# Архитектура ОС UNIX. Особенности программирования. Управление потоками и процессами

## Лекция

Ревизия: 0.1

Последняя модификация: Vlad Kovtun

© NRJETIX 2000 - 2010

Дата последней модификации: 21.11.2010 23:25:00

## История изменений

20.10.2010 – Версия 0.1. Первичный документ. Ковтун В.Ю.

# Содержание

История изменений 2
Содержание 3
Лекция 12. Архитектура ОС UNIX. Управление потоками и процессами 4
- Вопросы 4
- Задачи UNIX 4
  - Архитектура ОС UNIX 4
  - Оболочка UNIX 5
  - Утилиты UNIX 5
- Процессы в UNIX 7
  - Основные понятия 8
  - Системные вызовы управления процессами в UNIX 9
  - Реализация процессов в UNIX 12
- Потоки 15
  - Потоки в UNIX 15
  - Потоки в системе Linux 16
  - Системные вызовы управления потоками 17
  - Планирование в системе UNIX 18
  - Планирование в системе Linux 20

Литература 22

# Лекция 12. Архитектура ОС UNIX. Управление потоками и процессами

## Вопросы

1. Задачи и архитектура ОС UNIX.

2. Процессы в UNIX.

3. Потоки в UNIX.

## Задачи UNIX

ОС UNIX представляет собой интерактивную систему, разработанную для одновременной поддержки нескольких процессов и нескольких пользователей. Она была разработана программистами и для программистов, чтобы использовать ее в окружении, в котором большинство пользователей являются относительно опытными и занимаются разработкой проектов ПО. В ОС UNIX есть достаточное количество средств, позволяющих программистам работать вместе и управлять совместным использованием общей информации. Очевидно, что модель группы опытных программистов, совместно работающих над созданием передового ПО, существенно отличается от модели одиночного начинающего пользователя, сидящего за персональным компьютером в текстовом процессоре, и это отличие отражено в ОС UNIX от начала до конца.

Чего хотят от ОС хорошие программисты:

1. Большинство хотело бы, чтобы их система была простой, элегантной и непротиворечивой. Например, на нижнем уровне файл должен представлять собой просто набор байтов. Наличие различных классов файлов для последовательного и произвольного доступа, доступа по ключу, удаленного доступа и т. д.

2. Мощь и гибкость. Это означает, что в системе должно быть небольшое количество базовых элементов, которые можно комбинировать бесконечным числом способов, чтобы приспособить их для конкретного приложения. Одно из основных правил ОС UNIX заключается в том, что каждая программа должна выполнять всего одну функцию, но делать это хорошо. Таким образом, компиляторы не занимаются созданием листингов, так как другие программы могут лучше справиться с этой задачей.

### Архитектура ОС UNIX

ОС UNIX можно рассматривать в виде пирамиды (рис. 1). У основания пирамиды располагается аппаратное обеспечение, состоящее из CPU, памяти, дисков, терминалов и других устройств. На голом «железе» работает ОС UNIX. Ее функция заключается в управлении аппаратным обеспечением и предоставлении всем программам интерфейса системных вызовов. Эти системные вызовы позволяют программам создавать процессы, файлы и прочие ресурсы, а также управлять ими.

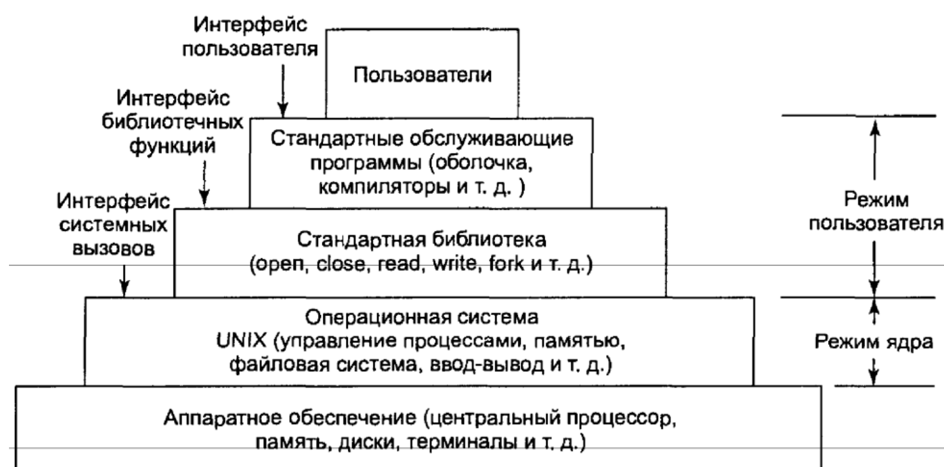



Рис. 1. Уровни ОС UNIX

Программы обращаются к системным вызовам, помещая аргументы в регистры CPU (или иногда в стек) и выполняя команду эмулированного прерывания для переключения из пользовательского режима в режим ядра и передачи управления ОС UNIX. Поскольку

на языке С невозможно написать команду эмулированного прерывания, этим занимаются библиотечные функции, по одной на системный вызов. Эти процедуры написаны на ассемблере, но они могут вызываться из программ, написанных на С. Каждая такая процедура помещает аргументы в нужное место и выполняет команду эмулированного прерывания TRAP. Таким образом, чтобы обратиться к системному вызову `read`, программа на С должна вызвать библиотечную процедуру `read`. В стандарте POSIX определен именно интерфейс библиотечных функций, а не интерфейс системных вызовов. Другими словами, стандарт POSIX определяет библиотечные процедуры, соответствующие системным вызовам, их параметры, что они должны делать и какой результат возвращать. В стандарте даже не упоминаются фактические системные вызовы.

Помимо ОС и библиотеки системных вызовов, все версии UNIX содержат большое количество стандартных программ, некоторые из них описываются стандартом POSIX 1003.2, тогда как другие могут различаться в разных версиях системы UNIX. К этим программам относятся командный процессор (оболочка), компиляторы, редакторы, программы обработки текста и утилиты для работы с файлами. Именно эти программы и запускаются пользователем с терминала.

Таким образом, можем говорить о трех интерфейсах в ОС UNIX:

1. интерфейсе системных вызовов;

2. интерфейсе библиотечных функций;

3. интерфейсе, образованным набором стандартных обслуживающих программ.

Хотя именно последний интерфейс большинство пользователей считает системой UNIX, в действительности он не имеет практически никакого отношения к самой ОС и легко может быть заменен.

В некоторых версиях ОС UNIX, например, этот ориентированный на ввод с клавиатуры интерфейс пользователя был заменен графическим интерфейсом пользователя, ориентированным на использование мыши, для чего не потребовалось никаких изменений в самой системе. Именно эта гибкость сделала систему UNIX столь популярной и позволила ей пережить многочисленные изменения технологии, лежащей в ее основе.

## Оболочка UNIX

У многих версий ОС UNIX имеется графический интерфейс пользователя, схожий с популярными интерфейсами, примененными на компьютере Macintosh и впоследствии в системе Windows. Однако истинные программисты до сих пор предпочитают интерфейс командной строки, называемый **оболочкой** (shell). Подобный интерфейс значительно быстрее в использовании, существенно мощнее, проще расширяется и не раздражает пользователя необходимостью постоянно хвататься за мышь, например оболочка Бурна (sh). С тех пор было написано много других оболочек (ksh, bash и т. д.). Хотя система UNIX полностью поддерживает графическое окружение (X Windows), даже в этом мире многие разработчики просто создают множество консольных окон и действуют так, как если бы у них была дюжина алфавитно-цифровых терминалов, на каждом из которых работала бы оболочка.

Когда оболочка запускается, она инициализируется, а затем печатает на экране символ приглашения к вводу (обычно это знак доллара или процента) и ждет, когда пользователь введет командную строку.

После того как пользователь введет командную строку, оболочка извлекает из нее первое слово и ищет файл с таким именем. Если такой файл удается найти, оболочка запускает его. При этом работа оболочки приостанавливается на время работы запущенной программы. По завершении работы программы оболочка снова печатает приглашение и ждет ввода следующей строки. Здесь важно подчеркнуть, что оболочка представляет собой обычную пользовательскую программу. Все, что ей нужно, — это способность ввода с терминала и вывода на терминал, а также возможность запускать другие программы.

## Утилиты UNIX

Пользовательский интерфейс UNIX состоит не только из оболочки, но также из большого числа стандартных обслуживающих программ, называемых также утилитами. Грубо говоря, эти программы можно разделить на шесть следующих категорий:

1. Команды управления файлами и каталогами.
2. Фильтры.
3. Средства разработки программ, такие как текстовые редакторы и компиляторы.
4. Текстовые процессоры.
5. Системное администрирование.
6. Разное.

Стандарт POSIX 1003.2 определяет синтаксис и семантику менее 100 из этих программ, в основном относящихся к первым трем категориям. Идея стандартизации данных программ заключается в том, чтобы можно было писать сценарии оболочки, которые работали бы на всех системах UNIX. Помимо этих стандартных утилит, разумеется, существует еще масса прикладных программ, таких как web-браузеры, программы просмотра изображений и т. д.

## Структура ядра

На рис. 1 была показана общая структура системы UNIX. Рассмотрим ядро системы. Обзор структуры ядра системы UNIX представляет собой довольно непростое дело, так как существует множество различных версий этой системы. Однако, хотя диаграмма на рис. 2 описывает UNIX 4.4BSD, она также применима ко многим другим версиям, возможно, с небольшими изменениями в тех или иных местах.

Системные вызовы
Аппаратные и эмулированные прерывания
Управление терминалом
Сокеты
Именование файла
Отображение адресов
Страничные прерывания
Обработка сигналов
Создание и завершение процессов
Необработанный телетайп
Обработанный телетайп
Сетевые протоколы
Файловые системы
Виртуальная память
Дисциплины линии связи
Маршрутизация
Буферный кэш
Страничный кэш
Планирование процесса
Символьные устройства
Драйверы сетевых устройств
Драйверы дисковых устройств
Диспетчеризация процессов
Аппаратура

Рис. 2. Структура ядра ОС UNIX 4.4BSD

Нижний уровень ядра состоит из драйверов устройств и процедуры диспетчеризации процессов. Все драйверы системы UNIX делятся на два класса: драйверы символьных устройств и драйверы блочных устройств. Основное различие между этими двумя классами устройств заключается в том, что на **блочных устройствах разрешается операция поиска, а на символьных нет**. Технически сетевые устройства представляют собой символьные устройства, но они обрабатываются по-иному, поэтому их, вероятно, правильнее выделить в отдельных класс, как это и было сделано на схеме. Диспетчеризация процессов производится при возникновении прерывания. При этом низкоуровневая программа останавливает выполнение работающего процесса, сохраняет его состояние в таблице процессов ядра и запускает соответствующий драйвер. Кроме того, диспетчеризация процессов производится также, когда ядро завершает свою работу и пора снова запустить процесс пользователя. Программа диспетчеризации процессов написана на ассемблере и представляет собой отдельную от процедуры планирования программу.

В более высоких уровнях программы отличаются в каждом из четырех «столбцов» диаграммы. Слева располагаются символьные устройства. Они могут использоваться двумя способами. Некоторым программам, таким как текстовые редакторы **vi** и **emacs**, требуется каждая нажатая клавиша без какой-либо обработки. Для этого служит ввод-

вывод с необработанного терминала (телетайпа). Другое программное обеспечение, например оболочка (sh), принимает на входе уже готовую текстовую строку, позволяя пользователю редактировать ее, пока не будет нажата клавиша ENTER. Такое ПО пользуется вводом с терминала в обработанном виде и линии связи.

Сетевое ПО часто бывает модульным, с поддержкой множества различных устройств и протоколов. Уровень выше сетевых драйверов выполняет своего рода функции маршрутизации, гарантируя, что правильный пакет направляется правильному устройству или блоку управления протоколами. Большинство систем UNIX содержат в своем ядре полноценный маршрутизатор Интернета, и хотя его производительность ниже, чем у аппаратного маршрутизатора, эта программа появилась раньше современных аппаратных маршрутизаторов. Над уровнем маршрутизации располагается стек протоколов, обязательно включая протоколы IP и TCP, но также иногда и некоторые дополнительные протоколы. Над сетевыми протоколами располагается интерфейс сокетов, позволяющий программам создавать сокеты для отдельных сетей и протоколов. Для использования сокетов пользовательские программы получают дескрипторы файлов.

Над дисковыми драйверами располагаются буферный кэш и страничный кэш файловой системы. В ранних системах UNIX буферный кэш представлял собой фиксированную область памяти, а остальная память использовалась для страниц пользователя. Во многих современных системах UNIX этой фиксированной границы уже не существует, и любая страница памяти может быть схвачена для выполнения любой задачи, в зависимости от того, что требуется в данный момент.

Над буферным кэшем располагаются файловые системы. Большинством систем UNIX поддерживаются несколько файловых систем, включая быструю файловую систему Беркли, журнальную файловую систему, а также различные виды файловых систем System V. Все эти файловые системы совместно используют общий буферный кэш. Выше файловых систем помещается именование файлов, управление каталогами, управление жесткими и символьными связями, а также другие свойства файловой системы, одинаковые для всех файловых систем.

Над страничным кэшем располагается система виртуальной памяти. В нем вся логика работы со страницами, например алгоритм замещения страниц. Поверх него находится программа отображения файлов на виртуальную память и высокоуровневая программа управления страничными прерываниями. Эта программа решает, что нужно делать при возникновении страничного прерывания. Сначала она проверяет допустимость обращения к памяти и, если все в порядке, определяет местонахождение требуемой страницы и то, как она может быть получена.

Последний столбец имеет отношение к управлению процессами. Над диспетчером располагается планировщик процессов, выбирающий процесс, который должен быть запущен следующим. Если потоками управляет ядро, то управление потоками также помещается здесь, хотя в некоторых системах UNIX управление потоками вынесено в пространство пользователя. Над планировщиком расположена программа для обработки сигналов и отправки их в требуемом направлении, а также программа, занимающаяся созданием и завершением процессов.

Верхний уровень представляет собой интерфейс системы. Слева располагается интерфейс системных вызовов. Все системные вызовы поступают сюда и направляются одному из модулей низших уровней в зависимости от природы системного вызова. Правая часть верхнего уровня представляет собой вход для аппаратных и эмулированных прерываний, включая сигналы, страничные прерывания, разнообразные исключительные ситуации процессора и прерывания ввода-вывода.

## Процессы в UNIX

Рассмотрим ядро и более пристально рассмотрим его основные концепции, поддерживаемые системой UNIX, а именно процессы, память, файловую систему и ввод-вывод. Эти сведения важны, так как системные вызовы — интерфейс самой ОС — управляют ими. Например, существуют системные вызовы для создания процессов, доступа к памяти, открытия файлов и ввода-вывода.

К сожалению, существует очень много версий ОС UNIX и между ними имеются определенные различия. Уделим особое внимание общим чертам всех версий, а не особенностям какой-либо одной версии.

## Основные понятия

Единственными активными сущностями в системе UNIX являются процессы. Процессы UNIX очень похожи на классические последовательные процессы. Каждый процесс запускает одну программу и изначально получает один поток управления. Другими словами, у процесса есть один счетчик команд, указывающий на следующую исполняемую команду процессора. Большинство версий UNIX позволяют процессу после того, как он запущен, создавать дополнительные потоки.

UNIX представляет собой многозадачную систему, так что несколько независимых процессов могут работать одновременно. У каждого пользователя может быть одновременно несколько активных процессов, так что в большой системе могут одновременно работать сотни и даже тысячи процессов. Действительно, на большинстве однопользовательских рабочих станций, даже когда пользователь куда-либо отлучается, работают десятки фоновых процессов, называемых демонами. Они запускаются автоматически при загрузке системы.

Типичным демоном является `cron daemon`. Он просыпается раз в минуту, проверяя, не нужно ли чего сделать. Если у него есть работа, он ее выполняет и отправляется спать дальше.

Этот демон позволяет планировать в системе UNIX активность на минуты, часы, дни и даже месяцы вперед. Например, представьте, что пользователю назначено явиться к зубному врачу в 3 часа дня в следующий вторник. Он может создать запись в базе данных демона `cron`, чтобы тот бибикнул ему, скажем, в 2:30. Когда наступает назначенный день, `cron daemon` видит, что у него есть работа, и запускает в назначенное время пищащую программу в виде нового процесса.

Демон `cron` также используется для периодического запуска задач, например ежедневной архивации диска в 4 часа ночи или напоминания забывчивым пользователям каждый год 31 октября купить новые страшненькие товары для веселого празднования Хэллоуина. Другие демоны управляют входящей и исходящей электронной почтой, очередями на принтер, проверяют, достаточно ли еще осталось свободных страниц памяти и т. д. Демоны реализуются в системе UNIX довольно просто, так как каждый из них представляет собой отдельный процесс, независимый ото всех остальных процессов.

Процессы создаются в ОС UNIX чрезвычайно несложно. Системный вызов `fork` создает точную копию исходного процесса, называемого **родительским процессом**. Новый процесс называется **дочерним процессом**. У родительского и у дочернего процессов есть свои собственные образы памяти. Если родительский процесс впоследствии изменяет какие-либо свои переменные, изменения остаются невидимыми для дочернего процесса, и наоборот.

Открытые файлы совместно используются родительским и дочерним процессами. Это значит, что если какой-либо файл был открыт до выполнения системного вызова `fork`, он останется открытым в обоих процессах и в дальнейшем. Изменения, произведенные с этим файлом, будут видимы каждому процессу. Такое поведение является единственно разумным, так как эти изменения будут также видны любому другому процессу, который тоже откроет этот файл.

Тот факт, что образы памяти, переменные, регистры и все остальное у родительского процесса и у дочернего идентично, приводит к небольшому затруднению: Как процессам узнать, который из них должен исполнять родительскую программу, а который дочернюю? Эта проблема решается просто: системный вызов `fork` возвращает дочернему процессу число 0, а родительскому — отличный от нуля **PID** (`Process IDentifier` — идентификатор процесса) дочернего процесса. Оба процесса могут проверить возвращаемое значение и действовать соответственно, как показано в рис. 5.

```
pid = fork(); /*если fork завершился успешно, pid>0 в родительском процессе*/

if (pid < 0) {

handle_error(); /*fork потерпел неудачу (например, память или какая-либо таблица переполнена)*/

} else it (pid > и) {

/*здесь располагается родительская программа.*/

}
```

```
else {
/*здесь располагается дочерняя программа.*/
}
```

Рис. 3. Создание процесса в системе UNIX

Процессы распознаются по своим PID-идентификаторам. Как уже говорилось выше, при создании процесса его PID выдается родителю нового процесса. Если дочерний процесс желает узнать свой PID, он может воспользоваться системным вызовом `getpid`. Идентификаторы процессов используются различным образом. Например, когда дочерний процесс завершается, его PID также выдается его родителю. Это может быть важно, так как у родительского процесса может быть много дочерних процессов. Поскольку у дочерних процессов также могут быть дочерние процессы, исходный процесс может создать целое дерево детей, внуков, правнуков и т, д.

В системе UNIX процессы могут общаться друг с другом с помощью разновидности обмена сообщениями. Можно создать канал между двумя процессами, в который один процесс может писать поток байтов, а другой процесс может его читать. Эти каналы иногда называют **трубами**. Синхронизация процессов достигается путем блокирования процесса при попытке прочитать данные из пустого канала. Когда данные появляются в канале, процесс разблокируется.

При помощи каналов организуются конвейеры оболочки. Когда оболочка видит строку вроде

```
sort <f | head
```

она создает два процесса, `sort` и `head`, а также устанавливает между ними канал таким образом, что стандартный поток вывода программы `sort` соединяется со стандартным потоком ввода программы `head`. При этом все данные, формируемые программой `sort`, попадают напрямую программе `head`, для чего не требуется временного файла. Если канал переполняется, система приостанавливает работу программы `sort`, пока программа `head` не удалит из него хоть сколько-нибудь данных.

Процессы также могут общаться другим способом: при помощи программных прерываний. Один процесс может послать другому так называемый **сигнал**. Процессы могут сообщить системе, какие действия следует предпринимать, когда придет сигнал. У процесса есть выбор: проигнорировать сигнал, перехватить его или позволить сигналу убить процесс (действие по умолчанию для большинства сигналов). Если процесс выбрал перехват посылаемых ему сигналов, он должен указать процедуры обработки сигналов. Когда сигнал прибывает, управление внезапно передается обработчику. Когда процедура обработки сигнала завершает свою работу, управление снова возвращается в то место процесса, в котором оно находилось, когда пришел сигнал. Обработка сигналов аналогична обработке аппаратных прерываний ввода-вывода. Процесс может посылать сигналы только членам его группы процессов, состоящей из его прямого родителя, всех прародителей, братьев и сестер, а также детей (внуков и правнуков). Процесс может также послать сигнал сразу всей своей группе за один системный вызов.

Сигналы используются и для других целей. Например, если процесс выполняет вычисления с плавающей точкой и непреднамеренно делит число на 0, он получает сигнал SIGFPE (Floating-Point Exception SIGnal — сигнал исключения при выполнении операции с плавающей точкой). В большинстве систем UNIX также имеются дополнительные сигналы, но программы, использующие их, могут оказаться непереносимыми на другие версии UNIX.

## Системные вызовы управления процессами в UNIX

Рассмотрим теперь системные вызовы UNIX, предназначенные для управления процессами. Основные системные вызовы перечислены в табл. 1, (В случае ошибки возвращаемое значение `s` равно -1, `pid` означает PID процесса, а `residual` — оставшееся время до предыдущего сигнала будильника.) Обсуждение системных вызовов проще всего начать с системного вызова `fork`. Этот системный вызов представляет собой единственный способ создания новых процессов в системах UNIX. Он создает точную копию оригинального процесса, включая все описатели файлов, регистры и все остальное. После выполнения системного вызова `fork` исходный процесс и его копия (родительский процесс и дочерний) идут каждый своим путем.

Сразу после выполнения системного вызова fork значение всех соответствующих переменных в обоих процессах одинаково, но затем изменения переменных в одном процессе не влияет на переменные другого процесса. Системный вызов fork возвращает значение, равное нулю для дочернего процесса и идентификатору (PID) дочернего процесса для родительского. Таким образом, два процесса могут определить, кто из них родитель, а кто дочерний процесс.

Таблица 1. Некоторые системные вызовы, относящиеся к процессам

| Системный вызов | Описание |
|---|---|
| pid=fork() | Создать дочерний процесс, идентичный родительскому |
| pid=waitpid(pid, &stdtloc, opLb) | Ждать завершения дочернего процесса |
| s=execve(name, argv, envp) | Заменить образ памяти процесса |
| exit(status) | Завершить выполнение процесса и вернуть статус |
| s=sigaction(sig, &act, &oldact) | Определить действие, выполняемое при приходе сигнала |
| s=sigreturn(&context) | Вернуть управление после обработки сигнала |
| s=sigprocmask(how, &set, &old) | Исследовать или изменить маску сигнала |
| s=sigpending(set) | Получить или установить блокированные сигналы |
| s=sigsuspend(sigmask) | Заменить маску сигнала и приостановить процесс |
| s=kill(pid, sig) | Послать сигнал процессу |
| residual=alarm(seconds) | Установить будильник |
| s=pause( ) | Приостановить выполнение процесса до следующего сигнала |

В большинстве случаев после системного вызова fork дочернему процессу потребуется выполнить программу, отличающуюся от программы, выполняемой родительским процессом. Рассмотрим работу оболочки. Она считывает команды с терминала, с помощью системного вызова fork выполняет введенную команду, затем ждет окончания работы дочернего процесса, после чего считывает следующую команду. Для ожидания завершения дочернего процесса родительский процесс обращается к системному вызову waitpid. У этого системного вызова три параметра. В первом параметре указывается PID процесса, завершение которого ожидается. Если вместо идентификатора процесса указать число -1, то в этом случае системный вызов ожидает завершения любого дочернего процесса. Второй параметр представляет собой адрес переменной, в которую записывается статус завершения дочернего процесса (нормальное или ненормальное завершение, а также возвращаемое на выходе значение). Третий параметр определяет, будет ли обращающийся к системному вызову waitpid процесс блокирован до завершения дочернего процесса или сразу получит управление после обращения к системному вызову.

В случае оболочки дочерний процесс должен выполнить команду, введенную пользователем. Он выполняет это при помощи системного вызова exec, который заменяет весь образ памяти содержимым файла, указанным в первом параметре системного вызова. Крайне упрощенный вариант оболочки, иллюстрирующей использование системных вызовов fork, waitpid и exec, показан на рис. 4.

```
while (TRUE) {                          /* вечный цикл */
    type_prompt();                      /* вывести приглашение ко вводу */
    read_command(command, params);      /* прочитать с клавиатуры строку */
    pid = fork( );                      /* ответвить дочерний процесс */
```

```
    if (pid < 0) {
        printf("Создать процесс невозможно");    /* ошибка */
        continue;                           /* следующий цикл */
    }
    if (pid != 0) {
        waitpid (-1, status, 0);     /*родительский процесс ждет завершения*/
                                      /* дочернего процесса */
    }else {
        execve(command, params, 0);
    }
}
```

Рис. 4. Сильно упрощенная оболочка

В самом общем случае у системного вызова exec три параметра: имя исполняемого файла, указатель на массив аргументов и указатель на массив строк окружения. Различные варианты этой процедуры, включая execl, execv, execle и execve, позволяют опускать некоторые параметры или указывать их иными способами. Все эти процедуры обращаются к одному и тому же системному вызову. Хотя сам системный вызов называется exec, библиотечной процедуры с таким именем нет.

Рассмотрим случай выполнения оболочкой команды

```
cp filel file2
```

используемой для копирования файла file1 в файл file2. После того как оболочка создает дочерний процесс, тот обнаруживает и исполняет файл cp и передает ему информацию о копируемых файлах.

Головной модуль файла cp (как и многие другие программы) содержит определение функции

```
main(argc, argv, envp)
```

где argc — счетчик слов (последовательностей символов, ограниченных пробелами) в командной строке, включая имя программы. Для вышеприведенного примера значение argc равно 3.

Второй параметр argv представляет собой указатель на массив, i-й элемент этого массива является указателем на i-е слово командной строки. В нашем примере элемент argv[0] указывает на строку «cp». Соответственно, элемент argv[1] указывает на строку «file1», а элемент argv[2] указывает на строку «file2».

Третий параметр процедуры main, envp, представляет собой указатель на переменные среды и является массивом, содержащим строки вида имя = значение, используемые для передачи программе такой информации, как тип терминала и имя рабочего каталога. На рис. 4 дочернему процессу переменные среды не передаются, поэтому третий параметр envp в данном случае равен нулю.

Если системный вызов exec показался вам слишком сложным, не отчаивайтесь. Это самый сложный системный вызов. Все остальные значительно проще. В качестве примера простого системного вызова рассмотрим exit, который процессы должны использовать, заканчивая исполнение. У него есть один параметр, статус выхода (от 0 до 255), возвращаемый родительскому процессу в переменной status системного вызова waitpid Младший байт переменной status содержит статус завершения, равный 0 при нормальном завершении или коду ошибки при аварийном завершении. Например, если родительский процесс выполняет оператор

```
n = waitpid(-l, status, 0);
```

он будет приостановлен до тех пор, пока не завертится какой-либо дочерний процесс. Если дочерний процесс завершится со, скажем, значением статуса, равным 4, в качестве параметра библиотечной процедуры exit, то родительский процесс получит PID дочернего процесса и значение статуса, равное 0x0400. Младший байт переменной

status относится к сигналам, старший байт представляет собой значение, задаваемое дочерним процессом в виде параметра при обращении к системному вызову exit.

Если процесс уже завершил свою работу, а родительский процесс не ожидает этого события, то дочерний процесс переводится в так называемое **состояние зомби**, то есть приостанавливается. Когда родительский процесс наконец обращается к библиотечной процедуре waitpid, дочерний процесс завершается.

Несколько системных вызовов относятся к сигналам, используемым различными способами. Например, если пользователь ненамеренно велит текстовому редактору отобразить содержание очень длинного файла, а затем осознает свою ошибку, то потребуется некий способ прервать работу редактора. Обычно для этого пользователь нажимает специальную клавишу (например, DEL или CTRL+C), в результате чего редактору посылается сигнал. Редактор перехватывает сигнал и останавливает вывод.

Чтобы заявить о своем желании перехватить тот или иной сигнал, процесс может воспользоваться системным вызовом sigaction. Первый параметр этого системного вызова — сигнал, который требуется перехватить. Второй параметр представляет собой указатель на структуру, в которой хранится указатель на процедуру обработки сигнала вместе с различными битами и флагами. Третий параметр указывает на структуру, в которой система возвращает информацию о текущем обрабатываемом сигнале, на случай, если позднее его нужно будет восстановить.

Обработчик сигнала может выполняться сколь угодно долго. Однако на практике обработка сигналов не занимает много времени. Когда процедура обработки сигнала завершает свою работу, она возвращается к той точке, в которой ее прервали.

Системный вызов sigaction может также использоваться для игнорирования сигнала или чтобы восстановить действие по умолчанию, заключающееся в уничтожении процесса.

Нажатие на клавишу **DEL** не является единственным способом послать сигнал. Системный вызов kill позволяет процессу послать сигнал любому родственному процессу. Выбор названия для данного системного вызова (kill - убить, уничтожить) не особенно удачен, так как по большей части он используется процессами не для уничтожения других процессов, а, наоборот, в надежде, что этот сигнал будет перехвачен и обработан соответствующим образом.

Во многих приложениях реального времени бывает необходимо прервать процесс через определенный интервал времени, чтобы заставить его сделать что-либо, например переслать повторно возможно потерянный пакет по ненадежной линии связи. Для обработки данной ситуации предоставлен системный вызов alarm (будильник). Параметр этого системного вызова задает временной интервал, по истечении которого процессу посылается сигнал SIGALRM. У процесса в каждый момент времени может быть только один будильник. Например, если процесс обращается к системному вызову alarm с параметром 10 с, а 3 с спустя снова обращается к нему с параметром 20 с, то он получит только один сигнал через 20 с после второго системного вызова. Первый сигнал будет отменен вторым обращением к системному вызову alarm. Если параметр системного вызова alarm равен нулю, то такое обращение отменяет любой сигнал будильника. Если сигнал будильника не перехватывается, то действие по умолчанию заключается в уничтожении процесса. Технически возможно игнорирование данного сигнала, но смысла такое действие не имеет.

Иногда случается так, что процессу нечем заняться, пока не придет сигнал. Рассмотрим, например, обучающую программу, проверяющую скорость чтения и понимание текста. Она отображает на экране некоторый текст, после чего обращается к системному вызову alarm, чтобы система послала ей сигнал через 30 с. Пока студент читает текст, у программы нет дел. Она может сидеть в коротком цикле, ничего не делая, но такая реализация будет напрасно расходовать время CPU, которое может понадобиться фоновому процессу или другому пользователю. Лучшее решение заключается в использовании системного вызова pause, велящего ОС UNIX приостановить работу процесса, пока не прибудет следующий сигнал.

## Реализация процессов в UNIX

Процесс в системе UNIX подобен айсбергу: то, что вы видите, представляет собой всего лишь выступающую над водой его часть, но не менее важная часть скрыта под водой. У каждого процесса есть пользовательская часть, в которой работает программа пользователя. Однако когда один из потоков обращается к системному вызову,

происходит эмулированное прерывание с переключением в режим ядра. После этого поток начинает работу в контексте ядра, с отличной картой памяти и полным доступом к ресурсам машины. Это все еще тот же самый поток, но теперь обладающий большей мощью, а также со своим стеком ядра и счетчиком команд в режиме ядра. Это важно, так как системный вызов может блокироваться на полпути, например, ожидая завершения дисковой операции. При этом счетчик команд и регистры будут сохранены таким образом, чтобы позднее поток можно было восстановить в режиме ядра.

Ядро поддерживает две ключевые структуры данных, относящиеся к процессам: **таблицу процессов и структуру пользователя**. **Таблица процессов является резидентной**. В ней содержится информация, необходимая для всех процессов, даже для тех процессов, которых в данный момент нет в памяти. Структура пользователя выгружается на диск, освобождая место в памяти, когда относящийся к ней процесс отсутствует в памяти, чтобы не тратить память на ненужную в данный момент информацию.

Информация в таблице процессов подразделяется на следующие категории:

1. **Параметры планирования**. Приоритеты процессов, процессорное время, потребленное за последний учитываемый период, количество времени, проведенное процессом в режиме ожидания. Вся эта информация используется для выбора процесса, которому будет передано управление следующим.

2. **Образ памяти**. Указатели на сегменты программы, данных и стека, или, если используется страничная организация памяти, указатели на соответствующие им таблицы страниц. Если программный сегмент используется совместно, то программный указатель указывает на общую таблицу программы. Когда процесса нет в памяти, то здесь также содержится информация о том, как найти части процесса на диске.

3. **Сигналы**. Маски, указывающие, какие сигналы игнорируются, какие перехватываются, какие временно заблокированы, а какие находятся в процессе доставки.

4. **Разное**. Текущее состояние процесса, события, ожидаемые процессом (если таковые есть), время до истечения интервала будильника, PID процесса, PID родительского процесса, идентификаторы пользователя и группы.

**В структуре пользователя содержится информация, которая не требуется, когда процесса физически нет в памяти и он не выполняется.** Например, хотя процессу, выгруженному на диск, можно послать сигнал, выгруженный процесс не может прочитать файл. По этой причине информация о сигналах должна храниться в таблице процессов, постоянно находящейся в памяти, даже когда процесс не присутствует в памяти. С другой стороны, сведения об описателях файлов могут храниться в структуре пользователя и загружаться в память вместе с процессом.

Данные, хранящиеся в структуре пользователя, включают в себя следующие пункты:

1. **Машинные регистры**. Когда происходит прерывание с переключением в режим ядра, машинные регистры (включая регистры с плавающей точкой) сохраняются здесь.

2. **Состояние системного вызова**. Информация о текущем системном вызове, включая параметры и результаты.

3. **Таблица дескрипторов файлов**. Когда происходит обращение к системному вызову, работающему с файлом, дескриптор файла используется в качестве индекса в данной таблице, что позволяет найти структуру данных (i-узел), соответствующую данному файлу.

4. **Учетная информация**. Указатель на таблицу, учитывающую процессорное время, использованное процессом в пользовательском и системном режимах. В некоторых системах здесь также ограничивается процессорное время, которое может использовать процесс, максимальный размер стека, количество страниц памяти и т. д.

5. **Стек ядра**. Фиксированный стек для использования процессом в режиме ядра.

Теперь, зная, что хранится в данных таблицах, легко объяснить, как в системе UNIX создаются процессы. Когда выполняется системный вызов `fork`, вызывающий процесс обращается в ядро и ищет свободную ячейку в таблице процессов, в которую можно записать данные о дочернем процессе. Если свободная ячейка находится, системный вызов копирует туда информацию из ячейки родительского процесса. Затем он выделяет память для сегментов данных и для стека дочернего процесса, куда копируются соответствующие сегменты родительского процесса. Структура

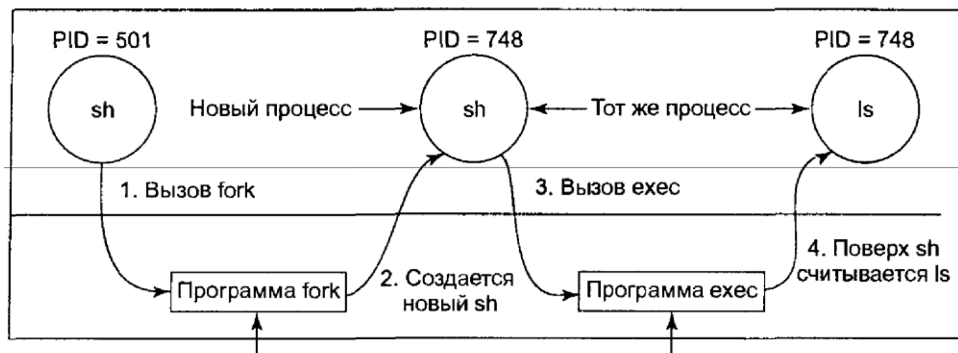
пользователя (которая часто хранится вместе с сегментом стека) копируется вместе со стеком. Программный сегмент может либо копироваться, либо использоваться совместно, если он доступен только для чтения. Начиная с этого момента дочерний процесс может быть запущен.

Когда пользователь вводит с терминала команду, например `Is`, оболочка создает новый процесс, клонируя свой собственный процесс с помощью системного вызова `fork`. Новый процесс оболочки затем вызывает системный вызов `exec`, чтобы считать в свою область памяти содержимое исполняемого файла `Is`. Эти действия показаны на рис. 5.

Механизм создания нового процесса довольно прост. Для дочернего процесса создается новая ячейка в таблице процессов, которая заполняется по большей мере из соответствующей ячейки родительского процесса. Дочерний процесс получает PID, затем настраивается его карта памяти. Кроме того, дочернему процессу предоставляется совместный доступ к файлам родительского процесса. Затем настраиваются регистры дочернего процесса, после чего он готов к запуску.

В принципе, следует создать полную копию адресного пространства, так как семантика системного вызова `fork` говорит, что никакая область памяти не используется совместно родительским и дочерним процессами. Однако копирование памяти является дорогим удовольствием, поэтому все системы UNIX слегка жульничают. Они выделяют дочернему процессу новые таблицы страниц, но эти таблицы указывают на страницы родительского процесса, помеченные как доступные только для чтения. Когда дочерний процесс пытается писать в такую страницу, происходит прерывание. При этом ядро выделяет дочернему процессу новую копию этой страницы, к которой этот процесс получает также и доступ записи. Таким образом, копируются только те страницы, в которые дочерний процесс пишет новые данные. Такой механизм называется **копированием при записи**. При этом сохраняется память, так как страницы с программой не копируются.

После того как дочерний процесс начинает работу, его программа (копия оболочки) выполняет системный вызов exec, задавая имя команды в качестве параметра. При этом ядро находит и проверяет исполняемый файл, копирует в ядро аргументы и строки окружения, а также освобождает старое адресное пространство и его таблицы страниц.




| | |
|---|---|
| 1. Выделить память для элемента таблицы дочернего процесса<br>2. Заполнить элемент таблицы дочернего процесса из элемента родительского процесса<br>3. Выделить память для стека и области пользователя дочернего процесса<br>4. Заполнить область пользователя дочернего процесса из соответствующей области<br>5. Выделить PID для дочернего процесса<br>6. Настроить дочерний процесс на использование программы родительского процесса<br>7. Копировать таблицы страниц для данных и стека<br>8. Настроить совместное использование открытых файлов | 1. Найти исполняемый файл<br>2. Проверить разрешение на выполнение<br>3. Прочитать и проверить заголовок<br>4. Копировать аргументы, среду в ядро<br>5. Освободить новое адресное пространство<br>6. Копировать аргументы, среду в стек<br>7. Сбросить сигналы<br>8. Инициализировать регистры |

| 9. Копировать регистры родительского процесса в дочерний | |
|---|---|

Рис. 5. Этапы выполнения команды ls, введенной в оболочке

После этого следует создать и заполнить новое адресное пространство. Если системой поддерживается отображение файлов на адресное пространство памяти, как, например, в System V, BSD и в большинстве других версий UNIX, то таблицы страниц настраиваются следующим образом: в них указывается, что страниц в памяти нет, кроме, возможно, одной страницы со стеком, а содержимое адресного пространства может подгружаться из исполняемого файла на диске. Когда новый процесс начинает работу, он немедленно вызывает страничное прерывание, в результате которого первая страница программы подгружается с диска. Таким образом, ничего не нужно загружать заранее, что позволяет быстро запускать программы, а в память загружать только те страницы, которые действительно нужны программам. Наконец, в стек копируются аргументы и строки окружения, сигналы сбрасываются, а все регистры устанавливаются на ноль. С этого момента новая команда начинает исполнение.

## Потоки

### Потоки в UNIX

Реализация потоков зависит от того, поддерживаются они ядром или нет. Если потоки ядром не поддерживаются, как, например, в 4BSD, реализация потоков целиком осуществляется в библиотеке, загружающейся в пространстве пользователя. Если ядро поддерживает потоки, как в системах System V и Solaris, то у ядра есть чем заняться.

Основная проблема реализации потоков заключается в поддержке корректной традиционной семантики UNIX. Рассмотрим сначала системный вызов `fork`. Предположим, что процесс с несколькими потоками (реализуемыми в ядре) выполняет системный вызов `fork`. **Следует ли при этом в новом процессе создать все потоки оригинального процесса**?

Предположим, что мы ответили на этот вопрос утвердительно – **создаются все потоки**. Допустим также, что один из потоков был блокирован, ожидая ввода с клавиатуры. Должна ли в этом случае копия этого потока также быть блокирована ожиданием ввода с клавиатуры? Если да, то какому потоку достанется следующая, набранная на клавиатуре строка? Если нет, то что должен делать этот поток в новом процессе? Проблема касается и других аспектов. В однопоточном процессе такой проблемы не возникает, так как единственный поток не может быть блокирован при обращении к системному вызову `fork`.

**Теперь рассмотрим случай, при котором в дочернем процессе другие потоки не создаются.** Предположим, что один из некопируемых потоков удерживает мьютекс, который пытается получить единственный созданный новый поток после выполнения системного вызова `fork`. В этом случае мьютекс никогда не будет освобожден, и новый поток повиснет навсегда. Существует еще множество подобных проблем. И простого решения у этих проблем нет.

**Файловый ввод-вывод представляет собой еще одну проблемную область.** Предположим, что один поток блокирован при чтении из файла, а другой поток закрывает файл и обращается к системному вызову `lseek`, чтобы изменить текущий указатель файла. Что произойдет в результате этих действий?

**Обработка сигналов тоже представляет собой сложный вопрос.** Должны ли сигналы направляться определенному потоку или всему процессу в целом? Вероятно, сигнал SIGFPE (Floating-Point Exception SIGnal — сигнал исключения при выполнении операции с плавающей точкой) должен перехватываться тем потоком, который его вызвал. Что следует делать, если он его не перехватывает? Следует ли убить этот поток? Следует ли убить все потоки процесса? Рассмотрим теперь сигнал SIGINT, посылаемый пользователем, когда он нажимает на определенную клавишу. Какой поток должен перехватывать этот сигнал? Должны ли у всех потоков быть общие маски сигналов? Любые попытки вытянуть нос в одном месте приводят к тому, что в каком-либо другом месте увязает хвост. Корректная реализация семантики потоков (не говоря уже о программе) представляет собой нетривиальную задачу.

## Потоки в системе Linux

ОС Linux поддерживает потоки в ядре довольно интересным способом, с которым следует познакомиться, В основе реализации системы Linux лежат идеи из системы 4.4BSD, но в 4.4BSD потоки на уровне ядра реализованы не были, так как у университета Калифорнии в Беркли кончились деньги прежде, чем библиотеки языка С могли быть переписаны так, чтобы решить все описанные выше проблемы. 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 табл. 3. 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 CPU `STORE` не виден остальным потокам. Такое поведение подобно поведению системного вызова `fork`. Создание нового адресного пространства равнозначно определению нового процесса.

Таблица 2. Биты массива `sharing_flags`

| 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` |

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 на 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. 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 (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.

## Системные вызовы управления потоками

В первой версии системы не было потоков. Это свойство было добавлено много лет спустя. Изначально применялось множество различных пакетов поддержки потоков, однако распространение этих различных пакетов привело к тому, что написать переносимую программу стало очень сложно. В конце концов, системные вызовы, используемые для управления потоков, были стандартизированы в виде части стандарта POSIX (P1003.1c).

В стандарте POSIX не указывается, должны ли потоки реализовываться в пространстве ядра или в пространстве пользователя. **Преимущество потоков в пользовательском пространстве** состоит в том, что они легко реализуются без необходимости изменения ядра, а переключение потоков осуществляется очень эффективно. **Недостаток потоков в пространстве пользователя** заключается в том, что если один из потоков заблокируется (например, на операции ввода-вывода, семафоре или страничном прерывании), все потоки процесса блокируются. Ядро полагает, что существует только один поток, и не передает управление процессу потока, пока блокировка не снимется. Таким образом, системные вызовы, определенные в стандарте P1003.1c, были тщательно отобраны так, чтобы потоки могли быть реализованы любым способом. До тех пор пока пользовательские программы четко придерживаются семантики стандарта P1003.1c, оба способа реализации должны работать корректно. Наиболее часто применяемые вызовы управления потоками перечислены в табл. 2. Когда используется системная реализация потоков, они являются настоящими системными вызовами. При использовании потоков на уровне пользователя они полностью реализуются в динамической библиотеке в пространстве пользователя.

Таблица 3. Основные вызовы управления потоками стандарта POSIX

| Вызов | Описание |
|---|---|
| pthread_create | Создать новый поток в адресном пространстве вызывающего процесса |
| pthread_exit | Завершить вызывающий процесс |
| pthread_join | Подождать, пока не завершится процесс |
| pthread_mutex_init | Создать новый мьютекс |

| | |
|---|---|
| pthread_mutex_destroy | Уничтожить мьютекс |
| pthread_mutex_lock | Заблокировать мьютекс |
| pthread_mutex_unlock | Разблокировать мьютекс |
| pthread_cond_init | Создать условную переменную |
| pthread_cond_destroy | Уничтожить условную переменную |
| pthread_cond_wait | Ждать условную переменную |
| pthread_cond_signal | Разблокировать один поток, ждущий условную переменную |

Поток, выполнивший свою работу и желающий прекратить свое существование, обращается к системному вызову pthread_exit. Поток может подождать, пока не завершится процесс, обратившись к системному вызову pthread_join. Если ожидаемый поток уже завершил свою работу, системный вызов pthread_join выполняется мгновенно. В противном случае обратившийся к нему поток блокируется.

Синхронизация потоков может осуществляться при помощи **мьютексов**. Как правило, мьютекс охраняет какой-либо ресурс, например буфер, совместно используемый двумя потоками. Чтобы гарантировать, что только один поток в каждый момент времени имеет доступ к общему ресурсу, предполагается, что потоки блокируют (захватывают) мьютекс перед обращением к ресурсу и разблокируют (отпускают) его, когда ресурс им более не нужен. До тех пор пока потоки соблюдают данный протокол, состояния состязания можно избежать. Мьютексы подобны двоичным семафорам, то есть семафорам, способным принимать только значения 0 и 1, Название **мьютекс** (mutex) образовано от английских слов mutual exclusion — взаимное исключение.

Мьютексы могут создаваться вызовом pthread_mutex_init и уничтожаться при помощи вызова pthread_mutex_destroy. Мьютекс может находиться в одном из двух состояний: блокированный и разблокированный. Поток может заблокировать мьютекс с помощью вызова pthread_mutex_lock. Если мьютекс уже заблокирован, то поток, обратившийся к этому вызову, блокируется. Когда поток, захвативший мьютекс, выполнил свою работу в критической области, он должен освободить мьютекс, обратившись к вызову pthread_mutex_unlock.

Мьютексы предназначены для кратковременной блокировки, например для защиты совместно используемой переменной. Они не предназначаются для долговременной синхронизации, например для ожидания, когда освободится накопитель на магнитной ленте. **Для долговременной синхронизации предоставляются переменные состояния.** Эти переменные создаются с помощью вызова pthread_condinit и уничтожаются вызовом pthread_cond_destroy.

Переменные состояния используются следующим образом: один поток ждет, когда переменная примет определенное значение, а другой поток сигнализирует ему изменением этой переменной. Например, обнаружив, что нужный ему накопитель на магнитной ленте занят, поток может обратиться к вызову pthread_condwait, задав в качестве параметра адрес переменной, которую все потоки согласились связать с накопителем на магнитной ленте. Когда поток, использующий накопитель на магнитной ленте, наконец, освободит это устройство (возможно, через несколько часов), он обращается к вызову pthread_cond_signal, чтобы изменить переменную состояния и тем самым сообщить ожидающим потокам, что магнитофон свободен. Если ни один поток в этот момент не ждет, когда освободится накопитель на магнитной ленте, этот сигнал просто теряется. **Другими словами, переменные состояния не считаются семафорами.** С потоками, мьютексами и переменными состояния также определены несколько других операций.

## Планирование в системе UNIX

Давайте теперь изучим алгоритм планирования системы UNIX. Поскольку UNIX всегда была многозадачной системой, ее алгоритм планирования с самого начала развития системы разрабатывался так, чтобы обеспечить хорошую реакцию в интерактивных процессах. У этого алгоритма два уровня. Низкоуровневый алгоритм выбирает следующий процесс из набора процессов в памяти и готовых к работе.

Высокоуровневый алгоритм перемещает процессы из памяти на диск и обратно, что предоставляет всем процессам возможность попасть в память и быть запущенными. У каждой версии UNIX свой слегка отличающийся низкоуровневый алгоритм планирования, но у большинства этих алгоритмов есть много общих черт, которые мы здесь и опишем. В низкоуровневом алгоритме используется несколько очередей. С каждой очередью связан диапазон непересекающихся значений приоритетов. Процессы, выполняющиеся в режиме пользователя (верхняя часть айсберга), имеют положительные значения приоритетов. У процессов, выполняющихся в режиме ядра (обращающихся к системным вызовам), значения приоритетов отрицательные. Отрицательные значения приоритетов считаются наивысшими, а положительные — наоборот, минимальными, как показано на рис. 6. В очередях располагаются только процессы, находящиеся в памяти и готовые к работе.

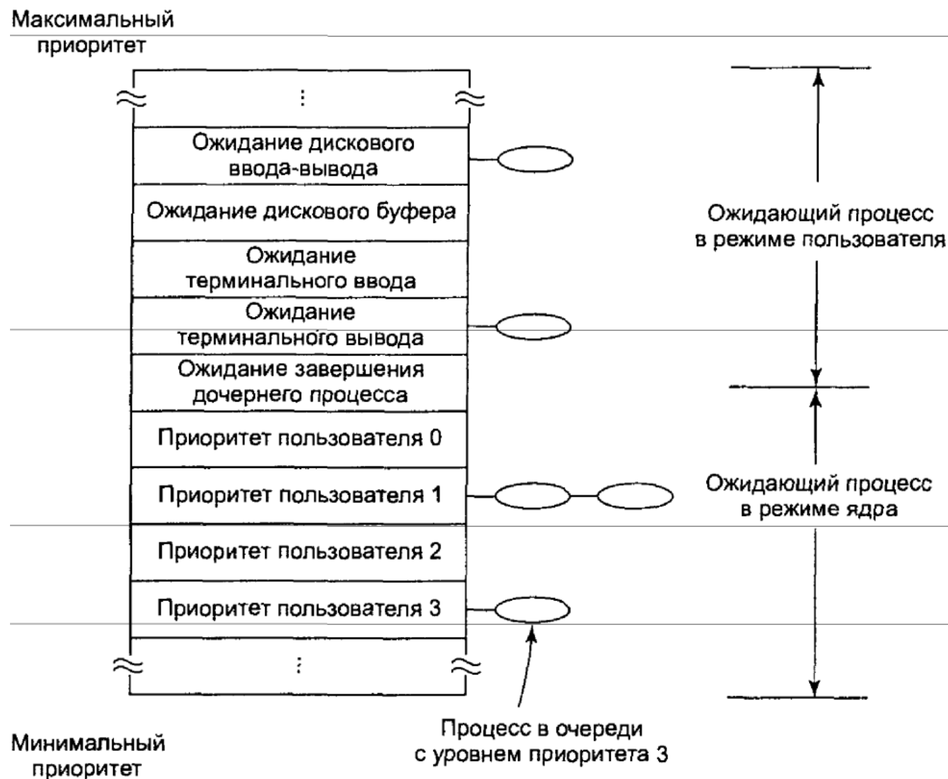



Рис. 6. Планировщик ОС UNIX основан на структуре многоуровневой очереди

Когда запускается (низкоуровневый) планировщик, он ищет очередь, начиная с самого высокого приоритета (то есть с наименьшего отрицательного значения), пока не находит очередь, в которой есть хотя бы один процесс. После этого в этой очереди выбирается и запускает первый процесс. Ему разрешается работать в течение некоего максимального кванта времени, как правило, 100 мс, или пока он не заблокируется. Если процесс использует весь свой квант времени, он помещается обратно, в конец очереди, а алгоритм планирования запускается снова. Таким образом, процессы, входящие в одну группу приоритетов, совместно используют CPU в порядке циклической очереди.

Раз в секунду приоритет каждого процесса пересчитывается по формуле, состоящей из трех компонентов:

```
priority = CPU_usage + nice + base.
```

На основе сосчитанного нового приоритета каждый процесс прикрепляется к соответствующей очереди на рис. 6. Для получения номера очереди приоритет, как правило, делится на некую константу. Изучим вкратце каждый из трех компонентов этой формулы приоритета.

Параметр `CPU_usage` (использование CPU) представляет собой среднее значение тиков таймера в секунду, которые процесс работал в течение последних нескольких секунд. При каждом тике (прерывании) таймера счетчик использования CPU в таблице процессов увеличивается на единицу. Этот счетчик в конце концов добавляется к приоритету процесса, увеличивая тем самым числовое значение его приоритета (что соответствует более низкому приоритету), в результате чего процесс попадает в менее приоритетную очередь.

Однако ОС UNIX не наказывает процесс за использование CPU навечно, и величина `CPU_usage` со временем уменьшается. В различных версиях UNIX это уменьшение выполняется по-разному. Один из способов состоит в том, что к `CPU_usage` прибавляется полученное число тиков $\Delta T$, после чего сумма делится на два. Такой алгоритм учитывает самое последнее значение $\Delta T$ с весовым коэффициентом 1/2, предшествующее ему — с весовым коэффициентом 1/4 и т. д. Алгоритм взвешивания очень быстр, так как состоит из всего одной операции сложения и одного сдвига, но также применяются и другие схемы взвешивания.

С каждым процессом связано значение `nice`. Его значение по умолчанию равно 0, но допустимый диапазон значений, как правило, составляет от —20 до +20. Процесс может установить значение `nice` в диапазоне от 0 до 20 с помощью системного вызова `nice`. Пользователь, вычисляющий в фоновом режиме число $\pi$ с миллиардом знаков после точки, может обратиться к этому системному вызову, чтобы быть вежливым по отношению к другим пользователям. Только системный администратор может запросить обслуживание с более высоким приоритетом (то есть значения `nice` от -20 до -1).

Когда процесс эмулирует прерывание для выполнения системного вызова в ядре, процесс, вероятно, должен быть блокирован, пока системный вызов не будет выполнен и не вернется в режим пользователя. Например, процесс может обратиться к системному вызову `waitpid`, ожидая, пока один из его дочерних процессов не закончит работу. Он может также ожидать ввода с терминала или завершения дисковой операции ввода-вывода и т. д. Когда процесс блокируется, он удаляется из структуры очереди, пока этот процесс снова не будет готов работать.

Однако когда происходит событие, которого ждал процесс, он снова помещается в очередь с отрицательным значением. Выбор очереди определяется событием, которого ждал процесс. На рис. 6 дисковый ввод-вывод показан как событие с наивысшим приоритетом, так что процесс, только что прочитавший или записавший блок диска, вероятно, получит центральный процессор в течение 100 мс. Отрицательные значения приоритета для дискового ввода-вывода, терминального ввода-вывода и т. д. жестко прошиты в операционной системе и могут быть изменены только путем перекомпиляции самой системы. Эти (отрицательные) значения представлены в приведенной выше формуле слагаемым `base` (база), и их величина достаточно отличается от нуля, чтобы перезапущенный процесс наверняка попадал в другую очередь.

**В основе этой схемы лежит идея как можно более быстрого удаления процессов из ядра**. Если процесс пытается читать дисковый файл, необходимость ждать целую секунду между обращениями к системным вызовам `read` замедлит его работу во много раз. Значительно лучше позволить ему немедленно продолжить работу сразу после выполнения запроса, так чтобы он мог быстро обратиться к следующему системному вызову. Если процесс был заблокирован ожиданием ввода с терминала, то, очевидно, это интерактивный процесс, и ему должен быть предоставлен наивысший приоритет, как только он перейдет в состояние готовности, чтобы гарантировать хорошее качество обслуживания интерактивных процессов. Таким образом, процессы, ограниченные производительностью CPU (то есть находящиеся в положительных очередях), в основном обслуживаются после того, как будут обслужены все процессы, ограниченные вводом-выводом (когда все эти процессы окажутся заблокированы в ожидании ввода-вывода).

## Планирование в системе Linux

Планирование представляет собой одну из немногих областей, в которых ОС Linux использует алгоритм, отличный от применяющегося в UNIX. Начнем с того, что потоки в системе Linux реализованы в ядре, поэтому планирование основано на потоках, а не на процессах. В ОС Linux алгоритмом планирования различаются три класса потоков:

1. **Потоки реального времени**, обслуживаемые по алгоритму FIFO (First in First Out — первым прибыл — первым обслужен).

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 ОС Linux эти классы называются классами реального времени, в том, что ОС Linux совместима со стандартом P1003.4 (расширение «реального времени» для UNIX), в котором они носят эти имена.

**У каждого потока есть приоритет планирования.** Значение по умолчанию равно 20, 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 времени CPU 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 == time_sharing && quantum > 0) goodness = quantum + priority;

if (class == time_sharing && 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 время CPU, 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. CPU 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 CPU, 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 времени CPU, поэтому в следующий раз значение его нового кванта будет больше, чем у потока, ограниченного производительностью CPU. 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 CPU. Таким образом, потоки, ограниченные вводом-выводом, получают преимущество при планировании.

Другое свойство этого алгоритма заключается в том, что когда потоки, ограниченные производительностью CPU, соревнуются за право использования CPU, **поток с большим приоритетом получает большую долю времени CPU**. В качестве примера рассмотрим два потока, ограниченных производительностью CPU: 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% времени CPU, а поток В получает 20 % времени CPU.

## Литература

1. Э. Таненбаум. Современные операционные системы. 2-ое изд. –СПб.: Питер, 2002. –1040 с.

2. Э. Таненбаум, А. Вудхалл. Операционные системы: разработка и реализация. Классика CS. –СПб.: Питер, 2006. –576 с.

3. Робачевский А.М. Операционная система UNIX. –СПб.: БХВ-Петербург, 2002. -528 с.