**Оглавление**



# 1 Понятие процесса UNIX.

**Процессом** в системе UNIX называется **единица исполнения программного кода** (классическая UNIX не поддерживает потоки).

В основе UNIX лежит концепция процесса - единицы управления и единицы потребления ресурсов.

**Процесс представляет собой программу в состоянии выполнения, причем в UNIX в рамках одного процесса не могут выполняться никакие параллельные действия.**

Каждый процесс работает в своем виртуальном адресном пространстве. Совокупность участков физической памяти, отображаемых на виртуальные адреса процесса, называется **образом процесса**.

# 2 Основные структуры данных процесса.

При управлении процессами операционная система использует два основных типа информационных структур:

- Дескриптор процесс(структура **proc**)
- Контекст процесса(структура **user**)

## *2.1 Дескриптор процесса.*

Дескриптор процесса (структура **proc**) содержит такую информацию о процессе, которая необходима ядру в течение всего жизненного цикла процесса, независимо от того, находится ли он в активном или пассивном состоянии, находится ли образ процесса в оперативной памяти или выгружен на диск. Дескрипторы отдельных процессов объединены в список, образующий **системную таблицу процессов**.

Память для таблицы процессов отводится динамически в области ядра.

На основании информации, содержащейся в таблице процессов, операционная система осуществляет планирование и синхронизацию процессов.

**В дескрипторе** прямо или косвенно (через указатели на связанные с ним структуры) содержится информация о

- **состоянии процесса,**
- **расположении образа процесса в оперативной памяти и на диске,**
- **значении отдельных составляющих приоритета, а также его итоговое значение - глобальный приоритет,**
- **идентификатор пользователя, создавшего процесс,**
- **информация о родственных процессах, о событиях, осуществления которых ожидает данный процесс**
- **некоторая другая информация.**

## *2.2 Контекст процесса.*

Контекст процесса (структура **user**) содержит менее оперативную, но более объемную часть информации о процессе, необходимую для возобновления выполнения процесса с прерванного места:

Эта информация **сохраняется**, когда выполнение процесса приостанавливается, и **восстанавливается**, когда планировщик предоставляет процессу вычислительные ресурсы.

Контекст процесса состоит из нескольких частей:

***Адресное пространство процесса в режиме задачи***. Сюда входят код, данные и стек процесса, а также другие области, например, разделяемая память или код и данные динамических библиотек.

***Управляющая информация***. Ядро использует две основные структуры данных для управления процессом — **proc** и **user**. Сюда же входят данные, необходимые для отображения виртуального адресного пространства процесса в физическое

***Окружение процесса.*** Переменные окружения процесса представляют собой строки пар вида:

*переменная=значение,*

которые наследуются дочерним процессом от родительского и обычно хранятся в нижней части стека.

***Аппаратный контекст***. Сюда входят значения общих и ряда системных регистров процессора.

К системным регистрам, в частности, относятся:

- указатель команд, содержащий адрес следующей команды, которую необходимо выполнить;
- указатель стека, содержащий адрес последнего элемента стека;
- регистры плавающей точки;
- регистры управления памятью, отвечающие за трансляцию виртуального адреса процесса в физический.

**Переключение между процессами** по существу выражается в ***переключении контекста***, когда контекст выполнявшегося процесса запоминается, и восстанавливается контекст процесса, выбранного планировщиком. Переключение контекста является достаточно ресурсоемкой операцией. Помимо сохранения состояния регистров процесса, ядро вынуждено выполнить множество других действий.

Существуют **четыре ситуации, при которых производится переключение контекста**:

1. Текущий процесс переходит в состояние сна, ожидая недоступного

ресурса.

2. Текущий процесс завершает свое выполнение.

3. После пересчета приоритетов в очереди на выполнение находится более высокоприоритетный процесс.

4. Происходит пробуждение более высокоприоритетного процесса.

**Первые два случая** соответствуют **добровольному** переключению контекста и действия ядра в этом случае достаточно просты. Ядро вызывает процедуру переключения контекста из функций **sleep ()** или **exit ().**

Третий и четвертый случаи переключения контекста происходят не по воле процесса, который в это время выполняется в режиме ядра и поэтому не может быть немедленно приостановлен. В этой ситуации ядро устанавливает специальный флаг **runrun**, который указывает, что в очереди находится более высокоприоритетный процесс, требующий предоставления вычислительных ресурсов. Перед переходом процесса из режима ядра в режим задачи ядро проверяет этот флаг и, если он установлен, вызывает функцию переключения контекста.

Контекст, так же как и дескриптор процесса, доступен только программам ядра, то есть **находится в виртуальном адресном пространстве операционной системы. О**днако контекст хранится не в области ядра, а непосредственно примыкает к образу процесса и перемещается вместе с ним, если это необходимо, из оперативной памяти на диск.

## 3 Уровни выполнения процессов

Выполнение пользовательских процессов в системе UNIX осуществляется на двух уровнях: **уровне пользователя** и **уровне ядра**.

Когда процесс производит обращение к операционной системе, режим выполнения процесса переключается с **режима задачи (пользовательского)** на **режим ядра**:

- операционная система пытается обслужить запрос пользователя, возвращая код ошибки в случае неудачного завершения операции.

**Основные различия между этими двумя режимами**:

- В режиме **задачи** процессы имеют доступ только к своим собственным командам и данным, но не к командам и данным ядра (либо других роцессов).
- В режиме **ядра** процессам уже доступны адресные пространства ядра и пользователей. Например, виртуальное адресное пространство процесса может быть поделено на адреса, доступные только в режиме ядра, и на адреса, доступные в любом режиме.
- Некоторые машинные команды являются привилегированными и вызывают возникновение ошибок при попытке их использования в режиме задачи: например, команда, управляющая регистром состояния процессора; процессам, выполняющимся в режиме задачи, она недоступна.

Несмотря на то, что система функционирует в одном из двух режимов, ядро действует от имени пользовательского процесса. Ядро не является какой-то особой совокупностью процессов, выполняющихся параллельно с пользовательскими, оно само выступает составной частью любого пользовательского процесса.

При необходимости выполнить привилегированные действия пользовательский процесс обращается с запросом к ядру в форме так называемого **системного вызова**.

В результате системного вызова управление передается соответствующей программе ядра. С момента начала выполнения системного вызова процесс считается системным.

Таким образом, **один и тот же процесс может находиться в пользовательской или системной фазах**. Эти фазы никогда не выполняются одновременно.

*В классической UNIX процесс, работающий в режиме системы, не мог быть вытеснен другим процессом. Из-за этого организация ядра, которое составляет привилегированную общую часть всех процессов, упрощалась, т.к. все функции ядра не были реентерабельными. Однако при этом реактивность системы страдала - любой процесс, даже низкоприоритетный, войдя в системную фазу, мог оставаться в ней*

*сколь угодно долго. Из-за этого свойства UNIX не мог использоваться в качестве ОС реального времени. В более поздних версия, организация ядра усложнилась и процесс можно вытеснить и в системной фазе, но не в произвольный момент времени, а только в определенные периоды его работы, когда процесс сам разрешает это сделать установкой специального сигнала.*

# 4 Системные вызовы управления процессами

## *4.1 Порождение процессов*

Порождение процессов в системе UNIX происходит следующим образом.

- При создании процесса( системный вызов **fork)** строится образ порожденного процесса, являющийся точной копией образа породившего процесса.
- Сегмент данных и сегмент стека отца действительно копируются на новое место, образуя сегменты данных и стека сына.
- Процедурный сегмент копируется только тогда, когда он не является разделяемым. В противном случае сын становится еще одним процессом, разделяющим данный процедурный сегмент.

После выполнения системного вызова **fork** оба процесса продолжают выполнение с одной и той же точки. Чтобы процесс мог опознать, является ли он отцом или сыном, системный вызов fork возвращает в качестве своего значения в породивший процесс идентификатор порожденного процесса, а в порожденный процесс NULL.

Идентификатор сына может быть присвоен переменной, входящей в контекст процесса-отца. Так как контекст процесса наследуется его потомками, то дети могут узнать идентификаторы своих старших братьев, так образом сумма знаний наследуется при порождении и может быть распространена между родственными процессами. Наследуются все характеристики процесса, содержащиеся в контексте.

На независимости идентификатора процесса от выполняемой процессом программы построен механизм, позволяющий процессу придти к выполнению другой программы с помощью системного вызова exec.

Таким образом, в UNIX **порождение нового процесса происходит в два этапа** –

- сначала создается копия процесса-родителя, то есть дублируется дескриптор, контекст и образ процесса.
- затем у нового процесса производится замена кодового сегмента на заданный.

Вновь созданному процессу операционная система присваивает целочисленный идентификатор, уникальный на весь период функционирования системы.

Любой процесс создается с помощью системного вызова **fork(),** возвращающего идентификатор процесса.

В ответ на вызов UNIX создает новое адресное пространство с полной копией содержимого адресного пространства родительского процесса.

Затем в адресное пространство с помощью системного вызова **execve()** может быть загружена для выполнения программа.

В этом случае существующее содержимое сегмента кода процесса заменяется кодом новой программы, а содержимое сегмента данных обновляется. Поскольку процесс часто создается для выполнения новой программы, работа по копированию адресного пространства родительского процесса оказывается напрасной.

## *4.2 Синхронизация между родительским и дочерним процессами*

Если родительский процесс создает дочерний процесс для выполнения определенной задачи, ему нужно знать, когда тот завершит свою работу.

С этой целью применяется следующий системный вызов, выполняемый дочерним процессом:

**exit(информация о состоянии)**

В ответ на него из системы удаляется вся информация адресного пространства процесса, за исключением управляющего блока, где хранится предназначенная для родительского процесса информация о состоянии.

Такой процесс в UNIX называется «зомби».

Родительский процесс, в свою очередь, выполняет блокирующий системный вызов **wait()**, указывая идентификатор дочернего процесса, а в ответ система возвращает байт состояния и некоторую дополнительную информацию.

Общая схема, иллюстрирующая порядок создания, завершения и синхронизации процессов, показана на **рис. 5**.

## 5 Состояния процесса

Сначала процесс создается и направляется на выполнение:

- **с-гп**: только что созданному процессу выделяется некоторый объем основной памяти;
- **гп-вя**: процесс направляется на выполнение с некоторым начальным приоритетом;
- **вя-вп**: перешел из режима ядра в пользовательский режим.

Далее процесс выполняется в пользовательском режиме - он либо будет прерван, либо выполнит системный вызов.

- **вп-вя**: процесс, работающий в пользовательском режиме, произвел системный вызов и теперь выполняет код ядра или же произошло прерывание, например от устройства или таймера, и это прерывание обрабатывается в контексте данного процесса;
- **вя-вп**: возврат из системного состояния в пользовательское после обработки прерывания или по окончании функционирования вызванной системной процедуры (либо сразу, либо после периода сна, когда процесс пробуждается и направляется на выполнение: **вя-сп-гп-вя**);
- **вя-в**: процесс вытеснен в тот момент, когда должен был перейти в пользовательское состояние (это могло произойти из-за того, что

процесс ядра был переведен в состояние готовности в результате прерывания или же по причине того, что после прерывания от таймера выяснилось, что время данного процесса истекло;

- **в- вп**: ни один из системных процессов не находится в состоянии готовности, поэтому на выполнение направляется вытесненный процесс;
- **вя-вя**: произошло прерывание, которое обрабатывается в контексте прерванного процесса; если этот процесс выполнялся в пользовательском режиме, механизм прерывания изменяет его состояние на системное;
- **вя-сп**: процесс, выполняющий код ядра, засыпает в ожидании события;
- **сп-гп**: произошло событие, которого ожидал процесс, и он переводится в состояние готовности;
- **гп-вп**: процесс направляется на выполнение в соответствии с тем приоритетом, который был у него на момент блокировки;
- **сп-со**: спящий процесс откачан из памяти на диск;
- **со-го**: произошло событие, которого ждал процесс, и он переводится в состояние готовности, но все еще находится на диске;
- **го-гп**: процесс загружается с диска в память;
- **гп-го**: готовый к выполнению процесс откачивается из памяти на диск;
- **с-го**: для нового процесса недостаточно памяти;
- **ря-з**: процесс выполняет системный вызов **exit()** (**вп-вс**) и удаляется из системы; он остается «зомби» до тех пор, пока его код не будет присоединен родительским процессом.

При переходах **вя-сп** и **гп-вя** производится проверка сигналов, связанных со взаимодействием процессов; кроме того, сигналы проверяются и обрабатываются, когда процесс намерен перейти в пользовательское состояние (**вя-вп** или **в-вп**).

Приведенная схема отражает процедурную природу UNIX.

**Системные вызовы выполняются в вызывающем процессе, который**

**лишь переходит в привилегированное состояние.**

Выделенный процесс для обработки прерываний от устройств отсутствует, поэтому все такого рода прерывания обрабатываются в контексте прерванного процесса.

Механизм прерываний гарантирует их обработку в привилегированном режиме.

Пользовательский процесс никогда не блокируется непосредственно: он делает это либо после попытки выполнения ввода или вывода, либо в результате вызова для взаимодействия с другим процессом, причем в обоих случаях сначала входит в ядро.

Процесс должен войти в ядро и для того, чтобы завершить свою работу.

## 6 Диспетчеризация процессов. Уровни приоритетов

В UNIX реализована диспетчеризация на основе **динамического приоритета** процессов, согласно которому для выполнения каждый раз выбирается процесс с наивысшим приоритетом.

При этом **диспетчер вычисляет приоритеты процессов с учетом их поведения, а не назначает им статические приоритеты при создании.**

- Процесс, который в настоящий момент находится в ядре, имеет более высокий приоритет, чем процессы пользователя. Процессы ядра планируются без вытеснения и поэтому, выполняя системный код, процесс работает до тех пор, пока не произведет некоторый блокирующий вызов или не будет временно прерван устройством.
- При блокировании процессу назначается приоритет, соответствующий событию, которое он ждет.
- Пользовательские процессы планируются в соответствии с принципом квантования по времени. Алгоритм планирования учитывает количество использованного процессом времени процессора и на основании этого динамически назначает ему приоритет.

  Планировщик, как и большинство других компонентов ядра UNIX,

выполняется в контексте пользовательского процесса.

# 7 Взаимодействие между процессами в классической UNIX

## *7.1 Использовании процессами общего файла.*

Для координации такого взаимодействия дочерний процесс, завершая работу, может возвращать известный родительскому процессу код, указывающий, что файл готов для чтения.

Однако это решение считается неудовлетворительным по нескольким причинам.

**Во-первых**, предполагается, что вначале файл полностью записывается, а затем начинает считываться,

**во-вторых**, нужен дополнительный механизм согласования имени используемого файла,

**в-третьих**, такое решение требует записи данных на диск или, по крайней мере в буферный кэш, даже если оба процесса остаются в памяти.

## *7.2 Каналы между процессами*

Канал-- однонаправленных потоков данных между двумя процессами. Этот механизм позволяет буферизировать данные в памяти, а также блокировать считывающий процесс до тех пор, пока записывающий процесс не поместит данные в канал.

Канал можно рассматривать как особый вид файла. Доступ к нему, как и доступ к обычному файлу, осуществляется с использованием идентификатора в таблице открытых файлов процесса.

У канала имеется **i-узел**, но при этом в исходной архитектуре UNIX у него нет путевого имени. Канал существует до тех пор, пока существуют процессы, у которых есть его дескрипторы. Поскольку канал поддерживаем однонаправленное взаимодействие, при его использовании один процесс записывает в него данные, а второй их считывает.

Отличия канала от обычного файла.

- У канала есть два указателя: для ввода и для вывода;
- Для создания канала используется иной системный вызов: **pipe (поля)** где поля - это целочисленный массив, состоящий из двух элементов. Данный вызов возвращает дескрипторы чтения и записи в элементах **поля[1]** и **поля[2**] соответственно.
- Непосредственное изменение положения указателя (при работе с файлом это делается с помощью вызова **seek()**) невозможно. Однажды прочитанные байты нельзя прочитать повторно.
- Если процесс пытается прочитать данные из **пустого** канала, он блокируется до их поступления в канал.
- Существует понятие «**полного**» канала - это когда записанные данные заполнили весь выделенный для канала буфер. При попытке записи в полный канал процесс блокируется до тех пор, пока второй процесс не прочитает из него часть данных. В случае использования для двунаправленного взаимодействия двух каналов существует риск взаимоблокировки.

Сходные черты канала и файла.

- После создания канала с двумя открытыми дескрипторами операции чтения и записи осуществляются так же, как для обычного файла.
- Дескрипторы канала передаются от родительского процесса к дочернему (подобно дескрипторам обычных файлов). В таблице открытых файлов процесса они не отличаются от дескрипторов файлов.

*Когда интерпретатор команд операционной системы создает процессы для параллельного выполнения введенных пользователем команд, он является их общим предком. Поэтому между программами, выполняющими подобные команды, могут использоваться каналы. Такая возможность обеспечивается, в частности, благодаря совместимости файлов и устройств в отношении ввода-вывода.*

*Практически все команды UNIX разработаны таким образом, чтобы вывод одной из них мог подаваться на вход другой.*

***Например**, для того чтобы интерпретатор команд создал каналы между командами command1, command2 и command3, нужно ввести их следующим образом:*

*command1 | command2 | command3*

*В данном случае будет создано два канала: между первой и второй командами, а также между второй и третьей.*

## *7.3 Сигналы*

Механизм предполагает, что процессы будут обмениваться сигналами - простейшими уведомлениями о событиях, идентифицируемых числовыми значениями. Сигналы могут доставляться процессам асинхронно.

Сигналы часто используются для уведомления об обнаруженных ошибках или, например, о поступлении с терминала очередного введенного пользователем символа.

Это не универсальный механизм синхронизации процессов, поскольку набор сигналов ограничен да к тому же большинство из них имеет предопределенное значение.

Первоначально для «пользовательских» целей было выделено всего два сигнала. В современных системах UNIX их уже больше, теперь процесс может ждать получения определенного сигнала, выполнив системный вызов **sigsuspend()**. Однако сигналы не используются ни для синхронизации с устройствами, ни для уведомления об освобождении ресурсов - указанные операции выполняются с помощью блокирующих системных вызовов.

В дескрипторе процесса устанавливается определенный бит, указывающий, какое из событий произошло. Если процесс вовремя не обнаружит сигнал, последующие события того же типа пройдут для него незамеченными.

*Первоначально сигналы выдавались исключительно системным кодом, но в более поздних версиях UNIX механизм сигнализации был расширен. Теперь пользовательские процессы могут посылать сигналы друг другу. Так в операционной системе BSD 4.3 имеется 20 видов сигналов, а в SVr4*

*и стандарте POSIX их по 19.*

Но сигнал, передаваемый одним пользовательским процессом другому, обнаруживается только тогда, когда этот процесс входит в ядро или прерывается. Тогда перед возвратом управления прерванному процессу пользовательского уровня система проверяет, не поступил ли для данного процесса какой-либо сигнал.