МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ им. Н.Э. БАУМАНА
Факультет «Информатика и системы управления»
Кафедра «Автоматизированные системы обработки информации и управления»


**Сёмкин П.С., Сёмкин А.П.**

Методические указания по выполнению лабораторных работ
по дисциплине
«Операционные системы»

Лабораторная работа № 12

# «Управление процессами ОС Ubuntu»

**Москва**
**2017 г.**

**ОГЛАВЛЕНИЕ**



# 1 Цель работы

Целью работы является знакомство со средствами управления процессами ОС Ubuntu

Продолжительность работы – 2 часа.

# 2 Теоретическая часть

## 2.1 *Подсистема управление процессами*

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

Хотя в первую очередь она отвечает **за распределение процессоров** для выполнения процессов, функциями подсистемы управления процессами также

являются **передача сигналов, загрузка модулей ядра и приемом прерываний.**

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

## 2.2 *Организация процессов и потоков*

### 2.2.1 Структуры данных процессов и потоков

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

Изнутри они представляют собой единую структуру данных.

Планировщик 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. 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 структура **task_struct**, служащей в роли дескриптора процесса (т.е. блока управления процессором (PCB).

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

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).

### 2.2.2 Граф состояния задачи

Задача переходит в состояние **running (выполнения)** после выделения ей процессора.

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

Состояние **zombie (зомби)** показывает, что выполнение задачи прекратилось, 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.**

Другие важные переменные позволяют планировщику вычислять время выполнения процесса 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.**

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

## 2.3 *Системные функции управления задачами*

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 же 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**.

## 2.4 *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, 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, пребывающие 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.

Сердцем реализации потоков в системе Linux является 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.Каждый бит управляет одним из аспектов совместного использования, и каждый из битов может быть установлен независимо от остальных битов.

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 chdir 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 chdir 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 1 seek одним потоком становится видимым для других потоков, что также обычно справедливо для потоков одного процесса, но не для потоков различных процессов.

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.

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 объектов (параметры планирования, образ памяти и т. д.), 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. Сам факт наличия такой высокой степени детализации совместного использования еще не означает, что она полезна, особенно учитывая, что в системе UNIX это не поддерживается. Если какая-либо программа в системе Linux пользуется этим преимуществом, это означает, что она не может без переделок работать в системе UNIX.

# 3 Планирование и диспетчеризация в системе Linux

## 3.1 *Алгоритм диспетчеризации*

Диспетчеризация представляет собой одну из немногих областей, в которых операционная система Linux использует алгоритм, отличный от применяющегося в UNIX.

Диспетчеризация основана на потоках, а не на процессах. В операционной системе Linux алгоритмом диспетчеризации различаются три класса потоков:

1. **Потоки реального времени**, обслуживаемые по алгоритму FIFO.
2. **Потоки реального времени**, обслуживаемые в порядке циклической очереди.

### 3. Потоки разделения времени

**Потоки реального времени, обслуживаемые по алгоритму FIFO**, имеют наивысшие приоритеты и не могут прерываться другими потоками, за исключением такого же потока реального времени FIFO, перешедшего в состояние готовности.

**Потоки реального времени, обслуживаемые в порядке циклической очереди**, представляют собой то же самое, что и потоки реального времени FIFO, но с тем отличием, что они могут прерываться таймером. Находящиеся в состоянии готовности потоки реального времени, обслуживаемые в порядке циклической очереди, выполняются в течение определенного кванта времени, после чего поток помещается в конец своей очереди. Ни один из этих классов на самом деле не является классом реального времени. Здесь нельзя задать предельный срок выполнения задания и предоставить гарантий его выполнения. Эти классы просто имеют более высокий приоритет, чем у потоков стандартного класса разделения времени. Причина, по которой в системе Linux эти классы называются классами реального времени, в том, что операционная система Linux совместима со стандартом P1003.4 (расширение «реального времени» для UNIX), в котором они носят эти имена.

## 3.2 *Приоритет планирования*

У каждого потока есть **приоритет планирования**. Значение по умолчанию равно 2**0**, но оно может быть изменено при помощи системного вызова **nice(value)**, вычитающего значение **value** из **20**. Поскольку **value** должно находиться в диапазоне от **-20** до **+19**, приоритеты всегда попадают в промежуток от **1** до **40**.

**Цель алгоритма планирования состоит в том, чтобы обеспечить грубое пропорциональное соответствие качества обслуживания приоритету, то есть чем выше приоритет, тем меньше должно быть время отклика и тем большая доля процессорного времени достанется процессу.**

Помимо приоритета с каждым процессом связан квант времени, то есть количество тиков таймера, в течение которых процесс может выполняться. По

умолчанию системные часы тикают с частотой **100 Гц**, так что каждый тик равен **10 мс**. Этот интервал в системе Linux называют «джиффи» (jiffy - мгновение, миг, момент).

Планировщик использует приоритет и квант следующим образом. Сначала он вычисляет называемую в системе Linux «**добродетелью**» (goodness) величину каждого готового процесса по следующему алгоритму:

**if (class ==real_time) goodness = 1000 + priority;**

**if (class == timesharing S& quantum > 0) goodness = quantum + priority;**

**if (class == timesharing && quantum == 0) goodness = 0;**

Для обоих классов реального времени выполняется первое условие.

Все, что дает пометка процесса, как процесса реального времени, — это гарантия, что этот процесс получит более высокое значение **goodness**, чем все процессы разделения времени.

У алгоритма есть еще одно дополнительное свойство: если у процесса, который запускался последним, осталось неиспользованное процессорное время, он получает бонус, позволяющий выиграть в спорных ситуациях. Идея состоит в том, что при прочих равных условиях более эффективным представляется запустить предыдущий процесс, так как его страницы и кэш с большой вероятностью еще находятся на своих местах.

В остальном алгоритм планирования очень прост: когда нужно принять решение, выбирается поток с максимальным значением «**добродетели**». Во время работы процесса его квант (переменная **quantum**) уменьшается на единицу на каждом тике. Центральный процессор отнимается у потока при выполнении одного из следующих условий:

1. **Квант потока уменьшился до 0.**
2. **Поток блокируется на операции ввода-вывода, семафоре и т. д.**
3. **В состояние готовности перешел ранее заблокированный поток с более высокой «добродетелью».**

Так как кванты постоянно уменьшаются, рано или поздно у любого потока квант станет нулевым. Однако у потока, блокированного вводом-выводом, может остаться некая ненулевая величина кванта. В этот момент планировщик пересчитывает значения квантов для всех потоков, как готовых, так и блокированных, по следующей формуле:

**quantum = (quantum/2) + priority**

где квант измеряется в «джиффи», то есть в тиках.

Поток, ограниченный производительностью центрального процессора, как правило, быстро истратит свой квант, и при пересчете кванта его новое значение будет равно приоритету потока.

В то же время у потока, ограниченного вводом-выводом, может остаться значительное количество неистраченного процессорного времени, поэтому в следующий раз значение его нового кванта будет больше, чем у потока, ограниченного производительностью процессора.

Если системный вызов **nice** не используется, приоритет потока будет равен 20 и квант станет равным 20 тикам или 200 мс.

С другой стороны, у потока, сильно ограниченного вводом-выводом, к моменту пересчета квантов может остаться квант, равный 20. Поэтому если его приоритет также равен 20, то новое значение его кванта будет равно 20/2 + 20 = 30 тиков.

Если он опять заблокируется вводом-выводом, прежде чем успеет истратить один тик, то в следующий раз его квант будет равен 30/2 + 20 = 35 тиков. Эта величина стремится снизу к удвоенному значению приоритета.

В результате применения данного алгоритма потоки, ограниченные вводом-выводом, получают большие кванты времени и, следовательно, считаются более «**добродетельными**», чем потоки, ограниченные производительностью процессора.

**Таким образом, потоки, ограниченные вводом-выводом, получают преимущество при планировании.**

Другое свойство этого алгоритма заключается в том, что когда потоки, ограниченные производительностью процессора, соревнуются за право использования процессора, поток с большим приоритетом получает большую долю процессорного времени.

В качестве примера рассмотрим два потока, ограниченных производительностью процессора: **поток А** с приоритетом **20** и **поток В** с приоритетом **5**. **Поток А** запускается первым, и через 20 тиков его квант истекает. Затем запускается **поток В**, которому разрешается работать в течение 5 тиков.

После 5 тиков, так как все кванты упали до нуля, они пересчитываются. Поток А снова получает 20 тиков, а поток В — 5 тиков. Так продолжается, пока один из потоков не выполнит всю свою работу, **таким образом, поток А получает 80 % процессорного времени, а поток В получает 20 % процессорного времени.**

Планировщик процессов позволяет решить проблему масштабируемости для высокопроизводительных компьютерных систем благодаря. поддержке архитектур SMP и NUMA и обеспечению структурного связывания процессоров. Одно из наиболее важных усовершенствований в версии 2.6 ядра заключается в том, что все функции планирования представляют собой операции с одинаковым временем выполнения, то есть время, требуемое для выполнения операций планирования, не зависит от количества задач в системе.

Каждое прерывание от системного таймера (генерируется с интервалом, зависящим от архитектуры системы, для IA-32 он равен 1 миллисекунде) вызывает обновление ядром различных учетных структур данных (например, счетчика времени, в течение которого выполняется данный процесс) и при необходимости — запуск операции 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.

### 3.3 *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. Данная проблема решается в Linux путем динамического изменения количества задач, выполняемых на том или ином процессоре системы.

### 3.4 *Изменение приоритета 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 информации 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

использующих 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.

## 3.5 *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.

## 3.6 *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 использования 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.

### 3.7 *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 планирования, 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 времени, поэтому она выполняется на процессоре до тех пор, пока не будет выполнен выход из нее, задача не перейдет в спящий режим либо ее выполнение не будет прервано.

Очевидно, что плохо 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**.

# 4 Задание на выполнение работы

**1.Запустить программу виртуализации Oracle VM VirtualBox.**

**2.Запустить виртуальную машину Uduntu.**

**3. Открыть окно интерпретатора команд**

**4. Вывести общую информацию о системе**

4.1 Вывести информацию о текущем интерпретаторе команд

4.2 Вывести информацию о текущем пользователе

4.3 Вывести информацию о текущем каталоге

4.4 Вывести информацию об оперативной памяти и области подкачки

4.5 Вывести информацию о дисковой памяти

**5.Выполнить команды получения информации о процессах**

5.1 Получить идентификатор текущего процесса(PID)

5.2 Получить идентификатор родительского процесса(PPID)

5.3 Получить идентификатор процесса инициализации системы

5.4 Получить информацию о выполняющихся процессах текущего пользователя в текущем интерпретаторе команд

5.5 Отобразить все процессы

**6. Выполнить команды управления процессами**

6.1 Получить информацию о выполняющихся процессах текущего пользователя в текущем интерпретаторе

6.2 Определить текущее значение **nice** по умолчанию

6.2 Запустить интерпретатор bash с понижением приоритета

**nice –n 10 bash**

6.3 Определить **PID** запущенного интерпретатора

6.4 Установить приоритет запущенного интерпретатора равным 5

**renice –n 5 <PID процесса>**

6.5 Получить информацию о процессах **bash**

**ps lax | grep bash**

# 5 Контрольные вопросы

1. Перечислите состояния задачи в ОС Ubuntu.
2. Как создаются задачи задачи в ОС Ubuntu?
3. Назовите классы потоков ОС Ubuntu
4. Как используется приоритет планирования при запуске задачи
5. Как можно изменить приоритет для выполняющейся задачи?

## *6 ЛИТЕРАТУРА*

1. Робачевский А.М. Операционная система UNIX.-СПб.: БХВ-Петербург, 2001. – 528 с.:ил.
2. Негус К. Ubuntu и Debian Linux для продвинутых. 2-е изд. – СПб.: Питер,2014. -384 с.: ил.

# 7 Приложение. Команды Linux

## 7.1 *Команды получения общей информации о системе*

В приглашении терминала отображаются имя пользователя(**student**) и имя хоста(**student-VirtualBox**)

**echo $SHELL** - получение информации о текущем интерпретаторе

**whoami** - получение информации о текущем пользователе

**pwd** - получение информации о текущем каталоге

**free** - получение информации об оперативной памяти и файле подкачки

**df** - получение информации о дисковой памяти

## 7.2 *Команды получения информации о процессах*

**echo $$** - -получение идентификатора текущего процесса**(PID)**

**echo $PPID** - получение идентификатора родительского процесса**(PPID)**

**pidof <имя процесса>** - получение идентификатора процесса по его имени

**ps <параметр>** - получение информации о выполняемых процессах

| **Параметр** | **Описание** |
|---|---|
| Без параметра | информации о выполняемых процессах текущего пользователя в текущем интерпретаторе команд |
| -a | отобразить все процессы, связанных с терминалом (отображаются процессы всех пользователей) |
| -e | отобразить все процессы |
| -t список терминалов | отобразить процессы, связанные с терминалами |
| -u идентификаторы пользователей | отобразить процессы, связанные с данными идентификаторами |
| -g идентификаторы групп | отобразить процессы, связанные с данными идентификаторами групп |
| -x | отобразить все процессы, не связанные с терминалом |
| lax | отобразить информацию о значения nice процессов |

## 7.3 *Команды управления процессами*

Определение текущего значения nice по умолчанию
**nice**

Понижение приоритета запускаемого процесса
**nice –n** [коэффициент понижения] команда [аргумент]

Понижение приоритета выполняемого процесса
**renice –n** [значение nice] **–p** [PID процесса]