаче необходимых временных интервалов, с учетом приоритета того или иного процесса, пытаясь достичь при этом максимальной эффективности использования ресурсов, высокого быстродействия и снизить накладные расходы, связанные с операциями планирования.

Поскольку планировщик допускает **приоритетное вытеснение**, каждая задача выполняется до тех пор, пока не истечет выделенный ей **квант времени**, управление не будет передано процессу с более высоким приоритетом, либо процесс не будет заблокирован.

Квант времени для каждой задачи рассчитывается исходя из ее приоритета (за исключением задач реального времени).

Чтобы запретить выделение чересчур малых квантов времени, уменьшающих производительность задачи, либо слишком больших квантов времени, приводящих к увеличению времени реакции системы, планировщик выделяет кванты времени в диапазоне от **10** до **200** интервалов прерываний таймера, что соответствует 10-200 миллисекундам на большинстве систем (как и многие другие параметры планировщика, эти значения подбирались эмпирически).

Когда выполнение задачи приостанавливается согласно принципу приоритетного вытеснения, планировщик сохраняет состояние задачи в структуре **task_struct**.

По истечении кванта времени, выделенного процессу, планировщик выполняет пересчет приоритета процесса, определяя величину следующего кванта времени для задачи, и передает управление следующему процессу.

2.4.4 Очередь выполнения

После создания задачи с помощью **clone**, она помещается в **очередь выполнения** (run queue) процессора, содержащую ссылки на все задачи, состязающиеся за процессорное время.

Очереди выполнения, напоминают многоуровневые очереди с обратной связью, позволяют присваивать задачам различные приоритеты.

Массив приоритетов (**priority array**) содержит указатели на отдельные уровни очереди выполнения.

Каждая запись массива приоритетов ссылается на список задач: задача с приоритетом i помещается в i-ю ячейку массива приоритетов очереди выполнения.

Планировщик помещает задачу в начало списка на самом высоком уровне массива приоритетов. Если на этом уровне массива приоритетов существует несколько задач, они циклически упорядочены.

Когда задача переходит в состояние блокировки либо спячки (т.е. ожидания), или же ее выполнение прекращается по какой-либо иной причине, задача удаляется из очереди выполнения.

Одной из целей планировщика является предотвращение ситуации бесконечного откладывания путем задания временных интервалов, называемых **периодами дискретизации (epoch),** в течение которых каждая задача из очереди выполнения должна быть запущена хотя бы раз.

Чтобы отличить процессы, обладающие правом на процессорное время, от процессов, которые вынуждены ожидать до наступления следующего периода дискретизации, в планировщике определены два состояния: **активный (active)** и **неактивный (expired)**. При этом планировщик осуществляет диспетчеризацию только тех процессов, которые находятся в активном состоянии.

Продолжительность периода дискретизации определяется исходя из **времени ожидания зависшего процесса** (starvation limit) — выбираемого эмпирически значения, которое позволяет задачам с высоким приоритетом добиваться быстрой реакции, а задачам с низким приоритетом — получать достаточно времени для выполнения продуктивной работы за разумные временные промежутки.

По умолчанию, время ожидания зависшего процесса равно **10*n** секунд, где **n** — это количество процессов в очереди выполнения.

Если текущий период дискретизации больше, чем время ожидания зависшего процесса, планировщик переводит все активные задачи из очереди выполнения в состояние ***expired*** (переход осуществляется после того, как истекут кванты времени, выделенные для выполнения каждой активной задачи).

В результате выполнение высокоприоритетных задач будет временно отложено (за исключением задач с приоритетом реального времени), чтобы дать возможность поработать процессам с низким приоритетом.

После того как все задачи в очереди выполнения будут запущены хотя бы раз, все они перейдут в неактивное состояние (***expired***). На этом этапе планировщик переведет все задачи из очереди выполнения назад в активное состояние, после чего начнется новый период дискретизации.

Для упрощения перехода из состояния **expired** в состояние **active** в конце периода дискретизации, планировщик Linux хранит для каждого процессора по **два массива приоритетов.**

Массив приоритетов, в котором хранятся активные задачи, называется **активным списком (active list)**.

Массив приоритетов, содержащий просроченные или неактивные задачи (т.е. задачи, выполнение которых отложено до наступления следующего периода дискретизации) называется **неактивным списком (expired list).**

Когда задача переходит из активного в неактивное состояние, она помещается в неактивный список на том же уровне массива приоритетов, на котором задача находилась в активном списке.

В конце каждого периода дискретизации, все задачи оказываются в неактивном состоянии, после чего они должны перейти в активное состояние.

Планировщик выполняет эту задачу очень быстро, меняя между собой указатели на список активных и неактивных задач.

Благодаря использованию двух массивов приоритетов для каждого процесса, планировщик может переводить все задачи из одного состояния в другое с помощью простой операции смены указателей.

Получаемый за счет этого прирост производительности обычно компенсирует расходы, связанные с выделением дополнительной памяти.

При работе в многопроцессорной системе планировщик Linux создает по одной очереди выполнения для каждого процессора. Одной из причин, оправдывающей создание очередей выполнения для каждого процессора является возможность структурной привязки процесса к определенному процессору (**processor affinity**).

В некоторых многопроцессорных архитектурах, таких как NUMA, высокое быстродействие процессов обеспечивается за счет размещения данных задачи в локальной памяти процессора и в его кэше. Поэтому максимального быстродействия может достичь задача, выполнение которой постоянно связано с одним и тем же процессором (или узлом). С другой стороны, использование отдельных очередей выполнения для каждого процессора чревато неравномерностью загрузки процессоров, что может привести к снижению производительности системы в целом.

2.4.5 Приоритет при планировании

В планировщике Linux приоритет задачи влияет на размер кванта времени и порядок выполнения задач процессором.

Во время создания каждой задаче присваивается статический приоритет **(static priority**), называемый также правильным значением (**nice value**).

Планировщик различает **40** различных уровней приоритета: от **-20** до **19**.

В соответствии с конвенцией UNIX наименьшее значение означает наибольший приоритет в алгоритме планирования (т.е. **-20 — это самый высокий приоритет. который может иметь процесс**).

Одной из целей планировщика Linux является обеспечение высокой степени интерактивности системы.

Поскольку обычно интерактивные задачи блокируют выполнение операций ввода/вывода либо переходят в спящий режим (ожидая реакции от пользователя), планировщик может динамически увеличивать значение приоритета (понижая уровень приоритета) задачи, занимающей процессорное время, до того, как истечет выделенный ей квант времени.

Такой подход вполне приемлем, поскольку процессы, связанные с вводом/выводом, обычно мало используют процессор, за исключением момента генерирования запроса ввода/вывода.

Поэтому предоставление высокого приоритета задачам, связанным с вводов/выводом информации, не повлияет на выполнение задач, интенсивно эксплуатирующих процессор, которые могут использовать его в течение многих часов, при условии доступности даже без приоритетного вытеснения.

Измененный уровень приоритета называют **эффективным приоритетом** (**effective priority**) задачи, вычисляемым во время пребывания задачи в спящем состоянии либо в рамках выделенного ей кванта времени.

Эффективный приоритет определяет уровень в массиве приоритетов, на который будет помещена данная задача. Поэтому задача, величина приоритета которой увеличилась, помещается на более низкий уровень массива приоритетов, то есть она будет выполнена раньше задачи с большим значением эффективного приоритета.

Ради дальнейшего повышения интерактивности, планировщик штрафует задачи, сильно загружающие процессор, увеличивая статическое значение их приоритета. В результате, задачи, выполнение которых сильно загружает процессор, помещаются на более высокий уровень массива приоритетов, за счет чего задачи с меньшим значением эффективного приоритета выполняются раньше.

Опять-таки, данная мера мало влияет на задачи, выполнение которых сильно загружает процессор, потому что интерактивные задачи с высоким приоритетом выполняются в течение коротких промежутков времени перед блокированием.

Чтобы обеспечить выполнение задачи с близким к заданному начальному значению приоритетом, планировщик задач запрещает устанавливать эффективный приоритет задачи, отличающийся от статического более чем на пять единиц.

Таким образом планировщик проявляет уважение к уровням приоритета, назначенным задачам во время их создания.

2.4.6 Операции планирования

Планировщик удаляет задачи из процессора в том случае, когда выполнение задачи прерывается, происходит ее приоритетное вытеснение (например, по окончании выделенного кванта времени) либо при блокировании задачи.

Каждый раз при удалении задачи из процессора, планировать вычисляет для нее следующий квант времени.

В случае блокирования задачи либо невозможности ее выполнения по иной причине, она деактивируется (**deactivate**), то есть удаляется из очереди выполнения до тех пор, пока. не будет снова готова к выполнению.

Во всех остальных случаях, определением списка, в который должна быть помещена задача (активный или неактивный) занимается планировщик.

Используемый для этого алгоритм был подобран опытным путем для обеспечения высокого быстродействия.

Основными параметрами данного алгоритма являются статические и эффективные приоритеты.

В общем случае задача с высоким и/или со значительным приростом приоритета подлежит перепланированию. Это позволяет запускать задачи, связанные с вводом/выводом, а также высокоприоритетные и интерактивные задачи по нескольку раз течение периода дискретизации.

Задачи с низким приоритетом либо оштрафованные задачи, попадающие в незаштрихованную область, помещаются в неактивный список.

При использовании пользовательским процессом системного вызова **clone** кажется оправданным выделение для каждого дочернего процесса собственного кванта времени.

Однако если задача породит много новых дочерних процессов, и каждому дочернему процессу будет выделен отдельный квант времени, другие задачи в системе могут пострадать от ухудшения времени реакции в текущем периоде дискретизации.

Для обеспечения равноправия планировщик требует, чтобы изначально каждый родительский процесс использовал один и тот же квант времени совместно с дочерним процессом, созданным при помощи clone. С этой целью планировщик отдает половину кванта времени родительскому процессу, а половину - созданному им дочернему процессу.

Чтобы предотвратить низкий уровень обслуживания легитимного процесса из-за порождения множества дочерних процессов, вышеупомянутое ограничение действует только в течение оставшейся части периода дискретизации, во время которого был создан дочерний процесс.

2.4.7 Многопроцессорное планирование

Поскольку планировщик процессов управляет задачами с помощью отдельных для каждого процессора очередей выполнения, задачи, как правило, являются структурно связанными с определенным процессором.

Это означает высокую вероятность отправки процесса в последующих периодах дискретизации на тот же самый процессор, что повышает быстродействие процесса, если его данные и инструкции все еще находятся в процессорном кэше.

Однако такая схема может привести к простою одного или нескольких процессоров в многопроцессорной системе даже в то время, когда система испытывает серьезную нагрузку. Во избежание подобной ситуации в случае обнаружения простоя процесса, планировщик осуществляет **балансировку**

загрузки (load balancing) для переноса задач с одного процессора на другой с целью повышения эффективности использована ресурсов.

Если система состоит из одного единственного процессора, подпрограммы балансировки загрузки не включаются в ядро во время его компиляции.

Планировщик определяет необходимость выполнения процедуры балансировки загрузки каждый раз после прихода прерывания системного таймера, генерируемого с периодом одна миллисекунда на системах IA-32.

Если процессор, породивший прерывание таймера простаивает (т.е. его очередь выполнения пуста), планировщик попытается выполнить миграцию задачи с процессора с наибольшей загрузкой (т.е. с процессора, в очереди выполнения которого содержится наибольшее количество процессов) на простаивающий процессор.

Чтобы снизить накладные расходы в том случае, когда процессор, породивший прерывание, не является простаивающим, планировщик будет пытаться перенести выполнение задачи на этот процессор через каждые 200 системных прерываний.

Определение загрузки процессора выполняется планировщиком на основе данных о средней длине каждой очереди выполнения в течение нескольких последних прерываний таймера, чтобы минимизировать эффект непостоянства процессорной загрузки в алгоритме балансировки.

Поскольку загрузка процессора имеет тенденцию к резким изменениям, то целью балансировки загрузки является не полное выравнивание длины двух очередей выполнения, а уменьшение различий между количеством задач в каждой очереди выполнения. В результате задачи удаляются из более длинной очереди выполнения до тех пор, пока разница между длиной двух очередей не сократится вдвое. Для уменьшения накладных расходов алгоритм балансировки загрузки запускается только тогда, когда самая длинная очередь выполнения содержит на 25% больше задач, чем очередь процессора, выполняющего балансировку загрузки.

Когда планировщик выбирает задачи для выравнивания загрузки, он старается выбрать такие процессы, быстродействие которых меньше всего пострадает от переноса на другой процессор. Высока вероятность того, что данные и инструкции задачи, которая выполнялась на процессоре раньше других, успели подвергнуться удалению из процессорного кэша (**слабо-кэшированная задача** (cache-cold process)), тогда как у **сильно-кэшированной задачи** (cache-hot process) в процессорном кэше находятся все (или почти все) данные. Поэтому планировщик выбирает для переноса те задачи, которые дольше всего не выполнялись.

2.4.8 Планирование реального времени

Планировщик поддерживает жесткое планирование реального времени, пытаясь минимизировать время, затрачиваемое задачей реального времени на ожидание отправки в процессор. В отличие от обычной задачи, которая, в конце концов, помещается в список неактивных (чтобы предотвратить бесконечное откладывание выполнения процессов с низким приоритетом), задача реального времени всегда помещается в активный список по истечению выделенного ей кванта времени.

Кроме того, задачи реального времени всегда выполняются с более высоким приоритетом, чем обычные задачи. Поскольку планировщик всегда отправляет на процессор задачи из очереди с самым высоким приоритетом активного списка (а задачи реального времени всегда находятся в активном списке), обычная задача никогда не сможет вытеснить задачу реального времени.

Планировщик согласуется со спецификациями POSIX для процессов реального времени, определяя расписание выполнения задач реального времени с помощью вышеописанного стандартного алгоритма планирована либо циклических алгоритмов и алгоритмов FIFO.

Если к определенной задаче применяется циклическое планирование, то по окончанию выделенного ей кванта времени, задача получает в свое

распоряжение новый квант времени, после чего она помещается в конец массива приоритетов активного списка.

В алгоритме FIFO задаче не выделяется квант времена. поэтому она выполняется на процессоре до тех пор, пока не будет выполнен выход из нее, задача не перейдет в спящий режим либо ее выполнена не будет прервано.

Очевидно, что безграмотно написанные процессы реального времени могут вызвать бесконечное откладывание выполнения одних процессов и медленную реакцию других.

Чтобы предотвратить случайное либо злонамеренное использование задач реального времени, право на их создание имеют только пользователи с привилегиями **root**.

3 Управление памятью в Linux

Каждый процесс системы Linux на 32-разрядной машине получает 3 Гбайта **виртуального адресного пространства для себя**, и 1 Гбайтм памяти для страничных таблиц и других данных ядра.

Один гигабайт ядра не виден в пользовательском режиме, но становится доступным, когда процесс переключается в режим ядра.

Адресное пространство создается при создании процесса и перезаписывается системным вызовом **exec**.

Виртуальное адресное пространство делится на однородные непрерывные области, выровненные по границам страниц.

Таким образом, **каждая область состоит из набора соседних страниц с одинаковым режимом зашиты и одинаковыми свойствами подкачки.**

Примерами областей являются **текстовый сегмент и файлы**, отображенные на память. Между областями в виртуальном адресном пространстве могут быть свободные участки.

Любое обращение процесса к памяти в этих свободных участках приводит к фатальному страничному прерыванию.

Размер страницы фиксирован, например 4 Кбайт для процессора Pentium и 8 Кбайт для процессора Alpha.

Каждая область описывается в ядре записью ***vm_area_struct***.

Все структуры ***vm_area_struct*** одного процесса связаны вместе в список, отсортированный по виртуальным адресам, что позволяет быстро находить все страницы.

Когда список становится слишком длинным (более 32 записей), создается дерево для ускорения поиска.

Запись ***vm_area_struct*** перечисляет свойства области.

К ним относятся

- режим защиты (например, только чтение или чтение/запись),
- является ли данная область фиксированной в памяти (невыгружаемой),
- и направление, в котором область может расти (вверх для сегментов данных, вниз для сегмента стека).

Структура ***vm_area_struct*** также содержит данные о том, является ли данная область приватной областью процесса или ее совместно используют несколько процессов.

После системного вызова **fork** система Linux создает копию списка областей для дочернего процесса, но у дочернего и родительского процессов оказываются указатели на одни и те же таблицы страниц. Области помечаются как доступные для чтения/записи, но страницы доступны только для чтения. Если любой из процессов пытается записать данные в такую страницу, происходит прерывание, ядро видит, что область логически доступна для записи, а страница недоступна, поэтому оно дает процессу копию страницы, которую помечает как доступную для чтения/записи. Таким образом реализован механизм **копирования при записи**.

Кроме того, в структуре ***vm_area_struct*** записано, есть ли у этой области памяти место хранения на диске, и если да, то где оно расположено.

Текстовые сегменты в качестве резервного хранения используют двоичные файлы, а отображаемые на адресное пространство памяти файлы выгружаются на диск в соответствующие им файлы. Всем остальным областям, таким как область стека, не назначаются области резервного хранения, пока не потребуется их выгрузка на диск,

В системе Linux используется трехуровневая схема страничной подкачки. Хотя эта схема была реализована в системе для процессора Alpha, она также используется (в упрощенном виде) для всех архитектур.

Каждый виртуальный адрес разбивается на четыре поля.

Каталоговое поле используется как индекс в глобальном каталоге, в котором есть личный каталог для каждого процесса. Содержание элемента каталога является указателем на одну из средних страничных таблиц, которые тоже проиндексированы полем виртуального адреса.

Элемент средней таблицы указывает на таблицу страниц, также проиндексированную полем страницы виртуального адреса.

Элемент в последней таблице содержит указатель на нужную страницу.

На компьютерах с процессором Pentium используется только двухуровневая организация страниц. В этом случае каждый средний страничный каталог содержит только одну запись. Таким образом, глобальный каталог фактически указывает на таблицу страниц.

Физическая память используется для различных целей. Само ядро жестко фиксировано. Ни одна его часть не выгружается на диск. Остальная часть памяти доступна для страниц пользователей, буферного кэша, используемого файловой системой, страничного кэша и других задач.

Буферный кэш содержит блоки файлов, которые были недавно считаны или были считаны заранее в надежде на то, что они скоро могут понадобиться. Его размер динамически меняется. Буферный кэш состязается за место в памяти со страницами пользователей.

Страничный кэш в действительности не является настоящим отдельным кэшем, а представляет собой просто набор страниц пользователя, которые более

не нужны и ожидают выгрузки на диск. Если страница, находящаяся в страничном кэше, потребуется снова, прежде чем она будет удалена из памяти, ее можно быстро объявить находящейся в памяти.

Кроме этого, операционная система Linux поддерживает динамически загружаемые модули, в основном драйверы устройств. Они могут быть произвольного размера и каждому из них должен быть выделен непрерывный участок в памяти ядра. Для выполнения всех этих требований система Linux управляет памятью таким образом, что она может получить по желанию участок памяти произвольного размера. Для этого используется алгоритм, известный как **«дружественный» алгоритм.**

Основная идея управления блоками памяти заключается в следующем. Изначально память состоит из **единого непрерывного участка**.

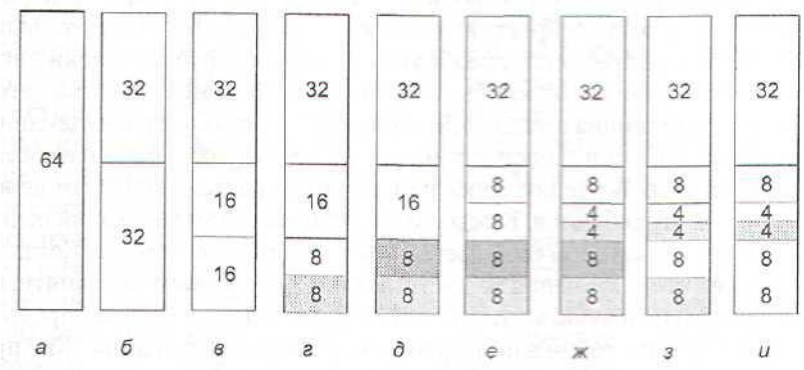

Рис. Этапы работы дружественного алгоритма

В данном примере размер этого участка равен **64 страницам**.

Когда поступает запрос на выделение памяти, он сначала округляется до степени двух, например, до 8 страниц.

Затем весь блок памяти делится пополам (**рис. б**).

Так как получившиеся в результате этого деления надвое участки памяти все еще слишком велики, нижняя половина делится пополам еще (**рис. в**) и еще (**рис. г**).

Теперь мы получили участок памяти нужного размера. Этот участок предоставляется обратившемуся процессу (**затененный на рис. г**).

Теперь предположим, что приходит второй запрос на 8 страниц. Он может быть удовлетворен немедленно (**рис. д**).

Следом поступает запрос на 4 страницы. При этом делится надвое наименьший участок (**рис. е**) и выделяется половина от половины (**рис., ж**).

Затем освобождается второй 8-страничный участок (**рис. з**).

Наконец, освобождается оставшийся 8-страничный участок.

Поскольку два участка были «приятелями», то есть они вышли из одного 16-страничного блока, они снова объединяются в 16-страничный блок (**рис. и**).

Операционная система Linux управляет памятью при помощи данного алгоритма.

К нему добавляется массив, в котором

первый элемент представляет собой начало списка блоков размером в **1 единицу**, **второй элемент** является началом блоков размером в 2 единицы,

третий элемент — началом списка блоков размером в **4 единицы** и т. д.

Таким образом, можно быстро найти любой блок, размером кратным степени 2.

Этот алгоритм приводит к существенной внутренней фрагментации, так как. если вам нужен 65-страничный участок, вы получите 128-страничный блок.

Чтобы как-то решить эту проблему, в системе Linux есть **второй алгоритм** выделения памяти, выбирающий блоки памяти при помощи «приятельского» алгоритма, а затем нарезающий из этих блоков более мелкие фрагменты и управляющий этими фрагментами отдельно.

Кроме того, существует **третий алгоритм** выделения памяти, использующийся, когда выделяемая память должна быть непрерывна только в виртуальном адресном пространстве, но не в физической памяти.

Все эти алгоритмы выделения памяти были взяты из системы System V.

Операционная система Linux является системой, предоставляющей страницы по требованию, без предварительной загрузки страниц и без

концепции рабочего набора (хотя в ней есть системный вызов для указания пользователем страницы, которая ему может скоро понадобиться).

Текстовые сегменты и отображаемые на адресное пространство памяти файлы подгружаются из соответствующих им файлов на диске.

Все остальное выгружается либо в область подкачки, если она присутствует, либо в файлы подкачки фиксированной длины, которых может быть от одного до восьми.

Файлы подкачки могут динамически добавляться и удаляться, и у каждого есть свой приоритет.

Выгрузка страниц в отдельный раздел диска, доступ к которому осуществляется как к отдельному устройству, не содержащему файловой системы, более эффективна, чем выгрузка в файл, по нескольким причинам. Во-первых, не требуется преобразование блоков файла в блоки диска. Во-вторых, физическая запись может быть любого размера, а не только размера блока файла. В-третьих, страница всегда пишется прямо на устройство в виде единого непрерывного участка, а при записи в файл подкачки это может быть и не всегда так.

Страницы на устройстве подкачки или дисковом разделе подкачки не выделяются, пока они не потребуются.

Каждое устройство или файл подкачки начинается с битового массива, в котором сообщается, какие страницы свободны. Когда страница, у которой нет места хранения на диске, должна быть удалена из памяти, из файлов (или разделов) подкачки, в которых еще есть свободное место, выбирается файл с наивысшим приоритетом, и в нем выделяется место для страницы. Как правило, у раздела подкачки (если таковой имеется) более высокий приоритет, чем у любого файла подкачки. Координата страницы на диске записывается в таблицу страниц.

Алгоритм замещения страниц работает следующим образом.

Система Linux пытается поддерживать некоторые страницы свободными, чтобы их можно было предоставить при необходимости. Конечно, этот пул

страниц должен постоянно пополняться, поэтому реальный алгоритм страничной подкачки заключается в том, как это происходит.

Во время загрузки процесс **init** запускает страничный демон **kswapd,** который работает **один раз в секунду**. Он проверяет, есть ли достаточное количество свободных страниц. Если да, он отправляется спать еще секунду, хотя он может быть разбужен и раньше, если внезапно понадобятся дополнительные страницы.

Страничный демон состоит из цикла, который выполняется до шести раз с возрастающей срочностью. Почему шесть? Вероятно, автор программы думал, что четырех будет недостаточно, а восемь будет слишком много. В отдельных местах операционная система Linux реализована именно так.

Тело цикла выполняет обращения к трем процедурам, каждая из которых пытается получить различные типы страниц. Значение срочности передается в виде параметра, сообщающего процедуре, сколько усилий требуется предпринять, чтобы получить некоторые страницы. Как правило, это означает, сколько страниц нужно проверить, прежде чем опустить руки. В результате этот алгоритм сначала выбирает легко доступные страницы каждой категории, после чего переходит к трудно доступным. Когда получено достаточное количество страниц, страничный демон снова отправляется спать.

Первая процедура пытается получить те страницы из страничного кэша и буферного кэша файловой системы, к которым в последнее время не было обращений. для чего используется алгоритм часов.

Вторая процедура ищет совместно используемые страницы, которыми никто из пользователей, похоже, не пользуется активно.

Третья процедура, пытающаяся получить страницы, используемые одиночными пользователями, является наиболее интересной, поэтому рассмотрим ее подробнее.

Сначала выполняется цикл по всем процессам, в котором определяется, у какого процесса больше всего страниц на данный момент находится в памяти. Как только такой процесс найден, сканируются все его структуры ***vm_area_struct***

и изучаются все страницы в порядке виртуальных адресов, начиная с того места, на котором этот процесс был отложен в прошлый раз. Если страница недействительна, отсутствует в памяти, используется совместно, фиксирована в памяти или используется для DMA, то она пропускается. Если у страницы установлен бит обращения к ней, этот бит сбрасывается и страница отправляется в резерв. Если же бит сброшен, эта страница отнимается у процесса. В результате данный алгоритм подобен алгоритму часов (с той разницей, что страницы не сканируются в порядке FIFO).

Если страница, выбранная для удаления из памяти, **чистая, она удаляется немедленно**.

Если страница **«грязная»** и у нее есть место резервного хранения на диске, она **устанавливается в очередь записи на диск**.

Наконец, если у «грязной» страницы нет места резервного хранения на диске, она отправляется в странички кэш, из которого она может быть снова получена позднее, если обращение к ней поступит прежде, чем она будет фактически выгружена на диск.

В основе идеи сканирования страниц в порядке виртуальных адресов лежит надежда на то, что страницы, расположенные близко друг к другу в виртуальном адресном пространстве, скорее всего будут использоваться или не использоваться вместе, как единая группа, поэтому их следует записывать на диск как группу, а затем вместе считывать в память.

В управлении памятью принимает участие еще один демон, **bdflush**. Он периодически просыпается (а в некоторых случаях его явно будят), чтобы проверить, не превысило ли количество «грязных» страниц определенного предельного уровня. Если превысило, демон начинает сохранять их на диске.